MODERN TENDENCY IN MODULAR PROGRAMMING

KENJAYEV SANJAR SOBIROVICH

Assistant-professor of Samarkand State University named after Sharaf Rashidov

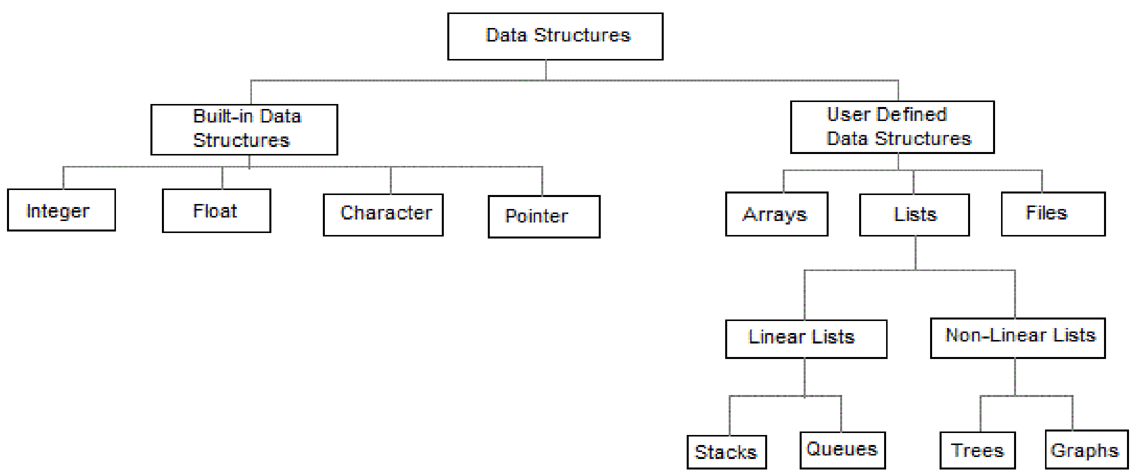
Abstract: *As such, an algorithm must be precise enough to be understood by human beings. However, in order to be executedby acomputer, we will generally need a program that is written in a rigorous formal language; and since computers are quite in flexible compared to the human mind, programs usually need to contain more details than algorithms.*

Key words: *Modul, Modular programming, Binary Tree, Binary Search Tree, Heap, Hashing Data Structure, Matrix and Trie.*

Introduction

- A **data structure** is a particular way of organizing data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks.
- The choice of good data structure makes it possible to perform a variety of critical operations effectively. An efficient data structure uses minimum memory space and execution time to process the structure as well.
- **Data structures are used in various field such as:**
- Operating system
- Graphics
- Computer Design

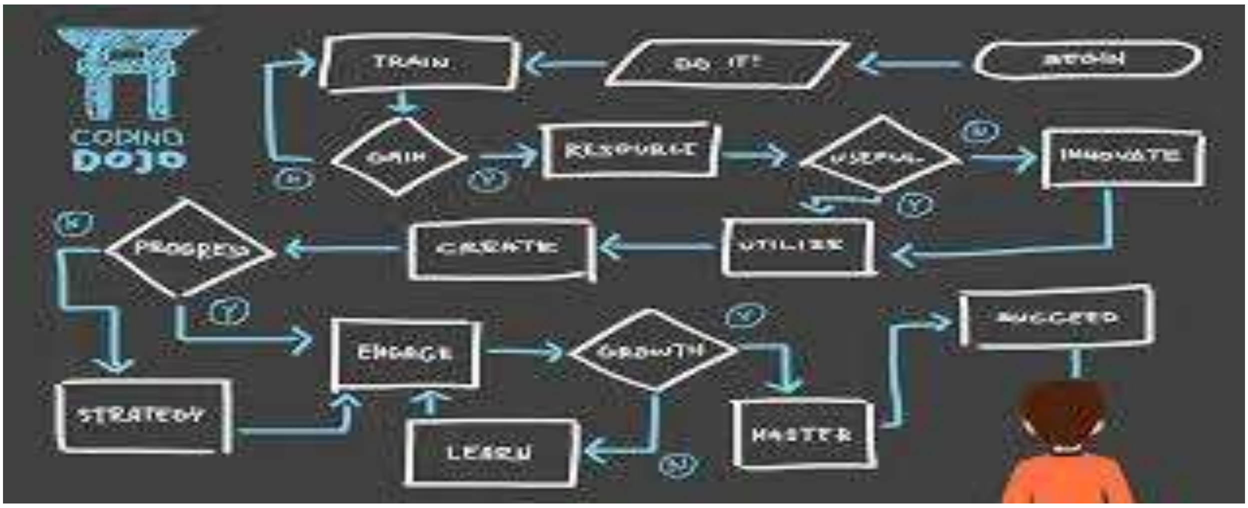
Simulation etc



Web of Scholars: Multidimensional Research Journal (MRJ)
Volume: 01 Issue: 06 | 2022 ISNN: (2751-7543)
http://innosci.org/

Algorithms as opposed to programs

- An **algorithm** for a particular task can be defened as “a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time".
- As such, an algorithm must be precise enough to be understood by human beings. However, in order to be executedby acomputer, we will generally need a program that is written in a rigorous formal language; and since computers are quite in flexible compared to the human mind, programs usually need to contain more details than algorithms.
- Here we shall ignore most of those programming details and concentrate on the design of algorithms rather than programs.



Imperative and Declarative Programming

- The task of implementing the discussed algorithms as computer programs is important, of course, but we will concentrate on the theoretical aspects and leave the practical programming aspects to be studied elsewhere.
- Having said that, we will often and it useful to write down segments of actual programs in order to clarify and test certain theoretical aspects of algorithms and their data structures.
- It is also worth bearing in mind the distinction between different programming paradigms: **Imperative Programming** describes computation in terms of instructions that change the program/data state, where as **Declarative Programming** species what the program should accomplish without describing how to do it. We will primarily be concerned with developing algorithms that map easily onto the imperative programming approach.

Pseudocode

- Algorithms can obviously be described in plain English, and we will sometimes do that.
- However, for computer scientists it is usually easier and clearer to use something that comes somewhere in between formatted English and computer program code, but is not runnable because certain details are omitted.
- This is called **pseudocode**, which comes in a variety of forms. We will present segments of pseudocode that are very similar to the languages we are mainly interested in, namely the

overlap of C and Java, with the advantage that they can easily be inserted into runnable programs.

Fundamental questions about algorithms

- Given an algorithm to solve a particular problem, we are naturally led to ask:
- 1. What is it supposed to do?
- 2. Does it really do what it is supposed to do?
- 3. How eficiently does it do it?
- The technical terms normally used for these three aspects are:
- 1. Specification.
- 2. Verification.
- 3. Performance analysis.

The details of these three aspects will usually be rather problem dependent

Specification

- The **specification** should formalize the crucial details of the problem that the algorithm is intended to solve. Sometimes that will be based on a particular representation of the associated data, and sometimes it will be presented more abstractly.
- Typically, it will have to specify how the inputs and outputs of the algorithm are related, though there is no general requirement that the specification is complete or non-ambiguous.
- For simple problems, it is often easy to see that a particular algorithm will always work, i.e. that it satisfies its specification.
- However, for more complicated specifications and/or algorithms, the fact that an algorithm satisfies its specification may not be obvious at all.

  The reason is that we want to concentrate on the data structures and algorithms. Formal verification techniques are complex and will normally be left till after the basic ideas have been studied.

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(including temporary 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 required by the program to run till its completion.
- It's generally a good practice to try to keep the time required minimum, so that our algorithm completes it's execution in the minimum time possible.

Space Complexity of Algorithms

- Whenever a solution to a problem is written some memory is required to complete. For any algorithm memory may be used for the following:
- Variables (This include the constant values, temporary values)
- Program Instruction
- Execution
- Sometime **Auxiliary Space** is confused with Space Complexity. But Auxiliary Space is the extra space or the temporary space used by the algorithm during it's execution.
- **Space Complexity** = **Auxiliary Space + Input space**

Memory Usage while Execution

- While executing, algorithm uses memory space for three reasons:
- **Instruction Space**
- It's the amount of memory used to save the compiled version of instructions.
- **Environmental Stack**
- Sometimes an algorithm(function) may be called inside another algorithm(function). In such a situation, the current variables are pushed onto the system stack, where they wait for further execution and then the call to the inside algorithm(function) is made.
- For example, If a function A() calls function B() inside it, then all th variables of the function A() will get stored on the system stack temporarily, while the function B() is called and executed inside the funciton A().
- **Data Space**
- Amount of space used by the variables and constants.

Static Data Structure vs Dynamic Data Structure

- **Static Data structure** has fixed memory size whereas **in Dynamic Data Structure**, the size can be randomly updated during run time which may be considered efficient with respect to memory complexity of the code.
- Static Data Structure provides more easier access to elements with respect to dynamic data structure.
- Unlike static data structures, dynamic data structures are flexible.
- **Different approach to design an algorithm**
- **1. Top-Down Approach:** A top-down approach starts with identifying major components of system or program decomposing them into their lower level components & iterating until desired level of module complexity is achieved . In this we start with topmost module & incrementally add modules that is calls.

Conclusion. In general, **testing** on a few particular inputs can be enough to show that the algorithm is incorrect. However, since the number of different potential inputs for most algorithms is infinite in theory, and huge in practice, more than just testing on particular cases is needed to be sure that the algorithm satisfies its specification. We need correctness proofs. Although we will discuss proofs, and useful relevant ideas like invariants, we will usually only do so in a rather informal manner (though, of course, we will attempt to be rigorous).

REFERENCES

1. D. Marr, Vision: a computational investigation into the human representation and processing of visual information. W. H. WH San Francisco: Freeman and Company.1982 .
2. B. Meyer, Object-oriented software construction. New York: Prentice hall, 1988
3. H. Topi, J. S. Valacich, R. T. Wright, K. M. Kaiser, J. F Nunamaker Jr, J. C Sipior, & G. J. De Vreede, Curriculum guidelines for undergraduate degree programs in information systems. ACM/AIS task force, 2010.
4. O. A. Aljohani, R. J. Qureshi, "Proposal to Decrease Code Defects to Improve Software Quality", International Journal of Information Engineering and Electronic Business (IJIEEB), vol.8, no.5, pp.44-51, 2016. DOI: 10.5815/ijieeb.2016.05.06
5. B.S. Mitchell and S. Mancoridis, “Modeling the Search Landscapeof Metaheuristic Software Clustering Algorithms,” Proc. Geneticand Evolutionary Computation Conf., 2003.