

Trær

I. EKSEMPLER, DEFINISJON

II. BINÆRE TRÆR

III. ADT TRE

IV. BASIS TREALGORITMER (TRAVERSERING)

V. IMPLEMENTASJON AV BINÆRE TRÆR

Lenket Struktur

Sequence (Array)

VI. NOEN ANVENDELSER

Syntaks trær

Ordbok Søking

Kap. 6 (kursorisk: 6.4.4; + DFS/BFS)

Noen eksempler

Trær av rekursive kall :

```
int fib(int n) {  
    if (n==0 || n==1) return 1;  
    else  
        return fib(n-1) + fib(n-2);  
}
```

Noen eksempler

Trær av rekursive kall :

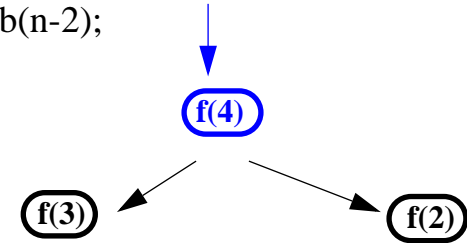
```
int fib(int n) {  
    if (n==0 || n==1) return 1;  
    else  
        return fib(n-1) + fib(n-2);  
}  
> fib(4)
```



Noen eksempler

Trær av rekursive kall :

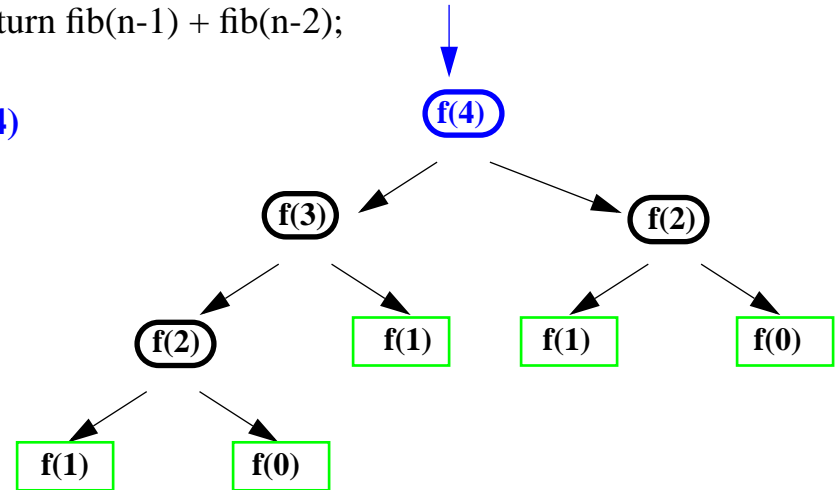
```
int fib(int n) {  
    if (n==0 || n==1) return 1;  
    else  
        return fib(n-1) + fib(n-2);  
}  
> fib(4)
```



Noen eksempler

Trær av rekursive kall :

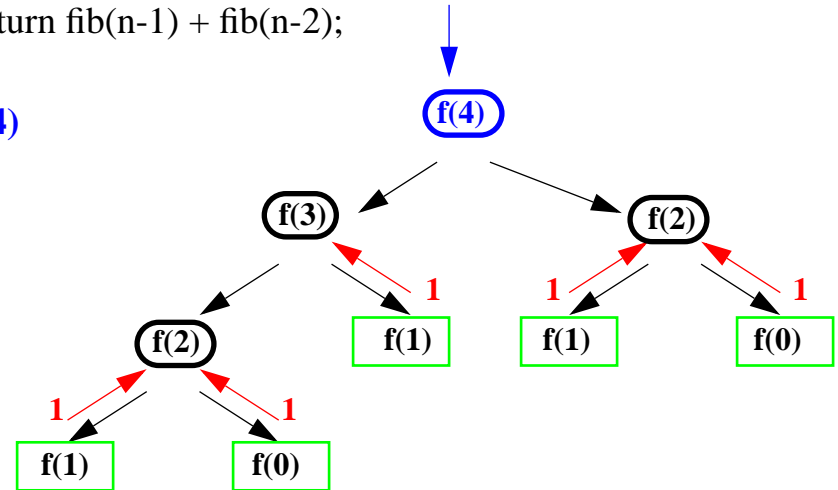
```
int fib(int n) {  
    if (n==0 || n==1) return 1;  
    else  
        return fib(n-1) + fib(n-2);  
}  
> fib(4)
```



Noen eksempler

Trær av rekursive kall :

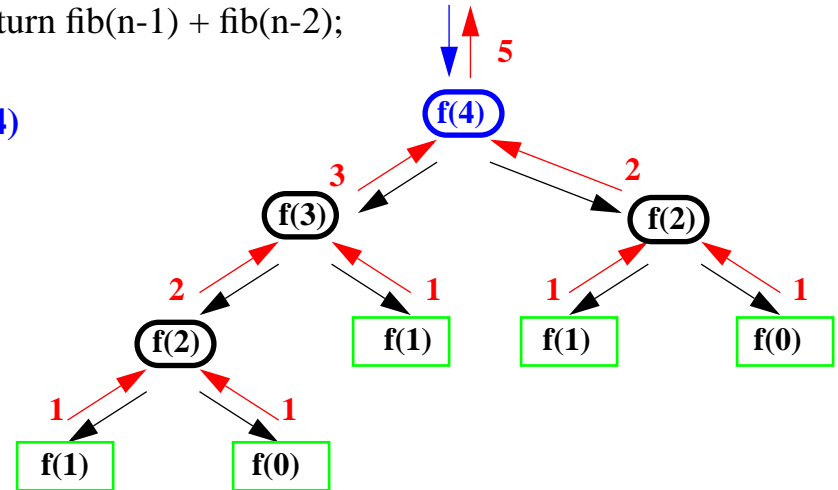
```
int fib(int n) {  
  if (n==0 || n==1) return 1;  
  else  
    return fib(n-1) + fib(n-2);  
}  
> fib(4)
```



Noen eksempler

Trær av rekursive kall :

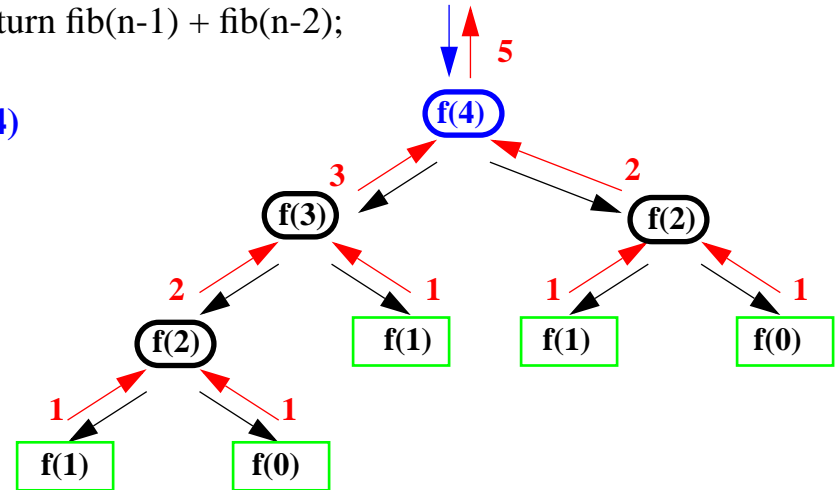
```
int fib(int n) {  
  if (n==0 || n==1) return 1;  
  else  
    return fib(n-1) + fib(n-2);  
}  
> fib(4)
```



Noen eksempler

Trær av rekursive kall :

```
int fib(int n) {  
    if (n==0 || n==1) return 1;  
    else  
        return fib(n-1) + fib(n-2);  
}  
> fib(4)
```



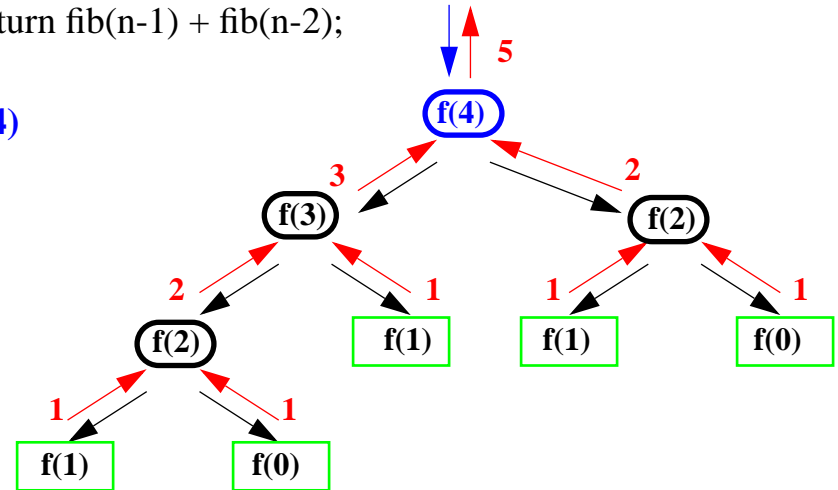
Filhierarki :

- > **cd**
- > **cd** arbeid
- > **ls**
- artikler
- memo
- > **cd ..**

Noen eksempler

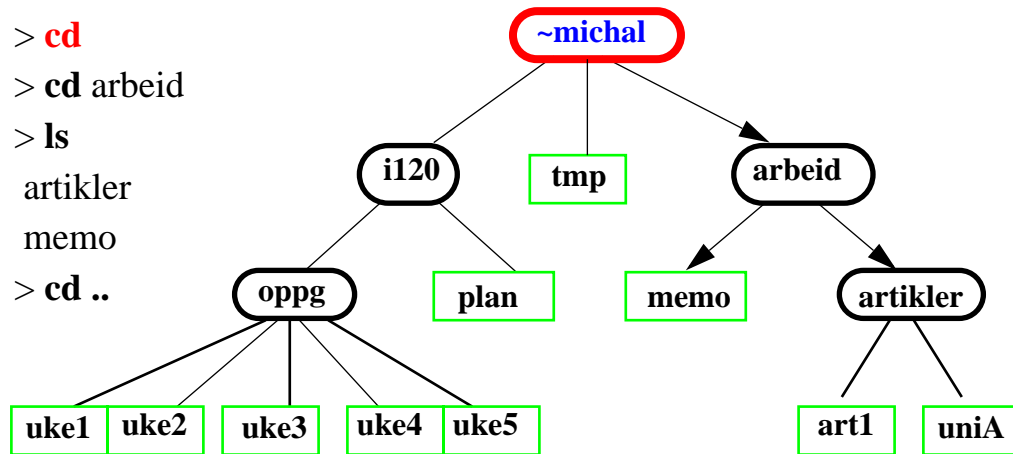
Trær av rekursive kall :

```
int fib(int n) {  
    if (n==0 || n==1) return 1;  
    else  
        return fib(n-1) + fib(n-2);  
}  
> fib(4)
```



Filhierarki :

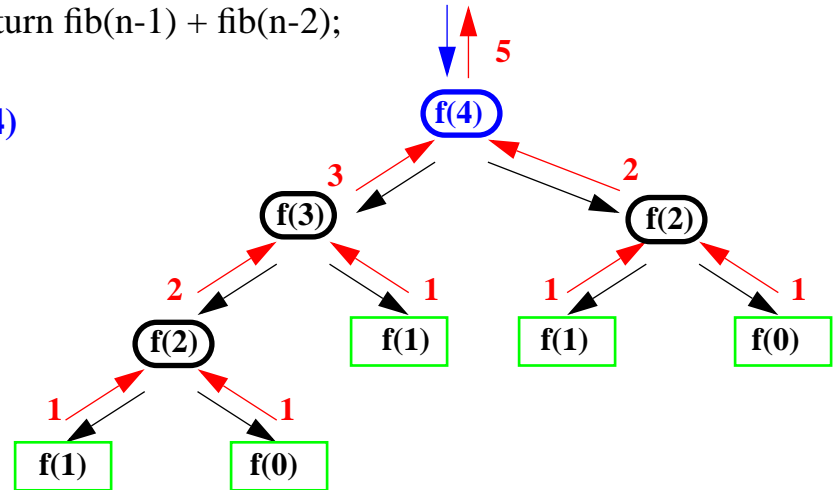
```
> cd  
> cd arbeid  
> ls  
artikler  
memo  
> cd ..
```



Noen eksempler

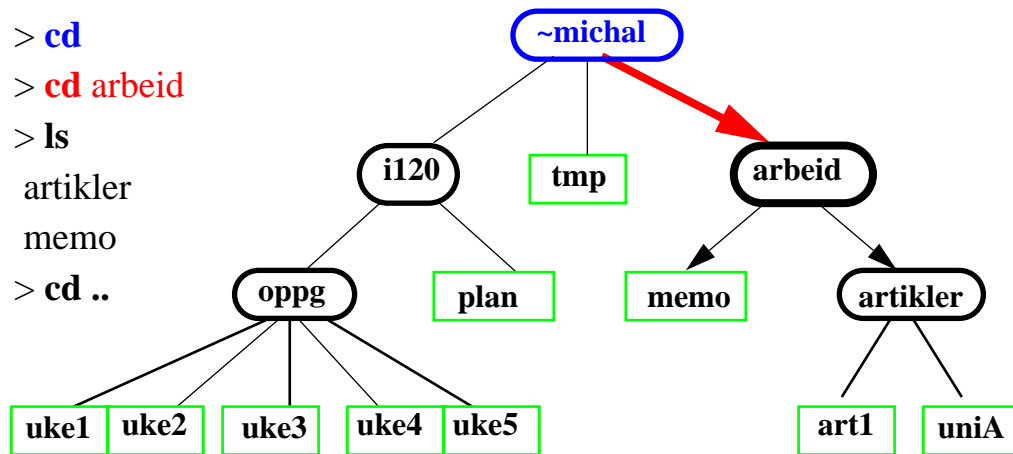
Trær av rekursive kall :

```
int fib(int n) {  
    if (n==0 || n==1) return 1;  
    else  
        return fib(n-1) + fib(n-2);  
}  
> fib(4)
```



Filhierarki :

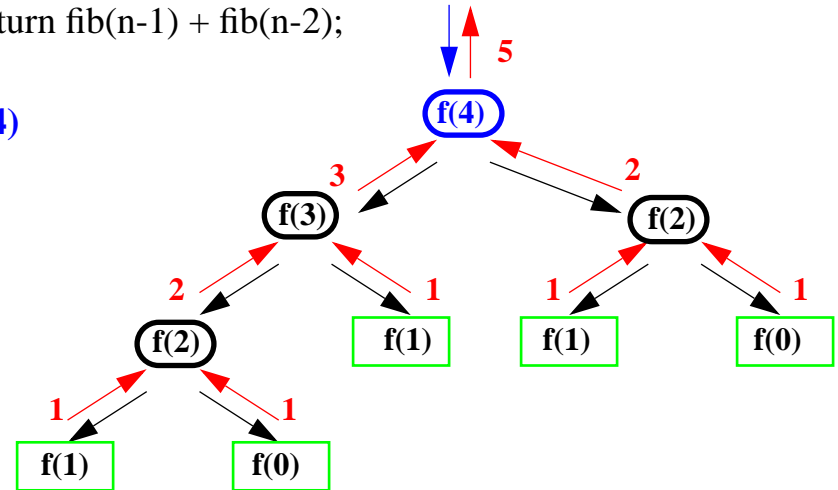
```
> cd  
> cd arbeid  
> ls  
artikler  
memo  
> cd ..
```



Noen eksempler

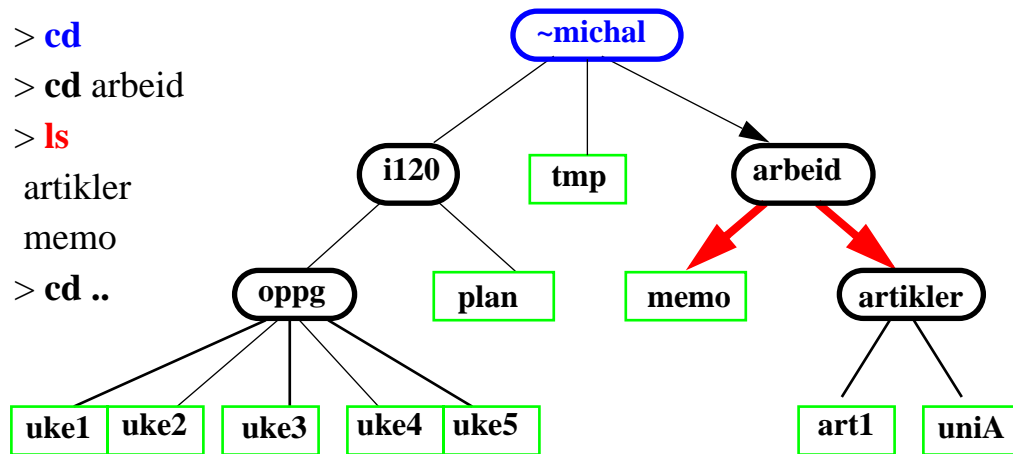
Trær av rekursive kall :

```
int fib(int n) {  
    if (n==0 || n==1) return 1;  
    else  
        return fib(n-1) + fib(n-2);  
}  
> fib(4)
```



Filhierarki :

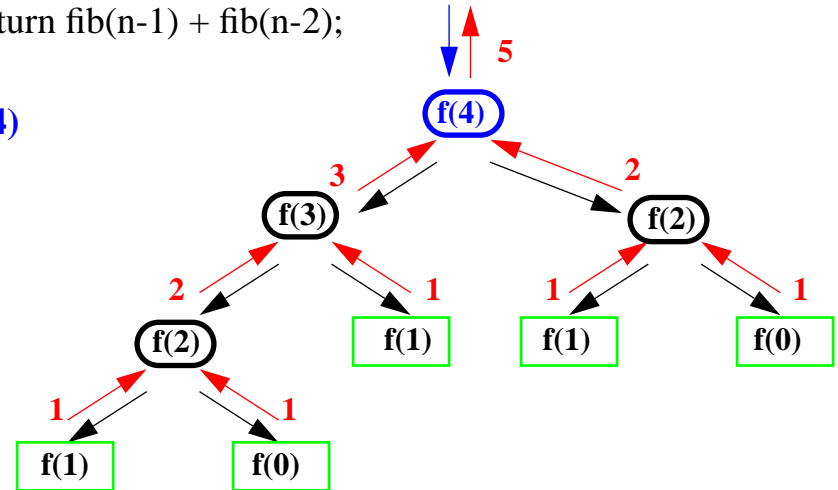
```
> cd  
> cd arbeid  
> ls  
artikler  
memo  
> cd ..
```



Noen eksempler

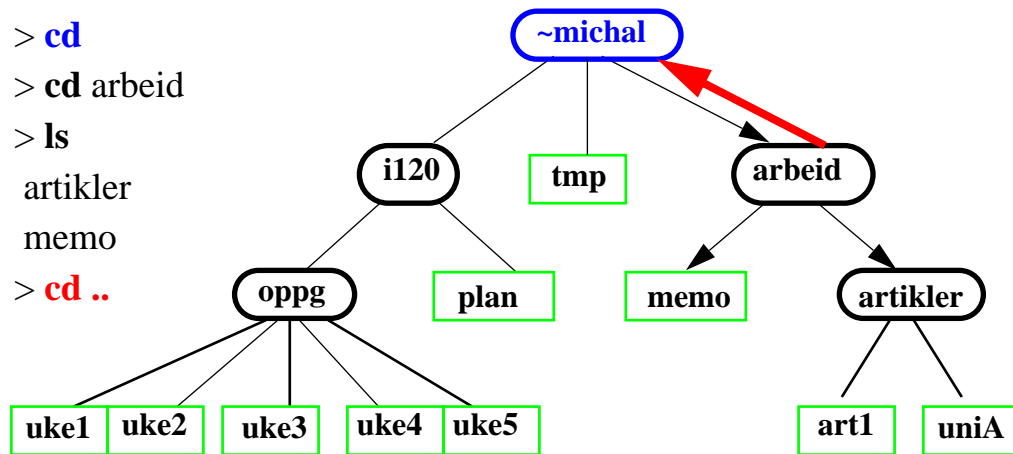
Trær av rekursive kall :

```
int fib(int n) {  
    if (n==0 || n==1) return 1;  
    else  
        return fib(n-1) + fib(n-2);  
}  
> fib(4)
```



Filhierarki :

```
> cd  
> cd arbeid  
> ls  
artikler  
memo  
> cd ..
```



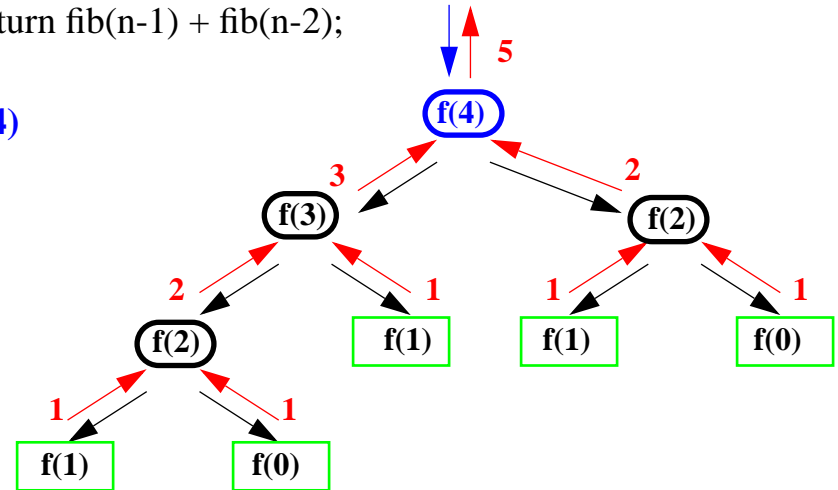
Noen eksempler

Syntakstrær :

$((3+1)*5)*((8-4)/2)$

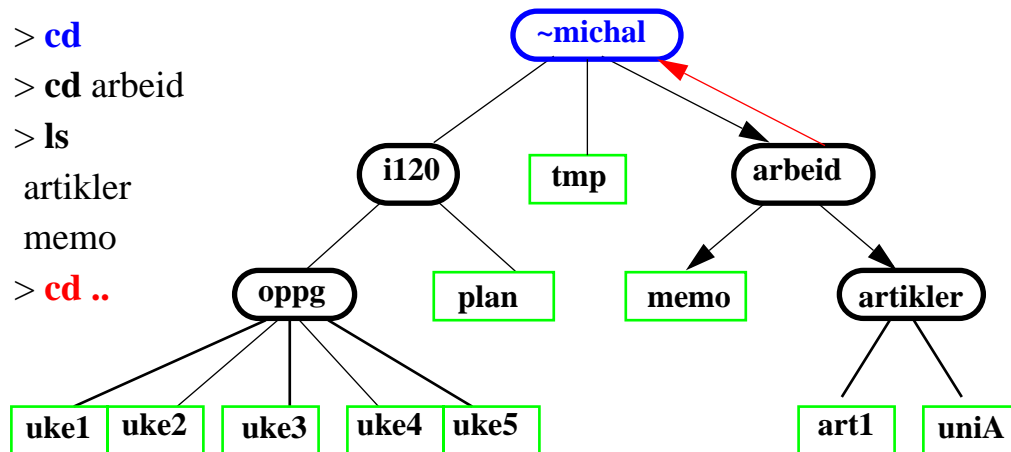
Trær av rekursive kall :

```
int fib(int n) {  
    if (n==0 || n==1) return 1;  
    else  
        return fib(n-1) + fib(n-2);  
}  
> fib(4)
```



Filhierarki :

```
> cd  
> cd arbeid  
> ls  
artikler  
memo  
> cd ..
```



Noen eksempler

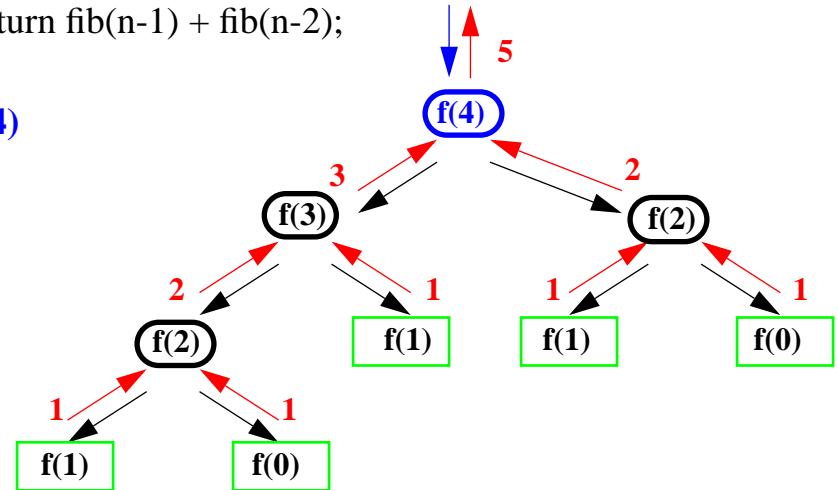
Syntakstrær :

$((3+1)*5)*((8-4)/2)$

$((3 + 1) \times 5) \times ((8 - 4) / 2)$

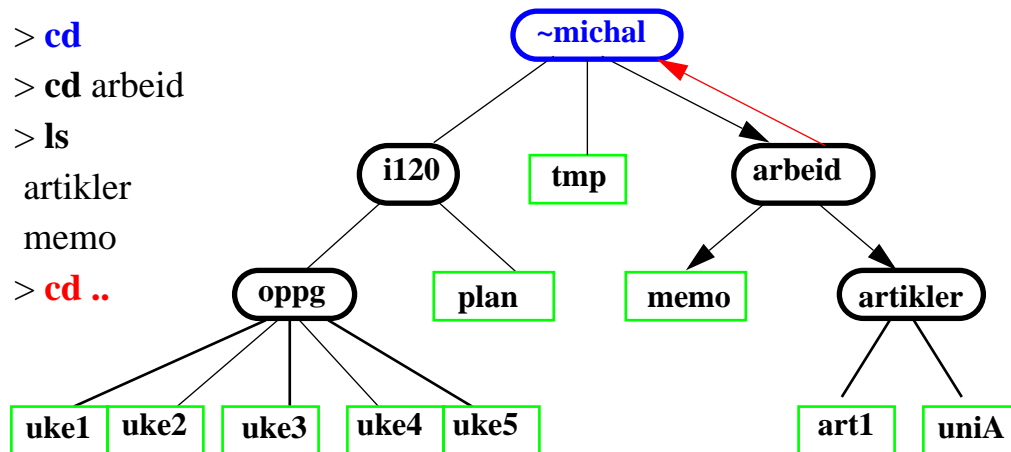
Trær av rekursive kall :

```
int fib(int n) {
    if (n==0 || n==1) return 1;
    else
        return fib(n-1) + fib(n-2);
}
> fib(4)
```



Filhierarki :

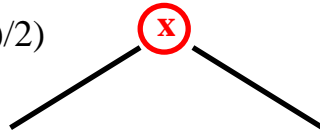
```
> cd
> cd arbeid
> ls
artikler
memo
> cd ..
```



Noen eksempler

Syntakstrær :

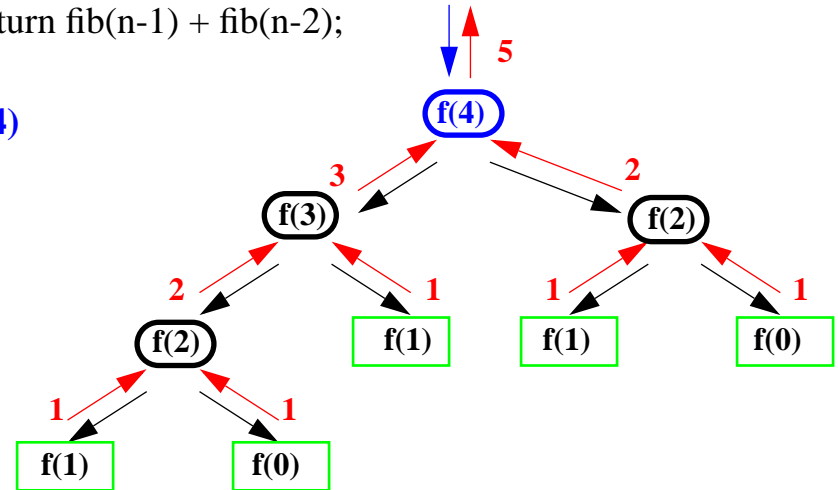
$((3+1)*5)*((8-4)/2)$



() x ()

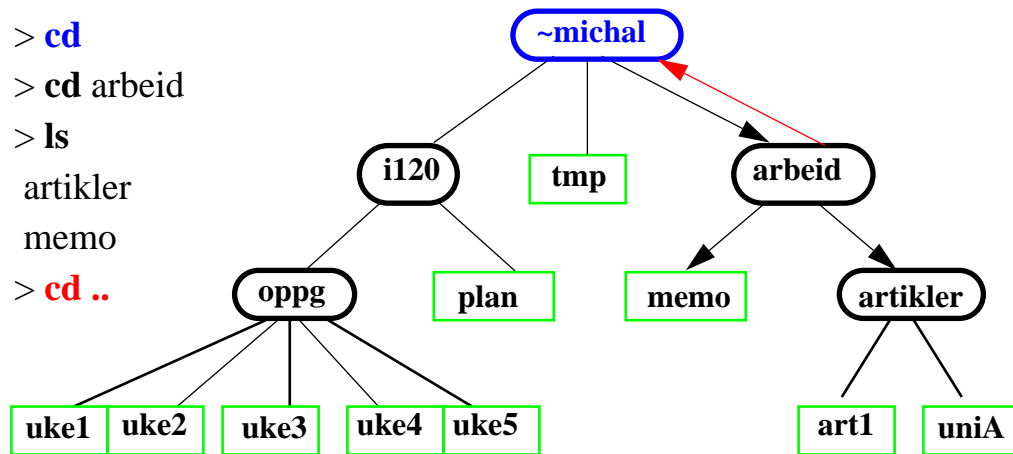
Trær av rekursive kall :

```
int fib(int n) {
  if (n==0 || n==1) return 1;
  else
    return fib(n-1) + fib(n-2);
}
> fib(4)
```



Filhierarki :

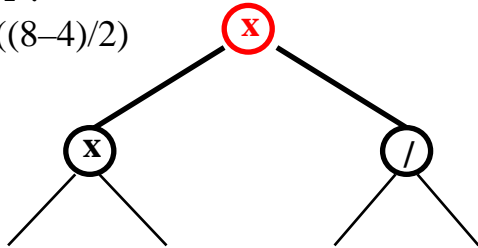
- > cd
- > cd arbeid
- > ls
- artikler
- memo
- > cd ..



Noen eksempler

Syntakstrær :

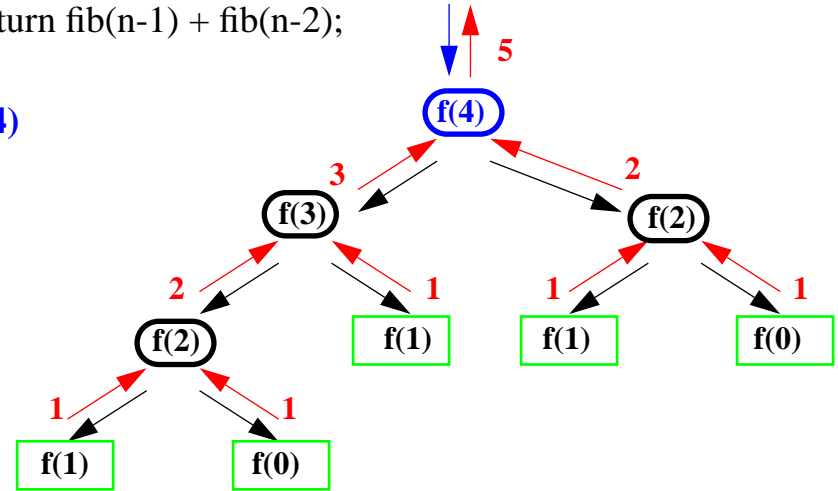
$((3+1)*5)*((8-4)/2)$



(() x) x (() /)

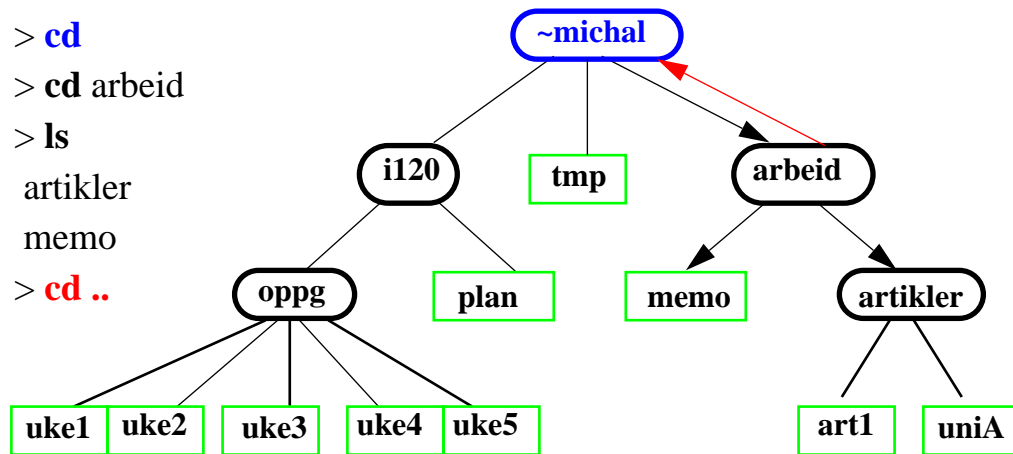
Trær av rekursive kall :

```
int fib(int n) {
  if (n==0 || n==1) return 1;
  else
    return fib(n-1) + fib(n-2);
}
> fib(4)
```



Filhierarki :

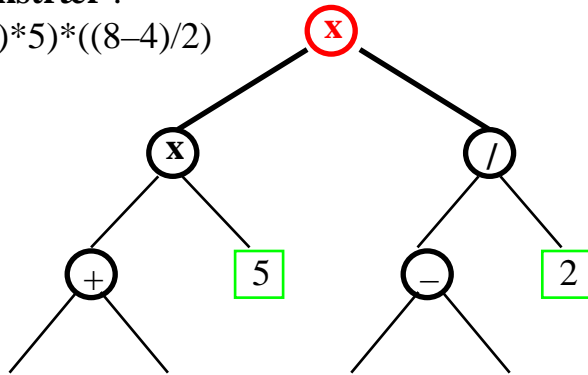
- > cd
- > cd arbeid
- > ls
- artikler
- memo
- > cd ..



Noen eksempler

Syntakstrær :

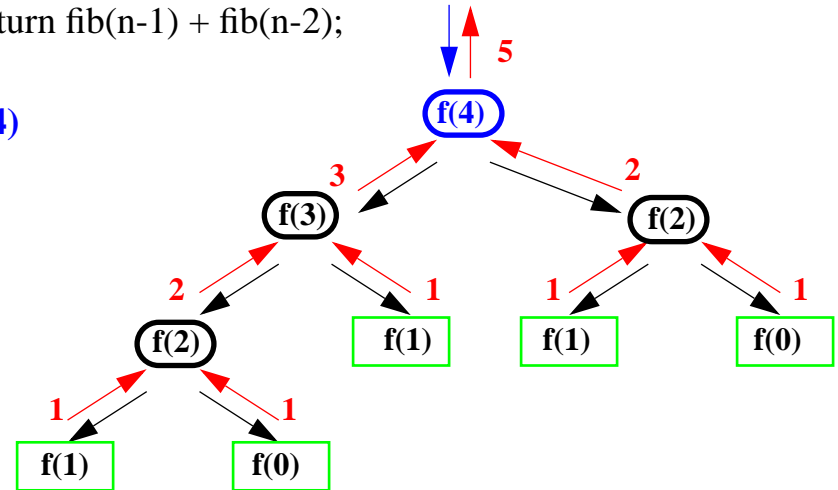
$((3+1)*5)*((8-4)/2)$



$((\quad + \quad) \times 5) \times ((\quad - \quad) / 2)$

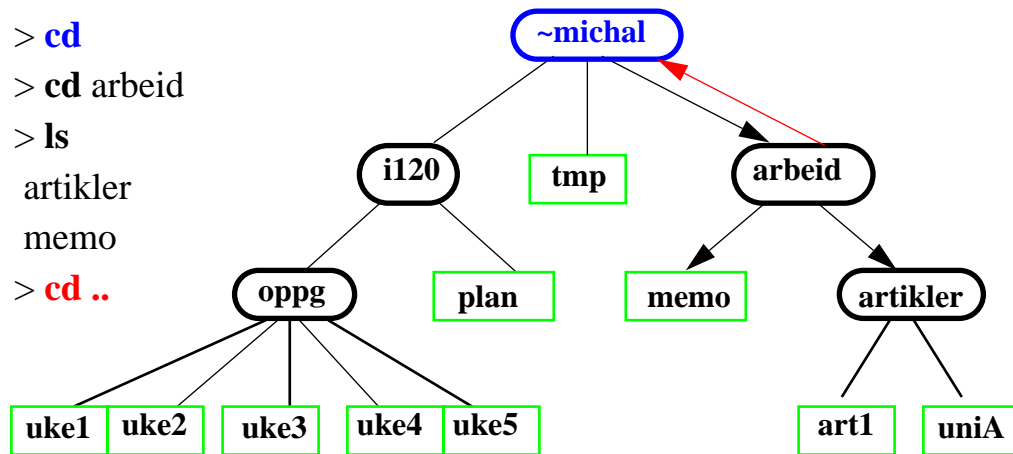
Trær av rekursive kall :

```
int fib(int n) {
  if (n==0 || n==1) return 1;
  else
    return fib(n-1) + fib(n-2);
}
> fib(4)
```



Filhierarki :

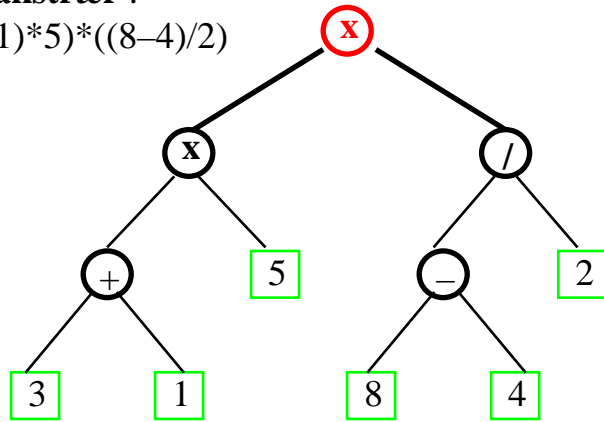
```
> cd
> cd arbeid
> ls
artikler
memo
> cd ..
```



Noen eksempler

Syntakstrær :

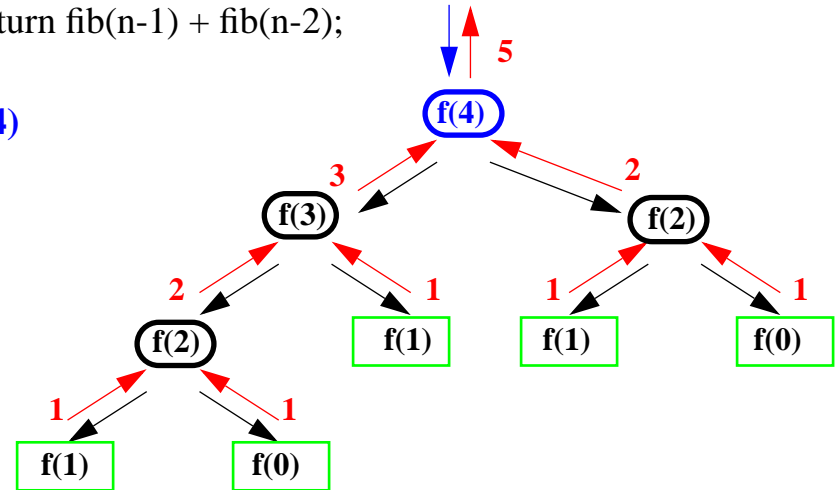
$((3+1)*5)*((8-4)/2)$



$((3 + 1) \times 5) \times ((8 - 4) / 2)$

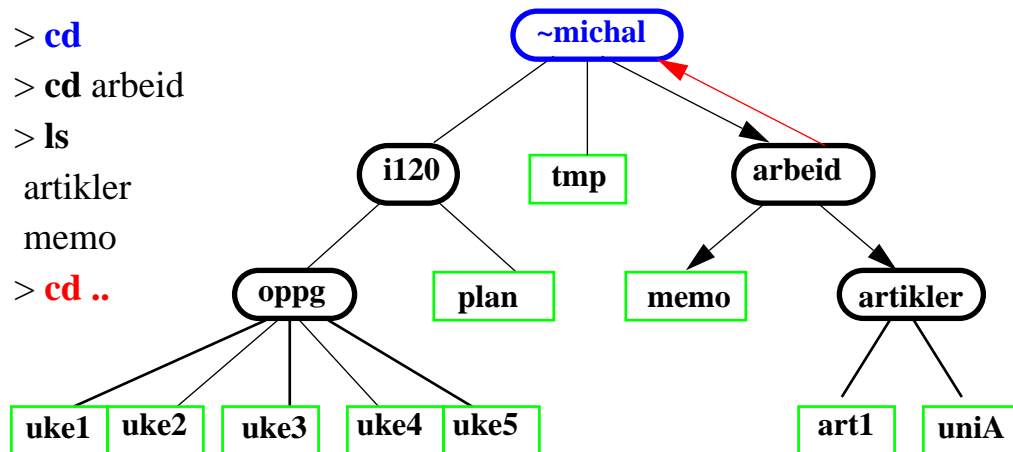
Trær av rekursive kall :

```
int fib(int n) {
  if (n==0 || n==1) return 1;
  else
    return fib(n-1) + fib(n-2);
}
> fib(4)
```



Filhierarki :

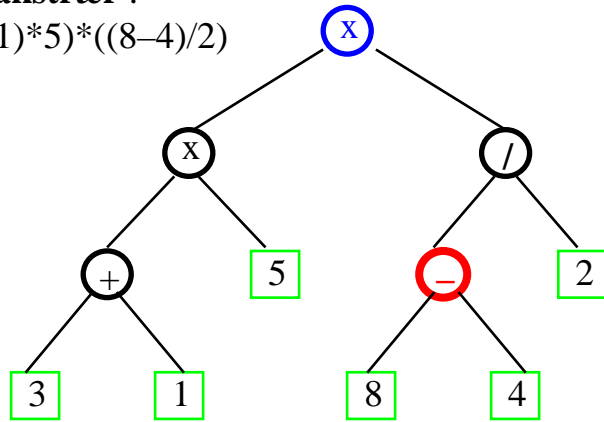
```
> cd
> cd arbeid
> ls
artikler
memo
> cd ..
```



Noen eksempler

Syntakstrær :

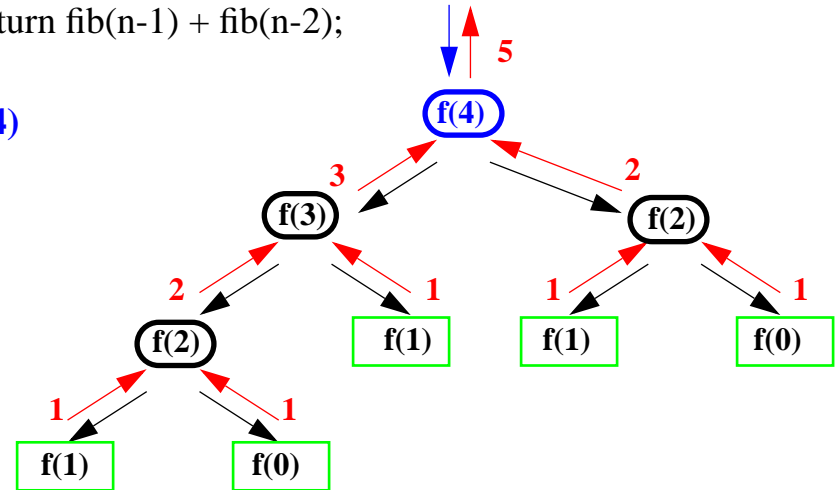
$((3+1)*5)*((8-4)/2)$



$((3 + 1) \times 5) \times ((8 - 4) / 2)$

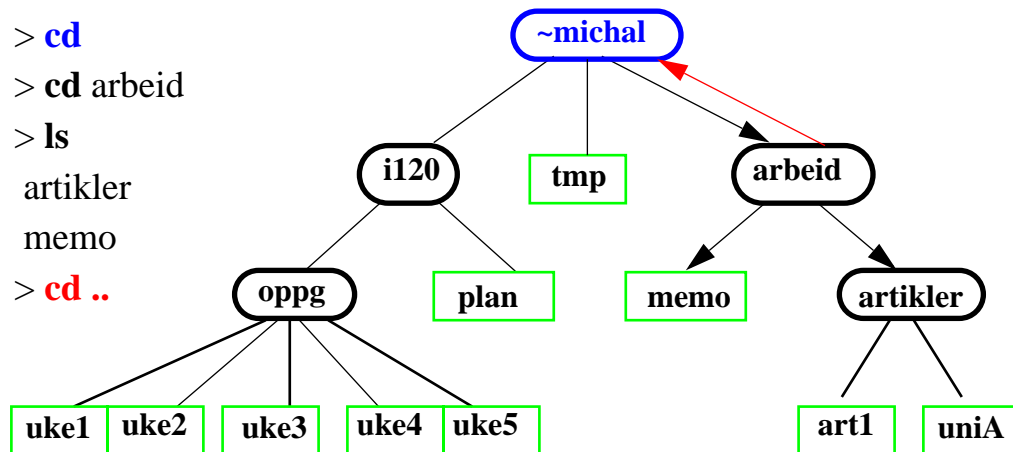
Trær av rekursive kall :

```
int fib(int n) {
  if (n==0 || n==1) return 1;
  else
    return fib(n-1) + fib(n-2);
}
> fib(4)
```



Filhierarki :

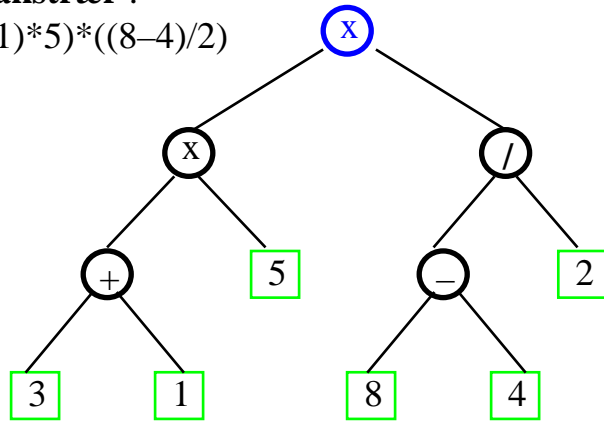
```
> cd
> cd arbeid
> ls
artikler
memo
> cd ..
```



Noen eksempler

Syntakstrær :

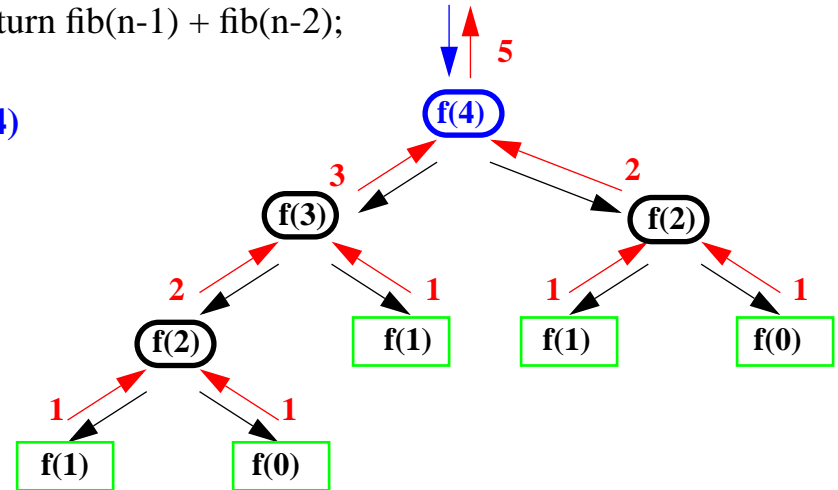
$((3+1)*5)*((8-4)/2)$



$((3 + 1) \times 5) \times ((8 - 4) / 2)$

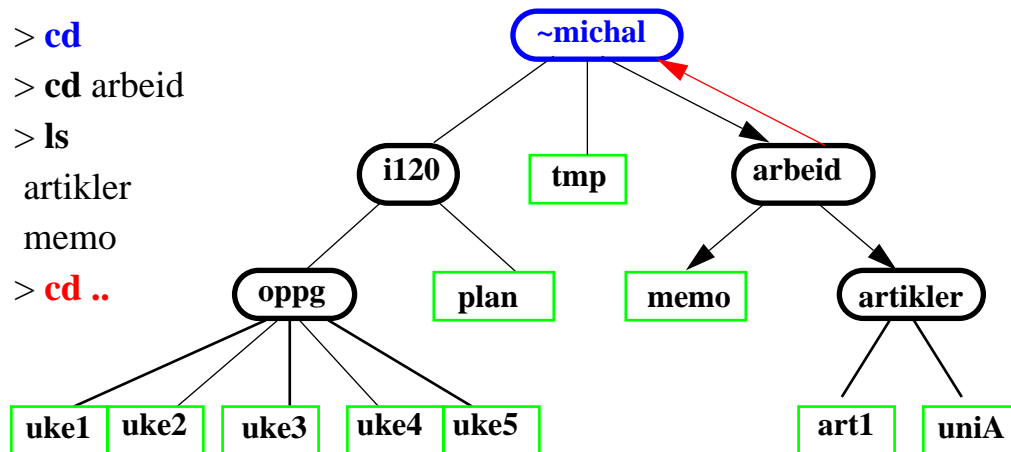
Trær av rekursive kall :

```
int fib(int n) {
  if (n==0 || n==1) return 1;
  else
    return fib(n-1) + fib(n-2);
}
> fib(4)
```



Filhierarki :

```
> cd
> cd arbeid
> ls
  artikler
  memo
> cd ..
```



et **Tre** T er enten :

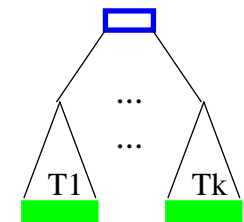
en enkelt node 

eller består av

en *node* r

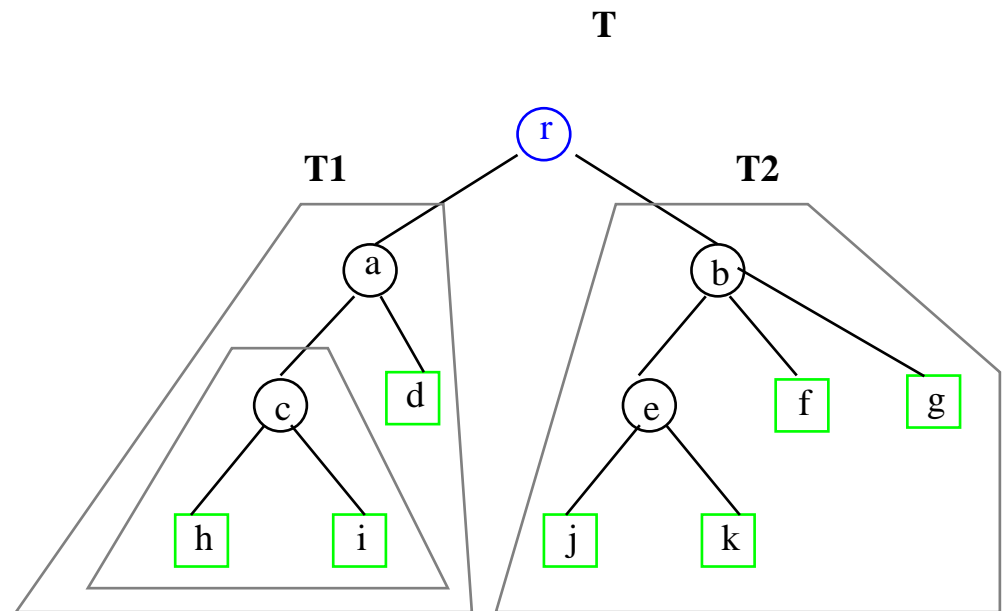
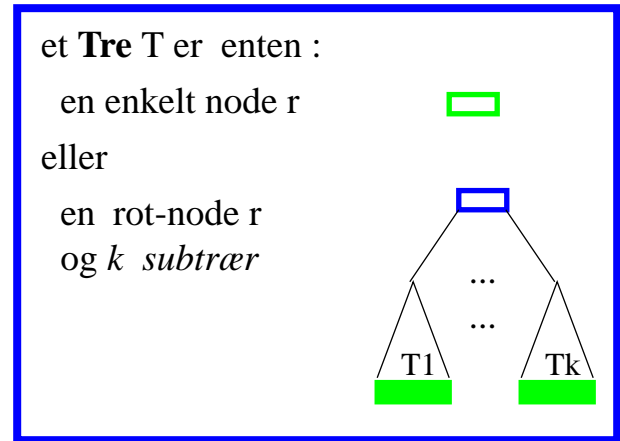
(r rot-node) og

k *subtrær*



Terminologi (fra slektskapstrær)

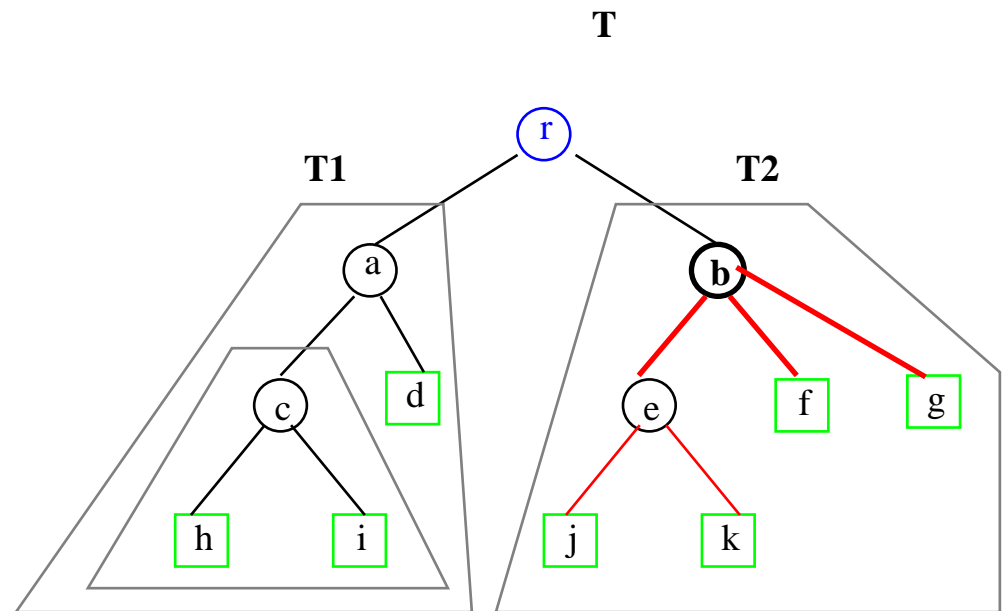
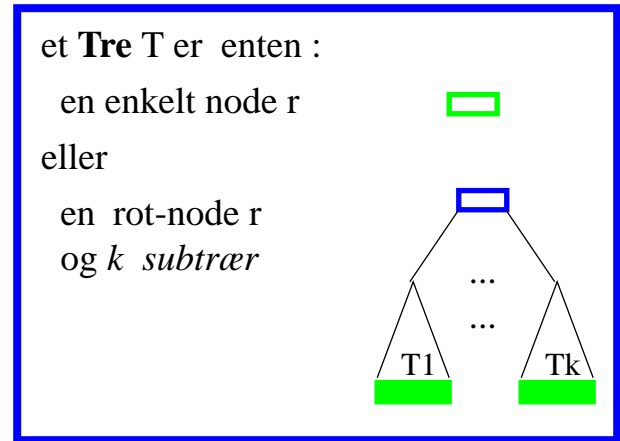
T i figuren har 12 noder (generellt ADT position), **r** er **roten** i **T**



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), r er **roten** i T

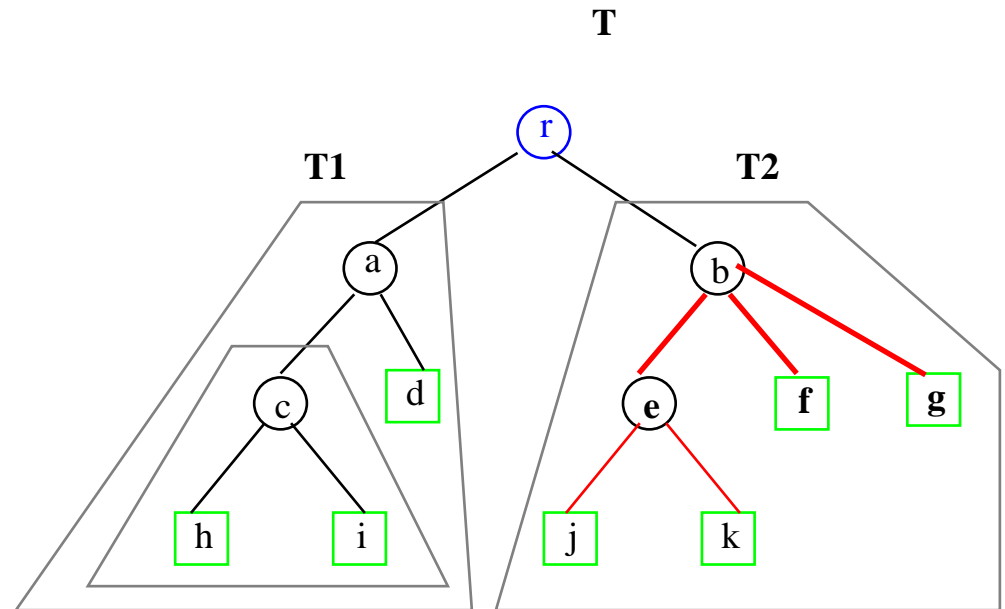
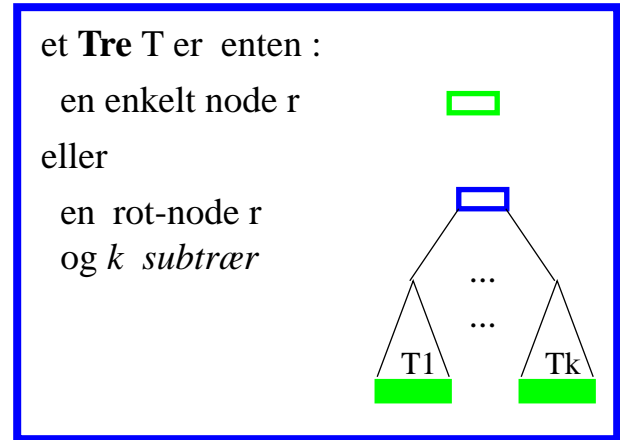
1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), r er **roten** i T

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), r er **roten** i T

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn

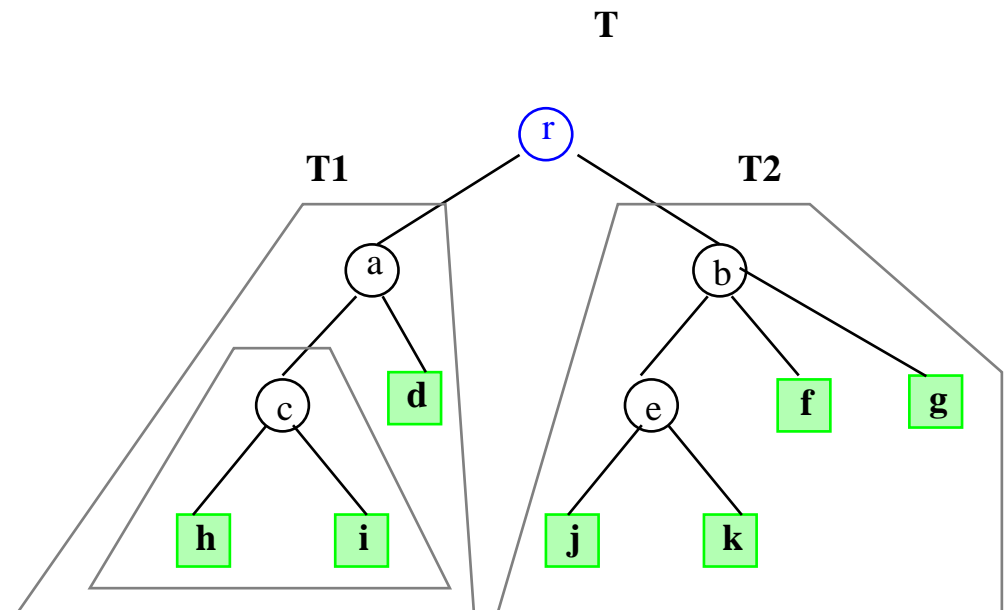
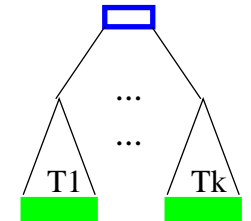
et **Tre** T er enten :

en enkelt node r



eller

en rot-node r
og k *subtrær*



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), r er **roten** i T

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv

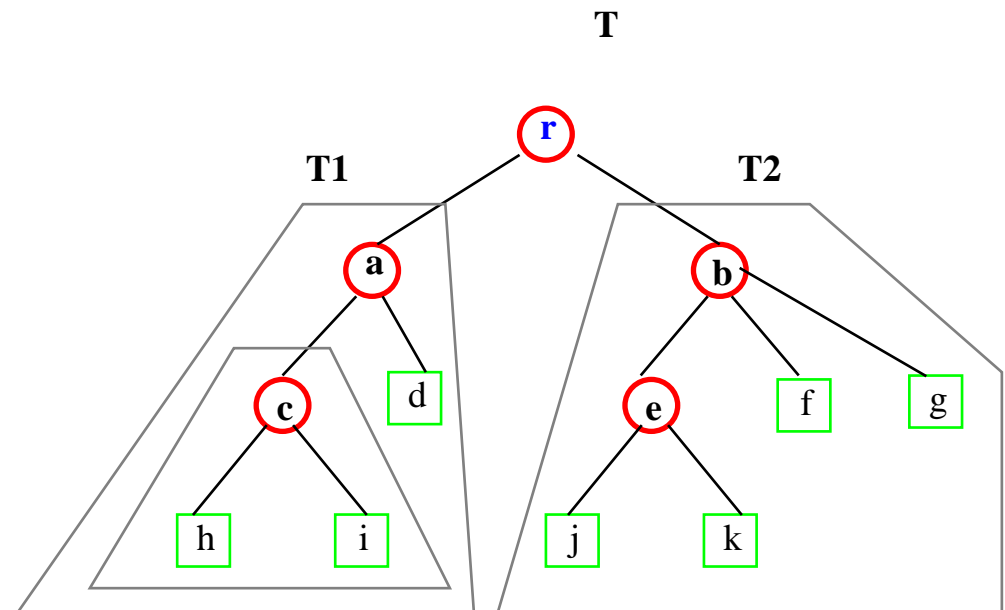
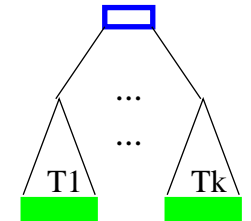
et **Tre** T er enten:

en enkelt node r



eller

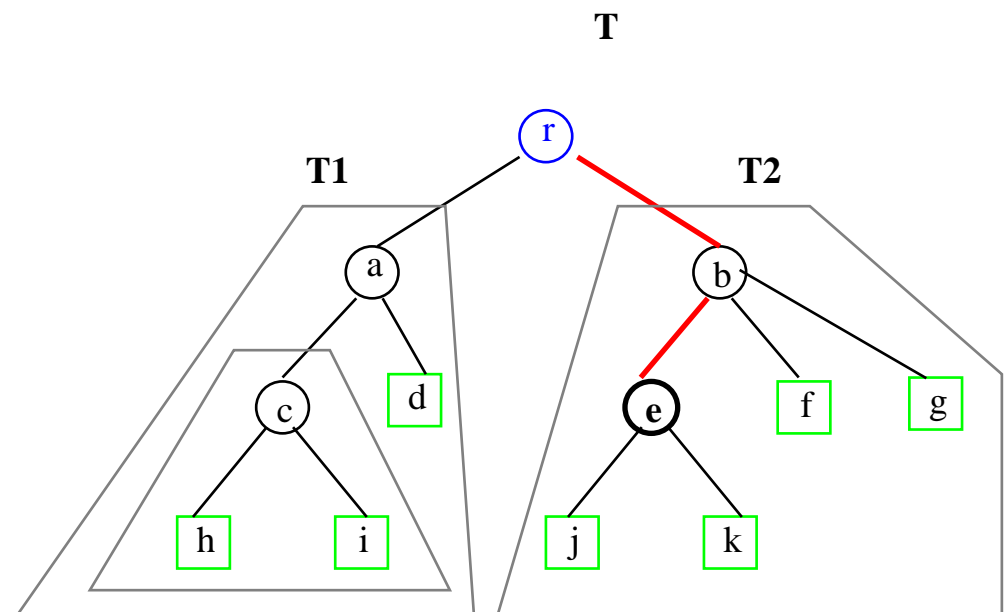
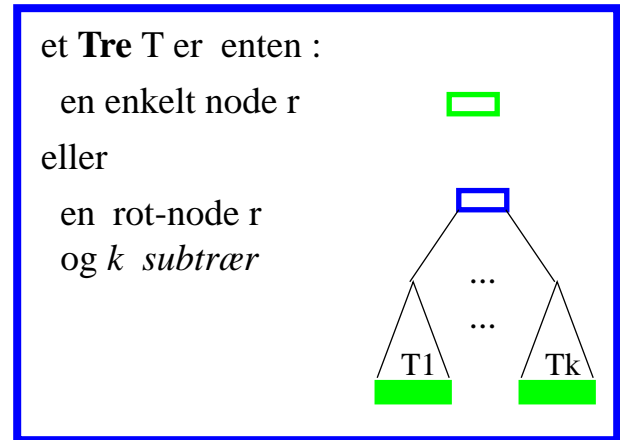
en rot-node r
og k *subtrær*



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), r er **roten** i T

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv
5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)
 $\text{depth}(p) = \text{if isRoot}(p) \text{ return } 0$
 $\text{else return } 1 + \text{depth}(\text{parent}(p))$



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), r er **roten** i T

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv
5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)
 $\text{depth}(p) = \begin{cases} \text{if isRoot}(p) \text{ return } 0 \\ \text{else return } 1 + \text{depth}(\text{parent}(p)) \end{cases}$
6. **høyden** av **b** = 2: avstand til fjerneste løv under b
 $\text{height}(p) = \begin{cases} \text{if isExternal}(p) \text{ return } 0 \\ \text{else return } 1 + \max\{ \text{height}(x) : x \text{ er et av children}(p) \} \end{cases}$

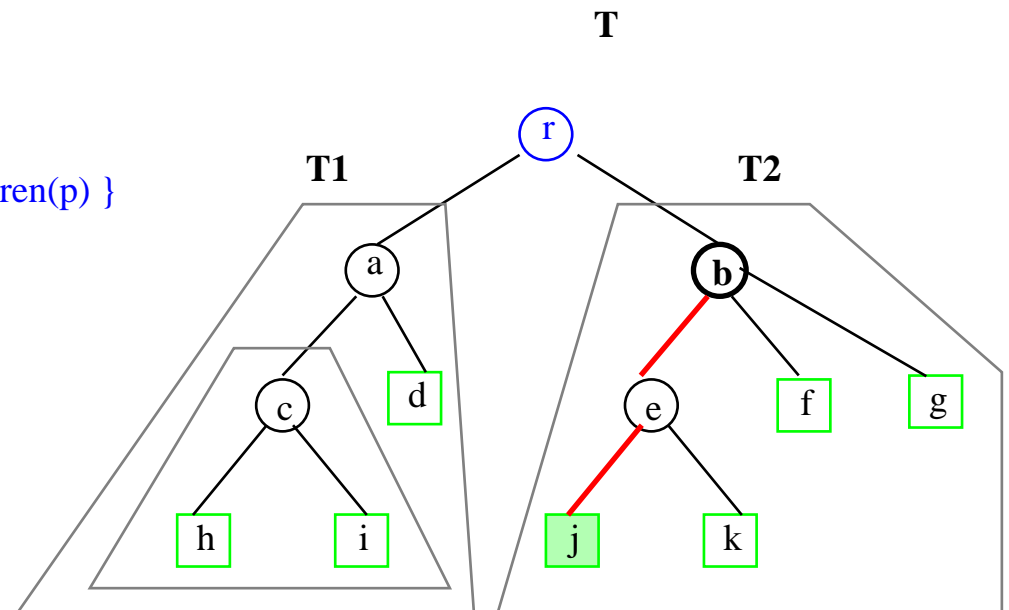
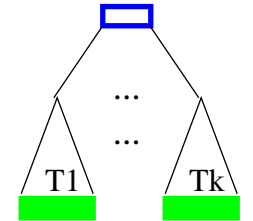
et **Tre** T er enten:

en enkelt node r



eller

en rot-node r
og k *subtrær*



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), r er **roten** i T

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv
5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)
 $\text{depth}(p) = \begin{cases} \text{return } 0 & \text{if isRoot}(p) \\ \text{return } 1 + \text{depth}(\text{parent}(p)) & \text{else} \end{cases}$
6. **høyden** av **b** = 2: avstand til fjerneste løv under b
 $\text{height}(p) = \begin{cases} \text{return } 0 & \text{if isExternal}(p) \\ \text{return } 1 + \max\{\text{height}(x) : x \text{ er et av children}(p)\} & \text{else} \end{cases}$

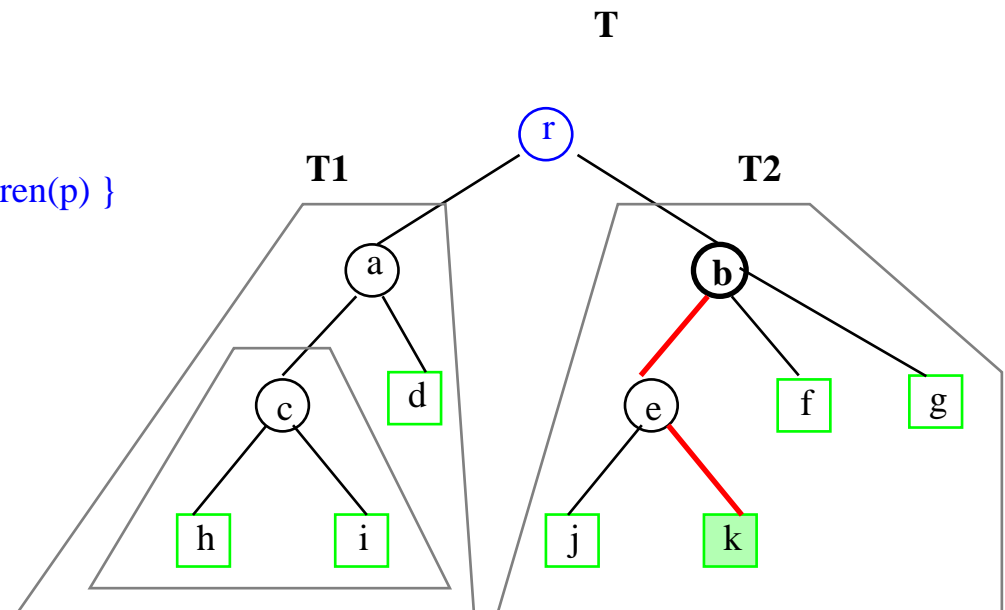
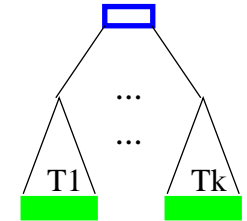
et **Tre** T er enten:

en enkelt node r



eller

en rot-node r
og k *subtrær*



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), r er **roten** i T

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv
5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)
 $depth(p) = \text{if isRoot}(p) \text{ return } 0$
 $\text{else return } 1 + depth(\text{parent}(p))$
6. **høyden** av **b** = 2: avstand til fjerneste løv under b
 $height(p) = \text{if isExternal}(p) \text{ return } 0$
 $\text{else return } 1 + \max\{ height(x) : x \text{ er et av children}(p) \}$
høyden av T = høyden av roten til T

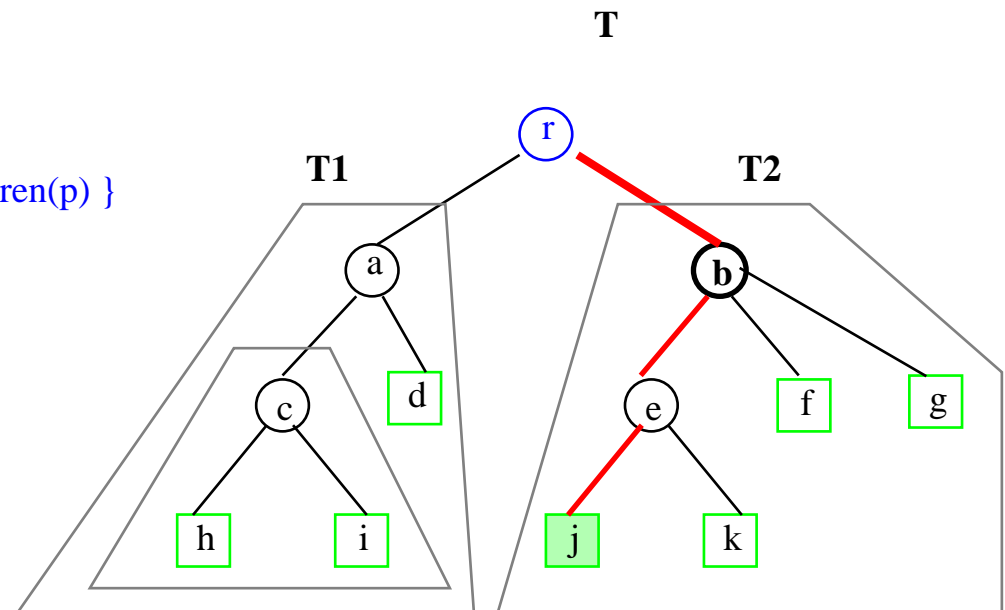
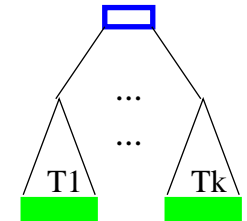
et **Tre** T er enten:

en enkelt node r



eller

en rot-node r
og k *subtrær*



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), r er **roten** i T

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv
5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)
 $\text{depth}(p) = \text{if isRoot}(p) \text{ return } 0$
 $\text{else return } 1 + \text{depth}(\text{parent}(p))$
6. **høyden** av **b** = 2: avstand til fjerneste løv under b
 $\text{height}(p) = \text{if isExternal}(p) \text{ return } 0$
 $\text{else return } 1 + \max\{ \text{height}(x) : x \text{ er et av children}(p) \}$
høyden av T = høyden av roten til T
7. **grad** av **b** = 3: antall barn

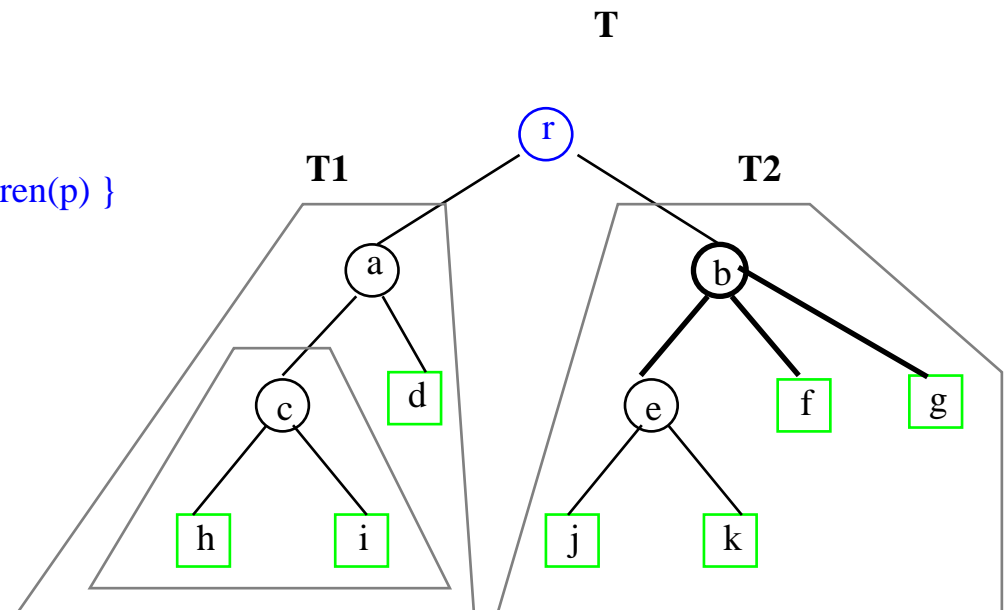
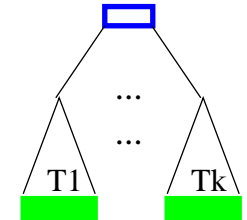
et **Tre** T er enten:

en enkelt node r



eller

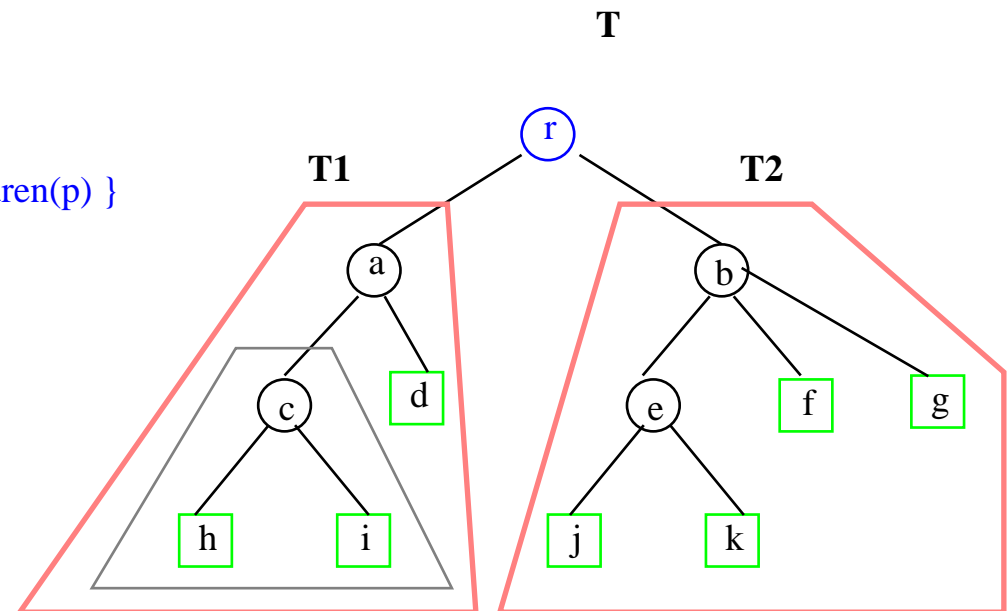
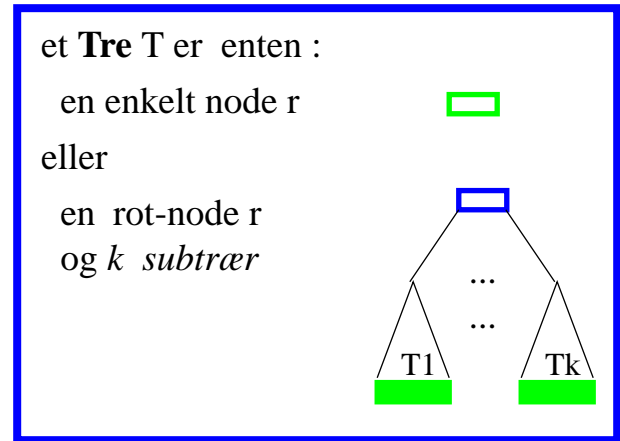
en rot-node r
og k *subtrær*



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), r er **roten** i T

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv
5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)
 $\text{depth}(p) = \text{if isRoot}(p) \text{ return } 0$
 $\text{else return } 1 + \text{depth}(\text{parent}(p))$
6. **høyden** av **b** = 2: avstand til fjerneste løv under b
 $\text{height}(p) = \text{if isExternal}(p) \text{ return } 0$
 $\text{else return } 1 + \max\{ \text{height}(x) : x \text{ er et av children}(p) \}$
høyden av **T** = høyden av roten til T
7. **grad** av **b** = 3: antall barn
8. **T1, T2** er **deltrær / subtrær** av T



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), r er **roten** i T

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv
5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)
 $\text{depth}(p) = \text{if isRoot}(p) \text{ return } 0$
 $\text{else return } 1 + \text{depth}(\text{parent}(p))$
6. **høyden** av **b** = 2: avstand til fjerneste løv under b
 $\text{height}(p) = \text{if isExternal}(p) \text{ return } 0$
 $\text{else return } 1 + \max\{ \text{height}(x) : x \text{ er et av children}(p) \}$
høyden av **T** = høyden av roten til T
7. **grad** av **b** = 3: antall barn
8. **T1, T2** er **deltrær / subtrær** av T

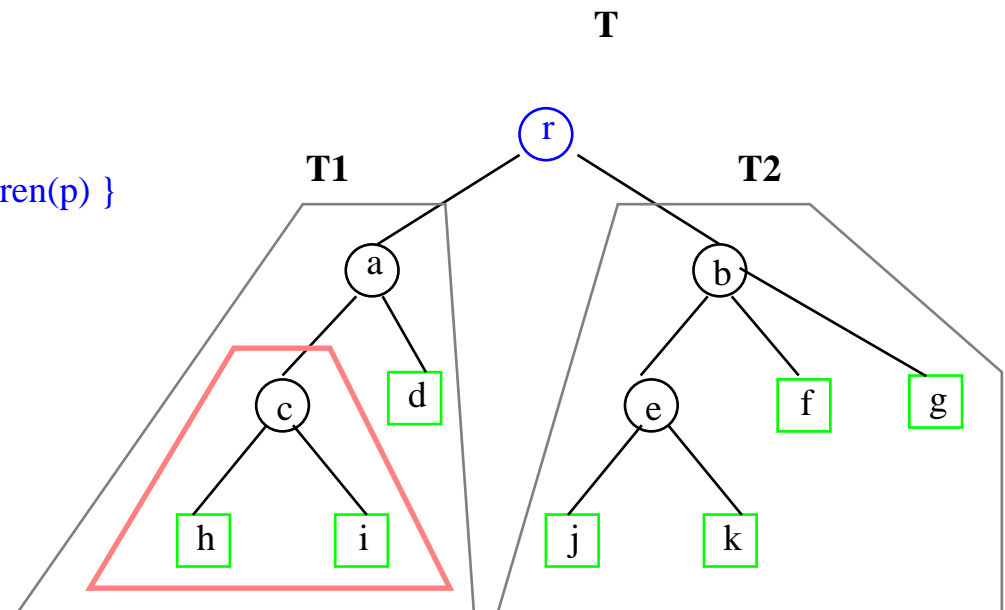
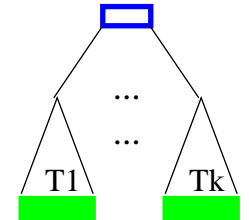
et **Tre** T er enten:

en enkelt node r



eller

en rot-node r
og k subtrær



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), **r** er **roten** i T

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv
5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)
 $\text{depth}(p) = \begin{cases} \text{return } 0 & \text{if isRoot}(p) \\ \text{return } 1 + \text{depth}(\text{parent}(p)) & \text{else} \end{cases}$

6. **høyden** av **b** = 2: avstand til fjerneste løv under b
 $\text{height}(p) = \begin{cases} \text{return } 0 & \text{if isExternal}(p) \\ \text{return } 1 + \max\{\text{height}(x) : x \text{ er et av children}(p)\} & \text{else} \end{cases}$

høyden av T = høyden av roten til T

7. **grad** av **b** = 3: antall barn
8. **T1, T2** er **deltrær** av T

Noen egenskaper

9. **antall kanter = antall noder - 1**

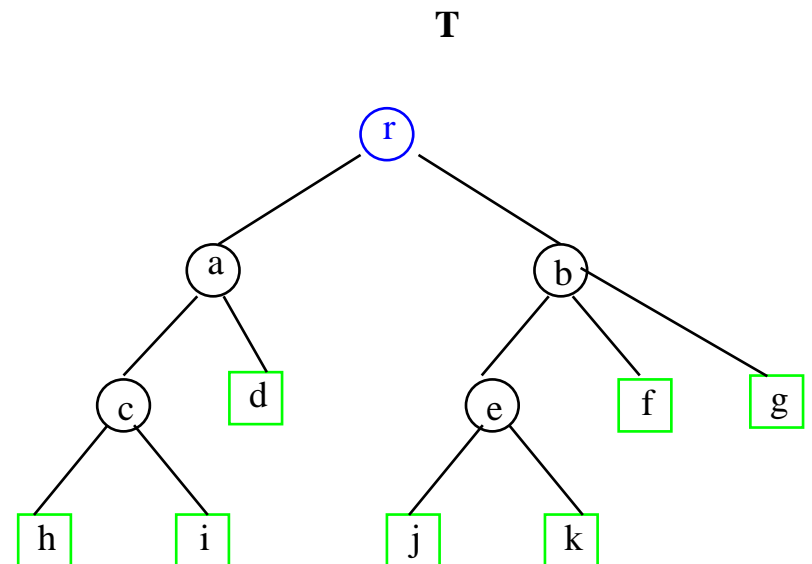
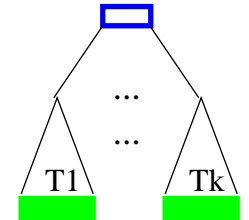
et **Tre** T er enten:

en enkelt node r



eller

en rot-node r
og k *subtrær*



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), **r** er **roten** i **T**

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv
5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)
 $\text{depth}(p) = \begin{cases} \text{return } 0 & \text{if isRoot}(p) \\ \text{return } 1 + \text{depth}(\text{parent}(p)) & \text{else} \end{cases}$

6. **høyden** av **b** = 2: avstand til fjerneste løv under **b**
 $\text{height}(p) = \begin{cases} \text{return } 0 & \text{if isExternal}(p) \\ \text{return } 1 + \max\{\text{height}(x) : x \text{ er et av children}(p)\} & \text{else} \end{cases}$

høyden av **T** = høyden av roten til **T**

7. **grad** av **b** = 3: antall barn
8. **T1, T2** er **deltrær** av **T**

Noen egenskaper

9. **antall kanter = antall noder - 1**

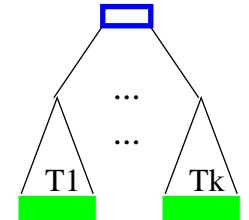
et **Tre** **T** er enten:

en enkelt node **r**

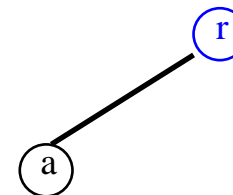


eller

en rot-node **r**
og **k** *subtrær*



T



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), **r** er **roten** i **T**

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn

4. **r, a, b, c, e** er **interne** noder: ikke løv

5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)

$\text{depth}(p) = \text{if isRoot}(p) \text{ return } 0$

$\text{else return } 1 + \text{depth}(\text{parent}(p))$

6. **høyden** av **b** = 2: avstand til fjerneste løv under **b**

$\text{height}(p) = \text{if isExternal}(p) \text{ return } 0$

$\text{else return } 1 + \max\{ \text{height}(x) : x \text{ er et av children}(p) \}$

høyden av **T** = høyden av roten til **T**

7. **grad** av **b** = 3: antall barn

8. **T1, T2** er **deltrær** av **T**

Noen egenskaper

9. antall kanter = antall noder - 1

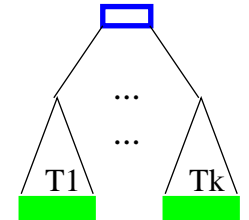
et **Tre** **T** er enten:

en enkelt node **r**

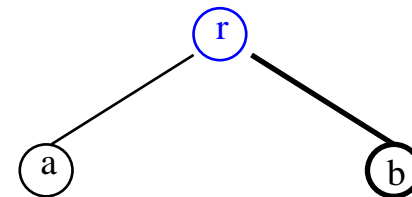


eller

en rot-node **r**
og **k subtrær**



T



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), **r** er **roten** i **T**

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv
5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)
 $\text{depth}(p) = \begin{cases} \text{return } 0 & \text{if isRoot}(p) \\ \text{return } 1 + \text{depth}(\text{parent}(p)) & \text{else} \end{cases}$

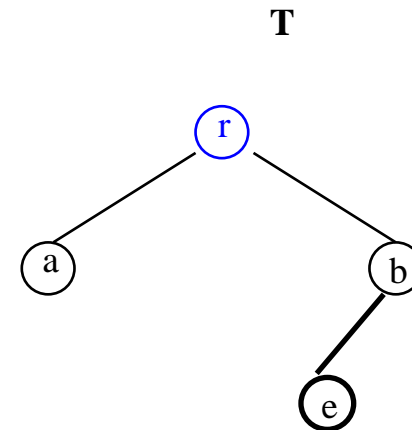
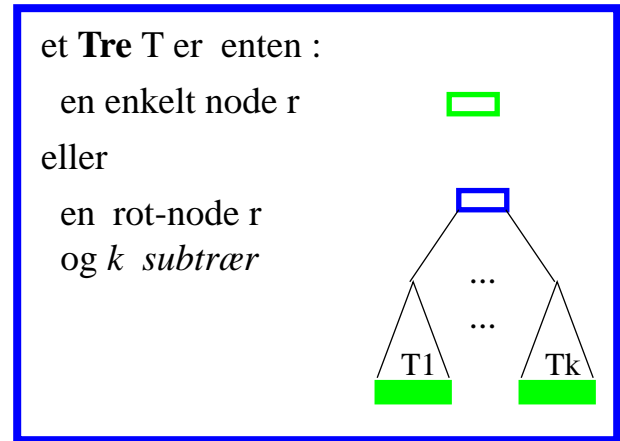
6. **høyden** av **b** = 2: avstand til fjerneste løv under **b**
 $\text{height}(p) = \begin{cases} \text{return } 0 & \text{if isExternal}(p) \\ \text{return } 1 + \max\{\text{height}(x) : x \text{ er et av children}(p)\} & \text{else} \end{cases}$

høyden av **T** = høyden av roten til **T**

7. **grad** av **b** = 3: antall barn
8. **T1, T2** er **deltrær** av **T**

Noen egenskaper

9. **antall kanter = antall noder - 1**



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), **r** er **roten** i **T**

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv
5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)
 $\text{depth}(p) = \text{if isRoot}(p) \text{ return } 0$
 $\text{else return } 1 + \text{depth}(\text{parent}(p))$

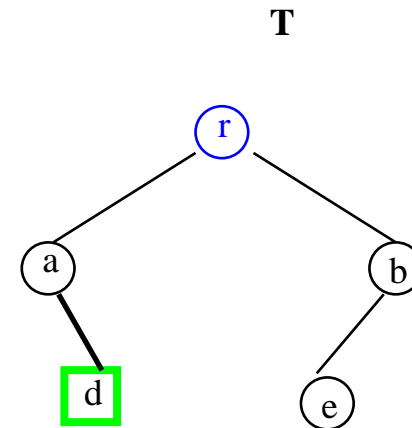
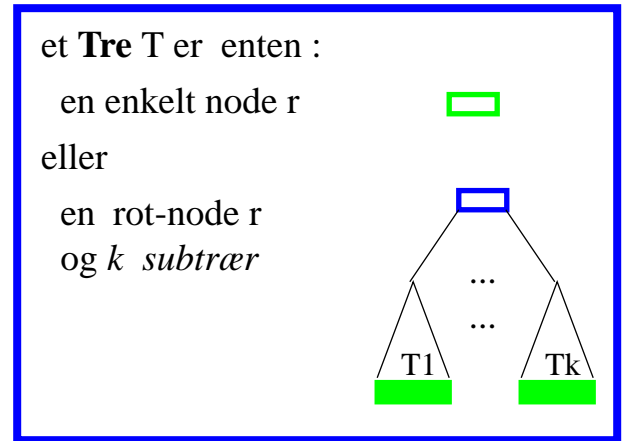
6. **høyden** av **b** = 2: avstand til fjerneste løv under **b**
 $\text{height}(p) = \text{if isExternal}(p) \text{ return } 0$
 $\text{else return } 1 + \max\{ \text{height}(x) : x \text{ er et av children}(p) \}$

høyden av **T** = høyden av roten til **T**

7. **grad** av **b** = 3: antall barn
8. **T1, T2** er **deltrær** av **T**

Noen egenskaper

9. **antall kanter = antall noder - 1**



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), **r** er **roten** i **T**

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv
5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)
 $\text{depth}(p) = \begin{cases} \text{return } 0 & \text{if isRoot}(p) \\ \text{return } 1 + \text{depth}(\text{parent}(p)) & \text{else} \end{cases}$
6. **høyden** av **b** = 2: avstand til fjerneste løv under b
 $\text{height}(p) = \begin{cases} \text{return } 0 & \text{if isExternal}(p) \\ \text{return } 1 + \max\{\text{height}(x) : x \text{ er et av children}(p)\} & \text{else} \end{cases}$
høyden av **T** = høyden av roten til **T**
7. **grad** av **b** = 3: antall barn
8. **T1, T2** er **deltrær** av **T**

Noen egenskaper

9. **antall kanter = antall noder - 1**

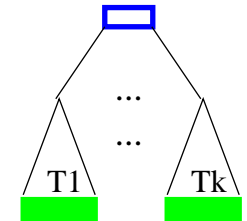
et **Tre** **T** er enten:

en enkelt node **r**

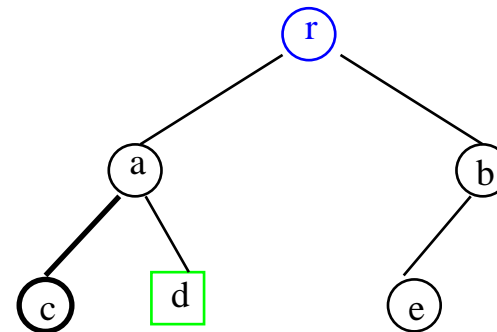


eller

en rot-node **r**
og k *subtrær*



T



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), r er **roten** i T

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv
5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)
 $\text{depth}(p) = \begin{cases} \text{return } 0 & \text{if isRoot}(p) \\ \text{return } 1 + \text{depth}(\text{parent}(p)) & \text{else} \end{cases}$

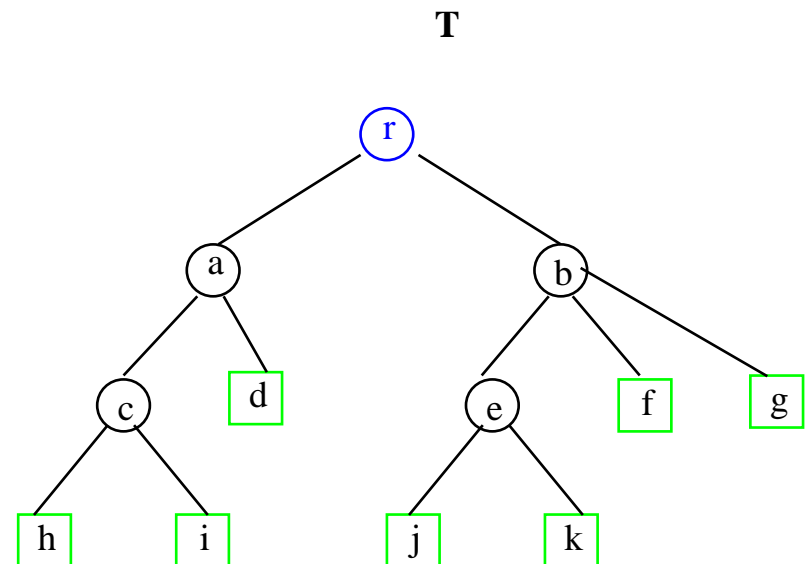
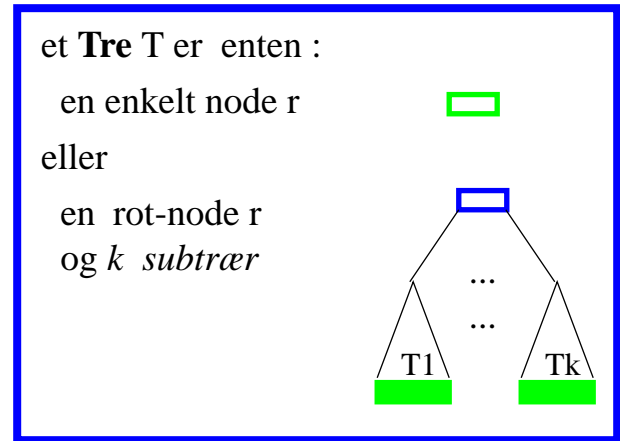
6. **høyden** av **b** = 2: avstand til fjerneste løv under b
 $\text{height}(p) = \begin{cases} \text{return } 0 & \text{if isExternal}(p) \\ \text{return } 1 + \max\{\text{height}(x) : x \text{ er et av children}(p)\} & \text{else} \end{cases}$

høyden av T = høyden av roten til T

7. **grad** av **b** = 3: antall barn
8. **T1, T2** er **deltrær** av T

Noen egenskaper

9. **antall kanter = antall noder - 1**



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), **r** er **roten** i T

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv
5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)
 $\text{depth}(p) = \begin{cases} \text{return } 0 & \text{if isRoot}(p) \\ \text{return } 1 + \text{depth}(\text{parent}(p)) & \text{else} \end{cases}$

6. **høyden** av **b** = 2: avstand til fjerneste løv under b
 $\text{height}(p) = \begin{cases} \text{return } 0 & \text{if isExternal}(p) \\ \text{return } 1 + \max\{\text{height}(x) : x \text{ er et av children}(p)\} & \text{else} \end{cases}$

høyden av T = høyden av roten til T

7. **grad** av **b** = 3: antall barn
8. **T1, T2** er **deltrær** av T

Noen egenskaper

9. antall kanter = antall noder - 1

10. hver node har en entydig sti til roten

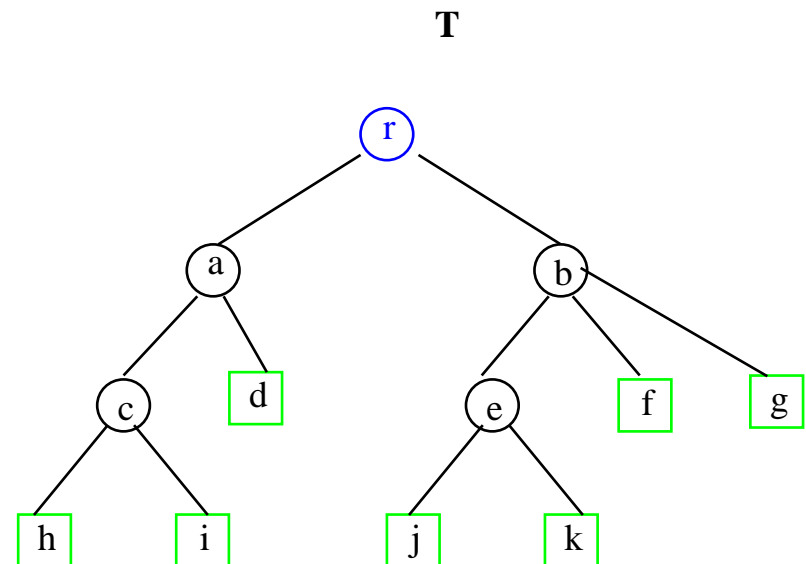
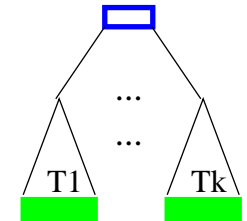
et **Tre** T er enten:

en enkelt node r



eller

en rot-node r
og k *subtrær*



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), **r** er **roten** i T

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv
5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)
 $\text{depth}(p) = \begin{cases} \text{return } 0 & \text{if isRoot}(p) \\ \text{return } 1 + \text{depth}(\text{parent}(p)) & \text{else} \end{cases}$

6. **høyden** av **b** = 2: avstand til fjerneste løv under b
 $\text{height}(p) = \begin{cases} \text{return } 0 & \text{if isExternal}(p) \\ \text{return } 1 + \max\{\text{height}(x) : x \text{ er et av children}(p)\} & \text{else} \end{cases}$

høyden av T = høyden av roten til T

7. **grad** av **b** = 3: antall barn
8. **T1, T2** er **deltrær** av T

Noen egenskaper

9. antall kanter = antall noder - 1

10. hver node har en entydig sti til roten

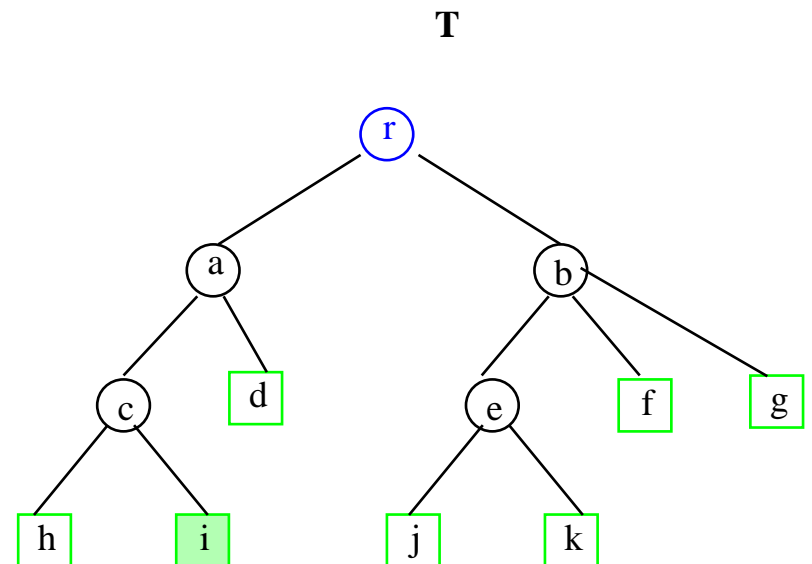
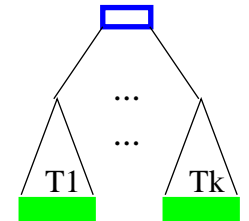
et **Tre** T er enten:

en enkelt node r



eller

en rot-node r
og k *subtrær*



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), r er **roten** i T

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv
5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)
 $\text{depth}(p) = \begin{cases} \text{return } 0 & \text{if isRoot}(p) \\ \text{return } 1 + \text{depth}(\text{parent}(p)) & \text{else} \end{cases}$

6. **høyden** av **b** = 2: avstand til fjerneste løv under b
 $\text{height}(p) = \begin{cases} \text{return } 0 & \text{if isExternal}(p) \\ \text{return } 1 + \max\{\text{height}(x) : x \text{ er et av children}(p)\} & \text{else} \end{cases}$

høyden av T = høyden av roten til T

7. **grad** av **b** = 3: antall barn
8. **T1, T2** er **deltrær** av T

Noen egenskaper

9. antall kanter = antall noder - 1

10. hver node har en entydig sti til roten

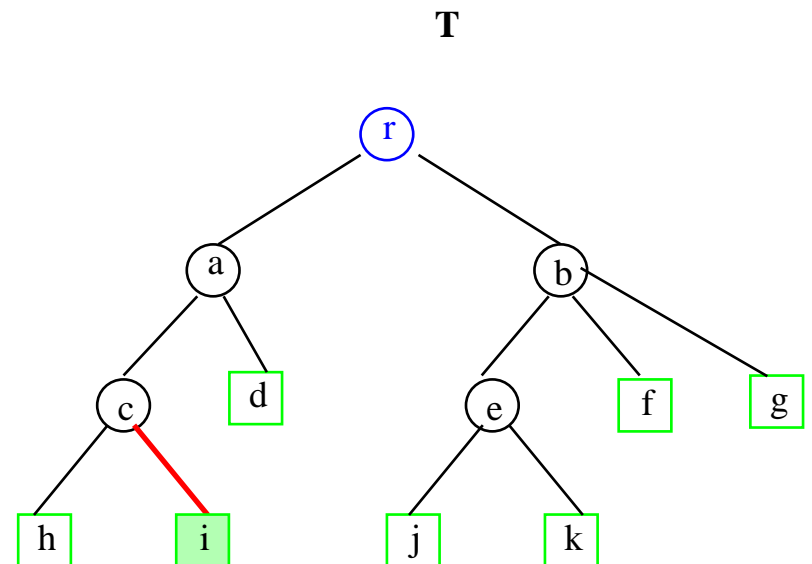
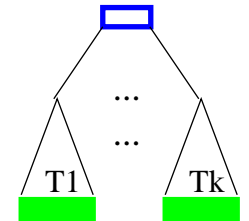
et **Tre** T er enten:

en enkelt node r



eller

en rot-node r
og k *subtrær*



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), **r** er **roten** i T

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv
5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)
 $\text{depth}(p) = \begin{cases} \text{return } 0 & \text{if isRoot}(p) \\ \text{return } 1 + \text{depth}(\text{parent}(p)) & \text{else} \end{cases}$

6. **høyden** av **b** = 2: avstand til fjerneste løv under b
 $\text{height}(p) = \begin{cases} \text{return } 0 & \text{if isExternal}(p) \\ \text{return } 1 + \max\{\text{height}(x) : x \text{ er et av children}(p)\} & \text{else} \end{cases}$

høyden av T = høyden av roten til T

7. **grad** av **b** = 3: antall barn
8. **T1, T2** er **deltrær** av T

Noen egenskaper

9. antall kanter = antall noder - 1

10. hver node har en entydig sti til roten

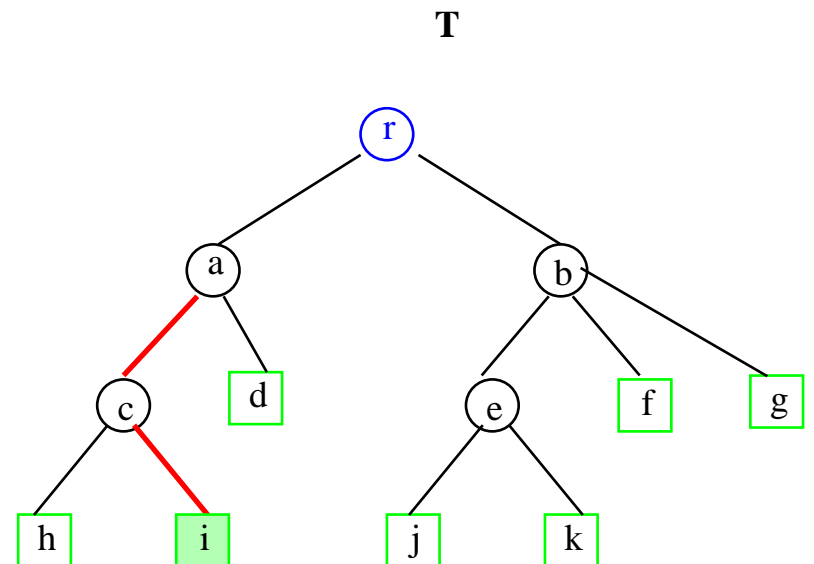
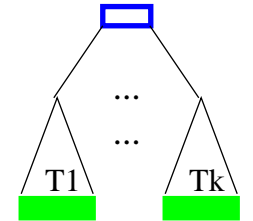
et **Tre** T er enten:

en enkelt node r



eller

en rot-node r
og k *subtrær*



Terminologi (fra slektskapstrær)

T i figuren har 12 noder (generellt ADT position), **r** er **roten** i T

1. **b** er **foreldrenode** (parent) til **e, f, g**, og **forgjenger** til **e, f, g, j, k**
2. **e, f, g** er **barna** til **b**: umiddelbare etterfølgere (og disse er **søsken**)
3. **d, f, g, h, i, j, k** **eksterne** noder (**løv**): har ingen barn
4. **r, a, b, c, e** er **interne** noder: ikke løv
5. **dybden** av **e** = 2: lengden av stien fra roten (**nivå**)
 $\text{depth}(p) = \begin{cases} \text{return } 0 & \text{if isRoot}(p) \\ \text{return } 1 + \text{depth}(\text{parent}(p)) & \text{else} \end{cases}$

6. **høyden** av **b** = 2: avstand til fjerneste løv under b
 $\text{height}(p) = \begin{cases} \text{return } 0 & \text{if isExternal}(p) \\ \text{return } 1 + \max\{\text{height}(x) : x \text{ er et av children}(p)\} & \text{else} \end{cases}$

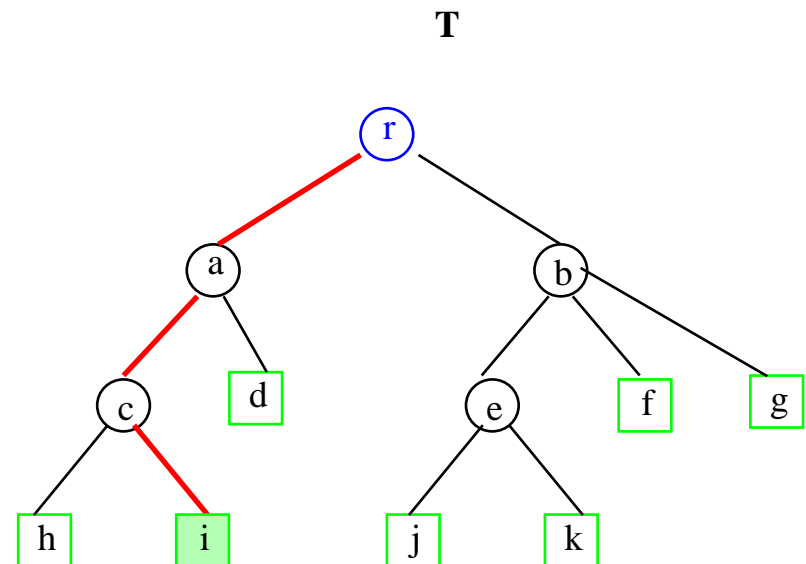
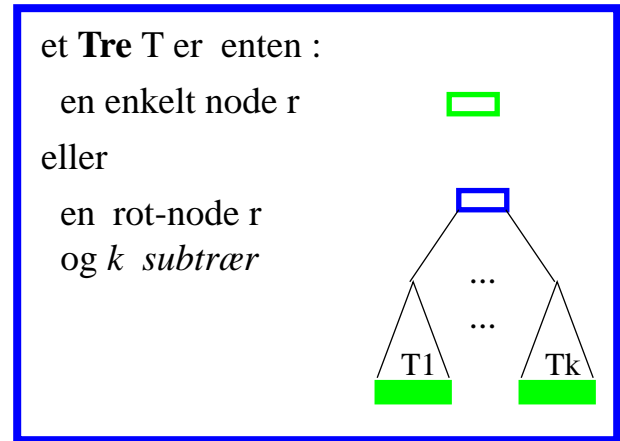
høyden av T = høyden av roten til T

7. **grad** av **b** = 3: antall barn
8. **T1, T2** er **deltrær** av T

Noen egenskaper

9. antall kanter = antall noder - 1

10. hver node har en entydig sti til roten



2. Binære Trær

- hver node har **2** eller **0** barn
- barna er ordnet: **venstre og høyre**

et **Binært Tre** er enten :

en *node* r

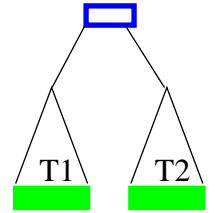


eller

en *node* r

og

2 binære subtrær
(venstre og høyre)



2. Binære Trær

- hver node har **2** eller **0** barn
- barna er ordnet: **venstre og høyre**

et **Binært Tre** er enten :

en *node* r

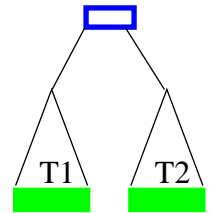


eller

en *node* r

og

2 binære subtrær
(venstre og høyre)



av og til : **høyst 2** binære subtrær

2. Binære Trær

- hver node har **2** eller **0** barn
- barna er ordnet: **venstre** og **høyre**

et **Binært Tre** er enten :

en *node* r



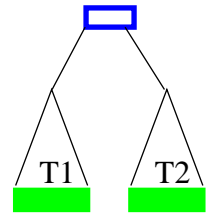
eller

en *node* r

og

2 binære subtrær

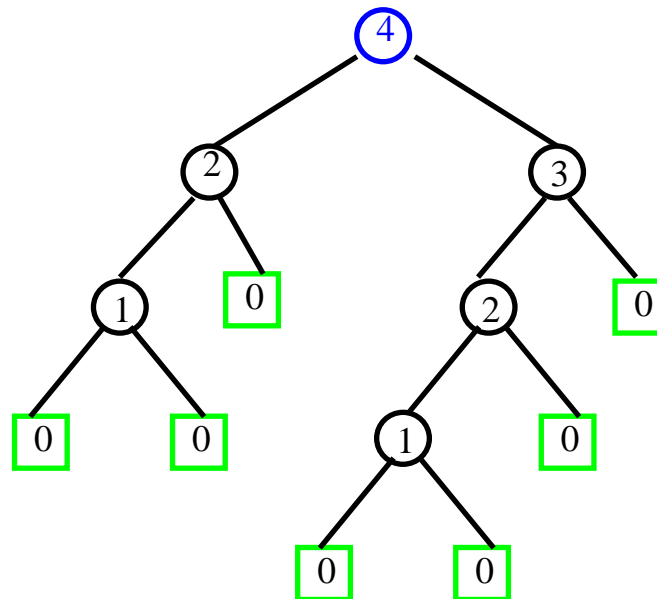
(venstre og høyre)



av og til : **høyst 2** binære subtrær

Noen egenskaper (h: høyde; n: antall noder = # noder)

A. # eksterne noder = # interne noder + 1



2. Binære Trær

- hver node har **2** eller **0** barn
- barna er ordnet: **venstre** og **høyre**

et **Binært Tre** er enten :

en *node* r



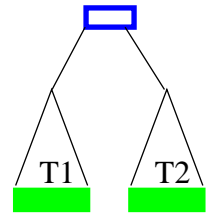
eller

en *node* r

og

2 binære subtrær

(venstre og høyre)

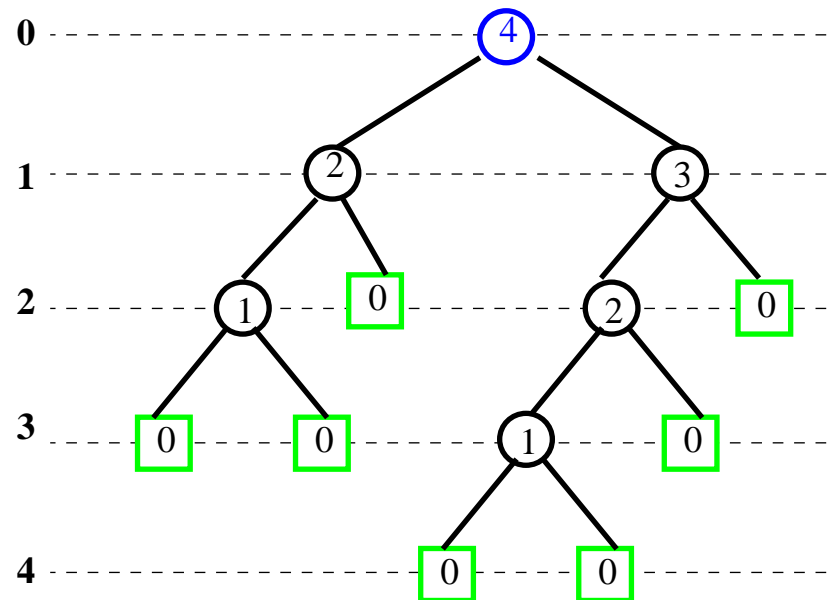


av og til : **høyst 2** binære subtrær

Noen egenskaper (h: høyde; n: antall noder = # noder)

- A. # eksterne noder = # interne noder + 1
- B. # noder på nivå i : $\leq 2^i$

Nivå / dybde



2. Binære Trær

- hver node har **2** eller **0** barn
- barna er ordnet: **venstre og høyre**

Noen egenskaper (h: høyde; n: antall noder = # noder)

- A. # eksterne noder = # interne noder + 1
- B. # noder på nivå i : $\leq 2^i$

et **Binært Tre** er enten :

en *node* r

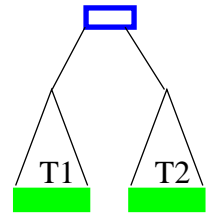


eller

en *node* r

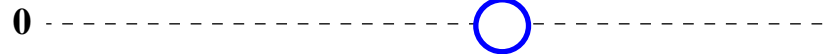
og

2 binære subtrær
(venstre og høyre)



av og til : **høyst 2** binære subtrær

Nivå / dybde



2. Binære Trær

- hver node har **2** eller **0** barn
- barna er ordnet: **venstre og høyre**

et **Binært Tre** er enten :

en *node* r



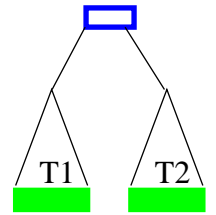
eller

en *node* r

og

2 binære subtrær

(venstre og høyre)

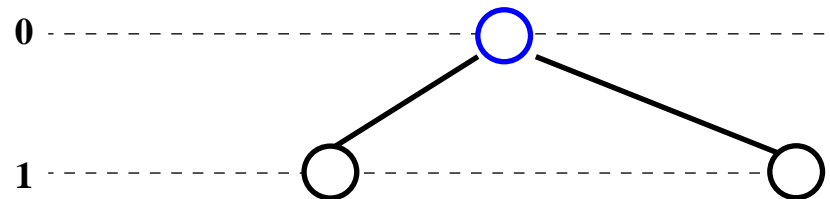


av og til : **høyst 2** binære subtrær

Noen egenskaper (h: høyde; n: antall noder = # noder)

- A. # eksterne noder = # interne noder + 1
- B. # noder på nivå i : $\leq 2^i$

Nivå / dybde



2. Binære Trær

- hver node har **2** eller **0** barn
- barna er ordnet: **venstre og høyre**

et **Binært Tre** er enten :

en *node* r



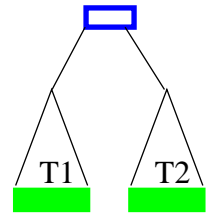
eller

en *node* r

og

2 binære subtrær

(venstre og høyre)

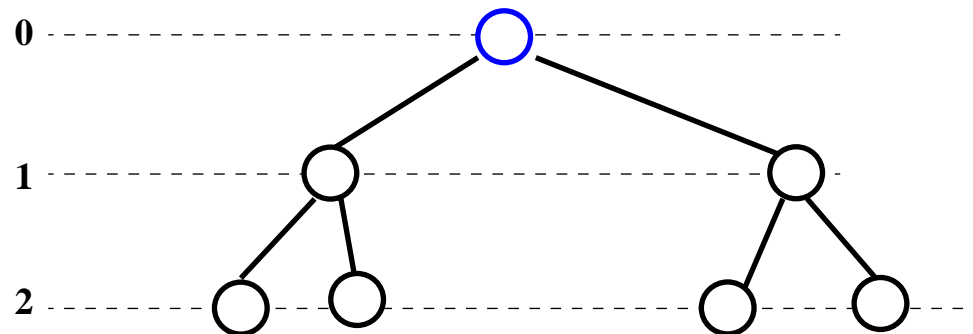


av og til : **høyst 2** binære subtrær

Noen egenskaper (h: høyde; n: antall noder = # noder)

- A. # eksterne noder = # interne noder + 1
- B. # noder på nivå i : $\leq 2^i$

Nivå / dybde



2. Binære Trær

- hver node har **2** eller **0** barn
- barna er ordnet: **venstre** og **høyre**

et **Binært Tre** er enten :

en *node* r



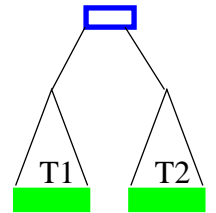
eller

en *node* r

og

2 binære subtrær

(venstre og høyre)

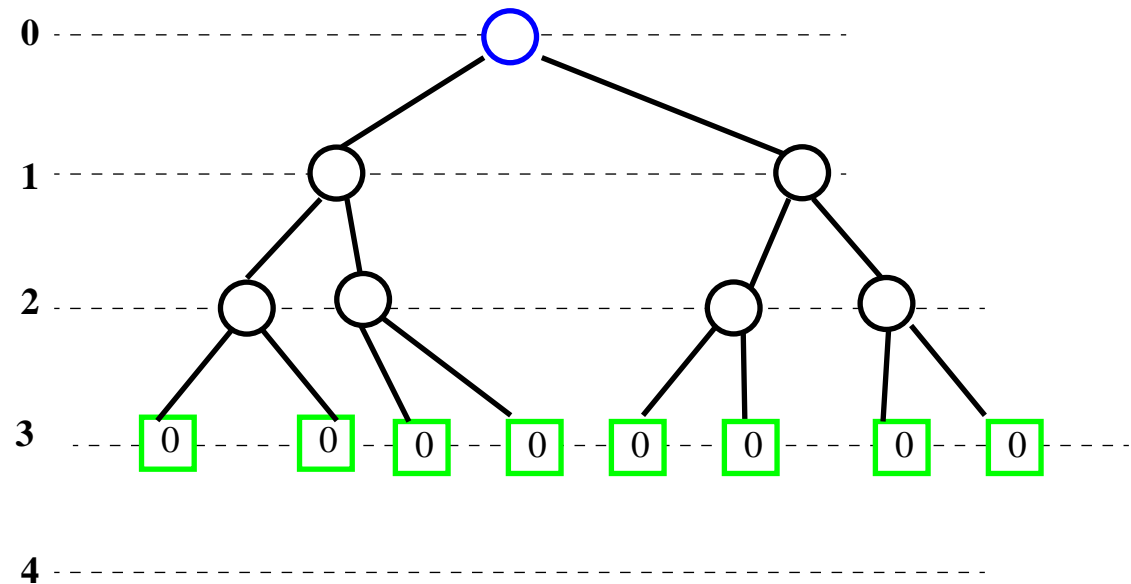


av og til : **høyst 2** binære subtrær

Noen egenskaper (h: høyde; n: antall noder = # noder)

- A. # eksterne noder = # interne noder + 1
- B. # noder på nivå i : $\leq 2^i$

Nivå / dybde



2. Binære Trær

- hver node har **2** eller **0** barn
- barna er ordnet: **venstre** og **høyre**

et **Binært Tre** er enten :

en *node* r



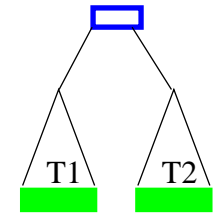
eller

en *node* r

og

2 binære subtrær

(venstre og høyre)

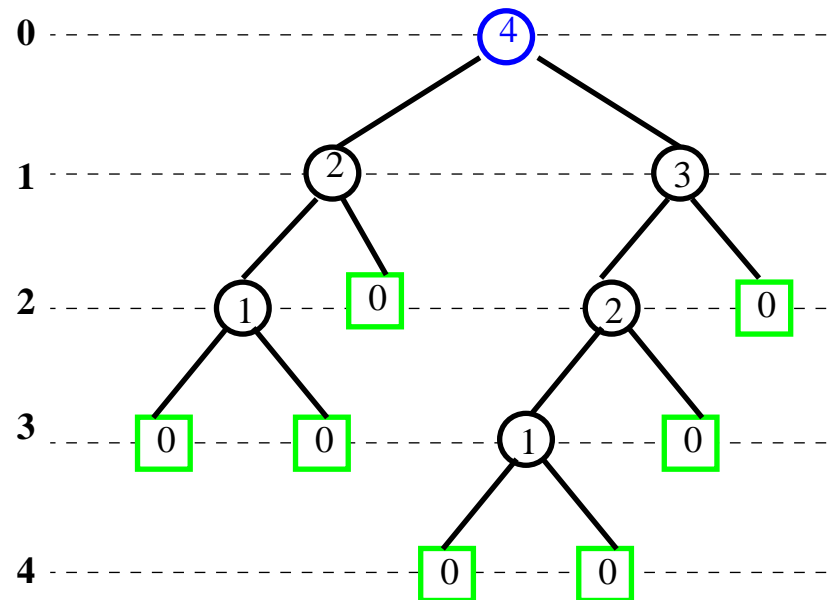


av og til : **høyst 2** binære subtrær

Noen egenskaper (h: høyde; n: antall noder = # noder)

- A. # eksterne noder = # interne noder + 1
- B. # noder på nivå i : $\leq 2^i$

Nivå / dybde



2. Binære Trær

- hver node har **2** eller **0** barn
- barna er ordnet: **venstre** og **høyre**

et **Binært Tre** er enten :

en *node* r



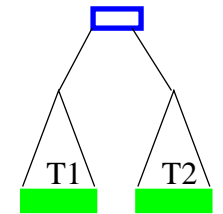
eller

en *node* r

og

2 binære subtrær

(venstre og høyre)

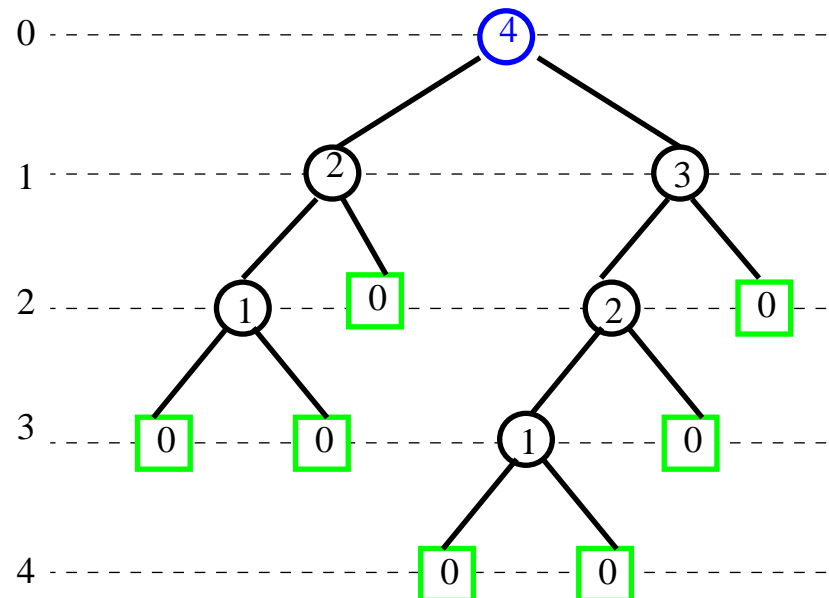


av og til : **høyst 2** binære subtrær

Noen egenskaper (h: høyde; n: antall noder = # noder)

- A. # eksterne noder = # interne noder + 1
- B. # noder på nivå i $\leq 2^i$
- C. $h+1 \leq (\# \text{eksterne noder}) \leq 2^h$
 $\log_2 (\# \text{eksterne noder}) \leq h$

Nivå / dybde



2. Binære Trær

- hver node har **2** eller **0** barn
- barna er ordnet: **venstre** og **høyre**

et **Binært Tre** er enten :

en *node* r



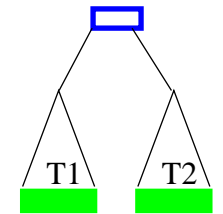
eller

en *node* r

og

2 binære subtrær

(venstre og høyre)

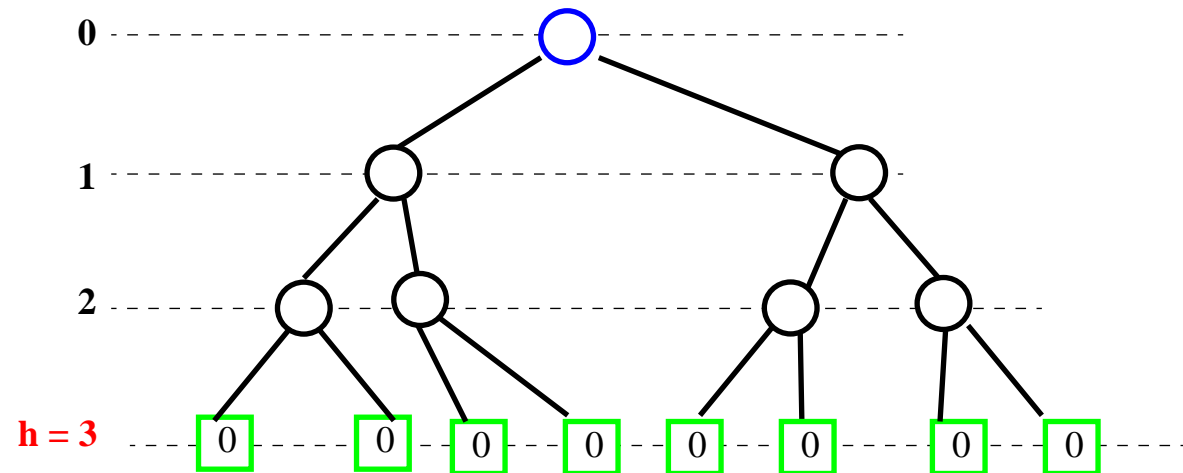


av og til : **høyst 2** binære subtrær

Noen egenskaper (h : høyde; n : antall noder = # noder)

- A. # eksterne noder = # interne noder + 1
- B. # noder på nivå $i \leq 2^i$
- C. $h+1 \leq (\text{\# eksterne noder}) \leq 2^h$

Nivå / dybde



2. Binære Trær

- hver node har **2** eller **0** barn
- barna er ordnet: **venstre** og **høyre**

et **Binært Tre** er enten :

en *node* r

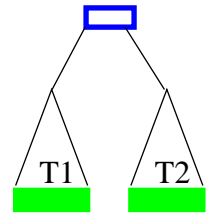


eller

en *node* r

og

2 binære subtrær
(venstre og høyre)

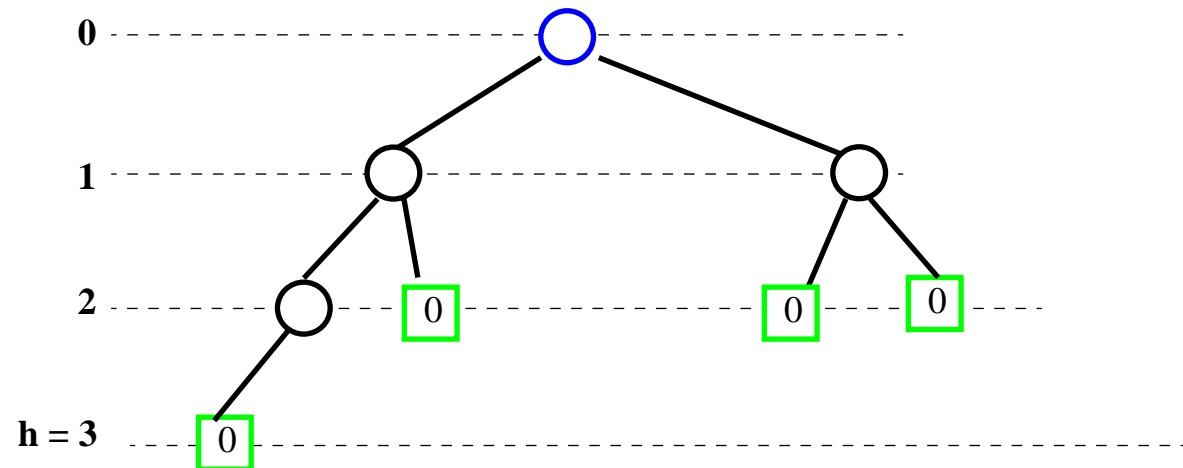


av og til : **høyst 2** binære subtrær

Noen egenskaper (h: høyde; n: antall noder = # noder)

- A. # eksterne noder = # interne noder + 1
- B. # noder på nivå i $\leq 2^i$
- C. **h+1** \leq (# eksterne noder) $\leq 2^h$

Nivå / dybde



2. Binære Trær

- hver node har **2** eller **0** barn
- barna er ordnet: **venstre** og **høyre**

et **Binært Tre** er enten :

en *node* r



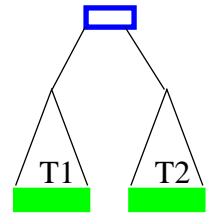
eller

en *node* r

og

2 binære subtrær

(venstre og høyre)

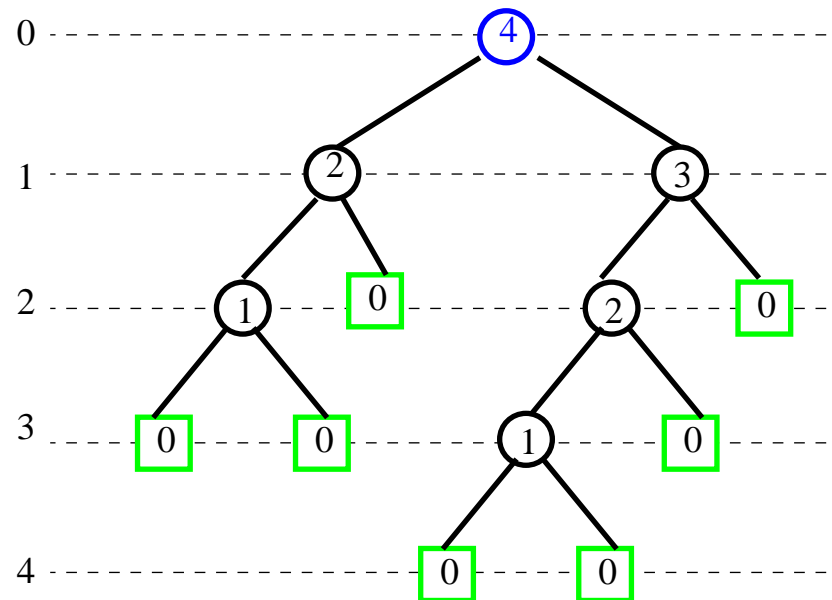


av og til : **høyst 2** binære subtrær

Noen egenskaper (h: høyde; n: antall noder = # noder)

- A. # eksterne noder = # interne noder + 1
- B. # noder på nivå $i \leq 2^i$
- C. $h+1 \leq (\# \text{ eksterne noder}) \leq 2^h$
 $\log_2 (\# \text{ eksterne noder}) \leq h$


Nivå / dybde



2. Binære Trær

- hver node har **2** eller **0** barn
- barna er ordnet: **venstre** og **høyre**

et **Binært Tre** er enten :

en *node* r 

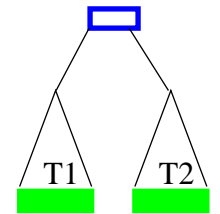
eller

en *node* r

og

2 binære subtrær

(venstre og høyre)



av og til : **høyst 2** binære subtrær

Noen egenskaper (h: høyde; n: antall noder = # noder)

A. # eksterne noder = # **interne noder** + 1

B. # noder på nivå i $\leq 2^i$

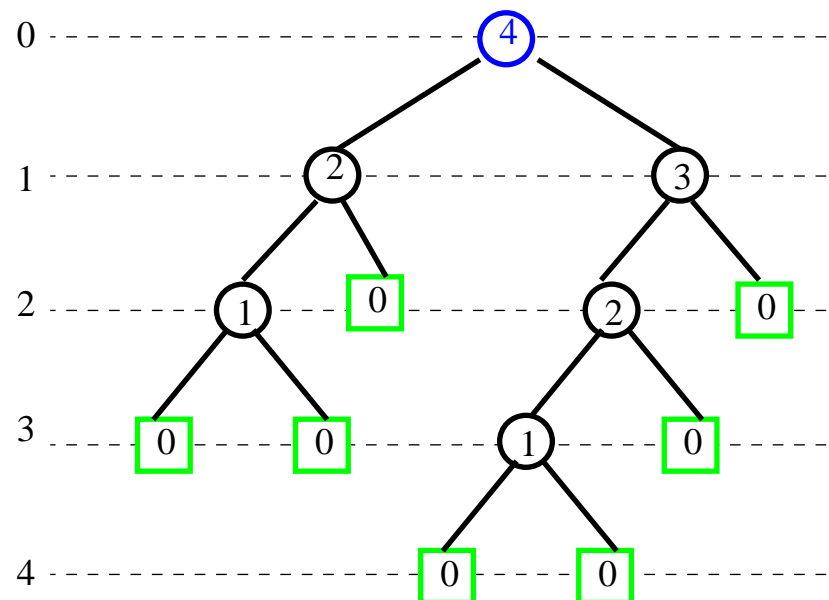
C. $h+1 \leq (\# \text{ eksterne noder}) \leq 2^h$

$\log_2 (\# \text{ eksterne noder}) \leq h$

D. $h \leq (\# \text{ interne noder}) \leq 2^h - 1$

$\log_2 (\# \text{ interne noder}) \leq h$


Nivå / dybde



2. Binære Trær

- hver node har **2** eller **0** barn
- barna er ordnet: **venstre** og **høyre**

et **Binært Tre** er enten :

en *node* r 

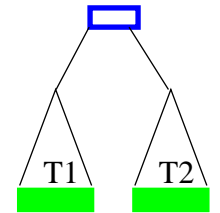
eller

en *node* r

og

2 binære subtrær

(venstre og høyre)



av og til : **høyst 2** binære subtrær

Noen egenskaper (h: høyde; n: antall noder = # noder)

A. # eksterne noder = # interne noder + 1

B. # noder på nivå i $\leq 2^i$

C. $h+1 \leq (\# \text{ eksterne noder}) \leq 2^h$

$\log_2 (\# \text{ eksterne noder}) \leq h$

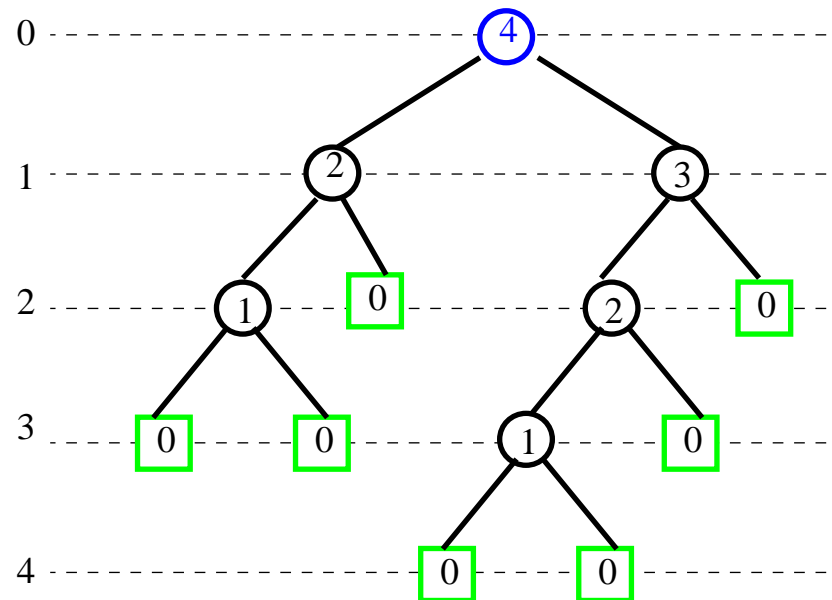
D. $h \leq (\# \text{ interne noder}) \leq 2^h - 1$

$\log_2 (\# \text{ interne noder}) \leq h$

E. $2h+1 \leq n \leq 2^{(h+1)} - 1$

$\log_2(n+1) - 1 \leq h \leq (n-1)/2$

Nivå / dybde



3. package `jdsl.core.api`;

```
public interface InspectableContainer
{ Enumeration elements();
  boolean isEmpty();
  int size();
  Container newCont
}
```

```
public interface InspectablePositionalContainer extends Container
{ Enumeration positions();
  void swap(Position p, Position q);
  Object replace(Position p, Object e);
}
```

```
/** Et tre har minst en ekstern node – rot */
public interface InspectableTree
  extends PositionalContainer {
  Position root();
  Position parent(Position v);
  /** Barn er ordnet fra venstre til høyre */
  PositionIterator children(Position p);
  boolean isInternal(Position v);
  boolean isExternal(Position v);
  boolean isRoot(Position v);
  PositionIterator siblings(Position p);
}
```

3. package `jdsl.core.api`;

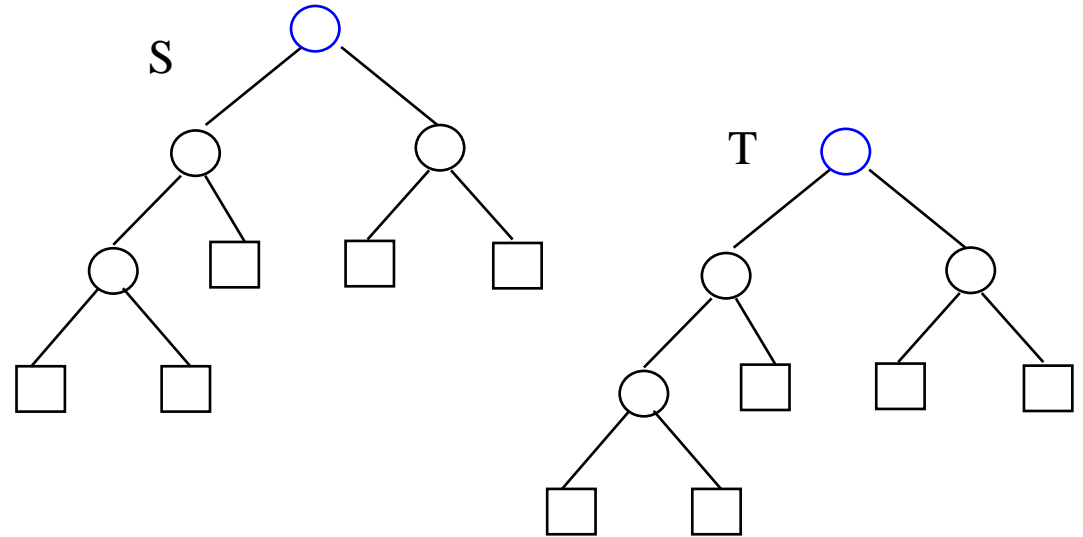
```
public interface InspectableContainer
{ Enumeration elements();
  boolean isEmpty();
  int size();
  Container newCont
}
```

```
public interface InspectablePositionalContainer extends Container
{ Enumeration positions();
  void swap(Position p, Position q);
  Object replace(Position p, Object e);
}
```

```
/** Et tre har minst en ekstern node – rot */
public interface InspectableTree
  extends PositionalContainer {
  Position root();
  Position parent(Position v);
  /** Barn er ordnet fra venstre til høyre */
  PositionIterator children(Position p);
  boolean isInternal(Position v);
  boolean isExternal(Position v);
  boolean isRoot(Position v);
  PositionIterator siblings(Position p);
}
```

```
public interface InspectableBinaryTree
  extends InspectableTree {
  Position leftChild(Position);
  Position rightChild(Position);
  Position sibling(Position); }
```

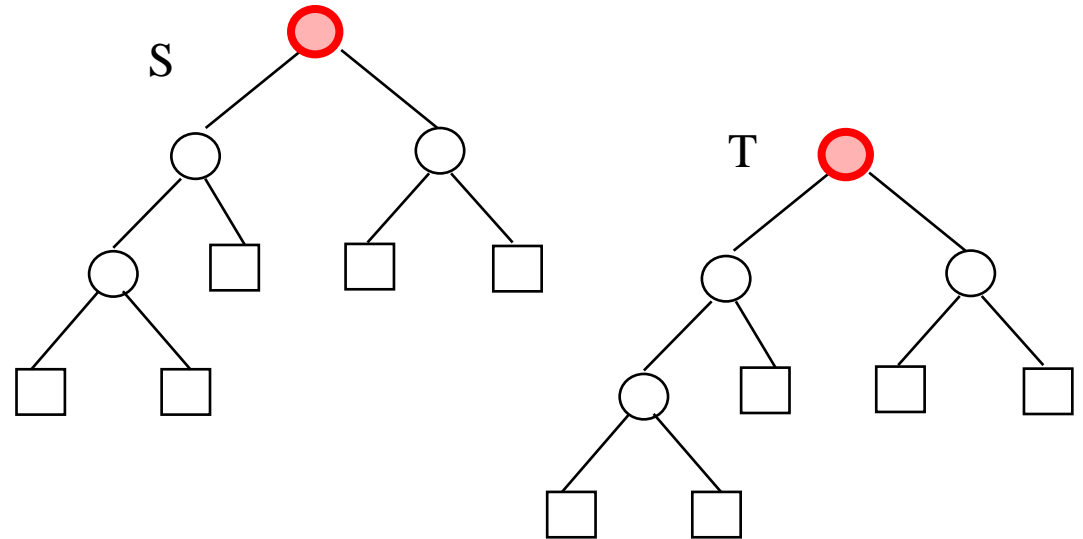
BinaryTree: likhet vs. isomorfisme



```
boolean iso (InspectableBinaryTree S, T)  
{ return iso(S.root, T.root(), S,T) ; }
```

```
boolean iso  
    (Position r, Position p, InspectableBinaryTree S, T)  
{ if (S.isExternal(r) && T.isExternal(p))  
    return true;  
else if (S.isInternal(r) && T.isInternal(p) )  
    return (  
        iso(S.leftChild(r), T.leftChild(p), S, T) &&  
        iso(S.rightChild(r), T.rightChild(p), S, T) );  
else return false;  
}
```

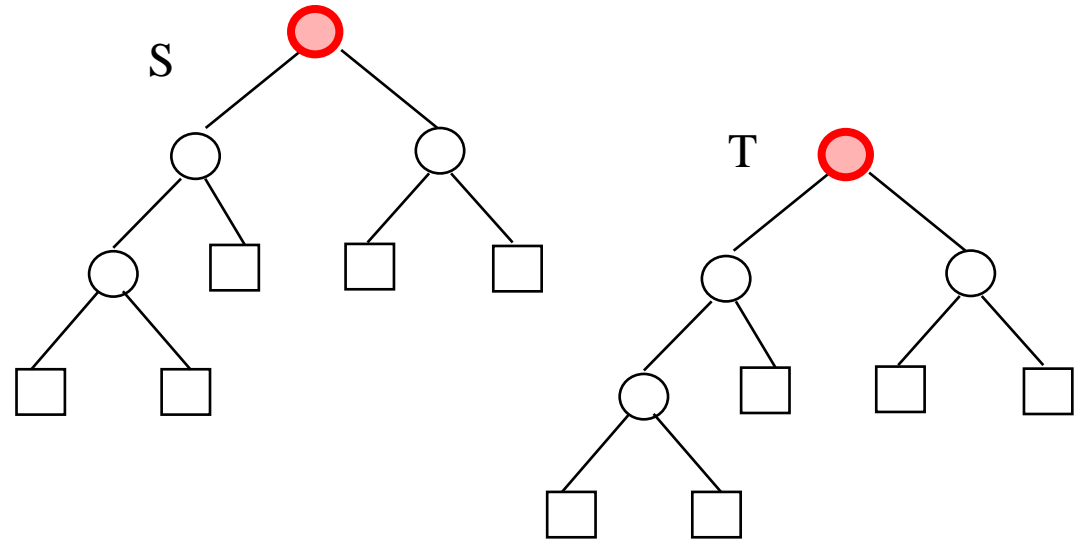
BinaryTree: likhet vs. isomorfisme



```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root(), T.root(), S,T) ; }
```

```
boolean iso
    (Position r, Position p, InspectableBinaryTree S, T)
{ if (S.isExternal(r) && T.isExternal(p))
    return true;
else if (S.isInternal(r) && T.isInternal(p) )
    return (
        iso(S.leftChild(r), T.leftChild(p), S, T) &&
        iso(S.rightChild(r), T.rightChild(p), S, T) );
else return false;
}
```

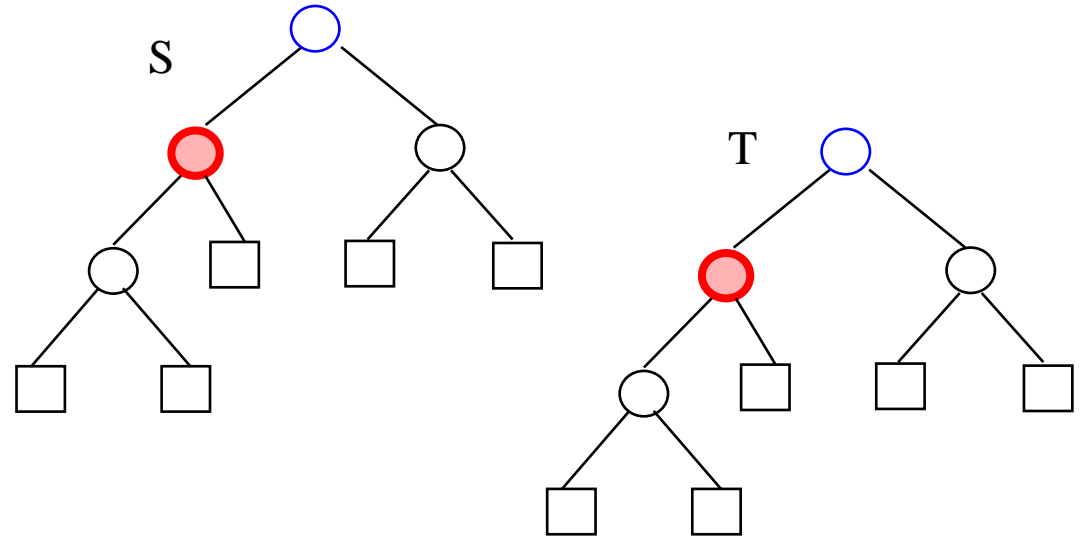
BinaryTree: likhet vs. isomorfisme



```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root, T.root(), S,T) ; }
```

```
boolean iso
    (Position r, Position p, InspectableBinaryTree S, T)
{ if (S.isExternal(r) && T.isExternal(p))
    return true;
  else if (S.isInternal(r) && T.isInternal(p) )
    return (
        iso(S.leftChild(r), T.leftChild(p), S, T) &&
        iso(S.rightChild(r), T.rightChild(p), S, T) );
  else return false;
}
```

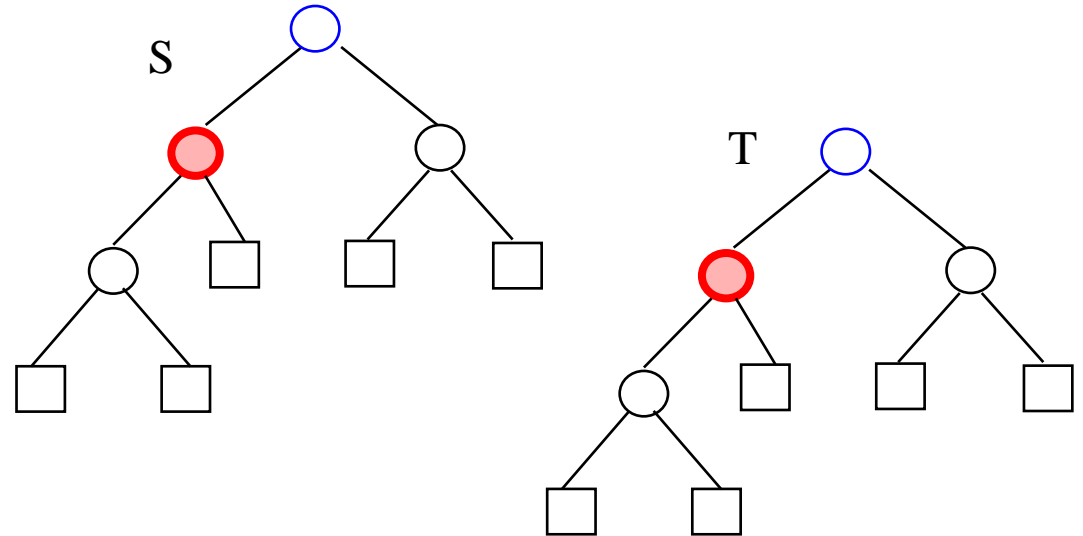

BinaryTree: likhet vs. isomorfisme



```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root, T.root(), S,T) ; }
```

```
boolean iso
    (Position r, Position p, InspectableBinaryTree S, T)
{ if (S.isExternal(r) && T.isExternal(p))
    return true;
else if (S.isInternal(r) && T.isInternal(p) )
    return (
        iso(S.leftChild(r), T.leftChild(p), S, T) &&
        iso(S.rightChild(r), T.rightChild(p), S, T) );
else return false;
}
```

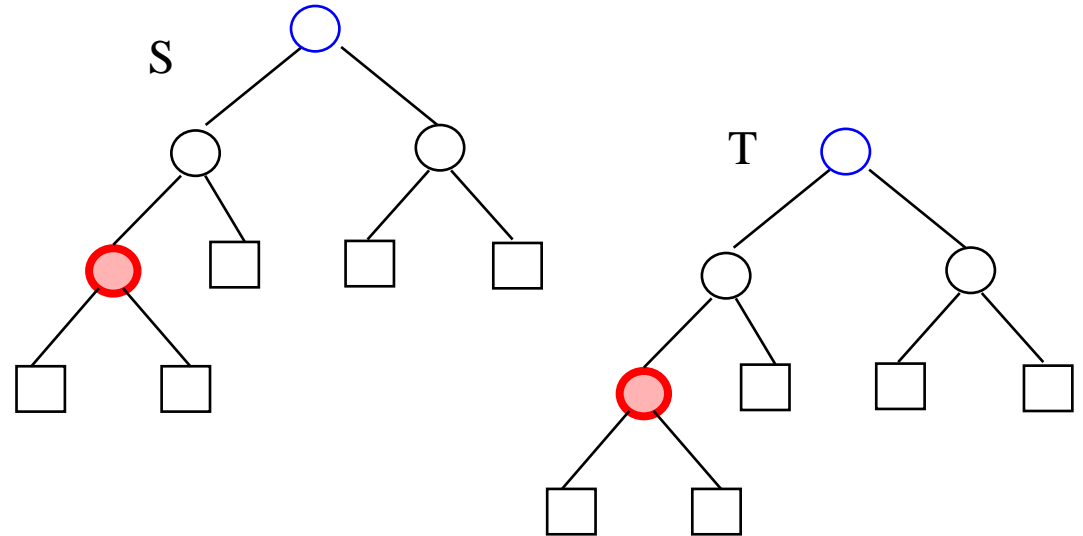
BinaryTree: likhet vs. isomorfisme



```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root, T.root(), S,T) ; }
```

```
boolean iso
    (Position r, Position p, InspectableBinaryTree S, T)
{ if (S.isExternal(r) && T.isExternal(p))
    return true;
  else if (S.isInternal(r) && T.isInternal(p) )
    return (
        iso(S.leftChild(r), T.leftChild(p), S, T) &&
        iso(S.rightChild(r), T.rightChild(p), S, T) );
  else return false;
}
```

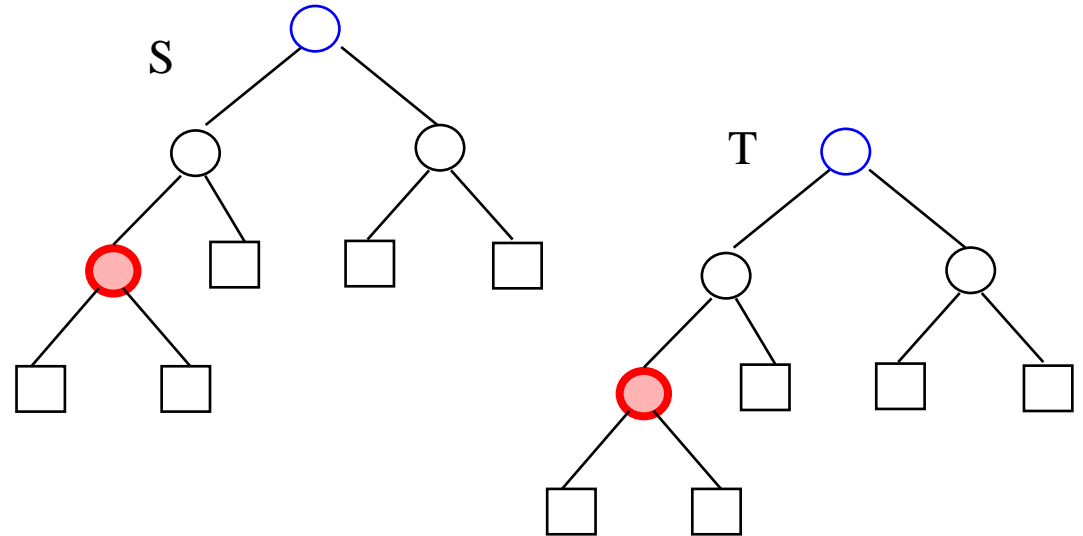
BinaryTree: likhet vs. isomorfisme



```
boolean iso (InspectableBinaryTree S, T)  
{ return iso(S.root, T.root(), S,T) ; }
```

```
boolean iso  
    (Position r, Position p, InspectableBinaryTree S, T)  
{ if (S.isExternal(r) && T.isExternal(p))  
    return true;  
else if (S.isInternal(r) && T.isInternal(p) )  
    return (  
        iso(S.leftChild(r), T.leftChild(p), S, T) &&  
        iso(S.rightChild(r), T.rightChild(p), S, T) );  
else return false;  
}
```

BinaryTree: likhet vs. isomorfisme

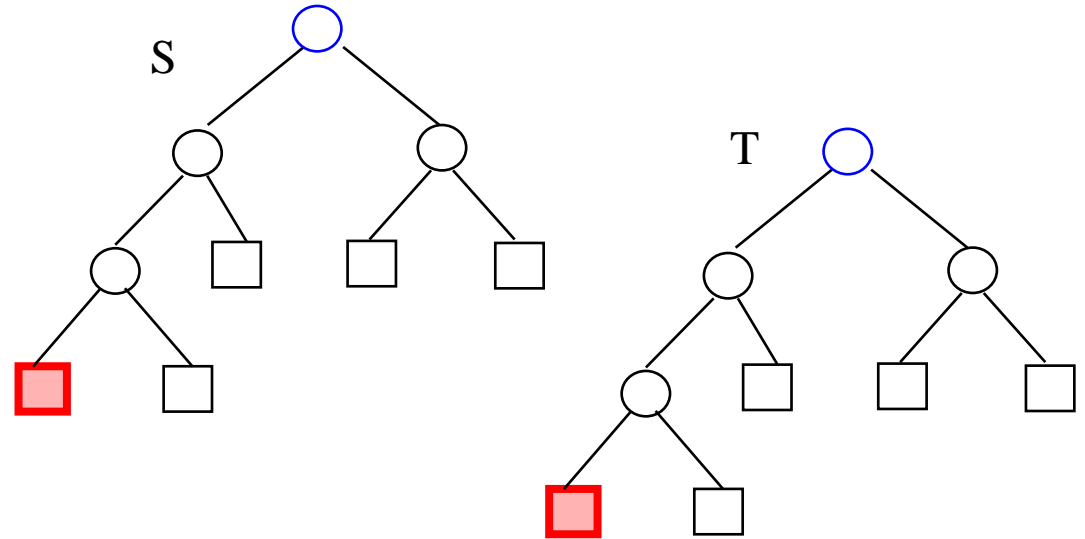


```
boolean iso (InspectableBinaryTree S, T)  
{ return iso(S.root, T.root(), S,T) ; }
```

```
boolean iso  
    (Position r, Position p, InspectableBinaryTree S, T)  
{ if (S.isExternal(r) && T.isExternal(p))  
    return true;  
  else if (S.isInternal(r) && T.isInternal(p) )  
    return (  
      iso(S.leftChild(r), T.leftChild(p), S, T) &&  
      iso(S.rightChild(r), T.rightChild(p), S, T) );  
  else return false;  
}
```

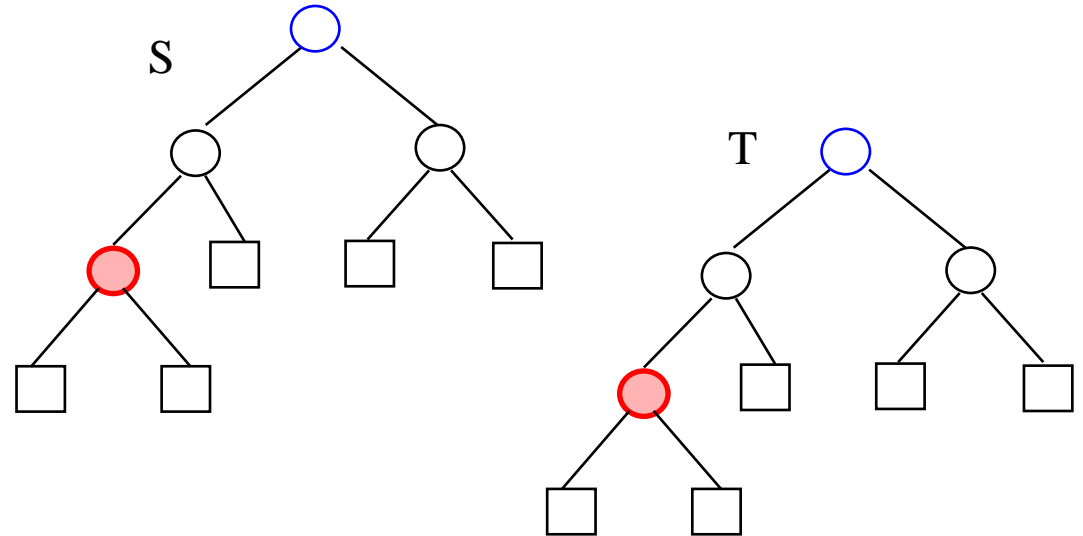
BinaryTree: likhet vs. isomorfisme

```
boolean iso (InspectableBinaryTree S, T)  
{ return iso(S.root, T.root(), S,T) ; }
```



```
boolean iso  
    (Position r, Position p, InspectableBinaryTree S, T)  
{ if (S.isExternal(r) && T.isExternal(p))  
    return true;  
  else if (S.isInternal(r) && T.isInternal(p) )  
    return (  
        iso(S.leftChild(r), T.leftChild(p), S, T) &&  
        iso(S.rightChild(r), T.rightChild(p), S, T) );  
  else return false;  
}
```

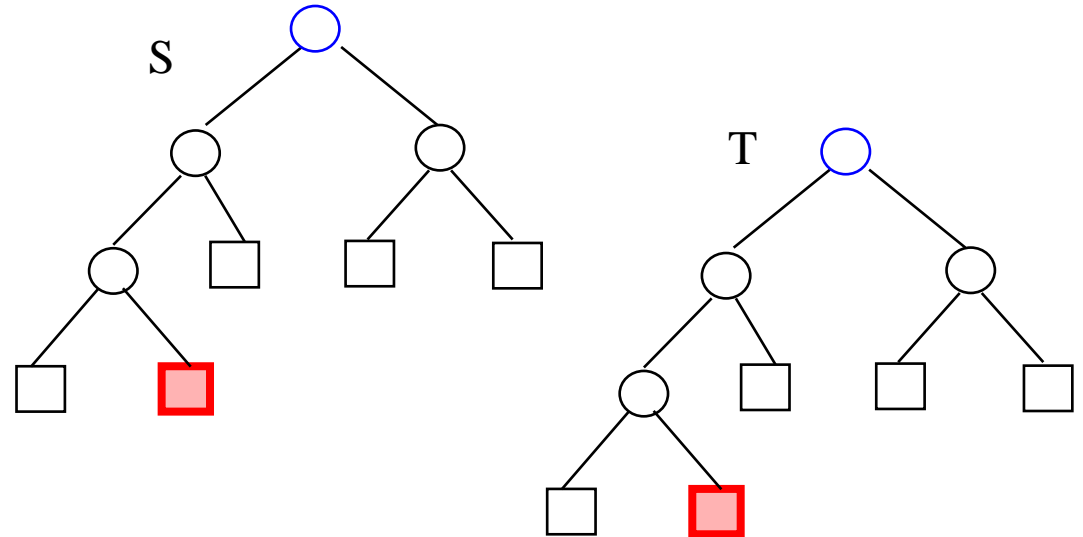
BinaryTree: likhet vs. isomorfisme



```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root, T.root(), S,T) ; }
```

```
boolean iso
    (Position r, Position p, InspectableBinaryTree S, T)
{ if (S.isExternal(r) && T.isExternal(p))
    return true;
  else if (S.isInternal(r) && T.isInternal(p) )
    return (
        iso(S.leftChild(r), T.leftChild(p), S, T) &&
        iso(S.rightChild(r), T.rightChild(p), S, T) );
  else return false;
}
```

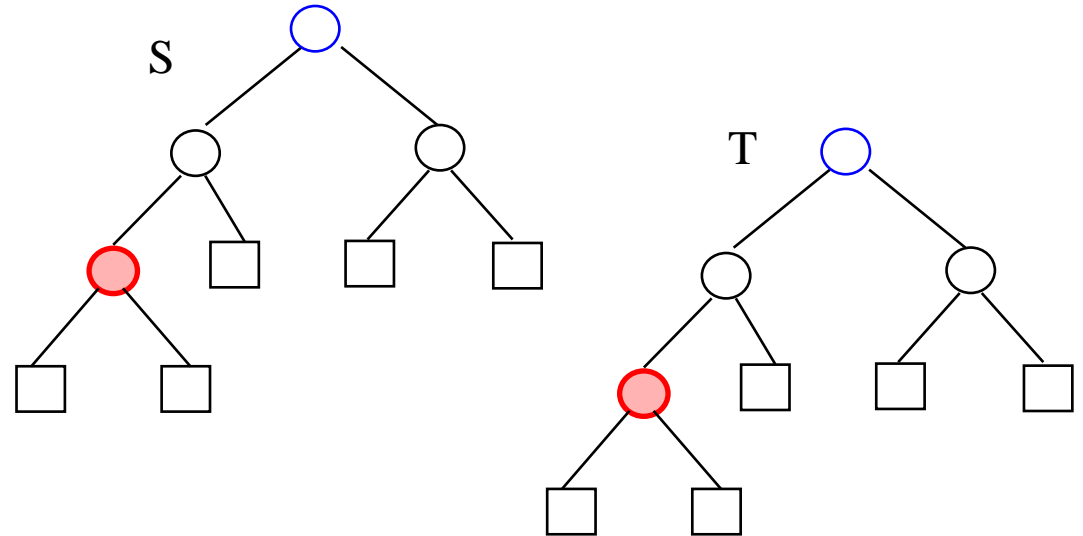
BinaryTree: likhet vs. isomorfisme



```
boolean iso (InspectableBinaryTree S, T)  
{ return iso(S.root, T.root(), S,T) ; }
```

```
boolean iso  
    (Position r, Position p, InspectableBinaryTree S, T)  
{ if (S.isExternal(r) && T.isExternal(p))  
    return true;  
else if (S.isInternal(r) && T.isInternal(p) )  
    return (  
        iso(S.leftChild(r), T.leftChild(p), S, T) &&  
        iso(S.rightChild(r), T.rightChild(p), S, T) );  
else return false;  
}
```

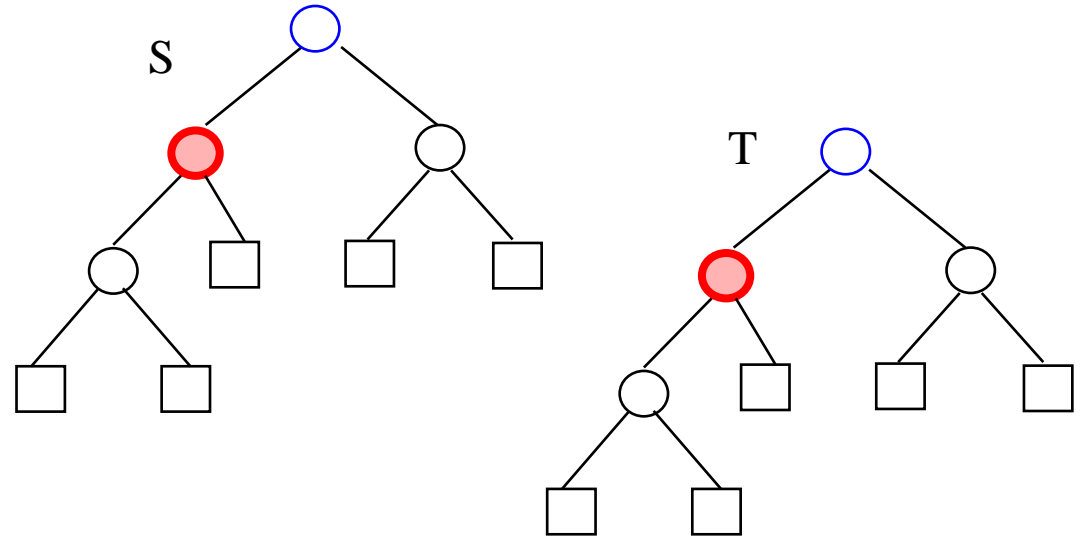
BinaryTree: likhet vs. isomorfisme



```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root, T.root(), S,T) ; }
```

```
boolean iso
    (Position r, Position p, InspectableBinaryTree S, T)
{ if (S.isExternal(r) && T.isExternal(p))
    return true;
else if (S.isInternal(r) && T.isInternal(p) )
    return (
        iso(S.leftChild(r), T.leftChild(p), S, T) &&
        iso(S.rightChild(r), T.rightChild(p), S, T) );
else return false;
}
```

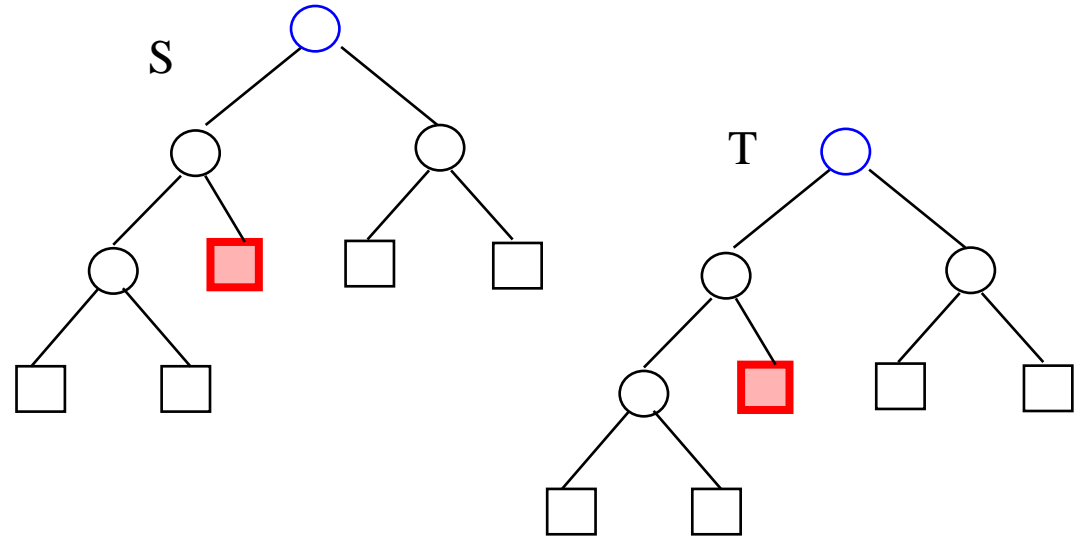

BinaryTree: likhet vs. isomorfisme



```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root, T.root(), S,T) ; }
```

```
boolean iso
    (Position r, Position p, InspectableBinaryTree S, T)
{ if (S.isExternal(r) && T.isExternal(p))
    return true;
  else if (S.isInternal(r) && T.isInternal(p) )
    return (
        iso(S.leftChild(r), T.leftChild(p), S, T) &&
        iso(S.rightChild(r), T.rightChild(p), S, T) );
  else return false;
}
```

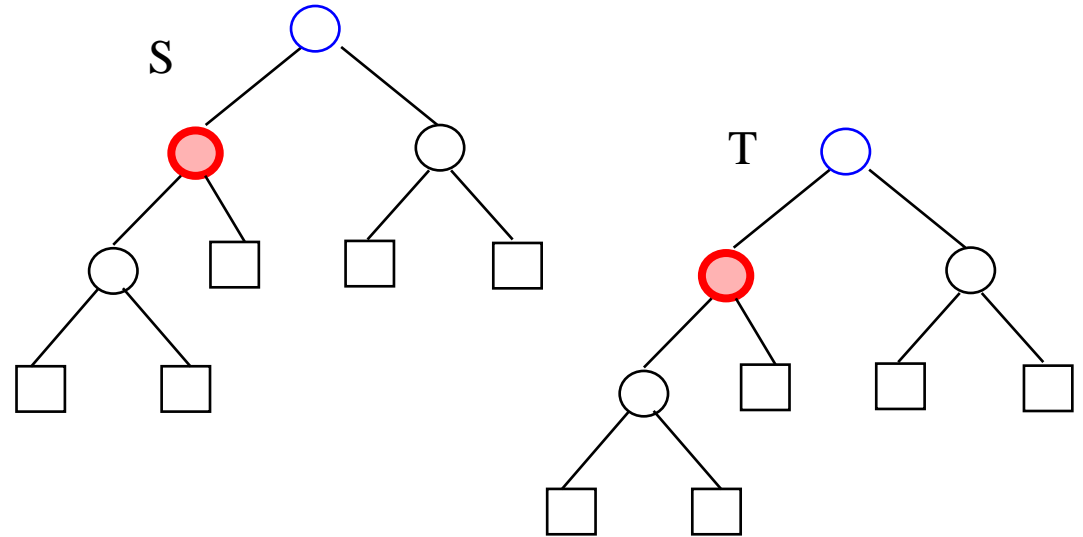
BinaryTree: likhet vs. isomorfisme



```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root, T.root(), S,T) ; }
```

```
boolean iso
    (Position r, Position p, InspectableBinaryTree S, T)
{ if (S.isExternal(r) && T.isExternal(p))
    return true;
else if (S.isInternal(r) && T.isInternal(p) )
    return (
        iso(S.leftChild(r), T.leftChild(p), S, T) &&
        iso(S.rightChild(r), T.rightChild(p), S, T) );
else return false;
}
```

BinaryTree: likhet vs. isomorfisme

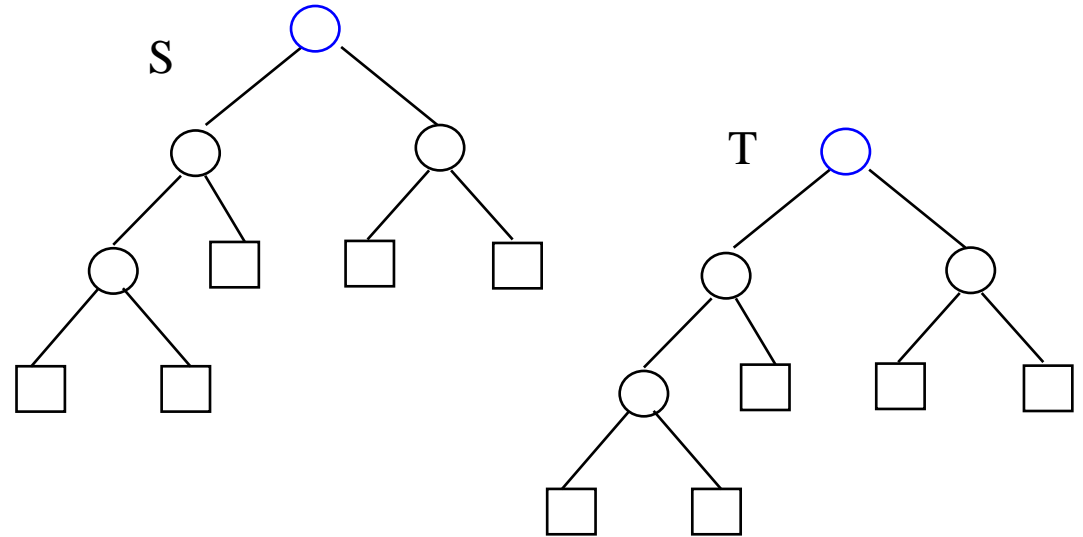


```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root, T.root(), S,T) ; }
```

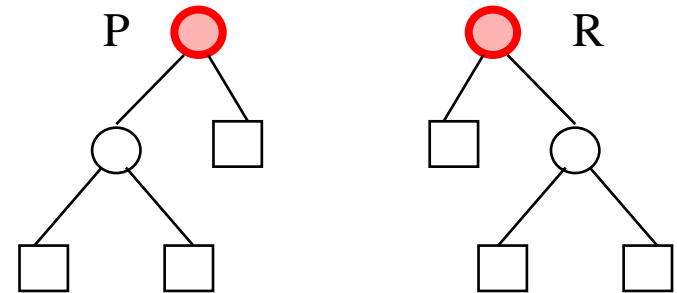
```
boolean iso
    (Position r, Position p, InspectableBinaryTree S, T)
{ if (S.isExternal(r) && T.isExternal(p))
    return true;
else if (S.isInternal(r) && T.isInternal(p) )
    return (
        iso(S.leftChild(r), T.leftChild(p), S, T) &&
        iso(S.rightChild(r), T.rightChild(p), S, T) );
else return false;
}
```

BinaryTree: likhet vs. isomorfisme

```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root, T.root(), S,T) ; }
```

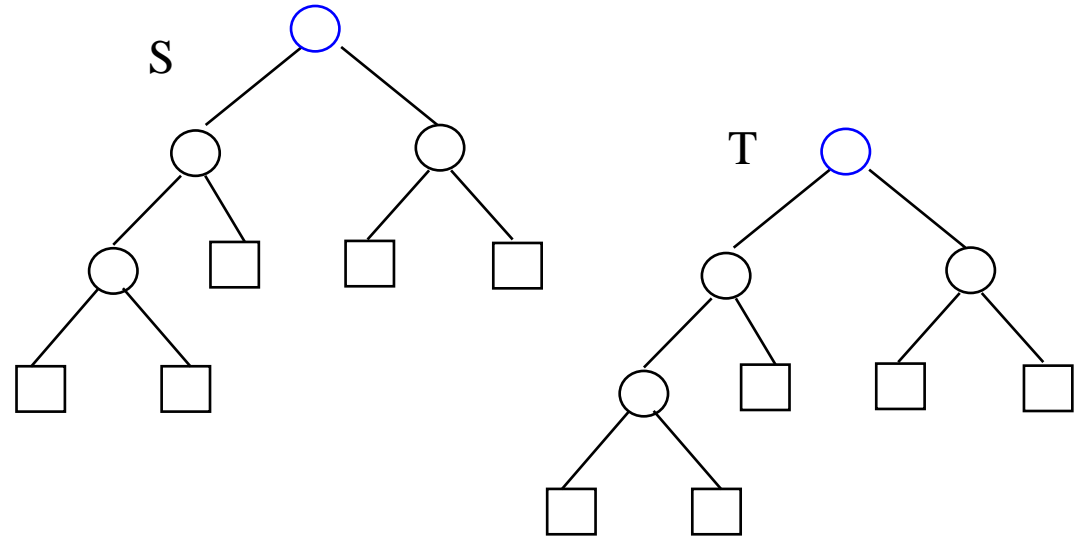


```
boolean iso
    (Position r, Position p, InspectableBinaryTree S, T)
{ if (S.isExternal(r) && T.isExternal(p))
    return true;
  else if (S.isInternal(r) && T.isInternal(p) )
    return (
        iso(S.leftChild(r), T.leftChild(p), S, T) &&
        iso(S.rightChild(r), T.rightChild(p), S, T) );
  else return false;
}
```

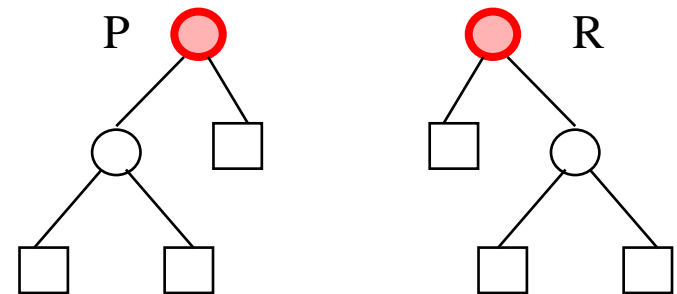


BinaryTree: likhet vs. isomorfisme

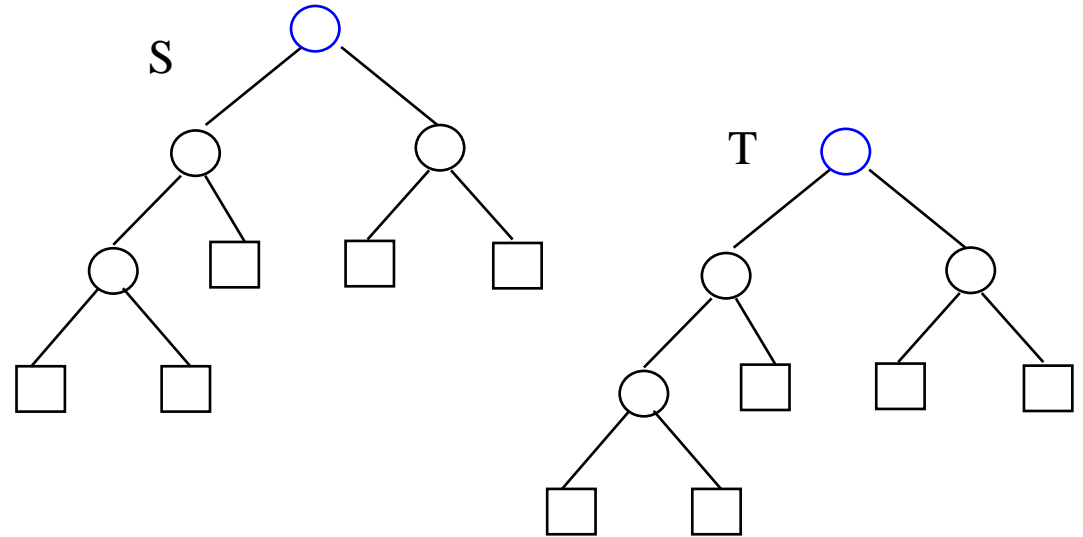
```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root, T.root(), S,T) ; }
```



```
boolean iso
    (Position r, Position p, InspectableBinaryTree S, T)
{ if (S.isExternal(r) && T.isExternal(p))
    return true;
  else if (S.isInternal(r) && T.isInternal(p) )
    return (
        iso(S.leftChild(r), T.leftChild(p), S, T) &&
        iso(S.rightChild(r), T.rightChild(p), S, T) );
  else return false;
}
```

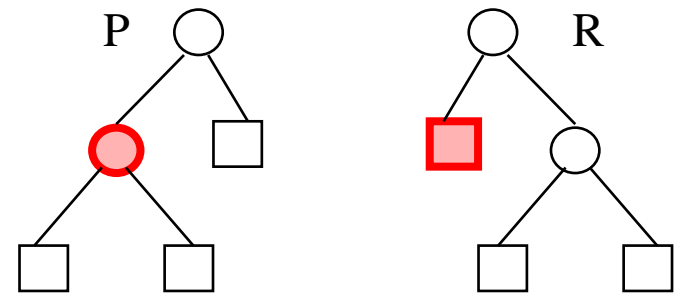


BinaryTree: likhet vs. isomorfisme

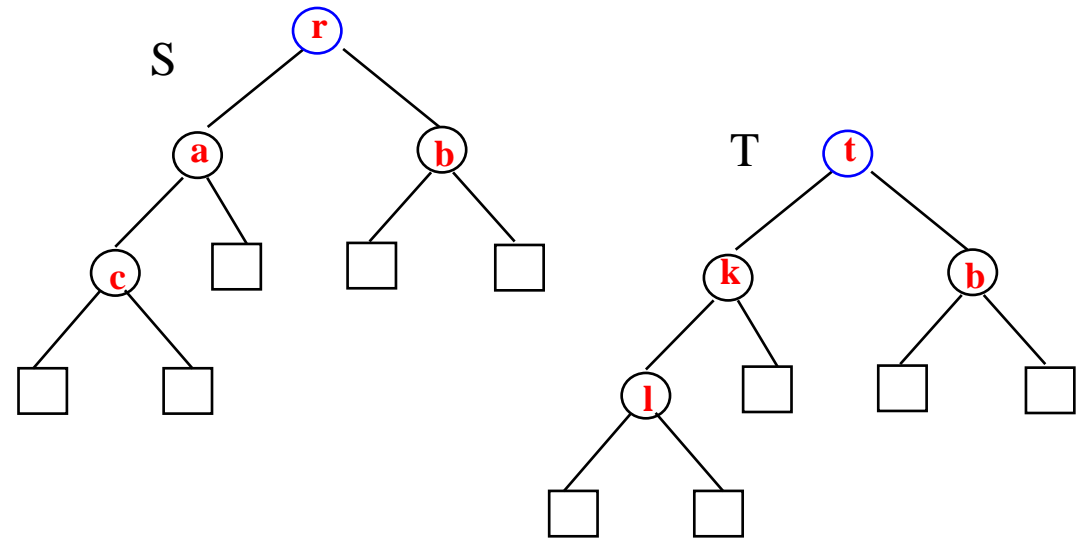


```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root, T.root(), S,T) ; }
```

```
boolean iso
    (Position r, Position p, InspectableBinaryTree S, T)
{ if (S.isExternal(r) && T.isExternal(p))
    return true;
else if (S.isInternal(r) && T.isInternal(p) )
    return (
        iso(S.leftChild(r), T.leftChild(p), S, T) &&
        iso(S.rightChild(r), T.rightChild(p), S, T) );
else return false;
}
```

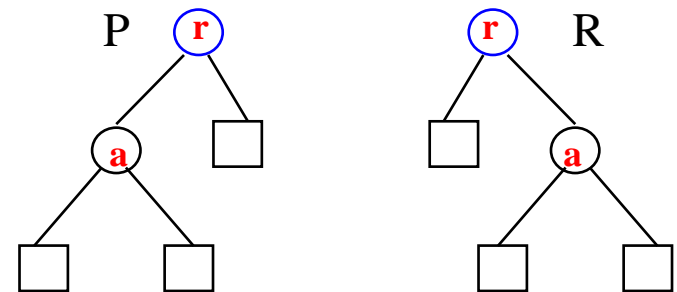


BinaryTree: likhet vs. isomorfisme

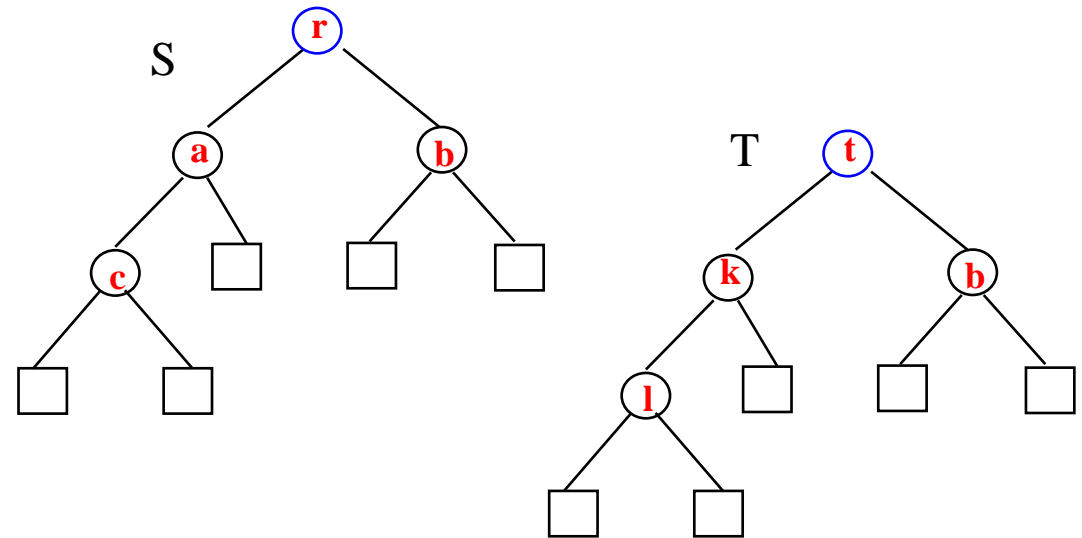


```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root, T.root(), S,T) ; }
```

```
boolean iso
    (Position r, Position p, InspectableBinaryTree S, T)
{ if (S.isExternal(r) && T.isExternal(p))
    return true;
  else if (S.isInternal(r) && T.isInternal(p) )
    return (
        iso(S.leftChild(r), T.leftChild(p), S, T) &&
        iso(S.rightChild(r), T.rightChild(p), S, T) );
  else return false;
}
```



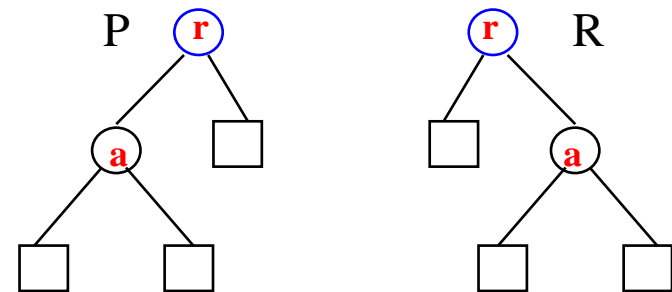
BinaryTree: likhet vs. isomorfisme



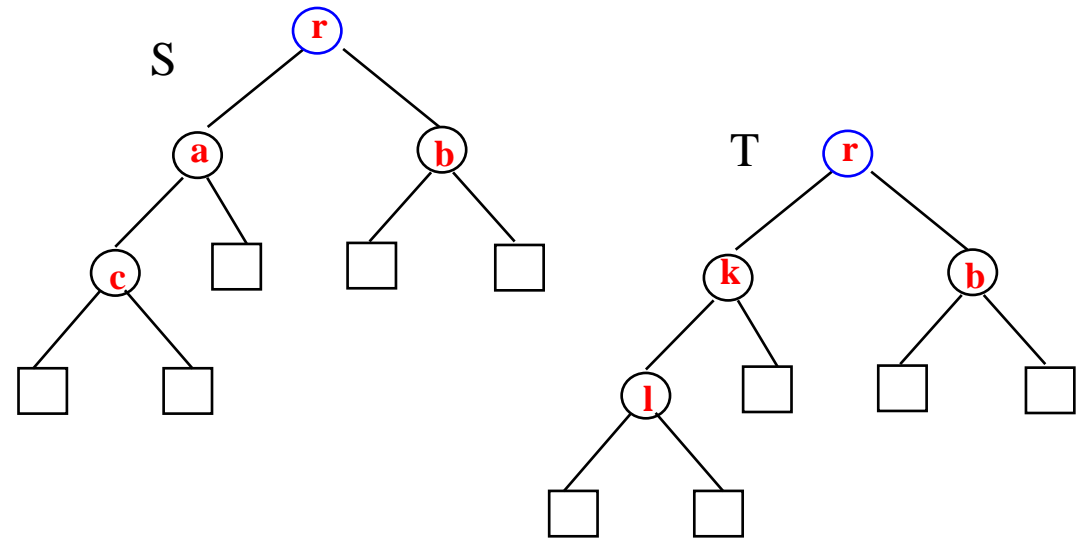
```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root, T.root(), S,T) ; }
```

```
boolean equals (InspectableBinaryTree S, T)
{ return equ(S.root, T.root(), S, T) ; }
```

```
boolean iso // equ
    (Position r, Position p, InspectableBinaryTree S, T)
{ if (S.isExternal(r) && T.isExternal(p))
    return true; // r.element().equals(p.element());
  else if (S.isInternal(r) && T.isInternal(p) )
    return ( // r.element().equals(p.element()) &&
             iso(S.leftChild(r), T.leftChild(p), S, T) && // equ
             iso(S.rightChild(r), T.rightChild(p), S, T) ); // equ
  else return false;
}
```



BinaryTree: likhet vs. isomorfisme

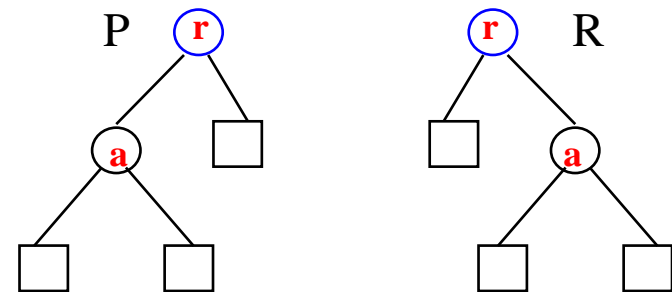


```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root, T.root(), S,T) ; }
```

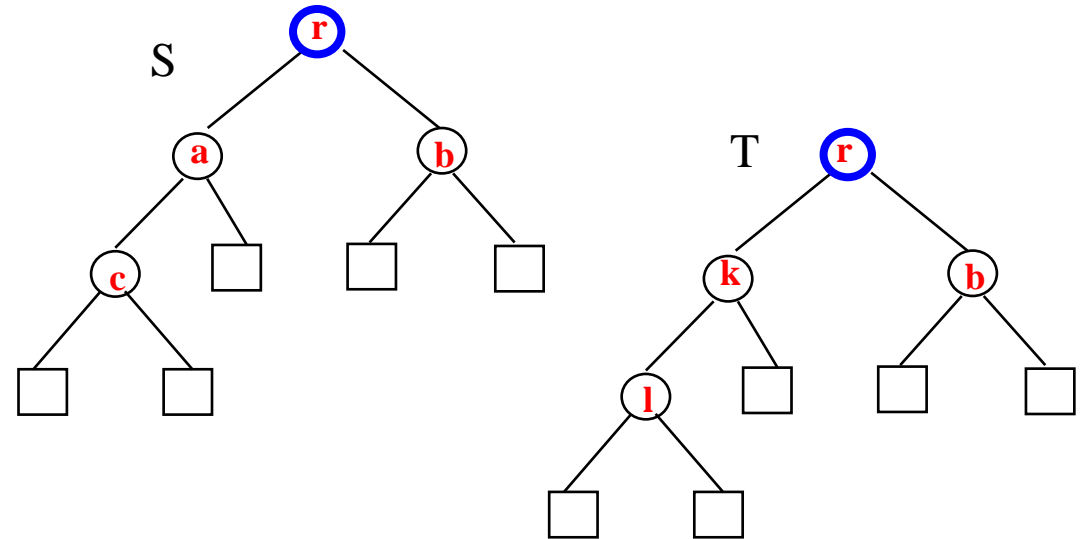
```
boolean equals (InspectableBinaryTree S, T)
{ return equ(S.root, T.root(), S, T) ; }
```

```
boolean equ
(Position r, Position p, InspectableBinaryTree S, T)
```

```
{ if (S.isExternal(r) && T.isExternal(p))
  return r.element().equals(p.element());
else if (S.isInternal(r) && T.isInternal(p) )
  return ( r.element().equals(p.element()) &&
    equ(S.leftChild(r), T.leftChild(p), S, T) &&
    equ(S.rightChild(r), T.rightChild(p), S, T) );
else return false;
}
```



BinaryTree: likhet vs. isomorfisme

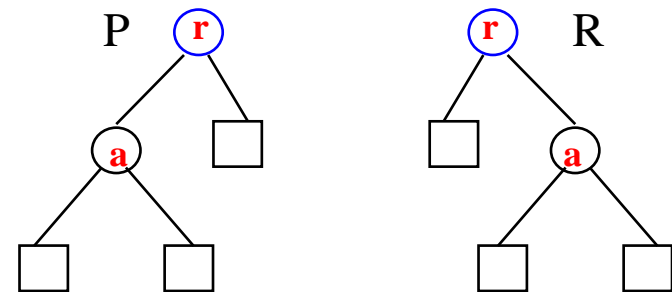


```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root, T.root(), S,T) ; }
```

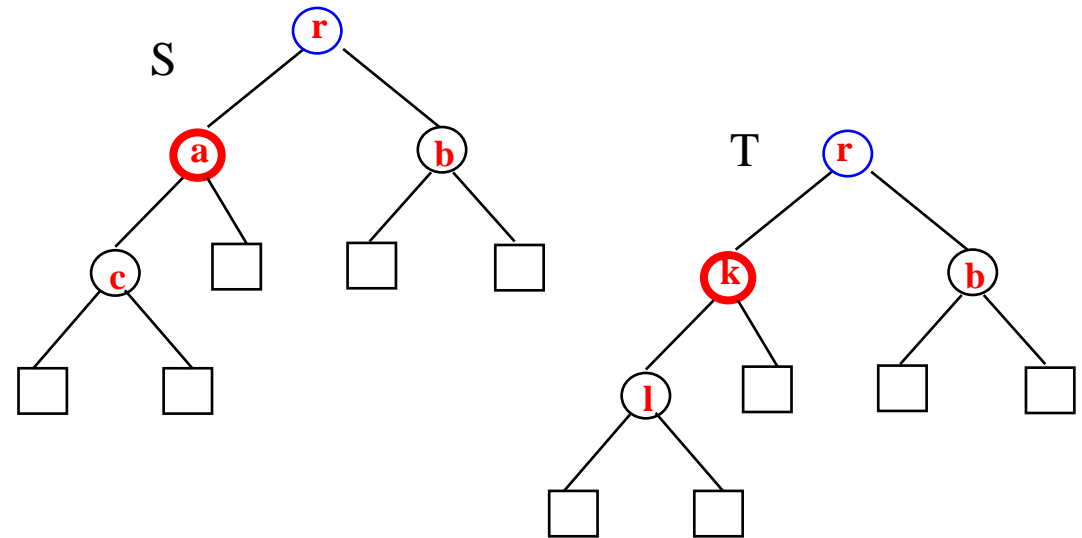
```
boolean equals (InspectableBinaryTree S, T)
{ return equ(S.root, T.root(), S, T) ; }
```

```
boolean equ
(Position r, Position p, InspectableBinaryTree S, T)
```

```
{ if (S.isExternal(r) && T.isExternal(p))
  return r.element().equals(p.element());
  else if (S.isInternal(r) && T.isInternal(p) )
    return ( r.element().equals(p.element()) &&
             equ(S.leftChild(r), T.leftChild(p), S, T) &&
             equ(S.rightChild(r), T.rightChild(p), S, T) );
  else return false;
}
```



BinaryTree: likhet vs. isomorfisme

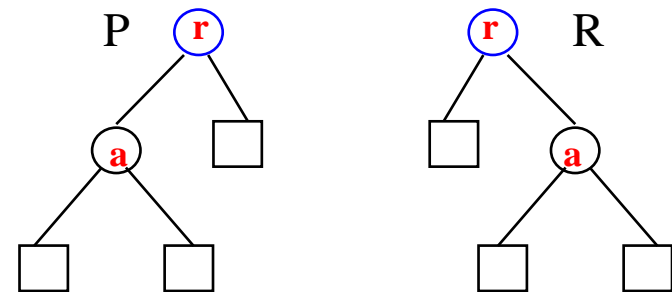


```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root, T.root(), S,T) ; }
```

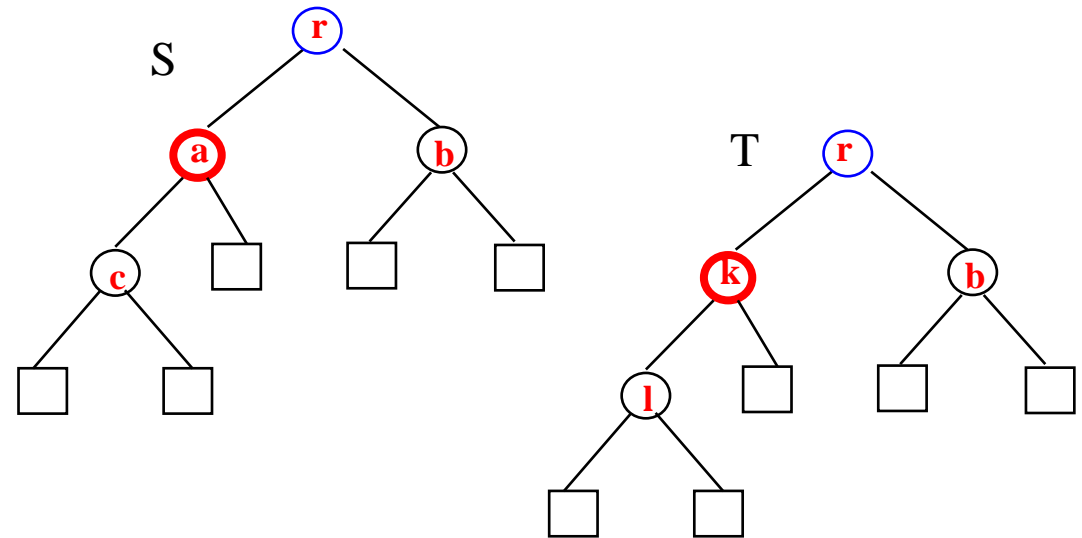
```
boolean equals (InspectableBinaryTree S, T)
{ return equ(S.root, T.root(), S, T) ; }
```

```
boolean equ
(Position r, Position p, InspectableBinaryTree S, T)
```

```
{ if (S.isExternal(r) && T.isExternal(p))
  return r.element().equals(p.element());
  else if (S.isInternal(r) && T.isInternal(p) )
  return ( r.element().equals(p.element()) &&
    equ(S.leftChild(r), T.leftChild(p), S, T) &&
    equ(S.rightChild(r), T.rightChild(p), S, T) );
  else return false;
}
```



BinaryTree: likhet vs. isomorfisme

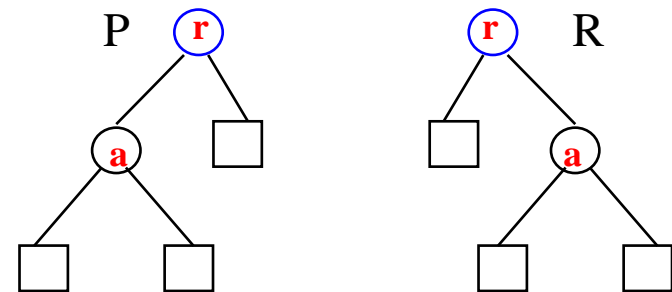


```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root, T.root(), S,T) ; }
```

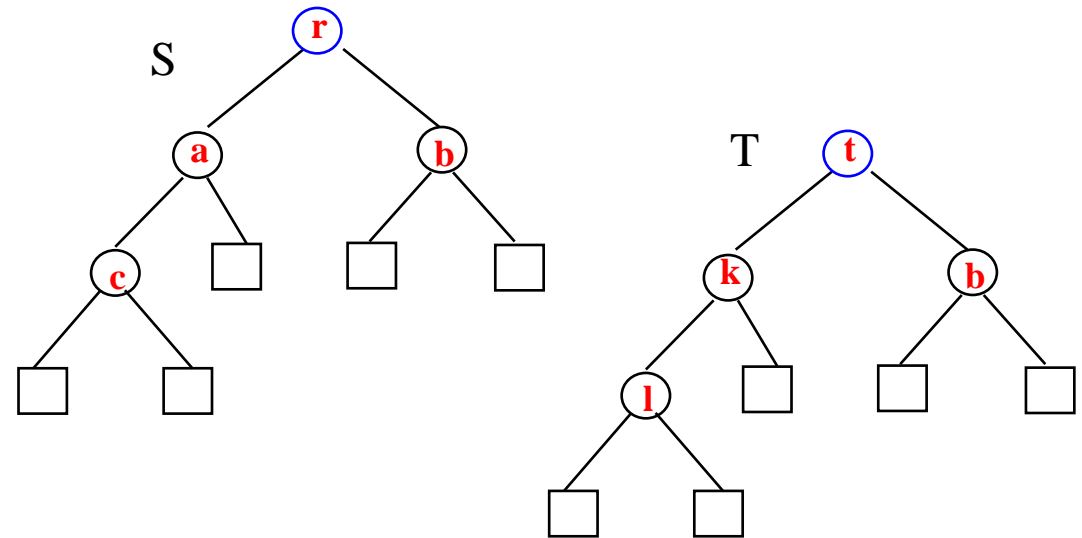
```
boolean equals (InspectableBinaryTree S, T)
{ return equ(S.root, T.root(), S, T) ; }
```

```
boolean equ
(Position r, Position p, InspectableBinaryTree S, T)
```

```
{ if (S.isExternal(r) && T.isExternal(p))
  return r.element().equals(p.element());
  else if (S.isInternal(r) && T.isInternal(p) )
  return ( r.element().equals(p.element()) &&
    equ(S.leftChild(r), T.leftChild(p), S, T) &&
    equ(S.rightChild(r), T.rightChild(p), S, T) );
  else return false;
}
```



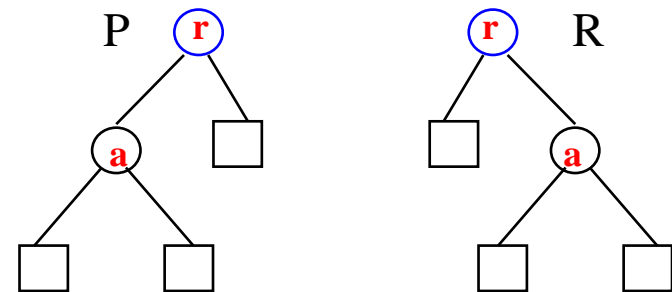
BinaryTree: likhet vs. isomorfisme



```
boolean iso (InspectableBinaryTree S, T)
{ return iso(S.root, T.root(), S,T) ; }
```

```
boolean equals (InspectableBinaryTree S, T)
{ return equ(S.root, T.root(), S, T) ; }
```

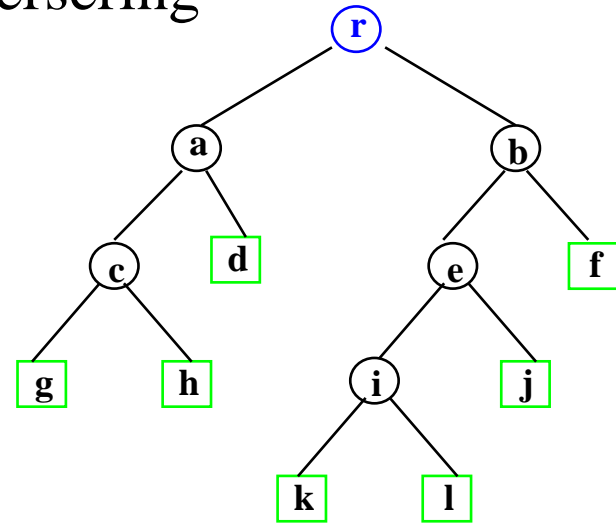
```
boolean iso // equ
    (Position r, Position p, InspectableBinaryTree S, T)
{ if (S.isExternal(r) && T.isExternal(p))
    return true; // r.element().equals(p.element());
else if (S.isInternal(r) && T.isInternal(p) )
    return ( // r.element().equals(p.element()) &&
            iso(S.leftChild(r), T.leftChild(p), S, T) && // equ
            iso(S.rightChild(r), T.rightChild(p), S, T) ); // equ
else return false;
}
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
  if (isExternal(v)) return 0
  else // 1+max{ height(p): p i children(v) }
    max=0
    for hver p i children(v)
      h=height(p); if (h>max) max=h
    return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
```

```
  if (isExternal(v)) return 0
```

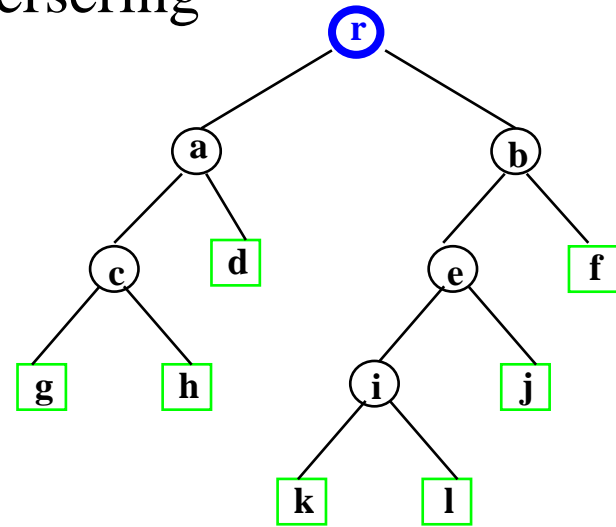
```
  else // 1+max{ height(p): p i children(v) }
```

```
    max=0
```

```
    for hver p i children(v)
```

```
      h=height(p); if (h>max) max=h
```

```
  return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
```

```
  if (isExternal(v)) return 0
```

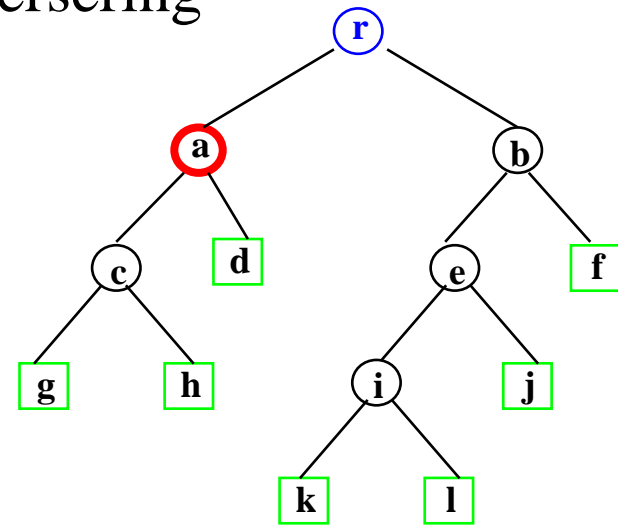
```
  else // 1+max{ height(p): p i children(v) }
```

```
    max=0
```

```
    for hver p i children(v)
```

```
      h=height(p); if (h>max) max=h
```

```
  return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
```

```
  if (isExternal(v)) return 0
```

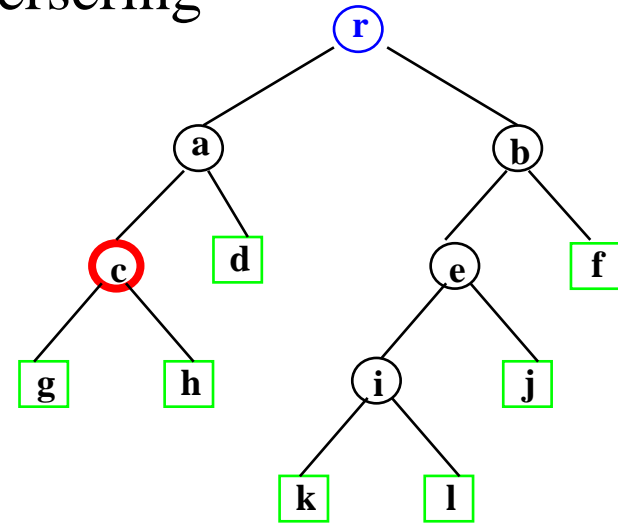
```
  else // 1+max{ height(p): p i children(v) }
```

```
    max=0
```

```
    for hver p i children(v)
```

```
      h=height(p); if (h>max) max=h
```

```
  return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
```

```
  if (isExternal(v)) return 0
```

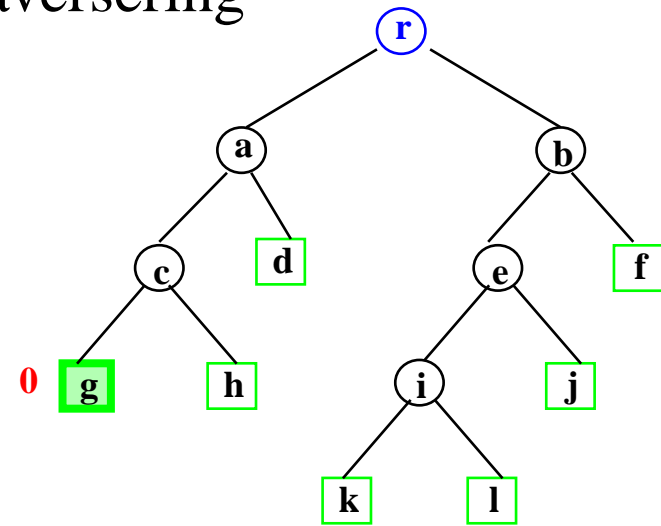
```
  else // 1+max{ height(p): p i children(v) }
```

```
    max=0
```

```
    for hver p i children(v)
```

```
      h=height(p); if (h>max) max=h
```

```
  return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
```

```
  if (isExternal(v)) return 0
```

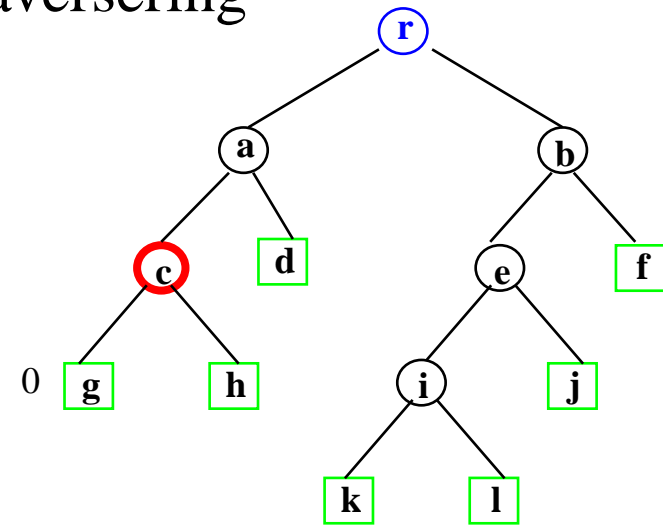
```
  else // 1+max{ height(p): p i children(v) }
```

```
    max=0
```

```
    for hver p i children(v)
```

```
      h=height(p); if (h>max) max=h
```

```
  return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
```

```
  if (isExternal(v)) return 0
```

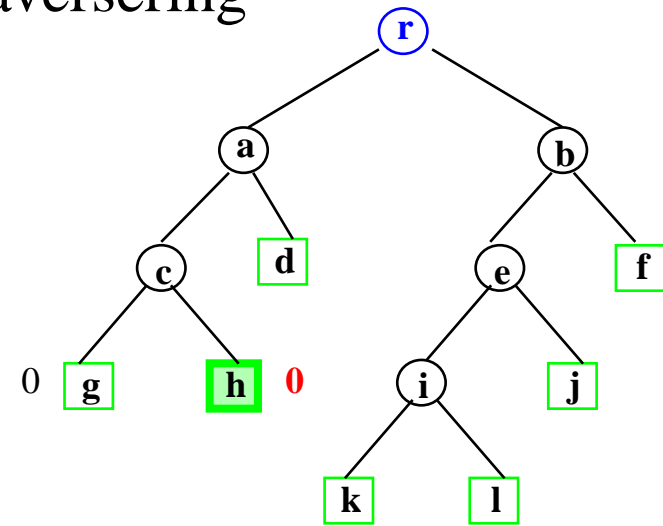
```
  else // 1+max{ height(p): p i children(v) }
```

```
    max=0
```

```
    for hver p i children(v)
```

```
      h=height(p); if (h>max) max=h
```

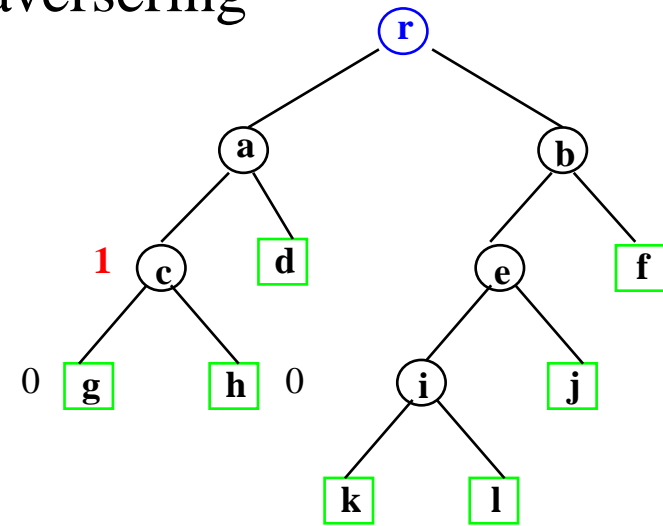
```
  return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
  if (isExternal(v)) return 0
  else // 1+max{ height(p): p i children(v) }
    max=0
    for hver p i children(v)
      h=height(p); if (h>max) max=h
  return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
```

```
  if (isExternal(v)) return 0
```

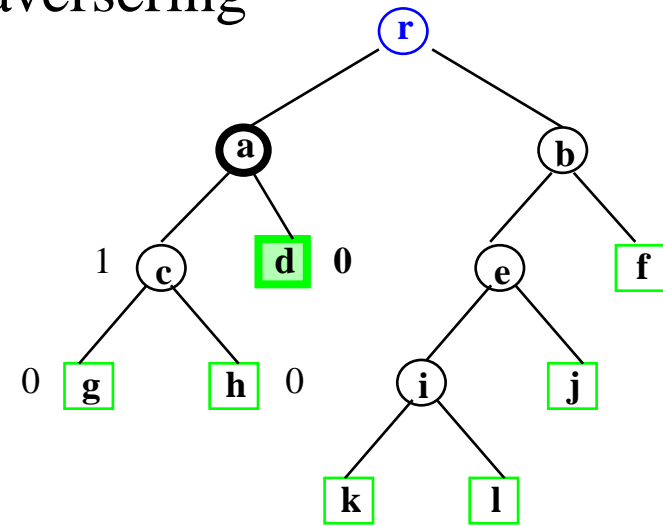
```
  else // 1+max{ height(p): p i children(v) }
```

```
    max=0
```

```
    for hver p i children(v)
```

```
      h=height(p); if (h>max) max=h
```

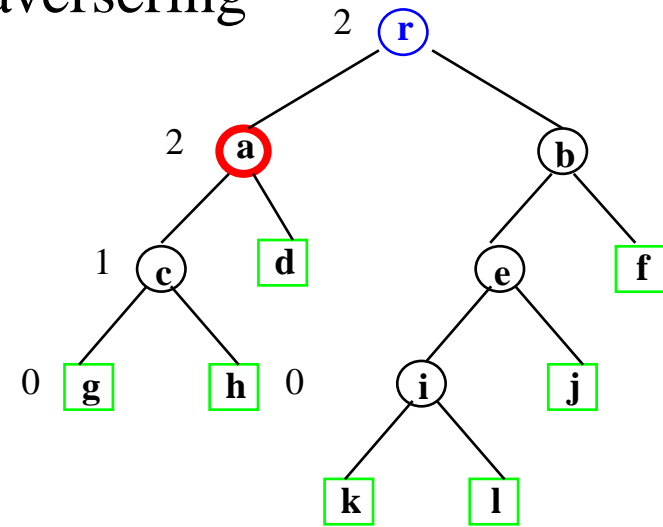
```
  return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

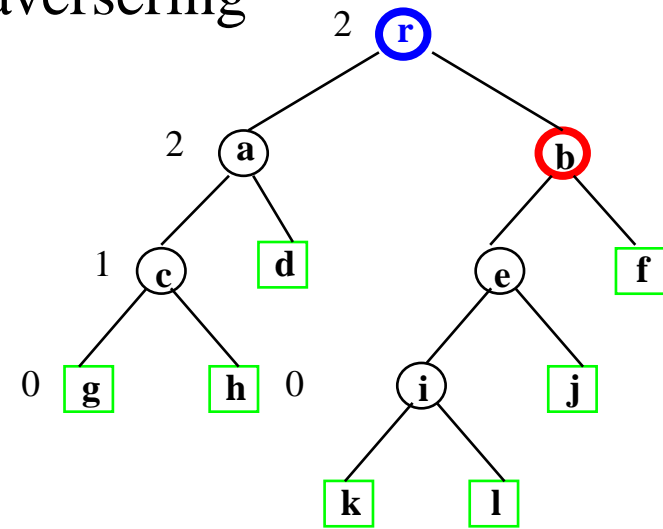
```
int height(Position v)
  if (isExternal(v)) return 0
  else // 1+max{ height(p): p i children(v) }
    max=0
    for hver p i children(v)
      h=height(p); if (h>max) max=h
  return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

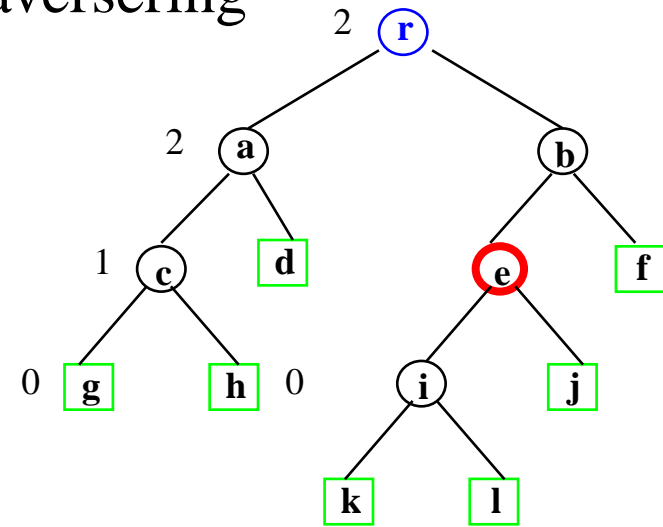
```
int height(Position v)
  if (isExternal(v)) return 0
  else // 1+max{ height(p): p i children(v) }
    max=0
    for hver p i children(v)
      h=height(p); if (h>max) max=h
    return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

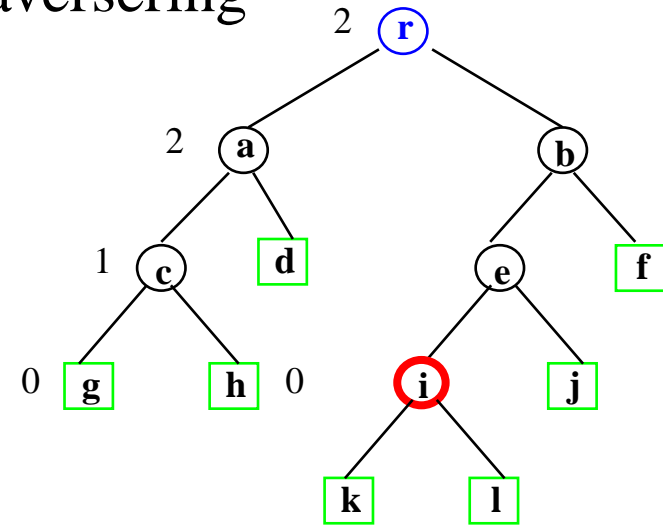
```
int height(Position v)
  if (isExternal(v)) return 0
  else // 1+max{ height(p): p i children(v) }
    max=0
    for hver p i children(v)
      h=height(p); if (h>max) max=h
    return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
  if (isExternal(v)) return 0
  else // 1+max{ height(p): p i children(v) }
    max=0
    for hver p i children(v)
      h=height(p); if (h>max) max=h
    return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
```

```
  if (isExternal(v)) return 0
```

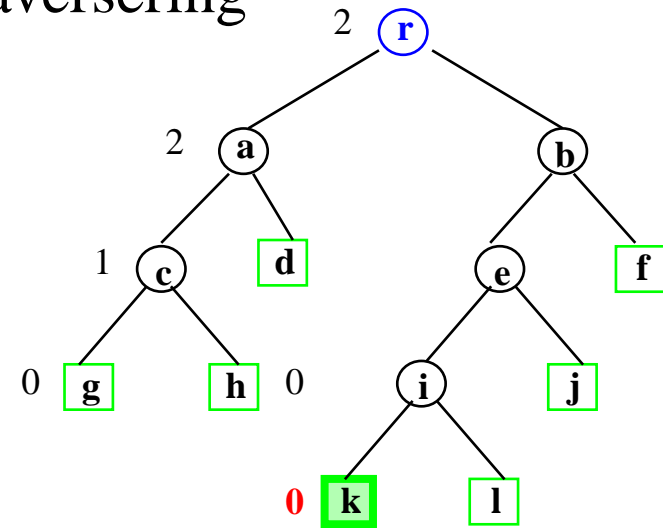
```
  else // 1+max{ height(p): p i children(v) }
```

```
    max=0
```

```
    for hver p i children(v)
```

```
      h=height(p); if (h>max) max=h
```

```
  return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
```

```
  if (isExternal(v)) return 0
```

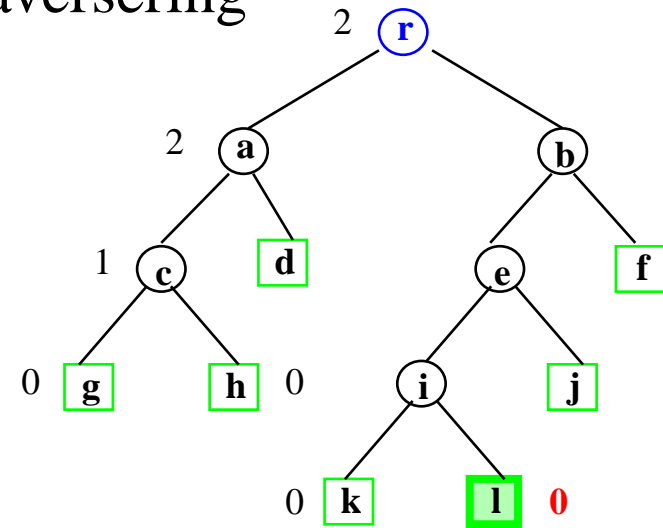
```
  else // 1+max{ height(p): p i children(v) }
```

```
    max=0
```

```
    for hver p i children(v)
```

```
      h=height(p); if (h>max) max=h
```

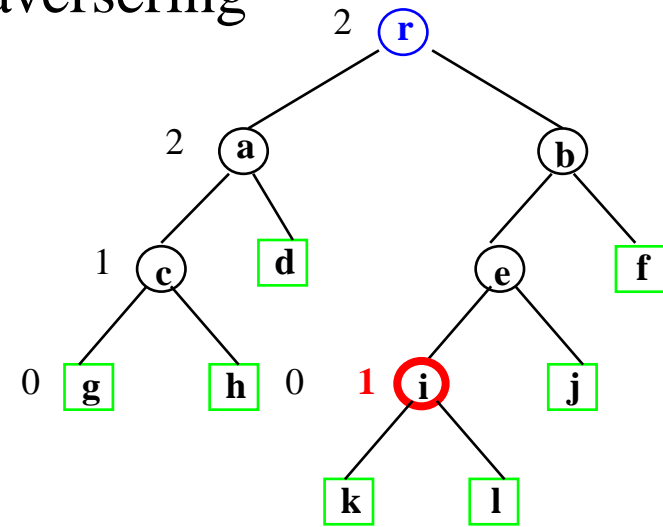
```
  return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

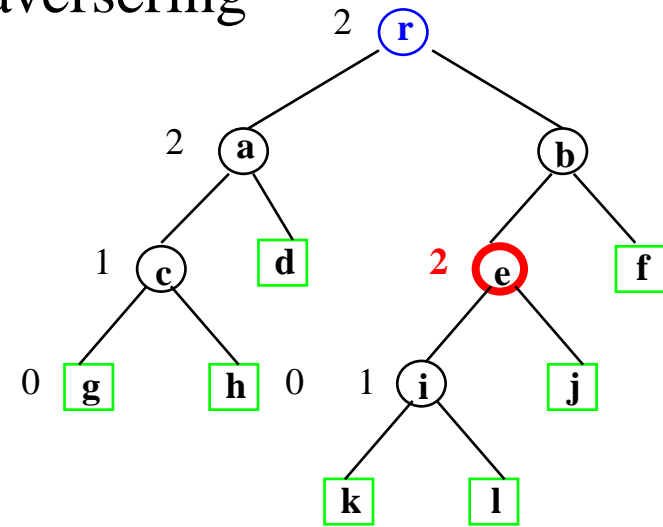
```
int height(Position v)
  if (isExternal(v)) return 0
  else // 1+max{ height(p): p i children(v) }
    max=0
    for hver p i children(v)
      h=height(p); if (h>max) max=h
  return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

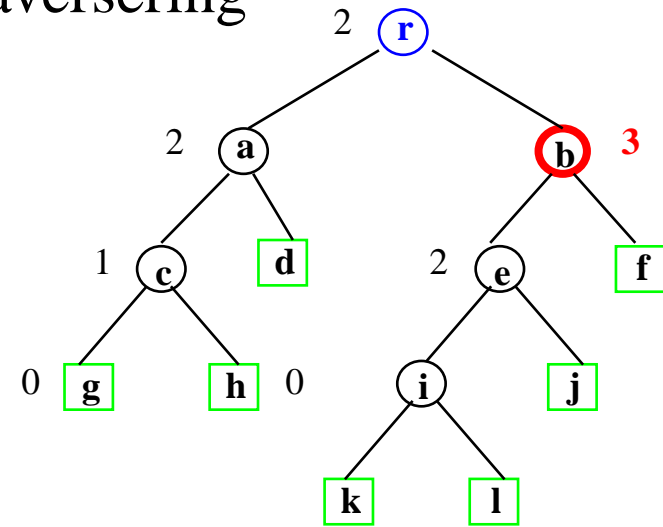
```
int height(Position v)
  if (isExternal(v)) return 0
  else // 1+max{ height(p): p i children(v) }
    max=0
    for hver p i children(v)
      h=height(p); if (h>max) max=h
  return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

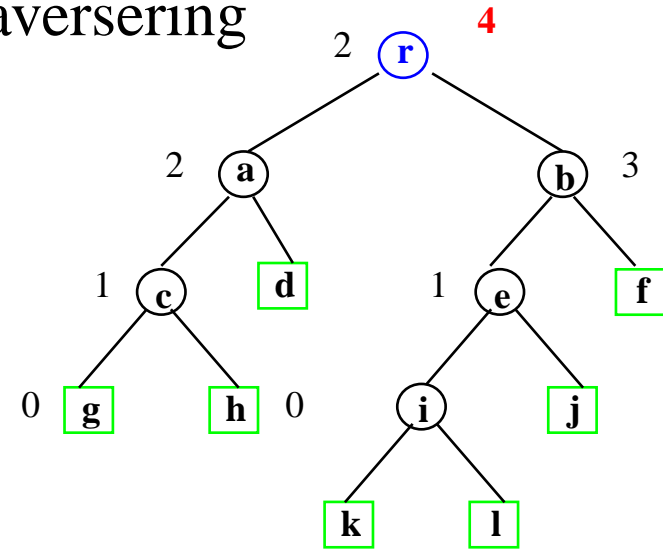
```
int height(Position v)
  if (isExternal(v)) return 0
  else // 1+max{ height(p): p i children(v) }
    max=0
    for hver p i children(v)
      h=height(p); if (h>max) max=h
  return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

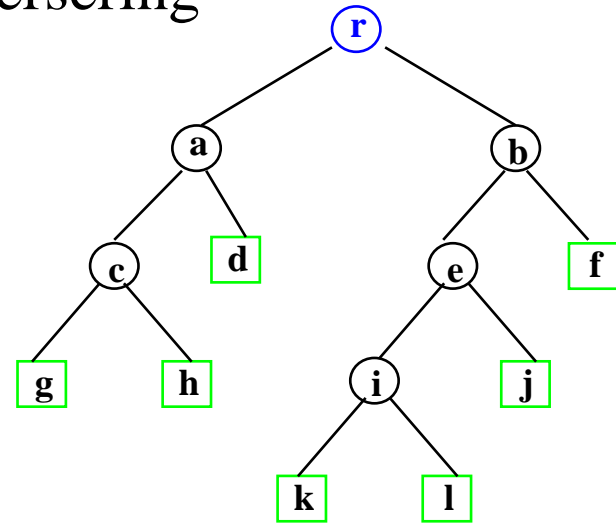
```
int height(Position v)
  if (isExternal(v)) return 0
  else // 1+max{ height(p): p i children(v) }
    max=0
    for hver p i children(v)
      h=height(p); if (h>max) max=h
  return 1+max
```



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
  if (isExternal(v)) return 0
  else // 1+max{ height(p): p i children(v) }
    max=0
    for hver p i children(v)
      h=height(p); if (h>max) max=h
    return 1+max
```



enumererer: **r, a,c,g,h,d, b, e,i,k,l,j, f**

4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
```

```
  if (isExternal(v)) return 0
```

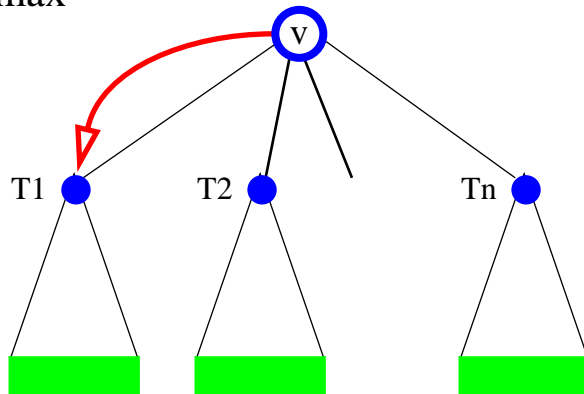
```
  else // 1+max{ height(p): p i children(v) }
```

```
    max=0
```

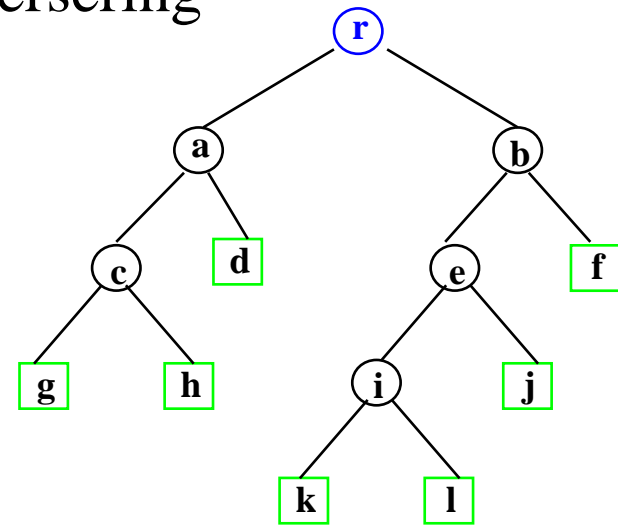
```
    for hver p i children(v)
```

```
      h=height(p); if (h>max) max=h
```

```
  return 1+max
```



enumererer: **r, a,c,g,h,d, b, e,i,k,l,j, f**



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
```

```
  if (isExternal(v)) return 0
```

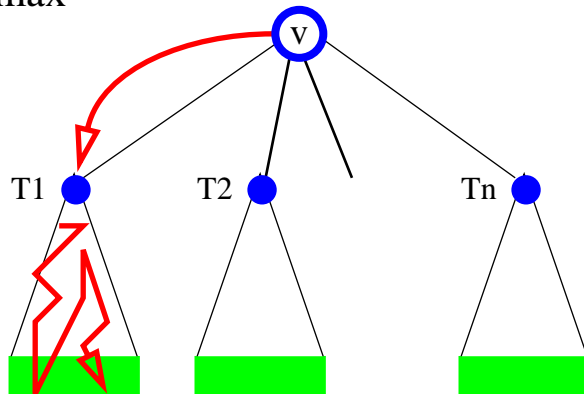
```
  else // 1+max{ height(p): p i children(v) }
```

```
    max=0
```

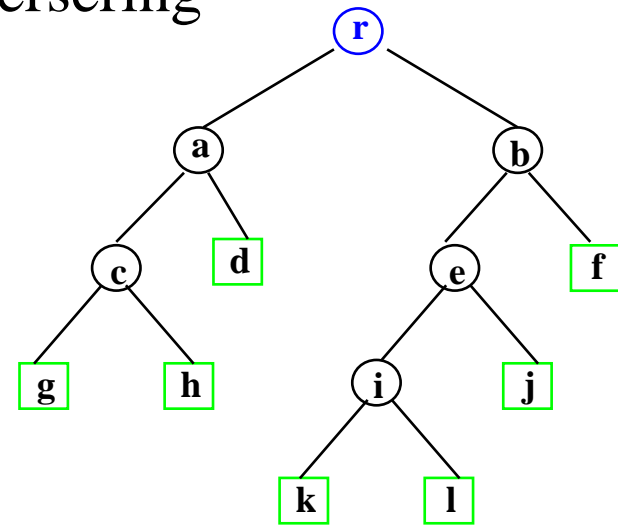
```
    for hver p i children(v)
```

```
      h=height(p); if (h>max) max=h
```

```
  return 1+max
```



enumererer: **r, a,c,g,h,d, b, e,i,k,l,j, f**



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
```

```
  if (isExternal(v)) return 0
```

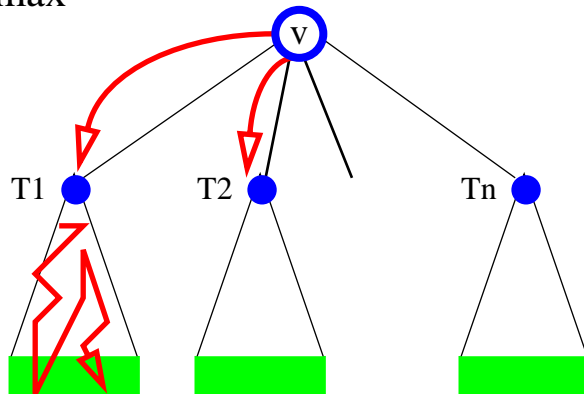
```
  else // 1+max{ height(p): p i children(v) }
```

```
    max=0
```

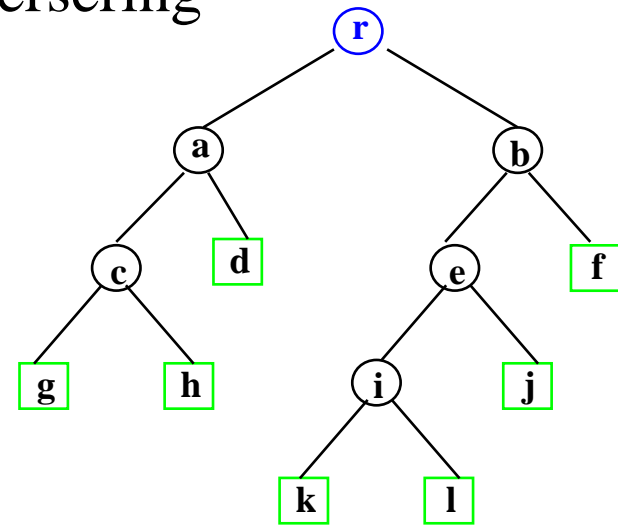
```
    for hver p i children(v)
```

```
      h=height(p); if (h>max) max=h
```

```
  return 1+max
```



enumererer: **r, a,c,g,h,d, b, e,i,k,l,j, f**



4. Tre-Algoritmer : traversering

PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
```

```
  if (isExternal(v)) return 0
```

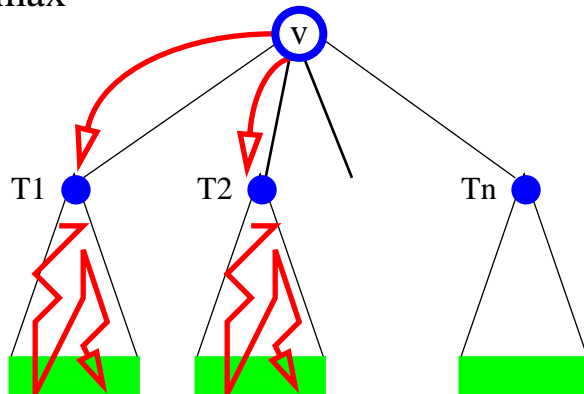
```
  else // 1+max{ height(p): p i children(v) }
```

```
    max=0
```

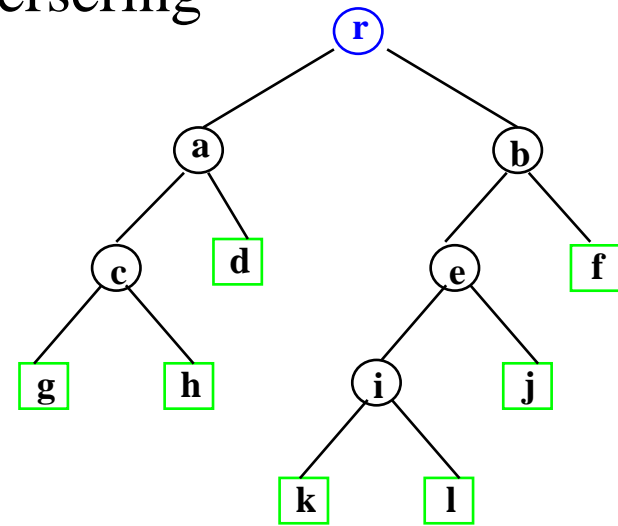
```
    for hver p i children(v)
```

```
      h=height(p); if (h>max) max=h
```

```
  return 1+max
```



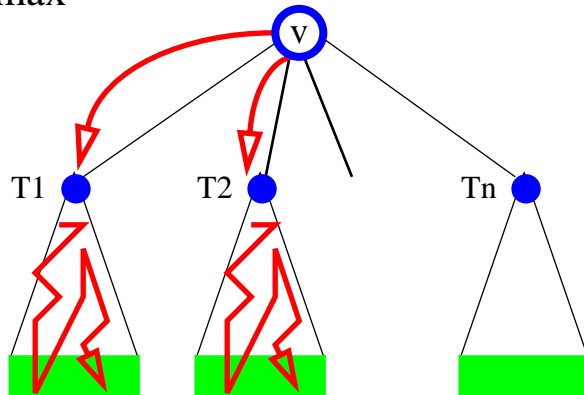
enumererer: **r, a,c,g,h,d, b, e,i,k,l,j, f**



4. Tre-Algoritmer : traversering

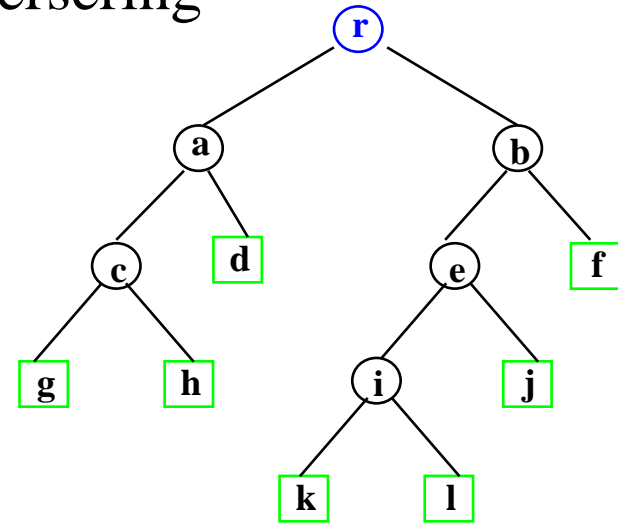
PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
  if (isExternal(v)) return 0
  else // 1+max{ height(p): p i children(v) }
    max=0
    for hver p i children(v)
      h=height(p); if (h>max) max=h
    return 1+max
```



```
void DFS(Tree T, Position v) {
  for hver p i T.children(v)
    DFS(T,p) }
}
```

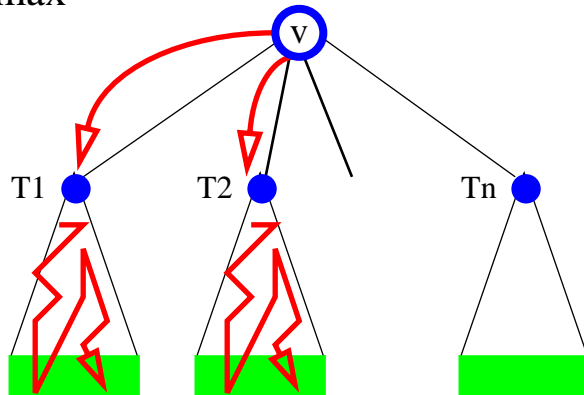
enumererer: **r, a,c,g,h,d, b, e,i,k,l,j, f**



4. Tre-Algoritmer : traversering

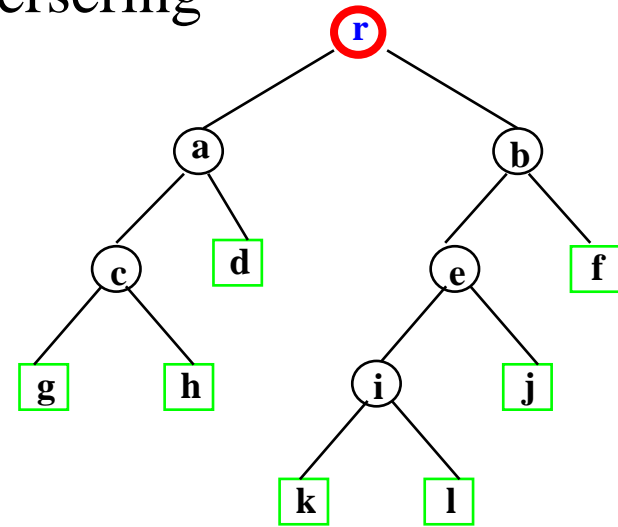
PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
  if (isExternal(v)) return 0
  else // 1+max{ height(p): p i children(v) }
    max=0
    for hver p i children(v)
      h=height(p); if (h>max) max=h
    return 1+max
```



```
void DFS(Tree T, Position v) {
  for hver p i T.children(v)
    DFS(T,p) }
}
```

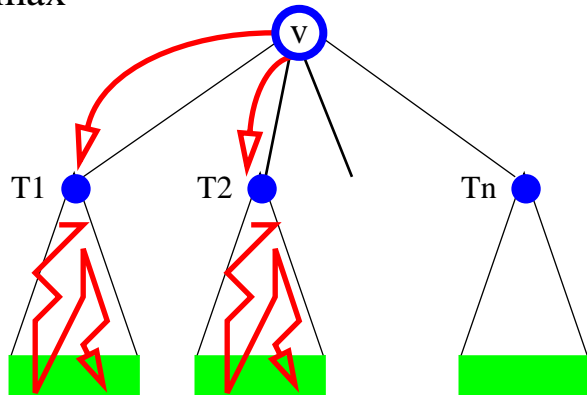
enumererer: **r, a,c,g,h,d, b, e,i,k,l,j, f**



4. Tre-Algoritmer : traversering

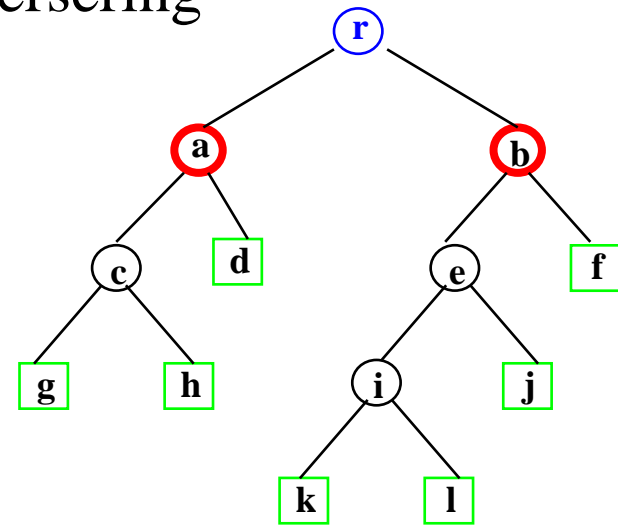
PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
  if (isExternal(v)) return 0
  else // 1+max{ height(p): p i children(v) }
    max=0
    for hver p i children(v)
      h=height(p); if (h>max) max=h
    return 1+max
```



```
void DFS(Tree T, Position v) {
  for hver p i T.children(v)
    DFS(T,p) }
}
```

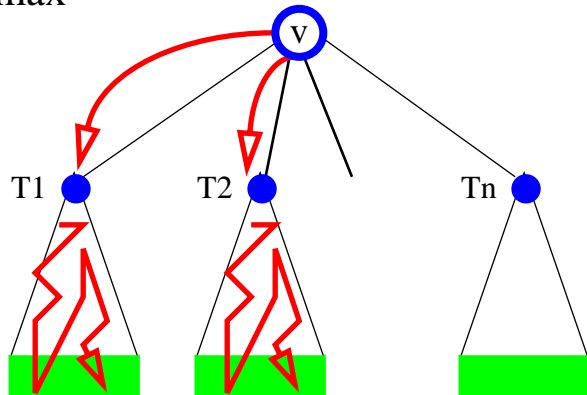
enumererer: **r, a,c,g,h,d, b, e,i,k,l,j, f**



4. Tre-Algoritmer : traversering

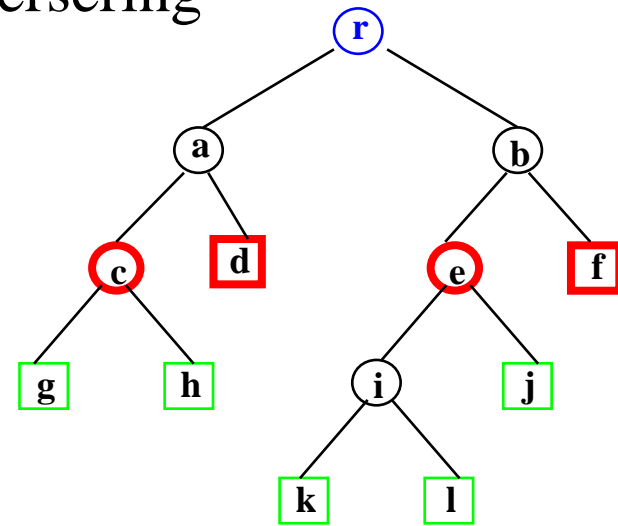
PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
  if (isExternal(v)) return 0
  else // 1+max{ height(p): p i children(v) }
    max=0
    for hver p i children(v)
      h=height(p); if (h>max) max=h
    return 1+max
```



```
void DFS(Tree T, Position v) {
  for hver p i T.children(v)
    DFS(T,p) }
}
```

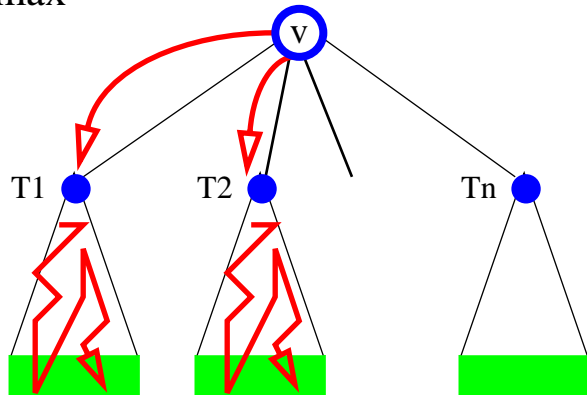
enumererer: **r, a,c,g,h,d, b, e,i,k,l,j, f**



4. Tre-Algoritmer : traversering

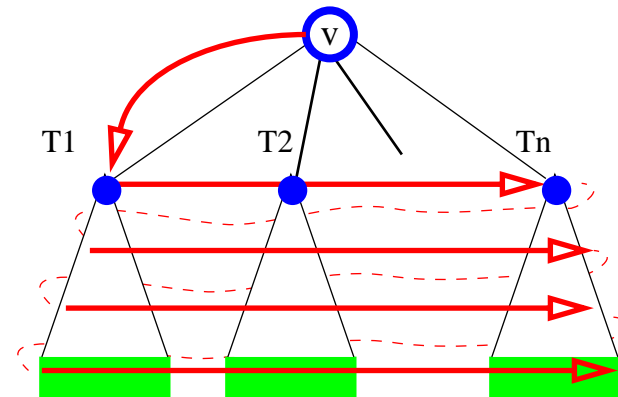
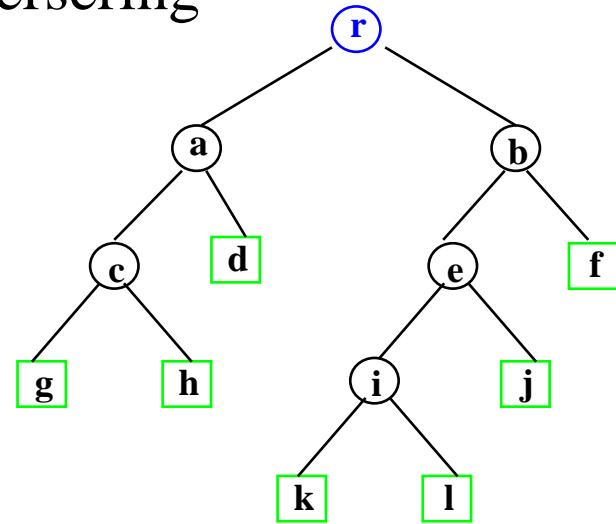
PositionIterator positions() – ok ... men i hvilken rekkefølge ?

```
int height(Position v)
  if (isExternal(v)) return 0
  else // 1+max{ height(p): p i children(v) }
    max=0
    for hver p i children(v)
      h=height(p); if (h>max) max=h
    return 1+max
```



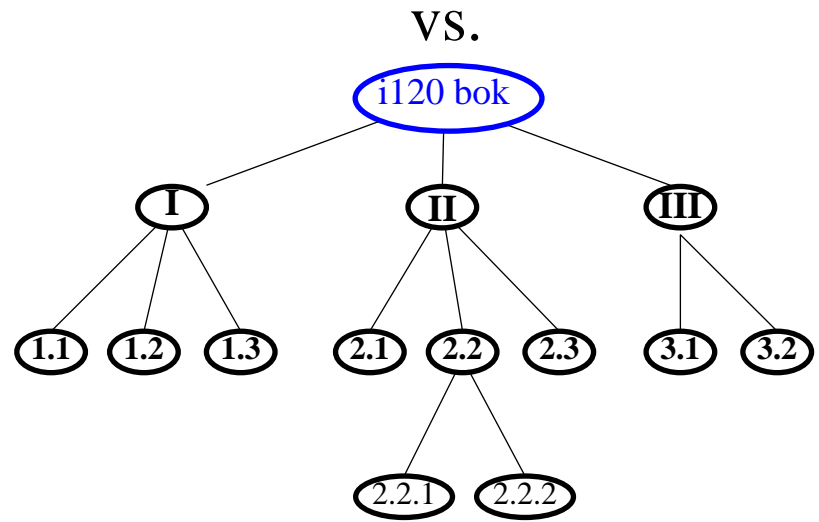
```
void DFS(Tree T, Position v) {
  for hver p i T.children(v)
    DFS(T,p) }
}
```

enumererer: **r, a,c,g,h,d, b, e,i,k,l,j, f**

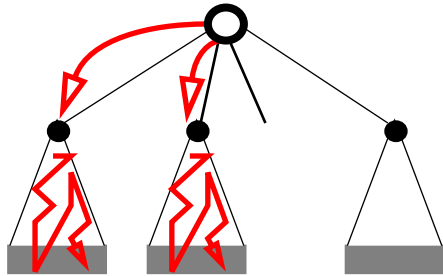


```
void BFS(Tree T, Position v)
```

enumererer: **r, a,b, c,d,e,f, g,h,i,j, k,l**



DFS



I-120 bok

Chap. I Design Principles

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

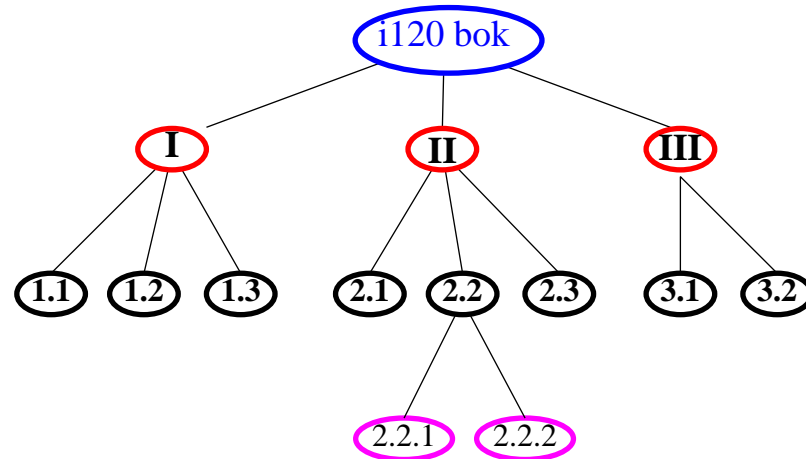
Chap. II Analysis Tools

- 2.1 Alg. Analysis
- 2.2 Running Time
 - 2.2.1 O-notation
 - 2.2.2 Average Case
- 2.3 Worst Case

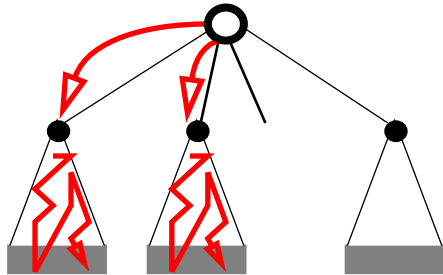
Chap. III Basic DS

- 3.1 Stacks
- 3.2 Queues

VS.



DFS



I-120 bok

Chap. I Design Principles

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

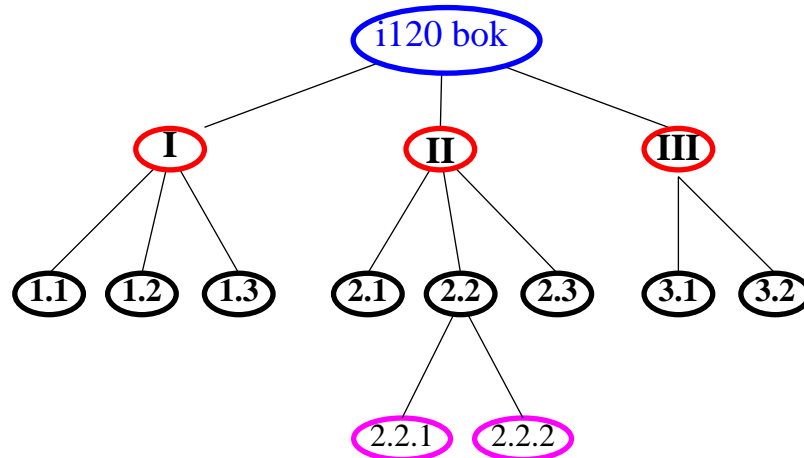
Chap. II Analysis Tools

- 2.1 Alg. Analysis
- 2.2 Running Time
 - 2.2.1 O-notation
 - 2.2.2 Average Case
- 2.3 Worst Case

Chap. III Basic DS

- 3.1 Stacks
- 3.2 Queues

vs.



I-120 bok

Chap. I Design Principles

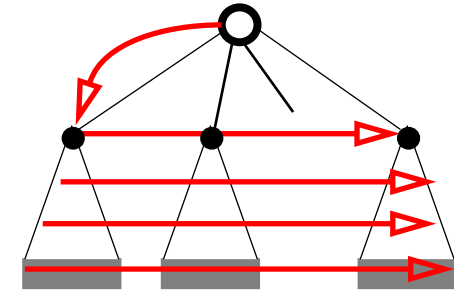
Chap. II Analysis Tools

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA
- 2.1 Alg. Analysis
- 2.2 Running Time
- 2.3 Worst Case

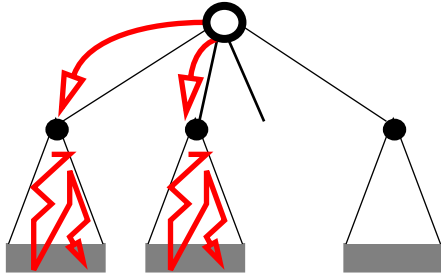
- 3.1 Stacks
- 3.2 Queues

- 2.2.1 O-notation
- 2.2.2 Average Case

BFS



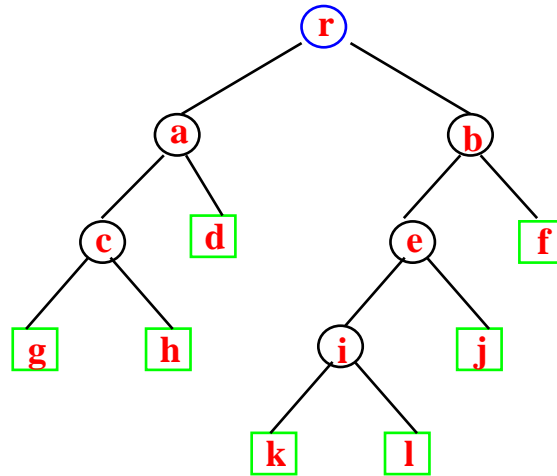
DFS



r, a,c, h,g, d, b, e,i,k,l, j, f

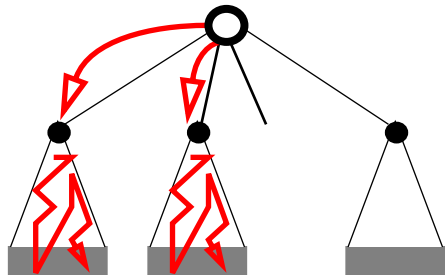
```
/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */
```

og

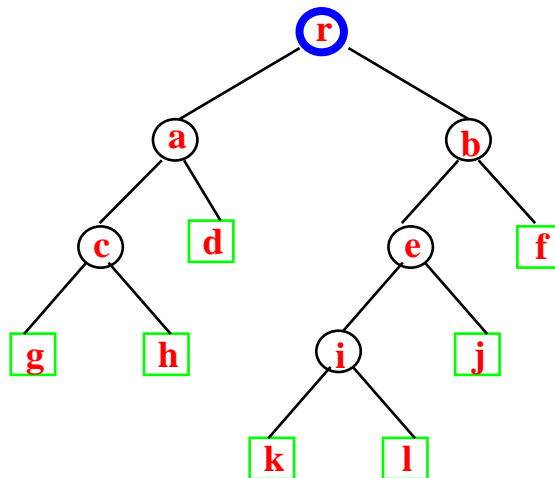


BFS

DFS



og

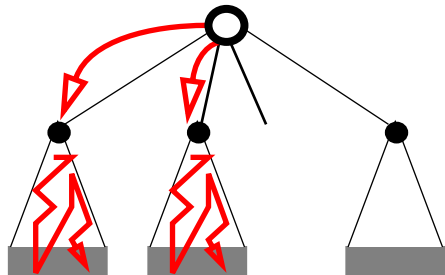


BFS

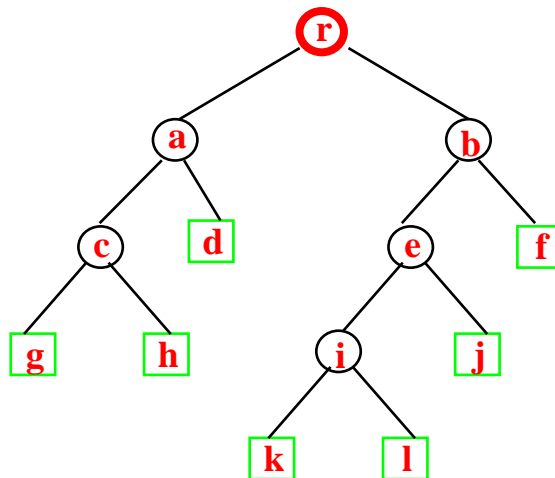
```
/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */
```

r
S

DFS



og



BFS

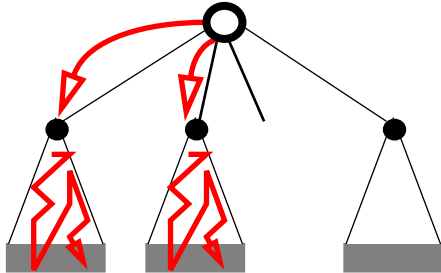
r

```
/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */
```

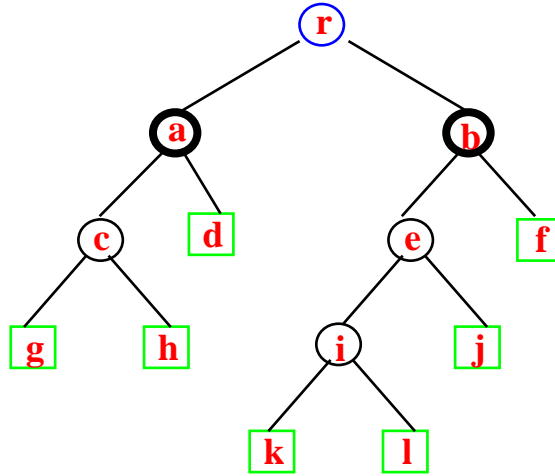
r

S

DFS



og



BFS

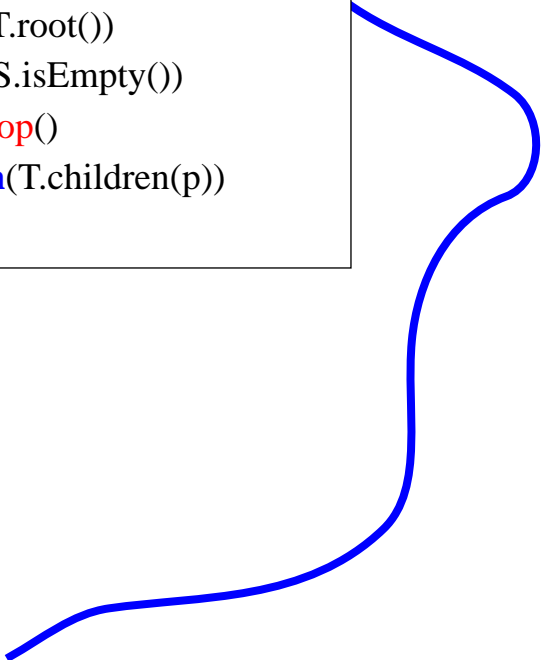
r

```

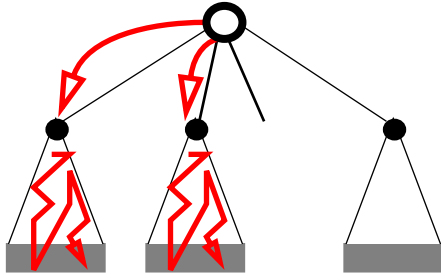
/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */

```

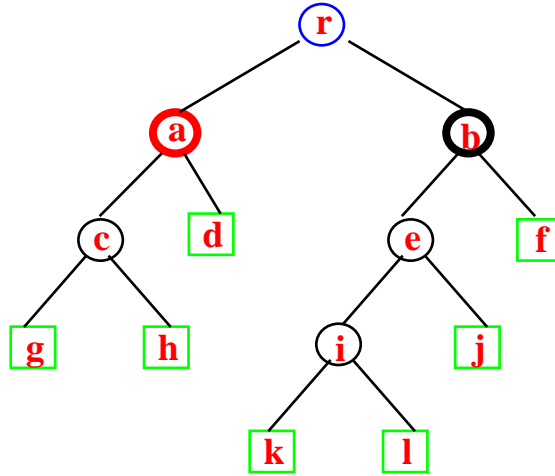
r	a	
	b	
S	S	



DFS



og



BFS

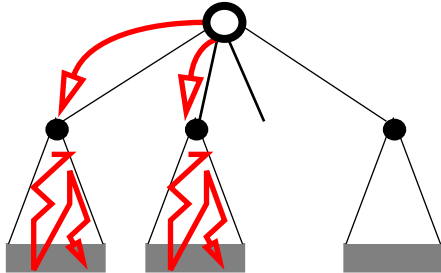
r, a,

```

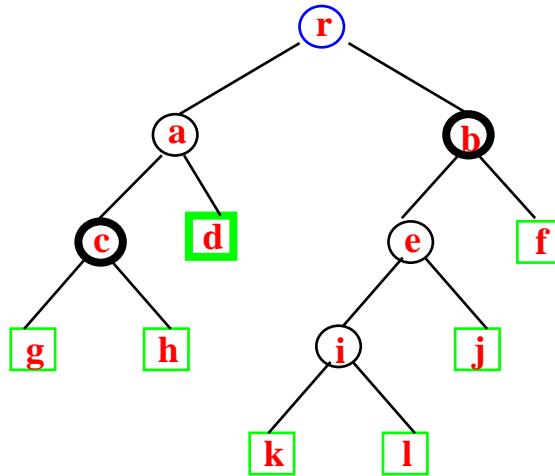
/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */
    
```

r	a	
S	b	S

DFS



og



BFS

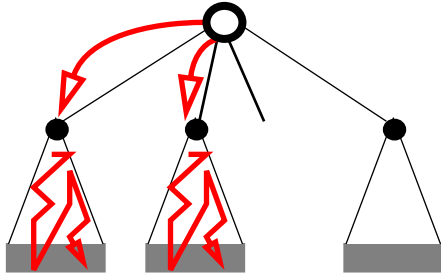
r, a

```

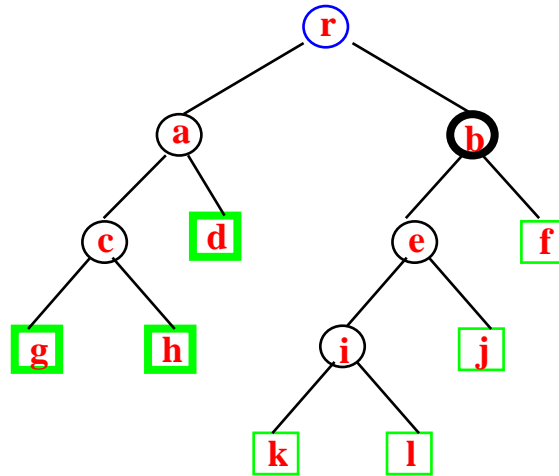
/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */
    
```

			c
	a	d	
r	b	b	
S	S	S	

DFS



og



BFS

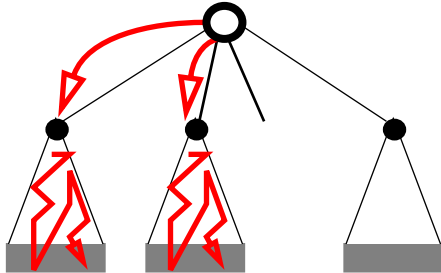
r, a, c

```

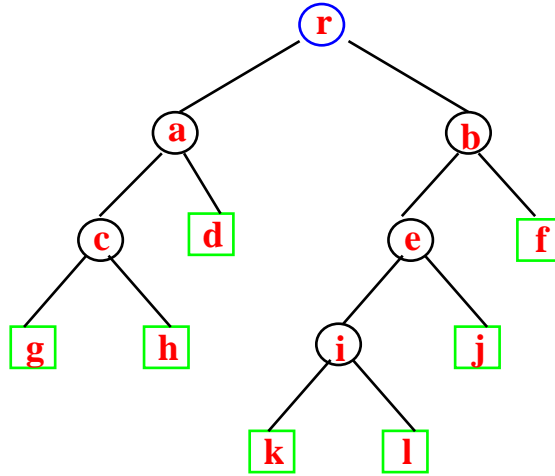
/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */
    
```

			h
		c	g
	a	d	d
r	b	b	b
S	S	S	S

DFS



og



BFS

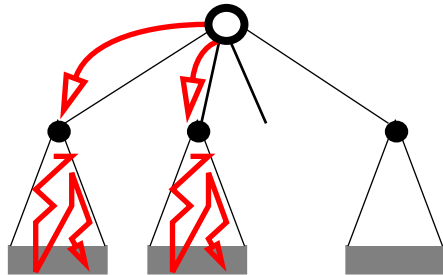
r, a,c, h,g, d, b

```

/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */
    
```

			h
	a	c	g
r	b	d	d
S	S	S	b ↓

DFS



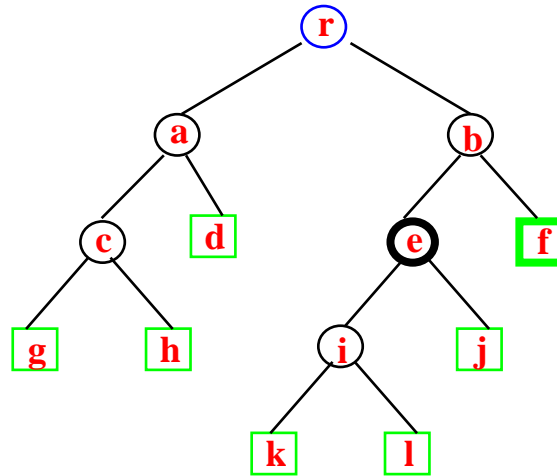
r, a, c, h, g, d, b, e

```

/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */
    
```

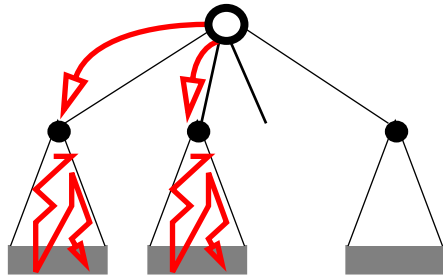
r	a	c	h	e
b	b	d	d	f
S	S	S	S	S

og

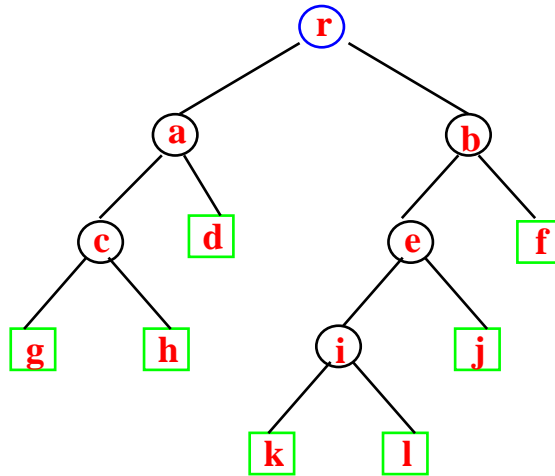


BFS

DFS



og



BFS

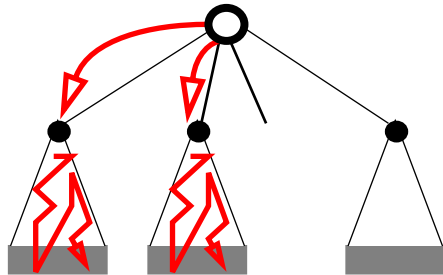
r, a,c, h,g, d, b, e,i,k,l, j, f

```

/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */
    
```

r	a	c	h	e	i	k
b	d	g	d	f	j	l
b	b	d	b	f	f	f
S	S	S	S	S	S	S

DFS



r, a,c, h,g, d, b, e,i,k,l, j, f

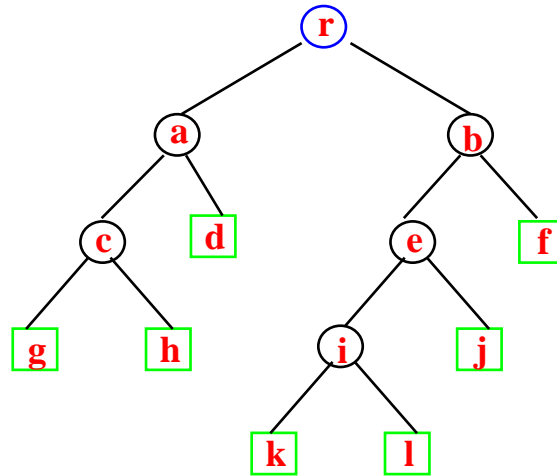
```

/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */
    
```

r	a	c	h	e	i	k
b	d	g	f	j	l	j
S	S	S	S	S	S	S

trav(T, new StackIm())

og



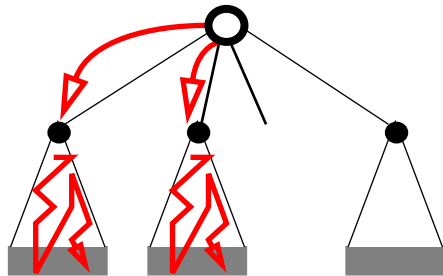
BFS

trav(Tree T, LiFi S)

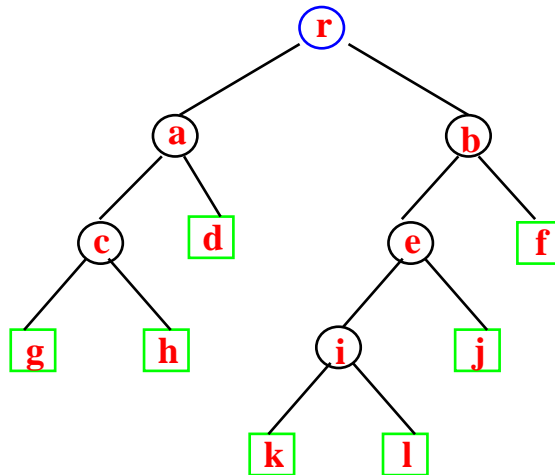
```

S.add(T.root());
while (! S.isEmpty())
  p = S.remove();
  S.add(T.children(p))
    
```

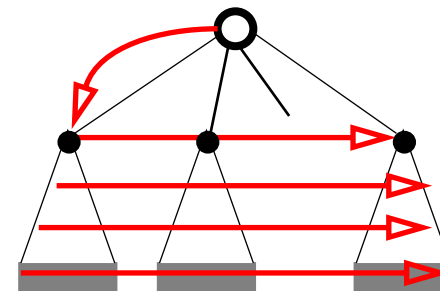

DFS



og



BFS



r, a,c, h,g, d, b, e,i,k,l, j, f

```

/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */
    
```

r	a	c	h	e	i	k
b	d	g	f	j	l	j
b	b	d	f	f	f	f
S	S	S	S	S	S	S

“samme” algoritme:

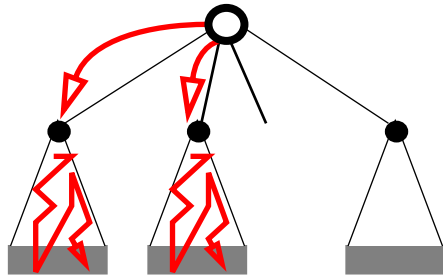
```

trav(Tree T, LiFi S)
 S.add(T.root());
 while (! S.isEmpty())
   p = S.remove();
   S.add(T.children(p))
    
```

trav(T, new StackIm())

trav(T, new QueueIm())

DFS



r, a,c, h,g, d, b, e,i,k,l, j, f

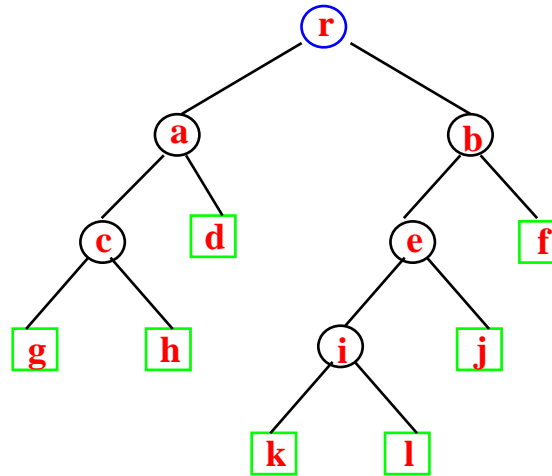
```

/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */
    
```

r	a	c	h	e	i	k
	b	d	g	f	j	l
S	S	S	S	S	S	S

trav(T, new StackIm())

og

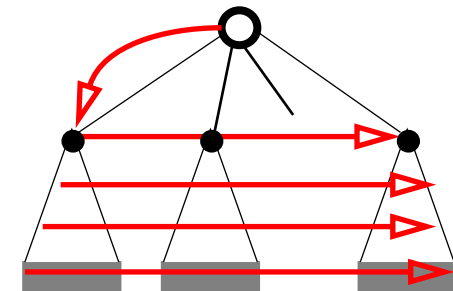


“samme” algoritme:

```

trav(Tree T, LiFi S)
 S.add(T.root());
 while (! S.isEmpty())
   p = S.remove();
   S.add(T.children(p))
    
```

BFS



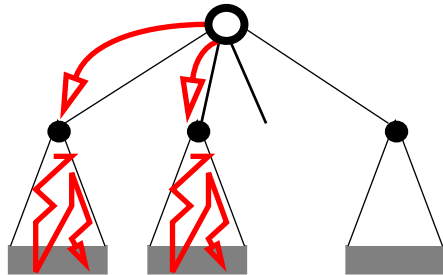
r, a,b, c,d,e,f, g,h,i,j, k,l

```

/*BFS(Tree T)
 * Queue S = new QueueIm()
 * S.enqueue(T.root())
 * while (!S.isEmpty())
 *   p= S.dequeue()
 *   S.enqueue(T.children(p))
 */
    
```

trav(T, new QueueIm())

DFS



r, a,c, h,g, d, b, e,i,k,l, j, f

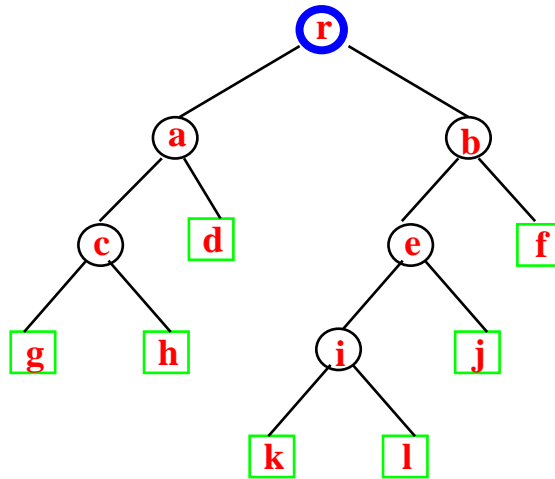
```

/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */
    
```

r	a	c	h	e	i	k
	b	d	d	f	j	l
S	S	S	S	S	S	S

trav(T, new StackIm())

og

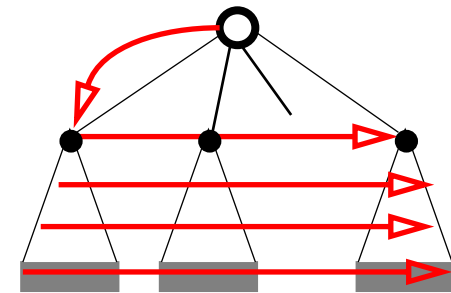


“samme” algoritme:

```

trav(Tree T, LiFi S)
 S.add(T.root());
 while (! S.isEmpty())
   p = S.remove();
   S.add(T.children(p))
    
```

BFS



r

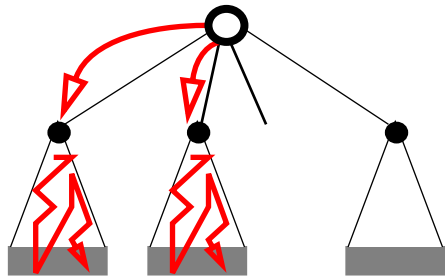
```

/*BFS(Tree T)
 * Queue S = new QueueIm()
 * S.enqueue(T.root())
 * while (!S.isEmpty())
 *   p= S.dequeue()
 *   S.enqueue(T.children(p))
 */
    
```

r

trav(T, new QueueIm())

DFS



r, a,c, h,g, d, b, e,i,k,l, j, f

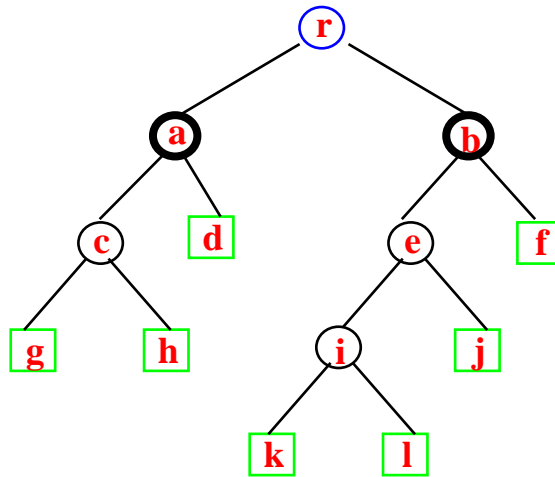
```

/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */
    
```

r	a	c	h	e	i	k
	b	d	d	f	j	l
		b	b		f	j
			↓			↓
S	S	S	S	S	S	S

trav(T, new StackIm())

og

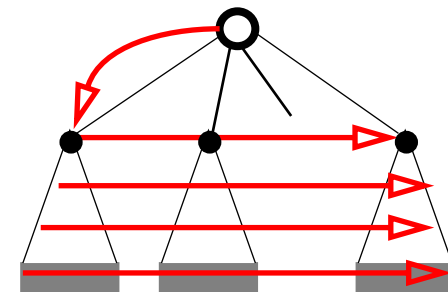


“samme” algoritme:

```

trav(Tree T, LiFi S)
 S.add(T.root());
 while (! S.isEmpty())
   p = S.remove();
   S.add(T.children(p))
    
```

BFS



r

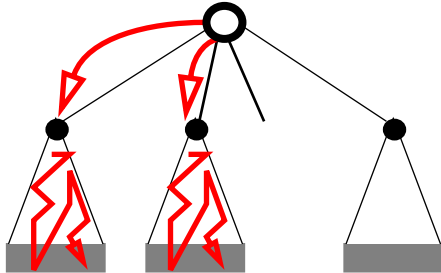
```

/*BFS(Tree T)
 * Queue S = new QueueIm()
 * S.enqueue(T.root())
 * while (!S.isEmpty())
 *   p= S.dequeue()
 *   S.enqueue(T.children(p))
 */
    
```

r		
	a	b

trav(T, new QueueIm())

DFS



r, a,c, h,g, d, b, e,i,k,l, j, f

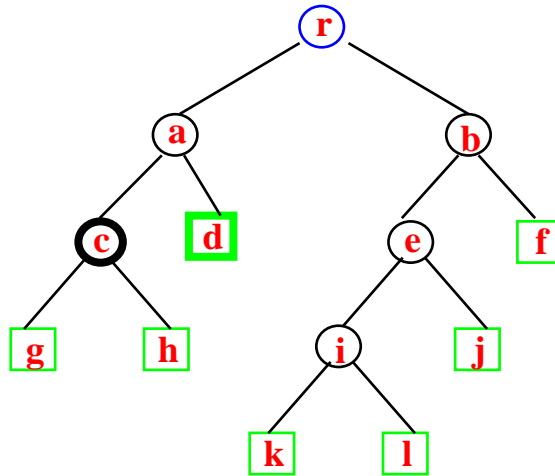
```

/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */
    
```

r	a	c	h	e	i	k
b	d	g	f	j	l	j
S	S	S	S	S	S	S

trav(T, new StackIm())

og

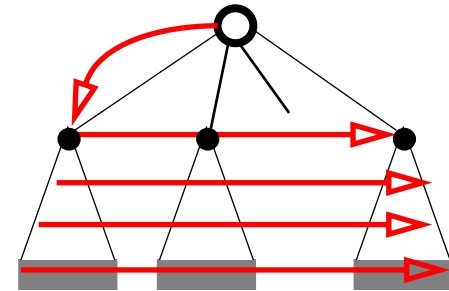


“samme” algoritme:

```

trav(Tree T, LiFi S)
 S.add(T.root());
 while (! S.isEmpty())
   p = S.remove();
   S.add(T.children(p))
    
```

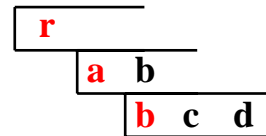
BFS



r, a

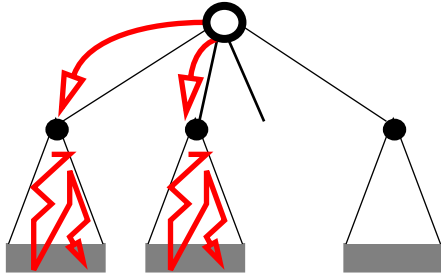
```

/*BFS(Tree T)
 * Queue S = new QueueIm()
 * S.enqueue(T.root())
 * while (!S.isEmpty())
 *   p= S.dequeue()
 *   S.enqueue(T.children(p))
 */
    
```



trav(T, new QueueIm())

DFS



r, a,c, h,g, d, b, e,i,k,l, j, f

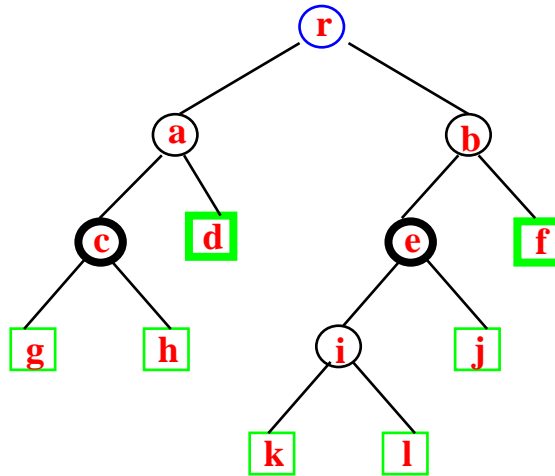
```

/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */
    
```

r	a	c	h	e	i	k
	b	d	g	f	j	l
S	S	S	S	S	S	S

trav(T, new StackIm())

og

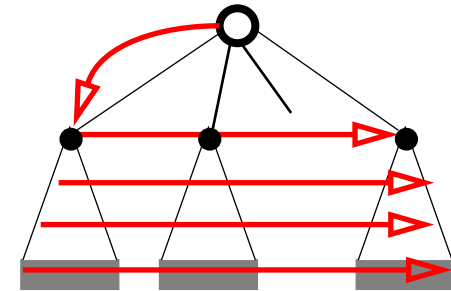


“samme” algoritme:

```

trav(Tree T, LiFi S)
 S.add(T.root());
 while (! S.isEmpty())
   p = S.remove();
   S.add(T.children(p))
    
```

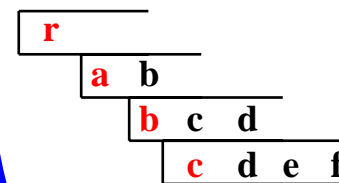
BFS



r, a, b

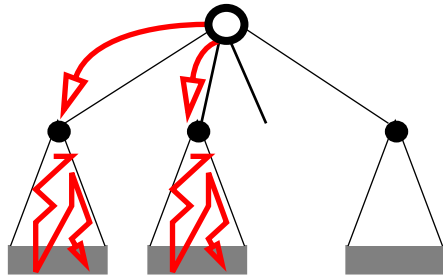
```

/*BFS(Tree T)
 * Queue S = new QueueIm()
 * S.enqueue(T.root())
 * while (!S.isEmpty())
 *   p= S.dequeue()
 *   S.enqueue(T.children(p))
 */
    
```



trav(T, new QueueIm())

DFS



r, a,c, h,g, d, b, e,i,k,l, j, f

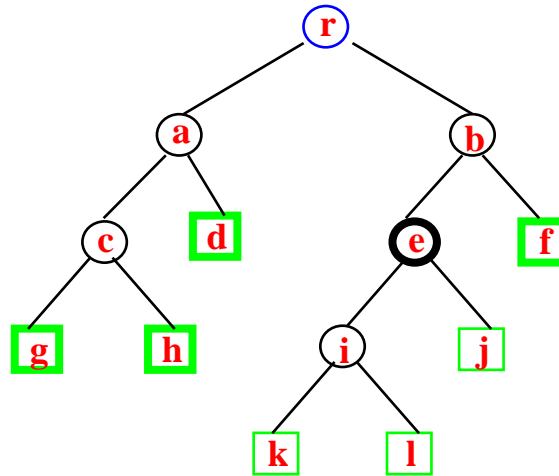
```

/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */
    
```

r	a	c	h	e	i	k
	b	d	d	f	j	l
S	S	S	S	S	S	S

trav(T, new StackIm())

og

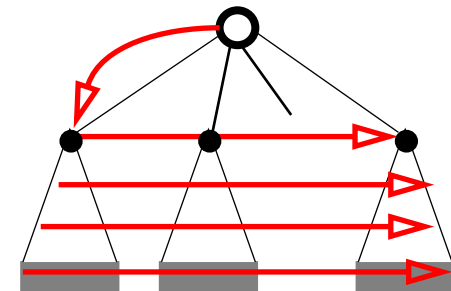


“samme” algoritme:

```

trav(Tree T, LiFi S)
 S.add(T.root());
 while (! S.isEmpty())
   p = S.remove();
   S.add(T.children(p))
    
```

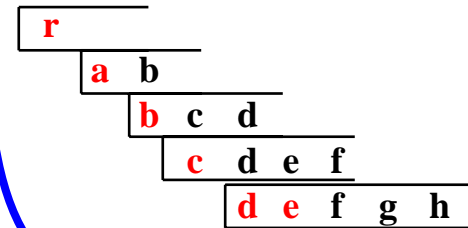
BFS



r, a,b, c

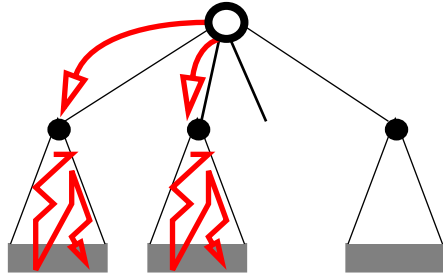
```

/*BFS(Tree T)
 * Queue S = new QueueIm()
 * S.enqueue(T.root())
 * while (!S.isEmpty())
 *   p= S.dequeue()
 *   S.enqueue(T.children(p))
 */
    
```



trav(T, new QueueIm())

DFS



r, a,c, h,g, d, b, e,i,k,l, j, f

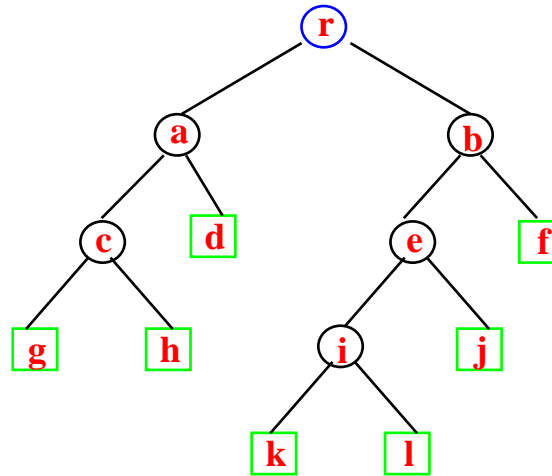
```

/*DFS(Tree T)
 * Stack S = new StackIm()
 * S.push(T.root())
 * while (!S.isEmpty())
 *   p= S.pop()
 *   S.push(T.children(p))
 */
    
```

r	a	c	h	e	i	k
	b	d	g	f	j	l
S	S	S	S	S	S	S

trav(T, new StackIm())

og

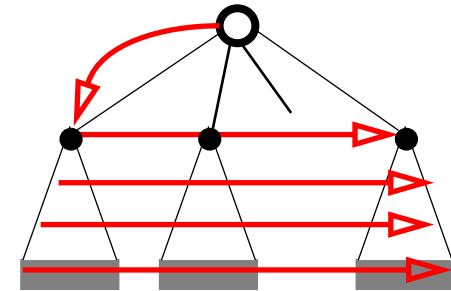


“samme” algoritme:

```

trav(Tree T, LiFi S)
 S.add(T.root());
 while (! S.isEmpty())
   p = S.remove();
   S.add(T.children(p))
    
```

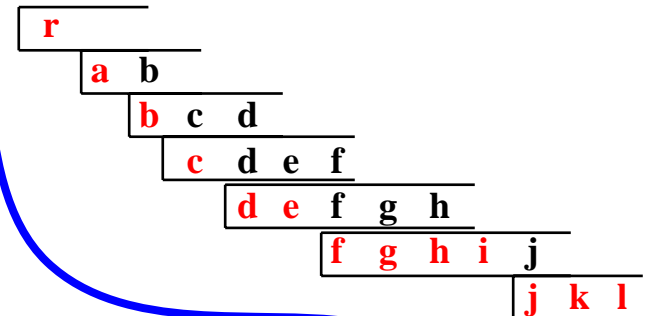
BFS



r, a,b, c,d,e,f, g,h,i,j, k,l

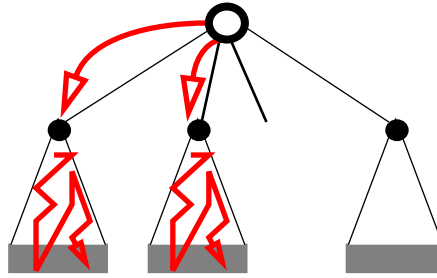
```

/*BFS(Tree T)
 * Queue S = new QueueIm()
 * S.enqueue(T.root())
 * while (!S.isEmpty())
 *   p= S.dequeue()
 *   S.enqueue(T.children(p))
 */
    
```



trav(T, new QueueIm())

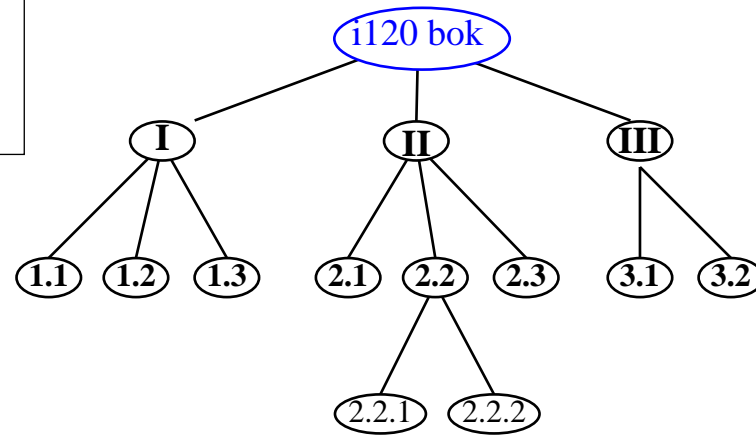
DFS



DFS(Tree T, Position v)

for hver p i T.children(v)

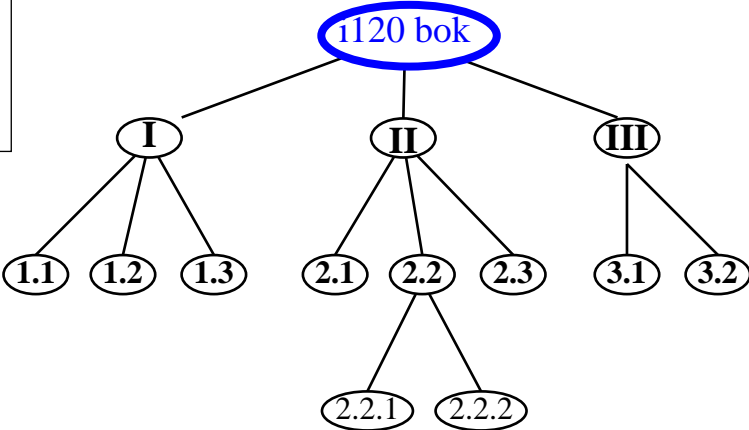
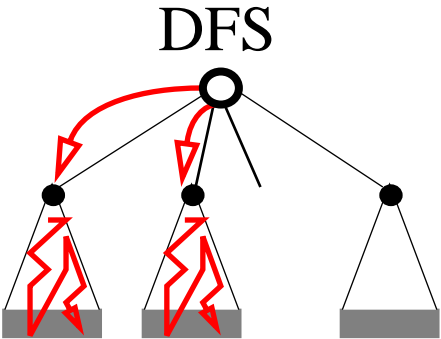
DFS(T,p)



PreOrder

*gjør jobben
FØR
rekursive kall*

```
DFS(Tree T, Position v)  
  Print(v.element())  
  for hver p i T.children(v)  
    DFS(T,p)
```

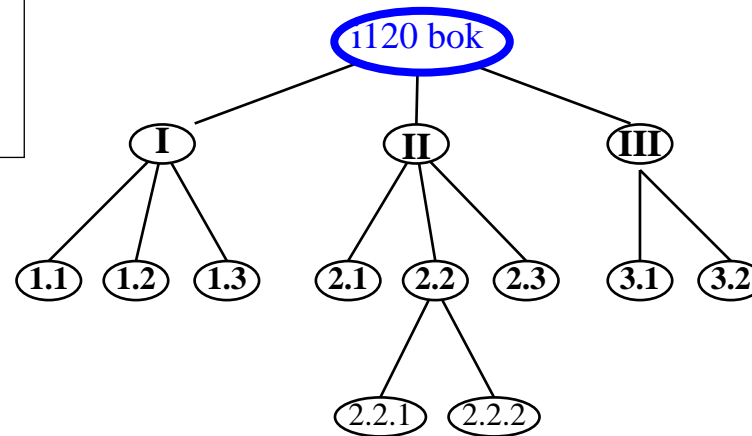
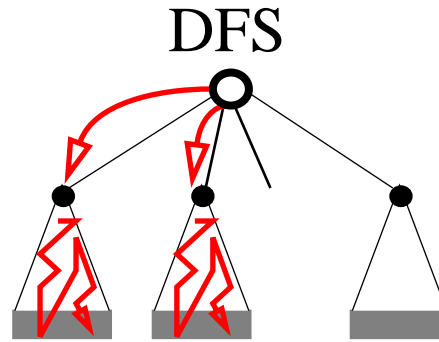


PreOrder

*gjør jobben
FØR
rekursive kall*

```
DFS(Tree T, Position v)
  Print(v.element())
  for hver p i T.children(v)
    DFS(T,p)
```

I-120 bok

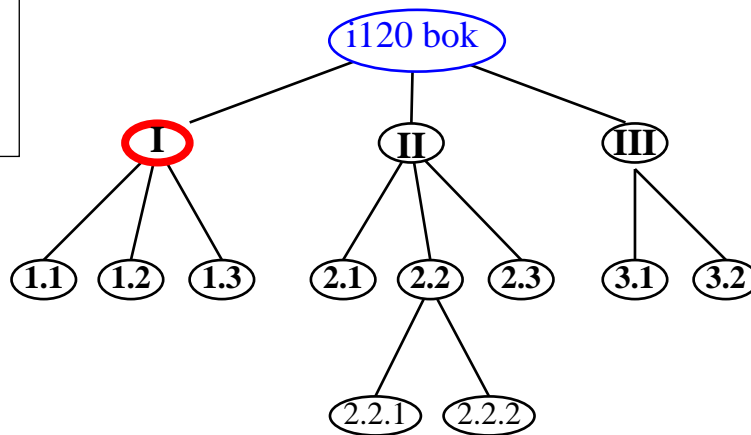
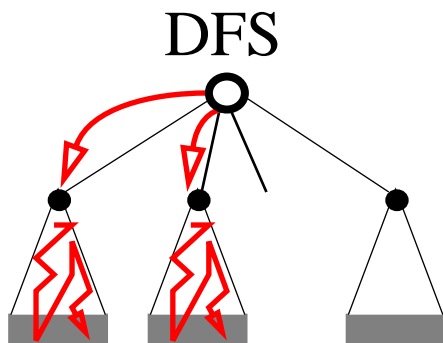


PreOrder

gjør jobben
FØR
rekursive kall

```
DFS(Tree T, Position v)
  Print(v.element())
  for hver p i T.children(v)
    DFS(T,p)
```

I-120 bok



PreOrder

*gjør jobben
FØR
rekursive kall*

DFS(Tree T, Position v)

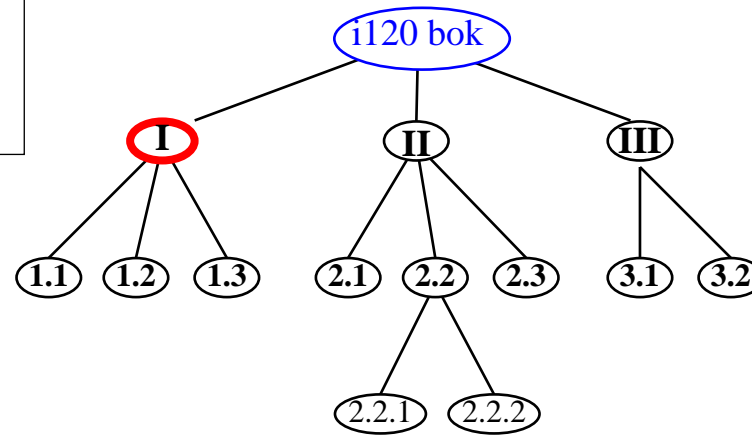
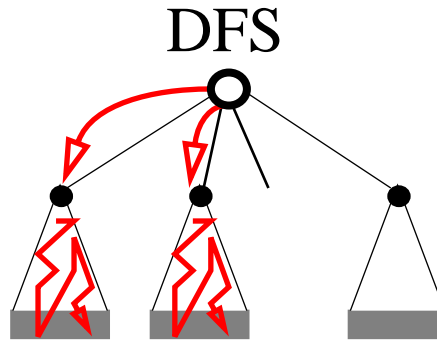
Print(v.element())

for hver p i T.children(v)

DFS(T,p)

I-120 bok

Chap. I Design Principles



PreOrder

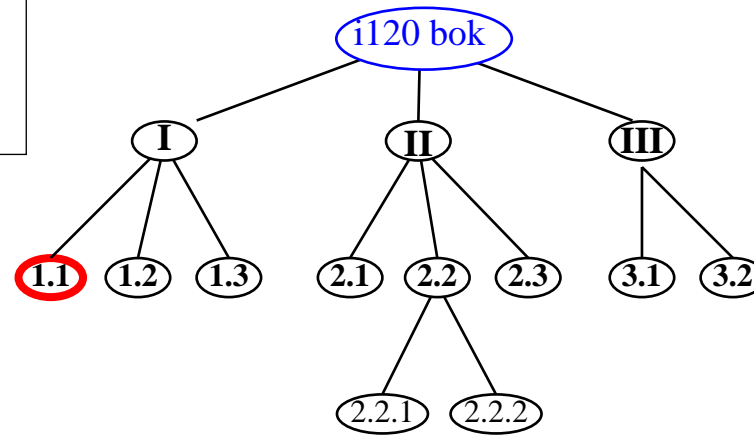
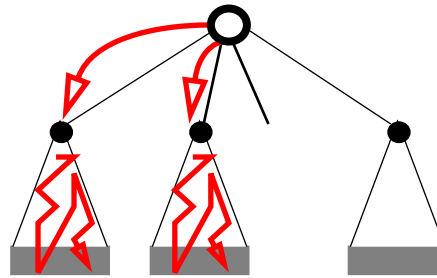
*gjør jobben
FØR
rekursive kall*

```
DFS(Tree T, Position v)
  Print(v.element())
  for hver p i T.children(v)
    DFS(T,p)
```

I-120 bok

Chap. I Design Principles
1.1 DS & Alg

DFS



PreOrder

*gjør jobben
FØR
rekursive kall*

DFS(Tree T, Position v)

Print(v.element())

for hver p i T.children(v)

DFS(T,p)

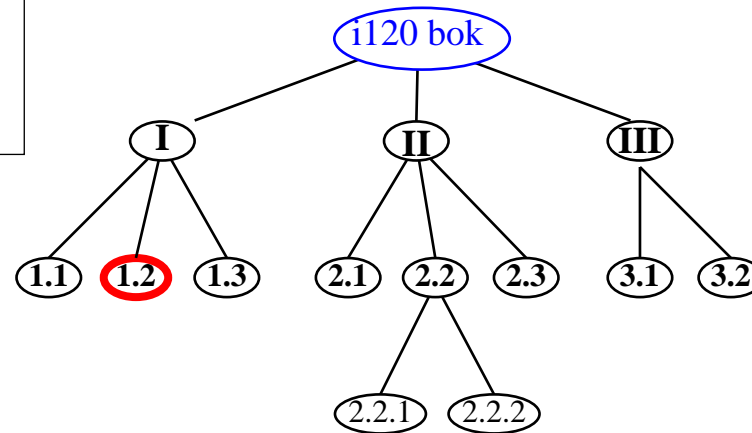
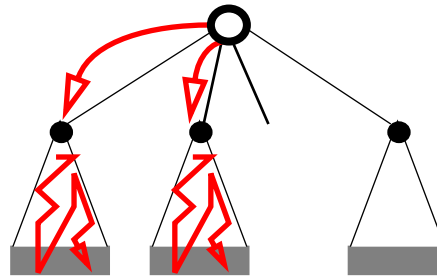
I-120 bok

Chap. I Design Principles

1.1 DS & Alg

1.2 OO-Programming

DFS



PreOrder

*gjør jobben
FØR
rekursive kall*

DFS(Tree T, Position v)

Print(v.element())

for hver p i T.children(v)

DFS(T,p)

I-120 bok

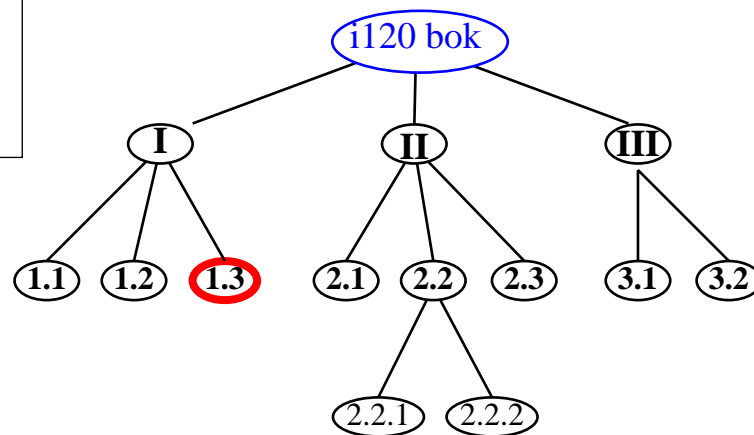
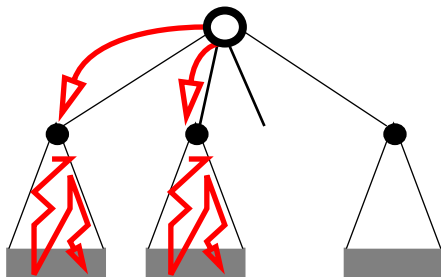
Chap. I Design Principles

1.1 DS & Alg

1.2 OO-Programming

1.3 JAVA

DFS



PreOrder

*gjør jobben
FØR
rekursive kall*

DFS(Tree T, Position v)

Print(v.element())

for hver p i T.children(v)

DFS(T,p)

I-120 bok

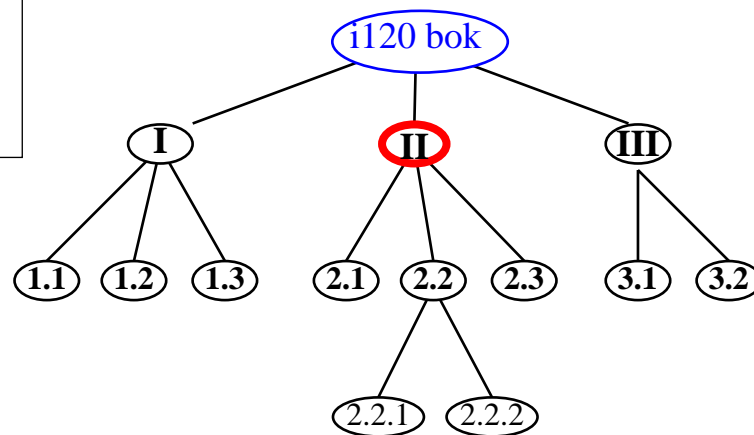
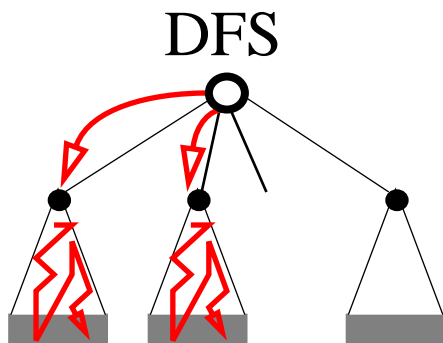
Chap. I Design Principles

1.1 DS & Alg

1.2 OO-Programming

1.3 JAVA

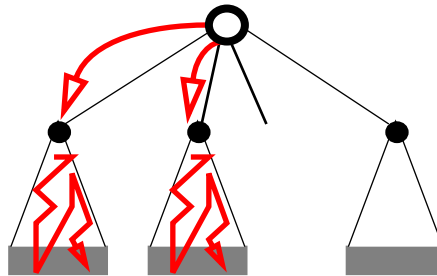
Chap. II Analysis Tools



PreOrder

*gjør jobben
FØR
rekursive kall*

DFS



DFS(Tree T, Position v)

Print(v.element())

for hver p i T.children(v)

DFS(T,p)

I-120 bok

Chap. I Design Principles

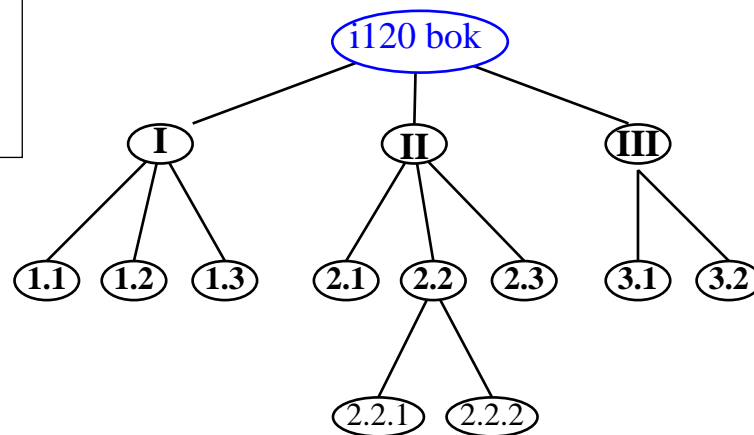
- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

Chap. II Analysis Tools

- 2.1 Alg. Analysis
- 2.2 Running Time
 - 2.2.1 O-notation
 - 2.2.2 Avarage Case
- 2.3 Worst Case

Chap. III Basic DS

- 3.1 Stacks
- 3.2 Queues



PreOrder

*gjør jobben
FØR
rekursive kall*

```
DFS(Tree T, Position v)
Print(v.element())
for hver p i T.children(v)
  DFS(T,p)
```

I-120 bok

Chap. I Design Principles

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

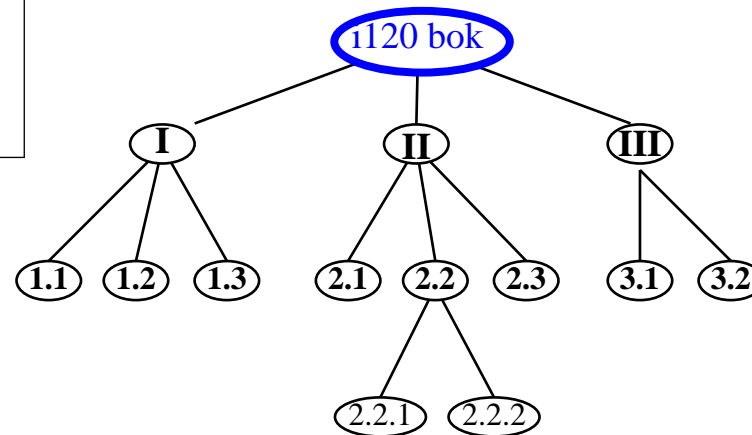
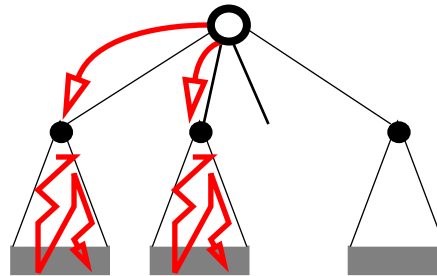
Chap. II Analysis Tools

- 2.1 Alg. Analysis
- 2.2 Running Time
 - 2.2.1 O-notation
 - 2.2.2 Avarage Case
- 2.3 Worst Case

Chap. III Basic DS

- 3.1 Stacks
- 3.2 Queues

DFS



PostOrder

*gjør jobben
ETTER
rekursive kall*

```
DFS(Tree T, Position v)

for hver p i T.children(v)
  DFS(T,p)
Print(v.element())
```

PreOrder

*gjør jobben
FØR
rekursive kall*

```
DFS(Tree T, Position v)
Print(v.element())
for hver p i T.children(v)
  DFS(T,p)
```

I-120 bok

Chap. I Design Principles

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

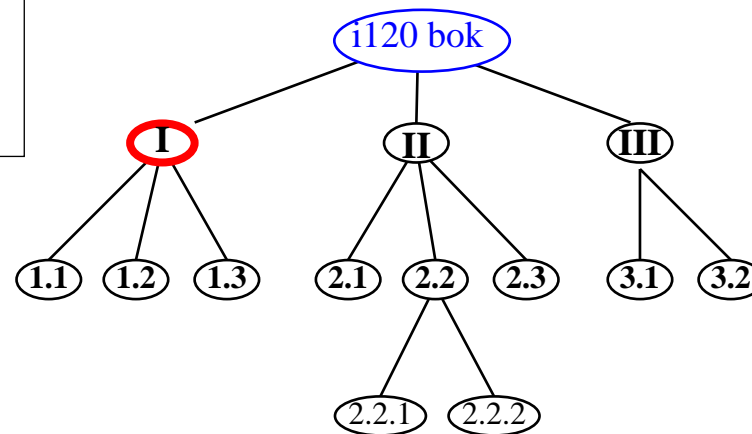
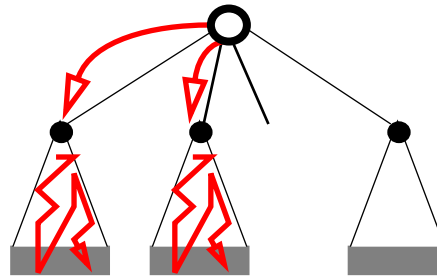
Chap. II Analysis Tools

- 2.1 Alg. Analysis
- 2.2 Running Time
 - 2.2.1 O-notation
 - 2.2.2 Average Case
- 2.3 Worst Case

Chap. III Basic DS

- 3.1 Stacks
- 3.2 Queues

DFS



PostOrder

*gjør jobben
ETTER
rekursive kall*

```
DFS(Tree T, Position v)

for hver p i T.children(v)
  DFS(T,p)
Print(v.element())
```

PreOrder

*gjør jobben
FØR
rekursive kall*

```
DFS(Tree T, Position v)
Print(v.element())
for hver p i T.children(v)
  DFS(T,p)
```

I-120 bok

Chap. I Design Principles

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

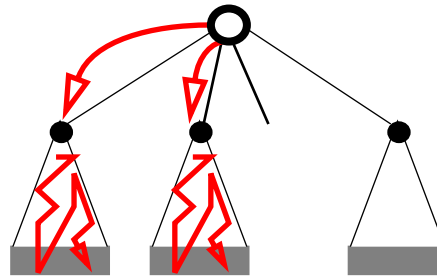
Chap. II Analysis Tools

- 2.1 Alg. Analysis
- 2.2 Running Time
 - 2.2.1 O-notation
 - 2.2.2 Avarage Case
- 2.3 Worst Case

Chap. III Basic DS

- 3.1 Stacks
- 3.2 Queues

DFS

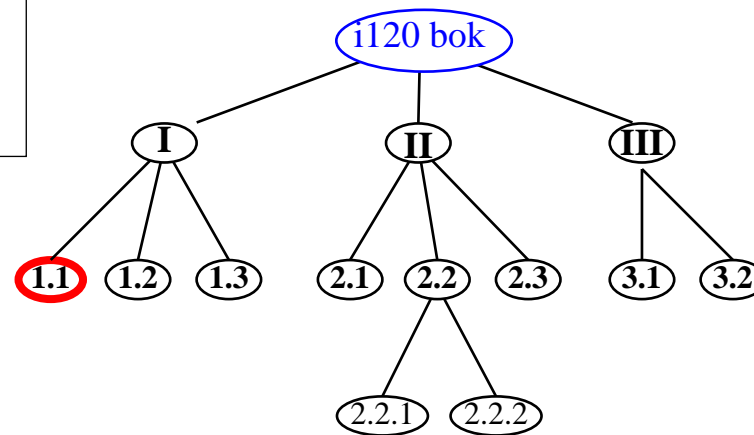


PostOrder

*gjør jobben
ETTER
rekursive kall*

```
DFS(Tree T, Position v)

for hver p i T.children(v)
  DFS(T,p)
Print(v.element())
```



PreOrder

*gjør jobben
FØR
rekursive kall*

```
DFS(Tree T, Position v)
Print(v.element())
for hver p i T.children(v)
  DFS(T,p)
```

I-120 bok

Chap. I Design Principles

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

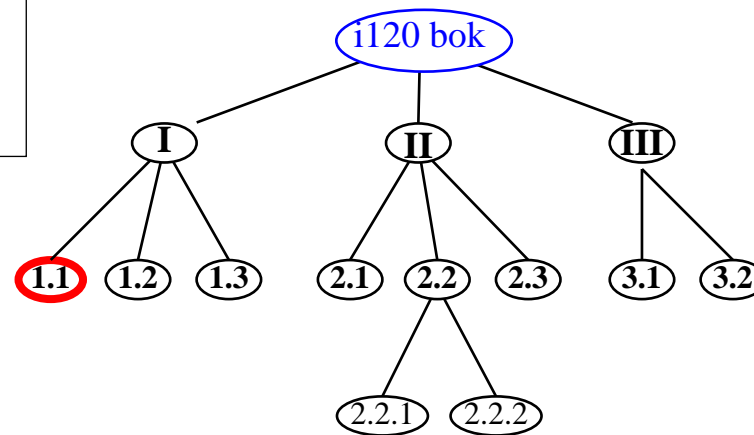
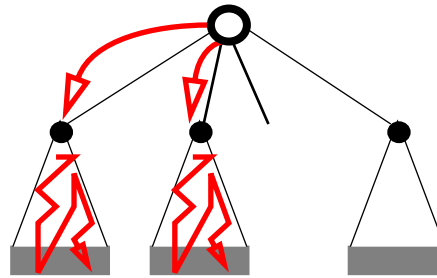
Chap. II Analysis Tools

- 2.1 Alg. Analysis
- 2.2 Running Time
 - 2.2.1 O-notation
 - 2.2.2 Avarage Case
- 2.3 Worst Case

Chap. III Basic DS

- 3.1 Stacks
- 3.2 Queues

DFS



PostOrder

*gjør jobben
ETTER
rekursive kall*

```
DFS(Tree T, Position v)

for hver p i T.children(v)
  DFS(T,p)
Print(v.element())
```

1.1 DS & Alg

PreOrder

*gjør jobben
FØR
rekursive kall*

```
DFS(Tree T, Position v)
Print(v.element())
for hver p i T.children(v)
  DFS(T,p)
```

I-120 bok

Chap. I Design Principles

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

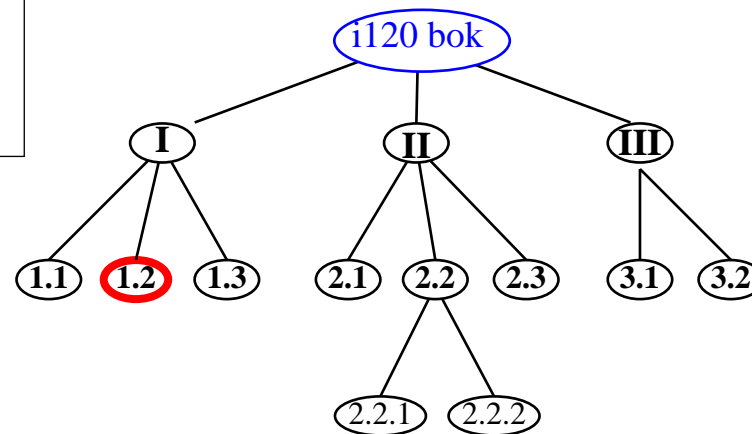
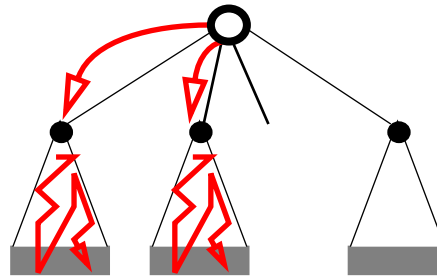
Chap. II Analysis Tools

- 2.1 Alg. Analysis
- 2.2 Running Time
 - 2.2.1 O-notation
 - 2.2.2 Avarage Case
- 2.3 Worst Case

Chap. III Basic DS

- 3.1 Stacks
- 3.2 Queues

DFS



PostOrder

*gjør jobben
ETTER
rekursive kall*

```
DFS(Tree T, Position v)

for hver p i T.children(v)
  DFS(T,p)
Print(v.element())
```

- 1.1 DS & Alg
- 1.2 OO-Programming

PreOrder

*gjør jobben
FØR
rekursive kall*

```
DFS(Tree T, Position v)
Print(v.element())
for hver p i T.children(v)
  DFS(T,p)
```

I-120 bok

Chap. I Design Principles

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

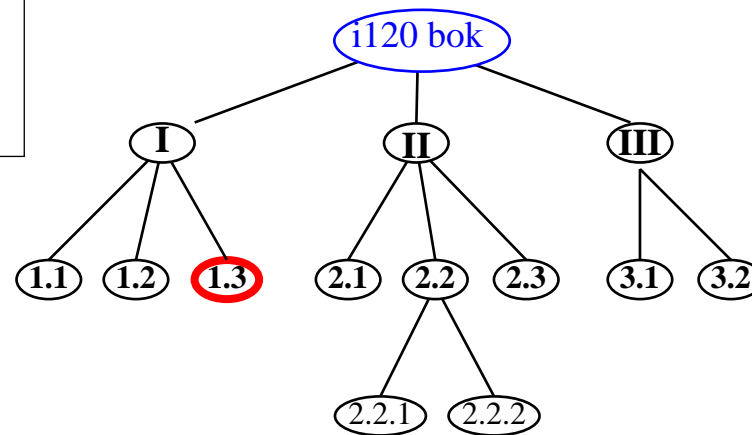
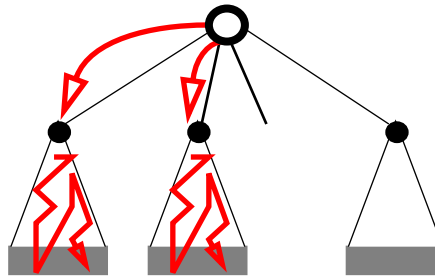
Chap. II Analysis Tools

- 2.1 Alg. Analysis
- 2.2 Running Time
 - 2.2.1 O-notation
 - 2.2.2 Avarage Case
- 2.3 Worst Case

Chap. III Basic DS

- 3.1 Stacks
- 3.2 Queues

DFS



PostOrder

*gjør jobben
ETTER
rekursive kall*

```
DFS(Tree T, Position v)

for hver p i T.children(v)
  DFS(T,p)
Print(v.element())
```

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

PreOrder

*gjør jobben
FØR
rekursive kall*

```
DFS(Tree T, Position v)
Print(v.element())
for hver p i T.children(v)
  DFS(T,p)
```

I-120 bok

Chap. I Design Principles

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

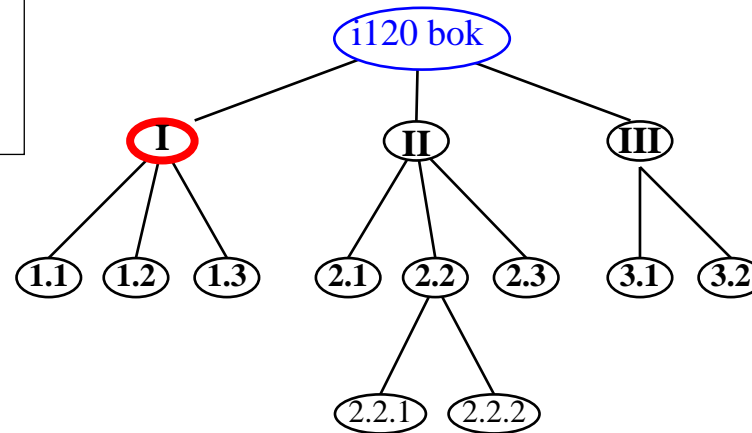
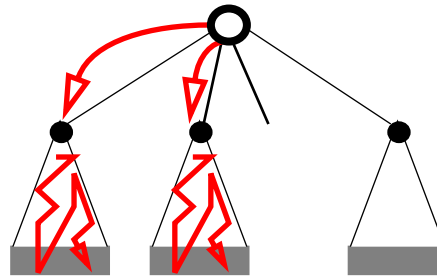
Chap. II Analysis Tools

- 2.1 Alg. Analysis
- 2.2 Running Time
 - 2.2.1 O-notation
 - 2.2.2 Avarage Case
- 2.3 Worst Case

Chap. III Basic DS

- 3.1 Stacks
- 3.2 Queues

DFS



PostOrder

*gjør jobben
ETTER
rekursive kall*

```
DFS(Tree T, Position v)

for hver p i T.children(v)
  DFS(T,p)
Print(v.element())
```

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

Chap. I Design Principles

PreOrder

*gjør jobben
FØR
rekursive kall*

```
DFS(Tree T, Position v)
Print(v.element())
for hver p i T.children(v)
  DFS(T,p)
```

I-120 bok

Chap. I Design Principles

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

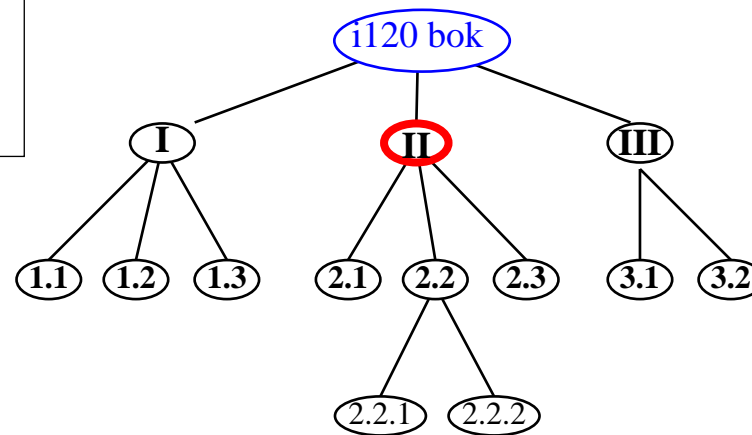
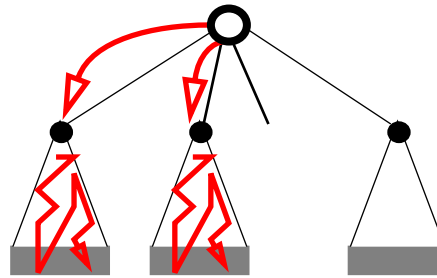
Chap. II Analysis Tools

- 2.1 Alg. Analysis
- 2.2 Running Time
 - 2.2.1 O-notation
 - 2.2.2 Avarage Case
- 2.3 Worst Case

Chap. III Basic DS

- 3.1 Stacks
- 3.2 Queues

DFS



PostOrder

*gjør jobben
ETTER
rekursive kall*

```
DFS(Tree T, Position v)

for hver p i T.children(v)
  DFS(T,p)
Print(v.element())
```

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

Chap. I Design Principles

PreOrder

*gjør jobben
FØR
rekursive kall*

```
DFS(Tree T, Position v)
Print(v.element())
for hver p i T.children(v)
    DFS(T,p)
```

I-120 bok

Chap. I Design Principles

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

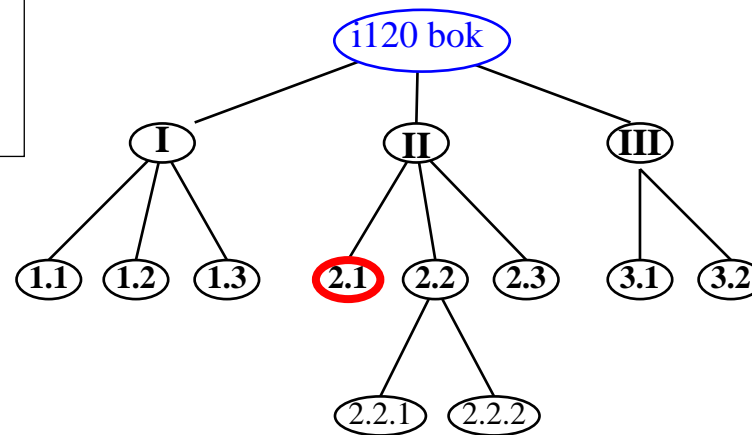
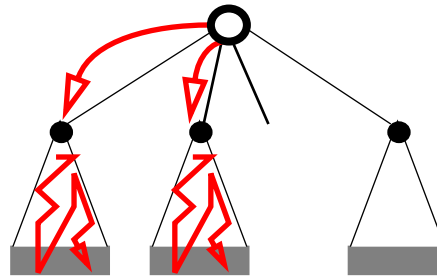
Chap. II Analysis Tools

- 2.1 Alg. Analysis
- 2.2 Running Time
 - 2.2.1 O-notation
 - 2.2.2 Avarage Case
- 2.3 Worst Case

Chap. III Basic DS

- 3.1 Stacks
- 3.2 Queues

DFS



PostOrder

*gjør jobben
ETTER
rekursive kall*

```
DFS(Tree T, Position v)

for hver p i T.children(v)
    DFS(T,p)
Print(v.element())
```

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

Chap. I Design Principles

PreOrder

*gjør jobben
FØR
rekursive kall*

```
DFS(Tree T, Position v)
Print(v.element())
for hver p i T.children(v)
  DFS(T,p)
```

I-120 bok

Chap. I Design Principles

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

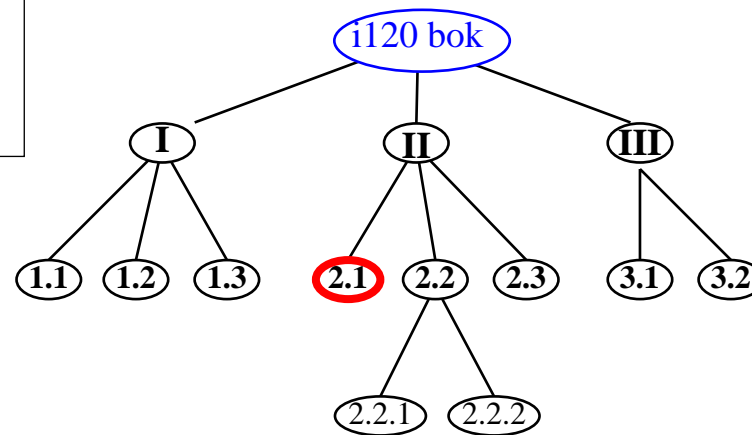
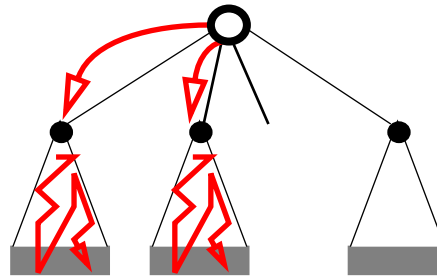
Chap. II Analysis Tools

- 2.1 Alg. Analysis
- 2.2 Running Time
 - 2.2.1 O-notation
 - 2.2.2 Avarage Case
- 2.3 Worst Case

Chap. III Basic DS

- 3.1 Stacks
- 3.2 Queues

DFS



PostOrder

*gjør jobben
ETTER
rekursive kall*

```
DFS(Tree T, Position v)

for hver p i T.children(v)
  DFS(T,p)
Print(v.element())
```

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

Chap. I Design Principles

- 2.1 Alg. Analysis

PreOrder

*gjør jobben
FØR
rekursive kall*

```
DFS(Tree T, Position v)
Print(v.element())
for hver p i T.children(v)
  DFS(T,p)
```

I-120 bok

Chap. I Design Principles

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

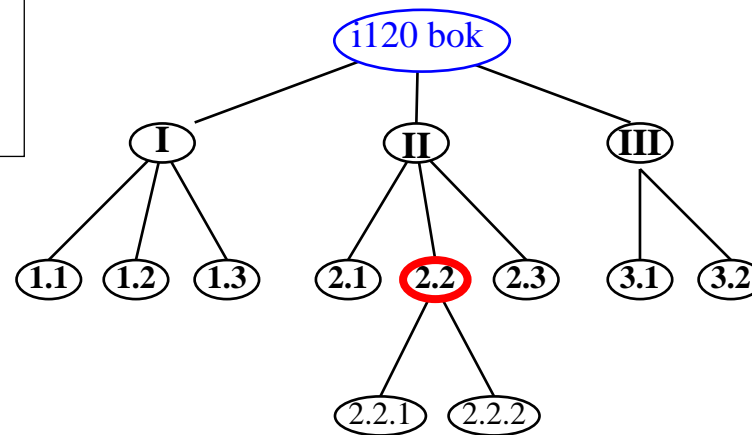
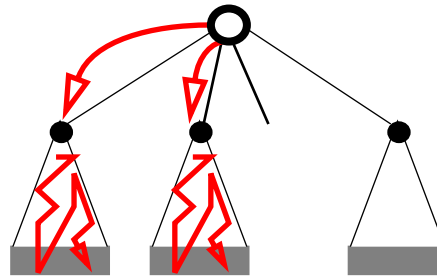
Chap. II Analysis Tools

- 2.1 Alg. Analysis
- 2.2 Running Time
 - 2.2.1 O-notation
 - 2.2.2 Avarage Case
- 2.3 Worst Case

Chap. III Basic DS

- 3.1 Stacks
- 3.2 Queues

DFS



PostOrder

*gjør jobben
ETTER
rekursive kall*

```
DFS(Tree T, Position v)
for hver p i T.children(v)
  DFS(T,p)
Print(v.element())
```

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

Chap. I Design Principles

- 2.1 Alg. Analysis

PreOrder

*gjør jobben
FØR
rekursive kall*

```
DFS(Tree T, Position v)
Print(v.element())
for hver p i T.children(v)
    DFS(T,p)
```

I-120 bok

Chap. I Design Principles

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

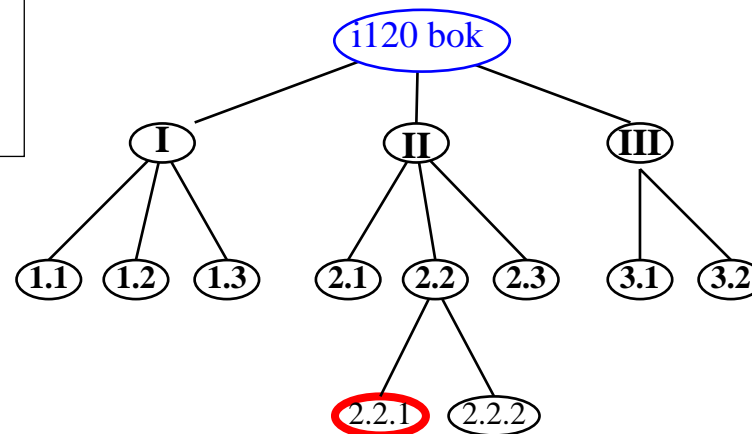
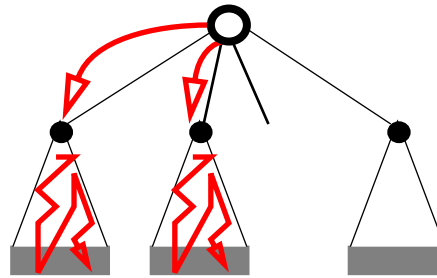
Chap. II Analysis Tools

- 2.1 Alg. Analysis
- 2.2 Running Time
 - 2.2.1 O-notation
 - 2.2.2 Avarage Case
- 2.3 Worst Case

Chap. III Basic DS

- 3.1 Stacks
- 3.2 Queues

DFS



PostOrder

*gjør jobben
ETTER
rekursive kall*

```
DFS(Tree T, Position v)

for hver p i T.children(v)
    DFS(T,p)
Print(v.element())
```

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

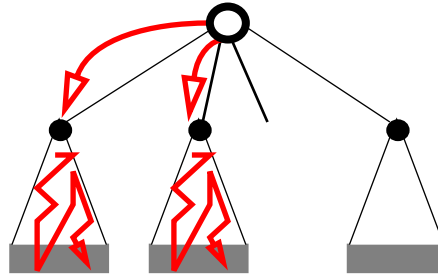
Chap. I Design Principles

- 2.1 Alg. Analysis

PreOrder

*gjør jobben
FØR
rekursive kall*

DFS



PostOrder

*gjør jobben
ETTER
rekursive kall*

DFS(Tree T, Position v)

Print(v.element())

for hver p i T.children(v)

DFS(T,p)

DFS(Tree T, Position v)

for hver p i T.children(v)

DFS(T,p)

Print(v.element())

I-120 bok

Chap. I Design Principles

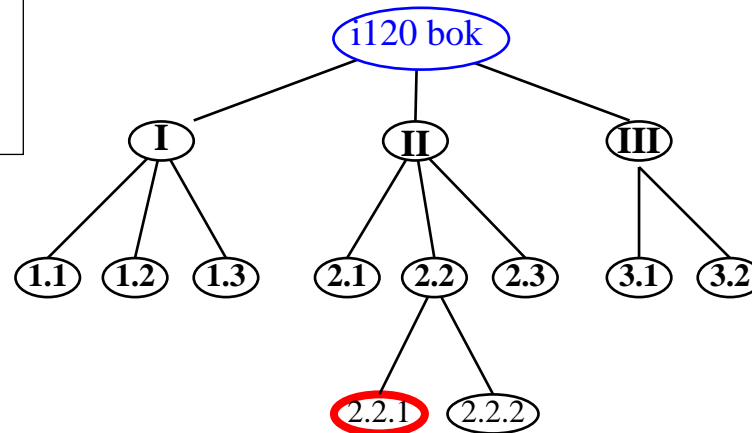
- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

Chap. II Analysis Tools

- 2.1 Alg. Analysis
- 2.2 Running Time
 - 2.2.1 O-notation
 - 2.2.2 Avarage Case
- 2.3 Worst Case

Chap. III Basic DS

- 3.1 Stacks
- 3.2 Queues



1.1 DS & Alg

1.2 OO-Programming

1.3 JAVA

Chap. I Design Principles

2.1 Alg. Analysis

2.2.1 O-notation

PreOrder

*gjør jobben
FØR
rekursive kall*

```
DFS(Tree T, Position v)
Print(v.element())
for hver p i T.children(v)
  DFS(T,p)
```

I-120 bok

Chap. I Design Principles

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

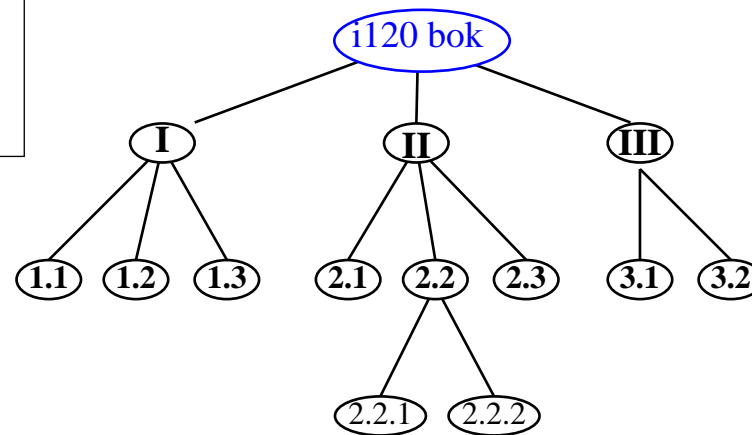
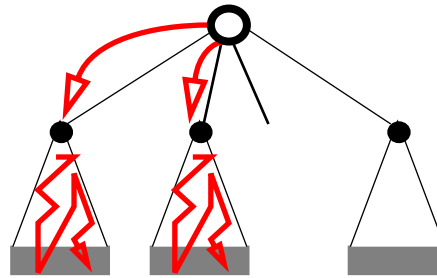
Chap. II Analysis Tools

- 2.1 Alg. Analysis
- 2.2 Running Time
 - 2.2.1 O-notation
 - 2.2.2 Avarage Case
- 2.3 Worst Case

Chap. III Basic DS

- 3.1 Stacks
- 3.2 Queues

DFS



PostOrder

*gjør jobben
ETTER
rekursive kall*

```
DFS(Tree T, Position v)

for hver p i T.children(v)
  DFS(T,p)
Print(v.element())
```

- 1.1 DS & Alg
- 1.2 OO-Programming
- 1.3 JAVA

Chap. I Design Principles

- 2.1 Alg. Analysis
 - 2.2.1 O-notation
 - 2.2.2 Avarage Case
- 2.2 Running Time
- 2.3 Worst Case

Chap. II Analysis Tools

- 3.1 Stacks
- 3.2 Queues

Chap. III Basic DS

I-120 bok

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! – pre

if (isInternal(v))

DFS(B,B.leftChild(v))

DFS(B,B.rightChild(v))

!!! – post

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! – pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! – in-order

DFS(B,B.rightChild(v))

!!! – post

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

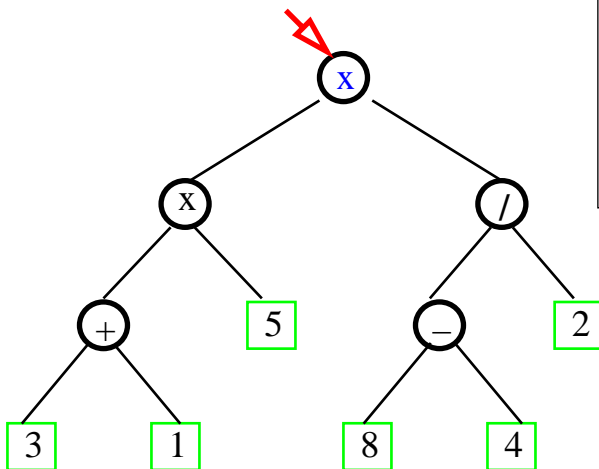
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

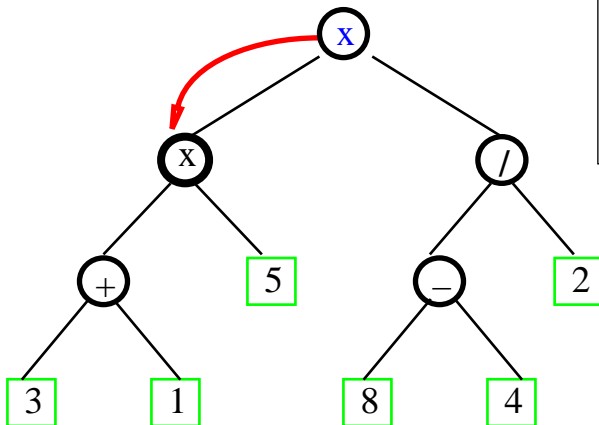
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

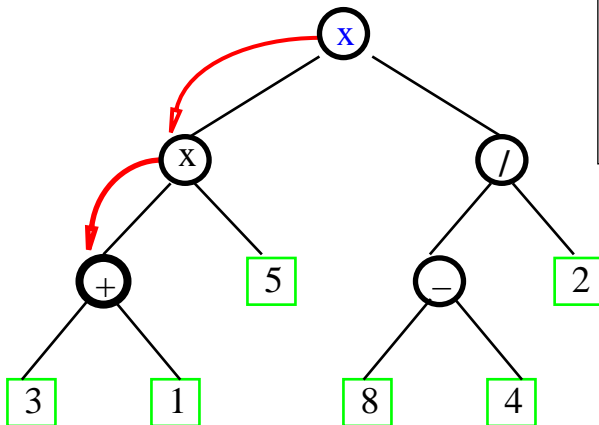
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (**B.isExternal(v)**) **!!! - ekst**

else

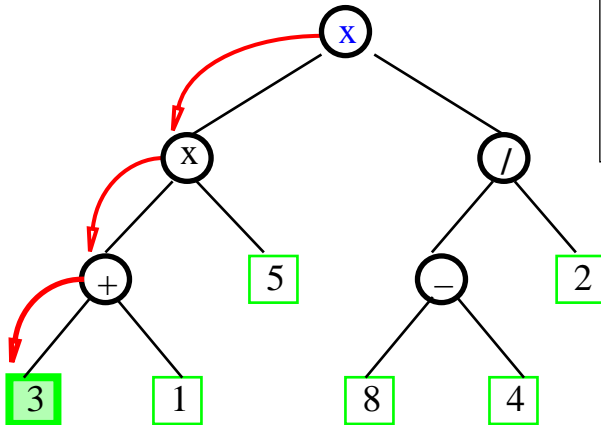
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

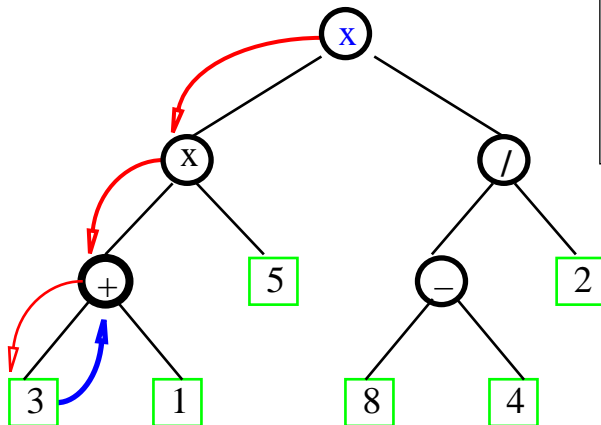
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

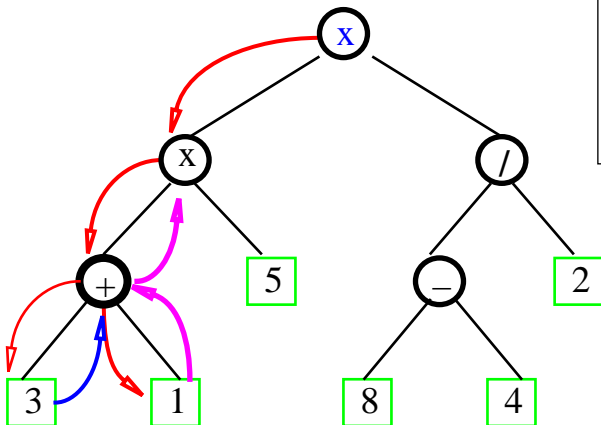
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

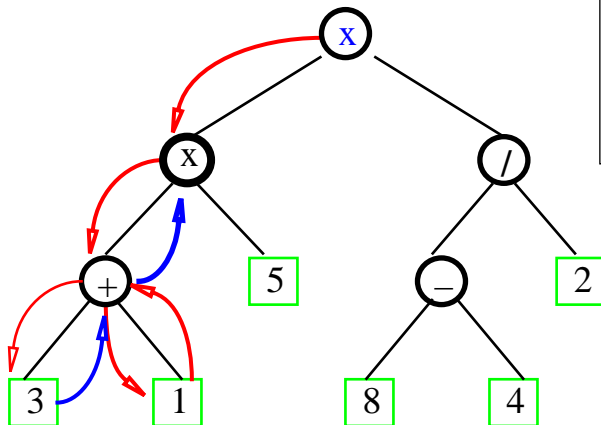
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! – pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! – in-order

DFS(B,B.rightChild(v))

!!! – post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! – ekst**

else

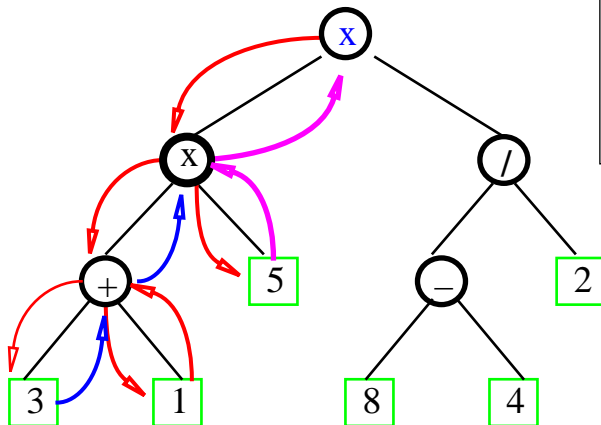
!!! – pre

DFS(B,B.leftChild(v))

!!! – in-order

DFS(B,B.rightChild(v))

!!! – post



Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

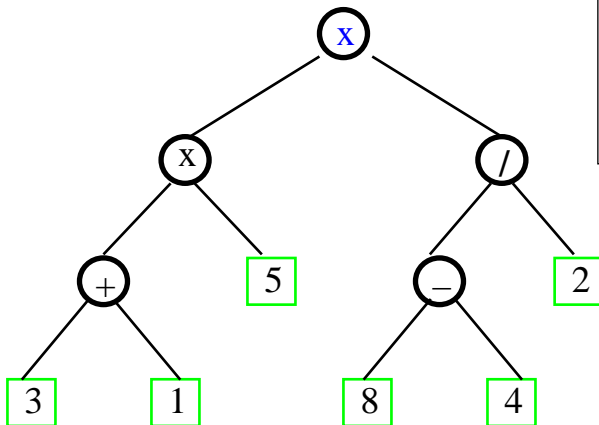
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



void **Pr**(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("("

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(")

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

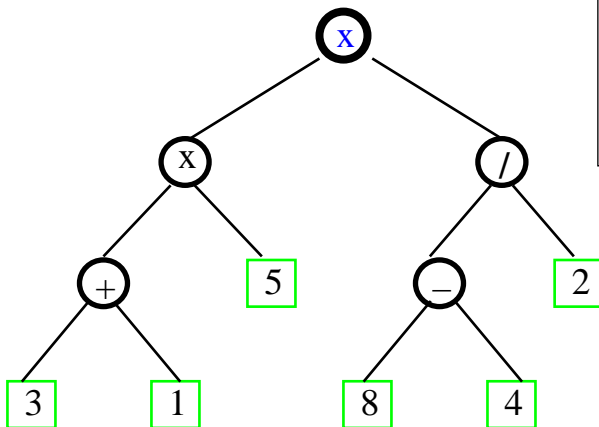
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



void Pr(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("("

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(")

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

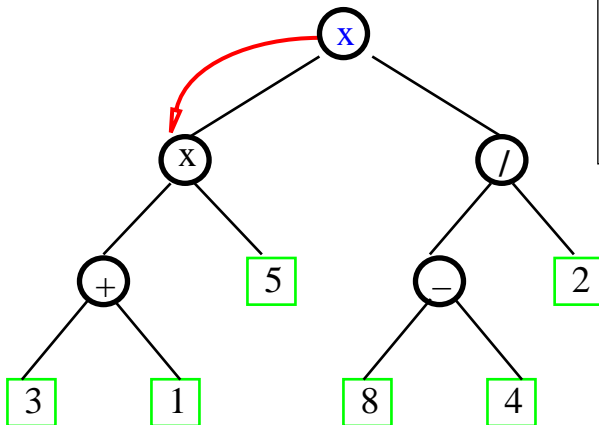
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



void Pr(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("(")

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(")")

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

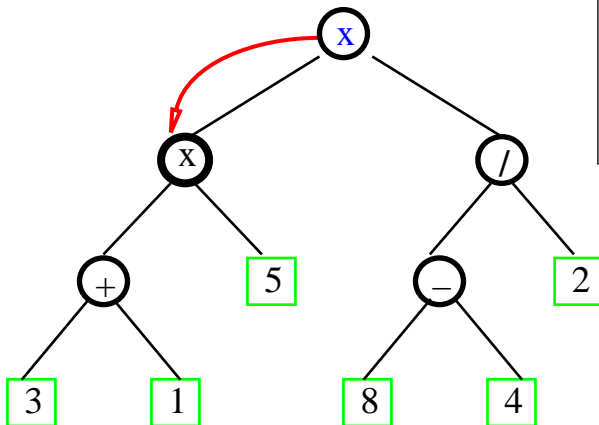
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



void Pr(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("(")

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(")")

((

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

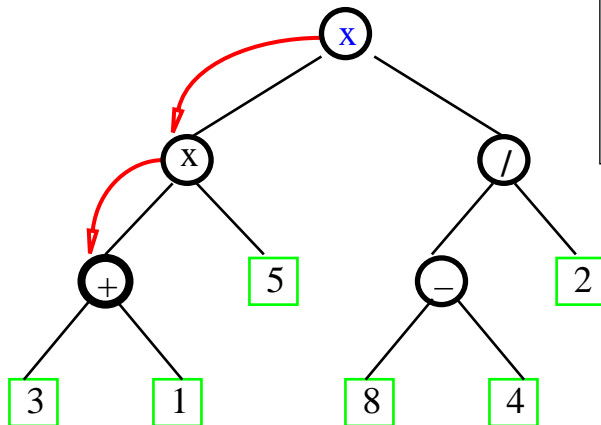
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



(((

void Pr(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("("

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(")

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

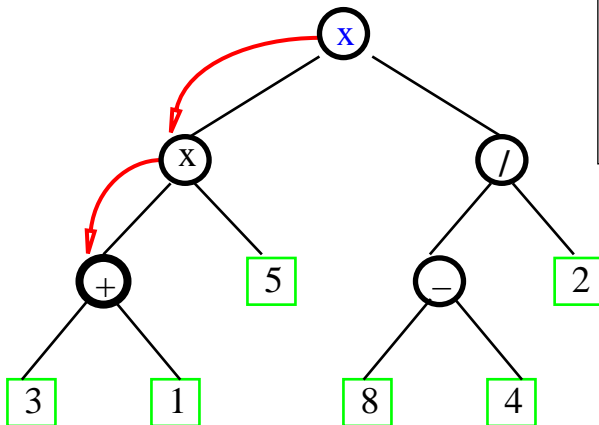
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



(((

void Pr(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("("

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(")

Binære Trær : Euler-Tour & InOrder

```
DFS(Tree T, Position v)
```

!!! - pre

for hver p i T.children(v)

 DFS(T,p)

!!! - post

```
DFS(BinTree B, Position v)
```

!!! - pre

if (isInternal(v))

 DFS(B,B.leftChild(v))

!!! - in-order

 DFS(B,B.rightChild(v))

!!! - post

```
DFS(BinTree B, Position v)
```

if (B.isExternal(v)) !!! - ekst

else

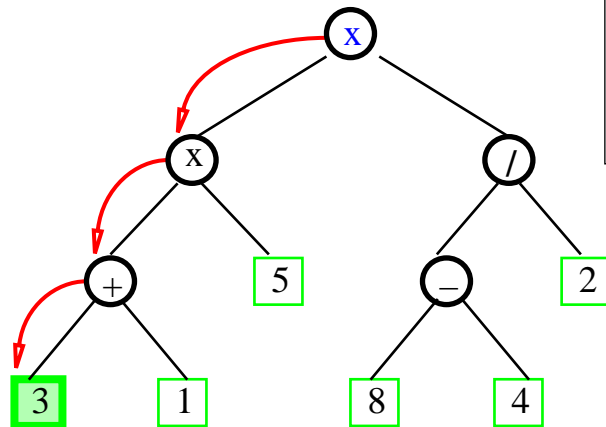
 !!! - pre

 DFS(B,B.leftChild(v))

 !!! - in-order

 DFS(B,B.rightChild(v))

 !!! - post



(((3

```
void Pr(BinTree B, Position v)
```

if (B.isExternal(v)) **print(v.element())**

else

print("("

 Pr(B,B.leftChild(v))

print(v.element())

 Pr(B,B.rightChild(v))

print(")

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

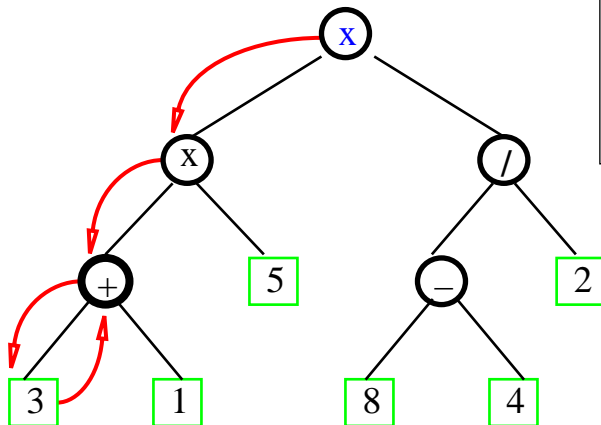
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



(((3 +

void Pr(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("(")

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(")

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

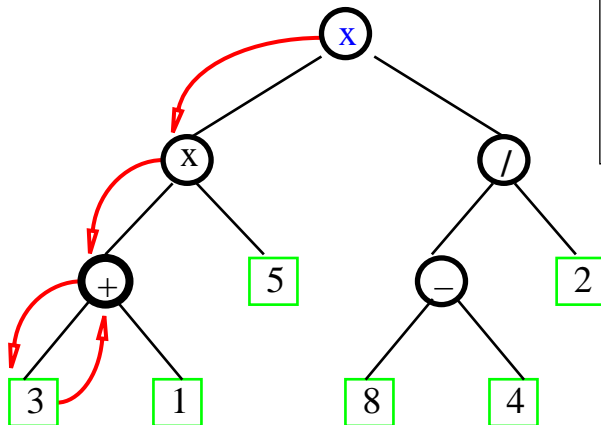
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



(((3 +

void Pr(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("("

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(")

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

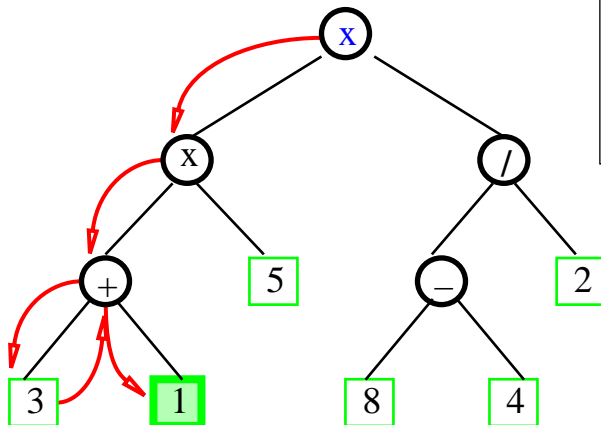
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



(((3 + 1

void Pr(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("("

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(")

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

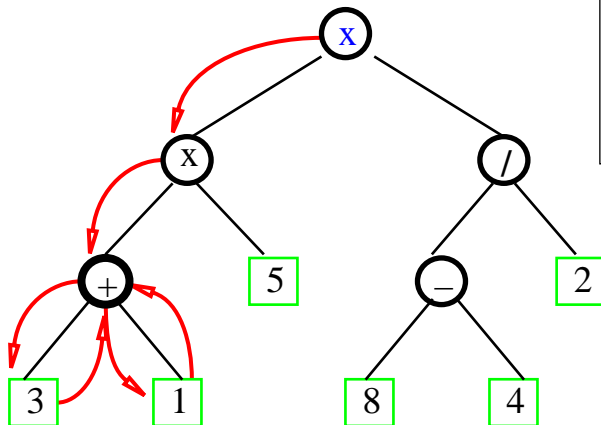
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



(((3 + 1

void Pr(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("("

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(")

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

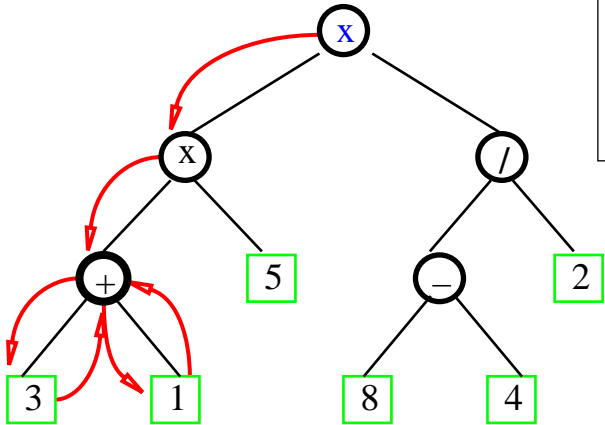
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



(((3 + 1)

void Pr(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("("

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(")")

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

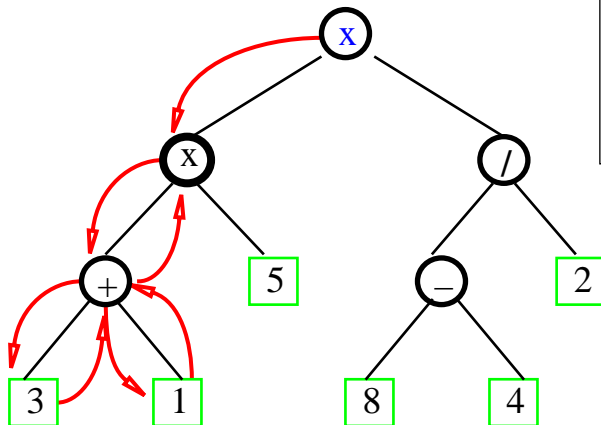
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



(((3 + 1)

void Pr(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("("

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(")

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

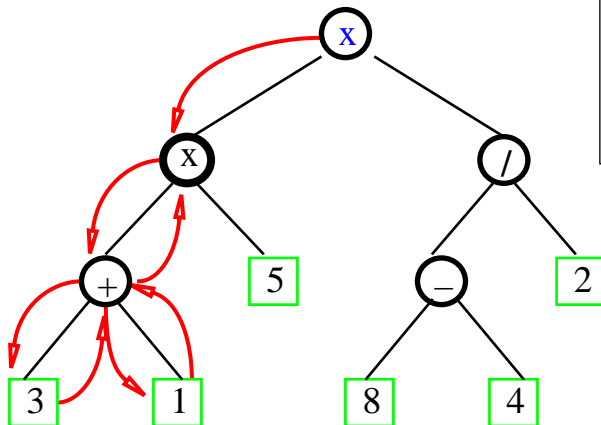
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



(((3 + 1) x

void Pr(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("(")

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(")

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

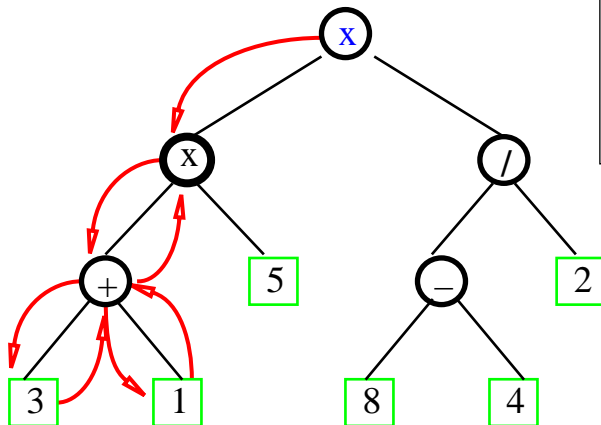
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



(((3 + 1) x

void Pr(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("(")

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(")")

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

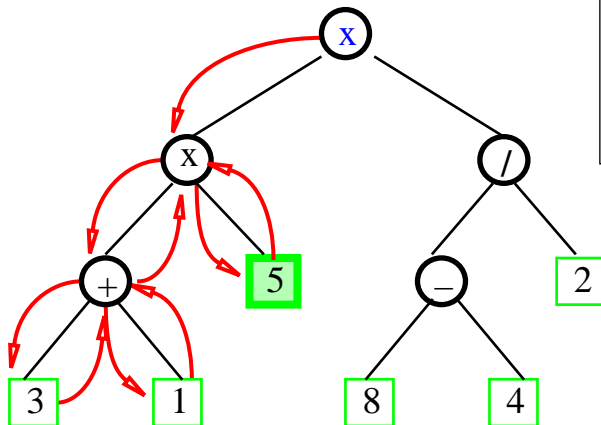
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



(((3 + 1) x 5

void Pr(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("(")

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(")")

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

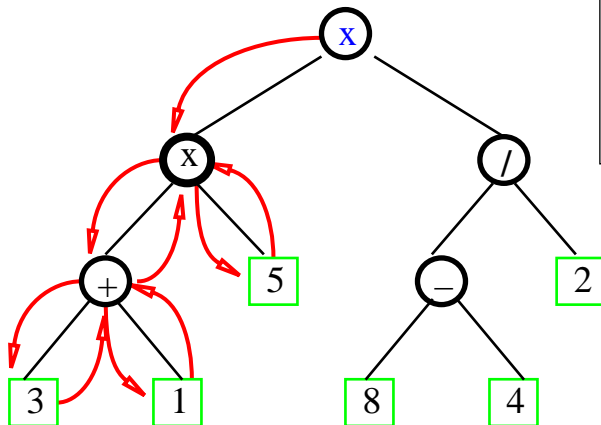
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



(((3 + 1) x 5) / (8 - 4))

void Pr(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("(")

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(")")

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

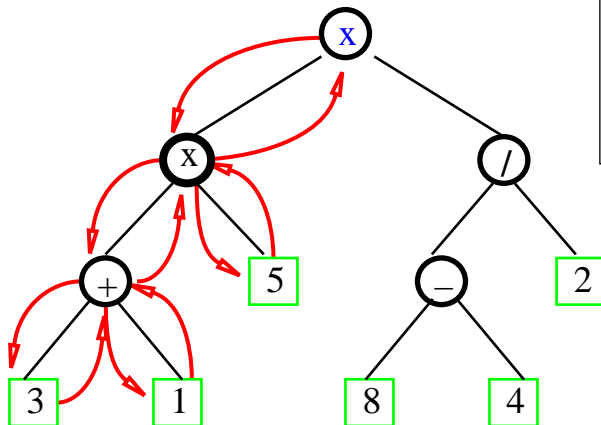
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



(((3 + 1) x 5)

void Pr(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("(")

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(" ")

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

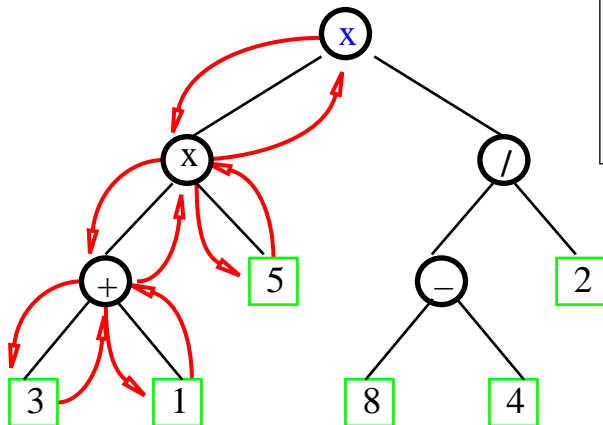
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



(((3 + 1) x 5) x ((8 - 4) / 2))

void **Pr**(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("("

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(")

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) !!! - ekst

else

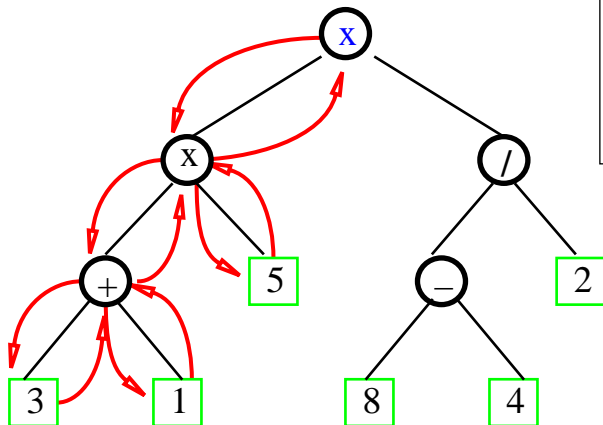
 !!! - pre

DFS(B,B.leftChild(v))

 !!! - in-order

DFS(B,B.rightChild(v))

 !!! - post



(((3 + 1) x 5) x ((8 - 4) / 2))

void **Pr**(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print(v.element())

Pr(B,B.leftChild(v))

Pr(B,B.rightChild(v))

P

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) !!! - ekst

else

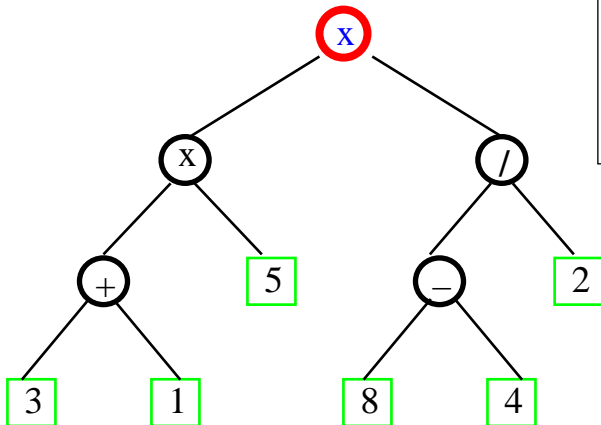
 !!! - pre

DFS(B,B.leftChild(v))

 !!! - in-order

DFS(B,B.rightChild(v))

 !!! - post



(((3 + 1) x 5) x ((8 - 4) / 2))

void **Pr**(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("P

print(v.element())

Pr(B,B.leftChild(v))

Pr(B,B.rightChild(v))

P

print(")

x

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) !!! - ekst

else

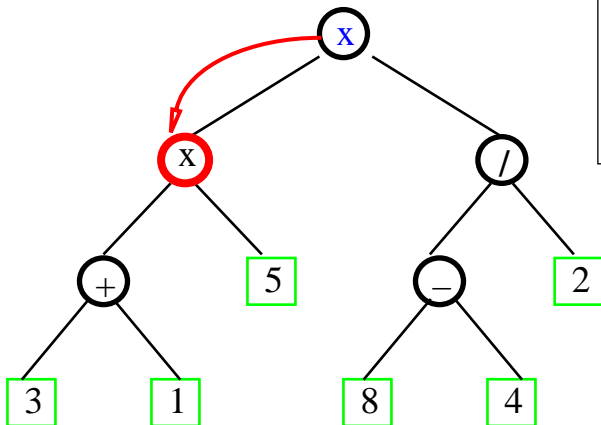
 !!! - pre

DFS(B,B.leftChild(v))

 !!! - in-order

DFS(B,B.rightChild(v))

 !!! - post



$(((3 + 1) \times 5) \times ((8 - 4) / 2))$

void **Pr**(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("P

print(v.element())

Pr(B,B.leftChild(v))

Pr(B,B.rightChild(v))

P

print(")

x x

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) !!! - ekst

else

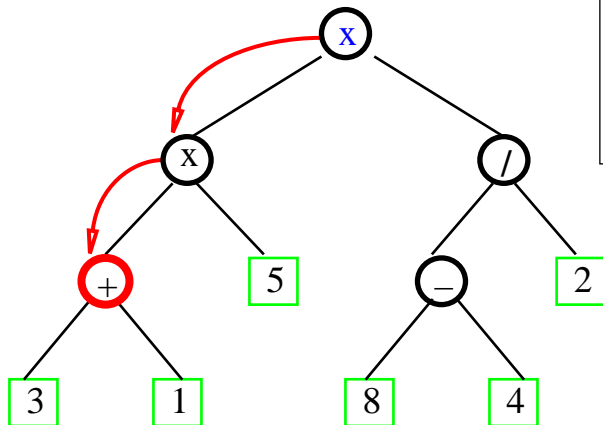
 !!! - pre

DFS(B,B.leftChild(v))

 !!! - in-order

DFS(B,B.rightChild(v))

 !!! - post



(((3 + 1) x 5) x ((8 - 4) / 2))

void **Pr**(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print(v.element())

Pr(B,B.leftChild(v))

Pr(B,B.rightChild(v))

print(v.element())

x x +

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) !!! - ekst

else

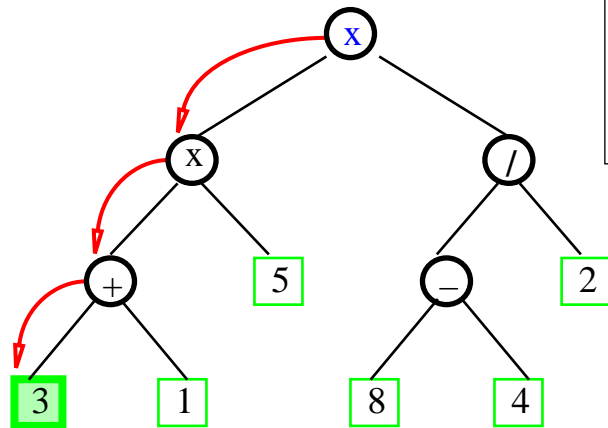
 !!! - pre

DFS(B,B.leftChild(v))

 !!! - in-order

DFS(B,B.rightChild(v))

 !!! - post



(((3 + 1) x 5) x ((8 - 4) / 2))

void **Pr**(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print(v.element())

Pr(B,B.leftChild(v))

Pr(B,B.rightChild(v))

print(v.element())

x x + 3

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

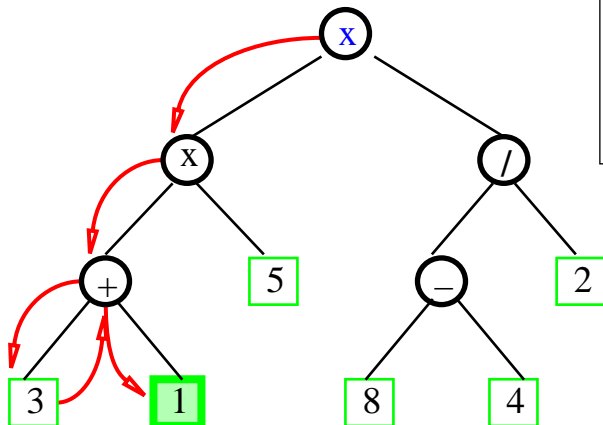
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



$(((3 + 1) \times 5) \times ((8 - 4) / 2))$

void **Pr**(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print

print(v.element())

Pr(B,B.leftChild(v))

Pr(B,B.rightChild(v))

x x + 3 1

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) **!!! - ekst**

else

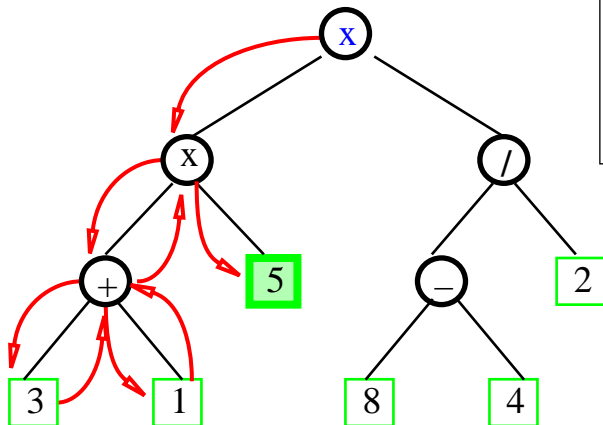
!!! - pre

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post



$(((3 + 1) \times 5) \times ((8 - 4) / 2))$

void **Pr**(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print

print(v.element())

Pr(B,B.leftChild(v))

Pr(B,B.rightChild(v))

x x + 3 1 5

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) !!! - ekst

else

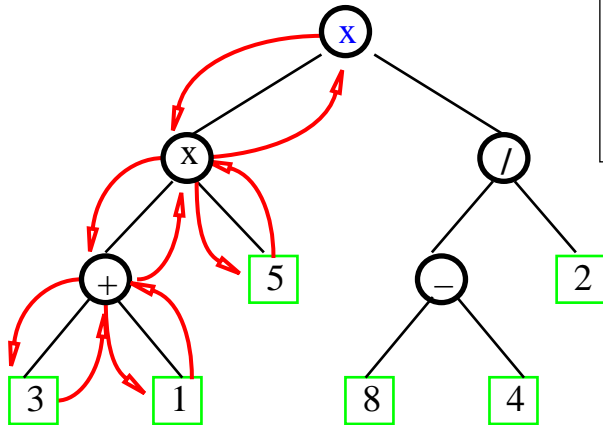
 !!! - pre

DFS(B,B.leftChild(v))

 !!! - in-order

DFS(B,B.rightChild(v))

 !!! - post



(((3 + 1) * 5) * ((8 - 4) / 2))

void **Pr**(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print(v.element())

Pr(B,B.leftChild(v))

Pr(B,B.rightChild(v))

print(v.element())

x x + 3 1 5

Binære Trær : Euler-Tour & InOrder

```

DFS(Tree T, Position v)
  !!! - pre
  for hver p i T.children(v)
    DFS(T,p)
  !!! - post
  
```

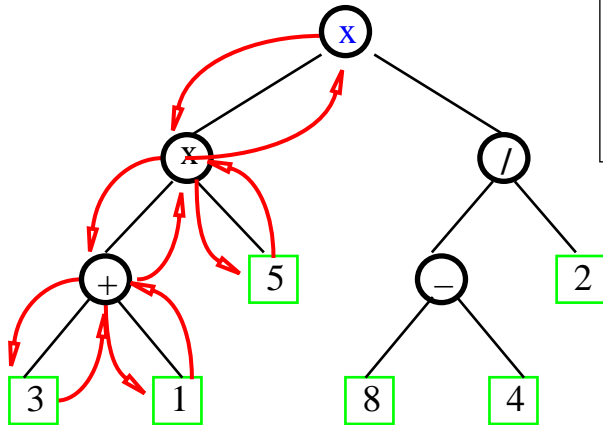
```

DFS(BinTree B, Position v)
  !!! - pre
  if (isInternal(v))
    DFS(B,B.leftChild(v))
  !!! - in-order

  DFS(B,B.rightChild(v))
  !!! - post
  
```

```

DFS(BinTree B, Position v)
  if (B.isExternal(v)) !!! - ekst
  else
    !!! - pre
    DFS(B,B.leftChild(v))
    !!! - in-order
    DFS(B,B.rightChild(v))
    !!! - post
  
```



$(((3 + 1) \times 5) \times ((8 - 4) / 2))$

```

void Pr(BinTree B, Position v)
  if (B.isExternal(v)) print(v.element())
  else
  
```

```

    print(v.element())
    Pr(B,B.leftChild(v))
    Pr(B,B.rightChild(v))
  
```

$x \ x + 3 \ 1 \ 5 \ / \ - \ 8 \ 4 \ 2$
 $x [x (+ (3 \ 1) \ 5) / (- (8 \ 4) \ 2)]$

Binære Trær : Euler-Tour & InOrder

DFS(Tree T, Position v)

!!! - pre

for hver p i T.children(v)

DFS(T,p)

!!! - post

DFS(BinTree B, Position v)

!!! - pre

if (isInternal(v))

DFS(B,B.leftChild(v))

!!! - in-order

DFS(B,B.rightChild(v))

!!! - post

DFS(BinTree B, Position v)

if (B.isExternal(v)) !!! - ekst

else

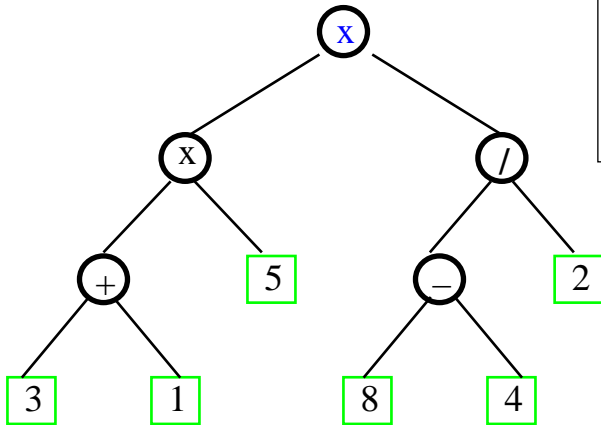
 !!! - pre

DFS(B,B.leftChild(v))

 !!! - in-order

DFS(B,B.rightChild(v))

 !!! - post



$(((3 + 1) \times 5) \times ((8 - 4) / 2))$

void Pr(BinTree B, Position v)

if (B.isExternal(v)) **print(v.element())**

else

print("(")

Pr(B,B.leftChild(v))

print(v.element())

Pr(B,B.rightChild(v))

print(" ")

print(v.element())

Pr(B,B.leftChild(v))

Pr(B,B.rightChild(v))

x x + 3 1 5 / - 8 4 2

x [x(+ (3,1), 5), / (-(8,4),2)]

Binære Trær : Euler-Tour & InOrder

```

DFS(Tree T, Position v)
  !!! - pre
  for hver p i T.children(v)
    DFS(T,p)
  !!! - post
  
```

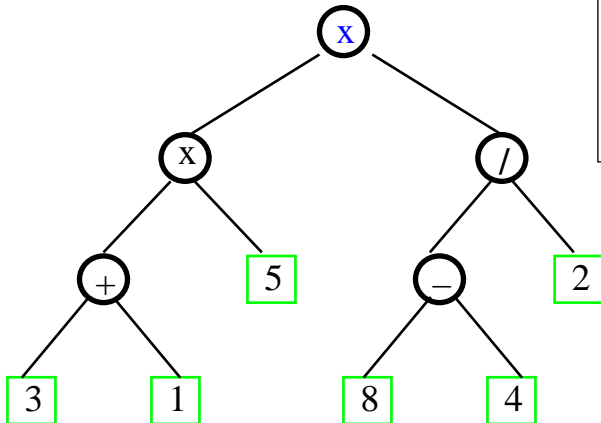
```

DFS(BinTree B, Position v)
  !!! - pre
  if (isInternal(v))
    DFS(B,B.leftChild(v))
  !!! - in-order

  DFS(B,B.rightChild(v))
  !!! - post
  
```

```

DFS(BinTree B, Position v)
  if (B.isExternal(v)) !!! - ekst
  else
    !!! - pre
    DFS(B,B.leftChild(v))
    !!! - in-order
    DFS(B,B.rightChild(v))
    !!! - post
  
```



$(((3 + 1) \times 5) \times ((8 - 4) / 2))$

```

void Pr(BinTree B, Position v)
  if (B.isExternal(v)) print(v.element())
  else
    print("(")
    Pr(B,B.leftChild(v))
    print(v.element())
    Pr(B,B.rightChild(v))
    print(")")
  
```

```

print(v.element())
Pr(B,B.leftChild(v))
Pr(B,B.rightChild(v))
  
```

x x + 3 1 5 / - 8 4 2
 x [x(+ (3,1), 5), / (-(8,4),2)]

```

int val(BinTree B, Position v)
  if (B.isExternal(v))
    return integer(v)
  else
    L= val(B,B.leftChild(v))
    R= val(B,B.rightChild(v))
    return operasjon(v)(L,R)
  
```

Binære Trær : Euler-Tour & InOrder

```

DFS(Tree T, Position v)
  !!! - pre
  for hver p i T.children(v)
    DFS(T,p)
  !!! - post
  
```

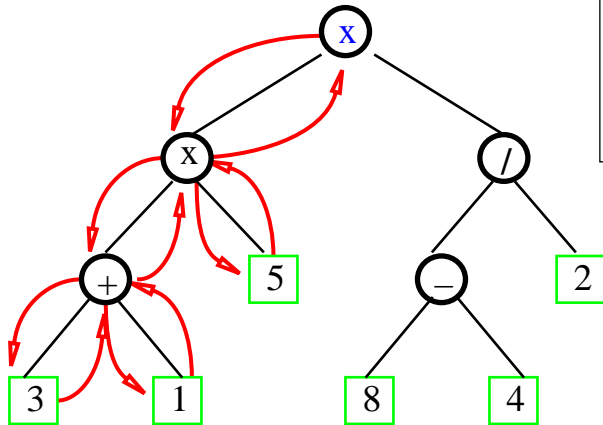
```

DFS(BinTree B, Position v)
  !!! - pre
  if (isInternal(v))
    DFS(B,B.leftChild(v))
  !!! - in-order

  DFS(B,B.rightChild(v))
  !!! - post
  
```

```

DFS(BinTree B, Position v)
  if (B.isExternal(v)) !!! - ekst
  else
    !!! - pre
    DFS(B,B.leftChild(v))
    !!! - in-order
    DFS(B,B.rightChild(v))
    !!! - post
  
```



$$(((3 + 1) \times 5) \times ((8 - 4) / 2)) = 40$$

```

void Pr(BinTree B, Position v)
  if (B.isExternal(v)) print(v.element())
  else
    print("(")
    Pr(B,B.leftChild(v))
    print(v.element())
    Pr(B,B.rightChild(v))
    print(")")
  
```

```

print(v.element())
Pr(B,B.leftChild(v))
Pr(B,B.rightChild(v))
  
```

x x + 3 1 5 / - 8 4 2
 x [x(+ (3,1), 5), / (-(8,4),2)]

```

int val(BinTree B, Position v)
  if (B.isExternal(v))
    return integer(v)
  else
    L= val(B,B.leftChild(v))
    R= val(B,B.rightChild(v))
    return operasjon(v)(L,R)
  
```

package jdsl.core.api;

```
public interface InspectableContainer
{ Enumeration elements();
  boolean isEmpty();
  int size();
  Container newCont
}
```

```
public interface InspectablePositionalContainer extends Container
{ Enumeration positions();
  void swap(Position p, Position q);
  Object replace(Position p, Object e);
}
```

```
/** Et tre har minst en ekstern node – rot */
public interface InspectableTree
  extends PositionalContainer {
  Position root();
  Position parent(Position v);
  /** Barn er ordnet fra venstre til høyre */
  PositionIterator children(Position p);
  boolean isInternal(Position v);
  boolean isExternal(Position v);
  boolean isRoot(Position v);
  PositionIterator siblings(Position p);
}
```

```
public interface InspectableBinaryTree
  extends InspectableTree {
  Position leftChild(Position);
  Position rightChild(Position);
  Position sibling(Position); }
```

package jdsl.core.api;

```
public interface InspectableContainer
{ Enumeration elements();
  boolean isEmpty();
  int size();
  Container newCont
}
```

```
public interface InspectablePositionalContainer extends Container
{ Enumeration positions();
  void swap(Position p, Position q);
  Object replace(Position p, Object e);
}
```

```
/** Et tre har minst en ekstern node – rot */
public interface InspectableTree
  extends PositionalContainer {
  Position root();
  Position parent(Position v);
  /** Barn er ordnet fra venstre til høyre */
  PositionIterator children(Position p);
  boolean isInternal(Position v);
  boolean isExternal(Position v);
  boolean isRoot(Position v);
  PositionIterator siblings(Position p);
}
```

```
public interface Container
{ Object replaceElement
  (Position p, Object o);
}
```

```
public interface InspectableBinaryTree
  extends InspectableTree {
  Position leftChild(Position);
  Position rightChild(Position);
  Position sibling(Position); }
}
```

```
interface PositionalContainer
swapElements(Position p, q);
}
```

```
public interface Tree extends InspectableTree {
  /** Node p erstattes med en ny ekstern node
   * med null element. */
  Tree cut(Position p);
  /** Node p (må være ekstern) erstattes med treet t */
  void link(Position p, Tree t);
  /** Erstatt subtre i node p med et nytt subtre t */
  Tree replaceSubtree(Position p, Tree t) }
}
```

package jdsl.core.api;

```
public interface InspectableContainer
{ Enumeration elements();
  boolean isEmpty();
  int size();
  Container newCont
}
```

```
public interface InspectablePositionalContainer extends Container
{ Enumeration positions();
  void swap(Position p, Position q);
  Object replace(Position p, Object e);
}
```

```
/** Et tre har minst en ekstern node – rot */
public interface InspectableTree
  extends PositionalContainer {
  Position root();
  Position parent(Position v);
  /** Barn er ordnet fra venstre til høyre */
  PositionIterator children(Position p);
  boolean isInternal(Position v);
  boolean isExternal(Position v);
  boolean isRoot(Position v);
  PositionIterator siblings(Position p);
}
```

```
public interface Container
{ Object replaceElement
  (Position p, Object o);
}
```

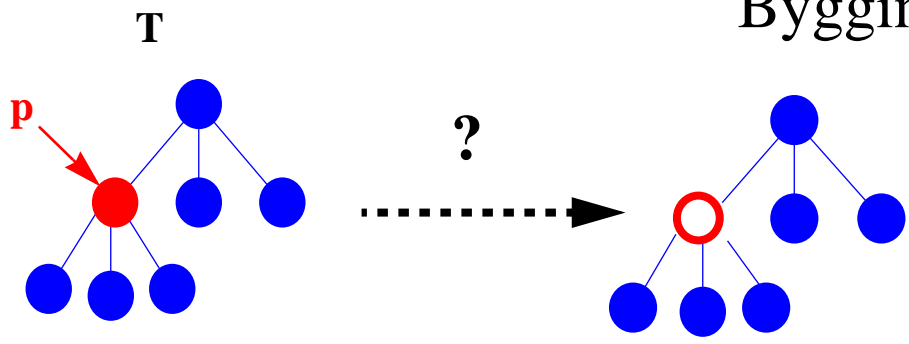
```
public interface InspectableBinaryTree
  extends InspectableTree {
  Position leftChild(Position);
  Position rightChild(Position);
  Position sibling(Position);
}
```

```
interface PositionalContainer
swapElements(Position p, q);
```

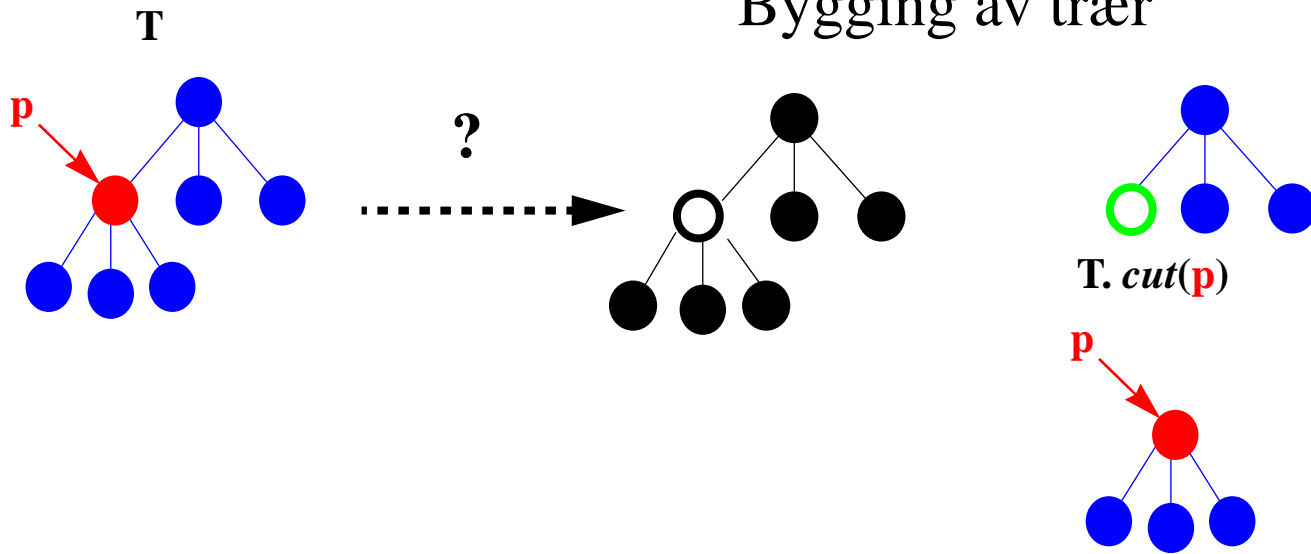
```
public interface BinaryTree
  extends InspectableBinaryTree {
  Tree cut(Position);
  void link(Position, BinaryTree);
  Tree replaceSubtree(Position, BinaryTree);
  void expandExternal(Position);
  void removeAboveExternal(Position);
}
```

```
public interface Tree extends InspectableTree {
  /** Node p erstattes med en ny ekstern node
   * med null element. */
  Tree cut(Position p);
  /** Node p (må være ekstern) erstattes med treet t */
  void link(Position p, Tree t);
  /** Erstatt subtree i node p med et nytt subtree t */
  Tree replaceSubtree(Position p, Tree t)
}
```

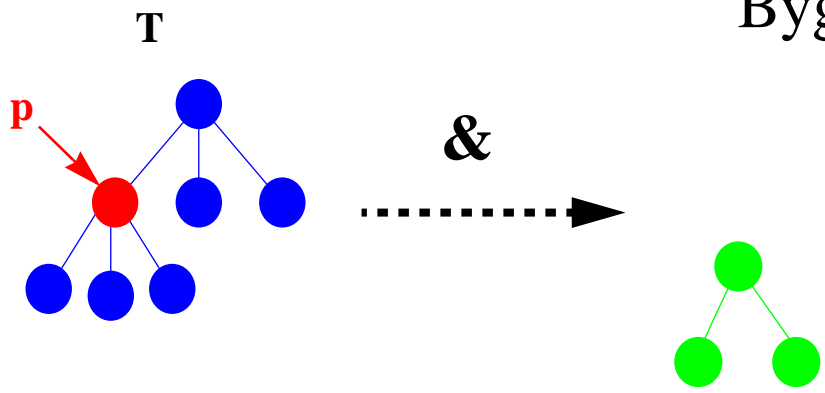
Bygging av trær



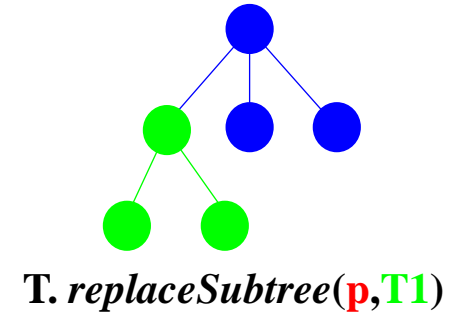
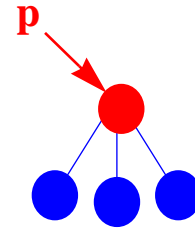
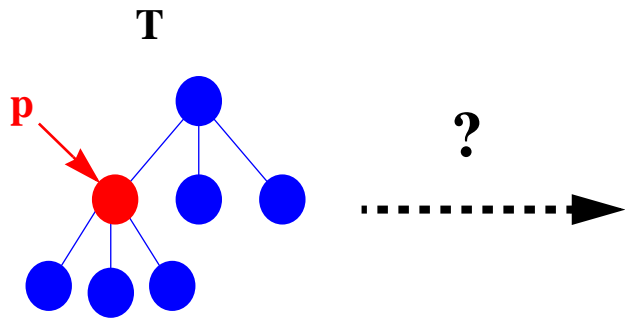
Bygging av trær



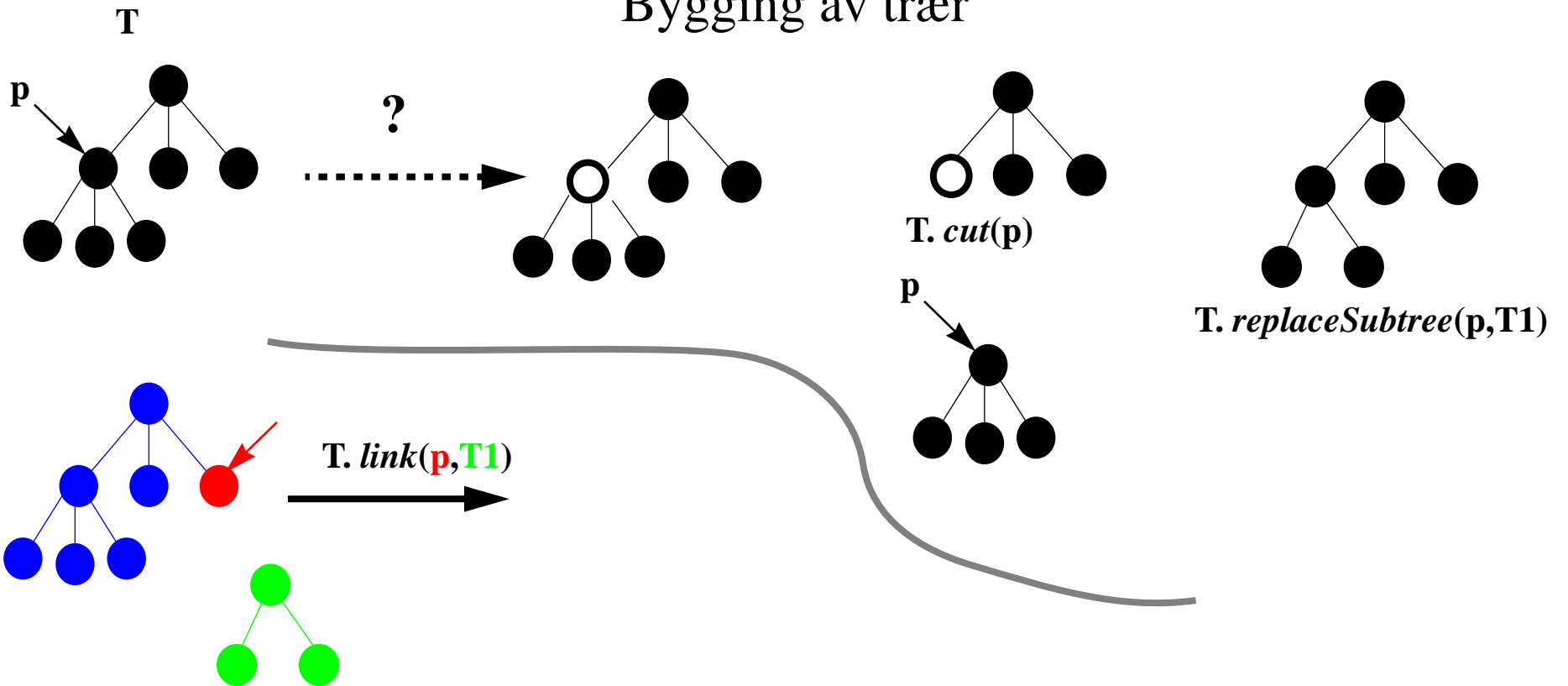
Bygging av trær



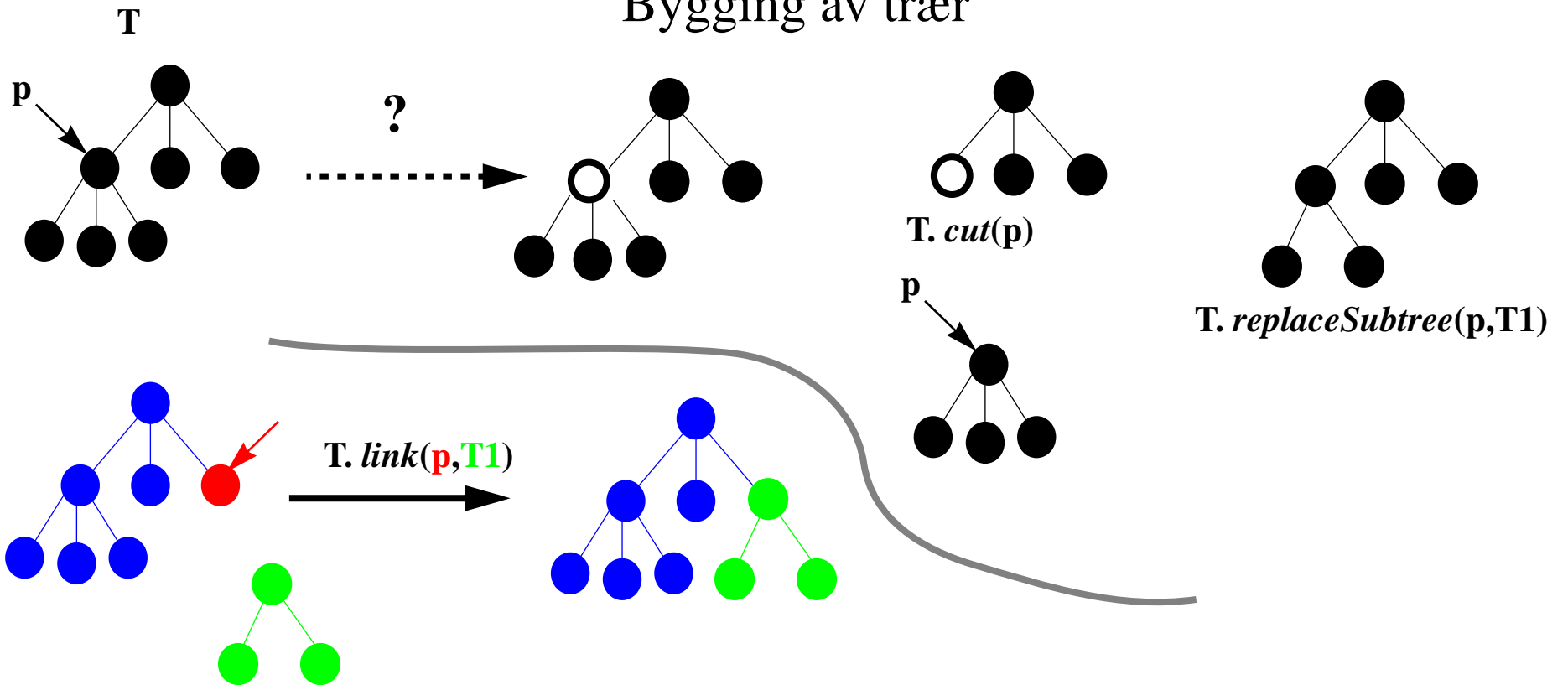
Bygging av trær



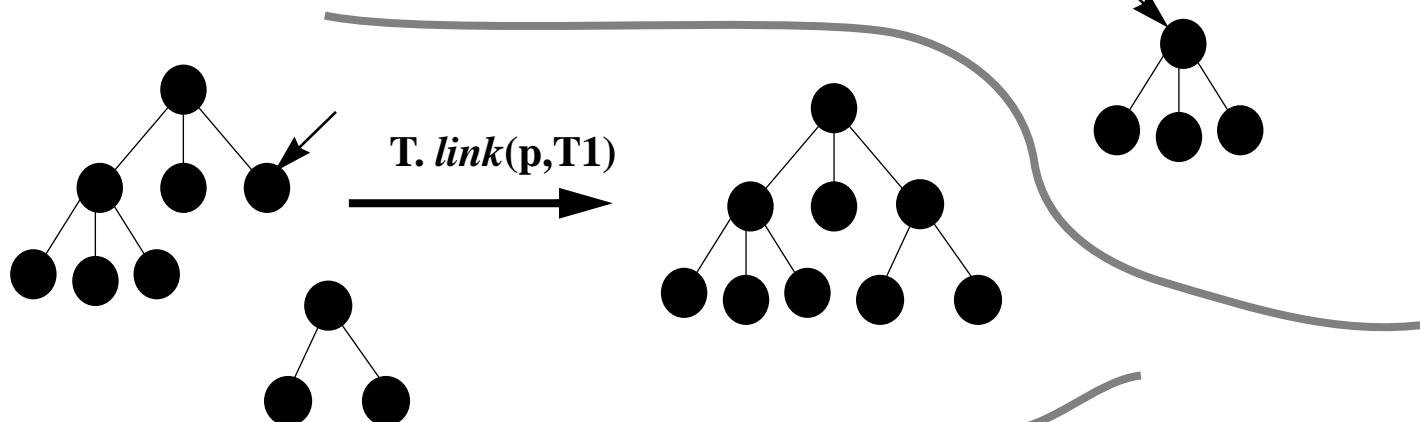
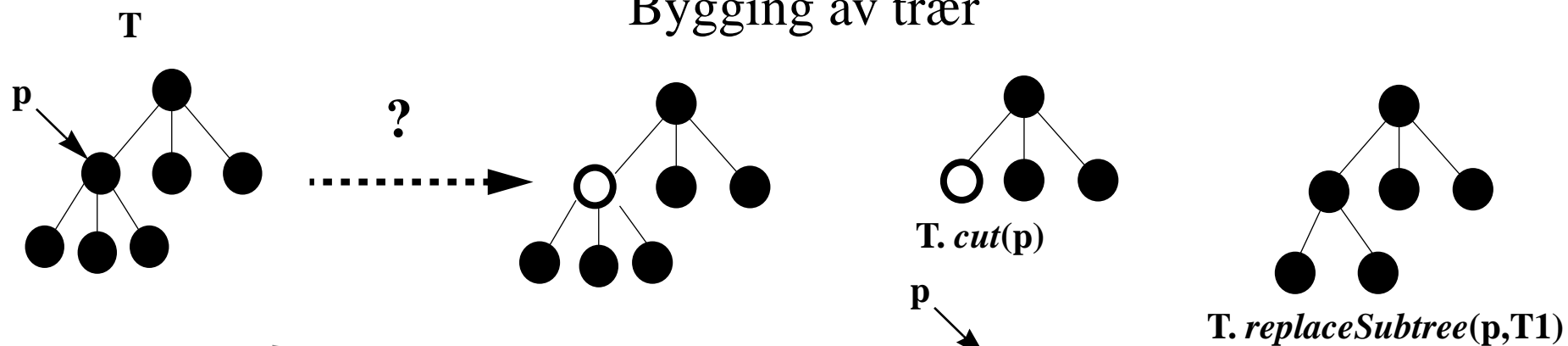
Bygging av trær



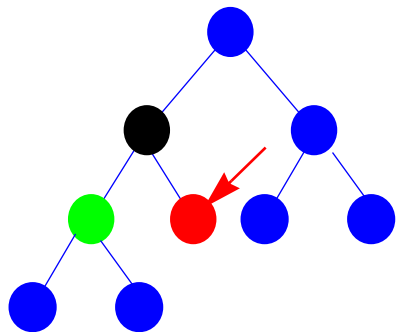
Bygging av trær



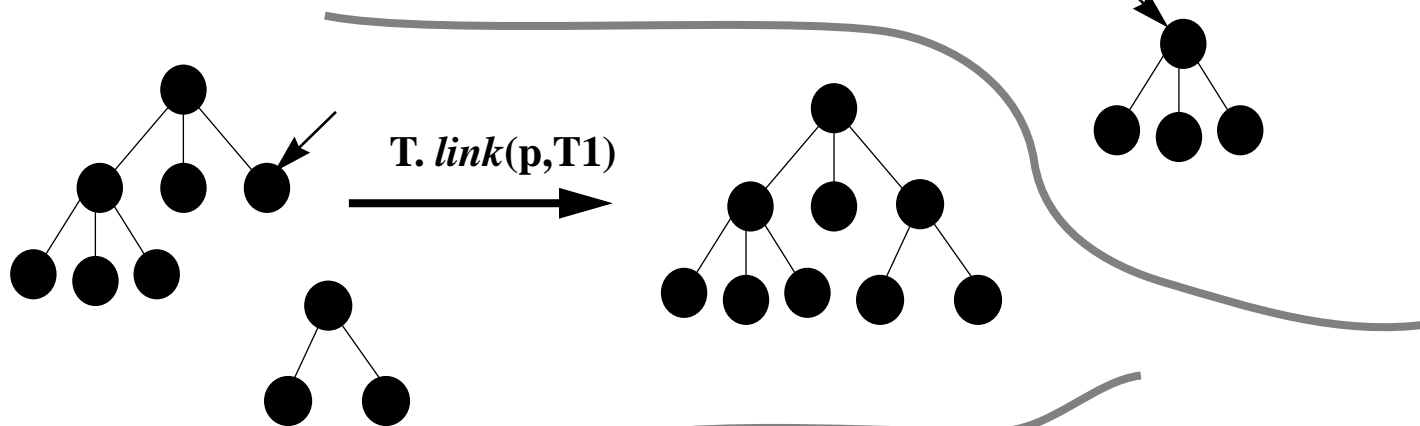
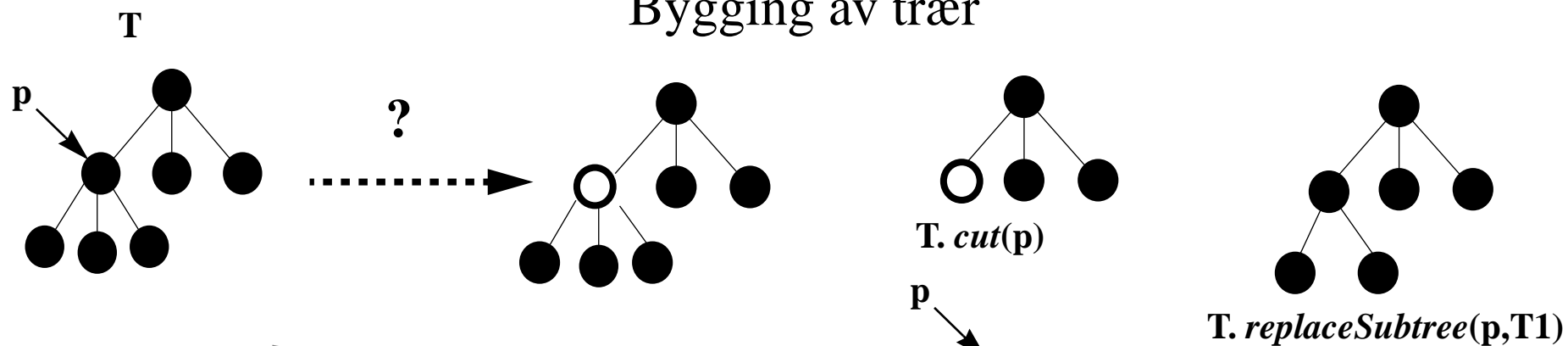
Bygging av trær



i et BinaryTree
 void *removeAboveExternal*(Position p)

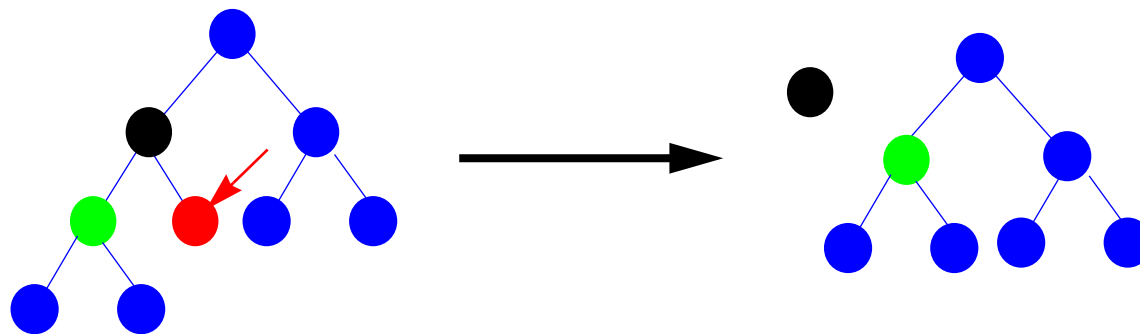


Bygging av trær



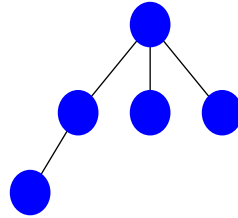
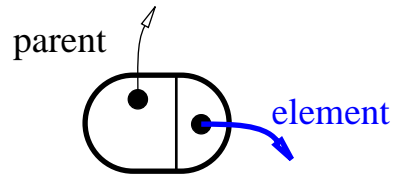
i et BinaryTree

`void removeAboveExternal(Position p)`



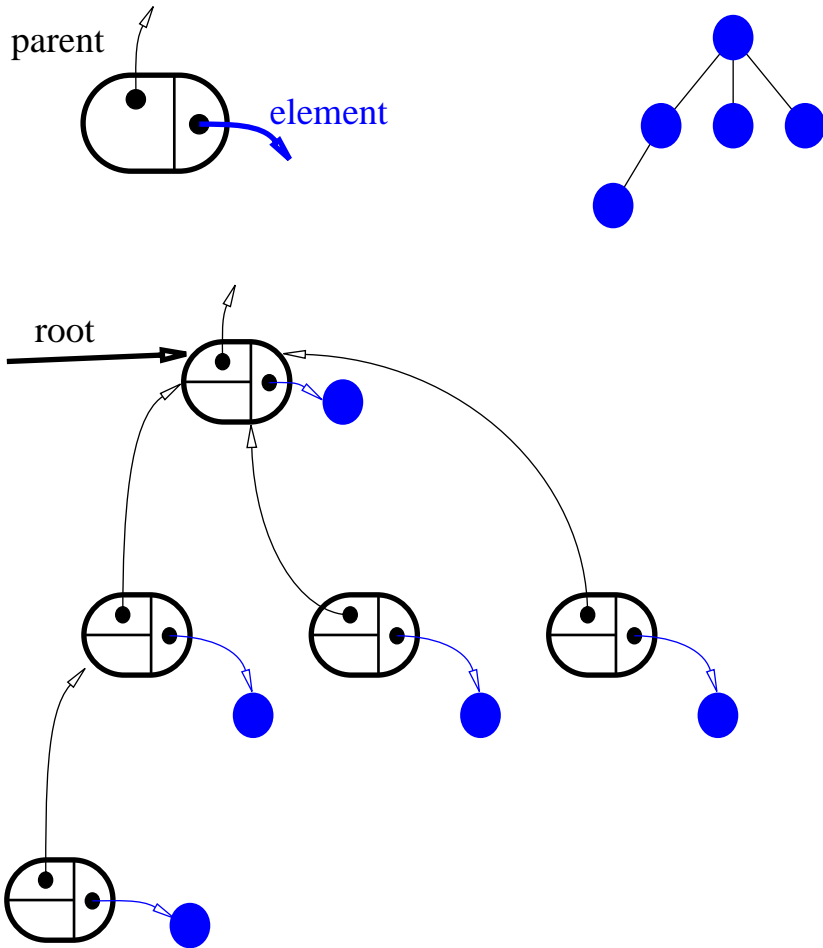
5. Implementasjon av Tree – *med LenketStruktur*

I. DATA REPRESENTASJON



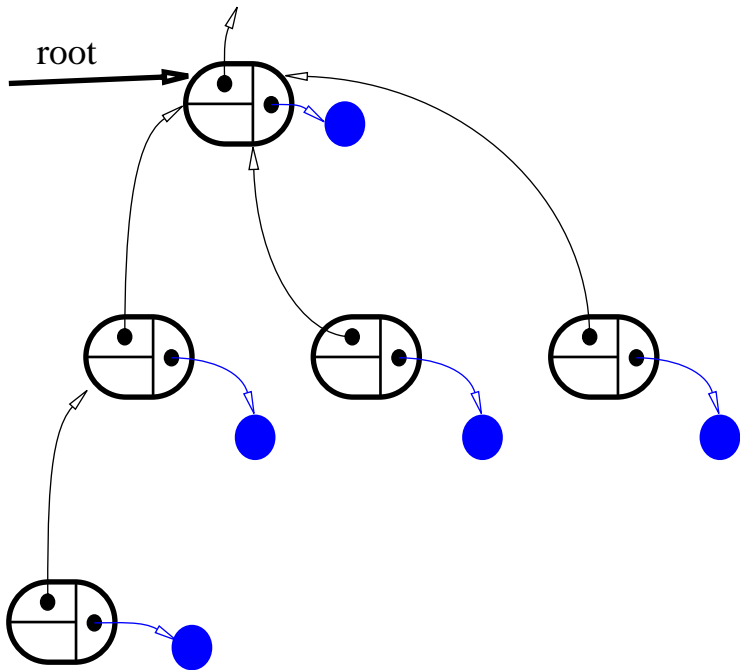
5. Implementasjon av Tree – med *LenketStruktur*

I. DATA REPRESENTASJON



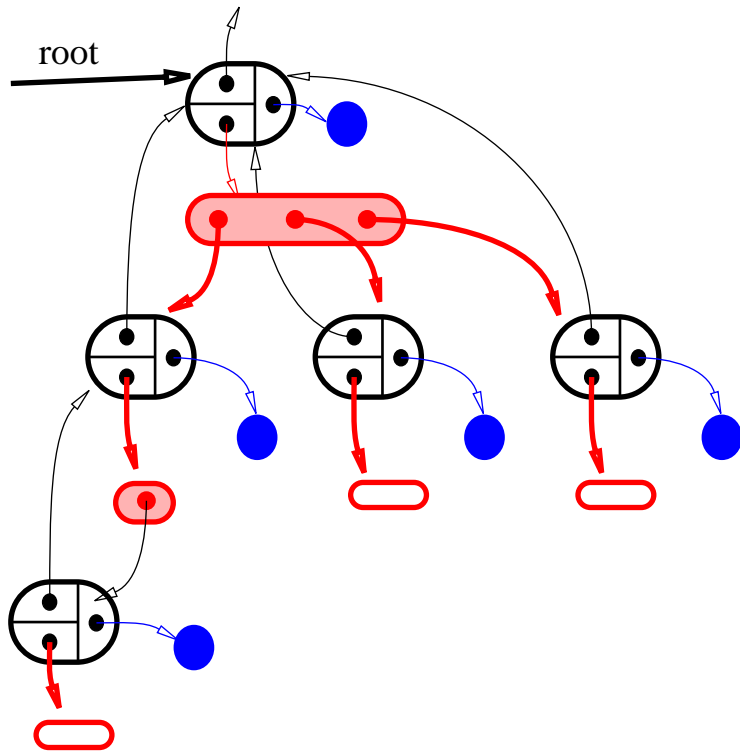
5. Implementasjon av Tree – med *LenketStruktur*

I. DATA REPRESENTASJON



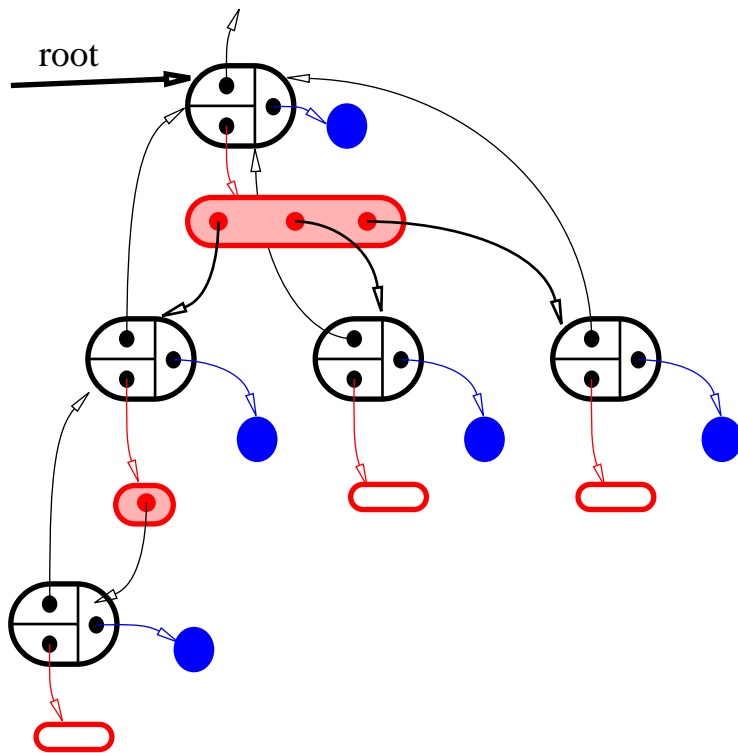
5. Implementasjon av Tree – med *LenketStruktur*

I. DATA REPRESENTASJON



5. Implementasjon av Tree – med *LenketStruktur*

I. DATA REPRESENTASJON



```
public class Node implements Position
{private Object elem;
private Node parent;
private Sequence children;

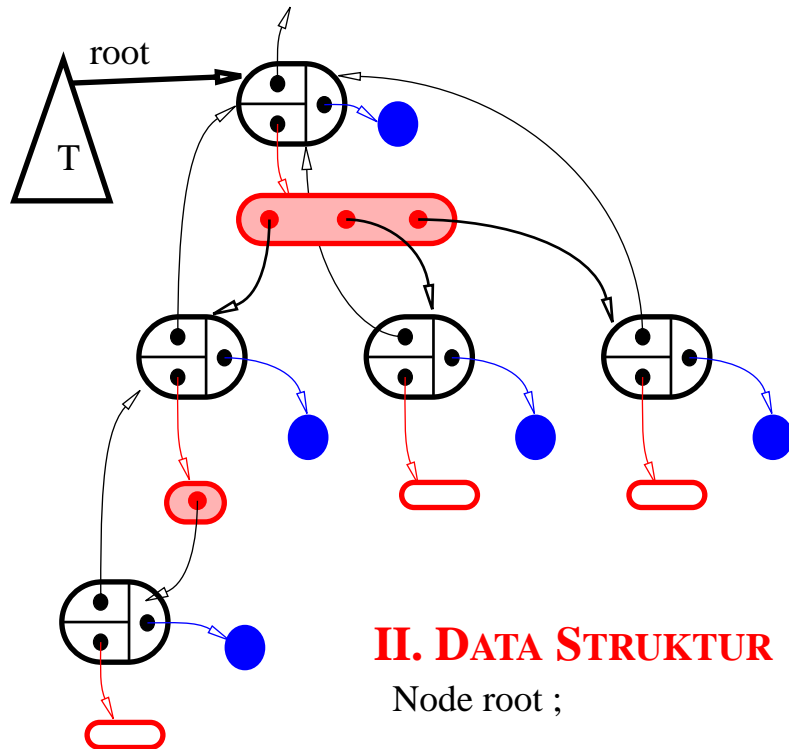
public Node(Object e, Node p) {
    setElement(e); setParent(p);
    children= new SequenceLL(); }

public Object element() { return elem; }
public void setElement(Object e) { elem= e; }

public Node parent() { return parent; }
public void setParent(Node p) { parent= p; }
public PositionIterator children() { return children.positions(); }
public void addChild(Node c) { ... }
public void removeChild(Node c) { ... }
```

5. Implementasjon av Tree – med *LenketStruktur*

I. DATA REPRESENTASJON



II. DATA STRUKTUR

Node root ;

```
public class Node implements Position
{private Object elem;
private Node parent;
private Sequence children;

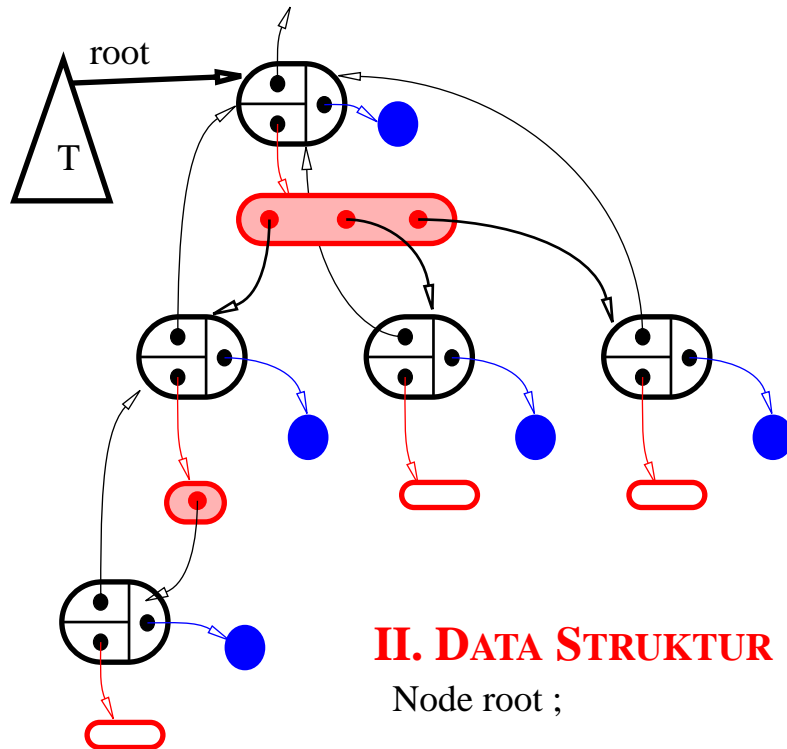
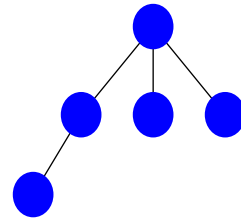
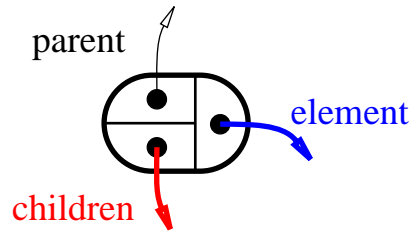
public Node(Object e, Node p) {
    setElement(e); setParent(p);
    children= new SequenceLL(); }

public Object element() { return elem; }
public void setElement(Object e) { elem= e; }

public Node parent() { return parent; }
public void setParent(Node p) { parent= p; }
public PositionIterator children() { return children.positions(); }
public void addChild(Node c) { ... }
public void removeChild(Node c) { ... }
```

5. Implementasjon av Tree – med LenketStruktur

I. DATA REPRESENTASJON



II. DATA STRUKTUR

Node root ;

III. DATA INVARIANT : for alle Position p i T:

- $T.root() \neq \text{null}$
- $\text{isRoot}(p) \leftrightarrow (p.parent() == \text{null})$
- $T.isEmpty() \leftrightarrow T.root().elem() == \text{null}$
- $\text{isExternal}(p) == p.children().isEmpty()$
- $\text{isInternal}(p) == !p.children().isEmpty()$
- $v.parent() == p \leftrightarrow v \text{ er bland } p.children()$
- $p.children() \neq \text{null}$

```
public class Node implements Position
```

```
{ private Object elem;
```

```
  private Node parent;
```

```
  private Sequence children;
```

```
public Node(Object e, Node p) {
```

```
    setElement(e); setParent(p);
```

```
    children= new SequenceLL(); }
```

```
public Object element() { return elem; }
```

```
public void setElement(Object e) { elem= e; }
```

```
public Node parent() { return parent; }
```

```
public void setParent(Node p) { parent= p; }
```

```
public PositionIterator children() { return children.positions(); }
```

```
public void addChild(Node c) { ... }
```

```
public void removeChild(Node c) { ... }
```

LenketStruktur implementasjon av Tree ADT

```
public class TreeLS implements Tree {  
    private Node root; private int size;
```

```
private Node ok(Position p) throws InvalidPosExc {  
    if ( p==null || !(p instanceof Node) )  
        throw new InvalidPosExc();  
    return (Node)p; }  
...
```

LenketStruktur implementasjon av Tree ADT

```
public class TreeLS implements Tree {
    private Node root; private int size;

    public TreeLS(Node n) { size= 1; root= n;
        n.setParent(null); }

    public TreeLS(Object e) {
        this(new Node(e, null)); }

    public Position root() { return root; }

    public Position parent(Position v) throws ... {
        return ok(v).parent(); }

    public PositionIterator children(Position v) throws.. {
        gjør om ok(v).children() til Iterator }
    //ikke ok(v).children().elements();}

    public boolean isInternal(Position v) throws ... {
        return !ok(v).children().isEmpty();}

    public boolean isExternal(Position v) throws ... {
        return ok(v).children().isEmpty();}

    public boolean isRoot(Position v) throws ... {
        return (v == root); }

    public int size() { return size; }
```

```
public Object replaceElement(Position v, Object e) throws..{
    Node p = ok(v);
    Object tmp = p . element();
    p.setElement(e); return tmp; }

private Node ok(Position p) throws InvalidPosExc {
    if ( p==null || !(p instanceof Node) )
        throw new InvalidPosExc();
    return (Node)p; }

    ...
```

LenketStruktur implementasjon av Tree ADT

```
public class TreeLS implements Tree {
    private Node root; private int size;

    public TreeLS(Node n) { size= 1; root= n;
        n.setParent(null); }

    public TreeLS(Object e) {
        this(new Node(e, null)); }

    public Position root() { return root; }

    public Position parent(Position v) throws ... {
        return ok(v).parent(); }

    public PositionIterator children(Position v) throws.. {
        gjør om ok(v).children() til Iterator }
    //ikke ok(v).children().elements();}

    public boolean isInternal(Position v) throws ... {
        return !ok(v).children().isEmpty();}

    public boolean isExternal(Position v) throws ... {
        return ok(v).children().isEmpty();}

    public boolean isRoot(Position v) throws ... {
        return (v == root); }

    public int size() { return size; }
```

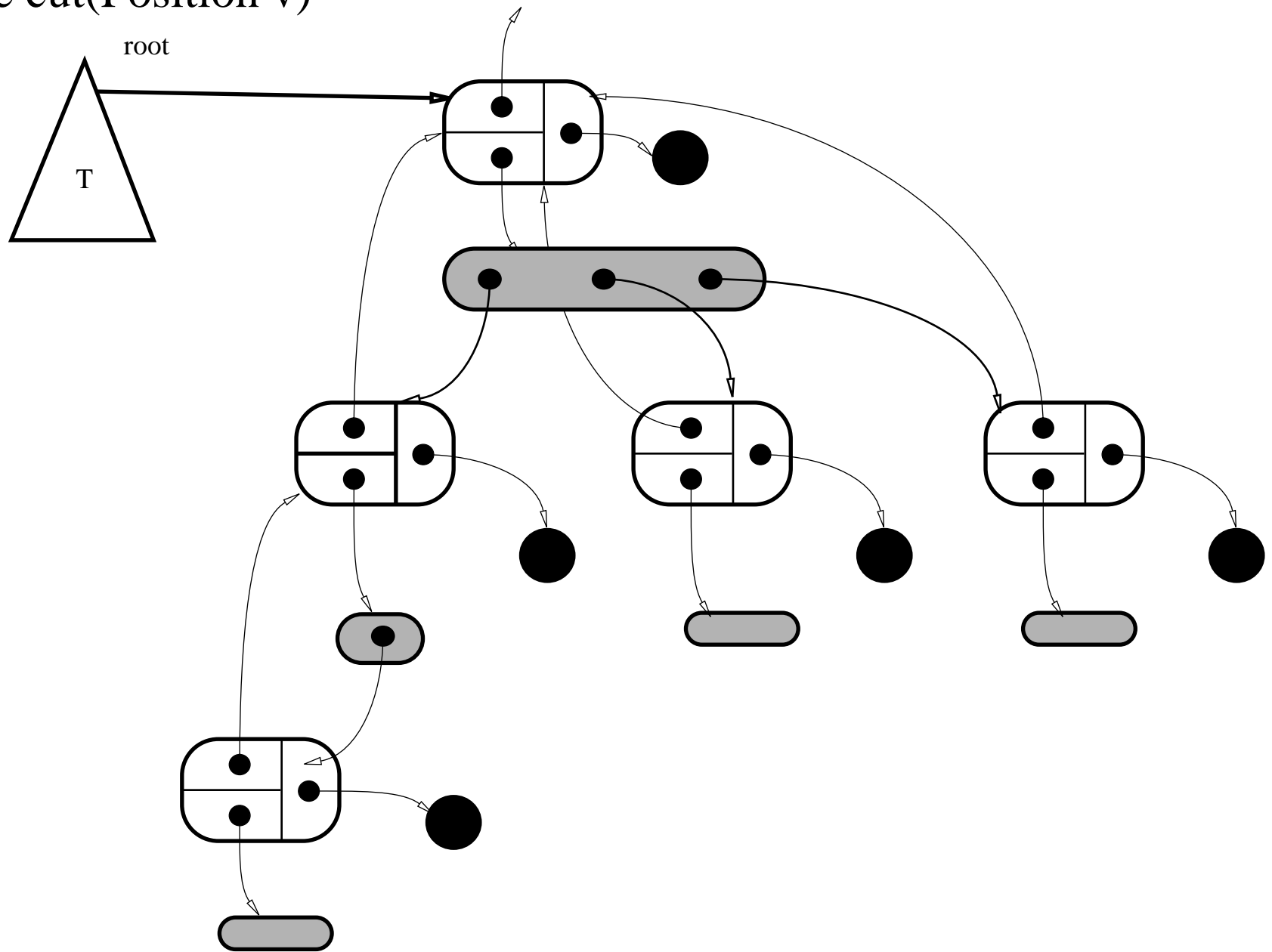
```
public Object replaceElement(Position v, Object e) throws..{
    Node p = ok(v);
    Object tmp = p . element();
    p.setElement(e); return tmp; }

    public Tree cut(Position v) throws ... {
        Node n = ok(v);
        Node p = ok(parent(v));
        int less = n . size();
        TreeLS ret = new TreeLS(n);
        p.removeChild(n);
        size = size – less ;
        return ret; } // v er ikke lenger Position i dette treet

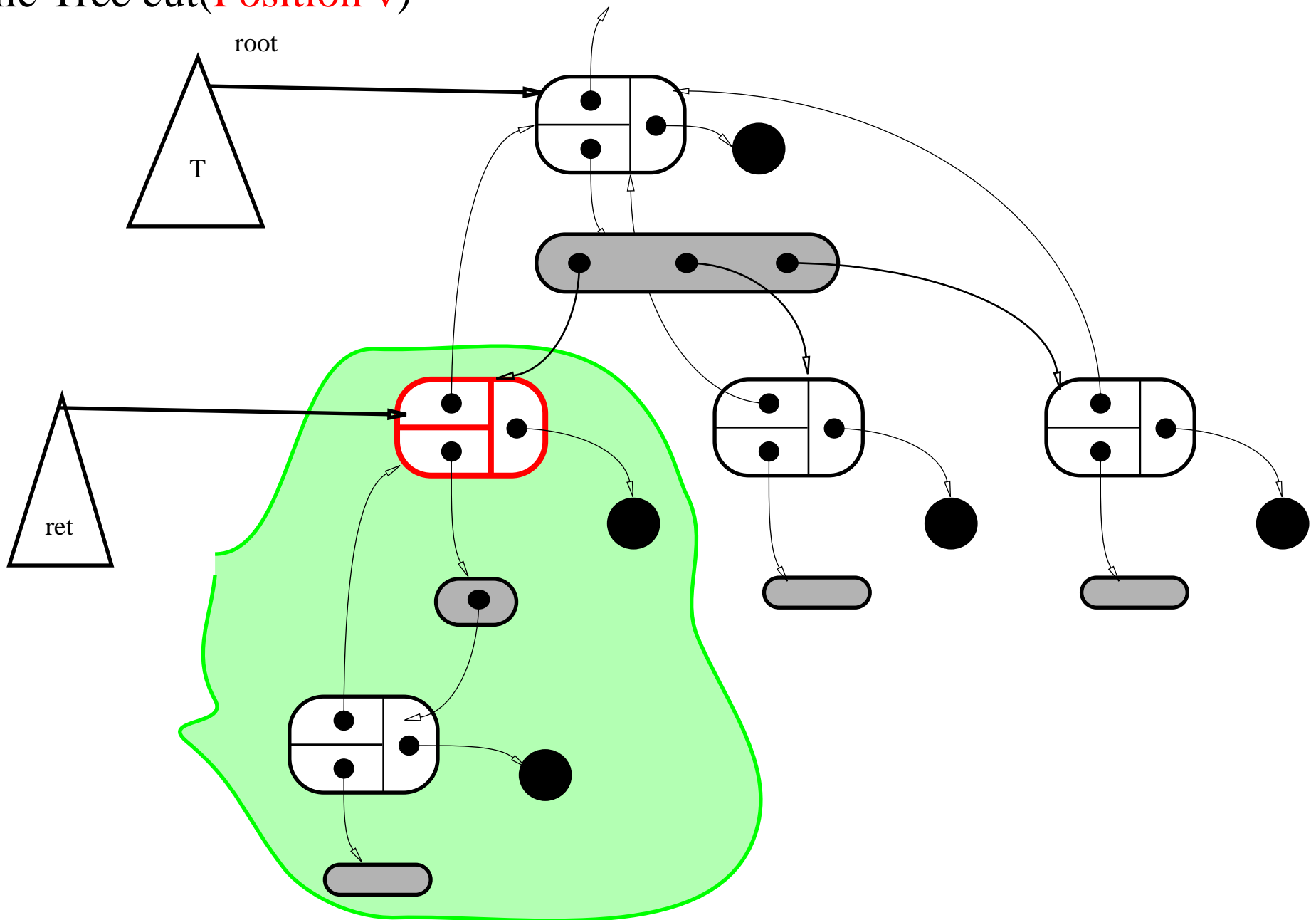
    private Node ok(Position p) throws InvalidPosExc {
        if ( p==null || !(p instanceof Node) )
            throw new InvalidPosExc();
        return (Node)p; }

    ...
```

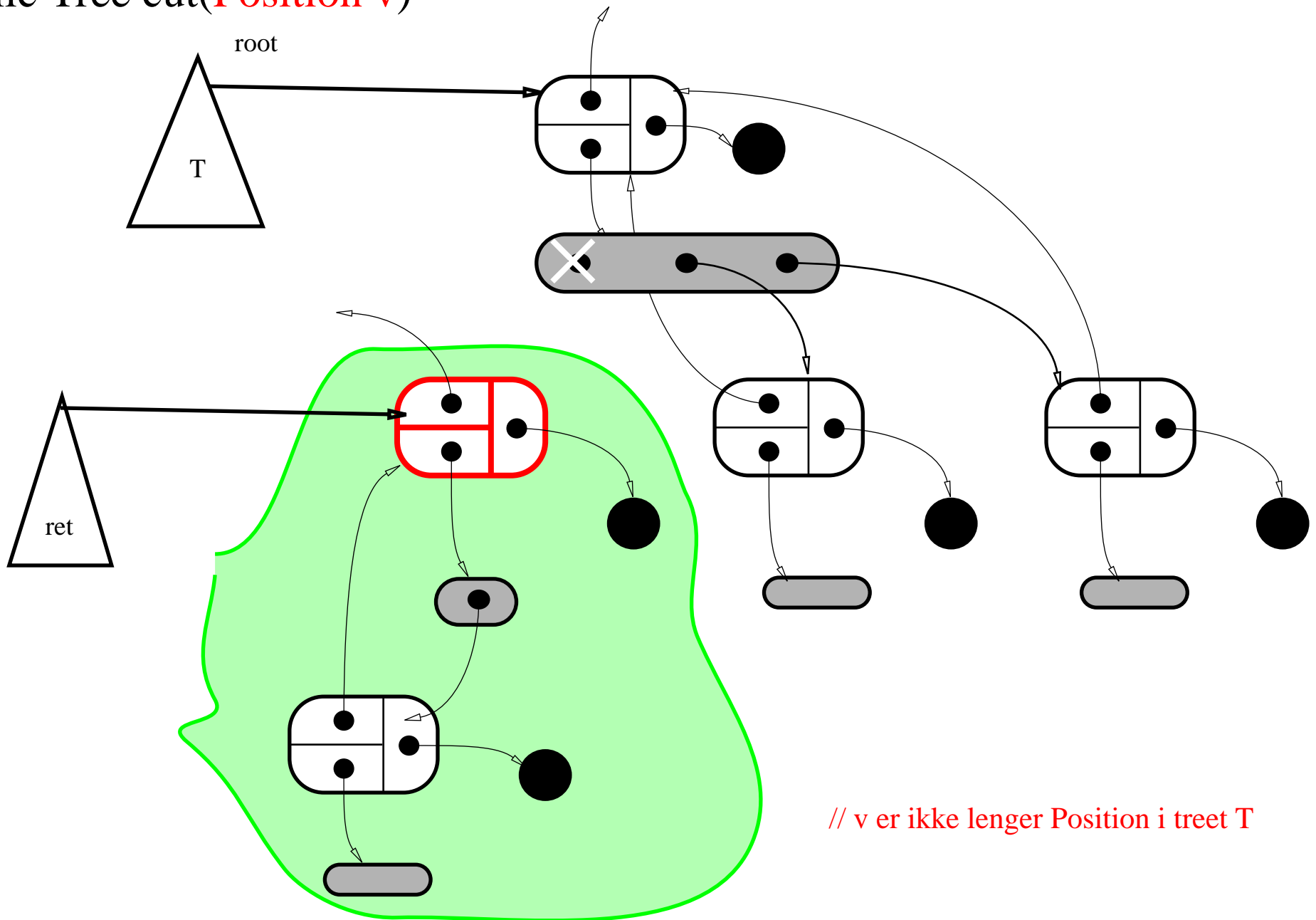
public Tree cut(Position v)



public Tree cut(**Position v**)



public Tree cut(Position v)



LenketStruktur implementasjon av Tree ADT

```
public class TreeLS implements Tree {
    private Node root; private int size;

    public TreeLS(Node n) { size= 1; root= n;
        n.setParent(null); }

    public TreeLS(Object e) {
        this(new Node(e, null)); }

    public Position root() { return root; }

    public Position parent(Position v) throws ... {
        return ok(v).parent(); }

    public PositionIterator children(Position v) throws.. {
        gjør om ok(v).children() til Iterator }
    //ikke ok(v).children().elements();}

    public boolean isInternal(Position v) throws ... {
        return !ok(v).children().isEmpty();}

    public boolean isExternal(Position v) throws ... {
        return ok(v).children().isEmpty();}

    public boolean isRoot(Position v) throws ... {
        return (v == root); }

    public int size() { return size; }
```

```
public Object replaceElement(Position v, Object e) throws..{
    Node p = ok(v);
    Object tmp = p . element();
    p.setElement(e); return tmp; }

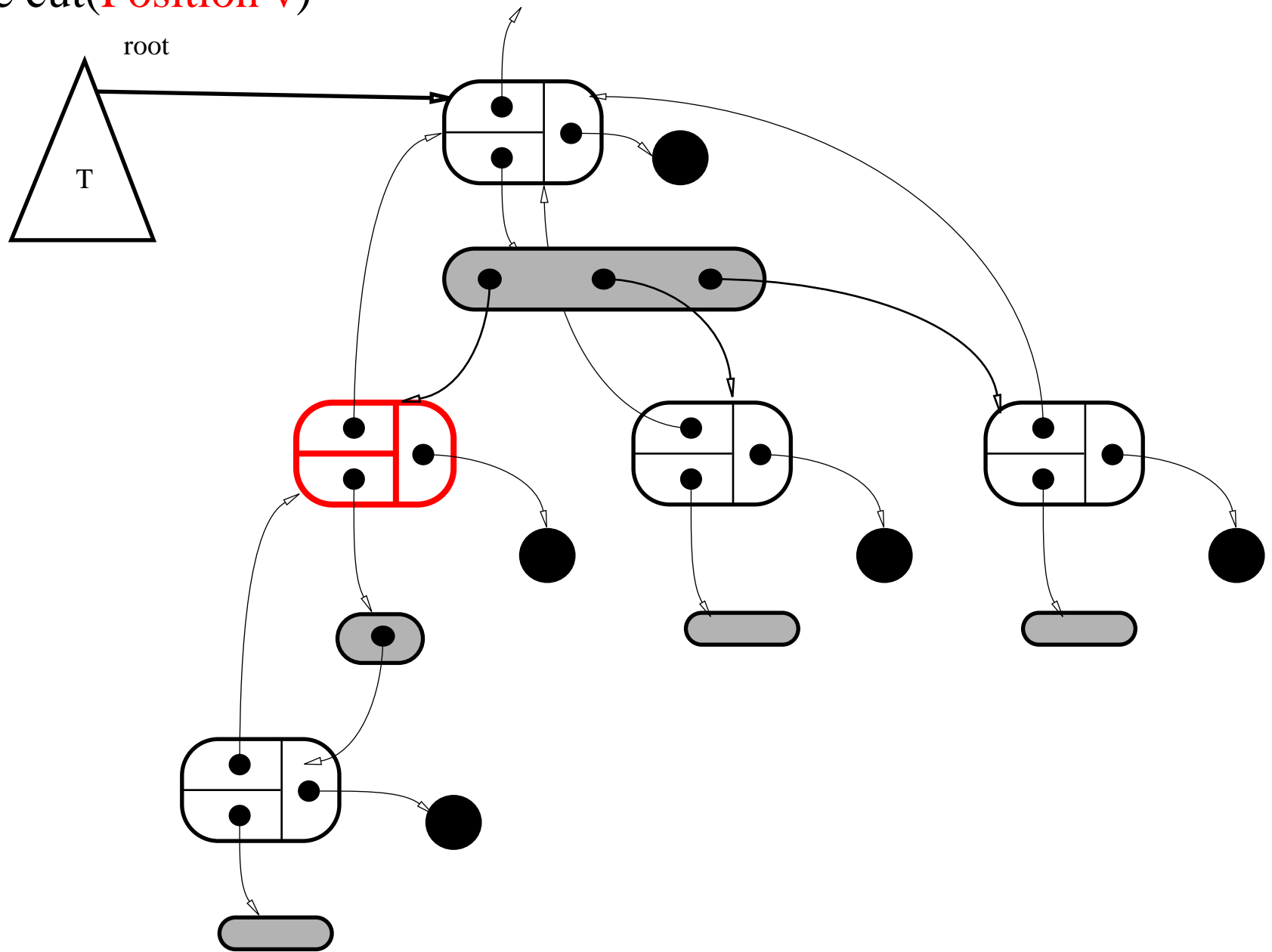
public Tree cut(Position v) throws ... {
    Node n= ok(v);
    TreeLS ret= new TreeLS(n.element());
    Node retRoot = (Node)ret.root();
    PositionIterator ch= children(n);
    while (ch.hasMoreElements()) {
        ((Node)ch.nextElement()).setParent(retRoot); }
    retRoot.setChildren(n.children());
    n.setElement(null); n.setChildren(null);
    size = size – ret.size() + 1;
    return ret; } // v er fortsatt Position i dette treet

private Node ok(Position p) throws InvalidPosExc {
    if ( p==null || !(p instanceof Node) )
        throw new InvalidPosExc();
    return (Node)p; }

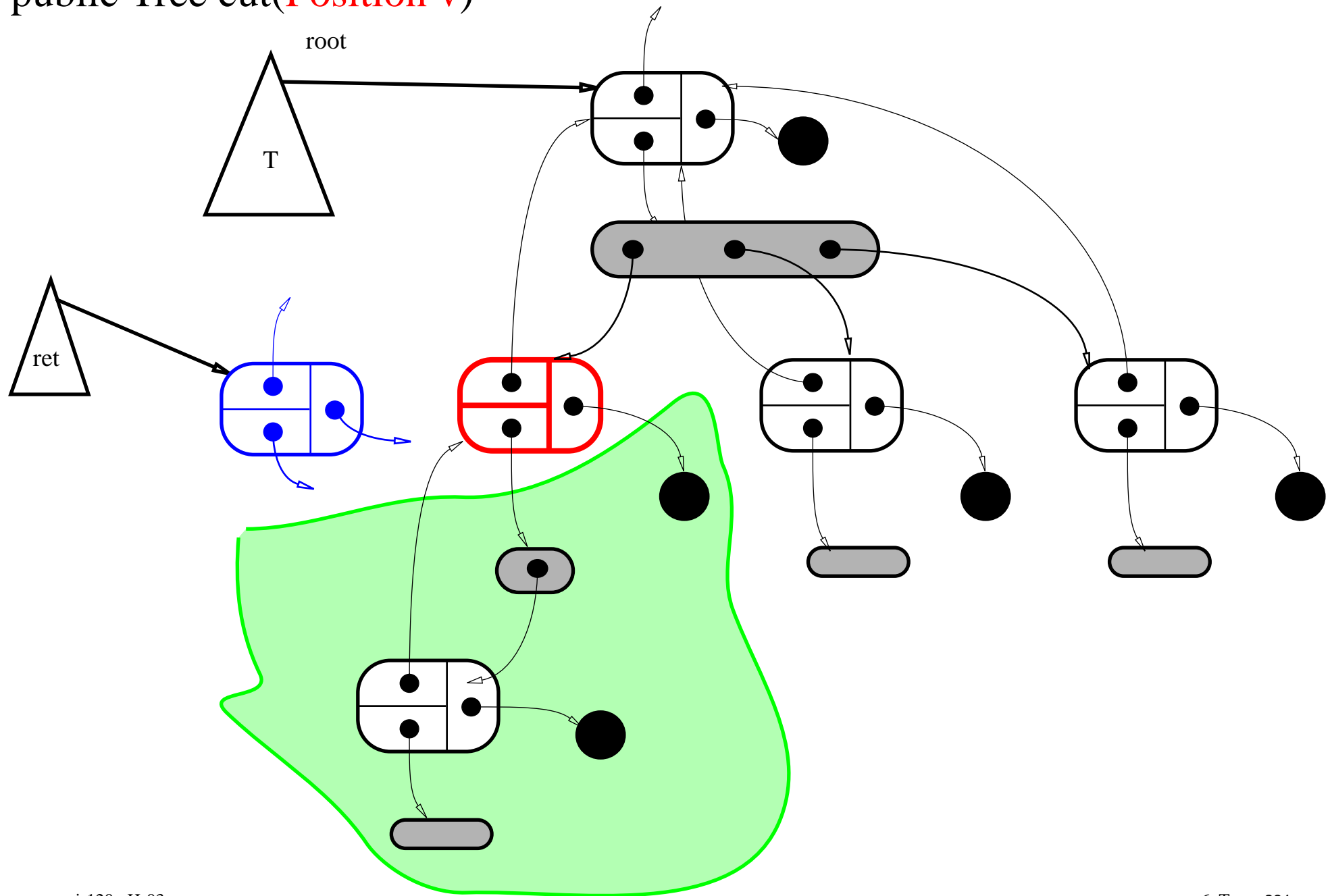
...

```

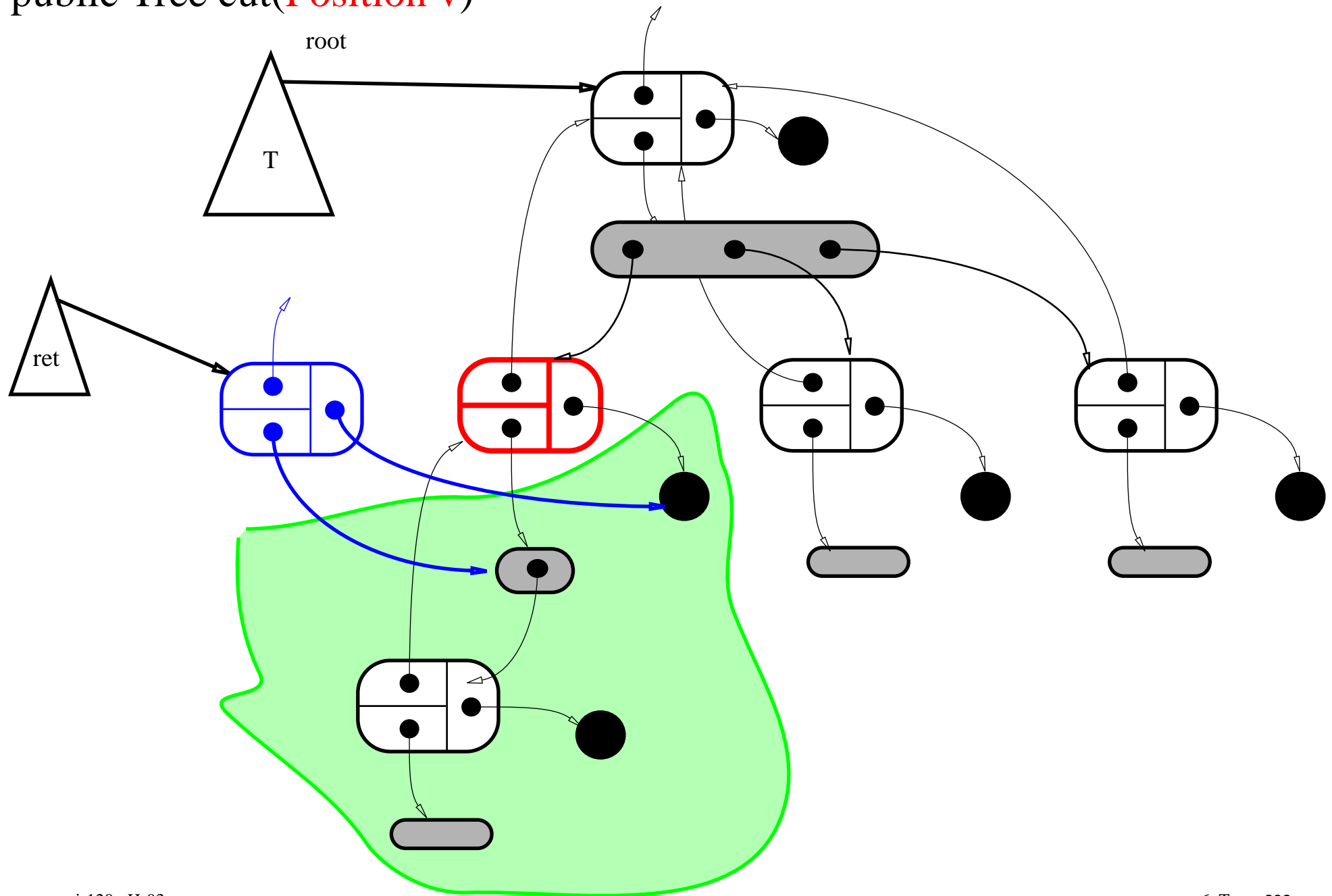
public Tree cut(**Position v**)



public Tree cut(**Position v**)



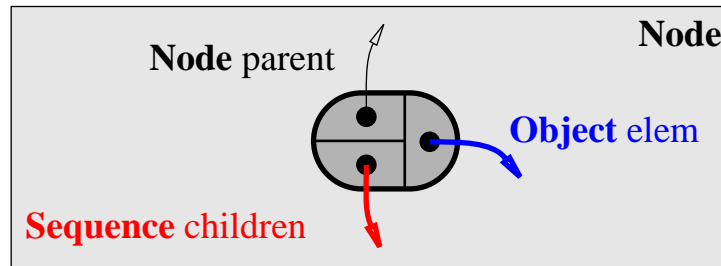
public Tree cut(**Position v**)



Implementasjon av BinaryTree ADT

I. UTVID KLASSEN TREEELS SLIK AT:

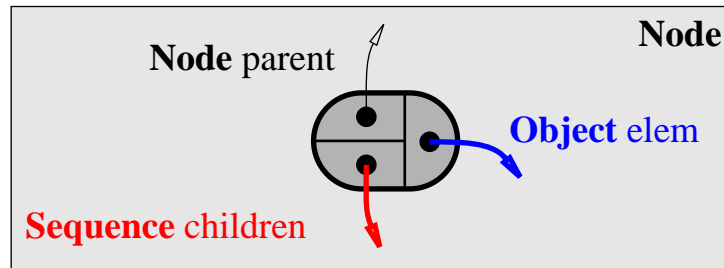
- Container `children()` alltid har 0 eller 2 elementer – pass på `link(p,T)`, `replaceSubtree(p,T)`
- implementer Position `leftChild()`, `rightChild()` – skill i `children()`



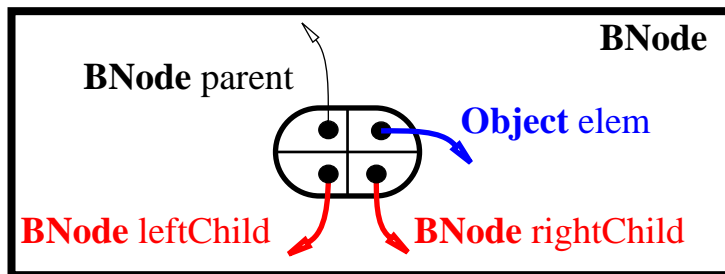
Implementasjon av BinaryTree ADT

I. UTVID KLASSEN TREEELS SLIK AT:

- Container `children()` alltid har 0 eller 2 elementer – pass på `link(p,T)`, `replaceSubtree(p,T)`
- implementer `Position leftChild()`, `Position rightChild()` – skill i `children()`



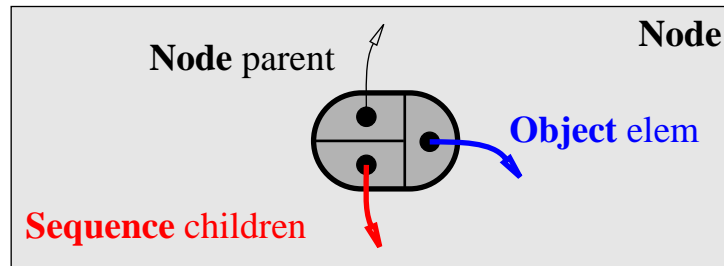
II. BRUK EN ANNEN BNODE



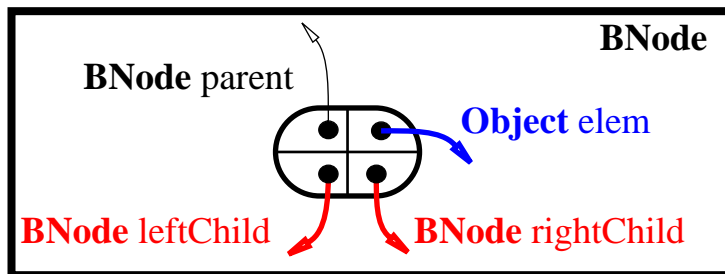
Implementasjon av BinaryTree ADT

I. UTVID KLASSEN TREEELS SLIK AT:

- Container `children()` alltid har 0 eller 2 elementer – pass på `link(p,T)`, `replaceSubtree(p,T)`
- implementer `Position leftChild()`, `Position rightChild()` – skill i `children()`



II. BRUK EN ANNEN BNODE



```
public class BNode implements Position
{ private Object elem;
  private BNode parent, left, right;
  public BNode(Object e, BNode p) {
    setElement(e); setParent(p); }
  public Object element() { return elem; }
  ...
  public BNode leftChild() { return left; }
  protected void setLeft(BNode p) {
    left = p; p.setParent(this); }
  public BNode rightChild() { return right; }
  protected void setRight(BNode p) {
    right = p; p.setParent(this); ; }
```

LenketStruktur implementasjon av BinaryTree ADT

```
public class BTreeLS implements BinTree {
    private Position root;
    public BTreeLS(Object e) {
        root = new BNode(e, null); }
    public Position parent(Position v) throws ... {
        return ok(v).parent(); }
    public Position leftChild(Position v) throws ... {
        return ok(v).leftChild(); }
    public Position setLeft(Position v, Object o) throws...{
        BNode n= ok(v);
        if (isExternal(n)) expandExternal(n);
        n.setLeft( new BNode(o, n) );
        return n.leftChild(); }
    public Position rightChild(Position v) throws..{ ... }
    public Position setRight(Position v, Object c) throws..{.}
    private BNode ok(Position p) throws InvalidPosExc {
        if ( p==null || !(p instanceof BNode) )
            throw new InvalidPosExc();
        return (BNode)p; }
    public int size() { return size(root); }
    private int size(Position p) { if (isExternal(p)) return 1;
        else return size(leftChild(p)) + size(rightChild(p)) +1; }
```

```
public boolean isInternal(Position v) throws ... {
    BNode n= ok(v);
    return n.leftChild()!=null || n.rightChild()!=null; }
public boolean isExternal(Position v) throws ... {
    return ! isInternal(v); }
```

LenketStruktur implementasjon av BinaryTree ADT

```
public class BTreeLS implements BinTree {
    private Position root;
    public BTreeLS(Object e) {
        root = new BNode(e, null); }
    public Position parent(Position v) throws ... {
        return ok(v).parent(); }
    public Position leftChild(Position v) throws ... {
        return ok(v).leftChild(); }
    public Position setLeft(Position v, Object o) throws...{
        BNode n= ok(v);
        if (isExternal(n)) expandExternal(n);
        n.setLeft( new BNode(o, n) );
        return n.leftChild(); }
    public Position rightChild(Position v) throws..{ ... }
    public Position setRight(Position v, Object c) throws..{.}
    private BNode ok(Position p) throws InvalidPosExc {
        if ( p==null || !(p instanceof BNode) )
            throw new InvalidPosExc();
        return (BNode)p; }
    public int size() { return size(root); }
    private int size(Position p) { if (isExternal(p)) return 1;
        else return size(leftChild(p)) + size(rightChild(p)) +1; }
```

```
public boolean isInternal(Position v) throws ... {
    BNode n= ok(v);
    return n.leftChild()!=null || n.rightChild()!=null; }
public boolean isExternal(Position v) throws ... {
    return ! isInternal(v); }
public void expandExternal(Position v) {
    BNode n= ok(v);
    if (isExternal(n)) {
        n.setLeft( new BNode(null, n) );
        n.setRight( newBNode(null, n) ); } }
```

LenketStruktur implementasjon av BinaryTree ADT

```

public class BTreeLS implements BinTree {
  private Position root;
  public BTreeLS(Object e) {
    root = new BNode(e, null); }
  public Position parent(Position v) throws ... {
    return ok(v).parent(); }
  public Position leftChild(Position v) throws ... {
    return ok(v).leftChild(); }
  public Position setLeft(Position v, Object o) throws...{
    BNode n= ok(v);
    if (isExternal(n)) expandExternal(n);
    n.setLeft( new BNode(o, n) );
    return n.leftChild(); }
  public Position rightChild(Position v) throws..{ ... }
  public Position setRight(Position v, Object c) throws..{.}
  private BNode ok(Position p) throws InvalidPosExc {
    if ( p==null || !(p instanceof BNode) )
      throw new InvalidPosExc();
    return (BNode)p; }
  public int size() { return size(root); }
  private int size(Position p) { if (isExternal(p)) return 1;
    else return size(leftChild(p)) + size(rightChild(p)) + 1; }

```

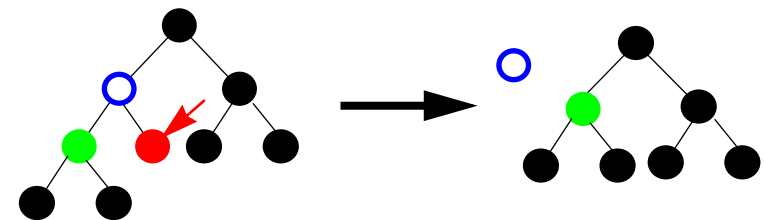
```

public boolean isInternal(Position v) throws ... {
  BNode n= ok(v);
  return n.leftChild()!=null || n.rightChild()!=null); }
public boolean isExternal(Position v) throws ... {
  return ! isInternal(v); }

public void expandExternal(Position v) {
  BNode n= ok(v);
  if (isExternal(n)) {
    n.setLeft( new BNode(null, n) );
    n.setRight( newBNode(null, n) ); } }

public void removeAboveExternal(Position v) throws...{
  BNode n = ok(v); // Object o = v.element();
  if ( isInternal(n) || isRoot(n) ) throw ...
  BNode par = ok(parent(n)) ;
  BNode sib = ok(sibling(n));
  par.setElement(sib.element());
  par.setLeft(sib.leftChild());
  par.setRight(sib.rightChild());
}

```



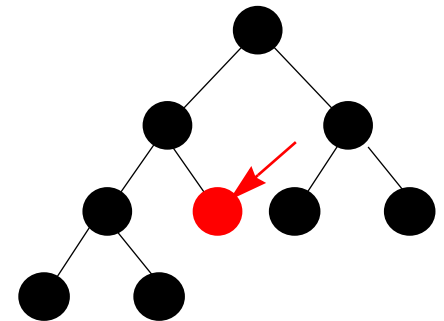
LenketStruktur implementasjon av BinaryTree ADT

```
public class BTreeLS implements BinTree {
    private Position root;
    public BTreeLS(Object e) {
        root = new BNode(e, null); }
    public Position parent(Position v) throws ... {
        return ok(v).parent(); }
    public Position leftChild(Position v) throws ... {
        return ok(v).leftChild(); }
    public Position setLeft(Position v, Object o) throws...{
        BNode n= ok(v);
        if (isExternal(n)) expandExternal(n);
        n.setLeft( new BNode(o, n) );
        return n.leftChild(); }
    public Position rightChild(Position v) throws..{ ... }
    public Position setRight(Position v, Object c) throws..{.}
    private BNode ok(Position p) throws InvalidPosExc {
        if ( p==null || !(p instanceof BNode) )
            throw new InvalidPosExc();
        return (BNode)p; }
    public int size() { return size(root); }
    private int size(Position p) { if (isExternal(p)) return 1;
        else return size(leftChild(p)) + size(rightChild(p)) +1; }
```

```
public boolean isInternal(Position v) throws ... {
    BNode n= ok(v);
    return n.leftChild()!=null || n.rightChild()!=null); }
public boolean isExternal(Position v) throws ... {
    return ! isInternal(v); }

public void expandExternal(Position v) {
    BNode n= ok(v);
    if (isExternal(n)) {
        n.setLeft( new BNode(null, n) );
        n.setRight( newBNode(null, n) ); } }

public void removeAboveExternal(Position v) throws...{
    BNode n = ok(v); // Object o = v.element();
```



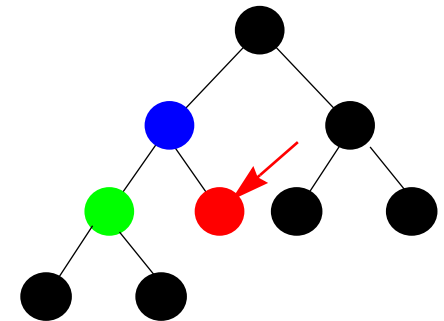
LenketStruktur implementasjon av BinaryTree ADT

```
public class BTreeLS implements BinTree {
  private Position root;
  public BTreeLS(Object e) {
    root = new BNode(e, null); }
  public Position parent(Position v) throws ... {
    return ok(v).parent(); }
  public Position leftChild(Position v) throws ... {
    return ok(v).leftChild(); }
  public Position setLeft(Position v, Object o) throws...{
    BNode n= ok(v);
    if (isExternal(n)) expandExternal(n);
    n.setLeft( new BNode(o, n) );
    return n.leftChild(); }
  public Position rightChild(Position v) throws..{ ... }
  public Position setRight(Position v, Object c) throws..{.}
  private BNode ok(Position p) throws InvalidPosExc {
    if ( p==null || !(p instanceof BNode) )
      throw new InvalidPosExc();
    return (BNode)p; }
  public int size() { return size(root); }
  private int size(Position p) { if (isExternal(p)) return 1;
    else return size(leftChild(p)) + size(rightChild(p)) +1; }
```

```
public boolean isInternal(Position v) throws ... {
  BNode n= ok(v);
  return n.leftChild()!=null || n.rightChild()!=null); }
public boolean isExternal(Position v) throws ... {
  return ! isInternal(v); }

public void expandExternal(Position v) {
  BNode n= ok(v);
  if (isExternal(n)) {
    n.setLeft( new BNode(null, n) );
    n.setRight( newBNode(null, n) ); } }

public void removeAboveExternal(Position v) throws...{
  BNode n = ok(v); // Object o = v.element();
  if ( isInternal(n) || isRoot(n) ) throw ...
  BNode par = ok(parent(n));
  BNode sib = ok(sibling(n));
```



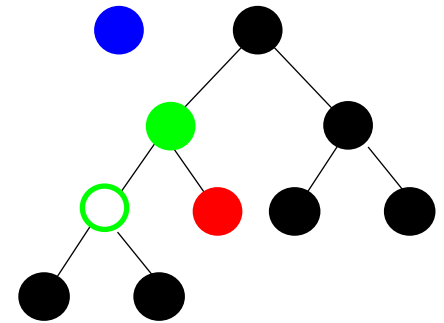
LenketStruktur implementasjon av BinaryTree ADT

```
public class BTreeLS implements BinTree {
    private Position root;
    public BTreeLS(Object e) {
        root = new BNode(e, null); }
    public Position parent(Position v) throws ... {
        return ok(v).parent(); }
    public Position leftChild(Position v) throws ... {
        return ok(v).leftChild(); }
    public Position setLeft(Position v, Object o) throws...{
        BNode n= ok(v);
        if (isExternal(n)) expandExternal(n);
        n.setLeft( new BNode(o, n) );
        return n.leftChild(); }
    public Position rightChild(Position v) throws..{ ... }
    public Position setRight(Position v, Object c) throws..{.}
    private BNode ok(Position p) throws InvalidPosExc {
        if ( p==null || !(p instanceof BNode) )
            throw new InvalidPosExc();
        return (BNode)p; }
    public int size() { return size(root); }
    private int size(Position p) { if (isExternal(p)) return 1;
        else return size(leftChild(p)) + size(rightChild(p)) +1; }
```

```
public boolean isInternal(Position v) throws ... {
    BNode n= ok(v);
    return n.leftChild()!=null || n.rightChild()!=null); }
public boolean isExternal(Position v) throws ... {
    return ! isInternal(v); }

public void expandExternal(Position v) {
    BNode n= ok(v);
    if (isExternal(n)) {
        n.setLeft( new BNode(null, n) );
        n.setRight( newBNode(null, n) ); } }

public void removeAboveExternal(Position v) throws...{
    BNode n = ok(v); // Object o = v.element();
    if ( isInternal(n) || isRoot(n) ) throw ...
    BNode par = ok(parent(n)) ;
    BNode sib = ok(sibling(n));
    par.setElement(sib.element());
```



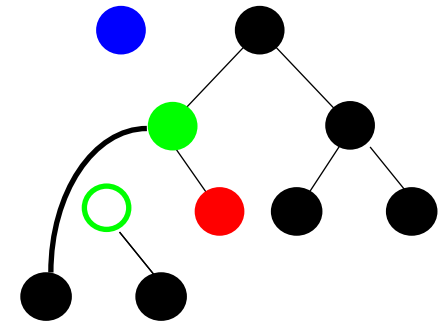
LenketStruktur implementasjon av BinaryTree ADT

```
public class BTreeLS implements BinTree {
    private Position root;
    public BTreeLS(Object e) {
        root = new BNode(e, null); }
    public Position parent(Position v) throws ... {
        return ok(v).parent(); }
    public Position leftChild(Position v) throws ... {
        return ok(v).leftChild(); }
    public Position setLeft(Position v, Object o) throws...{
        BNode n= ok(v);
        if (isExternal(n)) expandExternal(n);
        n.setLeft( new BNode(o, n) );
        return n.leftChild(); }
    public Position rightChild(Position v) throws..{ ... }
    public Position setRight(Position v, Object c) throws..{.}
    private BNode ok(Position p) throws InvalidPosExc {
        if ( p==null || !(p instanceof BNode) )
            throw new InvalidPosExc();
        return (BNode)p; }
    public int size() { return size(root); }
    private int size(Position p) { if (isExternal(p)) return 1;
        else return size(leftChild(p)) + size(rightChild(p)) +1; }
```

```
public boolean isInternal(Position v) throws ... {
    BNode n= ok(v);
    return n.leftChild()!=null || n.rightChild()!=null); }
public boolean isExternal(Position v) throws ... {
    return ! isInternal(v); }

public void expandExternal(Position v) {
    BNode n= ok(v);
    if (isExternal(n)) {
        n.setLeft( new BNode(null, n) );
        n.setRight( newBNode(null, n) ); } }

public void removeAboveExternal(Position v) throws...{
    BNode n = ok(v); // Object o = v.element();
    if ( isInternal(n) || isRoot(n) ) throw ...
    BNode par = ok(parent(n)) ;
    BNode sib = ok(sibling(n));
    par.setElement(sib.element());
    par.setLeft(sib.leftChild());
```



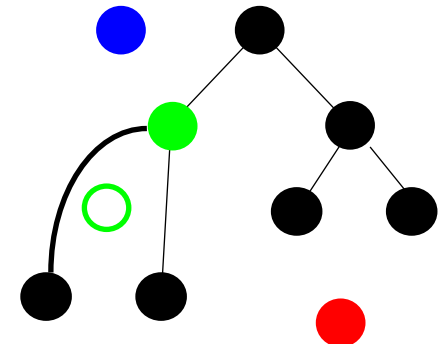
LenketStruktur implementasjon av BinaryTree ADT

```
public class BTreeLS implements BinTree {
    private Position root;
    public BTreeLS(Object e) {
        root = new BNode(e, null); }
    public Position parent(Position v) throws ... {
        return ok(v).parent(); }
    public Position leftChild(Position v) throws ... {
        return ok(v).leftChild(); }
    public Position setLeft(Position v, Object o) throws...{
        BNode n= ok(v);
        if (isExternal(n)) expandExternal(n);
        n.setLeft( new BNode(o, n) );
        return n.leftChild(); }
    public Position rightChild(Position v) throws..{ ... }
    public Position setRight(Position v, Object c) throws..{.}
    private BNode ok(Position p) throws InvalidPosExc {
        if ( p==null || !(p instanceof BNode) )
            throw new InvalidPosExc();
        return (BNode)p; }
    public int size() { return size(root); }
    private int size(Position p) { if (isExternal(p)) return 1;
        else return size(leftChild(p)) + size(rightChild(p)) +1; }
```

```
public boolean isInternal(Position v) throws ... {
    BNode n= ok(v);
    return n.leftChild()!=null || n.rightChild()!=null); }
public boolean isExternal(Position v) throws ... {
    return ! isInternal(v); }

public void expandExternal(Position v) {
    BNode n= ok(v);
    if (isExternal(n)) {
        n.setLeft( new BNode(null, n) );
        n.setRight( newBNode(null, n) ); } }

public void removeAboveExternal(Position v) throws...{
    BNode n = ok(v); // Object o = v.element();
    if ( isInternal(n) || isRoot(n) ) throw ...
    BNode par = ok(parent(n)) ;
    BNode sib = ok(sibling(n));
    par.setElement(sib.element());
    par.setLeft(sib.leftChild());
    par.setRight(sib.rightChild());
}
```



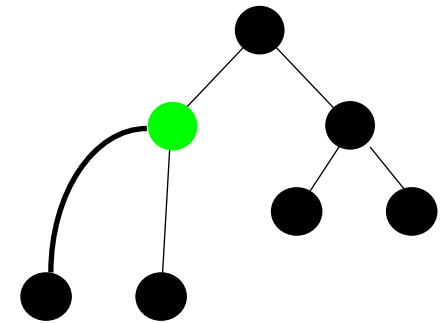
LenketStruktur implementasjon av BinaryTree ADT

```
public class BTreeLS implements BinTree {
    private Position root;
    public BTreeLS(Object e) {
        root = new BNode(e, null); }
    public Position parent(Position v) throws ... {
        return ok(v).parent(); }
    public Position leftChild(Position v) throws ... {
        return ok(v).leftChild(); }
    public Position setLeft(Position v, Object o) throws...{
        BNode n= ok(v);
        if (isExternal(n)) expandExternal(n);
        n.setLeft( new BNode(o, n) );
        return n.leftChild(); }
    public Position rightChild(Position v) throws..{ ... }
    public Position setRight(Position v, Object c) throws..{.}
    private BNode ok(Position p) throws InvalidPosExc {
        if ( p==null || !(p instanceof BNode) )
            throw new InvalidPosExc();
        return (BNode)p; }
    public int size() { return size(root); }
    private int size(Position p) { if (isExternal(p)) return 1;
        else return size(leftChild(p)) + size(rightChild(p)) +1; }
```

```
public boolean isInternal(Position v) throws ... {
    BNode n= ok(v);
    return n.leftChild()!=null || n.rightChild()!=null); }
public boolean isExternal(Position v) throws ... {
    return ! isInternal(v); }

public void expandExternal(Position v) {
    BNode n= ok(v);
    if (isExternal(n)) {
        n.setLeft( new BNode(null, n) );
        n.setRight( newBNode(null, n) ); } }

public void removeAboveExternal(Position v) throws...{
    BNode n = ok(v); // Object o = v.element();
    if ( isInternal(n) || isRoot(n) ) throw ...
    BNode par = ok(parent(n)) ;
    BNode sib = ok(sibling(n));
    par.setElement(sib.element());
    par.setLeft(sib.leftChild());
    par.setRight(sib.rightChild());
}
```



alt unntatt positions(), elements(), size() er O(1)

Sequence implementasjon av BinaryTree ADT

I. DATA REPRESENTASJON

for en **Position** v i treet T , la $sn(v)$ være et tall gitt ved:

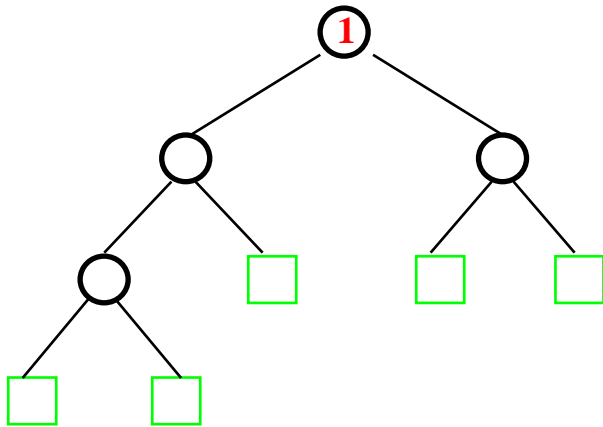
- hvis $T.isRoot(v)$ så $sn(v) = 1$
- hvis $T.leftChild(v) == u$ så $sn(u) = 2 * sn(v)$
- hvis $T.rightChild(v) == u$ så $sn(u) = 2 * sn(v) + 1$

Sequence implementasjon av BinaryTree ADT

I. DATA REPRESENTASJON

for en **Position** v i treet T , la $sn(v)$ være et tall gitt ved:

- hvis $T.isRoot(v)$ så $sn(v) = 1$
- hvis $T.leftChild(v) == u$ så $sn(u) = 2 * sn(v)$
- hvis $T.rightChild(v) == u$ så $sn(u) = 2 * sn(v) + 1$

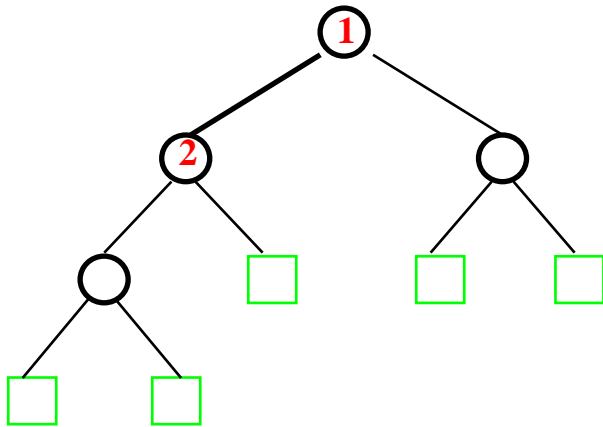


Sequence implementasjon av BinaryTree ADT

I. DATA REPRESENTASJON

for en **Position** v i treet T , la $sn(v)$ være et tall gitt ved:

- hvis $T.isRoot(v)$ så $sn(v) = 1$
- hvis $T.leftChild(v) == u$ så $sn(u) = 2*sn(v)$
- hvis $T.rightChild(v) == u$ så $sn(u) = 2*sn(v) + 1$

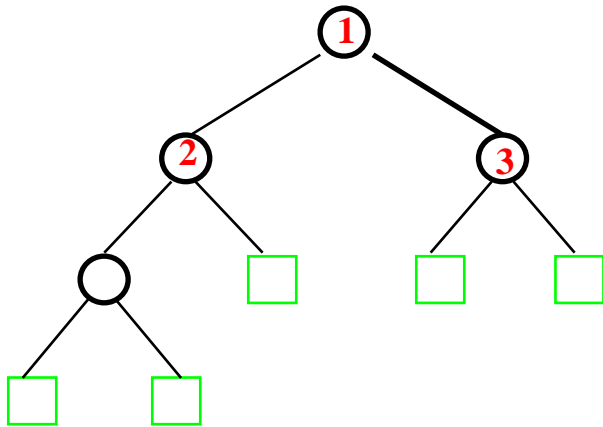


Sequence implementasjon av BinaryTree ADT

I. DATA REPRESENTASJON

for en **Position** v i treet T , la $sn(v)$ være et tall gitt ved:

- hvis $T.isRoot(v)$ så $sn(v) = 1$
- hvis $T.leftChild(v) == u$ så $sn(u) = 2 * sn(v)$
- hvis $T.rightChild(v) == u$ så $sn(u) = 2 * sn(v) + 1$

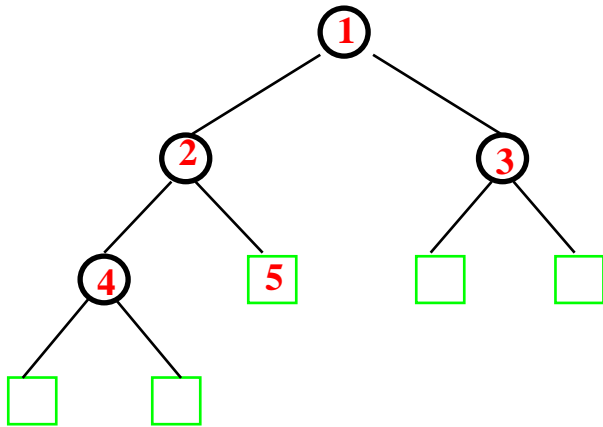


Sequence implementasjon av BinaryTree ADT

I. DATA REPRESENTASJON

for en Position v i treet T , la $sn(v)$ være et tall gitt ved:

- hvis $T.isRoot(v)$ så $sn(v) = 1$
- hvis $T.leftChild(v) == u$ så $sn(u) = 2 * sn(v)$
- hvis $T.rightChild(v) == u$ så $sn(u) = 2 * sn(v) + 1$

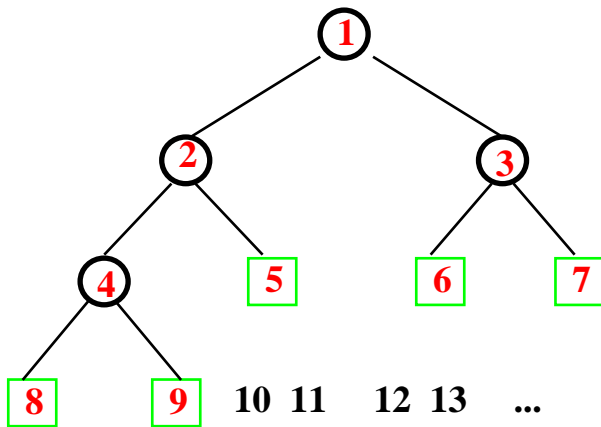


Sequence implementasjon av BinaryTree ADT

I. DATA REPRESENTASJON

for en Position v i treet T , la $sn(v)$ være et tall gitt ved:

- hvis $T.isRoot(v)$ så $sn(v) = 1$
- hvis $T.leftChild(v) == u$ så $sn(u) = 2 * sn(v)$
- hvis $T.rightChild(v) == u$ så $sn(u) = 2 * sn(v) + 1$



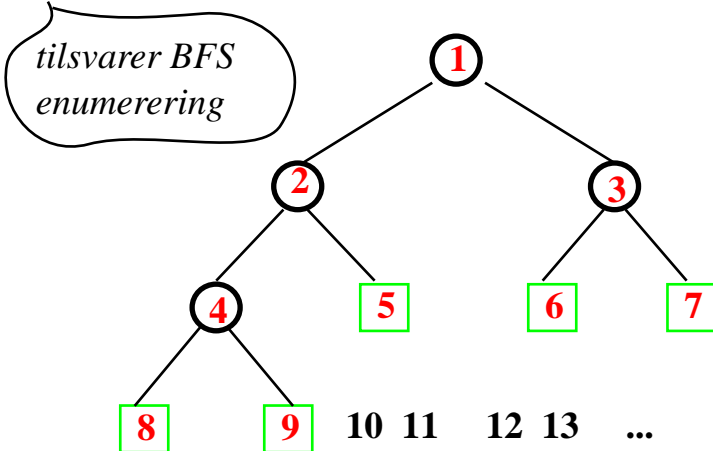
Sequence implementasjon av BinaryTree ADT

I. DATA REPRESENTASJON

for en **Position** v i treet T , la $sn(v)$ være et tall gitt ved:

- hvis $T.isRoot(v)$ så $sn(v) = 1$
- hvis $T.leftChild(v) == u$ så $sn(u) = 2 * sn(v)$
- hvis $T.rightChild(v) == u$ så $sn(u) = 2 * sn(v) + 1$

sn bestemmer nodens stilling i sekvensen



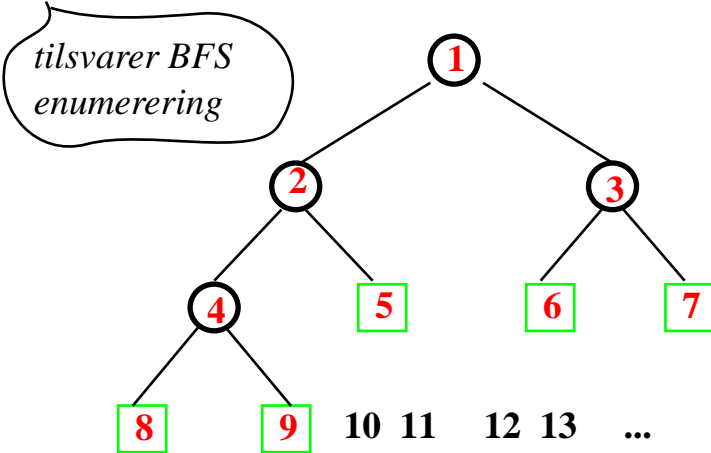
Sequence implementasjon av BinaryTree ADT

I. DATA REPRESENTASJON

for en Position v i treet T , la $sn(v)$ være et tall gitt ved:

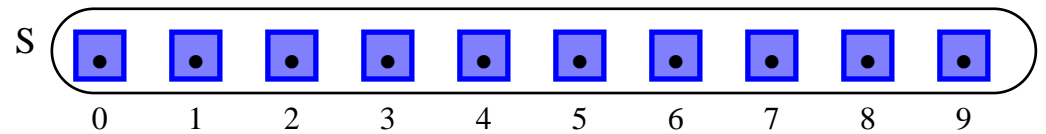
- hvis $T.isRoot(v)$ så $sn(v) = 1$
- hvis $T.leftChild(v) == u$ så $sn(u) = 2 * sn(v)$
- hvis $T.rightChild(v) == u$ så $sn(u) = 2 * sn(v) + 1$

sn bestemmer nodens stilling i sekvensen



II. DATA STRUKTUR

Sequence S // *Sekvens-Position er Tree-Position*



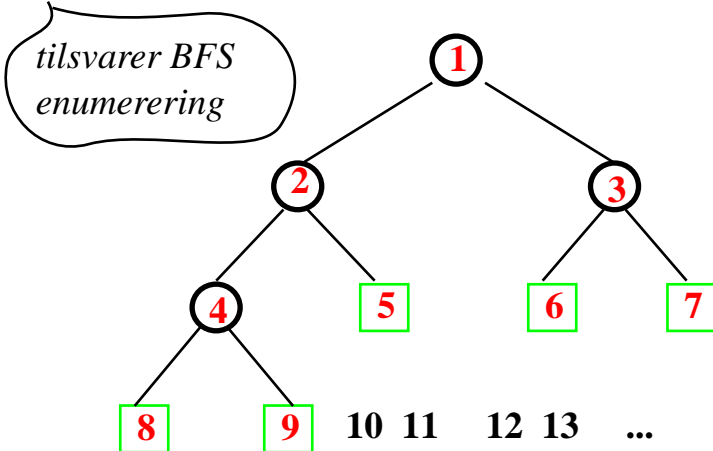
Sequence implementasjon av BinaryTree ADT

I. DATA REPRESENTASJON

for en Position v i treet T , la $sn(v)$ være et tall gitt ved:

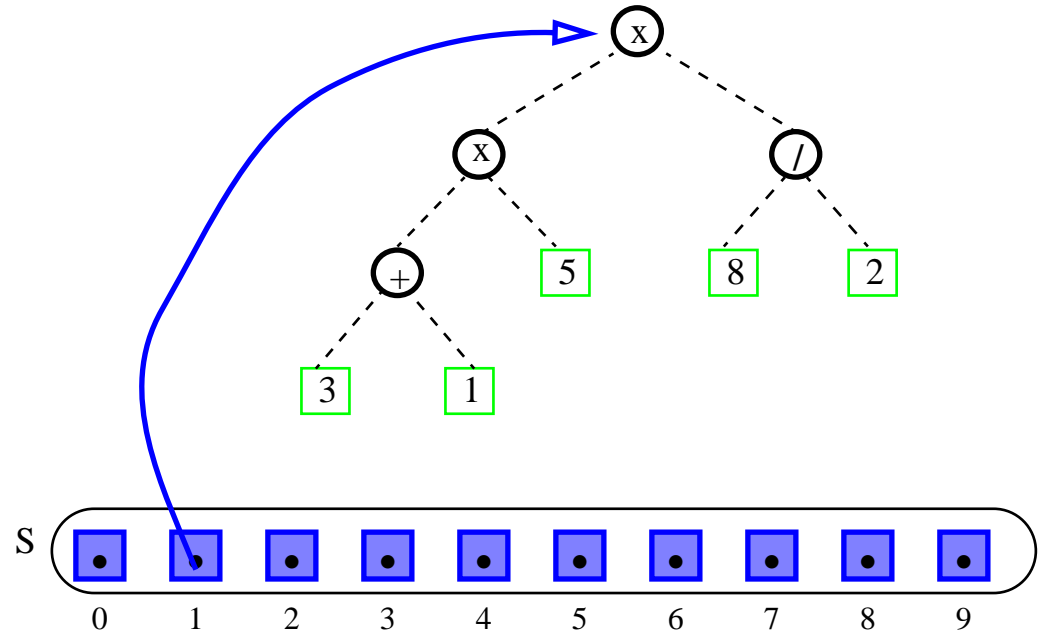
- hvis $T.isRoot(v)$ så $sn(v) = 1$
- hvis $T.leftChild(v) == u$ så $sn(u) = 2 * sn(v)$
- hvis $T.rightChild(v) == u$ så $sn(u) = 2 * sn(v) + 1$

sn bestemmer nodens stilling i sekvensen



II. DATA STRUKTUR

Sequence S // Sekvens-Position er Tree-Position



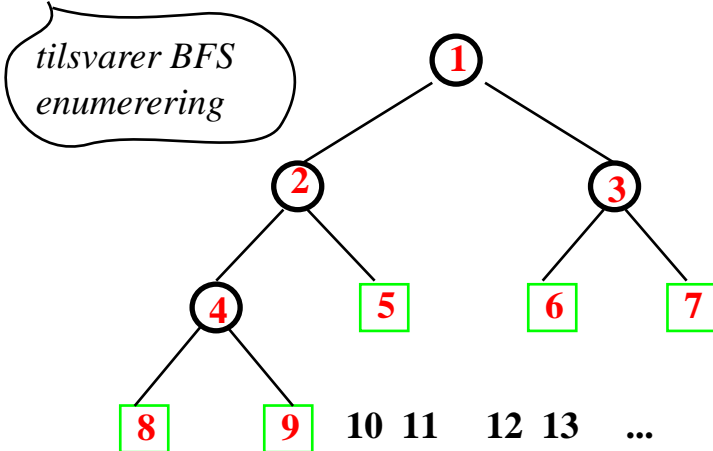
Sequence implementasjon av BinaryTree ADT

I. DATA REPRESENTASJON

for en Position v i treet T , la $sn(v)$ være et tall gitt ved:

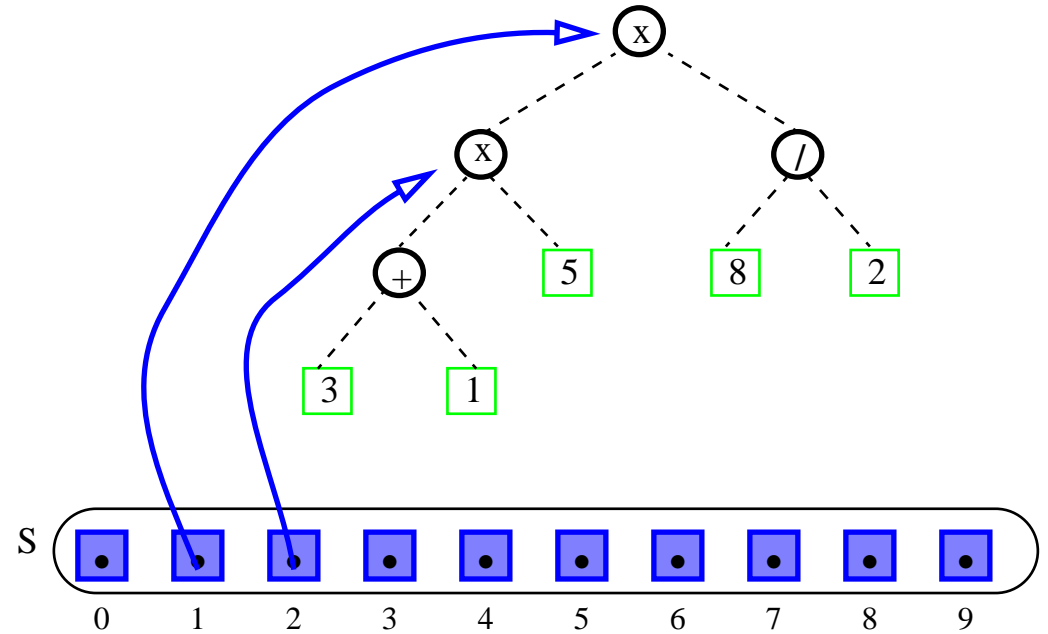
- hvis $T.isRoot(v)$ så $sn(v) = 1$
- hvis $T.leftChild(v) == u$ så $sn(u) = 2 * sn(v)$
- hvis $T.rightChild(v) == u$ så $sn(u) = 2 * sn(v) + 1$

sn bestemmer nodens stilling i sekvensen



II. DATA STRUKTUR

Sequence S // Sekvens-Position er Tree-Position



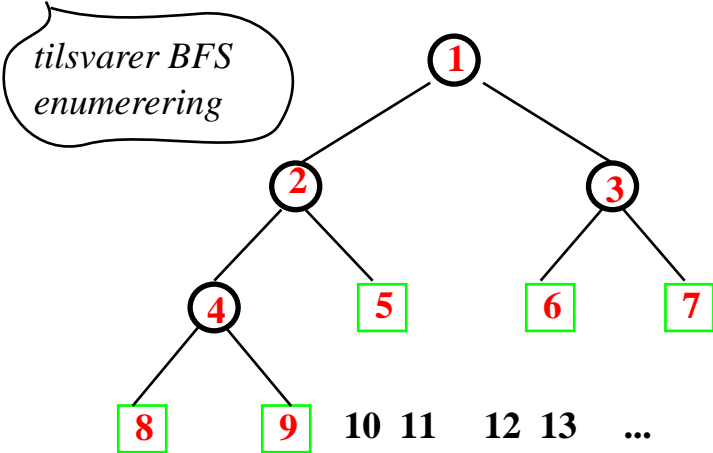
Sequence implementasjon av BinaryTree ADT

I. DATA REPRESENTASJON

for en Position v i treet T , la $sn(v)$ være et tall gitt ved:

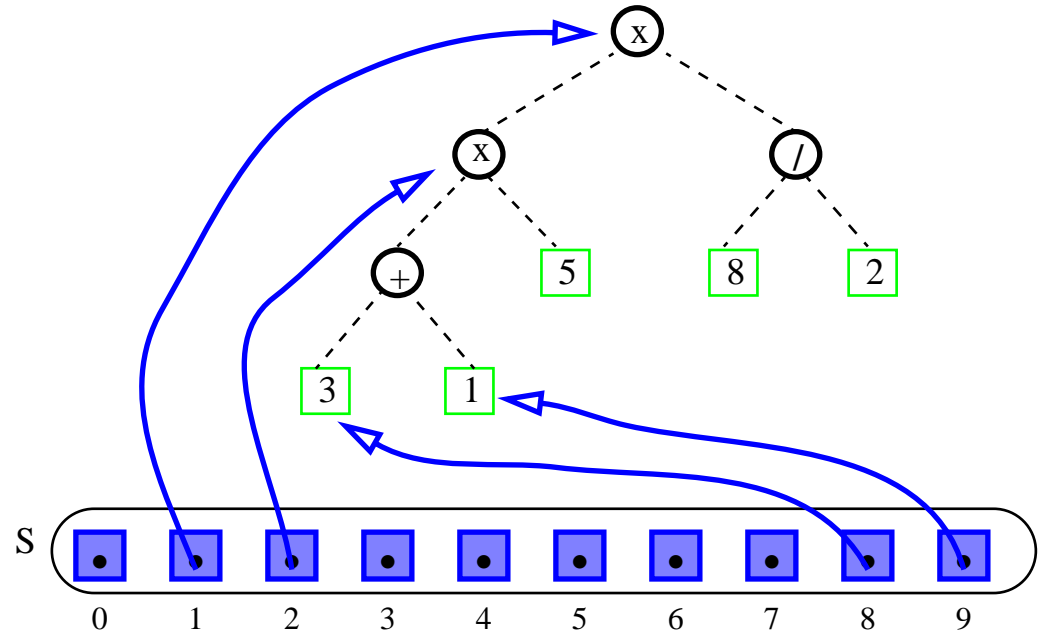
- hvis $T.isRoot(v)$ så $sn(v) = 1$
- hvis $T.leftChild(v) == u$ så $sn(u) = 2 * sn(v)$
- hvis $T.rightChild(v) == u$ så $sn(u) = 2 * sn(v) + 1$

sn bestemmer nodens stilling i sekvensen



II. DATA STRUKTUR

Sequence S // Sekvens-Position er Tree-Position



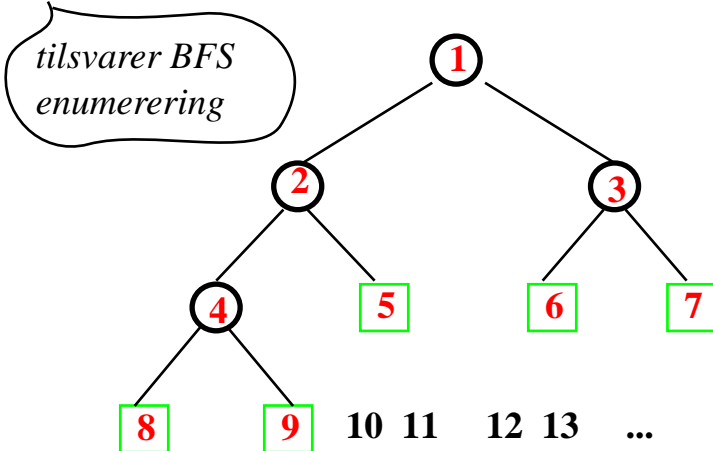
Sequence implementasjon av BinaryTree ADT

I. DATA REPRESENTASJON

for en Position v i treet T , la $sn(v)$ være et tall gitt ved:

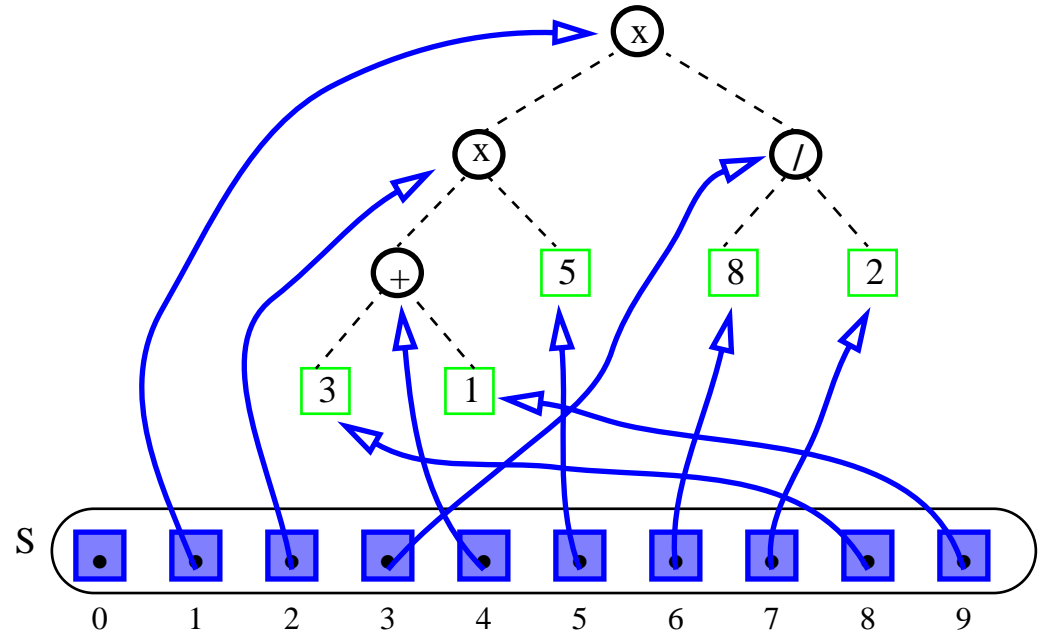
- hvis $T.isRoot(v)$ så $sn(v) = 1$
- hvis $T.leftChild(v) == u$ så $sn(u) = 2 * sn(v)$
- hvis $T.rightChild(v) == u$ så $sn(u) = 2 * sn(v) + 1$

sn bestemmer nodens stilling i sekvensen



II. DATA STRUKTUR

Sequence S // Sekvens-Position er Tree-Position



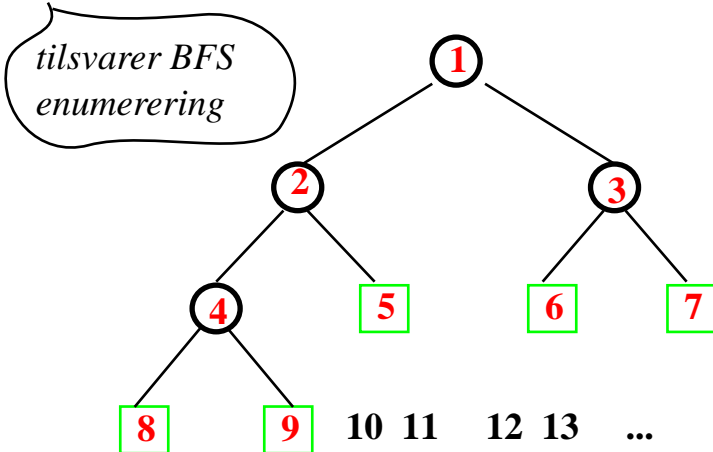
Sequence implementasjon av BinaryTree ADT

I. DATA REPRESENTASJON

for en Position v i treet T , la $sn(v)$ være et tall gitt ved:

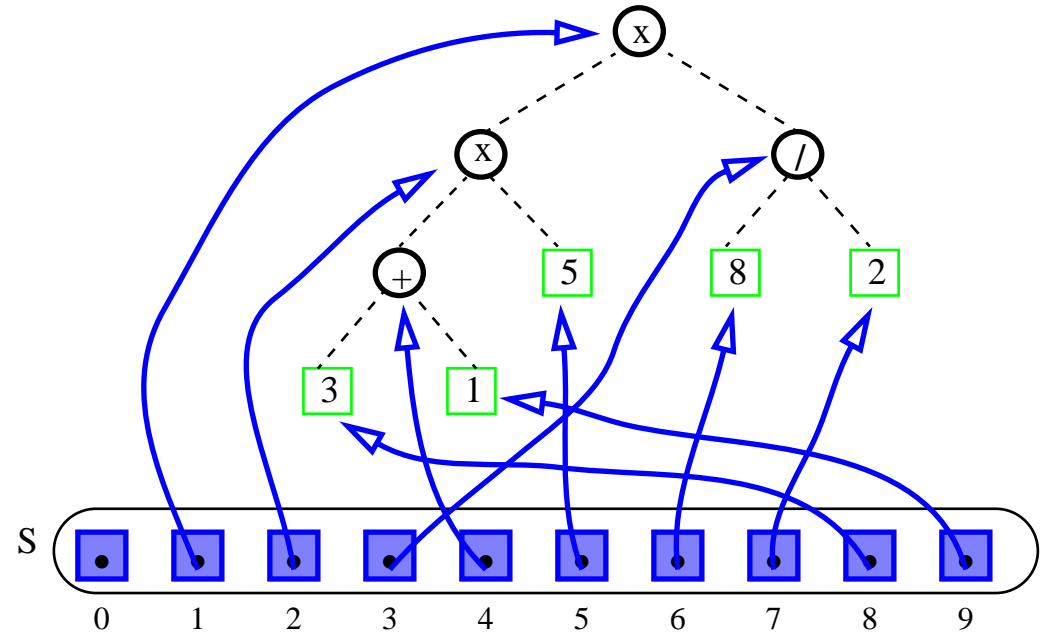
- hvis $T.isRoot(v)$ så $sn(v) = 1$
- hvis $T.leftChild(v) == u$ så $sn(u) = 2 * sn(v)$
- hvis $T.rightChild(v) == u$ så $sn(u) = 2 * sn(v) + 1$

sn bestemmer nodens stilling i sekvensen



II. DATA STRUKTUR

Sequence S // Sekvens-Position er Tree-Position



III. DATA INVARIANT :

- $S.atRank[1] == root$
- $S.atRank[2*v] == T.leftChild(S.atRank[v])$
- $S.atRank[2*v+1] == T.rightChild(S.atRank[v])$

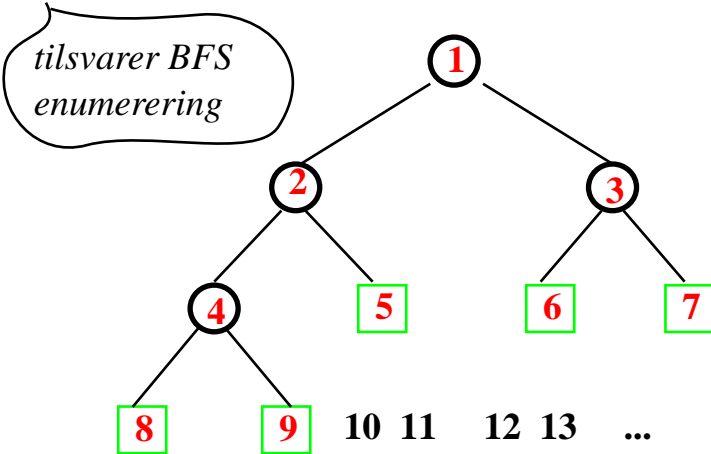
Sequence implementasjon av BinaryTree ADT

I. DATA REPRESENTASJON

for en Position v i treet T , la $sn(v)$ være et tall gitt ved:

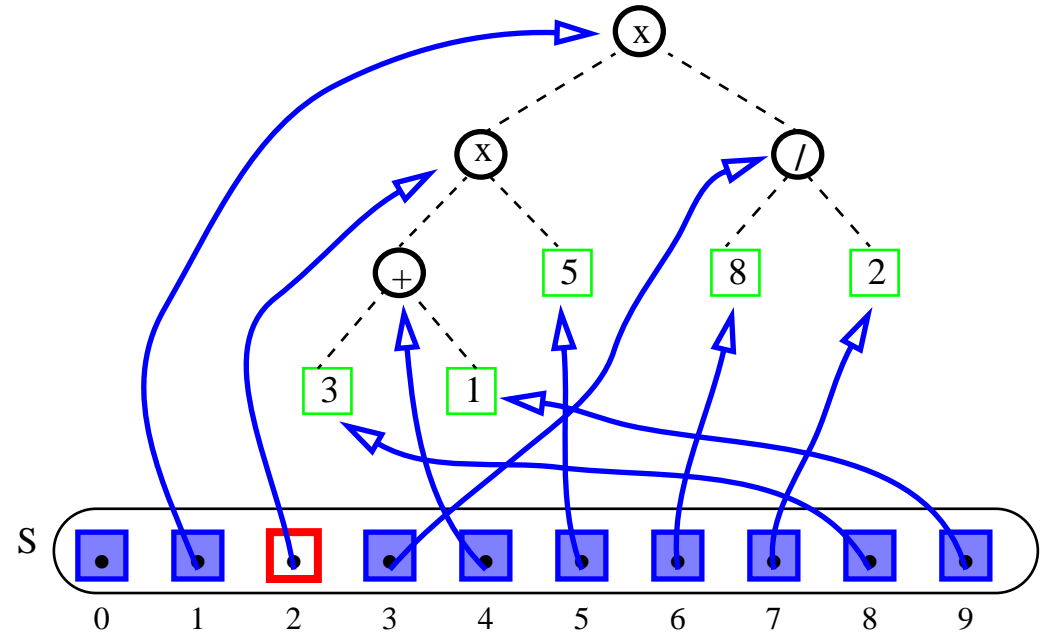
- hvis $T.isRoot(v)$ så $sn(v) = 1$
- hvis $T.leftChild(v) == u$ så $sn(u) = 2*sn(v)$
- hvis $T.rightChild(v) == u$ så $sn(u) = 2*sn(v) + 1$

sn bestemmer nodens stilling i sekvensen



II. DATA STRUKTUR

Sequence S // Sekvens-Position er Tree-Position



```
Position leftChild(Position p) {
    return S . atRank( S.rankOf(p)*2 );}
```

III. DATA INVARIANT :

- $S.atRank[1] == root$
- $S.atRank[2*v] == T.leftChild(S.atRank[v])$
- $S.atRank[2*v+1] == T.rightChild(S.atRank[v])$

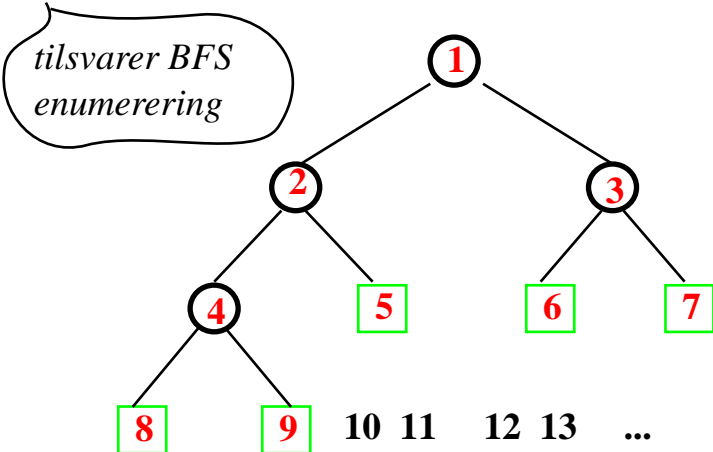
Sequence implementasjon av BinaryTree ADT

I. DATA REPRESENTASJON

for en **Position** v i treet T , la $sn(v)$ være et tall gitt ved:

- hvis $T.isRoot(v)$ så $sn(v) = 1$
- hvis $T.leftChild(v) == u$ så $sn(u) = 2*sn(v)$
- hvis $T.rightChild(v) == u$ så $sn(u) = 2*sn(v) + 1$

sn bestemmer nodens stilling i sekvensen

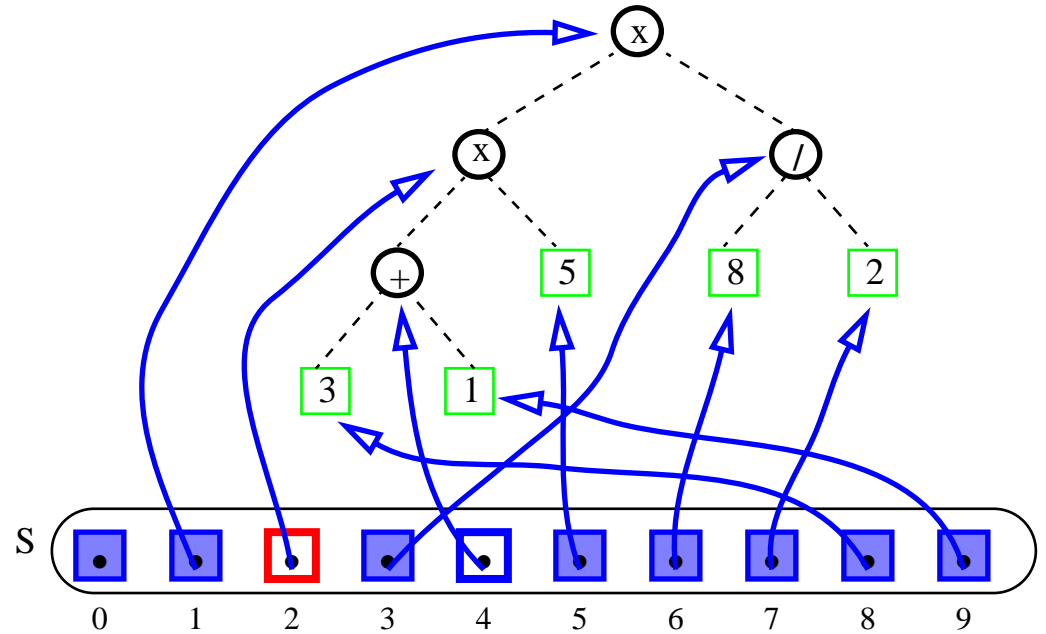


III. DATA INVARIANT :

- $S.atRank[1] == root$
- $S.atRank[2*v] == T.leftChild(S.atRank[v])$
- $S.atRank[2*v+1] == T.rightChild(S.atRank[v])$

II. DATA STRUKTUR

Sequence S // Sekvens-Position er Tree-Position



```
Position leftChild(Position p) {
    return S . atRank( S.rankOf(p)*2 );}
```

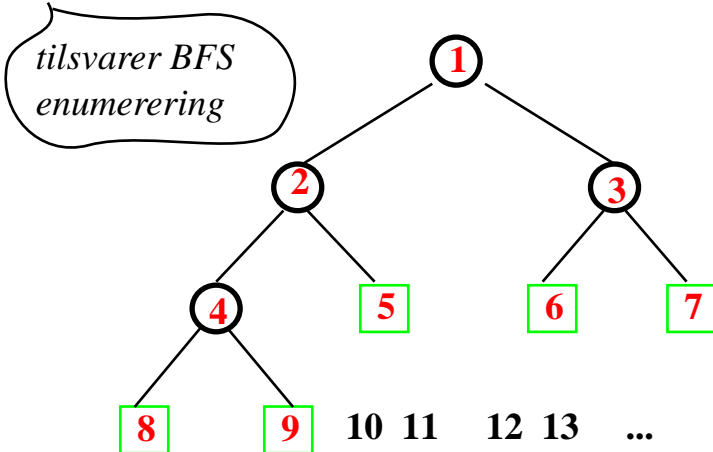
Sequence implementasjon av BinaryTree ADT

I. DATA REPRESENTASJON

for en Position v i treet T , la $sn(v)$ være et tall gitt ved:

- hvis $T.isRoot(v)$ så $sn(v) = 1$
- hvis $T.leftChild(v) == u$ så $sn(u) = 2*sn(v)$
- hvis $T.rightChild(v) == u$ så $sn(u) = 2*sn(v) + 1$

sn bestemmer nodens stilling i sekvensen

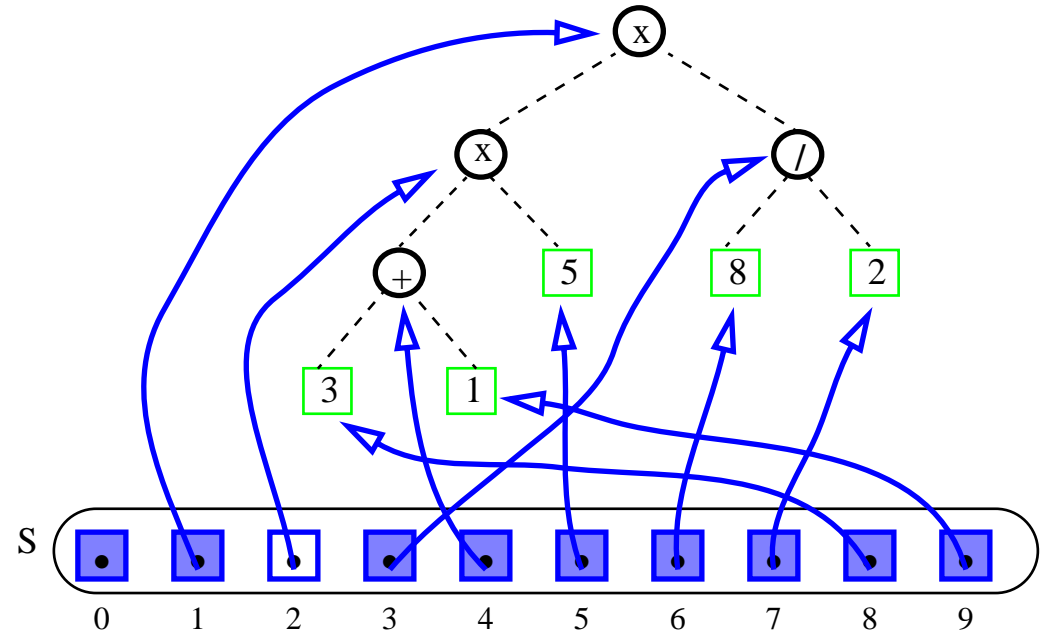


III. DATA INVARIANT :

- $S.atRank[1] == root$
- $S.atRank[2*v] == T.leftChild(S.atRank[v])$
- $S.atRank[2*v+1] == T.rightChild(S.atRank[v])$

II. DATA STRUKTUR

Sequence S // Sekvens-Position er Tree-Position



```
Position leftChild(Position p) {  
    return S . atRank( S.rankOf(p)*2 );  
}  
Position root() { return S . atRank(1); }
```

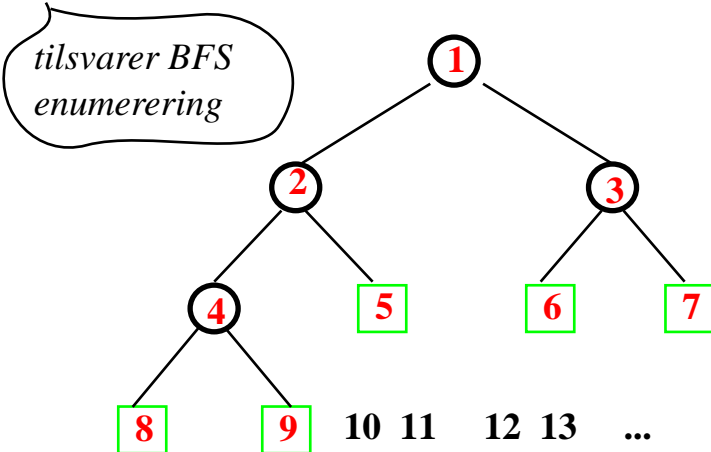
Sequence implementasjon av BinaryTree ADT

I. DATA REPRESENTASJON

for en Position v i treet T , la $sn(v)$ være et tall gitt ved:

- hvis $T.isRoot(v)$ så $sn(v) = 1$
- hvis $T.leftChild(v) == u$ så $sn(u) = 2*sn(v)$
- hvis $T.rightChild(v) == u$ så $sn(u) = 2*sn(v) + 1$

sn bestemmer nodens stilling i sekvensen

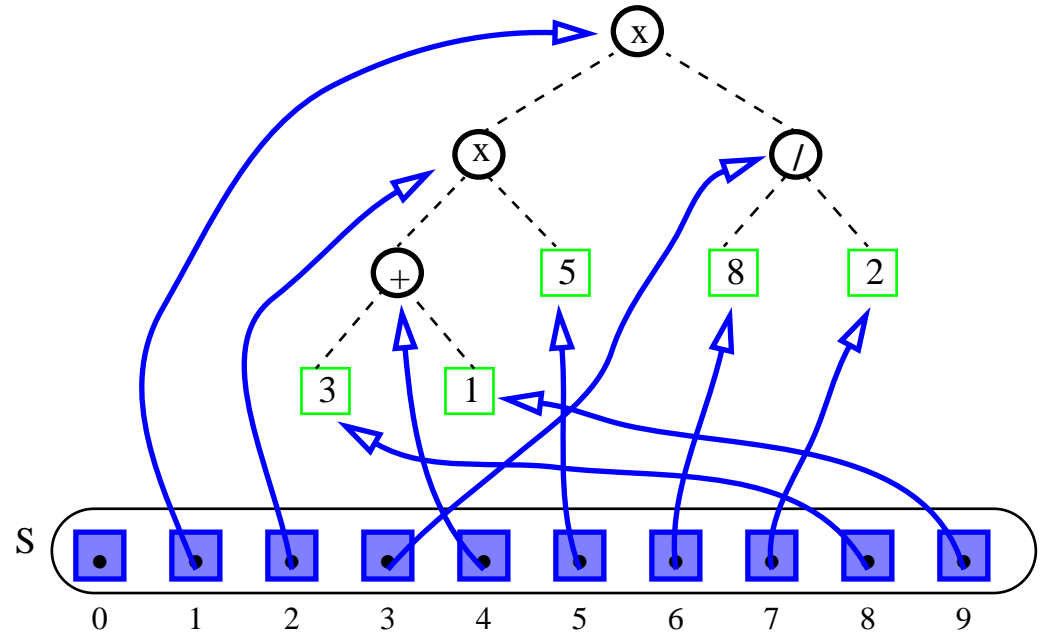


III. DATA INVARIANT :

- $S.atRank[1] == root$
- $S.atRank[2*v] == T.leftChild(S.atRank[v])$
- $S.atRank[2*v+1] == T.rightChild(S.atRank[v])$

II. DATA STRUKTUR

Sequence S // Sekvens-Position er Tree-Position

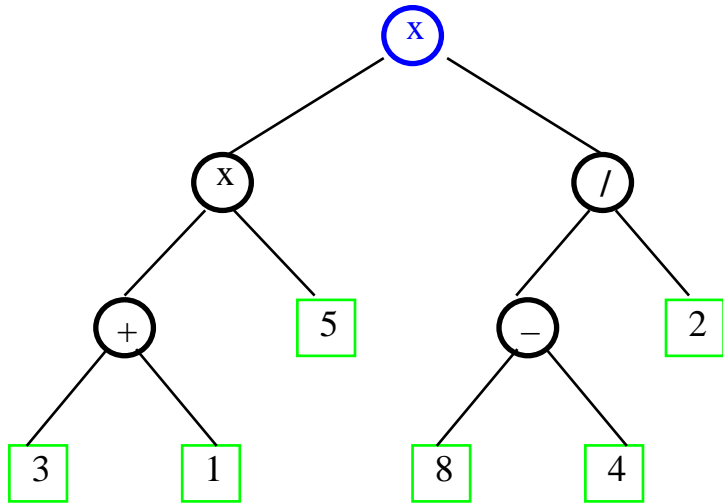


```
Position leftChild(Position p) {
    return S . atRank( S.rankOf(p)*2 );
}
Position root() { return S . atRank(1); }
```

- alle BinaryTree operasjoner (unntatt positions(), elements()) er $O(1)$ relativt til Sequence
- Spesielt adekvat for "komplette" binære trær

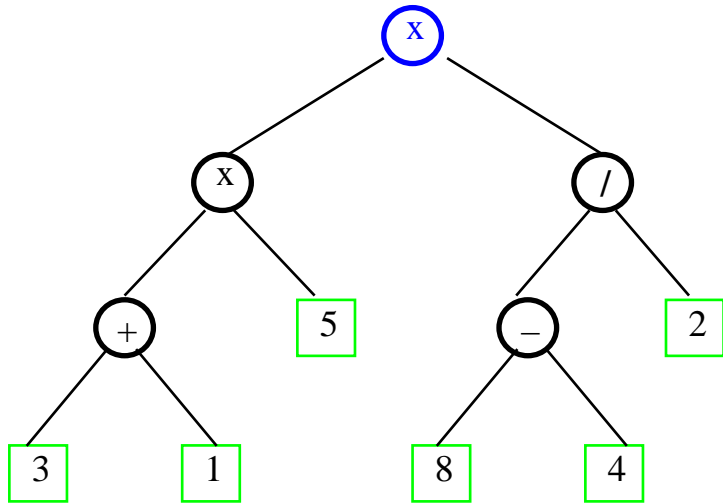
6. Syntakstrær

$((3 + 1) \times 5) \times (8 - 4) / 2$



6. Syntakstrær

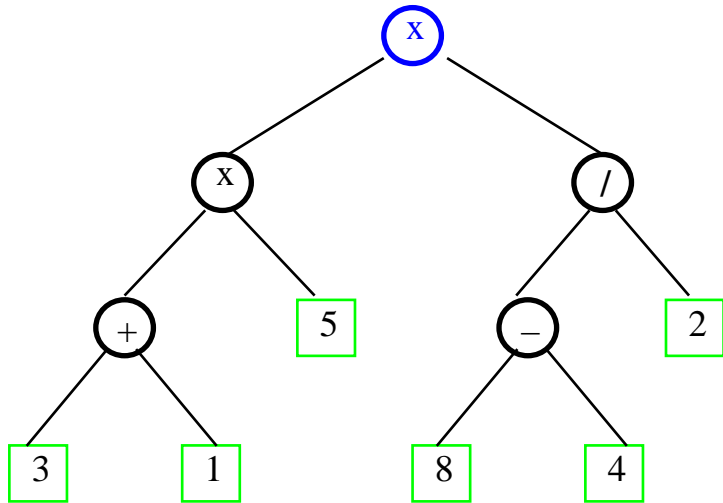
$((3 + 1) \times 5) \times (8 - 4) / 2$



```
if (a == b)
if (c == d) then x = 5;
else x = 10;
```


6. Syntakstrær

$((3 + 1) \times 5) \times (8 - 4) / 2$



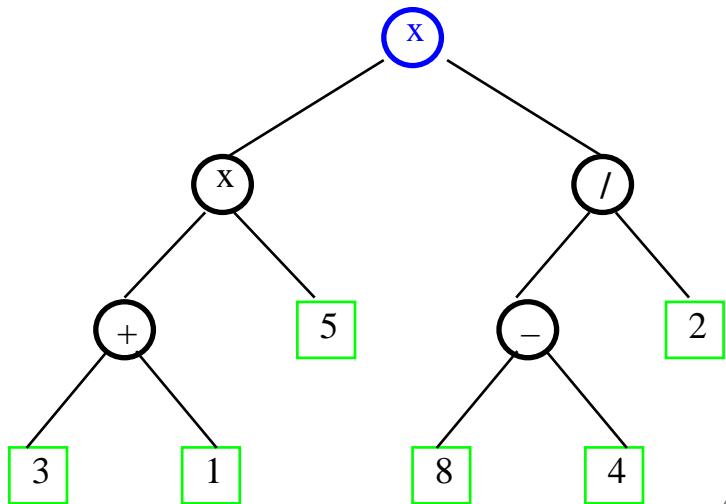
```
if (a == b)
  if (c == d) then x = 5;
  else x = 10;
```

```
if (a == b) {
  if (c == d) then x = 5;
  else x = 10;
}
```

```
if (a == b)
  { if (c == d) then x = 5; }
  else x = 10;
```

6. Syntakstrær

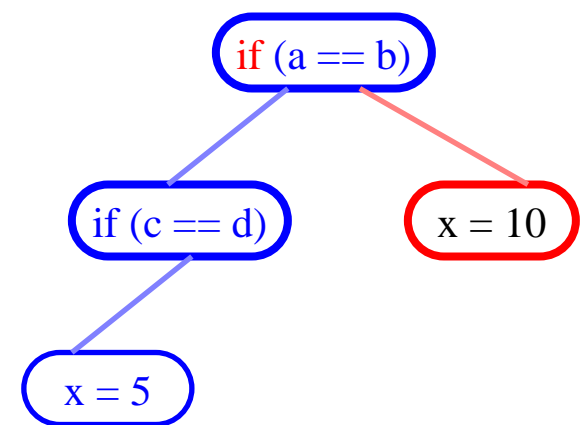
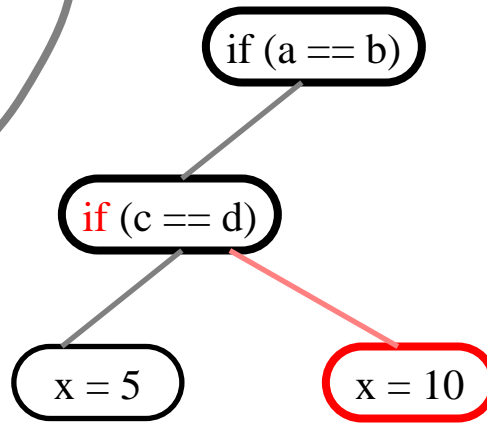
((3 + 1) x 5) x ((8 - 4) / 2)



if (a == b)
if (c == d) then x = 5;
else x = 10;

if (a == b) {
if (c == d) then x = 5;
else x = 10;
}

if (a == b)
{ if (c == d) then x = 5; }
else x = 10;



6. Ordbok-søking

...
bie
bil
bilateral
bilbelte
bilde
bildekk
bildende
bile
....

finn(String X)

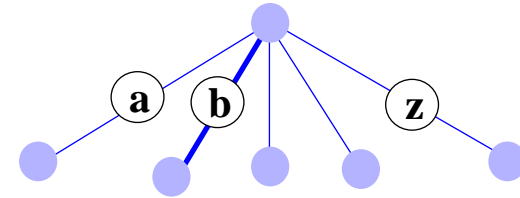
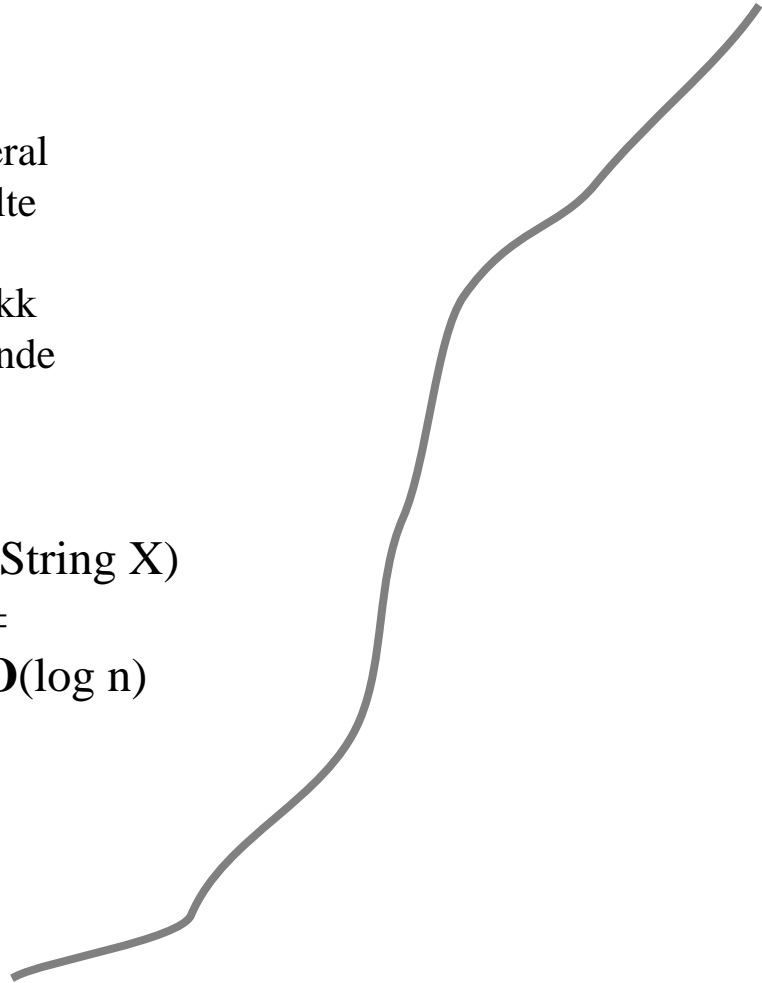
=

O(log n)

6. Ordbok-søking (totalt ordnede barn)

...
bie
bil
bilateral
bilbelte
bilde
bildekk
bildende
bile
....

finn(String X)
=
 $O(\log n)$

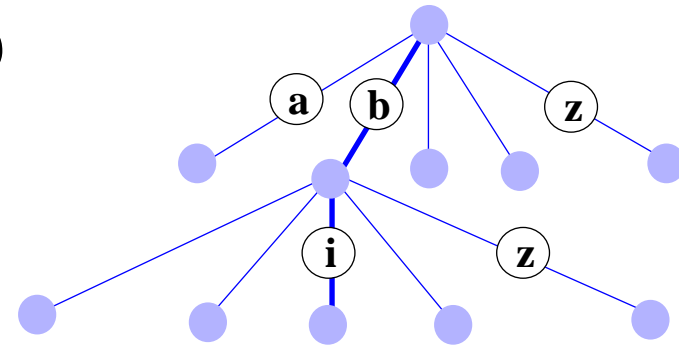


(tries)

6. Ordbok-søking (totalt ordnede barn)

...
bie
bil
bilateral
bilbelte
bilde
bildekk
bildende
bile
....

finn(String X)
=
 $O(\log n)$

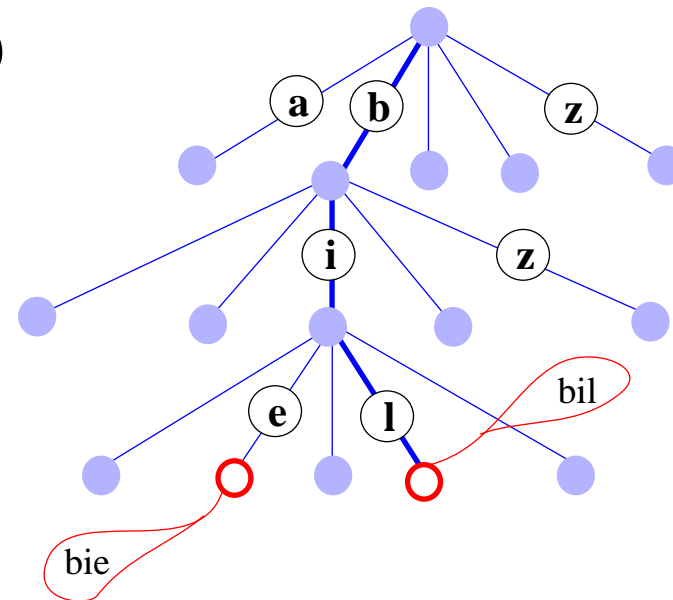


(tries)

6. Ordbok-søking (totalt ordnede barn)

...
bie
bil
bilateral
bilbelte
bilde
bildekk
bildende
bile
....

finn(String X)
=
 $O(\log n)$

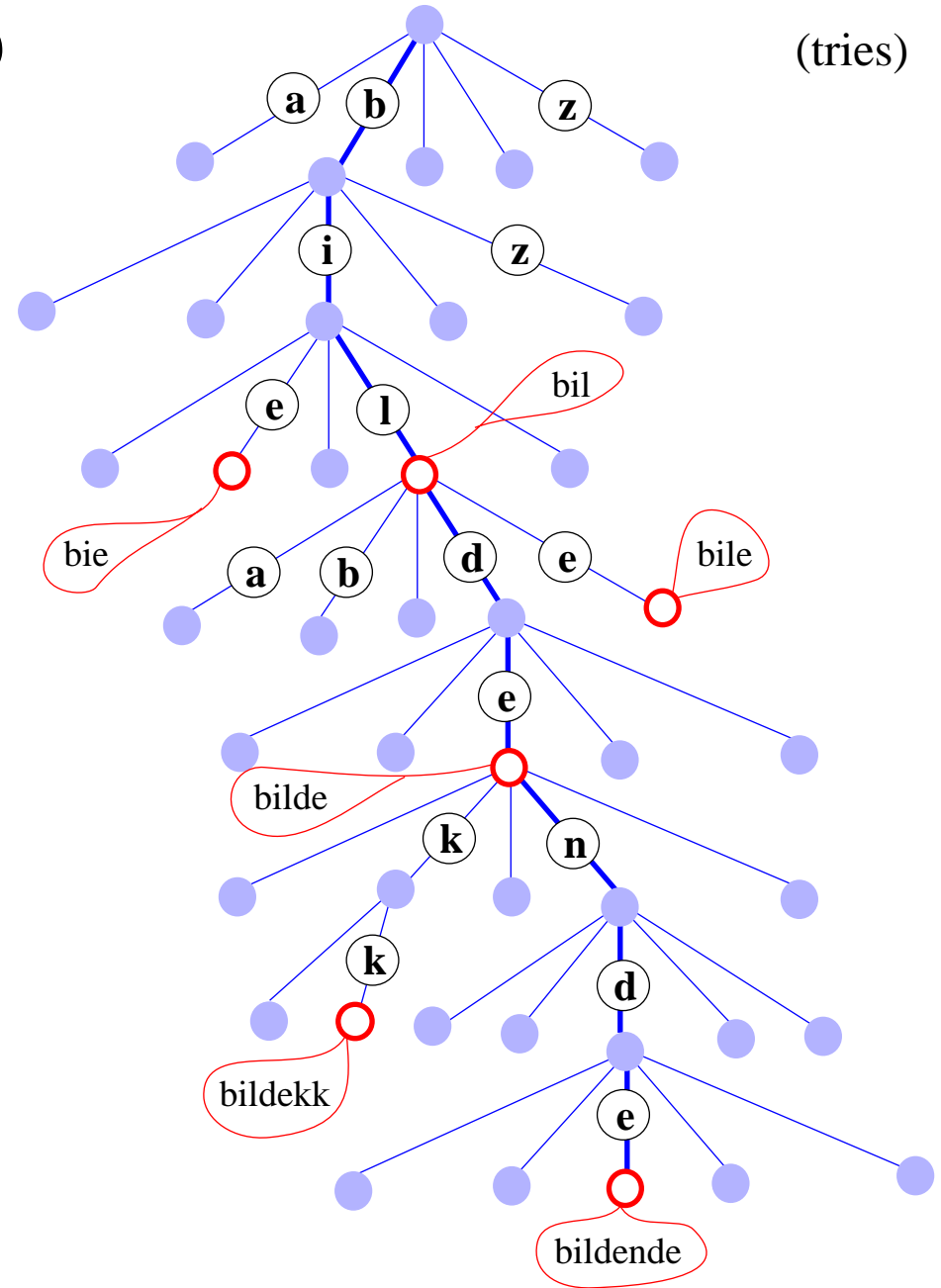


6. Ordbok-søking (totalt ordnede barn)

(tries)

...
bie
bil
bilateral
bilbelte
bilde
bildekk
bildende
bile
....

finn(String X)
=
O(log n)



6. Ordbok-søking (totalt ordnede barn)

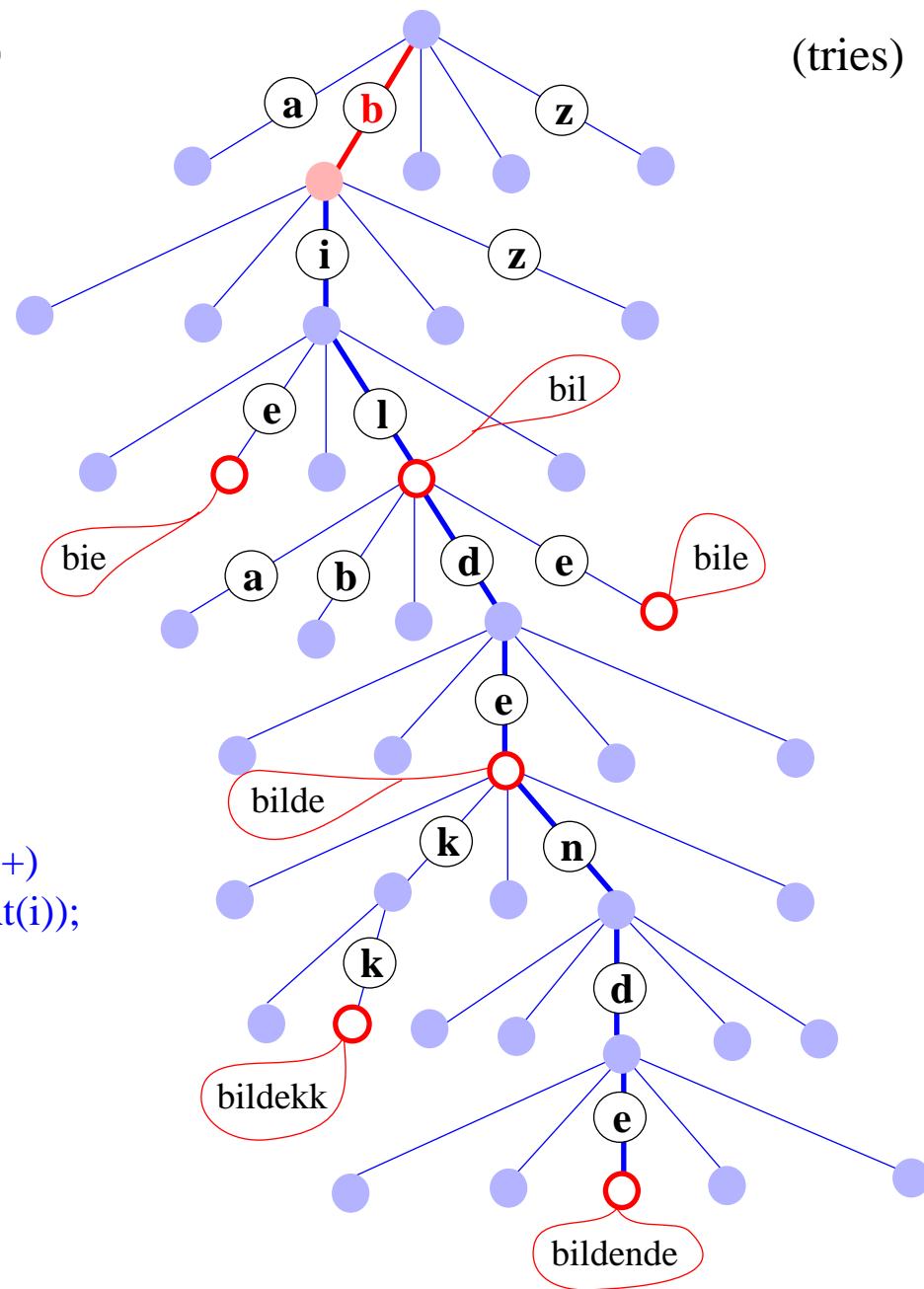
...
 bie
 bil
 bilateral
 bilbelte
 bilde
 bildekk
 bildende
 bile

finn(String X)
 =
 $O(\log n)$

finn(" b i e ")

```

Position finn(String X)
p = root;
for (int i=0; i<X.length;i++)
    p = p.childNo(X.charAt(i));
    if (p == null) break;
if (p == null || !p.isRed())
    finnes-ikke
else return p;
    
```



6. Ordbok-søking (totalt ordnede barn)

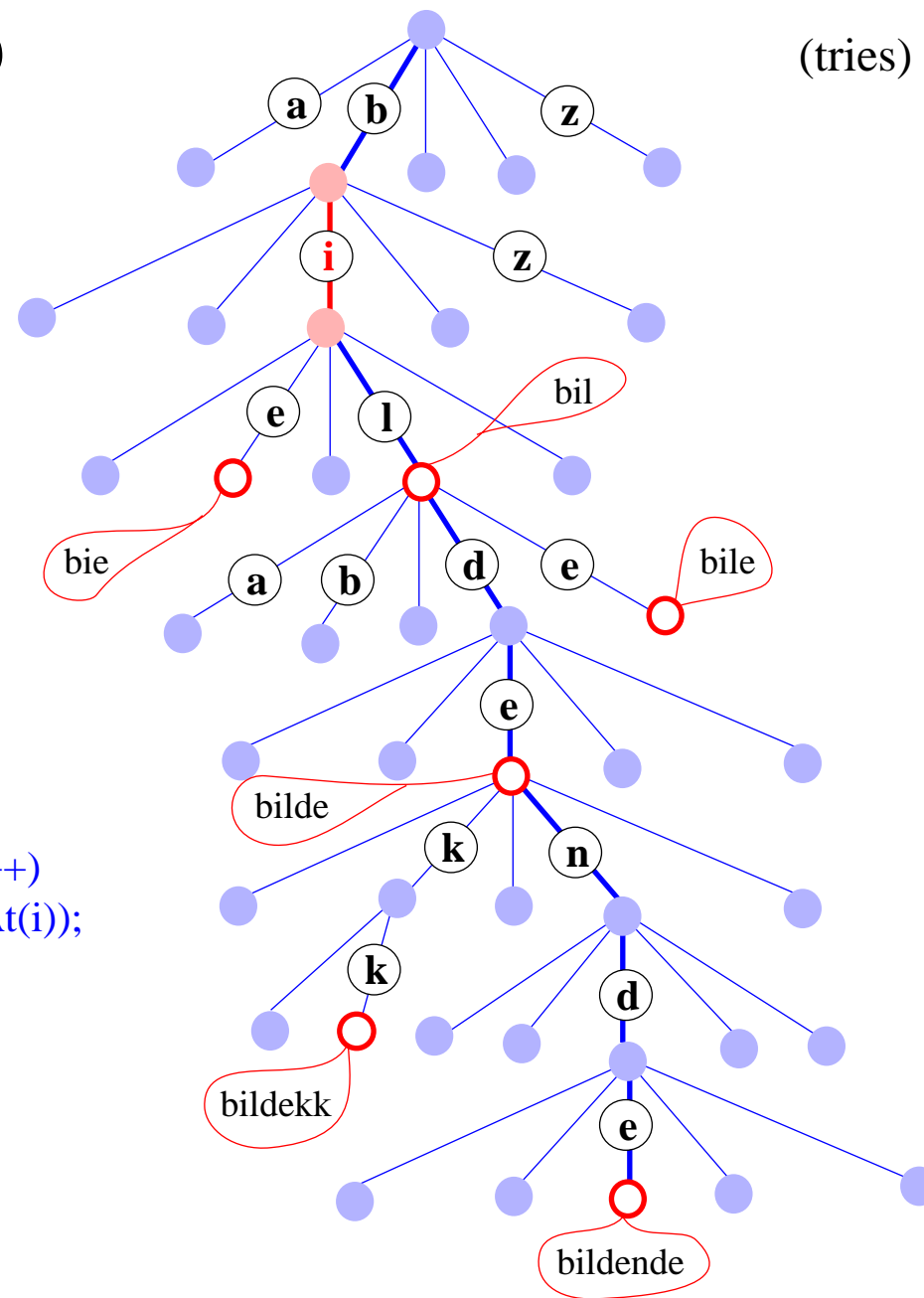
...
 bie
 bil
 bilateral
 bilbelte
 bilde
 bildekk
 bildende
 bile

finn(String X)
 =
 $O(\log n)$

finn(" b i e ")

```

Position finn(String X)
p = root;
for (int i=0; i<X.length;i++)
    p = p.childNo(X.charAt(i));
    if (p == null) break;
if (p == null || !p.isRed())
    finnes-ikke
else return p;
    
```



6. Ordbok-søking (totalt ordnede barn)

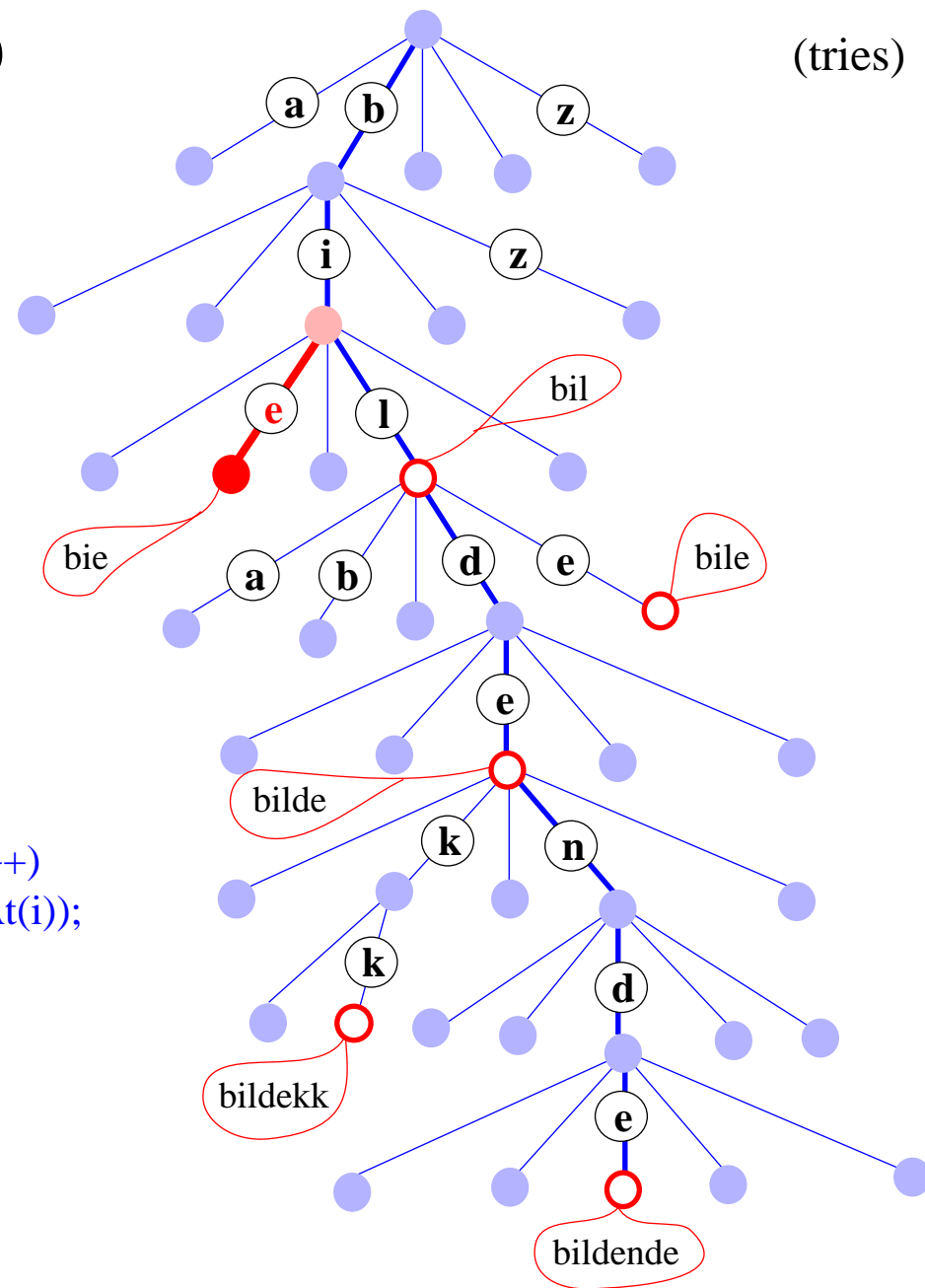
...
 bie
 bil
 bilateral
 bilbelte
 bilde
 bildekk
 bildende
 bile

finn(String X)
 =
 $O(\log n)$

finn(" b i e ")

```

Position finn(String X)
p = root;
for (int i=0; i<X.length;i++)
    p = p.childNo(X.charAt(i));
    if (p == null) break;
if (p == null || !p.isRed())
    finnes-ikke
else return p;
    
```



6. Ordbok-søking (totalt ordnede barn)

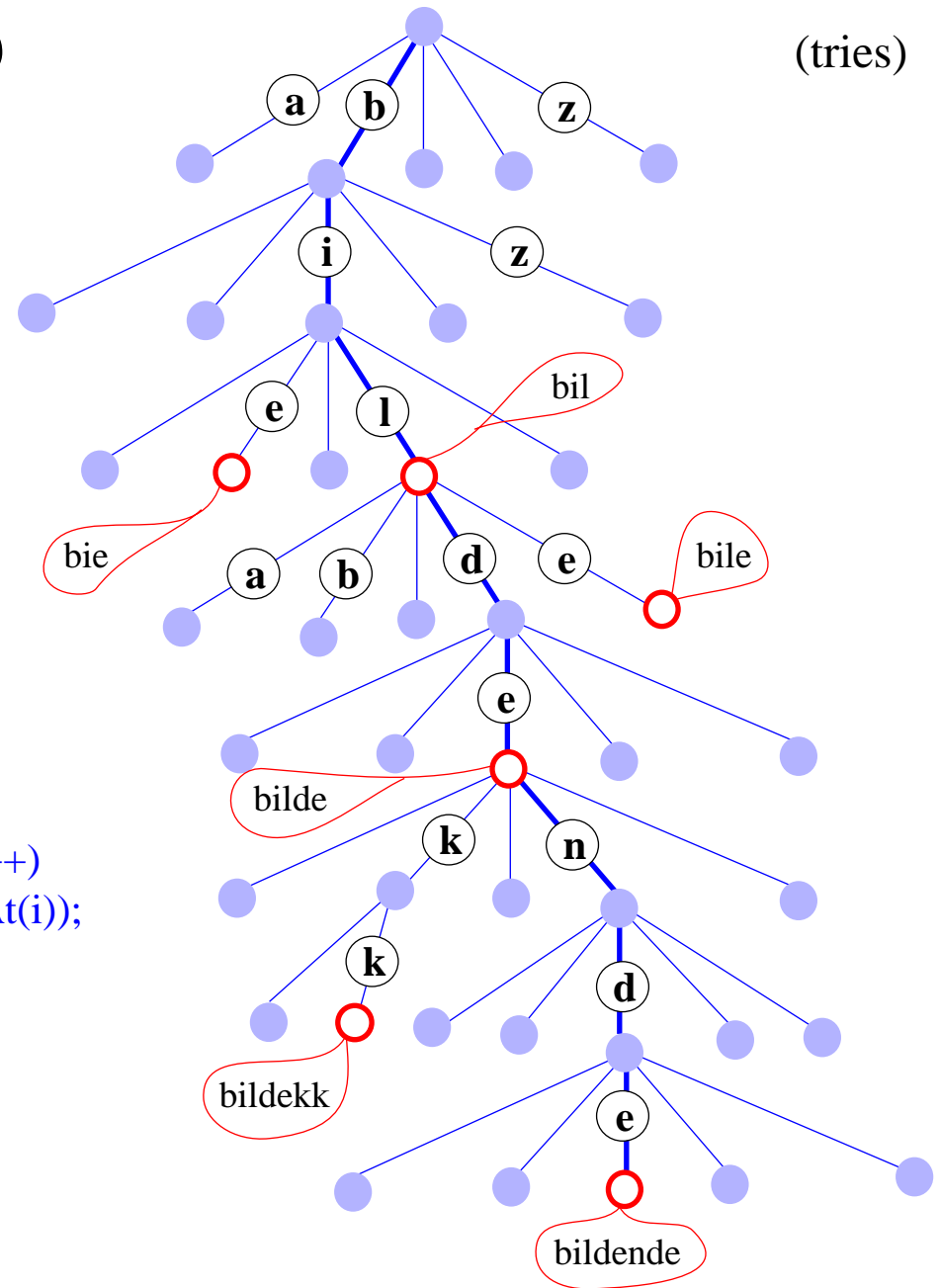
...
 bie
 bil
 bilateral
 bilbelte
 bilde
 bildekk
 bildende
 bile

finn(String X)
 =
 $O(\log n)$

Position **finn**(String X)

```
p = root;
for (int i=0; i<X.length;i++)
    p = p.childNo(X.charAt(i));
if (p == null) break;
if (p == null || !p.isRed())
    finnes-ikke
else return p;
```

= $O(X.length)$



Oppsummering

1. Trær og Binære Trær:

- *definisjoner og terminologi*
- *egenskaper*

2. Tre-algoritmer – traversering:

- *DFS og BFS*
- *DFS :*
 - *pre- og postorder,*
 - *inorder for BinaryTree*

3. Tree og BinaryTree ADT

4. Implementasjon av trær:

- *LenketStruktur*
- *Sekvens – BinaryTree*
- *kompleksitet*