

Data Structure:

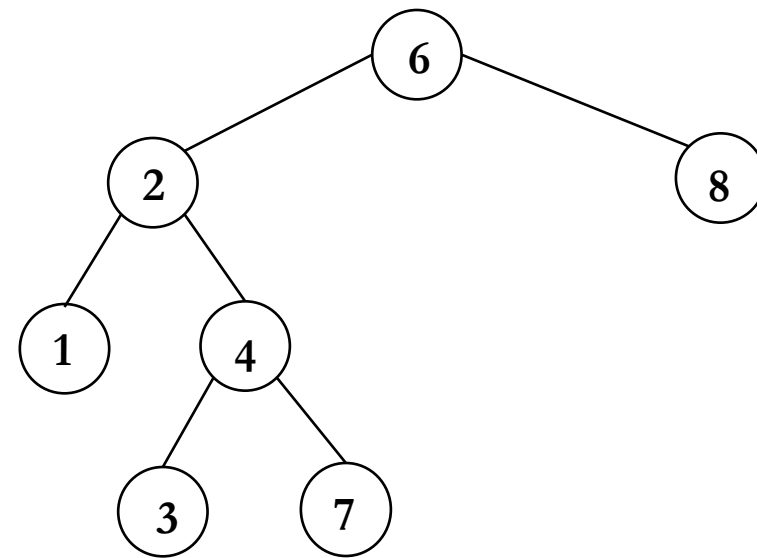
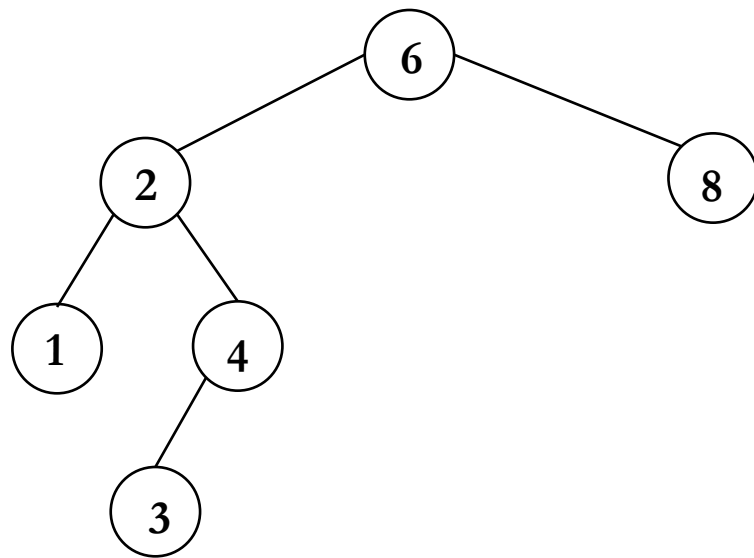
Binary Search Tree

Binary Search Tree

- one of the most fundamental problems in data structure design
 - there are a set of records R_1, R_2, \dots, R_n which are associated with distinct key values X_1, X_2, \dots, X_n , respectively.
 - given a search key x , find the record if it occurs in the set
- for every node X in the tree,
 - the values of all the **keys in its left subtree are smaller** than the key value in X
 - the values of all the **keys in its right subtree are larger** than the key value in X

Binary Search Tree

■ which one can be the binary search tree?



Search in a linear array

- sequential search
 - simply store the keys in a linear array and search sequentially
 - insertion: $O(1)$, searching $O(n)$
- binary search
 - searching: $O(\log n)$, insertion/deletion: $O(\log n)$

Binary Search Tree

A BST ADT can process the following requests

- `insert(x,T)`:
 - insert x into T
 - if x already exists, do appropriate action (e.g., do nothing, return error message, increment reference count.)

- `delete(x,T)`:
 - delete x from T
 - if x does not exist, issue an error message

- `find(x,T)`: search x in T
return either `True/False` or the pointer to the record

Binary Search Tree: data structure

```
struct TreeNode;  
typedef struct TreeNode* SearchTree;  
typedef struct TreeNode* Node;
```

```
struct TreeNode  
{  
    ElementType Element;  
    SearchTree Left;  
    SearchTree Right;  
}
```

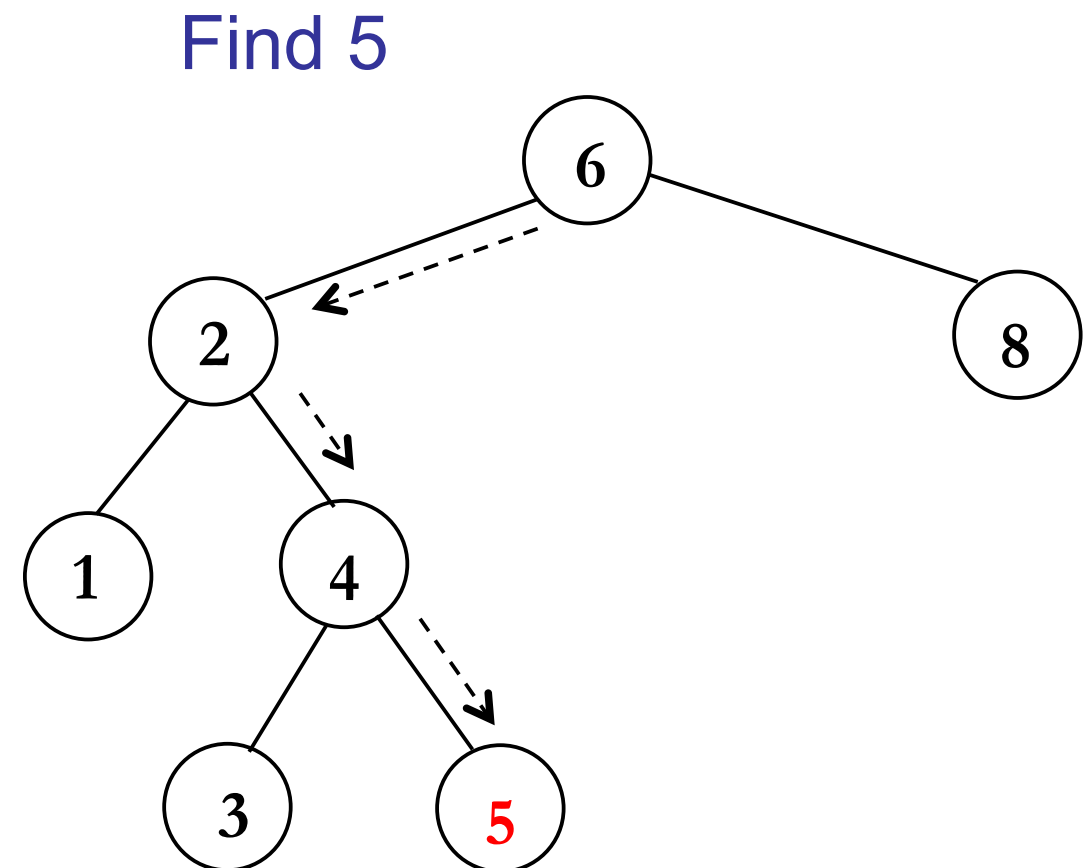
Binary Search Tree: Find

```
Node Find( ElementType 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 /* X == T->Element */
        return T;
}
```



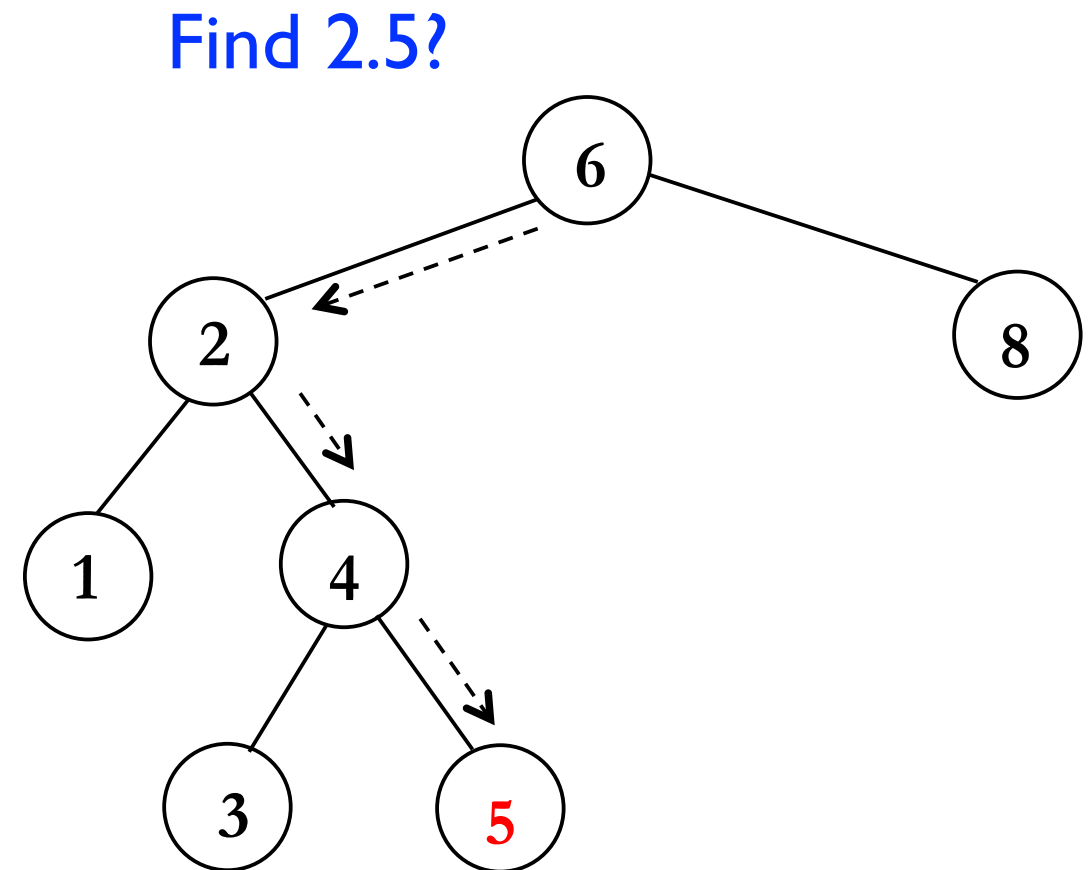
Binary Search Tree: Find

```
Node Find( ElementType X, SearchTree T )
{
    if ( T == NULL )
        return NULL;

    if ( X < T->Element )
        return Find( X, T->Left );

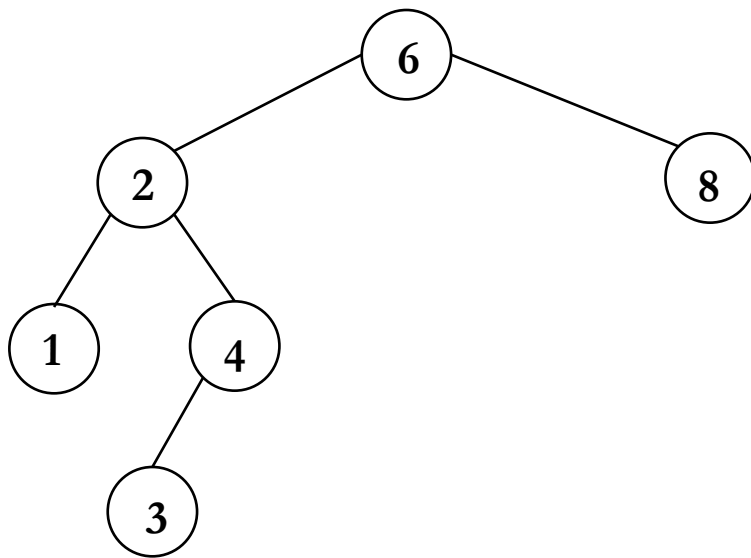
    else if ( X > T->Element )
        return Find( X, T->Right );

    else /* X == T->Element */
        return T;
}
```



Binary Search Tree: FindMax

nonrecursive implementation

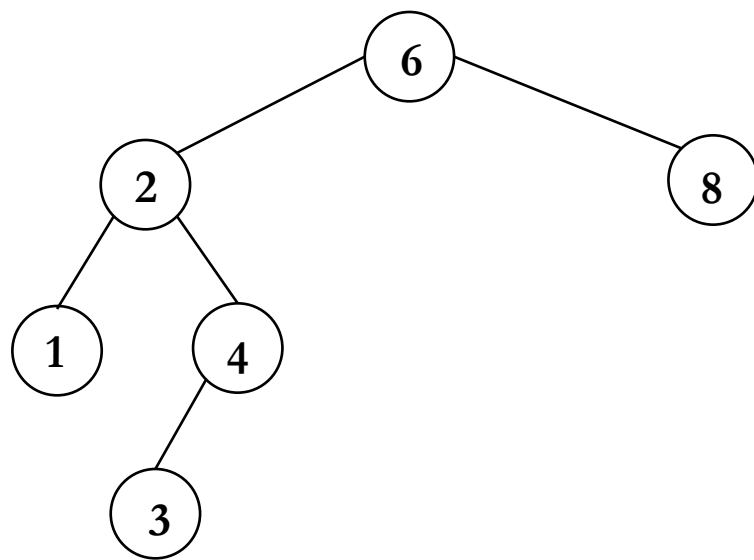


```
Node FindMax( SearchTree T )
{
    if (T == NULL)
        return NULL

    else
        while( T->Right != NULL )
            T = T->Right;

        return T;
}
```

Binary Search Tree: FindMin



recursive implementation

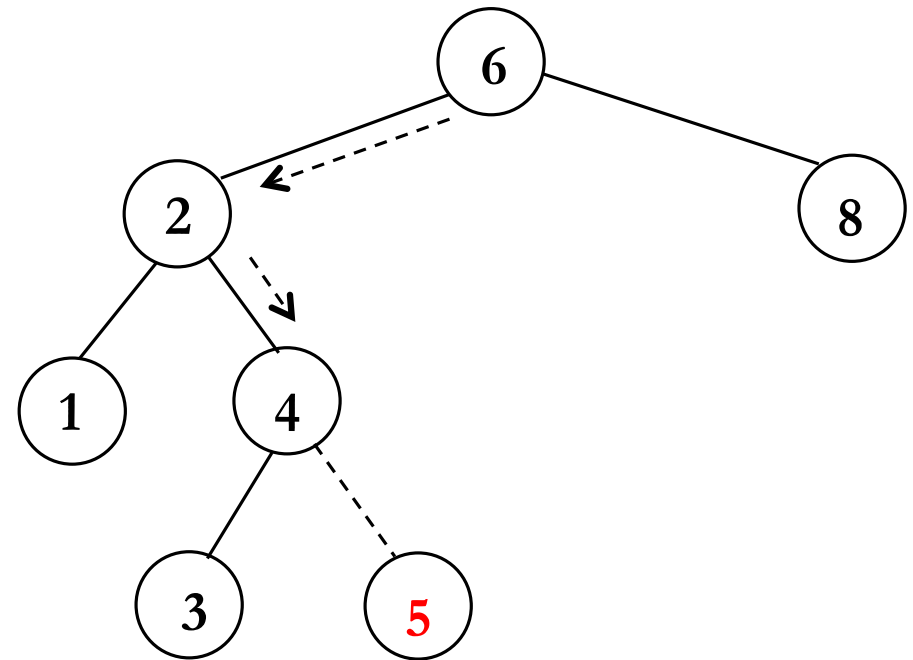
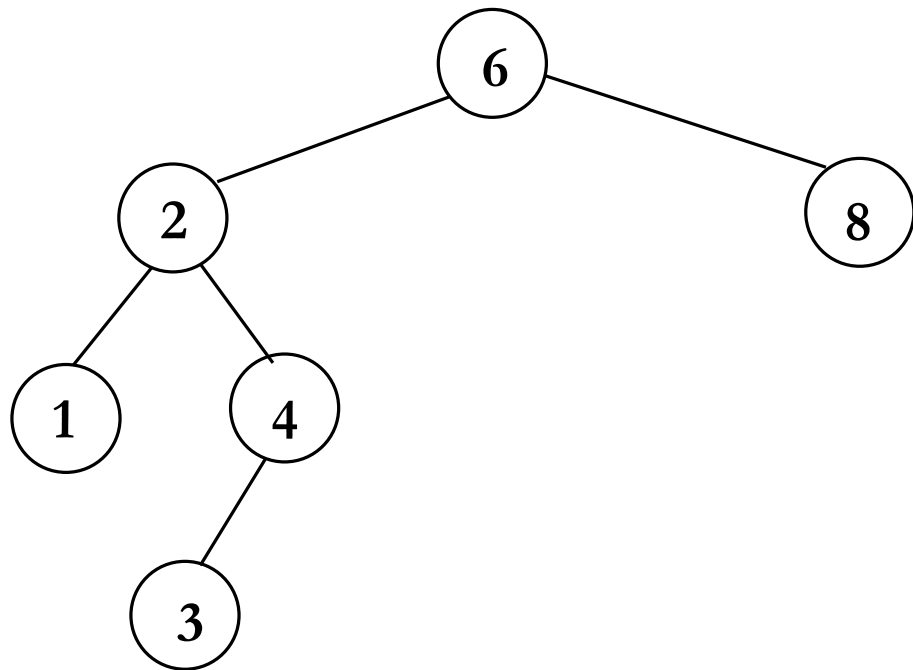
```
Node FindMin( SearchTree T )
{
    if( T == NULL )
        return NULL;

    else if ( T->Left == NULL )
        return T;

    else /* T->Left != NULL */
        return FindMin( T->Left );
}
```

Binary Search Tree: Insert

insertion of 5



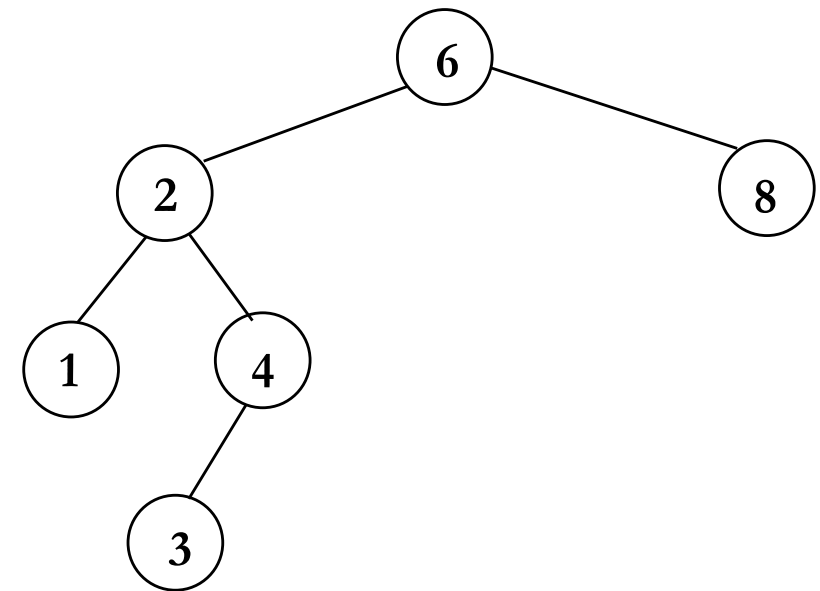
Binary Search Tree: Insert

```
SearchTree Insert ( ElementType X, SearchTree T )
{
    if( T == NULL ) {

        T = malloc( sizeof( struct TreeNode ) );
        if( T == NULL )
            FatalError( "Out of space!!!" );
        else
        {
            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 );

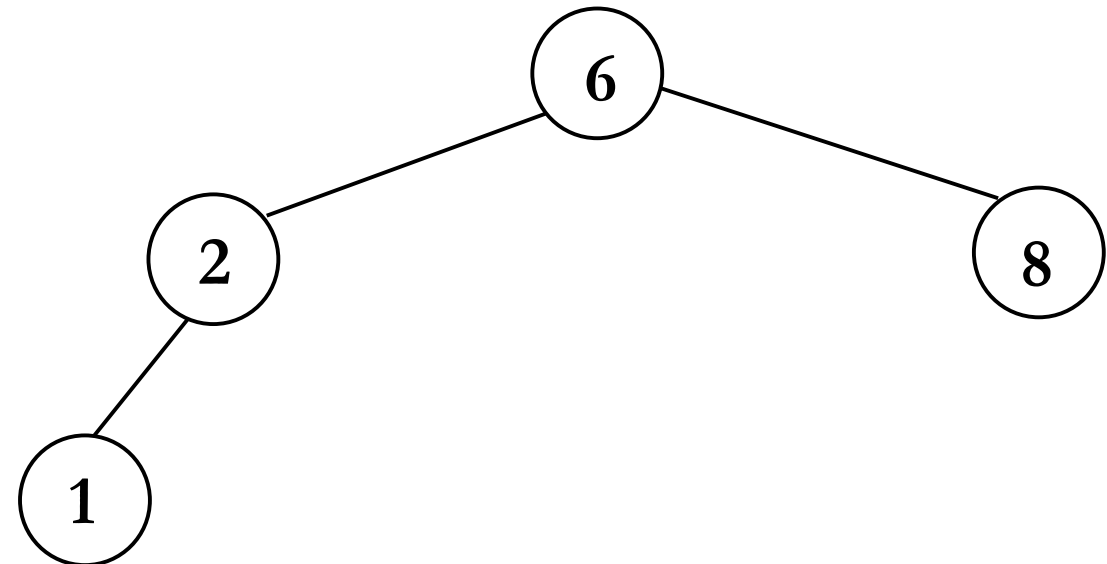
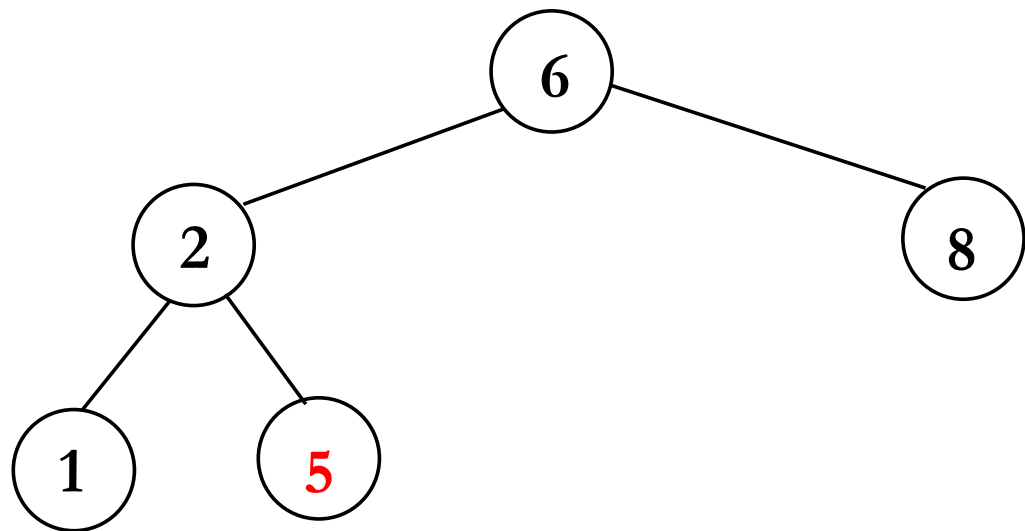
    /* Else X is in the tree already; we'll do nothing */

    return T; /* Do not forget this line! */
}
```



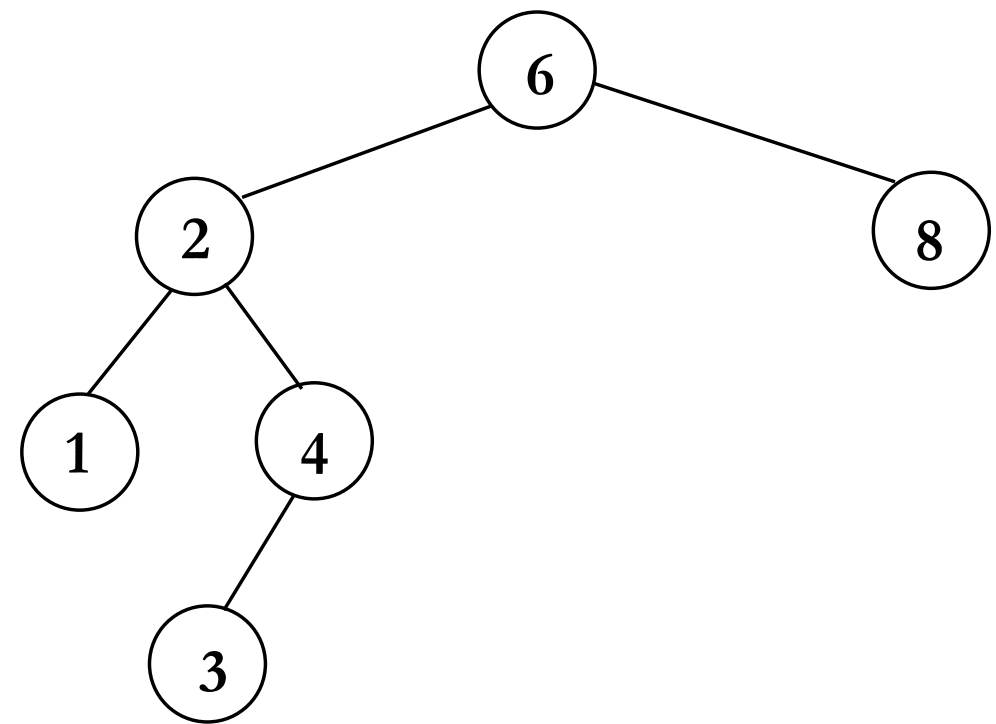
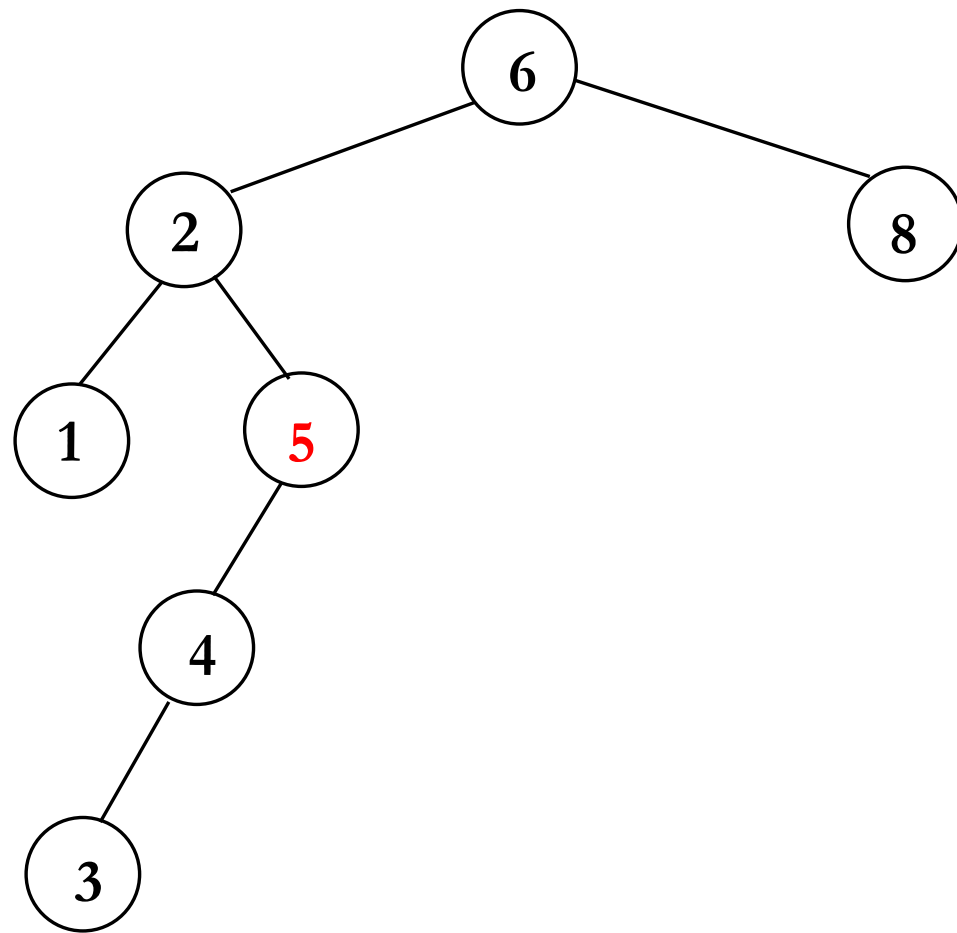
Binary Search Tree: Delete

If the node to be deleted is **a leaf**, just delete it!



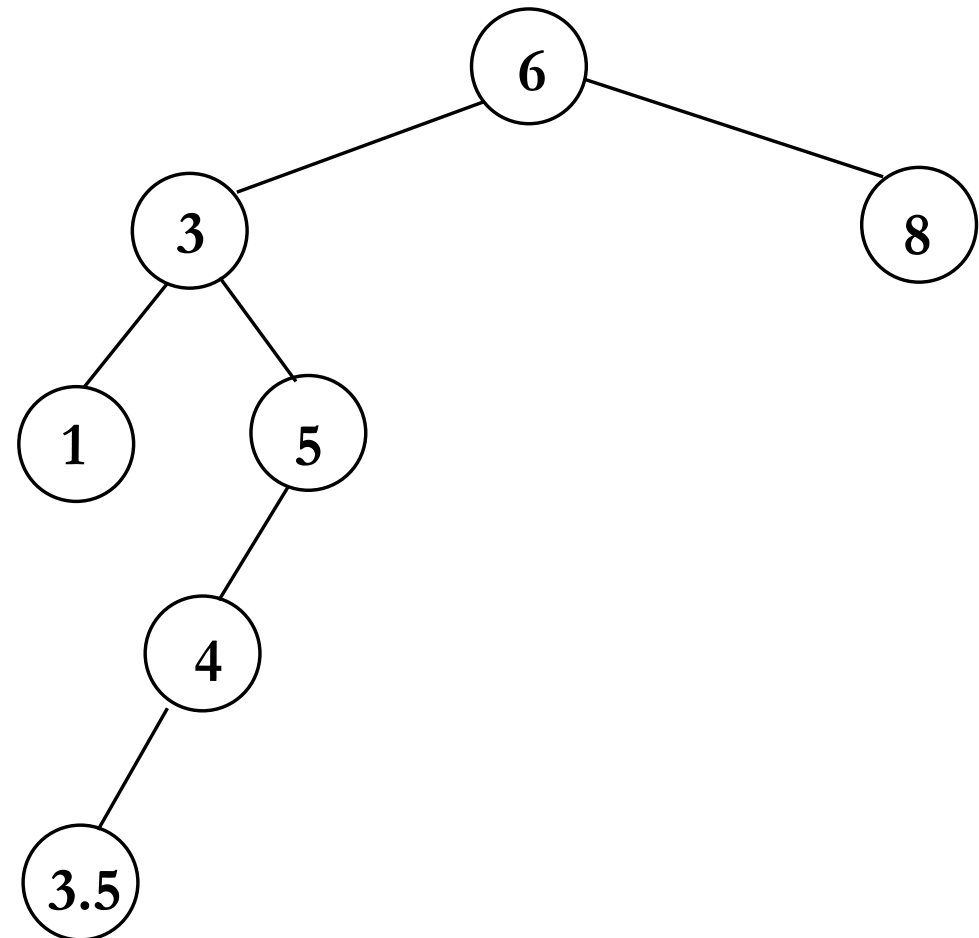
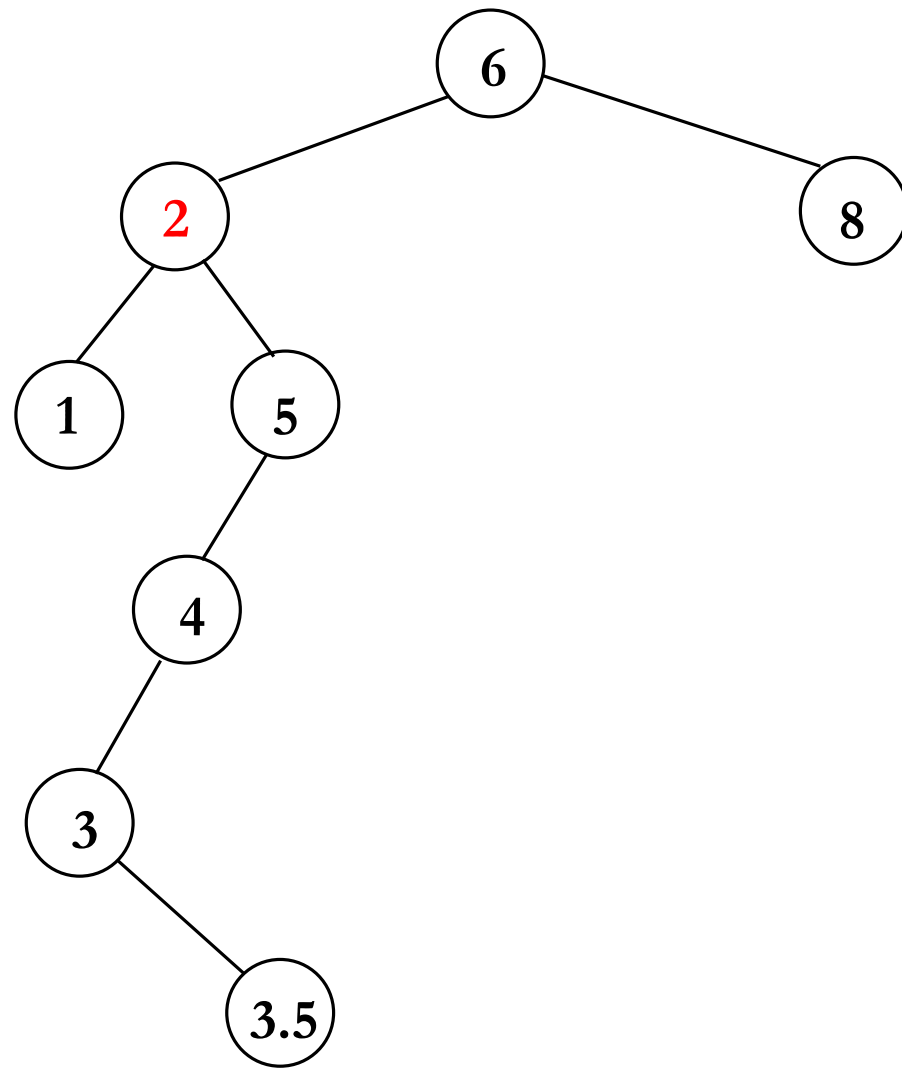
Binary Search Tree: Delete

If the node to be deleted has **one child**,
the child of the node is connected to the parent of the node



Binary Search Tree: Delete

If the node to be deleted has **both children**,
it is replaced with the smallest node in the right subtree.



Binary Search Tree: delete

```
SearchTree Delete( ElementType X, SearchTree T )
{
    Node TmpCell;

    if ( T == NULL )
        Error( "Element not found" );

    else if ( X < T->Element )      /* Go Left */
        T->Left = Delete( X, T->Left );

    else if ( X > T->Element )      /* Go Right */
        T->Right = Delete( X, T->Right );

    else if ( T->Left && T->Right ) { /* found the node to be deleted */

        TmpCell = FindMin( T->Right );
        T->Element = TmpCell->Element;
        T->Right = Delete( T->Element, T->Right );

    } else {                          /* 1 or 0 child */
        TmpCell = T;
        if( T->Left == NULL )
            T = T->Right;
        else if( T->Right == NULL )
            T = T->Left;
        free(TmpCell);
    }
    return T;
}
```


Binary Search Tree: average-case analysis

- **Internal Path Length** $D(N)$ is the sum of depths of all nodes in a binary search tree T with N nodes, which is $O(N \log N)$
 - An N -node tree consists of a root, an i -node left subtree and an $(N - i - 1)$ -node right subtree for $0 \leq i < N$
 - let $D(i)$ and $D(N - i - 1)$ are internal path lengths of the left and right subtree w.r.t. their roots, respectively.

$$D(0) = D(1) = 0$$

$$D(N) = D(i) + D(N - i - 1) + N - 1$$

for the edge from the root to the root of the left and right subtree

The average value of $D(i)$ (or $D(N - i - 1)$) is $\frac{1}{N} \sum_{j=0}^{N-1} D(j)$:

$$D(N) = \frac{2}{N} \sum_{j=0}^{N-1} D(j) + N - 1$$

Binary Search Tree: balanced binary tree

- non-random insertion to BST can produce unbalanced trees
- can we rebalance the tree so that the tree always has $O(\log n)$ height?
- We need *balance* information for each node.

