### ****Q1. A)** Write a program to insert, search, and delete an element 'x' at position 'p' in an unsorted array of size 'n'. Find the time complexity for each operation. Also, mention the programming language used. Code:** #include <bits/stdc++.h>

using namespace std;

typedef struct arr

{

int \*ar;

int len;

int size;

} arr;

void init(arr \*a, int size)

{

a->ar = new int(size);

a->len = 0;

a->size = size;

}

void insert(arr \*a, int p, int val) {

for (int i = MAX\_SIZE - 1; i > p; i--) {

a->ar[i] = a->ar[i - 1];

}

a->ar[p] = val;

}

void insertatend(arr \*a, int val)

{

if (a->len >= a->size)

return;

a->ar[a->len] = val;

a->len++;

}

void remove(arr \*a, int p)

{

for (int i = p; i < a->len - 1; i++)

{

a->ar[i] = a->ar[i + 1];

}

a->len--;

}

void print(arr a)

{cout<<endl<<" array : ";

for (int i = 0; i < a.len; i++)

{

cout << a.ar[i] << " ";

}

cout << endl <<endl;

}

int search(arr a, int val)

{

for (int i = 0; i < a.len; i++)

{

if (a.ar[i] == val)

return i;

}

return -1;

}

int main()

{

arr a;

init(&a, 100);

int p;

while (1)

{

int choice;

int val;

cout << "1.Insert element in array " << endl

<< "2.Insert at particular Position" << endl

<< "3.Delete element from particular position" << endl

<< "4.search an element" << endl

<< "5.Print the array" << endl

<< "Enter the choice:";

cin >> choice;

switch (choice)

{

case 1:

cout << "enter the value to be entered: ";

cin >> val;

insertatend(&a, val);

break;

case 2:

cout << "enter the value to be entered: ";

cin >> val;

cout << "Enter the position : ";

cin >> p;

insert(&a, p, val);

break;

case 3:

cout << "Enter the position of element to be deleted :" ;

cin >> p;

remove(&a, p);

break;

case 4:

cout << "Enter the element to be searched: ";

cin >> val;

cout<<"the position is: "<< search(a, val)<<endl;

break;

case 5:

print(a);

break;

default:

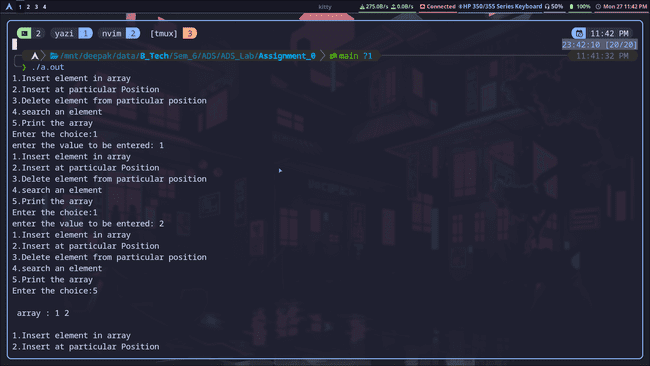
break;

}

}

}

### **OUTPUT:**



### 

#### ****Theoretical Explanation****

1. **Insertion at position 'p':**
   * Insert the element at position 'p' and shift elements to the right.
   * **Time Complexity:**
     + Best Case: O(1) (if inserted at the end).
     + Worst Case: O(n) (if inserted at the beginning).
2. **Search for 'x':**
   * Traverse the array to find 'x'.
   * **Time Complexity:**
     + Best Case: O(1) (if 'x' is the first element).
     + Worst Case: O(n) (if 'x' is not found or at the end).
3. **Deletion at position 'p':**
   * Remove the element at 'p' and shift the remaining elements to the left.
   * **Time Complexity:**
     + Best Case: O(1) (if deleting the last element).
     + Worst Case: O(n) (if deleting the first element).

### ****Q1. B)**** Write a program to insert, search, and delete an element 'x' at position 'p' in a Linked List of size 'n'. Find the time complexity for each operation. Also, mention the programming language used. Code: #include <bits/stdc++.h>

using namespace std;

typedef struct node {

int val;

node \*next;

node(int value) : val(value), next(nullptr) {}

} node;

typedef struct ll {

node \*head;

} ll;

void init(ll \*l) {

l->head = nullptr;

}

void insert(ll \*l, int val) {

if (l->head == nullptr) {

l->head = new node(val);

return;

}

node \*p = l->head;

while (p->next) {

p = p->next;

}

p->next = new node(val);

}

void print(ll l) {

node \*p = l.head;

while (p) {

cout << p->val << " ";

p = p->next;

}

cout << endl;

}

void insert(ll \*l, int pos, int val) {

if (pos == 0) {

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

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

l->head = new\_node;

return;

}

node \*p = l->head;

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

if (p == nullptr) return;

p = p->next;

}

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

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

p->next = new\_node;

}

void del(ll \*l, int pos) {

node \*p = l->head;

if (pos == 0) {

l->head = p->next;

delete p;

return;

}

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

p = p->next;

}

node \*q = p->next;

p->next = q->next;

delete q;

}

int search(ll l, int val) {

node \*p = l.head;

int i = 0;

while (p) {

if (p->val == val) {

return i;

}

i++;

p = p->next;

}

return -1;

}

int main() {

ll l;

init(&l);

int choice;

do {

cout << "\n--- Menu ---\n";

cout << "1. Insert at end\n";

cout << "2. Insert at position\n";

cout << "3. Delete at position\n";

cout << "4. Search value\n";

cout << "5. Print list\n";

cout << "6. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1: {

int value;

cout << "Enter value to insert: ";

cin >> value;

insert(&l, value);

break;

}

case 2: {

int pos, value;

cout << "Enter position and value to insert: ";

cin >> pos >> value;

insert(&l, pos, value);

break;

}

case 3: {

int pos;

cout << "Enter position to delete: ";

cin >> pos;

del(&l, pos);

break;

}

case 4: {

int value;

cout << "Enter value to search: ";

cin >> value;

int pos = search(l, value);

if (pos != -1) {

cout << "Value found at position: " << pos << endl;

} else {

cout << "Value not found in the list.\n";

}

break;

}

case 5: {

cout << "List contents: ";

print(l);

break;

}

case 6: {

cout << "Exiting program.\n";

break;

}

default: {

cout << "Invalid choice. Please try again.\n";

break;

}

}

} while (choice != 6);

return 0;

}

**OUTPUT:**

****

#### 

#### 

#### ****Theoretical Explanation****

1. **Insertion at position 'p':**
   * Traverse to position 'p' and insert the element.
   * **Time Complexity:** O(n) (as traversal is needed).
2. **Search for 'x':**
   * Traverse the linked list to find 'x'.
   * **Time Complexity:** O(n).
3. **Deletion at position 'p':**
   * Traverse to position 'p' and delete the element.
   * **Time Complexity:** O(n).

### ****Q1. C)**** Comparison between Array and Linked List

|  |  |  |
| --- | --- | --- |
| Feature | Array | Linked List |
| Memory Allocation | Contiguous | Dynamic |
| Insertion | O(n) (shift needed) | O(1) (at head or tail) |
| Deletion | O(n) (shift needed) | O(1) (if head is known) |
| Access Time | O(1) (direct indexing) | O(n) (sequential access) |
| Flexibility | Fixed size | Dynamic size |

### ****Q2)**** Given a number 'A', find the sum of its digits using recursion.

**CODE:**

#include <bits/stdc++.h>

using namespace std;

int sum(int n){

if(n==0) return 0;

return n%10 + sum(n/10);

}

int main(){

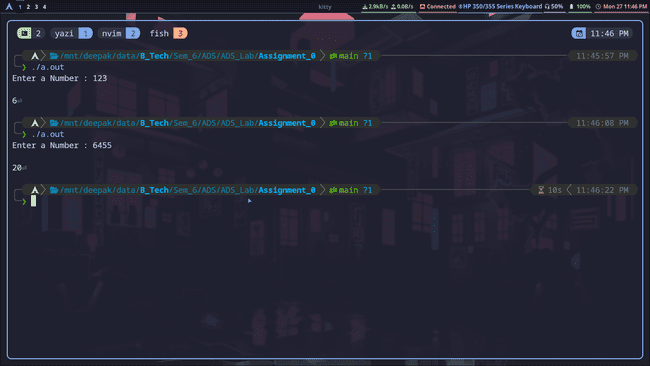
int num;

cout<<"Enter a Number : ";

cin>>num;

cout<<"The sum of digits is "<<sum(num);

}



#### Example:

* Input: A = 46
* Output: 10 (Explanation: 4 + 6 = 10)

#### ****Time Complexity:****

* **Best and Worst Case:** O(d), where 'd' is the number of digits in 'A'.

### ****Q3)**** Given a sorted array of distinct integers containing numbers from 0 to n, find the smallest missing number using binary search.

**CODE:**#include <bits/stdc++.h>

using namespace std;

int binarySearch(vector<int> arr) {

int low = 0;

int high = arr.size() - 1;

while (low <= high) {

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

if (arr[mid] != mid) {

high = mid - 1;

} else {

low = mid + 1;

}

}

return low;

}

int main() {

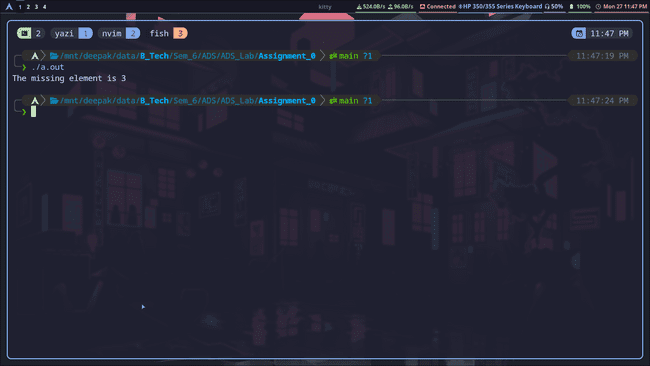
vector<int> arr = {0, 1, 2, 4, 5, 6};

cout << "The missing element is " << binarySearch(arr) << endl;

return 0;

}

**OUTPUT:**



#### Example:

* Input: Array = [0, 1, 2, 3, 5, 6]
* Output: 4 (Explanation: 4 is the smallest missing number).

#### ****Time Complexity:****

* **Best and Worst Case:** O(log n) (binary search divides the array into halves).

### ****Q4)**** The Fibonacci numbers are the numbers in the following integer sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

#### Problem:

Given a number 'A', find and return the A-th Fibonacci number using recursion.

**CODE:**

#include <bits/stdc++.h>

using namespace std;

int fibonacci(int num){

if (num == 0) return 0;

if (num == 1) return 1;

return fibonacci(num - 1) + fibonacci(num - 2);

}

int main(){

int num;

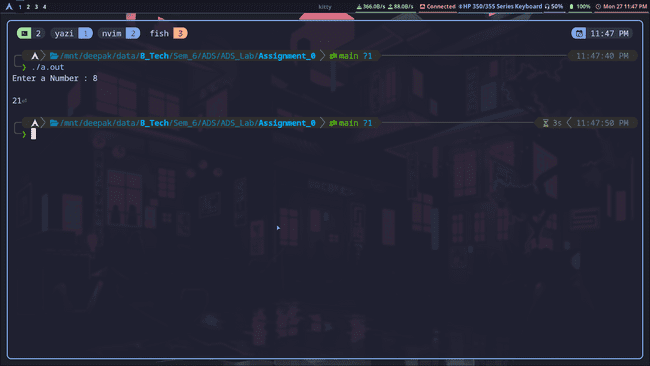
cout<<"Enter the num : ";

cin>>num;

cout<<"The num is "<<fibonacci(num);

}

**OUTPUT:**

****

#### Example:

* Input: A = 5
* Output: 5 (Explanation: The sequence is 0, 1, 1, 2, 3, 5, and the 5th Fibonacci number is 5).

#### ****Time Complexity:****

* **Best and Worst Case:** O(2^n) (due to repeated calculations in recursion).