Data Structure

Report

SUBMITTED TO
Richard Philip
Department Of CSE

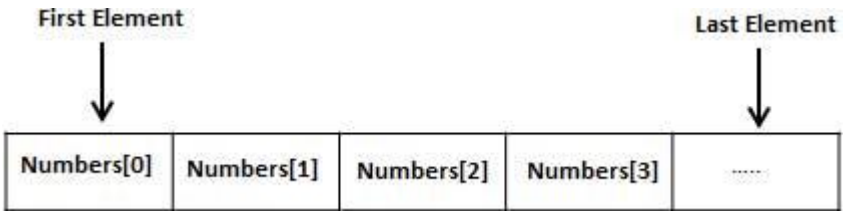
SUBMITTED BY
Sumaiya Zaman
Id# 171442649
Batch: 44th

1 Contents

2 ARRAY

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



There are two type of arrays, they are:

1. One-dimensional arrays
2. Multidimensional arrays

2.1 ONE DIMENTIONAL ARRAY

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows –

type arrayName [arraySize];

This is called a single-dimensional array. The arraySize must be an integer constant greater than zero and type can be any valid C data type.

For example,

float mark[5]

Here, we declared an array, *mark*, of floating-point type and size 5. Meaning, it can hold 5 floating-point values.

2.1.1 Initializing

There are various way to initial a array. They are:

- It's possible to initialize an array during declaration. For example,

  int mark[5] = {19, 10, 8, 17, 9};

- Another method to initialize array during declaration:

  int mark[] = {19, 10, 8, 17, 9};

2.1.2 Insert & Print

Step 1: Declare array

Step 2: Insert the number of element, n of the array

Step 3: Use for loop to scan and print 0 to n element.

For Example

```
for ( i = 0; i < n; i++ )
     {
         printf("a[%d][%d] = %d\n", i,j, a[i][j] );
```

2.2 Problem & Solution of One Dimensional Array

https://github.com/Sanemzm/DATA-STRUCTURE/tree/master/Mid%20term/one%20dimentional%20array%20problem%20%26%20solution

2.3 MULTI-DIMENTIONAL ARRAY

C programming language allows multidimensional arrays. The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows –

type arrayName [x][y];

A two-dimensional array **a**, which contains three rows and four columns can be shown as follows –

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in the array a is identified by an element name of the form a[i][j], where 'a' is the name of the array, and 'i' and 'j' are the subscripts that uniquely identify each element in 'a'.

2.3.1 Initializing

- It's possible to initialize an array during declaration. For example,

  int mark[2][2] = {10, 8, 17, 9};

2.3.2 Insert & Print

Step 1: Declare array

Step 2: Insert the number of row and column of the array

Step 3: Use two for loop to scan and print elements.

For example,

```
for ( i = 0; i < row; i++ ) {

      for ( j = 0; j < column; j++ ) {
         printf("%d", i,j, array[i][j] );
      }
   }
```

2.4 Problem & Solution of Two Dimensional Array

https://github.com/Sanemzm/DATA-STRUCTURE/tree/master/Mid%20term/Two%20Dimentional%20Array%20problem%20%26%20Solution

3 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).

Mainly the following three basic operations are performed in the stack:

3.1 **Push:**

Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.

For Example

```
int push(int data) {

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

3.2 **Pop:**

Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

For Example

```
int pop() {
   int data;
   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Stack is empty.\n");
   }
}
```

3.3 Peek:

Returns top element of stack.

For Example

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

There are other two operations to check if the stack is full or emty. The are

3.4 ISEMTY:

Returns true if stack is empty, else false.

For Example

```
int isempty() {

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

3.5 ISFULL

Returns true if stack is full, else false.

For Example

```
int isfull() {

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

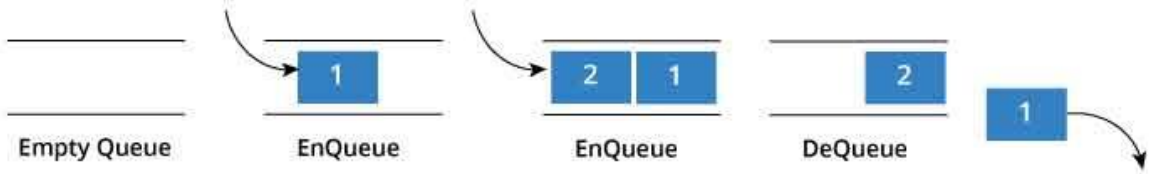
3.6 Problem & Solution of Stack

https://github.com/Sanemzm/DATA-STRUCTURE/tree/master/Mid%20term/Stack%20problem%20%26%20solution

4 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".

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

4.1 Enqueue:

Add element to end of queue, having a particular pointer .

```
enQueue(int value){
if(head == SIZE-1)
printf("\nQueue is Full!!!");
else{
head++;
queue[head] = value;
}
```

4.2 Dequeue

Remove element from front of queue, having a particular pointer

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

4.3 IsEmty:

Check if queue is empty

```
int isempty() {

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

4.4 IsFull:

Check if queue is full

```
int isfull() {

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

4.5 Peek:

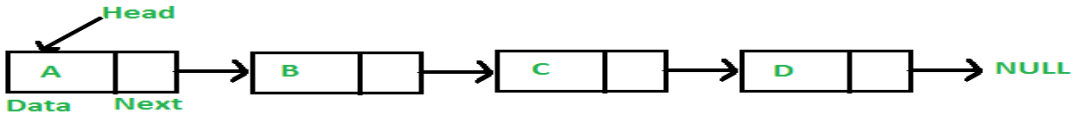
Get the value of the front of queue without removing it

4.6 Problem & Solution of QUEUE

https://github.com/Sanemzm/DATA-STRUCTURE/tree/master/Mid%20term/Queue%20problem%20%26%20Solution

5 LINKEDLIST

Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at contiguous location; the elements are linked using pointers.



5.1 Single Linkedlist

In a single linked list we perform the following operations...

1. Insertion
2. Deletion
3. Display

STRUCT A SINGLE LINKLIST NODE

```
struct Node
{
  int data;
  struct Node *next;
};
```

5.1.1Insertion

In a single linked list the insertion operation can be performed in those are as follows...

1. Inserting At Beginning of the list

2. Inserting At End of the list

5.1.1.1 Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list

```
/* Given a reference (pointer to pointer) to the head of a list
   and an int,  inserts a new node on the front of the list. */
void push(int new_data)
{
    /* 1. allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));

    /* 2. put in the data  */
    new_node->data  = new_data;

    /* 3. Make next of new node as head */
    new_node->next = (*head_ref);

    /* 4. move the head to point to the new node */
    (*head_ref)    = new_node;
}
```

5.1.1.2 Inserting At End of the list

We can use the following steps to insert a new node at ending of the single linked list...

```
   /* 1. allocate node */
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
    struct Node *last = *head_ref;  /* used in step 5*/
    /* 2. put in the data  */
    new_node->data  = new_data;
    /* 3. This new node is going to be the last node, so make next
          of it as NULL*/
    new_node->next = NULL;
    /* 4. If the Linked List is empty, then make the new node as head
*/
    if (*head_ref == NULL)
    {
       *head_ref = new_node;
    }
    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;

    /* 6. Change the next of last node */
    last->next = new_node;
```

5.1.1.3 Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list...

```
/* Given a node prev_node, insert a new node after the given
   prev_node */
void insertAfter(struct Node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
       printf("the given previous node cannot be NULL");
       return;
    }

    /* 2. allocate new node */
    struct Node* new_node =(struct Node*) malloc(sizeof(struct Node));

    /* 3. put in the data  */
    new_node->data  = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. move the next of prev_node as new_node */
    prev_node->next = new_node;
}
```

5.1.2 Deletion

In a single linked list the deletion operation can be performed in three ways those are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list

5.1.2.1 Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

```
void deleteBeginning()
{
/* 1.Check whether list is Empty (head == NULL)*/
   if(head == NULL)
/*  2. display 'List is Empty!!! Deletion is not possible' and terminat the function */
      printf("List is Empty!!! Deletion not possible!!!");
   else
   {
/* 3. a Node pointer 'temp' and initialize with head*/
      struct Node *temp = head;
/* 4. Check whether list is having only one node */
      if(temp -> next == NULL)
      {
/*5.If it is TRUE then set head = NULL and delete temp*/
         head = NULL;
         free(temp);
      }
      else{
         head = head -> next;
         free(temp);
      }
   }
}
```

5.1.2.2 Deleting from End of the list

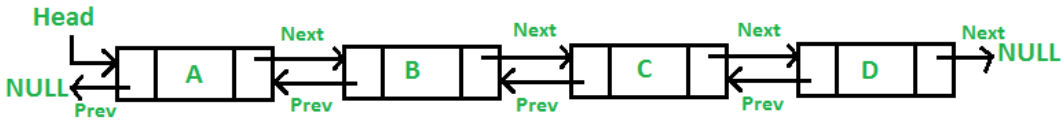
We can use the following steps to delete a node from end of the single linked list

```
void deleteEnd()
{
/*1: Check whether list is Empty (head == NULL)*/

   if(head == NULL)
      printf("List is Empty!!! Deletion not possible!!!");
   else
   {
/*2:define two Node pointers 'temp1' and 'temp2' and initialize 'temp1' with
head*/
      struct Node *temp1 = head, temp2;
/* 3.Check whether list has only one Node*/
      if(temp1 -> next == NULL)
/* 4.set head = NULL and delete temp1. And terminate from the function */
      {
         head = NULL;
         free(temp1);
      }
      else
/* 5.set 'temp2 = temp1 ' and move temp1 to its next node. Repeat the same
until it reach to the last node in the list*/
{
         while(temp1 -> next != NULL){
            temp2 = temp1;
            temp1 = temp1 -> next;
         }
/* 6. Set temp2 → next = NULL and delete temp1*/
         temp2 -> next = NULL;
         free(temp1);
```

Double Linked List

A **D**oubly **L**inked **L**ist (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.



Following is representation of a DLL node in C language.

```
/* Node of a doubly linked list */
struct Node {
    int data;
    struct Node* next; // Pointer to next node in DLL
    struct Node* prev; // Pointer to previous node in DLL
};
```

5.1.3 Insertion

A node can be added in following ways

1) At the front of the DLL
2) At the end of the DLL

5.1.3.1 At the front of the DLL

We can use the following steps to insert a new node at beginning of the double linked list

```
/* 1. allocate node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

    /* 2. put in the data  */
    new_node->data = new_data;

    /* 3. Make next of new node as head and previous as NULL */
    new_node->next = (*head_ref);
    new_node->prev = NULL;

    /* 4. change prev of head node to new node */
    if ((*head_ref) != NULL)
        (*head_ref)->prev = new_node;

    /* 5. move the head to point to the new node */
    (*head_ref) = new_node;
```

5.1.3.2 At the end of the DLL

We can use the following steps to insert a new node at the end of the double linked list

```
/* 1. allocate node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct
Node));
    struct Node* last = *head_ref; /* used in step 5*/
    /* 2. put in the data  */
    new_node->data = new_data;
    /* 3. This new node is going to be the last node, so
          make next of it as NULL*/
    new_node->next = NULL;
    /* 4. If the Linked List is empty, then make the new
          node as head */
    if (*head_ref == NULL) {
        new_node->prev = NULL;
        *head_ref = new_node; }
    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;
    /* 6. Change the next of last node */
    last->next = new_node;
    /* 7. Make last node as previous of new node */
    new node->prev = last;
```

5.1.4 Delete a node

A node can be deleted in following ways

1) From the front of DLL
2) From the end of the DLL

5.1.4.1 Delete at the beginning & end

We can use the following steps to delete a new node from the end of the double linked list

```
/* base case */
  if(*head_ref == NULL || del == NULL)
    return;
  /* If node to be deleted is head node */
  if(*head_ref == del)
    *head_ref = del->next;

  /* Change next only if node to be deleted is NOT the last node */
  if(del->next != NULL)
    del->next->prev = del->prev;

  /* Change prev only if node to be deleted is NOT the first node */
  if(del->prev != NULL)
    del->prev->next = del->next;

  /* Finally, free the memory occupied by del*/
  free(del);
```

5.2 Problem & Solution of linked list

https://github.com/Sanemzm/DATA-STRUCTURE/tree/master/Mid%20term/Linked%20list%20problem%20%26%20Solution

6 Shorting

Sorting is nothing but arranging the data in ascending or descending order. The term sorting came into picture, as humans realised the importance of searching quickly.

There are so many things in our real life that we need to search for, like a particular record in database, roll numbers in merit list, a particular telephone number in telephone directory, a particular page in a book etc. All this would have been a mess if the data was kept unordered and unsorted, but fortunately the concept of sorting came into existence, making it easier for everyone to arrange data in an order, hence making it easier to search.

Sorting arranges data in a sequence which makes searching easier.

6.1 Sorting Efficiency

If you ask me, how will I arrange a deck of shuffled cards in order, I would say, I will start by checking every card, and making the deck as I move on.

It can take me hours to arrange the deck in order, but that's how I will do it.

Well, thank god, computers don't work like this.

Since the beginning of the programming age, computer scientists have been working on solving the problem of sorting by coming up with various different algorithms to sort data.

The two main criterias to judge which algorithm is better than the other have been:

1. Time taken to sort the given data.
2. Memory Space required to do so.

6.2 Different Sorting Algorithms

There are many different techniques available for sorting, differentiated by their efficiency and space requirements. Following are some sorting techniques which we will be covering in next few tutorials.

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort
6. Heap Sort

6.3 Problem & Solution of Shorting

https://github.com/Sanemzm/DATA-STRUCTURE/tree/master/Final/Sorting

7 Binary Search Tree

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.

7.1 Problem & Solution of Shorting

https://github.com/Sanemzm/DATA-STRUCTURE/tree/master/Final/Binary%20Search%20Tree

References:

- https://www.geeksforgeeks.org/**array**-data-structure/
- https://www.geeksforgeeks.org/**stack**-data-structure/
- https://www.geeksforgeeks.org/**queue**-data-structure/
- https://www.geeksforgeeks.org/data-structures/**linked-list**/