Binary Search Trees

- Last lecture:
  - Tree terminology
  - Kinds of binary trees
  - Size and depth of trees
- This time:
  - binary search tree ADT
  - Java implementation

Keys and records

- So far most examples assumed that we sort, insert, delete and search for integers.
- In fact in most applications we manipulate more complicated items.
- In general, assume that we manipulate records which have keys which can be compared, searched for etc.

Example

- Record: employee's name, address, telephone, position, NI number, payroll number.
- Key: payroll number.
- Instead of sorting an array of integers in ascending order, could sort an array of records in the ascending order of keys.
- Binary search for a record given a key, etc.

Data structures for storing data

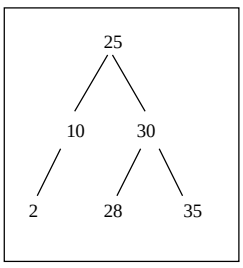
- How easy it is to insert a new record
- How easy it is to remove a record
- How easy it is to find a record given a key

Motivation

- Binary search on ordered arrays is efficient: $O(\log_2 N)$
- However insertion of an item in an ordered array is slow: O(N)
- Ordered arrays are best suited for static searching, where search space does not change.
- Binary search trees can be used for efficient dynamic searching.

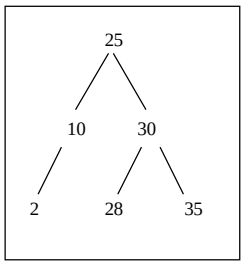
Binary Search Trees

- Binary trees
- Store data sorted in order of keys: all values stored in the left subtree of a node with key value K have key values < K, those in the right subtree have values >= K.



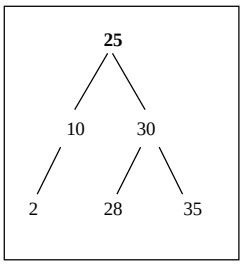
BST Search

- Searching for a record with key K in a tree:
- if tree is empty return null
- else check if K = key of the root (if yes return the found record)
- if the K < key of the root, search the left subtree
- otherwise search the right subtree



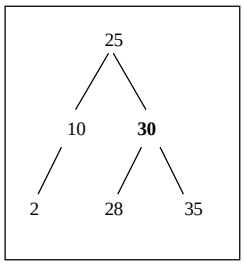
Example

- Searching for a record with key 27



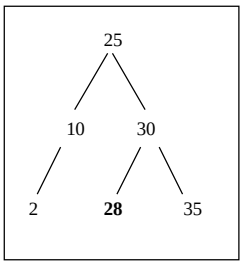
Example

- Searching for a record with key 27



Example

- Searching for a record with key 27
- Item not found.



Binary Search Tree ADT

- Logical domain: ordered binary trees which hold items of the form (key,record).
- Selected methods:
  - BST() : create an empty binary search tree
  - void insert(item) : insert item in BST, in key order
  - void remove(key) : remove item with the given key
  - item find(key): find and return item with the given key

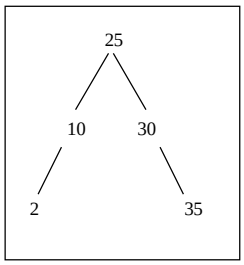
BST Insertion

Inserting an item with key K (not checking for duplicates) in a tree:

- If tree is empty, make this item the root
- Else compare K to the key of the root, R
- If K < R:
  - insert in the left subtree
- If K >= R:
  - insert in the right subtree

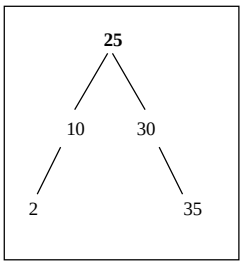
Example

- Inserting a record with key 15:



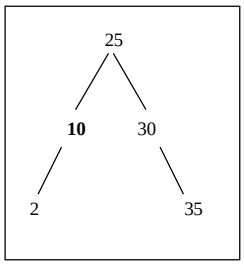
Example

- Inserting a record with key 15:



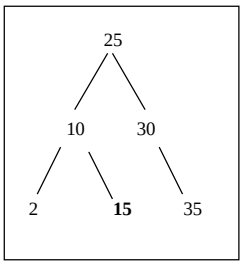
Example

- Inserting a record with key 15:



Example

- Inserting a record with key 15:

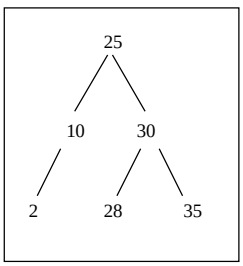


BST Deletion

- First, find the item
- If it is a leaf, just delete it
- If it is not a leaf:
  - if it has just one daughter, replace it by that daughter
  - if it has two daughters, replace it by the leftmost node of its right subtree

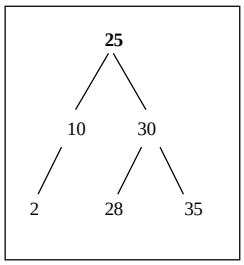
Example 1

- Easy case: delete a leaf (2)



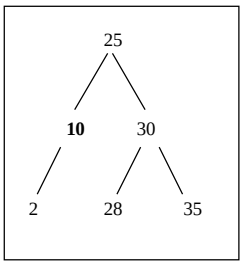
Example 1

- Easy case: delete a leaf (2)
- Find it



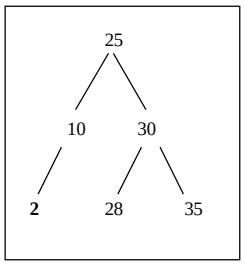
Example 1

- Easy case: delete a leaf (2)
- Find it



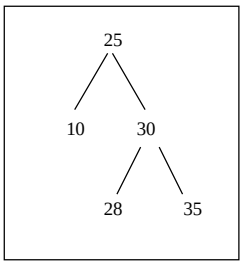
Example 1

- Easy case: delete a leaf (2)
- Find it



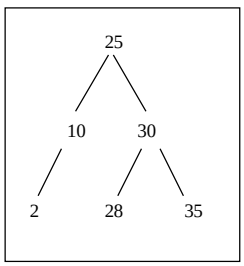
Example 1

- Easy case: delete a leaf (2)
- Find it
- Remove it



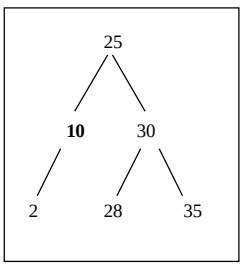
Example 2

- More difficult case: delete 10



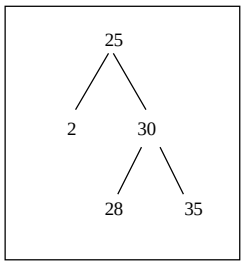
Example 2

- More difficult case: delete 10
- Find it



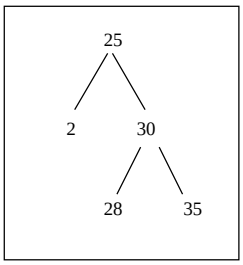
Example 2

- More difficult case: delete 10
- Find it
- Replace it by its daughter (2)



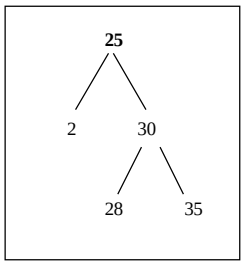
Example 3

- Last case: remove 25



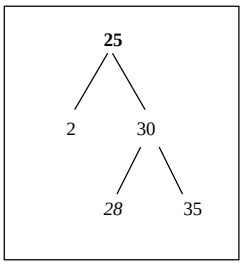
Example 3

- Last case: remove 25
- Find it



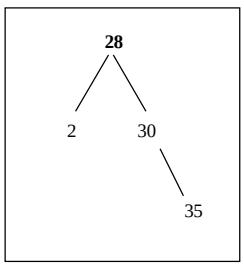
Example 3

- Last case: remove 25
- Find it
- Find its replacement (leftmost node in the right subtree: next larger key value)



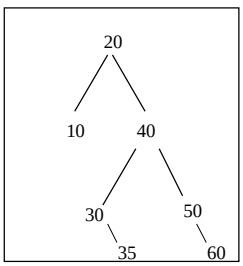
Example 3

- Last case: remove 25
- Find it
- Find its replacement (leftmost node in the right subtree: next larger key value)
- Replace



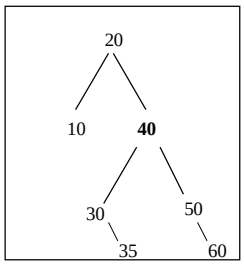
Exercise

- Remove 40



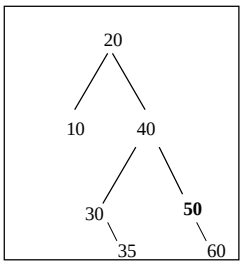
Exercise

- Remove 40
- Find it



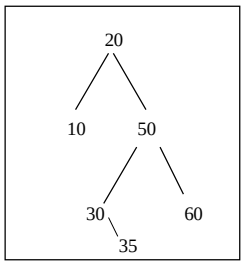
Exercise

- Remove 40
- Find it
- Find its replacement (leftmost node in the right subtree: next larger key value)



Exercise

- Remove 40
- Find it
- Find its replacement (leftmost node in the right subtree: next larger key value)
- Replace



Java Implementation

- Similar to linked lists:
- Define tree items (should have **`key()`** method)
- Define a tree node class (should have value and links to left and right daughters)
- Define a tree class which uses it (keeps the root value and has **`insert()`, `remove()`** and **`find()`** methods)
- Implementation taken from Shaffer's book.

BinNode interface

```
interface BinNode {
   public Object element();
   public Object setElement(Object v);
   public BinNode left();
   public BinNode setLeft(BinNode p);
   public BinNode right();
   public BinNode setRight(BinNode p);
   public boolean isLeaf();
}
```

BinNodePtr class

```
class BinNodePtr implements BinNode {
   private Object element;
   private BinNode left;
   private BinNode right;

   public BinNodePtr(){};
   public BinNodePtr(Object val){
      left=right=null;
      element = val;
   }
```

BinNodePtr class

```
public Object element(){
   return element;
}
public Object setElement(Object v){
   element = v;
   return v;
}
```

BinNodePtr class

```
BinNode left(){
   return left;
}
public BinNode setLeft(BinNode p){
   left = p;
   return p;
}
```

BinNodePtr class

```
public boolean isLeaf() {
  return((left==null)&&(right==null));
}
}
```

Items in the search tree

Assume that items implement the following interface (have integer keys):

```
interface Elem {
   public int key();
}
```

BST class

```
class BST {
   private BinNode root;
   public BST(){}

   public Elem find(int key){
     return findhelp(root, key);
   }
```

BST class

```
private Elem findhelp(BinNode r, int k){
   if (r == null) return null;
   Elem it = (Elem)r.element();
   if (it.key() > k) {
      return findhelp(r.left(), k);
   } else {
      if (it.key()==k) return it;
      else return findhelp(r.right(),k);
   }
}
```

BST class

```
public void insert(Elem val){
  root = inserthelp(root, val);
}
private BinNode inserthelp(BinNode r,
                           Elem v){
  if (r==null) return new BinNodePtr(v);
  Elem it = (Elem)r.element();
  if (it.key() > v.key()){
     r.setLeft(inserthelp(r.left(),v));
  }
```

BST class

```
  else {
     r.setRight(inserthelp(r.right(),v);
  }
  return r;
}
```

BST class

```
public void remove(int key){
   root = removehelp(root, key);
}
```

BST class

```
private BinNode removehelp(BinNode r,
                           int k){
 if (r==null) return null;
 Elem it = (Elem)r.element();
 if (k < it.key()){
   setLeft(removehelp(r.left(),k));
 } else if (k > it.key()){
   setRight(removehelp(r.right(),k));
 }
```

BST class

```
else { // Found the node to be deleted
   if (r.left()==null) r = r.right();
   else if (r.right()==null) r=r.left();
   else { // the node has two children
   Elem temp = getmin(r.right());
   r.setElement(temp);
   r.setRight(deletemin(r.right()));
   }
}
return r;
```

BST class

```
private Elem getmin(BinNode r){
  if (r.left() == null){
     return(Elem)r.element();
  } else {
     return getmin(r.left());
  }
}
```

BST class

```
private BinNode deletemin(BinNode r){
  if (r.left() == null){
     return r.right();
  } else {
     r.setLeft(deletemin(r.left());
     return r;
  }
}
```

Reading

- Shaffer, Chapter 5 (5.1, 5.2, 5.3, 5.5)