Q1.

```
public class ReverseString {
   public static String reverse(String st){
      StringBuilder sb = new StringBuilder();
      StringBuilder result = new StringBuilder();
      int length = st.length();
      for(int i=0;i<length;i++){
         char ch = st.charAt(length -1 - i);
         if(ch == 32){
            result.insert(0, sb + " ");
            sb.setLength(0);
            continue;
         }
         sb.append(ch);
      }
      result.insert(0, sb + " ");

      return result.toString();
   }

   public static void main(String[] args){
      System.out.println(reverse("we test coders"));
   }
}
```

Running Time Calculation: O(n); where n is the length of the string.

Work	Time Complexity
creating string builder, calculating length, inserting and converting to string from string builder	O(c) + O(c) + O(c) = O(c)
Work inside loop like comparision, appending , indexing, length calculation	O(n) * (O(c) + O(c) + O(c) + O(c)) = O(n) * O(c) = O(n)

Total complexity = O(c) + O(n) => O(n)

Q2.

```
class Node{
   int value;
   Node prev;
   Node next;
   Node(int val){
      this.value = val;
   }
}

class Q{
   Node head;
  Node tail;
  int size;

  Q(){
    this.head = null;
    this.tail = null;
    this.size = 0;
   }

   public int size(){
     return this.size;
   }

   public void push(int x){
        size++;
         Node node = new Node(x);
         if(tail==null || head==null){
           this.head = node;
           node.prev = this.head;
         }else{
            this.tail.next = node;
            node.prev = this.tail;
         }
          this.tail = node;
   }

    public int peekFirst(){
       if(size<=0) return -1;
     return this.head.value;
   }

   public int peekLast(){
      if(size<=0) return -1;
```

```
        return this.tail.value;
    }

    public int pop(){
        if(size<=0) return -1;
        size--;
       Node node = this.head;
       this.head = this.head.next;
       this.head.prev = null;

       node.next = null;
       node.prev = null;
      return node.value;
    }

    public int removeLast(){
         if(size<=0) return -1;
         size--;
         Node node = this.tail;
         this.tail.prev.next = null;
         this.tail = this.tail.prev;
         node.next = null;
         node.prev = null;
       return node.value;
    }
}

class MyStack {
    private Q queue;

    public MyStack() {
      this.queue = new Q();
    }
   public void push(int x) {
     this.queue.push(x);
   }
    public int pop() {
        return this.queue.removeLast();
    }
   public int top() {
     return this.queue.peekLast();
  }
  public boolean empty() {
    return this.queue.size()==0;
```

```
    }
}
```

Q3.

```
class Solution {
    public ListNode reverseList(ListNode head){
      // only one item in the list
      if(head == null || head.next == null) return head;
      ListNode tail = null;
      while(head.next!=null){
        ListNode node = head.next;
       head.next = tail;
       tail = head;
      head = node;
     }
    head.next = tail;
   return head;
  }
}
```

Q4.

```
class Node{
  int key;
  int value;
  Node next;
  Node prev;
  Node(int k, int v){
     this.key = k;
     this.value = v;
  }
}
```

```
class LRUCache {
    private final int capacity;
    private final HashMap<Integer, Node> dict = new HashMap<>();
    private Node head; // contains least used
```

```
private Node tail; // contains recently used

public LRUCache(int capacity) {
  this.capacity = capacity;
  this.head = new Node(-1,-1);
  this.tail = null;
}

public void addPrioritizedNode(Node node){
   if(this.tail == null){
      head.next = node;
      node.prev = head;
    }else{
       this.tail.next = node;
      node.prev = this.tail;
   }
   this.tail = node;
}

public void removeNode(Node node){
    Node prev = node.prev;
    Node next = node.next;
    node.prev = null;
    node.next = null;
    if(next!=null){
      next.prev = prev;
    }
   if(prev!=null){
      prev.next = next;
    }
 }

public int get(int key) {
      Node node = this.dict.get(key);
      if(node == null) return -1;
      // if this node is tail node, its priority is correct
      if(tail!=node){
          removeNode(node);
          addPrioritizedNode(node);
       }
      return node.value;
  }

 public void put(int key, int value) {
```

```
        Node node = this.dict.get(key);
        if (node != null) {
            if(tail==node){
              // already most recently used, just update the value
             node.value = value;
              return;
          }
          removeNode(node);
        }
        node = new Node(key, value);
        this.dict.put(key, node);
        addPrioritizedNode(node);
       if (this.dict.size() > this.capacity) {
          int deleteKey = this.head.next.key;
          Node deleteNode = this.dict.get(deleteKey);
          removeNode(deleteNode);
          this.dict.remove(deleteKey);
       }

    }
}
```