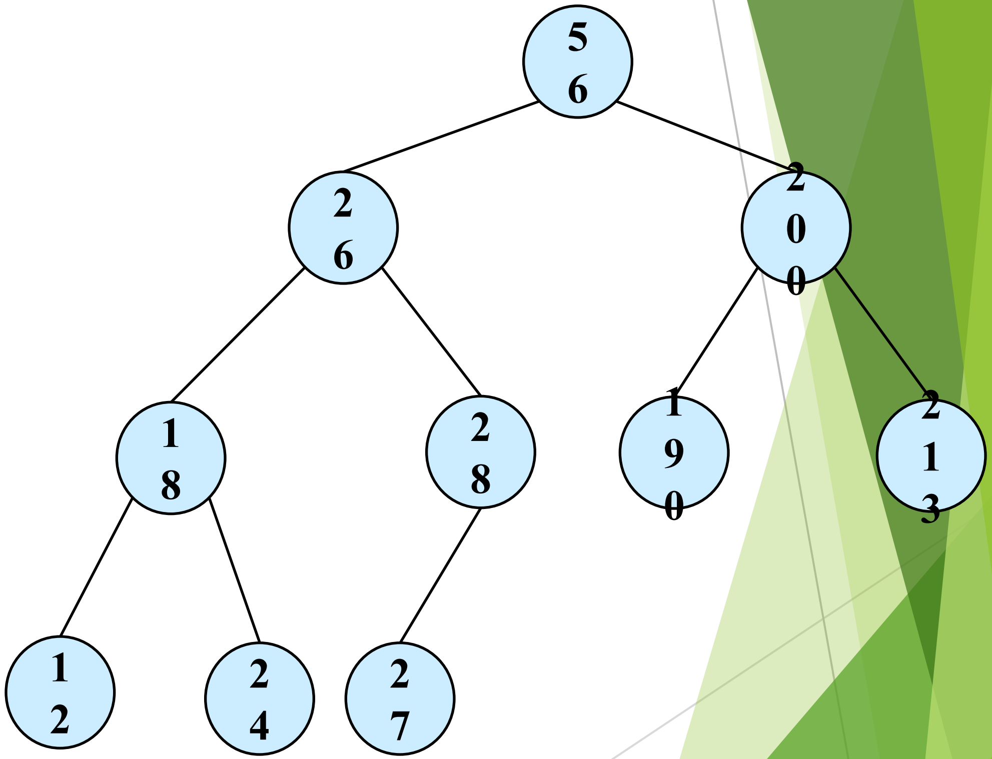
Binary Search Tree

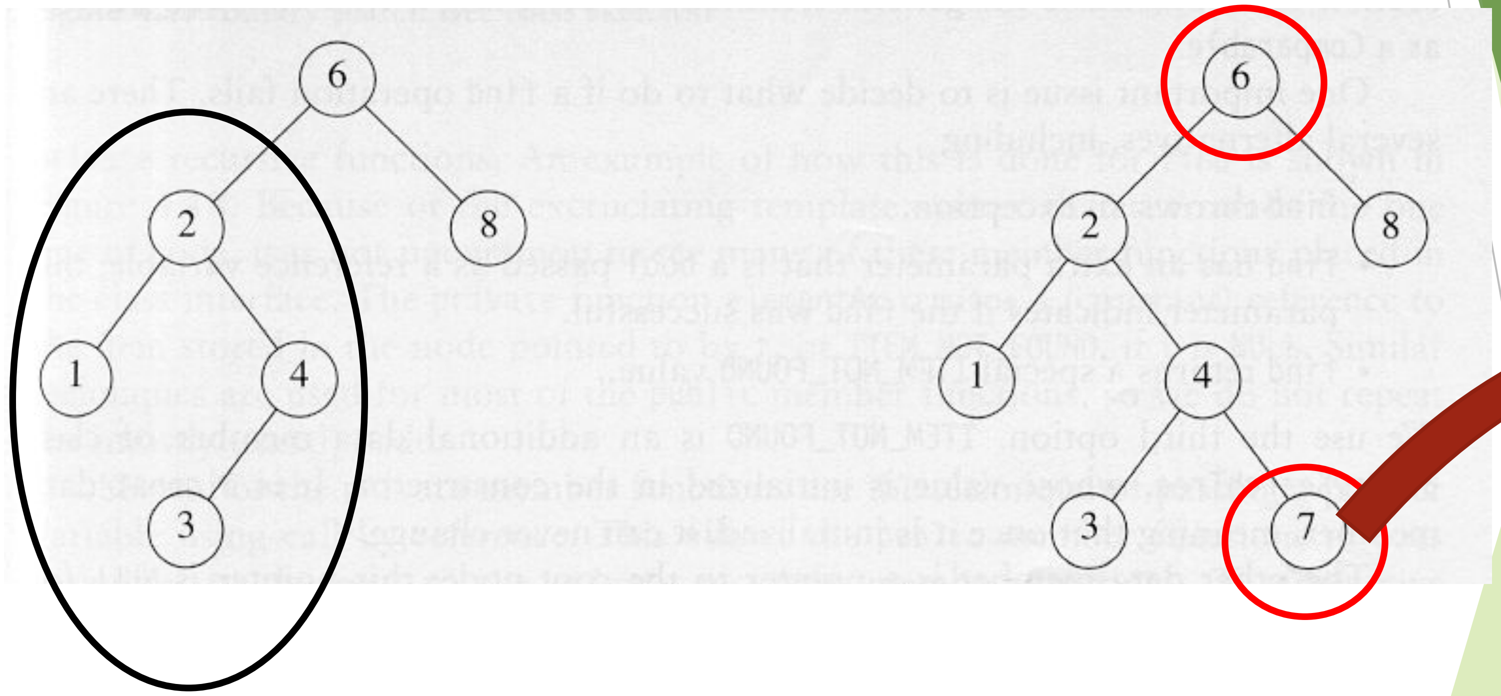
Dr. LALATENDU BEHERA

Binary Search Tree Property

- This is a binary tree such that
- Stored keys must satisfy the *binary search tree* property.
  - ∀ *y* in left subtree of *x*, then *key*[*y*] ≤ *key*[*x*].
  - ∀ *y* in right subtree of *x*, then *key*[*y*] ≥ *key*[*x*].



Binary Search Trees



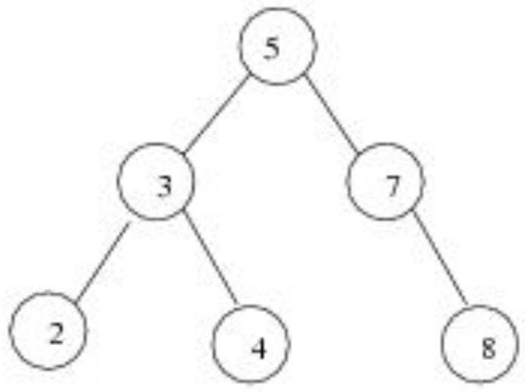
A binary search tree

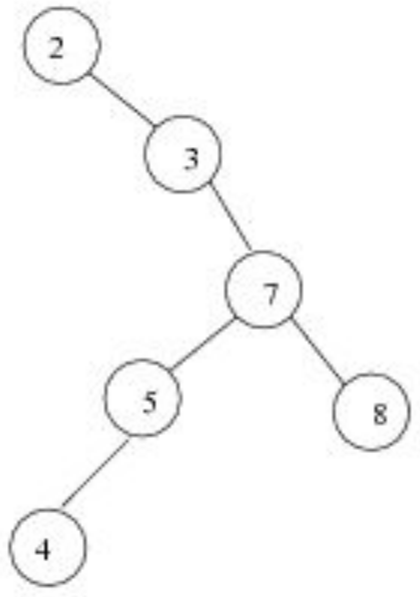
Not a binary search tree

7 > 6 which violates the property ∀ *y* in left subtree of *x*, then *key*[*y*] ≤ *key*[*x*].

BST

Two binary search trees representing the same set:



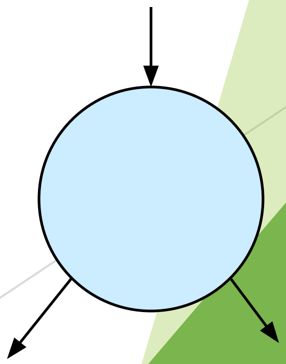


- Average height is O(log N);
- maximum height is O(N)

BST – Representation

- Represented by a linked data structure of nodes.
- *root*(*T*) points to the root of tree *T*.

- Each node contains fields:
  - *key*
  - *left* – pointer to left child: root of left subtree.
  - *right* – pointer to right child : root of right subtree.



A structure representation of BST in C

```
Struct Tree
{
int data;
Struct Tree *left;
Struct Tree *right;
};
```

Binary Search Trees

- Operations On BST.
  - Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete.
- Can be used to build
  - Dictionaries.
  - Priority Queues.

Basic Operations

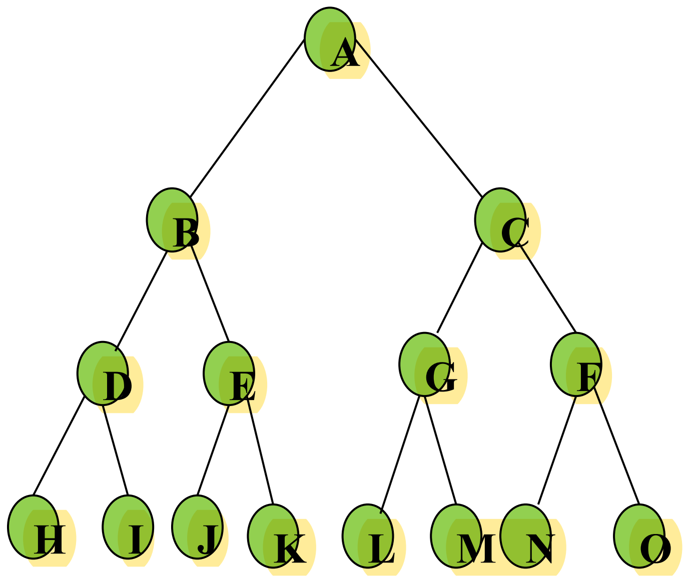
- ✔ Search - compare the values and proceed either to the left or to the right
- ✔ Insertion - unsuccessful search - insert the new node at the bottom where the search has stopped
- ✔ Deletion - replace the value in the node with the smallest value in the right subtree or the largest value in the left subtree.
- ✔ Retrieval in sorted order – inorder traversal

Complexity

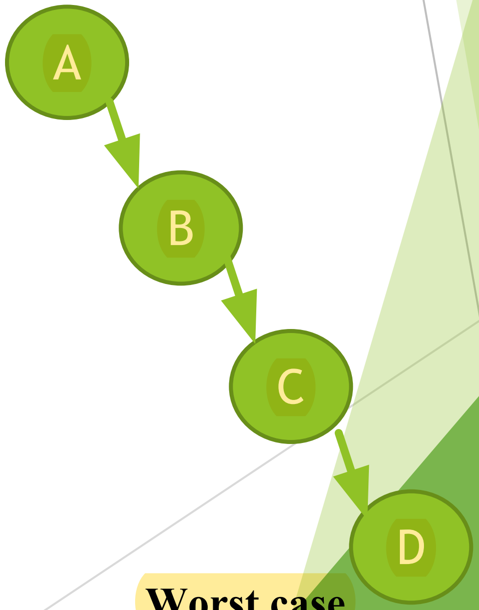
Logarithmic, depends on the shape of the tree

In the best case – O(lgN) comparisons

In the worst case – O(N) comparisons



Best case

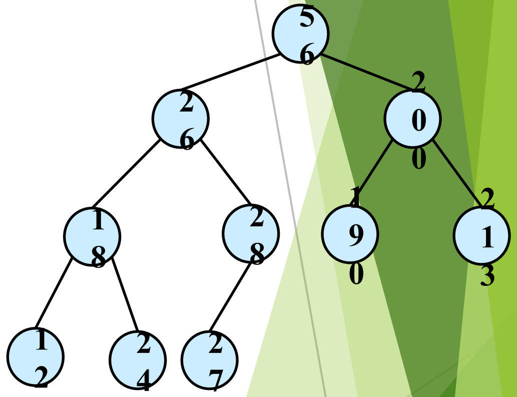


Worst case

Inorder Traversal

The binary-search-tree property allows the keys of a binary search tree to be printed, in (monotonically increasing) order, recursively.

```
Inorder-Tree-Walk (x)
1. if x ≠ NIL
2.     then Inorder-Tree-Walk(left[p])
3.          print key[x]
4.     Inorder-Tree-Walk(right[p])
```



- **How long does the walk take?**

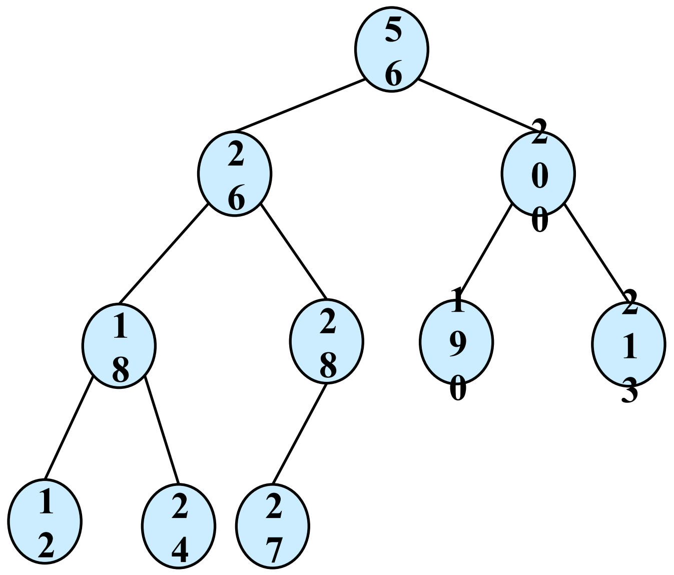
Querying a Binary Search Tree

- All dynamic-set search operations can be supported in $O(h)$ time.

- $h = \Theta(lg\ n)$ for a balanced binary tree (and for an average tree built by adding nodes in random order.)

- $h = \Theta(n)$ for an unbalanced tree that resembles a linear chain of n nodes in the worst case.

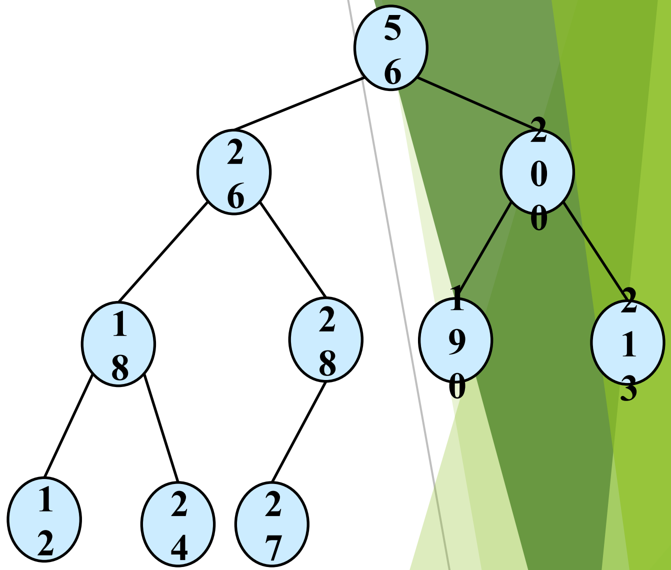
Tree Search



Running time: *O*(*h*)

Tree Search

```
struct Tree *Find(struct Tree *root, int data )
{
If(root ==Null)
return Null;
If (data<root->data)
return (Find ( root->left, data));
else if ( data>root->data)
return (Find ( root->right, data));
return root;
}
```



Finding Min & Max

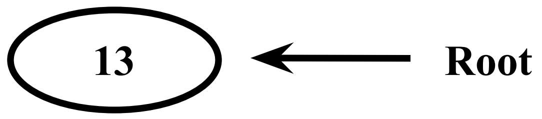
- The binary-search-tree property guarantees that:
  - » The minimum is located at the left-most node.
  - » The maximum is located at the right-most node.

```
Struct Tree *FindMin(Struct Tree
*root)
{
if(root ==Null)
return Null;
else If (root->left==Null)
return root;
else return FindMin(root->left);
}
```

```
Struct Tree *FindMax(Struct Tree
*root)
{
If(root ==Null)
Return Null;
Else If (root->right==Null)
Return root;
Else return FindMax(root->right);
}
```

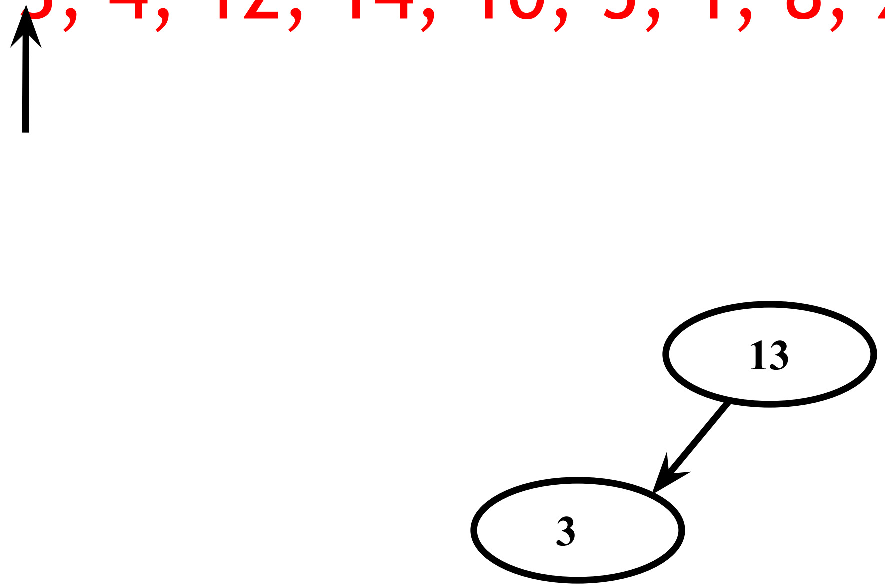
Insertion in a Binary Search Tree

13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18

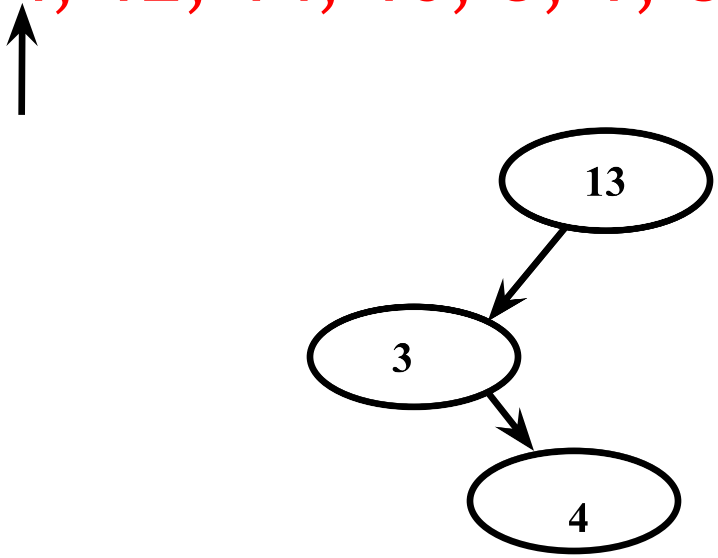


- You have to remember the property of the binary search tree and add the element at the position depending on its value.
- We have to start from the root node. If the key to be inserted is less than the root node, then the key will be inserted at the left subtree else at the right subtree. Then we have to do the same comparison in the subtrees as well. We have to do it until we get the position to insert the key.

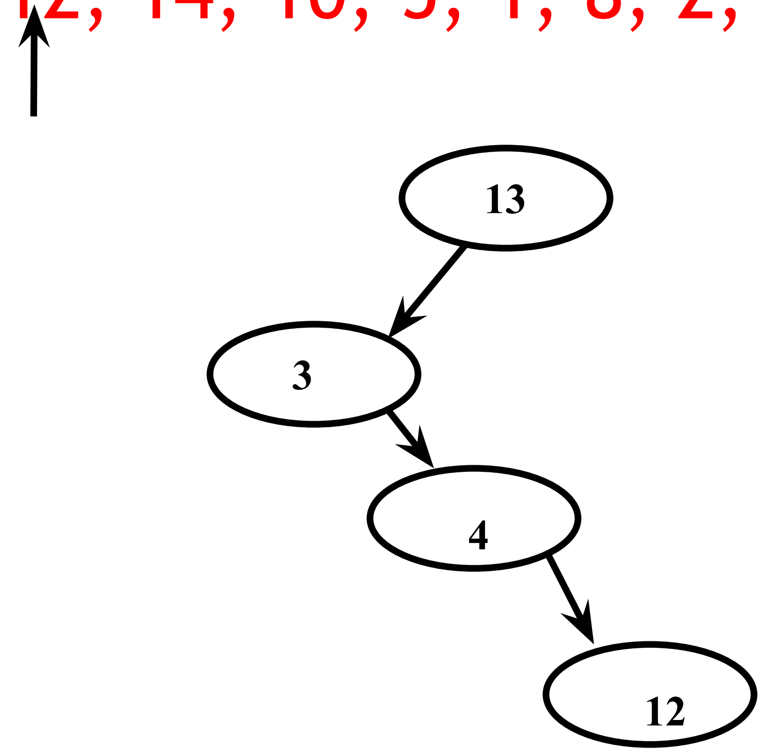
13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



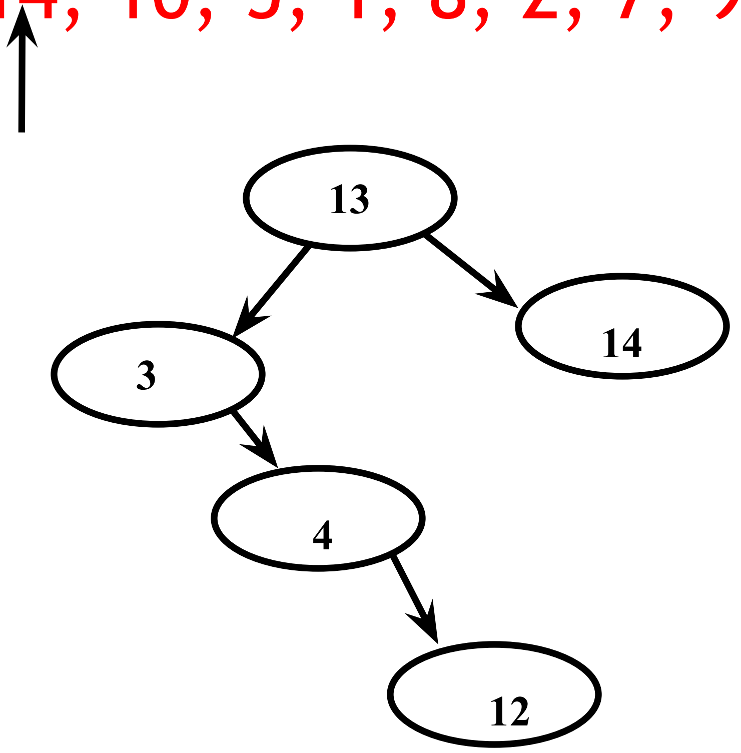
13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



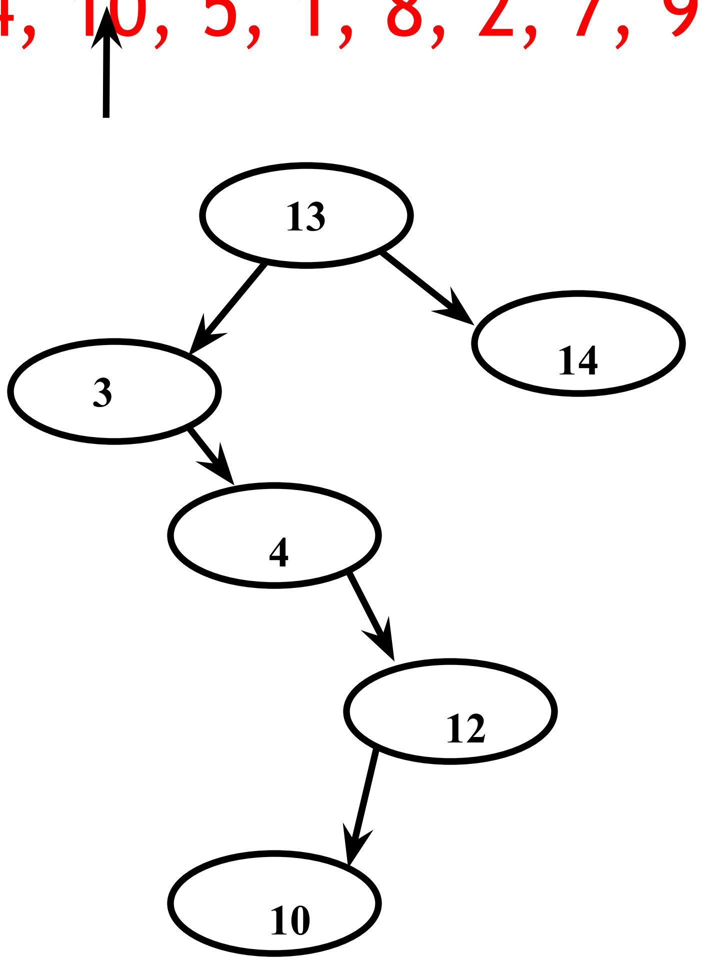
13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



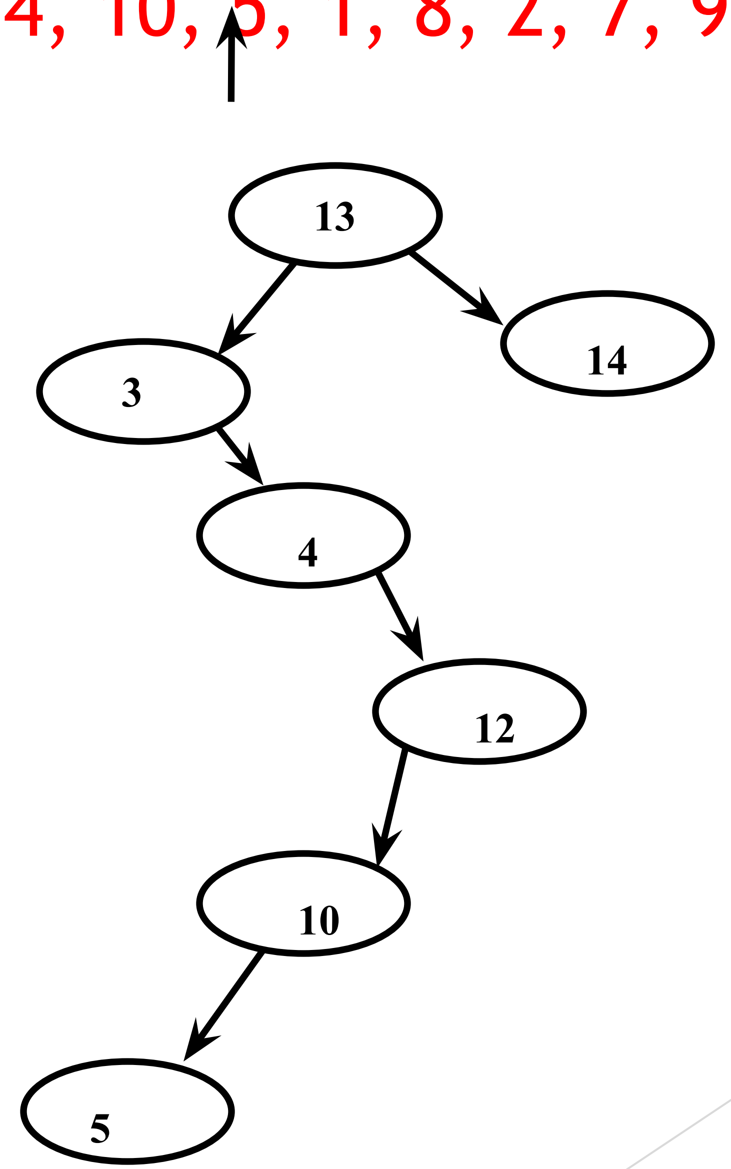
13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



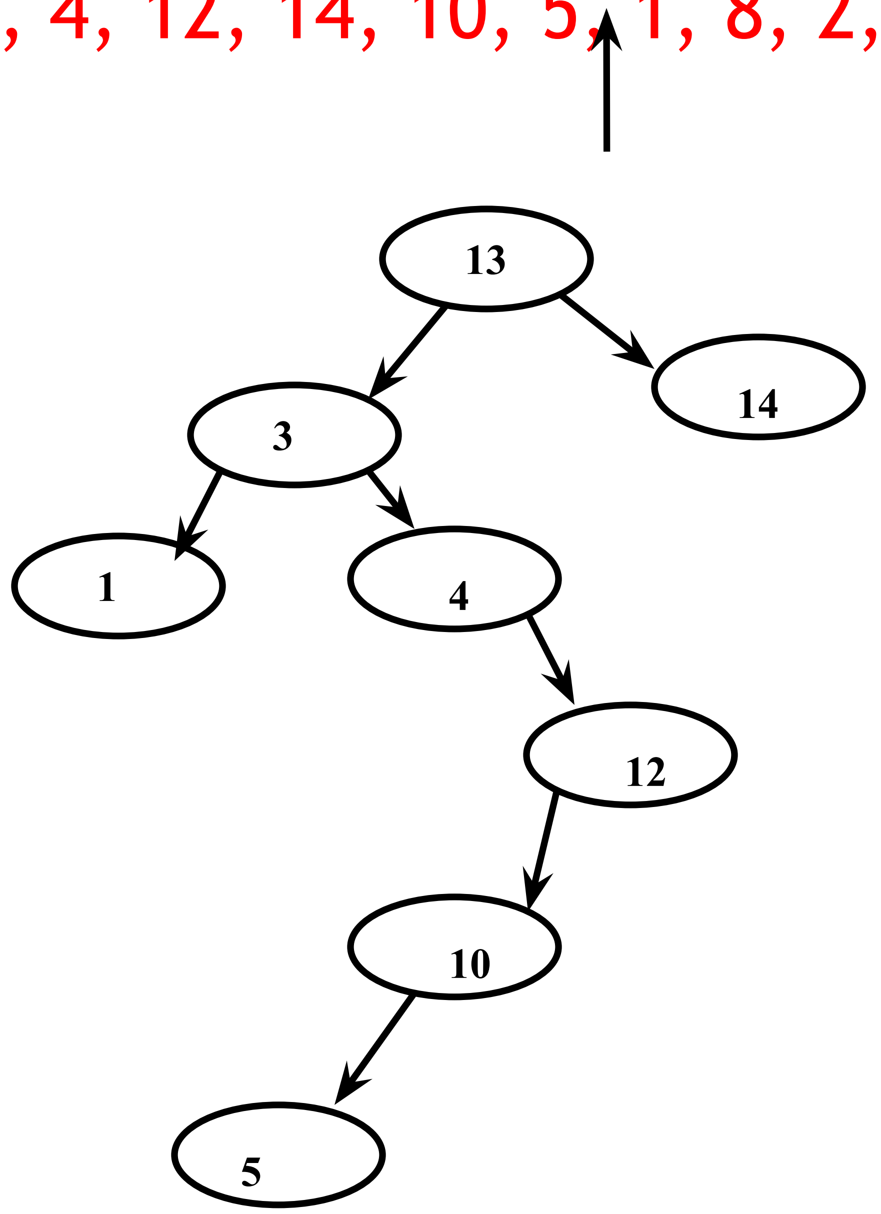
13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



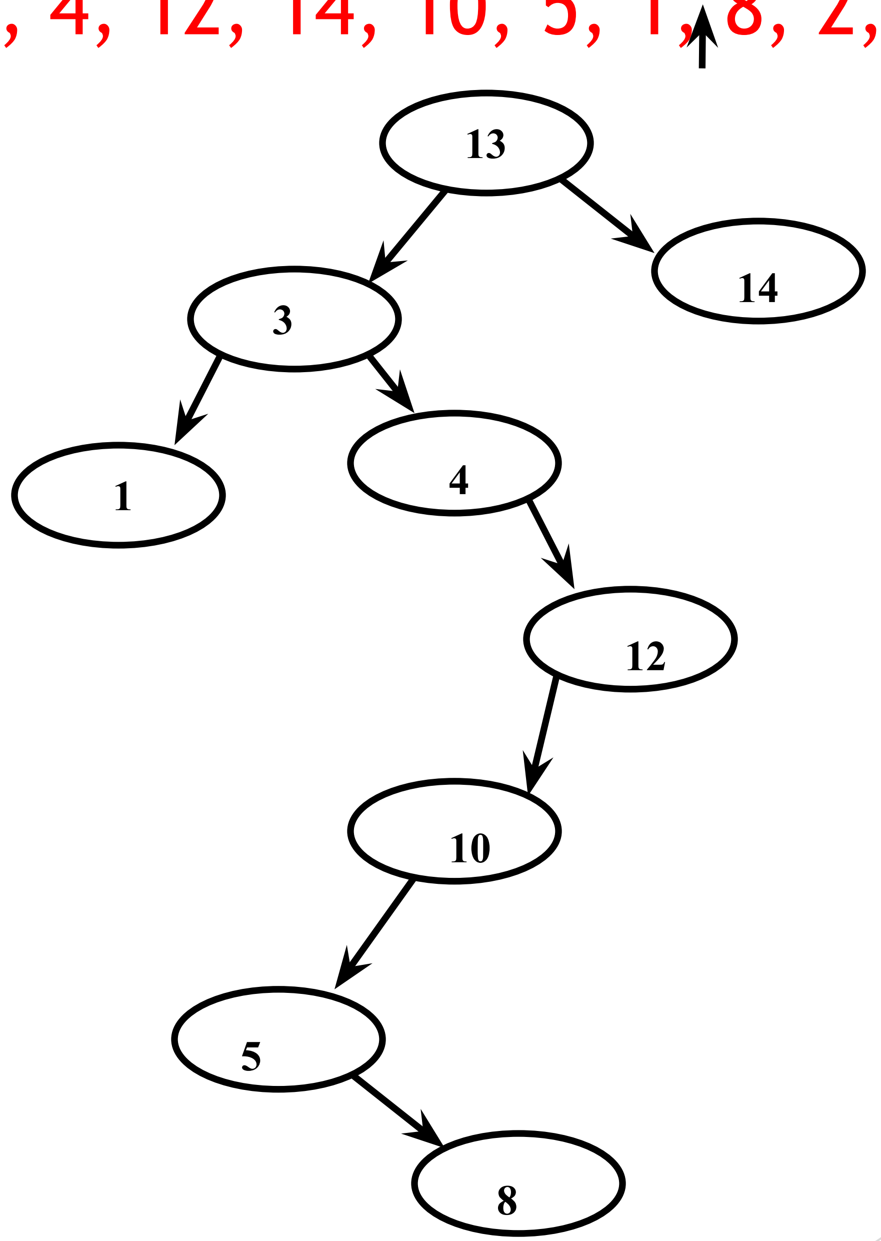
13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



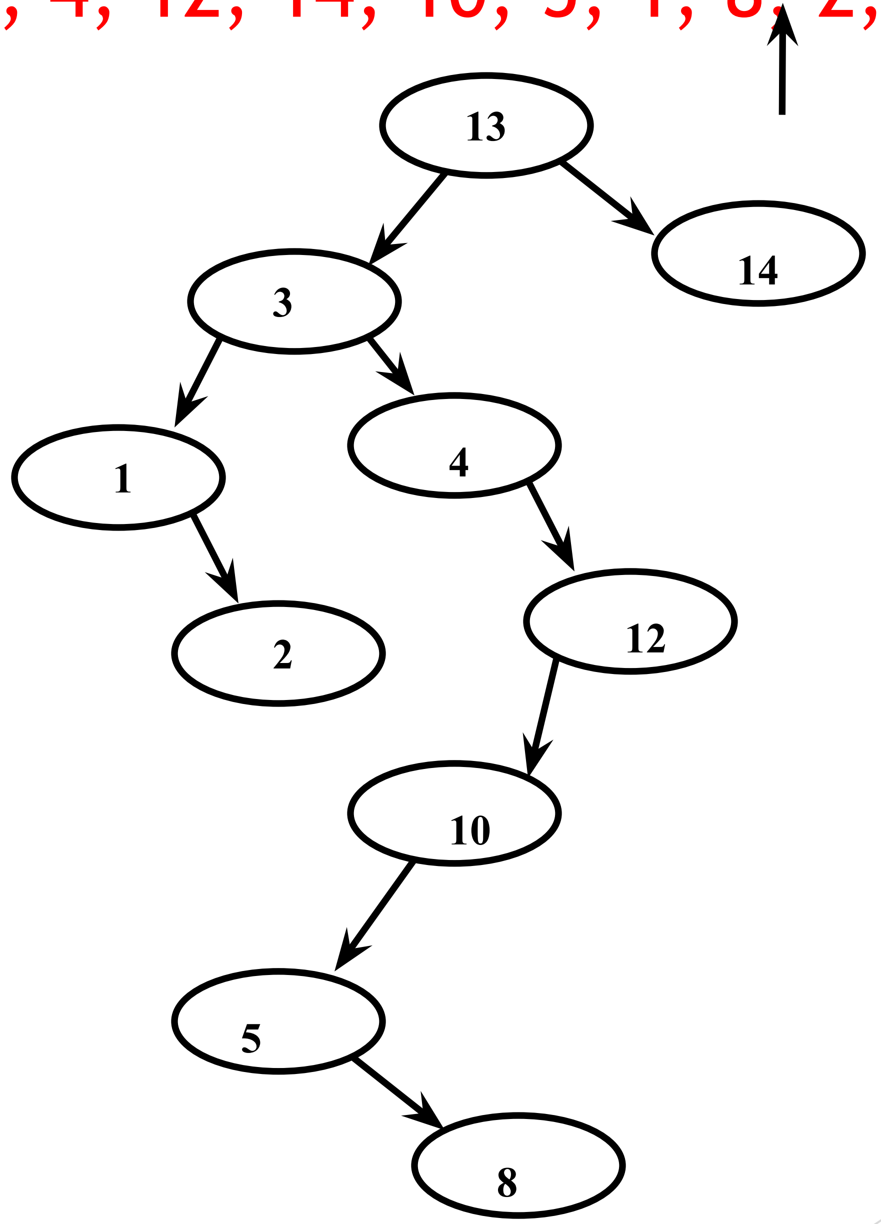
13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



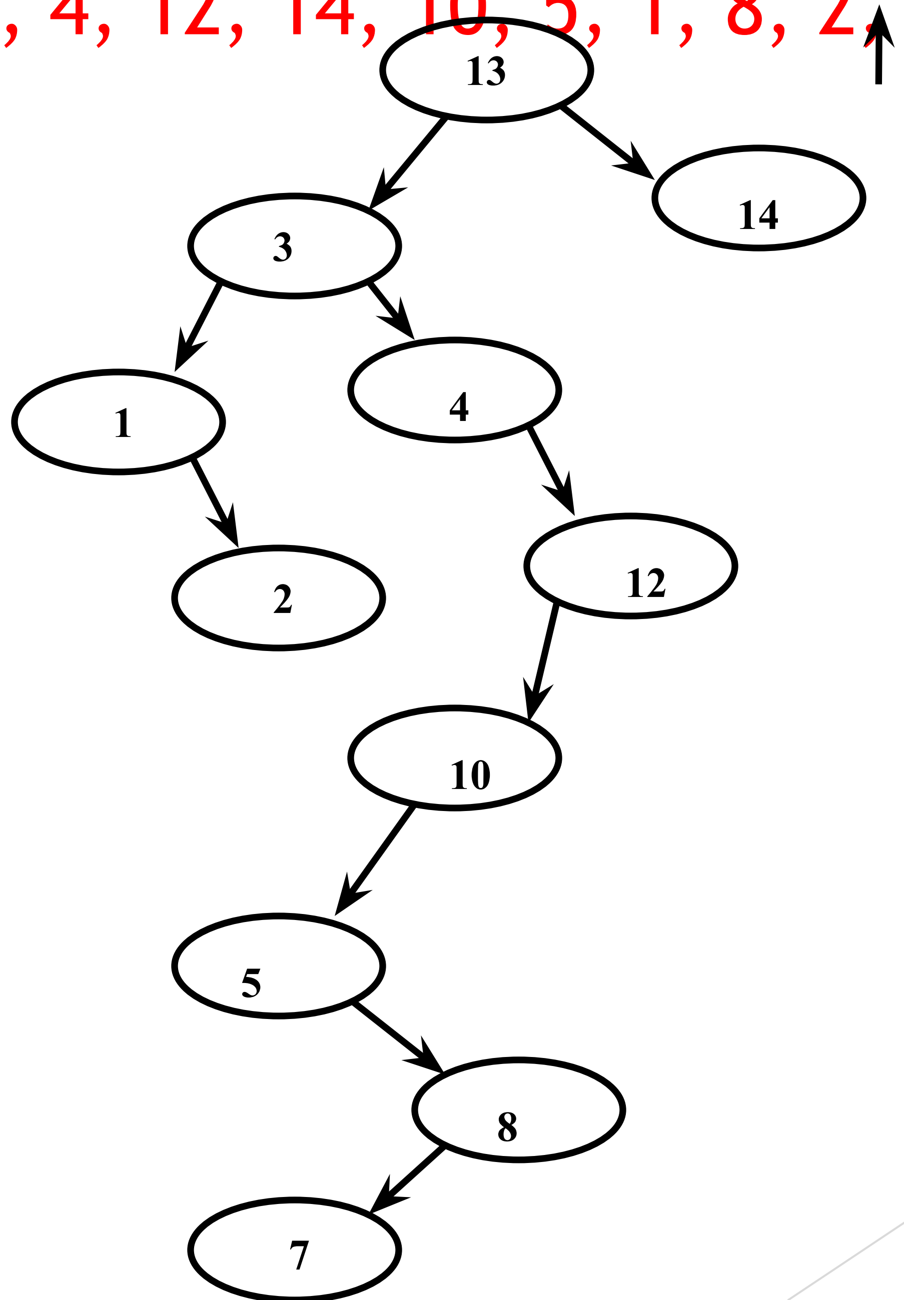
13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



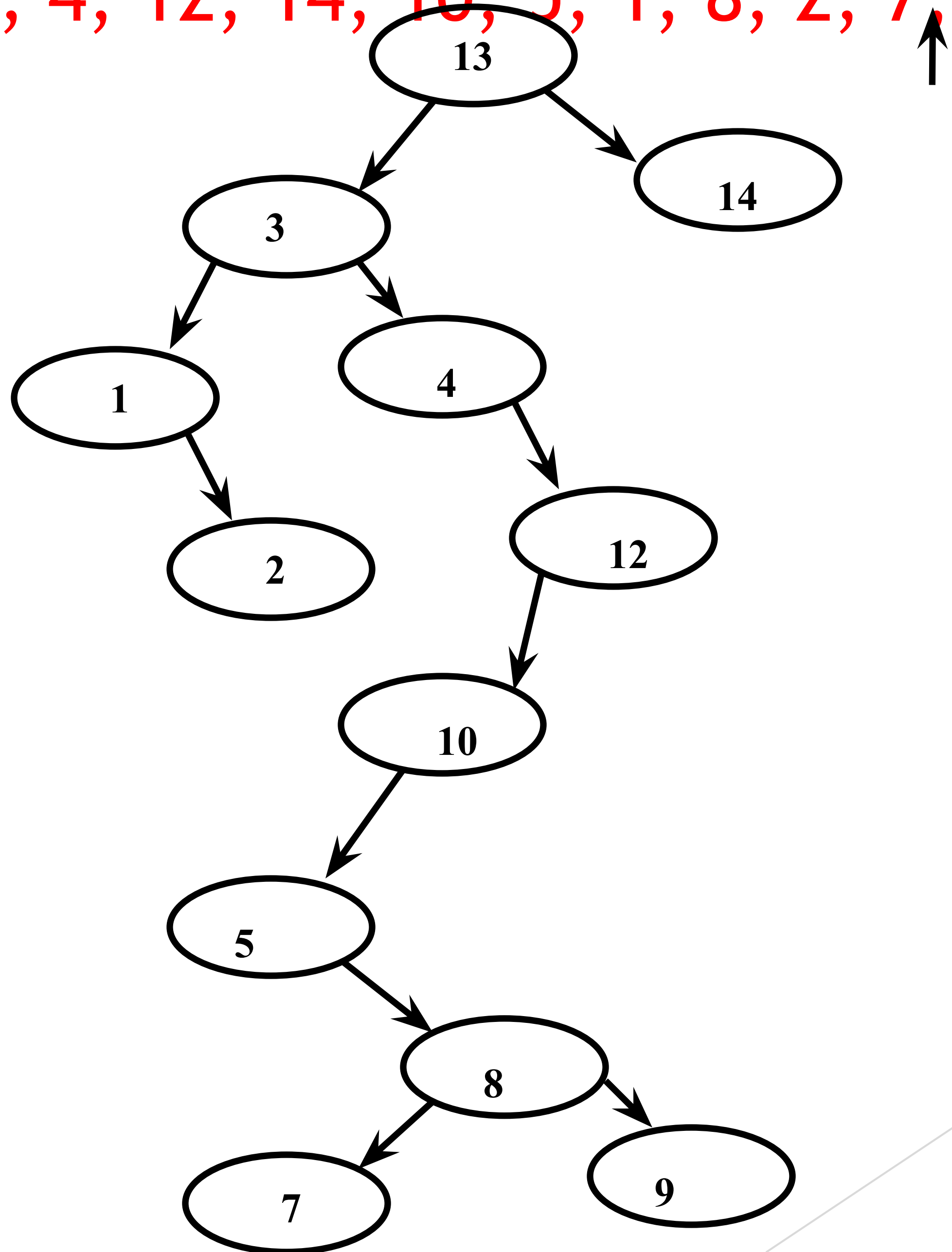
13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



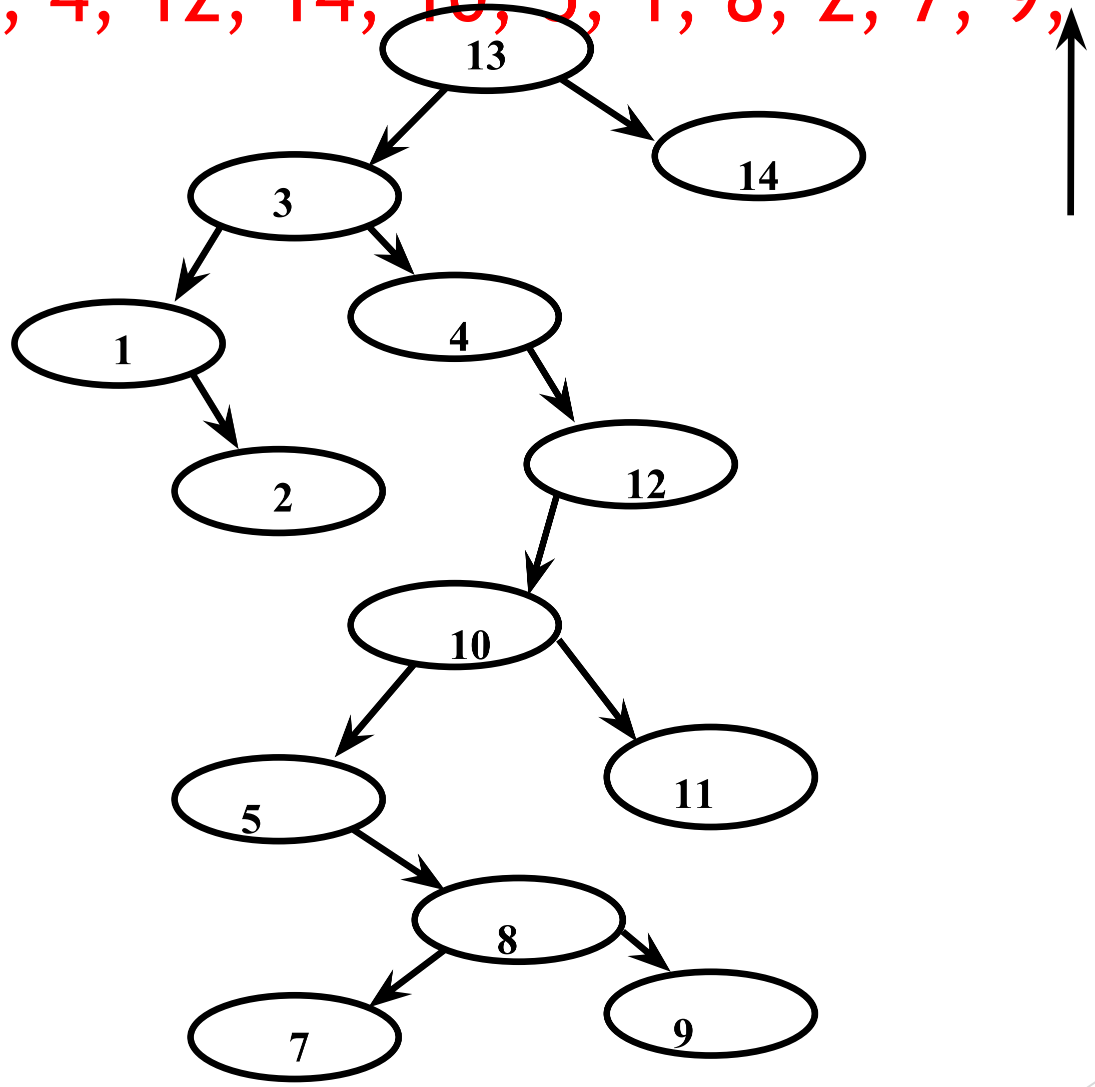
13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



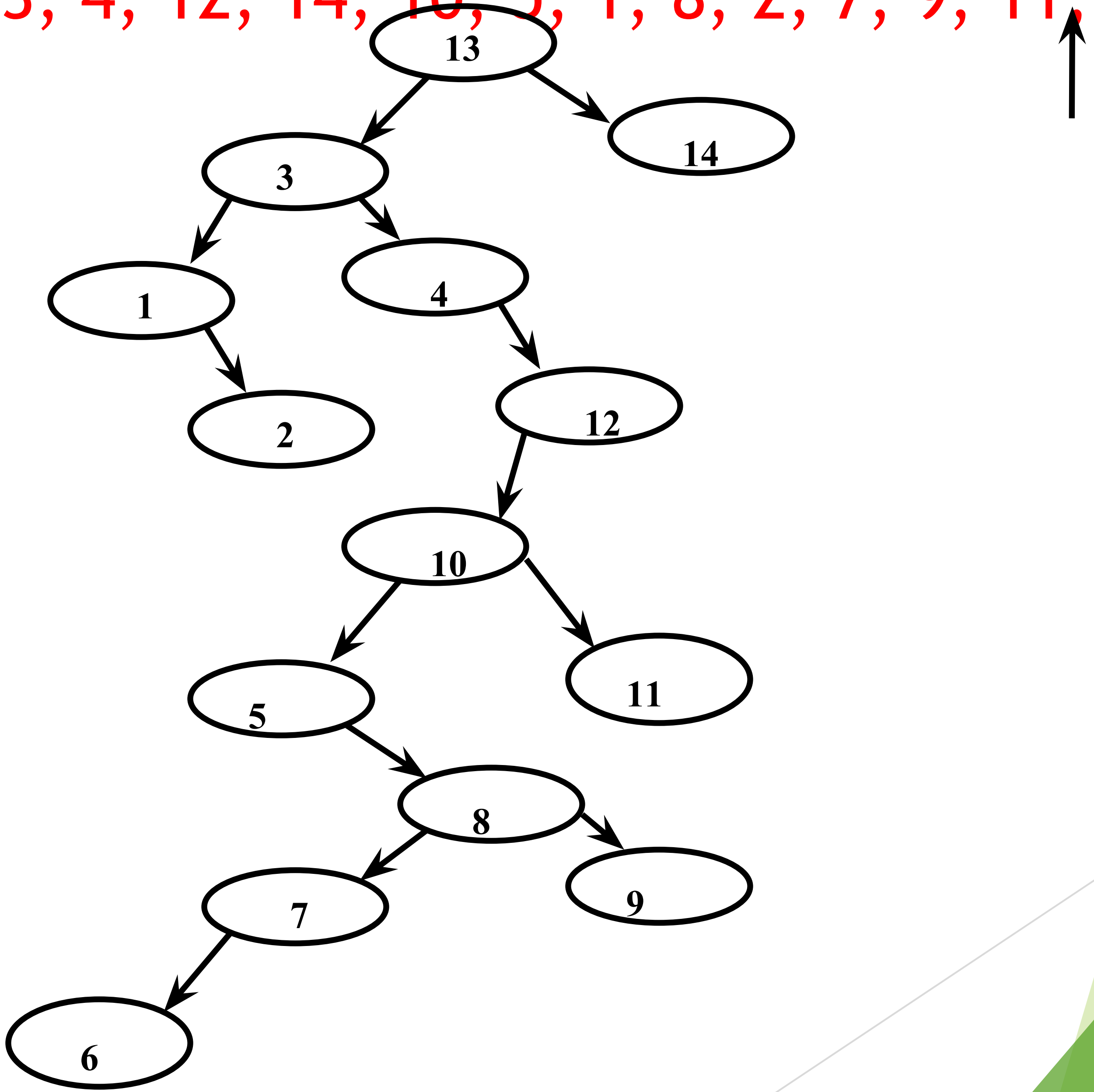
13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



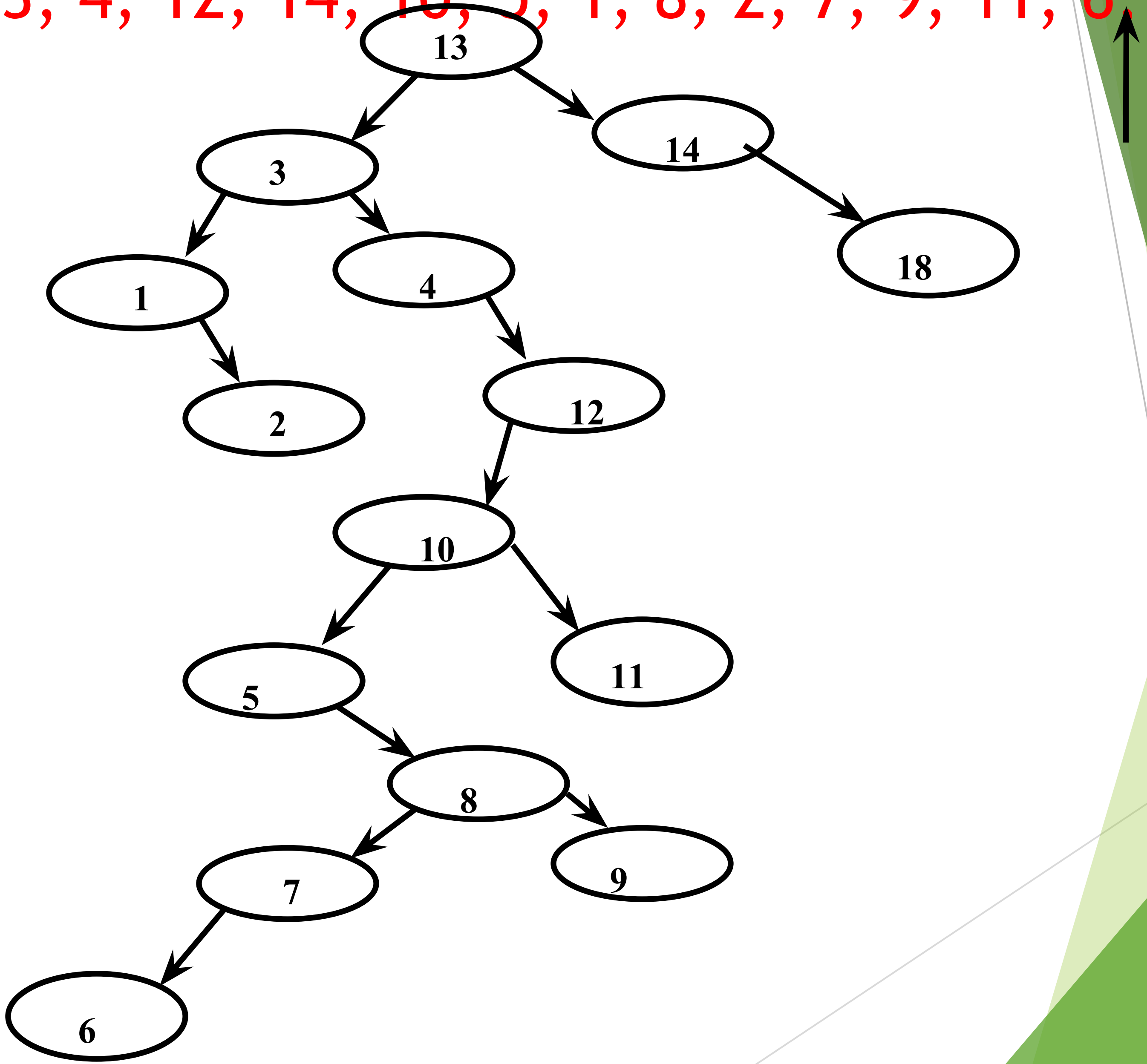
13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



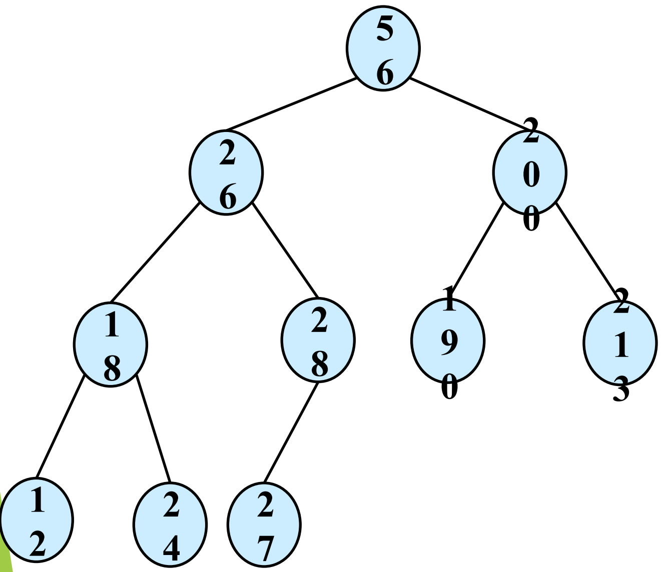
13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 9, 11, 6, 18



BST Insertion – Pseudocode



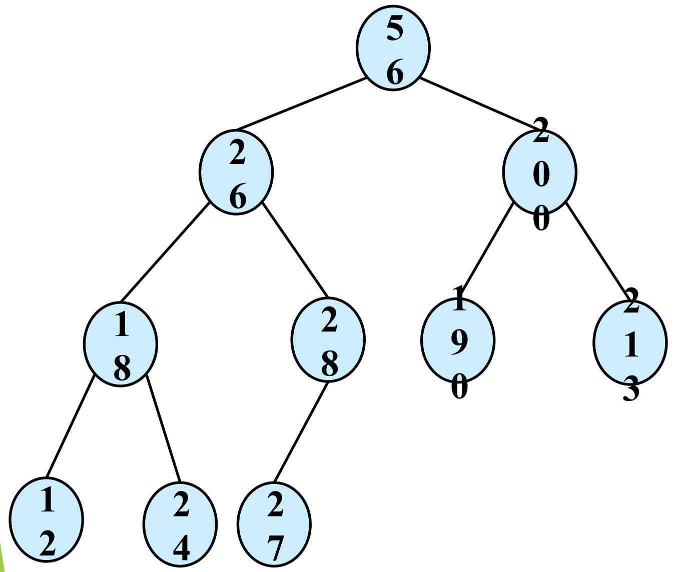
```
struct Tree *insert(struct Tree *root, int data)
{
If(root ==Null)
{
root=struct Tree* malloc(sizeof(struct Tree ));
root->data=data;
root->left=root->right=Null;
}
else if(data<root->data)
root->left=insert( root->left, data);
else if (data>root->data)
root->right=insert( root->right, data);
}
Return root;
}
```

Exercise: Sorting Using BSTs

```
Sort (A)
    for i ⟵ 1 to n
        do tree-insert(A[i])
    inorder-tree-walk(root)
```

- What are the worst case and best case running times?
- In practice, how would this compare to other sorting algorithms?

If we find the inorder sequence of a binary search tree, then it will always be in the increasing order of keys.



- Inorder sequence of the given tree is: 12 18 24 26 27 28 56 190 200 213
- In the above sequence, any node A followed by a node B is called inorder successor. In other words, B is inorder successor of A and A is inorder predecessor of B.
- In the given tree node 28 is predecessor of node 56 and node 190 is successor of node 56.

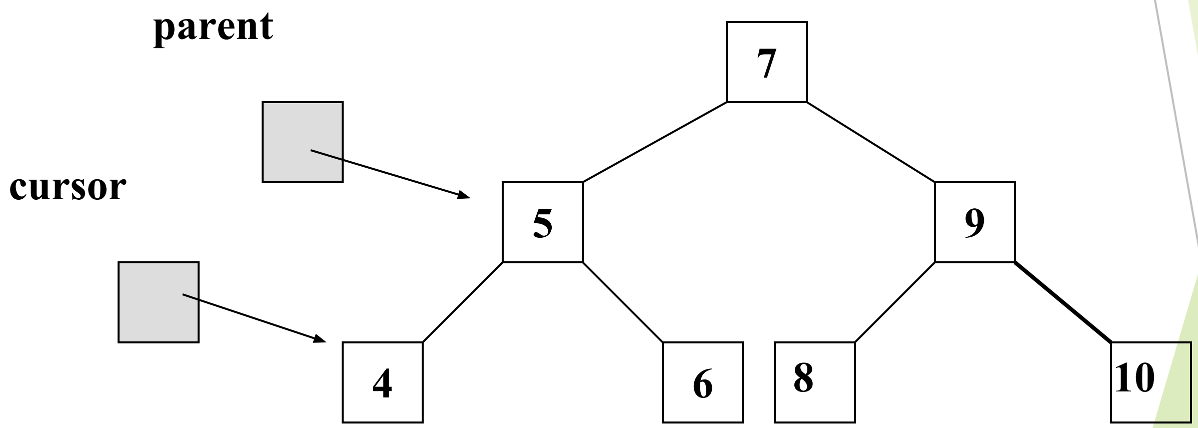
Tree-Delete (*T*, *x*)

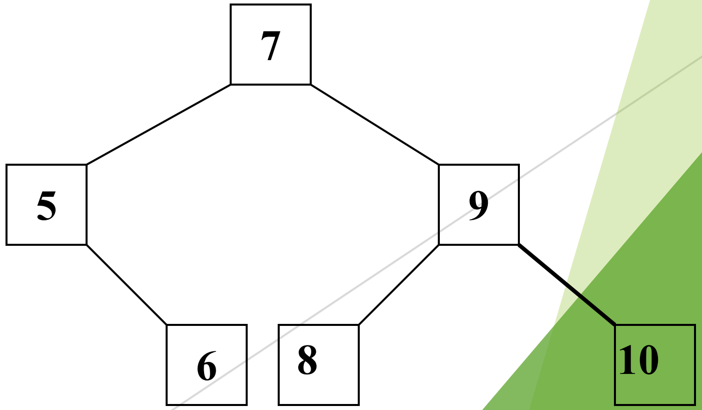
```
if x has no children                    ♦ case 0
    then remove x

if x has one child              ♦ case 1
    then make p[x] point to child

if x has two children (subtrees)        ♦ case 2
  then swap x with its successor
       perform case 0 or case 1 to delete it
```

⇒ TOTAL: $O(h)$ time to delete a node

Removal in BST: Example

Case 0: removing a node with 2 EMPTY SUBTREES



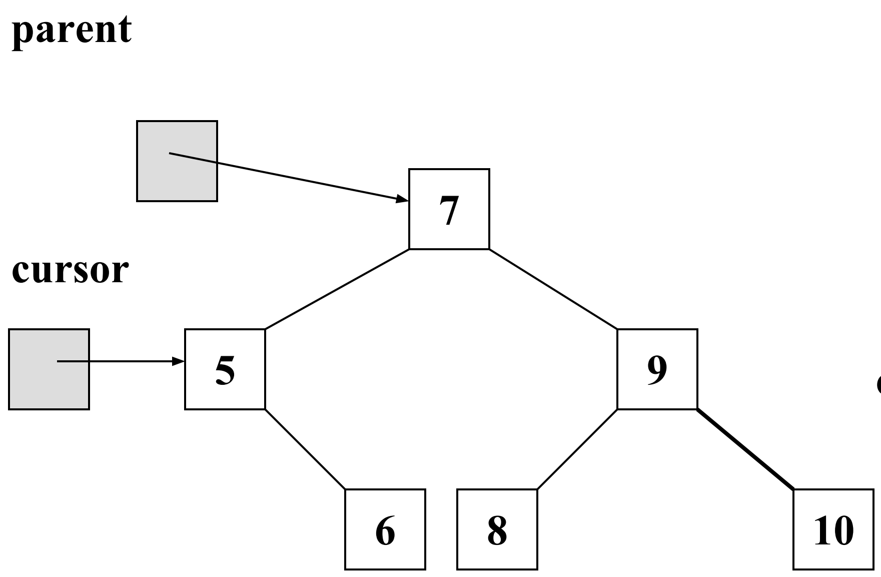


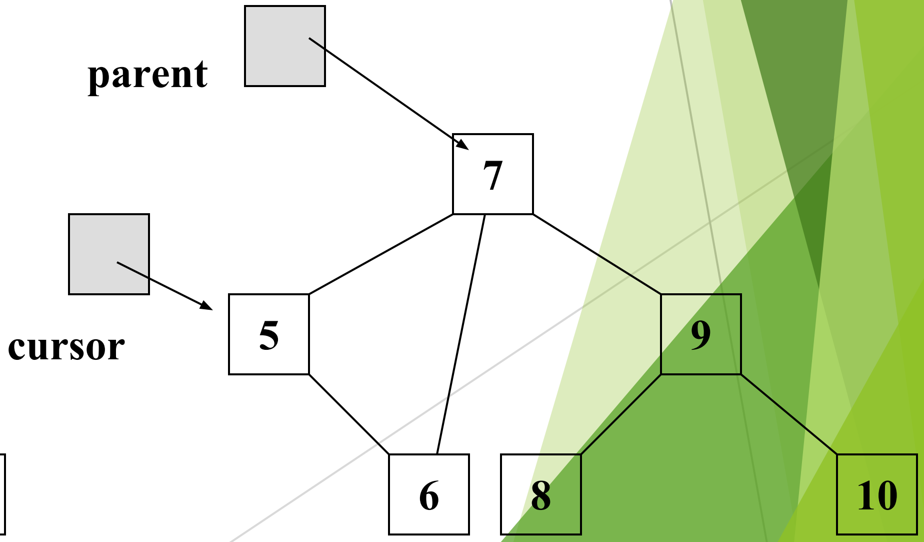
Removing 4
replace the link in the parent with
`null`

Removal in BST: Example

Case 1: **removing a node with 1 EMPTY SUBTREE**

the node has no left child:
link the parent of the node to the right (non-empty) subtree



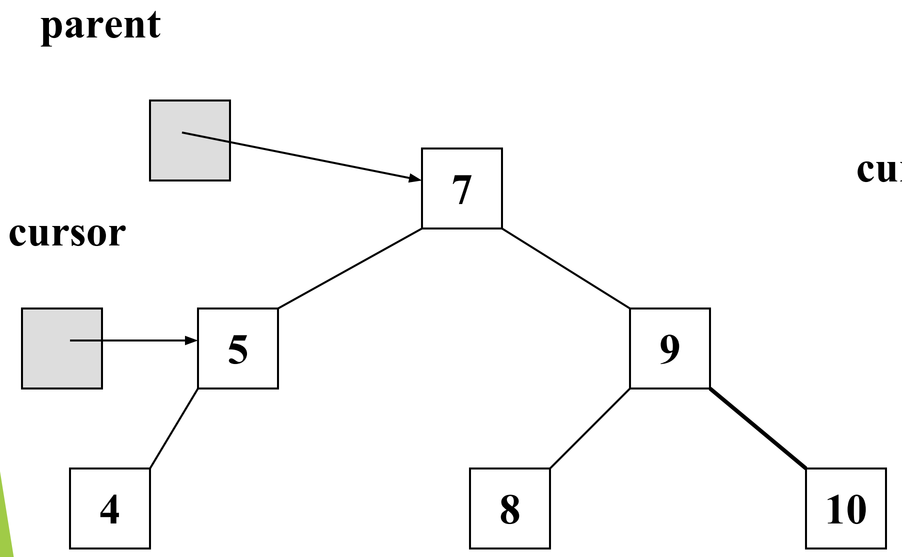


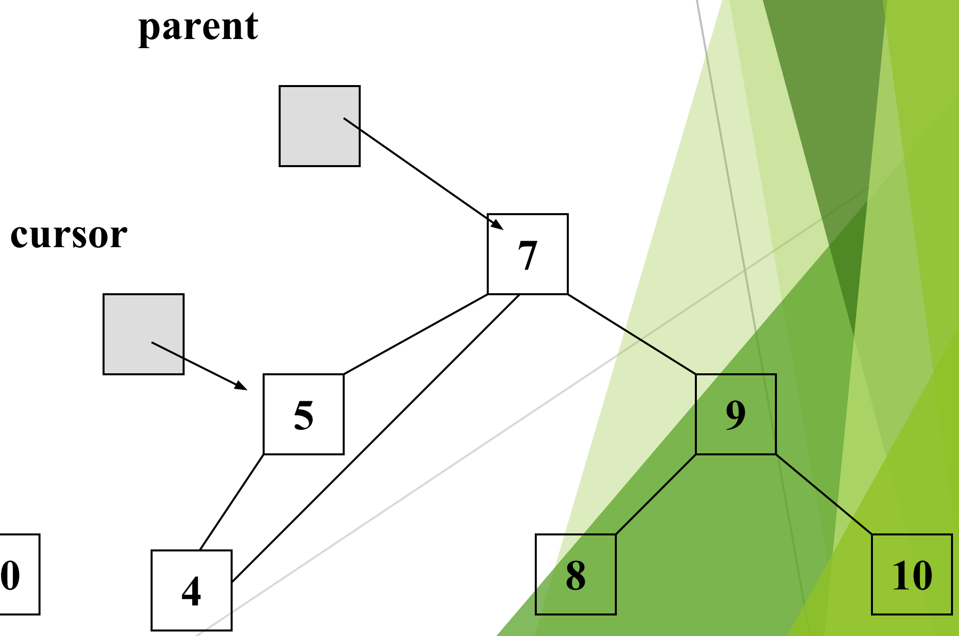
Removal in BST: Example

Case 1: removing a node with 1 EMPTY SUBTREE

the node has no right child:
link the parent of the node to the left (non-empty) subtree

Removing 5





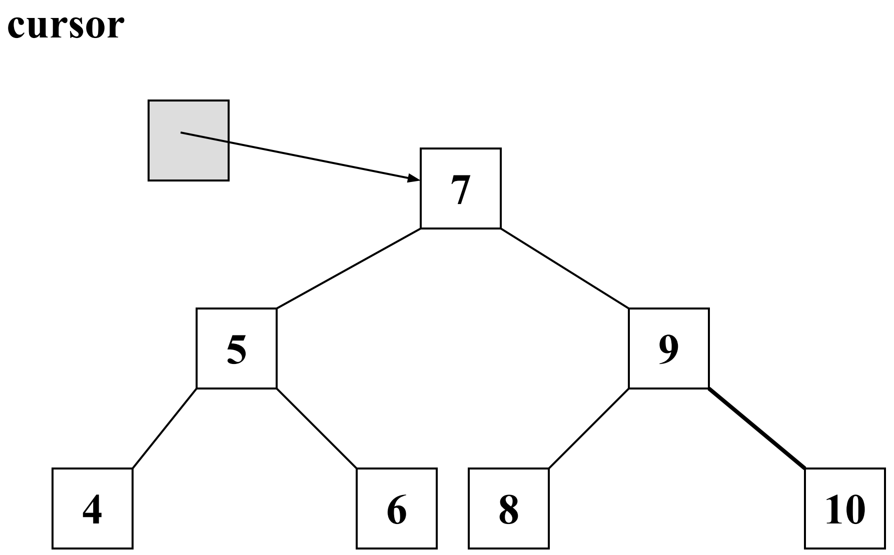
Removal in BST: Example

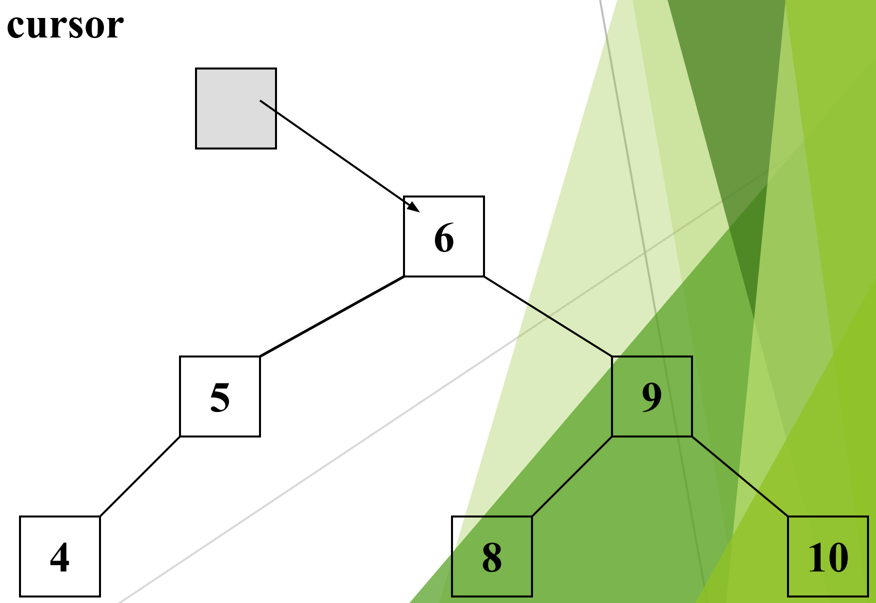
Case 2: removing a node with 2 SUBTREES

- replace the node's value with the max value in the left subtree
- delete the max node in the left subtree

What other element can be used as replacement?

Removing 7





Deletion – Pseudocode

Suppose we want to delete a node **z**.

1. If **z** has no children, then we will just replace **z** by nil.

2. If **z** has only one child, then we will promote the unique child to **z**’s place.

3. If **z** has two children, then we will identify **z**’s successor. Call it **y**. The successor **y** either is a leaf or has only the right child. Promote y to **z**’s place. Treat the loss of **y** using one of the above two solutions.

Deletion – Pseudocode

```
struct Tree *delete(struct Tree *root, int data)
{
struct Tree *Temp;
if(root ==Null)
printf(“element is not there”);
else if ( data < root->data)    /*leaf node */
      root-> left=delete(root-> left, data);
else if ( data > root-> data)    /*leaf node */
      root-> right=delete(root-> right, data);
else
    if (root-> left && root-> right)          /* /Case 2 */
      {
       /*replace with largest in left Subtree */
      Temp=FindMax( root-> left);
      root-> data=Temp-> data;
      root->left= delete(root-> left, root-> data);}
```

```
Struct Tree *FindMax(Struct Tree *root)
{
If(root ==Null)
Return Null
Else If (root->right==Null)
Return root
Else return FindMax(root->right);
}
```

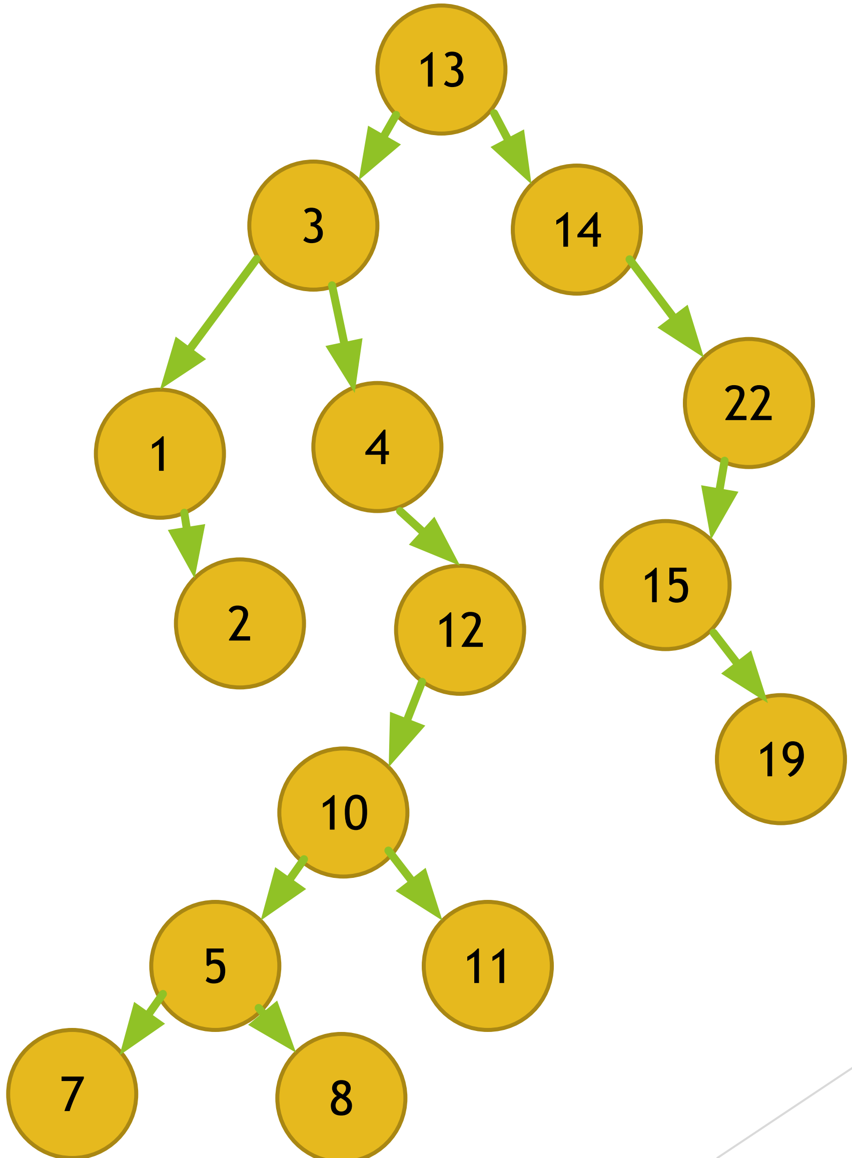
```
 else
    {         /* one child */
         Temp=root;
         if (root->left==Null)
              root=root->right;
         if (root->right==Null)
              root=root->left;
         free(Temp);
    }
}
return root;
}
```

Correctness of Tree-Delete

- How do we know case 2 should go to case 0 or case 1 instead of back to case 2?
  - Because when *x* has 2 children, its successor is the minimum in its right subtree, and that successor has no left child (hence 0 or 1 child).
- Equivalently, we could swap with predecessor instead of successor. It might be good to alternate to avoid creating lopsided tree.

Example

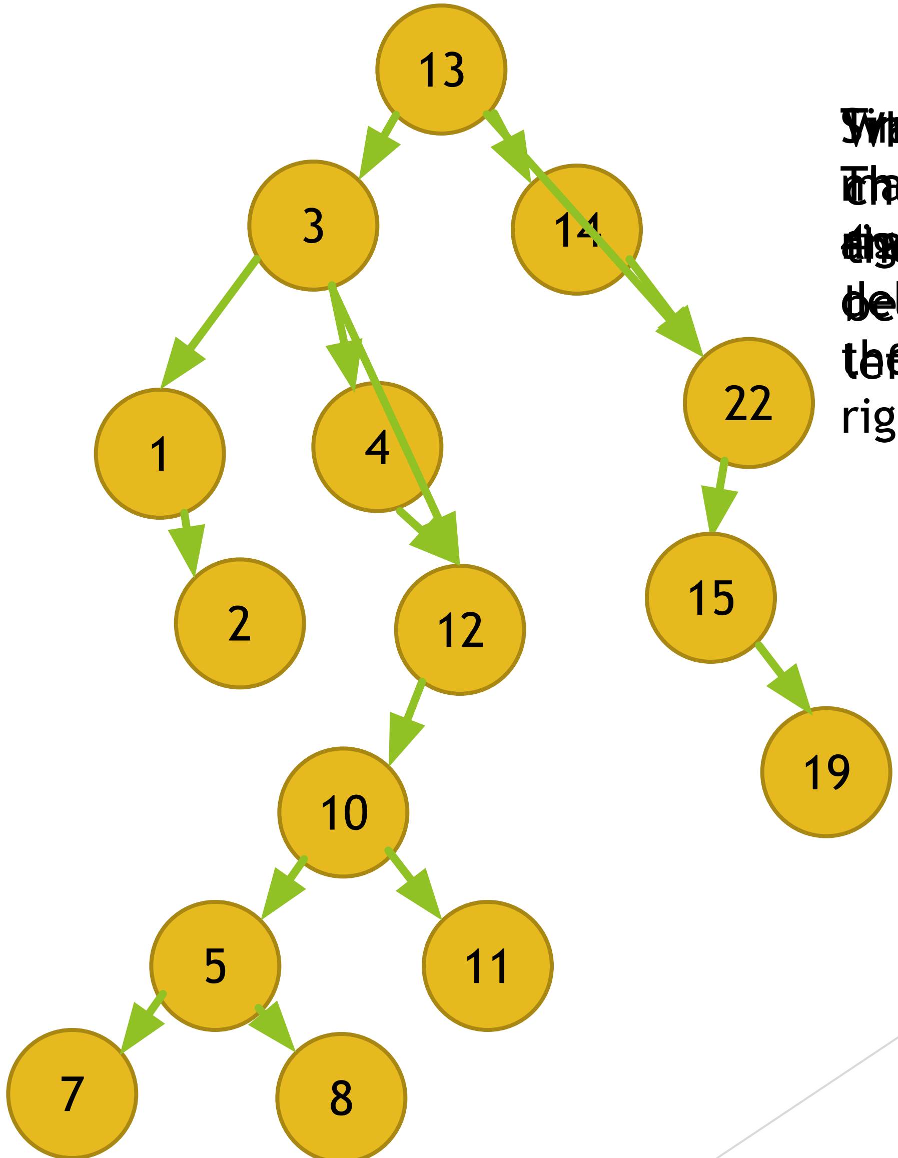
- Suppose we want to insert 13, 3, 4, 12, 14, 10, 5, 1, 8, 2, 7, 11, 22, 15, 19 elements into a BST.



[illegible] right of 4.

Example

- Suppose we want to delete 8, 22, 4, and 13 elements from the BST.



While deleting the node 13, we choose the greatest node among the children to replace node 13, because node 13 has both the left and right subtree.

Similarly to delete node 22, [illegible] delete the node 22 and make the left subtree of 22 as the right subtree of 14.

Advantages of BST

- ✓ Simple
- ✓ Efficient
- ✓ Dynamic

- ✓ One of the most fundamental algorithms in CS
- ✓ The method of choice in many applications

Disadvantages of BST

- ✓ The shape of the tree depends on the order of insertions, and it can be degenerated.
- ✓ When inserting or searching for an element, the key of each visited node has to be compared with the key of the element to be inserted/found.
- ✓ Keys may be long and the run time may increase much.

Improvements of BST

Keeping the tree balanced:

AVL trees (Adelson - Velskii and Landis)

Balance condition: left and right subtrees of each node can differ by at most one level.

It can be proved that if this condition is observed the depth of the tree is O(logN).

Reducing the time for key comparison:

Radix trees - comparing only the leading bits of the keys (not discussed here)

Complexity Issues

- AVL trees:
  - Search,Insertion, and Deletion: O(logN)
  - Creating the tree - O(N)

- Trees with multiple keys: $O(Log_m N)$

  m - branching factor

Next material will be on AVL trees