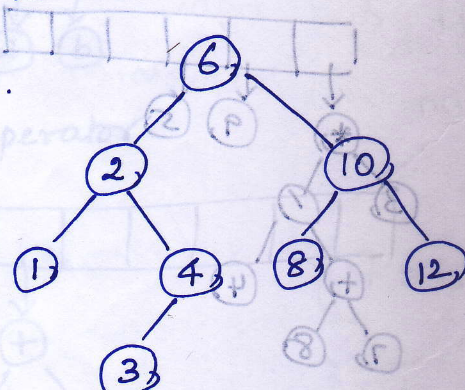
BINARY SEARCH TREE.

* It is a special type of binary tree. Each node in tree is assigned a key value. The property that makes a binary tree into a binary search tree is that for every node x in the tree, the value of all keys in its left subtree are smaller than key value in x and the values of all keys in its right subtree are larger than key value in x.

eg.



Algorithm:

(i) Declarations:

```
struct TreeNode;
typedef struct TreeNode *Position;
typedef struct TreeNode *SearchTree;
SearchTree MakeEmpty (SearchTree T);
Position Find (int x, SearchTree T);
Position FindMin (SearchTree T);
Position FindMax (SearchTree T);
SearchTree Insert (int x, SearchTree T);
SearchTree Delete (int x, SearchTree T);
struct TreeNode
{
 int x;
 SearchTree Left;
 SearchTree Right;
```

2) MakeEmpty:

- Routine to make a tree empty.

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

3) Find:

* It returns a pointer to the node in tree T that has key X or NULL if there is no such node.

(i) if T is NULL, return NULL.

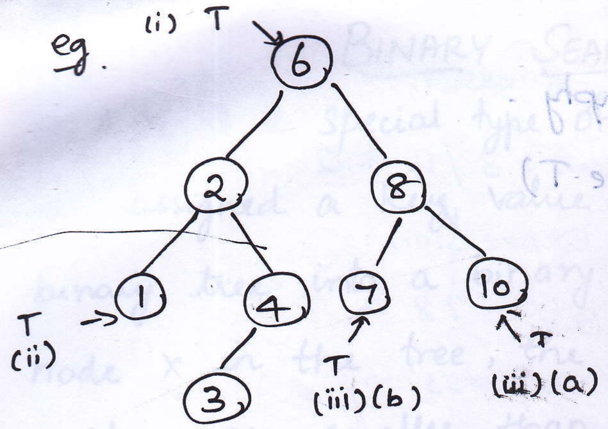
(ii) Otherwise, if key at T is X, return T.

(iii) Otherwise, make a recursive call on a subtree of T, either left or right, depending on relationship of X to key stored in T.

Alg:

```
Position Find (int 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;
```

eg.



(i) $X = 6$.

return T.

(ii) $X = 1$.

$X < 6$ (ie) $1 < 6 \Rightarrow$ True

Find (1, T→left)

$X < 2$ (ie) $1 < 2 \Rightarrow$ True

Find (1, T→left)

$1 < 1 \Rightarrow$ No.

$1 > 1 \Rightarrow$ No

return T.

(iii)(a) $X = 10$.

$X < 6$ (ie) $10 < 6 \Rightarrow$ No.

$X > 6$ (ie) $10 > 6 \Rightarrow$ Yes.

Find (10, T→right)

$X < 8$ (ie) $10 < 8 \Rightarrow$ No.

$X > 8$ (ie) $10 > 8 \Rightarrow$ yes

Find (10, T→right)

$X < 10$ (ie) $10 < 10 \Rightarrow$ No

$X > 10$ (ie) $10 > 10 \Rightarrow$ No.

return T.

(iii)(b) $X = 7$.

$X < 6$ (ie) $7 < 6 \Rightarrow$ No.

$X > 6$ (ie) $7 > 6 \Rightarrow$ Yes

Find (7, T→right)

$X < 8$ (ie) $7 < 8 \Rightarrow$ Yes.

Find (7, T→left)

$7 < 7 \Rightarrow$ No

$7 > 7 \Rightarrow$ No.

return T.

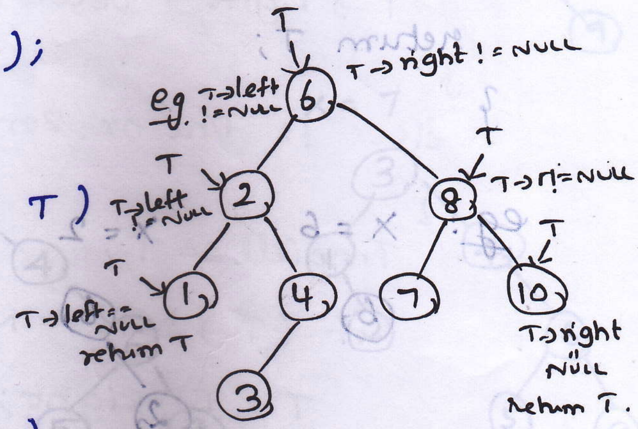
4. FindMin and FindMax:

* It returns the position of the smallest and largest elements in the tree.

* To perform FindMin, start at the root and left as long as there is a left child.

* Similarly for FindMax, start at root and go right as long as there is a right child.

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




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

5. <u>Insert</u> :

* To insert x into tree T, proceed down the tree similar to find routine.
* If x is found, do nothing. Otherwise insert x at the last spot on the path traversed.

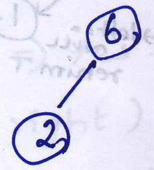
<u>Alg</u>:

```
SearchTree Insert (int x, SearchTree T)
{
  if (T== NULL)
  {
    T = malloc (sizeof (struct TreeNode));
    T->Element = x;
    T->Left = T->Right = NULL;
  }
  else
```

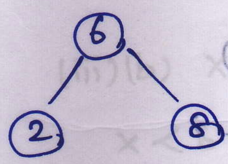
```
if (x < T->Element)
    T->Left = Insert (x, T->Left);
else if (x > T->Element)
    T->Right = Insert (x, T->Right);
/* else x is in tree already */
return T;
}
```

eg: x = 6

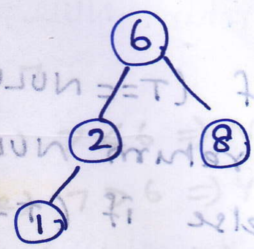
x = 2



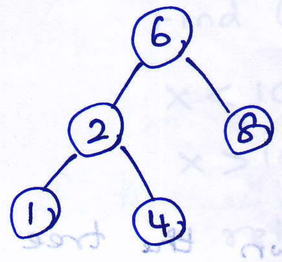
x = 8



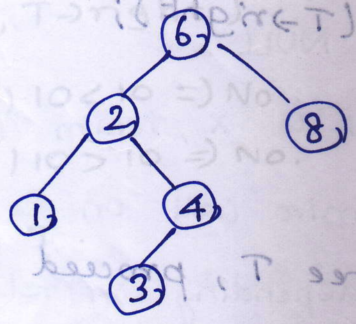
x = 1



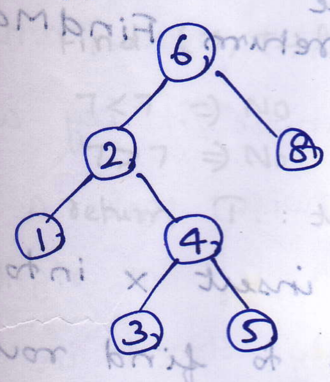
x = 4



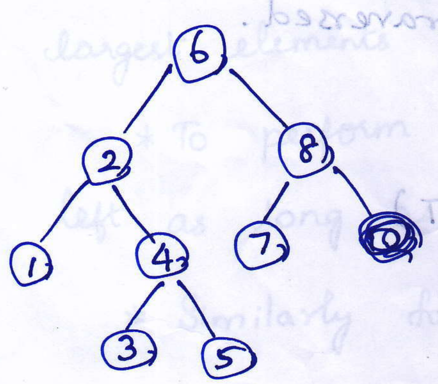
x = 3



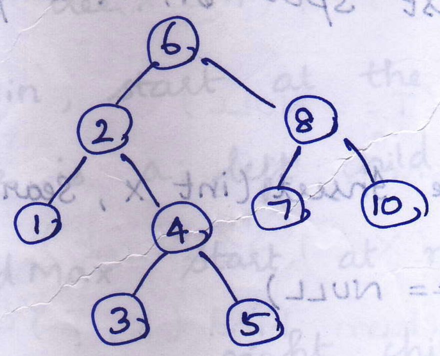
x = 5



x = 7

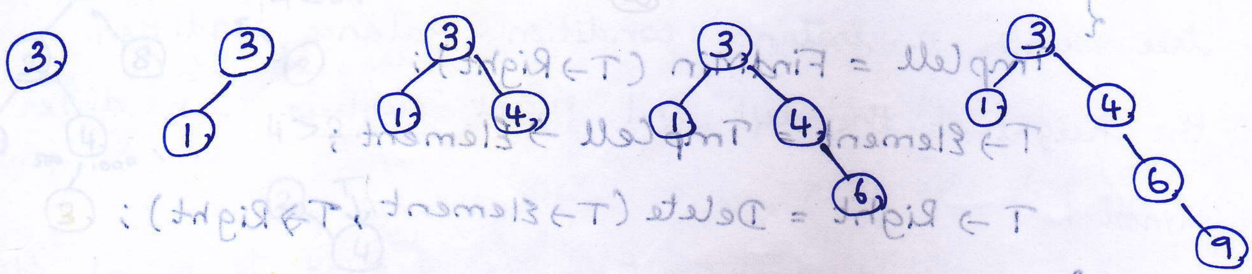


x = 10

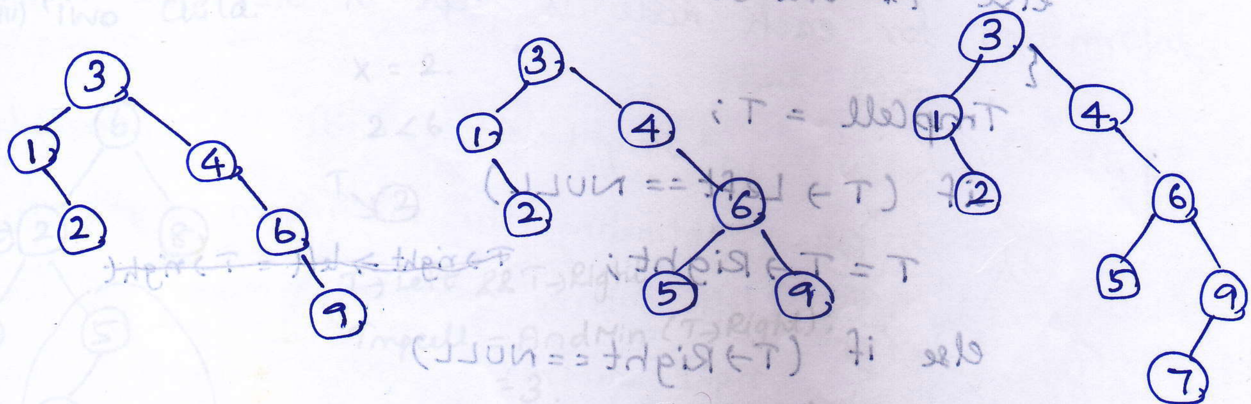


eg 2: 3, 1, 4, 6, 9, 2, 5, 7

x = 3 x = 1 x = 4 x = 6 x = 9



x = 2 x = 5 x = 7



5. Delete:

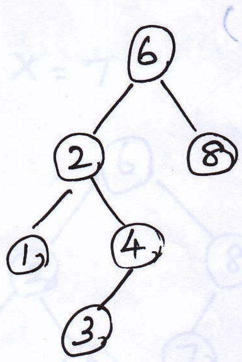
* There are 3 cases of deletion
  - If node to be deleted is a leaf node.
  - If node to be deleted has one child (either left or right)
  - If node to be deleted has 2 children (both left / right)

Alg:

```
SearchTree Delete (int x, SearchTree T)
{
  Position TmpCell;
  if (T == NULL)
    Error ("Element not found");
  else
    if (x < T→Element)
      T→Left = Delete (x, T→Left);
  else
     if (x > T→Element)
      T→Right = Delete (x, T→Right);
```

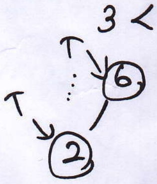
```
/* Found the element to be deleted */
if (T->Left && T->Right)   /* Two children */
{
  TmpCell = FindMin (T->Right);
  T->Element = TmpCell->Element;
  T->Right = Delete (T->Element, T->Right);
}
else  /* One or zero child */
{
  TmpCell = T;
  if (T->Left == NULL)
     T = T->Right;        T->right->left = T->right
  else if (T->Right == NULL)
     T = T->Left;         T->left->right = T->left;
  free (TmpCell);
}
return T;
}
```

Case (1) Leaf node



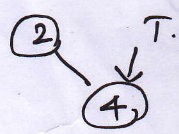
x = 3

3 < 6 => T.

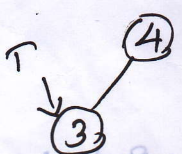


3 < 2 => f.

3 > 2 => T.



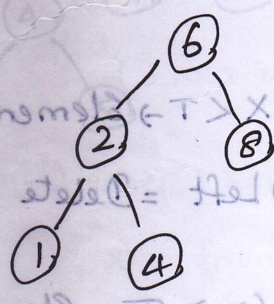
3 < 4 => T.



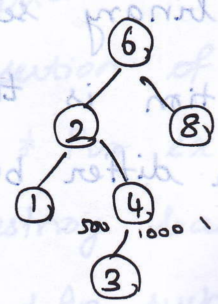
TmpCell = T;

T->Left & T->Right = NULL

free (TmpCell)



Case (ii) One child.

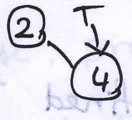


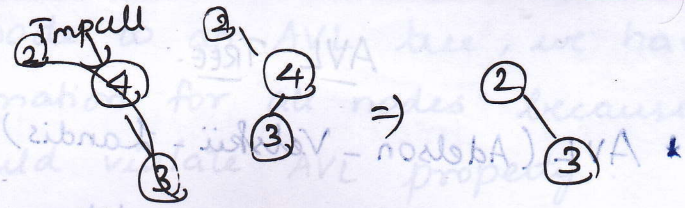
X = 4.

4 < 6.

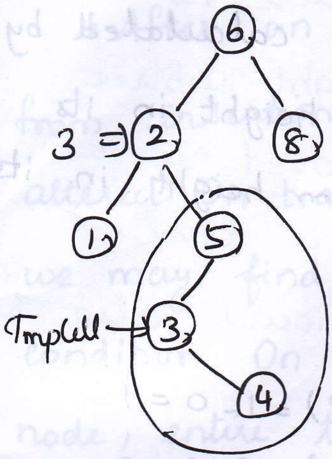
T → 2

4 > 2.





Case (iii) Two child.



X = 2.

2 < 6

T → 2

T→Left && T→Right

Tmpcell = FindMin (T→Right);

= 3.

T→Element = TmpCell→Element;

T→Right = Delete (3, T→Right);