Föreläsning 11
Innehåll

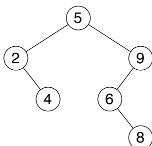
- Binära sökträd
  - algoritmer för sökning, insättning och borttagning
  - implementering
  - effektivitet
- balanserade binära sökträd, AVL-träd
- Jämföra objekt
  - interfacet `Comparable`
  - Interfacet `Comparator`

Datastrukturer som passar för sökning

- Antag att vi i ett program ska hantera ett stort antal element av något slag. Det ska gå snabbt att
  - **sätta** in ett element
  - **ta bort** ett element
  - **söka** efter ett visst element
- Vilken/vilka datastrukturer passar bra för detta?
  - vektor, enkellänkad lista
    - insättning snabb – $O(1)$
    - sökning (linjärsökning) och borttagning långsamma – $O(n)$
  - sorterad vektor
    - sökning (binärsökning) snabb – $O(logn)$
    - insättning och borttagning långsamma – $O(n)$
- Det finns datastrukturer som passar bättre för detta:
  - **Binära sökträd**
  - **Hashtabeller**

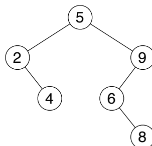
Binära sökträd
Definition

- Ett binärt sökträd är ett binärt träd där följande gäller för varje nod n:
  - Alla värden som finns i noder i *vänster subträd* till n är *mindre* än värdet som finns i n.
  - Alla värden som finns i noder i *höger subträd* till n är *större* än värdet som finns i n.
- Dubbletter tillåts alltså *inte*.



Inordertraversering av binära sökträd

- Genomgång av trädet i **inorder** besöker noderna i växande ordning.
- Exempel: En inordertraversering av trädet i figuren ger noderna i ordningen 2, 4, 5, 6, 8, 9



Binära sökträd – sökning

Då man söker efter ett element i ett binärt sökträd kan man utnyttja ordningen i trädet:

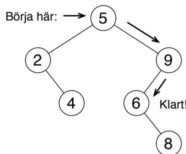
- Börja i roten, jämför med sökt element x, om likhet är vi klara.
- Om x är mindre än rotens element: gå till vänster barn annars gå till höger barn.
- Fortsätt på samma sätt tills vi hittar det sökta, eller tills den nod som står i tur att undersökas är null (misslyckad sökning).

Man följer alltså en gren i trädet:

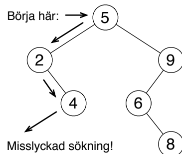
- Grenen börjar i roten.
- Man fortsätter tills man hittar det sökta eller till grenen tar slut (misslyckad sökning).

Binära sökträd - sökning
Exempel

Sök efter 6 i trädet:

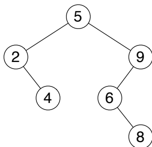


Sök efter 3 i trädet:



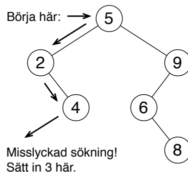
Diskutera

- Hur går insättning i ett binärt sökträd till?
  - Var i trädet ska 3 sättas in?
  - Var i trädet ska 7 sättas in?

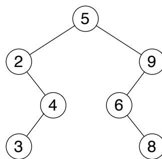


Binära sökträd - insättning
Exempel

Sätt in 3 i trädet:

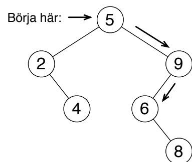


Efter insättning:



Binära sökträd - insättning
Exempel

Sätt in 6 i trädet:



Dubblett hittas.
Insättningen genomförs inte.

Binära sökträd – insättning
Sammanfattning

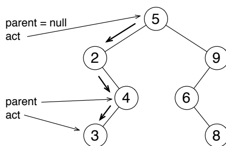
- Vid insättning av nytt element ska ordningen i trädet bevaras.
- Dubbletter får inte förekomma.
- Insättning kan tolkas som "misslyckad sökning":
  - Vi söker på en gren i trädet.
  - Om vi misslyckas med hitta det element som ska sättas in utförs insättningen som ett löv på den plats i trädet där misslyckandet konstaterats.

Binära sökträd – borttagning

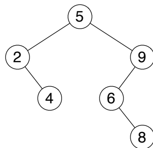
- För att kunna ta bort ett element ur trädet måste vi söka upp det.
- När vi tar bort det måste vi koppla ihop föräldern med något av barnen.
  - Vid sökningen behöver man därför hålla reda på en referens till föräldern.
- Sammankopplingen sköts på olika sätt beroende på hur många barn som finns till den nod som ska bort:
  - Enklaste fallen är noll eller ett barn.
  - Fallet två barn är lite krångligare.

Binära sökträd - borttagning
Exempel på enkelt fall - noll barn

Tag bort 3 ur trädet.
Börja med att söka efter noden (och föräldern).

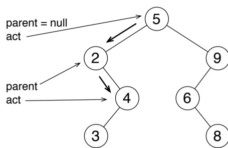


Noden, `act`, som ska bort är ett löv.
Sätt den referens i `parent` som refererar till `act` till null.

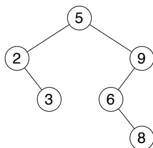


Binära sökträd - borttagning
Exempel på enkelt fall - ett barn

Tag bort 4 ur trädet.
Börja med att söka efter noden (och föräldern).

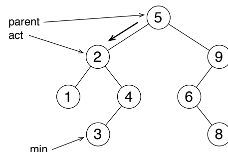


Noden, act, som ska bort har ett barn.
Sätt den referens i parent som refererar till act till att referera till act:s barn.



Binära sökträd - borttagning
Exempel - två barn

Tag bort 2 ur trädet.

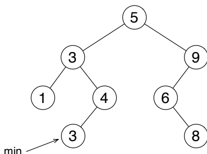


Noden, act, som ska bort har två barn:

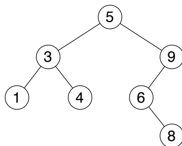
- Sök upp minsta noden (min) i act:s högra subträd
- Flytta data från denna till act
- Tag bort min

Binära sökträd - borttagning
Exempel - två barn, forts

Efter flyttning av data från min till act:



Efter borttagning av min:



Observera att borttagning av min är ett enklare fall eftersom denna nod kan ha högst ett barn (höger).

Binära sökträd – borttagning
Specialfall

- I några av de fall som beskrivits finns det ett alternativ som måste hanteras speciellt: förälder till den nod som skall tas bort saknas, d.v.s. roten tas bort.
  - Fall 1: roten ska då bli null.
  - Fall 2: roten ska referera till act:s barn.

Implementering av binära sökträd
Klassen BinarySearchTree

Exempel på hur en klass som representerar ett binärt sökträd kan se ut:

```
public class BinarySearchTree<E> {
   private Node<E> root;

   public BinarySearchTree() {
      root = null;
   }

   public boolean add(E e) {...}
   public E find(E x) {...}
   public boolean remove(E x){...}
   ...

   //  nästlad klass Node<E> med attributen data, left och right
   //  som representerar en nod ...
}
```


Sökning i binärt sökträd – pseudokod

```
E find(node, x) {
    om node == null)
        return null
    annars
        om x är lika med node.data
            return node.data;
        annars om x är mindre än node.data
            return find(node.left, x)  // sök i vänster subträd
        annars
            return find(node.right, x) // sök i höger subträd
```

Anropas `find(root, x)`

Diskutera

Hur ska jämförelserna mellan två element i trädet gå till?

```
om x är lika med node.data
om x är mindre än node.data
```


Metoden find

```
public E find(E x) {
      return find(root, x);
}

private E find(Node<E> n, E x) {
   if (n == null) {
      return null;
   }
   int compResult = ((Comparable<E>) x).compareTo(n.data);
   if (compResult == 0) {
      return n.data;
   } else if (compResult < 0) {
      return find(n.left, x);
   } else {
      return find(n.right, x);
   }
}
```

Interfacet Comparable
Repetition

- Metoden compareTo i interfacet Comparable används inuti add, find och remove för att jämföra två element.

```
public interface Comparable<T> {
   /**
    * Compares this object with the specified object for order.
    * Returns a negative integer, zero, or a positive integer as
    * this object is less than, equal to, or greater than the
    * specified object.
    */
   public int compareTo(T x);
}
```


Kommentarer

- Klassen som ersätter E måste implementera Comparable<E>.
  - Annars genereras ClassCastException när find exekveras.
- För att anropet av compareTo ska fungera måste vi typkonvertera x till Comparable<E>:

```
... ((Comparable<E>) x).compareTo(node.data) ...
```

- En annan lösning är att via en konstruktor förse klassen BinarySearchTree med ett komparator-objekt comp och använda metoden compare för att jämföra:

```
... comp.compare(x, node.data) ...
```


Exempel på användning av klassen BinarySearchTree

```
// Skapa ett träd  sorterat efter idnummer
BinarySearchTree<Person> bst = new BinarySearchTree<Person>();
bst.add(new Person("Fili", 1));
bst.add(new Person("Balin", 2));
...

// sök efter personen med idnummer 2
Person p = bst.find(new Person(null , 2);
```

- Klassen Person måste implementera Comparable<Person>.
- Observera att man måste skapa ett Person-objekt för att kunna söka efter en person i trädet.
  - Detta objekt måste få korrekta värden på de attribut som används för jämförelser i compareTo.

Implementering av interfacet Comparable
Klassen Person

```
public class Person implements Comparable<Person> {
   private String name;
   private int id;
   ...

   public int compareTo(Person other) {
      return Integer.compare(id, other.id);
   }

}
```

- Comparable är ett generiskt interface. I exemplet är Person typargument.
- Observera att parametern i compareTo därför har typen Person.

Diskutera

Nu kan vi söka efter personen med ett visst idnummer:

```
// Skapa ett träd  sorterat efter idnummer
BinarySearchTree<Person> bst = new BinarySearchTree<Person>();
bst.add(new Person("Fili", 1));
bst.add(new Person("Balin", 2));
...

// sök efter personen med idnummer 2
Person p = bst.find(new Person(null , 2);
```

Hur ska vi göra om vi också vill kunna söka efter personen med ett visst namn?

```
Person p = bst.find(new Person("Balin" , -1);
```


Interfacet Comparator

- Vi ska se till att det finns ett alternativ till `compareTo` för att jämföra element inuti trädklassen.
- Interfacet `Comparator` ger oss möjlighet att jämföra objekt av en klass på flera olika sätt.

```
public interface Comparator<T> {
   /**
    * Compares its two arguments for order.
    * Returns a negative integer, zero, or a positive
    * integer as the first argument is less than,
    * equal to, or greater than the second.
    */
   int compare(T e1, T e2);
}
```


Konstruktor med parameter av typen Comparator

Klassen `BinarySearchTree`

```
public class BinarySearchTree<E> {
    private Node<E> root;
    private Comparator<E> comp;

    /** Skapar ett tomt binärt sökträd. Elementen förutsätts
        vara av en klass som implementerar Comparable<E>. */
    public BinarySearchTree() {
        root = null;
        comp = null;
    }

    /** Skapar ett tomt binärt sökträd.
        Elementen jämförs med komparatorn comp. */
    public BinarySearchTree(Comparator<E> comp) {
        root = null;
        this.comp = comp;
    }
```


Metod för att välja mellan compareTo och compare

privat metod i klassen `BinarySearchTree`

```
...

private int compareElements(E e1, E e2) {
    if (comp == null) {
        return ((Comparable<E>) e1).compareTo(e2);
    } else {
        return comp.compare(e1, e2);
    }
}

...
```

Metoden find

Användning av metoden compareElements

```
public E find (E x) {
   return find(root, x);
}

private E find(Node<E> n, E x) {
   if (n == null) {
      return null;
   }
   int compResult = compareElements(x, n.data);  // nytt
   if (compResult == 0) {
      return n.data;
   } else if (compResult < 0) {
      return find(n.left, x);
   } else {
      return find(n.right, x);
   }
}
```


Två konstruktorer

- Nu har klassen två konstruktorer:
  1. `public BinarySearchTree();`
  2. `public BinarySearchTree(Comparator<? super E> c);`
- Används den första konstruktorn, förutsätts elementen implementera `Comparable`
  - annars genereras `ClassCastException`.
  - Inuti trädklassen används metoden `compareTo` för att jämföra två objekt.
- Den andra konstruktorn har en parameter av typen `Comparator`.
  - Vid anrop skickar man med en referens till ett objekt av en klass som implementerar interfacet `Comparator`.
  - Används denna kommer jämförelser att utföras med hjälp av komparatorns metod `compare`.

Skapa komparator-objekt med lambdauttryck

```
// Skapa ett träd sorterat efter namn
BinarySearchTree<Person> bst = new BinarySearchTree<Person>(
             (p1, p2) ->  p1.getName().compareTo(p2.getName()) );
bst.add(new Person("Fili", 1));
bst.add(new Person("Balin", 2));
...

// sök efter personen med namnet Balin
Person p = bst.find(new Person(Balin , -1);
```

- Interfacet `Comparator<E>` är ett funktionellt interface, d.v.s. den har bara en abstrakt metod.
- I sådana fall kan man använda lambdauttryck istället för att
  - skriva en klass som implementerar interfacet
  - skapa ett objekt av denna klass och skicka med en referens till ett sådant objekt som argument.

Skapa komparator-objekt – längre version

- Skriv en komparatorklass som implementerar interfacet `Comparator`:

```
public class NameComparator implements Comparator<Person> {
   public int compare(Person p1, Person p2) {
      return p1.getName().compareTo(p2.getName());
   }
}
```

- Skapa ett objekt av komparatorklassen och skicka med som argument till konstruktorn i trädklassen:

```
BinarySearchTree<Person> bst =
         new BinarySearchTree<Person>(new NameComparator());
```

Vad betyder `? super E`

Överkurs

- "`? super E`" kan utläsas "okänd superklass till `E` (inklusive `E`)"
- Typen `Comparator<? super E>` bör användas i trädklassen istället för `Comparator<E>`.
- Förklaring: Antag att vi har följande klasser:

```
class Person { ... }
class Student extends Person { ... }
class NameComparator implements Comparator<Person> { ... }
```

  - Om `Comparator<E>` används inuti trädklassen kan vi inte skriva:

```
BinarySearchTree<Student> bst =
    new BinarySearchTree<Student>(new NameComparator());
```

  Det finns ingen klass som implementerar `Comparator<Student>`.

  - Istället lättar vi på på kravet och kräver att komparatorklassen istället ska implementera `Comparator<? super E>`. Då kan vi skapa "studentträdet" eftersom `Person` är superklass till `Student`.

Diskutera

- En traversering genom alla noderna i ett träd med n noder har tidskomplexiteten $O(n)$.
  - Varje nod besöks en gång.
- Vad påverkar effektiviteten för operationerna sökning, insättning och borttagning?
- Vad får dessa metoder för tidskomplexitet?

Binära sökträd – tidskomplexitet

- Operationerna sökning, insättning och borttagning innebär sökning utmed en gren i trädet.
- I varje nod görs ett konstant arbete (väsentligen en jämförelse).
- Den längsta grenen i ett träd har h noder, där h är trädets höjd.
- Värsta fall för samtliga operationer är alltså $O(h)$.
- Vi vill uttrycka tidskomplexiteten som en funktion av antal noder, n.
- Vi måste alltså känna till sambandet mellan trädet höjd och antal noder.

Samband mellan höjd och antal noder i binära träd

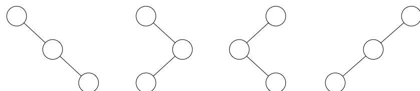
För alla binära träd T med n noder gäller för höjden h

$$h \leq n \qquad (1)$$

och

$$h \geq {}^{2}\log(n+1) \qquad (2)$$

(1) är enkelt att inse. Trädet får största möjliga höjd om man placerar en nod på varje nivå. Höjden blir då n.
Ex. för $n = 3$:



Samband mellan höjd och antal noder i binära träd, forts

(2):

På nivå ett finns en nod (roten).

På nivå två högst 2 noder, på nivå tre högst 4 nodet etc.
Allmänt finns det på nivå i högst 2^{i-1} noder.

Den högsta nivån i ett träd med höjd h är h.
$\Longrightarrow$ antal noder, $n \leq 1 + 2 + 4 + ... + 2^{i-1} + ... + 2^{h-1} = \frac{1-2^h}{1-2} = 2^h - 1$.

Vilket ger att $h \geq {}^2\log(n+1)$

Här har vi använt formeln för geometrisk summa:

$1 + a + a^2 + ... + a^k = \frac{1-a^{k+1}}{1-a}$

Binära sökträd – tidskomplexitet

- Samtliga operationer (sökning, insättning, borttagning) innebär sökning utmed en gren i trädet.
  - I varje nod görs ett konstant arbete (väsentligen en jämförelse).
  - Den längsta grenen i ett träd har h noder, där h är trädets höjd.
  - Värsta fall för samtliga operationer är alltså $O(h)$.
- Vi har tidigare visat att för alla binära träd med n noder gäller att
  - ${}^2\log(n+1) \leq \mathit{höjden} \leq n$
- De tre operationerna har tidskomplexitet $O(n)$ i värsta fall och $O({}^2\log n)$ i bästa fall.

Diskutera

Sätt in 1, 2, . . . 7
(i den ordningen)

Sätt in
4, 2, 1, 6, 5, 3, 7:

Sätt in
2, 5, 1, 6, 7, 3, 4:

Vilken insättningsordning ger bästa respektive sämsta formen på trädet?

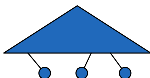
Binära sökträd – tidskomplexitet, forts

- Trädets höjd påverkas av insättningar och borttagningar.
- I värsta fall består trädet av en enda gren och värstafallstiden för sökning, insättning och borttagning blir $O(n)$.
- Vid slumpmässiga insättningar och borttagningar blir tidskomplexiteten i medelfall $O({}^2\log n)$.
- Vi ska strax se hur man kan hålla trädet balanserat och även få en värstafallstid på $O({}^2\log n)$.

Binära sökträd med minimal höjd

Den idealiska formen på ett binärt sökträd:

- Alla nivåer, utom möjligen den högsta, har så många noder som är möjligt.
- Det betyder att noderna ligger "så nära roten som möjligt".
- Då blir trädets höjd garanterat $O(^2 \log n)$.



Exempel på binära träd med minimal höjd

- Ett binärt träd är **perfekt** (eng: *perfect binary tree*) om alla noder utom löven har två barn och om alla löven befinner sig på samma nivå.

  Då har trädet formen:

- Ett binärt träd är **komplett** (eng: *complete binary tree*) om alla nivåer utom den högsta är fyllda med noder och om noderna på den högsta nivån är samlade "längst till vänster".

  Då har trädet formen:

Binära sökträd - tidskomplexitet

- Det finns ingen tillräckligt effektiv algoritm för att se till att ett binärt sökträd får den idealiska formen.
- Trädet behöver inte ha denna form för att garantera att alla operationer blir $O(^2 \log n)$ i värsta fall.
  - Det räcker att garantera att höjden är proprtionell mot $^2 \log n$, d.v.s. att $h = O(^2 \log n)$.
  - Det finns effektiva algoritmer för att i samband med insättning och borttagning garantera att trädets höjd alltid är $O(^2 \log n)$.

Balanserade binära sökträd – AVL-träd

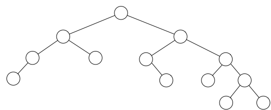
Georgy Adelson-Velsky and Evgenii Landis gjorde följande definition:

Balanserat binärt träd

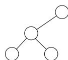
Ett binärt träd är balanserat om det för varje nod i trädet gäller att höjdskillnaden mellan dess båda subträd är högst ett.

De visade också att

- I balanserade träd är höjden $h \leq 1.44 * {}^2 \log n$.
- Det finns effektiva algoritmer för att se till att binära sökträd förblir balanserade vid insättningar och borttagningar.

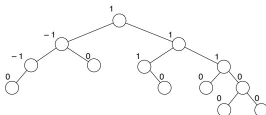


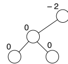
Balanserat träd



Obalanserat träd

- *balans* $= h_R - h_L$
  - $h_R =$ höger subträds höjd
  - $h_L =$ vänster subträds höjd

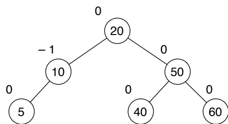




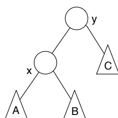
Balanserat träd

Obalanserat träd

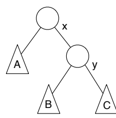
- Vad händer med trädets balansering om man lägger till 2?
- Hur kan man rätta till det?

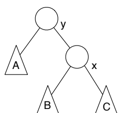


Balanseringsalgoritmerna arbetar med rotationer i trädet:

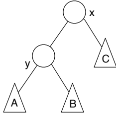


Enkel höger-
rotation vid y
$\Longrightarrow$





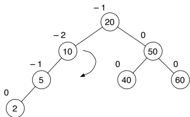
Enkel vänster-
rotation vid y
$\Longrightarrow$



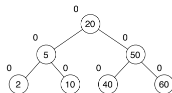
Balansering av binära sökträd

Exempel

Obalanserat vid 10:

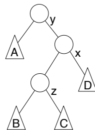


Efter en enkel högerrotation:

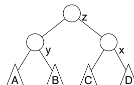


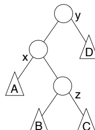
Balansering av binära sökträd

Ibland behövs dubbelrotationer:

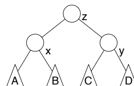


Höger-vänster-dubbelrotation





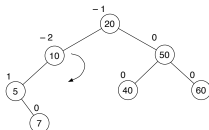
Vänster-höger-dubbelrotation



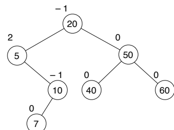
Balansering av binära sökträd

Exempel - dubbelrotationer

Efter insättning av 7 får man trädet:



Det råder nu obalans vid 10 men om man försöker med en enkel högerrotation blir det:

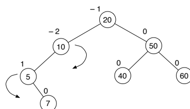


Detta träd är fortfarande obalanserat! Istället måste man göra en dubbel vänster-högerrotation (se nästa bild).

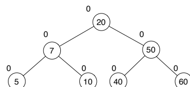
Balansering av binära sökträd

Exempel - dubbelrotationer

Obalanserat vid 10:



Efter en vänster-högerrotation:



Balansering av binära sökträd

Obalans måste kunna upptäckas:

- Man kan ha ett heltalsattribut `balance` i nodklassen.
- I `balance` bokförs höjdskillnaden mellan höger och vänster subträd.
- I samband med insättning/borttagning ändras ev. höjden av subträd och attributet uppdateras
- Om höjdskillnaden blir > 1 eller < -1 så åtgärdas det med rotation(er) som i sin tur förändrar höjd och `balance` hos subträd.
- Efter eventuella rotation(er) blir absolutbeloppet av `balance` åter ≤ 1.

Balansering av binära sökträd – kostnad

Man kan visa att:

- Om obalans uppstår till följd av en insättning i ett tidigare balanserat träd så räcker det med en enkel- eller dubbelrotation för att återställa balansen.
- Om obalans uppstår till följd av en borttagning ur ett tidigare balanserat träd så kan det behövas en enkel- eller dubbelrotation i varje nod på vägen från den nod där obalans uppstod till roten för att återställa balansen.

Balansering av binära sökträd – kostnad

- Höjden förblir $O(^2\log n)$ om man balanserat trädet vid varje förändring.
- En enskild rotation kostar bara $O(1)$.
- Kostnaden för balansering i samband med en insättning eller borttagning är $O(^2\log n)$.
- Om ett binärt sökträd hålls balanserat kommer sökning, insättning och borttagning därmed att kosta $O(^2\log n)$ i värsta fall.

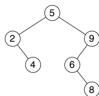
Exempel på vad du ska kunna

- Förklara begreppet binära sökträd.
- Förklara hur insättning, sökning och borttagning i ett binärt sökträd går till och kunna implementera sökning och insättning.
- Känna till och kunna implementera interfacet `Comparable`.
- Förklara begreppet balanserat binärt sökträd och varför man vill ha balanserade träd.
- Förklara begreppet AVL-träd.
- Ange tidskomplexitet för operationer på binära sökträd.
- Känna till och kunna implementera interfacet `Comparator`.
- Kunna skicka med lambdauttryck som argument till parametrar av typen `Comparator`.

Datorlaboration 5

Binära sökträd

- Implementera en egen generisk klass för binära sökträd.



- Tips:
  - I flera fall blir det en (kort) publik metod som anropar en privat rekursiv metod.
  - I en av metoderna ska ett nytt träd byggas upp från värden i en vektor. Hämta inspiration från den rekursiva algoritmen för binärsökning.
  - Rita för att förstå vad som händer i programmet!
- Innehåll: binära sökträd, rekursion, länkad struktur.

Datorlaboration 5, forts

Insättning i binära sökträd

- Nycklarna i vänster subträd < roten < nycklarna i höger subträd
- Dubbletter är ej tillåtna.
- Nya noder sätts alltid in som löv.
- Insättning börjar med en (förhoppningsvis) misslyckad sökning.
  - Exempel: Sätt in 3.