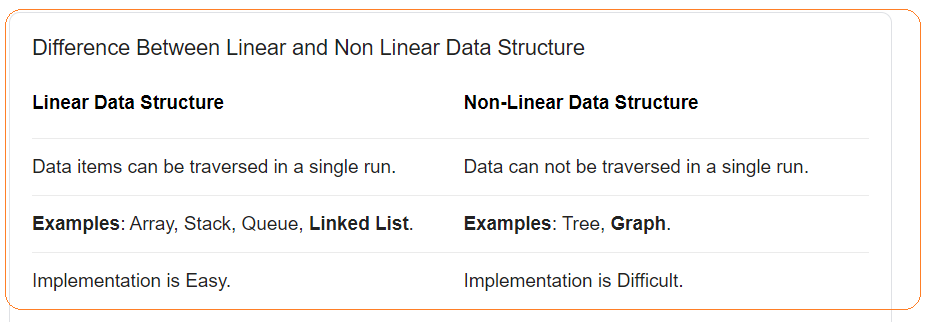
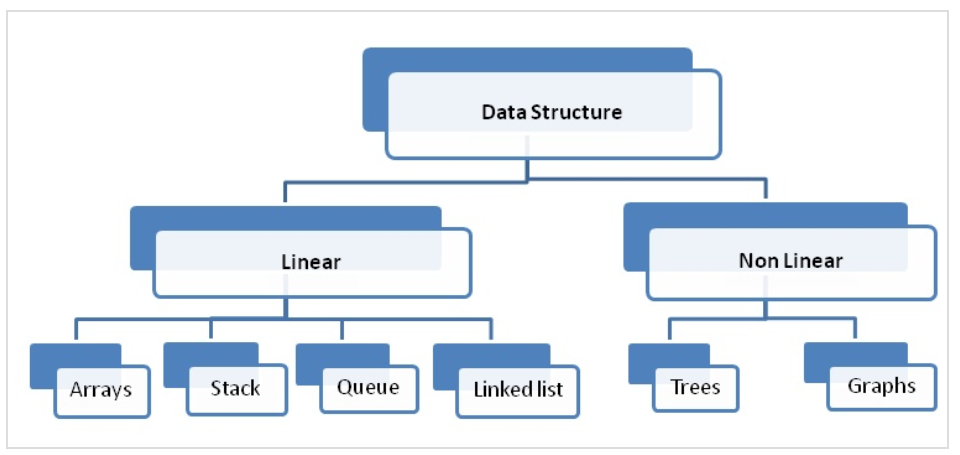
# LINEAR DATASTRUCTURES vs NON LINEAR





# LINKED LIST

Linked List is a linear data structure where each element is a separate object.

“+”:

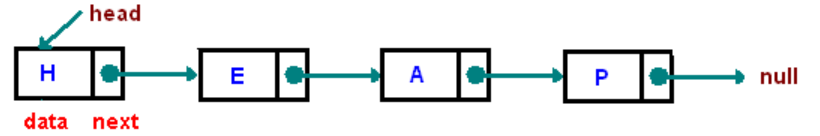
“-“:

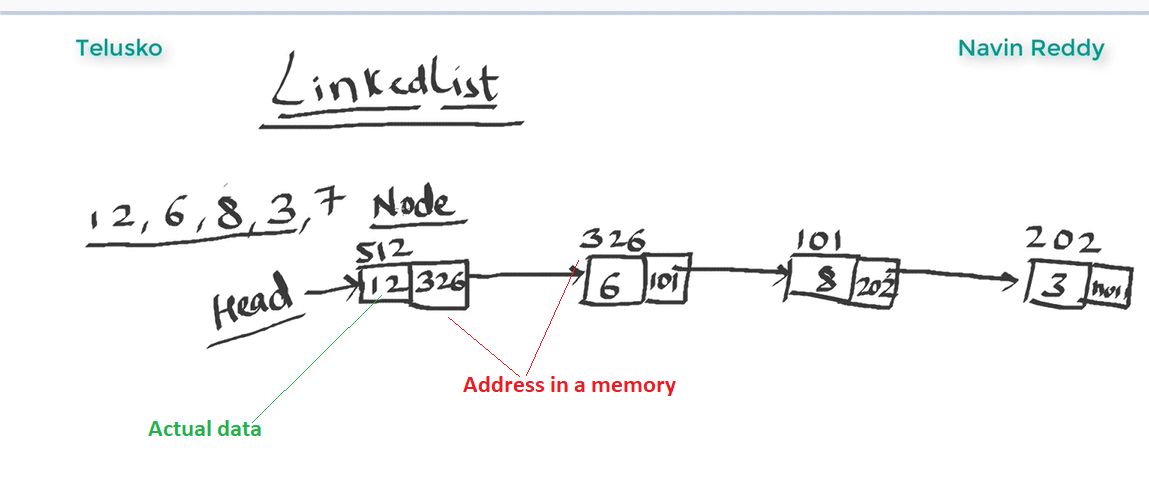
* One disadvantage of a linked list against an array is that it does not allow direct access to the individual elements. If you want to access a particular item then you have to start at the head and follow the references until you get to that item.
* Another disadvantage is that a linked list uses more memory compare with an array - we extra 4 bytes (on 32-bit CPU) to store a reference to the next node.

**Node** - Each element (we will call it a node) of a list is comprising of two items - the data and a reference to the next node. The last node has a reference to null.

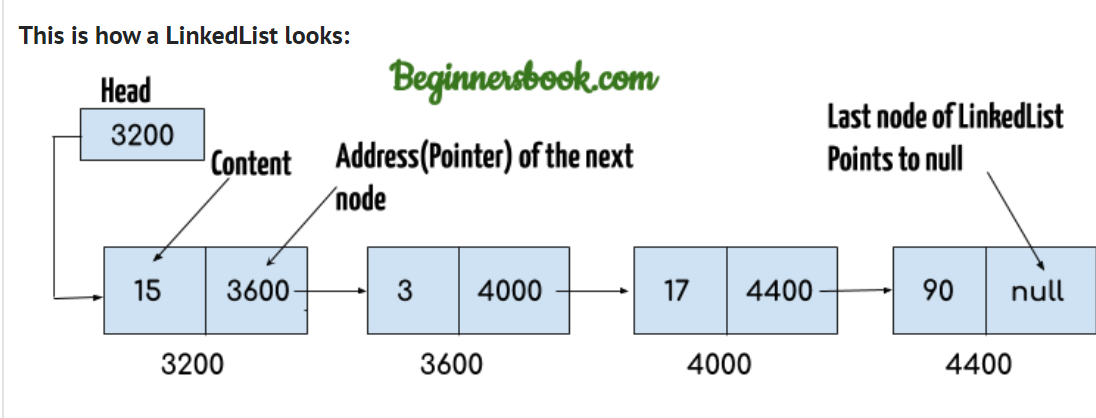
**Head** - The entry point into a linked list is called the head of the list. It should be noted that head is not a separate node, but the reference to the first node. If the list is empty then the head is a null reference.

The number of nodes in a list is not fixed and can grow and shrink on demand. The number of nodes in a list is not fixed and can grow and shrink on demand.





**Head** of the LinkedList only contains the Address of the **First element** of the List.

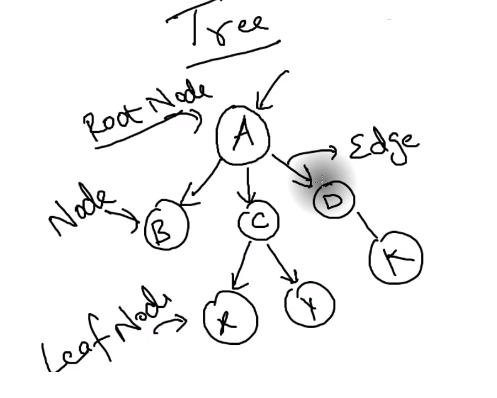


Arrays disadvantages:

Arrays are expensive to add new elements

# TREE

Tree – actually an instance of more general category called [graph]



**Leaf Node** – node without child

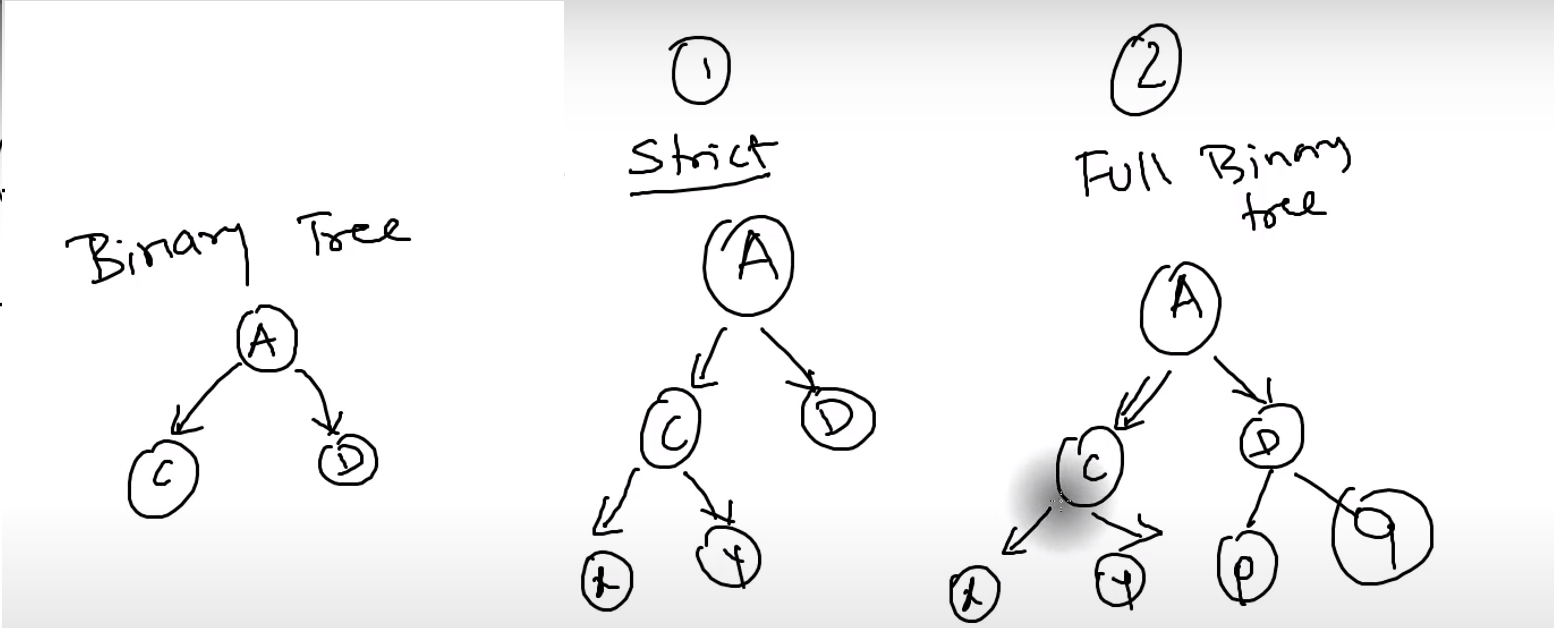
**Edge** – connection between nodes

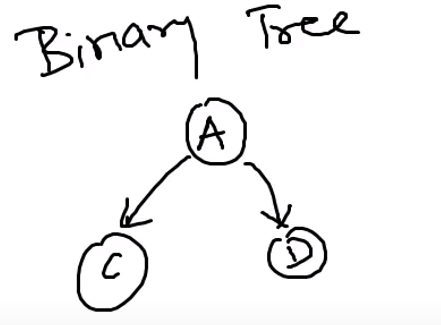
**Traversing** – means visit all nodes in a specified order

**Binary Tree** – is a tree where each node can have not more than 2 child

**Strict Binary Tree** – when node has 2 or 0 child

**Full Binary Tree** – when all children are on the same level





# BINARY TREE

Finding an element in [linkedlist] is not easy task. You need stat from a head to search it and for Deletion – the same case

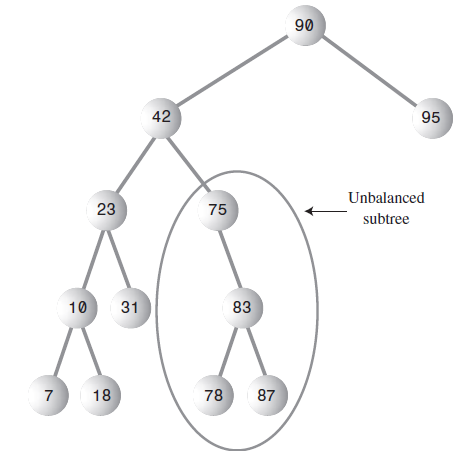
**Binary Tree** – is a tree where each node can have not more than 2 child

**Binary Search Tree** –

BST can be used for

* Searching
* Compression – Huffman Code

**Unbalanced Tree** – tree that have most of the nodes on one side



The most common way to store the nodes of Binary Tree is to store nodes in memory and connect them using references to each nodes

Complexity of finding a node depends on number of levels. This is O(logN) time

Node: *[binary tree] and [binary search tree] are different trees*.

## **TRAVERSING**

|  |  |  |
| --- | --- | --- |
| TYPE | METHOD | DESCRIPTION |
| **DEPTH-FIRST**  **ALGHORITHM** | PreOrder |  |
| InOrder |  |
| PostOrder |  |
| **BREADTH-FIRST**  **ALGHORITHM** |  |  |

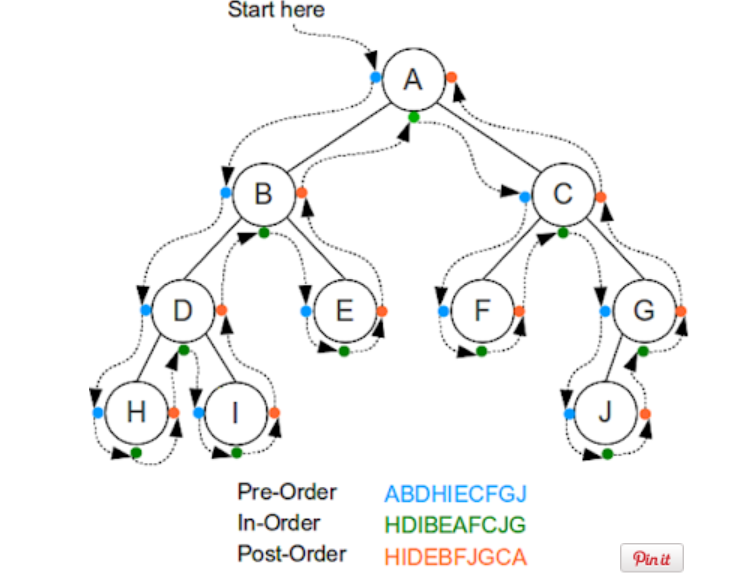
**Traversing** – means visit all nodes in a specified order

There are three ways to traverse tree:

* **inorder** - all nodes visited in ascending order
* **preorder** - root is visited, then left subtree, and right subtree
* **postorder** –

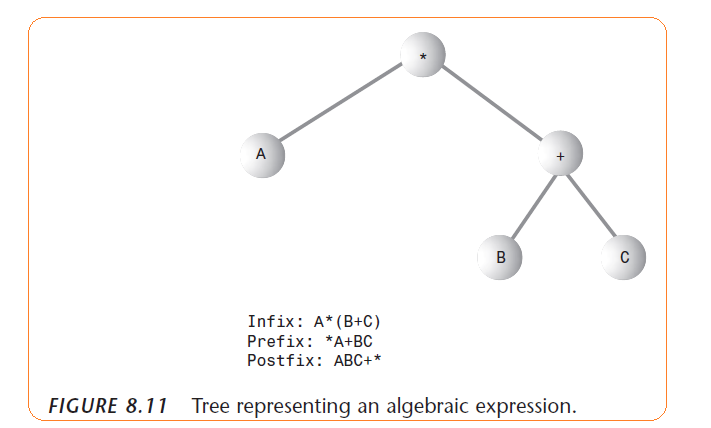
note:

* the most common way to traverse is [***inorder***].
* [preorder] and [postorder] can be used in [binary tree] to parse algebraic expressions



There are 2 implementation of traversal:

* **Recursion method** – it’s ease, but can throw [StackOverflow} for large tree
* **Iterative method** – can be implemented using [stack]

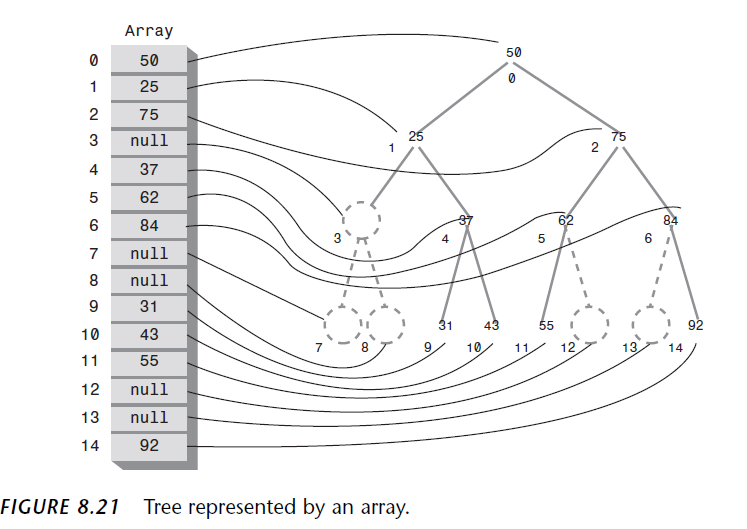


Approaches to traverse tree

* recursive – the simplest way
* iterative –

Representing tree ways:

* Based on [leftChild] and [rightChild]
* Based on [array] – in most situations this way is not efficient



## **BINARY SEARCH TREE**

pros:

it works well for randomly inserted data

cons:

it works slowly if we insert ordered data

## **HUFFMAN CODE**

Binary Search tree can also be used for compressions

Every character takes one byte for ACSII code

Every character takes two bytes for Unicode

## **TYPE OF INSERTION IN A TREE**

Top-Down Insertion –

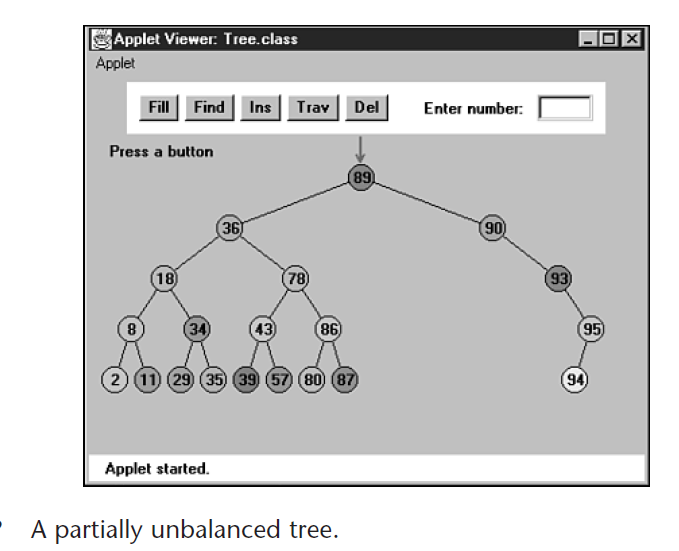
Bottom Up Insertion –

## **BALANCED AND UNBALANCED TREE**

**Unbalanced tree** – more node appears on one side compared to another side

Unbalanced Tree: O(N) = speed of search

Balanced Tree: O(logN) = speed of search



## **RED-BLACK TREE**

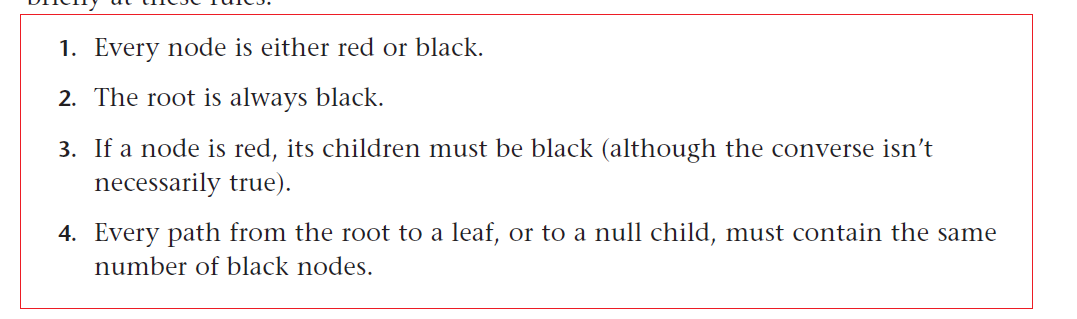
**Red-Black Tree** – is a Binary Search Tree with some feature:

* Nodes are colored
* For insertion and deletion rules are followed

**Red-Black Tree** – is a Binary Search Tree that guarantees O(logN) time for insertion, deletion and searching.However, there is one penalty is that storage required for each node is slightly increased

**To accommodate red**-black color (a Boolean variable)

Red-Black Tree solves the problem of Unbalanced Trees



Black height – number of the black nodes from the root to a given node

## **AVL TREE**

AVL = Adelson – Velskii and Landis

AVL Tree – is the earliest kind of balanced tree

In AVL tree each node stores an additional piece of data:

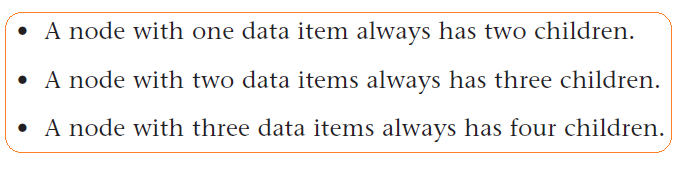
The difference between the heights of left and right subtree

Search time in AVL tree is O(logN)

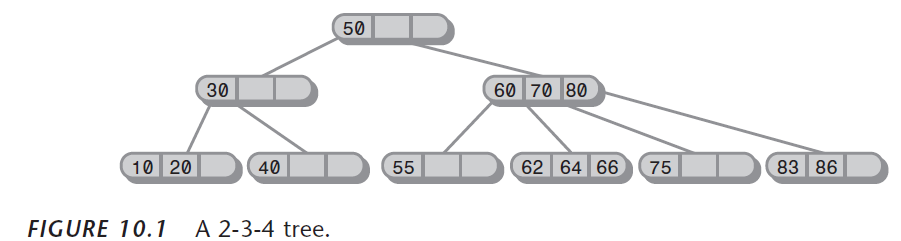
## **2-3-4 TREE**

**2-3-4 tree** – is a multiway tree that can have up to four children and three items per a node.

2-3-4 means how any children can be:

****

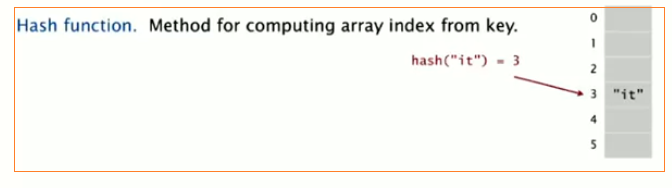
In [2-3-4] a node with a single link is not allowed

****

## **HASH TABLE**

**Hash Function. Def1.** – it is a calculation that converts a number of large range into a relatively small index number (modulo division) that corresponds in a position in hash table. This index number is a memory address

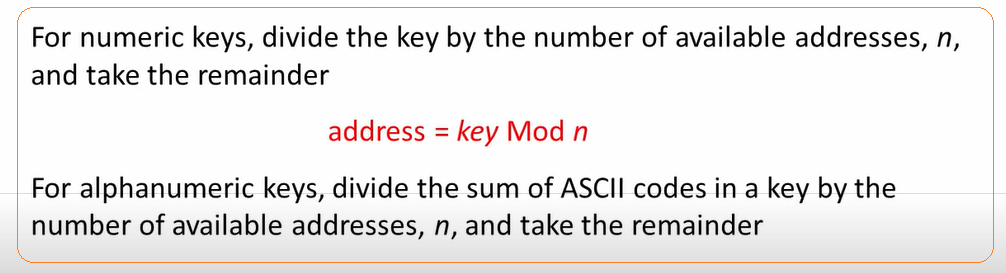
**Hash Function. Def2.** – in another words is a method for computing array index from key.



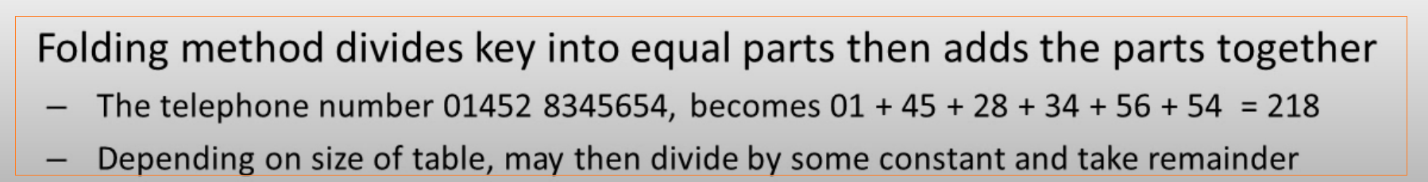
Hash Table – is an array into which data is inserted using a hash function

Hash Table – is a data structure that offers fast searching and insertion. It is often used to store key value pairs

**The first way of hash function:**



**The second way of hash function – folding method:**



HashMap –is key value hash table

Hash tables is used:

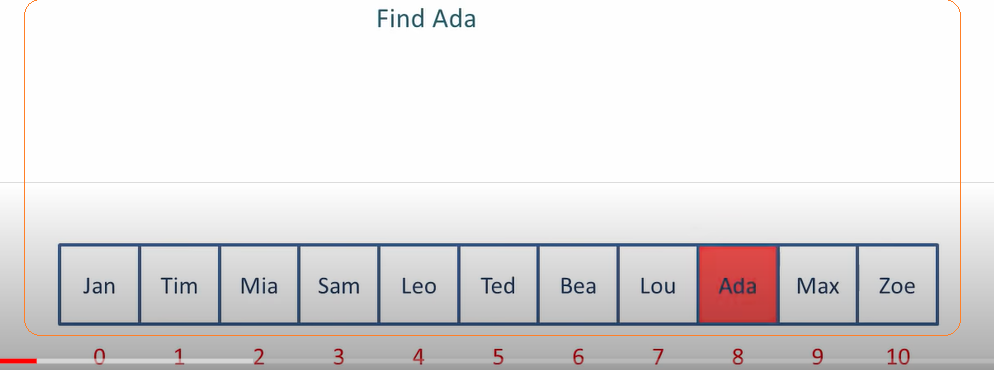
* In database indexing
* Caching

Overview

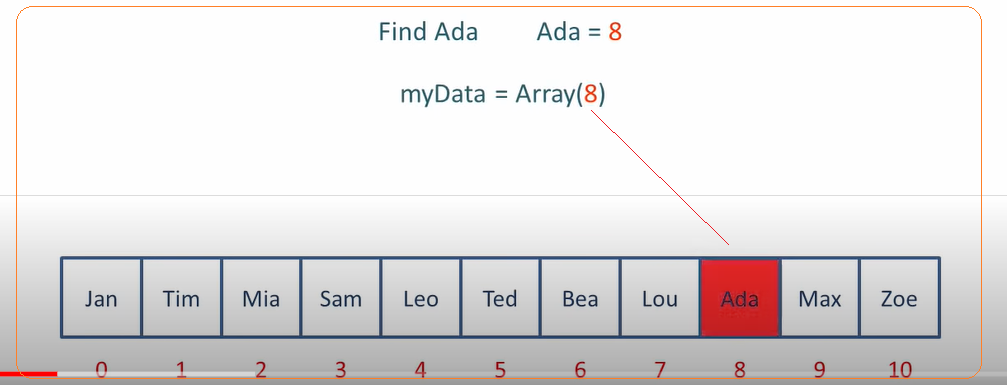
In you need to find an element in array you can make a search in a brute force way – check every element, but if you know index of element in array it can retrieved fast.

But how can you know an index number? Answer – an index number is a related to data. Therefore,

Brute force:



Based on hash alghorithm



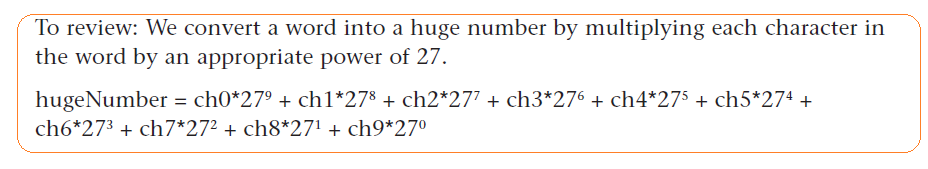
**Collisions** – a situation when two values will need to be inserted in same index array

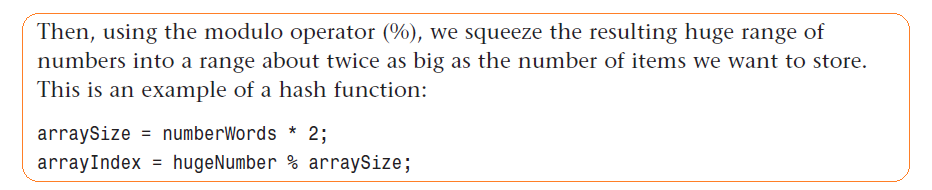
Pros:

* Fast searching and insertion. It can take a constant time O(1)
* With hashing we get O(1) search time on average (under reasonable assumptions) and O(n) in worst case.

Cons:

* It is based on an array and array is difficult to expand
* Performance can degrade when array is full for some kind of hash table

****

****

Hash algorithm characteristics:

* Performance – it should be reasonably fast. It cannot be slow and too fast. If it is too fast it can be broken from security pint of view – you can create a document that can have the same hash value
* If you change any symbol or byte, a hash value should be different
* Avoid collision

hash function can be used:

* for verifying algorithms

Example of hash function

* MD5 – now used rarely, because there is collision and some can maliciously use it
* SHA – Secure Hash algorithm. 160 bit value.

## **COLLISION**

There are a few approaches to deal with a collision

**Open addressing** – when a data item can not be inserted at the index calculated by a hash function, another location is sought. In [open addressing] all elements are stored in the hash table. Each table contains either a record or null. When searching for an element, we one by one examine table slots until the desired element is found. There are three methods of open adressing

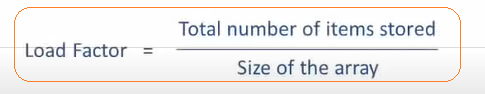
* Linear probing – we search sequentially for vacant cells where to insert data item
* Quadratic probing -
* Double hashing -

Cons:

* The issue with [open addressing] is **[primary clustering]**.
* A performance is degrade when hashtable is populated 1/3 -2/3 and above
* Rehashing –when array is full, you will need to create a new array and put items there (each y each using hash function)
* Soft delete

**Separate chaining or closed addressing** - is to create an array that consists of linked lists of words instead of words themselves. Then, when a collision happens, the new item is simply inserted in the list at

**Load factor** is a ration between [total number of items stored] and [size of the array]



## **LINEAR PROBING**

**Probe** – is a process of finding appropriate cell to insert value.

**Cluster** is a group of contentious keys

**Primary clustering** – this problem happens during linear probing. If hash func returns the same value then the hash algorithm insert new value to the end of cluser. If a cluster is big it takes longer time search by cluster.

**Secondary clustering** - elements that hash to the same hash key will always probe the same alternative cells. This means that once the table is more than half full, it's difficult to find an empty spot.

Quadratic probing solves the issue with primary clustering, but a new problem secondary clustering happen

[Double hashing] solves problems of primary and secondary hashing

**Rehashing** – when array is full we need to create a new array, but we can’t simply copy data there from previous array, because index is calculated based on by module division value%arraySize. We will need insert each value separately based on by module division

**Linear probing** – the algorithm searches for vacant cells sequintailyy

Cons:

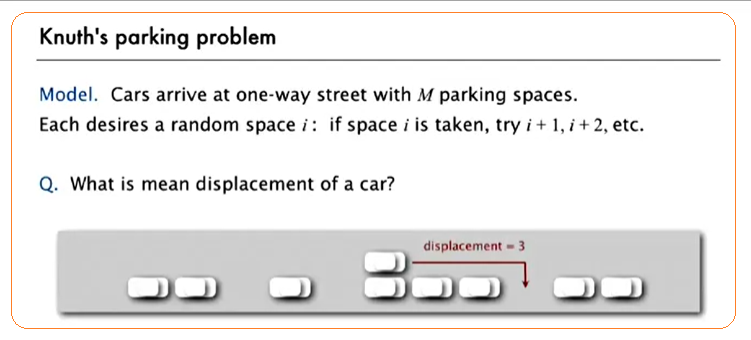
* It works until a size of array is significantly more than number of element
* Delete is soft – when we need to delete an element we mark as deleted. Why it is?

If we delete (that is put an null) then [find()] method will be broken\, because it searches values on a cluster until null is met

* Rehashing – when array is full we need to create a new array, but we can’t simply copy data there from previous array, because index is calculated based on by module division value%arraySize. We will need insert each value separately based on by module division

**Knukh’s Parking problem:**

What happens is that you are on a one way street and you are looking for a praking place and it is the idea is you start looking for a parking at particular times and say "oaky" , now I need a parking place. What you are doing is [linear problem hashing] if the current space taken, you try the next space and the one after and so forth



## **QUADRATIC PROBING**

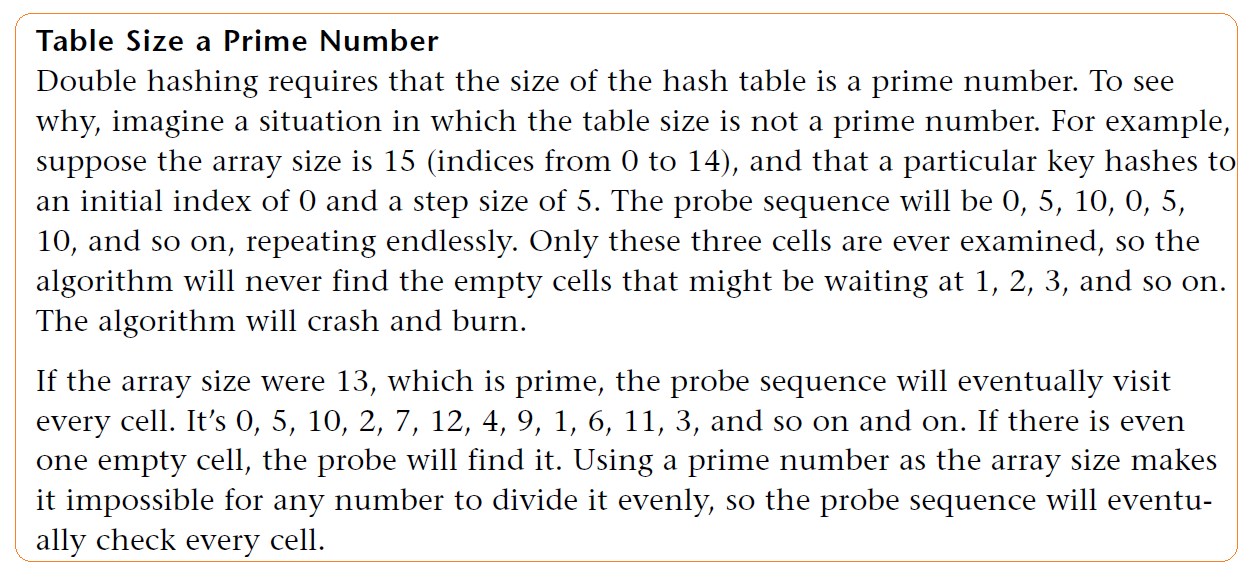
Quadratic Probing - probe is going to x + 12 + 22 + 32 + 52 + 72

Quadratic Probing is solving [Primary clustering] problem so it’s trying to avoid creating a big cluster

## **DOUBLE HASHING**

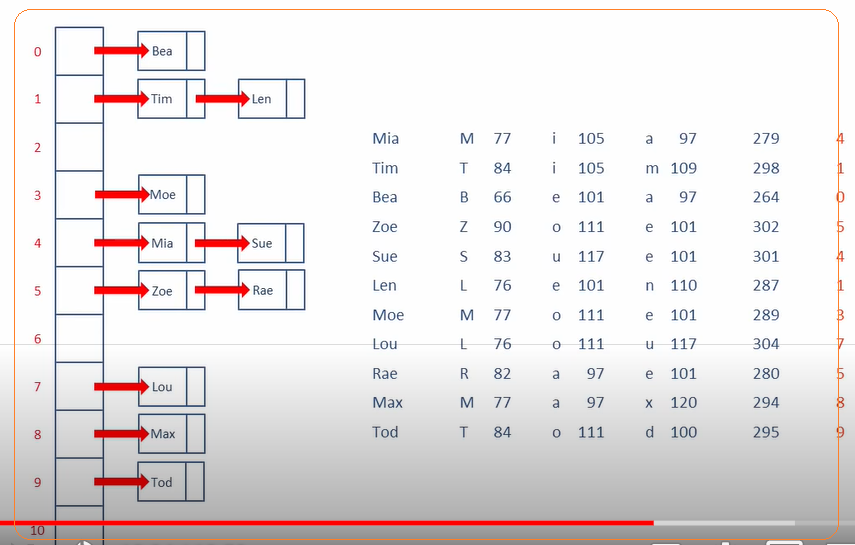
**Double hashing** - solves problems of primary and secondary hashing. It generates a code that depends on key instead of being the same for each key. The idea is it generates hash key the second time using different hash function.

**(!) arraySize must be prime for double hashing**



## **SEPARATE CHANING**

if [load factor] is a low it is better to use [open addressing]

****

## **PROBLEMS**

|  |  |  |  |
| --- | --- | --- | --- |
| ID | TOPIC | PROBLEM | DESCRIPTION |
| **1** | **HASHING. LINEAR PROBING** | **KNUKH’S**  **PARKING PROBLEM** | What happens is that you are on a one way street and you are looking for a parking place and it is the idea is you start looking for a parking at particular times and say "oaky" , now I need a parking place. What you are doing is [linear problem hashing] if the current space taken, you try the next space and the one after and so forth |
|  |  |  |  |
|  |  |  |  |

**LINKS**

Linked list

<https://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked%20Lists/linked%20lists.html>

* Linear vs Non Linear Data Structures

<https://knowshares.wordpress.com/2016/12/14/linear-vs-non-linear-datastructure/>