**ASSINGNMENT OF AOA**

**Program: CS**

**Semester: 4th**

**Section: A**

**Roll No: 231103**

**Subject: Analysis of Algorithm**

**Submitted By: M Salman khan**

**Submitted To: Sir Dr. Atif**

**TABLE OF CONTENTS**

QNO\_1...............................................................................................3

QNO\_2...................................................................................................7

QNO\_3................................................................................... 9

QNO\_4...................................................................................................14

QNO\_5...................................................................................................16

***Q1: What is the time complexity of search, insert and delete operations in an unsorted array i.e. 5,1,3,4,2,6. Also implement these operations in C++ language.***

**Answer 1 :-**

In an **unsorted array** like [5, 1, 3, 4, 2, 6], the time complexities for basic operations are:

**1. Search**

* **Time Complexity:** **O(n)**
* **Why:** You may have to scan each element in the array to find the target value (worst-case: it's the last element or not present at all).

**2. Insert**

* **Time Complexity:** **O(1)** (on average)
* **Why:** You can insert a new element at the end of the array in constant time **if there’s available space**. If the array is dynamically resized (like in Python lists or Java ArrayList), resizing might take **O(n)** occasionally, but the amortized cost is still **O(1)**.

**3. Delete**

* **Time Complexity:** **O(n)**
* **Why:** To delete an element, you first have to **search** for it (which is O(n)), and then **shift** the elements after it to fill the gap, which can also take **O(n)** time in the worst case.

**Implementation in C++:-**

#include <iostream>

using namespace std;

class DataContainer {

int storage[100];

int count = 0;

public:

// Search Operation (Linear Search)

int find(int key) {

for (int i = 0; i < count; i++) {

if (storage[i] == key) return i;

}

return -1; // Not found

}

// Insert Operation (Add at End)

void add(int val) {

if (count < 100) {

storage[count++] = val;

cout << "Added " << val << " at index " << count - 1 << endl;

} else {

cout << "Storage Full!" << endl;

}

}

// Delete Operation (Shift elements left after deletion)

void erase(int key) {

int index = find(key);

if (index == -1) {

cout << "Element " << key << " not found!" << endl;

return;

}

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

storage[i] = storage[i + 1];

}

count--;

cout << "Deleted " << key << " from index " << index << endl;

}

// print current array

void display() {

cout << "Current array: ";

for (int i = 0; i < count; i++) {

cout << storage[i] << " ";

}

cout << endl;

}

};

// Test in main function

int main() {

DataContainer dc;

**// Initial Elements**

dc.add(5);

dc.add(1);

dc.add(3);

dc.add(4);

dc.add(2);

dc.add(6);

dc.display();

**// Search Example**

int index = dc.find(4);

if (index != -1)

cout << "Element 4 found at index: " << index << endl;

else

cout << "Element 4 not found!" << endl;

**// Insert Example**

dc.add(10);

dc.display();

**// Delete Example**

dc.erase(3);

dc.display();

return 0;

}

***Q2: Determine the time complexity of search, insert and delete operations in sorted array i.e. 1,2,3,4,5,6. Also implement these operations in C++ language.***

In a sorted array, elements are maintained in ascending order. This order enables faster searches but makes insertions and deletions slower:

• Search: O(log n) — Binary search can quickly find elements.

• Insert: O(n) — Elements may need shifting to preserve order after an insertion.

• Delete: O(n) — Post-deletion, shifting is required to close the gap.

## C++ Code (implementation) :

class OrderedList {  
 int list[100];  
 int size = 0;  
  
public:

**// search**  
 int binarySearch(int key) {  
 int low = 0, high = size - 1;  
 while (low <= high) {  
 int mid = low + (high - low) / 2;  
 if (list[mid] == key) return mid;  
 else if (list[mid] < key) low = mid + 1;  
 else high = mid - 1;  
 }  
 return -1;  
 }  
**// insertion**  
 void orderedInsert(int val) {  
 if (size >= 100) {  
 cout << "List Full!\n";  
 return;  
 }  
 int i = size - 1;  
 while (i >= 0 && list[i] > val) {  
 list[i + 1] = list[i];  
 i--;  
 }  
 list[i + 1] = val;  
 size++;  
 }  
**// deletion**  
 void orderedDelete(int key) {  
 int pos = binarySearch(key);  
 if (pos == -1) return;  
 for (int i = pos; i < size - 1; i++)  
 list[i] = list[i + 1];  
 size--;  
 }  
};

**Q3: The following operations are associated with stack. Determine the time complexity of each operation, and also implement these stack operations using C++.**

**Answer 3 :-**

• empty() – Returns whether the stack is empty.

• size() – Returns the size of the stack.

• top() – Returns a reference to the top most element of the stack.

• push(g) – Adds the element ‘g’ at the top of the stack.

• pop() – Deletes the top most element of the stack

**Answer:-**

* **empty() – O(1)**
* **What it does:** Checks if the stack has no elements.
* **Why O(1):** It just checks an internal counter or pointer — no need to go through the elements.
* **size() – O(1)**
* **What it does:** Returns the total number of elements in the stack.
* **Why O(1):** The stack keeps track of its size internally, so it returns the value instantly.
* **top() – O(1)**
* **What it does:** Returns the top (last inserted) element without removing it.
* **Why O(1):** The stack directly points to the top element, so it's accessed immediately.
* **push(g) – O(1)**
* **What it does:** Adds the element g to the top of the stack.
* **Why O(1):** It just places the element at the next available position — no shifting or searching needed.
* **pop() – O(1)**
* **What it does:** Removes the top element from the stack.
* **Why O(1):** It just removes (or moves the top pointer) — again, no need to shift other elements.

**C++ Implementation :-**

#include<iostream>

using namespace std;

**//int maxvalue=5;**

int stack[5];

// top

int top=-1;

// push

void push(int item){

if(top==4){

cout<<"overflow\n";

} else{

stack[++top]=item;

}

}

// pop

void pop(){

if(top==-1){

cout<<"underflow\n";

} else{

int item=stack[top--];

cout<<"this item is deleted = "<<item<<endl;

}

}

// display

void display(){

if(top==-1){

cout<<"stack is empty \n";

}

else{

cout<<"the element of stack ";

for(int i=0;i<=top;i++){

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

}

cout<<endl;

}

}

int main(){

int choice,item;

cout<<"enter your choiceses\n 1:push\n 2:pop\n 3:display \n 4:exit\n";

while(1){

cout<<"enter your choice to perform any operation = ";

cin>>choice;

switch(choice){

**case 1:**

cout<<"enter value to stack = ";

cin>>item;

push(item);

break;

**case 2:**

pop();

break;

**case 3:**

display();

break;

**case 4:**

return 0;

default:

cout<<"sorry invalid character";

}

}

}

***Q4: The following operations are associated with queue. Determine the time complexity of each operation and also implement the queue operations using C++.***

**Answer 4 :-**

• **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

**• Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.

**• Front:** Get the front item from queue.

**• Rear:** Get the last item from queue

**Answer:-**

### Time Complexities of Queue Operations

| **Operation** | **Time Complexity** |
| --- | --- |
| **Enqueue** | **O(1)** (if using circular array or linked list) |
| **Dequeue** | **O(1)** (same as above) |
| **Front** | **O(1)** |
| **Rear** | **O(1)** |

A **Queue** works on **FIFO** – First In, First Out.  
You insert at the **rear**, and remove from the **front**.

## C++ Code(implementation):

class Node {  
public:  
 int data;  
 Node\* next;  
 Node(int val) : data(val), next(nullptr) {}  
};  
  
class Queue {  
 Node \*front, \*rear;  
  
public:  
 Queue() : front(nullptr), rear(nullptr) {}  
**// Enque**  
 void enqueue(int val) {  
 Node\* temp = new Node(val);  
 if (!rear) front = rear = temp;  
 else {  
 rear->next = temp;  
 rear = temp;  
 }  
 }  
**// deque**  
 void dequeue() {  
 if (!front) {  
 cout << "Queue Empty!\n";  
 return;  
 }  
 Node\* temp = front;  
 front = front->next;  
 if (!front) rear = nullptr;  
 delete temp;  
 }  
***// Front***  
 int getFront() {  
 if (front) return front->data;  
 throw runtime\_error("Queue Empty!");  
 }  
};

***Q5: The following operations are associated with Linked List. Determine the time complexity of each operation.***

**Answer 5 :-**

**Linked list operations:**

**insertAtBeginning – O(1)**

* **What it does:** Inserts a new node at the beginning (head) of the linked list.
* **Why O(1):** The head pointer is directly modified, and the new node is linked to the current first node. No traversal needed!

**insertAtEnd** – **O(n)**

* **What it does:** Inserts a new node at the end (tail) of the list.
* **Why O(n):** You must traverse through the entire list to reach the last node. Once you're at the end, the insertion itself is **O(1)**, but traversal makes it **O(n)**.

**insertAtPosition** – **O(n)**

* **What it does:** Inserts a new node at a specific position (e.g., 3rd, 5th, etc.).
* **Why O(n):** You need to traverse through the list to find the desired position. Once you’re there, the insertion itself is **O(1)**, but the traversal is **O(n)**.

**deleteAtBeginning – O(1)**

* **What it does:** Removes the first node (head) of the linked list.
* **Why O(1):** The head pointer is updated to point to the second node. This operation doesn't require traversal.

**deleteAtEnd** – **O(n)**

* **What it does:** Removes the last node (tail) of the list.
* **Why O(n):** You need to traverse the entire list to find the second-to-last node (since it's the only one that can point to nullptr after the last node is deleted). The deletion itself is **O(1)**, but traversal makes it **O(n)**.

**deleteAtPosition** – **O(n)**

* **What it does:** Deletes a node at a specific position (e.g., 3rd, 5th, etc.).
* **Why O(n):** You need to traverse through the list to find the node just before the one you want to delete, so the complexity is **O(n)**. Deleting the node itself is **O(1)** once you reach it.

**search** – **O(n)**

* **What it does:** Searches for an element in the list.
* **Why O(n):** In a singly linked list, you must start from the head and go through each node to find the element. If the element is at the end or not present, it requires traversing the entire list.

**traverse** – **O(n)**

* **What it does:** Goes through all the nodes in the list to either print them or perform some operation.
* **Why O(n):** You have to visit every node once, which is a linear operation.

**getLength** – **O(n)**

* **What it does:** Counts the number of nodes in the linked list.
* **Why O(n):** You need to traverse the entire list to count the nodes.

## C++ Code

class Node {  
public:  
 int value;  
 Node\* next;  
 Node(int val) : value(val), next(nullptr) {}  
};  
  
class LinkedList {  
 Node\* head;  
  
public:  
 LinkedList() : head(nullptr) {}  
**//insert**  
 void insertFront(int val) {  
 Node\* newNode = new Node(val);  
 newNode->next = head;  
 head = newNode;  
 }  
// delete  
 void deleteEnd() {  
 if (!head) return;  
 if (!head->next) {  
 delete head;  
 head = nullptr;  
 return;  
 }  
 Node\* current = head;  
 while (current->next->next)  
 current = current->next;  
 delete current->next;  
 current->next = nullptr;  
 }  
};

***THE END***