Chapter 5
Trees as Data Structures

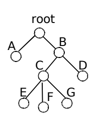
5.1 Binary Trees

5.1 Trees

Trees are structures that have *nodes* (vertices) and *edges*(links).

There is special node called the *root*. The root is at the top of the tree. The *children* of a node are the nodes that are connected to that node by an edge.

The nodes with no children are called *leaf* nodes or *terminal* nodes. The nodes that have one or more children are called *non-terminal* or *non-leaf* or internal nodes.



5.1 Trees

Trees are defined recursively as follows:

2. Any structure with just one node is a tree.
3. A structure that contains a node at the root and all its children are trees also is a tree.
4. No other structures are trees.

5.1 Properties of Trees

Property 1: The number of edges in any tree with n vertices is n-1.

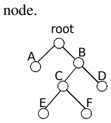
Property 2: There is only one path between any two nodes of a tree.

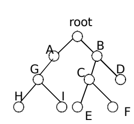
Property 3: Removal of any node results in disconnected pieces of the tree.

Property 4: A tree has maximum possible number of edges.

5.1 Binary Trees

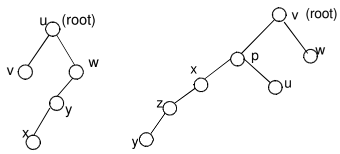
In general each node in a tree may have any number of children, but a tree in which every node has at most two children is called a *binary tree.* We refer the two children as *left child* and *right child* of a node.





The *depth* or *height* of a tree is the number of levels in the tree. The root is at level 0.

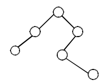
5.1 Examples



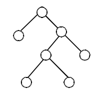
Find the depth, the leaf nodes, the non leaf nodes of the trees above.

5.1 Fully Binary Trees

A fully Binary tree is binary tree in which every node has exactly 0 or 2 children.



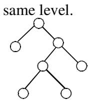
Not a fully binary tree



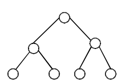
A fully binary tree

5.1 Complete Binary Trees

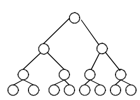
Complete Binary tree is a fully binary tree in which every non-leaf node has exactly 2 children and all leaves are at the same level.



Not a complete binary tree



A complete binary tree



A complete binary tree

Note: Fully binary trees and binary trees can be obtained from deleting one or more nodes of a complete binary tree.

5.1 Relationship Between Height and the Nodes

Let h be the height and n be the number of nodes in a complete binary tree

Number of nodes at level 0 is $1 = 2^0$
Number of nodes at level 1 is $2 = 2^1$
Number of nodes at level 2 is $4 = 2^2$
etc
Number of nodes at level h-1 is $= 2^{h-1}$

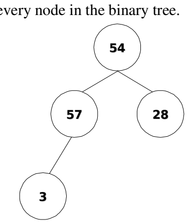
Total number of nodes n is : $2^0+2^1+2^2+ \ldots +2^{h-1} = 2^h-1$

That is , for complete binary trees, $n = 2^h - 1$
or $h = \log_2(n+1)$
How about fully binary trees and binary trees?

5.2 Implementation of Binary Trees

5.2 Implementation of BT

Data is stored at every node in the binary tree.

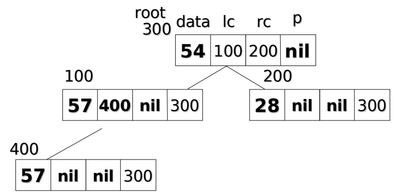


5.2 Implementation of BT

We will use the pointer implementation for binary trees.

```
class BTNode<T>{
   T elem;
   BTNode<T> leftChild;
   BTNode<T> rightChild;
   BTNode<T> parent;

   public BTNode(T elem);
   public BTNode(T elem, Node lc, Node rc, Node p);
   public BTNode rightChild();
   public BTNode leftChild();
   public BTNode parent();
   public T elem();
   public boolean equals(BTNode<T> n);
   public int numChildren();
   public String toString();
}
```


5.2 Implementation of BT

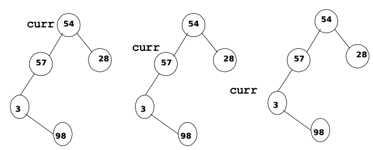


5.2 Operations on a BT

```
class BinaryTree<T>{
    BTNode<T> root;

  public BinaryTree( );
  public BinaryTree (T rootElem);
  public int depth();
  public void insert(T item);
  public int count();
  public  Iterator iteratorIrder()
  public String   pathToRoot(<BTNode> n)
  public  Iterator   iteratorPostorder()
  public  Iterator    iteratorpreorder()
  public Iterator     iteratorLevelorder()
  public BTNode<T> search(T item)
  public boolean isLeaf(BTNode<T>  n)
  public boolean isRoot(BTNode<T>  n)
  public String toString() -- gives the inorder traversal.
}
```


5.2 Insert into BT

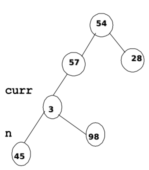
We will always insert the new node as the left child of leftmost node in the tree! So, we will need to find the leftmost leaf.



We will start at the root and keep going left until we find a node that has no left child. To move to the left child, we will use curr = leftChild(curr); To check if a node is left most , we can ask if leftChild() is nil.

5.2 Insert into a BT

If the key to insert is 45, then the 45 will be added as shown.

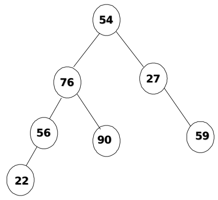


5.2 Insert into BT

```
public void insert(T item){
   BTNode<T> n, curr;
   n  = new BTNode(item);
   if ( root == null) root = n;
   else{
        curr = root;
        while ( leftChild(curr) != null)
          curr = leftChild(curr);
        setLeftChild(curr) = n;
        setParent(n) = curr;
   }
}
```

Time Complexity = ?????

5.2 pathToRoot



Path from 59 to root: 59 → 27 → 54

Path from 22 to root: 22 → 56 → 76 → 54

5.2 PrintPathToRoot

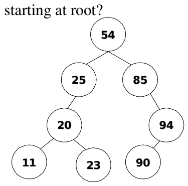
```
public  String pathToRoot(start:BTNode<T>){
   BTNode<T> curr;
   String result = “”
   curr = start;
   while (curr != null){
        result = result + curr.toString();
        toAdd = parent(toAdd);
   }
}
```

Time Complexity = O(d)

5.3 Traversal of Binary Trees

5.3 Traversal

How do we travel (visit) every node in the tree systematically, starting at root?

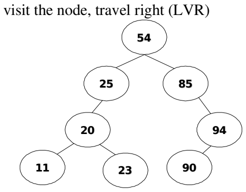


What if we wanted to print data at every node in the tree above?

5.3 Inorder Traversal

In order traversal is a traversal technique that travels each node in the following order:

travel left, visit the node, travel right (LVR)



11 20 23 25 54 85 90 94

5.3 InOrder

```
public void inorder(BTNode<T> current){
  if (current != null){
     inorder(leftChild(current));
     process(current));
     inorder(rightChild(current));
  }
}

Time complexity = ???
```

5.3 Inorder Iterator

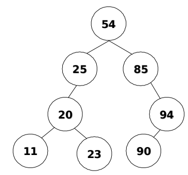
```
public Iterator iteratorInorder(){
   ArrayADT<T> arr = new ArrayADT<T>();
   inorder(root, arr);
   return(arr.iterator());
}


public void inorder(BTNode<T> current, ArrayADT<T> arr){
  if (current != null){
     inorder(leftChild(current));
     arr.add(curr.getData());
     inorder(rightChild(current));
  }
}
```

Time complexity = ???

5.3 Preorder

Preorder traversal is a traversal technique that travels each node in the following order:

visit the node, travel left, travel right (VLR)



54 25 20 11 23 85 94 90

5.3 Preorder

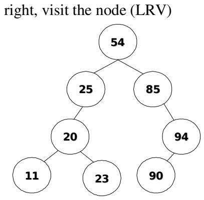
```
public void preorder(BTNode<T> current){
  if (current != null){
     process(current));
     preorder(leftChild(current));
     preorder(rightChild(current));
  }
}
```

Time Complexity = ???

5.3 Postorder

Postorder traversal is a traversal technique that travels each node in the following order:

travel left, travel right, visit the node (LRV)



11 23 20 25 90 94 85 54

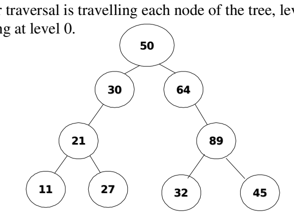
5.3 Postorder

```
public void postorder(BTNode<T> current){
  if (current != null){
     postorder(leftChild(current));
     postorder(rightChild(current));
     process(current));
  }
}
```

Time Complexity = ???

5.3 Level Order Traversal

Level order traversal is travelling each node of the tree, level by level starting at level 0.



50 30 64 21 89 11 27 32 45

How do we write the code for level order traversal? How do we move to the next level once one level is processed?

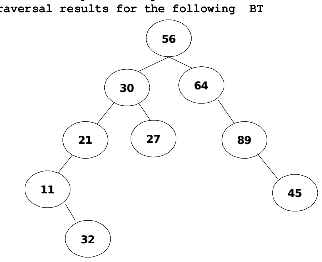
5.3 Level Order

This procedure requires the uses of queues which are discussed in another section.

```
public void levelOrderTraversal(){
    Queue<BTNode<T>>  Q = new Queue<BTNode<T>();
    BTNode<T> curr;
    Q.enque(root);

    while (!empty(Q)){
          curr = Q.deque();
          process(curr);
          if (leftChild(curr) != null)
              Q.enque(leftChild(curr));
          if (rightChild(curr) != null)
              Q.enque(rightChild(node));
    }
}

Time Complexity = ???
```

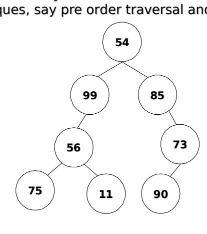

5.3 In Class Activity

```
Write the inorder, preorder, postorder and level
order traversal results for the following  BT
```



5.3 Search

How do we search for key = 73 in the tree? Use one of the traversal techniques, say pre order traversal and look for key.



54 99 56 75 11 85 $\boxed{73}$

5.3 Search

To Search, you can use any of the Traversal methods that have been presented. An example of a preOrder search has been provided.

```
public BTNode<T> search( T item) {
    Iterator preIt = iteratorPreorder();
    BTNode<T> curr;
    boolean found;
    while (preIt.hasNext() && !found){
        curr = preIt.next();
        if (curr.elem().equals(item))
          found = true;
   }

   if (found) return(curr);
   else return(null);
}
```

Time Complexity = O(n).

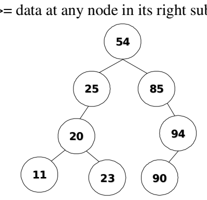
5.3 Time complexity for BT

public BinaryTree( ); -- O(1)
public BinaryTree (T rootElem); -- O(1)
public int depth(); -- O(?)
public void insert(T item); ---- O(d)
public int count(); ---- O(n)
public Iterator iteratorIrder() --- O(n)
public String pathToRoot(<BTNode> n) -- O(d)
public Iterator iteratorPostorder() -- O(n)
public Iterator iteratorpreorder() --O(n)
public Iterator iteratorLevelorder() ---O(n)
public BTNode<T> search(T item) ----O(n)
public boolean isLeaf(BTNode<T> n) -- O(1)
public boolean isRoot(BTNode<T> n) ---O(1)
public String toString() [Inorder] --- O(n)

5.4 Binary Search Trees

Binary Search Trees: Are binary trees in which the data is stored in a ordered format. This helps reduce the search time.

data at a node <= data at any node in its left subtree.
data at a node >= data at any node in its right subtree.



5.4 Implementation of BSTs

This makes search faster!
Data structure for BST node is same as the binary tree node!

```
class BTNode<T>{
   T elem;
   BTNode<T> leftChild;
   BTNode<T> rightChild;
   BTNode<T> parent;

   public BTNode(T elem);
   public BTNode(T elem, BTNode lc, BTNode rc, BTNode p);
   public BTNode rightChild();
   public BTNode leftChild();
   public BTNode parent();
   public  T elem();
   public boolean equals(BTNode n);
   pubic int numChildren();
   public String toString();
}
```


5.4 Operations on BST

insert: new to BST (the insert should done so that the resulting tree is a BST).
search: search we will see is faster!
inorder : same as binary tree, but produces special output.
preorder, postorder, levelorder: same as binary tree.
pathToRoot: same as binary tree.
successor: new to BST, returns the node that contains next element in order.
predecessor: new to BST, returns the node that contains previous element in order.
delete: new to BST (the delete should be done so that the resulting tree is a BST).

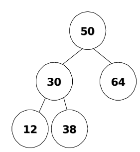
SINCE MANY OPERATIONS ARE SAME AS BT, BST WILL BE A SUBCLASS OF BT

5.4 Operations on BST

```
class BinarySearchTree<T> extends BinaryTree<T>{

    public BinarySearchTree( ) {
        super();
    }
    public BinarySearchTree (T rootElem){
       super(rootElem);
    }

    public BTNode<T> search(T item)  // will be overridden
    public void insert(T item) // will be overridden
    public BTNode<T> successor(<BTNode> n)  // is new in BST
   public BTNode<T> predecessor(<BTNode> n)  // is new in BST
   public void delete(T elem) //is new in BST
```


5.4 Insert into a BST

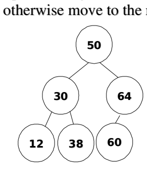
Suppose we want to insert the key= 60 in the following BST and still have a BST after the insert is done.
Where would we put the node containing 60?



What if we wanted to insert 35 after 60 is inserted?

5.4 Insert into a BST

We will find the right location by moving to the left sub tree or right sub tree at each node starting at root.
At each node we will compare the key with the data at the node, if the key is less, move to left, otherwise move to the right. **When do we stop?**



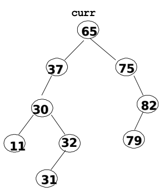
Now insert 35 in the tree above.

5.4 Insert into a BST – In class Activity

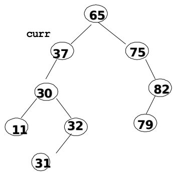
Write code for inserting an element into BST such that the resulting tree is also a BST

5.4 Search in a BST

How do we search for a key = 32 in the BST below. Again, we will move to left or right sub tree depending on whether the key is less than or greater the element at current node.

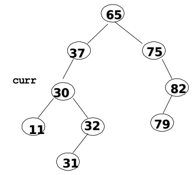


`Since 32 < 65`
`move left`

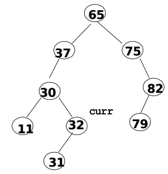


`Since 32 < 37`
`move left`

5.4 Search in a BST



`Since 32 > 30`
`move right`



`Since 32 = 32`
`STOP and return curr`

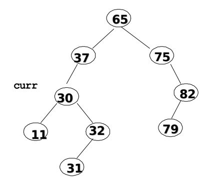
When do we stop if the key is not found?
What is the time complexity?

5.4 Search in BST

```
public BTNode<T> search(T key){
 boolean done = false;
 BTNode<T> curr = root;
   while ( curr != null && done == false){
      if (curr.elem().equals(key))
          done= true;
      else{
           if (curr.elem().compareTo(key) > 0)
                 curr = curr.leftchild();
           else
                 curr=curr.rightChild();
      }
   }
   return(curr);
}

Time = ????
```


5.4 Traversal in a BST

The algorithms for the four traversal techniques are same for BSTs as for BTs but do any of the traversals output "special" ordering for a BST?



Preorder: 65, 37, 30, 11, 32, 31, 75, 82, 79
Postorder: 11, 31, 32, 30, 37, 79, 82, 75, 65
Inorder: 11, 30, 31, 32, 37, 65, 75, 79, 82
Level Order: 65, 37, 75, 30, 82, 11, 32, 79, 31

5.4 Traversal in a BST

YES. In order of a BST gives the elements in sorted in order.

Are there any special properties of post order traversal of a BST.

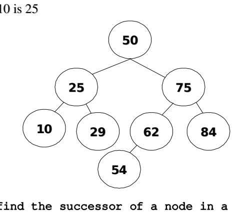
How about pre order traversal?

How about level order traversal?

5.4 Successor in a BST

Successor of a node in a BST is the node containing the next element in the sorted order of elements.

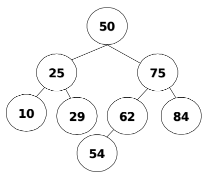
In the tree below, the successor of 29 is 50 and successor of 50 is 54 and the successor of 10 is 25



`How do we find the successor of a node in a BST?`

5.4 Successor in a BST

If a node has a right sub tree, its successor is the left-most leaf of right sub tree.
If a node has no right sub tree, then its successor is along the path to the root, until a left turn is made.



In the tree above, the successor of 29 is 50 and successor of 50 is 54 and the successor of 10 is 25

5.4 Successor in BST

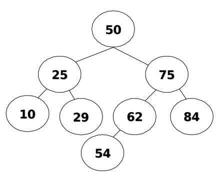
Write the code for finding the successor of a given node in BST.

What is the time complexity of the algorithm.

5.4 Predecessor in a BST

Predecessor of a node in a BST is the node containing the previous element in the sorted order of elements.

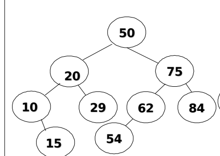
In the tree below, the predecessor of 50 is 29 and the predecessor of 54 is 50.

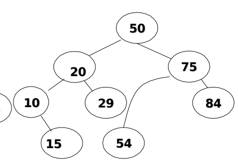


`Write the code of finding the predecessor.`

5.4 Deleting a node in BST

Deleting a leaf is easy, but deleting other nodes seems difficult.
How do we delete nodes with one child only?

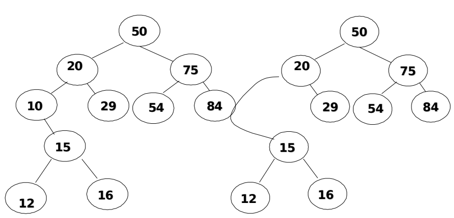




Suppose we wanted to delete the node 62, how do we proceed?

We simply re-chain the parent and the child "appropriately"

5.4 Deleting a node in BST



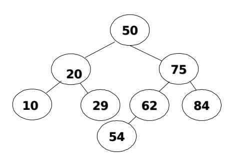
What if we wanted to delete the 10 in the tree above.

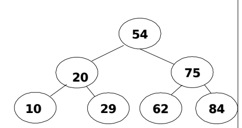
We re-chain the parent of 10 and child of 10 "appropriately"

5.4 Deleting a node in BST

What if we wanted to delete a node with two children?

We simply replace the data at the node with its successor and delete the successor.

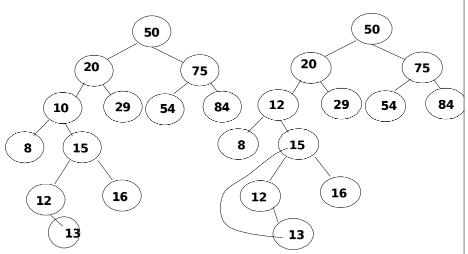




Suppose we wanted to delete the node 50, how do we proceed?

Replace 50 with 54 and delete 54.

5.4 Delete from a BST



delete 10

replace 10 with successor 12,
now delete the successor 12.

5.4 Deleting a node in BST

Is it easy to delete successor?

YES. Success has at most one child!!

So, here is the process for deletion:

If the node has no children, delete it.
If the node has one child, join the parent and the child appropriately.
If the node has two children, replace the node data with its successor's data and delete the successor.

write the code for deleting a node in a BST.

5.4 Summary of BST Operations

public BinarySearchTree( );
public BinarySearchTree (T rootElem);
public int depth();
public int count();
public boolean isLeaf(BTNode<T> n)
public boolean isRoot(BTNode<T> n)
public String pathToRoot(<BTNode> n)
public Iterator iteratorIrder()
public Iterator iteratorPostorder()
public Iterator iteratorPreorder()
public Iterator iteratorLevelorder()
public String toString() [Inorder]
public BTNode<T> search(T item)
public void insert(T item);
public BTNode<T> successor(<BTNode> n)
public BTNode<T> predecessor(<BTNode> n)
public void delete(T elem)

5.4 Time Complexity for BST

public BinarySearchTree( ); ---- O(1)
public BinarySearchTree (T rootElem); ----O(1)
public int depth(); -----O(?)
public int count(); --- O(n)
public boolean isLeaf(BTNode<T> n) --- O(1)
public boolean isRoot(BTNode<T> n) --- O(1)
public String pathToRoot(<BTNode> n) --- O(h)
public void insert(T item); -------O(h)
public Iterator iteratorIrder() ------ O(n)
public Iterator iteratorPostorder() ------ O(n)
public Iterator iteratorPreorder() -----O(n)
public Iterator iteratorLevelorder() ----- O(n)
public BTNode<T> search(T item) ----- O(h)
public BTNode<T> successor(<BTNode> n) --- O(h)
public BTNode<T> predecessor(<BTNode> n) ---- O(h)
public String toString() [Inorder] ---- O(n)
public void delete(T elem) ----- O(h)

5.4 Time Complexity of BST

We seem to have achieved a great reduction in time for searching. For BSTs, the search time is O(h), which O(logn), but for BTs the search time is O(n).

SO WHAT IS STILL WRONG WITH BSTs?

5.5 Applications of BSTs

5.5 Polish Notation

Let us consider an expression like

2-3*4+5,

Does this mean (2-3) *(4+5), or 2-(3*4+5) or 2 – (3*4) +5.

The notation we have used (in-fix notation) is bad, since it does reveal the precedence of each operator.

So, one needs to use a precedence table, and figure out that the expression actually 2 – (3*4) +5 is what is meant by the expression.

Or, we have to force the use the parenthesis in expressions.

But is there a better way to write expressions w/o parenthesis but still express the intended precedence?

5.5 Polish Notation

The answer is YES and notation is called polish notation.

The expression is written such that the operands come first and then the operator. For example,

3 5 + means 3 +5.

When evaluating polish notation, read input from left to right, when an operator is found use the previous two operands for that operation.

4 6 8 * - means

4 (6*8) – which means 4- (6*8)

5.5 Polish Notation

So the expressions we have considered earlier look different in Polish notation but they all look the same in regular in-fix notation.

2 3 – 4 5 + * means (2-3) *(4+5)

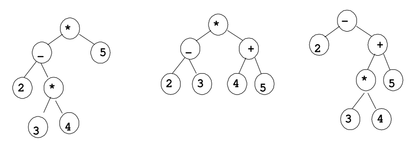
2 3 4 * 5 + - means 2-(3*4+5)

2 3 4 * 5 + - means 2 – (3*4) +5.

In in-fix notation they all read 2 - 3 * 4 + 5

Binary trees are natural way to store expressions. Each expression above corresponds to a different binary tree.

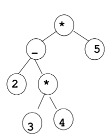
5.5 Expression Trees and Polish Notation



```
Preorder:  +-2*345        *-23+45      -2+*345
Inorder:   2-3*4 +5       2-3*4+5      2-3*4+5
Postorder: 234*-5+        23-45+*      234*5+-
(polish)
```


5.5 Construct expression trees.

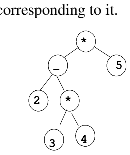
Given an expression in in-fix, develop an algorithm to construct a tree corresponding to it. With in-fix we will need the brackets to clarify precedence.



`In-fix: 2-(3*4)+5`

5.5 Construct expression trees.

Given an expression in postfix notation develop an algorithm to construct a tree corresponding to it.

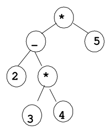


2

`Postorder: 234*-5+`

5.5 Expression Trees

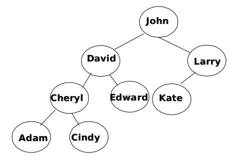
Expression trees are used by compilers to store expressions, and evaluate expressions and convert an expression from one notation(like postorder) to another (like in-order).



5.5 Dictionaries

Binary search trees can be used as dictionaries, where we can insert (sorted) according the lexicographic order of words in O(h) time.

Printing the dictionary would be an in-order traversal and searching for a word can be done in O(h) time.



5.6 Exercises

1)Write an algorithm to find the largest element in a BST. What is the time complexity of your algorithm.

2)Write an algorithm to find the smallest element in a BST. What is the time complexity of your algorithm.

3)Write an algorithm to find the largest element in BT. What is the time complexity of your algorithm.

4)Write an algorithm to check if the given BT is a BST. What is the time complexity of your algorithm.

5)Write an algorithm to perform a level order traversal, but separate the traversal of each level by a level number.

6)Write the algorithm to find the successor of a node in a BST.

7)Write an algorithm to search for a key in a BT. Use one of the recursive traversal techniques. If the key is found, return the node, if not, return nil.

8)Write an algorithm to count the number of nodes in a BT. Use one of the recursive traversal techniques.

5.6 Exercises

9) Write an algorithm to count the number of leaves in a BT. Use one of the recursive traversal techniques.

10) Write an algorithm to find the height of a BT. Use one of the recursive traversal techniques.

11) Print a path from one node A to another node B in a BT.

12) Show a tree in which preorder and inorder generate the same sequence.

13) Draw all possible BST of three elements 45 89 and 90.

14) Using the preorder, postorder and inorder traversal, visit only the leaves of the tree. What do you observe and can you explain the phenomena.

15) Write an algorithm to create an expression tree, given the input in the polish notation.

16) Write an algorithm to create an expression tree, given the input in the in-fix notation with parenthesis.

node

17) Write an algorithm to find all the names starting with the letter L in a dictionary BST.