UNIT-2

Stack & Queue

13. Stack

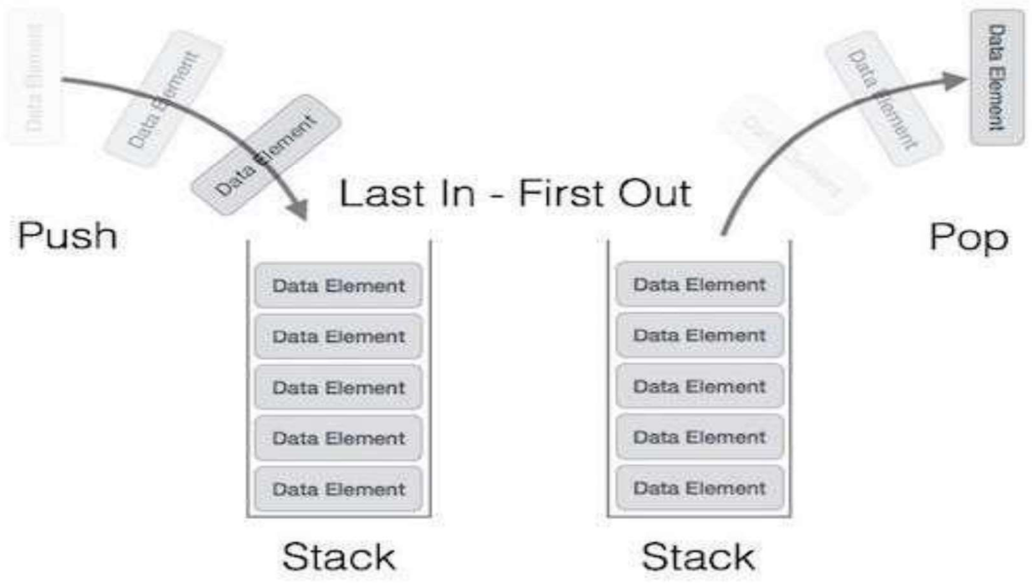
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations –



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations –

- **push()** – Pushing (storing) an element on the stack.
- **pop()** – Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks –

- **peek()** – get the top data element of the stack, without removing it.
- **isFull()** – check if stack is full.
- **isEmpty()** – check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions –

peek()

Algorithm of peek() function –

```
begin procedure peek

   return stack[top]

end procedure
```

Implementation of peek() function in C programming language –

```
int peek() {
   return stack[top];
}
```

isfull()

Algorithm of isfull() function –

```
begin procedure isfull


   if top equals to MAXSIZE
      return true
   else
      return
   false endif


end procedure
```

Implementation of isfull() function in C programming language –

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function –

```
begin procedure isempty


   if top less than 1
      return true
   else
      return
   false endif


end procedure
```

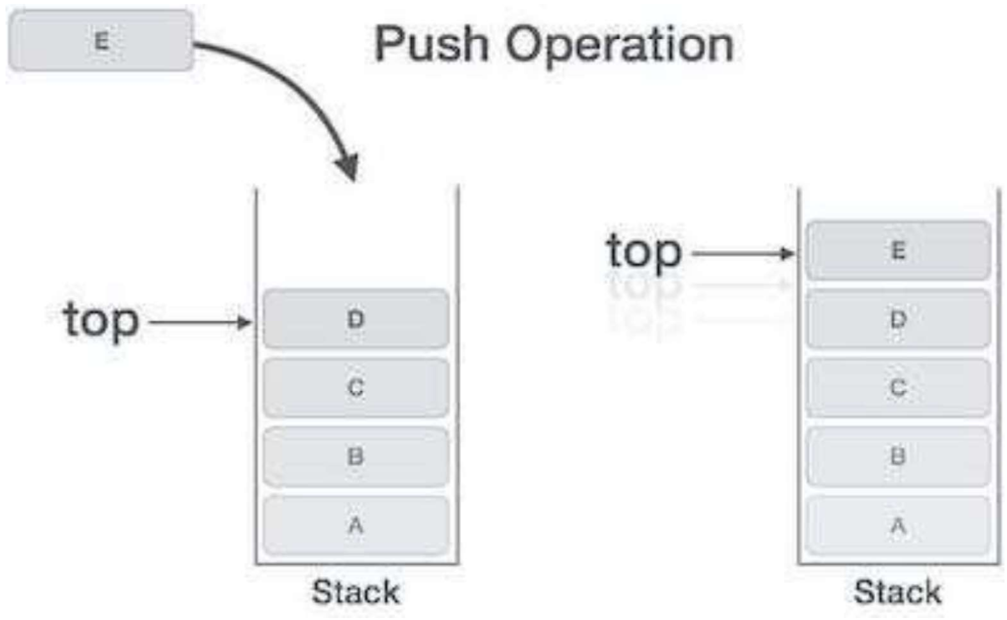
Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code –

```
bool isempty() {
   if(top == -1)
      return true;
   else
      return false;
}
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

- **Step 1** – Checks if the stack is full.
- **Step 2** – If the stack is full, produces an error and exit.
- **Step 3** – If the stack is not full, increments **top** to point next empty space.
- **Step 4** – Adds data element to the stack location, where top is pointing.
- **Step 5** – Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows –

```
begin procedure push: stack, data


   if stack is full
      return null
   endif


   top ← top + 1


   stack[top] ← data


end procedure
```

Implementation of this algorithm in C, is very easy. See the following code –

```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   }else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps –

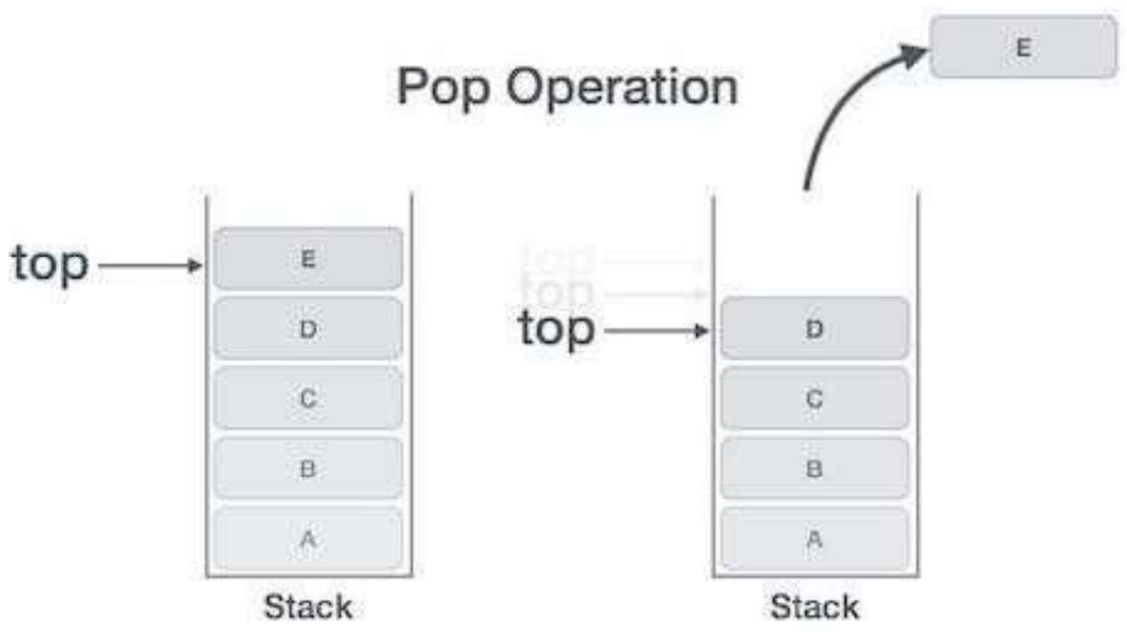
- **Step 1** – Checks if the stack is empty.

- **Step 2** – If the stack is empty, produces an error and exit.

- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.

- **Step 4** − Decreases the value of top by 1.

- **Step 5** − Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack


   if stack is empty
      return null
   endif


   data ← stack[top]


   top ← top - 1


   return data


end procedure
```

Implementation of this algorithm in C, is as follows –

```
int pop(int data) {


   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   }else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

For a complete stack program in C programming language, please click here.

Stack Program in C

We shall see the stack implementation in C programming language here. You can try the program by clicking on the Try-it button. To learn the theory aspect of stacks, click on visit previous page.

Implementation in C

```
#include <stdio.h>


int MAXSIZE = 8;
int stack[8];
int top = -1;

int isempty() {


   if(top == -1)
      return 1;
   else
      return 0;
}
```

```
int isfull() {

   if(top == MAXSIZE)
      return 1;
   else
      return 0;
}

int peek() {
   return stack[top];
}



int pop() {
   int data;

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   }else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}

int push(int data) {

   if(!isfull()) {
      top = top + 1;
      stack[top] =
   data; }else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

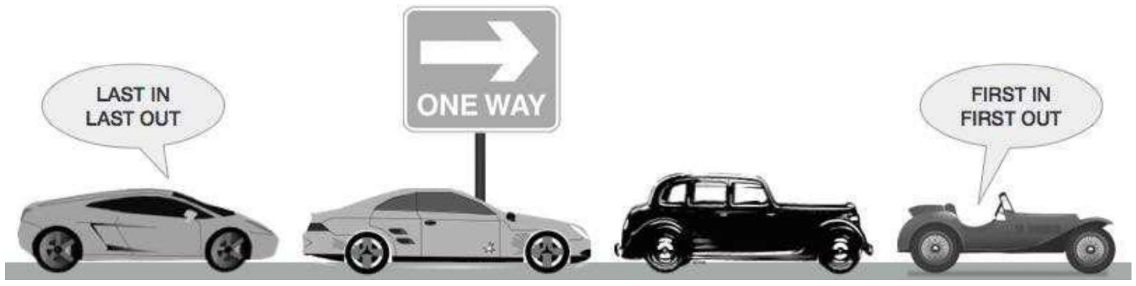
```
int main() {
    // push items on to the
    stack push(3);
    push(5);
    push(9);
    push(1);
    push(12);
    push(15);

    printf("Element at top of the stack: %d\n" ,peek());
    printf("Elements: \n");

     // print stack data
     while(!isempty()) {
       int data = pop();
       printf("%d\n",data);
    }

    printf("Stack full: %s\n" , isfull()?"true":"false");
    printf("Stack empty: %s\n" , isempty()?"true":"false");

    return 0;
 }
```

If we compile and run the above program, it will produce the following result –

```
 Element at top of the stack: 15
 Elements:
 15
 12
 1
 9
 5
 3
 Stack full: false

Stack empty: true
```

15. Queue

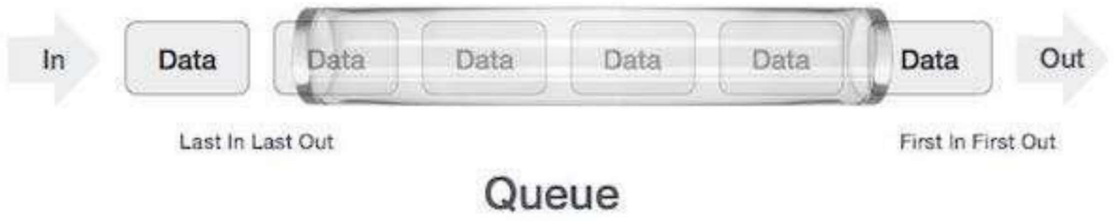
Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.



A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure –



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –

- **enqueue()** – add (store) an item to the queue.

- **dequeue()** – remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek()** − Gets the element at the front of the queue without removing it.
- **isfull()** − Checks if the queue is full.
- **isempty()** − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue −

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() function is as follows −

```
begin procedure peek


   return queue[front]


end procedure
```

Implementation of peek() function in C programming language −

```
int peek() {
   return queue[front];
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

```
begin procedure isfull


   if rear equals to MAXSIZE
      return true
   else
```

```
return false
   endif

end procedure
```

Implementation of isfull() function in C programming language –

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function –

```
begin procedure isempty

   if front is less than MIN OR front is greater than rear
      return true
   else
      return
   false endif

end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code –

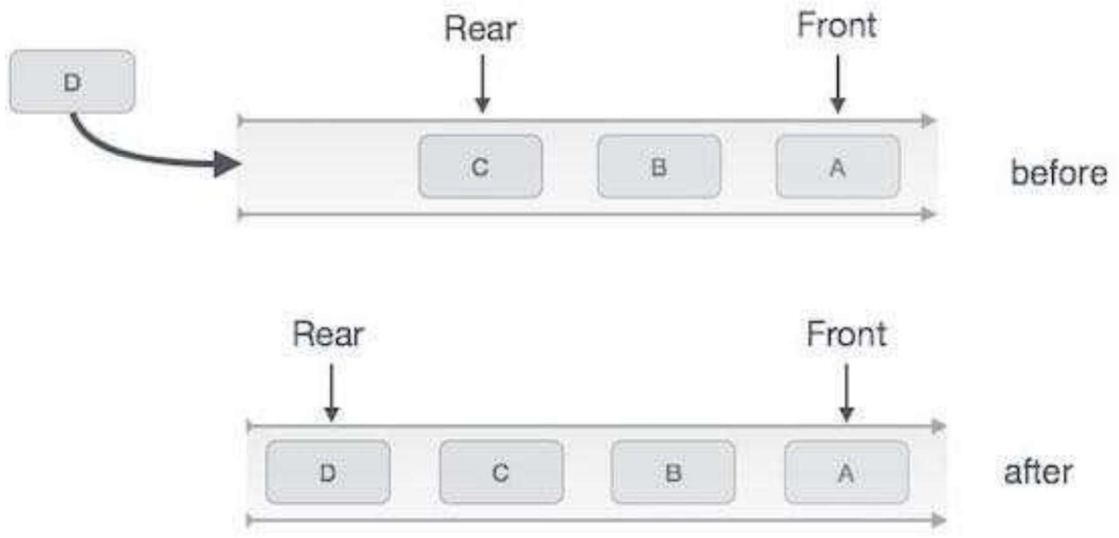
```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
}
```

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − Return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue Operation

```
procedure enqueue(data)
   if queue is full
      return
   overflow endif


   rear ← rear + 1
   queue[rear] ← data
   return true


end procedure
```

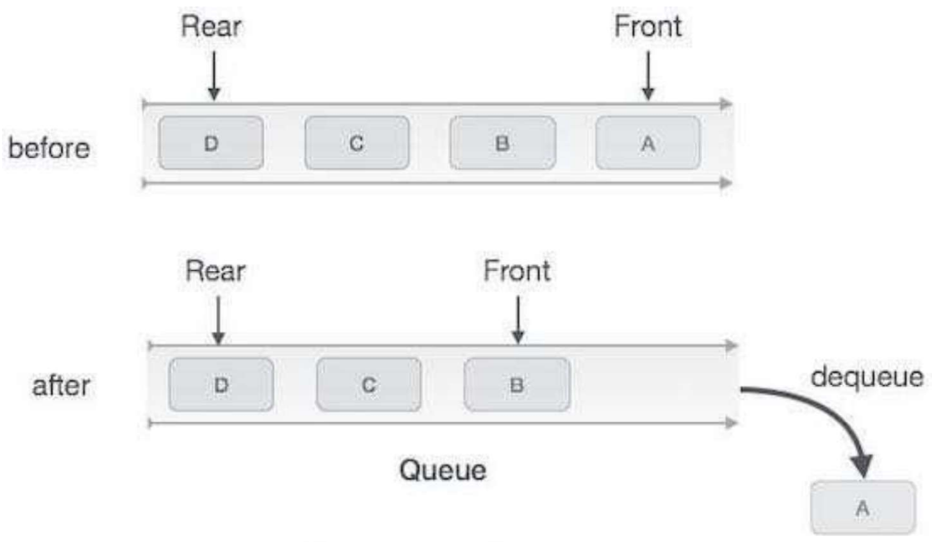
Implementation of enqueue() in C programming language –

```
int enqueue(int data)
   if(isfull())
      return 0;


   rear = rear + 1;
   queue[rear] = data;


   return 1;
end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks – access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation –

- **Step 1** – Check if the queue is empty.
- **Step 2** – If the queue is empty, produce underflow error and exit.
- **Step 3** – If the queue is not empty, access the data where **front** is pointing.
- **Step 4** – Increment **front** pointer to point to the next available data element.
- **Step 5** – Return success.



Queue Dequeue

Algorithm for dequeue Operation

```
procedure dequeue if
   queue is empty
      return
   underflow end if


   data = queue[front]
   front ← front + 1


   return true
end procedure
```

Implementation of dequeue() in C programming language –

```
int dequeue() {
   if(isempty())
      return 0;
   int data = queue[front];
   front = front + 1;


   return data;
}
```

Queue Program in C

We shall see the stack implementation in C programming language here. You can try the program by clicking on the Try-it button. To learn the theory aspect of stacks, click on visit previous page.

Implementation in C

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX 6

int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;

int peek(){
   return intArray[front];
}

bool isEmpty(){
   return itemCount == 0;
}
bool isFull(){
   return itemCount == MAX;
}

int size(){
   return itemCount;
}

void insert(int data){

   if(!isFull()){
      if(rear == MAX-
         1){ rear = -1;
      }

      intArray[++rear] = data;
      itemCount++;
   }
}

int removeData(){
   int data = intArray[front++];

   if(front == MAX){
      front = 0;
   }
   itemCount--;
   return data;
}

int main() {
   /* insert 5 items */
   insert(3);
   insert(5);
   insert(9);
   insert(1);
   insert(12);

   // front : 0
   // rear  : 4
   // ------------------
   // index : 0 1 2 3 4
   // ------------------
   // queue  :  3  5  9  1
   12 insert(15);

   // front : 0
   // rear  : 5
   // ---------------------
   // index : 0 1 2 3 4  5
   // ---------------------
   // queue : 3 5 9 1 12 15

   if(isFull()){
      printf("Queue is full!\n");
   }

   // remove one item
   int num = removeData();

   printf("Element removed: %d\n",num);
   // front : 1
   // rear  : 5
   // -------------------
   // index : 1 2 3 4  5
   // -------------------
   // queue : 5 9 1 12 15

   // insert       more
   items insert(16);

   // front : 1
   // rear  : -1
   // ----------------------
   // index : 0 1 2 3 4 5 //
   ----------------------
   // queue : 16 5 9 1 12 15

   // As   queue   is   full,   elements   will   not   be
   inserted. insert(17);
   insert(18);

   // ----------------------
   // index : 0 1 2 3 4 5 //
   ----------------------
```

```
   // queue : 16 5 9 1 12 15 printf("Element
   at front: %d\n",peek());

   printf("----------------------
   \n"); printf("index : 5 4 3 2 1
   0\n"); printf("-------------------
   ---\n"); printf("Queue: ");

   while(!isEmpty()){
      int n = removeData();
      printf("%d ",n);
   }
}
```

If we compile and run the above program, it will produce the following result –

```
Queue is full!
Element removed: 3
Element at front: 5
----------------------
index : 5 4 3 2 1 0
----------------------
Queue: 5 9 1 12 15 16
```

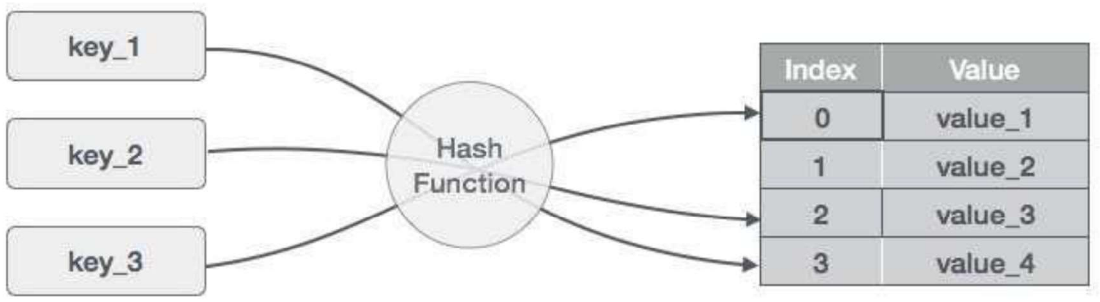
Hash Table

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

Sr. No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Linear Probing

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Sr. No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14

7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Basic Operations

Following are the basic primary operations of a hash table.

- **Search** – Searches an element in a hash table.
- **Insert** – inserts an element in a hash table.
- **Delete** – Deletes an element from a hash table.

Data Item

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

```
struct DataItem {
   int data;
   int key;
};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){
   return key % SIZE;
}
```

Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

```
struct DataItem *search(int key){
   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] != NULL){

      if(hashArray[hashIndex]->key == key)
         return hashArray[hashIndex];

      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   return NULL;
}
```

Insert Operation

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

```
void insert(int key,int data){
   struct DataItem *item = (struct DataItem*) malloc(sizeof(struct
   DataItem)); item->data = data;
   item->key = key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty or deleted cell while(hashArray[hashIndex]
   != NULL && hashArray[hashIndex]->key != -1){
      //go to next cell
      ++hashIndex;
      //wrap around the table
      hashIndex %= SIZE;
   }

   hashArray[hashIndex] = item;
}
```

Delete Operation

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

```
struct DataItem* delete(struct DataItem* item){
   int key = item->key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] !=NULL){

      if(hashArray[hashIndex]->key == key){
         struct DataItem* temp = hashArray[hashIndex];

         //assign a dummy item at deleted position
         hashArray[hashIndex] = dummyItem;
         return temp;
      }
      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }
   return NULL;
}
```

Hash Table Program in C

Hash Table is a data structure which stores data in an associative manner. In hash table, the data is stored in an array format where each data value has its own unique index value. Access of data becomes very fast, if we know the index of the desired data.

Implementation in C

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

#define SIZE 20

struct DataItem {
   int data;
   int key;
};

struct DataItem* hashArray[SIZE];
struct DataItem* dummyItem;
struct DataItem* item;

int hashCode(int key){
   return key % SIZE;
}

struct DataItem *search(int key){
   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] != NULL){

      if(hashArray[hashIndex]->key == key)
         return hashArray[hashIndex];
      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   return NULL;
}

void insert(int key,int data){

   struct DataItem *item = (struct DataItem*) malloc(sizeof(struct
   DataItem)); item->data = data;
   item->key = key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty or deleted cell while(hashArray[hashIndex]
   != NULL && hashArray[hashIndex]->key != -1){
      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   hashArray[hashIndex] = item;
}

struct DataItem* delete(struct DataItem* item){
   int key = item->key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] != NULL){

      if(hashArray[hashIndex]->key == key){
         struct DataItem* temp = hashArray[hashIndex];

         //assign a dummy item at deleted position
         hashArray[hashIndex] = dummyItem;
         return temp;
      }

      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   return NULL;
}

void display(){
   int i = 0;

   for(i = 0; i<SIZE; i++) {

      if(hashArray[i] != NULL)
         printf(" (%d,%d)",hashArray[i]->key,hashArray[i]->data);
      else
         printf(" ~~ ");
   }

   printf("\n");
}
```

```
int main(){
   dummyItem = (struct DataItem*) malloc(sizeof(struct
   DataItem)); dummyItem->data = -1;
   dummyItem->key = -1;

   insert(1, 20);
   insert(2, 70);
   insert(42, 80);
   insert(4, 25);
   insert(12, 44);
   insert(14, 32);
   insert(17, 11);
   insert(13, 78);
   insert(37, 97);

   display();

   item = search(37);

   if(item != NULL){
      printf("Element found: %d\n", item-
   >data); }else {
      printf("Element not found\n");
   }

   delete(item);

   item = search(37);

   if(item != NULL){
      printf("Element found: %d\n", item-
   >data); }else {
      printf("Element not found\n");
   }
}
```

If we compile and run the above program, it will produce the following result −

```
 ~~  (1,20)  (2,70)  (42,80)  (4,25)  ~~  ~~  ~~  ~~  ~~  ~~  ~~ (12,44)
(13,78)  (14,32)  ~~  ~~  (17,11)  (37,97)  ~~

Element found: 97

Element not  found
```