FEDERAL UNIVERSITY OF TECHNOLOGY, MINNA
NIGER STATE, NIGERIA

CENTRE FOR OPEN DISTANCE AND e-LEARNING (CODeL)

B.TECH. COMPUTER SCIENCE PROGRAMME

COURSE TITLE

ALGORITHMS

COURSE CODE

CPT 223

COURSE CODE

CPT 223

COURSE UNIT

3

Course Coordinator

Bashir MOHAMMED (Ph.D.)

Department of Computer Science

Federal University of Technology (FUT) Minna

Minna, Niger State, Nigeria.

Course Development Team

CPT 223: ALGORITHMS

Subject Matter Experts	O.A ABISOYE (PhD) FUT Minna, Nigeria.
Course Coordinator	Bashir MOHAMMED (Ph.D.) Department of Computer Science FUT Minna, Nigeria.
Instructional Designers	Oluwole Caleb FALODE (Ph.D.) Bushrah Temitope OJOYE (Mrs.) Centre for Open Distance & e-Learning, FUT Minna, Nigeria
ODL Experts	Amosa Isiaka GAMBARI (Ph.D.) Nicholas E. ESEZOBOR
Language Editors	Chinenye Priscilla UZOCHUKWU (Mrs.) Mubarak Jamiu ALABEDE
Centre Director	Abiodun Musa AIBINU (Ph.D.) Centre for Open Distance & e-Learning FUT Minna, Nigeria.

CPT 223 Study Guide

Introduction

CPT 223: Algorithms is a 3-credit unit course for students studying towards acquiring a bachelor of science in computer science and other related disciplines. The course is divided into 4 module and 7 study units. It will first take a brief review of the Algorithm and Problem Solving. This course will then go ahead to deal with the different algorithm techniques, the algorithm analysis and recursion. The course went further to explain the concepts of Graph and tree. The course also explains the concept of Software Engineering.

Course Guide

The course guide introduces to you what you will learn in this course and how to make the best use of the material. It brings to your notice the general guidelines on how to navigate through the course and on the expected actions you have to take for you to complete this course successfully. Also, the guide will hint you on how to respond to your Self-Assessment Exercise(s) and Tutor-Marked Assignments. Therefore, this course guide provides for you general idea of what the course CPT 213 is all about. The references provided will also help you. The course guide therefore gives you an overview of what the course. CPT 223 is all about, the textbooks and other materials to be reference, what you expect to know in each unit, and how to work through the course material.

What You Will Learn in This Course

The overall aim of this course, CPT 323 is to introduce you to basic concepts of algorithms in order to enable you to understand the basic rudiments of algorithms computation being use in core Computer Science and especially its major role in programming design.

In this course of your studies, you will be put through the kinds of procedure to compute a specific algorithm.

Course Aim

This course aim to introduce student to the basic concept of Algorithms and appreciate decision processes especially for critical and cost involving (life, money, etc) endeavours. It will help the reader to understand the level of disparity between outcome and realty and why we require an algorithm solve every computational problem.

Course Objectives

It is important to note that each unit has specific objective. Student should study them carefully before proceeding to subsequently unit. Therefore, it may be useful to refer to these objectives in the course of your study of the unit to assess your progress. You should always look at the unit objective after completing a unit. In this way, you can be sure that you have done what is required of you by the end of the unit.

However, below are overall objective of this course. On completing this course, you should be able to:

1. learn about problem solving skills;
2. learn about algorithm development;
3. the concept and properties of an algorithm;
4. representation of Algorithm;
5. learn the need of algorithm techniques;
6. basic Algorithm Paradigms;
7. see the need for the running analysis of an algorithm;
8. explain the complexity of algorithm;
9. explain three direction of time analysis;
10. discuss time and space analysis;
11. the need for amortized analysis;
12. the concept of recursion;
13. implementation of recursion and its relation;
14. recursive specialization of mathematical function such as Fibonacci and Factorial;
15. simple Recursive Procedures (Towers of Hanoi, permutations);
16. recursive backtracking;
17. perform representation of a tree;
18. define Operations on Tree;
19. perform Tree Traverse and know the terms used in traverse;
20. perform representation of a Graph;
21. define Operations on Graph;
22. perform Graph Traverse;
23. explain the need of Software Engineering;
24. explain the Applications of Software Engineering;
25. explain the need for project management;
26. list the Steps for managing software projects; and
27. explain how to ensure effective project management.

Working Through This Course

To complete this course, you are required to study all the units, the recommended text books, and other relevant materials. Each unit contain some self assessment exercises and tutor assignments, and at some point, in these courses, you required to submit the tutor marked assignments. There is also final examination at the end of these courses. Stated below are the component of these courses and what you have to do.

Course Materials

1. Course Guide
2. Study Units
3. Text Books
4. Assignment Files
5. Presentation Schedule

Study Unit

There are 10 study unit and 5 modules in this course, they are:

MODULE 1: ALGORITHM DESIGN

UNIT 1: Introduction to Algorithm

UNIT 2: Algorithm Technique

MODULE 2: ALGORITHM ANALYSIS

UNIT 1: Time and Space Complexity

UNIT 2: Recursion

MODULE 3: TREES AND GRAPH

UNIT 1: Trees

UNIT 2: Graph

MODULE 4: SOFTWARE ENGINEERING

UNIT 1: Concepts of Software Engineering

Recommended Texts

http://ww3.algorithmdesign.net/ch00-front.html

http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=805047.

http://www.cs.berkeley.edu/~luca/cs172/karp.pdf.

http://www-cs-faculty.stanford.edu/~knuth/gkp.html.

http://www.htdp.org/2003-09-26/Book/curriculum-Z-H-31.html.

http://www.saasblogs.com/2006/09/15/how-to-rewrite-standard-recursion-through-a-state-stack-amp-iteration/

http://www.refactoring.com/catalog/replaceRecursionWithIteration.html

http://www.ccs.neu.edu/home/shivers/papers/loop.pdf

http://lambda-the-ultimate.org/node/1014

http://docs.python.org/library/sys.html

http://ScottOnWriting.NET.

"http://en.wikipedia.org/w/index.php?title=Graph_(abstract_data_type)&oldid=508723425

http://www.swebok.org.

Assignment File

The assignment file will be given to you in due course. In this file you will find all the detail of the work you must submit to your tutor for marks for marking. The marks you obtain for these assignments will count toward the final mark for the course. Altogether, there are tutor marked assignments for this course.

Presentation Schedule

The presentation schedule included in this course guide provides you with importance date for completion of each tutor marked assignment. You should therefore endeavour to meet the deadline.

Assignment

There are two aspects to the assessment of this course. First there are tutor marked assignments: and second the written examination. Therefore, you are expected to take note of the fact information and problem solving gathered during the course. The tutor marked assignment must be submitted to your tutor for formal assessments in accordance to the deadline given. The work submitted will count for 40%of your total course mark. At the end of the course you will need to sit for a final written examination. This examination will account for 60%of your total score.

Tutor Marked Assignment (TMA)

There are TMAs in this course; you need to submit all the TMAs. The best 10 will therefore be counted. When you have completed each assignment, send them to your tutor as soon as possible and make certain that it gets to your tutor on or before the stipulated deadline. If for any reason you cannot complete your assignment on time, contact your tutor before the assignment is due to discuss the possible of extension. Extension will not be granted after the deadline unless on extraordinary cases.

Final Examination And Grading

Final examination for CPT 223 will last with period of 2hours and has a value of 60% of the total course grade. The examination will consist of questions which reflect the self assessment exercise and tutor marked assignment s that you have previously encountered. Furthermore, all areas of the course will be examined. It would be better to use the time between finishing the last unit and sitting for the examination, to revise the entire course. You might find it useful to review your TMAs and comment on them before the examination. The final examination covers information from all parts of the course.

The Following Are Practical Strategies For Working Through This Course.

1. Read the course guide thoroughly.
2. Organize a study schedule. Refer to the course overview for more details. Note the time you are expected to spend on each unit and how the assignment relates to the units. Important details, e.g. details of your tutorials and the date of the first day of the semester are available. You need to gather all these information in one place such as a diary, a wall chart calendar or an organizer. Whatever method you choose, you should decide on and write in your own dates for working on each unit.
3. Once you have created your own study schedule, do everything you can to stick to it. The major reason that students fail that they get behind with their course works. If you get into difficulties with your schedule, please let your tutor know before it is too late for help.
4. Turn to unit 1 and read the introduction and the objectives for the unit.
5. Assemble the study materials. Information about what you need for a unit is given in the table of content at the beginning of each unit. You will almost always need both the study unit you are working on and one of the materials recommended for further readings, on your desk at the same time.
6. Work through the unit, the content of the unit itself has been arranged to provide a sequence for you to follow. As you work through the unit, you will be encouraged to read from your set books.
7. Keep in mind that you will learn a lot by doing all your assignments carefully. They have been designed to help you meet the objectives of the course and will help you pas the examination.
8. Review the objectives of each study unit to confirm that you have achieved them. If you are not certain about any of the objectives, review the study material and consult your tutor.
9. When you are confident that you have achieved a unit’s objectives, you can start on the next unit. Proceed unit by unit through the course and try to place your study so that you can keep yourself on schedule.

10. When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. When the assignment is returned, pay particular attention to your tutor's comments, both on the tutor marked assignment form and also written on the assignment. Consult your tutor as soon as possible if you have and questions or problems.
11. After completing the last unit, review the course and prepare yourself for the final examination. Check that you have achieved the unit objectives (listed at the beginning of each unit) ant the course objectives (listed in this course guide).

Tutors and Tutorials

There are 8 hours of tutorial provide in support of this course. You will be notified of the dates, time and location together with the name and phone number of your tutor as soon as you are allocated a tutorial group. Your tour tutor will mark and comment on your assignments, keep a close watch on your progress and on any difficulties, you might encounter and provide assistance to you during the course. You must mail your tutor marked assignment to your tutor well before the due date. At least two working days are required for this purpose. They will be marked by your tutor and returned to you as soon as possible.

Don't hesitate to contact your tutor by telephone, e-mail or discussion board if you need help. The following might be circumstances in which you would find help necessary:

1. you don't not understand any part of the study units or the assigned readings;
2. you have difficulty with the self test or exercise;
3. you have questions or problems with an assignment, with your tutor's comments on an assignment or with the grading of an assignment.

You should endeavor to attend the tutorials. This is the only opportunity to have face to face contract with your tutor and ask questions which are answered instantly. You can raise any problem encountered in the course of your study. To gain the maximum benefit from the course tutorials, have some questions handy before attending them. You will learn a lot from participating actively in discussions.

Good luck!

Table of Contents

Course Development Team……ii

CPT 223 Study Guide……...iii

Table of Content……ix

MODULE 1: ALGORITHM DESIGN……..1

UNIT 1: Introduction to Algorithm……2

UNIT 2: Algorithm Technique……11

MODULE 2: ALGORITHM ANALYSIS……38

UNIT 1: Time and Space Complexity……39

UNIT 2: Recursion……..47

MODULE 3: TREES AND GRAPH……..71

UNIT 1: Trees……72

UNIT 2: Graph……..……100

MODULE 4: SOFTWARE ENGINEERING……..119

UNIT 1: Concepts of Software Engineering……..120

Module 1

Introduction to Algorithms

Unit 1: Algorithm and Problem Solving

Unit 2: Algorithm Techniques

Unit 1

Algorithm and Problem Solving

Contents

1.0 Introduction

2.0 Learning Outcomes

3.0 Learning Contents

- 3.1 Problem Solving
- 3.2 Algorithm
- 3.3 Problem Solving Process
- 3.4 The concept and properties of an algorithm
- 3.5 Representation of Algorithm
- 3.6 Control Structures

4.0 Conclusion

5.0 Summary

6.0 Tutor Marked Assignment (TMA)

7.0 References/ Further Reading

1.0 Introduction

Computing science is already having a major influence on our problem-solving skills, amounting to a revolution in the art of effective reasoning. Because of the challenges of programming (which means instructing a dumb machine how to solve each instance of a problem) and the unprecedented scale of programming problems, computing scientists have had to hone their problem-solving skills to a very fine degree. This has led to advances in logic, and to changes in the way that mathematics is practiced. This course note forms an introduction to problem-solving using the insights that have been gained in computing science.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

i. learn about problem solving skills;
ii. explore the algorithmic approach for problem solving;
iii. learn about algorithm development;
iv. become aware of problem solving process;
v. the concept and properties of an algorithm; and
vi. representation of Algorithm.

3.0 Learning Contents

3.1 Problem Solving

Programming is a process of problem solving. The Problem-solving techniques are:

1. Analyze the problem
2. Outline the problem requirements
3. Design steps (algorithm) to solve the problem

3.2 Algorithm

An Algorithm is a Step-by-step problem-solving process. It is a solution achieved in finite amount of time. An algorithm is the idea behind the computer program. Solutions to programming problems are formulated as so-called algorithms. An algorithm is a well-defined procedure, consisting of a number of instructions that are executed in turn in order to solve the given problem.

Normally, an algorithm will have certain inputs; for each input, the algorithm should compute an output which is related to the input by a certain so-called input-output relation. Formulating an algorithm makes problem-solving decidedly harder, because it is necessary to formulate very clearly and precisely the procedure for solving the problem. The more general the problem, the harder it gets.

The advantage, however, is a much greater understanding of the solution. The process of formulating an algorithm demands a full understanding of why the algorithm is correct. It is independent of the kind of hardware it is running on or which programming language it is written in and Solves a well-specified problem in a general way. It is specified by describing the set of instances (input) it must work on and describing the desired properties of the output.

Before a computer can perform a task, it must have an algorithm that tells it what to do. Informally: “An algorithm is a set of steps that define how a task is performed.” Formally: “An algorithm is an ordered set of unambiguous executable steps, defining a terminating process.”

Ordered set of steps: structure!

Executable steps: doable!

Unambiguous steps: follow the directions!

Terminating: must have an end!

Self-Assessment Exercise

1. State the essentiality of algorithm in program design.

Self-Assessment Answer

An algorithm is the idea behind the computer program. Solutions to programming problems are formulated as so-called algorithms. An algorithm is the problem-solving skill for the challenges of programming. It is independent of the kind of hardware it is running on or which programming language it is written in and Solves a well-specified problem in a general way.

3.3 Problem Solving Process

Step 1 - Analyze the problem

i. Outline the problem and its requirements:
ii. Thoroughly understand the problem and problem requirements
iii. Does program require user interaction?
iv. Does program manipulate data?
v. What is the output?
vi. If the problem is complex, divide it into subproblems
vii. Analyze each sub problem as above
viii. Design steps (algorithm) to solve the problem

Step 2 - Implement the algorithm

i. Implement the algorithm in code

ii. Verify that the algorithm works

Step 3 - Maintenance

Use and modify the program if the problem domain changes

In algorithmic problem solving, we deal with objects. Objects are data manipulated by the algorithm. To a cook, the objects are the various types of vegetables, meat and sauce. In algorithms, the data are numbers, words, lists, files, and so on.

In solving a geometry problem, the data can be the length of a rectangle, the area of a circle, etc. Algorithm provides the logic; data provide the values. They go hand in hand. Hence, we have this great truth:

Program = Algorithm + Data Structures

Data structures refer to the types of data used and how the data are organised in the program. Data come in different forms and types. Most programming languages provides simple data types such as integers, real numbers and characters, and more complex data structures such as arrays, records and files which are collections of data.

Because algorithm manipulates data, we need to store the data objects into variables, and give these variables names for reference. For example, in mathematics, we call the area of a circle *A*, and express *A* in terms of the radius *r*. (In programming, we would use more telling variable names such as *area* and *radius* instead of *A* and *r* in general, for the sake of readability.)

When the program is run, each variable occupies some memory location(s), whose size depends on the data type of the variable, to hold its value.

Self-Assessment Exercise

1. Explain the processes involve in solving a Problem

Self-Assessment Answer

Step 1 - Analyze the problem

i. Outline the problem and its requirements:

ii. Thoroughly understand the problem and problem requirements

iii. Does program require user interaction?

iv. Does program manipulate data?

v. What is the output?

vi. If the problem is complex, divide it into sub-problems

vii. Analyze each sub problem as above

viii. Design steps (algorithm) to solve the problem

Step 2 - Implement the algorithm

i. Implement the algorithm in code

ii. Verify that the algorithm works

Step 3 - Maintenance

Use and modify the program if the problem domain changes

3.4 The Concept and Properties of an Algorithm Correct/ Effective

An algorithm should always return the desired output for all legal instances of the problem. Again, this goes without saying. An algorithm must provide the correct answer to the problem.

Unambiguous: This means that it must solve every instance of the problem. For example, a program that computes the area of a rectangle should work on all possible dimensions of the rectangle, within the limits of the programming language and the machine.

An algorithm should also emphasised on the *whats*, and not the *hows*, leaving the details for the program version. However, this point is more apparent in more complicated algorithms at advanced level, which we are unlikely to encounter yet.

Precise/Exact: Each step of an algorithm must be exact. This goes without saying. An algorithm must be precisely and unambiguously described, so that there remains no uncertainty. An instruction that says “shuffle the deck of card” may make sense to some of us, but the machine will not have a clue on how to execute it, unless the detail steps are described. An instruction that says “lift the restriction” will cause much puzzlement even to the human readers.

Efficient: Can be measured in terms of:

1. Time
2. Space

Time tends to be more important

An algorithm must terminate: The ultimate purpose of an algorithm is to solve a problem. If the program does not stop when executed, we will not be able to get any result from it. Therefore, an algorithm must contain a finite number of steps in its execution. Note that an algorithm that merely contains a finite number of steps may not terminate during execution, due to the presence of ‘infinite loop’.

Self Assessment Exercise

1. Discuss the properties of a good algorithm

Self-Assessment Answer

Correct/ Effective: An algorithm should always returns the desired output for all legal instances of the problem. Again, this goes without saying. An algorithm must provide the correct answer to the problem.

Unambiguous: This means that it must solve every instance of the problem. For example, a program that computes the area of a rectangle should work on all possible dimensions of the rectangle, within the limits of the programming language and the machine.

Precise / exact: Each step of an algorithm must be exact. This goes without saying. An algorithm must be precisely and unambiguously described, so that there remains no uncertainty.

Efficient: Can be measured in terms of

i. Time

ii. Space

Time tends to be more important

An algorithm must terminate: The ultimate purpose of an algorithm is to solve a problem. If the program does not stop when executed, we will not be able to get any result from it. Therefore, an algorithm must contain a finite number of steps in its execution. Note that an algorithm that merely contains a finite number of steps may not terminate during execution, due to the presence of 'infinite loop'.

3.5 Representation of Algorithm

Pseudo-Codes and Flowcharts

We usually present algorithms in the form of some *pseudo-code*, which is normally a mixture of English statements, some mathematical notations, and selected keywords from a programming language. There is no standard convention for writing pseudo-code; each author may have his own style, as long as clarity is ensured.

Below are two versions of the same algorithm, one is written mainly in English, and the other in pseudo-code. The problem concerned is to find the minimum, maximum, and average of a list of numbers. Make a comparison.

Algorithm version 1

First, you initialize *sum* to zero, *min* to a very big number, and *max* to a very small number.

Then, you enter the numbers, one by one.

For each number that you have entered, assign it to *num* and add it to the *sum*.

At the same time, you compare *num* with *min*, if *num* is smaller than *min*, let *min* be *num* instead.

Similarly, you compare *num* with *max*, if *num* is larger than *max*, let *max* be *num* instead.

After all the numbers have been entered, you divide *sum* by the numbers of items entered, and let *ave* be this result.

End of algorithm.

Algorithms may also be represented by diagrams. One popular diagrammatic method is the *flowchart*, which consists of terminator boxes, process boxes, and decision boxes, with flows of logic indicated by arrows.

3.6 Control Structures

The pseudo-code and flowchart in the previous section illustrate the three types of *control structures*. They are:

1 Sequence

2 Branching (Selection)

3 Loop (Repetition)

These three control structures are sufficient for all purposes. The sequence is exemplified by sequence of statements place one after the other – the one above or before another gets executed first. In flowcharts, sequence of statements is usually contained in the rectangular process box.

The *branch* refers to a binary decision based on some condition. If the condition is true, one of the two branches is explored; if the condition is false, the other alternative is taken.

This is usually represented by the ‘if-then’ construct in pseudo-codes and programs. In flowcharts, this is represented by the diamond-shaped decision box. This structure is also known as the *selection* structure.

The *loop* allows a statement or a sequence of statements to be repeatedly executed based on some loop condition. It is represented by the ‘while’ and ‘for’ constructs in most programming languages, for unbounded loops and bounded loops respectively. (Unbounded loops refer to those whose number of iterations depends on the eventuality that the termination condition is satisfied; bounded loops refer to those whose number of iterations is known before-hand.)

In the flowcharts, a back arrow hints the presence of a loop. A trip around the loop is known as iteration. You must ensure that the condition for the termination of the looping must be satisfied after some finite number of iterations, otherwise it ends up as an infinite loop, a common mistake made by inexperienced programmers. The loop is also known as the *repetition* structure.

Combining the use of these control structures, for example, a loop within a loop (nested loops), a branch within another branch (nested if), a branch within a loop, a loop within a branch, and so forth, is not uncommon. Complex algorithms may have more complicated logic structure and deep level of nesting, in which case it is best to demarcate parts of the algorithm as separate smaller *modules*. Beginners must train themselves to be proficient in using and combining control structures appropriately and go through the trouble of tracing through the algorithm before they convert it into code.

4.0 Conclusion

An algorithm is a set of instructions, and an algorithmic problem lends itself to a solution expressible in algorithmic form. Algorithms manipulate data, which are represented as variables of the appropriate data types in programs. Data structures are collections of data.

The characteristics of algorithms are presented in this chapter, so are the two forms of representation for algorithms, namely, pseudo-codes and flowcharts. The three control structures: sequence, branch, and loop, which provide the flow of control in an algorithm, are introduced.

5.0 Summary

In this Module 1Study Unit 1, the following aspects have been discussed:

i. Programming is a process of problem solving.

ii. An Algorithm is a Step-by-step problem-solving process.

iii. The Concept and Properties of an Algorithm

iv. Algorithm are represented in Pseudo-Codes and Flowcharts

v. The three control structures: sequence, branch, and loop, provide the flow of control in an algorithm

6.0 Tutor Marked Assignments and Marking Scheme

A. Write an algorithm to compute the area of a circle

B. Draw the flowcharts of the question above.

C. Write an algorithm and draw the flowcharts to compute the perimeter of a cylinder

D. What do you understand by the word “Control Structure”?

7.0 References/ Further Reading

Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms* (MIT Press, 2000).

Brassard, G. and Bratley, P. Fundamental of Algorithmics, Prentice-Hall, 1996.

Anany V. Levitin, *Introduction to the Design and Analysis of Algorithms* (Addison Wesley, 2002).

Donald E. Knuth, *The Art of Computer Programming: Volume 3, Sorting and Searching*, second edition (Addison-Wesley, 1998).

Heideman, M. T., D. H. Johnson, and C. S. Burrus, "Gauss and the history of the fast Fourier transform," IEEE ASSP Magazine, 1, (4), 14–21 (1984)

Unit 2

Algorithm Techniques

Contents

1.0 Introduction

2.0 Learning Outcomes

3.0 Learning Contents

- 3.1 Algorithm Paradigm
  - 3.1.1 Divide and Conquer algorithm
  - 3.1.2 Dynamic Programming
  - 3.1.3 Greedy Algorithm
  - 3.1.4 Backtracking

4.0 Conclusion

5.0 Summary

6.0 Tutor Marked Assignments (TMAs)

7.0 References/ Further Reading

1.0 Introduction

Problem solving is an essential part of every scientific discipline. It has two components: (1) problem identification and formulation, and (2) solution of the formulated problem. One can solve a problem on its own using ad hoc techniques or follow those techniques that have produced efficient solutions to similar problems.

This requires the understanding of various algorithm design techniques, how and when to use them to formulate solutions and the context appropriate for each of them. This Module 1 study unit 2 advocates the study of algorithm design techniques by presenting most of the useful algorithm design techniques and illustrating them through numerous examples.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

1. Learn the need of algorithm techniques;
2. Basic Algorithm Paradigms.

3.0 Learning Contents

3.1 Algorithmic Paradigms

This is the General approaches to the construction of *efficient* solutions to problems. They provide templates suited to solving a broad range of diverse problems. Algorithmic Paradigms can be translated into common control and data structures provided by most high-level languages. Then the temporal and spatial requirements of the algorithms which result can be precisely analyzed.

Although more than one technique may be applicable to a specific problem, it is often the case that an algorithm constructed by one approach is clearly superior to equivalent solutions built using alternative techniques.

The choice of design paradigm is an important aspect of algorithm synthesis. These paradigms are discussed below:

3.1.1 Divide and Conquer algorithm

This is a method of designing algorithms that (informally) proceeds as follows:

Given an instance of the problem to be solved, split this into several, smaller, sub-instances (*of the same problem*) independently solve each of the sub-instances and then combine the sub-instance solutions so as to yield a solution for the original instance. This description raises the question:

By what methods are the *sub-instances* to be *independently solved*?

The answer to this question is central to the concept of *Divide-&-Conquer algorithm* and is a key factor in gauging their efficiency.

Consider the following: We have an algorithm, *alpha* say, which is known to solve all problem instances of size *n* in at most *c n^2* steps (where *c* is some constant). We then discover an algorithm, *beta* say, which solves the same problem by:

Dividing an instance into 3 sub-instances of size *n/2*.

Solves these 3 sub-instances.

Combines the three sub-solutions taking *d n* steps to do this.

Suppose our original algorithm, *alpha*, is used to carry out the `solves these sub-instances' step 2. Let

T(alpha)(n) = Running time of alpha

T(beta)(n) = Running time of beta

Then,

T(alpha)(n) = c n^2 (by definition of alpha)

But

T(beta)(n) = 3 T(alpha)(n/2) + d n

= (3/4)(cn^2) + dn

So if *dn < (cn^2)/4* (i.e. *d < cn/4*) then *beta* is *faster than alpha*

In particular for all large enough *n*, (*n > 4d/c = Constant*), *beta* is faster than *alpha*.

This realisation of *beta* improves upon *alpha* by just a constant factor. But if the problem size, *n*, is large enough then

n > 4d/c

n/2 > 4d/c

...

n/2^i > 4d/c

which suggests that using *beta* instead of *alpha* for the `*solves these*' stage *repeatedly until the sub-sub-sub..sub-instances are of size n0 < = (4d/c)* will yield a still faster algorithm.

So consider the following new algorithm for instances of size *n*

procedure *gamma* (*n* : problem size) is

```
  begin
    if n <= n^-0 then
       Solve problem using Algorithm alpha;
     else
       Split into 3 sub-instances of size n/2;
```

Use *gamma* to solve each sub-instance;

Combine the 3 sub-solutions;

end if;

end *gamma*;

Let *T(gamma)(n)* denote the running time of this algorithm.

cn^2 *if n < = n0*

T(gamma)(n) =

3T(gamma)(n/2)+dn *otherwise*

We shall show how relations of this form can be estimated later in the course. With these methods it can be shown that

T(gamma)(n) = O(n^{log3}) (=O(n^{1.59..})

This is an *asymptotic improvement* upon algorithms *alpha* and *beta*.

The improvement that results from applying algorithm *gamma* is due to the fact that it maximises the savings achieved *beta*.

The (relatively) inefficient method *alpha* is applied only to *"small"* problem sizes.

The precise form of a divide-and-conquer algorithm is characterised by:

The *threshold* input size, *n0*, below which the problem size is not sub-divided.

The *size* of sub-instances into which an instance is split.

The *number* of such sub-instances.

The algorithm used to combine sub-solutions.

In (II) it is more usual to consider the *ratio* of initial problem size to sub-instance size. In our example this was *2*. The *threshold* in (I) is sometimes called the *(recursive) base value*. In summary, the generic form of a divide-and-conquer algorithm is:

procedure *D-and-C* (*n* : input size) is

begin

if *n* < = *n0* then

Solve problem without further

sub-division;

else

Split into *r* sub-instances

each of size *n/k*;

for each of the *r* sub-instances do

D-and-C (n/k);

Combine the *r* resulting

sub-solutions to produce

the solution to the original problem;

end if;

end *D-and-C*;

Such algorithms are naturally and easily realised as:

A directory has a set of *names* and a telephone *number* associated with each name. The directory is sorted by alphabetical order of names. It contains *n* entries which are stored in 2 arrays: *names (1..n)* ; *numbers (1..n)*.Given a *name* and the value n find the *number* associated with the name.

Self Assessment Exercise

1. A directory has a set of *names* and a telephone *number* associated with each name. The directory is sorted by alphabetical order of names. It contains *n* entries which are stored in 2 arrays: *names (1..n)* ; *numbers (1..n)*.Given a *name* and the value n find the *number* associated with the name.

Self-Assessment Answer

Answer

We assume that any given input name actually *does occur* in the directory, in order to make the exposition easier. The Divide-&-Conquer algorithm (Binary Search) to solve this problem is the simplest example of the paradigm.

It is based on the following observation

Given a name, *X* say, *X* occurs in the *middle* place of the *names* array Or *X* occurs in the *first* half of the *names* array. (U) Or *X* occurs in the *second* half of the *names* array. (L)

U (respectively *L*) are true *only if X* comes *before* (respectively *after*) that name stored in the *middle* place.

This observation leads to the following algorithm:

function *search* (*X* : name;

start, finish : integer)

return integer is

middle : integer;

begin

middle := *(start+finish)/2*;

```
    if names(middle)=x then
        return numbers(middle);
    else if X<names(middle) then
        return search(X,start,middle-1);
    else -- X>names(middle)
        return search(X,middle+1,finish);
    end if;
end search;
```

3.1.2 Dynamic Programming

This paradigm is most often applied in the construction of algorithms to solve a certain class of *Optimisation Problem* That is: problems which require the *minimisation* or *maximisation* of some measure.

Dynamic programming solution is rather more involved than the recursive Divide-and-Conquer method, nevertheless its running time is practical. One disadvantage of using Divide-and-Conquer is that the process of recursively solving separate sub-instances can result in the same computations being performed repeatedly since *identical* sub-instances may arise.

The idea behind *dynamic programming* is to avoid this pathology by obviating the requirement to calculate the same quantity twice.

The method usually accomplishes this by maintaining a *table of sub-instance results*.

The binomial coefficient example illustrates the key features of dynamic programming algorithms.

A table of all sub-instance results is constructed.

The entries corresponding to the smallest sub-instances are initiated at the start of the algorithm.

The remaining entries are filled in following a precise order (that corresponds to increasing sub-instance size) using only those entries that have already been computed.

Each entry is calculated exactly once.

The final value computed is the solution to the initial problem instance.

Implementation is by iteration (*never* by recursion, even though the analysis of a problem may naturally suggest a recursive solution).

Bottom-Up Technique

Dynamic Programming is a Bottom-Up Technique in which the smallest sub-instances are *explicitly* solved first and the results of these used to construct solutions to progressively larger sub-instances.

Top-Down Technique

In contrast, Divide-and-Conquer is a Top-Down Technique which *logically* progresses from the initial instance down to the smallest sub-instances via intermediate sub-instances. We can illustrate these points by considering the problem of calculating the *Binomial Coefficient*, *"n choose k"*, i.e.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad (0 \le k \le n)$$

It is straightforward to show that

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0 \text{ or } k = n \end{cases}$$

Using this relationship, a rather crude Divide-and-Conquer solution to the problem of calculating the Binomial Coefficient *`n choose k'* would be:

```
function bin_coeff (n : integer;
                    k : integer)
                    return integer is
  begin
    if k = 0 or k = n then
      return 1;
    else
      return
      bincoeff(n-1, k-1) + bincoeff(n-1, k);
    end if;
  end bin_coeff;
```

By contrast, the Dynamic Programming approach uses the same relationship but constructs a table of all the (n+1)*(k+1) binomial coefficients *`i choose j'* for each value of *i* between *0* and *n*, each value of *j* between *0* and *k*.

These are calculated in a particular order:

First the table entries corresponding to the coefficients `i choose 0' and `1 choose 1' are fixed to the value *1*.

The remaining table entries corresponding to the binomial coefficient `i choose j' are calculated in *increasing order of the value of i+j*.

It should be noted that since the coefficient `*i choose j*' requires only the values of `*i-1 choose j-1*' and `*i-1 choose j*', computing the table entries in the order of increasing *i+j* ensures that the table entries needed for ``*i choose j*' have already been calculated, i.e.

(i-1)+(j-1) < (i-1)+j < i+j

The Dynamic Programming method is given by:

```
function bin_coeff (n : integer;
                    k : integer)
                    return integer is
type table is array (0..n, 0..k) of integer;
bc : table;
i, j, k : integer;
sum : integer;
begin
  for i in 0..n loop
    bc(i,0) := 1;
  end loop;
  bc(1,1) := 1;
  sum := 3; i := 2; j := 1;
  while sum <= n+k loop
    bc(i,j) := bc(i-1,j-1)+bc(i,j-1);
    i := i-1; j := j+1;
    if i < j or j > k then
      sum := sum + 1;
      if sum <= n+1 then
        i := sum-1; j := 1;
      else
        i := n; j := sum-n;
      end if;
```

```
      end if;
    end loop;
  return bc(n,k);
end bin_coeff;
```

The section of the function consisting of the lines:

```
if i < j or j > k then
  sum := sum + 1;
  if sum <= n+1 then
    i := sum-1; j := 1;
  else
    i := n; j := sum-n;
  end if;
end if;
```

is invoked when all the table entries `*i choose j*', for which *i+j* equals the current value of *sum*, have been found. The if statement increments the value of *sum* and sets up the new values of *i* and *j*.

Now consider the differences between the two methods: The Divide-and-Conquer approach recomputes values, such as *"2 choose 1"*, a very large number of times, particularly if *n* is large and *k* depends on *n*, i.e. *k* is not a constant.

It can be shown that the running time of this method is

$$\Omega\left(\binom{n}{k}\right)$$

In the worst-case ($k = n/2$) this is, asymptotically, $\Omega(2^n/n)$.

Despite the fact that the algorithm description is quite simple (it is just a direct implementation of the relationship given) it is completely infeasible as a practical algorithm.

The Dynamic Programming method, since it computes each value *"i choose j"* exactly once is far more efficient. Its running time is *O(n*k)*, which is *O(n^2)* in the worst-case, (again *k = n/2*).

Example: Shortest Path

Input: A directed graph, *G(V, E)*, with nodes

V = {1, 2 ,..., n }

and edges *E* as subset of *VxV*. Each edge in *E* has associated with it a non-negative length.

Output: An *nXn* matrix, *D*, in which *D^(i,j)* contains the *length* of the *shortest path* from node *i* to node *j* in *G*.

Informal Overview of Method

The algorithm, conceptually, constructs a *sequence of matrices*:

D0, D1 ,..., Dk ,..., Dn

For each *k* (with *1 < = k < = n*), the *(i, j)* entry of *Dk*, denoted *Dk(i,j)*, will contain the Length of the shortest path from node i to node j when only the nodes

{ 1, 2, 3 ,..., k } can be used as intermediate nodes on the path.

Obviously *Dn = D*.

The matrix, *D0*, corresponds to the `*smallest sub-instance*'. *D0* is initiated as:

0 if i=j

D0(i, j) = infinite if (i,j) not in E

Length(i,j) if (i,j) is in E

Now, suppose we have constructed *Dk*, for some *k < n*. How do we proceed to build *D(k+1)*?

The shortest path from *i* to *j* with only *{ 1, 2, 3 ,..., k, k+1 }* available as *internal nodes*

Either: *Does not* contain the node *k+1*.

Or: *Does* contain the node *k+1*.

In the former case:

D(k+1)(i, j) = Dk(i, j)

In the latter case:

D(k+1)(i, j) = D-k(i, k+1) + D-k(k+1, j)

Therefore *D(k+1)(i, j)* is given by

Dk(i,j)

minimum

Dk(i, k+1) + Dk(k+1, j)

Although these relationships suggest using a recursive algorithm, as with the previous example, such a realisation would be extremely inefficient.

Instead an *iterative* algorithm is employed.

Only *one nXn* matrix, *D*, is needed.

This is because after the matrix *D(k+1)* has been constructed, the matrix *Dk* is no longer needed. Therefore *D(k+1)* can overwrite *Dk*.

In the implementation below, *L* denotes the matrix of *edge lengths* for the set of edges in the graph *G(V, E)*.

```
type matrix is array (1..n, 1..n) of integer;
L : matrix
function shortest_path_length (L : matrix;
                               n : integer)
                               return matrix is
D : matrix; -- Shortest paths matrix
begin
  -- Initial sub-instance
  D(1..n,1..n) := L(1..n,1..n);
  for k in 1..n loop
    for i in 1..n loop
      for j in 1..n loop
        if D(i,j) > D(i,k) + D(k,j) then
          D( i,j ) := D(i,k) + D(k,j);
        end if;
      end loop;
    end loop;
  end loop;
  return D(1..n,1..n);
end shortest_path_length;
```

This algorithm, discovered by Floyd, clearly runs in time

O(n^3)

Thus *O(n)* steps are used to compute each of the *n^2* matrix entries.

Self Assessment Exercise

1. Explain the concept of Dynamic Programming?

Self-Assessment Answer

The idea behind *dynamic programming* is to avoid this pathology of Divide and conquer algorithm which recursively solving separate sub-instances and thereby result into the same computations being performed repeatedly since *identical* sub-instances may arise. By obviating the requirement to calculate the same quantity twice.

The method usually accomplishes this by maintaining a *table of sub-instance results*.

The binomial coefficient example illustrates the key features of dynamic programming algorithms.

A table of all sub-instance results is constructed.

The entries corresponding to the smallest sub-instances are initiated at the start of the algorithm.

The remaining entries are filled in following a precise order (that corresponds to increasing sub-instance size) using only those entries that have already been computed.

Each entry is calculated exactly once.

The final value computed is the solution to the initial problem instance.

Implementation is by iteration (*never* by recursion, even though the analysis of a problem may naturally suggest a recursive solution).

3.1.3 Greedy Algorithms

This is another approach that is often used to design algorithms for solving Optimization Problems. In contrast to dynamic programming, however, Greedy algorithms do not always yield a genuinely optimal solution. In such cases the greedy method is frequently the basis of a *heuristic approach*.

Even for problems which can be solved exactly by a greedy algorithm, establishing the correctness of the method may be a non-trivial process.

In order to give a precise description of the greedy paradigm we must first consider a more detailed definition of the environment in which typical optimization problems occur. Thus, in an optimization problem, one will have, in the context of greedy algorithms, the following:

1. A collection (set, list, etc) of candidates, e.g. nodes, edges in a graph, etc.
2. A set of candidates which have already been `used';
3. A predicate (*solution*) to test whether a given set of candidates give a *solution* (not necessarily optimal);

4. A predicate (*feasible*) to test if a set of candidates can be extended to a (not necessarily optimal) solution;
5. A selection function (*select*) which chooses some candidate which h as not yet been used; and
6. An objective function which assigns a *value* to a solution.

In other words: An optimization problem involves finding a subset, *S*, from a collection of candidates, *C*; the subset, *S*, must satisfy some specified criteria, i.e. be a solution and be such that the *objective function* is optimised by *S*. `*Optimised'* may mean

Minimised or Maximised Depending on the precise problem being solved. Greedy methods are distinguished by the fact that the selection function assigns a *numerical value* to each candidate, *x*, and chooses that candidate for which:

SELECT(x) is largest

or *SELECT(x)* is smallest

All Greedy Algorithms have exactly the same general form. A Greedy Algorithm for a particular problem is specified by describing the predicates `*solution*' and `*feasible*'; and the selection function `*select*'.

Consequently, Greedy Algorithms are often very easy to design for optimisation problems.

The General Form of a Greedy Algorithm is

function *select* (*C* : candidate_set) return candidate;

function *solution* (*S* : candidate_set) return

boolean;

function *feasible* (*S* : candidate_set) return

boolean;

--**********

function *greedy* (*C* : candidate set) return candidate set is

x : candidate;

S : candidate set;

begin

S := {};

while (not *solution(S)*) and *C /= {}* loop

x := select(C);

C := C - {x};

if *feasible(S union {x})* then

```
      S := S union { x };
    end if;
  end loop;
  if solution( S ) then
    return S;
  else
    return es;
  end if;
end greedy;
```

As illustrative examples of the greedy paradigm we shall describe algorithms for the following problems:

Minimal Spanning Tree.

Integer Knapsack.

For the first of these, the algorithm always returns an optimal solution.

Minimal Spanning Tree

The inputs for this problem is an (undirected) graph, *G(V, E)* in which each edge, *e* in *E*, has an associated positive edge length, denoted *Length(e)*.

The output is a spanning tree, *T(V, F)* of *G(V,E)* such that the *total edge length*, is minimal amongst all the possible spanning trees of *G(V,E)*.

Note: An *n*-node tree, *T* is a *connected n*-node graph with *exactly n-1 edges*.

T(V,F) is a spanning tree of *G(V,E)* if and only if *T* is a tree and the edges in *F* are a subset of the edges in *E*.

In terms of general template given previously:

The candidates are the edges of G(V,E).

A subset of edges, *S*, is a solution if the graph *T(V,S)* is a spanning tree of *G(V,E)*.

A subset of edges, *S*, is feasible if there is a spanning tree *T(V,H)* of *G(V,E)* for which *S sube H*.

The objective function which is to be minimised is the sum of the edge lengths in a solution.

The select function chooses the candidate (i.e. edge) whose length is smallest (from the remaining candidates).

The full algorithm, discovered by Kruskal, is:

```
function min_spanning_tree (E : edge_set)
                        return edge_set is
```

S : edge_set;

e : edge;

begin

S := (es;

while (*H(V,S)* not a tree)

and *E /= {}* loop

e := Shortest edge in *E*;

E := E - {e};

if *H(V, S union {e})* is acyclic then

S := S union {e};

end if;

end loop;

return *S*;

end *min_spanning_tree*;

Before proving the correctness of this algorithm, we give an example of it running.

The algorithm may be viewed as dividing the set of nodes, *V*, into *n* parts or *components*:

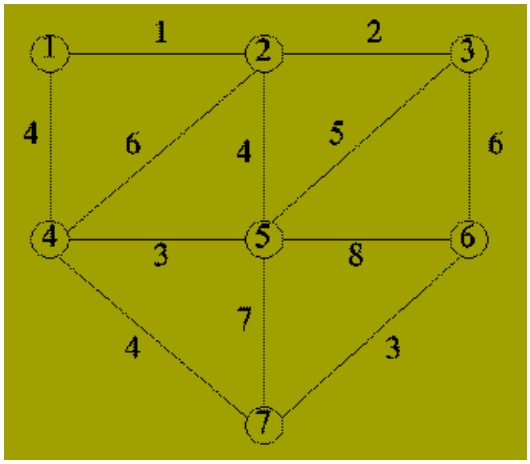
{1} ; {2} ; ... ; {n}

An edge is added to the set *S* if and only if it joins two nodes which belong to *different* components; if an edge is added to *S* then the two components containing its endpoints are coalesced into a single component.

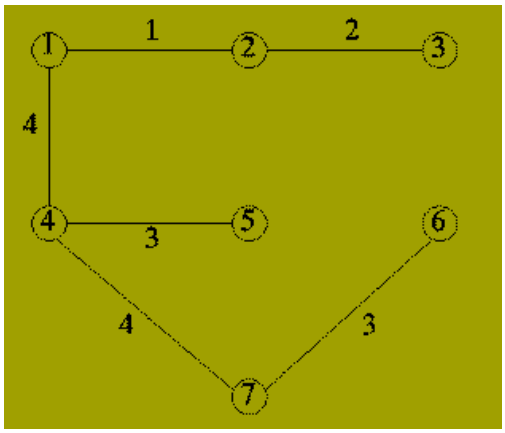
In this way, the algorithm stops when there is just a single component

{ 1, 2 ,..., n }

remaining.



Iteration	Edge	Components
0	-	{1}; {2}; {3}; {4}; {5}; {6}; {7}
1	{1,2}	{1,2}; {3}; {4}; {5}; {6}; {7}
2	{2,3}	{1,2,3}; {4}; {5}; {6}; {7}
3	{4,5}	{1,2,3}; {4,5}; {6}; {7}
4	{6,7}	{1,2,3}; {4,5}; {6,7}
5	{1,4}	{1,2,3,4,5}; {6,7}
6	{2,5}	Not included (adds cycle)
7	{4,7}	{1,2,3,4,5,6,7}



Question: How do we know that the resulting set of edges form a Minimal Spanning Tree?

In order to prove this, we need the following result.

For *G(V,E)* as before, a subset, *F*, of the edges *E* is called *promising* if *F* is a subset of the edges in a minimal spanning tree of *G(V,E)*.

Lemma: Let *G(V,E)* be as before and *W* be a subset of *V*.

Let *F*, a subset of *E* be a promising set of edges such that no edges in *F* has *exactly one* endpoint in *W*.

If *{p,q}* in *E-F* is a shortest edge having exactly one of *p* or *q* in *W* then: the set of edges *F union { {p,q} }* is promising.

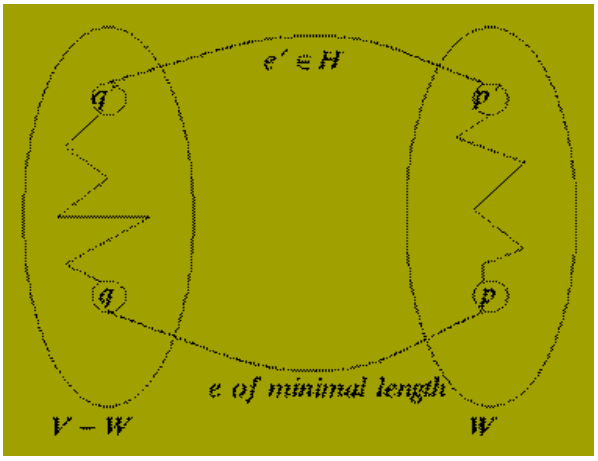
Proof: Let *T(V,H)* be a minimal spanning tree of *G(V,E)* such that *F* is a subset of *H*. Note that *T* exists since *F* is a promising set of edges.

Consider the edge *e = {p,q}* of the Lemma statement.

If *e* is in *H* then the result follows immediately, so suppose that *e* is not in *H*. Assume that *p* is in *W* and *q* is not in *W* and consider the graph *T(V, H union {e})*.

Since *T* is a tree the graph *T* (which contains one extra edge) must contain a cycle that includes the (new) edge *{p,q}*.

Now since *p* is in *W* and *q* is not in *W* there must be *some* edge, *e' = {p',q'}* in *H* which is also part of this cycle and is such that *p'* is in *W* and *q'* is not in *W*.



Now, by the choice of *e*, we know that

Length (e) < = Length (e')

Removing the edge *e'* from *T* gives a new spanning tree of *G(V,E)*.

The cost of this tree is exactly

cost(T) - Length(e') + Length(e)

and this is < = *cost(T)*.

T is a *minimal* spanning tree so either *e* and *e'* have the same length or this case cannot occur. It follows that there is a minimal spanning tree containing *F union {e}* and hence this set of edges is promising as claimed.

Theorem: Kruskal's algorithm always produces a minimal spanning tree.

Proof: We show by induction on *k > = 0* - the number of edges in *S* at each stage - that the set of edges in *S* is always promising.

Base (*k = 0*): *S = {}* and obviously the empty set of edges is promising.

Step: (< = *k-1 implies k*): Suppose *S* contains *k-1* edges. Let *e = {p,q}* be the next edge that would be added to *S*. Then:

p and *q* lie in different components.

{p,q} is a shortest such edge.

Let *C* be the component in which *p* lies. By the inductive hypothesis the set *S* is promising. The Inductive Step now follows by invoking the Lemma, with *W = Set of nodes in C* and *F = S*.

Integer Knapsack

In various forms this is a frequently arising optimisation problem. Input: A set of items *U = {u1, u2 ,..., uN}*

each item having a given *size s(ui)* and value *v(ui)*.

A capacity *K*.

Output: A subset *B* of *U* such that the sum over *u* in *B* of *s(u)* does not exceed *K* and the sum over *u* in *B* of *v(u)* is maximised.

No fast algorithm *guaranteed* to solve this problem has yet been discovered.

It is considered extremely improbable that such an algorithm exists.

Using a greedy approach, however, we can find a solution whose value is at worst 1/2 of the optimal value.

The items, *U*, are the candidates.

A subset, *B*, is a solution if the total size of *B* fits within the given capacity, but adding any other item will exceed the capacity.

The objective function which is to be maximised is the total value.

The selection function chooses that item, *ui* for which

$$\frac{v(ui)}{s(ui)}$$

is maximal

These yield the following greedy algorithm which *approximately* solves the integer knapsack problem.

```
function knapsack (U : item_set;
                   K : integer )
                   return item_set is
C, S : item_set;
x : item;
begin
  C := U; S := {};
  while C /= {} loop
    x := Item u in C such that
```

v(u)/s(u) is largest;

C := C - {x};

if *(sum over {u in S} s(u)) + s(x) < = K* then

S := S union {x};

end if;

end loop;

return *S*;

end *knapsack*;

A very simple example shows that the method can fail to deliver an optimal solution. Let

U = { u1, u2, u3 ,..., u12 }

s(u1) = 101 ; v(u1) = 102

s(ui) = v(ui) = 10 2 < = i < = 12

K = 110

Greedy solution: *S = {u1}*; Value is 102.

Optimal solution: *S = U - {u1}*; Value is 110.

Self Assessment Exercise

1. To what problem does backtracking procedure fit in? and state its approach

Self-Assessment Answer

Answer: Problem of detecting a particular class of subgraph in a graph.

Then the *backtracking* approach to solving such a problem would be:

Scan each node of the graph, *following a specific order*, until

A subgraph constituting a solution has been found.

or

It is discovered that the subgraph built so far cannot be extended to be a solution.

If (2) occurs then the search process is `*backed-up*' until a node is reached from which a solution might still be found.

3.1.4 Backtracking

In a number of applications graph structures occur. The graph may be an *explicit* object in the problem instance as in:

Shortest Path Problem

Minimal Spanning Tree Problem

Graphs, however, may also occur *implicitly* as an abstract mechanism with which to analyse problems and construct algorithms for these. Among the many areas where such an approach has been used are:

i. Game Playing Programs
ii. Theorem Proving Systems
iii. Semantic Nets
iv. Hypertext

Whether a graph is an explicit or implicit structure in describing a problem, it is often the case that *searching* the graph structure may be necessary. Thus, it is required to have methods which

Can `mark' nodes in a graph which have already been `examined'.

Determine which node should be examined next.

Ensure that every node in the graph can (but not necessarily will) be visited.

These requirements must be realised subject to the constraint that the search process respects the structure of the graph.

That is to say, (With the exception of the first node inspected)

Any new node examined must be adjacent to some node that has previously been visited.

So, *search methods* implicitly describe an ordering of the nodes in a given graph.

One search method that occurs frequently with implicit graphs is the technique known as

Backtracking

Suppose a problem may be expressed in terms of detecting a particular class of subgraph in a graph.

Then the *backtracking* approach to solving such a problem would be:

Scan each node of the graph, *following a specific order*, until

A subgraph constituting a solution has been found.

or

It is discovered that the subgraph built so far cannot be extended to be a solution.

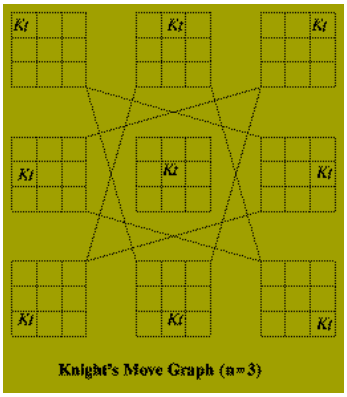
If (2) occurs then the search process is `*backed-up*' until a node is reached from which a solution might still be found.

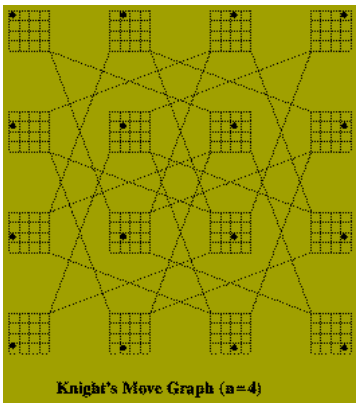
Simple Example

Knight's Tour

Given a natural number, *n*, describe how a Knight should be moved on an *n times n* chessboard so that it visits every square exactly once and ends on its starting square.

The *implicit graph* in this problem has *n^2* nodes corresponding to each position the Knight must occupy. There is an edge between two of these nodes if the corresponding positions are `one move apart'. The sub-graph that defines a solution is a cycle which contains each node of the implicit graph.





Of course it is not necessary to construct this graph explicitly, in order to to solve the problem. The algorithm below, recursively searches the graph, labeling each square (i.e. node) in the order in which it is visited. In this algorithm:

board is an *n times n* representation of the board; initiated to *0*.

(x,y) are the coordinates (row, column) of the current square.

move is the number of squares visited so far.

ok is a Boolean indicating success or failure.

type *chess_board* is array (*1..n*,*1..n*) of integer;

procedure *knight* (*board* : in out chess_board;

```
                    x,y,move : in out integer;
                    ok : in out boolean) is
w, z : integer;
begin
  if move = n^2+1 then
    ok := ( (x,y) = (1,1) );
  elsif board(x,y) /= 0 then
    ok := false;
  else
    board(x,y) := move;
    loop
      (w,z) := Next position from (x,y);
      knight(board, w, z, move+1, ok );
      exit when (ok or No moves remain);
    end loop;
    if not ok then
      board ( x,y ) :=0; -- Backtracking
    end if;
  end if;
end knight;
```

Depth-First Search

The Knight's Tour algorithm organises the search of the implicit graph using a depth-first approach.

Depth-first search is one method of constructing a search tree for *explicit graphs*.

Let *G(V,E)* be a connected graph. A search tree of *G(V,E)* is a spanning tree, *T(V, F)* of *G(V,E)* in which the nodes of *T* are labelled with unique values *k* (*1* < = *k* < = |*V*|) which satisfy:

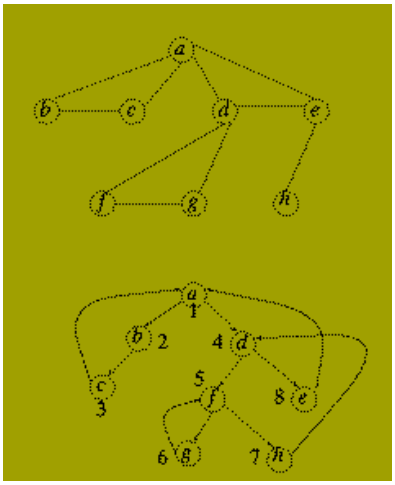
A distinguished node called the root is labelled *1*.

If *(p,q)* is an edge of *T* then the label assigned to *p* is less than the label assigned to *q*.

The labelling of a search tree prescribes the *order* in which the nodes of *G* are to be scanned.

Given an undirected graph *G(V,E)*, the depth-first search method constructs a search tree using the following recursive algorithm.

```
procedure depth_first_search (G(V,E) : graph;
                              v : node;
                              lambda : integer;
                              T : in outsearch_tree) is
begin
   label(v) := lambda;
   lambda := lambda+1;
   for each w such that {v,w} mem E loop
     if label(w) = 0 then
       Add edge {v,w} to T;
       depth_first_search(G(V,E),w,lambda,T);
     end if;
   end loop;
end depth_fist_search;
--*****************************
-- Main Program Section
--*****************************
begin
   for w mem V loop
     label( w ) := 0;
   end loop;
   lambda := 1;
   depthfirstsearch ( G(V,E), v, lambda, T);
end;
```



If *G(V,E)* is a *directed graph* then it is possible that not all of the nodes of the graph are reachable from a single root node. To deal with this the algorithm is modified by changing the `*Main Program Section'* to

```
begin
  for each w in V loop
    label( w ) := 0;
  end loop;
  lambda := 1;
  for each v mem V loop
    if label( v ) = 0 then
      depthfirstsearch ( G(V,E),v,lambda,T );
    end if;
  end loop;
end;
```

The running time of both algorithms, input *G(V,E)* is *O(|E|)* since each edge of the graph is examined only once.

It may be noted that the recursive form given, is a rather inefficient method of implementing depth first search for explicit graphs.

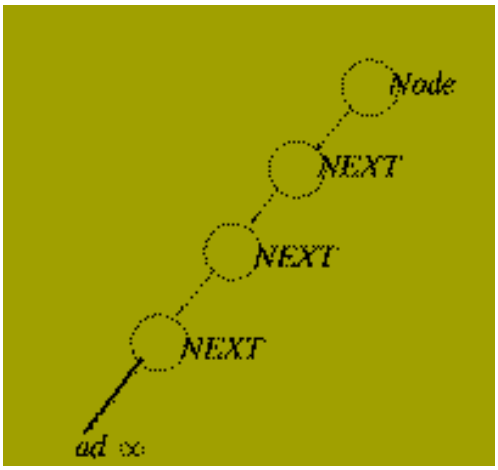
Iterative versions exist, e.g. the method of Hopcroft and Tarjan which is based on a 19th century algorithm discovered by Tremaux. Depth-first search has a large number of applications in solving other graph-based problems, for example:

Topological Sorting

Connectivity Testing

Planarity Testing

One disadvantage of depth first search as a mechanism for searching implicit graph structures, is that expanding some paths may result in the search process never terminating because no solution can be reached from these. For example this is a possible difficulty that arises when scanning proof trees in Prolog implementations.



Breadth-first Search is another search method that is less likely to exhibit such behaviour.

lambda := 1; -- First label

CurrentLevel := {v}; -- Root node

while *CurrentLevel /= (es* loop

for each *v mem CurrentLevel* loop

NextLevel := Nextlevel union

Unmarked neighbours of v;

if *label(v) = 0* then

label(v) := lambda;

lambda := lambda + 1;

end if;

end loop;

CurrentLevel := NextLevel;

NextLevel := (es;

end loop;

Thus, each vertex labelled on the *k*'th iteration of the outer loop is `expanded' before any vertex found later.

Self Assessment Exercise

1. Why Depth First Search algorithm does is preferable to Breadth First Search technique?

Self-Assessment Answer

Depth first search as a mechanism for searching implicit graph structures, expands some paths and may result in the search process never terminating because no solution can be reached from these. For example, this is a possible difficulty that arises when scanning proof trees in Prolog implementations. But Breadth-first Search is another search method that is less likely to exhibit such behavior.

4.0 Conclusion

One can solve a problem using algorithm techniques. The understanding of various algorithm design techniques, how and when to use them to formulate solutions and the context appropriate for each of them is essential in solving a particular problem. This gave rise to various algorithm paradigms namely: Divide and conquer algorithm, dynamic programming, greedy algorithm and backtracking and sorting.

The weakness of each paradigm gave rise to another paradigm. The concept of each paradigm was discussed This Module 1 study unit 2 advocates the study of algorithm design paradigms by presenting most of the useful algorithm design techniques and illustrating them through numerous examples.

5.0 Summary

In this Unit, the following aspects have been discussed:

A. advocates the study of algorithm design techniques;

B. various algorithm paradigms concept;

C. illustrating them through numerous examples.

6.0 Tutor Marked Assignments (TMAs)

1. Explain the concept of Divide and Conquer Algorithm
2. State the differences between Divide and Conquer Algorithm and Dynamic Programming.
3. State the reason why greedy algorithm is being used to design optimization problem.
4. Use depth First Search Technique to illustrate the playing of a footballer(X).
5. Differentiate between Depth first Search Technique and Breadth First Search Technique.

7.0 References/ Further Reading

Algorithm Design Paradigms - Overview by Paul Dunne at the University of Liverpool

Stony Brook Algorithm Repository by Steven S. Skiena, Department of Computer Science , State University of New York

Goodrich, Michael T.; Tamassia, Roberto (2002), *Algorithm Design: Foundations, Analysis, and Internet Examples*, John Wiley & Sons, Inc., ISBN 0-471-38365-1, http://ww3.algorithmdesign.net/ch00-front.html

Module 2

Algorithm Analysis

Unit 1: Time and Space Complexity

Unit 2: Recursion

Unit 1

Time and Space Complexity

Contents

1.0 Introduction

2.0 Learning Outcomes

3.0 Learning Contents

- 3.1 Complexity of an Algorithm
- 3.2 Time complexity
- 3.3 Three Direction of Time Analysis
  - 3.3.1 Worst Case
  - 3.3.2 Best Case
  - 3.3.3 Average Case
- 3.4 Space complexity
- 3.5 Amortized Analysis

4.0 Conclusion

5.0 Summary

6.0 Tutor Marked Assignment

7.0 References/ Further Reading

1.0 Introduction

An algorithm has to be analyzed not to only certify it correctness but also to obtain estimates as to its running time and storage requirements. When analyzing an algorithm such questions like this can be asked: How does one calculate the running time of an algorithm? How can we compare two different algorithms?

How do we know if an algorithm is `optimal'? To solve the first question one need to *Count* the number of basic operations performed by the algorithm on the worst-case input which are an assignment, a comparison between two variables, An arithmetic operation between two variables. The worst-case input is that input assignment for which the *most* basic operations are performed.

2.0 Learning Outcomes

At the end of this unit you should be able to:

i. outline the need for the running analysis of an algorithm;

ii. explain the complexity of algorithm;

iii. explain three direction of time analysis;

iv. discuss time and space analysis; and

v. describe the need for amortized analysis.

3.0 Learning Contents

3.1 Complexity of an Algorithm

To answer the last two questions, we need to Comparing the efficiency of algorithms which depends on the following important criteria: the amount of work done time complexity and the amount of space used space complexity. it does not depend on the type of computer used, programming language, programming skills, etc. Technology improves things by a constant factor only.

Even a supercomputer cannot rescue a “bad” algorithm. A faster algorithm on a slower computer will *always* win for sufficiently large inputs. Complexity of an algorithm can be obtained by deriving a function which expresses the running time of an algorithm as a function of the size of the input data to the algorithm.

3.2 Time Complexity

How long does this sorting program run? It possibly takes a very long time on large inputs (that is many strings) until the program has completed its work and gives a sign of life again. Sometimes it makes sense to be able to estimate the running time *before* starting a program. Nobody wants to wait for a sorted phone book for years! Obviously, the running time depends on the number n of the strings to be sorted. Can we find a formula for the running time which depends on n?

The number of (machine) instructions which a program executes during its running time is called its time complexity in computer science. Thus the running time T(n) is regarded as being proportional to a function say f(n). This is denoted as T(n) = c(fn). This number (n) depends primarily on the size of the program's input, that is approximately on the number of the strings to be sorted (and their length) and the algorithm used.

So approximately, the time complexity of the program “sort an array of n strings by minimum search” is described by the expression $c \cdot n^2$.

c is a constant which depends on the programming language used, on the quality of the compiler or interpreter, on the CPU, on the size of the main memory and the access time to it, on the knowledge of the programmer, and last but not least on the algorithm itself, which may require simple but also time consuming machine instructions.

(For the sake of simplicity we have drawn the factor 1/2 into c here.) So while one can make c smaller by improvement of external circumstances (and thereby often investing a lot of money), the term n^2, however, always remains unchanged.

Self Assessment Exercise

1. How can you analyze an algorithm?

Self-Assessment Answer

An algorithm has to be analyzed not to only certify it correctness but also to obtain estimates as to its running time and storage requirements. When analyzing an algorithm such question like this can be asked: how does one calculate the running time of an algorithm? How can we compare two different algorithms? How do we know if an algorithm is `optimal'?

3.3 Three Direction of Time Analysis

Under the running time analysis of an algorithm, three aspects are often considered namely, the worst, the best and the average cases analysis.

3.3.1 Worst Case

This is the reasonable time bound which the running time of an algorithm cannot exceed before providing the solution to the problem meant to solve is derived. The algorithm takes longest time to solve its problem for all valid data of size of n.

3.3.2 Best Case

Under the best case running analysis of an algorithm with input size of n. the algorithm has a reasonable lower bound of time (precisely the smallest time).

3.3.3 Average case

Under the best case running analysis of an algorithm with input size of n, the algorithm run on average over all the possible inputs.

Self Assessment Exercise

1. Write short note on complexity of an algorithm

Self-Assessment Answer

Complexity of an algorithm can be obtained by deriving a function which expresses the running time of an algorithm as a function of the size of the input data to the algorithm. Thus the running time T(n) is regarded as being proportional to a function say f(n). This is denoted as T(n) = c(fn). This number (n) depends primarily on the size of the program's input, that is approximately on the number of the strings to be sorted (and their length) and the algorithm used.

c is a constant which depends on the programming language used, on the quality of the compiler or interpreter, on the CPU, on the size of the main memory and the access time to it, on the knowledge of the programmer, and last but not least on the algorithm itself, which may require simple but also time consuming machine instructions.

The O-notation

In other words: c is not really important for the description of the running time! To take this circumstance into account, running time complexities are always specified in the so-called O-notation in computer science. One says: The sorting method has running time $O(n^2)$. The expression O is also called Landau's symbol.

Mathematically speaking, $O(n^2)$ stands for a set of functions, exactly for all those functions which, “in the long run”, do not grow faster than the function n^2, that is for those functions for which the function n^2 is an upper bound (apart from a constant factor.) To be precise, the following holds true: A function f is an element of the set $O(n^2)$ if there are a factor c and an integer number n_0 such that for all n equal to or greater than this n_0 the following holds:

$f(n) \leq c \cdot n^2$.

The function n^2 is then called an asymptotically upper bound for f. Generally, the notation f(n)=O(g(n)) says that the function f is asymptotically bounded from above by the function g.[12]

A function f from $O(n^2)$ may grow considerably more slowly than n^2 so that, mathematically speaking, the quotient f / n^2 converges to 0 with growing n. An example of this is the function f(n)=n.

However, this does not hold for the function f which describes the running time of our sorting method. This method *always* requires n^2 comparisons (apart from a constant factor of 1/2). n^2 is therefore also an asymptotically lower bound for f. This f behaves in the long run *exactly* like n^2. Expressed mathematically: There are factors c_1 and c_2 and an integer number n_0 such that for all n equal to or larger than n_0 the following holds:

$c_1 \cdot n^2 \leq f(n) \leq c_2 \cdot n^2$.

So f is bounded by n^2 from above *and* from below. There also is a notation of its own for the set of these functions: $\Theta(n^2)$.

Figure 1 contrasts a function f which is bounded from above by O(g(n)) to a function whose asymptotic behavior is described by Θ(g(n)): The latter one lies in a tube around g(n), which results from the two factors c_1 and c_2.

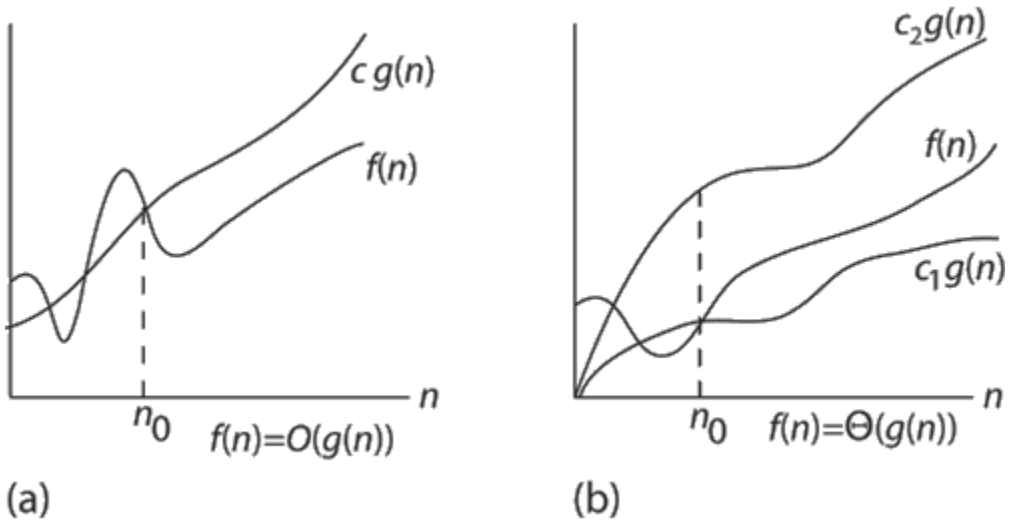


Figure 1 The asymptotical bounds O and Θ

These notations appear again and again in the LEDA manual at the description of non-trivial operations. Thereby we can estimate the order of magnitude of the method used; in general, however, we cannot make an exact running time prediction. (Because in general we do not know c, which depends on too many factors, even if it can often be determined experimentally. Frequently the statement is found in the manual that an operation takes "**constant time**".

By this it is meant that this operation is executed with a constant number of machine instructions, independently from the size of the input. The function describing the running time behavior is therefore in O(1). The expressions "**linear time**" and "**logarithmic time**" describe corresponding running time behaviors: By means of the O-notation this is often expressed as "takes time **O(n)** and **O(log(n))**", respectively.

Furthermore, the phrase "**expected time**" O(g(n)) often appears in the manual. By this it is meant that the running time of an operation can vary from execution to execution, that the expectation value of the running time is, however, asymptotically bounded from above by the function g(n).

Back to our sorting algorithm: A runtime of $\Theta(n^2)$ indicates that an adequately big input will always bring the system to its knees concerning its running time. So instead of investing a lot of money and effort in a reduction of the factor c, we should rather start to search for a better algorithm. Thanks to LEDA, we do not have to spend a long time searching for it: All known efficient sorting methods are built into LEDA.

Example: Quick Sort

Quicksort defeats sorting by minimum search in the long run: If n is large enough, the expression $c_1 \cdot n \cdot \log(n)$ certainly becomes smaller than the expression $c_2 \cdot n^2$, independently from how large the two system-dependent constants c_1 and c_2 of the two methods actually are; the quotient of the two expressions converges to 0. (For small n, however, $c_1 \cdot n \cdot \log(n)$ may definitely be larger than $c_2 \cdot n^2$; indeed, Quicksort does not pay on very small arrays compared to sorting by minimum search.)

Another sorting method which also takes only time O(n·log((n)) is Mergesort. It works recursively: One divides the array to be sorted into two halves, sorts each of the two halves recursively, and then merges the sorted halves to a sorted array.

Self Assessment Exercise

1. Can we sort phone books with our sorting algorithm in acceptable time?

Self-Assessment Answer

This depends, in accordance to what we said above, solely on the number of entries (that is the number of inhabitants of the town) and on the system-dependent constant c.

3.4 Space Complexity

The better the time complexity of an algorithm is, the faster the algorithm will carry out his work in practice. Apart from time complexity, its space complexity is also important: This is essentially the number of memory cells which an algorithm needs. A good algorithm keeps this number as small as possible, too.

There is often a time-space-tradeoff involved in a problem, that is, it cannot be solved with few computing time *and* low memory consumption. One then has to make a compromise and to exchange computing time for memory consumption or vice versa, depending on which algorithm one chooses and how one parameterizes it.

3.3 Amortized Analysis

Sometimes we find the statement in the manual that an operation takes amortized time O(f(n)). This means that the total time for n such operations is bounded asymptotically from above by a function g(n) and that f(n)=O(g(n)/n). So the amortized time is (a bound for) the *average time of an operation in the worst case*.

The special case of an amortized time of O(1) signifies that a sequence of n such operations takes only time O(n). One then refers to this as constant amortized time.

Such statements are often the result of an amortized analysis: Not each of the n operations takes equally much time; some of the operations are running time intensive and do a lot of "pre-work" (or also "post-work"), what, however, pays off by the fact that, as a result of the pre-work done, the remaining operations can be carried out so fast that a total time of O(g(n)) is not exceeded. So the investment in the pre-work or after-work amortizes itself.

Self Assessment Exercise

1. What does it mean to "**merge?**

Self-Assessment Answer

Merging is performed by pairwise comparing the elements of the two subarrays already sorted. Two index variables are used here, which run from left to right over the subarrays to be merged. The respectively smaller (or larger in a descending sorting) element is written into a temporary array. The index variable pointing to the smaller of the two values is advanced by one position to the right.

4.0 Conclusion

An algorithm has to be analyzed not to only certify it correctness but also to obtain estimates as to its running time and storage requirements. See the need for the running time analysis and space requirement of an algorithm is essential. We have been able to explore the complexity of algorithm, time and space analysis, three direction of time analysis and the need for amortized analysis.

5.0 Summary

In this Unit 1, the following aspects have been discussed:

1 Complexity of an Algorithm;

2 Time complexity;

3 Three Direction of Time Analysis: worst case, best case and average case;

4 The concept of Space complexity; and

5 The Amortized analysis.

6.0 Tutor Marked Assignments

1. Explain the concept of algorithm analysis to a lay man.
2. Explain quick sort algorithm.
3. Why is the need for space complexity?
4. What does it mean that an algorithm operation takes amortized time O(f(n))?

7.0 References/ Further Reading

Cook, Stephen (1971). "The complexity of theorem proving procedures". Proceedings of the Third Annual ACM Symposium on Theory of Computing. pp. 151–158. http://portal.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=805047.

Karp, Richard M. (1972). "Reducibility Among Combinatorial Problems". In Raymond E. Miller and James W. Thatcher (editors). Complexity of Computer Computations. New York: Plenum. pp. 85–103. ISBN 0-306-30707-3. http://www.cs.berkeley.edu/~luca/cs172/karp.pdf.

T. P. Baker; J. Gill, R. Solovay (1975). "Relativizations of the P = NP question". SIAM Journal on Computing 4 (4): 431–442. doi:10.1137/0204037.

Dekhtiar, M. (1969). "On the impossibility of eliminating exhaustive search in computing a function relative to its graph". Proceedings of the USSR Academy of Sciences 14: 1146–1148.(Russian)

Levin, Leonid (1973). "Universal search problems (Russian: Универсальные задачи перебора, Universal'nye perebornye zadachi)". Problems of Information Transmission (Russian: Проблемы передачи информации, Problemy Peredachi Informatsii) 9 (3): 265–266.(Russian), translated into English by Trakhtenbrot, B. A. (1984). "A survey of Russian approaches to perebor (brute-force searches) algorithms". Annals of the History of Computing 6 (4): 384–400. doi:10.1109/MAHC.1984.10036.

Garey, Michael R.; David S. Johnson (1979). Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman. ISBN 0-7167-1045-5.

Unit 2

Recursion

Contents

1.0 Introduction
2.0 Learning Outcomes
3.0 Learning Contents
    3.1 The Concept of Recursion
    3.2 Implementation of Recursion and its Relation
    3.3 Recursive Specialization of Mathematical Function
    3.4 Simple Recursive Procedures
        3.4.1 Divide and Conquer Strategy
        3.4.2 Recursive Backtracking
4.0 Conclusion
5.0 Summary
6.0 Tutor Marked Assignments
7.0 References/ Further Reading

1.0 Introduction

Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem.[1] The approach can be applied to many types of problems, and is one of the central ideas of computer science.[2] "The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions." [3]

Most computer programming languages support recursion by allowing a function to call itself within the program text. Some functional programming languages do not define any looping constructs but rely solely on recursion to repeatedly call code. Computability theory has proven that these recursive-only languages are mathematically equivalent to the imperative languages, meaning they can solve the same kinds of problems even without the typical control structures like “while” and “for”.

2.0 Learning Outcomes

At the end of this lesson the student should be able to:

i. the concept of recursion;
ii. implementation of recursion and its relation;
iii. recursive specialization of mathematical function such as Fibonacci and Factorial;
iv. simple Recursive Procedures (Towers of Hanoi, permutations); and
v. Recursive backtracking.

3.0 Learning Contents

3.1 The Concept of Recursion

A recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input. More generally if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive algorithm to solve that problem.

For example, the elements of a recursively defined set, or the value of a recursively defined function can be obtained by a recursive algorithm. If a set or a function is defined recursively, then a recursive algorithm to compute its members or values mirrors the definition. Initial steps of the recursive algorithm correspond to the basis clause of the recursive definition and they identify the basic elements.

They are then followed by steps corresponding to the inductive clause, which reduce the computation for an element of one generation to that of elements of the immediately preceding generation.
In general, recursive computer programs require more memory and computation compared with iterative algorithms, but they are simpler and for many cases a natural way of thinking about the problem.

Self Assessment Exercise

1. What is a recursion algorithm?

Self-Assessment Answer

A recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input.

3.1.1 Recursive data types

Many computer programs must process or generate an arbitrarily large quantity of data. Recursion is one technique for representing data whose exact size the programmer does not know: the programmer can specify this data with a self-referential definition. There are two types of self-referential definitions: inductive and conductive definitions.

3.1.2 Inductively defined data

An inductively defined recursive data definition is one that specifies how to construct instances of the data. For example, linked lists can be defined inductively (here, using Haskell syntax):

data ListOfStrings = EmptyList | Cons String ListOfStrings

The code above specifies a list of strings to be either empty, or a structure that contains a string and a list of strings.

The self-reference in the definition permits the construction of lists of any (finite) number of strings. Another example of inductive definition is the natural numbers (or non-negative integers):

A natural number is either 1 or n+1, where n is a natural number.

Similarly recursive definitions are often used to model the structure of expressions and statements in programming languages.

Language designers often express grammars in a syntax such as Backus-Naur form; here is such a grammar, for a simple language of arithmetic expressions with multiplication and addition:

<expr> ::= <number>

| (<expr> * <expr>)

| (<expr> + <expr>)

This says that an expression is either a number, a product of two expressions, or a sum of two expressions. By recursively referring to expressions in the second and third lines, the grammar permits arbitrarily complex arithmetic expressions such as (5 * ((3 * 6) + 8)), with more than one product or sum operation in a single expression.

Coinductively defined data and corecursion

A coinductive data definition is one that specifies the operations that may be performed on a piece of data; typically, self-referential coinductive definitions are used for data structures of infinite size. A coinductive definition of infinite streams of strings, given informally, might look like this:

A stream of strings is an object s such that

head(s) is a string, and

tail(s) is a stream of strings.

This is very similar to an inductive definition of lists of strings; the difference is that this definition specifies how to access the contents of the data structure—namely, via the accessor functions head and tail -- and what those contents may be, whereas the inductive definition specifies how to create the structure and what it may be created from.

Corecursion is related to coinduction, and can be used to compute particular instances of (possibly) infinite objects. As a programming technique, it is used most often in the context of lazy programming languages, and can be preferable to recursion when the desired size or precision of a program's output is unknown. In such cases the program requires both a definition for an infinitely large (or infinitely precise) result, and a mechanism for taking a finite portion of that result. The problem of computing the first n prime numbers is one that can be solved with a corecursive program

Self Assessment Exercise

1. Explain the concept of two data type of recursion

Self-Assessment Answer

An inductively defined recursive data definition is one that specifies how to construct instances of the data. An example of inductive definition is the natural numbers (or non-negative integers):

A natural number is either 1 or n+1, where n is a natural number.

Similarly recursive definitions are often used to model the structure of expressions and statements in programming languages. Language designers often express

grammars in syntax such as Backus-Naur form; here is such a grammar, for a simple language of arithmetic expressions with multiplication and addition:

<expr> ::= <number>

| (<expr> * <expr>)

| (<expr> + <expr>)

This says that an expression is either a number, a product of two expressions, or a sum of two expressions. By recursively referring to expressions in the second and third lines, the grammar permits arbitrarily complex arithmetic expressions such as (5 * ((3 * 6) + 8)), with more than one product or sum operation in a single expression.

Coinductively defined data and corecursion

A coinductive data definition is one that specifies the operations that may be performed on a piece of data; typically, self-referential coinductive definitions are used for data structures of infinite size.

A coinductive definition of infinite streams of strings, given informally, might look like this:

A stream of strings is an object s such that

head(s) is a string, and

tail(s) is a stream of strings.

This is very similar to an inductive definition of lists of strings; the difference is that this definition specifies how to access the contents of the data structure—namely, via the accessor functions head and tail -- and what those contents may be, whereas the inductive definition specifies how to create the structure and what it may be created from.

Corecursion is related to coinduction, and can be used to compute particular instances of (possibly) infinite objects. As a programming technique, it is used most often in the context of lazy programming languages, and can be preferable to recursion when the desired size or precision of a program's output is unknown. In such cases the program requires both a definition for an infinitely large (or infinitely precise) result, and a mechanism for taking a finite portion of that result. The problem of computing the first n prime numbers is one that can be solved with a corecursive program

3.2 Implementation of Recursion and Its Relation

Example 1:

Algorithm for finding the *k*-th even natural number

Note here that this can be solved very easily by simply outputting *2**(*k - 1*) for a given *k* . The purpose here, however, is to illustrate the basic idea of recursion rather than

solving the problem.

Algorithm 1: Even(positive integer *k*)
Input: *k* , a positive integer
Output: *k*-th even natural number (the first even being *0*)
Algorithm:
if *k = 1*, then return *0*;
else return Even(*k-1*) + *2* .

Here the computation of Even (*k*) is reduced to that of Even for a smaller input value that is Even (*k-1*). Even (*k*) eventually becomes even (*1*) which is *0* by the first line. For example, to compute Even (*3*), Algorithm Even (*k*) is called with *k = 2*. In the computation of Even (*2*), Algorithm Even (*k*) is called with *k = 1*. Since Even (*1*) = *0, 0* is returned for the computation of Even (*2*), and Even (*2*) = Even (*1*) + *2 = 2* is obtained. This value *2* for Even (*2*) is now returned to the computation of Even (*3*), and Even (*3*) = Even (*2*) + *2 = 4* is obtained.

As can be seen by comparing this algorithm with the recursive definition of the set of nonnegative even numbers, the first line of the algorithm corresponds to the basis clause of the definition, and the second line corresponds to the inductive clause.
By way of comparison, let us see how the same problem can be solved by an iterative algorithm.

Algorithm 1-a: Even(positive integer *k*)
Input: *k*, a positive integer
Output: *k*-th even natural number (the first even being *0*)

Algorithm:
int *i*, *even*;
i := *1*;
even := *0*;
while(*i* < *k*) {
even := *even* + *2*;
i := *i* + *1*;
}
return *even* .

Example 2:

Algorithm for computing the *k*-th power of *2*
Algorithm 2 Power_of_2(natural number k)
Input: *k* , a natural number

Output: *k*-th power of *2*
Algorithm:
if *k = 0*, then return *1*;
else return 2*Power_of_2(*k - 1*) .

By way of comparison, let us see how the same problem can be solved by an iterative algorithm.

Algorithm 2-a Power_of_2(natural number k)
Input: *k* , a natural number
Output: *k*-th power of *2*

Algorithm:
int *i*, *power*;
i := *0*;
power := *1*;
while(*i* < *k*) {
power := *power* * *2*;
i := *i* + *1*;
}
return *power* .

The next example does not have any corresponding recursive definition. It shows a recursive way of solving a problem.

Example 3:

Recursive Algorithm for Sequential Search

Algorithm 3 SeqSearch(*L, i, j, x*)

Input: *L* is an array, *i* and *j* are positive integers, $i \leq j$, and *x* is the key to be searched for in *L*.
Output: If *x* is in *L* between indexes *i* and *j*, then output its index, else output *0*.
Algorithm:
if $i \leq j$, then
{
if *L*(*i*) = *x*, then return *i* ;
else return SeqSearch(*L, i+1, j, x*)
}
else return *0*.
Recursive algorithms can also be used to test objects for membership in a set.

Example 4:

Algorithm for testing whether or not a number *x* is a natural number

Algorithm 4 Natural(a number *x*)
Input: A number *x*
Output: "Yes" if *x* is a natural number, else "No"

Algorithm:
if *x* < *0*, then return "No"
else
if *x* = *0*, then return "Yes"
else return Natural(*x* - *1*)

Self Assessment Exercise

1. Write an Algorithm for finding the *k*-th odd natural number

Self-Assessment Answer

Algorithm 1: Odd (positive integer *k*)
Input: *k* , a positive integer
Output: *k*-th odd natural number (the first odd being *1*)

Algorithm:
if *k* = *0*, then return *0*;
else return odd(*k-1*) + *2* .

3.3 Recursive specialization of mathematical function

(such as factorial & Fibonacci)

Recursive programs

Recursive procedures

Factorial

A classic example of a recursive procedure is the function used to calculate the factorial of a natural number:

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{fact}(n-1) & \text{if } n > 0 \end{cases}$$

Pseudocode (recursive):

function factorial is:
input: integer *n* such that *n* >= 0
output: [*n* × (*n*-1) × (*n*-2) × … × 1]

1. if *n* is 0, return 1
2. otherwise, return [*n* × factorial(*n*-1)]

end factorial

The function can also be written as a recurrence relation:

$$b_n = nb_{n-1}$$

$$b_0 = 1$$

This evaluation of the recurrence relation demonstrates the computation that would be performed in evaluating the pseudocode above:

Computing the recurrence relation for n = 4:

$b_4 = 4 * b_3$
= 4 * 3 * b_2
= 4 * 3 * 2 * b_1
= 4 * 3 * 2 * 1 * b_0
= 4 * 3 * 2 * 1 * 1
= 4 * 3 * 2 * 1
= 4 * 3 * 2
= 4 * 6
= 24

This factorial function can also be described without using recursion by making use of the typical looping constructs found in imperative programming languages:

Pseudocode (iterative):

function factorial is:
input: integer *n* such that *n* >= 0
output: [*n* × (*n*-1) × (*n*-2) × … × 1]

1. create new variable called *running_total* with a value of 1

2. begin loop
    1. if *n* is 0, exit loop
    2. set *running_total* to (*running_total* × *n*)
    3. decrement *n*
    4. repeat loop

3. return *running_total*

end factorial

The imperative code above is equivalent to this mathematical definition using an accumulator variable *t:*

$$\begin{aligned} \operatorname{fact}(n) &= \operatorname{fact}_{\mathrm{acc}}(n, 1) \\ \operatorname{fact}_{\mathrm{acc}}(n, t) &= \begin{cases} t & \text{if } n = 0 \\ \operatorname{fact}_{\mathrm{acc}}(n-1, nt) & \text{if } n > 0 \end{cases} \end{aligned}$$

The definition above translates straightforwardly to functional programming languages such as Scheme; this is an example of iteration implemented recursively.

Fibonacci

Another well known mathematical recursive function is one that computes the Fibonacci numbers:

$$\operatorname{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \operatorname{fib}(n-1) + \operatorname{fib}(n-2) & \text{if } n >= 2 \end{cases}$$

Pseudo code

function fib is:

input: integer n such that n >= 0

1. if n is 0, return 0
2. if n is 1, return 1
3. otherwise, return [fib(n-1) + fib(n-2)]

end fib

C language implementation:

```
int fib(int n)
{
        if(n < 2)
                return n;
         return fib(n-1) + fib(n-2);
}
```

Java language implementation:

```
    /**
     * Recursively calculate the kth Fibonacci number.
     *
     * @param k indicates which (positive) Fibonacci number to compute.
     * @return the kth Fibonacci number.
     */
    private static int fib(int k) {

        // Base Cases:
        //   If k == 0 then fib(k) = 0.
        //   If k == 1 then fib(k) = 1.
        if (k < 2) {
            return k;
        }
        // Recursive Case:
        //   If k >= 2 then fib(k) = fib(k-1) + fib(k-2).
        return fib(k-1) + fib(k-2);
    }
```

C# language implementation:

```
static int Fib(int n)
        {
            return n <= 1 ? n : Fib(n - 1) + Fib(n - 2);
        }
```

Python language implementation:

```
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Scheme language implementation:

```
(define (fib n)
    (if (< n 2)
        n
        (+ (fib (- n 1)) (fib ( - n 2)))))
```

JavaScript language implementation:

```
function fib (n) {
    if (!n) {
        return 0;
    } else if (n <= 2) {
        return 1;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

Common Lisp implementation:

```
(defun fib (n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (t (+ (fib (- n 1))
              (fib (- n 2))))))
```

Perl implementation:

```
sub fib {
    my ($n) = @_;
    ($n < 2) ? $n : fib($n - 2) + fib($n - 1);
}
```

Ruby implementation:

```
def fib(n)
  n < 2 ? n : fib(n - 1) + fib(n - 2)
end
```

Recurrence relation for Fibonacci:
$b_n = b_{n-1} + b_{n-2}$
$b_1 = 1, b_0 = 0$

Computing the recurrence relation for n = 4:

$$\begin{aligned} b_4 &= b_3 + b_2 \\ &= b_2 + b_1 + b_1 + b_0 \\ &= b_1 + b_0 + 1 + 1 + 0 \\ &= 1 + 0 + 1 + 1 + 0 \\ &= 3 \end{aligned}$$

This Fibonacci algorithm is a particularly poor example of recursion, because each time the function is executed on a number greater than one, it makes two function calls to itself, leading to an exponential number of calls (and thus exponential time complexity) in total.[6] The following alternative approach uses two accumulator variables TwoBack and OneBack to "remember" the previous two Fibonacci numbers constructed, and so avoids the exponential time cost:

Pseudocode

function fib is:

input: integer Times such that Times >= 0, relative to TwoBack and OneBack

long TwoBack such that TwoBack = fib(x)

long OneBack such that OneBack = fib(x)

1. if Times is 0, return TwoBack
2. if Times is 1, return OneBack
3. if Times is 2, return TwoBack + OneBack
4. otherwise, return [fib(Times-1, OneBack, TwoBack + OneBack)]

end fib

To obtain the tenth number in the Fib. sequence, one must perform Fib(10,0,1). Where 0 is considered TwoNumbers back and 1 is considered OneNumber back. As can be seen in this approach, no trees are being created, therefore the efficiency is much greater, being a linear recursion. The recursion in condition 4, shows that OneNumber back becomes TwoNumbers back, and the new OneNumber back is calculated, simply decrementing the Times on each recursion.

Alternatively, corecursion can produce the Fib. sequence in linear time.

Implemented in the Java or the C# programming language:

```
public static long fibonacciOf(int times, long twoNumbersBack, long
oneNumberBack) {


   if (times == 0) {                    // Used only for fibonacciOf(0, 0, 1)
      return twoNumbersBack;
   } else if (times == 1) {               // Used only for fibonacciOf(1, 0, 1)
      return oneNumberBack;
   } else if (times == 2) {               // When the 0 and 1 clauses are included,
      return oneNumberBack + twoNumbersBack;   // this clause merely stops one
additional
   } else {                              // recursion from occurring
      return fibonacciOf(times - 1, oneNumberBack, oneNumberBack +
twoNumbersBack);
   }
}
```

Self Assessment Exercise

1. Explain the concept of recursion in Fibonacci Search
2. Algorithm for testing whether or not an expression *w* is a proposition (propositional form)

Self-Assessment Answer

Algorithm Proposition(a string *w*)

Input: A string *w*
Output: "Yes" if *w* is a proposition, else "No"

Algorithm:
if *w* is *1*(true), *0*(false), or a propositional variable, then return "Yes"
else if $w = \sim w_1$, then return Proposition(w_1)
else
if ($w = w_1 \vee w_2$ or $w_1 \wedge w_2$ or $w_1 \rightarrow w_2$ or $w_1 \leftrightarrow w_2$) and
Proposition(w_1) = Yes and Proposition(w_2) = Yes
then return Yes
else return No
end

Greatest common divisor

Another famous recursive function is the Euclidean algorithm, used to compute the greatest common divisor of two integers. Function definition*:*

$$\gcd(x, y) = \begin{cases} x & \text{if } y = 0 \\ \gcd(y, \text{remainder}(x, y)) & \text{if } x \geq y \text{ and } y > 0 \end{cases}$$

Pseudocode (recursive):

function gcd is:

input: integer *x*, integer *y* such that *x* >= *y* and *y* >= 0

1. if *y* is 0, return *x*
2. otherwise, return [gcd(*y*, (remainder of *x*/*y*))]

end gcd

Recurrence relation for greatest common divisor, where $x\%y$expresses the remainder of x/y:

$$\gcd(x, y) = \gcd(y, x\%y)$$
$$\gcd(x, 0) = x$$

Computing the recurrence relation for x = 27 and y = 9:

gcd(27, 9) = gcd(9, 27 % 9)
= gcd(9, 0)
= 9

Computing the recurrence relation for x = 259 and y = 111:

gcd(259, 111) = gcd(111, 259 % 111)
= gcd(111, 37)
= gcd(37, 0)
= 37

The recursive program above is tail-recursive; it is equivalent to an iterative algorithm, and the computation shown above shows the steps of evaluation that would be performed by a language that eliminates tail calls. Below is a version of the same algorithm using explicit iteration, suitable for a language that does not eliminate tail calls. By maintaining its state entirely in the variables *x* and *y* and using a looping construct, the program avoids making recursive calls and growing the call stack.

Pseudocode (iterative):

function gcd is:
input: integer *x*, integer *y* such that *x* >= *y* and *y* >= 0

1. create new variable called *remainder*
2. begin loop
    1. if *y* is zero, exit loop
    2. set *remainder* to the remainder of x/y
    3. set x to y
    4. set y to *remainder*
    5. repeat loop
3. return *x*

end gcd

The iterative algorithm requires a temporary variable, and even given knowledge of the Euclidean algorithm it is more difficult to understand the process by simple inspection, although the two algorithms are very similar in their steps.

3.4 Simple Recursive Procedures (Towers of Hanoi)

Towers of Hanoi

Simply put the problem is this: given three pegs, one with a set of N disks of increasing size, determine the minimum (optimal) number of steps it takes to move all the disks from their initial position to another peg without placing a larger disk on top of a smaller one.

Function definition:

$$\text{hanoi}(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2 \cdot \text{hanoi}(n-1) + 1 & \text{if } n > 1 \end{cases}$$

Recurrence relation for hanoi:

$$h_n = 2h_{n-1} + 1$$
$$h_1 = 1$$

Computing the recurrence relation for n = 4:

hanoi(4) = 2*hanoi(3) + 1
= 2*(2*hanoi(2) + 1) + 1
= 2*(2*(2*hanoi(1) + 1) + 1) + 1
= 2*(2*(2*1 + 1) + 1) + 1
= 2*(2*(3) + 1) + 1
= 2*(7) + 1
= 15

Example Implementations:

Pseudocode (recursive):

function hanoi is:
input: integer *n*, such that *n* >= *1*

1. if n is 1 then return 1

2. return [2 * [call hanoi(n-1)] + 1]

end Hanoi

Although not all recursive functions have an explicit solution, the Tower of Hanoi sequence can be reduced to an explicit formula.[9]

An explicit formula for Towers of Hanoi:

$h_1 = 1 = 2^1 - 1$

$h_2 = 3 = 2^2 - 1$

$h_3 = 7 = 2^3 - 1$

$h_4 = 15 = 2^4 - 1$

$h_5 = 31 = 2^5 - 1$

$h_6 = 63 = 2^6 - 1$

$h_7 = 127 = 2^7 - 1$

In general:

$h_n = 2^n - 1$, for all n >= 1

3.4.1 Divide and Conquer Strategy

Binary search

The binary search algorithm is a method of searching an ordered array for a single element by cutting the array in half with each pass. The trick is to pick a midpoint near the center of the array, compare the data at that point with the data being searched and then responding to one of three possible conditions: the data is found at the midpoint, the data at the midpoint is greater than the data being searched for, or the data at the midpoint is less than the data being searched for.

Recursion is used in this algorithm because with each pass a new array is created by cutting the old one in half. The binary search procedure is then called recursively, this time on the new (and smaller) array. Typically the array's size is adjusted by manipulating a beginning and ending index. The algorithm exhibits a logarithmic order of growth because it essentially divides the problem domain in half with each pass.

Example implementation of binary search in C:

```
/*
  Call binary_search with proper initial conditions.

  INPUT:
    data is an array of integers SORTED in ASCENDING order,
    toFind is the integer to search for,
    count is the total number of elements in the array

  OUTPUT:
    result of binary_search
```

```
*/
int search(int *data, int toFind, int count)
{
   //  Start = 0 (beginning index)
   //  End = count - 1 (top index)
   return binary_search(data, toFind, 0, count-1);
}
 /*
```

Binary Search Algorithm.

```
   INPUT:
       data is a array of integers SORTED in ASCENDING order,
       toFind is the integer to search for,
       start is the minimum array index,
       end is the maximum array index
  OUTPUT:
       position of the integer toFind within array data,
       -1 if not found
*/
int binary_search(int *data, int toFind, int start, int end)
{
   //Get the midpoint.
   int mid = start + (end - start)/2;   //Integer division
    //Stop condition.
   if (start > end)
      return -1;
   else if (data[mid] == toFind)        //Found?
      return mid;
   else if (data[mid] > toFind)         //Data is greater than toFind, search lower half
      return binary_search(data, toFind, start, mid-1);
   else                                 //Data is less than toFind, search upper half
      return binary_search(data, toFind, mid+1, end);
}
```

Recursive data structures (structural recursion)

Recursive data type

An important application of recursion in computer science is in defining dynamic data structures such as Lists and Trees. Recursive data structures can dynamically grow to a theoretically infinite size in response to runtime requirements; in contrast, a static array's size requirements must be set at compile time.

"Recursive algorithms are particularly appropriate when the underlying problem or the data to be treated are defined in recursive terms." [10]

The examples in this section illustrate what is known as "structural recursion". This term refers to the fact that the recursive procedures are acting on data that is defined recursively.

As long as a programmer derives the template from a data definition, functions employ structural recursion. That is, the recursions in a function's body consume some immediate piece of a given compound value.[5]

Linked lists

Below is a simple definition of a linked list node. Notice especially how the node is defined in terms of itself. The "next" element of *struct node* is a pointer to another *struct node*, effectively creating a list type.

```
struct node
{
  int data;           // some integer data
  struct node *next;  // pointer to another struct node
};
```

Because the *struct node* data structure is defined recursively, procedures that operate on them can be implemented naturally as a recursive procedure. The *list_print* procedure defined below walks down the list until the list is empty (or NULL). For each node it prints the data element (an integer). In the C implementation, the list remains unchanged by the *list_print* procedure.

```
void list_print(struct node *list)
{
    if (list != NULL)              // base case
    {
       printf ("%d ", list->data);  // print integer data followed by a space
       list_print (list->next);     // recursive call on the next node
    }
}
```

Binary trees

Binary tree(Divide and Conquer Starategy)

Below is a simple definition for a binary tree node. Like the node for linked lists, it is defined in terms of itself, recursively. There are two self-referential pointers: left (pointing to the left sub-tree) and right (pointing to the right sub-tree).

```
struct node
{
  int data;           // some integer data
  struct node *left;   // pointer to the left subtree
  struct node *right;  // point to the right subtree
};
```

Operations on the tree can be implemented using recursion. Note that because there are two self-referencing pointers (left and right), tree operations may require two recursive calls:

```
// Test if tree_node contains i; return 1 if so, 0 if not.
int tree_contains(struct node *tree_node, int i) {
      if (tree_node == NULL)
            return 0;  // base case
      else if (tree_node->data == i)
            return 1;
      else
            return tree_c      ontains(tree_node->left,   i)   ||   tree_contains(tree_node->right, i);
}
```

At most two recursive calls will be made for any given call to *tree_contains* as defined above.

```
// Inorder traversal:
void tree_print(struct node *tree_node) {
      if (tree_node != NULL) {              // base case
            tree_print(tree_node->left);   // go left
            printf("%d ", tree_node->n);   // print the integer followed by a space
            tree_print(tree_node->right);  // go right
      }
```

}

The above example illustrates an in-order traversal of the binary tree. A Binary search tree is a special case of the binary tree where the data elements of each node are in order

Self Assessment Exercise

1. Explain the concept of recursion in Fibonacci Search

Self-Assessment Answer

Answer: Fibonacci Search mathematical recursive function is one that computes the Fibonacci numbers:

$$\text{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n >= 2 \end{cases}$$

Pseudocode

function fib is:

input: integer n such that n >= 0

1. if n is 0, return 0
2. if n is 1, return 1
3. otherwise, return [fib(n-1) + fib(n-2)]

end fib

C language implementation:

```
int fib(int n)
{
        if(n < 2)
                return n;

        return fib(n-1) + fib(n-2);
}
```

Java language implementation:

```
    /**
```

```
 * Recursively calculate the kth Fibonacci number.
 *
 * @param k indicates which (positive) Fibonacci number to compute.
 * @return the kth Fibonacci number.
 */
private static int fib(int k) {
    // Base Cases:
   //   If k == 0 then fib(k) = 0.
   //   If k == 1 then fib(k) = 1.
   if (k < 2) {
       return k;
   }
   // Recursive Case:
   //   If k >= 2 then fib(k) = fib(k-1) + fib(k-2).
   return fib(k-1) + fib(k-2);
    }
```

4.0 Conclusion

A recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input. The two types of recursive algorithm have been defined: inductive and co inductive. Several algorithms for the implementation of recursion have been defined.

Recursive specialization of mathematical function (such as factorial & Fibonacci) and Simple Recursive Procedures are also explained in detail all these are the recursion mechanism used in algorithm paradigm.

5.0 Summary

In this Module 2 Study Unit 2, the following aspects have been discussed:

1. The concept of recursion
2. Implementation of recursion and its relation
3. Recursive specialization of mathematical function such as Fibonacci and Factorial
4. Simple Recursive Procedures (Towers of Hanoi, permutations) and
5. Recursive backtracking

6.0 Tutor Marked Assignments

Discuss the concept of recursion

1. Explain the procedure of Recurrence relation for Tower of a Hanoi
2. How can you perform a binary search for an item K in a list of 100 elements?
3. How does binary tree fit in to divide and conquer strategy?

7.0 References/ Further Reading

Graham, Ronald; Donald Knuth, Oren Patashnik (1990). *Concrete Mathematics*. Chapter 1: Recurrent Problems. http://www-cs-faculty.stanford.edu/~knuth/gkp.html.

Epp, Susanna (1995). *Discrete Mathematics with Applications* (2nd ed.). p. 427.

Wirth, Niklaus (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall. p. 126.

Felleisen, Matthias; Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi (2001). *How to Design Programs: An Introduction to Computing and Programming*. Cambridge, MASS: MIT Press. p. art V "Generative Recursion". http://www.htdp.org/2003-09-26/Book/curriculum-Z-H-31.html.

Felleisen, Matthias (2002). "Developing Interactive Web Programs". In Jeuring, Johan. *Advanced Functional Programming: 4th International School*. Oxford, UK: Springer. p. 108

Abelson, Harold; Gerald Jay Sussman (1996). *Structure and Interpretation of Computer Programs*. Section 1.2.2. http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-11.html#%_sec_1.2.2.

Graham, Ronald; Donald Knuth, Oren Patashnik (1990). *Concrete Mathematics*. Chapter 1, Section 1.1: The Tower of Hanoi. http://www-cs-faculty.stanford.edu/~knuth/gkp.html.

Epp, Susanna (1995). *Discrete Mathematics with Applications* (2nd ed.). pp. 427–430: The Tower of Hanoi.

Epp, Susanna (1995). *Discrete Mathematics with Applications* (2nd ed.). pp. 447–448: An Explicit Formula for the Tower of Hanoi Sequence.

Wirth, Niklaus (1976). *Algorithms + Data Structures = Programs*. Prentice-Hall. p. 127.

http://www.saasblogs.com/2006/09/15/how-to-rewrite-standard-recursion-through-a-state-stack-amp-iteration/

http://www.refactoring.com/catalog/replaceRecursionWithIteration.html

http://www.ccs.neu.edu/home/shivers/papers/loop.pdf

http://lambda-the-ultimate.org/node/1014

http://docs.python.org/library/sys.html

Module 3

Trees and Graph

Unit 1: Trees
Unit 2: Graph

Unit 1

Trees

Contents

1.0 Introduction

2.0 Learning Outcomes

3.0 Learning Contents

- 3.1 Representation of a Tree
  - 3.1.1 Terminology
  - 3.1.2 Drawing Tree
  - 3.1.3 Representations
  - 3.1.4 Types of Trees
- 3.2 Operations on Tree
- 3.3 Common Uses
- 3.4 Tree Traverse

4.0 Conclusion

5.0 Summary

6.0 Tutor Marked Assignments

7.0 References/ Further Reading

1.0 Introduction

In computer science, a tree is a widely used data structure that simulates a hierarchical tree structure with a set of linked nodes.

A tree can be defined recursively (locally) as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of nodes (the "children"), with the constraints that no node is duplicated. A tree can be defined abstractly as a whole (globally) as an ordered tree, with a value assigned to each node.

Both these perspectives are useful: while a tree can be analyzed mathematically as a whole, when actually represented as a data structure it is usually represented and worked with separately by node (rather than as a list of nodes and an adjacency list of edges between nodes, as one may represent a digraph, for instance). For example, looking at a tree as a whole, one can talk about "the parent node" of a given node, but in general as a data structure a given node only contains the list of its children, but does not contain a reference to its parent (if any).

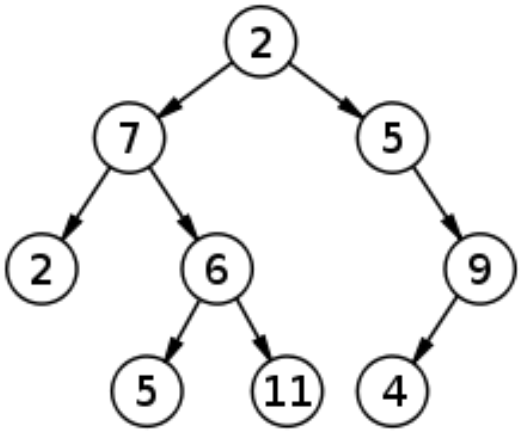


Figure 2: A simple unordered tree

In this Figure 2, the node labeled 7 has two children, labeled 2 and 6, and one parent, labeled 2. The root node, at the top, has no parent.

2.0 Learning Outcomes

At the end of this lesson the students should be able to:

i. perform representation of a tree;

ii. define Operations on Tree;

iii. Perform Tree Traverse and know the terms used in traverse.

3.0 Learning Contents

3.1 Representation of a Tree

3.1.1 Terminology

A **node** is a structure which may contain a value or condition, or represent a separate data structure (which could be a tree of its own). Each node in a tree has zero or more child nodes, which are below it in the tree (by convention, trees are drawn growing downwards). A node that has a child is called the child's parent node (or *ancestor node*, or superior). A node has at most one parent.

An internal node (also known as an inner node or branch node) is any node of a tree that has child nodes. Similarly, an external node (also known as an outer node, leaf node, or terminal node) is any node that does not have child nodes. The topmost node in a tree is called the root node. Being the topmost node, the root node will not have a parent. It is the node at which algorithms on the tree begin, since as a data structure, one can only pass from parents to children.

Note that some algorithms (such as post-order depth-first search) begin at the root, but first visit leaf nodes (access the value of leaf nodes), only visit the root last (i.e., they first access the children of the root, but only access the *value* of the root last). All other nodes can be reached from it by following edges or links. (In the formal definition, each such path is also unique.) In diagrams, the root node is conventionally drawn at the top.

In some trees, such as heaps, the root node has special properties. Every node in a tree can be seen as the root node of the subtree rooted at that node. The height of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree. The depth of a node is the length of the path to its root (i.e., its *root path*). This is commonly needed in the manipulation of the various self balancing trees, AVL Trees in particular.

The root node has depth zero, leaf nodes have height zero, and a tree with only a single node (hence both a root and leaf) has depth and height zero. Conventionally, an empty tree (tree with no nodes) has depth and height −1.

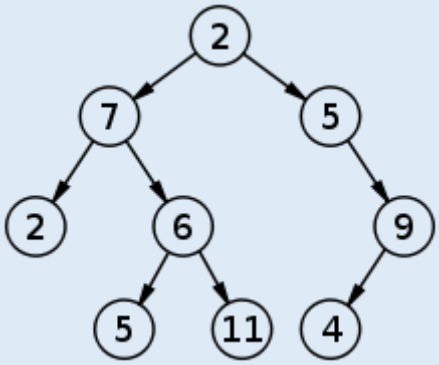
A subtree of a tree *T* is a tree consisting of a node in *T* and all of its descendants in *T*. Nodes thus correspond to subtrees (each node corresponds to the subtree of itself and all its descendants) – the subtree corresponding to the root node is the entire tree, and each node is the root node of the subtree it determines; the subtree corresponding to any other node is called a proper subtree (in analogy to the term proper subset).

Self Assessment Exercise

1. What is a tree?

Self-Assessment Answer

A tree can be defined as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of nodes (the "children"), with the constraints that no node is duplicated. A tree can be defined abstractly as a whole (globally) as an ordered tree, with a value assigned to each node.



This is a simple unordered tree.

3.1.2 Drawing Tree

Trees are often drawn in the plane. Ordered trees can be represented essentially uniquely in the plane, and are hence called *plane trees,* as follows: if one fixes a conventional order (say, counterclockwise), and arranges the child nodes in that order (first incoming parent edge, then first child edge, etc.), this yields an embedding of the tree in the plane, unique up to ambient isotopy.

Conversely, such an embedding determines an ordering of the child nodes.

If one places the root at the top (parents above children, as in a family tree) and places all nodes that are a given distance from the root (in terms of number of edges: the "level" of a tree) on a given horizontal line, one obtains a standard drawing of the tree. Given a binary tree, the first child is on the left (the "left node"), and the second child is on the right (the "right node").

Arbitrary trees

Tree is a widely-used data structure that emulates a tree structure with a set of linked nodes. The tree graphicaly is represented most commonly as on *Picture 1*. The circles are the nodes and the edges are the links between them.

Trees are usually used to store and represent data in some hierarchical order. The data are stored in the nodes, from which the tree is consisted of. A node may contain a value or a condition or represent a separate data structure or a tree of its own. Each

node in a tree has zero or more child nodes, which are one level lower in the tree hierarchy (by convention, trees grow down, not up as they do in nature).

A node that has a child is called the child's parent node (or ancestor node, or superior). A node has at most one parent. A node that has no child is called a leaf, and that node is of course at the bottommost level of the tree. The height of a node is the length of the longest path to a leaf from that node. The height of the root is the height of the tree. In other words, the "height" of tree is the "number of levels" in the tree. Or more formally, the height of a tree is defined as follows:

i. The height of a tree with no elements is 0

ii. The height of a tree with 1 element is 1

iii. The height of a tree with > 1 element is equal to 1 + the height of its tallest subtree.

The depth of a node is the length of the path to its root (i.e., its root path). Every child node is always one level lower than his parent. The topmost node in a tree is called the root node. Being the topmost node, the root node will not have parents. It is the node at which operations on the tree commonly begin (although some algorithms begin with the leaf nodes and work up ending at the root).

All other nodes can be reached from it by following edges or links. (In the formal definition, a path from a root to a node, for each different node is always unique). In diagrams, it is typically drawn at the top.

In some trees, such as heaps, the root node has special properties.

A subtree is a portion of a tree data structure that can be viewed as a complete tree in itself. Any node in a tree T, together with all the nodes below his height, that are reachable from the node, comprise a subtree of T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree (in analogy to the term proper subset).

Every node in a tree can be seen as the root node of the subtree rooted at that node.

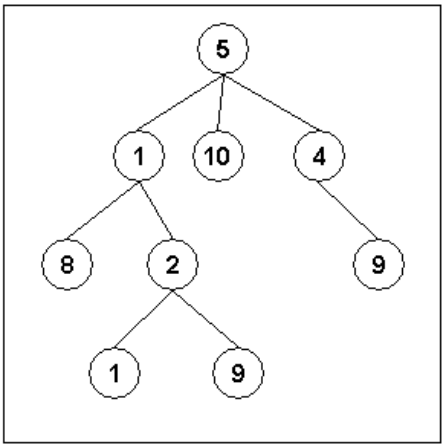


Figure 3. An example of a tree

An internal node or inner node is any node of a tree that has child nodes and is thus not have a leaf node.

There are two basic types of trees. In an unordered tree, a tree is a tree in a purely structural sense — that is to say, given a node, there is no order for the children of that node. A tree on which an order is imposed — for example, by assigning different natural numbers to each child of each node — is called an ordered tree, and data structures built on them are called ordered tree data structures. Ordered trees are by far the most common form of tree data structure. Binary search trees are one kind of ordered tree.

3.1.3 Representations

There are many different ways to represent trees; common representations represent the nodes as records dynamically allocated on the heap memory with pointers to their children, their parents, or both, and the children will point to their children and etc., and this repeats recursively until a leaf is reached or as items in an array, with relationships between them determined by their positions in the array (e.g., binary heap).

Here below is an example of the declaration of the structure of a tree node written in C, in which the node will be limited to have mostly two children (in this case a left and right child). This kind of tree is called binary tree:

typedef *type_of_data_in_the_node* info_t;

struct node{

info_t info;

struct node *llink;

struct node *rlink;

};

typedef node * nodep;

info will store the data in the node (for example: an integer), the pointers llink and rlink will point at the left and right child node (the two childs) of the node, which are dynamically allocated. The left and right child nodee can be dynamically allocated and llink and rlink can be made to point at them, and the child nodes can point to some other nodes and so on like this, so it can be formed a continual recursive chain structure between the root (or subroots) and his/their subtrees.

This is how actually the hierarchical order of the binary tree is formed, and there are very well known algorithms for creating tree structures consisted from nodes and links. We can traverse through the nodes of the tree by following the links of the nodes, that is, by reading the data at the address at which the llink and rlink pointers point.

If a node does not have a left or a right child or both, then the left or right link or both should have value NULL. In this case this value of the pointer tells that the node does

not have a left or right child node. The structure of the node can have also a pointer to his parent.

In languages like FORTRAN, that do not support records or pointers.

Pointers can be implemented by means of arrays of subscripts, for example, ILB(K) would be the subscript for the left child to node K, and IRB(K) would be the subscript for the right child. K is an integer number equal to the index of the array for the nodes values, and corresponds to the pointers of the addresses of the nodes.

Binary trees are not ordinary trees, they are special type of trees that have some specific characteristics and uses that do not apply to the ordinary trees.

For example: the existence, and the difference between a left and a right branch (link) and child of a node and not simply branches and childs, the use in binary searching, parsing math formulas in program languages, and etc.

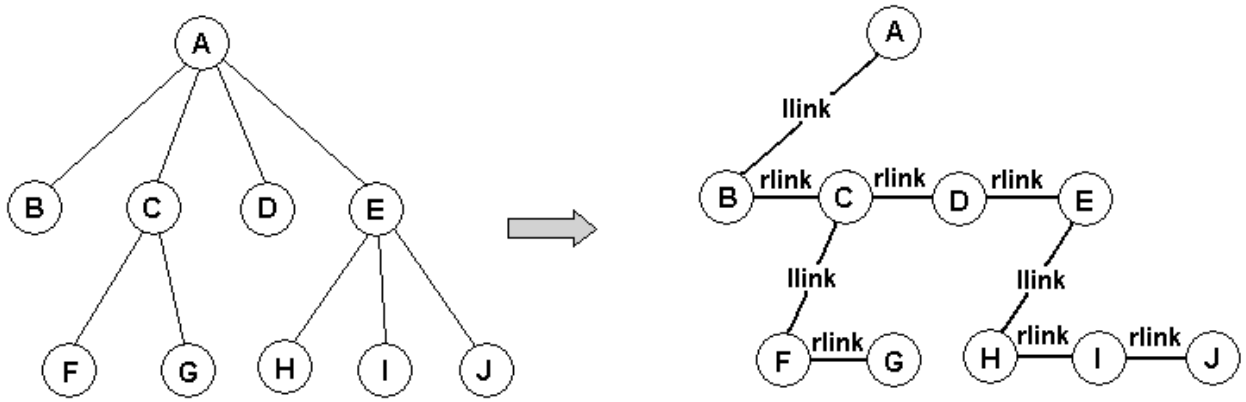
Unlike the binary trees, there are structural uncommodities when representing customary (ordinary) in structural meaning trees in which each node has arbitrary number of children nodes, as the tree on the Picture 1. One node in that kind of tree would be free to have zero, one, two, hundred, thousand or whatever number of child nodes. This makes problems in the declaration of the node structure in the program languages, and the limited memory of the computer.

It is not a good decision to define some maximal number of links (or pointers) to the children in the declaration of the node structure, because of the limits, and memory waste.

In this case, the structure of a binary tree can be used to represent a customary tree, where there will be no need of wasting large memory for declarating a maximal number of links for the children. The nodes will be dynamically allocated, and only two pointers for each node will be used.

The picture below explains this idea pretty good:

Figure 4*:* idea pretty good



>

Figure 4(a) Figure 4(b)

The left link of some node *N* will tell if that node has children nodes. If the left link (pointer) of *N* is NULL then this tells that the *N* has no children nodes. In the other case the llink pointer will point to one of the children. The node that is reached by the llink pointer is the one of the children nodes of *N*, and then if we recursively follow the right links of this node, we can visit all the rest of child nodes of *N* until rlink is not NULL.

The representation and traversing of the children nodes of a parent node, resembles a linked linear list. From the root down to the leaves we can recursively repeat this procedure for every node. A logical conclusion is that rlink in this representation points to the siblings of a node.

Usual graph representations can also be used to represent arbitrary tree structure.

In graph theory, a tree is a connected acyclic graph (or sometimes, a connected directed acyclic graph). A rooted tree is such a graph with a vertex singled out as the root. In this case, any two vertices connected by an edge inherit a parent-child relationship. An acyclic graph with multiple connected components or a set of rooted trees is sometimes called a forest.

Self Assessment Exercise

1. Explain these properties of a tree: a node, parent node, leaves, children node, subtree, height of a node, depth of a node, internal node and external node?

Self-Assessment Answer

A **node** is a structure which may contain a value or condition, or represent a separate data structure (which could be a tree of its own). Each node in a tree has zero or more child nodes, which are below it in the tree (by convention, trees are drawn growing downwards). A node that has a child is called the child's parent node (or *ancestor node*, or superior). A node has at most one parent.

An internal node (also known as an inner node or branch node) is any node of a tree that has child nodes. Similarly, an external node (also known as an outer node, leaf node, or terminal node) is any node that does not have child nodes.

The topmost node in a tree is called the root node. Being the topmost node, the root node will not have a parent. It is the node at which algorithms on the tree begin, since as a data structure, one can only pass from parents to children. Note that some algorithms (such as post-order depth-first search) begin at the root, but first visit leaf nodes (access the value of leaf nodes), only visit the root last (i.e., they first access the children of the root, but only access the *value* of the root last).

All other nodes can be reached from it by following edges or links. (In the formal definition, each such path is also unique.) In diagrams, the root node is conventionally drawn at the top. In some trees, such as heaps, the root node has

special properties. Every node in a tree can be seen as the root node of the subtree rooted at that node.

The height of a node is the length of the longest downward path to a leaf from that node. The height of the root is the height of the tree. The depth of a node is the length of the path to its root (i.e., its *root path*). This is commonly needed in the manipulation of the various self balancing trees, AVL Trees in particular.

The root node has depth zero, leaf nodes have height zero, and a tree with only a single node (hence both a root and leaf) has depth and height zero. Conventionally, an empty tree (tree with no nodes) has depth and height −1.

A subtree of a tree *T* is a tree consisting of a node in *T* and all of its descendants in *T*

3.1.4 Types of Tree

Binary Trees

The simplest form of tree is a binary tree. A binary tree consists of

a *node* (called the root node) and

left and right sub-trees.
Both the sub-trees are themselves binary trees.

You now have a *recursively defined data structure*. (It is also possible to define a list recursively: can you see how?)

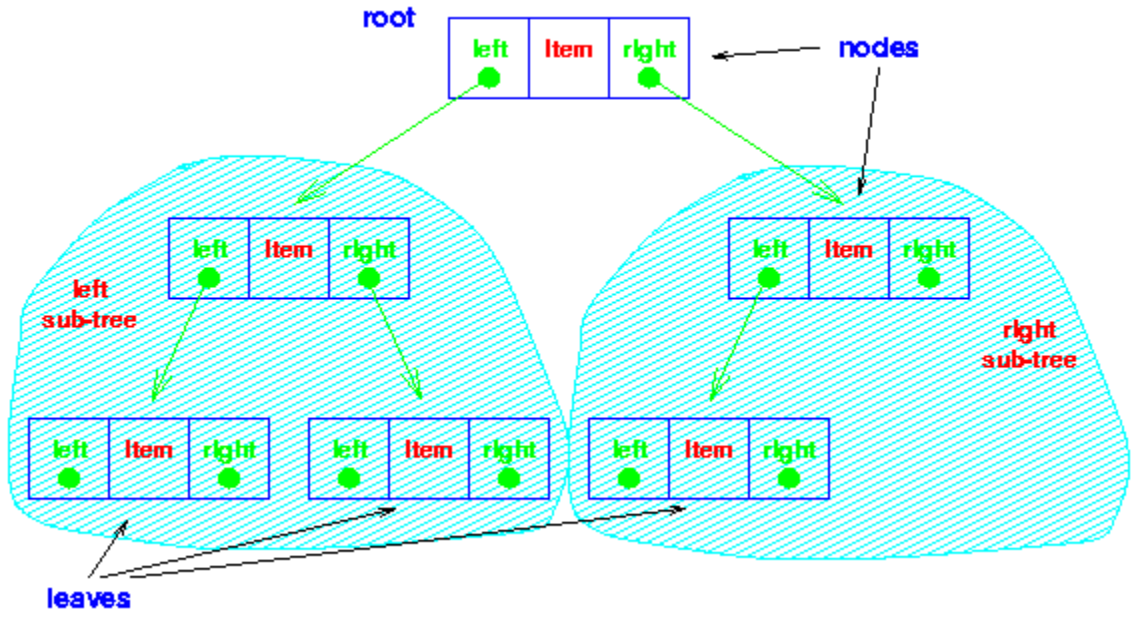


Figure 5: A binary tree

The nodes at the lowest levels of the tree (the ones with no sub-trees) are called leaves.

In an *ordered binary tree*,

the keys of all the nodes in the left sub-tree are less than that of the root,

the keys of all the nodes in the right sub-tree are greater than that of the root,

the left and right sub-trees are themselves ordered binary trees.

Binary Search Trees

Binary search tree (BST) or a lexicographic tree is a binary tree data structure which has the following binary search tree properties:

i. Each node has a value.

ii. The key value of the left child of a node is less than to the parent's key value.

iii. The key value of the right child of a node is greater than (or equal) to the parent's key value.

iv. And these properties hold true for every node in the tree.

If a BST allows duplicate values, then it represents a multiset. This kind of tree uses non-strict inequalities (<=, >=). Everything in the left subtree of a node is strictly less than the value of the node, but everything in the right subtree is either greater than or equal to the value of the node.

If a BST doesn't allow duplicate values, then the tree represents a set with unique values, like the mathematical set. Trees without duplicate values use strict inequalities, meaning that the left subtree of a node only contains nodes with values that are less than the value of the node, and the right subtree only contains values that are greater.

The choice of storing equal values in the right subtree only is arbitrary; the left would work just as well. One can also permit non-strict equality in both sides. This allows a tree containing many duplicate values to be balanced better, but it makes searching more complex.

Minimum **and** maximum

An element in a binary search tree whose key is a minimum can always be found by following the left child pointers from the root until a NULL is encountered. The maximal element can always be found by following the right child pointers from the root until NULL is encountered. The binary-search-tree property guarantees that these operations are correct.

Threaded Binary Trees

There are a lot of NULL pointers in the binary trees, these pointers exist, but they don’t point anywhere. They can be used in that maner, that they introduce a special kind of pointers, a so called threads. The threads are pointers that point on the predecessor, that is, on the successor of a given node in some of the traversal methods. In this

case, the tread points to the predecessor if it is a empty left link of the node, if the thread is unused right link of the node, then it points to the successor in the chosen notation (traversal).

This way the binary trees can be threaded in preorder, post order, and in order notation.

Here is an example of in order threaded binary tree:

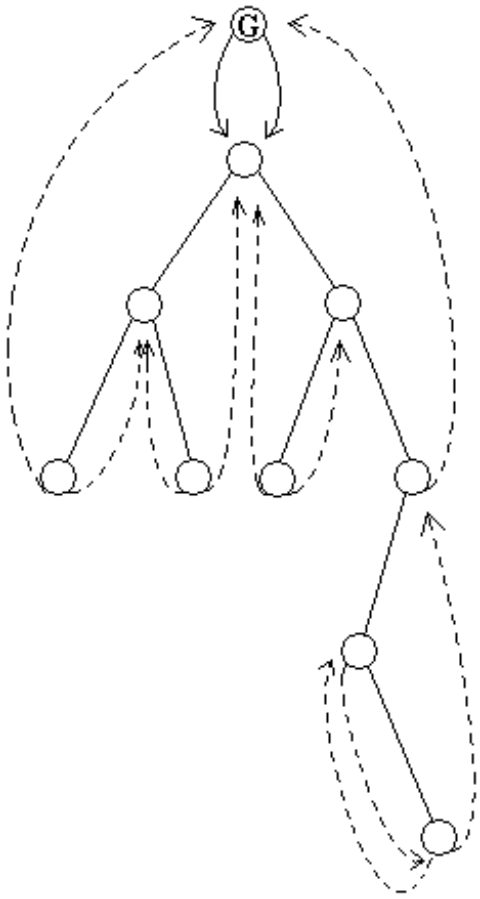


Figure *6:* Threaded binary trees

It can be noticed from the picture that the first and last node in the chosen traversal do not point at any node from the tree, while all other links in the tree are used. This way the procedures for traversing the tree are simplified.

Because of simpler programming implementation of the traversing of the tree (to know when to stop), and the two unused links to be used, it can be introduced a guide node, that will point at the root of the tree, and at him the two unused links will point.

The second problem that has to be solved after the introducing of the treads is the way of which the program that works with the tree will make difference from the moving across a real link or across a tread, because they both are realized with the same pointer. The simplest way is to be added an additional variable for each of the two pointers. These variables will tell if the appropriate pointer is a link or a tread.

In this case except for the info, llink and rlink, the node will contain two additional fields that will be as flags (labels) (commonly designated with ltag and rtag, that is, lbit and rbit). Its commonly taken that if rbit is 1, then the right link is a real link, while if its 0, then the right link is a tread. The same implies for the lbit.

Here is a typical representation of the treaded binary tree in C:

```
typedef int info_t;
struct node{
        info_t info;
struct node *llink, *rlink;
int lbit, rbit;
};
typedef struct node * nodep;
```

If the binary tree is empty then the guide node will point to himself.

Here below is a programming code in C for adding node in a binary tree that is inorder treaded:

```
int addNode(node * p, int n, char where)
{
node * temp;
if('l' == where){
if(p->ltag){
printf("There already exists a left subtree\n");
return (-1);
}
else{
temp=(node *)malloc(sizeof(node));
temp->info=n;
temp->rtag=temp->ltag=0;
temp->llink=p->llink;
temp->rlink=p;
p->llink=temp;
p->ltag=1;
return 0;
```

```
}


else if('r'==where){
if(p->rtag){
printf("There already exists a right subtree\n");
return(-1);
}
else{
temp=(node *)malloc(sizeof(node));
temp->info=n;
temp->rtag=temp->ltag=0;
temp->rlink=p->rlink;
temp->llink=p;
p->rlink=temp;
p->rtag=1;
return 0;
}
}
}
```

It is possible to discover the parent of a node from a inorder threaded binary tree, without explicit use of parent pointers or a stack, albeit slowly. This can be useful however where stack space is limited, or where a stack of parent pointers is unavailable.

This is possible, because if a node (k) has a right child (m) then m's left pointer must be either a child, or a thread back to k. In the case of a left child, that left child must also have a left child or a thread back to k, and so we can follow m's left children until we find a thread, pointing back to k. The situation is similar for when m is the left child of k, except in this case we have to follow the right children.

Similar analyst can be made for finding all the possible cases in which the discovering of the parents in preorder, and postorder treaded binary trees is possible.

Balanced binary trees

What is a complete tree:

The maximal number of nodes that can be found on the highest level (level 0) is one – 2^0, the root. The maximal number of nodes on level 1 are two – 2^1, on level 2 – four – 2^2, generaly on level m, there are 2^m number of nodes. So the maximal possible number of nodes in a tree with height h is:

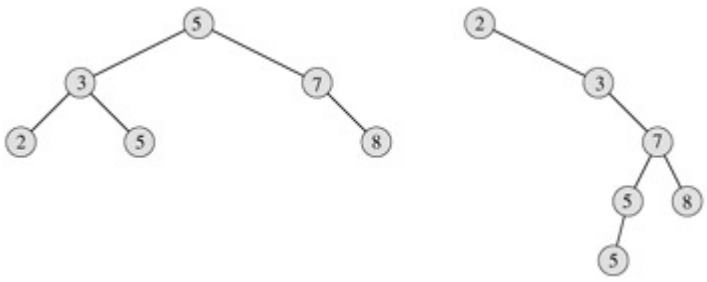
$2^0 + 2^1 + 2^2 + ... + 2^h = 2^{h+1} – 1$

If the tree have this property, then all unterminating nodes have two children, and this tree is then called a complete tree and then always the height *h* will be approximately lg*n.* So if the height of a tree is [lg*n*], and the number of nodes is *n,* then the tree is maximaly filled with the most minimal possible height. The completed tree has the property that all leaves are in the same level.

In every complete tree the number of nodes in the left subtree of the root (or every node) is equal to the number of nodes in the right subtree. The algorithms for binary searching, insertion, finding predecessor, successor can have the nice O(lg*n*) worst case of running time when the tree is complete. Every subtree in a complete tree is also a complete tree.

Also this tree has the property of being balanced. A balanced tree is every tree in which the difference of the heights between the two subtrees of any node in the tree, is zero or at most one. The tree on Figure *6*, *a*), is a balanced tree. But the tree on *b*) is not, the subtrees of the node 2 are very unbalanced, on one side we have an empty, totaly unexploited left subtree, and on the other side, we have a right subtree, that spreads to height five. The subtrees of the nodes 3 and 10, on *Figure 5* are also unbalanced, which is obvious.

The tree on *b*) has 6 nodes, but the height is 5, and lg6 is around 2.59, so the search alg. will not be optimal. To make it optimal we have to transform the tree to a balanced bin. tree with the same items: see *a*). In that case the search alg. will have compx. O(lg*n*). The tree is nearly completed that's why that 2.59, but it is balanced.



b)

Figure *7:* **Balanced binary trees**

Also with a balanced tree, in every next iteration of the binary search algorithm, around 1/2 of the all the nodes are skipped (taken out of consideration), and there are only 1/2 nodes left over to be searched, because one of the two subtrees of a root node is taken out of consideration in every next iteration. The iteration stopes when there is only 1 node left in consideration. From this we can also conclude why the binary searching of a balanced tree is always around O(lg*n*), and the text below in more details explains this:

The property of a balanced binary tree compulses the need of every node (beginning from the root) that has a left (or right) subtree that has height greater than 1, to have an opposite unemptied subtree that will have height as (or one level less) of the height of the opposite subtree. And this recursively holds true for every other lower level node. This also implies the two subtrees of every node to share nearly the same number of nodes.

Everything above said also implies that the balanced binary tree is always nearly completed. That's why the binary searching with balanced tree lasts in the worst case around O(lg*n*). To proof that every BST can be made balanced we'll say the following: The n items in every BST can be permuted in increasing sorted order.

If the median of the array of the sorted items is made the root of the tree, then for the left and right subtrees of the root there are left over almost equal number of nodes. For the left subtree we have the elements from 0, ...,n/2 – 1, and for the right subtree the elements from n/2 + 1, ...,n. If we take the median items of these two arrays as the roots of the left and right subtrees of the root, that is, left and right children of the root (we will name these nodes as R_l and R_r), then for the left and right subtrees of R_l and R_r we have of course also equal number of nodes in sorted order.

We'll continue this pattern until we don't insert all the nodes. This is how we can get a nearly completed tree – a balanced tree from every binary search tree, because we always trie to fill the two children of all nodes of the current level, and then go to the next level.

To implement this procedure in a programming language, we need to use some additional dinamically allocated structure (binary tree or a linked list) for saving the sorted array of nodes from the binary tree. We can express the complexity of this procedure by the recursive relation T(n)=n/2+2T(n/2).

Binary search trees that use algorithms for balancing are called self-balancing binary search trees.

AVL tree

AVL tree is a self-balancing binary search tree, and the first such data structure to be invented. The AVL algorithm is a dynamic method of balancing binary search trees. The AVL tree is named after its two inventors, Georgii Adel'son-Velsky and Yevgeniy Landis, who published it in their 1962 paper "An algorithm for the organization of information." In an AVL tree the heights of the two child subtrees of any node differ by

at most one, therefore it is also called height-balanced. Additions and deletions of tree items may require the tree to be rebalanced by one or more tree rotations.

The balance factor of a node is the height of its right subtree minus the height of its left subtree. A node with balance factor 1, 0, or -1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the heights of the subtrees.

This is a function in C for getting the height of some node:

```
/*level starts from –1*/

int h(node * root, int level){

if(root == NULL)

        return level+1;

else

return max(h(root->left,level+1),h(root->right,level+1));


}
```

AVL trees are often compared with red-black trees because red-black trees also take O(lg *n*) time for the basic operations. AVL trees perform better than red-black trees for lookup-intensive applications. The AVL tree balancing algorithm appears in many computer science curricula.

The AVL tree ensures at most (3 / 2) * *lg*(*n*) worst case scenario of binary search trees.

The idea behind maintaining the balanceness of an AVL tree is that whenever we insert or delete an item, we have disturbed the balancenes of the tree, and then we must restore it by performing a set of manipulations (called rotations) on the tree. These rotations come in two flavors: single rotations and double rotations. And each flavor has its corresponding left and right versions, these versions are symmetric to each other.

Permutation Trees

Trees can also be used to represent permutations, without repetition, and with repetitions. Not only that they are used to represent permutations, their tree traversals can be used for constructing algorithms that generate all permutations (with or without repetitions) for a given set of elements, only once for each different permutation.

In a traversal (preorder, postorder, inorder) of a permutation tree, every time when leaf is reached, the leaf together with the path of nodes from that leaf to the root is printed, and that’s how all the permutations are generated. Its more easy to generate in fly the

permutations (and put them in some linear structures) while the process of building the permutation tree is going on.

Here is an example of a permutation without repetition tree for the set: 1, 2, 3, 4:

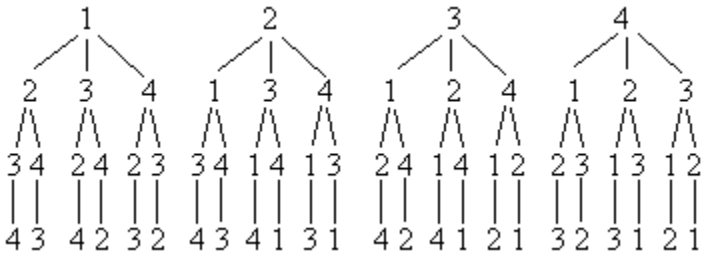


Figure 8: Permutation trees

A similar (it could be an infinite) tree can be made for permutations with repetitions.

Forest

While a subtree of a tree is again a tree, a disjoint union of two trees is no longer a tree, as it has no common root. If one generalizes trees to an ordered list of trees, one obtains an (ordered) forest. Given a node in a tree, its children define an ordered forest (the union of subtrees given by all the children, or equivalently taking the subtree given by the node itself and erasing the root).

Just as subtrees are natural for recursion (as in a depth-first search), forests are natural for corecursion (as in a breadth-first search).

Via mutual recursion, a forest can be defined as a list of trees (represented by root nodes), where a node (of a tree) consists of a value and a forest (its children).

Self-Assessment Exercise

1. Give the Declaration of the structure of a tree node
2. Differentiate between Binary tree and Binary search tree
3. Explain at least three types of tree you know

Self-Assessment Answer

```
typedef type_of_data_in_the_node info_t;
struct node{
info_t info;
struct node *llink;
struct node *rlink;
};
typedef node * nodep;
```

3.2 Operations on Tree

Enumerating all the items

Enumerating a section of a tree

Searching for an item

Adding a new item at a certain position on the tree(insertion)

Deleting an item

Pruning: Removing a whole section of a tree

Grafting: Adding a whole section to a tree

Finding the root for any node

Finding precessor and predecessor of a node

Some of these operations are explained below .

Successor and predecessor

Given a node in a binary search tree, it is sometimes important to be able to find its successor in sorted meaning of the word, which is determined by an inorder tree traverse. If all keys are distinct, the successor of a node *x* is the node with the smallest key greater than *key*[*x*]. The structure of a binary search tree allows us to determine the successor of a node without ever comparing keys.

The following procedure returns the successor of a node *x* in a binary search tree if it exists, and NULL if *x* has the largest key in the tree.

//*x* is the root, TREE-MINIMUM(*y*) returns the mostleft node (smallest) of the tree named *y*.

TREE-SUCCESSOR(*x*)

1 if *right*[*x*] ≠ NULL

2 then return TREE-MINIMUM (*right*[*x*])

3 *y* ← *p*[*x*]

4 while *y* ≠ NULL and *x* = *right*[*y*]

5 do *x* ← *y*

6 *y* ← *p*[*y*]

7 return *y*

The code for TREE-SUCCESSOR is broken into two cases. If the right subtree of node *x* is nonempty, then the successor of *x* is just the leftmost node in the right subtree, which is found in line 2 by calling TREE-MINIMUM(*right*[*x*]).

On the other hand, if the right subtree of node *x* is empty and *x* has a successor *y*, then *y* is the lowest ancestor of *x* whose left child is also an ancestor of *x.* To find *y*,

we simply go up the tree from *x* until we encounter a node that is the left child of its parent; this is accomplished by lines 3–7 of TREE-SUCCESSOR.

The running time of TREE-SUCCESSOR on a tree of height *h* is *O*(*h*), since we either follow a path up the tree or follow a path down the tree. The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in time *O*(*h*). Even if keys are not distinct, we define the successor and predecessor of any node *x* as the node returned by calls made to TREE-SUCCESSOR(*x*) and TREE-PREDECESSOR(*x*), respectively.

Insertion

Insertion begins as a search would begin; if the root is not equal to the value, we search the left or right subtrees as before. Eventually, we will reach a leaf and add the value as its right or left child, depending on the node's value. The Insert algorithm works like this: if the tree is empty we insert the node as a root of the tree, if not, then we examine the root value and recursively Insert the new node to the left subtree if the new value is less than the root, or the right subtree if the new value is greater than or equal to the root.

Here's how a typical binary search tree insertion might be performed in C, this insertion algorithm maintains the binary search tree properties in every next insertion, so the operations on the BST (like binary searching) can produce always correct results:

```
int insert(nodep * n, info_t x){
nodep p = *n;
if(p==NULL){
*n=(nodep)malloc(sizeof(node));
(*n)->info=x;
(*n)->left=(*n)->right=NULL;
return(1);
}
else if(x< p->info)
return(insert(&(p->left),x));
else if(x> p->info)
return(insert(&(p->right),x));
else
return(0);
}
```

The complexity of this algorithm is O(h), h is the height, and in average if the insertions of elements are in random series the complexity is around O(lgn), where n is total the number of nodes in the tree, and belongs to the set N.

O(h) is equivalent to O(lgn), because the height depends on the number of nodes (h=f(n)). We'll talk about this in more details later.

To insert a node in an empty tree, we simply make that node as a root, no mater what his value is. If we want to insert another node in order to build up the tree, we must insert that node as a child to the previosly added root node. If the key value of that node is less than the value of the root, then the node is inserted left of the root, as his left child, symmetrical operation is performed if the new node is greater than the root.

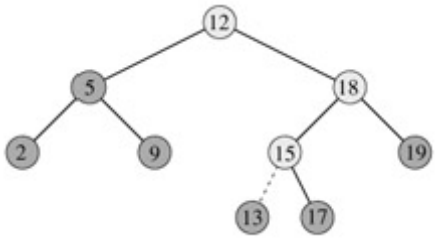


Figure 9: i*nserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.*

If we want to insert the items (7, 3, 10, 12, 2, 1, 13,) by that order, in a BST, by using the above algorithm, we'll get this tree:

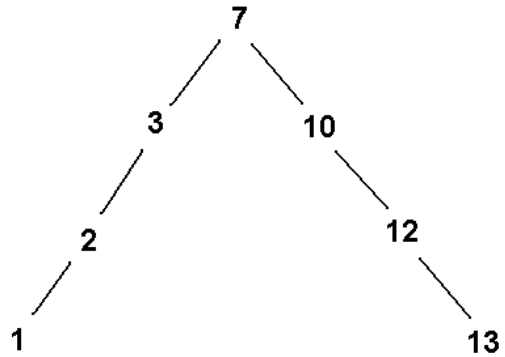


Figure 10

Searching

A binary search of a sorted array of *n* elements is a fast algorithm for searching, with complexity O(lg*n*), but the static size of the array can make the task sometimes uncomfortable. Linked list can increase the complexity of the binary searching from O(*lg*n) to the recursive comlexity function T(n) = n/2 + T(n/2), so linked lists are not used in binary searching. That's why a binary search tree can be used for BS. There is no need for sorting, because the elements are already ordered in sorted order, and

the complexity of a binary searching on a BST is averagely O(*lg*n), where n is the number of nodes.

Searching a binary tree for a specific value is a process that can be performed recursively because of the order in which values are stored. We begin by examining the root. If the value we are searching for equals the root, the value exists in the tree. If it is less than the root, then it must be in the left subtree, so we recursively search the left subtree in the same manner. Similarly, if it is greater than the root, then it must be in the right subtree, so we recursively search the right subtree. If we reach a leaf and have not found the value, then the item is not where it would be if it were present, so it is not contained in the tree at all. A comparison may be made with binary search, which operates in nearly the same way but using random access on an array instead of following links.

This operation requires O(*h*) (*h* is the height of the root). In average cases when the items of the tree are inserted in random (not sorted) series, the height is around lg*n* (*n* – number of items) and the operation requires O(lg*n*) time. But needs O(*n*) time in the worst-case, when the unbalanced tree resembles a linked list, that is, when the items are sorted while they are inserted.

Here is recursive implementation of binary searching in a BST, written in pseudocode:

//*x* is the root node of the tree; *k* is the key value that is in the interest of searching for.

TREE-SEARCH (*x*, *k*)

if *x*= NULL or *k* = *key*[*x*]

then return *x*

if *k* < *key*[*x*]

then return TREE-SEARCH(*left*[*x*], *k*)

else return TREE-SEARCH(*right*[*x*], *k*)

The search and insertion algorithms in every next iteration go one level lower, thus getting closer to the level which is equal to the height, and that's why they have complexity of O(*h*).

Deletion:

There are several cases to be considered:

Deleting a leaf: Deleting a node with no children is easy, as we can simply remove it from the tree. The node is deallocated and the pointer from the parent node, that pointed to the deleted node, will now have to get value NULL.

Deleting a node with one child: Delete it and replace the value of the parent pointer with the address of the child node of the deleted node.

Deleting a node with two children: A node with two children cannot be deleted easy as a node with one child. We well call this node *D*. Its obvious that the two children after the deletion of their parent cannot be attach with the parent of their parent.

The correct solution of the problem is the following:

D is the smallest in respect of all the nodes in his right subtree. And also the most-left node in the right subtree of *D* is for sure the smallest node of all nodes in the right subtree of *D,* and is also greater than all the nodes in the left subtree of *D*, we will call this node *N*.

So to keep the binarity and the lexicographical order of the tree we can remove the *N* node from his place as we remove a node with one or zero children and make the *N* node take the place of *D*, the pointers of *N* can take the values of the pointers of *D*, so *N* can point to the left and right subtrees of *D,* and then *D* can be deallocated.

The same, but opposite analogy can be applied with the left subtree of *D*.

The node *N* is the succesor of *D* in inorder traversal of the tree. The symmetcal node in the left subtree is the inorder predecessor of *D*.

Self Assessment Exercise

1. Differentiate between Binary tree and Binary search tree

Self-Assessment Answer

The simplest form of tree is a binary tree. A binary tree has these properties

a *node* (called the root node) and

left and right sub-trees.
Both the sub-trees are themselves binary trees.

While Binary search trees

Binary search tree (BST) or a lexicographic tree is a binary tree data structure which has the following binary search tree properties:

i. Each node has a value.
ii. The key value of the left child of a node is less than to the parent's key value.
iii. The key value of the right child of a node is greater than (or equal) to the parent's key value.
iv. And these properties holds true for every node in the tree.

3.3 Common Uses

i. Representing hierarchical data
ii. Storing data in a way that makes it easily searchable (see binary search tree and tree traversal)
iii. Representing sorted lists of data
iv. As a workflow for compositing digital images for visual effects

v. Routing algorithms

3.4 Tree Traverse

Stepping through the items of a tree, by means of the connections between parents and children, is called walking the tree, and the action is a walk of the tree (traverse). Often, an operation might be performed when a pointer arrives at a particular node (visiting the node – for example, printing the value/s that the node contains).

A walk in which each parent node is traversed before his children (or his subtrees) is called a pre-order walk; a walk in which the children are traversed before their respective parents are traversed, is called a postorder walk. There is also a level order traversal. In this traversal, until all nodes of one level (beginning from the zero level – the root) are not traversed, the traversing of the nodes on the next level is not started. Every tree traversal is of O(n) complexity, where n is the number of nodes in a tree.

Here are the more formal definitions of the preorder and postorder walks of a arbitrary tree: Let T be the tree, R the root. Let we have the structure of the tree, whose root is R, and it contains the subtrees T_1, T_2, ..., T_n. Then:

Preorder traverse of the nodes of T is an array, that contains the information from the root R, by which are followed the information from the nodes of the subtree T_1 traversed in preorder, and then the information from the nodes of the subtree T_2 traversed in preorder, ..., then T_3 in preorder, T_4, ..., to T_n in preorder.

Postorder traverse of the nodes of T is an array that contains the arrays of information from the nodes of the subtrees T_1, T_2, ..., T_n, traversed in postorder, starting from T_1, T_2, ..., to T_n, and on the end the information from the root R.

Here are recursive realizations of the preorder and postorder algorithms in C for traversing a customary (arbitrary) tree represented by a binary tree, and it's the same for preorder and postorder traversal of a binary tree:

```
void preorder(nodep p){
if(p!=NULL){
printf("%d ",p->info);
preorder(p->llink);
preorder(p->rlink);
}
}
```

```
void postorder(nodep p){
if(p!=NULL){
preorder(p->llink);
preorder(p->rlink);
printf("%d ",p->info);
}
}
```

Because every tree can be represented with a binary tree, the traversals for binary trees can be used for traversing arbitrary tree structure, if the tree is represented by a binary tree.

When we traverse a tree in postorder every child (subtree) must first be visited before his parent (root) is visited, and a node cannot be visited, if previously his children (subtrees) are not visited (printed on the screen).

If we want to implement the postorder traversal iteratively, we need to use stack, that will store the chain of ancestor nodes, previously encountered but not actually visited (printed on the screen), because they all have children (subtrees) that must be visited first before they are visited. First beginning from the root, if he has children then the root is pushed in top of the stack, then, after that, his children are attempted to be visited, starting from the first one, but if that one also has children (subtrees), and the rule holds first the children and then the root, then this one is also pushed on the top of the stack too, and then this activity goes one, until we reach a node that have children leaves, they are not pushed in the stack, just visited, and then the parent node of these nodes is visited, which is on the top of the stack now, then he is poped out, and then there is an attempt to visit the rest child nodes of the node that is now on the top of the stack, that is the parent of the previously visited node. So the same recursive procedure is executed on all the nodes in the tree (on the top of stack), until the stack is not empty.

Self Assessment Exercise

1. Explain at least other three types of tree you know

Self-Assessment Answer

AVL tree

AVL tree is a self-balancing binary search tree, and the first such data structure to be invented. The AVL algorithm is a dynamic method of balancing binary search trees. The AVL tree is named after its two inventors, Georgii Adel'son-Velsky and Yevgeniy Landis, who published it in their 1962 paper "An algorithm for the organization of information." In an AVL tree the heights of the two child subtrees of any node differ by at most one, therefore it is also called height-balanced. Additions and deletions of tree items may require the tree to be rebalanced by one or more tree rotations.

The balance factor of a node is the height of its right subtree minus the height of its left subtree. A node with balance factor 1, 0, or -1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the heights of the subtrees.

The AVL tree ensures at most (3 / 2) * *lg*(*n*) worst case scenario of binary search trees.

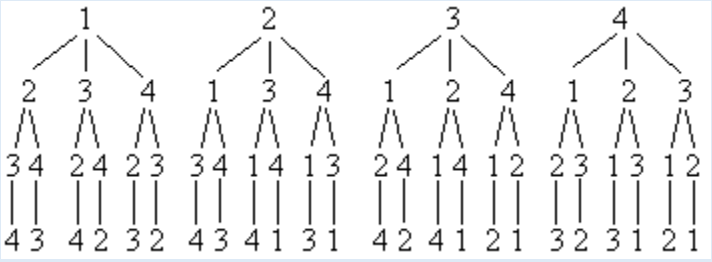
The idea behind maintaining the balanceness of an AVL tree is that whenever we insert or delete an item, we have disturbed the balanceness of the tree, and then we must restore it by performing a set of manipulations (called rotations) on the tree. These rotations come in two flavors: single rotations and double rotations. And each flavor has its corresponding left and right versions, these versions are symmetric to each other.

Permutation trees

Trees can also be used to represent permutations, without repetition, and with repetitions. Not only that they are used to represent permutations, their tree traversals can be used for constructing algorithms that generate all permutations (with or without repetitions) for a given set of elements, only once for each different permutation. In a traversal (preorder, postorder, inorder) of a permutation tree, every time when leaf is reached, the leaf together with the path of nodes from that leaf to the root is printed, and that's how all the permutations are generated.

Its more easy to generate in fly the permutations (and put them in some linear structures) while the process of building the permutation tree is going on.

Here is an example of a permutation without repetition tree for the set: 1, 2, 3, 4:



A similar (it could be an infinite) tree can be made for permutations with repetitions.

Forest

While a subtree of a tree is again a tree, a disjoint union of two trees is no longer a tree, as it has no common root. If one generalizes trees to an ordered list of trees, one obtains an (ordered) forest. Given a node in a tree, its children define an ordered forest (the union of subtrees given by all the children, or equivalently taking the subtree given by the node itself and erasing the root). Just as subtrees are natural for recursion (as in a depth-first search), forests are natural for corecursion (as in a breadth-first search).

Via mutual recursion, a forest can be defined as a list of trees (represented by root nodes), where a node (of a tree) consists of a value and a forest (its children):

An iterative version of preorder is implemented also by using stack. Only this time, after every visited node (starting from the root) the children (subtrees) must be visited, and this holds true for every next visited node. This means that after visiting a node, all his children are not visited at once.

There is actually an attempt of applying the same recursive procedure on his children (subtrees), one by one. So after a visited root, the children are attempted to be visited, but after a visited child, it turns out that, that node has children too, and the rule still holds, after the root, the children must be visited. The other children (subtrees) of the root will be visited after is done with the first one. The node (starting from the root) that is visited in the moment is not pushed in the stack.

After visiting the root, all his children are pushed in the stack. Then there is an attempt to visit them all, starting from the one on top of the stack, that one is visited, but if it was turned out that that one has children too, and according to the rule: after a visited node, there must be a visit to his children, the node is after that poped out of the stack, because it's done with it (it was visited), and then his children are pushed in the stack, and then the same recursive procedure is applied to the node on the top of the stack. The procedure stops when the stack gets empty.

A stack must be used in the iterative traversals because, for postorder: to remember the nodes in the right hierarchical order, because the visiting (printing) starts from the bottom, from the leaves and then level by level, it goes up in the tree hierarchy following the nodes (ordered chain from ancestors) in the stack, and to remember the children that are not yet traversed recursively.

For the preorder, a stack is used to store the children in the right hierarchical order that are not yet traversed. The detailed explanation of the traversals above, practically explains the need for stack. The recursive implementations use stack also to remember the positions of where the procedures stoped, because of the recursive calls, so they can be continued later, in right order, after the recursively called functions complete.

And finally saying, if the nodes had also pointer to their parent, then there would be no need for stack. Here is an example of preorder and postorder traversal of the tree on *Picture 1*:

When a node is visited, it is printed on the screen, as an output of the traverse algorithm.

Preorder:

5, 4, 9, 10, 1, 2, 9, 1, 8.

Postorder:

8, 1, 9, 2, 1, 10, 9, 4, 5.

Self Assessment Exercise

1. Write an algorithm for insertion operation on a tree.

Self-Assessment Answer

```
int insert(nodep * n, info_t x){
nodep p = *n;
if(p==NULL){
*n=(nodep)malloc(sizeof(node));
(*n)->info=x;
(*n)->left=(*n)->right=NULL;
return(1);
}
else if(x< p->info)
return(insert(&(p->left),x));
else if(x> p->info)
return(insert(&(p->right),x));
else
return(0);
}
```

4.0 Conclusion

A tree can be defined as a collection of nodes (starting at a root node), where each node is a data structure consisting of a value, together with a list of nodes (the "children"), with the constraints that no node is duplicated. A tree can be defined abstractly as a whole (globally) as an ordered tree, with a value assigned to each node. A tree has in different terminologies such as a node, root node, children, parent node, internal node, external node, height and depth of a node. The representation and the operations on a tree are defined. Also the uses of a tree as a data structure in algorithm and traverse method on a tree were all envisaged. All these are the concept of tree data structure in algorithm as a course.

5.0 Summary

In this Module 3 Study Unit 1, the following aspects have been discussed:

1. Representation of a tree
2. Operations and applications of a tree
3. Tree Traverse and the terms used in traverse

6.0 Tutor Marked Assignments

1. Explain the concepts of traversing on a tree
2. Explain deletion process of a node in a tree
3. How can you search for anode on a tree?

7.0 References/ Further Reading

Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89683-4 . Section 2.3: Trees, pp. 308–423.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7 . Section 10.4: Representing rooted trees, pp. 214–217. Chapters 12–14 (Binary Search Trees, Red-Black Trees, Augmenting Data Structures), pp. 253–320.

Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. "Introduction to Algorithms." MIT Press. 1990.

Scott Mitchell, Microsoft Web technologies (January 1998). University of California – San Diego. mitchell@4guysfromrolla.com, or via his blog at http://ScottOnWriting.NET.

Unit 2

Graph

Contents

1.0 Introduction

2.0 Learning Outcomes

3.0 Learning Contents

- 3.1 Representation of a Graph
- 3.2 Operations on Graph
- 3.3 Graph Traverse

4.0 Conclusion

5.0 Summary

6.0 Tutor Marked Assignments

7.0 References/ Further Readings

1.0 Introduction

In computer science, a graph is an abstract data type. A graph data structure consists of a finite (and possibly mutable) set of ordered pairs, called edges or arcs, of certain entities called nodes or vertices. As in mathematics, an edge (*x*,*y*) is said to point or go from *x* to *y*. The nodes may be part of the graph structure, or may be external entities represented by integer indices or references.

A graph data structure may also associate to each edge some edge value, such as a symbolic label or a numeric attribute (cost, capacity, length, etc.).

Figure 1 shows three examples of graphs. Notice that graphs, unlike trees, can have sets of nodes that are disconnected from other sets of nodes. For example, graph (a) has two distinct, unconnected set of nodes.

Graphs can also contain cycles. Graph (b) has several cycles. One such is the path from v1 to v2 to v4 and back to v1. Another one is from v1 to v2 to v3 to v5 to v4 and back to v1. (There are also cycles in graph (a).) Graph (c) does not have any cycles, as one less edge than it does number of nodes, and all nodes are reachable. Therefore, it is a tree.

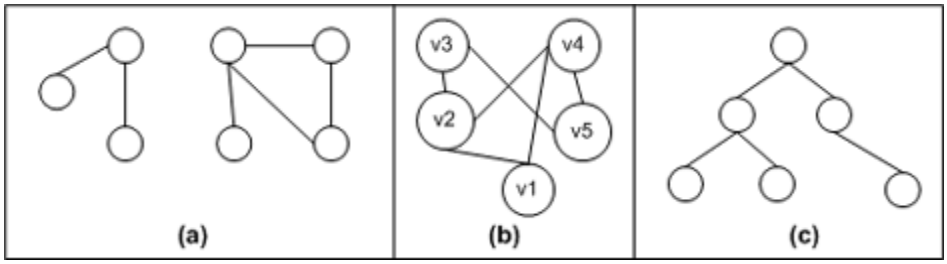


Figure 11. Three examples of graphs

Graph algorithms are a significant field of interest within computer science. Typical higher-level operations associated with graphs are: finding a path between two nodes, like depth-first search and breadth-first search and finding the shortest path from one node to another, like Dijkstra's algorithm. A solution to finding the shortest path from each node to every other node also exists in the form of the Floyd–Warshall algorithm.

A directed graph can be seen as a flow network, where each edge has a capacity and each edge receives a flow. The Ford–Fulkerson algorithm is used to find out the maximum flow from a source to a sink in a graph.

2.0 Learning Outcomes

At the end of the lesson the students should be able to:

i. Perform representation of a Graph;

ii. Define Operations on Graph;

iii. Perform Graph Traverse.

3.0 Learning Contents

3.1 Representation of a Graph

A graph is a set of *vertices* and *edges* which connect them. We write:

$G = (V,E)$

where V is the set of vertices and the set of edges,

$E = \{ (v_i,v_j) \}$

where v_i and v_j are in V.

Paths

A *path*, p, of length, k, through a graph is a sequence of connected vertices:

$p = <v_0,v_1,...,v_k>$

where, for all i in (0,k-1:

(v_i,v_{i+1}) is in E.

Cycles

A graph contains no *cycles* if there is no path of non-zero length through the graph, $p = <v_0,v_1,...,v_k>$ such that $v_0 = v_k$.

Spanning Trees

A *spanning tree* of a graph, G, is a set of |V|-1 edges that connect all vertices of the graph.

Minimum Spanning Tree

In general, it is possible to construct multiple spanning trees for a graph, G. If a cost, c_{ij}, is associated with each edge, $e_{ij} = (v_i,v_j)$, then the minimum spanning tree is the set of edges, E_{span}, forming a spanning tree, such that:

$C = sum(c_{ij} \mid all\ e_{ij}\ in\ E_{span})$

is a minimum.

Different data structures for the representation of graphs are used in practice:

Self Assessment Exercise

1. What is a graph?

Self-Assessment Answer

A graph is a set of *vertices* and *edges* which connect them. We write: G = (V,E)
where V is the set of vertices and the set of edges, E = { (v_i,v_j) }
where v_i and v_j are in V.

Adjacency List

Vertices are stored as records or objects, and every vertex stores a list of adjacent vertices. This data structure allows the storage of additional data on the vertices. The Graph class contains a NodeList holding the set of GraphNodes that constitute the nodes in the graph. That is, a graph is represented by a set of nodes, and each node maintains a list of its neighbors. Such a representation is called an adjacency list, and is depicted graphically in Figure 4.

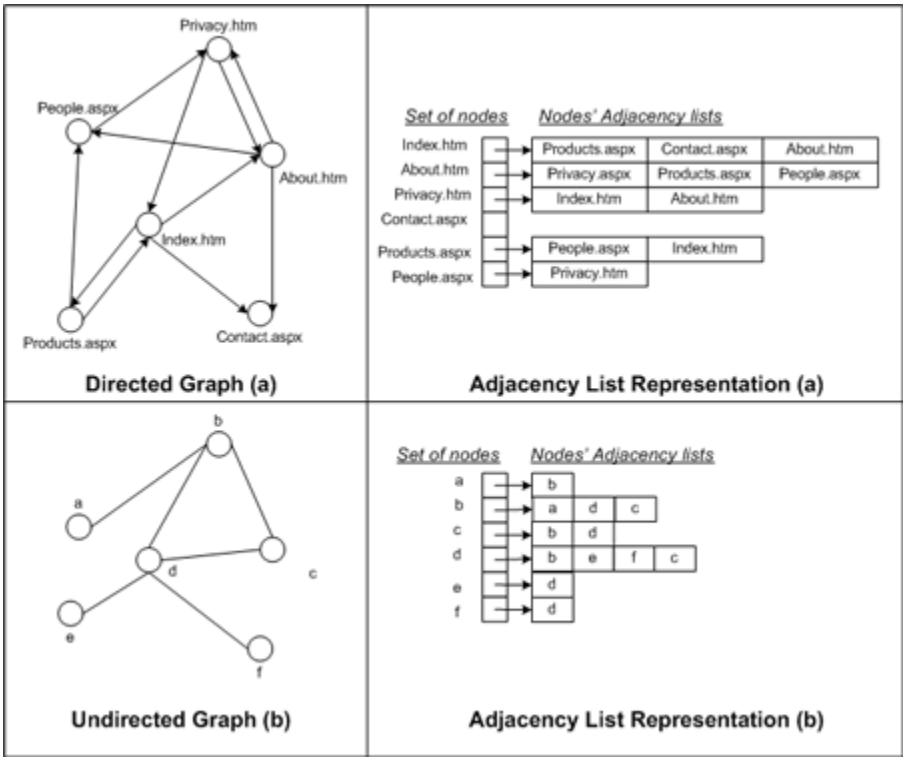


Figure 12: Adjacency list representation in graphical form

Vertices and edges are stored as records or objects. Each vertex stores its incident edges, and each edge stores its incident vertices. This data structure allows the storage of additional data on vertices and edges.

Adjacency matrix

A two-dimensional matrix, in which the rows represent source vertices and columns represent destination vertices. Data on edges and vertices must be stored externally. Only the cost for one edge can be stored between each pair of vertices. An alternative method for representing a graph is to use an adjacency matrix. For a graph with n

nodes, an adjacency matrix is an n x n two-dimensional array. For weighted graphs the array element (u, v) would give the cost of the edge between u and v (or, perhaps -1 if no such edge existed between u and v. For an unweighted graph, the array could be an array of Booleans, where a True at array element (u, v) denotes an edge from u to v and a False denotes a lack of an edge.

Figure 2 depicts how an adjacency matrix representation in graphical form

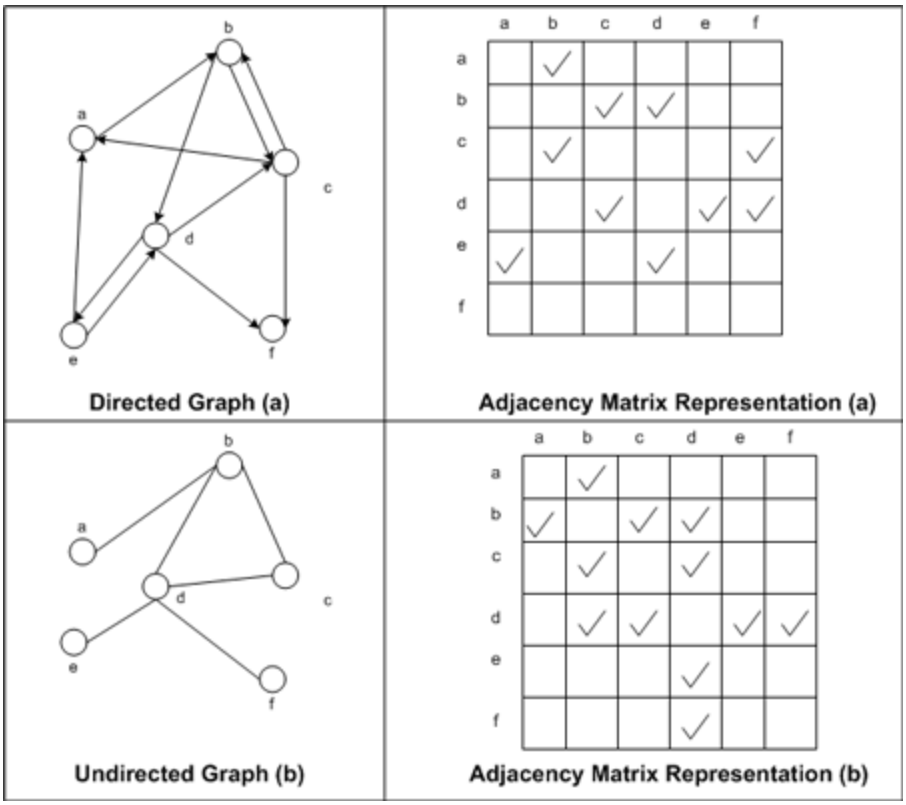


Figure 13. Adjacency matrix representation in graphical form

Incidence matrix

A two-dimensional Boolean matrix, in which the rows represent the vertices and columns represent the edges. The entries indicate whether the vertex at a row is incident to the edge at a column.

The following table gives the time complexity cost of performing various operations on graphs, for each of these representations. In the matrix representations, the entries encode the cost of following an edge. The cost of edges that are not present are assumed to be ∞.

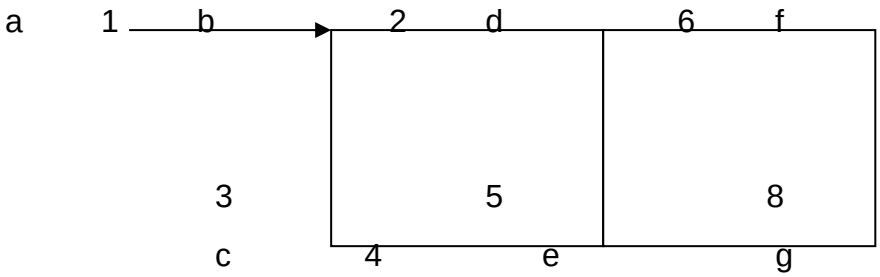
Adjacency lists are generally preferred because they efficiently represent sparse

	Adjacency list	Incidence list	Adjacency matrix	Incidence matrix
Storage	$O(\|V\| + \|E\|)$	$O(\|V\| + \|E\|)$	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$
Add vertex	$O(1)$	$O(1)$	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$
Add edge	$O(1)$	$O(1)$	$O(1)$	$O(\|V\| \cdot \|E\|)$
Remove vertex	$O(\|E\|)$	$O(\|E\|)$	$O(\|V\|^2)$	$O(\|V\| \cdot \|E\|)$
Remove edge	$O(\|E\|)$	$O(\|E\|)$	$O(1)$	$O(\|V\| \cdot \|E\|)$
Query: are vertices u, v adjacent? (Assuming that the storage positions for u, v are known)	$O(\|V\|)$	$O(\|E\|)$	$O(1)$	$O(\|E\|)$
Remarks	When removing edges or vertices, need to find all vertices or edges		Slow to add or remove vertices, because matrix must be resized/copied	Slow to add or remove vertices and edges, because matrix must be resized/copied

graphs. An adjacency matrix is preferred if the graph is dense, that is the number of edges $|E|$ is close to the number of vertices squared, $|V|^2$, or if one must be able to quickly look up if there is an edge connecting two vertices.

Example: Draw the graph of this incidence matrix

	1	2	3	4	5	6	7	8
a	1	0	0	0	0	0	0	0
b	1	1	1	0	0	0	0	0
c	0	0	1	1	0	0	0	0
d	0	1	0	0	1	1	0	0
e	0	0	1	1	0	0	1	0
f	0	0	0	0	0	1	0	1
g	0	0	0	0	0	0	1	1



Self Assessment Exercise

1. What is the difference between a graph and a tree?

Self-Assessment Answer

Graph (c) does not have any cycles, as one less edge than it does number of nodes, and all nodes are reachable. Therefore, it is a tree.

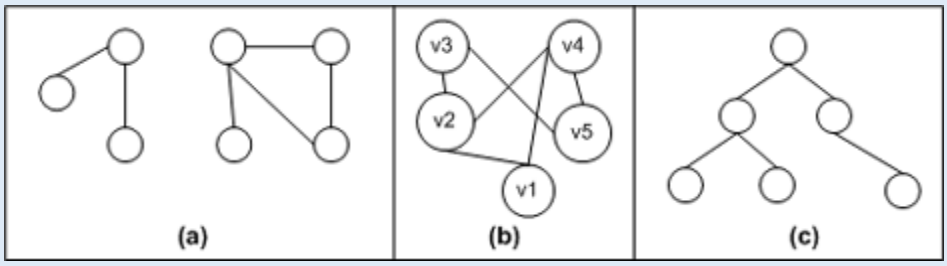


Figure 1. Three examples of graphs

3.2 Operations on Graph

The basic operations provided by a graph data structure *G* usually include:

adjacent(*G*, *x*, *y*): tests whether there is an edge from node *x* to node *y*.

neighbors(*G*, *x*): lists all nodes *y* such that there is an edge from *x* to *y*.

add(*G*, *x*, *y*): adds to *G* the edge from *x* to *y*, if it is not there.

delete(*G*, *x*, *y*): removes the edge from *x* to *y*, if it is there.

get_node_value(*G*, *x*): returns the value associated with the node *x*.

set_node_value(*G*, *x*, *a*): sets the value associated with the node *x* to *a*.

Structures that associate values to the edges usually also provide:

get_edge_value(*G*, *x*, *y*): returns the value associated to the edge (*x*,*y*).

set_edge_value(*G*, *x*, *y*, *v*): sets the value associated to the edge (*x*,*y*) to *v*.

Self Assessment Exercise(SAE 3)

Explain these operations been performed on a graph G

delete(*G*, *x*, *y*)

add(*G*, *x*, *y*)

neighbors(*G*, *x*)

set_node_value(*G*, *x*, *a*)

set_edge_value(*G*, *x*, *y*, *v*)

3.3 Graph Traverse

Graph traversal refers to the problem of visiting all the nodes in a graph in a particular manner. Tree traversal is a special case of graph traversal. In contrast to tree traversal, in general graph traversal, each node may have to be visited more than once, and a root-like node that connects to all other nodes might not exist.

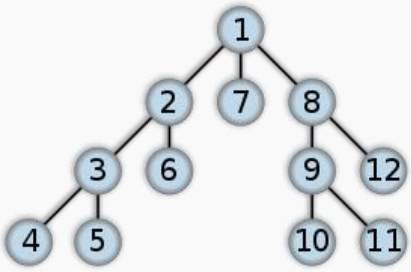


Figure 14 Order in which the nodes are visited

Class	Search algorithm
Data structure	Graph
Worst case performance	$O(\|V\| + \|E\|)$ for explicit graphs traversed without repetition, $O(b^d)$ for implicit graphs with branching factor *b* searched to depth *d*
Worst case space complexity	$O(\|V\|)$ if entire graph is traversed without repetition, O(longest path length searched) for implicit graphs without elimination of duplicate nodes

Breadth-first Search

A Breadth-first search (BFS) is another technique for traversing a finite undirected graph. BFS visits the sibling nodes before visiting the child nodes. Usually a queue is used in the search process. In graph theory, breadth-first search (BFS) is a strategy for searching in a graph when search is limited to essentially two operations: (a) visit and inspect a node of a graph; (b) gain access to visit the nodes that neighbor the currently visited node.

The BFS begins at a root node and inspect all the neighboring nodes. Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on. Compare it with the depth-first search.

How it works

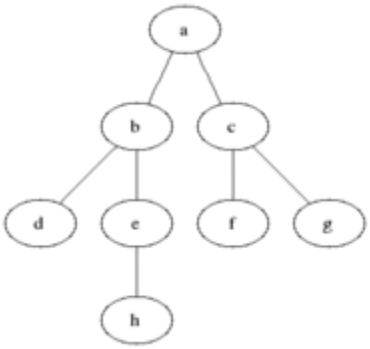


Figure 15: Animated example of a breadth-first search

BFS is an uninformed search method that aims to expand and examine all nodes of a graph or combination of sequences by systematically searching through every solution. In other words, it exhaustively searches the entire graph or sequence without considering the goal until it finds it.

From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a FIFO (i.e., First In, First Out) queue.

In typical implementations, nodes that have not yet been examined for their neighbors are placed in some container (such as a queue or linked list) called "open" and then once examined are placed in the container "closed".

Algorithm

The breadth-first tree obtained when running BFS on the given map and starting in Frankfurt

The algorithm uses a queue data structure to store intermediate results as it traverses the graph, as follows:

Enqueue the root node

Dequeue a node and examine it

If the element sought is found in this node, quit the search and return a result.

Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.

If the queue is empty, every node on the graph has been examined – quit the search and return "not found".

If the queue is not empty, repeat from Step 2.

Note: Using a stack instead of a queue would turn this algorithm into a depth-first search.

Pseudocode

Input: A graph *G* and a root *v* of G

```
1  procedure BFS(G,v):
2      create a queue Q
3      enqueue v onto Q
4      mark v
5      while Q is not empty:
6          t ← Q.dequeue()
7          if t is what we are looking for:
8              return t
9          for all edges e in G.incidentEdges(t) do
10             o ← G.opposite(t,e)
11             if o is not marked:
12                 mark o
13                 enqueue o onto Q
```

Features

Space complexity

When the number of vertices in the graph is known ahead of time, and additional data structures are used to determine which vertices have already been added to the queue, the space complexity can be expressed as $O(|V|)$where $|V|$is the cardinality of the set of vertices.

Time complexity

The time complexity can be expressed as $O(|E| + |V|)$since every vertex and every edge will be explored in the worst case. Note: $O(|E| + |V|)$may vary between $O(|V|)$and $O(|V|^2)$, depending on known estimates of the number of graph edges.

Applications

Breadth-first search can be used to solve many problems in graph theory, for example:

Finding all nodes within one connected component

Copying Collection, Cheney's algorithm

Finding the shortest path between two nodes *u* and *v* (with path length measured by number of edges)

Testing a graph for bipartiteness

(Reverse) Cuthill–McKee mesh numbering

Ford–Fulkerson method for computing the maximum flow in a flow network

Serialization/Deserialization of a binary tree vs serialization in sorted order, allows the tree to be re-constructed in an efficient manner.

Finding connected components

The set of nodes reached by a BFS (breadth-first search) form the connected component containing the starting node.

Testing bipartiteness

BFS can be used to test bipartiteness, by starting the search at any vertex and giving alternating labels to the vertices visited during the search. That is, give label 0 to the starting vertex, 1 to all its neighbours, 0 to those neighbours' neighbours, and so on. If at any step a vertex has (visited) neighbours with the same label as itself, then the graph is not bipartite. If the search ends without such a situation occurring, then the graph is bipartite.

Self Assessment Exercise

1. Explain these operations being performed on a graph G

delete(*G*, *x*, *y*)

add(*G*, *x*, *y*)

neighbors(*G*, *x*)

set_node_value(*G*, *x*, *a*)

set_edge_value(*G*, *x*, *y*, *v*)

Self-Assessment Answer

delete(*G*, *x*, *y*): removes the edge from *x* to *y*, if it is there.

add(*G*, *x*, *y*): adds to *G* the edge from *x* to *y*, if it is not there.

neighbors(*G*, *x*): lists all nodes *y* such that there is an edge from *x* to *y*.

set_node_value(*G*, *x*, *a*): sets the value associated with the node *x* to *a*.

set_edge_value(*G*, *x*, *y*, *v*): sets the value associated to the edge (*x*,*y*) to *v*.

Depth-first Search

A Depth-first search (DFS) is a technique for traversing a finite undirected graph. DFS visits the child nodes before visiting the sibling nodes, that is, it traverses the depth of the tree before the breadth. Usually a stack is used in the search process.

Depth-first search (DFS) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

A version of depth-first search was investigated in the 19th century by French mathematician Charles Pierre Trémaux[1] as a strategy for solving mazes.[2][3]

Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hasn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a stack for exploration.

Properties.

The time and space analysis of DFS differs according to its application area. In theoretical computer science, DFS is typically used to traverse an entire graph, and takes time O(|V| + |E|), linear in the size of the graph. In these applications it also uses space O(|V|) in the worst case to store the stack of vertices on the current search path as well as the set of already-visited vertices.

Thus, in this setting, the time and space bounds are the same as for breadth-first search and the choice of which of these two algorithms to use depends less on their complexity and more on the different properties of the vertex orderings the two algorithms produce. For applications of DFS to search problems in artificial intelligence, however, the graph to be searched is often either too large to visit in its entirety or even infinite, and DFS may suffer from non-termination when the length of a path in the search tree is infinite.

Therefore, the search is only performed to a limited depth, and due to limited memory availability one typically does not use data structures that keep track of the set of all previously visited vertices. In this case, the time is still linear in the number of expanded vertices and edges (although this number is not the same as the size of the entire graph because some vertices may be searched more than once and others not at all) but the space complexity of this variant of DFS is only proportional to the depth limit, much smaller than the space needed for searching to the same depth using breadth-first search.

For such applications, DFS also lends itself much better to heuristic methods of choosing a likely-looking branch. When an appropriate depth limit is not known a priori, iterative deepening depth-first search applies DFS repeatedly with a sequence of increasing limits; in the artificial intelligence mode of analysis, with a branching factor greater than one, iterative deepening increases the running time by only a constant factor over the case in which the correct depth limit is known due to the geometric growth of the number of nodes per level.

DFS may be also used to collect a sample of graph nodes. However, incomplete DFS, similarly to incomplete BFS, is biased towards nodes of high degree.

For the following graph:

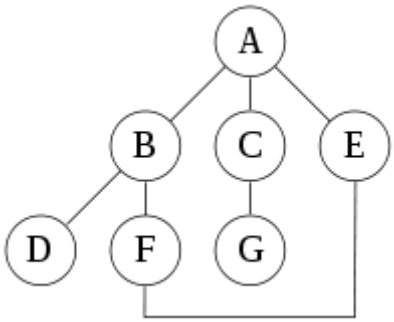


Figure 16: Showing the Depth First Search

Self Assessment Exercise

1. Discuss the concept and application of Breadth First Search?

Self-Assessment Answer

Concept of BFS

In graph theory, breadth-first search (BFS) is a strategy for searching in a graph when search is limited to essentially two operations: (a) visit and inspect a node of a graph; (b) gain access to visit the nodes that neighbor the currently visited node. The BFS begins at a root node and inspect all the neighboring nodes. Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on.

Application

Finding all nodes within one connected component

Copying Collection, Cheney's algorithm

Finding the shortest path between two nodes *u* and *v* (with path length measured by number of edges)

Testing a graph for bipartiteness

(Reverse) Cuthill–McKee mesh numbering

Ford–Fulkerson method for computing the maximum flow in a flow network

Serialization/Deserialization of a binary tree vs serialization in sorted order, allows the tree to be re-constructed in an efficient manner.

A depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G. The edges traversed in this search form a Trémaux tree, a structure with important applications in graph theory.

Performing the same search without remembering previously visited nodes results in visiting nodes in the order A, B, D, F, E, A, B, D, F, E, etc. forever, caught in the A, B, D, F, E cycle and never reaching C or G.

Iterative deepening is one technique to avoid this infinite loop and would reach all nodes.

Output of a depth-first search.

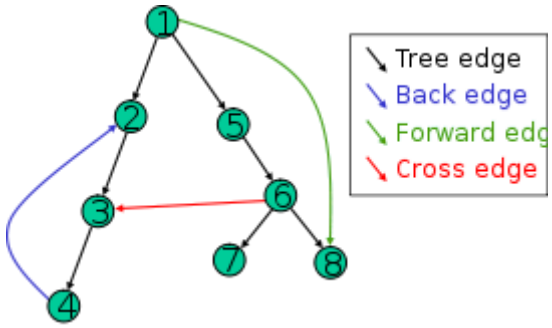


Figure 17: The four types of edges defined by a spanning tree

A convenient description of a depth first search of a graph is in terms of a spanning tree of the vertices reached during the search. Based on this spanning tree, the edges of the original graph can be divided into three classes: forward edges, which point from a node of the tree to one of its descendants, back edges, which point from a node to one of its ancestors, and cross edges, which do neither. Sometimes tree edges, edges which belong to the spanning tree itself, are classified separately from forward edges. If the original graph is undirected then all of its edges are tree edges or back edges.

Vertex orderings

It is also possible to use the depth-first search to linearly order the vertices of the original graph (or tree). There are three common ways of doing this:

A preordering is a list of the vertices in the order that they were first visited by the depth-first search algorithm. This is a compact and natural way of describing the progress of the search, as was done earlier in this article. A preordering of an expression tree is the expression in Polish notation.

A postordering is a list of the vertices in the order that they were *last* visited by the algorithm. A postordering of an expression tree is the expression in reverse Polish notation.

A reverse postordering is the reverse of a postordering, i.e. a list of the vertices in the opposite order of their last visit. Reverse postordering is not the same as preordering. For example, when searching the directed graph

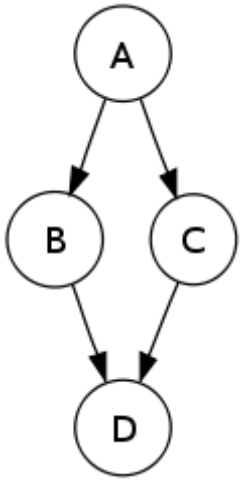


Figure 18: Reverse Postordering

Beginning at node A, one visits the nodes in sequence, to produce lists either A B D B A C A, or A C D C A B A (depending upon whether the algorithm chooses to visit B or C first). Note that repeat visits in the form of backtracking to a node, to check if it has still unvisited neighbours, are included here (even if it is found to have none).

Thus the possible preorderings are A B D C and A C D B (order by node's leftmost occurrence in above list), while the possible reverse postorderings are A C B D and A B C D (order by node's rightmost occurrence in above list). Reverse postordering produces a topological sorting of any directed acyclic graph. This ordering is also useful in control flow analysis as it often represents a natural linearization of the control flow. The graph above might represent the flow of control in a code fragment like

```
if (A) then {
  B
} else {
```

C

}

D

and it is natural to consider this code in the order A B C D or A C B D, but not natural to use the order A B D C or A C D B.

Pseudocode

Input: A graph *G* and a vertex *v* of G

Output: A labeling of the edges in the connected component of v as discovery edges and back edges

```
1  procedure DFS(G,v):
2      label v as explored
3      for all edges e in G.incidentEdges(v) do
4          if edge e is unexplored then
5              w ← G.opposite(v,e)
6              if vertex w is unexplored then
7                  label e as a discovery edge
8                  recursively call DFS(G,w)
9              else
10                 label e as a back edge
```

Figure 19: Randomized algorithm similar to depth-first search used in generating a maze.

Applications

Algorithms that use depth-first search as a building block include:

- ✓ Finding connected components.
- ✓ Topological sorting.
- ✓ Finding 2-(edge or vertex)-connected components.
- ✓ Finding 3-(edge or vertex)-connected components.
- ✓ Finding the bridges of a graph.
- ✓ Finding strongly connected components.
- ✓ Planarity testing
- ✓ Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)
- ✓ Maze generation may use a randomized depth-first search.
- ✓ Finding biconnectivity in graphs.

Self Assessment Exercise

1. Discuss the concept and application of Depth First Search?

Self-Assessment Answer

Concept of DFS

Depth-first search (DFS) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hasn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a stack for exploration.

Application of DFS

Finding connected components.

Topological sorting.

Finding 2-(edge or vertex)-connected components.

Finding 3-(edge or vertex)-connected components.

Finding the bridges of a graph.

Finding strongly connected components.

Planarity testing

Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)

Maze generation may use a randomized depth-first search.

Finding bi-connectivity in graphs.

4.0 Conclusion

Graphs are a commonly used data structure because they can be used to model many real-world problems. A graph consists of a set of nodes with an arbitrary number of connections, or edges, between the nodes. These edges can be either directed or undirected and weighted or unweighted.

5.0 Summary

A graph, like a tree, is a collection of nodes and edges, but has no rules dictating the connection among the nodes. In this module 3 Study Unit 2, we have learnt all about graphs, one of the most versatile data structures.

6.0 Tutor Marked Assignments

What do you understand by a graph and discuss the major operation on it.

Explain Graph traverse.

Write an algorithm to perform a graph traverse using Breadth First Search Technique

Draw Adjacent List Representation of this matrix

	a	b	c	d	e	f	g
a	0	1	0	0	0	0	0
b	1	0	1	1	0	0	0
c	0	1	0	0	1	0	0
d	0	1	0	0	1	1	0
e	0	0	1	1	0	0	1
f	0	0	0	1	0	0	1
g	0	0	0	0	1	1	0

Cite the differences between Depth First Search and Breadth First Search techniques.

7.0 References/ Further Readings

Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). Introduction to Algorithms (2nd ed.). MIT Press and McGraw–Hill. ISBN 0-262-53196-8.

Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. "Introduction to Algorithms." MIT Press. 1990.

Scott Mitchell, author of six books and founder of 4GuysFromRolla.com, has been working with Microsoft Web technologies because January 1998. Scott works as an independent consultant, trainer, and writer, and holds a Masters degree in Computer Science from the University of California – San Diego. He can be reached at mitchell@4guysfromrolla.com, or via his blog at http://ScottOnWriting.NET.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 22.3: Depth-first search, pp. 540–549.

Knuth, Donald E. (1997), *The Art Of Computer Programming Vol 1. 3rd ed*, Boston: Addison-Wesley, ISBN 0-201-89683-4, OCLC 155842391, http://www-cs-faculty.stanford.edu/~knuth/taocp.html

Goodrich, Michael T. (2001), *Algorithm Design: Foundations, Analysis, and Internet Examples*, Wiley, ISBN 0-471-38365-1

Module 4

Software Engineering

Unit 1: Concepts of Software Engineering

Unit 1

Concepts of Software Engineering

Contents

1.0 Introduction

2.0 Learning Outcomes

3.0 Learning Contents

- 3.1 Applications of Software Engineering
- 3.2 Software Project Management
  - 3.2.1 Fundamentals of Software Engineering Project Management
- 3.3 Steps Managing Software Projects
- 3.4 Steps towards More Effective Project Control
- 3.5 Building a medium-sized system in teams, with algorithm efficiency in mind

4.0 Conclusion

5.0 Summary

6.0 Tutor Marked Assignments

7.0 References/ Further Reading

1.0 Introduction

Software Engineering provides the software engineering fundamentals, principles and skills needed to develop and maintain high quality software products. The software engineering processes and techniques covered include requirements specification, design, implementation, testing and management of software projects. Software engineering from a project point of view embarks on software products that are both feasible technically and financially.

Moreover, software projects must be completed on time and within budgets and are important challenges to software engineers. Research has shown that one-third of software projects are never completed, another third do not fulfill their promises. As with any engineering activity, a software engineer starts with *problem definition* and applies tools of the trade to obtain a *problem solution*.

However, unlike any other engineering, software engineering seems to put great emphasis on *methodology* or *method* for obtaining a solution, rather than on tools and techniques. Experts justify this with the peculiar nature of the problems solved by software engineering. These “wicked problems” can be properly defined only after being solved. The role of the software engineering is to capture the customer’s business need and specify the “blueprints” for the system so that programmers can implement it.

2.0 Learning Outcomes

At the end of this unit, you should be able to:

i. explain the need of Software Engineering;

ii. explain the Applications of Software Engineering

iii. merge the application of structured engineering with that of disciplined project management for software development, results in the concept software engineering project management

iv. explain the need for project management;

v. list the Steps for managing software projects; and

vi. explain how to ensure effective project management

3.0 Learning Contents

3.1 Applications of Software Engineering

The days when computer software development could be handled as part of *documentation* or *general* on the agenda of a project meeting (if handled at all) are long gone.

For many engineering and other projects, software has become the pivotal part:

It controls generation and distribution of electricity;

water purification and distribution;

Robotic systems in production plants;

Vehicles, their engines, and traffic flows;

Household equipment;

Aircraft, air traffic, and passenger bookings;

Telecommunications; logistics; spacecraft and space missions, etc., etc.

Software also plays an ever-increasing role in business management:

It controls equipment maintenance management, logistics, resource allocations, business

Processes, financial transactions, accounting, communication, human resources

Self Assessment Exercise

1. Why is the need of Software Engineering in every aspects of life?

Self-Assessment Answer

Software Engineering controls generation and distribution of electricity; It is used for water purification and distribution;

Robotic systems in production plants;

Production of Vehicles, their engines, and traffic flows;

Production of Household equipment;

Running of Aircraft, air traffic, and passenger bookings;

Telecommunications; logistics; spacecraft and space missions, etc., etc.

Software also plays an ever-increasing role in business management:

It controls equipment maintenance management, logistics, resource allocations, business

Processes, financial transactions, accounting, communication, human resources

3.2 Software Project Management

Because of software's growing importance, its development must be managed even more carefully than other areas of large projects. Often, in the past, software was *randomly assembled* – almost in an artistic way. This approach is no longer appropriate; and the departure point for proper software development should be the

realization that software development has grown from an art, to a craft, to a proper engineering discipline. From this departure point it follows that :

Software is a product (although a rather “fluid” one), like any other result of engineering methodologies.

Software development needs the structured application of scientific and engineering principles in order to analyze, design, construct, document and maintain it.

Like any engineering development, large-scale software development also requires the disciplined application of project management principles.

3.2.1 Fundamentals of Software Engineering Project Management

In Managing Software Development, any project “stands” on three legs: cost, schedule and functionality (performance).

These three *project legs* must be balanced, planned in advance, and managed throughout the project’s lifetime in order to have a successful project.

Rigorous application of proper project management techniques on a software development project greatly improves balancing of the three project legs, and the chances of project success.

Proper project management” involves *planning, organizing, staffing, directing* and *control* of the project.

To do all these things successfully, a project manager must have good technical-management-, and people skills.

Over- or under emphasis of any of the project, management or skills elements will certainly result in a failed project.

3.3 Steps for Managing Software Projects

Planning involves:

Developing a strategy to deal with the cost schedules, functionality and maintainability

Organizing involves:

Establishing a structure to be filled by people, aimed at reaching the defined goals and objectives.

Defining job content, interfaces, responsibilities, authority, and resource allocation.

Staffing involves:

Filling the positions in the organizational structure with suitable people.

Keeping the positions filled, in order to execute the plan.

Directing (or Leading) involves:

Creating an environment in which individuals, working together in groups, can accomplish well-selected aims.

Influencing people to contribute to reaching the goals and objectives.

Using leadership styles, communication, conflict resolution, delegation, etc. in order to overcome the problems arising from *people issues* (attitudes, desires, motivations, behavior in groups, etc.) on a project.

Controlling (and co-ordination) involves:

Measuring actual performance.

Comparing actual- with desired results and implementing corrective actions –

e.g. by controlling the actions of the people doing the work.

Self-Assessment Exercise

1. As a software engineer, how can you manage your software project?

Self-Assessment Answer

Planning involves:

Developing a strategy to deal with the cost schedules, functionality and maintainability

Organizing involves:

Establishing a structure to be filled by people, aimed at reaching the defined goals and objectives.

Defining job content, interfaces, responsibilities, authority, and resource allocation.

Staffing involves:

Filling the positions in the organizational structure with suitable people.

Keeping the positions filled, in order to execute the plan.

Directing (or Leading) involves:

Creating an environment in which individuals, working together in groups, can

accomplish well-selected aims.

Influencing people to contribute to reaching the goals and objectives.

Using leadership styles, communication, conflict resolution, delegation, etc. in order to overcome the problems arising from *people issues* (attitudes, desires, motivations, behavior in groups, etc.) on a project.

Controlling (and co-ordination) involves:

Measuring actual performance.

Comparing actual- with desired results and implementing corrective actions –

e.g. by controlling the actions of the people doing the work.

3.4 Steps towards More Effective Project Control

- ✓ Break the overall program into phases and subsystems (WBS).
- ✓ Clearly define objectives, results and deliverables.
- ✓ Define measurable milestones and quantitative checkpoints.
- ✓ Obtain commitment from all team members and management.
- ✓ Ensure that different teams can work together, and that outputs are compatible.
- ✓ Project tracking - ensure proper project control.
- ✓ Ensure measurability of progress parameters.
- ✓ Hold regular reviews of project goals, plans, progress, etc.
- ✓ Ensure interesting work to maintain interest (take personal preferences into account).
- ✓ Communication, communication, communication!!
- ✓ Leadership - making people feel strong and in control.
- ✓ Minimize threats - manage conflict and power struggles, avoid surprises (up ordown) and unrealistic demands, foster mutual trust.
- ✓ Design an appropriate personnel appraisal and reward system.
- ✓ Assure continuous senior management involvement, endorsement and support.
- ✓ Personal drive - project manager must be enthusiastic about the project.

Self Assessment Exercise

1. List 10 steps towards effective project management
2. Building a medium-sized system in teams, with algorithm efficiency in mind.

Self-Assessment Answer

- ✓ Break the overall program into phases and subsystems (WBS).
- ✓ Clearly define objectives, results and deliverables.
- ✓ Define measurable milestones and quantitative checkpoints.
- ✓ Obtain commitment from all team members and management.
- ✓ Ensure that different teams can work together, and that outputs are compatible.
- ✓ Project tracking - ensure proper project control.
- ✓ Ensure measurability of progress parameters.
- ✓ Hold regular reviews of project goals, plans, progress, etc.
- ✓ Communication, communication, communication!!
- ✓ Leadership - making people feel strong and in control.

4.0 Conclusion

Software Engineering provides the software engineering fundamentals, principles and skills needed to develop and maintain high quality software products. The role of software engineering is to capture the customer's business need and specify the "blueprints" for the system so that programmers can implement it. Software Engineering is most useful in every aspects of life system. The skills needed to have a good software project management are very essential so as to achieve the desired goals. All these aforementioned are discussed.

5.0 Summary

At the end of this lesson we have been able to:

See the need of Software Engineering

Merge the application of structured engineering with that of disciplined project management for software development, results in the concept software engineering project management

Understand the Steps for Managing Software Projects

Explain the steps towards more effective project control

6.0 Tutor Marked Assignments

1. Explain the concept of Software Engineering to a lay man.
2. Discuss the Fundamentals of Software Engineering Project Management.

7.0 References/ Further Reading

IEEE Standard Glossary of Software Engineering Terminology," IEEE std 610.12-1990, 1990, quoted at the beginning of Chapter 1: Introduction to the guide "Guide to the Software Engineering Body of Knowledge

"IEEE Standard Glossary of Software Engineering Terminology," IEEE std 610.12-1990, 1990, quoted at the beginning of Chapter 1: Introduction to the guide "Guide to the Software Engineering Body of Knowledge" (February 6, 2004). Retrieved on 2008-02-21.

^ Pecht, Michael (1995). *Product Reliability, Maintainability, and Supportability Handbook*. CRC Press. ISBN 0-8493-9457-0.

^ Pehrson, Ronald J. (January 1996). "Software Development for the Boeing 777". *CrossTalk: The Journal of Defense Software Engineering*., "The 2.5 million lines of newly developed software were approximately six times more than any previous Boeing commercial airplane development program. Including

commercial-off-the-shelf (COTS) and optional software, the total size is more than 4 million lines of code."

Pierre Bourque and Robert Dupuis, ed (2004). *Guide to the Software Engineering Body of Knowledge - 2004 Version*. IEEE Computer Society. pp. 2–1. ISBN 0-7695-2330-7. http://www.swebok.org.

Stellman, Andrew and Greene, Jennifer (2005). *Applied software project management*. O'Reilly Media, Inc. pp. 308. ISBN 0596009488.

Flyvbjerg, Bent, "From Nobel Prize to Project Management: Getting Risks Right." *Project Management Journal*, vol. 37, no. 3, August 2006, pp. 5-15.

Sommerville, Ian [1982] (2007). "1.1.2 What is software engineering?", *Software Engineering*, 8th ed., Harlow, England: Pearson Education, P. 7. ISBN 0-321-31379-8.