Q1.

Linked List implementation:

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Operation Logic | Time Complexity | Space Complexity |
| getListSize() | private static int listSize = 0;  return listSize; | O(1) | O(1) |
| isEmpty() | return listSize == 0; | O(1) | O(1) |
| addNodeAtHead(data) | if(isEmpty()){  head.data = data;  head.next = null;  listSize++;  return;  }  else{  Node newHead = new Node();  Node oldHead = head;  head = newHead;  head.data = data;  head.next = oldHead;  listSize++;  return;  } | Ω(1)  O(1) | O(1) |
| addNodeAtTail(data) | if(isEmpty()){  head.data = data;  head.next = null;  listSize++;  return;  }  else{  Node newTail = new Node();  newTail.data = data;  Node pivot = head;  for(int i = 0; i < listSize; i++){  if(pivot.next == null){  listSize++;  pivot.next = newTail;  return;  }  pivot = pivot.next;  }    } | Ω(1)  O(n) | O(1) |
| addElementAtPosition(int n, Object data) | if(isEmpty()){  System.out.println("Linked List is empty. Cannot add the element!");  return;  }  Node newNode = new Node();  newNode.data = data;  Node insertAfterThisNode = head;  for(int i = 0; i < n-2; i++){  insertAfterThisNode = insertAfterThisNode.next;  }  newNode.next = insertAfterThisNode.next;  insertAfterThisNode.next = newNode;  listSize++; | Ω(1)  O(n) | O(n) |
| reverseAList() | Node previousNode = null;  Node nextNode = null;  Node oldHead = head;  while(oldHead!=null){  nextNode = oldHead.next;  oldHead.next = previousNode;  previousNode = oldHead;  oldHead = nextNode;  head = previousNode;  } | O(n) | O(n) |
| getHeadNode() | if(head==null){  System.out.println("No data in the List!");  return null;  }  else  return head; | O(1) | O(1) |
| getTailNode() | if(head==null){  System.out.println("No data in the List!");  return null;  }  else{  Node node = head;  while(node.next!=null){  node = node.next;  }  return node;  } | O(n) | O(1) |
| deleteElement(int n) | if(isEmpty()){  System.out.println("LinkList is empty!");  return;  }  else{  Node deleteAfterThisNode = head;  for(int i = 0; i < n-2; i++){  deleteAfterThisNode = deleteAfterThisNode.next;  }  deleteAfterThisNode.next = deleteAfterThisNode.next.next;  listSize--;    }  } | O(n) | O(n) |

Java Code Implemented.

Q2. Stack using Linked List

Linked List implementation:

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Operation Logic | Time Complexity | Space Complexity |
| push(Object data) | if(isEmpty()){  Node newTop = new Node(data);  top = newTop;  stackSize++;  }  else{  Node newTop = top;  top = new Node(data);  top.next = newTop;  stackSize++;  } | O(1) | O(1) |
| isEmpty() | return listSize == 0; | O(1) | O(1) |
| Object pop() | if(isEmpty()){  System.out.println("No data in the list! Cannot Pop!");  return null;  }  else{  Object data = top.data;  top = top.next;  stackSize--;  return data;  } | Ω(1)  O(1) | O(1) |
| Object peek() | if(isEmpty()){  System.out.println("No Data in the Stack!");  return null;  }  else{  return top.data;  }  } | Ω(1)  O(1) | O(1) |

Java Code Implemented.

Q3. Queue using Linked List

Linked List implementation:

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Operation Logic | Time Complexity | Space Complexity |
| enqueue(Object data) | if(isEmpty()){  Node newNode = new Node(data);  front = newNode;  rear = newNode;  queueSize++;  }  else{  Node newRear = new Node(data);  rear.setNext(newRear);  rear = rear.getNext();  queueSize++;  } | O(1) | O(1) |
| isEmpty() | return listSize == 0; | O(1) | O(1) |
| Object dequeue() | if(isEmpty()){  System.out.println("No data in the list! Cannot Pop!");  return null;  }  else{  Object data = front.data;  front = front.next;  queueSize--;  return data;  } | Ω(1)  O(n) | O(1) |
| Object peek() | if(isEmpty()){  System.out.println("No Data in the Stack!");  return null;  }  else{  return front.data; | Ω(1)  O(1) | O(1) |

Java Code Implemented.

Q4.

Doubly Linked List implementation:

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Operation Logic | Time Complexity | Space Complexity |
| int getListSize() | return listSize; | O(1) | O(1) |
| isEmpty() | return listSize == 0; | O(1) | O(1) |
| addNodeAtHead(Object data) | Node newNode = new Node();  newNode.data = data;  if(isEmpty()){  head = newNode;  tail = head;  listSize++;  return;  }  else{  head.prev = newNode;  newNode.next = head;  head = newNode;  listSize++;  } | Ω(1)  O(1) | O(1) |
| addNodeAtTail(Object data) | Node newNode = new Node();  newNode.setData(data);  newNode.setPrev(tail);  if(tail!=null){  tail.next = newNode;  }  tail = newNode;  if(head == null){  head = newNode;  }  listSize++; | Ω(1)  O(n) | O(1) |
| addElementAtPosition(int n, Object data) | if(isEmpty()){  System.out.println("Doubly Linked List is empty. Cannot add the element!");  return;  }  Node newNode = new Node();  newNode.data = data;  Node insertAfterThisNode = head;  for(int i = 0; i < n-2; i++){  insertAfterThisNode = insertAfterThisNode.next;  }  newNode.next = insertAfterThisNode.next;  insertAfterThisNode.next.prev = newNode;  insertAfterThisNode.next = newNode;  newNode.prev = insertAfterThisNode;  listSize++; | Ω(1)  O(n) | O(n) |
| displayList() | Node pivot = head;  for(int i = 0; i < listSize; i++){  System.out.print(pivot.data + " ");  pivot = pivot.next;  } | O(n) | O(n) |
| reverseAList() | Node current = head;  Node next;  Node previous = null;  while(current != null){  next = current.next;  current.next = previous;  previous = current;  current = next;  } | O(n) | O(n) |
| Node getHeadNode() | if(head==null){  System.out.println("No data in the List!");  return null;  }  else  return head; | O(1) | O(1) |
| Node getTailNode() | if(head==null){  System.out.println("No data in the List!");  return null;  }  else{  Node node = head;  while(node.next!=null){  node = node.next;  }  return node;  } | O(n) | O(n) |
| deleteElement(int n) | if(isEmpty()){  System.out.println("LinkList is empty!");  return;  }  else if(n > listSize){  System.out.println("Invalid List Size!");  return;  }  else if(n == 1){  if(listSize == 1){  head = null;  tail = null;  listSize--;  return;  }  else{  head = head.next;  head.prev = null;  listSize--;  return;  }  }  else if(n ==listSize){  tail = tail.prev;  tail.next = null;  listSize--;  return;  }  else{  Node deleteAfterThisNode = head;  for(int i = 0; i < n-2; i++){  deleteAfterThisNode = deleteAfterThisNode.next;  }  deleteAfterThisNode.next = deleteAfterThisNode.next.next;  deleteAfterThisNode.next.prev = deleteAfterThisNode;  listSize--;    }  } | O(n) | O(n) |

Java Code Implemented.

Q5. Can we implement a stack data structure using Arrays? Give reasons pros/cons if yes/no?

* Yes, we can implement a stack data structure using arrays. But the following points need to be taken care of:
* 1. Stack is a dynamic data structure and arrays are static way of representing data.
* 2. We need to maintain a variable of type int which will keep the bottom of the stack (which will also be the size of the stack – 1).

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | null | null |

0 1 2 3

Top = 1

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | C | null |

0 1 2 3

Top = 2

|  |  |  |  |
| --- | --- | --- | --- |
| null | null | null | null |

0 1 2 3

Top = -1

Pros of using array implementation of Stack:

1. Push and pop operation are easy.

Cons of using array implementation of Stack:

1. A new variable has to be maintained to keep track of the top of the stack.
2. Stacks are dynamic, the size of the array needs to be changed continuously while inserting or deleting data.
3. Size of array is known before implementing the array, but size of stack is not known beforehand.
4. If size of array is defined previously, there is a chance the stack over flow will happen or we may not use up the entire space resulting into memory wastage.

Q6.

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Operation Logic | Time Complexity | Space Complexity |
| push(int value, Queue queue1, Queue queue2) | if(queue1.isEmpty()){  queue1.enqueue(value);  }  else{  for(int i = queue1.size(); i > 0; i--){  int dataValue = queue1.dequeue();  queue2.enqueue(dataValue);  }  queue1.enqueue(value);  for(int j = queue2.size(); j > 0; j--){  queue1.enqueue(queue2.dequeue());  }  } | O(n) | O(n) |
| Object pop(Queue queue1, Queue queue2) | if(queue1.isEmpty()){  System.out.println("Queue is empty, cannot dequeue");  return null;  }  else{  return queue1.dequeue();  } | O(1) | O(1) |
| Object peek(Queue queue1, Queue queue2) | if(queue1.isEmpty()){  System.out.println("Queue is empty, cannot dequeue");  return null;  }  else{  return queue1.peek();  } | Ω(1)  O(1) | O(1) |

Java Code Implemented.

Q7. There are two stacks needed for the implementation.

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Operation Logic | Time Complexity | Space Complexity |
| void enqueue(int value, Stack stack1, Stack stack2) | if(stack1.isEmpty()){  stack1.push(value);  }  else{  int sizeOfStack1 = stack1.size();  for(int i = 0; i < sizeOfStack1; i++){  stack2.push(stack1.pop());  }  stack1.push(value);  for(int j = 0; j < sizeOfStack1; j++){  stack1.push(stack2.pop());  }    } | O(n) | O(n) |
| Object dequeue(Stack stack1, Stack stack2) | if(stack1.isEmpty()){  System.out.println("Queue is empty, cannot dequeue");  return null;  }  else{  return stack1.pop();  } | O(1) | O(1) |
| Object peek(Stack stack1, Stack stack2) | if(stack1.isEmpty()){  System.out.println("Queue is empty, cannot peek");  return null;  }  else{  return stack1.peek();  } | Ω(1)  O(1) | O(1) |

Java Code Implemented.

Q8.

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Operation Logic | Time Complexity | Space Complexity |
| boolean isCyclic() | if(head==null){  System.out.println("List is empty!");  return false;  }  else if(head.next == null){  System.out.println("Acyclic as only one element in the list");  return true;  }  Node fast = head;  Node slow = head;  while(true){  slow = slow.next;  if(fast.next != null){  fast = fast.next.next;  }  else  return false;  if(slow == null || fast ==null){  return false;  }  if(slow == fast){  return true;  }  } | O(n) | O(n) |

Java Code Implemented.

Q9.

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Operation Logic | Time Complexity | Space Complexity |
| boolean solve(String input) | String stringToValidate = input;  int checker = 0;  for (int i = 0; i < stringToValidate.length(); i++) {  int value = stringToValidate.charAt(i) - 'a';  if ((checker & (1 << value)) > 0) {  System.out.println("Not unique!");  return false;  }  checker |= (1 << value);  }  System.out.println("Unique!");  return true; | O(n) | O(1) |

Java Code Implemented.

Q10. The idea data structure to be followed is Stacks.

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Operation Logic | Time Complexity | Space Complexity |
| boolean solve(String input) | String stringToValidate = input;  Stack so = new Stack();  for(int i = 0; i < stringToValidate.length(); i++){  if(stringToValidate.charAt(i) == '('){  so.push('(');  }  else if(stringToValidate.charAt(i) == '{'){  so.push('{');  }  else if(stringToValidate.charAt(i) == '['){  so.push('[');  }  else if(stringToValidate.charAt(i) == ')'){  if(so.isEmpty()){  System.out.println("Parantheses Mismatch!");  return false;  }  if(!String.valueOf(so.pop()).equals("(")){  System.out.println("Parantheses Mismatch!");  return false;  }  }  else if(stringToValidate.charAt(i) == '}'){  if(so.isEmpty()){  System.out.println("Parantheses Mismatch!");  return false;  }  if(!String.valueOf(so.pop()).equals("{")){  System.out.println("Parantheses Mismatch!");  return false;  }  }  else if(stringToValidate.charAt(i) == ']'){  if(so.isEmpty()){  System.out.println("Parantheses Mismatch!");  return false;  }  if(!String.valueOf(so.pop()).equals("[")){  System.out.println("Parantheses Mismatch!");  return false;  }  }  else{  System.out.println("Not a Parantheses!");  return false;  }  }  if(so.isEmpty()){  System.out.println("Parantheses Match!");  return true;  }  else{  System.out.println("Parantheses Mismatch!");  return true;  } | O(n) | O(n) |

Java Code Implemented.