Data Structures and Algorithms

Lecture 5

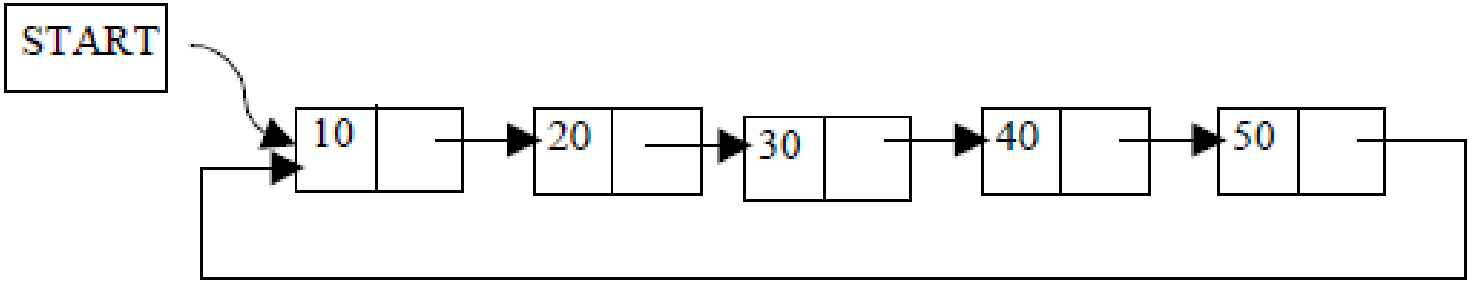
Linked List continue….

Dept. of ICT, Faculty of Technology- UoR, 2023

Circular Linked List

Circular Linked list

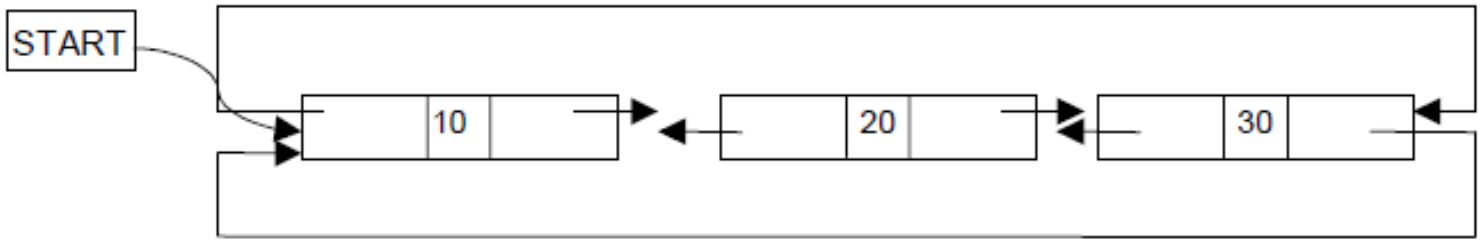
- A circular linked list is one, which has no beginning and no end.
- A singly linked list can be made a circular linked list by simply storing the address of the very first node in the linked field of the last node.



Circular Linked list

Circular Linked list

- A circular doubly linked list has both the successor pointer and predecessor pointer in circular manner.



Circular Doubly Linked list

Advantages of Circular Linked Lists

- Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
- Useful for implementation of queue. Unlike this implementation, we don’t need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
- useful in applications to repeatedly go around the list.

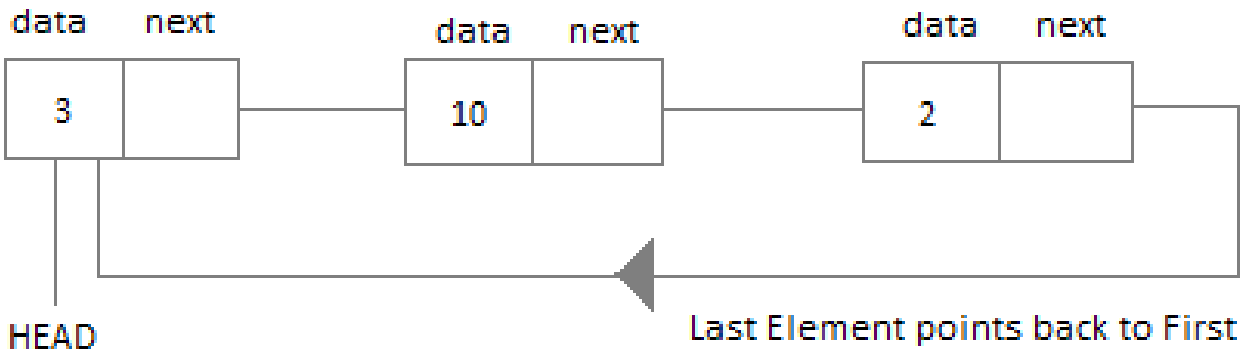
Example: when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

Application of Circular Linked List

- The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.
- Another example can be Multiplayer games. All the Players are kept in a Circular Linked List and the pointer keeps on moving forward as a player's chance ends.
- Circular Linked List can also be used to create Circular Queue. In a Queue we have to keep two pointers, FRONT and REAR in memory all the time, where as in Circular Linked List, only one pointer is required.

Implementing Circular Linked List

Implementing a circular linked list is very easy and almost similar to linear linked list implementation, with the only difference being that, in circular linked list the last Node will have it's next point to the Head of the List. In Linear linked list the last Node simply holds NULL in it's next pointer.



PRIMITIVE FUNCTIONS IN CIRCULAR LISTS

The structure definition of the circular linked lists and the linear linked list is the same:

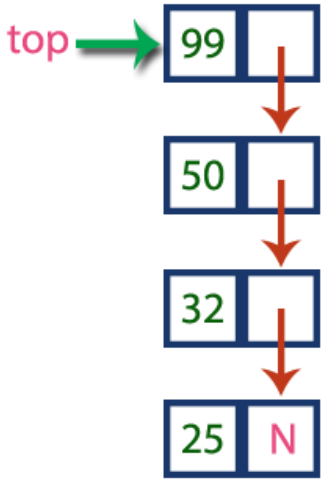
```
struct node{
int info;
struct node *next;
};
typedef struct node *NODEptr;
```

Stack and Queue implementation using Linked List

- The major problem with the stack/Queue implemented using array is, it works only for fixed number of data values.
- Stack/Queue implemented using array is not suitable, when we don't know the size of data which we are going to use.
- The stack/queue implemented using linked list can work for unlimited number of values.
- No need to fix the size at the beginning of the implementation.
- The Stack/Queue implemented using linked list can organize as many data values as we want.

Stack implementation using Linked List

- In linked list implementation of a stack, every new element is inserted as '**top**' element.
- Every newly inserted element is pointed by '**top**'.



In 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 implementation using Linked List

First thing required to make a stack using a linked list.

```
struct node
{
   int data;
   struct node *next;
};
typedef struct node node;

node *top;
```

to create a stack is the creation of the 'top' pointer. This is done in last line of the code.
i.e., node *top

```
void initialize()
{
   top = NULL;
}
```

Initialize function to initialize the stack by making the top NULL.

Stack implementation using Linked List

Most important operations on a stack are push and pop.

push

The steps for push operation are:

- Make a new node.
- Give the 'data' of the new node its value.
- Point the 'next' of the new node to the top of the stack.
- Make the 'top' pointer point to this new node.

```
void push(int value)
{
    /*
    Let the initial stack is:
       Top
     ___|___          _______          _______
    |   1   |____\ |   3   |____\ |   5   |____\ NULL
    |_______|    / |_______|    / |_______|    /
    and the value to be pushed is 10
    */
    node *tmp;
    tmp = malloc(sizeof(node));
    tmp -> data = value;
    /*
     This will make a new node
     _______
    |   10  |
    |_______|
    */
    tmp -> next = top;
    /*
    This step will do
                       Top
     _______          ___|___          _______          _______
    |   10  |____\ |   1   |____\ |   3   |____\| 5     |____\ NULL
    |_______|    / |_______|    / |_______|    /|_______|    /
    */
```

```
    top = tmp;
    /*
    This step will do
       Top
        |
     ___|__          ______          ______          ______
    |   10  |___\ |   1   |___\ |   3   |___\| 5     |___\ NULL
    |______|    / |______|    / |______|    /|______|    /
    */
}
```

Stack implementation using Linked List

pop

In pop operation, we delete the topmost node and returns its value.

Step1: Make a temporary node.

Step2: Point this temporary node to the top of the stack

Step3: Store the value of 'data' of this temporary node in a variable.

Step4: Point the 'top' pointer to the node next to the current top node.

Step5: Delete the temporary node using the 'free' function.

Step6: Return the value stored in step 3.

Stack implementation using Linked List

```
int pop()
{
    node *tmp;
    int n;
    tmp = top;
    n = tmp->data;
    top = top->next;
    free(tmp);
    return n;
}
```

node *tmp – Step 1

tmp = top – Step 2

n = tmp->data – Step 3

top = top->next – Step 4

free(tmp) – Step 5

return n – Step 6

Stack implementation using Linked List

create some functions for the other operations also e.g., top, isempty, etc

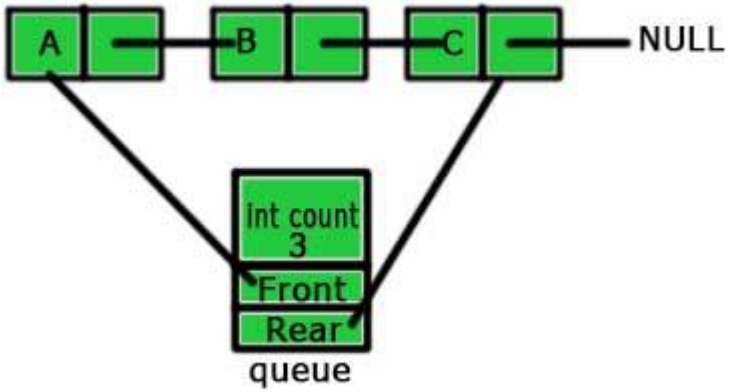
```
int Top()
{
    return top->data;
}

int isempty()
{
    return top==NULL;
}
```

Queue implementation using Linked List

The first thing required to make a queue using a linked list

```
struct node
{
   int data;
   struct node *next;
};
typedef struct node node;
```

The next thing is to create a structure 'queue' which will store the front node, rear node and the total number of nodes in the linked list. This is similar to the picture given below:

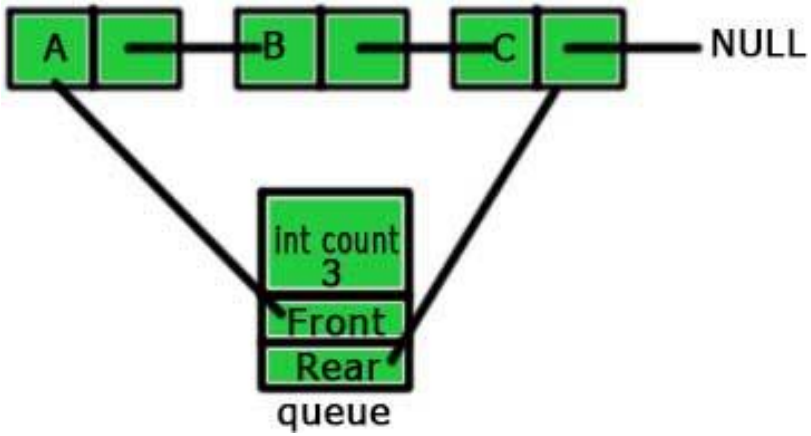


You can see that the structure 'queue' has three part – count, front and rear.

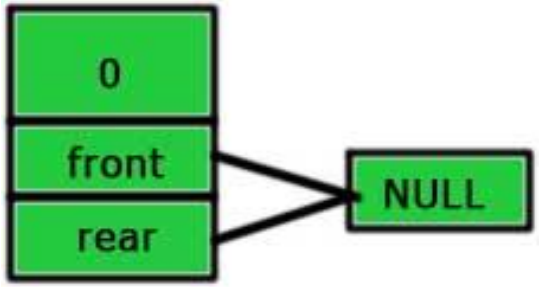
Queue implementation using Linked List

make the structure of this queue

```
struct queue
{    int count;
    node *front;
    node *rear;
};
typedef struct queue queue;
```



Initialize the queue and this will be done by making the count of the 'queue' 0 and pointing 'rear' and 'front' to NULL.




```
void initialize(queue *q)
{
    q->count = 0;
    q->front = NULL;
    q->rear = NULL;
}
```

Queue implementation using Linked List

```
struct node
{  int data;
   struct node *next;
};
typedef struct node node;

struct queue
{  int count;
   node *front;
   node *rear;
};
typedef struct queue queue;

void initialize(queue *q)
{
   q->count = 0;
   q->front = NULL;
   q->rear = NULL;
}
```

Queue implementation using Linked List

Check whether the queue is empty or not.

The ‘rear’ (or ‘front’) will be NULL for an empty queue.

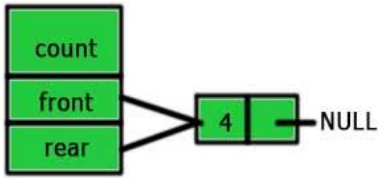
So, we can easily check whether a queue is empty or not by checking whether the ‘rear’ is NULL or not.

```
int isempty(queue *q)
{
    return (q->rear == NULL);
}
```

Queue implementation using Linked List

enqueue

The steps for the enqueue operations are:



- Make a new node

(node *tmp; tmp = malloc(sizeof(node))).

- Give the 'data' of the new node its value

(tmp -> data =value).

- If the queue is empty then point both 'front' and 'rear' of the queue to this node

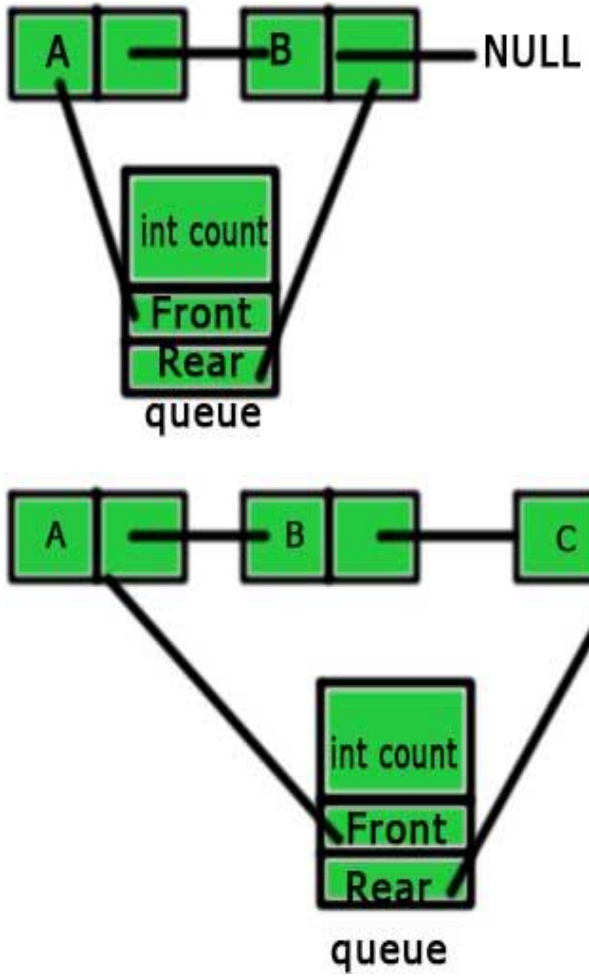
(q->front = q->rear = tmp;).

- If it is not, then point the rear of the queue to this new node and then make this new node rear

(q->rear->next = tmp; q->rear = tmp;).

Queue implementation using Linked List

```
void enqueue(queue *q, int value)
{
    node *tmp;
    tmp = malloc(sizeof(node));
    tmp->data = value;
    tmp->next = NULL;
    if(!isempty(q))
    {
        q->rear->next = tmp;
        q->rear = tmp;
    }
    else
    {
        q->front = q->rear = tmp;
    }
    q->count++;  }
```



Queue implementation using Linked List

dequeue

Step1: Make a temporary node.

Step2: Point this temporary node to the front node of the queue.

Step3: Store the value of 'data' of this temporary node in a variable.

Step4: Point the 'front' pointer to the node next to the current front node.

Step5: Delete the temporary node using the 'free' function.

Step6: Return the value stored in step 3.

Queue implementation using Linked List

```
int dequeue(queue *q)
{
    node *tmp;
    int n = q->front->data;
    tmp = q->front;
    q->front = q->front->next;
    q->count--;
    free(tmp);
    return(n);
}
```

node *tmp – Step 1

tmp = q->front – Step 2

n = q->front->data – Step 3

q->front = q->front->next – Step 4

free(tmp) – Step 5

return n – Step 6

END…