ET0736

Lesson 8

Implementation of ArrayList, LinkedList, Stack, Queue and Binary Search Tree

Topics

- Implementation of containers (ArrayList, LinkList, Stack, Queues)
- Binary Tree
- Binary Search Tree

Implementation of Containers

- ArrayList, LinkedList, Stacks and Queues are classic data structures typically covered in a data structure course
- They are supported and provided in Java Collections Framework.
- This lesson will examine how these structures are implemented under the hood.

Implementation of ArrayList

- Build a custom class called, MyArrayList
- Use array as the underlying storage.
- The array has an initial capacity of 10.
- Create a new array, with additional of 10 vacancies, when the size limit is reached and copy the content over (using Arrays.copyOf())
- For a start, include only these 2 methods ***add()*** and ***get()***:

```
class MyArrayList<E> {

   E a[] = (E[]) new Object[10];

   // add methods here

}
```

add(E)	add an element into the list
get (index) : E	Get the element at index

Implementation of ArrayList

MyArrayList

```
class MyArrayList<E> {
    int INITIAL_CAPACITY =10;
    E a[] = (E[]) new Object[INITIAL_CAPACITY];

    void add (E  x){
      if (a[a.length - 1]==null){
          int i=0;
          while (a[i]!=null)
              i++;
          a[i]=x;
      }
      else  {
          // increase size by 10
          int originalSize = a.length;
          int newSize = originalSize +10;
          E b[] = (E[]) new Object[newSize];
          b=Arrays.copyOf(a,newSize);
          a=b;
          a[originalSize]=x;
      }
    }

    E get(int index){
        return(a[index]);
    }
}
```

test program

```
public static void main(String[] args) {
  MyArrayList<Integer> x = new MyArrayList<>();
  for (int i=1; i<=50;i++)
      x.add(i*2);
  for (int i=10; i<15;i++)
      System.out.println(x.get(i));
}
```

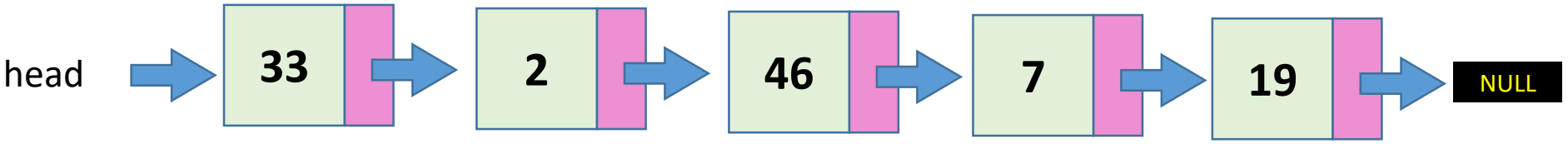
Output:

```
22
24
26
28
30
```

Implementation of LinkedList

Objectives:

- Build a custom class called, Node. A Node object is an element in the list.
- Build the actual class called, MyLinkedList.
- MyLinkedList keeps the record of the starting Node of the list, called ***head***.
- For a start, include only the ***append()*** which adds an element to the tail of the list



Implementation of LinkedList

First of all, implementing a node class for storing each element:

```
class Node<E> {
    E data;
    Node<E> next = null;

    Node (E data) {
      this.data = data;
    }
}
```

Next, implementing the MyLinkedList class in the next slide.

Implementation of LinkedList

```
class MyLinkedList<E> {
    Node<E> head = null;
    Node<E> current = null;
    Node<E> newNode;

    public void append (E x){
       newNode = new Node<E> (x);

       if (head==null) {
         // for very first mode
         head = new Node(x);
       }
       else {
          current = head;
          // track down to tail node
          while (current.next != null)
             current = current.next;

          // add in the new node
          current.next = newNode;
    }
}
```

append()

Add a new node to the end of the current list

Implementation of LinkedList

Next, add ***toString()*** to facilitate printing of elements.

```
class MyLinkedList<E> {
    Node<E> head = null;
    Node<E> current = null;
    Node<E> newNode;

    public void append (E x){
       newNode = new Node<E> (x);

       if (head==null) {
         // for very first mode
         head = newNode;
       }
       else {
          current = head;
          // track down to tail node
          while (current.next != null)
             current = current.next;

          // add in the new node
          current.next = newNode;
    }
```

```
  public String toString (){
     String s="";

     current = head;
     if (current.data!=null) {
          do {
            s += current.data.toString();
            current = current.next;

          } while (current != null);

       }

     return(s);
    }
}
```

Implementation of LinkedList

Create a LinkedList of String objects..

test program 1

```
public static void main(String[] args) {

  MyLinkedList<String> x = new MyLinkedList<>();
  x.append("I love SP");
  x.append("I love EEE");
  x.append("I love Java");
  System.out.println(x);
}
```

Output:

```
I love SP I love EEE I love Java
```

Implementation of LinkedList

Create a LinkedList of class ***Item*** objects..

```
class Item {
    String name;
    int qty;

    Item( String name, int qty){
        this.name = name;
        this.qty = qty;
    }

    public String toString() {
        return (name + "("+ qty + ")");
    }
}
```

test program 2

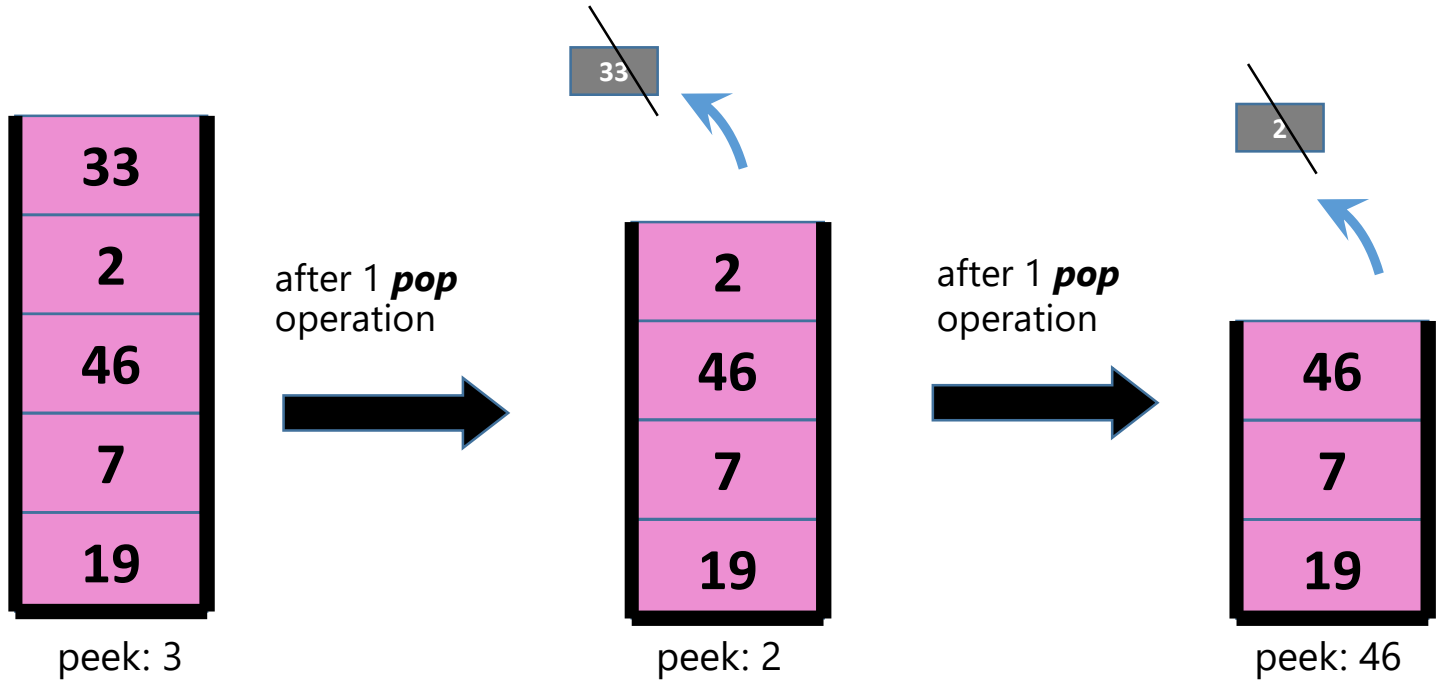
```
public static void main(String[] args) {

  MyLinkedList<Item> x = new MyLinkedList<>();
  x.append(new Item("CPU",8));
  x.append(new Item("Harddisk",5));
  x.append(new Item("Memory",32));
  System.out.println(x);
}
```

Output:

CPU(8) Harddisk(5) Memory(32)

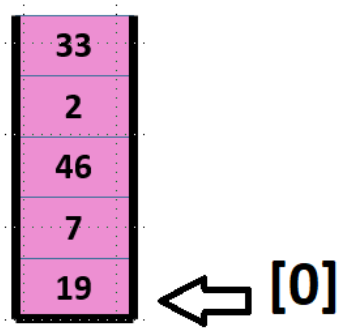
Implementation of Stack

- The stack is a linear data structure that is used to store the collection of objects in a Last-In-First-Out (LIFO) manner.
- Data can only be added (push)/removed (pop) from the 'top' of the stack



Implementation of Stack

- Build a custom class called, MyStack
- Use array as the underlying storage.
- The array has an initial capacity of 10.
- Create a new array, with additional of 10 vacancies, when the size limit is reached and copy the content over (using Arrays.copyOf())
- It is easier to assume that the index [0] of array represents the bottom of the stack
- For a start, include only these 3 methods ***push(), pop()*** and ***toString()***:




```
class MyStack<E> {

   E a[] = (E[]) new Object[10];

   // add methods here

}
```

push(E)	add an element onto top of the stack
pop ()	remove an element from top of
toString()	Display the objects

Implementation of Stack

MyStack

```
class MyStack<E> {
    int INITIAL_CAPACITY =10;
    E a[] = (E[]) new Object[INITIAL_CAPACITY];

    void push(E  x){
      if (a[a.length - 1]==null){
          int i=0;
          while (a[i]!=null)
              i++;
          a[i]=x;
      }
      else  {
          // increase size by 10
          int originalSize = a.length;
          int newSize = originalSize +10;
          E b[] = (E[]) new Object[newSize];
          b=Arrays.copyOf(a,newSize);
          a=b;
          a[originalSize]=x;
      }
    }

    E get(int index){
        return(a[index]);
    }
}
```

```
    void pop(){
       int i=0, last =0;
       while (a[i]!=null){
           last = i;
           if  (i== a.length)
              break;
           i++;
       }
       if (a[last]!= null)
          a[last] = null;
    }

    public String toString (){
      String s="";
      int i=0, last =0;
      while ((a[i]!=null) && (i<a.length)) {
         s += a[i].toString() + " ";
         i++;
      }
      return(s);
    }

}
```

Implementation of Stack

Create a Stack of class ***Item*** objects.

```
class Item {
    String name;
    int qty;

    Item( String name, int qty){
        this.name = name;
        this.qty = qty;
    }

    public String toString() {
        return (name + "("+ qty + ")");
    }
}
```

test program

```
public static void main(String[] args) {

  MyStack<Item> x = new Stack<>();
  x.push(new Item("CPU",8));
  x.push(new Item("Harddisk",5));
  x.push(new Item("Memory",32));
  System.out.println(x);
  x.pop();
  System.out.println(x);

}
```

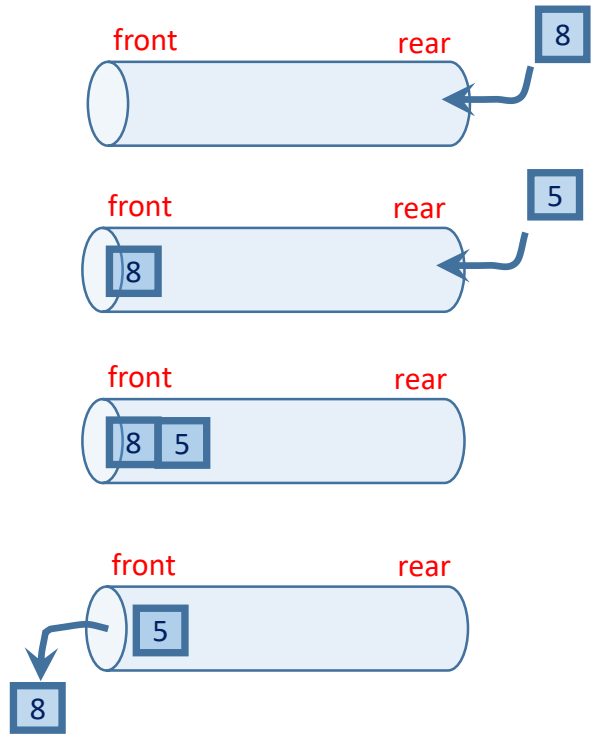
Output:

CPU(8) Harddisk(5) Memory(32)
CPU(8) Harddisk(5)

(last-in-first-out)

Recap: Custom Generic Class

Assuming that we want to defines a custom class ***MyQueue*** to simulate a queue data structure., which is a First-In-First-Out protocol – just like a line in front of a hawker stall.



Assuming that the class is supposed to support these 3 simple operations:

addToRear() : void
popFromFront() : E
size() : int

Assuming that internally, we use an ArrayList to store the elements.

Recap: Custom Generic Class

Use <E> to represent the generic type.

Class MyQueue

```
class MyQueue<E>{
    ArrayList<E> a = new ArrayList<>();

    void addToRear(E newObj){
        a.add(newObj);
    }

    E popFromFront() {
       E obj;
       if (a.size()>0){
         obj = a.get(0);
         a.remove(0);
          return (obj);
       }
        return(null);
    }

    int size(){
        return a.size();
    }
}
```

Test program

```
public static void main(String[] args) {
   MyQueue<Integer> q1  = new MyQueue<>();
   MyQueue<String> q2  = new MyQueue<>();
   q1.addToRear(111);
   q1.addToRear(222);
   q1.addToRear(333);
   q2.addToRear("SP");
   q1.popFromFront();

   System.out.println (q1.size());
   System.out.println (q2.size());
}
```

Output:

```
2
1
```

Recap: Custom Generic Method

Using the generic class MyQueue.
Create the following method to convert an array into a MyQueue object.

```
public static  <E> MyQueue<E> fromArrayToMyQueue(E arr[]){
      MyQueue<E> a = new MyQueue<>();
      for (int i=0; i<arr.length; i++){
          a.addToRear(arr[i]);
      }
      return(a);
  }
```

Output:

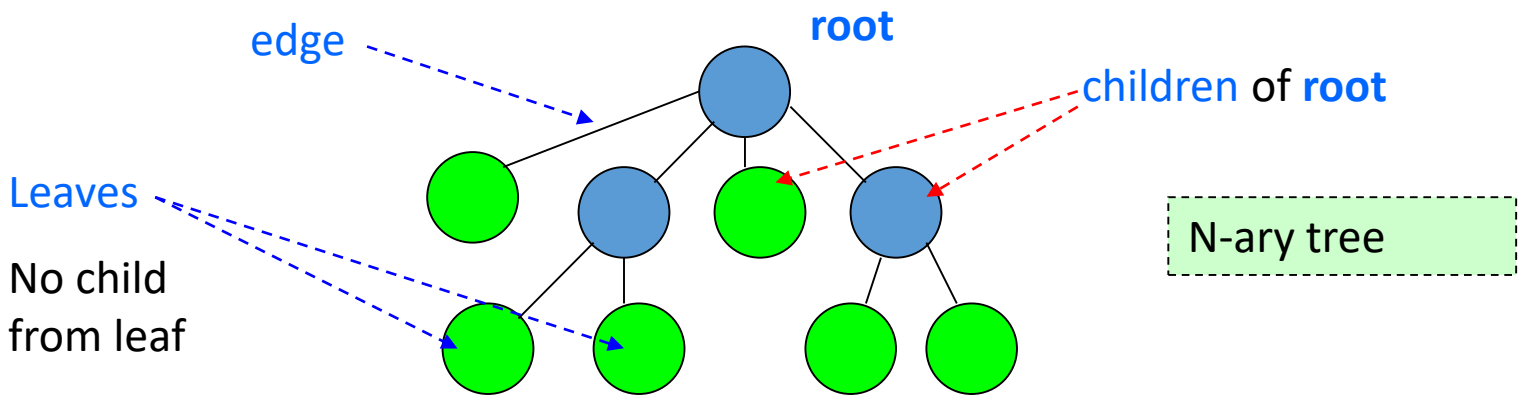
```
5
```

```
public static void main(String[] args) {
   Integer a[] = {111,222,333,444,555};
   MyQueue<Integer> q1  = fromArrayToMyQueue(a);
   System.out.println(q1.size());
}
```

Test program

A tree is a collection of nodes

- It has a root node from which sprouts child/children nodes:



A leaf is a node with no children

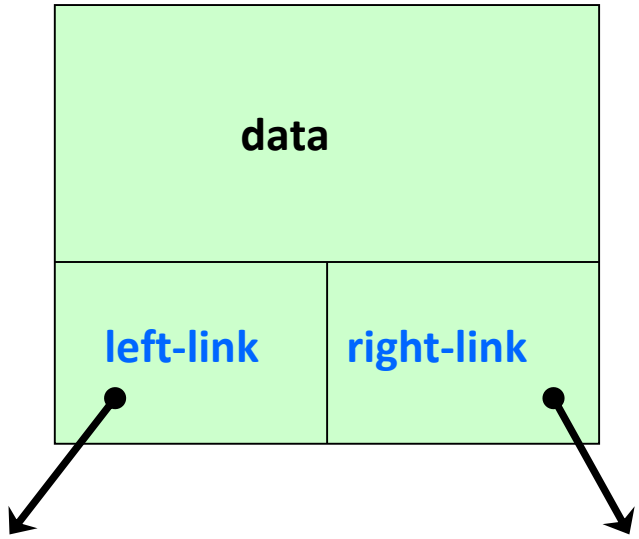
- A binary tree is one where each node(including the root) has up to two branches – a left and a right branches. The branch may be empty – no child
- Nodes are linked to their parents with edges
- We will focus on binary trees for this study

What a tree node consists of:

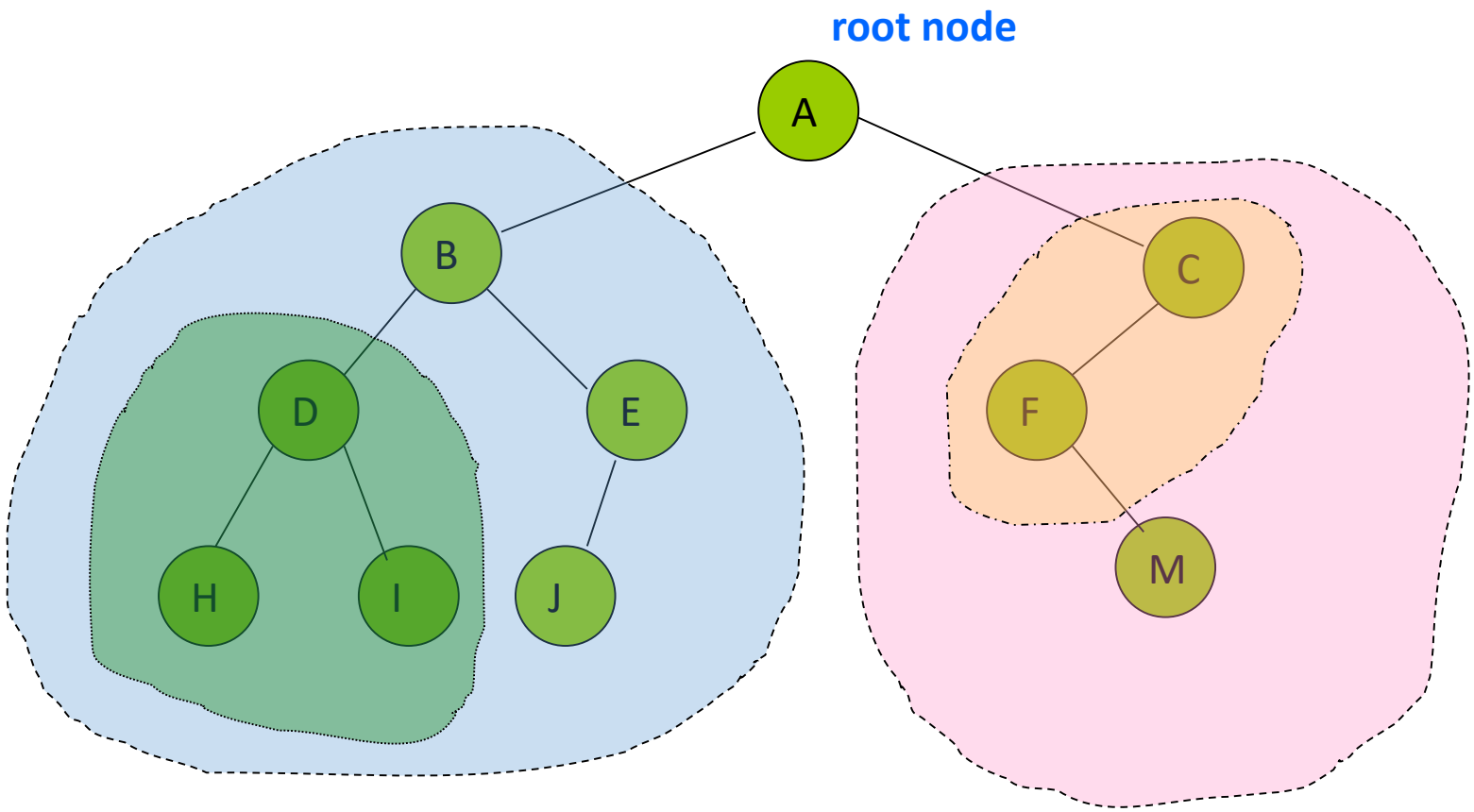
- Each node contains a field for data and a left-link and a right-link

Example:

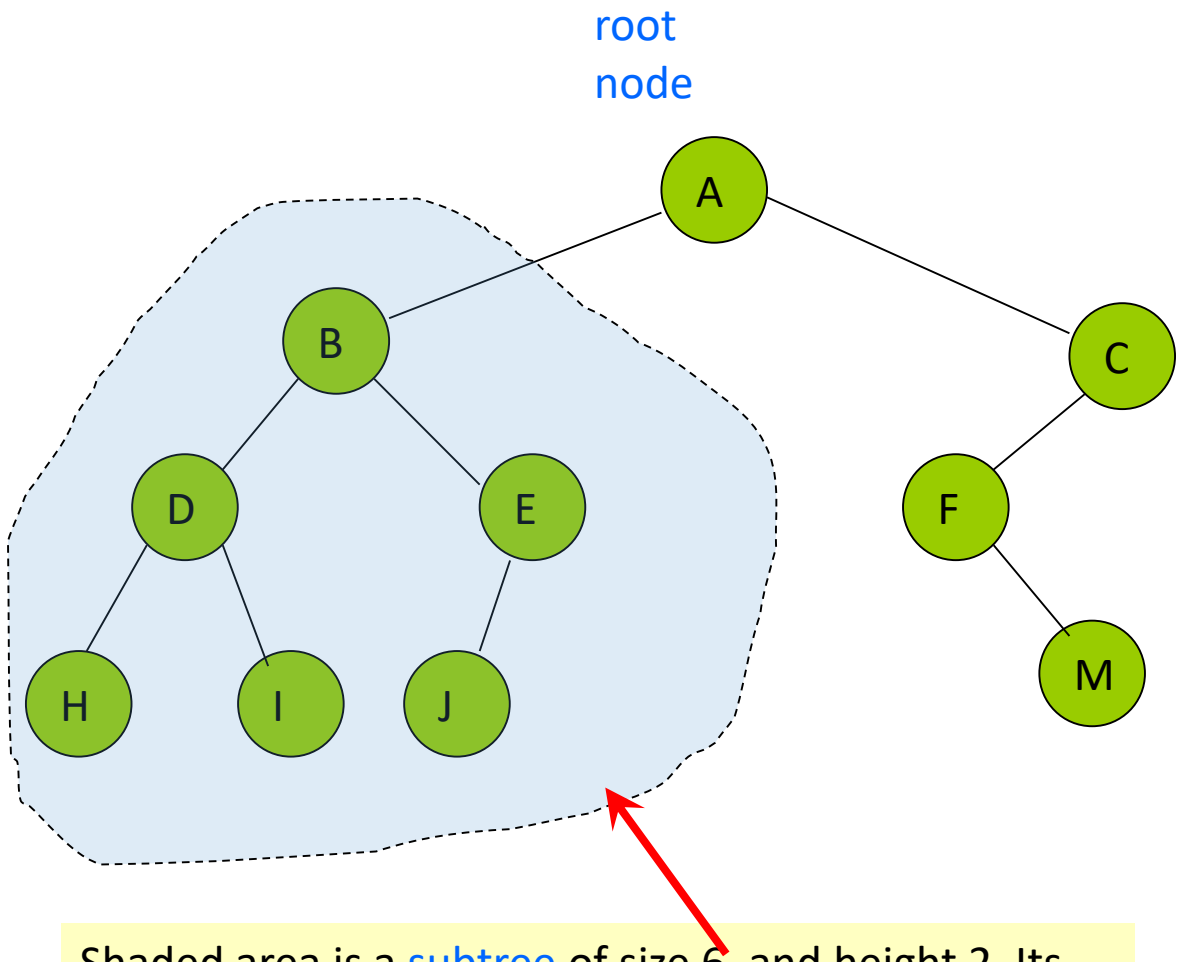
```
class Node
{
    int data;
    Node  left = null;
    Node  right = null;
}
```



- A binary tree has at most two children from each of its nodes:



Binary Tree Terminology:



Shaded area is a subtree of size 6, and height 2. Its root is B.

Size: 10

Height: 3

Root: A

Path from B to H has length 2

Node A has a left child B and a right child C

B is at level 1, H at level 3;

B is an ancestor D, H, J,…

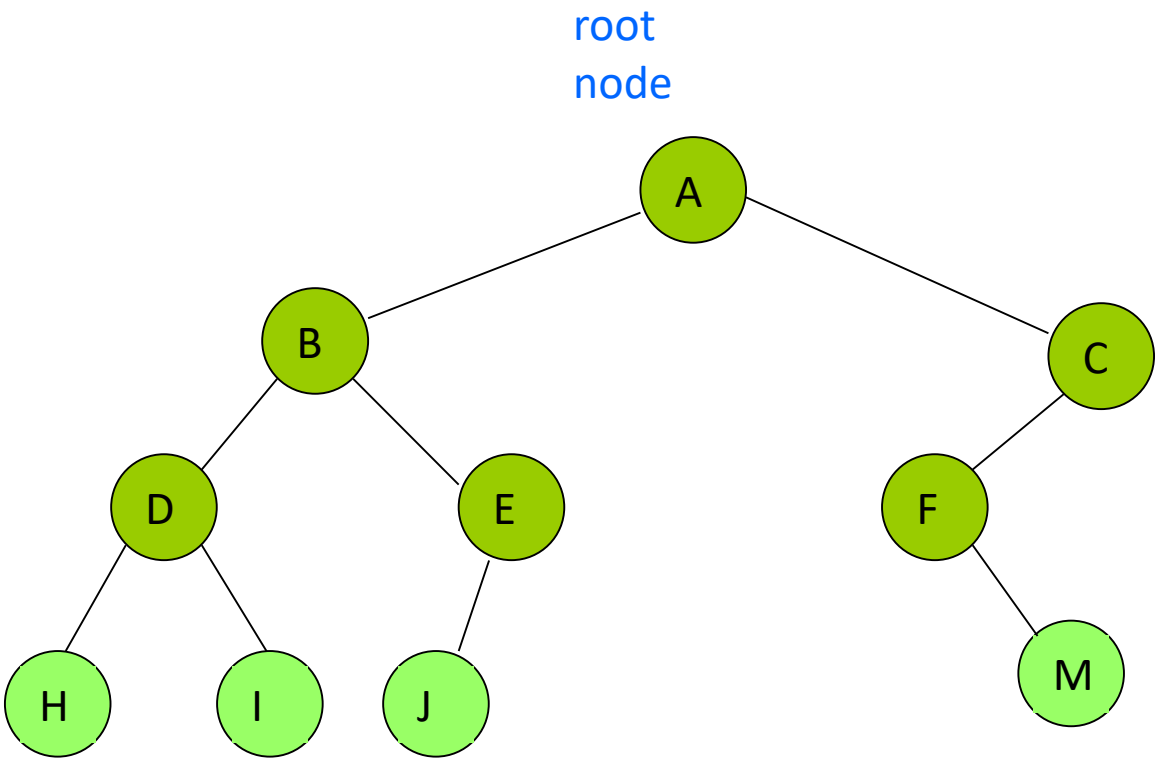
M is a descendant of F, C,..

D is the parent of H and I

D and E are siblings

H and I are sibilings

Binary Tree Terminology(2):

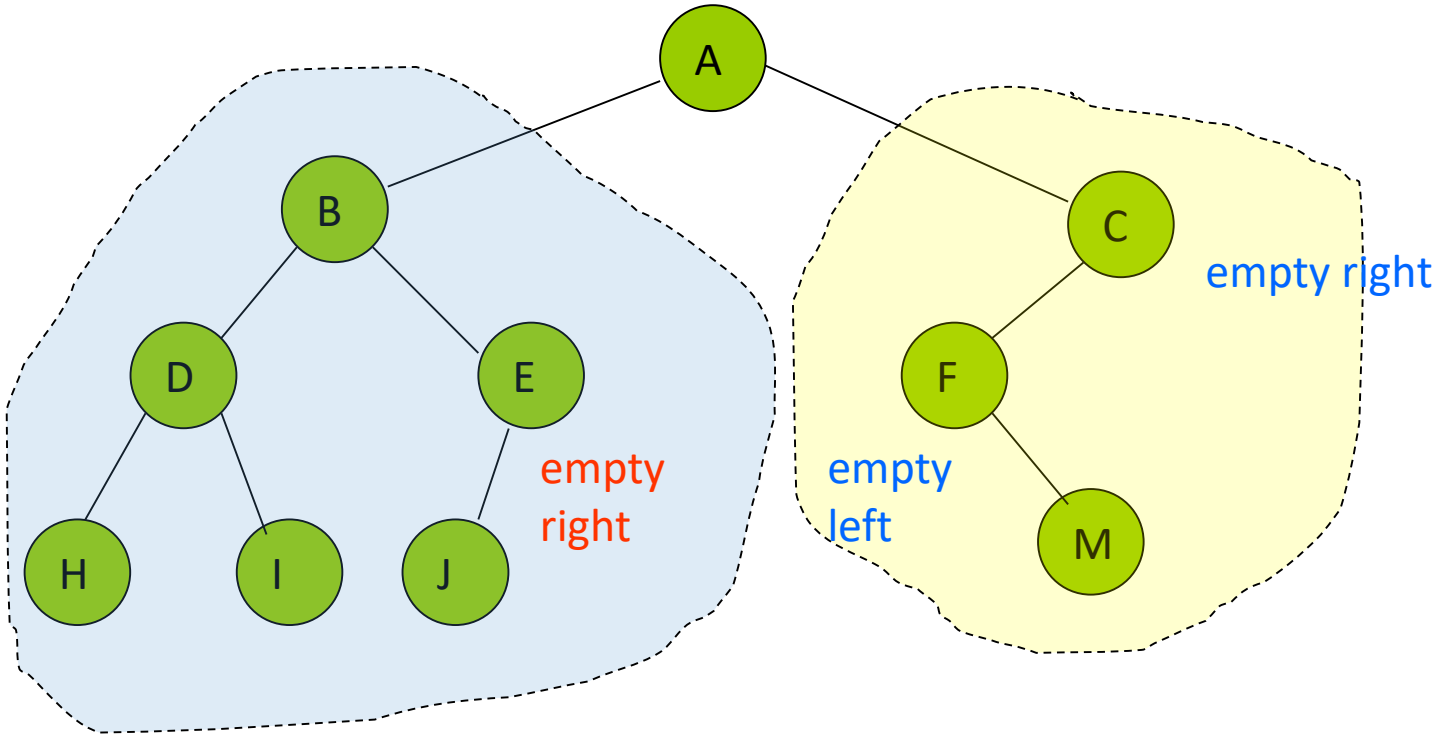


B is a child of A; E is a child of B; A is the parent of B and C;

B is a parent of E and D.
H, I,J & M are leaves as they don't have children

H, I are siblings as they have the same parent

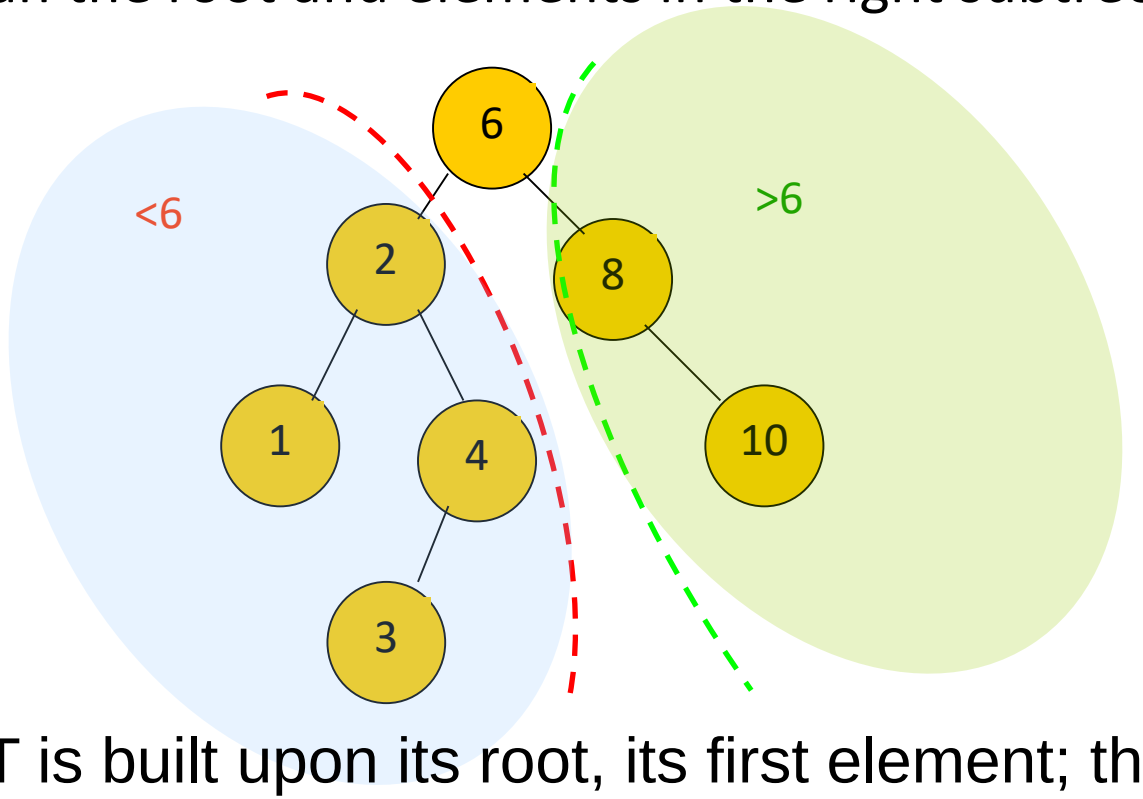
binary tree node has not more than two children:



- Each node will have another two branches
  - A branch may be empty
  - The branches are respectively the **left** and the **right** branches or subtrees

Binary Search Tree

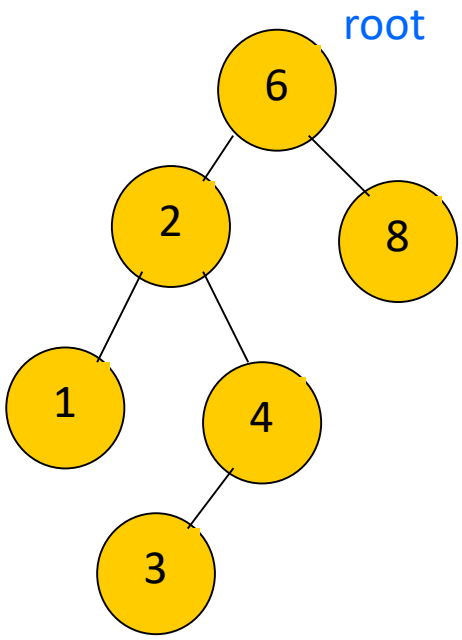
- A BST orders the elements such that elements in the left subtree are always less than the root and elements in the right subtree are always more.

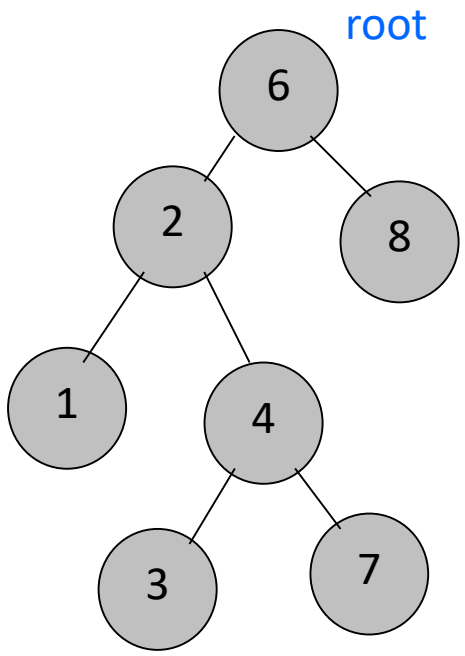


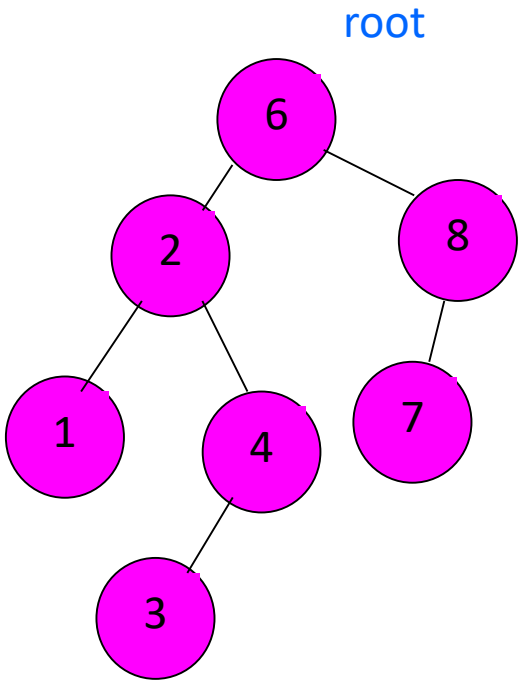
  - The BST is built upon its root, its first element; the next elements are arranged, left, if it is smaller and right if it is larger than the root.

Binary Search Tree(BST)

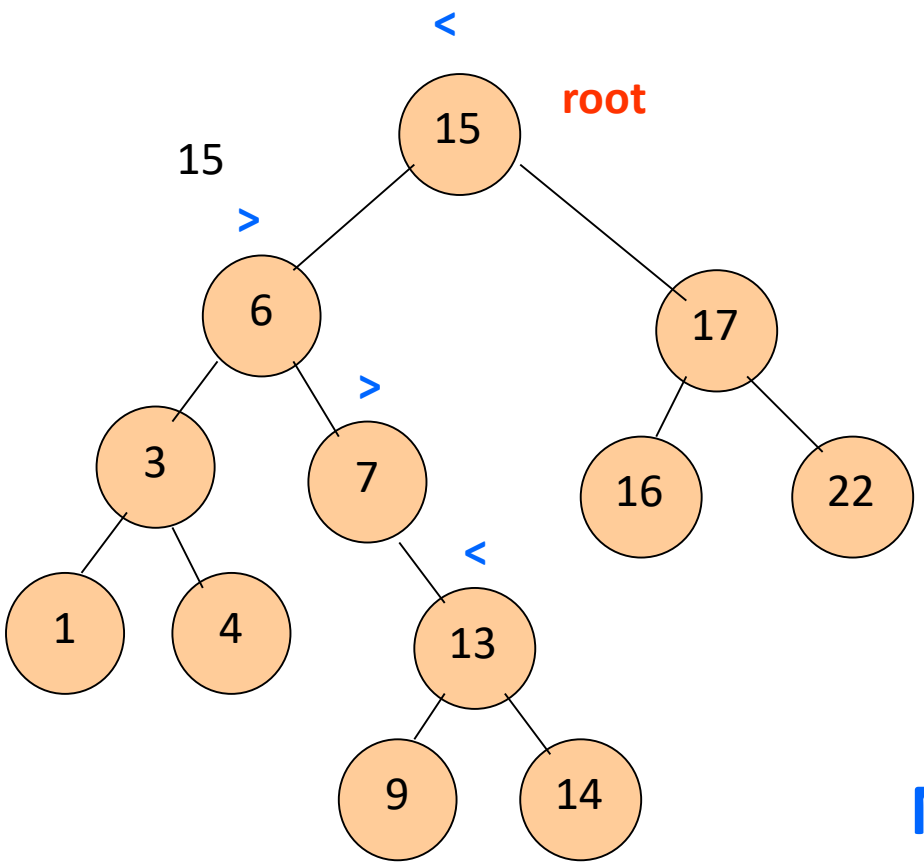
Which is not a BST?







A Binary Search on BST:



Start at root:

Compare 9 to 15(root), go left subtree

Compare 9 to 6, go right subtree

Compare 9 to 7, go right subtree

Compare 9 to 13, go left subtree

Compare 9 to 9, succeed.

Number of comparisons is not more than (`height+1`) of tree, in this case here 5

Quick Exercise(10min)

http://www.cosc.canterbury.ac.nz/mukundan/dsal/BST.html

http://www.cs.jhu.edu/~goodrich/dsa/trees/btree.html

- demo of insertion, deletion & searching in a Binary Tree structure

Enter the number sequence into the animated demo and see how the binary search tree is being built:

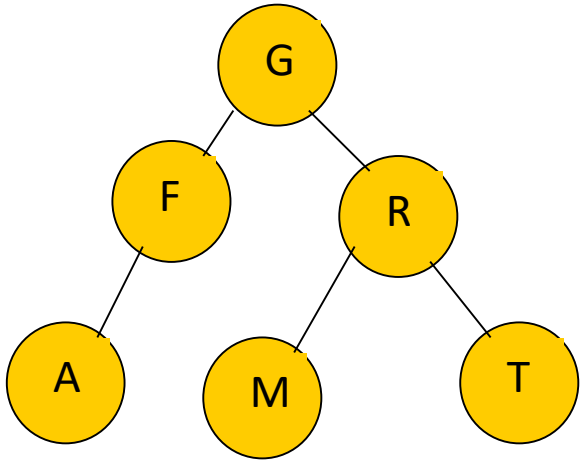
15, 6, 3, 17, 16, 4, 1, 7, 22, 13, 14, 8

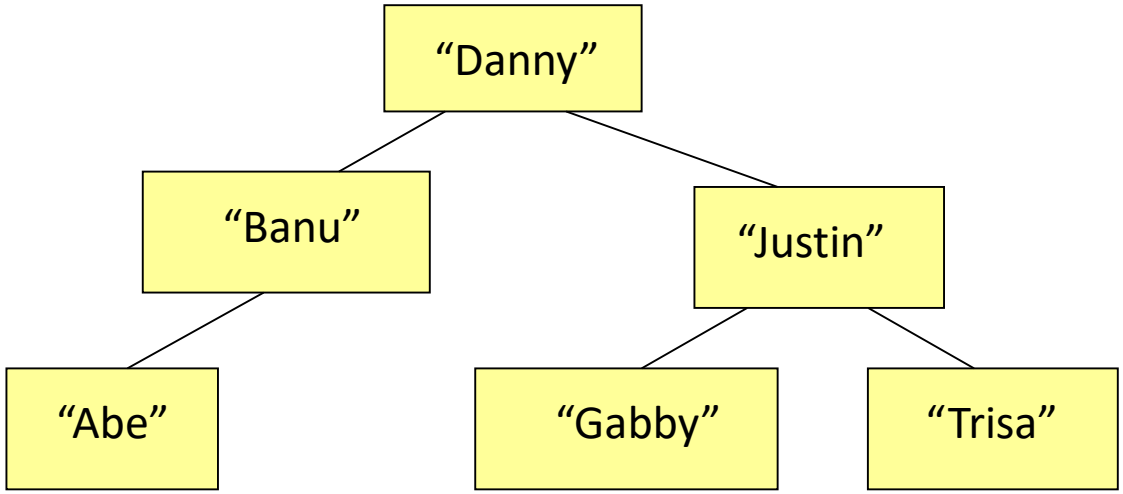
What is the height of this tree?

What is the max. number of comparisons to search for a number in this tree?

Binary Search Tree(BST)

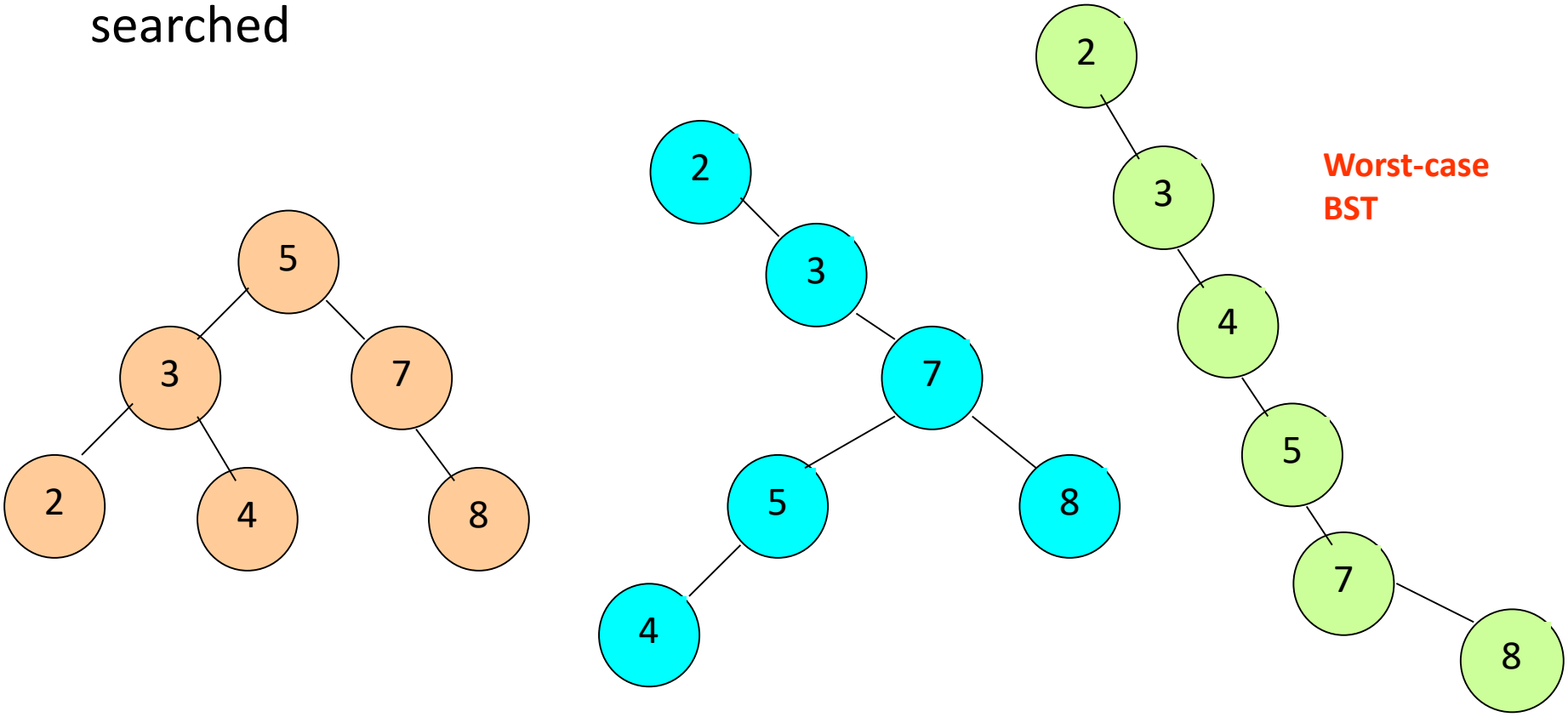
- The elements can be characters or string or any data so long as the comparison operators “>” and “<“ make sense on the elements





From the same data set, you can create different BSTs,

- It all depends on which is designated a root node and the sequence that follows when you enter the numbers into a BST. The different heights determine how quickly an element can be searched



Binary Tree Traversal

- Data are stored as elements of a binary tree
- Traversal means visiting the elements of a tree exactly once.
- There are 3 different orders one can follow:
  - **Inorder**
  - **Preorder**
  - **Postorder**

Traversal Rules

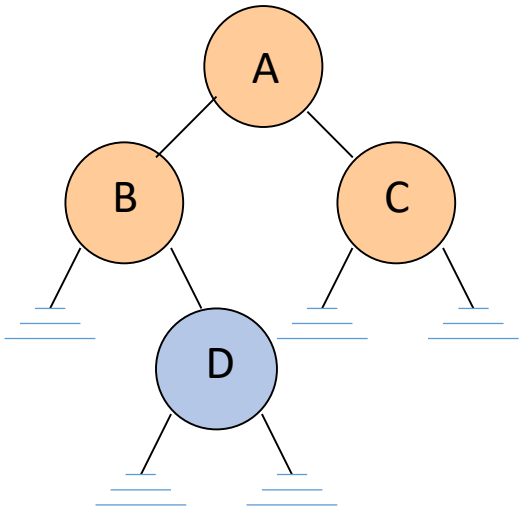
- Inorder
  - Traverse the left subtree
  - Visit the current parent
  - Traverse the right subtree
- Preorder
  - Visit the current parent
  - Traverse the left subtree
  - Traverse the right subtree
- Postorder
  - Traverse the left subtree
  - Traverse the right subtree
  - Visit the current parent

Quick Exercise(10min)

Inorder:

left, current parent, right

B D A C



preorder:

current parent, left, right

A B D C

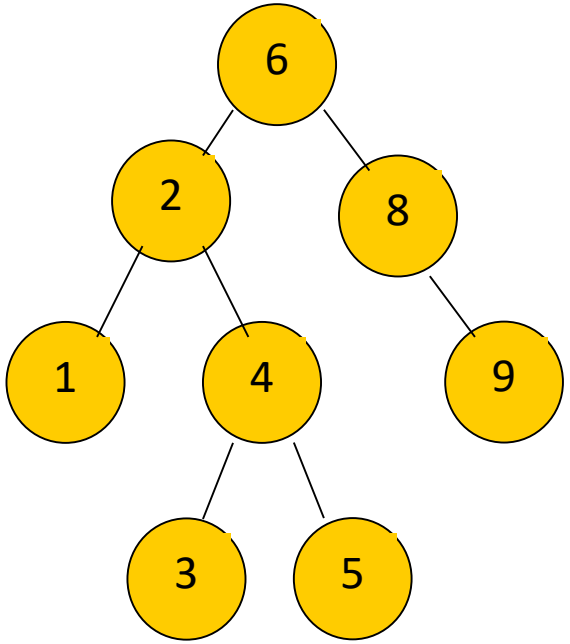
postorder:

left, right, current parent,

B D C A

What is the traversal sequence?

inorder?



postorder?

preorder?

Implementation of BST

- Build a custom class called, MySimpleBST
- This super simple BST will take integer data
- It will take in only 3 integers
- The 1st integer will be the ***root*** node
- The 2nd and 3rd integer – make it such that one of them > root and the other < root
- For a start, include only these 2 methods ***append() and postOrderTraversal()***:

```
class Node {
    int data;
    Node left = null;
    Node right = null;

    Node (int data) {
      this.data = data;
    }
}
```

append(int)	add an integer into the BST
postOrderTraversal ()	Visit nodes (by post-order) and display integers

Implementation of BST

```
class MySimpleBST {
    Node root = null;
    void append(int x) {
        Node newNode = new Node (x);
        if (root == null){
            root = newNode;
        }
        else {
            if (newNode.data < root.data){
                root.left = newNode;
            }
            else if (newNode.data > root.data){
                root.right = newNode;
            }
        }
    }

    void postOrderTraversal(){
        if (root!=null){
            if (root.left!=null){
                System.out.println(root.left.data );
            }

            if (root.right!=null){
                System.out.println(root.right.data );
            }
            System.out.println(root.data);
        }
    }
}
```

test program

```
public static void main(String[] args) {

   MySimpleBST bst = new MySimpleBST();
   bst.append(10);
   bst.append(99);
   bst.append(3);
   bst.postOrderTraversal();
}
```

Output:

3
99
10