FAKULTI TEKNOLOGI
KEJURUTERAAN KELAUTAN
DAN INFORMATIK

2022/2023

DATA STRUCTURE & ALGORITHM

Lab 7: Binary Search Tree & Priority Queue

VERSION 1

STUDENT INFORMATION

PLEASE FILL IN YOUR DETAILS:

NAME: OMAR ISMAIL ABDJALEEL ALOMORY

MATRIC NUMBER:S63955

GROUP:K2

LAB:MP3

DATE:3/1/2023

TABLE OF CONTENTS

INSTRUCTIONS

Manual makmal ini adalah untuk kegunaan pelajar-pelajar Fakulti Teknologi Kejuruteraan Kelautan dan Informatik, Universiti Malaysia Terengganu (UMT) sahaja. Tidak dibenarkan mencetak dan mengedar manual ini tanpa kebenaran rasmi daripada penulis.

Sila ikuti langkah demi langkah sebagaimana yang dinyatakan di dalam manual.

This laboratory manual is for use by the students of the Faculty of Ocean Engineering Technology and Informatics, Universiti Malaysia Terengganu (UMT) only. It is not permissible to print and distribute this manual without the official authorisation of the author.

Please follow step by step, as described in the manual.

TASK 1: IMPLEMENTING BINARY SEARCH TREE (BST)

OBJECTIVE

In this task, students must be able to:

- Understand the concept of Binary Search Tree (BST).
- Implement BST.

ESTIMATED TIME

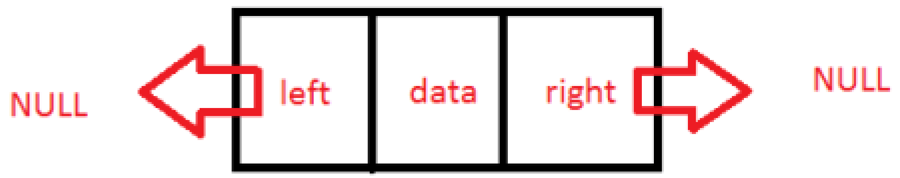
[60 Minutes]

DEFINITION OF BINARY SEARCH TREE

Binary Tree : A data structure in which we have nodes containing data and two references to other nodes, one on the left and one on the right.

Binary Tree consist of Nodes

- Nodes are nothing but objects of a class and each node has data and a link to the left node and right node.
- Usually we call the starting node of a tree as *root*.
- Left and right node of a Leaf node points to NULL so you will know that you have reached to the end of the tree.

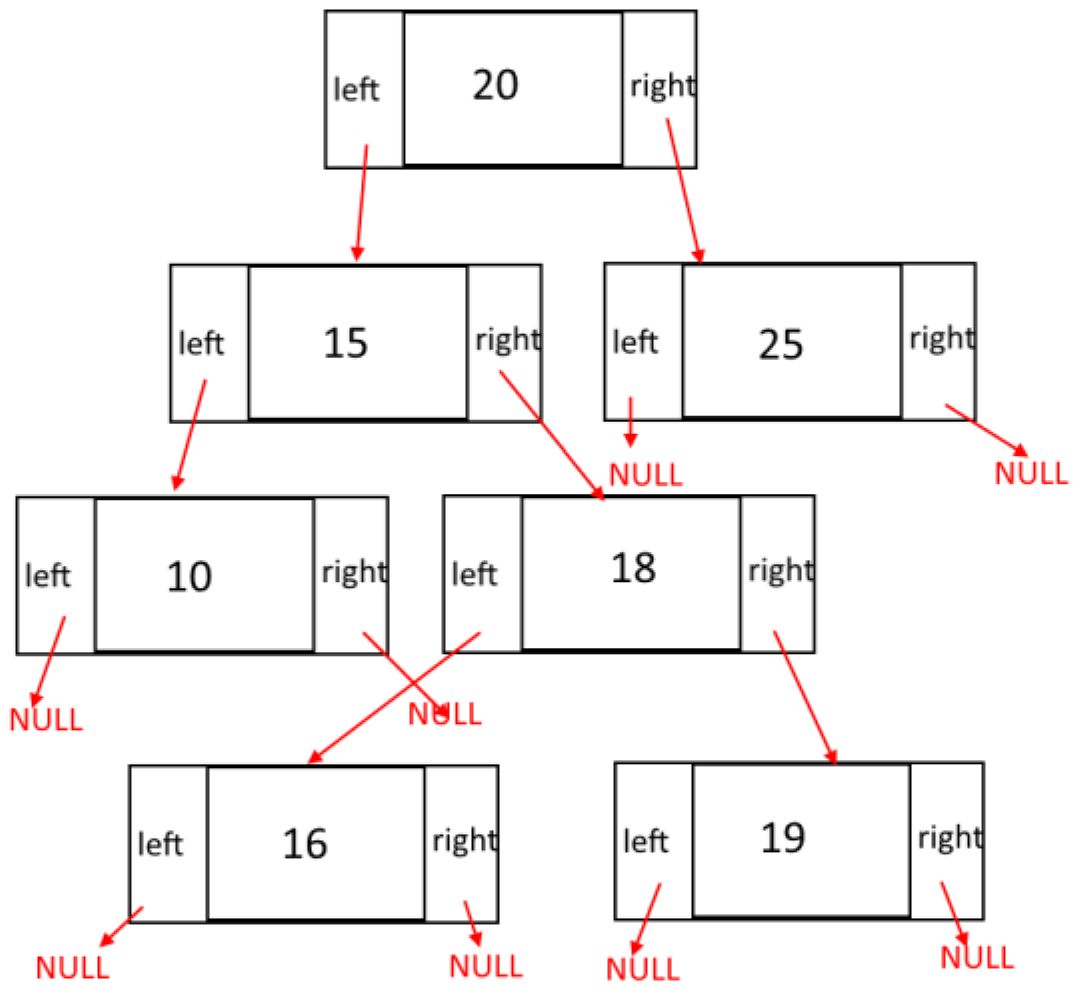


Node

Binary Search Tree:

Often we call it as BST, is a type of Binary tree which has a special property.

Nodes smaller than root goes to the left of the root and Nodes greater than root goes to the right of the root.



Operations:

Insert(int n) : Add a node the tree with value n. Its O(lgn)

Find(int n) : Find a node the tree with value n. Its O(lgn)

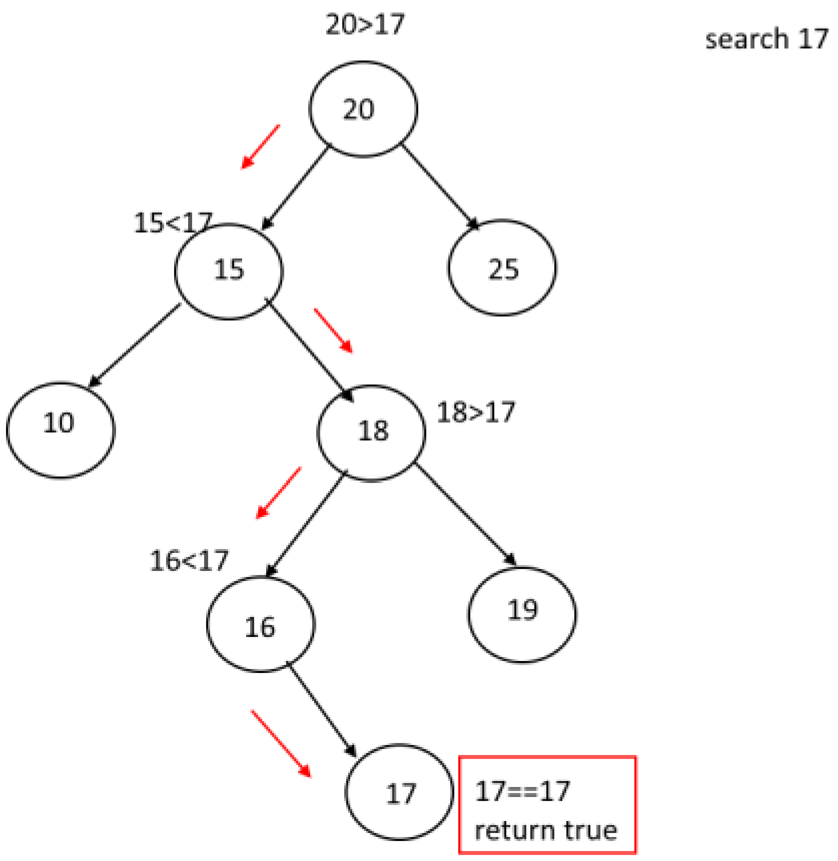
Delete (int n) : Delete a node the tree with value n. Its O(lgn)

Display(): Prints the entire tree in increasing order. O(n).

Detail Explanations for the Operations:

Find(int n):

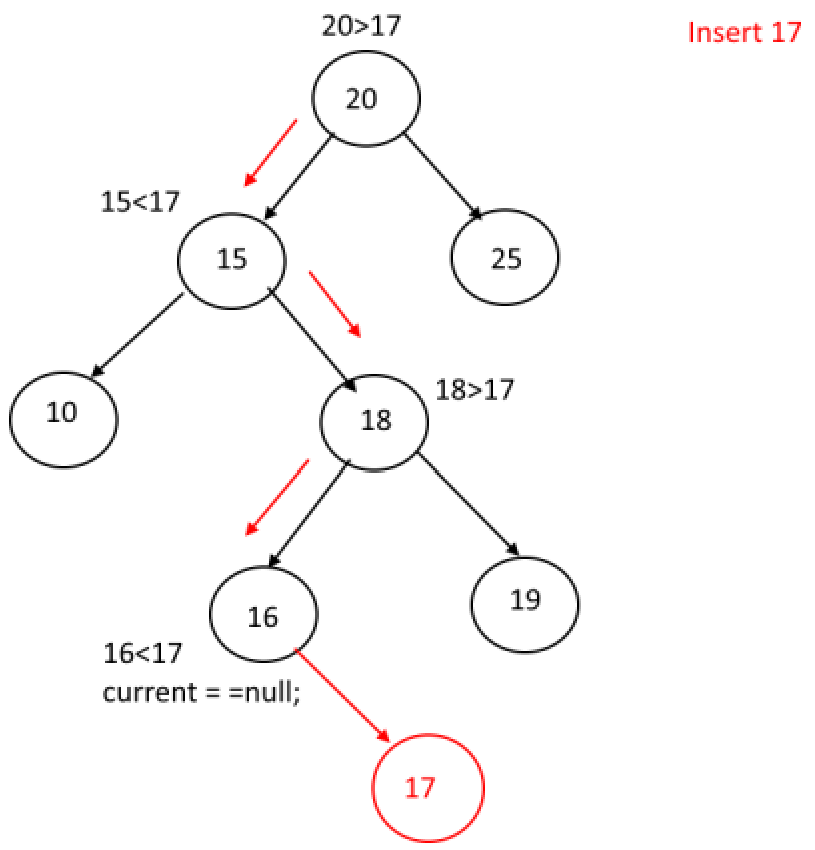
- Its very simple operation to perform.
- start from the root and compare root.data with n
- if root.data is greater than n that means we need to go to the left of the root.
- if root.data is smaller than n that means we need to go to the right of the root.
- if any point of time root.data is equal to the n then we have found the node, return true.
- if we reach to the leaves (end of the tree) return false, we didn't find the element



Insert(int n):

- Very much similar to find() operation.
- To insert a node our first task is to find the place to insert the node.

- Take current = root .
- start from the current and compare root.data with n
- if current.data is greater than n that means we need to go to the left of the root.
- if current.data is smaller than n that means we need to go to the right of the root.
- if any point of time current is null that means we have reached to the leaf node, insert your node here with the help of parent node. (See code)



Delete(int n):

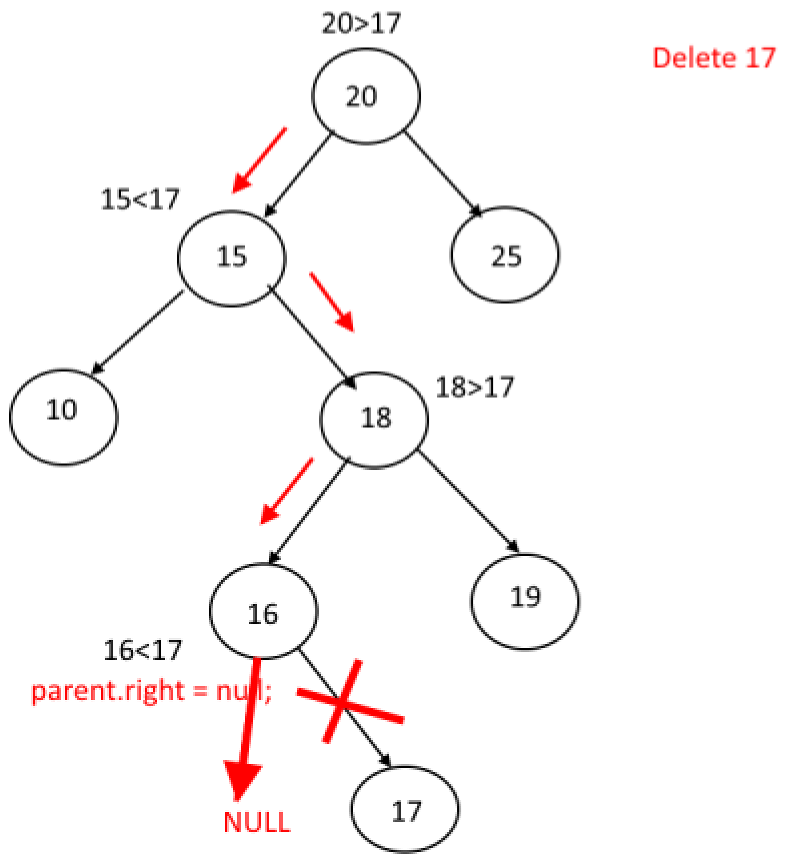
Complicated than Find() and Insert() operations. Here we have to deal with 3 cases.

- Node to be deleted is a leaf node (No Children).
- Node to be deleted has only one child.

- Node to be deleted has two childrens.

Node to be deleted is a leaf node (No Children).

its a very simple case, if a node to be deleted has no children then just traverse to that node, keep track of parent node and the side in which the node exist(left or right) and set ***parent.left = null or parent.right = null;***

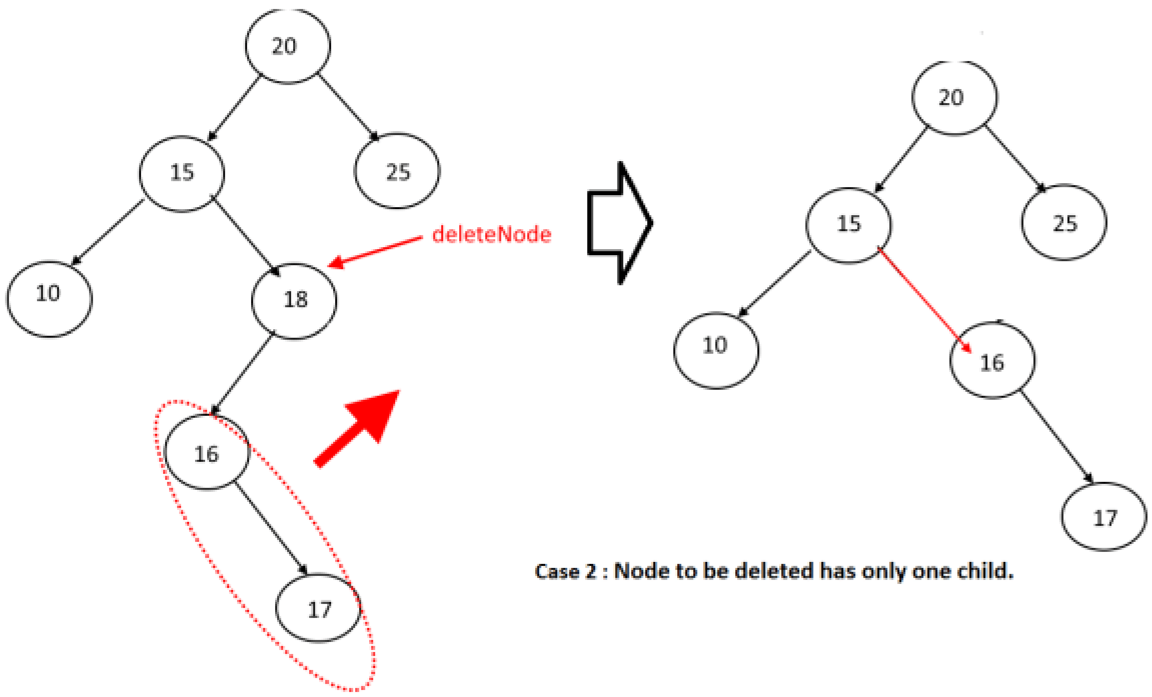


Case 1 : Node to be deleted is a leaf node (No Children).

Node to be deleted has only one child.

1. its a slightly complex case. if a node to be deleted(deleteNode) has only one child then just traverse to that node, keep track of parent node and the side in which the node exist(left or right).
2. check which side child is null (since it has only one child).

3. Say node to be deleted has child on its left side . Then take the entire sub tree from the left side and add it to the parent and the side on which deleteNode exist, see step 1 and example.

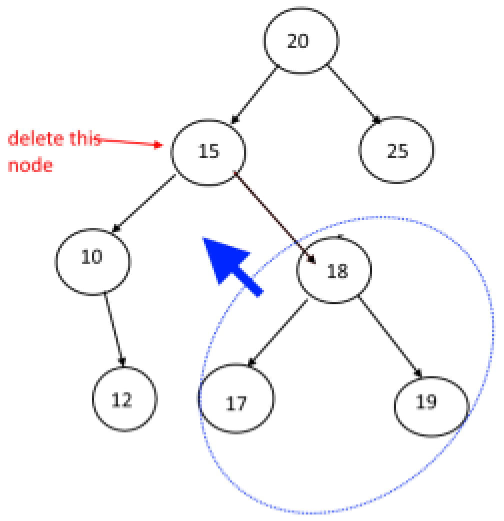


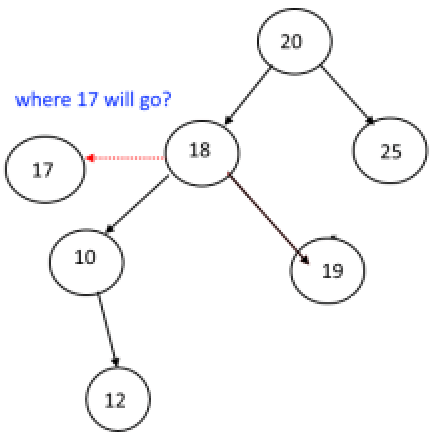
Case 2 : Node to be deleted has only one child.

Node to be deleted has two children.

Now this is quite exciting

You just cannot replace the deleteNode with any of its child, Why? Lets try out a example.





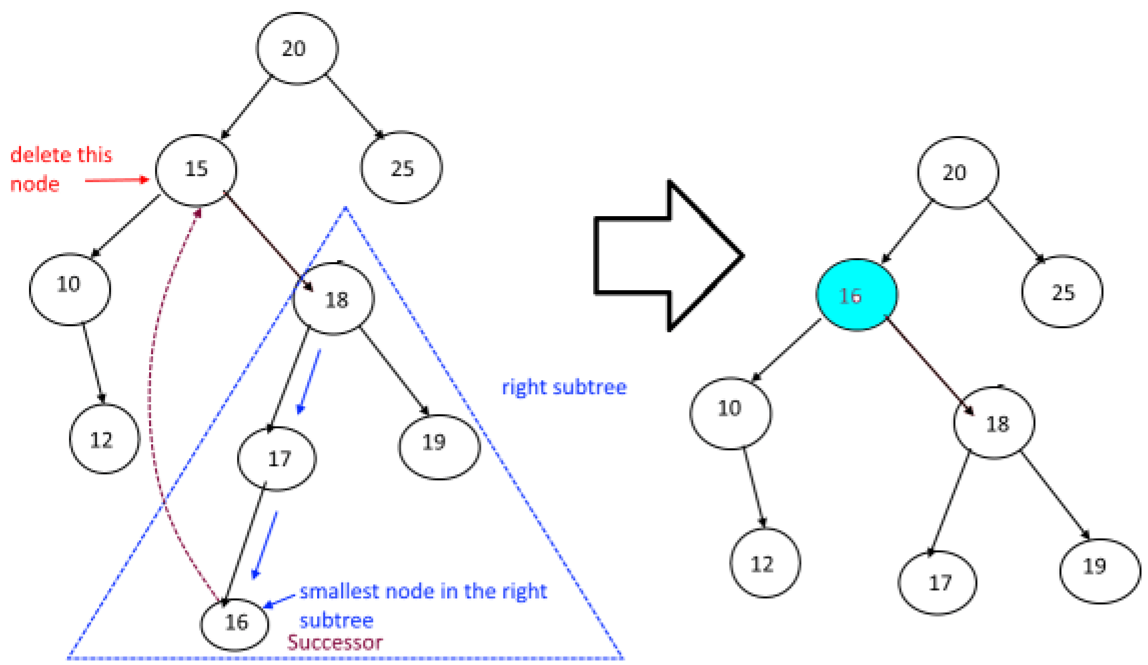
What to do now?????

Dont worry we have solution for this

Find The Successor:

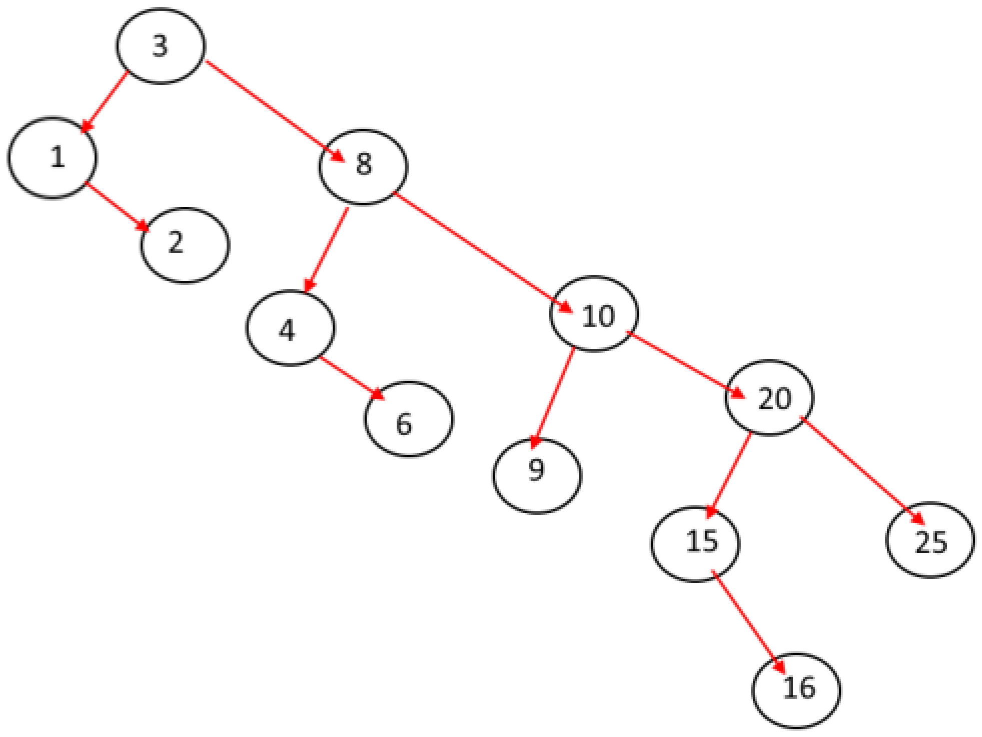
Successor is the node which will replace the deleted node. Now the question is to how to find it and where to find it.

Successor is the smaller node in the right sub tree of the node to be deleted.



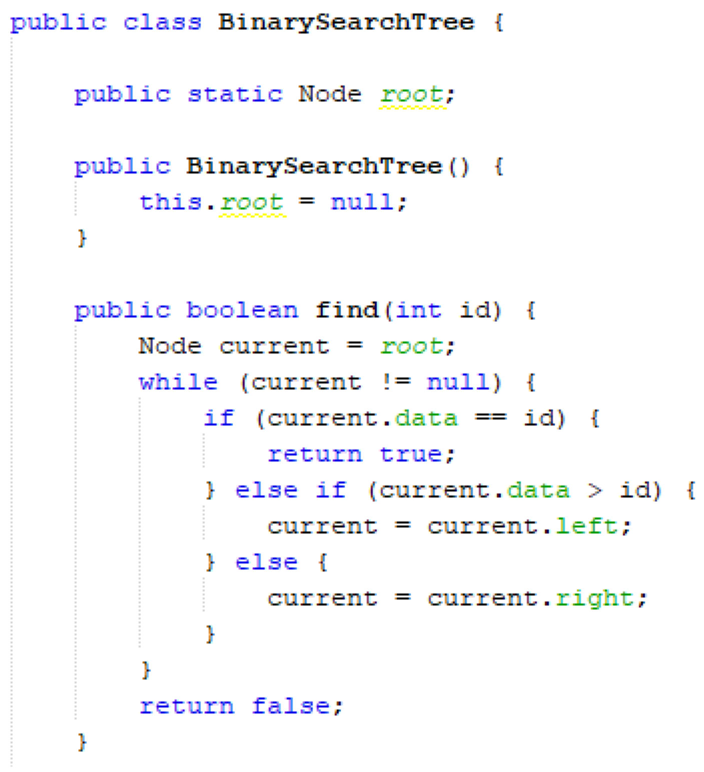
Display() : To know about how we are displaying nodes in increasing order, Click Here

Complete Example :



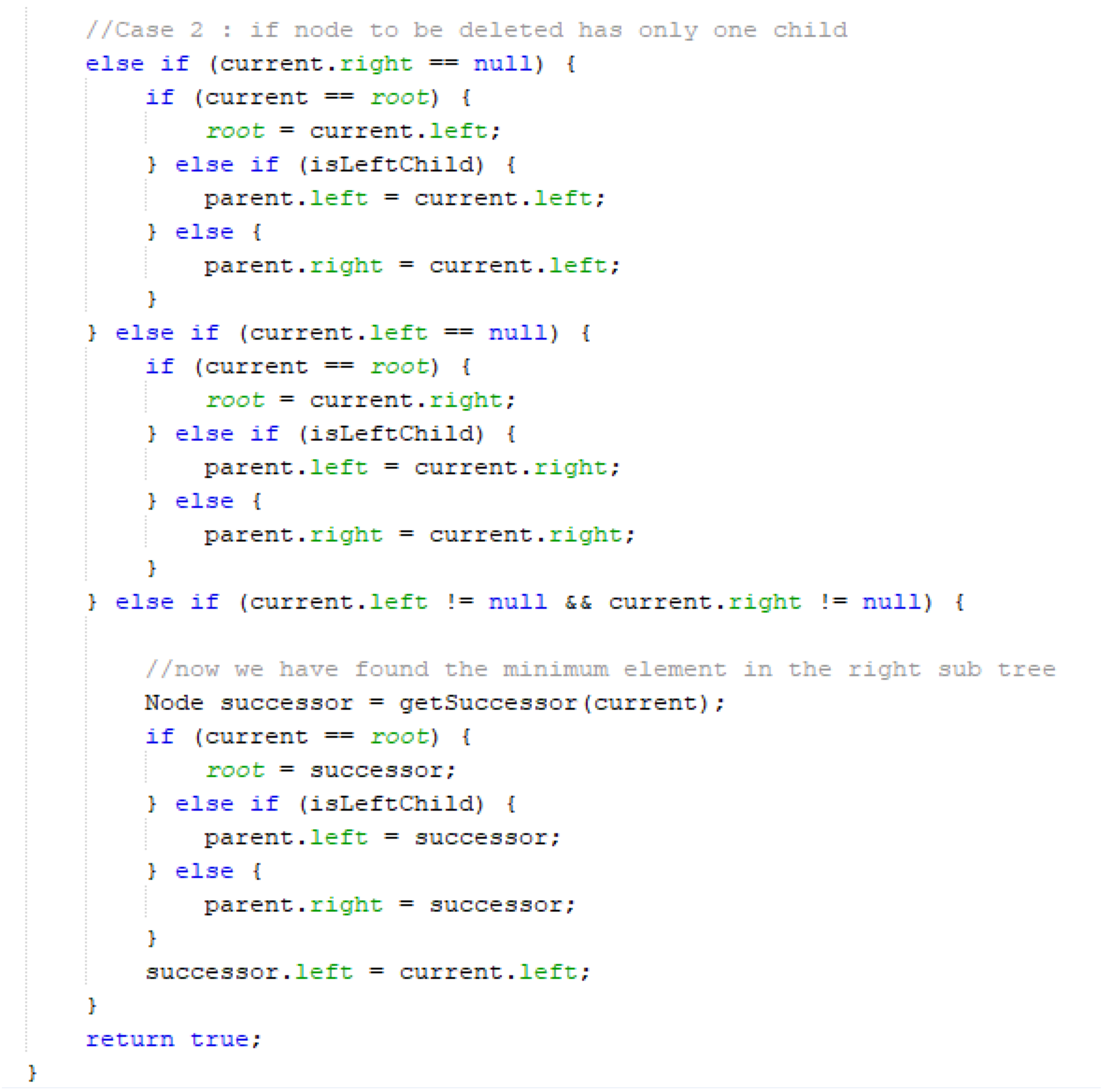
STEPS:

1. After you understand the explanation in the above section, open NetBeans and create a new java application project.
2. Name your project as `BinarySearchTreeExperiment` and click finish.
3. Change author profiles to :
    a. `Name :`
    b. `Program: <put your program. E.g., SMSK(SE) or SMSK with IM`
    c. `Course : CSF3104`
    d. `Lab : <enter lab number>`
    e. `Date : <enter lab date>`
4. In the same `BinarySearchTreeEXperiment` project's package, create a new file named `BinarySearchTree.java`.
5. Add the following codes to the file:
   Note: The codes below are quite long. Take your time to understand the codes and write your own comment where applicable. This will help you to study the codes later.

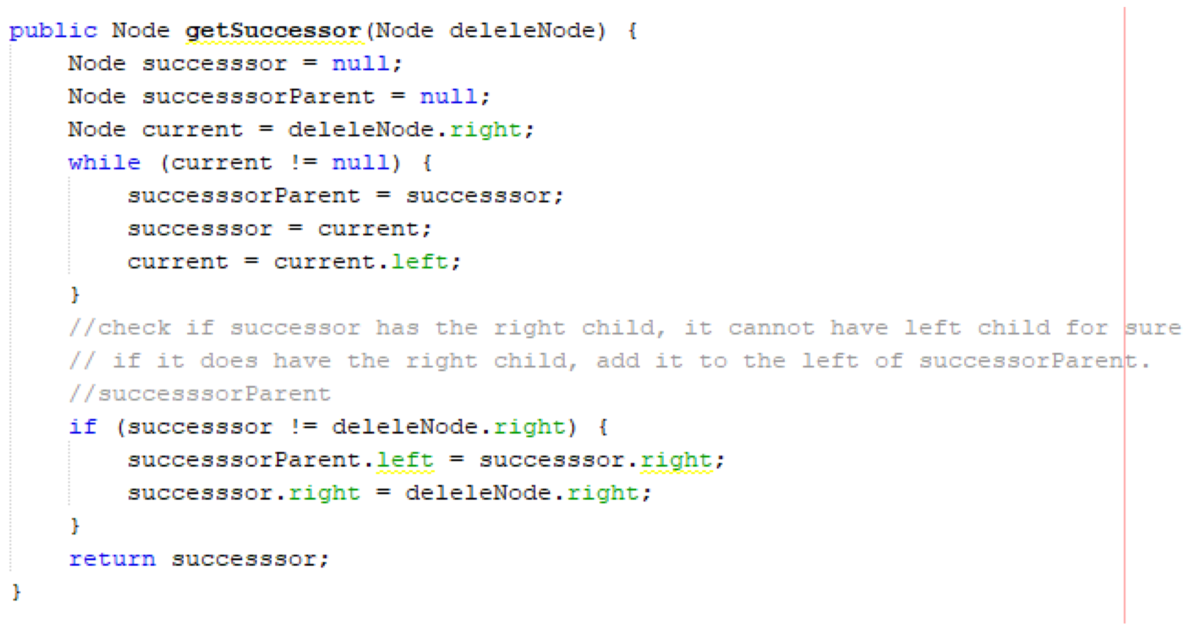


```java
public class BinarySearchTree {

    public static Node root;

    public BinarySearchTree() {
        this.root = null;
    }

    public boolean find(int id) {
        Node current = root;
        while (current != null) {
            if (current.data == id) {
                return true;
            } else if (current.data > id) {
                current = current.left;
            } else {
                current = current.right;
            }
        }
        return false;
    }
```

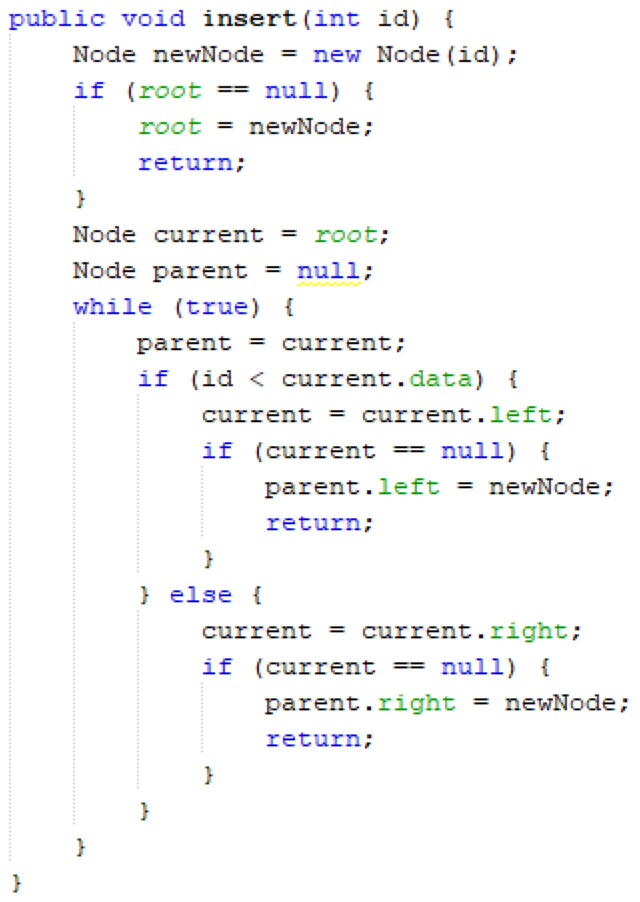
```
public boolean delete(int id) {
    Node parent = root;
    Node current = root;
    boolean isLeftChild = false;
    while (current.data != id) {
        parent = current;
        if (current.data > id) {
            isLeftChild = true;
            current = current.left;
        } else {
            isLeftChild = false;
            current = current.right;
        }
        if (current == null) {
            return false;
        }
    }

    //Case 1: if node to be deleted has no children
    if (current.left == null && current.right == null) {
        if (current == root) {
            root = null;
        }
        if (isLeftChild == true) {
            parent.left = null;
        } else {
            parent.right = null;
        }
    }
```



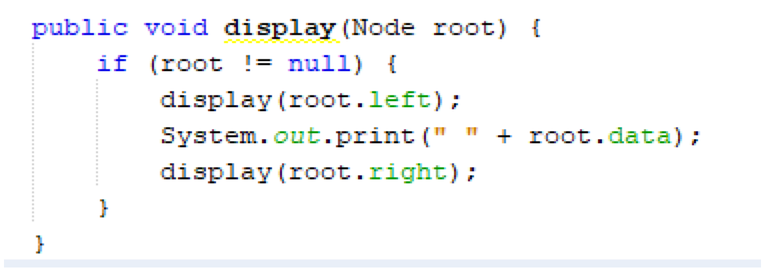
```java
    //Case 2 : if node to be deleted has only one child
    else if (current.right == null) {
        if (current == root) {
            root = current.left;
        } else if (isLeftChild) {
            parent.left = current.left;
        } else {
            parent.right = current.left;
        }
    } else if (current.left == null) {
        if (current == root) {
            root = current.right;
        } else if (isLeftChild) {
            parent.left = current.right;
        } else {
            parent.right = current.right;
        }
    } else if (current.left != null && current.right != null) {

        //now we have found the minimum element in the right sub tree
        Node successor = getSuccessor(current);
        if (current == root) {
            root = successor;
        } else if (isLeftChild) {
            parent.left = successor;
        } else {
            parent.right = successor;
        }
        successor.left = current.left;
    }
    return true;
}
```



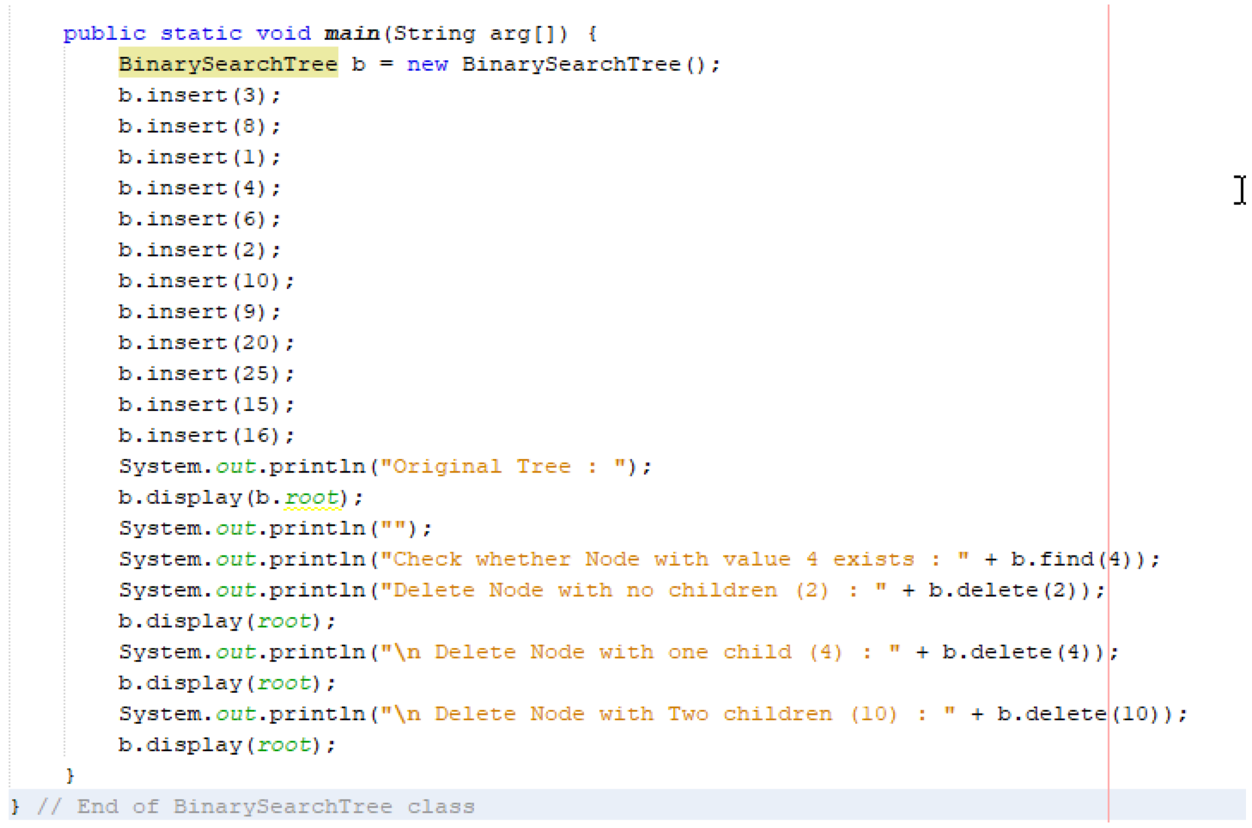
```java
public Node getSuccessor(Node deleleNode) {
    Node successsor = null;
    Node successsorParent = null;
    Node current = deleleNode.right;
    while (current != null) {
        successsorParent = successsor;
        successsor = current;
        current = current.left;
    }
    //check if successor has the right child, it cannot have left child for sure
    // if it does have the right child, add it to the left of successorParent.
    //successsorParent
    if (successsor != deleleNode.right) {
        successsorParent.left = successsor.right;
        successsor.right = deleleNode.right;
    }
    return successsor;
}
```



```java
public void insert(int id) {
    Node newNode = new Node(id);
    if (root == null) {
        root = newNode;
        return;
    }
    Node current = root;
    Node parent = null;
    while (true) {
        parent = current;
        if (id < current.data) {
            current = current.left;
            if (current == null) {
                parent.left = newNode;
                return;
            }
        } else {
            current = current.right;
            if (current == null) {
                parent.right = newNode;
                return;
            }
        }
    }
}
```

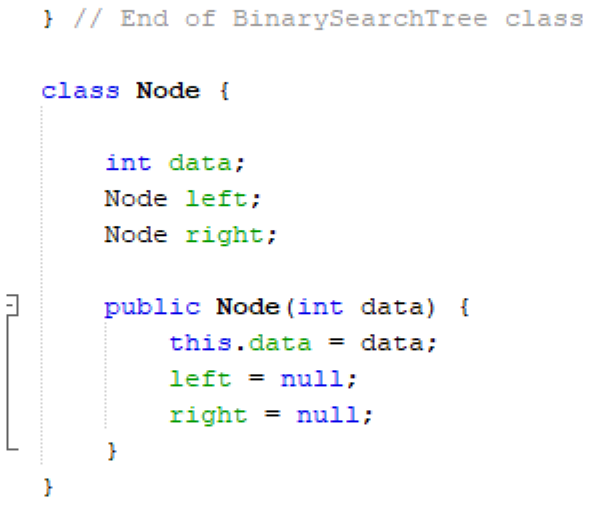


```java
public void display(Node root) {
    if (root != null) {
        display(root.left);
        System.out.print(" " + root.data);
        display(root.right);
    }
}
```



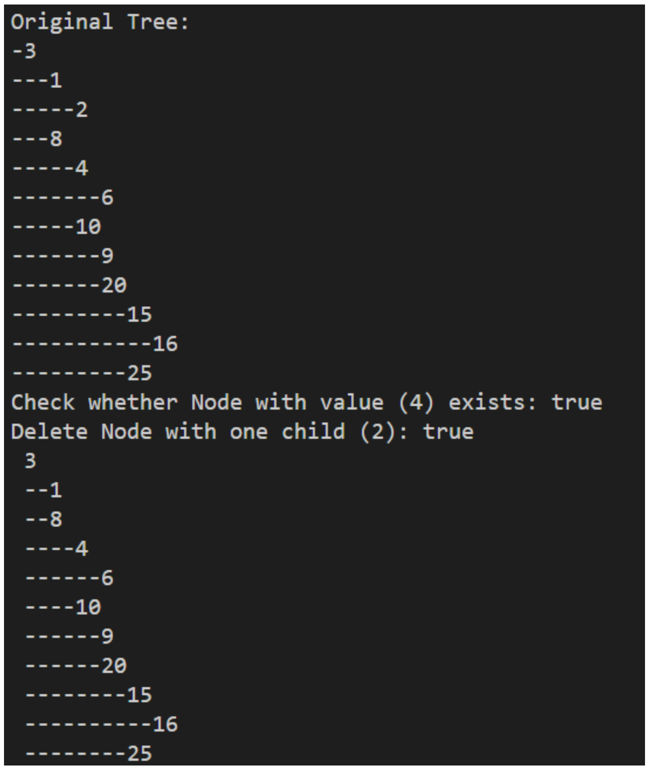
```java
    public static void main(String arg[]) {
        BinarySearchTree b = new BinarySearchTree();
        b.insert(3);
        b.insert(8);
        b.insert(1);
        b.insert(4);
        b.insert(6);
        b.insert(2);
        b.insert(10);
        b.insert(9);
        b.insert(20);
        b.insert(25);
        b.insert(15);
        b.insert(16);
        System.out.println("Original Tree : ");
        b.display(b.root);
        System.out.println("");
        System.out.println("Check whether Node with value 4 exists : " + b.find(4));
        System.out.println("Delete Node with no children (2) : " + b.delete(2));
        b.display(root);
        System.out.println("\n Delete Node with one child (4) : " + b.delete(4));
        b.display(root);
        System.out.println("\n Delete Node with Two children (10) : " + b.delete(10));
        b.display(root);
    }
} // End of BinarySearchTree class
```

6. Just after the end of BinarySearchTree class, add the Node class below it:

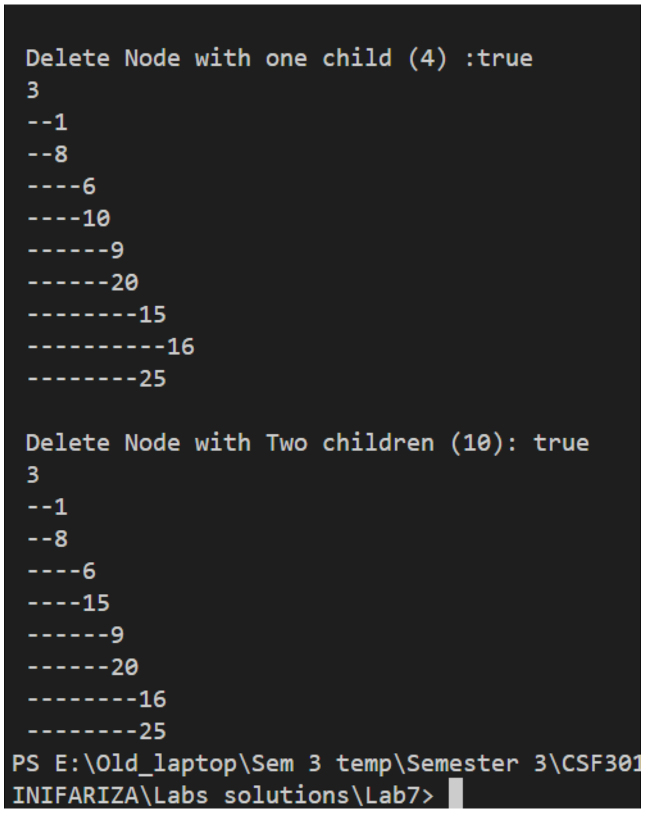


```java
} // End of BinarySearchTree class

class Node {

    int data;
    Node left;
    Node right;

    public Node(int data) {
        this.data = data;
        left = null;
        right = null;
    }
}
```

7. Save, compile and run your codes. Observe the output.
8. Upload your output using the control box below:



```
Original Tree:
-3
---1
-----2
---8
-----4
-------6
-----10
-------9
-------20
---------15
-----------16
---------25
Check whether Node with value (4) exists: true
Delete Node with one child (2): true
 3
 --1
 --8
 ----4
 ------6
 ----10
 ------9
 ------20
 --------15
 ----------16
 --------25
```



```

 Delete Node with one child (4) :true
 3
 --1
 --8
 ----6
 ----10
 ------9
 ------20
 --------15
 ----------16
 --------25

 Delete Node with Two children (10): true
 3
 --1
 --8
 ----6
 ----15
 ------9
 ------20
 --------16
 --------25
PS E:\Old_laptop\Sem 3 temp\Semester 3\CSF301
INIFARIZA\Labs solutions\Lab7>
```

9. Copy and paste your Java codes into the text box below:
   Answer:

   BinarySearchTree

```java
package BinarySearchExpriment;

/*
     * Name: OMAR ISMAIL ABDJALEEL ALOMORY
     * Course: CSF3013
     * Program: SMSK(KMA) K2
     * Lab: MP3
     * Date: 3/1/2023
     */

public class BinarySearchTree{

    public static Node root;

    public BinarySearchTree(){
        this.root = null;
    }

    // finding the data in the tree
    public boolean find(int id){
        Node current = root;
        while(current !=null){
            if(current.data == id){
                return true;
            }else if(current.data > id){
                current = current.left;
            }else{
                current = current.right;
            }
        }
        return false;
    }


    public boolean delete(int id){
        Node parent = root;
        Node current = root;
        boolean isLeftChild = false;
        while(current.data != id){
            parent = current;
            if(current.data > id  ){
                isLeftChild = true;
                current = current.left;
            }else{
                isLeftChild = false;
                current = current.right;
            }
            if(current == null) return false;
        }
        // Case 1: if node to be deleted has no children
        if(current.left == null && current.right == null){
            if(current == root){
                root = null;
            }
            if(isLeftChild == true){
                parent.left = null;
            }else{
                parent.right = null;
            }
        // case 2:if node to be deleted has only one child
        }else if(current.right == null){
            if(current ==  root ){
                root = current.left;
            }else if(isLeftChild){
                parent.left = current.left;
            }else{
                parent.right = current.left;
            }
        }else if (current.left == null){
            if(current == root ){
                root = current.right;
            } else if(isLeftChild){
                parent.left = current.right;
            }else {
                parent.right = current.right;
            }
        }else if(current.left != null && current.right != null){
            // now we have found the minimun element in the rihgt sub tree
            Node successor = getSuccessor (current);
            if(current == root ){
                root = successor;
            }else if(isLeftChild){
                parent.left = successor;
            }
            else{
                parent.right = successor;
            }
            successor.left = current.left;
        }
        return true;
    }

    public Node getSuccessor(Node deleteNode){
        Node successor = null;
        Node successorParent = null;
        Node current = deleteNode.right;
        while(current != null){
            successorParent = successor;
            successor = current;
            current =current.left;
        }
        /*
         * check if successor has the right child, it cannot have left child for
sure
         * if it does have the right child, add it to the left of
successorParent.
         *
         * successorParent
         */
        if(successor != deleteNode.right){
            successorParent.left = successor.right;
            successor.right = deleteNode.right;
        }
        return successor;
    }

    public void insert(int id){
        Node newNode = new Node(id);
        if(root == null){
            root = newNode;
            return;
        }
        Node current = root;
        Node parent = null;
        while(true){
            parent = current;
            if(id< current.data){
                current = current.left;
                if(current == null){
                    parent.left = newNode;
                    return;
                }
            }else{
                current = current.right;
                if(current == null){
                    parent.right = newNode;
                    return;
                }
            }
        }
    }

    public void display(Node root,String gap){

        if(root != null){
            System.out.println(gap+ root.data);
            gap+="--";
            display(root.left,gap);
            display(root.right,gap);
        }
    }



    class Node{
        int data;
        Node left;
        Node right;
        public Node (int data){
            this.data = data;
            left = null;
            right = null;
        }
    }
    public static void main(String[] args) {
        BinarySearchTree b = new BinarySearchTree();
        b.insert(3);
        b.insert(8);
        b.insert(1);
        b.insert(4);
        b.insert(6);
        b.insert(2);
        b.insert(10);
        b.insert(9);
        b.insert(20);
        b.insert(25);
        b.insert(15);
        b.insert(16);

        System.out.println("Original Tree: ");
        b.display(b.root,"-");
        System.out.println("Check whether Node with value (4) exists: 
"+b.find(4));
        System.out.println("Delete Node with one child (2): " + b.delete(2));
        b.display(root, " ");
        System.out.println("\n Delete Node with one child (4) :"+b.delete(4));
        b.display(root, " ");
        System.out.println("\n Delete Node with Two children (10): 
"+b.delete(10));
        b.display(root," ");
    }
}
```

QUESTIONS

1. Discuss the differences between General Tree and Binary Search Tree.
   Answer:
   A general tree is a tree data structure in which each node can have any number of children. There are no restrictions on the number of children that a node can have, and the nodes are not required to be sorted in any particular way. So, it would be diffecult to search for a specific value.

   A binary search tree (BST), on the other hand, is a specific type of tree data structure in which each node has at most two children and the values of the nodes are required to be sorted in a specific way. Specifically, the value of each node in a BST must be greater than the values of all the nodes in its left subtree and less than the values of all the nodes in its right subtree. And thus this can help in searching for a specific value.

2. What are the advantages of BST?
   Answer:

1. **Fast search: BSTs allow for fast search operations. On average, the time complexity for searching for a value in a BST is O(log n), where n is the number of nodes in the tree.**
2. **Fast insertion and deletion: BSTs also allow for fast insertion/deletion of new/old values.**
3. **Sorted data: BSTs store data in a sorted manner, which can be useful in certain applications.**
4. **Flexibility: BSTs are flexible data structures that can be used in a wide variety of contexts.**
5. **Efficient use of memory.**

TASK 2: IMPLEMENTING PRIORITY QUEUE

OBJECTIVE

During this activity, students will learn and apply the concept of priority queue data structure.

ESTIMATED TIME

[60 Minutes]

INTRODUCTION

The priority queue is a somewhat similar data structure to the queue. The difference lies in how the elements are being processed:

- A standard queue strictly follows the FIFO (First-In-Last-Out) principle.
- A priority queue does not follow the FIFO principle.

In a priority queue, the elements are being removed from the queue based on their *priority.* This translates to the requirement that:

Every element in a priority queue must have a priority associated with it.

As you might have guessed, the element with the highest priority is removed from the queue (dequeued).

But how do should you define the priority of the elements in the queue?

Basically, you have two alternatives for defining priorities to the elements in the queue. You can either:

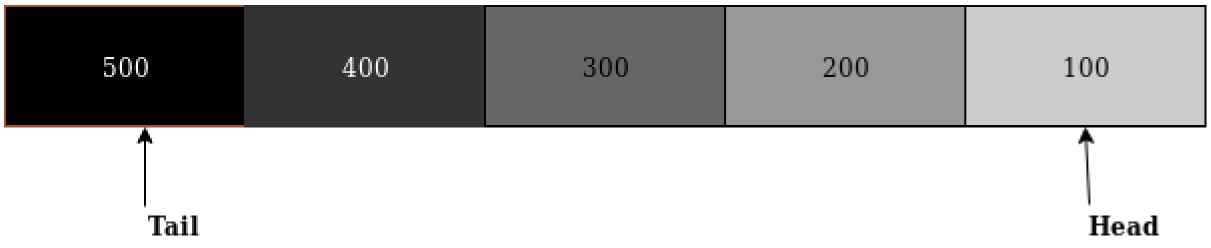
- Order elements based on their natural ordering.
- Order elements with a custom Comparator.

In this article, we'll focus on how to implement a priority queue. So for simplicity's sake, we'll order the elements based on their natural ordering.

In our example, the priority queue will be limited to `int`, so natural ordering is perfectly fine. However, keep in mind that this is for demonstration purposes only.

If you were to implement a priority queue in real life, you probably want to make use of generics — or just do like any other developer and use the built-in java.util.PriorityQueue.

To keep our example implementation compliant with the Java specification, the least element is defined as the element with the highest priority.



PRIORITY QUEUE OPERATIONS

The most basic set of operations for any implementation of a queue is:

`enqueue` — Inserting an element into the queue.

`dequeue` — Removing an element from the queue.

`isEmpty` — Returning true if the queue is empty.

`size` — Returning the size/number of elements in the queue.

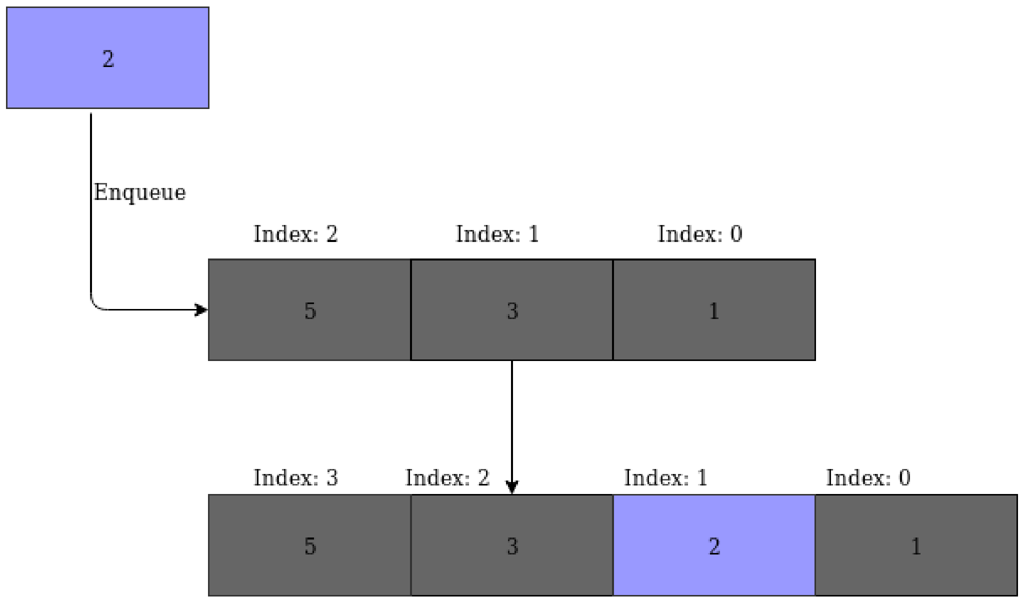
`contains` — Returning true if the queue contains the given element.

`peek` — Returning the front element of the queue, without removing it.

Please note that the Java implementation of the priority queue uses different names for the methods. In a production environment, I highly suggest that you use the default implementation of the priority queue, instead of "home-growing" it.

Implementing Enqueue and Dequeue

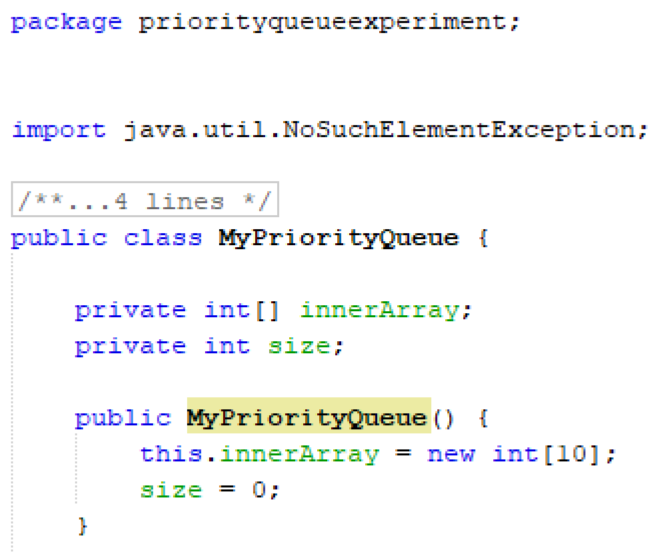
First, let's think about what we want to happen if we insert/enqueue an element. Note that we ignore the doubleArray method for now. We want inserted elements to be placed in the queue, in the correct position, according to its priority.



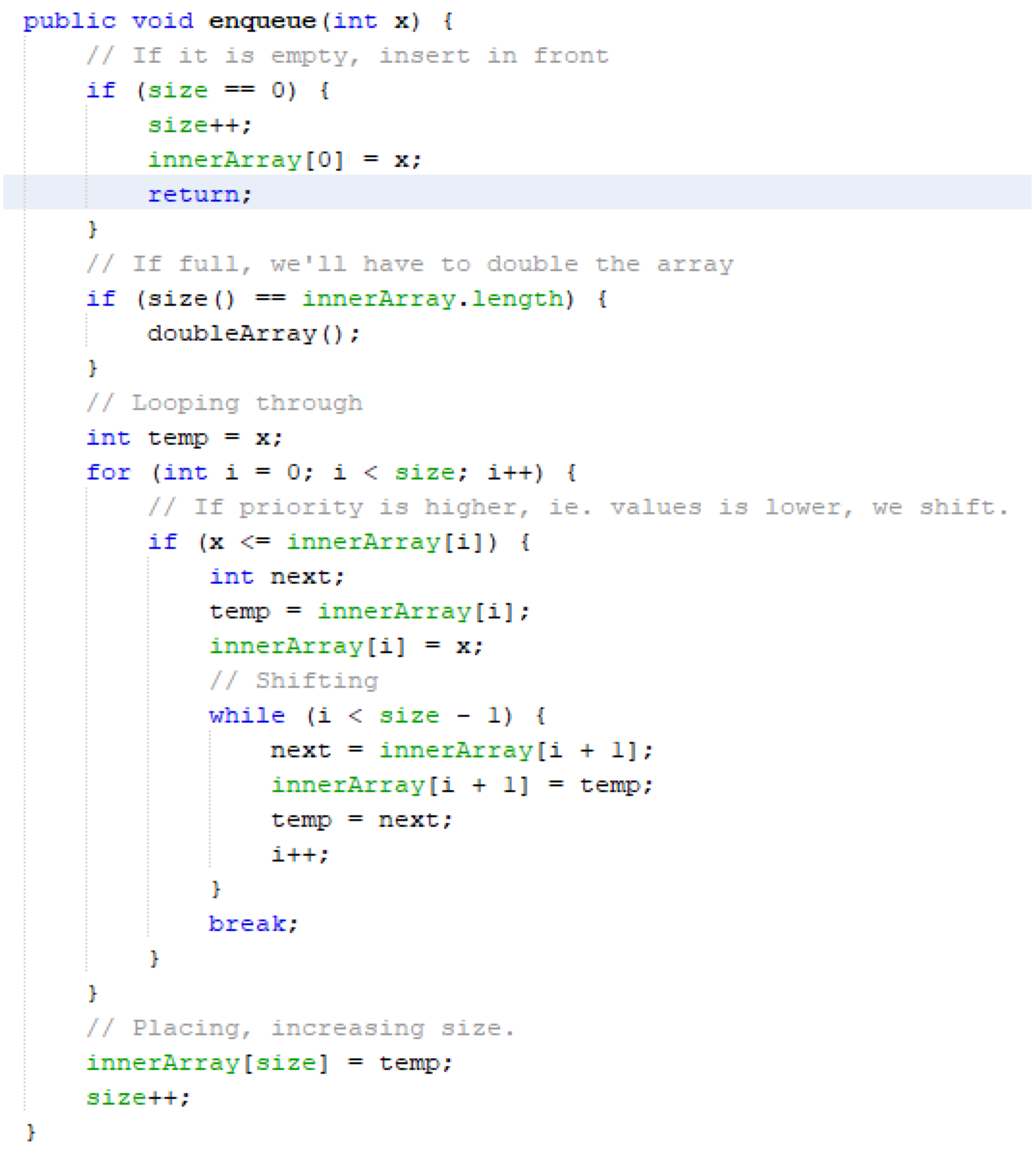
This visualization of the enqueue operation tells us that we'll have to shift all elements of lower priority one position up in the array.

STEPS:

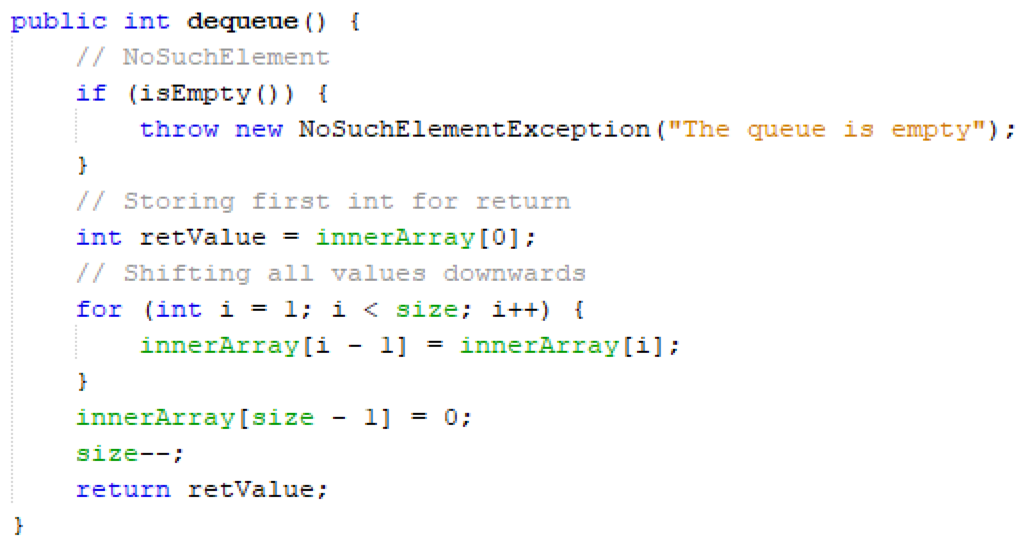
1. Create a new Netbeans project. Name the project as `PriorityQueueExperiment`.
2. Create a new class named `MyPriorityQueue`.
3. In `MyPriorityQueue.java,` insert the following codes, again, try to understand the codes and write your own comment:



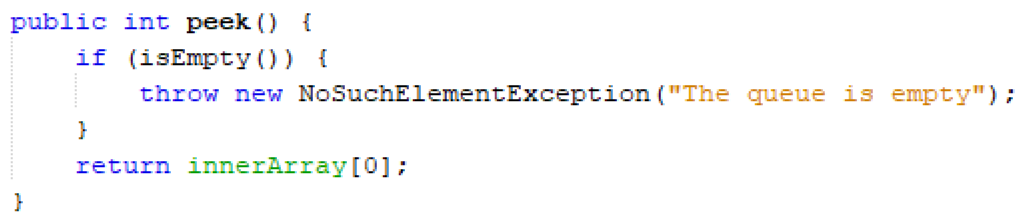
```java
package priorityqueueexperiment;


import java.util.NoSuchElementException;

/**...4 lines */
public class MyPriorityQueue {

    private int[] innerArray;
    private int size;

    public MyPriorityQueue() {
        this.innerArray = new int[10];
        size = 0;
    }
```



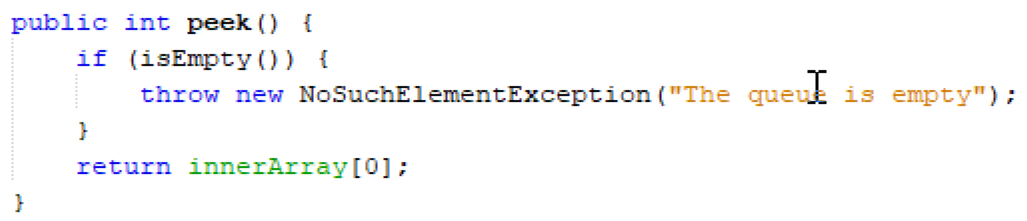
```java
public void enqueue(int x) {
    // If it is empty, insert in front
    if (size == 0) {
        size++;
        innerArray[0] = x;
        return;
    }
    // If full, we'll have to double the array
    if (size() == innerArray.length) {
        doubleArray();
    }
    // Looping through
    int temp = x;
    for (int i = 0; i < size; i++) {
        // If priority is higher, ie. values is lower, we shift.
        if (x <= innerArray[i]) {
            int next;
            temp = innerArray[i];
            innerArray[i] = x;
            // Shifting
            while (i < size - 1) {
                next = innerArray[i + 1];
                innerArray[i + 1] = temp;
                temp = next;
                i++;
            }
            break;
        }
    }
    // Placing, increasing size.
    innerArray[size] = temp;
    size++;
}
```



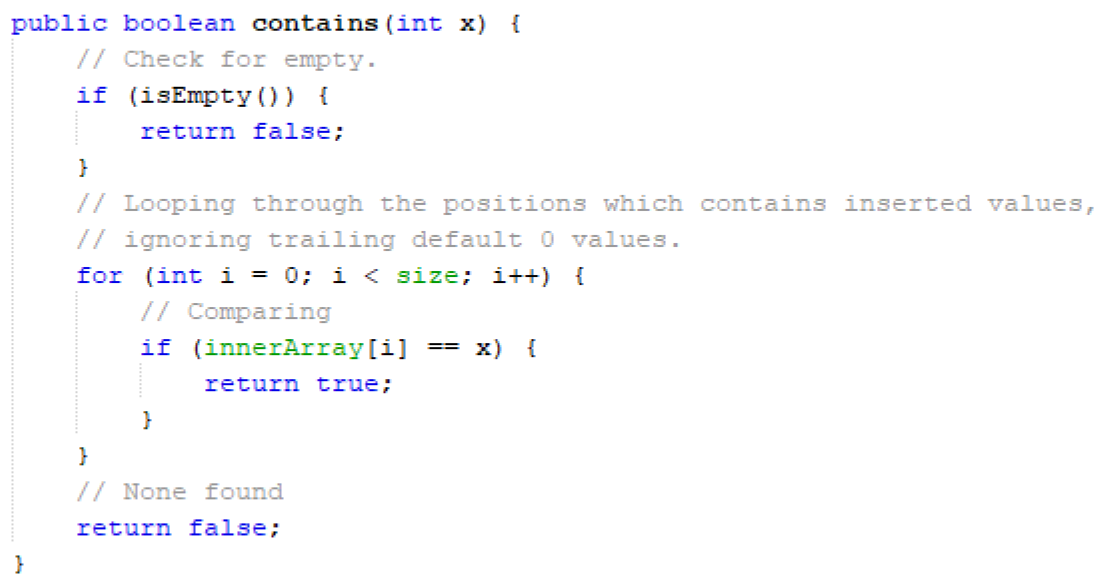
```java
public int dequeue() {
    // NoSuchElement
    if (isEmpty()) {
        throw new NoSuchElementException("The queue is empty");
    }
    // Storing first int for return
    int retValue = innerArray[0];
    // Shifting all values downwards
    for (int i = 1; i < size; i++) {
        innerArray[i - 1] = innerArray[i];
    }
    innerArray[size - 1] = 0;
    size--;
    return retValue;
}
```



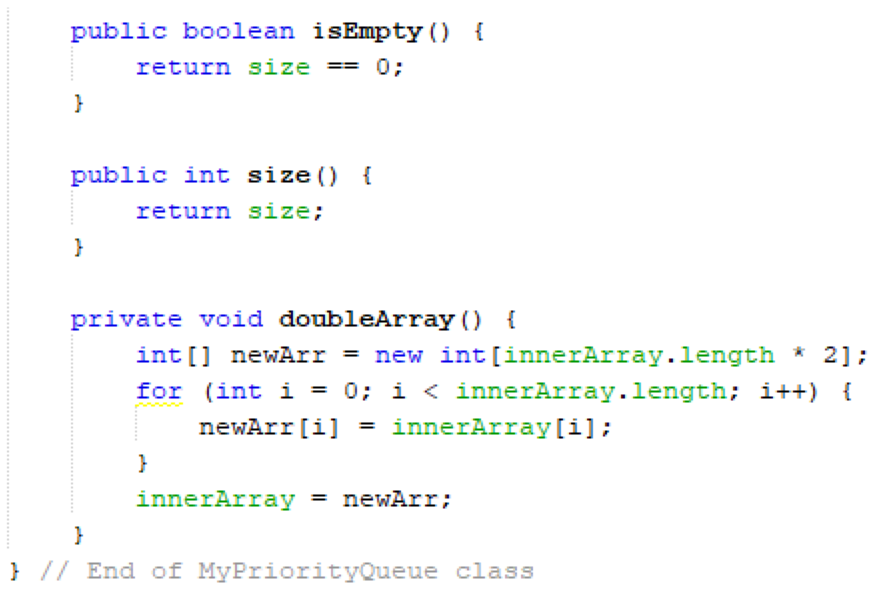
```java
public int peek() {
    if (isEmpty()) {
        throw new NoSuchElementException("The queue is empty");
    }
    return innerArray[0];
}
```



```java
public int peek() {
    if (isEmpty()) {
        throw new NoSuchElementException("The queue is empty");
    }
    return innerArray[0];
}
```

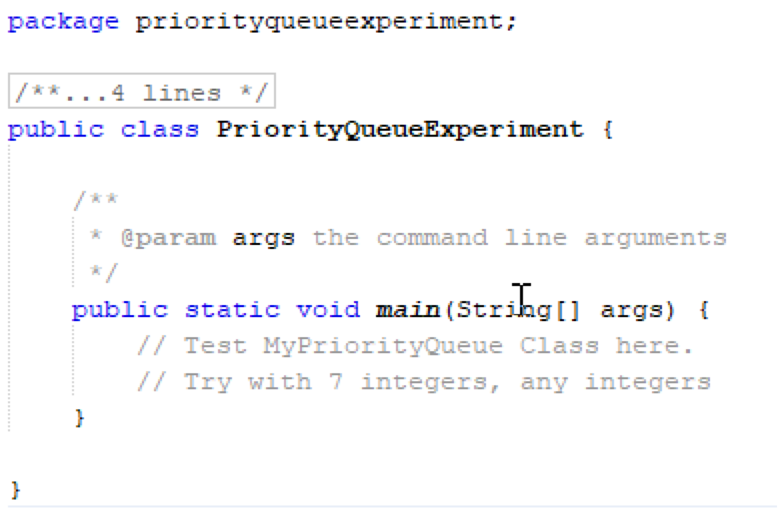


```java
public boolean contains(int x) {
    // Check for empty.
    if (isEmpty()) {
        return false;
    }
    // Looping through the positions which contains inserted values,
    // ignoring trailing default 0 values.
    for (int i = 0; i < size; i++) {
        // Comparing
        if (innerArray[i] == x) {
            return true;
        }
    }
    // None found
    return false;
}
```



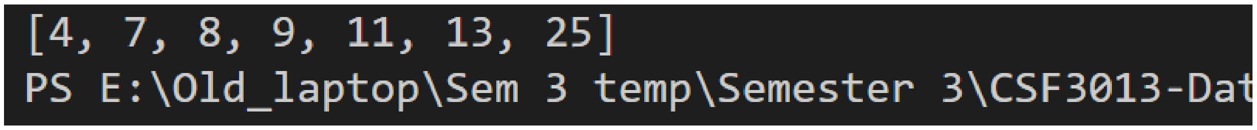
```java
    public boolean isEmpty() {
        return size == 0;
    }

    public int size() {
        return size;
    }

    private void doubleArray() {
        int[] newArr = new int[innerArray.length * 2];
        for (int i = 0; i < innerArray.length; i++) {
            newArr[i] = innerArray[i];
        }
        innerArray = newArr;
    }
} // End of MyPriorityQueue class
```

4. Write a test class by completing the codes section below for MyPriorityQueue class:



```java
package priorityqueueexperiment;

/**...4 lines */
public class PriorityQueueExperiment {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // Test MyPriorityQueue Class here.
        // Try with 7 integers, any integers
    }

}
```

5. Save, compile and execute your codes.
6. Upload your output using the control box below:



```
[4, 7, 8, 9, 11, 13, 25]
PS E:\Old_laptop\Sem 3 temp\Semester 3\CSF3013-Dat
```

7. Copy and paste your Java codes from NetBeans into text box below:

```java
package PriorityQueueExperiment;

/* Name: OMAR ISMAIL ABDJALEEL ALOMORY
 * Program: SMSK (KMA)
 * Course: CSF3013
 * Lab: 7
 * Date: 3/1/2023
 */
import java.util.NoSuchElementException;
public class MyPriorityQueue {
    private int[] innerArray;
    private int size;

    public MyPriorityQueue(){
        this.innerArray = new int[10];
        size = 0;
    }

    public void enqueue(int x){
        //if it is empty,insert in front
        if (size == 0){
            size++;
            innerArray[0] = x;
            return;
        }

        //if full, we'll have to double the array
        if (size() == innerArray.length){
            doubleArray();
        }

        //looping through
        int temp = x;
        int i ;
        for ( i = size-1;i>=0;i--){
            //if priority is higher, ie. values is lower, we shift
            if (x <= innerArray[i]){ // if we want to swap (ex Desc, Asc)
                innerArray[i+1]= innerArray[i];

            }else{
                 break;
            }
        }
        innerArray[i+1] = x;
        size++;
    }

    public int dequeue(){
        //noSuchElement
        if (isEmpty()){
            throw new NoSuchElementException("The queue is empty");
        }
        //Storing first int for return
        int retValue = innerArray[0];
        //shifting all values downwards
        for (int i = 1;i < size; i++){
            innerArray[i-1] = innerArray[i];
        }
        innerArray[size-1] = 0;
        size--;
        return retValue;
    }

    public int peek(){
        if (isEmpty()){
            throw new NoSuchElementException("The queue is empty");
        }
        return innerArray[0];
    }

    public boolean contains(int x){
        //check for empty
        if (isEmpty()){
            return false;
        }
        //looping through the positions which contains inserted values,
        //ignoring trailing default 0 values
        for (int i = 0;i < size;i++){
            //comparing
            if (innerArray[i] == x){
                return true;
            }
        }
        //none found
        return false;
    }

    public boolean isEmpty(){
        return size == 0;
    }

    public int size(){
        return size;
    }

    public void doubleArray(){
        int[] newArr = new int[innerArray.length * 2];
        for (int i = 0;i < innerArray.length;i++){
            newArr[i] = innerArray[i];
        }innerArray = newArr;
    }
    public String display(){
        // String for just fomatting to look like Array or something like that
        String format = "[";
        while(!isEmpty()){
            if(size == 1){
               format +=dequeue();
            }else{
                format +=dequeue()+", ";
            }

        }
        return format+"]";
    }
      // Test class (Main)
    public static void main(String[] args) {
        MyPriorityQueue p = new MyPriorityQueue();
        // 7 integers
        p.enqueue(7);// first one entered
        p.enqueue(11);
        p.enqueue(8);
        p.enqueue(25);
        p.enqueue(13);
        p.enqueue(9);
        p.enqueue(4);// last one entered
        System.out.println(p.display());
        // as the output show the queue sorted
    }
}
```

QUESTIONS

1. List the differences between common Queue and Priority Queue.

Answer:

A queue is a data structure that stores items in a first-in, first-out (FIFO) manner. A priority queue is a queue in which each item has a priority associated with it, and the queue is always sorted so that the highest priority items are at the front of the queue.

1. **Insertion order: In a common queue, the order in which items are added to the queue determines the order in which they are removed. In a priority queue, the order in which items are added to the queue does not necessarily determine the order in which they are removed, because the queue is sorted by priority.**

2. **Deletion order: In a common queue, items are always removed from the front of the queue. In a priority queue, items are removed from the front of the queue, but the front of the queue may change as items with higher priorities are added or removed.**

3. **Sorting: A common queue does not sort its items. A priority queue sorts its items by priority, so that the highest priority items are always at the front of the queue.**

4. **Time complexity: The time complexity for inserting and deleting elements in a common queue is typically O(1). The time complexity for inserting and deleting elements in a priority queue is typically O(log n), where n is the number of items in the queue**

2. In Java, there is a built-in class for PriorityQueue, how can you use it?

Answer:

By importing it 'import java.util.PriorityQueue;'

Create instance 'PriorityQueue<Integer> pq = new PriorityQueue<>();'

Adding element by using offer or add :

```
pq.add(10);
    pq.add(3);
    pq.add(5);

    pq.add(1);


// Print the elements of the queue in ascending order

    while (!pq.isEmpty()) {

      System.out.println(pq.remove());

    }
```

Output

```
1

3

5

10
```

You can also specify a custom comparator to be used for ordering the elements in the queue. For example, if you want to create a priority queue of strings that orders the elements in descending order of length:

```
Comparator<String> comparator = new Comparator<String>() {

   public int compare(String s1, String s2) {

      return s2.length() - s1.length();

   }

};

PriorityQueue<String> pq = new PriorityQueue<>(comparator);
```

3. What are the ADTs for PriorityQueue?

Answer:

insert(elem)/enqueue(): Inserts an element into the priority queue.

delete(elem)/dequeue(): Removes an element from the priority queue.

remove()/poll(): Removes and returns the highest priority element from the priority queue.

peak() Returns the highest priority element from the priority queue, without removing it.

isEmpty(): Returns true if the priority queue is empty, false otherwise.

Finally, read the instruction regarding submission carefully. Submit your answer using the link provided in Oceania UMT. Please ensure your codes are submitted to the correct group.