1 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.

2 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.

3 Binary search tree

It is observed that BST's worst-case performance is closest to linear search algorithms, that is (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 RightLeft 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 RightLeft 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 leftright 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.

4 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 lastin, 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.

5 Comparison between linear search and binary search

Basis for comparison	Linear search	Binary search
Time complexity	O(N)	O(Log)
Best case time	First element 0(1)	Centre element 0(1)
Prerequisite for an array	Not required	Array must be sorted in order
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	summary
Usefulness	Easy to use	Tricky algorithms

6 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 or the use of static data structures.

7 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.

8 Picture

9 Equations

$$e = mc^2 \tag{1}$$

$$(c+d)^2 = c^2 + 2cd + d^2 \tag{2}$$

$$log(xy) = log(x) + log(y) \tag{3}$$