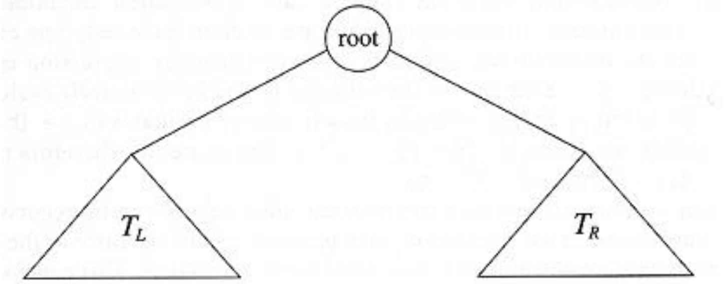
DATA STRUCTURES and ALGORITHMS in C++

FOURTH EDITION

Chapter 6: Binary Trees

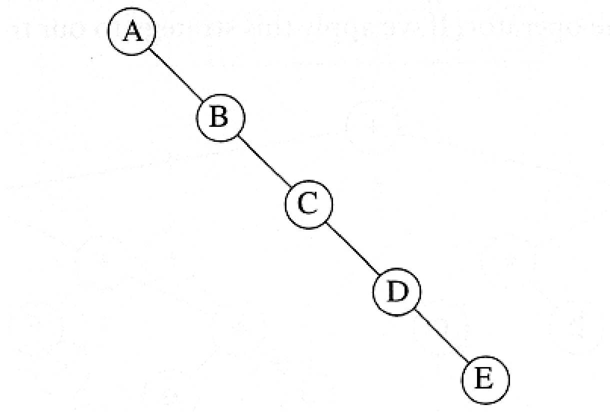
Binary Trees

- A tree in which no node can have more than two children



typical binary tree

- The depth of an "average" binary tree is considerably smaller than N, even though in the worst case, the depth can be as large as N – 1.



Worst-case binary tree

Node Struct of Binary Tree

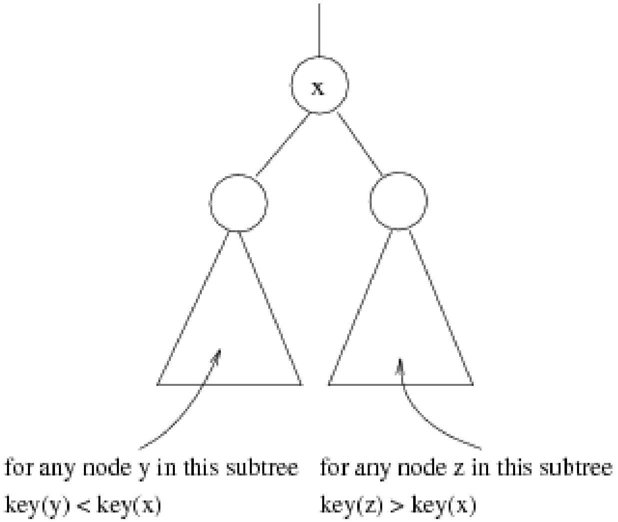
- **Possible operations on the Binary Tree ADT**
  - **Parent, left_child, right_child, sibling, root, etc**

- **Implementation**
  - **Because a binary tree has at most two children, we can keep direct pointers to them**

```
class Tree {
      int key;
      Tree* left, right;
}
```

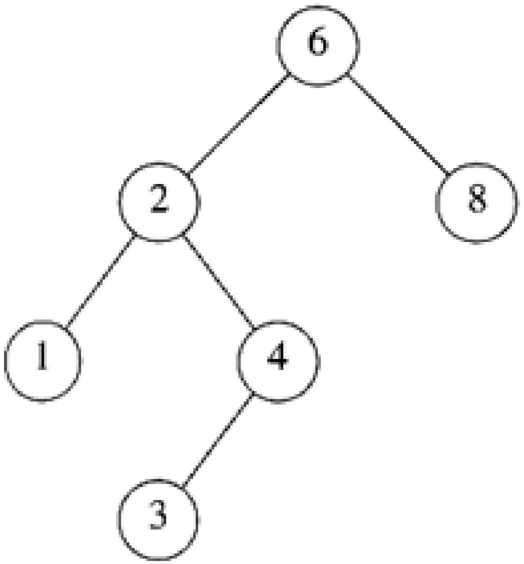
Binary Search Trees (BST)

- A data structure for efficient searching, insertion and deletion (dictionary operations)
  - All operations in worst-case O(log n) time
- Binary search tree property
  - For every node x:
  - All the keys in its left subtree are smaller than the key value in x
  - All the keys in its right subtree are larger than the key value in x



Binary Search Trees

Example:

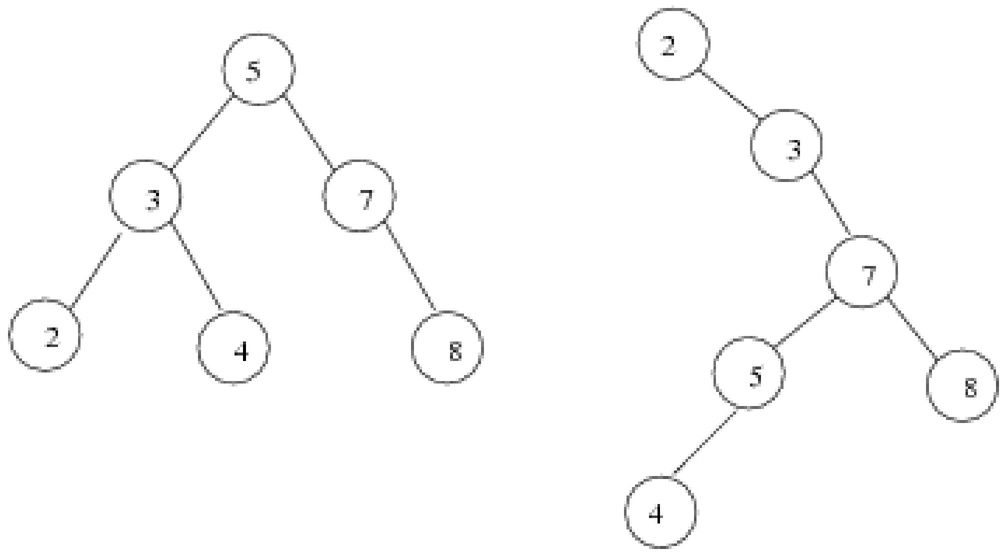


Tree height = 3

Key requirement of a BST: all the keys in a BST are distinct, no duplication

Binary Search Trees

The same set of keys may have different BSTs



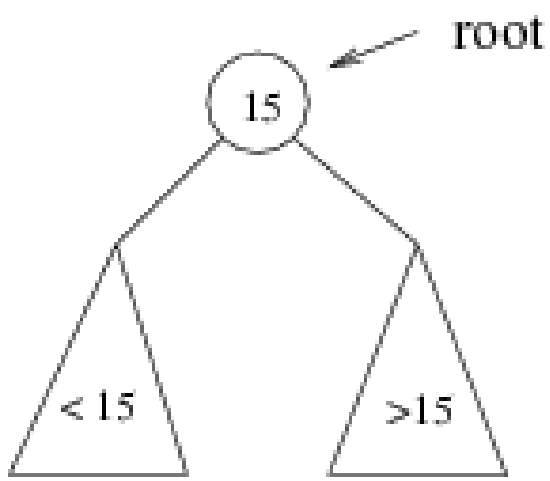
- **Average height of a binary search tree is O(log N)**
- **Maximum height of a binary search tree is O(N)**

(N = the number of nodes in the tree)

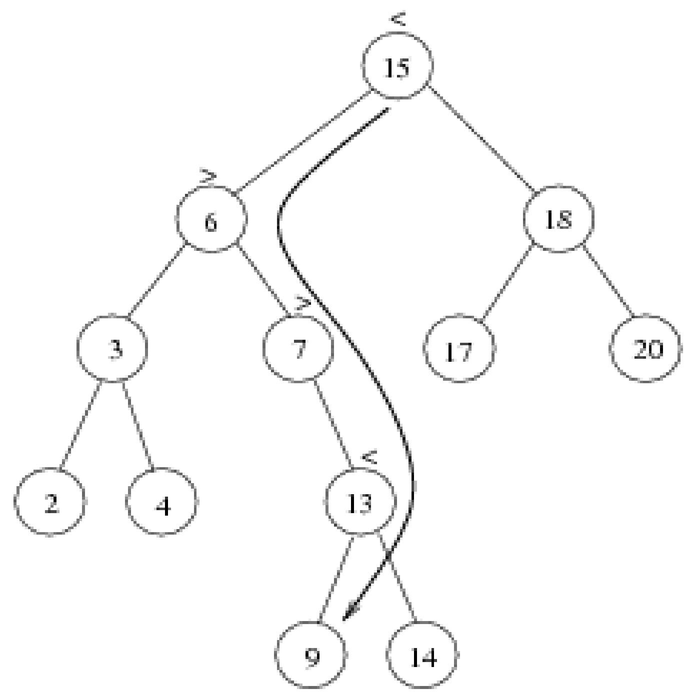
Searching BST

Example: Suppose T is the tree being searched:

- **If we are searching for 15, then we are done.**
- **If we are searching for a key < 15, then we should search in the left subtree.**
- **If we are searching for a key > 15, then we should search in the right subtree.**



Example: Search for 9 ...



Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!

Search (Find)

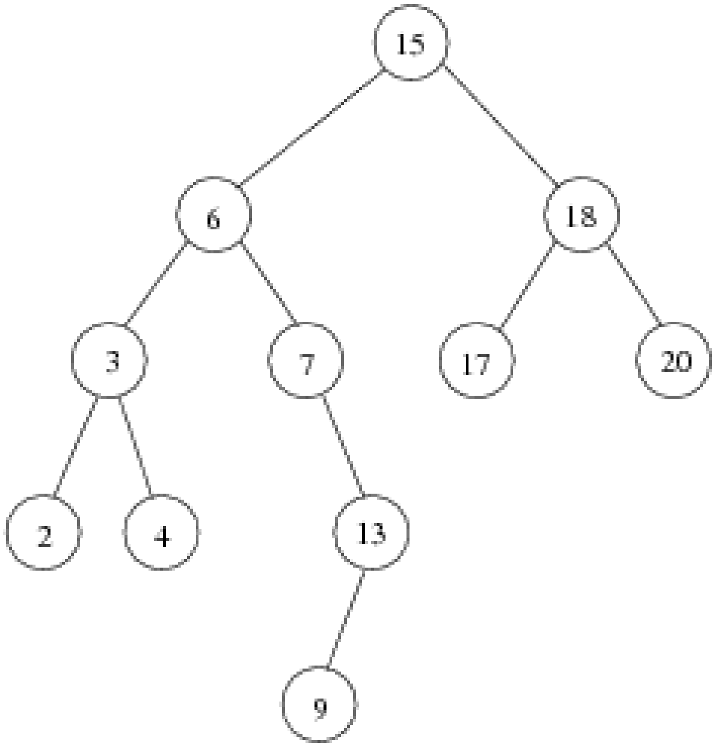
- **Find X: return a pointer to the node that has key X, or NULL if there is no such node**

```
Tree* find(int x, Tree* t) {
          if (t == NULL) return NULL;
          else if (x < t->key)
              return find(x, t->left);
          else if (x == t->key)
              return t;
          else return find(x, t->right);
       }
```

- **Time complexity: O(height of the tree) = O(log N) on average. (i.e., if the tree was built using a random sequence of numbers.)**

Inorder Traversal of BST

- **Inorder traversal of BST prints out all the keys in sorted order**



Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

findMin/ findMax

- Goal: return the node containing the smallest (largest) key in the tree
- Algorithm: Start at the root and go left (right) as long as there is a left (right) child. The stopping point is the smallest (largest) element

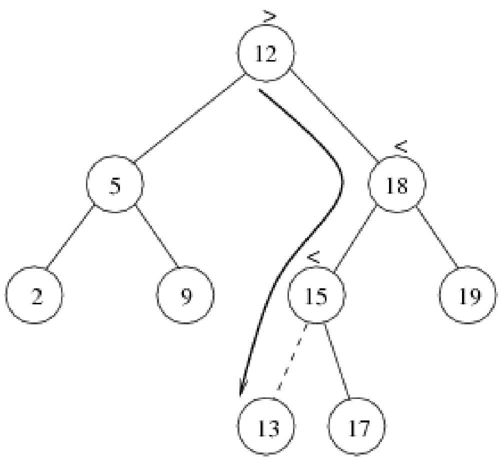
```
Tree* findMin(Tree* t) {
  if (t==NULL) return NULL;
  while (t->left != NULL)
      t = t->left;
  return t;
 }
```

- Time complexity = O(height of the tree)

Insertion

To insert(X):

- Proceed down the tree as you would for search.
- If x is found, do nothing (or update some secondary record)
- Otherwise, insert X at the last spot on the path traversed

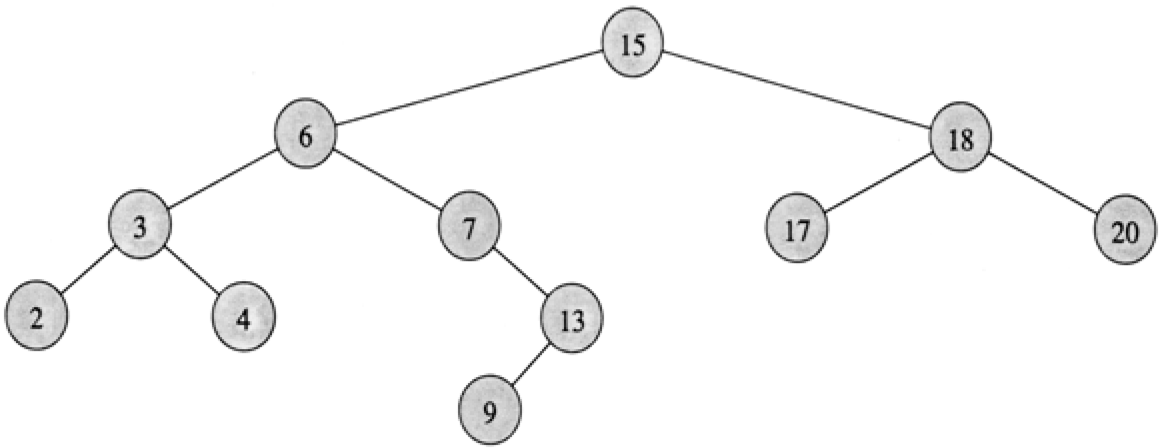


X = 13

- Time complexity = O(height of the tree)

Another example of insertion

Example: insert(11). Show the path taken and the position at which 11 is inserted.



Note: There is a <u>unique</u> place where a new key can be inserted.

Code for insertion

Insert is a recursive (helper) function that takes a pointer to a node and inserts the key in the subtree rooted at that node.

```
void insert(int x, Tree* & t) {
        if (t == NULL)
            t = new Tree(x, NULL, NULL);
        else if (x < t->key)
           insert(x, t->left);
        else if (x > t->key)
           insert(x, t->right);
       else ; // duplicate; do nothing
      }
```

Time complexity: O(h) as the algorithm just moves once from the root to a leaf.

Deletion under Different Cases

- Case 1: the node is a leaf
  - Delete it immediately
- Case 2: the node has one child
  - Adjust a pointer from the parent to bypass that node

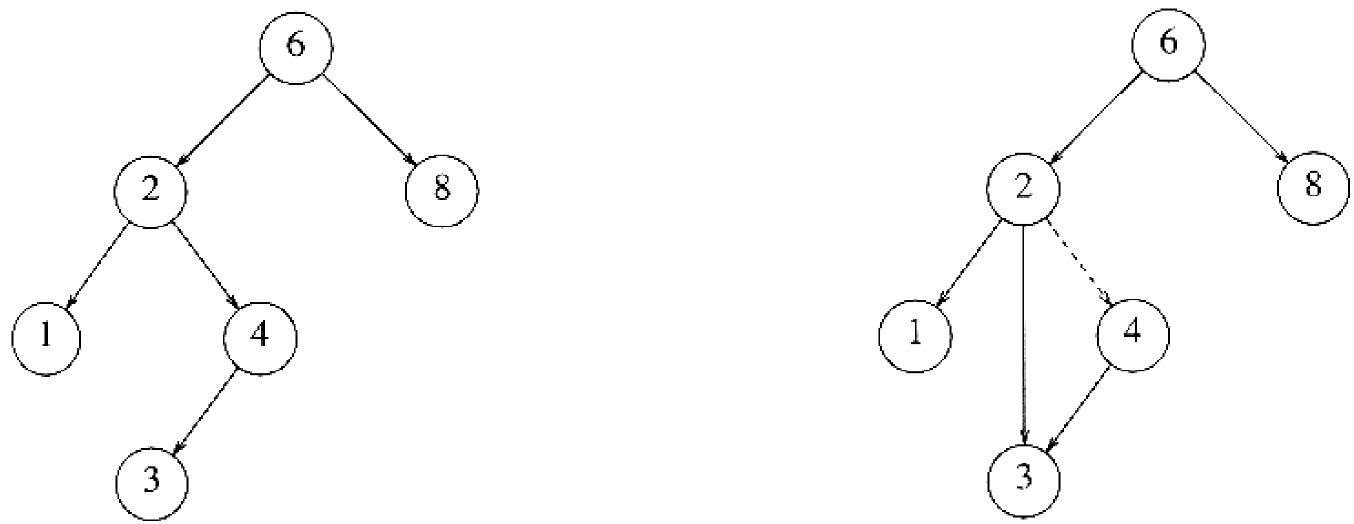


Figure 4.24 Deletion of a node (4) with one child, before and after

Deletion Case 3

- Case 3: the node has 2 children
  - Replace the key of that node with the minimum element at the right subtree
  - Delete that minimum element
    - Has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen. So invoke case 1 or 2.

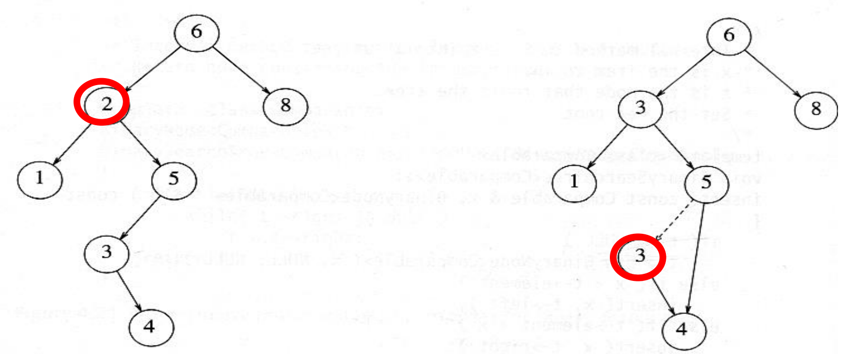


Figure 4.25 Deletion of a node (2) with two children, before and after

- Time complexity = O(height of the tree)

Code for Deletion

First recall the code for findMin.

```
Tree* findMin(Tree* t) {
      if (t==NULL) return NULL;
      while (t->left != NULL)
          t = t->left;
      return t;
     }
```

Code for Deletion

```
void remove(int x, BinaryTree* & t) {
   // remove key x from t
   if (t == NULL) return; // item not found; do nothing
   if (x < t->key)
        remove(x, t->left);
   else if (x > t->key) remove(x, t->right);
   else if (t->left != NULL && t->right != NULL) {
               t->key = findMin(t->right)->key;
               remove(t->key, t->right);
              }
   else {
         Tree* oldNode = t;
         t = (t->left != NULL) ? t->left: t->right;
         delete oldNode;
       }
}
```

Summary of BST

All the dictionary operations (search, insert and delete) as well as deleteMin, deleteMax etc. can be performed in O(h) time where h is the height of a binary search tree.

Good news:

- h is on average O(log n) (if the keys are inserted in a random order).
- code for implementing dictionary operations is simple.

Bad news:

- worst-case is O(n).
- some natural order of insertions (sorted in ascending or descending order) lead to O(n) height. (check this!)

Solution:

- enforce some condition on the tree structure that keeps the tree from growing unevenly.