DSA CHAPTER 4

Q1. Define Singly Linked List (SLL). Write and explain algorithms to insert a node at the beginning and at the end of a Singly Linked List.

=> Definition :- A **Singly Linked List (SLL)** is a linear data structure that consists of a sequence of elements called **nodes**. Each Node Contains two parts Data and Link part. Data part contains the information and link part contains the address of next node and first pointer contains the address of first node.

=> **Operations of Singly Link List**

1) Insertion at the Beginning of the linked list

```
Algorithm InsertBeg(first,value)
{
   Step 1 :  [Check For Availability Stack]
                  if avail = NULL then
              write "Available Stack is Empty"
              return first
   Step 2 : [Obtain address of next free node]
                  new <= avail
   Step 3 : [Remove free node from available stack]
                  avail <= avail(link)
   Step 4 : [Initialize node to the link list]
                  new(info)  <= value
                  new(link)  <= first
   Step 5 : [assign the address of the Temporary Node]
                  first <= new
   Step 6 : [finished]
                  return first
}
```

Explanation :-

- Check Availability If no free nodes are available (avail = NULL), print "Available Stack is Empty" and exit.
- Set new to the first node in the avail stack.
- Move avail to the next free node in the stack.
- Assign the given value to the new node's data field and set its link to point to the current first.
- Make new the new first of the list by setting first = new.

2) Insertion at the End of linked list

Algorithm InsertEnd(first,value)

{

Step 1 : [Check For Availability Stack]

if avail = NULL then

write "Available Stack is Empty"

return first

Step 2 : [Obtain address of next free node]

new <= avail

Step 3 : [Remove free node from available stack]

avail <= avail(link)

Step 4 : [Initialize node to the link list]

new(info) <= value

new(link) <= NULL

Step 5 : [is list is empty?]

if first = NULL then

first <= new

Step 6 : [initialize search for last node]

save <= new

Step 7 : [search end of the list]

Repeat while save(link) ≠ NULL

save <= save(link)

Step 8 : [set link field of last node to new]

save(link) <= new

Step 9 : [Finished]

return first

}

Explanation

- Check Availability If no free nodes are available (avail = NULL), print "Available Stack is Empty" and exit.
- Set new to the first node in the avail stack.
- Move avail to the next free node in the stack.
- Assign the given value to the new node's data field and set its link to point to the NULL.
- Make new the new first of the list by setting first = new.
- Check if List is Empty If the list is empty (first = NULL), set first to the new node and return it, as it will be the only node in the list.
- Use a temporary pointer save to traverse the list until you reach the last node (save(link) = NULL).
- Set the link field of the last node (save(link)) to point to the new node, thus appending it at the end.
- Return first with the new node successfully inserted at the end.

Q2. Write an algorithm to delete a node from the beginning and from the end of a Singly Linked List.

=> 1) Algorithm for Delete a node from the Beginning of Linked List

```
Algorithm DeleteBeg(first)
{
   Step 1 :  [Check For link list empty]
                  If first = NULL then
            write "link list is Empty"
            return first
   Step 2 : [store first node info in y]
                  y <= first(info)
   Step 3 : [Check link list has only one node]
                  if first(link) = NULL then
                     return NULL
   Step 4 : temp <= first
   Step 5 : first <= first(link)
   Step 6 : [return node to availability stack]
                  temp(link) <= avail
                  avail <= temp
   Step 7 : return first
}
```

2) Algorithm for Delete a node from the End of Linked List

```
Algorithm DeleteEnd(first)
{
   Step 1 :  [Check For link list]
                  If first = NULL then
            write "link list is Empty"
            return first
   Step 2 : [Check for the element in the list and delete it]
                  if first(link) = NULL then
                  y <= first(info)
                  temp <= first
                  first <= NULL
                  temp(link) <= avail
                  return NULL
   Step 3 : [Assign the address pointer by first pointer to save pointer]
                  save <= first
   Step 4 : Repeat while save(link) != NULL
                  pred <= save
                  save <= save(link)
   Step 5 : [Delete last node]
                  y <= save(info)
                  pred(link) <= NULL
   Step 6 : save(link) <= avail
                  avail <= save
   Step 7 : return first
}
```

Q3. Write and explain algorithms to search an element and count the number of nodes in a Singly Linked List.

=> 1) Algorithm for Search a element in Linked List

Algorithm search(first, target)

{

Step 1 : [initialize pos and count]

pos <= 0, count <= 0

Step 2 : [initialize pointer to first]

ptr <= first

Step 3 : Repeat steps 4,5 while ptr != NULL

count <= count + 1

Step 4 : if ptr(info) = target then

pos = count

write "target found"

return pos

go to step 6

else

ptr <= ptr(link)

Step 5 : [Target not found in the list]

write "target not found"

Step 6 : exit

}

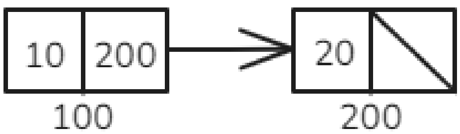
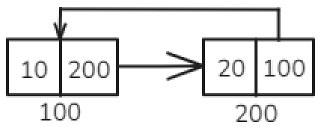
2) Algorithm for Count node in Linked List

```
Algorithm count(first)
{
   Step 1 : [Initialize count to 0]
                  count <= 0
   Step 2 : [Initialize pointer to first]
                  ptr <= first
   Step 3 :  [Repeat until the end of the list]
            Repeat steps 4,5 while ptr != NULL
   Step 4 : [Increment the count]
                  count <= count +1
   Step 5 : [Move pointer to the next node]
                  ptr <= ptr(link)
            [End of loop]
   Step 6 : return count
   Step 7 : exit
}
```

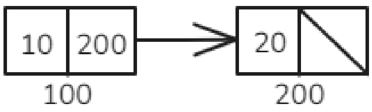
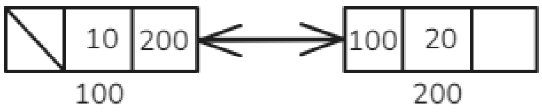
Q4. Differentiate following:

a. Singly Linked List and Circular Linked List
b. Singly Linked List and Doubly Linked List

=> Singly Linked List and Circular Linked List

Singly Linked List	Circular Linked List
Address part of the last node contains the NULL.	Address part of last node contains the address of first node.
We can traverse list in forward only manor.	We can traverse list in circular manor.
We cannot access previous nodes from the current node.	We can access previous node after visiting the last node.
Splitting and concatenation operation is difficult.	Splitting and concatenation operation is efficient.
Deletion of last node is easy.	Deletion of last node of list is difficult, so infinite loop is possible.
It is used if you want to access element only one time.	It is used if we want to access element in loop.
	

Singly Linked List and Doubly Linked List

Singly Linked List	Doubly Linked List
Each node has two parts data and a single link (pointer) to the next node.	Each node has three parts: data, a link to the next node, and a link to the previous node.
Consumes less memory as it stores only one pointer (next).	Consumes more memory as it stores two pointers (next and previous).
Can be traversed in only one direction (from head to tail).	Can be traversed in both directions (from head to tail and tail to head).
Inserting or deleting a node requires tracking the previous node. More complex for middle nodes.	Inserting or deleting a node is easier since the previous node can be accessed directly.
Requires more complex logic to reverse the links.	Easier to reverse as each node has pointers to both its neighbors.
	

Q5. List out applications of Linked List.

=>

1. Dynamic Memory Allocation
2. Memory Management
3. Handling Large Integers
4. Round-Robin Scheduling
5. Symbol Table Management in Compilers
6. Implementation of Stacks and Queues, Graphs, Hash Tables
7. Undo/Redo Functionality in Applications
8. Music and Playlist Management

Q.6 Explain Dynamic memory allocation with example.

=> **Definition:**

Dynamic memory allocation is the process of allocating memory during the runtime (execution) of a program, as opposed to static memory allocation, which occurs at compile time. It allows programs to request memory space while they are running, which is particularly useful when the size of data structures (such as arrays or linked lists) cannot be determined beforehand.

=> Common Functions for Dynamic Memory Allocation (C/C++)

malloc(): Allocates a specified number of bytes and returns a pointer to the first byte of the allocated memory. The memory is not initialized.

calloc(): Allocates memory for an array of elements, initializes it to zero, and returns a pointer to the memory.

realloc(): Changes the size of previously allocated memory.

free(): Deallocates previously allocated memory and makes it available for future use.

=> Example: Dynamic Memory Allocation in C

```
#include <stdio.h>
#include <stdlib.h>

int main() {
   int *arr;
   int n, i;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   arr = (int*) malloc(n * sizeof(int));
   if (arr == NULL) {
      printf("Memory allocation failed!\n");
      return 1
   }
   printf("Enter %d elements:\n", n);
   for (i = 0; i < n; ++i) {
      scanf("%d", &arr[i]);
```

```
    }
    printf("You entered: ");
    for (i = 0; i < n; ++i) {
        printf("%d ", arr[i]);
    }
    free(arr);
    return 0;
}
```