A study on usage of Data Structures in diverse aspects

Nekkanti Chakradhar

[a] Department of Computer Science and Engineering, Lovely Professional University Phagwara, Punjab 144402, India

ARTICLEINFO

Keywords
General-purpose Data Structures
Special purpose data structures
Sorting
Graphs
External Memory

ABSTRACT

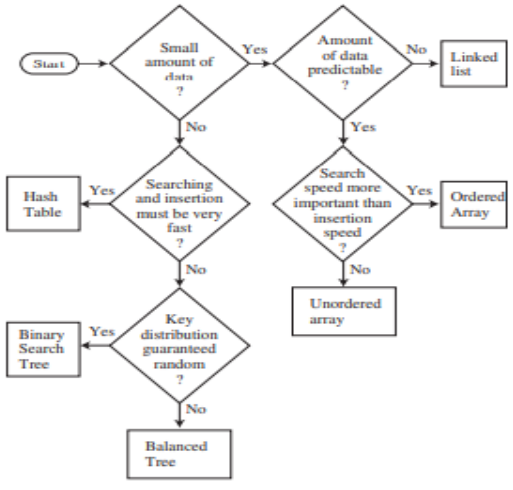
This paper tries to throw light in the usage of data structures in various aspects. Structuring the data is an eminent area of study which is gaining huge momentum as the need and urge for sharing and exploring information is growing day by day. In the field of computer science Data structures is being the most vital area of research for so many years. The need to have efficient data structures has become even more important as the data grows in an exponential nature and knowing where to use what gives a huge supremacy. Each data structure helps in data retrieval in so many ways our research focus on the situation particularity for the right usage of right data structure

1. Introduction

Data is essential for discovering and introducing new technologies. There is a huge amount of shareability of data now a days with the growing a lot of social networking media as well as updating of technology. The data which is processed is called as information. And finding all the usability ways for that data is done using a programming language in very easy and efficient manner. In every real-world situation, there will be a unique situation in this case we cannot give an exact answer for each and every problem. It will be very easy to implement data structures with prior knowledge. we can deal with huge amount of data and efficiently implement all the techniques like insertion, deletion, updating, retrieving the data, sorting and searching. We have two types of data structures which are general and specialized we can discuss in which situation which type of data structure is an exact fit.

2. General purpose Data Structures

If we want to store the real-world applicable data like for instance data records, inventory data, contact information or sales data, we have to implement general-purpose data structure. various data structures coming into these general-purpose data structures are arrays, linked lists, trees, and hash tables. data structures are mainly used for the storage of data and also helps in data retrieval using the key values. These implementations of the general-purpose database programs contradict with the specialized structures for instance stacks, queues as they allow the accessibility to only unique items of data. We can see which data structure is appropriate to use for the kind of problem in the below figure we can clearly see the relationships of the data structures. there will be many factors on the other hand and we will explore.



2.1 Libraries

Commercial data structure libraries are available for all major programming languages. Some structures may be built into the languages themselves. Vector, Stack, and Hashtable are examples of Java classes. The Standard Template Library (STL) in C++ comprises classes for a variety of data structures and algorithms.
Using a commercial library to generate the data structures described in this book may eliminate or at least reduce the amount of programming required. When this is the case, a sophisticated structure like a balanced tree or a delicate algorithm like rapid sort becomes a more appealing option. You must, however, guarantee that the lesson can be tailored to your specific needs.

2.2 Arrays

When it comes to storing and processing data, the array is often the first type of structure to examine. When the amount of data is relatively minimal and the volume of data is known ahead of time these situations arrays are very helpful. You can relax the second constraint if you have lots of memory; simply make the array large enough to process any incoming data. Use an unordered array if insertion speed is critical. Use an ordered array with a binary search if speed is crucial. In arrays, deletion is always slow since half of the items must be relocated to fill in the vacant cell. In an ordered array, traversal is quick, but it isn't possible in an unordered array. Vectors are arrays that grow when they become too full, such as the Vector class included with Java. When the amount of data is large enough, vectors may be useful.

2.3 Linked Lists

When the amount of data to be maintained cannot be specified advance or when data will be inserted and deleted regularly, consider a linked list. The linked list acquires whichever memory it requires when new elements are added, allowing it to extend to encompass all available memory; unlike arrays, there is no need to fill "holes" when deletion. In an unordered list, insertion is quick. Linked lists, like arrays, are usually best when the quantity of data is fairly small because searching and deletion are expensive however deletion is faster than in an array. A linked list is more difficult to implement than an array, but it is straightforward especially when compared to a tree or hash table.

2.4 Binary search trees

When arrays and linked lists are too slow, the first structure to consider is a binary tree. Insertion, searching, and deletion in a tree are all O(logN) fast. The maximum traversal for any data structure is O(N) (by definition, you must visit every item). One could also rapidly locate the minimum and maximum, as well as navigate among a variety of goods. Unbalanced binary trees are easier to program than balanced binary trees, however ordered data can slow them down to O(N) time, making them no faster as linked lists. A balanced tree, on the other hand, is useless if you know the

information would flow in randomized order.

2.5 Balanced Trees

We talk about red-black trees and 2-3-4 trees as examples of balanced trees. Both of these are balanced trees, ensuring O(logN) performance regardless of whether the input data is ordered or not. However, programming the balanced trees is complex, only red-black tree being the most demanding. They additionally incur extra memory overhead, that might or might not be noticeable. If a commercial class can be used for a tree, the problem of difficult programming may be alleviated. A hash table could be a best alternative than a balanced tree in some instances. When data is arranged, the performance of hash tables does not suffer. Various balanced trees exist, such as AVL trees, splay trees, 2-3 trees, and so forth, although they are not as popular as that of red-black tree.

2.6 Hash Tables

The quickest data storage structure is hash tables. As a result, they are required in circumstances where data is interacted with by a computer program rather than a human. Where a program must check thousands of words or symbols in a fraction of a second, hash tables are commonly employed in spelling checkers and as symbol tables in computer language compilers. When a person, rather than a computer, conducts data-access operations, hash tables may be valuable. Hash tables, as previously stated, are not affected by the order in which data is inserted, and hence can be used in place of a balanced tree. Programming is significantly easier for balanced trees than it is for unbalanced trees. Hash tables, particularly for open addressing, necessitate more memory. Because an array is utilized as the underlying structure, the amount of data to be saved must also be known rather precisely in advance. A hash table with distinct keys rather the quantity of data is known precisely in advance, in which case open addressing enables simpler programming because no linked list class is necessary, the most robust implementation is a hash table with separate chaining. Hash tables don't allow for ordered traversal or access to the minimum and maximum values. The binary search tree is a preferable alternative if these skills are important.

Comparison of General-Purpose Data Structures

In the below table we can give the summary of the General-Purpose Data Structures with the help of Big O notation in the aspect of speed

Data Structure	Search	Insertion	Deletion	Traversal
Array	O(N)	O(1)	O(N)	–
Ordered array	O(logN)	O(N)	O(N)	O(N)
Linked list	O(N)	O(1)	O(N)	–
Ordered linked list	O(N)	O(N)	O(N)	O(N)
Binary tree (average)	O(logN)	O(logN)	O(logN)	O(N)
Binary tree (worst case)	O(N)	O(N)	O(N)	O(N)
Balanced tree (average and worst case)	O(logN)	O(logN)	O(logN)	O(N)
Hash table	O(1)	O(1)	O(1)	–

In an unordered array, insertion is considered to occur only at last of the array. The ordered array utilizes a binary search, which really is quick, while insertion and deletion typically necessitate shifting half of the elements, that is slow. The term traversal refers to accessing data objects in ascending or descending order of keys; indicates that such a operation isn't really permitted.

3. Special purpose data Structures

The stack, queue, and priority queue are the special-purpose data structures Instead of providing a database for user-accessible data, these structures are typically utilized by a computer program to assist in the execution of certain algorithm. such as graph algorithms utilize stacks, queues, and priority queues. Stacks, queues, and priority queues all seem to be Abstract Data Types (ADTs) which are executed using the most fundamental structure like an array, linked list, or heap in the example of the priority queue). Such ADTs give the user a simple interface, usually allowing only one data item to be inserted and accessed or deleted.
In stacks the last item will be inserted
In queues the first item will be inserted
In priority queue the item with the highest priority.

3.1 Stacks

When you just want to retrieve the latest data item inserted, you use a stack; it's a Last In-First-Out (LIFO) structure. An array or a linked list is frequently used to implement a stack. And that the most recently inserted item is added to the end of the array, where it is also simple to delete, the array implementation is effective. Stack overflow can happen, although it's unlikely if the array is small enough, as stacks rarely carry large quantities of data. The linked list is a superior option than that of an array if a stack will hold a lot of information and the quantity can't be predicted properly in advance for instance when recursion is performed as stack. Because entries may be inserted and deleted fast from the list's head, a linked list is efficient. Stack overflow is not possible unless the whole memory is occupied. Because memory allocation is required to construct a new link for insertion and deallocation is required at some point when an item is removed from the list, a linked list is slightly slower than an array.

3.2 Queues

When you only require access to the first data item inserted, a queue is utilized; it's a First-In-First-Out (FIFO) structure. Queues could be implemented as arrays or linked lists, just as stacks. Both are really effective. To handle the circumstance where the queue loops around the outlet point of the array, more code is required. To allow for insertions for one endpoint and deletions at another, a linked list must really be double-ended. The option here between array implementation and a linked list implementation is dependent on how much the quantity of information can be estimated, same with stacks. When we knew how much data there would be, use an array; alternatively, if not the case uses linked list.

3.3 Priority queues

When only access to the data item with the highest priority is required, a priority queue is utilized. The item with the greatest or, in certain cases, the smallest key is this one. An ordered array or a heap can be used to build priority queues. Insertion into an ordered array takes a long time, but deletion takes only a few seconds. Insertion and deletion take O(logN) time with the heap implementation. If insertion speed isn't an issue, use an array or a double-ended linked list. When the amount of data to be saved can be predicted in advance, an array is used; when the amount of data is uncertain, a linked list is used. A heap is a superior alternative if speed is critical.

Comparison of Special-Purpose Data Structures

In the below table we can give the summary of the Special-purpose data structures with the help of Big O notation in the aspect of speed these special purpose data structures do not allow and support the searching and traversal.

Data Structure	Insertion	Deletion
Stack (array or linked list)	O(1)	O(1)
Queue (array or linked list)	O(1)	O(1)
Priority queue (ordered array)	O(N)	O(1)
Priority queue (heap)	O(logN)	O(logN)

- Stack(array or list) deletes the most recently inserted item.
- Queue(array or list) deletes the least recently inserted item.
- Priority queue(ordered array) deletes the highest priority item.
- Priority queue(heap) deletes the highest priority item.

4. Graphs

In the pantheon of data storage structures, graphs are unique. They don't store general-purpose data or serve as tools for programmers to use in other algorithms. Instead, they simulate real-world scenarios. The problem's structure is reflected in the graph's structure. Nothing else will suffice when a graph is required, so there is no need to decide when to use one. The most important decision is whether to use an adjacency matrix or adjacency lists to represent the graph. Your decision is based on whether the graph is dense, in which case the adjacency matrix is preferred, or dense, in which case the adjacency list is preferred. For adjacency matrix representation, depth-first search and breadth-first search take O(V2) time, where V is the number of vertices. For adjacency list representation, they run in O(V+E) time, where E is the number of edges. When using an adjacency matrix, minimum span ning trees and shortest paths take O(V2) time, while adjacency lists take O((E+V)logV) time. For your graph, you'll need to estimate V and E and do the math to figure out which representation is best.

5. Sorting

It's worth starting with a slow but simple sort, such as the insertion sort, just as it is with data structures. It's possible that the fast-processing speeds of today's computers will allow you to sort your data in a reasonable amount of time. On the off chance if there are fewer than 1,000 items, the slow sort would be adequate. Insertion sort is especially useful for almost-sorted files, as it runs in roughly O(N) time if there aren't too many out-of-place entries. When a few additional items are added to an already-sorted file, this is what happens. If the insertion sort proves to be too sluggish, the Shell sort becomes the next option. It's simple to set up and doesn't have a lot of quirks. It could be beneficial for up to 5,000 things, according to Sedgewick. Use one of the more sophisticated but faster sorts, such as merge sort, heapsort, or quicksort, only when the Shell sort becomes too slow. Merge sort necessitates additional memory, heapsort necessitates a heap data structure, and both are slower than quicksort, hence quicksort is the preferred method when the quickest sorting time is required. Quicksort, on the other hand, is suspect if there's a chance the data isn't random, in which case performance could degrade to O(N^2). Heapsort is preferable for potentially non-random data. If Quicksort is not done appropriately, it is also prone to subtle errors. Small coding errors can cause it to perform poorly for particular data configurations, a circumstance that can be difficult to discover. The running time for various sorting algorithms is summarized in below table. The Comparison column tries to evaluate minor speed differences between algorithms with similar average Big O timings. Shell sort isn't listed since no other algorithms with the same Big O performance exist.

Sort	Average	Worst	Comparison
Bubble	$O(N^2)$	$O(N^2)$	Poor
Selection	$O(N^2)$	$O(N^2)$	Fair
Insertion	$O(N^2)$	$O(N^2)$	Good
Shellsort	$O(N^{3/2})$	$O(N^{3/2})$	—
Quicksort	O(N*logN)	$O(N^2)$	Good
Mergesort	O(N*logN)	O(N*logN)	Fair
Heapsort	O(N*logN)	O(N*logN)	Fair

6. External storages

The main memory was used to store small chunks of data. However, data that is too large to fit in memory must be saved on external storage, which is usually in the form of disc files. Data is stored in a disc file in fixed-size units called blocks, each of which holds a number of records, we assumed. A record in a disc file is equivalent to an object in main memory in terms of data. A record, like an object, has a key value that is used to access it. We also assumed that reading and writing operations always involve a single block, and that these read and write operations take significantly longer than any data processing in main memory. As a result, the number of disc accesses must be kept to a minimum for quick operation.

6.1 Sequential Storage

When looking for a record with a specific key, the simplest option is to store them completely at random and read those chronologically. New records can easily be added to the file's end. Deleted records can be marked as such, or records can be shifted down to fill in the gap (as in an array). Because searching and deleting entails reading half of the blocks on average, sequential storage is slow, taking O(N) time. Even so, it might suffice for a small number of records.

6.2 Indexed files

When indexed files are used, performance improves considerably. In this scheme, main memory is used to store an index of keys and their matching block numbers. The index is used to find a record with a specific key. It provides the key's block number, and only one block needs to be read, which takes O(1) time. There are several indices that can be utilized, each with a distinct type of key (one for last names, one for Social Security numbers, and so on). This method works well until the index is too large to fit in memory. In most cases, index files are saved on disc and read into memory as needed. The downside of indexed files is that the index must be constructed at some time. Creating the index is likely to take a long time because it requires sequential reading of the file. When new entries are added to the file, the index will need to be updated as well.

6.3 Binary trees

Binary trees are multiway trees that correspond to blocks on the disc and are often employed in external storage. The algorithms work their way down the tree, reading one block at a time, just like in previous trees. B-trees allow for record finding, insertion, and deletion in O(logN) time. This is a quick method that works even with enormous files. The programming, on the other hand, is not simple.

6.4 Virtual Memory

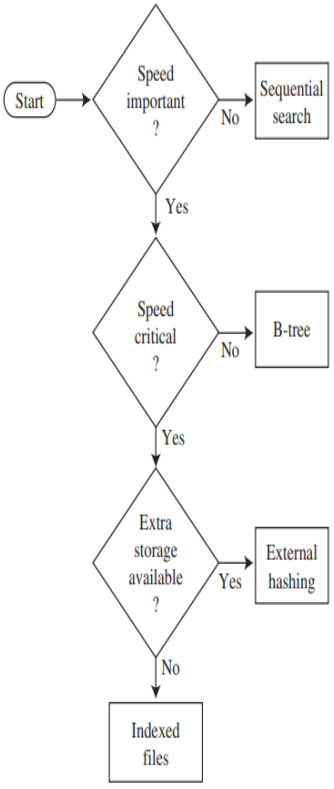
With a little programming work on your part, you can sometimes use your operating system's virtual memory capabilities (if it has them) to overcome disc access concerns. If you read a file that is too large for main memory, the virtual memory system will read only the parts of the file that fit and save the remainder on disc. Different parts of

the file will be read from the disc and put in memory when you access them. Internal algorithms can be applied to the full file as if it were all in memory at the same time, and the operating system can handle reading the necessary section of the file if it isn't already in memory. obviously, operations will be slower as when the entire file is in memory, but this will be true even if you worked with the file unit by block using one of the external-storage algorithms. It might be worth disregarding the fact that a file is too large to fit in memory and instead testing how well your algorithms operate with virtual memory. This may be a simple solution, especially for files that aren't much larger than the available capacity.

6.5 Hashing

External hashing may be a viable alternative if it's okay to utilise roughly twice as much external storage as a file would typically require. It takes the same amount of time to access files as indexed files, O(1), but it can handle larger files.

The below figure depicts these options for external storage structures in an impressionistic manner, relation of the external storage options



Conclusion:

We've completed our study of data structures and algorithms. The topic is vast and complicated. WHERE TO USE WHAT is our main aim of our study and we have succesfully achieved the point. With prior understanding, implementing data structures will be a breeze. We can handle large amounts of data and efficiently execute all data manipulation techniques. In the digital world, data structure is the best approach to organise information. The right way to use the right thing data structure is concluded.

References:

[1] From "Computer as Thinker/Doer: Problem-Solving Environments for Computational Science" by S. Gallopoulos, E. Houstis and J. Rice (IEEE Computational Science and Engineering, Summer 1994).
[2] J. Stasko, A. Badre, C. Lewis, Do Algorithm Animations Assist Learning? An Empirical Study and Analysis, Proceedings of the ITERCHI Conference
[3] R.M. Felder, L.K. Silverman, Learning Styles and Teaching Styles in Engineering Education, J. Engr. Education, 78, 1988, pp.674—681.
[4] R.M. Felder, R. Brent, Understanding Student Differences, J. Engr. Education, 94, 2005, pp. 57—72.