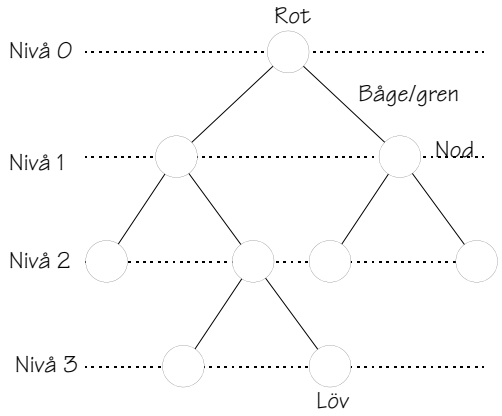
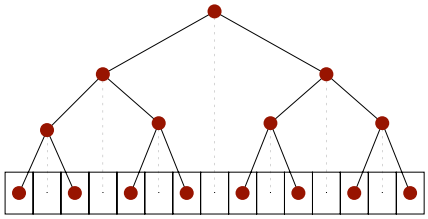
Träd



- Förälder
- Barn
- Intern nod
- Föregångare (eng. ancestors)
- Efterföljare (eng. descendants)
- Subträd

Binär sökning är trädsökning

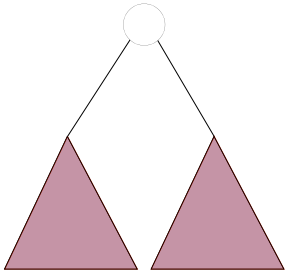


Binärt träd

Ett binärt träd är antingen ett tomt träd eller en trädnod med två binära subträd.

Tomt binärt träd:

Icke-tomt binärt träd:



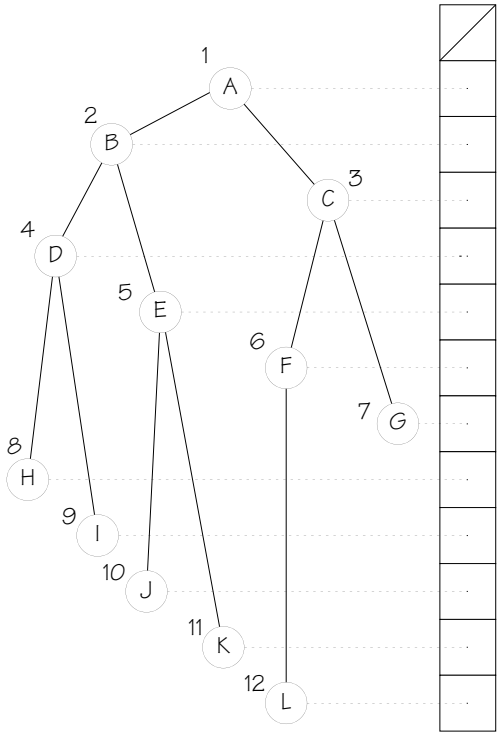
Komplett binärt träd

Låt T vara ett binärt träd.
Antag att T har n st nivåer.

T kallas ett komplett binärt träd om:

- Alla löv är på nivå n eller n-1;
- Alla löv på nivå n är "vänsterplacerade";
- Det finns inga "tomma platser" på nivåerna < n.

Kontinuerlig sekvensiell representation



Heap

- En heap är ett komplett binärt träd där varje nod har egenskapen att alla noder i de båda subträden har lägre nycklar.
- En heap kan representeras med en array!!!
- En prioritetskö kan representeras med en heap!!!

Prioritetskö med heap-representation

```
class PriorityQueue {

  private int count;
  private int capacity;
  private int capacityIncrement;

  private ComparisonKey[] items;

  public PriorityQueue() {
    count = 0;
    capacity = 16;
    capacityIncrement = 8;
    items = new ComparisonKey[capacity];
  }

  public int size() {
    return count;
  }

  public ComparisonKey remove() {
    // Se OH 9
  }

  public void insert(ComparisonKey item) {
    // Se OH 10
  }

} // end PriorityQueue class
```


Prioritetskö (forts.)

```
interface ComparisonKey {
  int compareTo(ComparisonKey value);
  String toString();
}


class PQItem implements ComparisonKey {

  int key;

  PQItem(int value) {
    key = value;
  }

  public String toString() {
    return Integer.toString(key);
  }

  public int compareTo(ComparisonKey value) {
    int a = this.key;
    int b = ((PQItem)value).key;
    if( a == b ) return 0;
    else if( a > b ) return 1;
    else return -1;
  }

}
```

Remove

```
public ComparisonKey remove() {

 if (count == 0) return null;
 else {
  int current;
  int child;
  ComparisonKey itemToPlace;
  ComparisonKey itemToReturn;

  itemToReturn = items[1];
  itemToPlace = items[count--];
  current = 1;
  child = 2 * current;

  while (child <= count) {
   if(child < count) {
    if(items[child+1].compareTo(items[child])>0){
      child++;
    }
   }
   if( items[child].compareTo(itemToPlace) > 0 ){
      items[current] = items[child];
      current = child;
      child = 2 * current;
   } else {
      items[current] = itemToPlace;
      return itemToReturn;
   }
  } //end while

  items[current] = itemToPlace;
  return itemToReturn;
 } //end if
} //end remove()
```


Insert

```
public void insert(ComparisonKey item) {

  if( count == capacity - 1 ) {
    capacity += capacityIncrement;
    ComparisonKey[] tempArray =
    new ComparisonKey[capacity];
    for( int i = 1; i <= count; i++ ) {
      tempArray[i] = items[i];
    }
    items = tempArray;
  }

  count++;
  int child = count;
  int parent = child/2;
  while (parent != 0) {
    if( item.compareTo(items[parent])<=0 ) {
      items[child] = item;
      return;
    } else {
      items[child] = items[parent];
      child = parent;
      parent = child/2;
    } //end if
  } //end while
  items[child] = item;

} //end insert()
```


Komplexitet

	Bygg kö[a]	remove	insert
Heap	O(n)	O(log n)	O(log n)
Sorterad lista	$O(n^2)$	O(1)	O(n)
Osorterad array	O(1)	O(n)	O(1)

a.) Från en osorterad array

Traversering av binärt träd

- Pre-order
  (1) roten
  (2) vänster delträd i pre-order
  (3) höger delträd i pre-order

- In-order
  (1) vänster delträd i in-order
  (2) roten
  (3) höger delträd i in-order

- Post-order
  (1) vänster delträd i post-order
  (2) höger delträd i post-order
  (3) roten

- Level-order
  (1) nivå 0
  (2) nivå 1
  (3) nivå 2,
  etc.

Traversering av länkad representation

```
class TreeNode {
  String   key;
  TreeNode left;
  TreeNode right;
}



static void preorder(TreeNode T) {
  if( nonempty(T) ) {
    visit(T);
    preorder(T.left);
    preorder(T.right);
  }
}

static void inorder(TreeNode T) {
  if( nonempty(T) ) {
    inorder(T.left);
    visit(T);
    inorder(T.right);
  }
}

static void postorder(TreeNode T) {
  if( nonempty(T) ) {
    postorder(T.left);
    postorder(T.right);
    visit(T);
  }
}
```

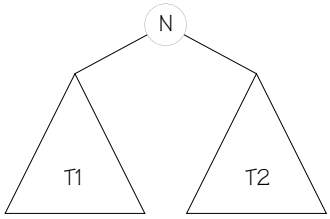

Traversering (ickerekursiv)

```
void traverse(TreeNode T) {

  Stack S = new Stack();
  TreeNode N;

  S.push(T);

  while( ! S.empty() ) {

    N = (TreeNode)S.pop();

    if( nonempty(N) ) {
      visit(N);
      S.push(N.right);
      S.push(N.left);
    }

  }

}
```


Traversering (ickerekursiv)

```
void traverse(TreeNode T) {

  Queue Q = new Queue();
  TreeNode N;

  Q.insert(T);

  while( ! Q.empty() ) {

    N = (TreeNode)Q.remove();

    if( nonempty(N) ) {
      visit(N);
      Q.insert(N.left);
      Q.insert(N.right);
    }

  }

}
```


Binärt sökträd



Ett binärt träd kallas ett binärt sökträd om det för varje nod gäller att om noden har nyckeln N så är

- alla nycklar i T1 mindre än N
- alla nycklar i T2 är större än N