Report on
Data Structure

DATA STRUCTURE

MAHBUBUR RAHMAN

ID : 171442523

Data structures

Data structures are inevitable part of programs. Computer programs frequently process data so we require efficient ways in which we can access or manipulate data. Some applications may require modification of data frequently and in others new data is constantly added or deleted. So we need efficient ways of accessing data so as to act on it and build efficient applications.

Before Mid Trem Exam, we know the Topic . Which is :

- **About Function**
- **About Array**
  - ✓ **1D Array**
  - ✓ **2D Array**
- **About Stack**
- **About Queue**
- **About Structure**
- **About Linked List**

The data structure can be subdivided into major types:

- **Linear Data Structure**
- **Non-linear Data Structure**

The common examples of the linear data structure are:

- **Arrays**
- **Queues**
- **Stacks**
- **Linked lists**

FUNCTION

A function is a group of statements that together perform a task. A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function. The C standard library provides numerous built-in functions that your program can call.

Example:-

```
#include<stdio.h>
#include<conio.h>
int sum(int,int);
void main()
{
int a,b,c;
printf("\nEnter the two numbers : ");
scanf("%d%d",&a,&b);
c = sum(a,b);
printf("\nAddition of two number is : %d",c);
getch();
}
int sum (int num1,int num2)
{
```

ARRAY

Array is a linear data structure. It is a collection of similar data items which may be integer, float etc... or any user defined type such as structure. Elements are stored in consecutive memory locations. Elements referred by the common name, i.e. Array name.

E.g. int a[5]

Here,

a is the array which can hold upto 5 values of type integer and Index locations are 0 to 4.

If

```
int a[5]= { 1,2,3,4,5}
```

Then a[0]= 1, a[1]=2, a[2]=3, a[3]=4, a[4] =5

There are mainly 2 types of array.

1. One - dimensional Array:

It is defined by specifying its name, size, data types of the items

Syntax:

```
1.    DataType ArrayName[size] ;
```

E.g.

```
1.    float number [10];
```

2. Two dimensional Array:

They are defined in number of square brackets are more in multi dimensional Array. There will be two pairs of square brackets for 2-dimensional Array, three pair for 3- dimensional Array and so on...

Syntax:

```
1.DataType ArrayName [m1][m2]...[mn];
```

Where m1,m2...mn are the dimensions of the array.

E.g. For two dimensions array,

```
1.int array[3][4];
```

int array [3][4] can contains 3*4 = 12 elements.

If

```
1.int array [3][2]={ 1,2,3,4,5,6};
2.int i,j;
```

Then

array[0][0]=1

array[0][1]=2

array[1][0]=3 , so on..

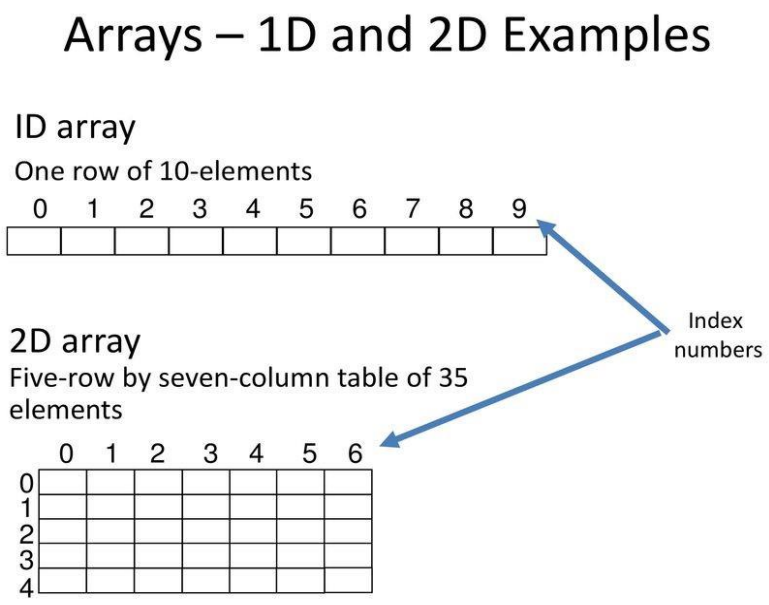
Diagrammatic:

| 1 2 |

| 3 4 |

| 5 6 |

Graphical examples of 1D and 2D are showing below,



Simple programs of 1D and 2D array,

1D array:

```
#include<stdio.h>
int main()
{
int a[10];
int i, n;
printf("How many element you want to save \n");
scanf("%d", &n);
printf("Enter element one by one \n");
for(i = 0; i<n; i++)
{
scanf("%d", &a[i]);
}
printf("The List you entered\n");
for(i = 0; i<n; i++)
{
printf("%d ", a[i]);
}
return 1;
}
```

Output will be:

How many element you want to save

3

Enter element one by one

10

20

30

The List you entered

10 20 30

2D array:

```
#include<stdio.h>
void main(){
int a[10][10];
int i,j,rows, columns;
printf("How many rows you want \n");
scanf("%d", &rows);
printf("How many columns you want \n");
scanf("%d", &columns);
printf("Enter your array Element one by one \n");
for(i=0;i<rows;i++){
   for(j=0;j<columns;j++){
        scanf("%d", &a[i][j]);
   }
}
printf("Your array \n");
for(i=0;i<rows;i++){
   for(j=0;j<columns;j++){
        printf("%d\t ", a[i][j]);
   }
   printf("\n");
}
}
```

Output will be:

How many rows you want

2

How many columns you want

2

Enter your array Element one by one

2

2

2

2

Your array

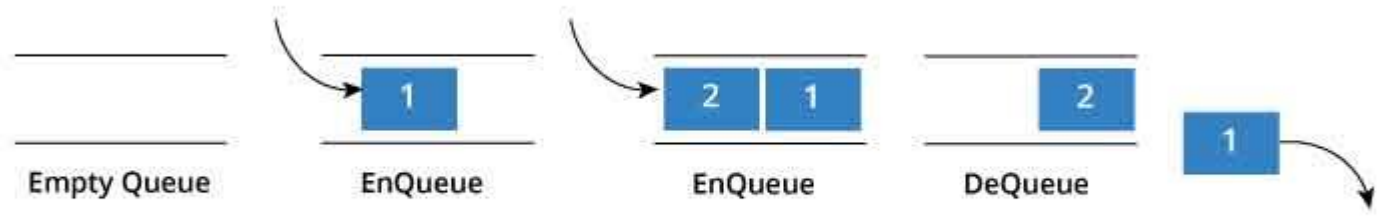
2 2

2 2

Queue

A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

Queue follows the First In First Out(FIFO) rule - the item that goes in first is the item that comes out first too.



In the above image, since 1 was kept in the queue before 2, it was the first to be removed from the queue as well. It follows the FIFO rule.

In programming terms, putting an item in the queue is called an "enqueue" and removing an item from the queue is called "dequeue".

We can implement queue in any programming language like C, C++, Java, Python or C#, but the specification is pretty much the same.

Queue Specifications

A queue is an object or more specifically an abstract data structure(ADT) that allows the following operations:

- **Enqueue: Add element to end of queue**
- **Dequeue: Remove element from front of queue**
- **IsEmpty: Check if queue is empty**
- **IsFull: Check if queue is full**
- **Peek: Get the value of the front of queue without removing it**

How Queue Works

Queue operations work as follows:

1. **Two pointers called `FRONT` and `REAR` are used to keep track of the first and last elements in the queue.**
2. **When initializing the queue, we set the value of `FRONT` and `REAR` to -1.**
3. **On enqueing an element, we increase the value of `REAR` index and place the new element in the position pointed to by `REAR`.**
4. **On dequeueing an element, we return the value pointed to by `FRONT` and increase the `FRONT` index.**
5. **Before enqueing, we check if queue is already full.**
6. **Before dequeuing, we check if queue is already empty.**
7. **When enqueing the first element, we set the value of `FRONT` to 0.**
8. **When dequeing the last element, we reset the values of `FRONT` and `REAR` to -1.**

Queue implementation in C language:

```
#include<stdio.h>

#define SIZE 100                //Queue size declaration

int queue[SIZE], head = -1, tail = 0; //Empty Queue declaration


void enQueue(int value){

  if(head == SIZE-1)
```

```
    printf("\nQueue is Full!!!");
  else{
    head++;
    queue[head] = value;
  }
}

void deQueue(){
  if(head == -1)
    printf("\nQueue is Empty!!! ");
  else{
    printf("\nDeleted: %d", queue[tail]);
    tail++;
  }
}

void display(){
  if(head == -1)
    printf("\nQueue is Empty!!!");
  else{
    int i;
    printf("\nQueue elements are:\n");
    for(i=tail; i<=head; i++)
          printf("%d\t",queue[i]);
  }
}

void main()
{

enQueue(10);
enQueue(20);
enQueue(30);
enQueue(40);
display();
deQueue();
display();
deQueue();
display();
```

```
}
```

Output will be:

Queue elements are:

10 20 30 40

Deleted: 10

Queue elements are:

20 30 40

Deleted: 20

Queue elements are:

30 40

- **Linked List**

A linked list is represented by a pointer to the first node of the linked list. The first node is called head. If the linked list is empty, then value of head is NULL. In C, we can represent a node using structures. Linked list consists many node structures those are point each other and create a link.

```
struct node       //Node Structure
{
 int data;
 struct node *next;
};
```

How to traverse a linked list

Displaying the contents of a linked list is very simple. We keep moving the temp node to the next one and display its contents.

When temp is NULL, we know that we have reached the end of linked list so we get out of the while loop.

```
struct node *temp = head;
printf("\n\nList elements are - \n");
while(temp != NULL)
{
    printf("%d --->",temp->data);
    temp = temp->next;
}
```

The output of this program will be:

List elements are -

1 --->2 --->3 --->

How to add elements to linked list

We can add elements to either beginning, middle or end of linked list. Here we will see only the beginning & end operations.

Add to beginning

- ***Allocate memory for new node***
- ***Store data***
- ***Change next of new node to point to head***
- ***Change head to point to recently created node***

For an example,

```
struct node *newNode;

newNode = malloc(sizeof(struct node));

newNode->data = 4;

newNode->next = head;

head = newNode;
```

Add to end

- **Allocate memory for new node**
- **Store data**
- **Traverse to last node**
- **Change next of last node to recently created node**

```
struct node *newNode;

newNode = malloc(sizeof(struct node));

newNode->data = 4;

newNode->next = NULL;


struct node *temp = head;

while(temp->next != NULL){

  temp = temp->next;

}

temp->next = newNode;
```

How to delete from a linked list

You can delete either from beginning, end or from a particular position.

Delete from beginning

- **Point head to the second node**

```
head = head->next;
```

Delete from end

- Traverse to second last element
- Change its next pointer to null

```
struct node* temp = head;
while(temp->next->next!=NULL){
  temp = temp->next;
}
temp->next = NULL;
```

Complete program for linked list operations is showing below,

```
#include <stdio.h>
struct Node
{
  int data;
  struct Node *next;
};
struct Node *head = NULL;

void insertAtBeginning(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode->data = value;
  newNode->next = NULL;
  if(head == NULL)
  {
    head = newNode;
  }
  else
  {
    newNode->next = head;
    head = newNode;
  }

}
void display()
{
  if(head == NULL)
```

```
{
   printf("\nList is Empty\n");
}
else
{
   struct Node *temp = head;
   printf("\n\nList elements are - \n");
   while(temp->next != NULL)
   {
    printf("%d ",temp->data);
    temp = temp->next;
     }
    printf("%d ",temp->data);
   }
}
void insertAtEnd(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->next = NULL;
   if(head == NULL)
   head = newNode;
   else
   {
      struct Node *temp = head;
      while(temp->next != NULL){
       temp = temp->next;
      }
      temp->next = newNode;
   }

}
void removeBeginning()
{
   if(head == NULL)
   printf("\n\nList is Empty!!!");
   else
   {
```

```
   struct Node *temp = head;
   if(temp->next == NULL)
   {
    head = NULL;
    free(temp);
   }
   else
   {
   head = temp->next;
   free(temp);
   }
}

void removeEnd()
{
  if(head == NULL)
  {
    printf("\nList is Empty!!!\n");
  }
  else
  {
    struct Node *temp1 = head,*temp2;
    if(temp1->next == NULL)
    {
     head = NULL;
    }
    else
    {
     while(temp1->next != NULL)
     {
       temp2 = temp1;
       temp1 = temp1->next;
     }
     temp2->next = NULL;
     }
       free(temp1);
   }
}
```

```
int main()
{
  insertAtEnd(10);// 10
  insertAtEnd(20); // 10 20
  insertAtBeginning(40); // 40 10 20
  insertAtBeginning(50); // 50 40 10 20
  insertAtEnd(100); // 50 40 10 20 100
  display();
  removeBeginning(); // 40 10 20 100
  display();
  removeEnd(); // 40 10 20
  display();
   return 0;
}
```

Output will be:

List elements are -

50 40 10 20 100

List elements are -

40 10 20 100

List elements are -

40 10 20