OL Academy

Second Semester, 2025-2026
ITCS214 (Data Structures)

Final Exam Revision

Part1: Single Linked List & Double Linked List

Question 1

Write a method called **getData** of the generic class **SingleLinkedList** that accepts one parameter **loc** of type **int**. The method will return the data at index **loc**.If the list is empty or the parameter **loc** is *invalid* throw **IndexOutOfBoundsException**.

Do not call any method of class Single LinkedList in your method.

Method heading: public E getData (int loc)

Question 2

Write a method called **deleteAlternate** to be included in the class **SingleLinkedList** that accepts a parameter list1 of type **SingleLinkedList**. If "this" list is empty, it will return false, otherwise it will remove alternative nodes from "this" list and copy them to list1 and return true. *Assume that list1 is initially empty.*

Method heading: public void **reverseNodes ()**

Do not call any method of class SingleLinkedList in your method.

Example

Before method call: "this" list: 5 4 7 3 2 6
list1: (empty)

After method call: "this" list: 4 3 6
list1: 5 7 2

```
public  Boolean deleteAlternate ( SingleLinkedList <E> list1)
{
```

Question 3

Write a method called **findLargest** to be included in class **KWLinkedList** that finds all nodes in ***(this)*** list with their data values greater than **item**, passed as the first parameter and inserts these values in a new linked list, called **list2** passed as the second parameter to the method. If ***(this)*** list is empty or no values greater than **item** exists in the list, then the method returns false, otherwise it returns true. Assume that list2 is initially empty.

Write your method by using **ListIterator**. ***Do not call any other method of class KWLinkedList in your method.***

Method heading: public boolean **findLargest (E item, KWLinkedList<E> list2)**

Example: **Item: 8**

Before method call

(this) list: 9 8 25 10 4 7 40 2 11
list2: (empty)

After method call

(this) list: 9 8 25 10 4 7 40 2 11
list2: 9 25 10 40 11

Question 4

Write a method called **reverseNodes** to be included in the **KWLinkedList** class which reverses the nodes of a doubly linked list, Write the method by swapping the data fields of the nodes, if the list is empty or contains only one node display this message (List is Empty or has one node only), Assume that the list contains even number of elements.

Do not use Iterators and you are not allowed to call any of the KWLinkedList class methods.

Method heading: public void **reverseNodes ()**

Example

Before method call

(this) list: 15 37 55 88 10 59

After method call

(this) list: 59 10 88 55 37 15

```
public  void reverseNodes ()
{
```

Part2: Stacks & Queues

Question 1 : What is the output of the following program?

```
ArrayStack<Integer> st1=new ArrayStack<Integer>();
ArrayStack<Integer> st2=new ArrayStack <Integer>();
for(int i = 1; i <= 6; i++)
        st1.push(i);
while(!st1.isEmpty())
{
        int item = st1.pop();
        if(item % 2 == 0)
                st2.push(item);
}
while(!st2.isEmpty())
        st1.push(st2.pop());

while(!st1.isEmpty())
        System.out.print(st1.pop() + "  ");
```

Question 2

Write a method called **deleteAboveKey** in a class called **`StackEx`** that accepts an object **st1** of type **ArrayStack** as the first parameter and **key** of type **E** as the second parameter. The method will delete all elements from the stack st1 which are greater than or equal to the key, *starting from the first occurrence of the key*. All the remaining elements of **st1** should be in the original relative order. If stack **st1** is empty or key is not found in st1, then do not delete any element from **st1** and return false, otherwise return true.

Example: key=9

Stack **st1** before method call

14	2	15	**9**	32	5	8	40
Top							

Stack **st1** after method call

14	2	15	**9**	5	8

Assume that class ArrayStack is available for use. Use common stack operations only such as push, pop, peek, isEmpty and copy constructor.

Public class `StackEx{`

Question 3:

Consider the following postfix expression. Use stack to evaluate it and show all the push and pop operations by clearly drawing the stack status.

12 24 6 / + 5 3 - 10 * -

Question 4 : What would be the contents of q1 after applying the following operations?

```
ArrayQueue<Integer> q1 = new ArrayQueue<Integer>(5);
for (int i = 1; i < 5; i++)
   q1.offer(i+1);

Iterator<Integer> iter = q1.iterator();
int a,b;
while (iter.hasNext())
{
   a = iter.next();
   if (iter.hasNext())
   {
      b = iter.next();
      if (a > b)
         q1.poll();
      else
         q1.offer(b);
   }
}
```

Question 5:

Write a method **SwapQueueHalfs** to be included in class **`ArrayQueue`**. The method swaps the first half of the elements of the queue with the second half of the queue elements. Assume that the queue contains even number of elements.

Method heading: **public void SwapQueueHalfs ()**

Do not call any other method of the class **ArrayQueue** in your method.

Example :

Before method call:					
24	56	78	90	6	44
front					rear

After method call:					
90	6	44	24	56	78
front					rear

public void SwapQueueHalfs ()

{

Question 6:

Write a method called **`countSimilar`** to be included in class **`ArrayQueue`** that accepts a parameter **q2** of type **ArrayQueue**.

The method compares the corresponding elements of **q2** and **"this"** queue and counts those elements which are same. You are not allowed to use any of the methods of **ArrayQueue** class or arrays, Assume "this" and q2 are of the same size.

Example :

"this" :	**10**	12	**30**	**5**	17	16
q2:	**10**	15	**30**	**5**	25	12

`The method will return 3.`

public int countSimilar(ArrayQueue<E> q2)

Part3: Trees, Graphs & Hash Tables

Question (1):

(A) Write a recursive private method called **CountLeftChild** to be included in class **BinaryTree** as discussed in the lectures. The method returns the number of nodes in a binary tree having only left child node.

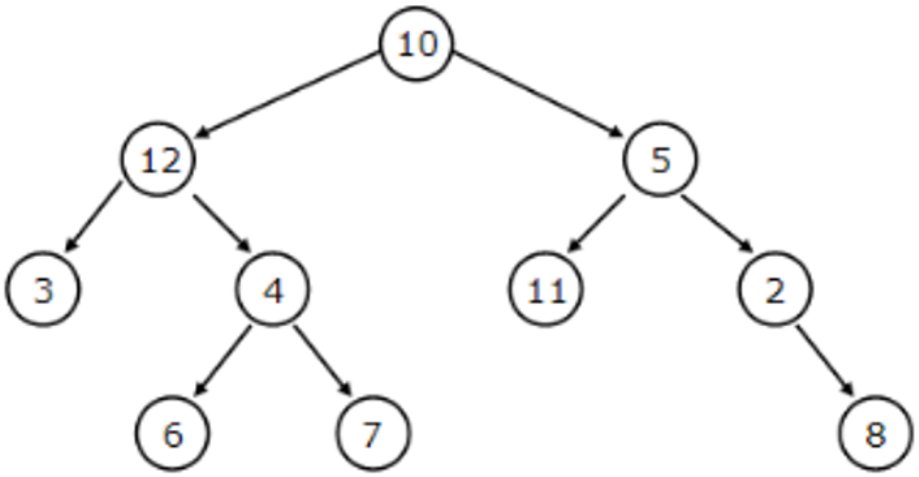
This method is called from a public method treeCountLeftChild, given as follows:

```
public   int   treeCountLeftChild ( )
{
        return   CountLeftChild (root );
}
```

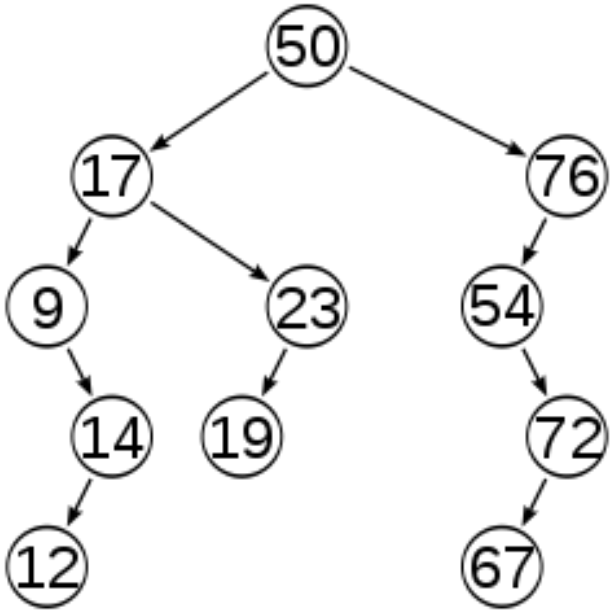
Method heading:

private int **CountLeftChild** (Node<E> node)

(B) Find the sequence of nodes, if the binary tree is traversed in **preOrder and inOrder** traversal.



(C) Consider the following binary search tree:



(i) List all the leaf nodes of this binary tree.

(ii) Find the sequence of nodes, if the binary tree is traversed in **postorder** traversal.

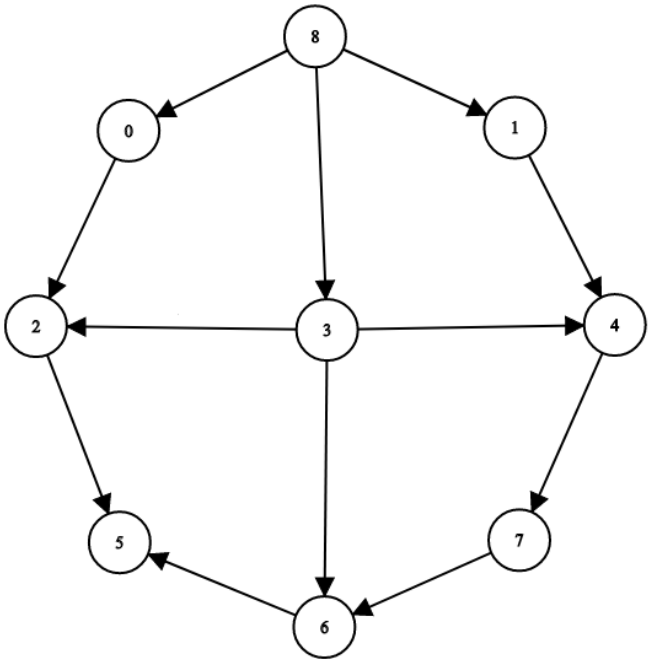
(iii) Write the sequence of nodes to be compared to search the **key = 67** in the above binary search tree.

(iv) Redraw the above binary search tree after inserting the nodes with keys **10** and **53** consecutively, in the original binary search tree.

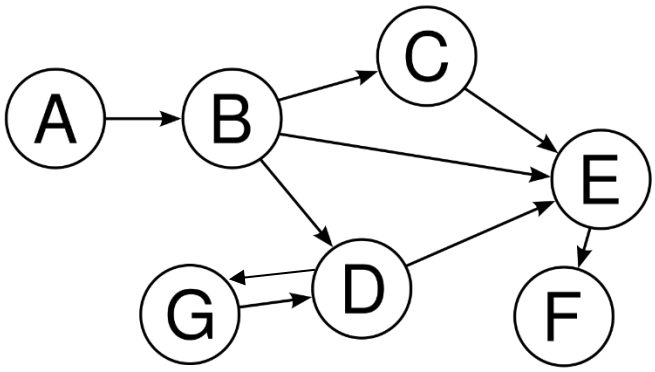
(v) Redraw the above binary search tree after deleting the nodes with keys **54** and **17** consecutively, from the original binary search tree.

Question (2):

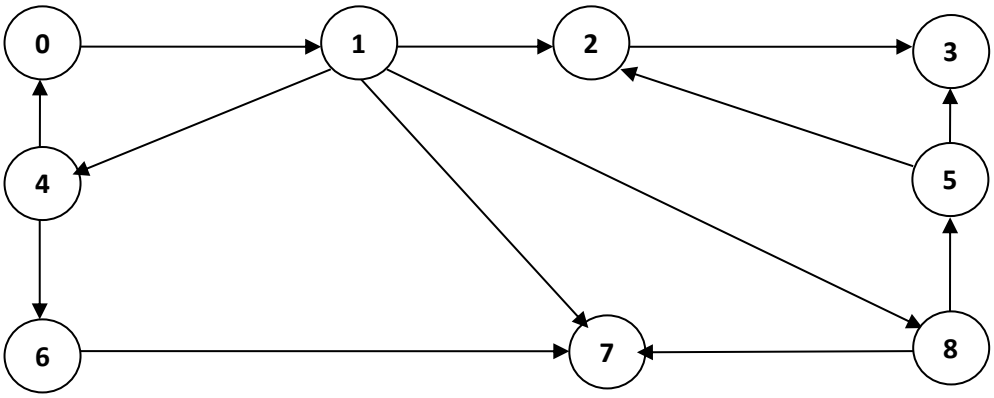
(A) For the following graph, find the adjacency matrix and adjacency List representation of the graph:



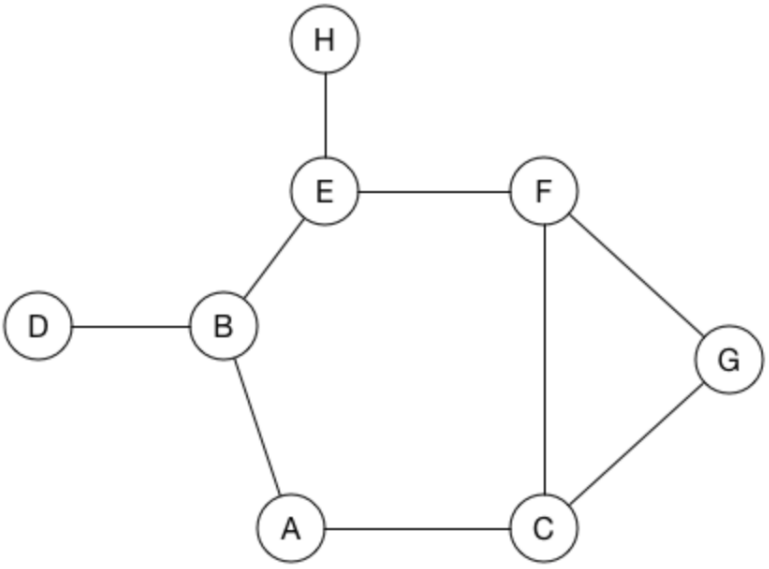
(B) For the following graph, find the sequence of vertices in the graph, if the graph is traversed using **Breadth-First Traversal** algorithm. Assume vertex **A** as the starting vertex.



(C) For the following graph, find the sequence of vertices in the graph, if the graph is traversed using **Depth-First Traversal** algorithm. Assume vertex 0 as the starting vertex.



(D) For the following graph, find the sequence of vertices in the graph, if the graph is traversed using **Depth-First Traversal** & **Breadth-First Traversal** algorithm. Assume vertex **A** as the starting vertex



Question (3): Given the following input keys: 24, 23, 30, 108, 35, 42, 54,
hash function h(X) = X mod 12, HTSize = 12 and bucket size = 1.
Obtain the resulting hash table when open addressing with **linear probing** is used to resolve collisions.

Question (4):

Given the following input keys: **512, 72, 45, 88, 106, 251, 99, 25, 51**

hash function **h(X) = X mod 9**, HTSize = 9 and bucket size = 1.

(A) Obtain the resulting hash table when open addressing with quadratic probing is used to resolve collisions.

(B) Obtain the resulting hash table when chaining is used to resolve collisions.

Question (5):

Part1: What is the state of hash table after the insertion of the following values:

```
HashTable<Integer> htList= new HashTable(7);
htList.insertKeyLinear(49);
htList.insertKeyLinear(14);
htList.insertKeyQuadratic(42);
htList.insertKeyQuadratic(28);
htList.insertKeyLinear(7);
htList.insertKeyLinear(56);
```

Part2: Implement the **insertKeyQuadratic** method using in part 1.