Data structure in c++

List in C++

Lists are sequence containers that allow non-contiguous memory allocation. As compared to vector, the list has slow traversal, but once a position has been found, insertion and deletion are quick. Normally, when we say a List, we talk about a doubly linked list. For implementing a singly

```
#include <iostream>

#include <iterator>

#include <list>

using namespace std;


// function for printing the elements in a list

void showlist(list<int> g)

{

    list<int>::iterator it;

    for (it = g.begin(); it != g.end(); ++it)

        cout << '\t' << *it;

    cout << '\n';

}


// Driver Code

int main()
```

Data structure in c++

```
{


    list<int> gqlist1, gqlist2;


    for (int i = 0; i < 10; ++i) {
        gqlist1.push_back(i * 2);
        gqlist2.push_front(i * 3);
    }
    cout << "\nList 1 (gqlist1) is : ";
    showlist(gqlist1);


    cout << "\nList 2 (gqlist2) is : ";
    showlist(gqlist2);


    cout << "\ngqlist1.front() : " << gqlist1.front();
    cout << "\ngqlist1.back() : " << gqlist1.back();


    cout << "\ngqlist1.pop_front() : ";
    gqlist1.pop_front();
    showlist(gqlist1);


    cout << "\ngqlist2.pop_back() : ";
    gqlist2.pop_back();
```

```
    showlist(gqlist2);

    cout << "\ngqlist1.reverse() : ";
    gqlist1.reverse();
    showlist(gqlist1);

    cout << "\ngqlist2.sort(): ";
    gqlist2.sort();
    showlist(gqlist2);

    return 0;
}
```

Output

```
List 1 (gqlist1) is :     0    2    4    6    8    10    12
14    16    18


List 2 (gqlist2) is :     27    24    21    18    15    12    9
6    3    0


gqlist1.front() : 0

gqlist1.back() : 18
```

```
gqlist1.pop_front() :     2    4    6    8    10    12    14
16    18


gqlist2.pop_back() :     27    24    21    18    15    12    9
6    3


gqlist1.reverse() :     18    16    14    12    10    8    6
4    2


gqlist2.sort():     3    6    9    12    15    18    21    24
27
```

Initialize Vector in C++

A vector can store multiple data values like arrays, but they can only store object references and not primitive data types. They store an object's reference means that they point to the objects that contain the data, instead of storing them. Unlike an array, vectors need not be initialized with size. They have the flexibility to adjust according to the number of object references, which is possible because their storage is handled automatically by the container. The container will keep an internal copy of alloc, which is used to allocate storage for lifetime. Vectors can be located and traversed using iterators, so they are placed in contiguous storage. Vector has safety features also, which saves programs from crashing, unlike Array. We can give reserve space to vector, but not to arrays. An array is not a class, but a vector is a class. In vector, elements can be deleted, but not in arrays.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
  vector<int> vec;
  vec.push_back(1);
  vec.push_back(2);
  vec.push_back(3);
  vec.push_back(4);
  vec.push_back(5);
  vec.push_back(6);
  vec.push_back(7);
  vec.push_back(8);
```

```
vec.push_back(9);
vec.push_back(101);
for (int i = 0; i < vec.size(); i++)
{
  cout << vec[i] << " ";
}
return 0;
}
```

```
#include <iostream>
#include <vector>
using namespace std;

int main() {

  // initializer list
  vector<int> vector1 = {1, 2, 3, 4, 5};

  // uniform initialization
  vector<int> vector2{6, 7, 8, 9, 10};

  // method 3
  vector<int> vector3(5, 12);

  cout << "vector1 = ";
```

Data structure in c++

```
  // ranged loop
  for (const int i : vector1) {
    cout << i << "  ";
  }

  cout << "\nvector2 = ";

  // ranged loop
  for (const int i : vector2) {
    cout << i << "  ";
  }

  cout << "\nvector3 = ";

  // ranged loop
  for (int i : vector3) {
    cout << i << "  ";
  }

  return 0;
}
```

Output

```
vector1 = 1  2  3  4  5
vector2 = 6  7  8  9  10
vector3 = 12  12  12  12  12
```

Data structure in c++

What is a Struct in C++?

A **STRUCT** is a C++ data structure that can be used to store together elements of different data types. In C++, a structure is a user-defined data type. The structure creates a data type for grouping items of different data types under a single data type.

For example:

Suppose you need to store information about someone, their name, citizenship, and age. You can create variables like name, citizenship, and age to store the data separately.

However, you may need to store information about many persons in the future. It means variables for different individuals will be created. For example, name1, citizenship1, age1 etc. To avoid this, it’s better to create a struct

Difference between class and structure :

Data structure in c++

Class	Structure
Members of a class are private by default.	Members of a structure are public by default.
Member classes/structures of a class are private by default.	Member classes/structures of a structure are public by default.
It is declared using the **class** keyword.	It is declared using the **struct** keyword.
It is normally used for data abstraction and further inheritance.	It is normally used for the grouping of data

```
#include <iostream>
using namespace std;
struct Person
{
        int citizenship;
        int age;
};
int main(void) {
```

```
        struct Person p;

        p.citizenship = 1;

        p.age = 27;

        cout << "Person citizenship: " << p.citizenship << endl;

        cout << "Person age: " << p.age << endl;


        return 0;

}
```

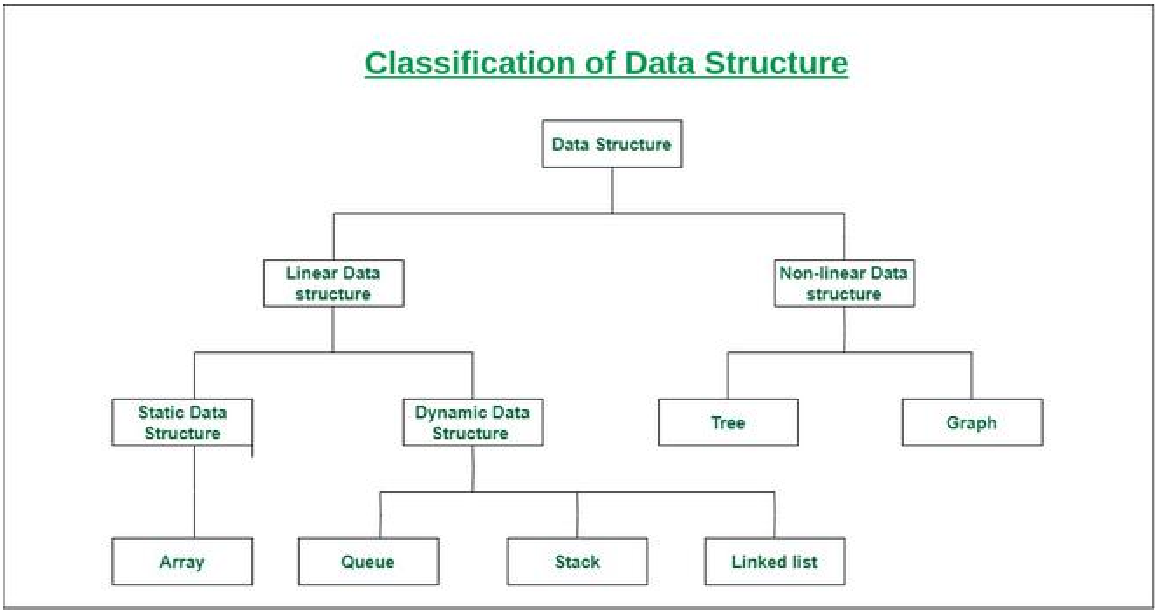
What is Data Structure:

A data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge about data structures.

Classification of Data Structure:

Data structure in c++



Classification of Data Structure

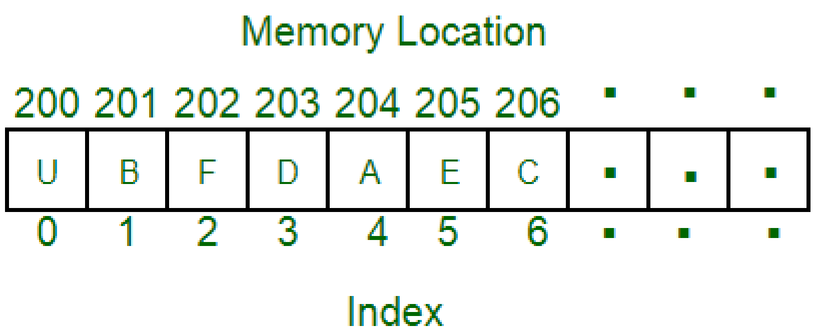
- **Linear data structure:** Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.
  Examples of linear data structures are array, stack, queue, linked list, etc.
    - **Static data structure:** Static data structure has a fixed memory size. It is easier to access the elements in a static data structure.
      An example of this data structure is an array.
    - **Dynamic data structure:** In dynamic data structure, the size is not fixed. It can be randomly updated during the runtime which may be considered efficient concerning the memory (space) complexity of the code.
      Examples of this data structure are queue, stack, etc.

Data structure in c++

- **Non-linear data structure:** Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only.

  Examples of non-linear data structures are trees and graphs.

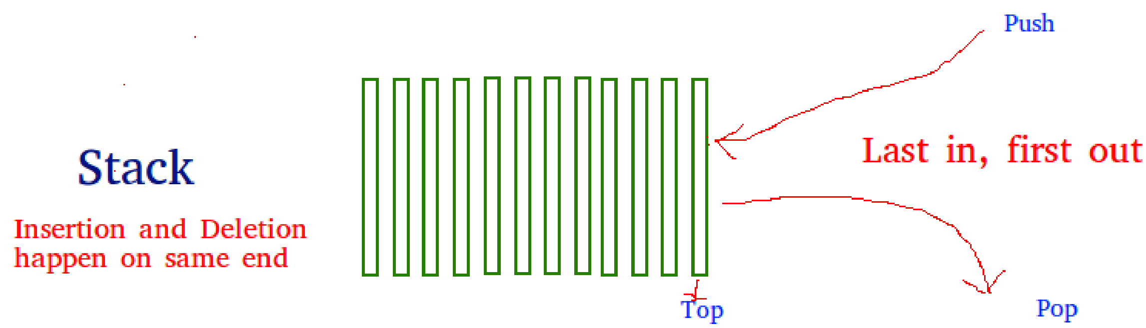
For example, we can store a list of items having the same data-type using the *array* data structure.



Stack

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

Data structure in c++



There are many real-life examples of a stack. Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO(Last In First Out)/FILO(First In Last Out) order.

A stack is an abstract data structure that contains a collection of elements. Stack implements the LIFO mechanism i.e. the element that is pushed at the end is popped out first. Some of the principle operations in the stack are –

- Push - This adds a data value to the top of the stack.
- Pop - This removes the data value on top of the stack
- Peek - This returns the top data value of the stack

A program that implements a stack using array is given as follows.

```
#include <iostream>
```

Data structure in c++

```
using namespace std;

int stack[100], n=100, top=-1;

void push(int val) {

  if(top>=n-1)

  cout<<"Stack Overflow"<<endl;

  else {

    top++;

    stack[top]=val;

  }

}

void pop() {

  if(top<=-1)

  cout<<"Stack Underflow"<<endl;

  else {

    cout<<"The popped element is "<< stack[top] <<endl;

    top--;

  }

}

void display() {

  if(top>=0) {
```

```
    cout<<"Stack elements are:";

    for(int i=top; i>=0; i--)

    cout<<stack[i]<<" ";

    cout<<endl;

  } else

  cout<<"Stack is empty";

}

int main() {

  int ch, val;

  cout<<"1) Push in stack"<<endl;

  cout<<"2) Pop from stack"<<endl;

  cout<<"3) Display stack"<<endl;

  cout<<"4) Exit"<<endl;

  do {

    cout<<"Enter choice: "<<endl;

    cin>>ch;

    switch(ch) {

      case 1: {

        cout<<"Enter value to be pushed:"<<endl;

        cin>>val;
```

```
      push(val);

      break;

    }

    case 2: {

      pop();

      break;

    }

    case 3: {

      display();

      break;

    }

    case 4: {

      cout<<"Exit"<<endl;

      break;

    }

    default: {

      cout<<"Invalid Choice"<<endl;

    }

  }

}while(ch!=4);
```

```
  return 0;

}
```

Output

```
1) Push in stack

2) Pop from stack

3) Display stack

4) Exit



Enter choice: 1

Enter value to be pushed: 2

Enter choice: 1

Enter value to be pushed: 6

Enter choice: 1

Enter value to be pushed: 8

Enter choice: 1

Enter value to be pushed: 7

Enter choice: 2

The popped element is 7
```

```
Enter choice: 3

Stack elements are:8 6 2

Enter choice: 5

Invalid Choice

Enter choice: 4

Exit
```

By creating class named stack

```
class Stack

{

   static int MAX = 1000; // Maximum Stack size

   int top;

   int myStack[] = new int[MAX];


   boolean isEmpty()

   {

      return (top < 0);

   }

   Stack() {

      top = -1;

   }
```

Data structure in c++

```
    boolean push(int item)
    {
       if (top >= (MAX-1))
       {
          Cout<<"Stack Overflow";
          return false;
       }
    else
    {
    myStack[++top] = item;
    cout<<item;
    return true;
    }
 }

 int pop()
    {
       if (top < 0)
       {
          Cout<<"Stack Underflow";
          return 0;
       }
 else
       {
          int item = myStack[top--];
```

```
            return item;
         }
      }
   }
   //Main class code
   void Main ()
   {
         Stack st();
         Cout<<"Stack Push:";
         st.push(1);
         st.push(3);
         sta.push(5);
         cout<<"Stack Pop:";
         while(!st.isEmpty())
         {
            Cout<<st.pop();
         }
      }
   }
```

Output:

Stack Push:

1

3

5

Data structure in c++

Stack Pop:

5

3

1

Another code :

```
#include <iostream>
#include <cstdlib>
using namespace std;

// Define the default capacity of the stack
#define SIZE 10

// A class to represent a stack
class Stack
{
```

```
    int *arr;

    int top;

    int capacity;


public:

    Stack(int size = SIZE);        // constructor

    ~Stack();                      // destructor


    void push(int);

    int pop();

    int peek();


    int size();

    bool isEmpty();

    bool isFull();

};


// Constructor to initialize the stack

Stack::Stack(int size)

{
```

Data structure in c++

```
    arr = new int[size];
    capacity = size;
    top = -1;
}

// Destructor to free memory allocated to the stack
Stack::~Stack() {
    delete[] arr;
}

// Utility function to add an element `x` to the stack
void Stack::push(int x)
{
    if (isFull())
    {
        cout << "Overflow\nProgram Terminated\n";
        exit(EXIT_FAILURE);
    }

    cout << "Inserting " << x << endl;
```

```
    arr[++top] = x;
}

// Utility function to pop a top element from the stack
int Stack::pop()
{
    // check for stack underflow
    if (isEmpty())
    {
        cout << "Underflow\nProgram Terminated\n";
        exit(EXIT_FAILURE);
    }

    cout << "Removing " << peek() << endl;

    // decrease stack size by 1 and (optionally) return the popped element
    return arr[top--];
}

// Utility function to return the top element of the stack
```

Data structure in c++

```
int Stack::peek()

{

   if (!isEmpty()) {

      return arr[top];

   }

   else {

      exit(EXIT_FAILURE);

   }

}


// Utility function to return the size of the stack

int Stack::size() {

   return top + 1;

}


// Utility function to check if the stack is empty or not

bool Stack::isEmpty() {

   return top == -1;             // or return size() == 0;

}
```

Data structure in c++

```
// Utility function to check if the stack is full or not

bool Stack::isFull() {

   return top == capacity - 1;    // or return size() == capacity;

}


int main()

{

   Stack pt(3);


   pt.push(1);

   pt.push(2);


   pt.pop();

   pt.pop();


   pt.push(3);


   cout << "The top element is " << pt.peek() << endl;

   cout << "The stack size is " << pt.size() << endl;
```

```
    pt.pop();


    if (pt.isEmpty()) {
        cout << "The stack is empty\n";
    }
    else {
        cout << "The stack is not empty\n";
    }


    return 0;
}
```

Output:

```
Inserting 1
Inserting 2
Removing 2
Removing 1
Inserting 3
The top element is 3
The stack size is 1
Removing 3
The stack is empty
```

Data structure in c++

Or using stack class (already exist) by calling stack library :

```
#include <iostream>

#include <stack>

using namespace std;


// Stack implementation in C++ using `std::stack`

int main()

{

   stack<string> s;
```

Data structure in c++

```
s.push("A");    // Insert `A` into the stack

s.push("B");    // Insert `B` into the stack

s.push("C");    // Insert `C` into the stack

s.push("D");    // Insert `D` into the stack


// returns the total number of elements present in the stack

cout << "The stack size is " << s.size() << endl;


// prints the top of the stack (`D`)

cout << "The top element is " << s.top() << endl;


s.pop();       // removing the top element (`D`)

s.pop();       // removing the next top (`C`)


cout << "The stack size is " << s.size() << endl;
```

```
    // check if the stack is empty
    if (s.empty()) {
        cout << "The stack is empty\n";
    }
    else {
        cout << "The stack is not empty\n";
    }

    return 0;
}
```

Output:

```
The top element is D
The stack size is 2
The stack is not empty
```

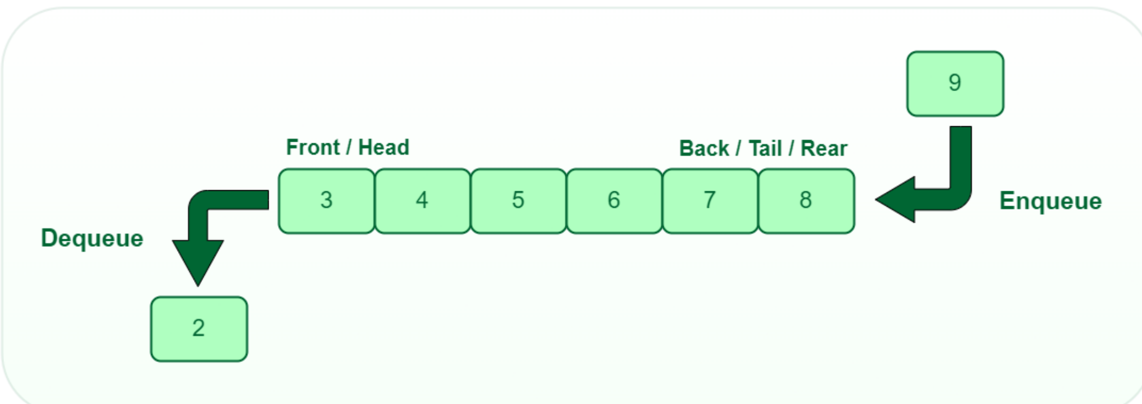
Data structure in c++

Queue

What is Queue?

A queue is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.

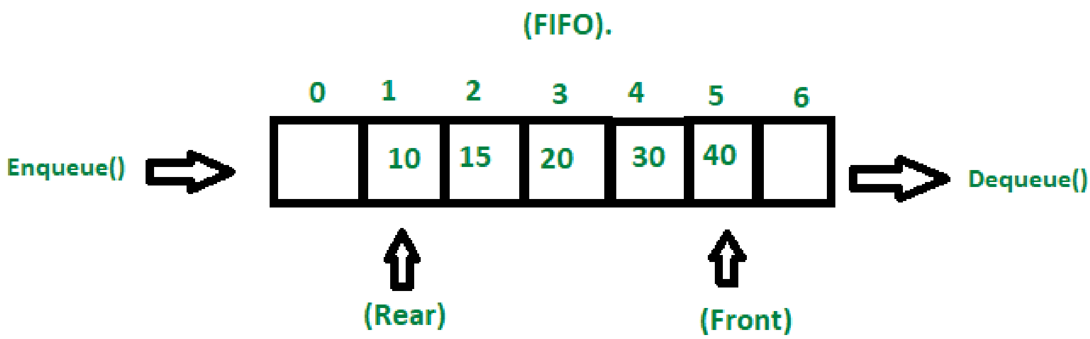
We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. The element which is first pushed into the order, the operation is first performed on that.



FIFO Principle of Queue:

- A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e. First come first serve).
- Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the **front** of the queue(sometimes, **head** of the queue), similarly, the position of the last entry in the queue, that is, the one most recently added, is called the **rear** (or the **tail**) of the queue. See the below figure.

Data structure in c++



A queue is an abstract data structure that contains a collection of elements. Queue implements the FIFO mechanism i.e. the element that is inserted first is also deleted first. In other words, the least recently added element is removed first in a queue.

```
#include <iostream>

using namespace std;

int queue[100], n = 100, front = - 1, rear = - 1;

void Insert() {

  int val;

  if (rear == n - 1)

  cout<<"Queue Overflow"<<endl;

  else {

    if (front == - 1)

    front = 0;
```

```
    cout<<"Insert the element in queue : "<<endl;

    cin>>val;

    rear++;

    queue[rear] = val;

  }

}

void Delete() {

  if (front == - 1 || front > rear) {

    cout<<"Queue Underflow ";

    return ;

  } else {

    cout<<"Element deleted from queue is : "<< queue[front] <<endl;

    front++;;

  }

}

void Display() {

  if (front == - 1)

  cout<<"Queue is empty"<<endl;

  else {

    cout<<"Queue elements are : ";
```

```
    for (int i = front; i <= rear; i++)

    cout<<queue[i]<<" ";

      cout<<endl;

  }

}

int main() {

  int ch;

  cout<<"1) Insert element to queue"<<endl;

  cout<<"2) Delete element from queue"<<endl;

  cout<<"3) Display all the elements of queue"<<endl;

  cout<<"4) Exit"<<endl;

  do {

    cout<<"Enter your choice : "<<endl;

    cin>>ch;

    switch (ch) {

      case 1: Insert();

      break;

      case 2: Delete();

      break;

      case 3: Display();
```

```
      break;

      case 4: cout<<"Exit"<<endl;

      break;

      default: cout<<"Invalid choice"<<endl;

    }

  } while(ch!=4);

  return 0;

}
```

The output of the above program is as follows

```
1) Insert element to queue

2) Delete element from queue

3) Display all the elements of queue

4) Exit

Enter your choice : 1

Insert the element in queue : 4

Enter your choice : 1

Insert the element in queue : 3

Enter your choice : 1

Insert the element in queue : 5
```

```
Enter your choice : 2

Element deleted from queue is : 4

Enter your choice : 3

Queue elements are : 3 5

Enter your choice : 7

Invalid choice

Enter your choice : 4

Exit
```

Basic Operations

The queue data structure includes the following operations:

- **EnQueue:** Adds an item to the queue. Addition of an item to the queue is always done at the rear of the queue.
- **DeQueue:** Removes an item from the queue. An item is removed or de-queued always from the front of the queue.
- **isEmpty:** Checks if the queue is empty.
- **isFull:** Checks if the queue is full.
- **peek:** Gets an element at the front of the queue without removing it.

Data structure in c++

```
using namespace std;


class Queue {

private:

int myqueue[MAX_SIZE], front, rear;


public:

Queue(){

front = -1;

rear = -1;

    }


boolisFull(){

if(front == 0 && rear == MAX_SIZE - 1){

return true;

        }

return false;

    }


boolisEmpty(){

if(front == -1) return true;

else return false;

    }


void enQueue(int value){

if(isFull()){
```

Data structure in c++

```
cout << endl<< "Queue is full!!";

        } else {

if(front == -1) front = 0;

rear++;

myqueue[rear] = value;

cout << value << " ";

        }

    }

int deQueue(){

int value;

if(isEmpty()){

cout << "Queue is empty!!" << endl; return(-1); } else { value = myqueue[front]; if(front >= re

front = -1;

rear = -1;

            }

else {

front++;

            }

cout << endl << "Deleted => " << value << " from myqueue";

return(value);

        }

    }


    /* Function to display elements of Queue */

void displayQueue()

    {

int i;
```

```
if(isEmpty()) {

cout << endl << "Queue is Empty!!" << endl;

        }

else {

cout << endl << "Front = " << front;

cout << endl << "Queue elements : ";

for(i=front; i<=rear; i++)

cout << myqueue[i] << "\t";

cout << endl << "Rear = " << rear << endl;

        }

    }

};

int main()

{

    Queue myq;


myq.deQueue();      //deQueue


cout<<"Queue created:"<<endl; myq.enQueue(10); myq.enQueue(20); myq.enQueue(30); myq.enQueue(40); myq.enQueue(50);
myq.enQueue(60);


myq.displayQueue();


    //deQueue =>removes 10

myq.deQueue();
```

```
    //queue after dequeue
myq.displayQueue();


return 0;
}
```

Output:

Queue is empty!!

Queue created:

10 20 30 40 50

Queue is full!!

Front = 0

Queue elements : 10 20 30 40 50

Rear = 4

Deleted => 10 from myqueue

Front = 1

Queue elements: 20 30 40 50

Rear = 4

By creating class named queue another code :

Data structure in c++

```
// A class representing a queue

class Queue

{

int front, rear, size;

int  max_size;


public Queue(int max_size) {

this.max_size = max_size;

front = this.size = 0;

rear = max_size - 1;

int myqueue[this.max_size];


    }


    //if size = max_size , queue is full

boolean isFull(Queue queue)

{  return (queue.size == queue.max_size);

    }


    // size = 0, queue is empty

boolean isEmpty(Queue queue)

{  return (queue.size == 0); }


    // enqueue - add an element to the queue

void enqueue( int item)

    {
```

Data structure in c++

```
if (isFull(this))

return;

this.rear = (this.rear + 1)%this.max_size;

this.myqueue[this.rear] = item;

this.size = this.size + 1;

cout<<(item + " " );

    }

    // dequeue - remove an elment from the queue

int dequeue()

    {

if (isEmpty(this))

return Integer.MIN_VALUE;


int item = this.myqueue[this.front];

this.front = (this.front + 1)%this.max_size;

this.size = this.size - 1;

        return item;

    }


    // move to front of the queue

int front()

    {

if (isEmpty(this))

return Integer.MIN_VALUE;


return this.myqueue[this.front];
```

```
    }


    // move to the rear of the queue
int rear()
    {
if (isEmpty(this))
return Integer.MIN_VALUE;


return this.myqueue[this.rear];
    }
}


// main class
void Main
{


        Queue q (1000);


cout <<"Queue created as:";
q.enqueue(10);
q.enqueue(20);
q.enqueue(30);
q.enqueue(40);


cout <<("\nElement " + queue.dequeue() +    " dequeud from queue\n");
cout <<("Front item is " +    queue.front());
```

Data structure in c++

```
cout <<("Rear item is " +     queue.rear());

    }

}
```

Output:

Queue created as:

10   20   30   40

Element 10 dequeued from queue

Front item is 20

Rear item is 40

Another code :

```
/ CPP program for array

// implementation of queue

#include <bits/stdc++.h>

using namespace std;


// A structure to represent a queue

class Queue {

public:

    int front, rear, size;

    unsigned capacity;

    int* array;

};
```

Data structure in c++

```
// function to create a queue
// of given capacity.
// It initializes size of queue as 0
Queue* createQueue(unsigned capacity)
{
    Queue* queue = new Queue();
    queue->capacity = capacity;
    queue->front = queue->size = 0;

    // This is important, see the enqueue
    queue->rear = capacity - 1;
    queue->array = new int[queue->capacity];
    return queue;
}

// Queue is full when size
// becomes equal to the capacity
int isFull(Queue* queue)
{
    return (queue->size == queue->capacity);
}

// Queue is empty when size is 0
int isEmpty(Queue* queue)
{
```

```
    return (queue->size == 0);

}


// Function to add an item to the queue.

// It changes rear and size

void enqueue(Queue* queue, int item)

{

    if (isFull(queue))

        return;

    queue->rear = (queue->rear + 1)

                  % queue->capacity;

    queue->array[queue->rear] = item;

    queue->size = queue->size + 1;

    cout << item << " enqueued to queue\n";

}


// Function to remove an item from queue.

// It changes front and size

int dequeue(Queue* queue)

{

    if (isEmpty(queue))

        return INT_MIN;

    int item = queue->array[queue->front];

    queue->front = (queue->front + 1)

                   % queue->capacity;
```

```
    queue->size = queue->size - 1;

    return item;

}


// Function to get front of queue

int front(Queue* queue)

{

    if (isEmpty(queue))

        return INT_MIN;

    return queue->array[queue->front];

}


// Function to get rear of queue

int rear(Queue* queue)

{

    if (isEmpty(queue))

        return INT_MIN;

    return queue->array[queue->rear];

}


// Driver code

int main()

{

    Queue* queue = createQueue(1000);
```

```
    enqueue(queue, 10);
    enqueue(queue, 20);
    enqueue(queue, 30);
    enqueue(queue, 40);

    cout << dequeue(queue)
         << " dequeued from queue\n";

    cout << "Front item is "
         << front(queue) << endl;
    cout << "Rear item is "
         << rear(queue) << endl;

    return 0;
}
```

.by using queue class :

```
#include <iostream>
#include <queue>

using namespace std;
```

Data structure in c++

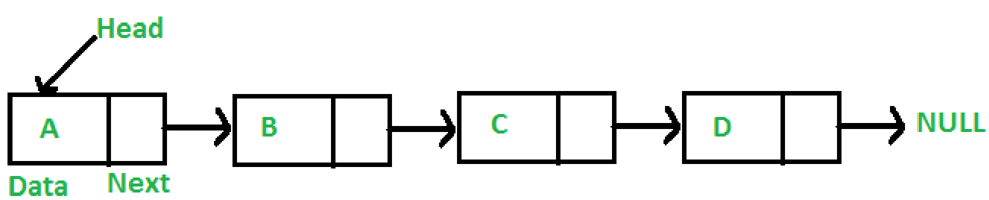
```
int main() {

  // create a queue of string
  queue<string> animals;


  // push elements into the queue
  animals.push("Cat");
  animals.push("Dog");


  cout << "Queue: ";

  // print elements of queue
  // loop until queue is empty
  while(!animals.empty()) {

    // print the element
```

Data structure in c++

```
    cout << animals.front() << ", ";


    // pop element from the queue

    animals.pop();

  }



  cout << endl;



  return 0;

}
```

Linked List Data Structure

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



Data structure in c++

```
#include <iostream>

using namespace std;


// A linked list node

struct Node

{

   int data;

   struct Node *next;

};

//insert a new node in front of the list

void push(struct Node** head, int node_data)

{

   /* 1. create and allocate node */

   struct Node* newNode = new Node;


   /* 2. assign data to node */

   newNode->data = node_data;


   /* 3. set next of new node as head */

   newNode->next = (*head);


   /* 4. move the head to point to the new node */

   (*head) = newNode;

}


//insert new node after a given node
```

Data structure in c++

```
void insertAfter(struct Node* prev_node, int node_data)

{

  /*1. check if the given prev_node is NULL */

if (prev_node == NULL)

{

   cout<<"the given previous node is required,cannot be NULL"; return; }


   /* 2. create and allocate new node */

   struct Node* newNode =new Node;


   /* 3. assign data to the node */

   newNode->data = node_data;


   /* 4. Make next of new node as next of prev_node */

   newNode->next = prev_node->next;


    /* 5. move the next of prev_node as new_node */

    prev_node->next = newNode;

}


/* insert new node at the end of the linked list */

void append(struct Node** head, int node_data)

{

/* 1. create and allocate node */

struct Node* newNode = new Node;
```

```
struct Node *last = *head; /* used in step 5*/


/* 2. assign data to the node */

newNode->data = node_data;


/* 3. set next pointer of new node to null as its the last node*/

newNode->next = NULL;


/* 4. if list is empty, new node becomes first node */

if (*head == NULL)

{

*head = newNode;

return;

}


/* 5. Else traverse till the last node */

while (last->next != NULL)

last = last->next;


/* 6. Change the next of last node */

last->next = newNode;

return;

}


// display linked list contents

void displayList(struct Node *node)
```

Data structure in c++

```
{
   //traverse the list to display each node
   while (node != NULL)
   {
      cout<<node->data<<"-->";
      node = node->next;
   }

if(node== NULL)
cout<<"null";
}
/* main program for linked list*/
int main()
{
/* empty list */
struct Node* head = NULL;

// Insert 10.
append(&head, 10);

// Insert 20 at the beginning.
push(&head, 20);

// Insert 30 at the beginning.
push(&head, 30);
```

Data structure in c++

```
// Insert 40 at the end.

append(&head, 40); //



Insert 50, after 20.

insertAfter(head->next, 50);



cout<<"Final linked list: "<<endl;

displayList(head);



return 0;

}
```

Output:

Final linked list:

30–>20–>50–>10–>40–>null