GPCET
Pioneering Innovative Education

G. PULLAIAH COLLEGE OF ENGINEERING AND TECHNOLOGY

Accredited by NAAC with 'A' Grade of UGC, Approved by AICTE, New Delhi
Permanently Affiliated to JNTUA, Ananthapuramu
(Recognized by UGC under 2(f) and 12(B) & ISO 9001:2008 Certified Institution)
Nandikotkur Road, Venkayapalli, Kurnool – 518452

Department of Computer Science and Engineering

Bridge Course on Data Structures

By

B. Venkateswarlu

Data Structures

Data Definition defines a particular data with the following characteristics.

- **Atomic** − Definition should define a single concept.
- **Traceable** − Definition should be able to be mapped to some data element.
- **Accurate** − Definition should be unambiguous.
- **Clear and Concise** − Definition should be understandable.

Data Object

Data Object represents an object having a data.

Data Type

Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. There are two data types −

- Built-in Data Type
- Derived Data Type

Built-in Data Type

Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provide the following built-in data types.

- Integers
- Boolean (true, false)
- Floating (Decimal numbers)
- Character and Strings

Derived Data Type

Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built-in data types and associated operations on them. For example −

- List
- Array
- Stack
- Queue

Basic Operations

The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

- Traversing
- Searching
- Insertion
- Deletion
- Sorting
- Merging

Before introducing data structures we should understand that computers do store, retrieve, and process a large amount of data. If the data is stored in well organized way on storage media and in computer's memory then it can be accessed quickly for processing that further reduces the latency and the user is provided fast response.

Data structure introduction refers to a scheme for organizing data, or in other words a data structure is an arrangement of data in computer's memory in such a way that it could make the data quickly available to the processor for required calculations. A data structure should be seen as a logical concept that must address two fundamental concerns. First, how the data will be stored, and second, what operations will be performed on it? As data structure is a scheme for data organization so the functional definition of a data structure should be independent of its implementation. The functional definition of a data structure is known as ADT (Abstract Data Type) which is independent of implementation. The implementation part is left on developers who decide which technology better suits to their project needs.

For example, a stack ADT is a structure which supports operations such as push and pop. A stack can be implemented in a number of ways, for example using an array or using a linked list.

Along with data structures introduction, in real life, problem solving is done with help of data structures and algorithms. An algorithm is a step by step process to solve a problem. In programming, algorithms are implemented in form of methods or functions or routines. To get a problem solved we not only want algorithm but also an efficient algorithm. One criteria of efficiency is time taken by the algorithm, another could be the memory it takes at run time.

Sometimes, we can have more than one algorithm for the same problem to process a data structure, and we have to choose the best one among available algorithms. This is done by algorithm analysis. The best algorithm is the one which has a fine balance between time taken and memory consumption. But, as we know the best exists rarely, and we generally give more priority to the time taken by the algorithm rather than the memory it consumes. Also, as memory is getting cheaper and computers have more memory today than previous time, therefore, run time analysis becomes more significant than memory. The analysis of algorithms is an entirely separate topic and we will discuss that separately.

Classification Data Structures

Data structures can be broadly classified in two categories - *linear structures* and *hierarchical structures*. Arrays, linked lists, stacks, and queues are linear structures, while trees, graphs, heaps etc. are hierarchical structures.

Every data structure has its own strengths, and weaknesses. Also, every data structure specially suits to specific problem types depending upon the operations performed and the data organization. For example, an array is suitable for read operations. Following is a quick introduction to important data structures.

Arrays

Arrays are statically implemented data structures by some programming languages like C and C++; hence the size of this data structure must be known at compile time and cannot be altered at run time. But modern programming languages, for example, Java implements arrays as objects and give the programmer a way to alter the size of them at run time. Arrays are the most common data structure used to store data.

Arrays are unarguably easier data structures to use and access. But inserting an item to an array and deleting it from the array are situation dependent. If you want to insert an item at a particular position which is already occupied by some element then you have to shift all items one position right from the position new element has to be inserted then insert the new item. The time taken by insert operation is depend on how big the array is, and at which position the item is being inserted. The same is true about deletion of an item.

If the array is unsorted then search operation is also proved costly and takes O(N) time in worst case, where N is size of the array. But if the array is sorted then search performance is improved magically and takes O(logN) time in worst case.

Linked List

Linked list data structure provides better memory management than arrays. Because linked list is allocated memory at run time, so, there is no waste of memory. Performance wise linked list is slower than array because there is no direct access to linked list elements.

Linked list is proved to be a useful data structure when the number of elements to be stored is not known ahead of time.

There are many flavours of linked list you will see: linear, circular, doubly, and doubly circular.

Stack

Stack is a last-in-first-out strategy data structure; this means that the element stored in last will be removed first. Stack has specific but very useful applications; some of them are as follows:

- Solving Recursion - recursive calls are placed onto a stack, and removed from there once they are processed.
- Evaluating post-fix expressions
- Solving Towers of Hanoi
- Backtracking
- Depth-first search
- Converting a decimal number into a binary number

Queue

Queue is a first-in-first-out data structure. The element that is added to the queue data structure first, will be removed from the queue first. Desuetude, priority queue, and circular queue are the variants of queue data structure. Queue has the following application uses:

- Access to shared resources (e.g., printer)
- Multiprogramming
- Message queue

Trees

Tree is a hierarchical data structure. The very top element of a tree is called the root of the tree. Except the root element every element in a tree has a parent element, and zero or more children elements. All elements in the left sub-tree come before the root in sorting order, and all those in the right sub-tree come after the root.

Tree is the most useful data structure when you have hierarchical information to store. For example, directory structure of a file system; there are many variants of tree you will come across. Some of them are Red-black tree, threaded binary tree, AVL tree, etc.

Heap

Heap is a binary tree that stores a collection of keys by satisfying heap property. Max heap and min heap are two flavours of heap data structure. The heap property for max heap is: each node should be greater than or equal to each of its children. While, for min heap it is: each node should be smaller than or equal to each of its children. Heap data structure is usually used to implement priority queues.

Dictionary

Dictionary is a data structure that maintains a set of items indexed on basis of keys. Dictionary stores data in form of key-element pairs.

Hash Table

Hash Table is again a data structure that stores data in form of key-element pairs. A key is a non-null value which is mapped to an element. And, the element is accessed on the basis of the key associated with it. Hash table is a useful data structure for implementing dictionary.

Graph

Graph is a networked data structure that connects a collection of nodes called vertices, by connections, called edges. An edge can be seen as a path or communication link between two nodes. These edges can be either directed or undirected. If a path is directed then you can move in one direction only, while in an undirected path the movement is possible in both directions.

Last Word

In this tutorial we saw a brief introduction of various important data structures. We also talked of what data structures are, and why data structures so much important for information systems. Data structures along with algorithms are a core subject of computer science. Hope you have enjoyed reading this tutorial. Please do write us if you have any suggestion/comment or come across any error on this page. Thanks for reading!

In computer science, an abstract data type (ADT) is a mathematical model for data types, where a data type is defined by its behaviour (semantics) from the point of view of a *user* of the data, specifically in terms of possible values, possible operations on data of this type, and the behaviour of these operations. This contrasts with data structures, which are concrete representations of data, and are the point of view of an implementer, not a user.

Formally, an ADT may be defined as a "class of objects whose logical behaviour is defined by a set of values and a set of operations";[1] this is analogous to an algebraic structure in mathematics. What is meant by "behaviour" varies by author, with the two main types of formal specifications for behaviour being *axiomatic (algebraic) specification* and an *abstract model;*[2]these correspond to axiomatic semantics and operational semantics of an abstract machine, respectively. Some authors also include the computational complexity ("cost"), both in terms of time (for computing operations) and space (for representing values). In practice many common data types are not ADTs, as the abstraction is not perfect, and users must be aware of issues like arithmetic overflow that are due to the representation. For example, integers are often stored as fixed width values (32-bit or 64-bit binary numbers), and thus experience integer overflow if the maximum value is exceeded.

ADTs are a theoretical concept in computer science, used in the design and analysis of algorithms, data structures, and software systems, and do not correspond to specific features of computer languages—mainstream computer languages do not directly support formally specified ADTs. However, various language features correspond to certain aspects of ADTs, and are easily confused with ADTs proper; these include abstract types, opaque data types, protocols, and design by contract. ADTs were first proposed by Barbara Loskop and Stephen N. Zilles in 1974, as part of the development of the CLU language.

Introduction to Data Structures

Data Structure is a way of collecting and organising data in such a way that we can perform operations on these data in an effective way. Data Structures is about rendering data elements in terms of some relationship, for better organization and storage. For example, we have data player's name "Virat" and age 26. Here "Virat" is of **String** data type and 26 is of **integer** data type.

We can organize this data as a record like **Player** record. Now we can collect and store player's records in a file or database as a data structure. For example: "Dhoni" 30, "Gambhir" 31, "Sehwag" 33

In simple language, Data Structures are structures programmed to store ordered data, so that various operations can be performed on it easily.

Basic types of Data Structures

As we discussed above, anything that can store data can be called as a data strucure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as Primitive Data Structures.

Then we also have some complex Data Structures, which are used to store large and connected data. Some example of Abstract Data Structure are :

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more

details in our later lessons.

What is Algorithm

An algorithm is a finite set of instructions or logic, written in order, to accomplish a certain predefined task. Algorithm is not the complete code or program, it is just the core logic(solution) of a problem, which can be expressed either as an informal high level description as pseudocode or using a flowchart.

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties :

1. Time Complexity
2. Space Complexity

Space Complexity

Its the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

An algorithm generally requires space for following components :

- **Instruction Space :** Its the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
- **Data Space :** Its the space required to store all the constants and variables value.
- **Environment Space :** Its the space required to store the environment information needed to resume the suspended function.

Time Complexity

Time Complexity is a way to represent the amount of time needed by the program to run to completion. We will study this in details in our section.

NOTE: Before going deep into data structure, you should have a good knowledge of programming either in C or in C++ or Java.

Time Complexity of Algorithms

Time complexity of an algorithm signifies the total time required by the program to run to completion. The time complexity of algorithms is most commonly expressed using the big O notation.

Time Complexity is most commonly estimated by counting the number of elementary functions performed by the algorithm. And since the algorithm's performance may vary with different types of input data, hence for an algorithm we usually use the worst-case Time complexity of an algorithm because that is the maximum time taken for any input size.

Calculating Time Complexity

Now lets tap onto the next big topic related to Time complexity, which is How to Calculate Time Complexity. It becomes very confusing some times, but we will try to explain it in the simplest way.

Now the most common metric for calculating time complexity is Big O notation. This removes all constant factors so that the running time can be estimated in relation to N, as N approaches infinity. In general you can think of it like this :

```
statement;
```

Above we have a single statement. Its Time Complexity will be Constant. The running time of the statement will not change in relation to N.

```
for(i=0; i < N; i++)
{
  statement;
}
```

The time complexity for the above algorithm will be **Linear**. The running time of the loop is directly proportional to N. When N doubles, so does the running time.

```
for(i=0; i < N; i++)
{
  for(j=0; j < N;j++)
  {
    statement;
  }
}
```

This time, the time complexity for the above code will be Quadratic. The running time of the two loops is proportional to the square of N. When N doubles, the running time increases by N * N.

```
while(low <= high)
{
  mid = (low + high) / 2;
  if (target < list[mid])
    high = mid - 1;
  else if (target > list[mid])
    low = mid + 1;
  else break;
}
```

This is an algorithm to break a set of numbers into halves, to search a particular field(we will study this in detail later). Now, this algorithm will have a **Logarithmic** Time Complexity. The running time of the algorithm is proportional to the number of times N can be divided by 2(N is high-low here). This is because the algorithm divides the working area in half with each iteration.

```
void quicksort(int list[], int left, int right)
{
  int pivot = partition(list, left, right);
  quicksort(list, left, pivot - 1);
  quicksort(list, pivot + 1, right);
}
```

Taking the previous algorithm forward, above we have a small logic of Quick Sort(we will study this in detail later). Now in Quick Sort, we divide the list into halves every time, but we repeat the iteration N times(where N is the size of list). Hence time complexity will be **N*log(N)**. The running time consists of N loops (iterative or recursive) that are logarithmic, thus the algorithm is a combination of linear and logarithmic.

NOTE : In general, doing something with every item in one dimension is linear, doing something with every item in two dimensions is quadratic, and dividing the working area in half is logarithmic.

Types of Notations for Time Complexity

Now we will discuss and understand the various notations used for Time Complexity.

1. **Big Oh** denotes "*fewer than or the same as*" <expression> iterations.
2. **Big Omega** denotes "*more than or the same as*" <expression> iterations.
3. **Big Theta** denotes "*the same as*" <expression> iterations.
4. **Little Oh** denotes "*fewer than*" <expression> iterations.
5. **Little Omega** denotes "*more than*" <expression> iterations.

Understanding Notations of Time Complexity with Example

O(expression) is the set of functions that grow slower than or at the same rate as expression.

Omega(expression) is the set of functions that grow faster than or at the same rate as expression.

Theta(expression) consist of all the functions that lie in both O(expression) and Omega(expression).

Suppose you've calculated that an algorithm takes f(n) operations, where,

f(n) = 3*n^2 + 2*n + 4.   // n^2 means square of n

Since this polynomial grows at the same rate as n^2, then you could say that the function **f** lies in the set Theta(n^2). (It also lies in the sets O(n^2) and Omega(n^2) for the same reason.)

The simplest explanation is, because Theta denotes *the same as* the expression. Hence, as f(n) grows by a factor of n^2, the time complexity can be best represented as Theta(n^2).