Trær

I. Eksempler, Definisjon

II. Binære Trær

III. Tre ADT

IV. Basis TreAlgoritmer (traversering)

V. Implementasjon av Binære Trær

Linket Struktur

Sequence (Array)

VI. Anvendelse (eksempel)

Ordbok Søking

Streng Komprimering

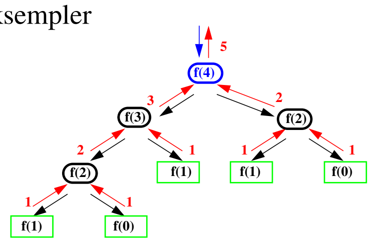
Kap. 5 (kursorisk: 5.4.4; unntatt 5.5; + DFS/BFS)

Eksempler

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

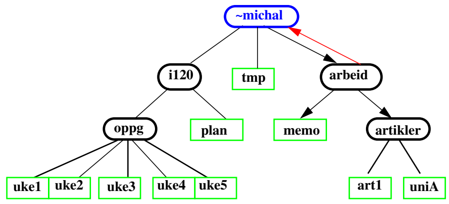
> cd arbeid

> ls

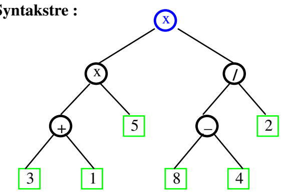
artikler

memo

> cd ..



Syntakstre :



((3 + 1) x 5) x ((8 – 4) / 2)

en **Tre-struktur** T er enten :

en *Position*

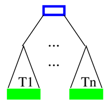
eller

en *Position r*

og

n disjunkte trær

(r ikke i Ti)



Definisjon og Terminologi

1. **r** er **roten** (posisjon) i **T**
2. **b** er faren (**parent**) til **e**, **f**, **g** : umiddelbar forgjenger
   og **forgjenger** til **e**, **f**, **g**, samt **j**, **k**
3. **e**, **f**, **g** er **barna** til **b** : umiddelbare etterfølgere (og er **søsken**)
4. **d**, **f**, **g**, **h**, **i**, **j**, **k** **eksterne** posisjoner (**blader**) : de uten etterfølgere
5. **r**, **a**, **b**, **c**, **e** er **interne** posisjoner : ikke eksterne
6. **dybden** av **e** = 2 : lengden av stien fra roten (**nivå**)
   depth(p) = if *isRoot*(p) 0
   else 1+ *depth*(*parent*(p))
7. **høyden** av **b** = 2 : avstand til fjerneste bald under b
   height(p) = if *isExternal*(p) 0
   else 1+ max{ *height*(x) : x er bland *children*(p) }
   høyden av **T** = høyden av r(oten til T)
8. **graden** av **b** = 3 : antall barn
9. **T1**, **T2** er **subtrær** av **T**

Noen egenskaper

10. (# kanter) = (# posisjoner) – 1
11. for hver posisjon finnes det en *entydig sti* til roten

en **Tre-struktur** T er enten :

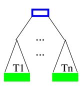
en *Position* r

eller

en *Position* r og

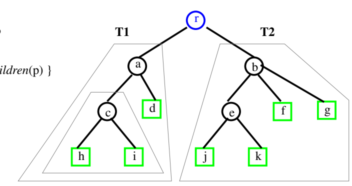
n disjunkte subtrær

(r ikke i Ti)



blad=min < subtre < tre

T



Binære Trær

graden til enhver *Pos.* er **2** el. **0**

et **Binær-Tre** er enten :

en *Position* r

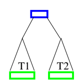
eller

en *Position* r

og

***2** disjunkte subtrær*

(r ikke i Ti)



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

Noen egenskaper (h: høyde; n: #noder)

A. (# eksterne noder) = (# interne noder) +1

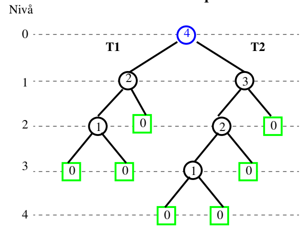
B. (# noder på nivå i) $\leq 2^i$

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

T



Tree ADT

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

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

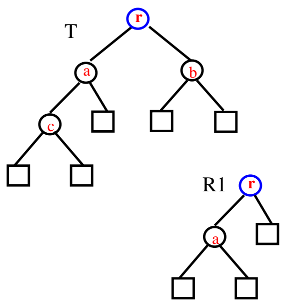
```
public interface PosSequence
{ ... }
```

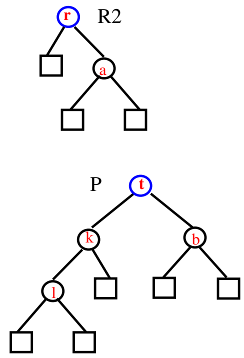
```
public interface Sequence
{ ... }
```

```
public interface Tree extends PosSamling
{   Position root();
    Position parent(Position v);
    Enumeration children(Position p);

    boolean isInternal(Position v);
    boolean isExternal(Position v);
    boolean isRoot(Position v);

    // int depth(Position v);
    // int height(Position v);

    Position addChild(Position v, Object c);
}
```

```
public interface BinaryTree extends PosSam.
{   Position root();
    Position parent(Position v);
    Position leftChild(Position v);
    Position rightChild(Position v);

    boolean isInternal(Position v);
    boolean isExternal(Position v);
    boolean isRoot(Position v);

    Position setLeft(Position v, Object c);
    Position setRight(Position v, Object c);
    Position expandExt(Position v);
    Object removeAboveExt(Position v);
}
```


BinTree-likhet og isomorfisme

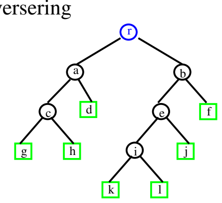




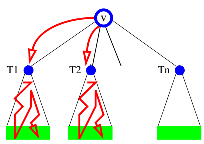

```
boolean iso (BinTree P)
{   return iso(root, P.root(), P) ;  }

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

boolean iso  // equ
             (Position r, Position p, BinTree P)
{   if (isExternal(r) && P.isExternal(p))
        return true
    else if (isInternal(r) && P.isInternal(p) )
        return ( r.elem().equals(p.elem())  &&
               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 ?

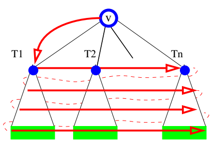



```
int height(Position v)
  if (isExternal(v)) return 0
  else //1+max{height(p): p i children(v)}
      max=0
      for hver p i children(v)
          h=height(p); if (h>max) max=h
      return 1+max
```




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

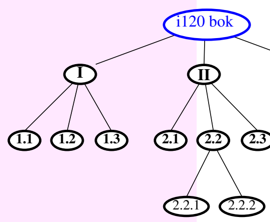
enumererer: **r, a,c,g,h,d, b, e,i,k,l,j, f**

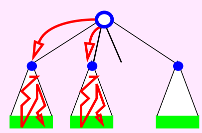


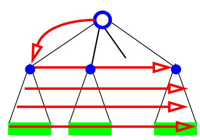

```
void BFS(Tree T, Position v)
```

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

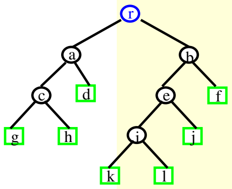
Chap. II Analysis Tools
- **2.1** Alg. Analysis
- **2.2** Running Time
  - **2.2.1** O-notation
  - **2.2.2** Avarage Case
- **2.3** Worst Case

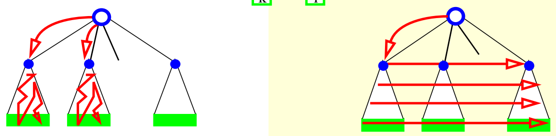
Chap. III Basic DS
- **3.1** Stacks
- **3.2** Queues

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

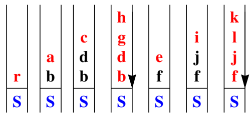
DFS BFS





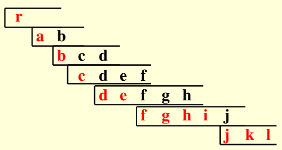
r, a,c,h,g,d, b, e,i,k,l,j, f

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



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

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



"Samme" algoritme

```
public interface FiLi extends Samling {
    void settInn(Object e);
    Position hent();
}
```

```
/*Trav(Tree T, Position v)
 *  FLifo S = new FLifo()
 *  S.add(v)
 *  while (!S.isEmpty())
 *     p= S.remTop()
 *     S.add(T.children(p))
 */
```

```
public interface Stack
    extends FiLi
{ ... }
```

```
public interface Queue
    extends FiLi
{ ... }
```

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

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

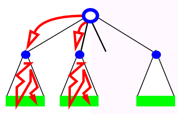
Algoritme-mønster

```
/*  Trav(Tree T, Position v, FiLi S)
 *    // empty S !
 *    S.settInn(v)
 *    while (!S.isEmpty())
 *       p= S.hent()
 *       S.settInn(T.children(p))
 */
```

Trav(T, v, new Stab())   **Trav(T, v, new Koe())**

PreOrder DFS PostOrder

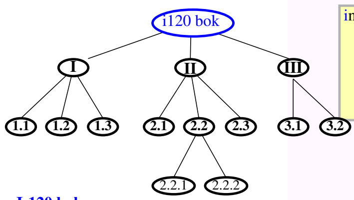
gjør jobben **FØR** rekursjon



gjør jobben **ETTER** rekursjon

```
... 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())
```




```
int du(Tree T, Position v)
  int v_du = 0 // lokal variabel
  for hver p i T.children(v)
     v_du= v_du + du(T,p)
  ( (Fil) element() ).size = v_du
```

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**

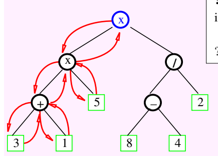
I-120 bok

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

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

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

```
... DFS(BinTree B, Position v)
  if (isExternal(v)) ??? – ekst
  else
     ??? – pre
     DFS(B,B.leftChild(v))
     ??? – inn
     DFS(B,B.rightChild(v))
     ??? – post
```



((3 + 1) x 5) x ((8 – 4) / 2) = 40

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

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

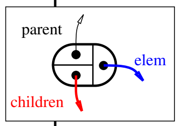
```
int val(BinTree B, Position v)
  if (isExternal(v))
    ((Integer)v.elem()).intValue()
  else
    L= val(B,B.leftChild(v))
    R= val(B,B.rightChild(v))
    ((Oper)v.elem()).op(L,R)
```

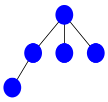
x x + 3 1 5 / – 8 4 2

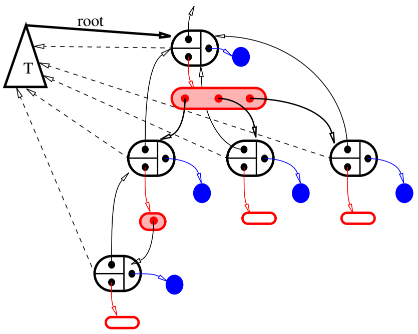
x [x(+ (3,1), 5), / (–(8,4),2)]

Implementasjon av Tree: Linket Struktur

```
public class Node implements Position
{   private Object elem;
    private Samling cont;
    private Node parent;
    private Samling children;
    public Node(Object e, Node p, Struktur c) {...}
    public Object elem() { return elem; }
    public void setElem(Object e) { elem= e; }
    public Samling container() { return cont;}
    public Node parent() { return parent; }
    public Samling children() { return children; }
}
```







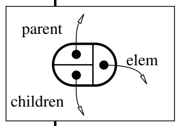
DATA INVARIANT:

for alle Position p, v i T:

- **p.children() != null**
- **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()**
- **...**

Linket Struktur for Tree

```
public class Node implements Position
{ private Object elem;
  private Samling cont;
  private Node parent;
  private Samling children;
  public Node(Object e, Node p, Samling c) {
     setElem(e); setParent(p); setCont(c); }
  public Object element() { return elem; }
  public void setElem(Object e) { elem = e; }
  public Samling container() { return cont;}
  protected void setCont(Samling c) { cont = c; }
  public Node parent() { return parent; }
  protected void setParent(Node p) { parent = p; }
  public Samling children() { return children; }
```




```
public class LL-Tree implements Tree {
   private Position root;
   private int size;
   public LL-Tree(Object e) {
     root = new Node(e, null, this);
     size = 1;  }
   public int size() { return size; }
   ...
   public Object replace(Position p, Object e) {
     Object tmp = p.elem();
     p.setElem(e);
     return tmp;
   }
   ...
   public Position root() { return root; }
   public Position parent(Position v) {
     return ((Node) v).parent(); }
   public Enumeration children(Position v) {
     return ((Node)v).children().elements();}
   public boolean isInternal(Position v) {
     return !((Node)v).children().isEmpty());}
   public boolean isExternal(Position v) {
     return ((Node)v).children().isEmpty());}
   public boolean isRoot(Position v) {
     return (v == root); }
```

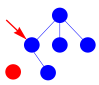
```
public interface Samling
{ int size();
  boolean isEmpty();
  Enumeration elements();
}
```

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

```
public interface Tree
{ ...}
```

insett nye noder i Tree ?

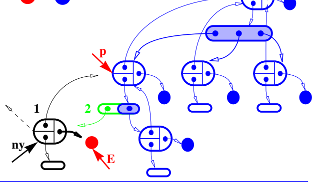
Innsetting



```
class Node implements Position { ...
  private Sequence children; // of Node's
  public Sequence children() { return children; }
  public void addChild(Node n) {
    children.insertLast(n);}
  public void setParent(Position p) {
    parent=p;
    if (p!=null) p.addChild(this);}
  public Node(Object e,Node p,Samling c)
  { setElem(e); setParent(p); setCont(c);
    children= new Seq(); }
  ... }
```

```
public class Seq
  implements Sequence
{ ...
  Enumeration elements()
  {...= new Enum()...}
}
```

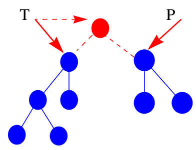
```
public class Enum
  implements Enumeration
{...}
```



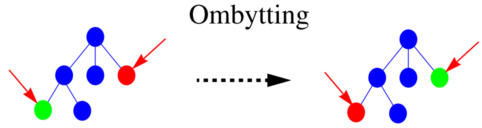

```
class LL-Tree ...
 public Position addChild(Position p, Object E)
 {   Node ny = new Node(E, (Node) p, this);          // 1
   ((Node) p).addChild(ny);                          // 2
   return ny; }                        // pass på repetisjoner !!!
 public Enumeration children(Position p) { // of Node
   return ((Node) p).children().elements(); }
 public Enumeration positions() { // of Node's
   /* =new Enum()                                        O(n)
     traverser (DFS?): root() + children(root()) +
      children(..) + ...  */ }
```


Sammenslåing av trær

```
// kalles for LL-Tree T
void samslaa(LL-Tree P, Object e)
{     Node tmp = (Node)root;
      root= new Node(e,null, this);
      addChild(tmp);
      addChild((Node)P.root());
}
```

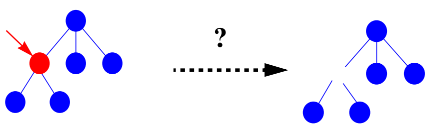


Ombytting



```
void swap(Position p, Position q) {
      Object tmp = p.elem();
      p.setElem(q.elem());
      q.setElem(tmp);
}
```

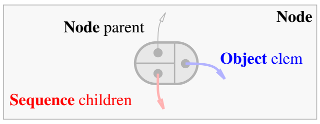
Fjerning ...



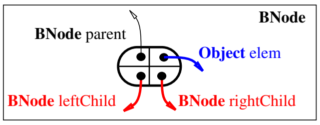
Implementasjon av BinTree (FDT)

I. Utvid klassen LL-Tree slik at:

- Sequence children() alltid har 0 eller 2 elementer – pass på addChild(n)
- implementer Position leftChild(), Position rightChild() – skill i children()



II. Bruk en annen BNode



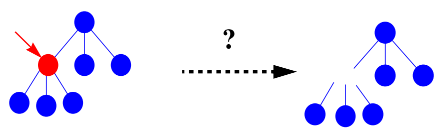

```
public class BNode implements Position
{ private Object elem;
  private Samling cont;
  private BNode parent, left, right;
  // private Samling children;
  public BNode(Object e, BNode p, Samling c) {
      setElem(e); setParent(p); setCont(c); }
  public Object element() { return elem; }
  public void setElem(Object e) { elem = e; }
  public Samling container() { return cont;}
  protected void setCont(Samling c) { cont = c; }
  public BNode parent() { return parent; }
  protected void setParent(BNode p) { parent = p; }
  // public Samling children() { return children; }
  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); }
```


BinTree med BNode-Liste

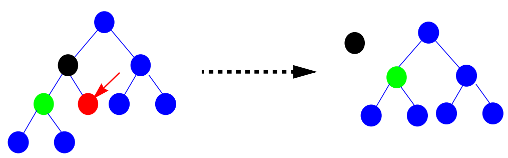
```
public class BTreeLL implements BinTree {
  private Position root;
  private int size;
  public BTreeLL(Object e) {
    root = new BNode(e, null, this); size = 1;  }
  public int size() { return size; }
  public Object replace(Position p, Object e) {
    Object tmp = p.elem(); p.setElem(e); return tmp;  }
  ...
  public Position root() { return root; }
  public Position parent(Position v) {
    return ((BNode) v).parent(); }
  public Position leftChild(Position v) {
    return ((BNode) v).leftChild().element(); }
  public Position setLeft(Position p, Object c) {
    if (isExternal(p)) expandExt(p);
    ((BNode) p).setLeft(new BNode(c, (BNode)p, this);
    return ((BNode) p).leftChild(); }
  public Position rightChild(Position v) { ... }
  public Position setRight(Position p, Object c) { ... }
  public boolean isInternal(Position v) {
    return ((BNode v).leftChild()!=null || (BNode v).rightChild()!=null); }
  public boolean isExternal(Position v) { return ! isInternal(v) ; }
  public boolean isRoot(Position v) { return (v == root); }
  public void expandExt(Position v) {
    if (isExternal(v)) {
      ((BNode)v).setLeft(new BNode(null,(BNode)v,this);
      ((BNode)v).setRight(new BNode(null,(BNode)v,this);
      size= size+1; }}
  public Object removeAboveExt(Position p) { ... }
}
```

alle operasjoner (unntatt positions(), elements()) er **O(1)**

Fjerning ...



fra et BinTree
Object removeAboveExt(Position p)



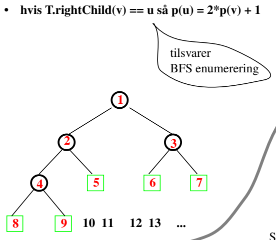
```
{    if (isExternal(p)) {
          BNode rem = (BNode)parent(p) ; Object o = rem.element() ;
          if (rem.leftChild()==p) BNode sib= rem.rightChild() ;
          else BNode sib= rem.leftChild() ;
          rem.setElem(sib.element());
          rem.setLeft(sib.leftChild());
          rem.setRight(sib.rightChild());
          return o ;
     } else throw new NotExternal();
}
```


III. BinTree med Sequence

Ideen :

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

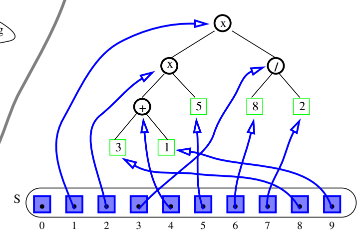
- **hvis T.isRoot(v) så p(v) = 1**
- **hvis T.leftChild(v) == u så p(u) = 2*p(v)**
- **hvis T.rightChild(v) == u så p(u) = 2*p(v) + 1**




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

Invariant :

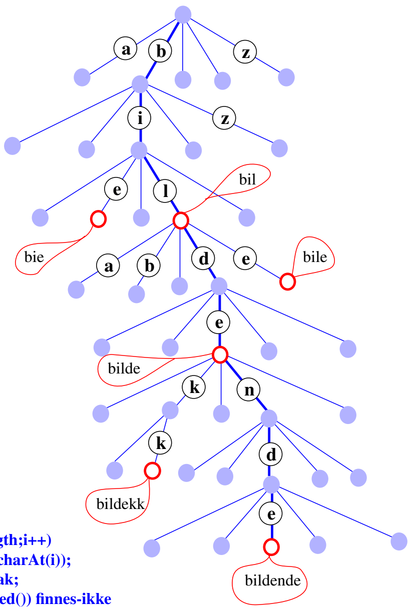
- **S(1) == root**
- **S(2*v) == T.leftChild(S(v))**
- **S(2*v+1) == T.rightChild(S(v))**



Alle BinTree-operasjoner (unntatt (positions(), elements()) kan implementeres med **O(1)** – relativt til implementasjon av Sequence-operasjoner

Ordboksøking (FDT) (tries)

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

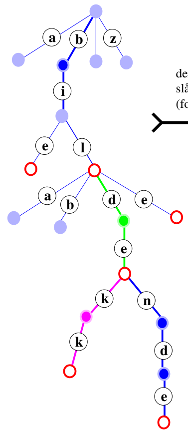


finn(String X) = O(n)
ev. O(log n)

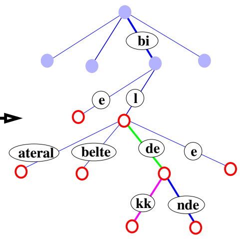
```
finn(String X)
   p = root;
   for (int i=0; i<X.length;i++)
      p = p.childNo(X.charAt(i));
      if (p == null) break;
   if (p == null || !p.isRed()) finnes-ikke
```

= O(X.length)

Komprimering



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




```
Node finn(String X, Node p)    // finn(S,root)
   if (X==ε && p.isRed()) return p
   else if (X==ε) return null
   else if (p.isExternal()) return null
   else  Node 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

D e t t e e r l e t t .

iterer gjennom hele teksten S:

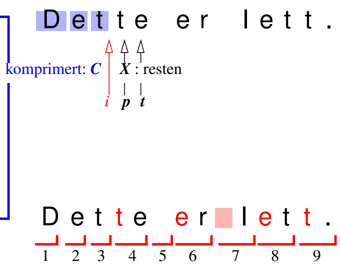
S = S[0...*i*-*1*]+ S[*i*...length]
C + *X*

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*



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