KARNATAKA STATE OPEN UNIVERSITY

DATA STRUCTURES

BCA

I SEMESTER

SUBJECT CODE: BCA 04

SUBJECT TITLE: DATA STRUCTURES

STRUCTURE OF THE COURSE CONTENT

BLOCK 1 PROBLEM SOLVING
Unit 1: Top-down Design – Implementation
Unit 2: Verification – Efficiency
Unit 3: Analysis
Unit 4: Sample Algorithm

BLOCK 2 LISTS, STACKS AND QUEUES
Unit 1: Abstract Data Type (ADT)
Unit 2: List ADT
Unit 3: Stack ADT
Unit 4: Queue ADT

BLOCK 3 TREES
Unit 1: Binary Trees
Unit 2: AVL Trees
Unit 3: Tree Traversals and Hashing
Unit 4: Simple implementations of Tree

BLOCK 4 SORTING
Unit 1: Insertion Sort
Unit 2: Shell sort and Heap sort
Unit 3: Merge sort and Quick sort
Unit 4: External Sort

BLOCK 5 GRAPHS
Unit 1: Topological Sort
Unit 2: Path Algorithms
Unit 3: Prim's Algorithm
Unit 4: Undirected Graphs – Bi-connectivity

Books:

1. R. G. Dromey, "How to Solve it by Computer" (Chaps 1-2), Prentice-Hall of India, 2002.
2. M. A. Weiss, "Data Structures and Algorithm Analysis in C", 2nd ed, Pearson Education Asia, 2002.
3. Y. Langsam, M. J. Augenstein and A. M. Tenenbaum, "Data Structures using C", Pearson Education Asia, 2004
4. Richard F. Gilberg, Behrouz A. Forouzan, "Data Structures – A Pseudocode Approach with C", Thomson Brooks / COLE, 1998.
5. Aho, J. E. Hopcroft and J. D. Ullman, "Data Structures and Algorithms", Pearson education Asia, 1983.

BLOCK I

BLOCK I PROBLEM SOLVING

Computer problem-solving can be summed up in one word – it is demanding! It is an intricate process requiring much thought, careful planning, logical precision, persistence, and attention to detail. At the same time it can be challenging, exciting, and satisfying experience with considerable room for personal creativity and expression. It computer problem=-solving is

approached in this spirit then the chances of success are greatly amplified. In the discussion which follows in this introductory chapter we will attempt to lay the foundations for our study of computer problem-solving.

This block consists of the following units:
Unit 1: Top-down Design – Implementation
Unit 2: Verification – Efficiency
Unit 3: Analysis
Unit 4: Sample Algorithm

UNIT
1
TOP-DOWN DESIGN – IMPLEMENTATION

STRUCTURE

1.0 AIMS AND OBJECTIVES

At the end of this lesson, students should be able to demonstrate appropriate skills, and show an understanding of the following:

- Aims and objectives of Top-down design
- Introduction
- Top-down Design
- Implementation
- Advantages of this methodology

1.3 INTRODUCTION

The primary goal in computer problem-solving is an algorithm which is capable of being implemented as a correct and efficient computer program. In our discussion leading up to the consideration of algorithm design we have been mostly concerned with the very broad aspects of problem solving.

We now need consider those aspects of problem-solving. We now need to consider those aspects of problem-solving and algorithm design which are closer to the algorithm implementation. Once we have defined the problem to be solved and we have at least a vague idea of how to solve it, we can begin to bring to bear powerful techniques for designing algorithms. The key to being able to successfully design algorithms lies in being able to manage the inherent complexity of most problems that require computer solution. People as problem-solvers are only able to focus on, and comprehend at one time, a very limited span

of logic or instructions. A technique for algorithm design that tries to accommodate this human limitation is known as top-down design or stepwise refinement.

1.2 TOP-DOWN DESIGN

Top-down design is a strategy that we can apply to take the solution of a computer problem from a vague outline to a precisely defined algorithm and program implementation. Top-down design provides us with a way of handling the inherent logical complexity and detail frequently encountered in computer algorithms. It allows us to build our solutions to a problem in a stepwise fashion. In this way, specific and complex details of the implementation are encountered only at the stage when we have done sufficient ground work on the overall structure and relationships among the various parts of the problem. Solution method where the problem is broken down into smaller sub-problems, which in turn are broken down into smaller sub-problems, continuing until each sub-problem can be solved in few steps. Problem-solving technique in which the problem is divided into sub problems; the process is applied to each sub problem

Modules: Self-contained collection of steps that solve a problem or sub problem
Abstract Step: An algorithmic step containing unspecified details
Concrete Step: An algorithm step in which all details are specified

1.2.1 Example:

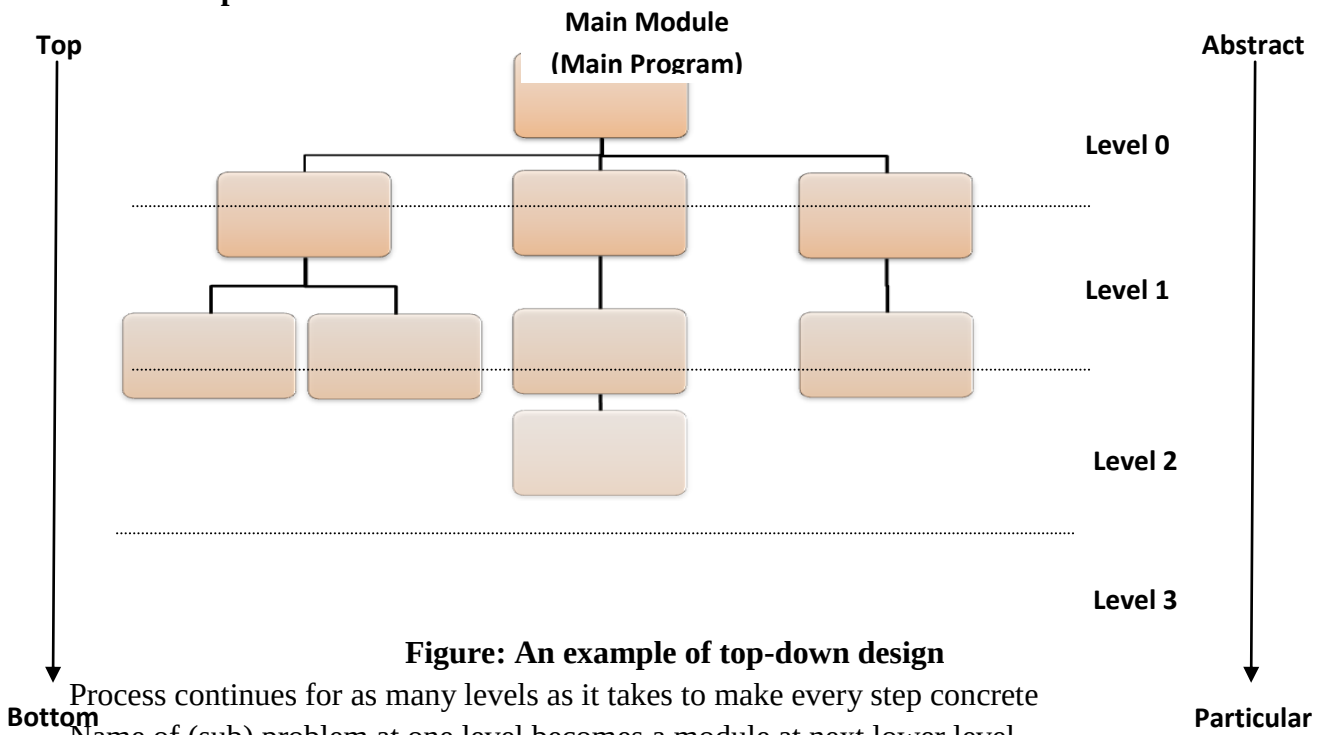


Figure: An example of top-down design

Process continues for as many levels as it takes to make every step concrete
Name of (sub) problem at one level becomes a module at next lower level

1.2.2 Breaking a problem into sub problems

Before we can apply top-down design to a problem we must first do the problem-solving ground work that gives us at least the broadest of outlines of our solution. Sometimes this might demand a lengthy and creative investigation into the problems while at the other times the problem description may in itself provide the necessary starting point for top-down design. The general outline may consist of single statement or a set of statements. Top-down design suggests that we take the general statements that we have about the solution, one at a time, and break them down into a set of more precisely defined subtasks. These subtasks should more accurately describe how the final goal is to be reached. With each splitting of a task into subtask it is essential that the way in which the subtasks need to interact with each other be precisely defined. Only in this way is it possible to preserve the overall structure of the solution to the problem. Preservation of the overall structure into the solution to a problem is important both for making the algorithm comprehensible and also for making it possible to prove the correctness of the solution. The process of repeatedly breaking a task down into subtasks and then each subtasks into still smaller subtasks must continue until the eventually end up with subtasks that can be implemented as program statements. For most algorithms we only need to go down to two or three levels although obviously for large software projects this is not true. The larger and more complex the problem, the more it will need to be broken down to be made tractable. The process of breaking down the solution to a problem into subtasks in the manner described results in an implement able set of subtasks that fit quiet naturally into block-structured languages like Pascal and Algol. There can therefore be a smooth and natural interface between the stepwise refined algorithm and the final program implementation—a highly desirable situation for keeping the implementation task as simple as possible.

A General Example

Planning a large party

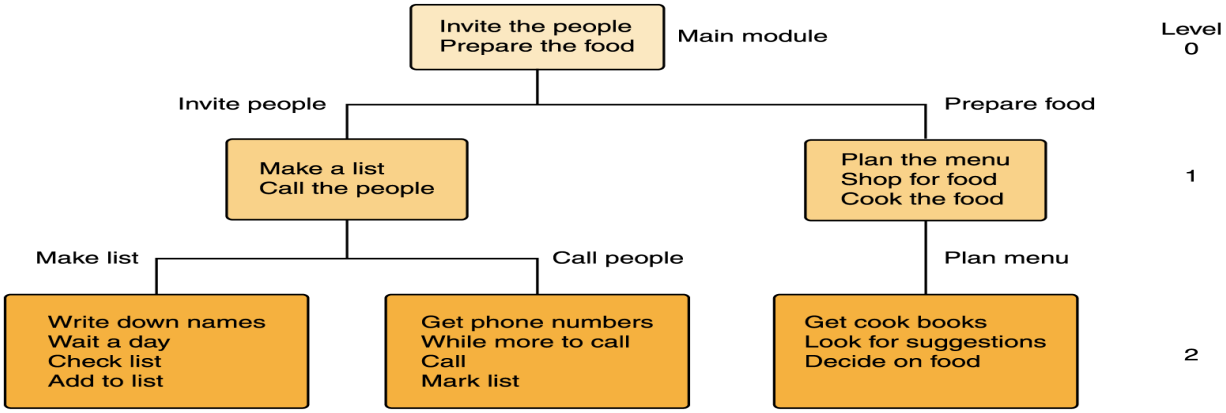


Figure: Sub dividing the party planning

Problem

- Create an address list that includes each person's name, address, telephone number, and e-mail address
- This list should then be printed in alphabetical order
- The names to be included in the list are on scraps of paper and business cards

1.2.3 Choice of suitable data structure:

One of the most important decisions we have to make in formulating computer solutions to problems is the choice of appropriate data structures.

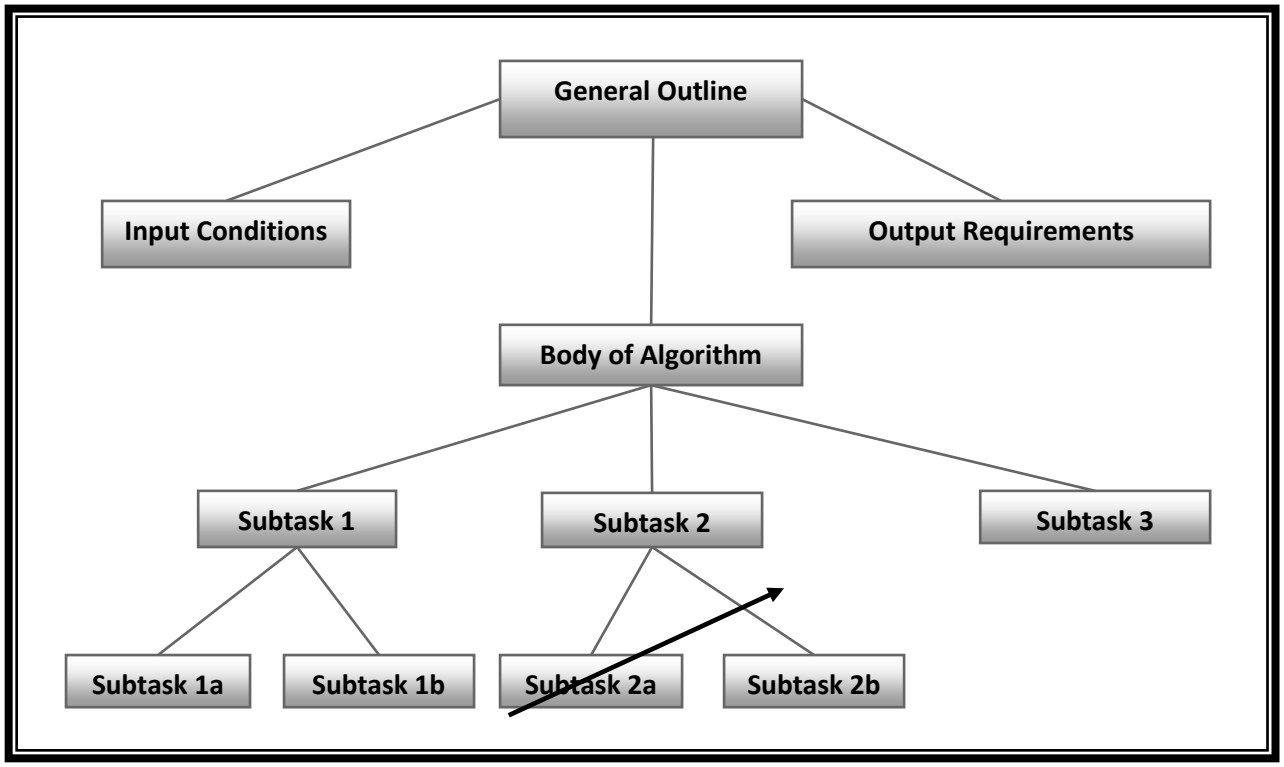


Figure: Schematic breakdown of a problem into subtasks as employed in top-down design.

All programs operate on data and consequently the way the data is organized can have a profound effect on every aspect of the final solution. In particular, an inappropriate choice of data structure often leads to clumsy, inefficient, and difficult implementations. On the other hand, an appropriate choice usually leads to a simple, transparent, and efficient implementation.

1.2.4 Construction of loops

In moving from general statements about the implementation towards sub-tasks that can be realized as computations, almost invariable we are led to a series of iterative constructs, or loops, and structures that are conditionally executed.
These structures, together with input/output statements, computable expressions, and assignments, make up the heart of the program implementations.

1.2.5 Establishing initial conditions for loops:

To establish the initial conditions for a loop, a usually effective strategy is to set the loop variables to the values that would have to assume in order to solve the smallest problem associated with the loop.

i:= 0
s:=0 } Solution for n=0

1.2.6 Finding the iterative construct

Once we have the conditions for solving the smallest problem the next step is to try to extend it to the next smallest problem (in this case when i=1). That is we want to build on the solution to the problem for i=0 to get the solution for i=1
The solution for n=1 is:

i:= 0
s:=a[1] Solution for n=1

1.2.7 Termination of loops

There are a number of ways in which loops can be terminated. In general the termination conditions are dictated by the nature of the problem. The simplest conditions for terminating a loop occur when it is known in advance how much iteration need to be made. In these circumstances we can use directly the termination facilities built into programming languages. For example, in Pascal the **for-** loop can be used for such computations:

```
For i:=1to n do
Begin
.
.
end
```

this loop terminates unconditionally after *n* iterations.

1.3 THE ADVANTAGES OF ADOPTING A TOP-DOWN DESIGN METHODOLOGY

- Breaking the problem into parts helps us to clarify what needs to be done.
- At each step of refinement, the new parts become less complicated and, therefore, easier to figure out.
- Parts of the solution may turn out to be reusable.
- Breaking the problem into parts allows more than one person to work on the solution.
- The possibility to perform system architectural exploration and a better overall system optimization (e.g. finding an architecture that consumes less power) at a high level before starting detailed circuit implementations.
- The elimination of problems that often cause overall design iterations, like the anticipation of problems related to interfacing different blocks.

- The possibility to do early test development in parallel to the actual block design, etc.
- The ultimate advantage of top-down design therefore is to catch problems early in the design flow and as a result have a higher chance of first-time success with fewer or no overall design iterations, hence shortening design time, while at the same time obtaining a better overall system design.
- The methodology however does not come for free and requires some investment from the design team, especially in terms of high-level modeling and setting up a sufficient model library for the targeted application domain. Even then there remains the risk that also at higher levels in the design hierarchy low-level details (e.g. matching limitations, circuit no idealities, layout effects) may be important to determine the feasibility or optimality of an analog solution. The high-level models used therefore must include such effects to the extent possible, but it remains difficult in practice to anticipate or model everything accurately at higher levels. Besides the models, efficient simulation methods are also needed at the architectural level in order to allow efficient interactive explorations.
- An Example of Top-Down Design:
    - **Problem:**
        - We own a home improvement company.
        - We do painting, roofing, and basement waterproofing.
        - A section of town has recently flooded (zip code 21222).
        - We want to send out pamphlets to our customers in that area.

1.4 IMPLEMENTATION OF ALGORITMS

The implementation of an algorithm that has been properly designed in a top-down fashion should be an almost mechanical process. There are, however, a number of points that should be remembered.If an algorithm has been properly designed the path of execution should flow in a straight line from top to bottom. It is important that the program implementation adheres to this top-to-bottom rule. Programs (and subprograms) implemented in this way are usually much easier to understand and debug. They are also usually much easier to modify should the need arise because the relationships between the various parts of the program are much more apparent.

Implementation phase:

1. **Write code** -- translate the algorithm into a programming language.
2. **Test** -- have the computer follow the steps (execute the program).
3. **Debug** -- check the results and make corrections until the answers are correct.
4. **Use the program**.

Use of Procedures to emphasize modularity

To assist with both the development of the implementation and the readability of the main program it is usually helpful to modularize the program along the lines that follow naturally

from the top-down design. This practice allows as implementing the set of independent procedures to perform specific and well defined tasks.

For example, if as part of an algorithm it is required to sort an array, then a specific independent procedure should be used for the sort. In applying modularization in an implementation one thing to watch is that the process is not taken too far, to a point at which the implementation again becomes difficult to read because of the fragmentation.

When it is necessary to implement somewhat larger software projects a good strategy is to first complete the overall design in a top-down fashion. The mechanism for the main program can then be implemented with calls to the various procedures that will be needed in the final implementation. In the first phase of the implementation, before we have implemented any of the procedures, we can just place a right statement in skeleton procedures which simply writes out the procedure's name when it is called; for example,

```
procedure sort;
begin
writeln ('sort called')
end
```

This practice allows us to test the mechanism of the main program at an early stage and implement and test the procedures one by one. When a new procedure has been implemented the simply substitute it for its "dummy" procedure.

1.5 LET US SUM UP

The primary goal in computer problem-solving is an algorithm which is capable of being implemented as a correct and efficient computer program. In our discussion leading up to the consideration of algorithm design we have been mostly concerned with the very broad aspects of problem solving. We now need consider those aspects of problem-solving. We now need to consider those aspects of problem-solving and algorithm design which are closer to the algorithm implementation.

Top-down design is a strategy that we can apply to take the solution of a computer problem from a vague outline to a precisely defined algorithm and program implementation. Top-down design provides us with a way of handling the inherent logical complexity and detail frequently encountered in computer algorithms. It allows us to build our solutions to a problem in a stepwise fashion. In this way, specific and complex details of the implementation are encountered only at the stage when we have done sufficient ground work on the overall structure and relationships among the various parts of the problem.

A solution method where the problem is broken down into smaller sub-problems, which in turn are broken down into smaller sub-problems, continuing until each sub-problem can be solved in few steps. Problem-solving technique in which the problem is divided into sub problems; the process is applied to each sub problem

1.6 KEYWORDS

1.7 QUESTIONS FOR DISCUSSION

a. Explain the advantages of top down methodology
b. Discuss about Implementation of Algorithms

1.8 SUGGESTION FOR BOOKS READING

UNIT 2

VERIFICATION – EFFICIENCY

STRUCTURE

2.0 AIMS AND OBJECTIVES

At the end of this lesson, students should be able to demonstrate appropriate skills, and show an understanding of the following:

- Introduction to program verification
- Efficiency of algorithm
- Measuring Efficiency of algorithm

2.1 INTRODUCTION

The cost of development of computing software has become a major expense in the application of computers. Experience in working with computer systems has led to the observation that generally more than half of all programming effort and resources is spent in correcting errors in programs and carrying out modification of programs.

As larger and more complex programs are developed, both these tasks become harder and more time consuming. In some specialized military, space and medical applications, program correctness can be a matter of life and death. This suggests two things. Firstly, that

considerable savings in the time for program modification should be possible if more care is put into the creation of clearly written code at the time of program development. We have already seen that top-down design can serve as a very useful aid in the writing of programs that are readable and able to be understood at both superficial and detailed levels of implementation. The other problem of being able to develop correct as well as clear code also requires the application of a systematic procedure. Proving even simple programs correct turns out to be far from easy task. It is not simply a matter of testing a program's behavior under the influence of a variety of input conditions to prove its correctness. This approach to demonstrating a program's correctness may, in some cases, show up errors but it cannot guarantee their absence. It is this weakness in such an approach that makes it necessary to resort to a method of program verification that is based on sound mathematical principles.

2.2 PROGRAM VERIFICATION

Program verification refers to the application of mathematical proof techniques to establish that the results obtained by the execution of a program with arbitrary inputs are in accord with formally defined output specification.
The following are the steps for program verification:

- Computer model for program execution
- Input and output assertions
- Implications and symbolic execution
- Verification of straight-line program segments
- Verification of program segments with branches
- Verification of program segments with Loops
- Verification of program segments that employ arrays
- Proof of termination

2.3 THE EFFICIENCY OF ALGORITHMS

Efficiency consideration for algorithms is inherently tied in with the design, implementation, and analysis of algorithms. Every algorithm must use up some of a computer's resources to complete its task. The resources most relevant in relation to efficiency are central processor time (CPU time) and internal memory. The efficiency of algorithm can be measured in the following ways:

2.3.1 An Obvious Way to Measure Efficiency:

- The Process:
  - Write the necessary code
  - Measure the running time and space requirements

- An Important Caveat:
  - It would be a big mistake to run the application only once

- Instead, one must develop a sampling plan (i.e., a way of developing/selecting sample inputs) and test the application on a large number of inputs

2.3.2 Problems with This Approach:

- It is often very difficult to test algorithms on a large number of representative inputs.
- The efficiency can't be known *a priori* and it if often prohibitively expensive to implement and test an algorithm only to learn that it is not efficient.

2.3.3 An Alternative Approach: Bounds:

- The Idea:
  - Obtain an upper and/or lower *bound* on the time and space requirements
- An Observation:
  - Bounds are used in a variety of contexts
- A Common Question at Job Interviews:
  - How many hairs do you have on your head?
- Obtaining a Bound:
  - Suppose a human hair occupies at least a square of 100 microns on a side (where one micron is one millionth of a meter)
  - Then, each hair occupies at least 0.00000001 m^2 in area
  - Suppose further that your head has at most 400 cm^2 in total area (or 0.04 m^2)
  - Then, your head has at most 4 million hairs
- Worst Case Analysis:
  - Instead of trying to determine the "actual" performance experimentally, one instead attempts to find theoretical upper bounds on performance
- Some Notation:
  - The worst case running time for an input of size n is denoted by T (n)
  - The worst case space requirement for an input of size n is denoted by S (n)

2.4 COMPARING DIFFERENT ALGORITHMS

- Which is better?
  - An algorithm with worst case time of n 2 +5
  - An algorithm with worst case time of n 2 +2

- Do You Care When n = 1000?
  - 1,000,005
  - 1,000,002

2.4.1 Asymptotic Dominance

- The Idea:
  - When comparing algorithms based on their worst case time or space it seems best to consider their *asymptotic* performance (i.e., their performance on "large" problems)
- Applying this Idea:
  - Consider the limit of the worst case time or space

I Hate Math - Does Efficiency Really Matter?

- Compare Two Algorithms When:
  - One requires n 3 iterations and one requires 2 n iterations
  - Each iteration requires one microsecond (i.e., one millionth of a second)
- Different Problem Sizes:

T (n)	**n = 10**	**n = 25**	**n = 50**	**n = 75**
n 3	0.001 sec	0.016 sec	0.125 sec	0.422 sec
2 n	0.001 sec	33.554 sec	35.702 yrs	11,979,620.707 centuries

2.4.2 "big O" Notation:

- Interpreting "little o":
  - When T is "little o" of f it means that T is of lower order of magnitude than f
- Defining "big O":
  - T = O [f (n)] iff there exist positive k and n 0 such that T (n) ≤ k f (n) for all n ≥ n 0
- Interpreting "big O":
  - T is not of higher order of magnitude than f
- Show:
  - n 2 +5 is O (n 2)
- Choose k = 3 and n 0 = 2 and observe:
  - n 2 +5 ≤ 3 n 2 for all n ≥ 2

2.4.3 Lower Bounds on Growth Rates

- Defining "Big Omega":
  - T = Ω [g (n)] iff there exists a positive constant c such that T (n) ≥ c g (n) for an infinite number of values of n
- An Example: n 2 +5
  - Is Ω (n 2) since, setting c = 1 :
  - n 2 +5 ≥ n 2 for all n ≥ 0

2.4.4 Categorizing Performance:

Asymptotic Bound	Name
O (1)	Constant
O (log n)	Logarithmic
O (n)	Linear
O (n 2)	Quadratic
O (n 3)	Cubic
O (a n)	Exponential
O (n !)	Factorial

2.4.5 Determining Asymptotic Bounds

1. If T 1 = O [f 1 (m)] then k T 1 = O [f 1 (m)] for any constant k
2. If T 1 = O [f 1 (m)] and T 2 = O [f 2 (m)] then T 1 T 2 = O [f 1 (m)] O [f 2(m)]
3. If T 1 = O [f 1 (m)] and T 2 = O [f 2 (m)] then T 1 + T 2 = max { O [f 1 (m)] , O [f 2 (m)] }

- Suppose an algorithm has three "steps" with running times of O (n 4) , O (n 2) , and O (log n) .
- Then, it follows from rule 3 that the running time for the entire algorithm is O (n 4) .
- Suppose an algorithm has one "step" with a running time of O (n) and that it repeats this "step" (in a loop) 1000 times.
- Then, it follows from rule 1 that the running time for the entire algorithm is O (n) .

2.5 AN EXAMPLE: FACTORIALS:

What is the asymptotic bound on the worst case time efficiency of the following recursive algorithm?

```
int factorial(int n)
{
// Check for valid input
if (n > 12) return -1;

if ((n == 0) || (n == 1)) return 1;

return n*factorial(n-1);
}
```

What is the asymptotic bound on the worst case time efficiency of the following iterative algorithm?

```
int factorial(int n)
{
int i, value;

// Check for valid input
if (n > 12) return -1;

value = 1;
for (i=2; i<=n; i++) {

value *= i;
}
return value;
}
```

What is the asymptotic bound on the worst case time efficiency of the following algorithm? What about the space efficiency?

```
int factorial(int n)
{
int f[13]={1,1,2,6,24,120,720,5040,40320,
362880, 3628800, 39916800, 479001600};

// Check for valid input
if (n > 12) return -1;

return f[n];
}
```

Efficiency considerations for algorithms are inherently tied in with the design, implementation, and analysis of algorithms. Every algorithm must use up of a computer's resources to complete its task. The resources most relevant in relation to efficiency are central processor time (CPU time) and internal memory. Because of the high cost of computing resources it is always desirable to design algorithms that are economical in the use of CPU time and memory.

This is an easy statement to make but one that is often difficult to follow either because of bad design habits, or the inherent complexity of the problem, or both. As with most other aspects of algorithm design, there is no recipe for designing efficient algorithms. Despite

there being some generalities each problem has its own characteristics which demand specific responses to solve the problem efficiently. Within the framework of this last statement we will try to make a few suggestions that can sometimes be useful in designing efficient algorithms.

2.6 MEASURING THE EFFICIENCY OF ALGORITHMS

The topic of algorithms is a topic that is central to computer science. Measuring an algorithm's efficiency is important because your choice of an algorithm for a given application often has a great impact. Word processors, ATMs, video games and life support systems all depend on efficient algorithms. Consider two searching algorithms. What does it mean to compare the algorithms and conclude that one is better than the other?

The analysis of algorithms is the area of computer science that provides tools for contrasting the efficiency of different methods of solution. Notice the use of the term methods of solution rather than programs; it is important to emphasize that the analysis concerns itself primarily with significant differences in efficiency – differences that you can usually obtain only through superior methods of solution and rarely through clever tricks in coding. Although the efficient use of both time and space is important, inexpensive memory has reduced the significance of space efficiency. Thus, we will focus primarily on time efficiency. How do you compare the time efficiency of two algorithms that solve the same problem? One possible approach is to implement the two algorithms in C++ and run the programs. There are three difficulties with this approach:

- How are the algorithms coded? Does one algorithm run faster than another because of better programming? We should not compare implementations rather than the algorithms. Implementations are sensitive to factors such as programming style that cloud the issue.

- What computer should you use? The only fair way would be to use the same computer for both programs. But even then, the particular operations that one algorithm uses may be faster or slower than the other – and may be just the reverse on a different computer. In short, we should compare the efficiency of the algorithms independent of a particular computer.

- What data should the programs use? There is always a danger that we will select instances of the problem for which one of the algorithms runs uncharacteristically fast. For example, when comparing a sequential search and a binary search of a sorted array. If the test case happens to be that we are searching for an item that happens to be the smallest in the array, the sequential search will find the item more quickly than the binary search.

To overcome these difficulties, computer scientists employ mathematical techniques that analyze algorithms independently of specific implementations, computers or data. You begin this analysis by counting the number of significant operations in a particular solution.

As an example of calculating the time it takes to execute a piece of code, consider the nested for loops below:

```
for (i = 1; i <= N; ++i)
        for (j = 1; j <= i; ++j)
for (k = 0; k < 5; ++k)
Task T;
```

If task T requires t time units, the innermost loop on K requires 5*t time units. We will discuss how to calculate the total time, which is: 5*t*N*(N+1)/2 time units.

This example derives an algorithm's time requirements as a function of problem size. The way to measure a problem's size depends on the application. The searches we have discussed depend on the size of the array we are searching. The most important thing to learn is how quickly the algorithm's time requirement grows as a function of the problem size.

A statement such as:
Algorithm A requires time proportional to f (N)
enables you to compare algorithm A with another algorithm B which requires
g(N) time units.

2.6.1 Definition of the Order of an Algorithm:

Algorithm A is order f(N) – denoted O(f(N)) – if constants c and N0 exist such that A requires no more than c*f(N) time units to solve a problem of size N >= N0. That is, g(N) is O(f(N)) if the constants c and N0 exist such that g(N) < c*f(N) for N >= N0. If g(N) is the time required to run Algorithm A, the A is O(F(N)).

- **Worst case efficiency:** is the maximum number of steps that an algorithm can take for any collection of data values.

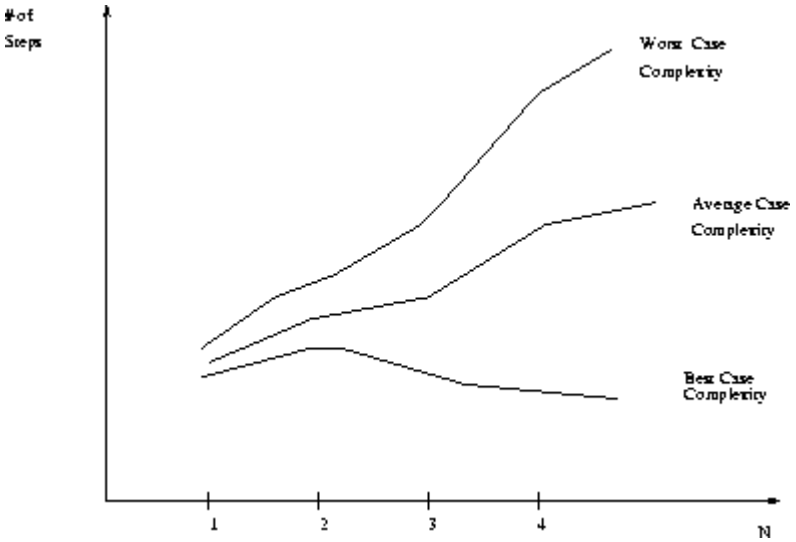
- **Best case efficiency:** is the minimum number of steps that an algorithm can take any collection of data values.

- **Average case efficiency:**

  - The efficiency averaged on all possible inputs
  - must assume a distribution of the input
  - We normally assume uniform distribution (all keys are equally probable)

2.6.2 The Diagram explains the complexity of algorithm:

The *worst case complexity* of the algorithm is the function defined by the maximum number of steps taken on any instance of size *n*.



The best case complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size n.

The average-case complexity of the algorithm is the function defined by an average number of steps taken on any instance of size n.

Each of these complexities defines a numerical function - time vs. size!

Worst-case analysis: A particular algorithm might require different times to solve different problems of the same size. For example, the time an algorithm requires to search N items might depend on the nature of the items. Usually you consider the maximum amount of time that an algorithm can require to solve a problem of size N – that is, the worst case. Although worst-case analysis can produce a pessimistic time estimate, such an estimate does not mean that your algorithm will always be slow. Instead, you have shown that the algorithm will never be slower than your estimate. An algorithm's worst case might happen rarely, if at all, in practice.

Tightness: We want the "tightest" big-O upper bound we can prove. If f(N) is O(N2), we want to say so even though the statement f(N) is O(N3) is technically true but "weaker".

Simplicity: We will generally regard f (N) as "simple" if it is a single term and the coefficient of that term is 1. N2 is simple; 2N2 and N2+ N are not. We want simple and tight bounds to describe the order of our algorithms.

2.7 LET US SUM UP

Program verification refers to the application of mathematical proof techniques to establish that the results obtained by the execution of a program with arbitrary inputs are in accord with formally defined output specification.

Efficiency considerations for algorithms are inherently tied in with the design, implementation, and analysis of algorithms. Every algorithm must use up of a computer's resources to complete its task. The resources most relevant in relation to efficiency are central processor time (CPU time) and internal memory. Because of the high cost of computing resources it is always desirable to design algorithms that are economical in the use of CPU time and memory.

2.8 KEYWORDS

2.9 QUESTIONS FOR DISCUSSION

1. Describe the efficiency of algorithm
2. What are the steps for program verification?

2.10 SUGGESTION FOR BOOKS READING

UNIT 3 ANALYSIS

STRUCTURE

3.0 AIMS AND OBJECTIVES

At the end of this lesson, students should be able to demonstrate appropriate skills, and show an understanding of the following:

- Introduction to the analysis of algorithm
- Definition
- Problem Analysis

3.1INTRODUCTION TO THE ANALYSIS OF ALGORITHMS

There are usually many ways to solve any given problem. In computing, as in most efforts of human endeavor, we are generally concerned with "good" solutions to problems. This raises certain questions as to just what do we mean by a "good" solution to a problem? In algorithm design "good" has both qualitative and quantitative aspects. There are often certain esthetic and personal aspects to this but, on a more practical level; we are usually interested in a solution that is economical in the use of computing and human resources.

Among other things, good algorithms usually possess the following qualities and capabilities:

1. They are simple but powerful and general solutions.
2. They can be easily understood by others, that is, the implementation is clear and concise without being "tricky".
3. They can be easily modified if necessary.

4. They are correct for clearly defined solutions.
5. They are able to be understood on a number of levels.
6. They are economical in the use of computer time, computer storage and peripherals.
7. They are documented well enough to be used by others who do not have detailed knowledge of their inner workings.
8. They are not dependent on being run on a particular computer.
9. They are able to be used as a sub procedure for other problems.
10. The solution is pleasing and satisfying to its designer—a product that the designer feels proud to have created.

These qualitative aspects of a good algorithm are very important but it is also necessary to try to provide some quantitative measures to complete the evaluation of "goodness" of an algorithm. Quantitative measures are valuable in that they can give us a way of directly predicting the performance of two or more algorithms that are intended to solve the same problem. This can be important because the use of an algorithm that is more efficient means a saving in computing resources which translates into a saving in time and money.

3.2 DEFINITION

"People who analyze algorithms have double happiness. First of all they experience the sheer beauty of elegant mathematical patterns that surround elegant computational procedures. Then they receive a practical payoff when their theories make it possible to get other jobs done more quickly and more economically.... The appearance of this long-awaited book is therefore most welcome. Its authors are not only worldwide leaders of the field; they also are masters of exposition." **--D. E. Knuth**

3.3 Problem analysis

Despite the large interest in the mathematical analysis of algorithms, basic information on methods and models in widespread use has not been directly accessible for work or study in the field. The authors here address this need, combining a body of material that gives the reader both an appreciation for the challenges of the field and the requisite background for keeping abreast of the new research being done to meet these challenges.

For any given problem, it is quite possible that there is more than one algorithm that represents a correct solution. A good example of this is the problem of sorting. Dozens of different algorithms have been written to solve this problem. Given such a wide range of solutions, how can we determine which algorithm is the best one to use? To do this, we must analyze our algorithms is such a way that we can gauge the efficiency of the algorithm. Once we have calculated the efficiency of an algorithm, we can compare our measurements and select the best solution.

Let's analyze the three sorting algorithms in this section and determine which one was the most efficient solution for sorting the list of seven numbers in our examples. To do this we need a way to measure the efficiency of our algorithms. We can actually measure the efficiency in two different ways: space efficiency and time efficiency. An algorithm that is space-efficient uses the least amount of computer memory to solve the problem of sorting. An algorithm that is time-efficient uses the least amount of time to solve the problem of sorting. Since most of the sorting operations are comparisons, copies, and swaps, we can count these operations and use our results as a measure of time efficiency. The table below summarizes our measures of time and space efficiency in sorting.

Type of Efficiency	**Measures**
Space	Amount of computer memory
Time	# of items copied # of items swapped # of items compared

3.3.1 Space Efficiency

Let's begin our analysis by determining which sort was the most space-efficient. We discussed in our previous lesson that space efficiency can be measured by calculating the number of memory cells a particular sort requires. For simplicity, we will assume that a memory cell can hold one number. This means that a sort with five numbers would require at least five memory cells just to store the numbers in the computer. The total number of memory cells needed for sorting will depend on how many additional cells the algorithm requires to order the numbers.

Type of Efficiency	**Measure**
Space	Amount of computer memory

For each sort, calculate the number of memory cells that the algorithm required, and then enter your answers in the boxes below to see if they are correct. You can return to the pages on sorting by clicking the name of the sort.

3.3.2 Time Efficiency

Now let's determine which sort was the most time-efficient. To do this, we will count the number of operations each sort performed. Most of the operations that are done by the computer during sorting fall into two groups: copying numbers or comparing numbers. The algorithm which requires the least copying and comparing is the one that will execute the fastest.

Type of Efficiency	Measures
Time	# of items copied # of items swapped # of items compared

For the Insertion Sort and the Selection Sort, it will be easier to count the number of swaps that are done rather than the number of copies. Remember that the swap operation requires three copies. We can find the total number of copies that the algorithms perform by counting the number of swaps and multiplying by three. The Simple Sort does not use the swap operation, so you can count the number of copies directly.

1.3.3.3 Comparison of Sorts

Now that you have completed your calculations, let's summarize the results. We already know that the Insertion Sort and the Selection Sort were the most space-efficient, but we have yet to determine which sort is the most time-efficient. We will see that this answer is a little more difficult to determine.

Space Efficiency

Algorithm	# of memory cells
Simple Sort	14
Insertion Sort	8
Selection Sort	8

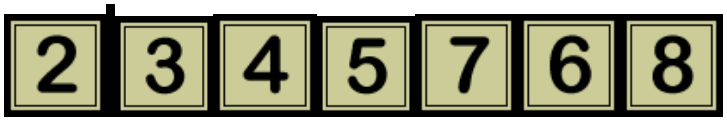
Time Efficiency

Algorithm	# of copies	# of comparisons
Simple Sort	7	42
Insertion Sort	45	19
Selection Sort	15	21

Notice that the Simple Sort required the least amount of copies. We would expect this to be true since it does not swap numbers while sorting. Instead the numbers are copied to a new list in the computer. This is a common tradeoff between time and space. Although the Simple Sort loses space efficiency by using two lists, it gains time efficiency because less copy is required. Of course this does not mean that it is always best to use the Simple Sort to gain more speed. If we are trying to sort a list of 5 million names the Simple Sort would use too much space in the computer's memory. It would be much better to swap items within the list rather than create two lists.

For number of comparisons, the Selection Sort and Insertion Sort were nearly the same. The Simple Sort, however, required twice as many comparisons. We can see the reason for this difference by thinking about how the algorithms work. Each algorithm repeatedly searches for the smallest number and then places this number in the correct position. For the Insertion Sort and the Selection Sort, each iteration of this process reduces the unsorted section by one number. During the next search, the computer does not need to make as many comparisons to find the smallest number. The Simple Sort, however, replaces sorted numbers with a marker called MAX. Each time the computer searches for the smallest number, it must compare all seven memory cells. This approach is much less efficient.

Given the particular set of seven numbers we sorted, the Selection Sort was the most time-efficient. However, it is important to understand that this may not be true for every set of seven numbers. Consider the following example.



If we use the Insertion Sort on these numbers only 8 comparisons and 1 swap would be needed to sort them. However, if we use the Selection Sort, 21 comparisons and 1 swap would be needed. In this case, the Insertion sort is more efficient.

3.4 LETS SUM UP

There are usually many ways to solve any given problem. In computing, as in most efforts of human endeavor, we are generally concerned with "good" solutions to problems. This raises certain questions as to just what do we mean by a "good" solution to a problem? In algorithm design "good" has both qualitative and quantitative aspects. There are often certain esthetic and personal aspects to this but, on a more practical level; we are usually interested in a solution that is economical in the use of computing and human resources.

For any given problem, it is quite possible that there is more than one algorithm that represents a correct solution. A good example of this is the problem of sorting. Dozens of different algorithms have been written to solve this problem. Given such a wide range of solutions, how can we determine which algorithm is the best one to use? To do this, we must analyze our algorithms is such a way that we can gauge the efficiency of the algorithm. Once we have calculated the efficiency of an algorithm, we can compare our measurements and select the best solution.

3.5 KEYWORDS

3.6 Questions for Discussion

1. Define program analysis?
2. Write the steps for the program analysis?

3.7 SUGGESTION FOR BOOKS READING

UNIT
4
SAMPLE ALGORITHMS

STRUCTURE

4.0 Aims and Objectives

At the end of this lesson, students should be able to demonstrate appropriate skills, and show an understanding of the following:

- Introduction to Problem Summary
- Algorithm and Sample code
- Programs

4.1 INTRODUCTION TO PROBLEM SUMMARY

Design an interactive program to read the height and radius of the base of a cone. The program should compute the surface area and volume of the cone. As output it should display the input values and the computed results. Assume that the formulas for computing surface area and volume have been provided elsewhere. Required Constant: PI = 3.14159
Required Variables (all double):
height, radius
area, circum, volume, slant_ht, surf_area
ALGORITHM:
//prompt for and read height of cone
//prompt for and read radius of base of cone

area = PI * radius * radius //compute area of base of cone
circum = 2.0 * PI * radius //compute circumference of base of cone
volume = 1.0/3.0 * area * height //compute volume of cone
slant_ht = sqrt(radius*radius + height*height) //compute slant height
surf_area = area + 1./2.* circum * slant_ht //compute surface area of cone
print radius, height, volume, and surf_area with appropriate labels

4.2 SAMPLE ALGORITHM:

Simple Algorithms:

Goal: To explore various algorithms for solving problems using the built-in simple types in C++

a. Use only the four simple types: String, Integer, Double, Boolean
b. Do not use any functions in java.math;
c. Verify your work with a menu-driven program ("simple.java") from your teacher.

Algorithm and Sample C Code to Check if a Number is a Palindrome – Without Reversing the Digits:

How do we find if a Number is a Palindrome?

Note: No conversion to string. And also, no traditional method of dividing the number to construct the reversed number and then compare.
Here is how you go about it.

- Let's say the number is 1987891 (odd number of digits)
- Reduce it to 198891. If the number were 19877891 (even number of digits) don't change it (basically change it to a number with even number of digits, by excluding the middle digit)
- Now Split it into 198 and 891
- If ((891-198)%9 == 0 and (891*1000+198)%11 == 0 and (198*1000+891)%11 == 0) then its a palindrome

4.3 SAMPLE PROGRAM

Sample C Code

```
#include <stdio.h>
int numdigits(int n) {
int count = 0;
while(n != 0)  {
n /= 10;
++count;
}
return count;
```

```
}
int main(void) {
int n;
int l;
int a,x;
int n1, n2, n3;
scanf("%d",&n);
l=numdigits(n);     /* Find the number of digits */
a=1;
for(x=0;x<l/2;x++)
a*=10;
n1=n%a;
n2=n/a;
if(l%2)
n2/=10;
if (((n1-n2)%9 == 0) && ((n2*a+n1)%11 == 0) && ((n1*a+n2)%11 == 0)) {
printf("%d\t",n);
return 0;
}
return 1;
}
```

4.4 LET US SUM UP

The program should compute the surface area and volume of the cone. As output it should display the input values and the computed results. Assume that the formulas for computing surface area and volume have been provided elsewhere.

To explore various algorithms for solving problems using the built-in simple types. Use only the four simple types: String, Integer, Double, Boolean. Do not use any functions in java. math. Verify your work with a menu-driven program ("simple.java") from your teacher.

4.5 KEYWORDS

1.4.5 Questions for Discussion

1. Write a function factors (Integer num) that will print all the factors of a given positive integer. For example, factors (30) should produce the following formatted output (note that a period terminates the list): The factors of 30 are 1, 2, 3, 5, 6, 10, 15, 30.

2. Write a function GCD (Integer a, Integer b) that returns the greatest common divisor of its two positive integer parameters.

4.6 SUGGESTION FOR BOOKS READING

BLOCK II

BLOCK II LISTS, STACKS AND QUEUES

In this chapter we are going to discuss three of the most simple and data structures. Virtually every significant program will use at least one of these structures explicitly, and a stack is always implicitly used in a program, whether or not you declare one. Among the highlights to this chapter, we will

- Introduce the concept of Abstract Data Types (ADTS).
- Show how to efficiently perform operations on lists.
- Introduce the stack ADT and its use in implementing recursion.
- Introduce the queue ADT and its use in operating systems and algorithm design.

This block consists of four units:

Unit 1: Abstract Data Type (ADT)
Unit 2: List ADT
Unit 3: Stack ADT
Unit 4: Queue ADT

UNIT
1
ABSTRACT DATA TYPE (ADT)

STRUCTURE

1.0 AIMS AND OBJECTIVES

At the end of this lesson, students should be able to demonstrate appropriate skills, and show an understanding of the following:

- Aims and objectives of Abstract Data Types (ADTs)
- Introduction to Abstract Data Types

1.1 Introduction of Abstract Data Type (ADT)

One of the basis rules concerning programming is that no routine should ever exceed a page. This is accomplished by breaking the program down into modules. Each module is a logical unit and does a specific job. Its size is kept small by calling other modules. Modularity has several advantages. First it is much easier to debug small routines than large routines. Second, it is easier for several people to work on a modular program simultaneously. Third, a well-written modular program places certain dependencies in only one routine, making changes easier.

1.2 DEFINITION

Abstract data types or **ADTs** are a mathematical specification of a set of data and the set of operations that can be performed on the data. They are abstract in the sense that the focus is on the definitions of the constructor that returns an abstract handle that represents the data, and the various operations with their arguments. The actual implementation is not defined, and does not affect the use of the ADT. For example, rational numbers (numbers that can be written in the form a/b where a and b are integers) cannot be represented natively in a computer.

1.3 MEANING OF ADT

Data abstraction, or abstract data types, is a programming methodology where one defines not only the data structure to be used, but the processes to manipulate the structure like process abstraction, ADTs can be supported directly by programming languages To support it, there needs to be mechanisms for defining data structures encapsulation of data structures and their routines to manipulate the structures into one unit by placing all definitions in one unit, it can be compiled at one time information hiding to protect the data structure from outside interference or manipulation the data structure should only be accessible from code encapsulated with it so that the structure is hidden and protected from the outside objects are one way to implement ADTs, but because objects have additional properties.

- An Abstract Data Type is some sort of data together with a set of functions (interface) that operate on the data.
- Access is only allowed through that interface.
- Implementation details are 'hidden' from the user.
- An abstract data type (ADT) is an abstraction of a data structure
- An Abstract Data Type (ADT) is:
    - a set of values
    - a set of operations, which can be applied uniformly to all these values
- To *abstract* is to leave out information, keeping (hopefully) the more important parts
    - What part of a Data Type does an ADT leave out?
- An ADT specifies:
    - Data stored
    - Operations on the data
    - Error conditions associated with operations
- Example: ADT modeling a simple stock trading system
    - The data stored are buy/sell orders
    - The operations supported are
        - order buy(stock, shares, price)
        - order sell(stock, shares, price)
        - void cancel(order)
    - Error conditions:
        - Buy/sell a nonexistent stock
        - Cancel a nonexistent order

1.4 ADT Design Issues

- **Encapsulation:** it must be possible to *define* a unit that contains a data structure and the subprograms that access (manipulate) it
    - **design issues**:
        - Will ADT access be restricted through pointers?
        - Can ADTs be parameterized (size and/or type)?
- **Information hiding:** controlling access to the data structure through some form of interface so that it cannot be directly manipulated by external code
    - this is often done by using two sections of an ADT definition
        - public part (interface) constitutes those elements that can be accessed externally (often the interface permits only access to subprograms and constants)
        - the private part, which remains secure because it is only accessible by subprograms of the ADT itself
- Unit for encapsulation called a module
    - modules can be combined to form libraries of ADTs
- To define a module:
    - definition module: the interface containing a partial or complete type definition (data structure) and the subprogram headers and parameters
    - implementation module: the portion of the data structure that is to be hidden, along with all operation subprograms
- If the complete type declaration is given in the definition module, the type is "transparent" otherwise it is "opaque"
    - opaque types represent true ADTs and *must* be accessed through pointers

This restriction allows the ADT to be entirely hidden from user programs since the user program need only define a pointer.

1.5 Data representation in an ADT

1.5.1 Data Structures

- Many kinds of data consist of multiple parts, organized (structured) in some way
- A **data structure** is simply some way of organizing a value that consists of multiple parts
    - Hence, an array is a data structure, but an integer is not
- When we talk about data structures, we are talking about the implementation of a data type
- If I talk about the possible values of, say, complex numbers, and the operations I can perform with them, I am talking about them as an ADT
- If I talk about the way the parts ("real" and "imaginary") of a complex number are stored in memory, I am talking about a data structure

- An ADT may be implemented in several different ways
    - A complex number might be stored as two separate doubles, or as an array of two doubles, or even in some bizarre way
- An ADT must obviously have some kind of representation for its data
    - The user need not know the representation
    - The user should not be allowed to tamper with the representation
    - Solution: Make all data private
- But what if it's really more convenient for the user to have direct access to the data?
    - Solution: Use setters and getters

1.5.2 Example of setters and getters:

```
class Pair {
private int first, last;
public getFirst() { return first; }
public setFirst(int first) { this.first = first; }
public getLast() { return last; }
public setLast(int last) { this.last = last; }
}
```

1.5.4 Naming setters and getters:

- Setters and getters should be named by:
    - Capitalizing the first letter of the variable (first becomes First), and
    - Prefixing the name with get or set (setFirst)
    - For boolean variables, replace get with is (for example, isRunning)

2.1.5.5 Contracts:

- Every ADT should have a contract (or specification) that:
    - Specifies the set of valid values of the ADT
    - Specifies, for each operation of the ADT:
        - Its name
        - Its parameter types
        - Its result type, if any
        - Its observable behavior
    - Does *not* specify:
        - The data representation
        - The algorithms used to implement the operations

2.1.5.6 Importance of the contract

- A contract is an agreement between two parties; in this case
    - The implementer of the ADT, who is concerned with making the operations correct and efficient
    - The applications programmer, who just wants to *use* the ADT to get a job done

- It doesn't matter if you are *both* of these parties; the contract is *still essential* for good code
- This separation of concerns is essential in any large project

1.6 IMPLEMENTING AN ADT

- To implement an ADT, you need to choose:
    - a data representation
        - must be able to represent all necessary values of the ADT
        - should be private
    - an algorithm for each of the necessary operations
        - must be consistent with the chosen representation
        - all auxiliary (helper) operations that are not in the contract should be private
        - Remember: Once other people (or other classes) are using your class:
    - It's easy to add functionality
    - You can only remove functionality if no one is using it.

1.7 LET US SUM UP

- A Data Type describes values, representations, and operations
- An Abstract Data Type describes values and operations, but not representations
    - An ADT should protect its data and keep it valid
        - All, or nearly all, data should be private
        - Access to data should be via getters and setters
    - An ADT should provide:
        - A contract
        - A necessary and sufficient set of operations

1.8 KEYWORDS

1.9 QUESTIONS FOR DISCUSSION

1. Explain Abstract Data Types (ADT)?
2. Describe ADT's Implementation?

1.10 SUGGESTION FOR BOOKS READING

UNIT
2
LIST ADT

STRUCTURE

2.0 AIMS AND OBJECTIVES

At the end of this lesson, students should be able to demonstrate appropriate skills, and show an understanding of the following:

- Aims and objectives of List ADTs
- Linked List description and its implementation
- Types of Linked List

2.1 INTRODUCTION OF LIST ADT

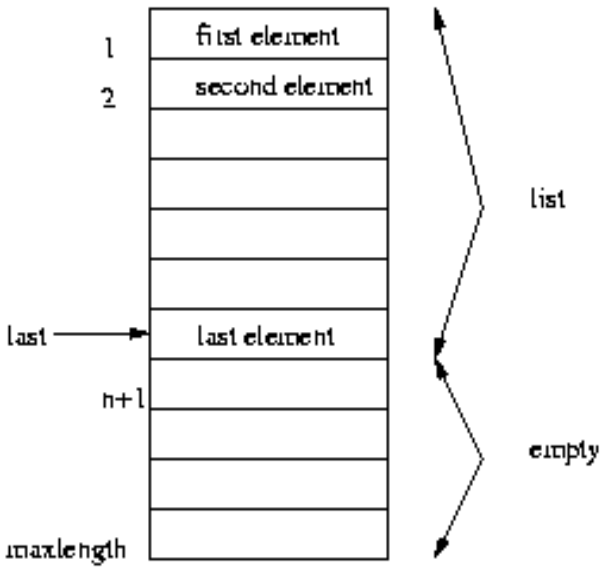
We will deal with a general list of the form A_1, A_2, A_3, … A_N. we say that the size of this list is N. we will call the special list of size 0 an empty list.

- A sequence of zero or more elements
    1. A_1, A_2, A_3, A_N
- N: length of the list
- A_1: first element
- A_N: last element
- A_i: position i
- If N=0, then empty list
- Linearly ordered
    2. A_i precedes A_{i+1}
    3. A_i follows A_{i-1}

2.2 SIMPLE ARRAY IMPLEMENTATION OF LISTS

All of these instructions can be implemented just by using an array. Even if the array is dynamically allocated, an estimate of the maximum size of the list is required. Usually this requires a high overestimate, which wastes considerable space. This could be a serious limitation, especially if there are many lists of unknown size.

- Elements are stored in contiguous array positions



- Requires an estimate of the maximum size of the list
    - waste space

- **print List and find:** linear
- **findKth:** constant
- **insert and delete:** slow
    - e.g. insert at position 0 (making a new element)
        - requires first pushing the entire array down one spot to make room
    - e.g. delete at position 0
        - requires shifting all the elements in the list up one
    - On average, half of the lists needs to be moved for either operation

2.3 POINTER IMPLEMENTATION (LINKED LIST)

- Ensure that the list is not stored contiguously
    - Use a linked list
    - A series of structures that are not necessarily adjacent in memory

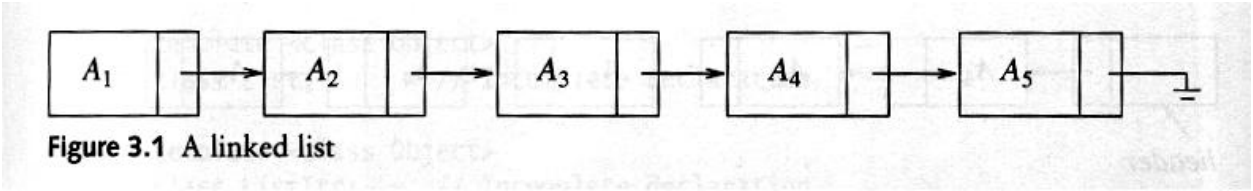


Figure 3.1 A linked list

- Each node contains the element and a pointer to a structure containing its successor
    - The last cell's next link points to NULL
- Compared to the array implementation,
    - The pointer implementation uses only as much space as is needed for the elements currently on the list, but requires space for the pointers in each cell

2.4 TYPES OF LINKED LISTS

- **Types of linked lists:**
    - **Singly linked list**
        - Begins with a pointer to the first node
        - Terminates with a null pointer
        - Only traversed in one direction
    - **Circular, singly linked**
        - Pointer in the last node points back to the first node
    - **Doubly linked list**
        - Two "start pointers" – first element and last element
        - Each node has a forward pointer and a backward pointer
        - Allows traversals both forwards and backwards
    - **Circular, doubly linked list**
        - Forward pointer of the last node points to the first node and backward pointer of the first node points to the last node

2.5 SINGLY LINKED LISTS

In real time systems most often the number of elements will not be be known in advance. The major drawback of an array is that the number of elements must be known in advance. Hence an alternative approach is required. This gives rise to the concept called linked list.

A linked list allows data to be inserted into and deleted from a list without physically moving any data. Additionally, if the linked list is implemented using dynamic data structures, the list size can vary as the program executes.

Linked list is efficient because of the following reasons:

- Insert and delete operations in $O(1)$ time
- Searching operations in $O(n)$ time

Where O represents **Big O notation** or **Big Oh notation**, and also **Landau notation** or **asymptotic notation**, a mathematical notation used to describe the asymptotic behavior of functions. It is an asymptotic tight bound which is useful in the analysis of the complexity of algorithms.

But, memory overhead may arise in a linked list, but it is allocated only to entries that are present.

Operations:

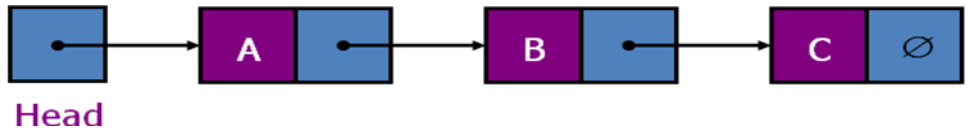
- **Print List:** print the list
- **Make Empty:** create an empty list
- Find: locate the position of an object in a list
  - list: 34,12, 52, 16, 12
  - find(52) $\rightarrow$ 3
- **Insert:** insert an object to a list
  - insert(x,3) $\rightarrow$ 34, 12, 52, x, 16, 12
- **Remove:** delete an element from the list
  - remove(52) $\rightarrow$ 34, 12, x, 16, 12
- **FindKth:** retrieve the element at a certain position

2.6 IMPLEMENTATION OF A LIST ADT

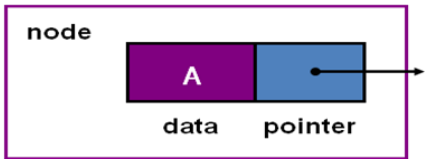
- Choose a **data structure** to represent the ADT
  - E.g. arrays, records, etc.
- Each operation associated with the ADT is implemented by one or more subroutines
- Two standard implementations for the list ADT
  - Array-based

- Linked list

Linked Lists:



- A linked list is a series of connected nodes
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- Head: pointer to the first node
- The last node points to NULL



A Simple Linked List Class

- We use two classes: **Node** and **List**
- Declare Node class for the nodes
  - data: double-type data in this example
  - next: a pointer to the next node in the list

```
class Node {
public:
double data;          // data
Node* next;           // pointer to next
};
```

- Declare List, which contains
  - Head: a pointer to the first node in the list.
- Since the list is empty initially, head is set to NULL
- Operations on List

```
class List {
public:
List(void) { head = NULL; }  // constructor
~List(void);                          // destructor
bool IsEmpty() { return head == NULL; }
Node* InsertNode(int index, double x);
int FindNode(double x);
int DeleteNode(double x);
void DisplayList(void);
private:
Node* head;
```

};

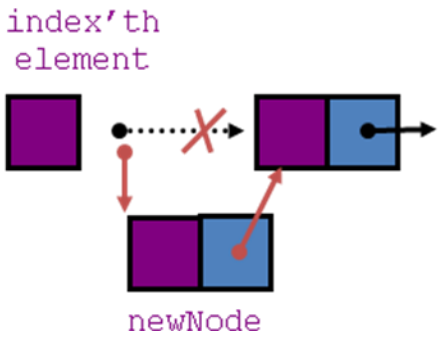
2.6.1 Operations of List

- IsEmpty: determine whether or not the list is empty
- InsertNode: insert a new node at a particular position
- FindNode: find a node with a given value
- DeleteNode: delete a node with a given value
- DisplayList: print all the nodes in the list

2.6.2 Inserting a new node

- Node* InsertNode(int index, double x)
  - Insert a node with data equal to x after the index'th elements. (i.e., when index = 0, insert the node as the first element; when index = 1, insert the node after the first element, and so on)
  - If the insertion is successful, return the inserted node. Otherwise, return NULL.
  - (If index is < 0 or > length of the list, the insertion will fail.)

- Steps
  - Locate index'th element
  - Allocate memory for the new node
  - Point the new node to its successor
  - Point the new node's predecessor to the new node



- Possible cases of InsertNode
  - Insert into an empty list
  - Insert in front
  - Insert at back
  - Insert in middle

- But, in fact, only need to handle two cases
  - Insert as the first node (Case 1 and Case 2)
  - Insert in the middle or at the end of the list (Case 3 and Case 4)

```
Node* List::InsertNode(int index, double x) {
if (index < 0) return NULL;
int currIndex   =       1;
Node* currNode      =       head;
while (currNode && index > currIndex) {
currNode        =       currNode->next;
currIndex++;
}
if (index > 0 && currNode == NULL) return NULL;
```

Try to locate index'th node. If it doesn't exist, return NULL.

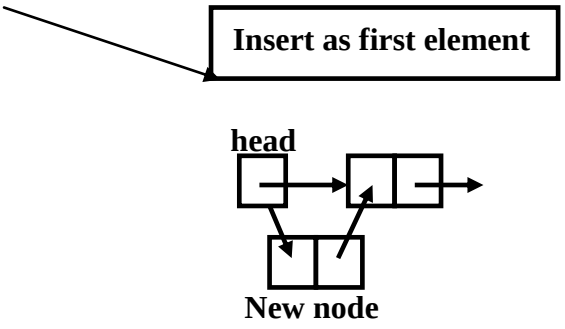
```
Node* newNode       =       new     Node;
newNode->data       =       x;
if (index == 0) {
newNode->next       =       head;
head            =       newNode;
}
else {
newNode->next       =       currNode->next;
currNode->next      =       newNode;
}
return newNode;
}
Node* List::InsertNode(int index, double x) {
if (index < 0) return NULL;

int currIndex   =       1;
Node* currNode      =       head;
while (currNode && index > currIndex) {
currNode        =       currNode->next;
currIndex++;
}
if (index > 0 && currNode == NULL) return NULL;
```

```
Node* newNode       =       new     Node;
newNode->data       =       x;
```

Create a new node

```
if (index == 0) {
newNode->next       =       head;
head            =       newNode;
}
```

```
else {
newNode->next     =     currNode->next;
currNode->next    =     newNode;
}
return newNode;
}
Node* List::InsertNode(int index, double x) {
if (index < 0) return NULL;

int currIndex  =     1;
Node* currNode    =     head;
while (currNode && index > currIndex) {
currNode     =     currNode->next;
currIndex++;
}
if (index > 0 && currNode == NULL) return NULL;

Node* newNode     =     new     Node;
newNode->data     =     x;
if (index == 0) {
newNode->next     =     head;
head              =     newNode;
}
```




```
else {
newNode->next     =     currNode->next;
currNode->next    =     newNode;
}
return newNode;
}
```

2.6.3 Finding a node:

- int FindNode(double x)
  - Search for a node with the value equal to x in the list.
  - If such a node is found, return its position. Otherwise, return 0.

```
int List::FindNode(double x) {
Node* currNode      =      head;
int currIndex   =      1;
while (currNode && currNode->data != x) {
currNode      =      currNode->next;
currIndex++;
}
if (currNode) return currIndex;
return 0;
}
```

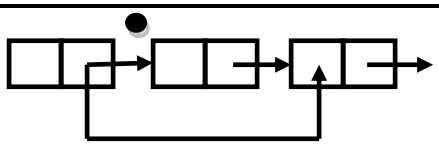
6.4 DELETING A NODE

- Int Delete Node(double x)
  - Delete a node with the value equal to x from the list.
  - If such a node is found, return its position. Otherwise, return 0.
- Steps
  - Find the desirable node (similar to FindNode)
  - Release the memory occupied by the found node
  - Set the pointer of the predecessor of the found node to the successor of the found node
- Like Insert Node, there are two special cases
  - Delete first node
  - Delete the node in middle or at the end of the list

```
int List::DeleteNode(double x) {
Node* prevNode      =      NULL;
Node* currNode      =      head;
int currIndex   =      1;
while (currNode && currNode->data != x) {
prevNode      =      currNode;
currNode      =      currNode->next;
currIndex++;
}
if (currNode) {
if (prevNode) {
prevNode->next      =      currNode->next;
delete currNode;
}
else {
head          =      currNode->next;
```

Try to find the node with its value equal to x

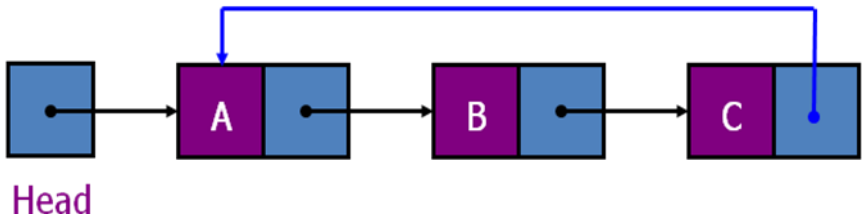
```
delete currNode;
}
return currIndex;
}
return 0;
}
// Deleting a node
int List::DeleteNode(double x) {
Node* prevNode = NULL;
Node* currNode = head;
int currIndex = 1;
while (currNode && currNode->data != x) {
prevNode = currNode;
currNode = currNode->next;
currIndex++;
}
if (currNode) {
if (prevNode) {
prevNode->next = currNode->next;
delete currNode;
}
```



```
else {
head = currNode->next;
delete currNode;
}
return currIndex;
}
return 0;
}
```

2.2.7 Circular linked lists

- The last node points to the first node of the list
- How do we know when we have finished traversing the list? (Tip: check if the pointer of the current node is equal to the head.)



Now we shall see the class implementation of a linked list:-

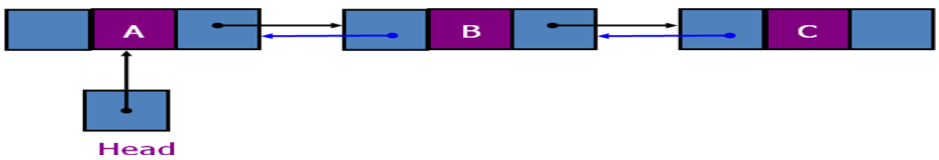
```
//Implementation of a linked list
```

```
#include<iostream.h>
#include<conio.h>
 class linklist{
private:
struct node{
int data;
node *link;
}*p;
public:
linklist();
void addend(int num);
void addbeg(int num);
void addafter(int c,int num);
void del(int num);
void display();
int count();
 ~linklist();
};
//add a new node at the beginning of the linked list
void linklist::addbeg(int num)
{
 node *q;
//add newnode
q=new node;
q->data=num;
q->link=p;
p=q;
}
```

The above function adds a new node at the beginning.

2.8 DOUBLY LINKED LISTS

- Each node points to not only successor but the predecessor
- There are two NULL: at the first and last nodes in the list
- Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards



- A doubly linked list provides a natural implementation of the List ADT
- Nodes implement Position and store:
    - element

- link to the previous node
- link to the next node

- Special trailer and header nodes

Sample program of Double linked list:

```
// Node for doubly linked list of floats.
struct Node {
Node* next;  // Pointer to next element
Node* prev;  // Pointer to previous element
float data;
};
```

The common operations of inserting and deleting become easy. There are still some problems that remain.

- The ends present special cases in insertion and deletion.

2.8.1 Circular doubly linked list

- precede pointer of the dummy head node points to the last node
- next reference of the last node points to the dummy head node
- No special cases for insertions and deletions

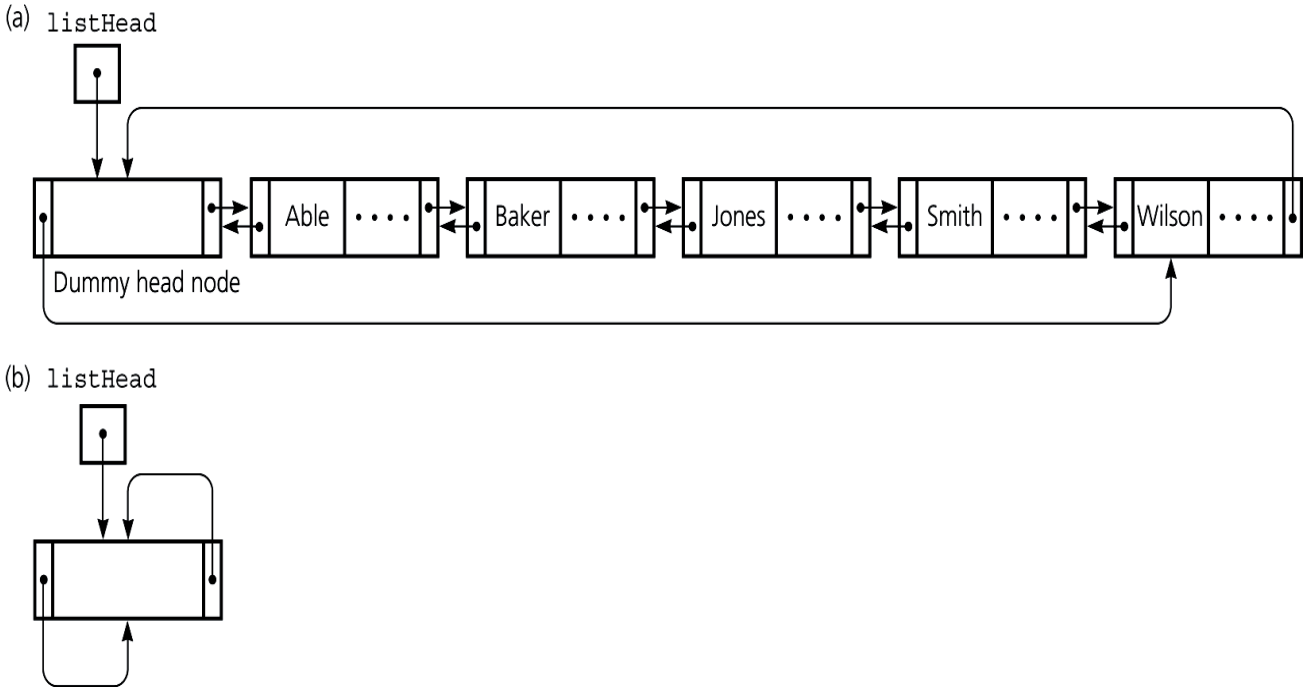


Figure 4.29 (a) A circular doubly linked list with a dummy head node
(b) An empty list with a dummy head node

- To delete the node to which cur points
  (cur->precede)->next = cur->next;

(cur->next)->precede = cur->precede;

- To insert a new node pointed to by newPtr before the node pointed to by cur

```
newPtr->next = cur;
newPtr->precede = cur->precede;
cur->precede = newPtr;
newPtr->precede->next = newPtr;
```

Performance

- In the implementation of the List ADT by means of a doubly linked list
  - The space used by a list with n elements is O(n)
  - The space used by each position of the list is O(1)
  - All the operations of the List ADT run in O(1) time
  - Operation element() of the Position ADT runs in O(1)time

2.9 LET US SUM UP

- Each pointer in a linked list is a pointer to the next node in the list
- Array-based lists use an implicit ordering scheme; pointer-based lists use an explicit ordering scheme
- Algorithms for insertions and deletions in a linked list involve traversing the list and performing pointer changes
  - Inserting a node at the beginning of a list and deleting the first node of a list are special cases
- A class that allocates memory dynamically needs an explicit copy constructor and destructor
- Recursion can be used to perform operations on a linked list
- In a circular linked list, the last node points to the first node
- Dummy head nodes eliminate the special cases for insertion into and deletion from the beginning of a linked list

2.10 QUESTIONS FOR DISCUSSION

1. Why linked list is efficient?
2. How memory is managed dynamically in linked list?
3. How a linked list could be traversed?
4. Create a doubly linked list with the following set of data: 12 15 78 96 32
5. State some applications of linked list.

2.11 SUGGESTION FOR BOOKS READINGS

UNIT
3
STACK ADT

STRUCTURE

3.0 AIMS AND OBJECTIVES

At the end of this lesson, students should be able to demonstrate appropriate skills, and show an understanding of the following:

- Aims and objectives of Stack ADTs
- Stack and its implementation

3.1 INTRODUCTION OF STACK ADT

A Stack is a linear data structure. It is more restrictive than linked list. You may wonder, then, why is it necessary. The answer is that there are times when it is nice to describe an algorithm in terms of very primitive data structures such as stacks and queue—then you are not as distracted with extra operations that you get with liked lists.

A stack is a list in which all insertions and deletions are made at one end, called the top. The last element to be inserted into the stack will be the first to be removed. Thus stacks are sometimes referred to as Last In First Out (LIFO) lists. A stack is a last in first out structure (LIFO). It can be visualized as a pile of plates. Plates are added to the top of pile and the plate at the top of the pile which was the last to be added will be the first to be removed.

3.1.1 Operations:

InitStack(Stack): creates an empty stack. This creates a stack initially which contains NULL value.

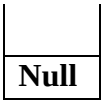


Figure: Creating Empty stack

Push (Item): pushes an item on the stack Push operation inserts the new element at the top of the stack.

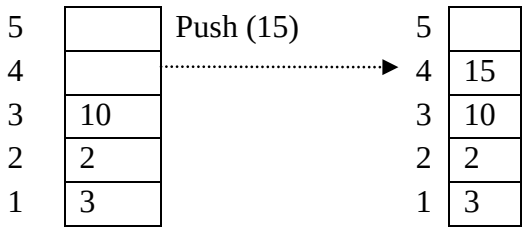
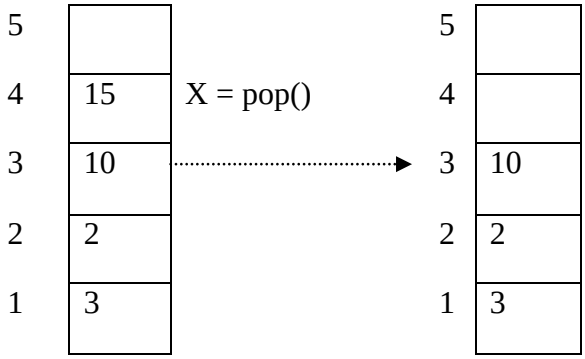


Figure: Push operation

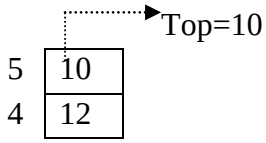
Pop (Stack): removes the first item from the stack Pop operation removes the first element (recently added) from the top of the stack.



X=15

Figure: Pop operation

Top (Stack): returns the first item from the stack without removing it . This operation retrieves the value stored in the top of the stack.



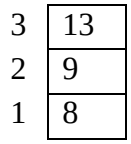


Figure: Returning Top of stack

Is Empty (Stack): returns true if the stack is empty . This operation is used to check whether the stack is empty or not.

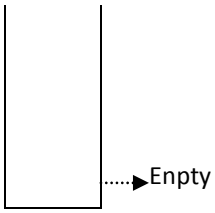


Figure: Empty stack

3.2 STACK IMPLEMENTATION USING ARRAYS

Stacks are a subclass of Linear Lists; all access to a stack is restricted to one end of the list, called the top of stack. Visually, it can be a stack of books, coins, plates, etc. Addition of books, plates, etc. occurs at the top of the stack; removal also occurs at the top of the stack.

A Stack is an ordered (by position, not by value) collection of data (usually homogeneous), with the following operations defined on it:

3.2.1 Operations

An array-based stack requires two values as a priori: the type of data contained in the stack, and the size of the array.

The stack itself is a structure containing two elements: Data, the array of data, and Top, an index that keeps track of the current top of stack; that is, it tells us where data is added for a Push and removed for a Pop.

Operations performed in a Stack are:

Initialize : Initialize internal structure; create empty stack. This operation creates an empty stack.

Pseudo-Code:

Set Top to 1
Return Stack Array and Top to the user.

Pus: Add new element to top of stack. This operation inserts the element to be added at the top of the stack.

Pseudo-Code:

If Stack is Full Then
Output “Overflow, Stack is full, cannot push.”
Else
Place Element in Stack (Top)
Increment Top
Return Stack and Top to the user.

Example:

```
void Push(int elem)
{
top++;
StackArray[top] = elem;
}
```

Pop: Remove top element from stack. This operation removes the element from the top of the stack.

Pseudo-Code:
If Stack is Empty Then
Output “Underflow, Stack is empty, cannot pop.”
Else
Top:= Top-1;
Return element in Stack (Top)

Example:

```
int Pop()
{
int elem = StackArray[top];
top--;
return elem;
}
```

Empty : True iff stack has no elements. This operation checks whether the stack is empty or not.

Pseudo-Code:
If Top = 1 Then
Return Empty_Stack := True
Else
Return Empty_Stack := False

StackTop : Returns copy of top element of the stack (without popping it). This operation retrieves the value at the top of the stack.

Pseudo-Code :
If Top = 1 Then
Output "Stack is Empty – Cannot get Top"
Else
Return Stack(Top-1)

Size: This operation returns number of elements in the stack.

Pseudo-Code:

Return (Top-1) ; // index of the top element of the stack
int StackArray[50]; // StackArray can contain up to 50 numbers
Here -1 used to indicate an empty stack. So Top-1 indicates the TOP element.

3.3 ABSTRACT DATA TYPE - STACK

Abstract data types or ADTs are a mathematical specification of a set of data and the set of operations that can be performed on the data. It is a user-defined type with a set of values and a collection of operations that may be performed on those values. Access to the data is allowed only through the interface. It is a type whose implementation details are hidden and can only be handled using the publicly accessible operations provided for it.
An example of a Stack ADT is given below:

The Stack-ADT

```
# ifndf  stack.h
#define STACKSIZE 50
Stack specification
struct Stack
{
int item[STACKSIZE];
int top;
}
void InitStack(Stack &st);
```

```
void Push(Stack &st, int elem);
int Pop (Stack &st);
int Top (Stack st);
bool isEmpty(Stack st);

# include <stack.h>
void main() {
Stack st1, st2; // declare 2 stack variables
InitStack(st1); // initialise them
InitStack(st2);
Push(st1, 13); // push 13 onto the first stack
Push(st2, 32); // push 32 onto the second stack
int i = Pop(st2); // now popping st2 into i
int j = Top(st1); // returns the top of st1 to j
// without removing element
}
```

3.3.1 Stack Implementation using Linked list

One disadvantage of using an array to implement a stack is the wasted space---most of the time most of the array is unused. A more elegant and economical implementation of a stack uses a *linked list*, which is a data structure that links together individual data objects as if they were ``links'' in a ``chain'' of data.

A linked-list is somewhat of a dynamic array that grows and shrinks as values are added to it. Rather than being stored in a continuous block of memory, the values in the dynamic array are linked together with pointers. Each element of a linked list is a structure that contains a value and a link to its neighbor. The link is basically a pointer to another structure that contains a value and another pointer to another structure, and so on.

A linked list takes up only as much space in memory as is needed for the length of the list. The list expands or contracts as you add or delete elements. Elements can be added to (or deleted from) either end, or added to (or deleted from) the middle of the list.

A standard method of implementing a stack within applications is by means of a single linked list. The stack can only access the head of the list which is known as the 'top' of the stack.

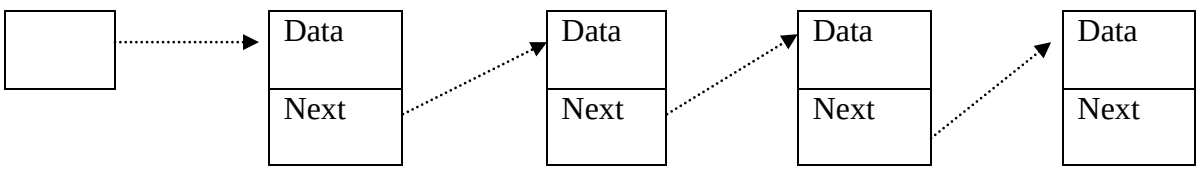


Figure: Linked stack

3.3.2 Operations

IsEmpty() - return 1 if stack is empty; otherwise, return 0.

Pseudo-Code :

```
IsEmpty(Top t)
{
if (top=Null)
return true;
else
return false;
}
```

Top () - return the value of the top element of the stack.

Pseudo-Code :

```
DATA Top(TOP t)
{
return t;
}
```

Push (Data): This operation inserts an element at the top of the stack.

Steps for the push operation:

A push operation is used for inserting an element in to a stack. Elements are always inserted in the beginning of the stack.

1. Create a new node.
2. Make the new node's next point to the node pointed by Top.
3. Make the top point to the new node.

Pseudo-Code:

```
Definition of a record type that describes a node in a linked stack:
type
stacknode = record
info: information being saved
nextnode: pointer to (i.e. address of) next node in stack
end
Definition of  the structure of the stack header:
type
```

stackheader = record
info: string (maybe just a description like "StkHdr")
top: pointer to (i.e. address of) top node in stack
end

Push(P: pointer to node to be pushed onto the stack)
P-> nextnode = stack-> top //put in P's nextnode field the address found in stackheader's top field
stack-> top = P

Example:

```
struct stack
{
int no;
struct stack *next;
} *top = NULL;
typedef struct stack st;

void push()
{
st *node;
node = (st*)malloc(sizeof(st));
printf("Enter the Number");
scanf("%d",&node->no);
node->next = top;
top = node;
}
```

POP() : This operation deletes an element from the top of the stack and returns the value.
Steps for the pop operation:

A pop operation is basically used for retrieving a value from the stack. This also removes the element from the stack.

1. Save tempnode as the node pointed by Top
2. Retrieve value as data of the tempnode.
3. Make Top point to the next node.

Pseudo-Code :
Pop(P: pointer to hold address of node popped off)
if stack-> top = null
error - stack is empty
else

```
P = stack-> top                 //assign to pointer P the address of the top node on the stack
stack-> top = P-> nextnode      //move P's nextnode address into the stackheader top field
endif
```

Example:

```
int pop()
{
st *temp;
temp = start;
if(top==NULL)
{
printf("stack is already empty");
getch();
exit();
}
else
{
top = top -> next;
free(temp);
}
return (temp->no);
}
```

3.4 APPLICATION OF STACKS

Applications of the stack are Recursion, evaluation of arithmetic expressions and control transfers to subprograms. Recursion is the process of a function calling itself till a condition is satisfied. This is an important feature in many programming languages. There are many algorithmic descriptions for recursive function. Evaluation of arithmetic expressions is done using stacks. Stacks are suitable data structures to backtrack the program flow or control transfers.

2.3.4.1 Evaluation of expressions

Computers solve arithmetic expressions by restructuring them so that the order of each calculation is embedded in the expression. Once converted, to the required notation (Either Prefix or Postfix), an expression can be solved

Types of Expression

Any expression can be represented in 3 ways namely – infix, Prefix and Postfix. Consider the expression 4 + 5 * 5. This can be represented as follows:

Infix - 4 + 5 * 5.
Prefix - + 4 * 5 5 (The Operators precede the operands.)
Postfix - 4 5 5 * + (The Operators follow the operands.)
The default way of representing a mathematical expression is in the form of Infix notation. This method is called Polish Notation (discovered by the Polish mathematician Jan Lukasiewicz).

The Valuable aspect of RPN (Reverse Polish Notation or Postfix)

- Parentheses are not necessary
- Easy for a compiler to evaluate an arithmetic expression.

Postfix (Reverse Polish Notation)

Postfix notation arises from the concept of post-order traversal of an expression tree. For now, consider postfix notation as a way of redistributing operators in an expression so that their operation is delayed until the compile time.
Consider quadratic formula:

X = (-b + (b^2-4*a*c)^0.5)/(2*a)
In postfix form the formula becomes
X **b @ b 2 ^ 4 a * c * - 0.5 ^ 2a*/=**
Where '@' represents the unary '-' operator.
Notice the order of the operands remain the same but the operands are redistributed in a unconventional way.

Purpose: The reason for using postfix notation is that a fairly simple algorithm exists to evaluate such expressions based on using a stack:

Postfix Evaluation

Consider the postfix expression:
6523+8*+3+*

Algorithm

initialize stack to empty;
while (not end of postfix expression){
get next postfix item;
if(item is value)
push it onto the stack;
else if (item is binary operator){
Pop the stack to x;
Pop the stack to y;

Perform y operator x;
Push the results onto the stack;
}else if (item is unary operator){
pop the stack to x;
Perform operator (x);
Push the results onto the stack
}
}

Unary operators are Unary minus, square root, sin, cos, exp, etc.,
So for 6523+8*+3+*

The first item to be pushed into the stack is value (6)
The next item is value (5)
The next item is value (2)
The next item is value (3)

And the stack becomes

TOS=>

The remaining items are now: +8*+3+*, So next a '+' is read (a binary operator), so 3 and 2 are popped from the stack and their sum '5' is pushed onto the stack:

TOS=>

Next 8 id pushed and the next time is the operator*:

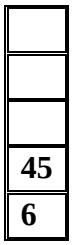
TOS=>

TOS=>

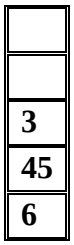
(8, 5 popped, 40 pushed)

Next the operator + followed by 3:

TOS=>



TOS=>



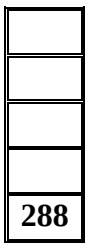
(40,5 popped, 45 pushed, 3 pushed)

Next is operator +, so 3 and 45 are popped and 45+3=48 is pushed

TOS=>

Next is operator *, so 48 and 6 are popped, and 6 * 48=288 is pushed

TOS=>



Now there are no more items and there is a single value on the stack, representing the final answer 288. The answer was found with a single traversal of the postfix expression, with the stack being used as a kind of memory storing values that are waiting for their operands.

Infix to Postfix (RPN) Conversion''

Of course postfix notation is of little use unless there is an easy method to convert standard (Infix) expressions to Postfix. Again a simple algorithm exists that uses a stack:

Algorithm

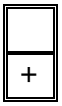
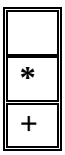
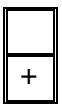
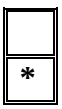
initialize stack and postfix output to empty;
while (not end of infix expression){
get next infix item

if(item is value) append item to pfix o/p
else if(item = = ‘(‘) push item onto stack
else if(item = = ‘)‘){
pop stack to x
while (x!= ‘(‘)
app.x to pfix o/p & pop stack to x
}else{
while(precedence(stack top)> = precedence (item))
pop stack to x & app.x to pfix o/p
push item onto stack
}
}
while(stack not empty)
pop stack to x and append x to pfix o/p

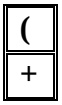
Operator Precedence (for this algorithm):

4: ‘(‘- only popped if a matching ‘)’ is found.
3: All unary operatos
2: /*
1: + -

The following diagram illustrates the algorithm:
eg., consider the infix expression **a + b * c + (d*e + f) * g**

		Stack	Output
TOS=>		ab	
	abc TOS=>		
Abc*+			
	TOS=>		
			

TOS=>



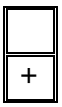
Abc*+de

Abc*+de*f

TOS=>

Abc*+de*f+

TOS=>



Abc*+de*f+g

TOS=>

empty

Abc*+de*f+g+

Evaluation of arithmetic expressions:

Usually, arithmetic expressions are written in infix notation, e.g.
A+B*C
An expression can as well be written in postfix notation (also called reverse polish notation):

A+B becomes AB+
A*C becomes AC*
A+B*C becomes ABC*+
(A+B)*C becomes AB+C*

Evaluating expressions

Given an expression in postfix notation. Using a stack they can be evaluated as follows:

- Scan the expression from left to right
- When a value (operand) is encountered, push it on the stack
- When an operator is encountered, the first(a) and second element(b) from the stack are popped and the operator(x) is applied (b x a)
- The result is pushed on the stack

Evaluating Expression

Example: 7 1 3 + -4 *
Value : 12

Another Stack Example

- Are stacks only useful for making sense of postfix notation expressions?
- Not so, Stacks have many uses!
- Another e.g. : Reversing word order
  STACKS .
  SKCATS

-Simply push each letter onto the stack, then pop them back off again
The next function 'Initialize a stack to be empty before it is first used in a program:
/*Initialize: initialize the stack to be empty*/
Void initialize (stack_type *stack_ptr)
{stack_ptr->top=0;}

Stack when represented using a single-dimension array, has the first or bottom element in the stack stored at stack[0], the second at stack [1], and the i^{th} element at stack [i-1] positions. Associated with the array is an index variable, called top, which points to the top element in the stack. The stack is said to be empty if the stack pointer value is less than or equal to 0. Similarly the stack is said to be full if the stack pointer value reaches a value greater than the maximum size of the stack.The access time from stack is always 0, (1), ie., independent of the number of elements in it. Space requirement is 0(n).

Infix to Postfix Conversion

Preconditions: A non-empty input string containing the expression in infix form
Postconditions: A string in postfix form that is equivalent to the infix expression
Pseudocode:

1. Create a user Stack
2. Get the infix expression from the user as a string, say Infix
3. Check if the paranthesis are balanced as follows:
   For I in 1.. length(Infix) do
   - If Infix(I) = '(' then Push onto the Stack
   - If Infix(I) = ')' then Pop one element form the Stack
4. If Stack is non-empty
   a. Display "non-balanced expression"
   b. Goto 2
5. Create a new string Postfix
6. Set Postfix_Index to 1

7. For I in 1 .. Length(Infix)
   a. If Infix(I) is an operand, append it to postfix string as follows:
      i. Postfix(Postfix_Index) := Infix(I);
      ii. Postfix_Index:=Postfix_Index + 1;

   b. If the Infix(I) is an operator, process operator as follows
      1. Set done to false_
      2. Repeat
      a. If Stack is empty or Infix(I) is '(' then
      i. push Infix(I) onto stack
      ii. set done to true
      b. Else if precedence(Infix(I)) > precedence(top operator)
      i. Push Infix(I) onto the stack (ensures higher precedence operators evaluated first)
      ii. set done to true
      c. Else
      i. Pop the operator stack
      ii. If operator popped is '(', set done to true
      iii. Else append operator popped to postfix string
      3. Until done
8. While Stack is not empty
   a. Pop operator
   b . Append it to the postfix string
9. Return Postfix

3.4.2 Recursion

A Recursive function is a function whose definition is based upon itself, i.e a function contains either a call statement to another function that may eventually result in a call statement to the original function then that function is called a recursive function. For example, finding the factorial of a given number.

Each call to a subroutine requires that the subprogram have a storage area where it can keep its local variables, its calling parameters and its return address. For a recursive function a storage areas for subprogram calls are kept in a stack. Therefore any recursive function may be rewritten in a non-recursive form using stack.

3.4.3 Function Calls and Stack

A stack is used by programming languages for implementing function calls.

3.4.4 Tower Of Hanoi

Tower of hanoi is a historical problem, which can be easily expressed using recursion. There are N disks of decreasing size stacked on one needle, and two other empty needles. It is required to stack all the disks onto a second needle in the decreasing order of size. The third needle can be used as a temporary storage.

The movement of the disks must confirm to the following rules,

1. Only one disk may be moved at a time
2. A disk can be moved from any needle to any other.
3. The larger disk should not rest upon a smaller one.

3.5 LET US SUM UP

A Stack is a linear data structure. A stack is a list in which all insertions and deletions are made at one end, called the top. The last element to be inserted into the stack will be the first to be removed. Thus stacks are sometimes referred to as Last in First out (LIFO) lists. It can be implemented using Arrays as well as Linked list.

3.6 QUESTIONS FOR DISCUSSION

1. What is meant by LIFO technique?
2. How a stack can be visualized?
3. Define Stack ADT.
4. How the size of the stack can be verified?
5. Describe the push and pop operation in a linked implementation of a Stack.

3.7 SUGGESTION FOR BOOKS READING

UNIT
4
QUEUE ADT

STRUCTURE

4.0 AIMS AND OBJECTIVES

At the end of this lesson, students should be able to demonstrate appropriate skills, and show an understanding of the following:

- Aims and objectives of Queue ADTs
- Queue and its implementation

2.4.1 Introduction of Queue ADT

Like stacks, queues are lists, with a queue; however, insertion is done at one end, whereas deletion is performed at the other end.

In everyday life, we encounter queues everywhere - a line of people waiting to buy a ticket or waiting to be served in a bank; all such lines of people are queues. The first person in line is the next one served, and when another person arrives, he/she joins the queue at the rear.

A queue is a chronologically ordered list; the only difference between a stack and a queue is that, in a stack, elements are added and removed at the same end (the top), whereas in a queue, values are added at one end (the rear) and removed from the other end (the front).

The order of elements in a queue, therefore, perfectly reflects the order in which they were added: the first element added will be the first one removed (so queues are called FIFO - ``first in first out''), and the last one added will be the last one removed. A Queue is an ordered collection of items from which items may be deleted at one end (called the front of the queue) and into which items may be inserted at the other end (the rear of the queue).

4.1.1 Definition:

A Queue is an ordered collection of items from which items may be deleted at one end (called the front of the queue) and into which items may be inserted at the other end (the rear of the queue).

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out (FIFO) scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
    - enqueue(object o): inserts element o at the end of the queue
    - dequeue(): removes and returns the element at the front of the queue

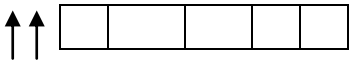
- **Auxiliary queue operations:**
    - front(): returns the element at the front without removing it
    - size(): returns the number of elements stored
    - isEmpty(): returns a Boolean value indicating whether no elements are stored
- **Exceptions**
    - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException

Exercise: Queues

- **Describe the output of the following series of queue operations**
    - enqueue(8)
    - enqueue(3)
    - dequeue()
    - enqueue(2)
    - enqueue(5)
    - dequeue()
    - dequeue()
    - enqueue(9)
    - enqueue(1)

2.4.1.2 Operations

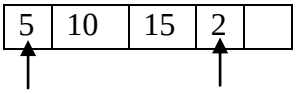
The following operations can be applied to a queue:
InitQueue(Queue): creates an empty queue



F R

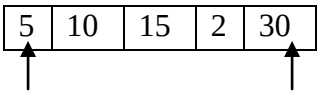
Figure: Empty Queue

append(Item): inserts an item to the rear of the queue



Front **Rear**

APPEND (30)

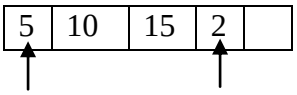


Front **Rear**

Figure: Append operation

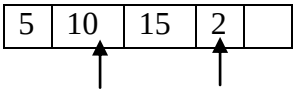
remove(Queue): removes an item from the front of the queue

Elements can only be added to the rear of the queue and removed from the front of the queue.



Front **Rear**

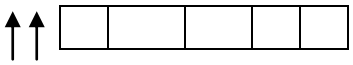
X=remove()



X=5 **Front** **Rear**

Figure: Remove operation

isEmpty(Queue): returns true if the queue is empty



F R

Figure: Empty Queue

4.2 ARRAY IMPLEMENTATION

Queues are a subclass of Linear Lists, which maintain the First-In-First-Out order of elements. Insertion of elements is carried out at the 'Tail' of the queue and deletion is carried out at the 'Head' of the queue.

A queue is an ordered (by position, not by value) collection of data (usually homogeneous), with the following operations defined on it:

4.2.1 Operations

An array-based queue requires us to know two values a priori: the type of data contained in the queue, and the size of the array. For our implementation, we will assume that the queue stores integer numbers and can store 10 numbers.

The queue itself is a structure containing three elements: Data, the array of data, Head, an index that keeps track of the first element in the queue (location where data is removed from the queue), and Tail, an index that keeps track of the last element in the queue (location where elements are inserted into the queue).

The operations that can be performed in a Queue are:
Initialize: Initialize internal structure; create an empty queue.

Pseudo-Code:
Set Head to 1
Set Tail to 1
Return the Queue to the user.

Enqueue: Add new element to the tail of the queue. This operation adds a new element at the rear end of the queue.

Pseudo-Code:
If Queue is Full (Tail = Size of Queue + 1) Then
    Output "Overflow, Queue is full, cannot Enqueue."
Else
    Place Element in Queue (Tail)
    Increment Tail (Tail = Tail + 1)
    Return the queue to the user.

Dequeue : Remove an element from the head of the queue. This operation removes an element only from the front of the queue.

Pseudo-Code:
If Queue is Empty (Head = Tail) Then
    Output "Underflow, Queue is empty, cannot dequeue."
Else

Element: = Queue(Head);
Move all the elements from head+1 to Size of Queue one step to the left
Return Element

Empty **:** True if the queue has no elements. This operation checks whether the queue is empty or not.

Pseudo-Code :

If Head = Tail Then
Return Empty_Queue := True
Else
Return Empty_Queue:= False
Full
Preconditions : Queue
Post-Conditions: Return True if the Queue is full
Pseudo-Code :
If Tail = Queue_Size+1 Then
Return True
Else
Return False

Full **:** True iff no elements can be inserted into the queue.This operation checks whether the queue is overflowing or not.

Size **:** Returns number of elements in the queue.

Pseudo-Code:

Return (Tail - Head)

Display **:** Display the contents of the Queue. This operation displays all the contents of the queue.

Pseudo-Code:

If head < 1 then
Lb :=1;
Else
Lb := Head;
If tail > max_queue_ size + 1 then
Ub := max_queue_size;
Else
Ub := Tail;
For I:= Lb to Ub
Display Queue(I)

4.3 IMPLEMENTATION OF QUEUES USING LINKED LIST

A queue is a first in first out structure (FIFO). It can be visualized as a queue of people waiting for a bus. The person first in the queue is the first on to the bus.

A standard method of implementing a queue is a single linked list. A queue has a 'front' and a 'rear'. New data items are inserted into the rear of the list and deleted from the front of the list.

4.3.1 Operations

IsEmpty() - return 1 if queue is empty; otherwise, return 0.

Pseudo-Code :

```
IsEmpty(Queue q)
{
if (FRONT=Null)
return true;
else
return false;
}
```

Top ( ) - return the value of the first element of the queue.

Pseudo-Code:

```
DATA Top (Queue q)
{
return Front;
}
```

Enqueue(Data): This operation inserts an element at the rear end of the queue.

Steps for inserting an element into a Queue

Inserting into a queue can happen only at the REAR end.

1. Create a new node
2. Make the next of the new node as NULL as it is the last element always
3. If the queue is empty, then make FRONT point to new node.
4. Otherwise make the previous node's next(the node pointed by REAR is always the previous node) point to this node.
5. Make REAR point to new node.

Pseudo-Code:

Definition of a record type for the structure of the queue header:

```
type
queueheader = record
```

info: string
rear: pointer (points to last node in queue or is null if the queue is empty)
front: pointer (points to first node in queue or is null if the queue is empty)
end

Definition of a record type for the structure of a node in the queue:

type
queuenode = record
info: data of some type
nextnode: pointer (to next node in queue or null if this is the last one in the queue)
end

Insert(P: pointer to a new node)

```
P-> nextnode = null      //since it will be last in queue, set nextnode field to null
if que-> rear = null      //queue is empty
que-> front = P         //point front to first and only node
else
(que-> rear)-> nextnode = P    //point formerly last node's nextnode field to new last node P
endif
que-> rear = P          //in either case, point rear to new last node
```

Example:

```
struct queue
{
int no;
struct queue *next;
}*start = NULL;
void add()
{
struct queue *p, *temp;
temp = start;
p = (struct queue *) malloc (sizeof(struct queue));
printf("Enter the data");
scanf("%d", &p->no);
p - next = NULL;
if (start == NULL)
{
start = p;
}
else
{
while(temp->next!=NULL)
```

```
{
temp = temp->next;
}
temp->next = p;
}
}
```

DEQUEUE (): This operation deletes an element from the front end of the queue and returns the value.

Steps for deleting an element from the Queue

Deletions always happen in the FRONT end

1. Store the node pointed by FRONT in a temporary Pointer.
2. Make FRONT as FRONT's next.
3. Delete the node pointed by temporary pointer.
4. If all the nodes are deleted from the Queue make the FRONT and REAR as NULL.

Pseudo-Code:

```
Delete(P: pointer to node removed or null if queue is empty)
if Que-> front = null
P = null                          //queue is empty
else
P = Que-> front                   //assign address of first node in queue to pointer P
Que-> front = P-> nextnode//point que's front pointer to node after node pointed to by P
if que-> front = null             //if there are no more nodes after node pointed to by P
que-> rear = null                 //adjust rear to reflect null also
endif
endif
Return (P)
```

Example:

```
int del()
{
struct queue *temp;
int value;
if (start==NULL)
{
printf("queue is empty");
getch();
return(0);
}
else
{
temp = start;
```

```
value = temp->no;
start = start->next;
free(temp);
}
return (value);
}
```

3.4 APPLICATIONS OF QUEUES

3.4.1 CPU Scheduler

In a multitasking operating system, the CPU time is shared between multiple processes. At a given time, only one process is running, all the others are 'sleeping'. The CPU time is administered by the scheduler. The scheduler keeps all current processes in a queue with the active process at the front of the queue.

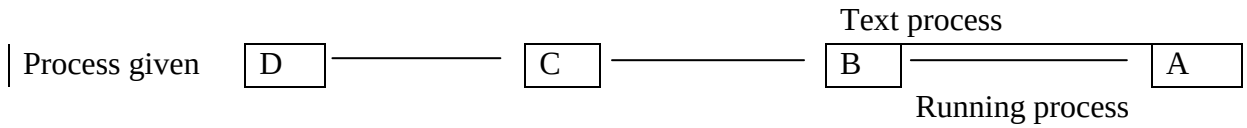


Figure: Scheduler

2.3.4.2 Round-Robin Scheduling

Every process is granted a specific amount of CPU time, its 'quantum'. If the process is still running after its quantum run out, it is suspended and put towards the end of the queue.

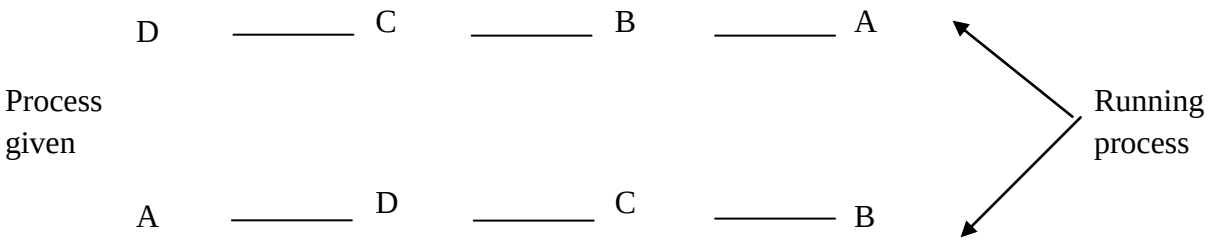


Figure: Round Robin scheduler

3.4.3 Serving Requests

In computing, it often happens that many processes (human users or programs) require the service of a single, shared resource. For example, usually several people share the use of a printer. If requests for printing are submitted faster than they can be satisfied by the printer, the requests are placed on a queue so as to preserve the order in which the requests were submitted. New requests are added at the end of the queue, and, when the printer finishes one request it starts on the request at the front of the queue. Another computing resource that is usually shared by many users is mass-storage devices, such as large disks.

3.4.4 Buffers

The other main use of queues is storing data that is being transferred asynchronously between two processes. Asynchronous means that the sender and receiver are not synchronized ... i.e. that the receiver is not necessarily ready/able to receive the data at the time and speed at which the sender is sending it.
A queue is placed between the two processes: the sender *enqueues* the data whenever it likes, the receiver *serves* out the data whenever *it* likes. Such a queue is sometimes called a buffer.

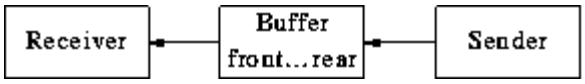


Figure: Buffers

3.4.5 Simulation of an Airport

There is a small busy airport with only one runway. In each unit of time one plane can land or one plane can take off, but not both. Planes arrive ready to land or to take off at random times, so at any given unit of time, the runway may be idle or a plane may be landing or taking off. There may be several planes waiting either to land or to take off. Follow the steps given below to design the program.

1. Create two queues one for the planes landing and the other for planes taking off.
2. Get the maximum number of units <endtime> for which the simulation program would run.
3. Get the expected number of planes arriving in one unit <expectarrive> and number of planes ready to take off in one unit <expectdepart>.
4. To display the statistical data concerning the simulation, declare following data members.
    a. idletime - to store the number of units the runway was idle
    b. landwait - to store total waiting time required for planes landed
    c. nland - to store number of planes landed
    d. nplanes - to store number of planes processed
    e. nrefuse - to store number of planes refused to land on airport
    f. ntakeoff - to store number of planes taken off
    g. takeoffwait - to store total waiting time taken for take off

Initialize the queue used for the plane landing and for the take off
Get the data for <endtime>, <expectarrive> and <expectdepart> from the user.

The process of simulation would run for many units of time, hence run a loop in main( ) that would run from <curtime> to <endtime> where <curtime> would be 1 and <endtime> would be the maximum number of units the program has to be run.

Generate a random number. Depending on the value of random number generated, perform following tasks.

1. If the random number is less than or equal to 1 then get data for the plane ready to land. Check whether or not the queue for landing of planes is full. If the queue is full then refuse the plane to land. If the queue is not empty then add the data to the queue maintained for planes landing.
2. If the random number generated is zero, then generate a random number again. Check if this number is less than or equal to 1. If it is , then get data for the plane ready to take off. Check whether or not the queue for taking a plane off is full. If the queue is full then refuse the plane to take off otherwise add the data to the queue maintained for planes taking off.
3. It is better to keep a plane waiting on the ground than in the air, hence allow a plane to take off only, if there are no planes waiting to land.
4. After receiving a request from new plane to land or take off, check the queue of planes waiting to land, and only if the landing queue is empty, allow a plane to take off.
5. If the queue for planes landing is not empty then remove the data of plane in the queue else run the procedure to land the plane.
6. Similarly, if the queue for planes taking off is not empty then remove the data of plane in the queue else run the procedure to take off the plane.
7. If both the queues are empty then the runway would be idle.
8. Finally, display the statistical data As given below.
    - Total number of planes processed
    - Number of planes landed :
    - Number of planes taken off :
    - Number of planes refused use :
    - Number of planes left ready to land :
    - Number of planes left ready to take off :
    - Percentage of time the runway was idle :
    - Average wait time to land :
    - Average wait time to take off :

3.4.6 Dequeue

A dequeue is a queue in which insertions and deletions can happen at both ends of. A dequeue ordouble ended queue is a data structure, which unites the properties of a queue and a stack. Like the stack, items can be pushed into the dequeue, once inserted into the dequeue the last item pushed in may be extractd from one side (popped, as a stack) and the first item pushed in may be pulled out of the other side(as in a queue)

In an input restricted deque the insertion of elements is at one end only, but the deletion of elements can be done at both the ends of a queue.In an output restricted deque, the deletion of

elements is done at one end only, and allows insertion to be done at both the ends of a dequeue

3.4.7 Priority Queue

A Priority queue is one in which each element will have a priority associated with it. The element with the highest priority is the one that will be processed /deleted first. If two or more nodes have the same priority then they will be processed in the same order as they were entered in to the queue.

3.5 LET US SUM UP

A Queue is an ordered collection of items from which items may be deleted at one end (called the front of the queue) and into which items may be inserted at the other end (the rear of the queue). It can be implemented using Arrays as well as Linked list.

3.6 QUESTIONS FOR DISCUSSION

1. What are the two pointers involved in a queue.
2. Define Init Queue.
3. How enqueue and dequeue happens in an array implementation of a queue?
4. Define some applications of Queue.

3.7 SUGGESTIOIN FOR BOOKS READING

BLOCK III

BLOCK III TREES

This lesson takes you to know the need of non-linear data structure. Non-linear data structure is a structure in which data's are not stored in a sequential order. This non-linear representation not only reduces the searching time as well as it maintains the sorted list of data. Trees and Graphs are examples of Non linear data structure. We are going to see the trees in detail in this section.

This block consists of the following:

Unit 1: Binary Trees
Unit 2: AVL Trees
Unit 3: Tree Traversals and Hashing
Unit 4: Simple implementations of Tree

UNIT
1
BINARY TREES

STRUCTURE

1.0 AIMS AND OBJECTIVES

At the end of this lesson, students should be able to demonstrate appropriate skills, and show an understanding of the following:

- Aims and objectives of Binary Trees
- Nodes of a binary tree
- Binary tree representation
- Binary tree traversal

1.1 INTRODUCTION OF BINARY TREES

A binary tree is a finite set of nodes that is either empty or it consists of a root and two disjoint binary trees called the left sub tree and the right sub tree.

A **binary tree** is a **rooted** tree in which every node has at most two children. A **full binary tree** is a tree in which every node has zero or two children. Also known as a **proper binary tree**. A **perfect binary tree** is a full binary tree in which all **leaves** (vertices with zero children) are at the same **depth** (distance from the **root**, also called **height**).
Sometimes the **perfect binary tree** is called the **complete binary tree**. An **almost complete binary tree** is a tree where for a right child, there is always a left child, but for a left child there may not be a right child.

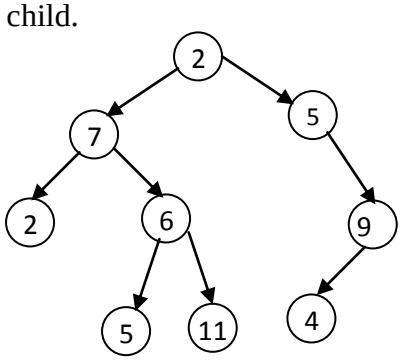
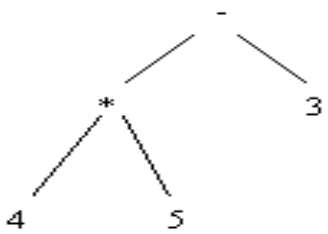


Figure: Binary Tree

A **binary tree** is similar to a tree, but not quite the same. For a binary tree, each node can have zero, one, or two children. In addition, each child node is clearly identified as either the left child or the right child.

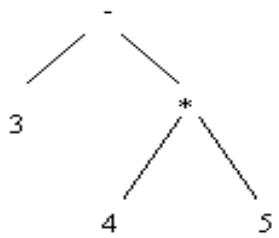
As an example, here you can see the **binary expression tree** for the expression 4 * 5 - 3. Note that a child drawn to the left of its parent is meant to be the left child and that a child drawn to the right of its parent is meant to be the right child.



Expression: 4*5-3

Figure: Binary Expression Tree

Reversing the left to right order of any siblings gives a different binary tree. For example, reversing the sub trees rooted at * and at 3 gives the following different binary tree.
It happens to be the binary expression tree for 3 - 4 * 5.



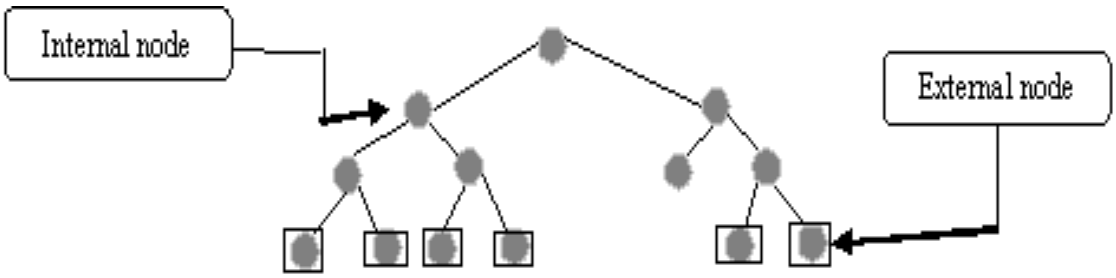
Expression: 3-4*5

Figure: Binary Expression Tree

1.2 BINARY TREES – EXTERNAL AND INTERNAL NODES

In a binary tree, all the nodes must have the same number of children. All internal nodes have two children and external nodes have no children. A binary tree with n internal nodes has n + 1 external nodes. Let s1, s2, …, sn be the internal nodes, in order. Then,

key(s1) < key(s2) < … < key(sn).



- Internal nodes - External nodes

Figure: Internal and External node representation

Minimum number of nodes in a binary tree whose height h is **h+1** and maximum number of nodes is about $\mathbf{2^{h+1} - 1}$. A binary tree with N nodes (internal and external) has N-1 edges. A binary tree with N internal nodes has N+ 1 external node.

1.3 BINARY TREE REPRESENTATIONS

A full binary tree of depth k is a binary tree of depth k having 2^k-1 nodes. This is the maximum number of the nodes such a binary tree can have. A very elegant sequential representation for such binary trees results from sequentially numbering the nodes, starting with nodes on level 1, then those on level 2 and so on. Nodes on any level are numbered from left to right. This numbering scheme gives us the definition of a complete binary tree. A binary tree with n nodes and a depth k is complete iff its nodes correspond to the nodes which

are numbered one to n in the full binary tree of depth k. The nodes may be represented in an array or using a linked list.

1.3.1 Array Representation of Trees

This method is easy to understand and implement. It's very useful for certain kinds of tree applications, such as heaps, and fairly useless for others.
Steps to implement binary trees using arrays:

- Take a complete binary tree and number its nodes from top to bottom, left to right.
- The root is 0, the left child 1, the right child 2, the left child of the left child 3, etc.
- Put the data for node i of this tree in the ith element of an Array.
- If you have a partial (incomplete) binary tree, and node i is absent, put some value that represents "no data" in the ith position of the array.

Three simple formulae allow you to go from the index of the parent to the index of its children and vice versa:

- if index(parent) = N, index(left child) = 2*N+1
- if index(parent) = N, index(right child) = 2*N+2
- if index(child) = N, index(parent) = (N-1)/2 (integer division with truncation)

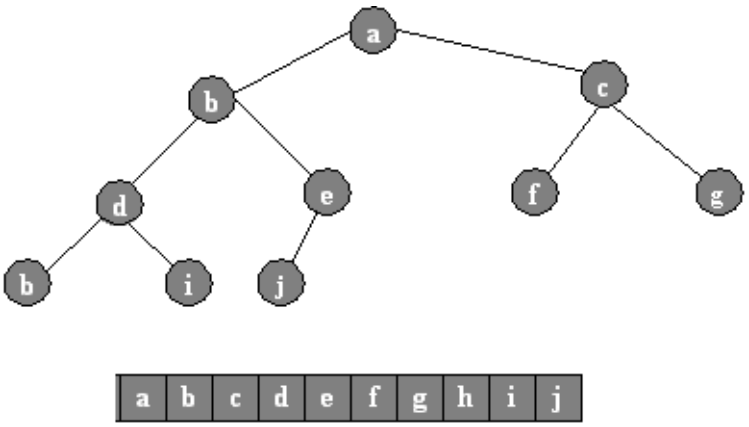


Figure: Array Representation of Trees

Advantages of linear representation:

1. Simplicity.
2. Given the location of the child (say, k), the location of the parent is easy to determine (k / 2).

Disadvantages of linear representation:

1. Additions and deletions of nodes are inefficient, because of the data movements in the array.
2. Space is wasted if the binary tree is not complete. That is, the linear representation is useful if the number of missing nodes is small.

1.3.2 Linked Representation of Trees

For a linked representation, a node in the tree has

- a data field
- a left child field with a pointer to another tree node
- a right child field with a pointer to another tree node
- optionally, a parent field with a pointer to the parent node

The most important thing you must remember about the linked representation is that a tree is represented by the pointer to the root node, not a node. The empty tree is simply the NULL pointer, not an empty node.

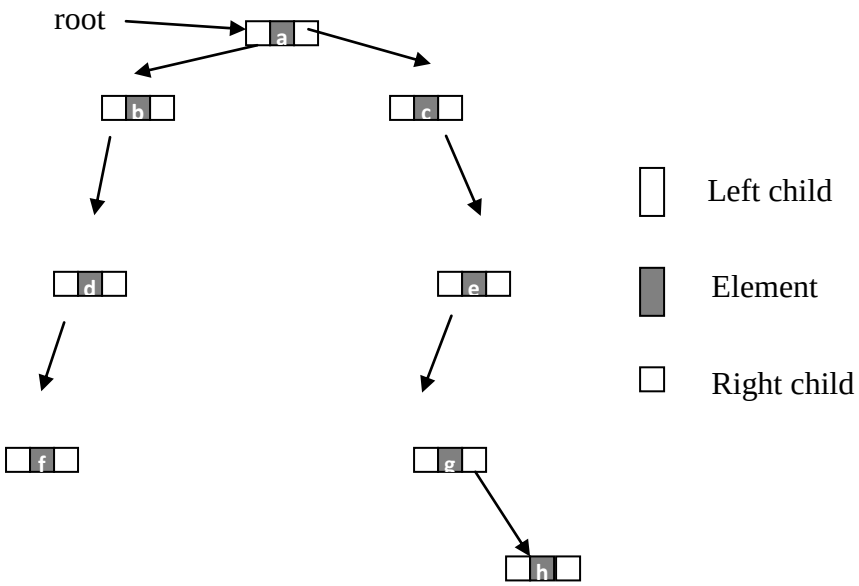


Figure: Linked Representation of Trees

Nodes consisting of w data field and two pointers: a pointer to the first child, and a pointer to the next sibling.

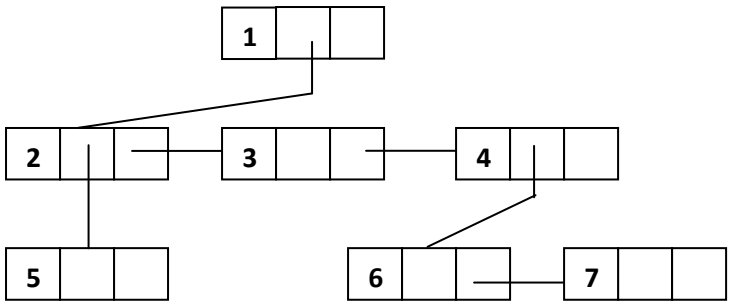


Figure: Linked Trees

1.3 BINARY TREE TRAVERSAL

There are many operations that we often want to perform on trees. One notion that arises frequently is the idea of traversing a tree or visiting each node in the tree exactly once. Traversing a tree means visiting all the nodes of a tree in order. A full traversal produces a linear order for the information in a tree. This linear order may be familiar and useful. If we traverse the standard ordered binary tree *in-order*, then we will visit all the nodes in sorted order.

Types of Traversals are:

- Preorder traversal
    1. Visit the root
    2. Traverse the left sub tree
    3. Traverse the right sub tree

- In order traversal
    1. Traverse the left sub tree
    2. Visit the root
    3. Traverse the right sub tree

1. Post order traversal
    1. Traverse the left subtree
    2. Traverse the right subtree
    3. Visit the root

1.3.1 Pre-order traversal

1. Start at the root node
2. Traverse the left subtree
3. Traverse the right subtree

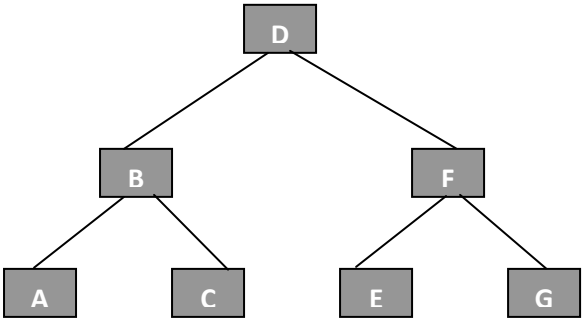


Figure: Preorder Traversal

The nodes of this tree would be visited in the order: **D B A C F E G**

Procedure for Pre-Order Traversal

Preorder (current node: tree pointer);

{Current node is a pointer to a node in a binary tree. For full
Tree traversal, pass preorder, the pointer to the top of the tree}

{//preorder

```
If current node <> nil then
{
Write (current node ^.data);

Preorder (current node ^.left child);

Preorder (current node ^.right child);
 }
}// preorder
```

1.3.2 In-order traversal

1. Traverse the left sub tree
2. Visit the root node
3. Traverse the right sub tree

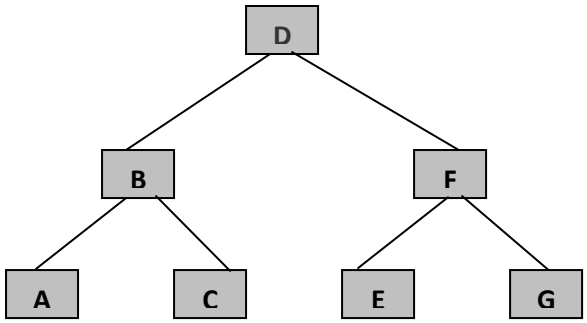


Figure: In-Order Traversal

The nodes of this tree would be visited in the order: **A B C D E F G**

Procedure for In Order Traversal:

```
In order (currentnode: treepointer);
{Current node is a pointer to a node in a binary tree. For full

Tree traversal, pass in order, the pointer to the top of the tree}
{// in order
If current node <> nil
Then
{
In order (current node ^. left child);
Write (currentnode^.data);
In order (currentnode^.rightchild);
}
}// in order
```

1.3.3 Post-order traversal

1. Traverse the left subtree
2. Traverse the right subtree
3. Visit the root node

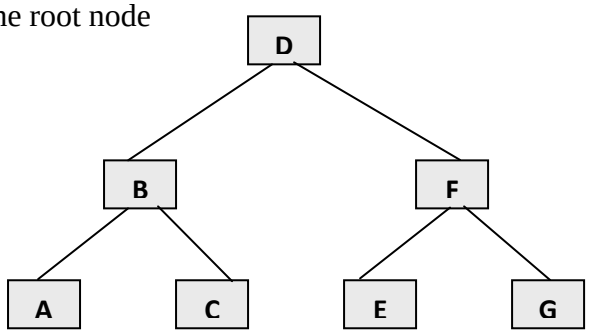


Figure: Post order Traversal

The nodes of this tree would be visited in the order: **A C B E G F D**

Procedure for Post Order Traversal

```
Post order (current node: tree pointer);
{Current node is a pointer to a node in a binary tree. For full
Tree traversal, pass post order, the pointer to the top of the tree}
{// post order
If current node<> nil then
{
Post order (current node ^.left child);
Post order (currentnode^.rightchild);
Write (currentnode^.data);
}
}//post order
```

1.4 BINARY SEARCH TREES

1.4.1 Introduction of Binary Trees

A **binary search tree** is a binary tree in which the data in the nodes are ordered in a particular way. To be precise, starting at any given node, the data in any nodes of its left sub tree must all be less than the item in the given node, and the data in any nodes of its right sub tree must be greater than or equal to the data in the given node. For numbers this can obviously be done. For strings, alphabetical ordering is often used. For records of data, a comparison based on a particular field (the **key field**) is often used. An example of a binary search tree is shown in Figure.

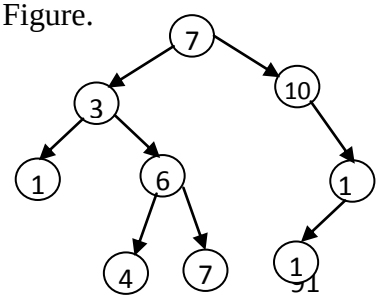


Figure: Binary Search Tree

The above figure shows a binary search tree of size 9 and depth 3, with root 7 and leaves 1, 4, 7 and 13. Every node (object) in a binary tree contains information divided into two parts. The first one is proper to the structure of the tree, that is, it contains a key field (the part of information used to order the elements), a parent field, a left child field, and a right child field. The second part is the object data itself. It can be endogenous (that is, data resides inside the tree) or exogenous (that is nodes only contains a references to the object's data). The root node of the tree has its parent field set to null. Whenever a node does not have a right child or a left child, then the corresponding field is set to null.

A binary search tree is a binary tree with more constraints. If x is a node with key value key [x] and it is not the root of the tree, then the node can have a left child (denoted by left [x]), a right child (right [x]) and a parent (p [x]). Every node of a tree possesses the following *Binary* **Search Tree properties:**

1. For all nodes y in left sub tree of x, key [y] < key [x]
2. For all nodes y in right sub tree of x, key[y] > key [x]

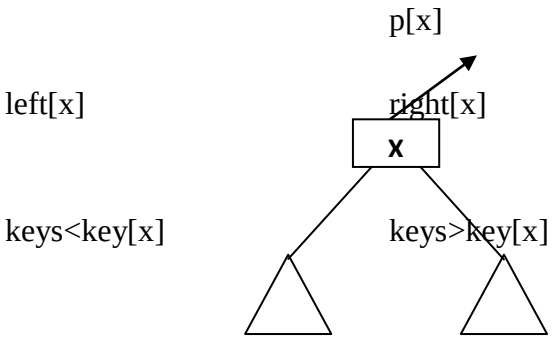


Figure: Keys of Binary Search Tree

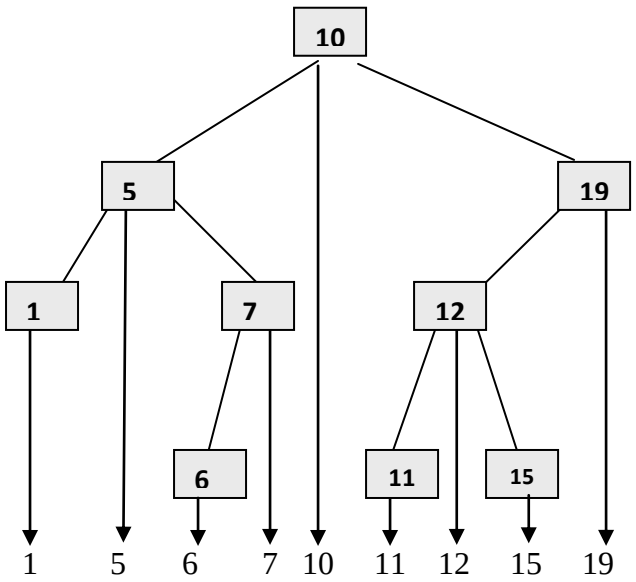


Figure: Binary Search Tree

2.2 BINARY SEARCH TREE - OPERATIONS

2.2.1 Finding Minimum and Maximum

The minimum element of a binary search tree is the last node of the left roof, and its maximum element is the last node of the right roof. Therefore, we can find the minimum and the maximum by tracking on the left child and the right child, respectively, until an empty sub tree is reached.

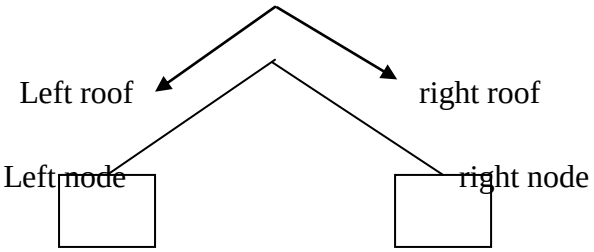


Figure: Left and Right Nodes

2.2.2 Searching in a Binary Search Tree

You can search for a desirable value in a Binary search tree through the following procedure.

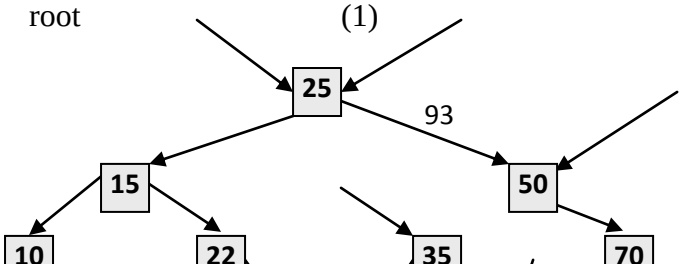
Search for a matching node

1. Start at the root node as current node
2. If the search key's value matches the current node's key then found a match
3. If search key's value is greater than current node's key
    1. If the current node has a right child, search right
    2. Else, no matching node in the tree
4. If search key is less than the current node's key
    1. If the current node has a left child, search left
    2. Else, no matching node in the tree

Example:

Search for 45 in the tree:

1. Start at the root, 45 is greater than 25, search in right sub tree
2. 45 is less than 50, search in 50's left sub tree
3. 45 is greater than 35, search in 35's right sub tree
4. 45 is greater than 44, but 44 has no right sub tree so 45 is not in the BST



(2)
(3)
(4)

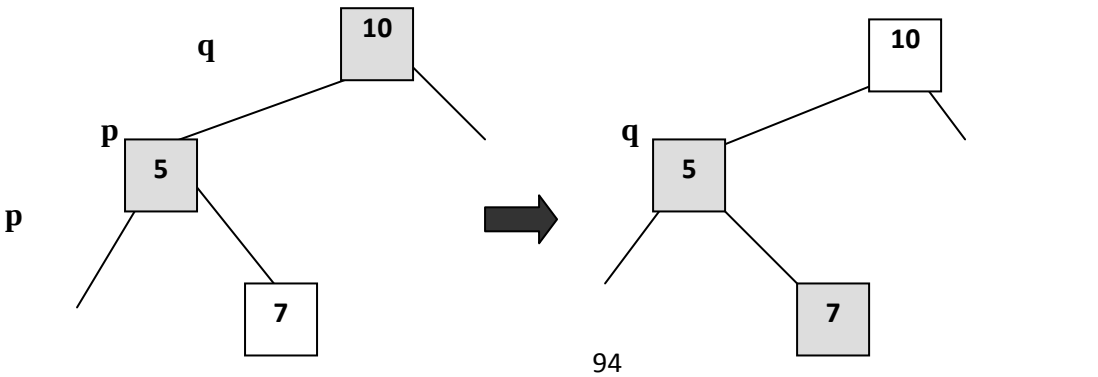
Figure: Searching in a binary search tree

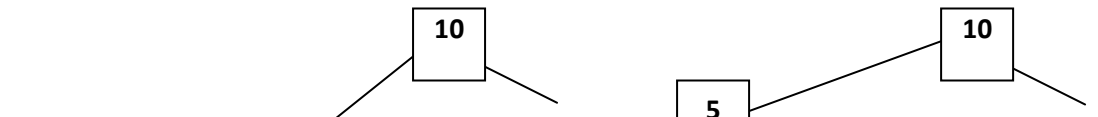
TREE-SEARCH (*x*, *k*)
if *x* = NIL. OR. *k* = key[*x*]
then return *x*
if (*k* < key[*x*])
then
// Search Left Tree

return TREE-SEARCH (left[*x*], *k*)
// Search Right Tree
else **return TREE-SEARCH (right[*x*], *k*)**

3.2.2.3 Inserting an element

Both insertion and deletion operations cause changes in the data structure of the dynamic set represented by a binary search tree. For a standard insertion operation, we only need to search for the proper position that the element should be put in and replace NIL by the element. If there are n elements in the binary search tree, then the tree contains (n+1) NIL pointers. Therefore, for the (n+1)th element to be inserted, there are (n+1) possible places available. In the insertion algorithm, we start by using two pointers: p and q, where q is always the parent of p. Starting from the top of the tree, the two pointers are moved following the algorithm of searching the element to be inserted. Consequently, we end with p=NIL (assuming that the key is not already in the tree). The insertion operation is then finished by replacing the NIL key with the element to be inserted





q **q**
p **p**

Figure: Insertion in a Binary Search Tree

We can insert a node in the binary search tree using the following procedure:

1. Always insert new node as leaf node
2. Start at root node as current node
3. If new node's key < current's key
    1. If current node has a left child, search left
    2. Else add new node as current's left child
4. If new node's key > current's key
    1. If current node has a right child, search right
    2. Else add new node as current's right child

TREE-INSERT (T, z)

y ← NIL
x ← root [T] // Assigning root of tree
while x ≠ NIL do
y ← x
if key [z] < key[x]
then x ← left[x] // Traversing left tree
else x ← right[x] // Traversing right tree
p[z] ← y
if y = NIL
then root [T] ← z
else if key [z] < key[y]
then left [y] ← z // Inserting as left child
else right [y] ← z // Inserting as right child

3.2.2.4 Deleting an element

Deleting an item from a binary search tree is little harder than inserting one. Before writing a code, let's consider how to delete nodes from a binary search tree in an abstract fashion. Here's a BST from which we can draw examples during the discussion:

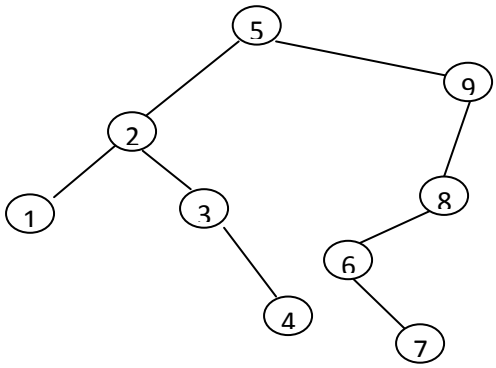


Figure: BST Deletion

It is more difficult to remove some nodes from this tree than to remove others. Here, let us see three distinct cases, described in detail below in terms of the deletion of a node designated *p*.

Case 1: *p* has no right child

It is trivial to delete a node with no right child, such as node 1, 4, 7, or 8 above. We replace the pointer leading to *p* by *p*'s left child, if it has one, or by a null pointer, if not. In other words, we replace the deleted node by its left child. For example, the process of deleting node 8 looks like this:

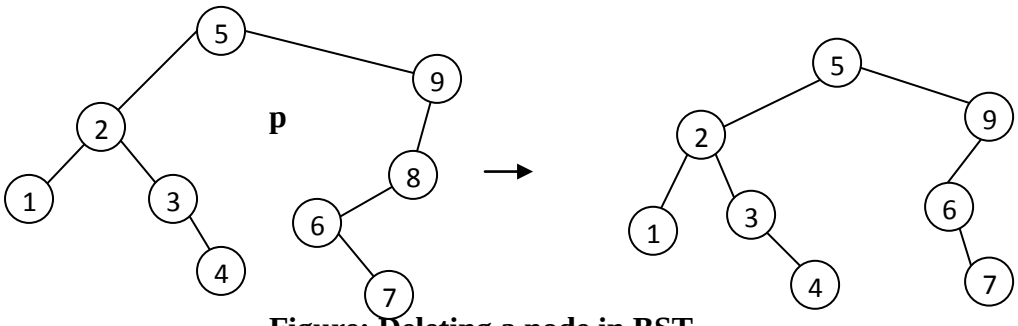
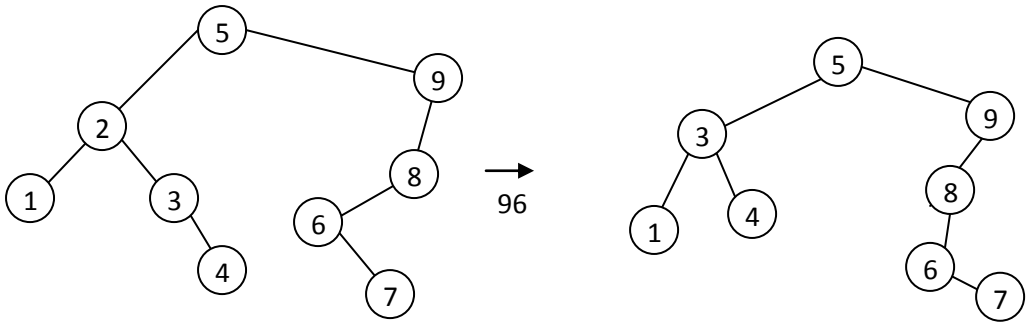


Figure: Deleting a node in BST

Case 2: p's right child has no left child

This case deletes any node *p* with a right child *r* that itself has no left child. Nodes 2, 3, and 6 in the tree above are examples. In this case, we move *r* into *p*'s place, attaching *p*'s former left subtree, if any, as the new left subtree of *r*. For instance, to delete node 2 in the tree above, we can replace it by its right child 3, giving node 2's left child 1 to node 3 as its new left child. The process looks like this:



p **r**

Figure: Deleting a node in BST

Case 3: p's right child has a left child

This is the "hard" case, where p's right child r has a left child. But if we approach it properly we can make it make sense. *Let p's* in order successor that is, the node with the smallest value greater than p, be s.

Then, our strategy is to detach s from its position in the tree, which is always an easy thing to do, and put it into the spot formerly occupied by p, which disappears from the tree. In our example, to delete node 5, we move inorder successor node 6 into its place, like this:

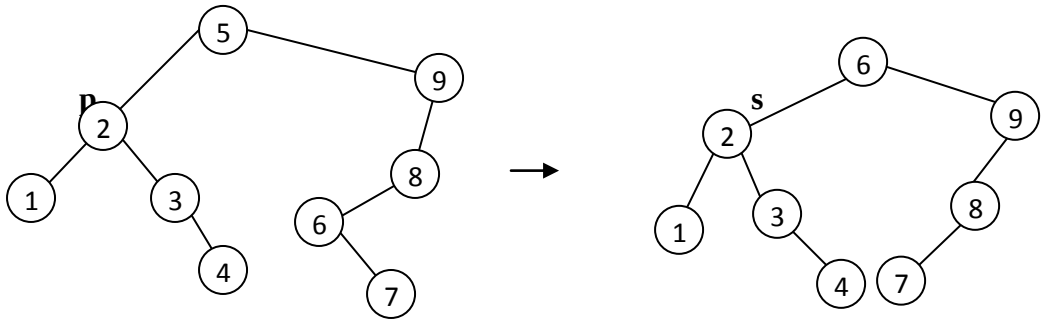


Figure: Deleting a node in BST

But how to know that node *s* exists and that we can delete it easily? We know that it exists because otherwise, this would be case 1 or case 2 (consider their conditions). We can easily detach from its position for a more subtle reason: *s* is the inorder successor of *p* and is therefore has the smallest value in *p*'s right subtree, so *s* cannot have a left child. (If it did, then this left child would have a smaller value than *s*, so it, rather than *s*, would be *p*'s inorder successor.) Because *s* doesn't have a left child, we can simply replace it by its right child, if any. This is the mirror image of case 1.

TREE-DELETE (T, z)

if left [z] = NIL .OR. right[z] = NIL
then y ← z
else y ← TREE-SUCCESSOR (z)
if left [y] ≠ NIL
then x ← left[y] // Assigning left node if it exists
else x ← right [y] // Assigning right node if it exists
if x ≠ NIL
then p[x] ← p[y]
if p[y] = NIL

then root [T] ← x
else if y = left [p[y]]
then left [p[y]] ← x
else right [p[y]] ← x
if y ≠ z
then key [z] ← key [y]
if y has other field, copy them, too return y

Tree-Successor(*x*)

if $right[x] \neq NIL$
then return Tree-Minimum(*right*[*x*])
$y \leftarrow p[x]$
while $(y \neq NIL)$ and (x = *right*[y])
do $x \leftarrow y$
$y \leftarrow p[y]$
return *y*

1.4 LET US SUM UP

Non-linear data structure is a structure in which data's are not stored in a sequential order. Trees and Graphs are examples of Non linear data structure. A tree comprises of an arrangement of *nodes* each of which holds information. Nodes are linked by *arcs* (or *edges*). Each node has zero or more **child nodes**, which are below it in the tree (by convention in computer science, trees grow down - not up as they do in nature). A **child** is a node connected directly below the starting node. Nodes with the same parent are called **siblings**. The topmost node in a tree is called the **root node**. A **branch** is a sequence of nodes such that the first is the parent of the second; the second is the parent of the third, etc. The **leaves** of a tree (sometimes also called **external nodes**) are those nodes with no children. The other nodes of the tree are called **non-leaves** (or sometimes **internal nodes**). The **height** of a tree is the maximum length of a branch from the root to a leaf.

A **binary tree** is a **rooted** tree in which every node has at most two children. A **full binary tree** is a tree in which every node has zero or two children. Also known as a **proper binary tree.** A binary tree with N nodes (internal and external) has N-1 edges. A binary tree with N internal nodes has N+ 1 external node. Binary trees can be represented in array representation as well as linked representation. Binary trees can be traversed in preorder, in order and post order traversals.

A **binary search tree** is a binary tree in which the data in the nodes are ordered in a particular way. Insertion, Deletion and Search operations can be performed in a binary search tree. Huffman compression belongs into a family of algorithms with a variable codeword length. That means that individual symbols (characters in a text file for instance) are replaced by bit sequences that have a distinct length. So symbols that occur often in a file are given a short sequence while that are seldom used get a longer bit sequence. Applications of trees would be dictionaries, translators and so on.

1.5 QUESTIONS FOR DISCUSSION

1. What are the various types of traversals in binary trees?
2. How post order traversal differs from preorder?
3. What is a perfect binary tree?
4. Define binary expression tree.
5. Differentiate internal and external nodes of a binary tree.
6. State the advantages and disadvantages of a linear representation.
7. How do you represent a tree in a linked representation?
8. How a node can be searched in a binary search tree?
9. Construct a binary search tree containing the nodes 21,45,78,52,14,8,12.
10. What are the three cases to be considered while deleting a node in a binary search tree?

1.6 SUGGESTION FOR BOOKS READING

UNIT
2
AVL Trees

STRUCTURE

2.0 AIMS AND OBJECTIVES

At the end of this lesson, students should be able to demonstrate appropriate skills, and show an understanding of the following:

- Aims and objectives of Binary Search Trees
- Operations of Binary search trees
- Binary search tree representation

2.1 INTRODUCTION OF BINARY TREES

2.2 AVL TREES

Balanced binary tree

- The disadvantage of a binary search tree is that its height can be as large as N-1
- This means that the time needed to perform insertion and deletion and many other operations can be O(N) in the worst case
- We want a tree with small height
- A binary tree with N node has height at least $\Theta(\log N)$
- Thus, our goal is to keep the height of a binary search tree O(log N)
- Such trees are called balanced binary search trees. Examples are AVL tree, red-black tree.

AVL tree

Height of a node

- The height of a leaf is 1. The height of a null pointer is zero.
- The height of an internal node is the maximum height of its children plus 1

Note that this definition of height is different from the one we defined previously (we defined the height of a leaf as zero previously).

- An AVL tree is a binary search tree in which
- for *every* node in the tree, the height of the left and right subtrees differ by at most 1.

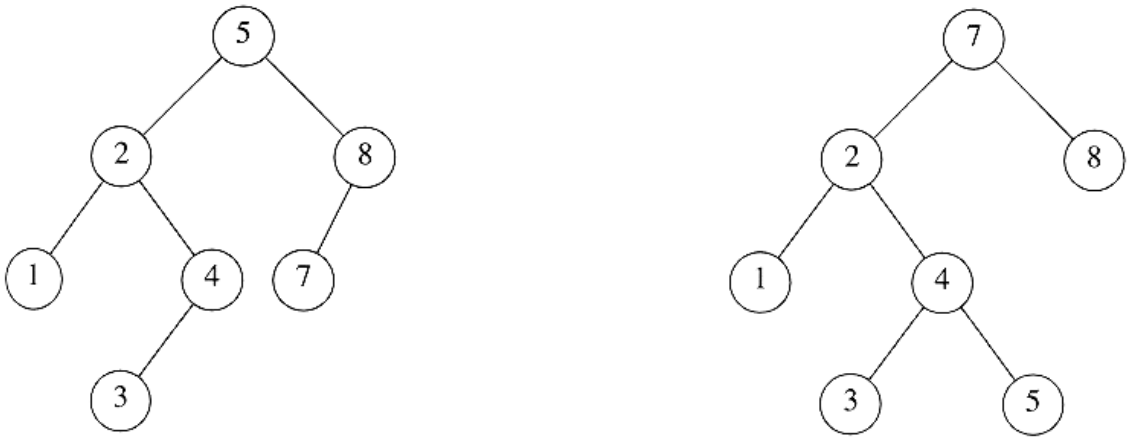


Figure 4.32 Two binary search trees. Only the left tree is AVL.

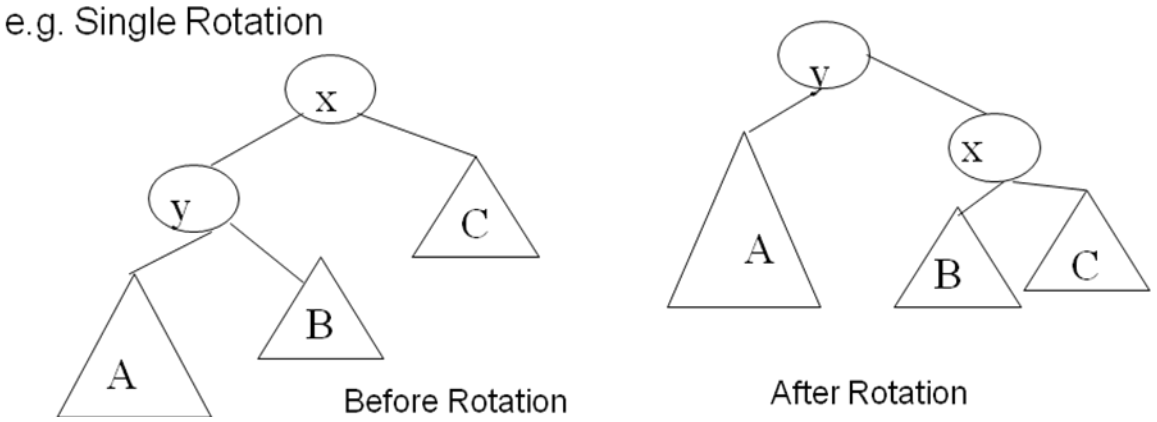
- Let x be the root of an AVL tree of height h
- Let N_h denote the minimum number of nodes in an AVL tree of height h
- Clearly, $N_i \geq N_{i-1}$ by definition
- We have

$$N_h \geq N_{h-1} + N_{h-2} + 1$$
$$\geq 2N_{h-2} + 1$$
$$> 2N_{h-2}$$

- By repeated substitution, we obtain the general form
- $N_h > 2^i N_{h-2}$
- The boundary conditions are: $N_1=1$ and $N_2 =2$. This implies that $h = O(\log N_h)$.
- Thus, many operations (searching, insertion, deletion) on an AVL tree will take O(log N) time.

Rotations

- When the tree structure changes (e.g., insertion or deletion), we need to transform the tree to restore the AVL tree property.
- This is done using single rotations or double rotations.



- Since an insertion/deletion involves adding/deleting a single node, this can only increase/decrease the height of some subtree by 1
- Thus, if the AVL tree property is violated at a node x, it means that the heights of left(x) ad right(x) differ by exactly 2.
- Rotations will be applied to x to restore the AVL tree property.

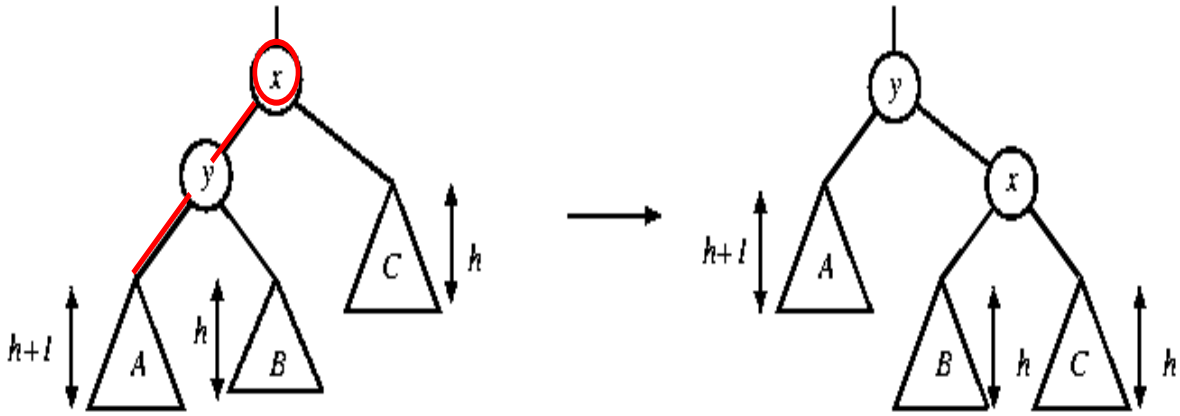
Insertion

- First, insert the new key as a new leaf just as in ordinary binary search tree
- Then trace the path from the new leaf towards the root. For each node x encountered, check if heights of left(x) and right(x) differ by at most 1.
- If yes, proceed to parent(x). If not, restructure by doing either a single rotation or a double rotation [next slide].
- For insertion, once we perform a rotation at a node x, we won't need to perform any rotation at any ancestor of x.

- Let x be the node at which left(x) and right(x) differ by more than 1
- Assume that the height of x is h+3
- There are 4 cases
    - Height of left(x) is h+2 (i.e. height of right(x) is h)
        - Height of left(left(x)) is h+1 ⇒ single rotate with left child
        - Height of right(left(x)) is h+1 ⇒ double rotate with left child
    - Height of right(x) is h+2 (i.e. height of left(x) is h)
        - Height of right(right(x)) is h+1 ⇒ single rotate with right child
        - Height of left(right(x)) is h+1 ⇒ double rotate with right child
- **Note:** Our test conditions for the 4 cases are different from the code shown in the textbook. These conditions allow a uniform treatment between insertion and deletion.

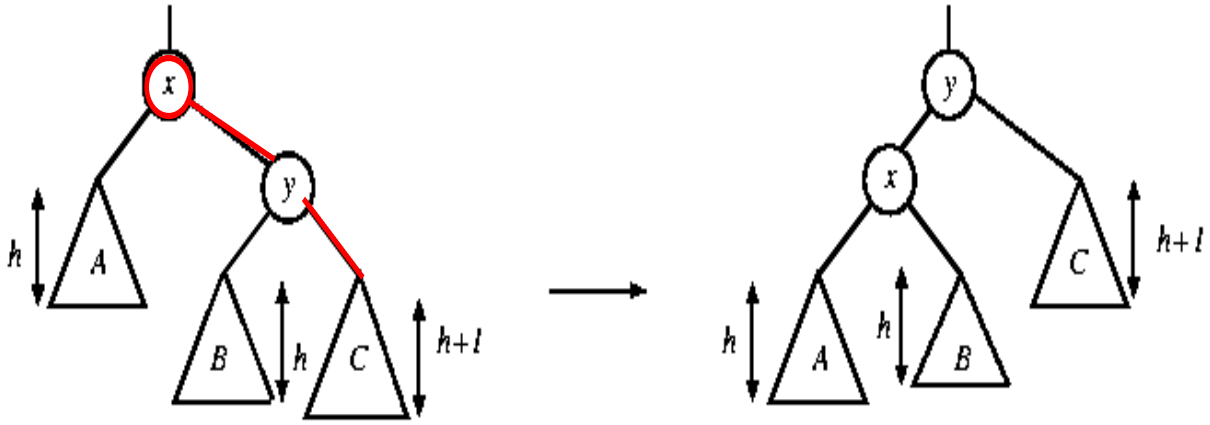
Single rotation

- The new key is inserted in the subtree A.
- The AVL-property is violated at x
    - height of left(x) is h+2
    - height of right(x) is h.



Rotate with left child

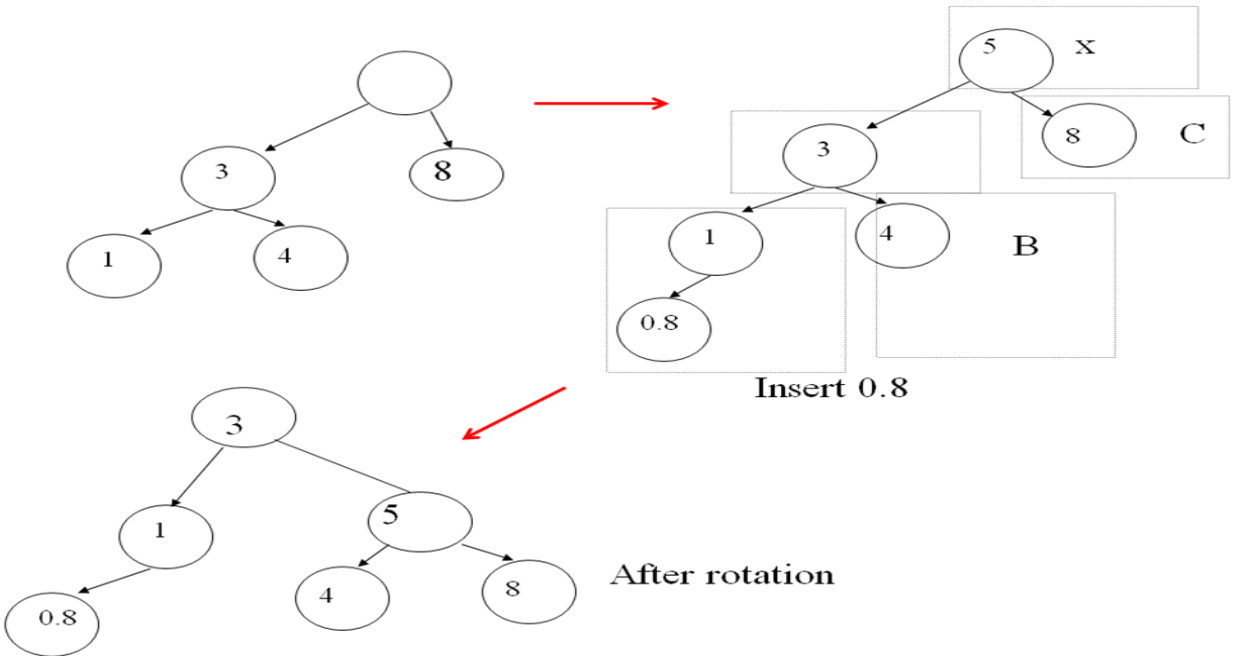
The new key is inserted in the subtree C.
The AVL-property is violated at x.



Rotate with right child

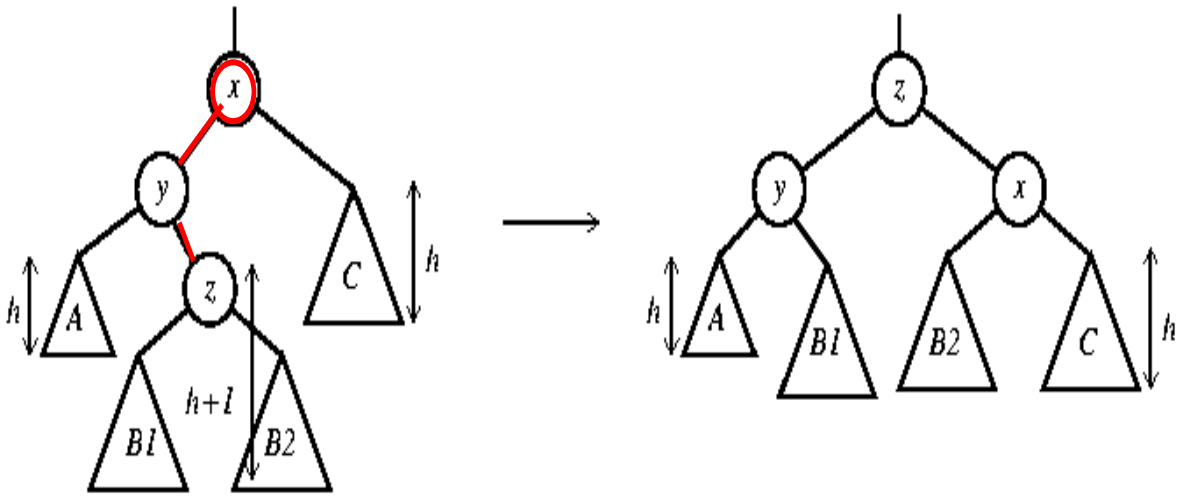
Single rotation takes O (1) time.
Insertion takes O (log N) time.

Example AVL Tree:



Double rotation

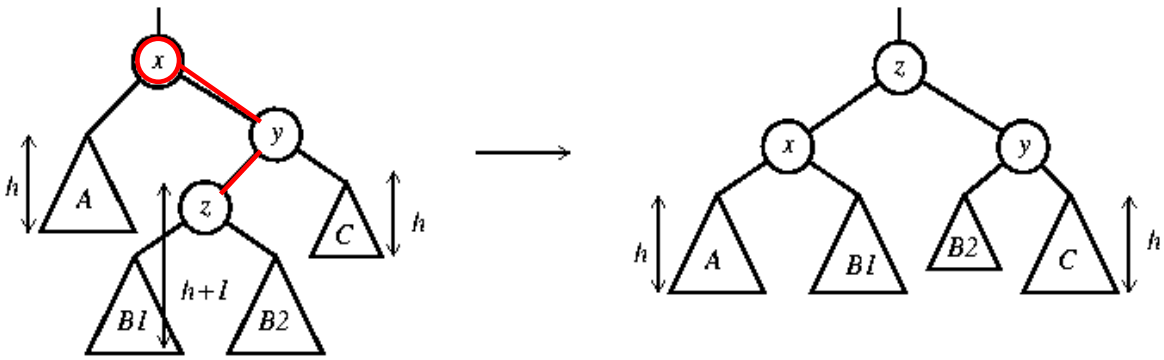
The new key is inserted in the subtree B1 or B2.
The AVL-property is violated at x.
x-y-z forms a zig-zag shape



Double rotate with left child

Also called left-right rotate.

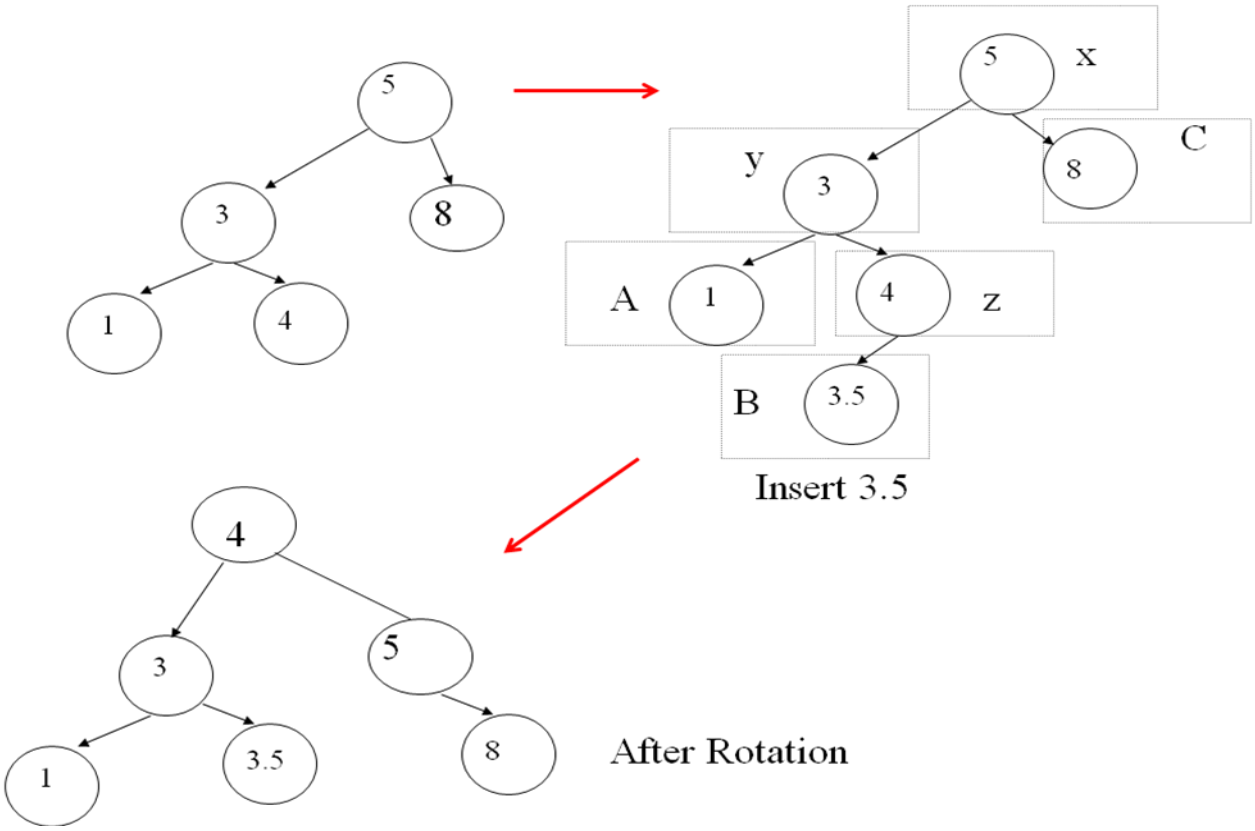
The new key is inserted in the sub tree B1 or B2.
The AVL-property is violated at x.



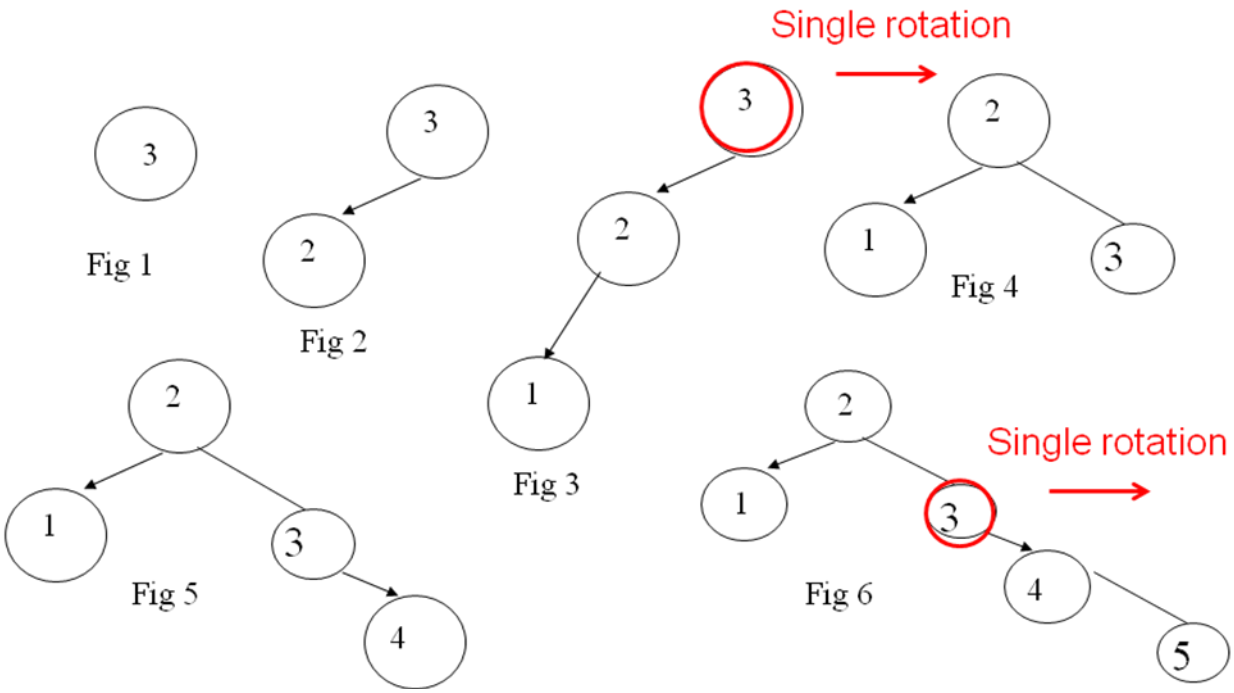
Double rotate with right child

also called right-left rotate.

AVL Tree:



An Extended Example:
Insert 3,2,1,4,5,6,7, 16,15,14



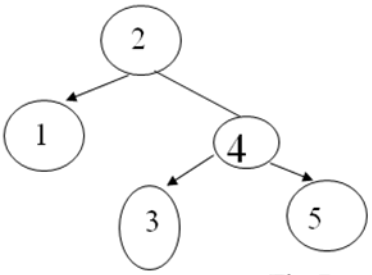
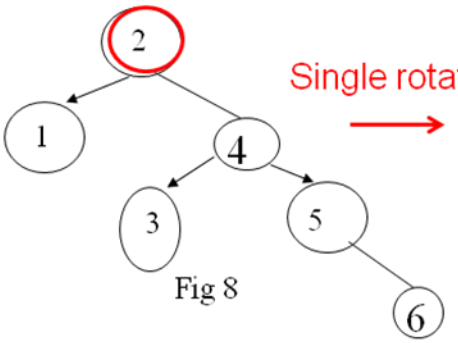


Fig 7



Single rotation

Fig 8

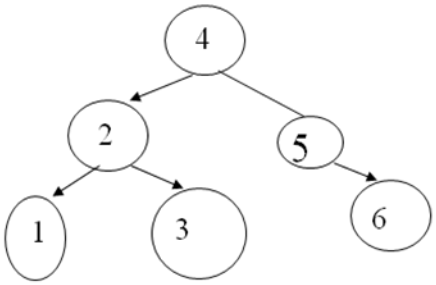
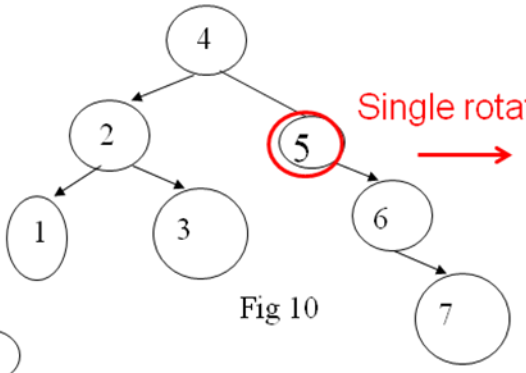


Fig 9



Single rotation

Fig 10

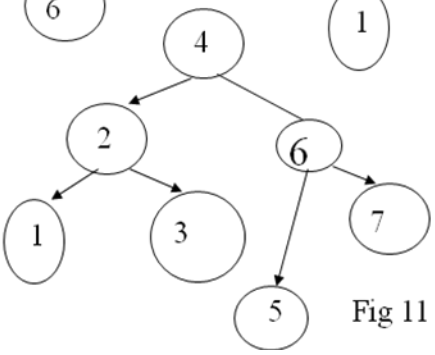


Fig 11

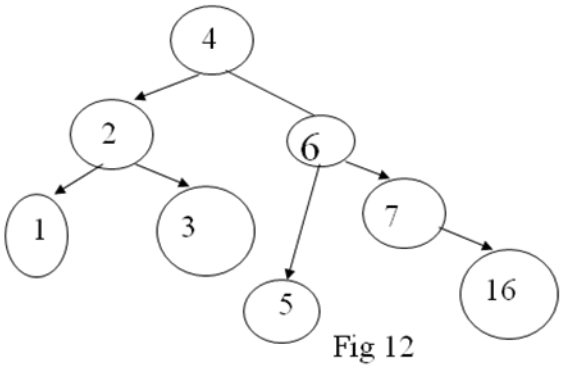


Fig 12

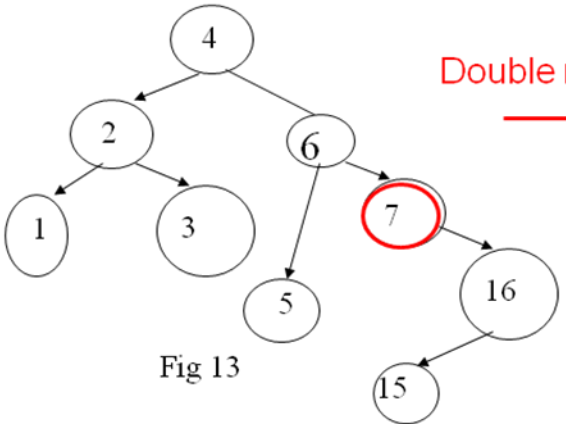


Fig 13

Double rotation

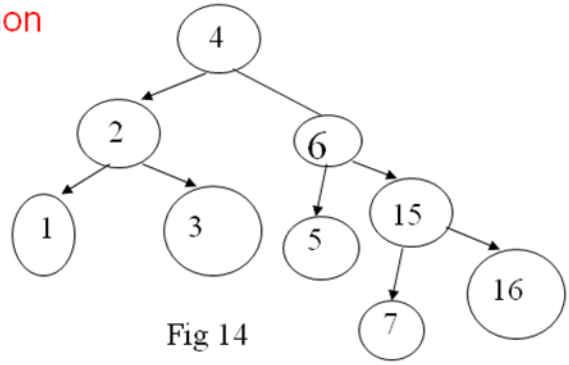


Fig 14

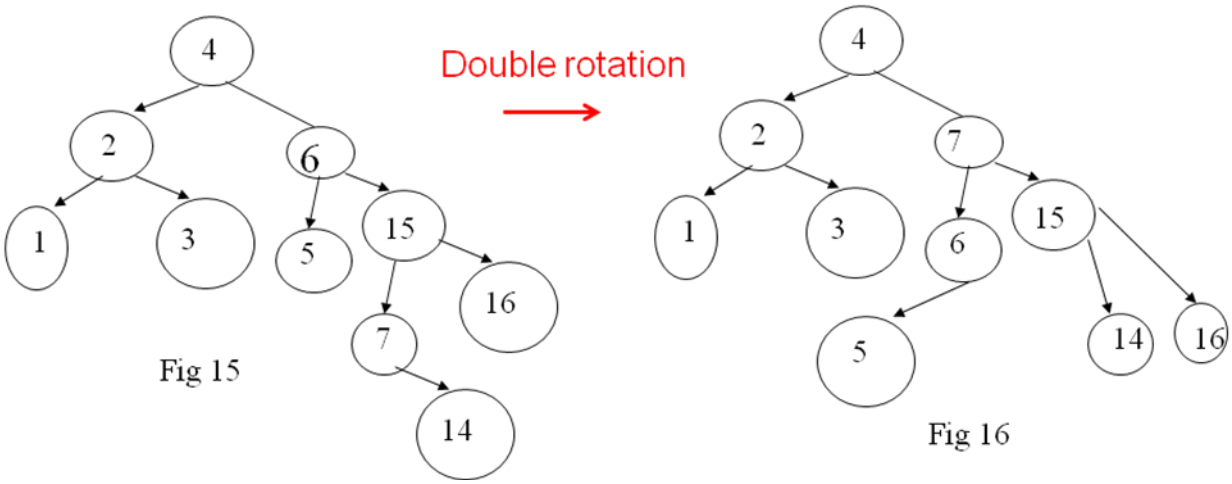


Fig 15

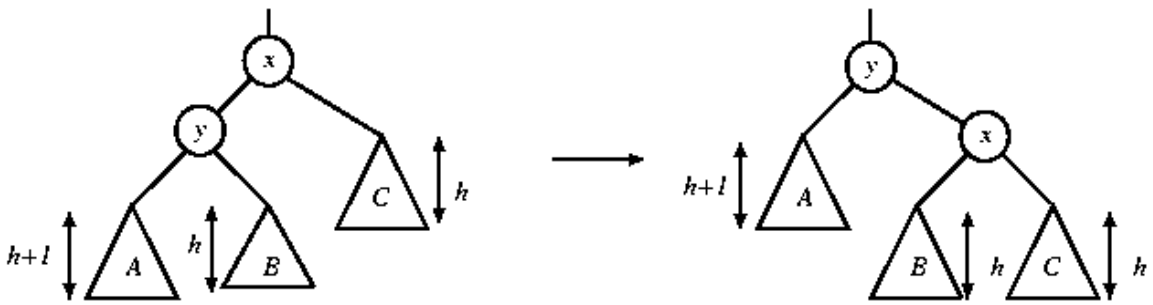
Fig 16

Deletion:

- Delete a node x as in ordinary binary search tree. Note that the last node deleted is a leaf.
- Then trace the path from the new leaf towards the root.
- For each node x encountered, check if heights of left(x) and right(x) differ by at most 1. If yes, proceed to parent(x). If not, perform an appropriate rotation at x. There are 4 cases as in the case of insertion.
- For deletion, after we perform a rotation at x, we may have to perform a rotation at some ancestor of x. Thus, we must continue to trace the path until we reach the root.
- On closer examination: the single rotations for deletion can be divided into 4 cases (instead of 2 cases)
    - Two cases for rotate with left child
    - Two cases for rotate with right child

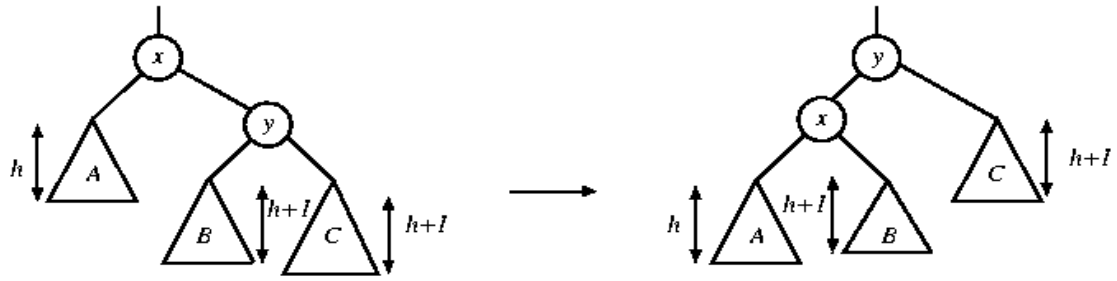
Single rotations in deletion

In both figures, a node is deleted in sub tree C, causing the height to drop to h. The height of y is h+2. When the height of sub tree A is h+1, the height of B can be h or h+1. Fortunately, the same single rotation can correct both cases.



Rotate with left child

In both figures, a node is deleted in sub tree A, causing the height to drop to h. The height of y is h+2. When the height of sub tree C is h+1, the height of B can be h or h+1. A single rotation can correct both cases.



rotate with right child

Rotations in deletion

- There are 4 cases for single rotations, but we do not need to distinguish among them.
- There are exactly two cases for double rotations (as in the case of insertion)
- Therefore, we can reuse exactly the same procedure for insertion to determine which rotation to perform

Hashing

Hashing is function that maps each key to a location in memory. A key's location does not depend on other elements, and does not change after insertion unlike a sorted list. A good hash function should be easy to compute. With such a hash function, the dictionary operations can be implemented in O(1) time.

- Static Hashing
    - File Organization
    - Properties of the Hash Function
    - Bucket Overflow
    - Indices
- Dynamic Hashing
    - Underlying Data Structure
    - Querying and Updating
- Comparisons
    - Other types of hashing
    - Ordered Indexing vs. Hashing

Static Hashing

- Hashing provides a means for accessing data without the use of an index structure.
- Data is addressed on disk by computing a function on a search key instead.

Organization

- A bucket in a hash file is unit of storage (typically a disk block) that can hold one or more records.
- The hash function, h, is a function from the set of all search-keys, K, to the set of all bucket addresses, B.
- Insertion, deletion, and lookup are done in constant time.

Querying and Updates

- To insert a record into the structure compute the hash value $h(K_i)$, and place the record in the bucket address returned.
- For lookup operations, compute the hash value as above and search each record in the bucket for the specific record.
- To delete simply lookup and remove.

Properties of the Hash Function

- The distribution should be uniform.
    - An ideal hash function should assign the same number of records in each bucket.
- The distribution should be random.
    - Regardless of the actual search-keys, the each bucket has the same number of records on average
    - Hash values should not depend on any ordering or the search-keys

Bucket Overflow

- How does bucket overflow occur?
    - Not enough buckets to handle data
    - A few buckets have considerably more records then others. This is referred to as skew.
        - Multiple records have the same hash value
        - Non-uniform hash function distribution.

Solutions

- Provide more buckets then are needed.
- Overflow chaining
    - If a bucket is full, link another bucket to it. Repeat as necessary.
    - The system must then check overflow buckets for querying and updates. This is known as closed hashing.

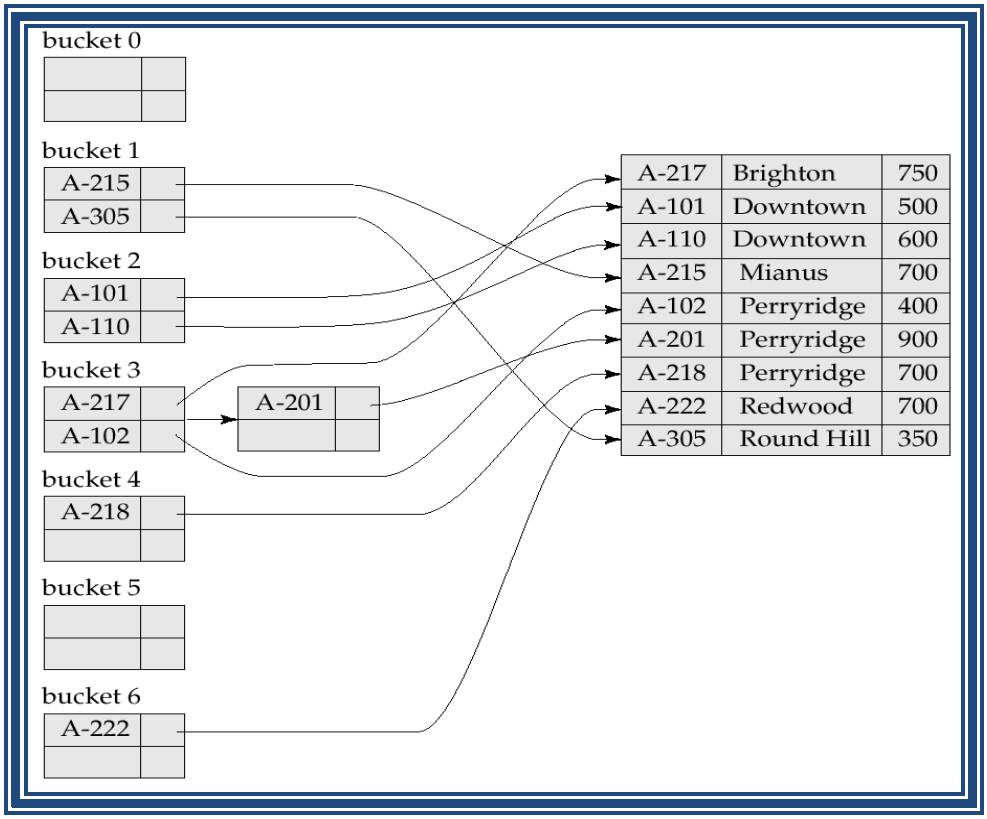
Alternatives

- Open hashing
    - The number of buckets is fixed
    - Overflow is handled by using the next bucket in cyclic order that has space.
        - This is known as linear probing.
- Compute more hash functions.
- Note: Closed hashing is preferred in database systems.

Indices

- A hash index organizes the search keys, with their pointers, into a hash file.
- Hash indices never primary even though they provide direct access.

Example of Hash Index



Dynamic Hashing

- More effective then static hashing when the database grows or shrinks
- Extendable hashing splits and coalesces buckets appropriately with the database size.
    - i.e. buckets are added and deleted on demand.

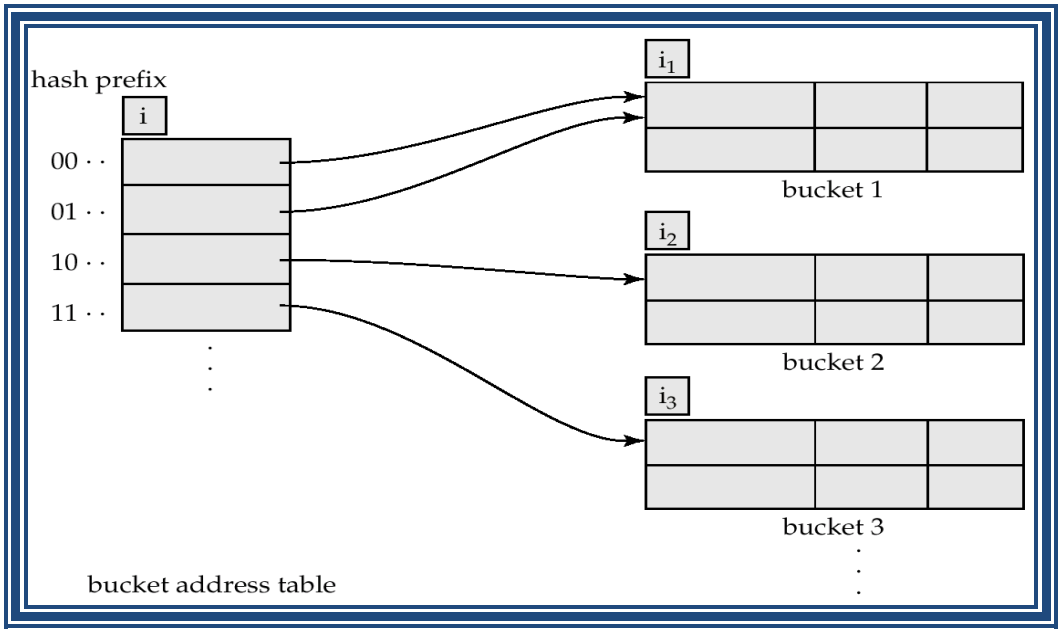
The Hash Function

- Typically produces a large number of values, uniformly and randomly.
- Only part of the value is used depending on the size of the database.

Data Structure

- Hash indices are typically a prefix of the entire hash value.
- More then one consecutive index can point to the same bucket.
    - The indices have the same hash prefix which can be shorter then the length of the index.

General Extendable Hash Structure



In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$
Queries and Updates

- Lookup
  - Take the first i bits of the hash value.
  - Following the corresponding entry in the bucket address table.
  - Look in the bucket.
- Insertion
  - Follow lookup procedure
  - If the bucket has space, add the record.
  - If not…

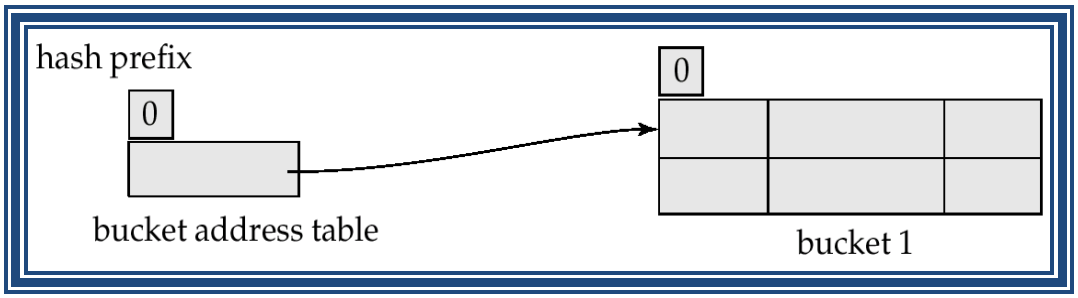
Insertion:

- Case 1: $i = i_j$
  - Use an additional bit in the hash value
    - This doubles the size of the bucket address table.
    - Makes two entries in the table point to the full bucket.
  - Allocate a new bucket, z.
    - Set i_j and i_z to i
    - Point the second entry to the new bucket
    - Rehash the old bucket
  - Repeat insertion attempt

- Case 2: $i > i_j$
  - Allocate a new bucket, z
  - Add 1 to i_j, set i_j and i_z to this new value
  - Put half of the entries in the first bucket and half in the other
  - Rehash records in bucket j
  - Reattempt insertion
- If all the records in the bucket have the same search value, simply use overflow buckets as seen in static hashing.

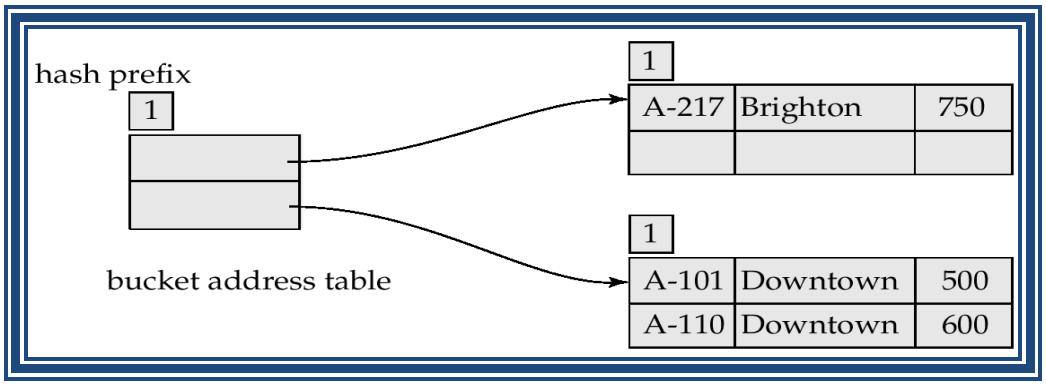
Use of Extendable Hash Structure: Example

branch-name	h(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001

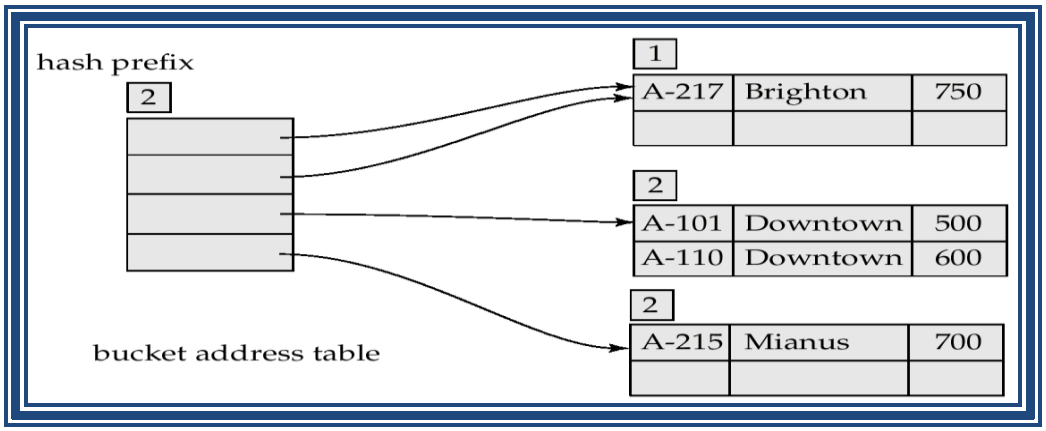


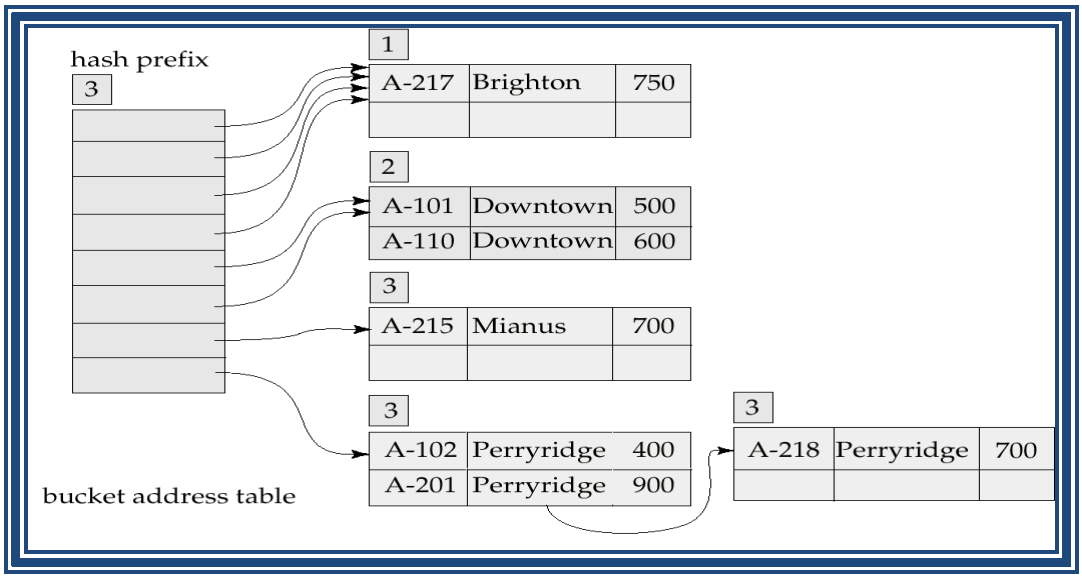
Initial Hash structure, bucket size = 2

- Hash structure after insertion of one Brighton and two Downtown records



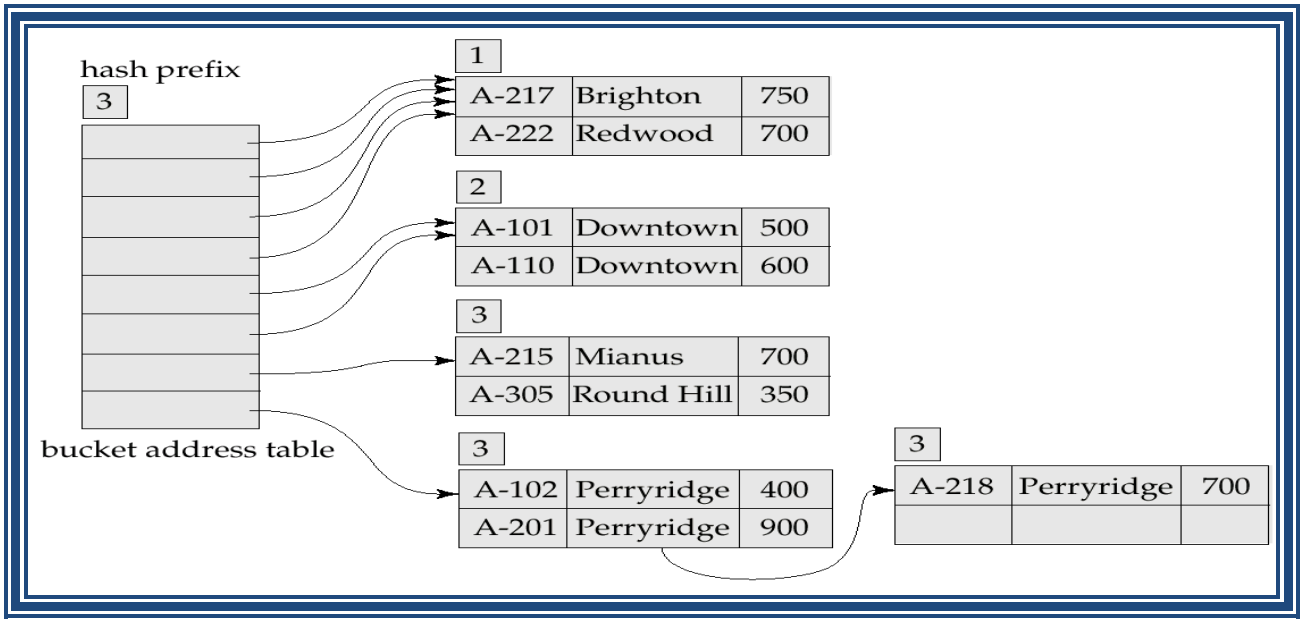
Hash structure after insertion of Minus record





Hash structure after insertion of three Perry ridge records

- Hash structure after insertion of Redwood and Round Hill records



Comparison to Other Hashing Methods

- Advantage: performance does not decrease as the database size increases
    - Space is conserved by adding and removing as necessary
- Disadvantage: additional level of indirection for operations
    - Complex implementation

Ordered Indexing vs. Hashing

- Hashing is less efficient if queries to the database include ranges as opposed to specific values.
- In cases where ranges are infrequent hashing provides faster insertion, deletion, and lookup then ordered indexing.

2.3 LET US SUM UP

- A hash table is a convenient data structure for storing items that provides O(1) access time.
- The concepts that drive selection of the key generation and compression functions can be complicated, but there is a lot of research information available.
- There are many areas where hash tables are used.
- Modern programming languages typically provide a hash table implementation ready for use in applications.

2.4 REFERENCES

1. Knuth, Donald A. The Art of Computer Programming. Philippines: Addison-Wesley Publishing Company, 1973.
2. Loudon, Kyle. Mastering Algorithms with C. Sebastopol: O'Reilly & Associates, 1999
3. Watt, David A., and Deryck F. Brown. Java Collections. West Sussex: John Wiley & Sons, 2001
4. Dewdney, A. K. The New Turing Omnibus. New York: Henry Holt and Company, 2001

2.5 SUGGESTION FOR BOOKS READING

BLOCK IV

UNIT 1 SORTING

STRUCTURE

1.0 AIMS AND OBJECTIVES

At the end of this lesson, students should be able to demonstrate appropriate skills, and show an understanding of the following:

- Aims and objectives of Sorting
- Preliminaries
- Insertion sort
- Shell Sort And Heap Sort
- Merge Sort and Quick sort
- External sort
- Sorting Implementations

1.1 INTRODUCTION

One common programming task is sorting a list of values. For example, we may want to view a list of student records sorted by student ID or alphabetically by student name. We may want to sort the list from the lowest value (of the student ID or name) to the highest (ascending order), or from highest to lowest (descending order).

Here, we will briefly review a few different sorting algorithms and their performance. We will assume without loss of generality that the different algorithms sort an array in ascending order, and we shall call this array data. We will also assume the availability of a standard swap routine.

Definition:

Many computer applications involve sorting the items in a list into some specified order. For example, we have seen that a list may be searched more efficiently if it is sorted.

To sort a group of items, the following relationships must be clearly defined over the items to be sorted:

a < b

a > b

a = b

Ascending order: smallest ... largest
Descending order: largest … smallest

When designing or choosing an algorithm for sorting, one goal is to minimize the amount of work necessary to sort the list of items. Generally the amount of work is measured by the number of comparisons of list elements and/or the number of swaps of list elements that are performed.

- Given a set (container) of n elements
  - E.g. array, set of words, etc.
- Suppose there is an order relation that can be set across the elements
- Goal Arrange the elements in ascending order
  - Start → 1 23 2 56 9 8 10 100
  - End → 1 2 8 9 10 23 56 100
- Desirable Characteristics
  - Fast
  - In place (don't need a secondary array)
  - Able to handle any values for the elements
  - Easy to understand

1.2 PRELIMINARIES

Selection Sort

In this algorithm, we rearrange the values of the array so that data [0] is the smallest, data[1] is the next smallest, and so forth up to data[n-1], the largest. Pseudo-code for this algorithm would be:
for(i=0; i<n; i++)
Put the next smallest element in location data[i];

Sorting Algorithms Using Priority Queues

- Remember Priority Queues = queue where the dequeue operation always removes the element with the smallest key → removeMin
- **Selection Sort**
  - insert elements in a priority queue implemented with an unsorted sequence
  - remove them one by one to create the sorted sequence
- **Insertion Sort**
  - insert elements in a priority queue implemented with a sorted sequence
  - remove them one by one to create the sorted sequence

Selection Sort

	Sequence S	Priority Queue P
Input	(7, 4, 8, 2, 5, 3, 9)	()
Phase 1:		
(a)	(4, 8, 2, 5, 3, 9)	(7)
(b)	(8, 2, 5, 3, 9)	(7, 4)
...	...	...
(g)	()	(7, 4, 8, 2, 5, 3, ,9)
Phase 2:		
(a)	(2)	(7, 4, 8, 5, 3, 9)
(b)	(2, 3)	(7, 4, 8, 5, 9)
(c)	(2, 3, 4)	(7, 8, 5, 9)
(d)	(2, 3, 4, 5)	(7, 8, 9)
(e)	(2, 3, 4, 5, 7)	(8, 9)
(f)	(2, 3, 4, 5, 7, 8)	(9)
(g)	(2, 3, 4, 5, 7, 8, 9)	()

- **insertion: O(1 + 1 + … + 1) = O(n)**
- **selection: O(n + (n-1) + (n-2) + … + 1) = O(n^2)**

Bubble Sort

This Sorting looks at pairs of entries in the array, and swaps their order if needed. After the first step of the algorithm the maximal value will "bubble" up the list and will occupy the last entry. After the second iteration, the second largest value will "bubble" up the list and will occupy the next to last entry, and so on.

For example, if our initial array is:

(8, 2, 5, 3, 10, 7, 1, 4, 6, 9)

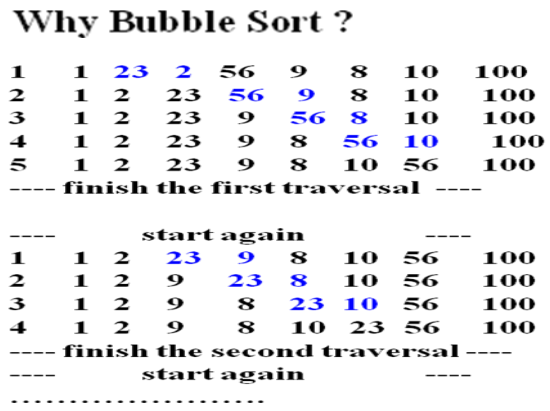
Then, after the first step it will be:
(2, 5, 3, 8, 7, 1, 4, 6, 9, 10)
After the second step:
(2, 3, 5, 7, 1, 4, 6, 8, 9, 10)
And, after the third:
(2, 3, 5, 1, 4, 6, 7, 8, 9, 10)
Pseudo-code for this algorithm would be:

```
for (i=n-1; i>=0; i--)
for (j=0; j<i; j++)
if (data[j] > data[j+1])
swap(data[j], data[j+1])
```

Note that this algorithm can be improved if we keep track of whether a swap took place during outer loop iteration. If there were no swaps, the algorithm may stop because the array is sorted.

Bubble sort has the following:

- Simplest sorting algorithm
- **Idea:**
  - Set flag = false
  - Traverse the array and compare pairs of two elements
    - 1.1 If E1 ≤ E2 - OK
    - 1.2 If E1 > E2 then Switch(E1, E2) and set flag = true
  - If flag = true goto 1.
- What happens?



Implement Bubble Sort with an Array

```
void bubbleSort (Array S,  length n) {
boolean isSorted = false;
while(!isSorted) {
isSorted = true;
for(i = 0; i<n; i++) {
```

```
if(S[i] > S[i+1]) {
int aux = S[i];
S[i] = S[i+1];
S[i+1] = aux;
        isSorted = false;
}
}
}
```

Running Time for Bubble Sort

- One traversal = move the maximum element at the end
- Traversal #i : n – i + 1 operations
- Running time:
    - (n – 1) + (n – 2) + … + 1 = (n – 1) n / 2 = $O(n^2)$
- When does the worst case occur?
- Best case?

1.3 INSERTION SORT

Definition

- Simple, able to handle any data
- Grow a sorted array from a beginning
    - Create and empty array for the proper size
    - Pick the elements one at a time in any order
    - Put them in the new array in sorted order
        - If the element is not last, make room for it
    - Repeat until done
- Can be done in place if well designed

This sorting takes one value at a time and builds up another sorted list of values. It helps if you can imagine what you do when you play a game of cards: you pick a card and insert it to an already sorted list of cards. For example, if our initial array is:

(8, 2, 5, 3, 10, 7, 1, 4, 6, 9)
Then, after the first step it will be unchanged:
(8, 2, 5, 3, 10, 7, 1, 4, 6, 9)
After the second step:
(2, 8, 5, 3, 10, 7, 1, 4, 6, 9)
After the third:
(2, 5, 8, 3, 10, 7, 1, 4, 6, 9)
And so on until the entire array is sorted. Notice that the first part of the list, which is colored with red is always sorted.

Pseudo-code for this algorithm would be:

```
for(i=1; i<n; i++)
ordered_insert(data, i, data[i]);
```

Where ordered_insert() inserts a new value in a sorted array, so that the resulting array will also be sorted.

All the above sorting algorithms run in quadratic time, or O(n^2). We will not give a rigorous proof, but rather give the intuition why this is so. The reason is that in all of the above algorithms an O(n) process is performed *n* times. Another explanation, which is closer to the rigorous proof, is to realize that if you count the operations (and sum them up) in each of the above algorithms then you get the formula:

1 + 2 + 3 + ... + n = n*(n+1) / 2 = n^2/2 + n/2 = O(n^2).

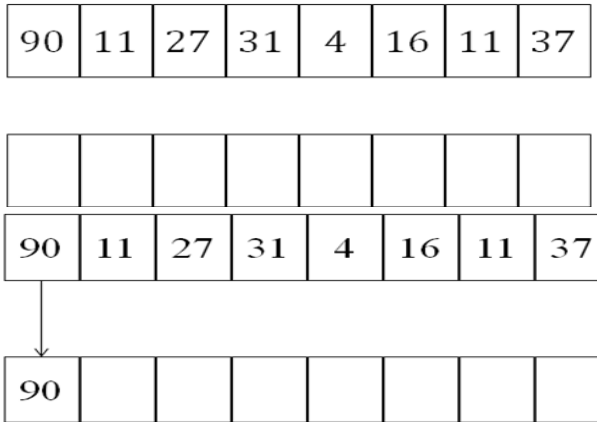
The following algorithms are more efficient, and by careful analysis can be shown to run in time O(nlogn). For the following algorithms, the design and analysis are much simplified, if we consider array sizes which are powers of 2.

	Sequence S	Priority Queue P
Input	(7, 4, 8, 2, 5, 3, 9)	()
Phase 1:		
(a)	(4, 8, 2, 5, 3, 9)	(7)
(b)	(8, 2, 5, 3, 9)	(4, 7)
(c)	(2, 5, 3, 9)	(4, 7, 8)
(d)	(5, 3, 9)	(2, 4, 7, 8)
(e)	(3, 9)	(2, 4, 5, 7, 8)
(f)	(9)	(2, 3, 4, 5, 7, 8)
(g)	()	(2, 3, 4, 5, 7, 8, 9)
Phase 2:		
(a)	(2)	(3, 4, 5, 7, 8, 9)
(b)	(2, 3)	(4, 5, 7, 8, 9)
...	...	...
(g)	(2, 3, 4, 5, 7, 8, 9)	()

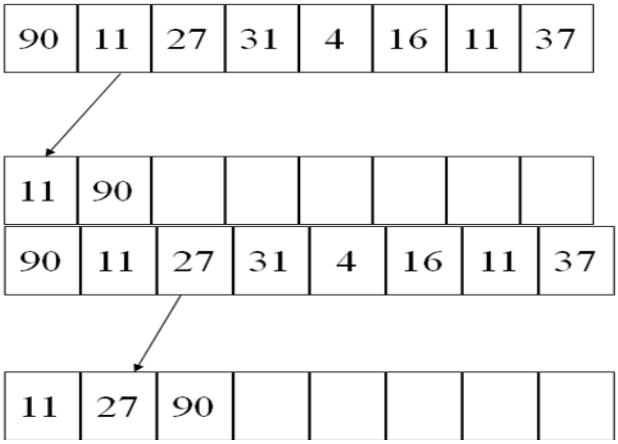
- **insertion: $O(1 + 2 + \ldots + n) = O(n^2)$**
- **selection: $O(1 + 1 + \ldots + 1) = O(n)$**

The below diagram explains the Insertion sort:

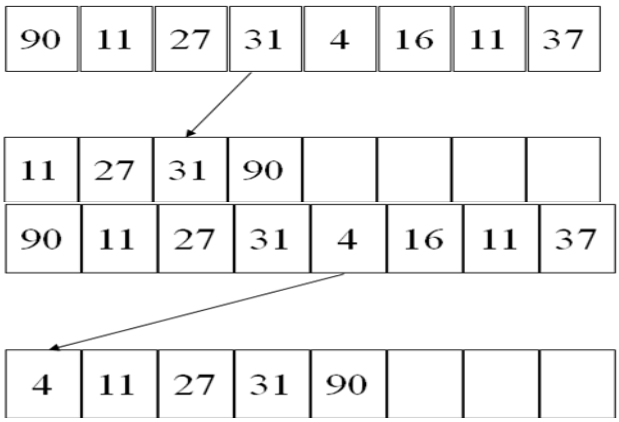
(a) **(b)**


90 11 27 31 4 16 11 37
90 11 27 31 4 16 11 37
90

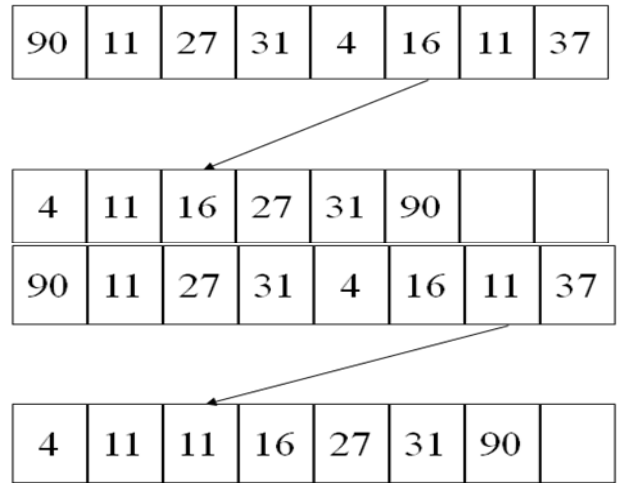
(c) **(d)**


90 11 27 31 4 16 11 37
11 90
90 11 27 31 4 16 11 37
11 27 90

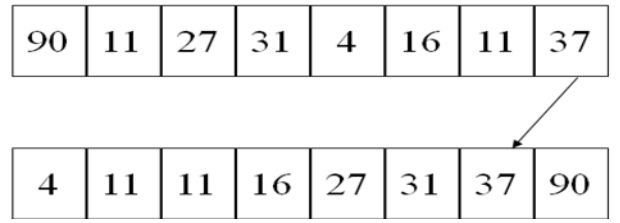
(e) **(f)**


90 11 27 31 4 16 11 37
11 27 31 90
90 11 27 31 4 16 11 37
4 11 27 31 90

(g) **(h)**



(i)



- Sorting can actually be done in place
  - Never need the same element in both arrays
- Every insertion can cause lots of copying
  - If there are N elements, need to do N insertions
  - Worst case is about N/2 copies per insertion
  - N elements can take nearly N operations to sort
- But each operation is very fast
  - So this is fine if N is small (20 or so)

Sorting with Binary Trees

- Using heaps (see lecture on heaps)
  - How to sort using a minHeap?
- Using binary search trees (see lecture on BST)
  - How to sort using BST
  -

1.7 HEAP SORTING

This type of sorting starts by building a *heap* from the initial array. Since the maximum element of the array is stored at the root, data [0], it can be put into its correct final position by exchanging it with data [n-1].

If we then "discard" node *n-1* from the heap, we observe that data [0..(n-2)] can easily be made into a heap. The children of the root remain heaps, but the new root element may violate the heap property. All that is needed to restore the heap property, however, is one call

to reheapify_down(), which leaves a heap indata[0..(n-2)]. The heapsort algorithm then repeats this process for the heap of size *n-2* down to a heap of size 1.

```
heapsort(vector<Item>& v)
build_heap(v); // create a heap vector corresponding to v
for(i=v.size()-1; i>=1; i--)
swap(heap[0], heap[i]);
heap-size = heap-size - 1;
reheapify_down();

buildheap(vector<Item>& v)
for(i=0; i<v.size(); i++)
insert(v[i]);
```

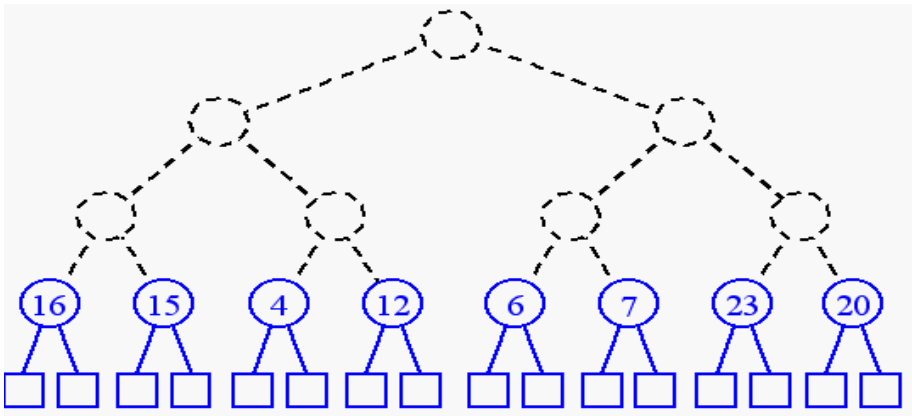
A more efficient buildheap() procedure will require a slight modification to the reheapify_down() routine to take as an argument the index *i* of a node. So we would call reheapify_down(i) to reheapify down from node *i*, and reheapify_down(0) to reheapify down from the root.

```
buildheap(vector<Item> &v)
heap-size = v.size();
for(i=v.size()/2 - 1; i >= 0; i--)
reheapify_down(i);  // reheapify from node i
```

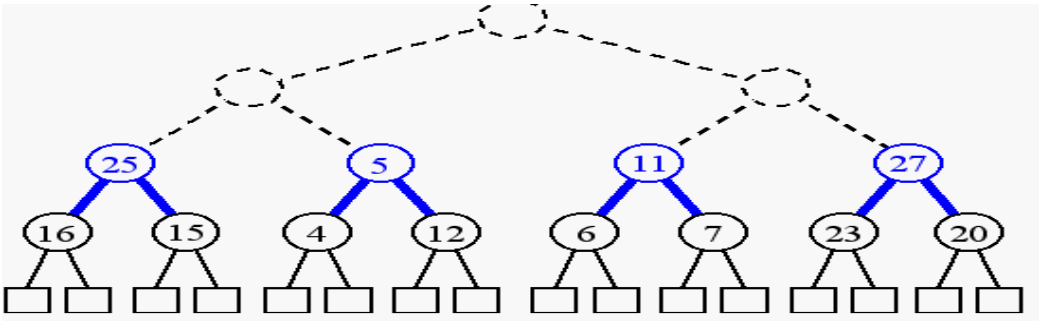
- **Step 1: Build a heap**
- **Step 2: removeMin( )**

Recall: Building a Heap:

- **build (n + 1)/2 trivial one-element heaps**

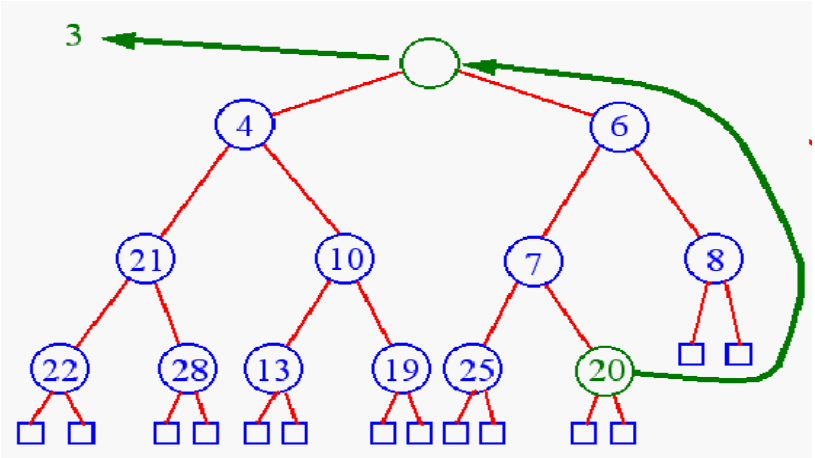


- **build three-element heaps on top of them**



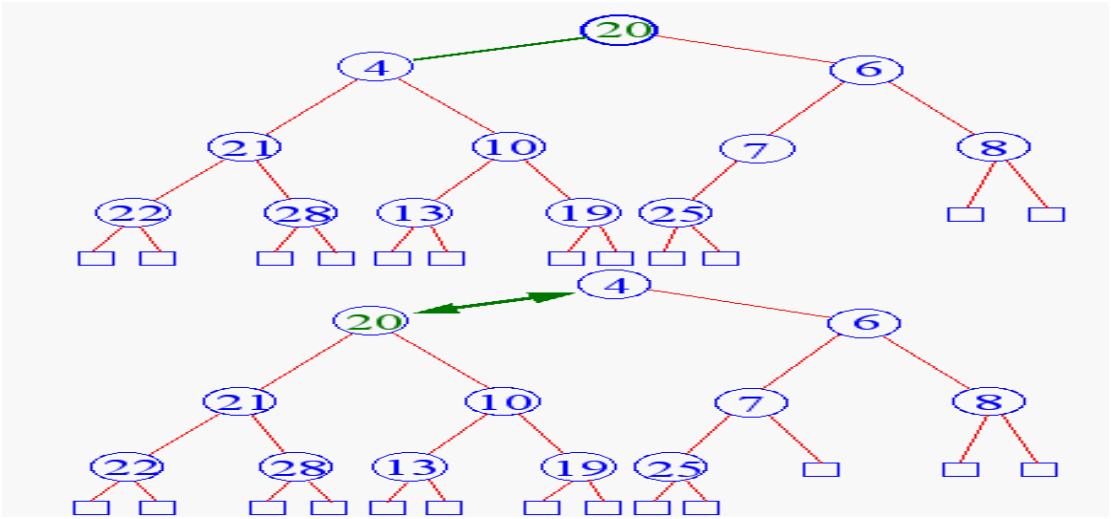
Recall: Heap Removal:

- **Remove element from priority queues?**
  - **Remove Min( )**



Recall: Heap Removal

Begin downheap



Sorting with BST

- Use binary
- search trees for
- sorting
- Start with
- unsorted sequence
- Insert all
- elements in a
- BST
- Traverse the
- tree…. how?
- Running time?
- Sorting
- algorithms that rely on the "DIVIDE AND CONQUER" paradigm
  - ❖ One of the most widely used paradigms
  - ❖ Divide a problem into smaller sub problems, solve the sub problems, and combine the solutions
  - ❖ Learned from real life ways of solving problems

Divide-and-Conquer

- **Divide and Conquer** is a method of algorithm design that has created such efficient algorithms as Merge Sort.
- In terms or algorithms, this method has three distinct steps:
  - **Divide:** If the input size is too large to deal with in a straightforward manner, divide the data into two or more disjoint subsets.
  - **Recur:** Use divide and conquer to solve the sub problems associated with the data subsets.
- Conquer: Take the solutions to the sub problems and "merge" these solutions into a solution for the original problem.

1.8 MERGE-SORT

Definition:

- Fast, able to handle any data
  - But can't be done in place
- View the array as a set of small sorted arrays
  - Initially only the 1-element "arrays" are sorted
- Merge pairs of sorted arrays
  - Repeatedly choose the smallest element in each
  - This produces sorted arrays that are twice as long

Repeat until only one array remains

Algorithm:

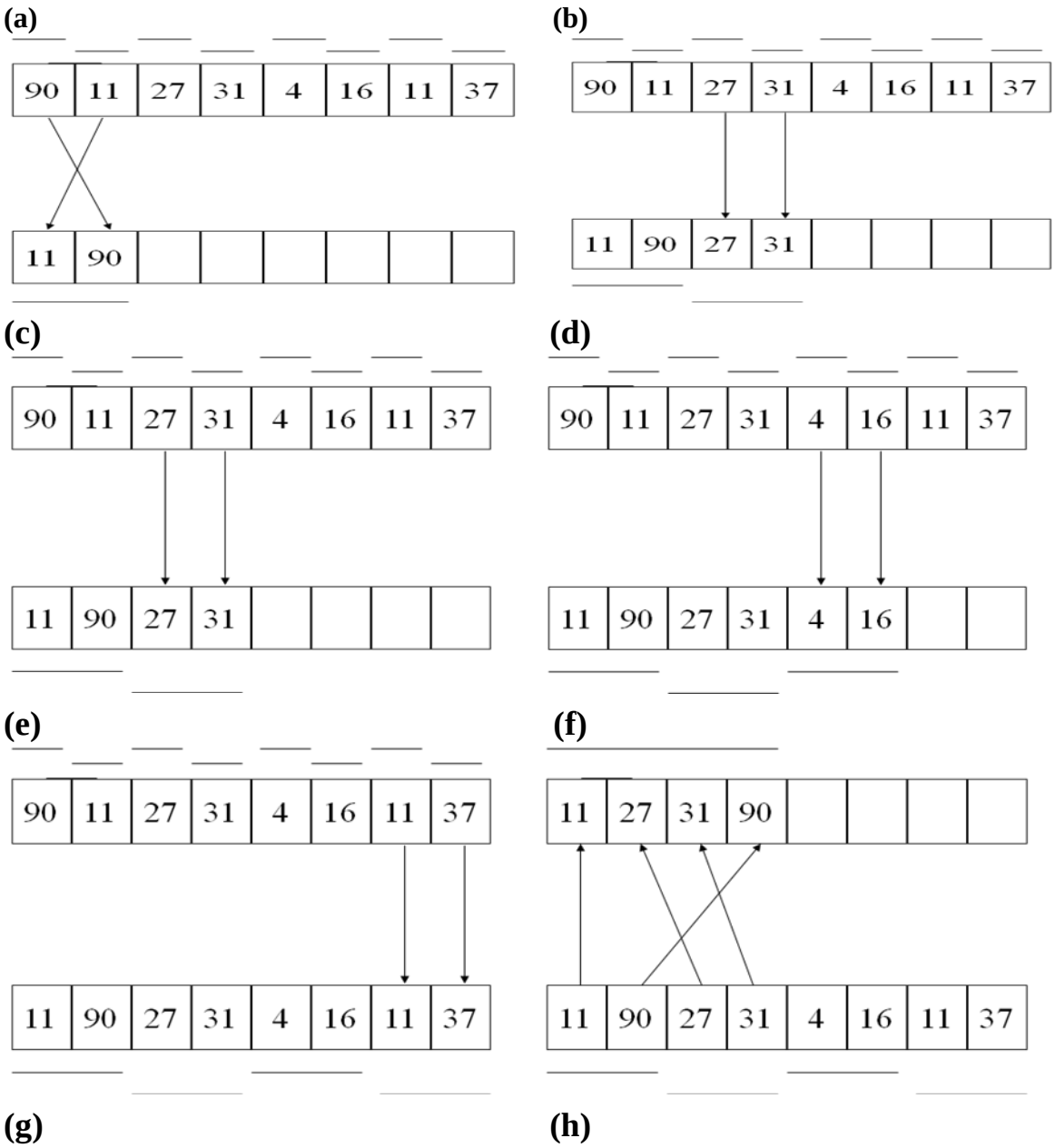
1. **Divide:** If S has at leas two elements (nothing needs to be done if S has zero or one elements), remove all the elements from S and put them into two sequences, S_1 and S_2, each containing about half of the elements of S. (i.e. S_1 contains the first $\lceil n/2 \rceil$ elements and S_2 contains the remaining $\lfloor n/2 \rfloor$ elements.

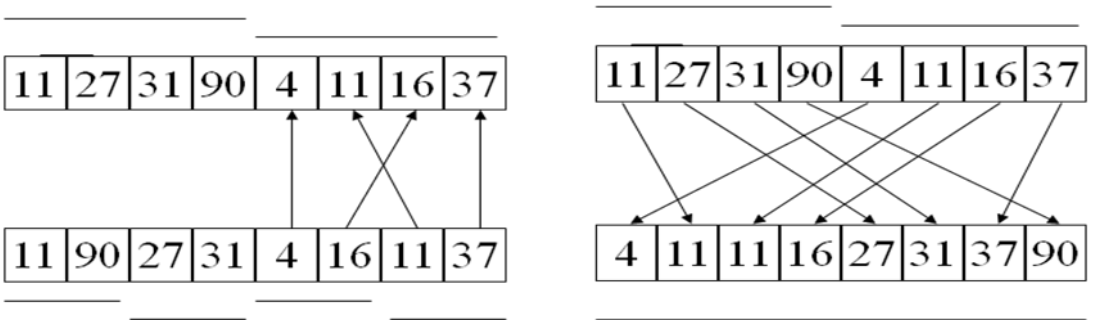
2. **Recur:** Recursive sort sequences S_1 and S_2.
3. **Conquer:** Put back the elements into S by merging the sorted sequences S_1 and S_2 into a unique sorted sequence.

This recursive algorithm belongs to the very important **divide-and-conquer** paradigm. It consists of the following steps:

1. **Divide** the elements to be sorted into two groups of equal (or almost equal) size. (This is why it is easier if we assume that the array size is a power of 2.)
2. **Conquer** - Sort each of these smaller groups of elements (by recursive calls).
3. **Merge** - Combine the two sorted groups into one large sorted list.

For example:



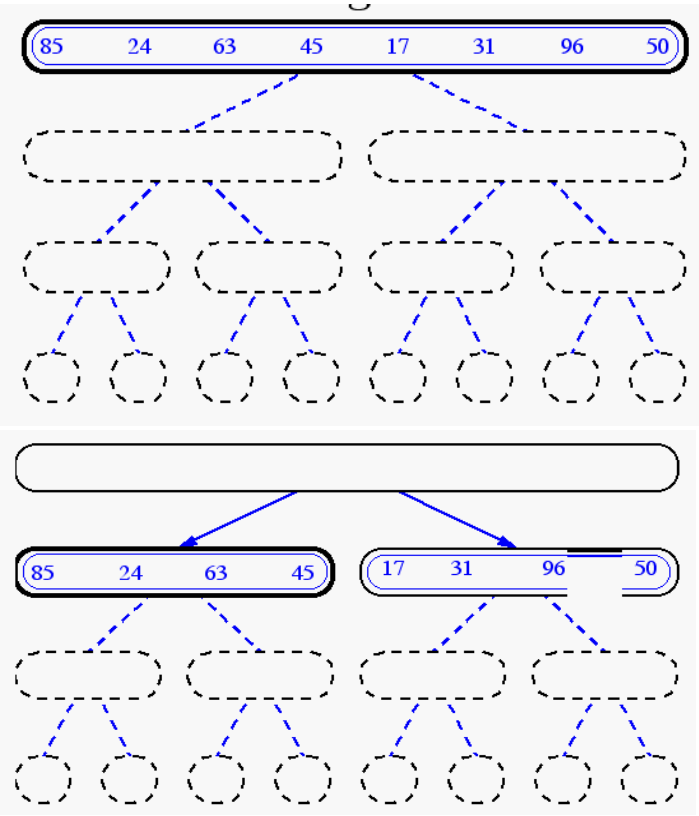


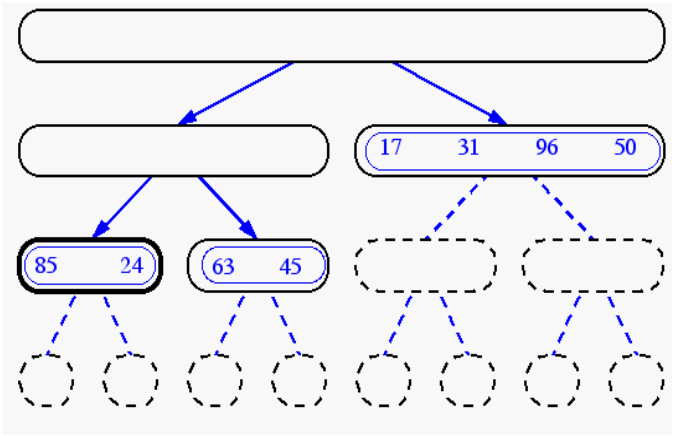
- Each array size requires N steps
    - But 8 elements requires only 3 array sizes
- In general, 2 elements requires k array sizes
    - So the complexity is N*log(N)
- No faster sort (based on comparisons) exists
    - Faster sorts require assumptions about the data
    - There are other N*log(N) sorts, though

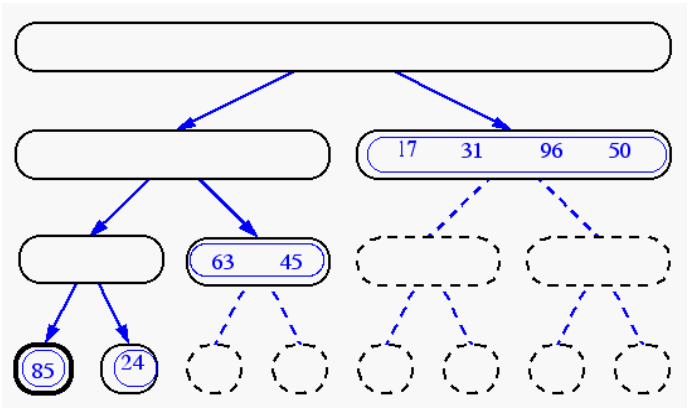
Merge sort is most often used for large disk files

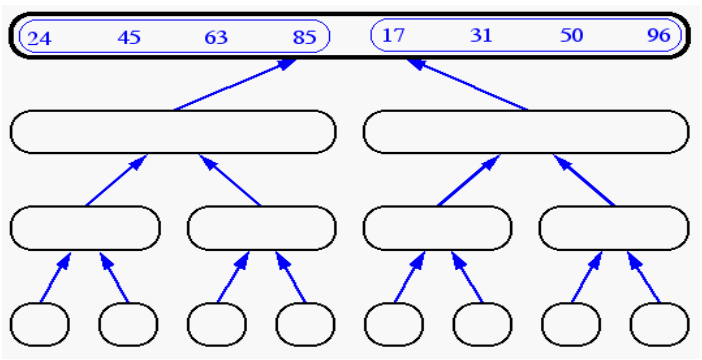
Merge Sort Tree:

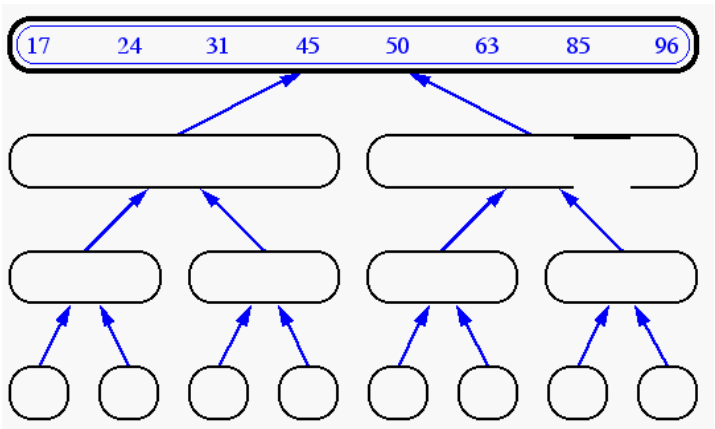
- Take a binary tree T
- Each node of T represents a recursive call of the merge sort algorithm.
- We associate with each node v of T a the set of input passed to the invocation v represents.
- The external nodes are associated with individual elements of S, upon which no recursion is called.




17 31 96 50
85 24
63 45


17 31 96 50
63 45
85
24


24 45 63 85
17 31 50 96


17 24 31 45 50 63 85 96

Merging Two Sequences

- Pseudo-code for merging two sorted sequences into a unique sorted sequence

```
Algorithm merge (S1, S2, S):
  Input: Sequence S1 and S2 (on whose elements a
  total order relation is defined) sorted in nondecreas
  ing order, and an empty sequence S.
  Ouput: Sequence S containing the union of the ele
  ments from S1 and S2 sorted in nondecreasing order;
  sequence S1 and S2 become empty at the end of the
  execution
  while S1 is not empty and S2 is not empty do
    if S1.first().element() ≤ S2.first().element() then
      {move the first element of S1 at the end of S}
      S.insertLast(S1.remove(S1.first()))
    else
      { move the first element of S2 at the end of S}
      S.insertLast(S2.remove(S2.first()))
  while S1 is not empty do
    S.insertLast(S1.remove(S1.first()))
    {move the remaining elements of S2 to S}
  while S2 is not empty do
    S.insertLast(S2.remove(S2.first()))
```

1.9 QUICK-SORT

Given an array of *n* elements (e.g., integers):

If array only contains one element, return

Else

Pick one element to use as *pivot.*

Partition elements into two sub-arrays:

Elements less than or equal to pivot

Elements greater than pivot

Quicksort two sub-arrays

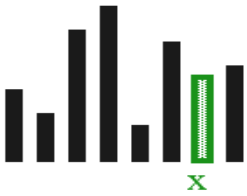
Return results

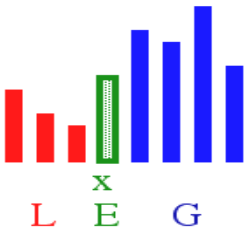
Another divide-and-conquer sorting algorihm.To understand quick-sort, let's look at a high-level description of the algorithm. Divide: If the sequence S has 2 or more elements, select an element x from S to be your pivot. Any arbitrary element, like the last, will do. Remove all the elements of S and divide them into 3 sequences:

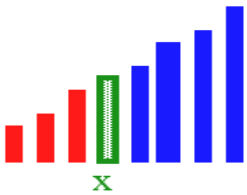
- L, holds S's elements less than x
- E, holds S's elements equal to x
- G, holds S's elements greater than x
- Recurse: Recursively sort L and G
- Conquer: Finally, to put elements back into S in order, first inserts the elements of L, then those of E, and those of G.
- Here are some diagrams....

Idea of Quick Sort

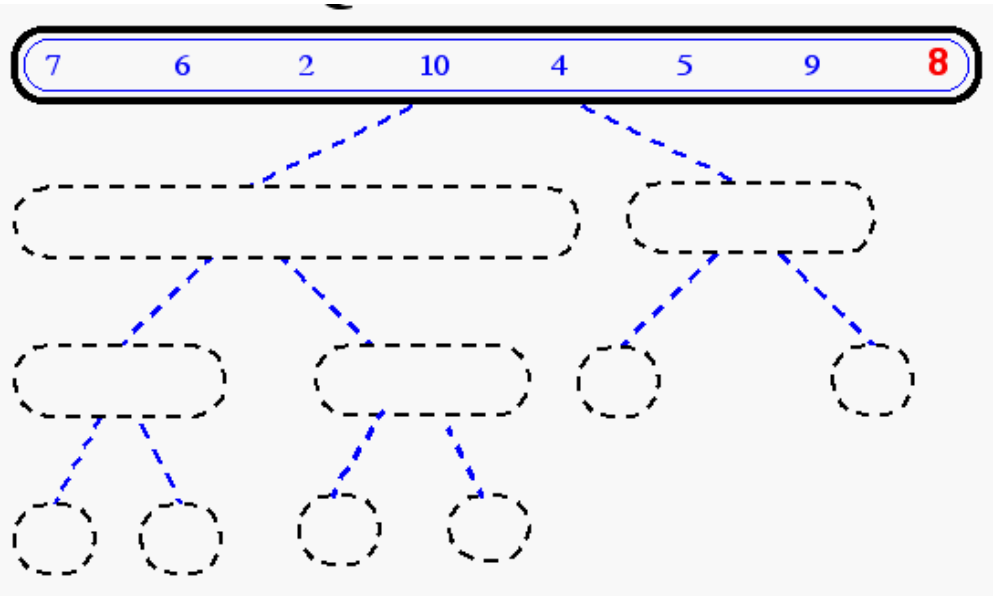
1. **Select: pick an element**
2. **Divide: rearrange elements so that x goes to its final position E**
3. **Recurse and Conquer: recursively sort**

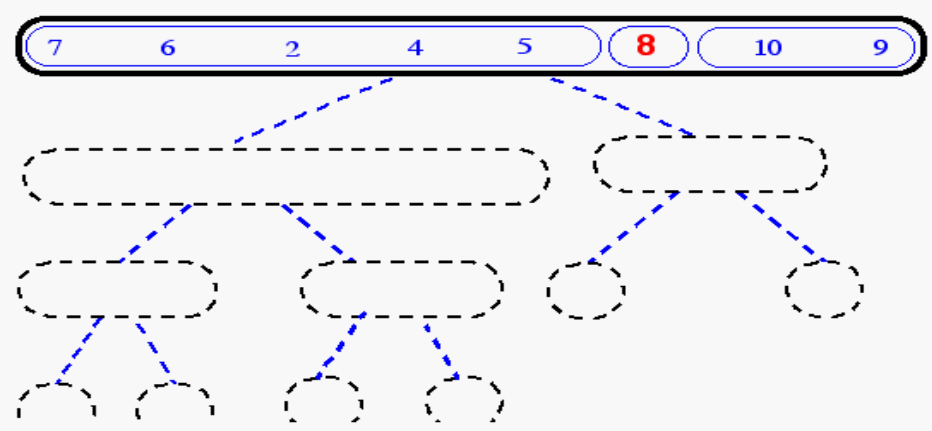






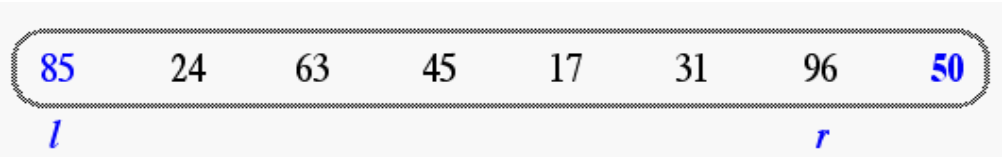
Quick-Sort Tree



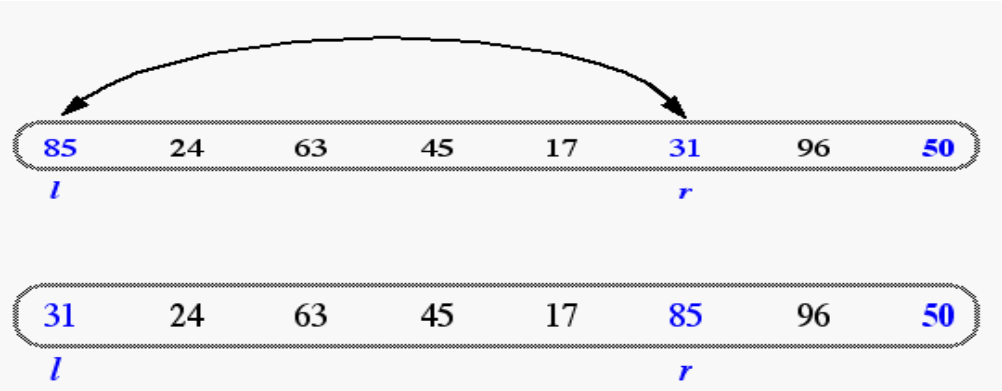


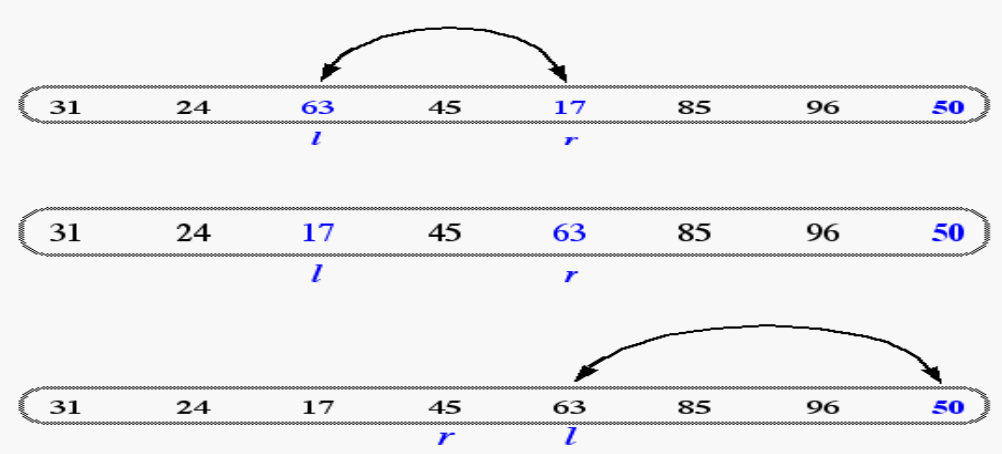
In-Place Quick-Sort

Divide step: l scans the sequence from the left, and r from the right.

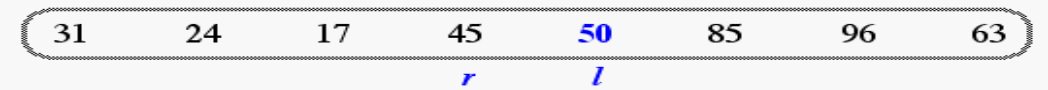


A swap is performed when l is at an element larger than the pivot and r is at one smaller than the pivot.





A final swap with the pivot completes the divide step



Quick sort Analysis

- Assume that keys are
- random, uniformly distributed.
- Best case
- running time:
- $O(n \log_2 n)$
- Worst case
- running time?
- Recursion:
- Partition splits
- array in two
- sub-arrays:
    - one sub-array of
    - size 0
    - the other sub-
    - array of size n-1
- Quicksort each
- sub-array
    - Depth of
    - recursion tree? O(n)
    - Number of
    - accesses per
    - partition? O(n)

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!
- What can we do to avoid worst case?

Running time analysis

★ Worst case: when the pivot does not divide the sequence in two

★ At each step, the length of the sequence is only reduced by 1

★ Total running time $\sum_{i=n}^{1} length(S_i) = O(n^2)$

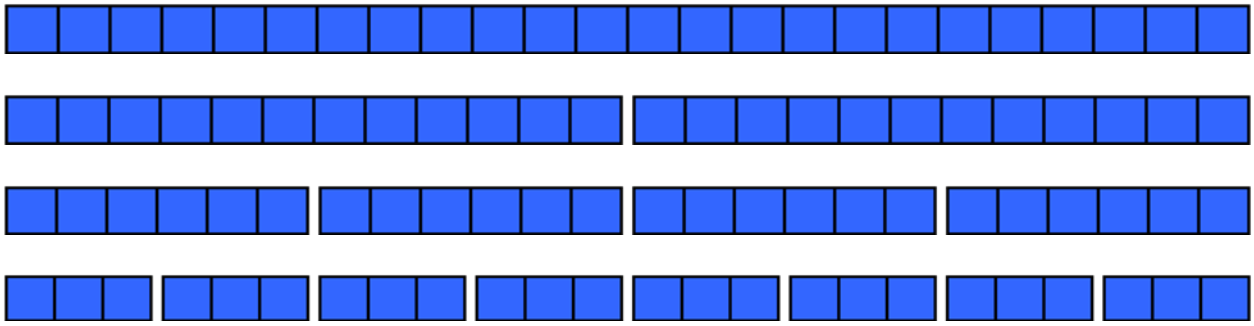
★ **General case:**
- Time spent at level i in the tree is O(n)
- Running time: O(n) * O(height)

★ **Average case:**
- O(n log n)

Quick sort: Best case

- Suppose each partition operation divides the array almost exactly in half
- Then the depth of the recursion in $\log_2 n$
    - Because that's how many times we can halve n
- However, there are many recursions!
    - How can we figure this out?
    - We note that
    - Each partition is linear over its subarray
    - All the partitions at one level cover the array

Partitioning at various levels:



Best case II

- We cut the array size in half each time
- So the depth of the recursion in $\log_2 n$
- At each level of the recursion, all the partitions at that level do work that is linear in n
- $O(\log_2 n) * O(n) = O(n \log_2 n)$
- Hence in the average case, quicksort has time complexity $O(n \log_2 n)$
- What about the worst case?

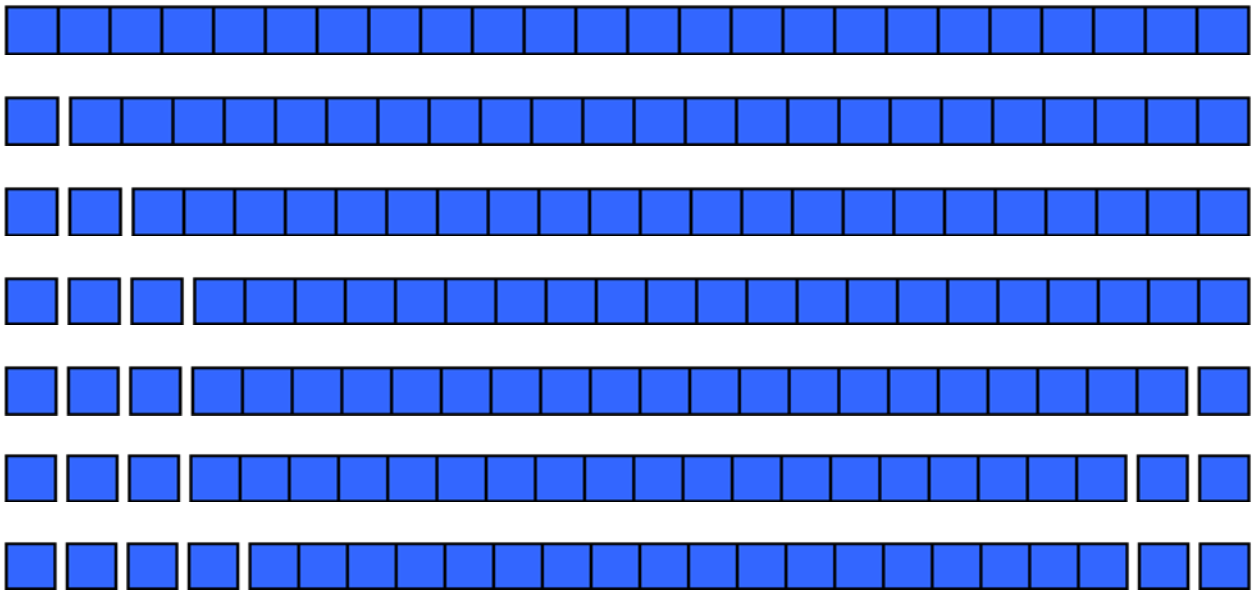
For Example:

$$
\begin{aligned}
T(N) &= 2T(N/2) + cN \\
\frac{T(N)}{N} &= \frac{T(N/2)}{N/2} + c \\
\frac{T(N/2)}{N/2} &= \frac{T(N/4)}{N/4} + c \\
\frac{T(N/4)}{N/4} &= \frac{T(N/8)}{N/8} + c \\
&\vdots \\
\frac{T(2)}{2} &= \frac{T(1)}{1} + c \\
\frac{T(N)}{N} &= \frac{T(1)}{1} + c \log N \\
T(N) &= cN \log N + N = O(N \log N)
\end{aligned}
$$

Quick sort: Worst Case

- In the worst case, partitioning always divides the size n array into these three parts:
  - A length one part, containing the pivot itself
  - A length zero part, and
  - A length n-1 part, containing everything else
- We don't recur on the zero-length part
- Recurring on the length n-1 part requires (in the worst case) recurring to depth n-1

Worst case partitioning



- In the worst case, recursion may be n levels deep (for an array of size n)
- But the partitioning work done at each level is still n
- O(n) * O(n) = $O(n^2)$
- So worst case for Quicksort is $O(n^2)$
- When does this happen?
  - When the array is sorted to begin with!

For Example:

$$
\begin{aligned}
T(N) &= T(N-1) + cN \\
T(N-1) &= T(N-2) + c(N-1) \\
T(N-2) &= T(N-3) + c(N-2) \\
&\vdots \\
T(2) &= T(1) + c(2) \\
T(N) &= T(1) + c\sum_{i=2}^{N} i = O(N^2)
\end{aligned}
$$

Improved Pivot Selection

Pick median value of three elements from data array:
data[0], data[n/2], and data[n-1].
Use this median value as pivot.

Improving Performance of Quick sort

- Improved selection of pivot.
- For sub-arrays of size 3 or less, apply brute force search:
    - Sub-array of size 1: trivial
    - Sub-array of size 2:
    - if(data[first] > data[second]) swap them
    - Sub-array of size 3: left as an exercise.

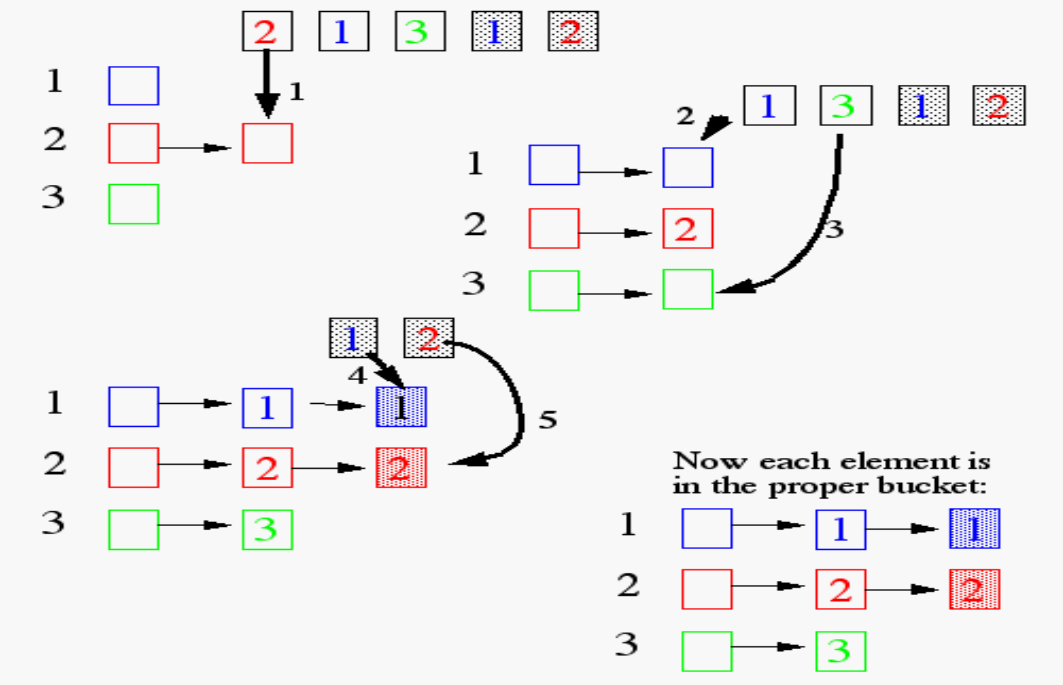
1.10 MORE SORTING ALGORITHMS

- Bucket Sort
- Radix Sort
- Stable sort
    - A sorting algorithm where the order of elements having the same key is not changed in the final sequence
    - Is bubble sort stable?
    - Is merge sort stable?

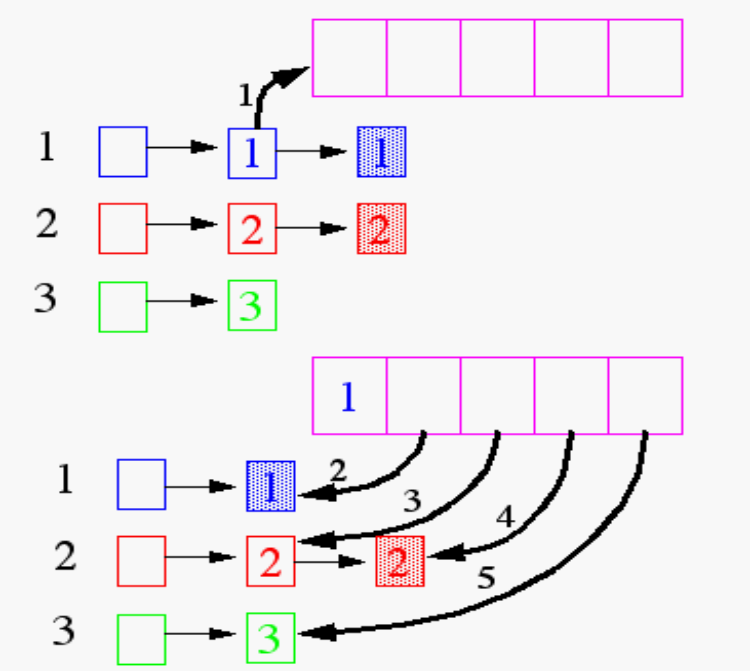
1.10.1 Bucket Sort

- Bucket sort
- Assumption: the keys are in the range [0, N)
- Basic idea:
- Create *N* linked lists (*buckets*) to divide interval [0,N) into subintervals of size 1
- Add each input element to appropriate bucket
    - Concatenate the buckets
- Expected total time is O(n + N), with n = size of original sequence
    - if N is O(n) → sorting algorithm in O(n) !

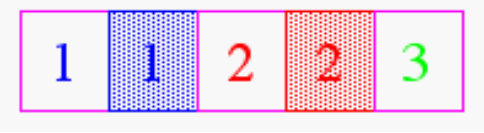
Each element of the array is put in one of the N "buckets"



Now, pull the elements from the buckets into the array



At last, the sorted array (sorted in a stable way):



Does it Work for Real Numbers?

- What if keys are not integers?
  - Assumption: input is *n* reals from [0, 1)
  - Basic idea:
- Create *N* linked lists (*buckets*) to divide interval [0,1) into subintervals of size 1/*N*
- Add each input element to appropriate bucket and sort buckets with insertion sort
  - Uniform input distribution → O(1) bucket size
  - Therefore the expected total time is O(n)
  - Distribution of keys in buckets similar with …. ?

1.10.2 Radix Sort

- **How did IBM get rich originally?**
- **Answer:** punched card readers for census tabulation in early 1900's.
  - In particular, a *card sorter* that could sort cards into different bins
  - Each column can be punched in 12 places
  - (Decimal digits use only 10 places!)
  - **Problem:** only one column can be sorted on at a time
- Intuitively, you might sort on the most significant digit, then the second most significant, etc.
- **Problem:** lots of intermediate piles of cards to keep track of
- **Key idea:** sort the *least* significant digit first

  RadixSort(A, d)

  for i=1 to d
  - **StableSort(A) on digit i**
- **Can we prove it will work?**
- **Inductive argument:**
  - Assume lower-order digits {j: j<i}are sorted
  - Show that sorting next digit i leaves array correctly sorted
- If two digits at position i are different, ordering numbers by that digit is correct (lower-order digits irrelevant)
- If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order
- **What sort will we use to sort on digits?**
  - Bucket sort is a good choice
  - Sort *n* numbers on digits that range from 1..*N*
  - Time: O(*n* + *N*)
  - Each pass over *n* numbers with *d* digits takes time O(*n*+*k*), so total time O(*dn*+*dk*)
  - When *d* is constant and *k*=O(*n*), takes O(*n*) time

Radix Sort Example

- Problem: sort 1 million 64-bit numbers
  - Treat as four-digit radix 2^{16} numbers
  - Can sort in just four passes with radix sort!
  - Running time: $4(1 \text{ million} + 2^{16}) \approx 4$ million operations

- Compare with typical O(n lg n) comparison sort
  - Requires approx lg $n = 20$ operations per number being sorted
  - Total running time $\approx$ 20 million operations

- In general, radix sort based on bucket sort is
  - Asymptotically fast (i.e., O(n))
  - Simple to code
  - A good choice
  - Can radix sort be used on floating-point numbers?

Summary: Radix Sort

- **Radix sort:**
  - Assumption: input has d digits ranging from 0 to k
- **Basic idea:**
  - Sort elements by digit starting with *least* significant
- Use a stable sort (like bucket sort) for each stage
  - Each pass over n numbers with *1* digit takes time O(n+k), so total time O(dn+dk)
  - When d is constant and k=O(n), takes O(n) time
  - Fast, Stable, Simple
  - Doesn't sort in place

- **Sorting Algorithms: Running Time**

- Assuming an input sequence of length n
  - Bubble sort
  - Insertion sort
  - Selection sort
  - Heap sort
  - Merge sort
  - Quick sort
  - Bucket sort
  - Radix sort

Sorting Algorithms: In-Place Sorting

- A sorting algorithm is said to be *in-place* if
  - it uses no auxiliary data structures (however, O(1) auxiliary variables are allowed)
  - it updates the input sequence only by means of operations replace Element and swap Elements
- Which sorting algorithms seen so far can be made to work in place?

bubble-sort	Y
selection-sort	
insertion-sort	
heap-sort	
merge-sort	
quick-sort	
radix-sort	
bucket-sort	

1.11 EXTERNAL SORTING

External Sorting:

- Very large files (overheads in disk access)
  - seek time
  - latency time
  - transmission time
- merge sort
  - phase 1
    Segment the input file & sort the segments (runs)
  - phase 2
    Merge the runs

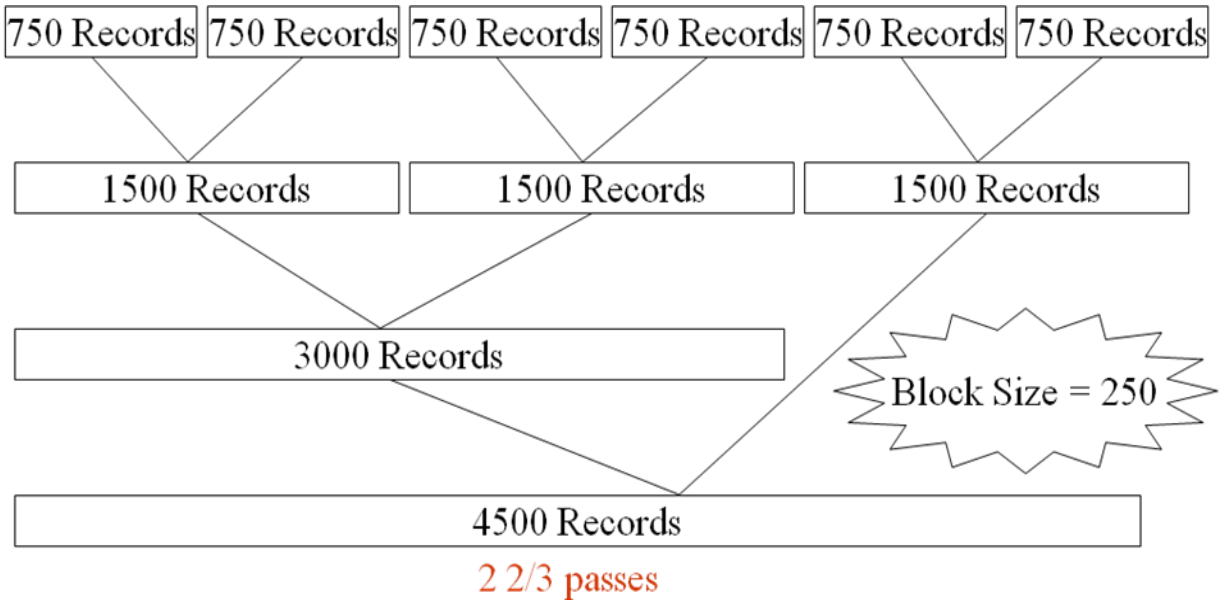
Why Sort?

- A classic problem in computer science!
- Data requested in sorted order
  - e.g., find students in increasing *gpa* order
- Sorting is first step in *bulk loading* B+ tree index.
- Sorting useful for eliminating *duplicate copies* in a collection of records (Why?)
- *Sort-merge* join algorithm involves sorting.
- Problem: sort 1Gb of data with 1Mb of RAM.
- Why not virtual memory?

Example:

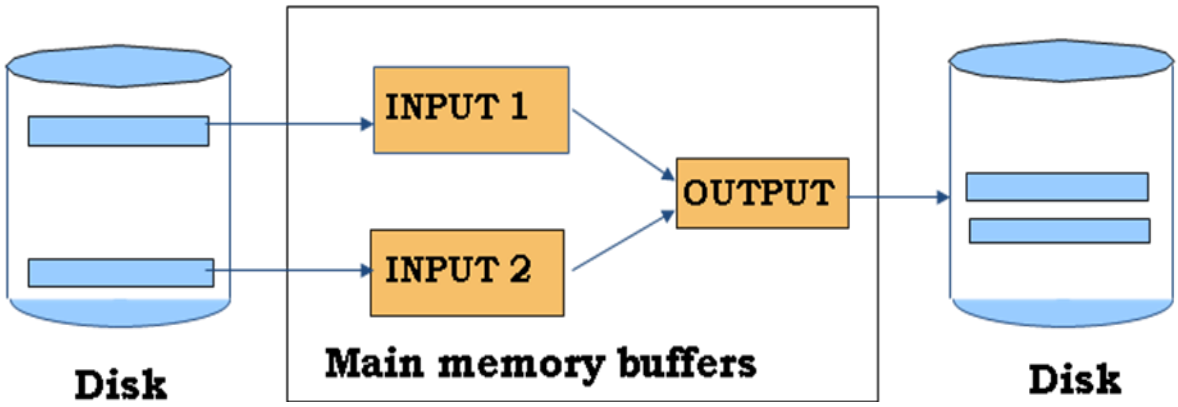
File: 4500 records, A1, …, A4500
internal memory: 750 records (3 blocks)
block length: 250 records
input disk vs. scratch pad (disk)

1. **Sort three blocks at a time and write them out onto scratch pad**
2. **Three blocks: two input buffers & one output buffer**



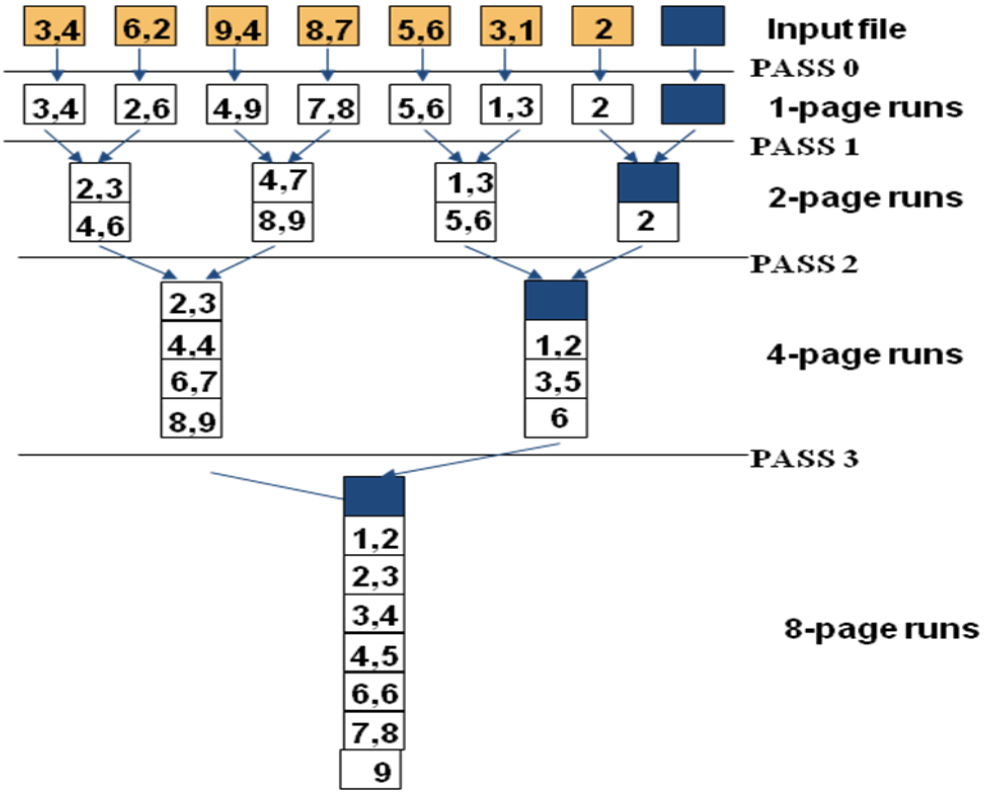
2-Way Sort: Requires 3 Buffers

- Pass 1: Read a page, sort it, write it.
  - only one buffer page is used
- Pass 2, 3, …, etc.:
  - three buffer pages used.



Two-Way External Merge Sort

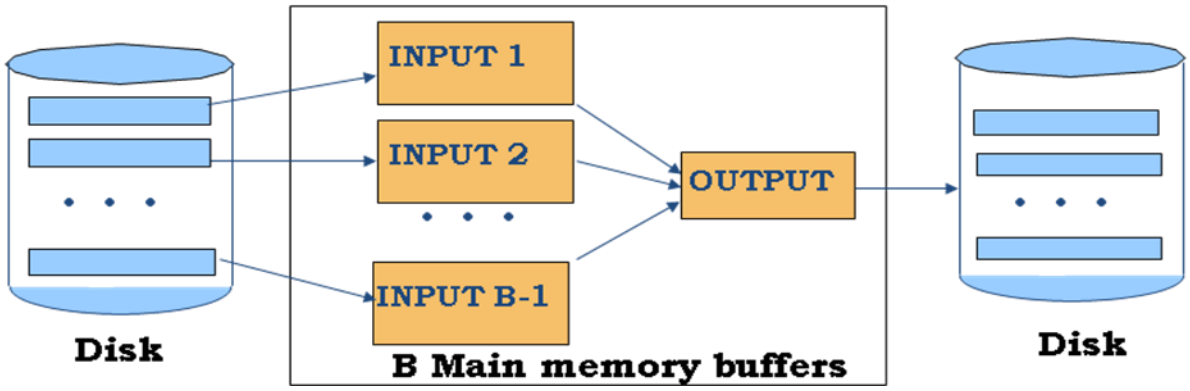
- Each pass we read + write each page in file.
- N pages in the file => the number of passes
  $= \lceil \log_2 N \rceil + 1$
- So total cost is:
- $2N(\lceil \log_2 N \rceil + 1)$
- Idea: **Divide and conquer:** sort sub files and merge



General External Merge Sort

- **More than 3 buffer pages. How can we utilize them?**
  - To sort a file with *N* pages using *B* buffer pages:

- Pass 0: use B buffer pages. Produce $\lceil N / B \rceil$ sorted runs of B pages each.
- Pass 2, …, etc.: merge $B-1$ runs.



Cost of External Merge Sort

- Number of passes: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Cost = 2N * (# of passes)
- E.g., with 5 buffer pages, to sort 108 page file:
  - Pass 0: $\lceil 108 / 5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: $\lceil 22 / 4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages

Time Complexity of External Sort

- input/output time
- *ts* = maximum seek time
- *tl* = maximum latency time
- *trw* = time to read/write one block of 250 records
- *tIO* = *ts* + *tl* + *trw*
- cpu processing time
- *tIS* = time to internally sort 750 records
- *ntm* = time to merge *n* records from input buffers to the output buffer

	Operation	time
(1)	**read 18 blocks of input , 18*tIO*, internally sort, 6*tIS*, write 18 blocks, 18*tIO***	**36 *tIO* +6 *tIS***
(2)	**merge runs 1-6 in pairs**	**36 *tIO* +4500 *tm***
(3)	**merge two runs of 1500 records each, 12 blocks**	**24 *tIO* +3000 *tm***
(4)	**merge one run of 3000 records with one run of 1500 records**	**36 *tIO* +4500 *tm***
	Total Time	**96 *tIO* +12000 *tm*+6 *tIS***

Fig: Critical factor: number of passes over the data
Runs: m, pass: $\lceil \log_2 m \rceil$
Number of Passes of External Sort

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

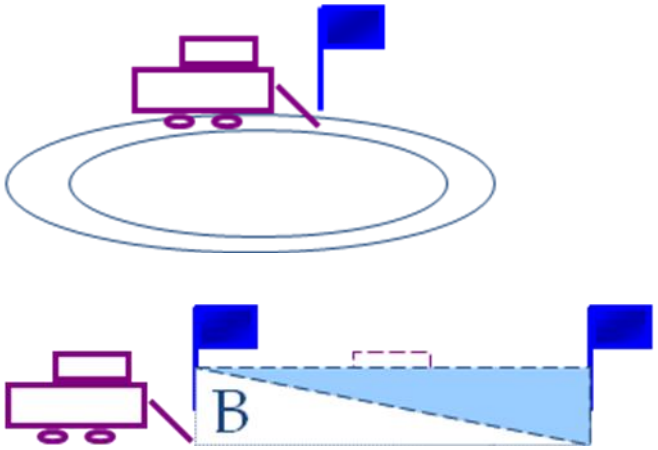
Internal Sort Algorithm

- Quicksort is a fast way to sort in memory.
- An alternative is "tournament sort" (a.k.a. "heapsort")
  - **Top:** Read in ***B*** blocks
  - **Output:** move smallest record to output buffer
  - Read in a new record *r*
  - insert *r* into "heap"
  - if *r* not smallest, then **GOTO Output**
  - else remove *r* from "heap"
  - output "heap" in order; **GOTO Top**

More on Heapsort

- Fact: average length of a run in heapsort is *2B*

- The "snowplow" analogy
- Worst-Case:
  - What is min length of a run?
  - How does this arise?
- Best-Case:
  - What is max length of a run?
  - How does this arise?



- Quicksort is faster, but…

- **I/O for External Merge Sort**

  - … longer runs often means fewer passes!
  - Actually, do I/O a page at a time
  - In fact, read a *block* of pages sequentially!
  - Suggests we should make each buffer (input/output) be a *block* of pages.
  - But this will reduce fan-out during merge passes!
  - In practice, most files still sorted in 2-3 passes.

Number of Passes of Optimized Sort

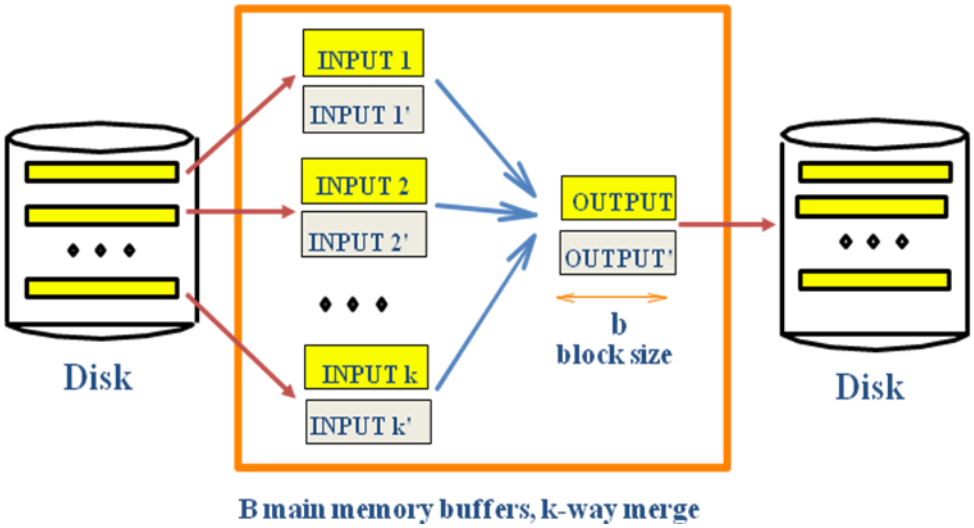
N	B=1,000	B=5,000	B=10,000
100	**1**	**1**	**1**
1,000	**1**	**1**	**1**
10,000	**2**	**2**	**1**
100,000	**3**	**2**	**2**
1,000,000	**3**	**2**	**2**
10,000,000	**4**	**3**	**3**
100,000,000	**5**	**3**	**3**
1,000,000,000	**5**	**4**	**3**

- *Block size = 32, initial pass produces runs of size 2B.*

Double Buffering

- To reduce wait time for I/O request to complete, can *prefetch* into `shadow block'.

- Potentially, more passes; in practice, most files *still* sorted in 2-3 passes.



B main memory buffers, k-way merge

Sorting Records!

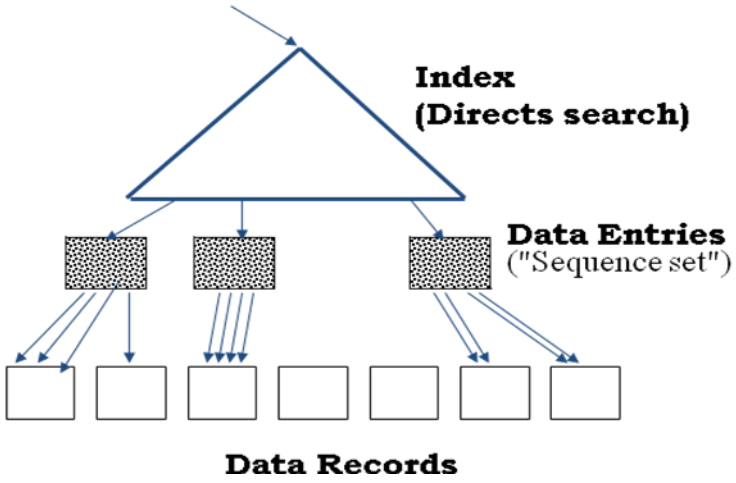
- Sorting has become a blood sport!
  - Parallel sorting is the name of the game ...
- Datamation: Sort 1M records of size 100 bytes
  - Typical DBMS: 15 minutes
  - World record: 3.5 ***seconds***
    - 12-CPU SGI machine, 96 disks, 2GB of RAM
- New benchmarks proposed:
  - Minute Sort: How many can you sort in 1 minute?
  - Dollar Sort: How many can you sort for $1.00?

Using B+ Trees for Sorting

- **Scenario:** Table to be sorted has B+ tree index on sorting column(s).
- **Idea:** Can retrieve records in order by traversing leaf pages.
- ***Is this a good idea?***
- Cases to consider:
  - B+ tree is **clustered** ***Good idea!***
  - B+ tree is **not clustered** ***Could be a very bad idea!***

Clustered B+ Tree Used for Sorting

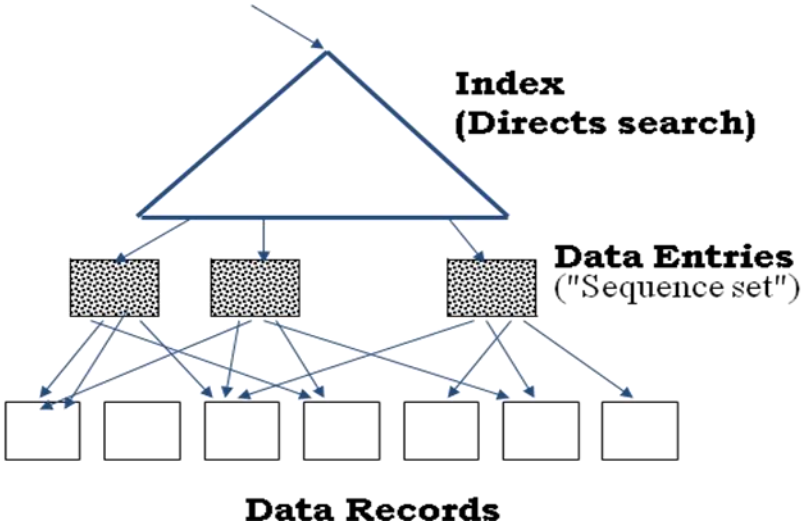
- Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)
- If Alternative 2 is used? Additional cost of retrieving data records: each page fetched just once.



- ***Always better than external sorting***

Un-clustered B+ Tree Used for Sorting

- Alternative (2) for data entries; each data entry contains *rid* of a data record. In general, one I/O per data record!



Summary

- External sorting is important; DBMS may dedicate part of buffer pool for sorting!
- External merge sort minimizes disk I/O cost:
  - Pass 0: Produces sorted *runs* of size *B* (# buffer pages). Later passes: *merge* runs.
  - # of runs merged at a time depends on *B,* and *block size.*
  - Larger block size means less I/O cost per page.
  - Larger block size means smaller # runs merged.
  - In practice, # of runs rarely more than 2 or 3.
- Choice of internal sort algorithm may matter:

- Quicksort: Quick!
- Heap/tournament sort: slower (2x), longer runs

- The best sorts are wildly fast:
  - Despite 40+ years of research, we're still improving!
- Clustered B+ tree is good for sorting; unclustered tree is usually very bad.

Homework

- **Problem 1.** Read the paper:
  - External memory algorithms and data structures: dealing with massive data.
  - Survey by Jeff Vitter at Duke Univ.
  - Www.cs.duke.edu/~jsv/Papers/catalog/node38.html#Vit:I_Osurvey.

- **Problem 2.** Describe the external memory computing model from the paper above briefly. Summarize the status of various sorting algorithms on this model.

- **Problem 3.** Overcoming memory bottlenecks in suffix tree construction.
  - Farach-Colton, Ferragina and Muthukrishnan
  - www.cs.rutgers.edu/~muthu
  - Summarize the "block streaming model" from this paper. What conclusions do they reach about sorting?

BLOCK 5 GRAPHS

In this chapter we discuss several common problems in graph theory. Not only are these algorithms useful in practice, they are interesting because in many real- life applications they are too slow unless careful attention is paid to the choice of data structures. We will

- Show several real-life problems, which can be converted to problems on graphs.
- Give algorithms to solve several common graph problems.
- Show how the proper choice of data structures can drastically reduce the running time of these algorithms.
- See an important technique, known as depth-first search, and show how it can be used to solve several seemingly nontrivial problems in linear time.

This block consist of the following units:

Unit 1: Topological Sort
Unit 2: Path Algorithms
Unit 3: Prim's Algorithm
Unit 4: Undirected Graphs – Bi-connectivity

Unit 5.1: Topological Sort

CONTENTS

5.1.0 AIMS AND OBJECTIVES

At the end of this lesson, students should be able to demonstrate appropriate skills, and show an understanding of the following:

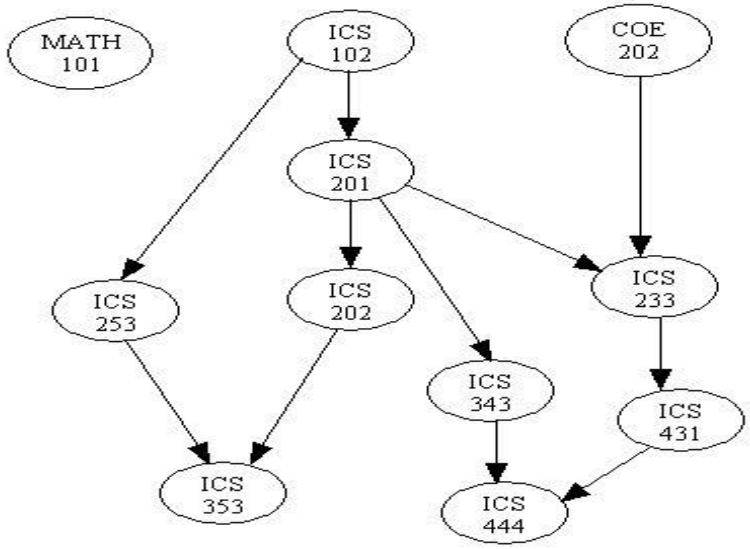
- Aims and objectives of Topological sort
- Introduction
- Definition of Topological Sort.
- Topological Sort is Not Unique.

- Topological Sort Algorithm.
- An Example.
- Implementation.

5.1.1 Introduction

There are many problems involving a set of tasks in which some of the tasks must be done before others. For example, consider the problem of taking a course only after taking its prerequisites.

Is there any systematic way of linearly arranging the courses in the order that they should be taken?

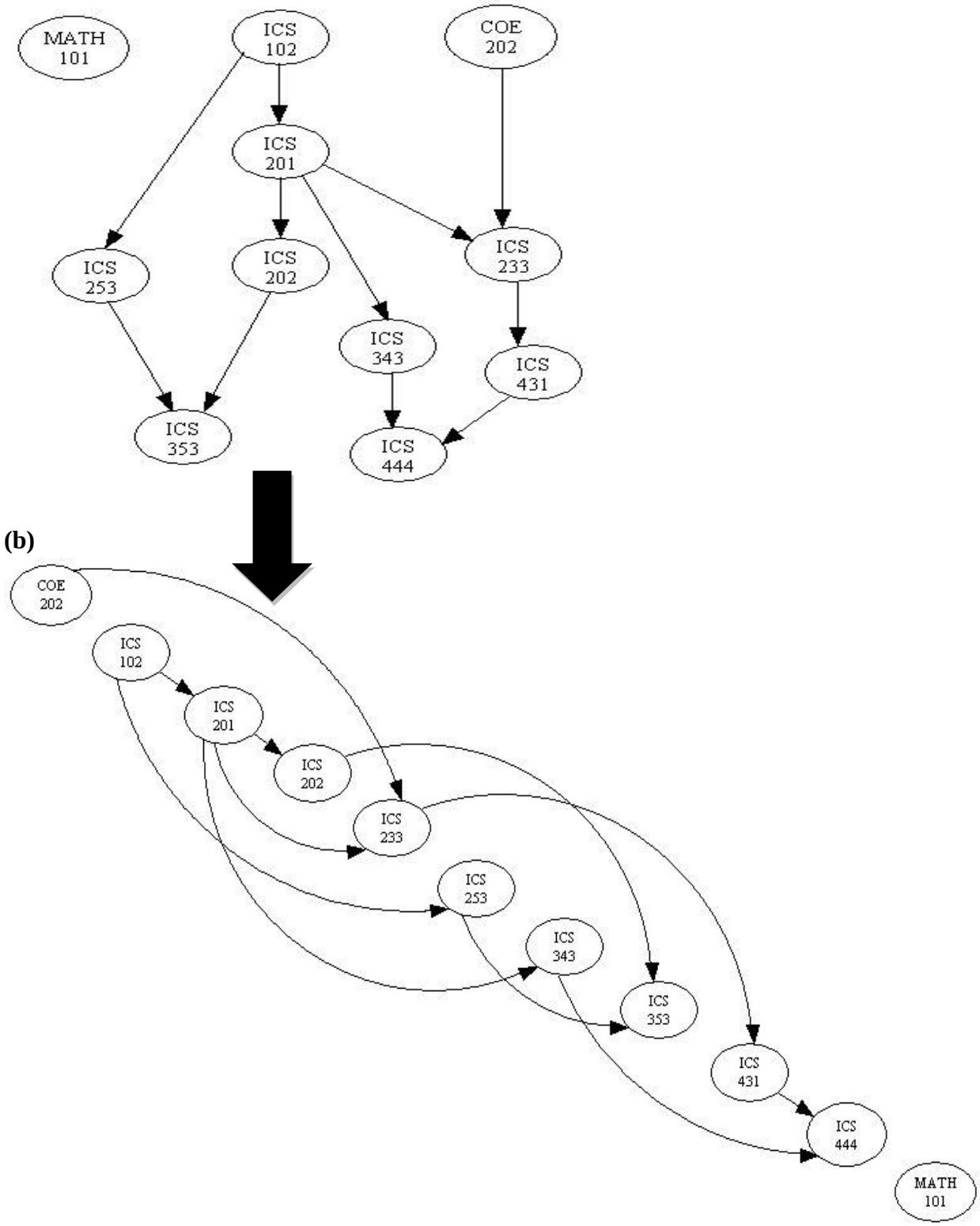


Yes! - Topological sort.

5.1.2 Definition of Topological Sort

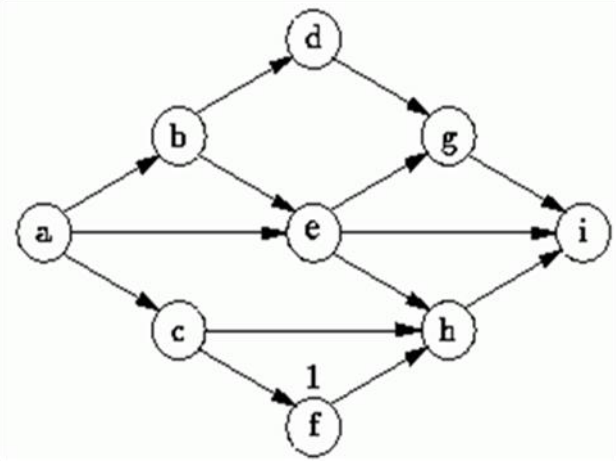
- Topological sort is a method of arranging the vertices in a directed acyclic graph (DAG), as a sequence, such that no vertex appears in the sequence before its predecessor.
- The graph in (a) can be topologically sorted as in (b)

(a)



5.1.3 Topological Sort is not unique

- Topological sort is not unique.
- The following are all topological sort of the graph below:



s1 = {a, b, c, d, e, f, g, h, i}

s2 = {a, c, b, f, e, d, h, g, i}

s3 = {a, b, d, c, e, g, f, h, i}

s4 = {a, c, f, b, e, h, d, g, i}

etc.

5.1.4 Topological Sort Algorithm

Simple algorithms for finding a topological order

- **topSort1**
  - Find a vertex that has no successor
  - Remove from the graph that vertex and all edges that lead to it, and add the vertex to the beginning of a list of vertices
  - Add each subsequent vertex that has no successor to the beginning of the list
  - When the graph is empty, the list of vertices will be in topological order
- **topSort2**
  - A modification of the iterative DFS algorithm
  - Strategy
    - Push all vertices that have no predecessor onto a stack
    - Each time you pop a vertex from the stack, add it to the beginning of a list of vertices
    - When the traversal ends, the list of vertices will be in topological order
- One way to find a topological sort is to consider in-degrees of the vertices.
- The first vertex must have in-degree zero -- every DAG must have at least one vertex with in-degree zero.

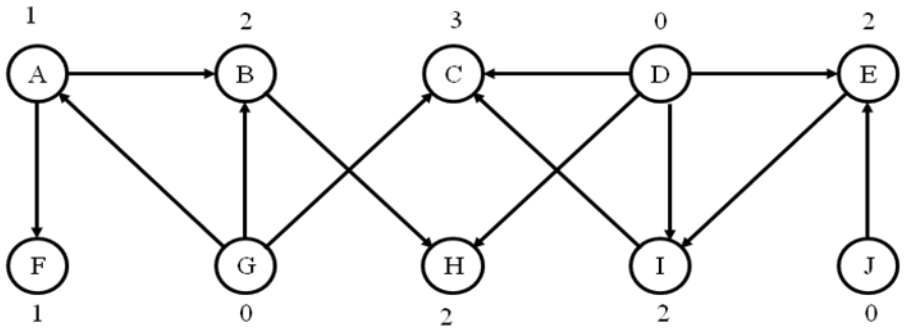
The Topological sort algorithm is:

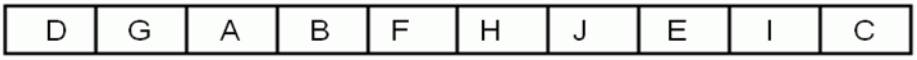
```
int topologicalOrderTraversal( ){
int numVisitedVertices = 0;
while(there are more vertices to be visited){
if(there is no vertex with in-degree 0)
break;
else{
select a vertex v that has in-degree 0;
```

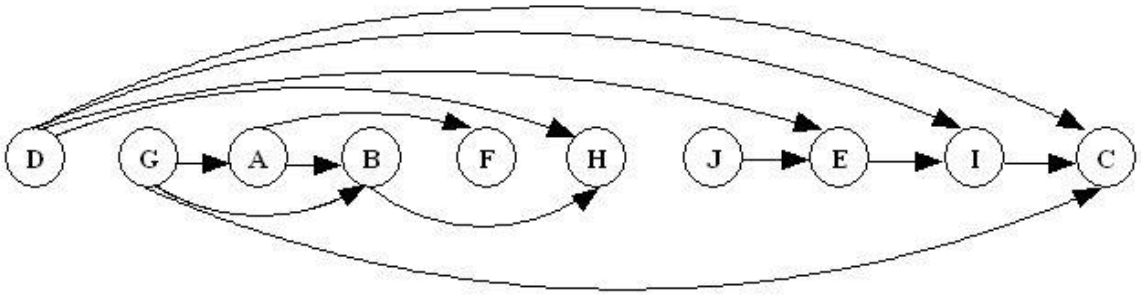
```
visit v;
numVisitedVertices++;
delete v and all its emanating edges;
}
}
return numVisitedVertices;
}
```

5.1.5 Topological Sort Example

- Demonstrating Topological Sort.







5.1.6 Applications of Graphs: Topological Sorting

- Topological order
  - A list of vertices in a directed graph without cycles such that vertex x precedes vertex y if there is a directed edge from x to y in the graph
  - There may be several topological orders in a given graph
- Topological sorting
  - Arranging the vertices into a topological order

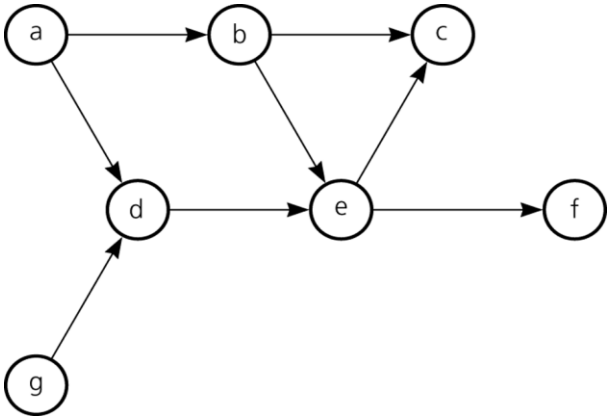


Figure: A directed graph without cycles

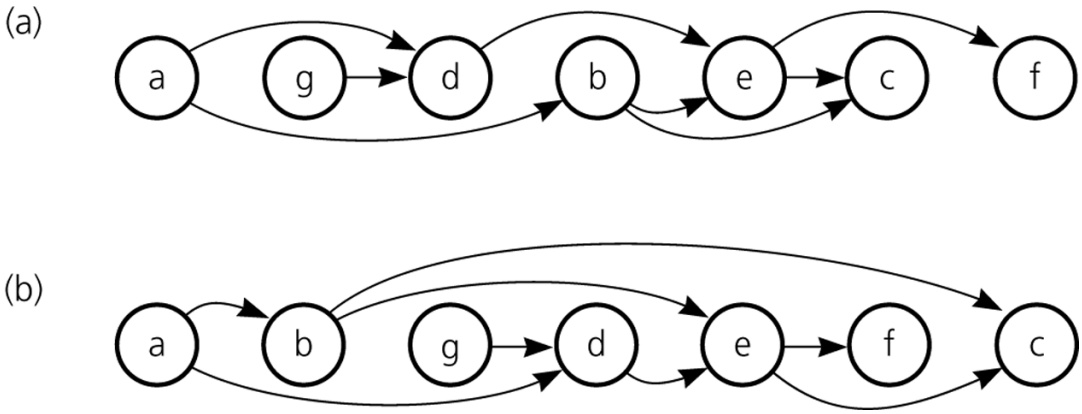


Figure: The figure is arranged according to the topological orders
(a) *a, g, d, b, e, c, f* and (b) *a, b, g, d, e, f, c*

5.1.7 Implementation of Topological Sort

- The algorithm is implemented as a traversal method that visits the vertices in a topological sort order.
- An array of length |V| is used to record the in-degrees of the vertices. Hence no need to remove vertices or edges.
- A priority queue is used to keep track of vertices with in-degree zero that are not yet visited.

For Example:

```
public int topologicalOrderTraversal(Visitor visitor){
int numVerticesVisited = 0;
int[ ] inDegree = new int[numberOfVertices];
for(int i = 0; i < numberOfVertices; i++)
inDegree[i] = 0;
```

```
Iterator p = getEdges();
while (p.hasNext()) {
Edge edge = (Edge) p.next();
Vertex to = edge.getToVertex();
inDegree[getIndex(to)]++;
}
BinaryHeap queue = new BinaryHeap(numberOfVertices);
p = getVertices();
while(p.hasNext()){
Vertex v = (Vertex)p.next();
if(inDegree[getIndex(v)] == 0)
queue.enqueue(v);
}

while(!queue.isEmpty() && !visitor.isDone()){
Vertex v = (Vertex)queue.dequeueMin();
visitor.visit(v);
numVerticesVisited++;
p = v.getSuccessors();
while (p.hasNext()){
Vertex to = (Vertex) p.next();
if(--inDegree[getIndex(to)] == 0)
queue.enqueue(to);
}
}
return numVerticesVisited;
}
```

5.1.8 Implementation: Topological Sort Example

- This job consists of 10 tasks with the following precedence rules:
- Must start with 7, 5, 4 or 9.
- Task 1 must follow 7.
- Tasks 3 & 6 must follow both 7 & 5.
- 8 must follow 6 & 4.
- Must follow 4.
- 10 must follow 2.

Make a directed graph and then a list of ordered pairs that represent these relationships.

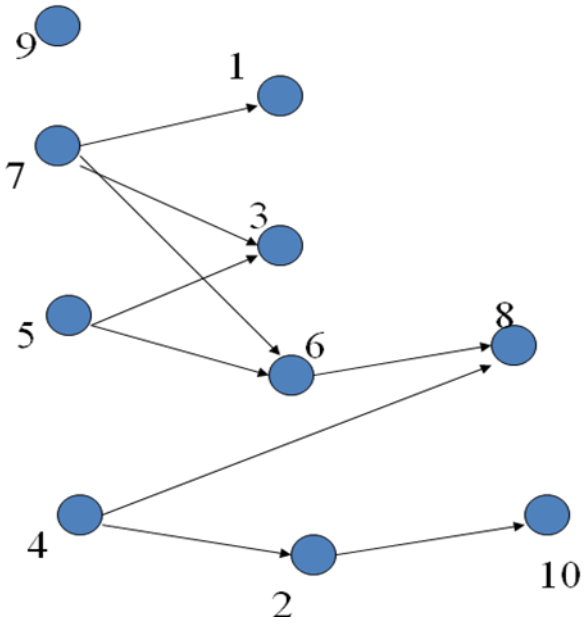


Fig: Tasks shown as a directed graph.

Tasks listed as ordered pairs:

7,1 7,3 7,6 5,3 5,6 6,8 4,8 4,2 2,10

Predecessor Counts

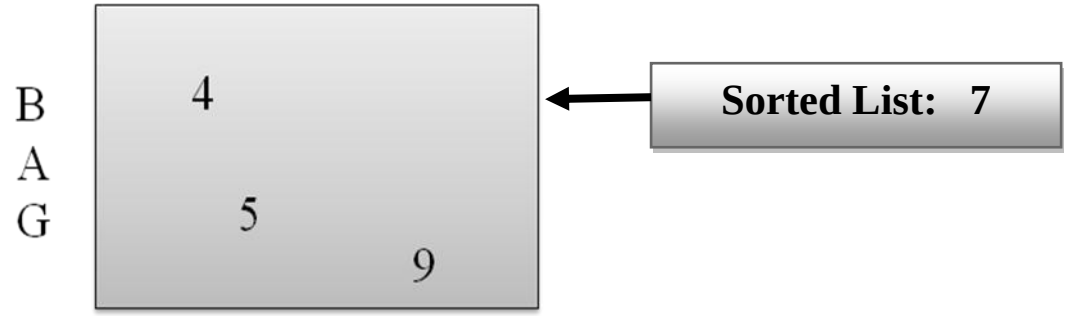
1	1
2	1
3	2
4	0
5	0
6	2
7	0
8	2
9	0
10	1

Successors

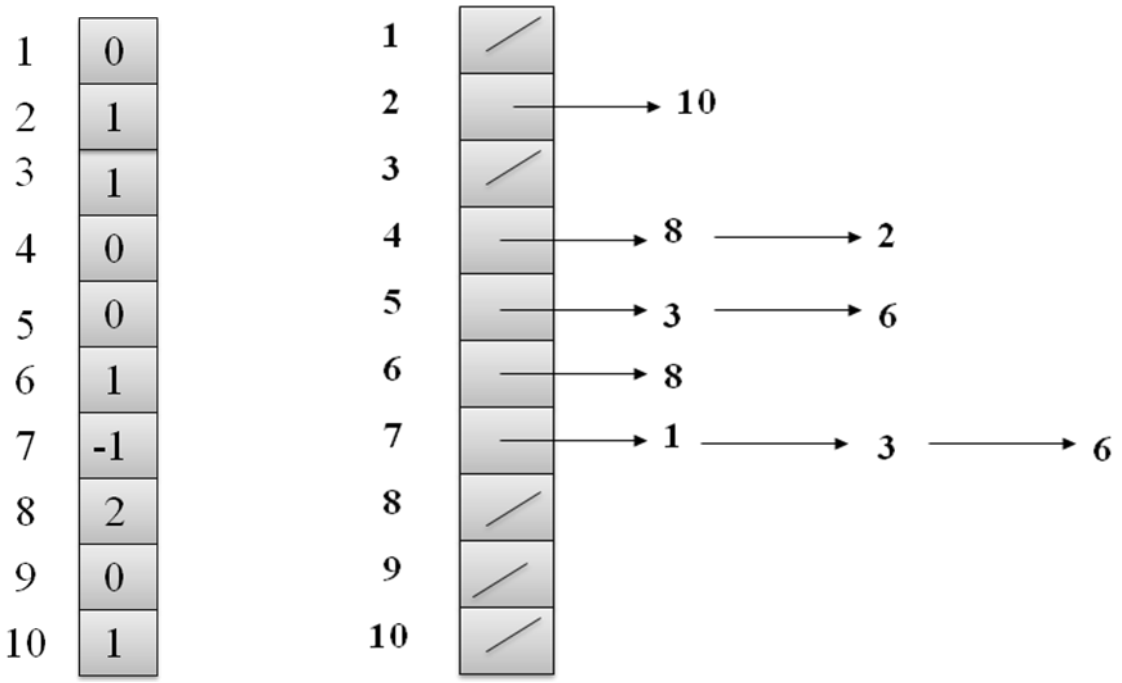
1	
2	10
3	
4	8 → 2
5	3 → 6
6	8
7	1 → 3 → 6
8	
9	
10	

Place all tasks that have zero predecessors in a "bag".

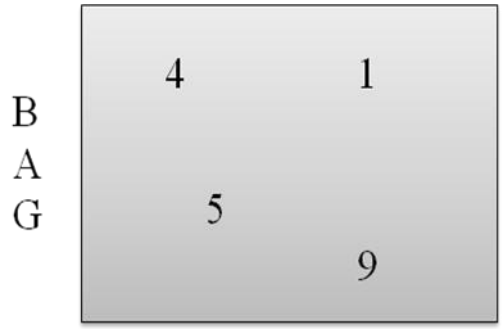
Step 1: Remove any task from the bag and place in sorted list.



Step 2: Update Predecessor and Successor arrays as needed.



Step 3: Add to bag any tasks that now have zero predecessors.



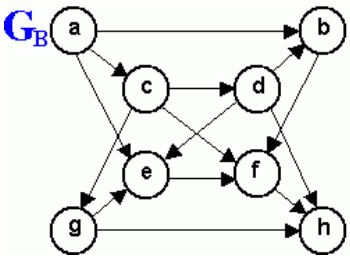
Step 4: Repeat steps 1, 2, and 3 until all predecessor counts are -1.
(We are performing a "**loop"** or **repetition**.)

5.1.9 Let us Sum Up

- **Topological sorting produces a linear order of the vertices in a directed graph without cycles**

5.1.12 Questions for Discussion:

1. List the order in which the nodes of the directed graph GB are visited by topological order traversal that starts from vertex a.
2. What kind of DAG has a unique topological sort?
3. Generate a directed graph using the required courses for your major. Now apply topological sort on the directed graph you obtained.



Unit 2: Path Algorithms

CONTENTS		
Sl. No.	**Topics**	**Page No.**
5.2.0	**Aims and Objectives**	
5.2.1	**Introduction to Path Algorithm**	
5.2.2	**Shortest-Path Algorithms**	
5.2.3	**Dijkstra's Algorithm**	
5.2.4	**Single-Source Shortest Path on Unweighted Graphs**	
5.2.5	**Single-Source Shortest Path on Weighted Graphs**	
5.2.6	**Correctness of Dijkstra's algorithm**	
5.2.7	**Run time of Dijkstra's algorithm**	
5.2.8	**Graphs with Negative Edge Costs**	
5.2.9	**Acyclic Graphs**	
5.2.10	**All-Pairs Shortest Path**	
5.2.11	**Let us Sum Up**	

5.2.0 AIMS AND OBJECTIVES

At the end of this lesson, students should be able to demonstrate appropriate skills, and show an understanding of the following:

- Aims and objectives of Path Algorithm
- Introduction to Path algorithm
- Shortest- Path Algorithms
- Unweighted shortest paths
- Dijkstra's Algorithm
- Graphs with negative edge costs
- Acyclic graphs and All-Pairs shortest path

5.2.1 Introduction

The main purpose of this study is to evaluate the computational efficiency of optimized shortest path algorithms. Our study establishes the relative order of the shortest path algorithms with respect to their known computational efficiencies is not significantly modified when the algorithms are adequately coded. An important contribution of this study is the inclusion of the re-distributive heap algorithm. In spite of having its best theoretical efficiency, the re-distributive heap algorithm does not emerge as the best method in terms of computational efficiency. The complexity of Dijkstra's algorithm depends heavily on the complexity of the priority queue Q. If this queue is implemented naively (i.e. it is re -ordered at every iteration to find the minimum node), the algorithm performs in O (n2), where n is the number of nodes in the graph. With a real priority queue kept ordered at all times, as we implemented it, the complexity averages O (n log m). The logarithm function stems from the collections class, a red-black tree implementation which performs in O (log (m)).

With this article, we'll revisit the so-called "max-flow" problem, with the goal of making some practical analysis of the most famous augmenting path algorithms. We will discuss several algorithms with different complexity from $O(nm^2)$ to $O(nm\log U)$ and reveal the most efficient one in practice. As we will see, theoretical complexity is not a good indicator of the actual value of an algorithm.

In the first section we remind some necessary definitions and statements of the maximum flow theory. Other sections discuss the augmenting path algorithms themselves. The last section shows results of a practical analysis and highlights the best in practice algorithm. Also we give a simple implementation of one of the algorithms.

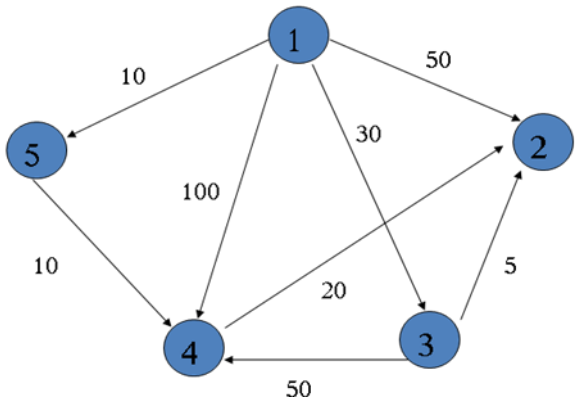
5.2.2 Shortest-Path Algorithms

- Shortest path between two vertices in a weighted graph
  - The path that has the smallest sum of its edge weights

- For a directed graph G=(N,A), each arc (x,y) has associated with it a number d(x,y) that represents the length (previously known as weight) of the arc.
- The length of a path is defined as the sum of the lengths of the individuals' arcs comprising the path.
- The shortest path problem is to determine the path with the minimum path length from s to t.

Given a connected graph G=(V,E), a weight d:E->R+ and a fixed vertex s in V, find a shortest path from s to each vertex v in V.

- Weight of a path $p = \langle v_0, v_1, \ldots, v_n \rangle$
  - $w(p) = \sum_{i=1}^{n} w(v_{i-1}v)$
- Shortest path weight from u to v
  - $\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\mapsto} v\} & \text{if a u, v path exits} \\ \infty & \text{otherwise} \end{cases}$
  - Shortest path from u to v: Any path from u to v with w(p) = d(u,v)
  - p[v]: predecessor of v on a path



Variants

- Single-source shortest paths:
  - find shortest paths from source vertex to every other vertex
- Single-destination shortest paths:
  - find shortest paths to a destination from every vertex
- Single-pair shortest-path
  - find shortest path from u to v
- All pairs shortest paths

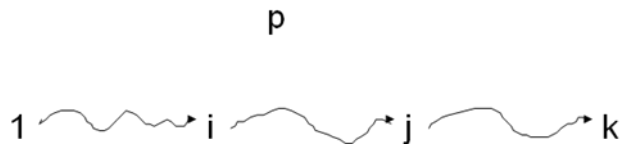
Lemma

Subpaths of shortest paths are shortest paths.

Given G=(G,E) w: E $\rightarrow$ R

Let p = p = $\langle v_1, v_2, \ldots, v_k \rangle$ be a shortest path from v_1 to v_k

For any i,j such that 1 £ i £ j £ k, let p_{ij} be a subpath from v_i to v_j.. Then p_{ij} is a shortest path from v_i to v_j.



Corollary

Let G = (V,E) w: E → R

Suppose shortest path p from a source s to vertex v can be decomposed into

p'

s u v

for vertex u and path p'.

Then weight of the shortest path from s to v is

d(s,v) = d(s,u) + w(u,v)

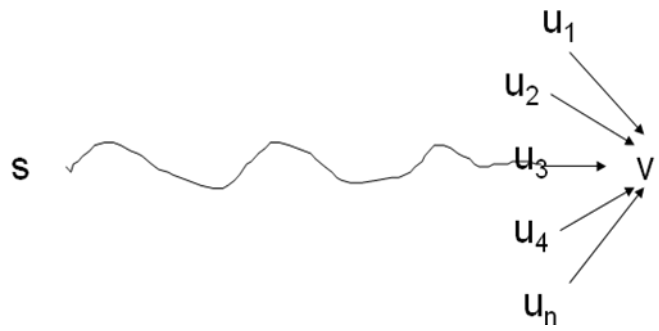
Lemma

Let G = (V,E) w: E → R

Source vertex s

For all edges (u,v) Î E

d(s,v) £ d(s,u) + w(u,v)

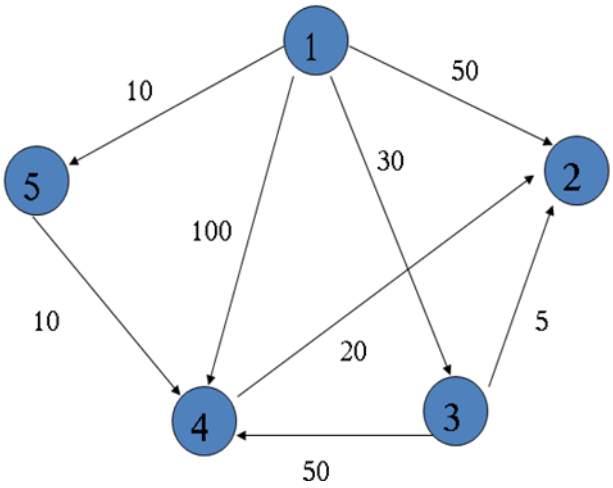


Relaxation

- Shortest path estimate
- d[v] is an attribute of each vertex which is an upper bound on the weight of the shortest path from s to v
- Relaxation is the process of incrementally reducing d[v] until it is an exact weight of the shortest path from s to v

INITIALIZE-SINGLE-SOURCE(G, s)

1. for each vertex v Î V(G)
2. do d[v] ¬ ¥
3. p[v] ¬ nil
4. d[s] ¬ 0



Relaxing an Edge (u,v)

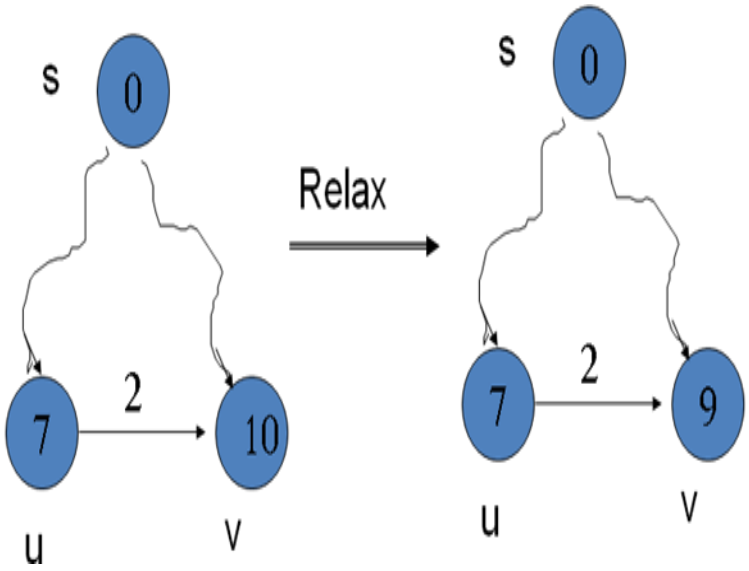
– Question: Can we improve the shortest path to v found so far by going through u?

– If yes, update d[v] and p[v]

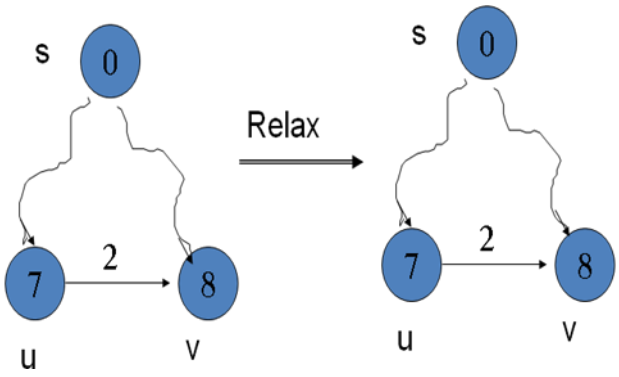
RELAX(u,v,w)

1. if d[v] > d[u] + w(u,v)
2. then d[v] ¬ d[u] + w(u,v)
3. p[v] ¬ u

EXAMPLE 1



EXAMPLE 2



In this section we examine various shortest-path problems.

5.2.3 Dijkstra's Shortest Path Algorithm

Dijkstra's algorithm is probably the best-known and thus most implemented shortest path algorithm. It is simple, easy to understand and implement, yet impressively efficient. By getting familiar with such a sharp tool, a developer can solve efficiently and elegantly problems that would be considered impossibly hard otherwise.

Definition:

- To find the shortest path between points, the weight or length of a path is calculated as the sum of the weights of the edges in the path.
- A path is a shortest path is there is no path from x to y with lower weight.
- Dijkstra's algorithm finds the shortest path from x to y in order of increasing distance from x. That is, it chooses the first minimum edge, stores this value and adds the next minimum value from the next edge it selects.
- It starts out at one vertex and branches out by selecting certain edges that lead to new vertices.
- It is similar to the minimum spanning tree algorithm, in that it is "greedy", always choosing the closest edge in hopes of an optimal solution.
- If no negative edge weights, we can beat BF
- Similar to breadth-first search
  - Grow a tree gradually, advancing from vertices taken from a queue
- Also similar to Prim's algorithm for MST
  - Use a priority queue keyed on d[v]

5.2.3.1 Characteristics of a Shortest Path Algorithm:

Dijkstra's algorithm selects an arbitrary starting vertex and then branches out from the tree constructed so far. Each of the nodes it visits could be in the tree, in the fringes or unseen. The designators:

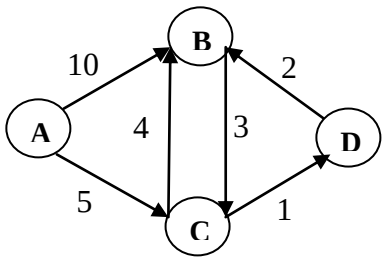
TREE - nodes in the tree constructed so far
FRINGE - not in the tree, but adjacent to some vertex in the tree
UNSEEN - all others

designate for each node in the graph whether the node is part of the **shortest path tree**, a part of the set of fringes adjacent to the nodes in the tree or a part of, as of yet, unseen set of graph

nodes. A crucial step in the algorithm is the **selection of the node from the fringe edge**. The algorithm always takes the edge with **least weight** from the tree to the fringe node.

An Example:

```
Dijkstra(G)
for each v ∈ V
d[v] = ∞;
d[s] = 0; S = ∅; Q = V;
while (Q ≠ ∅)
u = ExtractMin(Q);
S = S U {u};
for each v ∈ u->Adj[]
if (d[v] > d[u]+w(u,v))
d[v] = d[u]+w(u,v);
```



Ex: run the algorithm

Relaxation Step

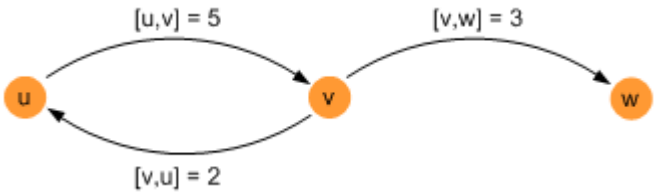
Note: this is really a call to Q->DecreaseKey()

5.2.3.2 What It Does

Dijkstra's algorithm, when applied to a graph, quickly finds the shortest path from a chosen source to a given destination. (The question "how quickly" is answered later in this article.) In fact, the algorithm is so powerful that it finds all shortest paths from the source to all destinations! This is known as the *single-source* shortest paths problem. In the process of finding all shortest paths to all destinations, Dijkstra's algorithm will also compute, as a side-effect if you will, a *spanning tree* for the graph. While an interesting result in itself, the spanning tree for a graph can be found using lighter (more efficient) methods than Dijkstra's.

How It Works

First let's start by defining the entities we use. The graph is made of *vertices* (or nodes, I'll use both words interchangeably), and *edges* which link vertices together. Edges are directed and have an associated *distance*, sometimes called the weight or the cost. The distance between the vertex *u* and the vertex *v* is noted [*u*, *v*] and is always positive.



Dijkstra's algorithm partitions vertices in two distinct sets, the set of *unsettled* vertices and the set of *settled* vertices. Initially all vertices are unsettled, and the algorithm ends once all vertices are in the settled set. A vertex is considered settled, and moved from the unsettled set to the settled set, once its shortest distance from the source has been found.

We all know that *algorithm + data structures = programs*, in the famous words of Niklaus Wirth. The following data structures are used for this algorithm:

d stores the best estimate of the shortest distance from the source to each vertex

π stores the predecessor of each vertex on the shortest path from the source

S the set of settled vertices, the vertices whose shortest distances from the source have been found

Q the set of unsettled vertices

With those definitions in place, a high-level description of the algorithm is deceptively simple. With *s* as the source vertex:

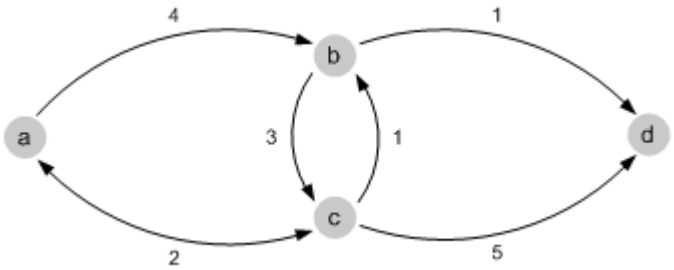
```
// initialize d to infinity, π and Q to empty
d = ( ∞ )
π = ()
S = Q = ()

add s to Q
d(s) = 0

while Q is not empty
{
u = extract-minimum(Q)
add u to S
relax-neighbors(u)
}
```

Dead simple isn't it? The two procedures called from the main loop are defined below:

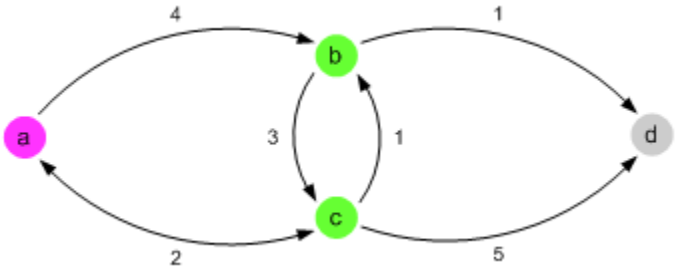
```
relax-neighbors(u)
{
for each vertex v adjacent to u, v not in S
{
if d(v) > d(u) + [u,v]    // a shorter distance exists
{
d(v) = d(u) + [u,v]
π(v) = u
add v to Q
}
}
}

extract-minimum(Q)
{
find the smallest (as defined by d) vertex in Q
remove it from Q and return it
}
```

An Example:

So far I've listed the instructions that make up the algorithm. But to really understand it, let's follow the algorithm on an example. We shall run Dikjstra's shortest path algorithm on the following graph, starting at the source vertex *a*.

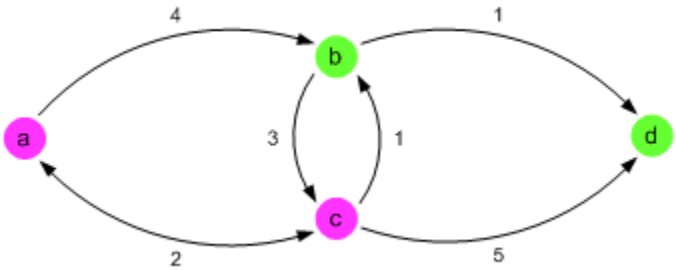


We start off by adding our source vertex *a* to the set *Q*. *Q* isn't empty, we extract its minimum, *a* again. We add *a* to *S*, then relax its neighbors. (I recommend you follow the algorithm in parallel with this explanation.)



Those neighbors, vertices adjacent to *a*, are *b* and *c* (in green above). We first compute the best distance estimate from *a* to *b*. *d*(*b*) was initialized to infinity, therefore we do:
d(b) = d(a) + [a,b] = 0 + 4 = 4

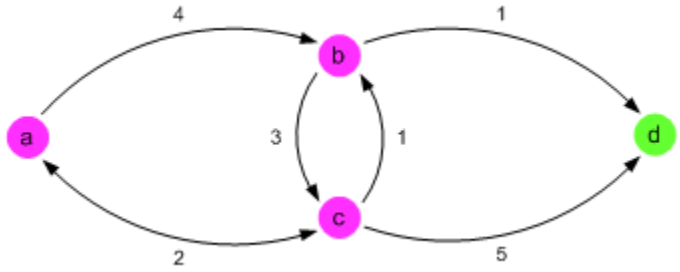
π(*b*) is set to *a*, and we add *b* to *Q*. Similarily for *c*, we assign *d*(*c*) to 2, and *π*(*c*) to *a*. Nothing tremendously exciting so far.
The second time around, *Q* contains *b* and *c*. As seen above, *c* is the vertex with the current shortest distance of 2. It is extracted from the queue and added to *S*, the set of settled nodes. We then relax the neighbors of *c*, which are *b*, *d* and *a*.



a is ignored because it is found in the settled set. But it gets interesting: the first pass of the algorithm had concluded that the shortest path from *a* to *b* was direct. Looking at *c*'s neighbor *b*, we realize that:
d(b) = 4 > d(c) + [c,b] = 2 + 1 = 3

yes! We have found that a shorter path going through *c* exists between *a* and *b*. *d*(*b*) is updated to 3, and *π*(*b*) updated to *c*. *b* is added again to *Q*. The next adjacent vertex is *d*, which we haven't seen yet. *d*(*d*) is set to 7 and *π*(*d*) to *c*.

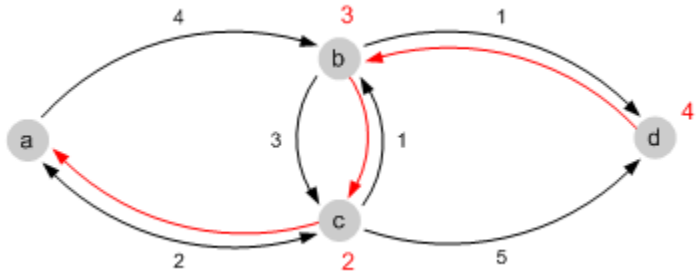
The unsettled vertex with the shortest distance is extracted from the queue, it is now *b*. We add it to the settled set and relax its neighbor's *c* and *d*.



We skip *c*, it has already been settled. But a shorter path is found for *d*:
d(d) = 7 > d(b) + [b,d] = 3 + 1 = 4

Therefore we update $d(d)$ to 4 and $\pi(d)$ to *b*. We add *d* to the Q set.
At this point the only vertex left in the unsettled set is *d*, and all its neighbors are settled. The algorithm ends. The final results are displayed in red below:

- π - the shortest path, in predecessor fashion
- *d* - the shortest distance from the source for each vertex



This completes our description of Dijkstra's shortest path algorithm. Other shortest path algorithms exist (see the References section at the end of this article), but Dijkstra's is one of the simplest, while still offering good performance in most cases.

5.2.4 Single-Source Shortest Path on Unweighted Graphs

Let's consider a simpler problem: solving the single-source shortest path problem for an unweighted directed graph. In this case we are trying to find the smallest number of edges that must be traversed in order to get to every vertex in the graph. This is the same problem as solving the weighted version where all the weights happen to be 1.
Do we know an algorithm for determining this? Yes: breadth-first search. The running time of that algorithm is $O(V+E)$ where V is the number of vertices and E is the number of edges, because it pushes each reachable vertex onto the queue and considers each outgoing edge from it once. There can't be any faster algorithm for solving this problem, because in general the algorithm must at least look at the entire graph, which has size $O(V+E)$.
We saw in recitation that we could express both breadth-first and depth-first search with the same simple algorithm that varied just in the order in which vertices are removed from the queue. We just need an efficient implementation of sets to keep track of the vertices we have visited already. A hash table fits the bill perfectly with its $O(1)$ amortized run time for all

operations. Here is an imperative graph search algorithm that takes a source vertex v_0 and performs graph search outward from it:

```
(* Simple graph traversal (BFS or DFS) *)
let val q: queue = new_queue()
val visited: vertexSet = create_vertexSet()
fun expand(v: vertex) =
let val neighbors: vertex list = Graph.outgoing(v)
fun handle_edge(v': vertex): unit =
if not (member(visited,v'))
then ( add(visited, v');
push(q, v') )
else () )
in
app handle_edge neighbors
end
in
add(visited, v0);
expand(v0);
while (not (empty_queue(q)) do expand(pop(q))
end
```

This code implicitly divides the set of vertices into three sets:

1. The **completed vertices**: visited vertices that have already been removed from the queue.
2. The **frontier**: visited vertices on the queue
3. The **unvisited** vertices: everything else

Except for the initial vertex v_0, the vertices in set 2 are always neighbors of vertices in set 1. Thus, the queued vertices form a frontier in the graph, separating sets 1 and 3. The expand function moves a frontier vertex into the completed set and then expands the frontier to include any previously unseen neighbors of the new frontier vertex.

The kind of search we get from this algorithm is determined by the pop function, which selects a vertex from a queue. If q is a FIFO queue, we do a breadth-first search of the graph. If q is a LIFO queue, we do a depth-first search.

If the graph is unweighted, we can use a FIFO queue and keep track of the number of edges taken to get to a particular node. We augment the visited set to keep track of the number of edges traversed from v_0; it becomes a hash table implementing a map from vertices to edge counts (ints). The only modification needed is in expand, which adds to the frontier a newly found vertex at a distance one greater than that of its neighbor already in the frontier.

```
(* unweighted single-source shortest path *)
let val q: queue = new_queue()
val visited: vertexMap = create_vertexMap()
(* visited maps vertex->int *)
fun expand(v: vertex) =
let val neighbors: vertex list = Graph.outgoing(v)
```

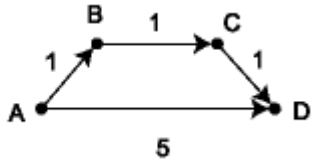
```
val dist: int = valOf(get(visited, v))
fun handle_edge(v': vertex) =
case get(visited, v') of
SOME(d') => ()  (* d' <= dist+1 *)
| NONE => ( add(visited, v', dist+1);
push(q, v') )
in
app handle_edge neighbors
end
in
add(visited, v0, 0);
expand(v0);
while (not (empty_queue(q)) do expand(pop(q))
end
```

5.2.5 Single-Source Shortest Path on Weighted Graphs

Now we can generalize to the problem of computing the shortest path between two vertices in a weighted graph. We can solve this problem by making minor modifications to the BFS algorithm for shortest paths in unweighted graphs. As in that algorithm, we keep a visited map that maps vertices to their distances from the source vertex v_0. We change expand so that Instead of adding 1 to the distance, its adds the weight of the edge traversed. Here is a first cut at an algorithm:

```
let val q: queue = new_queue()
val visited: vertexMap = create_vertexMap()
fun expand(v: vertex) =
let val neighbors: vertex list = Graph.outgoing(v)
val dist: int = valOf(get(visited, v))
fun handle_edge(v': vertex, weight: int) =
case get(visited, v') of
SOME(d') =>
if dist+weight < d'
then add(visited, v', dist+weight)
else ()
| NONE => ( add(visited, v', dist+weight);
push(q, v') )
in
app handle_edge neighbors
end
in
add(visited, v0, 0);
expand(v0);
while (not (empty_queue(q)) do expand(pop(q))
end
```

This is nearly Dijkstra's algorithm, but it doesn't work. To see why, consider the following graph, where the source vertex is v_0 = A.



The first pass of the algorithm will add vertices B and D to the map visited, with distances 1 and 5 respectively. D will then become part of the completed set with distance 5. Yet there is a path from A to D with the shorter length 3. We need two fixes to the algorithm just presented:

1. In the SOME case a check is needed to see whether the path just discovered to the vertex v' is an improvement on the previously discovered path (which had length d)
2. The queue q should not be a FIFO queue. Instead, it should be a *priority queue* where the priorities of the vertices in the queue are their distances recorded in visited. That is, pop(q) should be a priority queue extract_min operation that removes the vertex with the smallest distance.

The priority queue must also support a new operation increase_priority(q,v) that increases the priority of an element v already in the queue q. This new operation is easily implemented for heaps using the same bubbling-up algorithm used when performing heap insertions.

With these two modifications, we have Dijkstra's algorithm:

```
(* Dijkstra's Algorithm *)
let val q: queue = new_queue()
val visited: vertexMap = create_vertexMap()
fun expand(v: vertex) =
let val neighbors: vertex list = Graph.outgoing(v)
val dist: int = valOf(get(visited, v))
fun handle_edge(v': vertex, weight: int) =
case get(visited, v') of
SOME(d') =>
if dist+weight < d'
then ( add(visited, v', dist+weight);
incr_priority(q, v', dist+weight) )
else ()
| NONE => ( add(visited, v', dist+weight);
push(q, v', dist+weight) )
in
app handle_edge neighbors
end
in
add(visited, v0, 0);
expand(v0);
while (not (empty_queue(q)) do expand(pop(q))
```

end

There are two natural questions to ask at this point: Does it work? How fast is it?

5.2.6 Correctness of Dijkstra's algorithm

Each time that expand is called, a vertex is moved from the frontier set to the completed set. Dijkstra's algorithm is an example of a **greedy algorithm**, because it just chooses the closest frontier vertex at every step. A locally optimal, "greedy" step turns out to produce the global optimal solution. We can see that this algorithm finds the shortest-path distances in the graph example above, because it will successively move B and C into the completed set, before D, and thus D's recorded distance has been correctly set to 3 before it is selected by the priority queue.

The algorithm works because it maintains the following two invariants:

> For every completed vertex, the recorded distance (in visited) is the shortest-path distance to that vertex from v_0.
>
> For every frontier vertex *v*, the recorded distance is the shortest-path distance to that vertex from v_0, considering just the paths that traverse only completed vertices and the vertex *v* itself. We will call these paths **internal paths**.

We can see that these invariants hold when the main loop starts, because the only completed vertex is v_0 itself, which has recorded distance 0. The only frontier vertices are the neighbors of v_0, so clearly the second part of the invariant also holds. If the first invariant holds when the algorithm terminates, the algorithm works correctly, because all vertices are completed. We just need to show that each iteration of the main loop preserves the invariants.

Each step of the main loop takes the closest frontier vertex *v* and promotes it to the completed set. For the first invariant to be maintained, it must be the case that the recorded distance for the closest frontier vertex is also the shortest-path distance to that vertex. The second invariant tells us that the only way it could fail to be the shortest-path distance is if there is another, shorter, non-internal path to *v*. Any non-internal path must go through some other frontier vertex *v''* to get to *v*. But this path must be longer than the shortest internal path, because the priority queue ensures that *v* is the closest frontier vertex. Therefore the vertex *v''* is already at least as far away than *v*, and the rest of the path can only increase the length further (note that the assumption of nonnegative edge weights is crucial!).

We also need to show that the second invariant is maintained by the loop. This invariant is maintained by the calls to incr_priority andpush in handle_edge. Promoting v to the completed set may create new internal paths to the neighbors of v, which become frontier vertices if they are not already; these calls ensure that the recorded distances to these neighbors take into account the new internal paths.

We might also be concerned that incr_priority could be called on a vertex that is not in the priority queue at all. But this can't happen because incr_priority is only called if a shorter path has been found to a completed vertex v'. By the first invariant, a shorter path cannot exist.

Notice that the first part of the invariant implies that we can use Dijkstra's algorithm a little more efficiently to solve the simple shortest-path problem in which we're interested only in a particular destination vertex. Once that vertex is popped from the priority queue, the traversal can be halted because its recorded distance is correct. Thus, to find the distance to a

vertex *v* the traversal only visits the graph vertices those are at least as close to the source as *v* is.

5.2.7 Run time of Dijkstra's algorithm

Every time the main loop executes, one vertex is extracted from the queue. Assuming that there are *V* vertices in the graph, the queue may contain *O*(*V*) vertices. Each pop operation takes *O*(lg *V*) time assuming the heap implementation of priority queues. So the total time required to execute the main loop itself is *O*(*V* lg *V*). In addition, we must consider the time spent in the function expand, which applies the functionhandle_edge to each outgoing edge. Because expand is only called once per vertex, handle_edge is only called once per edge. It might call push(v'), but there can be at most *V* such calls during the entire execution, so the total cost of that case arm is at most *O*(*V* lg *V*). The other case arm may be called *O*(*E*) times, however, and each call to increase_priority takes *O*(lg *V*) time with the heap implementation. Therefore the total run time is *O*(*V* lg *V* + *E* lg *V*), which is *O*(*E* lg *V*) because *V* is *O*(*E*) assuming a connected graph.

(There is another more complicated priority-queue implementation called a **Fibonacci heap** that implements increase_priority in *O*(1)time, so that the asymptotic complexity of Dijkstra's algorithm becomes *O*(*V* lg *V* + *E*); however, large constant factors make Fibonacci heaps impractical for most uses.)

5.2.8 Graphs with Negative Edge Costs

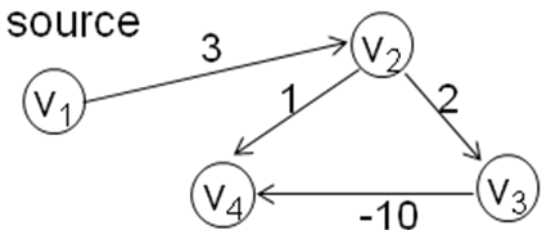
With this algorithm, the graph is still weighted but it is now possible to have an edge with a negative weight. The addition of the negative costs causes Dijkstra's algorithm to fail because once a vertex is processed, it is never processed again. But with a negative cost, it's possible that a path back to a vertex that costs less can be found.

To fix the problem: the weighted and unweighted algorithms will be combined. This drastically increases the running time!

To do this, the vertices will no longer be marked as processed.

1. Start by pushing the start vertex onto a queue
2. Pop a vertex v
3. Find each vertex w that is adjacent to v such that w's cost is greater than v's cost + edge cost. If that is the case, update w's cost and push it onto the queue if it's not already there.
4. repeat from step 2 until the queue is empty

- Will the O(|E| log|V|) Dijkstra's algorithm work as is?



deleteMin	Updates to dist

V_1	**V_2.dist = 3**
V_2	**V_4.dist = 4, V_3.dist = 5**
V_4	**No change**
V_3	**No change and so v_4. dist will remain 4.** **<u>Correct answer:</u> v_4.dist should be updated to -5**

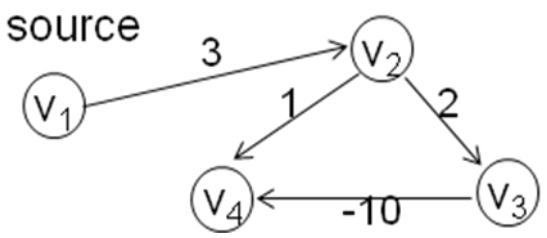
<u>Solution:</u>

Do not mark any vertex as "known".

Instead allow multiple updates.

```
void Graph::weightedNegative( Vertex s )
{
    Queue<Vertex> q;

    for each Vertex v
        v.dist = INFINITY;

    s.dist = 0;
    q.enqueue( s );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );

        for each Vertex w adjacent to v
            if( v.dist + cvw < w.dist )
            {
                // Update w
                w.dist = v.dist + cvw;
                w.path = v;
                if( w is not already in q )
                    q.enqueue( w );
            }
    }
}
```

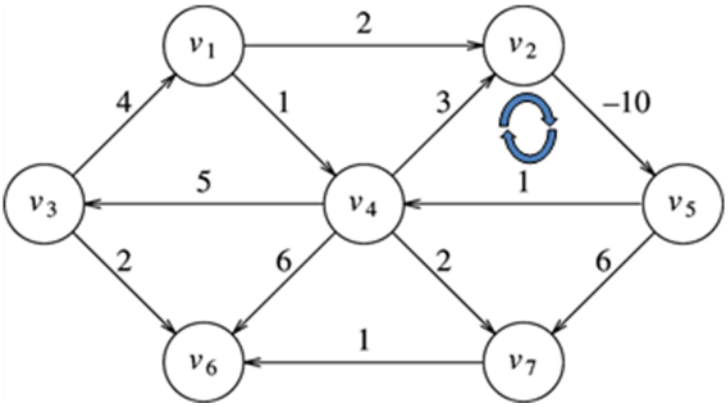
Running time: O(|E|·|V|)



Queue	Dequeue	Updates to dist
V_1	V_1	V_2.dist = 3
V_2	V_2	V_4.dist = 4, V_3.dist = 5
V_4, V_3	V_4	No updates
V_3	V_3	V_4.dist = -5
V_4	V_4	No updates

Running time: O(|E|·|V|)

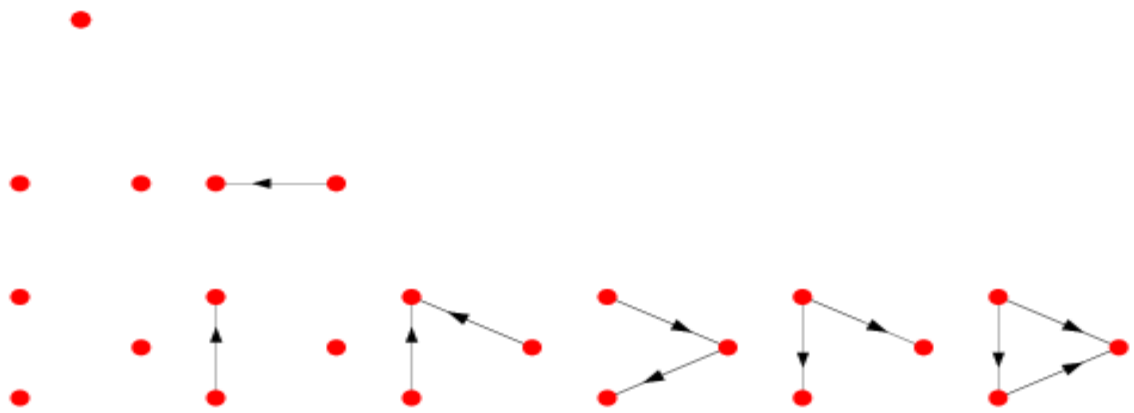
Negative weight cycles?



5.2.9 Acyclic Graphs

Dijkstra's algorithm can be improved if the graph is known to be acyclic. The improvement is made by changing the order that the vertices are processed. It will now be done in a topological order.

This works because once a vertex is selected its cost cannot be lowered since it will no longer have any incoming edges from unprocessed vertices.



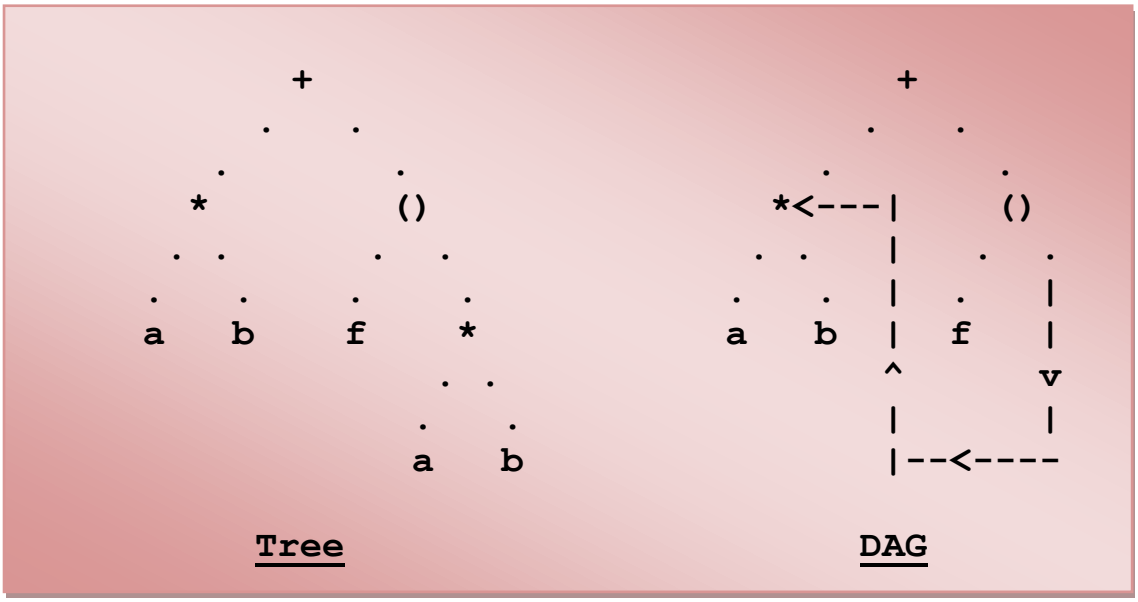
An acyclic digraph is a directed graph containing no directed cycles, also known as a directed acyclic graph or a "DAG." Every finite acyclic digraph has at least one node of outdegree 0. The numbers of acyclic digraphs on $n = 1$, 2 ... vertices are 1, 2, 6, 31, 302, 5984...
The numbers of labeled acyclic digraphs on $n = 1$, 2, ... nodes are 1, 3, 25, 543, 29281, ...Weinstein's conjecture proposed that positive Eigen valued $(0, 1)$-matrices were in one-to-one correspondence with labeled acyclic digraphs on n nodes, and this was subsequently proved by McKay *et al.* (2004). Counts for both are therefore given by the beautiful recurrence equation

$$a_n = \sum_{k=1}^{n} (-1)^{k-1} \binom{n}{k} 2^{k(n-k)} a_{n-k}$$

with $a_0 = 1$.

Direct Acyclic Graph:

A *directed acyclic graph* (DAG!) is a directed graph that contains no cycles. A rooted tree is a special kind of DAG and a DAG is a special kind of directed graph. For example, a DAG may be used to represent common sub expressions in an optimizing compiler.



expression: a*b+f(a*b)

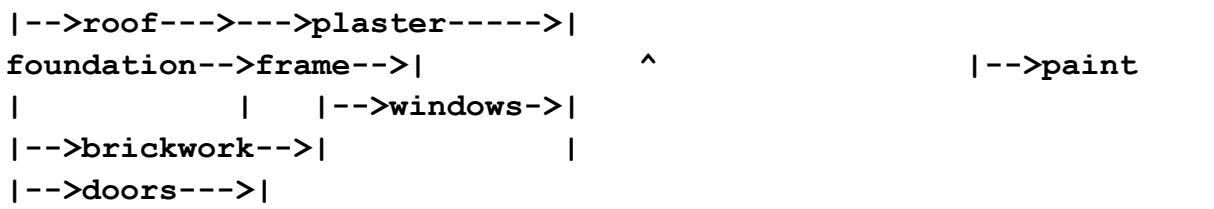
Example of Common Subexpression.

The common sub-expression a*b need only be compiled once but its value can be used twice. A DAG can be used to represent prerequisites in a university course, constraints on operations to be carried out in building construction, in fact an arbitrary partial-order `<'. An edge is drawn from a to b whenever a<b. A partial order `<' satisfies:

(i) transitivity, a<b and b<c implies a<c

(ii) non-reflexive, not(a < a)

These condition prevent cycles because $v_1<v_2<...<v_n<v_1$ would imply that $v_1<v_1$. The word `partial' indicates that not every pair or values are ordered. Examples of partial orders are numerical less-than (also a total order) and `subset-of'; note that {1,2} is a subset of {1,2,3} but that {1,2} and {2,3} are *incomparable*, i.e. there is no order relationship between them.
Constraints for a small building example are given below.



Simplified Construction Constraints

Note that no order is imposed between `roof' and `brick-work', but the plaster cannot be applied until the walls are there for it to stick to and the roof exists to protect it.

5.2.10 All Pairs Shortest Path Problem

Given G(V,E), find a shortest path between all pairs of vertices.

- Assume G=(V,E) is a graph such that c[v,w] ≥ 0, where C is the matrix of edge costs.
- Find for each pair (v,w), the shortest path from v to w. That is, find the matrix of shortest paths
- Certainly this is a generalization of Dijkstra's.
- Note: For later discussions assume |V| = n and |E| = m

Solutions:

(brute-force)

Solve Single Source Shortest Path for each vertex as source

There are more efficient ways of solving this problem (e.g., Floyd-Warshall algo).

The *all-pairs shortest path* problem can be considered the mother of all routing problems. It aims to compute the shortest path from each vertex *v* to every other *u*. Using standard single-source algorithms, you can expect to get a naive implementation of *O(n^3)* if you use Dijkstra for example -- i.e. running a *O(n^2)* process *n* times. Likewise, if you use the Bellman-Ford-Moore algorithm on a dense graph, it'll take about *O(n^4)*, but handle negative arc-lengths too.

Storing all the paths explicitly can be very memory expensive indeed, as you need one spanning tree for each vertex. This is often impractical in terms of memory consumption, so these are usually considered as all-pairs shortest distance problems, which aim to find just the distance from each to each node to another.

The result of this operation is an **n * n** matrix, which stores estimated distances to the each node. This has many problems when the matrix gets too big, as the algorithm will scale very poorly.

Naive Implementation

So, this would give us something like this:

```
/**
* A third-party implementation of the shortest path algorith, that
* uses a common distance matrix to store distances.
*
* This could be Dijkstra, BFM or any of their variants.
*/
void SingleSourceSP( Node* root );

// run the algorithm for each node
for (Node* n = nodes.begin(); n != nodes.end(); n++)
{
SingleSourceSP( n );
}
```

Not very nice, admittedly, but it can work in O(n^3) for non-negative arc lengths. As you will see, its quite tough to beat that.

Optimizations

There are a few simple simplifications that can be made to such an algorithm, which can improve the performance in practice. Though it does not help the theoretical bounds of the algorithm.

If the graph is undirected, then its symmetry properties can be exploited. Only half a matrix is required. This is fairly simple to implement in practice, as you can wrap your matrix inside an oriented object class, and treat any operation on the cell (i,j) as (j,i) if j<="" p="">

During the graph traversal, the naive implementation cannot assume much from the distances that have already been computed. It can use them as heuristic, but little more.

Floyd-Warshall

The Floyd Warshal algorithm takes the dynamic programming approach. This essentially means that independent sub-problems are solved and the results are stored for later use. The algorithm allows negative edges, but no negative cycles, as per usual with such shortest path problems.

The basic concept is simple. If, for a path *(u,v)* and its length estimate D[u][v], if you can take a detour via w and shorten the path, then it should be taken. This translates into the following equation:

D[u][v] = min(D[u][v], D[u][w], D[w][v])

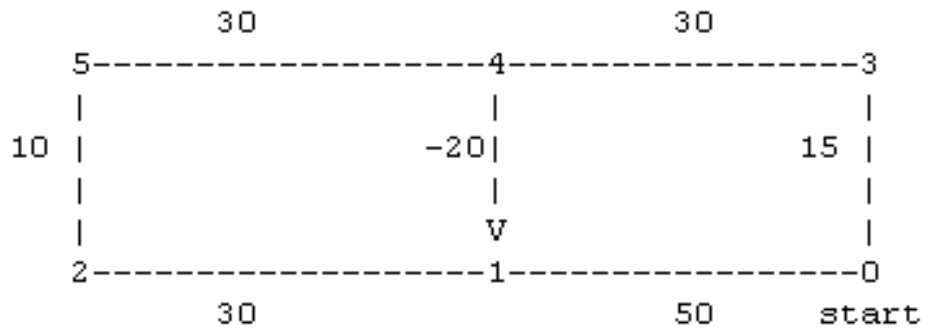
The easiest way to calculate this is bottom up. You start with basic assumptions that can be made from the graphs connectivity, and correct those assumptions "n" times for each vertex pair and the results will converge.

```
/**
* Initialise the original estimates from nodes and edges.
*/
for (int i=0; i<nodes; i++)
{
```

```
D[i][i] = 0.0f;
}

for (Edge* e = edges->begin(); e != edges->end(); e++)
{
D[e->start][e->end] = e->length;
}

/**
* And run the main loop of the algorithm.
*/
for (int k=0; k<nodes.size(); k++)
{
for (int i=0; i<nodes.size(); i++)
{
for (int j=0; j<nodes.size(); j++)
{
D[i][j] = min( D[i][j], D[i][k] + D[k][j] );
}
}
}
```

Floyd's Algorithm

- **A[i][j] = C(i,j) if there is an edge (i,j)**
- **A[i][j] = infinity(inf) if there is no edge (i,j)**



Graph

```
          0    1    2    3    4    5
        +-----------------------------+
      0 | 0   50   inf  15   inf  inf  |
      1 | 50   0   30   inf  inf  inf  |
A  =  2 |inf  30    0   inf  inf   10  |
      3 | 15  inf  inf   0    30  inf  |
      4 |inf  -20  inf   30   0    30  |
      5 |inf  inf   10  inf   30   0   |
        +-----------------------------+
```

"Adjacency" matrix

A is the shortest path matrix that uses 1 or fewer edges

- To find shortest paths that uses 2 or fewer edges find A^2, where multiplication defined as *min of sums* instead sum of products
- That is $(A^2)_{ij} = \min\{ A_{ik} + A_{kj} \mid k = 1..n\}$
- This operation is $O(n^3)$
- Using A^2 you can find A^4 and then A^8 and so on
- Therefore to find A^n we need log n operations
- Therefore this algorithm is $O(\log n * n^3)$
- We will consider another algorithm next

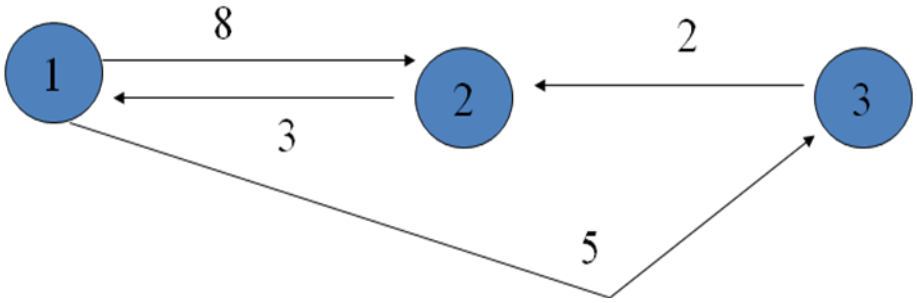
Floyd-Warshall Algorithm

- This algorithm uses nxn matrix A to compute the lengths of the shortest paths using a dynamic programming technique.
- Let A[i,j] = c[i,j] for all i,j & $i \neq j$
- If (i,j) is not an edge, set A[i,j]=infinity and A[i,i]=0
- $A_k[i,j] = \min (A_{k-1}[i,j] , A_{k-1}[i,k] + A_{k-1}[k,j])$

Where A_k is the matrix after k-th iteration and path from i to j does not pass through a vertex higher than k

Example – Floyd-Warshall Algorithm

Find the all pairs shortest path matrix



- **$A_k[i,j] = \min (A_{k-1}[i,j] , A_{k-1}[i,k] + A_{k-1}[k,j])$**

Where A_k is the matrix after k-th iteration and path from i to j does not pass through a vertex higher than k

Floyd-Warshall Implementation

- initialize A[i,j] = C[i,j]
- initialize all A[i,i] = 0
- for k from 1 to n

```
for i from 1 to n
for j from 1 to n
if (A[i,j] > A[i,k]+A[k,j])
A[i,j] = A[i,k]+A[k,j];
```

- The complexity of this algorithm is $O(n^3)$

Johnson's Algorithm

Johnson has defined an algorithm that takes advantage of the sparse nature of most graphs, with directed edges and no cycles. With these assumptions, he gets better results than the naive algorithm.

The first pass consists of adding a new vertex with zero length arcs to every other node, and running the Bellman-Ford-Moore algorithm thereon. This can detect negative cycles, as well as find *d(u)*, which is the final estimated distance from the new node to vertex *u*. Then, the edges are "reweighted" based on these values, and Dijkstras algorithm is run for each of the nodes. The final weight is adjusted to take into account the reweighting process.

The time complexity is officially O(n^2 log n + mn), though improvements to both Dijkstra and BFM can be used to improve that by a fair amount.

The original paper (Efficient algorithms for shortest paths in sparse graphs, 77) is quite hard to find, so check the book "Introduction to Algorithm" (Cormen Thomas, MIT Press) for more details.

Challenges of Sub-cubic Implementations

Without the assumptions made by Johnson, its difficult to break the O(n^3) boundary. The problem has been shown to be at least as hard as Boolean matrix multiplication, and mappings from one problem to the other can be performed at a small cost.

That said, implementations that are combinatorial in nature, rather than matrix based has advantages in some cases (like Johnson assumed), may have many benefits. This has become a challenge for many researchers, and some nice parallel algorithms have arisen.

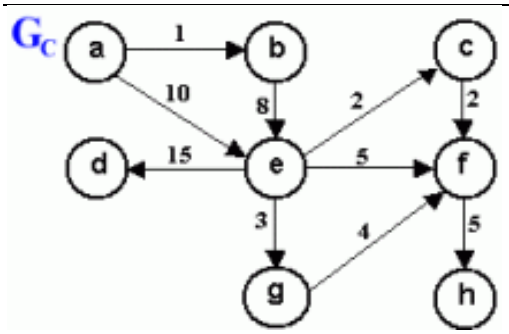
Almost Short Enough

Often, approximations are cheaper to compute than exact values, and some recent work has been done on such approximations. These allow breaking of the sub-cubic barrier, but at a cost, naturally; only approximations of the path lengths are returned.

Stretch-t paths are defined as having a length of at most t times the distance between its ends. For paths of stretch (4 + e) you can get a running time of *O(n^(5/2) polylog n)* -- which doesn't mean much in theory, does it?

Dor, Halperin and Zwick (in "All pairs almost shortest path", 97) propose an efficient algorithm for stretch 3 paths. I won't go into it here for fear of confusing you, and myself in the process, but bare in mind that it exists

5.2.13 Questions for Discussion



- Use the graph Gc shown above to trace the execution of Dijkstra's algorithm as it solves the shortest path problem starting from vertex a.
- Dijkstra's algorithm works as long as there are no negative edge weights. Given a graph that contains negative edge weights, we might be tempted to eliminate the negative weights by adding a constant weight to all of the edges. Explain why this does not work.
- Dijkstra's algorithm can be modified to deal with negative edge weights (but not negative cost cycles) by eliminating the known flag and by inserting a vertex back into the queue every time its tentative distance decreases. Implement this modified algorithm.
- What is the asymptotic run time of Dijkstra (adjacency matrix version)?
  - $O(n^2)$
- What is the asymptotic running time of Floyd-Warshall?

References:

http://skylight.wsu.edu/s/053eadf6-6157-44ce-92ad-cbc26bde3b53.srv

Unit 3: Prim's Algorithms

CONTENTS

5.3.0 AIMS AND OBJECTIVES

At the end of this lesson, students should be able to demonstrate appropriate skills, and show an understanding of the following:

- Aims and objectives of Prims Algorithm
- Introduction
- Definition
- Implementation of Prims Algorithms
- Invariants of Prims algorithm
- Running time of Prims algorithm

5.3.1 Introduction

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. Therefore it is also sometimes called the **DJP algorithm**, the **Jarník algorithm**, or the **Prim–Jarník algorithm**.

The only spanning tree of the empty graph (with an empty vertex set) is again the empty graph. The following description assumes that this special case is handled separately.

The algorithm continuously increases the size of a tree, one edge at a time, starting with a tree consisting of a single vertex, until it spans all vertices.

- **Input:** A non-empty connected weighted graph with vertices *V* and edges *E* (the weights can be negative).

- **Initialize:** V_{new} = {x}, where x is an arbitrary node (starting point) from V, E_{new} = { }
- Repeat until V_{new} = V:
- Choose an edge (u, v) with minimal weight such that u is in V_{new} and v is not (if there are multiple edges with the same weight, any of them may be picked)
- Add v to V_{new}, and (u, v) to E_{ne}
- **Output:** V_{new} and E_{new} describe a minimal spanning tree

Minimum Spanning Tree - Prim's algorithm

Given: Weighted graph.
Find a spanning tree with the minimal sum of the weights.
Similar to shortest paths in a weighted graph.
Difference: we record the weight of the current edge, not the length of the path .

5.3.2 Definition:

- Maintains a set of vertices S already in the spanning tree.
- Initially, S consists of one vertex r, selected arbitrarily.
- For every vertex u in V – S, maintain the weight of the lightest edge between u and any vertex in S.
- If there is no edge between u and S, then this weight associated with u is infinity.
- Add the vertex with the least weight to S.
- This is in effect adding the light weight edge crossing the cut S and V-S.
- Whenever a vertex is added to S, the weights of all its neighbors are reduced, if necessary.
- Let V ={1,2,..,n} and U be the set of vertices that makes the MST and T be the MST
- Initially : U = {1} and T = ϕ
- while (U $\neq$ V)
  let (u,v) be the lowest cost edge such that
  u$\in$ U and v $\in$ V-U
  T = T $\cup$ {(u,v)}
  U = U $\cup$ {v}

Pseudo-Code

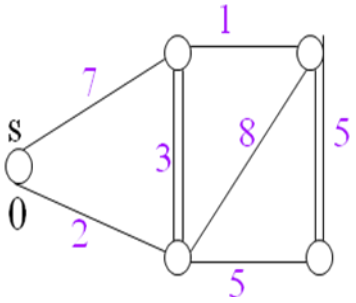
```
S = ϕ
For each u in V, key[u] = α
Key[r] = 0
Pred[r] = NULL
While V != S
u = Extract_Min(V-S)
        For each (v in Adj(u))
if (v not in S) key(v) = min(w(u, v), key(v))
```

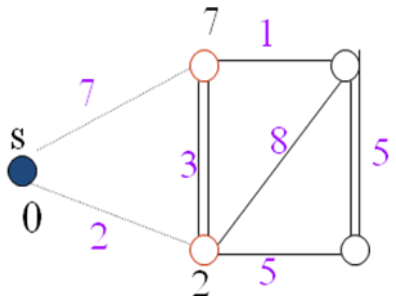
and pred(v) = u

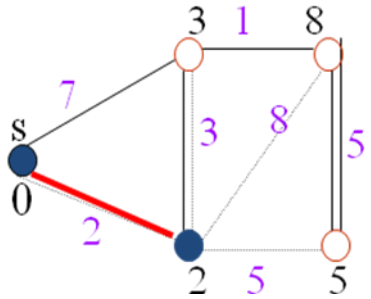
Add u in S

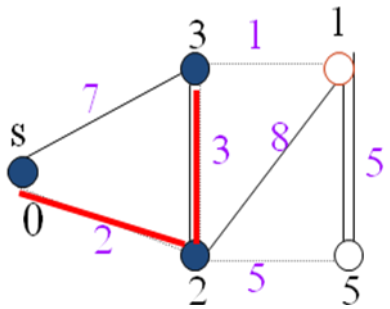
For each v in Adj[u]…. can be done in E complexity, Rest of the loop can be done in V^2 complexity. So, overall $O(V^2)$ Using heaps we can solve in O((V + E)logV)

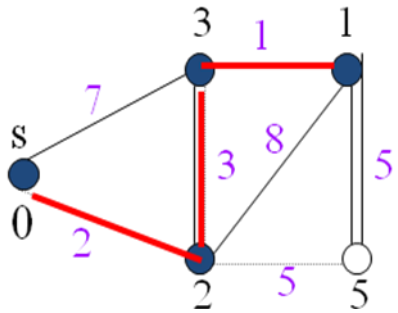
Example

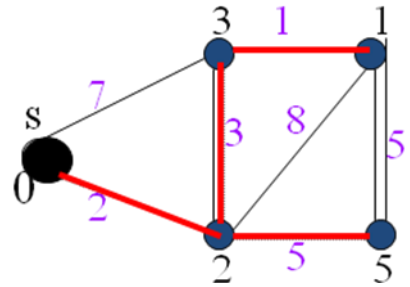








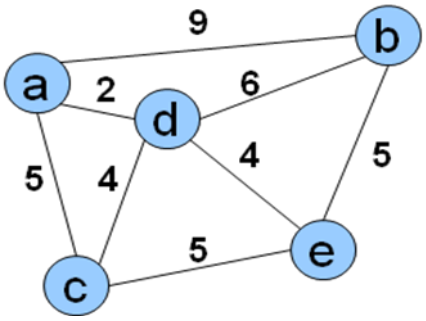




5.3.3 Prim's Algorithm implementation

Initialization

- Pick a vertex *r* to be the root
- Set *D*(*r*) = *0*, *parent*(*r*) = *null*
- For all vertices $v \in V$, $v \neq r$, set $D(v) = \infty$
- Insert all vertices into priority queue *P*,
- using distances as the keys



e	a	b	c	d
0	∞	∞	∞	∞

Vertex	Parent
e	-

While *P* is not empty:

1. Select the next vertex *u* to add to the tree

u = *P.deleteMin*()

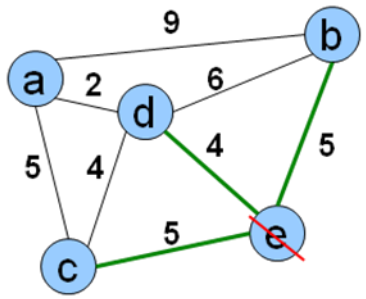
2. Update the weight of each vertex *w* adjacent to *u* which is not in the tree (i.e., $w \in P$)

If *weight*(*u*,*w*) < *D*(*w*),

a. *parent*(*w*) = *u*

b. *D*(*w*) = *weight*(*u*,*w*)

c. Update the priority queue to reflect new distance for *w*



e	d	b	c	a
0	∞	∞	∞	∞

Vertex	Parent
e	-
b	-
c	-
d	-

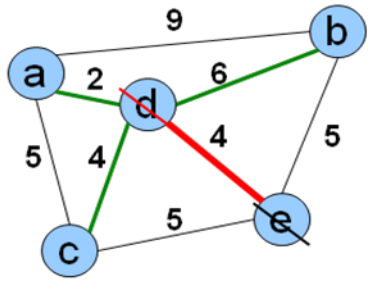
d	b	c	a
4	**5**	**5**	∞

Vertex	Parent
e	-
b	e
c	e
d	e

The MST initially consists of the vertex *e*, and we update the distances and parent for its adjacent vertices

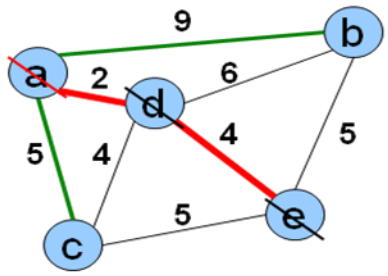
d	b	c	a
4	**5**	**5**	∞

Vertex	Parent
e	-
b	e
c	e
d	e



a	c	b
2	**4**	**5**

Vertex	Parent
e	-
b	e
c	d
d	e
a	d



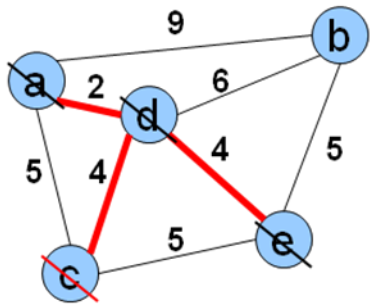
a	c	b
2	4	5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

---

c	b
4	5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d



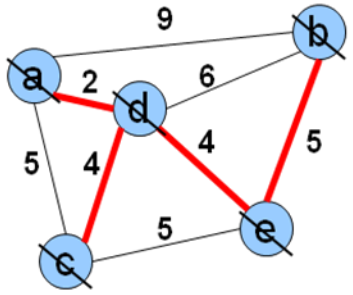
c	b
4	5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

---

b
5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d



b
5

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

The final minimum spanning tree

Vertex	Parent
e	-
b	e
c	d
d	e
a	d

5.3.4 Prim's Algorithm Invariant

- At each step, we add the edge (u,v) s.t. the weight of (u,v) is minimum among all edges where u is in the tree and v is not in the tree
- Each step maintains a minimum spanning tree of the vertices that have been included thus far
- When all vertices have been included, we have a MST for the graph!

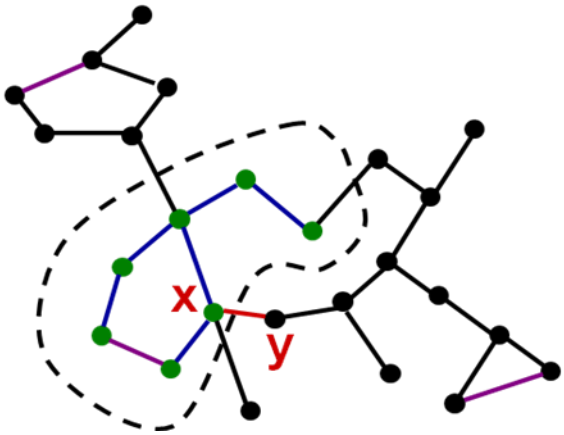
Correctness of Prim's

- This algorithm adds *n-1* edges without creating a cycle, so clearly it creates a spanning tree of any connected graph (*you should be able to prove this*).

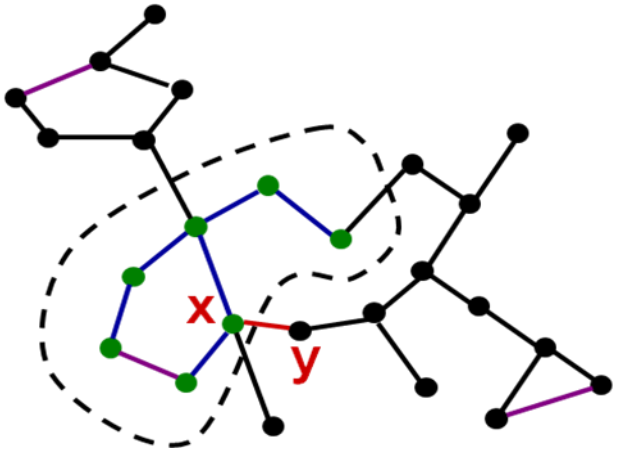
But is this a *minimum* spanning tree?

Suppose it wasn't.

- There must be point at which it fails, and in particular there must a single edge whose insertion first prevented the spanning tree from being a minimum spanning tree.
- Let *G* be a connected, undirected graph
- Let S be the set of edges chosen by Prim's algorithm *before* choosing an errorful edge (x,y)



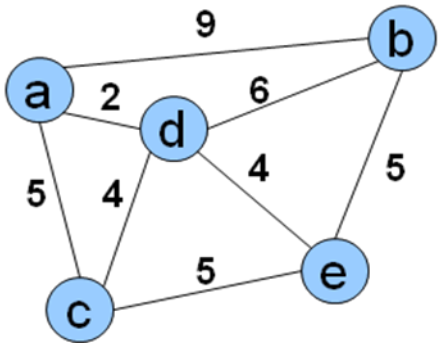
- Let **V'** be the vertices incident with edges in **S**
- Let **T** be a MST of ***G*** containing all edges in **S**, but not **(*x*,*y*)**.
- Edge **(*x*,*y*)** is not in **T**, so there must be a path in **T** from ***x*** to ***y*** since **T** is connected.
- Inserting edge **(*x*,*y*)** into **T** will create a cycle



- There is exactly one edge on this cycle with exactly one vertex in **V'**, call this edge **(*v*,*w*)**
- Since Prim's chose **(*x*,*y*)** over **(*v*,*w*)**, w(**v**,**w**) >= w(**x**,**y**).
- We could form a new spanning tree **T'** by swapping **(*x*,*y*)** for **(*v*,*w*)** in **T** (*prove this is a spanning tree*).
- w(**T'**) is clearly no greater than w(**T**)
- But that means **T'** is a MST
- And yet it contains all the edges in **S**, and also **(*x*,*y*)**

Another Approach

- Create a forest of trees from the vertices
- Repeatedly merge trees by adding "**safe edges**" until only one tree remains
- A "safe edge" is an edge of minimum weight which does not create a cycle



forest: {a}, {b}, {c}, {d}, {e}

5.3.5 Running time of Prim's algorithm (without heaps)

Initialization of priority queue (array): O(|***V***|)

Update loop: |***V***| calls

- Choosing vertex with minimum cost edge: O(|***V***|)
- Updating distance values of unconnected vertices: each edge is considered only **once** during entire execution, for a **total** of O(|***E***|) updates

Overall cost without heaps: $O(|E| + |V|^2)$

When heaps are used, apply same analysis as for Dijkstra's algorithm.

5.3.4.1 Time complexity

Minimum edge weight data structure	Time complexity (total)
adjacency matrix, searching	$O(V^2)$
binary heap and adjacency list	O((V + E) log(V)) = O(E log(V))
Fibonacci heap and adjacency list	O(E + V log(V))

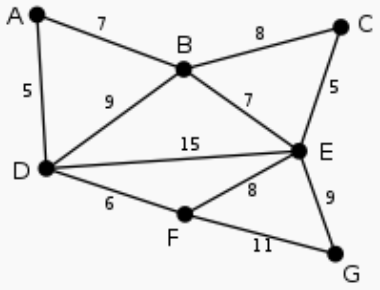
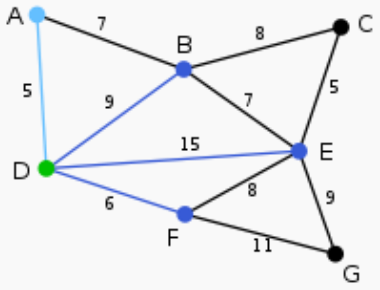
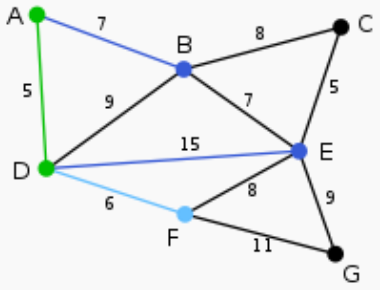
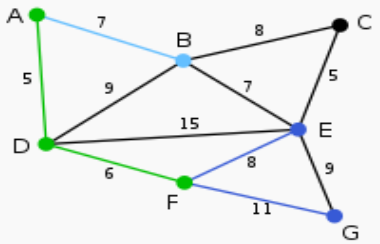
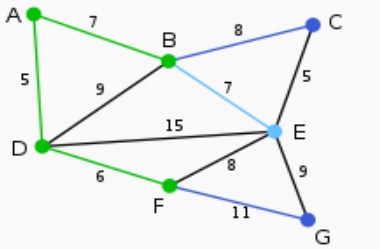
A simple implementation using an adjacency matrix graph representation and searching an array of weights to find the minimum weight edge to add requires $O(V^2)$ running time.

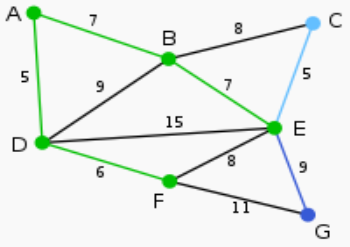
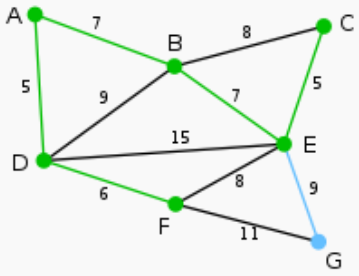
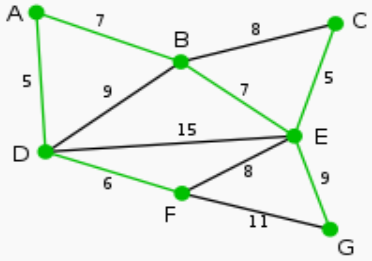
Using a simple binary heap data structure and an adjacency list representation, Prim's algorithm can be shown to run in time $O(E \log V)$ where E is the number of edges and V is the number of vertices.

Using a more sophisticated Fibonacci heap, this can be brought down to $O(E + V \log V)$, which is asymptotically faster when the graph is dense enough that E is $\Omega(V)$.

Example run

Image	Description

	This is our original weighted graph. The numbers near the edges indicate their weight.
	Vertex **D** has been arbitrarily chosen as a starting point. Vertices **A**, **B**, **E** and **F** are connected to **D**through a single edge. **A** is the vertex nearest to **D** and will be chosen as the second vertex along with the edge **AD**.
	The next vertex chosen is the vertex nearest to *either* **D** or **A**. **B** is 9 away from **D** and 7 away from **A**,**E** is 15, and **F** is 6. **F** is the smallest distance away, so we highlight the vertex **F** and the arc **DF**.
	The algorithm carries on as above. Vertex **B**, which is 7 away from **A**, is highlighted.
	In this case, we can choose between **C**, **E**, and **G**. **C** is 8 away from **B**, **E** is 7 away from **B**, and **G** is 11 away from **F**. **E** is nearest, so we highlight the vertex **E** and the arc **BE**.

	Here, the only vertices available are **C** and **G**. **C** is 5 away from **E**, and **G** is 9 away from **E**. **C** is chosen, so it is highlighted along with the arc **EC**.
	Vertex **G** is the only remaining vertex. It is 11 away from **F**, and 9 away from **E**. **E** is nearer, so we highlight it and the arc **EG**.
	Now all the vertices have been selected and the minimum spanning tree is shown in green. In this case, it has weight 39.

U	**Edge(u,v)**	**V \ U**
{}		{A,B,C,D,E,F,G}
{D}	(D,A) = 5 **V,** (D,B) = 9, (D,E) = 15, (D,F) = 6	{A,B,C,E,F,G}
{A,D}	(D,B) = 9, (D,E) = 15, (D,F) = 6 **V**, (A,B) = 7	{B,C,E,F,G}
{A,D,F}	(D,B) = 9, (D,E) = 15, (A,B) = 7 **V**, (F,E) = 8, (F,G) = 11	{B,C,E,G}
{A,B,D,F}	(B,C) = 8, (B,E) = 7 **V**, (D,B) = 9 cycle, (D,E) = 15, (F,E) = 8, (F,G) = 11	{C,E,G}
{A,B,D,E,F}	(B,C) = 8, (D,B) = 9 cycle, (D,E) = 15 cycle, (E,C) = 5 **V**, (E,G) = 9,(F,E) = 8 cycle, (F,G) = 11	{C,G}

{A,B,C,D,E,F}	(B,C) = 8 cycle, (D,B) = 9 cycle, (D,E) = 15 cycle, (E,G) = 9 **V**, (F,E) = 8 cycle, (F,G) = 11	{G}
{A,B,C,D,E,F,G}	(B,C) = 8 cycle, (D,B) = 9 cycle, (D,E) = 15 cycle, (F,E) = 8 cycle, (F,G) = 11 cycle	{}

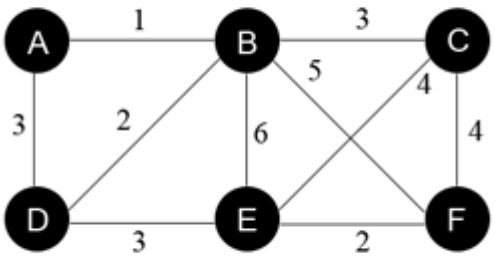
Proof of correctness

Let *P* be a connected, weighted graph. At every iteration of Prim's algorithm, an edge must be found that connects a vertex in a sub-graph to a vertex outside the sub-graph. Since *P* is connected, there will always be a path to every vertex. The output *Y* of Prim's algorithm is a tree, because the edge and vertex added to *Y* are connected. Let Y_1 be a minimum spanning tree of P. If Y_1=*Y* then *Y* is a minimum spanning tree. Otherwise, let *e* be the first edge added during the construction of *Y* that is not in Y_1, and *V* be the set of vertices connected by the edges added before *e*. Then one endpoint of *e* is in *V* and the other is not. Since Y_1 is a spanning tree of *P*, there is a path in Y_1 joining the two end points. As one travels along the path, one must encounter an edge *f* joining a vertex in *V* to one that is not in *V*. Now, at the iteration when *e* was added to *Y*,*f* could also have been added and it would be added instead of *e* if its weight was less than *e*. Since *f* was not added, we conclude that

$$w(f) \geq w(e).$$

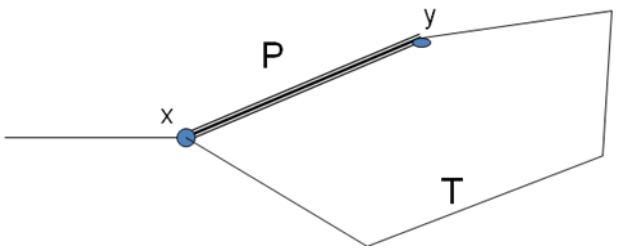
Let Y_2 be the graph obtained by removing *f* from and adding *e* to Y_1. It is easy to show that Y_2 is connected, has the same number of edges as Y_1, and the total weights of its edges is not larger than that of Y_1, therefore it is also a minimum spanning tree of *P* and it contains *e* and all the edges added before it during the construction of *V*. Repeat the steps above and we will eventually obtain a minimum spanning tree of *P* that is identical to *Y*. This shows *Y* is a minimum spanning tree.

SET: { }



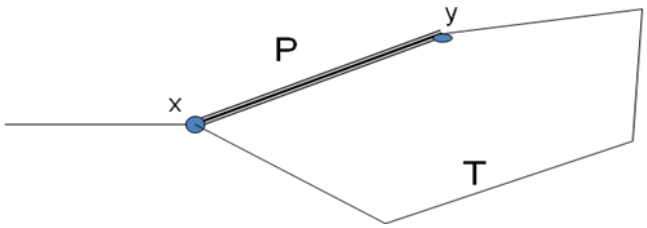
5.3.6 Prim's Algorithm: A Proof

- Suppose we have a tree T that is the minimal spanning tree for a graph. Let P be the spanning tree from Prim's alg. and $P \neq T$ Then there is an arc, a, in P that is not in T. If arc a connects node x to node y, then there is a path within the tree of T that also connects x to y:



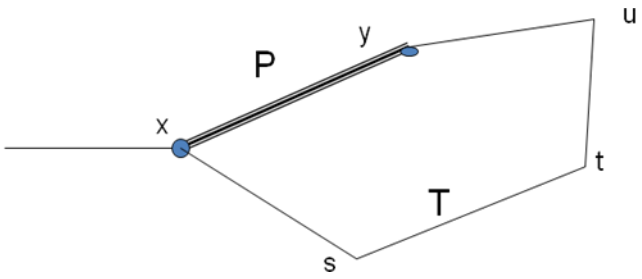
Case 1: The weight of P is smaller than one of the edges on the path from the tree T

- Then we remove the edge with highest weight from T and replace it with the edge from P. This gives a smaller spanning tree than T – and contradicts the fact that T is the minimal spanning tree.



Case 2: The weight of P is greater than the weight of all the edges in the path from T.

- **Then consider the step – in Prim's algorithm- when we add edge P.**
- **P is the shortest edge remaining which can be added to the graph.**
- **But we would add xs before P (or st or tu or uy – whichever is still available to add.**
- **So we never add P – a contradiction!**



References:

Introduction to Algorithms, Third Edition. MIT Press, 2009. Section 23.2: The algorithms of Kruskal and Prim, pp. 631–638.

Unit 4: Undirected Graphs – Bi-connectivity

CONTENTS		
Sl. No.	**Topics**	**Page No.**
5.4.0	**Aims and Objectives**	
5.4.1	**Introduction**	

5.4.0 AIMS AND OBJECTIVES

At the end of this lesson, students should be able to demonstrate appropriate skills, and show an understanding of the following:

- Aims and objectives of Biconnectivity
- Introduction
- Definition
- Undirected Graph
- Biconnectivity

5.4.1 Introduction

Graph: Let a graph $G = (V, E)$ represent a communication network, where V denotes the set of sites and E denotes the links between them. An edge $e = (a, b)$ denotes the feasibility of adding a link between the sites a and b, and $w(e)$ represents the costs of constructing the link e.

When designing communication networks, a minimum spanning tree is usually the cheapest network that will allow a given set of sites to communicate. However such a network is not robust against failures, since it might not survive the break of even a single link or site. Besides the minimization of connection costs, reliability is a very important issue. The network should be robust against failures in connections or switching nodes in the sense that any two nodes do not loose connection in case of up to a certain maximum number of simultaneous failures. To accomplish this, redundant communication routes must exist for any pair of nodes.

In graph theory, the terms vertex-connectivity and edge-connectivity are used to describe this kind of robustness. A connected, undirected graph $G(V, E)$ has edge-

connectivity $C_E(G)$ ($C_E(G) >= 1$) if at least $C_E(G)$ edges need to be deleted in order to separate G into disconnected components. Similarly, the graph has vertex-connectivity $C_V(G)$ ($C_V(G) >= 1$) if at least $C_V(G)$ vertices with their adjacent edges must be deleted for disconnecting G.

Note that C_V is always less than or equal to $C_E(G)$, since at least one incident vertex for any of $C_E(G)$ edges disconnecting G need to be deleted. Furthermore, $C_E(G)$ is always less than or equal to the minimum-degree of all vertices V.

Undirected Graph

- Undirected Graph is the same as a directed graph except that edges are bi-directional
  - A bi-directional edge does not mean two directed edges, but instead one edge that can be taken in either direction

Two vertices in an undirected graph are called adjacent if there is an edge from the first to second. Hence, in the undirected graph shown below, vertices 1 and 2 are adjacent, as are 3 and 4 , but 1 and 4 are not adjacent.

A path is a sequence of distinct vertices, each adjacent to the next. Figure below shows the path.

A cycle is a path containing at least three vertices such that the last vertex on the path is adjacent to the first.

A graph is called connected if there is a path from any vertex to any other vertex. Directed Graph When all edges in a path or cycle have the same direction, so that following a path or a cycle means always moving in the direction indicated by the arrows. Such a path (cycle) is called a directed path. A directed graph is strongly connected if there is a directed path from any vertex to any other vertex. When the directions of the edges are suppressed and the resulting undirected graph is connected, then it is said to be weakly connected.

Connectivity for undirected Graphs

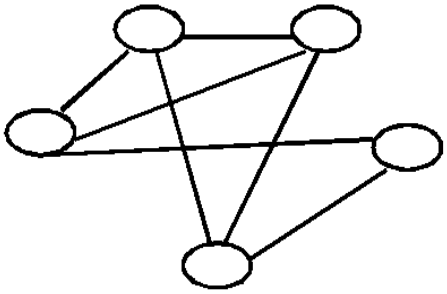
Definition: An undirected graph is said to be **connected if** for any pair of nodes of the graph, the two nodes are reachable from one another (i.e. there is a path between them).

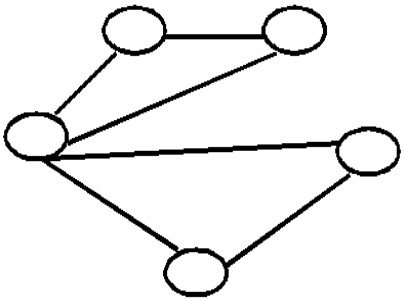
If starting from any vertex we can visit all other vertices, and then the graph is connected

Biconnectivity

A graph is **biconnected**, if there are no vertices whose removal will

disconnect the graph.





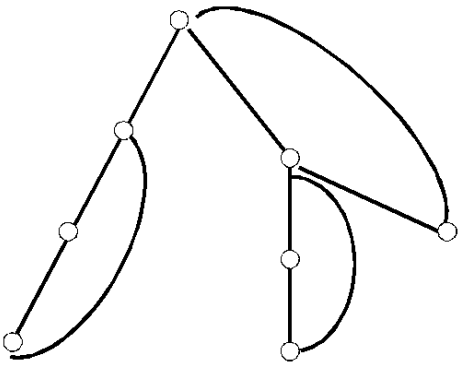
Biconnected **Not Biconnected**

Biconnected Components

- A subgraph S of G is a *biconnected* component if S is biconnected and adding any edge of G to S will destroy this property.
- The set of biconnected components is a partition of edges of G. Vertices may appear more than once (i.e., articulation points).

Example: Biconnected Components of Graph G

Maximal subgraphs of G which are biconnected.



Biconnected graph

In the mathematical discipline of graph theory, a **biconnected graph** is a connected graph with no articulation vertices.

In other words, a **biconnected** graph is connected and **non separable**, meaning that if any vertex were to be removed, the graph will remain connected.

The property of being 2-connected is equivalent to bi-connectivity, with the caveat that the complete graph of two vertices is sometimes regarded as biconnected but not 2-connected.

This property is especially useful in maintaining a graph with a two-fold redundancy, to prevent disconnection upon the removal of a single edge (or connection).

The use of **biconnected** graphs is very important in the field of networking, because of this property of redundancy.

Definition

A **biconnected** undirected graph is a connected graph that is not broken into disconnected pieces by deleting any single vertex (and its incident edges).

A **biconnected** directed graph is one such that for any two vertices *v* and *w* there are two directed paths from *v* to *w* which have no vertices in common other than *v* and *w*.

A biconnected graph is one in there is a path between every pair of vertices after any one vertex and its edges is removed

A biconnected component of an undirected graph is a subgraph not contained in any larger biconnected subgraph

An articulation point is a vertex that, if removed, disconnects subgraphs

- We can discover if a vertex is an articulation point by removing it and then seeing if there is a path (using DFS or BFS) starting at every other vertex and seeing if the path extends to every other non-removed vertex

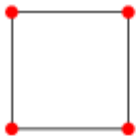
Example:

2-path graph

triangle graph

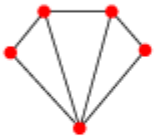
diamond graph

square graph

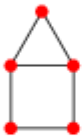


tetrahedral graph

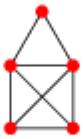
gem graph



house graph



house X graph



pentatope graph

5-graph 31

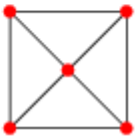
(2,3)-complete bipartite graph

5-cycle graph

(3,2)-fan graph

Johnson solid skeleton 12

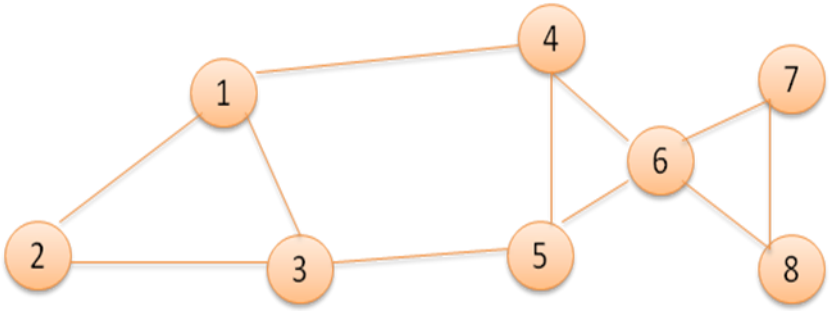
5-wheel graph



Equivalent definitions of a bi-connected graph *G*

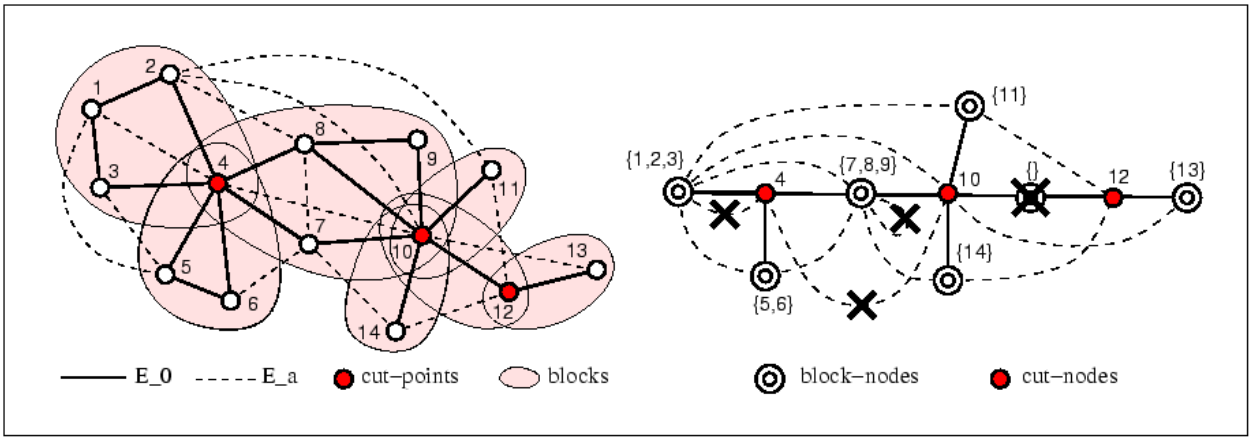
- Graph G has no separation edges and no separation vertices
- For any two vertices u and v of G, there are two disjoint simple paths between u and v (i.e., two simple paths between u and v that share no other vertices or edges)
- For any two vertices u and v of G, there is a simple cycle containing u and v

Example



Biconnectivity Augmentation Problems:

Figure: Transforming given network $G = (V, E_0 \cup E_a)$ into a block-cut tree, to prepare it for efficient vertex biconnectivity augmentation.



Biconnectivity Augmentation Problems

In our work, we concentrate on the NP-hard biconnectivity augmentation problem, which could be split in two subproblems:

- vertex-biconnectivity augmentation (V2AUG) problem.
- edge-biconnectivity augmentation (E2AUG) problem.

In both cases, given are a weighted, undirected graph $G(V, E_0 \cup E_a)$ with vertex (edge)-connectivity $C_V(G) \geq 2$ ($C_E(G) \geq 2$), and a spanning subgraph $G_0(V, E_0)$. The goal is to identify a set of augmenting edges $AUG \subset E_a$ with minimum total weight

$$w(AUG) = \sum_{e \in AUG} w(e) \qquad (1)$$

such that graph $G_{AUG}(V, E_0 \cup AUG)$ is edge- or vertex-biconnected, i.e. $C_E(G_{AUG}) \geq 2$ ($C_V(G_{AUG}) \geq 2$).

Algorithm BICONN-AUGMENT

Algorithm BICONN-AUGMENT

Input: (1) Multihop wireless network M = (N, L) (2) Least-power function λ (3) Initial power assignment inducing connected network

Output: Power levels p for each node that induces a biconnected graph.

begin

1. sort node pairs in non-decreasing order of distance
2. G = graph induced by (A, λ, p)
3. **for** each (u, v) in sorted order **do**
4. **if** biconn-comp(G,u) $\neq$ biconn-comp(G,v)
5. $q = \lambda$(distance(u,v))
6. $p(u) = \max(q, p(u))$
7. $p(v) = \max(q, p(v))$
8. add (u, v) to G
9. perNodeMinimalize(M, λ, p, 2)

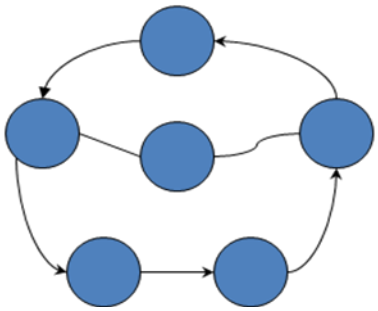
end

- Identify the bi-connected components in the graph induced by the power assignment from algorithm CONNET
- This is done using method based on depth-first search
- Node pairs are selected in non-decreasing order of their mutual distance and joined only if they are in different bi- connected components
- This is continued until the network is biconnectd.

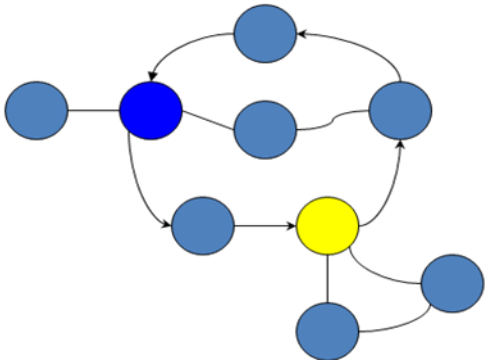
Intuition – Biconnected Components

- Now we only care about the external face of the current embedding, so we always search along the external face.
- Each biconnected component has a complete circle as the outer face, and different biconnected components correspond to different circles.
- These circles are connected by cut vertices, and it is hard to define the “next” vertex when we are at a cut vertex.
- Therefore, it is natural to consider biconnected components.

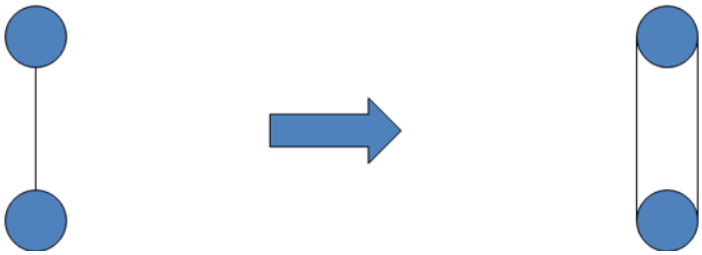
An illustration on biconnectivity



For a biconnected graph, it is easy to define the next vertex on the external face when we have a order – either clockwise or counterclockwise



Not biconnected graph: each biconnected component has a bounding cycle, and these cycles are joined by cut vertices.



If a biconnected component has only two vertices, it is helpful to view it as a cycle of two parallel edges. Thus every biconnected component has a bounding cycle.

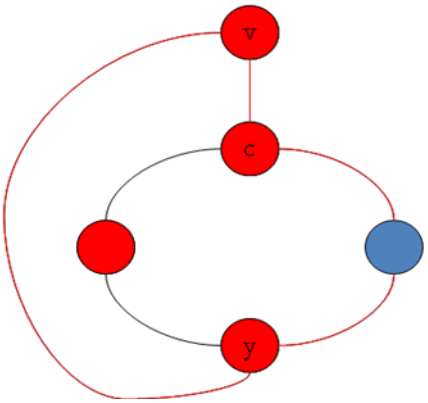
Illustration of the algorithm:

- Now we have enough preliminaries to give a more detailed picture of the algorithm.
- When we process a vertex *v*, we first embed all the tree edges. Then we perform two searches along the external face, one in clockwise order and one in counterclockwise order. If we find a vertex that has a back edge to *v*, we embed this edge and keep going; if we encounter a stopping vertex then we stop.
- What is a stopping vertex? This will be explained later.

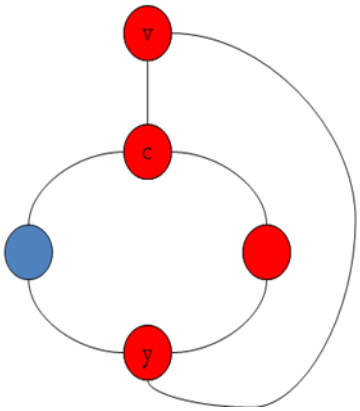
Embedding an edge

Two things happened after embedding the edge:

1. C is no longer a cut vertex.
2. We have a new external face.



■ **We traverse along the external face in and find *y*, so we embed the edge from *y* to *v*.**



■ **In the last picture, we may also traverse in clockwise order, then the edge embedded will be in the different direction.**

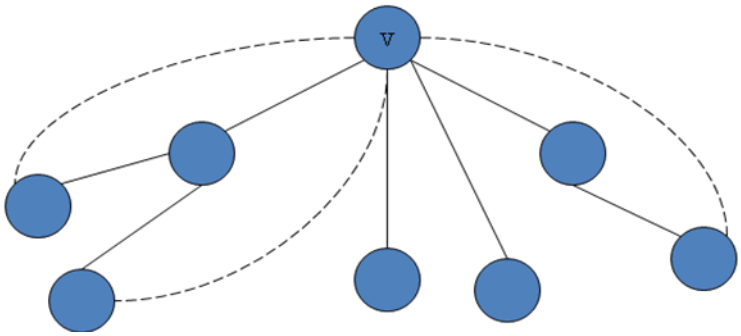
Observation

- Notice that in the last two pictures when we embed an edge, the path on the external face we traversed will NOT stay on the external face after embedding the edge, which means it will not be traversed in the future. This is essential for achieving linear runtime.
- This also indicates that the path we traversed can not contain any externally active vertex (with an exception that will be discussed later), so we stop the traverse if we encounter an externally active vertex.

Embedding an Edge

- We talked about "embedding an edge" several times. How do we actually embed an edge in the code?
- It is very easy. For each vertex we equip a list recording its neighbors in clockwise order. At first the list contains only the tree edge. If we traverse in clockwise order and find an edge, the vertex is added to the front of list; if we traverse in counterclockwise order then add it to the back of the list.
- The above process will be run on each child biconnected component separately, and lists from different child biconnected component can be concatenated. (Illustration on next slide)

Embedding Edges – Concatenation

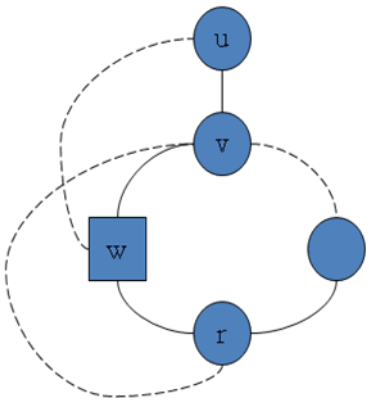


- For each child biconnected component of *v* we can get a list of edges adjacent to *v*. There are no edges between different child biconnected components, so the lists can be combined in any order. However, we would like a particular order (which is described later) to help future embedding.

Externally Activity

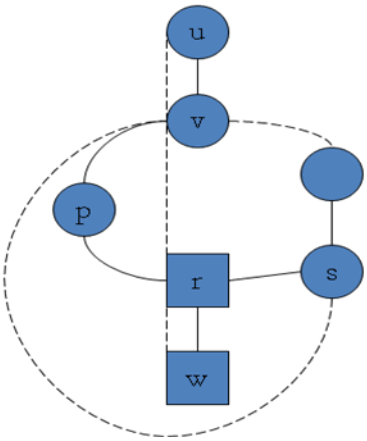
- When processing a vertex *v*, a processed vertex *w* is called externally active if either it has a back edge pointing to an ancestor of *v*, or there is an externally active vertex in one of its child biconnected components.
- Notice that the second condition only applies when the vertex has at least one child biconnected components, i.e., it must be a cut vertex.
- We will use several pictures to illustrate these two types of externally active vertices.

Type 1



- We are processing *v*. *u* is an ancestor of *v*. *w* has a back edge pointing to *u* so *w* is externally active. Suppose we pass *w* and find *r* and embed the edge from *r* to *v*, then there is no way for us to embed the back edge of *w*.
- Later on we will always use square nodes to indicate externally active nodes, and dotted lines for back edges.

Type 2

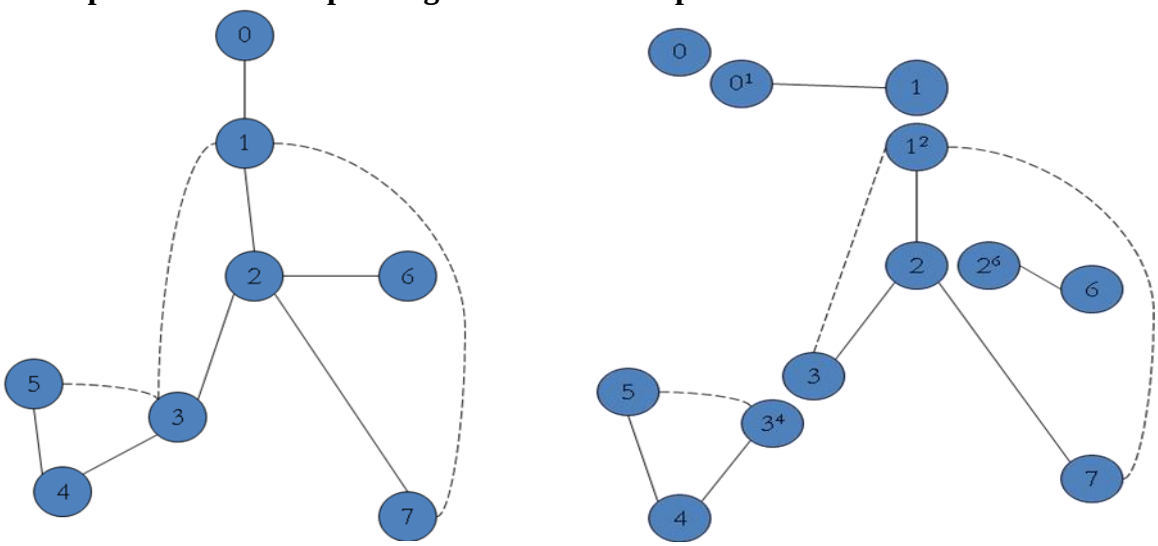


- *r* has a path to an externally active vertex *w*, so *r* is also externally active. If we pass *r* and embed the edge from *s* to *v*, then we can't embed the back edge of *w* in the future.
- Notice *p* also has a path to *w*, but *p* is not a cut vertex, so *p* is not externally active.

How to determine external activity?

- For type-1, we can just check if it has a back edge to the ancestor of *v*.
- For type-2, we shall use the concept *lowpoint value*. These will be discussed later.

A Graph and the corresponding biconnected components



Dictionary of Data Structures

A

absolute performance guarantee
abstract data type
(a,b)-tree

M

Malhotra-Kumar-Maheshwari blocking flow
Manhattan distance
many-one reduction

linear product
linear program
linear quadtree
linear search
link
linked list
list
list contraction
little-o notation
L_m distance
load factor
local alignment
locality-sensitive hashing
local optimum
logarithmic
longest common subsequence
longest common substring
Lotka's law
lower bound
lower triangular matrix
lowest common ancestor
l-reduction
lucky sort
LZW compression: see Lempel-Ziv-Welch
worst-case minimum access
X
xor
Y
Yule distribution: see Zipfian distribution
Z
Zeller's congruence
0-ary function
0-based indexing
0-1 knapsack problem: see knapsack problem
Zhu-Takaoka
Zipfian distribution
Zipf's law
zipper
ZPP

Bibliography

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

Gaston H. Gonnet and **Ricardo Baeza-Yates**, *Handbook of Algorithms and Data Structures -- in Pascal and C*, 2nd edition, Addison-Wesley, 1991.

Ellis Horowitz and **Sartaj Sahni**, *Fundamentals of Data Structures*, Computer Science Press, 1983.

Robert Sedgewick, *Algorithms in C*, Addison-Wesley, 1997.

Thomas Standish, *Data Structures in Java*, Addison-Wesley, 1998.

Website:

Eric W. Weisstein. "Biconnected Graph." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/BiconnectedGraph.html