Honors Project
Efficiency Of Different Search Methods In C++

By: Alon Gottlieb Chernov

March 22, 2023

1 Abstract

Binary search, hashing, and linear search are all searches in the C++ language, and the purpose of searching in programming languages is in order to find an element or value in an array or in a list. As we explore deeper into each search we will learn which ones are more effective at finding the element over other searches. If we did not have search algorithms then then it would take a long to find the piece of data that a user would be searching for. In this paper, I will explore binary search, different types of hashing, linear search, and separate chaining. All searches have varying complexity and some are better than others in efficiency and memory used. I will also explore the use and application of a binary search tree over a regular binary tree.

Keywords: C++, Binary Search, Linear Search, Hashing

2 Introduction

Before we get into the background of how different searches are implemented in C++, we must know the background of C++. C++ is a 35-year-old programming language that is currently undergoing a revival, according to Tiobe Software, which says it is the fastest-growing language of any right now. C++'s heyday was in 2003 when it took up 17.53 percent share of the programming market. However, since then, C++ popularity has gone down substantially. C++'s popularity hit is most likely due to Python and Java Script having full run time capabilities.

3 Template Overview

The template that is used in this paper is the ACM writing format. The paper uses citations in the APA format.

4 Authors and Affiliations

Alon Gottlieb Chernov
aloncher.got@gmail.com
Faramarz Mortezaie
fmortezaie@laspositascollege.edu

5 Linear Search

This method is also known as sequential search or serial search.The list is searched sequentially which means that it starts at the first element and works its way up until the element is found. It is one of the more simpler search algorithms.

C++ Linear Search Example:

```
template <class T>
void Linearsearch(T *a, T item, int n)
{
    int z = 0;//declare int
    for(int i = 0; i < n; i++)
    {
        if(a[i] == item)
        {
            cout << "Item found at position = " << i+1 << endl;//
    declare position item is found
            z = 1 ;
            return;//return when if statement is done
        }//if
    }//for
}//Linearsearch

int main(int argc, const char * argv[]) {
    int arrayInt[10] = {2,42,56,86,87,99,323,546,767,886}; //
    Populate template array with ints
    double arrayDouble[6] = {2.4, 5.53,44.4, 54.45, 65.7,89.54};//
    Populate template array with doubles
    int userInt;//int for user to insert value
    double userDouble;//double for user to insert value

    cout << "\n Elements of Integer Array \n";

    for(int i = 0;i < 10;i++)
    {
        cout<< arrayInt[i] << " " << endl;
    }

    cout << "Enter an item to be search: \n";
    cin >> userInt;

    cout << "Linear Search Method\n";
    Linearsearch(arrayInt,userInt,10);//calling Linearsearch
    function

    cout<<"Elements of double Array \n";
    for(int i=0;i<6;i++)
    {
        cout << arrayDouble[i] << " " << endl;
    }
    cout << "Enter an item to be search: \n";
    cin >> userDouble;

    cout << "Linear Search Method\n";
    Linearsearch(arrayDouble,userDouble,6); // calling Linearsearch
     function
```

```cpp


    cout << "Elapsed time in nanoseconds: "
         << chrono::duration_cast<chrono::nanoseconds>(end - start)
    .count()
         << " ns" << endl;

    cout << "Elapsed time in milliseconds: "
         << chrono::duration_cast<chrono::milliseconds>(end - start
    ).count()
         << " ms" << endl;

    cout << "Elapsed time in seconds: "
         << chrono::duration_cast<chrono::seconds>(end - start).
    count()
         << " sec" << endl;

    return 0;
}
```

Advantages: A linear search can search a value in small to medium arrays relatively quickly. A big advantage of a linear search is that the list does not need to sorted compared to binary search. Linear search is one of the easiest searches to understand for an entry level programmer.

Disadvantages: A big disadvantage of linear search is it's efficiency with dealing with large arrays. Like stated before a linear search would take an average of 500,000 comparisons, whereas binary search would take 20. If the element does not exist in the list a linear search would search until the last element which is highly inefficient.

Time test: To test the time that the program took I used chrono file from the library. I measured how long the program took in nanoseconds, miliseconds, and finally seconds. In the program I tested how long it took for the linear search method to find the last element in an int array, and a double array. The results are as follows, Elapsed time in nanoseconds:8286519817 ns, Elapsed time in milliseconds: 8286 ms, Elapsed time in seconds: 8 sec.

Applications: An example of how a linear search can be used in a non-programming way is finding your Uber car. Pretend there are 100 cars in a parking, and you are trying to find the car of your driver. Using linear search you would go car by car to find your driver, and the best case if that you find your driver in the first couple of cars or the worst case which would be the very last car. If there were only 10 cars this would be quick and efficient, but in the instance of 100 cars or 1000 cars this would be slow and inefficient. This once again shows how using linear search is efficient only for data structures with small values, but using data structures with a lot of values would be slow and inefficient.

6 Binary Search

Binary search is an efficient algorithm for finding an item from a sorted list of items. It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one. This is also known as a divide and conquer method. It is also known as logarithmic search or half-interval search.

C++ Binary Search Example:

```cpp
// Binary Search Function
template <class T>
T BinarySearch(T arr[],T n, int x)
{
    int start = 0; //starting value
    int end = n-1; // ending value
    while(start<=end)//while loop from start fo finsih
    {
        int mid = (start+end)/2;// mid point is declared, and start+end is divided by 2
        if(arr[mid] == x){ // if value is found return the mid point
            return mid;
        }
        else if(arr[mid]<x){ // if the value is lower than the higher point add one
            start = mid + 1;
        }
        else{ // otherwise subrtract one
            end = mid - 1; //
        }
    }
    return -1;//for a value not present
}
int main(int argc, const char * argv[]) {
    auto start = chrono::steady_clock::now();

    int arrayInt[10] = {2,42,56,86,87,99,323,546,767,886}; // Populate template array with ints
    double arrayDouble[6] = {2.4, 5.53,44.4, 54.45, 65.7,89.54};// Populate template array with doubles
    int userInt;//int for user to insert value
    double userDouble;//double for user to insert value
    int num;

    cout << "\n Elements of Integer Array \n";

    for(int i = 0; i < 10; i++)
    {
        cout<< arrayInt[i] << " " << endl;
    }

    cout << "Enter an item to be search: \n";
    cin >> userInt;

    cout << "Binary Search Method\n";
    int index1 = BinarySearch(arrayInt,userInt, num);//calling
    BinarySearch function
    if(index1 == -1){
        cout << "Value is not present\n";
    }
    else{
        cout << userInt << " is present at index " << index1 <<
    endl;
    }


    cout<<"Elements of double Array \n";
    for(int i = 0; i < 6 ;i++)
    {
        cout << arrayDouble[i] << " " << endl;
    }
    cout << "Enter an item to be search: \n";
    cin >> userDouble;

    cout << "Binary Search Method\n";
    int index2 = BinarySearch(arrayDouble, userDouble, num);

    if(index2 == -1){
        cout << "Value is not present\n";
    }
    else{
        cout << userDouble << " is present at index " << index2 <<
    endl;
    }

    auto end = chrono::steady_clock::now();

    cout << "Elapsed time in nanoseconds: "
         << chrono::duration_cast<chrono::nanoseconds>(end - start)
    .count()
         << " ns" << endl;

    cout << "Elapsed time in milliseconds: "
         << chrono::duration_cast<chrono::milliseconds>(end - start
    ).count()
         << " ms" << endl;

    cout << "Elapsed time in seconds: "
         << chrono::duration_cast<chrono::seconds>(end - start).
    count()
         << " sec" << endl;

    return 0;
}
```

Advantages: Compared to linear search, binary search is much faster. For a million elements, linear search would take an average of 500,000 comparisons, whereas binary search would take 20. A binary search is more efficient than large data. It is also a fairly easy algorithm that a programmer can implement in their code.

Disadvantages: A big disadvantage of binary search is that you can only use a binary search if you have a sorted array. Binary search does not work on a unsorted array. A binary search is not useful for a very small number of elements whereas a linear search is.

Time test: To test the time that the program took I used chrono file from the library. I measured how long the program took in nanoseconds, miliseconds, and finally seconds. In the program I tested how long it took for the linear search method to find the last element in an int array, and a double array. The results are as follows, Elapsed time in nanoseconds: 4844517837 ns, Elapsed time in milliseconds: 4844 ms, Elapsed time in seconds: 4 sec. This program took less time than the linear search program further proving the point that binary search is more efficient over linear search.

Applications: There are many more applications to binary search as compared to linear search. An example for an application of binary search is using a dictionary that contains many words in it. If we needed to find the word zebra it would be extremely time consuming flipping through every word trying to find zebra which we would have to do with linear search. With binary search we can split the dictionary in two and focus on the right hand side. We would continue dividing the dictionary in two parts until we have found the word. This quote unquote "divide and conquer" method is why binary search is efficient.

7 Hashing with Quadratic Probing

Quadratic Probing is a technique that is used to avoid collisions in hash tables if the result has already been generated. Quadratic probing is a type of hash function, and a hash function is "the process of transforming any given key or a string of characters into another value" (Zola, 2021).

C++ Quadratic Probing Example

```
using namespace std;

// Function to print an array
void printArray(int arr[], int n)
{
    // Printing the array
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
}

// Quadratic probing
void QuadraticProbing(int table[], int tsize,
             int arr[], int N)
{

    for (int i = 0; i < N; i++)
    {
        // Finding the hash value
        int hv = arr[i] % tsize;

        if (table[hv] == -1){
            table[hv] = arr[i];
        }
        else
        {
            for (int j = 0; j < tsize; j++)
            {
                int t = (hv + j * j) % tsize;
                if (table[t] == -1)
                {
                    table[t] = arr[i];
                    break;
                }
            }
        }
    }
    printArray(table, N);
}

// Driver code
int main()
{

    auto start = chrono::steady_clock::now();
    std::chrono::time_point<std::chrono::high_resolution_clock> end
    ;


    // Declare values in the array
    int arr[] = {50, 700, 76, 85, 92, 73, 101};
    int N = 7;

    int L = 7;
    int hash_table[7];

    // Initializing the hash table
    for (int i = 0; i < L; i++)
    {
        hash_table[i] = -1;
    }

    // Quadratic probing
    QuadraticProbing(hash_table, L, arr, N);

    cout << "Elapsed time in nanoseconds: "
            << chrono::duration_cast<chrono::nanoseconds>(start -
    end).count()
            << " ns" << endl;

    cout << "Elapsed time in milliseconds: "
            << chrono::duration_cast<chrono::milliseconds>(start -
    end).count()
            << " ms" << endl;

    cout << "Elapsed time in seconds: "
            << chrono::duration_cast<chrono::seconds>(start - end).
    count()
            << " sec";
    return 0;
```

Advantages: Quadratic probing works the best in a closed hash table which means that we use only one array for everything and store collisions in said array. Quadratic problem also offers faster inserts, and reduced amount of storage needed.

Disadvantages: An influential disadvantage for quadratic probing is second clustering which essentially is when two keys have the same probe sequence. A probe sequence is maintaining a collection of key–value pairs, and trying to find the value associated with said key.

Time test: To test the time that the program took I used chrono file from the library. I measured how long the program took in nanoseconds, miliseconds, and finally seconds. In the program I tested how long it took for the quadratic probing to insert integer values in my array, and print them out. The results are as follows, Elapsed time in nanoseconds: 521972068 ns, Elapsed time in milliseconds: 5219 ms, Elapsed time in seconds: 5 sec.

Applications: Since quadratic probing essentially is a hashing function it carries most of the benefits and applications of one. An example of an application for quadratic probing is what is above in the code for this section which essentially is creating hash tables which inside have key-value pairs. Each key has a unique value that is different from other keys.

8 Hashing with Linear Probing

Linear probing is a simple and fast implementation for creating a hash table. Linear probing uses open addressing which is allows a value in the array to spill over into other positions in the array. Linear probing was invented in 1954, and was analyzed for time complexity by Donald E. Knuth in 1962. It is remarkable that linear probing is still used today.

C++ Implementation of Linear Probing

```
#include <iostream>
#include <chrono>
using namespace std;

// template for generic type
template <class Key, class Value>

// Hashnode class
class HashNode {
public:
    Value value;
    Key key;
    // Constructor of hashnode
    HashNode(Key key, Value value)
    {
        this->value = value;
        this->key = key;
    }
};

// template for generic type
```

```cpp
template <class Key, class Value>

class HashMap {

    // Hash element array
    HashNode<Key, Value>** arr;

    //Capacity of int
    int capacity;

    // current size
    int size;
public:
    HashMap()
    {
        // Initial capacity of hash array
        capacity = 20;
        size = 0;
        arr = new HashNode<Key, Value>*[capacity];

        // Initialise all elements of array as NULL
        for (int i = 0; i < capacity; i++)
            arr[i] = nullptr;
    }

    //Function to find index
    int hashCode(Key key)
       {
           return key % capacity;
       }

    // Function to add key value pair
    void insertNode(Key key, Value value)
    {
        HashNode<Key, Value>* temp = new HashNode<Key, Value>(key,
    value);

        // Apply hash function to find index for given key
        int hashIndex = hashCode(key);

        // Find next free space to put value in
        while (arr[hashIndex] != nullptr && arr[hashIndex]->key !=
    key && arr[hashIndex]->key != -1) {
            hashIndex++;
            hashIndex %= capacity;
        }

        if (arr[hashIndex] == nullptr || arr[hashIndex]->key == -1)
    {
            size++;
            arr[hashIndex] = temp;
        }
    }

    // Function to display the stored key value pairs
    void display()
    {
        for (int i = 0; i < capacity; i++) {
            if (arr[i] != NULL && arr[i]->key != -1)
                cout <<" value = " << arr[i]->value << endl;
        }
    }
};

// Driver method to test map class
int main()
{
    // Clock timer
    auto start = chrono::steady_clock::now();
    std::chrono::time_point<std::chrono::high_resolution_clock> end
    ;

    // Declare HashMap
    HashMap<int, int>* h = new HashMap<int, int>;

    //Insert values into keys
    h->insertNode(0, 50);
    h->insertNode(1, 700);
    h->insertNode(3, 76);
    h->insertNode(4, 85);
    h->insertNode(5, 92);
    h->insertNode(6, 73);
    h->insertNode(7, 101);

    //Display hash table
    h->display();

    // Time implementation
      cout << "Elapsed time in nanoseconds: "
                 << chrono::duration_cast<chrono::nanoseconds>(
    start - end).count()
                 << " ns" << endl;

      cout << "Elapsed time in milliseconds: "
                 << chrono::duration_cast<chrono::milliseconds>(
    start - end).count()
                 << " ms" << endl;

      cout << "Elapsed time in seconds: "
                 << chrono::duration_cast<chrono::seconds>(start -
     end).count()
                 << " sec";

    return 0;
}
```

Advantages: Some benefits of linear probing is that it offers low memory use because all it needs to be implemented is an array and a hash function. When collisions occur in linear probing the next location for the value to "spill over" are locations adjacent to the value.

Disadvantages: An influential disadvantage for linear probing is primary clustering. Primary clustering is when the number of collisions grow as a function of the number of existing collisions. This affects the efficiency of linear probing.

Time test: To test the time that the program took I used chrono file from the library. I measured how long the program took in nanoseconds, miliseconds, and finally seconds. In the program I tested how long it took for the quadratic probing to insert integer values in my array, and print them out. The results are as follows, Elapsed time in nanoseconds: 422921068 ns, Elapsed time in milliseconds: 4229 ms, Elapsed time in seconds: 4 sec. This is slighty faster than quadratic probing.

Applications: Linear probing is a hashing function just like quadratic probing which means it carries most of the benefits and applications of one. An example of an application for linear probing would be to differentiate different keywords in a programming language by the compiler. The compiler is able to do this by storing keywords in a set which is implemented using a hash table.

9 Hashing with Separate Chaining

Separate chaining is another way of handling hashing in which each position in the hash table has a list to handle collisions. Each position may be just a link to the list or may be an item and a link, essentially, the head of a list.

C++ Implementation of Separate Chaining

```cpp
class Hash
{
    int Bucket;     // No. of buckets
    // Pointer to an array containing buckets
    list<int> *table;
public:
    Hash(int H);

    void insertItem(int x);

    void deleteItem(int key);

    int hashFunction(int x) {
        return (x % Bucket);
    }
    void displayHash();
};

// Constructor
Hash::Hash(int b)
{
    this->Bucket = b;
    table = new list<int>[Bucket];
}
// Inserting items into the array
void Hash::insertItem(int key)
{
    int index = hashFunction(key);
    table[index].push_back(key);
}

```

```
// Function to display hash table
void Hash::displayHash() {
    for (int i = 0; i < Bucket; i++) {
     for (auto x : table[i]){
         cout << x << endl;
  }
 }
}

// Driver program
int main()
{
  // Clock timer
  auto start = chrono::steady_clock::now();
  std::chrono::time_point<std::chrono::high_resolution_clock> end;

  // Array that contains keys to be mapped
  int a[] = {50, 700, 76, 85, 92, 73, 101};
  int n = sizeof(a)/sizeof(a[0]);

  // Insert the keys into the hash table
  Hash h(7);
    for (int i = 0; i < n; i++){
        h.insertItem(a[i]);
    }

  // Display the Hash table
  h.displayHash();

 // Time implementation
  cout << "Elapsed time in nanoseconds: "
              << chrono::duration_cast<chrono::nanoseconds>(start -
     end).count()
              << " ns" << endl;

  cout << "Elapsed time in milliseconds: "
              << chrono::duration_cast<chrono::milliseconds>(start
    - end).count()
              << " ms" << endl;

  cout << "Elapsed time in seconds: "
              << chrono::duration_cast<chrono::seconds>(start - end
    ).count()
              << " sec";
  return 0;
```

Advantages: Some benefits of separate chaining are that it is relatively easy for a programmer to create, and implement one. You can add more values to the elements to the chain since the hash map never fills up.

Disadvantages: An influential disadvantage for linear probing is it has poor cache performance, and some parts are never used in the hash table. It also has a poor time complexity when the chain becomes too long.

Time test: To test the time that the program took I used chrono file from the library. I measured how long the program took in nanoseconds, miliseconds, and finally seconds. In the program I tested how long it took for the separate chaining to insert integer values in my array, and print them out. The results are as follows,Elapsed time in nanoseconds: 958214649 ns, Elapsed time in milliseconds: 9582 ms, Elapsed time in seconds: 9 secs. This relates to the disadvantage of it having a poor cache performance and taking unneeded storage which explains why it takes longer.

Applications: Separate chaining uses hashing which is similar, but not the same as the algorithm that is used in the famous crypto currency bitcoin. The algorithm is used specifically during the mining where ”block requires the miner to produce a value (a nonce) that, after being hashed” (Hayes, 2021). The reason bitcoin uses this technique is to make sure that the value of bitcoin does not go down. For example, gold is only valuable because it takes a long time to mine it, and if it did not and there was an abundance supply of gold there would be no value in gold.

10 Binary Search Tree

A binary tree is a non-linear data structure where each node is either 0,1 and can have a maximum of 2 nodes. Binary search trees are sorted binary trees that provide faster insertion, deletion, and searches as compared to a regular binary tree.

C++ Implementation of Binary Search Tree

```
#include <iostream>
#include <chrono>
using namespace std;

class BinarySearchTree
{
    int data;
    BinarySearchTree *leftNode, *rightNode;

public:
    //Default Constructor
    BinarySearchTree();

    // Constructor specifying in passing int value
    BinarySearchTree(int);

    // Insert function.
    BinarySearchTree* Insert(BinarySearchTree*, int);

    // Display Function
    void Display(BinarySearchTree*);
};

// Default Constructor definition.
BinarySearchTree::BinarySearchTree(){
    data = 0;
    leftNode = nullptr;
    rightNode= nullptr;
}

```

```cpp
// Constructor specifying in passing int value
BinarySearchTree::BinarySearchTree(int value){
    data = value;
    leftNode = nullptr;
    rightNode = nullptr;
}


// Insert function definition.
BinarySearchTree* BinarySearchTree::Insert(BinarySearchTree* root,
    int value)
{
    if (!root)
    {
        //Insert the first node if root is not full
        return new BinarySearchTree(value);
    }

    if (value > root->data)
    {
        // Insert right node if value going to be inserted is
    greater than root node data
        root->rightNode = Insert(root->rightNode, value);
    }
    else
    {
        // Insert left node if value going to be inserted is
    greater than root node data
        root->leftNode = Insert(root->leftNode, value);
    }

    // Return root node after insertion
    return root;
}

void BinarySearchTree::Display(BinarySearchTree* root)
{
    Display(root->leftNode);
    //Printing values in nodes in sorted order
    cout << root->data << endl;
    Display(root->rightNode);
}

// Driver code
int main()
{
    // Clock timer
    auto start = chrono::steady_clock::now();
    std::chrono::time_point<std::chrono::high_resolution_clock> end
    ;

    // Populating values to be inserted into the binary search tree
    BinarySearchTree b, *root = nullptr;
    root = b.Insert(root, 50);
    b.Insert(root, 700);
    b.Insert(root, 76);
    b.Insert(root, 85);
    b.Insert(root, 92);
    b.Insert(root, 73);
    b.Insert(root, 101);

    // Displaying sorted binary search tree
    b.Display(root);

    // Time implementation
      cout << "Elapsed time in nanoseconds: "
                   << chrono::duration_cast<chrono::nanoseconds>(
    start - end).count()
                   << " ns" << endl;

      cout << "Elapsed time in milliseconds: "
                   << chrono::duration_cast<chrono::milliseconds>(
    start - end).count()
                   << " ms" << endl;

      cout << "Elapsed time in seconds: "
                   << chrono::duration_cast<chrono::seconds>(start -
     end).count()
                   << " sec";

    return 0;
}
```

Advantages: A benefit of a binary search tree is that it "does not store an index of its data elements. Instead, it relies on its implicit structure (left or right of each node) to keep a record of where each element is. The result is insertion and deletion at logarithmic time, or O(log n)" (Ang , 2017). A binary search tree is also faster at inserting, and deleting than a regular array.

Disadvantages: An influential disadvantage for a binary search tree is that it can become unbalanced if left unregulated.

Time test: To test the time that the program took I used chrono file from the library. I measured how long the program took in nanoseconds, miliseconds, and finally seconds. In the program I tested how long it took for the separate chaining to insert integer values in my array, and print them out. The results are as follows,Elapsed time in nanoseconds: 759234659 ns, Elapsed time in milliseconds: 7592 ms, Elapsed time in seconds: 7 secs. This shows how efficient a binary search tree is in inserting values and displaying them over a regular array.

Applications: Binary search trees are mainly used in sorting data, multi-level indexing, and finally various searches.

11 Conclusion

In conclusion, some searches are better in time and efficiency compared to other searches. However, that does not mean that each search does not have a specific job that is aligned for the search. Linear search is inefficient like stated before if we have a large input size. It is more efficient with small input size over binary search. Comparing with Linear search, Binary search is more efficient when the input size is large.

12 References

References

[1] Kush Agra *Applications of hashing* Retrieved November 15, 2021

[2] Nick Ang *Why use Binary Search Tree?* Retrieved November 29, 2021

[3] Connor Fehrenbach *Separate chaining* Retrieved October 13, 2021

[4] Adam Hayes *Target hash defintion* Retrieved October 18, 2021

[5] Nicole Sobnom *Linear Search, Experts Exchange* Retrieved November 11, 2021

[6] Liam Tung *C++ Programming language Rust: Mozilla job cuts have hit us badly but here's how we'll survive* Retrieved October 27, 2021

[7] Shristi Uniyal *Quadratic probing in hashing* Retrieved October 27, 2021

[8] Ambika Choudry *C++ Why Microsoft Is Dumping C C++ For This New Programming Language* Retrieved October 27, 2020

[9] Andrew Zola *What is hashing and how does it work?* Retrieved October 29, 2021