Binary Trees, Binary Search Trees

Trees

- Linear access time of linked lists is prohibitive
  - Does there exist any simple data structure for which the running time of most operations (search, insert, delete) is O(log N)?
- Trees
  - Basic concepts
  - Tree traversal
  - Binary tree
  - Binary search tree and its operations

Trees

- A tree is a collection of nodes
  - The collection can be empty (the root (*T*) is NULL)
  - (recursive definition) If not empty, a tree consists of a distinguished node *T* (the *root*), and zero or more nonempty *subtrees* T_1, T_2, ..., T_k, each of whose roots are connected by a directed *edge* from r

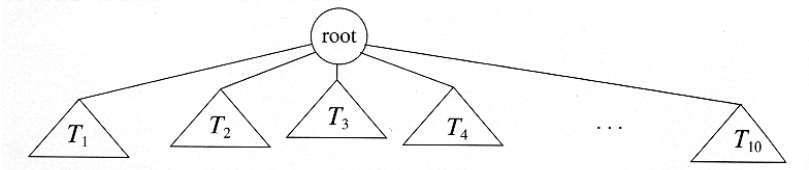


Figure 4.1 Generic tree

Some Terminologies

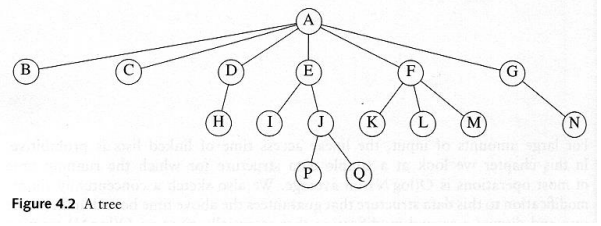


Figure 4.2 A tree

- *Child* and *Parent*
  - Every node except the root has one parent
  - A node can have an zero or more children
- *Leaves*
  - Leaves are nodes with no children
- *Sibling*
  - nodes with same parent

More Terminologies

- *Path*
  - A sequence of edges
  - A path from n_1 to n_k *is defined as a sequence of nodes,* n_1, n_2 … n_k, *such that* n_i *is the parent of* n_{i+1} *for* $\mathbf{1 \le I < k}$.
- *Length of a path*
  - number of edges on the path
- *Depth* of a node
  - length of the unique path from the root to that node
- *Height* of a node
  - length of the longest path from that node to the deepest leaf
  - all leaves are at height 0
- The height of a tree = the height of the root = the depth of the deepest leaf
- *Ancestor* and *descendant*
  - If there is a path from n1 to n2
  - n1 is an ancestor of n2, n2 is a descendant of n1
  - *Proper ancestor* and *proper descendant*

Example: UNIX Directory

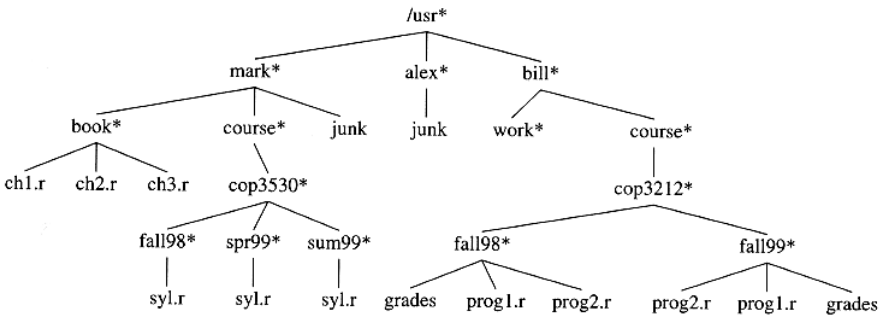


Figure 4.5 UNIX directory

Tree Implementation

```
class TreeNode
{
    Object   element;
    TreeNode firstChild;
    TreeNode nextSibling;
}
```

Figure 4.3 Node declarations for trees

```
typedef struct TreeNode *PtrToNode;

struct TreeNode
{
    ElementType Element;
    PtrToNode   FirstChild;
    PtrToNode   NextSibling;
}
```

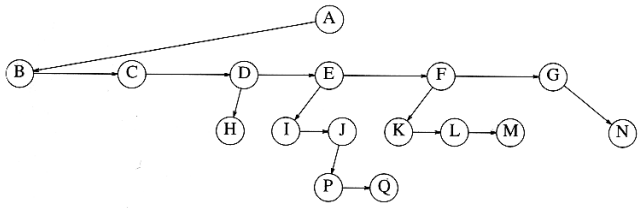


Figure 4.4 First child/next sibling representation of the tree shown in Figure 4.2

Tree Traversal

- Used to print out the data in a tree in a certain order
- **Pre-order traversal**
  - Print the data at the root
  - Recursively print out all data in the left subtree
  - Recursively print out all data in the right subtree
- **In-order traversal**
  - Recursively print out all data in the left subtree
  - Print the data at the root
  - Recursively print out all data in the right subtree
- **Post-order traversal**
  - Recursively print out all data in the left subtree
  - Recursively print out all data in the right subtree
  - Print the data at the root

Preorder, Postorder and Inorder

- **Preorder traversal**
  - root, left, right
  - prefix expression
    - ++a*bc*+*defg
- **Inorder traversal**
  - left, root, right
  - infix expression
    - a+b*c+d*e+f*g
- **Postorder traversal**
  - left, right, root
  - postfix expression
    - abc*+de*f+g*+

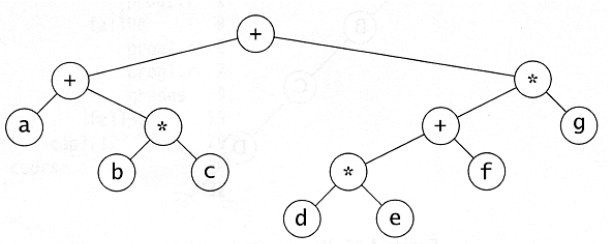
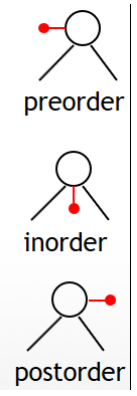
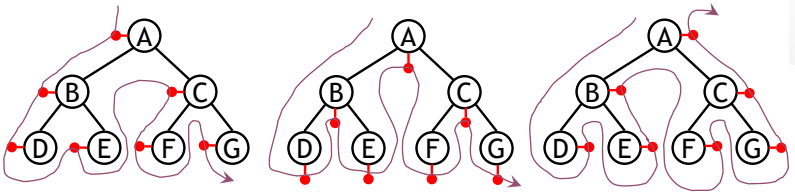


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Tree Traversal … Example

- ❑ Preorder: visits nodes as root → left → right
- ❑ Inorder: visits nodes as left → root → right
- ❑ Postorder: visits nodes as left → right → root





Preorder Inorder Postorder

Preorder, Postorder and Inorder Pseudo Code

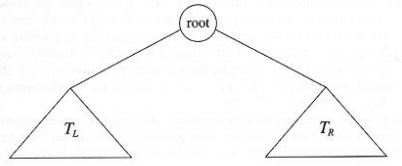
```
Algorithm Preorder(x)
Input: x is the root of a subtree.
1.   if x ≠ NULL
2.      then output key(x);
3.           Preorder(left(x));
4.           Preorder(right(x));
```

```
Algorithm Postorder(x)
Input: x is the root of a subtree.
1.   if x ≠ NULL
2.      then Postorder(left(x));
3.           Postorder(right(x));
4.           output key(x);
```

```
Algorithm Inorder(x)
Input: x is the root of a subtree.
1.   if x ≠ NULL
2.      then Inorder(left(x));
3.           output key(x);
4.           Inorder(right(x));
```

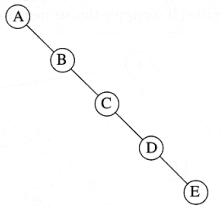
Binary Trees

- A tree in which each node has at most two children (0, 1, or 2)



Generic 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

Full and Complete Binary Tree

There are two common forms of binary trees:

1. **Full binary tree**: a binary tree T is full if each node is either **a leaf** or possesses **exactly two child nodes**.

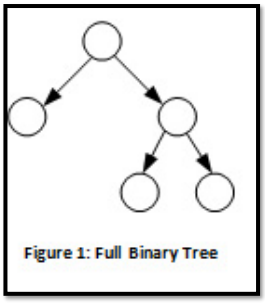
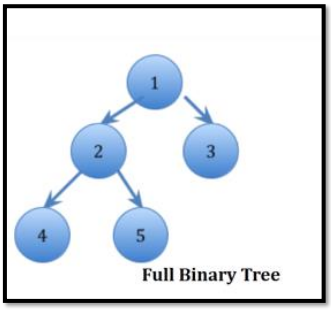


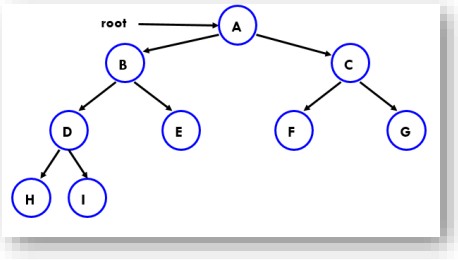
Figure 1: Full Binary Tree

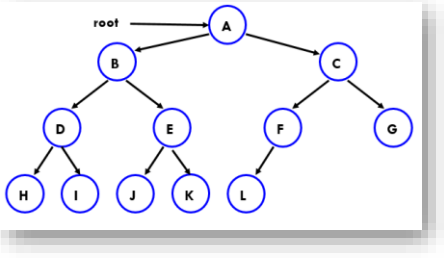


Full Binary Tree

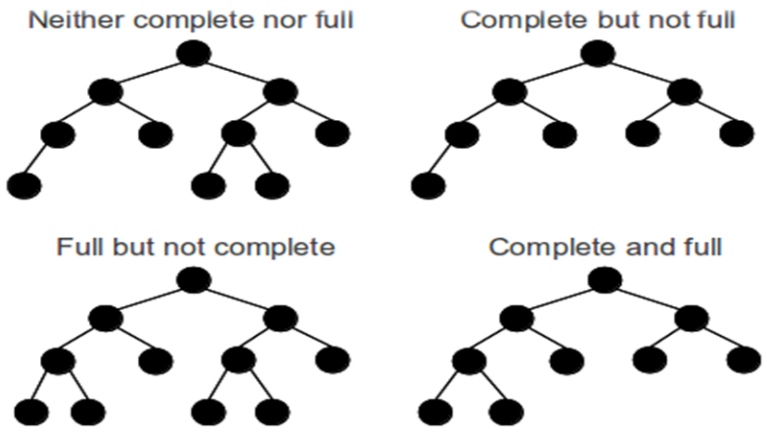
Full and Complete Binary Tree

A complete binary tree: is a binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right.





Full and Complete Binary Tree



Node class/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

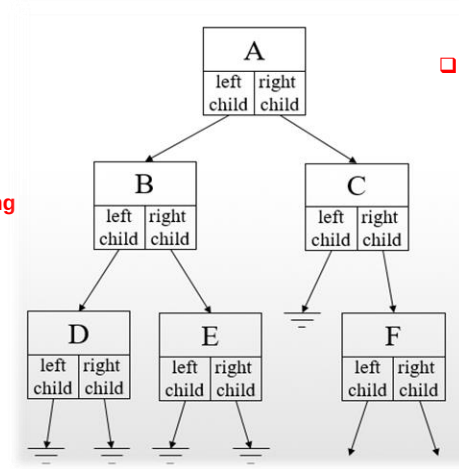
```
class BinaryNode
{
        // Friendly data; accessible by other package
    Object     element;      // The data in the node
    BinaryNode left;         // Left child
    BinaryNode right;        // Right child
}
```

```
struct TreeNode
{
    ElementType Element;
    SearchTree  Left;
    SearchTree  Right;
};
```

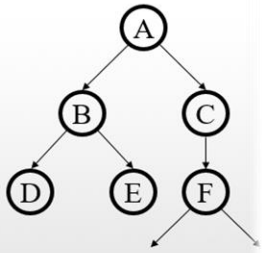
Figure 4.13 Binary tree node class

Binary Tree Representation

Each node is labeled as being either
a left child or
a right child

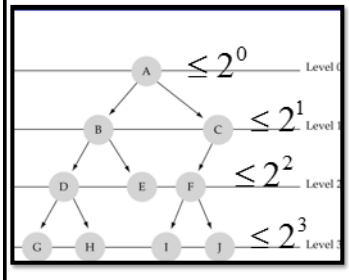


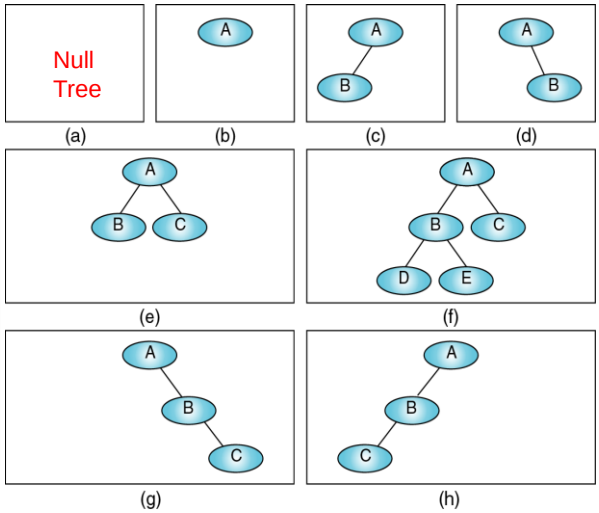
- Maximum number of children for each node is 2



Binary Tree Representation

Max number of node in each level <= 2^L where L=0,1,2,…,L-1





Example: Expression Trees

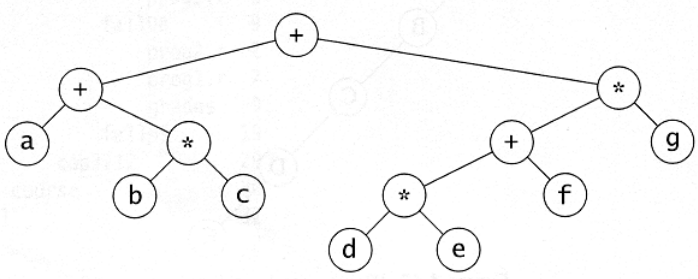


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- Leaves are operands (constants or variables)
- The internal nodes contain operators
- Will not be a binary tree if some operators are not binary

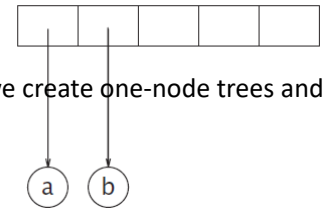
Constructing an Expression Tree

- An algorithm to convert a postfix expression into an expression tree
  - Read the expression one symbol at a time.
    - If the symbol is an operand, create a one-node tree and push it onto a stack.
    - If the symbol is an operator, pop two trees *T*1 and *T*2 from the stack (*T*1 is popped first) and
  - Form a new tree whose root is the operator and whose left and right children are *T*2 and *T*1, respectively. This new tree is then pushed onto the stack.

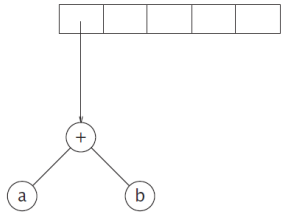
Example

- Suppose the input is

  a b + c d e + * *

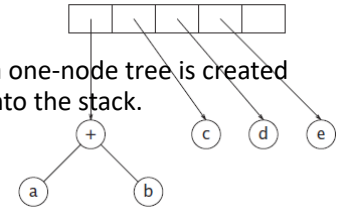


- The first two symbols are operands, so we create one-node trees and push them onto a stack.
- Next, a + is read, so two trees are popped, a new tree is formed, and it is pushed onto the stack.

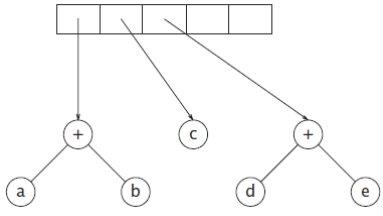


Example cont…

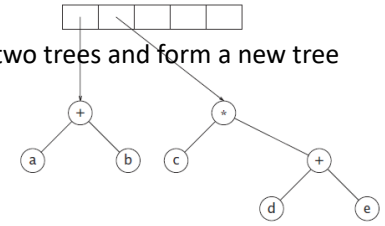
- Next, c, d, and e are read, and for each a one-node tree is created and the corresponding tree is pushed onto the stack.



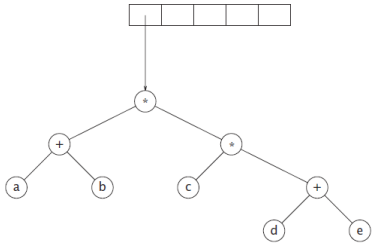
- Now a + is read, so two trees are merged.



- Continuing, a * is read, so we pop two trees and form a new tree with a * as root.



- Finally, the last symbol is read, two trees are merged, and the final tree is left on the stack.

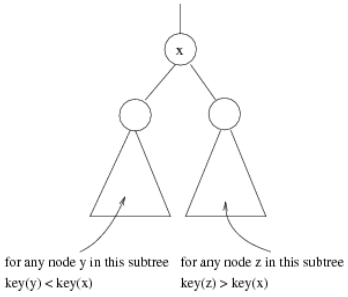


What about the time?

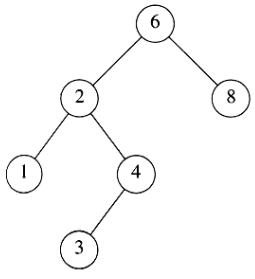
- Search ?
- Insert?
- Delete?
- …

Binary Search Trees (BST)

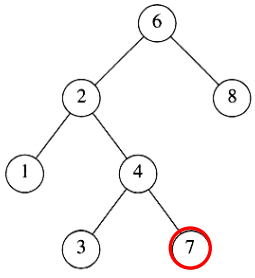
- A data structure for efficient searching, insertion and deletion
- Binary search tree property
  - For every node X
  - All the keys in its left subtree are smaller than the key value in X
  - All the keys in its right subtree are larger than the key value in X



Binary Search Trees



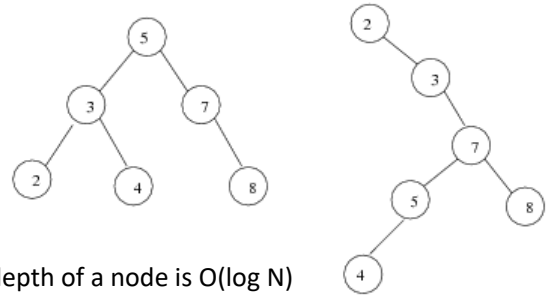
A binary search tree



Not a binary search tree

Binary Search Trees

The same set of keys may have different BSTs

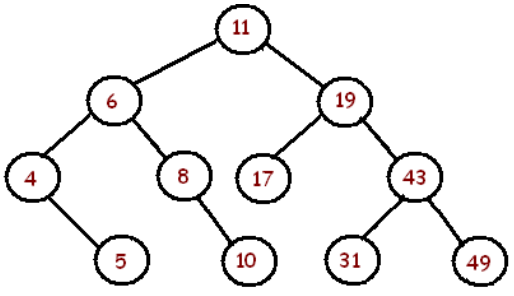


- Average depth of a node is O(log N)
- Maximum depth of a node is O(N)

Constructing a Binary Search Tree

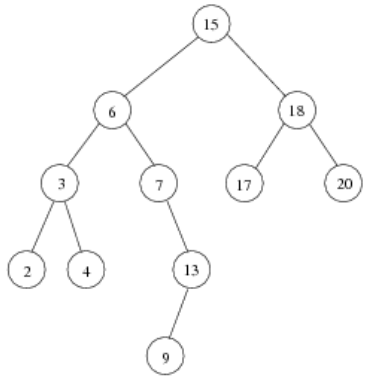
Exercise. Given a sequence of numbers:

```
11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31
```



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

BST Implementation

```
public BinarySearchTree( )
  { root = null; }

public void makeEmpty( )
  { root = null; }
public boolean isEmpty( )
  { return root == null; }
```

BST Implementation

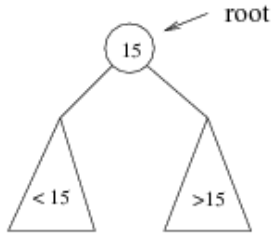
```
struct TreeNode;
typedef struct TreeNode *Position;
typedef struct TreeNode *SearchTree;
```

```
struct TreeNode
{
    ElementType Element;
    SearchTree  Left;
    SearchTree  Right;
};
```

```
SearchTree
MakeEmpty( SearchTree T )
{
    if( T != NULL )
    {
        MakeEmpty( T->Left );
        MakeEmpty( T->Right );
        free( T );
    }
    return NULL;
}
```

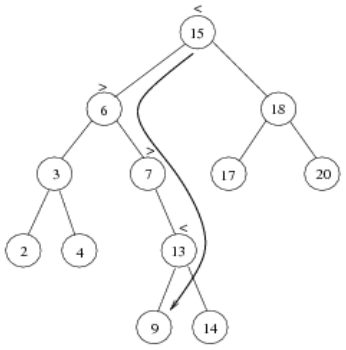
Searching BST

- 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.



Searching (Find)

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!

Searching (Find)

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

```
private boolean contains( AnyType x, BinaryNode<AnyType> t )
{
    if( t == null )
        return false;

    int compareResult = x.compareTo( t.element );

    if( compareResult < 0 )
        return contains( x, t.left );
    else if( compareResult > 0 )
        return contains( x, t.right );
    else
        return true;    // Match
}
```

- Time complexity: O(height of the tree)

Searching (Find)

```
Position
Find( ElementType X, SearchTree T )
{
    if( T == NULL )
        return NULL;
    if( X < T->Element )
        return Find( X, T->Left );
    else
    if( X > T->Element )
        return Find( X, T->Right );
    else
        return T;
}
```

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

Recursive implementation of findMin:

```
private BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
{
    if( t == null )
       return null;
    else if( t.left == null )
        return t;
    return findMin( t.left );
}
```

Non-recursive implementation of findMax:

```
private BinaryNode<AnyType> findMax( BinaryNode<AnyType> t )
{
    if( t != null )
        while( t.right != null )
            t = t.right;

    return t;
}
```

Time complexity = O(height of the tree)

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

Recursive implementation of findMin:

```
Position
FindMin( SearchTree T )
{
    if( T == NULL )
        return NULL;
    else
    if( T->Left == NULL )
        return T;
    else
        return FindMin( T->Left );
}
```

Non-recursive implementation of findMax:

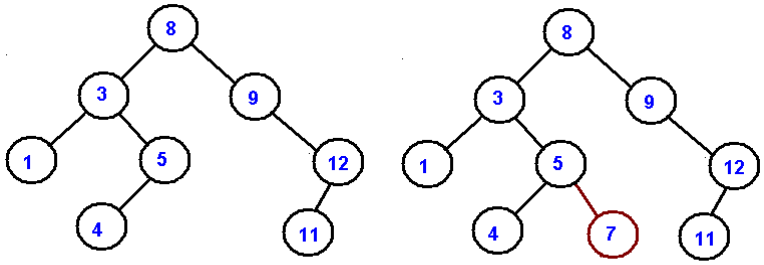
```
Position
FindMax( SearchTree T )
{
    if( T != NULL )
        while( T->Right != NULL )
            T = T->Right;

    return T;
}
```

Time complexity = O(height of the tree)

Insert

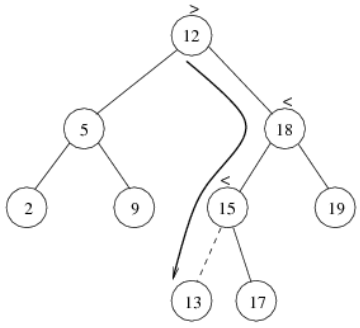
- The insertion procedure is quite similar to searching.
  - Start at the root and recursively go down the tree searching for a location in a BST to insert a new node.
  - If the element to be inserted is already in the tree, we are done (we do not insert duplicates). The new node will always replace a NULL reference.

  Example: insert (7)



before insertion after insertion

Insertion … Example



Time complexity = O(height of the tree)

```
private BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )
{
    if( t == null )
        return new BinaryNode<>( x, null, null );

    int compareResult = x.compareTo( t.element );

    if( compareResult < 0 )
        t.left = insert( x, t.left );
    else if( compareResult > 0 )
        t.right = insert( x, t.right );
    else
        ;  // Duplicate; do nothing
    return t;
}
```

```
SearchTree
Insert( ElementType X, SearchTree T )
{
    if( T == NULL )
    {
        /* Create and return a one-node tree */
        T = malloc( sizeof( struct TreeNode ) );
        if( T == NULL )
            FatalError( "Out of space!!!" );
        else
        {
            T->Element = X;
            T->Left = T->Right = NULL;
        }
    }
    else
    if( X < T->Element )
        T->Left = Insert( X, T->Left );
    else
    if( X > T->Element )
        T->Right = Insert( X, T->Right );
    /* Else X is in the tree already; we'll do nothing */

    return T;  /* Do not forget this line!! */
}
```

Deletion

- When we delete a node, we need to consider how we take care of the children of the deleted node.
  - This has to be done such that the property of the search tree is maintained.
- Deletion is somewhat more tricky than insertion.
- There are several cases to consider. A node to be deleted may be:
  1. not in a tree;
  2. is a leaf;
  3. has only one child;
  4. has two children.

Deletion under Different Cases

- Case 1: the node is a leaf
  - Delete it immediately
- Case 2: the node has one child
  - The procedure of deletion is identical to deleting a node from a linked list - Adjust a pointer from the parent to bypass that node

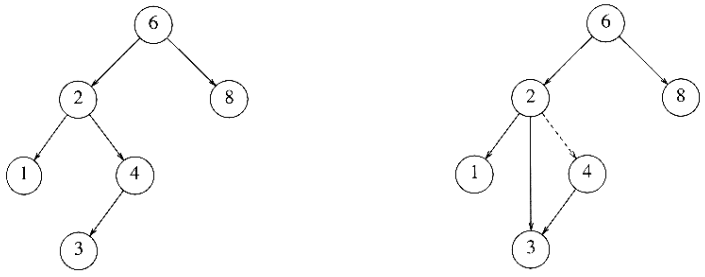
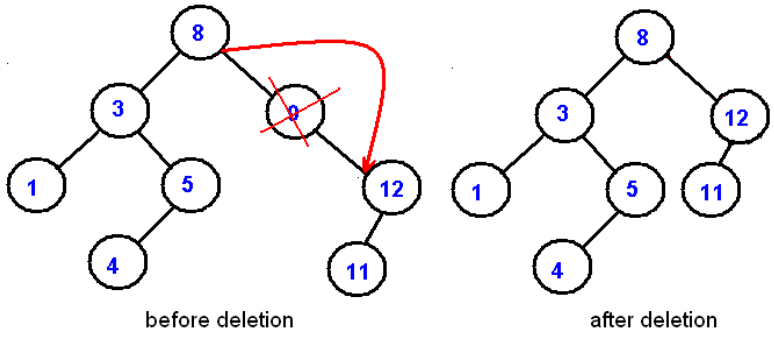


Figure 4.24 Deletion of a node (4) with one child, before and after

Case 2: Example



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.

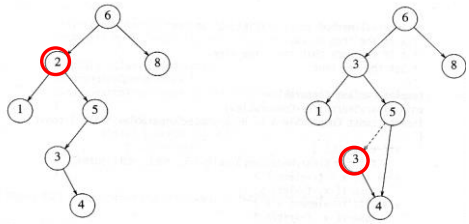
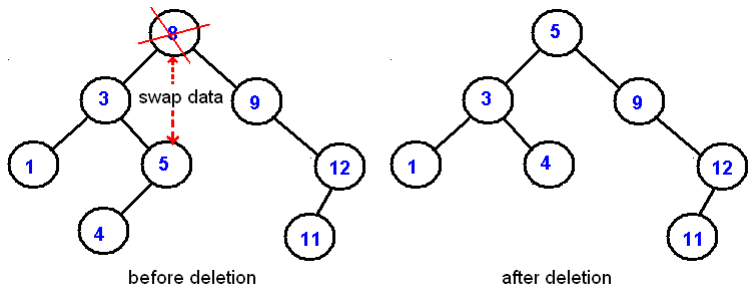


Figure 4.25 Deletion of a node (2) with two children, before and after

- Time complexity = O(height of the tree)

Case 3: Example



Delete implementation

```
private Node<T> delete(Node<T> p, T toDelete)
{
    if (p == null)  throw new RuntimeException("cannot delete.");
    else
        if (compare(toDelete, p.data) < 0)
            p.left = delete (p.left, toDelete);
        else
            if (compare(toDelete, p.data)  > 0)
                p.right = delete (p.right, toDelete);
            else
            {
                if (p.left == null) return p.right;
                else
                    if (p.right == null) return p.left;
                    else
                    {
                        // get data from the rightmost node in the left subtree
                        p.data = retrieveData(p.left);
                        // delete the rightmost node in the left subtree
                        p.left =  delete(p.left, p.data) ;
                    }
            }
    return p;
}
```

```
SearchTree
Delete( ElementType X, SearchTree T )
{
    Position TmpCell;

    if( T == NULL )
        Error( "Element not found" );
    else
    if( X < T->Element )  /* Go left */
        T->Left = Delete( X, T->Left );
    else
    if( X > T->Element )  /* Go right */
        T->Right = Delete( X, T->Left );
    else  /* Found element to be deleted */
    if( T->Left && T->Right )  /* Two children */
    {
        /* Replace with smallest in right subtree */
        TmpCell = FindMin( T->Right );
        T->Element = TmpCell->Element;
        T->Right = Delete( T->Element, T->Right );
    }
    else  /* One or zero children */
    {
        TmpCell = T;
        if( T->Left == NULL ) /* Also handles 0 children */
            T = T->Right;
        else if( T->Right == NULL )
            T = T->Left;
        free( TmpCell );
    }

    return T;
}
```