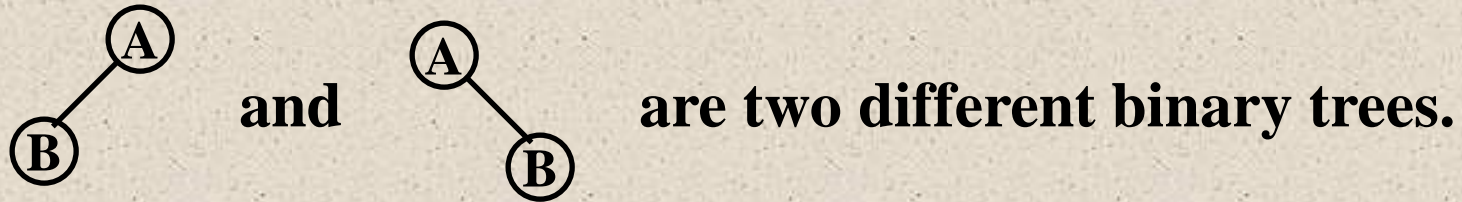
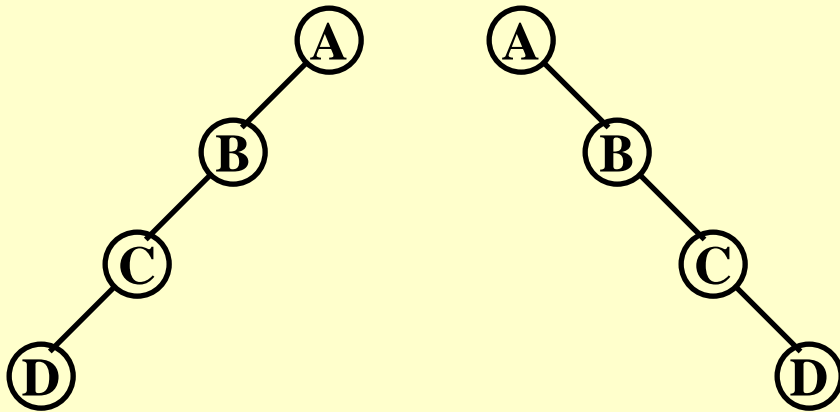


Note: In a tree, the order of children does not matter. But in a binary tree, left child and right child are different.

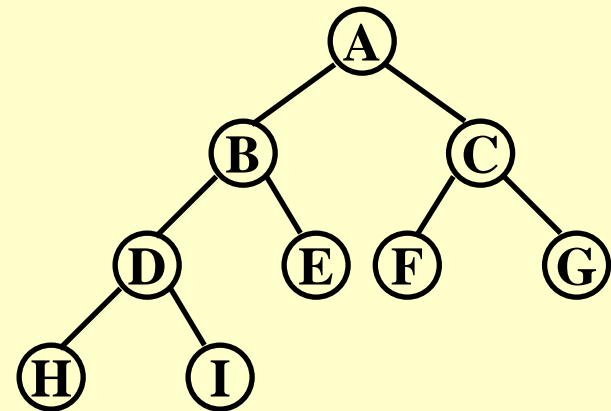


Skewed Binary Trees



Skewed to the left **Skewed to the right**

Complete Binary Tree



All the leaf nodes are on two adjacent levels

☞ Properties of Binary Trees

- ☑ The maximum number of nodes on level i is 2^{i-1} , $i \geq 1$.
The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.
- ☑ For any nonempty binary tree, $n_0 = n_2 + 1$ where n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2.

Proof: Let n_1 be the number of nodes of degree 1, and n the total number of nodes. Then

$$n = n_0 + n_1 + n_2 \quad \textcircled{1}$$

Let B be the number of branches. Then $n = B + 1$. $\textcircled{2}$

Since all branches come out of nodes of degree 1 or 2, we have $B = n_1 + 2 n_2$. $\textcircled{3}$

$$\Rightarrow n_0 = n_2 + 1$$

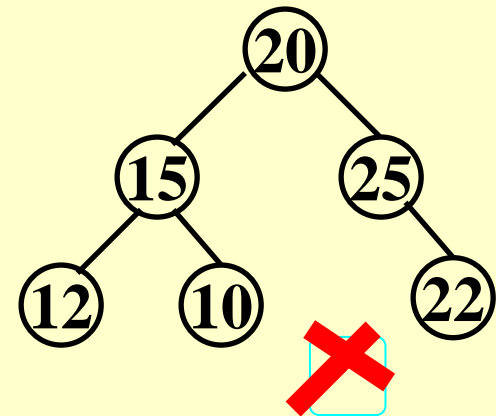
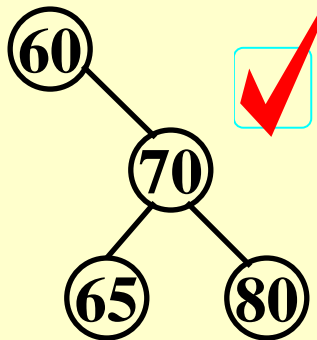
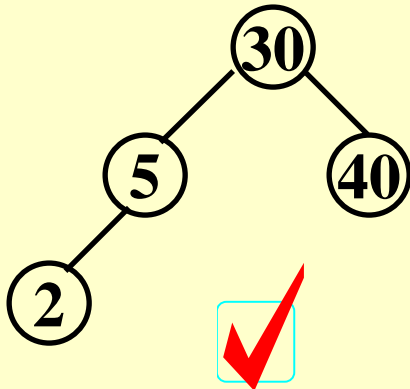


§ 3 The Search Tree ADT -- Binary Search Trees

1. Definition

【Definition】 A **binary search tree** is a binary tree. It may be empty. If it is not empty, it satisfies the following properties:

- (1) Every node has a **key** which is an **integer**, and the keys are **distinct**.
- (2) The keys in a nonempty **left** subtree must be **smaller** than the key in the root of the subtree.
- (3) The keys in a nonempty **right** subtree must be **larger** than the key in the root of the subtree.
- (4) The left and right subtrees are also binary search trees.



2. ADT

Objects: A finite ordered list with zero or more elements.

Operations:

☞ **SearchTree** **MakeEmpty**(SearchTree T);

☞ **Position** **Find**(ElementType X, SearchTree T);

☞ **Position** **FindMin**(SearchTree T);

☞ **Position** **FindMax**(SearchTree T);

☞ **SearchTree** **Insert**(ElementType X, SearchTree T);

☞ **SearchTree** **Delete**(ElementType X, SearchTree T);

☞ **ElementType** **Retrieve**(Position P);

3. Implementations

👉 Find

Must this test
be performed first?

```
Position Find( ElementType X, SearchTree T )
```

```
{
```

```
    if ( T == NULL )
```

```
        return NULL; /* not found in an empty tree */
```

```
    if ( X < T->Element ) /* if smaller than root */
```

```
        return Find( X, T->Left ); /* search left subtree */
```

```
    else
```

```
        if ( X > T->Element ) /* if larger than root */
```

```
            return Find( X, T->Right ); /* search right subtree */
```

```
        else /* if X == root */
```

```
            return T; /* found */
```

```
}
```

These are
tail recursions.

$T(N) = S(N) = O(d)$ where d is the depth of X

```
Position Iter_Find( ElementType X, SearchTree T )
{
    /* iterative version of Find */
    while ( T ) {
        if ( X == T->Element )
            return T ; /* found */
        if ( X < T->Element )
            T = T->Left ; /*move down along left path */
        else
            T = T->Right ; /* move down along right path */
    } /* end while-loop */
    return NULL ; /* not found */
}
```

👉 FindMin

Position FindMin(SearchTree T)

$$T(N) = O(d)$$

```
{
    if ( T == NULL )
        return NULL; /* not found in an empty tree */
    else
        if ( T->Left == NULL ) return T; /* found left most */
        else return FindMin( T->Left ); /* keep moving to left */
}
```

👉 FindMax

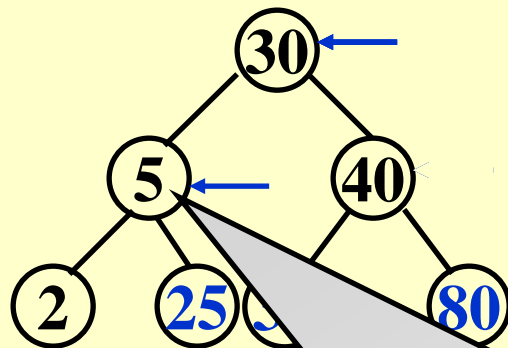
Position FindMax(SearchTree T)

$$T(N) = O(d)$$

```
{
    if ( T != NULL )
        while ( T->Right != NULL )
            T = T->Right; /* keep moving to find right most */
    return T; /* return NULL or the right most */
}
```

👉 Insert

Sketch of the idea:



Insert **80**

① check if **80** is already in the tree

② **80** > 40, so it must be the right child of 40

Insert **35**

Insert **25**

This is the last node we encounter when search for the key number. It will be the parent of the new node.

② **25** > 5, so it must be the right child of 5


```

SearchTree Insert( ElementType X, SearchTree T )
{
    if ( T == NULL ) { /* Create and return a one-node tree */
        T = malloc( sizeof( struct TreeNode ) );
        if ( T == NULL )
            FatalError( "Out of space!!!" );
        else {
            T->Element = X;
            T->Left = T->Right = NULL; }
    } /* End creating a one-node tree */
    else /* If there is a tree */
        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!! */
}

```

How would you
Handle duplicated
Keys?

👉 Delete

❖ Delete a leaf node.

❖ Delete a degree 1 node.

❖ Delete a degree 2 node.

Note: These kinds of nodes have degree at most 1.

NULL.

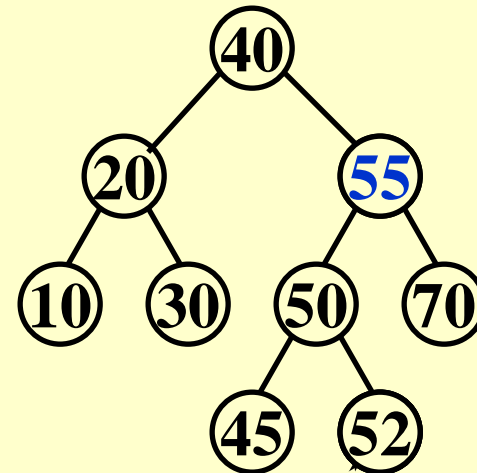
its single child.

- ① Replace the node by the **largest** one in its **left** subtree or the **smallest** one in its **right** subtree.
- ② Delete the replacing node from the subtree.

[[Example]] Delete 60

Solution 1: reset left subtree.

Solution 2: reset right subtree.



```

SearchTree Delete( ElementType X, SearchTree T )
{
    Position 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 /* 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 ); } /* End if */
        else { /* One or zero child */
            TmpCell = T;
            if ( T->Left == NULL ) /* Also handles 0 child */
                T = T->Right;
            else if ( T->Right == NULL ) T = T->Left;
            free( TmpCell ); } /* End else 1 or 0 child */

    return T;
}

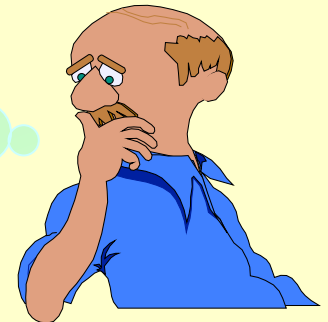
```

$T(N) = O(h)$ where h is the height of the tree

Note:

If there are not many deletions, then *lazy deletion* may be employed: add a flag field to each node, to **mark** if a node is active or is deleted. Therefore we can delete a node without actually freeing the space of that node. If a deleted key is reinserted, we won't have to call malloc again.

While the number of deleted nodes is the same as the number of active nodes in the tree, will it seriously affect the efficiency of the operations?



4. Average-Case Analysis

Question: Place n elements in a binary search tree. How high can this tree be?

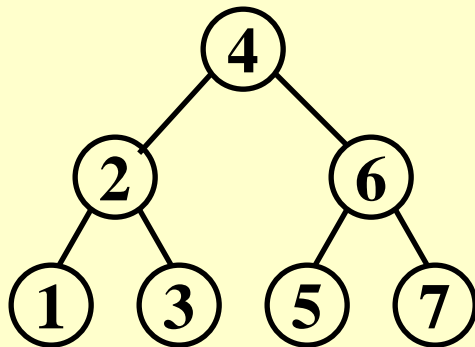
Answer: The height depends on the order of insertion.

[[Example]] Given elements 1, 2, 3, 4, 5, 6, 7. Insert them into a binary search tree in the orders:

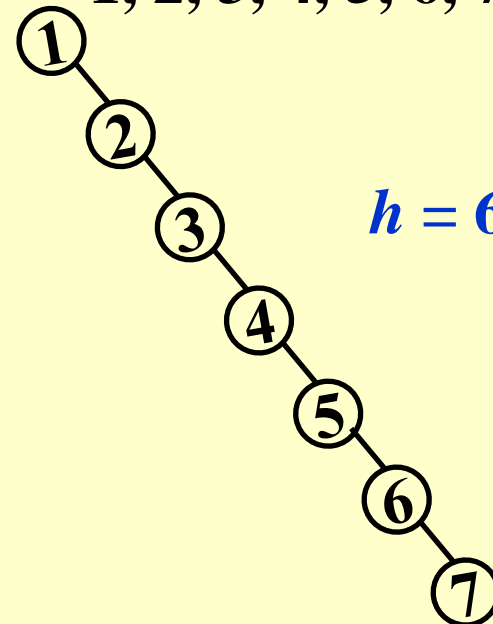
4, 2, 1, 3, 6, 5, 7

and

1, 2, 3, 4, 5, 6, 7



$h = 2$



$h = 6$



Laboratory Project 2

Normal: $A+B$ with Binary Search Trees

Hard: Autograd for Algebraic Expressions

Due: Monday, April 1st, 2024 at 10:00pm