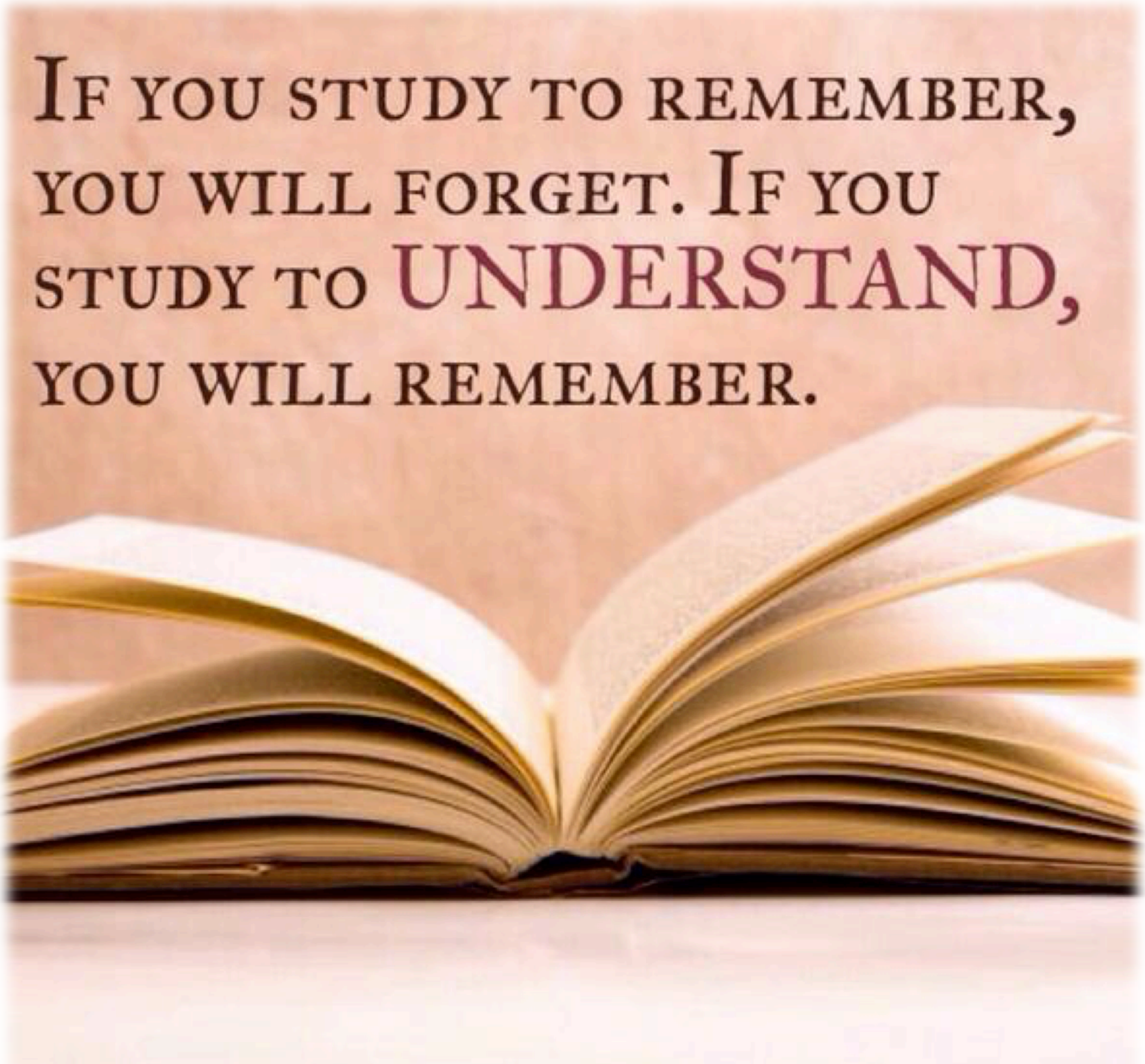
CS 214 – DATA STRUCTURES FALL 2015

Lecture 8 – Binary Search Trees November 20^{th} 2017

Group **B & C**

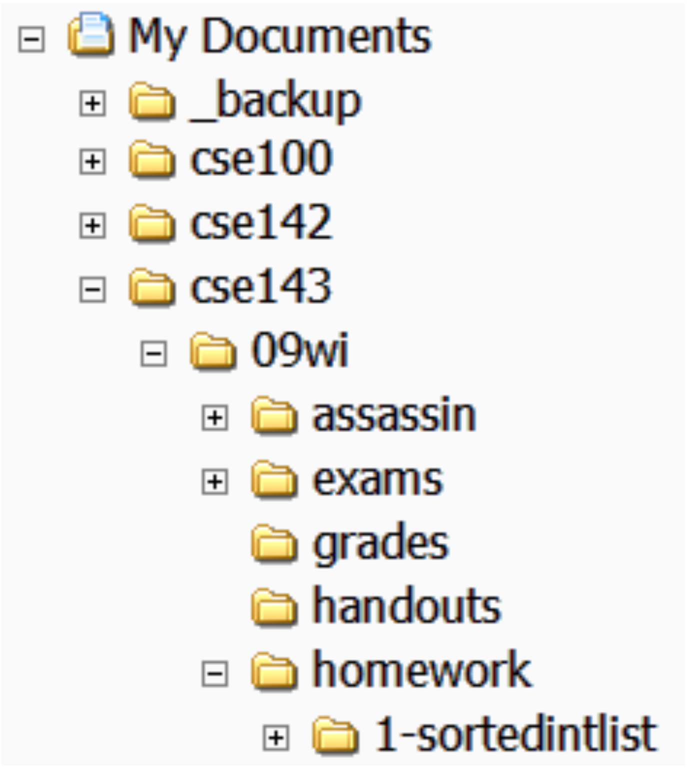
Dr. Mai Hamdalla

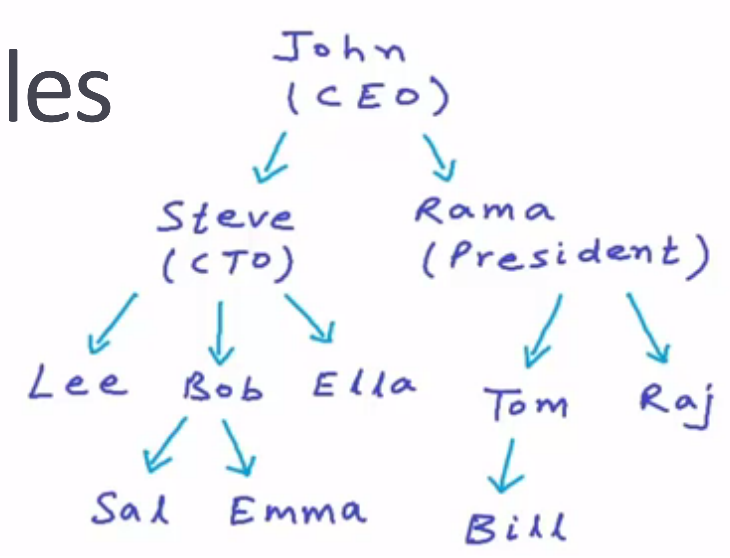
mai@fci.helwan.edu.eg

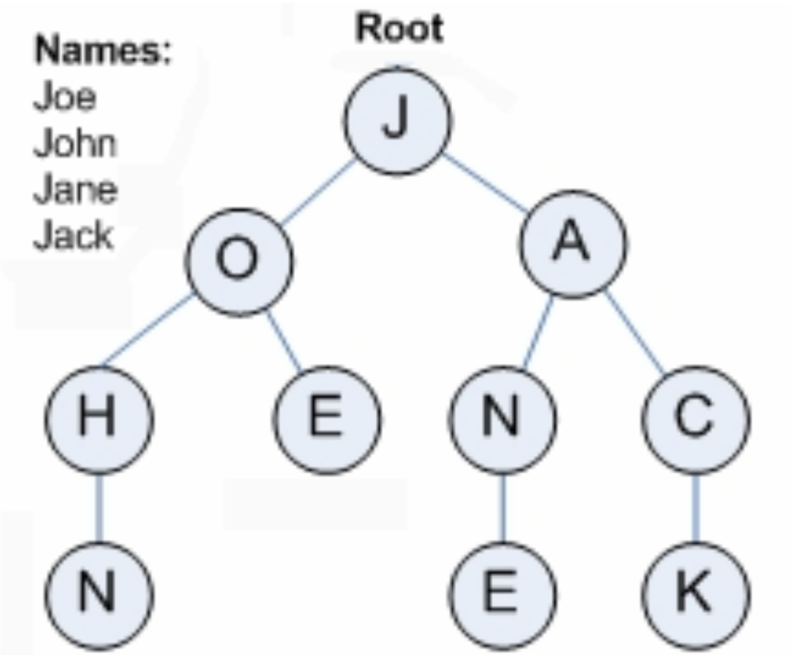

If you study to remember,
you will forget. If you
study to UNDERSTAND,
you will remember.

TREES

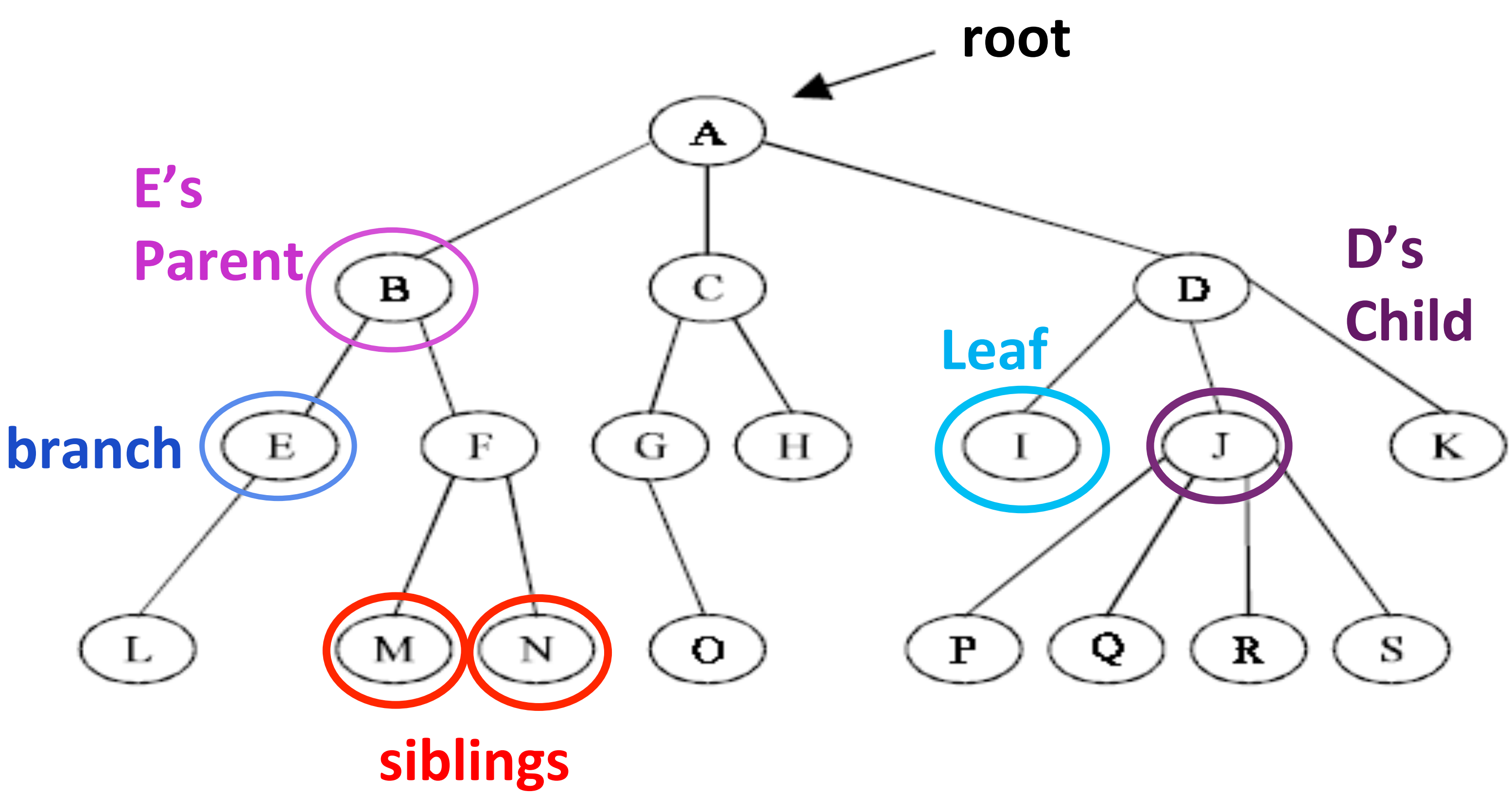
Tree Structures - Examples





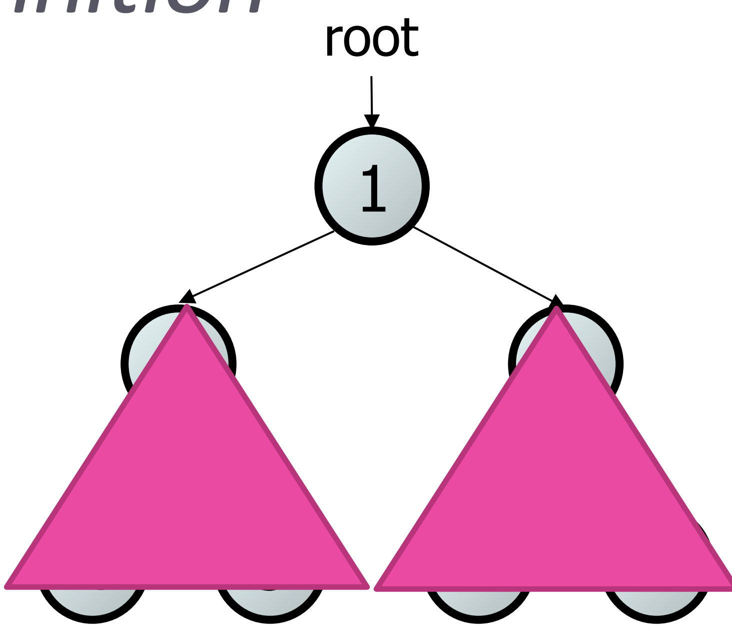


Terminology (II)



BINARY TREE

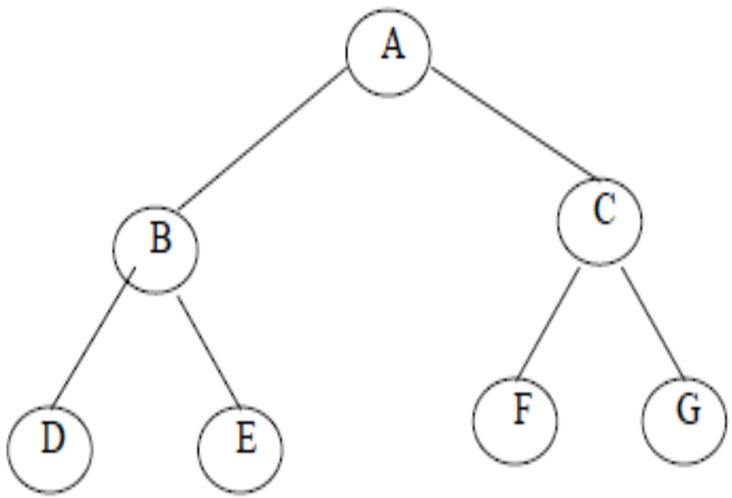
Binary Tree - *Recursive definition*

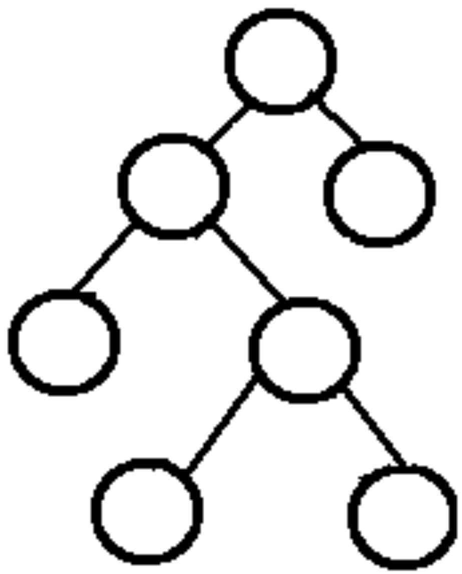


- A tree is either:
  - empty (null), or
  - a **root** node (vertex) that contains:
    - **data**,
    - a **left** subtree, and
    - a **right** subtree.

The left and/or right subtree could be empty.

Full Binary Tree

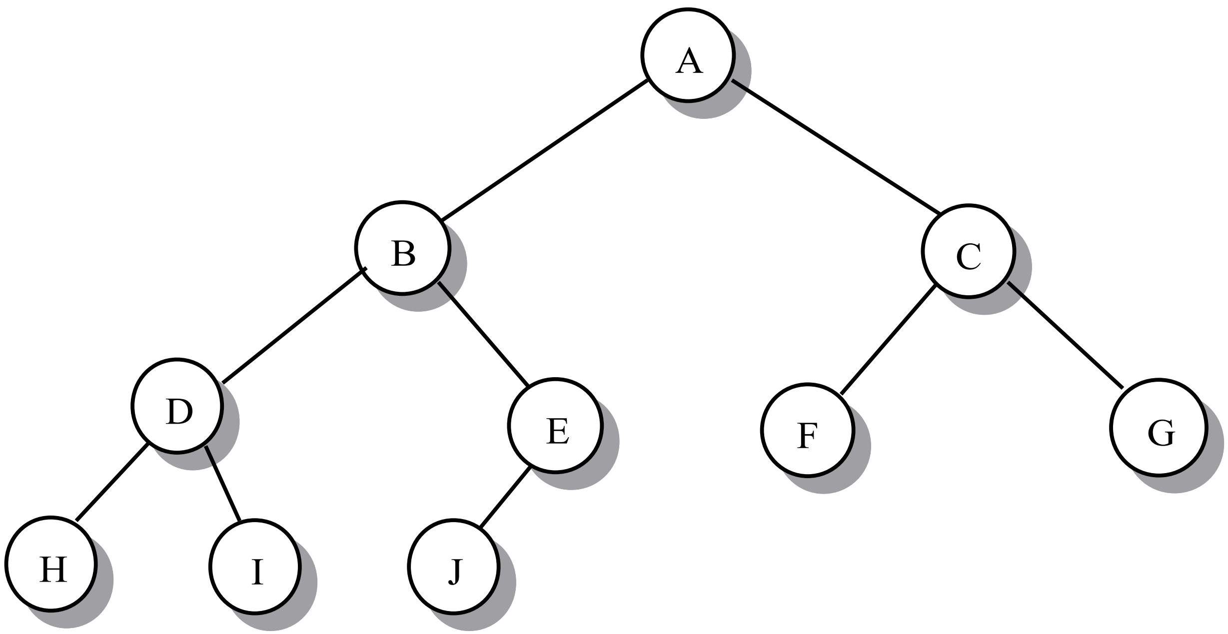




- A binary tree in which every node other than the leaves has **exactly** two children.

Complete Binary Tree

- A binary tree in which every level, except possibly the last, is **completely filled**, and all nodes are as far left as possible.



Complete Tree (Depth 3)

BINARY TREE TRAVERSAL

Binary Tree Traversal

Traversal Type	Inorder	Preorder	Postorder
Shortcut	L V R	V L R	L R V

Binary Tree Traversal

- Preorder traversal

F, B, A, D, C, E, G, I, H

(root, left, right)

- Inorder traversal

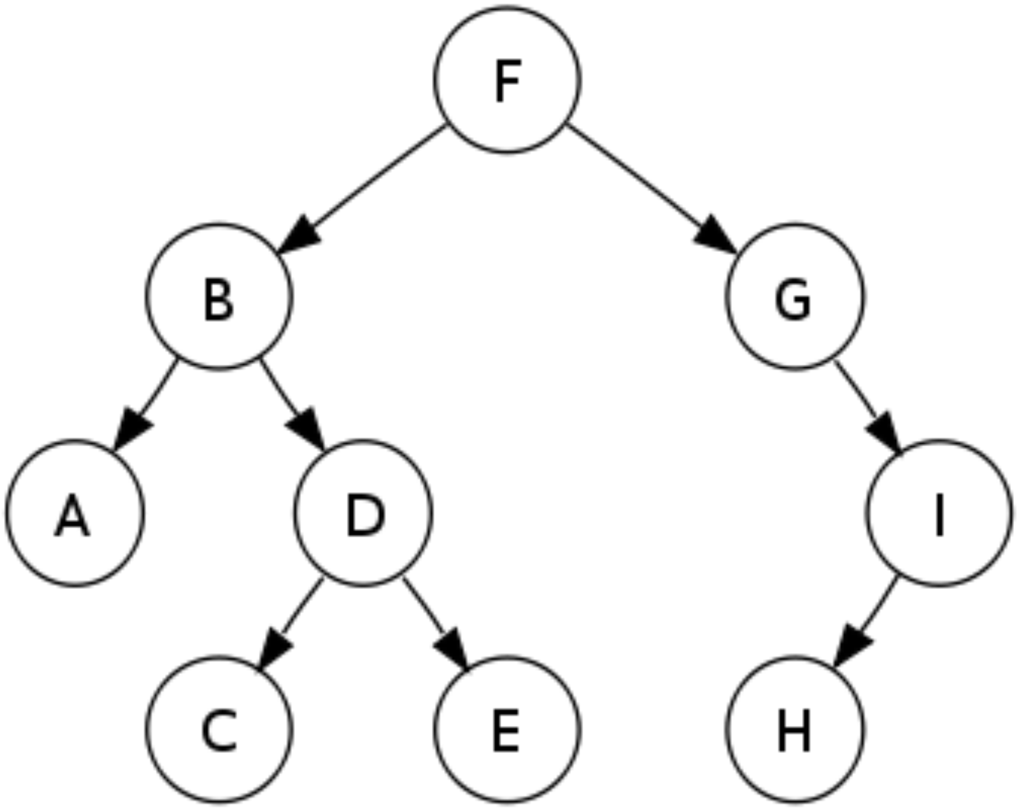
A, B, C, D, E, F, G, H, I

(left, root, right)

- Postorder traversal

A, C, E, D, B, H, I, G, F

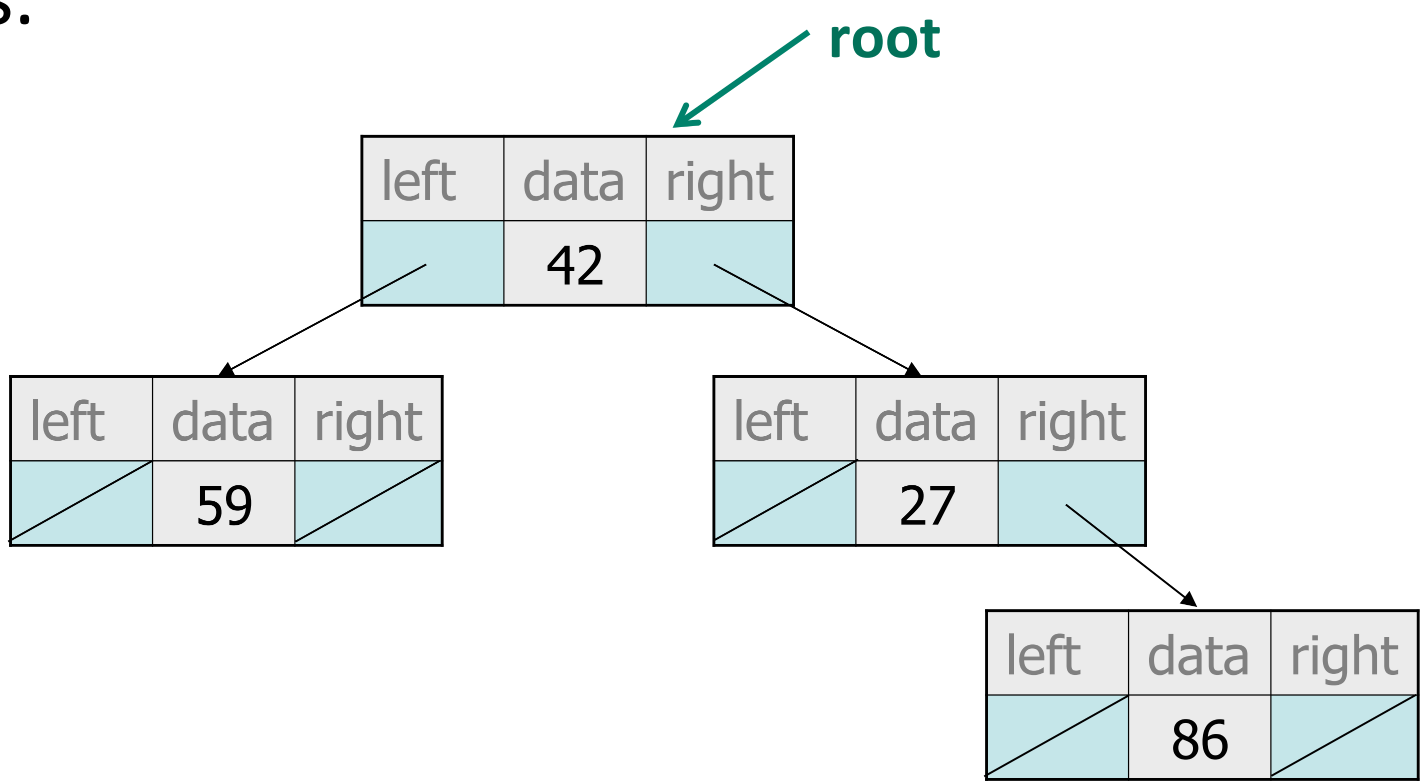
(left, right, root)



BINARY TREE IMPLEMENTATION

A tree node for integers

- A basic **tree node** stores data and left/right links.



Basic Binary Tree Operations

- Create the tree, leaving it empty.
- Determine whether the tree is empty or not
- Determine whether the tree is full or not
- Find the size of the tree.
- Find the depth of the tree.
- Traverse the tree, visiting each entry
- Clear the tree to make it empty

Tree Implementation

```
typedef struct node_type{
    tree_entry info;
    struct node_type *right,*left;
} tree_node;

typedef tree_node *tree;
```

In-order Tree Traversal

Pre: The tree is initialized.

Post: The tree has been traversed in infix order sequence.

```
void inorder_traversal(tree t,
             void(*pvisit)(entry_type)){
  if(t){
    inorder_traversal(t->left, pvisit);
    (*pvisit)(t->info);
    inorder_traversal(t->right, pvisit);
  }
}
```

Tree Size

```
int tree_size(tree t){
    if (!t)
        return 0;
    return(1 + tree_size (t->left)
             +tree_size (t->right));
}
```

Tree Depth

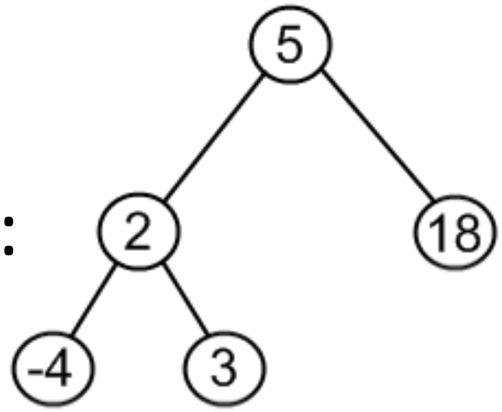
```
int tree_depth(tree t){
    if (!t)
        return 0;
    int a= tree_depth(t->left);
    int b= tree_depth(t->right);
    return (a>b)? 1+a : 1+b;
}
```

Clear Tree

```
void clear_tree(tree *t){
   if (*t){
      clear_tree(&(*t)->left);
      clear_tree(&(*t)->right);
      free(*t);
      *t=NULL;
   }
}
```

BINARY SEARCH TREES

Binary Search Trees (BSTs)



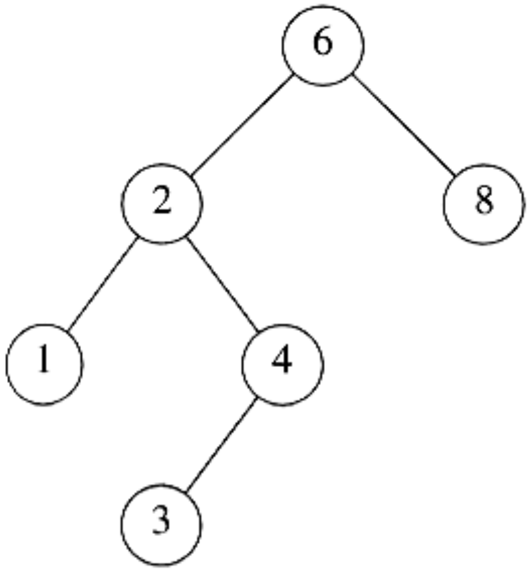
- Meets the following requirements:
  - It’s a binary tree
  - each node contains a ***unique*** value
  - a total order is defined on these values (every two values can be *compared* with each other);
  - **left subtree** of a node contains only values **lesser**, than the node's value;
  - **right subtree** of a node contains only values **greater**, than the node's value.

When to use BSTs?

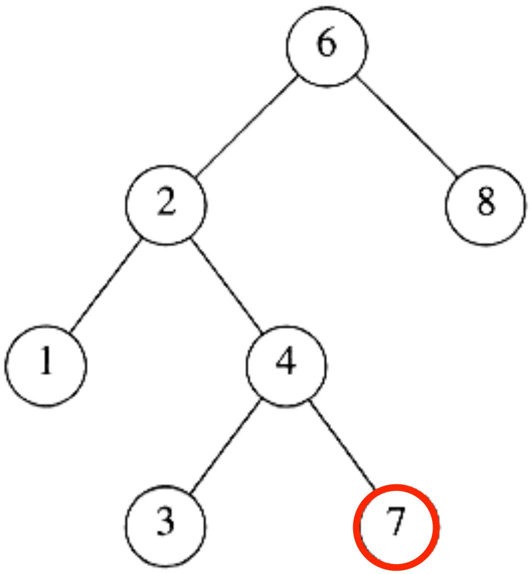
Main advantage of binary search trees is rapid search, while addition is quite cheap.

- Data is often associated with some unique key.
- Storing data in binary search tree allows to look up for the record by key faster, than if it was stored in unordered list.
  - Example: phone book (key is a telephone number).

Binary Search Trees



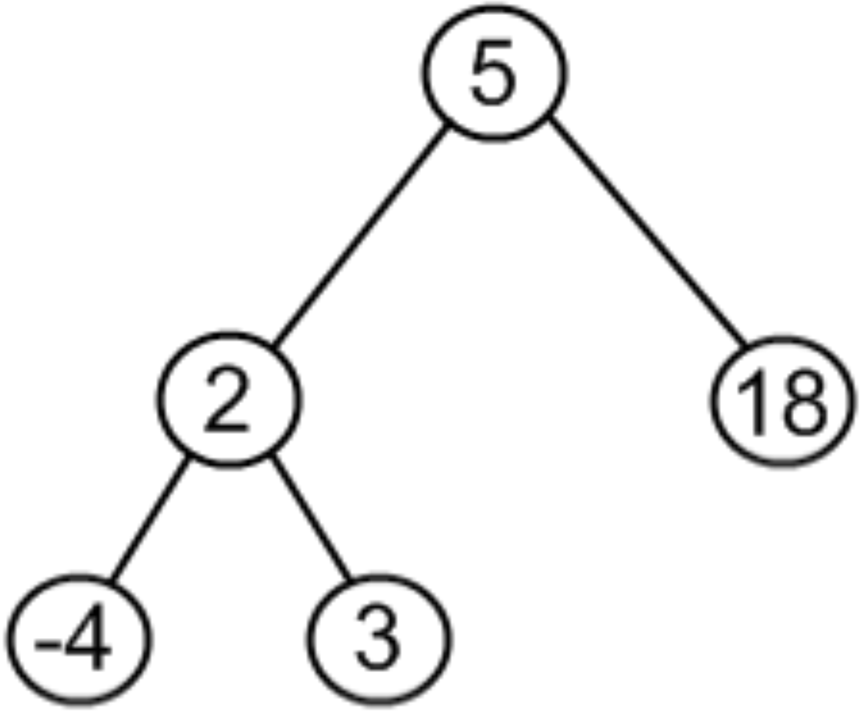
A binary search tree



Not a binary search tree

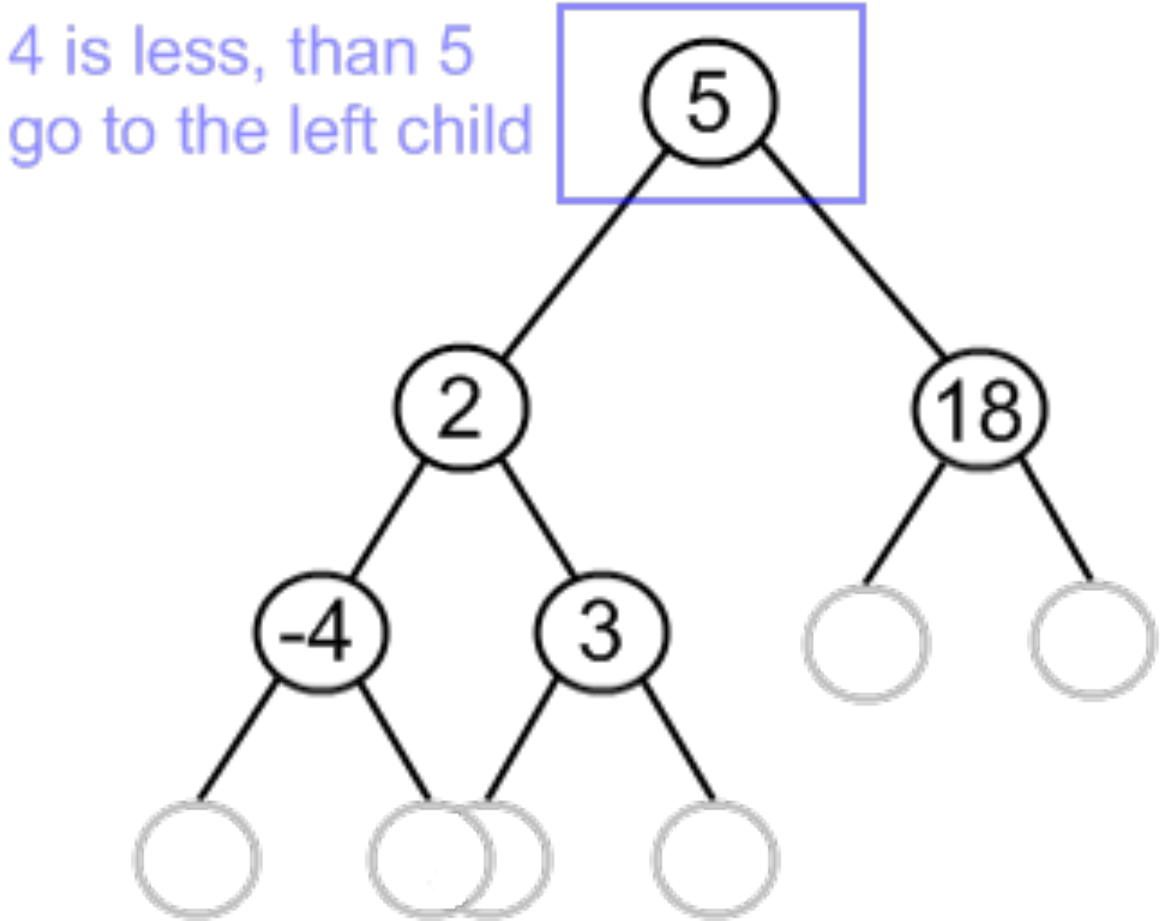
Insert Operation

- Insert **'4'** into a BST.



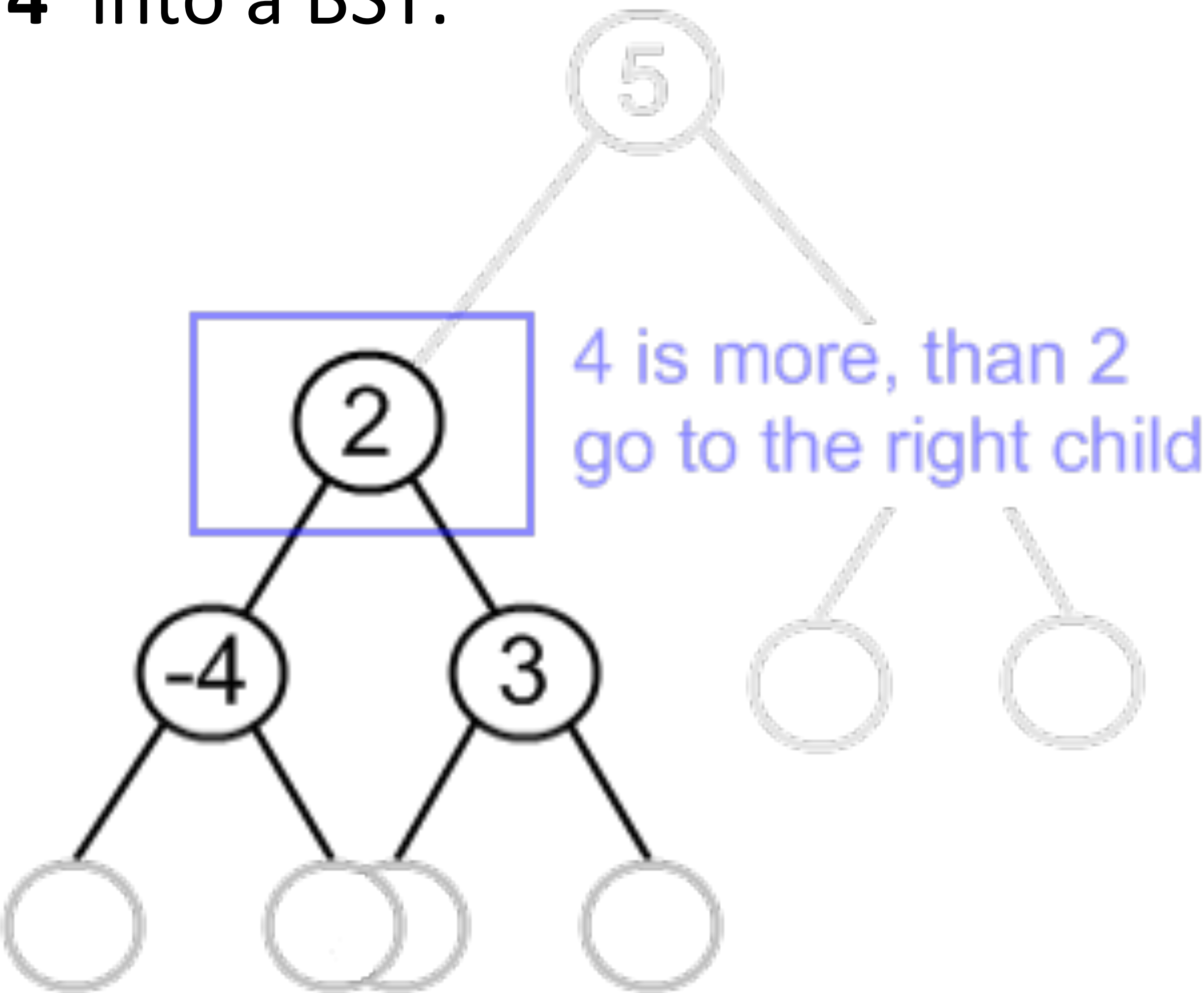
Insert Operation

- Insert **'4'** into a BST.



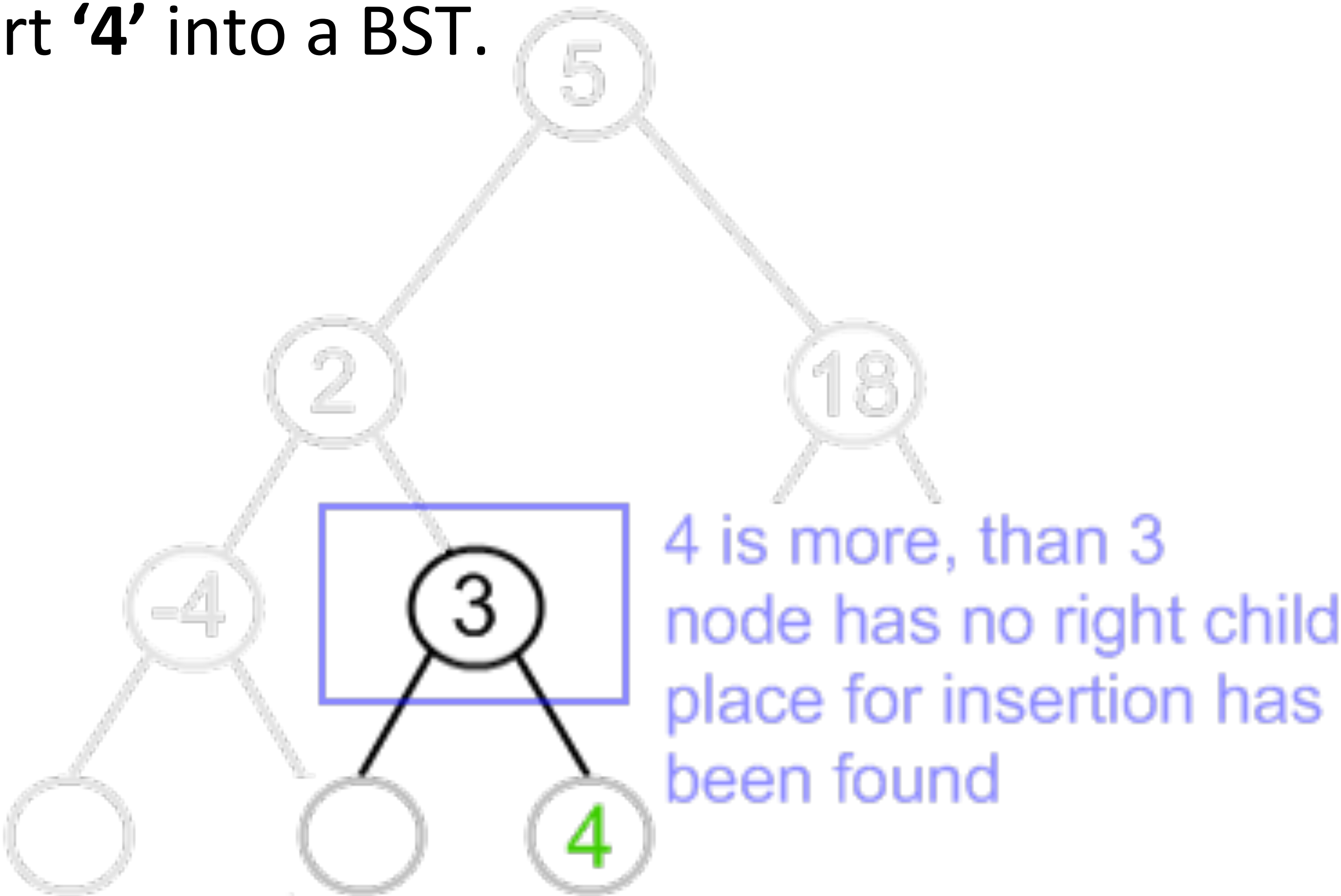
Insert Operation

- Insert **'4'** into a BST.



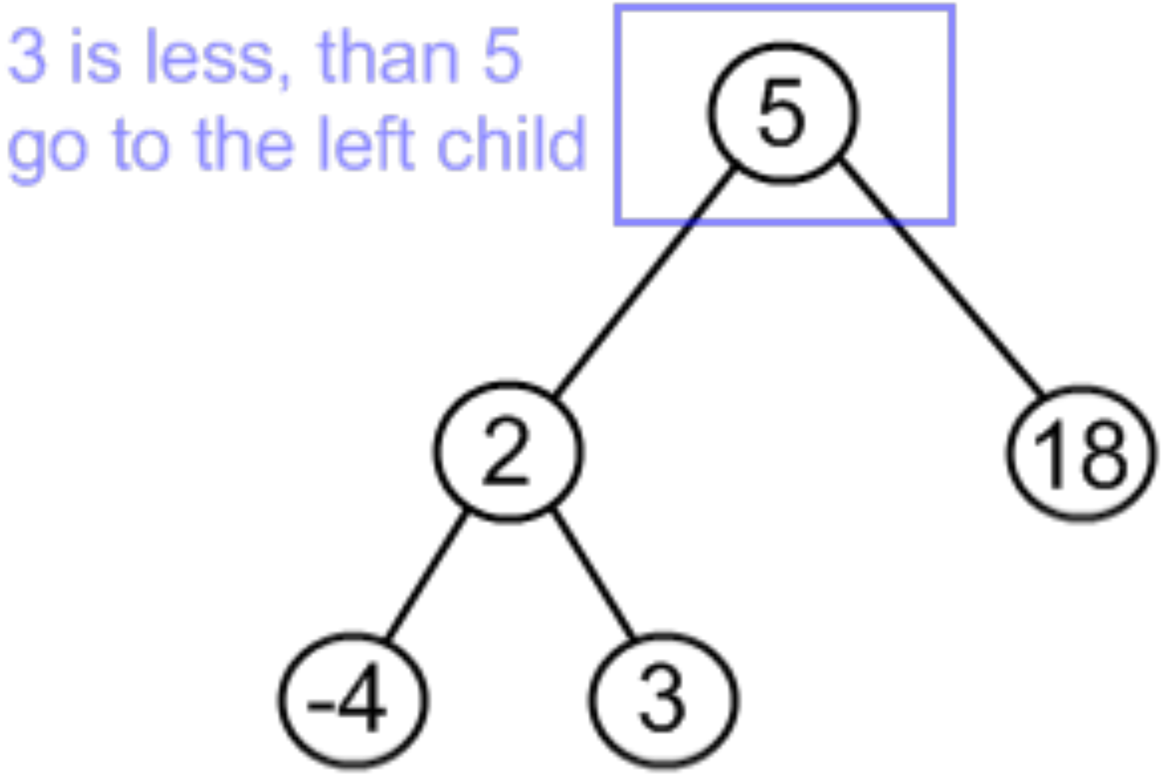
Insert Operation

- Insert **'4'** into a BST.



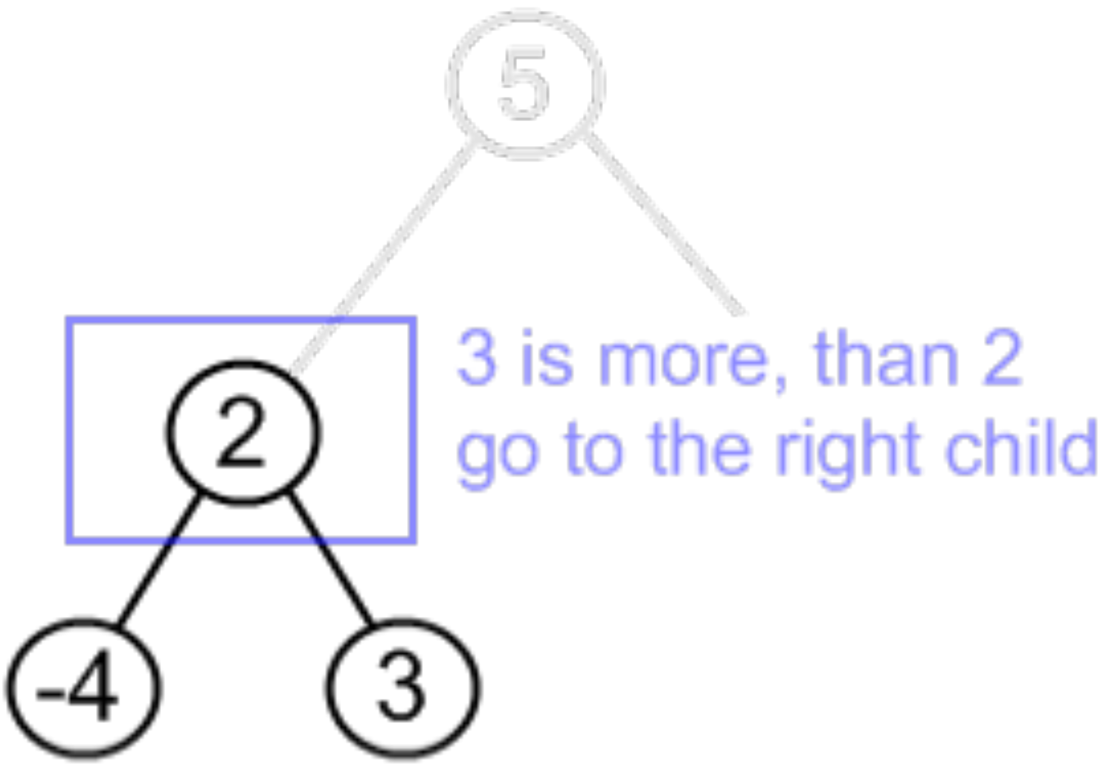
Search Operation

- Search for **'3'** in a BST.



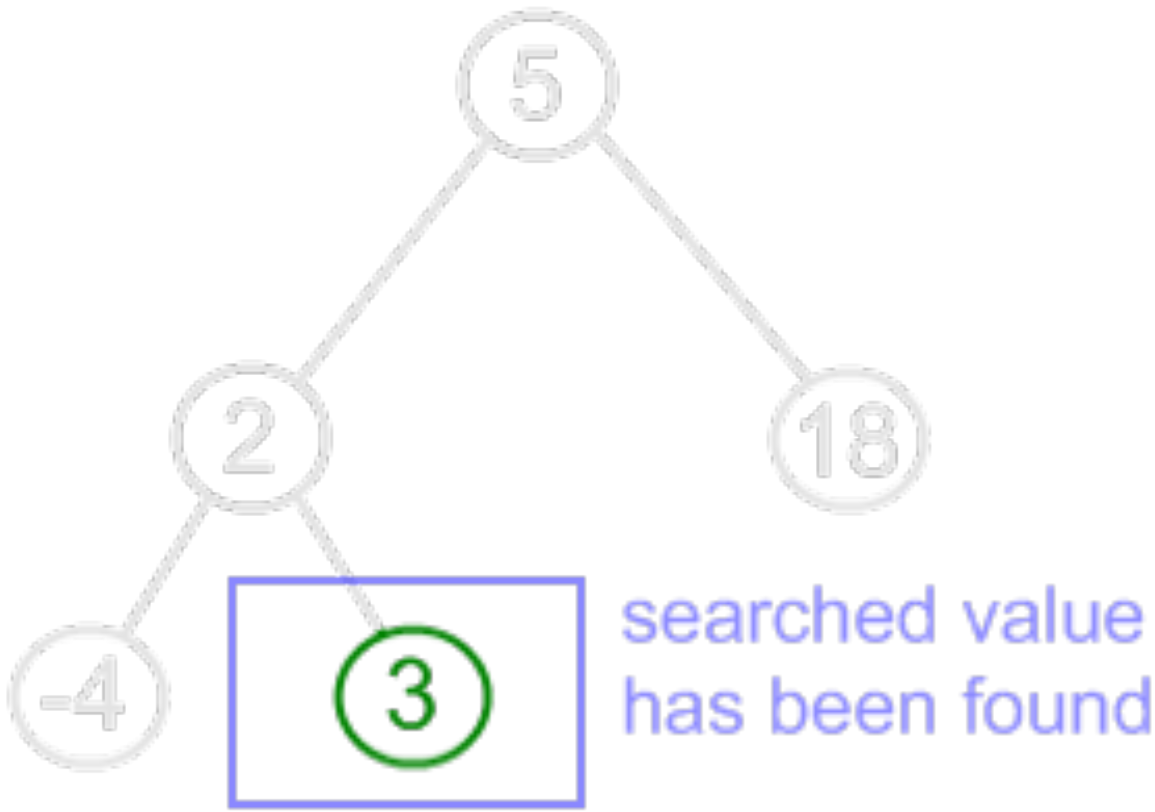
Search Operation

- Search for **‘3’** in a BST.



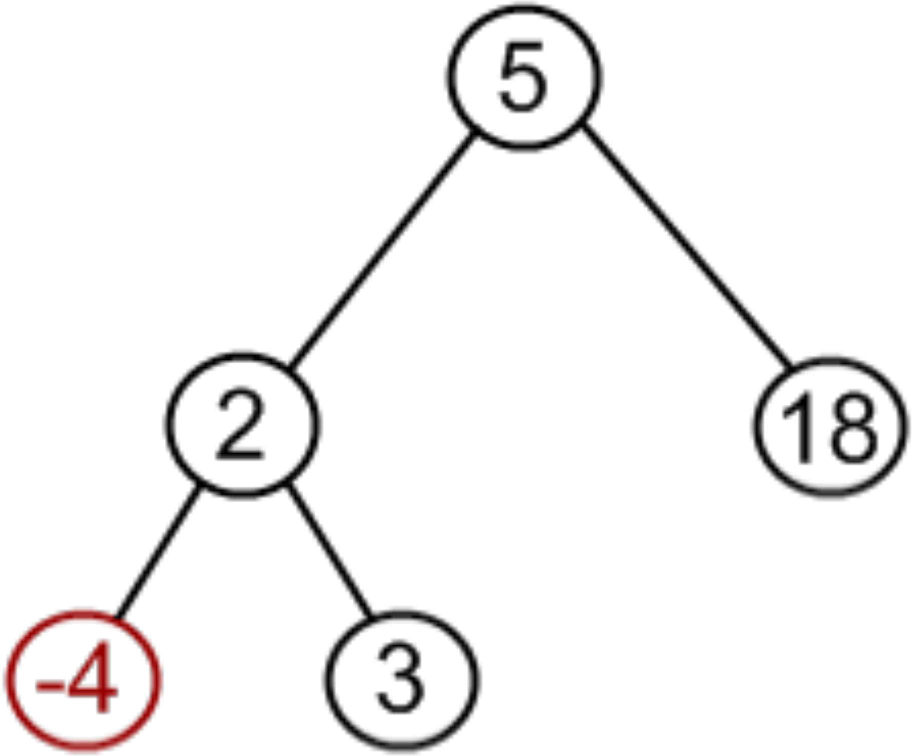
Search Operation

- Search for **'3'** in a BST.



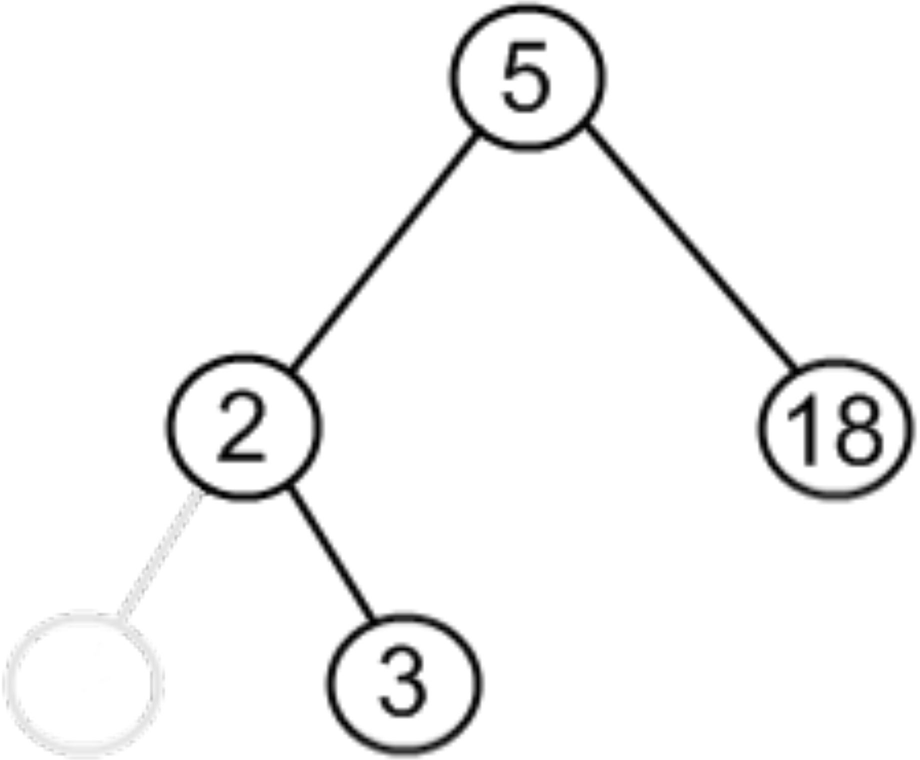
Remove Operation (I)

- Remove **'-4'** from a BST.



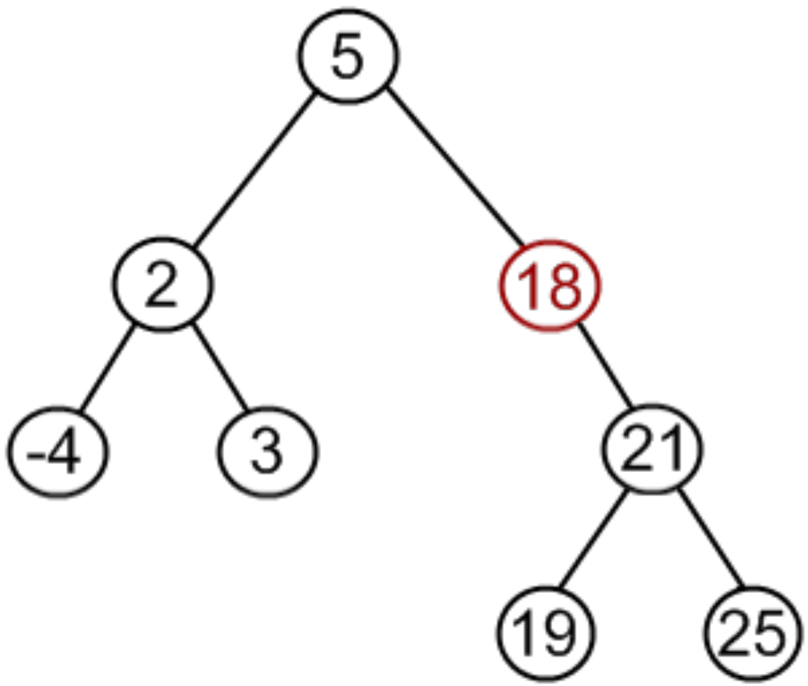
Remove Operation (I)

- Remove **'-4'** from a BST.



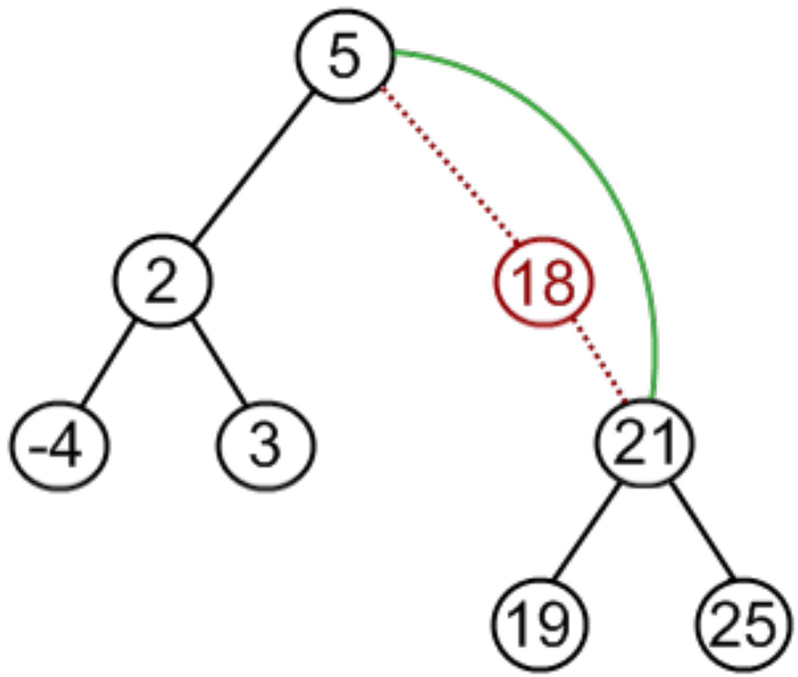
Remove Operation (II)

- Remove **'18'** from a BST.



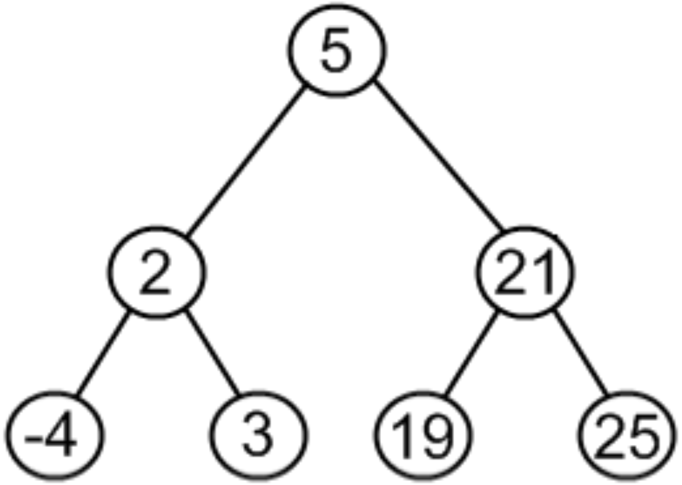
Remove Operation (II)

- Remove **'18'** from a BST.



Remove Operation (II)

- Remove **'18'** from a BST.

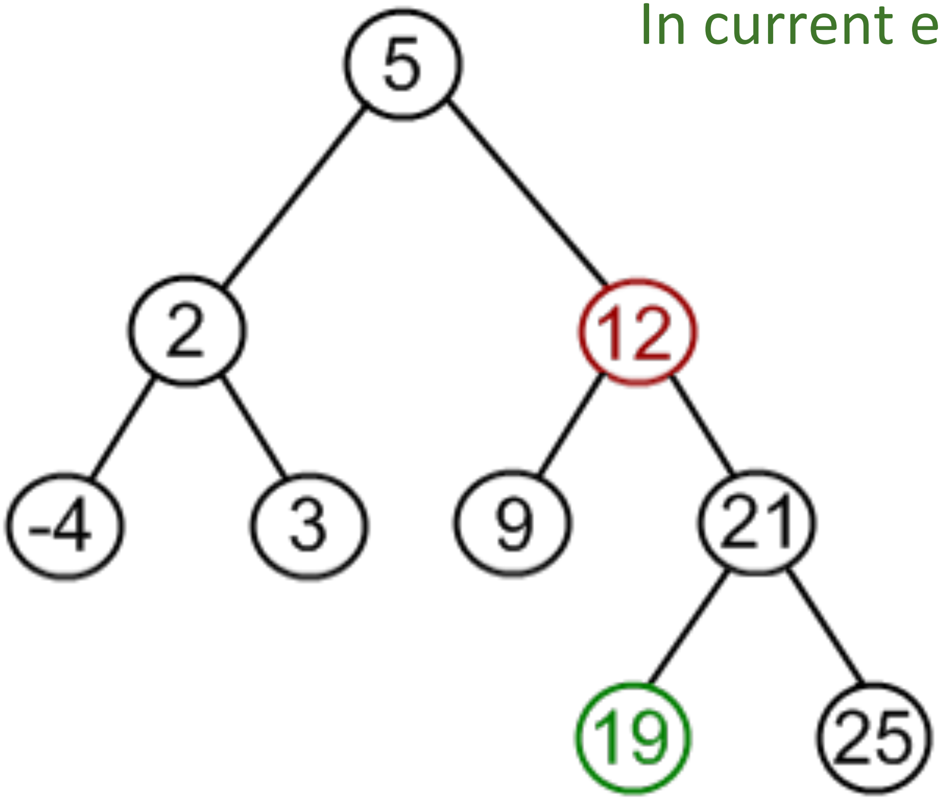


Remove Operation (III)

- Remove **’12’** from a BST.

Find minimum element in the right subtree of the node to be removed.

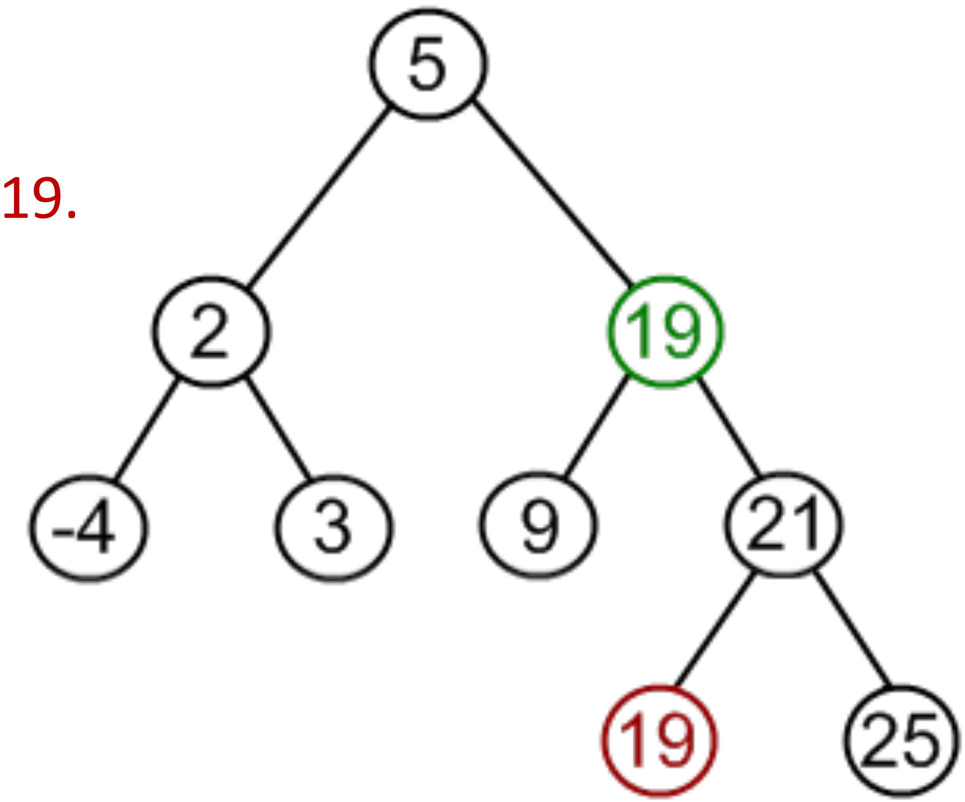
In current example it is 19.



Remove Operation (III)

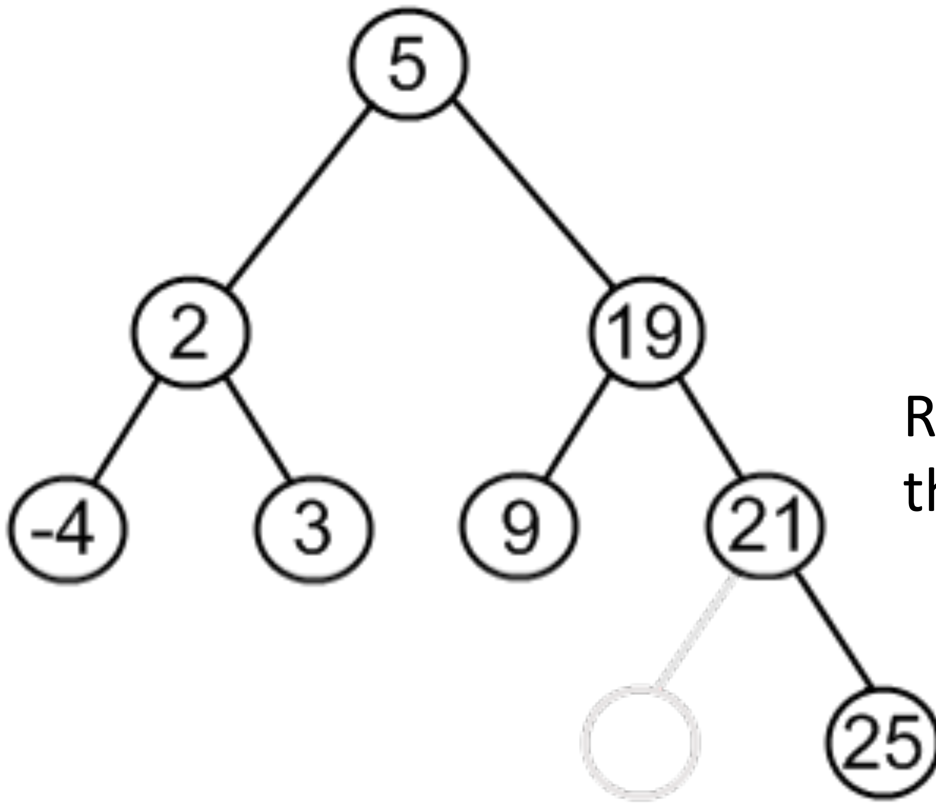
- Remove **’12’** from a BST.

Replace 12 with 19.



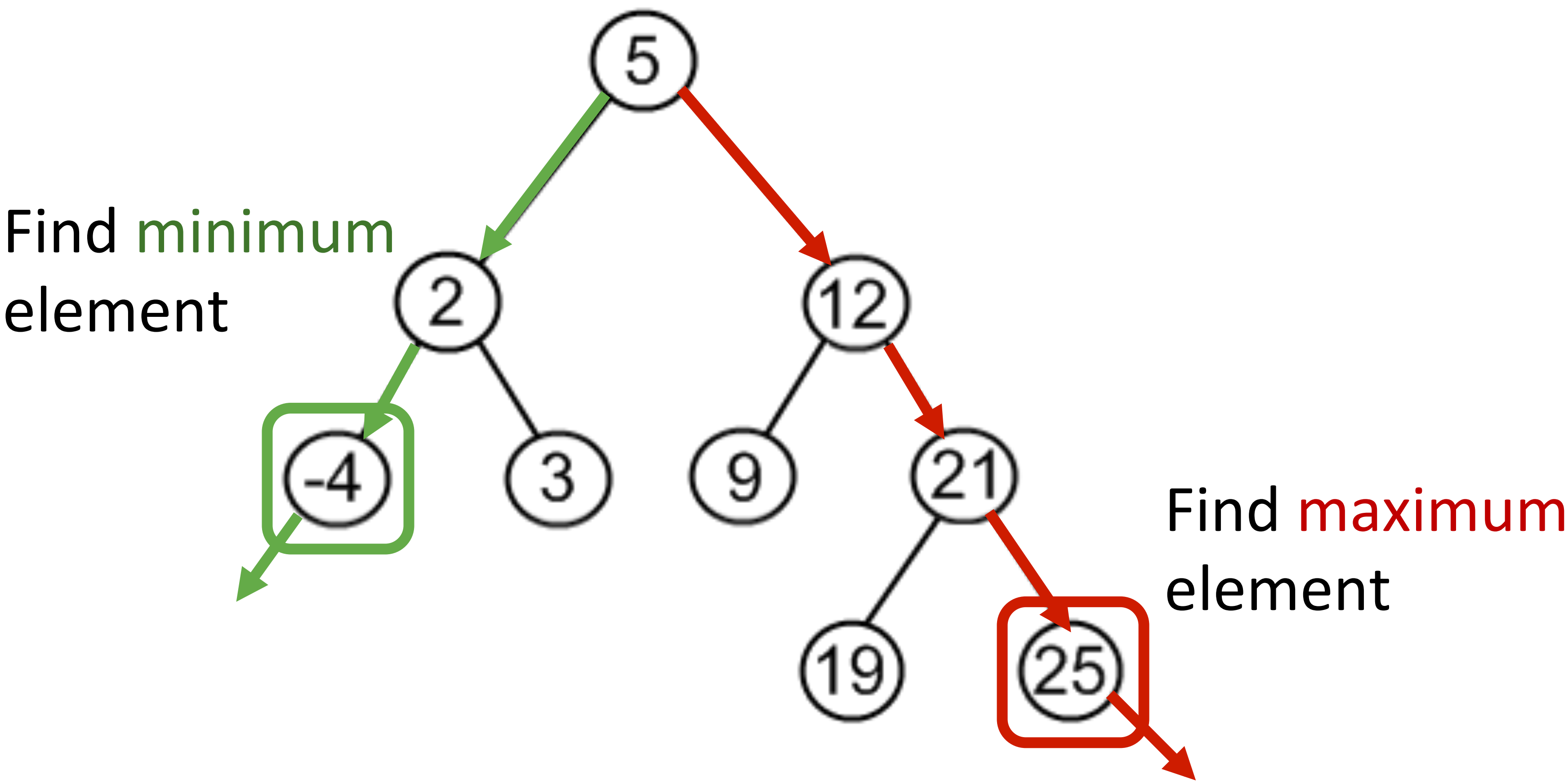
Remove Operation (III)

- Remove **'12'** from a BST.



Remove 19 from the left subtree.

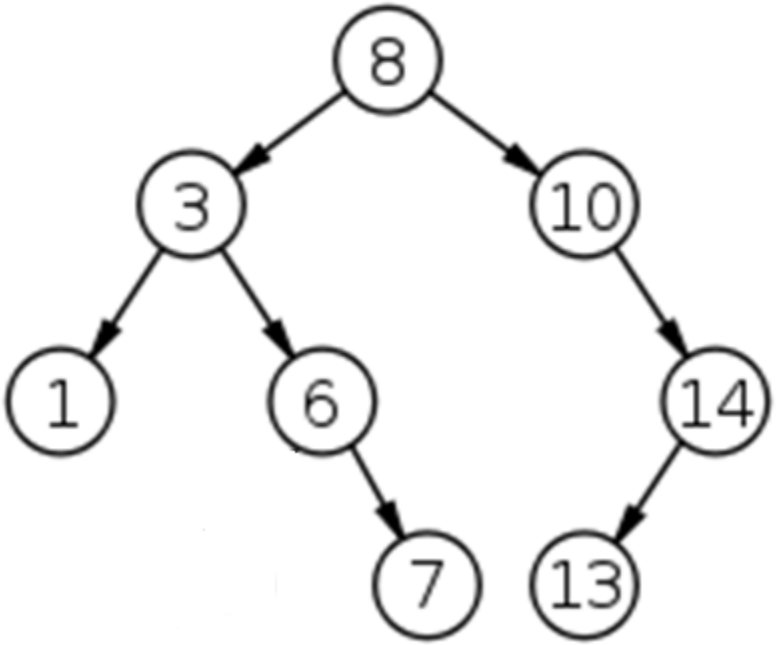
Other BST Operations



Exercise

Given the following BST:Is a Binary Tree with the following properties:

- Insert ‘4’
- Search for ‘9’
- Remove ‘1’
- Remove ’13’
- Remove ‘8’
- Find Min value
- Find Max value



Exercise

Draw the binary tree which would be created by inserting the following numbers in the order given:

50 30 25 75 82 28 63 70 4 43 74 35

BST IMPLEMENTATION

Basic BST Operations

- Insert a node into a BST.
- Search for a value within a BST.
- Remove a node from a BST.

Tree Implementation

```
typedef struct node_type{
    tree_entry info;
    struct node_type *right,*left;
} tree_node;

typedef tree_node *tree;
```

```
void insert(tree *t, tree_entry item){
 tree_node *p =(tree_node*)malloc(sizeof(tree_node));
 p->info = item;  p->left = NULL;  p->right = NULL;
  if (!(*t))
    *t= p;
 else{ tree_node *pre,*cur;
       cur = *t;
        while(cur){
          pre = cur;
          if(item < cur->info)
             cur = cur->left;
          else  cur = cur->right;
       }
       if(item <pre->info)  pre->left = p;
         else  pre->right = p;
  } }
```

```c
int delete(tree *t, tree_entry k){
    int found = 0; tree_node *q=*t, *r = NULL;
    while(q && !(found=(k==q->info))){
        r = q;
        if(k < q->info)     q = q->left;
        else                q = q->right;
    }
    if (found){
        if(!r) //Case of deleting the root
            delete_node(t);
        else if((k < r->info))
            delete_node(&r->left);
        else
            delete_node(&r->right);
    }
    return found;
}
```

```
void delete_node(tree *pt){
   tree_node*q=*pt;       tree_node*r=Null;
   if(!(*pt)->left)      *pt = (*pt)->right;
   else if(!(*pt)->right) *pt = (*pt)->left;
   else { //third case
     q = (*pt)->left;
     while(q->right){
        r = q;
        q = q->right;
     }
     (*pt)->info = q->info;
      if(!r)  (*pt)->left = q->left;
      else     r->right = q->left;
   }
   free(q);
}
```

CS 214 – DATA STRUCTURES FALL 2015

Lecture 8 – Binary Search Trees November 20th 2017

Group **B & C**

Dr. Mai Hamdalla

mai@fci.helwan.edu.eg