**Binary Search, Linear Search, Fibonacci Series, Hashing, Tower of Hanoi**

**Objectives:**

1. To understand and implement Binary Search and Linear Search
2. To understand and implement Fibonacci Series
3. To understand and implement Hashing
4. To understand and implement Tower of Hanoi

**Theory:**

**Binary Search**

Binary Search is an efficient algorithm for finding a target value within a sorted array. It works by repeatedly dividing the search interval in half. If the target value is less than the value in the middle of the interval, the search continues in the left half. If the target is greater, the search continues in the right half. This process repeats until the target is found or the search interval is empty. Time complexity: O(log n), where n is the number of elements in the array.

**Linear Search**

Linear Search is a straightforward algorithm used to find a target value in an unsorted array. It works by checking each element in the array one by one, starting from the first element, until the target is found or the entire array is traversed. This means that even if the array is not sorted, the algorithm will still check every element. While simple to implement, it is less efficient compared to other search algorithms, especially with large datasets. The algorithm performs well when dealing with small arrays or when the target value is near the beginning of the array. Time complexity: O(n), where n is the number of elements in the array.

**Fibonacci Series**

The Fibonacci Series is a sequence of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1. The sequence begins as: 0, 1, 1, 2, 3, 5, 8, 13, and so on. Mathematically, it can be expressed as F(n) = F(n-1) + F(n-2), with the first two numbers defined as F(0) = 0 and F(1) = 1. The Fibonacci sequence appears frequently in various areas of mathematics, nature, computer science, and art. It can be generated using a simple iterative or recursive approach, but the recursive method can be inefficient for larger values due to repeated calculations. Time complexity: O(2^n).

**Hashing**

Hashing is a technique used to efficiently store and retrieve data using a hash function. A hash function maps a given input (or "key") to a fixed-size value, typically a unique index in a hash table. This allows for quick access to data, as the value can be retrieved directly by computing the hash of the key. Hashing is commonly used in implementing data structures like hash maps or hash sets, where average time complexity for operations such as insertion, deletion, and searching is O(1). However, collisions can occur when two keys hash to the same index, and techniques like chaining or open addressing are used to resolve these collisions. Time complexity (average case): O(1) for insert, delete, and search operations. However, in the worst case (e.g., many collisions), it can degrade to O(n).

**Tower of Hanoi**

The Tower of Hanoi is a classic puzzle that involves moving a set of disks from one peg to another, following a set of rules. The puzzle consists of three pegs and a number of disks of different sizes. The objective is to move all the disks from the source peg to the destination peg, while following these rules: only one disk can be moved at a time, a disk can only be placed on top of a larger disk, and no disk can be placed on top of a smaller disk. The minimum number of moves required to solve the puzzle with n disks is 2^n - 1. This makes the puzzle an example of exponential time complexity.The Tower of Hanoi is often used to demonstrate recursion, as the solution involves breaking down the problem into smaller subproblems, each involving moving a subset of the disks. Although the algorithm is simple, the number of moves grows exponentially with the number of disks, making it impractical for large values of n. Time complexity: O(2^n), where n is the number of disks.

**Algorithm:**

**Binary Search**

1. **Main Function:**
   1. Initialize a vector list with a set of integers.
   2. Call the sort() function to sort the list in ascending order.
   3. Call the binarySearch() function to find the index of the target value (in this case, 322) in the sorted list.
   4. If the target value is found, print the position and the number at that position.
   5. If the target value is not found, print a message saying the number is not in the list.
2. **Sort Function:**
   1. For each element i from 0 to the second-last element:
      * Compare it with every element j after it.
      * If list[i] is greater than list[j], swap the elements.
   2. Repeat this process until the list is sorted.
3. **Binary Search Function:**
   1. Start with two pointers: low (start of the list) and high (end of the list).
   2. Calculate the midpoint midpoint as the average of low and high.
   3. If the element at the midpoint is equal to the target value, return midpoint (position of the target).
   4. If the element at the midpoint is less than the target, search the right half by calling binarySearch() with updated bounds.
   5. If the element at the midpoint is greater than the target, search the left half by calling binarySearch() with updated bounds.
   6. If the low index exceeds high, return -1, indicating that the target is not present in the list.

**Linear Search**

1. **Main Function:**
   1. Initialize a vector list with a set of integers.
   2. Call the linearSearch() function to find the index of the target value (in this case, 322) in the list.
   3. If the target value is found, print the position and the number at that position.
   4. If the target value is not found, print a message saying the number is not in the list.
2. **Linear Search Function:**
   1. Iterate through each element of the vector list:
      * For each element, compare it to the target value.
      * If the element matches the target, return the index where the target is found.
   2. If the target value is not found after iterating through the entire list, return -1 to indicate that the target is not present.

**Fibonacci Series**

1. **Main Function:**
   1. Set the value n to 7 (or any desired value for the length of the Fibonacci sequence).
   2. Call the fibonacci() function to generate the first n Fibonacci numbers.
   3. Print the Fibonacci series by iterating over the returned vector series.
2. **Fibonacci Function:**
   1. The fibonacci() function initializes an empty vector series to store the Fibonacci numbers.
   2. It then calls recursiveFibonacci() to populate the series with the first n Fibonacci numbers.
   3. Once the Fibonacci series is generated, return the series.
3. **Recursive Fibonacci Function:**
   1. The function recursiveFibonacci() takes two parameters: n (the desired number of Fibonacci numbers) and series (the vector storing the Fibonacci sequence).
   2. Base case: If the series has n elements, the recursion stops (i.e., the Fibonacci sequence has been fully generated).
   3. The Fibonacci sequence starts with the numbers 0 and 1. If series has fewer than two elements, it appends 0 and 1.
   4. If series already has at least two elements, it calculates the next Fibonacci number by adding the last two elements of series, and then it appends this number to series.
   5. The function then calls itself recursively to continue building the series until the required n Fibonacci numbers are generated

**Hashing**

1. **HashTable Class:**
   1. The HashTable class implements a hash table with separate chaining to handle collisions using linked lists (in this case, vectors).
   2. It has an array of buckets, each represented as a vector of KeyValue pairs.
   3. **KeyValue struct**: Stores a key (string) and its corresponding value (int).
   4. **Insert**: Adds a key-value pair to the hash table. If the key already exists in the same bucket, it is added at the end of the list (separate chaining).
   5. **Search**: Finds a key in the hash table and prints the associated value if found.
   6. **Remove**: Removes a key-value pair from the hash table if the key is found.
2. **Hash Function:**
   1. The hashFunction takes a string key, sums the ASCII values of each character in the string, and then computes the modulus with the table\_size to get an index in the range of the hash table array.
3. **Insert Function:**
   1. Takes a key and value, computes the hash using the hash function, and then adds the key-value pair to the corresponding bucket (vector) in the hash table.
4. **Search Function:**
   1. Takes a key, calculates the hash index, and searches the corresponding bucket (vector) for the key. If found, it prints the key and its associated value; otherwise, it prints that the key was not found.
5. **Remove Function:**
   1. Takes a key, calculates the hash index, searches for the key in the corresponding bucket (vector), and removes it if found. If not found, it prints an appropriate message.

**Tower of Hanoi**

1. **Input**:
   1. n: Number of disks.
   2. fromRod: The rod where the disks are initially placed.
   3. toRod: The rod where the disks need to be moved.
   4. auxRod: The auxiliary rod used to help in the transfer.
2. **Base Case**:
   1. If n == 0, return without doing anything (i.e., there's no disk to move).
3. **Recursive Case**:
   1. If n > 0, the process can be broken down into three steps:
      * **Move n-1 disks** from the fromRod to the auxRod using the toRod as an auxiliary rod. This step is recursive and calls the towerOfHanoi(n - 1, fromRod, auxRod, toRod) function.
      * **Move the nth disk** (the largest disk) from the fromRod to the toRod. This step involves printing a message indicating the move.
      * **Move n-1 disks** from the auxRod to the toRod using the fromRod as an auxiliary rod. This step is recursive and calls the towerOfHanoi(n - 1, auxRod, toRod, fromRod) function.
4. **Output**:
   1. For each recursive call, print the disk number being moved from one rod to another in the format: Move disk <n> from rod <fromRod> to rod <toRod>.
5. **Termination**:
   1. The recursion will eventually reduce n to 0, at which point the function terminates.

**Questions:**

**Binary Search**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | *#include <iostream>*  *#include <vector>*  **using** **namespace** std;  **int** binarySearch(**int** low, **int** high, vector<**int**> &list, **int** target){  **if** (low > high){  **return** -1;  }  **int** midpoint = (low + high) / 2;  **if** (list[midpoint] == target){  **return** midpoint;  }  **else** **if** (list[midpoint] < target){  **return** binarySearch(midpoint + 1, high, list, target);  }  **else**{  **return** binarySearch(low, midpoint - 1, list, target);  }  }  **void** sort(vector<**int**> &list) {  **for** (**int** i = 0; i < list.size(); i++){  **for** (**int** j = i + 1; j < list.size(); j++){  **if** (list[i] > list[j]){  **int** temp = list[i];  list[i] = list[j];  list[j] = temp;  }  }  }  }  **int** main(){  vector<**int**> list = {1, 2, 3, 54, 32, 6, 4, 234, 2, 423, 22, 34, 322};  sort(list);  **int** position = binarySearch(0, list.size() - 1, list, 322);  **if** (position != -1){  cout << *"The number is at position "* << position << endl;  cout << *"The number at the position is "* << list[position] << endl;  }  **else**{  cout << *"The number is not in the list"* << endl;  }  **return** 0;  } |

Output:

The number is at position 11

The number at the position is 322

**Linear Search**

*#include <iostream>*

*#include <vector>*

**using** **namespace** std;

**int** linearSearch(vector<**int**> &list, **int** target) {

**for** (**int** i = 0; i < list.size(); i++){

**if** (list[i] == target) {

**return** i;

}

}

**return** -1;

}

**int** main(){

vector<**int**> list = {1, 2, 3, 54, 32, 6, 4, 234, 2, 423, 22, 34, 322};

**int** position = linearSearch(list, 322);

**if** (position != -1) {

cout << *"The number is at position "* << position << endl;

cout << *"The number at the position is "* << list[position] << endl;

}

**else**{

cout << *"The number is not in the list"* << endl;

}

**return** 0;

}

Output:

The number is at position 12

The number at the position is 322

**Fibonacci Series**

*#include <iostream>*

*#include <vector>*

**using** **namespace** std;

**void** recursiveFibonacci(**int** n, vector<**int**> &series){

**if** (series.size() == n) {

**return**;

}

**if** (series.size() < 2) {

series.push\_back(series.size());

}

**else**{

series.push\_back(series[series.size() - 1] + series[series.size() - 2]);

}

recursiveFibonacci(n, series);

}

vector<**int**> fibonacci(**int** n) {

vector<**int**> series;

recursiveFibonacci(n, series);

**return** series;

}

**int** main(){

**int** n = 7;

cout << *"Fibonacci series: "*;

vector<**int**> series = fibonacci(n);

**for** (**int** i = 0; i < series.size(); i++){

cout << series[i] << *" "*;

}

cout << endl;

**return** 0;

}

Output:

Fibonacci series: 0 1 1 2 3 5 8

**Hashing**

*#include <iostream>*

*#include <vector>*

*#include <string>*

**using** **namespace** std;

**class** **HashTable**{

private:

**static** **const** **int** table\_size = 10;

**struct** KeyValue{

string key;

**int** value;

};

vector<KeyValue> table[table\_size];

**int** hashFunction(**const** string &key) {

**int** hash = 0;

**for** (**size\_t** i = 0; i < key.length(); ++i) {

hash += key[i];

}

**return** hash % table\_size;

}

public:

**void** insert(**const** string &key, **int** value) {

**int** index = hashFunction(key);

KeyValue kv = {key, value};

table[index].push\_back(kv);

}

**bool** search(**const** string &key) {

**int** index = hashFunction(key);

**for** (**int** i = 0; i < table[index].size(); i++){

**if** (table[index][i].key == key) {

cout << *"Found key '"* << key <<*"' with value: "*<< table[index][i].value << endl;

**return** true;

}

}

cout << *"Key '"* << key << *"' not found."* << endl;

**return** false;

}

**void** remove(**const** string &key) {

**int** index = hashFunction(key);

vector<KeyValue> &bucket = table[index];

**for** (**size\_t** i = 0; i < bucket.size(); ++i) {

**if** (bucket[i].key == key) {

bucket.erase(bucket.begin() + i);

cout << *"Removed key '"* << key << *"' from the table."* << endl;

**return**;

}

}

cout << *"Key '"* << key << *"' not found to remove."* << endl;

}

};

**int** main(){

HashTable ht;

ht.insert(*"a"*, 5);

ht.insert(*"b"*, 34);

ht.insert(*"o"*, 7);

ht.search(*"b"*);

ht.remove(*"a"*);

ht.search(*"a"*);

**return** 0;

}

Output:

Found key 'b' with value: 34

Removed key 'a' from the table.

Key 'a' not found.

**Tower of Hanoi**

*#include <iostream>*

**using** **namespace** std;

**void** towerOfHanoi(**int** n, **char** fromRod, **char** toRod, **char** auxRod){

**if** (n == 0) {

**return**;

}

towerOfHanoi(n - 1, fromRod, auxRod, toRod);

cout << *"Move disk "* << n << *" from rod "* << fromRod<< *" to rod "* << toRod << endl;

towerOfHanoi(n - 1, auxRod, toRod, fromRod);

}

**int** main(){

**int** n = 3;

towerOfHanoi(n, *'A'*, *'C'*, *'B'*);

**return** 0;

}

Output:

Move disk 1 from rod A to rod C

Move disk 2 from rod A to rod B

Move disk 1 from rod C to rod B

Move disk 3 from rod A to rod C

Move disk 1 from rod B to rod A

Move disk 2 from rod B to rod C

Move disk 1 from rod A to rod C

**DISCUSSION AND CONCLUSION**

In this lab work, we understood the concept of Binary Search, Linear Search, Fibonacci Series, Hashing and Tower of Hanoi. We first studied the theory behind all these algorithms and designed algorithms for implementing in the program code. We then executed the code for each of the program and observed output for various input cases and situations. We noted down all the outputs from the program codes and included them in the report.