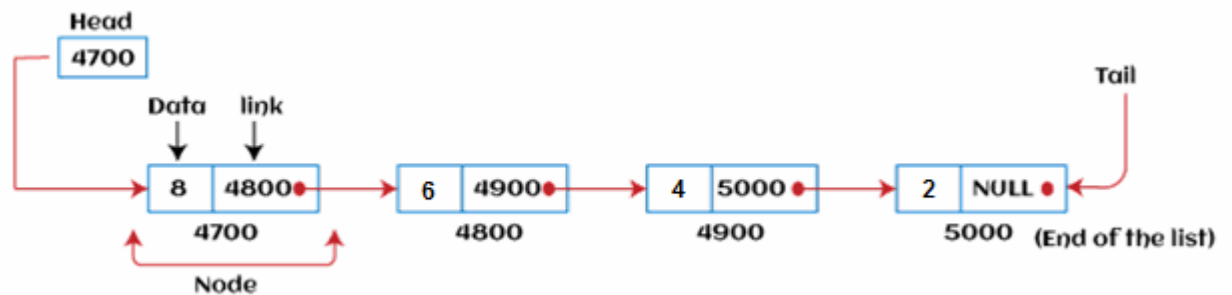


Linked list

Linked list is a linear data structure that includes a series of connected nodes. Linked list can be defined as the nodes that are randomly stored in the memory. A node in the linked list contains two parts, i.e., first is the data part and second is the address part. The last node of the list contains a pointer to the null. After array, linked list is the second most used data structure. In a linked list, every link contains a connection to another link.

Representation of a Linked list

Linked list can be represented as the connection of nodes in which each node points to the next node of the list. The representation of the linked list is shown below -



Till now, we have been using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages that must be known to decide the data structure that will be used throughout the program.

Now, the question arises why we should use linked list over array?

Why use linked list over array?

Linked list is a data structure that overcomes the limitations of arrays. Let's first see some of the limitations of arrays -

- The size of the array must be known in advance before using it in the program.
- Increasing the size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
- All the elements in the array need to be contiguously stored in the memory. Inserting an element in the array needs shifting of all its predecessors.

Linked list is useful because -

- It allocates the memory dynamically. All the nodes of the linked list are non-contiguously stored in the memory and linked together with the help of pointers.
- In linked list, size is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

How to declare a linked list?

It is simple to declare an array, as it is of single type, while the declaration of linked list is a bit more typical than array. Linked list contains two parts, and both are of different types, i.e., one is the simple variable, while another is the pointer variable. We can declare the linked list by using the user-defined data type **structure**.

The declaration of linked list is given as follows -

1. struct node
2. {
3. int data;
4. struct node *next;
5. }

In the above declaration, we have defined a structure named as **node** that contains two variables, one is **data** that is of integer type, and another one is **next** that is a pointer which contains the address of next node.

Advertisement DS

Now, let's move towards the types of linked list.

Types of Linked list

Linked list is classified into the following types -

- **Singly-linked list** - Singly linked list can be defined as the collection of an ordered set of elements. A node in the singly linked list consists of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node, while the link part of the node stores the address of its immediate successor.
- **Doubly linked list** - Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly-linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), and pointer to the previous node (previous pointer).
- **Circular singly linked list** - In a circular singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.
- **Circular doubly linked list** - Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the nodes. The last node of the list contains the address of the first node of the list. The first node of the list also contains the address of the last node in its previous pointer.

Now, let's see the benefits and limitations of using the Linked list.

Advantages of Linked list

The advantages of using the Linked list are given as follows -

- **Dynamic data structure** - The size of the linked list may vary according to the requirements. Linked list does not have a fixed size.
 - **Insertion and deletion** - Unlike arrays, insertion, and deletion in linked list is easier. Array elements are stored in the consecutive location, whereas the elements in the linked list are stored at a random location. To insert or delete an element in an array, we have to shift the elements for creating the space. Whereas, in linked list, instead of shifting, we just have to update the address of the pointer of the node.
 - **Memory efficient** - The size of a linked list can grow or shrink according to the requirements, so memory consumption in linked list is efficient.
 - **Implementation** - We can implement both stacks and queues using linked list.
-

Disadvantages of Linked list

The limitations of using the Linked list are given as follows -

- **Memory usage** - In linked list, node occupies more memory than array. Each node of the linked list occupies two types of variables, i.e., one is a simple variable, and another one is the pointer variable.
 - **Traversal** - Traversal is not easy in the linked list. If we have to access an element in the linked list, we cannot access it randomly, while in case of array we can randomly access it by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.
 - **Reverse traversing** - Backtracking or reverse traversing is difficult in a linked list. In a doubly-linked list, it is easier but requires more memory to store the back pointer.
-

Applications of Linked list

The applications of the Linked list are given as follows -

- With the help of a linked list, the polynomials can be represented as well as we can perform the operations on the polynomial.
 - A linked list can be used to represent the sparse matrix.
 - The various operations like student's details, employee's details, or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
 - Using linked list, we can implement stack, queue, tree, and other various data structures.
 - The graph is a collection of edges and vertices, and the graph can be represented as an adjacency matrix and adjacency list. If we want to represent the graph as an adjacency matrix, then it can be implemented as an array. If we want to represent the graph as an adjacency list, then it can be implemented as a linked list.
 - A linked list can be used to implement dynamic memory allocation. The dynamic memory allocation is the memory allocation done at the run-time.
-

Operations performed on Linked list

The basic operations that are supported by a list are mentioned as follows -

Advertisement

- **Insertion** - This operation is performed to add an element into the list.
- **Deletion** - It is performed to delete an operation from the list.
- **Display** - It is performed to display the elements of the list.
- **Search** - It is performed to search an element from the list using the given key.

Complexity of Linked list

Now, let's see the time and space complexity of the linked list for the operations search, insert, and delete.

1. Time Complexity

| Operations | Average case time complexity | Worst-case |
|------------------|------------------------------|------------|
| Insertion | O(1) | O(1) |
| Deletion | O(1) | O(1) |
| Search | O(n) | O(n) |

Where 'n' is the number of nodes in the given tree.

2. Space Complexity

| Operations | Space complexity |
|------------------|------------------|
| Insertion | O(n) |
| Deletion | O(n) |
| Search | O(n) |

The space complexity of linked list is **O(n)**.

Types of Linked List

Before knowing about the types of a linked list, we should know what is **linked list**. So, to know about the linked list, click on the link given below:

Types of Linked list

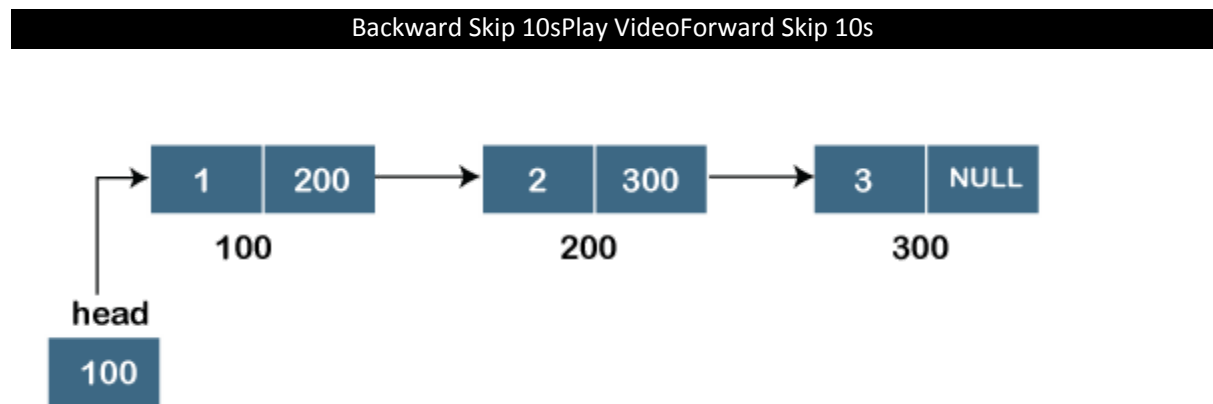
The following are the types of linked list:

- [Singly Linked list](#)
- [Doubly Linked list](#)
- [Circular Linked list](#)
- [Doubly Circular Linked list](#)

Singly Linked list

It is the commonly used linked list in programs. If we are talking about the linked list, it means it is a singly linked list. The singly linked list is a data structure that contains two parts, i.e., one is the data part, and the other one is the address part, which contains the address of the next or the successor node. The address part in a node is also known as a **pointer**.

Suppose we have three nodes, and the addresses of these three nodes are 100, 200 and 300 respectively. The representation of three nodes as a linked list is shown in the below figure:



We can observe in the above figure that there are three different nodes having address 100, 200 and 300 respectively. The first node contains the address of the next node, i.e., 200, the second node contains the address of the last node, i.e., 300, and the third node contains the NULL value in its address part as it does not point to any node. The pointer that holds the address of the initial node is known as a **head pointer**.

The linked list, which is shown in the above diagram, is known as a singly linked list as it contains only a single link. In this list, only forward traversal is possible; we cannot traverse in the backward direction as it has only one link in the list.

Representation of the node in a singly linked list

1. struct node
2. {
3. int data;
4. struct node *next;
5. }

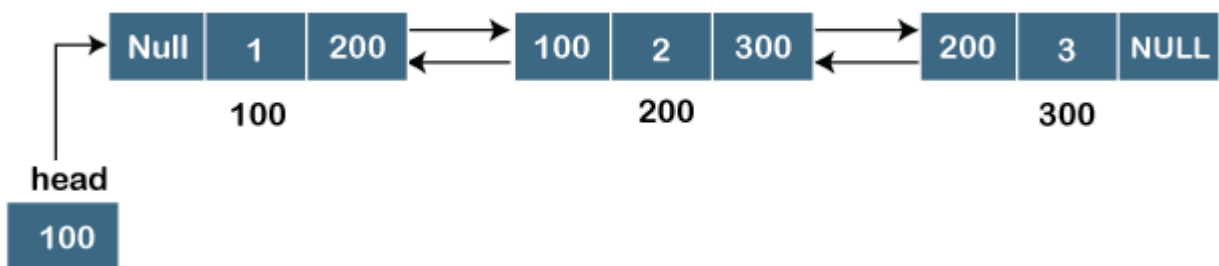
In the above representation, we have defined a user-defined structure named a **node** containing two members, the first one is data of integer type, and the other one is the pointer (next) of the node type.

To know more about a singly linked list, click on the link given below:

Doubly linked list

As the name suggests, the doubly linked list contains two pointers. We can define the doubly linked list as a linear data structure with three parts: the data part and the other two address part. In other words, a doubly linked list is a list that has three parts in a single node, includes one data part, a pointer to its previous node, and a pointer to the next node.

Suppose we have three nodes, and the address of these nodes are 100, 200 and 300, respectively. The representation of these nodes in a doubly-linked list is shown below:



As we can observe in the above figure, the node in a doubly-linked list has two address parts; one part stores the **address of the next** while the other part of the node stores the **previous node's address**. The initial node in the doubly linked list has the **NULL** value in the address part, which provides the address of the previous node.

Representation of the node in a doubly linked list

1. struct node
2. {
3. int data;
4. struct node *next;
5. struct node *prev;
6. }

In the above representation, we have defined a user-defined structure named **a node** with three members, one is **data** of integer type, and the other two are the pointers, i.e., **next** and **prev** of the node type. The **next pointer** variable holds the address of the next node, and the **prev pointer** holds the address of the previous node. The type of both the pointers, i.e., **next** and **prev** is **struct node** as both the pointers are storing the address of the node of the **struct node** type.

To know more about doubly linked list, click on the link given below:

Advertisement

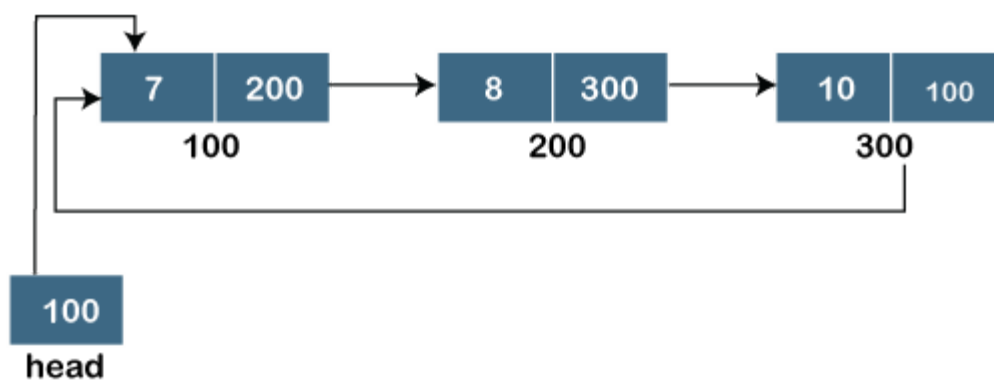
Circular linked list

A circular linked list is a variation of a singly linked list. The only difference between the **singly linked list** and a **circular linked list** is that the last node does not point to any node in a singly linked list, so its link part contains a NULL value. On the other hand,

the circular linked list is a list in which the last node connects to the first node, so the link part of the last node holds the first node's address. The circular linked list has no starting and ending node. We can traverse in any direction, i.e., either backward or forward. The diagrammatic representation of the circular linked list is shown below:

1. struct node
2. {
3. int data;
4. struct node *next;
5. }

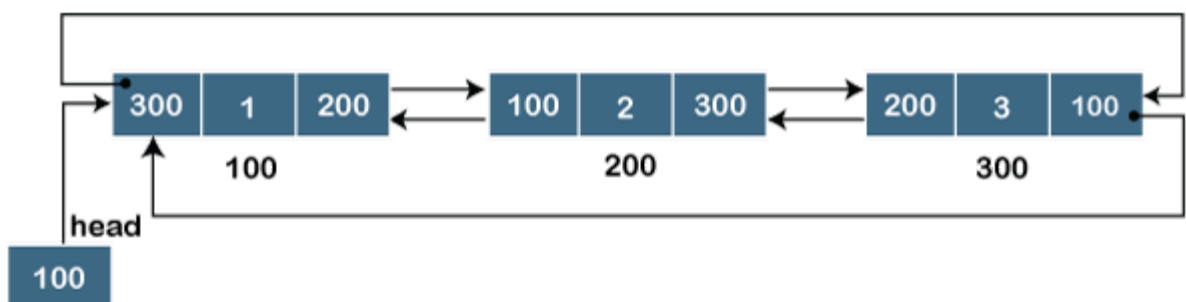
A circular linked list is a sequence of elements in which each node has a link to the next node, and the last node is having a link to the first node. The representation of the circular linked list will be similar to the singly linked list, as shown below:



To know more about the circular linked list, click on the link given below:

Doubly Circular linked list

The doubly circular linked list has the features of both the *circular linked list* and *doubly linked list*.



Advertisement

The above figure shows the representation of the doubly circular linked list in which the last node is attached to the first node and thus creates a circle. It is a doubly linked list also because each node holds the address of the previous node also. The main difference between the doubly linked list and doubly circular linked list is that the doubly

circular linked list does not contain the NULL value in the previous field of the node. As the doubly circular linked contains three parts, i.e., two address parts and one data part so its representation is similar to the doubly linked list.

1. struct node
2. {
3. **int** data;
4. struct node *next;
5. struct node *prev;
6. }

Singly linked list in C

Data structures are crucial elements in computer programming because they make **data handling** and storage efficient. The **linked list** is a typical data structure. In this blog post, we will examine the concept of a **single linked list** in the C programming language. We'll go over its **operations, definition, and example code syntax and outputs**.

Each node in a singly linked list has a data element and a reference to the node after it in the list, making it a linear data structure. The **list's head node** is the first node, while the last node points to **NULL** to denote the list's end.

A structure for each list node must be constructed to define a single linked list in C. The structure should have two fields: a **data field** for storing the actual data and a **pointer field** for holding a reference to the subsequent node.

1. **struct** Node {
2. **int** data;
3. **struct** Node* next;
4. };

Test it Now

Compiler

Run *Compilation Error /usr/bin/ld: /usr/lib/x86_64-linux-gnu/crt1.o: in function `_start': (.text+0x20): undefined reference to `main' collect2: error: ld returned 1 exit status*

We must first initialize the head reference to **NULL**, which denotes an empty list, in order to establish a single linked list. After that, we may add nodes to the list by **dynamically allocating memory** for each node and connecting them with the next pointer.

1. **struct** Node* head = NULL; // Initializing an empty list
- 2.
3. **void** insert(**int** value) {


```

4.   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
5.  newNode->data = value;
6.  newNode->next = NULL;
7.
8.   if (head == NULL) {
9.     head = newNode;
10.  } else {
11.    struct Node* current = head;
12.    while (current->next != NULL) {
13.      current = current->next;
14.    }
15.    current->next = newNode;
16.  }
17. }

```

Test it Now

Compiler

Run

Syntax Error !!! Please check your Code

We must first initialize the head reference to **NULL**, which denotes an empty list, in order to establish a single linked list. After that, we may add nodes to the list by dynamically allocating memory for each node and connecting them with the next pointer.

```

1. void traverse() {
2.   struct Node* current = head;
3.   while (current != NULL) {
4.     printf("%d ", current->data);
5.     current = current->next;
6.   }
7. }

```

Test it Now

Compiler

un

Syntax Error !!! Please check your Code

A singly linked list can be searched for an element by traversing the list and comparing each node's data with the desired value. If a match is made, the relevant node is **returned**; if not, **NULL** is returned to show that the element is not present in the list.

```

1. struct Node* search(int value) {
2.   struct Node* current = head;
3.   while (current != NULL) {
4.     if (current->data == value) {
5.       return current;
6.     }
7.     current = current->next;
8.   }

```

9. **return** NULL;
10. }

Test it Now

Compiler

Run

Syntax Error !!! Please check your Code

To remove a component from a singly linked list, locate the node that contains the desired value and modify the links to the preceding and following nodes accordingly. Additionally, the memory that the destroyed node used must be freed.

1. **void delete**(int value) {
2. **if** (head == NULL) {
3. printf("List is empty.");
4. **return**;
5. }
- 6.
7. **struct** Node* current = head;
8. **struct** Node* previous = NULL;
- 9.
10. **while** (current != NULL) {
11. **if** (current->data == value) {
12. **if** (previous == NULL) {
13. head = current->next;
14. } **else** {
15. previous->next = current->next;
16. }
17. free(current);
18. **return**;
19. }
20. previous = current;
21. current = current->next;
22. }
- 23.
24. printf("Element not found in the list.");
25. }

Test it Now

Compiler



Syntax Error !!! Please check your Code

Example:

Let's create a *list*, *add entries*, and perform various operations on it to show how a single linked list is used.

```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. // Singly Linked List structure
5. struct Node {
6.     int data;
7.     struct Node* next;
8. };
9.
10. // Global head pointer
11. struct Node* head = NULL;
12.
13. // Function to insert a node at the end of the list
14. void insert(int value) {
15.     // Create a new node
16.     struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
17.     newNode->data = value;
18.     newNode->next = NULL;
19.
20.     // Check if the list is empty
21.     if (head == NULL) {
22.         head = newNode;
23.     } else {
24.         // Traverse to the end of the list
25.         struct Node* current = head;
26.         while (current->next != NULL) {
27.             current = current->next;
28.         }
29.         // Link the new node to the last node
30.         current->next = newNode;
31.     }
32. }
33.
34. // Function to traverse and print the list
35. void traverse() {
36.     struct Node* current = head;
37.     while (current != NULL) {
38.         printf("%d ", current->data);
39.         current = current->next;
40.     }
41. }
42.
43. int main() {
44.     // Insert elements into the list
45.     insert(10);
46.     insert(20);
47.     insert(30);
```

```

48. insert(40);
49.
50. // Traverse and print the list
51. printf("List: ");
52. traverse();
53.
54. return 0;
55. }

```

Test it Now

Compiler

Run

List: 10 20 30 40

List: 10 20 30 40

Single linked list program

```

1. #include<stdio.h>
2. #include<stdlib.h>
3. struct node
4. {
5.     int data;
6.     struct node *next;
7. };
8. struct node *head;
9.
10. void begininsert ();
11. void lastinsert ();
12. void randominsert();
13. void begin_delete();
14. void last_delete();
15. void random_delete();
16. void display();
17. void search();
18. void main ()
19. {
20.     int choice =0;
21.     while(choice != 9)
22.     {
23.         printf("\n\n*****Main Menu*****\n");
24.         printf("\nChoose one option from the following list ...\n");
25.         printf("\n=====

```

26.

```
printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random location\n4
.Delete from Beginning\n
```

27.

```
5.Delete from last\n6.Delete node after specified location\n7.Search for an ele
ment\n8.Show\n9.Exit\n");
```

```
28. printf("\nEnter your choice?\n");
```

```
29. scanf("\n%d",&choice);
```

```
30. switch(choice)
```

```
31. {
```

```
32.     case 1:
```

```
33.         beginsert();
```

```
34.         break;
```

```
35.     case 2:
```

```
36.         lastinsert();
```

```
37.         break;
```

```
38.     case 3:
```

```
39.         randominsert();
```

```
40.         break;
```

```
41.     case 4:
```

```
42.         begin_delete();
```

```
43.         break;
```

```
44.     case 5:
```

```
45.         last_delete();
```

```
46.         break;
```

```
47.     case 6:
```

```
48.         random_delete();
```

```
49.         break;
```

```
50.     case 7:
```

```
51.         search();
```

```
52.         break;
```

```
53.     case 8:
```

```
54.         display();
```

```
55.         break;
```

```
56.     case 9:
```

```
57.         exit(0);
```

```
58.         break;
```

```
59.     default:
```

```
60.         printf("Please enter valid choice..");
```

```
61.     }
```

```
62. }
```

```
63. }
```

```
64. void beginsert()
```

```
65. {
```

```
66.     struct node *ptr;
```

```
67.     int item;
```

```
68.     ptr = (struct node *) malloc(sizeof(struct node *));
```

```
69.     if(ptr == NULL)
```

```

70. {
71.     printf("\nOVERFLOW");
72. }
73. else
74. {
75.     printf("\nEnter value\n");
76.     scanf("%d",&item);
77.     ptr->data = item;
78.     ptr->next = head;
79.     head = ptr;
80.     printf("\nNode inserted");
81. }
82.
83.}
84. void lastinsert()
85. {
86.     struct node *ptr,*temp;
87.     int item;
88.     ptr = (struct node*)malloc(sizeof(struct node));
89.     if(ptr == NULL)
90.     {
91.         printf("\nOVERFLOW");
92.     }
93.     else
94.     {
95.         printf("\nEnter value?\n");
96.         scanf("%d",&item);
97.         ptr->data = item;
98.         if(head == NULL)
99.         {
100.             ptr -> next = NULL;
101.             head = ptr;
102.             printf("\nNode inserted");
103.         }
104.         else
105.         {
106.             temp = head;
107.             while (temp -> next != NULL)
108.             {
109.                 temp = temp -> next;
110.             }
111.             temp->next = ptr;
112.             ptr->next = NULL;
113.             printf("\nNode inserted");
114.         }
115.     }
116. }
117. }
118. void randominsert()
119. {

```

```

120.     int i,loc,item;
121.     struct node *ptr, *temp;
122.     ptr = (struct node *) malloc (size of(struct node));
123.     if(ptr == NULL)
124.     {
125.         printf("\nOVERFLOW");
126.     }
127.     else
128.     {
129.         printf("\nEnter element value");
130.         scanf("%d",&item);
131.         ptr->data = item;
132.         printf("\nEnter the location after which you want to insert ");
133.         scanf("\n%d",&loc);
134.         temp=head;
135.         for(i=0;i<loc;i++)
136.         {
137.             temp = temp->next;
138.             if(temp == NULL)
139.             {
140.                 printf("\ncan't insert\n");
141.                 return;
142.             }
143.
144.         }
145.         ptr ->next = temp ->next;
146.         temp ->next = ptr;
147.         printf("\nNode inserted");
148.     }
149. }
150. void begin_delete()
151. {
152.     struct node *ptr;
153.     if(head == NULL)
154.     {
155.         printf("\nList is empty\n");
156.     }
157.     else
158.     {
159.         ptr = head;
160.         head = ptr->next;
161.         free(ptr);
162.         printf("\nNode deleted from the begining ...\n");
163.     }
164. }
165. void last_delete()
166. {
167.     struct node *ptr,*ptr1;
168.     if(head == NULL)
169.     {

```

```

170.     printf("\nlist is empty");
171. }
172. else if(head -> next == NULL)
173. {
174.     head = NULL;
175.     free(head);
176.     printf("\nOnly node of the list deleted ...\n");
177. }
178.
179. else
180. {
181.     ptr = head;
182.     while(ptr->next != NULL)
183.     {
184.         ptr1 = ptr;
185.         ptr = ptr ->next;
186.     }
187.     ptr1->next = NULL;
188.     free(ptr);
189.     printf("\nDeleted Node from the last ...\n");
190. }
191. }
192. void random_delete()
193. {
194.     struct node *ptr,*ptr1;
195.     int loc,i;
196.

```

printf("\n Enter the location of the node after which you want to perform deletion \n");

```

197.     scanf("%d",&loc);
198.     ptr=head;
199.     for(i=0;i<loc;i++)
200.     {
201.         ptr1 = ptr;
202.         ptr = ptr->next;
203.
204.         if(ptr == NULL)
205.         {
206.             printf("\nCan't delete");
207.             return;
208.         }
209.     }
210.     ptr1 ->next = ptr ->next;
211.     free(ptr);
212.     printf("\nDeleted node %d ",loc+1);
213. }
214. void search()
215. {
216.     struct node *ptr;

```



```

217.     int item,i=0,flag;
218.     ptr = head;
219.     if(ptr == NULL)
220.     {
221.         printf("\nEmpty List\n");
222.     }
223.     else
224.     {
225.         printf("\nEnter item which you want to search?\n");
226.         scanf("%d",&item);
227.         while (ptr!=NULL)
228.         {
229.             if(ptr->data == item)
230.             {
231.                 printf("item found at location %d ",i+1);
232.                 flag=0;
233.             }
234.             else
235.             {
236.                 flag=1;
237.             }
238.             i++;
239.             ptr = ptr -> next;
240.         }
241.         if(flag==1)
242.         {
243.             printf("Item not found\n");
244.         }
245.     }
246. }
247.
248.
249. void display()
250. {
251.     struct node *ptr;
252.     ptr = head;
253.     if(ptr == NULL)
254.     {
255.         printf("Nothing to print");
256.     }
257.     else
258.     {
259.         printf("\nprinting values . . . . \n");
260.         while (ptr!=NULL)
261.         {
262.             printf("\n%d",ptr->data);
263.             ptr = ptr -> next;
264.         }
265.     }
266. }

```

267.

Types of Queue

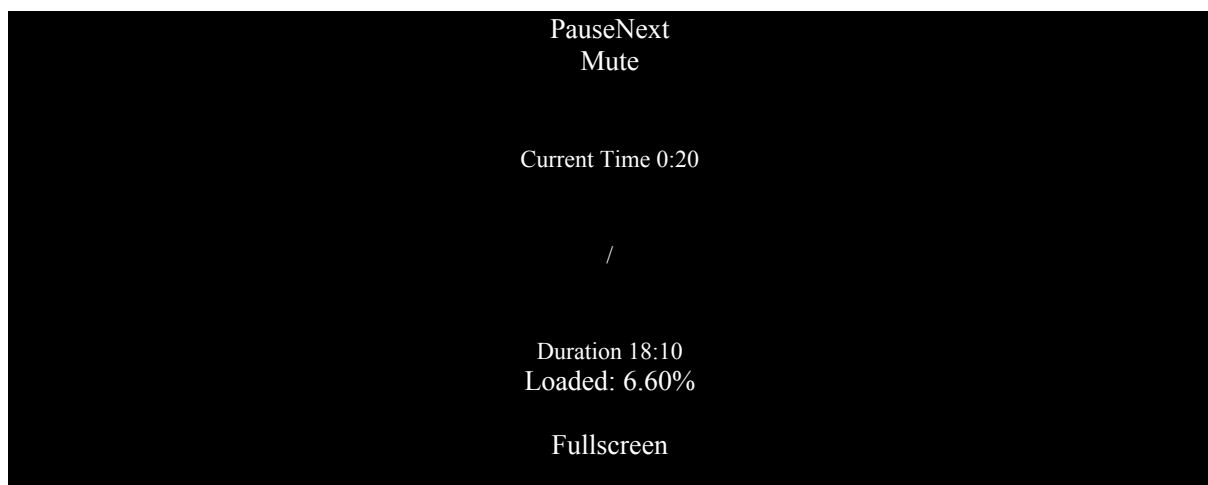
In this article, we will discuss the types of queue. But before moving towards the types, we should first discuss the brief introduction of the queue.

What is a Queue?

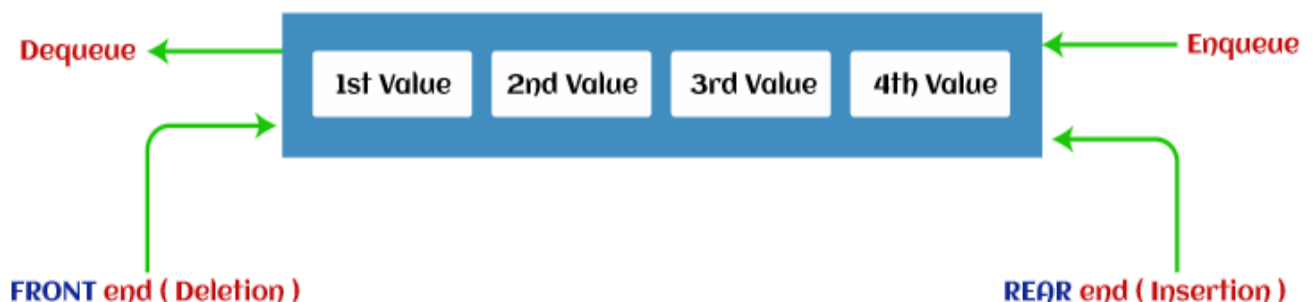
Queue is the data structure that is similar to the queue in the real world. A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy. Queue can also be defined as the list or collection in which the insertion is done from one end known as the **rear end** or the **tail** of the queue, whereas the deletion is done from another end known as the **front end** or the **head** of the queue.

The real-world example of a queue is the ticket queue outside a cinema hall, where the person who enters first in the queue gets the ticket first, and the last person enters in the queue gets the ticket at last. Similar approach is followed in the queue in data structure.

The representation of the queue is shown in the below image -



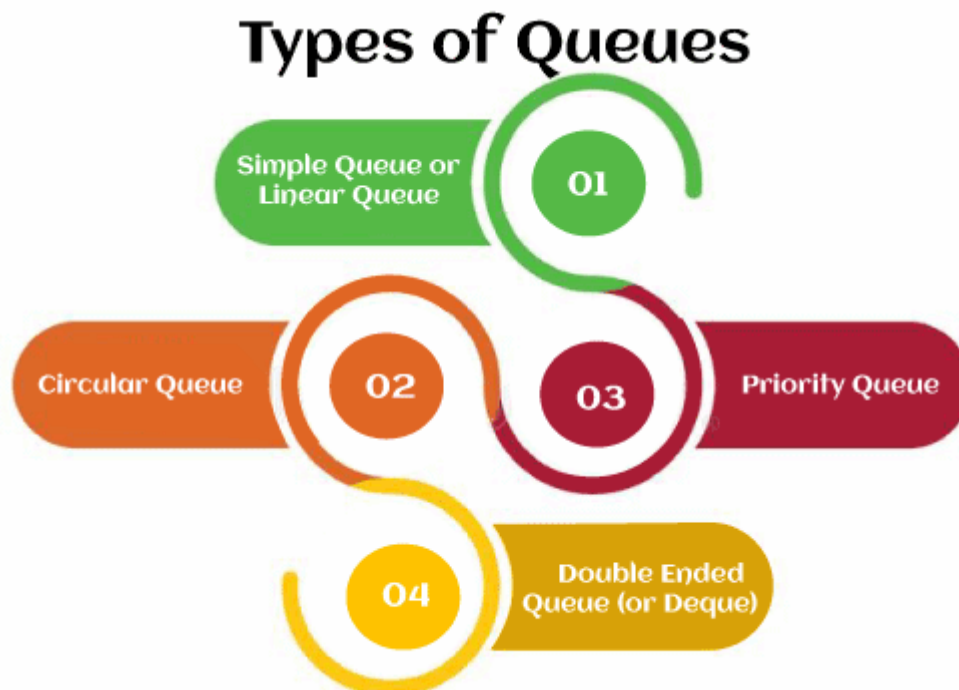
Now, let's move



towards the types of queue.

Types of Queue

There are four different types of queue that are listed as follows -



- Simple Queue or Linear Queue
- Circular Queue
- Priority Queue
- Double Ended Queue (or DE queue)

Let's discuss each of the type of queue.

Simple Queue or Linear Queue

In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.



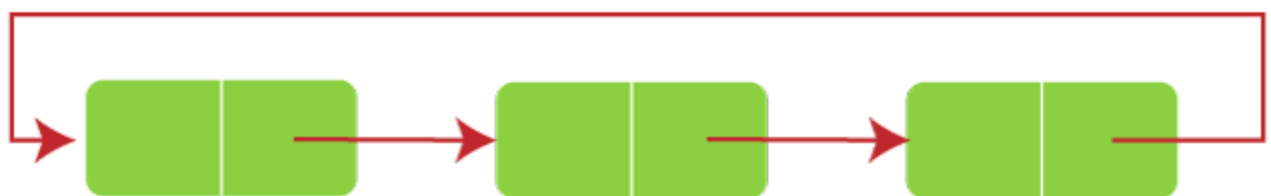
The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

To know more about the queue in data structure, you can click the link - <https://www.javatpoint.com/data-structure-queue>

Advertisement

Circular Queue

In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as Ring Buffer, as all the ends are connected to another end. The representation of circular queue is shown in the below image -



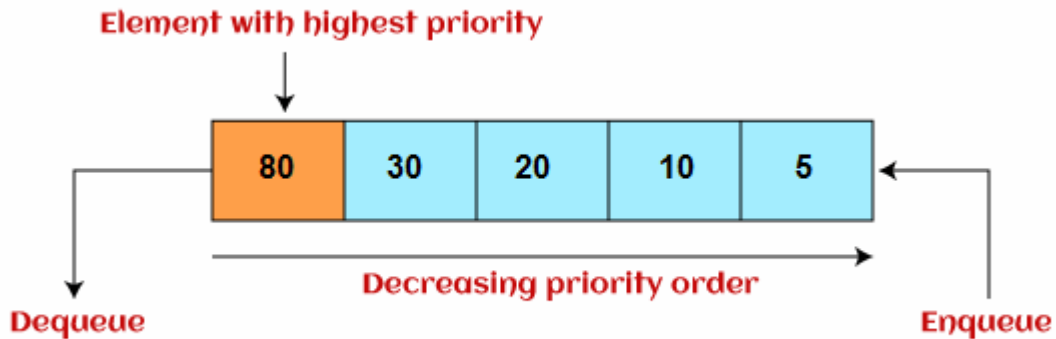
Circular Queue

The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear. The main advantage of using the circular queue is better memory utilization.

To know more about the circular queue, you can click the link - <https://www.javatpoint.com/circular-queue>

Priority Queue

It is a special type of queue in which the elements are arranged based on the priority. It is a special type of queue data structure in which every element has a priority associated with it. Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle. The representation of priority queue is shown in the below image -



Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algorithms.

There are two types of priority queue that are discussed as follows -

- **Ascending priority queue** - In ascending priority queue, elements can be inserted in arbitrary order, but only smallest can be deleted first. Suppose an array with elements 7, 5, and 3 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 3, 5, 7.
- **Descending priority queue** - In descending priority queue, elements can be inserted in arbitrary order, but only the largest element can be deleted first. Suppose an array with elements 7, 3, and 5 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 7, 5, 3.

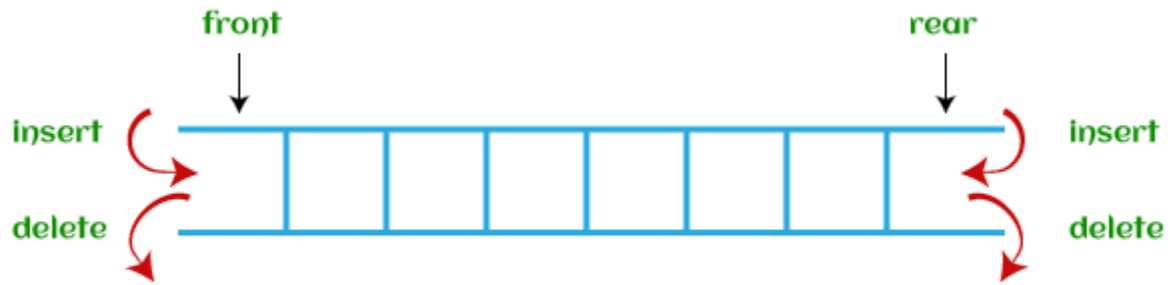
To learn more about the priority queue, you can click the link - <https://www.javatpoint.com/ds-priority-queue>

Deque (or, Double Ended Queue)

In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear. It means that we can insert and delete elements from both front and rear ends of the queue. Deque can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.

Deque can be used both as stack and queue as it allows the insertion and deletion operations on both ends. Deque can be considered as stack because stack follows the LIFO (Last In First Out) principle in which insertion and deletion both can be performed only from one end. And in deque, it is possible to perform both insertion and deletion from one end, and Deque does not follow the FIFO principle.

The representation of the deque is shown in the below image -

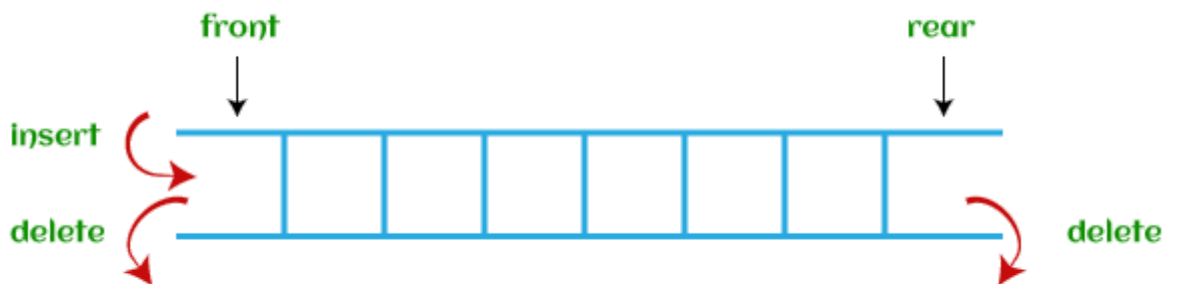


double ended queue

To know more about the DQueue, you can click the link - <https://www.javatpoint.com/ds-deque>

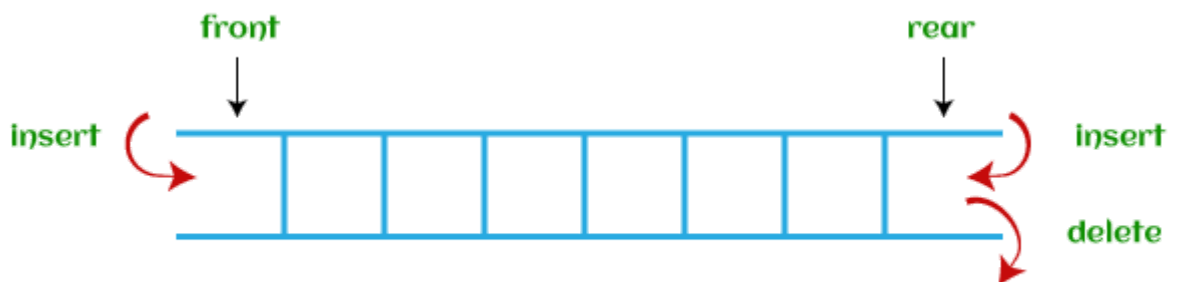
There are two types of deque that are discussed as follows -

- **Input restricted deque** - As the name implies, in input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



input restricted double ended queue

- **Output restricted deque** - As the name implies, in output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Output restricted double ended queue

Now, let's see the operations performed on the queue.

Operations performed on queue

The fundamental operations that can be performed on queue are listed as follows -

- **Enqueue:** The Enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **Dequeue:** It performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value.
- **Peek:** This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
- **Queue overflow (isfull):** It shows the overflow condition when the queue is completely full.
- **Queue underflow (isempty):** It shows the underflow condition when the Queue is empty, i.e., no elements are in the Queue.

Now, let's see the ways to implement the queue.

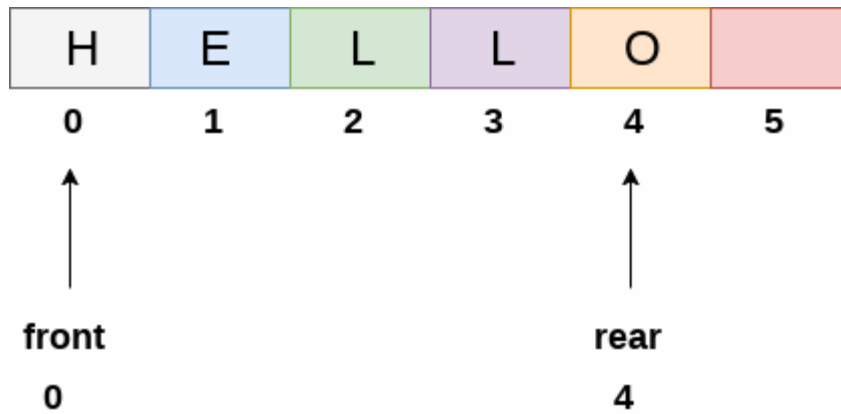
Ways to implement the queue

There are two ways of implementing the Queue:

- **Implementation using array:** The sequential allocation in a Queue can be implemented using an array. For more details, click on the below link: <https://www.javatpoint.com/array-representation-of-queue>
- **Implementation using Linked list:** The linked list allocation in a Queue can be implemented using a linked list. For more details, click on the below link: <https://www.javatpoint.com/linked-list-implementation-of-queue>

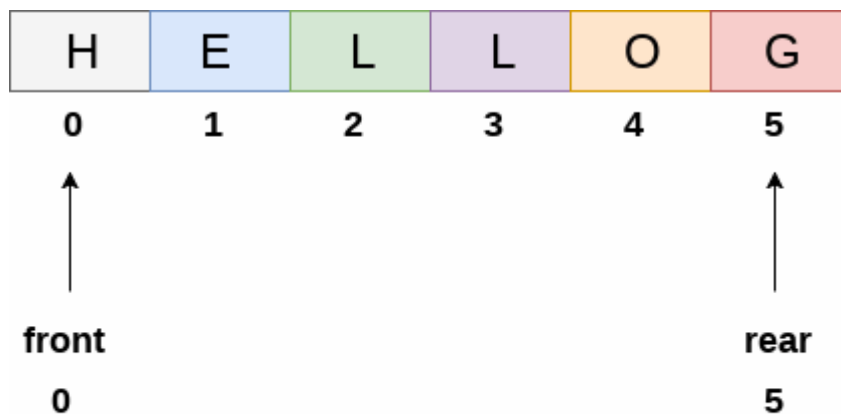
Array representation of Queue

We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



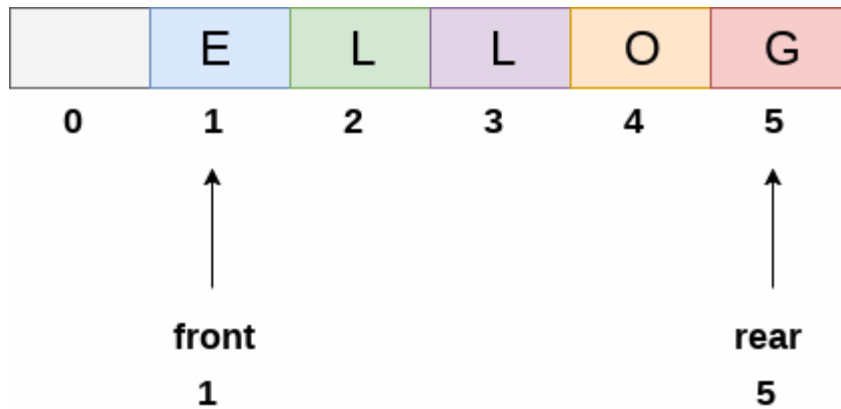
Queue

The above figure shows the queue of characters forming the English word "HELLO". Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

- **Step 1:** IF REAR = MAX - 1
Write OVERFLOW
Go to step
[END OF IF]
- **Step 2:** IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE
SET REAR = REAR + 1
[END OF IF]
- **Step 3:** Set QUEUE[REAR] = NUM
- **Step 4:** EXIT

C Function

```

1. void insert (int queue[], int max, int front, int rear, int item)
2. {
3.     if (rear + 1 == max)
4.     {
5.         printf("overflow");

```

```

6.  }
7.  else
8.  {
9.      if(front == -1 && rear == -1)
10.     {
11.         front = 0;
12.         rear = 0;
13.     }
14.     else
15.     {
16.         rear = rear + 1;
17.     }
18.     queue[rear]=item;
19. }
20.}

```

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm

- **Step 1:** IF FRONT = -1 or FRONT > REAR
Write UNDERFLOW
ELSE
SET VAL = QUEUE[FRONT]
SET FRONT = FRONT + 1
[END OF IF]
- **Step 2:** EXIT

C Function

```

1. int delete (int queue[], int max, int front, int rear)
2. {
3.     int y;
4.     if (front == -1 || front > rear)
5.     {
6.         printf("underflow");
7.     }
9.     else
10.    {
11.        y = queue[front];
12.        if(front == rear)
13.        {

```

```

14.     front = rear = -1;
15.     else
16.     front = front + 1;
17.
18.     }
19.     return y;
20. }
21.}

```

Menu driven program to implement queue using array

```

1. #include<stdio.h>
2. #include<stdlib.h>
3. #define maxsize 5
4. void insert();
5. void delete();
6. void display();
7. int front = -1, rear = -1;
8. int queue[maxsize];
9. void main ()
10. {
11.     int choice;
12.     while(choice != 4)
13.     {
14.         printf("\n*****Main Menu*****\n");
15.
16.
17.         printf("\n=====
=====
=====\\n");
18.
19.         printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\
n");
20.         printf("\nEnter your choice ?");
21.         scanf("%d",&choice);
22.         switch(choice)
23.         {
24.             case 1:
25.                 insert();
26.                 break;
27.             case 2:
28.                 delete();
29.                 break;
30.             case 3:
31.                 display();
32.                 break;
33.             case 4:
34.                 exit(0);
35.                 break;
36.             default:

```

```
34.     printf("\nEnter valid choice??\n");
35.     }
36. }
37.}
38. void insert()
39. {
40.     int item;
41.     printf("\nEnter the element\n");
42.     scanf("\n%d",&item);
43.     if(rear == maxsize-1)
44.     {
45.         printf("\nOVERFLOW\n");
46.         return;
47.     }
48.     if(front == -1 && rear == -1)
49.     {
50.         front = 0;
51.         rear = 0;
52.     }
53.     else
54.     {
55.         rear = rear+1;
56.     }
57.     queue[rear] = item;
58.     printf("\nValue inserted ");
59.
60. }
61. void delete()
62. {
63.     int item;
64.     if (front == -1 || front > rear)
65.     {
66.         printf("\nUNDERFLOW\n");
67.         return;
68.     }
69.     }
70.     else
71.     {
72.         item = queue[front];
73.         if(front == rear)
74.         {
75.             front = -1;
76.             rear = -1 ;
77.         }
78.         else
79.         {
80.             front = front + 1;
81.         }
82.         printf("\nvalue deleted ");
83.     }
```

```

84.
85.
86.}
87.
88. void display()
89.{
90.  int i;
91.  if(rear == -1)
92.  {
93.      printf("\nEmpty queue\n");
94.  }
95.  else
96.  { printf("\nprinting values ..... \n");
97.      for(i=front;i<=rear;i++)
98.      {
99.          printf("\n%d\n",queue[i]);
100.         }
101.     }
102. }

```

Output:

```
*****Main Menu*****
```

```
=====
```

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?1

Enter the element
123

Value inserted

```
*****Main Menu*****
```

```
=====
```

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?1

Enter the element
90

Value inserted

*****Main Menu*****

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?2

value deleted

*****Main Menu*****

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?3

printing values

90

*****Main Menu*****

=====

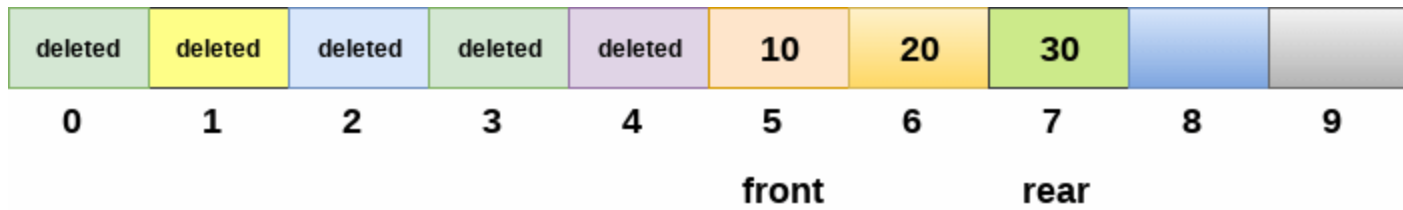
- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?4

Drawback of array implementation

Although, the technique of creating a queue is easy, but there are some drawbacks of using this technique to implement a queue.

- **Memory wastage** : The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.



limitation of array representation of queue

The above figure shows how the memory space is wasted in the array representation of queue. In the above figure, a queue of size 10 having 3 elements, is shown. The value of the front variable is 5, therefore, we can not reinsert the values in the place of already deleted element before the position of front. That much space of the array is wasted and can not be used in the future (for this queue).

- **Deciding the array size**

One of the most common problems with array implementation is the size of the array which requires to be declared in advance. Due to the fact that, the queue can be extended at runtime depending upon the problem, the extension in the array size is a time-taking process and almost impossible to be performed at runtime since a lot of reallocations take place. Due to this reason, we can declare the array large enough so that we can store queue elements as much as possible but the main problem with this declaration is that, most of the array slots (nearly half) can never be reused. It will again lead to memory wastage.

What is a priority queue?

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.

- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Let's understand the priority queue through an example.

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

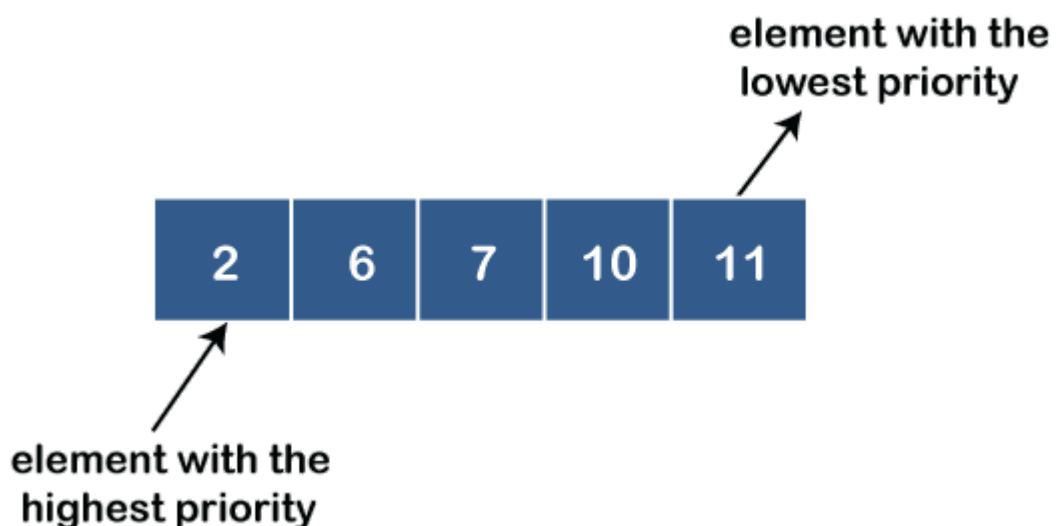
All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

- **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- **add(2):** This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.
- **poll():** It will remove '2' element from the priority queue as it has the highest priority.
- **add(5):** It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

Types of Priority Queue

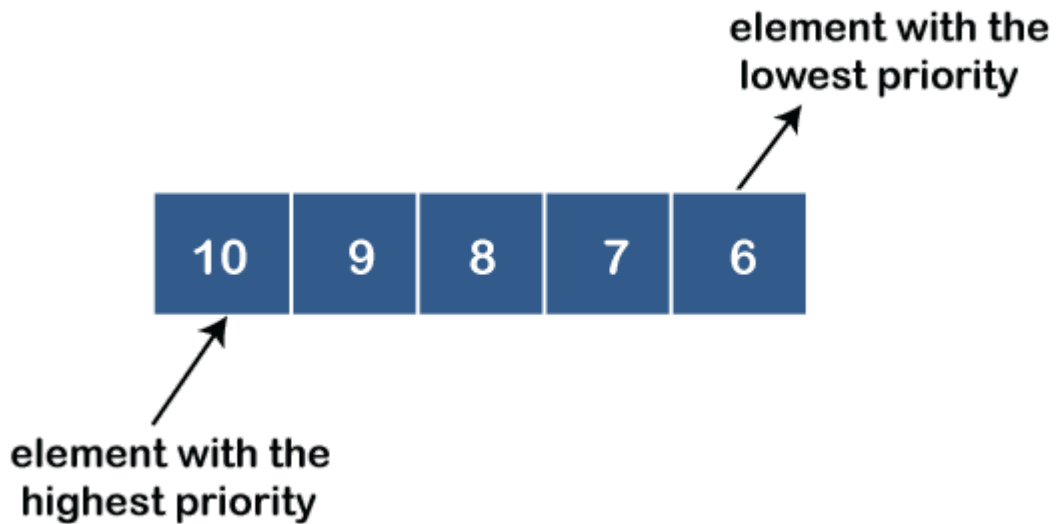
There are two types of priority queue:

- **Ascending order priority queue:** In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.



- **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2,

1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.



Representation of priority queue

Now, we will see how to represent the priority queue through a one-way list.

We will create the priority queue by using the list given below in which **INFO** list contains the data elements, **PNR** list contains the priority numbers of each data element available in the **INFO** list, and **LINK** basically contains the address of the next node.

| | INFO | PNR | LINK |
|---|-------------|------------|-------------|
| 0 | 200 | 2 | 4 |
| 1 | 400 | 4 | 2 |
| 2 | 500 | 4 | 6 |
| 3 | 300 | 1 | 0 |
| 4 | 100 | 2 | 5 |
| 5 | 600 | 3 | 1 |
| 6 | 700 | 4 | |

Let's create the priority queue step by step.

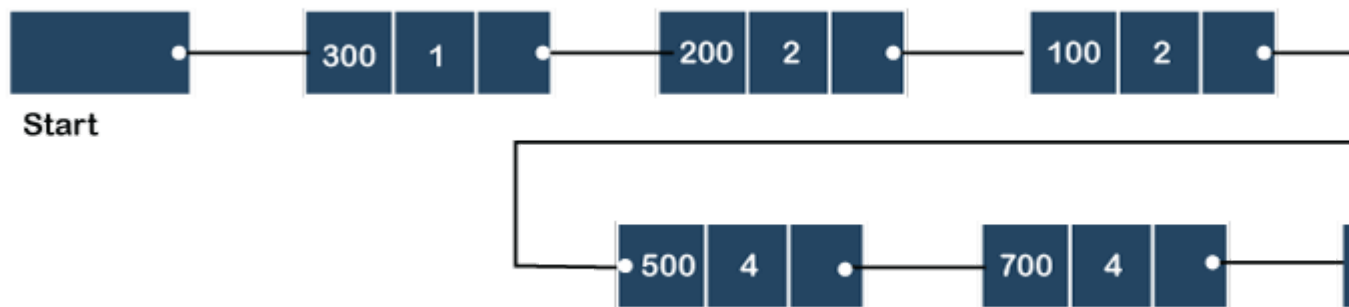
In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.

Step 1: In the list, lower priority number is 1, whose data value is 333, so it will be inserted in the list as shown in the below diagram:

Step 2: After inserting 333, priority number 2 is having a higher priority, and data values associated with this priority are 222 and 111. So, this data will be inserted based on the FIFO principle; therefore 222 will be added first and then 111.

Step 3: After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 444, 555, 777. In this case, elements would be inserted based on the FIFO principle; therefore, 444 will be added first, then 555, and then 777.

Step 4: After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 666, so it will be inserted at the end of the queue.



Implementation of Priority Queue

The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree. The heap data structure is the most efficient way of implementing the priority queue, so we will implement the priority queue using a heap data structure in this topic. Now, first we understand the reason why heap is the most efficient way among all the other data structures.

Analysis of complexities using different implementations

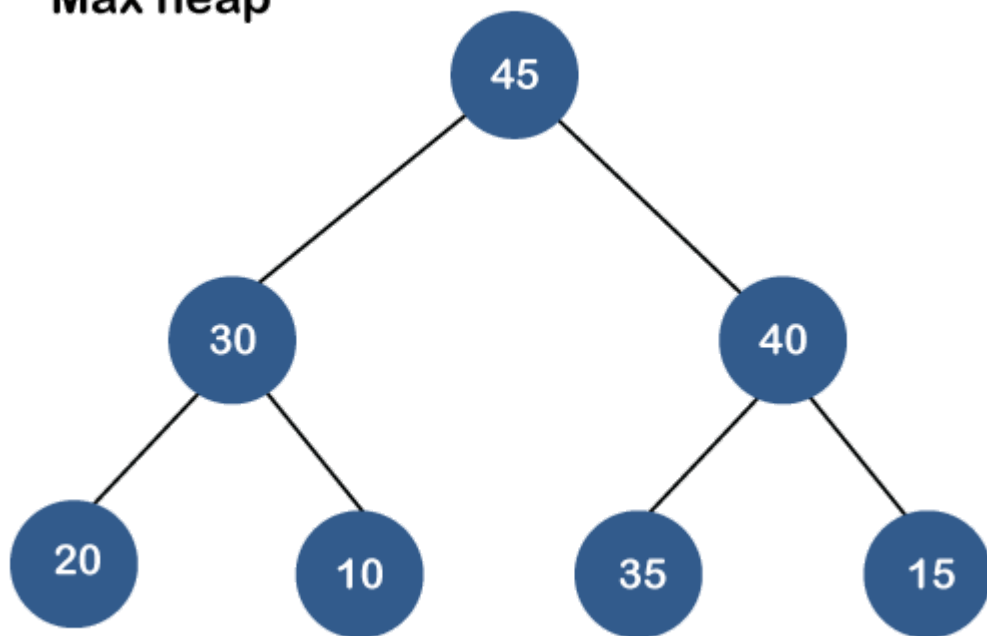
| Implementation | add | Remove | peek |
|--------------------|-------------|-------------|--------|
| Linked list | $O(1)$ | $O(n)$ | $O(n)$ |
| Binary heap | $O(\log n)$ | $O(\log n)$ | $O(1)$ |
| Binary search tree | $O(\log n)$ | $O(\log n)$ | $O(1)$ |

What is Heap?

A heap is a tree-based data structure that forms a complete binary tree, and satisfies the heap property. If A is a parent node of B, then A is ordered with respect to the node B for all nodes A and B in a heap. It means that the value of the parent node could be more than or equal to the value of the child node, or the value of the parent node could be less than or equal to the value of the child node. Therefore, we can say that there are two types of heaps:

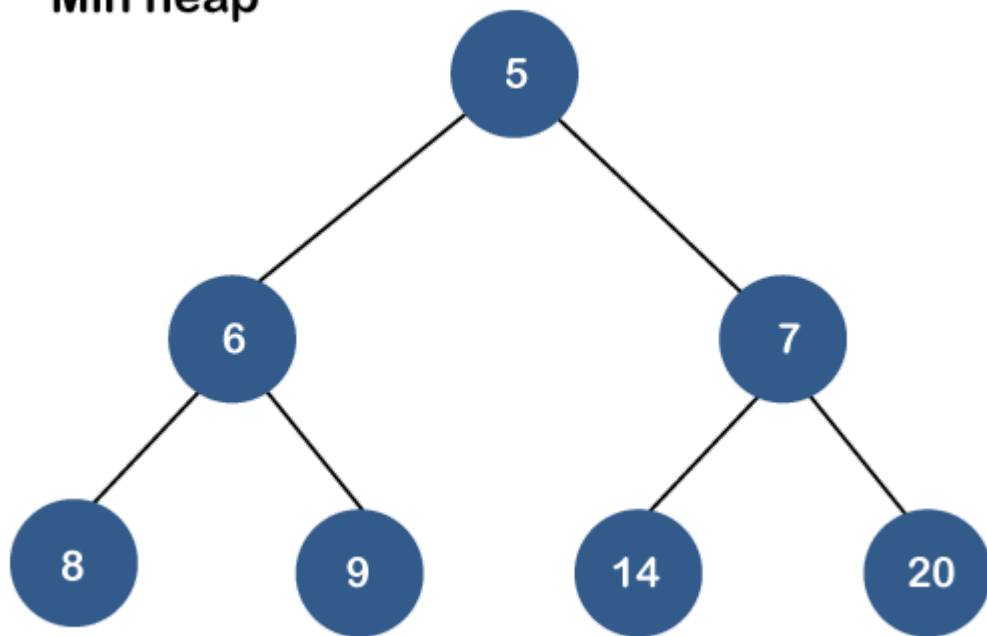
- **Max heap:** The max heap is a heap in which the value of the parent node is greater than the value of the child nodes.

Max heap



- **Min heap:** The min heap is a heap in which the value of the parent node is less than the value of the child nodes.

Min heap



Both the heaps are the binary heap, as each has exactly two child nodes.

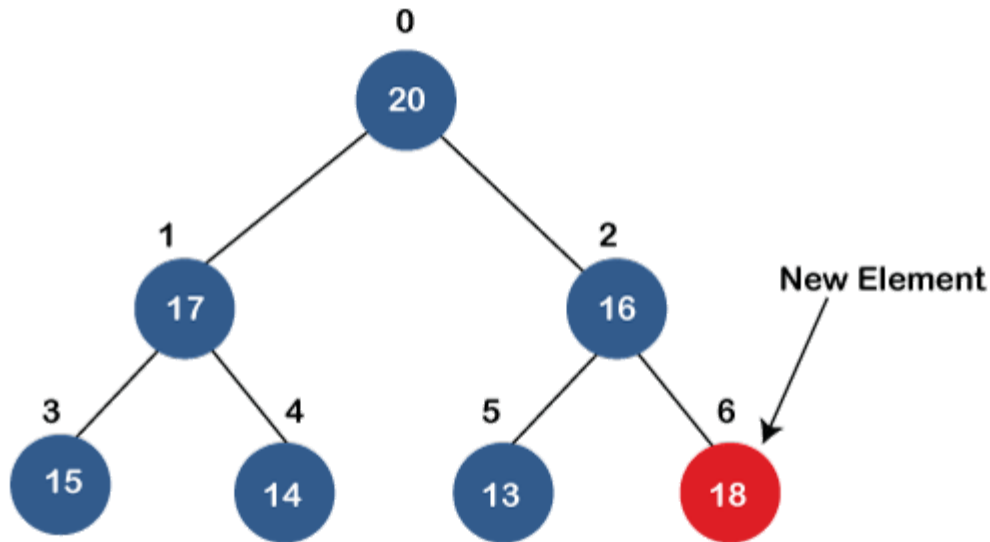
Priority Queue Operations

The common operations that we can perform on a priority queue are insertion, deletion and peek. Let's see how we can maintain the heap data structure.

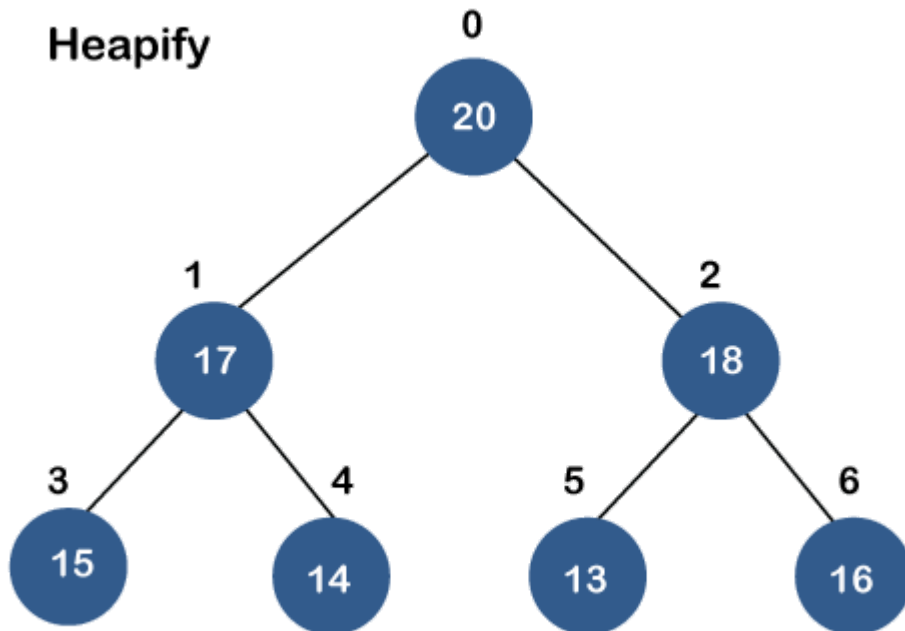
- **Inserting the element in a priority queue (max heap)**

If we insert an element in a priority queue, it will move to the empty slot by looking from top to bottom and left to right.

If the element is not in a correct place then it is compared with the parent node; if it is found out of order, elements are swapped. This process continues until the element is placed in a correct position.



Heapify



- **Removing the minimum element from the priority queue**

As we know that in a max heap, the maximum element is the root node. When we remove the root node, it creates an empty slot. The last inserted element will be added in this empty slot. Then, this element is compared with the child nodes, i.e., left-child and right child, and swap with the smaller of the two. It keeps moving down the tree until the heap property is restored.

Applications of Priority queue

The following are the applications of the priority queue:

- It is used in the Dijkstra's shortest path algorithm.

- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupt handling.

Program to create the priority queue using the binary max heap.

```

1. #include <stdio.h>
2. #include <stdio.h>
3. int heap[40];
4. int size=-1;
5.
6. // retrieving the parent node of the child node
7. int parent(int i)
8. {
9.
10.     return (i - 1) / 2;
11.}
12.
13.// retrieving the left child of the parent node.
14.int left_child(int i)
15.{
16.    return i+1;
17.}
18.// retrieving the right child of the parent
19.int right_child(int i)
20.{
21.    return i+2;
22.}
23.// Returning the element having the highest priority
24.int get_Max()
25.{
26.    return heap[0];
27.}
28.//Returning the element having the minimum priority
29.int get_Min()
30.{
31.    return heap[size];
32.}
33.// function to move the node up the tree in order to restore the heap property.
34.void moveUp(int i)
35.{
36.    while (i > 0)
37.    {
38.        // swapping parent node with a child node
39.        if(heap[parent(i)] < heap[i]) {
40.
41.            int temp;
42.            temp=heap[parent(i)];
43.            heap[parent(i)]=heap[i];

```

```

44.     heap[i]=temp;
45.
46.
47. }
48.     // updating the value of i to i/2
49.     i=i/2;
50. }
51.}
52.
53.//function to move the node down the tree in order to restore the heap property.
54.void moveDown(int k)
55.{
56.     int index = k;
57.
58.     // getting the location of the Left Child
59.     int left = left_child(k);
60.
61.     if (left <= size && heap[left] > heap[index]) {
62.         index = left;
63.     }
64.
65.     // getting the location of the Right Child
66.     int right = right_child(k);
67.
68.     if (right <= size && heap[right] > heap[index]) {
69.         index = right;
70.     }
71.
72.     // If k is not equal to index
73.     if (k != index) {
74.         int temp;
75.         temp=heap[index];
76.         heap[index]=heap[k];
77.         heap[k]=temp;
78.         moveDown(index);
79.     }
80.}
81.
82.// Removing the element of maximum priority
83.void removeMax()
84.{
85.     int r= heap[0];
86.     heap[0]=heap[size];
87.     size=size-1;
88.     moveDown(0);
89.}
90.//inserting the element in a priority queue
91.void insert(int p)
92.{
93.     size = size + 1;

```



```

94. heap[size] = p;
95.
96. // move Up to maintain heap property
97. moveUp(size);
98.}
99.
100. //Removing the element from the priority queue at a given index i.
101. void delete(int i)
102. {
103.     heap[i] = heap[0] + 1;
104.
105.     // move the node stored at ith location is shifted to the root node
106.     moveUp(i);
107.
108.     // Removing the node having maximum priority
109.     removeMax();
110. }
111. int main()
112. {
113.     // Inserting the elements in a priority queue
114.
115.     insert(20);
116.     insert(19);
117.     insert(21);
118.     insert(18);
119.     insert(12);
120.     insert(17);
121.     insert(15);
122.     insert(16);
123.     insert(14);
124.     int i=0;
125.
126.     printf("Elements in a priority queue are : ");
127.     for(int i=0;i<=size;i++)
128.     {
129.         printf("%d ",heap[i]);
130.     }
131.     delete(2); // deleting the element whose index is 2.
132.     printf("\nElements in a priority queue after deleting the element are : ");
133.     for(int i=0;i<=size;i++)
134.     {
135.         printf("%d ",heap[i]);
136.     }
137.     int max=get_Max();
138.     printf("\nThe element which is having the highest priority is %d: ",max);
139.
140.
141.     int min=get_Min();
142.     printf("\nThe element which is having the minimum priority is : %d",min);
143.     return 0;

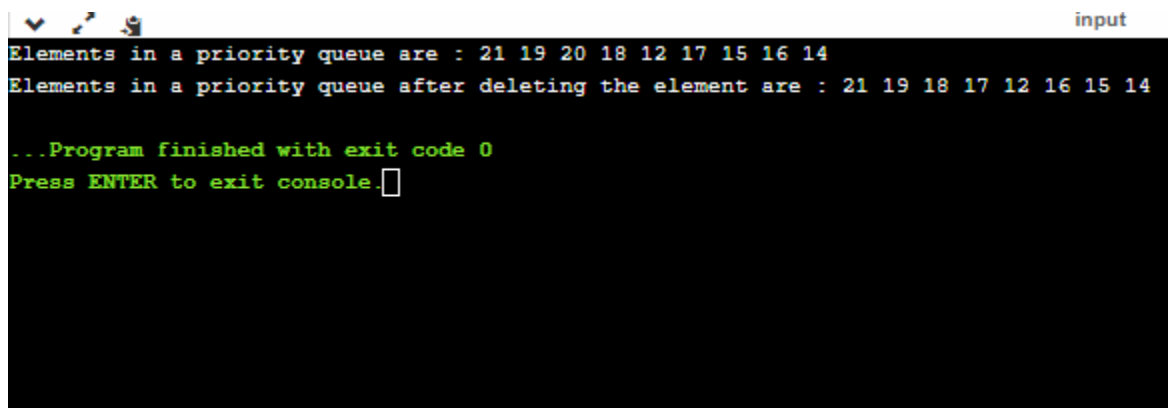
```

144. }

In the above program, we have created the following functions:

- **int parent(int i):** This function returns the index of the parent node of a child node, i.e., i.
- **int left_child(int i):** This function returns the index of the left child of a given index, i.e., i.
- **int right_child(int i):** This function returns the index of the right child of a given index, i.e., i.
- **void moveUp(int i):** This function will keep moving the node up the tree until the heap property is restored.
- **void moveDown(int i):** This function will keep moving the node down the tree until the heap property is restored.
- **void removeMax():** This function removes the element which is having the highest priority.
- **void insert(int p):** It inserts the element in a priority queue which is passed as an argument in a function.
- **void delete(int i):** It deletes the element from a priority queue at a given index.
- **int get_Max():** It returns the element which is having the highest priority, and we know that in max heap, the root node contains the element which has the largest value, and highest priority.
- **int get_Min():** It returns the element which is having the minimum priority, and we know that in max heap, the last node contains the element which has the smallest value, and lowest priority.

Output



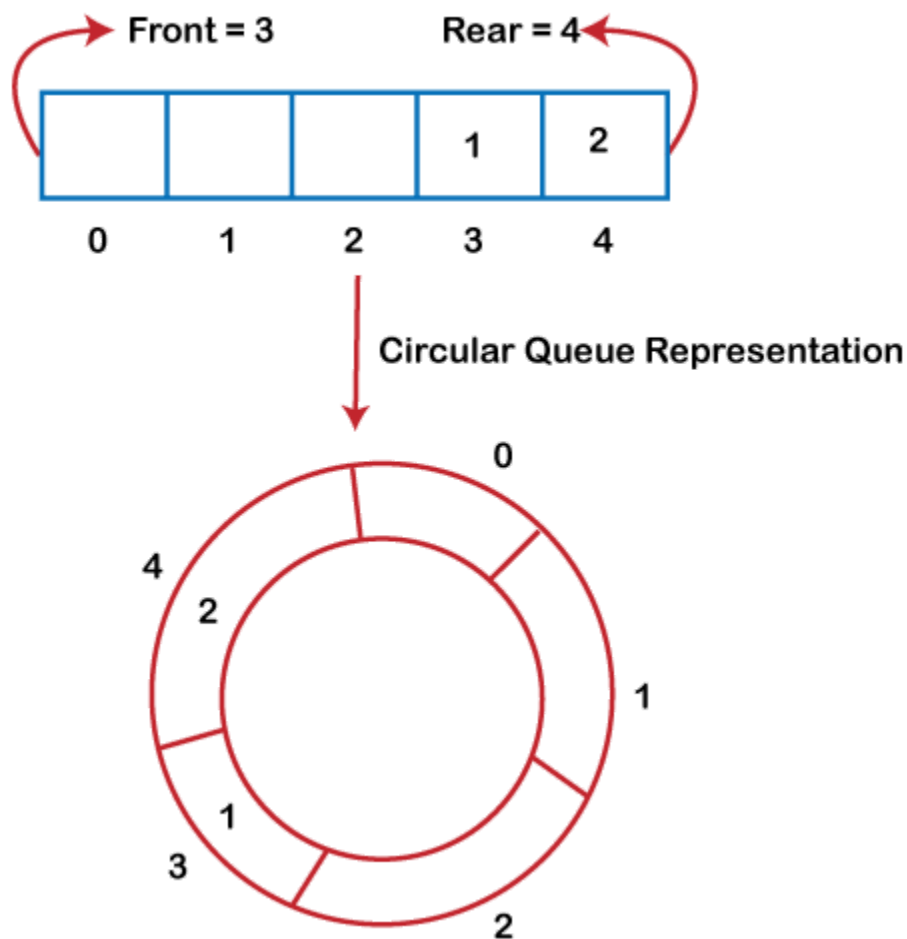
```
input
Elements in a priority queue are : 21 19 20 18 12 17 15 16 14
Elements in a priority queue after deleting the element are : 21 19 18 17 12 16 15 14
...Program finished with exit code 0
Press ENTER to exit console. □
```

Circular Queue

Why was the concept of the circular queue introduced?

There was one limitation in the array implementation of **Queue**. If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are

left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.



As we can see in the above image, the rear is at the last position of the Queue and front is pointing somewhere rather than the 0th position. In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue. There is one solution to avoid such wastage of memory space by shifting both the elements at the left and adjust the front and rear end accordingly. It is not a practically good approach because shifting all the elements will consume lots of time. The efficient approach to avoid the wastage of the memory is to use the circular queue data structure.

What is a Circular Queue?

A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a **Ring Buffer**.

Operations on Circular Queue

The following are the operations that can be performed on a circular queue:

- **Front:** It is used to get the front element from the Queue.
 - **Rear:** It is used to get the rear element from the Queue.
 - **enqueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
 - **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.
-

Applications of Circular Queue

The circular Queue can be used in the following scenarios:

- **Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.
 - **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.
 - **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every interval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.
-

Enqueue operation

The steps of enqueue operation are given below:

- First, we will check whether the Queue is full or not.
 - Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
 - When we insert a new element, the rear gets incremented, i.e., **rear=rear+1**.
-

Scenarios for inserting an element

There are two scenarios in which queue is not full:

- **If rear != max - 1**, then rear will be incremented to **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
 - **If front != 0 and rear = max - 1**, it means that queue is not full, then set the value of rear to 0 and insert the new element there.
-

There are two cases in which the element cannot be inserted:

- When **front == 0 && rear = max-1**, which means that front is at the first position of the Queue and rear is at the last position of the Queue.
 - **front == rear + 1;**
-

Algorithm to insert an element in a circular queue

Step 1: IF (REAR+1)%MAX = FRONT

Write " OVERFLOW "
Goto step 4
[End OF IF]

Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE IF REAR = MAX - 1 and FRONT != 0
SET REAR = 0
ELSE
SET REAR = (REAR + 1) % MAX
[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT

Dequeue Operation

The steps of dequeue operation are given below:

- First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by 1.
- If there is only one element left which is to be deleted, then the front and rear are reset to -1.

Algorithm to delete an element from the circular queue

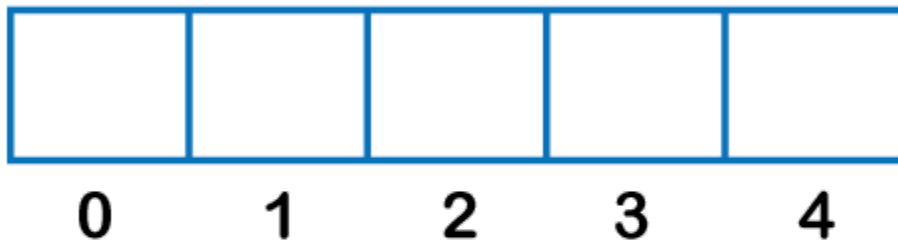
Step 1: IF FRONT = -1
Write "UNDERFLOW "
Goto Step 4
[END of IF]

Step 2: SET VAL = QUEUE[FRONT]

Step 3: IF FRONT = REAR
SET FRONT = REAR = -1
ELSE
IF FRONT = MAX -1
SET FRONT = 0
ELSE
SET FRONT = FRONT + 1
[END of IF]
[END OF IF]

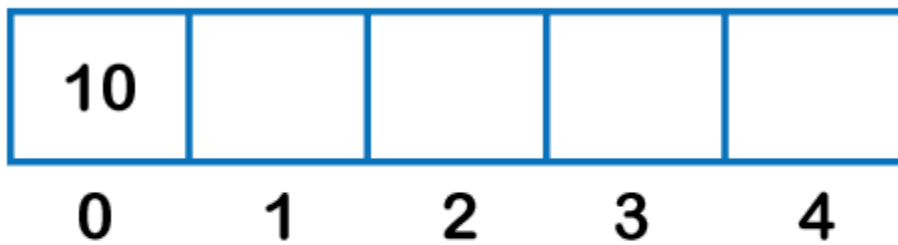
Step 4: EXIT

Let's understand the enqueue and dequeue operation through the diagrammatic representation.



Front = -1

Rear = -1



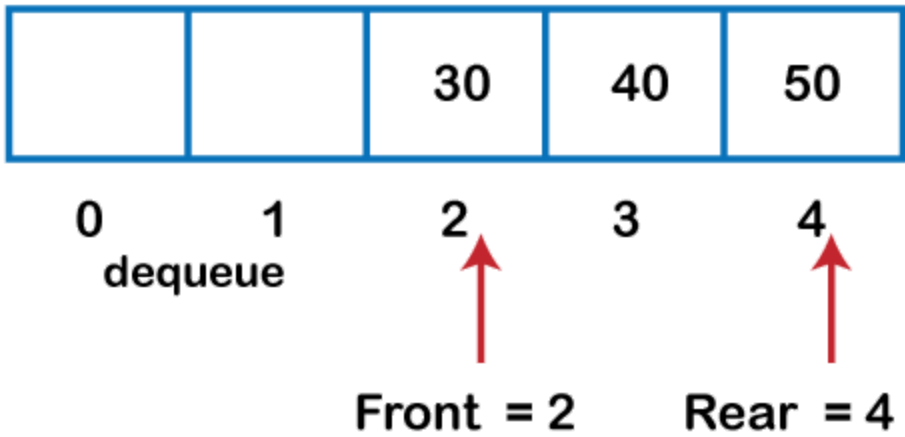
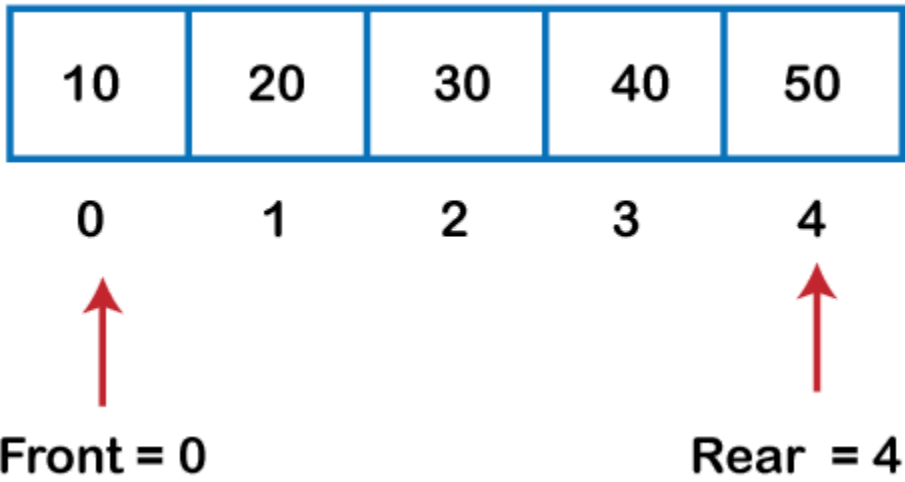
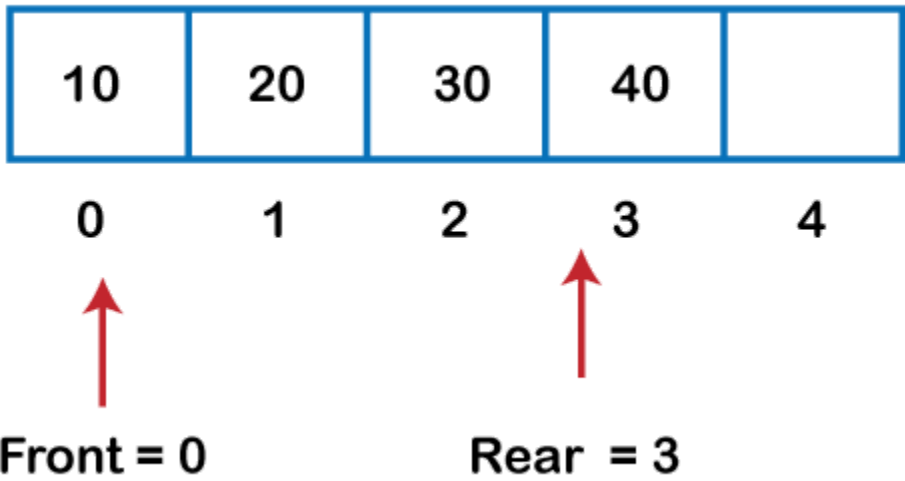
Front = 0

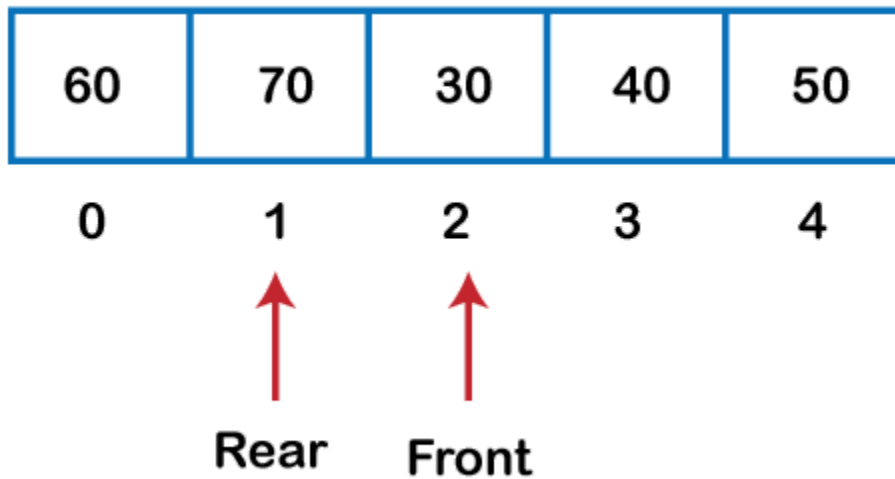
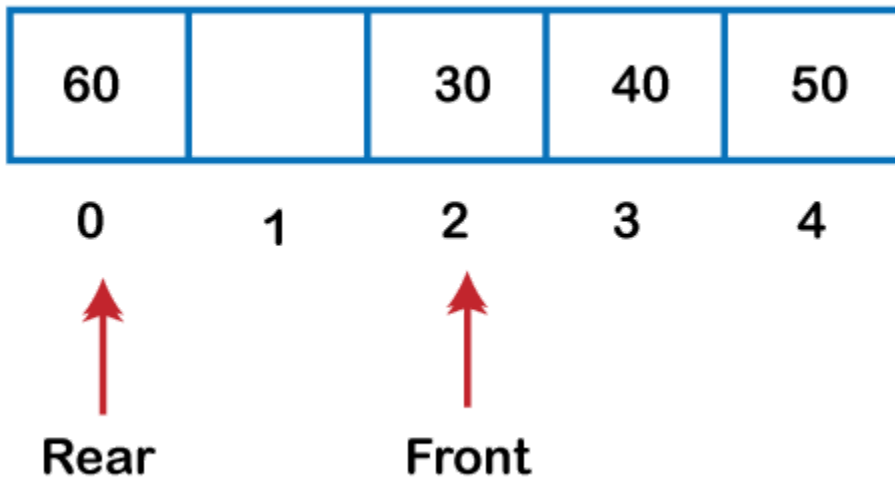
Rear = 0



Front = 0

Rear = 2





Implementation of circular queue using Array

```

1. #include <stdio.h>
2.
3. # define max 6
4. int queue[max]; // array declaration
5. int front=-1;
6. int rear=-1;
7. // function to insert an element in a circular queue
8. void enqueue(int element)
9. {
10.  if(front== -1 && rear== -1) // condition to check queue is empty
11.  {
12.   front=0;
13.   rear=0;
14.   queue[rear]=element;

```



```

15. }
16. else if((rear+1)%max==front) // condition to check queue is full
17. {
18.     printf("Queue is overflow..");
19. }
20. else
21. {
22.     rear=(rear+1)%max;    // rear is incremented
23.     queue[rear]=element; // assigning a value to the queue at the rear position.
24. }
25.}
26.
27.// function to delete the element from the queue
28.int dequeue()
29.{
30.    if((front== -1) && (rear== -1)) // condition to check queue is empty
31.    {
32.        printf("\nQueue is underflow..");
33.    }
34.    else if(front==rear)
35.    {
36.        printf("\nThe dequeued element is %d", queue[front]);
37.        front=-1;
38.        rear=-1;
39.    }
40.    else
41.    {
42.        printf("\nThe dequeued element is %d", queue[front]);
43.        front=(front+1)%max;
44.    }
45.}
46.// function to display the elements of a queue
47.void display()
48.{
49.    int i=front;
50.    if(front== -1 && rear== -1)
51.    {
52.        printf("\n Queue is empty..");
53.    }
54.    else
55.    {
56.        printf("\nElements in a Queue are :");
57.        while(i<=rear)
58.        {
59.            printf("%d,", queue[i]);
60.            i=(i+1)%max;
61.        }
62.    }
63.}
64.int main()

```

```
65. {
66.     int choice=1,x; // variables declaration
67.
68.     while(choice<4 && choice!=0) // while loop
69.     {
70.         printf("\n Press 1: Insert an element");
71.         printf("\n Press 2: Delete an element");
72.         printf("\n Press 3: Display the element");
73.         printf("\n Enter your choice");
74.         scanf("%d", &choice);
75.
76.         switch(choice)
77.         {
78.
79.             case 1:
80.
81.                 printf("Enter the element which is to be inserted");
82.                 scanf("%d", &x);
83.                 enqueue(x);
84.                 break;
85.             case 2:
86.                 dequeue();
87.                 break;
88.             case 3:
89.                 display();
90.
91.         }}
92.     return 0;
93. }
```

Output:

```
Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
10

Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
20

Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
30

Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
3

Elements in a Queue are :10,20,30,
Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
2

The dequeued element is 10
```

Implementation of circular queue using linked list

As we know that linked list is a linear data structure that stores two parts, i.e., data part and the address part where address part contains the address of the next node. Here, linked list is used to implement the circular queue; therefore, the linked list follows the properties of the Queue. When we are implementing the circular queue using linked list then both the *enqueue and dequeue* operations take $O(1)$ time.

1. #include <stdio.h>
2. // Declaration of struct type node
3. **struct** node
4. {
5. **int** data;
6. **struct** node *next;

```

7. };
8. struct node *front=-1;
9. struct node *rear=-1;
10. // function to insert the element in the Queue
11. void enqueue(int x)
12. {
13.     struct node *newnode; // declaration of pointer of struct node type.
14.
15.     newnode=(struct node *)malloc(sizeof(struct node)); // allocating the memory to
the newnode
16.     newnode->data=x;
17.     newnode->next=0;
18.     if(rear== -1) // checking whether the Queue is empty or not.
19.     {
20.         front=rear=newnode;
21.         rear->next=front;
22.     }
23.     else
24.     {
25.         rear->next=newnode;
26.         rear=newnode;
27.         rear->next=front;
28.     }
29.
30. // function to delete the element from the queue
31. void dequeue()
32. {
33.     struct node *temp; // declaration of pointer of node type
34.     temp=front;
35.     if((front== -1)&&(rear== -1)) // checking whether the queue is empty or not
36.     {
37.         printf("\nQueue is empty");
38.     }
39.     else if(front==rear) // checking whether the single element is left in the queue
40.     {
41.         front=rear=-1;
42.         free(temp);
43.     }
44.     else
45.     {
46.         front=front->next;
47.         rear->next=front;
48.         free(temp);
49.     }
50. }
51.
52. // function to get the front of the queue
53. int peek()

```

```

54. {
55.     if((front== -1) &&(rear== -1))
56.     {
57.         printf("\nQueue is empty");
58.     }
59.     else
60.     {
61.         printf("\nThe front element is %d", front->data);
62.     }
63. }
64.
65. // function to display all the elements of the queue
66. void display()
67. {
68.     struct node *temp;
69.     temp=front;
70.     printf("\n The elements in a Queue are : ");
71.     if((front== -1) && (rear== -1))
72.     {
73.         printf("Queue is empty");
74.     }
75.
76.     else
77.     {
78.         while(temp->next!=front)
79.         {
80.             printf("%d,", temp->data);
81.             temp=temp->next;
82.         }
83.         printf("%d", temp->data);
84.     }
85. }
86.
87. void main()
88. {
89.     enqueue(34);
90.     enqueue(10);
91.     enqueue(23);
92.     display();
93.     dequeue();
94.     peek();
95. }

```

Deque (or double-ended queue)

In this article, we will discuss the double-ended queue or deque. We should first see a brief description of the queue.

What is a queue?

A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy. Insertion in the queue is done from one end known as the **rear end** or the **tail**, whereas the deletion is done from another end known as the **front end** or the **head** of the queue.

The real-world example of a queue is the ticket queue outside a cinema hall, where the person who enters first in the queue gets the ticket first, and the person enters last in the queue gets the ticket at last.

What is a Deque (or double-ended queue)

The deque stands for Double Ended Queue. Deque is a linear data structure where the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule. The representation of a deque is given as follows -



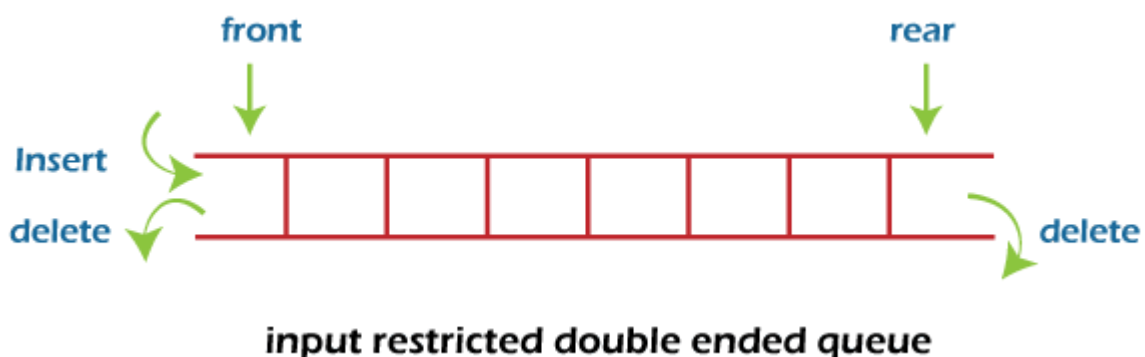
Types of deque

There are two types of deque -

- Input restricted queue
- Output restricted queue

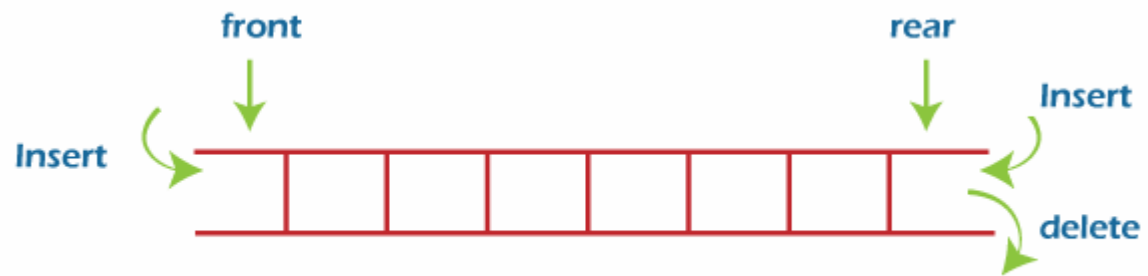
Input restricted Queue

In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



Output restricted Queue

In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



Output restricted double ended queue

Operations performed on deque

There are the following operations that can be applied on a deque -

- Insertion at front
- Insertion at rear
- Deletion at front
- Deletion at rear

We can also perform peek operations in the deque along with the operations listed above. Through peek operation, we can get the deque's front and rear elements of the deque. So, in addition to the above operations, following operations are also supported in deque -

- Get the front item from the deque
- Get the rear item from the deque
- Check whether the deque is full or not
- Checks whether the deque is empty or not

Now, let's understand the operation performed on deque using an example.

Insertion at the front end

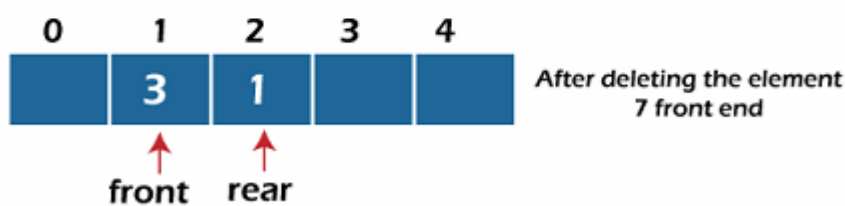
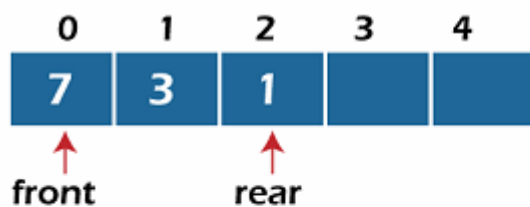
In this operation, the element is inserted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is full or not. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, check the position of the front if the front is less than 1 ($\text{front} < 1$), then reinitialize it by **front = n - 1**, i.e., the last index of the array.

If the deque has only one element, set rear = -1 and front = -1.

Else if front is at end (that means front = size - 1), set front = 0.

Else increment the front by 1, (i.e., front = front + 1).



Deletion at the rear end

In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

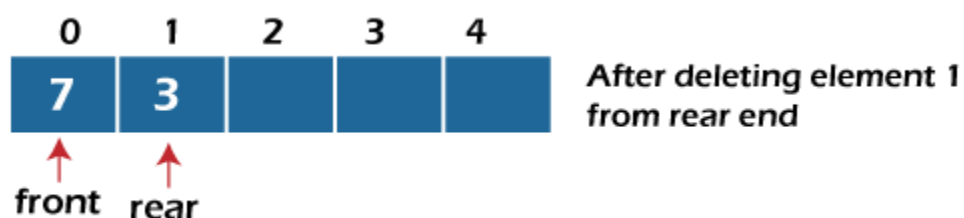
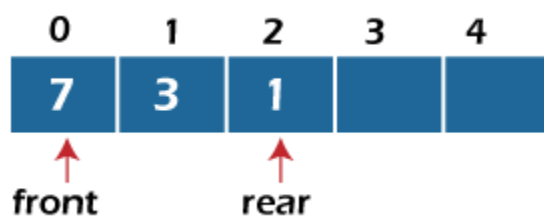
If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion.

If the deque has only one element, set rear = -1 and front = -1.

If rear = 0 (rear is at front), then set rear = n - 1.

Advertisement

Else, decrement the rear by 1 (or, rear = rear - 1).



Check empty

This operation is performed to check whether the deque is empty or not. If front = -1, it means that the deque is empty.

Check full

This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n - 1 it means that the deque is full.

The time complexity of all of the above operations of the deque is $O(1)$, i.e., constant.

Applications of deque

- Deque can be used as both stack and queue, as it supports both operations.
- Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.

Implementation of deque

Now, let's see the implementation of deque in C programming language.

```
1. #include <stdio.h>
2. #define size 5
3. int deque[size];
4. int f = -1, r = -1;
5. // insert_front function will insert the value from the front
6. void insert_front(int x)
7. {
8.     if((f==0 && r==size-1) || (f==r+1))
9.     {
10.         printf("Overflow");
11.     }
12.     else if((f==-1) && (r==-1))
13.     {
14.         f=r=0;
15.         deque[f]=x;
16.     }
17.     else if(f==0)
18.     {
19.         f=size-1;
20.         deque[f]=x;
21.     }
22.     else
23.     {
24.         f=f-1;
25.         deque[f]=x;
26.     }
27. }
28.
29. // insert_rear function will insert the value from the rear
```

```

30. void insert_rear(int x)
31. {
32.     if((f==0 && r==size-1) || (f==r+1))
33.     {
34.         printf("Overflow");
35.     }
36.     else if((f==-1) && (r==-1))
37.     {
38.         r=0;
39.         deque[r]=x;
40.     }
41.     else if(r==size-1)
42.     {
43.         r=0;
44.         deque[r]=x;
45.     }
46.     else
47.     {
48.         r++;
49.         deque[r]=x;
50.     }
51.
52. }
53.
54. // display function prints all the value of deque.
55. void display()
56. {
57.     int i=f;
58.     printf("\nElements in a deque are: ");
59.
60.     while(i!=r)
61.     {
62.         printf("%d ",deque[i]);
63.         i=(i+1)%size;
64.     }
65.     printf("%d",deque[r]);
66. }
67.
68. // getfront function retrieves the first value of the deque.
69. void getfront()
70. {
71.     if((f==-1) && (r==-1))
72.     {
73.         printf("Deque is empty");
74.     }
75.     else
76.     {
77.         printf("\nThe value of the element at front is: %d", deque[f]);
78.     }
79.

```

```

80.}
81.
82.// getrear function retrieves the last value of the deque.
83.void getrear()
84.{
85.    if((f==-1) && (r==-1))
86.    {
87.        printf("Deque is empty");
88.    }
89.    else
90.    {
91.        printf("\nThe value of the element at rear is %d", deque[r]);
92.    }
93.
94.}
95.
96.// delete_front() function deletes the element from the front
97.void delete_front()
98.{
99.    if((f==-1) && (r==-1))
100.    {
101.        printf("Deque is empty");
102.    }
103.    else if(f==r)
104.    {
105.        printf("\nThe deleted element is %d", deque[f]);
106.        f=-1;
107.        r=-1;
108.    }
109.    }
110.    else if(f==(size-1))
111.    {
112.        printf("\nThe deleted element is %d", deque[f]);
113.        f=0;
114.    }
115.    else
116.    {
117.        printf("\nThe deleted element is %d", deque[f]);
118.        f=f+1;
119.    }
120.}
121.
122.// delete_rear() function deletes the element from the rear
123.void delete_rear()
124.{
125.    if((f==-1) && (r==-1))
126.    {
127.        printf("Deque is empty");
128.    }
129.    else if(f==r)

```

```

130.     {
131.         printf("\nThe deleted element is %d", deque[r]);
132.         f=-1;
133.         r=-1;
134.
135.     }
136.     else if(r==0)
137.     {
138.         printf("\nThe deleted element is %d", deque[r]);
139.         r=size-1;
140.     }
141.     else
142.     {
143.         printf("\nThe deleted element is %d", deque[r]);
144.         r=r-1;
145.     }
146. }
147.
148. int main()
149. {
150.     insert_front(20);
151.     insert_front(10);
152.     insert_rear(30);
153.     insert_rear(50);
154.     insert_rear(80);
155.     display(); // Calling the display function to retrieve the values of deque
156.     getfront(); // Retrieve the value at front-end
157.     getrear(); // Retrieve the value at rear-end
158.     delete_front();
159.     delete_rear();
160.     display(); // calling display function to retrieve values after deletion
161.     return 0;

```

Output:

```

Elements in a deque are: 10 20 30 50 80
The value of the element at front is: 10
The value of the element at rear is 80
The deleted element is 10
The deleted element is 80
Elements in a deque are: 20 30 50

```

Output:

```
The elements in a Queue are : 34,10,23  
The front element is 10
```

```
...Program finished with exit code 24  
Press ENTER to exit console.
```

Searching

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

There are two popular search methods that are widely used in order to search some item into the list. However, choice of the algorithm depends upon the arrangement of the list.

- Linear Search
- Binary Search

Linear Search

Linear search is the simplest search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match found then location of the item is returned otherwise the algorithm return NULL.

Linear search is mostly used to search an unordered list in which the items are not sorted. The algorithm of linear search is given as follows.

Algorithm

- LINEAR_SEARCH(A, N, VAL)
- **Step 1:** [INITIALIZE] SET POS = -1
- **Step 2:** [INITIALIZE] SET I = 1
- **Step 3:** Repeat Step 4 while I<=N
- **Step 4:** IF A[I] = VAL
SET POS = I
PRINT POS
Go to Step 6
[END OF IF]
SET I = I + 1
[END OF LOOP]
- **Step 5:** IF POS = -1
PRINT " VALUE IS NOT PRESENTIN THE ARRAY "
[END OF IF]
- **Step 6:** EXIT

Complexity of algorithm

| Complexity | Best Case | Average Case | Worst Case |
|------------|-----------|--------------|------------|
| Time | O(1) | O(n) | O(n) |

list

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

search element **12**

Step 1:

search element (12) is compared with first element (65)

list

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

list

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

list

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

list

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

list

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

list

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

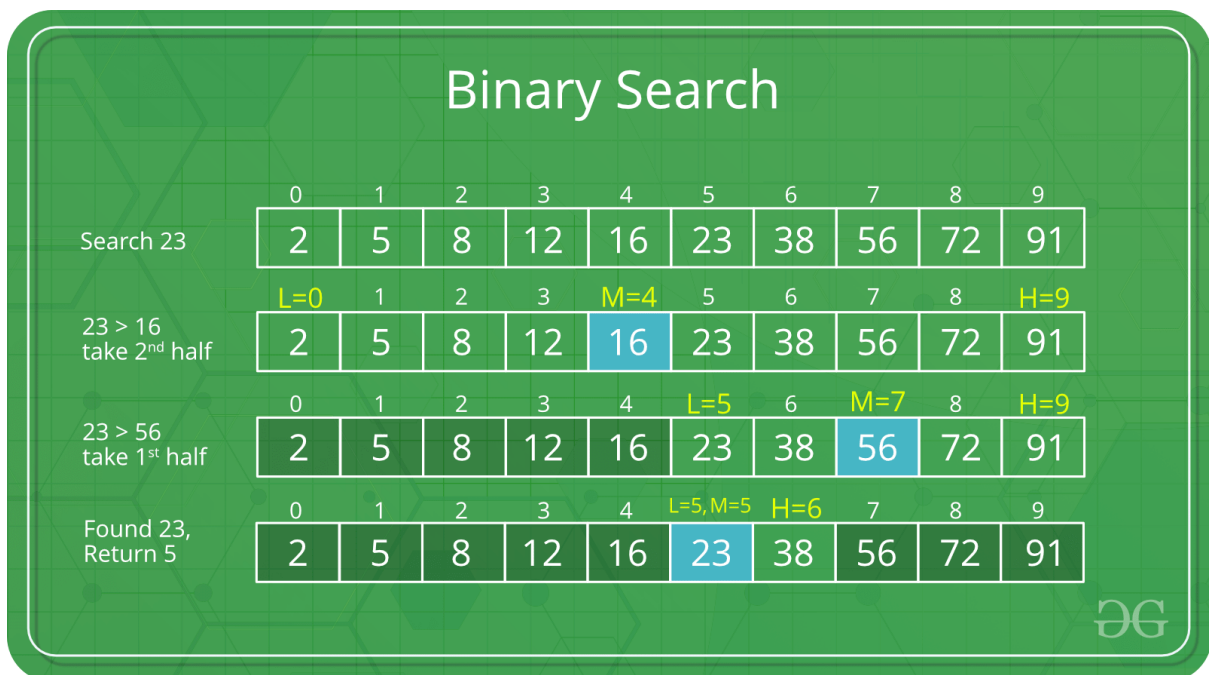
Both are matching. So we stop comparing and display element found at index 5.

Program:

```
#include <stdio.h>
void main()
{
int a[100],i,n,flag=0,ele,pos;
    printf("please enter array size");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("\n enter a[%d] value",i);
        scanf("%d",&a[i]);
    }
    printf("please enter any value to search in the given array list \n ");
    scanf("%d",&ele);
    for(i=0;i<n;i++)
    {
if(ele==a[i])
        {
            flag=1;
            pos=i;
        }
    }
    if(flag==1)
        printf("element is founded at %d index",pos);
    else
        printf("element is not founded ");
}
```

Binary search: Binary search is a very fast and efficient searching technique. It requires the list to be in sorted order. In this method, to search an element you can compare it with the present element at the center of the list. If it matches, then the search is successful otherwise the list is divided into two halves: one from the 0th element to the middle element which is the center element (first half) another from the center element to the last element (which is the 2nd half) where all values are greater than the center element.

The searching mechanism proceeds from either of the two halves depending upon whether the target element is greater or smaller than the central element. If the element is smaller than the central element, then searching is done in the first half, otherwise searching is done in the second half.



Program:

```
#include<stdio.h>

void main()
{
    int arr[50],i,n,x,flag=0,first,last,mid;

    printf("Enter size of array:");
    scanf("%d",&n);
    printf("\nEnter array element(ascending order)\n");

    for(i=0;i<n;++i)
        scanf("%d",&arr[i]);

    printf("\nEnter the element to search:");
    scanf("%d",&x);

    first=0;
```

```

last=n-1;

while(first<=last)
{
    mid=(first+last)/2;

    if(x==arr[mid]){
        flag=1;
        break;
    }
    else
        if(x>arr[mid])
            first=mid+1;
        else
            last=mid-1;
}

if(flag==1)
    printf("\nElement found at position %d",mid+1);
else
    printf("\nElement not found");
}

```

Algorithm:

BINARY_SEARCH(A, lower_bound, upper_bound, VAL)

- **Step 1:** [INITIALIZE] SET BEG = lower_bound
END = upper_bound, POS = - 1
- **Step 2:** Repeat Steps 3 and 4 while BEG <=END
- **Step 3:** SET MID = (BEG + END)/2
- **Step 4:** IF A[MID] = VAL
SET POS = MID
PRINT "VALUE IS PRESENT IN THE ARRAY"
Go to Step 6
ELSE IF A[MID] > VAL
SET END = MID - 1
ELSE
SET BEG = MID + 1
[END OF IF]
[END OF LOOP]
- **Step 5:** IF POS = -1
PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
[END OF IF]
- **Step 6:** EXIT

Searching

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

There are two popular search methods that are widely used in order to search some item into the list. However, choice of the algorithm depends upon the arrangement of the list.

- Linear Search
- Binary Search

Linear Search

Linear search is the simplest search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match found then location of the item is returned otherwise the algorithm return NULL.

Linear search is mostly used to search an unordered list in which the items are not sorted. The algorithm of linear search is given as follows.

Algorithm

- LINEAR_SEARCH(A, N, VAL)
- **Step 1:** [INITIALIZE] SET POS = -1
- **Step 2:** [INITIALIZE] SET I = 1
- **Step 3:** Repeat Step 4 while I<=N
- **Step 4:** IF A[I] = VAL
SET POS = I
PRINT POS
Go to Step 6
[END OF IF]
SET I = I + 1
[END OF LOOP]
- **Step 5:** IF POS = -1
PRINT " VALUE IS NOT PRESENTIN THE ARRAY "
[END OF IF]
- **Step 6:** EXIT

Complexity of algorithm

| Complexity | Best Case | Average Case | Worst Case |
|------------|-----------|--------------|------------|
| Time | O(1) | O(n) | O(n) |

list

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

search element **12**

Step 1:

search element (12) is compared with first element (65)

list

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

list

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

list

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

list

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

list

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

list

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 65 | 20 | 10 | 55 | 32 | 12 | 50 | 99 |

12

Both are matching. So we stop comparing and display element found at index 5.

Program:

```
#include <stdio.h>
void main()
{
int a[100],i,n,flag=0,ele,pos;
    printf("plese enter array size");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("\n enter a[%d] value",i);
        scanf("%d",&a[i]);
    }
    printf("please enter any value to search in the given array list \n ");
    scanf("%d",&ele);
    for(i=0;i<n;i++)
    {
if(ele==a[i])
        {
            flag=1;
            pos=i;
        }
    }
    if(flag==1)
        printf("element is founded at %d index",pos);
    else
        printf("element is not founded ");
}
```

Insertion sort is similar to arranging the documents of a bunch of students in order of their ascending roll number. Starting from the second element, we compare it with the first element and swap it if it is not in order. Similarly, we take the third element in the next iteration and place it at the right place in the subarray of the first and second elements (as the subarray containing the first and second elements is already sorted). We repeat this step with the fourth element of the array in the next iteration and place it at the right position in the subarray containing the first, second and the third elements. We repeat this process until our array gets sorted. So, the steps to be followed are:

1. Compare the current element in the iteration (say A) with the previous adjacent element to it. If it is in order then continue the iteration else, go to step 2.
2. Swap the two elements (the current element in the iteration (A) and the previous adjacent element to it).
3. Compare A with its new previous adjacent element. If they are not in order then proceed to step 4.
4. Swap if they are not in order and repeat steps 3 and 4.
5. Continue the iteration.

The diagram given below will make you understand this better.

Working of insertion sort

Initial array

| | | | | |
|----|----|----|----|----|
| 16 | 19 | 11 | 15 | 10 |
|----|----|----|----|----|

First iteration

| | | | | |
|----|----|----|----|----|
| 16 | 19 | 11 | 15 | 10 |
|----|----|----|----|----|

No swapping –

| | | | | |
|----|----|----|----|----|
| 16 | 19 | 11 | 15 | 10 |
|----|----|----|----|----|

Second iteration

| | | | | |
|----|----|----|----|----|
| 16 | 19 | 11 | 15 | 10 |
|----|----|----|----|----|

Swap

| | | | | |
|----|----|----|----|----|
| 16 | 11 | 19 | 15 | 10 |
|----|----|----|----|----|

Swap

| | | | | |
|----|----|----|----|----|
| 11 | 16 | 19 | 15 | 10 |
|----|----|----|----|----|

Third iteration

| | | | | |
|----|----|----|----|----|
| 11 | 16 | 19 | 15 | 10 |
|----|----|----|----|----|

Swap

| | | | | |
|----|----|----|----|----|
| 11 | 16 | 15 | 19 | 10 |
|----|----|----|----|----|

Swap

| | | | | |
|----|----|----|----|----|
| 11 | 15 | 16 | 19 | 10 |
|----|----|----|----|----|

No swapping –

| | | | | |
|----|----|----|----|----|
| 11 | 15 | 16 | 19 | 10 |
|----|----|----|----|----|

Fourth iteration

| | | | | |
|----|----|----|----|----|
| 11 | 15 | 16 | 19 | 10 |
|----|----|----|----|----|

Swap

| | | | | |
|----|----|----|----|----|
| 11 | 15 | 16 | 10 | 19 |
|----|----|----|----|----|

Swap

| | | | | |
|----|----|----|----|----|
| 11 | 15 | 10 | 16 | 19 |
|----|----|----|----|----|

Swap

| | | | | |
|----|----|----|----|----|
| 11 | 10 | 15 | 16 | 19 |
|----|----|----|----|----|

Swap

| | | | | |
|----|----|----|----|----|
| 10 | 11 | 15 | 16 | 19 |
|----|----|----|----|----|

//w a c p for insertion sort

```
#include <stdio.h>
```

```
void main()
```

```
{
```

```
    int a[100];
```

```
    int i,j,n;
```

```
clrscr();
```

```
printf("please enter n value");
```

```
scanf("%d",&n);
```

```
printf("enter array values");
```

```
for(i=0;i<n;i++)
```

```
{
```

```
scanf("%d",&a[i]);
```

```
}
```

```
printf("\n array values before sorting \n ");
```

```
    for(i=0;i<n;i++)
```

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

```
    for(i=0;i<n;i++)
```

```
{
    j = i;
    //i is not the first element
    while(j>0)
    {
        //not in order
        if(a[j-1] > a[j])
        {
            //swapping
            int temp = a[j-1];
            a[j-1] = a[j];
            a[j] = temp;
        }
        //in order
        else
        {
            break;
        }
        j--;
    }
}
printf("\n array values after sorting \n ");
for(i=0;i<n;i++)
    printf("%d\n",a[i]);
}
```

Bubble Sort Algorithm

Bubble Sort is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will swap both the elements, and then move on to compare the second and the third element, and so on.

If we have total n elements, then we need to repeat this process for $n-1$ times.

It is known as bubble sort, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

NOTE: If you are not familiar with Sorting in data structure, you should first learn [what is sorting](#) to know about the basics of sorting.

Implementing Bubble Sort Algorithm

Following are the steps involved in bubble sort(for sorting a given array in ascending order):

Starting with the first element(index = 0), compare the current element with the next element of the array.

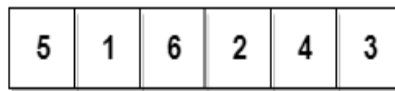
If the current element is greater than the next element of the array, swap them.

If the current element is less than the next element, move to the next element. Repeat Step 1.

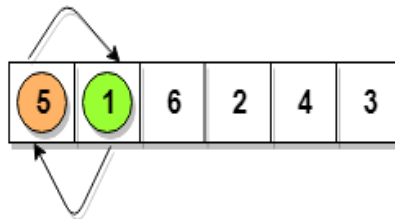
Let's consider an array with values {5, 1, 6, 2, 4, 3}

Below, we have a pictorial representation of how bubble sort will sort the given array.

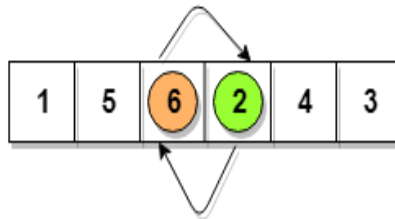
5>1
so interchange



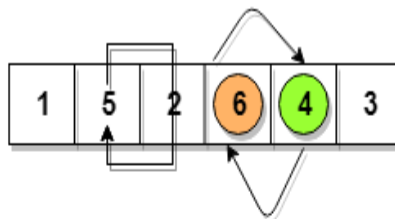
5<6
No swapping



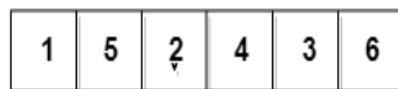
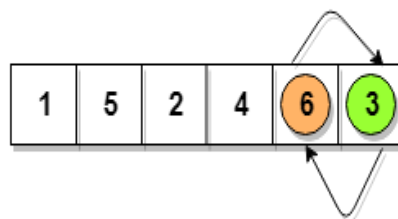
6>2
so interchange



6>4
so interchange



6>3
so interchange



This is first insertion

similarly, after all the iterations, the array gets sorted

So as we can see in the representation above, after the first iteration, 6 is placed at the last index, which is the correct position for it.

Similarly after the second iteration, 5 will be at the second last index, and so on.

ALGORITHM:

Step 1: Repeat Steps 2 and 3 for $i=1$ to $n-1$

Step 2: Set $j=1$

Step 3: Repeat while $j \leq n-i$

(A) if $a[j] > a[j+1]$

Then interchange $a[j]$ and $a[j+1]$

[End of if]

(B) Set $j = j+1$

[End of Inner Loop]

[End of Step 1 Outer Loop]

Step 4: Exit

program for bubble sort

```
#include <stdio.h>
void main()
{
    int arr[100], i,j, n, step, temp;
        // ask user for number of elements to be sorted
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    // input elements if the array
    for(i = 0; i < n; i++)
    {
        scanf("%d", &arr[i]);
    }
    for(i = 0; i <=(n-2); i++)
    {
        for(j = 0; j <n-1-i; j++)
        {
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
    // print the sorted array
    printf("Sorted Array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
    getch();
}
```

Quicksort algorithm is one of the fastest internal sorting algorithms

1. **Split or Partition:** Select a random element called pivot from the given sequence of elements to be sorted. Suppose the element is X, where X is any number. Now split the list into two small arrays or lists Y and Z such that, all elements in Y are smaller than X whereas all elements in Z are larger than X,
2. **Sort** the sub-arrays,
3. **Merge**(join or concatenate) the sorted sub-arrays.

The split divides the arrays into two smaller arrays. When these sub-arrays are ultimately sorted recursively using quicksort these sub-arrays are called **conquered**. Therefore, quicksort is based on **divide and conquer algorithm**.

The Quick Sort Algorithm

Suppose there are N elements as a[0], a[1], ..., a[N-1]. The steps for using the quick sort algorithm are given below,

#1: Select any element as a **pivot**. For example, we select the first element here. It'll help to split the array into two parts.

```
PIVOT = a[ FIRST ] //pivot selected, FIRST=0 here
```

#2: Initialize two pointers **i** and **j** as,

```
i = FIRST + 1 //First index of the array  
j = LAST //Last index of array
```

#3: Now we increase the value of **i** until we locate an element that is greater than the pivot element,

```
WHILE i<=j and a[i]<= PIVOT  
    i=i+1
```

#4: We decrease the value of **j** until we find a value less than the pivot element,

```
WHILE j<=j and a[j]>= PIVOT  
    j=j-1
```

#5: If $i < j$ interchange $a[i]$ and $a[j]$.

#6: Repeat Steps 2 to 4 until $i > j$,

#7: Interchange the selected pivot and $a[j]$,

#8: And finally recursively call this Quicksort for the two sub-arrays hence created at the two sides of the pivot element.

Understanding the Algorithm with an Example

Now, let us understand how the algorithm works using a simple example.

We have considered an array having values **50, 30, 10, 90, 80, 20, 40, 60** as shown in **figure 1**. At first, we select a pivot element which in our case is the first element, **50**.

figure 1

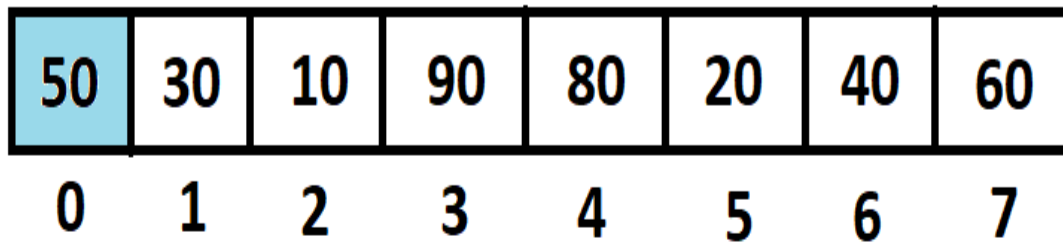


figure 2

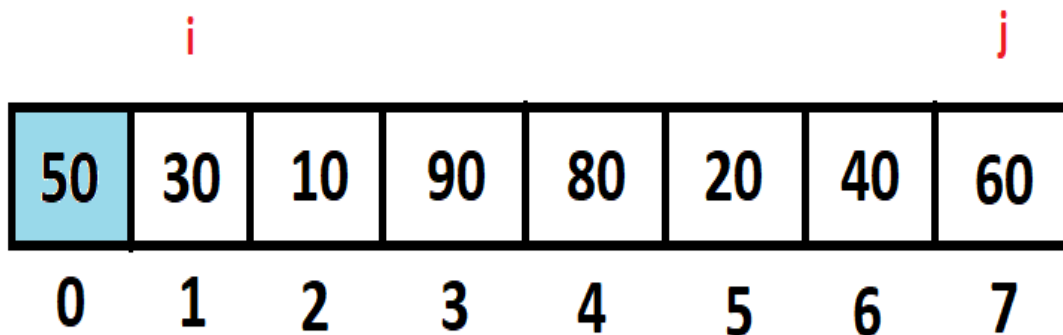
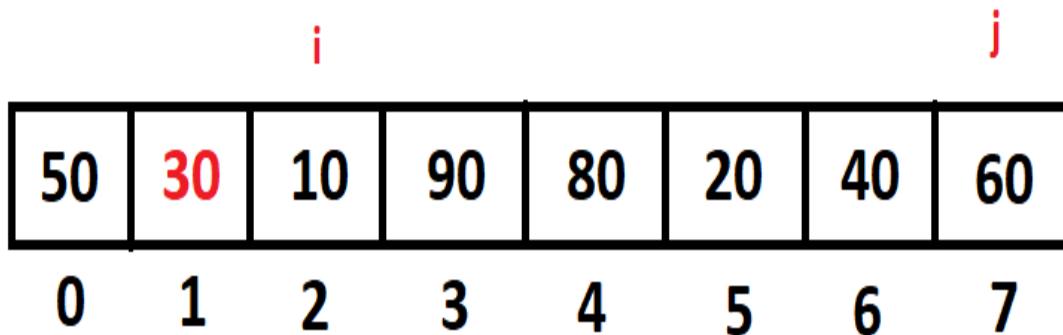


figure 3



Quicksort Algorithm

Then we initialize i and j pointers as shown in **figure 2** pointing on 30 and 60 respectively. Hence we move the i pointer towards the right until the element is greater than the pivot one. Which in our case we get at index 3, that is the value **90**.

figure 4

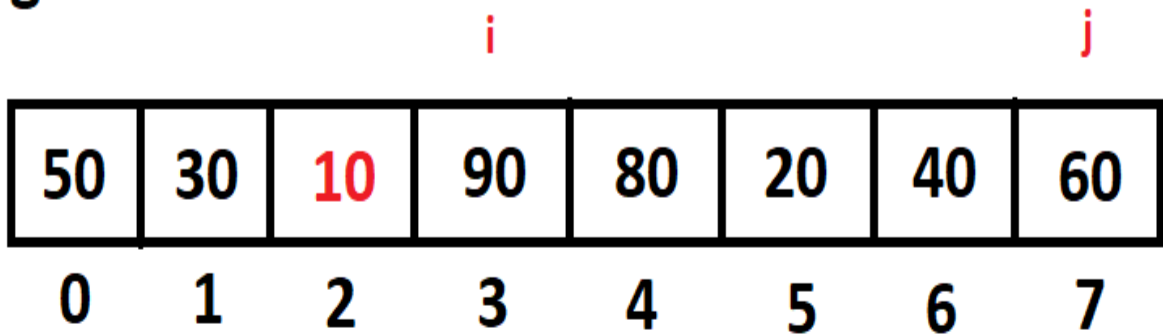


figure 5

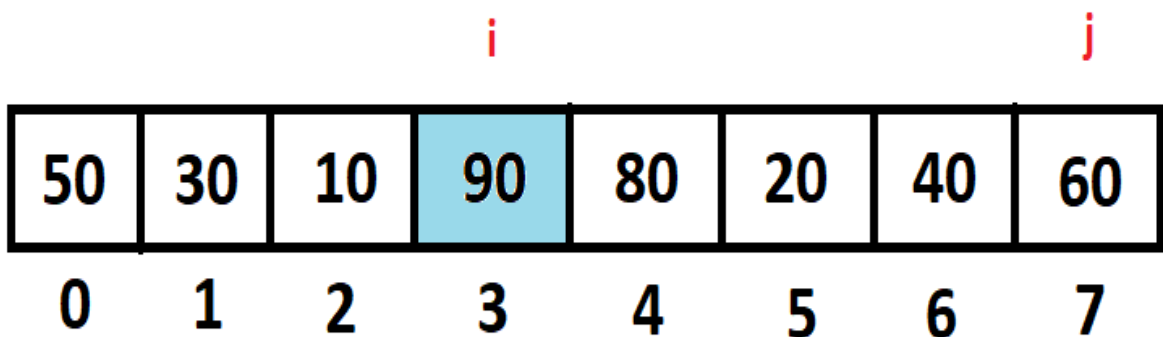
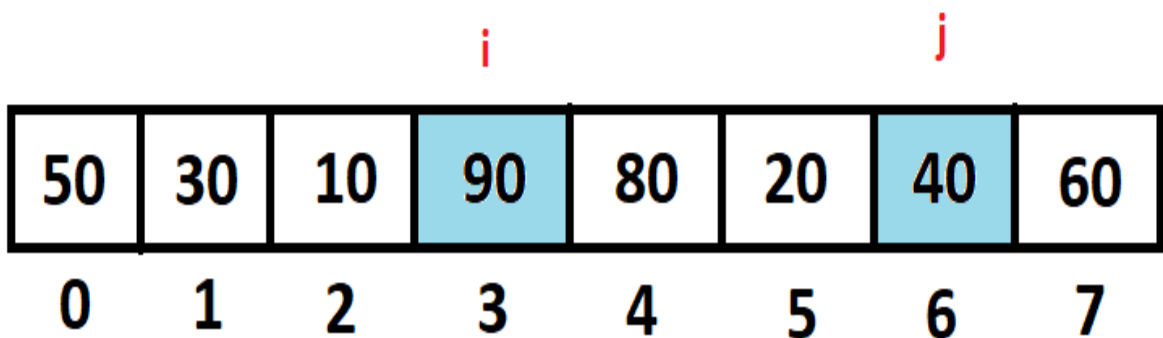


figure 6



Quicksort Algorithm

Similarly, j is moved towards the left until it finds a value smaller than the pivot, which we get at index 6 that is a value of 40. At this point in **figure 6**, we can see we have got both i and j values. Finally, we **swap** the corresponding values and get to a position shown in **figure 7**.

figure 7

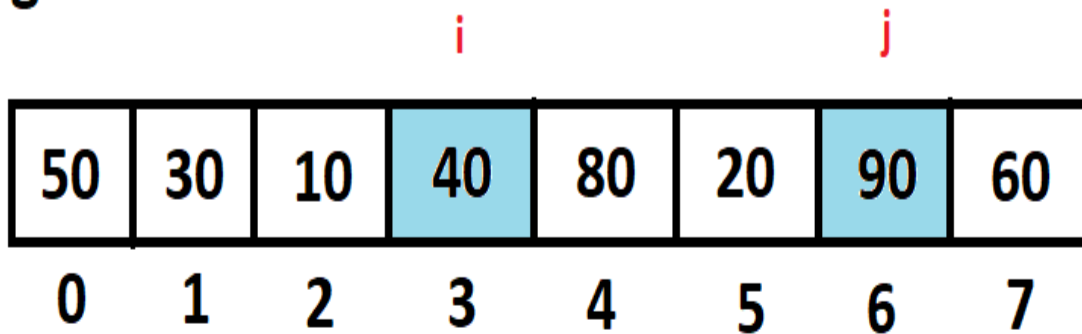


figure 8

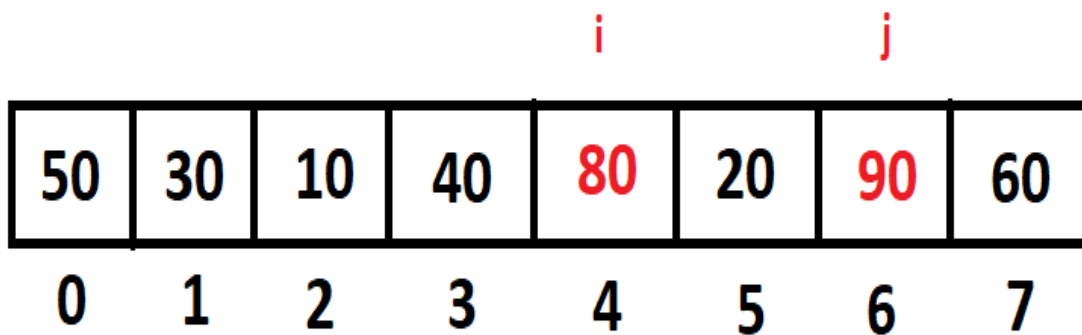
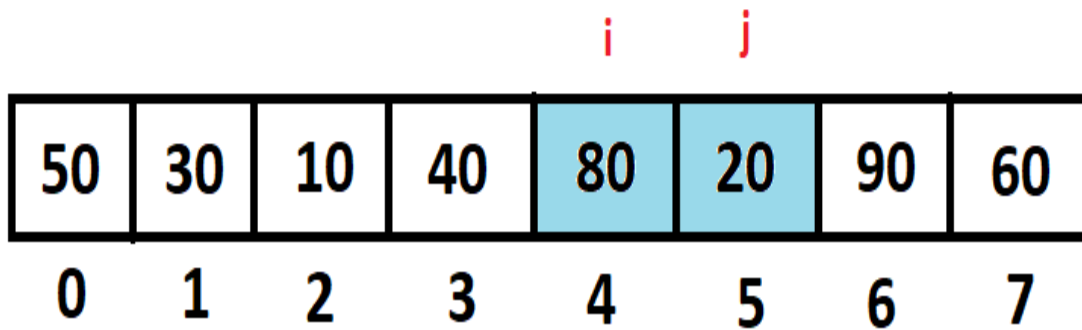


figure 9



Quicksort Algorithm

Then we again go through the i and j index searching and hence get to **figure 9**, where both our pointers, i and j stand at 4th and 5th index.

Similar to the previous case, we **swap** the corresponding values at i th and j th positions. So after this, we need to move the pointer i towards the right and j towards the left. Since $20 < 50$, so we need to increase i by 1 towards the right. As $80 > 50$, we stop moving i further. As shown in **figure 10** and **11**.

figure 10

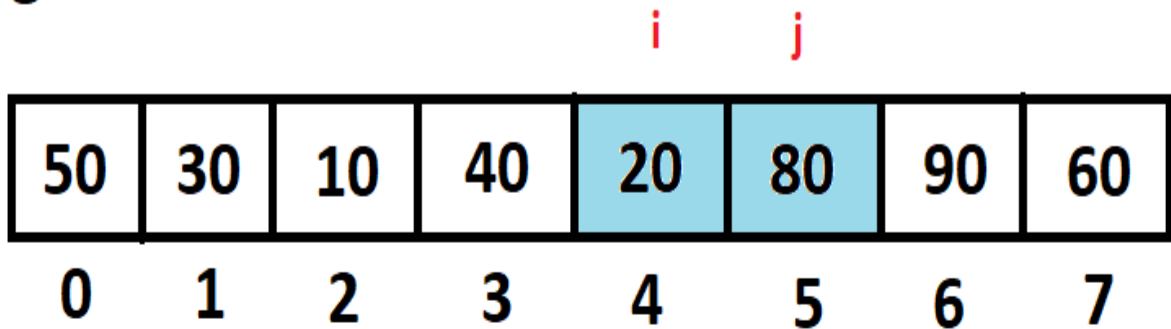


figure 11

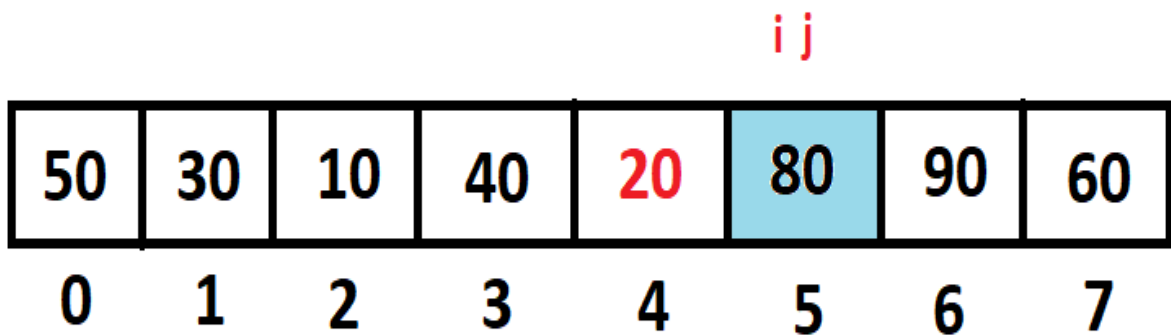
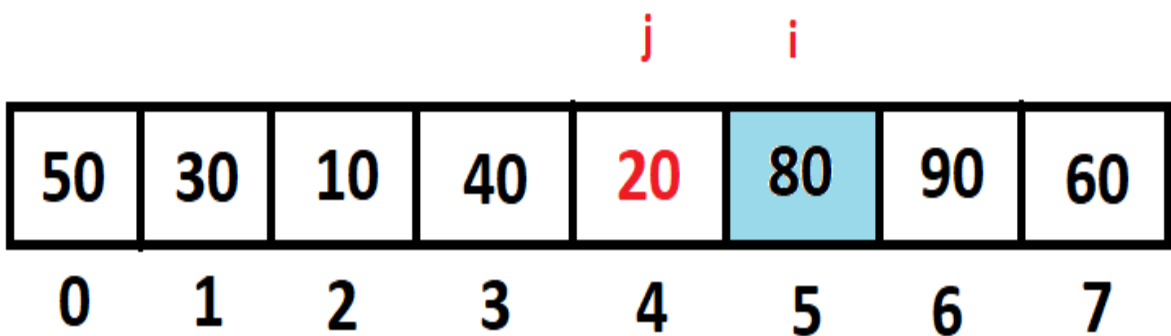


figure 12

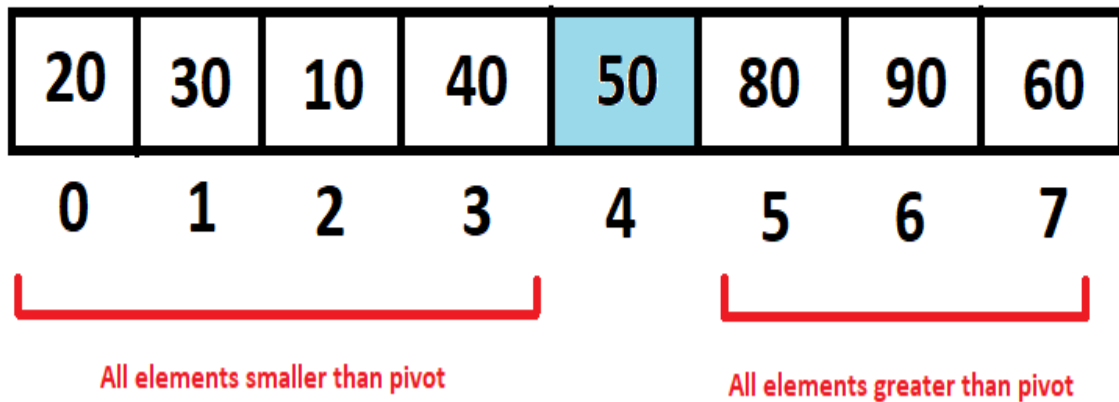


Quicksort Algorithm

Then we start moving j towards the left. As $20 < 50$, j is decreased by 1 and now stands at the 4th index as shown in figure 12. From the above condition, it is clear that i crosses j . Thus we stop at the point where $j < i$. Therefore the position of j is the **split** point.

Hence we swap the pivot and the value at index j points at giving us the array as in **figure 13**.

figure 13



Quicksort Algorithm

After this, we have to apply the same method for the sub-arrays to the left and right of the pivot element, 50. By this **divide and conquer** method, finally, we will get our sorted array.

Implementing the QuickSort Algorithm

1. QuickSort Algorithm in C

```
#include<stdio.h>
void quicksort(int a[25],int first,int last)
{
    int i, j, pivot, temp;

    if(first<last){
        pivot=first;
        i=first;
        j=last;

        while(i<j){
            while(a[i]<=a[pivot] && i<last)
                i++;
            while(a[j]>a[pivot])
                j--;
            if(i<j){
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
            }
        }

        temp=a[pivot];
        a[pivot]=a[j];
```

```
        a[j]=temp;
        quicksort(a,first,j-1);
        quicksort(a,j+1,last);
    }
}

int main()
{
    int i, n, a[25];

    printf("Enter total no.of elements: ");
    scanf("%d",&n);

    printf("Enter the elements: ");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    quicksort(a,0,n-1);

    printf("Sorted Array: ");
    for(i=0;i<n;i++)
        printf(" %d",a[i]);

    return 0;
}
```

Merge Sort

Merge Sort is a [Divide and Conquer](#) algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one. See the following C implementation for details.

ALGORITHM:

MergeSort(arr[], l, r)

If $r > l$

1. Find the middle point to divide the array into two halves:

$middle\ m = l + (r - l) / 2$

2. Call mergeSort for first half:

Call mergeSort(arr, l, m)

3. Call mergeSort for second half:

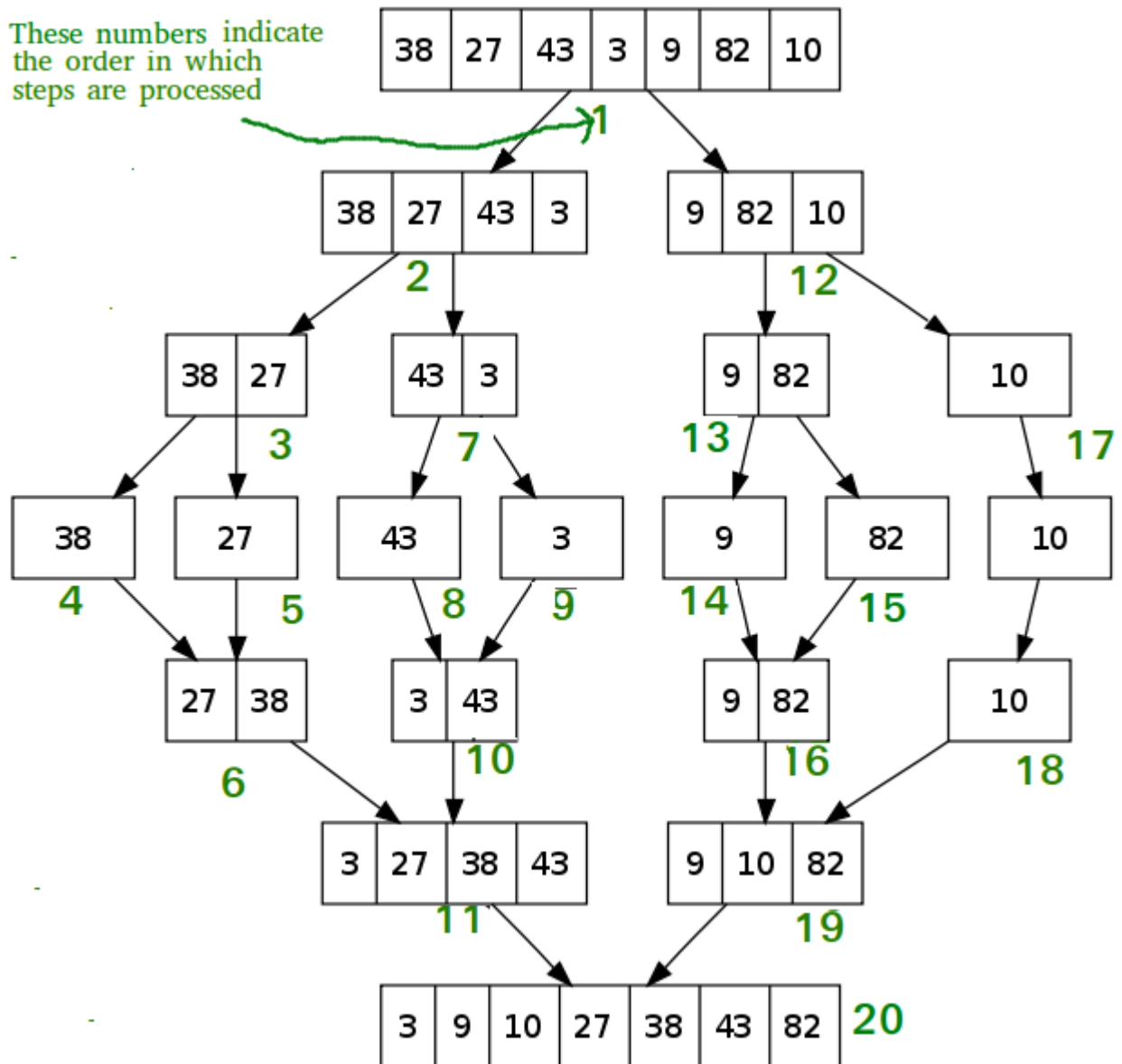
Call mergeSort(arr, m+1, r)

4. Merge the two halves sorted in step 2 and 3:

Call merge(arr, l, m, r)

The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.

These numbers indicate the order in which steps are processed



```
/* C program for Merge Sort */
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
// Merges two subarrays of arr[].
```

```
// First subarray is arr[l..m]
```

```
// Second subarray is arr[m+1..r]
```

```

void merge(int arr[], int l, int m, int r)
{
    int i, j, k;

    int n1 = m - l + 1;
    int n2 = r - m;

    /* create temp arrays */
    int L[100],R[100];

    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];

```



```
        j++;  
    }  
    k++;  
}
```

```
/* Copy the remaining elements of L[], if there  
are any */
```

```
while (i < n1) {  
    arr[k] = L[i];  
    i++;  
    k++;  
}
```

```
/* Copy the remaining elements of R[], if there  
are any */
```

```
while (j < n2) {  
    arr[k] = R[j];  
    j++;  
    k++;  
}  
}
```

```
/* l is for left index and r is right index of the  
sub-array of arr to be sorted */
```

```
void mergeSort(int arr[], int l, int r)  
{
```

```
if (l < r) {  
    // Same as (l+r)/2, but avoids overflow for  
    // large l and h  
    int m = l + (r - l) / 2;  
  
    // Sort first and second halves  
    mergeSort(arr, l, m);  
    mergeSort(arr, m + 1, r);  
  
    merge(arr, l, m, r);  
}  
}
```

```
/* UTILITY FUNCTIONS */  
/* Function to print an array */  
void printArray(int A[], int size)  
{  
    int i;  
    for (i = 0; i < size; i++)  
        printf("%d ", A[i]);  
    printf("\n");  
}
```

```
/* Driver code */  
void main()  
{
```

```
int arr[100],i,n;
    clrscr();
    printf("\n enter n value");
    scanf("%d",&n);
    printf("\n enter array values");
    for(i=0;i<n;i++)
    {
        scanf("%d",&arr[i]);
    }
    printf("Given array is \n");
    printArray(arr, n);

    mergeSort(arr, 0, n - 1);

    printf("\nSorted array is \n");
    printArray(arr, n);

}
```

Selection sort is [a sorting algorithm](#) that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

Working of Selection Sort

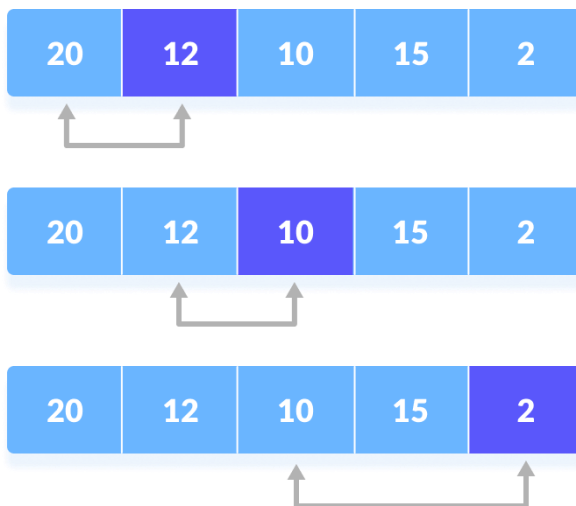


1. Set the first element as `minimum`.

Select first element as minimum

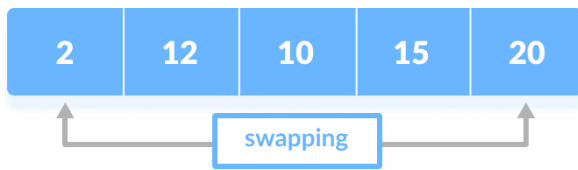
Compare `minimum` with the second element. If the second element is smaller than `minimum`, assign the second element as `minimum`.

Compare `minimum` with the third element. Again, if the third element is smaller, then assign `minimum` to the third element otherwise do nothing. The process goes on until the last element.



Compare minimum with the remaining elements

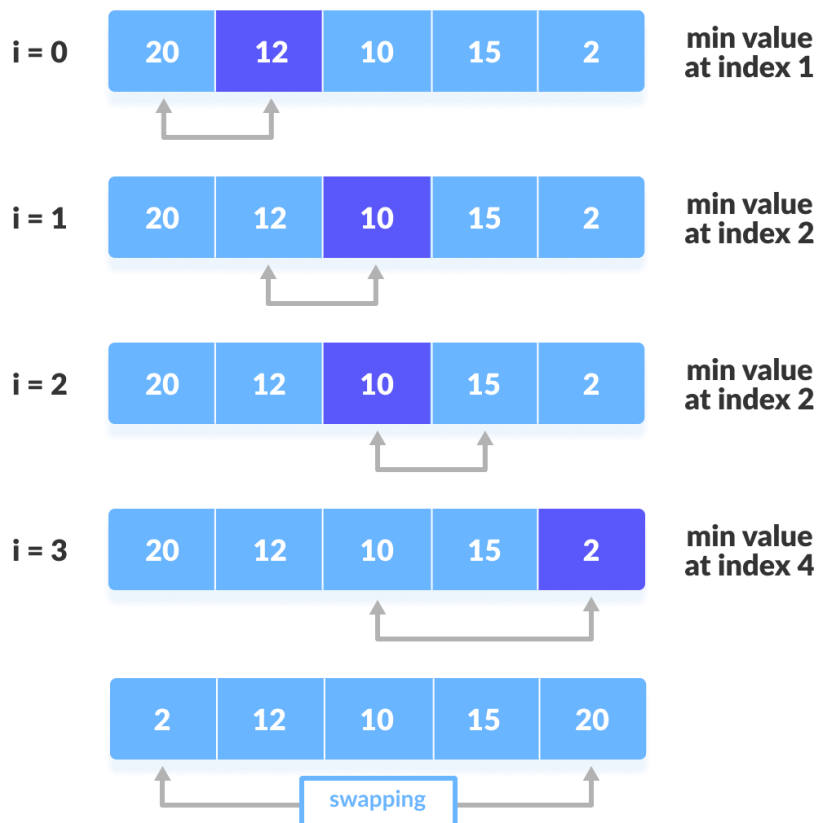
After each iteration, **minimum** is placed in the front of the unsorted list.



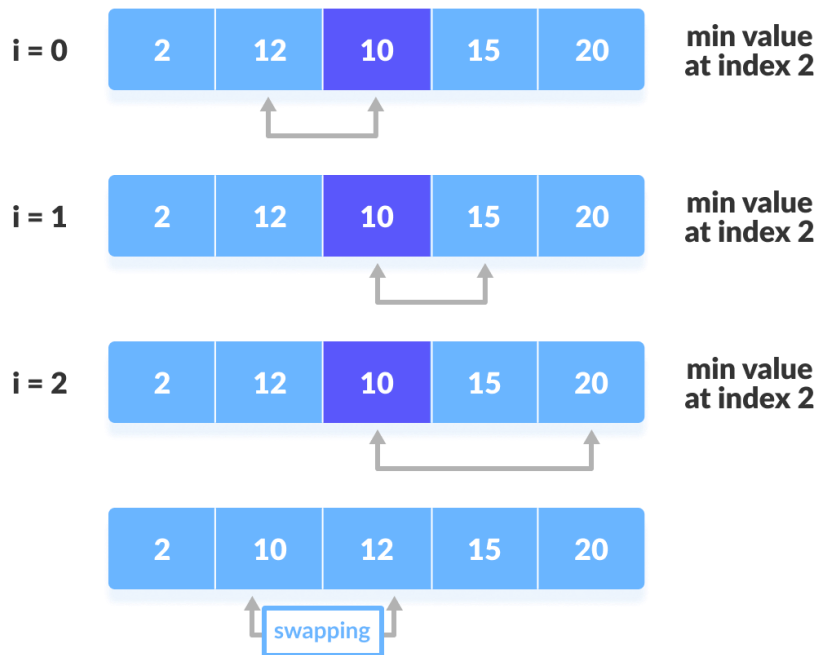
Swap the first with minimum

For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.

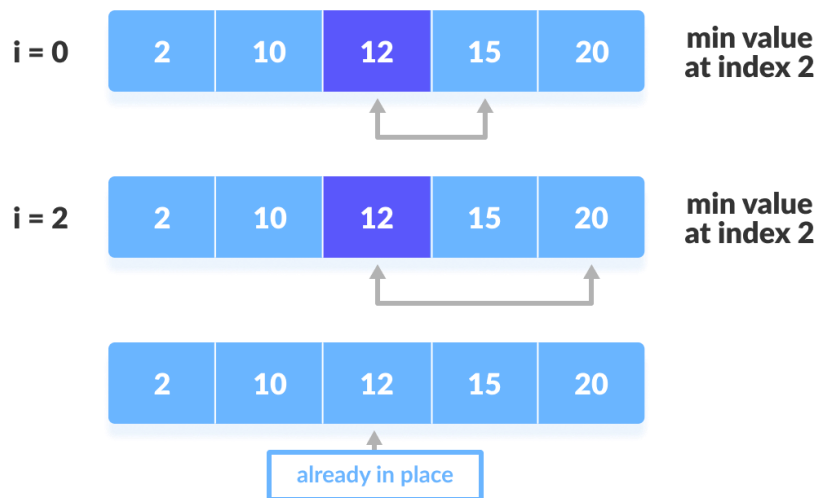
step = 0



step = 1



step = 2



step = 3



Selection Sort Algorithm

selectionSort(array, size)

repeat (size - 1) times

set the first unsorted element as the minimum

for each of the unsorted elements

if element < currentMinimum

set element as new minimum

swap minimum with first unsorted position

end selectionSort

Code for Selection Sort:

```
#include <stdio.h>
void main()
{
int a[100], n, i, j, minvalposition, temp;
printf("Enter number of elements");
scanf("%d", &n);
printf("Enter array Numbers");
for (i = 0; i < n; i++)
```

```

scanf("%d", &a[i]);

for(i = 0; i < n - 1; i++)
{
    minvalposition=i;
    for(j = i + 1; j < n; j++)
    {
        if(a[minvalposition] > a[j])
            minvalposition=j;
    }
    temp=a[i];
    a[i]=a[minvalposition];
    a[minvalposition]=temp;
}
printf("\n Sorted Array: \n");
for(i = 0; i < n; i++)
    printf("%d \n", a[i]);
}

```

Selection Sort Complexity

| | |
|-------------------------|----------|
| Time Complexity | |
| Best | $O(n^2)$ |
| Worst | $O(n^2)$ |
| Average | $O(n^2)$ |
| Space Complexity | $O(1)$ |
| Stability | No |

| Cycle | Number of Comparison |
|-------|----------------------|
| 1st | (n-1) |
| 2nd | (n-2) |
| 3rd | (n-3) |
| ... | ... |
| last | 1 |

Number of comparisons: $(n - 1) + (n - 2) + (n - 3) + \dots + 1 = n(n - 1)$

$/ 2$ nearly equals to n^2 .

Complexity = $O(n^2)$

Also, we can analyze the complexity by simply observing the number of loops. There are 2 loops so the complexity is $n*n = n^2$.

Time Complexities:

- **Worst Case Complexity:** $O(n^2)$

If we want to sort in ascending order and the array is in descending order then, the worst case occurs.

- **Best Case Complexity:** $O(n^2)$

It occurs when the array is already sorted

- **Average Case Complexity:** $O(n^2)$

It occurs when the elements of the array are in jumbled order (neither ascending nor descending).

The time complexity of the selection sort is the same in all cases. At every step, you have to find the minimum element and put it in the right place.

The minimum element is not known until the end of the array is not reached.

Space Complexity:

Space complexity is $O(1)$ because an extra variable `temp` is used.

Selection Sort Applications

The selection sort is used when

- a small list is to be sorted
- cost of swapping does not matter
- checking of all the elements is compulsory
- cost of writing to a memory matters like in flash memory (number of writes/swaps is $O(n)$ as compared to $O(n^2)$ of bubble sort)