COMP2421—DATA STRUCTURES & ALGORITHMS

Binary Search Trees (BST)

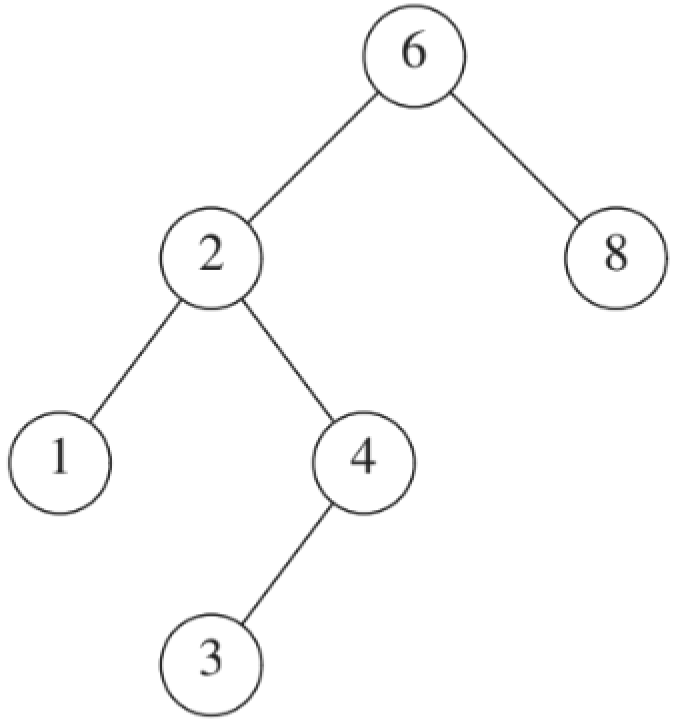
Dr. Radi Jarrar
Department of Computer Science
Birzeit University

Binary Search Trees

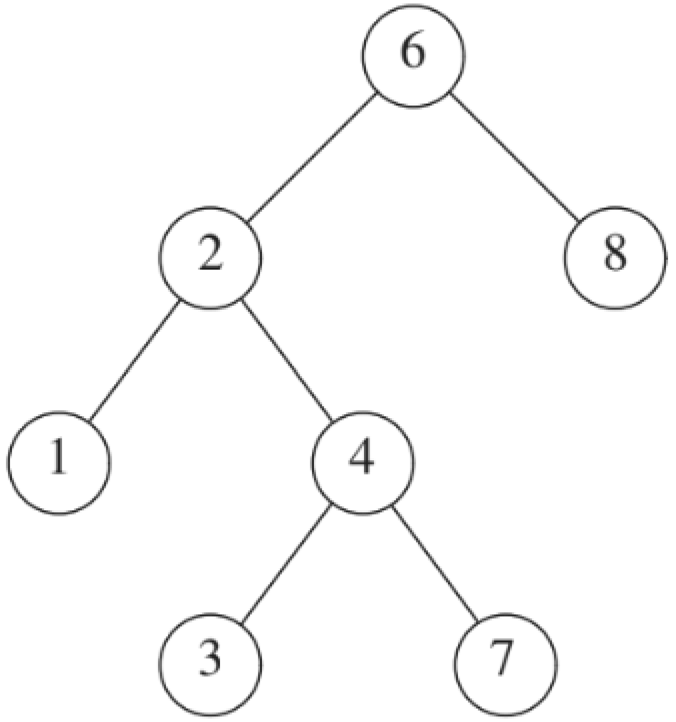
- An important application of Binary trees is their use of searching.
- Each node is assigned a key value. Assume the key is integer, and assume no duplicate keys (distinct keys).
- For every node X in the tree, the values of all the keys in its left subtree are smaller than the key value in X. And the values of all they keys in its right subtree are larger than the key value in X. This means that all elements in the tree can be ordered in some consistent manner.

Binary Search Trees (2)

- This means that all elements in the tree can be ordered in some consistent manner.



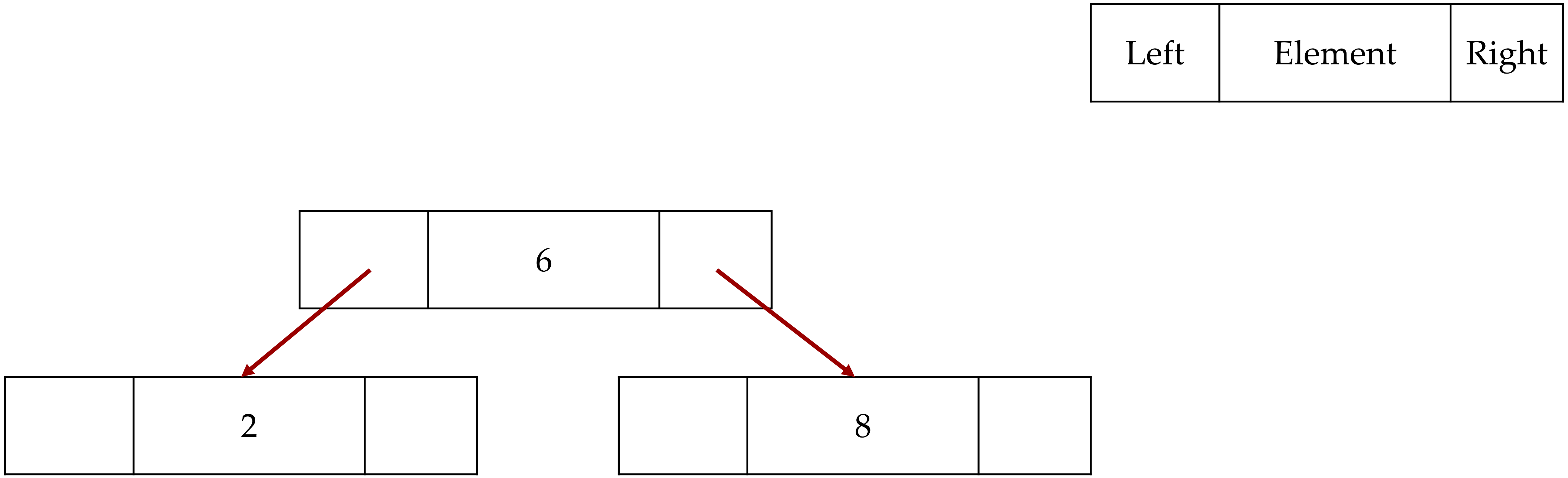
A Binary Search Tree



NOT a Binary Search Tree

Structure and Operations on BST

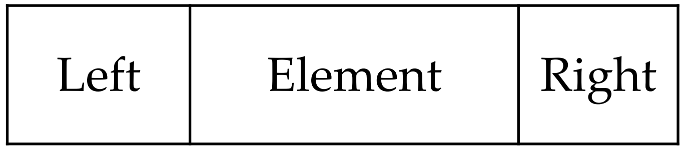
- The average depth of BST is O(log n).
- Implementation of BST using linked structure:



Struct

```
struct Node{
    int Element;
    struct Node* Left;
    struct Node* Right;
};

typedef struct Node* TNode;
```



MakeEmpty

```
//used to initialise a tree
TNode MakeEmpty( TNode T ){
    if( T != NULL ){
        MakeEmpty( T->Left );
        MakeEmpty( T->Right );
        free( T );
    }
    return NULL;
}
```

Find

- *returns a pointer to the node in tree T that has key X:*
  - if T is NULL, then return NULL;
  - if the KEY stored at T is X, then return T;
  - Otherwise, make a recursive call on a subtree of T, either left or right, depends on the relationship of X to the key stored in T (greater than or less than)

Find (2)

```
TNode Find( int X, TNode T ){
    if( T == NULL)
        return NULL;
    else if( X < T->element )
        return Find( X, T->Left );
    else if( X > T->element )
        return Find( X, T->Right );
    else
        return T;
}
```

FindMin & FindMax

- Return the position of the smallest and largest elements in the tree. They return position not the values (keys). This is to be consistent with the Find method.
- *FindMin*: start from the root, go left as long as there is a left child. The stopping point in the smallest element.

FindMin – Recursive Logic

```
//recursive implementation of the FindMin
TNode FindMin( TNode T ){
    if( T == NULL)
        return NULL;
    else if( T -> Left == NULL)
        return T;
    else
        return FindMin( T->Left );
}
```

FindMin – Iterative Logic

```
//non-recursive implementation of the FindMin

TNode FindMin( TNode T ){
    if( T != NULL)
        while( T->Left != NULL)
            T = T->Left;
    return T;
}
```

FindMax – Recursive Logic

- FindMax: the same, except you have to go to the right child.

```
TNode FindMax( TNode T ){
    if( T == NULL)
        return NULL;
    else if( T -> Right == NULL)
        return T;
    else
        return FindMax( T->Right);
}
```

FindMax – Iterative Logic

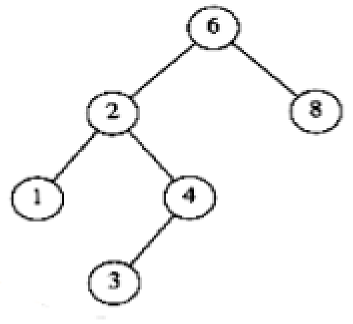
```
//non-recursive implementation of FindMax

TNode FindMax( TNode T ){
    if( T != NULL)
        while( T->Right != NULL)
            T = T->Right;
    return T;
}
```

Insert Routine

- To insert X into tree T, proceed down the tree as you would with a FIND.
- If X is found, do nothing (or update, duplicates are handled by keeping an extra field in the node record indicating the frequency of occurrence).
- Otherwise (X is not found), insert X at the last spot on the path traversed.

Insert Routine



```
TNode Insert( int X, TNode T ){
     if( T == NULL){
          //create and return a 1-node tree
          T = (struct Node*)malloc( sizeof( struct Node ) );

          if( T == NULL)
               printf("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; do nothing

     return T; }
```

Delete

- The hardest operation – there are several possibilities (scenarios) to consider once a node is found to be deleted.
  1. If the node is **leaf**, it can be deleted immediately;
  2. If the node **has one child**, the node can be deleted after its parents adjust a pointer to bypass the node (draw the pointer directions explicitly for clarity as below);

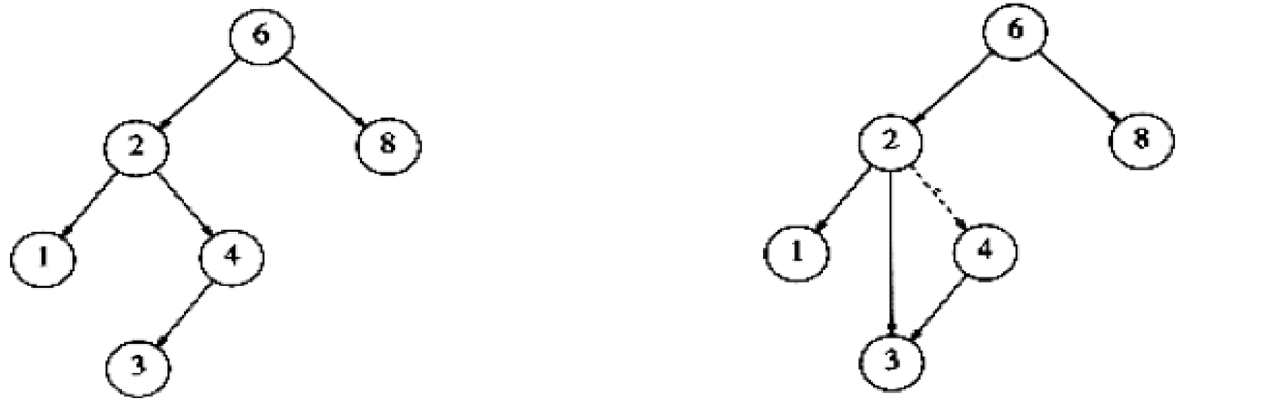


Figure 4.23 Deletion of a node (4) with one child, before and after

Delete (2)

3. Deleting a **node with two children**, steps:

a) replace the key of this node with the smallest key of the right subtree (which is easily found)

b) recursively delete that node (which is now empty)

- Because the smallest node in the right subtree cannot have a left child, the second delete is an easy one (i.e., a leaf node). The other case is that a node will have one child which is case#2.

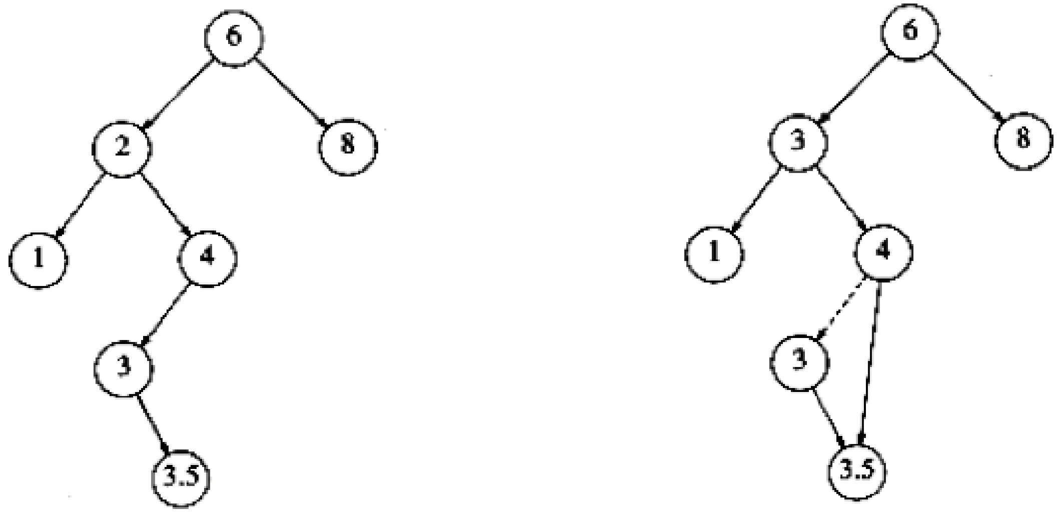


Figure 4.24 Deletion of a node (2) with two children, before and after

Delete (2)

```
TNode Delete( int X, TNode T )
{
    TNode TmpCell;

    if( T == NULL )
        printf( "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  /* Found element to be deleted */
        if( T->Left && T->Right )
        /* Two children */
        {
        /* Replace with smallest in right
subtree */
            TmpCell = FindMin( T->Right );
            T->Element = TmpCell->Element;
            T->Right = Delete( T->Element, T-
>Right );
        }
        else  /* One or zero children */
        {
            TmpCell = T;

            if( T->Left == NULL )
        /* Also handles 0 children */
                T = T->Right;
            else if( T->Right == NULL )
                T = T->Left;

            free( TmpCell );
        }

    return T;
} //end of Delete routine
```

Time Analysis

- The previous operations should take O(log n) time except MakeEmpty.
- This is because we descend a level in the tree in a constant time.
- Thus we are operating on a tree that is roughly half large.