Trær

I. EKSEMPLER, DEFINISJON

II. BINÆRE TRÆR

III. ADT TRE

IV. BASIS TREALGORITMER (TRAVERSERING)

V. IMPLEMENTASJON AV BINÆRE TRÆR

Lenket Struktur
Sequence (Array)

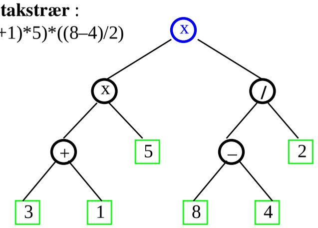
VI. NOEN ANVENDELSER

Ordbok Søking
Streng Komprimering

Kap. 5 (kursorisk: 5.4.4; unntatt 5.5; + DFS/BFS)

Noen eksempler

Syntakstrær :
((3+1)*5)*((8–4)/2)

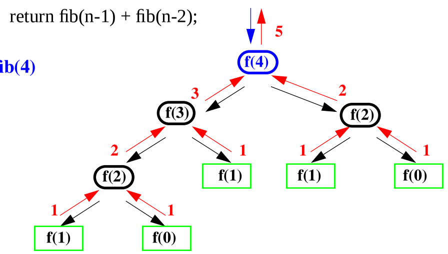


((3 + 1) x 5) x ((8 – 4) / 2)

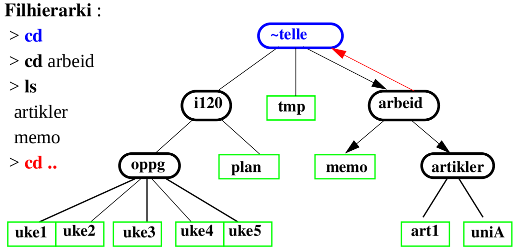
Trær av rekursive kall :

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

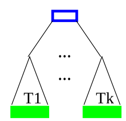
> **fib(4)**



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)

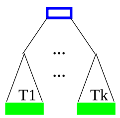
T i figuren har 12 noder (generellt ADT position), **r** er **roten** i **T**

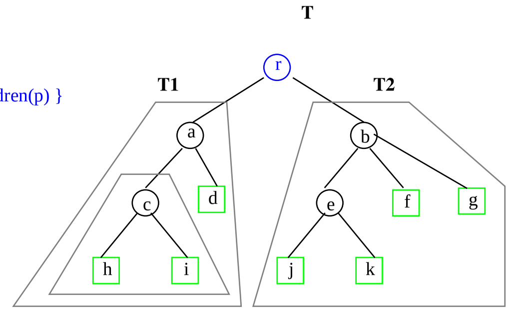
1. **b** er **foreldrenode** (parent) til **e**, **f**, **g** , og **forgjenger** til **e**, **f**, **g**, **j**, **k**
2. **e**, **f**, **g** er **barna** til **b** : umiddelbare etterfølgere (og disse er **søsken**)
3. **d**, **f**, **g**, **h**, **i**, **j**, **k** **eksterne** noder (**løv**) : har 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) = if isRoot(p) return 0
   else return 1+ depth(parent(p))
6. **høyden** av **b** = 2 : avstand til fjerneste løv under b
   height(p) = if isExternal(p) return 0
   else return 1+ max{ height(x) : x 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
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*

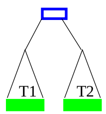




Binære Trær

-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)

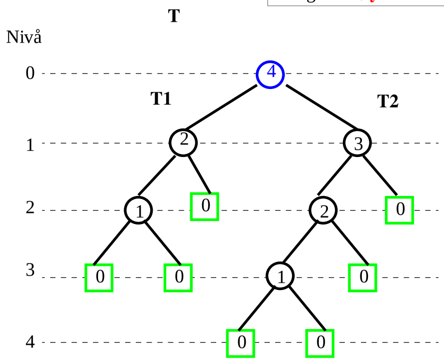
A. # eksterne noder = # interne noder +1

B. # noder på nivå i $\leq 2^i$

C. h+1 $\leq$ (# eksterne noder) $\leq 2^h$
$\log_2$ (# eksterne noder) $\leq$ h

D. h $\leq$ (# interne noder) $\leq 2^h-1$
$\log_2$ (# interne noder) $\leq$ h

E. $2h+1 \leq n \leq 2^{(h+1)}-1$
$\log_2(n+1)-1 \leq h \leq (n-1)/2$



package jdsl.core.api;

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

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

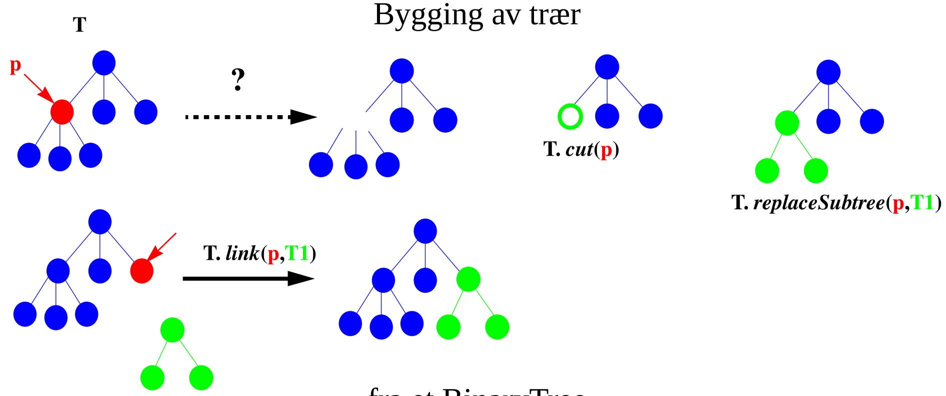
```
/**  Et tre har minst en ekstern node – rot */
public interface InspectableTree
            extends PositionalContainer {
      Position root();
      Position parent(Position v);
/**   Barn er ordnet fra venstre til høyre */
      Enumeration children(Position p);
      boolean isInternal(Position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

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

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

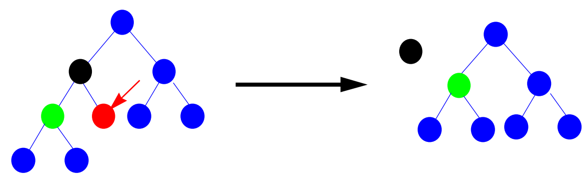
```
public interface Tree extends InspectableTree {
/** Node p erstattes med en ny ekstern node
 *   med null element. */
      Tree cut(Position p);
/** Node p (må være ekstern) erstattes med treet t */
      void link(Positionp p, Tree t);
/** Erstatt subtre i node p med et nytt subtre t */
      Tree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


Bygging av trær

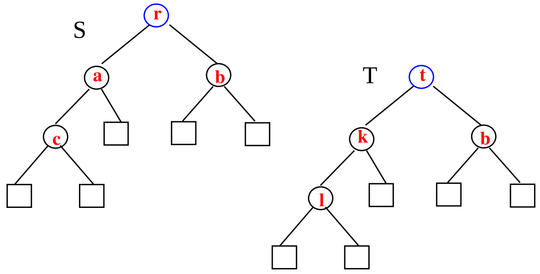


fra et BinaryTree

void *removeAboveExternal*(Position p)



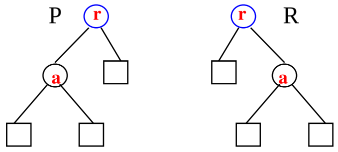
BinaryTree: likhet vs. isomorfisme



boolean **iso** (InspectableBinaryTree P)
{ return **iso**(root, P.root(), P) ; }

boolean **equals** (InspectableBinaryTree P)
{ return **equ**(root, P.root(), P) ; }

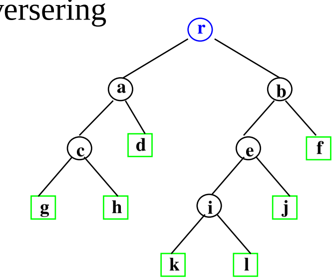
boolean **iso** // equ
(Position r, Position p, InspectableBinaryTree P)
{ **if** (**isExternal**(r) && **P.isExternal**(p))
return true; // r.element().equals(p.element());
else if (**isInternal**(r) && **P.isInternal**(p))
return (// r.element().equals(p.element()) &&
iso(**leftChild**(r), P.**leftChild**(p),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

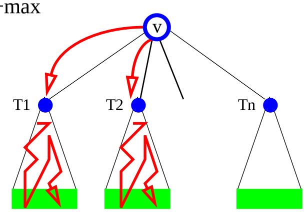


Tre-Algoritmer : traversering

Enumeration positions() – *ok ... men i hvilken rekkefølge ?*

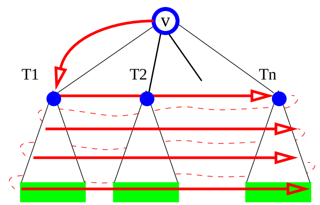
int **height**(Position v)
if (isExternal(v)) return 0
else // 1+max{ height(p): p i children(v) }
max=0
for hver p i children(v)
h=**height**(p); if (h>max) max=h
return 1+max





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

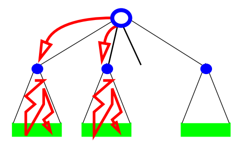
enumererer: **r, a,c,g,h,d, b, e,i,k,l,j, f**

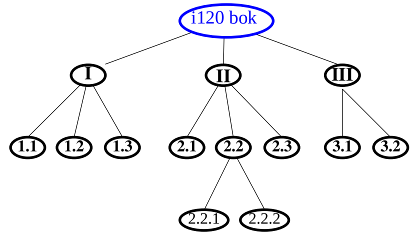


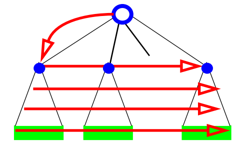
void **BFS**(Tree T, Position v)

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

DFS vs. BFS



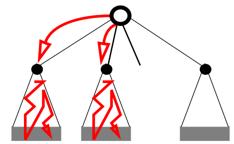


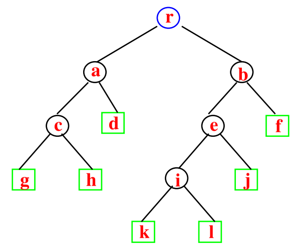


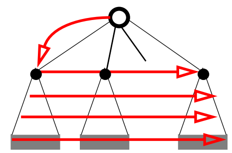
I-120 bok
- **Chap. I Design Principles**
  - **1.1** DS & Alg
  - **1.2** OO-Programming
  - **1.3** JAVA
- **Chap. II Analysis Tools**
  - **2.1** Alg. Analysis
  - **2.2** Running Time
    - **2.2.1** O-notation
    - **2.2.2** Avarage Case
  - **2.3** Worst Case
- **Chap. III Basic DS**
  - **3.1** Stacks
  - **3.2** Queues

I-120 bok
- **Chap. I Design Principles**
- **Chap. II Analysis Tools**
- **Chap. III Basic DS**
  - **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** Avarage Case

DFS og BFS

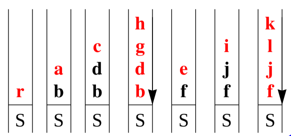






r, a,c, h,g, d, b, e,i,k,l, j, f

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



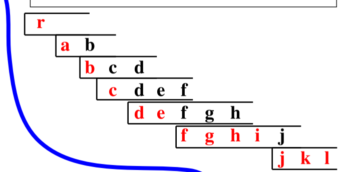
trav(T, new **StackIm**())

"samme" algoritme:

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

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))
 */
```

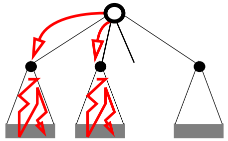


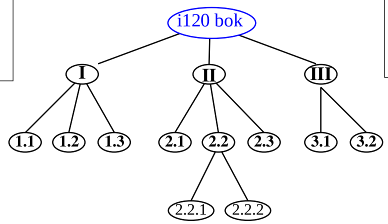
trav(T, new **QueueIm**())

PreOrder DFS PostOrder

gjør jobben
FØR
rekursive kall

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





gjør jobben
ETTER
rekursive kall

```
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
    - **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**

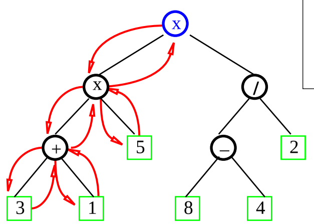
I-120 bok

Binære Trær : **Euler-Tour & InOrder**

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

```
DFS(BinTree B, Position v)
  ??? – pre (left)
  if (isInternal(v))
     DFS(B,B.leftChild(v))
  ??? – in-order (below)
  if (isInternal(v))
     DFS(B,B.rightChild(v))
  ??? – post (right)
```

```
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) = 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))
```

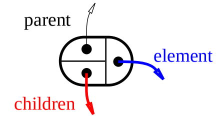
x x + 3 1 5 / – 8 4 2

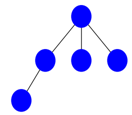
x [x(+ (3,1), 5), / (–(8,4),2)]

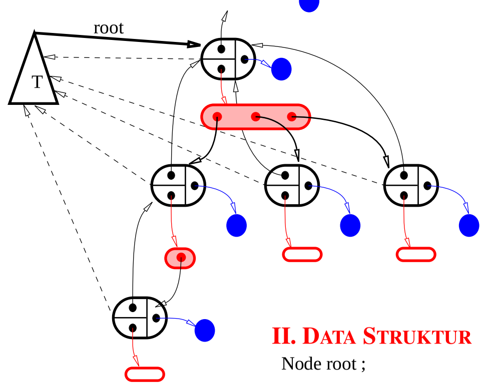
```
int val(BinTree B, Position v)
  if (B.isExternal(v))
     return integer(v)
  else
     L= val(B,B.leftChild(v))
     R= val(B,B.rightChild(v))
     return operasjon(v)(L,R))
```

Implementasjon av Tree – *med LenketStruktur*

I. Data Representasjon







II. Data Struktur

Node root ;

III. Data Invariant : *for alle Position* p, v *i* T:

- p.container() == T • T.root() != null
- isRoot(p) ↔ (p.parent()==null)
- T.isEmpty() ↔ T.root().elem() == null
- isExternal(p) == p.children().isEmpty()
- isInternal(p) == !p.children().isEmpty()
- v.parent()==p ↔ v er bland p.children()
- p.children() != null

```
public class Node implements Position
{ private Object elem; private Node parent;
  private Container  cont;  private Sequence children;
  public Node(Object e, Node p, Container c) {
    setElement(e); setParent(p);  setContainer(c);
    children= new SequenceLL(); }
  public Object element() { return elem; }
  public void setElement(Object e) { elem= e; }
  public Container container() { return cont;}
  public void setContainer(Container c) { cont= c;
    for alle x i children() : x.setContainer(c); }
  public Node parent() { return parent; }
  public void setParent(Node p) { parent= p; }
  public Container children() { return children; }
  public void setChildren(Container c) { children= c; }
```


LenketStruktur implementasjon av Tree ADT

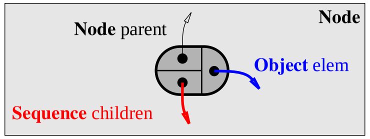
```
public class TreeLS implements Tree {
  private Node root; private int size;

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

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

  public Position root() { return root; }

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

  public Enumeration children(Position v) throws.. {
    gjør om ok(v).children() til Enumeration }
  //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 replace(Position v, Object e) throws..{
    Object tmp = ok(v).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();
  Enumeration ch= children(n);
  while (ch.hasMoreElements()) {
     ((Node)ch.nextElement()).setParent(retRoot); }
  retRoot.setChildren(n.children());
  retRoot.setContainer(ret);
  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.container() != this ||
      !(p instanceof Node) ) throw new InvalidPosExc();
  return (Node)p; }

public void link(Position v, Tree T) throws ... {
  // lettest når T også er TreeLS  . . .
```

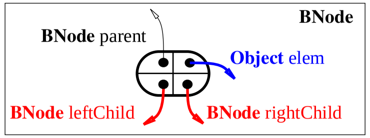

Implementasjon av BinaryTree ADT

I. Utvid klassen TreeLS 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

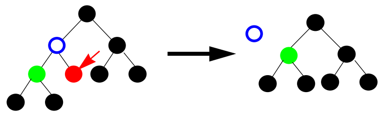



```
public class BNode implements Position
{ private Object elem; private Container cont;
  private BNode parent, left, right;
  public BNode(Object e, BNode p, Container c) {
    setElement(e); setParent(p); setContainer(c); }
  public Object element() { return elem; }
  . . .
  protected void setContainer(Container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


LenketStruktur implementasjon av BinaryTree ADT

```
public class BTreeLS implements BinTree {
  private Position root;
  public BTreeLS(Object e) {
    root = new BNode(e, null, this);  }
  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, this) );
    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.container() != this ||
       !(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, this) );
    n.setRight( newBNode(null, n, this) ); }    }
public Object 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());
  return o;
}
```



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

Sequence implementasjon av BinaryTree ADT

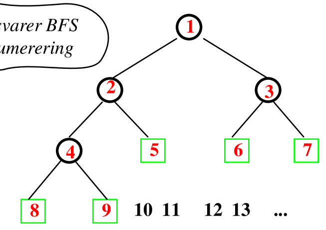
I. Data Representasjon

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

sn *bestemmer nodens stilling i sekvensen*

tilsvarer BFS enumerering

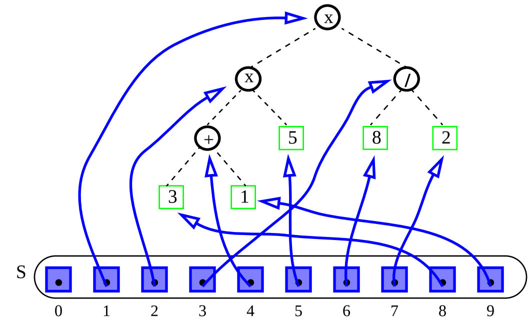


III. DataInvariant :

- 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*



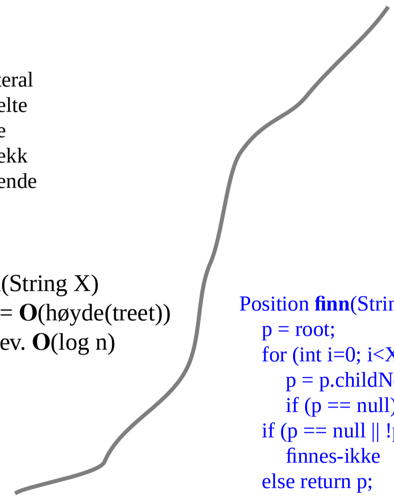
Position **leftChild**(Position p) {
return S . atRank(S.rankOf(p)*2);}
Position **root**() { return S . atRank(1); }

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

Ordbok-søking (totalt ordnede barn)

(tries)

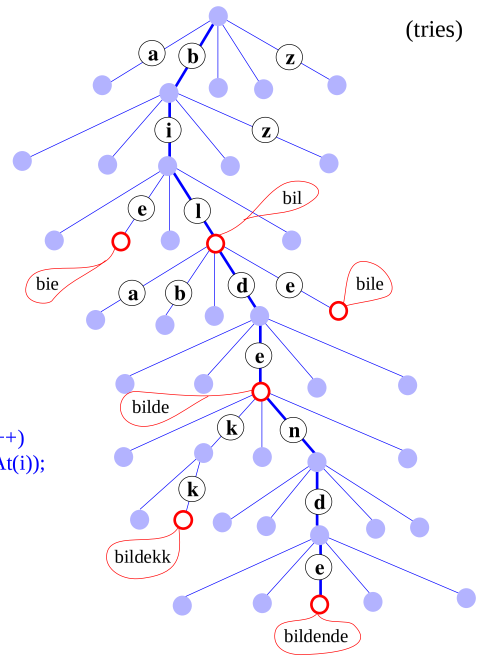
...
bie
bil
bilateral
bilbelte
bilde
bildekk
bildende
bile
....



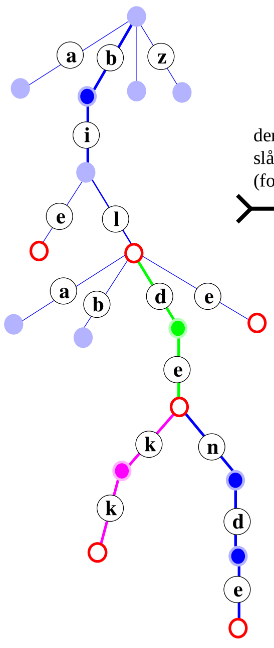
finn(String X)
= **O**(høyde(treet))
ev. **O**(log n)

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;

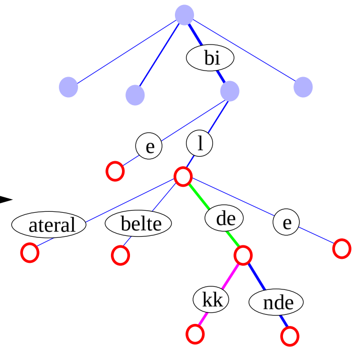
= **O**(X.length)



Komprimering



dersom node **n** har bare 1 barn **b**,
slå **n** og **b** sammen til 1 ny node
(fortsett nedover treet)




```
Position finn(String X, Position p)     // finn(S,root)
    if (X==e && p.isRed()) return p
    else if (X==e) return null
    else if (p.isExternal()) return null
    else  Position r= null
        for (hver kant e = (p,w))
            t = e.string(); l = t.length;
            if (X.startsWith(t))
                r= finn(X[l+1...X.length], w); break ;
        return r;
```

= **O**(X.length)

Streng komprimering

iterer gjennom hele teksten S:

S = S[0...*i*-*1*]+ S[*i*...length]
C + ***X***

p = lengste prefiks av ***X*** som er komprimert i ***C*** som ***cp***

t = tegnet i ***X*** rett etter ***p***

· utvid ***cp*** med **+** ***t***

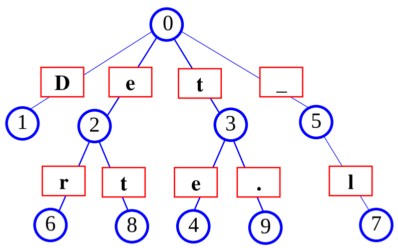
· fortsett med ***i*** rett etter ***t*** i ***X***

D e t t e er lett .
1 2 3 4 5 6 7 8 9

0D	0e	0t	3e	0_	2r	5l	2t	3.
1	2	3	4	5	6	7	8	9

D e t t e er lett .

D e t t e er lett .
komprimert: *C* | *X* : resten
i *p* *t*



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*