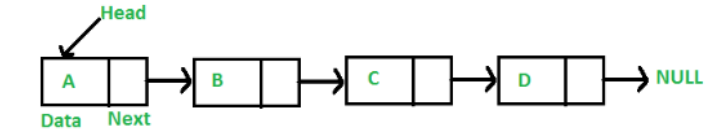
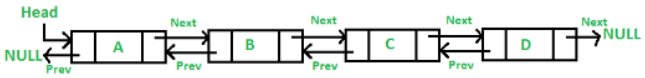
Data Structures – ITPC 203 Midsem Answer keys

1. Answer the following with brief justification:
    a. Iv) All of the above. Since an int array A[26] can contain no elements (i.e., 0), or it can contain 10 elements, or it can be full with 26 elements.
    b. ii) A[50] since, the requirement is to store the frequencies of 50 numbers (51-100).
    c. False, since last-in-first-out computations are efficiently supported by stacks. Queues support first in first out computations more efficiently.
    d. ii) q[1]. The rear index if incremented by 7 from 4 will cross q[9], then q[0], till it reaches q[1], as it is a standard circular queue.
    e. i) 1, since this is a special stack implementation which supports REVERSE. Generally, to implement queue using stacks, two standard stacks are needed, to reverse the elements.
2. Linked list vs doubly linked list: The singly linked list can be traversed only in the forward direction. The doubly linked list can be accessed in both directions.





Code for Reversal of Linked list:

```
struct Node {    // Link list node
   int data;
   struct Node* next;};
static void reverse(struct Node** head_ref){
        struct Node* prev = NULL;
        struct Node* current = *head_ref;
        struct Node* next = NULL;
        while (current != NULL) { //traverse the list
                next = current->next;    // Store next node
                current->next = prev;    // Reverse current node's pointer
                prev = current;  // Move pointers one position ahead.
                current = next;
        }
        *head_ref = prev; //last node of original LL is now head of reversed LL
}
int main() /* Driver code*/{
        struct Node* head = NULL;      /* Start with empty list */
        push(&head, 16);          //push() code not needed
        push(&head, 20);
        push(&head, 24);
        printf("Given linked list\n");
        printList(head);          //code not needed
        reverse(&head);//only this function is needed in your answer
        printf("\nReversed linked list \n");
        printList(head);}
```

3. Push: to insert an element into the stack

   Pop: to delete an element from the stack

   Overflow: the condition that occurs when we try to insert an element into a full stack.

   Underflow: the condition that occurs when we try to delete an element from an empty stack.

   Infix to postfix:

Infix Expression: 4 $ 2 * 3 - 3 + 8 / 4 / (1 + 1)		
Char Scanned	**Stack Contents**	**Postfix Expression**
4	Empty	4
$	$	4
2	$	4 2
*	*	4 2 $
3	*	4 2 $ 3
-	-	4 2 $ 3 *
3	-	4 2 $ 3 * 3
+	+	4 2 $ 3 * 3 -
8	+	4 2 $ 3 * 3 - 8
/	+/	4 2 $ 3 * 3 - 8
4	+/	4 2 $ 3 * 3 - 8 4
/	+/	4 2 $ 3 * 3 - 8 4 /
(	+/ (	4 2 $ 3 * 3 - 8 4 /
1	+/ (	4 2 $ 3 * 3 - 8 4 / 1
+	+/ (+	4 2 $ 3 * 3 - 8 4 / 1
1	+/ (+	4 2 $ 3 * 3 - 8 4 / 1 1
)	+/	4 2 $ 3 * 3 - 8 4 / 1 1 +
	Empty	4 2 $ 3 * 3 - 8 4 / 1 1 + / +

4. Aim: Given standard stack data structure, two ways to implement a queue:
   a. Method 1 - By making enQueue operation costly
      Summary: This method makes sure that oldest entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.
      enQueue(q, x):
      I. While stack1 is not empty, push everything from stack1 to stack2.
      II. Push x to stack1.
      III. Push everything back to stack1.

      deQueue(q):

      I. If stack1 is empty then error
      II. Pop an item from stack1 and return it

   b. Method 2 - By making deQueue operation costly

Summary: In this method, in enqueue operation, the new element is entered at the top of stack1. In dequeue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

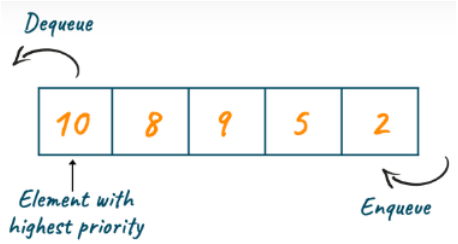
enQueue(q, x):

I. Push x to stack1

deQueue(q):

i. If both stacks are empty then error.
ii. If stack2 is empty, While stack1 is not empty, push everything from stack1 to stack2.
iii. Pop the top element from stack2 and return it.

5. Priority queue: queue that arranges elements based on their priority values. Elements with higher priority values are generally retrieved before elements with lower priority values.

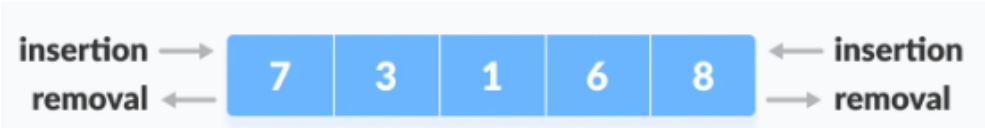


Types of priority queue:

Ascending Order Priority Queue: the element with a lower priority value is given a higher priority. For example, in a priority queue arranged in ascending order like 1,3,4,6,8,9,10. Here, 1 is the smallest number, therefore, it will get the highest priority. So, dequeue operation will return 1.

Descending order Priority Queue: the element with a higher priority value is given a higher priority. For example, in a priority queue arranged in descending order like 10,9,8,7,6,5. Here, 10 is the highest number, therefore, it will get the highest priority. So, dequeue operation will return 10.

Deque: Double Ended Queue is a type of Queue data structure that allows insert and delete at both ends of the queue.



Types of deque:

Input Restricted Deque. In this deque, input is restricted at a single end but allows deletion at both the ends. In the above diagram, if input at any one end is restricted it will result in an input restricted deque.

Output Restricted Deque. In this deque, output is restricted at a single end but allows insertion at both the ends. In the above diagram, if output at any one end is restricted it will result in an output restricted deque.

Condition to check if a circular queue is full:

if((rear == n -1 && front == 0) || (rear + 1 == front))

where rear and front indicates the rear and front indices of the queue and n is the maximum size of the queue.

6. Binary Search code:

```c
#include<stdio.h>
int binarySearch(int array[], int x, int low, int high) {
  while (low <= high) {
    int mid = low + (high - low) / 2;
    if (array[mid] == x) return mid;
    if (array[mid] < x)  low = mid + 1;
    else                 high = mid - 1;  }
  return -1;}

int main(void) {
  int array[] = {12, 13, 14, 15, 16, 17, 18, 19, 20};
  int n = sizeof(array) / sizeof(array[0]);
  int x = 12; //key
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)     printf("Not found");
  else     printf("Element is found at index %d", result);
  return 0;
}
```

Time Complexities

Best case complexity: O(1)
Average case complexity: O(log n)
Worst case complexity: O(log n)