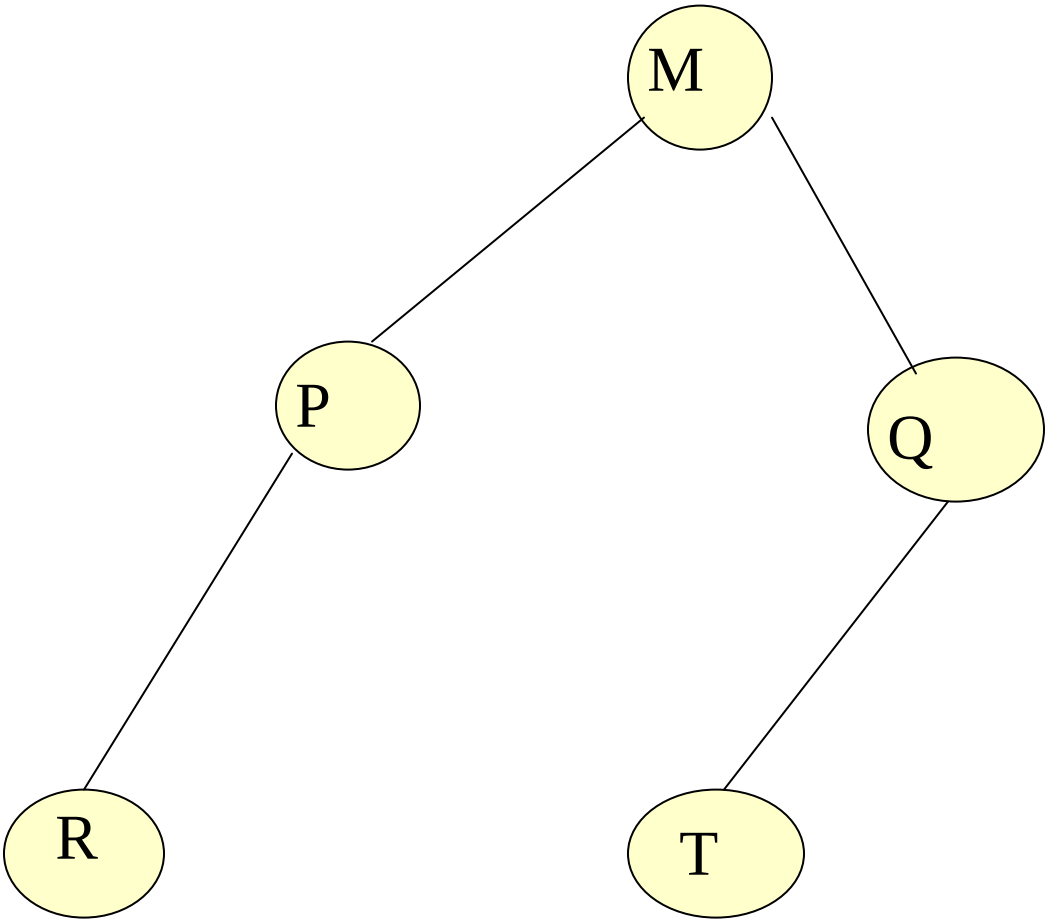
Trees – II

Non-Recursive PreOrder Traversal
Expression Trees
Binary Search Tree

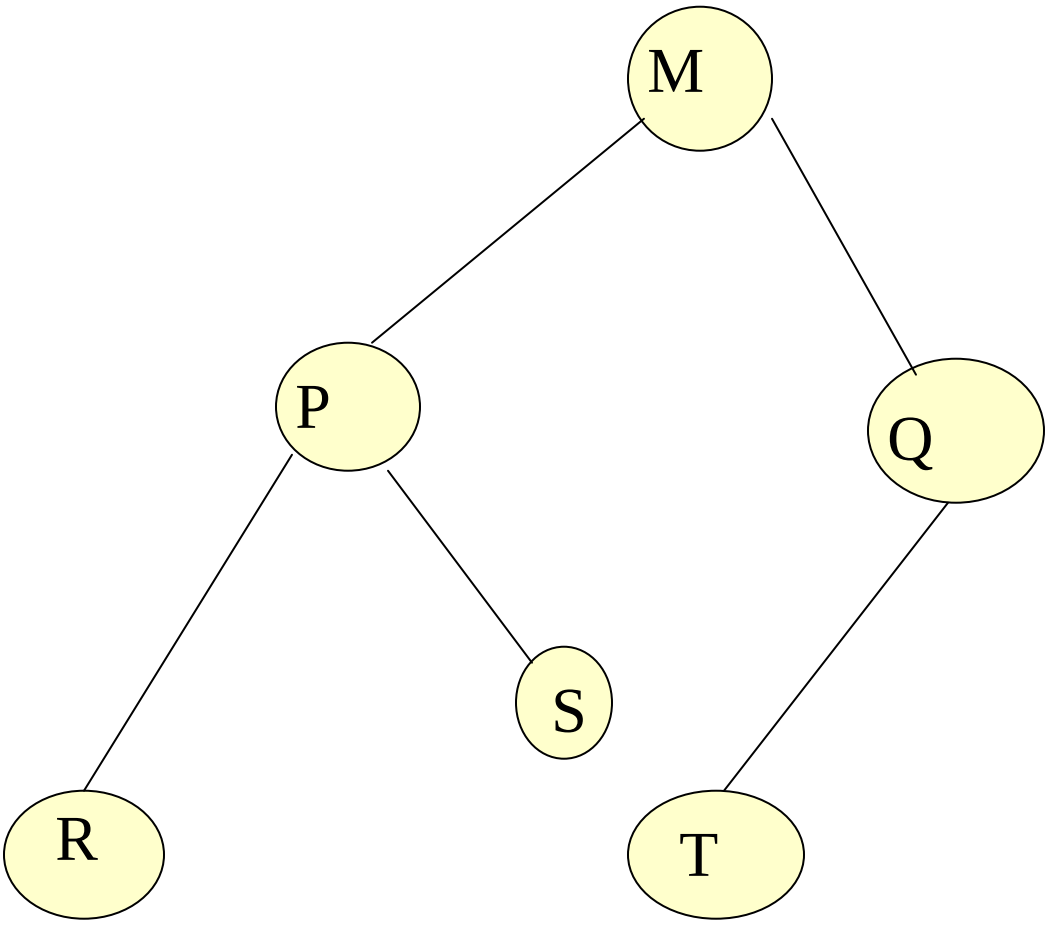
- **Searching**
- **Insertion**
- **Traversal**
- **Creation**

Carry out a pre-order traversal of this tree



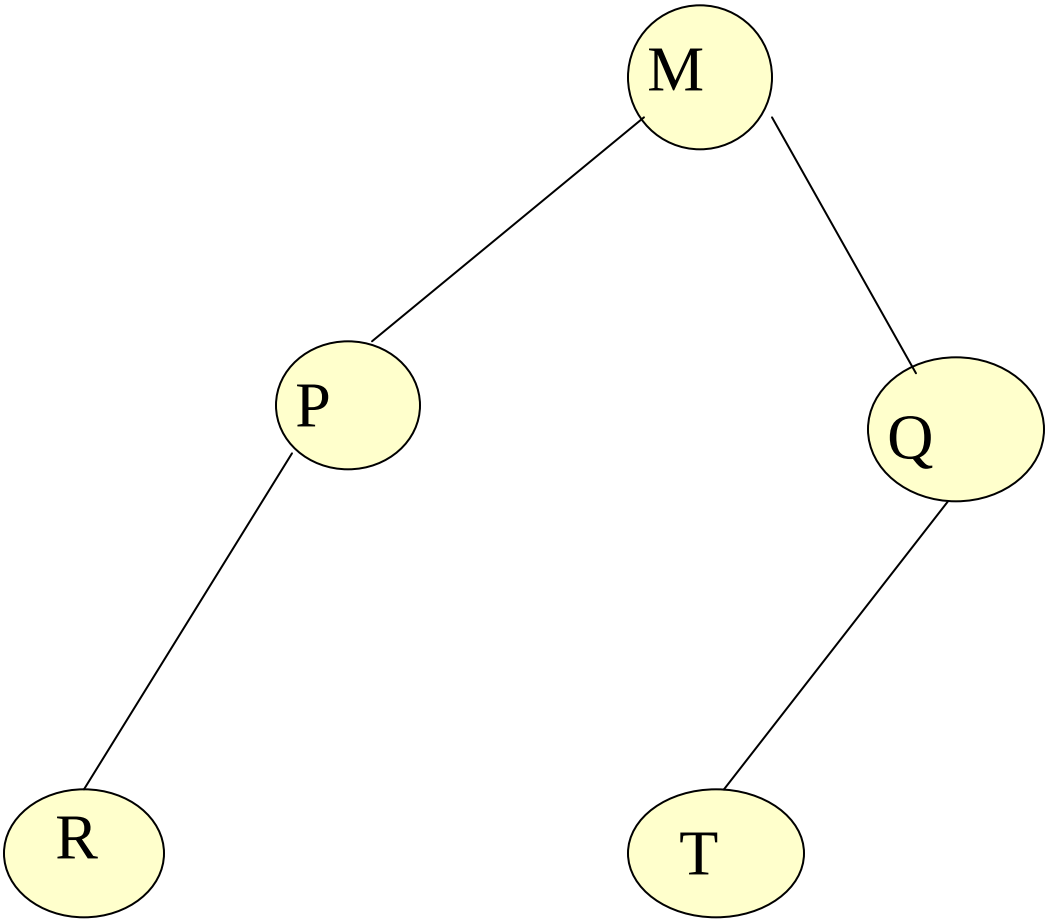

```
void preorder(struct tree_node * p)
{   if (p !=NULL) {
        printf("%d", p->data);
        preorder(p->left_child);
        preorder(p->right_child);
    }
}
```

M P R Q T



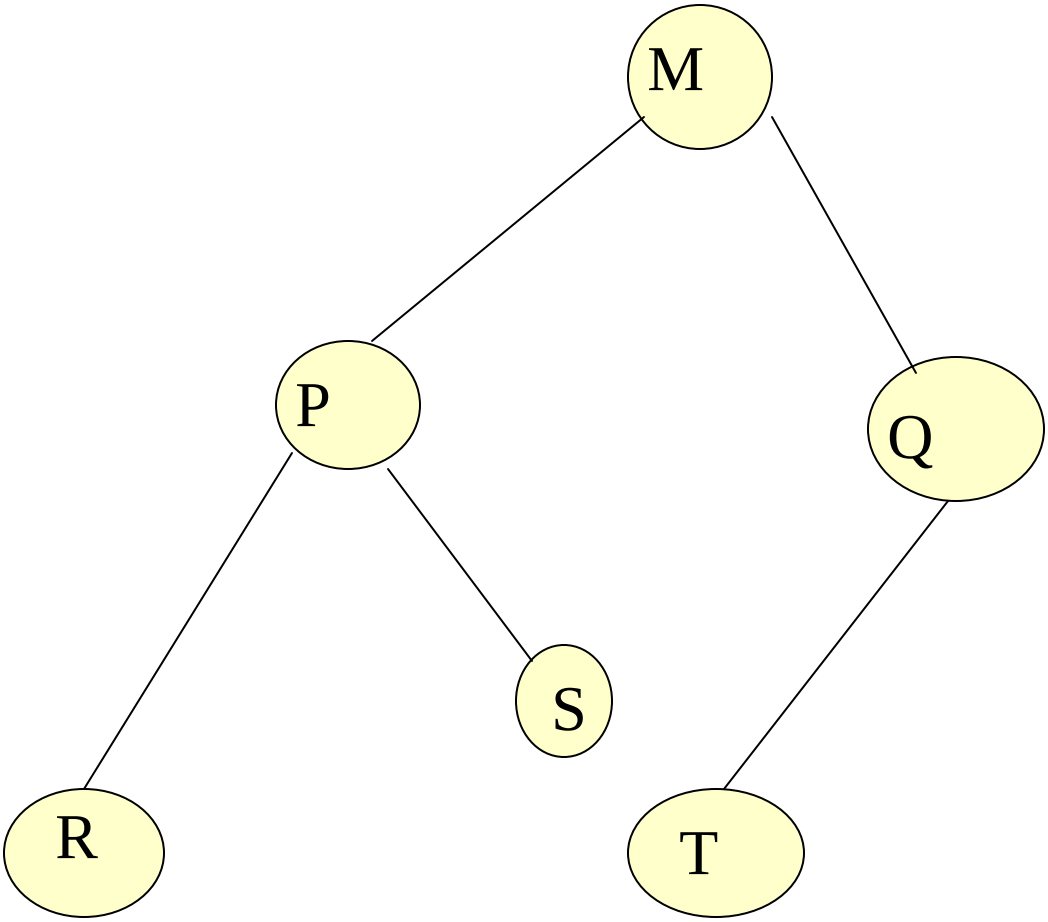
Preorder Traversal: M P R S Q T

Carry out in-order traversal of this tree




```
void inorder(struct tree_node *p)
{  if (p !=NULL) {
       inorder(p->left_child);
       printf("%d\n", p->data);
       inorder(p->right_child);
   }
}
```

IN ORDER TRAVERSAL: R P M T Q



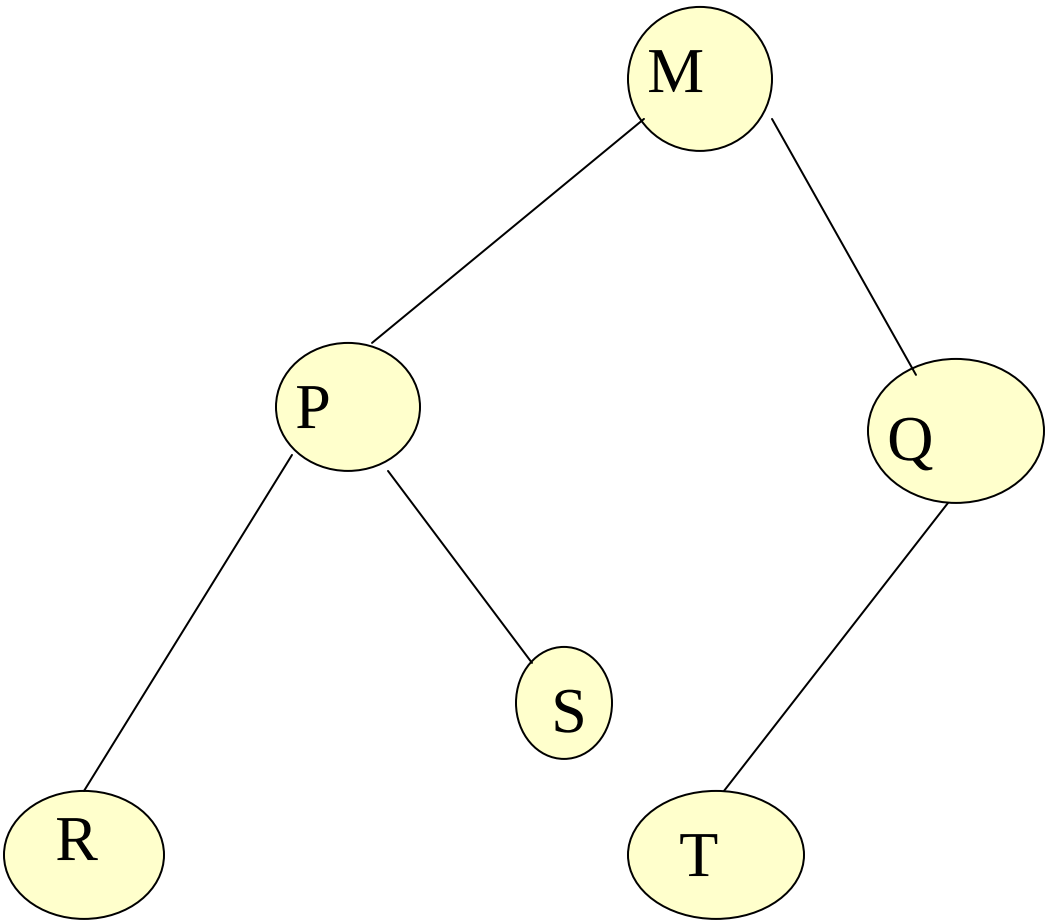
In ORDER TRAVERSAL : R P S M T Q

Non Recursive implementation of preorder tree traversal

In a preorder traversal, we visit the root node first, then we visit its left subtree (all the nodes) and finally visit its right subtree. How do we actually visit all the nodes of a subtree? We can write a non-recursive algorithm by making use of a stack. To start with we push the root node on the stack. Then we push the right child and the left child of the current node on a stack recursively. Then we pop them and print them. Look at this implementation:

```
*p = root;
if (p != NULL)
{
   push(p);
   while ( stack not empty)
    {
        p = pop stack;
       printf("%d ",p->data);
      if ( p->right_child != NULL)
           push ( p-> right_child);
      if (p ->left_child != NULL)
          push (p -> left_child);
     }
}
```

/* note the left child is pushed on top of right node, so that it gets popped up first */



Non Recursive Preorder Traversal:

Stack position	Popped values
M	
	M
Q P	
Q	M P
Q S R	
Q S	M P R
Q	M P R S
	M P R S Q
T	
-	M P R S Q T

Expression Trees

Many compilers make use of trees, as they serve as ideal representations for the hierarchical structure of a program. Design of a compiler is a complex process, which you shall learn in a later course. Let us look at one particular aspect of compiler, which deals with evaluation of arithmetic expressions. An arithmetic expression can be represented as a binary tree. Such a tree is called an Expression Tree. An expression tree is a binary tree representing an arithmetic expression where the leaf nodes of the tree represent the operands (variables or constants) and the internal nodes and the root node represent the operators.

Constructing an Expression Tree

Constructing an expression tree involves two parts:
Lexical Analysis: Dividing the input into tokens, each of which represents either integer constant, an operator or a variable name.
Parsing: Determining if the individual tokens represent a legal expression and finding out the structure of that expression.
Generating an expression tree from an infix expression is not very difficult if there is no ambiguity in the expression and it is overly parenthesized.

Example :

3 + 5 / 9 + 8 could mean

(3 + (5/9) + 8) or (3 + 5 / (9 +8))

There will be one distinct expression tree for each of the interpretations depending on the parsing scheme.

In this section we consider building an expression tree given the expression in its **postfix** form.

The construction starts with reading the postfix expression one symbol at a time.

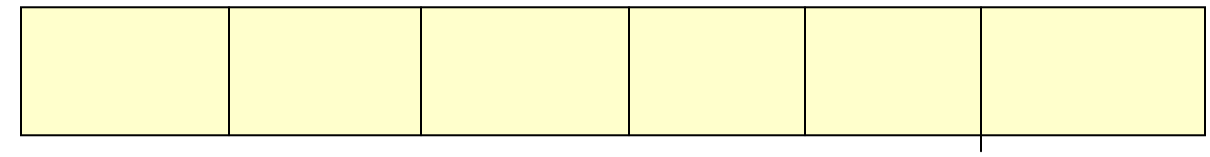
If the symbol is an operand , we create a one-node tree and push it onto a stack.

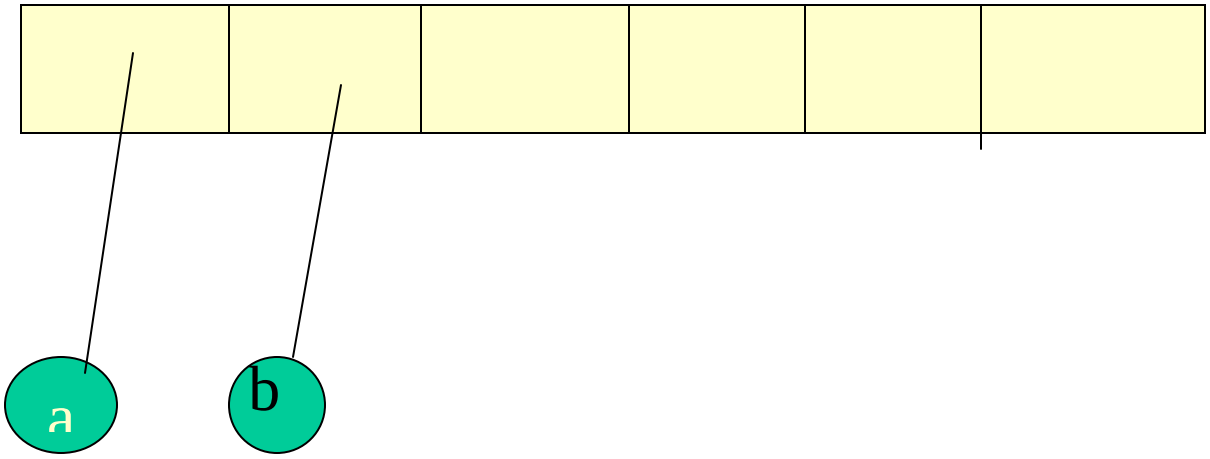
If the symbol is an operator, we pop two trees T1 and T2 from the stack (popping up T1 first), and form a new tree with the operator as the root, T2 as the left child and T1 as the right child.

The new tree is then pushed onto the stack.

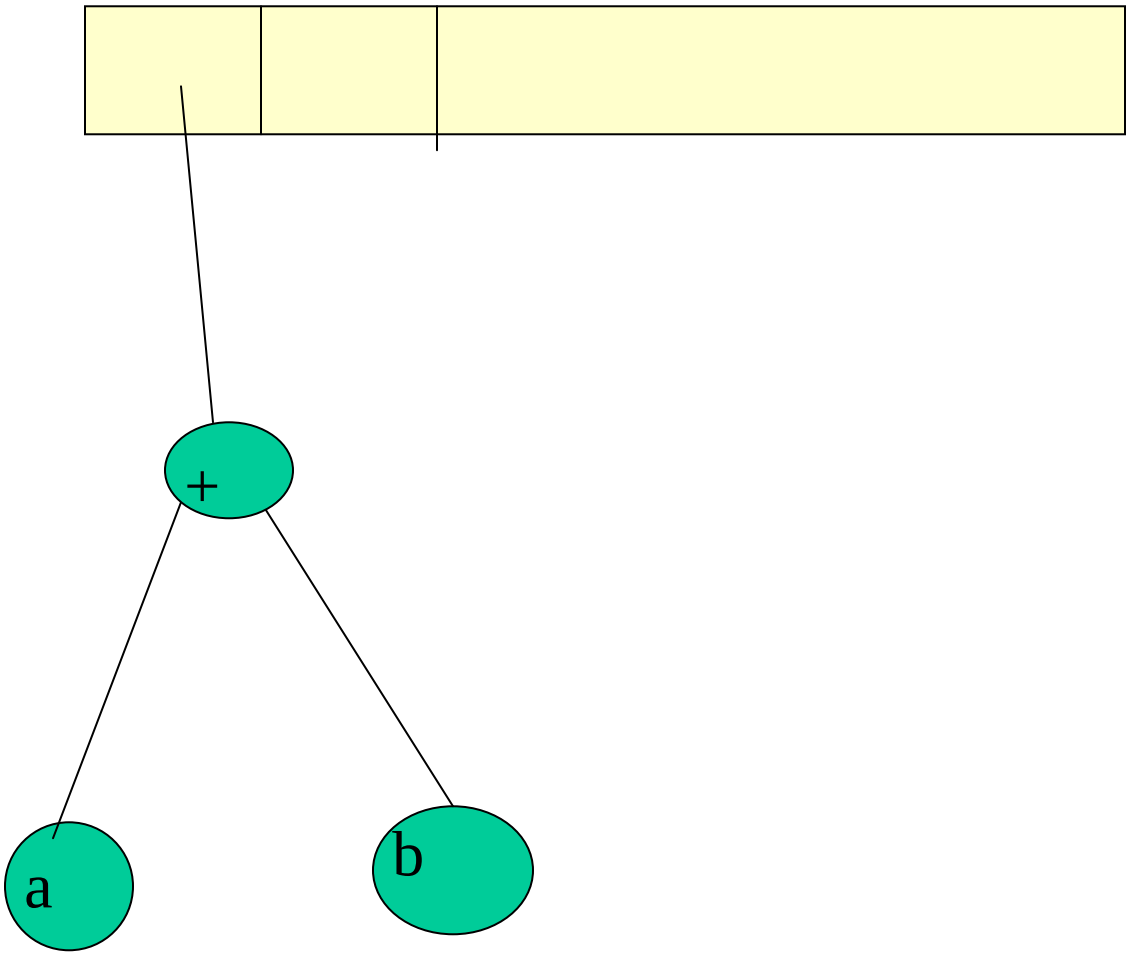
As an example, let the input expression be

a b +
The first two symbols are operands. So we create one node trees and push them onto a stack.





The next symbol is an operator +, so the two one node trees are popped, and a new tree is formed. The first tree to be popped is the one containing b, so this becomes the right child of the operator, the second tree to be popped contains the operand a , and this becomes the left child of the operator.



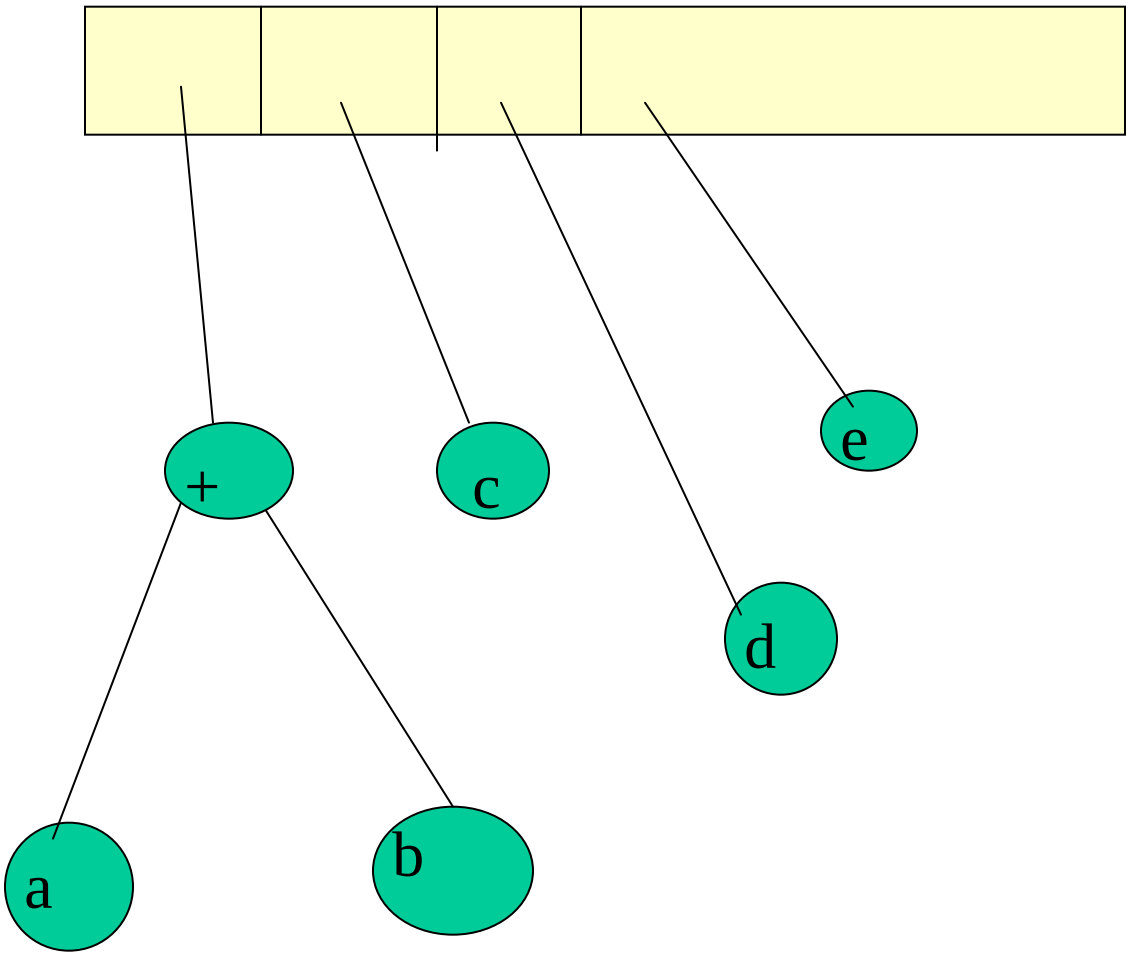
The expression tree is now constructed. It is obvious that he evaluation of the expression is nothing but in-order traversal to yield a + b.

Let us take a larger expression:

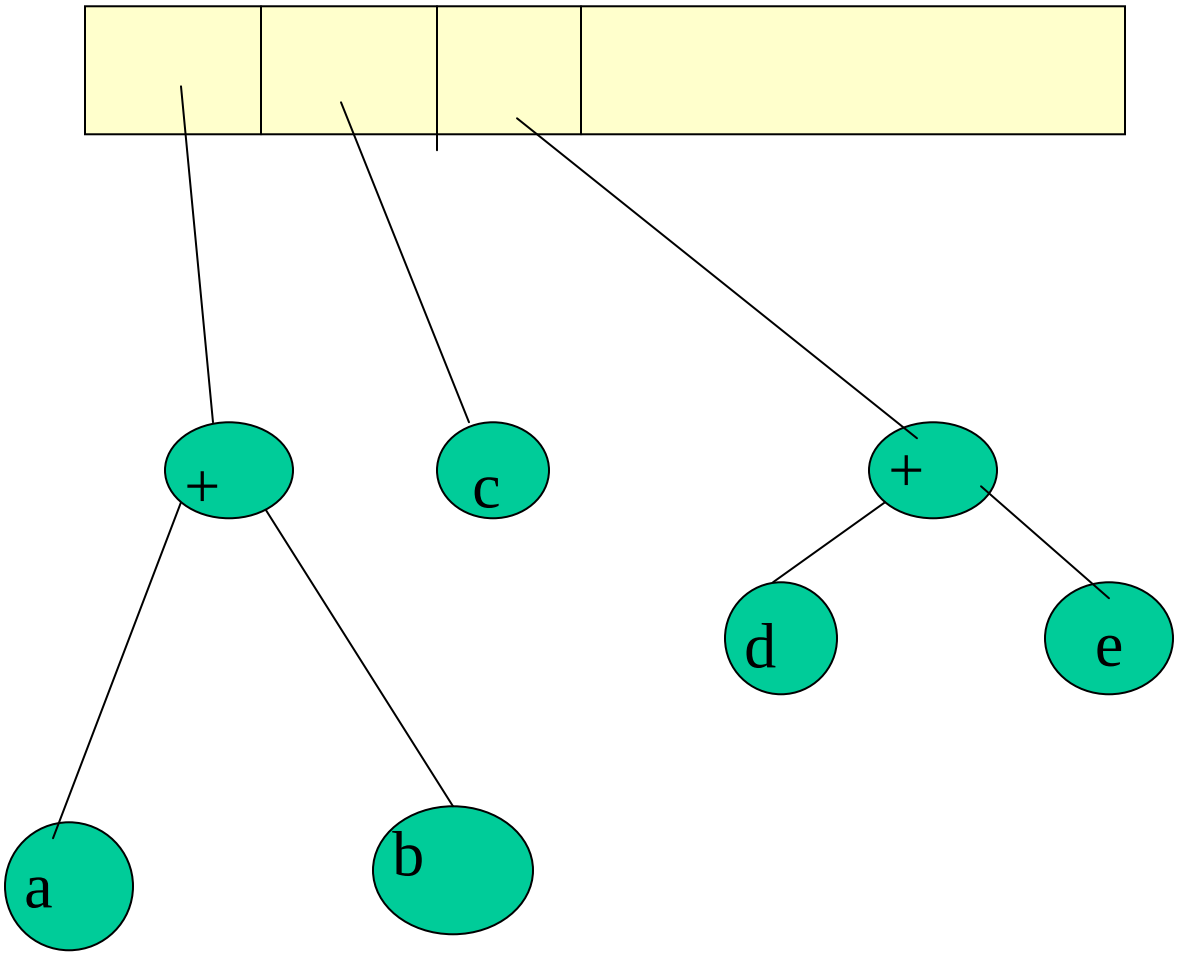
a b + c d e + * *

Let us say we have processed the first three tokens, and have already reached the '+' operator.

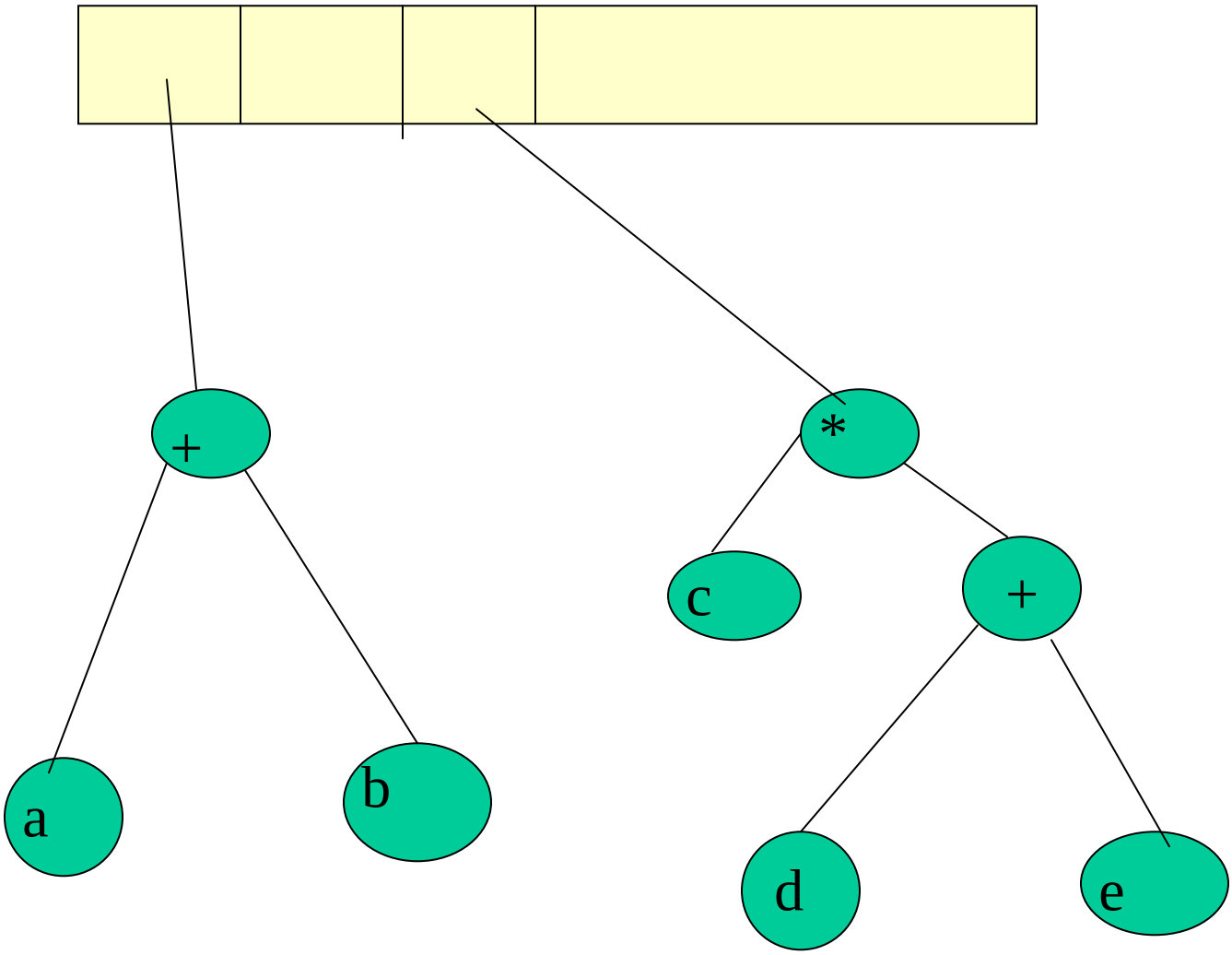
Next we read c, d, e and construct one node trees and push them onto the stack.



Next is an operator ‘+’, so last two trees are popped out and a new tree formed with this as root.



Next symbol is another operator ‘*’, so again two trees are popped from the stack and merged to form a new tree with this operator at the root:



Finally the last operator ‘*’ causes the last two entries from the stack to be pulled out , i.e. the tree containing ‘+’ at the root and the tree containing ‘*’ at the root.
These are merged with the second ‘*’ operator as the root, and the expression tree is complete, and now it is the only entry in the stack.

Let us now see how do we evaluate this tree.

```
int eval (struct tree_node *p)
{
  int lhs, rhs;
  char op;
  if ( p -> left_child = NULL && (p->
right_child = NULL)
      return ( p->data);
  else
       lhs = eval( p ->left_child);
       rhs = eval( p ->right_child);
       op = p ->data;
       switch (op) {
          case '+': return (lhs+rhs);
          case '-': return (lhs-rhs);
          case '*': return (lhs*rhs);
          case '/': return (lhs/rhs);
}
```

Binary Search Tree (BST)

We have seen earlier that if the values in nodes of a binary tree are arranged in a specific order, with all elements smaller than the root stored in left subtree and all elements greater than the root stored as right subtree, it represents a sorted list. The search complexity reduces considerably, as the height of the tree is much less than total number of elements. Of course, one has to keep in mind that the tree has to be organized in a specific manner. Such a tree is known as a Binary Search tree, because it permits us to carry out a search similar to the **Binary search** method that we have used on a sorted array.

Let us first of all define a BST.

A Binary search tree (BST) is a binary tree that is

either empty , or

each node contains a data value satisfying the following:

a) all data values in the left subtree are smaller than the data value in the root.

b) the data value in the root is smaller than all values in its right subtree.

c) the left and right subtrees are also binary search tees.

Searching for a target

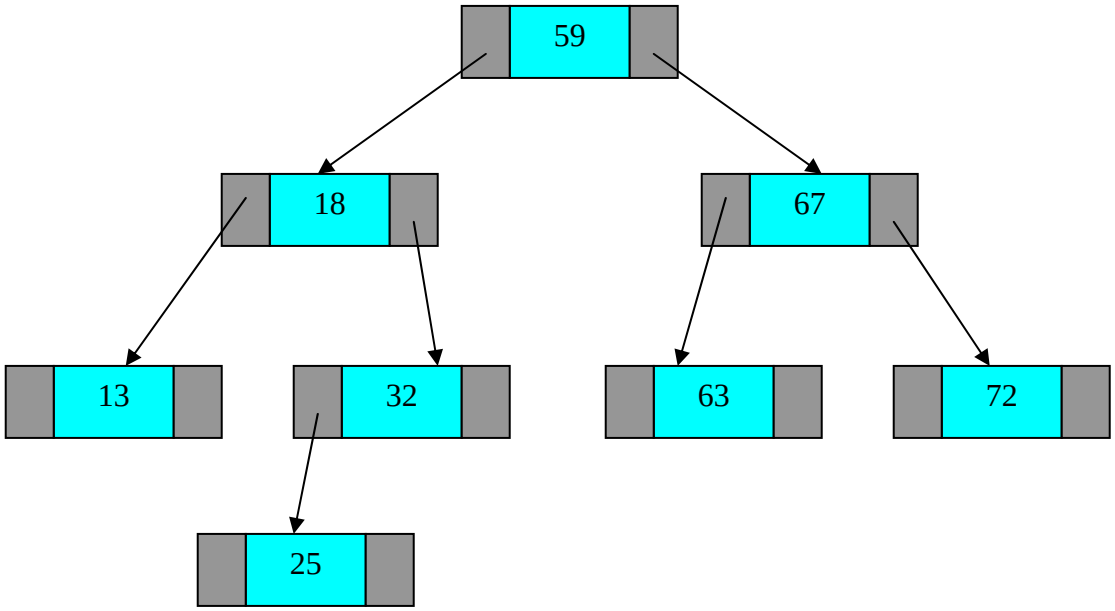
in the Binary Search Tree

The definition of a binary search tree allows us to quickly search for a particular value in the BST. Check the given value with the value in the root node. If it matches, return 1, else if given value is smaller , look into left subtree, else look into right subtree. If subtree is null, return 0.

```
struct tree_node{
    int data;
    struct tree_node *left_child;
    struct tree_node *right_child;
};

int treeSearch( struct tree_node *p, int
target)
{   if (p!=NULL)
        {
          if (p->data == target)
          return 1;
          else if (p->data > target)
               treeSearch(p->left_child);
          else
              treeSearch(p->right_child);
         }
     return 0;
}
```

Example Tree:



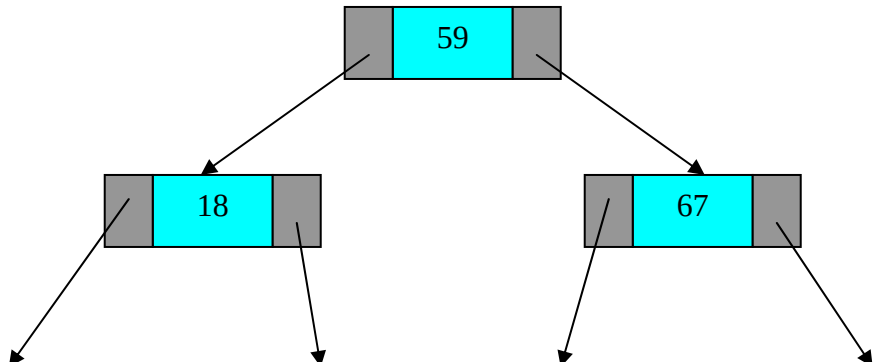
Inserting a node in a BST

Insertion of a new node in a BST has to be done only at the appropriate place for it, so that overall BST structure is still maintained, i.e. the value at any node should be less than that at right node and less than that at the left node.

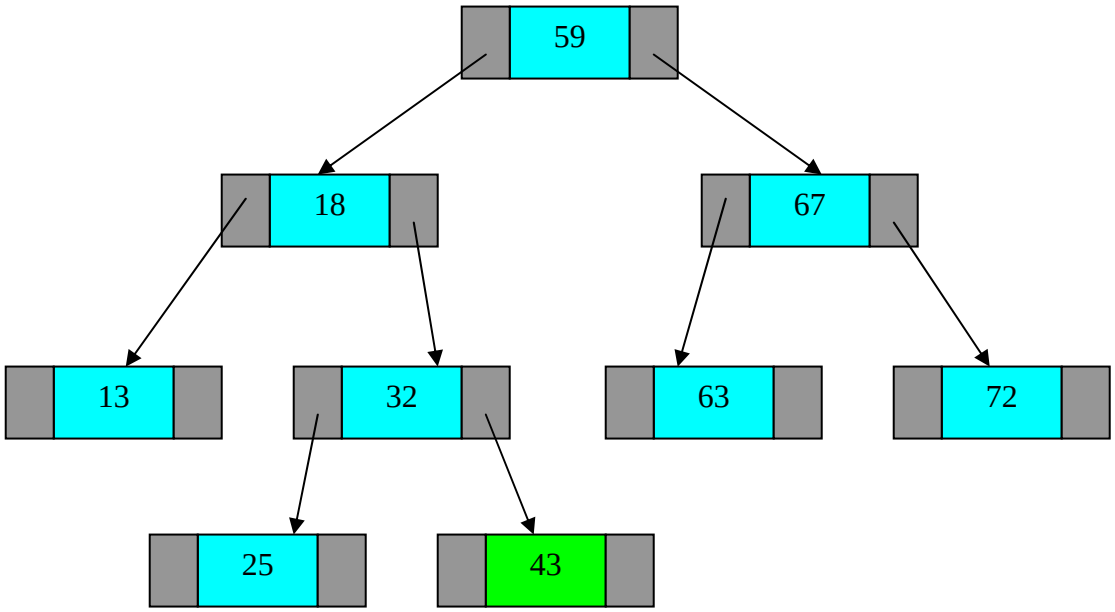
Inserting a new node into a BST **always** occurs at a NULL pointer.
There is never a case when existing nodes need to be rearranged to accommodate the new node.

As an example, consider inserting the new value 43 into the BST shown above. Where is the new node supposed to go?

Hint: search for node containing 43. Obviously, you won't find it, but the search algorithm has taken you to the NULL pointer where it should be placed, so this is the appropriate place to insert it.



43 should
be here, so put it here!



Assume that a node “new” has been created containing the data, and that both the left and right child fields have been set to NULL. It is now desired to insert the node at its proper place in an existing BST with root node p.

```
struct node* insert(struct tree_node *p, struct
tree_node *new)
{

  if (p == NULL)
       p = new;
  else
     if(p->data >new->data)
       p->left_child= insert(p->left_child, new);
     else
       p->right_child=insert(p->right_child, new);

     return p;
}
```

Traversing a Binary Search Tree

The above BST can be traversed starting with the root node (Preorder traversal) to result in the sequence

59,18,13,32,25,43,67,63,72

However, an interesting feature is revealed with the Inorder Traversal which yields

13, 18, 25, 32, 43, 59, 63, 67, 72

What do you notice? This is an ordered listing of the values of BST nodes, with the left most node being the smallest element and the right most node being the largest element.

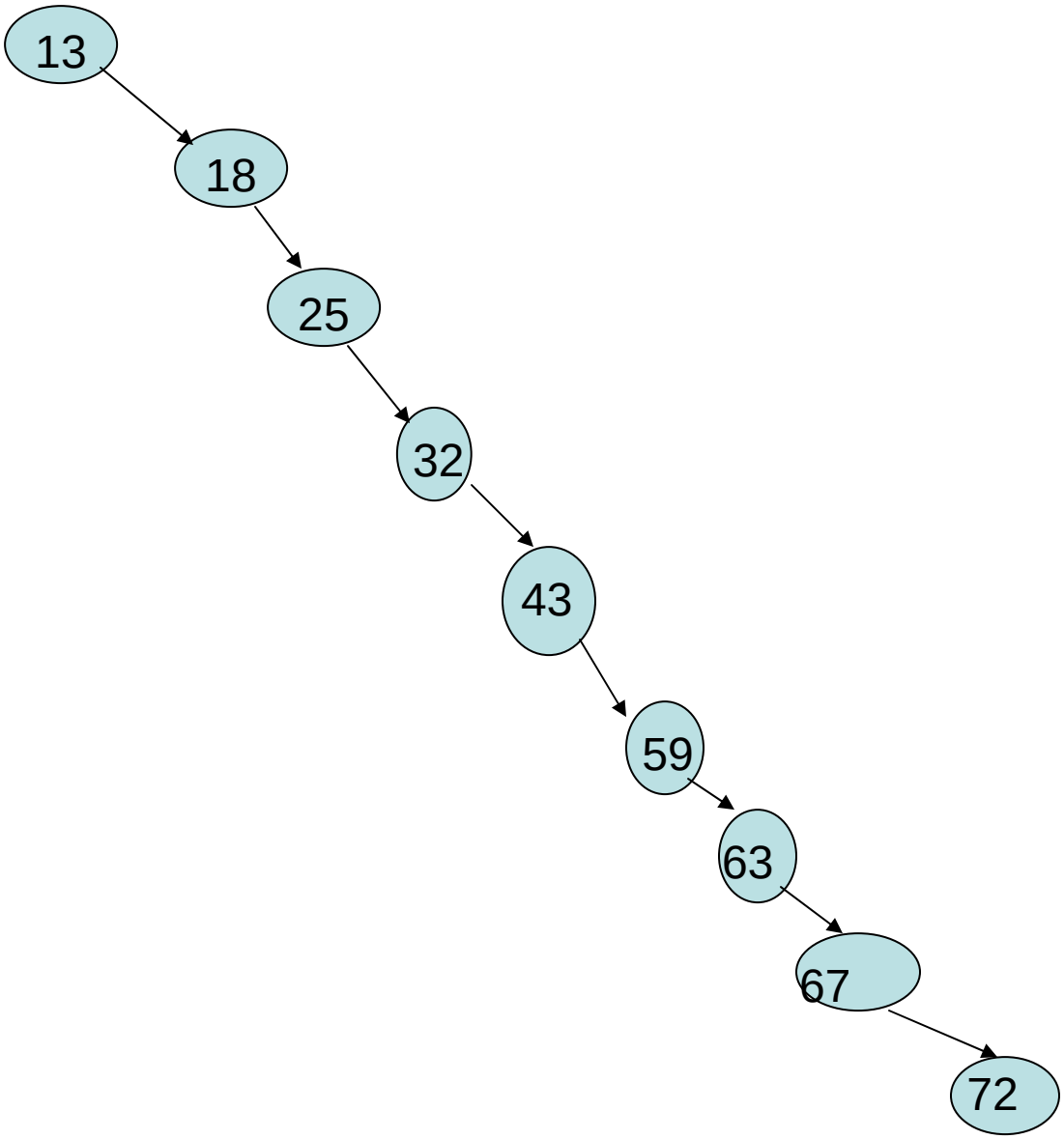
Creating a Binary Search Tree

To create a binary search tree, keep on inserting the nodes as and when they arrive. Note that the shape of the BST will depend on the order of insertion of the nodes.
The above tree was created from the sequence

59, 18, 13, 67, 32, 72, 25, 63, 43

If the values arrived in the order
13, 18, 25, 32, 43, 59, 63, 67, 72

the BST would take the following shape



Notice that it is not a balanced tree, but skewed towards right. In such a case, the search and insertion complexity would be O(n) instead of O(log n). If the sequence were

entered in descending order then it would result in a left-skewed tree. For any other ordering of the sequence the complexity would lie in between O(n) and O(log n).

Suppose we were interested in generating a balanced BST, given any arbitrary sequence of values.
We could this by first storing all the elements in an array and sorting them in ascending order.

- Once sorted, the element at the midpoint of the array will become the root of the BST. The array can now be viewed as consisting of two subarrays, one to the left of the midpoint and one to the right of the midpoint.
- The middle element in the left subarray becomes the left child of the root node and the middle element in the right subarray becomes the right child of the root.
- This process continues with further subdivision of the original array until all the elements in the array have been positioned in the BST.
- Take care to completely generate the left subtree of the root before generating the right subtree of the root. If this is done, a simple recursive procedure can be used to generate a balanced BST.

```
void balance( int sequence[], int
first, int last)
{
    int mid;
       if (first <= last) {
```

```
        mid = (first + last)/2;
        insert_BST(data[mid]);
        balance(data, first, mid-1);
        balance(data, mid+1, last);
    }
}
```

where the function insert_BST creates a new node with the value data[mid] and calls the ‘insert ‘ function to insert the new node into the BST.