**Design and Analysis of Algorithm Project Report**

**On**

**“Hashing”**

Submitted in the Partial fulfillment of the requirement for the Award of Degree of

**Bachelor of Technology**

in

**COMPUTER SCIENCE & ENGINEERING**



|  |  |
| --- | --- |
| **Submitted to**  **Er. Bhuvnesh Kumar** | **Submitted by**  **AayushKhanna (1701214)**  **Anuj Sharma (1701234)**  **ArshpreetKaur (1701237)**  **BadalKashyap (1701242)**  **Jobandeep Singh(1701275)** |

**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

**AMRITSAR GROUP OF COLLEGES, Amritsar**

**(Autonomous college under UGC Act – 1956[2(f) and 12(b)]**

**ACKNOWLEDGEMENT**

This is a humble effort to express our sincere gratitude towards those who have guided and helped us to complete this project.

A project is major milestone during the study period of a student. As such this project was a challenge to us and was an opportunity to prove our caliber. We are highly grateful and obliged to each and everyone making us help out of problems being faced by us.

It would not have been possible to see through the undertaken project without the guidance of

Er. Sarabpreet Singh. It was purely on the basis of their experience and knowledge that we able to clear all the theoretical and technical hurdles during the development phases of this project work.

Last but not the least we are very thankful to our Head of Department Er. Vinod Sharma and all Members of Computer Science Department, who gave us an opportunity to face real time problems while fulfilling need of an organization by making projects for them.

**DECLARATION**

We hereby declare that the project work entitled **“Hashing”** is an authentic record of our work carried out as requirements of Institutional Training project for the award of degree of B.Tech(CSE), **Amritsar Group Of Colleges, Amritsar,** under the guidance of **Er**. **Sarabpreet** **Singh.**

(Signature of student)

**Aayush Khanna (1701214)**

**Anuj Sharma (1701234)**

**Arshpreet Kaur (1701237)**

**Badal Kashyap (1701242)**

**Jobandeep Singh(1701275)**

Certified that the above statement made by the student is correct to the best of our knowledge and belief.

**Faculty Coordinator**

Er. Sarabpreet Singh (Assistant Professor – CSE Department)

INDEX PAGE

|  |  |  |
| --- | --- | --- |
| Sno | Contents | Page No |
| 1 | Introduction to Design and analysis of algorithm | 5 |
| 2 | Applications of DAA | 7 |
| 3 | About Hashing | 9 |
| 4 | About Project | 12 |
| 5 | Code | 15 |
| 6 | Snapshots of output | 24 |
|  |  |  |

**INTRODUCTION TO DESIGN AND ANALYSIS OF ALGORITHM**

An **Algorithm** is a sequence of steps to solve a problem. **Design** and **Analysis** of **Algorithm** is very important for designing **algorithm** to solve different types of problems in the branch of computer science and information technology.

An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks. An algorithm is an efficient method that can be expressed within finite amount of time and space.

An algorithm is the best way to represent the solution of a particular problem in a very simple and efficient way. If we have an algorithm for a specific problem, then we can implement it in any programming language, meaning that the algorithm is independent from any programming languages.

**Algorithm Design**

The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space.

To solve a problem, different approaches can be followed. Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient. However, one has to keep in mind that both time consumption and memory usage cannot be optimized simultaneously. If we require an algorithm to run in lesser time, we have to invest in more memory and if we require an algorithm to run with lesser memory, we need to have more time.

**Problem Development Steps**

The following steps are involved in solving computational problems.

* Problem definition
* Development of a model
* Specification of an Algorithm
* Designing an Algorithm
* Checking the correctness of an Algorithm
* Analysis of an Algorithm
* Implementation of an Algorithm
* Program testing
* Documentation

**Characteristics of Algorithms**

The main characteristics of algorithms are as follows −

* Algorithms must have a unique name
* Algorithms should have explicitly defined set of inputs and outputs
* Algorithms are well-ordered with unambiguous operations
* Algorithms halt in a finite amount of time. Algorithms should not run for infinity, i.e., an algorithm must end at some point

In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. The term **"analysis of algorithms"** was coined by Donald Knuth.

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Most algorithms are designed to work with inputs of arbitrary length. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as **time complexity**, or volume of memory, known as **space complexity**.

## The Need for Analysis

By considering an algorithm for a specific problem, we can begin to develop pattern recognition so that similar types of problems can be solved by the help of this algorithm.

Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis −

* **Worst-case** − The maximum number of steps taken on any instance of size **a**.
* **Best-case** − The minimum number of steps taken on any instance of size **a**.
* **Average case** − An average number of steps taken on any instance of size **a**.
* **Amortized** − A sequence of operations applied to the input of size **a** averaged over time.

To solve a problem, we need to consider time as well as space complexity as the program may run on a system where memory is limited but adequate space is available or may be vice-versa. In this context, if we compare **bubble sort** and **merge sort**. Bubble sort does not require additional memory, but merge sort requires additional space. Though time complexity of bubble sort is higher compared to merge sort, we may need to apply bubble sort if the program needs to run in an environment, where memory is very limited.

**APPLICATIONS OF DESIGN AND ANALYSIS OF ALGORITHM**

Design and analysis of algorithm serve as the basis for [abstract data types](https://en.wikipedia.org/wiki/Abstract_data_type) (ADT). The ADT defines the logical form of the data type. The design and analysis of algorithm implements the physical form of the data type.

Different types of design and analysis of algorithm are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, relational databases commonly use [B-tree](https://en.wikipedia.org/wiki/B-tree) indexes for data retrieval ,while [compiler](https://en.wikipedia.org/wiki/Compiler) implementations usually use [hash tables](https://en.wikipedia.org/wiki/Hash_table) to look up identifiers.

Design and analysis of algorithm provide a means to manage large amounts of data efficiently for uses such as large [databases](https://en.wikipedia.org/wiki/Database) and [internet indexing services](https://en.wikipedia.org/wiki/Web_indexing). Usually, efficient data structures are key to designing efficient [algorithms](https://en.wikipedia.org/wiki/Algorithm). Some formal design methods and [programming languages](https://en.wikipedia.org/wiki/Programming_language) emphasize data structures, rather than algorithms, as the key organizing factor in software design. Design and analysis of algorithm can be used to organize the storage and retrieval of information stored in both [main memory](https://en.wikipedia.org/wiki/Main_memory) and [secondary memory](https://en.wikipedia.org/wiki/Secondary_memory).

Data structures are generally based on the ability of a [computer](https://en.wikipedia.org/wiki/Computer) to fetch and store data at any place in its memory, specified by a [pointer](https://en.wikipedia.org/wiki/Pointer_(computer_programming))—a bit string, representing a [memory address](https://en.wikipedia.org/wiki/Memory_address), that can be itself stored in memory and manipulated by the program. Thus, the [array](https://en.wikipedia.org/wiki/Array_data_structure) and [record](https://en.wikipedia.org/wiki/Record_(computer_science)) data structures are based on computing the addresses of data items with [arithmetic operations](https://en.wikipedia.org/wiki/Arithmetic_operations), while the [linked data structures](https://en.wikipedia.org/wiki/Linked_data_structure) are based on storing addresses of data items within the structure itself.

The implementation of a data structure usually requires writing a set of [procedures](https://en.wikipedia.org/wiki/Subroutine) that create and manipulate instances of that structure. The efficiency of a data structure cannot be analyzed separately from those operations. This observation motivates the theoretical concept of an [abstract data type](https://en.wikipedia.org/wiki/Abstract_data_type), a data structure that is defined indirectly by the operations that may be performed on it, and the mathematical properties of those operations (including their space and time cost).

There are numerous types of data structures, generally built upon simpler [primitive data types](https://en.wikipedia.org/wiki/Primitive_data_type):[[9]](https://en.wikipedia.org/wiki/Data_structure" \l "cite_note-9)

* An [array](https://en.wikipedia.org/wiki/Array_data_structure) is a number of elements in a specific order, typically all of the same type (depending on the language, individual elements may either all be forced to be the same type, or may be of almost any type). Elements are accessed using an integer index to specify which element is required. Typical implementations allocate contiguous memory words for the elements of arrays (but this is not always a necessity). Arrays may be fixed-length or resizable.
* A [linked list](https://en.wikipedia.org/wiki/Linked_list) (also just called list) is a linear collection of data elements of any type, called nodes, where each node has itself a value, and points to the next node in the linked list. The principal advantage of a linked list over an array, is that values can always be efficiently inserted and removed without relocating the rest of the list. Certain other operations, such as [random access](https://en.wikipedia.org/wiki/Random_access) to a certain element, are however slower on lists than on arrays.
* A [record](https://en.wikipedia.org/wiki/Record_(computer_science)) (also called tuple or struct) is an aggregate data structure. A record is a value that contains other values, typically in fixed number and sequence and typically indexed by names. The elements of records are usually called fields or members.
* A [union](https://en.wikipedia.org/wiki/Union_(computer_science)) is a data structure that specifies which of a number of permitted primitive types may be stored in its instances, e.g. float or long integer. Contrast with a [record](https://en.wikipedia.org/wiki/Record_(computer_science)), which could be defined to contain a float and an integer; whereas in a union, there is only one value at a time. Enough space is allocated to contain the widest member datatype.
* A [tagged union](https://en.wikipedia.org/wiki/Tagged_union) (also called [variant](https://en.wikipedia.org/wiki/Variant_type), variant record, discriminated union, or disjoint union) contains an additional field indicating its current type, for enhanced type safety.
* An [object](https://en.wikipedia.org/wiki/Object_(computer_science)) is a data structure that contains data fields, like a record does, as well as various [methods](https://en.wikipedia.org/wiki/Method_(computer_programming)) which operate on the data contents. An object is an in-memory instance of a class from a taxonomy. In the context of [object-oriented programming](https://en.wikipedia.org/wiki/Object-oriented_programming), records are known as [plain old data structures](https://en.wikipedia.org/wiki/Plain_old_data_structure) to distinguish them from objects.

In addition, [graphs](https://en.wikipedia.org/wiki/Graph_(computer_science)) and [binary trees](https://en.wikipedia.org/wiki/Binary_trees) are other commonly used data structures.

**HASHING**

**Hashing** is the transformation of a string of [character](https://whatis.techtarget.com/definition/character)s into a usually shorter fixed-length value or key that represents the original string. Hashing is used to index and retrieve items in a [database](https://searchsqlserver.techtarget.com/definition/database) because it is faster to find the item using the shorter hashed key than to find it using the original value. It is also used in many [encryption](https://searchsecurity.techtarget.com/definition/encryption) algorithms.

The hashing [algorithm](https://whatis.techtarget.com/definition/algorithm) is called the **hash function***--* probably the term is derived from the idea that the resulting hash value can be thought of as a "mixed up" version of the represented value.

In addition to faster data retrieval, hashing is also used to encrypt and decrypt digital signatures (used to authenticate message senders and receivers). The [digital signature](https://searchsecurity.techtarget.com/definition/digital-signature) is transformed with the hash function and then both the hashed value (known as a **message-digest**) and the signature are sent in separate transmissions to the receiver. Using the same hash function as the sender, the receiver derives a message-digest from the signature and compares it with the message-digest it also received. (They should be the same.)

The hash function is used to index the original value or key and then used later each time the data associated with the value or key is to be retrieved. Thus, hashing is always a one-way operation. There's no need to "reverse engineer" the hash function by analyzing the hashed values. In fact, the ideal hash function can't be derived by such analysis. A good hash function also should not produce the same hash value from two different inputs. If it does, this is known as a **collision.** A hash function that offers an extremely low risk of collision may be considered acceptable.

Here are some relatively simple hash functions that have been used:

* **Division-remainder method*:*** The size of the number of items in the table is estimated. That number is then used as a divisor into each original value or key to extract a quotient and a remainder. The remainder is the hashed value. (Since this method is liable to produce a number of collisions, any search mechanism would have to be able to recognize a collision and offer an alternate search mechanism.)
* **Folding method:** This method divides the original value (digits in this case) into several parts, adds the parts together, and then uses the last four digits (or some other arbitrary number of digits that will work ) as the hashed value or key.
* **Radix transformation method:** Where the value or key is digital, the number base (or radix) can be changed resulting in a different sequence of digits. (For example, a decimal numbered key could be transformed into a hexadecimal numbered key.) High-order digits could be discarded to fit a hash value of uniform length.
* **Digit rearrangement method:** This is simply taking part of the original value or key such as digits in positions 3 through 6, reversing their order, and then using that sequence of digits as the hash value or key.

There are several well-known hash functions used in cryptography. These include the message-digest hash functions [MD2](https://searchsecurity.techtarget.com/definition/MD2), [MD4](https://searchsecurity.techtarget.com/definition/MD4), and [MD5](https://searchsecurity.techtarget.com/definition/MD5), used for hashing digital signatures into a shorter value called a message-digest, and the Secure Hash Algorithm (SHA), a standard algorithm, that makes a larger (60-bit) message digest and is similar to MD4. A hash function that works well for database storage and retrieval, however, might not work as for cryptographic or error-checking purposes.

A hash function is any [function](https://en.m.wikipedia.org/wiki/Function_(mathematics)) that can be used to map [data](https://en.m.wikipedia.org/wiki/Data_(computing)) of arbitrary size to fixed-size values. The values returned by a hash function are called **hash values**, hash codes, digests, or simply hashes. The values are used to index a fixed-size table called a **hash table**. Use of a hash function to index a hash table is called **hashing** or scatter storage addressing.

Hash functions and their associated hash tables are used in data storage and retrieval applications to access data in a small and nearly constant time per retrieval, and storage space only fractionally greater than the total space required for the data or records themselves. Hashing is a computationally and storage space efficient form of data access which avoids the non-linear access time of ordered and unordered lists and structured trees, and the often exponential storage requirements of direct access of state spaces of large or variable-length keys.

Use of hash functions relies on statistical properties of key and function interaction: worst case behavior is intolerably bad with a vanishingly small probability, and average case behavior can be nearly optimal (minimal collisions).[[1]](https://en.m.wikipedia.org/wiki/Hash_function#cite_note-1)

Hash functions are related to (and often confused with) [checksums](https://en.m.wikipedia.org/wiki/Checksums), [check digits](https://en.m.wikipedia.org/wiki/Check_digit), [fingerprints](https://en.m.wikipedia.org/wiki/Fingerprint_(computing)), [lossy compression](https://en.m.wikipedia.org/wiki/Lossy_compression), [randomization functions](https://en.m.wikipedia.org/wiki/Randomization_function), [error-correcting codes](https://en.m.wikipedia.org/wiki/Error_correction_code), and [ciphers](https://en.m.wikipedia.org/wiki/Cipher). Although the concepts overlap to some extent, each one has its own uses and requirements and is designed and optimized differently.

A hash function takes as input a key, which is associated with a datum or record and used to identify it to the data storage and retrieval application. The keys may be fixed length, like an integer, or variable length, like a name. In some cases, the key is the datum itself. The output is a hash code used to index a hash table holding the data or records, or pointers to them.

A hash function may be considered to perform three functions:

* Convert variable length keys into fixed length (usually machine word length or less) values, by folding them by words or other units using a parity-preserving operator like ADD or XOR.
* Scramble the bits of the key so that the resulting values are uniformly distributed over the key space.
* Map the key values into ones less than or equal to the size of the table

A good hash function satisfies two basic properties: 1) it should be very fast to compute; 2) it should minimize duplication of output values (collisions). Hash functions rely on generating favorable probability distributions for their effectiveness, reducing access time to nearly constant. High table loading factors, pathological key sets and poorly designed hash functions can result in access times approaching linear in the number of items in the table. Hash functions can be designed to give best worst-case performance,[[Notes 1]](https://en.m.wikipedia.org/wiki/Hash_function" \l "cite_note-2) good performance under high table loading factors, and in special cases, perfect (collisionless) mapping of keys into hash codes. Implementation is based on parity-preserving bit operations (XOR and ADD), multiply, or divide. A necessary adjunct to the hash function is a collision-resolution method that employs an auxiliary data structure like linked lists, or systematic probing of the table to find an empty slot.

Hash tables [Edit](https://en.m.wikipedia.org/w/index.php?title=Hash_function&action=edit&section=2)

[**Hash table**](https://en.m.wikipedia.org/wiki/Hash_table)

Hash functions are used in conjunction with hash tables to store and retrieve data items or data records. The hash function translates the key associated with each datum or record into a hash code which is used to index the hash table. When an item is to be added to the table, the hash code may index an empty slot (also called a bucket), in which case the item is added to the table there. If the hash code indexes a full slot, some kind of collision resolution is required: the new item may be omitted (not added to the table), or replace the old item, or it can be added to the table in some other location by a specified procedure. That procedure depends on the structure of the hash table: In chained hashing, each slot is the head of a linked list or chain, and items that collide at the slot are added to the chain. Chains may be kept in random order and searched linearly, or in serial order, or as a self-ordering list by frequency to speed up access. In open address hashing, the table is probed starting from the occupied slot in a specified manner, usually by [linear probing](https://en.m.wikipedia.org/wiki/Linear_probing), [quadratic probing](https://en.m.wikipedia.org/wiki/Quadratic_probing), or [double hashing](https://en.m.wikipedia.org/wiki/Double_hashing) until an open slot is located or the entire table is probed (overflow). Searching for the item follows the same procedure until the item is located, an open slot is found or the entire table has been searched (item not in table).

**INTRODUCTION TO PROJECT**

**Aim:-** The project deals with access techniques on internet protocol using various techniques.

**Description:-**

In this project we are using 4 services :-

* **Naïve Approach** :- A **Naive** **algorithm** is usually the most obvious solution when one is asked a problem. It may not be a smart **algorithm** but will probably get the job done (...eventually.) Eg. Trying to search for an element in a sorted array. A **Naive** **algorithm** would be to use a Linear Search. A Not-So **Naive** Solution would be to use the Binary Search.

**Pros:**

* It is easy and fast to predict class of test data set. It also perform well in multi class prediction
* When assumption of independence holds, a Naive Bayes classifier performs better compare to other models like logistic regression and you need less training data.
* It perform well in case of categorical input variables compared to numerical variable(s). For numerical variable, normal distribution is assumed (bell curve, which is a strong assumption).

**Cons:**

* If categorical variable has a category (in test data set), which was not observed in training data set, then model will assign a 0 (zero) probability and will be unable to make a prediction. This is often known as “Zero Frequency”. To solve this, we can use the smoothing technique. One of the simplest smoothing techniques is called Laplace estimation.
* On the other side naive Bayes is also known as a bad estimator, so the probability outputs from predict\_proba are not to be taken too seriously.
* Another limitation of Naive Bayes is the assumption of independent predictors. In real life, it is almost impossible that we get a set of predictors which are completely independent.

## 4 Applications of Naive Bayes Algorithms

* **Real time Prediction:-**Naive Bayes is an eager learning classifier and it is sure fast. Thus, it could be used for making predictions in real time.
* **Multi class Prediction:-**This algorithm is also well known for multi class prediction feature. Here we can predict the probability of multiple classes of target variable.
* **Text classification/ Spam Filtering/ Sentiment Analysis:-** Naive Bayes classifiers mostly used in text classification (due to better result in multi class problems and independence rule) have higher success rate as compared to other algorithms. As a result, it is widely used in Spam filtering (identify spam e-mail) and Sentiment Analysis (in social media analysis, to identify positive and negative customer sentiments)
* **Recommendation System:-**Naive Bayes Classifier and [Collaborative Filtering](https://en.wikipedia.org/wiki/Collaborative_filtering) together builds a Recommendation System that uses machine learning and data mining techniques to filter unseen information and predict whether a user would like a given resource or not
* **Direct Addressing:-** Direct Address Table is a data structure that has the capability of mapping records to their corresponding keys using arrays. In direct address tables, records are placed using their key values directly as indexes. They facilitate fast searching, insertion and deletion operations.

**Advantages:-**

1. **Searching in O(1) Time:** Direct address tables use arrays which are random access data structure, so, the key values (which are also the index of the array) can be easily used to search the records in O(1) time.
2. **Insertion in O(1) Time:** We can easily insert an element in an array in O(1) time. The same thing follows in a direct address table also.
3. **Deletion in O(1) Time:** Deletion of an element takes O(1) time in an array. Similarly, to delete an element in a direct address table we need O(1) time.

**Limitations:**

1. Prior knowledge of maximum key value
2. Practically useful only if the maximum value is very less.
3. It causes wastage of memory space if there is a significant difference between total records and maximum value.

* **Hash Table:-** **Hash** **Table** is a **data** structure which stores **data** in an associative manner. In a **hash** **table**, **data** is stored in an array format, where each **data** value has its own unique index value. Access of **data** becomes very fast if we know the index of the desired **data**.
* Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.
* Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.
* Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.

**Time Complexity for various Techniques:-**

1. Naive Approach:-O(n)
2. Direct Addressing :- O(1) ( But the space complexity is O(n) that can be upto 2^32 According to IPv4.)
3. Hash Table :- Constant
4. List :- O(n)

**CODING**

#include<iostream>

#include<math.h>

#include<string>

#include<vector>

#include<map>

using namespace std;

void ll(int,string);

int B[]={0,0,0,0,0,0,0};

string A[]={"192.0.0.1","122.11.12.1","133.90.45.6","221.22.34.5","144.23.1.1","0.0.8.12","0.1.9.78"};

int N=7;

vector<string> v;

map<string,int> m;

void naive()

{

cout<<"Service A Selected"<<endl;

int r,i;

r=(rand()%(N-1+1))+1;

r=r-1;

string a[]={"192.0.0.1","122.11.12.1","133.90.45.6","221.22.34.5","144.23.1.1","0.0.8.12","0.1.9.78"}; int b[]={0,0,0,0,0,0,0};

string ip=a[r];

for(i=0;i<N;i++)

{

int z=ip.compare(a[i]);

if(z==0)

{

b[i]=b[i]+1;

break;

}

}

for(i=0;i<N;i++)

{

int z=ip.compare(A[i]);

if(z==0)

{

B[i]=B[i]+1;

break;

}

}

cout<<"A hash function is any function that can be used to map data of arbitrary size to fixed-size values. The values returned by a hash function are called hash values, hash codes, digests, or simply hashes. The values are used to index a fixed-size table called a hash table. Use of a hash function to index a hash table is called hashing or scatter storage addressing."<<endl<<endl;

int c;

cout<<"Press 1-To Get Connected IP Inforamtion Or 0 To Skip: ";

cin>>c;

if(c==1)

{

cout<<"Currently Used IP: "<<ip<<endl;

cout<<"IP: "<<ip<<" Has Been Used For "<<B[i]<<" Times"<<endl;

}

string s1=A[i];

m[s1]=B[i];

ll(1,ip);

}

void da()

{

int a[100000],i,j;

cout<<"Service B Selected"<<endl;

for(i=0;i<100000;i++)

a[i]=0;

int r=(rand()%(N-6+1))+6;

r=r-1;

string ip=A[r];

int nn[]={0,0,0,0};

i=0;

for(j=0;j<4;j++)

{

while(ip[i]!='.')

{

int x=(int)ip[i];

x=x-48;

nn[j]=nn[j]\*10+x;

i++;

if(ip[i]=='\0')

break;

}

i++;

}

int v=nn[0]\*pow(2,24)+nn[1]\*pow(2,16)+nn[2]\*pow(2,8)+nn[3]\*pow(2,0);

a[v]=a[v]+1;

for(i=0;i<N;i++)

{

int z=ip.compare(A[i]);

if(z==0)

{

B[i]=B[i]+1;

break;

}

}

cout<<"It is clear that a term of address is always closely linked with the pronoun 'you,' which in itself has vocative qualities. One could say, in fact, that whenever pronominal 'you' is used in direct address, vocative 'you' is implicitly present. The two kinds of 'you' are inextricably bound together, though in an utterance like 'You! What do you think you're doing!"<<endl<<endl;

int c;

cout<<"Press 1-To Get Connected IP Inforamtion Or 0 To Skip: ";

cin>>c;

if(c==1)

{

cout<<"Currently Used IP: "<<ip<<endl;

cout<<"IP: "<<ip<<" Has Been Used For "<<B[i]<<" Times"<<endl;

}

string s1=A[i];

m[s1]=B[i];

ll(1,ip);

}

void ll(int x,string ipp)

{

int i;

if(x==0)

{

cout<<"Service C Selected"<<endl;

int r=(rand()%(N-1+1))+1;

r=r-1;

string ip=A[r];

v.push\_back(ip);

for(i=0;i<N;i++)

{

int z=ip.compare(A[i]);

if(z==0)

{

B[i]=B[i]+1;

break;

}

}

}

else

{

v.push\_back(ipp);

/\* for(i=0;i<N;i++)

{

int z=ipp.compare(A[i]);

if(z==0)

{

B[i]=B[i]+1;

break;

}

}

\*/

}

if(x==0)

{

cout<<"This is a List Based approach to perform the Operations.It uses a dynamic array that implements the IP addreses counts them"<<endl<<endl;

int c;

cout<<"Press 1-To Get Connected IP Inforamtion Or 0 To Skip: ";

cin>>c;

if(c==1)

{

cout<<"Currently Used IP: "<<A[i]<<endl;

int co=0;

for(int k=0;k<v.size();k++)

{

int z=A[i].compare(v[k]);

if(z==0)

co++;

}

cout<<"IP: "<<A[i]<<" Has Been Used For "<<co<<" Times"<<endl;

}

}

void ht()

{

int i;

cout<<"In computing, a hash table (hash map) is a data structure that implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found."<<endl<<endl;

cout<<endl<<endl;

cout<<"[ IP ADDRESS "<<"COUNT ]"<<endl;

for(map<string,int>::iterator it=m.begin();it!=m.end();it++)

cout<<" "<<it->first<<" : "<<it->second<<endl;

}

int main()

{

cout<<"|||||||||||||||||||| INTERNET PROTOCOL ACCESS TECNIQUES ||||||||||||||||||||"<<endl<<endl;

int i,j=0,k,p,r;

char c;

string s;

for(i=0;i<N;i++)

m.insert(pair<string,int>(A[i],0));

cout<<"Enter Pin: ";

cin>>p;

r=(p%(100-1+1))+1;

if((r>=10)&&(r<=20))

{

cout<<"Connecting...."<<endl;

cout<<"Connected"<<endl<<endl<<endl;

while(1)

{

if(j==1)

{

cout<<"Press Y To Enter Service Again Or N To Skip: ";

cin>>c;

}

if((j==0)||((c=='Y')||(c=='y')))

{

cout<<"Select Key In Order To Get Service"<<endl<<endl;

cout<<"1-Service A\n"<<"2-Service B\n"<<"3-Service C\n"<<"4-Service D\n"<<endl;

cout<<"Enter Key: ";

cin>>k;

cout<<endl;

switch(k)

{

case 1:

naive();

break;

case 2:

da();

break;

case 3:

ht();

break;

case 4:

ll(0,s);

break;

case 5:

break;

}

}

else {

break;

j=1;

}

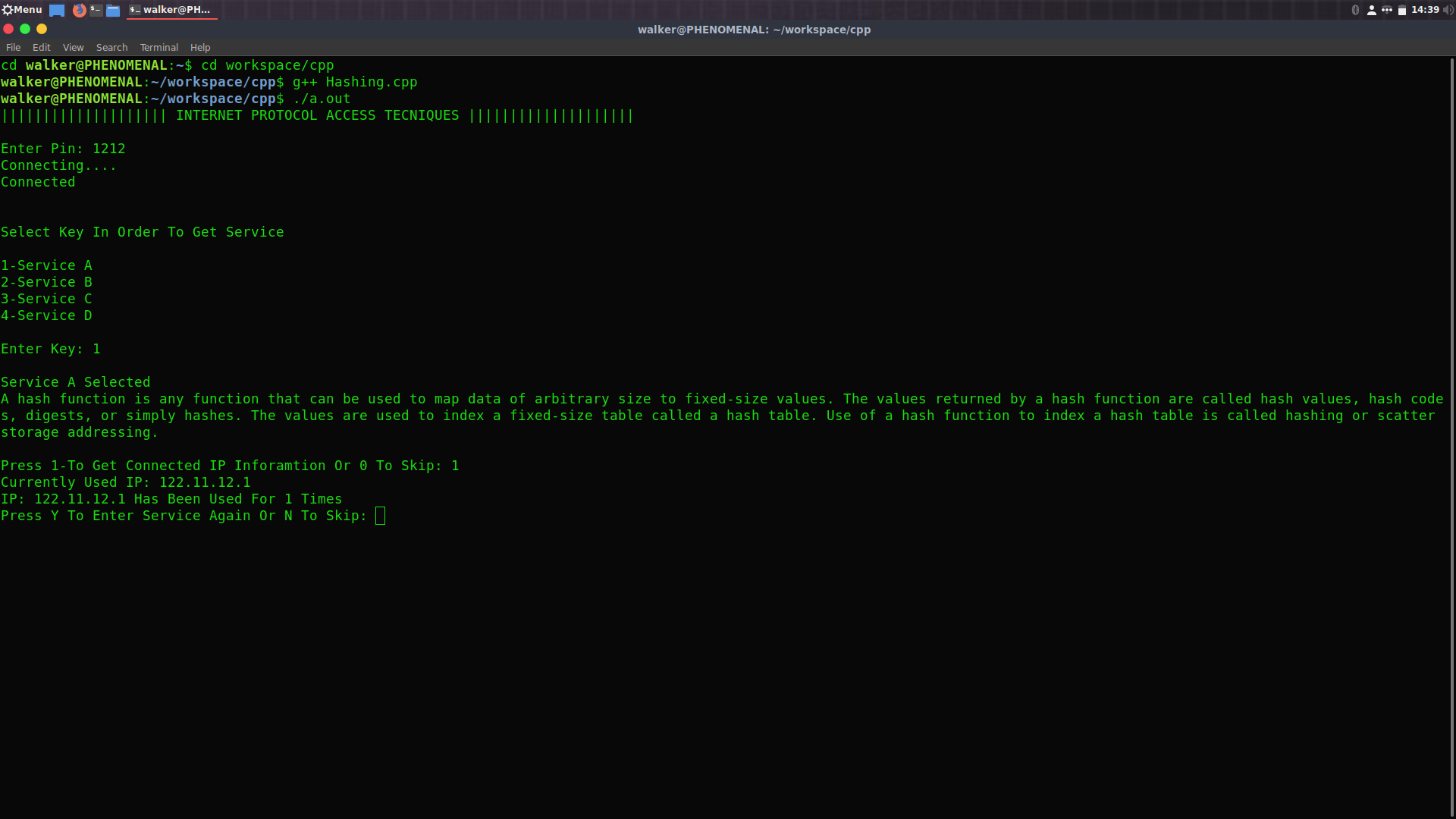
}

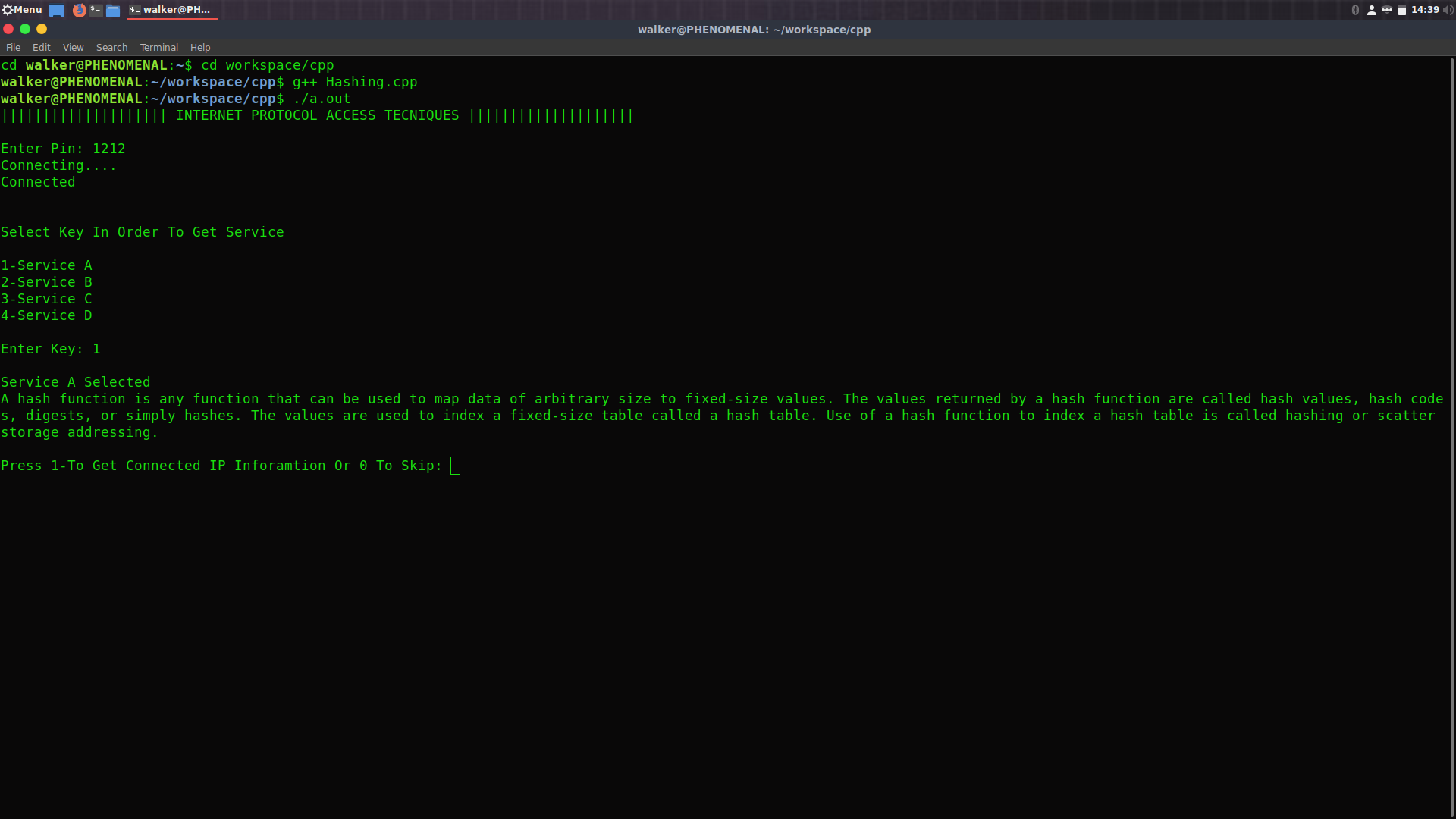
}

cout<<endl<<"Thanks For Visiting"<<endl<<endl;

return 0;

}

**SNAPSHOTS**

****

