**Data Structures**

(revision material)

In every computational task we try to perform in our day to day life is nothing but doing some operation on data. For example, a simple mouse clicks on computer or tapping on our mobile touch screen invokes some operations that are associated with that region of screen/monitor like selecting text over there or changing the brightness of the screen or anything for that matter involves some operations on data relevant to it. All these aspects of computations finally boil down to accessing or manipulating data. How fast and how easily we can perform these data manipulations is crucial. Here we try to discuss some data structures that can develops some intuition of this concept.

**HASH TABLE**

In C language we have arrays which can arrange similar type of data elements in consecutive memory locations. This feature allows us to access any element of the array, by its index, just in constant time, O(1). At the same time C arrays are not elastic in size, one can’t change their size once it is declared. On the other hand, C allows us to implement linked lists using structures and pointers using which one can arrange collection of data elements dynamically expandable in size.

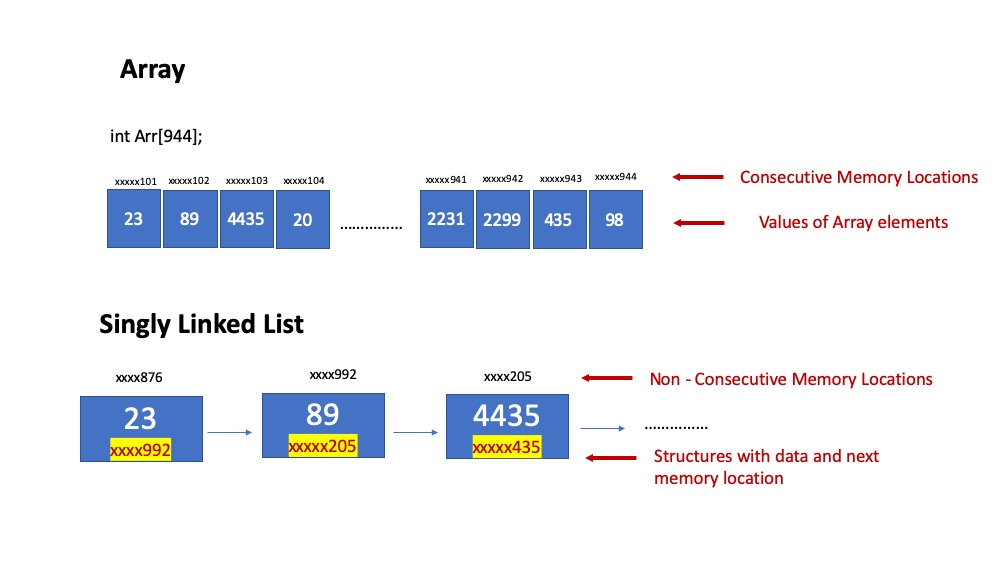


Figure 1: Array & Linked list

Hash tables are combined representation of both Arrays and linked lists. When we have data of indefinite size but can be categorized into definite number of groups basing on some function Hash table representation is very useful. The function that categorizes the data into non-overlapping is called hash function. Groups are represented using array as its size is fixed. Groups of data elements are stored in linked lists whose first address is stored in the array element that representing the group.

English (any language for that matter) dictionary can be represented using hash table as shown in *Figure 2*. All the words in the dictionary in grouped basing on their starting letter so number of possible groups is fixed now so it can be implemented using an array where each array element holds the starting address of corresponding linked list. In the Figure 2 one can observe array (orange like color) of singly linked lists(blue). Each array element stores the address of starting point of the corresponding linked list. This representation of dictionary data helps us in many operations such as adding a new word to our dictionary. First, we try to find out what is the first letter of that word and corresponding linked lists starting address. In the next step the new word will be added in its appropriate position of the linked list. The complexity of accessing the starting address of relevant linked list is ~ O(1). And the complexity of adding the word to its linked list is always very less than O(n) where n is the total number of words in the dictionary. The complexity is same for other operations like editing a word, deleting a word or searching word.

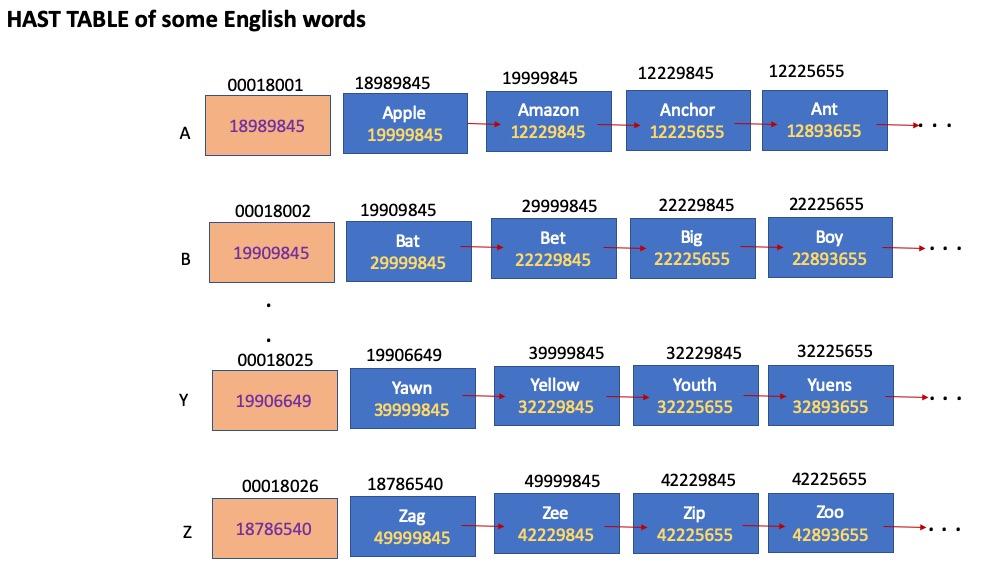


Figure 2 Hash table representation of collection of some English words

Some important aspects of Hash tables:

Hashing function that categorizes the data into non-overlapping groups. In the above case hashing function just finds the first character of the given word and maps it to the array index of corresponding linked list. Data collection need not always be collection of single items like our dictionary example. Data items can be anything, student’s information, employee information, customer information, company’s information like in stock markets, properties of molecules, atom co-ordinates in protein structures etc. One has to choose proper key that represents the item and its category. If students’ records are our data of interest, each student record may consist lot of information such as his data of birth, institute joining date, his father’s name etc. out all that student year of joining can be our key of categorization. If students with same year of joining are chained form a linked list their starting address should be kept in an array whose index can be obtained by hashing function for the given student record.

Activity:

1. What will be complexity if dictionary, like above, is implemented using just an array or just a linked list?
2. Think of some other collections of data that can be represented as hash table and how?
3. How a hash function can be chosen for a given set of data?
4. Write an algorithm/ pseudo code to implement hash table for given set of data in C.

**Tree Structures**

As we have seen in hash tables, categorizing data for organizing in memory help us in efficiently access it and manipulate it. One of the main goals of data structuring is to minimize the complexity of accessing and manipulating the data as per our requirement. The requirements may vary from problem to problem. In some cases, we may wish to access only the highest value in one particular dimension in others it may be the smallest. So, data structuring also varies from one problem to other.

But what is the generalized form of data organization?

Nature has given us the solution in the form of trees, circulatory systems animals, where very complex activities are being carried out. In a very minimalist form, we try to use these tree structures for our purpose of data organization and see how it helps us.

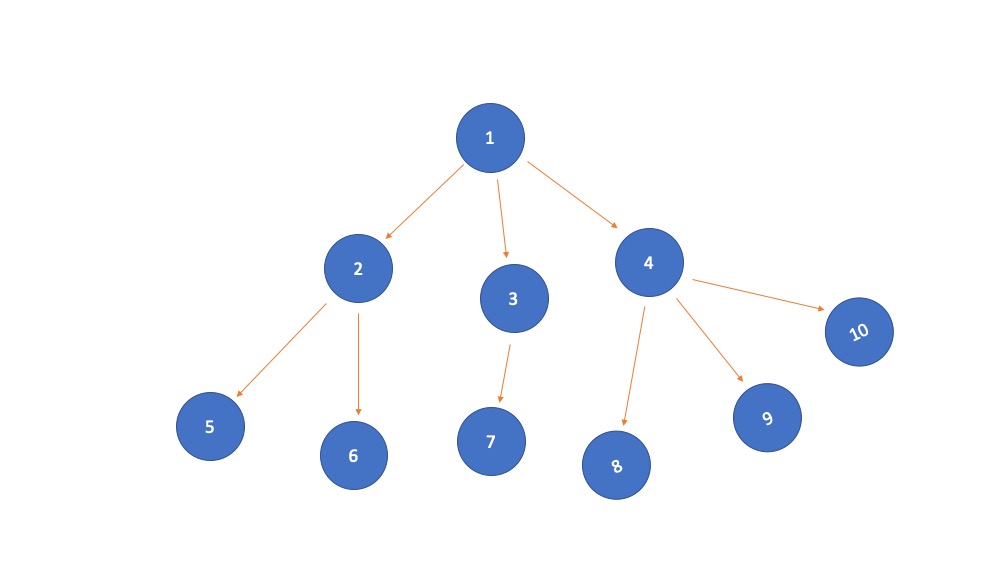


Figure : Tree structure

A generalized tree structure consists of nodes and branches as shown in figure 3. Node 1 is called the root node, nodes 5,6,7,8,9,10 are leaf nodes. Nodes 5&6 are child nodes of node 2 which is having root node as its parent. This nomenclature is often used in our discussions further. Nodes hold some data and represented by a key. In our example the numbers 1 to 10 are keys corresponding to each node. Usually keys are unique and collection of keys represents a set. If students’ records are saved in tree structure each node consists of data corresponding to each student and its key can be anything that uniquely represents the student such as his roll number or rank in some entrance exam.

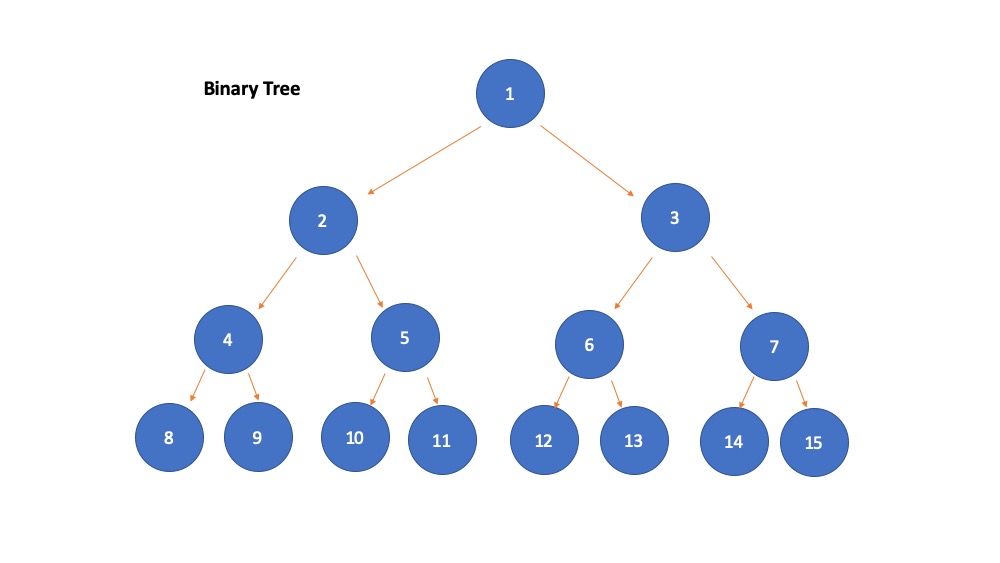


Figure : In a **binary tree** any node can have at most two child nodes.

**Heap/ Min-max tree/ priority queue:**