BATU-EXAM

Made by batuexams.com

at MET Bhujbal Knowledege City

Data Structures Department

The PDF notes on this website are the copyrighted property of batuexams.com.

All rights reserved.

DOWNLOADED FROM BATU-EXAMS.in

BTCOC303: Data Structures

[UNIT 1] Introduction

Data, Data types, Data structure, Abstract Data Type (ADT), representation of Information, characteristics of algorithm, program, analysing programs. Arrays and Hash Tables Concept of sequential organization, linear and non-linear data structure, storage representation, array processing sparse matrices, transpose of sparse matrices, Hash Tables, Direct address tables, Hash tables, Hash functions, Open addressing, Perfect hashing.

Data

Data is defined as a collection of individual facts or statistics. (While “datum” is technically the singular form of “data,” it’s not commonly used in everyday language.) Data can come in the form of text, observations, figures, images, numbers, graphs, or symbols. For example, data might include individual prices, weights, addresses, ages, names, temperatures, dates, or distances.

Data is a raw form of knowledge and, on its own, doesn’t carry any significance or purpose. In other words, you have to interpret data for it to have meaning. Data can be simple—and may even seem useless until it is analysed, organized, and interpreted.

Data Types

A *data type* is the most basic and the most common classification of data. It is this through which the compiler gets to know the form or the type of information that will be used throughout the code. So basically, data type is a type of information transmitted between the programmer and the compiler where the programmer informs the compiler about what type of data is to be stored and also tells how much space it requires in the memory. Some basic examples are int, string etc. It is the type of any variable used in the code.

```
#include <iostream.h>
using namespace std;

void main()
{
     int a;
     a = 5;

     float b;
     b = 5.0;

     char c;
     c = 'A';

     char d[10];
     d = "example";
}
```

DOWNLOADED FROM BATU-EXAMS.in

As seen from the theory explained above we come to know that in the above code, the variable ‘a’ is of data type integer which is denoted by int a. So the variable ‘a’ will be used as an integer type variable throughout the process of the code. And, in the same way, the variables ‘b’, ‘c’ and ‘d’ are of type float, character and string respectively. And all these are kinds of data types.

Data Structure

- **Data structure** is a representation of the logical relationship existing between individual elements of data.
- Data Structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.
- We can also define data structure as a mathematical or logical model of a particular organization of data items.
- The representation of particular data structure in the main memory of a computer is called as **storage structure.**
- The storage structure representation in auxiliary memory is called as **file structure**.
- It is defined as the way of storing and manipulating data in organized form so that it can be used efficiently.
- Data Structure mainly specifies the following four things
    - Organization of Data
    - Accessing methods
    - Degree of associativity
    - Processing alternatives for information
- Algorithm + Data Structure = Program
- Data structure study covers the following points
    - Amount of memory require to store.
    - Amount of time require to process.
    - Representation of data in memory.
    - Operations performed on that data.

A *data structure* is a collection of different forms and different types of data that has a set of specific operations that can be performed. It is a collection of data types. It is a way of organizing the items in terms of memory, and also the way of accessing each item through some defined logic. Some examples of data structures are stacks, queues, linked lists, binary tree and many more.

Data structures perform some special operations only like insertion, deletion and traversal. For example, you have to store data for many employees where each employee has his name, employee id and a mobile number. So, this kind of data requires complex data management, which means it requires data structure comprised of multiple primitive data types. So, data structures are one of the most important aspects when implementing coding concepts in real-world applications.

DOWNLOADED FROM BATU-EXAMS.in

Classification of Data Structure

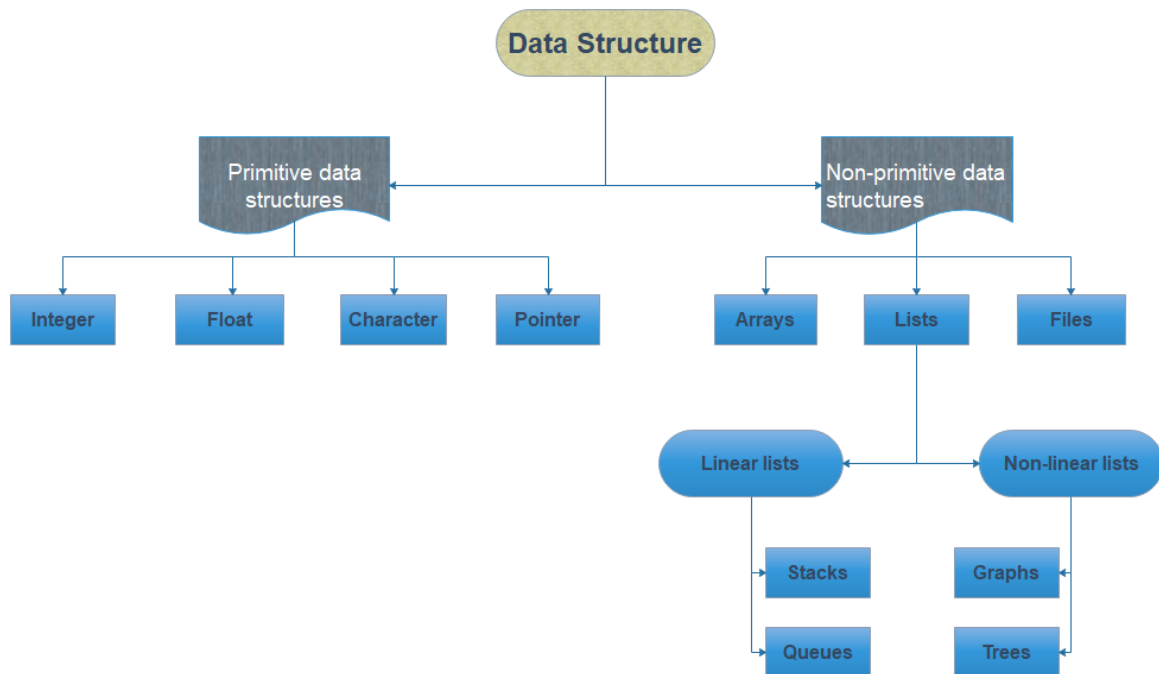


Figure: Classification of Data Structure.

Data Structures are normally classified into two broad categories

1. Primitive Data Structure
2. Non-primitive data Structure

Data types

A particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.

Primitive Data Structure

- Primitive data structures are basic structures and are directly operated upon by machine instructions.
- Primitive data structures have different representations on different computers.
- Integers, floats, character and pointers are examples of primitive data structures.
- These data types are available in most programming languages as built in type.
  - Integer: It is a data type which allows all values without fraction part. We can use it for whole numbers.
  - Float: It is a data type which use for storing fractional numbers.
  - Character: It is a data type which is used for character values.
- **Pointer:** A variable that holds memory address of another variable are called pointer.

DOWNLOADED FROM BATU-EXAMS.in

Non-primitive Data Type

- These are more sophisticated data structures.
- These are derived from primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.
- Examples of Non-primitive data type are Array, List, and File etc.
- A Non-primitive data type is further divided into Linear and Non-Linear data structure
    - **Array:** An array is a fixed-size sequenced collection of elements of the same data type.
    - **List:** An ordered set containing variable number of elements is called as Lists.
    - **File:** A file is a collection of logically related information. It can be viewed as a large list of records consisting of various fields.

Linear data structures

- A data structure is said to be Linear, if its elements are connected in linear fashion by means of logically or in sequence memory locations.
- There are two ways to represent a linear data structure in memory,
    - Static memory allocation
    - Dynamic memory allocation
- The possible operations on the linear data structure are: Traversal, Insertion, Deletion, Searching, Sorting and Merging.
- Examples of Linear Data Structure are Stack and Queue.
- **Stack:** Stack is a data structure in which insertion and deletion operations are performed at one end only.
    - The insertion operation is referred to as 'PUSH' and deletion operation is referred to as 'POP' operation.
    - Stack is also called as Last in First out (LIFO) data structure.
- **Queue:** The data structure which permits the insertion at one end and Deletion at another end, known as Queue.
    - End at which deletion is occurs is known as FRONT end and another end at which insertion occurs is known as REAR end.
    - Queue is also called as First in First out (FIFO) data structure.

Nonlinear data structures

- Nonlinear data structures are those data structure in which data items are not arranged in a sequence.
- Examples of Non-linear Data Structure are Tree and Graph.
- **Tree:** A tree can be defined as finite set of data items (nodes) in which data items are arranged in branches and sub branches according to requirement.
    - Trees represent the hierarchical relationship between various elements.
    - Tree consist of nodes connected by edge, the node represented by circle and edge lives connecting to circle.
- **Graph:** Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.
    - A tree can be viewed as restricted graph.
    - Graphs have many types:

DOWNLOADED FROM BATU-EXAMS.in

- Un-directed Graph
- Directed Graph
- Mixed Graph
- Multi Graph
- Simple Graph
- Null Graph
- Weighted Graph

Difference between Linear and Non-Linear Data Structure

Linear Data Structure	Non-Linear Data Structure
Every item is related to its previous and next time.	Every item is attached with many other items.
Data is arranged in linear sequence.	Data is not arranged in sequence.
Data items can be traversed in a single run.	Data cannot be traversed in a single run.
E.g. Array, Stacks, linked list, queue.	E.g. tree, graph.
Implementation is easy.	Implementation is difficult.

Difference between data type and data structure:

Data Types	Data Structures
Data Type is the kind or form of a variable which is being used throughout the program. It defines that the particular variable will assign the values of the given data type only	Data Structure is the collection of different kinds of data. That entire data can be represented using an object and can be used throughout the entire program.
Implementation through Data Types is a form of abstract implementation	Implementation through Data Structures is called concrete implementation
Can hold values and not data, so it is data less	Can hold different kind and types of data within one single object
Values can directly be assigned to the data type variables	The data is assigned to the data structure object using some set of algorithms and operations like push, pop and so on.
No problem of time complexity	Time complexity comes into play when working with data structures
Examples: int, float, double	Examples: stacks, queues, tree

Operation on Data Structures

Design of efficient data structure must take operations to be performed on the data structures into account. The most commonly used operations on data structure are broadly categorized into following types

DOWNLOADED FROM BATU-EXAMS.in

1. **Create:-** The create operation results in reserving memory for program elements. This can be done by declaration statement. Creation of data structure may take place either during compile-time or run-time. malloc() function of C language is used for creation.
2. **Destroy:-** Destroy operation destroys memory space allocated for specified data structure. free() function of C language is used to destroy data structure.
3. **Selection:-** Selection operation deals with accessing a particular data within a data structure.
4. **Updation:-** It updates or modifies the data in the data structure.
5. **Searching:-** It finds the presence of desired data item in the list of data items, it may also find the locations of all elements that satisfy certain conditions.
6. **Sorting:-** Sorting is a process of arranging all data items in a data structure in a particular order, say for example, either in ascending order or in descending order.
7. **Merging:-** Merging is a process of combining the data items of two different sorted list into a single sorted list.
8. **Splitting:-** Splitting is a process of partitioning single list to multiple list.
9. **Traversal:-** Traversal is a process of visiting each and every node of a list in systematic manner.

Abstract Data Types (ADT)

In this, we will learn about ADT but before understanding what ADT is let us consider different in-built data types that are provided to us. Data types such as int, float, double, long, etc. are considered to be in-built data types and we can perform basic operations with them such as addition, subtraction, division, multiplication, etc. Now there might be a situation when we need operations for our user-defined data type which have to be defined. These operations can be defined only as and when we require them. So, in order to simplify the process of solving problems, we can create data structures along with their operations, and such data structures that are not in-built are known as Abstract Data Type (ADT).

Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it gives an implementation-independent view.

The process of providing only the essentials and hiding the details is known as abstraction.

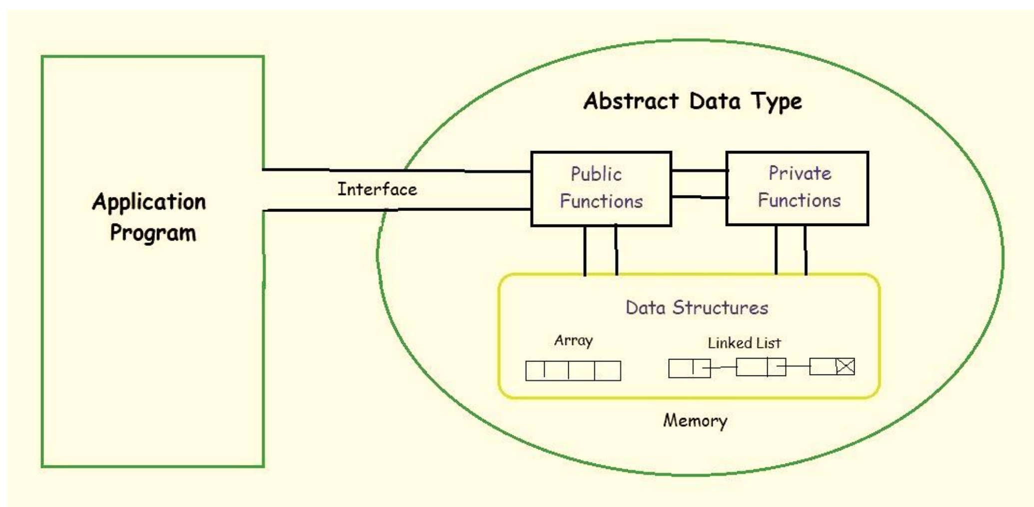


Figure: Abstract data type

DOWNLOADED FROM BATU-EXAMS.in

The user of data type does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented.
So, a user only needs to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type. Now we'll define three ADTs namely List ADT, Stack ADT, Queue ADT.

1. List ADT

- The data is generally stored in key sequence in a list which has a head structure consisting of *count*, *pointers* and *address of compare function* needed to compare the data in the list.
- The data node contains the *pointer* to a data structure and a *self-referential pointer* which points to the next node in the list.
- The **List ADT Functions** is given below:
- get() – Return an element from the list at any given position.
- insert() – Insert an element at any position of the list.
- remove() – Remove the first occurrence of any element from a non-empty list.
- removeAt() – Remove the element at a specified location from a non-empty list.
- replace() – Replace an element at any position by another element.
- size() – Return the number of elements in the list.
- isEmpty() – Return true if the list is empty, otherwise return false.
- isFull() – Return true if the list is full, otherwise return false.

2. Stack ADT

- In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.
- The program allocates memory for the *data* and *address* is passed to the stack ADT.
- The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
- The stack head structure also contains a pointer to *top* and *count* of number of entries currently in stack.
- push() – Insert an element at one end of the stack called top.
- pop() – Remove and return the element at the top of the stack, if it is not empty.
- peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
- size() – Return the number of elements in the stack.
- isEmpty() – Return true if the stack is empty, otherwise return false.
- isFull() – Return true if the stack is full, otherwise return false.

DOWNLOADED FROM BATU-EXAMS.in

3. Queue ADT

- The queue abstract data type (ADT) follows the basic design of the stack abstract data type.
- Each node contains a void pointer to the *data* and the *link pointer* to the next element in the queue. The program's responsibility is to allocate memory for storing the data.
- enqueue() – Insert an element at the end of the queue.
- dequeue() – Remove and return the first element of the queue, if the queue is not empty.
- peek() – Return the element of the queue without removing it, if the queue is not empty.
- size() – Return the number of elements in the queue.
- isEmpty() – Return true if the queue is empty, otherwise return false.
- isFull() – Return true if the queue is full, otherwise return false.

Features of ADT:

- **Abstraction:** The user does not need to know the implementation of the data structure.
- **Better Conceptualization:** ADT gives us a better conceptualization of the real world.
- **Robust:** The program is robust and has the ability to catch errors.

From these definitions, we can clearly see that the definitions do not specify how these ADTs will be represented and how the operations will be carried out. There can be different ways to implement an ADT, for example, the List ADT can be implemented using arrays, or singly linked list or doubly linked list. Similarly, stack ADT and Queue ADT can be implemented using arrays or linked lists.

What is an Algorithm?

The word **Algorithm** means " A set of finite rules or instructions to be followed in calculations or other problem-solving operations " Or " A procedure for solving a mathematical problem in a finite number of steps that frequently involves recursive operations".

Therefore, Algorithm refers to a sequence of finite steps to solve a particular problem.

Algorithms can be simple and complex depending on what you want to achieve.

DOWNLOADED FROM BATU-EXAMS.in

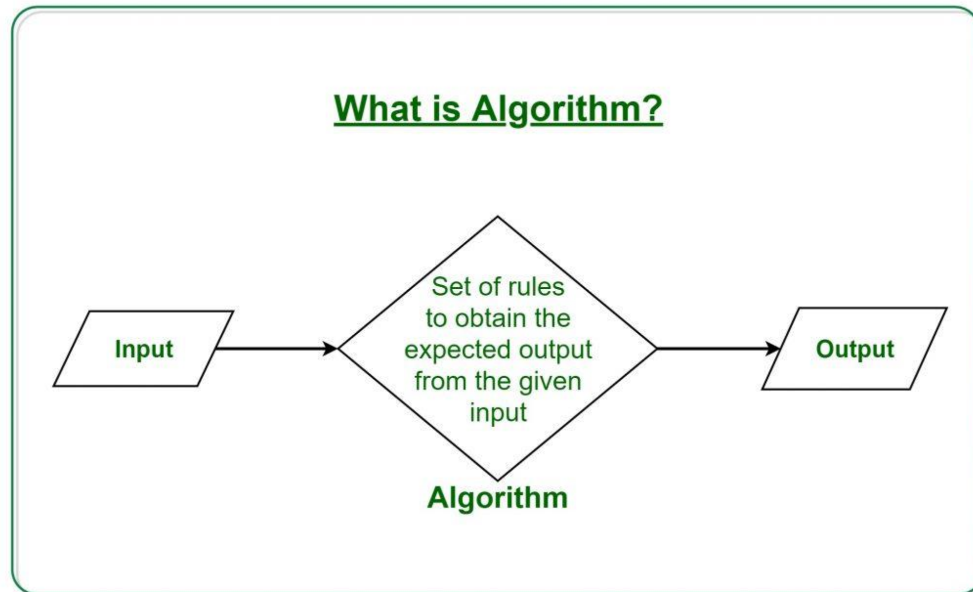


Figure: Algorithm

It can be understood by taking the example of cooking a new recipe. To cook a new recipe, one reads the instructions and steps and executes them one by one, in the given sequence. The result thus obtained is the new dish cooked perfectly. Every time you use your phone, computer, laptop, or calculator you are using Algorithms. Similarly, algorithms help to do a task in programming to get the expected output.

The Algorithm designed are language-independent, i.e. they are just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

Characteristics of an Algorithm

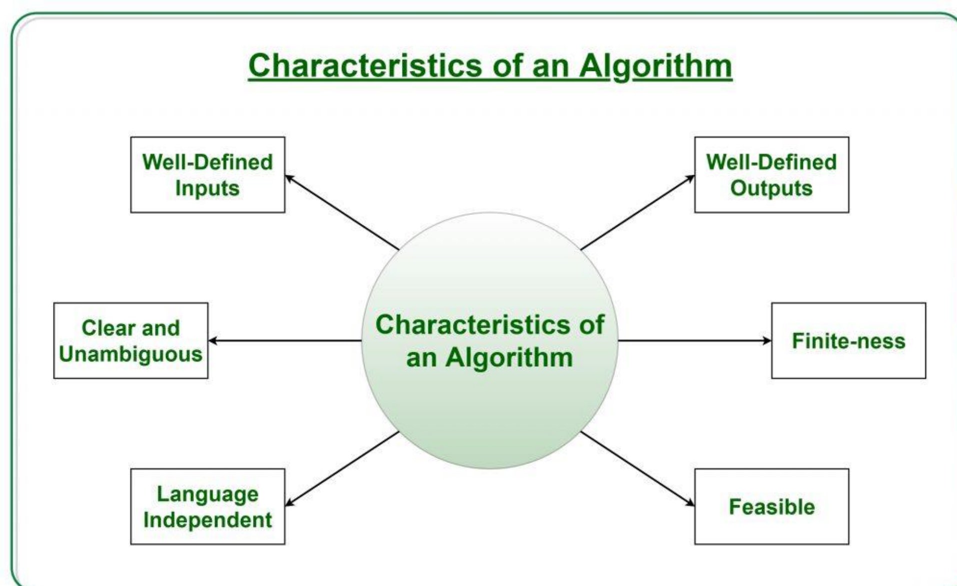


Figure: Characteristics of an Algorithm.

DOWNLOADED FROM BATU-EXAMS.in

As one would not follow any written instructions to cook the recipe, but only the standard one. Similarly, not all written instructions for programming is an algorithm. In order for some instructions to be an algorithm, it must have the following characteristics:

- **Clear and Unambiguous**: The algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs**: If an algorithm says to take inputs, it should be well-defined inputs. It may or may not take input.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well. It should take at least 1 output.
- **Finite-ness:** The algorithm must be finite, i.e. it should terminate after a finite time.
- **Feasible:** The algorithm must be simple, generic, and practical, such that it can be executed with the available resources. It must not contain some future technology or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

Properties of Algorithm:
- It should terminate after a finite time.
- It should produce at least one output.
- It should take zero or more input.
- It should be deterministic means giving the same output for the same input case.
- Every step in the algorithm must be effective i.e. every step should do some work.

Advantages of Algorithms:
- It is easy to understand.
- An algorithm is a step-wise representation of a solution to a given problem.
- In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Disadvantages of Algorithms:
- Writing an algorithm takes a long time so it is time-consuming.
- Understanding complex logic through algorithms can be very difficult.
- Branching and Looping statements are difficult to show in Algorithms**(imp)**.

How to Design an Algorithm?
In order to write an algorithm, the following things are needed as a pre-requisite:

1. The **problem** that is to be solved by this algorithm i.e. clear problem definition.
2. The **constraints** of the problem must be considered while solving the problem.
3. The **input** to be taken to solve the problem.
4. The **output** to be expected when the problem is solved.
5. The **solution** to this problem, is within the given constraints.

Then the algorithm is written with the help of the above parameters such that it solves the problem.

Example: Consider the example to add three numbers and print the sum.

DOWNLOADED FROM BATU-EXAMS.in

- **Step 1: Fulfilling the pre-requisites**
  As discussed above, in order to write an algorithm, its pre-requisites must be fulfilled.
  1. **The problem that is to be solved by this algorithm**: Add 3 numbers and print their sum.
  2. **The constraints of the problem that must be considered while solving the problem**: The numbers must contain only digits and no other characters.
  3. **The input to be taken to solve the problem:** The three numbers to be added.
  4. **The output to be expected when the problem is solved:** The sum of the three numbers taken as the input i.e. a single integer value.
  5. **The solution to this problem, in the given constraints:** The solution consists of adding the 3 numbers. It can be done with the help of '+' operator, or bit-wise, or any other method.

- **Step 2: Designing the algorithm**
  Now let's design the algorithm with the help of the above pre-requisites:
  Algorithm to add 3 numbers and print their sum:
  1. START
  2. Declare 3 integer variables num1, num2 and num3.
  3. Take the three numbers, to be added, as inputs in variables num1, num2, and num3 respectively.
  4. Declare an integer variable sum to store the resultant sum of the 3 numbers.
  5. Add the 3 numbers and store the result in the variable sum.
  6. Print the value of the variable sum
  7. END

- **Step 3: Testing the algorithm by implementing it.**
  In order to test the algorithm, let's implement it in C++ language.

```
// C++ program to add three numbers
// with the help of above designed
// algorithm
#include <bits/stdc++.h>
using namespace std;

int main()
{

     // Variables to take the input of
     // the 3 numbers
     int num1, num2, num3;

     // Variable to store the resultant sum
     int sum;

     // Take the 3 numbers as input
     cout << "Enter the 1st number: ";
     cin >> num1;
     cout << " " << num1 << endl;

     cout << "Enter the 2nd number: ";
```

DOWNLOADED FROM BATU-EXAMS.in

```
    cin >> num2;
    cout << " " << num2 << endl;

    cout << "Enter the 3rd number: ";
    cin >> num3;
    cout << " " << num3;

    // Calculate the sum using + operator
    // and store it in variable sum
    sum = num1 + num2 + num3;

    // Print the sum
    cout << "\nSum of the 3 numbers is: "
         << sum;

    return 0;
}
```

Output

```
Enter the 1st number: 0

Enter the 2nd number: 0

Enter the 3rd number: -1577141152


Sum of the 3 numbers is: -1577141152
```

How to analyze an Algorithm?

For a standard algorithm to be good, it must be efficient. Hence the efficiency of an algorithm must be checked and maintained. It can be in two stages:

1. **Priori Analysis:** "Priori" means "before". Hence Priori analysis means checking the algorithm before its implementation. In this, the algorithm is checked when it is written in the form of theoretical steps. This Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation. This is done usually by the algorithm designer. This analysis is independent of the type of hardware and language of the compiler. It gives the approximate answers for the complexity of the program.
2. **Posterior Analysis:** "Posterior" means "after". Hence Posterior analysis means checking the algorithm after its implementation. In this, the algorithm is checked by implementing it in any programming language and executing it. This analysis helps to get the actual and real analysis report about correctness (for every possible input/s if it shows/returns correct output or not), space required, time consumed etc. That is, it is dependent on the language of the compiler and the type of hardware used.

DOWNLOADED FROM BATU-EXAMS.in

What is Algorithm complexity and how to find it?

An algorithm is defined as complex based on the amount of Space and Time it consumes. Hence the Complexity of an algorithm refers to the measure of the Time that it will need to execute and get the expected output, and the Space it will need to store all the data (input, temporary data and output). Hence these two factors define the efficiency of an algorithm.

The two factors of Algorithm Complexity are:

- **Time Factor**: Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor**: Space is measured by counting the maximum memory space required by the algorithm to run/execute.

What is Algorithm complexity and how to find it?

An algorithm is defined as complex based on the amount of Space and Time it consumes. Hence the Complexity of an algorithm refers to the measure of the Time that it will need to execute and get the expected output, and the Space it will need to store all the data (input, temporary data and output). Hence these two factors define the efficiency of an algorithm.

The two factors of Algorithm Complexity are:

- **Time Factor**: Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor**: Space is measured by counting the maximum memory space required by the algorithm to run/execute.

Therefore, the **complexity of an algorithm can be divided into two types**:

1. Space Complexity: The space complexity of an algorithm refers to the amount of memory required by the algorithm to store the variables and get the result. This can be for inputs, temporary operations, or outputs.

How to calculate Space Complexity?

The space complexity of an algorithm is calculated by determining the following 2 components:

- **Fixed Part:** This refers to the space that is definitely required by the algorithm. For example, input variables, output variables, program size, etc.
- **Variable Part:** This refers to the space that can be different based on the implementation of the algorithm. For example, temporary variables, dynamic memory allocation, recursion stack space, etc.
  Therefore Space complexity **S(P)** of any algorithm P is **S(P) = C + SP(I)**, where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I.

2. Time Complexity: The time complexity of an algorithm refers to the amount of time that is required by the algorithm to execute and get the result. This can be for normal operations, conditional if-else statements, loop statements, etc.

How to calculate Time Complexity?

The time complexity of an algorithm is also calculated by determining the following 2 components:

- **Constant time part:** Any instruction that is executed just once comes in this part. For example, input, output, if-else, switch, arithmetic operations etc.
- **Variable Time Part:** Any instruction that is executed more than once, say n times, comes in this part. For example, loops, recursion, etc.

DOWNLOADED FROM BATU-EXAMS.in

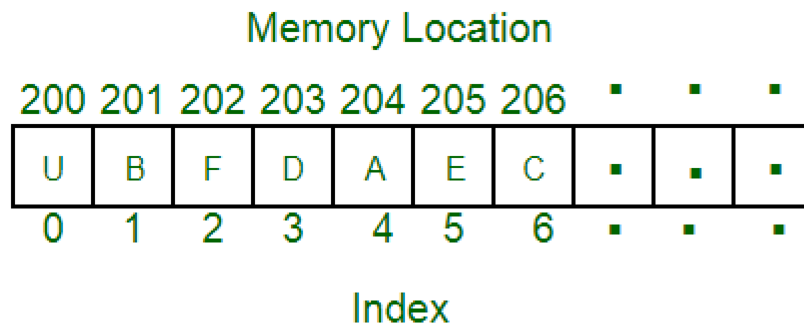
Therefore Time complexity **T(P)** of any algorithm P is **T(P) = C + TP(I)**, where C is the constant time part and TP(I) is the variable part of the algorithm, which depends on the instance characteristic I.

How to express an Algorithm?

1. **Natural Language: -** Here we express the Algorithm in natural English language. It is too hard to understand the algorithm from it.
2. **Flow Chart: -** Here we express the Algorithm by making graphical/pictorial representation of it. It is easier to understand than Natural Language.
3. **Pseudo Code: -** Here we express the Algorithm in the form of annotations and informative text written in plain English which is very much similar to the real code but as it has no syntax like any of the programming language, it can't be compiled or interpreted by the computer. It is the best way to express an algorithm because it can be understood by even a layman with some school level programming knowledge.

Array Data Structure

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).



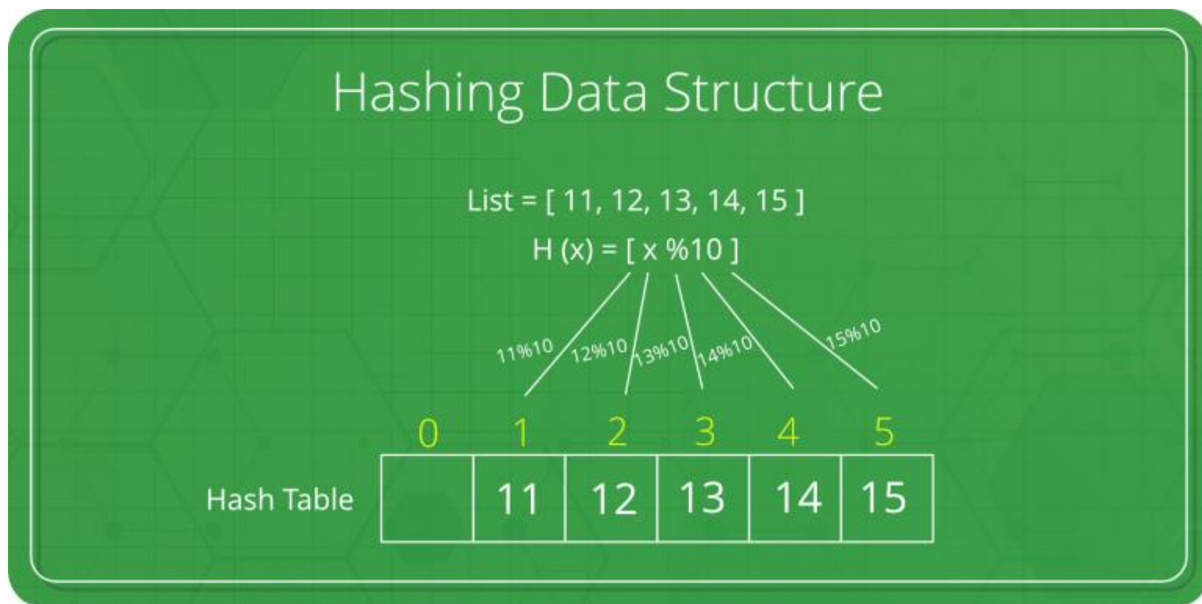
The above image can be looked as a top-level view of a staircase where you are at the base of the staircase. Each element can be uniquely identified by their index in the array (in a similar way as you could identify your friends by the step on which they were on in the above example).

Hashing Data Structure

Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.

Let a hash function H(x) maps the value ***X*** at the index **x%10** in an Array. For example, if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.

DOWNLOADED FROM BATU-EXAMS.in



Hash Table

Hash table is one of the most important data structures that uses a special function known as a hash function that maps a given value with a key to access the elements faster.

A Hash table is a data structure that stores some information, and the information has basically two main components, i.e., key and value. The hash table can be implemented with the help of an associative array. The efficiency of mapping depends upon the efficiency of the hash function used for mapping.

For example, suppose the key value is John and the value is the phone number, so when we pass the key value in the hash function shown as below:

Hash(key)= index;

When we pass the key in the hash function, then it gives the index.

Hash(john) = 3;

The above example adds the john at the index 3.

Drawback of Hash function

A Hash function assigns each value with a unique key. Sometimes hash table uses an imperfect hash function that causes a collision because the hash function generates the same key of two different values.

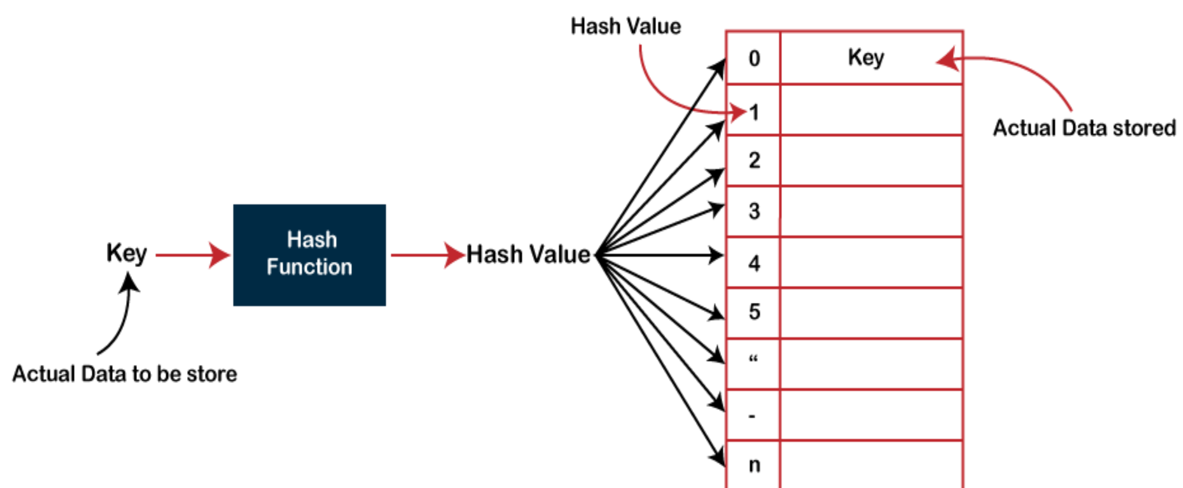
DOWNLOADED FROM BATU-EXAMS.in

Hashing

Hashing is one of the searching techniques that uses a constant time. The time complexity in hashing is O(1). Till now, we read the two techniques for searching, i.e., linear search and binary search. The worst time complexity in linear search is O(n), and O(logn) in binary search. In both the searching techniques, the searching depends upon the number of elements but we want the technique that takes a constant time. So, hashing technique came that provides a constant time.

In Hashing technique, the hash table and hash function are used. Using the hash function, we can calculate the address at which the value can be stored.

The main idea behind the hashing is to create the (key/value) pairs. If the key is given, then the algorithm computes the index at which the value would be stored. It can be written as:



There are three ways of calculating the hash function:

- **Division method**
- **Folding method**
- **Mid square method**

In the division method, the hash function can be defined as:

$$h(k_i) = k_i \% m;$$

where **m** is the size of the hash table.

For example, if the key value is 6 and the size of the hash table is 10. When we apply the hash function to key 6 then the index would be:

$h(6) = 6\%10 = 6$

The index is 6 at which the value is stored.

DOWNLOADED FROM BATU-EXAMS.in

Storage Representation

Data Structure is the way of storing data in computer's memory so that it can be used easily and efficiently. There are different data-structures used for the storage of data. It can also be defined as a mathematical or logical model of a particular organization of data items. The representation of particular data structure in the main memory of a computer is called as storage structure. **For Examples:** Array, Stack, Queue, Tree, Graph, etc.

Operations on different Data Structure:

There are different types of operations that can be performed for the manipulation of data in every data structure. Some operations are explained and illustrated below:

- **Traversing:** Traversing a Data Structure means to visit the element stored in it. It visits data in a systematic manner. This can be done with any type of DS.
  Below is the program to illustrate traversal in an array:

```
// C++ program to traversal in an array
#include <iostream>
using namespace std;

// Driver Code
int main()
{
    // Initialise array
    int arr[] = { 1, 2, 3, 4 };

    // size of array
    int N = sizeof(arr) / sizeof(arr[0]);

    // Traverse the element of arr[]
    for (int i = 0; i < N; i++) {

        // Print the element
        cout << arr[i] << ' ';
    }

    return 0;
}
```

Output:
1 2 3 4

- **Searching:** Searching means to find a particular element in the given data-structure. It is considered as successful when the required element is found. Searching is the operation which we can performed on data-structures like array, linked-list, tree, graph, etc. Below is the program to illustrate searching an element in an array:

```
// C++ program to searching in an array
#include <iostream>
using namespace std;
```

DOWNLOADED FROM BATU-EXAMS.in

```
// Function that finds element K in the
// array
void findElement(int arr[], int N, int K)
{

    // Traverse the element of arr[]
    // to find element K
    for (int i = 0; i < N; i++) {

        // If Element is present then
        // print the index and return
        if (arr[i] == K) {
            cout << "Element found!";
            return;
        }
    }

    cout << "Element Not found!";
}

// Driver Code
int main ()
{
    // Initialise array
    int arr[] = {1, 2, 3, 4};

    // Element to be found
    int K = 3;

    // size of array
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function Call
    findElement(arr, N, K);
    return 0;
}
```

Output:
Element found!

- **Insertion:** It is the operation which we apply on all the data-structures. Insertion means to add an element in the given data structure. The operation of insertion is successful when the required element is added to the required data-structure. It is unsuccessful in some cases when the size of the data structure is full and when there is no space in the data-structure to add any additional element. The insertion has the same name as an insertion in the data-structure as an array, linked-list, graph, tree. In stack, this operation is called Push. In the queue, this operation is called Enqueue. Below is the program to illustrate insertion in stack:

```
// C++ program for insertion in array
```

DOWNLOADED FROM BATU-EXAMS.in

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

// Function to print the array element
void printArray(int arr[], int N)
{
    // Traverse the element of arr[]
    for (int i = 0; i < N; i++) {

        // Print the element
        cout << arr[i] << ' ';
    }
}

// Driver Code
int main()
{
    // Initialise array
    int arr[4];

    // size of array
    int N = 4;

    // Insert elements in array
    for (int i = 1; i < 5; i++) {
        arr[i - 1] = i;
    }

    // Print array element
    printArray(arr, N);
    return 0;
}
```

Output:
1 2 3 4

- **Deletion:** It is the operation which we apply on all the data-structures. Deletion means to delete an element in the given data structure. The operation of deletion is successful when the required element is deleted from the data structure. The deletion has the same name as a deletion in the data-structure as an array, linked-list, graph, tree, etc. In stack, this operation is called Pop. In Queue this operation is called Dequeue. Below is the program to illustrate dequeue in Queue:

```
// C++ program for insertion in array
#include <bits/stdc++.h>
using namespace std;

// Function to print the element in stack
void printStack(stack<int> St)
{
```

DOWNLOADED FROM BATU-EXAMS.in

```
        // Traverse the stack
        while (!St.empty()) {

                // Print top element
                cout << St.top() << ' ';

                // Pop top element
                St.pop();
        }
}

// Driver Code
int main()
{
        // Initialise stack
        stack<int> St;

        // Insert Element in stack
        St.push(4);
        St.push(3);
        St.push(2);
        St.push(1);

        // Print elements before pop
        // operation on stack
        printStack(St);

        cout << endl;

        // Pop the top element
        St.pop();

        // Print elements after pop
        // operation on stack
        printStack(St);
        return 0;
}
```

Output:

1 2 3 4

2 3 4

Some other method:

Create: –

It reserves memory for program elements by declaring them. The creation of data structure Can be done during

1. Compile-time
2. Run-time.

DOWNLOADED FROM BATU-EXAMS.in

You can use malloc() function.

Selection: -
It selects specific data from present data. You can any select specific data by giving condition in loop.

Update
It updates the data in the data structure. You can also update any specific data by giving some condition in loop like select approach.

Sort
Sorting data in a particular order (ascending or descending).
We can take the help of many sorting algorithms to sort data in less time. Example: bubble sort which takes O(n^2) time to sort data. There are many algorithms present like merge sort, insertion sort, selection sort, quick sort, etc.

Merge
Merging data of two different orders in a specific order may ascend or descend. We use merge sort to merge sort data.

Split Data
Dividing data into different sub-parts to make the process complete in less time.

Array Processing Sparse Matrices

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements.

Example:

0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**

Sparse Matrix Representations can be done in many ways following are two common representations:

DOWNLOADED FROM BATU-EXAMS.in

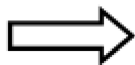
1. Array representation
2. Linked list representation

Using Array

2D array is used to represent a sparse matrix in which there are three rows named as

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non-zero element located at index – (row, column)

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$



Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

```
// C++ program for Sparse Matrix Representation
// using Array
#include <iostream>
using namespace std;

int main()
{
        // Assume 4x5 sparse matrix
        int sparseMatrix[4][5] =
        {
                {0 , 0 , 3 , 0 , 4 },
                {0 , 0 , 5 , 7 , 0 },
                {0 , 0 , 0 , 0 , 0 },
                {0 , 2 , 6 , 0 , 0 }
        };

        int size = 0;
        for (int i = 0; i < 4; i++)
                for (int j = 0; j < 5; j++)
                        if (sparseMatrix[i][j] != 0)
                                size++;

        // number of columns in compactMatrix (size) must be
        // equal to number of non - zero elements in
        // sparseMatrix
        int compactMatrix[3][size];

        // Making of new matrix
        int k = 0;
```

DOWNLOADED FROM BATU-EXAMS.in

```
        for (int i = 0; i < 4; i++)
            for (int j = 0; j < 5; j++)
                if (sparseMatrix[i][j] != 0)
                {
                    compactMatrix[0][k] = i;
                    compactMatrix[1][k] = j;
                    compactMatrix[2][k] = sparseMatrix[i][j];
                    k++;
                }

        for (int i=0; i<3; i++)
        {
            for (int j=0; j<size; j++)
                cout <<" "<< compactMatrix[i][j];

            cout <<"\n";
        }
        return 0;
    }
```

Output

```
0 0 1 1 3 3

2 4 2 3 1 2

3 4 5 7 2 6
```

Transpose of Sparse Matrices

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements.

Example:

```
0 0 3 0 4

0 0 5 7 0

0 0 0 0 0

0 2 6 0 0
```

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**

DOWNLOADED FROM BATU-EXAMS.in

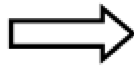
Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

Method 1: Using Arrays:
2D array is used to represent a sparse matrix in which there are three rows named as

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non-zero element located at index – (row, column)

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix}$$



Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

```
// C++ program for Sparse Matrix Representation
// using Array
#include <iostream>
using namespace std;

int main()
{
	// Assume 4x5 sparse matrix
	int sparseMatrix[4][5] =
	{
		{0 , 0 , 3 , 0 , 4 },
		{0 , 0 , 5 , 7 , 0 },
		{0 , 0 , 0 , 0 , 0 },
		{0 , 2 , 6 , 0 , 0 }
	};

	int size = 0;
	for (int i = 0; i < 4; i++)
		for (int j = 0; j < 5; j++)
			if (sparseMatrix[i][j] != 0)
				size++;
```

DOWNLOADED FROM BATU-EXAMS.in

```
    // number of columns in compactMatrix (size) must be
    // equal to number of non - zero elements in
    // sparseMatrix
    int compactMatrix[3][size];

    // Making of new matrix
    int k = 0;
    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 5; j++)
            if (sparseMatrix[i][j] != 0)
            {
                compactMatrix[0][k] = i;
                compactMatrix[1][k] = j;
                compactMatrix[2][k] = sparseMatrix[i][j];
                k++;
            }

    for (int i=0; i<3; i++)
    {
        for (int j=0; j<size; j++)
            cout <<" "<< compactMatrix[i][j];

        cout <<"\n";
    }
    return 0;
}
```

Output

```
 0 0 1 1 3 3

 2 4 2 3 1 2

 3 4 5 7 2 6
```

Given two sparse matrices, perform operations such as add, multiply or transpose of the matrices in their sparse form itself. The result should consist of three sparse matrices, one obtained by adding the two input matrices, one by multiplying the two matrices and one obtained by transpose of the first matrix.

Example: Note that other entries of matrices will be zero as matrices are sparse.

```
Input:

Matrix 1: (4x4)

Row Column Value

1   2       10
```

DOWNLOADED FROM BATU-EXAMS.in

```
1   4       12
3   3       5
4   1       15
4   2       12

Matrix 2: (4X4)
Row Column Value
1   3       8
2   4       23
3   3       9
4   1       20
4   2       25

Output:
Result of Addition: (4x4)
Row Column Value
1   2      10
1   3      8
1   4      12
2   4      23
3   3      14
4   1      35
4   2      37

Result of Multiplication: (4x4)
Row Column Value
1   1      240
1   2      300
1   4      230
3   3      45
4   3      120
4   4      276

Result of transpose on the first matrix: (4x4)
```

DOWNLOADED FROM BATU-EXAMS.in

Row	Column	Value
1	4	15
2	1	10
2	4	12
3	3	5
4	1	12

The sparse matrix used anywhere in the program is sorted according to its row values. Two elements with the same row values are further sorted according to their column values.

Now to **Add** the matrices, we simply traverse through both matrices element by element and insert the smaller element (one with smaller row and col value) into the resultant matrix. If we come across an element with the same row and column value, we simply add their values and insert the added data into the resultant matrix.

To **Transpose** a matrix, we can simply change every column value to the row value and vice-versa, however, in this case, the resultant matrix won't be sorted as we require. Hence, we initially determine the number of elements less than the current element's column being inserted in order to get the exact index of the resultant matrix where the current element should be placed. This is done by maintaining an array index [] whose i^{th} value indicates the number of elements in the matrix less than the column i.

To **Multiply** the matrices, we first calculate transpose of the second matrix to simplify our comparisons and maintain the sorted order. So, the resultant matrix is obtained by traversing through the entire length of both matrices and summing the appropriate multiplied values.

Any row value equal to x in the first matrix and row value equal to y in the second matrix (transposed one) will contribute towards result[x][y]. This is obtained by multiplying all such elements having col value in both matrices and adding only those with the row as x in first matrix and row as y in the second transposed matrix to get the result[x][y].

For example: Consider 2 matrices:

Row	Col	Val	Row	Col	Val
1	2	10	1	1	2
1	3	12	1	2	5
2	1	1	2	2	1
2	3	2	3	1	8

The resulting matrix after multiplication will be obtained as follows:

Transpose of second matrix:

Row	Col	Val	Row	Col	Val
1	2	10	1	1	2
1	3	12	1	3	8
2	1	1	2	1	5

DOWNLOADED FROM BATU-EXAMS.in

2   3   2   2   2   1

Summation of multiplied values:

result[1][1] = A[1][3]*B[1][3] = 12*8 = 96

result[1][2] = A[1][2]*B[2][2] = 10*1 = 10

result[2][1] = A[2][1]*B[1][1] + A[2][3]*B[1][3] = 2*1 + 2*8 = 18

result[2][2] = A[2][1]*B[2][1] = 1*5 = 5

Any other element cannot be obtained
by any combination of row in
Matrix A and Row in Matrix B.

Hence the final resultant matrix will be:

Row	Col	Val
1	1	96
1	2	10
2	1	18
2	2	5

Following is the implementation of above approach:

```
// C++ code to perform add, multiply
// and transpose on sparse matrices
#include <iostream>
using namespace std;

class sparse_matrix
{

	// Maximum number of elements in matrix
	const static int MAX = 100;

	// Double-pointer initialized by
	// the constructor to store
	// the triple-represented form
	int **data;
```

DOWNLOADED FROM BATU-EXAMS.in

```
        // dimensions of matrix
        int row, col;

        // total number of elements in matrix
        int len;

public:
        sparse_matrix(int r, int c)
        {

                // initialize row
                row = r;

                // initialize col
                col = c;

                // initialize length to 0
                len = 0;

                //Array of Pointer to make a matrix
                data = new int *[MAX];

                // Array representation
                // of sparse matrix
                //[,0] represents row
                //[,1] represents col
                //[,2] represents value
                for (int i = 0; i < MAX; i++)
                        data[i] = new int[3];
        }

        // insert elements into sparse matrix
        void insert(int r, int c, int val)
        {

                // invalid entry
                if (r > row || c > col)
                {
                        cout << "Wrong entry";
                }
                else
                {

                        // insert row value
                        data[len][0] = r;

                        // insert col value
                        data[len][1] = c;

                        // insert element's value
```

DOWNLOADED FROM BATU-EXAMS.in

```
                data[len][2] = val;

                // increment number of data in matrix
                len++;
            }
        }

        void add(sparse_matrix b)
        {

            // if matrices don't have same dimensions
            if (row != b.row || col != b.col)
            {
                cout << "Matrices can't be added";
            }

            else
            {
                int apos = 0, bpos = 0;
                sparse_matrix result(row, col);

                while (apos < len && bpos < b.len)
                {

                    // if b's row and col is smaller
                    if (data[apos][0] > b.data[bpos][0] ||
                    (data[apos][0] == b.data[bpos][0] &&
                        data[apos][1] > b.data[bpos][1]))

                    {

                        // insert smaller value into result
                        result.insert(b.data[bpos][0],
                                        b.data[bpos][1],
                                        b.data[bpos][2]);

                        bpos++;
                    }

                    // if a's row and col is smaller
                    else if (data[apos][0] < b.data[bpos][0] ||
                            (data[apos][0] == b.data[bpos][0] &&
                            data[apos][1] < b.data[bpos][1]))

                    {

                        // insert smaller value into result
                        result.insert(data[apos][0],
                                        data[apos][1],
                                        data[apos][2]);
```

DOWNLOADED FROM BATU-EXAMS.in

```
                        apos++;
                    }

                    else
                    {

                        // add the values as row and col is same
                        int addedval = data[apos][2] +
                                        b.data[bpos][2];

                        if (addedval != 0)
                            result.insert(data[apos][0],
                                            data[apos][1],
                                            addedval);
                        // then insert
                        apos++;
                        bpos++;
                    }
                }

                // insert remaining elements
                while (apos < len)
                    result.insert(data[apos][0],
                                    data[apos][1],
                                    data[apos++][2]);

                while (bpos < b.len)
                    result.insert(b.data[bpos][0],
                                    b.data[bpos][1],
                                    b.data[bpos++][2]);

                // print result
                result.print();
            }
        }

        sparse_matrix transpose()
        {

            // new matrix with inversed row X col
            sparse_matrix result(col, row);

            // same number of elements
            result.len = len;

            // to count number of elements in each column
            int *count = new int[col + 1];

            // initialize all to 0
```

DOWNLOADED FROM BATU-EXAMS.in

```
        for (int i = 1; i <= col; i++)
            count[i] = 0;

        for (int i = 0; i < len; i++)
            count[data[i][1]]++;

        int *index = new int[col + 1];

        // to count number of elements having
        // col smaller than particular i

        // as there is no col with value < 0
        index[0] = 0;

        // initialize rest of the indices
        for (int i = 1; i <= col; i++)

            index[i] = index[i - 1] + count[i - 1];

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

            // insert a data at rpos and
            // increment its value
            int rpos = index[data[i][1]]++;

            // transpose row=col
            result.data[rpos][0] = data[i][1];

            // transpose col=row
            result.data[rpos][1] = data[i][0];

            // same value
            result.data[rpos][2] = data[i][2];
        }

        // the above method ensures
        // sorting of transpose matrix
        // according to row-col value
        return result;
    }

    void multiply(sparse_matrix b)
    {
        if (col != b.row)
        {

            // Invalid multiplication
            cout << "Can't multiply, Invalid dimensions";
            return;
```

DOWNLOADED FROM BATU-EXAMS.in

```
        }

        // transpose b to compare row
        // and col values and to add them at the end
        b = b.transpose();
        int apos, bpos;

        // result matrix of dimension row X b.col
        // however b has been transposed,
        // hence row X b.row
        sparse_matrix result(row, b.row);

        // iterate over all elements of A
        for (apos = 0; apos < len;)
        {

            // current row of result matrix
            int r = data[apos][0];

            // iterate over all elements of B
            for (bpos = 0; bpos < b.len;)
            {

                // current column of result matrix
                // data[,0] used as b is transposed
                int c = b.data[bpos][0];

                // temporary pointers created to add all
                // multiplied values to obtain current
                // element of result matrix
                int tempa = apos;
                int tempb = bpos;

                int sum = 0;

                // iterate over all elements with
                // same row and col value
                // to calculate result[r]
                while (tempa < len && data[tempa][0] == r &&
                    tempb < b.len && b.data[tempb][0] == c)
                {
                    if (data[tempa][1] < b.data[tempb][1])

                        // skip a
                        tempa++;

                    else if (data[tempa][1] > b.data[tempb][1])

                        // skip b
                        tempb++;
```

DOWNLOADED FROM BATU-EXAMS.in

```
                    else

                        // same col, so multiply and increment
                        sum += data[tempa++][2] *
                            b.data[tempb++][2];
                }

                // insert sum obtained in result[r]
                // if its not equal to 0
                if (sum != 0)
                    result.insert(r, c, sum);

                while (bpos < b.len &&
                    b.data[bpos][0] == c)

                    // jump to next column
                    bpos++;
            }
            while (apos < len && data[apos][0] == r)

                // jump to next row
                apos++;
        }
        result.print();
    }

    // printing matrix
    void print()
    {
        cout << "\nDimension: " << row << "x" << col;
        cout << "\nSparse Matrix: \nRow\tColumn\tValue\n";

        for (int i = 0; i < len; i++)
        {
            cout << data[i][0] << "\t " << data[i][1]
                << "\t " << data[i][2] << endl;
        }
    }
};

// Driver Code
int main()
{

    // create two sparse matrices and insert values
    sparse_matrix a(4, 4);
    sparse_matrix b(4, 4);

    a.insert(1, 2, 10);
    a.insert(1, 4, 12);
```

DOWNLOADED FROM BATU-EXAMS.in

```
        a.insert(3, 3, 5);
        a.insert(4, 1, 15);
        a.insert(4, 2, 12);
        b.insert(1, 3, 8);
        b.insert(2, 4, 23);
        b.insert(3, 3, 9);
        b.insert(4, 1, 20);
        b.insert(4, 2, 25);

        // Output result
        cout << "Addition: ";
        a.add(b);
        cout << "\nMultiplication: ";
        a.multiply(b);
        cout << "\nTranspose: ";
        sparse_matrix atranspose = a.transpose();
        atranspose.print();
}
```

Output

```
Addition:
Dimension: 4x4
Sparse Matrix:
Row     Column     Value
1       2       10
1       3       8
1       4       12
2       4       23
3       3       14
4       1       35
4       2       37


Multiplication:
Dimension: 4x4
Sparse Matrix:
Row     Column     Value
1       1       240
1       2       300
1       4       230
```

DOWNLOADED FROM BATU-EXAMS.in

```
3     3     45
4     3     120
4     4     276

Transpose:
Dimension: 4x4
Sparse Matrix:
Row    Column    Value
  1      4      15
  2      1      10
  2      4      12
  3      3      5
  4      1      12
```

Hash Tables

Hash tables are the data structures which favor efficient storage and retrieval of data elements which are **linear** in nature.

Dictionaries:

- Dictionaries is a collection of data elements uniquely identified by field called Key. A directory supports operations of search, insert and delete.
- A dictionary supports both Sequential and Random Access. A sequential access is the process in which the data elements of the dictionary are ordered and accessed according to the order of the process in which the data elements of the dictionary are not accessed according to a particular order.
- Hash tables are ideal data structures for dictionaries.

Hash Search:

- Hash selection is a search in which the key, through an algorithmic function, determines the location of the data.
- Hashing it is a key to address transformation in which the keys map to addresses in a list.

Figure: Hash Search

DOWNLOADED FROM BATU-EXAMS.in

Hash Functions:

- A hash function is a mathematical function which maps a given key of the dictionary to its corresponding location in the storage table (known as hash table).
- The process of mapping the keys to their respective position in hash table is called as **Hashing**.

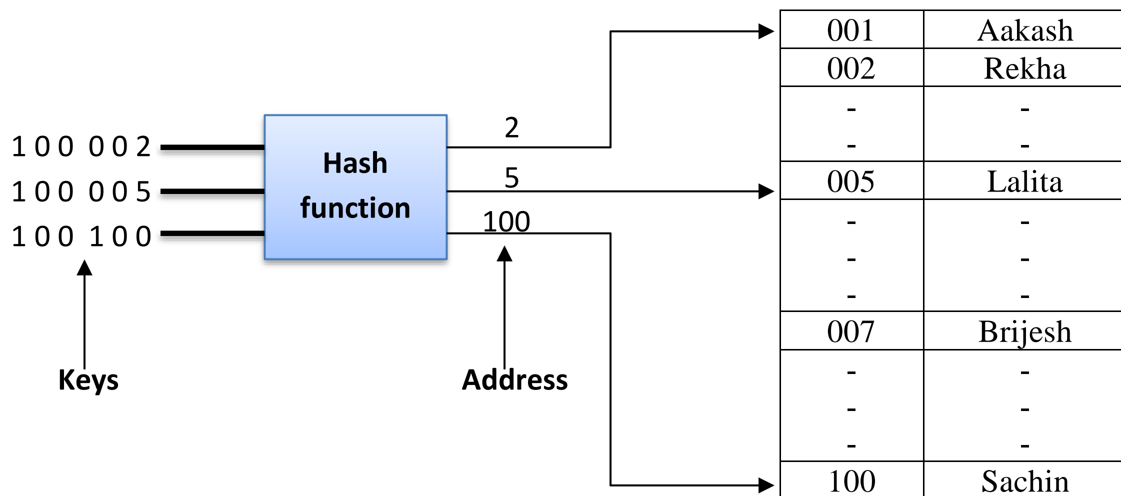


Figure: Hash Search

- The choice of the hash function plays a significant role in the performance of the hash table. It is therefore essential that a hash function satisfies following characteristics:

Characteristics of Hash Functions:

- Easy and quick to compute.
- Even distribution of keys across the hash table.
- A hash function must minimize collision.

Basic Definitions of Hashing:

- **Synonyms:** The set of Keys that hash to the same location in our list is called as synonyms.
- **Collision:** Collision is the event that occurs when a hashing algorithm produce an address for an insertion key and that address is already occupied.
- **Home Address:** The address produced by the hashing algorithms is known as Home address.
- **Prime Area:** The memory that contains all of the home addresses is known as the prime area.
- **Probe:** Each calculation of an address and test for success is known as probe.
- **Bucket:** A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.
- **Overflow:** An overflow occurs when the home bucket for a new pair (key, element) is full.
- **Open Hashing:** In open hashing, keys are stored in linked lists attached to cell of a hash table.
- **Closed Hashing:** In closed hashing, all keys are stored in linked lists attached to cell of a hash table.

DOWNLOADED FROM BATU-EXAMS.in

- **Load Density/Load Factor:** The loading density or loading factor of a hash table is a = n/(sb)
- s is the number of slots.
- b is the number of buckets.

Issues in Hashing
Following are some basic issues which are consider while hashing:
- Computing the hash function.
- **Collision Resolution**: Algorithm and data structure to handle two keys that hash to the same index.
- **Equality Test**: Method for changing whether two keys are equal.

Properties of Good Hashing Function

Hash functions should have the following properties:
- Fast computation of the hash value (O(1)).
- Hash value should be distributed (nearly) uniformly:
    - Every hash value (cell in the hash table) has equal probability.
    - This should hold even if keys are non-uniformly distributed.
- The goal of a hash function is: ‘disperse’ the key in an apparently random way.
- A hash function must minimize collisions.

Forms of Hashing Data Structure

(1) Linear Open Addressing: It allows any number of records to be stored, because the space is dynamic.

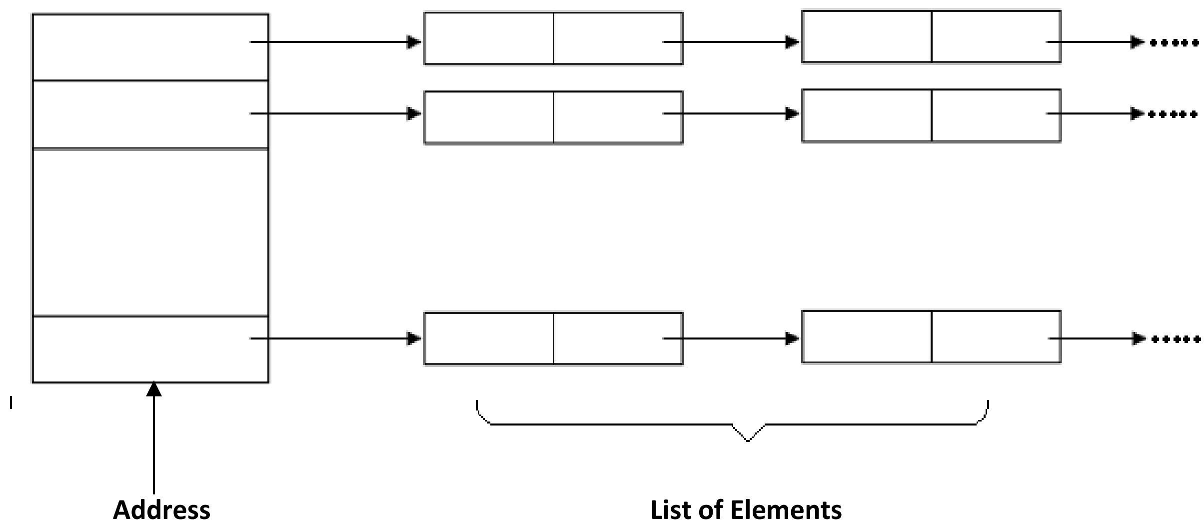


Figure: Linear Open Addressing

DOWNLOADED FROM BATU-EXAMS.in

(2) **Linear Closed Addressing:** It uses a fixed space for storage and hence this limits the size of hash table.

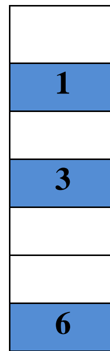


Figure: Linear closed addressing

In this case maximum 7 elements can be stored as array size is only 7 and that is fixed.

Direct Address Tables

- 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.
- We can understand the concept using the following example. We create an array of size equal to maximum value plus one (assuming 0 based index) and then use values as indexes. For example, in following diagram key 21 is used directly as index.

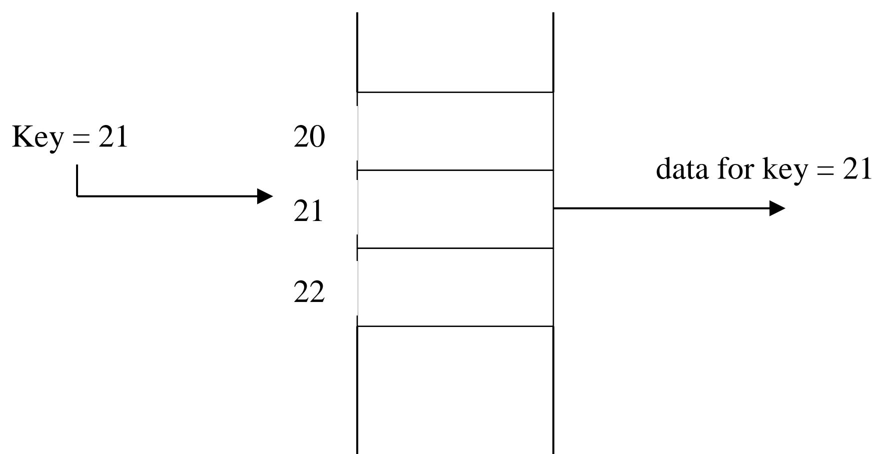


Figure: Direct address table

DOWNLOADED FROM BATU-EXAMS.in

Advantages:

- **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.
- **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.
- **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:

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

Hashing can overcome these limitations of direct address tables.

How to Handle Collisions?
Collisions can be handles like hashing. We can either use chaining or open addressing to handle collisions. The only difference from hashing here is, we do not use hash function to find the index. We rather directly use values as indexes.

Hash Functions

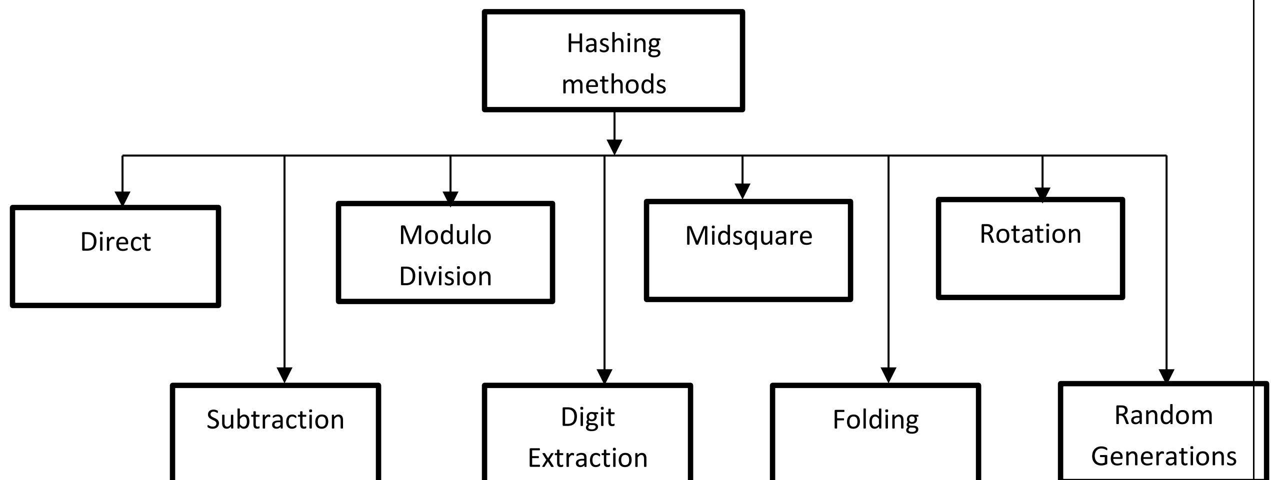


Figure: Basic Hashing Techniques.

(1) Direct Hashing:
In direct hashing, address for a key is generated without any algorithmic manipulation. Therefor the data structure must contain an address for every possible key.
Example:

DOWNLOADED FROM BATU-EXAMS.in

A small organization has 100 employees. Each employee is assigned an employee. Each employee is assigned an employment number between 1 to 100. Hence, we create an array of 100 employee records; the employee umber can be directly used as the address of any individual record.

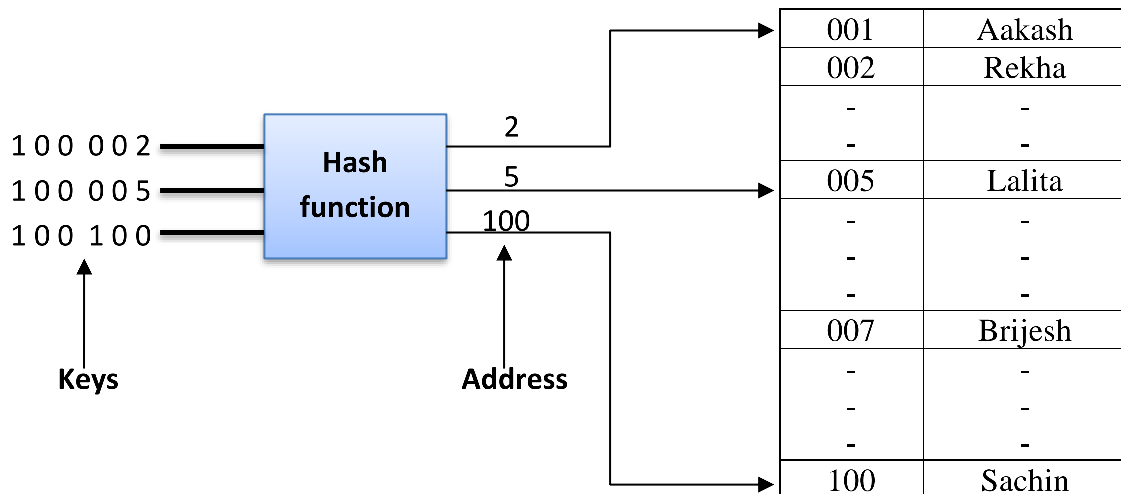


Figure: Hash Function.

(2) Subtraction Method:

Sometimes we have keys that are consecutive but do not start from one. This method is simple and it guarantees no collisions. Limitations is this method can be used for small lists in which the keys map to a densely filled list.

Example: Consider a company has 100 employees, but their employee number start from 1000 up to 1100 consecutively. Then we use very simple hashing function that subtracts 1000 from the key to determine the address.

(3) Modulo-Division Method/Division Remainder:

- This method divides the key by an array or bucket size and uses the reminder plus one for the address.
  ∴Address = (key % list size) + 1
- A list size that is a prime number produces fewer collisions than other list sizes.

Example: Suppose we have 300 employees. The first prime number greater than 300 is 307. We therefore choose 307 as our list size.

∴Employee number – 121267

∴(121267 % 307) + 1 = 2 + 1 = 3

(4) Digit Extraction Method:

Using digit extraction, selected digits are extracted from the key and used as the address.

Example: 379452 -> 394

121267 -> 112

378845 -> 388

160252 -> 102

045128 -> 051

DOWNLOADED FROM BATU-EXAMS.in

(5) Midsquare Method:

Key is squared and the address is selected from the middle of the squared number.

Example: 9452 * 9452 = 89**3403**04

Address is 3403

(6) Folding Method:

There are 2 folding methods:

a) **Fold Shift:** The key value is divided into parts whose size matches the size of the required address. Then the left and right parts are shifted and added with middle part.

Example: Suppose we have 3-digit addresses and key is 123456789

```
    1 2 3
  + 4 5 6
  ----------
(1) 3 6 8
```

Discard (1) address is 368.

b) **Fold Boundary:**

Left and right numbers are folded on a fixed boundary between them and the center number. This results in the two outside values being reversed.

Example: Suppose we have 3 digit addresses and key are 123456789.

```
    3 2 1          -> Reversed digits of 123
  + 4 5 6
  + 9 8 7          -> Reversed digits of 789
-------------
(1) 7 6 4
```

Discard (1). So, address is 764.

(7) Rotation Methods:

Rotation method is incorporated in combination with other hashing methods. It is most useful when keys are assigned serially, such as we often see in employee numbers and part numbers.

Example:

Original Key	Rotation	Rotated Key
6 0 0 1 0 1	6 0 0 1 0	6 0 0 1 0
6 0 0 1 0 2	6 0 0 1 0	6 0 0 1 0
6 0 0 1 0 3	6 0 0 1 0	6 0 0 1 0
6 0 0 1 0 4	6 0 0 1 0	6 0 0 1 0

(8) Pseudorandom Method:

The key used as the seed in pseudorandom number generator and the random number then scaled into the possible address range using modulo division. Common random generator is Y = ax + c.

Example: Consider a = 17 and c = 7. Also consider list size is 307. Key is 121267.

∴ y = ((17 * 121267) + 7) % 307 + 1

DOWNLOADED FROM BATU-EXAMS.in

y = (2061539 + 7) % 307 + 1
y = (2061546 % 307) + 1
y = 41 + 1
y = 42
∴ Address is 42.

(9) Multiplicative Hash Function:
Example:

H(k) = floor (p * fractional part of key * A)

Where Example

P = constant integer
A = Constant real number
k = 107, p = 50,
A = 0.61803398987

H(k) = floor (50 * (107 * 0.61803398987))
= floor (3306.4818458045)
H(k) = 3306

OPEN Addressing

In Open addressing when a collision occurs, the home area addresses are searched for an unoccupied element where the new data can be placed.

(1) Linear Probe:

(a) Linear Probing without Chaining:

When collision occurs, we resolve the collision by finding the next empty cell.

Example: Keys 3, 33, 42, 63, 89, 45, 93

Hash function -> key % 10

	Empty	After 3	After 33	After 42	After 63	After 89	After 45	After 93
0	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-
2	-	-	-	**42**	42	42	42	42
3	-	**3**	3	3	3	3	3	3
4	-	-	**33**	33	33	33	33	33
5	-	-	-	-	**63**	63	63	63
6	-	-	-	-	-	-	**45**	45
7	-	-	-	-	-	-	-	**93**
8	-	-	-	-	-	-	-	-
9	-	-	-	-	-	**89**	89	89

(b) Linear probing with Chaining (without replacement):

Excessive collisions can be dealt by means of chaining. All the records mapped to same location are stored in a chain.

Example: Keys – 3, 33, 42, 63, 89, 45, 93

Hash function => key % 10

DOWNLOADED FROM BATU-EXAMS.in

Index	Key	Chain
0	-	-1
1	-	-1
2	-	-1
3	-	-1
4	-	-1
5	-	-1
6	-	-1
7	-	-1
8	-	-1
9	-	-1

-1 shows there is no chaining yet.

Index	Key	Chain
0	-	-1
1	-	-1
2	42	-1
3	3	4
4	33	5
5	63	6
6	45	-1
7	93	-1
8	-	-1
9	89	-1

Here 3, 33, 63 and 93 supposed to be mapped at location 3. Hence all these are chained by index number at chain column.

Index of 33
Index 0f 63
Index of 93

PERFECT Hashing

- A Perfect hash function for a set S is a hash function that maps distinct elements into S to a set of integers, with no collisions, in mathematical terms, it is an injective function.
- Perfect hash function may be to implement a lookup table with constant worst-case access time. A perfect hash function has many of the same applications as other hash functions, but with the advantage that no collision resolution has to be implemented.
- A perfect hash function for a specific set S that can be evaluated in constant time, and with values in a small range, can be found by a randomized algorithm in a number of operations that is proportional to the size of S. The original construction of Fredman, Kolmos & Szemeredi (1984) uses a two-level scheme to map a set S of n elements to a range of O(n) indices, and then map each index to a range of hash values.
- The first level of their construction chooses a large prime p (larger than the size of the universe from which S is drawn), and a parameter k, and maps each element x of S to the index.

DOWNLOADED FROM BATU-EXAMS.in

- If k is chosen randomly, this step is likely to have collisions, but the number of elements ni that are simultaneously mapped to the same index i is likely to be small.
- The second level of their construction assigns disjoint ranges of O(ni2) integers to each index i. IT uses a second set of linear modular functions, one for each index I, to map each member x of S into the range associated with g(x).
- As Fredman, Kolmos, & Szemeredi (1984) show, there exists choice of the parameter k such that the sum of the lengths of the ranges for the n different values of g(x) id O(n).
- Additionally, for each value of g(x), there exits a linear modular function that maps the corresponding subset of S into the range associated with that value. Both k, and the second level functions for each value of g(x), can be found in polynomial time by choosing values randomly until finding one that works.
- The hash function itself requires storage space O(n) to store k, p, and all of the second-level linear modular functions. Computing the hash value of a given key x may be performed in constant time by computing g(x), looking up the second-level function associated with g(x), and applying this function to x.
- A modified version of this two-level scheme with large number of values at the top level can be used to construct a perfect hash function that maps S into a smaller range of length n + o(n).

BATU-EXAM

Made by batuexams.com

at MET Bhujbal Knowledege City

The PDF notes on this website are the copyrighted property of batuexams.com.

All rights reserved.