MODULE 10

TREE

DATA STRUCTURE AND ALGORITHMS

FACULTY OF COMPUTING
UNIVERSITI TEKNOLOGI MALAYSIA

OBJECTIVES FOR STUDENTS

1. Understand the tree concept and terms related to tree.
2. Identify characteristics of general tree, binary tree and binary search tree.
3. Identify basic operations of a tree such as tree traversals, insert node, delete node, searching.
4. Understand and know how to apply and implement tree concept in problem solving and programming.

KEY CONCEPT

1.0 INTRODUCTION TO TREE

1.1. **Tree** is a non-linear data structure. Data in a tree is stored in hierarchy.

1.2. **Example** of tree applications:

1. Represent algebraic formulas. Terminal nodes store operands, non-terminal nodes store operators.

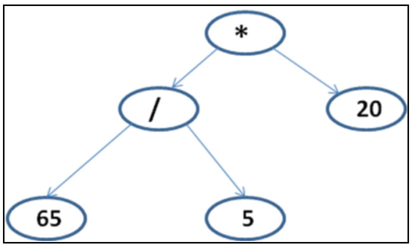


Figure 10.1 : Algebraic formula : (65 / 5) * 20

2. Organization chart that organize information in hierarchy form.

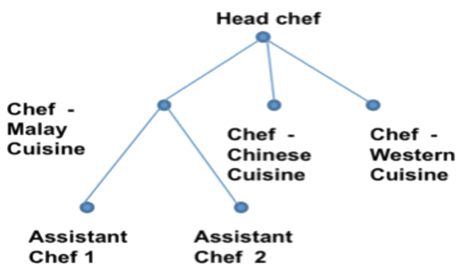


Figure 10.2 : Organization Chart for Chef Department in a Restaurant

3. Artificial intelligence – information is accessed based on certain decision which is stored in a tree.
   - Binary tree associated with a decision process.
   - Internal nodes: questions with yes/no answer.
   - External nodes: decisions.

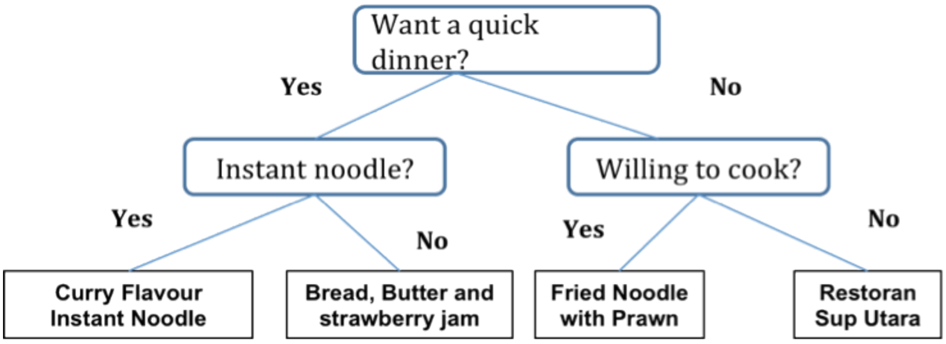


Figure 10.3 : Decision Tree : Dining Decision

1.3. **Tree Definition** :

- A tree is a collection of nodes and edges that connect the nodes.
- The collection can be empty.
- If not empty, a tree consists of a distinguished node *r* (the root), and zero or more nonempty subtrees T_1, T_2, ..., T_k, each of whose roots are connected.

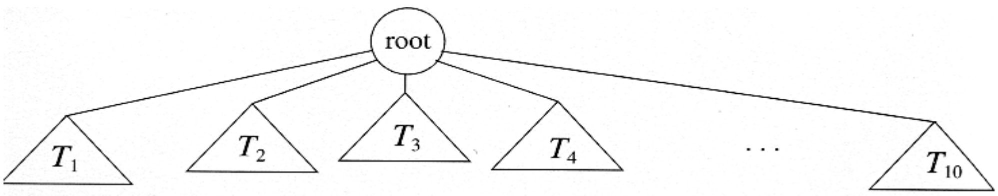


Figure 10.4 : Generic Tree

- Any two vertices must have one and only one path between them else it's not a tree .

- Trees are hierarchical
  - Parent-child relationship between two nodes.
  - Ancestor-descendant relationships among nodes.

2.0 TREE TERMINOLOGY

2.1. **General Tree** - A general tree *T* is a set of one or more nodes such that *T* is partitioned into disjoint subsets:

1. A single node *r*, the root.
2. Sets that are general trees, called subtrees of *r*.

- Each node in general tree can have unlimited children.

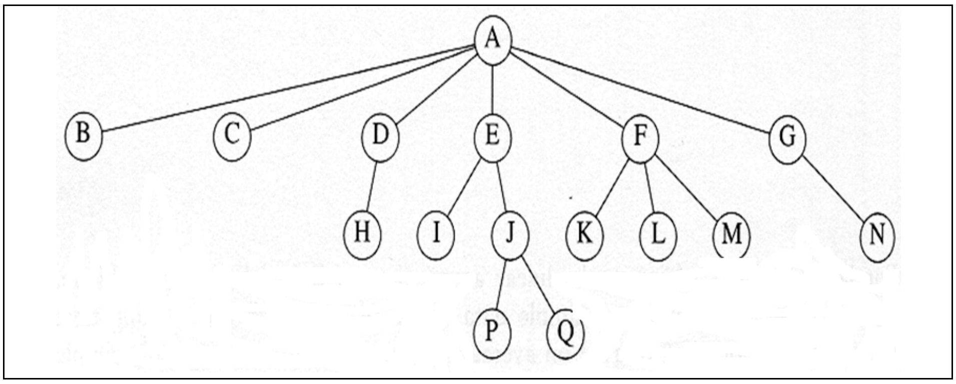
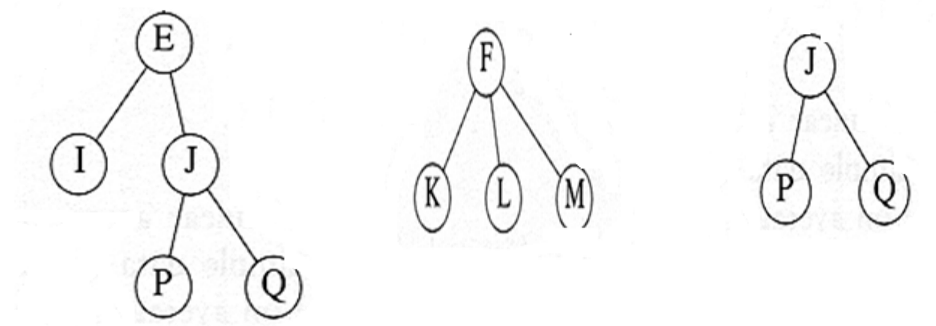


Figure 10.5 : A General Tree.

2.2. **Subtree** of a tree: Any node and its descendants.

- Subtree of node *n* - A tree that consists of a child of node *n* and the child's descendants.



Subtree of A   Another subtree of A   Right subtree of E

Figure 10.6 : The Subtree extracted from the general tree.

2.3. **Root** – The only node in the tree with no parent.

- A tree has only one root.
- Example of root from Figure 10.5 : A

2.4. **Child and Parent**

- Every node except the root has one parent.

Parent of node *n* - The node directly above node *n* in the tree.

- A is Parent to B,C,D,E,F,G.
- E is parent to I and J.
- A node can have an arbitrary number of children.

Child of node *n* - A node directly below node *n* in the tree.
- B,C,D,E,F,G are children of A.
- K, L and M are children of F.

2.5. **Leaves** – Also known as terminal nodes. Nodes with no children.
- B, C, H, I, P, Q, K, L, M and N are example of children.

2.6. **Sibling –** Nodes that have the same parent.
- P and Q are siblings.
- K, L and M are siblings.

2.7. **Ancestor of node *n*** – A node on the path from the root to *n*.
- Ancestor P : J, E and A.
- Nod A is ancestor for all nodes in the tree.

2.8. **Descendant of node *n***
- A node on a path from *n* to a leaf.
- Descendant of E: I, J, P and Q
- All nodes in the tree are descendant to the root.

2.9. **Path -** sequence of nodes in which each node is adjacent to the next one.
- Example: Path from root to L: AFL.
- Path from root to P: AEJP

2.10. **Length** – number of edges on the path.
- Length of Tree : 3

2.11. **Depth of a node** – length of the unique path from the root to that node.
- The depth of a tree is equal to the depth of the deepest leaf
- Depth of Tree : 3
- Depth of K : 2

2.12. **Height of a tree** – Number of nodes along the longest path from the root to a leaf. The level of the leaf in the longest path from the root plus 1.

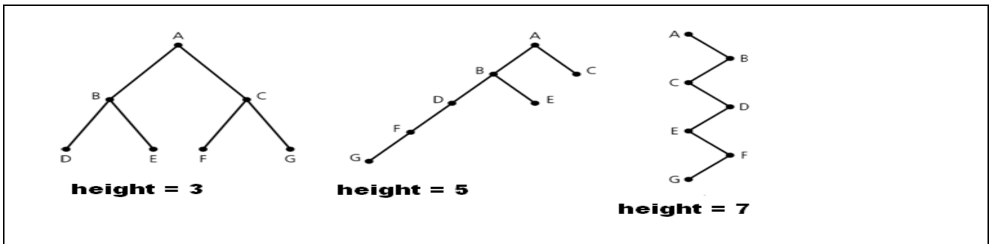


Figure 10.7 : Binary trees with the same nodes but different heights.

3.0 BINARY TREE

3.1. **Binary Tree -** a tree in which no node can have more than two children/

subtrees (left subtree and right subtree).

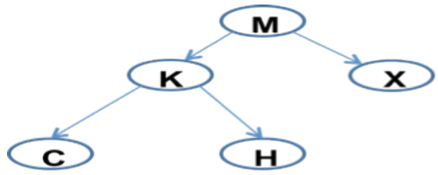


Figure 10.8: Binary Tree

3.2. A **binary tree** is a set *T* of nodes such that either :

- *T* is empty, or
- *T* is partitioned into three disjoint subsets:
  - A single node *r*, the root and,
  - Two possibly empty sets that are binary trees, called the left subtree of *r* and the right subtree of *r*.

3.3. Collection of Binary Trees.

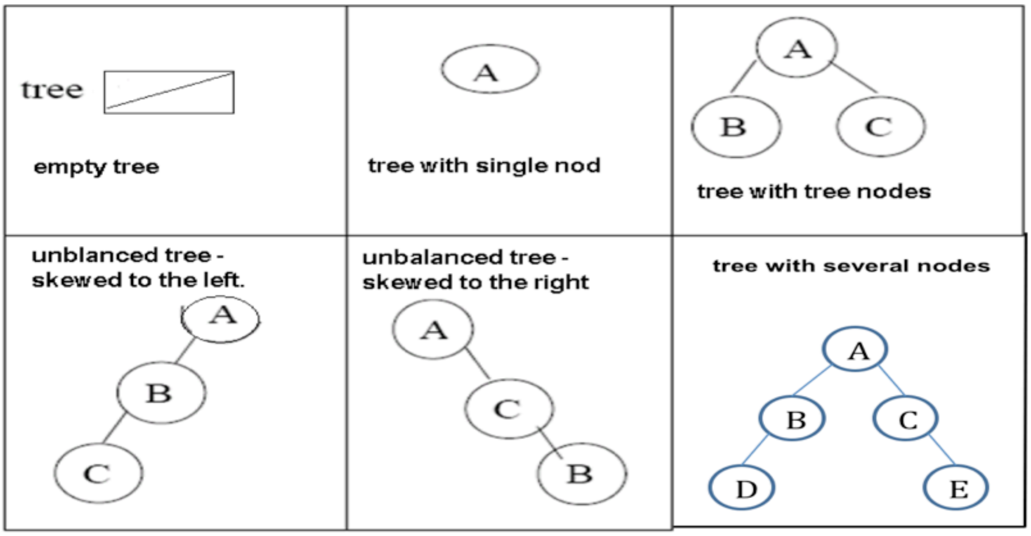


Figure 10.9 Collection of binary trees.

3.4. 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***.

3.5. **Binary Search Tree**

A binary tree that has the following properties for each node *n*

- *n*'s value is > all values in *n*'s left subtree T_L
- *n*'s value is < all values in *n*'s right subtree T_R
- Both T_L and T_R are binary search trees

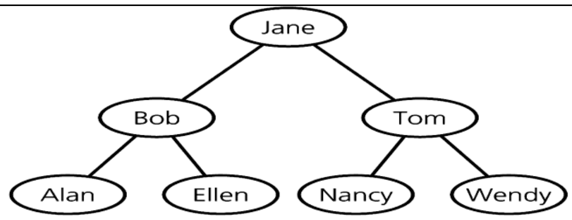


Figure 10.10: Binary Search Tree that store names

3.6. **Full Tree –** A binary tree of height h is full if, nodes at levels < *h* have two children each.

- Recursive definition
  - If *T* is empty, *T* is a full binary tree of height 0
  - If *T* is not empty and has height *h* > 0, *T* is a full binary tree if its root's subtrees are both full binary trees of height *h* – 1.

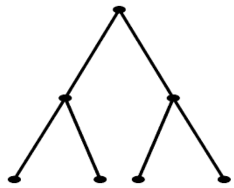


Figure 10.11 Full Tree

3.7. **Complete Binary Tree** – A binary tree of height *h* is complete if,

- It is full to level *h*–1, and
- Level *h* is filled from left to right

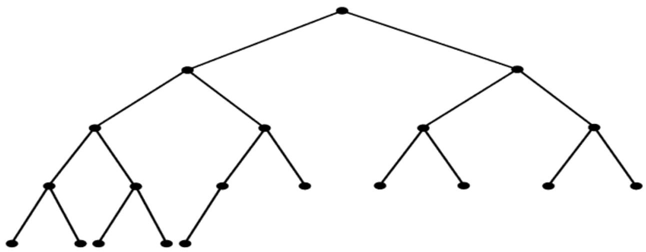


Figure 10.12 Complete Binary Tree

3.8. Counting Nodes in a Complete Binary Tree

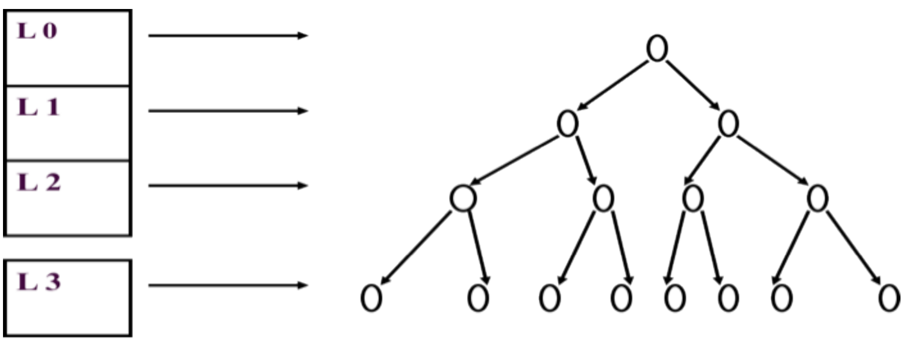


Figure 10.13 Complete and Full Binary Tree

At each level the number of the nodes is doubled.

total number of nodes: $1 + 2 + 2^2 + 2^3 = 2^4 - 1 = 15$

Number of the nodes in a tree with **M** levels:

$$1 + 2 + 2^2 + \ldots.\ 2^M = 2^{(M+1)} - 1 = 2*2^M - 1$$

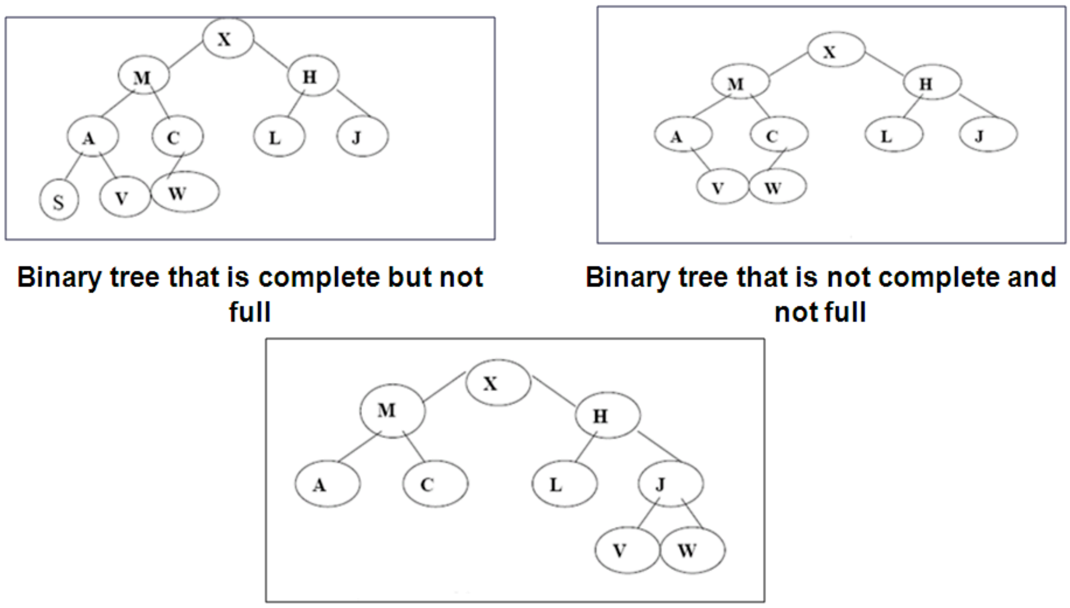
Let **N** be the number of the nodes.

$$N = 2*2^M - 1$$
$$2*2^M = N + 1$$
$$2^M = (N+1)/2$$
$$M = \log((N+1)/2)$$

N nodes : $\log((N+1)/2) = O(\log(N))$ levels

M levels: $2^{(M+1)} - 1 = O(2^M)$ nodes

3.9. Binary Tree Variations



Binary tree that is complete but not full

Binary tree that is not complete and not full

Binary tree that is not complete and not full

Figure 10.14: Variations of Binary Tree

3.10. Balanced Binary Tree

- A binary tree is *balanced* if the heights of any node's two subtrees differ by no more than 1
- Complete binary trees are balanced
- Full binary trees are complete and balanced

3.11. Expression Tree

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

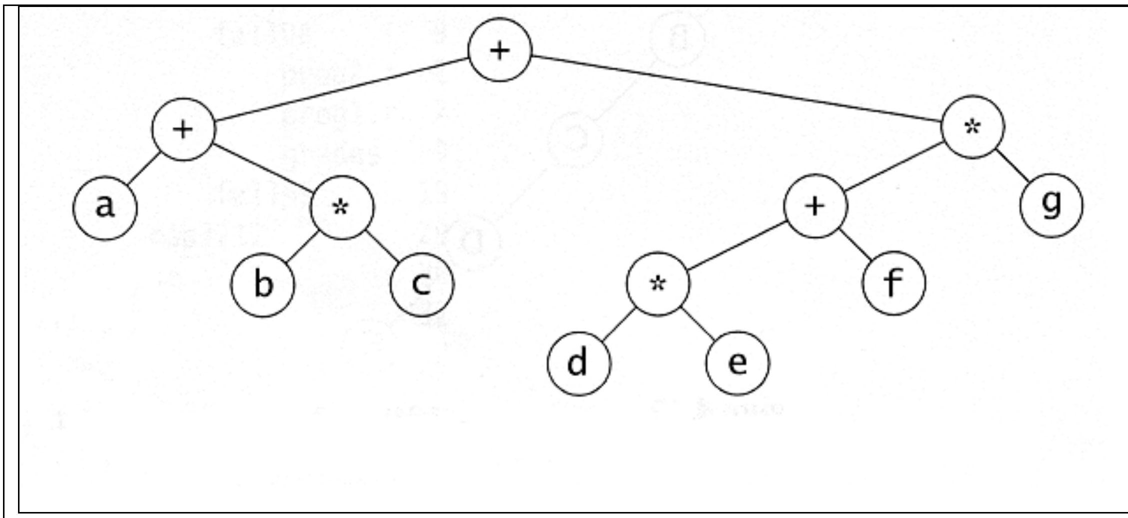


Figure 10.15 Expression Tree for (a + b * c) + ((d * e + f) * g)

3.12. **Tree Traversal**

- A traversal visits every node in a tree
- You do something with the node during a visit
  - For example, display the data in the node
  - Used to print out the data in a tree in a certain order
- Type of traversal - Inorder traversal, Preorder traversal and Postorder traversal

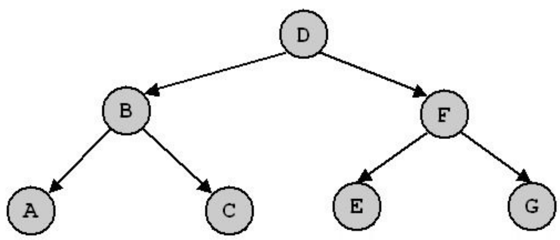


Figure 10.16 Traversing Binary Tree

3.13. **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

D	B A C	F E G
Root	left subtree (root left right)	right subtree (root left right)

- Pre-order traversal : D B A C F E G

3.14. **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

A C B	E G F	D
left subtree (left right root)	right subtree (left right root)	root

- Postorder traversal : A C B E G F D

3.15. **Inorder 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

A B C	D	E F G
left subtree (left root right)	root	right subtree (left root right

3.16. **Traversing an expression tree**

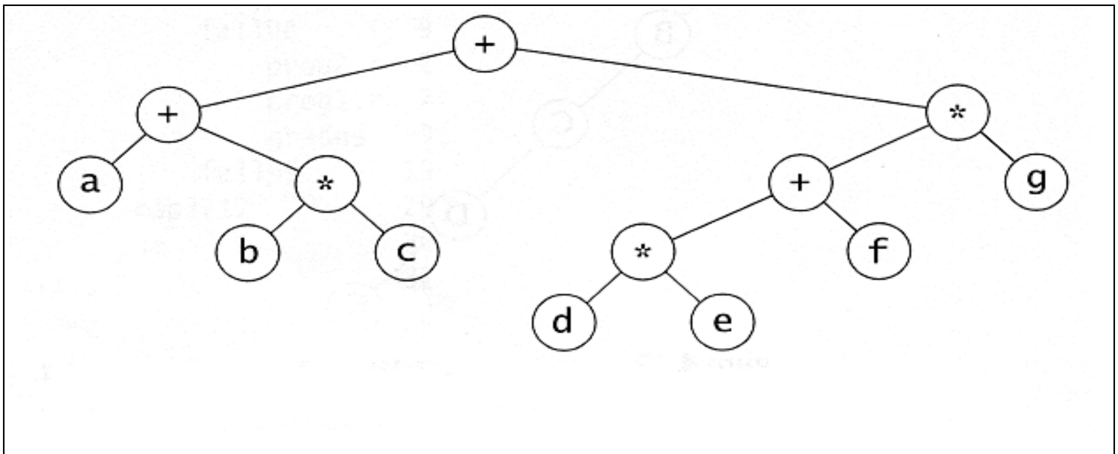


Figure 10.17 Traversing Expression Tree for (a + b * c) + ((d * e + f) * g)

- Pre-order traversal (prefix expression) : ++a*bc*+*defg
- Postorder traversal (postfix expression) : abc*+de*f+g*+
- Inorder traversal (infix expression) : a+b*c+d*e+f*g

4.0 BINARY SEARCH TREE

4.1. For every node X, all the keys in its **left subtree are smaller** than the key value in X, and all the keys in its **right subtree are larger** than the key value in X.

4.2. Stores keys in the nodes in a way so that searching, insertion and deletion can be done efficiently.

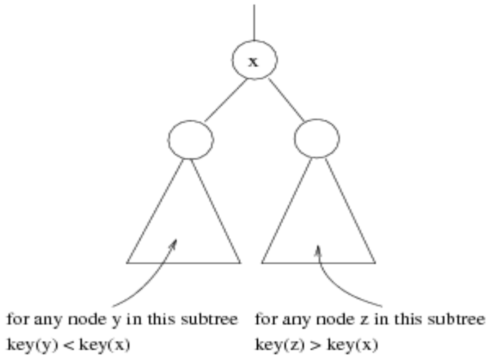
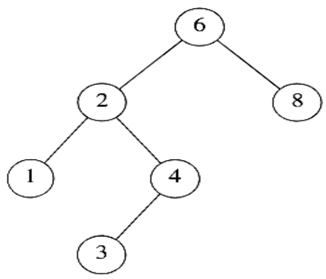
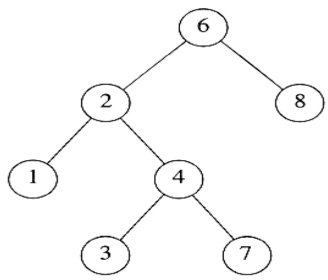


Figure 10.18 Binary Search Tree

4.3. **Example** of binary search tree :



Binary Search Tree



Not a Binary Search Tree

Figure 10.19 Binary Search Tree Example

4.4. **Advantages of Binary Search Tree**

- Simple
- Efficient
- Dynamic
- One of the most fundamental algorithms in Computer Science
- The method of choice in many applications

4.5. Disadvantages of Binary Search Tree

- The shape of the tree depends on the order of insertions, and it can be degenerated.
- When inserting or searching for an element, the key of each visited node has to be compared with the key of the element to be inserted/found.
- Keys may be long and the run time may increase much.

4.6. Binary search trees come in many shapes. Figure 10.20 shows two binary search trees representing the same set:

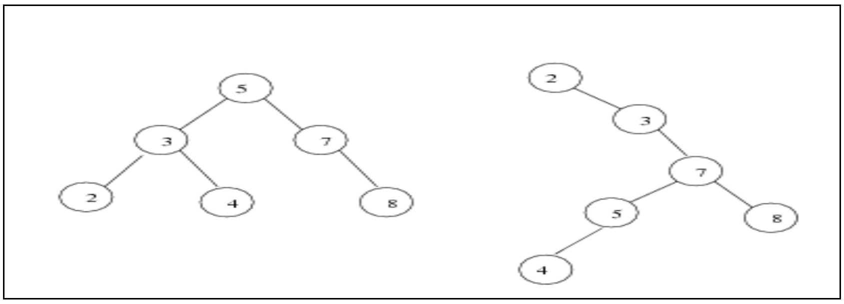


Figure 10.20 Different shapes of binary search tree.

4.7. The shape of a binary search tree determines the efficiency of its operations.

4.8. Average depth of a node is O(log N); maximum depth of a node is O(N).

4.9. The height of a binary search tree with n nodes can range from a minimum of $O(\log_2(n + 1))$ to a maximum of n

5.0 BINARY SEARCH TREE IMPLEMENTATION

5.1. **Pointer-based** ADT Binary Tree

- Elements in a binary tree is represented by using nodes.
- Nodes store the information in a tree.

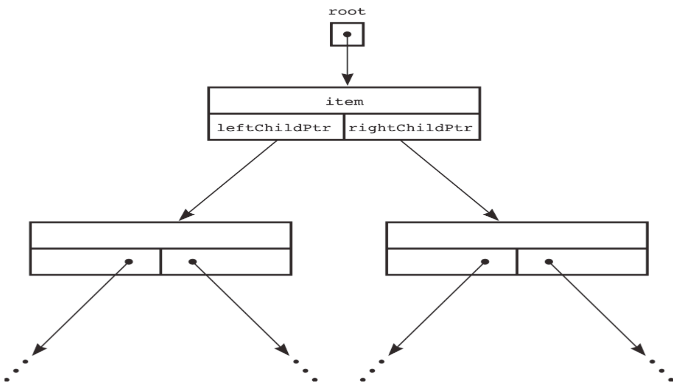
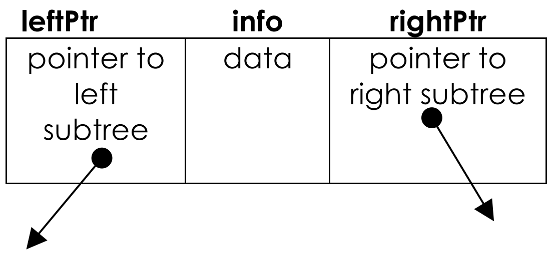


Figure 10.21 Pointer- based Binary Tree

5.2. **Node Representation.**



- Each node in the tree must contain at least 3 fields containing:
  item to be store in the tree, **info**
  pointer to left subtree, **leftPtr**
  pointer to right subtree, **rightPtr**

5.3. **Node Implementation.**

```
//Program 10.1

typedef char ItemType;
struct TreeNode
{  ItemType info;
   TreeNode *leftPtr;
        TreeNode * rightPtr;
};
```

- The node store char value; **info**, pointer to left subtree; **leftPtr** and pointer to right subtree; **rightPtr**.

5.4. **Tree Implementation :** Declaration of class Tree

```
//Program 10.2

class TreeType {
public:
  TreeType();
  ~TreeType();
  bool IsEmpty()const;
  int NumberOfNodes()const;
  void RetrieveItem(ItemType&,bool& found);
  void InsertItem(ItemType);
  void DeleteItem(ItemType);
  void PrintTree() const;
private:
  TreeNode * root;
};
```

5.5. The tree declaration above, declare the binary search tree using class **TreeType.**

5.6. The tree can be accessed using pointer variable **root**, which is a pointer to root of the tree.

5.7. Among the tree operations in the class:

1. Initialize tree , using constructor.
2. Destroy tree, destructor.
3. check for empty tree, **IsEmpty()**.
4. Count number of nodes in the tree, **NumberOfNodes()**.
5. Search item in the tree, **RetrieveItem()**.
6. Insert item into a tree, **InsertItem()**.
7. Delete item from tree, **DeleteItem()**.
8. Print all items in the tree, **PrintTree()** (Inorder traversal).

5.8. **Tree Constructor**

```
//Program 10.3

TreeType::TreeType()
{
        root = NULL;
}
```

- The constructor creates an empty tree by initializing root to null value.

5.9. **Tree Destructor**

```
//Program 10.4

TreeType::~TreeType()
{
        Destroy(root);
}
void Destroy(TreeNode*& tree)
{  if (tree != NULL)
   {     Destroy(tree->left) ;
                 Destroy(tree->right) ;
         delete(tree);
   }
}
```

- Destructor will destroy all the nodes in the tree. Function **Destroy()** is implemented recursively whereby the function will destroy all nodes in left subtree, followed by destroying nodes in right subtree. Lastly, the root node will be destroyed.
- Example in Figure 10.22 – Figure 10.24 : Destroying a tree using destructor.

STEP 1

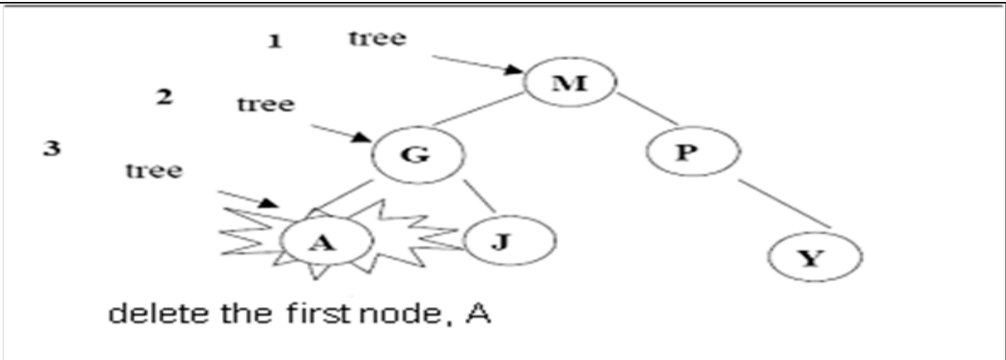


Figure 10.22 Destroying the first node using destructor.

STEP 2

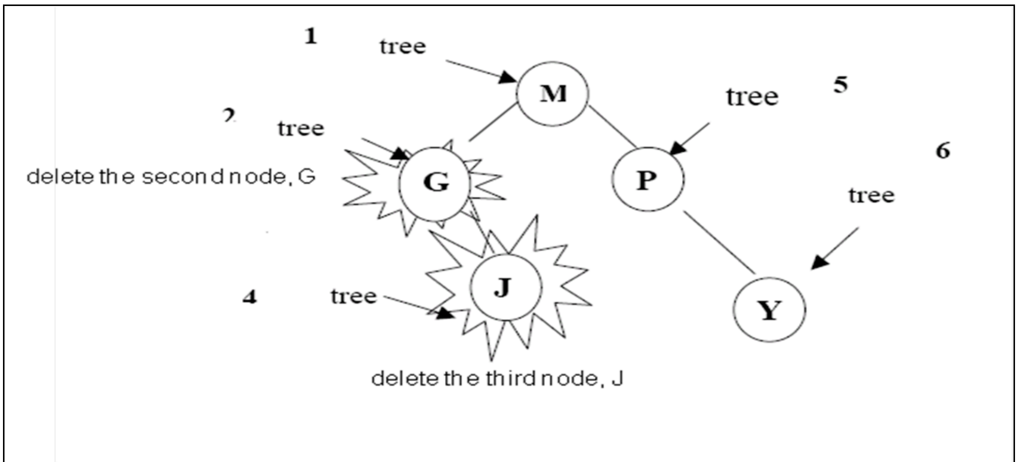


Figure 10.23 Destroying the second and third node using destructor.

STEP 3

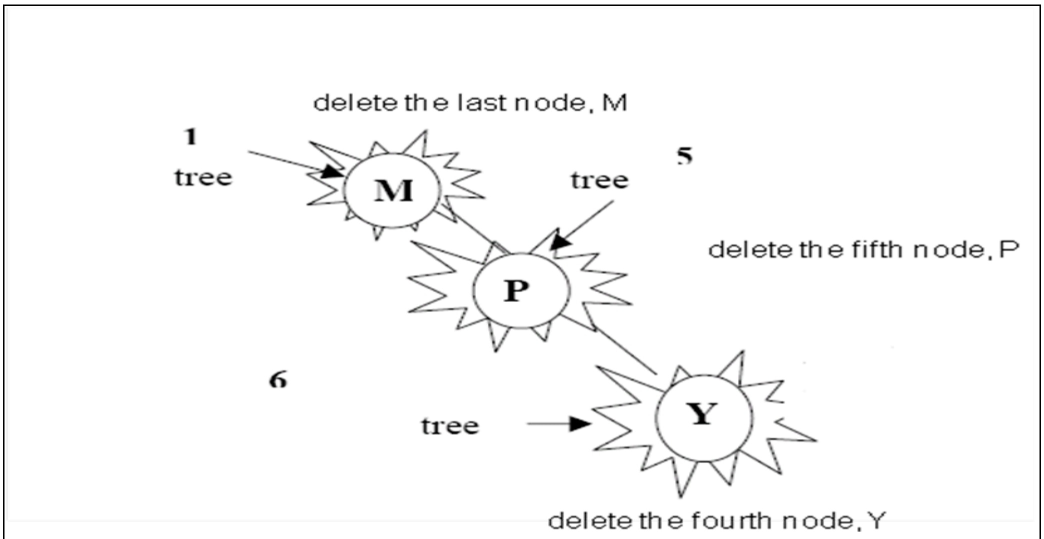


Figure 10.24 Destroying the whole tree using destructor.

5.10. IsEmpty() Function

- Binary Search Tree is empty when there is no node in the tree.
- Pointer root has **NULL** value.
- Function **IsEmpty()** below, return **True** if root is **NULL** and will return **False** if the root is not **NULL**.

```
//Program 10.5
bool IsEmpty() const
{
   if (root == NULL)
       return True; // tree is empty
     else
       return False; // tree is not empty
}
```

5.11. Insert Node to a Binary Search Tree (BST)

- The insertion operation will insert a node to a tree and the new node will become leaf node.
- Before the node can be inserted into a BST, the position of the new node must be determined. This is to ensure that after the insertion, the BST characteristics are still maintained.
- Steps to insert a new node in BST :
  1. Find the position of the new node in the tree.
  2. Allocate new memory for the new node.
  3. Set **NULL** value to left and right pointer.
  4. Assign the value to be stored in the tree.

5.12. Insert new node Implementation

```
//Program 10.6

void TreeType::InsertItem(ItemType item)
{ Insert(root, item);}

void Insert(TreeNode*& tree, ItemType item)
{   if (tree == NULL) { // base case
          tree = new TreeNode;
          tree->right = NULL;
          tree->left = NULL;
          tree->info = item;
     }
     else if (item < tree->info)
          Insert(tree->left, item);
     else
          Insert(tree->right, item);
}
```

5.13. ADT Binary Search Tree: Insertion

- Example 1 – Insert into empty and non-empty tree

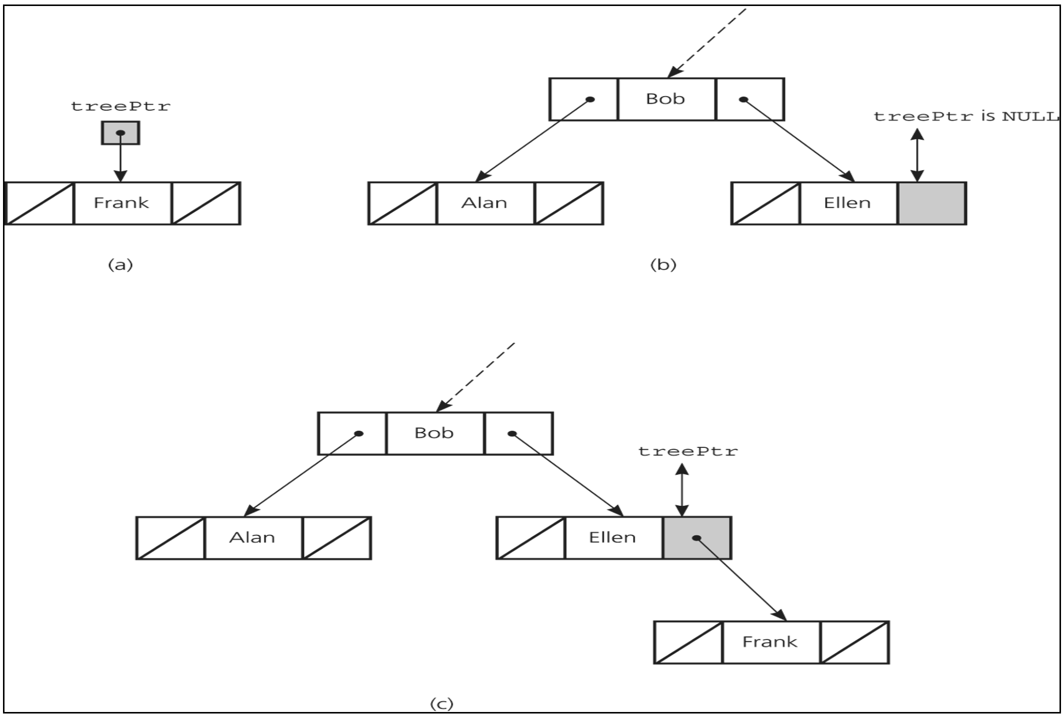


Figure 10.25 Insertion in Binary Search Tree

(a) Insert Frank into an empty tree;
(b) Insert Frank into non-empty tree, search terminates at a leaf;
(c) insert Frank at a leaf

- Example 2 -Insert 13 into **non-empty tree**
  STEPS:
  - Proceed down the tree until found the position.
  - Insert 13 at the last spot on the path traversed

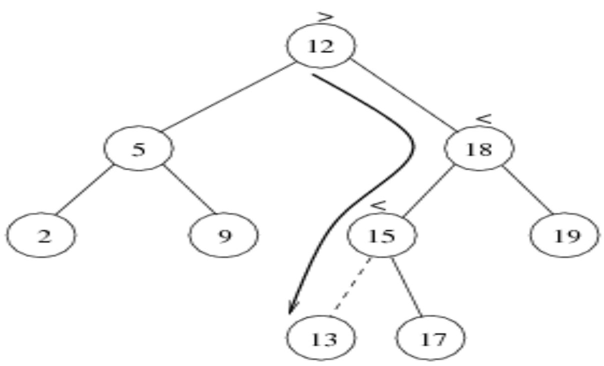


Figure 10.26 Steps to insert 13 into binary search tree.

- Example 3 - Insert 5, 10, 8, 3 , 4, 15 and 2 in a BST

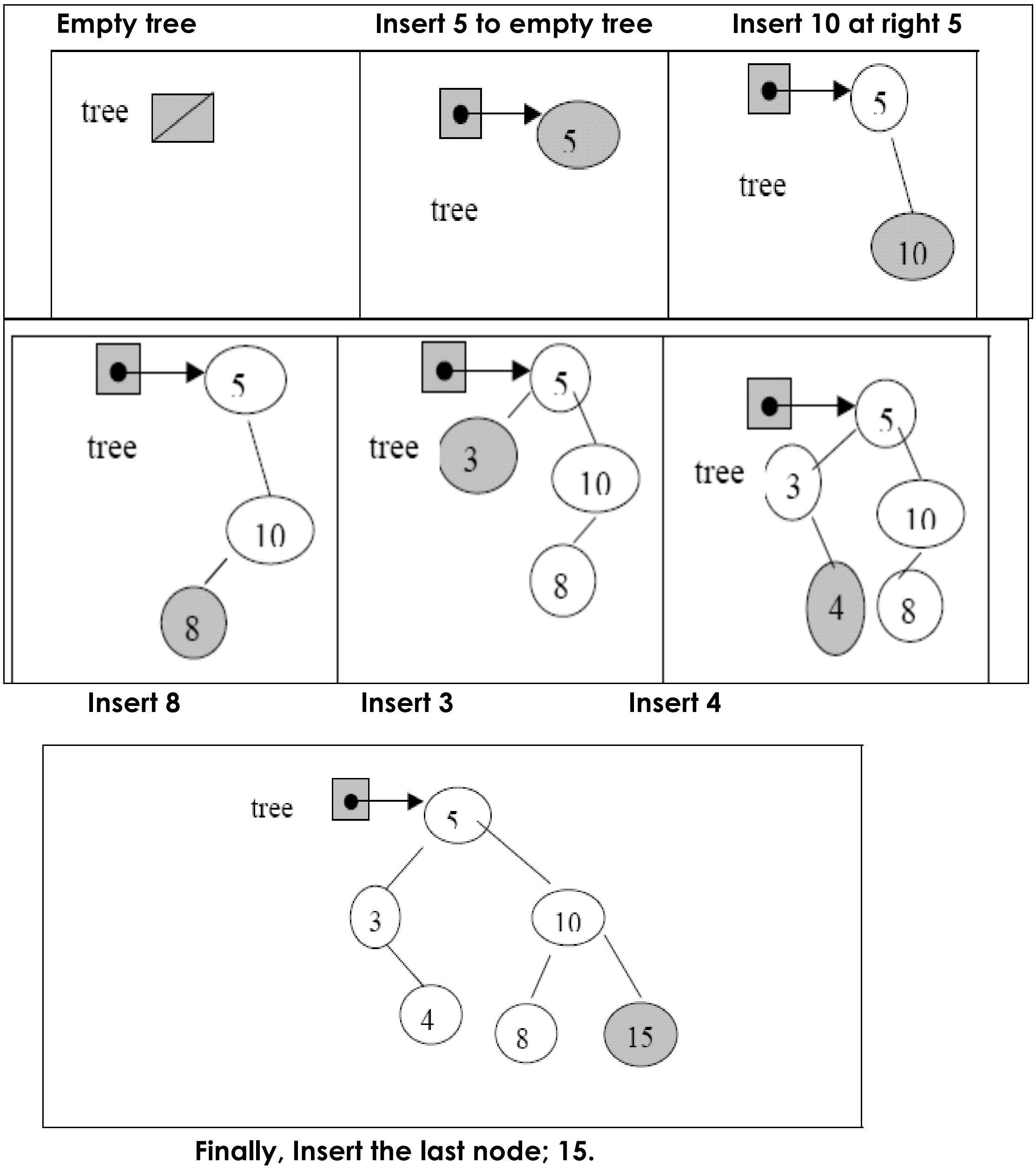


Figure 10.27 Insertion to Binary Search Tree

5.14. Sequence of input being used to create a binary search tree is an important factor for creating a balance tree.

- Figure 10.27 is a balanced tree from input : 5, 10, 8, 3, 4, 15, 2.

5.15. Time Complexity - O(height of the tree)

5.16. Searching a key in unbalanced tree take a longer time, O(n) for worse case. Therefore, creating an unbalanced tree must be avoided.

5.17. **Searching** a key in BST

- Searching is an operation to traverse a tree in order to find a key and determine whether the key exists in the tree or not.

- Searching from a tree starts at the root, and will recursively search in the left subtree or right subtree until the key is found or until reach a leaf.

5.18. Searching Process

- From the figure, if we are searching for 15, then we are done at the root.
- 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.

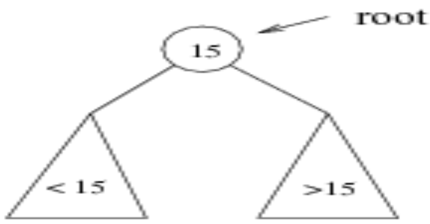


Figure 10.28 Binary Search Tree

5.19. **Searching implementatation**: **RetrieveItem()** function

```
//Program 10.7

void TreeType::RetrieveItem(ItemType &item, bool
&found)
{ Retrieve(root, item, found);  }

void Retrieve(TreeNode* tree, ItemType &item, bool
&found)
{   if (tree == NULL) { // base case
           found = FALSE;
        else if (item < tree->info)
           Retrieve(tree->left, item, found);
        else if (item > tree->info)
           Retrieve(tree-> right, item, found);
     else
           found = TRUE;
}
```

5.20. Steps to search item from a binary search tree using **Retrieve()** function.

a. **Retrieve()** function implement recursive function and call itself until the key is found, (**item==tree->info**) or until **tree** has **NULL** value. If **tree** equal **NULL**, the search key is not found.

b. If the key being searched is smaller than the value at the node being compared, then the next search will be done at the left subtree by sending the left pointer of the node pointed by **tree** to **Retrieve()** function.

Retrieve(tree->left, item, found);

c. If the key being searched is larger than the value at the node being compared, then the next search will be done at the right subtree by sending the right pointer of the node pointed by **tree** to **Retrieve()** function.

Retrieve(tree->right, item, found);

d. If the key being searched is found, **Retrieve()** function will return TRUE implying that the key is found.

e. If the key being searched is not found, **Retrieve()** function will return FALSE implying that the key is not found.

5.21. Searching Example:

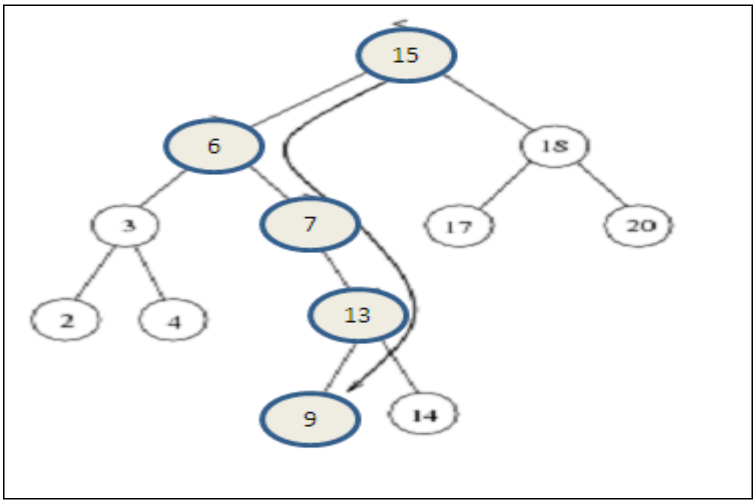


Figure 10.29 Searching item found in a tree.

5.22. Search for 9

1. Compare 9 with the value at root, 15, go to left subtree;
2. Compare 9 with 6, go to right subtree.
3. Compare 9 with 7, go to right subtree.
4. Compare 9 with 13, go to left subtree;
5. Compare 9 with 9 , **found** is **TRUE**.

5.23. Search for 5 – not found

1. Compare 5 with the value at root, 15, go to left subtree;
2. Compare 5 with 6, go to left subtree.
3. Compare 5 with 3, go to right subtree.
4. Compare 5 with 4, go to right subtree;
5. **tree** become **NULL** , found is **FALSE**.

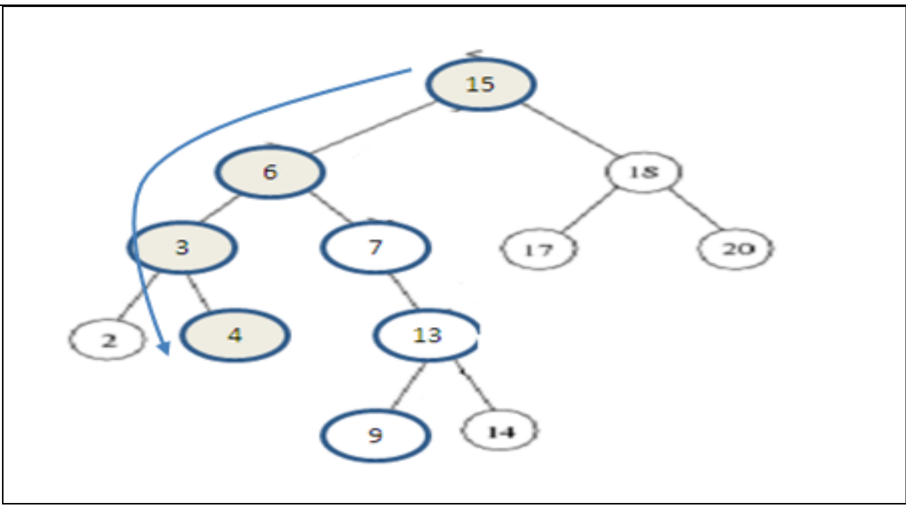


Figure 10.30 Searching item not in a tree

5.24. **Delete node** operation.

- When delete a node from binary search tree, we need to take care of the children of the deleted node. This has to be done in order to ensure the property of the search tree is maintained.
- Three possible cases for deleting the item in node *N*
    1. *N* is a leaf : Set the pointer in *N*'s parent to NULL
    2. *N* has only one child : Let *N*'s parent adopt *N*'s child
    3. *N* has two children :
        - Locate another node *M* that is easier to delete.
            - *M* is the leftmost node in *N*'s right subtree
            - *M* will have no more than one child
            - *M*'s search key is called the inorder successor of *N*'s search key
        - Copy the item that is in *M* to *N*
        - Remove the node *M* from the tree

5.25. **Delete leaf node**

- When delete leaf node, *N*, set the pointer in *N*'s parent to **NULL** and delete it immediately
- Example : Delete leaf Node: Z

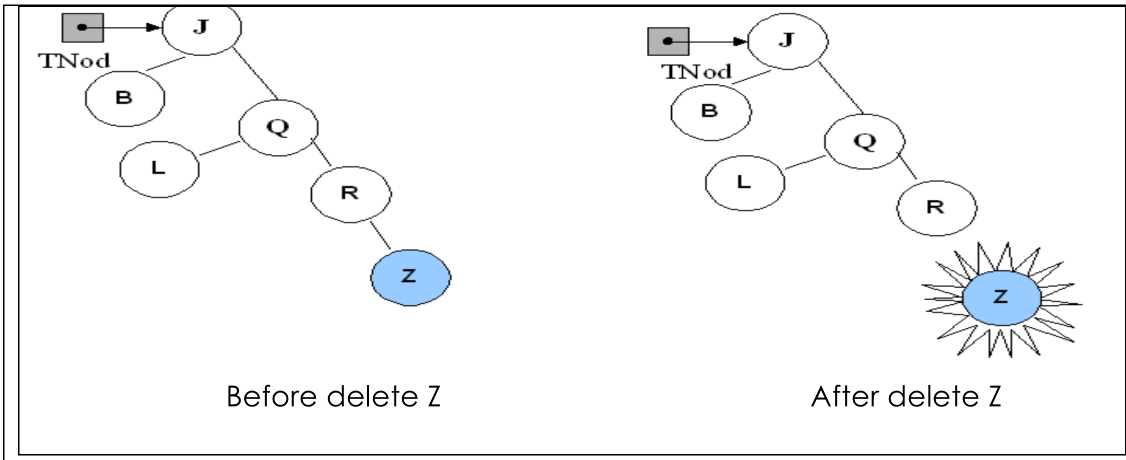


Figure 10.31 Delete leaf node.

5.26. **Delete node with one child**

- When delete the node that has one child, adjust a pointer from the parent to bypass that node.
- Example: Delete node R.
    - o Adjust a pointer from the parent to bypass that node.

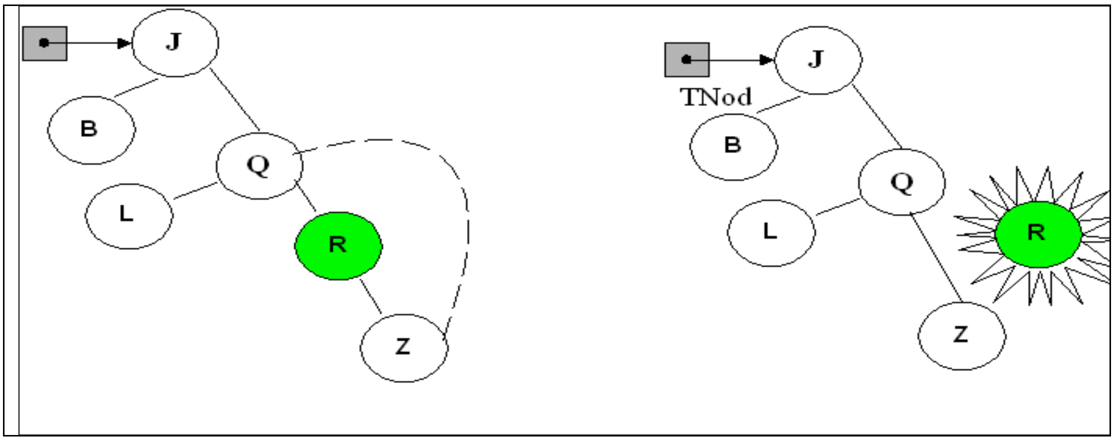
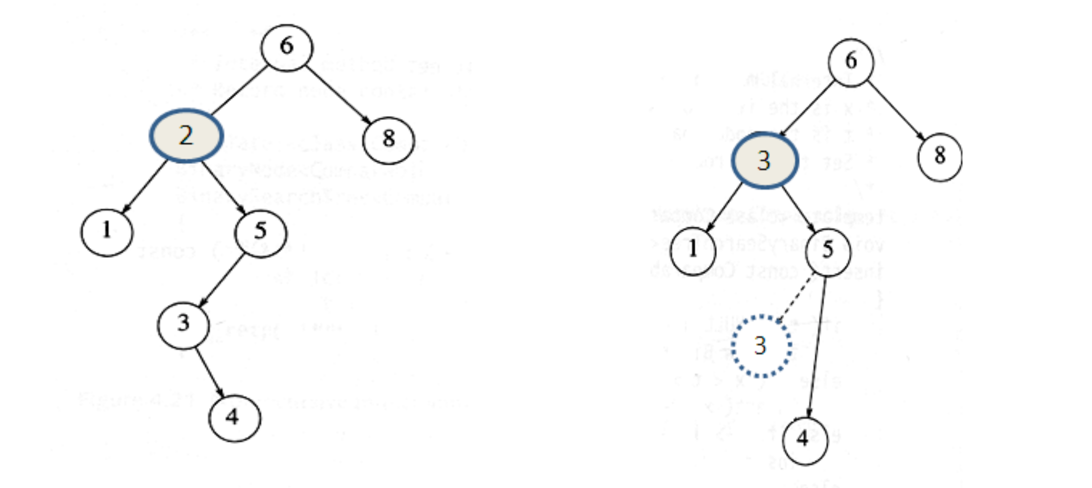


Figure 10.32 Delete node with a child.

5.27. Delete node with 2 children

- Replace the key of that node with the minimum element at the right subtree
- Delete the node with minimum element
    - o 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.
    - o Figure 10.33 shows the deletion of a node with value 2 that has 2 children.
        - ▪ To replace 2, choose minimum element from right subtree of 2.
        - ▪ The minimum value is 3 and the node has one child.

- Replace 2 with value 3, and let parent of 3, node 5 to point to child of 3, node 4.



Before delete 2

After delete 2 – replace with value 3 and delete 3.

Figure 10.33 Delete node with 2 children.

- Figure 10.34 shows the deletion of a node with value Q that has 2 children.

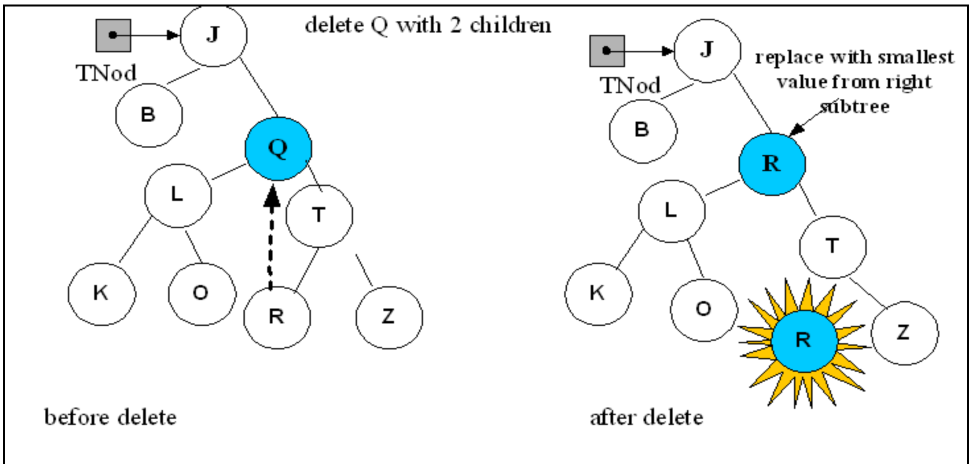


Figure 10.34 Delete node with 2 children

5.28. Delete Node Implementation.

```
//Program 10.8
// Get the sucessor of the deleted node
void GetSuccessor(TreeNode* &tree, ItemType& data )
{
  while(tree->left!= NULL)
```

```
      tree = tree->left;
   data = tree->info;
}
// Delete node function
void DeleteNode(TreeNode*& tree)
{
   ItemType data;
   TreeNode* tempPtr;
   // delete node with one child at right
   tempPtr = tree;
   if(tree->left == NULL) { //right child
      tree = tree->right;
      delete tempPtr;
   }
   // delete node with one child at left
   else if(tree->right == NULL) { // left child
      tree = tree->left;
   delete tempPtr;
   }
   // delete node with two children
   else {
   // get the successor
   GetSuccessor(tree->right, data);
   //copy successor node value to the deleted
   // node, become new value of the node that
   // should be deleted
   tree->info = data;
         // delete successor node
    Delete(tree->left, data);
   }
}
void Delete(TreeNode*& tree, ItemType item)
{ if (tree == NULL)
      cout << "\n Node not found";
   else  if(item < tree->info)
        // search the node on left subtree
        Delete(tree->left, item);
   else if(item > tree->info)
        // search the node on right subtree
        Delete(tree->right, item);
   else
        // the node is found and going to be
        // deleted
        DeleteNode(tree);
}
```

5.29. **Print values** in BST

- Function **PrintTree()** print all values in BST using inorder traversal. **Print()**

function will be called recursively, starting from left subtree, root and right subtree.

```
//Program 10.9

void Print(TreeNode* tree)
{
  if(tree != NULL) {
     Print(tree->left);
     cout << tree->info;
    Print(tree->right);
  }
}
void TreeType::PrintTree() const
{  Print(root);  }
```

- Inorder traversal of BST
- Print out all the keys in sorted order
- Value in Figure 10.35 will be printed using Inorder traversal approach:
  2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

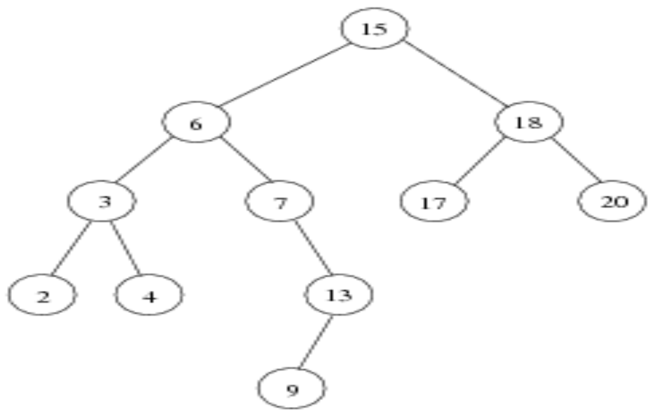


Figure 10.35 Printing from binary search tree.

5.30. The **Efficiency of Binary Search Tree Operations**

- The maximum number of comparisons required by any BST operation is the number of nodes along the longest path from root to a leaf—that is, the tree's height
- The order in which insertion and deletion operations are performed on a binary search tree affects its height
- Insertion in random order produces a binary search tree that has near-minimum height

5.31. Tree implementation (**main()** program)

```
//Program 10.10

main()
{
  TreeType tree1;  // declare tree object
  //Print the content of the tree
  if (tree1.IsEmpty())
     cout << "\nEmpty Tree " ;
  else
  {  cout << "\nContent of Tree:   " ;
     tree1.PrintTree();
  {
  // insert item to tree
  tree1.InsertItem('N');
  tree1.InsertItem(‘J');
  tree1.InsertItem('I');
  cout << "\nContent of Tree:   " ;
  tree1.PrintTree();
  tree1.DeleteItem('M');
}
```

5.32. Comparison of Efficiency of Binary Search Tree Operations

Operation	Average Case	Worse Case
Retrieval	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)
Traversal	O(n)	O(n)