International Journal of Trend in Scientific Research and Development (IJTSRD)

International Open Access Journal | www.ijtsrd.com

ISSN No: 2456 - 6470 | Volume - 3 | Issue – 1 | Nov – Dec 2018

Data Structure: Theoretical Approach

Durgesh Raghuvanshi
B-Tech Department of Computer Science,
IILM Academy of Higher Learning, Greater Noida, Uttar Pradesh, India

ABSTRACT
Run with accordance with significance. The first if these this paper explains about the basic terminologies used in this paper in data structure. Better running times will be other constraints, such as memory use which will be paramount. The most appropriate data structures and algorithms rather than through hacking removing a few statements by some clever coding. Data structures serve as the basis for abstract data types (ADT). "The ADT defines the logical form of the data type. The data structure implements the physical form of the data type."Different types of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, relational databases commonly use B-tree indexes for data retrieval, while compiler implementations usually use hash tables to look up identifiers.

INTRODUCTION:

Data structures serve as the basis for abstract data types (ADT). "The ADT defines the logical form of the data type. The data structure implements the physical form of the data type."Different types of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. For example, relational databases commonly use B-tree indexes for data retrieval, while compiler implementations usually use hash tables to look up identifiers. Data structures provide a means to manage large amounts of data efficiently for uses such as large databases and internet indexing services. Usually, efficient data structures are key to designing efficient algorithms. Some formal design methods and programming languages emphasize data structures, rather than algorithms, as the key organizing factor in software design. Data structures can be used to organize the storage and retrieval of information stored in both main memory and secondary memory. Data structures are generally based on the ability of a computer to fetch and store data at any place in its memory, specified by a pointer—a bit string, representing a memory address, that can be itself stored in memory and manipulated by the program. Thus, the array and record data structures are based on computing the addresses of data items with arithmetic operations, while the linked data structures are based on storing addresses of data items within the structure itself. Many data structures use both principles, sometimes combined in non-trivial ways (as in XOR linking).[citation needed]

The implementation of a data structure usually requires writing a set of procedures that create and manipulate instances of that structure. The efficiency of a data structure cannot be analyzed separately from those operations. This observation motivates the theoretical concept of an abstract data type, a data structure that is defined indirectly by the operations that may be performed on it, and the mathematical properties of those operations (including their space and time cost).[citation needed]An array is a number of elements in a specific order, typically all of the same type (depending on the language, individual elements may either all be forced to be the same type, or may be of almost any type). Elements are accessed using an integer index to specify which element is required. Typical implementations allocate contiguous memory words for the elements of arrays (but this is not necessity). Arrays may be fixed-length or resizable. A linked list (also just called list) is a linear collection of data elements of any type, called nodes, where each node has itself a value, and points to the next node in the linked list. The principal advantage of a linked list over an array, is that values can always be efficiently inserted and removed without relocating the rest of the list. Certain other operations, such as random access to a certain element, are however slower on lists than on arrays. Most assembly

languages and some low-level languages, such as BCPL (Basic Combined Programming Language), lack built-in support for data structures. On the other hand, many high-level programming languages and some higher-level assembly languages, such as MASM, have special syntax or other built-in support for certain data structures, such as records and arrays.

Sequential search

When data items are stored in a collection such as a list, we say that they have a linear or sequential relationship. Each data item is stored in a position relative to the others. In Python lists, these relative positions are the index values of the individual items. Since these index values are ordered, it is possible for us to visit them in sequence. This process gives rise to our first searching technique, the sequential search. Starting at the first item in the list, we simply move from item to item, following the underlying sequential ordering until we either find what we are looking for or run out of items. If we run out of items, we have discovered that the item we were searching for was not present.

Algorithm complexity

Algorithm	Best case	Expected
Selection sort	O(N2)	O(N2)
Merge sort	O(NlogN)	O(NlogN)
Linear search	O(1)	O(N)
Binary search	O(1)	O(logN)

Depth of node

The depth of node is the length of the path from the root to the node. A rooted tree with only one node has a depth of zero.

Threaded binary tree

In a threaded binary tree all the null pinters which wasted the space in linked representation is converted into useful links called threads thus representation of a binary tree using these threads is called threaded binary tree.

Analysis of sequential search

To analyze searching algorithms, we need to decide on a basic unit of computation. Recall that this is typically the common step that must be repeated in order to solve the problem. For searching, it makes sense to count the number of comparisons performed. Each comparison may or may not discover the item we are looking for. In addition, we make another assumption here. The list of items is not ordered in any way. The items have been placed randomly into the list. In other words, the probability that the item we are looking for is in any particular position is exactly the same for each position of the list.

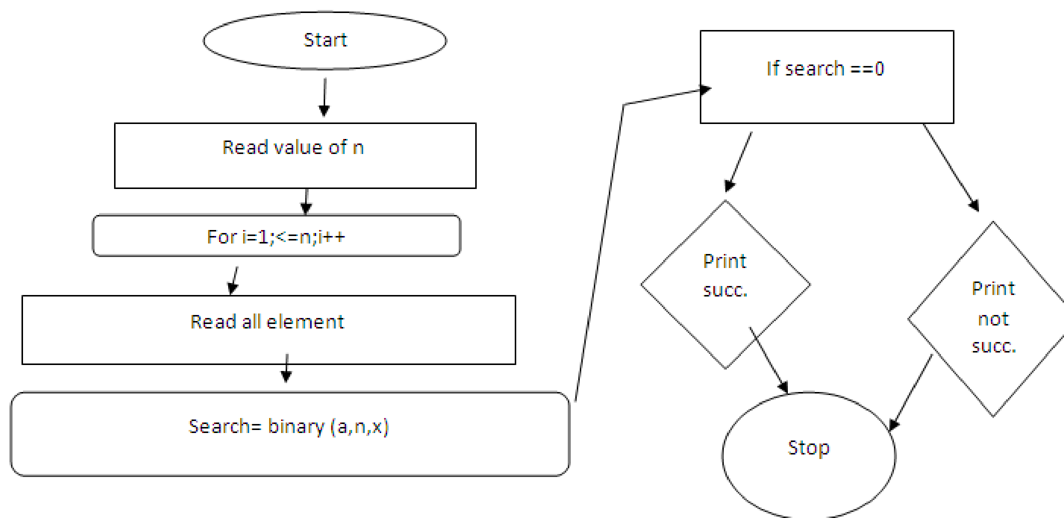
If the item is not in the list, the only way to know it is to compare it against every item present. If there are \(n\) items, then the sequential search requires \(n\) comparisons to discover that the item is not there. In the case where the item is in the list, the analysis is not so straightforward. There are actually three different scenarios that can occur. In the best case we will find the item in the first place we look, at the beginning of the list. We will need only one comparison. In the worst case, we will not discover the item until the very last comparison, the nth comparison.

Binary search

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the sub array reduces to zero. B-trees are generalizations of binary search trees in that they can have a variable number of sub trees at each node. While child-nodes have a pre-defined range, they will not necessarily be filled with data, meaning B-trees can potentially waste some space. The advantage is that B-trees do not need to be re-balanced as frequently as other self-balancing trees.

Due to the variable range of their node length, B-trees are optimized for systems that read large blocks of data. They are also commonly used in databases. A ternary search tree is a type of tree that can have 3 nodes: a lo kid, an equal kid, and a hi kid. Each node stores a single character and the tree itself is ordered the same way a binary search tree is, with the exception of a possible third node. Searching a ternary search tree involves passing in a string to test whether any path contains it. The time complexity for searching a balanced ternary search tree is O(log n).



Binary Search Tree, is a node-based binary tree data structure which has the following properties: The left subtree of a node contains only nodes with keys lesser than the node's key. The right subtree of a node contains only nodes with keys greater than the node's key. The left and right subtree each must also be a binary search tree. There must be no duplicate nodes. delete operation is discussed. When we delete a node, three possibilities arise. 1) Node to be deleted is leaf: Simply remove from the tree.3) Node to be deleted has two children: Find in order successor of the node. Copy contents of the in order successor to the node and delete the in order successor. Note that in order predecessor can also be used.2) Node to be deleted has only one child: Copy the child to the node and delete the child The important thing to note is, in order successor is needed only when right child is not empty. In this particular case, in order successor can be obtained by finding the minimum value in right child of the node. Time Complexity: The worst case time complexity of delete operation is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf Now when we want to search for an item, we simply use the hash function to compute the slot name for the item and then check the hash table to see if it is present. This searching operation is \(O(1)\), since a constant amount of time is required to compute the hash value and then index the hash table at that location. If everything is where it should be, we have found a constant time search algorithm.

You can probably already see that this technique is going to work only if each item maps to a unique location in the hash table. For example, if the item 44 had been the next item in our collection, it would have a hash value of 0 (\(44 \% 11 == 0\)). Since 77 also had a hash value of 0, we would have a problem. According to the hash function, two or more items would need to be in the same slot. This is referred to as a collision (it may also be called a "clash"). Clearly, collisions create a problem for the hashing technique. We will discuss them in detail later. We now return to the problem of collisions. When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table. This process is called collision resolution. As we stated earlier, if the hash function is perfect, collisions will never occur. However, since this is often not possible, collision resolution becomes a very important part of hashing.

Collision resolution strategies

One method for resolving collisions looks into the hash table and tries to find another open slot to hold the item that caused the collision. A simple way to do this is to start at the original hash value position and then move in a sequential manner through the slots until we encounter the first slot that is empty. Note that we may need to go back to the first slot (circularly) to cover the entire hash table. This collision resolution process is referred to as open addressing in that it tries to find the next open slot or address in the hash table. By systematically visiting each slot one at a time, we are performing an open addressing technique called linear probing. Once the hash values have been computed, we can insert each item into the hash table at the designated position as shown in Figure 5. Note that 6 of the 11 slots are now occupied. This is referred to as the load factor, and is commonly denoted by \(\lambda = \frac {number of items}{table size}\). For this example, \(\lambda = \frac {6}{11}\).Once we have built a hash table using open addressing and linear probing, it is essential that

we utilize the same methods to search for items. When we compute the hash value, we get 5. Looking in slot 5 reveals 93, and we can return True. What if we are looking for 20? Now the hash value is 9, and slot 9 is currently holding 31. We cannot simply return False since we know that there could have been collisions. We are now forced to do a sequential search, starting at position 10, looking until either we find the item 20 or we find an empty slot. A disadvantage to linear probing is the tendency for clustering; items become clustered in the table. This means that if many collisions occur at the same hash value, a number of surrounding slots will be filled by the linear probing resolution. This will have an impact on other items that are being inserted, as we saw when we tried to add the item 20 above. A cluster of values hashing to 0 had to be skipped to finally find an open position. Hash Functions Given a collection of items, a hash function that maps each item into a unique slot is referred to as a perfect hash function. If we know the items and the collection will never change, then it is possible to construct a perfect hash function (refer to the exercises for more about perfect hash

Binary search tree
It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor Adelson, Velski& Landis, AVL trees are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the Balance Factor.

Here we see that the first tree is balanced and the next two trees are not balanced −In the second tree, the left subtree of C has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of A has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques. In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making A the left-subtree of B.

Right Rotation AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.AVL Rotations To balance itself, an AVL tree may perform the following four kinds of rotations −

Left rotation Right rotation Left-Right rotation Right-Left rotation The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −Right-Left Rotation The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation. As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

An internal sort is any data sorting process that takes place entirely within the main memory of a computer. This is possible whenever the data to be sorted is small enough to all be held in the main memory. For sorting larger datasets, it may be necessary to hold only a chunk of data in memory at a time, since it won't all fit. The rest of the data is normally held on some larger, but slower medium, like a hard-disk. Any reading or writing of data to and from this slower media can slow the sortation process considerably. This issue has implications for different sort algorithms.

Some common internal sorting algorithms include:
Bubble Sort Insertion Sort Quick Sort Heap Sort Radix Sort Selection sort Consider a Bubblesort, where adjacent records are swapped in order to get them into the right order, so that records appear to "bubble" up and down through the dataspace. If this has to be done in chunks, then when we have sorted all the records in chunk 1, we move on to chunk 2, but we find that some of the records in chunk 1 need to

"bubble through" chunk 2, and vice versa (i.e., there are records in chunk 2 that belong in chunk 1, and records in chunk 1 that belong in chunk 2 or later chunks). This will cause the chunks to be read and written back to disk many times as records cross over the boundaries between them, resulting in a considerable degradation of performance. If the data can all be held in memory as one large chunk, then this performance hit is avoided. On the other hand, some algorithms handle external sorting rather better. A Merge sort breaks the data up into chunks, sorts the chunks by some other algorithm (maybe bubblesort or Quick sort) and then recombines the chunks two by two so that each recombined chunk is in order. This approach minimises the number or reads and writes of data-chunks from disk, and is a popular external sort method. It is useful to understand how storage is managed in different programming languages and for different kinds of data. Three important cases are: static storage allocation stack-based storage allocation heap-based storage allocation Static Storage Allocation Static storage allocation is appropriate when the storage requirements are known at compile time. For a compiled, linked language, the compiler can include the specific memory address for the variable or constant in the code it generates. (This may be adjusted by an offset at link time.) Examples: code in languages without dynamic compilation all variables in FORTRAN IV global variables in C, Ada, Algol constants in C, Ada, Algol Stack-Based Storage Allocation Stack-based storage allocation is appropriate when the storage requirements are not known at compile time, but the requests obey a last-in, first-out discipline. Examples: local variables in a procedure in C/C++, Ada, Algol, or Pascal procedure call information (return address etc).Stack-based allocation is normally used in C/C++, Ada, Algol, and Pascal for local variables in a procedure and for procedure call information. It allows for recursive procedures, and also allocates data only when the procedure or function has been called -- but is reasonably efficient at the same time. Typically a pointer to the base of the current stack frame is held in a register, say R0. A reference to a local scalar variable can be compiled as a load of the contents of R0 plus a fixed offset. Note that this relies on the data having known size. To compile a reference to a dynamically sized object, e.g. an array, use indirection. The stack contains an array descriptor, of fixed size, at a known offset from the base of the current stack frame. The descriptor then contains the actual address of the array, in addition to bounds information. References to non-local variables can be handled by several techniques -- the most common is using static links. This is beyond the scope of what we'll cover in 341 this year. Most variable references are either to local variables or global variables, so often compilers will handle global variable references more efficiently than references to arbitrary non-local variables. Scalar local variables (especially parameters) can be handled efficiently as they are often passed through registers.

There are two important limitations to pure stack-based storage allocation.

First, the only way to return data from a procedure or function is to copy it -- if you try to simply return a reference to it, the storage for the date will have vanished after the procedure or function returns. This isn't an issue for scalar data (integers, floats, booleans), but is an issue for large objects such as arrays. For this reason you can't for example return a loclly declared array from a function in C. Second, you can't return a procedure or function as a value, or assign a procedure or function to a global variable (procedures or function values aren't first class citizens).

Comparison between linear search and binary search

Basis for comparison	Linear search	Binary search
Time complexity	O(N)	$O(log2N)$
Best case time	First element 0(1)	Centre element 0(1)
Prerequisite for an array	Not required	Array must be sorted in order
Worst case for N number of elements	N comparisons are required	Can conclude after only $(logN)$
Can be implemented on	Array and linked list	Cannot be directly implemented to linked list
Algorithm type	Iterative in nature	Divide and conquer in nature
Insert operation	Easily inserted t end of list	Require processing to insert at its proper place to maintain a sorted list
Usefulness	Easy to use	Tricky algorithms
Lines of code	less	More

Conclusion

This paper covered the basics of data structures. With this we have only scratched the surface.

Although we have built a good foundation to move ahead. Data Structures is not just limited to Stack, Queues, and Linked Lists but is quite a vast area. There are many more data structures which include Maps, Hash Tables, Graphs, Trees, etc. Each data structure has its own advantages and disadvantages and must be used according to the needs of the application. A computer science student at least know the basic data structures along with the operations associated with them. Many high level and object oriented programming languages like C#, Java, Python come built in with many of these data structures. Therefore, it is important to know how things work under the hood. Dynamic data structures require dynamic storage allocation and reclamation. This may be accomplished by the programmer or may be done implicitly by a high-level language. It is important to understand the fundamentals of storage management because these techniques have significant impact on the behavior of programs. The basic idea is to keep a pool of memory elements that may be used to store components of dynamic data structures when needed. Allocated storage may be returned to the pool when no longer needed. In this way, it may be used and reused. This contrasts sharply with static allocation, in which storage is dedicated for the use of static data structures. It cannot then be reclaimed for other uses, even when no needed for the static data structure. As a result, dynamic allocation makes it possible to solve larger problems that might otherwise be storage-limited. Garbage collection and reference counters are two basic techniques for implementing storage management. Combinations of these techniques may also be designed. Explicit programmer control is also possible. Potential pitfalls of these techniques are garbage generation, dangling references, and fragmentation. High-level language may take most of the burden for storage management from the programmer. The concept of pointers or pointer variables underlies the use of these facilities, and complex algorithms are required for their implementation.

References

1. Book of Data structures through C G. S Baluja.
2. Pieren Garry Department of computer science New York University.
3. Paul Xavier department of algorithms in c Amsterdam.
4. Surendrakumar Ahuja IItdelhi department of computer science delhi .
5. Nick jones department of data mining Australia.
6. Wikipedia sequential search.