**UNIT-I**

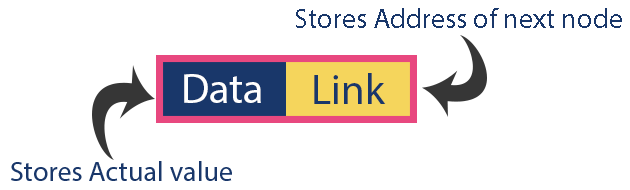
**1.Single Linked List**

**What is Single Linked List?**

Simply a list is a sequence of data, and the linked list is a sequence of data linked with each other.   
The formal definition of a single linked list is as follows...

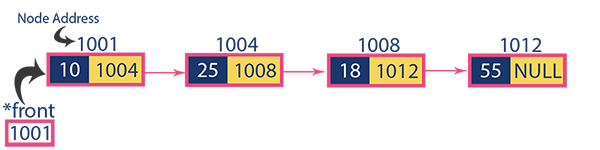
**Single linked list is a sequence of elements in which every element has link to its next element in the sequence.**

In any single linked list, the individual element is called as "Node". Every "Node" contains two fields, data field, and the next field. The data field is used to store actual value of the node and next field is used to store the address of next node in the sequence.  
The graphical representation of a node in a single linked list is as follows...



**Importent Points to be Remembered**  
  In a single linked list, the address of the first node is always stored in a reference node known as "front" (Some times it is also known as "head").  
  Always next part (reference part) of the last node must be NULL.

**Example**



**Operations on Single Linked List**

The following operations are performed on a Single Linked List

* **Insertion**
* **Deletion**
* **Display**

Before we implement actual operations, first we need to set up an empty list. First, perform the following steps before implementing actual operations.

* **Step 1 -**Include all the **header files** which are used in the program.
* **Step 2 -**Declare all the **user defined functions**.
* **Step 3 -**Define a **Node** structure with two members **data** and **next**
* **Step 4 -**Define a Node pointer **'head'** and set it to **NULL**.
* **Step 5 -**Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

**Insertion**

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

**Inserting At Beginning of the list**

We can use the following steps to insert a new node at beginning of the single linked list...

* **Step 1 -**Create a **newNode** with given value.
* **Step 2 -**Check whether list is **Empty** (**head** == **NULL**)
* **Step 3 -**If it is **Empty** then, set **newNode→next** = **NULL** and **head** = **newNode**.
* **Step 4 -**If it is **Not Empty** then, set **newNode→next** = **head** and **head** = **newNode**.

**Inserting At End of the list**

We can use the following steps to insert a new node at end of the single linked list...

* **Step 1 -**Create a **newNode** with given value and **newNode → next** as **NULL**.
* **Step 2 -**Check whether list is **Empty** (**head** == **NULL**).
* **Step 3 -**If it is **Empty** then, set **head** = **newNode**.
* **Step 4 -**If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
* **Step 5 -**Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
* **Step 6 -**Set **temp → next** = **newNode**.

**Inserting At Specific location in the list (After a Node)**

We can use the following steps to insert a new node after a node in the single linked list...

* **Step 1 -**Create a **newNode** with given value.
* **Step 2 -**Check whether list is **Empty** (**head** == **NULL**)
* **Step 3 -**If it is **Empty** then, set **newNode → next** = **NULL** and **head** = **newNode**.
* **Step 4 -**If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
* **Step 5 -**Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
* **Step 6 -**Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
* **Step 7 -**Finally, Set '**newNode → next** = **temp → next**' and '**temp → next** = **newNode**'

**Deletion**

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

**Deleting from Beginning of the list**

We can use the following steps to delete a node from beginning of the single linked list...

* **Step 1 -**Check whether list is **Empty** (**head** == **NULL**)
* **Step 2 -**If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
* **Step 3 -**If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
* **Step 4 -**Check whether list is having only one node (**temp → next** == **NULL**)
* **Step 5 -**If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)
* **Step 6 -**If it is **FALSE** then set **head** = **temp → next**, and delete **temp**.

**Deleting from End of the list**

We can use the following steps to delete a node from end of the single linked list...

* **Step 1 -**Check whether list is **Empty** (**head** == **NULL**)
* **Step 2 -**If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
* **Step 3 -**If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
* **Step 4 -**Check whether list has only one Node (**temp1 → next** == **NULL**)
* **Step 5 -**If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)
* **Step 6 -**If it is **FALSE**. Then, set '**temp2 = temp1**' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1 → next** == **NULL**)
* **Step 7 -**Finally, Set **temp2 → next**= **NULL** and delete **temp1**.

**Deleting a Specific Node from the list**

We can use the following steps to delete a specific node from the single linked list...

* **Step 1 -**Check whether list is **Empty** (**head** == **NULL**)
* **Step 2 -**If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
* **Step 3 -**If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
* **Step 4 -**Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
* **Step 5 -**If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
* **Step 6 -**If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
* **Step 7 -**If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
* **Step 8 -**If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).
* **Step 9 -**If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.
* **Step 10 -**If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).
* **Step 11 -**If **temp1** is last node then set **temp2 → next** = **NULL** and delete **temp1** (**free(temp1)**).
* **Step 12 -**If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

**Displaying a Single Linked List**

We can use the following steps to display the elements of a single linked list...

* **Step 1 -**Check whether list is **Empty** (**head** == **NULL**)
* **Step 2 -**If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.
* **Step 3 -**If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
* **Step 4 -**Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node
* **Step 5 -**Finally display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

**Implementation of Single Linked List using C Programming**

#include<stdio.h>

#include<conio.h>

void insertAtBeginning(int);

void insertAtEnd(int);

void insertBetween(int,int,int);

void display();

void removeBeginning();

void removeEnd();

void removeSpecific(int);

struct Node

{

int data;

struct Node \*next;

}\*head = NULL;

void main()

{

int choice,value,choice1,loc1,loc2;

clrscr();

while(1){

mainMenu: printf("\n\n\*\*\*\*\*\* MENU \*\*\*\*\*\*\n1. Insert\n2. Display\n3. Delete\n4. Exit\nEnter your choice: ");

scanf("%d",&choice);

switch(choice)

{

case 1: printf("Enter the value to be insert: ");

scanf("%d",&value);

while(1){

printf("Where you want to insert: \n1. At Beginning\n2. At End\n3. Between\nEnter your choice: ");

scanf("%d",&choice1);

switch(choice1)

{

case 1: insertAtBeginning(value);

break;

case 2: insertAtEnd(value);

break;

case 3: printf("Enter the two values where you wanto insert: ");

scanf("%d%d",&loc1,&loc2);

insertBetween(value,loc1,loc2);

break;

default: printf("\nWrong Input!! Try again!!!\n\n");

goto mainMenu;

}

goto subMenuEnd;

}

subMenuEnd:

break;

case 2: display();

break;

case 3: printf("How do you want to Delete: \n1. From Beginning\n2. From End\n3. Spesific\nEnter your choice: ");

scanf("%d",&choice1);

switch(choice1)

{

case 1: removeBeginning();

break;

case 2: removeEnd();

break;

case 3: printf("Enter the value which you wanto delete: ");

scanf("%d",&loc2);

removeSpecific(loc2);

break;

default: printf("\nWrong Input!! Try again!!!\n\n");

goto mainMenu;

}

break;

case 4: exit(0);

default: printf("\nWrong input!!! Try again!!\n\n");

}

}

}

void insertAtBeginning(int value)

{

struct Node \*newNode;

newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

if(head == NULL)

{

newNode->next = NULL;

head = newNode;

}

else

{

newNode->next = head;

head = newNode;

}

printf("\nOne node inserted!!!\n");

}

void insertAtEnd(int value)

{

struct Node \*newNode;

newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

newNode->next = NULL;

if(head == NULL)

head = newNode;

else

{

struct Node \*temp = head;

while(temp->next != NULL)

temp = temp->next;

temp->next = newNode;

}

printf("\nOne node inserted!!!\n");

}

void insertBetween(int value, int loc1, int loc2)

{

struct Node \*newNode;

newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

if(head == NULL)

{

newNode->next = NULL;

head = newNode;

}

else

{

struct Node \*temp = head;

while(temp->data != loc1 && temp->data != loc2)

temp = temp->next;

newNode->next = temp->next;

temp->next = newNode;

}

printf("\nOne node inserted!!!\n");

}

void removeBeginning()

{

if(head == NULL)

printf("\n\nList is Empty!!!");

else

{

struct Node \*temp = head;

if(head->next == NULL)

{

head = NULL;

free(temp);

}

else

{

head = temp->next;

free(temp);

printf("\nOne node deleted!!!\n\n");

}

}

}

void removeEnd()

{

if(head == NULL)

{

printf("\nList is Empty!!!\n");

}

else

{

struct Node \*temp1 = head,\*temp2;

if(head->next == NULL)

head = NULL;

else

{

while(temp1->next != NULL)

{

temp2 = temp1;

temp1 = temp1->next;

}

temp2->next = NULL;

}

free(temp1);

printf("\nOne node deleted!!!\n\n");

}

}

void removeSpecific(int delValue)

{

struct Node \*temp1 = head, \*temp2;

while(temp1->data != delValue)

{

if(temp1 -> next == NULL){

printf("\nGiven node not found in the list!!!");

goto functionEnd;

}

temp2 = temp1;

temp1 = temp1 -> next;

}

temp2 -> next = temp1 -> next;

free(temp1);

printf("\nOne node deleted!!!\n\n");

functionEnd:

}

void display()

{

if(head == NULL)

{

printf("\nList is Empty\n");

}

else

{

struct Node \*temp = head;

printf("\n\nList elements are - \n");

while(temp->next != NULL)

{

printf("%d --->",temp->data);

temp = temp->next;

}

printf("%d --->NULL",temp->data);

}

# 2.Stack Using Array

A stack data structure can be implemented using a one-dimensional array. But stack implemented using array stores only a fixed number of data values. This implementation is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable called **'top'**. Initially, the top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

# Stack Operations using Array

A stack can be implemented using array as follows...  
  
Before implementing actual operations, first follow the below steps to create an empty stack.

* **Step 1 -**Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
* **Step 2 -**Declare all the **functions** used in stack implementation.
* **Step 3 -**Create a one dimensional array with fixed size (**int stack[SIZE]**)
* **Step 4 -**Define a integer variable **'top'** and initialize with **'-1'**. (**int top = -1**)
* **Step 5 -**In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

## push(value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at **top** position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

* **Step 1 -**Check whether **stack** is **FULL**. (**top == SIZE-1**)
* **Step 2 -**If it is **FULL**, then display **"Stack is FULL!!! Insertion is not possible!!!"** and terminate the function.
* **Step 3 -**If it is **NOT FULL**, then increment **top** value by one (**top++**) and set stack[top] to value (**stack[top] = value**).

### pop() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

* **Step 1 -**Check whether **stack** is **EMPTY**. (**top == -1**)
* **Step 2 -**If it is **EMPTY**, then display **"Stack is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
* **Step 3 -**If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top--**).

#### display() - Displays the elements of a Stack

We can use the following steps to display the elements of a stack...

* **Step 1 -**Check whether **stack** is **EMPTY**. (**top == -1**)
* **Step 2 -**If it is **EMPTY**, then display **"Stack is EMPTY!!!"** and terminate the function.
* **Step 3 -**If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top. Display **stack[i]** value and decrement **i** value by one (**i--**).
* **Step 3 -**Repeat above step until **i** value becomes '0'.

# Implementation of Stack using Array

#include<stdio.h>

#include<conio.h>

void push(int);

void pop();

void display();

int stack[SIZE], top = -1;

void main()

{

int value, choice;

clrscr();

while(1){

printf("\n\n\*\*\*\*\* MENU \*\*\*\*\*\n");

printf("1. Push\n2. Pop\n3. Display\n4. Exit");

printf("\nEnter your choice: ");

scanf("%d",&choice);

switch(choice){

case 1: printf("Enter the value to be insert: ");

scanf("%d",&value);

push(value);

break;

case 2: pop();

break;

case 3: display();

break;

case 4: exit(0);

default: printf("\nWrong selection!!! Try again!!!");

}

}

}

void push(int value){

if(top == SIZE-1)

printf("\nStack is Full!!! Insertion is not possible!!!");

else{

top++;

stack[top] = value;

printf("\nInsertion success!!!");

}

}

void pop(){

if(top == -1)

printf("\nStack is Empty!!! Deletion is not possible!!!");

else{

printf("\nDeleted : %d", stack[top]);

top--;

}

}

void display(){

if(top == -1)

printf("\nStack is Empty!!!");

else{

int i;

printf("\nStack elements are:\n");

for(i=top; i>=0; i--)

printf("%d\n",stack[i]);

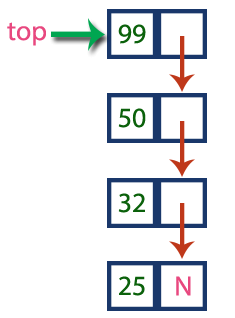
}

}

**3.Stack Using Linked List**

The major problem with the stack implemented using an array is, it works only for a fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.   
  
In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.

**Example**



In the above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

**Stack Operations using Linked List**

To implement a stack using a linked list, we need to set the following things before implementing actual operations.

* **Step 1 -**Include all the **header files** which are used in the program. And declare all the **user defined functions**.
* **Step 2 -**Define a '**Node**' structure with two members **data** and **next**.
* **Step 3 -**Define a **Node** pointer '**top**' and set it to **NULL**.
* **Step 4 -**Implement the **main** method by displaying Menu with list of operations and make suitable function calls in the **main**method.

**push(value) - Inserting an element into the Stack**

We can use the following steps to insert a new node into the stack...

* **Step 1 -**Create a **newNode** with given value.
* **Step 2 -**Check whether stack is **Empty** (**top** == **NULL**)
* **Step 3 -**If it is **Empty**, then set **newNode → next** = **NULL**.
* **Step 4 -**If it is **Not Empty**, then set **newNode → next** = **top**.
* **Step 5 -**Finally, set **top** = **newNode**.

**pop() - Deleting an Element from a Stack**

We can use the following steps to delete a node from the stack...

* **Step 1 -**Check whether **stack** is **Empty** (**top == NULL**).
* **Step 2 -**If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
* **Step 3 -**If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
* **Step 4 -**Then set '**top** = **top → next**'.
* **Step 5 -**Finally, delete '**temp**'. (**free(temp)**).

**display() - Displaying stack of elements**

We can use the following steps to display the elements (nodes) of a stack...

* **Step 1 -**Check whether stack is **Empty** (**top** == **NULL**).
* **Step 2 -**If it is **Empty**, then display **'Stack is Empty!!!'** and terminate the function.
* **Step 3 -**If it is **Not Empty**, then define a Node pointer **'temp'** and initialize with **top**.
* **Step 4 -**Display '**temp → data** --->' and move it to the next node. Repeat the same until **temp** reaches to the first node in the stack. (**temp → next** != **NULL**).
* **Step 5 -**Finally! Display '**temp → data** ---> **NULL**'.

**Implementation of Stack using Linked List | C Programming**

#include<stdio.h>

#include<conio.h>

struct Node

{

int data;

struct Node \*next;

}\*top = NULL;

void push(int);

void pop();

void display();

void main()

{

int choice, value;

clrscr();

printf("\n:: Stack using Linked List ::\n");

while(1){

printf("\n\*\*\*\*\*\* MENU \*\*\*\*\*\*\n");

printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");

printf("Enter your choice: ");

scanf("%d",&choice);

switch(choice){

case 1: printf("Enter the value to be insert: ");

scanf("%d", &value);

push(value);

break;

case 2: pop(); break;

case 3: display(); break;

case 4: exit(0);

default: printf("\nWrong selection!!! Please try again!!!\n");

}

}

}

void push(int value)

{

struct Node \*newNode;

newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

if(top == NULL)

newNode->next = NULL;

else

newNode->next = top;

top = newNode;

printf("\nInsertion is Success!!!\n");

}

void pop()

{

if(top == NULL)

printf("\nStack is Empty!!!\n");

else{

struct Node \*temp = top;

printf("\nDeleted element: %d", temp->data);

top = temp->next;

free(temp);

}

}

void display()

{

if(top == NULL)

printf("\nStack is Empty!!!\n");

else{

struct Node \*temp = top;

while(temp->next != NULL){

printf("%d--->",temp->data);

temp = temp -> next;

}

printf("%d--->NULL",temp->data);

}

}

# 4.Queue Datastructure Using Array

A queue data structure can be implemented using one dimensional array. The queue implemented using array stores only fixed number of data values. The implementation of queue data structure using array is very simple. Just define a one dimensional array of specific size and insert or delete the values into that array by using **FIFO (First In First Out) principle** with the help of variables **'front'** and '**rear**'. Initially both '**front**' and '**rear**' are set to -1. Whenever, we want to insert a new value into the queue, increment '**rear**' value by one and then insert at that position. Whenever we want to delete a value from the queue, then delete the element which is at 'front' position and increment 'front' value by one.

**Queue Operations using Array**

Queue data structure using array can be implemented as follows...  
  
Before we implement actual operations, first follow the below steps to create an empty queue.

* **Step 1 -**Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
* **Step 2 -**Declare all the **user defined functions** which are used in queue implementation.
* **Step 3 -**Create a one dimensional array with above defined SIZE (**int queue[SIZE]**)
* **Step 4 -**Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**. (**int front = -1, rear = -1**)
* **Step 5 -**Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

**enQueue(value) - Inserting value into the queue**

In a queue data structure, enQueue() is a function used to insert a new element into the queue. In a queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as a parameter and inserts that value into the queue. We can use the following steps to insert an element into the queue...

* **Step 1 -**Check whether **queue** is **FULL**. (**rear == SIZE-1**)
* **Step 2 -**If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
* **Step 3 -**If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **queue[rear]** = **value**.

**deQueue() - Deleting a value from the Queue**

In a queue data structure, deQueue() is a function used to delete an element from the queue. In a queue, the element is always deleted from **front** position. The deQueue() function does not take any value as parameter. We can use the following steps to delete an element from the queue...

* **Step 1 -**Check whether **queue** is **EMPTY**. (**front == rear**)
* **Step 2 -**If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
* **Step 3 -**If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as deleted element. Then check whether both **front** and **rear** are equal (**front** == **rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

**display() - Displays the elements of a Queue**

We can use the following steps to display the elements of a queue...

* **Step 1 -**Check whether **queue** is **EMPTY**. (**front == rear**)
* **Step 2 -**If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
* **Step 3 -**If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front+1**'.
* **Step 4 -**Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value reaches to **rear** (**i** <= **rear**)

**Implementation of Queue Datastructure using Array**

#include<stdio.h>

#include<conio.h>

void enQueue(int);

void deQueue();

void display();

int queue[SIZE], front = -1, rear = -1;

void main()

{

int value, choice;

clrscr();

while(1){

printf("\n\n\*\*\*\*\* MENU \*\*\*\*\*\n");

printf("1. Insertion\n2. Deletion\n3. Display\n4. Exit");

printf("\nEnter your choice: ");

scanf("%d",&choice);

switch(choice){

case 1: printf("Enter the value to be insert: ");

scanf("%d",&value);

enQueue(value);

break;

case 2: deQueue();

break;

case 3: display();

break;

case 4: exit(0);

default: printf("\nWrong selection!!! Try again!!!");

}

}

}

void enQueue(int value){

if(rear == SIZE-1)

printf("\nQueue is Full!!! Insertion is not possible!!!");

else{

if(front == -1)

front = 0;

rear++;

queue[rear] = value;

printf("\nInsertion success!!!");

}

}

void deQueue(){

if(front == rear)

printf("\nQueue is Empty!!! Deletion is not possible!!!");

else{

printf("\nDeleted : %d", queue[front]);

front++;

if(front == rear)

front = rear = -1;

}

}

void display(){

if(rear == -1)

printf("\nQueue is Empty!!!");

else{

int i;

printf("\nQueue elements are:\n");

for(i=front; i<=rear; i++)

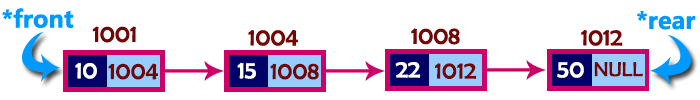
printf("%d\t",queue[i]);

} }

**5.Queue Using Linked List**

The major problem with the queue implemented using an array is, It will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.   
In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.

**Example**



In above example, the last inserted node is 50 and it is pointed by '**rear**' and the first inserted node is 10 and it is pointed by '**front**'. The order of elements inserted is 10, 15, 22 and 50.

**Operations**

To implement queue using linked list, we need to set the following things before implementing actual operations.

* **Step 1 -**Include all the **header files** which are used in the program. And declare all the **user defined functions**.
* **Step 2 -**Define a '**Node**' structure with two members **data** and **next**.
* **Step 3 -**Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.
* **Step 4 -**Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

**enQueue(value) - Inserting an element into the Queue**

We can use the following steps to insert a new node into the queue...

* **Step 1 -**Create a **newNode** with given value and set '**newNode → next**' to **NULL**.
* **Step 2 -**Check whether queue is **Empty** (**rear** == **NULL**)
* **Step 3 -**If it is **Empty** then, set **front** = **newNode** and **rear** = **newNode**.
* **Step 4 -**If it is **Not Empty** then, set **rear → next** = **newNode** and **rear** = **newNode**.

**deQueue() - Deleting an Element from Queue**

We can use the following steps to delete a node from the queue...

* **Step 1 -**Check whether **queue** is **Empty** (**front == NULL**).
* **Step 2 -**If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function
* **Step 3 -**If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.
* **Step 4 -**Then set '**front** = **front → next**' and delete '**temp**' (**free(temp)**).

**display() - Displaying the elements of Queue**

We can use the following steps to display the elements (nodes) of a queue...

* **Step 1 -**Check whether queue is **Empty** (**front** == **NULL**).
* **Step 2 -**If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.
* **Step 3 -**If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.
* **Step 4 -**Display '**temp → data** --->' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp → next** != **NULL**).
* **Step 5 -**Finally! Display '**temp → data** ---> **NULL**'.

**Implementation of Queue Datastructure using Linked List**

#include<stdio.h>

#include<conio.h>

struct Node

{

int data;

struct Node \*next;

}\*front = NULL,\*rear = NULL;

void insert(int);

void delete();

void display();

void main()

{

int choice, value;

clrscr();

printf("\n:: Queue Implementation using Linked List ::\n");

while(1){

printf("\n\*\*\*\*\*\* MENU \*\*\*\*\*\*\n");

printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");

printf("Enter your choice: ");

scanf("%d",&choice);

switch(choice){

case 1: printf("Enter the value to be insert: ");

scanf("%d", &value);

insert(value);

break;

case 2: delete(); break;

case 3: display(); break;

case 4: exit(0);

default: printf("\nWrong selection!!! Please try again!!!\n");

}

}

}

void insert(int value)

{

struct Node \*newNode;

newNode = (struct Node\*)malloc(sizeof(struct Node));

newNode->data = value;

newNode -> next = NULL;

if(front == NULL)

front = rear = newNode;

else{

rear -> next = newNode;

rear = newNode;

}

printf("\nInsertion is Success!!!\n");

}

void delete()

{

if(front == NULL)

printf("\nQueue is Empty!!!\n");

else{

struct Node \*temp = front;

front = front -> next;

printf("\nDeleted element: %d\n", temp->data);

free(temp);

}

}

void display()

{

if(front == NULL)

printf("\nQueue is Empty!!!\n");

else{

struct Node \*temp = front;

while(temp->next != NULL){

printf("%d--->",temp->data);

temp = temp -> next;

}

printf("%d--->NULL\n",temp->data);

} }