MODULE 5

GRAPHS, HASHING, SORTING, FILES

A graph is an abstract data structure that is used to implement the mathematical concept of graphs. It is basically a collection of vertices (also called nodes) and edges that connect these vertices. A graph is often viewed as a generalization of the tree structure, where instead of having a purely parent-to-child relationship between tree nodes, any kind of complex relationship can exist.

Graphs - Terminology and Representation

Definitions: Graph, Vertices, Edges

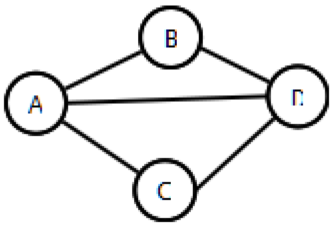
- Define a graph *G = (V, E)* by defining a pair of sets:
    1. V = a set of **vertices**
    2. E = a set of **edges**
- Edges:
    - Each edge is defined by a pair of vertices
    - An edge **connects** the vertices that define it

- Vertices:

    - Vertices also called **nodes**
    - Denote vertices with labels

  Representation:
    - Represent vertices with circles, perhaps containing a label
    - Represent edges with lines between circles

  Example:
    - V = {A,B,C,D}
    - E = {(A,B),(A,C),(A,D),(B,D),(C,D)}



  Many algorithms use a graph representation to represent data or the problem to be solved

- Examples of Graph applications:
    - Cities with distances between
    - Roads with distances between intersection points
    - Course prerequisites
    - Network and shortest routes
    - Social networks
    - Electric circuits, projects planning and many more...

Graph Classifications

- There are several common kinds of graphs
    - Weighted or unweighted
    - Directed or undirected

- Cyclic or acyclic
- Multigraphs

Kinds of Graphs: Weighted and Unweighted

- Graphs can be classified by whether or not their edges have **weights**
- **Weighted graph**: edges have a weight
  - Weight typically shows cost of traversing
  - Example: weights are distances between cities
- **Unweighted graph**: edges have no weight
  - Edges simply show connections
  - Example: course prerequisites

Kinds of Graphs: Directed and Undirected

- Graphs can be classified by whether or their edges are have direction
  - **Undirected Graphs**: each edge can be traversed in **either direction**
  - **Directed Graphs**: each edge can be traversed **only in a specified direction**

Undirected Graphs

- **Undirected Graph**: no implied direction on edge between nodes
  - The example from above is an undirected graph

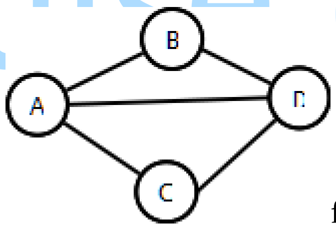


fig 1

  - In diagrams, edges have no direction (ie there are no arrows)
  - Can traverse edges in either directions
- In an undirected graph, an edge is an **unordered** pair
  - Actually, an edge is a set of 2 nodes, but for simplicity we write it with parenthesis
    - For example, we write (A, B) instead of {A, B}
    - Thus, (A,B) = (B,A), etc
    - If (A,B) ∈ E then (B,A) ∈ E

Directed Graphs

- **Digraph**: A graph whose edges are directed (ie have a direction)
  - Edge drawn as arrow
  - Edge can only be traversed in direction of arrow
  - Example: E = {(A,B), (A,C), (A,D), (B,C), (D,C)}

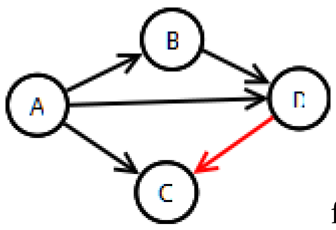


fig 2

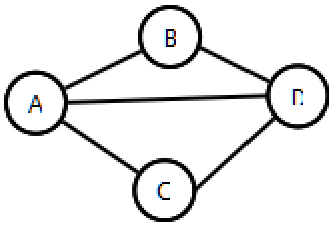
- In a digraph, an edge is an **ordered** pair
    - Thus: (u,v) and (v,u) are not the same edge
    - In the example, (D,C) ∈ E, (C,D) ∉ E

Degree of a Node

- The **degree** of a node is the number of edges incident on it.
- In the example above: (fig 1)
    - Degree 2: B and C
    - Degree 3: A and D
- A and D have **odd degree**, and B and C have **even degree**
- Can also define **in-degree** and **out-degree**
    - In-degree: Number of edges pointing **to** a node
    - Out-degree: Number of edges pointing **from** a node

Graphs: Terminology Involving Paths

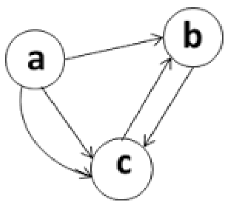
- **Path**: sequence of vertices in which each pair of successive vertices is connected by an edge
- **Cycle**: a path that starts and ends on the same vertex
- **Simple path**: a path that does not cross itself
    - That is, no vertex is repeated (except first and last)
- Simple paths cannot contain cycles
- **Length** of a path: Number of edges in the path
- Examples



Cyclic and Acyclic Graphs

- A **Cyclic** graph contains cycles
    - Example: roads (normally)
- An **acyclic** graph contains no cycles
    - Example: Course prerequisites

Multigraph: A graph with self loops and parallel edges is called a multigraph.



Connected and Unconnected Graphs and Connected Components

- An *undirected* graph is **connected** if every pair of vertices has a path between it
    - Otherwise it is unconnected

- A *directed* graph is **strongly connected** if every pair of vertices has a path between them, in **both directions**

Data Structures for Representing Graphs

- Two common data structures for representing graphs:
  - Adjacency lists
  - Adjacency matrix

Adjacency List Representation

An adjacency list is a way in which graphs can be represented in the computer's memory. This structure consists of a list of all nodes in G. Furthermore, every node is in turn linked to its own list that contains the names of all other nodes that are adjacent to it. The key advantages of using an adjacency list are:

- It is easy to follow and clearly shows the adjacent nodes of a particular node.
- It is often used for storing graphs that have a small-to-moderate number of edges. That is, an adjacency list is preferred for representing sparse graphs in the computer's memory; otherwise, an adjacency matrix is a good choice.
- Adding new nodes in G is easy and straightforward when G is represented using an adjacency list. Adding new nodes in an adjacency matrix is a difficult task, as the size of the matrix needs to be changed and existing nodes may have to be reordered. Each node has a list of adjacent nodes

Example (undirected graph): **(fig 1)**

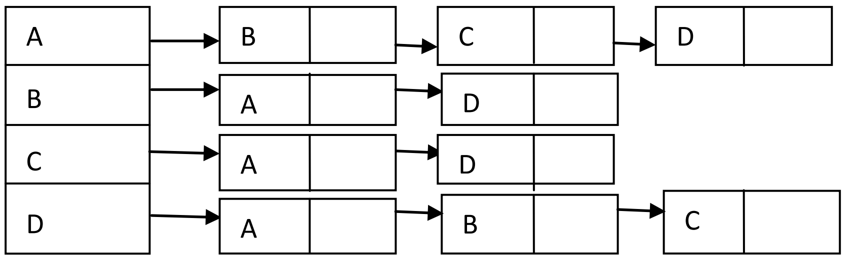


Fig (1) adjacency lsit for the graph of fig3

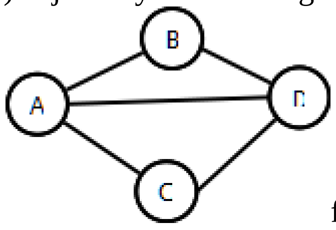


fig 3

  - Example (directed graph):
  - A: B, C, D
  - B: D
  - C: Nil
  - D: C

Adjacency Matrix Representation

An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them. In a directed graph G, if node v is adjacent to node u, then there is definitely an edge from u to v. That is, if v is adjacent to u, we can get from u to v by traversing one edge. For any graph G having n nodes, the adjacency matrix will have the dimension of n * n. In an adjacency matrix, the rows and columns are labelled by graph vertices. An entry aij in the adjacency matrix will contain 1, if vertices vi and vj are adjacent to each other. However, if the nodes are not adjacent, aij will be set to zero. It. Since an adjacency matrix contains only 0s and 1s, it is called a bit matrix or a Boolean matrix. The entries in the matrix depend on the ordering of the nodes in G. Therefore, a change in the order of nodes will result in a different adjacency matrix.

Aij = 1 if there is an edge from Vi to Vj

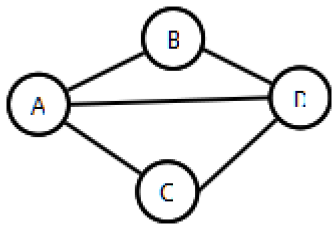
0 otherwise

Adjacency Matrix: 2D array containing weights on edges

- Row for each vertex
- Column for each vertex
- Entries contain weight of edge from row vertex to column vertex
- Entries contain ∞ if no edge from row vertex to column vertex
- Entries contain 0 on diagonal (if self edges not allowed)

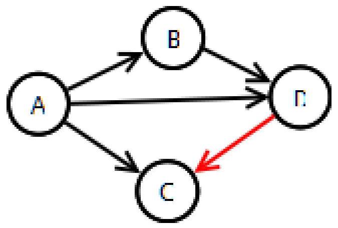
- Example undirected graph (assume self-edges not allowed):

$$\begin{array}{c|cccc} & A & B & C & D \\ A & 0 & 1 & 1 & 1 \\ B & 1 & 0 & \infty & 1 \\ C & 1 & \infty & 0 & 1 \\ D & 1 & 1 & 1 & 0 \end{array}$$



- Example directed graph (assume self-edges allowed):

$$\begin{array}{c|cccc} & A & B & C & D \\ A & \infty & 1 & 1 & 1 \\ B & \infty & \infty & \infty & 1 \\ C & \infty & \infty & \infty & \infty \\ D & \infty & \infty & 1 & \infty \end{array}$$



Disadv:Adjacency matrix representation is easy to represent and feasible as long as the graph is small and connected. For a large graph ,whose matrix is sparse, adjacency matrix representation wastes a lot of memory. Hence list representation is preferred over matrix representation.

Graph traversal algorithms

Traversing a graph, is the method of examining the nodes and edges of the graph. There are two standard methods of graph traversal. These two methods are:

1. Breadth-first search 2. Depth-first search

While breadth-first search uses a queue as an auxiliary data structure to store nodes for further processing, the depth-first search scheme uses a stack.

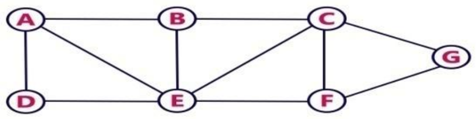
Breadth-first search algorithm

Breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighbouring nodes. Then for each of those nearest nodes, the algorithm explores their unexplored neighbour nodes, and so on, until it finds the goal. That is, we start examining the node A and then all the neighbours of A are examined. In the next step, we examine the neighbours of neighbours of A, so on and so forth. This means that we need to track the neighbours of the node and guarantee that every node in the graph is processed and no node is processed more than once. This is accomplished by using a queue that will hold the nodes that are waiting for further processing.

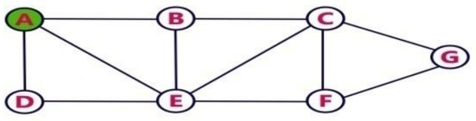
Algorithm for BFS traversal

- **Step 1:** Define a Queue of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3:** Visit all the **adjacent** vertices of the verex which is at front of the Queue which is not visited and insert them into the Queue.
- **Step 4:** When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.
- **Step 5:** Repeat step 3 and 4 until queue becomes empty.
- **Step 6:** When queue becomes Empty, then the enqueue or dequeue order gives the BFS traversal order.

Consider the following example graph to perform BFS traversal



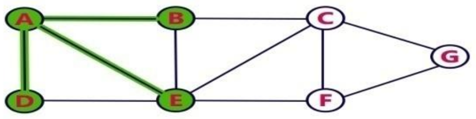
Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



Queue

A						

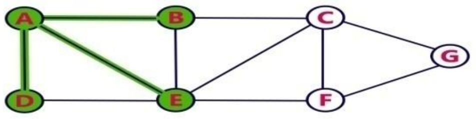
Step 2:
- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



Queue

	D	E	B			

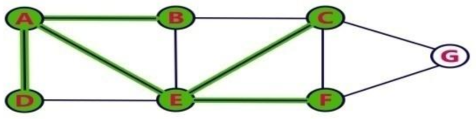
Step 3:
- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



Queue

		E	B			

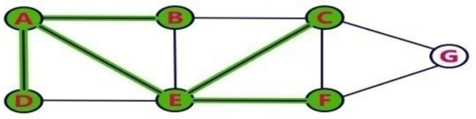
Step 4:
- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



Queue

			B	C	F	

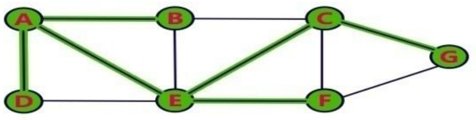
Step 5:
- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.



Queue

				C	F	

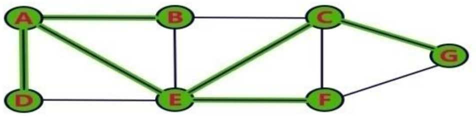
Step 6:
- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



Queue

					F	G

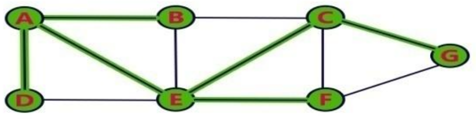
Step 7:
- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.



Queue

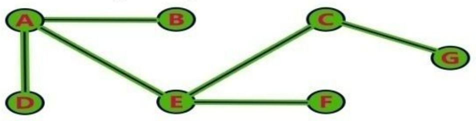
						G

Step 8:
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



Depth-first Search Algorithm

Depth-first search begins at a starting node A which becomes the current node. Then, it examines each node N along a path P which begins at A. That is, we process a neighbour of A, then a neighbour of neighbour of A, and so on.

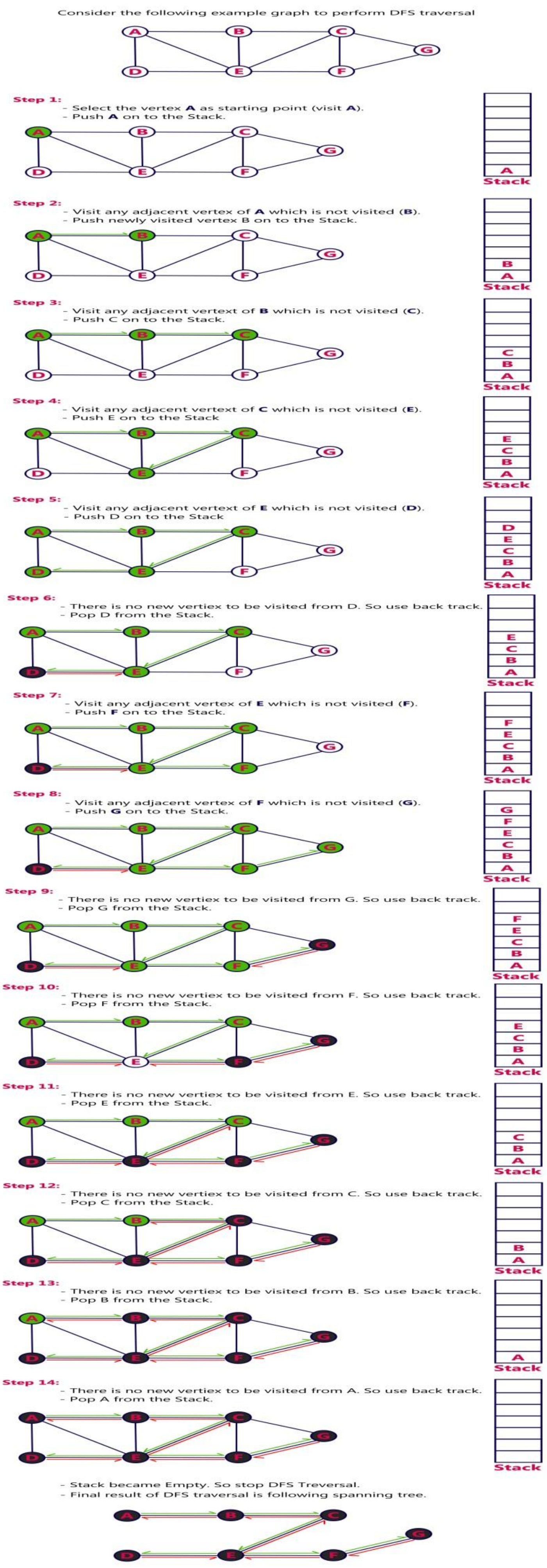
During the execution of the algorithm, if we reach a path that has a node N that has already been processed, then we backtrack to the current node. Otherwise, the unvisited (unprocessed) node becomes the current node. The algorithm proceeds like this until we reach a dead-end (end of path P). On reaching the deadend, we backtrack to find another path P. The algorithm terminates when backtracking leads back to the starting node A.

In this algorithm, edges that lead to a new vertex are called discovery edges and edges that lead to an already visited vertex are called back edges. Observe that this algorithm is similar to the in-order traversal of a binary tree. Its implementation is similar to that of the breadth-first search algorithm but here we use a stack instead of a queue.

We use the following steps to implement DFS traversal...

- **Step 1:** Define a Stack of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3:** Visit any one of the **adjacent** vertex of the verex which is at top of the stack which is not visited and push it on to the stack.
- **Step 4:** Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
- **Step 5:** When there is no new vertex to be visit then use **back tracking** and pop one vertex from the stack.
- **Step 6:** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7:** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

PTO



Consider the following example graph to perform DFS traversal

Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

Step 2:
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

Step 3:
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

Step 4:
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

Step 5:
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

Step 6:
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

Step 7:
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

Step 8:
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

Step 9:
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

Step 10:
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

Step 11:
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

Step 12:
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

Step 13:
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

Step 14:
- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.

- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.

Applications OF graphs

- Graphs are constructed for various types of applications such as:
- In circuit networks where points of connection are drawn as vertices and component wires become the edges of the graph.
- In transport networks where stations are drawn as vertices and routes become the edges of the graph.
- In maps that draw cities/states/regions as vertices and adjacency relations as edges.
- In program flow analysis where procedures or modules are treated as vertices and calls to these procedures are drawn as edges of the graph.
- Once we have a graph of a particular concept, they can be easily used for finding shortest paths, project planning, etc.
- In flowcharts or control-flow graphs, the statements and conditions in a program are represented as nodes and the flow of control is represented by the edges.
- In state transition diagrams, the nodes are used to represent states and the edges represent legal moves from one state to the other.
- Graphs are also used to draw activity network diagrams. These diagrams are extensively used as a project management tool to represent the interdependent relationships between groups, steps, and tasks that have a significant impact on the project.

Introduction to sorting

Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending. That is, if A is an array, then the elements of A are arranged in a sorted order (ascending order) in such a way that A[0] < A[1] < A[2] < ...... < A[N]. For example, if we have an array that is declared and initialized as int A[] = {21, 34, 11, 9, 1, 0, 22}; Then the sorted array (ascending order) can be given as: A[] = {0, 1, 9, 11, 21, 22, 34; A sorting algorithm is defined as an algorithm that puts the elements of a list in a certain order, which can be either numerical order, lexicographical order, or any user-defined order

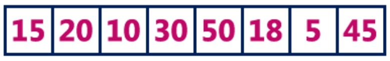
Insertion Sort

Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time. We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards while playing bridge. The main idea behind insertion sort is that it inserts each item into its proper place in the final list. To save memory, most implementations of the insertion sort algorithm work by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place. Insertion sort is less efficient as compared to other more advanced algorithms such as quick sort, heap sort, and merge sort.

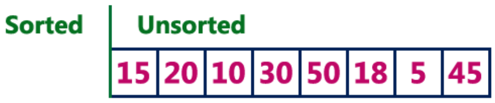
Technique:

- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially, the element with index 0 (assuming LB = 0) is in the sorted set. Rest of the elements are in the unsorted set.
- The first element of the unsorted partition has array index 1 (if LB = 0).
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

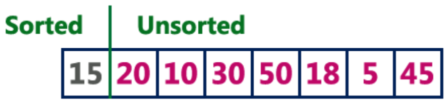
Consider the following unsorted list of elements...



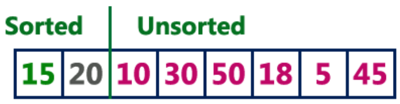
Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...



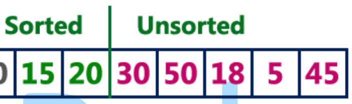
Move the first element 15 from unsorted portion to sorted portion of the list.



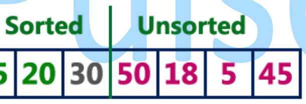
To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position



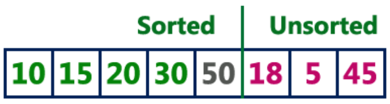
To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.



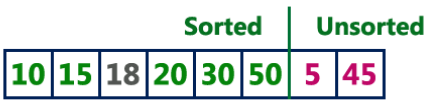
To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.



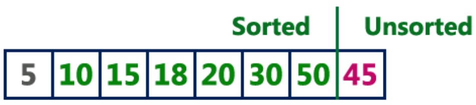
To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.



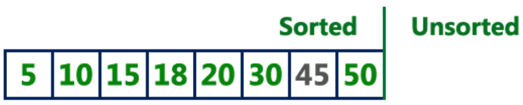
To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.



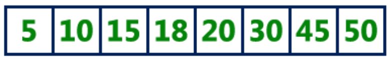
To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.



To move element 45 from unsorted to sorted portion, Compare 45 with 50 and 30. Since 45 is larger than 30, move 50 one position to the right in the list and insert 45 after 30 in the sorted list.



Unsorted portion of the list has became empty. So we stop the process. And the final sorted list of elements is as follows...



```
ALGORITHM INSERTION-SORT (ARR, N)
Step 1: Repeat Steps 2 to 5 for K = 1 toN–1
Step 2: SET TEMP = ARR[K]
 Step 3: SET J = K - 1
Step 4: Repeat while TEMP <= ARR[J]
            SET ARR[J + 1] = ARR[J]
             SETJ=J-1 [END OF INNER LOOP]
Step 5: SET ARR[J + 1] = TEMP [END OF LOOP]
Step 6: EXIT
```

To insert an element A[K] in a sorted list A[0], A[1], ..., A[K–1], we need to compare A[K] with A[K–1], then with A[K–2], A[K–3], and so on until we meet an element A[J] such that A[J] <= A[K]. In order to insert A[K] in its correct position, we need to move elements A[K–1], A[K–2], ..., A[J] by one position and then A[K] is inserted at the (J+1)th location..

Radix Sort

Radix sort is a linear sorting algorithm for integers and uses the concept of sorting names in alphabetical order. When we have a list of sorted names, the radix is 26 (or 26 buckets) because there are 26 letters in the English alphabet. So radix sort is also known as bucket sort. Observe that words are first sorted according to the first letter o/f the name. That is, 26 classes are used to arrange the names, where the first class stores the names that begin with A, the second class contains the names with B, and so on. During the second pass, names are grouped according to the second letter. After the second pass, names are sorted on the first two letters. This processis continued till the nth pass, where n is the length of the name with maximum number of letters. After every pass, all the names are collected in order of buckets. That is, first pick up the names in the first bucket that contains the names beginning with A. In the second pass, collect the name from the second bucket, and so on. When radix sort is used on integers, sorting is done on each of the digits in the number. The sorting procedure proceeds by sorting the least significant to the most significant digit. While sorting the numbers, we have ten buckets, each for one digit (0, 1, 2, …, 9) and the number of passes will depend on the length of the number having maximum number of digits.

```
Algorithm for RadixSort (ARR, N)
Step 1: Find the largest number in ARR as LARGE
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE
Step 3: SET PASS = 0
Step 4: Repeat Step 5 while PASS <= NOP-1
```

Step 5: SET= I=0 and INITIALIZE buckets

Step 6: Repeat Steps 7 to 9 while I<N-1

Step 7: SET DIGIT = digit at PASSth place in A[I]

Step 8: Add A[I] to the bucket numbered DIGIT

Step 9: INCREMENT bucket count for bucket numbered DIGIT [END OF LOOP]

Step 10: Collect the numbers in the bucket [END OF LOOP]

Step 11: END

Sort the numbers given below using radix sort. 345, 654, 924, 123, 567, 472, 555, 808, 911

In the first pass, the numbers are sorted according to the digit at ones place.

Bin 0	Bin 1	Bin2	Bin 3	Bin4	Bin 5	Bin 6	Bin7	Bin8	Bin 9
	911	472	123	654	345		67	808	
				924	555				

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. 911,472,123,654,924,345,555,67,808

In the second pass, the numbers are sorted according to the digit at the tens place

Bin 0	Bin 1	Bin2	Bin 3	Bin4	Bin 5	Bin 6	Bin7	Bin8	Bin 9
808	911	123		345	654	67	472		
		924			555				

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. 808,911,123,924,345,654,555,67,472

In the third pass, the numbers are sorted according to the digit at the hundreds place

Bin 0	Bin 1	Bin2	Bin 3	Bin4	Bin 5	Bin 6	Bin7	Bin8	Bin 9
67	123		345	472	555	654		808	911
									924

The sorted order is as above :67,123,345,472,555,654,808,911,924

Address Calculation Sort

- Is based on hashing.
- It is a distribution based sorting technique.
- A hash function is applied on each element of the unsorted list
- The address generated is taken as the position where the element is stored in a hash table organised as a array of pointers to elements.
- The elements that map to the same location are stored as a linked list of elements.

- If more than one element maps to the same location, they are inserted into the linked list in order using any sorting technique.
- After all elements have been hashed, the linked lists are concatenated to form the sorted list.
- The hash function should satisfy the property that if a key x is less than y,then f(x)<f(y).This is called order preserving property.

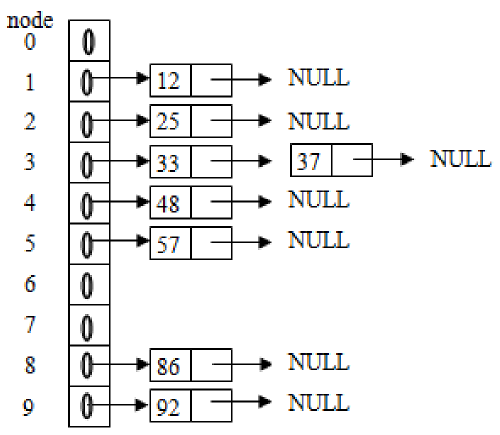
Example:

Sort 25 57 48 37 12 92 86 33 using address calculation sort

Let us create 10 sub lists. Initially each of these sublist is empty. An array of pointer f(10) is declared, where f(i) refers to the first element in the file, whose first digit is i. The number is passed to hash function, which returns its last digit (ten's place digit), which is placed at that position only, in the array of pointers.

num= 25	–	f(25) gives 2
57	–	f(57) gives 5
48	–	f(48) gives 4
37	–	f(37) gives 3
12	–	f(12) gives 1
92	–	f(92) gives 9
86	–	f(86) gives 8
33	–	f(33) gives 3 which is repeated.

Thus it is inserted in 3rd sublist (4th) only, but must be checked with the existing elements for its proper position in this sublist.



Hashing

Why Hashing?

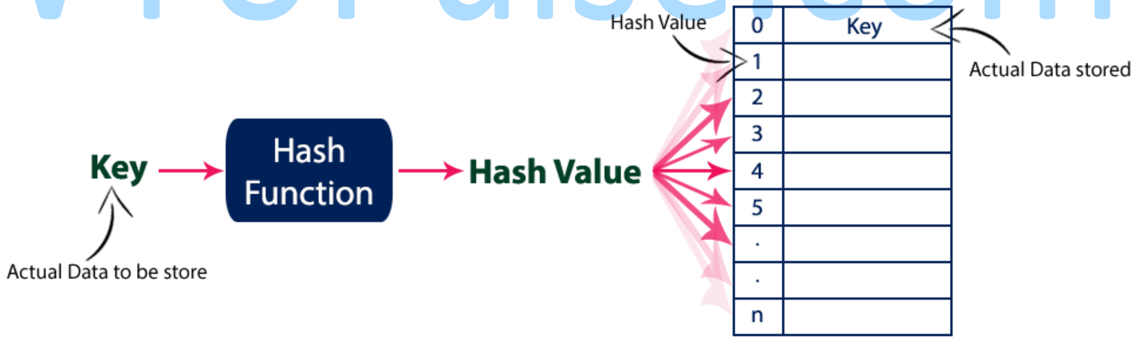
Internet has grown to millions of users generating terabytes of content every day. According to internet data tracking services, the amount of content on the internet doubles every six months. With this kind of growth, it is impossible to find anything in the internet, unless we develop new data structures and algorithms for storing and accessing data. So what is wrong with traditional data structures like Arrays and Linked Lists? Suppose we have a very large data set stored in an array. The amount of time required to look up an element in the array is either O(log n) or O(n) based on whether the array is sorted or not. If the array is sorted then a technique such as binary search can be used to search the array. Otherwise, the array must be searched linearly. Either case may not be desirable if we need to process a very large data set. Therefore we discuss a new technique called hashing that allows us to update and retrieve any entry in constant time O(1). The constant time or O(1) performance means, the amount of time to perform the operation does not depend on data size n.

The Map Data Structure(Hash Map)(Hash function)

In a mathematical sense, a map is a relation between two sets. We can define Map M as a set of pairs, where each pair is of the form (key, value), where for given a key, we can find a value using some kind of a "function" that maps keys to values. The key for a given object can be calculated using a function called a **hash function**. In its simplest form, we can think of an array as a Map where key is the index and value is the value at that index. For example, given an array A, if i is the key, then we can find the value by simply looking up A[i]. The idea of a hash table is more generalized and can be described as follows.

The concept of a hash table is a generalized idea of an array where key does not have to be an integer. We can have a name as a key, or for that matter any object as the key. The trick is to find a hash function to compute an index so that an object can be stored at a specific location in a table such that it can easily be found.

STATIC HASHING



This kind of hashing is called **static hashing** since the size of the hash table is fixed.(an array)

Example:

Suppose we have a set of strings {"abc", "def", "ghi"} that we'd like to store in a table. Our objective here is to find or update them quickly from a table, actually in O(1). We are not concerned about ordering them or maintaining any order at all. Let us think of a simple schema to do this. Suppose we assign "a" = 1, "b"=2, … etc to all alphabetical characters. We can then simply compute a number for each of the strings by using the sum of the characters as follows.

"abc" = 1 + 2 + 3=6, "def" = 4 + 5 + 6=15 , "ghi" = 7 + 8 + 9=24

If we assume that we have a table of size 5 to store these strings, we can compute the location of the string by taking the sum mod 5. So we will then store "abc" in 6 mod 5 = 1, "def" in 15 mod 5 = 0, and "ghi" in 24 mod 5 = 4 in locations 1, 0 and 4 as follows.

0	1	2	3	4
def	abc			ghi

Now the idea is that if we are given a string, we can immediately compute the location using a simple hash function, which is sum of the characters mod Table size. Using this hash value, we can search for the string.

Problem with Hashing -collision

The method discussed above seems too good to be true as we begin to think more about the hash function. First of all, the hash function we used, that is the sum of the letters, is a bad one. In case we have permutations of the same letters, "abc", "bac" etc in the set, we will end up with the same value for the sum and hence the key. In this case, the strings would hash into the same location, creating what we call a "collision". This is obviously not a good thing. Secondly, we need to find a good table size, preferably a prime number so that even if the sums are different, then collisions can be avoided, when we take mod of the sum to find the location. So we ask two questions.

Question 1: How do we pick a good hash function?

Question 2: How do we deal with collisions?

The problem of storing and retrieving data in O(1) time comes down to answering the above questions. Picking a "good" hash function is key to successfully implementing a hash table. What we mean by "good" is that the **function must be easy to compute and avoid collisions as much as possible.** If the function is hard to compute, then we lose the advantage gained for lookups in O(1). Even if we pick a very good hash function, we still will have to deal with "some" collisions.

The process where two records can hash into the same location is called collision. We can deal with collisions using many strategies, such as linear probing (looking for the next available location i+1, i+2, etc. from the hashed value i), quadratic probing (same as linear probing, except we look for available positions i+1 , i + 4, i + 9, etc from the hashed value i and separate chaining, the process of creating a linked list of values if they hashed into the same location.This is called collision resolution.

Popular hash functions

Hash functions that use numeric keys are very popular.. However, there can be cases in real-world applications where we can have alphanumeric keys rather than simple numeric keys. In such cases, the ASCII value of the character can be used to transform it into its equivalent numeric key. Once this transformation is done, any hash function can be applied to generate the hash value.

Division Method

It is the most simple method of hashing an integer x. This method divides x by M and then uses the remainder obtained. In this case, the hash function can be given as

h(x) = x mod M

The division method is quite good for just about any value of M and since it requires only a single division operation, the method works very fast. However, extra care should be taken to select a suitable value for M. Generally, it is best to choose M to be a prime number because making M a prime number increases the likelihood that the keys are mapped with a uniformity in the output range of values.

A potential drawback of the division method is that while using this method, consecutive keys map to consecutive hash values. On one hand, this is good as it ensures that consecutive keys do not collide, but on the other, it also means that consecutive array locations will be occupied. This may lead to degradation in performance.

Example :

Calculate the hash values of keys 1234 and 5462. Solution Setting M = 97, hash values can be calculated as:

h(1234) = 1234 % 97 = 70

h(5642) = 5642 % 97 = 16

Mid-Square Method

The mid-square method is a good hash function which works in two steps:

Step 1: Square the value of the key. That is, find k^2.

Step 2: Extract the middle r digits of the result obtained in Step 1.

The algorithm works well because most or all digits of the key value contribute to the result. This is because all the digits in the original key value contribute to produce the middle digits of the squared value. Therefore, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value. In the mid-square method, the same r digits must be chosen from all the keys. Therefore, the hash function can be given as:

h(k) = s where s is obtained by selecting r digits from k^2.

Example Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations. Solution Note that the hash table has 100 memory locations whose indices vary from 0 to 99.

This means that only two digits are needed to map the key to a location in the hash table, so r = 2.

When k = 1234, k^2 = 1522756, h (1234) = 27

When k = 5642, k^2 = 31832164, h (5642) = 21

Observe that the 3rd and 4th digits starting from the right are chosen.

Folding Method

The folding method works in the following two steps:

Step 1: Divide the key value into a number of parts. That is, divide k into parts k1, k2, ..., kn, where each part has the same number of digits except the last part which may have lesser digits than the other parts.

Step 2: Add the individual parts. That is, obtain the sum of k1 + k2 + ... + kn. The hash value is produced by ignoring the last carry, if any. Note that the number of digits in each part of the key will vary depending upon the size of the hash table. .

Example Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678, 321, and 34567. Solution Since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits. The hash values can be obtained as shown below:

key	5678	321	34567
Parts	56 and 78	32 and 1	34, 56 and 7
Sum	134	33	97
Hash value	34 (ignore the last carry)	33	97

Collision Resolution Strategies

1. Open Addressing/Closed Hashing
2. Chaining

Once a collision takes place, open addressing or closed hashing computes new positions using a probe sequence and the next record is stored in that position

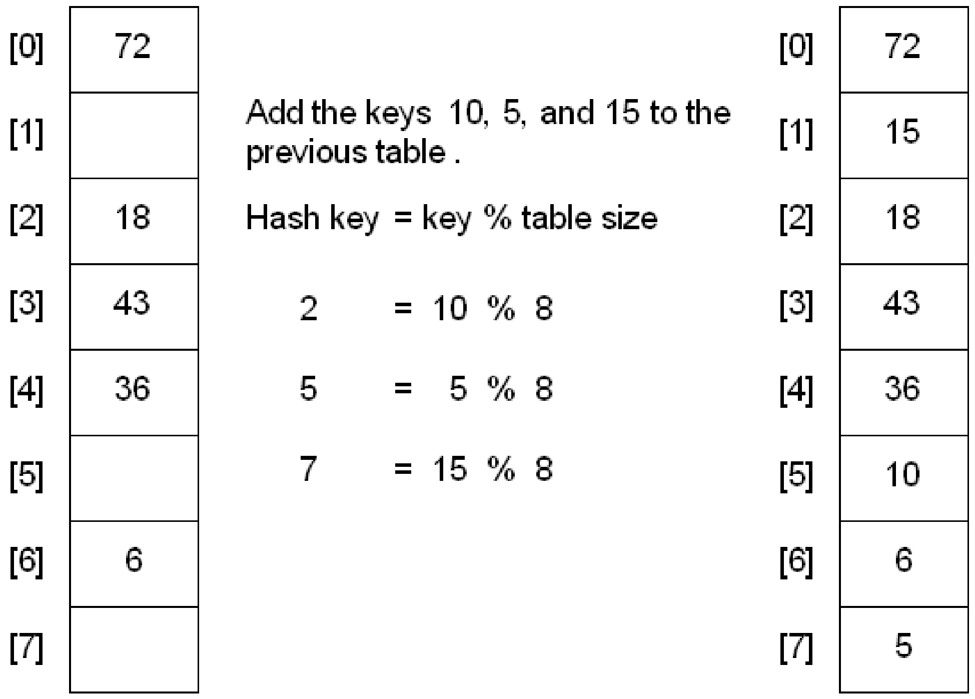
The process of examining memory locations in the hash table is called probing. Open addressing technique can be implemented using linear probing, quadratic probing, double hashing.

Linear Probing

When using a linear probe to resolve collision, the item will be stored in the next available slot in the table, assuming that the table is not already full.

This is implemented via a linear search for an empty slot, from the point of collision. If the physical end of table is reached during the linear search, the search will wrap around to the beginning of the table and continue from there.If an empty slot is not found before reaching the point of collision, the table is full.

If h is the point of collision, probe through h+1,h+2,h+3.................h+i. till an empty slot is found



[0]	72
[1]	
[2]	18
[3]	43
[4]	36
[5]	
[6]	6
[7]	

Add the keys 10, 5, and 15 to the previous table .

Hash key = key % table size

```
2    =  10  %  8
5    =   5  %  8
7    =  15  %  8
```

[0]	72
[1]	15
[2]	18
[3]	43
[4]	36
[5]	10
[6]	6
[7]	5

Searching a Value using Linear Probing

The procedure for searching a value in a hash table is same as for storing a value in a hash table. While searching for a value in a hash table, the array index is re-computed and the key of the element stored at that location is compared with the value that has to be searched. If a match is found, then the search operation is successful. The search time in this case is given as O(1). If the key does not match, then the search function begins a sequential search of the array that continues until:

- the value is found,or
- the search function encounters a vacant location in the array, indicating that the value is not present, or
- the search function terminates because it reaches the end of the table and the value is not present.

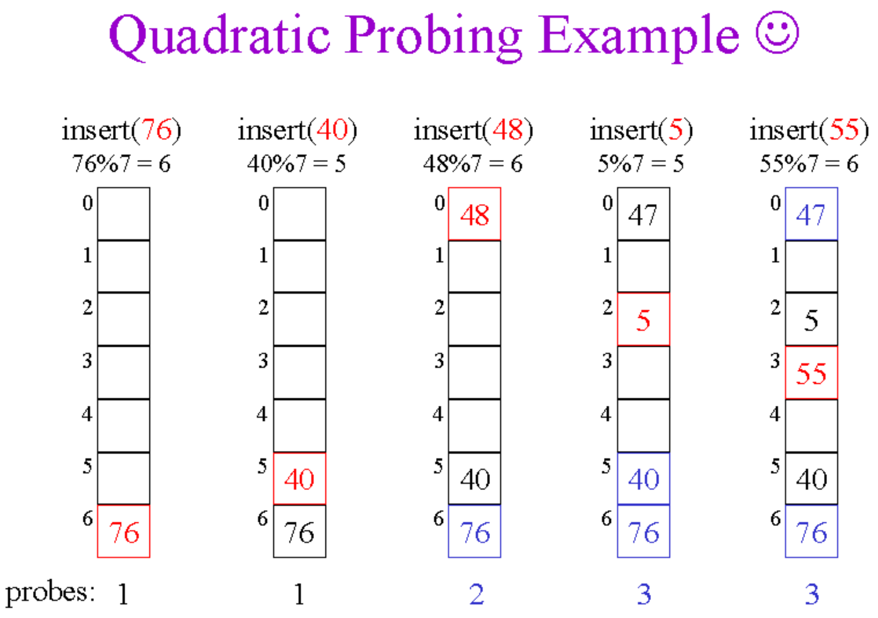
Probe Sequence ::(h+i)%Table size

Disadvantage:

As the hash table fills, clusters of consecutive cells are formed and the time required for a search increases with the size of the cluster. It is possible for blocks of data to form when collisions are resolved. This means that any key that hashes into the cluster will require several attempts to resolve the collision. More the number of collisions, higher the probes that are required to find a free location and lesser is the performance. This phenomenon is called clustering. To avoid clustering, other techniques such as quadratic probing and double hashing are used.

Quadratic Probing

A variation of the linear probing idea is called **quadratic probing**. Instead of using a constant "skip" value, if the first hash value that has resulted in collision is *h*, the successive values which are probed are *h*+1, *h*+4, *h*+9, *h*+16, and so on. In other words, quadratic probing uses a skip consisting of successive perfect squares.



Probe sequence :h,h+1^2,h=2^2,h=3^2..............................h+i^2

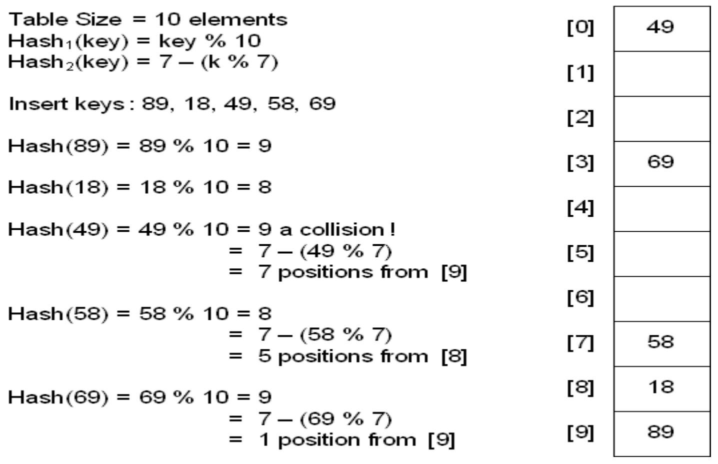
H(k)=(h+i$^{2)}$%Tablesize

<u>Double Hashing</u>

In double hashing, we use two hash functions rather than a single function. Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions form the point of collision to insert.There are a couple of requirements for the second function:

- it must never evaluate to 0
- must make sure that all cells can be probed

A popular second hash function is: Hash$_2$(key) = R - (key % R) where R is a prime number that is smaller than the size of the table.But any independent hash function may also be used.



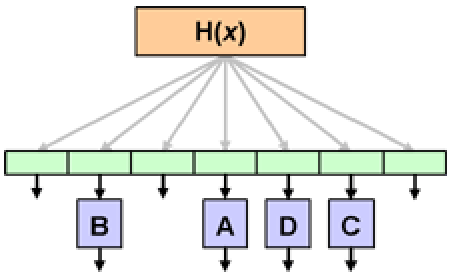
```
Table Size = 10 elements
Hash1(key) = key % 10
Hash2(key) = 7 – (k % 7)

Insert keys: 89, 18, 49, 58, 69

Hash(89) = 89 % 10 = 9

Hash(18) = 18 % 10 = 8

Hash(49) = 49 % 10 = 9 a collision !
                   = 7 – (49 % 7)
                   = 7 positions from [9]

Hash(58) = 58 % 10 = 8
                   = 7 – (58 % 7)
                   = 5 positions from [8]

Hash(69) = 69 % 10 = 9
                   = 7 – (69 % 7)
                   = 1 position from [9]
```

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

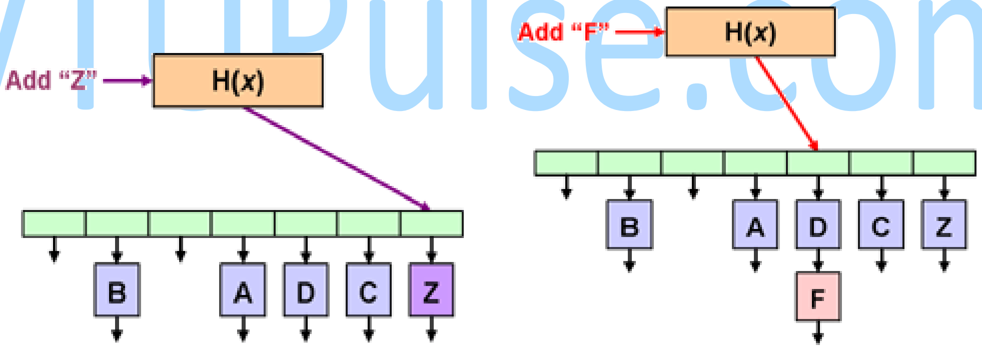
Double hashing minimizes repeated collisions and the effects of clustering.

Chaining

Chaining is another approach to implementing a hash table; instead of storing the data directly inside the structure, have a linked list structure at each hash element. That way, all the collision, retrieval and deletion functions can be handled by the list, and the hash function's role is limited mainly to that of a guide to the algorithms, as to which hash element's list to operate on.



The linked list at each hash element is often called a chain. A *chaining* hash table gets its name because of the linked list at each element -- each list looks like a 'chain' of data strung together.Operations on the data structure are made far simpler, as all of the data storage issues are handled by the list at the hash element, and not the hash table structure itself.



Chaining overcomes collision but increases search time when the length of the overflow chain increases

Drawbacks of static hashing

1. Table size is fixed and hence cannot accommodate data growth.

2. Collsions increases as data size grows.

Avoid the above conditions by doubling the hash table size. This increase in hash table size is taken up, when the number of collisions increase beyond a certain threshold. The threshold limit is decided by the load factor.

Load factor

The load factor α of a hash table with n key elements is given by **α= n / hash table size**

The probability of a collision increases as the load factor increases. We cannot just double the size of the table and copy the elements from the original table to the new table ,since when the table size is doubled from N to 2N+1, the hash function changes. It requires reinserting each element of the old table into the new table (using the modified hash function).This is called Rehashing. Rehashing in large databases is a tedious process and hence dynamic hashing.

Dynamic hashing schemes

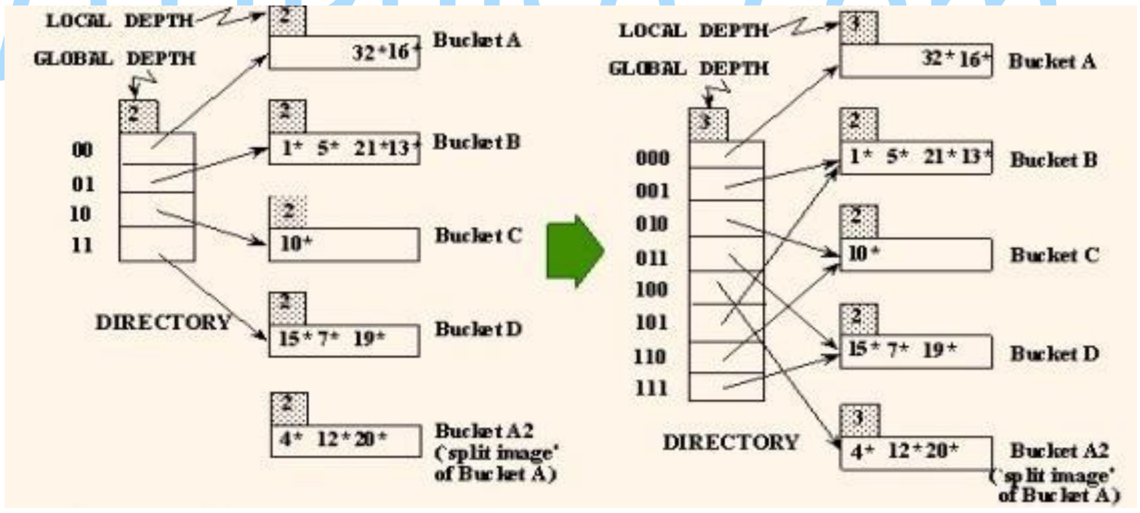
Dynamically increases the size of the hash table as collision occurs.There are two types:

Extendible hashing (directory): uses a directory that grows or shrinks depending on the data distribution. No overflow buckets

Linear hashing(directory less): No directory. Splits buckets in linear order, uses overflow buckets.

Extendible hashing :

- Uses a directory of pointers to buckets/bins which are collections of records
- The number of buckets are doubled by doubling the directory, and splitting just the bin that overflowed.
- Directory much smaller than file, so doubling it is much cheaper. Only one bin of data entries is split and rehashed.



Global Depth

– Max number of bits needed to tell which bucket an entry belongs to.

Local Depth

- The number of least significant digits that is common for all the numbers sharing the same bin.

On overflow:

If global depth =Local depth

1. Double the hash directory
2. SPlit the overflowing bin
3. Redistribute elements of the overflowing bin

4. Increment the global and local depth

If global depth >Local depth

1. SPlit the overflowing bin
2. Redistribute elements of the overflowing bin
3. Increment the local depth

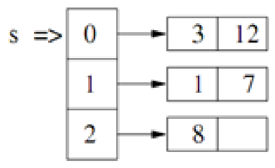
Linear Hashing

Basic Idea:

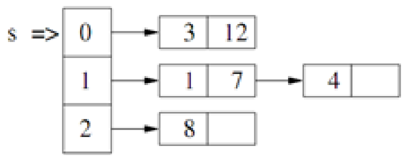
- Pages are split when overflows occur – but not necessarily the page with the overflow.
- Directory avoided in LH by using overflow pages. (chaining approach)
- Splitting occurs in turn, in a round robin fashion.one by one from the first bucket to the last bucket.
- Use a family of hash functions $\mathbf{h}_0$, $\mathbf{h}_1$, $\mathbf{h}_2$, ...
  - Each function's range is twice that of its predecessor.
- When all the pages at one level (the current hash function) have been split, a new level is applied.

- Insert in Order using linear hashing: 1,7,3,8,12,4,11,2,10,13.....

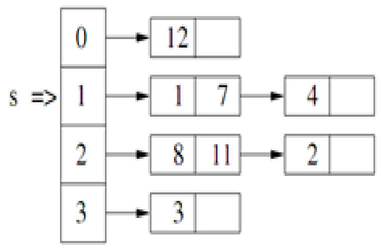
After insertion till 12:



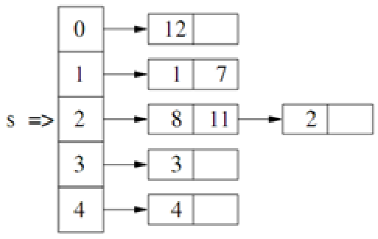
When 4 inserted overflow occurred. So we split the bucket (no matter it is full or partially empty). And increment pointer.



So we split bucket 0 and rehashed all keys in it. Placed 3 to new bucket as (3 mod 6 = 3) and (12 mod 6 = 0). Then 11 and 2 are inserted. And now overflow. *s* is pointing to bucket 1, hence split bucket 1 by re- hashing it.



After split:



Insertion of 10 and 13: as (10 mod 3 = 1) and bucket 1 < s, we need to hash 10 again using h1(10) = 10 mod 6 = 4th bucket.

Note :Files topics can be referred from the text book reference(reema thareja)