Data Structure: Theoretical Approach

Durgesh Raghuvanshi

May 19, 2022

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.

1 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.

2 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	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)

Table 1: Sequential search

3 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.

4 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.

5 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.

Figure 1: 3D view of Bee

6 Binary search

Binary search is a fast search algorithm with run-time complexity of (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 subarray 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).

7 Here is equation

Equation is $(a+b)^2 = a^2 + 2ab + b^2$

8 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.