Linked List

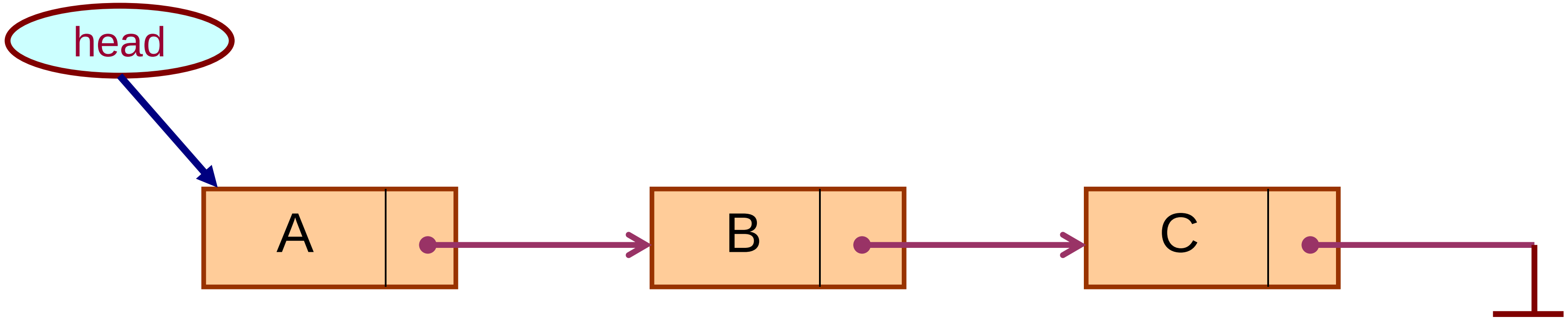
ICT 4303

Why Linked Lists?

- Advantages of Arrays:
  - Data access is faster
  - Simple
- Disadvantages:
  - Size of the array is fixed.
  - Array items are stored contiguously.
  - Insertion and deletion operations involve tedious job of shifting the elements with respect to the index of the array.

Introduction

- A linked list is a data structure which can change during execution.
  - Successive elements are connected by pointers.
  - Last element points to `NULL`.
  - It can grow or shrink in size during execution of a program.
  - It can be made just as long as required.
  - It does not waste memory space.



Linked List

- Keeping track of a linked list:
  - Must know the pointer to the first element of the list (called *start, head*, etc.).

- Linked lists provide flexibility in allowing the items to be rearranged efficiently.
  - Insert an element.
  - Delete an element.

Illustration: Insertion

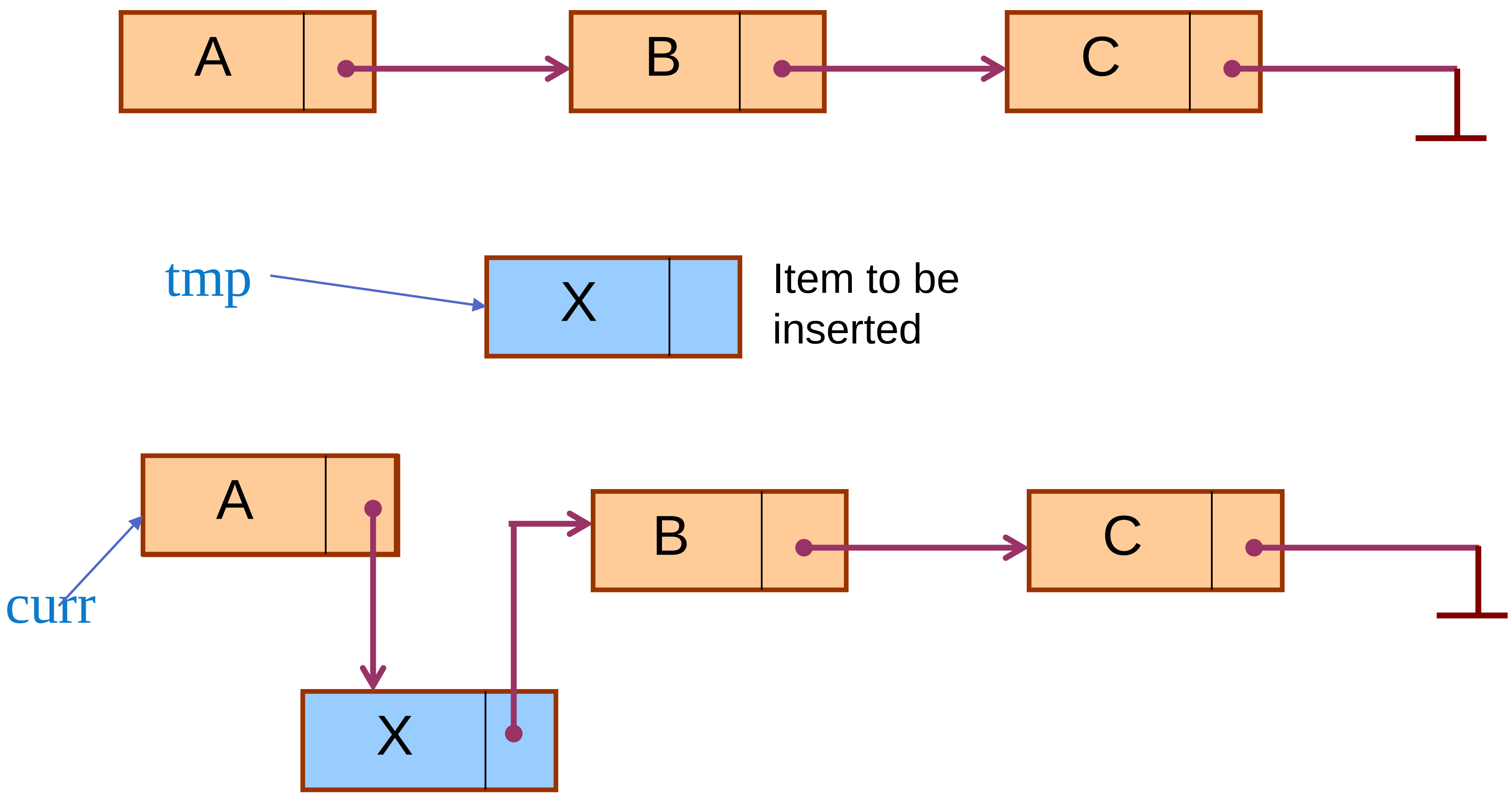
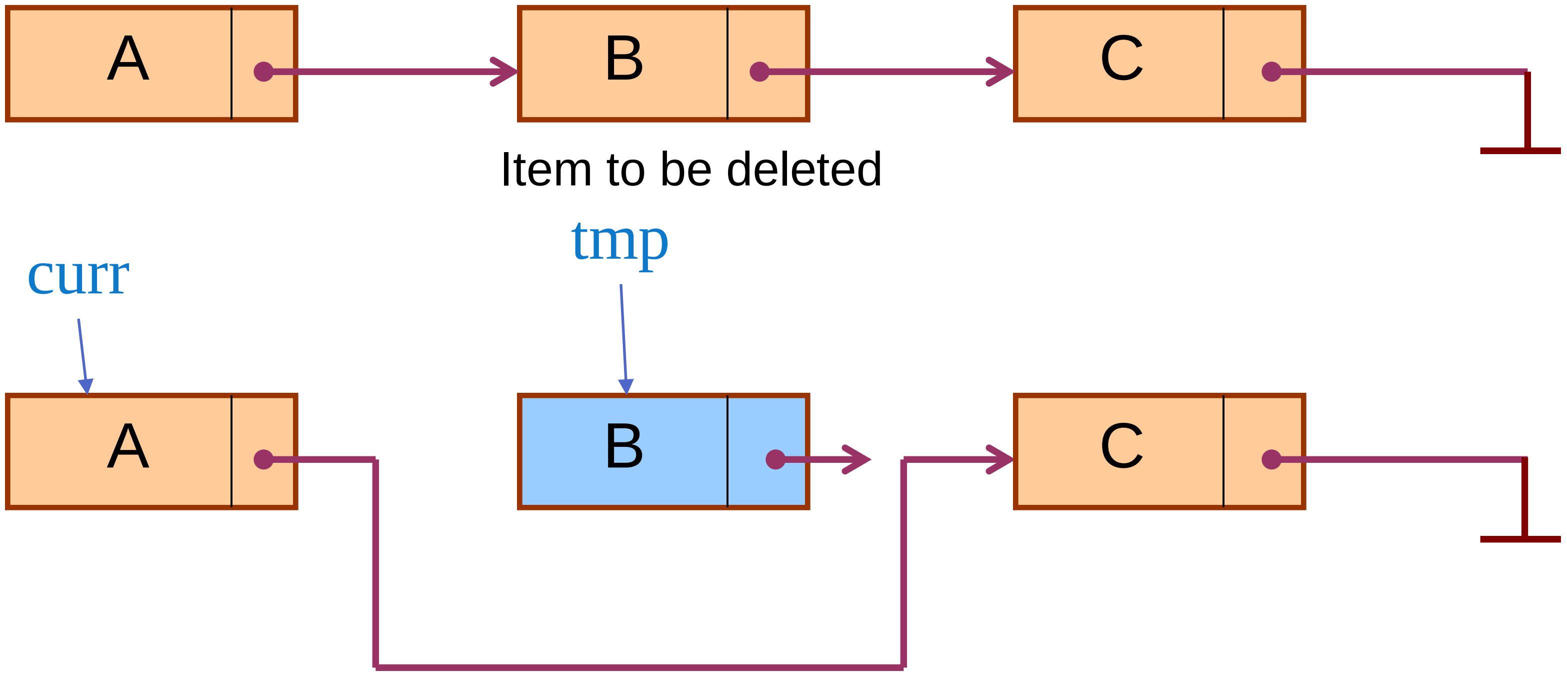


Illustration: Deletion



In essence ...

- For insertion:
  - A record is created holding the new item.
  - The next pointer of the new record is set to link it to the item which is to follow it in the list.
  - The next pointer of the item which is to precede it must be modified to point to the new item.
- For deletion:
  - The next pointer of the item immediately preceding the one to be deleted is altered and made to point to the item following the deleted item.

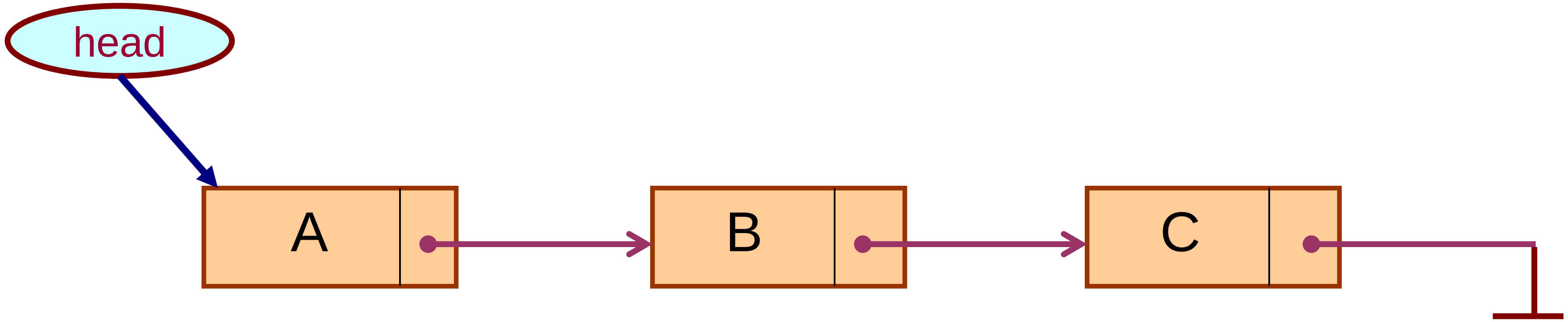
Array versus Linked Lists

- Arrays are suitable for:
  - Inserting/deleting an element at the end.
  - Randomly accessing any element.
  - Searching the list for a particular value.
- Linked lists are suitable for:
  - Inserting an element.
  - Deleting an element.
  - Applications where sequential access is required.
  - In situations where the number of elements cannot be predicted beforehand.

Types of Lists

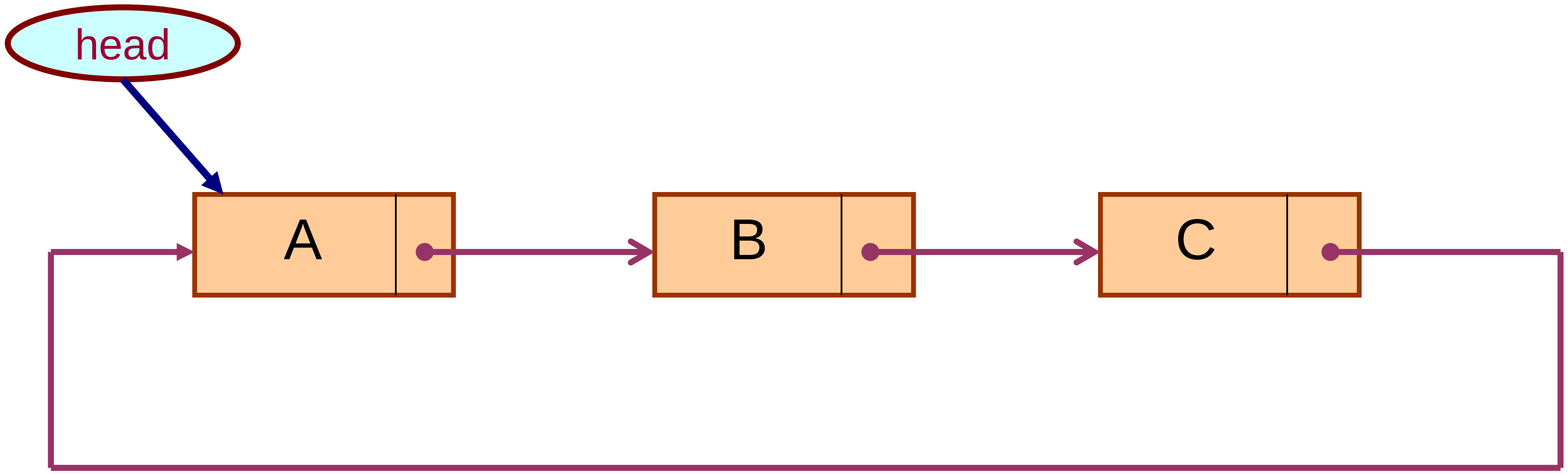
- Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.

  - Linear singly-linked list (or simply linear list)
    - One we have discussed so far.



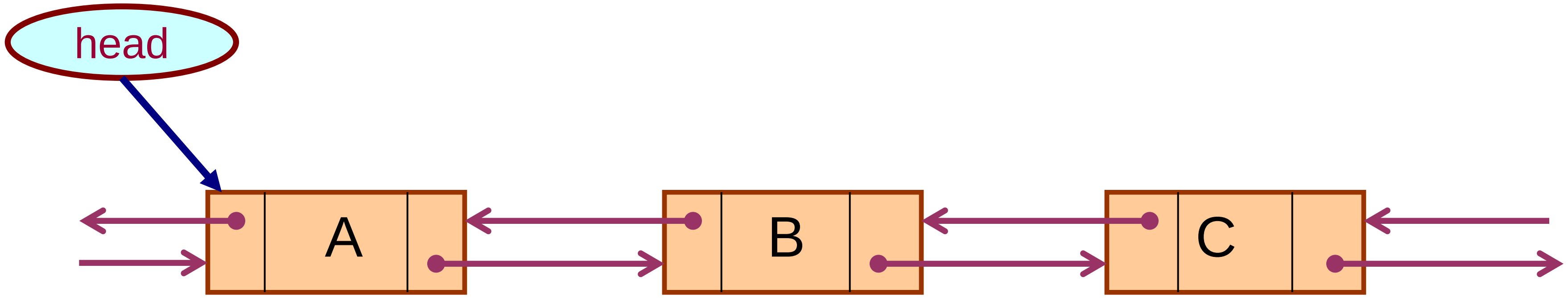
Types of Lists

- Circular linked list
  - The pointer from the last element in the list points back to the first element.



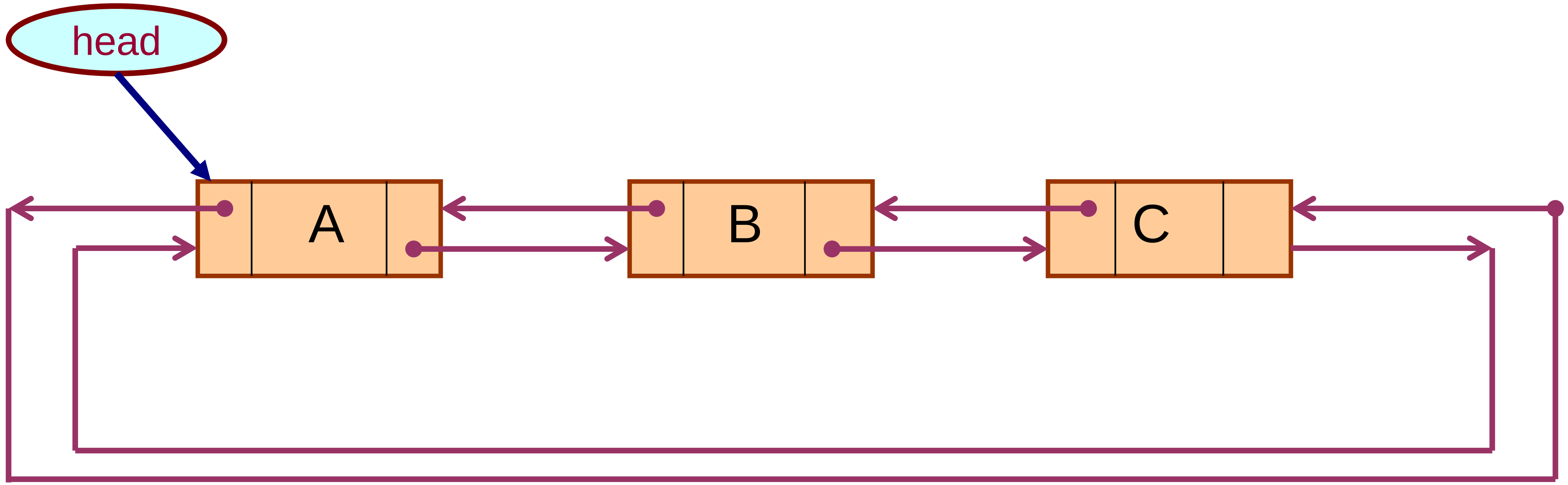
Types of Lists

- Doubly linked list
  - Pointers exist between adjacent nodes in both directions.
  - The list can be traversed either forward or backward.



Types of Lists

- Circular Doubly linked list
  - Pointers exist between adjacent nodes in both directions.
  - The list can be traversed either forward or backward.



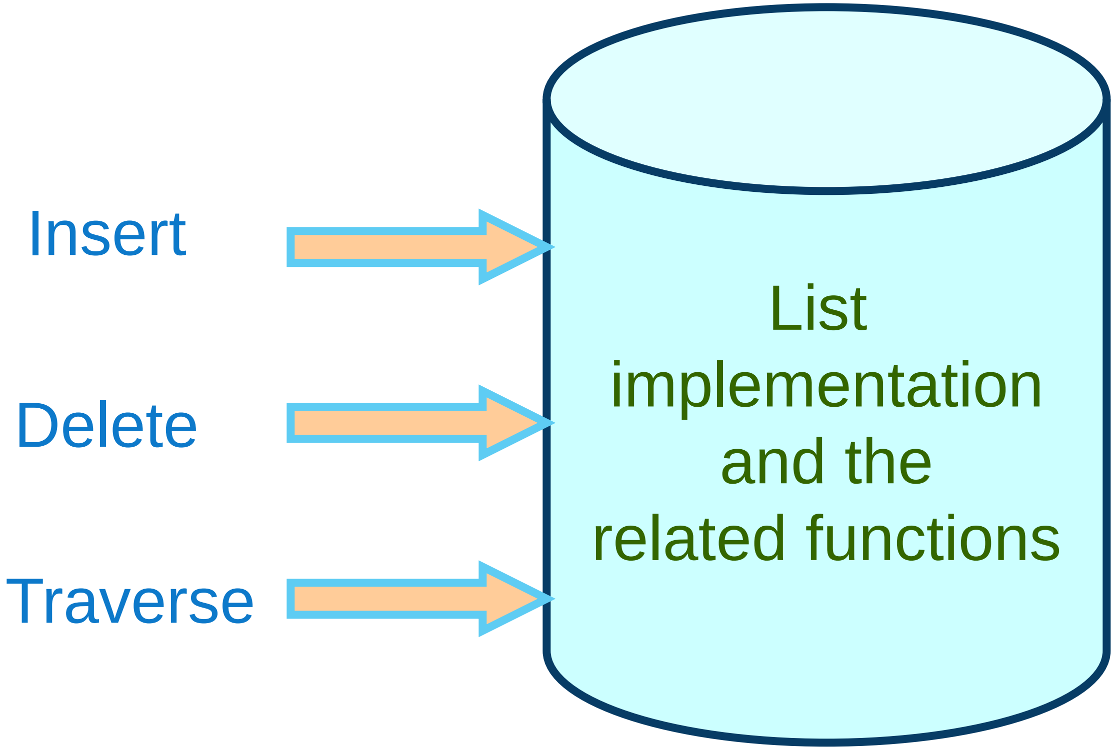
Basic Operations on a List

- Creating a list
- Traversing the list
- Inserting an item in the list
- Deleting an item from the list
- Concatenating two lists into one.
- Merging

List is an Abstract Data Type

- What is an abstract data type?
  - It is a data type defined by the user.
  - Typically, more complex than simple data types like *int, float,* etc.
- Why abstract?
  - Because details of the implementation are hidden.
  - When you do some operation on the list, say insert an element, you just call a function.
  - Details of how the list is implemented or how the insert function is written is no longer required.

Conceptual Idea



Linked List Implementation

We implement linked list using 2 classes:

1. node class
2. Linked List class

Node Class

```
class node{
        public:
                int data;
                node *next;
                node(){
                        next = NULL;
                }
 };
```

Linked List Class

```
class LinkedList{
      node *head;
      public:
/* Member functions to perform different operations on Linked
List */
             void insert_at_beginning(int data);
             void insert_at_end(int data);
             void insert_at_given_position(int data, int p);
             void delete_at_beginning();
             void delete_at_end();
             void delete_at_given_position(int p);
             void print();
};
```

Insert at Beginning

```
void insert_at_beginning(int data){
        node *temp = new node();
        temp->data = data;
        temp->next = head;
        head = temp;
    }
```

Insert at Beginning

1. First, initialize a new node **temp** with value **data**.
2. Set **temp->next** to the address of the first node of the linked list, that is **head**.
3. Make **temp** the new **head.**

Insert at End

```
void insert_at_end(int data){
        node *temp = new node();
        temp->data = data;
        if (head == NULL){
        /* if linked list is empty, that is head == NULL, Make temp the new head */
                head = temp;
                }
        else{
        /* if linked list is not empty, go to the last node of the linked list*/
                node *ptr = head;
        /* the loop sets ptr to last node of the linked list */
        while (ptr->next != NULL){
                ptr = ptr->next;
                }
        /*ptr now points to the last node, store temp address in the next of ptr*/
                ptr->next = temp;
        }
}
```

Insert at End

1. Initialize a new node **temp** with value **data**.
2. If the linked list is empty, simply make **temp** the new **head**.
3. If the linked list is not empty, move to the last node of the linked list and set the next of the last node to **temp**.

Insert at a given Position

```
void insert_at_given_position(int data, int p){
        node *temp = new node();
        temp->data = data;
        if (p == 0){
                // if p==0 then insert temp at beginning
                temp->next = head;
                head = temp;
        }
        else{
                node *ptr = head;
                // the loop sets ptr to (p-1)th node
        while(p>1) {
                ptr = ptr->next;
                --p;
                }
                // ptr now points to (p-1)th node
        temp->next = ptr->next;
        ptr->next = temp;
        }
}
```

Delete at Beginning

```
void delete_at_beginning(){
        if (head == NULL){
                cout<<"List is Empty"<<endl;
                }
        else{
                cout<<"Element Deleted:"<<head->data <<endl;
        // if linked list is not empty, store address of first node in temp
                node *temp = head;
        // set second node as the new head of the linked list
                head = head->next;
                delete(temp);
                }
        }
```

Delete at End

```
void delete_at_end(){
	if (head == NULL){
		cout<<"List is Empty"<<endl;
		}
	else if (head->next == NULL){
		cout<<"Element Deleted: "<<head->data<<endl;
		delete(head);
		head = NULL;
		}
	else{
		node *temp = head;
		while (temp->next->next != NULL) {
			temp = temp->next;
			}
// temp now points to the 2nd last element of the linked list
		cout<<"Deleted: "<<temp->next->data<<endl;
		delete(temp->next);
		temp->next = NULL;
		}
 }
```

Delete at a given Position

```
void delete_at_given_position(int p){
        if (head == NULL){
                cout<<"List is Empty"<<endl;
                }
        else{
                node *temp, *ptr;
                if (p == 0) {
                // if p==0, perform delete at beginning
                     cout<<"Element Deleted: "<<head->data<<endl;
                     ptr = head;
                     head = head->next;
                     delete(ptr);
                }
```

Delete at a given Position

```
else{
        // if p > 0, set ptr to pth node and temp to (p-1)th node
        temp = ptr = head;
        while(p>0){
                --p;
                temp = ptr;
                ptr = ptr->next;
                }
        cout<<"Element Deleted: "<<ptr->data<<endl;
        // set next of (p-1)th node to next of pth node
        temp->next = ptr->next;
        free(ptr);
        }
    }
}
```

Print the List

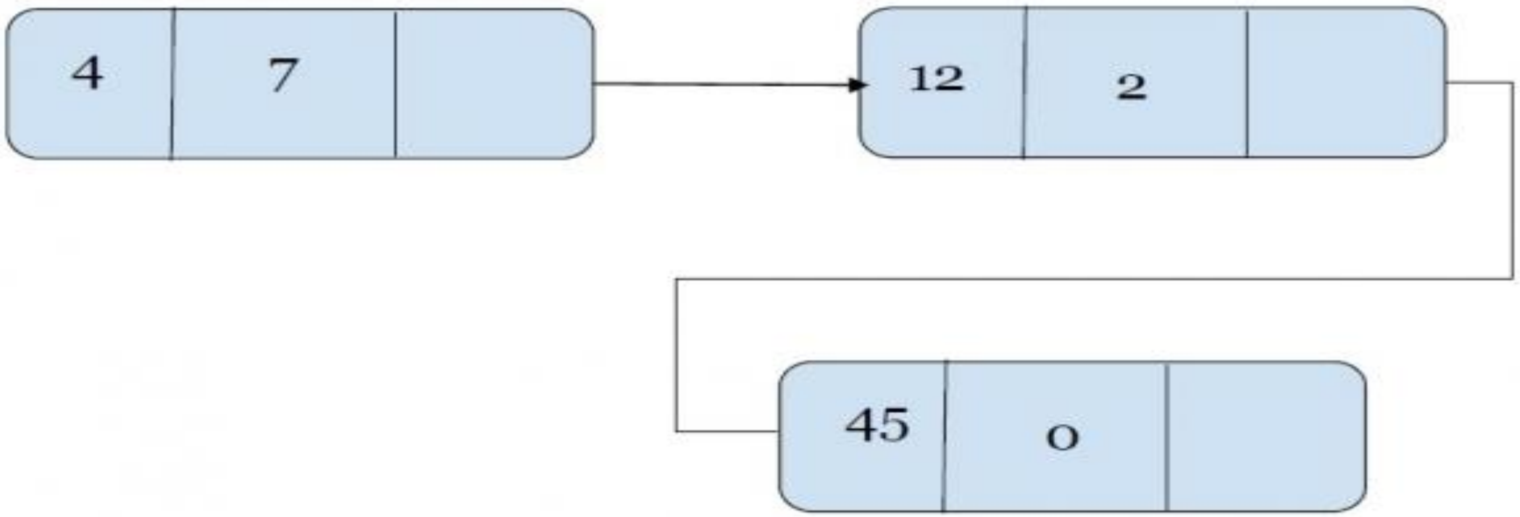
```
void print(){
	if (head == NULL){
		cout<<"List is empty"<<endl;
		}
	else{
		node *temp = head;
		cout<<"Linked List: ";
		while (temp != NULL){
			cout<<temp->data<<"->";
			temp = temp->next;
			}
		cout<<"NULL"<<endl;
		}
	}
```

Main Program

```
int main() {
        printf("1 to Insert at the beginning");
        printf("\n2 to Insert at the end");
        printf("\n3 to Insert at mid");
        printf("\n4 to Delete from beginning");
        printf("\n5 to Delete from the end");
        printf("\n6 to Delete from mid");
        printf("\n7 to Display");
        printf("\n0 to Exit");
        int choice,data,p;
        LinkedList ll;
        do {
                cout<<"\nEnter Your Choice: ";
                cin>>choice;
                switch (choice) {
                        case 1:
                                cout<<"Enter Element: ";
                                cin>>data;
                                ll.insert_at_beginning(v);
                                break;
                        case 2:  /*similar way*/
                        }
        } while (choice != 0);
  }
```

Polynomial Representation:

Polynomial : $4x^7 + 12x^2 + 45$




```
class Node{
        int coeff;
        int pow;
        Node *next;
};
```

Question: Add 2 polynomials.

Practice Questions

- Concatenate two lists
- Merge two lists
- Reverse a list.

Doubly Linked Lists

```
class dnode
{
        int info;
        dnode *next;
        dnode *prev;
 public:
        dnode* insb(dnode*);
        dnode* inse(dnode*);
        void delev(int);
        void print(dnode*);
};
```

Insert at Beginning

```
dnode* dnode::insb(dnode *head){
        dnode *temp=new dnode;
        cout<<"\n Info: ";
        cin>>temp->info;
        temp->prev=temp->next=NULL;
        if(head==NULL) {
                head=temp;
                return head;
            }
        head->prev=temp;
        temp->next=head;
        head=temp;
        return head;
}
```

Insert at End

```
dnode *dnode::inse(dnode *head){
        dnode *temp=new dnode;
        cout<<"\n Info: ";
        cin>>temp->info;
        temp->prev=temp->next=NULL;
        if(head==NULL) {
                head=temp;
                return head;
               }
        dnode *cur=head;
        while(cur->next!=NULL) {
                cur=cur->next;
               }
        cur->next=temp;
        temp->prev=cur;
        return head;
       }
```

Delete A Value

```
void delv(int num)
{
    if(head != NULL)
    {
        link * cur_ptr, *prev_ptr, *del_ptr;
        cur_ptr = head;
        prev_ptr = cur_ptr;
        while(cur_ptr->next != NULL)
        {
            if(head->data == num)
            {
                del_ptr = cur_ptr;
                head = cur_ptr->next;
                head->prev = NULL;
                free(del_ptr);
             }
```

Delete A Value

```
if(cur_ptr->data == num)
        {
           del_ptr = cur_ptr;
           prev_ptr->next = cur_ptr->next;
           cur_ptr->next->prev = prev_ptr;
           free(del_ptr);
           cur_ptr = prev_ptr;
        }
        prev_ptr = cur_ptr;
        cur_ptr = cur_ptr->next;
     }
```

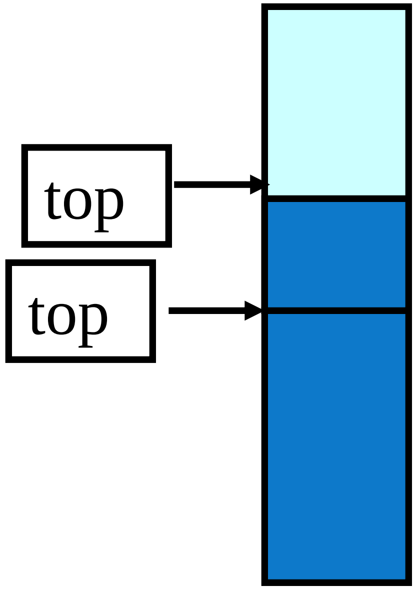
Print/Display

```
void dnode::print(dnode *head)
{
 dnode *f=head;
 while(f!=NULL)
 {
   cout<<f->info<<"->";
   f=f->next;
}}
```

Stack Implementations: Using Array and Linked List

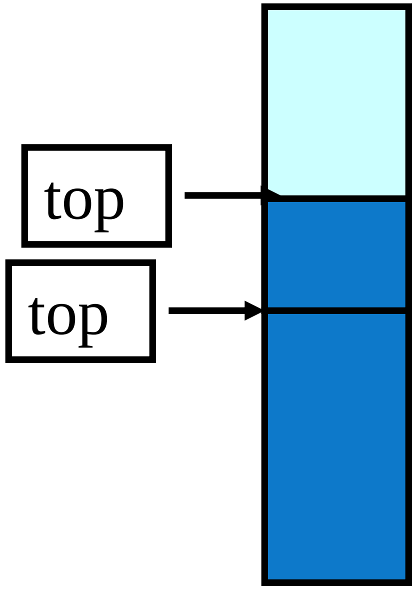
Stack Using Array

PUSH



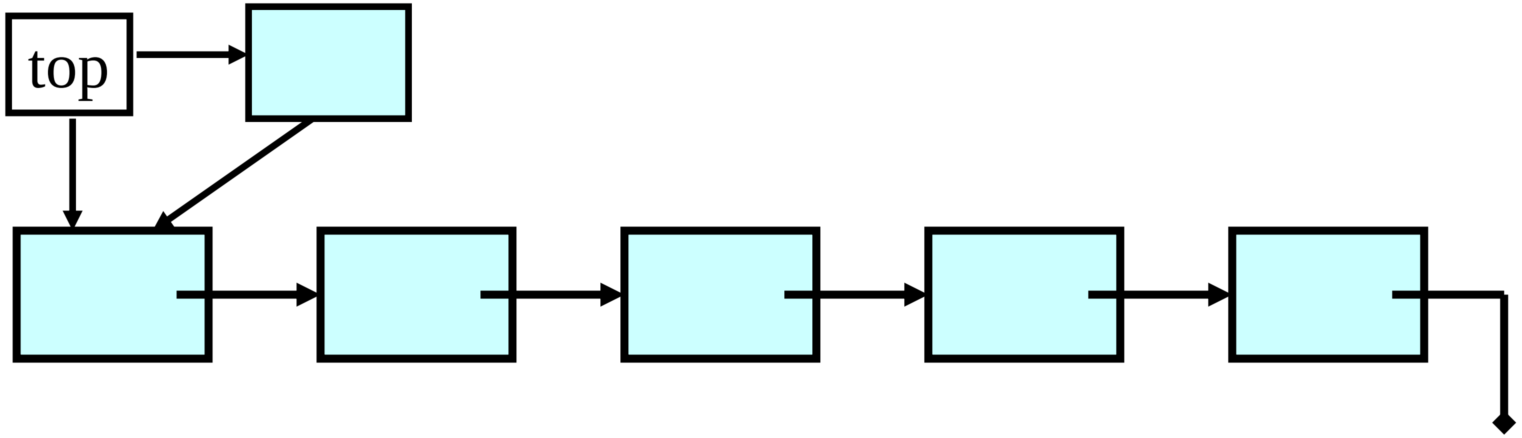
Stack Using Array

POP

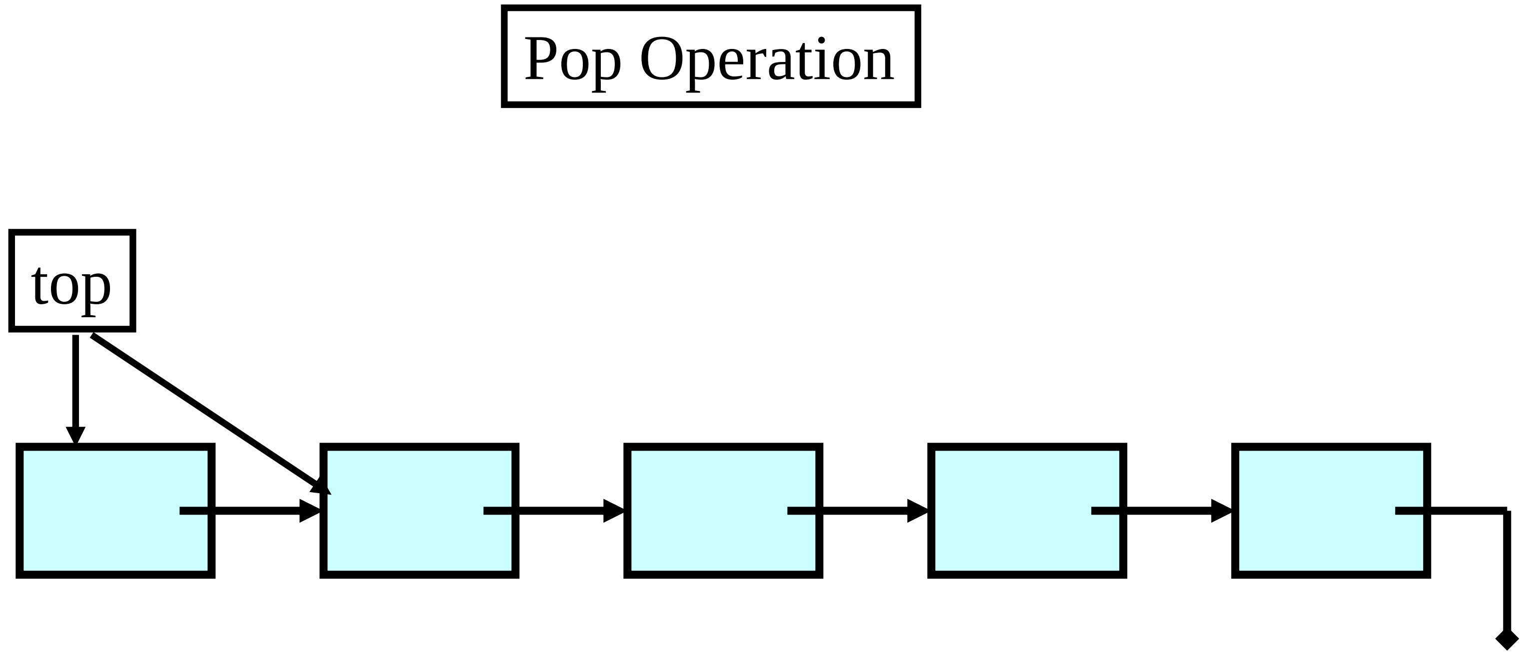


Stack: Linked List Structure

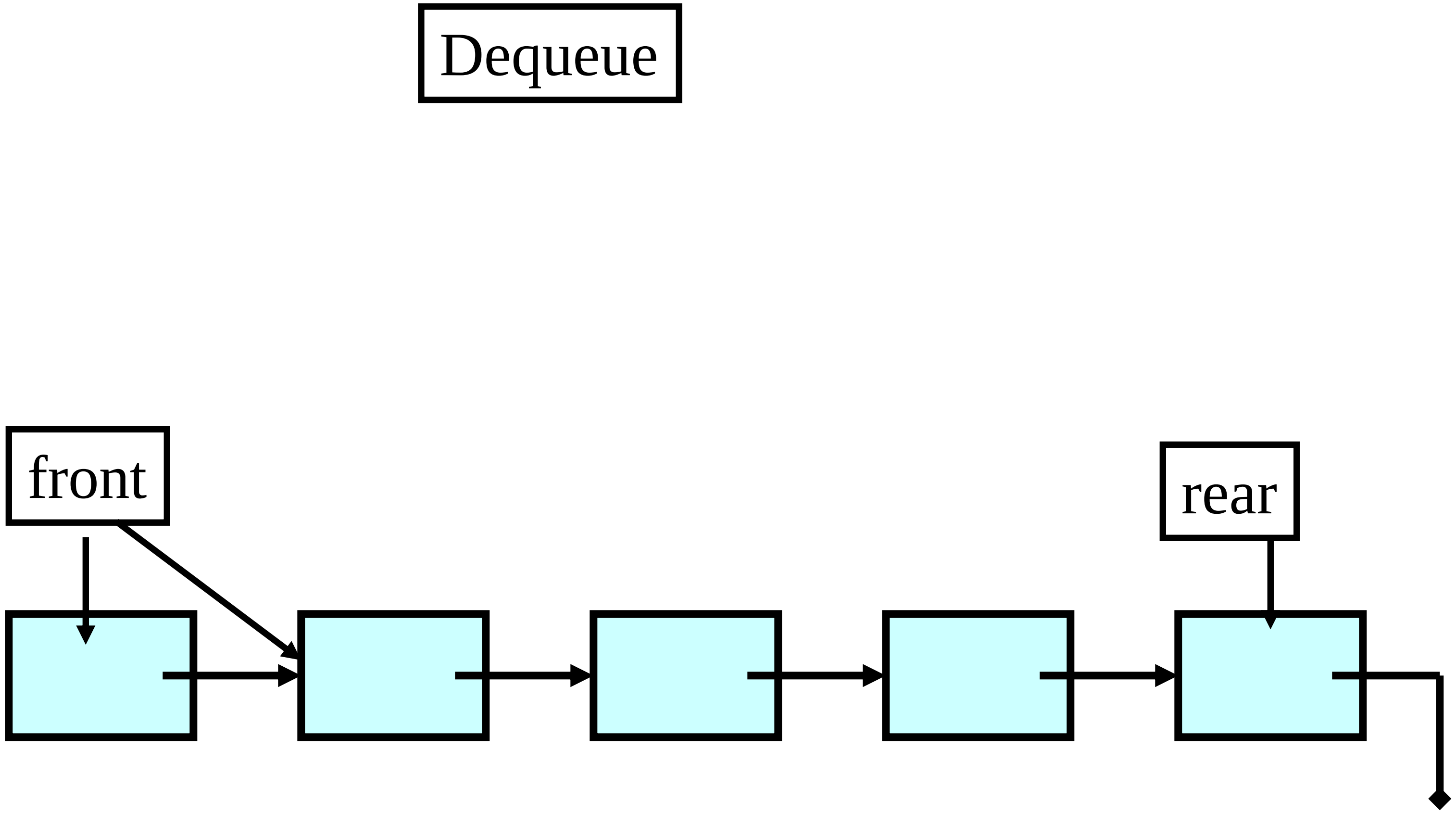
Push Operation



Stack: Linked List Structure



Queue: Linked List Structure



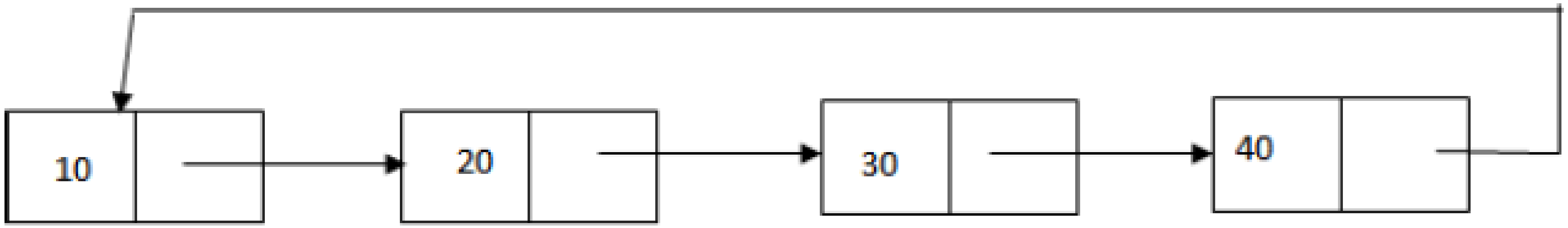
Circular Singly Linked List

Disadvantages of Singly Linked List

- There is only one link field and hence traversing is done in only one direction.
- To delete a designated node X, address of the first node in the list should be given.

Circular Singly Linked List

- A circular list is a variation of the ordinary list in which link field of the last node contains the address of the first node.

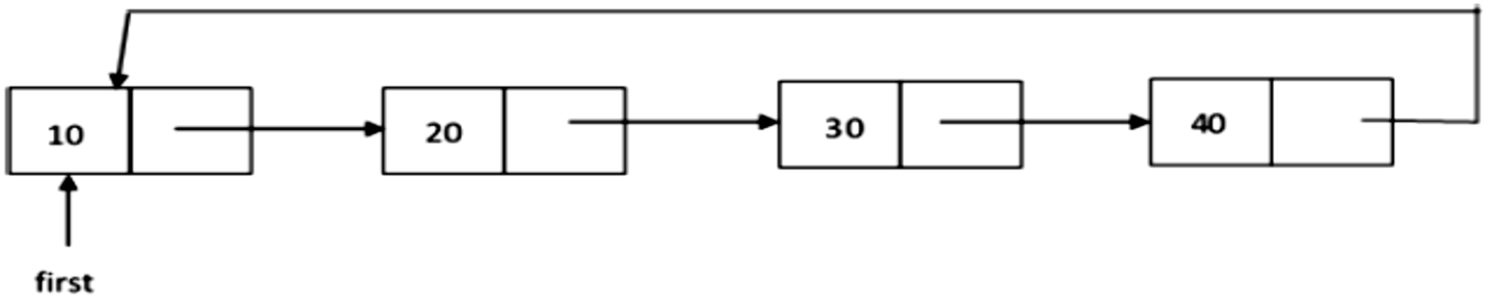


Advantages of Circular List

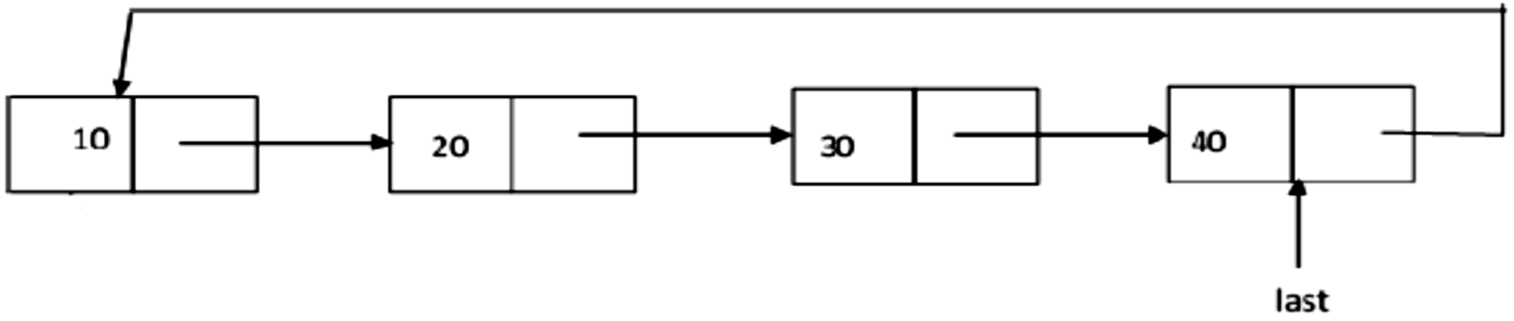
- Every node is accessible from a given node by traversing successively using the link field.
- To delete a node, the address of the first node is not necessary. Search for the predecessor of the current node can be initiated by *curr* itself.

Approaches

- A pointer ***first*** is designated to the starting node of the list. Traverse the list till the last element (which is the predecessor of the designated first).



- A pointer variable last is designated to the last node and the node that follows last, will be the first node of the list.



Circular Linked List Class

```
class cnode {
        int info;
        cnode* next;
 public:
        cnode* insrt(cnode*);
        cnode* insfrnt(cnode*);
        cnode* insfrl(cnode*);
        cnode* inslas(cnode*);
        cnode* rem_dup(cnode*);
        cnode* delle(cnode*);
        cnode* dellb(cnode*);
        cnode* delfe(cnode*);
        void print(cnode*);
        void printl(cnode*);
};
```

Insert : Beginning

Using Last Pointer

```
//Inserting in beginning using the last pointer
cnode* cnode::insfrl(cnode *last) {
        cnode *temp=new cnode;
        cout<<"\nEnter the element:\n";
        cin>>temp->info;
        if(last==NULL) {
                last=temp;
                temp->next=last;
                }
        else {
                temp->next=last->next;
                last->next=temp;
                }
        return last;
        }
```

Insert: End

Using Last Pointer


```
//Inserting in end using the last ptr
cnode* cnode::inslas(cnode *last) {
        cnode *temp=new cnode;
        cout<<"\nEnter the element:\n";
        cin>>temp->info;
        if(last==NULL) {
                last=temp;
                temp->next=last;
                }
        else {
                temp->next=last->next;
                last->next=temp;
                last=temp;
                }
        return last;
        }
```

Insert : Beginning

Using First Pointer

```
//Inserting in beginning using the first ptr
cnode* cnode::insfrnt(cnode *head) {
        cnode *temp=new cnode,*cur=head;
        cout<<"Enter the value to be inserted:";
        cin>>temp->info;
        temp->next=NULL;
        if(head==NULL) {
                head=temp;
                temp->next=head;
               }
        else {
                temp->next=head;
                while(cur->next!=head)
                        cur=cur->next;
                cur->next=temp;
                head=temp;
               }
        return head;
       }
```

Insert: End

Using First Pointer

```
//Inserting in end using the first ptr
cnode* cnode::insrt(cnode *head) {
        cnode *temp=new cnode;
        cnode *cur;
        cout<<"Enter the value to be inserted:";
        cin>>temp->info;
        temp->next=NULL;
        if(head==NULL) {
                head=temp;
                temp->next=head;
               }
        else {
                cur=head;
                while(cur->next!=head)
                        cur=cur->next;
                cur->next=temp;
                temp->next=head;
               }
        return head;
       }
```

Print the Nodes

```
void cnode::print(cnode *head) {
        cnode *h=head;
        cout<<h->info<<"->";
        h=h->next;
        while(h!=head) {
                cout<<h->info<<"->";
                h=h->next;
                }
        }

void cnode::printl(cnode *last) {
        cnode *h=last->next;
        while(h!=last) {
                cout<<h->info<<"->";
                h=h->next;
                }
        cout<<h->info;
        }
```

Delete: End

Using First Pointer

```
//Deleting an element from the end using first pointer
cnode* cnode::delfe(cnode *head) {
        cnode *cur;
        if(head==NULL) {
                cout<<"\nNo records to delete";
                return NULL;
               }
        if(head->next==head) {
                cout<<"\nDeleted item:"<<head->info;
                delete head;
                return NULL;
              }
       cur=head;
       while((cur->next)->next!=head) {
                cur=cur->next;
                }
       cnode *t=cur->next;
       cur->next=head;
       cout<<"\nItem deleted:"<<t->info;
       delete t;
       return head;
       }
```

Delete: End

Using Last Pointer

```
//Deleting an element from the end using a last pointer
cnode* cnode::delle(cnode *last) {
        if(last==NULL) {
                cout<<"\nNo elements to delete:";
                return NULL;
                }
        if(last->next==last) {
                cout<<"Element deleted is:"<<last->info;
                delete (last);
                return NULL;
                }
       cnode *cur=last->next;
       while(cur->next!=last) {
                 cur=cur->next;
                }
        cur->next=last->next;
        cout<<"\nItem deleted: "<<last->info;
        delete(last);
        last=cur;
       return last;
       }
```

Delete: Beginning

Using Last Pointer

```
//Deleting an element from the beginning using last pointer
cnode* cnode::dellb(cnode* last) {
        cnode *cur;
        if(last==NULL) {
                cout<<"\nNo nodes to delete";
                return NULL;
               }
        if(last->next==last) {
                cout<<"Element deleted is: "<<last->info;
                delete (last->next);
                return NULL;
               }
        cur=last->next;
        last->next=cur->next;
        cout<<"\n Item deleted:"<<cur->info;
        delete cur;
        return last;
        }
```

Books

- Ellis Horowitz, Sartaj Sahni, Susan Anderson-Freed, Fundamentals of Data structures in C (2e), Silicon Press, 2008.
- Ellis Horowitz, Sartaj Sahni, Dinesh Mehta, Fundamentals of Data Structures in C++ (2e), Galgotia Publications, 2008.