Unit 5: Linked List

a. Concept and Definition
b. Inserting and deleting nodes
c. Linked implementation of a stack (PUSH/POP)
d. Linked implementation of a queue (Insert/Remove)
e. Circular List
   - Stack as a circular list (PUSH/POP)
   - Queue as a circular list (Insert/Remove)
f. Doubly Linked List (Insert/Remove)

List

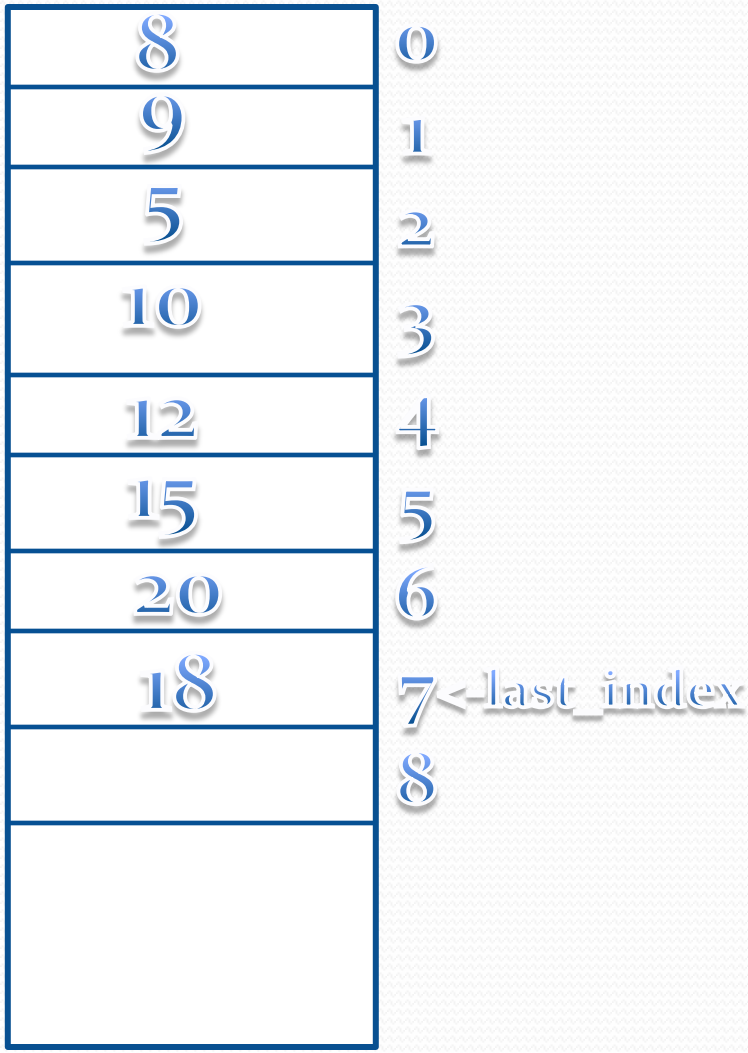
- A list is a set of items organized sequentially. For example, an array is a list where the sequential organization is provided implicitly by its index.
- We can represent a list in two ways:
  - Contiguous List
  - Linked List

Contiguous List

- A contiguous list is a list in array implementation. In this representation, each item is contiguous to next item. The operations of insertion and deletion of items can be done to/from anywhere within the list.

Implementation of Contiguous List

- To implement a contiguous list, we need to define a large enough array to hold all the items of the list. The first item of the list is placed at 0^{th} position of array and successive items in successive positions of the array. The last item of the list is indicated by "last_index". For example, in the figure last_index=7



Implementation of Contiguous List…

- **Assumptions**
  - ***MAXLIST*** is defined.
  - ***CList[MAXLIST]*** is defined as an array to hold the items of the list.
  - ***last_index*** is defined to indicate index the last element.
  - Initially, ***last_index=-1.***
  - ***Insertion:***
    1. Read item ***x*** to insert.
    2. If ***last_index==MAXLIST-1***, declare list full and return.
    3. If ***last_index==-1***, increment ***last_index*** and perform ***Clist[o]=x*** and return.
    4. Read position ***pos*** to insert.
    5. If ***pos>last_index+1***, declare invalid and return.
    6. ***i=last_index;***
    7. ***while(i>=pos)***
       CList[i+1]=CList[i];
       i=i-1;
    8. ***CList(pos)=x;***
    9. ***Increment last_index***
    10. ***Return***

Implementation of Contiguous List…

- **Assumptions**
  - ***Deletion:***
    1. If ***last_index==-1***, declare list empty and return.
    2. Read position ***pos*** to delete.
    3. If ***pos>last_index***, declare invalid and return.
    4. ***x=CList[pos];***
    5. ***i=pos;***
    6. ***while(i<last_index)***

       CList[i]=CList[i+1];

       i=i+1;
    7. ***Decrement last_index***
    8. ***Return x as deleted item***

Classwork

- **Implement Contiguous List.**

C code for Contiguous List

```
#define MAXLIST 100
void insert(struct CList *, int );
int remove(struct CList *);
void display(struct CList *);

struct CList
   {
   int items[MAXLIST];
   int last_index;
   }list;
```

```
void main()
{
int x;
int option;
char ch='y';
list.last_index=-1;
printf("\n What do you want to do?");
printf("\n1.Add an item");
printf("\n2.Delete an item");
printf("\n3.Display List");
printf("\n4.Exit");
while(ch=='y')
    {
    printf("\n Enter your option:");
    scanf("%d", &option);
    switch(option)
       {
        case 1:
              printf("\n Enter item to insert:");
              scanf("%d", &x);
              insert(&list, x);
              break;
        case 2:
              x=remove(&list);
              printf("\n The removed item is: %d", x);
                    break;
        case 3:
              display(&list);
              break;
        case 4:
              exit();
        default:
              printf("Invalid Option");
              }
        printf("\n Do you want to continue(y/n)?");
        scanf(" %c", &ch);
        }
getch();
}
```

```
void insert(struct CList *list, int x)
{
int pos;
int i;
if(list->last_index==MAXLIST-1)
    {
    printf("\n List if Full.");
    return;
    }
if(list->last_index==-1)
    {
    ++(list->last_index);
    list->items[0]=x;
    return;
    }

printf("\n Enter position to insert:");
scanf("%d", &pos);
if(pos>(list->last_index)+1)
    {
    printf("\n Invalid");
    return;
    }
else
    {
    i=list->last_index;
    while(i>=pos)
        {
        list->items[i+1]=list->items[i];
         i=i-1;
         }
    list->items[pos]=x;
    (list->last_index)++;
    return;
    }
}
```

```
int remove(struct CList *list)
{
int pos;
int i;
int x;
if(list->last_index==-1)
    {
    printf("\n List is empty.");
    return -1;
    }
else
    {
    printf("\n Enter position to delete:");
    scanf("%d", &pos);

    if(pos>list->last_index)
            {
            printf("\n Invalid");
            return -1;
            }
else
            {
            x=list->items[pos];
            i=pos;
            while(i<list->last_index)
                    {
                    list->items[i]=list->items[i+1];
                    i=i+1;
                    }
            (list->last_index)--;
            return x;
            }
    }
}

void display(struct CList *list)
{
int i;
for(i=0;i<=list->last_index;i++)
    printf("\n  CList[%d]=%d", i, list->items[i]);
}
```

Linked List

- A ***linked list*** is defined as a collection of ***nodes*** in which each ***node*** has two parts:
  - (i) information part and
  - (ii) link part
- To store a set of items in linked list representation, the information part of a ***node*** contains the actual item of the list while the link part of the ***node*** contains the ***address*** of the next ***node*** in the list. The address of the last node of a ***linked list*** is NULL.

Fig: Linked list with two nodes

C Implementation of Linked List

- In C, a linked list template is created using a self-referential structure and nodes of the linked list are created and removed using dynamic memory allocation functions like ***malloc()*** and ***free()***. We consider ***startnode*** as an external pointer. This pointer helps in creating and accessing other nodes in the linked list.
- The following code defines the structure for linked list and creates a ***startnode***.

```
struct linkedlist
        {
        int info;
        struct linkedlist *link;
        };
struct linkedlist *startnode;
startnode=NULL;
```

- *Note: The pointer member variable **link** points to (i.e. contains address of) a structure variable of the structure type **linkedlist**.*

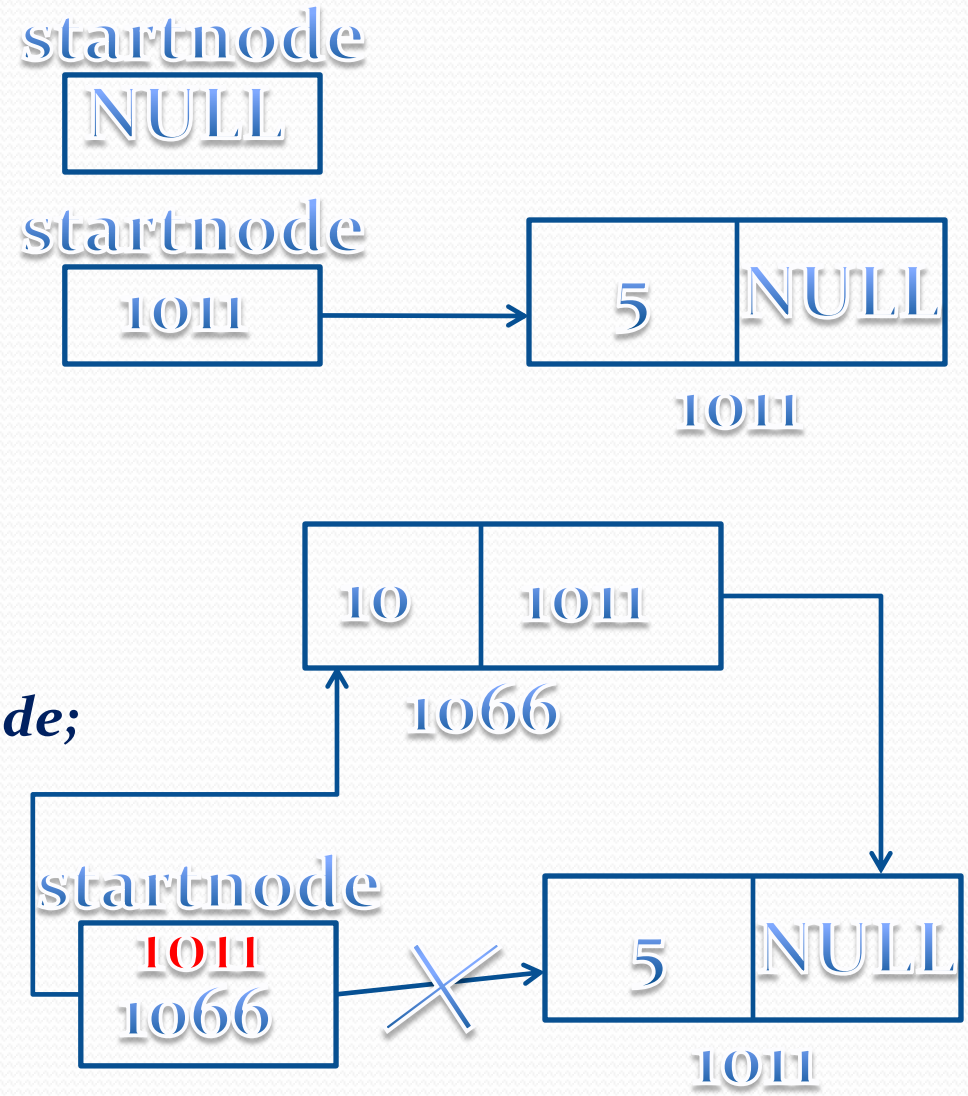
C Implementation of Linked List…

- Whenever ***startnode==NULL***, the list is empty.
- ***Insertion***: For insertion, we need to do the following three things:
  1. ***Allocating a node***: This can be done as,
     ```
     struct linkedlist *node;
     node=(struct linkedlist *)malloc(sizeof(struct linkedlist));
     ```
  2. ***Assigning the data***: This can be done as,
     ```
     node->info=x;
     ```
  3. ***Adjusting the pointers***: This depends on where we want to insert the node in the linked list. There are three options available:
     I. **Insert at Beginning**
     II. **Insert at End**
     III. **Insert after any position**

C Implementation of Linked List…

I. ***Insert at Beginning***

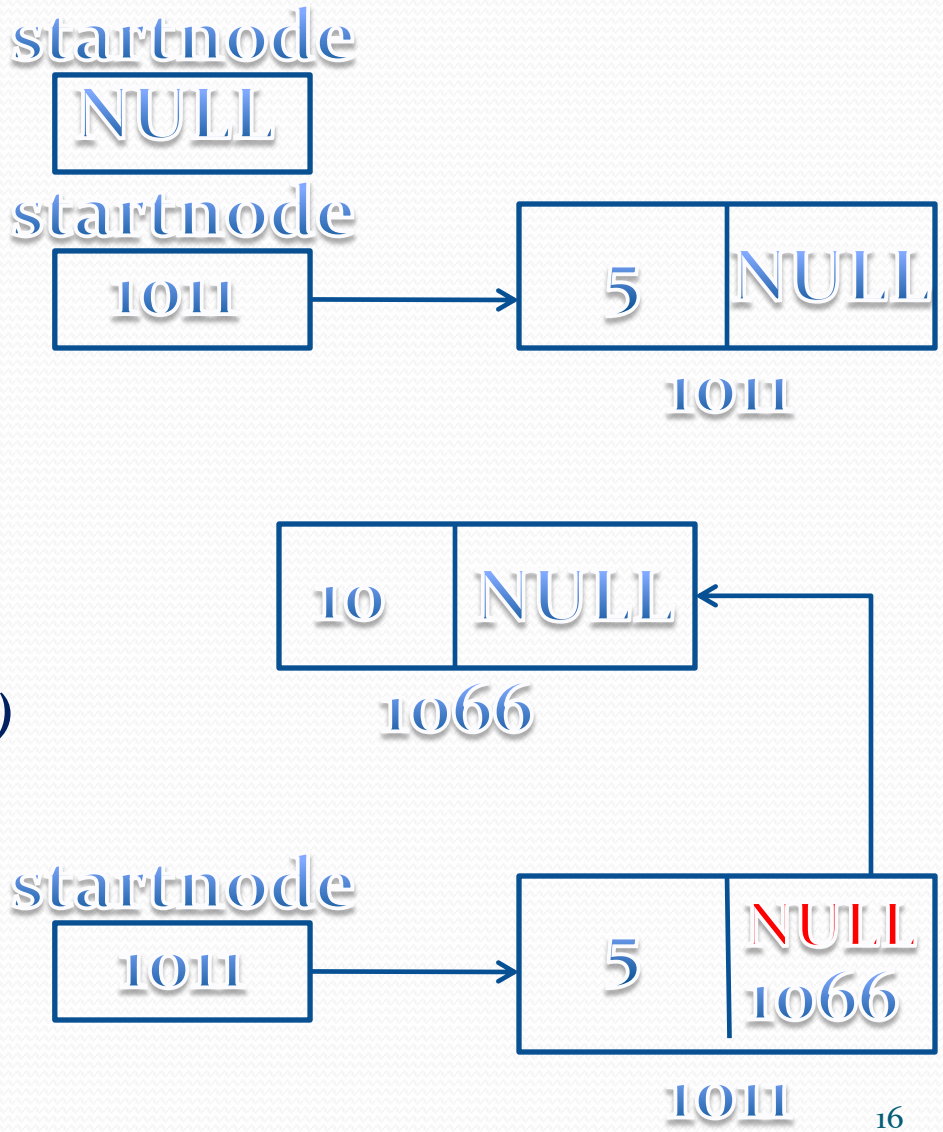
```
if (startnode==NULL)
        {
        node->link=NULL;
        startnode=node;
        }
else
        {
        node->link=startnode;
        startnode=node;
        }
```



C Implementation of Linked List…

II. ***Insert at End***

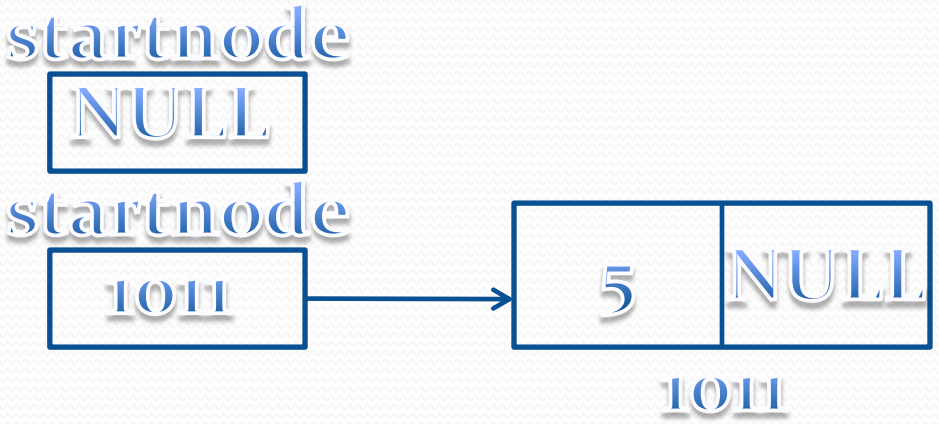
```
struct linkedlist *temp;
node->link=NULL
if(startnode==NULL)
        startnode=node;
else
        {
        temp=startnode;
        while(temp->link!=NULL)
                temp=temp->link
        temp->link=node;
        }
```



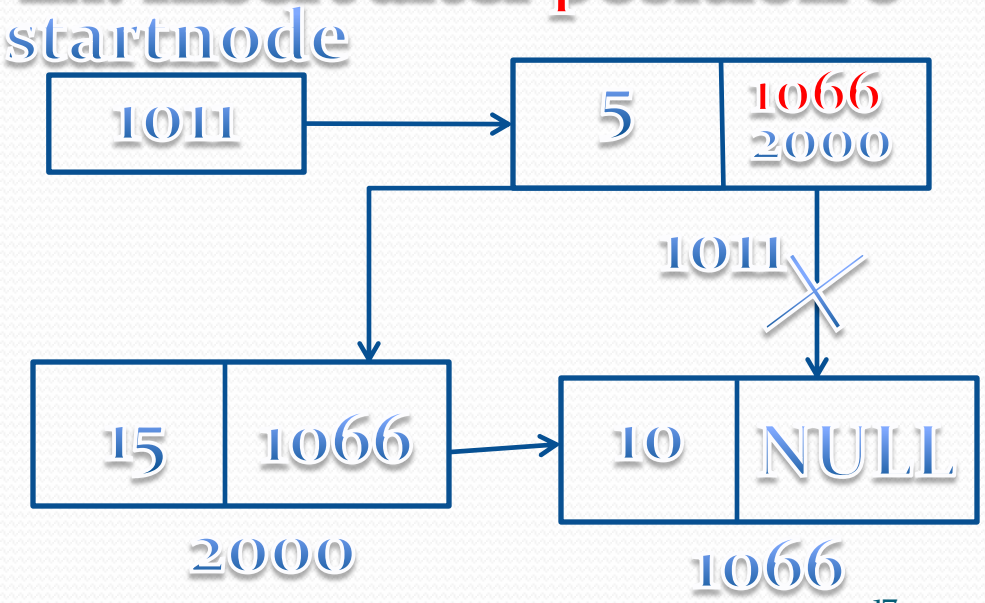
C Implementation of Linked List…

III. ***Insert after any position***

```
struct linkedlist *temp;
if(startnode==NULL)
        {
        node->link=NULL;
        startnode=node
        }
else
        {
        temp=startnode;
        for(i=0;i<position;i++)
                temp=temp->link;
        node->link=temp->link;
        temp->link=node;
        }
```



Ex: Insert after position 0



C Implementation of Linked List…

- ***Deletion***: For deletion, we have three cases:
  I. **Deleting first node of linked list**
  II. **Deleting last node of linked list**
  III. **Deleting specified node of linked list**

- **Assignment**
  - **Write the steps involved in deleting nodes of a linked list in all three cases mentioned above.**
  - **Write the steps involved for:**
    - **Displaying a linked list**
    - **Reversing a linked list**
    - **Searching a node with the specified information in the linked list**

Complete Implementation of Linked List

```
#include <stdio.h>
struct linkedlist
    {
    int info;
    struct linkedlist *link;
    };

void insertatfirst(struct linkedlist **,int);
void insertatlast(struct linkedlist **, int);
void insertafteranyposition(struct linkedlist **, int, int);
void deletefromfirst(struct linkedlist **);
void deletefromlast(struct linkedlist **);
void deletefromanyposition(struct linkedlist **,int);
void display(struct linkedlist **);
void reverse(struct linkedlist **);
```

```
void main()
{
struct linkedlist *startnode;
int item;
int position;
int option;
char ch='y';
startnode=NULL;
clrscr();
printf("What do you want to do?");
printf("\n1.Insert node at first");
printf("\n2.Insert node at last");
printf("\n3.Insert node at any position");
printf("\n4.Delete node at first");
printf("\n5.Delete node at last");
printf("\n6.Delete node at any position");
printf("\n7.Display the linked list");
printf("\n8.Display reverse of linked list");
while(ch=='y')
    {
    printf("\n Enter your option:");
    scanf("%d", &option);
    switch(option)
            {
            case 1:
                        printf("\n Enter the information part:");
                        scanf("%d", &item);
                        insertatfirst(&startnode, item);
                        break;
```

```
case 2:
        printf("\n Enter the information part:");
        scanf("%d", &item);
        insertatlast(&startnode, item);
        break;
case 3:
        printf("\n Enter the information part:");
        scanf("%d", &item);
        printf("\n Enter the position to insert after:");
        scanf("%d", &position);
   insertafteranyposition(&startnode,item,position);
        break;
case 4:
        deletefromfirst(&startnode);
        break;
case 5:
        deletefromlast(&startnode);
        break;
case 6:
        printf("\n Enter the position to delete:");
        scanf("%d", &position);
        deletefromanyposition(&startnode,position);
        break;

case 7:
        display(&startnode);
        break;
case 8:
        reverse(&startnode);
        break;
default:
        printf("\n WRONG OPTION");
        }
    printf("\n Do you want to continue(y/n)?");
    scanf(" %c", &ch);
    }
getch();
}
```

```
void insertatfirst(struct linkedlist **start,int x)
{
struct linkedlist *node;
node=(struct linkedlist *)malloc(sizeof(struct linkedlist));
node->info=x;
if(start==NULL)
    {
    node->link=NULL;
    }
else
    {
    node->link=*start;
    }
*start=node;
}
```

```
void insertatlast(struct linkedlist **start, int x)
{
struct linkedlist *node,*temp;
node=(struct linkedlist *)malloc(sizeof(struct linkedlist));
node->info=x;
node->link=NULL;
if(start==NULL)
    {
    *start=node;
    }
else
    {
    temp=*start;
    while(temp->link!=NULL)
            temp=temp->link;
    temp->link=node;
    }
}
```

```
void insertafteranyposition(struct linkedlist **start, int x, int pos)
{
int i;
struct linkedlist *node,*temp;
node=(struct linkedlist *)malloc(sizeof(struct linkedlist));
node->info=x;
temp=*start;
for(i=o;i<pos;i++)
     temp=temp->link;
node->link=temp->link;
temp->link=node;
}

void deletefromfirst(struct linkedlist **start)
{
struct linkedlist *temp;
if(start==NULL)
     {
     printf("\n List is empty");
     return;
     }
temp=*start;
*start=(*start)->link;
free(temp);
}
```

```
void deletefromlast(struct linkedlist **start)
{
struct linkedlist *temp,*ptr;
if(start==NULL)
    {
    printf("\nList is empty");
    return;
    }

else if((*start)->link==NULL)
    {
    temp=*start;
    *start=NULL;
    free(temp);
    }
else
    {
    ptr=*start;
    temp=(*start)->link;
    while(temp->link!=NULL)
            {
            ptr=temp;
            temp=temp->link;
            }
    ptr->link=NULL;
    free(temp);
    }
}
```

```
void deletefromanyposition(struct linkedlist **start,int pos)
{
struct linkedlist *temp,*ptr;
int i;
if(start==NULL)
      {
      printf("\n List is empty");
      return;
      }
else
      {
      ptr=*start;
      for(i=o;i<pos;i++)
                  {
                  temp=ptr;
                  ptr=ptr->link;
                  }
      temp->link=ptr->link;
      free(ptr);
      }
}

void display(struct linkedlist **start)
{
struct linkedlist *temp;
printf("\n Items of linked list are:\n");
for(temp=*start;temp!=NULL;temp=temp->link)
      {
      printf("-->%d", temp->info);
      }
}
```

```
void reverse(struct linkedlist **start)
{
struct linkedlist *temp,*ptr;
ptr=(*start)->link;
(*start)->link=NULL;
while(ptr!=NULL)
   {
   temp=ptr->link;
   ptr->link=*start;
   *start=ptr;
   ptr=temp;
   }
}
```

Model Question (2008)

- ***Explain the advantages and disadvantages of representing a group of items as an array versus a linear linked list with suitable examples.***

- ***Answer:*** Memory (RAM) is one of the most important resource of a computer. It must be used efficiently.

  When we represent a group of items as an ***array***, then memory for all these items are allocated at compile time in contiguous memory locations and is fixed. This process is also called ***static memory allocation*** and the list so formed is called ***contiguous list***. **[Give structure of contiguous list and the process of insertion and deletion of items]**

  The disadvantages with this approach are:

  - It is actually quite difficult to know the exact size of the array in advance (prior to execution).
  - If the size needed at run time is small than the specified size, then we will have wastage of memory space.
  - If the size needed at run time is greater than the specified size, then we will have shortage of memory space.
  - Insertion and deletion of items is difficult in this representation.

However, the following are the advantages of array representation of a group of items:

- It is quite easy to access any item within the array by just providing the array name followed by its corresponding subscript.
- It is easy to edit/modify any data item within the array.

A ***linear linked list*** is defined as a collection of nodes in which each node has two parts: (i) information part and (ii) link part.

When we represent a group of items as a ***linear linked list***, the items are not contiguously aligned in memory but instead they are linked to each other via pointers. The information part of each node of the ***linear linked list*** contains the value of the items and the link part contains the address of the next node in the ***linear linked list***. The last node of the ***linear linked list*** contains the NULL pointer in its link part indicating the end of the ***linear linked list***. **[Give structure of linked list and the process of insertion and deletion of items]**

The advantages of ***linear linked lists*** are as follows:

- ***Linear Linked lists are dynamic data structures:*** They can grow or shrink during the execution of a program.
- ***Efficient memory utilization:*** Memory is not pre-allocated. Memory is allocated whenever required and deallocated (removed) when it is not needed.
- ***Insertion and deletions are easier and efficient:*** Linear linked lists provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position.

The disadvantages of ***linked lists*** are as follows:

- Access to an arbitrary data item is little bit cumbersome and also time-consuming.
- A ***linked list*** uses more storage than an ***array*** with the same number of items. This is because each item has an additional link field.

TU Exam Question (2066)

- State relative merits and demerits of contiguous list and linked list. Explain the steps involved in inserting and deleting a node in singly linked list.

Homework!!!

Linked Implementation of a Stack (PUSH/POP)

- For the linked list implementation of a stack, a ***linkedliststack*** template is defined using a self-referential structure in linked list format and after this, nodes of the stack are pushed and popped using dynamic memory allocation functions like ***malloc()*** and ***free()***. We consider ***TOS*** (denoting top of stack) as an external pointer that keeps the address of the current node on top of stack. The following code defines the structure for linked list implementation of stack and creates a ***TOS***.

```
struct linkedliststack
        {
        int info;
        struct linkedliststack *link;
        };
struct linkedliststack *TOS;
TOS=NULL;
```

Linked Implementation of a Stack (PUSH/POP)…

- Whenever ***TOS==NULL***, the stack is empty.
- ***Push:*** Insert at beginning (at first).
  - ***C function for push***

    Function call: *push(&TOS, x);*

    Function definition:

```
void push(struct linkedliststack **top,int x)
{
struct linkedliststack *node;
node=(struct linkedliststack *)malloc(sizeof(struct linkedliststack));
node->info=x;
node->link=*top;
*top=node;
}
```

Linked Implementation of a Stack (PUSH/POP)…

- ***Pop:*** Remove at beginning (at first).
  - ***C function for pop***

    Function call: *x=pop(&TOS);*

    Function definition:

```
int pop(struct linkedliststack **top)
{
int item;
struct linkedliststack *temp;
if(*top==NULL)
        {
        printf("\n Stack is empty");
        return -999;
        }
else
        {
        item=(*top)->info;
        temp=*top;
        *top=(*top)->link;
        free(temp);
        return item;
        }
}
```

- ***Display:*** **Do it yourself.**

C code to implement stack as a linked list

```
#include <stdio.h>
struct linkedliststack
    {
    int info;
    struct linkedliststack *link;
    };

void push(struct linkedliststack **,int);
int pop(struct linkedliststack **);
void display(struct linkedliststack **);

void main()
{
struct linkedliststack *TOS;
int x;
char ch='y';
int option;
TOS=NULL;
clrscr();
printf("\n What do you want to do?");
printf("\n1.Push");
printf("\n2.Pop");
printf("\n3.Display");
```

```
while(ch=='y')
    {
    printf("\n Enter your option:");
    scanf("%d", &option);
    switch(option)
            {
            case 1:
                    printf("\n Enter item to insert:");
                    scanf("%d", &x);
                    push(&TOS, x);
                    break;
            case 2:
                    x=pop(&TOS);
                    printf("\n The deleted item is:%d", x);
                    break;
            case 3:
                    display(&TOS);
                    break;
            default:
                    printf("\n WRONG OPTION");
            }
    printf("\n Do you want to continue(y/n)?");
    scanf(" %c", &ch);
    }
getch();
}
```

```
void push(struct linkedliststack **top,int x)
{
struct linkedliststack *node;
node=(struct linkedliststack *)malloc(sizeof(struct linkedliststack));
node->info=x;
node->link=*top;
*top=node;
}

int pop(struct linkedliststack **top)
{
int item;
struct linkedliststack *temp;
if(*top==NULL)
    {
    printf("\n Stack is empty");
    return -999;
    }
else
    {
    item=(*top)->info;
    temp=*top;
    *top=(*top)->link;
    free(temp);
    return item;
    }
}
```

```
void display(struct linkedliststack **top)
{
struct linkedliststack *temp;
if(*top==NULL)
    {
    printf("\n Stack is empty");
    return;
    }
else
    {
    printf("Stack contents are:\n");
    temp=*top;
    printf("top-->");
    while(temp!=NULL)
            {
            printf("\t%d", temp->info);
            printf("\n\t|\n");
            printf("\n\tV\n");
            temp=temp->link;
            }
    }
}
```

Linked Implementation of a Queue (INSERT/REMOVE)

- For the linked list implementation of a queue, a ***linkedlistqueue*** template is defined using a self-referential structure in linked list format and after this, nodes of the queue are inserted i.e. enqueued and removed i.e. dequeued using dynamic memory allocation functions like ***malloc()*** and ***free()***. We consider ***FRONT*** and ***REAR*** as two external pointers that keeps the addresses of the current nodes in the **front** and the **rear** of the queue. The following code defines the structure for linked list implementation of queue and creates ***FRONT*** & ***REAR***.

```
struct linkedlistqueue
        {
        int info;
        struct linkedlistqueue *link;
        };
struct linkedlistqueue *FRONT, *REAR;
FRONT=NULL;
REAR=NULL;
```

Linked Implementation of a Queue (INSERT/REMOVE)…

- ***Enqueue:*** Insert at last and update rear.
  - ***Function Call:*** ***enqueue(&FRONT, &REAR, x);***
  - ***Function Definition:***

```
void enqueue(struct linkedlistqueue **front, struct linkedlistqueue **rear, int x)
{
struct linkedlistqueue *node;
node=(struct linkedlistqueue *)malloc(sizeof(struct linkedlistqueue));
node->info=x;
node->link=NULL;
if(*rear==NULL)
                {
                *front=node;
                *rear=node;
                }
else
                {
                (*rear)->link=node;
                *rear=(*rear)->link;
                }
}
```

Linked Implementation of a Queue (INSERT/REMOVE)…

- ***Dequeue:*** Update front and remove at first.
  - ***Function Call:*** ***x=dequeue(&FRONT, &REAR);***
  - ***Function Definition:***

```
int dequeue(struct linkedlistqueue **front, struct linkedlistqueue **rear)
{
int item;
struct linkedlistqueue *temp;
if(*front==NULL)
                {
                printf("\n Queue is empty");
                return -999;
                }
else
                {
                temp=*front;
                item=temp->info;
                *front=(*front)->link;
                                if(*front==NULL)
                                *rear=NULL;
                free(temp);
                return item;
                }
}
```

C code to implement queue as a linked list

```
#include <stdio.h>
void enqueue(struct linkedlistqueue **, struct linkedlistqueue **, int);
int dequeue(struct linkedlistqueue **, struct linkedlistqueue **);
void display(struct linkedlistqueue **);

struct linkedlistqueue
    {
    int info;
    struct linkedlistqueue *link;
    };
```

```
void main()
{
struct linkedlistqueue *FRONT, *REAR;
int x;
char ch='y';
int option;
FRONT=NULL;
REAR=NULL;
clrscr();
printf("\n What do you want to do?");
printf("\n1.Enqueue");
printf("\n2.Dequeue");
printf("\n3.Display");
while(ch=='y')
    {
    printf("\n Enter your option:");
    scanf("%d", &option);
    switch(option)
            {
            case 1:
               printf("\n Enter item :");
               scanf("%d", &x);
                enqueue(&FRONT, &REAR, x);
                break;

        case 2:
                x=dequeue(&FRONT,&REAR);
                printf("\nThe deleted item is:%d", x);
                break;
        case 3:
                display(&FRONT);
                break;
        default:
                printf("\n WRONG OPTION");
                }
        printf("\n Do you want to continue(y/n)?");
        scanf(" %c", &ch);
        }
getch();
}
```

```
void enqueue(struct linkedlistqueue **front, struct linkedlistqueue **rear, int x)
{
struct linkedlistqueue *node;
node=(struct linkedlistqueue *)malloc(sizeof(struct linkedlistqueue));
node->info=x;
node->link=NULL;
if(*rear==NULL)
    {
    *front=node;
    *rear=node;
    }
else
    {
    (*rear)->link=node;
    *rear=(*rear)->link;
    }
}
```

```
int dequeue(struct linkedlistqueue **front, struct linkedlistqueue **rear)
{
int item;
struct linkedlistqueue *temp;
if(*front==NULL)
    {
    printf("\n Queue is empty");
    return -999;
    }
else
    {
    temp=*front;
    item=temp->info;
    *front=(*front)->link;
            if(*front==NULL)
                    *rear=NULL;
    free(temp);
    return item;
    }

}
```

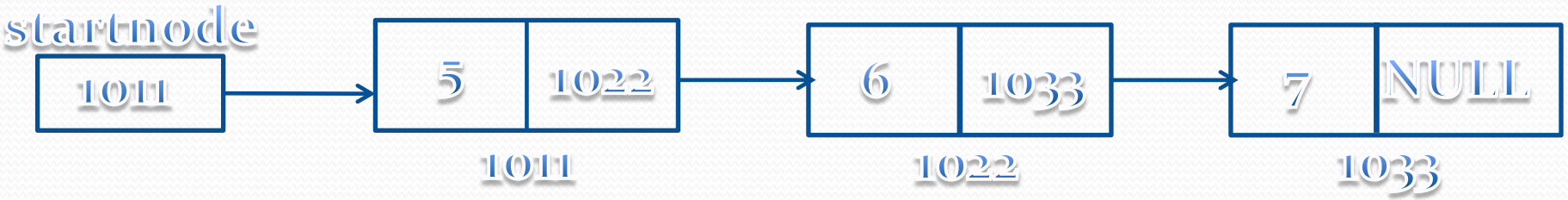
```
void display(struct linkedlistqueue **front)
{
struct linkedlistqueue *temp;
if(*front==NULL)
    {
    printf("\n Queue is empty");
    return;
    }
else
    {
    printf("Queue contents are:\n");
    temp=*front;
    printf("front");
    while(temp!=NULL)
            {
            printf("->%d->",temp->info);
            temp=temp->link;
            }
    printf("rear");
    }
}
```

Types of Linked Lists

- Linked lists are of following four types:
  1. ***Singly Linked List***
  2. ***Circular Linked List***
  3. ***Doubly Linked List***
  4. ***Circular Doubly Linked List***

1. Singly Linked List

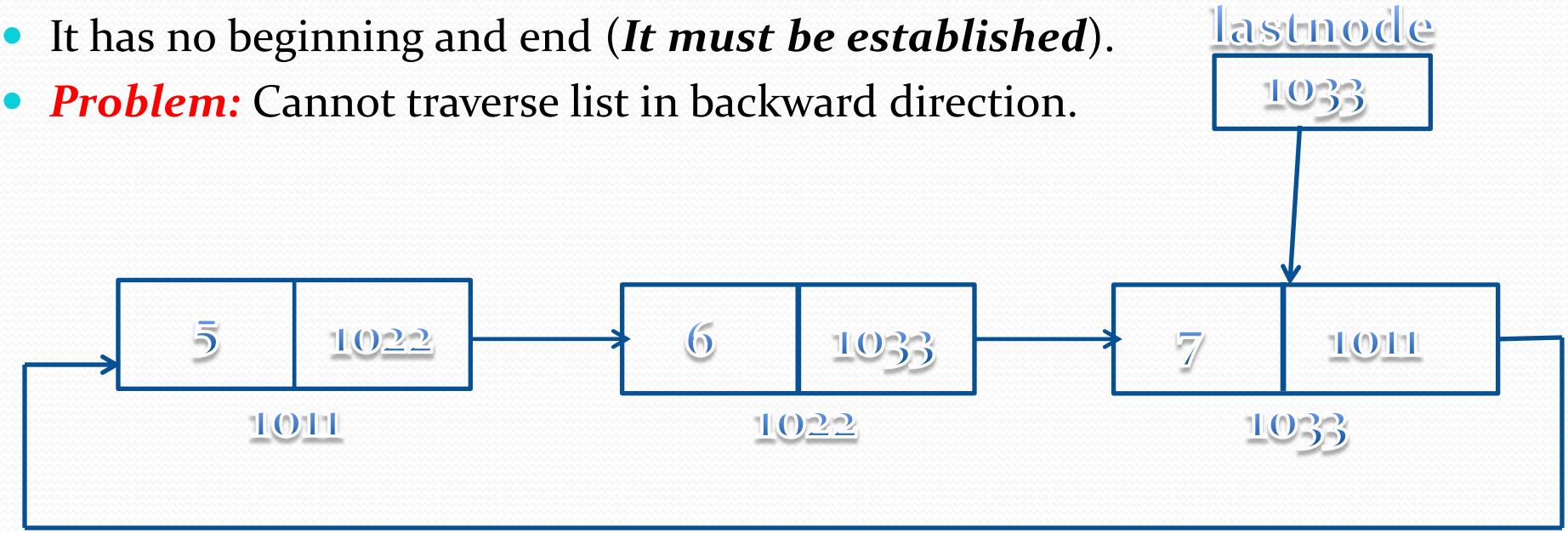
- A ***singly linked list*** is one in which all nodes are linked together in some sequential manner. Hence, it is also called ***linear linked list***.
- It has a beginning and an end.



- ***Problem:*** We cannot access the predecessor node from the current node. This problem can be overcome by using ***doubly linked list***.

2. Circular Linked List

- A ***circular linked list*** is just a ***singly linked list*** in which the ***link*** field of the last node contains the address of the first node of the list. That is, the ***link*** field of the last node does not point to NULL, rather it points back to the beginning of the linked list.
- It has no beginning and end (***It must be established***).
- ***Problem:*** Cannot traverse list in backward direction.



Implementation of Circular Linked List

- A circular linked list does not have a natural “first” or “last” node.
- We must therefore establish a first and last node by convention.
- One useful convention is to let an external pointer (say ***“lastnode”***) to the circular linked list to point to the last node, and to allow the following node to be the first node.

C code for Circular Linked List

```
#include <stdio.h>
struct circularlinkedlist
    {
    int info;
    struct circularlinkedlist *link;
    };

void insertatfirst(struct circularlinkedlist **,int);
void insertatlast(struct circularlinkedlist **, int);
void insertafteranyposition(struct circularlinkedlist **, int, int);
void deletefromfirst(struct circularlinkedlist **);
void deletefromlast(struct circularlinkedlist **);
void deletefromanyposition(struct circularlinkedlist **,int);
void display(struct circularlinkedlist **);
```

```
void main()
{
struct circularlinkedlist *lastnode;
int item;
int position;
int option;
char ch='y';
lastnode=NULL;

clrscr();
printf("What do you want to do?");
printf("\n1.Insert node at first");
printf("\n2.Insert node at last");
printf("\n3.Insert node after any position");
printf("\n4.Delete node at first");
printf("\n5.Delete node at last");
printf("\n6.Delete node at any position");
printf("\n7.Display the circular linked list");

while(ch=='y')
    {
    printf("\n Enter your option:");
    scanf("%d", &option);
```

```
switch(option)
{
case 1:
    printf("\n Enter the information part:");
    scanf("%d", &item);
    insertatfirst(&lastnode, item);
    break;
case 2:
    printf("\n Enter the information part:");
    scanf("%d", &item);
    insertatlast(&lastnode, item);
    break;
case 3:
    printf("\n Enter the information part:");
    scanf("%d", &item);
    printf("\n Enter the position to insert after:");
    scanf("%d", &position);
    insertafteranyposition(&lastnode,item,position);
    break;
case 4:
    deletefromfirst(&lastnode);
    break;
case 5:
    deletefromlast(&lastnode);
    break;
case 6:
    printf("\nEnter the position to delete:");
    scanf("%d",&position);
    deletefromanyposition(&lastnode,position);
    break;
case 7:
    display(&lastnode);
    break;
default:
    printf("\n WRONG OPTION");
    }
    printf("\n Do you want to continue(y/n)?");
    scanf(" %c", &ch);
  }
getch();
}
```

```
void insertatfirst(struct circularlinkedlist **last,int x)
{
struct circularlinkedlist *node;
node=(struct circularlinkedlist *)malloc(sizeof(struct circularlinkedlist));
node->info=x;
if(*last==NULL)
    {
    *last=node;
    node->link=*last;
    }
else
    {
    node->link=(*last)->link;
    (*last)->link=node;
    }
}
```

```
void insertatlast(struct circularlinkedlist **last, int x)
{
struct circularlinkedlist *node,*temp;
node=(struct circularlinkedlist *)malloc(sizeof(struct circularlinkedlist));
node->info=x;
node->link=NULL;
if(*last==NULL)
    {
    *last=node;
    node->link=*last;
    }
else
    {
    node->link=(*last)->link;
    (*last)->link=node;
    *last=node;
    }
}
```

```
void insertafteranyposition(struct circularlinkedlist **last, int x, int pos)
{
int i;
struct circularlinkedlist *node,*temp;
node=(struct circularlinkedlist *)malloc(sizeof(struct circularlinkedlist));
node->info=x;
temp=(*last)->link;
for(i=o;i<pos;i++)
    temp=temp->link;
node->link=temp->link;
temp->link=node;
}
```

```
void deletefromfirst(struct circularlinkedlist **last)
{
struct circularlinkedlist *temp;
if(*last==NULL)
    {
    printf("\n List is empty");
    return;
    }
else if((*last)->link==*last)
    {
    temp=*last;
    *last=NULL;
    free(temp);
    }
else
    {
    temp=(*last)->link;
    (*last)->link=temp->link;
    free(temp);
    }
}
```

```
void deletefromlast(struct circularlinkedlist **last)
{
struct circularlinkedlist *temp,*ptr;
if(*last==NULL)
    {
    printf("\n List is empty");
    return;
    }

else if((*last)->link==*last)
    {
    temp=*last;
    *last=NULL;
    free(temp);
    }
else
    {
    temp=*last;
    ptr=(*last)->link;
    while(ptr->link!=*last)
            ptr=ptr->link;
    ptr->link=(*last)->link;
    *last=ptr;
    free(temp);
    }
}
```

```
void deletefromanyposition(struct circularlinkedlist **last,int pos)
{
struct circularlinkedlist *temp,*ptr;
int i;
ptr=(*last)->link;
for(i=o;i<pos;i++)
      {
      temp=ptr;
      ptr=ptr->link;
      }
temp->link=ptr->link;
free(ptr);
}

void display(struct circularlinkedlist **last)
{
struct circularlinkedlist *temp;
if(*last==NULL)
      {
      printf("\n List is empty");
      return;
      }
printf("\nItems of linked list are:\n");
for(temp=(*last)->link;temp!=*last;temp=temp->link)
      {
      printf("%d-->",temp->info);
      }
      printf("%d",(*last)->info);
}
```

Stack as a Circular List (PUSH/POP)

- For the circular linked list implementation of a stack, a ***cllstack*** template is defined using a self-referential structure in linked list format and after this, nodes of the stack are pushed and popped using dynamic memory allocation functions like ***malloc()*** and ***free()***. We consider ***lastnodestack*** to be an external pointer that keeps the address of the last node of the stack and the first node of the circular linked list to be the top of the stack. The following code defines the structure for linked list implementation of stack and creates a ***lastnodestack***.

```
struct cllstack
        {
        int info;
        struct cllstack *link;
        };
struct cllstack *lastnodestack;
lastnodestack=NULL;
```

Stack as a Circular List (PUSH/POP)…

- Whenever ***lastnodestack==NULL***, the stack is empty.
- ***Push:*** Insert at first in circular list.
  - ***C function for push***

    Function call: *push(&lastnodestack, x);*

    Function definition:

```
void push(struct cllstack **last, int x)
{
struct cllstack *node;
node=(struct cllstack *)malloc(sizeof(struct cllstack));
node->info=x;
if(*last==NULL)
                *last=node;
else
                node->link=(*last)->link;
(*last)->link=node;
}
```

Stack as a Circular List (PUSH/POP)…

- ***Pop:*** Remove at beginning of circular list (at first).
  - ***C function for pop***

    Function call: *x=pop(&lastnodestack);*

    Function definition:

```
int pop(struct cllstack **last)
{
int item;
struct cllstack *temp;
if(*last==NULL)
                {
                printf("\n Stack is empty");
                return -999;
                }
else if(*last==(*last)->link)
                {
                item=(*last)->info;
                temp=*last;
                *last=NULL;
                free(temp);
                return item;
                }
else
                {
                temp=(*last)->link;
                item=temp->info;
                (*last)->link=temp->link;
                free(temp);
                return item;
                }
}
```

COMPLETE IMPLEMENTATION

```
#include <stdio.h>
struct cllstack
      {
      int info;
      struct cllstack *link;
      };

void push(struct cllstack **,int);
int pop(struct cllstack **);
void display(struct cllstack **);

void main()
{
struct cllstack *lastnodestack;
int x;
char ch='y';
int option;
lastnodestack=NULL;
clrscr();
printf("\nWhat do you want to do?");
printf("\n1.Push");
printf("\n2.Pop");
printf("\n3.Display");
while(ch=='y')
      {
      printf("\n Enter your option:");
      scanf("%d", &option);
```

```
switch(option)
            {
            case 1:
                        printf("\nEnter item to insert:");
                        scanf("%d",&x);
                        push(&lastnodestack,x);
                        break;
            case 2:
                        x=pop(&lastnodestack);
                        printf("\nThe deleted item is:%d",x);
                        break;
            case 3:
                        display(&lastnodestack);
                        break;
            default:
                        printf("\n WRONG OPTION");
            }
    printf("\n Do you want to continue(y/n)?");
    scanf(" %c", &ch);
    }
getch();
}
```

```
void push(struct cllstack **last, int x)
{
struct cllstack *node;
node=(struct cllstack *)malloc(sizeof(struct cllstack));
node->info=x;
if(*last==NULL)
   *last=node;
else
   node->link=(*last)->link;
(*last)->link=node;
}
```

```
int pop(struct cllstack **last)
{
int item;
struct cllstack *temp;
if(*last==NULL)
    {
    printf("\n Stack is empty");
    return -999;
    }
else if(*last==(*last)->link)
    {
    item=(*last)->info;
    temp=*last;
    *last=NULL;
    free(temp);
    return item;
    }
else
    {
    temp=(*last)->link;
    item=temp->info;
    (*last)->link=temp->link;
    free(temp);
    return item;
    }
}
```

```
void display(struct cllstack **last)
{
struct cllstack *temp;
if(*last==NULL)
    {
    printf("\n Stack is empty");
    return;
    }
else
    {
    printf("Stack contents are:\n");
    temp=(*last)->link;
    printf("Top-->");
    while(temp!=*last)
            {
            printf("\t%d\n", temp->info);
            temp=temp->link;
            }
            printf("%d",(*last)->info);
    }
}
```

Queue as a Circular List (INSERT/REMOVE)

- For the circular linked list implementation of a queue, a ***cllqueue*** template is defined using a self-referential structure in linked list format and after this, nodes of the queue are inserted i.e. enqueued and removed i.e. dequeued using dynamic memory allocation functions like ***malloc()*** and ***free()***. We consider ***lastnodequeue*** to be an external pointer that keeps the address of the recently inserted node onto the queue. The following code defines the structure for circular linked list implementation of queue and creates ***lastnodequeue***.

```
struct cllqueue
        {
        int info;
        struct cllqueue *link;
        };
struct cllqueue *lastnodequeue;
lastnodequeue=NULL;
```

Queue as a Circular List (INSERT/REMOVE)…

- **Whenever lastnodequeue==NULL, queue is empty.**
- ***Enqueue:*** Insert at last and update last.
  - ***Function Call:*** ***enqueue(&lastnodequeue, x);***
  - ***Function Definition:***

```
void enqueue(struct cllqueue **last, int x)
{
struct cllqueue *node;
node=(struct cllqueue *)malloc(sizeof(struct cllqueue));
node->info=x;
        if(*last==NULL)
                *last=node;
        else
                node->link=(*last)->link;
(*last)->link=node;
*last=node;
}
```

Queue as a Circular List (INSERT/REMOVE)…

- ***Dequeue:*** Remove from first.
  - ***Function Call:*** ***x=dequeue(&lastnodequeue);***
  - ***Function Definition:***

```
int dequeue(struct cllqueue **last)
{
int item;
struct cllqueue *temp;
if(*last==NULL)
                {
                printf("\n Queue is empty");
                return -999;
                }
else if(*last==(*last)->link)
                {
                item=(*last)->info;
                temp=*last;
                *last=NULL;
                free(temp);
                return item;
                }
else
                {
                temp=(*last)->link;
                item=temp->info;
                (*last)->link=temp->link;
                free(temp);
                return item;
                }
}
```

COMPLETE IMPLEMENTATION

```
#include <stdio.h>
struct cllqueue
    {
    int info;
    struct cllqueue *link;
    };
void enqueue(struct cllqueue **, int);
int dequeue(struct cllqueue **);
void display(struct cllqueue **);
void main()
{
struct cllqueue *lastnodequeue;
int x;
char ch='y';
int option;
lastnodequeue=NULL;
clrscr();
printf("\n What do you want to do?");
printf("\n1.Insert");
printf("\n2.Remove");
printf("\n3.Display");
while(ch=='y')
    {
    printf("\n Enter your option:");
    scanf("%d", &option);
```

```
switch(option)
        {
        case 1:
                printf("\n Enter item to insert:");
                scanf("%d", &x);
                enqueue(&lastnodequeue,x);
                break;
        case 2:
                x=dequeue(&lastnodequeue);
                printf("\n The deleted item is:%d",x);
                break;
        case 3:
                display(&lastnodequeue);
                break;
        default:
                printf("\n WRONG OPTION");
        }
    printf("\n Do you want to continue(y/n)?");
    scanf(" %c", &ch);
    }
getch();
}
```

```
void enqueue(struct cllqueue **last, int x)
{
struct cllqueue *node;
node=(struct cllqueue *)malloc(sizeof(struct cllqueue));
node->info=x;
   if(*last==NULL)
         *last=node;
   else
         node->link=(*last)->link;
(*last)->link=node;
*last=node;
}
```

```
int dequeue(struct cllqueue **last)
{
int item;
struct cllqueue *temp;
if(*last==NULL)
    {
    printf("\n Queue is empty");
    return -999;
    }
else if(*last==(*last)->link)
    {
    item=(*last)->info;
    temp=*last;
    *last=NULL;
    free(temp);
    return item;
    }
else
    {
    temp=(*last)->link;
    item=temp->info;
    (*last)->link=temp->link;
    free(temp);
    return item;
    }
}
```

```
void display(struct cllqueue **last)
{
struct cllqueue *temp;
if(*last==NULL)
    {
    printf("\n Queue is empty");
    return;
    }
else
    {
    printf("Queue contents are:\n");
    temp=(*last)->link;
    while(temp!=*last)
            {
            printf("%d->",temp->info);
            temp=temp->link;
            }
            printf("%d",(*last)->info);
    }
}
```

3. Doubly Linked List

- A ***doubly linked list*** is one in which all nodes are linked together by multiple links which help in accessing both the successor node (i.e. next node) and the predecessor node (i.e. previous node) from any arbitrary node within the list.
- Therefore each node in a ***doubly linked list*** requires two pointers: one to the left node (previous node) and other to the right node (next node). This helps to traverse the list in the forward direction and also to the backward direction (bi-directional traversing).

Fig: A node in doubly linked list

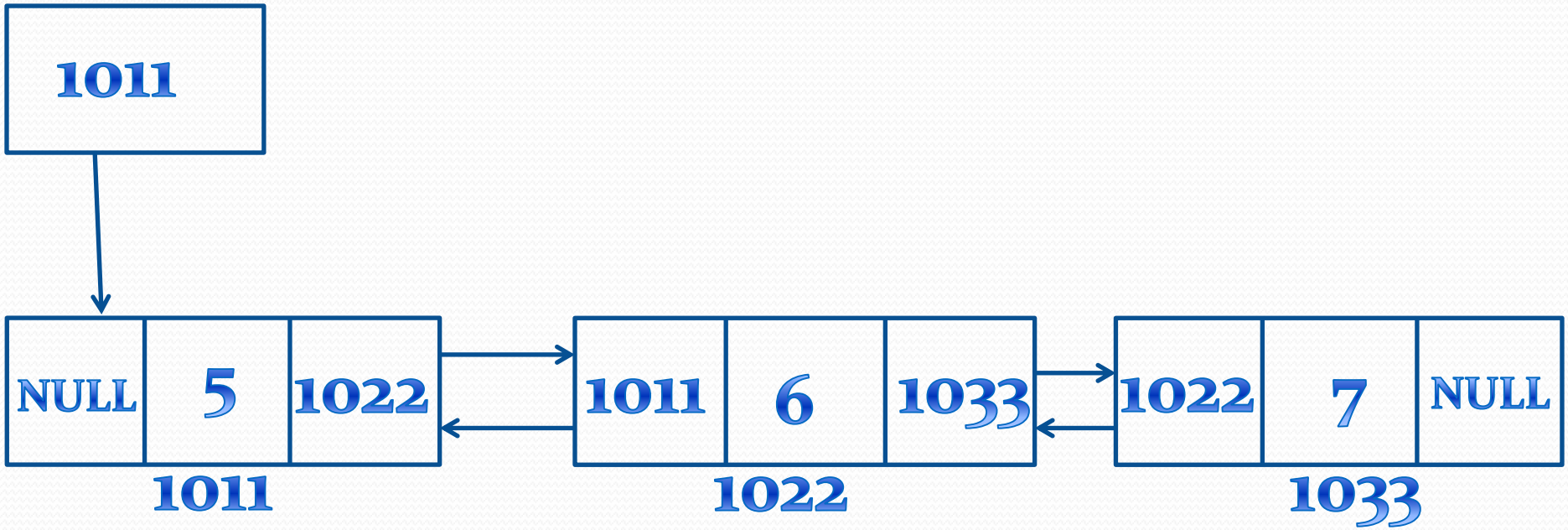


Fig: A doubly linked list

Two types of implementation

- ***Array Implementation***

```
struct doublylinkedlist
{
int info;
int prev;
int next;
};
struct doublylinkedlist node[MAXSIZE];
```

- ***Dynamic Implementation***

```
struct doublylinkedlist
{
int info;
struct doublylinkedlist *prevlink;
struct doublylinkedlist *nextlink;
};
struct doublylinkedlist *node;
```

Dynamic Implementation of Doubly Linked List

```
#include <stdio.h>
struct doublylinkedlist
    {
    int info;
    struct doublylinkedlist *prevlink;
    struct doublylinkedlist *nextlink;
    };

void insertatfirst(struct doublylinkedlist **,int);
void insertatlast(struct doublylinkedlist **, int);
void insertafteranyposition(struct doublylinkedlist **, int, int);
void deletefromfirst(struct doublylinkedlist **);
void deletefromlast(struct doublylinkedlist **);
void deletefromanyposition(struct doublylinkedlist **,int);
void display(struct doublylinkedlist **);
```

```
void main()
{
struct doublylinkedlist *startnode;
int item;
int position;
int option;
char ch='y';
startnode=NULL;
clrscr();
printf("What do you want to do?");
printf("\n1.Insert node at first");
printf("\n2.Insert node at last");
printf("\n3.Insert node at any position");
printf("\n4.Delete node at first");
printf("\n5.Delete node at last");
printf("\n6.Delete node at any position");
printf("\n7.Display the doubly linked list");
while(ch=='y')
    {
    printf("\n Enter your option:");
    scanf("%d", &option);
```

```
switch(option)
 {
 case 1:
      printf("\n Enter the information part:");
      scanf("%d", &item);
      insertatfirst(&startnode, item);
      break;

case 2:
      printf("\n Enter the information part:");
      scanf("%d", &item);
      insertatlast(&startnode, item);
      break;

case 3:
      printf("\n Enter the information part:");
      scanf("%d", &item);
      printf("\n Enter the position to insert after:");
      scanf("%d",&position);
      insertafteranyposition(&startnode, item, position);
      break;
case 4:
      deletefromfirst(&startnode);
      break;
case 5:
      deletefromlast(&startnode);
      break;

case 6:
      printf("\n Enter the position to delete:");
      scanf("%d", &position);
      deletefromanyposition(&startnode, position);
      break;

case 7:
      display(&startnode);
      break;

default:
         printf("\n WRONG OPTION");
        }
      printf("\n Do you want to continue(y/n)?");
      scanf(" %c", &ch);
      }
getch();
}
```

```
void insertatfirst(struct doublylinkedlist **start, int x)
{
struct doublylinkedlist *node;
node=(struct doublylinkedlist *)malloc(sizeof(struct doublylinkedlist));
node->info=x;
if(*start==NULL)
    {
    node->prevlink=NULL;
    node->nextlink=NULL;
    *start=node;
    }
else
    {
    node->prevlink=NULL;
    node->nextlink=*start;
    (*start)->prevlink=node;
    *start=node;
    }
}
```

```
void insertatlast(struct doublylinkedlist **start, int x)
{
struct doublylinkedlist *node, *temp;
node=(struct doublylinkedlist *)malloc(sizeof(struct doublylinkedlist));
node->info=x;
node->nextlink=NULL;
if(*start==NULL)
    {
    node->prevlink=NULL;
    *start=node;
    }
else
    {
    temp=*start;
    while(temp->nextlink!=NULL)
            temp=temp->nextlink;
    temp->nextlink=node;
    node->prevlink=temp;
    }
}
```

```
void insertafteranyposition(struct doublylinkedlist **start, int x, int pos)
{
int i;
struct doublylinkedlist *node,*temp;
node=(struct doublylinkedlist *)malloc(sizeof(struct doublylinkedlist));
node->info=x;
temp=*start;
for(i=o;i<pos;i++)
    temp=temp->nextlink;
temp->nextlink->prevlink=node;
node->nextlink=temp->nextlink;
node->prevlink=temp;
temp->nextlink=node;
}

void deletefromfirst(struct doublylinkedlist **start)
{
struct doublylinkedlist *temp;
if(*start==NULL)
    {
    printf("\n List is empty");
    return;
    }
temp=*start;
*start=(*start)->nextlink;
(*start)->prevlink=NULL;
free(temp);
}
```

```
void deletefromlast(struct doublylinkedlist **start)
{
struct doublylinkedlist *temp,*ptr;
if(*start==NULL)
    {
    printf("\n List is empty");
    return;
    }
else if((*start)->nextlink==NULL)
    {
    temp=*start;
    *start=NULL;
    free(temp);
    }
else
    {
    ptr=*start;
    temp=(*start)->nextlink;
    while(temp->nextlink!=NULL)
            {
            ptr=temp;
            temp=temp->nextlink;
            }
    ptr->nextlink=NULL;
    free(temp);
    }
}
```

```
void deletefromanyposition(struct doublylinkedlist **start,int pos)
{
struct doublylinkedlist *temp,*ptr;
int i;
if(*start==NULL)
      {
      printf("\n List is empty");
      return;
      }
else if((*start)->nextlink==NULL)
      {
      temp=*start;
      *start=NULL;
      free(temp);
      }
else
      {
      ptr=*start;
      for(i=o;i<pos;i++)
                  {
                  temp=ptr;
                  ptr=ptr->nextlink;
                  }
      ptr->nextlink->prevlink=temp;
      temp->nextlink=ptr->nextlink;
      free(ptr);
      }
}
```

```
void display(struct doublylinkedlist **start)
{
struct doublylinkedlist *temp;
printf("\n Items of linked list are:\n");
for(temp=*start;temp!=NULL;temp=temp->nextlink)
    {
    printf("%d-->",temp->info);
    }
    printf("End");
}
```

4. Circular Doubly Linked List

- A ***circular doubly linked list*** is one in which every node has both the successor pointer and the predecessor pointer in circular manner.

TU Exam Question (2066)

- Explain CLL, DLL, DCLL (Circular, Doubly, Doubly Circular Linked List).