DS FILE



**Submitted to: Submitted by:**

DR. NISHTA JATANA Name: Ambuj Kumar

Enrol. No: 00796302820

ECE SHIFT: 2ND

**2020-2024**

**EXPERIMENT NO.-01**

**AIM: Basic about data structure**

* 1. **Definition of data structure?**

**Ans.**A data structure is a specialized format for organizing, processing, retrieving and storing data. ... In computer science and computer programming, a data structure may be selected or designed to store data for the purpose of using it with various algorithms.

* 1. **Types of data structure?**

Ans.

**Arrays:** An array is a collection of similar type of data items and each data item is called an element of the array. The data type of the element may be any valid data type like char, int, float or double.

The elements of array share the same variable name but each one carries a different index number known as subscript. The array can be one dimensional, two dimensional or multidimensional.

The individual elements of the array age are:

age[0], age[1], age[2], age[3],......... age[98], age[99].

**Linked List:** Linked list is a linear data structure which is used to maintain a list in the memory. It can be seen as the collection of nodes stored at non-contiguous memory locations. Each node of the list contains a pointer to its adjacent node.

**Stack:** Stack is a linear list in which insertion and deletions are allowed only at one end, called **top**.

A stack is an abstract data type (ADT), can be implemented in most of the programming languages. It is named as stack because it behaves like a real-world stack, for example: - piles of plates or deck of cards etc.

**Queue:** Queue is a linear list in which elements can be inserted only at one end called **rear** and deleted only at the other end called **front**.

It is an abstract data structure, similar to stack. Queue is opened at both end therefore it follows First-In-First-Out (FIFO) methodology for storing the data items.

**Trees:** Trees are multilevel data structures with a hierarchical relationship among its elements known as nodes. The bottommost nodes in the hierarchy are called **leaf node** while the topmost node is called **root node**. Each node contains pointers to point adjacent nodes.

Tree data structure is based on the parent-child relationship among the nodes. Each node in the tree can have more than one children except the leaf nodes whereas each node can have atmost one parent except the root node. Trees can be classified into many categories which will be discussed later in this tutorial.

**Graphs:** Graphs can be defined as the pictorial representation of the set of elements (represented by vertices) connected by the links known as edges. A graph is different from tree in the sense that a graph can have cycle while the tree cannot have the one.

* 1. **Operation on data structure**

Ans.

1) **Traversing:** Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

**Example:** If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will divide that sum by the number of subjects i.e. 6, in order to find the average.

2) **Insertion:** Insertion can be defined as the process of adding the elements to the data structure at any location.

If the size of data structure is **n** then we can only insert **n-1** data elements into it.

3) **Deletion:** The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location.

If we try to delete an element from an empty data structure, then **underflow** occurs.

4) **Searching:** The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search. We will discuss each one of them later in this tutorial.

5) **Sorting:** The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) **Merging:** When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

1.4 **Complexities of data structures**

Time complexity

Time complexity is **the amount of time taken by an algorithm to run**, as a function of the length of the input. It measures the time taken to execute each statement of code in an algorithm

Space Complexity

The space complexity of an algorithm or a computer program is **the amount of memory space required to solve an instance of** the computational problem as a function of characteristics of the input. It is the memory required by an algorithm until it executes completely.

Time Space Trade-off Complexity

In computer science, a space-time or time-memory trade-off is a **way of solving a problem or calculation in less time by using more storage space** (or memory), or by solving a problem in very little space by spending a long time.

**Average time complexity of different data structures for different operations**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Data structure** | **Access** | **Search** | **Insertion** | **Deletion** |
| **Array** | O(1) | O(N) | O(N) | O(N) |
| **Stack** | O(N) | O(N) | O(1) | O(1) |
| **Queue** | O(N) | O(N) | O(1) | O(1) |
| **Singly Linked list** | O(N) | O(N) | O(1) | O(1) |
| **Doubly Linked List** | O(N) | O(N) | O(1) | O(1) |
| **Binary Search Tree** | O(log N) | O(log N) | O(log N) | O(log N) |
| **AVL Tree** | O(log N) | O(log N) | O(log N) | O(log N) |
| **B Tree** | O(log N) | O(log N) | O(log N) | O(log N) |

**Worst Case time complexity of different data structures for different operations**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Data structure** | **Access** | **Search** | **Insertion** | **Deletion** |
| **Array** | O(1) | O(N) | O(N) | O(N) |
| **Stack** | O(N) | O(N) | O(1) | O(1) |
| **Queue** | O(N) | O(N) | O(1) | O(1) |
| **Singly Linked list** | O(N) | O(N) | O(1) | O(1) |
| **Doubly Linked List** | O(N) | O(N) | O(1) | O(1) |
| **Binary Search Tree** | O(N) | O(N) | O(N) | O(N) |
| **AVL Tree** | O(log N) | O(log N) | O(log N) | O(log N) |
| **Binary Tree** | O(N) | O(N) | O(N) | O(N) |

**EXPERIMENT NO.-02**

**AIM: Write a program of linear Search**

**Code:**

#include <bits/stdc++.h>

using namespace std;

#define ll long long int

int linear\_search(vector<int> vec1, int target)

{

for (int i = 0; i < vec1.size(); i++)

{

if (vec1[i] == target)

{

return i + 1;

}

}

return -1;

}

int main()

{

ios\_base::sync\_with\_stdio(false);

cin.tie(nullptr);

cout.tie(nullptr);

vector<int> vec1 = {7,6,8,2,9,3,4};

cout<<linear\_search(vec1,4);

return 0;

}

**EXPERIMENT NO.-03**

**AIM: Write a program of binary Search**

**Code:**

#include <bits/stdc++.h>

using namespace std;

#define ll long long int

int binary\_search(int \*arr, int size, int target)

{

int low = 0;

int mid, high = size - 1;

while (low <= high)

{

int mid = low + (high - low) / 2;

if (arr[mid] > target)

{

high = mid - 1;

}

else if (arr[mid] < target)

{

low = mid + 1;

}

else if (arr[mid] == target)

return mid+1;

}

return -1;

}

int main()

{

ios\_base::sync\_with\_stdio(false);

cin.tie(nullptr);

cout.tie(nullptr);

int arr[] = {2,4,6,8,10};

int size = sizeof(arr)/sizeof(int);

cout<<binary\_search(arr,size,10);

return 0;

}

**EXPERIMENT NO.-04**

**AIM: Write a program of Operations and Implimentation Stack Data Structure**

**Code:**

#include <iostream>

using namespace std;

class stack

{

public:

int size;

int top = -1;

int \*arr;

};

int isFull(stack \*s)

{

if (s->top == s->size - 1)

{

return 1;

}

return 0;

}

int isEmpty(stack \*s)

{

if (s->top == -1)

{

return 1;

}

return 0;

}

void push(stack \*s,int data)

{

if(isFull(s))

{

cout<<"Stackoverflow"<<endl;

return;

}

else

{

s->arr[s->top+1] = data;

s->top++;

}

}

int pop(stack \*s)

{

if(isEmpty(s))

{

cout<<"StackUnderflow"<<endl;

return 404;

}

else

{

int element = s->arr[s->top];

s->top--;

return element;

}

}

int peek(stack \*s , int index)

{

int Inindex = s->top - index +1;

if (Inindex < 0)

{

cout<<"Invalid Index"<<endl;

return 101;

}

else

{

return s->arr[Inindex];

}

}

int main()

{

stack \*s = new stack;

s->size = 50;

s->arr = new int[s->size];

push(s,5);

push(s,15);

push(s,25);

push(s,35);

push(s,35);

push(s,35);

push(s,35);

pop(s);

pop(s);

pop(s);

pop(s);

for (int i = 1 ; i <= s->top+1; i++)

{

cout<<"The Element at Index "<<i<<" is "<<peek(s,i)<<endl;

}

return 0;

}

**EXPERIMENT NO.-05**

**AIM: Write a program of Operations and Implimentation Queue Data Structure**

**Code:**

#include <iostream>

using namespace std;

class queue

{

public:

int size;

int \*arr;

int b = -1;

};

int isfull(queue \*q)

{

if (q->b == q->size - 1)

{

cout << "\*\*\* Queue Is Full \*\*\*" << endl;

return 1;

}

return 0;

}

int isempty(queue \*q)

{

if (q->b == -1)

{

cout << "\*\*\* Queue Is Empty \*\*\*" << endl;

return 1;

}

return 0;

}

void enqueue(queue \*q, int data)

{

if (isfull(q))

{

cout << "Enqueue Unsucessfull" << endl;

return;

}

q->b++;

q->arr[q->b] = data;

}

int dequeue(queue \*q)

{

if (isempty(q))

{

cout << "Dequeue Unsucessfull" << endl;

return 1;

}

int data = q->arr[0];

for (int i = 0; i < q->b; i++)

{

q->arr[i] = q->arr[i + 1];

}

q->b--;

return data;

}

main()

{

queue \*q = new queue;

q->size = 5;

q->arr = new int[q->size];

enqueue(q, 5);

enqueue(q, 7);

enqueue(q, 6);

cout << "Dequeue Element : " << dequeue(q) << endl;

cout << "Dequeue Element : " << dequeue(q) << endl;

cout << "Dequeue Element : " << dequeue(q) << endl;

enqueue(q, 16);

enqueue(q, 26);

enqueue(q, 36);

cout << "Dequeue Element : " << dequeue(q) << endl;

cout << "Dequeue Element : " << dequeue(q) << endl;

cout << "Dequeue Element : " << dequeue(q) << endl;

isempty(q);

isfull(q);

return 0;

}

**EXPERIMENT NO.-06**

**AIM: Write a program of Operations and Implimentation Circular Queue Data Structure**

**Code:**

#include <iostream>

using namespace std;

class queue

{

public:

int size;

int f = 0;

int b = 0;

int \*arr;

};

int isfull(queue \*q)

{

if (((q->b + 1) % q->size) == q->f)

{

cout << " Queue Is Full" << endl;

return 1;

}

return 0;

}

int isempty(queue \*q)

{

if (q->f == q->b)

{

cout << " Queue Is Empty" << endl;

return 1;

}

return 0;

}

void enqueue(queue \*q, int data)

{

if (isfull(q))

{

return;

}

q->b = (q->b + 1) % q->size;

q->arr[q->b] = data;

}

int dequeue(queue \*q)

{

if (isempty(q))

{

return 0;

}

q->f = (q->f + 1) % q->size;

return q->arr[q->f];

}

int main()

{

queue \*q = new queue;

q->size = 10;

q->arr = new int[q->size];

enqueue(q, 10);

cout << "Dequeue Element : " << dequeue(q) << endl;

enqueue(q, 11);

enqueue(q, 12);

enqueue(q, 13);

enqueue(q, 14);

enqueue(q, 15);

enqueue(q, 16);

enqueue(q, 17);

enqueue(q, 18);

enqueue(q, 28);

enqueue(q, 38);

enqueue(q, 38);

cout << "Dequeue Element : " << dequeue(q) << endl;

cout << "Dequeue Element : " << dequeue(q) << endl;

cout << "Dequeue Element : " << dequeue(q) << endl;

cout << "Dequeue Element : " << dequeue(q) << endl;

cout << "Dequeue Element : " << dequeue(q) << endl;

cout << "Dequeue Element : " << dequeue(q) << endl;

cout << "Dequeue Element : " << dequeue(q) << endl;

cout << "Dequeue Element : " << dequeue(q) << endl;

cout << "Dequeue Element : " << dequeue(q) << endl;

cout << "Dequeue Element : " << dequeue(q) << endl;

cout << "Dequeue Element : " << dequeue(q) << endl;

return 0;

}

**EXPERIMENT NO.-07**

**AIM: Write a program of Operations and Implimentation Dynamic Queue Data Structure**

**Code:**

#include <iostream>

using namespace std;

class queue

{

public:

int data;

queue \*next;

};

void linkedlisttraversal(queue \*q)

{

cout << "printing the elements of linked list" << endl;

while (q != NULL)

{

cout << " Element :" << q->data << endl;

q = q->next;

}

}

int isfull(queue \*q)

{

queue \*ptr = new queue;

if (ptr == NULL)

{

cout << "Queue Is Full" << endl;

return 1;

}

return 0;

}

int isempty(queue \*q)

{

if (q->next == NULL)

{

cout << "Queue Is Empty" << endl;

return 1;

}

return 0;

}

void enqueue(queue \*\*front, queue \*\*back, int data)

{

if (isfull(\*front))

{

return;

}

else

{

queue \*new1 = new queue;

new1->data = data;

new1->next = NULL;

if (\*front == NULL)

{

\*front = \*back = new1;

}

else

{

(\*back)->next = new1;

\*back = new1;

}

}

}

int dequeue(queue \*\*front)

{

if (isempty(\*front))

{

return 0;

}

queue \*ptr = \*front;

int data = (\*front)->data;

\*front = (\*front)->next;

free(ptr);

return data;

}

int main()

{

queue \*front = NULL;

queue \*back = NULL;

enqueue(&front, &back, 45);

enqueue(&front, &back, 47);

enqueue(&front, &back, 48);

enqueue(&front, &back, 48);

enqueue(&front, &back, 48);

enqueue(&front, &back, 48);

enqueue(&front, &back, 48);

linkedlisttraversal(front);

cout<<"dequeue element :"<<dequeue(&front)<<endl;

cout<<"dequeue element :"<<dequeue(&front)<<endl;

cout<<"dequeue element :"<<dequeue(&front)<<endl;

cout<<"dequeue element :"<<dequeue(&front)<<endl;

cout<<"dequeue element :"<<dequeue(&front)<<endl;

cout<<"dequeue element :"<<dequeue(&front)<<endl;

linkedlisttraversal(front);

return 0;

}

**EXPERIMENT NO.-08**

**AIM: Write a program of Operations and Implimentation Dynamic Stack Data Structure**

**Code:**

#include <iostream>

using namespace std;

class node

{

public:

int data;

node \*next;

};

void linkedlisttraversal(node \*top)

{

while (top != NULL)

{

cout<<"Element : "<<top->data<<endl;

top = top->next;

}

cout<<endl;

}

int Isfull(node \*top)

{

node \*ptr = new node;

if (ptr == NULL)

{

return 1;

}

return 0;

}

int Isempty(node \*top)

{

if (top == NULL)

{

return 1;

}

return 0;

}

void push(node \*\*top, int data)

{

if (Isfull(\*top))

{

cout << "Stack OverFlow" << endl;

}

else

{

node \*ptr = new node;

ptr->data = data;

ptr->next = \*top;

\*top = ptr;

}

}

int pop(node \*\*top)

{

if (Isempty(\*top))

{

cout<<"Stack UnderFlow"<<endl;

return 404;

}

else

{

node \*ptr = \*top;

int data = ptr->data;

\*top = ptr->next;

free(ptr);

return data;

}

}

int main()

{

node \*top =NULL;

push(&top,5);

push(&top,15);

push(&top,55);

push(&top,25);

push(&top,35);

push(&top,45);

linkedlisttraversal(top);

cout<<pop(&top)<<endl;

cout<<pop(&top)<<endl;

cout<<pop(&top)<<endl;

linkedlisttraversal(top);

return 0;

}

**EXPERIMENT NO.-09**

**AIM: Write a program To Create A Singly Linked List Data Structure**

**Code:**

#include <bits/stdc++.h>

using namespace std;

#define ll long long int

class node

{

public:

int data;

node \*next;

node(int data)

{

this->data = data;

this->next = NULL;

}

};

void linked\_list\_traversal(node \*ptr)

{

while (ptr != NULL)

{

cout << "Element :" << ptr->data << endl;

ptr = ptr->next;

}

}

void insert(node \*ptr, int data)

{

node \*new1 = new node(data);

new1->next = ptr->next;

ptr->next = new1;

}

void add\_end(node \*ptr, int data)

{

node \*ptr1 = new node(data);

while (ptr->next != NULL)

{

ptr = ptr->next;

}

ptr1->next = ptr->next;

ptr->next = ptr1;

}

void delete\_given(node \*ptr, node \*head)

{

int i = 0;

node \*p = head;

while (p->data != ptr->data)

{

p = p->next;

i++;

}

for (int j = 0; j < i - 1; j++)

{

head = head->next;

}

head->next = ptr->next;

free(p);

}

int main()

{

ios\_base::sync\_with\_stdio(false);

cin.tie(nullptr);

cout.tie(nullptr);

node \*ptr = new node(5);

add\_end(ptr, 6);

add\_end(ptr, 7);

add\_end(ptr, 8);

linked\_list\_traversal(ptr);

delete\_given(ptr->next->next, ptr);

cout << "After Deletion :" << endl;

linked\_list\_traversal(ptr);

return 0;

}

**EXPERIMENT NO.-10**

**AIM: Write a program To Create A Doubly Linked List Data Structure And perform following operations:**

1. **Insertion at a particular position**
2. **Deletion in the beginning**
3. **Traverse the list**

**Code:**

#include <iostream>

using namespace std;

class node

{

public:

int data;

node \*prev;

node \*next;

node(int data)

{

this->data = data;

this->next = NULL;

this->prev = NULL;

}

};

void traversal(node \*head)

{

if (head == NULL)

{

printf("\nList is empty\n");

return;

}

int i = 0;

while (head->next != NULL)

{

i++;

cout << "Element " << i << ": " << head->data << endl;

head = head->next;

if (head->next == NULL)

{

cout << "Element " << i + 1 << ": " << head->data;

}

}

cout << endl;

}

void delete\_front(node \*\*head)

{

node \*ptr = \*head;

\*head = (\*head)->next;

(\*head)->prev = NULL;

free(ptr);

}

void insert\_at\_index(node \*head, int index, int data)

{

node \*new\_node = new node(data);

// new\_node->data = data;

if (head == NULL)

{

head = new\_node;

new\_node->next = NULL;

new\_node->prev = NULL;

}

else if (head->next == NULL)

{

head->next = new\_node;

new\_node->prev = head;

new\_node->next = NULL;

}

else

{

for (int i = 0; i < index - 1; i++)

{

head = head->next;

}

cout << head->data;

new\_node->next = head->next;

head->next->prev = new\_node;

head->next = new\_node;

}

}

int main()

{

node \*head = new node(1);

node \*head1 = new node(2);

node \*head2 = new node(3);

node \*head3 = new node(5);

head->next = head1;

head1->next = head2;

head2->next = head3;

head1->prev = head;

head2->prev = head1;

head3->prev = head2;

insert\_at\_index(head, 3, 6);

delete\_front(&head);

traversal(head);

return 0;

}