Tree Structures

• Definitions:
  o A tree is a connected acyclic graph.

  o A disconnected acyclic graph is called a forest

  o A tree is a connected digraph with these properties:
    ▪ There is exactly one node (Root) with in-degree=0
    ▪ All other nodes have in-degree=1
    ▪ A leaf is a node with out-degree=0
    ▪ There is exactly one path from the root to any leaf

  o The degree of a tree is the maximum out-degree of the nodes in the tree.

  o If (X,Y) is a path:
    X is an ancestor of Y, and
    Y is a descendant of X.
• Level of a node:

Level

0 or 1

1 or 2

2 or 3

3 or 4

• Height or depth:
  
  o The depth of a node is the number of edges from the root to the node.
  
  o The root node has depth zero
  
  o The height of a node is the number of edges from the node to the deepest leaf.
  
  o The height of a tree is a height of the root.
  
  o The height of the root is the height of the tree
  
  o Leaf nodes have height zero
  
  o A tree with only a single node (hence both a root and leaf) has depth and height zero.
  
  o An empty tree (tree with no nodes) has depth and height −1.
  
  o It is the **maximum level** of any node in the tree.
Example:

- Children, Parents, and Siblings

- Subtree

- Properties:
  (1) for a tree $T = (V, E)$, where $n = |V|$ and $e = |E|$, we have

$$e = n - 1$$
• Binary Trees
  o Definitions:
    ▪ It is a tree whose \textbf{degree is} \leq 2
    ▪ The two children are called \textbf{left and right} children
  o Properties:
    ▪ \textbf{Strictly binary:}
      • Each node has either two children or 0
    ▪ \textbf{Full Binary} tree:
      • A tree is a full binary tree of depth \( h \) iff each node of level \( h \) is a leaf and each intermediate node has left and right children.
    ▪ \textbf{Complete Binary} tree:
      • Every intermediate node in levels between 0 and \( h-2 \) have 2 children
      • Every node in level \( h-1 \) has either 2 children or 1 child. If there is one child, then it is a left child.
- **Balanced Binary Tree:**
  - A tree is a balanced (or height balanced) BT iff for each node X in T, the depth of the left and right subtrees of X differ by at most 1.

- **Lemma 1:**
  - The maximum number of nodes on level i of a binary tree is $2^i$ (starting from level 0).
  - The maximum number of nodes in a binary tree of depth k is: $2^{k+1} - 1$, k>0 (starting from level 0).

- **Lemma 2:**
  - For any non empty binary tree, T, if $n_0$ is the number of leaves and $n_2$ is the number of nodes of degree 2, then
    $$n_0 = n_2 + 1$$
  - Proof:
    - The total number of nodes in a BT T is: $n = n_0 + n_1 + n_2$
      $n_i$ is the number of nodes of degree i (i children) for i=0, 1, and 2.
    - We have $e = n - 1$ from property 1 where e is the number of links in T.
    - The number of links e can also be computed as follows:
      - $n_0$ contribute by 0 links
      - $n_1$ contribute by $n_1 \times 1 = n_1$ links
      - $n_2$ contribute by $n_2 \times 2 = 2n_2$ links
    - Therefore,
      $$e = n_1 + 2n_2 = n - 1 = n_0 + n_1 + n_2 - 1$$
      $$\Rightarrow \quad n_0 = n_2 + 1$$
• **Representations:**
  
  - Sequential
  - Linked-list

• **Sequential representation:**
  
  - For a complete tree of \( n \) nodes:

    (1) The parent of a node \( i \) is:

    \[
    \text{Parent}(i) = \begin{cases} 
    \frac{i}{2} & \text{if } i \neq 1 \\
    \text{No parent} & \text{if } i = 1 (i \text{ is the root})
    \end{cases}
    \]

    (2) The leftchild of a node \( i \) is:

    \[
    \text{Leftchild}(i) = \begin{cases} 
    2i & \text{if } 2i \leq n \\
    \text{No leftchild} & \text{if } 2i > n
    \end{cases}
    \]

    (3) The rightchild of a node \( i \) is:

    \[
    \text{Rightchild}(i) = \begin{cases} 
    2i + 1 & \text{if } 2i + 1 \leq n \\
    \text{No Rightchild} & \text{if } 2i + 1 > n
    \end{cases}
    \]

• **Linked-list representation:**

[Diagram of a tree with node labels and connections]
Binary Tree Traversals

- There are three traversals:
  - Inorder: LNR
  - Preorder: NLR
  - Postorder: LRN

- Inorder Traversal: LNR

- Procedure:

  Procedure LNR (t:tree);
  Begin
      If t=null
      then return
      else Begin
          LNR(t->left);
          visit(t-data);
          LNR(t->right);
      end;

- Complexity:

  \[ T(n) = O(n) \text{ where } n \text{ is the number of nodes in } T. \]
Example:

LNR: 4-8-2-9-5-10-1-13-11-14-6-3-7-12

NLR: 1-2-4-8-5-9-10-3-6-11-13-14-7-12

LRN: 8-4-9-10-5-2-13-14-11-6-12-7-3-1
Binary Search Tree ADT

- **Objective:**
  - Insertion, deletion, and Find take $O(\log(n))$ where $n$ is the number of elements in the list.

- **Definition:**
  - Let us assume that every node in a binary search tree (BST) is assigned a key value $X$. For every node $X$ in a BST, the keys in the left subtree of the node containing $X$ are smaller than $X$ and the keys in the right subtree of the node containing $X$ are greater than $X.$

```
      X
     /
    /
   /
  /
 /
```

- **Example:**

```
      7
    /  \
  3    10
 / \
1   5   9
 / \
2   4   8
     \
     6
        \
     12
        \
```

---

CSci 1112 - Algorithms and Data Structures, A. Bellaachia

Page 9
• **Operations:**
  - Search or Find
  - Find_min
  - Find_max
  - Insert
  - Delete

• **Search:**

  • function:
    
    Node Search(Node T; int x);
    Begin
      If ( T == null)
      then return(null);
    else Begin
      If (x < T.data)
      then return(Search(T.left));
      else if (x > T.data)
        then return(Search(T.right));
      else return(T);
    End;
  End;

  • Complexity:
    
    O(h) where h is the depth of the tree.
• **Insertion in a BST:**
  o There are three steps:
    ▪ create a new node for the element to be inserted
    ▪ Search or Find the location at which the new node will be inserted
    ▪ Insert the new node
  o Procedure `Insert(Node Root; int x)`
    
    ```cpp
    Begin /* The element to be inserted is x */
    /* Create new node */
    t = create_node(); /* Allocate space for x */
    t.leftChild = null; t.rightChild = null; t.data = x;
    /* Search for the insertion location */
    p = Root; q = nil;
    While (p!=null) do Begin
      q = p;
      if p.data > x
        then p = p.left;
      else p = p.right;
    End;
    /* Insert the new element */
    If (q == null) /* Empty tree */
      then Root = t;
    else Begin
      if q.data > x
        then q.left = t;
      else q.right = t;
    End;
    End;
    ```
• Complexity:
  \[ O(h) \text{ where } h \text{ is the depth of the tree.} \]

• Example:
• Deletion in a BST:
  o There are three cases:
    ▪ Node to be deleted has no children (Leaf).
    ▪ Node to be deleted has one child.
    ▪ Node to be deleted has two children (complicated).

  o Case 1: Node to be deleted has no children (Leaf):

  ![Deletion in a BST](image)

  Deletion steps:
  ```
  //Delete node with value 12 in a BST with root R
  //T is the parent of the node that contains 12
  T = findParent(R, 12)
  T.rightChild = null;
  ```
Case 2: node to be deleted has one child

Example:

- Deletion steps:
  //Delete node with value 1 in a BST with root R
  //T is the parent of the node that contains 1
  T = findParent(R, 1)
  //Delete the node
  T = T.rightChild;  //Since the left child is null.
• Case 3: node to be deleted has two children

X \quad \text{Node to be deleted}

Left Subtree \quad \text{Right Subtree}

- X must be replaced by either its:
  - predecessor (Max in the left subtree)
  - successor (Min in the right subtree)

- Example 1:

```
R
7
3
T
2
S
4
5
6
9
8
10
11

\text{Delete}(T.\text{rightChild}, T.\text{data});
```

- Deletion steps:

//Delete node with value 5 in a BST with root R
//T is the parent of the node that contains 5
T = \text{findParent}(S, 5);
S = \text{findSuccessor}(T); //Find the min of the right subtree.
//Delete the node
T.\text{data} = S.\text{data};
\text{Delete}(T.\text{rightChild}, T.\text{data});
Tree after deleting node 5:

Example 2:

- Deletion steps:
  //Delete node with value 7 in a BST with root R
  //T is the parent of the node that contains 7
  T = findParent(S, 7);
  S = findSuccessor(T); //Find the min of the right subtree.
  //Delete the node
  T.data = S.data;
  Delete(T.rightChild, T.data);
o Tree after deleting node 7:
Procedure Delete(Node Root; int x)
Begin
    If (T ==null)  then print (“Sorry the element is not found “);
    else if (x< T.data)
        then Delete(T.leftChild,x); /* Go left */
    else if (x>T.data)
        then Delete(T.rightChild,x) /* Go Right */
    else Begin
        If (T.leftChild == null) /* only a right child or none*/
            then begin
                temp = T; T = T.rightChild; free(temp);
            end;
        else if (T.rightChild ==null) /* only a left child */
            then begin temp = T; T = T.leftChild; end;
        else begin /* Case 3: Two children. Replace with successor */
            temp = Find_min(T.rightChild);
            T.data = temp.data;
            Delete(T.rightChild,T.data)
        end;
    End;
End;
**Time Complexity:**

- If the tree is a complete binary tree with $n$ nodes, then the worst-case time is $O(\log n)$.

- If the tree is very unbalanced (i.e. the tree is a linear chain), the worst-case time is $O(n)$.

- Luckily, the expected height of a randomly built binary search tree is $O(\log n)$
  - basic operations take time $O(\log n)$ on average.
Threaded Binary Trees

- **Motivations:**
  - To do traversal in languages that do not support recursion
  - Non-recursive traversals

- In a binary tree of n nodes there are 2n links out of which n+1 are null links. In case of full tree of depth k, we have \( n = 2^{k+1} - 1 \). The number of leaves is \( 2^k = \frac{n+1}{2} \). Therefore, the number of null links is: \( 2 \times \frac{n+1}{2} = n+1 \).

- **Objective:**
  - Make use of the null links (by A.J. Perlis & C. Thornton).
  - Replace null links by pointers, called threads, to other nodes in the tree.

- **Threads setup:**
  - If \( p->right == \text{null} \)
    then \( p->right = \) the node which would be printed after \( p \) (inorder successor of \( p \)) when traversing the tree in inorder.
  - If \( p->left == \text{null} \)
    then \( p->left = \) the node which would be printed before \( p \) (inorder predecessor of \( p \)) when traversing the tree in inorder.
• Example:

```
         A
        / \
       B   C
      / \  /  \
     D E F G
    / \   /   /
   H   I  F   G
```

LNR: H D I B E A F C G

• Implementation:
  o How to distinguish between threads and normal pointers?

<table>
<thead>
<tr>
<th>Leftthread</th>
<th>Leftchild</th>
<th>Rightchild</th>
<th>Rightthread</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Data</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

• Application:
  o Perform a non-recursive inorder traversal without a stack to simulate recursion.
• Code Example

public class BinaryTreeNode {

   private int key;
   private BinaryTreeNode leftChild;
   private BinaryTreeNode rightChild;

   public BinaryTreeNode(){
      key = 0;
      leftChild = null;
      rightChild = null;
   }
   public BinaryTreeNode(int d, BinaryTreeNode left, BinaryTreeNode right){
      key = d;
      leftChild = left;
      rightChild = right;
   }
   public int getKey(){
      return (key);
   }
   public BinaryTreeNode getLeftChild(){
      return (leftChild);
   }
   public BinaryTreeNode getRightChild(){
      return (rightChild);
   }
   public void setLeftChild(BinaryTreeNode node){
      leftChild = node;
   }
   public void setRightChild(BinaryTreeNode node){
      rightChild = node;
   }
}

public class BinarySearchTree {

   private BinaryTreeNode root;

   public BinarySearchTree(){
      this.root = null;
   }
   public BinaryTreeNode getRoot(){
      return (root);
   }

   private void findPosition(BinaryTreeNode node, BinaryTreeNode start){
      int sKey = start.getKey();
      if (sKey>node.getKey()){
if (start.getLeftChild() == null)
    start.setLeftChild(node);
else{
    findPosition(node, start.getLeftChild());
}
}
else{
    if (start.getRightChild() == null)
        start.setRightChild(node);
    else{
        findPosition(node, start.getRightChild());
    }
}

public void insertNode(BinaryTreeNode node){
    if (root == null){
        root = node;
    } else{
        findPosition(node, this.root);
    }
}

private boolean findElement(BinaryTreeNode node, int x){
    if (node == null)
        return false;
    if (x == node.getKey())
        return true;
    else if (x < node.getKey())
        return findElement(node.getLeftChild(), x);
    else
        return findElement(node.getRightChild(), x);
}

public int countLeaves(BinaryTreeNode node) {
    if (node == null)
        return 0;
    else if (node.getLeftChild() == null && node.getRightChild() == null)
        return 1;
    else
        return countLeaves(node.getLeftChild()) +
        countLeaves(node.getRightChild());
}

public int computeDepth(BinaryTreeNode node){
    if (node == null)
        return 0;

return (1 + Math.min(computeDepth(node.getLeftChild()),
    computeDepth(node.getRightChild())));
}

public void inorderPrint(BinaryTreeNode node){
}

public void preorderPrint(BinaryTreeNode node){
}

public int countNodes(BinaryTreeNode node){
}

public int findMin(BinaryTreeNode node){
}

public int findMax(BinaryTreeNode node){
}

}

o Programming Assignment:
  – Design and implement the missing operations in the Binary Search Tree ADT:
    - findMin
    - findMax
    - countNodes
    - inorderPrint
    - preorderPrint

  – Test your implementation.