1. You have learned some fundamental data structure concepts such as array, queue and priority queue, stack, list and linked list, sequence, and unordered set, and you understand the concept of interface or abstract data type that defines the set of operations supported by a data structure and the semantics, or meaning, of those operations. You can use the interface of one particular data structure to define or implement the operations of a different data structure.
   1. Describe the meaning of the essential methods add(x), deleteMin(), and size() that are supported by the priority queue interface. Implement those methods using a singly-linked list. Analyze the running time of the add(x) and deletMin() operations based on this implementation.

**Solution:**

**add(x):** This adds a new value in the queue in its appropriate place according to the priority. In my implementation, I simple add the item/value in my linkedList ( because priority is handled by remove() method ) so running time is O(1). However, we can implement this by adding sorted element, in the case, complexity will be O(n).

**deleteMin():** This deletes the minimum value/item from the queue, usually the item with highest priority and then returns the deleted value. In my implementation, complexity is O(n) since we iterate through entire list. However, if items are added already sorted, complexity would be O(1).

**size():** This returns the size of the queue, i.e. the number of items present in the queue. Complexity is O(1).

**Code:**

import java.util.LinkedList;

public class MyPriorityQueue<B extends Comparable<B>> {

private LinkedList<B> linkedList;

public MyPriorityQueue()

{

linkedList = new LinkedList<>();

}

public B deleteMin() // returns the min value being deleted, similar to poll() method

{

if(linkedList.size() == 0)

{

return null;

}

else

{

//start from zero, iterate through the entire list to find the minimum value

int minimum = 0;

for(int i=0; i<linkedList.size(); i++)

{

if(linkedList.get(minimum).compareTo(linkedList.get(i)) > 0) // if current index value is smaller than min index, change min

{

minimum = i;

}

}

B data = linkedList.get(minimum); // data to return

// remove element

linkedList.remove(minimum);

return data;

}

}

// Here we are not adding according to priority since our remove method gives the min , i.e highest priority element

public void add(B data)

{

if(data == null)

{

return;

}

linkedList.add(data);

// System.out.println("ll = " + linkedList.toString());

}

public int size()

{

return linkedList.size();

}

}

* 1. Implement the stack methods push(x) and pop() using two queues. Analyze the running time of the push(x) and pop() operations based on this implementation.

# **Solution:**

import java.util.LinkedList;

import java.util.Queue;

public class StackUsingQueue

{

private Queue<Integer> primaryQueue = new LinkedList<Integer>();

private Queue<Integer> secondaryQueue = new LinkedList<Integer>();

public StackUsingQueue()

{

Queue<Integer> primaryQueue = new LinkedList<Integer>();

Queue<Integer> secondaryQueue = new LinkedList<Integer>();

}

public void push(int data)

{

if(primaryQueue.size() == 0 || primaryQueue.peek() == null)

{

// System.out.println("Adding " + data + " in PQ");

primaryQueue.add(data);

}

else

{

for (int i = primaryQueue.size(); i > 0; i--)

{

int temp = primaryQueue.remove();

// System.out.println("Adding " + temp + " in SQ");

secondaryQueue.add(temp);

}

primaryQueue.add(data);

// System.out.println("Adding " + data + " in PQ");

for (int j = secondaryQueue.size(); j > 0; j--)

{

int temp = secondaryQueue.remove();

// System.out.println("Adding " + temp + " in PQ");

primaryQueue.add(temp);

}

}

}

public Integer pop()

{

if(primaryQueue.size() < 1)

{

System.out.println("Nothing to pop");

return null;

}

else

{

int dataToReturn = primaryQueue.remove();

System.out.println(dataToReturn);

System.out.println("\nNow stack is : "+ primaryQueue + "\n");

return dataToReturn;

}

}

public void printStack()

{

System.out.println(primaryQueue);

}

}

|  |  |
| --- | --- |
| **Method** | **Complexity** |
| push() | O(n) |
| pop() | O(1) |

1. Swap two adjacent elements in a list by adjusting only the links (and not the data) using
   1. singly-linked list

**SOLUTION:**  
public void swapAdjacent(int firstIndex)

{

if (numberOfNodes <= 1)

{

System.out.println("There is only one element in the list");

}

else if (firstIndex >= getNumberOfNodes() - 1)

{

System.out.println("Index is greater than size");

}

else

{

Node currentNode = tail;

Node previousNode = new Node(-1);

previousNode.setNext(tail);

for (int i = 0; i < firstIndex; i++)

{

previousNode = currentNode;

currentNode = currentNode.getNext();

}

previousNode.setNext(previousNode.getNext().getNext());

currentNode.setNext(currentNode.getNext().getNext());

previousNode.getNext().setNext(currentNode);

}

}

* 1. doubly-linked list.  
     **SOLUTION:**

public void swapAdjacent(int firstIndex)

{

if (firstIndex >= numberOfNodes - 1)

{

System.out.println("Index is greater than size");

}

else

{

Node currentNode = tail;

Node previousNode = new Node(-1);

Node nextNode = tail.getNext();

previousNode.setNext(tail);

for (int i = 0; i < firstIndex; i++)

{

previousNode = currentNode;

currentNode = currentNode.getNext();

nextNode = nextNode.getNext();

}

previousNode.setNext(previousNode.getNext().getNext());

currentNode.setNext(currentNode.getNext().getNext());

previousNode.getNext().setNext(currentNode);

nextNode.setPrevious(previousNode);

currentNode.setPrevious(nextNode);

currentNode.getNext().setPrevious(currentNode);

}

}

1. (left by option)
2. (left by option)
3. Write a method, reverse(), that reverses the order of elements in a DLList.

# **Solution:**

public void reverse()

{

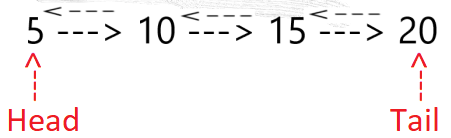
Node temp = head;

head = tail;

tail = temp;

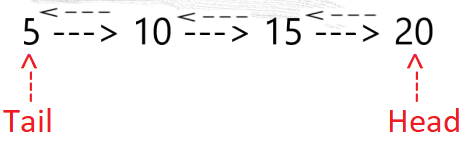
}  
**Explanation:**

By switching the head and tail references, we can achieve the desired goal, i.e. reversing the order of elements in DLL. Consider the following example of a DLL:



Iteration from Head to Tail would give: 5, 10, 15, 20.

Now consider the following DLL after reverse function:



Iteration from Head to Tail would give: 20, 15, 10, 5.

1. Design and implement a MinStack data structure that can store comparable elements and supports the stack operations push(x), pop(), and size(), as well as the min() operation, which returns the minimum value currently stored in the data structure. All operations should run in constant time.

# **Solution:**

We use 2 normal stacks for this implementation. One stack is used to keep track of the items inserted and the other for the min value in the first stack.

# **Code:**

import java.util.Stack;

public class MyMinStack<B extends Comparable<B>>

{

private Stack<B> s1;

private Stack<B> s2;

public MyMinStack()

{

s1 = new Stack<>();

s2 = new Stack<>();

}

public void push(B x)

{

s1.push(x);

if (s2.isEmpty())

{

s2.push(x);

}

else

{

// push the Math.min(s2.peek(), x)

if (x.compareTo(s2.peek()) == 0 || x.compareTo(s2.peek()) == -1)

{

s2.push(x);

}

else

{

s2.push(s2.peek());

}

}

}

public void pop()

{

s1.pop();

s2.pop();

}

public int size()

{

return s1.size();

}

public B min()

{

return s2.peek();

}

}