**Unit II**

**Linked List**

**Structures provided by linked list:**

1. Singly Linked-list (SLL)
2. Doubly Linked-list (DLL)
3. Circular Linkedlist (CLL)

3.1)Circular Singly Linked-list (CSLL)

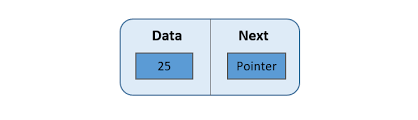
3.2)Circular Doubly Linked-list (CDLL)

1. **Singly Linkedlist:**

* Singly Linked list is one of the structure of dynamic linear data strucutre which provide the structure like… Each element or node have a link with its next node(except first and last node).
* Or simple we can say each node next part have a reference of next node.
* Singly Linked list is a uni directional structure.we cannot access the elements in reverse order.

**Node creation for SLL:**

Each node have two memory parts,one is data part another one is refereance of next node.



struct Node{

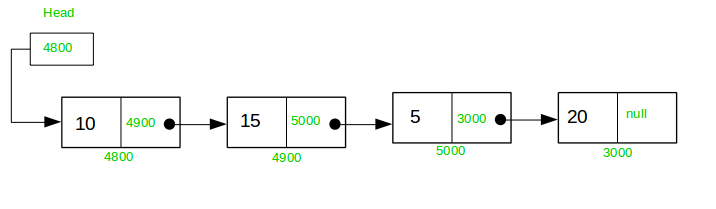
int data; //for example integet data

struct Node \*next; //next part going to store reference of next node so pointer

};

**Operations in SLL:**

1. Insertion (Beginning , End , position)
2. Deletion (Beginning , End , based on position,Based on data)
3. Empty
4. Full
5. Count
6. Searching
7. Sorting ect……



**Programmatic implementation of SLL:**

#include<stdio.h>

#include<stdlib.h>

**struct Node{**

**int data;**

**struct Node \*next;**

**};**

struct Node \*head=NULL;

**void insert\_begin()**

{

struct Node\* newnode;

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

int value;

if(newnode==NULL)

printf("No memory");

else{

scanf("%d",&value);

if(head==NULL)

{

newnode->data = value;

newnode->next=NULL;

head =newnode;

}

else{

newnode->data = value;

newnode->next = head;

head=newnode;

}

}

}

**void display()**

{

struct Node \*temp;

temp = head;

if(head==NULL)

printf("List is empty");

else{

while(temp!=NULL)

{

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

temp = temp->next;

}

}

}

**void insert\_end(int value)**

{

struct Node \*newnode,\*temp;

temp =head;

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

if(newnode==NULL)

printf("No memory");

else{

if(head==NULL)

{

newnode->data = value;

newnode->next=NULL;

head =newnode;

}

else{

while(temp->next!=NULL)

temp = temp->next;

newnode->data = value;

newnode->next=NULL;

temp->next = newnode;

}

}

}

**int insert\_position(int pos,int value)**{

struct Node \*newnode,\*temp;

temp=head;

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

if(newnode==NULL)

printf("No memory");

else{

if(head==NULL)

{

newnode->data = value;

newnode->next=NULL;

head =newnode;

}

else{

for(int itr=1;itr<=pos-2;itr+=1){

if(temp->next==NULL){

printf("Cannot insert");

return -1;

}

temp = temp->next;

}

newnode->data = value;

newnode->next = temp->next;

temp->next=newnode;

}

}

return 1;

}

**void delete\_begin**()

{

struct Node\* temp;

temp = head;

if(head==NULL)

printf("Empty");

else{

head = head->next;

free(temp);

}

}

**void delete\_end()**

{

struct Node\* temp;

temp = head;

if(head==NULL)

printf("List is empty");

else{

while(temp->next->next!=NULL)

temp = temp->next;

struct Node\* temp1=temp->next;

temp->next=NULL;

free(temp1);

}

}

**int delete\_position(int pos)**

{

struct Node\* temp,\*temp1;

temp = head;

if(head==NULL)

printf("List is empty");

else{

for(int itr=1;itr<=pos-2;itr+=1)

{

if(temp->next==NULL){

printf("Cannot insert");

return -1;

}

temp=temp->next;

}

temp1 = temp->next;

temp->next = temp1->next;

temp1->next=NULL;

free(temp1);

}

return 1;

}

**int linear\_search()**

{

int key;

printf("\nEnter a key");

scanf("%d",&key);

struct Node\* temp;

temp= head;

if(head==NULL)

printf("List is empty");

else{

while(temp!=NULL)

{

if(temp->data==key)

return 0;

temp =temp->next;

}

}

return 1;

}

**int length()**

{

struct Node \*temp;

temp = head;

int count=0;

while(temp!=NULL)

{

count++;

temp=temp->next;

}

return count;

}

**void bubble\_sort(int len)**

{

struct Node\* temp;

if(head==NULL)

printf("list is empty");

else

{

int third;

for(int pass=1;pass<=len-1;pass+=1)

{

temp=head;

while(temp->next!=NULL)

{

if(temp->data < temp->next->data)

{

third = temp->data;

temp->data = temp->next->data;

temp->next->data =third;

}

temp =temp->next;

}

}

}

}

**int main()**

{

//1.to create sll list call insert\_begin or insert\_end at any no of times

//2.insert node at beginning call insert\_begin()

//3.insert node at end call insert\_end()

//4.insert node at position call insert\_position()

//5.delete node at beginning call begin\_begin()

//6.delete node at end call delete\_end()

//7.delete node at position call delete\_position()

//8.to find length of the list call length()

//9. to search a element call linear\_search()

//10.to sort the list call binary\_sort

//11.to display the list call display()

}

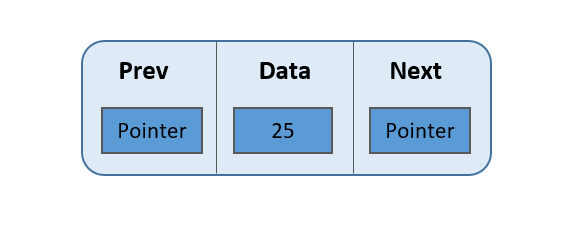
**Key points to remember:**

1. Initially assign head as NULL then always we should maintain first node as head node(it indicates beginning of list)
2. Always we should maintain last node next part as NULL(it indicates the last node of the list)
3. We should link each node with its next node(except last node).
4. If you want maintain last node as tail(but not mandatory)
5. **Doubly Linkedlist:**

* Doubly Linkedlist is one of the Dynamic linear data structure is which provide the structure like…Each element or node have link with its adjacent nodes (previous and next node).(except first and last node)
* Simply we can say each node have links of previous node reference and next node reference.
* DLL is bidirectional structure.We can access the elements both forward and backward order.

**Node creation for DLL:**

Each node have three memory parts,one is data part another two are reference of previous and next node.



struct Node{

struct Node \*prev; //previous part going to store reference of previous node so pointer

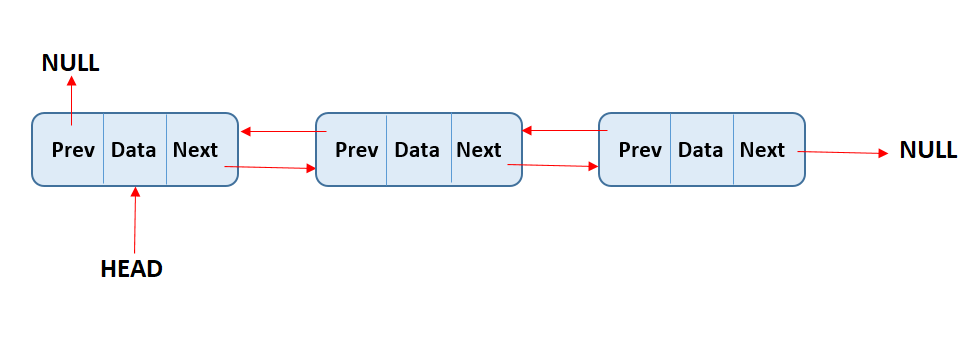
int data; //for example integet data

struct Node \*next; //next part going to store reference of next node so pointer

};

**Operations in DLL:**

1. Insertion (Beginning , End , position)
2. Deletion (Beginning , End , based on position,Based on data)
3. Empty
4. Full
5. Count
6. Searching
7. Sorting ect……



**Programmatic implementation of DLL:**

#include<stdio.h>

#include<stdlib.h>

**struct Node**

**{**

**struct Node \*prev;**

**int data;**

**struct Node \*next;**

**};**

struct Node \*head =NULL;

**void insert\_begin()**

{

struct Node \*newnode;

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

if(newnode==NULL)

printf("No memory");

else

{

int value;

scanf("%d",&value);

if(head==NULL)

{

newnode->data =value;

newnode->next=NULL;

newnode->prev=NULL;

head=newnode;

}

else

{

newnode->data = value;

newnode->next=head;

newnode->prev=NULL;

head->prev=newnode;

head=newnode;

}

}

}

**void display\_forward()**

{

struct Node \*temp =head;

if(head==NULL)

printf("LIst is empty");

else

{

while(temp!=NULL)

{

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

temp=temp->next;

}

}

}

**void display\_backward()**

{

struct Node \*temp=head;

if(head==NULL)

printf("List is empty");

else

{

while(temp->next!=NULL)

temp=temp->next;

while(temp!=NULL)

{

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

temp =temp->prev;

}

}

}

**void insert\_end()**

{

struct Node\* newnode,\*temp;

temp =head;

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

if(newnode==NULL)

printf("No memory");

else

{

int value;

scanf("%d",&value);

if(head==NULL)

{

newnode->data =value;

newnode->next=NULL;

newnode->prev=NULL;

head=newnode;

}

else

{

while(temp->next!=NULL)

temp=temp->next;

newnode->data =value;

newnode->next=NULL;

newnode->prev=temp;

temp->next=newnode;

}

}

}

**int insert\_position()**

{

struct Node\* newnode;

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

if(newnode==NULL)

printf("No memory");

else

{

int pos,value;

scanf("%d%d",&pos,&value);

struct Node \*temp=head;

if(head==NULL)

{

newnode->data =value;

newnode->next=NULL;

newnode->prev=NULL;

head=newnode;

}

else

{

int itr;

for(itr=1;itr<=pos-2;itr+=1)

{

if(temp->next==NULL)

{

printf("Cannot insert");

return -1;

}

temp = temp->next;

}

newnode->data =value;

newnode->next=temp->next;

newnode->prev=temp;

temp->next->prev =newnode;

temp->next=newnode;

}

}

return 1;

}

void delete\_begin()

{

struct Node \*temp=head;

if(head==NULL)

printf("List is empty");

else{

head=head->next;

head->prev=NULL;

temp->next=NULL;

free(temp);

}

}

void delete\_end()

{

struct Node\* temp,\*temp1;

temp=head;

if(head==NULL)

printf("List is empty");

else{

while(temp->next->next!=NULL)

temp=temp->next;

temp1=temp->next;

temp1->prev=NULL;

temp->next=NULL;

free(temp1);

}

}

void delete\_position()

{

int pos;

scanf("%d",&pos);

struct Node \*temp=head;

if(head==NULL)

printf("list is empty");

else{

for(int itr=1;itr<pos;itr+=1)

{

if(temp->next==NULL)

{

printf("cannot delete");

return 1;

}

temp=temp->next;

}

temp->prev->next=temp->next;

temp->next->prev=temp->prev;

temp->prev=NULL;

temp->next=NULL;

free(temp);

}

return -1;

}

int main()

{

//1.to create Dll list call insert\_begin or insert\_end at any no of times

//2.insert node at beginning call insert\_begin()

//3.insert node at end call insert\_end()

//4.insert node at position call insert\_position()

//5.delete node at beginning call begin\_begin()

//6.delete node at end call delete\_end()

//7.delete node at position call delete\_position()

//8.to display forward the list call display\_forward()

//9.to display backward the list call display\_backward()

}

**Key points to remember:**

1. Initially assign head as NULL then always we should maintain first node as head node(it indicates beginning of list)
2. Always we should maintain last node next part as NULL(it indicates the last node of the list) and first node previous part also NULL.
3. Always maintain two way connection between each nodes
4. If you want maintain last node as tail(but not mandatory)

**3.Circular Linkedlist:**

* Circular Linkedlist is a dynamic linear data structure which provide the structure like the next node of last node again it connect with first node like circular manner.
* Instead of pointing last node next is null it points the first node that’s it.
* We can implement this circular concept in previous two structure (SLL & DLL)

**3.1)Circular singly Linked list (CSLL)**

* It is similar to singly linked list each node have link with its next node
* It is also uni directional
* Only one change is last node next part have link with again list’s first node.

**Node creation for CSLL:**

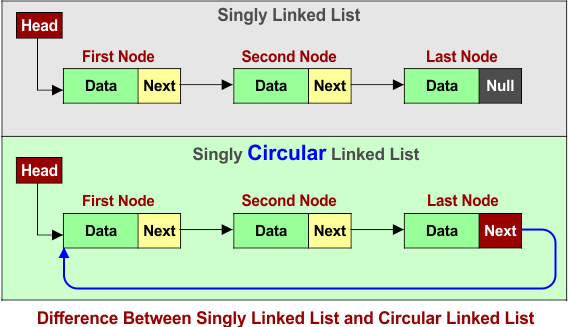
Each node have two memory parts,one is data part another one is refereance of next node.

struct Node{

int data; //for example integet data

struct Node \*next; //next part going to store reference of next node so pointer

};



**Programmatic implementation of CSLL:**

#include<stdio.h>

#include<stdlib.h>

**struct Node**

**{**

**int data;**

**struct Node \*next;**

**};**

struct Node \*head=NULL;

struct Node \*tail=NULL;

**void insert\_begin()**

{

struct Node \*newnode;

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

if(newnode==NULL)

printf("No memory");

else{

int value;

scanf("%d",&value);

if(head==NULL)

{

newnode->data =value;

newnode->next=newnode;

head=newnode;

tail =newnode;

}

else

{

newnode->data=value; //100 10 20 30 40 50

newnode->next=head;

head=newnode;

tail->next=head;

}

}

}

**void display()**

{

struct Node \*temp=head;

if(head==NULL)

printf("List is empty");

else{

do{

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

temp=temp->next;

}while(temp!=head);

}

}

**void insert\_end()**

{

struct Node \*newnode;

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

if(newnode==NULL)

printf("No memory");

else{

int value;

scanf("%d",&value);

if(head==NULL)

{

newnode->data =value;

newnode->next=newnode;

head=newnode;

tail =newnode;

}

else

{

newnode->data=value;

tail->next=newnode;

newnode->next=head;

tail=newnode;

}

}

}

**int insert\_position()**

{

struct Node \*newnode,\*temp;

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

if(newnode==NULL)

printf("No memory");

else{

int value,pos;

printf("\nEnter pos then value\n");

scanf("%d%d",&pos,&value);

if(head==NULL)

{

newnode->data =value;

newnode->next=newnode;

head=newnode;

tail =newnode;

}

else{

temp=head;

for(int i=1;i<=pos-2;i+=1)

{

if(temp==head && i!=1)

{

printf("Cannot insert");

return -1;

}

temp=temp->next;

}

newnode->data=value;

newnode->next=temp->next;

temp->next=newnode;

}

}

return 1;

}

**void delete\_begin()**

{

struct node \*temp=head;

if(head==NULL)

printf("List is empty");

else

{

head=head->next;

tail->next=head;

free(temp);

}

}

**void delete\_end()**

{

struct Node \*temp=head;

if(head==NULL)

printf("List is empty");

else

{

while(temp->next!=tail)

{

temp=temp->next;

}

tail=temp;

temp=tail->next;

tail->next=head;

free(temp);

}

}

**int delete\_position()**

{

struct Node \*temp=head;

if(head==NULL)

printf("List is empty");

else

{

int pos;

printf("\nEnter position");

scanf("%d",&pos);

for(int itr=1;itr<=pos-2;itr+=1)

{

if(temp==head && itr!=1)

{

printf("Cannot delete");

return -1;

}

temp=temp->next;

}

struct Node \*temp1;

temp1=temp->next;

temp->next=temp1->next;

free(temp1);

}

return 1;

}

**int main()**

{

//1.to create csll list call insert\_begin or insert\_end at any no of times

//2.insert node at beginning call insert\_begin()

//3.insert node at end call insert\_end()

//4.insert node at position call insert\_position()

//5.delete node at beginning call begin\_begin()

//6.delete node at end call delete\_end()

//7.delete node at position call delete\_position()

//8.to display the list call display()

}

**Key points to remember:**

1. Initially assign head as NULL then always we should maintain first node as head node(it indicates beginning of list)
2. Always we should maintain last node next part as head node(it indicates the list is in circular manner)
3. We should link each node with its next node and link last node with head node
4. If you want maintain last node as tail(but not mandatory)

**3.2 Circular doubly Linkedlist:**

* CDLL is similar to DLL each node have a link with its adjacent nodes( previous and next node)
* Only one change is last node next and previous have link with first node and forms one circular structure that's it.
* DLL is bidirectional structure.We can access the elements both forward and backward order.

**Node creation for DLL:**

Each node have three memory parts,one is data part another two are reference of previous and next node.

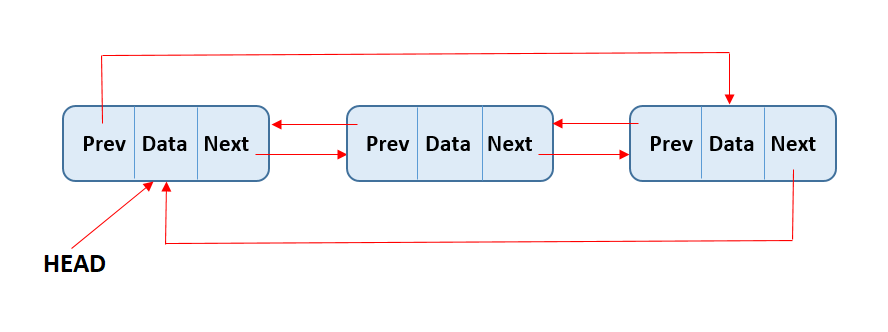
struct Node{

struct Node \*prev; //previous part going to store reference of previous node so pointer

int data; //for example integet data

struct Node \*next; //next part going to store reference of next node so pointer

};



**Operations in DLL:**

1. Insertion (Beginning , End , position)
2. Deletion (Beginning , End , based on position,Based on data)
3. Empty
4. Full
5. Count
6. Searching
7. Sorting ect……

**Programmatic implementation of CDLL:**

#include<stdio.h>

#include<stdlib.h>

**struct Node**

**{**

**struct Node \*prev;**

**int data;**

**struct Node \*next;**

**};**

struct Node \*head=NULL;

struct Node \*tail=NULL;

**void insert\_begin()**

{

struct Node \*newnode;

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

if(newnode==NULL)

printf("No memory");

else{

int value;

scanf("%d",&value);

if(head==NULL)

{

newnode->data=value;

newnode->next=newnode;

newnode->prev=newnode;

head=newnode;

tail=newnode;

}

else

{

newnode->data =value;

newnode->next =head;

newnode->prev=head->prev;

tail->next=newnode;

head->prev =newnode;

head=newnode;

}

}

}

**void display\_forward()**

{

struct Node \*temp=head;

if(head==NULL)

printf("List is empty");

else{

do{

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

temp=temp->next;

}while(temp!=head);

}

}

**void insert\_end()**

{

struct Node \*newnode;

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

if(newnode==NULL)

printf("No memory");

else{

int value;

scanf("%d",&value);

if(head==NULL)

{

newnode->data=value;

newnode->next=newnode;

newnode->prev=newnode;

head=newnode;

tail=newnode;

}

else

{

newnode->data=value;

tail->next=newnode;

newnode->prev=tail;

newnode->next=head;

head->prev=newnode;

tail=newnode;

}

}

}

**int insert\_position()**

{

struct Node \*newnode,\*temp;

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

if(newnode==NULL)

printf("No memory");

else{

int value,pos;

printf("\nEnter pos then value\n");

scanf("%d%d",&pos,&value);

if(head==NULL)

{

newnode->data=value;

newnode->next=newnode;

newnode->prev=newnode;

head=newnode;

tail=newnode;

}

else{

temp=head;

for(int i=1;i<=pos-2;i+=1)

{

if(temp==head && i!=1)

{

printf("Cannot insert");

return -1;

}

temp=temp->next;

}

newnode->data=value;

newnode->prev=temp;

newnode->next=temp->next;

temp->next->prev=newnode;

temp->next=newnode;

}

}

return 1;

}

**void delete\_begin()**

{

struct node \*temp=head;

if(head==NULL)

printf("List is empty");

else

{

head=head->next;

head->prev=tail;

tail->next=head;

free(temp);

}

}

**void delete\_end()**

{

struct Node \*temp=head;

if(head==NULL)

printf("List is empty");

else

{

while(temp!=tail)

{

temp=temp->next;

}

tail=temp->prev;

tail->next=head;

head->prev=tail;

free(temp);

}

}

**int delete\_position()**

{

struct Node \*temp=head;

if(head==NULL)

printf("List is empty");

else

{

int pos;

printf("\nEnter position");

scanf("%d",&pos);

for(int itr=1;itr<=pos-1;itr+=1)

{

if(temp==head && itr!=1)

{

printf("Cannot delete");

return -1;

}

temp=temp->next;

}

if(temp==head)

{

delete\_begin();

}

else{

temp->prev->next=temp->next;

temp->next->prev=temp->prev;

free(temp);

}

}

return 1;

}

**int main()**

{

//1.to create cdll list call insert\_begin or insert\_end at any no of times

//2.insert node at beginning call insert\_begin()

//3.insert node at end call insert\_end()

//4.insert node at position call insert\_position()

//5.delete node at beginning call begin\_begin()

//6.delete node at end call delete\_end()

//7.delete node at position call delete\_position()

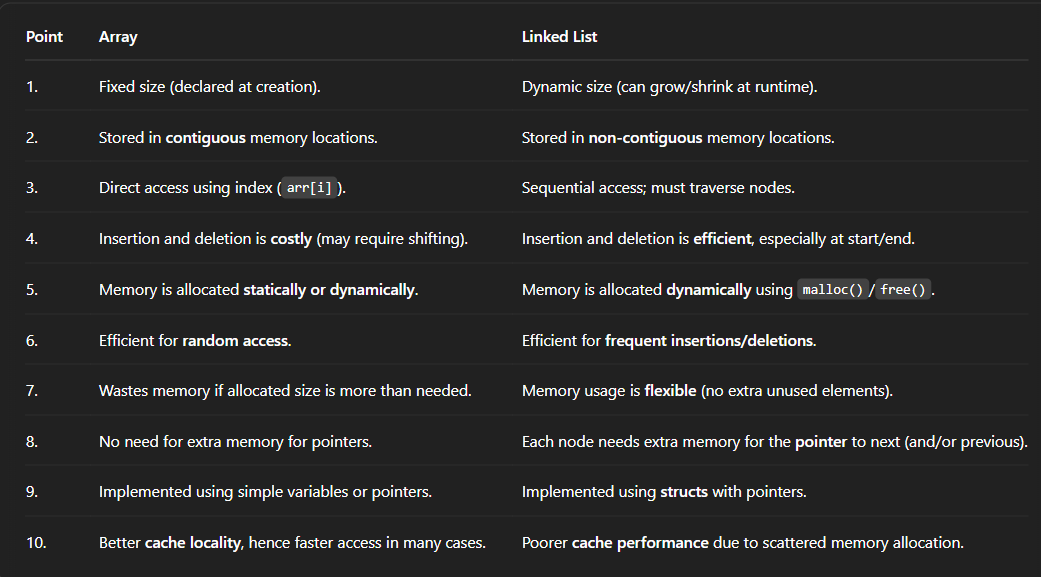
//8.to display forward the list call display\_forward()

}

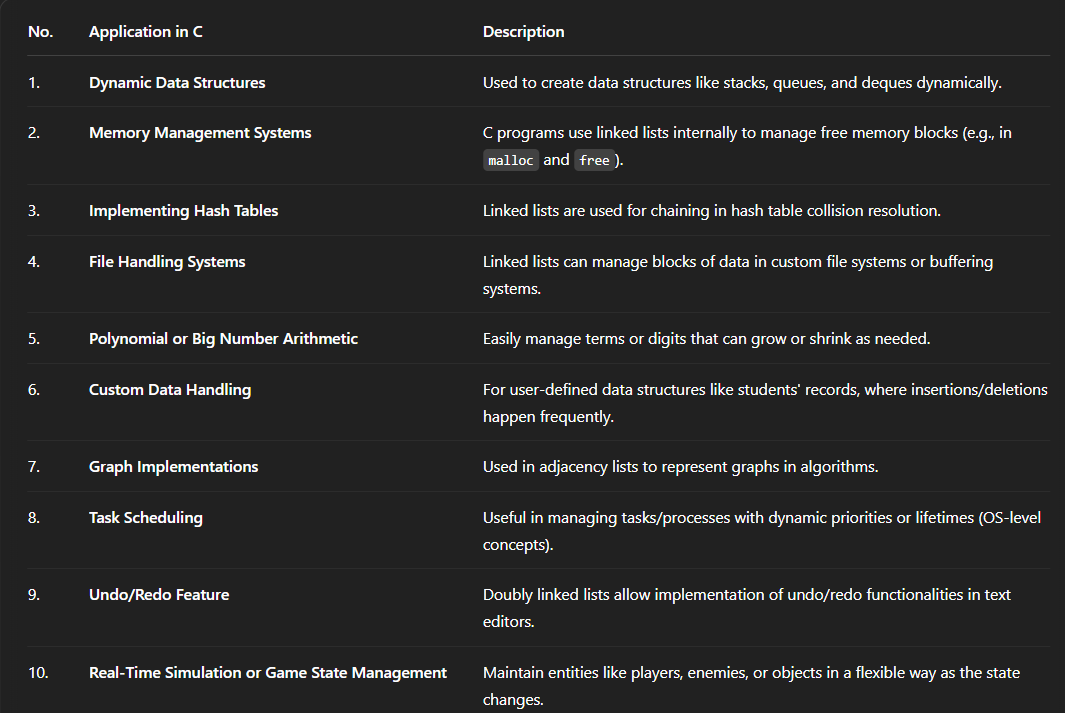
**Key points to remember:**

1. Initially assign head as NULL then always we should maintain first node as head node(it indicates beginning of list)
2. Always we should maintain last node next part as head node(it indicates the last node of the list) and first node previous part also head node.
3. Always maintain two way connection between each nodes
4. If you want maintain last node as tail(but not mandatory)

**Difference between arrays and linked list**



**Applications of linkedlist**



**Searching algorithms**

* Searching is a process of to find something in particular no of things .
* In group of elements (arrays or Lists) Searching operation is used to find a particular

element or datais present or not.

**Types of searching algorithms**

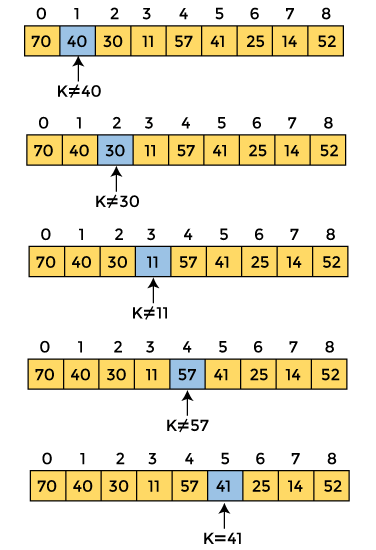
* Linear search algorithm
* Binary search algorithms

**1.Linear search algorithm**

* Linear Search is the simplest search algorithm. It goes through each element one by one in a group of elements (list or array) until the desired element (called the "key") is found, or the end of the list is reached.

**Execution steps of linear search**

* Start from the first element.
* Compare the current element with the key.
* If it matches, element present
* If not, move to the next element.
* If the key is not found after checking all elements, it indicate "not found"



**Linear search implementation in arrays**

int linear\_search(int \*arr,int key,int size)

{

int itr;

for(itr=0;itr<size;itr+=1) //traverse array elements one by one (linear manner)

{

if(arr[itr]= =key)

return 0;

}

return -1;

}

If the function return 0 means key “element found “

If the function return -1 means key element “not found”

**Time Complexity:**

Best case: O(1) (if the key is at the start)

Worst case: O(n) (if the key is at the end or not present)

Space Complexity: O(1) (no extra memory used)

**Important :**Works on both sorted and unsorted arrays.

**Drawback of linear search alsogithm:**

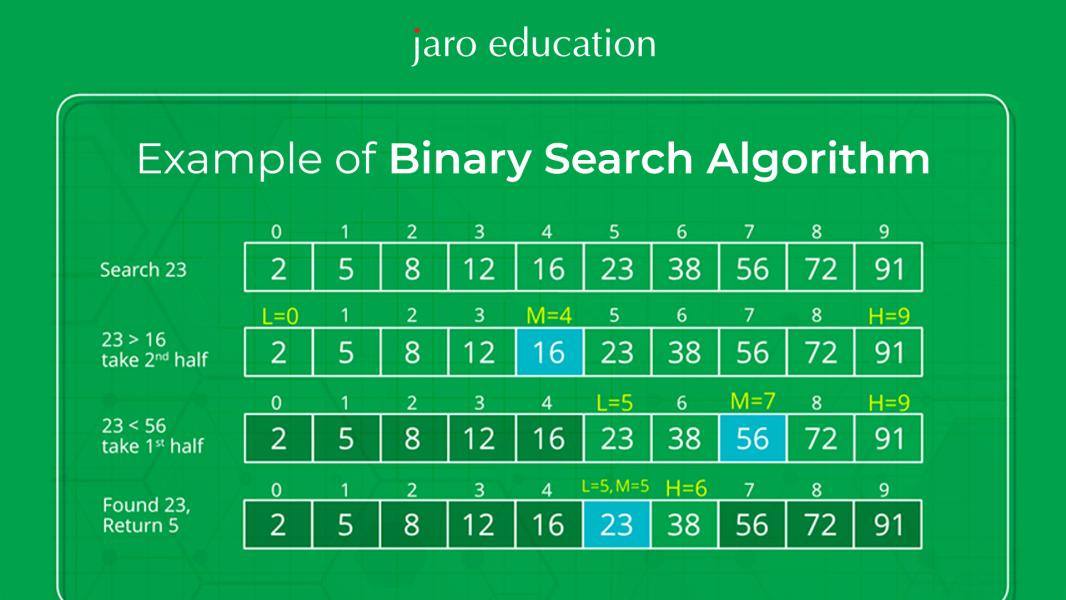
The worst time complexity is O(n-1) or O(n) .It perform n number of unnecessary comparisons

**2.Binary search algorithm**

* Binary Search algorithm is an efficient algorithm for finding an element in a sorted group of elements (array or list).
* It works by repeatedly dividing the search interval in half.
* Binary search algorithm follows **“Divide and Conquer strategy”**

**Execution steps of linear search**

* Start with two pointers: low and high (pointing to the beginning and end of the array).
* Find the middle index: mid = (low + high) / 2.
* Compare the key with arr[mid]:
* If arr[mid] == key, you've found the element.
* If key < arr[mid], search the left half (high = mid - 1).
* If key > arr[mid], search the right half (low = mid + 1).
* Repeat until the element is found or the interval becomes empty (low > high).



**Binary implementation in arrays**

int Binary\_search(int \*arr,int key,int size) //If list should be ascending order

{

int first=0,last=size-1,mid;

while(first <= last)

{

mid = (first +last)/2;

if(arr[mid] = =key)

return 0;

else

{

if(arr[mid] < key) //Suppose list is descending order

First = mid+1; //last = mid-1;

else

Last= mid-1; //first =mid+1;

}

}

return -1;

}

If the function return 0 means key “element found “

If the function return -1 means key element “not found”

**Time Complexity:**

Best case: O(1)

Worst case: O(log n)

Space Complexity: O(1) for iterative version (O(log n) for recursive)

**Drawback :**

It Works on only sorted arrays.

**Sorting Algorithms**

* Sorting is a process of arrange the group of elements either ascending order or descending order
* Ascending order should be n1<n2<n3……..<n
* Descending order should be n1>n2>n3……..>n

**List of sorting algorithms**

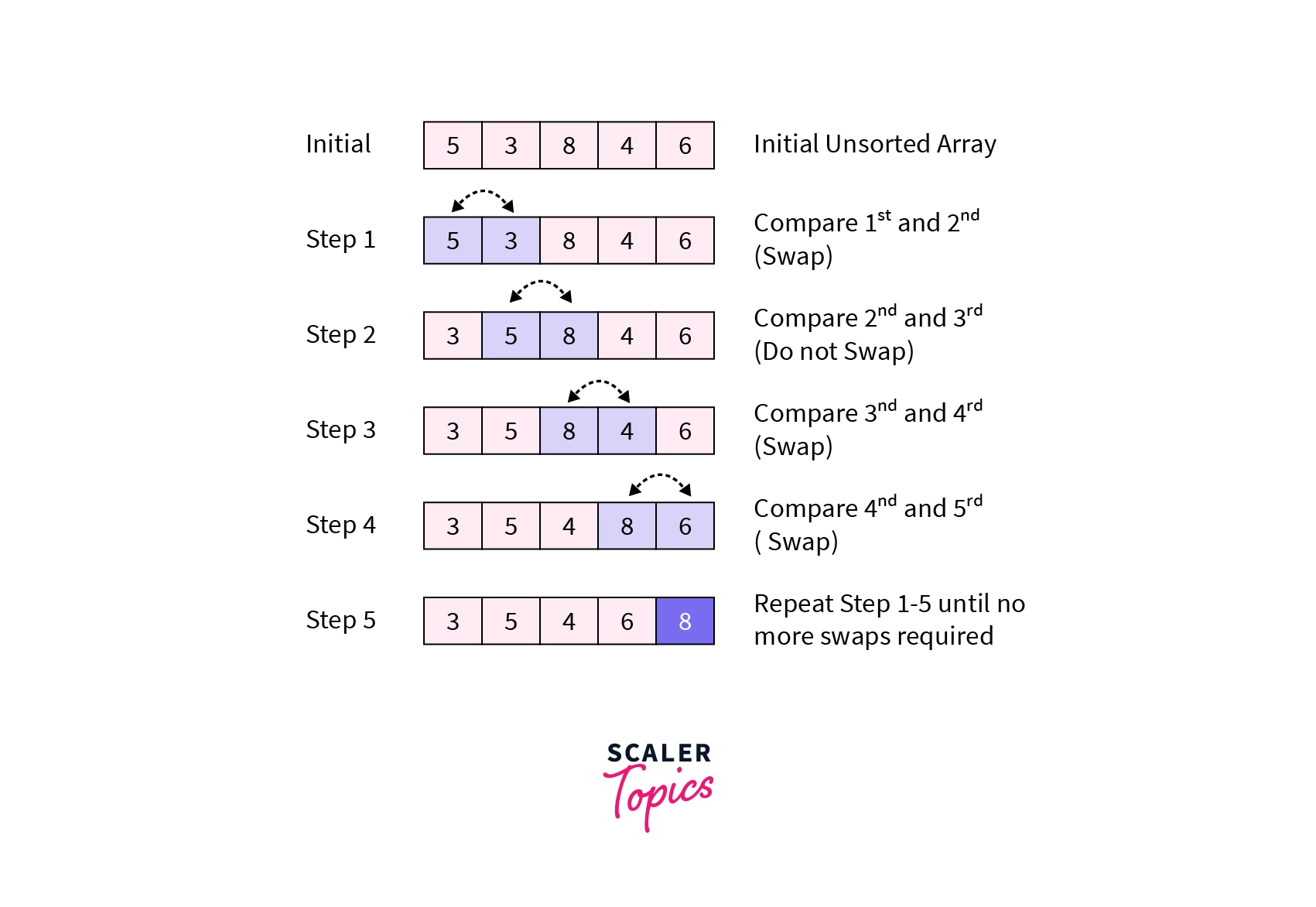
1. Selection sort.
2. Bubble sort.
3. Insertion sort.
4. Merge sort.
5. Quick sort.
6. Heap sort.
7. Counting sort.
8. Radix sort ect…….

**1.Bubble sort algorithm**

* Bubble Sort is a simple sorting algorithm that works by repeatedly swapping adjacent elements if they are in the wrong order. Each one of the array element getting sorted at the end of the each pass.
* We can perform both ascending and descending order.
* While ascending order the largest element "bubbles up" to the end of the array in each pass.
* While descending order the smallest element "bubbles up" to the end of the array in each pass.

**Execution steps of bubble sort**

1. Compare the first and second elements.
2. If the first is greater than the second, swap them.
3. Move to the next pair and repeat.
4. After the first pass, the largest element is at the end.
5. Repeat the process for the remaining unsorted part.
6. Continue until the entire array is sorted.



**In above diagram end of 1st pass one element will be sorted . we need this process for N-1 times.**

**Process:**

**initial 5 3 8 4 6**

**Pass 0 3 5 4 6 8**

**Pass 1 3 4 5 6 8 note:** in between array is sorted we can stop our process

**Pass 2 3 4 5 6 8**

**Pass 4 3 4 5 6 8**

**Bubble sort Implementation in Arrays:**

void bubble\_sort(int \*arr,int size) //ascending order sorting

{

int pass,itr,temp;

for(pass=0;pass<size-1;pass+=1)

{

for(itr=0;itr<size-1-pass;itr+=1)

{

if(arr[itr] > arr[itr+1]) { //If you want to do descending order change instead of ‘>’ use ‘<’

temp = arr[itr];

arr[itr] = arr[itr+1];

arr[itr+1] = temp;

}

}

}

}

Note: we should call this function inside of the main

**Time Complexity:**

Best case: O(n) (if already sorted, with optimization)

Average & Worst case: O(n²)

**Space Complexity:** O(1) (in-place sorting)

**Stable sort:** Keeps equal elements in the same order.

**Drawbacks:**

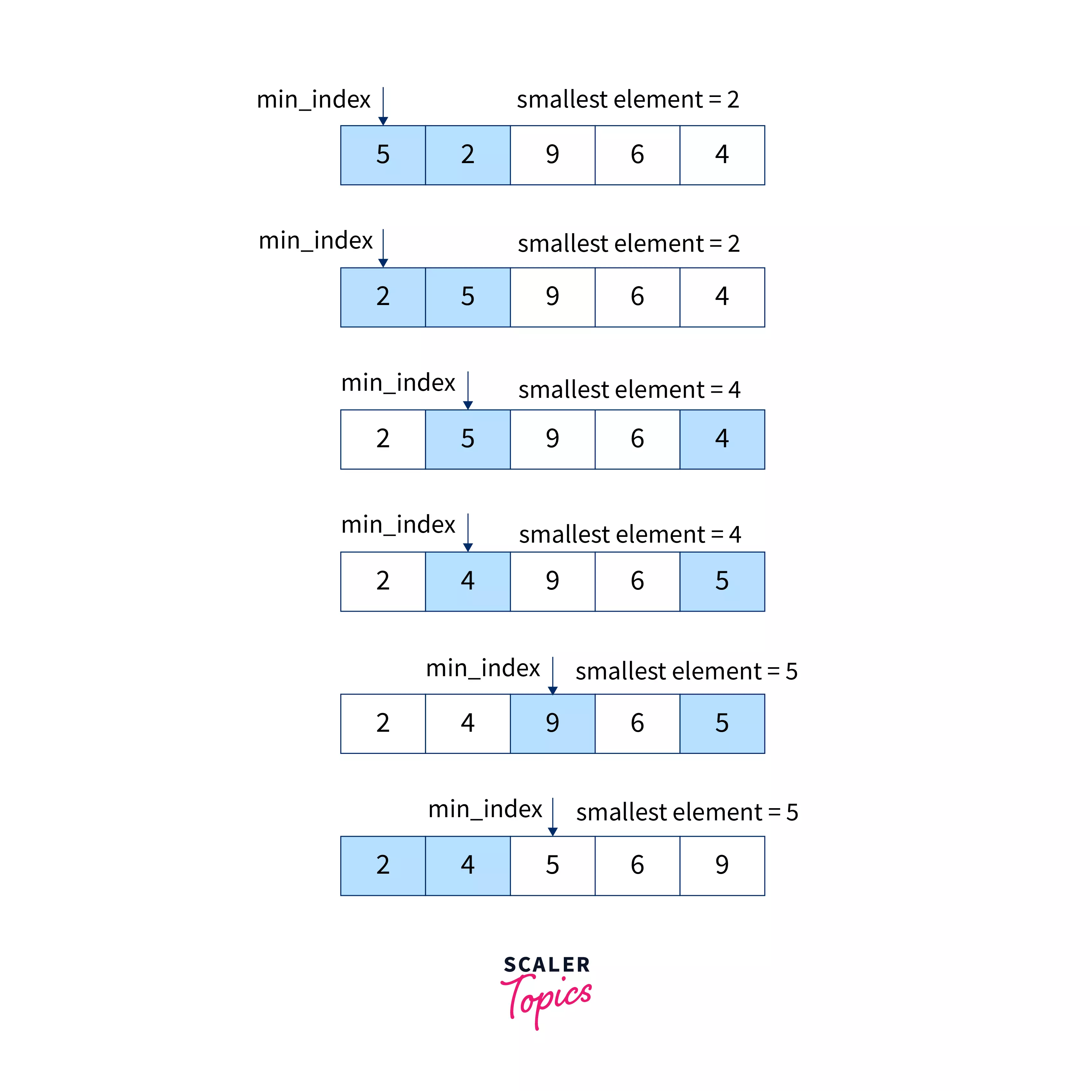
Easy to implement, but not efficient for large datasets.

**2.Selection sort Algorithm**

* Selection Sort is a simple comparison-based sorting algorithm. It works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the array and placing it in its correct position.
* In ascending order select smallest element from the unsorted portion of the array and placing it in its correct position.
* In descending order select largest element from the unsorted portion of the array and placing it in its correct position.

**Execution steps of selection sort**

1. Start from the first element, and assume it's the minimum.
2. Compare this minimum with the rest of the array.
3. If you find a smaller element, update the minimum.
4. After checking the whole array, swap the minimum element with the first element.
5. Repeat the same steps for the remaining unsorted portion.



**selection sort Implementation in Arrays:**

void Selection\_sort(int arr[], int size)

{

int i,j,minInd,temp;

for (i = 0; i < size - 1; i+=1)

{

minInd = i;

for (j = i + 1; j < size; j+=1) // Find the minimum element in the unsorted part

{

if (arr[j] < arr[minInd])

{

minInd = j;

}

}

if (minInd != i) // Swap the found minimum element with the first element

{

temp = arr[i];

arr[i] = arr[minInd];

arr[minInd] = temp;

}

}

}

Note: we should call this function inside of the main

**Time Complexity :** Best, Average, Worst: O(n²)

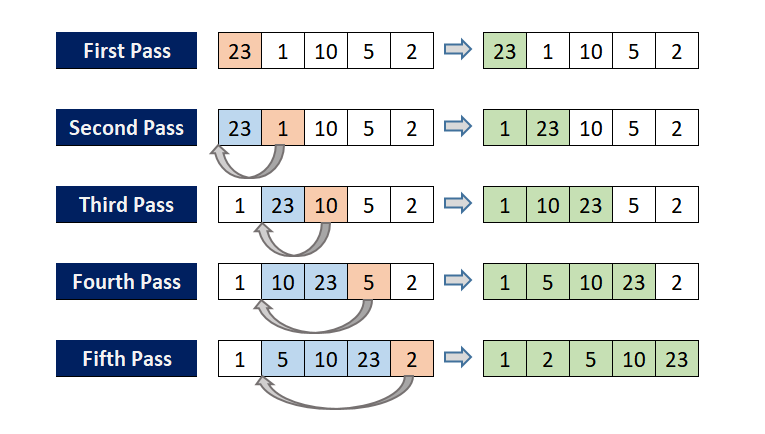
**Space Complexity :** O(1) (In-place sort, no extra space needed)

**Insertion sort Algorithm**

* Insertion Sort is a simple and intuitive comparison-based sorting algorithm.
* It builds the final sorted array one element at a time.here we need to took a element and insert at its right position.
* like how we sort playing cards in hand.

**Execution steps of selection sort**

* Start with the second element (index 1).
* Compare it with the element before it.
* Shift larger elements one position to the right.
* Insert the current element into the correct position.
* Repeat for all elements.



**Insertion sort Implementation in Arrays:**

void Insertion\_sort(int arr[], int size)

{

int i,j,temp;

for (i = 1; i < size; i++)

{

temp = arr[i];

for (j=i - 1;j >= 0 && arr[j] > temp;j-=1)

{

arr[j + 1] = arr[j]; // Move elements greater than key to one position ahead

}

arr[j + 1] = temp; // replacing the temp value

}

}

Note: we should call this function inside of the main

**Time Complexity:**

* Best: O(n) (when already sorted)
* Average :O(n²)
* Worst: O(n²)

**Space Complexity :**

* O(1) (in-place sorting)