Data Structures

A Complete Course

By

Syed Waqar ul Qounain Jaffry

Punjab University College of Information Technology

— to my parents, teachers and students

who have made it possible for me through their respective

level of contribution, and Dr. M. A. Pasha

Contents at Glance

Table of Contents

Abstract

This document contains the lecture notes delivered to the students in the course CS-208 Data Structures taught at PUCIT Lahore during the Fall semester, 2003. The class covered the following set of topics: ADT, Stack, Queue, List, Tree, Graph, Searching, Sorting and Recursion. Between 2 and 3 lectures were dedicated to each topic. The class was an under-graduate class and assumed that the students came in with a working knowledge of the material covered in any standard Object Oriented Programming textbook, such as Dietel & Dietel, “C++ How to Program". A goal of the class was to have the students gain an appreciation of how interesting data-structures are used in many computer science applications.

The notes contained in this document are based on what was covered in the lectures, tutorials, and lab sessions etc.

I would like to thank all my T.As the students who took the notes that appear in this document. They did an excellent job and in many cases went far beyond the call of duty in executing the contents covered.

Preface

As the computer systems are becoming complex and the input sizes increase, the need for the organization of data, its storage and efficient retrieval becomes more vital. Data structures serve the purpose. So it is a core subject of any computer science curriculum.

Material present in this collection is organized into ten different parts based on various subject activities executed during semester.

PART–1 Course Outline contains the basic information regarding course plan and timings.
PART–2 Lecture Notes contains the contents delivered in classroom during the 32 lectures of 1 and half hour period.
PART–3 Laboratory Manual contains the practical labs work that was executed in 14 labs of 4-hour duration.
PART–4 Case Studies this part comprises of 4 programs with detailed description tha .
PART–5 Tutorials contains the contents of 6 tutorials delivered to eager students in separate lectures of 2-hours duration each.
PART–6 Programming Problems contains the 55 different programming exercises that evaluate ones problem solving and programming skills and the comprehension of data structure concepts.
PART–7 Assignments and Quizzes contains the different sessional assessment activities including 2 assignments and 4 quizzes.
PART–8 Term Paper and Project contains necessary information regarding subject project and term paper.
PART–9 Examination contains the midterm and final term papers and practical conducted.
PART–10 Subject Resources FALL 2001 contains the quizzes assignments papers etc. of last session of data structures subject.
Glossary at the end of this document a small dictionary of the terms used in this course is included.

I am hopeful that the right mix of above contents can give the students an opportunity to learn data structure effectively at under graduated level.

Syed Wqar ul Qunain Jaffry
Tuesday, September 16, 2003
PUCIT

Part 1

Course Outline

1.1. Course Description

This subject deals with the organization of information, usually in memory, for better algorithm efficiency. It will provide tools and techniques necessary to design and implement large-scale computer applications using Object Oriented Techniques in C++ programming language.

1.2. Course Objectives

- Apply appropriate data structures and abstract data types (ADT) such as bags, lists, stacks, queues, trees, hash tables, and graphs in problem solving.
- Apply object-oriented principles of polymorphism, inheritance, and generic programming when implementing ADTs for data structures.
- Create alternative representations of ADTs either from implementation or the standard libraries.
- Apply recursion as a problem solving technique.
- Determine appropriate ADTs and data structures for various sorting and searching algorithms.
- Determine time and space requirements of common sorting and searching algorithms.

1.3. Course Text and recommended books

Text Books:

Ellis Horowitz, Sartaj Sahni, and Dinesh Mehta, *Fundamentals of Data Structures in C++*, Computer Science Press, New York, 1995.

Aron M. Tenneunbaum, *Data Structures Using C/C++*, 2nd Edition *Prentice Hall International*

Recommended Books:

Glenn W. Rowe, *Introduction to Data Structures and Algorithms with C++*, Prentice Hall India,

Thomas A. Standish, *Data Structures, Algorithms, and Software Principle*, Addison Wesley Publishing Comp.

Mark Allen Weiss, *Algorithms, Data Structures, and Problem Solving in C++*, Addison Wesley Publishing Comp.

Thomas H. Cormen, *Introduction to Algorithms*, 2nd Edition MIT Press,

1.4. Course Requirements

Assigned Readings

Students are expected to study the text thoroughly and carefully. You will find it helpful to read assigned material from the text both before and after it is covered in class. Students may find it useful to also have a C++ reference manual available. Use the class syllabus and current topic in the class to guide your readings.

Quizzes & In-class Assignments

Occasional quizzes will be given. These will cover material covered in class and in the assigned reading in the textbook. In-class assignments will also be given regularly throughout the class. They will generally be shorter tasks or programs designed to illustrate some concept or C++ implementation technique being described. In-class assignments will not be collected or graded. However, it is vitally important that you do these.

Homework Assignments

There will be a homework assignment approximately one every two weeks. The assignments will cover the important course concepts. In general, assignments will consist of written component or/and programming component. Each assignment is due **at the beginning of class** on the assignment's due date.

Written Assignments/Self Checks

These assignments will contain a few written questions to test your mastery of the material. These will be graded according to the correctness of your solutions. You are also expected to be able to answer all the questions at the end of each chapter, whether or not they have been assigned as written homework.

Programming Assignments

These assignments will contain programming. The following are some of the general grading criteria that will used to evaluate the quality of most of your programs.

- Program correctness
- Algorithm and data structure design
- Program robustness
- Program style
- Program documentation
- Appearance of the program's output
- Following specifications
- Testing

In addition, all work should be clear, legible, well organized, and clearly identified.

Expect to invest a considerable amount of time to correctly complete each program. Programming requires repetition to master. You will write many programs during this semester. If you are diligent in writing complete, accurate, well-documented programs, you will improve your programming ability considerably and will gain a skill that will serve you well in future courses and in your career.

One of the abilities that I want you to acquire this semester is the ability to debug your programs. You may come to me for help with your programs during my office hours. I will expect you to have already begun the process of debugging. This includes printing out the value of variables around the problem area. I will give you suggestions about how to find and fix problems. However, I will generally not completely debug your program for you.

You are expected to spend a minimum of 3 to 4 hours outside of class for every hour in class. This time is often not uniformly distributed over the semester, but tends to cluster around program due dates. It is important for you to consistently complete and understand the assignments (programs and exercises) since the material is cumulative. Please see me, if you begin to have difficulties. Homework should be a learning experience. I am happy to help you with difficulties that you are experiencing with homework assignments or difficult concepts.

Collaboration

In this course it is acceptable for you to discuss approaches and difficulties on homework assignments with other students; this can be a learning experience for both of you. It is not acceptable to copy another's solutions, code, or logic; this is not a learning experience, but is academic dishonesty. The penalties for academic dishonesty are severe.

Examinations

There will be a midterm and a final examination. We will discuss the topics to be covered before each examination. Any exam missed will count as a zero unless arrangements have been made prior to the day of the exam through Academic Coordinator.

Term Paper

You have to write a term paper of your choosing. It will reflect your writing skills and your competence in the subject. You can find typographical Format and standards for term paper in term paper guide notes. Following are the tentative schedule for Term Paper

Submission of Term Paper Abstract	Wednesday, Jun 18, 2003
Submission of Term Paper	Wednesday, Jun 25, 2003

Term Practical

Term practical is 2 to 3 hour programming session at the end of your semester to evaluate you problem comprehension, analysis and programming skills. You have to program 2 to 3 problems in provided time. General grading criteria that will be use to evaluate the quality of your programs is mentioned above (programming assignments section).

Term Project

You have to do a project of your choosing so as to achieve a better appreciation for data structures and their role in the software development life cycle. The project will be worth up to 10% of your grade in credit. You may work alone or in teams of two, the size and scope of the project of teams should be larger than that of individuals and will be graded accordingly. You may use data structures discussed in class, modified versions of these, or you may create new ones. For project submission you will need to meet the following two submission deadlines:

Project Proposal	(2%)	Monday, Jun 30, 2003
Final Product (Software/Documentation)	(8%)	Monday, September 03, 2003

Proposal:

The proposal should be a short (no more than a couple paragraphs or so) abstract briefly describing the problem your project will solve and roughly how you intend to solve it. The purpose of the proposal milestone is dual: to let me know who will be doing a project and to ensure that you start thinking about it early enough in the semester. I will give you feedback on the reasonableness of the scope. What is reasonable will depend on the nature of the project; for example, a project that involves designing new data structures will probably require a smaller problem domain than one that combines standard data structures. You will receive 2% (at max) for submitting the proposal, but this is conditional upon submitting the final product.

Final Product:

As with your homework assignments, you must submit your executables, test cases, and well-documented code. You will also be expected to submit a short report describing the product, discussing your design and implementation choices and their tradeoffs, experiments, performance analyses and measurements, and conclusions. Your code will be graded according to the same criteria used for grading assignments. Your report will be graded according to its completeness and clarity and should demonstrate to me that you have thought deeply about the issues inherent in your problem domain and your solution. Finally, your project will be judged based on its quality and novelty.

1.5. Grading

The reason for giving a grade is to assess accurately and fairly your knowledge and performance in the class. I attempt to create exams that accurately assess your understanding. However, some of the material in this class is difficult and the exams will reflect the degree of difficulty.

You should critically evaluate the grading of all programs and tests. If you feel any items were miss-graded or the question was not clearly stated, state your argument on a separate piece of paper and I will consider your reasoning and give you a prompt reply.

Grading Categories

The final grade will be a weighted average of the following categories:

In-class assignments	00%
Quizzes, assignments and labs	20%
Midterm examination	25%
Final term examination	25%
Term Paper	05%
Term Practical	15%
Term project	10%

Grading Scale

Grading of the subject will in accordance with the grading rules of College.

1.6. Tentative Schedule

Time Line

Start of Semester:	Monday, April 14, 2003
Midterm Examination:	Monday, Jun 09, 2003
Submission of Term Paper Abstract	Wednesday, Jun 18, 2003
Submission of Term Paper	Wednesday, Jun 25, 2003
Submission of DS Project Proposal	Monday, Jun 30, 2003
DS Final Term Examination:	Monday, August 29, 2003
DS Term Practical:	Monday, September 01, 2003
Submission of Project (Software/Doc.)	Monday, September 03, 2003
DS Term Viva (Project):	Monday, September 04, 2003
Announcement of Result:	Monday, September 08, 2003

Contents Before Midterm

Week	Topic	Reading
1	Course introduction, Object-oriented design Abstract data types (ADTs) and data structures	1.1-1.4
	Algorithm specification, Recursion	1.5
2	Performance analysis and Performance measurement, Asymptotic notation	1.6-1.7
	ADTs and C++ classes Array ADT, Polynomial ADT	2.1-2.3
3	Representation of arrays and Matrices Assignment 1 distributed	2.4-2.6
	Sparse matrices, String ADT	2.7-2.8
4	Templates and Stack ADT	3.1-3.3
	Applications of stacks	3.5-3.9
5	Applications of stacks continued Assignment 2 distributed	3.5-3.9
	Queue ADT	3.4
6	Applications of Queue	OS Reference
	Singly linked lists, Representing lists in C++	4.1-4.2
7	Linked lists with templates, Circular lists, Linked stacks and queues Assignment 3 distributed	4.3-4.5
	Linked Sparse Matrices Doubly linked lists, Iterators	4.6,4.9
8	Polynomials, Generalized lists Virtual functions	4.10-4.11-4.12
	Heterogeneous lists	4.13
	Midterm	

Contents After Midterm

Week	Topic	Reading
9	Binary Tree, Terminology, insertion, deletion, and traversal	Chapter 5
10	Binary Threaded Tree, IBTT, insertion, deletion, and iterative traversal	Chapter 5
11	Priority Queue and Heaps, min and max heaps	Chapter 5
12	Binary Search Tree, insertion, deletion, and traversal	Chapter 6
13	Graph, array and linked list based, directed and weighted graphs, implementation, shortest path	Chapter 6
14	Searching and Hashing	Chapter 8
15	Sorting	Chapter 7
16	Sorting and Course Review	Chapter 7
	Final Term	

Time Table

CS-208 Data Structure Time Table

	DAYS	8:00-9:30	9:30-11:00	11:00-12:30	12:30-2:00	02:00-03:30	03:30-05:00	05:00-06:30
Morning	**Monday**		Lecture			Counseling		Counseling
	Tuesday	`				Counseling		Counseling
	Wednesday		Lecture			Counseling		Counseling
	Thursday					Counseling		Counseling
	Friday	Contents Development	Tutorial					
	Saturday	Lab	Lab					
	DAYS	**8:00-9:30**	**9:30-11:00**	**11:00-12:30**	**12:30-2:00**	**02:00-03:30**	**03:30-05:00**	**05:00-06:30**
After A	**Monday**					Counseling		Counseling
	Tuesday		Lecture			Counseling		Counseling
	Wednesday					Counseling		Counseling
	Thursday		Lecture			Counseling		Counseling
	Friday	Contents Development	Tutorial					
	Saturday			Lab	Lab			
	DAYS	**8:00-9:30**	**9:30-11:00**	**11:00-12:30**	**12:30-2:00**	**02:00-03:30**	**03:30-05:00**	**05:00-06:30**
After B	**Monday**	Lab	Lab	Counseling				Counseling
	Tuesday			Counseling	Lecture			Counseling
	Wednesday			Counseling				Counseling
	Thursday			Counseling	Lecture			Counseling
	Friday	Contents Development		Tutorial				
	Saturday							
	DAYS	**8:00-9:30**	**9:30-11:00**	**11:00-12:30**	**12:30-2:00**	**02:00-03:30**	**03:30-05:00**	**05:00-06:30**
After C	**Monday**	Lab	Lab	Counseling	Lecture			Counseling
	Tuesday			Counseling				Counseling
	Wednesday			Counseling	Lecture			Counseling
	Thursday			Counseling				Counseling
	Friday	Contents Development		Tutorial				
	Saturday							

Part 2

Lecture Notes

2.1. Lecture Plan

Here is the lecture plan for the course

Lecture Notes 0A – Review Programming History
Lecture Notes 0B – Review of Object Oriented Technology
Lecture Notes 01 – Overview of Data Structures
Lecture Notes 02 – Algorithms and Recursion
Lecture Notes 03 – Performance analysis and Measurement
Lecture Notes 04 – Array and Polynomial ADTs
Lecture Notes 05 – Sparse Matrix and String ADT
Lecture Notes 06 – Bag and Stack ADT
Lecture Notes 07 – Expression Evaluation
Lecture Notes 08 – Recursion
Lecture Notes 09 – Queue ADT and Applications
Lecture Notes 10 – Singly Linear Linked List
Lecture Notes 11 – Reusable Linked List
Lecture Notes 12 – Circular and Doubly Linked List
Lecture Notes 13 – Variants of Linked List
Lecture Notes 14 – Pre Mid Review
Lecture Notes 15 – Binary Tree
Lecture Notes 16 – Binary Trees (cont.)
Lecture Notes 17 – Threaded Trees
Lecture Notes 18 – Priority Queues
Lecture Notes 19 – Heaps
Lecture Notes 20 – Binary Search Tree
Lecture Notes 21 – AVL Tree
Lecture Notes 22 – AVL Tree (cont.)
Lecture Notes 23 – Graphs
Lecture Notes 24 – Graph Operations
Lecture Notes 25 – Graph Operations (cont)
Lecture Notes 26 – Hashing
Lecture Notes 27 – Sorting
Lecture Notes 28 – Sorting (cont)
Lecture Notes 29 – Sorting (cont)
Lecture Notes 30 – Pre Final Review

2.2. Lecture Notes 0A – Review Programming History

Lecture Outline:

- B Sc. Computer Science four year Degree Program
- History of Programming Languages

B. Sc. Computer Sciences

- B. Sc. Computer Sciences is a four-year degree program.
- You have completed 25% of your degree.
- You have to change your attitude towards your studies.
    - Life and Profession
    - Science and Technology
    - Computer Sciences and Information Technology
    - IT as Profession
    - Why Slump in IT, which is responsible, our responsibilities.
    - Trends and Future of IT
- Degree program can be divided into three period
    - Recruitment Period: It is the first 50% of your studies you can afford the luxury to feel that you are the students of entire computer science filed. You have to effort to get the knowledge for its each and every bit. You have to get as maximum understanding as you can.
    - Field Selection Period: Try for select an appropriate filed of in next 25% of degree time. Recruitment in the first period will help you a lot to take a right decision according to your aptitude.
    - Implementation Period: After the selection of a field for your carrier you have to show your achievement in it through your research and its implementing in 1-year final degree project.
- Comprehend and thoroughly Study the rules and regulations (statues) that governs you in your degree program, to take advantage of each clause of it.
- Read thoroughly your Roadmap and be convenient with it. Discuses if every thing that fund difficult and confused.
- Survey the standard curriculum for Computer Sciences provided by ACM (Association of Computing Machinery) and compare it with your studies notify if any thing found perplexed and confused.
- Review the courses that you have studied so far and try to prepare a handbook (of 3-4 pages) for each subject at your own that crystallize objectivity and achievements of the course (might include)
    - Course objective and its requirements.
    - The text followed and contents covered.
    - Coming subject that follows this course.
    - Your interest and aptitude towards this course and concept.
    - Your achievement in terms of extra studies and projects etc.

This activity may look too formal but it is required that will help to give confidence (about the thing that you have covered and discovered) and make you conscious about your weaknesses if any in any subject and filed.

History of Programming

- A solid attentions to words computer and computer programming is given after World War II due to heavy mathematical and computational activities encountered by the humankind i.e.
    - Calculation and record keeping of casualties and damages.
    - Mathematical computation for precision of different arms i.e. project motion of missiles.
    - Decryption of enemy messages etc.
- First generation of PL was unstructured computational languages (initial FORTRAN)
- After some time, programmers realize that it is difficult to implement large-scale programs using unstructured gotos. This gave birth to the control structures and well indented structured programming.
- When programmers had to assemble same tasks at different places of the program repetitively they decided to design procedures to perform them using a single call rather then to write same code at all places.
- Procedural programming languages made programming easy by providing set of libraries to perform verity of tasks.
- Soon after the dawn of functional and procedural programming, programmers recognized that they could call a function several times but that function cannot save its state at each call separately due to local variables and their lifetime. They introduced static variables and global arrays to save the state of each call, but this state was unsecured and troublesome due to their free access for whole program.
- Ultimately it was decided to keep state and functionality at the same place to provide encapsulation, security, and abstraction, the origin of object-based programming.
- Now the programmers had achieved a natural approach to program the problem, to make it more realistic, they introduced the concepts of inheritance that converted object based programming to object oriented programming.

2.3. Lecture Notes 0B – Review of Object Oriented Technology

Lecture Outline:

- Object Oriented Paradigm
- Read Tutorial Object Technology
- Object Oriented System Development
- Example Application
- C++ Review

Object-Oriented Paradigm Review

Let us review the object-oriented paradigm.

- Systems are made up of objects interacting with each other.
    - Example: students/courses/teachers/classrooms database
    - Benefit: Reuse.
- Each object is an instantiation of a class. Classes are related through inheritance and other types of relationships.
- OOP is built on two principles:
    - *Data encapsulation*: hiding of implementation details from user.
    - *Data abstraction*: separation of a data objects *specification* from its *implementation*.
- Benefits in the System Development Life Cycle:
    - **Divide** development into independent subtasks.
    - **Debug**ging and testing can also be divided.
    - **Change** internal representations without changing interface.
    - **Reuse** components.
- Key: Modularity

Object Oriented System Development:

- While analyzing the system you have to deal with nouns and verbs. Nouns are objects belonging to classes and verbs are their actions. Nouns belonging to nouns are their attributes.
    - Example: students/courses
    - Name of the student, title of the course
    - Course register
- In designing keep the abstraction principle under consideration
    - Write specification of classes independent of their implementation
    - Provide all required interfaces to programmer
    - May use UML (unified modeling language) for design purpose
- During implementation keep following issues in mind
    - Classes should be reusable, Secure, Intelligent

- To achieve above objectives you have to pass signals between objects and application. Avoid **cout** in member function implementation. Use access specifiers, utility functions, and principle of least privilege for security. Write appropriate logic in member functions using control structures to make them intelligent against different actions.
- Use standard naming conventions while writing your classes and code.
  - Class name must be singular noun strating with **C** and then in proper case, i.e. CPlayer
  - Follow Hungarian notation for naming purpose.
- Place classes in different files and use preprocessor directives to avoid their re-inclusion in the main code.

Example Application:

Windows based application interface.

- You require interface in each of your application. Standard interface for the application is windows based interface. Which receives massages from the user through mouse and keyboard and respond accordingly.
- Example: A simple windows based application interface may include the following classes and
  - CWindow
  - CApplication, CButton, etc
  - CMessage

  CWindow is an abstract class and all displayable objects i.e. CApplication, CButton, etc are inherited from it. CMessage is responsible for user input.
- In the **main()** function application and a message objects are instantiated all components that are required in the application i.e. buttons, textboxes, static text areas etc. are also instantiated from their respective class and added to main application using member function of m_fnAddItem(CWindows *). Address of these items are stored in a state variable of CApplication named

  ```
  CWindow * m_paApplicationItems []
  ```

  that can store addresses of all objects that are derived from CWindows class. Than the potential of polymorphism can be used for all displayable components that are inherited by CWindows class.
- A message loop is responsible for taking and dispatching message that will be stored in CMessage object to CApplication object. CApplication object receives message and sends it to its customizable (whose implementation can be overridden by the main() programmer) function

  ```
  Bool m_fnActionListener(const CMessage &)
  ```

- Each displayable class contains an inherited customizable function

Bool m_fnActionListener(const CMessage &)

from parent class CWindows.

- CApplication object sends this CMessage object to all of its associated components using a single for loop through their action listeners.
- If this message is for any of these components then they will respond against it through their overloaded action listener function.
- If none of the component responds to the message, it is the responsibility of CApplication to show its default or customized behavior.

C++ Review

Most of section 1.4 should have been covered in CS-205. Review on your own as needed. See me if you have questions.

2.4. Lecture Notes 01 – Overview of Data Structures

Lecture Outline:

- Course Outline
- Read Tutorial Fundamentals of Programming Languages
- Programs and Data Structures
- Abstract Data Type (ADT)
- Assignment 1

Course Outline

- See the course outline for details.
- In this course, we will learn how to develop, choose, and implement abstract data types within an object-oriented framework. We will study in detail a number of standard data structures, considering their advantages and disadvantages and appropriate uses.
- The underlying goal is to develop programming skills that can be applied to the construction of large, complex systems as are found in "real life."
- *Collaboration*: You may discuss approaches and difficulties, but you may not share solutions, code, or logic.
- We will be covering a great deal of material in this course. Should you feel like you are falling behind, please come and talk to me.

Programs and Data Structures

- Program: A set of instruction that are executed on a set of values in a specific sequence to achieve desired task.
    - Program = Values + Instructions
    - By now you have developed some proficiency with programming. However, actual systems are often large and complex, and their development is often distributed.
    - Implications:
        - Need methodology.
    - Performance becomes important.
    - Program = Data Structures + Algorithms
- Data Structures: It is the subject that deals with the storage and organization of data so that it can be retrieved efficiently and effectively.
- Data type can completely be defined using two sets
    - Set of possible values that can be stored
    - Set of possible operations that can be performed
- Data structure is an extensions of data type, it is a user defined data type
- Appropriate languages to implement data structures are those which support following characteristics
    - Provide a facility to design new data types.

  - Provide a facility to encapsulate both set of values and set of operations as a single package.
  - C++ is a programming language that provides both features.
- Facilities of C++ that will be used during course
  - Classes and their associations
  - Operator Overloading
  - Pointer and Dynamic Memory Allocation
  - Templates

Software Development Life Cycle

- stages
  - *requirements*: understanding and making precise and complete the system's input/output specifications
  - *design*
    - top-down: standard divide-and-conquer approach
    - modular: determine objects and operations performed on them, i.e., define abstract data types, and their interactions
    - abstract level: language and implementation-independent; postponing details until next phase maximizes flexibility
  - *implementation*
    - choosing object representations and operation algorithms
      - Choice of one affects choice of the other
      - Performance analysis used to aid choice
      - Space-time tradeoff
    - coding: should be well-documented and modular
      - modular: beneficial when redesigning, debugging, testing, maintaining, and for reuse
      - well-documented (including good naming conventions): should be easily understood by others or yourself later on.
  - *verification*
    - correctness proofs: help minimize errors, but often time-consuming and difficult in practice; strategy - use standard algorithms when possible
    - testing: need to check all scenarios; best (but sometimes psychologically difficult) to test each component of the system as it is written.
    - debugging: often consumes largest portion of cycle time; facilitated if program is well-documented and consists of independently tested modules.
    - performance measuring
  - *maintenance*: a primary motivation for using good software development skills.
- These stages are not necessarily (or usually) linear or mutually exclusive.
  - Design, implementation, and verification often interleaved.

- Discoveries in later stages often affect decisions in earlier stages. For example, we often decide to switch designs during the implementation stage. This is made easier when a top-down, least-commitment strategy has been followed
- Correctness proofs can be done during design and implementation phases

Abstract Data Types

- *Data type*: a collection of objects and a set of operations acting on those objects. Examples: ints, chars, floats, arrays, strings, etc.
- *Abstract Data Type*: a data type where the specification of the objects and operations are separated from their implementation. Specification of data type should be independent of its implementation i.e. it should not contain implementation details
- Example: Students/instructors/courses database
  - Objects: students, instructors, courses, rooms
  - Operation examples: register for a course (student), reserve a room (room), check if a course is full (courses).
- No standard mechanism to write ADT
- You are supposed to follow the style of your text book for writing ADTs

Assignment 1:

Write ADTs and C++ implementation for the following data types

1. Boolean
2. Rational Number
3. Complex Number
4. Vector
5. String

Date of announcement:	after Lecture-1	Week-2
Date of submission:	before Lecture-3	Week-3

2.5. Lecture Notes 02 – Algorithms and Recursion

Lecture Outlines

- Algorithm specification
- Correctness proofs
- Recursion

Algorithm specification

- Algorithm design is inextricably connected with data structure design. This is because the choice of data structure affects the feasibility, choice, and performance of the algorithms used to implement the desired object operations.
- You will study algorithm design more closely in a later course, I believe, but it is still an important consideration when designing data structures so we take a look at it today.
  - **Definition** *Algorithm*: a finite set of instructions that, if followed, accomplish some task. An algorithm must satisfy the following criteria:
    - Input: Zero or more input.
    - Output: At least one output.
    - Definiteness: Clear and unambiguous instructions (and I/O).
    - Finiteness: In all cases, the algorithm terminates after a finite number of steps.
    - Effectiveness: Feasible instructions.
  - Programs differ from algorithms in that they don't necessarily terminate.
  - To turn a problem specification into an algorithm (and, in turn, into a program), it helps to use a top-down, divide-and-conquer approach. Pseudo-code can come in handy during this process.
  - Two classes of algorithms we will have special interest in this semester are sorting algorithms and searching algorithms.
  - Example: selective sort
    - *Problem*: sort a collection of items.
    - *Specification*: Assume the items are `n` integers represented as an array `a`. We return the sorted array.
    - *Approach*: selection sort, i.e., iteratively check the set of unsorted integers for the smallest one and place it in the sorted list.
    - *Algorithm*: Program 1.5 (p. 22)
    - *Program*: Program 1.6 (p. 22)
    - *Correctness*: We use the two-step process of finding an invariant that the program satisfies and showing that the invariant implies that the result is correct. Invariant – after swap of ith item, $\forall$ `j < k ≤ i.a[j] <= a[k]`.

      Informally, this invariant holds since the swap procedure puts the ith integer in its correct place and none of the previous integers are moved. Obviously, if the program satisfies the invariant, the program will output

the correct result since the invariant will hold in the particular case where i=n+1.
    - Note that this is an induction-like proof and, in fact, this problem could easily be solved using recursion.
- Example: binary search
    - *Problem*: Determine if an item occurs in a given group of items.
    - *Specification*: We assume the items are integers and that the group is represented as a sorted array `a`. The item we want to search for is `x`. We return bool `true` if the integer is found, `false` otherwise.
    - *Approach*: binary search, i.e., repeatedly divide list in half and check which half the item is in.
    - *Algorithm*: Program 1.8 (p. 24) except we return a bool.
    - *Program*: Program 1.9 (p. 24) except we return a bool.
    - *Correctness*: Beginning of proof – at entrance to first for loop, if `x` is present, the following invariant holds:

`Left ≤ right ∧ a[left] ≤ x ≤ a[right] ∧ a is sorted`

Correctness proofs

- Tips:
    - Find invariants.
    - Try on small examples first.
    - Use induction.

Recursion and induction

- *Recursion*: a function calls itself or calls other functions that call it.
- Any program that using assignment, if-else, and while can be written using assignment, if-else, and recursion.
- Use induction to prove correctness.
    - Understand desired solution.
    - Check termination occurs.
    - Check termination computation is correct.
    - Check recursive case computation is correct.
- Appropriate uses of recursion:
    - Problem is defined recursively, e.g., the factorial.
    - Data structure is recursive, e.g., lists, trees.
- Advantages:
    - Clarity and conciseness, especially for recursive problems.
    - For some problems defined recursively, it may be extremely difficult to define a non-recursive solution.
- Disadvantages:
    - May require a lot of memory to hold intermediate results.
    - Additional overhead time spent in function calls.

- Example: binary search (p. 27)
  - *Algorithm/program*: Program 1.10 (p.27), except we return a bool.
  - *Termination occurs?* Yes, since BinarySearch is called on successively shorter lists so that eventually each list will only have one member. If this member is not x, then `false` will be returned.
  - *Termination condition correct?* Yes, if `x` is found, true is returned, otherwise `false` is.
  - *Recursive condition correct?* Yes, if x is less or greater than the middle element, the result of searching the left or right sublists, respectively, is returned
- In-class exercise: computing factorial and binomial coefficients

2.6. Lecture Notes 03 – Performance analysis and Measurement

Announcements

- The second printing (and maybe others) of the textbook has an error on page 46. The last line of paragraph 3 should read "$t_{rsum}(n)=2n+2=\Theta(n)$…" not "…$\Theta(mn)$…."
- Subject Mailing list: datastructure2002@yahoogroups.com
- Semester mailing list: BCSS02@yahoogroups.com

Today's Schedule

- Performance analysis
- Performance measurement

Performance Analysis and Measurement

- Performance analysis refers to the process of estimating the complexity of a program. It is often done as part of the design phase of software development, especially as a way to choose best among competing approaches.
- Performance measurement refers to doing actual testing to measure the programs performance.
- *Space complexity*: a program's memory requirements.
- *Time complexity*: a program's time requirements.
- **Important:** Complexity is in terms of some input characteristics that we must choose. Note that our choice may not be able to fully characterize complexity. Solution: use best-, worst-, or average-case complexity.
- In both space and time cases, complexity is made up of a constant part and a variable part that is dependent on the characteristics of the inputs and outputs. We are usually interested in the latter part because it is that part, which matters for large applications. Typical characteristics we are interested in are the number of and the size of the inputs.

Space Complexity

- Space complexity: Memory space needed for component and referenced variables whose size depends on the input characteristics of interest, and the system stack during recursion.
- **Example:** Program 1.13 (p.31)

Time Complexity

- Time complexity = compile time + run time. We're generally only interested in the latter since the former doesn't depend on the input characteristics and is a one-time cost.

- We generally can't determine time exactly beforehand: it depends on the machine, compiler, arguments to operators, system load, memory configuration, network activity, etc. Even if we measure experimentally, these factors will make identical runs to have different time complexities.
- Exact numbers are not very correct and useful anyway. Estimates are useful, however.
- We might count operations, but this is usually more detail than we need.
- We may **count program steps**, that is, segments of the program that are independent of the input characteristics. Essentially, one step per unit of work.
- Example counts:
    - *Expressions* get a count of 1 unless they involve function calls, in that case they get the sum of the function call counts.
    - *Assignments* get the count of the right expression plus the size of the left expression if it depends on the input characteristics.
    - *For loops* get 1 count per iteration unless its arguments are a function of the input characteristics, in which case count the cost of the initializer once, the cost of the check once per iteration, and the cost of the iterator once per iteration minus 1.
    - *Function invocations* get a count of 1 unless the sizes of pass-by-value arguments depend on the input characteristics, in which case we sum their sizes. If there is recursion, add sizes of local variables as well.
- One easy method is to construct a table listing for each line count/execution and frequency, multiplying them, and summing up the results.
- **Example:** Program 1.13 (p. 31)
- Finding a non-recursive expression for the complexity of a recursive function is not always easy. Techniques for doing this involve solving recurrence relations.

Asymptotic Notation

- Determining step counts is often difficult. It is admittedly somewhat arbitrary, so the exact counts don't mean much anyway.
- Often it is sufficient to give a general bound on the complexity that ignores the constants.
- **Definition:** $f(n) = O(g(n))$ iff there are positive constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.
- That is, when n gets large enough, f(n) will asymptotically grow no faster than g(n); g(n) "bounds" f(n) from above.
- **Example**: $2n^2+100n+83 = O(n^2)$ – $c=3$, $n_0=101$
- Usually suffices to drop low terms of f(n) and find constant larger than that of largest term(s).
- In practice, we usually use the smallest g(n) we can come up with. If something is O(n), we usually don't say it is $O(n^8)$ although this is correct.
- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$

- We also try to choose the simplest g(n)'s we can get away with. For example, coefficients are usually chosen to be 1 since they can be bundled into the hidden constants.
- **Definition:** $f(n)=\Omega(g(n))$ iff there are positive constants c and n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$.
- That is, g(n) asymptotically bounds f(n) from below.
- **Definition:** $f(n)=\Theta(g(n))$ if both $f(n) = O(g(n))$ and $f(n)=\Omega(g(n))$. (Note: The hidden constants may be different for the two.)
- Intuitively, g(n) precisely characterizes the rate of growth of f(n).
- A programs asymptotic complexity is often much easier to come by than the step count.
- **Important:** "Sufficiently large n" means that although program A may have a higher asymptotic complexity than program B, it may actually be faster than program B for the problem instances we are interested in.
- The more general definitions in terms of multiple input characteristics (such as n and m) are defined in the obvious way (e.g., $f(n,m) = O(g(n,m))$ iff there exist c, n_0 such that $f(n,m) \leq cg(n,m)$ for all $n,m \geq n_0$.
- In above all examples g(n) and f(n) can roughly be treated as program and complexity respectively.
- O is worst case, Ω is best case and θ is average case complexity.
- For more details chapter 1-3 see of Introduction to Algorithms, 2nd *Edition MIT Press,* by Thomas H. Cormen

Example

- Consider the following functions which return the sum of first n integers
-
```
1. int sum ( int n ) {
2. return (n * ( n + 1 ) / 2 );
3. }
```
-
```
1. int sum ( int n ) {
2. int sum = 0;
3. for (int i = 1 ; i<=n ; i++)
4. sum +=  i;
5. return (sum);
6. }
```
-
```
1. int sum ( int n ) {
2. if (n == 0 )
3. return n;
4. return (sum(n-1) + n);
5. }
```
- All functions returns the same output against all positive integers but they have different time and space complexities.

Additional Material
Binomial coefficients solution

- We assume n ≥ m. For each solution we simply call binomial (n,m).
- Recursive solution:

```
1 int binomial(int n, int m) {
2       if (n==m || m==0) return 1
3       else return (binomial(n-1,m) + binomial(n-1,m-1));
4 }
```

- Brute force iterative solution:

```
1 int binomial(int n, int m) {
2       int i;
3       int nf=1;
4       for (i=1; i<=n; i++) mf *= i;
5
6       int mf=1;
7       for (i=1; i<=m; i++) mf *= i;
8
9       int nmf=1;
10      for (i=1; i<=n-m; i++) nmf *= i;
11
12      return (nf/(mf*nmf));
13 }
```

  Takes about 2n steps.

- Faster iterative solution:

```
1 int binomial(int n, int m) {
2       int i;
3       int d1=1;
4       for (i=1; i<=min(n-m,m); i++) d1 *= i;
5
6       int d2=d1;
7       if (n-m != m)
8               for (i=min(n-m,m)+1; i<=max(n-m,m); i++) d2 *= i;
9
10      int n1=d2;
11      if (max(n-m,m) != n)
12              for (i=max(n-m,m)+1; i<=n; i++) n1 *= i;
13
14      return (n1/(d1*d2));
15 }
```

  Takes about n steps.

2.7. Lecture Notes 04 – Array and Polynomial ADTs

Announcements

- Bring textbook to class.
- Reminder: You are expected to be able to do all the exercises in the book, whether or not they have been assigned.

Today's Schedule

- Performance measurement (continued)
- ADTs
- Array ADT
- Ordered List ADT
- Polynomial ADT

Performance Measurement (continued)

- We will focus on time complexity.
- Insert timing code in program.
- Note that timing code itself takes time and may need to be accounted for.
- Important to carefully choose input values depending on the kind of measurement we are doing (best-, worst-, or average-case). If not obvious, run a number of experiments over a wide range.
- Results may not exactly follow asymptotic curve because input may be too small.
- For each input instance, run multiple times and average, particularly for inputs that are too small to be exhibiting asymptotic behavior.
- For short events, need to worry about subtracting out timing function time, loop time, etc., especially if we need absolute times for predictive purposes rather than relative times for comparative purposes.

ADTs

- Note: review section 2.1 on your own as necessary.
- Review: An ADT is a data structure where the specification is separated from the implementation. They allow us to abstract away the details of implementation, an important tool in developing complex software systems.
- Questions we ask of each ADT:
    - What is the specification/definition of the ADT? What operations should it support?
    - What are the possible representations for the ADT?
    - What are the implications of each representation for the space and time complexity of the ADT operations? What is the best we can hope to achieve with each choice of representation? Can we improve the complexity by exploiting domain structure?

- What are the tradeoffs between space complexity and time complexity, code and complexity, etc., for the different choices of representation? Can we take advantage of these?
- What is the actual complexity of our implementation of each operation?
- What kinds of applications the ADT is most appropriate?

Array ADT

- The first ADT we will consider is something that should be fairly familiar to everyone: the *array*.
- At first cut, we may think of an array as a contiguous series of memory locations, but this is very implementation dependent.
- Abstractly, an array is a set of pairs <index, value> such that each defined index has a value associated with it and the indices are ordered.
- The C++ array assumes the set of indices to be consecutive, non-negative integers starting with 0. However, in general, the indices could be an arbitrary finite, ordered set. This includes multi-dimensional indices.
- An array should support the operations of retrieve a value given an index and store a new value given an index. The C++ array supports both of these.
  - Retrieve: x = A[i]
  - Store: A[i] = x
- Other operations: assignment, size, union (need to address multiple values for a given index).
- Because of the C++ representation as a set of contiguous memory locations where indices are consecutive, non-negative integers starting with 0, the implementation of these operations is very fast; for example, for integer members of a 1-dimensional array, A[i] = *(A+i).
- How fast are retrieve and store for C++ arrays? Retrieve and store take constant time for C++ arrays.
- In addition, we might want an array to support bounds checking in each of these operations, that is, making sure that the index is in the index set. C++ arrays don't do this.
- Let us specify the general class ADT (:

```
template <class Type>
class GeneralArray {
// Objects are a set of pairs <index, value> where for each value of index in finite,
// ordered set IndexSet, there is some value of type Type associated..
public:
    // Constructor creates an array where indices are drawn from indexSet.
    GeneralArray(integer Size, Type initValue = defaultValue);

    // If i is in IndexSet, return the corresponding value in the array, otherwise
    // signal an error.
    Type Retrieve(Index i);
```

```
    // If i is in IndexSet, set the corresponding value in the array to be x, otherwise
    // signal an error.
    void Store(Index i, Type x);

    // Destructor.
    ~GeneralArray();
}
```

- Here we've made no assumptions about the index set. In the book, they assume that the index set is multi-dimensional and, thus, specify it with two parameters, the number of dimensions and a list of ranges.

Ordered List ADT

- Another very important ADT is the *ordered list*.
- An ordered list is a sequence of items where each item is indexed by its position in the sequence.
- Common operations for a list of size n:
  - Retrieve element at position $0 \le i < n$.
  - Store a new element at position $0 \le i < n$.
  - Insert a new element at position i, shifting elements i and above up by 1 and incrementing n.
  - Delete element at position $0 \le i < n$, shifting elements $i+1$ and above down by 1 and decrementing n.
  - Compute the length of the list.
  - Read list from right to left or visa-versa.
  - Concatenate two lists.
- We will use one ADT to implement another. We use an array to implement a list.
- Which operations are fast, which are slow? Retrieval, storing, length, and listing are fast, O(1) for the first three, O(n) for the last. Insertion, deletion, and concatenation are slow, O(n) for the first two, O(m) for the last where m is the length of the list to concatenate. We will see later in the course that other representations make these operations faster.

Polynomial ADT

- Let us consider an important ADT that motivates the need for an efficient implementation of an ordered-list ADT: the *polynomial ADT*.
- An example of a polynomial is $A(x) = x^4+3x^2+2x+4$. A polynomial can be characterized as a mapping from the set of non-negative integers (the exponents of each term) to coefficients. The largest exponent with a nonzero coefficient is referred to as the *degree* of the polynomial.
- Operations: add, subtract, multiply, divide, evaluate, get degree, get coefficient given exponent, get root.
- Array representations

- Representation 1 (brute force): use an array of length MaxDegree+1 to store coefficients.

  private:
  int *degree;*
  float *coef*[MaxDegree + 1];

  Note that *a.coef*[*i*] = a_{n-i} where *n* = *degree*. This simplifies algorithms for many of the operations.

  Problem – wastes space when *degree* is much less than MaxDegree.
- Representation 2: dynamically allocate space for array once *degree* is known.

  private:
  int *degree*;
  float *coef;

  Problem – wastes space for sparse polynomials where there are many zero terms.
- Representation 3: use array of terms, where only terms with nonzero coefficients are stored. Also, use one array to store multiple polynomials.

```
class Term {
friend Polynomial;
private:
    float coef;
    int exp;
}

class Polynomial {
…
private:
    static Term termArray[MaxTerms];
    static int free;
    int start, finish;
}
```

  Usually preferable to other two representations, but takes more space if there are few zero terms.

  Problem – difficult to reuse space of polynomials that are no longer useful since moving polynomials around in an array is expensive.

  Problem – if we want each array to represent one polynomial (a more natural, encapsulated representation for many applications), we either potentially waste space if MaxTerms is too high, or waste time computing

the number of terms in each polynomial. For example, it takes as much work to determine the number of terms in the result of adding two polynomials as it takes to do the addition.

- Representation 4: Use linked lists – coming soon …

- Add function for representation 3:
  - Based on iterative comparison of exponents of the two functions: if equal, add the coefficients and insert into new polynomial, otherwise insert the term with the larger exponent. This function doesn't benefit from the decreasing exponent representation.
  - Analysis: The key observation is that in each iteration of the while loop, one of a or b are incremented, so the loop can be iterated at most m+n-1 times. The –1 is due to the fact that we exit as soon as one polynomial is exhausted. The remaining loops serve to exhaust the remaining polynomial, so the time complexity is O(m+n).

2.8. Lecture Notes 05 – Sparse Matrix and String ADT

Today's Schedule

- Sparse Matrix ADT
- String ADT

Sparse Matrix ADT

- A matrix consists of *m* rows by *n* columns of numbers.
- Sparse matrices are matrices with many zero entries. Large, sparse matrices are very common in many applications e.g., the matrix representing a large image taken by a NASA spacecraft.
- Operations: access element, change element, transpose, add, multiply, etc.
- Standard two-dimensional arrays waste a lot of space if used to store sparse matrices. Preferable to only store the nonzero entries.
- Representation (also consider other representations which are briefly discussed in class as well): Array of objects corresponding to nonzero terms represented as <*row*, *col*, *val*> triples and stored by row then by column. Also store number of rows, columns, and terms.

```
class MatrixTerm {
friend SparseMatrix;
private:
        int row, col, val;
}

class SparseMatrix {
…
private:
        int numRows, numCols, numTerms;
        MatrixTerm smArray[MaxTerms];
}
```

- Transpose
  - Brute force: iterate by row, taking each element (*i,j,val*) and storing it as element (*j,i,val*) in transpose. Problem – need to shift around elements a lot.
  - Better approach: iterate by column. Since rows are in order, column elements will be found in the correct order as well.
  - Problem - Computing time is O(*numTerms* x *numCols*) which, in the worst case where the matrix is not too sparse, is O(*numRows* x $numCols^2$), much more expensive than if we had used the standard O(*numRows* x *numCols*) algorithm on a two-dimensional array.
  - FastTranspose solution: use extra storage. Compute the number of terms in each row of the transpose and use additional array *RowSize* to store them. This allows us to know where each row begins. With the help of an additional array *RowNext* (called *RowStart* in the textbook) to keep track of the next free spot for each row, it is easy to compute the location in the transpose for each term.

  - Computing time is now O(*numCols* + *numTerms*) which is O(*numRows* x *numCols*) in the worst case. This is the same as the standard approach, but note that the hidden constant for FastTranspose is going to be much higher so that FastTranspose is really only appropriate if the matrices are expected to be sparse.
  - Note space/time tradeoff. However, also note that if *numTerms* is sufficiently small, we save both space and time over the standard two-dimensional array approach because the additional arrays take much less space than the space that would be used to store zeros. Furthermore, we can use the same space to store both arrays *RowSize* and *RowNext.*
- **Multiplication**
  - Issue: Multiplication of sparse matrices may not be sparse.
  - Algorithm transposes second matrix then "merges" the two.
  - Study the algorithm and subsequent analysis for tricks you can use in your own designs and analyses. Note the use of case analysis to determine the time complexity. Also note the use of a variable tr that later gets summed out.
- Disadvantage of representation: access is slower than with an *m*x*n* array. Using an array to store the number of terms in each row is one possible solution to this problem.
- Disadvantage of representation: we still waste space if MaxTerms is much less than numTerms. We may consider using one array to represent multiple sparse matrices, but, as in the case of polynomials, it is difficult to allocate space from the array for new matrices.
- Alternative representation: one array per matrix with space dynamically allocated. Problem – not always easy to know how much space to allocate, e.g., as in the case of polynomials, computing the space needed for the result of addition essentially requires computing the addition.
- We'll encounter these difficulties again when we study stacks and queues.

String ADT

- A string is a sequence of characters S=s0,…,sn-1 where n is its length.
- Operations: creation, reading/printing, concatenation, comparing, searching for a substring (pattern matching), inserting/deleting a substring, get length.
- Representation: char* array.
- Pattern matching
  - Problem: Find string p in string s.
  - **Brute force:** traverse s, at each point comparing the substring beginning with that character to p.
  - Time complexity: O(length(p)*length(s))
  - Problem: We are potentially redoing a lot of work by reconsidering characters in s we've already seen.
  - One solution: use heuristics that will make performance faster for certain combinations of s and p. However, the worst-case performance will still be O(length(p)*length(s)).

- Ideally, we want an algorithm with complexity O(length(p)+length(s)) since, in general, we have to consider each character in each string at least once.
- The **Knuth-Morris-Pratt algorithm** does just this by using a little more storage space. The algorithm computes and stores a function that measures, for each character p[j], the size f(j)+1 of the longest strict substring of p[0…j] (i.e., it is smaller than p[0…j]) that ends in p[j] and is also a prefix of p (thus, if no such substring exist, it returns −1). Then, whenever a comparison between s and p fails at s[i] and p[j], it is sufficient to start the next comparison between s[i] and p[f(j-1)+1], unless j≠0, in which case we must start by comparing s[i+1] and p[0].
- **Correctness proof:** Let us analyse why this works. Use the following example for intuition.

```
   s[i-j] s[k] s[i]
      ↓     ↓   ↓
s = … a b a a b ? ? ? … ?
p =   a b a a b a a a
      ↑         ↑
    p[0]      p[j]
```

Suppose $s[i] \neq p[j]$ ('a' in the above example), so we need to move p to the right with respect to s before we start comparing again. We consider three cases.

***Case 1*:** Suppose $j=0$. Then $p[0] \neq s[i]$, so we obviously need to compare $p[0]$ against $s[i+1]$ next.

***Case 2*:** Suppose there exists k such that $i-j+1 \leq k \leq i-1$ and the substring starting at $s[k]$ matches p. But then $s[k \ldots i-1]$ must be a prefix of p (i.e., it matches the first $i-k$ characters of p). Thus, if we move p down so that $p[0]$ lines up with $s[k]$, it is sufficient to start comparing $s[i]$ against the corresponding character in p, that is, $p[i-k]$. To ensure that we don't accidentally miss a potential match, we find the smallest such k. This k corresponds to the largest prefix of p that ends at $s[i-1]$ and is a strict substring of $s[i-j \ldots i-1]$. But we know that $s[i-j \ldots i-1]$ matched $p[0 \ldots j-1]$, so it also corresponds to the largest prefix of p that ends at $p[j-1]$ and is a strict substring of $p[0 \ldots j]$. The length of this prefix is precisely $f(j-1)+1$ which, thus, equals $i-k$. Therefore, $p[i-k]$ (which we will next be comparing $s[i]$ against) is $p[f(j-1)+1]$.

***Case 3*:** Otherwise, there is no such k, so we need to shift p all the way to the right and start comparing $p[0]$ against $s[i]$. But since there is no such k and $s[i-j \ldots i-1]$ matched $p[0 \ldots j-1]$, there is no strict substring of p ending at $p[j-1]$ that is a prefix of p, so $f(j-1)=-1$. Thus, $p[0]$ (which we will next be comparing $s[i]$ against) is $p[f(j-1)+1]$.

2.9. Lecture Notes 06 – Bag and Stack ADT

Note:

- The second printing (and maybe others) of the textbook has an error on page 46. The last line of paragraph 3 should read "trsum(n)=2n+2=Θ(n)…" not "…Θ(mn)…."
- Review Sections 3.1 (templates) and 3.4 (subtyping and inheritance) on your own.

Today's Schedule

- Bag ADT
- Stack ADT

Bag ADT

- Container classes: classes that store a set of objects.
- We've already encountered two types of container classes: the array and the ordered list.
- One of the most general container classes is the bag. It imposes no constraints on location, duplicity of elements, or order of access. It supports insertion and deletion.
- Both arrays and ordered lists can be viewed as special kinds of bags with additional constraints on location, duplicity, and access order.
- Operations: insertion, deletion, check empty/full status.
- Representation: one-dimensional array.
- We insert objects at the end of the array and delete them from the center, but we could use any other policy we liked.
- Note that in the implementation of delete, a reference parameter is passed to be used as a return pointer to the deleted element so as to accommodate the situation where the bag is empty (in which case null is returned).
  - First note that we pass a reference parameter because for some element types, deletion may free the memory used by the element before it can be returned.
  - Also, although not necessary, the return type is explicitly stated to be a pointer to an element for the sake of transparency.

```
template <class BagType >
class CBag {
public:
CBag (int MaxNoOfElements = DEFAULT);
CBag (const CStack &);
void  Add(const BagType &);
Boolean IsFull();
Boolean IsEmpty();
BagType * Delete (BagType &);
```

```
        ~ CBag ();
private:
            int size;
            int Top;
            BagType *data;
        };
```

Stack ADT

- A stack can be viewed as a special kind of ordered list where insertions (called pushes) and deletions (called pops) are constrained to occur at one end, the top. This results in a Last-In-First-Out (LIFO) policy.
- Classical example: the system stack used to process function calls. Each function call results in an activation record being generated and placed on the stack. This activation record stores the function's local variables and parameters, return address, etc. The activation record is pop'ed when the function terminates.
- Operations: push, pop, check full/empty status.
- Representation: one-dimensional array with additional data members to store the top (index of the top element in the array) and the maximum size of the stack.
- We push and pop from the top and adjust top accordingly. In both cases, the time cost is O(1).

```
template <class StackType>
class CStack {
public:
CStack (int MaxNoOfElements = DEFAULT);
CStack (const CStack &);
Push (StackType);
Boolean IsFull();
Boolean IsEmpty();
StackType Pop();
        StackType SeeTop() ;
        ~ CStack();
private:
            int size;
            int Top;
            StackType *data;
        };
```

Disadvantage of this representation: wasted space when the stack is not full and difficulty of extending the stack if it does fill up.

2.10. Lecture Notes 07 – Expression Evaluation

Today's Schedule

- Applications of Stack
- Expression Validation
- Conversion of expressions from Infix to Postfix
- Expression Evaluation

Stack Application:

- Compiler: Expression validation and evaluation
- Graphics: Boundary fill algorithm etc
- Algorithms: Removal of Recursion
- Artificial Intelligence: Backtracking etc

Expression Validation:

- **Homogeneous Expression:** Expression that contains only one type of brackets.

```
Create an empty parentheses stack
Repeat
        Get the next token in the infix string
                If next token is '('
                        Push it onto parentheses stack
                If next token is ')'
                        If parentheses stack is empty
                                return FALSE
                        else
                        Pop from parentheses stack
                else
                do nothing
        Until at the end of the string
If parentheses stack is empty
                return TRUE
        else
                return FALSE
```

- **Heterogeneous expression:** Expression that contains different types of bracket.

```
Create an empty parentheses stack
Repeat
        Get the next token in the infix string
                If next token is '(' or '{' or '['
                        Push it onto parentheses stack
                If next token is ')' or '}' or ']'
                        If parentheses stack is empty
                                return FALSE
                        else if top of parentheses stack is not exact opposite
                        of token
                                return FALSE
                        else
                        Pop from parentheses stack
                else
                do nothing
        Until at the end of the string
If parentheses stack is empty
                return TRUE
        else
                return FALSE
```

Conversion of Expressions from Infix to Postfix:

```
Set postfix string to empty string
Create an empty operator stack
Repeat
        Get the next token in the infix string
                If next token is an operand (begins with digit),
                        append it to postfix string
                If next token is an operator
                Process the next operator
        Until at the end of the string
```

```
Process the next operator (next op)
repeat
        If operator stack is empty,
                push next op onto stack.
        Else if precedence (next op) > precedence (top operator)
                Push next op onto the stack  (ensures higher precedence
                operators evaluated first)
Else
                Pop the operator stack and append operator to postfix string
Until next op is pushed onto the stack.
```

Example 1:

Convert:

x – y * a + b / c becomes x y a * - b c / +

Operator Precedence: *, /
+, -

Effect of parentheses:

'(' indicates the start of an expression that is evaluated separately – just push it on the stack –Acts like a fence to separate the subexpression.

')' indicates the end of a subexpression. Pop off all operators and append them to postfix string until you reach a '(' and then pop '(' -- don't append ')' and '(' to postfix string.

To have this happen seamlessly in the algorithm, give parentheses the lowest precedence (,) - 0 precedence.

Process the next operator (with parentheses)

```
Set done to false
Repeat
If operator stack is empty or next op is ‘(‘,
push next op onto stack and set done to true
Else if precedence (next op) > precedence(top operator)
Push next op onto the stack  (ensures higher precedence operators evaluated first) and set done to true
Else
        Pop the operator stack
        If operator popped is ‘(‘,
set done to true
        Else append operator popped to postfix string
Until done
```

Example 2:

Convert: 3 + 5 * 6 – 7 * (8 + 5) to postfix

Precedence: *, /
+, -
(,)

postfix string is null string, operator stack is empty

next character is 3, postfix is "3", next token is +, push it onto empty stack

| +

next character is 5, postfix is "3 5", next character is *, push *

| + *

next character is 6, postfix is "3 5 6", next character is -, pop *pop +
postfix is "3 5 6 *",
| +
next character still -, pop +, postfix is "3 5 6 * +", operator stack is empty, push -

| -

next character is 7, postfix is "3 5 6 * + 7", next character is *, push *
next character is (, push (onto stack but give it lowest precedence

| - * (

next character is 8, postfix is "3 5 6 * + 7 8"
next character is '+', push it onto stack

| - * (+

next character is 5, postfix is "3 5 6 * + 7 8 5"
next character is ')', pop and append all operators to postfix string until matching '(' is on top of stack, then pop '(' and set done to true

postfix is "3 5 6 * + 7 8 5 +"

| - *
At end of string, keep popping stack until stack is empty

Postfix form: "3 5 6 * + 7 8 5 + * -"

Evaluation of Expressions:

Using 2 stacks you can evaluate an infix expression in 1 pass without converting to postfix first:

```
Create an empty operator stack
Create an empty operand stack
Repeat
        Get the next token in the infix string
        If next token is an operand, place it on the operand stack
        If next token is an operator
                Evaluate the operator (next op)
Until at the end of the string
while operator stack is not empty
        pop operator and operands (left and right),
        evaluate left operator right and push result onto operand stack
        Pop result from operator stack
```

Evaluate the operator (next op)

```
Set done to false
Repeat
        If operator stack is empty or next op is '(',
                push next op onto stack and set done to true
        Else if precedence(next op) > precedence(top operator)
                Push next op onto the stack  (ensures higher precedence operators
                evaluated first) and set done to true
        Else
                Pop the operator stack
                If operator popped is '(',
                        set done to true.
                Else
                        pop right and left from operand stack
                        evaluate left operator right
                        push result onto operand stack
until done
```

Example 3: 3 + 5 * 6 – 7 * (8 + 5) = using an operator stack and an operand stack. Operator = denotes the end of the expression and has precedence of –1.

Stack affected next character, action:

3 ,	3, push 3
+	next op is +, push +
3 5	5, push 5
+ *	next op is *, push *
3 5 6	6, push 6
+	next op is -, pop *
3 30	6, 5 popped, evaluate 5 * 6, push 30
	next op is -, pop +
33	30, 3 popped, evaluate 3 + 30, push 33
-	next op is -, push – (stack is empty)
33 7	6, push 7
- *	next op is *, push *
- * (	next op is (, push (
33 7 8	8, push 8
- * (+	next op is +, push +
33 7 8 5	5, push 5
- * (	next op is ‘)’, pop +
33 7 13	5, 8 popped, evaluate 8 + 5, push 13
- *	next op is ‘)’, pop ‘(‘ and discard

At end of string. One way to “detect” end of string is to use an operator symbol = with precedence of –1.Because its precedence is < anything else, all operators will be popped from the operator stack and evaluated automatically.

-	next op is ‘=’, pop * (at end of string)
33 91	pop 13, 7, evaluate 7 * 13, push 91
	next op is =, pop -
-58	91, 33 popped, evaluate 33 – 91, push -58
=	next op is =, operator stack is empty, push =
	pop result –-51

Pop operator stack – it should be empty or there is an error.

2.11. Lecture Notes 08 – Recursion

Today's Schedule

- Recursion and Recursive Definition
- Removing Recursion
- Complexity Analysis of Recursive Algorithm
- Examples
    - Recursive Power and Factorial
    - Fibonacci Numbers
    - Binomial Coefficients
    - Binary search
    - Palindrome Checker
    - Tower of Hanoi
    - Euclidian algorithms
    - Recursion to Iteration

Recursion and Recursive Definition

- Recursive Definition: a definition in which an object is defined in terms of a simpler case of itself is called recursive definition.
    - For a recursive definition to be correct it is required that, it not generate an infinite sequence of calls on itself
    - Instance of a recursive definition of a recursive algorithm must eventually reduce to some manipulation of one or more simple, non-recursive cases.
- *Recursion*: a function calls itself or calls other functions that call it.
- Direct recursion: if a *F* function calls itself.
- Indirect recursion: if a function *F* calls another function/s that calls *F*.
- Any program that using assignment, if-else, and while can be written using assignment, if-else, and recursion.
- Use induction to prove correctness.
    - Understand desired solution.
    - Check termination occurs.
    - Check termination computation is correct.
    - Check recursive case computation is correct.
- Appropriate uses of recursion:
    - Problem is defined recursively, e.g., the factorial.
    - Data structure is recursive, e.g., lists, trees.
- Advantages:
    - Clarity and conciseness, especially for recursive problems.
    - For some problems defined recursively, it may be extremely difficult to define a non-recursive solution.
- Disadvantages:
    - May require a lot of memory to hold intermediate results.
    - Additional overhead time spent in function calls.

Removing Recursion:

- To convert recursive procedure into equivalent iterative procedure, recursion can be divided into two type
    - Tail recursion: if a recursive procedure encounter recursive call as a last statement.
    - Body recursion: if a recursive procedure encounter recursive call within body of the code.
    - Tail recursion is always redundant.
    - Tail recursion can be converted into an equivalent iteration, just replacing base case if control structure into iterative control structure, inverting base conditions, and readjusting the value of control variables. (See example at end)
    - In body recursion, there are some instructions that executes before the recursive call and other afterwards.
    - Statements that execute after the recursive call require updated/processed values of local variables.
    - To convert a body recursion into iteration we use two loops, first loop processes those statements that executes before recursive call and pushes the values of local variables on the stack for each iteration. Second loop pops those values one by one and processes all those statements on currently popped data that should be executed after the recursive call. (See example at end)

Complexity Analysis of Recursive Algorithm

- Complexity analysis of recursive algorithm depends upon the solubility of its recurrence relation.
- General technique is to let T (n) be the number of steps required to carry out the algorithm on a problem of size n. The recursive part of the algorithm translates into a recurrence relation on T (n) whose solution is then the complexity function for the algorithm.
- Example: Complexity of Merge Sort is
- T (n) = T (n/2) + T (n/2) + O (n)
- T (n/2) steps required sorting half of the array, O (n) steps required to merge two arrays of size n/2

Examples:

- In-class exercise: Computing Power and factorial
- **Example 1: Fibonacci series**: a series whose first two elements are 0 and 1 then every element is the sum of previous two elements.
    - Fib (1) = 0, Fib (2) = 0, Fib (3) = Fib (3-1) + Fib (3-2)
    - Fib (n) = Fib (n-1) + Fib (n-2) $\forall$ n > 1
- **Example 2: binary search** (p. 27)

- *Algorithm/program*: Program 1.10 (p.27), except we return a bool.
- *Termination occurs?* Yes, since BinarySearch is called on successively shorter lists so that eventually each list will only have one member. If this member is not x, then `false` will be returned.
- *Termination condition correct?* Yes, if `x` is found, true is returned, otherwise `false` is.
- *Recursive condition correct?* Yes, if x is less or greater than the middle element, the result of searching the left or right sublists, respectively, is returned

- **Example 3: Palindrome Checker**
  - To check if *number* is a palindrom (number is a number of form 123321 or 12321)
- **Example 4: Tower if Hanoi**
  - *Problem*: Excercise 16 (p.29). n disk are placed on tower A and sre supposed to be moved on tower C, using B as auxilry tower. (p.29)
  - *Algorithm/program*:
    - If there is only one disk move disk from tower A to tower C directly.
    - If there are more then one disks then
    - *Move n*-1 disk from tower A to tower B using tower C as auxilry
    - move disk n from tower A to tower C directly
    - *Move n*-1 disk from tower B to tower C using tower A as auxilry
  - *Termination occurs?* Yes, since function is called on successively shorter number of disks so that eventually last call will only have one disk. This disk will be drectly moved from source to destination without recursive call.
- **Example 5: Euclidian algorithm**
  - *Problem*: Greatest common divisor of two integers a and b is GCD (a,b) not both of which are zero, is the largest positive integer that divides the botha and b. Euclidian algorithm for finding GCD is as follows
  - *Algorithm /program*:
    - If b = 0 then GCD (a, b) = a
    - Divide a by b to obtain the integer quotient and remainder r, so that a = b*q + r then GCD (a, b) = GCD (b, r).
    - Replace a by b and b by r then repeat this process again the last non zero remainder is GCD (a, b).
  - *Termination occurs?* As the remainder is decreasing, eventually a remainder of zero will result.
- **Example 6: Recursion to Iteration**:
  - Following is the example of tail recursion

Tail Recursion to Iteration

```
Factorial (n)
    If (n = 0 Or n = 1)
```

```
            Ret n;
        Ret n * Factorial (n-1)
```

```
    Factorial (n)
        While (n > 1)
            Fac = Fac * n
            n = n –1
        ret Fac
```

Body Recursion to Iteration

```
    DecimalToBinary (value, base)
        If (value < base)
            Display value
            Ret;
        DecimalToBinary (value/base, base)
            Display value % base;
```

```
    DecimalToBinary (value, base)
        While (value > base)
            Stack.Push(value%base)
            value = value / base
        While (! Stack.Empty() )
            Display Stack.Pop()
```

2.12. Lecture Notes 09 – Queue ADT and Applications

Today's Schedule

- Queue ADT
- Multiple Stacks and Queues
- Queue Applications

Queue ADT

- A *queue* can be viewed as a special kind of ordered list where insertions are constrained to happen at one end, the *rear*, and deletions are constrained to happen at the other end, the *front*. This results in a First-In-First-Out (FIFO) policy.
- Classical example: simple operating system job scheduling.
- Operations: insert, delete, check full/empty status.
- Representation: one-dimensional array with additional data members to store the front and rear indices (actually, *front* stores one less than the front element to simplify the empty status check) and the maximum size of the queue. *front* ≤ *rear*.
- We insert at the rear and delete from the front. The queue is empty when *front*==*rear*.
- Disadvantage of this representation: the queue tends to shift to the right. Try to shift it back every time we delete an element can be expensive in an array, O(*n*) in the worst-case where *n* is the size of the queue.
- Solution: Use a circular array. Rather than insert at *rear*+1, insert at (*rear*+1)%*MaxSize* if available. Rather than delete from *front*+1, delete from (*front*+1)%*MaxSize* if available.
- Note that to be able to distinguish between an empty queue and a full queue, we must use one element less than *MaxSize*. The queue is full when *front* equals *rear* incremented by 1 (circularly). We could get around this limitation by using an extra variable, but it makes the operations more expensive and usually isn't worth the effort.

Multiple Stacks and Queues

- Easy; to represent m stacks and n queues, just use an array of pointers to stack objects and an array of pointers to queue objects. Because stacks and queues are represented as lists, we don't run into the same space issues we faced when representing multiple ones.

Queue Applications

- Application of Queue data structure is mainly in Operating Systems Scheduling algorithms.

- Process management involves different queuing algorithms e.g. First Come First Server (FCFS) [Simple queue is used], Shortest Job First (SJF) [Ascending priority queue w.r.t. total time of execution of process], Shortest Remaining Time First SRTF [primitive SJF], Multiple Feed Back queue etc.
- See Operating Systems by Collin Ritchie.

2.13. Lecture Notes 10 – Singly Linear Linked List

Today's Schedule

- Singly Linear Linked Lists

Singly Linked Lists

- Singly Linked list is a linear, dynamic, and sequential access data structure. That is an alternative approach to array, i.e. linear, constant size, and random access data structure.
- As we've seen, many common ordered list operations such as insert and delete be expensive when the ordered list is implemented using sequential mapping, i.e., with an array. Storing multiple lists also proved to have time and space complexity drawbacks.
- Solution: linked lists. By storing a little more information with each item of the list – the memory address of the next item – we make the following gains:
  - The extra flexibility to store items anywhere in memory.
  - The ability to change the conceptual ordering of the list without needing to move its elements. In particular, insertions and deletions can take constant time.
  - Only space currently in use by the list is allocated.
- A linked list consists of a set of nodes, each containing data fields and link fields. Link fields point to other link nodes. (Note that a node may have more than one link fields. Examples include nodes in doubly linked lists and trees.)

  Example:

```
class Node {
private:
        int data;
        Node *link;
};
```

- Linked list design:
  - *Representation 1*: Global pointer to *first*. This is the approach in non-OOP languages. Problem: Cannot access data members of nodes because of encapsulation.
  - *Representation 2*: Make the data members of *Node* public. Problem: Violates encapsulation principle.
  - *Representation 3*: Add accessor functions to *Node* class. Problem: Gives global access to *Node*'s data members. We only want to give access to those using a linked list.
  - *Representation 4*: Create a linked list class that *conceptually* contains the whole list. This is implemented by giving the linked list class a private *Node* pointer *first* which points to the rest of the list. For the linked list class to have access to the data members of the nodes, we either make it a

friend of the *Node* class or nest the *Node* class in the linked list class definition. In the latter case, we make the *Node* class' members public so that the list can access them, but include the *Node* class definition in the private section of the list class so that other classes cannot access it. The downside is that we cannot use the *Node* class in any other class. For example, we may want to use the same *Node* structure in different kinds of linked lists, say, singly linked and circular.

- Insert and delete take constant time if we are given a pointer to the parent node. Otherwise, it can be very expensive to find the parent node to complete the operation.

2.14. Lecture Notes 11 – Reusable Linked List

Today's Schedule

- Reusable Linked List Class
- Polynomials

Reusable Linked List Class

- Most linked list operations don't depend on the kind of data in the nodes. To make our linked list design maximally reusable, we make the node and list classes template classes.
- Answer to question on making lists of multiple types: if you don't specify the type of the pointer to a template class, C++ won't recognize it when you try to instantiate it.
- Operators: Important design decision. Too many operators, and the class is too bulky; also, some operators only apply to select types. Too few operators, and the class is less useful. We should include the standard operations for a reusable class: default and copy constructors, destructor, assignment operator (**operator=**), equality test operator (**operator==**), input operator (**operator>>**), and output operator (**operator<<**). Useful general linked list-specific operations include: insertions and deletions of various kinds, list reversal, concatenation, etc.
- Iterators
  - Many operations on linked lists involve iterating through the list.
  - The problem is that our list definition only allows linked list operations to iterate since the link members are private. Thus, we would need to include all such operations as members of the linked list class. The downsides are that
    - There may be many such operations.
    - The class author cannot guess all the operations a future user might want, but allowing the user to modify the implementation to add operations as needed is contrary to the principle of encapsulation.
    - Some of these operations may not be meaningful for all possible instantiations of the class.
  - Since many of these iterative operators don't modify list members, we can safely define them outside of the class. Rather than make them all friends of the list and node classes, we extract the iteration operation into a separate class ListIterator and make it a friend of the these two classes. This class includes operations First, Next, NotNull, and NextNotNull.
- Let us print out the contents of an arbitrary list, but define the function outside of the class and use a list iterator to traverse the list.

```
template <class Type>
void printList(List<Type>& l)
{
```

```
        ListIterator<Type> it(l);
        if (it.NotNull()) cout << *it.First();
        while (it.NextNotNull()) cout << *it.Next();
}
```

- When not to use the reusable class:
  - When efficiency is important.
  - When the application requires specialized operations.

Polynomials

- We've looked at reimplementing specialized lists stacks and queues with linked lists rather than arrays. Let's now revisit one of the applications of lists: polynomials.
- Note textbook's definition of IMPLEMENTED-BY on p. 190.
- Rather than store and maintain an array of terms, we store and maintain a linked list of terms. Specifically, we define a simple term structure that holds a coefficient and an exponent, and define the polynomial class to have one private data member, a List of terms.

```
struct Term {
        int coef, exp;
        void init(int c, int e) {coef = c; exp = e;}
};

class Polynomial {
public:
        …
private:
        List<Term> poly;
};
```

- Once again, the problems of wasted space and expensive maintenance disappear.
- Let us reconsider the addition operation. The underlying algorithm is the same as in the case of arrays. We use two list iterators to stream along the two polynomials, comparing exponents and adding as we go. For each new term in the new polynomial, we create a new node, copying in the values of the term. As with arrays, the time complexity of addition is the optimal O(m+n).
- If we are dynamically working with many polynomials, the time spent deleting polynomials represented as linked lists becomes excessive. For such applications, it is better to use circular linked lists to represent polynomials and use our own available space list to handle space allocation. This makes polynomial deletion O(1) time.

2.15. Lecture Notes 12 – Circular and Doubly Linked List

Today's Schedule

- Circular Linked Lists
- (De) Allocating Linked Lists
- Doubly Linked Lists

Circular Linked Lists

- Last node points to first.
- Better to have pointer to last node than to first.
- For some applications, we introduce a dummy node to avoid treating the empty list as special case.

(De) Allocating Linked Lists

- If no list destructor is defined, list nodes never get freed. Deleting lists requires first deleting each node via the list destructor. Obviously, this takes O(n) time in general, where n is the length of the list.
- To save time when allocating new nodes, a good idea is to maintain our own free space stack *av* (for "available") which is a static linked list member of the list class. When we delete nodes, rather than free the space using delete, we add them to the *av* using *RetNode*() which we define. To allocate a new node, rather than use **new**, we use *GetNode*() which first checks to see whether there are any pre-allocated nodes on the *av* stack. If there are, it returns the top one; only if there aren't does it allocate new space.
- If lists are circular, using *av* also allows us to efficiently delete lists as well. We simply break the circle, attach one end to *av*, and point *av* to the other end.
- Of course, the downside of using *av* is other classes (or processes) cannot use that memory.

Doubly Linked Lists

- To allow us to easily move in either direction in a linked list, we extend nodes to have two link fields: *llink* (or *prev*) and *rlink* (or *next*).
- Recall that deleting a node from a list given only a pointer to the node was expensive because we needed to have access to the node's parent and finding the parent required traversing the list. Doubly linked lists solve this problem.
- Similarly, they solve the problem of needing to point to the last node in a circular list to efficiently access both the first and last node.
- Example function: Delete for a chain.

```
void DblList::Delete(DblListNode *x) {
    if (x == first) {
            first = first->rlink;
```

```
                first->llink = 0;
        } else if (x->rlink = 0) {
                x->llink->rlink = 0;
        } else {
                x->llink->rlink = x->rlink->llink;
                x->rlink->llink = x->llink->rlink;
        }
}
```

2.16. Lecture Notes 13 – Variants of Linked List

Today's Schedule

- Generalized Lists
- Heterogeneous Lists

Generalized Lists

- We have only considered lists so far whose elements are atoms. In general, the elements in a list may be either atoms or other lists. As a simple example, recall the powerset. We call such lists *generalized lists*. We refer to the list elements of a list as *sublists* and use capital letters to denote them.
- Examples: A = (a, (b, c)), B = (A, A, ()), C = (a, A, C), D = (a, E), and E = (b, D).
- Thus, generalized lists are recursively defined. Two important consequences:
    - The same list may be an sublist of multiple lists; e.g., A is a sublist of both B and C.
    - Lists may be recursive, i.e., a list may have itself as sublist either directly (e.g., C) or indirectly (e.g., D and E). Note that although we still require each list to have a finite number of elements, lists may be infinite may due to recursion (e.g. C).

  These create important implications for implementing generalized lists. We will address these later. For now, assume lists are not shared non-recursive.
- Terminology: ***head*** – first element of a list; ***tail*** – list of remaining elements.
- A concrete example of a generalized list: multivariate polynomials.
    - Sequential representation requires a different number of fields for a different number of variables. A linear list requires nodes to vary in size according to the number of variables. Both are inelegant.
    - Observation: Each polynomial can be represented as a list of coefficient-exponent pairs where each coefficient is itself a polynomial in one less variable.
    - E.g.: p. 222
- Other examples: lists in Lisp, arithmetic expressions.
- C++ representation:
    - We need nodes to be general enough to represent either an atom or another list, and we need to be able to distinguish between the two.
    - Solution: In addition to the link field pointing to the next element in the list, each node has two data fields, an atom and a link to a list, and a boolean field indicating which is being used.
    - We use **union** to enforce the restriction that only one gets used at a time, and to minimize the space used (**union** reserves only enough space to store the largest of its members).
    - **class** GenListNode {
      friend GenList;
      private:
      bool tag;

```
        union {
            char data;
            GenListNode *dlink;
        }
        GenListNode *link;
};

class GenList {
public:
        …

private:
        GenListNode *first;
};
```

- Operations
  - Usually recursive and made up of two components, the (private) *workhorse* which does the recursive work and the (public) *driver* which a user calls and which makes the first call to the workhorse.
  - Examples: copy, ==, depth, etc.
- Handling shared lists
  - Allowing shared lists can save a lot of space.
  - No changes need to be made to our implementation; lists that share a sublist simply point to the same sublist.
  - Problem: Adding and deleting nodes at the front of a sublist requires knowing which lists are pointing to the sublist so as to update their pointers.
  - Solution: Use headnodes. Allow tag field to hold one of three values corresponding to header node, atom element, or list element.
  - Problem: Don't know when a list is no longer pointed to by other lists and can be returned to the storage pool.
  - Solution: Store the list's *reference count*, that is, the number of lists referencing the list, in the data field of the header node. When a call is made to delete a list, this counter is simply decremented if it is greater than 0. When it becomes 0, the list can be safely deleted.
  - **class** GenListNode {
    …
    }
  - Erase algorithm
- Handling recursive lists
  - Problem: Reference count never goes to 0, so space is never freed.
  - There is no easy way to supplement our implementation to determine which recursive lists can truly be freed, so we will keep leaking memory.

Heterogeneous Lists

- When we first introduced linked lists, the question arose about how to represent lists that contain different types since even the reusable linked list class assumes all elements are of one type.
- We could use **union**, but this potentially wastes space since it reserves enough space to store the largest type. Furthermore, it is a bit artificial.
- Someone suggested using public inheritance…
- Solution:
  - Abstract class *Node* contains only the information shared by all node types: a non-virtual pointer *link* to another node and a pure virtual function *GetData* that returns the node's data via a *Data* structure which, in turn, uses **union** of type variables and a tag field indicating which variable to use.

```
struct Data { // example Data structure
        int id; // 0, 1, 2 if char, int, or float, respectively
        union {
                int i;
                char c;
                float f;
        };
};

class Node {
friend class List; friend class ListIterator;
protected:
        Node *link;
        virtual Data GetData()=0;
};
```

  - Template class *DerivedNode*<*Type*> is publicly derived from *Node* and contains the storage space for the data in variable *data*. For each data type, we implement *GetData* to instantiate the appropriate fields in a *Data* object to return.

```
template<class Type>
class DerivedNode : public Node {
friend class List; friend class ListIterator;
public:
        DerivedNode(Type item) : data(item) {link = 0;};

private:
        Type data;
        Data GetData();
};
```

```
Data DerivedNode<char>::GetData() {
        Data d;
        d.id = 0; d.c = data;
        return d;
}
…
```

- *List* class remains essentially unchanged. (Note that *Node* no longer takes a template argument.) *ListIterator* class also stays essentially unchanged, excecpt we must change *First* and *Next* to use *GetData* to access data and to return pointers to *Data* objects. See Program 4.45 on p. 244 of the textbook.
- Because the data is stored in the derived nodes, space is dynamically reserved for only the kind of data held by each particular node, so we don't waste any space.

2.17. Lecture Notes 14 – Pre Mid Review

Today's Schedule

- **Course Activities:** Lectures, Labs, Tutorials, Assignments and Projects, Quizzes, Case Studies.
- **Content Covered:** Software Development Issues, Tools, Abstract Data Types, Activities related to ADTs, Applications.

Activities:

- **Lectures:**

  Lecture 0A – Review of Programming History
  Lecture 0B – Review of Object Oriented Technology
  Lecture 01 – Overview of Data Structures
  Lecture 02 – Algorithms and Recursion
  Lecture 03 – Performance Analysis and Measurement
  Lecture 04 – Array and Polynomial ADTs
  Lecture 05 – Sparse Matrix and String ADT
  Lecture 06 – Bag and Stack ADT
  Lecture 07 – Expression Validation and Evaluation
  Lecture 08 – Recursion
  Lecture 09 – Queue ADT and Applications
  Lecture 10 – Singly Linear Linked List
  Lecture 11 – Reusable Linked List and Iterators
  Lecture 12 – Circular and Doubly Linked List
  Lecture 13 – Variants of Linked List
  Lecture 14 – Review

- **Labs:**

  Lab 01 – ADT Implementation
  Lab 02 – Performance Measurement
  Lab 03 – Implementation of Polynomial ADT
  Lab 04 – Implementation of Stack ADT
  Lab 05 – Implementation of Recursion and Queue ADT
  Lab 06 – Implementation of Linear Linked List ADT
  Lab 07 – Implementation of Circular and Doubly Linked List ADT
  Lab 08 – Midterm Practical

- **Tutorials:**

  Tutorial 01 – Fundaments of Programming Languages
  Tutorial 02 – Object Technology
  Tutorial 03 – Templates in C++

Tutorial 04 – Backtracking an Algorithm
Tutorial 05 – Queues in Operating System

- **Assignments and Projects:**

  Assignment 01 – ADTs (Bag, Set, Rational No, Complex No, String)
  Assignment 02 – Midterm Project – Simple Database Management System (string processing, linked list, file handling)

- **Quizzes:**

  Quiz 01 – ADTs, Arrays, Stacks, and Queues.
  Quiz 02 – PreMid Test – (ADTs, Arrays, Stacks, Queues, Linked List)

- **Case Studies:**

  Case Study 01 – Game of Life
  Case Study 02 – Stack Application: A Mazing Problem
  Case Study 03 – Free Cell Game
  Case Study 04 – C to Assembly Converter

Contents Covered:

- **Software Development Issues:**
  - Time complexity
  - Space complexity
  - Software maintenance
  - Algorithmic maintenance
  - Reusability
  - Encapsulation and
  - Abstraction

- **Tools:**
  - **Algorithmic:** recursion, searching, sorting
  - **Data Structures:** arrays, linked lists, iterators, STL
  - **Performance:** step tables, asymptotic notation, measurement
  - **Correctness proofs:** induction
  - **C++:** classes and templates

- **Abstract Data Types:**

    - Set and Bag
    - Complex and Rational Number,
    - Array and Polynomial,
    - String
    - Matrix, Sparse Matrix, and ASymmetricSparse Matrix,
    - Stack, DStack, and MultiStack,
    - Queue, DQueue, and MultiQueue,
    - Linked List,
    - Order List,
    - Liner, Circular, and Doubly Linked List,
    - Recursive List
    - Generalized List
    - Heterogeneous List

- **Activities related to ADTs:**

    - Definitions
    - Operations
    - Alternative representations of each ADT and corresponding operator implementations with associated tradeoffs.
    - Optimizations (e.g., queue to circular queues)
    - Extensions (e.g., compactly representing multiple structures)

- **Applications:**

    - Array: Polynomials
    - String: Strings Processing
    - Matrix: Sparse Matrices
    - Recursion: Tower of Hanoi
    - Stack: Expression Validation and Evaluation
    - Stack: Expression to Assembly Mnemonic generation
    - Queue: Process Management
    - Linked List: Data Base Management System

2.18. Lecture Notes 15 – Binary Tree

Today's Schedule

- Trees
- Binary Trees

Trees

- Definition: A *tree* is a finite set of nodes where one node is designated the *root* node and the remaining nodes are partitioned into sets and each set is a tree. Note that this is another example of a recursively defined ADT.
- Terminology: *node degree, tree degree, leaf, child, parent, sibling, ancestor, level, height / depth*
- Representations
    - List as a string. Problem: not easy to traverse or manipulate
    - Generalized lists. Problem: nodes too general; wastes space and not as easy to traverse.
    - Nodes with data field and tree-degree fields for children. Problem: wastes space and is difficult to extend to accommodate a tree of higher degree.
    - Nodes storing data, leftmost child, and sibling to the right. This defines a binary tree where the left child is the leftmost child and the right child is the sibling to the right. Problem: same traversal problems as generalized lists.
- Thus, there is a trade-off between space and flexibility on the one hand, and efficiency on the other hand.

Binary Trees

- Definition: A *binary tree* is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees, the left and right subtrees.
- Distinction from trees: May be empty, restricted to have only two children, and the children are distinguished.
- Terminology: *full*, *complete.*

2.19. Lecture Notes 16 – Binary Trees (cont.)

Announcements

- Read section 5.4.3 of the textbook (the satisfiability problem) on your own. It describes an application of trees.

Today's Schedule

- Binary Trees (cont.)
- Binary Tree Operations

Binary Trees (cont.)

- Properties: Let n be the total number of nodes in a tree and n_0, n_1, and n_2 are the number of nodes of degree 0, 1, and 2, respectively.
  - Maximum number of nodes on level *i*: 2^{i-1}
  - Maximum number of nodes in tree of depth *k*: 2^k-1
  - $n_0 = n_2 + 1$
  - Height if tree is complete: $\lceil \log_2(n+1) \rceil$
- Operations: insert/delete node, isEmpty, parent/left child/ right child of a node, traversal operations (inorder, preorder, postorder, level-order), copy, equality, etc.
- Array representation
  - Store nodes in slots corresponding to their level-order numbering in a full tree. Technicality: So as to differentiate between occupied and unoccupied cells, store pointers to data objects in the array rather than the objects themselves.
  - Advantage: Easy to find parent and children of a node. For node i, parent is at $\lfloor i/2 \rfloor$, left child is at 2i, and right child is at 2i+1. Traversals can also be done efficiently.
  - Disadvantages: Wastes a lot of space if trees aren't almost full. Add and delete are easy for particular leaf nodes, but more general additions and deletions may require the movement of many elements.
- Linked list representation
  - ```
    class TreeNode {
    friend class Tree;
    private:
            char data;
            TreeNode *leftChild, *rightChild;
    };

    class Tree {
    public:
            …

    private:
    ```
    ```

```
        TreeNode *root;
};
```

- Advantages: Insertion, deletion, and finding node children easy. Also, no wasted space.
- Disadvantage: Expensive to determine node parent. We can add parent field, but this adds O(n) space and is often unnecessary.
- Note: 2n link fields (2 per node), n+1 of which are 0.

Binary Tree Operations

- Inorder, preorder, postorder traversal implementation
  - Recursive implementation: uses system stack.
  - Non-recursive implementation: uses our own stack. Iterators use (and, in fact, motivate) this implementation.
  - "Stack-free" implementations: Both the recursive and nonrecursive versions use a stack that requires O(n) extra space. We want to avoid this extra overhead. Possible solutions:
    - Use parent field: requires extra pointer field per node, so still O(n) extra space.
    - Use threaded trees (coming soon): requires two extra bits per node, so still O(n) extra space, but with a smaller hidden constant.
    - Use threaded trees, but use public inheritance to derive nodes of different degrees so that it is unnecessary to store the extra bits. Problem: C++ uses extra space to label the type of each node. Also, inserts and deletes are complicated by the fact that nodes change from one type to another.
    - Change children pointers of nodes to point to parents as we traverse the tree, then change them back once we are done; take advantage of n+1 unused leaf node fields, using them to maintain a stack of nodes where we went right: requires only O(1) extra space, no extra time. Downside: Algorithm is complicated and opaque (hence, difficult to maintain).
- Level-order traversal implementation: uses a queue.
- Other operations such as copy, equality, etc. are easily implemented using recursion.

2.20. Lecture Notes 17 – Threaded Binary Trees

Announcements

- Skip sections 5.7.5, 5.8-5.12

Today's Schedule

- Threaded Binary Trees

Threaded Trees

- Motivation: The traversal implementations we have studied require a stack. However, since there are more 0-links than actual pointers (n+1 of the 2n links), why not use this space to store the stack rather than allocate extra space for a stack?
- We define threaded trees as follows:
  - For convenience, we assume the tree has a head node with the actual tree as its left subtree and itself as its right child. This helps to simplify the algorithms.
  - We make the following modifications to tree nodes:
    - The left field of a node with no left child contains a *thread*, that is, a link, to its predecessor according to the infix ordering (to the head if it is the first node).
    - The right field of a node with no right child contains a link to its infix successor (to the head if it is the last node).
    - Each node contains two boolean fields *LeftThread* and *RightThread* indicating whether the respective link fields are being used as threads or regular tree links.

  We call these modified nodes threaded tree nodes.
- Note: This representation still requires O(n) extra space for the boolean fields, but the hidden constant is potentially much smaller than if we use a stack.
- Traversal: Inorder traversal is simple, efficient, and can be done without an extra stack.
  - The approach described in the textbook is to implement an iterator *ThreadedInorderIterator* and simply apply its member function *Next*() repeatedly to traverse the tree. The iterator interface looks as follows:

```
template <class T>
class ThreadedInorderIterator {
public:
        ThreadedInorderIterator(ThreadedTree<T>& tree) : t(tree)
        {current = t.root;};

        T* Next();
        …
```

```
private:
        ThreadedTree<T> t;
        ThreadedTreeNode<T>* current;
};
```

The constructor takes a tree argument *t* that is to be traversed. *current* points to the current node in a given traversal. To find the node in *t* that follows *current* in the infix ordering of nodes, *Next*() must consider two cases:

- Case 1: *current*'s right link is a thread (i.e., *RightThread* is true). Then, by construction, this link points to the next node.
- Case 2: *current*'s right link is not a thread. Then this link points to a right child. To find *current*'s successor, move right, then repeatedly move left until it is no longer possible to do so.

The following code does just this:

```
template <class T>
T* ThreadedInorderIterator<T>::Next() {
        if (current->RightThread) current = current->RightChild;
        else {
                current = current->RightChild;
                while (!current->LeftThread) current = current->LeftChild;
        }
        if (current == t.root) return 0;
        else return &(current->data);
}
```

(This algorithm is equivalent to Program 5.14 on p. 281 of the textbook.)

The inorder traversal algorithm is then very simple. For example, if we want to print out the data stored in the tree in infix order, we do the following:

```
template <class T>
void InorderPrint(ThreadedTree<T>& t) {
        ThreadedInorderIterator<T> it(t);
        for (T *node = it.Next(); node; node = it.Next())
                cout << *node << endl;
}
```

Note that, because we have a head node, we don't need to have a First() to get the first node, and we don't need a special check to see if the tree is empty.

- A second approach (which we considered in class) is, perhaps, not quite as easy to follow. Each iteration in the algorithm considers the next node

visited rather than the next node to operate on. The algorithm applies simple local checks to determine whether to operate on it and which link to follow. The algorithm proceeds as follows:

- Move left from the root.
- For each node, consider three:
  - Case 1: Node is the root; we are done.
  - Case 2: Node was arrived at via a normal child link. Then we need to keep moving left down the tree if possible to find the next node to operate on. If the node's left link is a thread, it is no longer possible to move left down the tree; instead, operate on the node (e.g., print out its data) and move right.
  - Case 3: Node was arrived at via a thread, so it is the next node to operate on. Once done, move right.

The following code implements this algorithm for a traversal that prints out the data stored in a tree:

```
template <class T>
void InorderPrint(ThreadedTree<T>& t) {
        ThreadedTreeNode<T>* node = t.root->LeftChild;
        bool viaThread = t.root->LeftThread;
        while (node != t.root) {
                if (!viaThread) { // arrived at via child link
                        if (!node->LeftThread) { // has left child; move left
                                viaThread = false;
                                node = node->LeftChild;
                        } else { // can't move left any further.
                                cout << node->data << endl;
                                viaThread = node->RightThread;
                                node = node->RightChild;
                        }
                } else { // arrived at via thread
                        cout << node->data << endl;
                        viaThread = node->RightThread;
                        node = node->RightChild;
                }
        }
}
```

- Both algorithms take O (n) time. In both cases, we visit nodes with left children twice, other nodes once.
- Note that we don't use the left threads at all. These could be used if we wanted to efficiently traverse the tree backwards.

- Insertion: Consider inserting a node r as the right child of a node s.
  - There are two cases to consider:

- Case 1: s's right subtree is empty, so s's right link is a thread to s's successor. Inserting r will make s's successor r's successor, so set r's right link to s's right link and set r's *LeftThread* to true.

Note that a predecessor thread pointing to a node always originates from a descendant in the right subtree. Since s had no right subtree and r has no right subtree, no predecessor threads (besides the one from r to s – see below) need to be updated.

- Case 2: s's right subtree is not empty, so s has a right child. We make s's right child r's right child and set r's *RightThread* to false. Now, since s had a right subtree, its original successor must have a left thread pointing to it. This successor will now be r's successor, so we must find the successor and point its thread to r instead.

In either case, r becomes s's right child (and we set s's *RightThread* to false) and r's left link becomes a thread back to s (and we set r's *LeftThread* to true). Also note that in both cases r's right link gets set to s's right link and r's right thread switch gets set to s's right thread switch. Program 5.16 on p. 283 of the textbook implements this algorithm.

2.21. Lecture Notes 18 – Priority Queue ADT

Today's Schedule

- Priority Queue ADT
- Heap ADT

Priority Queue ADT

- **Definition:** A queue where elements are removed, not in order of arrival, but in order of priority as specified in a key data member, where elements are

```
template <class T>
struct Element<T> {
        int key;
        T data;
};
```

- In max priority queues, elements are removed in decreasing order of key value. In min priority queues, they are removed in increasing order of key value. We will focus on max priority queues since min priority queues are similar.
- **Example:** insert 2, 10, 5, 6, 35, 4 -> removed in 5^{th}, 2^{nd}, 4^{th}, 3^{rd}, 7^{th}, 1^{st} order of insertion
- **Operations:** insert, delete
- **Interface:** we make it abstract since we can implement it using different data structures.

```
template <class T>
class MaxPQ {
public:
        virtual insert(const Element<T>& e) = 0;
        virtual Element<T>* deleteMax(Element<T>& e) = 0;
};
```

- **Representations**
    - unordered linear list (sequential – i.e., array – representaion):
        - insert – add at end; Θ(1) time
        - delete – search for largest; Θ(n) time
    - unordered linear list (linked list representaion):
        - insert – add at front; Θ(1) time
        - delete – search for largest; Θ(n) time
    - ordered list (sequential representation):
        - insert – find correct place, shift remaining elements, and insert; O(n) time for shifting

        - delete – delete from end; Θ(1) time
    - ordered list (linked list representation):
        - insert – find correct place and insert; O(n) time for search
        - delete – delete from front; Θ(1) time
- Suppose m is the combined number of insertions and deletions in a sequence. In all of the above representations, the sequence of operations will take O (m n) time if the number of insertions and deletions are proportionally related (e.g., evenly split). It turns out that we can do much better. We will study next a tree-based representation where both insertions and deletions take Θ(log n) so that a sequence of these operations will takes only Θ(m log n).

Heap ADT

- A general approach to data structure design is to find a way to add easily maintainable structure (i.e., invariants) to a standard data structure such that desired operations are efficient. We will study two different ways of adding structure to trees – corresponding to heaps and binary search trees – to efficiently solve different problems.
- Heap motivation: priority queues
- Definition: *Max tree* – children keys are no larger than parent key. *Max heap*– complete max tree. For priority queues, keys correspond to priorities.

2.22. Lecture Notes 19 – Heap ADT

Today's Schedule

- Heap ADT (cont.)
- Binary Search Tree ADT

Heap ADT (cont.)

- **Operations:** insertion, delete, isEmpty. Note: operations need to maintain two properties, the max tree property and the completeness property
- **Representation:** Tree is complete, so an array *heap* is a reasonable choice; *heap*[0] is not used. Alternatively, we may used a linked representation, some operations are no longer as efficient.
- Insertion
    - *Operation*: Insert node in next available leaf position that maintains completeness property. Θ(1) time if we use sequential representation; potentially Θ(log n) time to find location if we use a linked list representation.
    - *Maintenance*: Move node up tree (iteratively switching positions with parents) until we find correct location for new element, i.e., the max tree property is restored. Θ(log n) time for either sequential or linked representation.
    - Example
- Deletion
    - *Operation*: Remove root element. Θ(1) time for either representation.
    - *Maintenance*: Move hole down tree, iteratively shifting up child with max key until last element (according to level-ordering) fits in hole according to max tree property. Θ(1) time to find last element and Θ(log n) time to shift for sequential representation; Θ(log n) time to both find last element and to shift for linked list representation.
    - Example

Binary Search Tree ADT

- **Motivation:** We want to be able to delete elements with arbitrary keys efficiently.
- **Definition:** *Binary Search Tree (BST)* – A tree where every element has a unique key, the keys in the left and right subtree are smaller and larger, respectively, than the key in the root, and the sub trees are also BSTs.
- **Operations:** insertion, deletion, search (by key or rank)

2.23. Lecture Notes 20 – BST ADT

Today's Schedule

- BST ADT (cont.)

BST ADT (cont.)

- Unlike in previous data structures, these operations can fail. We will see that the time complexity is different for successful and unsuccessful operations.
- Search:
    - Two kinds: by key and by rank
    - For key search, compare input against key of each node; if equal, terminate with success; if < or >, go left or right, respectively; if not possible to move, terminate with failure.
    - For rank search, assume nodes also have a ***LeftSize*** data member that stores the number of nodes in the node's left subtree plus 1. Move down tree until reach node where the number of nodes above and to the left of the node plus itself equals the desired rank.
    - Example
    - O(h) time, both by key and by rank
- Insertion:
    - Algorithm
        - *Operation*:
            - search for key (O(h) time)
            - If found, insert failed (can't have duplicate keys in BST); if not found, insert where search ended (O (1) time)
        - *Maintenance*: none required; insertion always at bottom of tree in a location where BST properties are satisfied
    - O (h) time total
    - Example
- Deletion:
    - Algorithm:
        - *Operation*: delete node (O (1) time)
        - *Maintenance*: 3 cases
            - Node is a leaf: no maintenance required (O (1) time)
            - Node has one subtree: move subtree up to replace node (O (1) time)
            - Node has two subtrees: find either the largest node in the left subtree or the smallest node in the right subtree; copy it into the space left by the node; delete it from its original location using one of the above two cases (O (h) time to find node)
    - O (h) time total

- Example
- Height: h and, thus, the time complexity of the algorithms, is O (n) in the worst-case, O (log n) on average. There are search trees with guaranteed O (log n) height, e.g., AVL trees and red/black trees. These are called *balanced* search trees.

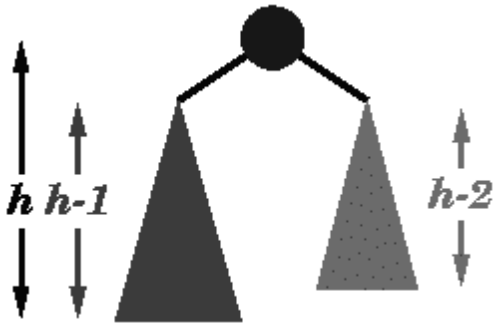
2.24. Lecture Notes 21 – AVL Trees

Today's Schedule

- AVL ADT

AVL Tree ADT

- **Motivation:** We want to have a search tree that guarantee O (log n) height, e.g., AVL trees and red/black trees. These are called *height balanced* search trees.
- **Definition:** An AVL tree is a binary search tree which has the following properties:
    - The sub-trees of every node differ in height by at most one.
    - Every sub-tree is an AVL tree.



- The balance factor of a node is $h_L - h_R$ where h_L is the height of its left subtree and h_R is the height of its right subtree. As soon as a node's balance factor becomes +2 or –2, it is rebalanced and balance factor becomes 0.
- AVL is a BST in each node has balance factor –1, 0 or 1.
- **Operations:** insertion, deletion, search (by key or rank)
- Unlike in previous data structures, these operations can fail. We will see that the time complexity is different for successful and unsuccessful operations.
- Search:
    - Two kinds: by key and by rank
    - For key search, compare input against key of each node; if equal, terminate with success; if < or >, go left or right, respectively; if not possible to move, terminate with failure.
    - For rank search, assume nodes also have a *LeftSize* data member that stores the number of nodes in the node's left subtree plus 1. Move down tree until reach node where the number of nodes above and to the left of the node plus itself equals the desired rank.
    - Example
    - O (h) time, both by key and by rank
- Insertion:
    - Insertion is somewhat complex and involves a number of cases. Implementations of AVL tree insertion may rely on adding an

extra attribute, the **balance factor** to each node. This factor indicates whether the tree is *left-heavy* (the height of the left sub-tree is 1 greater than the right sub-tree), *balanced* (both sub-trees are the same height) or *right-heavy* (the height of the right sub-tree is 1 greater than the left sub-tree). If the balance would be destroyed by an insertion, a rotation is performed to correct the balance.

- We can maintain an AVL tree by making some changes to the algorithm for binary tree insertion. We can visualize the change as retracing the path we took to do the insertion and checking to see whether the insertion caused a node to become unbalanced. When we return from an insertion, we set a boolean flag to true or false indicating whether the tree below has increased in height. If we're returning from a left subtree and the tree below has increased in height, there are 3 situations:
- Left subtree height has increased:
  - If current balance is +1, its balance will become +2 (critical), so we need to rebalance (either LL or LR situation). The current node's new balance factor will be 0 and we will return to its parent with a value of false for height increase.
  - If current balance is 0, its balance will become +1 (unbalanced but not critical), so we return to parent with a height increase value of true.
  - If current balance is –1, its balance will become 0, so we return to parent with a height increase value of false.
- We can come up with analogous situations for a return from a right subtree.
- Algorithm
  - *Operation*:
    - search for key (O(h) time)
    - If found, insert failed (can't have duplicate keys in AVL); if not found, insert where search ended (O (1) time)
  - *Maintenance*: none required; insertion always at bottom of tree calculate the balance factor of the tree if unbalanced the rotate the nodes to make them height balanced

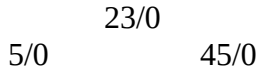
- **Example 1**: each node shows its key over its balance factor

45/2

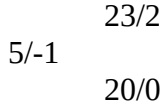
23/1

15/0

This is considered a LL situation. The first unbalanced node (45/2) is left heavy (+2) and so is its left child (+1). Need to rebalance. The result of a right rotation would be



- **Example 2**:



This is a LR situation. Unbalanced node is left-heavy (+2), and its left child is right heavy (-1). Result of first a left and then a right rotation would be:

After left rotation of bottom 2 nodes:

After right rotation:

There are 2 mirror-image situations: RR and RL.

For RR: Parent balance factor (?), right child balance factor (?)
For RL: Parent balance factor (?), right child balance factor (?)

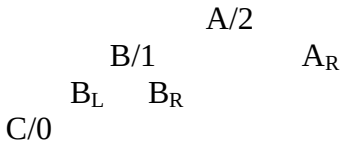
2.25. Lecture Notes 22 – AVL Trees (cont.)

Today's Schedule

- AVL ADT (cont.)

AVL Tree ADT

- We discussed a very simple case with just 3 nodes in pervious lecture. In actual fact there may be many nodes involved in the rotation. But there are only 4 canonical forms. The LL is shown first:

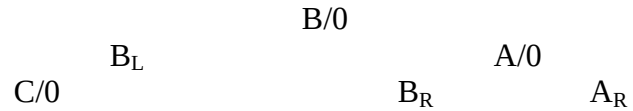


B_L , B_R , A_R are subtrees of height n (n could be 0) Node C was just inserted into B_L, the left subtree of node B, causing the tree to become unbalanced.

From properties of binary search trees, we know the following:

$B_L < B < B_R < A < A_R$

This requires a right rotation. B will become the root. Its right subtree will be A. A will no longer have B as its right subtree but will have B_L instead. B and A have balance factors of 0. The other nodes balance factors remain the same as before.



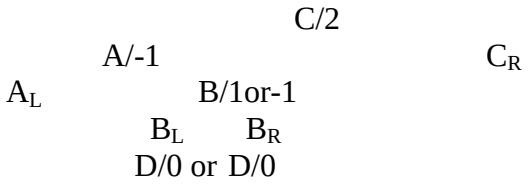
$B_L < B < B_R < A < A_R$

How do we accomplish this rotation:

Go back to original tree. p points to node A. Make the following link changes:

```
q = p -> left;          // q points to node B
p.left = q -> right;   // B_R becomes left subtree of A.
q -> right = p;         // A becomes right subtree of B
q -> bal = 0;
p = q;                  // p points to node B
p -> bal = 0;
```

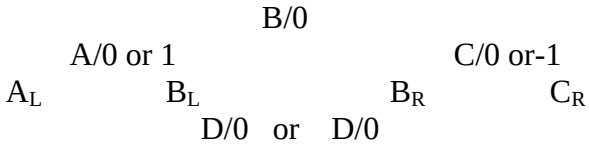
- Canonical LR situation:



A_L and C_R have n+1 nodes; B_L and B_R have n nodes. Also the situation when all subtrees are empty and we just have nodes C, A, and B.

$A_L < A < B_L < B < B_R < C < C_R$

A left rotation (around A) and then a right rotation (around C) does the following:



$A_L < A < B_L < B < B_R < C < C_R$

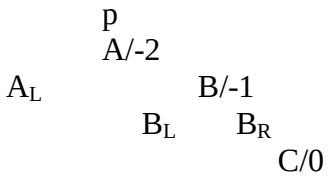
Consider it as rotating B left up to where A was, and then rotating B right up to where C was (Left, right rotation)

- Link changes:

Let p point to node C
q = p -> left; // q points to A
r = q -> right; // r points to B
q -> right = r -> left; // // B_L is left subtree of A is right subtree of A
p -> left = r -> right; // B_R is left subtree of C
r -> left = q; // A is left subtree of B
r -> right = p; // C is right subtree of B
if (r -> bal == 1) // inserted D in B_L
  q -> bal = 0; // A bal is 0
else
  q -> bal = 1;
if (r -> bal == -1) // inserted D in B_R
  p -> bal = 0; // C bal is 0
else
  p -> bal = -1;
p = r;
p -> bal = 0;

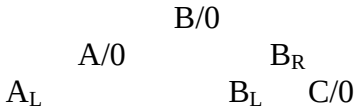
Note: if r -> bal = 0 (the case when all subtrees are empty), the balance factor for nodes A, B, and C will also be 0.

- We can maintain an AVL tree by making some changes to the algorithm for binary tree insertion. We can visualize the change as retracing the path we took to do the insertion and checking to see whether the insertion caused a node to become unbalanced. When we return from an insertion, we set a boolean flag to true or false indicating whether the tree below has increased in height. If we're returning from a left subtree and the tree below has increased in height, there are 3 situations:
- Left subtree height has increased:
    - If current balance is +1, its balance will become +2 (critical), so we need to rebalance (either LL or LR situation). The current node's new balance factor will be 0 and we will return to its parent with a value of false for height increase.
    - If current balance is 0, its balance will become +1 (unbalanced but not critical), so we return to parent with a height increase value of true.
    - If current balance is –1, its balance will become 0, so we return to parent with a height increase value of false.
- We can come up with analogous situations for a return from a right subtree. AVL Trees – 4 canonical forms – discussed 2 of them. The others are RR and RL
- RR form: both node p and its child are right heavy



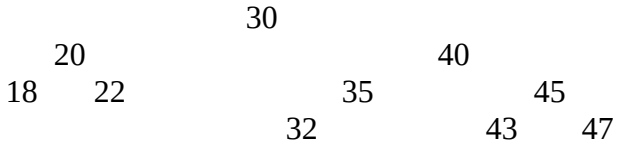
$A_L < A < B_L < B < B_R$

Rotate left to fix:



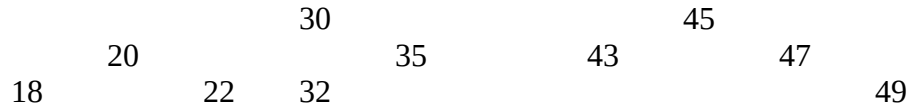
$A_L < A < B_L < B < B_R$

- Example:

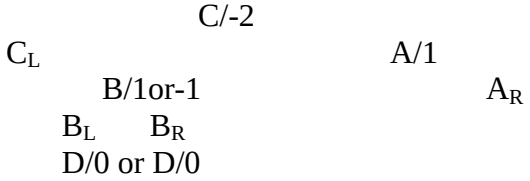


At this point, we can insert any number less than 40 without causing tree to be unbalanced. Consider inserting 44. Tree is unbalanced.

Correction: (rotate left)



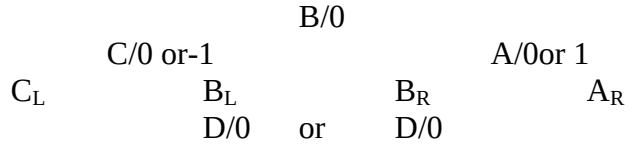
RL form:



$C_L < C < B_L < B < B_R < A < A_R$

Fix: Do a right and then a left rotation:

Rotate A to right, placing B where A was, then rotate C to the left, placing B where C was.



$C_L < C < B_L < B < B_R < A < A_R$

2.26. Lecture Notes 23 – Graph

Today's Schedule

- Graph ADT

Graph ADT

- Graph applications are ubiquitous: path-finding, circuits, task planning, and much more.
- Definition: A *graph* G is a pair (V, E) where V is a finite, non-empty set of *vertices* and E is a set of pairs of vertices from V called *edges*. Note: self-edges are possible; the book calls such graphs *graphs with self-edges*.
- Terminology: undirected graph, directed graph, head and tail (for directed graphs), subgraph, adjacent, degree, in-/out-degree (for directed graphs)
- If G = (V, E) is a graph, let n = |V| and e = |E|, the number of nodes and edges in G, respectively.
- More terminology:
  - *complete* – G has maximum number of edges
    - $e_{max} = n^2$ for directed graphs (n(n-1) if self-edges are disallowed)
    - e_{max} = n(n+1)/2 for undirected graphs (n(n-1)/2 if self-edges are disallowed).
  - *path* from u to v – sequence of vertices {u, w_1, … , w_k, v} such that (u, w_1), …, (w_k, v) are in E for undirected graphs, and similarly for directed graphs. We often call it a *directed path* if G is directed.
    - *length* – number of (not-necessarily unique) edges on path; equivalently, length of vertex sequence – 1.
    - *simple path* – path where all vertices, except, possibly, the first and last, are distinct. We often call it a *simple directed path* if G is directed.
    - *cycle* – simple path where the first and last vertices are the same. We often call it a *directed cycle* if G is directed.
    - Thus, if (u, v) is an edge in undirected graph G, then both {u, v} and {u, v, u} are simple paths using only that edge. However, the latter is a cycle. Note that the only simple path that includes a self-edge (u, u) is the cycle {u, u}.
  - *connected pair* (u, v) (<u, v>) – there exists a path from u to v.
  - *connected (undirected) graph G* – there exists a path between every pair of *distinct* vertices in the G. (Thus, a graph can be connected whether or not it has self-loops.)
  - *connected (directed) graph G* – there exists a path between every pair of distinct vertices in the underlying undirected graph G' implied by G (i.e., <u, v> is an edge in G iff (u, v) (and, thus, (v, u)) is an edge in G'). (This term is not commonly used.)
  - *strongly connected (directed) graph G* – there exists a directed path from u to v and v to u for every pair of distinct vertices u and v.

  - *component* – a maximal connected subgraph of G.
  - *tree* – a connected, acyclic graph. Thus, connected directed graphs where children have multiple parents are considered trees as long as there are no directed cycles.
  - *weighted graph* (aka a *network*) – graph where each edge is associated with a weight.
- Operations: insertVertex, insertEdge, deleteVertex, deleteEdge, isEmpty, adjacentVertices, numVertices, numEdges, degree (of vertex for undirected graph), in-/out-degree (for directed graphs), traversal

2.27. Lecture Notes 24 – Graph Operations

Today's Schedule

- Graph ADT (cont.)

Graph ADT (cont.)

- Graph operations:
  - Depth-First Search (DFS):
    - Definition: Operate on a node. Then operate on its first (unvisited) child, then that child's first (unvisited) child, etc., until it is no longer possible, then backtrack to the first ancestor with an unvisited child and repeat.
    - Similar to pre-order traversal of trees.
    - Example
    - Algorithm: Use recursion or explicit stack.
    - Complexity: $\Theta(n)$ time, $O(n)$ worst-case space for stack (e.g., when the graph is one chain of n nodes and the start node is at one end).
  - Breadth-First Search (BFS):
    - Definition: Operate on a node, then all its (unvisited) children, then all its (unvisited) grandchildren, etc.
    - Similar to level-order traversal of trees.
    - Example
    - Algorithm: Use queue.
      - Add first node to queue and mark as visited.
      - While queue is not empty:
        - Remove a node and operate on it.
        - Add node's unvisited children to queue and mark them as visited.
      - Note that when a node is marked as visited and when it is operated on happen at two different times – when it is enqueued and dequeued, respectively. If it we waited until a node's dequeuing to mark it, we would run the risk of putting multiple copies of it on the queue.
    - Complexity: $\Theta(n)$ time, $O(n)$ worst-case space for stack (e.g., when the start node has all n-1 other nodes as neighbors).
  - Spanning trees
    - Definition: A spanning tree is a subgraph $G' = (V,E')$ of a connected graph $G = (V,E)$ such that $E' \subseteq E$ and G forms a tree rooted at a designated vertex in V. Intuitively, G' is a minimal connected subgraph of G that includes all the nodes in G.
    - Minimum-cost spanning trees: A spanning tree of a weighted graph such that the sum of the weights is minimal.
    - Application: communication networks – e.g., minimizing the cost of connecting a set of cities.

- Shortest path
  - Assume we are given a weighted graph.
  - There are multiple path-related questions one might ask: Does a path exist between vertices v and w? What is the shortest path between v and w? What are the shortest paths between v and every other element in the graph? What are the shortest paths between every pair in the graph? We will consider the second to last problem for directed graphs.
  - The standard algorithm to solve this problem is attributed to Dijkstra. It works as follows:
    - Suppose S is the set of nodes – initialized to {v} – whose shortest paths from v have already been determined, let dist(w) be the length of the shortest path from v to w, and let weight(w, u) be the weight of edge <w, u>, ∞ if no edge exists. (In practice, we choose some number larger than the weights in the graph to represent ∞, but we must be sure that, when added to the longest shortest path, it doesn't cause an overflow.)
    - We iteratively add nodes to S in nondecreasing order of shortest path length. To do so, it suffices to find a node u not in S with a neighbor w in S such that dist(w) + length(w, u) is minimal for all nodes not in S with neighbors in S. The shortest path from v to u is the path from v to w to u, and its length is dist(w) + length(w, u) as calculated above.
  - Complexity: $O(n^2)$ since we must visit each edge at least once.

2.28. Lecture Notes 25 – Graph Operations (cont.)

Today's Schedule

- Spanning trees
- Shortest path
- Sorting Motivation

Spanning Trees

- Definition: A spanning tree is a subgraph G' = (V,E') of a connected graph G = (V,E) such that E' $\subseteq$ E and G forms a tree rooted at a designated vertex in V. Intuitively, G' is a minimal connected subgraph of G that includes all the nodes in G.
- Minimum-cost spanning trees: A spanning tree of a weighted graph such that the sum of the weights is minimal.
- Application: communication networks – e.g., minimizing the cost of connecting a set of cities.

Shortest Path

- Assume we are given a weighted graph.
- There are multiple path-related questions one might ask: Does a path exist between vertices v and w? What is the shortest path between v and w? What are the shortest paths between v and every other element in the graph? What are the shortest paths between every pair in the graph? We will consider the second to last problem for directed graphs.
- The standard algorithm to solve this problem is attributed to Dijkstra. It works as follows:
    - Suppose S is the set of nodes – initialized to {v} – whose shortest paths from v have already been determined, let dist(w) be the length of the shortest path from v to w, and let weight(w, u) be the weight of edge <w, u>, ∞ if no edge exists. (In practice, we choose some number larger than the weights in the graph to represent ∞, but we must be sure that, when added to the longest shortest path, it doesn't cause an overflow.)
    - We iteratively add nodes to S in nondecreasing order of shortest path length. To do so, it suffices to find a node u not in S with a neighbor w in S such that dist(w) + length(w, u) is minimal for all nodes not in S with neighbors in S. The shortest path from v to u is the path from v to w to u, and its length is dist(w) + length(w, u) as calculated above.
- Complexity: $O(n^2)$ since we must visit each edge at least once.

2.29. Lecture Notes 26 – Hashing

Today's Schedule

- Hashing
- Hash Functions
- Overflow Handling
- Dynamic Hashing

Hashing

- We now study a different implementation approach for which these operations are much more efficient in practice. This approach is called ***hashing*** and is probably one of the most important techniques you should take with you into the real world – if nothing else, for passing interviews.
- As we will see, hashing, when done well, can provide essentially O (1) time search, insert, and delete operations on average.
- Idea: Store elements in a table called a ***hash table***. A hash table is made up of a set of b buckets, each having s slots. We'll assume s=1 for now. Identifiers are mapped to a particular bucket using some arithmetic function h called a ***hash function***.
- For example, if we view the solution to the dictionary problem of assignment 3 as a hash table, then the hash function mapping an identifier to the ith bucket, where i is the alphabetical ordering of the identifier's first letter. (We will see that this is not a very good hash function.)
- Thus, hashing is unlike the techniques we have studied so far in that it is not based on comparisons between identifiers.
- Because the number of identifiers is usually much larger than the b we can afford, multiple identifiers of necessity map to the same bucket. Such identifiers are called ***synonyms***. When an identifier maps to a bucket already containing an element, we say a ***collision*** has occurred. When this happens to a full bucket, we say an ***overflow*** has occurred.
- When no overflow occurs, the time to search for an element depends only on the time to compute the hash function and the time to search the indexed bucket's s slots. There are a number of good O(1) time candidates for the former, and the latter is O(s) which is also fast since s is usually small and independent of n.
- The trick is to find hash functions that are fast and minimize collisions, and to find a good way to handle overflows.

Hash Functions

- Division
  - $h(x) = x \% M$, where $M \geq b$
  - Choose M to be a prime number to avoid problems of identifiers with the same suffix or identifiers that are permutations of each other from

colliding. (In practice, it is sufficient to choose M to be some number that has no prime divisors less than 20.)
    - This function is one of the most commonly used because it works very well for general-purpose applications.
- Folding:
    - Identifier is partitioned into several parts, all but last being of same length
    - Partitions are then added together
        - Shift folding (all partitions are added together that will be the resulting index)
        - Folding at boundaries (digits in alternative partitions are swapped mutually and then added together) see example 8.2 p. 471.
- Other functions described in the textbook include mid-square, and digit analysis.

Overflow Handling:

- Open addressing
    - When bucket is full, put element in next empty bucket.
    - To search for an element, start with hashed bucket, then linearly search buckets until element is found, empty bucket is reached, or search returns to original bucket.
    - Problem: identifiers tend to cluster together and clusters tend to combine, increasing search time.
    - Average number of comparisons is about $(2-\alpha)/(2-2\alpha)$ which is fairly small, but the worst case is very bad (size of the table).
    - Some solutions: quadratic probing, rehashing using multiple hash functions, and random probing.
- Chaining
    - Maintain a linked list of all elements hashing to bucket.
    - Search is only over list of elements in bucket.
    - Requires more space for link nodes, but smaller headnodes, so empty buckets take less space.
    - In practice, performs much better than linear open addressing.
- **Complexity:** Let $\alpha = n/b$ be the loading density. If we use a uniform hashing function *h* together with chaining, then the expected number of comparisons when searching for an element is $1+\alpha/2$ if the element is in the table, α if it is not.

Dynamic Hashing

- **Motivation:** The hashing schemes we have discussed are not ideal for handling very large databases because they require statically allocating a large portion of memory to hold the hash table, thus, suffering from the problems we have encountered before with arrays.
- Dynamic hashing represents a set of techniques for dynamically growing the hash table on an as-needed basis while retaining the fast access time of hashing.

Sorting

- Motivation:
  - A sorted list is often much easier to manipulate than one that isn't sorted.
  - For instance, a number of standard search algorithms are more efficient when the elements are already sorted, e.g., O (log n) for binary search on an ordered set vs. O (n) for sequential search on an unordered set.
  - Another example: comparing two lists is much faster if they are ordered, O (max {n log n, m log m}) rather than O (m n), where m and n are the lengths of the two lists, respectively.

2.30. Lecture Notes 27 Sorting

Today's Schedule

- Sorting
- Bubble Sort
- Selection Sort
- Insertion Sort

Sorting

- Desirable properties:
  - Fast, both average-case and worst-case
  - In-place or, more generally, no more than constant additional space required to accomplish sort. Note: we prefer to use arrays if possible to avoid the extra space for links in linked lists.
  - Stable – elements with the same key maintain their original order with respect to each other
- Two classes of sort algorithms
  - Internal: sort can all be done in memory
  - External: list is too large to be held in memory so that portions are always stored on external media

Selection Sort

- Algorithm
  - Selection sort works by finding the item that goes at the end of the list (or the beginning), moving that item to the correct location, and then sorting the rest of the items (either iteratively or recursively).
  - This continues until the entire array is sorted.
- Example
  if we start with: 3 2 4 5 1.
  The first step is to find the largest (5), and swap it with the last element yielding:
  3 2 4 1 5
  The next step finds the next largest and swaps it with the next to last position:
  3 2 1 4 5.
- Properties:
  - O (n^2) worst-case and average-case time (O (n^2)
  - In-place
  - Stable
  - Simplicity and low overhead make it good for small lists or lists with few out-of-order elements.

Bubble Sort

- Algorithm

- Comparisons are only performed between adjacent elements in the array. The elements are swapped if they are in the wrong order. Each pass through the data considers each pair of adjacent elements that are unsorted.
- After the first pass, the largest element must be in the last element of the array, so the next pass only has to consider one less element.

- Example
  if we start with: 3 2 4 5 1.
  We first compare the first two elements (swapping them, since they are out of order):
  2 3 4 5 1
  We then compare the second and third elements (not swapped), the third and fourth (not swapped) and the fourth and fifth elements (swapped)
  2 3 4 1 5
  This has "bubbled" the 5 "up" to the last position. Since, the array isn't sorted yet, we have to do another pass, but this time we don't have to go as far, since we know the largest element is in the last position. This continues until the array is sorted.

- Properties:
  - O (n^2) worst-case and average-case time (O (n^2)
  - In-place
  - Stable
  - Simplicity
  - Bubble sort performs many more swaps then other sorting techniques. If the data elements are very large, this is a significant drawback.

Insertion Sort

- Algorithm
  - Starting with the second element, iteratively insert each element into its correct location among the already sorted elements to its left.
  - This is accomplished by iteratively swapping the element with every element to its left that is larger than it.
- Example
- Properties:
  - O (n^2) worst-case and average-case time (O (kn) for k out-of-order elements)
  - In-place
  - Stable
  - Simplicity and low overhead make it good for small lists or lists with few out-of-order elements.

2.31. Lecture Notes 28 Sorting (cont.)

Today's Schedule

- Quick Sort
- Merge Sort
- How fast can we sort?

Quick Sort

- Algorithm
  - Choose a pivot element, effectively place it in its correct position in the sorted list by sorting all other elements to either side appropriately.
  - This is accomplished as follows: move a left pointer from the second element towards the right until it reaches an element with a key larger than the pivot element's key. Similarly, move a right pointer from the end towards the left until it reaches an element with a key as small as the pivot element's key. Swap the two elements pointed to by the pointers. Repeat until the left pointer moves passed the right pointer, then swap the pivot element with the element pointed to by the left pointer.
  - Finally, recursively sort the sublists on either side of the pivot element.
- Example: 3 2 5 4 1

  Choose pivot: 3
  Partition: 2 1 3 5 4
  Sort recursively: 1 2 3 4 5

- Picking pivot element: a better choice than the first element in the list would be the median of the first, middle, and last elements of the list.
- Properties:
  - O(n log n) average-case time, $O(n^2)$ worst-case time
  - Ω(log n) and O(n) space for stack (although can be made Θ(log n) by always sorting smaller sublists first)
  - *Un*stable.
  - Best average-case time; good for larger lists.

Merge Sort

- Algorithm
  - Iteratively merge sublists starting with lists of 1 element each.
  - Equivalently, divide list into two, recursively sort the sublists, then merge the sorted results.
- Example: 3 2 5 4 1

  Array: 3 2 5 4 1
  Split: 3 2 5 4 1
  Sort (recursively): 2 3 5 1 4

Merge: 1 2 3 4 5

- Properties
  - O (n log n) worst-case (and, hence, average-case) time.
  - O (n) additional space if linked lists, recursion, or simple merge algorithm are used; can be reduced to O (1) using more complicated merge algorithm, but this slows down the algorithm (though not asymptotically).
  - Stable.
  - Best worst-case time of methods.

How fast can we sort?

- Decision tree: A tree representing the steps that any algorithm based only on comparisons and swaps must do. Every one of the sort algorithms we are studying effectively computes a decision tree. (See p. 409 of the textbook for an example of a decision tree for three elements.)
- For n possible elements, there are n! possible permutions, so the decision tree has at least n! leaves. Since a decision tree is binary, one with n! leaves must have a height of at least $\log_2(n!) + 1$ which, with a little mathematical manipulation, can be shown to be $\Omega(n \log n)$.
- Thus, any sort algorithm that uses only comparisons and swaps has $\Omega(n \log n)$ worst-case time. We can also show that the same holds true for the average-case time.

2.32. Lecture Notes 29 Sorting (cont.)

Today's Schedule

- Heap Sort
- Radix Sort
- Summary

Heap sort

- Idea: insert n elements into an initially empty heap, then extract them one by one in (descending) order. Each insert and each extraction takes O (log n) time (both worst-case and average-case), so the whole sort takes O (n log n) time.
- So as to make the sort algorithm in-place and not require more than O(1) extra space, we can use a function ***heapify*** (***adjust*** in the textbook) to both create the initial heap and readjust it after each extraction. This function assumes that a tree is almost a heap, that is, the subtrees of the root are heaps but the root may have a key that is smaller than one of its children. It moves the root down the tree until the heap property is satisfied. This function takes O(h) time where h is the size of the tree.
- Sort algorithm using ***heapify***:
  - Assume list is implemented as an array.
  - Create initial heap by iteratively calling ***heapify*** on the n nodes in reverse level order. O (n) time.
  - Iteratively swap first element (root of tree) with last element of heap, decrement heap size by 1 to exclude swapped element that was the root, and call ***heapify*** on this smaller heap until heap size is 1. O (n log n) time.
- Thus, the whole algorithm takes O (n log n) time.

Radix sort (briefly)

- Algorithm
  - No comparisons are ever done between array elements.
  - The basic idea is to group the elements into sets, where the sets can be placed in a known order. An example is in sorting 3 digit numbers between 000 and 999. All numbers that start with a 0 can be placed in a group, all numbers starting with 1 in another, and so. When we are done with the first digit, we know what order to place the sets in, but we must still sort within each set. This is performed using the same process on the second digit of the number (recursively or iteratively). In practice, this is usually done in the reverse order of the digits, so that the most important digit is examined last. However, for each step, this means we must keep the elements with the same digit at the current position in the same order that they were in from the previous step to make sure that the numbers stay sorted correctly.
- Example: 532 294 534 256 123

Sorting by the least significant digit yields:
532 123 294 534 256
Sorting by the second digit yields:
123 532 534 256 294
Finally, sorting by the first digit yields:
123 256 294 532 534

- Properties:
  - Even if each element has only one key, we can decompose keys into multiple keys and apply MSD or LSD. This is called *radix sort*. If each key is decomposed into d keys, each of which can take on r values, then radix sort takes O (d (n + r)) time.
  - Simplicity and low overhead make it good for small lists or lists with few out-of-order elements.
- Consider the situation where we want to sort a list of elements lexicographically using multiple keys. There are two main algorithms:
  - Most-Significant-Digit-First (MSD) – this is what humans typically use: first sort on the primary key, then sort each sublist on the secondary key, etc.
  - Least-Significant-Digit-First (LSD) – use a stable sort on the least significant key first, then re-sort the whole list using a stable sort on the second-to-least significant key, and so on. Benefit: No need to have independent sorts on sublists, so less overhead.

Summary

- We have looked at several algorithms for sorting arrays in the last two lectures. They can be summarized as follows:

	Worst case	Average case	Comments
Selection sort	n^2	n^2	Use only for small arrays
Bubble sort	n^2	n^2	Use only for small arrays or nearly sorted arrays
Insertion sort	n^2	n^2	Use only for small arrays or incremental processing
Merge sort	n log n	n log n	Good algorithm, but requires extra space
Quick sort	n^2	n log n	Good general purpose algorithm
Radix sort	n	n	Not suitable for many applications

2.33. Lecture Notes 30 Course Review

Today's Schedule

- Course Review

Course Review

- **Software development issues:** time complexity, space complexity, software maintenance, algorithmic maintenance, reusability, encapsulation and abstraction
  - How data structure, algorithm, and implementation choices trade these off
- **Tools**
  - Implementation: recursion, delegation, templates, inheritance, union
    - Use of each
  - Performance: asymptotic notation, measurement
    - Asymptotic notation proofs
    - Asymptotic complexity estimations
    - Measurement techniques and issues
  - Correctness proofs: induction
  - Testing: good testing practices
- For each data structure or abstract data type, consider the following issues (as applicable):
  - Definition and operations
  - Usage (how and when)
  - Alternative representations of each ADT and corresponding operator implementations, with associated tradeoffs
  - Complexity considerations
  - Optimizations (e.g., circular array optimization for queues)
  - Extensions (e.g., heaps extend binary trees)
  - Applications (e.g., polynomials)
- **Data structures:** arrays, linked lists, iterators, hash tables
  - Arrays
  - Linked lists
    - Alternative representations and tradeoffs
    - Extensions:
      - Circular linked lists
        - Implementation
        - Advantages
      - Doubly linked lists
        - Operation implementation: insert, delete
        - Time/space tradeoff
        - Optimizations: using XOR to reduce storage requirements
      - Heterogeneous lists
        - Implementation: inheritance vs. **union**
        - Usage

- Iterators
  - Definition and operations
  - Usage
  - Implementation
- Hash tables
  - Definition and operations
  - Issues: criteria for a good hash table
  - Hash functions
    - criteria for a good function
    - division function
  - Handling collisions and overflow
    - open addressing
    - chaining

- **Abstract data types:** array, sparse matrices, strings, set, bags, ordered lists, stacks, queues, priority queues, trees, binary trees (BST, AVL), heaps, threaded trees, graphs,
  - Set and Bag ADT
    - Definition and operations
    - Possible implementations and complexity implications
  - Ordered list ADT
    - Definition and operations
    - Alternative implementations and tradeoffs: array vs. linked list
    - Extension: Generalized list ADT
      - Definition
      - Implementing recursive operations
      - Linked list representation
      - Issues: Recursive and shared sublists issues
  - Stack ADT
    - Definition and operations
    - Alternative implementations: array, linked list
  - Queue ADT
    - Definition and operations
    - Alternative implementations
    - Optimization: circular array
  - Priority Queue ADT
    - Definition and operations
    - Alternative implementations and tradeoffs: array, linked list, ordered list, heap
  - Tree ADT
    - Representations: generalized list, left-child/right-sibling, binary tree
    - Conversion between representations
  - Binary tree ADT
    - Definition and terminology
    - Properties: height, maximum number of nodes at a level

        - Operations: insertion, deletion, traversals (in-/pre-/post-order), copy, equality, etc.
        - Representations: array-based, linked list-based
        - Operation implementation: iterative and recursive
        - Optimizations: header nodes
        - Extension: Heap ADT
            - Definition
            - Operation implementation
            - Property maintenance
            - Complexity considerations
            - Application: priority queue
        - Extension: Binary search tree (BST) ADT
            - Definition
            - Operation implementation
            - Property maintenance
            - Complexity considerations
            - Application: search
        - Extension: ALV Tree ADT
            - Definition
            - Operation implementation
            - Property maintenance
            - Complexity considerations
            - Application: search
    - Graph ADT
        - Definition and terminology
        - Operations: DFS and BFS traversal, shortest path (Dijkstra's algorithm), finding spanning trees (definition)
        - Representations: adjacency matrix, adjacency lists
        - Extension: network (weighted graph)
        - Minimum Cost Spanning Trees
    - Hash table
        - Definition and operations
        - Implementation
- **Algorithms and applications**
    - Polynomials
        - Alternative representations and tradeoffs
        - Operations: addition
    - Expression evaluation
    - Search
        - Alternative implementations and tradeoffs: array, linked list, BST, hash table, etc.
    - Sorting
        - Alternative sorting algorithms: bubble sort, selection sort, insertion sort, quick sort, b sort, mean sort, shell sort, merge sort, heap sort, radix sort, external sorting (introduction)

- Characteristics of sorting algorithms: time and space complexity (worst-case and average-case), in-place, stability
- Choosing between algorithms

Part 3

Laboratory Manual

3.1. Lab 01 ADT Implementation

Objectives:

- Concepts of ADT
- Practice OO concepts i.e. class, constructor/destructor, operator overloading etc.
- Design and Implementation of Array ADT

Overview:

A kind of data abstraction where a type's internal form is hidden behind a set of access functions. Values of the type are created and inspected only by calls to the access functions. This allows the implementation of the type to be changed without requiring any changes outside the module in which it is defined.

Objects and ADTs are both forms of data abstraction, but objects are not ADTs. Objects use procedural abstraction (methods), not type abstraction. A classic example of an ADT is a stack data type for which functions might be provided to create an empty stack, to push values onto a stack and to pop values from a stack.

As **Abstract Data Type** (**ADT**) is a data structure and a collection of functions or procedures, which operate on the data structure. To align ourselves with Object-oriented (OO) theory, we'll call the functions and procedures as **methods** and the data structure and its methods a **class**, *i.e.* we'll call our ADTs as classes. However our classes do not have the full capabilities associated with classes in OO theory. An instance of the class is called an *object*. ***Objects*** represent objects in the real world and appear in programs as variables of a type defined by the class. These terms have exactly the same meaning in OO design methodologies, but they have additional properties such as inheritance that we will not be discussing in this course.

An Example: Collections

Programs often deal with collections of items. These collections may be organized in many ways and use many different program structures to represent them, yet, from an abstract point of view, there will be a few common operations on any collection. These might include:

`Create`	Create a new collection
`Add`	Add an item to a collection
`Delete`	Delete an item from a collection
`Find`	Find an item matching some criterion in the collection
`destroy`	Destroy the collection

Constructors and destructors

Create and destroy methods - often called constructors and destructors - are usually implemented for any abstract data type. Occasionally, the data type's use or semantics are such that there is only ever one object of that type in a program. In that case, it is possible to hide even the object's "handle" from the user. However, even in these cases, constructor and destructor methods are often provided.

Of course, specific applications may call for additional methods, e.g. we may need to join two collections (form a union in set terminology) - or may not need all of these.

Pre-Lab Exercise: Dynamic Arrays

Study article 1.5 (p. 20), 1.6.1.2 (p. 32), 1.6.2 (p. 51), 1.6.3 (p. 60) from your textbook and the following paragraphs.

Probably the most common way to aggregate data is to use an array. While the C++ programming language does indeed provide built-in support for arrays, that support is not without its shortcomings. Arrays in C++ are not first-class data types. There is no such thing as an array-valued expression. Consequently, you cannot use an array as an actual value parameter of a function; you cannot return an array value from a function; you cannot assign one array to another. (Of course, you *can* do all of these things with a *pointer* to an array). In addition, array subscripts range from zero to *N*-1, where *N* is the array size, and there is no bounds checking of array subscript expressions. And finally, the size of an array is static and fixed at compile time, unless dynamic memory allocation is explicitly used by the programmer.

Some of these characteristics of arrays are due in part to the fact that in C++, given a pointer, **`T* ptr`**, to some type, **`T`**, it is not possible to tell, just from the pointer itself, whether it points to a single instance of a variable of type **`T`** or to an array of variables of type **`T`**. Furthermore, even if we know that the pointer points to an array, we cannot determine the actual number of elements in that array.

It is primarily to address these deficiencies that we introduce the **`Array`** object, which is implemented as a generic class. Figure 1 below illustrates the `Array` object as represented in the memory of the computer. Two structures are used. The first is a structure, which comprises three fields - **`data,`** and **`length`**. The member variable **`data`** is a pointer to the array **data**. Variables **`length`** is used in the array subscript calculation. The second structure comprises contiguous memory locations, which hold the array elements. In the implementation given below, this second structure is allocated dynamically.

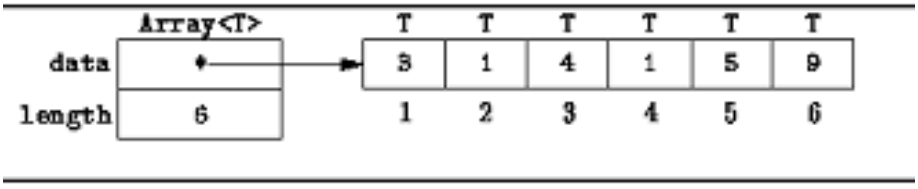


Figure 1: Memory Representation of Array Objects

A C++ declaration of the **`Array<T>`** class template is given in program below. The **`Array<T>`** class has two **private** member variables, **`data,`** and **`length`**, constructors, destructor, and various member functions. The number of member functions has been kept to the bare minimum in this example. In the "real world" you can expect that such a class would contain many more useful member functions.

Bridge Exercise:

Unfortunately, since `Array<T>` is a generic class, we have no ***a priori*** knowledge of the amount of storage used by an object of type `T`.

// `Array<T>` Class Definition

```
template <class T>
class CArray
      {
         private:
              T* data;
              unsigned int length;
          public:
              Array ();
              ~Array ();
              Array (Array const&);
              unsigned int m_fnLength ();
              void m_fnSetLength (unsigned int);
              void m_fnSetData (T Data, int Index);
              T m_fnGetData (int Index);
      };
```

Driver Program:

Driver Program for CArray class is as follows

```
int main(void) {
CArray <int> FirstArray;

CArray <int> SecondArray (FirstArray);

FirstArray.m_fnSetLength (5);
FirstArray.m_fnSetData (21, 2);
FirstArray.m_fnSetData (-15, 3);
FirstArray.m_fnSetData (11, 1);
cout  << FirstArray.m_fnGetData (1) << endl;
cout  << FirstArray.m_fnGetData (2) << endl;
cout  << FirstArray.m_fnGetData (3) << endl;
return 0;
}
```

In-Lab Exercise 1:

Write the following functions for CArray class.

Boolean m_fnIsFull ();
Boolean m_fnIsEmpty ();
int m_fnSpaceLeft ();
Boolean m_fnSearch (T);

Add following statements at the end of driver programs,

cout << FirstArray.m_fnIsFull () << endl;
cout << FirstArray.m_fnIsEmpty () << endl;
cout << FirstArray.m_fnSpaceLeft () << endl;
cout << FirstArray.m_fnSearch (21) << endl;

Write the output of above programs.

In-Lab Exercise 2:

Overload following operators for CArray class

ostream & operator << (ostream &, const CArray <Type>&);
//display all contents of array

Type & operator [] (int); // similar to m_fnGetData

Add following statements at the end of driver programs,

cout << FirstArray << endl;
cout << FirstArray [3]<< endl;

Files to be submitted:

Put the following files in a zip archive/folder name it on your roll number and upload at \\pucit-waqar\assignments. Your code will be compiled and run to make sure that it works as required.

The files to be submitted are:

1. `Bridge.cpp`
   `Contains the driver program for bridge exercise`
2. `Exarecise1.cpp`
   `Contains the driver program for bridge exercise`
3. `Extra.cpp`
   `Contains source for extra lab exercise`

3.2. Lab 2: Performance Measurement

Objectives:

- Compute the Time Complexity of a program.

Overview:

There are many criteria to evaluate a program i.e.

- Program correctness
- Logical decomposition
- Program style
- Program documentation
- Appearance of the program's output
- Following specifications etc.

But there are many others that directly relates to program performance and robustness. These have to do with program computing time and storage requirements, which are called its time and space complexity.

Performance evaluation of a program can be divided into two major phases: (1) a priori estimate (2) posteriori measurement, these are called performance analysis and performance measurement respectively.

Time Complexity:

Time complexity of a program is the numbers of steps taken by the program to complete the task it is written for. Number of steps is itself a function of instance characteristics (number, size, type etc. of input and output).

Time complexity of a program is measured using a global count variable that is incremented after the execution of program statement.

Time complexity of a program is dependent on size and arrangement of data inputted to the program i.e. if we input an array and a search key to a program the searches the key then the search time is dependent on the size of array and the arrangement of values in it. Time complexity of a program can further be divided into three types best, average and worst time complexities.

Pre-Lab Exercise:

Study article 1.5 (p. 20), 1.6.1.2 (p. 32), 1.6.2 (p. 51), and 1.6.3 (p. 60) from your textbook.

Bridge Exercise:

Code the following function for

int fnLinearSearch (int Array [], unsigned int Size, int SearchKey);
Precondition:
Post condition: return first index of search key in the array if found –1 otherwise

int fnBinarySearch (int Array [],unsigned int Size, int SearchKey);
Precondition: Array is sorted in ascending order
Post condition: return first index of search key in the array if found –1 otherwise

void fnBubbleSort (int Array [],unsigned int Size, int SortKey);
Precondition:
Post condition: Array should be ordered according to sort key.

void fnSelectionSort (int Array [],unsigned int Size, int SortKey);
Precondition:
Post condition: Array should be ordered according to sort key.

In-Lab Exercise 1:

Write a driver program that populates an array of size n with random values from a specified range of values. Select a random search key from the same set of values and fill the following table after the execution of fnLinearSearch

Serial No.	Input Size	Input Range	Search Key	Instruction Count
1				
2				
3				

Write a driver program that populate an array of size n with random values from a specified range of values and sort it in ascending order. Select a random search key from the same set of values and fill the following table after the execution of fnBinarySearch

Serial No.	Input Size	Input Range	Search Key	Instruction Count
1				
2				
3				

In-Lab Exercise 2:

Write a driver program that populates an array of size n with random values from a specified range of values. Sort the instance array using fnBubbleSort

Serial No.	Input Size	Input Range	Instruction Count
1			
2			
3			

Write a driver program that populates an array of size n with random values from a specified range of values. Sort the instance array using fnSelectionSort

Serial No.	Input Size	Input Range	Instruction Count
1			
2			
3			

Mark best, average and worst cases for both sorting algorithms. Calculate the average percentage efficiency achieved using selection sort.

In-Lab Exercise 3:

Using time.h get system time before the execution and after completion of above functions and write your calculated time for execution below

Program	Input Size	Input Range	Search/Sort Key	Calculated Time
Linear Search				
Binary Search				
Bubble Sort				
Selection Sort				

Driver Program:

```
#include <stdio.h>
#include <dos.h>
#include <constream.h>

int main(void)
{
struct time First, Second;

clrscr();
gettime(&First);

//
// Write your function call here
//
gettime(&Second);
//
// Compute the difference and display as following
```

```
//

printf("The current time is: %2d:%02d:%02d.%02d\n", First.ti_hour, First.ti_min,
First.ti_sec, First.ti_hund);

printf("The current time is: %2d:%02d:%02d.%02d\n", Second.ti_hour, Second.ti_min,
Second.ti_sec, Second.ti_hund);

return 0;
}
```

Extra-Lab Exercise:

Modify binary search and selection sort functions to their equivalent recursive functions and compare their time efficiency with non-recursive functions.

Files to be submitted:

Put the following files in a zip archive/folder name it on your roll number and upload at \\pucit-waqar\assignments. Your code will be compiled and run to make sure that it works as required.

The files to be submitted are:

1. `Driver.cpp`
   `Contains the driver program and sorting functions`
2. `Test.doc`
   `MS Word file that contains filled tables for test data`
3. `Extra.cpp`
   `Contains source for extra lab exercise`
4. `Extra.doc`
   `MS Word file that contains filled tables of test data for extra lab.`

3.3. Lab 03: Implementation of Polynomial ADT

Objectives:

- Revision of different C++ concepts i.e. dynamic memory allocation, operator overloading, pointer operations, copy constructor etc
- Implementation of Polynomial ADT

Overview:

Polynomial:

Let us consider an important ADT that motivates the need for an efficient implementation of an ordered-list ADT: the *polynomial ADT*. Polynomial of a variable **'x'** is defined as

$$P(x) = \sum_{i=0}^{n} a_i x^i$$
$$= a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_2x^2 + a_1x^1 + a_0$$

Where **a's** are integer constants and **n** is any nonnegative integer. An example of a polynomial is

$$A(x) = x^4+3x^2+2x+4.$$

A polynomial can be characterized as a mapping from the set of non-negative integers (the exponents of each term) to coefficients. The largest exponent with a nonzero coefficient is referred to as the *degree* of the polynomial.

Operations: add, subtract, multiply, divide, evaluate, get degree, get coefficient given exponent, add term, get root etc.

The sum of two polynomials is obtained by adding together the coefficients sharing the same powers of variables (i.e., the same terms) so, for example,

$A(x) = (a_2 x^2 + a_1x^1 + a_0)$
$B(x) = (b_1x^1 + b_0)$
$A(x) + B(x) = (a_2 x^2 + a_1x^1 + a_0) + (b_1x^1 + b_0)$
$= a_2 x^2 + (a_1 + b_1) x^1 + (a_0 + b_0)$

and has degree equal to the maximum degree of the original two polynomials. Similarly, the product of two polynomials is obtained by multiplying term by term and combining the results, for example,

$A(x) B(x)$
$= (a_2 x^2 + a_1x^1 + a_0) (b_1x^1 + b_0)$
$= a_2 x^2 (b_1x^1 + b_0) + a_1x^1 (b_1x^1 + b_0) + a_0 (b_1x^1 + b_0)$
$= a_2 b_1 x^3 + (a_2 b_0 + a_1 b_1) x^2 + (a_1 b_0 + a_0 b_1) x + a_0 b_0$

and has degree equal to the sum of the degrees of the two original polynomials.

Array Representations of Polynomial:

Representation 1 (brute force): use an array of length MaxDegree+1 to store coefficients.

```
private:
        int degree;
        float coef[MaxDegree + 1];
```

Note that $a.coef[i] = a_{n-i}$ where $n = degree$. This simplifies algorithms for many of the operations.

Problem – wastes space when ***degree*** is much less than MaxDegree.

Representation 2**:** dynamically allocate space for array once ***degree*** is known.

```
private:
        int degree;
        float *coef;
```

Problem – wastes space for sparse polynomials where there are many zero terms.

Representation 3**:** use array of terms, where only terms with nonzero coefficients are stored. Also, use one array to store multiple polynomials.

```
class CTerm {
friend Polynomial;
private:
        float coef;
        int exp;
};

class CPolynomial {
…
private:
        static Term termArray[MaxTerms];
        static int free;
        int start, finish;
};
```

Usually preferable to other two representations, but takes more space if there are few zero terms.

Problem – difficult to reuse space of polynomials that are no longer useful since moving polynomials around in an array is expensive.

Problem – if we want each array to represent one polynomial (a more natural, encapsulated representation for many applications), we either potentially waste space if

MaxTerms is too high, or waste time computing the number of terms in each polynomial. For example, it takes as much work to determine the number of terms in the result of adding two polynomials as it takes to do the addition.

***Representation 4*:** use array of terms, where only terms with nonzero coefficients are stored.

```
class CTerm {
friend SparsePolynomial;
private:
        float coef;
        int exp;
};

class CSparsePolynomial {
…
private:
        Term *data;
        int NumberOfTerms, Degree;
};
```

Use constructor function to take number of terms, after allocating required memory store it into term pointer data.

***Representation 5*:** Use linked lists – coming soon …

Public functions:

The Polynomial ADT or class has the following public functions

1. **Constructor ()**
   Construct zero polynomial
2. **Constructor (degree/no of Terms)**
   Construct a polynomial with provided degree or no of terms for sparse polynomial
3. **SetDegree (Degree)**
   Sets the degree of the polynomial to provided Degree. Truncate higher degree terms is degree is smaller then previous one. Note: this function is not for implementation 4 of sparse polynomial.
4. **GetDegree ()**
   Returns the degree of a polynomial. For example, when p1 = $4x^5 + 7x^3 - x^2 + 9$, `p1.degree ()` is 5.
5. **GetCoefficient (power)**
   Returns the coefficient of the x^{power} term. For example, `p1.getCoefficient (3)` is 7
6. **SetCoefficient (newCoefficient, power)**
   Sets the coefficient of the x^{power} term with newCoeffcient. For instance, `p1.setCoefficient (-3, 7)` produces the polynomial p1 = $-3x^7 + 4x^5 + 7x^3 - x^2 + 9$. Note that if the term does not exist in the polynomial, it is added.

7. **AddToCoefficient (amount, power)**
   Adds the given amount to the coefficient of the x^{power} term. For example, `p1.addToCoefficient (2.5, 3)` changes the coefficient of x^3 to 9.5.
8. **AddTerm (coefficient, power)**
   Add a new term x^{power} term in the polynomial and sets it coefficient.
9. **NextTerm (power)**
   Returns the exponent 'n' which is **larger** than the exponent 'power' and coefficient of the term is not zero, i.e, getCoefficient(n) != 0. For example, given $p1 = 4x^5 + 0x^4 - 7x^3 - x^2 + 9$, p1.nextTerm (3) will return 5 because the coefficient of x^4 term is zero. If there is no such term then the function returns 0.
10. **PrevTerm (power)**
    Returns the exponent 'n' which is smaller than the exponent 'power' and coefficient of the term is not zero, i.e, getCoefficient(n) != 0.. For example, given $p1 = 4x^5 + 0x^4 - 7x^3 - x^2 + 9$, p1.nextTerm (3) will return 2. If there is no such term then the function returns UINT_MAX. This constant is defined in the file include via `#include <limits.h>`.
11. **Clear ()**
    Sets the coefficient of all terms in the polynomial to zero. Notice that the terms are not deleted.
12. **Derivative ()**
    Returns a polynomial that is the derivative of the given polynomial. For instance, assuming $p1 = 4x^5 + 7x^3 - x^2 + 9$, the derivative polynomial would be $20x^4 + 21x^2 - 2x$.
13. **AntiDerivative ()**
    Returns a polynomial that is the anti-derivative of the given polynomial. For instance, assuming $p1 = 20x^4 + 21x^2 - 2x$, the derivative polynomial would be. $4x^5 + 7x^3 - x^2 + C$. Where C is a random constant value.
14. **Density ()**
    Density of the polynomial is defined as
    $$\rho (P) = (\text{Number of none zero Coefficients}) / (\text{Degree})$$
    It is very significant for sparse polynomials.
15. **Arithmetic operations *, + and -**
    Add two polynomials or subtract the 2nd polynomial from the 1st one. For instance, assuming $p1 = 4x^5 + 7x^3 - x^2 + 9$ and $p2 = 6x^4 + 3x^2 + 2x$, then the answer of p1 + p2 must be $4x^5 + 6x^4 + 7x^3 + 2x^2 + 2x + 9$. Example of multiplying two polynomials has been given earlier.
16. **Assignment operator =**
    Note that you need to take care of self-assignment when implementing =.
17. **operator (value)**
    For example, given a polynomial in "p1", p1(x) will evaluate the polynomial for the given value of x.
18. **Output operator <<**
    The output must have the form of $c_nx\text{^}n + c_{n-1}x\text{^}(n-1) + ... + c_1x + c_0$ For example, $p1 = -3x^7 + 4x^5 + 7x^3 - x^2 + 9$ should be printed out in the form of -3x^7 + 4x^5 + 7x^3 - x^2 + 9. Note that you don't have to implement operator>>.
19. **Copy constructor**

Takes care of a deep copy of the original link representation.

20. **Destructor**
    Takes care of de-allocating memory to the system.

Pre-Lab Exercise:

Study article 2.1 (p. 66), 2.2 (p. 75), 2.3 (p. 77), and review Exercise (p. 85) from your textbook.

Bridge Exercise:

Implement a class CPolynomial for representation 2 from array representation section.

```
class CPolynomial {
public:
CPolynomial();
CPolynomial(int Deg);
CPolynomial(const CPolynomial &);
SetDegree(int Deg);
int GetDegree();
        float GetCoefficient (int power) ;
        SetCoefficient (float newCoefficient, int power) ;
        AddToCoefficient (int amount, int power) ;
        AddTerm (float coefficient, int power) ;
        int NextTerm (int power);
        int PrevTerm (int power) ;
        Clear () ;
        CPolynomial operator = ();
        float operator ( value );
        ostream & operator << ();
        ~ CPolynomial();

private:
                int degree;
                float *coef;
        };
```

Implement a class CSparsePolynomial for representation 4 from array representation section.

```
class CTerm {
friend SparsePolynomial;
private:
        float coef;
        int exp;
};
```

```
class CSparsePolynomial {

public:
        CSparsePolynomial();
        CSparsePolynomial(int Deg);
        CSparsePolynomial(const CPolynomial &);
        SetNoOfTerms (int NewNoOfTerms);
        int GetNoOfTerms ();
        int GetDegree();
        float GetCoefficient (int power) ;
        SetCoefficient (float newCoefficient, int power) ;
        AddToCoefficient (int amount, int power) ;
        AddTerm (float coefficient, int power) ;
        int NextTerm (int power);
        int PrevTerm (int power) ;
        float Density ();
        Clear () ;
        CSparsePolynomial operator = ();
        float operator ( value );
        ostream & operator << ();
        ~ CSparsePolynomial();

private:
            Term *data;
            int NumberOfTerms, Degree;
        };
```

Driver Programs:

Driver Program 1: Driver Program for CPolynomial class is as follows

```
int main(void) {

CPolynomial FirstPolynomial, SecondPolynomial (4);

FirstPolynomial.SetDegree(5);
FirstPolynomial.AddTerm(15, 5);
FirstPolynomial.AddTerm(-4, 3);
FirstPolynomial.AddToCoefficient (22.7, 3);
FirstPolynomial.AddTerm(9, 1);

SecondPolynomial.AddTerm(21.5, 2);
SecondPolynomial.AddTerm(12, 1);
SecondPolynomial.AddTerm(3, 0);

cout  << "1:-" << FirstPolynomial << endl;
```

```
cout  << "2:-" << SecondPolynomial << endl;

cout  << "Previous none zero exponent for exponent 5 in First polynomial is " <<
FirstPolynomial.PrevTerm(5) << endl;

cout  << "Next none zero exponent for exponent 1 in Second polynomial is " <<
FirstPolynomial.NextTerm(1) << endl;
return 0;
}
```

Driver Program 2: Driver Program for CSparsePolynomial class is as follows

```
int main(void) {

CSparsePolynomial FirstPolynomial, SecondPolynomial (5);

FirstPolynomial.SetNoOfTerms(7);
FirstPolynomial.AddTerm(15, 50);
FirstPolynomial.AddTerm(-4, 13);
FirstPolynomial.AddToCoefficient (22.7, 13);
FirstPolynomial.AddTerm(9, 1);

SecondPolynomial.AddTerm(21.5, 120);
SecondPolynomial.AddTerm(12, 1);
SecondPolynomial.AddTerm(3, 0);

cout  << "Polynomial 1:- " << First Polynomial << endl;
cout  << "Polynomial 2:- " << SecondPolynomial << endl;

cout  << "Density of Polynomial 1:- " << FirstPolynomial.Density() << endl;
cout  << "Density of Polynomial 2:- " << SecondPolynomial Density() << endl;

cout  << "Previous none zero exponent for exponent 5 in First polynomial is " <<
FirstPolynomial.PrevTerm(50) << endl;

cout  << "Next none zero exponent for exponent 1 in Second polynomial is " <<
SecondPolynomial.NextTerm(1) << endl;
return 0;
}
```

Write the output of above programs.

In-Lab Exercise 1:

Overload Arithmetic operations *, + and – for CPolynomial and CSparsePolynomial class.

Add following statements at the end of driver programs,

```
cout << "Sum of Polynomials = " << FirstPolynomial + SecondPolynomial << endl;
cout << "Difference of Polynomials = " << FirstPolynomial - SecondPolynomial << endl;
cout << "Product of Polynomials = " << FirstPolynomial * SecondPolynomial << endl;
```

Write the output of above programs.

In-Lab Exercise 2:

Write the following functions for CPolynomial and CSparsePolynomial classes Copy Constructor, Derivative (), AntiDerivative (), operator ( ) for Polynomial evaluation.

Add following statements at the end of driver programs,

```
CPolynomial ThirdPolynomial (FirstPolynomial);
CPolynomial ThirdPolynomial (FirstPolynomial.Derivative();
CPolynomial FourthPolynomial = FirstPolynomial.AntiDerivative();

cout << "Derivative of Polynomial 1:- " << ThirdPolynomial << endl;
cout << "Anti Derivative of Polynomial 1:- " << FourthPolynomial << endl;
cout << "Result of FirstPolynomial for x = 2 is " << FirstPolynomial (2)<< endl;
```

Write the output of above programs.

Extra Lab:

Try to construct objects of CPolynomial and CSparsePolynomial classes to hold the two polynomials that are obtained from following statements

```
FirstPolynomial.AddTerm(15, 50);
FirstPolynomial.AddTerm(-4, 13);
FirstPolynomial.AddToCoefficient (22.7, 13);
FirstPolynomial.AddTerm(9, 1);

SecondPolynomial.AddTerm(21.5, 120);
SecondPolynomial.AddTerm(12, 1);
SecondPolynomial.AddTerm(3, 0);
```

Compare the time and space complexity for above two objects to store in the RAM using both implementations.
Compare the time complexity for following operations as well
+, *

Files to be submitted:

Put the following files in a zip archive/folder name it on your roll number and upload at \\pucit-waqar\assignments. We will compile and run your code to make sure it works as required.

The files to be submitted are:

1. `polynomialmain.cpp`
   `Contains the driver program for CPolynomial class`
2. `sparsepolynomialmain.cpp`
   `Contains the driver program for CSparsePolynomial class`
3. `Polynomial.h`
4. `Contains the class definition and implementation for CPolynomial class`
5. `SparsePolynomial.h`
6. `Contains the class definition and implementation for CSparsePolynomial class`
7. `Ouput.txt this file contains the outputs of bridge and lab exercise 1 and 2.`

3.4. Lab 04: Stack ADT

Objectives:

- Implementations of the Stack ADT— based on an array representation of stack.
- Use templates to produce a generic stack data structure.
- Check brackets validity of mathematical expressions.
- Write a function that converts an infix expression into equivalent postfix expression
- Create a program that evaluates arithmetic expressions in postfix form.

Overview:

Stack:

Stack is an example of a restricted linear data structure that follows the principle of Last In First Out (LIFO).

Stack can be considered as a container object whose one of two faces is ceiled. User can only insert and remove items in this container from opened face. That's way lastly inserted item in the stack will be removed first. In a stack, the data items are ordered from most recently added (the **top**) to least recently added (the **bottom**) this property can be used for developing several algorithms that have a wide variety of applications.

Stacks are often used to hold information about "postponed" operations that need to be performed later, it is called backtracking i.e. Operating systems often use stacks to keep track of recursive function calls. Same service of the stack helps programmer to create Artificial Intelligence in a program, where computer has to respond intelligently against a set of expected user actions.

Stacks are used by compilers to help in the process of evaluating expressions and generating machine language code. In this and the next exercise, we investigate how compilers evaluate arithmetic expressions consisting only of constants, operators, and parentheses.

Humans generally write expressions like **3 + 4** and **7 / 9** in which the operator (+ or / here) is written between its operands; this is called *infix*. Computers "prefer" *postfix notation* in which the operator is written to the right of its two operands. The preceding infix expressions would appear to postfix notation as **3 4 +** and **7 9 /,** respectively.

To evaluate a complex infix expression, a compiler would first convert the expression to postfix notation, and then evaluate the postfix version of the expression. Each of these algorithms requires only a single left-to-right pass of the expression. Each algorithm uses a stack object in support of its operation, and in each algorithm the stack is used for a different purpose.

Validation and evaluation of expression is the main object of today's Lab.

Array Representations of Stack:

Representation 1 (brute force): use an array of length MaxDegree+1 to store items.

```
private:
        int size;
        int top;
        int data[NumbersOfItems];
```

Problem – every stack object will have fixed number of Items.

Representation 2**:** dynamically allocate space for array once Size is known.

```
private:
        int size;
        int top;
        int *data;
```

Problem – only creates integer stack.

Representation 3**:** dynamically allocate space for array once Size is known using templates.

```
template <class StackType>
CStack      {
private:
        int size;
        int top;
        StackType *data;
        ……
};
```

Problem – creates problem if number of items to be inserted are not known at the time of stack definition.

Representation 4**:** Use linked lists – coming soon …

Public functions:

The Stack ADT or class has the following public functions

1. **Constructor (int size = DEFAULT)**
   Construct stack with user provided size
2. **Push (StackType Element)**
   Places the given Element the top of the stack. Generates error of overflow is stack is already full.
3. **StackType Pop()**

Returns most recently pushed element and discards it from stack. Generates error of underflow is stack is empty.

4. **Boolean IsFull ()**
   Returns TRUE if stack is full FALSE otherwise.
5. **Boolean IsEmpty ()**
   Returns TRUE if stack is empty FALSE otherwise.
6. **StackType SeeTop()**
   Returns most recently pushed element without discarding it from stack. Generates error of underflow is stack is empty.
7. **Copy constructor**
   Takes care of a deep copy of the original representation.
8. **Destructor**
   Takes care of de-allocating memory to the system.

Pre-Lab Exercise:

Study article 3.1 (p. 119), 3.2 (p. 124), 3.6 (p. 147), and review Exercises (p. 130, p.153) from your textbook.

Bridge Lab Exercise:

Implement a class CStack for representation 3 from array representation section.

```
template <class StackType>
class CStack {
public:
CStack (int MaxNoOfElements = DEFAULT);
CStack (const CStack &);
Push (StackType);
Boolean IsFull();
Boolean IsEmpty();
StackType Pop();
        StackType SeeTop() ;
        ~ CStack();
private:
                int size;
                int Top;
                StackType *data;
        };
```

Driver Programs:

Driver Program 1: Driver Program for CStack class is as follows

```
int main(void) {
```

```
CStack <int> IntStack(5);
CStack <char > CharStack(3);
CStack <char *> StringStack(5);

cout  << IntStack.Pop() <<  endl ;
for (int i=0; i<5; i++)
        IntStack.Push(i);

for (i=0; i<6; i++)
        cout << IntStack.Pop() << endl;
cout  << IntStack.Pop() <<  endl ;

CharStack.Push('P');
CharStack.Push('O');
CharStack.Push('P');

cout << CharStack.Pop() << CharStack.Pop() << CharStack.Pop() << endl;
StringStack.Push("to be");
StringStack.Push("or");
StringStack.Push("not");
StringStack.Push("to be");

cout  << StringStack.Pop() <<  endl << StringStack.Pop()<<  endl ;
cout  << StringStack.Pop() <<  endl << StringStack.Pop()<<  endl;
return 0;
}
```

Write the output of above programs.

In-Lab Exercise 1:

Write a functions that takes a ExpressionValidation string as argument and returns and CSparsePolynomial class.

Boolean fnExpressionValidation (String SExpression)
Precondition: A String that contains a mathematical expression with heterogeneous set of brackets might be null.
Post condition: return TRUE if expression SExpression is valid with respect to brackets i.e.

1. Number of opening brackets are equal to number of closing brackets
2. No closing bracket is preceding its respective opening bracket
3. Validation of nested brackets

Invalided expressions:
(A + (B + C)
(A + B) + C)

) A + B (
{(A + B} * C)

```
int main(void) {

CString SExpression(20);

cout  << "Enter a mathematical expression including brackets"<<  endl ;
cin >> SExpression;

if (fnExpressionValidation (SExpression))
        cout << "Valid expression" << endl;
else
        cout << "Invalid expression" << endl;

return 0;
}
```

Write the output of above programs for sample input data.

In-Lab Exercise 2:

In this exercise, you will write a C++ version of the infix-to-postfix conversion algorithm. In the next exercise, you will write a C++ version of the postfix expression evaluation algorithm. Using similar code, you could implement a complete working compiler.

Write a program that converts an ordinary infix arithmetic expression (assume a valid expression is entered) with single digit integers such as

(6 + 2) - 5 - 8 / 4

to a postfix expression. The postfix version of the preceding infix expression is

6 2 + 5 - 8 4 / -

The following arithmetic operations are allowed in an expression:

+ addition
- subtraction
* multiplication
/ division
^ exponentiation
% modulus

Some of the functional capabilities you may want to provide are:

a) Function **fnInfixToPostfix** that converts the infix expression to postfix notation.
b) Function **fnIsOperator** that determines if **ch** is an operator.
c) Function **fnPrecedence** that determines if the precedence of **Operator1** is less than, equal to, or greater than the precedence of **Operator2.** The function returns -1, 0, and 1, respectively.

Primarily you have to write a function to convert an infix expression into postfix expression.

CString fnInfixToPostfix (CString SInfixExpression)
Precondition: A String SInfixExpression that contains a valid mathematical expression in infix form with homogenous set of brackets i.e. ().
Post condition: return CString object that contains an equivalent postfix expression for SInfixExpression.

Sample Input and Outputs data:

A * B + C	A B * C +
A + B * C	A B C * +
A + B – C	A B + C –
A * (B + C)	A B C + *
A + (B – C) / (D – E)	A B C – D E – / +
(((A + B) * (C – D / E) – F * G) / (H – I)) * J)	A B + C D E / – * F G * – H I – / J *

Driver Program:

```
int main(void) {

CString SInfixExpression (32);
cout  << "Enter a mathematical expression including brackets"<<  endl ;
cin >> SExpression;

if (fnExpressionValidation (SExpression)) {
        cout << "Postfix of Expression " << SInfixExpression << "is "  <<
        endl;
        cout << fnInfixToPostfix (SInfixExpression) ;
}
else
        cout << "Invalid expression" << endl;

return 0;
}
```

Write the output of above programs for sample input data.

In-Lab Exercise 3:

Write a function to evaluate a postfix expression.

float fnEvaluatePostfix (CString SPostfixExpression)
Precondition: A String SPostfixExpression that contains a valid mathematical expression in postfix form with homogenous set of brackets i.e. ().
Post condition: return the value of evaluated expression.

Sample Input and Outputs data:

2 5 * 3 +	13
9 2 7 * +	23
2 1 + 3 –	0
9 2 3 + *	45
5 2 7 – 4 9 – / +	6
2 3 + 3 5 2 / – * 6 3 * – 9 2 – / 2 *	1.107142

Driver Program:

```
int main(void) {
CString SPostfixExpression (32);

cout  << "Enter a Postfix expression"<<  endl ;
cin >> SPostfixExpression;
cout << "Answer of postfix expression" << SPostfixExpression << "is " <<
endl;
cout << fnEvaluatePostfix (SPostfixExpression) ;

return 0;
}
```

Write the output of above programs for sample input data.

Extra Lab:

1. Modify the function fnInfixToPostfix() from Lab Exercise 2 so that it can receive a infix expression having heterogeneous bracket set and unary operators.
2. Modify the function fnEvaluatePostfix() from Lab Exercise 3 so that it can receive a postfix expression that contains integers with more then one digit and having unary operator associated with them.
3. Write a following function that converts an value in base 10 to user required base.

   CString fnDecimalToAnyBase(Value, Base)

Precondition: An integer value and valid integer base (i.e.2 – 36 include A–Z characters to show digits greater then 9).
Post condition: return a String that contains converted numbers in required base. In case of wrong input return Null string.

4. Calculate Time and Space complexities of functions fnExpressionValidate(), fnInfixToPostfix() and fnEvaluatePostfix().
5. Analyze Time and Space complexities for function **fnDecimalToAnyBase** in extra lab 3 in terms of value and base.

Files to be submitted:

Put the following files (in a zip archive or folder), name it on your roll number and upload at \\pucit-waqar\assignments. We will compile and run your code to make sure it works as required.

The files to be submitted are:

```
1. Stack.cpp
   Contains the definition and driver program for CStack
   class of bridge exercise
2. ExpValid.cpp
   Contains the driver program and function definition for
   Lab exercise 1
3. InToPost.cpp
   Contains the driver program and function definition for
   Lab exercise 2
4. EvalPost.h
   Contains the driver program and function definition for
   Lab exercise 3
5. ExtraLab.cpp
   This file contains the extra lab exercises 1, 2 and 3
6. ExtraLab.doc
   This file should contains solution for extra lab 4 and 5.
```

3.5. Lab 05: Implantation of Recursion and Queue ADT

Objectives:

- Implementations of Queue ADT —based on an array representation.
- Implementations of Recursion and its simulation.
- Simulation of Job scheduling in Operating System

Overview:

Recursion:

An essential ingredient of recursion is there must be a "termination condition"; i.e. the call to oneself must be conditional to some test or predicate condition, which will cease the recursive calling. A recursive program must cease the recursion o n some condition or be in a circular state, i.e. an endless loop that will "crash" the system.

In a computer implementation of a recursive algorithm, a function will conditionally call itself. When any function call is performed, a new complete copy of the function's "information" (parameters, return addresses, etc..) is placed into general data and/or stack memory. When a function returns or exits, this information is returned to the free memory pool and the function ceases to actively exist. In recursive algorithms, many levels of function calls can be initiated resulting in many copies of the function being currently active and copies of the different functions information residing in the memory spaces. Thus recursion provides no savings in storage nor will it be faster than a good non-recursive implementation. However, recursive code will often be more compact, easier to design, develop, implement, integrate, test, and debug.

The Tower of Hanoi (sometimes referred to as the Tower of Brahma or the End of the World Puzzle) was invented by the French mathematician, Edouard Lucas, in 1883. He was inspired by a legend that tells of a Hindu temple where the pyramid puzzle might have been used for the mental discipline of young priests. In this lab you will implement and simulate this puzzle

Queue:

Queue is an example of a restricted linear data structure that follows the principle of First In First Out (FIFO).

Queue can be considered as a container object whose both faces are opened. User can only insert items on one side and remove from other. That’s way firstly inserted item in the queue will be removed first. In a queue, the data items are ordered from most recently added (the **front**) to least recently added (the **rear**) this property can be used for developing several algorithms that have a wide variety of applications particularly in operating system, system scheduling.

One of the very important applications of queue is process scheduling for operating system. Jobs are handled in the order; they are entered in the queue, if there is no priority used. In this way small processes has to suffer a lot of time in waiting for their execution. To avoid this different techniques are used one of these techniques is round robin scheduling. In this lab we will explore and implement this method for process scheduling.

Array Representations of Queue:

Representation 1 (brute force): use an array of length MaxDegree+1 to store items.

```
private:
        int size;
        int front;
        int rear;
        int data[NumbersOfItems];
```

Problem – every queue object will have fixed number of Items.

Representation 2**:** dynamically allocate space for array once Size is known.

```
private:
        int size;
        int front;
        int rear;
        int *data;
```

Problem – only creates integer Queue.

Representation 3**:** dynamically allocate space for array once Size is known using templates.

```
template <class QueueType>
CQueue      {
private:
        int size;
        int front;
        int rear;
        QueueType *data;
        ……
};
```

Problem – creates problem if number of items to be inserted are not known at the time of Queue definition.

Logic Implementation**:**

All above representations can be implemented in linear as well as circular logic as follows

1. Linear Queue
    a. With continuous refresh
    b. With out continuous refresh
2. Circular Queue

***Representation 4*:** Use linked lists – coming soon …

Public functions:

The Queue ADT or class has the following public functions

1. **Constructor (int size = DEFAULT)**
   Construct queue with user provided size
2. **Insert (QueueType Element)**
   Places the given Element the front of the queue. Generates error of overflow is queue is already full.
3. **QueueType Remove()**
   Returns firstly inserted element and discards it from queue. Generates error of underflow is queue is empty.
4. **Boolean IsFull ()**
   Returns TRUE if queue is full FALSE otherwise.
5. **Boolean IsEmpty ()**
   Returns TRUE if queue is empty FALSE otherwise.
6. **Copy constructor**
   Takes care of a deep copy of the original representation.
7. **Destructor**
   Takes care of de-allocating memory to the system.

Pre-Lab Exercise:

Study article 3.3 (p. 131), Example 3.2 (p. 133), and review Exercises (p. 136) from your textbook.

Bridge Lab 1:

Tower of Hanoi Puzzle:

Problem Statement:

The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883. We are given a tower of eight disks (The picture below just shows four disks for the interest of space), initially stacked in decreasing size on one of three pegs. The objective is to transfer the entire tower to one of the other pegs (the third one in the picture below), moving only one disk at a time and never a larger one onto a smaller.

Let's call these 8 disks 1, 2, 3, 4, …, to 8, 8 being the largest disk and 1 being the smallest. Let's call the pegs A, B, C. Design an algorithm to produce one solution for these 8 disks. Then output the sequence of disk movement. For instance, the correct movement sequence for 3 disks should be:

```
move disk 1 from peg A to peg C
move disk 2 from peg A to peg B
move disk 1 from peg C to peg B
move disk 3 from peg A to peg C
move disk 1 from peg B to peg A
move disk 2 from peg B to peg C
move disk 1 from peg A to peg C
```

Hints: Recursive solution

Let's call the three pegs Source, Aux (Auxiliary) and Destination. However one solves the problem, sooner or later the bottom disks will have to be moved from Source to Destination. At this point in time all the remaining disks will have to be on Aux. After moving the bottom disk from Source to Destination these disks will have to be moved from Aux to Destination. Therefore, for a given number N of disks, the problem appears to be solved if we know how to accomplish the following tasks:

Move the top N-1 disks from Source to Aux (using Destination as an intermediary peg)
Move the bottom disk from Source to Destination
Move N-1 disks from Aux to Destination (using Source as an intermediary peg)

Write a function that displays all disk movements at the screen in following format

```
move disk n from peg Source to peg Destination
```

Bridge Lab 2:

Implement a class CQueue for representation 3 from array representation section using circular logic.

```
template <class QueueType>
class CQueue{
public:
CQueue (int MaxNoOfElements = DEFAULT);
CQueue (const CQueue &);
Insert (QueueType);
Boolean IsFull();
Boolean IsEmpty();
QueueType Remove();
        ~ CQueue ();
```

```
    private:
                    int size;
                    int front;
                    int rear;
                    QueueType *data;
            };
```

Driver Program 1: Driver Program for CQueue class is as follows

```
int main(void) {

CQueue <int> IntQueue (5);
CQueue <char > CharQueue (3);
CQueue <char *> StringQueue (5);

cout  << IntQueue.Remove() <<  endl ;
for (int i=0; i<5; i++)
        IntQueue.Insert(i);

for (i=0; i<6; i++)
        cout << IntQueue.Remove() << endl;

cout  << IntQueue.Remove() <<  endl ;

CharQueue.Insert('P');
CharQueue.Insert('O');
CharQueue.Insert('P');

cout << CharQueue.Remove() <<
CharQueue.Remove()<<CharQueue.Remove()<< endl;
StringQueue.Insert("to be");
StringQueue.Insert("or");
StringQueue.Insert("not");
StringQueue.Insert("to be");

cout  << StringQueue.Remove() <<  endl << StringQueue.Remove()<<  endl ;
cout  << StringQueue.Remove() <<  endl << StringQueue.Remove()<<  endl;
return 0;
}
```

Write the output of above programs.

Bridge Lab 3:

Implement array based double-ended queue for linear without continuous refresh logic. CDQueue ADT expressed as follows.

```
template <class CDQueueType>
class CDQueue{
public:
CDQueue (int MaxNoOfElements = DEFAULT);
CDQueue (const CDQueue &);
InsertLeft (CDQueueType);
InsertRight (CDQueueType);
CDQueueType RemoveLeft ();
CDQueueType RemoveRight ();
Boolean IsFull();
Boolean IsEmpty();
        ~ CDQueue ();
private:
                int size;
                int frontLeft;
                int rearLeft;
                int fronRight;
                int rearRight;
                QueueType *data;
        };
```

Driver Program 1: Driver Program for CQueue class is as follows

```
int main(void) {
CDQueue <int> IntDQueue (10);

cout  << IntDQueue.RemoveLeft() <<  endl ;
for (int i=0; i<5; i++)
        IntDQueue.InsertLeft(i);

for ( i = 5; i<0; i--)
        IntDQueue.InsertRight(i);

for (i=0; i<3; i++)  {
        cout << IntDQueue.RemoveLeft() << endl;
        cout << IntDQueue.RemoveRight() << endl;
}

for (int i=0; i<3; i++)
        IntDQueue.InsertLeft(i);
for ( i = 2; i<0; i--)
        IntDQueue.InsertRight(i);
cout  << IntDQueue.RemoveLeft() <<  endl ;

for (i=0; i<2; i++)  {
```

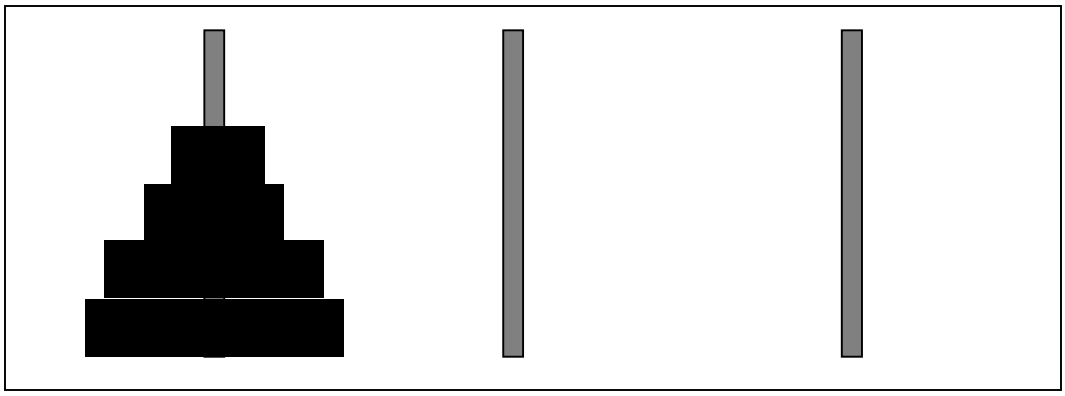
```
            cout << IntDQueue.RemoveLeft() << endl;
            cout << IntDQueue.RemoveRight() << endl;
    }

    cout  << IntDQueue.RemoveLeft()   <<  endl ;
    cout  << IntDQueue.RemoveRight() <<  endl;
    cout  << IntDQueue.RemoveLeft()   <<  endl ;
    return 0;
    }
```

Write the output of above programs.

In-Lab Exercise 1:

Write a function that simulates the disk movements of Bridge Lab 1 at the screen in following format for at least 7 numbers of disks.



In-Lab Exercise 2:

In this exercise, you will write a C++ program to simulate Round Robin Scheduling Technique for process management in Operating System.

It is one of the oldest, simplest, fairest and most widely used scheduling algorithms, designed especially for time-sharing systems. A small unit of time, called time-slice or **quantum**, is defined. All runnable processes are kept in a circular queue. The CPU scheduler goes around this queue, allocating the CPU to each process for a time interval of one quantum. New processes are added to the tail of the queue.

The CPU scheduler picks the first process from the queue, sets a timer to interrupt after one quantum, and dispatches the process.

If the process is still running at the end of the quantum, the CPU is preempted (blocked) process is added to the tail of the queue. If the process finishes before the end of the quantum, the process itself releases the CPU voluntarily. In either case, the CPU scheduler assigns the CPU to the next process in the ready queue. Every time a process is

granted the CPU, a **context switch** occurs, which adds overhead to the process execution time.

```
template  <class ProcessType >
struct Process {
                    ProcessType *data;
                    int ExecutionTime;
};

template <class ProcessType >
class CRRQueue{
public:
            CRRQueue (int MaxNoOfProcesses = DEFAULT);
            CRRQueue (const CRRQueue &);
            AddProcess (ProcessType, int ExecutionTime);
            SetQuantum (int Quantum = DEFAULT_ QUANTUM);
            Update (int Time);
            ProcessType RemoveProcess();
            Boolean IsFull();
            Boolean IsEmpty();
            ostream & operator << (const ostream &,  const CRRQueue &);
            ~ CRRQueue ();
private:
            int size;
            int front;
            int Quantum;
            int rear;
            Process <ProcessType> *data;
    };
```

Driver Program 1: Driver Program for *CRRQueue* class that simulates the process scheduling in CPU.

```
int main(void) {

CRRQueue <int> ProcessQueue (10);

ProcessQueue.SetQuantum (3);

for (int i=1; i<= 5; i++)
        ProcessQueue.InsertProcess ( i, random(10));

for ( i = 0; i<=5; i--)    {
        ProcessQueue.Update (3);
        cout  << ProcessQueue;
}
```

Write the output of above programs.

Extra Lab:

1. Calculate Time and Space complexities of Tower of Hanoi function.
2. Convert recursive Tower of Hanoi puzzle into an equivalent iterative logic.
3. Write recursive procedures for binary search and merge sort.
4. Extend the CQueue ADT to convert into CDQueue class using circular logic.
5. Extend the CQueue ADT to convert into CMultiQueue class for managing N number of queues in a single array.
6. Make the above extension for CStack ADT as well
7. Write the ADT for HybridMultiStackQueue data structure as discussed in the class.

Files to be submitted:

Put the following files (in a zip archive or folder), name it on your roll number and upload at \\pucit-waqar\assignments. We will compile and run your code to make sure it works as required.

The files to be submitted are:

1. TOH.cpp
   Contains code and driver program for Tower of Hanoi puzzle of bridge exercise 1
2. Queue.cpp
   Contains the definition and driver program for CQueue class for bridge exercise 2
3. DQueue.cpp
   Contains the definition and driver program for CDQueue class for bridge exercise 3
4. STOH.cpp
   Contains code and driver program for the simulation of the Tower of Hanoi puzzle for in-lab exercise 1.
5. Process.cpp
   Contains code and driver program for the simulation of the Round Robin scheduling of operating system process management for in-lab exercise 2.
6. ExtraLab.cpp
   This file contains the extra lab exercises 1, 2 and 3 etc.
7. ExtraLab.doc
   This file should contain solution for extra lab 1.

3.6. Lab 06: Implementation of Linear Linked List ADT

Objectives:

- Implementations of the Linear Linked List ADT.
- Use templates to produce a generic Linear Linked List data structure.
- Implement Iterator for Linear Linked List container.
- To write different generic algorithms for Linear Linked List container.

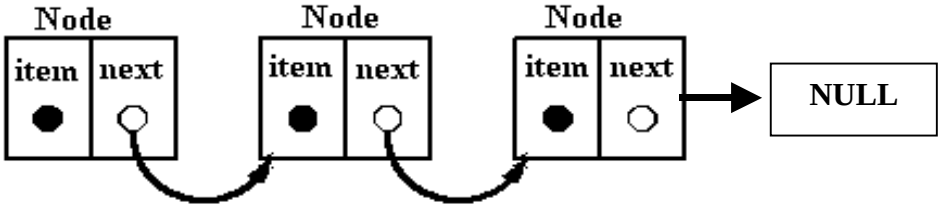
Overview:

The linked list is a very flexible **dynamic data structure.** Items may be added to it or deleted from it at will. A programmer need not worry about how many items a program will have to accommodate: this allows us to write robust programs which require much less maintenance. A very common source of problems in program maintenance is the need to increase the capacity of a program to handle larger collections: even the most generous allowance for growth tends to prove inadequate over time!

In a linked list, each item is allocated space as it is added to the list. A link is kept with each item to the next item in the list.

Singly linked List:

The singly linked list is the most basic of all the pointer based data structures. A singly linked list is simply a sequence of dynamically allocated storage elements, each containing a pointer to its successor. The figure below gives a simple illustration.



Each node of the list has two elements

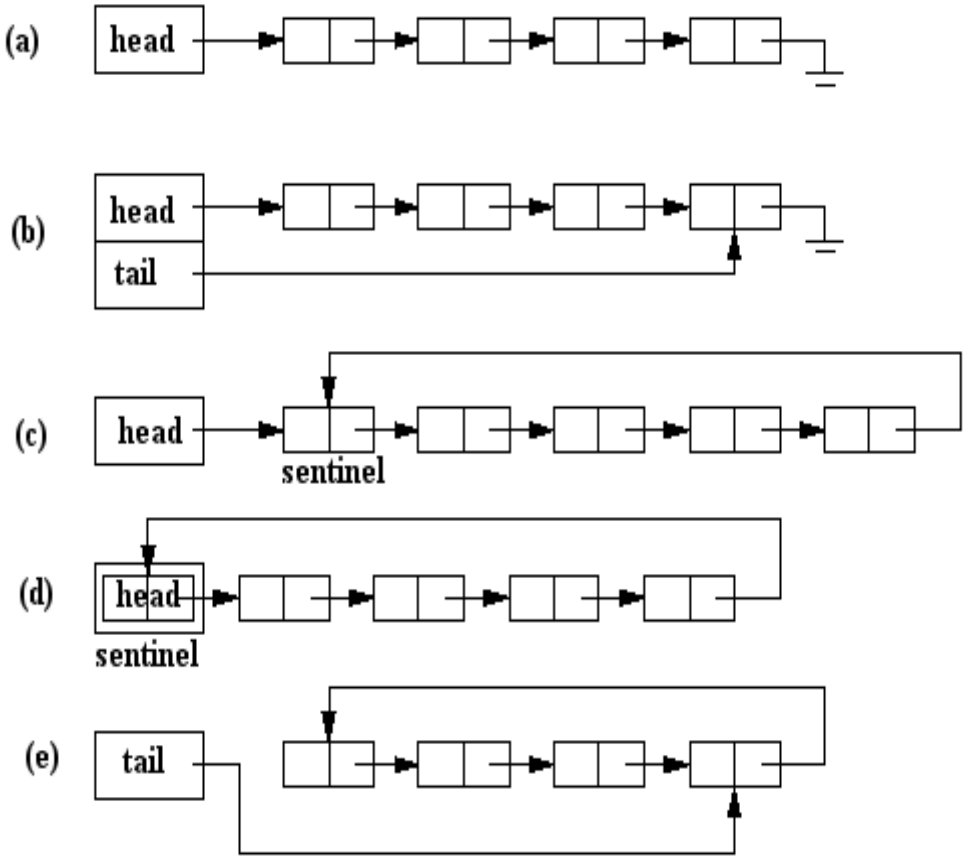
1. the item being stored in the list *and*
2. a pointer to the next item in the list

The last node in the list contains a NULL pointer to indicate that it is the end or *tail* of the list.

As items are added to a list, memory for a node is dynamically allocated. Thus the number of items that may be added to a list is limited only by the amount of memory available.

Representations of Singly Linked List:

In spite of its simplicity, there are many variations of its implementation. The following figures show several of the most common singly linked list variants.



Representation A:

The basic singly-linked list is shown in Figure (a). Each element of the list contains a pointer to its successor; the last element contains a null pointer. A pointer to the first element of the list, labeled **`head`** in Figure (a), is used to keep track of the list.

The basic singly-linked list is inefficient in those cases when we wish to add elements to both ends of the list. While it is easy to add elements at the head of the list, to add elements at the other end (the *tail*) we need to locate the last element. If the basic basic singly-linked list is used, the entire list needs to be traversed in order to find its tail.

Representation B:

Figure (b) shows a way in which to make adding elements to the tail of a list more efficient. The solution is to keep a second pointer, **`tail`**, which points to the last element

of the list. Of course, this time efficiency comes at the cost of the additional space used to store the **tail** pointer.

Representation C and D:

The singly linked lists labeled (c) and (d) illustrate two common programming tricks. The list (c) has an extra element at the head of the list called a ***sentinel***. This element is never used to hold data and it is always present. The principal advantage of using a sentinel is that it simplifies the programming of certain operations. E.g., since there is always a sentinel standing guard, we never need to modify the `head` pointer. Of course, the disadvantage of a sentinel such as that shown in (c) is that extra space is required, and the sentinel needs to be created when the list is initialized.

The list (c) is also a ***circular list***. Instead of using a null pointer to demarcate the end of the list, the pointer in the last element points to the sentinel. The advantage of this programming trick is that insertion at the head of the list, insertion at the tail of the list, and insertion at an arbitrary position of the list are all identical operations.

Figure (d) shows a variation of a singly linked list using a sentinel in which instead of keeping a pointer to the sentinel, the sentinel itself serves as the handle for the list. This variant eliminates the need to allocate storage for the sentinel separately.

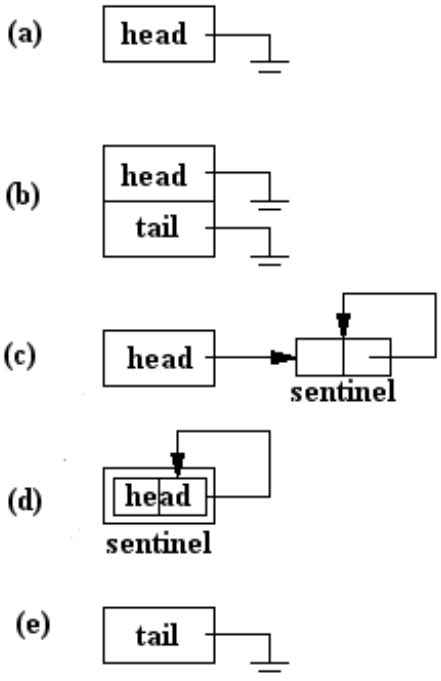


Figure: Empty Singly Linked Lists

Representation E:

Of course, it is also possible to make a circular, singly linked list that does not use a sentinel. Figure (e) shows a variation in which a single pointer is used to keep track of the list, but this time the pointer, **`tail`**, points to the last element of the list. Since the list is circular in this case, the first element follows the last element of the list. Therefore, it is relatively simple to insert both at the head and at the tail of this list. This variation minimizes the storage required, at the expense of a little extra time for certain operations. The figures below illustrate how the empty list (i.e., the list containing no list elements) is represented for each of the variations given in the figures above. Notice that the sentinel is always present in those list variants, which use it. On the other hand, in the list variants, which do not use a sentinel, null pointers are used to indicate the empty list.

The list variant (b) introduces a potential source of very subtle programming errors. A conservative programmer would insist that ***both*** the `head` and `tail` pointers must be null. However, a clever programmer might realize that it is sufficient require only that the `head` pointer be null in the case of an empty list. Since, the `tail` pointer is not used when the list is empty, its value may be left undefined. Of course, if that is the case, extreme care must be taken to ensure that the `tail` pointer is never used when the `head` pointer is null.

Linked List ADT

The variable (or handle), which represents the list, is simply a pointer to the node at the ***head*** of the list.

The fundamental functions (methods) associated to a list ADT are:

1. Insert a node at the head
2. Insert a node at the tail
3. Insert a node before a particular node
4. Insert a node after a particular node
5. Change the contents of a node with new value
6. Delete a particular node
7. Delete a node from the head
8. Delete a node from the tail
9. Delete a node before a particular node
10. Delete a node after a particular node
11. Test for empty list
12. Output contents of list.

Adding to a list:

The simplest strategy for adding an item to a list is at its head is:

(a) Allocate space for a new node,
(b) Copy the item into it,
(c) Make the new node's `next` pointer point to the current head of the list ***and***

(d) Make the head of the list point to the newly allocated node.

This strategy is fast and efficient, but each item is added to the head of the list.

Pre-Lab Exercise:

Study article 4.1 (p. 161), 4.2 (p. 165), 4.3 (p. 175), and review Exercises (p. 173, p. 183) from your textbook.

Bridge Exercise 1:

Implement a class CLSLL (Linear Single Linked List) for representation A from linked list representation section.

```
template <class Type>
class Node {
    friend class CLSLL < Type >;
    Type Record;
    Node < Type > * Next;
};

template <class Type>
class CLSLL {
public:
        CLSLL ();
        CLSLL (const CLSLL < Type > &);
        AddAtHead (Type Info);
        AddAtTail (Type Info);
        AddBefore (Type ID, Type Info);
        AddAfter (Type ID, Type Info);
        Type DeleteAtHead ();
        Type DeleteAtTail ();
        Type Delete (Type ID);
        Type DeleteBefore (Type ID);
        Type DeleteAfter (Type ID);
        EditNode (Type PreviousInfo, Type NewInfo)
        Boolean Find (Type ID);
        Sort ();
        CLSLL operator = (const CLSLL &);
        friend ostream & operator << ();
        Clear ();
        ~ CLSLL ();

private:
        Node < Type >*Head;
    };
```

Driver Program:

Driver Program for CLSLL class is as follows

```
int main(void) {
CLSLL <int> First;
CLSLL <int> Second;

First.AddAtHead(5);
First.AddAtHead(15);
First.AddAtTail (25);
First.AddBefore (125, 12);
First.AddAfter (5, 7);
First.DeleteAtTail ();
First.DeleteAtHead ();
First.AddAtHead(13);
First.AddAtTail (29);
First.AddBefore (5, 9);
First.Delete (5);

Second = First;
Second.Sort();

cout  << "Unordered Values are " << endl;
cout  << First  << endl;
cout  << "Ordered Values after sorting are" << endl;
cout  << Second << endl;

int Value;
cin >> Value;

if (First.Find(Value))
        cout  << "Value " << Value << " is in the list" ;
else
        cout  << "Value " << Value << " is not in the list" ;

return 0;
}
```

Write the output of above program.

Bridge Exercise 2:

Implement the Forward Iterator for linear singly linked list class.

```
template <class Type>
class CLSLLIterator {
        friend class Node < Type >;
        friend class CLSLL < Type >;
public:
        CLSLLIterator (const CLSLL < Type > &);
        CLSLLIterator (const CLSLLIterator < Type > &);
        Type * Get ();
        Set (Type);
        Begin ();
        Next ();
        End ();
        CLSLLIterator operator = (CLSLLIterator);
        AssociateList(const CLSLL < Type > &);
private:
        Node < Type > * Current;
        CLSLL < Type > * AssociatedList;
    };
```

Driver Program:

Driver Program for CLSLL class is as follows

```
int main(void) {
CLSLL <int> First;
CLSLLIterator <int> I (First);

First.AddAtHead(5);
First.AddAtHead(15);
First.AddAtTail (25);
First.AddBefore (125, 12);
First.AddAfter (5, 7);
First.DeleteAtTail ();
First.DeleteAtHead ();
First.AddAtHead(13);
First.AddAtTail (29);
First.AddBefore (5, 9);
First.Delete (5);

Second = First;
Second.Sort();

cout  << "Unordered Values are " << endl;
for (I.Begin(); ! I.End(); I.Next())
        cout  << (*I.Get()) << endl;
```

```
I.AssociateList(Second);

cout  << "Ordered Values after sorting are" << endl;

for (I.Begin(); ! I.End(); I.Next())
        cout  << (*I.Get()) << endl;

return 0;
}
```

Write the output of above program.

In-Lab Exercise 1:

Overload operations + and – for CLSLL class,

CLSLL operator + (const CLSLL &);

C = A + B C will contain all elements of A and B without duplication
A = (a_1, a_2, a_3, … a_m), B = (b_1, b_2, b_3, … b_n)
C = (a_1, b_1, a_2, b_2, a_3, b_3, … , a_m, … , b_n),

CLSLL operator – (const CLSLL &);

C = A – B C will contain all elements of A that does not belong to B
A = {a_1, a_2, a_3, … a_m} B = {b_1, b_2, b_3, … b_n}
C = {$\forall$ a_i | $a_i \in$ A and $\notin$ B}

Add following statements at the end of driver programs,

```
cout << "merge of both lists is  = " << First  + Second  << endl;
cout << "Difference of both lists is = " << First  – Second  << endl;
```

Write the output of above programs.

In-Lab Exercise 2:

Write the following generic algorithms for linear singly linked list class for bridge labs.

```
template <class Type>
Boolean Search (const CLSLL <Type> & Conatiner, Type SearchKey)

template <class Type>
Boolean Sort  (CLSLL <Type> & Conatiner, Type SortKey)

template <class Type>
```

Type Max (const CLSLL <Type> & Conatiner)

template <class Type>
Type Min (const CLSLL <Type> & Conatiner)

template <class Type>
Type Average (const CLSLL <Type> & Conatiner)

Extra Lab:

Implement CPolynomial class with all its functions using linear singly linked class defined in bridge lab exercise 1.

Implement CSparsePolynomial class with all its functions using linear singly linked class defined in bridge lab exercise 1.

Overload +, - and * operators for above two classes

Files to be submitted:

Put the following files in a zip archive/folder name it on your roll number and upload at \\pucit-waqar\assignments. We will compile and run your code to make sure it works as required.

The files to be submitted are:

1. LSLLmain.cpp
   Contains the driver program for CLSLL class of bridge lab-1
2. TLSLLmain.cpp
   Contains the driver program CLSLL class of bridge lab-2
3. CLSLL.h
   Contains the class definition and implementation for CLSLL class using template.
4. InLab1.cpp
   Contains the definition and implementation for in-lab exercise 1.
5. InLab1.cpp.
   Contains the definition and implementation for in-lab exercise 2.
6. Extra.Cpp
   Contains the definition and implementation for extra-lab exercise.

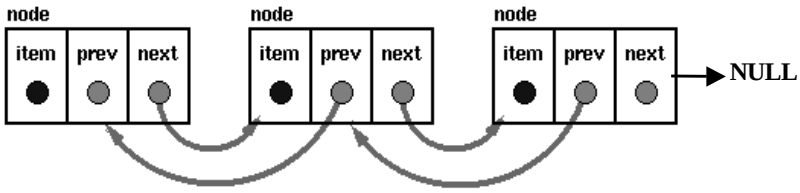
3.7. Lab 07 - Implementation of Doubly Linked List ADT and MultiStack ADT

Objectives:

- Implementations of the Doubly Linked List ADT.
- Implement of Iterator for Doubly Linked List container.
- Implementation of MultiStack and MultiQueue.

Overview:

They permit scanning or searching of the list in both directions. (To go backwards in a simple list, it is necessary to go back to the start and scan forwards.) Many applications require searching backwards and forwards through sections of a list so the backward links become very useful. In this case, the node structure is altered to have two links:



Doubly linked lists have a pointer to the preceding item as well as one to the next.

```
Template <class Type>
class DNode {
    Type item;
    DNode <Type> *previous;
    DNode <Type> *next;
    };
```

Representations of Doubly Linked List:

In a *doubly-linked list* , each list element contains two pointers - one to its successor and one to its predecessor. There are many different variations of doubly-linked lists: The figures below illustrate four of them.

Figure (a) shows the simplest case: Two pointers, say *head* and *tail*, are used to keep track of the list elements. One of them points to the first element of the list, the other points to the last. The first element of the list has no predecessor, therefore that pointer is null. Similarly, the last element has no successor and the corresponding pointer is also null. In effect, we have two overlapping singly-linked lists which go in opposite directions. The figure also shows the representation of an empty list. In this case the head and tail pointers are both null.

Figure (b) shows a case which uses sentinels. In this variation *two* sentinels are used! Because there are in effect two overlapping linked lists that go in opposite directions, two sentinels are used - one for each singly-linked list. Recall, the use of a sentinel is motivated by the fact that the code for insertion and deletion is often simpler to write because there are fewer special cases to consider. Figure (b) shows that in the empty list, the two sentinels point to each other.

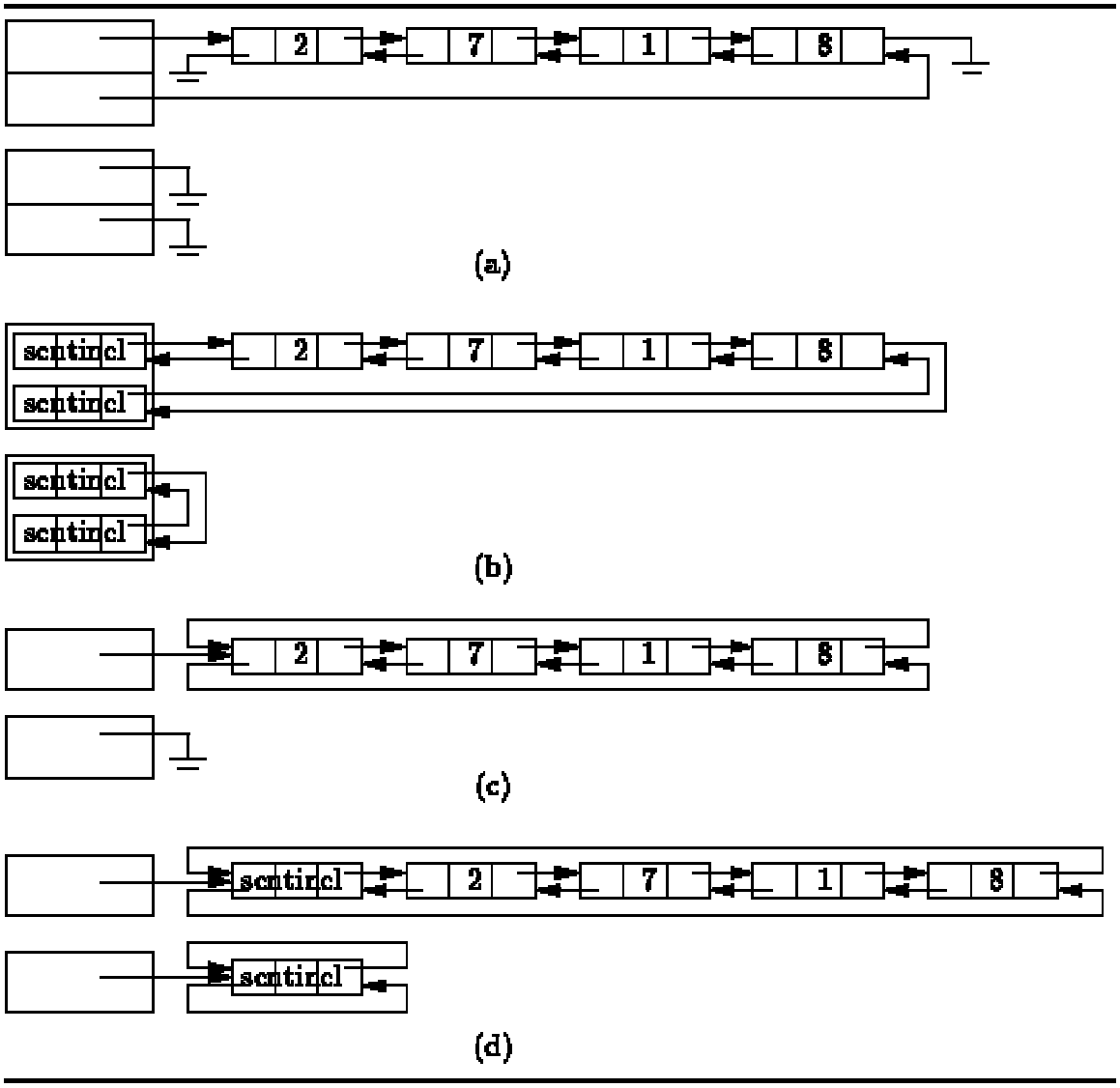


Figure: Doubly-Linked and Circular List Variations

A *circular, doubly-linked list* is shown in Figure (c). A circular list is formed by making use of pointers which would otherwise be null. The last element of the list is made the predecessor of the first element; the first element, the successor of the last. The upshot is that we no longer need both a head and tail pointer to keep track of the list. Even if only a single pointer is used, both the first and the last list elements can be found in constant time.

Finally, Figure (d) shows a circular, doubly-linked list which has a single sentinel. This variation is similar to the preceding one in that both the first and the last list elements can be found in constant time. This variation has the advantage that no special cases are required when dealing with an empty list. The figure shows that the empty list is represented by a list with exactly one element - the sentinel. In the case of the empty list, the sentinel is both its own successor and predecessor. Since the sentinel is always present, and since it always has both a successor and a predecessor, the code for adding elements to the empty list is identical to that for adding elements to a non-empty list.

Linked List ADT

The variable (or handle), which represents the list, is simply a pointer to the node at the ***head*** of the list.

The fundamental functions (methods) associated to a list ADT are:

1. Insert a node at the head
2. Insert a node at the tail
3. Insert a node before a particular node
4. Insert a node after a particular node
5. Change the contents of a node with new value
6. Delete a particular node
7. Delete a node from the head
8. Delete a node from the tail
9. Delete a node before a particular node
10. Delete a node after a particular node
11. Test for empty list
12. Output contents of list.

Pre-Lab Exercise:

Study article 4.8 (p. 207), 4.9 (p. 217), 4.2 (p. 165), 4.3 (p. 175), and review Exercises (p. 214, p. 218) from your textbook.

Bridge Exercise 1:

Implement a class CLDLL (Linear Doubly Linked List) for representation A from linked list representation section.

```
template <class Type>
class DNode {
friend class CLDLL < Type >;
Type Record;
DNode < Type > * Next;
DNode < Type > * Previous;
};
```

```
template <class Type>
class CLDLL {
public:
        CLDLL ();
        CLDLL (const CLDLL < Type > &);
        AddAtHead (Type Info);
        AddAtTail (Type Info);
        AddBefore (Type ID, Type Info);
        AddAfter (Type ID, Type Info);
        Type DeleteAtHead ();
        Type DeleteAtTail ();
        Type Delete (Type ID);
        Type DeleteBefore (Type ID);
        Type DeleteAfter (Type ID);
        EditNode (Type PreviousInfo, Type NewInfo)
        Boolean Find (Type ID);
        Sort ();
        CLDLL operator = (const CLDLL &);
        friend ostream & operator << ();
        Clear ();
        ~ CLDLL ();
private:
        Node < Type >*Head;
    };
```

Driver Program:

Driver Program for CLDLL class is as follows

```
int main(void) {
CLDLL <int> First;
CLDLL <int> Second;

First.AddAtHead(5);
First.AddAtHead(15);
First.AddAtTail (25);
First.AddBefore (125, 12);
First.AddAfter (5, 7);
First.DeleteAtTail ();
First.DeleteAtHead ();
First.AddAtHead(13);
First.AddAtTail (29);
First.AddBefore (5, 9);
First.Delete (5);

Second = First;
```

```
Second.Sort();

cout  << "Unordered Values are " << endl;
cout  << First  << endl;
cout  << "Ordered Values after sorting are" << endl;
cout  << Second << endl;

int Value;
cin >> Value;

if (First.Find(Value))
        cout  << "Value " << Value << " is in the list" ;
else
        cout  << "Value " << Value << " is not in the list" ;

return 0;
}
```

Write the output of above program.

Bridge Exercise 2:

Implement the Forward Iterator for linear doubly linked list class. CDNode and CLSLL will grant friendship to this class with following statement,

```
friend class CLDLLIterator < Type >;

template <class Type>
class CLDLLIterator {
public:
        CLDLLIterator (const CLDLL < Type > &);
        CLDLLIterator (const CLDLLIterator < Type > &);
        Type * Get ();
        Set (Type);
        Begin ();
        Next ();
        Previous ();
        End ();
        IsBegin ();
        IsEnd ();
        CLDLLIterator operator = (CLDLLIterator);
        CLDLLIterator operator = (CLDLL);
        AssociateList(const CLDLL < Type > &);
private:
        DNode < Type > * Current;
        CLDLL < Type > * AssociatedList;
```

```
};
```

Driver Program:

Driver Program for CLDLL class is as follows

```
int main(void) {
CLDLL <int> First;
CLDLLIterator <int> I (First);

First.AddAtHead(5);
First.AddAtHead(15);
First.AddAtTail (25);
First.AddBefore (125, 12);
First.AddAfter (5, 7);
First.DeleteAtTail ();
First.DeleteAtHead ();
First.AddAtHead(13);
First.AddAtTail (29);
First.AddBefore (5, 9);
First.Delete (5);

Second = First;
Second.Sort();

cout  << "Unordered Values are " << endl;
for (I.Begin(); ! I.IsEnd(); I.Next())
        cout  << (*I.Get()) << endl;

for (I.End(); ! I.IsBegin(); I.Previous())
        cout  << (*I.Get()) << endl;

I.AssociateList(Second);

cout  << "Ordered Values after sorting are" << endl;

for (I.Begin(); ! I.End(); I.Next())
        cout  << (*I.Get()) << endl;

return 0;
}
```

Write the output of above program.

In-Lab Exercise 1:

Implement **MultiStack** ADT using linked list.

```
template <class Type>
struct StackNode {
Type Data;
StackNode <Type> * Next;
};

template <class Type>
struct  StackHeader {
StackNode <Type> * StackHead;
StackNode <Type> * NextStack;
int NoOfItems;
};

template <class Type>
class MultiSatck {
        StackHeader <Type> * MultiStackHead;
        int NoOfStacks;

public:
        MultiStack ();                          // creates empty multistack
        MultiStack (int NoOfStacks);            // creates multistack with NoOfStacks stack
        MultiStack (const & MultiStack);        // copy constructor
        AddStack();
        AddStackAtBeg();
        AddStackAtEnd();
        DeleteStack(StackID);
        Push (int StackID, Type Value);
        Type Pop (int StackID);
        Type SeeTop (int StackID);
        IsEmpty (int StackID);
        MultiStackIsEmpty ();
        };
```

In-Lab Exercise 2:

Implement **MultiQueue** ADT using linked list.

```
template <class Type>
struct QueueNode {
Type Data;
QueueNode <Type> * Next;
};
```

```
template <class Type>
struct  QueueHeader {
QueueNode <Type> * QueueHead;
QueueNode <Type> * NextQueue;
int NoOfItems;
};

template <class Type>
class MultiQueue {
QueueHeader <Type> * MultiQueueHead;
int NoOfQueue;

public:
MultiQueue ();                          // creates empty multiQueue
MultiQueue (int NoOfQueue);             // creates multiQueue with NoOfQueue Queue
MultiQueue (const & MultiQueue);   // copy constructor
AddQueue ();
AddQueueAtBeg();
AddQueueAtEnd();
DeleteQueue (int QueueID);
Push (int QueueID, Type Value);
Type Pop (int QueueID);
Type SeeTop (int QueueID);
IsEmpty (int QueueID);
MultiQueueIsEmpty ();
};
```

Extra Lab:

1. Implement Stack of Queues and Queue of Stacks using Linked list ADT.
2. Implement MultiStackQueueHybridDataStructure using Linked list.
3. Implement Polynomial ADT using Linked list.
4. Implement Sparse Matrix ADT using Linked list.

Files to be submitted:

Put the following files in a zip archive/folder name it on your roll number and upload at \\pucit-waqar\assignments. We will compile and run your code to make sure it works as required.

The files to be submitted are:

1. `LDLL.cpp`
   `Contains the driver program for CLDLL class of bridge lab-1`

2. ILDLL.h
   Contains the class definition and implementation of bridge lab-2.
3. InLab1.cpp
   Contains the definition and implementation for in-lab exercise 1.
4. InLab1.cpp.
   Contains the definition and implementation for in-lab exercise 2.
5. Extra.Cpp
   Contains the definition and implementation for extra-lab exercise.

3.8. Lab 08: Implementation of Binary Tree

Objectives:

- Implementations of the binary tree ADT.
- Use templates to produce a generic binary tree data structure.
- Implement Iterator for binary tree container.
- To write different generic algorithms for binary tree container.

Overview:

Binary Tree

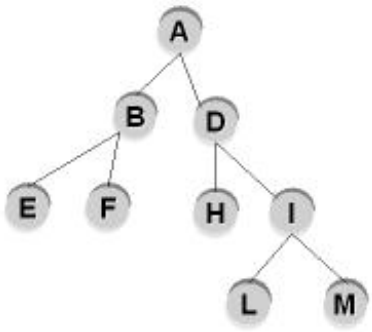
A binary tree is made up of a finite set of nodes that is either empty or consists of a node called the root together with two binary trees, called the left and right subtrees, which are disjoint from each other and from the root, i.e. every node in the tree can have at most two children, the tree is called binary tree.

Full Binary Trees

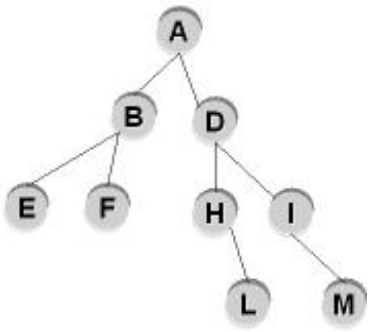
A binary tree where each node is either a leaf or is an internal node with exactly two non-empty children. That means, a node is allowed to have either none or two children (but not one!)

Complete Binary Trees

A binary tree whereby if the height is d, and all levels, except possibly level d, are completely full. If the bottom level is incomplete, then it has all nodes to the left side. That is the tree has been filled in the level order from left to right.



Complete Binary Tree



Not Complete Binary Tree

Binary Tree ADT

The variable (or handle), which represents the list, is simply a pointer to the node at the ***head*** of the list.

The fundamental functions (methods) associated to a list ADT are:

1. Insert a node.
2. Change the contents of a node with new value.
3. Delete a particular node.
4. Test for empty Tree
5. Output contents of Tree.

Pre-Lab Exercise:

Study article 5.1 (p. 246), 5.2 (p. 252), 5.3 (p. 261), 5.4 (p. 271), and review Exercises (p. 251, p. 260, p. 269, p. 276) from your textbook.

Bridge-Lab Exercise:

Implement a class CBT (binary tree) for

```
template <class Type>
class Node {
        friend class CBT <Type>;
        Type Record;
        Node <Type> * left;
        Node <Type> * right;
};

template <class Type>
class CBT {
        public:
        CBT ();
        CBT (const CBT < Type > &);
        Add (Type ID);
        Type Delete (Type ID);
        Edit (Type PreviousInfo, Type NewInfo);
        Boolean Find (Type ID);
        void PreTraverse();
        void InTraverse();
        void PostTraverse();
        void PreTraverseStack ();
        void TraverseQueue();
        CBT operator = (const CBT &);
        friend ostream & operator << ();
        ~ CBT ();
        private:
```

```
        Node < Type >*Head;
};
```

After the functions of add, delete and edit the tree should remain a BST. (You may need to implement a utility function that makes this property satisfy.)

Driver Program:

Driver Program for CBT class is as follows

```
int main(void) {
        CBT <int> First;
        CBT <int> Second;
        First.Add(5);
        First.Add(15);
        First.Add (25);
        First.Edit (125, 12);
        First.Add (7);
        First.Delete (25);
        First.Delete (7);
        Second = First;
        cout << First << endl;
        cout << Second << endl;
        int Value;
        cin >> Value;

        if (First.Find(Value))
                cout << ” Value „ << Value << ” is in the list„ ;
        else
                cout << ” Value „ << Value << ” is not in the list„ ;
        return 0;
}
```

Write the output of above program.

Lab Exercise 1:

Implement the Forward Iterator for linear singly linked list class.

```
template <class Type>

class CBTIterator {
        friend class Node <Type>;
        friend class CBT <Type>;
        public:
        CBTIterator (const CBT < Type > &);
```

```
        CBTIterator (const CBTIterator < Type > &);
        Type * Get ();
        Set (Type);
        Begin ();
        Next ();
        Bool End ();
        CBTIterator operator = (CBTIterator);
        Associate(const CBT < Type > &);
        private:
        Node <Type> * Current;
        CBT <Type> * Associated;
};
```

Driver Program:

Driver Program for CBTIterator class is as follows

```
int main(void) {
        CBT <int> First;
        CBT <int> Second;
        First.Add(5);
        First.Add(15);
        First.Add (25);
        First.Edit (125, 12);
        First.Add (7);
        First.Delete (25);
        First.Delete (7);
        Second = First;
        cout<<"the first tree elements are :"<<endl;
                for (I.Begin(); ! I.End(); I.Next())
                cout << (*I.Get()) << endl;
        I.AssociateList(Second);
        cout<<"the first tree elements are :"<<endl;
                for (I.Begin(); ! I.End(); I.Next())
                cout << (*I.Get()) << endl;
        return 0;
}
```

Write the output of above program.

Extra Lab:

1. Write class definition for binary in-threaded tree BITT.
2. Implement following function for BITT

    1. BITT()

2. BITT(const BITT &)
3. AddNode(eType id);
4. DeleteNode(eType id);
5. PreTraverse();
6. InTraverse();
7. PostTraverse();
8. = operator
9. << operator
10. ~BITT()

Files to be submitted:

Put the following files in a zip archive/folder name it on your roll number and upload at \\pucit-waqar\assignments. We will compile and run your code to make sure it works as required.

The files to be submitted are:

1. `CBT.cpp`
   `Contains the driver program for CBT class of bridge`
2. `Lab1.cpp`
   `Contains the definition and implementation for in-lab exercise 1.`
3. `Extra.Cpp`
   `Contains the definition and implementation for extra-lab exercise.`

3.9. Lab 9: Implementation of Binary Search Tree ADT

Objectives:

- Implementations of the binary search tree ADT.
- Use templates to produce a generic binary search tree data structure.
- Implement Iterator for binary tree container.
- To write different generic algorithms for binary search tree container.

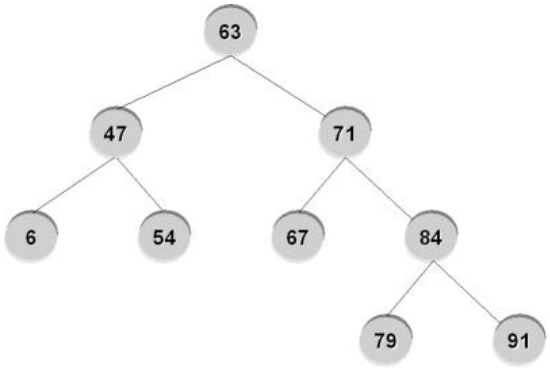
Overview:

Binary search tree, as the name specifies, is a tree with two nodes (left node and the right node) and with search functionality of binary search algorithm. Searching an element in a BST takes only O (log N) time complexity. The basic idea is to implement a tree whose left child nodes carry the elements that are less the root node (and also the parent node) and the right child nodes are greater than are equal to the root node and all parent nodes also. It makes it easier to search an element in the list because at each node, you are discarding (N log N) elements out of total N elements.

BST Property

- All elements stored in the left subtree of a node whose value is ***K*** have values less than ***K***. All elements stored in the right subtree of a node whose value is ***K*** have values greater than or equal to ***K***.
- That is, a node's left child must have a key less than its parent, and a node's right child must have a key greater or equal to its parent

BST – Example tree



Operations on BST ADT

- Create a BST
- Insert an element
- Remove an element

- Find an element
- Clear (remove all elements)
- Display all elements in a sorted order

Pre-Lab Exercise:

Study article 4.8 (p. 207), 4.9 (p. 217), 4.2 (p. 165), 4.3 (p. 175), and review Exercises (p. 214, p. 218) from your textbook.

Bridge Exercise:

Implement a class CBST (binary search tree class) Specification of BST ADT are as folllows

```
template <class eType>
CBST                    //Binary search tree
{
        private:
        //
        // BST Element(s)
        //
        public:
        CBST();                                 //Constructor
        ~CBST();                                //Destructor
        void clear();                           //Clear the tree
        Boolean insert(const eType val);            //Insert a node
        Boolean remove(const eType);                //Remove a node
        Boolean find(const eType) const;            //Find a node
        Boolean isEmpty() const;                    //True if BST is empty
        void print() const;                     //Print BST (inorder)
};
```

Pointer based implementation

```
template <class eType>
CBSTNode            // Binary tree node class
{
        private:
        eType element;                  // The node's value
        CBSTNode <eType >* left;        // Pointer to left child
        CBSTNode<eType >* right;        // Pointer to right child
         // Two constructors -- with and without initial values
        public:
        CBSTNode() { left = right = NULL; }
        CBSTNode(eType  val, BSTNode <eType >* l =NULL, BSTNode <eType > * r
        =NULL)    { element = val; left = l; right = r; }
```

```
        ~CBSTNode ( ) { }              // Destructor
        CBSTNode<eType >* leftchild ( ) const { return left; }
        CBSTNode<eType >* rightchild ( ) const { return right; }
        eType value ( ) const { return element; };
        void setValue(eType> val) { element = val; }
        Boolean isLeaf ( ) const          // Return TRUE if is a leaf
   };
```

Lab Exercise 1:

Implement a class CBST (binary search tree)

```
template <class Type>
class BSTNode {
        friend class CBST <Type>;
        Type Record;
        BSTNode <Type> * left;
        BSTNode <Type> * right;
};

template <class Type>
class CBST {
        public:
        CBST ();
        CBST (const CBST < Type > &);
        Add (Type ID);
        Type Delete (Type ID);
        Edit (Type PreviousInfo, Type NewInfo);
        Boolean Find (Type ID);
        void PreTraverse();
        void InTraverse();
        void PostTraverse();
        void PreTraverseStack ();
        void TraverseQueue();
        CBST operator = (const CBT &);
        friend ostream & operator << ();
        ~ CBST ();

        private:
        BSTNode < Type >*Head;
};
```

After the functions of add, delete and edit the tree should remain a BST. (You may need to implement a utility function that makes this property satisfy.)

Driver Program:

Driver Program for CBST class is as follows

```
int main(void) {
        CBST <int> First;
        CBST <int> Second;
        First.Add(5);
        First.Add(15);
        First.Add (25);
        First.Edit (125, 12);
        First.Add (7);
        First.Delete (25);
        First.Delete (7);
        Second = First;
        cout << First << endl;
        cout << Second << endl;
        int Value;
        cin >> Value;

        if (First.Find(Value))
                cout << ” Value „ << Value << ” is in the list„ ;
        else
                cout << ” Value „ << Value << ” is not in the list„ ;
        return 0;
}
```

Write the output of above program.

Lab Exercise 2:

Implement the Forward Iterator for linear singly linked list class.

```
template <class Type>

class CBSTIterator {
        friend class BSTNode <Type>;
        friend class CBST <Type>;
        public:
        CBSTIterator (const CBST < Type > &);
        CBSTIterator (const CBSTIterator < Type > &);
        Type * Get ();
        Set (Type);
        Begin ();
        Next ();
        Bool End ();
```

```
        CBSTIterator operator = (CBSTIterator);
        Associate(const CBST < Type > &);
        private:
        BSTNode <Type> * Current;
        CBST <Type> * Associated;
};
```

Driver Program:

Driver Program for CBTIterator class is as follows

```
int main(void) {
        CBST <int> First;
        CBST <int> Second;
        First.Add(5);
        First.Add(15);
        First.Add (25);
        First.Edit (125, 12);
        First.Add (7);
        First.Delete (25);
        First.Delete (7);
        Second = First;
        cout<<"the first tree elements are :"<<endl;
        for (I.Begin(); ! I.End(); I.Next())
        cout << (*I.Get()) << endl;
        I.AssociateList(Second);
        cout<<"the first tree elements are :"<<endl;
        for (I.Begin(); ! I.End(); I.Next())
        cout << (*I.Get()) << endl;
        return 0;
}
```

Write the output of above program.

Extra Lab:

3. Implement a class representation of the threaded trees with following methods

    1. AddNode(type id);
    2. DeleteNode(type id);
    3. PreTraverse();
    4. InTraverse();
    5. PostTraverse();

Files to be submitted:

Put the following files in a zip archive/folder name it on your roll number and upload at \\pucit-waqar\assignments. We will compile and run your code to make sure it works as required.

The files to be submitted are:

1. CBST.cpp
   Contains the driver program for CBST class of bridge
2. Lab1.cpp
   Contains the definition and implementation for in-lab exercise 1.
3. Extra.Cpp
   Contains the definition and implementation for extra-lab exercise.

3.10. Lab 10 – Implementation of Heap ADT

Objectives:

- Implementations Heap ADT

Heap

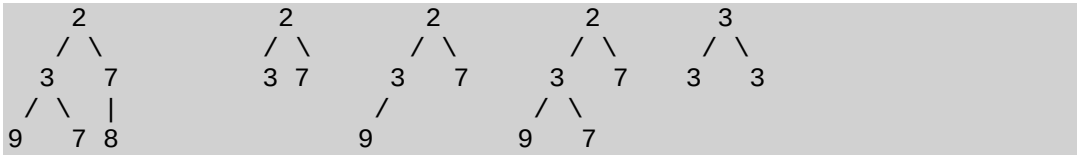
Heap is a complete binary tree in which from root to leaf the parent node has more priority than its child sub trees. This priority is based on some heap property as in max heap every parent node has larger key value than its children's key value while in min heap every parent node has smaller key value than its children's key value.

One of the reasons that heaps are so efficient is that they can be stored in arrays in a straightforward way. The items of the heap are stored in an array H, and the internal nodes of the tree are associated with the indices of the array by numbering them level-wise from the left to the right. Thus the root is associated with 1, the left child of the root with 2, the right child of the root with 3, the leftmost grandchild of the root with 4, etc. We leave the index 0 for our convenience.

1. Parent of node i has index |i/2| (root has index 1).
2. Left and right children of node i have indices 2i and 2i + 1 respectively.

A max heap is a complete binary tree in which each node has larger key value than the key values of its children.

Here are some heaps of varying sizes.



Insertion Into Max Heap

The followings steps involves while inserting into a max heap.

1. If the max heap is empty then simply place the element at the 1st location.
2. But if the max heap is not empty then follow the following steps.
3. Place the element, which is to be inserted into the next available location.
4. If the key value of the newly inserted element is smaller than the key value of its parent then it's all right.
5. If the key value of the newly inserted element is larger than the key value of its parent then swap the element with its parent, repeat the process until the key value of the parent does not becomes larger than the key value of its child.

Deletion From A Max Heap

The followings steps involves while deleting an element from a max heap.

1. Swap the last element with root element.
2. Remove the last element.
3. Swap the root element with its larger child element; repeat the process until the parent element becomes larger than its child element.

Complexity

The time complexity for insertion into max heap is O (log n).
The time complexity for deletion operation from a max heap is O (log n).
The time complexity for finding the maximum element is O (1) as the maximum element is placed at the root.

Pre-Lab Exercise:

Study article 6.2 (p. 345), 6.3 (p. 356), 6.4 (p. 364), and review Exercises (p. 354, p.363) from your textbook.

Bridge Lab Exercise:

Implement following CMaxHeap class.

```
template <class type>
calss CMaxHeap
{
        protected:
        Element <type> *data;
        int Size;
        int MaxSize;
        m_fnShiftUp();
        m_fnShiftDown();

        Public:
        CMaxHeap(const int );
        ~CMaxHeap( );
        void m_fnInsert( const Element <type>&);
        Element<type>* m_fnDeleteMax(Element<type>&);
        Element<type>* m_fnFindMax(Element<type>&);
        Boolean IsEmpty();
        Boolean IsFull();
};
```

In-Lab Exercise 1:

Implement class CMinHeap with all its functionalities.

Files to be submitted:

Put the following files (in a zip archive or folder), name it on your roll number and upload at \\pucit-waqar\assignments. We will compile and run your code to make sure it works as required.

The files to be submitted are:

```
1. Bridge.cpp
   Contains the function definition for bridge exercise.
2. Lab1.cpp
   Contains the function definition for Lab exercise 1
```

3.11. Lab 11 – AVL Tree ADT

Objectives:

- Implementations of the AVL Tree ADT — based on self-referential representation.
- Use templates to produce a generic AVL Tree data structure.
- Check brackets validity of mathematical expressions.
- Compare the searching performance of BST and AVL tree.

Overview:

Height Balanced Trees: AVL Trees

The concept of height balanced trees was presented in 1962 by two Russian mathematicians, G.M. Adelson-Velskii and E.M. Landis, and the resulting binary search tree are called ***AVL trees*** in their honor.

The behavior of an AVL tree closely approximates that of the ideal, completely balanced binary search tree.

Definition:

An AVL tree is a binary search tree in which the heights of the left and right subtrees of the root differ by at most 1 and in which the left and right subtrees are again AVL trees.

With each node of an AVL tree is associated a ***balance factor*** that is left higher , or right higher according, respectively , as the left subtree has height greater than, equal to, or less than that of the right subtree.

Thus balance factor of a binary tree (*bf*) is defined as

$$bf = \text{height of left subtree } (h_L) - \text{height of the right subtree } (h_R)$$

Where h = the number of nodes visited in traversing a branch which leads to a leaf node at the deepest level of the tree.

Therefore, a binary search tree is said to be height balanced binary search tree if all its nodes have a balance factor of 1, 0 or -1. That is below, for every node in the tree.

$$|bf| = |h_L| - |h_R| \leq 1,$$

It can be noted that a height balanced binary tree is always a binary search tree and a complete binary search tree is always height balanced, but the reverse may not be true.

Insertion in AVL:

We can insert a new node into an AVL tree by using the following steps:

1. Insert into a binary search tree: Insert the node into its proper position following the properties of binary search tree, comparing the key value of the new node with that in the root, and inserting the new node into the left or right subtree as appropriate.
2. Compute the balance factor: On the path starting from the root node to the node newly inserted, compute the balance factor of each node. It can be verified that change in balance factors will occur only in this path.
3. Decide the pivot node: On the path traced in step 2, determine whether the absolute value of any node's balance factor is switched from 1 to 2. if so, the tree becomes unbalanced. The node whose absolute value of balance factor is switched from 1 to 2, mark it as a special node called pivot node.
4. Balance the unbalanced tree: It is necessary to manipulate pointers centered at the pivot node to bring the tree back into height balance. This pointer manipulation is well known as AVL rotation.

AVL Rotations:

There are four cases of rotation possible, which is discussed as below:

Case 1

Unbalance occurred due to insertion in the left subtree of the left child of the pivot node. In this case, following manipulations in pointers take place:

- Right subtree (A_R) of the left child (A) of the pivot node (P) becomes the left subtree of P.
- P becomes the right child of A.
- Left subtree (A_L) of A remains the same.

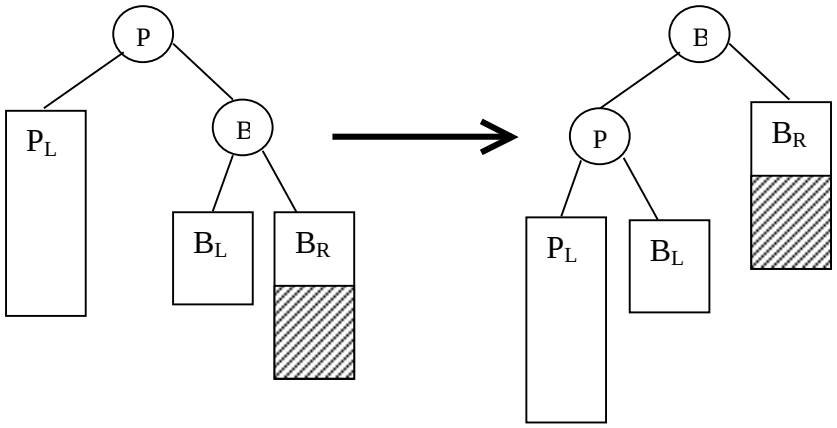
This is called LEFT-TO-LEFT insertion.

Case 2

Unbalance occurred due to the insertion in the right subtree of the right child of the pivot node. This case is reverse and symmetric to case 1. This rotation is illustrated in the following figures. In this case, the following manipulations in pointers take place:

- Left subtree (B_L) of right child (B) of pivot node (P) becomes the right subtree of P.
- P becomes the left child of B.
- Right subtree (B_R) of B remains same.

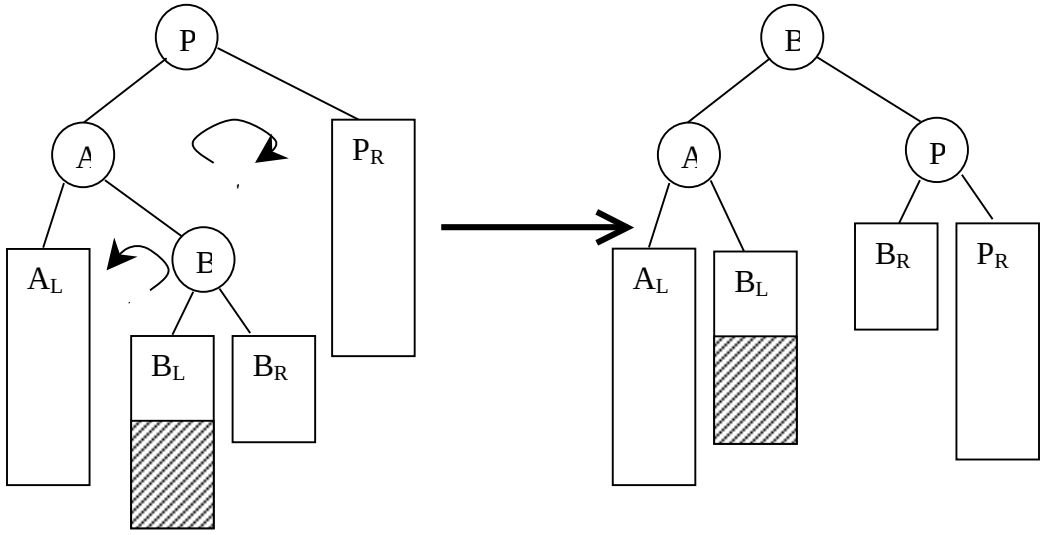
  This case is known as RIGHT-TO-RIGHT insertion.



Case 3

Unbalance occurred due to insertion in the right subtree of the left child of the pivot node. The figure given below illustrates the rotation required for this case of insertion. This case is known as LEFT-TO-RIGHT insertion.

Case 3 involves two rotations for the manipulation in pointers:



Rotation 1

- Left subtree (B_L) of the right child (B) of the left child of pivot node (P) becomes the right subtree of the left child (A).
- Left child (A) of the pivot node(P) becomes the left child of B.

Rotation 2

- Right subtree (B_R) of the right child (B) of the left child (A) of the pivot node (P) becomes the left subtree of P.
- P becomes the right child of B.

This case is known as LEFT-TO-RIGHT insertion.

Case 4

Unbalance occurred due to the insertion in the left subtree of right child of the pivot node. This case is symmetric to case 3.

This case is known as RIGHT-TO-LEFT insertion. There are two rotations for the manipulations of pointers in this case, these are:

Rotation 1

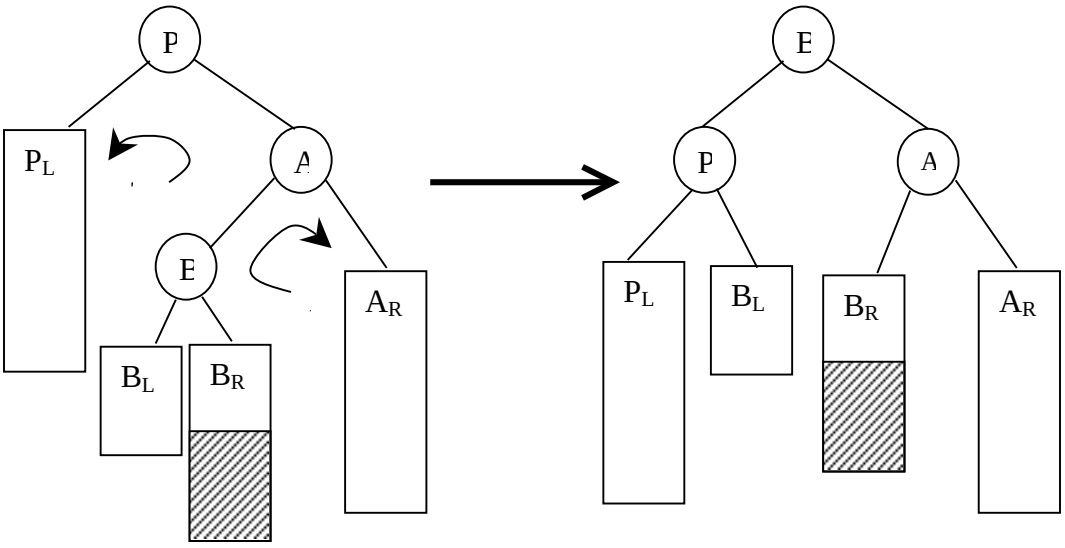
- Right subtree (B_R) of the left child (B) of the right child (A) of the pivot node (P) becomes the left subtree of A.
- Right child (A) of the pivot node (P) becomes the right child of B.

Rotation 2

- Left subtree (B_L) of the right child (B) of the right child (A) of the pivot node (P) becomes the right subtree of P.
- P becomes the left child of B.

This case is known as RIGHT-TO-LEFT insertion.

Illustration: fig. illustrates the case 4 of AVL rotation required during RIGHT-TO-LEFT insertion.



Pre-Lab Exercise:

Study article 10.1 (p. 514), 10.2 (p. 551), and review Exercises (p. 565) from your textbook.

Bridge Lab Exercise:

Implement a following class AVLTree.

```
template <class eType>
class CAVLNode
{
        friend class CAVLTree;
        private:
        eType data;
        CAVLNode < eType > *leftchild, < eType >*Rightchild;
        int bf;
        public:
        CAVLNode ();
};

template <class eType>
class CAVLTree
{
        CAVLNode < eType > *root;
        // . . .
        // utility functions goes here
        // . . .
        public:
        CAVLTree ();
        void m_fnInsert ( eType);
        void m_fnDelete (eType);
        Boolean m_fnSearch (eType);
        void m_fnInOrderTraverse();
        void m_fnPreOrderTraverse();
        void m_fnPostOrderTraverse();
        ~ CAVLTree ();
};
```

Driver Program 1: Driver Program for CAVLTree class is as follows

```
int main(void) {

        CAVLTree <int> AVL;

        for (int i=0; i<12; i++)
                AVL.m_fnInsert(i);

        AVL. m_fnInOrderTraverse();
        AVL.m_fnDelete(7);
        AVL. m_fnPreOrderTraverse();
        AVL.m_fnDelete(3);
        AVL. m_fnPostOrderTraverse();
```

```
        return 0;
}
```

Write the output of above programs.

In-Lab Exercise 1:

Overload <<, =, +, -, and == operator for AVL Tree which perform inorder reavesal, assignment, UNION, INTERSECTION and compression of two AVL trees class respectively..

```
int main(void) {

        CAVLTree <int> FirstAVL, SecondAVL, ThirdAVL;

        for (int i=0; i<6; i++)
                FirstAVL.m_fnInsert(i);

        for ( i=6; I>0; i--)
                SecondAVL.m_fnInsert(i);

        ThirdAVL =  FirstAVL + SecondAVL        // Third should not contain common
                                                // elements twice i.e. it is UNION
        cout << ThirdAVL;                       // Inorder traversal

        ThirdAVL  = SecondAVL – FirstAVL;

        if (ThirdAVL = = SecondAVL)
                cout << "TRUE" << endl;
        else
                cout << "FALSE" << endl;

        return 0;
}
```

Write the output of above programs for sample input data.

Extra Lab:

1. Analyze time complexity of insert, delete and search functions of CAVLTree class.
2. Measure the time for searching a particular value for the same insertion sequence of values in BST and AVL objects. (See Lab 2 performance analysis and measurement)
3. For a fixed k, k >= 1, we define a height balanced tree HB (k) as below:

Definition:
An empty binary tree is an HB (k) tree. If T is nonempty binary tree with T_L and T_R as its left and right suntrees, then T is HB (k) iff

(a) T_L and T_R are HB (k) trees
(b) $| h_L| - | h_R | \leq k$, , where hl and hr are the heights of T_L and T_R respectively.

Write insertion and deletion algorithms for HB (k) tree

Files to be submitted:

Put the following files (in a zip archive or folder), name it on your roll number and upload at \\pucit-waqar\assignments. We will compile and run your code to make sure it works as required.

The files to be submitted are:

1. `Bridge.cpp`
   `Contains the definition and driver program for bridge exercise`
2. `InLab1.cpp`
   `Contains the driver program and function definition for In-Lab exercise 1.`
1. `ExtraLab.doc`
   `This file should contain solution for extra lab 1.`
3. `Extra2.cpp`
   `Contains the driver program and function definition for HB (k) tree of extra lab 3.`
4. `Extra3.cpp`
   `Contains the driver program and function definition for HB (k) tree of extra lab 3.`

3.12. Lab 12 – Hash Table ADT

Objectives:

- Comprehension and Implementations of the Hash Table ADT.
- Implementation of Open addressing
- Implementation of Close addressing
- Comprehension and implementation of different collision resolution techniques

Overview:

Hashing is a method for sorting and searching data. Hashing makes it easier to add and remove elements from a data structure. The worst-case behavior for locating a key is linear O (n). Hashing usually implements a data structure called a ***hash table***. A hash table is an effective data structure. A hash table is a generalization of an array. A hash table requires a key to access data.

A hash table uses an array whose length is proportional to the number of keys actually stored. The array index is computed from the key, rather than using the key to access the array. We can understand the concept of hashing by the following example,

Suppose a teacher is checking assignments of students and a student come to ask about his marks than the teacher has to search all the assignments and find the required assignment. If the teacher arranges his record and store the assignments in file student number-wise than after knowing the roll no from the student he can tell the student his marks in less time and efficiently by directly checking the numbers before the roll no.

The ideal hash table data structure is merely an array of some fixed size, containing of the keys. Typically, a key is a string with an associated value (for instance, salary information). We will refer to the table size as HSIZE, with the understanding that this is a part of a hash data structure and not merely some variable floating around globally. The common convention is to have the table run from 0 to HSIZE-1; we will see why shortly.

Each key is mapped into some number in the range 0 to HSIZE-1 and placed in the appropriate cell. The mapping is called a hash function, which ideally should be simple to compute and should ensure that any two distinct keys get different cells. Since there are a finite number of cells and a virtually inexhaustible supply of keys, this is clearly impossible, and thus we seek a hash function that distributes the evenly among the cells. Figure 1 is typically of perfect situation. In this example, john hashes to 3, Phil hashes to 4, aver hashes to 6, and marry hashes to 7.

0	
1	
2	
3	John 25000
4	Hill 31250
5	
6	Dave 27500
7	Mary 28200

Hash table.

Hashing is used for performing insertions, deletions, and finds in constant average time. Tree operations that require any ordering information among the elements are not supported efficiently. Thus, operations such as Find_Min, Find_Max, and printing of the entire table in sorted order in linear time are not supported.

This is the basic idea of hashing. The only remaining problem deals with choosing a function, deciding what to do when two keys hash to the same value (this is known as collision), and deciding on the table size.

Hashing functions:

The choice of a hash function is the most important process in hashing. If we have succeeded in finding such a hash function, which is perfect, then it is a big achievement. A perfect hash function is that function which is capable of storing all the record in hash table without any clash. Here we shall discuss different hash functions and also discuss which is a better choice.

If the input keys are integers, then simply returning Key mod HSize is generally a reasonable strategy, unless Key happens to have some undesirable properties. In this case, the choice of hash function needs to be carefully considered. For instance, if the table size is 10 and the keys all end in 0, then the standard hash function is obviously a bad choice. For reasons we shall see later, and to avoid situations like the one above, it is usually a good idea to ensure that the table size is prime. When the input keys are random integers, then this function is not very simple to compute but also distributes the Keys evenly.

Usually, the keys are strings: in this case, the hash function needs to be chosen carefully. One option is to add up the ASCII values of the characters of the string. We have shown this in the function1.

```
/*        Function1        */
unsigned int
Hash(const String &Key,const int HSize) {
```

```
        const char *Key_Ptr=Key;
        unsigned int hash_Val=0;

        while(*Key_Ptr)
                Hash_Val+=*Key_Ptr++;
        return Hash_Val%HSize;
}

/*      Function2       */
unsigned int
Hash(const String &Key,const int HSize)     {
        return (Key[0]+27*Key[1]+729*Key[2])%HSize;
}

/*      Function3       */
unsigned int
Hash(const String &Key,const int HSize) {
        const char Key_Ptr=Key;
        unsigned int Hash_Val=0;

        while(*Key_Ptr)
                Hash_Val=(Hash_val<<5)+*Key_Ptr++;
        return Hash_Val%HasHSize;
}
```

Function – 1

The function of function-1 is simple to implement and computes the answer quickly. However, if the table size is large, the function does not distribute the keys well. For instance, suppose that HSize =10,007(10,007 is a prime number). Suppose all the keys are eight or fewer character long. Since a char is an integer value that is always at most 127, the hash function can only assume values between 0 and 1016, which is 127*8. This is clearly not an equitable distribution!

Function – 2

Another hash function is shown in function-2. This hash function assumes that key has at least length 2.27 represents the number of letters in the English alphabet, plus the blank, and 729 is 27^2. This function examines only the first three characters, but if these are random, and the table size is 10,007 as before, then we expect a reasonable equitable distribution. Unfortunately, English is not random although there are 26^3=17,576 possible combinations of three characters (ignoring blanks); a check of a reasonably large on-line dictionary reveals that the number of different combinations is actually only 2,851, even if none of these combinations collide; only 28 percent of the table can actually be hashed to. Thus this function, although easily computable, is also not appropriate if the hash table is reasonable large.

Function – 3

Function-3 shows a third attempt at a hash function. This hash function involves all characters in the key and can generally be expected to distribute well (it computes

$$\sum_{i=0}^{KeySize-1} Key[KeySize - i].32^{i}$$, and brings the result into proper range).

The code computes a polynomial function (of 32) by using Horner's rule. For instance another way of computing $h_k = k_1 + 27k_2 + 729k_3$ is by the formula $h_k = ((k_3)*27 + K_2)*27 + k1$. Horner's rule extends this to an nth degree polynomial.

We have used 32 instead of 27 because multiplication by 32 is not really a multiplication, but amounts to bit-shifting by five.

The main programming detail left is collision resolution. If, when inserting an element, it hashes to the same value as an already inserted element, then we have a collision and need to resolve it. There are several methods for dealing with this. We will discuss open hashing and closed hashing.

Main goal of a hashing function is speed. Keys should be uniformly distributed over the address space. Given are some methods of hashing and their names are quite famous.

Folding:

Folding is also an important method for finding unique keys.

Folding by Shifting:
Partition X into several parts and then adds the parts ignoring the carry.

Folding by boundary:
Partition X into several parts, reverse the digits in the first and the last partition and then add them all ignoring the carry

Mid-square hash function:

The mid-square hash function converts the key to an integer then doubles the key. The function returns the middle digits of the results.

Multiplicative hash function

The multiplicative hash function converts the key to an integer and multiplies it by a constant less than one. The function returns the first few digits of the fractional part of the result.

Division hash function

The division hash function depends upon the remainder of division.

Math.abs(H(k)) % TableLength

When using the division hash function, it is best to have a table size that is a prime number of the form 4n + 3. Using the division hash function can result in many collisions.

Handling the collisions

Collision:

When we insert different values in a hash table then it is possible that key determined by the hash function for different elements give the same place or index this situation is called the collision of elements.

Handling of collision:

In the small number of cases, where multiple keys map to the same integer, then elements with different keys may be stored in the same "slot" of the hash table. It is clear that when the hash function is used to locate a potential match, it will be necessary to compare the key of that element with the search key(Key determined by the hash function). But there may be more than one element which should be stored in a single slot of the table. Various techniques are used to manage this problem:

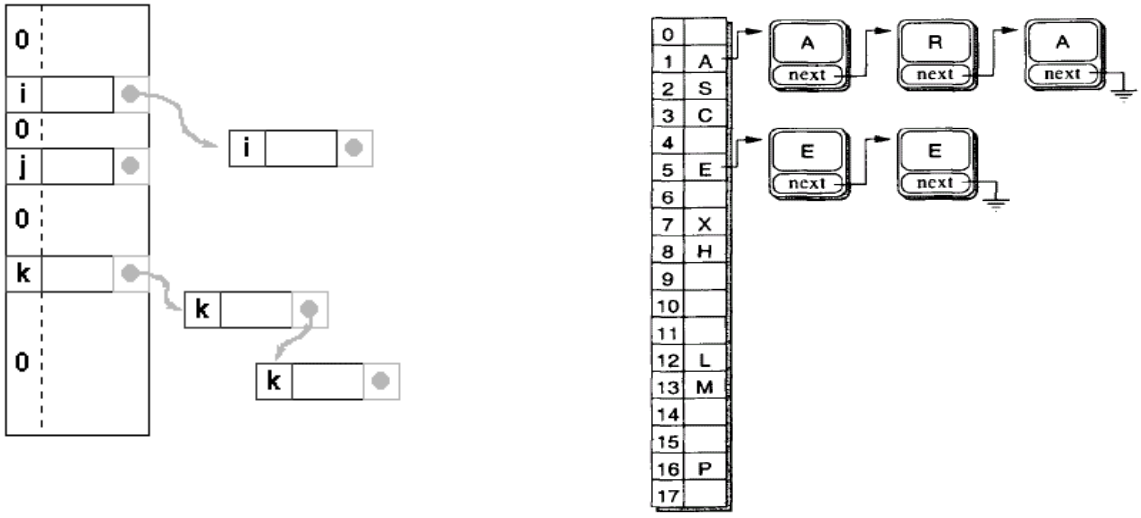
1. Open hashing (separate chaining)
2. Closed hashing (open addressing)
3. Overflow areas
4. Re-hashing

Open hashing (separate chaining)

The first strategy, commonly known as either open hashing or separate chaining, is to keep a list of all elements that hash to the same value. For convenience, our lists have headers. This makes the list implementation the same as of original list. If space is tight, it might be preferable to avoid their use. We can see from the diagram that when there are two elements of key k, in the table they are connected as a list.

To perform a find, we use the hash function to determine which list to traverse; we then traverse this list in a normal manner, returning the position where the item is found. To perform an insert, we traverse down the appropriate list to check whether the element is already in place (if duplicates are expected, an extra field is usually kept, and this field would be incremented in the event of a match). If the element turns out to be new, it is

inserted either at the front of the list or at the end of the list, whichever is easiest. This is an issue most easily addressed while the code is being written. Some time new elements are inserted at the front of the list, since it is convenient and also because frequently it happens that recently inserted elements are the most likely to be accessed in the near future.



Separate Chaining

Closed hashing (open addressing)

Open hashing has the disadvantage of requiring pointers. This tends to slow the algorithm down a bit because of the time required to allocate new cell, and it also essentially requires the implementation of the second data structure. Closed hashing, also known as open addressing, is an alternative to resolve collision with linked lists. In a closed hashing system, if a collision occurs, alternative cells are tried until an empty cell is found. More formally, cells $h_0(x),h_1(x),h_2(x),\ldots$are tried in succession where $h_i(x)=(HASH(i)+f(i))$ mod HSIZE, with f(0)=0. The function f is the collision resolution strategy. Because all the data goes inside the table, a bigger table is needed for closed hashing than for open hashing. We now look at four common collision resolution strategies.

1. Linear probing
2. Quadratic probing
3. Double Hashing

Linear probing

In the linear probing (probe means jump in the table), f is a linear function of i typically f(i) = j. This amounts to trying cells sequentially (with wraparound) in search of an empty cell. Inserting keys are placed into a closed table using the hash function (key % HSize), and collision strategy, f (i) = j.

As long as the table is big enough, a free cell can always be found, but the time to do so can get quite large. Of course, even if the table is relatively empty, blocks of occupied cells starts forming. This effect, known as *primary clustering*, means that any key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.

A problem associated with Linear Probing is called, primary clustering. Primary clustering occurs when many items hash into the same slot and long runs of slots are filled up. This results in increased search times. The key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.

Quadric probing

Quadric probing is a collision resolution method that eliminates the primary clustering problem of linear probing. Quadric probing is what you would expect the collision function is quadric. The popular choice is $f(i) = i^2$ for closed table with this function on the same input as of linear probing.

Double hashing

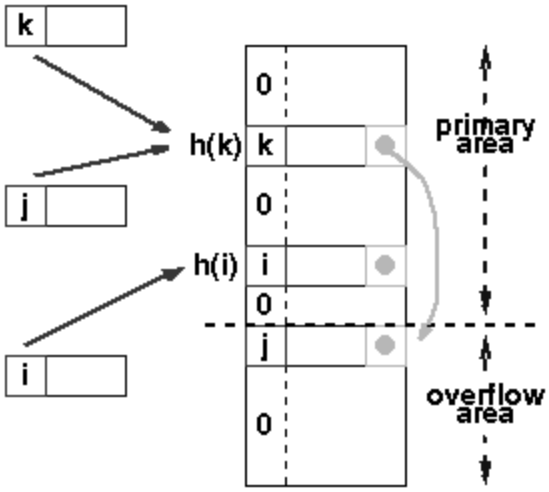
Double hashing is one of the best methods for dealing with collisions. The slot location is calculated based upon the hash function (H1 (k)). If the slot is full, then a second hash function is calculated and combined with the first hash function (H (k, i)) to determine a new slot.

For double hashing, one popular choice is f (i) = i. $hash_2$ (x). This formula says that we apply a second hash function to x and probe (travel) at a distance $hash_2$ (x), $2hash_2$ (x),…, and so on. A poor choice of $hash_2$(x) would be disastrous.

Overflow area

Another scheme will divide the pre-allocated table into two sections: the *primary area* to which keys are mapped and an area for collisions, normally termed the *overflow area.*

When a collision occurs, a slot in the overflow area is used for the new element and a link from the primary slot established as in a chained system. This is essentially the same as chaining, except that the overflow area is pre-allocated and thus possibly faster to access. As with re-hashing, the maximum number of elements must be known in advance, but in this case, two parameters must be estimated: the optimum size of the primary and overflow areas. Of course, it is possible to design systems with multiple overflow tables, or with a mechanism for handling overflow out of the overflow area, which provide flexibility without losing the advantages of the overflow scheme.

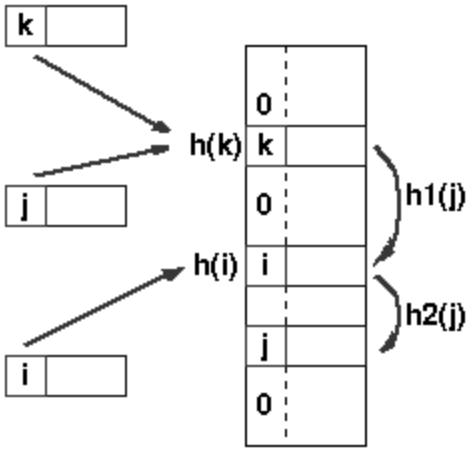


Rehashing

Re-hashing schemes use a second hashing operation when there is a collision. If there is a further collision, we *re-hash* until an empty "slot" in the table is found.

The re-hashing function can either be a new function or a re-application of the original one. As long as the functions are applied to a key in the same order, then a sought key can always be located

If the table gets too full, the running time for the operation will start taking too long and insert might fail for closed hashing with quadratic resolution. This can happen if there are too many removals intermixed with insertion. A solution, then, is to build another table that is twice as big (with associated new hash function) and scan down the entire original hash table, computing the new hash value for each (no deleted) element and inserting it in the new table.



Pre-Lab Exercise:

Study article 8.1 (p. 464), 8.2 (p. 466) and review Exercises (p. 480) from your textbook.

Bridge Lab Exercise:

Implement collision resolution techniques for followings for array based static hash table.

1. linear probing
2. Quadric probing
3. Re Hashing
4. Mid-square hashing

In-Lab Exercise 1:

Implement the class CHashTable using open hashing technique i.e. tables have headers and these headers points to a list.

```
static const DefaultSize=101;
template <class EType>
class CHashTable
{
        private:
        unsigned int HSize;
        List<EType> * TheLists;
        unsigned int CurrentList;
        void m_fnAllocateLists();
        CHashTable(CHashTable &value);
        public:
        CHashTable(unsigned int Initial_size=DefaultSize);
        ~CHashTable(){delete [] TheLists;}
        const CHashTable& operator=(const CHashTable & value);
        const EType &operator ()()const {return TheLists[CurrentLists];}
        void m_fnInitializeTable();
        void m_fnInsert(const EType &Key);
        void m_fnRemove(const EType &Key);
        int m_fnFind(const EType &Key);
};
```

In-Lab Exercise 2:

Implement the class CHashTable using closed hashing technique.

```
const int DefaultSize=101;
template <class EType>
class CHashTable {
```

```
        public:
        enum KindOfEntry{Legitimate, Empty, Deleted};

        protected:
                struct HashEntry {
                EType Element;
                KindOfEntry Information;
                HashEntry(EType E=0, KindOfEntry i=Empty):
                Element(E),Information(i){};
                };

        unsigned int HSize;
        HashEntry * TheCells;           //The last cell accessed
        unsigned int Current_Cell;
        void m_fnAllocateCells();
        CHashTable(CHashTable &value);     //Disabled

        public:
        CHashTable(unsigned int InitialSize=DefaultSize);
        virtual ~CHashTable(){delete [] TheCells;}

        const CHashTable& operator=(const CHashTable & value);
        const EType &operator()()const {return TheCells[CurrentCell].Element;}

        virtual void m_vfnInitializeTable();
        virtual void m_vfnInsert(const EType &Key);
        void m_fnRemove(const EType &Key);
        int m_fnFind(const EType &Key);
};
```

In-Lab Exercise 3:

Implement the class CDynamicHashTable using rehashing technique. Note that Dynamic Hash Table is inherited from Hash Table. It uses the functions of hash table and performs its functionality.

```
template <class EType>
class CDynamicHashTable: public CHashTable<EType>
{
        private:
        unsigned int NumberOFElementsInTable;
        void m_fnRehash();
        protected:
                struct HashEntry {
                EType Element;
```

```
            KindOfEntry information;
            HashEntry(EType E=0, KindOfEntry i=Empty):
            Element(E),Info(i){};
            };

        unsigned int HSize;
        HashEntry *TheCells;          //The last cell accessed
        unsigned int CurrentCell;
        void m_fnAllocateCells();
        CHashTable(CHashTable &value);    //Disabled

        public:
        CDynamicHashTable(unsigned int InitialSize=DefaultSize) {
            CHashTable<EType>::CHashTable(InitialSize);
            NumberOfElementsInTable=0;
        }
        const CHashTable& operator=(const CHashTable & value);

        virtual void m_vfnInitializeTable()   {
            CHashTable<EType>::Initialize_Table();
            NumberOfElementsInTable=0;
        }
        virtual void m_vfnInsert(const EType &Key);
};
```

Extra Lab:

1. Write down hashing functions for followings
    a. Folding by Shifting
    b. Folding by boundary
    c. Mid-square hash function
    d. Multiplicative hash function
    e. Division hash function
2. Compare the performance of above hashing techniques for different data set.
3. When using the division hash function, it is best to have a table size that is a prime number of the form 4n + 3. Test the validity of above statement for different data set.

Files to be submitted:

Put the following files (in a zip archive or folder), name it on your roll number and upload at \\pucit-waqar\assignments. We will compile and run your code to make sure it works as required.

The files to be submitted are:

1. `Bridge.cpp`

Contains the solution of bridge exercise.

2. Lab1.cppp
   Contains the implementation of CHashTable class for Lab exercise 1.
3. Lab2.cpp
   Contains the implementation of CHashTable class for Lab exercise 2.
4. Lab3.cpp
   Contains the implementation of CDynamicHashTable class for Lab exercise 3.
5. ExtraLab.cpp
   This file contains the extra lab exercises 1 and 3.
6. ExtraLab.doc
   This file should contains solution for extra lab 2.

3.13. Lab 13 – Implementation of Graph ADT

Objectives:

- Implementations of the Graph ADT— based on an array representation and multi list.
- Use templates to produce a generic Graph data structure.

Overview:

A set of items connected by *edges*. Each item is called a *vertex* or node. Formally, a graph is a *set* of vertices and a *relation* between vertices, adjacency.

Graphs are so general that many other data structures, such as trees, are just special kinds of graphs. A simple undirected graph G consists of a set of vertices V and a set of edges E The elements of E are defined as pairs of elements of V, e_k = *(u, v)* such that *u* not equal to *v* and *(u, v)* an element of E implies that *(v, u)* is also an element of E. (In other words *(u, v)* and *(v, u)* represent the same edge). If the graph is undirected, the adjacency relation is symmetric. If the graph does not allow self-loops, adjacency is irreflexive.

A graph is like a road map. Cities are vertices. Roads from city to city are edges. You could consider junctions to be vertices, too. If you don't want to count them as vertices, a road may connect more than two cities. So strictly speaking you have hyper edges in a hyper graph. If you want to allow more than one road between each pair of cities, you have a multi-graph, instead. It all depends on how you want *to define it.)*

Another way to think of a graph is as a bunch of dots connected by lines. Because mathematicians stopped talking to regular people long ago, the dots in a graph are called vertices, and the lines that connect the dots are called edges. The important things are edges and the vertices: the dots and the connections between them. The actual position of a given dot or the length or straightness of a given line isn't at issue. Thus the dots can be anywhere, and the lines that join them are infinitely stretchy. Moreover, a mathematical graph is not a comparison chart, nor a diagram with an x- and y-axis, nor a squiggly line on a stock report. A graph is simply dots and lines between them, vertices and edges.

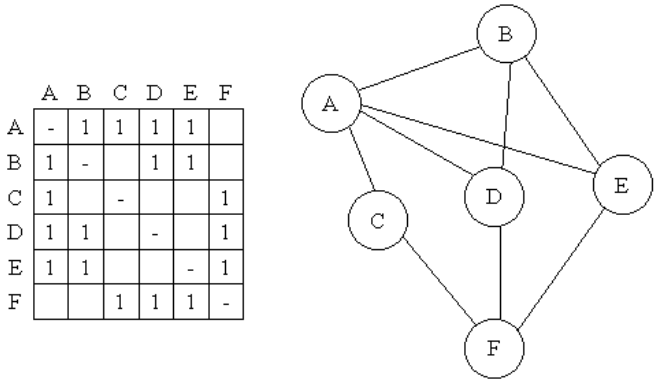
Graph Representations

Adjacency-Matrix Representation

Representation of a directed graph with n vertices using an n × n Boolean (or equivalent) matrix is called Adjacency Matrix Representation. We label the row and columns of the matrix with the names of the vertices. Each element of the matrix represents a potential edge in the graph as defined by its associated vertex pair. We set an element of the matrix to TRUE if there is an edge connecting the two corresponding vertices, otherwise we set the element to FALSE. We can also use 1's and 0's to indicate the presence or absence of an edge.

In this way, if the entry at (i, j) is 1 there is an edge from vertex i to vertex j; otherwise the entry is 0. A weighted graph may be represented using the weight as the entry. An undirected graph may be represented using the same entry in both (i, j) and (j, i) or using an upper triangular matrix.

The adjacency-list representation is more compact for a sparse matrix.



	A	B	C	D	E	F
A	-	1	1	1	1	
B	1	-		1	1	
C	1		-			1
D	1	1		-		1
E	1	1			-	1
F			1	1	1	-

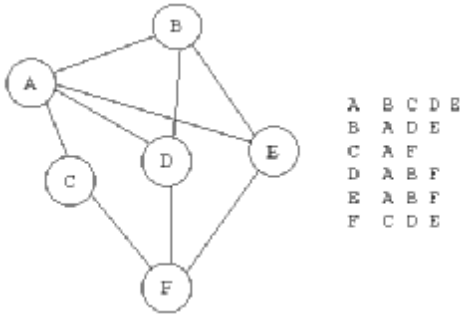
A Graph and Its Adjacency Matrix

In the example above the 1's represent the presence of an edge and a blank indicates that there is no edge connecting the corresponding pair of vertices. The dashes indicate that there can be no edge connecting a vertex to it. For a weighted graph the edge value will replace the 1's.

Adjacency List

For each vertex we list the vertices connected to this vertex by an edge in the graph. For an ordinary n-vertex connected graph the number of vertices connected to a particular vertex is no greater than n-1. Therefore the edge list can be as large as n(n-1), or n(n-1)/2 if each edge is represented only once.

Simple Adjacency Matrix

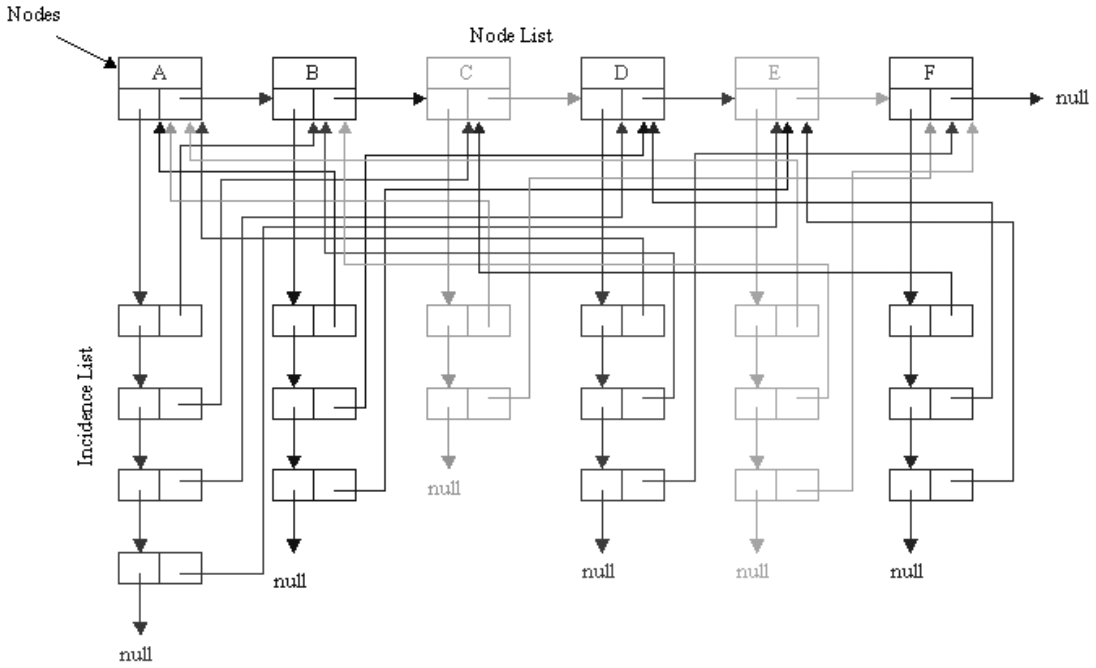


If the graph has many more edges than vertices (approaching a complete graph) then the adjacency matrix is the preferred method of representation. If the graph is sparse (i.e. the number of edges is much less than the maximum number of edges) then an edge list is probably preferred.

Dynamic Memory Representations

The graph representation methods described above use static data structures such as arrays and lists. Sometimes we need to use a method of representation that permits arbitrary growth and/or restructuring of the graph. We can use dynamic memory to accomplish this. Representing graphs using pointer records and pointers should be considered only in those cases in which the adjacency matrix or adjacency list representations are not feasible.

A graphical representation of the dynamic memory data structure for our six-node sample graph is provided below. In this representation we have two types of nodes. There is a node record for each vertex as shown in the node list. A node record has a pointer to the next node (this pointer does not represent an edge in the graph) and another pointer to an incidence list. The incidence record contains two pointers. One pointer points to the next incidence record for this node and the other points to one of the vertices that is adjacent to this vertex.



A Dynamic Memory Graph Representation

In our example vertex A is adjacent to 4 vertices so its incidence list has 4 records. The edge pointers for vertex A point to records B, C, D and E. The complexity of the data structure for this simple graph indicates that this representation method should be considered only if warranted by the application.

Pre-Lab Exercise:

Study article 6.1 (p. 330), and review Exercises (p. 343) from your textbook.

Bridge Lab Exercise:

If the number of nodes in the graph is constant: that is, arcs may be added or deleted but nodes may not, then array-based implementation of graph is suitable.

Implement following classes to represent array-based graph ADT.

```
template <class eType>
class CNode {
friend class Graph < eType >;
        eType Data;
        /*  information related with each node  */
};

template <class eType>
class CArc {
friend class Graph < eType >;
        Boolean Adj;
        /*  information related with each arc  */
};

template <class eType>
class CGraph {
        private:
        CNode < eType > * Nodes;    // one dimensional array for Node information
                                    // [MAXNODES];
        CArc < eType > * Arcs;      // one dimensional array for arc information
                                    //[MAXNODES][ MAXNODES];
        /*  information related with Graph  */
        public:
        CGraph(int NoOfNodes);
        Boolean m_fnJoin (int NodeNumber1, int NodeNumber1);
        Boolean m_fnRemove (int NodeNumber1, int NodeNumber1);
        Boolean m_fnIsAdjacent (int NodeNumber1, int NodeNumber1);
        ~CGraph();
};
```

Driver Program 1: Driver Program for CGraph class is as follows

```
int main(void) {

CGraph <int> IntGraph(5);

for (int i=0, j=5; i<5; i++, j--)
        IntGraph.m_fnJoin(i, j);

cout  << IntGraph.m_fnRemove(--i, ++j ) <<  endl ;

for (i=0, j=0; i<5; i++)
        cout << IntGraph.m_fnIsAdjacent() << endl;

return 0;
}
```

Write the output of above programs.

In-Lab Exercise 1:

In the dynamic implementation of Graphs note that header nodes and list nodes have different formats and must be represented by different structures. This necessitates either keeping two distinct available lists or defining a union. Even in the case of a weighted graph in which each list node contains an *info* field to hold the weight of an arc, two different structures may be necessary if the information in the header nodes is not an integer. However, for simplicity we make the assumption that both header and list nodes have the same format and contain two pointers and a single integer information field. So finally two classes are required, one is of Node and the other is of Graph. The dynamic implementation of Graphs is illustrated as follows:

```
template <class eType>
class CNode {
        private:
        eType info;
        CNode <eType> *NextNode;
        CArc <eType> *ArcHeader;

        Public:
        Node (eType data=0)  {
                Info = data;
                NextNode =NULL:
                ArcHeader =NULL;
        }
};
```

```
template <class eType>
class CArc {
        private:
        eType info;
        CNode <eType> *DestinationNode;
        CArc <eType> *NextArc;

        public:
        Node (eType data=0)  {
                Info = data;
                NextNode =NULL:
                ArcHeader =NULL;
        }
};

template <class eType>
class CGraph {
        private:
        CNode  <eType> *Head;

        public:
        CGraph();
        Boolean m_fnFindNode (eType Node);
        Boolean m_fnAddNode (eType Val);
        Boolean m_fnDeleteNode (eType Val);
        Boolean m_fnAdjacent (eType Node1, eType Node2);
        Boolean m_fnJoinLink (eType Node1, eType Node2);
        Boolean m_fnRemoveLink (eType Node1, eType Node2);
        ~CGraph();
};
```

In-Lab Exercise 2:

Write following functions for CGraph class discussed in In–Lab exercise 1.

1. Copy constructor
2. Assignment operator
3. Comparison operator

Extra Lab:

1. Modify CGraph class discussed in In–Lab exercise 1 for weighted graphs.
2. Calculate Time complexity for function remove that removes link between two node.

Files to be submitted:

Put the following files (in a zip archive or folder), name it on your roll number and upload at \\pucit-waqar\assignments. We will compile and run your code to make sure it works as required.

The files to be submitted are:

3. `Bridge.cpp`
   Contains the definition and driver program for CGraph class of bridge exercise
4. `Lab1.cpp`
   Contains the class implementation and function definition for Lab exercise 1
5. `Labe2.cpp`
   Contains the function definition for Lab exercise 2
6. `EvalPost.h`
   Contains the driver program and function definition for Lab exercise 3
7. `ExtraLab.cpp`
   This file contains the extra lab exercises 1, and 2.

3.14. Lab 14 – Implementation of Graph Operations

Objectives:

- Implementations of following operations for CGraph ADT
    - Depth First Search
    - Breadth First Search
    - Shortest Path
    - Minimum Cost Spanning Tree

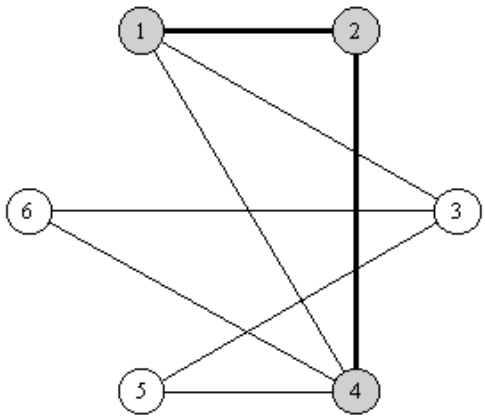
Graph Traversals

An important operation on graphs is the ability to move through the data representation, testing each vertex and/or edge value. This evaluate-and-move operation is referred to as a traversal. There are two popular methods of traversal used in solving many of the problems involving graphs and trees. They are called depth-first traversal and breadth-first traversal.

In any traversal we need to establish a convention for ordering the vertices. In our examples we will use alphanumeric ordering of the vertex labels. This means that if two or more vertices can be chosen next, we will select the vertex with a label that would come first in an alpha-numeric sort.

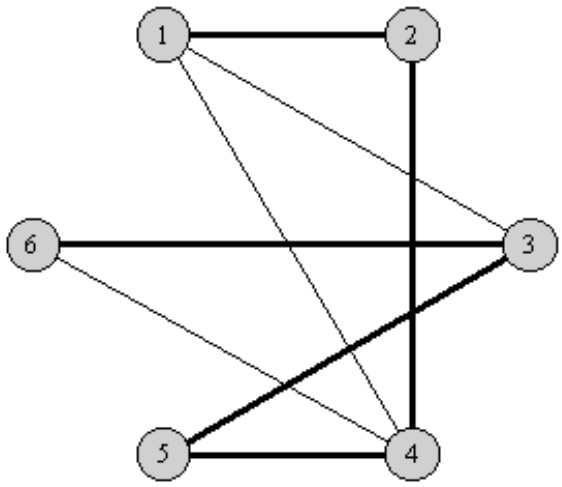
Depth-First Traversal

A DFT can be implemented for a graph. In a graph traversal we need to keep a record of the nodes that have been traversed so that they are not entered more than once. In the example below we will perform a depth-first search of the graph shown below starting at node 1.



Sample Graph Showing the Beginning of a Depth-First Traversal (DFT)

The nodes adjacent to node 1 are 2, 3 and 4 so we choose 2 (by our convention). From node 2 we choose 4 since node 1 has already been expanded. From node 4 we choose node 5 since it is the first unexpanded node in an alphanumeric ordering of 1, 2 and 5 (actually its the only unexpanded node reachable from node 4). From node 5 we choose 3 and 6.



Completed DFT

Breadth-First Traversal

In a breadth-first traversal (BFT) all the children of the current node (i.e. the node currently being evaluated) are placed onto the queue before any of their children are considered. A breadth-first traversal is obtained by following the stepwise procedure below:

Step 1: Place the starting node onto the queue.
Step 2: If the queue is not empty retrieve the node that is at the front of the queue and add its label to the order-of-traversal list, otherwise STOP.
Step 3: Place all the unencountered children of this node onto the queue (in alpha-numeric order) and tag them as having been encountered.
Step 4: Return to Step 2.

The order-of-traversal list is a list of all the reachable nodes in the graph or tree arranged in the order in which they were encountered in the breadth-first traversal. Note that, just as in the depth-first traversal, we have to keep track of which nodes have been encountered so that we don't evaluate any node more that once.

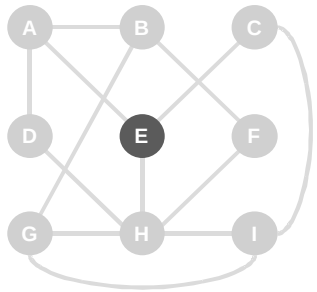
Let's try this procedure on the sample graph shown below. We will start with node E and use our alphanumeric ordering convention for arranging the nodes encountered at each step.

We choose node E as our arbitrary start node (the actual starting node would be determined by the requirements of the application or problem being solved). We place E onto our queue and tag it as a used node.

Queue: E <-front of queue

Tagged: A B C D **E** F G H I

Order-Of-Traversal: [empty]



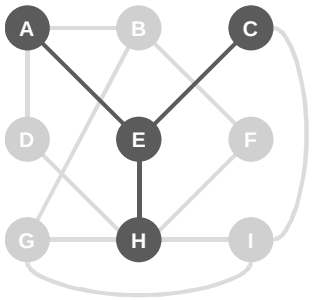
Sample BFT Graph

The node at the front of the queue is E, we remove it and add its label to the Order-Of-Traversal (OOT) list. The children of E are A, C and H. Since none of these nodes have been encountered, they are all placed onto the queue.

Queue: H C A <-front of queue

Tagged: **A** B **C** D **E** F G **H** I

Order-Of-Traversal: E

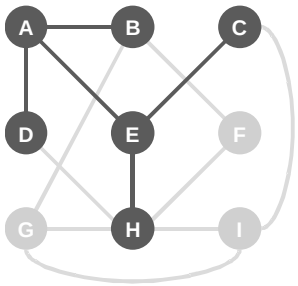


Node A is now at the front of the queue so we remove it, add its label to the OOT list and place the unused children of A onto the queue.

Queue: D B H C <-front of queue

Tagged: **A B C D E** F G **H** I

Order-Of-Traversal: E A

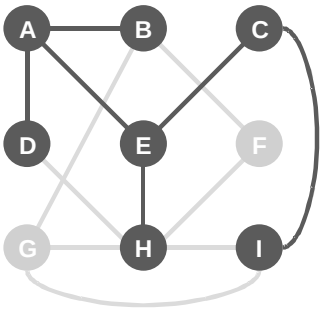


Node C is the next node at the front of the queue. We remove C add its label to the OOT list and place it child, node I onto the queue.

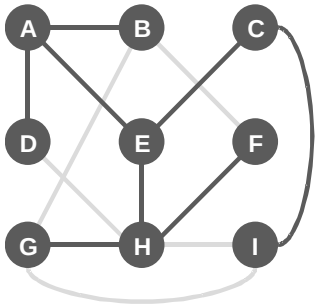
Queue: I D B H <-front of queue

Tagged: **A B C D E** F G **H I**

Order-Of-Traversal: E A C



Node H is the next node at the front of the queue. The children of H are D, E, F, G and I but only F and G are new nodes, so we place these two nodes onto the queue.



Queue: G F I D B <-front of queue

Tagged: A B C D E F G H I

Order-Of-Traversal: E A C H

Since all the nodes have been encountered we can flush the queue as we add the nodes to the OOT list. The final list is,

Order-Of-Traversal: E A C H B D I F G

Prim's Algorithm

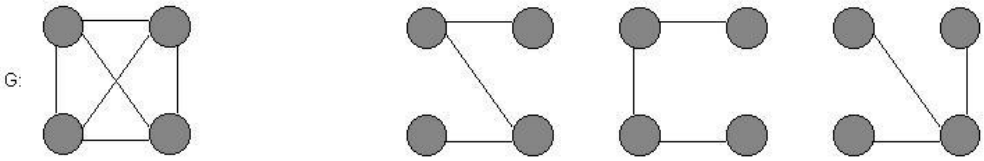
This algorithm builds the MST one vertex at a time. It starts at any vertex in a graph (vertex A, for example), and finds the least cost vertex (vertex B, for example) connected to the start vertex. Now, from either 'A' or 'B', it will find the next least costly vertex connection, without creating a cycle (vertex C, for example). Now, from either 'A', 'B', or 'C', it will find the next least costly vertex connection, without creating a cycle, and so on it goes. Eventually, all the vertices will be connected, without any cycles, and an MST will be the result. (NOTE: Two or more edges may have the same cost, so when there is a choice by two or more vertices that is exactly the same, then one will be chosen, and an MST will still result)

Minimum Spanning Trees

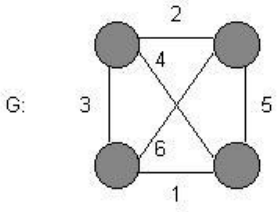
A tree is defined to be an undirected, acyclic and connected graph (or more simply, a graph in which there is only one path connecting each pair of vertices).

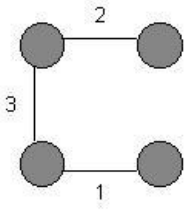
Assume there is an undirected, connected graph G. A spanning tree is a subgraph of G, is a tree, and contains all the vertices of G. A minimum spanning tree is a spanning tree, but has weights or lengths associated with the edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum. Here are some examples:

A graph G: Three (of the many possible) spanning trees from graph G:



A weighted graph G: The minimum spanning tree from weighted graph G:





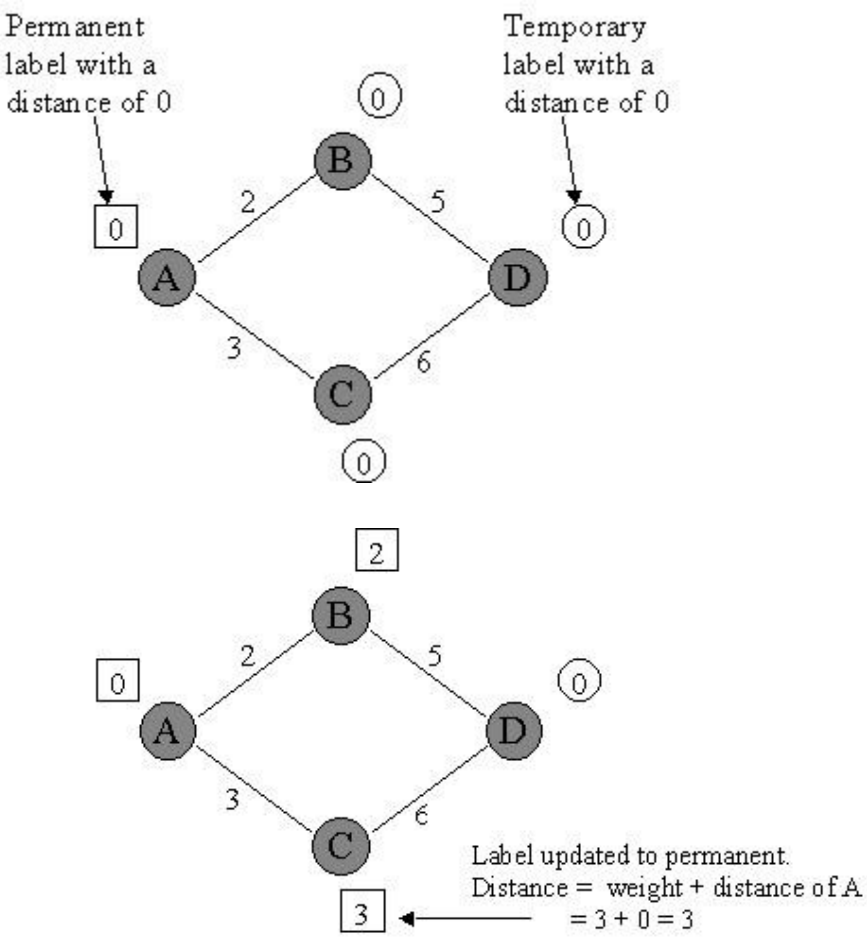
Dijkstra's Algorithm

There are many different operations that can be done on graphs. Methods such as Kruskal's algorithm and Prim's algorithm find the most efficient way to traverse an entire graph. However, if the distance (cost) between two given vertices needed to be calculated, an alternate method would be required. Dijkstra's algorithm determines the distances (costs) between a given vertex and all other vertices in a graph. This may be useful to determine alternatives in decision-making. For example, a telephone company may forgo the decision to install a new telephone cable in a rural area when presented with the option of installing the same cable in a city, reaching twice the people at half the cost.

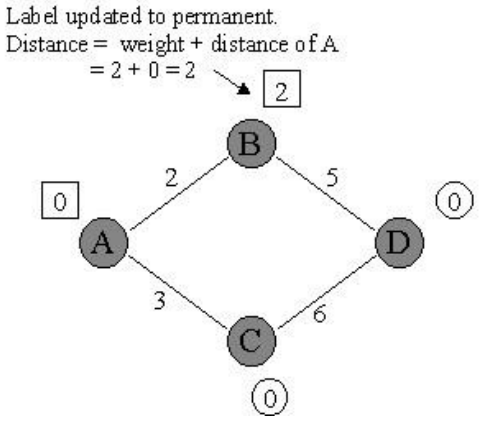
Dijkstra's algorithm is almost identical to that of Prim's. The algorithm begins at a specific vertex and extends outward within the graph, until all vertices have been reached. The only distinction is that Prim's algorithm stores a minimum cost edge whereas Dijkstra's algorithm stores the total cost from a source vertex to the current vertex. More simply, Dijkstra's algorithm stores a summation of minimum cost edges whereas Prim's algorithm stores at most one minimum cost edge.

Dijkstra's algorithm creates labels associated with vertices. These labels represent the distance (cost) from the source vertex to that particular vertex. Within the graph, there exists two kinds of labels: temporary and permanent. The temporary labels are given to vertices that have not been reached. The value given to these temporary labels can vary. Permanent labels are given to vertices that have been reached and their distance (cost) to the source vertex is known. The value given to these labels is the distance (cost) of that vertex to the source vertex. For any given vertex, there must be a permanent label or a temporary label, but not both.

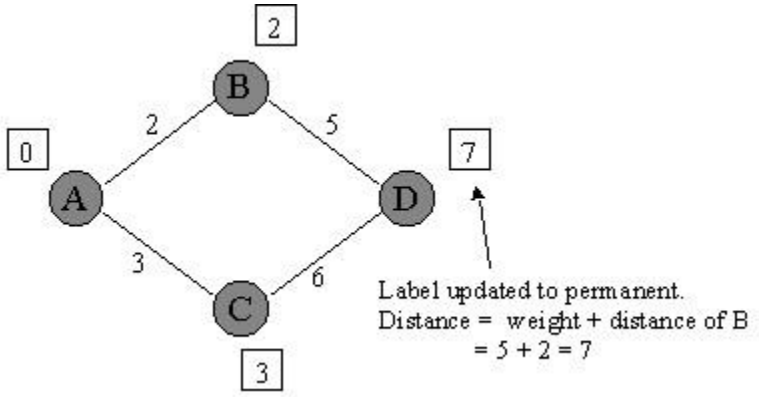
Vertex A has a temporary label with a distance of 0; Vertex B has a permanent label with a distance of 5



The algorithm then proceeds to select the least cost edge connecting a vertex with a permanent label (currently vertex A) to a vertex with a temporary label (vertex B, for example). Vertex B's label is then updated from a temporary to a permanent label. Vertex B's value is then determined by the addition of the cost of the edge with vertex A's value.



The next step is to find the next least cost edge extending to a vertex with a temporary label from either vertex A or vertex B (vertex C, for example), change vertex C's label to permanent, and determine its distance to vertex A.



This process is repeated until the labels of all vertices in the graph are permanent.

Pre-Lab Exercise:

Study article 6.2 (p. 345), 6.3 (p. 356), 6.4 (p. 364), and review Exercises (p. 354, p.363) from your textbook.

Bridge Lab Exercise:

Implement DFS and BFS for CGraph class discussed in In-Lab exercise 1 from lab 12.

In-Lab Exercise 1:

Write the following functions for CGraph class discussed in In-Lab exercise 1 from lab 12.

```
template <class eType>
Boolean m_fnIsConnecetd (eType Node1, eType Node2 );

template <class eType>
int m_fnShortestPath (eType Node1, eType Node2 );

template <class eType>
int m_fnMinimumCostSpanningTree (void);
```

Files to be submitted:

Put the following files (in a zip archive or folder), name it on your roll number and upload at \\pucit-waqar\assignments. We will compile and run your code to make sure it works as required.

The files to be submitted are:

1. `Bridge.cpp`
   `Contains the function definition for bridge exercise.`
2. `Lab1.cpp`
   `Contains the function definition for Lab exercise 1.`

Part 4

Case Studies

4.1. Case Study 01 – Game of Life

The *Game of Life* is a simulation invented by John H. Conway as a genetic model of some simple laws of birth, death, and survival in a constrained environment. You will replicate a variation of this simulation.

The playing board is a rectangular configuration of squares. The number of squares can be varied. The world the board represents is actually ***spherical*** so that cells at the edges are adjacent to cells at opposite edges (e.g., for an m x n board, the m^{th} row is followed by the first row, the nth column is followed by the first column, and so on).

Each square contains an organism, or is empty. In the next generation, the fate of each "organism" is determined by the following rules:

1. If an empty square has exactly three neighbors, an organism is born.
2. If an existing organism has four or more neighbors, it dies from overcrowding.
3. If an existing organism has fewer than two neighbors, it dies from loneliness.
4. If an existing organism has two or three neighbors, it survives for another generation.

Each cell, including those on the edges, has 8 neighbors – right, left, up, down, and diagonal.

Your C++ program should allow the user to input the size of the playing board (number of rows and columns) and an initial configuration where 1 means an organism is in that square and 0 means the square is empty. For example:

```
5    5

0    1    1    0    0
1    0    1    1    0
0    0    1    1    1
0    1    0    0    1
1    1    0    0    0
```

Note that in the input of the rows of 0's and 1's, each digit should be separated by at least one space from the next, e.g.,"0 1 1 0 0" rather than "01100." The program calculates the values for the next generation, and prints out this new configuration in a similar form of rows of 0's and 1's, then asks the user:

Do you want to see another generation? (Y or N)

This continues until the user answers "No". Depending on the initial configuration, some interesting patterns can develop in the passing generations of this game.

Method:

You will create three ADT's: cell, board, and SphereMatrix. Instances of the *cell* class will represent each cell. This class will have two data members – one value to represent the existence or absence of an organism in the current generation, and one to represent the condition in the next generation. The member functions for this class include a simple constructor, an accessor, and a function to calculate the new value given the current number of neighbors (but without actually changing the cell's status). You will also need an output and an input function.

An instance of the *board* class will represent the playing board. This class will have data elements to store the dimensions of the board, and a data element that is a *SphereMatrix* (described below) of cells. You may want a constructor function that takes the dimensions of the board as a parameter. You will need a *numNeighbors0* function that will take a row and a column as parameters, and returns the number of occupied neighbors of the corresponding cell. (Don't worry about double-counting neighbors. This is only an issue for boards smaller than 3x3.) You will also need a member function to calculate a new generation. It will do this by determining the number of neighbors for each cell, passing this information to that cell to let it determine its status for the next generation.

The board class uses a *SphereMatrix* object to represent the board. A SphereMatrix is like a regular matrix, except that indices "wrap around." For example, if the matrix is 3x3 and a request is made for item (-1,4), item (2,1) will be returned (assuming rows and columns are numbered 0 to 2). You will need to create this class. I recommend creating a class template since this does not require much additional work and make your ADT more reusable, but this is not necessary. The class will have data elements for the dimensions of the board and a standard C++ array. You will want to provide a constructor, a copy constructor, and an overloaded assignment operator. Also provide an accessor method that overloads operator() (the function-call operator). This method should take integer row and column arguments and return the correct element of the matrix, accounting for any wrap-around effects.

Use other member and ordinary functions as needed.

Testing:

To test your program, you may want to create an input file that contains input data to the program in exactly the same form as you type when the program runs. Then, from a DOS prompt, you can re-direct input using <. For example, if you program is called *prog1.exe* and you have placed input in the text file *test1.txt,* you can give the following command at the DOS prompt:

```
prog1.exe < test1.txt
```

This allows you to test many times without having to type in a lot of data over and over again. In fact, you can also redirect output to another file to hand in. For example, at a DOS prompt type:

prog1.exe < test1.txt > answers1.txt

Documentation Guidelines:

Good documentation includes:

- Descriptive names for variables, constants, functions, etc., that follows a consistent naming methodology.
- Titles for subsections of functions.
- Short explanations for pieces of code that are not immediately transparent.
- Headers for ADTs (i.e., class specifications), functions, and the main file. The following is the template of a sample header for the main file of this assignment:

```
//*******************************************************
// Program: Game of Life
//
// Description:
//
// Inputs:
// Outputs:
//
// Author:
// Class:
// Date created:
// Last modified:
//*******************************************************
```

Please use your creativity. You are not limited by the above guidelines.

4.2. Case Study 02 – Stack Application: A Maze

Problem:

Find a path from the entrance to the exit of a maze if one exists with as little backtracking as possible. This is a *search* problem.

Definition:

A maze is an *m* x *p* two-dimensional space containing an entrance, an exit, paths, and walls. Paths may turn in any of eight directions – N, NE, E, SE, S, SW, W, NW. We assume the entrance is at location <1,1> and the exit is at location <*m*,*p*>.

Objects:

maze, path?, step?.

Maze operations:

isClear, isExit, visited?.

Representation:

- walls (rather than maze): (m+2)x(p+2) array of boolean, where false (0) means the path is clear and true (1) means the path is blocked. To avoid checking for border conditions, surround maze with walls.
- visited (rather than mark): (m+2)x(p+2) array of boolean, instantiated to false, to record locations we've visited.
- path (rather than stack): mxp stack of steps we've made to this point, minus the false steps.
- step (rather than items): class/struct of triple <x, y, dir> where dir is the next direction to try from location <x,y>. Note: dir not really necessary since we theoretically have enough information on the stack to compute it, but this may be more trouble than it's worth given our representation.
- Implementation of operations: straight-forward

Algorithm:

- For convenience, we predefine the possible directions to move and the corresponding relative offsets:

```
enum directions {N, NE, E, SE, S, SW, W, NW};
struct offsets {
        int r, c;
};
offsets move[8];
```

move[N].*r* = -1; *move*[N].*c* = 0;
…

From a given cell, try all possible directions. As you move in a particular direction, push the last location on to the stack. Should you reach a point that every direction from a cell is either blocked or already explored, back up by popping your last move off the stack and continuing. Once you reach the end, your stack will contain your path. (Note that the textbooks implementation requires printing out the last two steps explicitly since these never get pushed on the stack.)

4.3. Case Study 03 – FreeCell Game

Objective:

To develop free cell game.

Rules of the Game:

1. There are 52 cards in the game.
2. These cards are in two different colors (Red and Black).
3. The symbols on these cards are ♣ ♦ ♥ ♠.
4. The numbers on these cards are from 2 to 10 and J, Q, K and A.
5. **K** has the highest priority followed by, Q, J, 10, 9, 8, 7, 6, 5, 4, 3, 2, A.
6. There are 4 cards of each number and that card contain the above mentioned symbol e.g. let we have a number 6 then 4 cards contain this number but with different symbol i.e. 6 ♣, 6 ♦, 6 ♥, 6 ♠.
7. You can place one card on the other if it is of different color and one less the number e.g. 6 can be pushed on 7.
8. There are 4 Frames/Areas in the game i.e. F1, F2 and F3.
9. F1 contains 4 stacks and the size of all these stacks is 1.
10. You can place the card in these 4 stacks and pick from them.
11. F2 contains 4 stacks and the size of all these stacks is 13.
12. You can place the cards in these stacks but can't pick them.
13. F3 contains 8 stacks and you can place and pick cards in these stacks maximum size of stacks in F8 is 20.
14. F4 is the prompt or command area.
15. Initially, All stacks of F1, F2 are empty and F3's stacks contains random cards in the following order:
16. First 4 stacks contain 7 cards each and last 4 stacks contain 6 cards each.
17. These cards are in random order.
18. The goal of this game is to push all 52 cards in the 4 stacks of F2 in regular order.
19. The Proposed format of command is:
20. Push (n) Cards of Frame (e), Stack (s) to Stack (t) of Frame (f).

NOTE:

If any rule is still missing, then play this game placed in your windows operating system and considers it.

4.4. Case Study 04 – C to Assembly Converter

Problem:

Assume that a machine has a single register AX and instruction set consisting of only six commands described bellow:

Command	Operands	Description
LD	AX, A	Loads the value of variable A into the register AX
ST	AX, A	Stores the value of the register AX into variable A
ADD	AX, A	Adds the contents of the variable A in the contents of register AX and leave the result in register AX
SUB	AX, A	Subtracts the contents of variable of A from the contents of register AX and leave the result in register AX
MUL	AX, A	Multiplies the contents of variable A and the contents of register AX and leave the result in the register AX
DIV	AX, A	Divides the contents of register AX by the contents of variable A and leave the result in register AX

Write a program in C++, which will accept an expression in IN-FIX form and convert this expression into the POST-FIX form and then generate its Assembly code using the above-mentioned instruction set. The program will also simulate all these steps i.e. program will show the steps while evaluating the expression, and a pointer IP will show that what line of the code is executing.

If the user input the expression with numerics then it is ok and it is easy to evaluate the expression. But if the user input the expression like the example given below, then your program suppose to prompt the user to give the numeric input for the the alphabatical operands of the expression after converting into postfix, so that the expression can be evaluated.

Program Screen:

The screen of this program may look like:

Input String in IN-FIX Form:

Input Values of A, B, C, D, E . . . *(user will input the values of variables)*

Current status of Expression Stack

Current status of Operator Stack

POST-FIX Expression:

Assembly Code	Register AX
	Value
	TEMP
	Value
	. . . . .

(User will press successive enters for step execution of simulation program)

Demonstrationary Example:

Input String in IN-FIX Form: A + B + B * (D + E)

Input Values of A, B, C, D, E . . . *(user will input the values of variables)*

A	B	C	+	+	D

*	(	+			

POST-FIX Expression: ABC++DE+*

Assembly Code

```
LD    AX, A
ADD   AX, B
ADD   AX, C
ST    AX, TEMP
LD    AX, D
ADD   AX, E
MUL   AX, TEMP
```

Register AX

Value

TEMP

Value

. . . . . .

(User will press successive enters for step execution of simulation program)

Part 5

Toutorials

5.1. Tutorial 01 – Fundaments of Programming Languages

Programming the art of driving computer according to ones requirement.

Programming has started a long ago; with the change of requirements and technology it has reshaped itself continuously, but by the day one can look back and see what are the components that are mostly adopted by each language for its survival. That is something that can be termed as fundamentals of programming languages.

Program: A set of instruction that are executed on a set of values in a specific sequence to achieve desired task.

Program = Values + Instructions

As our programs have to deal with values so each language should at least provide a mechanism for their storage. The interface provided by the languages for data storage is called *data types*, which is the fundamental of all programming languages. Primarily this data can be divided in to two categories, numeric and character.

Performing different operations at data to convert it into information is one of prime responsibility of each language. Approach provided by the language for this purpose is called *operators*.

A next issue that matters a program chiefly is the sequence of execution of operations over the data. This sequence is controlled by the *control structures* provided by language. In early languages the same task was performed using goto statements, structured programming made gotos out. Control structures have three types in principle

Sequential control structure used to executed each and every operation once and in the order they are written
Selection control structure used to execute an operation or a set of operations when a specific operation returns true value
Iteration control structure used to execute an operation or a set of operations repetitively until a specific operation returns true value.

Last building block for computer languages is the support of *functions*. Function is a set of instructions to perform a specific task in the program. If we have to do the same task at the different places of our program then it is better to make function and call it at those places instead of writing same code again and again. It was not the part of early programming languages but after the origin of procedural language it becomes the fundamental.

If there is some thing else in a programming language then it is a facility to the programmer by the language. Those facilities make programming convenient and versatile, which is the its worth. Today some other facilities can also be treated as fundamental that includes structures/records, and file handling (disk I/O).

5.2. Tutorial 02 – Object Technology

Origin of Object Technology:

Software, like many things, has undergone a huge transformation in last thirty years. In the beginning programs were consider state of the art if they used higher-level language. Development techniques were still rooted in assembly language, which were using gotos. After the discovery of structured programming *Gotos were out and indented control structures were in*, enhancing programs readability and maintainability.

Structured programming promised to solve software crisis by creating small, focused modules, which could be reused in many different programming situations. Unfortunately modules were invariably dependent on specific data types that were defined external to modules. *Modules encapsulated behaviors, but not data.* When programmers made minor changes to these data structures that resides external to the modules, the modules were no longer useful.

Programmers sought a technique for encapsulating both behavior and data. Thus were born objects. It was starting to look as if the age of software maintainability and reuse had actually arrived.

Problems:

Unfortunately, what looked like good idea in the academic world quickly ran into real world problem. It was true that we had encapsulated both data and behavior, allowing both to be managed as a coherent package. In real world data had to be stored, and existing data had to be read in. our make-believe world of simple, encapsulated object methods quickly began to deteriorate. Our method started getting ugly when dealing with database.

Object Oriented database vendors gave us the solution. Throw away your terabyte of existing data, rewrite your millions of lines of code, bring your corporation to a halt for ten years while updating your system, and your problem will be solved.

Software was undergoing other changes as well. Problem domains were becoming distributed. Our concepts of process began as a deck of punch cards, evolved to executable program, to multi processes on a machine communicating with each other, to processes communicating over a homogeneous network, to processes communicating over a corporate-wide network consisting of many manufactures machines. Watching the enormous advances in the Internet over the last ten years, it is clear that we have just barely understood the concepts of distributed software.

Where is this driving the industry? Now the software will consist of components tied into the huge variety of database products. These complements will be distributed over mega-networks. If we think that software was complicated, clearly we are not seen any thing yet.

If the objects are the answer to current problems of the software industry, how do we get our objects back to manageable state? How do we create object that can be tied to any database that can be distributed on any network that can cooperate with other objects written in any programming language.

Who does the objects resolved these problems and up to what extent. What is the state of the art of Object Oriented Technology?

Proposed Solutions:

OMG (Object Management Group)
IDL (Interface Definition Language)
POS (Persistent Object Service)
CORBA (Common Object Request Broker Architecture)

Microsoft
COM (Component Object Model)
DCOM (Distributed Component Object Model)

IBM (International Business Machine)
SOM (System Object Model)
DSOM (Distributed System Object Model)

5.3. Tutorial 03 – Tempaltes in C++

Template The Literal Meaning:

Template is a thing that is used as a model for producing other similar examples.

Motivation

Templates are molds from which compiler can generate a family of classes or functions. The template is one of the C++'s most sophisticated and high-powered features. Templates have come a long way since they were first introduced to the language in 1991.Back then, they were merely clever macros. This feature is most useful in designing and implementing general-purpose libraries like Standard Template Library. Templates are also used to create generic or template classes or functions.

Advantages of Templates

Templates are powerful features of C++.

1. Template is coded only once.
2. Errors caused by multiple coding are reduced.
3. A template presents a uniform solution for similar problems as it allows type-independent code.

Function Template

A function template defines a group of statements for a function using a parameter instead of a concrete type.

Difference Between Function Template & Template Function

Function template is a generic function, which is applicable to any kind of data type and template function is a specific version of function template, which is created for a specific data type.

Why To Use Function Templates

Before the development of templates in the C++, programmers use alternative techniques to implement the generally used algorithms. These techniques include macros, void pointers and common root base. But all of these techniques had certain drawbacks, which are separately discussed as follows.

Macros

Before templates, macros were used for the implementation of generic programming. To some extent, they can be used effectively but they have certain drawbacks as well. The

macros, which are originally preprocessor directives, are in-fact in-lined functions with very limited know-how of scope rules and type checking. There may be several arguments to the macros that should be of the same type, but the compiler whether or not they are. Also, the type of value returned is not specified, so the compiler can't tell if you are assigning it to an incompatible variable. They are difficult to debug once used in a code. They can easily increase the size of a program as each macro function call is in-lined. So when macros are called repeatedly, it can dramatically increase the size of program.

Void Pointers

An alternate of macros is to use generic pointer, void *, which contains the address of any data type. But void pointers also have certain limitations. They are not type-safe and also their repeated function callbacks can cause runtime overhead.

A Common Root Base

In some object-oriented languages, every object is ultimately derived from a common base class but C++ does not force a common root class. Therefore, it is the programmer's responsibility to make it sure that every class is derived from a common base class. Moreover, the algorithms using common root base are not unable to handle fundamental types because they are limited to class objects exclusively. Finally the extensive use of runtime type checking imposes an unacceptable performance on general-purpose algorithms.

Function templates are free from all these drawbacks. They are type-safe, they can be in-lined by the compiler to enhance performance and they are applicable to fundamental types and user defined types equally.

Defining Function Templates

A function template is generically declared using the keyword template. The general form of a template function definition is shown below:

```
template < class T >
return- type  function-name ( parameter list )
{
     body of function
 }
```

which can also be written as

```
template <class T> return-type  function-name ( parameter list )
{
    body of function
}
```

The key factor in function templates is to represent the data type used by the function not as a specific type but by a name that can stand for any type. In the preceding declaration, this place-holder type-name is T but it can be anything like TYPE , R , FOO. The keyword template signals the compiler that a function template is about to be defined. The keyword class, within the angle brackets may be called as type. The variable following the keyword class, (here T), is called the template argument.
Creating Specialization

When the compiler creates a specific version of function template, it is said to have created a specialization. This is also called a generated function.

Instantiating The Function Template

The act of generating a function is referred to as instantiating the function template and each instantiated version of the function is called a template function. This means that a template function is a specific instance of a function template.

Example of A Simple Function Template

The following program defines a template version of the function that finds absolute value of a number and then in main invokes the function with different data types to prove that it works.

```
template < class T >
T abs ( T n )
{
    return ( n < 0 ) ? -n : n ;
}
```

This function can be called as follows

```
abs( 5 ) = 5
abs( -6 ) = 6
abs( 70000 ) = 70000
abs( -80000 ) = 80000
abs( 9. 95 ) = 9. 95
abs( -10.15 ) = 10.15
```

Function Using More Than One Template Arguments

A template statement may include more than one template argument by using a comma-separated list.

```
template < class T1, class T2, class U)
void print ( T1 k, T2 l, U m )
{
```

```
    cout << k << "\t" << l << "\t" << "m \n"
}
```

It can be called as

```
print ( 34, "RAANA",67.98 );
```

Explicitly Overloading A Template Function:

Even though a template function can overload itself when required, but we can do this explicitly also. This is called explicit specialization. If we overload a template function with a hand-written function for a specific data type then that overloaded function will hide the template version of that function for that specific data type.

Example:

```
template < class T >
void print ( Ta, Tb)
{
    cout <<  a  <<  b ;
}

// This overrides the template version of print for int

void print ( int a, int b )
{
    cout <<  a  <<  b;
}
```

Overloading A Function Template

Not only the template functions can be overloaded explicitly but the template specification itself can be overloaded. This can be done by simply creating another version of template that differs from any others in its parameter list.

Example:

```
template < class T >
void print ( T a)     { cout << a ; }

template < class T, class U >
void print ( T a, U b)        { cout << a << b ;}
```

Template Functions Using Standard Parameters

A template function can use in its parameter list a type name parameter as well as the standard parameters, particularly pointers and references. The standard parameter work just like they do with any other function.

template < class T, int n >

But the following restrictions are to be kept in mind which apply to template parameters other than type parameters:

They cannot be modified n i.e. n++ will generate an Error because of changing template parameter. They cannot be floating-point types i.e. template < class T, double d > // Error because of double data type.

Class Template

A class template specifies a class definition using a parameter instead of a concrete type. A class template can provide a generic definition that can be used to define all the algorithms used by that class; however, the actual type of the data being manipulated will be specified as a parameter when objects of that class are created.

Defining Class Template

A class template can be generally defined as

```
template < class T >
class class-name
{
     // class definition
}
```

Here, T is the place-holder type name, which will be specified when a class is instantiated. Once a template class has been created, we can use the following general form to create a specific instance of that class.

class-name < type > object ;

Here, type is the type name of the data that the class will be operating upon. Member functions of a generic class are themselves automatically generic. It is not necessary to use template to explicitly specify them.

Instantiation And Specialization

A class template is not a class. The process of instantiating a class from a class template and a type argument is called template instantiation**.** A template id, that is a template name followed by a list of arguments in angular brackets is called a specialization**.**

Template Arguments

A template can take type parameters. For example

```
template < class T >
class Vector { /*……..*/ }
```

Class Templates And Non-Type Parameters

A template can also take non-types as parameters. The syntax to accomplish this is similar as for normal function parameters. For example

Template < class T , int n > ,

Then, a declaration such as

Stack < double, 100 > ary;

would instantiate at compile time a 100-element **stack** template class named **ary** of **double** values. Moreover, a non-type parameter can have a default argument and the non-type parameter is treated as **constant.** For example

A template can take take a template as an argument. For example

vector < vector < float > > ary ;

When a template is used as an argument, there should be a space between the left two angular brackets. Otherwise the >> sequence is parsed as the right shift operator.

Using Default Arguments With Template Classes:

A template class can have default arguments associated with a generic type. For example,

```
template <class T=int  >
class ary   {//….};
```

Here the type int is used if no other type is specified when an object of class ary is instantiated.

It is also permissible for non-type arguments to take default arguments. The default value is used when no explicit value is specified when the class is instantiated. Default arguments for non-type parameters are specified using the same syntax as default arguments for function parameters.

Explicit Class Specializations:

Explicit specializations of a template class can be created. For example

```
template < class T > class Vector
template < > class Vector < int >    //explicit specialization for int
```

it tells the compiler that an explicit integer specialization of Vector being created. This same general syntax is used for any type of class specialization.

Friendship

A friend of a class template can be a function template or a class template. a specialization of a function template or class template , or an ordinary (non-template)function or class .

Non –Templates Friends

Non-templates friend declaration of a class template look similar to friend declarations of a non-template class. In the following example the class template Vector declares the ordinary function f ( ) and class Thing as its friends.

```
class  Thing;
template < class T > class Vector
{
    public:
    friend void  f ();
    friend class Thing ;
};
```

Each specialization of Vector has a function f ( ) and class Thing as friends. Thus, every member function of class Thing has become a friend of every template class produced from the class template for Vector. Inside a class template for class Vector that has been declared with

```
template < class T > class Vector
```

A friendship declaration of the form

```
friend void f2 ( Vector < T > & )
```

For a particular type T such as float makes function

```
f2 ( Vector < float > & )
```

a friend of Vector < float > only.

Member Fnctions And Templates

Inside a class template, a member function of another class can be declared as a friend of any template class generated from the class template. F or example, inside a class template for class Vector that has been declared with

template < class T > class Vector

a friendship declaration of the form

friend void Y :: dull ( )

makes member function dull a friend of every template class instantiated from the preceding class template.

Inside a class template for class Vector that has been declared as

template < class T > class Vector

a friendship declaration of the form

friend void C < T > :: full (vector < T > &);

for a particular type T such as float makes member function

C <float > :: full (Vector <float > &)

a friend function of only template class Vector < float >.

Specializations

A specialization can be declared as a friend of a class template. In the following example, the class template Vector declares the specializations

C < int > as its friend:

```
template < class T > class C {/……/};
template < class T > class Vector
{
    public:
    //…….
    friend class C < int >;
}
```

which shows that specializations other than int are not friends of Vector.

Template Friends

Inside a class template for class **Vector** that has been declared with

template < class T > class Vector

a second class Z can be declared with

friend class Z < T >;

then when a template class is instantiated with a particular type for T such as int all members of class Z < int > become friends of template class Vector < int >. A friend of a class template can be a template by itself. For example, you can declare a class template as a friend of another class template:

```
template < class U > class D{ //……  };

template < class T > class Vector
{
     public:
     //…..
     template< class U> friend class D;
};
```

every specialization of D is a friend of every specialization of Vector.

Virtual Member Functions:

A member function template should not be virtual. However ordinary member functions in a class template can be virtual. For example

```
template < class T > class A
{
public:
template < class S > virtual void f ( S ) ;                    // error
virtual int g ( ) ;                          // OK
};
```

A specialization of a member function template does not override a virtual function that is defined in a base class. For example

```
class Base
{
public:
  virtual void f (char ) ;
};

class Derived : public Base
```

```
{
public:
template < class T > void f ( T) ;     //does not override B:: f( int )
```

Static Data Members And Templates

In case of non-template class, only one copy of a static data member is shared among all objects of the class and the static data member must be initialized at file scope.

Templates can also have static data members. For example:

```
template < class T > class C
{
      public:
      static T stat;
 };

template < class T > T C < T > :: stat = 5;      // definition
```

A static data member can be accessed as follows:

```
void f ( )
{
      int n = C < int > :: stat;
}
```

Each template class instantiated from a class template has its own copy of each static data member of the class template; all objecyts of that template class share that one static data member. Static data members of template classes must be initialized at file scope. Each template class gets its own copy of the class template's static member functions.

Errors Concerned With Templates

1. If you forget to place the keyword **class** or **typename** before every formal type parameter is an error.
2. If a template is invoked with a user defined class type and if that template uses operators like **==, +, <=,** etc. with objects of that class type, ten those operators must be overloaded otherwise it will cause an error as the compiler generates calls to the appropriate overloaded operator functions despite of the fact that thse functions are not present.

Some Useful Tips

1. Use templates to express algorithms that apply to many argument types.
2. Use templates to express containers.
3. Provide specializations for container of pointers to minimize code size.

4. Always declare the general form of template before specialization.
5. Declare a specialization before its use.
6. Minimize a template definition's dependence on its instantiation contexts.
7. Define every specialization you declare.
8. Use specialization and overloading to provide single interface implementations of the same concept for different types.
9. Provide a simple interface for simple cases and use overloading and default arguments to express less common cases.
10. Debug concrete classes before generalizing to a template.
11. Where necessary, constraint template arguments using a constraint () member functions.
12. Use explicit instantiation to minimize compile time and link time.
13. Prefer template over derived classes when run time efficiency is at premium.
14. Prefer derived class to a template if adding new variants without recompilation is important.
15. Prefer a template to a derived class when no common base can be defined.
16. Prefer a template over a derived class when built in types and structures with compatibility constraints are important.

*A container class is one that operates on a collection of 0 or more objects of a particular type. A linked list is a container class. So is an array.

Conclusion

Templates help to achieve one of the most important goals in programming: the creation of reusable code. Through the use of template classes, such frameworks can be created that can be applied over and over again to a variety of programming situations. By using templates, we are saved from the effort of creating separate implementations for each data type with which we want the class to work. Template functions and classes are becoming commonplace in programming, and this trend is expected to continue. For example, the Standard Template Library defined by C++ is built upon templates.

5.4. Tutorial 04 – File Allocation Table (FAT)

Exploring The Disk

By far the most widely used storage mediums are the floppy disks and the fixed disks (hard disks). Floppy disks and hard disks come in various sizes and capacities but they all work basically in the same way - information is magnetically encoded on their surface in patterns. These patterns are determined by the disk drive and the software that controls the drive.

Although the type of storage device is important, it is the way the stored information is laid out and managed that concerns programmers most. Therefore we would focus our attention on how information is organized and stored on the disk.

The Disk Structure

As most of us know, the disk drives in DOS and Windows are organized as zero-based drives. That is, drive A is drive number 0, drive B is drive number 1, drive C is drive number 2, etc. The hard disk drive can be further partitioned into logical partitions. Each drive consists of four logical parts-Boot Sector, File Allocation Table (FAT), Directory and Data space. Of these, the Boot Sector contains information about how the disk is organized. That is, how many sides does it contain, how many tracks are there on each side, how many sectors are there per track, how many bytes are there per sector, etc. The files and the directories are stored in the Data Space. The Directory contains information about the files like its attributes, name, size, etc. The FAT contains information about where the files and directories are stored in the data space.

When a file/directory is created on the disk, instead of allocating a sector for it, a group of sectors is allocated. This group of sectors is often known as a cluster. How many sectors together form one cluster depends upon the capacity of the disk. As the capacity goes on increasing, so also does the maximum cluster number. Accordingly, we have 12-bit, 16-bit or 32-bit FAT. In a 12-bit FAT each entry is of 12 bits. Since each entry in FAT represents a cluster number, the maximum cluster number possible in a 12-bit FAT is 212 (4096). Similarly, in case of a 16-bit FAT the maximum cluster number is 216 (65536). Also, for a 32-bit FAT the maximum cluster number is 228 (268435456. Only 28 of the 32 bits are used in this FAT). All FAT systems are not supported by all versions of Windows. For example, the 32-bit FAT system is supported only in Win 95 OSR2 version or later. There are differences in the organization of contents of Boot Sector, FAT and Directory in FAT12/ FAT16 system on one hand and FAT32 on the other.

The File Allocation Table

The File Allocation Table (FAT) maps the usage of the data space of the disk. It contains information about the space used by each individual file, the unused disk space and the space that is unusable due to defects in the disk. Since FAT contains vital information,

two copies of FAT are usually stored on the disk. In case one gets destroyed, the other can be used. A typical FAT entry can contain any of the following:

- Unused cluster
- Reserved cluster
- Bad cluster
- Last cluster in the file
- Next cluster number in the file

There is one entry in the FAT for each cluster in the file area. If the value in a FAT entry doesn't mark an unused, reserved or defective cluster, then the cluster corresponding to the FAT entry is part of a file, and the value in the FAT entry would indicate the next cluster in the file.

This means that the space that belongs to a given file is mapped by a chain of FAT entries. Each FAT entry points to the next entry in the chain. The first cluster number in the chain is the starting cluster number in the file's directory entry. When a file is created or extended, a cluster is allocated to the file by searching the FAT for unused clusters and adding them to the chain. Vice versa, when a file is deleted, the cluster that has been allocated to the file is freed by clearing corresponding FAT entries (by setting them to 0). The FAT chain for a file ends with an entry FFFFh in the FAT.

This file occupies cluster number 3, 5, 6 and 8 on the disk. Hence the starting cluster number in the directory entry for the file is 3. Suppose this file is to be loaded into memory then OS would first load starting cluster number-3's contents into memory. To find out the next cluster belonging to this file OS looks at entry number 3 in FAT where it finds a value 5. Therefore, now it loads the contents of cluster number 5 into memory. Once again OS looks at the FAT and finds in entry number 5 a value 6, hence it loads the contents of cluster 6 into memory. This process goes on till the OS finds an entry FFFFh in FAT, which indicates that there are no more clusters belonging to the file. Hence the process stops.

Now that we have understood how the FAT chain is traversed, let's dig a little deeper into the FAT. The entries present in FAT are 12, 16 or 32 bits long depending on the storage capacity of the disk. Though a 12-bit FAT can handle 4096 clusters only 4078 clusters are available for use since some values are reserved. Similarly, for a 16-bit FAT out of the possible 65536 clusters that it can handle only 65518 are available for use.

In a 12-bit FAT three bytes form two entries. The first two entries (0 and 1) in the FAT are reserved for use by the OS. This means that first 3 bytes in a 12-bit FAT, first 4 bytes in 16-bit FAT and first 8 bytes in a 32-bit FAT are not used for storing cluster numbers. Out of these 3 (or 4, or 8) bytes, the first byte is the media descriptor byte and the balance contains the value FFh. These balance bytes remain unused. The media descriptor byte specifies the type of the disk. It typically has a value FDh, F9h, F0h, F8h for a 360 KB, 1.2 MB, 1.44 MB and a hard disk respectively. The contents of a FAT entry are interpreted as shown below.

Values			Meaning
12-bit	16-bit	32-bit	
000h	0000h	0000000h	Cluster available
FF0h-F6h	FFFFh-FFFF6h	FFFFFFFh-FFFFFF6h	Reserved cluster
FF7h	FFF7h	FFFFFF7h	Bad cluster if not part of chain
FF8h-FFh	FFF8h-FFFFh	FFFFFF8h-FFFFFFh	Last cluster of file
Xxx	xxxx	xxxxxxx	Next cluster in file

Meaning of FAT Entries.

As we saw earlier, two identical copies of FAT are maintained on the disk. All copies are updated simultaneously whenever files are modified. If access to a FAT fails due to a read error, the OS tries the other copy. Thus, if one copy of the FAT becomes unreadable due to wear or a software accident, the other copy may still make it possible to salvage the files/directories on the disk.

Here is a program that prints the contents of the first sector of two copies of FAT for a 12-bit or a 16-bit FAT. On similar lines it can be extended to work for a 32-bit FAT.

Each disk contains two copies of FAT. In the function fat_info( ) the starting sector of each copy of FAT is determined. Next, the function read_fat_info( ) is called for reading and displaying contents of each FAT copy. Since each copy contains several entries, we have displayed only the first 16 entries for a 12-bit & 16-bit FAT. The organization of the FAT types is shown in Figure 7.3.

12-bit FAT

8 bits 8 bits 8 bits

E2 E3 O3 E1 O1 O2

16-bit FAT

8 bits 8 bits 8 bits 8 bits

E3 E4 E1 E2 O3 O4 O1 O2

32-bit FAT

8 bits 8 bits 8 bits 8 bits 8 bits 8 bits 8 bits 8 bits

E7 E8 E5 E6 E3 E4 E1 E2 O7 O8 O5 O6 O3 O4 O1 O2

For a 32-bit FAT the seven nibbles (a nibble is a group of 4 bits) E1-E2-E3-E4-E5-E6-E7-E8 form the even entry. Note that the arrangement of these nibbles is E7-E8-E5-E6-E3-E4-E1-E2 because the lower byte is always stored in memory earlier than the higher byte. This means if the value of the 4-byte FAT entry is ABCD, it would be stored as DCBA. The odd entry is represented using the set of nibbles O1-O2-O3-O4-O5-O6-O7-O8. In reality the nibble E8 and O8 don't contribute to the cluster number since each entry in the 32-bit FAT is only 28 bits long.

On similar lines in a 16-bit FAT the four nibbles E1-E2-E3-E4 form the even entry whereas the set O1-O2-O3-O4 form the odd entry. Similarly, the even and odd entries in a 12-bit FAT are formed by E1-E2-E3 and O1-O2-O3 respectively. Picking up the values present in odd or even entries from a 32-bit FAT or a 16-bit FAT a relatively simple job. However, to pick up the values from a 12-bit FAT we have to use bitwise operators to discard one nibble out of a group of 4 nibbles. This is done in our program through the functions getfat_12().

5.5. Tutorial 05 – Huffman Code

Huffman is a coding algorithm presented by David Huffman in 1952. It's an algorithm, which works with integer length codes. In fact if we want an algorithm, which does integer length codes, Huffman is the best option because it's optimal.

We use Huffman for example, for compressing the bytes outputted by lzp. First we have to know the probabilities of them, we use a qsm model for that matter. Based on the probabilities it makes the codes, which then can be outputted. Decoding is more or less the reverse process, based on the probabilities and the coded data, it outputs the decoded byte.

Definition

A *Huffman Code* for an information source **S** over a code alphabet **B** is an instantaneous coding with the shortest possible average length, which we denote by

$$L_{\min}(S)$$

Remark

Every information source must have a Huffman code.

Given a set of messages with probabilities p1≤ p2 ≤ … ≤ pn, the Human code tree is constructed by recursively combining subtrees:

To make the probabilities the algorithm uses a binary tree. It stores there the symbols and their probabilities. The position of the symbol depends on its probability. Then it assigns a code based on its position in the tree. The codes have the prefix property and are instantaneously decodable thus they are well suited for compression and decompression.

The amount of compression that can be achieved by a given algorithm depends on both the amount of redundancy in the source and the efficiency of its extraction.

Huffman compression reduces the average code length used to represent the symbols of an alphabet. Symbols of the source alphabet, which occur frequently, are assigned with short length codes. The general strategy is to allow the code length to vary from character to character and to ensure that the frequently occurring character has shorter codes.

Constructing a binary tree using a simple example set performs Huffman compression. This is done by arranging the symbols of the alphabets in descending order of probability. Then repeatedly adding two lowest probabilities and resorting. This process goes on until the sum of probabilities of the last two symbols is 1. Once this process is complete, a Huffman binary tree can be generated. If we do not obtain a probability of 1 in the last two symbols, most likely there is a mistake in the process. This probability of 1, which forms the last symbol, is the root of the binary tree.

The resultant codewords are then formed by tracing the tree path from the root node to the endnodes codewords after assigning 0s and 1s to the branches.

Properties

1. Short code words for data words, which occur frequently, Longer ones for symbols, which occur rarely
2. Loss-less compression
3. Length of text minimized

Huffman code purpose

There can be many reasons for encoding a message. The first that comes to mind is security. An encrypted message would not be readable as is. But there is also coding for efficient transmission of data. The jpeg compression scheme for images is a common example. Huffman coding is an example of coding for efficient transmission. A shorter code takes less time to transmit.

Operation of the Huffman algorithm

The time complexity of the Huffman algorithm is O(nlogn). Using a heap to store the weight of each tree, each iteration requires O(logn) time to determine the cheapest weight and insert the new weight. There are O(n) iterations, one for each item.

How Huffman Code is an encoding method

Unlike the ASCII code, Huffman code uses a variable- length code to represent characters. Shorter codes are assigned to characters that occur frequently; longer codes as used for characters that are used less often. Also the Huffman codes are not unique and one can't guess the text massage from the code like 0111 and like that.

Steps for Creating a Huffman Code

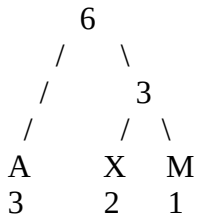
1. For each character, find out the relative frequency ofoccurrence of characters in the language or message.
2. List the characters as leaves of a tree, ordered from highest to lowest frequency.
3. Group the smallest two nodes and designate the sum of the frequencies as the frequency of the parent node.
4. Continue grouping the smallest two leaves or subtrees until all the character leaf nodes are in a single tree.
5. The resultant codewords are then formed by tracing the tree path from the root node to the end nodes codewords after assigning 0s and 1s to the branches.
6. Once the process is complete, a Huffman tree can be generated. If we don't obtain a probability of 1 in the last two symbols, most likely there is a mistake in the process. The probability of 1, which forms the last symbol, is the root of the binary tree.

Example 1

After reading a text file let the frequencies of characters ‘P’, ‘B’, ’F’, ’Z’, ’A’, ’X’ and ’M’ be 18, 16, 5, 4, 3, 2 and 1 respectively. Arranging the relative frequencies in the descending order.

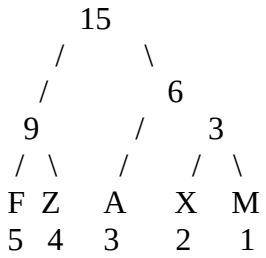
‘P’,	‘B’,	’F’,	’Z’,	’A’,	’X’,	’M’
18,	16,	5,	4,	3,	2	1

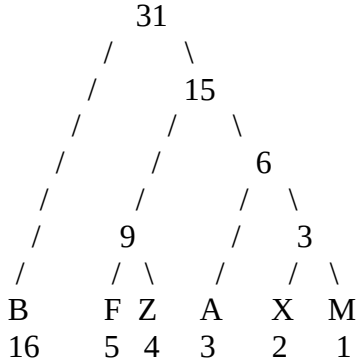
Combine the two nodes with the minimum frequency, we get the shape of tree like that:

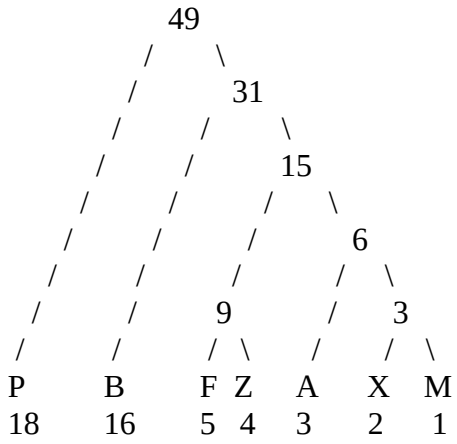


Here a new node is created with weight 3, the nodes X and M will not be under consideration but the new node will be under consideration.

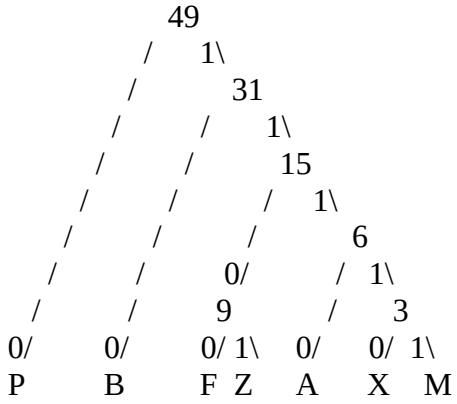
‘A’ will not be under consideration now. Again looking for 2 nodes that have the minimum weight or frequency.







Now the binary tree is build. Next step is to assign 0 to left child and 1 to right child of each node starting from root. In this way the most probable character will get the shortest code.



Code to a character is the path to that character from the root. For example the code for 'B' is 10.

Relative frequency of characters and their code is shown below

Character	'P',	'B',	'F',	'Z',	'A',	'X',	'M'
Frequency	18,	16,	5,	4,	3,	2	1
Code	0	10	1100	1101	1110	11110	11111

- Note that the most used character has the shortest code.
- It does not matter how the characters are arranged. I have arranged it above so that the final code tree looks nice and neat.
- It does not matter how the final code tree are labeled (with 0s and 1s). I chose to label the upper branches with 0s and the lower branches with 1s.
- There may be cases where there is a *tie* for the two least probable characters. In such cases, any tie-breaking procedure is acceptable.
- Huffman codes are not unique.

Encoding the following message

Message text is: PAMBAZ

Message coded is: 01110111111011101101

P	A	M	B	A	Z
0	1110	11111	10	1110	1101

How to decode coded message

Message coded is: 01110111111011101101

Read the code bit by bit, as the bits correspond to some Character, accept that series of bits as that character

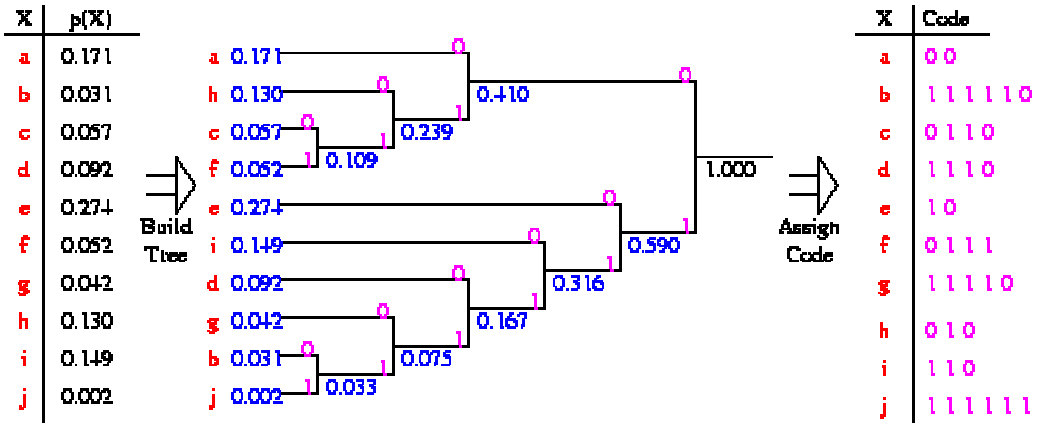
01110111111011101101
P1110111111011101101
PA 111111011101101
PA M 1011101101
PA M B 11011101
PA M B A 1101
PA M B A Z

Example 2

After reading a text file the probabilities of symbols were calculated and binary tree is built according to the table.

X	p (X)
a	0.171
b	0.031
c	0.057
d	0.092
e	0.274
f	0.052
g	0.042
h	0.130
i	0.149
j	0.002

The final static code tree is given below:



Transmission and storage of Huffman-encoded Data

If your system is continually dealing with data in which the symbols have similar frequencies of occurrence, then both encoders and decoders can use a standard encoding table/decoding tree. However, even text data from various sources will have quite different characteristics. For example, ordinary English text will have generally have 'e' at the root of the tree, with short encoding for 'a' and 't', whereas C programs would generally have ';' at the root, with short encoding for other punctuation marks such as '(' and ')' (depending on the number and length of comments!). If the data has variable frequencies, then, for optimal encoding, we have to generate an encoding tree for each data set and store or transmit the encoding with the data. The extra cost of transmitting the encoding tree means that we will not gain an overall benefit unless the data stream to be encoded is quite long - so that the savings through compression more than compensate for the cost of the transmitting the encoding tree also.

What's Wrong with Huffman?

The standard Huffman compression (also known as "static Huffman") has two major drawbacks:

1. The Huffman tree (or at least the symbol probabilities) must be transmitted along with the encoded data (or stored in the compressed file)
2. Encoding requires two passes through the data: To compute the probabilities of the input symbols and To encode the data.

Expansions of Huffman code theory

Adaptive Huffman code enables dynamically change of the code words accordingly to the change of probabilities of the symbols. In this way, the produced code is more effective then the primary Huffman code.

Extended Huffman codes have the characteristic that the coding scheme is coding group of symbols rather than a single symbol.

5.6. Tutorial 06 – Introduction To The Standard Template Library (STL)

1) Using STL can save considerable time and effort and result in higher quality programs.
2) The choice of what Standard Library container to use in a particular application is often based on performance considerations.
3) STL containers are all "templatized" so that you can tailor them to hold the type of data relevant to your particular applications.
4) STL includes many popular data structures as containers and provides many algorithms that programs use to process data in these containers.
5) STL avoids virtual functions in favor of using generic programming with templates to achieve better execution-time performance.

Introduction To Containers

1) STL containers are in three major categories:
   a) Sequence containers.
   b) Associative containers.
   c) Container adapters.
2) The sequence containers and associative containers are collectively referred to as the first-class containers.
3) Four other types are considered "near containers" because they exhibit similar capabilities to the first-class containers, but do not support all the capabilities of first-class containers. They are:
   a) Array.
   b) String.
   c) Bitset (for maintaining sets of 1/0 flag values).
   d) Valarray (for performing high-speed mathematical vector operations).
      i) This class is optimized for computation performance and is not as flexible as the first-class containers.
4) The sequence containers are:
   a) Vector.
      i) A vector provides rapid insertion and deletion at the back of the vector and direct access to any element.
      ii) Vectors support random-access iterators.
   b) Deque.
      i) A deque provides rapid insertion and deletion at the front or back of the deque and direct access to any element.
      ii) Deques support random-access iterators.
   c) List.
      i) A list provides rapid insertion and deletion anywhere in the list.
      ii) Lists support bidirectional iterators.
5) The associative containers are:
   a) Set.
      i) A set provides rapid lookup of a key.

ii) No duplicate keys are allowed.
iii) Sets support bidirectional iterators.

b) Multiset.
i) A multiset provides rapid lookup of a key.
ii) Duplicate keys are allowed.
iii) Multisets support bidirectional iterators.

c) Map.
i) A map provides rapid lookup of a key and its corresponding "mapped" value.
ii) No duplicate keys are allowed (i.e. a one-to-one mapping).
iii) Maps support bidirectional iterators.

d) Multimap.
i) A multimap provides rapid lookup of a key and its corresponding "mapped" values.
ii) Duplicate keys are allowed (i.e. a one-to-many mapping).
iii) Multipmaps support bidirectional iterators.

6) The container adapters are:
a) Stack.
i) A stack provides a last-in-first-out (LIFO) data structure.

b) Queue.
i) A queue provides a first-in-first-out (FIFO) data structure.

c) Priority_queue.
i) A priority_queue provides a first-in-first-out (FIFO) data structure with the highest priority item always at the front of the priority_queue.

7) STL has been carefully designed so that the containers provide similar functionality.
a) There are many generic operations that apply to all containers, and other operations that apply to subsets of similar containers.
b) This contributes to the extensibility of the STL.

8) Common member functions for all STL containers are:
a) Default constructor.
i) A constructor to provide a default initialization of the container.
ii) Normally, each container has several constructors that provide a variety of initialization methods for the container.

b) Copy constructor.
i) A constructor that initializes the container to be a copy of an existing container of the same type.

c) Destructor.
i) Destructor function for cleanup after a container is no longer needed.

d) Empty.
i) Returns true if there are no elements in the container; otherwise returns false.

e) Max_size.
i) Returns the maximum number of elements for a container.

f) Size.
i) Returns the number of elements currently in the container.

g) Operator=
i) Assigns one container to another.

h) Operator<, operator<=, operator>, operator>=, operator== and operator!=

i) Returns the correct boolean value for the comparison of the two containers.

i) Swap.

i) Swaps the elements of the two containers.

9) The following are functions that are found only in first-class containers:
   a) Begin.
      i) The two versions of this function return either an iterator or a const_iterator that refers to the first element of the container.
   b) End.
      i) The two versions of this function return either an iterator or a const_interator that refers to the next position after the end of the container.
   c) Rbegin.
      i) The two versions of this function return either a reverse_iterator or a const_reverse_iterator that refers to the last element of the container.
   d) Rend.
      i) The two versions of this function return either a reverse_iterator or a const_reverse_iterator that refers to the position before the first element of the container.
   e) Erase.
      i) Erases one or more elements from the container.
   f) Clear.
      i) Erases all elements from the container.
10) It is important to ensure that the type of element being stored in an STL container supports a minimum set of functionality including a copy constructor, an assignment operator and--for associative containers--a less than operator.

Introduction To Iterators

1) Iterators have many features in common with pointers and are used to point to the elements of first-class containers (among other usages).
2) Iterators are used with sequences that may be in containers, or may be input sequences or output sequences.
3) Input iterators are used to read an element from a container.
   a) An input iterator can move only in the forward direction (i.e. from the beginning of the container to the end of the container) one element at a time.
   b) Input iterators support only one-pass algorithms.
4) Output iterators are used to write an element to a container. An output iterator can move only in the forward direction one element at a time. Output iterators support only one-pass algorithms.
5) Example of input and output stream iterators:

```
// Fig. 20.5: fig20_05.cpp
// Demonstrating input and output with iterators.
#include <iostream>
#include <iterator>

using namespace std;

int main(){
```

```
    cout << "Enter two integers: ";

    istream_iterator< int > inputInt( cin );
    int number1, number2;

    number1 = *inputInt;  // read first int from standard input
    ++inputInt;           // move iterator to next input value
    number2 = *inputInt;  // read next int from standard input

    cout << "The sum is: ";

    ostream_iterator< int > outputInt( cout );

    *outputInt = number1 + number2;  // output result to cout
    cout << endl;
    return 0;
}
```

6) Forward iterators combine the capabilites of input and output iterators.
   a) Forward iterators retain their position in the container (as state information).
   b) Forward iterators support multi-pass algorithms.
7) Bidirectional iterators combine the capabilities of a forward iterator with the abilities to move in the backward direction.
8) Random-access iterators combine the capabilities of a bidirectional iterator with the ability to directly access any element of the container, i.e. to jump forward or backward by an arbitrary number of elements.
9) The category of iterator supported by each container determines whether that container can be used with specific algorithms in the STL. Containers that support random-access iterators can be used with all algorithms in the STL.
10) Pointers into arrays may be used in place of iterators in all STL algorithms.

Introduction To Algorithms

1) A crucial aspect of the STL is that it provides algorithms that can be used generically across a variety of containers.
   a) The algorithms can be used to manipulate the containers.
   b) There are algorithms to do insertion, deletion, searching, sorting and other manipulations of containers.
2) STL includes approximately 70 standard algorithms.
   a) Mutating-sequence algorithms result in modifications to container elements.
   b) Non-mutating sequence algorithms do not modify the container elements.

Vector Sequence Containers

1) Class vector provides a data structure with contiguous memory locations.
   a) This enables efficient, direct access to any element of a vector via the subscript operator [].
   b) Class vector is most commonly used when the data in the container must be sorted and easily accessible via a subscript.

c) When a vector's memory is exhausted, the vector automatically allocates a larger contiguous area of memory, copies the original elements into the new memory and deallocates the old memory.

2) A vector supports random-access iterators (see fig. 20.10 in textbook).
3) All STL algorithms can operate on a vector.
4) The iterators for a vector are normally implemented as pointers to elements of the vector.
5) You must include the header file <vector>.
6) Example of using a vector class template:
   a) The program declares an instance called v of class vector that stores int values.
      i) When this object is instantiated, an empty vector is created with a size of 0 (the number of elements stored in the vector) and a capacity of 0 (the number of elements that can be stored without allocating more memory to the vector).
      ii) As elements are added to the vector, the capacity doubles each time the total space allocated to the vector is full and another element is added. In this example, the space allocation would go from 0 to 1 to 2 to 4 to 8, etc.
   b) The program demonstrates the use of the size and capacity functions.
   c) The function push_back adds an element to the end of the vector (i.e. the next available position).
   d) The const_iterator used in the printVector function iterates through the vector allowing the function to output its contents.
      i) A const_iterator enables the program to read the elements of the vector but does not allow the program to modify the elements.
   e) The vector member function begin returns a const_iterator to the first element in the vector and the program loops until the p1 iterator reaches the end of the vector (determined by the function end).
   f) The reverse_iterator is used to iterate through the vector backwards.
      i) Functions rbegin (the starting point for iterating in reverse through the container) and rend (the ending point for iterating in reverse through the container) delineate the range of elements to output in reverse.
   g) The code is:

```
// Fig. 20.14: fig20_14.cpp
// Testing Standard Library vector class template
#include <iostream>
#include <vector>

using namespace std;

template < class T >
void printVector( const vector< T > &vec );

int main()
{
   const int SIZE = 6;
   int a[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
   vector< int > v;

   cout << "The initial size of v is: " << v.size()
        << "\nThe initial capacity of v is: " << v.capacity();
```

```
v.push_back( 2 );  // method push_back() is in
v.push_back( 3 );  // every sequence collection
v.push_back( 4 );
cout << "\nThe size of v is: " << v.size()
     << "\nThe capacity of v is: " << v.capacity();
cout << "\n\nContents of array a using pointer notation: ";

for ( int *ptr = a; ptr != a + SIZE; ++ptr )
   cout << *ptr << ' ';

cout << "\nContents of vector v using iterator notation: ";
printVector( v );

cout << "\nReversed contents of vector v: ";

vector< int >::reverse_iterator p2;

for ( p2 = v.rbegin(); p2 != v.rend(); ++p2 )
      cout << *p2 << ' ';

cout << endl;
return 0;
}

template < class T >
void printVector( const vector< T > &vec )
{
   vector< T >::const_iterator p1;

    for ( p1 = vec.begin(); p1 != vec.end(); p1++ )
      cout << *p1 << ' ';
}
```

h) The output of the program is:
The initial size of v is: 0
The initial capacity of v is: 0
The size of v is: 3
The capacity of v is: 4

Contents of array a using pointer notation: 1 2 3 4 5 6
Contents of vector v using iterator notation: 2 3 4
Reversed contents of vector v: 4 3 2

7) Example of vector class template element-manipulation functions:
   a) The following program demonstrates functions that retrieve and manipulate the elements of a vector.
   b) The statement vector<int> v(a, a + SIZE); uses an overloaded vector constructor that takes two arguments.
      i) It creates integer vector v and initializes it with the contents of integer array a from location a up to, but not including, a + SIZE.

c) The statement ostream_iterator<int> output(cout, " "); declares an ostream_iterator called output that can be used to output integers separated by single spaces via cout.

d) The statement copy (v.begin(), v.end(), output); uses the STL algorithm copy to output the entire contents of vector v to standard output.
   i) Algorithm copy copies each element in the container starting with the location specified by the iterator in its first argument up to, but not including, the location specified by the iterator in its second argument.
   ii) The elements are copied to the location specified by the last argument.
   iii) To use algorithms of the STL, you must include the header file <algorithm>.

e) v[0] = 7; and v.at(2) = 10; both act in a similar manner.

f) The statement v.insert(v.begin() + 1, 22); inserts the integer 22 as the second element.
   i) All elements to the right of the insertion point right shift one position to allow the number 22 to be inserted.

g) v.erase(v.begin()); removes the element at the beginning of the array and shifts all remaining elements one position left.

h) v.erase(v.begin(), v.end()); removes all elements from the array.

i) v.clear() also removes all elements from the array.

j) The program code is:

```
// Fig. 20.15: fig20_15.cpp
// Testing Standard Library vector class template
// element-manipulation functions
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
   const int SIZE = 6;
   int a[ SIZE ] = { 1, 2, 3, 4, 5, 6 };
   vector< int > v( a, a + SIZE );
   ostream_iterator< int > output( cout, " " );
   cout << "Vector v contains: ";
   copy( v.begin(), v.end(), output );

   cout << "\nFirst element of v: " << v.front()
        << "\nLast element of v: " << v.back();

   v[ 0 ] = 7;        // set first element to 7
   v.at( 2 ) = 10;    // set element at position 2 to 10
   v.insert( v.begin() + 1, 22 );  // insert 22 as 2nd element
   cout << "\nContents of vector v after changes: ";
   copy( v.begin(), v.end(), output );

   try {
      v.at( 100 ) = 777;   // access element out of range
   }
   catch ( out_of_range e ) {
```

```
        cout << "\nException: " << e.what();
    }

    v.erase( v.begin() );
    cout << "\nContents of vector v after erase: ";
    copy( v.begin(), v.end(), output );
    v.erase( v.begin(), v.end() );
    cout << "\nAfter erase, vector v "
         << ( v.empty() ? "is" : "is not" ) << " empty";

    v.insert( v.begin(), a, a + SIZE );
    cout << "\nContents of vector v before clear: ";
    copy( v.begin(), v.end(), output );
    v.clear();  // clear calls erase to empty a collection
    cout << "\nAfter clear, vector v "
         << ( v.empty() ? "is" : "is not" ) << " empty";

    cout << endl;
    return 0;
}
```

k) The output is:

```
Vector v contains: 1 2 3 4 5 6
First element of v: 1
Last element of v: 6
Contents of vector v after changes: 7 22 2 10 4 5 6
Exception: invalid vector<T> subscript
Contents of vector v after erase: 22 2 10 4 5 6
After erase, vector v is empty
Contents of vector v before clear: 1 2 3 4 5 6
After clear, vector v is empty
```

List Sequence Container

1) The list sequence container provides an efficient implementation for insertion and deletion operations at any location in the container.
2) The list class is implemented as a doubly-linked list.
   a) Class list supports bidirectional iterators that allow the container to be traversed both forwards and backwards.
   b) Any algorithm that requires input, output, forward or bidirectional iterators can operate on a list.
3) Many of the list member functions manipulate the elements of the container as an ordered set of elements.
4) In addition to the member functions of all STL containers and the common member functions of all sequence containers, the list class provides eight other member functions--splice, push_front, pop_front, remove, unique, merge, reverse and sort.
5) The header file <list> must be included to use class list.
6) Example of a list class template:
   a) The following example instantiates two list objects capable of storing integers.

b) The function push_front inserts integers at the beginning of values.
c) The function push_back inserts integers at the end of values.
d) The statement values.sort() uses list member function sort to arrange the elements in the list in ascending order.
   i) Note that this is different from the sort in the STL algorithms.
e) The statement values.splice(values.end(), otherValues); uses the list function splice to remove the elements in otherValues and insert them into values before the iterator position specified as the first argument.
f) After inserting more elements in the list instance otherValues and sorting both values and otherValues, the statement values.merge(otherValues); use the list member function merge to remove all elements of otherValues and insert them in sorted order into values.
g) The list function pop_front removes the first element in the list.
h) The function pop_back removes the last element in the list.
i) The statement values.unique(); removes duplicate elements in the list.
   i) The list should be in sorted order before this operation is performed.
j) The statement values.swap(otherValues); uses function swap to exchange the contents of values with the contents of otherValues.
k) The statement values.assign(otherValues.begin(), otherValues.end()); uses the list function assign to replace the contents of values with the contents of otherValues in the range specified by the two iterator arguments.
l) The statement values.remove(4); uses the list function remove to delete all copies of the value 4 from the list.
m) The code is:

```
// Fig. 20.17: fig20_17.cpp
// Testing Standard Library class list
#include <iostream>
#include <list>
#include <algorithm>


using namespace std;

template < class T >
void printList( const list< T > &listRef );

int main()
{
   const int SIZE = 4;
   int a[ SIZE ] = { 2, 6, 4, 8 };
   list< int > values, otherValues;

   values.push_front( 1 );
   values.push_front( 2 );
   values.push_back( 4 );
   values.push_back( 3 );

   cout << "values contains: ";
   printList( values );
   values.sort();
```

```
   cout << "\nvalues after sorting contains: ";
   printList( values );

   otherValues.insert( otherValues.begin(), a, a + SIZE );
   cout << "\notherValues contains: ";
   printList( otherValues );
   values.splice( values.end(), otherValues );
   cout << "\nAfter splice values contains: ";
   printList( values );

   values.sort();
   cout << "\nvalues contains: ";
   printList( values );
   otherValues.insert( otherValues.begin(), a, a + SIZE );
   otherValues.sort();
   cout << "\notherValues contains: ";
   printList( otherValues );
   values.merge( otherValues );
   cout << "\nAfter merge:\n   values contains: ";
   printList( values );
   cout << "\n   otherValues contains: ";
   printList( otherValues );

   values.pop_front();
   values.pop_back();   // all sequence containers
   cout <<
      "\nAfter pop_front and pop_back values contains:\n";
   printList( values );

   values.unique();
   cout << "\nAfter unique values contains: ";
   printList( values );

   // method swap is available in all containers
   values.swap( otherValues );
   cout << "\nAfter swap:\n   values contains: ";
   printList( values );
   cout << "\n   otherValues contains: ";
   printList( otherValues );

   values.assign( otherValues.begin(), otherValues.end() );
   cout << "\nAfter assign values contains: ";
   printList( values );

   values.merge( otherValues );
   cout << "\nvalues contains: ";
   printList( values );
   values.remove( 4 );
   cout << "\nAfter remove( 4 ) values contains: ";
   printList( values );
   cout << endl;
   return 0;
}

template < class T >
void printList( const list< T > &listRef )
{
```

```
      if ( listRef.empty() )
         cout << "List is empty";
      else {
         ostream_iterator< T > output( cout, " " );
         copy( listRef.begin(), listRef.end(), output );
      }
   }
```

n) The output is:

```
values contains:  2 1 4 3
values after sorting contains:  1 2 3 4
otherValues contains:  2 6 4 8
After splice values contains:  1 2 3 4 2 6 4 8
values contains:  1 2 2 3 4 4 6 8
otherValues contains:  2 4 6 8
After merge:
   values contains:  1 2 2 2 3 4 4 4 6 6 8 8
   otherValues contains:  List is empty
After pop_front and pop_back values contains:
2 2 2 3 4 4 4 6 6 8
After unique values contains: 2 3 4 6 8
After swap:
   values contains:  List is empty
   otherValues contains:  2 3 4 6 8
After assign values contains:  2 3 4 6 8
values contains:  2 2 3 3 4 4 6 6 8 8
After remove( 4 ) value contains:  2 2 3 3 6 6 8 8
```

Deque Sequence Container

1) Class deque provides many of the benefits of a vector and a list in one container.
   a) The term deque (pronounced "deek") is short for double-ended queue.
   b) Class deque is implemented to provide efficient indexed access using subscripting for reading and modifying its elements similar to a vector.
   c) Class deque is also implemented for efficient insertion and deletion operations at its front and back much like a list.
      i) However, a list is also capable of insertions and deletions in the middle of the list.
   d) Class deque provides support for random-access iterators, so deque can be used with all STL algorithms.
2) One of the most common uses of a deque is to maintain a first-in-first-out queue of elements.
3) Additional storage for a deque can be allocated at either end of the deque.
4) Class deque provides the same basic operations as class vector, but adds functions push_front and pop_front to allow insertion and deletion at the beginning of the deque, respectively.
5) Header file <deque> must be included to use class deque.
6) Example of a deque sequence container:

a) The following program demonstrates features of class deque.
b) The statement deque<double> values; instantiates a deque that can store double values.
c) Functions push_front and push_back insert elements at the beginning and end of the deque, respectively.
d) The for structure in the program uses the subscript operator to retrieve the value in each element of the deque for output.
   i) Note the use of function size in the condition to ensure the program does not access an element outside the bounds of the deque.
e) The function pop_front is used to remove the first element of the deque.
f) The statement values[1] = 5.4; uses the subscript operator to create an lvalue.
g) The code is:

```
// Fig. 20.18: fig20_18.cpp
// Testing Standard Library class deque
#include <iostream>
#include <deque>
#include <algorithm>

using namespace std;

int main()
{
   deque< double > values;
   ostream_iterator< double > output( cout, " " );

   values.push_front( 2.2 );
   values.push_front( 3.5 );
   values.push_back( 1.1 );

   cout << "values contains: ";

   for ( int i = 0; i < values.size(); ++i )
      cout << values[ i ] << ' ';

   values.pop_front();
   cout << "\nAfter pop_front values contains: ";
   copy ( values.begin(), values.end(), output );

   values[ 1 ] = 5.4;
   cout <<
      "\nAfter values[ 1 ] = 5.4 values contains: ";
   copy ( values.begin(), values.end(), output );
   cout << endl;
   return 0;
}
```

h) The output is:

```
values contains:  3.5 2.2 1.1
After pop_front values contains:  2.2 1.1
After values[ 1 ] = 5.4 values contains:  2.2 5.4
```

Container Adapters

1) The STL provides three so-called container adapters:
   a) Stack.
   b) Queue.
   c) Priority_queue.
2) Adapters are not first-class containers because:
   a) They do not provide the actual data structure implementation in which elements can be stored and
   b) Adapters do not support iterators.
3) The benefit of an adapter class is that the programmer can choose an appropriate underlying data structure.
4) All three adapter classes provide member functions push and pop that implement the proper method of inserting an element into each adapter data structure and the proper method of removing an element from each adapter data structure.

The Stack Adapter

1) Class stack enables insertions into and deletions from the underlying data structure at one end (commonly referred to as a last-in-first-out data structure).
2) A stack can be implemented with any of the sequence containers: vector, list and deque.
3) Header file <stack> must be included to use a stack.
4) Example of a stack adapter:
   a) The following program creates three integer stacks using each of the sequence containers of the Standard Library as the underlying data structure to represent the stack.
   b) By default, a stack is implemented with a deque.
   c) The stack operations are:
      i) push to insert an element at the top of the stack.
         (1) This is implemented by calling function push_back of the underlying container.
      ii) pop to remove the top element of the stack.
         (1) This is implemented by calling function pop_back of the underlying container.
      iii) top to get a reference to the top element of the stack.
         (1) This is implemented by calling function back of the underlying container.
      iv) empty to determine if the stack is empty.
         (1) This is implemented by calling function empty of the underlying container.
      v) size to get the number of elements in the stack.
         (1) This is implemented by calling function size of the underlying container.
      vi) The code is:

```
// Fig. 20.23: fig20_23.cpp
// Testing Standard Library class stack
#include <iostream>
#include <stack>
#include <vector>
#include <list>

using namespace std;

template< class T >
void popElements( T &s );

int main()
{
   // default is deque-based stack
   stack< int > intDequeStack;
   stack< int, vector< int > > intVectorStack;
   stack< int, list< int > > intListStack;
   for ( int i = 0; i < 10; ++i ) {
      intDequeStack.push( i );
      intVectorStack.push( i );
      intListStack.push( i );
   }

   cout << "Popping from intDequeStack: ";
   popElements( intDequeStack );
   cout << "\nPopping from intVectorStack: ";
   popElements( intVectorStack );
   cout << "\nPopping from intListStack: ";
   popElements( intListStack );

   cout << endl;
   return 0;
}

template< class T >
void popElements( T &s )
{
   while ( !s.empty() ) {
      cout << s.top() << ' ';
      s.pop();
   }
}
```

vii) The output is:

```
Popping from intDequeStack:  9 8 7 6 5 4 3 2 1 0
Popping from intVectorStack:  9 8 7 6 5 4 3 2 1 0
Popping from intListStack:  9 8 7 6 5 4 3 2 1 0
```

Queue Adapter

1) Class queue enables insertions at the back of the underlying data structure and deletions from the front of the underlying data structure (commonly referred to as a first-in-first-out data structure).
2) A queue can be implemented with STL data structures list and deque.
3) By default, a queue is implemented with a deque.
4) The common queue operations are:
   a) push to insert an element at the back of the queue.
   b) pop to remove the element at the front of the queue.
   c) front to get a reference to the first element in the queue.
   d) back to get a reference to the last element in the queue.
   e) empty to determine if the queue is empty.
   f) size to get the number of elements in the queue.
5) Header file <queue> must be included to use a queue.
6) Example of a queue adapter:
   a) The statement queue<double> values; instantiates a queue that stores double values.
   b) The function push adds element to the queue.
   c) The while loop uses function front to get the element from the queue and then invokes the function pop to remove the element from the queue until the queue is empty.
   d) The code is:

```
// Fig. 20.24: fig20_24.cpp
// Testing Standard Library adapter class template
// queue
#include <iostream>
#include <queue>

using namespace std;

int main()
{
   queue< double > values;

   values.push( 3.2 );
   values.push( 9.8 );
   values.push( 5.4 );

   cout << "Popping from values: ";

   while ( !values.empty() ) {
      cout << values.front()
           << ' ';           // does not remove
       values.pop();      // removes element
   }

   cout << endl;
   return 0;
}
```

   e) The output is:

Popping from values: 3.2 9.8 5.4

The Priority Queue Adapter

1) Class priority_queue enables insertions in sorted order into the underlying data structure and deletions from the front of the underlying data structure.
2) A priority_queue can be implemented with STL data structures vector and deque.
3) By default, a priority_queue is implemented with a vector as the underlying data structure.
4) When adding elements to a priority_queue, the elements are automatically inserted in priority order such that the highest priority element (i.e. the largest value) will be the first element removed from the priority queue.
   a) This is usually accomplished by using a sorting technique called heapsort that always maintains the largest value (i.e. highest priority) at the front of the data structure--such a data structure is called a heap.
   b) The comparison of elements is performed with comparator function object less<T> by default, but the programmer can supply a different comparator (e.g. greater<T>, greater_equal<T>, etc.).

5) The common priority_queue operations are:
   a) push to insert an element at the appropriate location based on priority order of the priority_queue.
   b) pop to remove the highest priority element of the priority_queue.
   c) top to get a reference to the top element of the priority_queue.
   d) empty to determine if the priority_queue is empty.
   e) size to get the number of elements in the priority_queue.
6) The header file <queue> must be included to use class priority_queue.
7) Example of a priority_queue adapter:
   a) The statement priority_queue<double> priorities; instantiates a priority_queue that stores double values and uses a deque as the underlying data structure.
   b) The function push adds elements to the priority_queue.
   c) The while loop:
      i) Uses function top to retrieve the highest priority element in the priority_queue.
      ii) Uses the function pop to remove elements from the priority_queue until the priority_queue is empty.
   d) The code is:

```
// Fig. 20.25: fig20_25.cpp
// Testing Standard Library class priority_queue
#include <iostream>
#include <queue>
#include <functional>

using namespace std;

int main()
{
```

```
   priority_queue< double > priorities;

   priorities.push( 3.2 );
   priorities.push( 9.8 );
   priorities.push( 5.4 );

   cout << "Popping from priorities: ";

   while ( !priorities.empty() ) {
      cout << priorities.top() << ' ';
      priorities.pop();
   }

   cout << endl;
   return 0;
}
```

e) The output is:

Popping from priorities: 9.8 5.4 3.2

Algorithms

1) STL algorithms do not depend on the implementation details of the containers on which they operate.A partial list of algorithms:
2) Function to fill container elements or generate container elements:
   a) Functions fill and fill_n set every element in a range of container elements to a specific value.
   b) Functions generate and generate_n use a generator function to create values for every element in a range of container elements.
3) Comparison functions:
   a) Function equal compares two sequences of values for equality.
   b) Function mismatch compares two sequences of values and returns a pair of iterators indicating the location in each sequence of the mismatched elements. If all the elements match, the pair contains the result of function end for each sequence.
   c) Function lexicographical_compare compares the contents of two sequences to determine if one sequence is less than another sequence (similar to a string comparison).
4) Functions to remove values from a sequence:
   a) Functions remove and remove_copy delete all elements in a sequence that match a specified value. Functions remove_if and remove_copy_if delete all elements in a sequence for which the unary predicate function passed to the functions returns true.
   b) Functions replace and replace_copy replace with a new value all elements in a sequence that match a specified value. Functions replace_if and replace_copy_if replace with a new value all elements in a sequence for which the unary predicate function passed to the functions returns true.
5) Mathematical algorithms:

a) Function random_shuffle randomly order the elements in a sequence.
b) Function count counts the elements with the specified value in a sequence. Function count_if counts the elements in a sequence for which the supplied unary predicate function returns true.
c) Function min_element locates the smallest element in a sequence. Function max_element locates the largest element in a sequence.
d) Function accumulate sums the values in a sequence. A second version of the function receives a pointer to a general function that takes two arguments and returns a result. The general function determines how the elements in a sequence are accumulated.
e) Function for_each applies a general function to every element in a sequence. The general function takes one argument (that it should not modify) and returns void.
f) Function transform applies a general function to every element in a sequence. The general function takes one argument (that it can modify) and returns the transformed result.

6) Searching and sorting algorithms:
   a) Function find locates an element in a sequence and, if the element is found, returns an iterator to the element; otherwise, find returns an iterator indicating the end of the sequence. Function find_if locates the first element for which the supplied unary predicate function returns true.
   b) Function sort arranges the elements in a sequence in sorted order (ascending by default or in the order indicated by a supplied binary predicate function).
   c) Function binary_search determines if an element is in sorted sequence.
7) Element swapping functions:
   a) Function swap exchanges two values.
   b) Function iter_swap exchanges two values referred to by iterators.
   c) Function swap_ranges exchanges the elements in one sequence with the elements in another sequence.
8) Copying and merging:
   a) Function copy_backward copies elements in a sequence and places the elements in another sequence starting from the last element in the second sequence and working toward the beginning of the second sequence.
   b) Function merge combines two sorted ascending sequences of values into a third sorted ascending sequence. Note that merge also works on unsorted sequences, but does not produce a sorted sequence in that case.
      i) A back_inserter argument uses the container's default capability for inserting an element at the end of the container. When an element is inserted into a container that has no more elements available, the container automatically grows in size.
      ii) There are two other inserters--front_inserter and inserter. A front_inserter inserts an element at the beginning of a container (specified in its argument) and an inserter inserts an element before the iterator supplied as its second argument in the container supplied as its first argument.
   c) Function unique removes all duplicates from a sorted sequence.
   d) Function reverse reverses all the elements in a sequence.

e) Function inplace_merge merges two sorted sequences of elements in the same container.
f) Function unique_copy makes a copy of all the unique elements in a sorted sequence.
g) Function reverse_copy makes a reversed copy of the elements in a sequence.

9) Set operations:
   a) Function includes compares two sorted sets of values to determine if every element of the second set is in the first set. If so, includes returns true; otherwise, includes returns false.
   b) Function set_difference determines the elements from the first set of sorted values that are not in the second set of sorted values (both sets of values must be in ascending order using the same comparison function).
   c) Function set_intersection determines the elements from the first set of sorted values that are in the second set of sorted values (both sets of values must be in ascending order using the same comparison function).
   d) Function set_symmetric_difference determines the elements in the first set that are not in the second set and the elements in the second set that are not in the first set (both sets of value must be in ascending order using the same comparison function).
   e) Function set_union creates a set of all the elements that are in either or both of the two sorted sets (both sets of values must be in ascending order using the same comparison function).
10) Lower bounds, upper bounds and equal ranges:
   a) Function lower_bound determines the first location in a sorted sequence of values at which the third argument could be inserted in the sequence and the sequence would still be sorted in ascending order.
   b) Function upper_bound determines the last location in a sorted sequence of values at which the third argument could be inserted in the sequence and the sequence would still be sorted in ascending order.
   c) Function equal_range returns a pair of forward iterators containing the combined results of performing both a lower_bound and an upper_bound operation.
11) Heapsort:
   a) Heapsort is a sorting algorithm in which an array of elements is arranged into a special binary tree called a heap. The key features of a heap are that the largest element is always at the top of the heap and the values of the children of any node in the binary tree are always less than or equal to that node's value. A heap arranged in this manner is often called a maxheap.
   b) Function make_heap takes a sequence of values and creates a heap that can be used to produce a sorted sequence.
   c) Function sort_heap sorts a sequence of values that are already arranged in a heap.
   d) Function push_heap adds a new value into a heap. push_heap assumes that the last element currently in the container is the element being added to the heap and that all other elements in the container are already arranged as a heap.
   e) Function pop_heap removes the top element of the heap. This function assumes that the elements are already arranged as a heap.
12) Minimum and maximum:

a) Function min determines the minimum of two values.
b) Function max determines the maximum of two values.

Class Bitset

1) Class bitset makes it easy to create and manipulate bit sets.
   a) Bit sets are useful for representing a set of boolean flags.
   b) Bitsets are fixed in size at compile time.

Bibliography

Deitel, H.M. and P.J. Deitel, *C++ How to Program, Second Edition*, Eaglewood Cliffs, New Jersey: Prentice Hall, 1998.

Part 6

Programing Problems

6.1. Vehicle parking lot system:

You are required to model a vehicle parking lot system. The parking lot has a facility to park cars and scooters. The parking lot contains four parking lanes-two for cars and two for scooters.

Each lane can hold ten vehicles. There is an operator with a console at the East end of the parking lot. The system should be able to handle following scenarios.

Arrival of a vehicle:

4. Type of vehicle (car or scooter) and Registration No. of vehicle should be entered
5. Program should display suitable parking slot
6. Vehicles arrive at East end, leave from West end

Departure of Car:

1. If not western most, all cars should be moved to the west
2. When the car is driven out, rest of the cars should be moved to the left
3. Vehicle data should be updated

Departure of Scooter:

1. Scooters should be able to drive out at random
2. Vehicle data should be updated

Note that when desired the operator must be able to obtain information like number of vehicles, number of scooters or number of cars currently parked in the parking lot. Also, the system should be able to display all the parking lots (currently occupied) if desired.

6.2. Working of lifts in a multi-story building:

A program that demonstrates how an array can be used to show working of lifts in a multi-storeyed building.

A 30- storeyed building has got about 5 wings where there would be a lift in each of the wing. You have to make available a lift to the person who presses a button to get a lift. Follow the steps given below for designing the program.

1. The floors of the building should be numbered as 0 - 29.
2. The lifts should be numbered as 0 - 4.
3. Display a menu that would have following options
    a. Do you wish to use a lift?
    b. Show lift status
    c. Exit

If user selects, option 1 then get following information from the user.

1. Get the floor number where the person is standing
2. Whether user wishes to go up/down.
3. Get the floor number where the user wishes to go.

The validation checks should be there for the data that user enters.
The validation checks are as given below.

1. The floor number where the user is standing and the floor number where user wishes to go should be in a range of 0 to 29.
2. If the user is standing on ground floor, i.e. 0th floor, then selecting the direction for the lift to go down would be invalid.
3. If the user is standing on topmost floor, i.e. 29th floor, then selecting the direction for the lift to go up would be invalid.
4. The direction for the lift whether up or down should be entered as 'u' or 'd' only.

Check for the lift that can be made available to the person. Before making a lift available to the user consider the following.

1. The lift, which is nearer to the user, i.e., the floor where he is standing should be made available.
2. If the user is standing on a floor, say 8th floor for example. And there are two lifts one on the 7th floor and the other on the 9th floor. Both the lifts are nearer to the user. In such a situation, first check where the user wishes to go, up/down? If up then the lift on the 7th floor should be made available. And if down then the lift on 9th floor should be made available.
3. If the user is standing on say 4th floor for example. And there are 3 lifts, lift number 0, 2, and 4, standing on the 0th floor. All the three are nearer to the user. In such a situation lift that comes first in the order should be made available.

6.3. Acrostic:

Write a program that checks if given set of strings form an acrostic. The set of strings is said to form an acrostic when the strings read in any direction come in same order. For example,

ROTAS
OPERA
TENET
AREPO
SATOR

the above strings if read horizonatally from left to right and right to left, or vertically from top to bottom and from bottom to top come in same order.

6.4. Dictionary using linked list

Write a program to simulate a dictionary using linked list. It should be a menu driven program with the options for adding a word and its meanings, searching a word and displaying the dictionary. Steps to develop the program are as given below:

1. Declare a structure with the fields as

- a word,
- meaning of a word
- counter that holds the number of meanings
- link to the next node.

Each word added to the list can have maximum 5 meaning(s). Hence, variable used to store meaning(s) of a word would be a two-dimensional character array.

2. The program should have following menu.

- Add a word
- Search for a word
- Show dictionary
- Exit

6.5. 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.

6.6. 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:

6.7. City Parking Garage

Suppose City Parking Garage contains 10 parking lanes, each with a capacity to hold 10 cars at a time. As each car arrives/departs, the values A/D (representing arrival /departure) is entered along with the car registration number. If a car is departing the data should get updated. If a new car is arriving then on the screen a message should be displayed indicating suitable parking slot for the car. Cars arrive at the south end of the garage and leave from the north end. If a customer arrives to pick up a car that is not the nothernmost, all cars to the north of the car are moved out, the car is driven out, and the other cars are restored in the same order that they were in originally. Whenever a car leaves, all cars to the south are moved forward so that at all times all the empty spaces are in the south part of the garage. Write a program that implements this parking system.

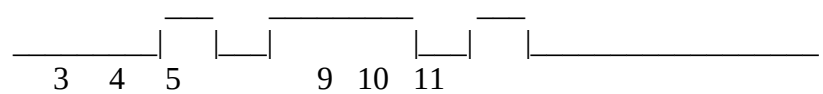
6.8. Files with duplicate names

When the hard disk is new the files and directories are properly organized. As the hard disk grows old, some laxity sets in and one tends to create files with duplicate names in different directories. Write a program to locate these duplicate filenames.

6.9. Boolean OR of two digital signals:

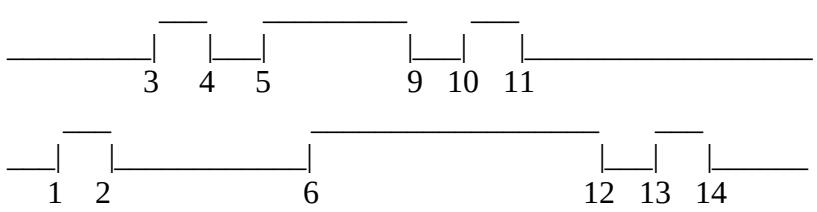
A signal has two states, a high state and a low state. It is given as an array of positive integers in ascending order. Each pair of integer indicates the range of the signal in a high state. The array size will be even, so that the signal starts with a low state and ends with a low state.

A signal
3 4 5 9 10 11
would look like



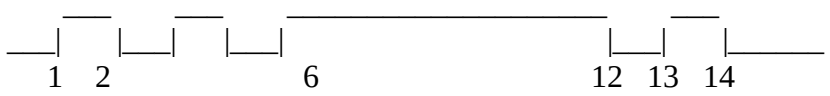
So given two signals

3 4 5 9 10 11
1 2 6 12 13 14



The boolean OR is

1 2 3 4 5 12 13 14



The input will be two signals each with exactly five high states So each signal would be ten positive integers seperated by a space. A number would not be duplicated in the same signal. A number used in the first signal would not be used in the second signal. Output should be a signal which is the boolean OR of the two input signals.

Sample Input

2 5 6 10 20 23 38 45 51 52
1 4 7 9 14 19 21 24 30 31

Output

1 5 6 10 14 19 20 24 30 31 38 45 51 52

3 4 5 12 13 14

6.10. Knapsack problem:

Given an integer K and n item of different sizes such that the ith item has an integer size ki, find a subset of the items whose sizes sum to exactly K, or determine that no such subset exists.

Input :

2, 3 ,5, 6

Try for K=4 K=8 ...

6.11. Stuttering subsequence problem:

Given two sequences A,B, find the maximal value of i such that B raised to i is a subsequence of A.

6.12. Kth smallest element:

Given a sequence of n elements and an integer k such that 1 <= k <= n , find the kth smallest element.

6.13. Pattern of an odd matrix:

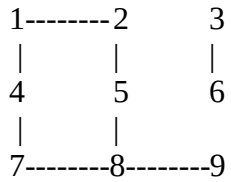
Write a program to print out a particular pattern from a given odd matrix.

Sample Input

for e.g., if the given matrix is 3*3 and is as follows:

1 2 3
4 5 6
7 8 9

With the pattern, the matrix looks like this.



Output:

5 2 1 4 7 8 9 6 3

6.14. Superimposing of matrix:

Superimposing one matrix on another and finding the result:

A matrix (say a) is superimposed on another matrix (say b) such that the middle element of a is placed on an element (say position(i,j) of b. The sum of the products of elements of a and b (except the product of the element(i,j) of b with the middle element of a) is placed at
(i,j) of the resultant matrix. This process repeats for each and every of b. The resultant matrix is would be as shown in sample output.

Sample Input

5 (matrix b)
1 2 1 2 3
1 0 0 3 1

3 4 1 0 2
2 1 2 1 3
0 0 2 4 1

3 (matrix a)
1 1 0
0 1 1
1 0 1

Sample Output

2 2 5 4 3
5 7 10 7 5
6 6 2 10 5
4 11 10 7 6
2 5 7 4 4

6.15. Set operations:

Sample Input

3 lines containing the no. and elements of a set.
The next 4 lines contain the set operations to be performed.

(eg.) 6 1 2 3 4 5 6
4 1 5 6 7
3 1 4 5
A*B
A+B
B*C
A-C

Sample Output

4 lines giving the number of elements in the resultant set due to each operation

(e.g.) 3
7
2
3

6.16. Largest contiguous sum of an array:

Write a program that finds the largest contiguous sum of an array of numbers given as input to your program. The program should first accept the number of elements in the array as the input. Then it should accept the elements of the array. The output should be the largest contiguous sum.

The input to the program will be as follows.(Each on a separate line).

<number of elements in the array>
<element 0>
<element 1>
.
.
<element n-1>

The output should be in the following format.(Each on separate line).

<The largest contiguous sum>

Sample Input

8
100
-200
-300
40
400
-90
999
234

So here the number of elements in the array is 8. The elements are the next 8 numbers given as input. So here it is clear that the largest contiguous sum is the sum of the elements 40,400,-90,999,234

Sample Output

1583

6.17. Largest sum in a square matrix:

Given the order of a square matrix and the elements as inputs, write a program that calculates the largest sum possible along a row, a column or a diagonal. The output is the largest sum out of all these possible combinations. The first input to your program will be the order of the square matrix. Then the elements will be fed in the row major order. i.e. all the columns of a row are to be completely filled before proceeding on to the next row. Note that the diagonals can be not only from Right hand top to Left hand bottom and from left hand top to right hand bottom but also along the subsidiary diagonals.

Sample Input

```
   1   2  -3
-2   4    9
-5   9    8
```

column/row sums are easily understood but for the consideration of the diagonal sums ..

-5; -2 + 9; 1+4+8; 2+9; -3; 1; 2 + (-2); -3 + 4+(-5); 9+9; 8;

The input to the program will be as follows.(Each on a separate line).

<order of the square matrix>
<element [0][0]>
<element [0][1]>
.
.
<element [0][n-1]>
<element [1][0]>
<element [1][1]>
.
.
<element [1][n-1]
.
.
<element [n-1][0]>
<element [n-1][1]>
.
.
<element [n-1][n-1]

The output should be in the following format.(Each on separate line).

<the largest sum>

Sample input for the above matrix problem:

```
3
1
2
3
-2
4
9
-5
9
8
```

Here calculate all the possible sums along rows e.g. 1+2+3, etc , along columns e.g. (-3)+9+8 etc, and along diagonals 9+9 etc. The output should be the maximum of all these.

Expected output to the example input:
18

6.18. Patients in a hospital:

Write a program for maintaining the appointments of patients in different departments of a hospital. Assume that each patient has a unique integer-identifier(id) and so also every department. A patient can seek appointments to one or more departments and is also allowed to cancel an appointment from any department. The program should be able to list out at any time the number of patients who have valid appointments with a given department or the number of departments with which a given patient has appointments.

Input

The input to your program is a sequence of commands each command on a separate line. There are four commands as follows:

1) S patient-id department-id
   Example
   S 3013 1023
   (Patient with id 3013 is seeking appointment from department with id 1023)

2) C patient-id department-id
   Example:
   C 10293 0112
   (Patient with id 10293 is cancelling his appointment from department with id 0112)

3) D department-id
   Example
   D 23

Output

Print the total number of patients who have valid appointments with the department having the id 23. Cancelled appointments should not be counted. Terminate your output by EOLN. If there are no appointments for the department then your program must output 0 followed by an EOLN. The output must not have any sign and there should be no leading zeroes in the output.

4) P patient-id
   Example:
   P 101

Print the total number of departments with which the patient with id 101 has valid appointments. Cancelled appointments should not be counted. Terminate your output by an EOLN. If the patient does not have any valid appointment with any department then your program must output 0 followed by an EOLN. The output must not have any sign and there should be no leading zeroes in the output.

Your program must be capable of handling any number of departments and any number of patients. You may assume that there are no receding blanks before the command character and that erroneous input will not be given to your program. Each command line is terminated by EOLN and the input is terminated by an EOF.

Sample Input

S 100 200
S 100 201
S 100 203
S 101 203
S 101 240
C 100 200
S 101 207
C 101 240
P 100
D 203
D 200
S 100 200
S 101 231
D 200
S 102 245
D 203
P 104

6.19. Set Sets:

A set can be implemented using arrays. Write a program to implement the basic set operations on sets of the following specifications. The set may hold maximum of 100 elements. The elements of the sets are integers. The operations to be performed on these sets are

1. Intersection denoted by the letter I
   An intersection of sets A and B yields a set whose elements are both in A and B
2. Difference denoted by the letter D
   A difference B yields a set whose elements are the elements of A not in B

The elements in a set will be given as a sequence of integers on a line and EOLN will mark the end of the sequence.

The input consists of 3 lines

line 1: elements of set A
line 2: elements of set B
line 3: set operator(I or D)

The output should consist of the result of operation

A <operator> B

The elements of the resultant set must be output on a line separated by blanks and without any leading zeroes. The line must be terminated by an EOLN.

Note that the elements of the input sets will be given in non-decreasing order. The elements in the output must also be in non-decreasing order.

Sample Input

```
 0 23 456 765 1000 5678 7654 8001 10000
23 234 543 765 1235 10000 12347
   I
```

Sample Output

```
 23       765      10000
```

Sample Input

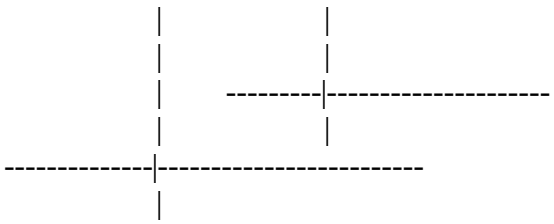
```
 -10 0 15 25 125
 -15 -10 0 20
  D
```

Sample Output

```
  15 25 125
```

6.20. Crossings lines:

Given a set of vertical and a set of horizontal line segments, in the format specified below, write a program to find the total number of crossings amongst the line segments. (See figure below).



Input

The input to your program will be as follows:

line 1: an integer m {number of horizontal line segments}
line 2: an integer n {number of vertical line segments}
line 3: x11 x12 y1 x21 x22 y2 ...xm1 xm2 ym
{3m integers specifying the horizontal lines}
line 4: y11 y12 x1 y21 y22 x2 ... yn1 yn2 xn
{3n integers specifying the vertical lines}

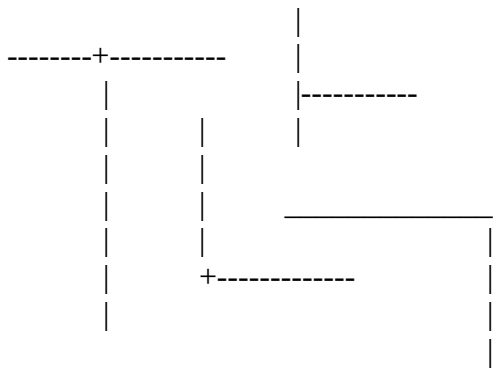
In line3, three integers define a line segment

Example:- x11 x12 and y1 define a horizontal line segment with
endpoint co-ordinates (x11, y1) and (x12, y1).

Similarly in line 4, y11 y12 and x1 define a vertical line segment with endpoint co-ordinates (x1, y11) and (x1, y12).

Assume that the maximum number of horizontal line segments is 20. Similarly the maximum number of vertical line segments is 20.

Note that if two line segments merely touch each other and do not cross then it is not a valid crossing. For example the following are not valid crossings.



Sample Input

3
2
1 10 3 30 2 5 4 20 6

10 1 5 25 30 100

Sample Output

3

Sample Input

1
1
10 20 10
20 15 15

Sample Output
0

6.21. Saddle point:

A "saddle point" of a matrix A is defined as the element A[i,j] such that A[i,j] is the smallest element in row "i" of the matrix and the largest element in column "j" of the matrix. Given a 9x9 matrix whose elements are non-negative integers, you have to output its saddle point if it exists or -1 (minus one) if the saddle point does not exist. The saddle point if it exists, is always unique.

Assume that the rows and the columns of the matrix are numbered from 1 to 9. The input consists of the 9x9 matrix ie. there are 9 lines with 9 integers on each line. If the given matrix has a saddle point, you have to output three integers on a line terminated by EOLN. The three integers, in order, are the saddle point, the row index and the column index of the saddle point.

Integer numbers on the same line must be output separated by blanks and without any leading zeroes. If the given matrix does not have a saddle point, you must output -1 on a line terminated by EOLN.

Sample Input

2 3 4 5 6 7 8 9 10
4 5 6 7 8 9 10 11 12
8 9 10 11 12 13 14 15 16
2 3 4 5 6 7 8 9 10
3 5 6 7 8 9 10 11 12
2 3 4 5 6 7 8 9 10
7 5 6 7 8 9 10 11 12
2 3 4 5 6 7 8 9 10
6 3 4 5 6 7 8 9 10

Sample Output

8 3 1

In the above matrix, the integer 8 at position [3,1] is the saddle point, as per the definition of the saddle point given above.

6.22. Left circular shifts:

Consider the string 'abbacd'. Strings obtained by all possible left circular shifts of this string are given below:

Index of string (Value of k) String Obtained
0abbacd
1bbacda
2bacdab
3acdabb
4cdabba
5dabbac

The index of the string is the number of left circular shifts that have to be applied on the original string to obtain this string. The sorted form of the above strings is given below:

Index of string(value of k)String
0abbacd
3acdabb
2bacdab
1bbacda
4cdabba
5dabbac

Input Format:

You will be given the length of the string (i.e., number of characters in the string) followed by the characters in the string, each in a different line.

<Number of Characters n>
<character 0>
<character 1>
. . .
. . .

<character n-1>

Input Example:

The input corresponding to the above example will be

6
a
b
b
a
c
d

Output Format:

You are required to output the indices of the sorted strings, each in a different line. For example, the output corresponding to the above example will be :

0
3
2
1
4
5

Note that you are required to output the indices of the strings only, and not the strings themselves.

6.23. Permutations of a string:

Write a program to list all the permutations possible for a given input string. The characters fed to the program will be from the sets a-z and A-Z only. Note that 'a' and 'A' are different (case-sensitive).

The input to the program will be as follows.
<a string of characters>

The output should be in the following format.
<all possible permetuations each on a separate line>

Sample Input:

abca

Give all the permutations of this string as the output

Sample Output

abca
abac
acba
acab
aabc
aacb
baca
baac
bcaa
caba
caab
cbaa

6.24. String pattern 2-D matrix:

Write a program to count the number of times a pattern string appears in a 2-D matrix, horizontally, vertically, or diagonally from top to bottom.

The input to the program consists of m, n (size of the matrix), and then the matrix itself. The next line consists of the pattern string, which is to be searched in the matrix.
The output consists of the number of times the pattern string appeared in the matrix horizontally, vertically, or diagonally.

Sample Input

4 4
isti
siti
isis
otsa
is

Sample Output

7

6.25. Expression Evaluation:

In a given line of text, all the words starting with 'r' or 'R' should be reversed and the entire line is output.

Sample Input

this is reverse program

Sample Output

this is esrever program

6.26. Expression Evaluator:

In a given line of input n non-negative integers and (n-1) arithmetic operators from the set (+, -, *) are given on a single line as follows:

<num1><num2>...<numn><op1><op2>...<opn-1>

Write a program to evaluate the expression formed left to right as:

<mun1><op1><mun2><op2>...<mun(n-1)><op(n-1)><mun(n)>

In the above mun1 is obtained by reversing the digits of num1, mun2 by reversing the digits of num2 and so on.

Thus 241 56 * has to be evaluated as 142 * 65.

Input:

The input will consist of 2 lines, each terminated by EOLN.

Line 1: n, the number of integers
Line 2: a sequence of n integers (with reversed digits) followed by (n-1) arithmetic operators.

There is exactly one space separating any two items on the second line (integers or operators), and there are no leading spaces either. The range of values for 'n' is from 2 to 10.
The operators used are '+' meaning addition. '-' meaning subtraction. '*' meaning multiplication.

Output:

Your output must be an integer without any leading zeroes and with a sign only if the result is negative.The output must be terminated by an EOLN.

Sample Input

```
3
12 03 123 + +
```

Sample Output

```
372      { note: 21 + 30 + 321 = 372 }
```

Sample Input

```
4
54 0 50 94 * + -
```

Sample Output

```
-44      {note: 45 * 0 = 0; 0 + 05 =5; 5 - 49 = -44 }
```

Sample Input

```
3
21 3 5 + *
```

Sample Output

```
75       {note: 12 + 3 = 15; 15 * 5 = 75 }
```

6.27. Cards:

A pack of 52 playing cards has four suites (Hearts, Spades, Clubs and Diamonds) with 13 cards in each suite. A card is represented by a letter {h,s,c or d} denoting each of the four suites (Hearts, Spades, Clubs and Diamonds) respectively. The letter is immediately followed by a number from 1 to 13 denoting the particular card in that suite. Thus, the four of Hearts is represented by h4, Ten of Spades by s10 and so on.

A "4-sequence" is a set of four cards of the same suite in consecutive order, e.g. {c10,c11,c12,c13}. Wrap-around sequences like {d11,d12,d13,d1}, {d13,d1,d2,d3} etc. are not valid. A "3-trail" is a set of three cards of the same number but belonging to different suites, e.g. {c6,h6,s6}, {d1,c1,s1}, etc.

Given a set of 13 distinct cards you have to determine if the given set contains a 4-sequence or a 3-trail. Assume that the given set will contain a single 4-sequence or a single 3-trail or neither. That is, it will not contain both.

The input consists of the 13 cards on one line, each card represented as explained above. Each "card" is separated from the other by a single space. There are no leading spaces before the first card and no trailing spaces after the last card.

Output Specification:

If the input set contains a 4-sequence, output the cards that form the 4-sequence in increasing order of the card numbers separated by one or more spaces terminated by an EOLN. If the input set contains a 3-trail, output the cards that form the 3-trail in the order of {h,s,c,d} separated by one or more spaces terminated by an EOLN. If the input set

contains neither of the two, then output "none" on a line. The output must be in lower-case letters only.

Sample Input

h1 d7 s2 d8 d3 h13 d12 h11 d5 d6 h2 s1 h10

Sample Output

d5 d6 d7 d8

Sample Input

h1 h2 h10 h11 h13 s1 s2 d1 d3 d5 d6 d7 d12

Sample Output

h1 s1 d1

Sample Input

h12 d7 s2 c1 d1 h13 d2 h11 d5 d6 h9 s11 c2

Sample Output

s2 c2 d2

Sample Input

s1 h2 s3 h4 s5 h6 s7 h8 s9 h10 c11 d12 c13

Sample Output

none

6.28. Words Frequency:

Given a piece of text, find the frequency distribution of word lengths (ie. how many words have one letter, how many have two letters,...). Find the word-length having the highest frequency and print that word-length followed by its occurrence count. In case two or more word-lengths have the highest frequency, print the largest word-length and its occurrence count.

A "word" is made up of alphanumeric characters and delimited by white-spaces or period (ie. full stop). The first line of input specifies the number of lines of text to follow (say N). Each word in the piece of text can be assumed to be smaller than 15 characters. Your

output will be two integers as above, separated by one or more spaces with no leading zeroes and terminated by an EOLN.

Sample Input
2

This is a sample text.
The text has sentences and sentences have words.
Some words are small. Some words are big.

Output:

4 6

6.29. Anagrams of a word:

An anagram of the word x is the word formed by the characters of the word x with each letter being used only those many times as in word x. For example, post, opst, tops are some anagrams of the word 'stop'.

Input to the program will be the word (less than 8 chars) terminated by EOLN.

Output of the program should be the list of all anagrams of the input word; alphabetically sorted with one word per line. The upper case letters are lower than the lower case letters. Terminate the last word also with eoln.

Sample Input

tea

Sample Output

aet
ate
eat
eta
tae
tea

6.30. Sets and Characters:

Write a program to accept a finite set of alphabets and all its nonempty subsets in lexicographic order. The set of alphabetic symbols will have at most eight chars. Note that the no. of such non-empty subsets is equal to (2^n - 1), where n is the no. of elements in the given set.

For example, given a set {a,b,c}, the list of its non-empty subsets is
{a},{b},{c},{a,b},{a,c},{b,c},{a,b,c}.

Input Format:

The input will consist of two lines.
First line indicating the no. of elements in the input set say n.
Second will have a string of n non-blank alphabetical characters forming a set. Note that the characters themselves will be given in alphabetical order.

Output Format:

The output must list each of the required subsets on a separate line (i.e. the no. of lines in the output must be equal to the no. of non-empty subsets of the given set.) The characters in each line, also the subsets themselves should be in alphabetical order.

Sample Input

```
3
abc
```

Sample Output

```
a
ab
abc
ac
b
bc
c
```

6.31. Sorted List:

Write a program to create a linked list of numbers in sorted (ascending) order and then to delete some specified elements from the list.

Input/Output Specification:

Input to your program is a positive integer N on a line followed by N numbers in the next line. You have to build the sorted list for these N numbers. Next line in the input is another positive integer M followed by M integers in the next line. These M integers are a subset of N integers given earlier. You must delete these elements from the linked list. Next line in the input is a positive integer K. Your program should print the Kth element of the sorted list.

Sample Input

```
8
2 5 9 2 3 1 8 4
2
2 5
5
```

Sample Output

```
8
```

6.32. Round Robin Scheduling Problem:

One of the scheduling techniques that multitasking operating systems use to allocate the CPU amongst running processes is the Round Robin Scheduling Technique. Write a program in to simulate the technique as described below.

In this technique, the running processes are arranged in a queue and are scheduled sequentially, each for a time-slice of fixed duration. If a process is unable to finish in the given time-slice, it is suspended and placed at the end of the queue and next process in the queue is scheduled. When a process finishes execution, it is removed from the queue. If a process finishes execution before exhausting the given time-slice, the remaining time in the time-slice is used to create the time-slice for the next process in the queue. If the total time of the simulation finishes while a process is executing, the time for which the process ran in that time slice is deducted from the time entry of the process and the process is retained as the first process of the queue.

Input Specification:

The input to the program will be an integer(<num>), indicating the number of processes in the queue, followed by <num> pairs of integers (<pid> <time>), with the first integer specifying the process id and the second integer specifying the number of time units required by the process to complete execution. These pairs will be followed by an integer (<len>), indicating the number of time units making up one time-slice in the schedule. The total duration of the simulation (<total>) (in time units) is specified as the last integer. Each line in the input is terminated by a newline.

Input Structure:

<num>
<pid> <time>
<pid> <time>
...
<len>
<total>

Output Specification

The output of the program should be a series of integer pairs denoting the state of the processes in the queue, with the first integer indicating the process-id and the second integer indicating the remaining time of the process. Each pair should be on a separate line. If there are no processes remaining in the queue, output a pair of zeros(0 0), separated by a space, terminated by a newline.

6.33. Merging of two linked lists:

Write a program to merge two linked lists.

Input specification

The input to the program will be two lines, each terminated by EOLN.

Line 1 : sequence of integers for the first list
Line 2 : sequence of integers for the second list

The integers appearing on each of the lines may not be in ascending order. So you must arrange the integers on a line as a linked list in non-decreasing order. Once a line of input is read, the integers on that line must be output as an ordered list of numbers on the same line terminated by an EOLN.

After obtaining two ordered linked lists, you must merge them such that there are no duplicate elements in the resultant list and the new list has elements in a non-decreasing order.

The elements in the new list must be printed on the same line terminated by EOLN.

The elements in the input list will be integers and duplicate integers will not be provided on a line of input.

Output specification

The output of your program must consist of three lines, each terminated by EOLN.

Line 1 : elements of the first list in ascending order.
Line 2 : elements of the second list in ascending order.
Line 3 : elements of the merged list in ascending order.

In case the output list is empty, output an EOLN.

Sample Input

20 10 5 13 30

5 2 10 9

Sample Output

5 10 13 20 30
2 5 9 10
2 5 9 10 13 20 30

6.34. Circular Array:

N numbers, starting from 1, are placed around a circular table. Starting from 1 every Kth number around the table is removed. This process is continued until all the numbers are removed from the table.

For example If N = 10 and K = 3 the order in which numbers removed are

1 4 7 10 5 9 6 3 8 2

Write a program, which prints out the sequence of numbers in the same order as removed for any given N and K.

Input Specification

Two numbers N and K given on a single line terminated by an EOLN.

Sample Input

10 3

Sample Output

1 4 7 10 5 9 6 3 8 2

6.35. Binary Search Tree:

Construct a binary search tree based on a set of keys given as input. The criteria to be met by each node is:

info(left(node)) < info(node) <= info(right(node)).

After the tree is constructed, you have to rotate the tree to right or left by one step and then print out the pre-order traversal of the tree.

The input will consist of a number N (< 50) followed by a sequence of N integers, all in one line. These N numbers will be the keys to construct the tree. After this there will be a

single word (either left or right) on a separate line. This is the instruction as to which way the tree is to be rotated.

The output from the program will be the pre-order traver-sal of the resulting tree. All keys are to be printed in a single line, separated by spaces.

If the desired rotation cannot be performed, ignore the command and do pre-order traversal of the tree con-structed from the input.

Sample Input

6 1 2 4 3 5 6
left

Sample Output

2 1 4 3 5 6

Sample Input

4 4 3 5 6
right

Sample Output

3 4 5 6

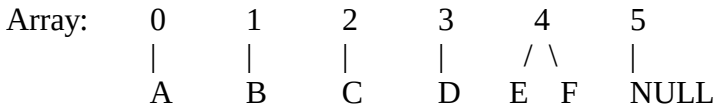
6.36. Binary Tree Problem:

In this problem, you are given the probabilities of occurrence of a sequence of tokens. You are required to con-struct a suitable encoding of the input token to efficiently represent these tokens while storing. In this problem, each token will be represented by only one character. The detailed algorithm is given below.

Construct a binary tree using the probability sequence, as follows.

For example, consider the tokens: A, B, C, D, E and F, and the respective probabilities of occurrence: 40, 30, 10, 10, 6, and 4. We represent probabilities as an integer in the range 0 to 100.

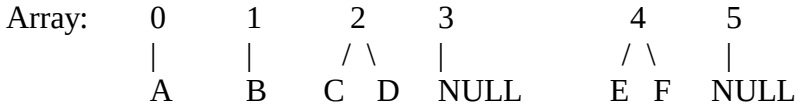
Represent these as single tree-nodes consisting of the token-name and the probability. It is useful to store these nodes in an array as shown in the figure. Scan the array and find the lowest two probabilities. These will be E and F, at positions 4 and 5 (assuming A is at zero). Combine these nodes into a binary tree, by creating a new parent node and marking these nodes as its children. The node with smaller array index, will form the left subtree and the other will be the right subtree.



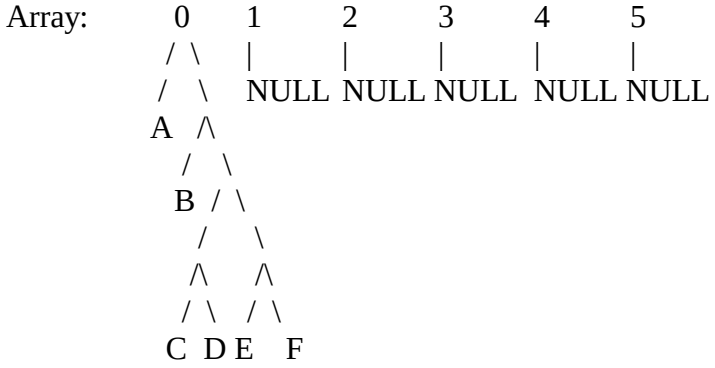
Replace the array entry which has smaller array index with the new parent of this subtree and delete the other entry (mark it with NULL).

Now the array will contain: 40, 30, 10, 10 and 10.

Repeating the above process once more, we get to combine two of the 10 nodes. In case of a tie, we choose the first in the array. Thus we will combine nodes C and D to create another subtree. This resulting tree will replace the node C and the node D will be marked as NULL.



Repeat the process, till the array has only one element remaining. The final tree is as follows:



Input specification

The input consists of one line giving the number of tokens N(< 100), followed by N lines each containing a token name and a probability.

Output specification

Traverse the tree and print out the depth of each leaf node, left-to-right.

Some part of the code for solving this problem will be given to you in the grader. This is also reproduced below for your information in planning the rest of the code.

The code given to you implements the output part of the solution, ie. printing the depths of the leaf nodes from left to right. This code should not be changed, and should be included in your program file as "prob1.h". Please note that the tree-node structure is also defined in this file. You must use this same structure.

Sample Input

```
6
A  40
B  30
C  10
D  10
E  6
F  4
```

Sample Output

```
1 2 4 4 4 4
```

6.37. Multi-way Search tree:

Write a program to construct a multi-way search tree of order 5 using a set of numbers as input. The ordering to be used is descending order. The basic approach to insertion of a key (say, k) is similar to construction of binary search trees.

For each node (from the root onwards) if the node has space, insert the key there at a suitable place, else find a position i such that, key k(i-1) > k and ki < k. Then continue the search with the ith subtree. If the subtree is empty, create node structure there.

The input will consist of a number N (> 0) followed by N numbers. The different numbers will be on separate lines. The program must, for each number read-in, print out the number of nodes (not keys!) in the tree after insertion of the number into the tree. You can assume that there will be no duplicate numbers in the input. All output must be on one line.

Sample Input

```
10
1001
231
134
2356
347
455
678
745
```

1010
444

Sample Output

1 1 1 1 2 2 2 2 3 4

6.38. Line segments problem:

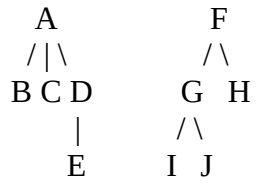
Given a set of horizontal and vertical line segments (coordinates given), find the points of intersection of these lines.

1. Given a tree's root node, make an exact copy of the tree and return the new root.
2. Given a tree's root node, build a mirror image of it and return the new root.
3. Given a forest of trees, construct a single equivalent binary tree from them. E.g.:

Sample Input

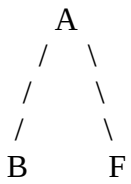
1 A
2 B
2 C
2 D
3 E
1 F
2 G
3 I
3 J
2 H

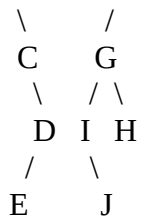
The numbers correspond to the level of the node.



Preorder: A B C D E F G I J H

Output Tree:



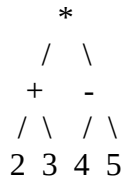


Inorder: B C E D A I J G H F

6.39. Expression Tree:

Construct a tree with the given input and evaluate the arithmetic expression. The leaf nodes of the trees are numbers whereas the other nodes are operators.

e.g.:



This is equivalent to (2+3) * (4-5) = -5 (Output)

6.40. An other Binary Tree Problem:

The input to your program will be a string of letters, which symbolize the nodes of a binary tree [refer Notes]. The order of appearance of the letters is the pre-order traversal [refer Notes] of the tree. A 0(zero) in the string specifies that the relevant child is null. Your program should then accept two letters as input and give as output the path from the first to the second.

The input to the program will be as follows.

<string representing the tree >
<string of exactly two letters representing the two nodes>

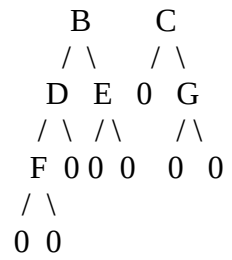
The output should be in the following format.

<String representing the path from first node to second node>

The input to your program will be of the form:

ABDF000E00C0G00
DG

the sample tree for this example string is as below..



the output for this problem should be:
DBACG

Notes:

1. Tree: A tree is a data structure in which there is one root node(e.g. A in the above example) and from each node sprout one or more branches(e.g. AB, AC) leading to one or more nodes(B & C).All the nodes in the tree except the root node can be null(e.g. F has two null children). A null node cannot have children.
2. Binary tree: A tree in which each node has only two children.
3. Pre-order Traversal: Here the tree is traversed such that the left child of a node and its progeny is completely traversed before traversing the children of its right node. e.g. After specifying the children of A i.e. B & C we go on to specify the children of B(the left child) before we specify the children of C(right node).

6.41. Lexicographically sort:

You are given a string of characters. Your job is to lexicographically sort the strings obtained by doing all possible left circular shifts of the original string. A left circular shift by k positions is defined as follows:

Left_Circular_Shift (a0,a1,a2, ..., ak-1, ak, ak+1,... an-1)= ak,ak+1,..., an-1,a0, a1, ... ak-1

If the given string length is n, then the total number of strings to be sorted is n.

Consider the string abbacd. Strings obtained by all possible left circular shifts of the above string are given below:

Index of string (Value of k) String Obtained

0abbacd
1bbacda
2bacdab
3acdabb
4cdabba
5dabbac

The index of the string is the number of left circular shifts that have to be applied on the original string to obtain this string. The sorted form of the above strings is

Index of string (value of k) String

0abbacd
3acdabb
2bacdab
1bbacda
4cdabba
5dabbac

Input Format:

You will be given the length of the string (i.e., number of characters in the string) followed by the characters in the string, each in a different line.

<Number of Characters n>
<character 0>
<character 1>
. . .
. . .

<character n-1>

Input Example

The input corresponding to the above example will be

6
a
b
b
a
c
d

Output specification

You are required to output the indices of the sorted strings, each in a different line. For example, the output corresponding to the above example will be :

0
3
2

1
4
5

Important: You are required to output the indices of the strings only, and not the strings themselves.

6.42. Variation of the Quick Sort:

Implement a variation of the Quick Sort algorithm as given below.

X is an array of N integers to be sorted. The array is partitioned using an element, say X[J], as the pivot such that after partitioning, all elements in the array at positions less than J are less than X[J] and all elements in the array at positions greater than J are greater than X[J]. This is done recursively on each of the two partitions, till all the partitions contain one element, which means the array is sorted.

The partitioning works as follows. The partitioning procedure gives the position J. Let LB and UB be the lower and upper bound of the subarray respectively. Here, the pivot element A is the middle element of the subarray i.e. the element at position (LB+UB)/2. Two pointers, DOWN and UP, are initialized to the upper and lower bounds of the subarray respectively. At any point during execution, each element in a position greater than UP is greater than or equal to A; and each element in a position lesser than DOWN is lesser than or equal to A. The two pointers UP and DOWN are moved towards each other as follows:

Step 1: repeatedly increase the pointer DOWN by one position until X[DOWN] > A.
After DOWN is increased each time, if X[DOWN] < X[DOWN - 1], swap X[DOWN] and X[DOWN - 1].

Step 2: repeatedly decrease the pointer UP by one position until X[UP] <= A.
After UP is decreased each time, if X[UP] > X[UP + 1], swap X[UP] and X[UP + 1].

Step 3: If UP > DOWN, swap X[DOWN] and X[UP].
If X[DOWN] < X[DOWN - 1], swap X[DOWN] and X[DOWN - 1].
If X[UP] > X[UP + 1], swap X[UP] and X[UP + 1].

The above steps are repeated till UP <= DOWN. The position UP is the position J required.

Input specification

The input will be a single line containig an integer N, followed by N integers. Assume N < 50.

Output specification

You have to print out the contents of the array X each time dpartitioning is done. The output should be on a single line containing the array elements separated by a space.

Sample Input

8 25 57 48 37 12 92 86 33

Sample Output

Explanation:

25 12 33 37 48 57 92 86	sort(0,7) up = 3 X[down] = 48 X[up] = 37
12 25 33 37 48 57 92 86	sort(0,2) up = 0 X[down] = 25 X[up] = 12
12 25 33 37 48 57 92 86	sort(1,2) up = 1 X[down] = 33 X[up] = 25
12 25 33 37 48 57 86 92	sort(4,7) up = 5 X[down] = 86 X[up] = 57
12 25 33 37 48 57 86 92	sort(6,7) up = 6 X[down] = 92 X[up] = 86

6.43. Closest pair of points:

You are given a set of points (in 2-dimension) with integer co-ordinates. You have to find out the closest pair of points. (i.e., the distance between these two points should be less than distance between any other two points.)

Input Format:

The number of points, followed by x and y co-ordinates (all integers) of each point is given in different line.

<Number of Points n>
<x0>
<y0>
<x1>
<y1>
<x2>
<y2>
. . .
. . .
<xn-1>
<yn-1>

Sample Input

4
100
100

110
110
200
100
200
300

Output Format:

You should output the indices of the closest points, each in a new line. For example, if you find (xi,yi) and (xj,yj) to be the closest points, you should output i and j in separate lines.

1. You should output the indices only and not the points themselves.
2. Indices of points begin from 0.

Output Example:

The output for the above example could be

1
0

or

0
1

6.44. Railway network system:

A railway network system is spread across many islands. There are N (N <= 50) stations in all, identified by numbers from 1 to N. Railway lines interconnect the stations. On a given line a train can travel in any direction between the stations connected by that line. Every island has at least two stations and at least one line. The network has the following properties:

1. If two stations are on the same island, then it is possible to travel between them by rail. It may be necessary to go through several lines and intermediate stations, for this purpose.
2. If two stations are on two different islands, then there is no railway route to go from one to the other. That is, there are no lines connecting stations on different islands.

Input:

The input to the program will be

1. 2 integers N and M representing the number of stations and the number of lines respectively
2. M lines representing the different railway-lines. Each line will have two numbers representing the two stations connected by that line.
3. An integer Z
4. Z lines each having the numbers of two stations

Output:

The program should output

1. The number of islands in the network, followed by a new-line. (Hint : The number of islands is equal to the number of different trees in the spanning forest of the railway network.)
2. For each of the Z lines in (4) above print YES or NO on separate lines depending on whether the two stations are in one-island or not. Hint: Check if they are on the same spanning tree. There are different ways to do this. One possible approach is sketched below.

For each of the Z lines, take one end point and do a traver-sal from that vertex. If during this traversal, you visit the other vertex, then the two vertices are in the same spanning tree.

Sample input

```
8 6
1 2
3 1
5 7
1 4
7 8
6 8
4
1 5
5 6
3 2
7 8
```

Sample output

```
2
NO
YES
YES
YES
```

6.45. Maze Problem:

Conside a n x n matrix . This represents a maze. The input is a set of pairs of numbers each denoting the obstacles in the maze. Find a path from the starting point(0,0) to the target point(n-1,n-1) without meeting any of the obstacles. Also, find the shortest path.

Input:

(3,2) (6,6) (7,0) (2,8) (5,9) (8,4) (2,4) (0,8) (1,3) (6,3)
(9,3) (1,9) 3,0) (3,7) (4,2) (7,8) (2,2) (4,5) (5,6) (10,5) (6,2)
(6,10) (4,0) (7,5) (7,9) (8,1) (5,7) (4,4) (8,7) (9,2) (10,9) (2,6)

6.46. Traveling Salesperson Problem:

A salesperson must start at a specified city, visit each of the n-1 other cities exactly once, and then return to the initial city. The cost of going from city i to city j is C[i,j]. The objective is to find a route through the cities that minimizes the total cost.

6.47. Game of LIFE:

The game of LIFE takes place on a 2d array of cells, each of which may contain an organism. Let occ(i) be the no. of cells adjacent to cell i that are occupied by an org., is obtained from the previous generation applying the following rules:

1. An org. in cell i survive to the next generation if 2<=occ(i)<=3; otherwise it dies.
2. An org. born in empty cell i if 2<=occ(i)<=3; otherwise it remains empty

Write a program that reads initial configuration of occupied cells and prints a series of gene.. Note that the program must maintain two copies of the configuration, since all changes occur simultaneously.

6.48. Message Decoding

Some message encoding schemes require that an encoded message be sent in two parts. The first part, called the header, contains the characters of the message. The second part contains a pattern that represents the message. You must write a program that can decode messages under such a scheme.

The heart of the encoding scheme for your program is a sequence of "key" strings of 0's and 1's as follows:

0,00,01,10,000,001,010,011,100,101,110,0000,0001,. . .,1011,1110,00000, . . .

The first key in the sequence is of length 1, the next 3 are of length 2, the next 7 of length 3, the next 15 of length 4, etc. If two adjacent keys have the same length, the second can

be obtained from the first by adding 1 (base 2). Notice that there are no keys in the sequence that consist only of 1's.

The keys are mapped to the characters in the header in order. That is, the first key (0) is mapped to the first character in the header, the second key (00) to the second character in the header, the kth key is mapped to the kth character in the header. For example, suppose the header is:

AB#TANCnrtXc

Then 0 is mapped to A, 00 to B, 01 to #, 10 to T, 000 to A, ..., 110 to X, and 0000 to c.

The encoded message contains only 0's and 1's and possibly carriage returns, which are to be ignored. The message is divided into segments. The first 3 digits of a segment give the binary representation of the length of the keys in the segment. For example, if the first 3 digits are 010, then the remainder of the segment consists of keys of length 2 (00, 01, or 10). The end of the segment is a string of 1's which is the same length as the length of the keys in the segment. So a segment of keys of length 2 is terminated by 11. The entire encoded message is terminated by 000 (which would signify a segment in which the keys have length 0). The message is decoded by translating the keys in the segments one-at-a-time into the header characters to which they have been mapped.

Input and Output

The input file contains several data sets. Each data set consists of a header, which is on a single line by itself, and a message, which may extend over several lines. The length of the header is limited only by the fact that key strings have a maximum length of 7 (111 in binary). If there are multiple copies of a character in a header, then several keys will map to that character. The encoded message contains only 0's and 1's, and it is a legitimate encoding according to the described scheme. That is, the message segments begin with the 3-digit length sequence and end with the appropriate sequence of 1's. The keys in any given segment are all of the same length, and they all correspond to characters in the header. The message is terminated by 000. Carriage returns may appear anywhere within the message part. They are *not* to be considered as part of the message.

For each data set, your program must write its decoded message on a separate line. There should not be blank lines between messages. Sample input and corresponding correct output are shown below.

Sample input	**Sample output**
TNM AEIOU 0010101100011 1010001001110110011 11000 $#**\	TAN ME ##*\$

01000001011011000111000010000

6.49. Code Generation

Your employer needs a backend for a translator for a very SIC machine (Simplified Instructional Computer, apologies to Leland Beck). Input to the translator will be arithmetic expressions in postfix form and the output will be assembly language code.

The target machine has a single register and the following instructions, where the operand is either an identifier or a storage location.

L	load the operand into the register
A	add the operand to the contents of the register
S	subtract the operand from the contents of the register
M	multiply the contents of the register by the operand
D	divide the contents of the register by the operand
N	negate the contents of the register
ST	store the contents of the register in the operand location

An arithmetic operation replaces the contents of the register with the expression result. Temporary storage locations are allocated by the assembler for an operand of the form "$n" where n is a single digit.

Input and Output

The input file consists of several legitimate postfix expressions, each on a separate line. Expression operands are single letters and operators are the normal arithmetic operators (+, -, *, /) and unary negation (@). Output must be assembly language code that meets the following requirements:

1. One instruction per line with the instruction mnemonic separated from the operand (if any) by one blank.
2. One blank line must separate the assembly code for successive expressions.
3. The original order of the operands must be preserved in the assembly code.
4. Assembly code must be generated for each operator as soon as it is encountered.
5. As few temporaries as possible should be used (given the above restrictions).
6. For each operator in the expression, the minimum number of instructions must be generated (given the above restrictions).

A sample input file and corresponding correct outputs are on the reverse of this paper.

Sample input	**Sample output**
AB+CD+EF++GH+++	L A A B ST $1

```
                    L C
                    A D
                    ST $2
                    L E
                    A F
                    A $2
                    ST $2
                    L G
                    A H
                    A $2
                    A $1

AB+CD+-             L A
                    A B
                    ST $1
                    L C
                    A D
                    N
                    A $1
```

6.50. Variable Radix Huffman Encoding

Huffman encoding is a method of developing an optimal encoding of the symbols in a *source alphabet* using symbols from a *target alphabet* when the frequencies of each of the symbols in the source alphabet are known. Optimal means the average length of an encoded message will be minimized. In this problem you are to determine an encoding of the first N uppercase letters (the source alphabet, S_1 through S_N, with frequencies f_1 through f_N) into the first R decimal digits (the target alphabet, T_1 through T_R).

Consider determining the encoding when R=2. Encoding proceeds in several passes. In each pass the two source symbols with the lowest frequencies, say S_1 and S_2, are grouped to form a new "combination letter" whose frequency is the sum of f_1 and f_2. If there is a tie for the lowest or second lowest frequency, the letter occurring earlier in the alphabet is selected. After some number of passes only two letters remain to be combined. The letters combined in each pass are assigned one of the symbols from the target alphabet. The letter with the lower frequency is assigned the code 0, and the other letter is assigned the code 1. (If each letter in a combined group has the same frequency, then 0 is assigned to the one earliest in the alphabet. For the purpose of comparisons, the value of a "combination letter" is the value of the earliest letter in the combination.) The final code sequence for a source symbol is formed by concatenating the target alphabet symbols assigned as each combination letter using the source symbol is formed. The target symbols are concatenated in the reverse order that they are assigned so that the first symbol in the final code sequence is the last target symbol assigned to a combination letter. The two illustrations below demonstrate the process for R=2.

Symbol	Frequency
A	5
B	7
C	8
D	15

Pass 1: A and B grouped
Pass 2: {A,B} and C grouped
Pass 3: {A,B,C} and D grouped
Resulting codes: A=110, B=111, C=10, D=0
Avg. length = (3*5+3*7+2*8+1*15)/35=1.91

Symbol	Frequency
A	7
B	7
C	7
D	7

Pass 1: A and B grouped
Pass 2: C and D grouped
Pass 3: {A,B} and {C,D} grouped
Resulting codes: A=00, B=01, C=10, D=11
Avg. length = (2*7+2*7+2*7+2*7)/28=2.00

When ***R*** is larger than 2, ***R*** symbols are grouped in each pass. Since each pass effectively replaces ***R*** letters or combination letters by 1 combination letter, and the last pass must combine ***R*** letters or combination letters, the source alphabet must contain $k*(R-1)+R$ letters, for some integer ***k***. Since ***N*** may not be this large, an appropriate number of fictitious letters with zero frequencies must be included. These fictitious letters are not to be included in the output. In making comparisons, the fictitious letters are later than any of the letters in the alphabet.

Now the basic process of determining the Huffman encoding is the same as for the ***R***=2 case. In each pass, the ***R*** letters with the lowest frequencies are grouped, forming a new combination letter with a frequency equal to the sum of the letters included in the group. The letters that were grouped are assigned the target alphabet symbols 0 through ***R***–1. 0 is assigned to the letter in the combination with the lowest frequency, 1 to the next lowest frequency, and so forth. If several of the letters in the group have the same frequency, the one earliest in the alphabet is assigned the smaller target symbol, and so forth. The illustration below demonstrates the process for ***R***=3.

Symbol	Frequency
A	5
B	7
C	8
D	15

Pass 1: ? (fictitious symbol), A and B are grouped
Pass 2: {?,A,B}, C and D are grouped
Resulting codes: A=11, B=12, C=0, D=2
Avg. length = (2*5+2*7+1*8+1*15)/35=1.34

Input

The input will contain one or more data sets, one per line. Each data set consists of an integer value for ***R*** (between 2 and 10), an integer value for ***N*** (between 2 and 26), and the

integer frequencies f_1 through f_N, each of which is between 1 and 999. The end of data for the entire input is the number 0 for *R*; it is not considered to be a separate data set.

Output

For each data set, display its number (numbering is sequential starting with 1) and the average target symbol length (rounded to two decimal places) on one line. Then display the *N* letters of the source alphabet and the corresponding Huffman codes, one letter and code per line. The examples below illustrate the required output format.

Sample Input

```
2 5 5 10 20 25 40
2 5 4 2 2 1 1
3 7 20 5 8 5 12 6 9
4 6 10 23 18 25 9 12
0
```

Output for the Sample Input

```
Set 1; average length 2.10
   A: 1100
   B: 1101
   C: 111
   D: 10
   E: 0
Set 2; average length 2.20
   A: 11
   B: 00
   C: 01
   D: 100
   E: 101
Set 3; average length 1.69
   A: 1
   B: 00
   C: 20
   D: 01
   E: 22
   F: 02
   G: 21
Set 4; average length 1.32
   A: 32
   B: 1
   C: 0
   D: 2
   E: 31
   F: 33
```

6.51. Simple Uncompress

A simple scheme for creating a compressed version of a text file can be used for files which contain no digit characters. The compression scheme requires making a list of the words in the uncompressed file. When a non-alphabetic character is encountered in the uncompressed file, it is copied directly into the compressed file. When a word is

encountered in the uncompressed file, it is copied directly into the compressed file only if this is the first occurrence of the word. In that case, the word is put at the front of the list. If it is not the first occurrence, the word is not copied to the compressed file. Instead, its position in the list is copied into the compressed file and the word is moved to the front of the list. The numbering of list positions begins at 1.

Write a program that takes a compressed file as input and generates a reproduction of the original uncompressed file as output. You can assume that no word contains more than 50 characters and that the original uncompressed file contains no digit characters.

For the purposes of this problem, a word is defined to be a maximal sequence of upper- and lower-case letters. Words are case-sensitive—the word abc is not the same as the word Abc. For example,

x-ray	contains 2 words:	x and ray
Mary's	contains 2 words:	Mary and s
It's a winner	contains 4 words:	It and s and a and winner

There is no upper limit on the number of different words in the input file. The end of the input file is signified by the number 0 on a line by itself. The terminating 0 merely indicates the end of the input and should not be part of the output produced by your program.

Sample Input

Dear Friend,

  Please, please do it--1 would 4
Mary very, 1 much. And 4 6
8 everything in 5's power to make
14 pay off for you.

  -- Thank 2 18 18--
0

Output for the Sample Input

Dear Friend,

  Please, please do it--it would please
Mary very, very much. And Mary would
do everything in Mary's power to make
it pay off for you.

  -- Thank you very much--

6.52. Defragment

You are taking part in the development of a "New Generation" operating system and the NG file system. In this file system all disk space is divided into N clusters of the equal sizes, numbered by integers from 1 to N. Each file occupies one or more clusters in arbitrary areas of the disk. All clusters that are not occupied by files are considered to be free. A file can be read from the disk in the fastest way, if all its clusters are situated in the successive disk clusters in the natural order.
Rotation of the disk with constant speed implies that various amounts of time are needed for accessing its clusters. Therefore, reading of clusters located near the beginning of the disk performs faster than reading of the ones located near its ending. Thus, all files are numbered beforehand by integers from 1 to K in the order of descending frequency of access. Under the optimal placing of the files on the disk the file number 1 will occupy clusters 1, 2, ..., S_1, the file number 2 will occupy clusters S_1+1, S_1+2, ..., S_1+S_2 and so on (here S_i is the number of clusters which the i-th file occupies).

In order to place the files on the disk in the optimal way cluster-moving operations are executed. One cluster-moving operation includes reading of one occupied cluster from the disk to the memory and writing its contents to some free cluster. After that the first of them is declared free, and the second one is declared occupied.

Your goal is to place the files on the disk in the optimal way by executing the minimal possible number of cluster-moving operations.

Input

The first line of the input file contains two integers N and K separated by a space ($1 \le K < N \le 10000$). Then K lines follow, each of them describes one file. The description of the i-th file starts with the integer S_i that represents the number of clusters in the i-th file ($1 \le S_i < N$). Then S_i integers follow separated by spaces, which indicate the cluster numbers of this file on the disk in the natural order.

All cluster numbers in the input file are different and there is always at least one free cluster on the disk.

Output

Your program should write to the output file any sequence of cluster-moving operations that are needed in order to place the files on the disk in the optimal way. Two integers P_j and Q_j separated by a single space should represent each cluster-moving operation. P_j gives the cluster number that the data should be moved FROM and Q_j gives the cluster number that this data should be moved TO.

The number of cluster-moving operations executed should be as small as possible. If the files on the disk are already placed in the optimal way the output should contain only the string "No optimization needed".

Sample input	**Output**
20 3 4 2 3 11 12 1 7 3 18 5 10	2 1 3 2 11 3 12 4 18 6
Sample input	10 8 5 20 7 5 20 7
30 4 2 1 2 3 3 4 5 2 6 7 8 8 9 10 11 12 13 14 15	No optimization needed

6.53. Spell checker

You, as a member of a development team for a new spell-checking program, are to write a module that will check the correctness of given words using a known dictionary of all correct words in all their forms.

If the word is absent in the dictionary then it can be replaced by correct words (from the dictionary) that can be obtained by one of the following operations:

1. Deleting of one letter from the word;
2. Replacing of one letter in the word with an arbitrary letter;
3. Inserting of one arbitrary letter into the word.

Your task is to write the program that will find all possible replacements from the dictionary for every given word.

Input

The first part of the input file contains all words from the dictionary. Each word occupies its own line. This part is finished by the single character '#' on a separate line. All words are different. There will be at most 10000 words in the dictionary.

The next part of the file contains all words that are to be checked. Each word occupies its own line. This part is also finished by the single character '#' on a separate line. There will be at most 50 words that are to be checked.

All words in the input file (words from the dictionary and words to be checked) consist only of small alphabetic characters and each one contains 15 characters at most.

Output

Write to the output file exactly one line for every checked word in the order of their appearance in the second part of the input file. If the word is correct (i.e. it exists in the

dictionary) write the message: "<checked word> is correct". If the word is not correct then write this word first, then write the character ':' (colon), and after a single space write all its possible replacements, separated by spaces. The replacements should be written in the order of their appearance in the dictionary (in the first part of the input file). If there are no replacements for this word then the line feed should immediately follow the colon.

Sample input	**Output**
i	me is correct
is	aware: award
has	m: i my me
have	contest is correct
be	hav: has have
my	oo: too
more	or:
contest	i is correct
me	fi: i
too	mre: more me
if	
award	
#	
me	
aware	
m	
contest	
hav	
oo	
or	
i	
fi	
mre	
#	

6.54. The PATH

"PATH" is a game played by two players on an N by N board, where N is a positive integer. (If N = 8, the board looks like a chess board.) Two players "WHITE" and "BLACK" compete in the game to build a path of pieces played on the board from the player's "home" edge to the player's "target" edge, opposite the home edge. WHITE uses white pieces and BLACK uses black pieces.

For this problem you will play the "referee" for the game, analyzing boards containing black and white pieces to determine whether either of the players has won the game or if one of the players can win by placing one of their pieces in an unfilled position. WHITE's turn is next.

A representation of a board on paper (and in a computer) is an N x N matrix of characters 'W', 'B', and 'U'; where W represents white pieces, B represents black pieces, and 'U' represents unfilled positions on the board.

When we view a matrix representation of the board on paper, WHITE's home edge is the left edge of the board (the first column), and WHITE's target edge is the right edge (the last column). BLACK's home edge is the top edge of the board (the first row), and BLACK's target edge is the bottom edge (the last row).

Thus WHITE wants to build a path from left to right, and BLACK wants to build a path from top to bottom.

Two locations on the board are "adjacent" if one is immediately to the left, to the right above, or below the other. Thus an interior location on the board is adjacent to four other locations. For N>1, corner locations each have two adjacent locations, and for N>2, other border locations have three adjacent locations.

A path is a sequence of distinct positions on the board, L_0, L_1,…, L_k, such that each pair L_i and L_{i+1} are adjacent for I=0,…,k-1. A winning path for a player is a path L_0, L_1,…, L_k filled with the player's pieces such that L_0 is a position on the player's home edge and L_k is a position on the player's target edge. It is clear that if one player has a winning path then the other player is blocked from having a winning path. Thus if all the squares contain pieces, either there are no winning paths or exactly one of the players has at least one winning path.

Input is a series of game board data sets. Each set begins with a line containing an integer specifying the size N, 0<N<81of the (N x N) game board. This is followed by a blank line and then by N lines, one for each row of the game board, from the top row to the bottom row. Each line begins with character and consists of N adjacent characters from the set {'B', 'W', 'U'}. A blank line separates each non-zero data set from the following data set. The input is terminated by a single line containing a board size of zero.

As referee, for each game board in a series of game boards your program should report one of the five types of answers below:

White has a winning path.
Black has a winning path.
White can win in one move. *(White can place a piece in an unfilled position)*
Black can win in one move. *(White can't win in one move AND Black can)*
There is no winning path.

Sample Input:

7

WBBUUUU
WWBUWWW
UWBBBWB
BWBBWWB
BWWBWBB
UBWWWBU
UWBBBWW

3

WBB
WWU
WBB

0

Sample Output:

White has a winning path.
White can win in one move.

Discussion:

For the 7x7 board, the shortest winning path for WHITE covers 15 locations. For the 3x3 board, WHITE appears to have a path from top to bottom, but remember, WHITE wants a path from left to right.

6.55. The Boggle Game

The language PigEwu has a very simple syntax. Each word in this language has exactly 4 letters, the first cannot be a vowel. Also each word contains one or two vowels (y is consider a vowel in PigEwu). For instance, maar and beer are legitimate words, arts is not a legal word.

In the game boggle, you are given a 4x4 array of letters and asked to find all words contained in it. A word in our case (PigEwu) will thus be a sequence of 4 distinct squares (letters) that form a legal word and such that each square touches (have a corner or edge in common) the next square.

Example:

A S S D
S B E Y
G F O I
H U U K

In this board a (partial) list of legal words include:

BASS SABS FOIK FOYD SYDE HUGS

BOBS is a legal word but it is not on this boggle board (there are no two B's here).

Write a program that reads a pair of Boggle boards and lists all PigEwu words that are common to both boards.

The input file will include a few data sets. Each data set will be a pair of boards as shown in the sample input. All entries will be upper case letters. Two consecutive entries on same board will be separated by one blank. The first row in the first board will be on the same line as the first row of the second board. They will be separated by a few tabs, the same will hold for the remaining 3 rows. Board pairs will be separated by a blank line. The file will be terminated by "#".

For each pair of boggle boards, output a sorted list of all common words, each word on a separate line; or the statement "There are no common words for this pair of boggle boards." Separate the output for each pair of boggle boards with a blank line.

Sample Input:

D F F B	W A S U
T U G I	B R E T
O K J M	Y A P Q
K M B E	L O Y R

Sample Output:

There are no common words for this pair of boggle boards.

Part 7

Assignments and Quizzes

7.1. Assignment 01 – Abstract Data Types (ADTs)

Objectives:

- To comprehend and write ADTs
- Conversion of ADT into concrete data type in C++

Prerequisite Skills (C and C++):

- Control Structures
- Pointers and Dynamic Memory Management
- Classes
- Operator Overloading

Overview:

Abstract Data Type: Data type where the specification of the objects and its operations are separated from their implementation. Specification of data type should be independent of its implementation i.e. it should not contain implementation details.

Write ADTs of

- Bag and Set
- Rational and Complex Number
- String

Implement above ADTs in C++ and write driver programs for each of the above ADT to demonstrate its functionality. Apply the following concepts in your ADTs

- Software and Algorithmic maintenance
- Reusability
- Encapsulation and Abstraction
- Principle of least privilege etc
- Time and Space complexity

Files to be submitted:

Put the following files in a zip archive/folder name it on your roll number and upload at \\pucit-waqar\assignments*YourSection*. We will compile and run your code to make sure it works as required.

The files to be submitted are:

1. Set.doc, Set.h, and Set.cpp
2. Bag.doc, Bag.h, and Bag.cpp

3. Rational.doc, Rational.h, and Rational.cpp
4. Complex.doc, Complex.h, and Complex.cpp
5. String.doc, String.h, and String.cpp

.doc will contain ADTs,
.h will contain class definition
.cpp will contain driver programs

Deadline:

Due date after one week of announcement: (April 21, 2003 to April 28, 2003)

7.2. Assignment 02 – Simple Database Management System (SDBMS)

Objectives:

- To design software that can manage a Simple Database System using Linked List and file handling.
- To Comprehend Fundamental Concepts of DBMS.

Prerequisite Skills:

- String Processing
- Linked List
- File Handling

Overview:

Simple Database Management System (SDBMS) is software that provides the facility to store and retrieve customized user records in a database.

This system will provide a text-based environment for data definition and data manipulation.

Following commands will be supported for above-mentioned activities.

Data Definition Commands:

Create Table [Create tablename {fieldname1 datatype1, fieldname2 datatype2 …}]

Create **Student** {Name String (20), RollNo integer};

```
Creates a new table student with two fields Name and
RollNo.
```

Open Table [open tablename]

Open Student

```
Open already existing table named student
```

Close Table [close tablename]

```
close students
closes currently opened table named student
```

Delete Table [delete or delete tablename]

Delete Student

deletes currently opened table or the tables whose name is provided.

Save Table [save]

Save
Save currently opened table at hard disk.

Data Manipulation Commands:

Insert [insert **OR** insert into tablename values (value1, value2, …);]

Insert all field values for a new record at cursor location. Values should be provided for all fields in appropriate order.

Insert [insert **OR** insert into tablename (fieldname1, fieldname2 …) values (value1, value2, …);]

Insert all provided field values for a new record at cursor location. Values should be provided for all fields in respective order.

Delete [delete **OR** delete from tablename all | where *condition (single filed value pair)*;]

Deletes will delete all records, or only those recodes that are matching with the provided criteria.

Update [update **OR** update into tablename set filedname1= value1, filedname2= value2 … where *condition (single filed value pair)*;]

Updates all fields whose values are provided, matching with the given criteria.

Select [Select **OR** Select * from tablename where *condition (up to two filed value pairs)*;]

Displays all fields of the records that are matching with the given criteria.

Select [Select **OR** Select filedname1, filedname2 from tablename where *condition (up to two filed value pairs)*;]

Displays only mentioned fields of the records that are matching with the given criteria.

Cursor Control Commands:

Goto [goto recordnumber | begin | end]
`Set cursor to provided record number, start or end of table`

Display [display]
`Displays record that is pointed by cursor.`

Step [step NoOfRecordsToBeSkiped]
`Move cursor, no of records from current cursor location ahead or back.`

Supported Data Types:

Integer	(long)
Float	
String	(max size must be specified at definition)
Boolean	(contains TRUE and FALSE)

Extra Credits:

1. Implement INNER and OUTER JOINS for SELECT query of multiple tables.
2. Implement ORDER BY clause for SELECT query.
3. Write Seek cursor control command

   Seek [seek *condition (single filed value pair)*]
   `Move cursor to the first record matching the criteria.`

4. Help for the Simple Data Base Management System
5. Support date and time data types for above application
6. Manage two different modes for open table command

   Design (user can modify structure [field names, field data types, add new field, delete field] of the table)
   Edit (user can view, add, delete and modify the records of table)

7. Modify the above application that supports primary key, composite key and individual field constraints i.e. Not Null, default values etc.
8. Write the following commands to alter the structure of the table using given syntax,

   Modify Table [Modify tablename {PreviousFieldname NewFieldname …}]

   Modify **Student** Name StName;
   `Modifies field name to StName in table student.`

Add Table [Add OR Add tablename {fieldname1 datatype1, fieldname2 datatype2 …}]

Add **Student** {Marks float};
```
Adds field Marks as float in table student.
```

Delete Table [Delete OR Delete tablename {fieldname1, fieldname2…}]

Delete **Student** {Marks};
```
Delete field Marks from table student.
```

9. Implement average, count, sum , max, and min, functions for integer and float data types in above system.

Files to be submitted:

Put the following files in a zip archive/folder name it on your roll number and upload at \\pucit-waqar\assignments*YourSection*. We will compile and run your code to make sure it works as required.

The files to be submitted are:

All header files, .cpp, driver programs and sample data files.

Deadline:

Due date after one week of announcement: (June 04, 2003 to June 11, 2003)

7.3. Quiz 01 – Class Quiz

Punjab University College of Information Technology
B.Sc. Spring 2002-2006 (Morning & Afternoon) 3rd Semester
Data Structures Class Quiz

Time Allowed 60 Min **Max Marks 35**

Name: _____ Roll No: _____

Section: _____ Shift: _____

Q1 Define the following, with at least one example: (2*3)

1. Recursive Definition
2. Tri diagonal Symmetric Sparse Matrix
3. α – β Band Symmetric Sparse Matrix

Q2 Convert the following: (3*2)

1. A*B*C-D+E/F/(G+H) (Infix to Prefix)
2. AB+C*DE—FG+$ (Postfix to Infix)

Q3 Generate recursive function from the following definition, and convert the generated function into equivalent iterative code? (2*3)

$$\text{Multiply}(X, Y) = \begin{cases} X & Y = 1 \\ X + \text{Multiply}(X, Y - 1) & \text{Otherwise} \end{cases}$$

Q4 Write a function and calculate its complexity, which takes in an array Ary of integers its size, and an integer N as arguments and print all pairs of integers in Ary whose sum is equal to N. (2*3)

Q5 Write the following functions for Queue ADT that contains only Stack objects as its data members. (2*3)

1. void insert (int)
2. int remove (void)

Q6 Overload * operator for CPolynomial class in following definition. (5)

```
class CPolynomial {
…
private:
        int degree;
    float *coef;
};
```

7.4. Quiz 02 – Pre–Mid Quiz

Punjab University College of Information Technology
University of The Punjab, Lahore.

B.Sc. Spring 2002-2006 (Morning & Afternoon) 3rd Semester
Data Structures Pre – Mid Quiz

Maximum Marks: 4**0** Time Allowed: 6**0 minutes**

Name: _____ Roll # _____

Section: _____ Morning/Afternoon: _____ Dated: _____

PART-1 (THEORY AND DEFINITIONS 20%)

Question 1. Define and write down differences between followings (2x4)

(1) Program and Algorithm
(2) Generalized and Heterogeneous list
(3) Performance Analysis and Measurement
(4) Forward and Bi-directional Iterator

PART-2 (SHORT ANSWER QUESTIONS 40%)

Question 2. Give short answer to followings (16)

i. Write a transformational formula for a 2 dimensional matrix of I x J stored in column major order in an array. **(2)**

T(I, J) =

ii. Evaluate if the postfix expression 2 * 9 2 / 7 + 3 - is correct. **(2)**

iii. Draw the recursive tree for the following recursive function if x = 75, y = 10 **(4)**

$$GCD(x, y) = \begin{cases} y & y >= x \text{ and } x \% y = 0 \\ GCD(y, x) & x < y \\ GCD(y, x \% y) & \text{otherwise} \end{cases}$$

iv. **Scenario:** **(3)**

Each mature rabbit pair produces one rabbit pair after each month and a pair gets matured after one month. At the start of first month there is only one immature rabbit pair.

Write a recursive function to count total number of Rabbit pairs after N number of months.

long int fnRabbitCount (int NumberOfMonths);

v. Write insert before function for Singly Linear Linked list. **(5)**

template <class Type >
void CSLL< Type >:: m_fnInsertBefore (Type Id, Type Info);

PART-3 (PROGRAMMING EXERCISES 40%)

Question 3. **(8)**

template <class Type>
CSLLL <Type> fnIntersection (const CSLLL <Type> &, const CSLLL <Type> &);

```
// Precondition: The lists are Singly circular linked lists that
// might be empty or non-empty.
// Postcondition: List containing the intersection of atoms of
// both lists will be returned.
```

Question 4. **(8)**

Write following functions for class PStack of integers such that first priority is given to single digit numbers, second priority is given to double digit numbers and last priority is given to other numbers? The behaviour of stack is such that pop should return the number having highest priority first, in case of same priority it should behave like ordinary stack? Prototype of the function is given below

int PStack :: m_fnPop( );

7.5. Quiz 03 – Class Quiz

Punjab University College of Information Technology
University of The Punjab, Lahore.

B.Sc. Spring 2002-2006 (Morning & Afternoon) 3rd Semester
Data Structures Class Quiz

Maximum Marks: 4**0** Time Allowed: 6**0 minutes**

Name: _____ Roll # _____

Section: _____ Morning/Afternoon: _____ Dated: _____

PART-1 (THEORY AND DEFINITIONS 20%)

Question 1. Define and write down differences between followings (2x4)

(1) Depth First and Breadth First Search
(2) Heaps and Binary Search Tree
(3) Kruskal and Prims Algorithms
(4) InDegree and OutDegree

PART-2 (SHORT ANSWER QUESTIONS 40%)

Question 2. Give short answer to followings (16)

i. Write down the formulas for Array based implementation of BST **(2)**

ParentIndex(LeftChildIndex) = ?
RightChildIndex (ParentIndex) = ?

ii. What are the maximum and minimum no of leafs possible in a Binary Tree having n number of levels. **(2+2)**

iii. Define Optimal Binary Search Trees and trace the construction of the AVL Tree that results from inserting the C++ keywords in the given order. Show the tree and the balance factors for each node before and after each rebalancing.

typedef, bool, new, return, struct, while, case, enum, while, this **(2+5)**

iv. What are the total numbers of null values in tri-diagonal sparse matrix of order n? **(3)**

PART-3 (PROGRAMMING EXERCISES 40%)

Question 3. **(5)**

```
template <class Type>
BST <Type> fnCopy (const BST <Type> &);

// Precondition: The BST might be empty or non-empty.
// Postcondition: Create a copy of BST and returns it.
```

Question 4. **(6)**

```
template <class Type>
int  BST<Type>::fnBalanceFactor (Type Key);

// Precondition: The BST might be empty or non-empty.
// Postcondition: Search the Key value form BST and return its
// balance factor.
```

Question 5. **(5)**

```
template <class Type>
int  BST<Type>::fnPredecessor (Type Key);

// Precondition: The BST might be empty or non-empty.
// Postcondition: Search the predecessor of Key value form BST
// and returns it.
```

7.6. Quiz 04 – Pre–Final Quiz

Punjab University College of Information Technology
University of The Punjab, Lahore.

B.Sc. Spring 2002-2006 (Morning & Afternoon) 3rd Semester
Data Structures Pre-Final Quiz

Maximum Marks: 5**0** Time Allowed: 6**0 minutes**

Name: _____ Roll # _____

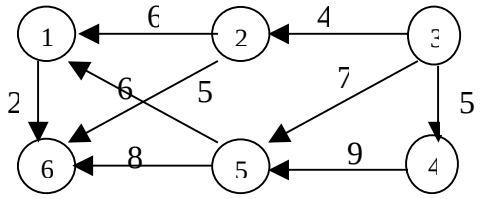
Section: _____ Morning/Afternoon: _____ Dated: _____

Question 1. Define and explain, with one example (briefly): (3*4)

1. Hashing
2. Connected Graph
3. Radix Sort
4. Cyclic & Acyclic Graph

Question 2. Give answers of the following:

1. Differentiate between linked list and tree. **(2)**
2. Write apiece of code to swap two variables without taking any extra variable. **(4)**
3. What is the minimum time complexity we can achieve in sorting algorithm (comparison based)? **(2)**
4. Consider the following graph **(4+3+2+5)**



   a. Draw Adjacency Matrix for the above graph.
   b. Write down in degrees and out degrees of all nodes.
   c. Write down the type of graph.
   d. Find shortest possible distance for every node to every other node.

Question 3. Write code for the following functions. (2*8)

1.

```
template <class T>
void CBinaryTree<T>::m_fnBTMirrorImage (void)
```

```
// Precondition: Binary Tree (*this) may or may not be empty.
// Post condition: The tree is now the mirror image of its
// original value.
    // Example original tree:           Example new tree:
    //          1                                 1
    //         / \                               / \
    //        2   3                             3   2
    //       / \                                   / \
    //      4   5                                 5  4
```

2.

int CLinkedGraph::m_fnLGTotalSum (void)

```
// Precondition: Linked Graph (*this) may or may not be empty.
// Graph Node contains an integer Val as information part
// along with usual node definition.
// Post condition: returns the total sum of all nodes in the
// graph (*this).
```

Part 8

Term Paper and Project

GUIDE NOTES OF TERM PAPER AND TERM PROJECT FOR DATA STRUCTURES

B. Sc. Honours in Computer Scienec

SPTING 2002

By

INSTRUCTOR

Syed Waqar ul Qounain Jaffry

Email: swjaffry@yahoo.com

University College of Information Technology

University of The Punjab Allama Iqbal Campus Lahore

8.1. Aims of the Term paper

Term paper allows you to demonstrate your competence as a computer science professional, and to apply what you have learnt in the different components of your course. Specifically, it allows you to show your ability:

- To carry out a substantial task of your own choosing, in different areas of data structure and computer science.
- To develop and check yours technical writing skills.
- To take decisions, and to justify them convincingly.

Usually the term paper will consist in writing a substantial documented report. During this process you will need to evaluate alternative approaches and representation to the subject under consideration.

Term papers of your seniors are accessible at campus network that will evoke a variety of ideas for you. You should look at these, and discuss any areas that interest you with the relevant senior. Proposed theme of term papers and project for Spring 2002 is implementations of data structures in data base management systems. You should make sure that what you are proposing is imaginative, and substantial enough to justify about 21 hours of work [3 hours per day including everything (background reading, comprehension, and compilation etc.)]. If your proposal is not sufficiently substantial, you risk putting a ceiling on the maximum mark you could achieve right from the outset.

8.2. Other sources of help

You can get help and ideas from other people, such as other teachers, TAs or your student colleagues. Different newsgroups and the WWW can be used to get help. It is important to acknowledge substantial help you get, no matter what the source. Failure to do this could be interpreted as plagiarism, and is a serious offence. ***If you pass off other people's work as your own, you will automatically get a zero mark***, and more severe disciplinary action will be considered.

8.3. The presentation or Viva

After the submission of your term paper class presentation or Viva will be conducted for evaluation purpose. You *must* be prepared to answer questions about it and to be able to convince the evaluators about your work that you are the author and you know it inside out.

8.4. Report

Structure

The appropriate structure of the report varies according to the kind of paper you are writing, and the features you have chosen to emphasize. In consultation with your teacher and TAs, you should decide how much emphasis to give to different issues, to algorithms and data structures, etc.

The following report structure should therefore be seen as a guideline only. Probably no report will stick to it rigidly. It is your responsibility to adapt it to suit your particular term paper.

1. **Title page**: including title, author, degree (BSc, MSc, etc), session, name of supervisor, name of institution (`University College of Information Technology, University of Punjab'), and date.

2. **Preamble:** including (a) Statement of submission, names and signatures of Supervisor, (b) Acknowledgements; (c) Abstract/synopsis (suggested length: half a page); d) Table of Contents; e) List of figures; f) List of Tables;

3. **Introduction:** In this first section, you should describe the motivation for the term paper. Explain whatever background the reader will need in order to understand your contribution. Include a clear and detailed statement of the term paper aims.

   The introduction is the most important section of the document; everyone who reads your term paper will read the introduction, and many will read only that section. You have to make it as inviting and compelling as you can. Think carefully about what you want to say, and in what order you should say it. As well as being the most important section, it's also the most difficult one to write, because it is hard to balance the requirements of giving adequate explanation without entering into too much detail.

   For these reasons, you should write the introduction *last*. It is only when you've written the rest of the term paper that you know what you have to introduce.

   Conventionally, the last part of the introduction outlines the remainder of the term paper, explaining what comes in each section.

4. **Term Paper Body:** The next three or four sections form the main part of the term paper, and cover the relevant topics of your choice. How much space you give to these topics is something you should judge carefully.

5. **Conclusions:** Here you will summarize your achievements and also the deficiencies of your program. You can also say what you would or could have done, if you had had more time or if things had worked out differently. It is important to be completely honest about the deficiencies and inadequacies of your

work, such as they are. Part of your aim is to demonstrate your ability to recognize problems that remain.

6. **References and/or bibliography:** List any books, articles, lecture notes, conference proceedings, manuals or other documents that you refer to in the dissertation and/or that were important in your work. Look in any published book for how to do this. Cite at least the authors, the title, the date and the publication details of each document.

7. **Appendices:** You may include other documents here, such as: glossary; manual index; etc. You may include the source code as an appendix. Don't include voluminous appendices (these should also be submitted electronically if required).

Typographical Format and Binding

Page Format:

Page size:	A4
Top margin:	1.00 inch
Bottom margin:	1.00 inch
Left margin:	1.25 inch
Right margin:	1.00 inch

Page numbering: Bottom right - part of the footnote
Title page not numbered
All other pages before the page of chapter one numbered in lower roman numerals (i, ii, iii, …)
All other pages starting from first page of chapter one to last page of the report numbered in integers (1, 2, 3, …)

Footnote: Each page shall have a footnote giving the title of the project / thesis only.
Left aligned
In case of long titles shorter versions should be used.
There shall be a line over the footnote.

Chapter/Section Startup: Each chapter shall be numbered as Chapter/ Section 1, Chapter/ Section 2, etc. The name of the chapter shall be written immediately below. Both shall be centered horizontally as well as vertically.
The actual chapter/ Section content shall start from the next page.

Text: Only one side of the paper shall be used.
The other side shall be blank.

When a report is opened the right side would contain text, figures, or tables and the left side would be blank.

Tables and Figures: Each table / figure shall be numbered.
For example "Table 1.2: File Allocation Table" or "Figure 3.2: Logical Structure of Boot Drive".

Paragraph:

Single-spaced.
Line entered paragraph.
DONOT put indents at the beginning of the paragraph.
Left aligned or justified.

Text Format

Normal and plane text:

Font Type:	Times New Roman
Font Size:	12

Headings:

Chapter or section Heading:	Times New Roman Bold Size 16 Title Case
Heading 1:	Times New Roman Bold Size 14 Title Case
Heading 2:	Times New Roman Bold Size 12 Title Case
Heading 3:	Times New Roman Bold Size italic 12 Title Case

Sections and Subsections

In case of sections and subsections follow this format:

1 Section
1.1 Sub Section
1.1.1 Nested Sub Section
a

The subsequent reference to a any section shall be made using the section and its number. For example **section 2.1.3** means chapter/section 2 sub-section 1 nested subsection 3.

References

References are to be placed in square brackets and interlaced in the text. For example "A comprehensive detail of how to prevent accidents and losses caused by technology can be found in the literature [1]. A project report / thesis cannot be

accepted without proper references. The references shall be quoted in the following format:

The articles from journals, books, and magazines are written as:

[1] Abe, M., S. Nakamura, K. Shikano, and H. Kuwabara. Voice conversion through vector quantization. *Journal of the Acoustical Society of Japan*, April 1990, E-11 pp 71-76.

[2] Hermansky, H. Perceptual linear predictive (PLP) analysis for speech. *Journal of the Acoustical Society of America*, January 1990, pp 1738-1752.

The books are written as:

[1] Nancy G. Leveson, Safeware System Safety and Computers, A guide to preventing accidents and losses caused by technology, Addison-Wesley Publishing Company, Inc. America, 1995.

[2] Richard R. Brooks, S. S. Iyengar, *Multi-Sensor Fusion Fundamentals and Applications with Software*, The Prentice-Hall Inc. London, 1998.

The Internet links shall be complete URLs to the final article.

[1] http://www.pucit.edu.pk/cs/seminars/ds.html

Length

The length of your term paper depends on the topic you have selected. An excellent dissertation will often be brief but effective (its author will have said a lot in a small amount of space). Voluminous data can be submitted electronically on Floppy. For some topics coding/ demonstration is also required.

8.5. Marks Division

Marks division for different markers is mentioned below,

Markers	***Percentage***
Abstract	20%
Presentation or Viva	40%
Term Paper	40%
Extra credits for cross questioning during class presentation from other students	5% for each new valid question

For details about 45% of viva marks see section 8.

For any query and further information contact.

Syed Waqar ul Qounain Jaffry

TITLE OF THE PROJECT OR TERM PAPER

(Submission Title)

B. Sc. Hounour Spring 2002

Resource Person
Mr. A

Submitted By
MR. X (BCSS02M001)

Date: Monday, September 03, 2003

University College of Information Technology
University of The Punjab Lahore

8.6. Rules about Abstract and Project Proposal

1. A project group can have maximum 2 and minimum 1 number/s of student/s.
2. In term paper each student will work individually.
3. Students from same section can form group together.
4. Viva of the Final Project will be comprised of all courses of B. Sc. Curriculum.
5. Project Proposal and Term Paper abstract (doc. of 5 to 6 pages min.) must be submitted.
6. Proposed Format of Project Proposal and term paper abstract will comprise of followings
    a. Title page: including title, degree (B.Sc. (Honors)), batch, name of supervisor and group members, name of institution (`University College of Information Technology, University of Punjab'), and date.
    b. Objective
    c. Scope
        i. Inclusion
        ii. Exclusion
        iii. Optional
    d. Project Plan (brief timeline for project only)
    e. Hardware and Software specifications (if any for project only)
    f. Application Architecture (if any)
    g. Tools and Technologies used with reasoning (if any for project only)
    h. Features of proposed work (bulleted form for project only)

8.7. List of Term Projects

Project ID	Project Title
DSS2P01	A Fear Game
DSS2P02	Calling Card
DSS2P03	Car Parking System
DSS2P04	Cartoon Simulator
DSS2P05	Checker Board
DSS2P06	Communication Between PC's
DSS2P07	Contents & Index Generator
DSS2P08	Data Compression
DSS2P09	Data Recovery
DSS2P10	Dictionary
DSS2P11	DLD
DSS2P12	Englist Text Editor & Spell Checker
DSS2P13	Excel Spread Sheet
DSS2P14	File System
DSS2P15	Financial System
DSS2P16	Find Fast (Search)
DSS2P17	Happy Travelling
DSS2P18	Kernal (OS)
DSS2P19	Lexical & Syntax Analyzer
DSS2P20	My City
DSS2P21	RAM DISK
DSS2P22	RDBMS
DSS2P23	Simulation of DS
DSS2P24	Sound Recorder
DSS2P25	STL
DSS2P26	Text Editor (Two Files Comparison)
DSS2P27	Turbo Gear
DSS2P28	Word Processor

Part 9

Term Exam and Practicals

9.1. Data Structures Mid Term Examination

Punjab University College of Information Technology
University of The Punjab, Lahore.

B.Sc. Spring 2002-2006 (Morning & Afternoon) 3rd Semester
Data Structures Midterm Examination

Maximum Marks: 5**0** Time Allowed: 9**0 minutes**

Name: _____ Roll # _____

Section: _____ Morning/Afternoon: _____ Dated: _____

INSTRUCTIONS

(2) Answer all questions

(3) No books, notes, paper or other material is allowed in your possession

(4) This paper consist of three types of questions printed on 3 pages (**Please check before commencing**)

(5) PART-1 (theory and definitions 20%)

   a) Remain precise and effective

(6) PART-2 (Short answer questions 40%)

   a) Remain precise and effective

(7) PART-3 (Programming exercises 40%)

   a) Consider followings, while writing your code

   b) Follow requirements, naming conventions, code/algorithmic efficiency, appropriate comments, clarity of the code etc.

(8) You will not be penalised for any wrong answer.

(9) Proposed time division is as follows

   a) 10 minutes (pre and post write reading)

   b) 20 minutes (PART-1)

   c) 30 minutes (PART-2)

   d) 30 minutes (PART-3)

(10) GOOD LUCK

DO NOT TURN THE PAGE OVER UNTIL YOU ARE TOLD TO DO SO

PART-1 (THEORY AND DEFINITIONS 20%)

Question 1. Define and write down differences between followings (5 x 2)

(1) Bag and Set ADT
(2) Recursive and Skip List
(3) Triangular and Tri-diagonal Sparse Matrix
(4) Bubble and Selection Sort
(5) Definiteness and Finiteness of Algorithm

PART-2 (SHORT ANSWER QUESTIONS 40%)

Question 2. Give short answer to followings (20)

a) Consider a matrix of order 3x4 stored in column major order as shown below

a	b	c	d	e	f	g	h	i	j	k	l

what is the position of element **h** in the matrix. **(3)**

b) Draw the memory diagram for following linked list **(3)**

A = (a_1, b_2, B, a_3)
B = (b_1, a_2, B, A)

c) Evaluate if the prefix expression * 2 - 3 + 7 3 is correct. **(2)**

d) Draw the recursive tree for the following recursive function if a = 4 **(2)**

$$\text{Fib}(a) = \begin{cases} 0 & a = 0 \\ 1 & a = 1 \\ \text{Fib}(a-1) + \text{Fib}(a-2) & a > 1 \end{cases}$$

e) **Scenario:** **(5)**

Bacteria Generation Puzzle tells about the total number of bacteria after a specific amount of time. Rules for bacteria generation are as under:

1. If at starting time = 0 there is a single Bacteria, B1.
2. It starts reproducing two Bacteria after each minute when it get 2 minute older.
3. It gets died after 4^{th} minute of its life.

Write a function that takes initial number of bacteria and the time after which the total number of bacteria is required.

long int BGP (long int NoOfBacteria, int Time);

f) Write delete at head function for Circular Singly Linked list. **(5)**

template <class Type>
Type CSCLL < Type >:: m_fnDeleteAtHead ( );

PART-3 (PROGRAMMING EXERCISES 40%)

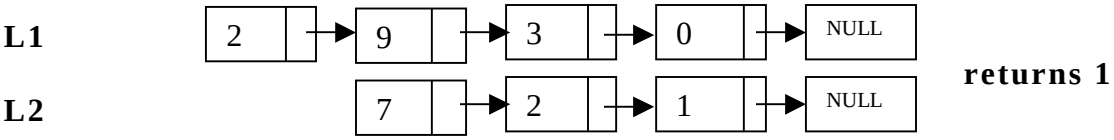
Question 3. **(7)**

template <class Type>
CSLLL <Type> fnUnion (const CSLLL <Type> &, const CSLLL <Type> &);

```
// Precondition: The lists are singly linear linked lists that
// might be empty or non-empty.
// Postcondition: returns a list containing union of atoms of
// both lists.
```

Question 4. **(7)**

A very long integer is stored in a link list where its each digit is placed in separate node. Write a function **Compare** that compares two long integers LI1 and LI2 stored in linked list and returns 0, 1, and –1 if LI1 == LI2, L1 > LI2 and L1 < LI2 respectively.



template <class Type>
int fnCompare(const CSLLL <Type> & LI1, const CSLLL <Type> & LI2);

Question 5. **(6)**

Write a member function of Singly linear linked list class that displays the frequency of each element in it.

template <class Type>
void CSLLL <Type> :: m_fnDisplayFrequency ( );

9.2. Data Structures Mid Term Practical

Punjab University College of Information Technology
University of The Punjab Lahore

B.Sc. Spring 2002-2006 (Morning & Afternoon) 3rd Semester
Data Structures Midterm Practical

Maximum Marks: 5**0** Time Allowed: 15**0 minutes**

Name : _____ Roll # : _____

Section: _____ Morning/Afternoon: _____ Dated: _____

Question 1. Implement class Rational at least with following functions (08)

1. Class definition and Constructor (2)
2. Overload stream extraction operation (p/q), (-p/-q) ->(p/q) (3)
3. Normalize (21/9 -> 7/3) (3)

Question 2. Implement a Doubly linear linked list class at least with following functions (17)

1. Class definition and Constructor (2)
2. Overload stream extraction operation (2)
3. Insert at head (2)
4. Delete at tail (2)
5. Remove Duplicates (4)
6. Implement Iterator for the above class (5)

Question 3. Design and implement a data structure to manipulate vary large integers with at least following properties (25)

1. Class definition and Constructor (5)
2. Overload stream extraction operation (2)
3. Overload stream insertion operation (5)
4. Overload = operator (3)
5. Overload + operator (5)
6. Overload - operator (5)

9.3. Data Structures Final Term Examination

Punjab University College of Information Technology
University of The Punjab Lahore

B.Sc. Spring 2002-2006 (Morning & Afternoon) 3rd Semester
Data Structures Final Term Examination

Maximum Marks: 10**0** Time Allowed: 12**0 minutes**

Name: _____ Roll # _____

Section: _____ Morning/Afternoon: _____ Dated: _____

INSTRUCTIONS

(1) Answer all questions.
(2) No books, notes, paper or other material is allowed in your possession.
*(3) This paper consist of four types of questions printed on 9 pages (**Please check before commencing**)*
(4) PART-I (Multiple Choice Questions 25%)
  a) 1 mark will be granted for each correct answer, 0.20 marks will be deducted for each wrong answer while unattempted questions are ignored.
(5) PART-II (theory and definitions 10%)
  a) Remain precise and effective
  b) You will not be penalised for any wrong answer.
(6) PART-III (Short Questions 40%)
  a) Remain precise and effective
  b) You will not be penalised for any wrong answer.
(7) PART-IV (Programming Questions 25%)
  a) Consider followings, while writing your code
  b) Follow requirements, naming conventions, code/algorithmic efficiency, appropriate comments, clarity of the code etc.
  c) You will not be penalised for any wrong answer.
(8) Proposed time division is as follows
  a) 10 minutes (pre and post write reading)
  b) 35 minutes (PART-I)
  c) 10 minutes (PART-II)
  d) 35 minutes (PART-III)
  e) 30 minutes (PART-IV)
(9) GOOD LUCK

DO NOT TURN THE PAGE OVER UNTIL YOU ARE TOLD TO DO SO

PART-I (Multiple Choice Questions)

1. Consider a singly linked list where F pointer to the first element in the list and L is a pointer to the last element in the list the time of which of the following operations depends on the length of the list

    a. Delete the last element of the list
    b. Delete the first element of the list
    c. Add an element after the last element of the list
    d. Add an element before the first element of the list
    e. Interchange the first two elements of the list

2. p = 1; k = 0;
   while (k < n) {
   p = 2 * p;
   k = k + 1;
   }

   Which of the following is the loop invariant i.e. true at the beginning of each execution of the loop and at the completion of the loop

    a. p = k + 1
    b. $p = (k + 1)^2$
    c. $p = (k + 1)\ 2^k$
    d. $p = 2^k$
    e. $p = 2^{k+1}$

3. If the expression ((2 + 3) * 4 + 5 * (6 + 7) * 8) + 9 is evaluated with + having precedence over * then the value obtained is the same as the value of which of the following prefix expression

    a. * * * + + 2 3 + 4 5 + 6 7 8 9
    b. + + * + 2 3 4 * * 5 + 6 7 8 9
    c. + * + + 2 3 4 * * 5 + 6 7 8 9
    d. + * * * + 2 3 + 4 5 + 6 7 8 9
    e. * + + + 2 3 4 * * 5 + 6 7 8 9

4. Which of the following segments of code deletes the elements pointed by X from a doubly linked list if it is assumed that X points to neither the first nor last element of the list

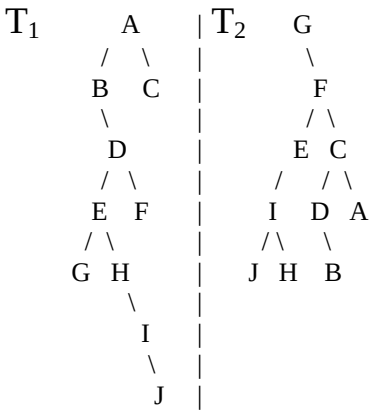
    a. X->prePtr->forPrt = X->forPtr; X->forPtr->prePtr = X->prePtr;
    b. X->prePtr->forPtr = X->prePtr; X->forPtr->prePtr = X->forPtr;
    c. X->prePtr->prePtr = X->forPtr; X->forPtr->forPtr = X->prePtr;
    d. X->prePtr->prePtr = X->prePtr; X->forPtr->forPtr = X->forPtr;
    e. X->prePtr = X->forPtr; X->forPtr = X->prePtr;

5. Shell sort algorithm has been shown to have running time O ($N^{1.5}$) where N is the size of input which of the following is NOT TRUE about shell sort

a. There exists constant C_1 and C_2 such that ∀ N the running time is less then $C_1 N^{1.5} + C_2$ seconds
b. ∀ N, there may be some inputs for which the running time is less than $N^{1.4}$ seconds
c. ∀ N, there may be some inputs for which the running time is less than $N^{1.6}$ seconds
d. ∀ N, there may be some inputs for which the running time is more than $N^{1.4}$ seconds
e. ∀ N, there may be some inputs for which the running time is more than $N^{1.6}$ seconds

6. T_1 and T_2 are the trees indicated below, which of the traversal T_1 and T_2 respectively will produce the same sequence of output

a. preorder, postorder
b. postorder, inorder
c. postorder preorder
d. inorder, preorder
e. postorder, postorder



7. Number of none zero elements in a lower right triangular symmetric sparse matrix of order N are

a. $N^2 – N$
b. $N (N – 1) / 2$
c. $(N^2)/2 – N$
d. $N (N + 1) / 2$
e. None of the above

8. Which of the following is TRUE

a. Bag is a Set
b. Stack is a Bag
c. Stack is a Set
d. Stack has a Set
e. (b) and (c)

9. What is the time complexity of merging two sorted arrays A and B of size m and n respectively, where A is in ascending and B is in descending order?

a. O (m x n)
b. O (m + n)
c. O (m – n)
d. $O (m^n)$

e. $O(n^m)$

10. What is the relation between level x and y of a binary tree T if possible number of nodes at level y are one fourth of possible number of nodes at level x

    a. $y = x + 2$
    b. $x = y/2$
    c. $y >= x$
    d. $x = y - 1$
    e. $y = x - 2$

11. If we let L, V, and R stand for moving left, visiting the node, and moving right then RLV denotes

    a. Pre order traversal
    b. In order traversal
    c. Post order traversal
    d. Level order traversal
    e. none of above

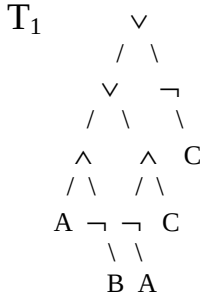
12. Algorithms A1 does a job in time of N^3, where Algorithms A2 does a job in time of 3N + 1000 when does A1 will be efficient then A2

    a. Never
    b. Always
    c. Negative value of N
    d. N + 10 > 20
    e. (c) and (d)

13. Tree T_1 represents a propositional expression for what values of A, B and C it results FALSE. In Propositional calculus operator $\wedge$, $\vee$, $\neg$ are used as and, or and not respectively.

    a. 0, 0, 0
    b. 1, 0, 1
    c. 1, 1, 1
    d. 0, 1, 0
    e. 1, 0, 0

T_1



14. Which of the following sorting techniques has best worst-case time complexity

    a. Insertion sort
    b. Selection sort
    c. Merge sort
    d. Quick sort
    e. Bubble sort

15. Worst-case time complexity of binary search is

    a. n lg (n)
    b. lg (n)
    c. n
    d. 2 lg (n)
    e. none of above

16. What is the address for Identifier 12320324111220 when it is inserted in a hash-table using shift folding technique that use three digit partition

    a. 699
    b. 798
    c. 897
    d. 12212
    e. 65742

17. What will be the Loading factor of hash table having 20 identifiers, table has 20 buckets of size 5.

    a. 1/5
    b. 4/5
    c. 4
    d. 5
    e. 8

18. Which sorting algorithm will take shortest time to sort an already sorted array.

    a. Bubble
    b. Selection
    c. Insertion
    d. Quick
    e. Merge

19. Reordering an array of pointers to list elements, rather than sorting the elements themselves, is a good idea if

    a. The number of elements is very large
    b. Individual elements are large in size
    c. Sort is recursive in nature
    d. There are multiple keys on which to sort the elements
    e. All of above

20. Consider this function declaration. How many asterisks are printed by the function call quiz(5)?

```
void quiz(int i)        {
  if (i > 1) {
     quiz(i / 2);
 quiz(i / 2);
  }
```

```
    cout << "*";
}
```

a. 3
b. 4
c. 7
d. 8
e. Some other number

21. What is the logical relation between graph and tree

    a. Every graph is a tree.
    b. Every tree is a graph.
    c. Tree may be a graph
    d. (a) and (b)
    e. none of above

22. Consider this binary search tree:

```
      14
     /  \
    2    16
   / \
  1   5
     /
    4
```

Suppose we remove the root, replacing it with predecessor. What will be the new root?

    a. 1
    b. 2
    c. 4
    d. 5
    e. 16

23. Consider the usual algorithm for determining whether a sequence of parentheses is balanced. What is the maximum number of parentheses that will appear on the stack AT ANY ONE TIME when the algorithm analyzes: (()(())(()())) ?

    a. 1
    b. 2
    c. 3
    d. 4
    e. none of the above

24. An algorithm for searching a large sorted array for a specific entry x compares every fourth item in the array to x until it finds one that is larger than or equal to x. Whenever a larger item is found, the algorithm examines the preceding three entries. If the array is sorted smallest to largest, which of the following describes all cases when this algorithm might use fewer comparisons to find x than would a binary tree search?

    a. It will never use fewer comparisons.
    b. When x is very close to the beginning of the array.

c. When x is in the middle position in the array.
d. When x is very close to the end of the array.
e. It will always use fewer comparisons.

25. Which of the following is a necessary and sufficient condition for the function WhatIsIt to return a value if it is assumed that the values of n and x are small in magnitude?

```
int WhatIsIt(int x, int n) {
        if (n==1)          return x;
        else               return x * WhatIsIt(x,n-1);
}
```

a. n > 0
b. n = 0
c. n > 0 and x > 0
d. x <= n and n > 0
e. n <= x and n > 0

PART-II (Definitions)

Question 1. Define and explain the followings, with one example (2*5)

a. Bag
b. Mean Sort and B-Sort
c. Recursive List
d. Binary In-Threaded Tree
e. Simple Graph

PART-III (Short Questions)

Question 2. Give the answers of followings

1. The following array is to be sorted ascending using **(6)**

   i. Quick Sort
   ii. Radix Sort

   102, 317, 423, 912, 91,3, 509, 71, 45

   Show your resultant array for each of the above algorithms after one iteration.

2. What is the Maximum and Minimum Number of edges in undirected complete graph having N vertexes? **(3)**

3. An Internal Hash table has 9 buckets numbered 0, 1, . . . ,9 keys are integers and the hash functions

   H (key) = key mod 9

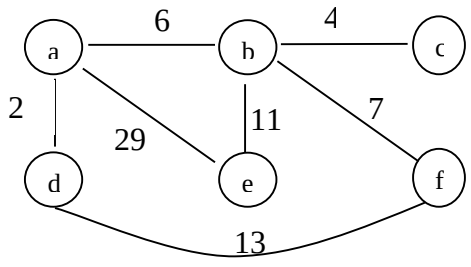
   Show the contents of the hash table and total number of collisions for following insertion sequence

   9, 32, 41, 72, 10, 77, 12

   When quadratic probing is used for collision resolution. **(5+3)**

4. Consider the following graph **(3*2+4)**

   iii. Write down the output of the graph when it is traversed using DFS and BFS considering vertex **a** as starting vertex.
   iv. Draw minimum Cost Spanning Tree using Kruskal's algorithm

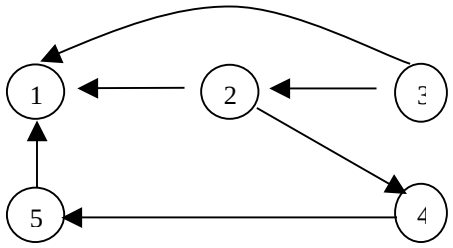


5. Define Optimal Binary Search Trees and trace the construction of the AVL Tree that results from inserting the following words in the given order. Show the tree and the balance factors for each node before and after each rebalancing. **(2+5)**

   December, January, April, March, July, August, October, February

6. Calculate **O** (Time Complexity) of Tower of Hanoi Problem for **n** numbers of disks. **(3)**

7. Draw the Memory diagram for multi-list representation of following graph **(3)**



PART-IV (Programming Questions)

Question 3. Write code for the following functions. (7+9+9)

```
1.
template <class T>
CArray <T> CArray <T>::m_fnRemoveDuplicate (void);
// Precondition: (*this) is a linear array having E number of
// elements with duplicates.(Size>=E>=0)
// Post condition: (*this) will contain distinct elements.

2.
template <class T>
void CBST <T>::m_fnLevelOrder (void);
// Precondition: Binary Search Tree may or may not be empty.
// Post condition: display contents of tree in level order.

3.
template <class T>
Boolean CLinkedGraph <T>::m_fnDeleteArc (T Source, T
Destination)
// Precondition: Linked Graph (*this) may or may not be empty.
// Post condition: delete the Arc from Source to Destination
// Node, return TRUE in success FALSE otherwise.
```

Punjab University College of Information Technology
University of The Punjab, Lahore.

B.Sc. Spring 2002-2006 (Morning & Afternoon) 3rd Semester
Data Structures Final Term Examination
Multiple Choice Question Answer Sheet

Maximum Marks: 25 Time Allowed: 35 **minutes**

Name : _____ Roll # : _____

Section: _____ Morning/Afternoon: _____ Dated: _____

INSTRUCTIONS

(1) Select the most suitable answers.
(2) 1 mark will be granted for each correct answer, 0.20 marks will be deducted for each wrong answer while unattempted questions are ignored.
(3) To mark an answer, completely fill in the box on the answer as follows. Use non-removable ink only. Overwriting is disallowed and will be considered wrong.

a	b	c	d	e
q	q	n	q	q

Fig.: How to mark an answer

	a	b	c	d	e
1	q	q	q	q	q
2	q	q	q	q	q
3	q	q	q	q	q
4	q	q	q	q	q
5	q	q	q	q	q
6	q	q	q	q	q
7	q	q	q	q	q
8	q	q	q	q	q
9	q	q	q	q	q
10	q	q	q	q	q
11	q	q	q	q	q
12	q	q	q	q	q
13	q	q	q	q	q
14	q	q	q	q	q
15	q	q	q	q	q
16	q	q	q	q	q
17	q	q	q	q	q
18	q	q	q	q	q
19	q	q	q	q	q
20	q	q	q	q	q
21	q	q	q	q	q
22	q	q	q	q	q
23	q	q	q	q	q
24	q	q	q	q	q
25	q	q	q	q	q

9.4. Data Structures Final Term Practical

Punjab University College of Information Technology
University of The Punjab Lahore

B.Sc. Spring 2002-2006 (Morning & Afternoon) 3rd Semester
Final Term Practical

Maximum Marks: 100+15(extra) Time allowed: 180 minutes

Name: _____ Roll No: _____

Section: _____ Shift: _____ Dated: _____

PAPER PARTICULARS:

Total numbers of Questions	2	
Total numbers of pages	5	
Marks Division:	Question 1:	40 marks
	Question 2:	60 marks + 15 extra marks
Instructor:	Syed Waqar ul Qounain Jaffry	

INSTRUCTIONS:

(a) You are free to use the Borland C++ help and your data structure textbooks.
(b) This practical contains 15 extra credits, which will be given to the student who will attempt extra practical question mentioned below.
(c) Switch your monitor off as soon as you finish or the allowed time limit is reached.
(d) You can raise queries about the problems sheet and working system within first 15 minutes.
(e) Get your solution confirmed before you leave the labs.
(f) Continuously save you data on Hard Disk no relaxation will be given in case of data lose or power failure.
(g) Meaningful name for identifier, comments, better logic, OO approach and indentation will be considered as good programming practice.
(h) No electronic or paper material is allowed except of clause a.

DO NOT TURN THE PAGE OVER UNTIL YOU ARE TOLD TO DO SO

Q. 1 DeStack : (40 marks)

Define class DeStack and write its member functions for performing the execution of follo wing driver program.

```
template <class eType>
CDStack{

    private:

    // …
    // data member goes here
    // …
    // …
    // Utility Functions for empty and full condition of left and right stacks
    // …

    public:

    CDStack (int Size);

    // Interface Functions
    void m_fnSetValueLeft (eType);
    void m_fnSetValueRight (eType);

    eType m_fnGetValueLeft ( );
    eType m_fnGetValueRight ( );

    m_fnMarkDisplayStack (Boolean Direction);
    // Mark the stack to be displayed for ostream operator

    ostream & operator << ( ostream &, CDStack <eType>);
    // Displays the contents of Marked stack from set in m_fnMarkDisplayStack
    // function.

    ~CDStack ();
};
```

Driver Program:

```
int main (void) {

CDStack Object <int> (10);

Object.m_fnSetValueLeft (1);
Object.m_fnSetValueRight (2);
Object.m_fnSetValueLeft (3);
Object.m_fnSetValueRight (4);
Object.m_fnSetValueLeft (5);
Object.m_fnSetValueRight (6);
```

```
cout  << Object.m_fnGetValueLeft ( ) << endl;
cout  << Object.m_fnGetValueRight ( ) << endl;

Object.m_fnSetValueLeft (5);
Object.m_fnSetValueRight (6);

Object.m_fnMarkDisplayStack (FALSE);

cout  << Object ( ) << endl;

return TRUE;
}
```

Sample Run:

```
5
6
5
3
1
```

Marks Division:

CStackQueue class definition, Constructor and Destructor:	10
m_fnSetValueLeft and m_fnSetValueRight:	5
m_fnGetValueLeft and m_fnGetValueRight:	5
m_fnSetDisplayDirection and ostream & operator <<:	10
Functions for empty and full condition of left and right stacks:	10

Q. 2 BST Operations: (60 marks)

Define class CBST and write its member functions for performing the execution of following driver program.

```
template <class eType>
CBST{
        private:

        // …
        // data member goes here
        // …

        public:

        CBST ();
        CBST (CBST <eType &>);

        void m_fnInsert (eType);

        ostream & operator << ( ostream &, BST <eType>);
```

```
        // display in-order traversal
        // 15 Extra credits will be awarded if tree is displayed tree format max of 5 levels

        CBST <eType> operator +  (CBST <eType> TempBST);
        // Return union of *this and TempBST for the insertion sequence of elements
        // when both are traversed In-Order respectively
        // Union: All elements of both trees without duplication

        CBST <eType> operator *  (CBST <eType> TempBST);
        // Return intersection of *this and TempBST for the insertion sequence of
        // elements when both are traversed In-Order respectively.
        // Intersection: Common elements of both trees without duplication

        CBST <eType> operator –  (CBST <eType> TempBST);
        // Return difference of *this and TempBST for the insertion sequence of elements
        // when both are traversed In-Order respectively.
        // Difference: Those elements of  *this that are not present in TempBST

        CBST <eType> operator =  (CBST <eType>);

        ~CBST ();
};
```

Driver Program:

```
int main (void) {

CBST FirstBST<int>, SecondBST<int>, ThirdBST<int>;

FirstBST.m_fnInsert (13);
FirstBST.m_fnInsert (21);
FirstBST.m_fnInsert (15);

SecondBST.m_fnInsert (91);
SecondBST.m_fnInsert (15);
SecondBST.m_fnInsert (45);

ThirdBST = FirstBST + SecondBST;

cout  << ThirdBST << endl;

SecondBST = FirstBST * ThirdBST;

cout  << SecondBST << endl;

FirstBST = ThirdBST – SecondBST;

cout  << SecondBST << endl;

return TRUE;
}
```

Sample Run:

```
13      15      21      45      91
13      15      21
45      91
```

Marks Division:

CBST class definition, constructor, copy constructor, and destructor:	15
void m_fnInsert (eType):	5
ostream & operator << (ostream &, BST <eType>):	5
CBST <eType> operator + (CBST <eType> TempBST):	10
CBST <eType> operator * (CBST <eType> TempBST):	10
CBST <eType> operator – (CBST <eType> TempBST);	10
CBST <eType> operator = (CBST <eType>);	5

Part 10

Subject Resources

FALL 2001

10.1. Quiz 01 – Afternoon A

Name:		Roll No:	
Section:	**A**	Shift:	**Afternoon**
Instructor:	**Syed waqar ul Qounain Jaffry**	Session	**B.Sc. Fall 2001**

Time Allowed 30 Min **Max Marks 15**

Q1. **(1*4)**

a) For the following definition give recursive function?

$$Q(n1, n2) = \begin{cases} 0 & \text{if } a < b \\ Q(a - b, b) + 1 & \text{otherwise} \end{cases}$$

b) Following code is using integer type stack class. What will be the output of following code?

```
void main ( )     {
        stack s(30);
        for ( int i = 65 ; i < 91 ; i++ )
                s . push ( i ) ;
        while (( i  = = s.pop( ) ! = NULL )        // until stack is not empty
                putchar ( i );
}
```

c) I have a database whose size and contents will not change during the run of my program. The items in my database have integer keys. I need fast searches and it is very important that I conserve memory & keep my code simple.

d) What is the advantage of converting an infix expression to postfix expression?

Q2. Queue ADT is implemented using a solely Single Stack as data member. Write insert (), and remove() functions for above ADT.(Temp. Stacks can be used in member functions) (**3)**

Q3. Given below is the stack class? Check for any possible errors and correct them? (Follow the example given in bold characters) **(3)**

```
class stack       {
        private:
                int  size,  top,  *data ;
        public:
                stack ( int n )     {
                        top = n;            size = 0;           data = new int [n] ;
                }
                int pop ( void )  {
                        if ( top = = size )          return 0;          return data [ top ++ ];
                }
                void push ( int n  )        {
                        if ( top = = 0 )   data [ ++ top ] = n ;
                }
}
```

Q 4. We want to use a single array to represent both a stack and a queue simultaneously. Suppose the array is M[n]. The stack and queue will not fill until the array is filled. If **isFull ( )** function ever detects that there is no space between stack and queue and array is not filled yet, it will shift the queue to create space between stack and queue. Write following functions for the given below class of StackQueue: **(5)**

a) int isFull ( )
b) void insert (int x)
c) int remove ( )
d) void push (int x)

```
class StackQueue        {
        private:
                int * data; int  top ,  front ,   rear;
        public:
                StackQueue ( int n = 10 ) {
                        data =  new int [n];
                        top = 0;
                        front = rear = n –1 ;
                }
}
```

10.2. Quiz 01 – Morning A

Time Allowed 30 Min **Max Marks 15**

Name:		Roll No:	
Section:	**A**	Shift:	**Morning**
Instructor:	**Syed waqar ul Qounain Jaffry**	Session	**B.Sc. Fall 2001**

Q1 For a given integer n>1, the smallest integer d>1 that divides n is a prime factor. We can find the prime factorization of n if we find d and replace n by the quotient of n divided by d, repeating this until n becomes 1. Write a procedure that determines the prime factorization of n in this manner but that displays the prime factors in descending order. **(3)**

Q2 Write the isEmpty and isFull function for the MultiStack ADT.

```
class MultiStack          {
        private:
                int *Base_Index;
                int *Stack_Pointer;
                int *Arr;
        public:
                MultiStack (int No_of_Stacks, int Size_of_Array);
};
```
(3)

Q3 Convert the following:

A*B*C-D+E/F/(G+H) (Infix to Prefix)
*+AB-CD (Prefix to Postfix)
AB+C*DE—FG+$ (Postfix to Infix) **(2+2+2)**

Q4 A symmetric matrix is this whose transpose gives the original matrix. You have given a matrix M (n, n). Find weather it is symmetric or not. (Solve it without using any other matrix). **(2)**

10.3. Quiz 01 – Afternoon B

Time Allowed 30 Min **Max Marks 15**

Name:		Roll No:	
Section:	**B**	Shift:	**Afternoon**
Instructor:	**Syed waqar ul Qounain Jaffry**	Session	**B.Sc. Fall 2001**

Q1 Write a function that takes in an array *Ary* of integers, and an integer *N* as arguments and print all pairs of integers in *Ary* whose sum is equal to *N*. **(3)**

Q2 Write an interface for an abstract data type *Polynomial*. Make sure it has a fairly complete set of operations. **Polynomial** is an algebraic expression in which the power of the variable is non-negative integer, e.g.,

i. $3x^2 + 5x - 7$.
ii. $4x^4 + 5x^2 + 6x - 4$

You have to build the class that can perform the functions (Given below) on the polynomials.

Polynomial operator +(Polimonial P);
Polynomial operator - (Polynomial P);

The definition of the Polynomial class is given as:

```
class Polynomial {
private :
    int dimension;
    int *coef;
public:
   // write prototypes of all possible functions
};
```
(5)

Q3 Both stacks and queues are a kind of priority queue. What exactly does this statement mean? **(1)**

Q4 For the recursive version of the Fibinacci method

```
int fibonacci(int n) {
  return (n <= 1) ? n : fibonacci(n-2) + fibonacci(n-1);
}
```
how many calls are made to the method to compute fibonacci(7)? **(2)**

Q5 You are provided with class stack and queue. Using both check whether or not given string ***S*** is palindrome? (Palindrome is ***S*** = rev (***S***))

(4)

10.4. Quiz 01 – Moring B

Time Allowed 30 Min **Max Marks 15**

Name:		Roll No:	
Section:	B	Shift:	**Morning**
Instructor:	**Syed waqar ul Qounain Jaffry**	Session	**B.Sc. Fall 2001**

Q1 What is ADT? Give precise definition. Also give comparison between ADT and Data Structure. **(2+3)**

Q2

a. What would be appropriate Data Structure to convert decimal number into binary form?
b. Which Data structure should be used in scheduling algorithm for microprocessor in order execute different processes in an appropriate sequence?
c. What is being bypassed by implementing Queue as circular Queue and what is to be sacrificed to implement it?
d. Convert the following postfix form "AB/C-DEF*/+AC*+" into infix. **(1+1+1+2)**

Q3 You are provided with integer stack class. Write a function using this class, which can print a number vertically in single digits? **(5)**

For example:

245
2
4
5

Q4 Suppose we have list of processes along with their respective priorities. These processes will be executed on the priority bases. i.e. the process having more priority say 3 will be executed first and the process with less priority say 1 will be executed later, when all the processes of higher priorities currently have been executed. But in case if we have more than one processes with the same priority then the process which is traversed first will be executed first among the tied processes. Write appropriate ADT for given problem and give remove function for that ADT. **(5)**

10.5. Quiz 02 – Pre Mid Test Afternoon

Time Allowed 45 Min **Max Marks 30**

Name:		Roll No:	
Section:	**A and B**	Shift:	**Afternoon**
Instructor:	**Syed waqar ul Qounain Jaffry**	Session	**B.Sc. Fall 2001**

Question 1. Define and write down differences between following: **(2+2+2)**

1. Pointer and Reference variables
2. Linear and Recursive List
3. Iteration and Recursion

Question 2. Implement the following functions as a new function for the linked list. (Use the usual node definition with member variables called data and next.) **(5+5+5)**

1.

void CSingleLinkedList::m_fnReverse(void);
// Precondition: linear linked list may or may not be *empty.*
// Post condition: order of the atoms in the list should be reversed.

2.

template <class T>
CSingleLinkedList CSingleLinkedList::Operator += (CSingleLinkedList <T> SLL);
// Precondition: *this and SLL are two linear ordered lists. The lists might be empty or it
// might be non-empty.
// Postcondition: *this will be an ordered list containing the elements of both lists.

3.

void CDoubleLinkedList::m_fnNextSum(void);
// Precondition: (*this) linear double linked list may or may not be *empty.*
// Post condition: Each node contains the sum of itself and its entire left predecessors
// (parent) nodes.

Question 3. Who much memory can be saved using linear array implementation of Diagonal Sparse Matrix, also write its transformational formula. (i.e. F(i , j) = ?) **(2)**

Question 4. Write class definition for CSparseMatrix when it is implemented using Linked List? **(3)**

Question 5. What do you mean by term memory leakage? **(2)**

Question 6. Do you have any advantage using virtual destructor in a linked list class? **(2)**

10.6. Quiz 02 – Pre Mid Test Morning

Time Allowed 30 Min **Max Marks 15**

Name:		Roll No:	
Section:	**A and B**	Shift:	**Morning**
Instructor:	**Syed waqar ul Qounain Jaffry**	Session	**B.Sc. Fall 2001**

Question 1. Define and write down differences between following: **(2+2+2)**

1. Default and Copy Constructor
2. Array and Linked List
3. Time and Space Complexity

Question 2. Implement the following functions as a new function for the linked list. (Use the usual node definition with member variables called data and next.) **(5+5+5)**

1.
void CSingleLinkedList::m_fnSort(void);
// **Precondition:** linear linked list may or may not be *empty.*
// **Post condition:** order of the atoms in the list should be in non decreasing.

2.
template <class T>
CSingleLinkedList <T>
CSingleLinkedList::m_fnReveresOrderIntersection(CSingleLinkedList <T> SLL);
// **Precondition:** *this and SLL are two linear non-decreasing ordered lists The
// lists might be empty or it might be non-empty.
// **Postcondition:** localTempList which will be a reverse (non-increasing) order list, and
// containing the same elements of both will be returned.

3.
template <class T>
T CDoubleLinkedList::m_fnShiftLeft(T Data);
// **Precondition:** (*this) linear double linked list may or may not be *empty.*
// **Post condition:** First node contains Data and previous values in propagated
// next (Each next node contains the data of its left predecessor (parent) node.
// Value of last node is returned

Question 3. Suppose that ptrA and ptrB are node pointers. Write one clear sentence to tell me when the expression (ptrA==ptrB) will be true **(2)**

Question 4. Consider the following polynomial and write class definition for CPolyNode?

$$P(x, y, z) = 2xy^2z^3+3x^2yz^2+4xy^3z+5x^2y^2+6y^3z+7x^3z+9$$ **(3)**

Question 5. When all pointers to a dynamically created object are moved elsewhere without deleting the object, it is known as _____. **(2)**

Question 6. Why should we always add a destructor to a linked list? **(2)**

10.7. Quiz 03 – Pre Final Test

Time Allowed 60 Min **Max Marks 50**

Name:		Roll No:	
Section:	**A, B**	Shift:	**Morning and Afternoon**
Instructor:	**Syed waqar ul Qounain Jaffry**	Session	**B.Sc. Fall 2001**

NOTE: Ignore if any Syntax error.

Question 1. Differentiate between following: (2*4)

1. Forward and Bi-directional Iterators
2. Connected and Complete Graph
3. Binary Search and AVL Trees
4. Heterogamous and Generalized Linked list

Question 2. Write code for following functions for different data structures. (7+7)

1.

```
template <class T>
Boolean fn_BSTAndListCompare(SingleLinkedList <T> & List, BST <T> & Tree);
// Precondition: linear single linked list is sorted in ascending order.
// Post condition: return True if both contains similar elements False otherwise.
```

2.

```
void CBinaryTree::m_fnLevelOrderTraversal (void);
// Precondition: Tree may or may not be empty.
// Postcondition: Display elements of Tree by their level order.
```

Question 2. Give answers of the following:

1. What is the Maximum Number of comparisons in a BST of level L? **(1)**

2. What is the Maximum and Minimum Number of Levels a BST of 100 nodes can have? **(2)**

3. How many different AVL Trees can be formed from three nodes that contains the key values 1, 2,3 **(3)**

4. Draw recursive tree of the following code if fn_Fibonacci is called with parameter 5,

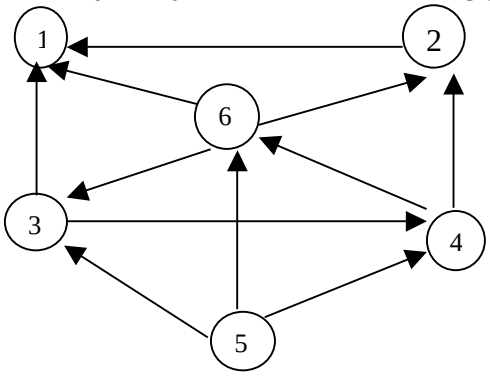
```
int fn_Fibonacci (int iNumber)          {
if (iNumber <= 1)
      return iNumber;
return fn_Fibonacci (iNumber-1) + fn_Fibonacci (iNumber-2);
}
```
(4)

5. Write an equivalent Iterative code for fn_Fibonacci in part 4. **(5)**

6. Draw a Binary Tree of Height 3 for which pre-order and in-order traversals generate the same sequence. **(3)**

7. Use the regular stack operations (push, pop, empty) to construct a new operation ReturnIthElementFromTop that returns the ith element (from the top of stack), leaving stack unchanged. **(5)**

8. Draw adjacency list and adjacency matrix for the following graph **(3+2)**



10.8. Midterm Examination FALL 2001

Punjab University College of Information Technology
University of The Punjab, Lahore.

B.Sc. Fall 2001-2005 (Morning & Afternoon) 3rd Semester
Data Structures Midterm Examination

Maximum Marks: 5**0** Time Allowed: 9**0 minutes**

Name: _____ Roll # _____

Section: _____ Morning/Afternoon: _____ Dated: _____

INSTRUCTIONS

(1) Answer all questions
(2) No books, notes, paper or other material is allowed in your possession
*(3) This paper consist of three types of questions printed on 3 pages (**Please check before commencing**)*
(4) PART-1 (theory and definitions 20%)
   a) Remain precise and effective
(5) PART-2 (Short answer questions 40%)
   a) Remain precise and effective
(6) PART-3 (Programming exercises 40%)
   a) Consider followings, while writing your code
   b) Follow requirements, naming conventions, code/algorithmic efficiency, appropriate comments, clarity of the code etc.
(7) You will not be penalised for any wrong answer.
(8) Proposed time division is as follows
   a) 10 minutes (pre and post write reading)
   b) 20 minutes (PART-1)
   c) 30 minutes (PART-2)
   d) 30 minutes (PART-3)
(9) GOOD LUCK

DO NOT TURN THE PAGE OVER UNTIL YOU ARE TOLD TO DO SO

PART-1 (THEORY AND DEFINITIONS 20%)

Question 1. Define and write down differences between followings (10)

1. ADT and Data type
2. Input and Output Restricted Queues
3. Container and Iterator
4. Big O and Big Ω
5. Sequential and Random access

PART-2 (SHORT ANSWER QUESTIONS 40%)

Question 5. Give short answer to followings (2x10)

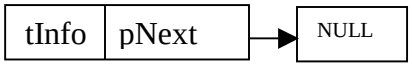
1. Consider the declaration long double Ary[12][15] assume that the array is stored in column major order with indices starting with 0. Assuming that the address of the first element of Ary is 255 what is the address of Ary [5][7].
2. Write a transformational formula for accessing elements (i,j) from a Upper Right Triangular Sparse Matrix of order NxN when it is stored in a linear array using row major order.
3. Stack and Queue are the examples of Priority Queue justify.
4. Which stack operations could result in stack underflow?
5. What is the time complexity of insert and remove operation in a Queue when it is implemented using two stacks?
6. Whether DeQueue can be used as Priority Queue? If it is state the scenario.
7. What is the value of the postfix expression 2 9 7 3 + - *:
8. Consider the usual algorithm for determining whether a sequence of parentheses is balanced. What is the maximum number of parentheses that will appear on the stack AT ANY ONE TIME when the program execution: (( ) (( )) (( )) ( ))?
9. Here is an infix expression: (5+3)*(9*7-2)+1. Suppose that we are using the usual stack algorithm to convert the expression from infix to postfix notation. What is the maximum number of symbols (operators and brackets) that will appear on the stack AT ONE TIME during the conversion of this expression?
10. Convert infix expression into postfix expression

PART-3 (PROGRAMMING EXERCISES 40%)

Question 2. Write code for the followings containers (4+4)

a. **STL Search algorithm for singly linked list.**
b. **STL Iterator prototype (class def.) for doubly liked list.**

Question 3. Implement the following functions for linked list container. (Use the usual node definition with data members called info and next.) (4+4)



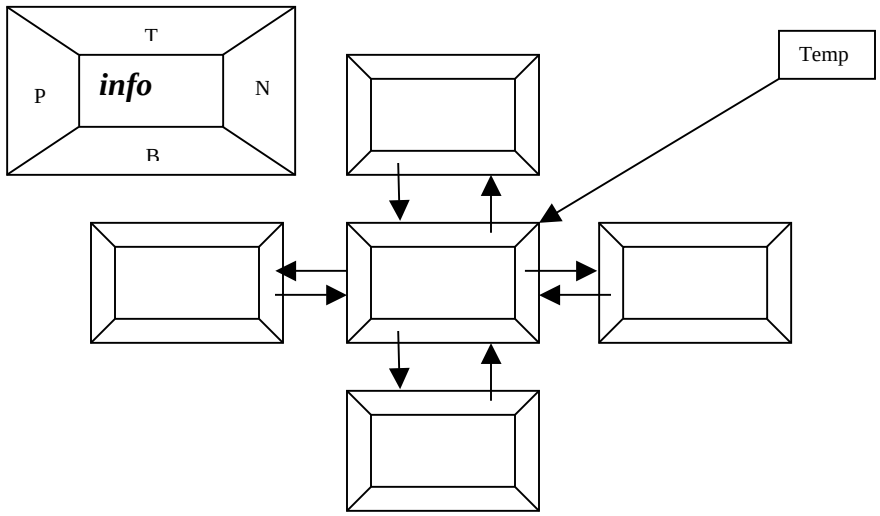
1.

```
template <class eType>
void CSingleLinkedlist <eType> :: fnlistTailInsert (eType
tData);
// Precondition: linked list may be empty or non-empty.
// Postcondition: A new node has been added at the tail
// of the list. The data in the new node is taken from
// the parameter called tData.
```

2.

```
template <class eType>
void CSingleLinkedlist <eType> :: fnSwap(eType tVal1,
eType tVal2);
// Precondition: The lists might be empty or it might be
// non-empty.
// Postcondition: tVal1 and TVal2 are interchanged
```

Question 4. Consider the following Node structure and scenario with four pointers and info part. Write code that establishes doubly links between top-bottom and pre-next nodes of node pointed by Temp, and finally delete the node pointed by Temp. (4)



10.9. Midterm Examination FALL 2001

Punjab University College of Information Technology
University of The Punjab, Lahore.

B.Sc. Fall 2001-2005 (Morning & Afternoon) 3rd Semester
Data Structures Midterm Examination

Maximum Marks: 1**00** Time Allowed: 1**20 minutes**

Name: _____ Roll # _____

Section: _____ Morning/Afternoon: _____ Dated: _____

INSTRUCTIONS

(1) Answer all questions
(2) No books, notes, paper or other material is allowed in your possession
*(3) This paper consist of three types of questions printed on 3 pages (**Please check before commencing**)*
(4) PART-1 (theory and definitions 20%)
   a) Remain precise and effective
(5) PART-2 (Short answer questions 40%)
   a) Remain precise and effective
(6) PART-3 (Programming exercises 40%)
   a) Consider followings, while writing your code
   b) Follow requirements, naming conventions, code/algorithmic efficiency, appropriate comments, clarity of the code etc.
(7) You will not be penalised for any wrong answer.
(8) Proposed time division is as follows
   a) 10 minutes (pre and post write reading)
   b) 20 minutes (PART-1)
   c) 30 minutes (PART-2)
   d) 30 minutes (PART-3)
(9) GOOD LUCK

DO NOT TURN THE PAGE OVER UNTIL YOU ARE TOLD TO DO SO

Question 1. Define and explain with two examples (briefly): (3*5)

1. α—β Band Sparse Matrix
2. Recursive Definition
3. Skewed Binary Tree
4. Double Rotation
5. Connected Graph

Question 2. Give answers of the following:

1. The following array is to be sorted ascending using **(9)**

    1. Bubble Sort
    2. Selection Sort
    3. Insertion Sort

    12, 37, 42, 50, 9, 5, 50, 7, 45, 39, 92

    Show your resultant array for each of the above algorithms after two outer iterations.

2. What are the total numbers of null values in tri-diagonal sparse matrix of order n? **(2)**

3. What is the Maximum and Minimum Number of Levels a BST of 50 nodes can have? **(2)**

4. Suppose an array A contains 25 positive integers. Write a function, which prints all pairs of elements whose sum are 25 and returns the total number of even elements of A. **(5)**

5. An Internal Hash table has 7 buckets numbered 0, 1, 2, 3, 4, 5, 6 keys are integers and the hash functions

    H (key) = key mod 7

    Is used with quadratic probing for collision resolution, show the contents of the hash table and total number of collisions with keys 9, 32, 41, 72, 10, 77 are inserted, in that order, into an initially blank hash table. **(5+3)**

6. Define Optimal Binary Search Trees and trace the construction of the AVL Tree that results from inserting the C++ keywords in the given order. Show the tree and the balance factors for each node before and after each rebalancing.
    bool, new, return, struct, while, case, enum, typedef, while, this **(2+5)**

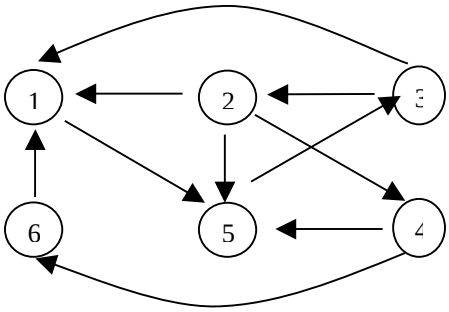
7. What will be the output (return value) of the fn_Function if it is called with parameters 435, 7? Also draw recursive tree.

```
int fn_Function (int iNumber1, int iNumber1)    {
if (iNumber1 = = 0)
return iNumber1;
return fn_Function (iNumber1/ iNumber2, iNumber2) + iNumber1% iNumber2;
}
```

    (2+3)

8. Convert the code of fn_Function in part 7 into an equivalent Iterative code. **(4)**

9. Differentiate between Depths First and Breadth First Search. Write the BFS and DFS traversing output when the following graph is traversed using node 1 as starting node **(3+2*3)**



Question 3. Write code for the following functions. **(4*10)**

1.

```
template <class T>
Bool CBST<T>::m_fnBSTDelete (T DelKey);
// Precondition: Binary Search Tree may or may not be empty.
// Post condition: delete the node containing DelKey. Return
// True id deleted False otherwise
```

2.

```
template <class T>
T CDoubleLinkedList<T>::m_fnDLLShiftLeft(T Data);
// Precondition: (*this) linear double linked list may or may
// not be empty.
// Post condition: First node contains Data and previous
// values in propagated next (Each next node contains the data
// of its left predecessor (parent) node. Value of last node
// is returned.
```

3.

```
template <class T>
void CBinaryTree<T>::m_fnBTMirrorImage (void)
// Precondition: Binary Tree (*this) may or may not be empty.
// Post condition: The tree is now the mirror image of its
// original value.
    // Example original tree:            Example new tree:
    //          1                                1
    //         / \                              / \
    //        2   3                            3   2
    //       / \                                  / \
    //      4   5                                5   4
```

4.

```
int CLinkedGraph::m_fnLGTotalSum (void)
// Precondition: Linked Graph (*this) may or may not be empty.
// Graph node contains an integer Val as information part
// along with usual node definition.
// Post condition: returns the total sum of all nodes in the
// graph (*this).
```

10.10. List of Term Papers FALL 2001

Paper ID	Term Paper Title
DSTP01	3D Animation in C++
DSTP02	Abstract Data types
DSTP03	Algorithmics
DSTP04	Applications of Data Structures in Operating Systems
DSTP05	Applications of DS in Different Environments
DSTP06	Audio File Formats
DSTP07	Binary Trees
DSTP08	C and C++ in LINUX/UNIX Operating System
DSTP09	C/C++ and Data Structures under Linux Environment
DSTP10	C/C++ Programms in LINUX/UNIX Operating System
DSTP11	Class Lecture Notes of Data Structures
DSTP12	Compact in-memory Models for Compressions in Large Txt DB
DSTP13	Containers
DSTP14	Converting Bitmap to JPEG Format
DSTP15	Cryptography
DSTP16	Data Compression
DSTP17	Data Compression Algorithms
DSTP18	Data Encryption
DSTP19	Data Structure Applications in Operating System
DSTP20	Data Structures
DSTP21	Data Structures and Algorithms on Net
DSTP22	Encryption
DSTP23	File Management System
DSTP24	File Organization and Design
DSTP25	File Structure
DSTP26	Fundamentals of an Operating Systems
DSTP27	Game Trees
DSTP28	Graph Template Library
DSTP29	Graph Theory
DSTP30	Graph Theory and its Applications
DSTP31	Hashing
DSTP32	Heap Structures
DSTP33	Huffman Code
DSTP34	Interrupts
DSTP35	Introduction to Algorithms and Its Complexity
DSTP36	Job Scheduling in Operating System
DSTP37	Language Processing
DSTP38	Linked Lists
DSTP39	Low Level Access to Secondry Storage
DSTP40	Memory Management
DSTP41	Memory Management by Operating Systems
DSTP42	Memory Management by Windows NT
DSTP43	Neural Networks
DSTP44	Notes on Data Structures

DSTP45	Operating System Message Queue
DSTP46	Operating Systems
DSTP47	Performance Comperision of Different Sorting and Searching
DSTP48	Process and Memory Management in OS
DSTP49	Process Management
DSTP50	Protected Mode Programming
DSTP51	Queues
DSTP52	Recursion
DSTP53	Recursion and Its Applications in Computer Science
DSTP54	Recursion in C++
DSTP55	Recursive Algorithms
DSTP56	Sandard Template Library
DSTP57	Searching Algorithms
DSTP58	Sorting and Searching
DSTP59	Sound Card Interfacing Using C++
DSTP60	Sound Files
DSTP61	Stacks
DSTP62	Standard Template Library
DSTP63	Storage Management
DSTP64	String Processing
DSTP65	Text to Speech in C/C++
DSTP66	Theory of Automata
DSTP67	Trees
DSTP68	Trees and their Applications
DSTP69	VGA Programming
DSTP70	Video File Formats
DSTP71	Viruses
DSTP72	Wave File Format

10.11. List of Term Projects FALL 2001

Project ID	Project Title
DSFP01	3D Animation in Turbo/Borland C++
DSFP02	A Movie Maker in C++
DSFP03	Character Recognition by ANN
DSFP04	Checker Board
DSFP05	Cryptography Using RSA Public Key System
DSFP06	Data Compression
DSFP07	English to Urdu Dictionary
DSFP08	Draft Game
DSFP09	English Grammar Checker
DSFP10	English Text Editor and Spell Checker
DSFP11	Excel Worksheet
DSFP12	File Management
DSFP13	File Manager
DSFP14	File System
DSFP15	Finding Shortest Route in Map and Finding Nearest Places

DSFP16	Genealogical Tree
DSFP17	Graphical Simulation of Data Structures
DSFP18	Lahore Transport System
DSFP19	LANdotCOM LAN Based Communication Software
DSFP20	Ludo Game
DSFP21	Natural Language Processing
DSFP22	Network Communication between two PC
DSFP23	Operating System Boot Loader
DSFP24	Pakistan Transport System
DSFP25	PDF Converter and Text Extractor
DSFP26	Project Management Tool
DSFP27	Public Key Encryption
DSFP28	Relation Database Management System
DSFP29	Shortest Distance Between Two Points
DSFP30	Simulation of Data Structure
DSFP31	Simulation of Sorting and Searching Algorithms
DSFP32	Small SQL Type Engine
DSFP33	Solitair Game
DSFP34	Standard Template Library
DSFP35	String Validation Using Automata
DSFP36	Text to Speech
DSFP37	Twelve Teny
DSFP38	UML to C++ Converter
DSFP39	C to Assembly Converter

Grossary

Abstract Data Type
a data type where the specification of the objects and operations are separated from their implementation. Specification of data type should be independent of its implementation i.e. it should not contain implementation details

Adaptive Huffman code
Adaptive Huffman code enables dynamically change of the code words accordingly to the change of probabilities of the symbols. In this way, the produced code is more effective then the primary Huffman code.

Adjacency
Two vertices are adjacent if there is an edge between them. Two edges are adjacent if there is a vertex common to both the edges.

Algorithm
A finite set of instructions that if followed accomplishes a specific task is called an algorithm. An algorithm possesses the following characteristics. It is finite and terminates at a given point. A program may be infinite but an algorithm is not. Input of an algorithm is zero or greater than zero. There is always an output given by an algorithm. Instructions should be atomic

Algorithm Efficiency
There are two factors on which an algorithm depends. Space complexity (deals with the RAM). Time complexity (deals with the CPU)

Average-case
It is the step count in which the average number of steps that can be executed for the given parameters are calculated.

AVL Trees
An AVL tree is a binary search tree in which the heights of the left and right subtrees of the root differ by at most 1 and in which the left and right subtrees are again AVL trees.

Balance Factor
Balance factor of a binary tree (bf) is defined as, bf = height of left subtree (hL) – height of the right subtree (hR), where h = the number of nodes visited in traversing a branch which leads to a leaf node at the deepest level of the tree.

Best-Case
It is the step count in which the minimum number of steps that can be executed for the given parameters are calculated.

Binary Search Trees
A binary search tree is a binary tree that is either empty or in which every node contains a key and satisfies the conditions. 1. The key in the left child of a node (if it exists) is less than the key in its parent node. 2. The key in the right child of a node (if it exists) is

greater than the key in its parent node.3. The left and right sub trees of the root are again binary search trees.Every element has a key and no two elements have the same key.

Binary Threaded Trees
Binary threaded trees are an efficient method to utilize the null link field in a binary tree and to store pointers of some nodes in these links fields; these extra pointers are called threads and the tree is specially termed as binary threaded trees. Thus, each node in a binary threaded tree contains links to its child or threads to some other node in the tree.

Binary tree
A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees, the left and right subtrees.

Bipartite graph
If the vertex set of a graph G can be split into two disjoint sets A and B, such that each edge joins a vertex of A with one vertex of B, then the graph is said to be bipartite. It requires that the vertices of A and B be not adjacent. If A has m vertices and B has n vertices then bipartite graph is represented by Km, n.

Breadth-first search
In breadth first search, we first fan out to as many vertices as possible, before penetrating deeper into the tree.

Breadth-First Search (BFS)
Operate on a node, then all its (unvisited) children, then all its (unvisited) grandchildren, etc.

Bubble Sort
Comparisons are only performed between adjacent elements in the array. The elements are swapped if they are in the wrong order. Each pass through the data considers each pair of adjacent elements that are unsorted. After the first pass, the largest element must be in the last element of the array, so the next pass only has to consider one less element.

Child
If the immediate predecessor of a node is the parent of the node then all immediate successor of a node are known as child.

Circular Linked List
A circular list is a list in which the last node of the list (at the tail) points to the head instead of NULL.

Circular Queues
The circular queues are same as linear queues except that they have a circular implementation. This implementation of the queue is the most effective as compared to the two other implementations as there is no refreshing of the queue to keep a check on the status of the queue.

Closed hashing or Open addressing
Closed hashing, also known as open addressing, is an alternative to resolve collision with linked lists. In a closed hashing system, if a collision occurs, alternative cells are tried until an empty cell is found

Collision
When we insert different values in a hash table then it is possible that key determined by the hash function for different elements give the same place or index this situation is called the collision of elements.

Complete Binary Trees
The binary tree T is said to be complete if all its levels, except possibly the last, have the maximum number of possible nodes, and if all the nodes at the last level appear as far left as possible. Thus there is a unique complete tree Tn with exactly n nodes.

Complete graphs
A complete graph is a graph in which every vertex is joined to each vertex of the graph. In other words, if each distinct pair of vertices in a graph is adjacent, then the graph is complete.

Connected (directed) graph
connected (directed) graph G – there exists a path between every pair of distinct vertices in the underlying undirected graph G' implied by G (i.e., <u, v> is an edge in G iff (u, v) (and, thus, (v, u)) is an edge in G'). (This term is not commonly used.)

Connected (undirected) graph
connected (undirected) graph G – there exists a path between every pair of distinct vertices in the G. (Thus, a graph can be connected whether or not it has self-loops.)

Connected graph
A graph is said to be connected if every two vertices are joined by a path. That means that the graph is in one piece.

Connected pair
connected pair (u, v) (<u, v>) – there exists a path from u to v.

Container classes:
classes that store a set of objects.

Cycle
cycle – simple path where the first and last vertices are the same. We often call it a directed cycle if G is directed.

Data abstraction
Data abstraction is separation of a data objects specification from its implementation.

Data encapsulation
Data encapsulation is hiding of implementation details from user.

Data Structures
Data structures refer to structures of data, which are used to organize and store data in such a way that their retrieval is possible with maximum efficiency.

Data Type
A data type is a basic facility in any language, which is used to store data in RAM. It can be defined as, A data type is a collection of objects and a set of operations that act on those objects.

Degree
Maximum number of children that is possible for a node is known as the degree of a node.

Degree of a vertex
The degree of a vertex is defined as the number of edges incident to that particular vertex.

Depth-First Search (DFS)
Operate on a node. Then operate on its first (unvisited) child, then that child's first (unvisited) child, etc., until it is no longer possible, then backtrack to the first ancestor with an unvisited child and repeat.

Directed Graph
If this set E (G) contains ordered pairs, then the graph is said to be a directed graph.

Disconnected graph
A graph is said to be disconnected if there are any two vertices such that there is no way to reach from one to the other via a path in the graph.

Division hash function
The division hash function depends upon the remainder of division. Math.abs(H(k)) % table.length

Double Ended Queues
Double ended queues are those which can have insertions and deletions made at both ends i.e., enqueue and dequeue operations can be performed at both the head and the tail.

Double-Ended Stack
A double ended stack is a variation of the original stack where two stacks are combined in such a way that the insertions and deletions can be made from two sides.

Doubly Linked List

In a doubly linked list, each item contains a pointer to both the next and the previous item in the list.

Dynamic Arrays
Those arrays for which the size can be specified at run time are called dynamic arrays. For example, Array A(size); The size specified in the constructor is taken fron the user and an equivalent memory is allocated. Although one can create a dynamic array, it should be noted here that an array does not grow dynamically.

Edge
An edge is a line joining two nodes (or vertices). It may stand for a link of specific type or just a join between two nodes. An edge is also called an arc.

Eulerian path (Trail)
If a path includes every edge exactly once and ends at the initial vertex, it is called Eulerian path. It is also called a trail.

Extended Huffman
Extended Huffman codes have the characteristic that the coding scheme is coding group of symbols rather than a single symbol.

Folding
Folding is an important method for finding unique keys for hash table.

Folding by boundary
A type of Folding technique for finding unique keys for hash table in which we partition key value X into several parts, reverse the digits in the first and the last partition and then add them all ignoring the carry

Folding by Shifting
A type of Folding technique for finding unique keys for hash table in which we partition key value X into several parts and then adds the parts ignoring the carry.

Forest
A graph that has more than one component and contains no cycles is called a forest.

Full Binary Trees
The tree T is said to be full if all its levels have the maximum possible nodes.

Generalized Lists
A linear linked list is a list comprising only of nodes. The generalized list can be described as, A generalized list, A, is a finite sequence of n elements (where n is greater than or equal to 0), $\alpha 0$, …, $\alpha n-1$, where αi is either an atom or a list. The elements that are not atoms are said to be sub lists of A.

Graph

A graph G is a pair (V, E) where V is a finite, non-empty set of vertices and E is a set of pairs of vertices from V called edges. Note: self-edges are possible; the book calls such graphs graphs with self-edges.

Hamiltonian path
If a path includes every vertex exactly once and ends at the initial vertex, it is called Eulerian path.

Height
Maximum number of nodes that is possible in a path starting from root node to a leaf node is called the height of a tree.

Heterogeneous Lists
Heterogeneous lists are those lists in which data portion of the nodes may be different. One node may contain integer, the other float and so on.

Homogeneous Lists
All nodes in homogeneous list are of same kind of data.

Huffman Code
Huffman Code for an information source S over a code alphabet B is an instantaneous coding with the shortest possible average length

In order Traversal
At each node, First, perform an in-order traversal of the left sub tree, Then visit the node, Then perform an in-order traversal of the right sub tree. Also called an LNR traversal (Left - Node – Right)

Inorder Binary Threaded Tree
In and inorder Binary threaded tree is an binary three If any node N has its right (left) link empty then this field is be initialized by the inorder successor (predecessor) of N."

Inorder Predecessor
Y will be termed as inorder predecessor of N if it comes just before N when the tree T is traversed. Inorder threading can be defined as below

Inorder Successor
Consider a node N in a binary tree T. A node X will be termed as inorder successor of N if it comes just after N when the tree T is traversed in inorder fashion.

Insertion sort
Starting with the second element, iteratively insert each element into its correct location among the already sorted elements to its left. This is accomplished by iteratively swapping the element with every element to its left that is larger than it.

Isomorphism

Two graphs G1 and G2 are said to be isomorphic if there is a one-one correspondence between the vertices of G1 and those of G2 that is the number of edges is equal in both graphs.

Leaf
The node, which does not have the child is called leaf node.

Length
length – number of (not-necessarily unique) edges on path; equivalently, length of vertex sequence – 1.

Level
Level is the rank of the hierarchy and root node is termed as in level 1. If node is at level l then its Childs are in level l+1 and parent is at level l-1.this is true for all nodes except the root node.

Level Order Traversal
In this type of traversal, the root node is visited first and then its sub trees are visited level wise. All the nodes on one level are visited before visiting any nodes on deeper levels.

Linear Data Structures
Data structures that are arranged regularly in a sequence are called linear data structures. Examples are arrays, stacks, queues and lists.

Link
This is a pointer to a node in a tree.

Linked List Iterators
An iterator is an object that is used to traverse all the elements of a container class.

List
A list is one of the most fundamental data structures used to store a collection of data items. A list may be defined as a dynamic n-tuple, L = {l1,l2,…,ln} Where li is the ith element of the list. The use of the word dynamic is meant to emphasize that the elements in the n-tuple may change over time. These elements have a linear order that is based upon their position in the list. The first element in the list, l1, is called the head of the list; while the last element, ln, is referred to as the tail of the list.

Loop
A graph node can be linked to itself. Such a link is called a loop.

Max Heap
A max heap tree (H) is a complete binary tree, which satisfies the following properties, For each node N in H, the value at N is greater than or equal to the value of each of the children of N.Or in other words, N has the value, which is greater than or equal to the value of every successor of N.

Max tree
Max tree – children keys are no larger than parent key. Max heap– complete max tree. For priority queues, keys correspond to priorities.

Merge Sort
Iteratively merge sublists starting with lists of 1 element each. Equivalently, divide list into two, recursively sort the sublists, then merge the sorted results.

Mid-square hash function
The mid-square hash function converts the key to an integer then doubles the key. The function returns the middle digits of the results.

Min Heap
A min heap tree (H) is a complete binary tree, which satisfies the following properties, For each node N in H, the value at N is less than or equal to the value of each of the children of N.Or in other words, N has the value which is less than or equal to the value of every successor of N.

Minimum-cost spanning trees:
A spanning tree of a weighted graph such that the sum of the weights is minimal.

Multiplicative hash function
The multiplicative hash function converts the key to an integer and multiplies it by a constant less than one. The function returns the first few digits of the fractional part of the result.

Muti Stack Queue:
Rrepresention of m stacks and n queues, just use an array of pointers to stack objects and an array of pointers to queue objects. Because stacks and queues are represented as lists, we don't run into the same space issues we faced when representing multiple ones.

Mutual recursion
Type of recursion that involves methods that cyclically calls each other. This is known as cyclical or mutual recursion.

Native Data Types
Data types, which are given by any language to facilitate the user. On the other hand

Node Tree
This is the main part of any tree. The node contains three parts data, pointer to left child and pointer to right child. Simply the node contains the actual data and links to the other node.

Node Graph

A node is the point in a graph, which is joined by lines and is the end point of a line. It can represent some entity. A node is also called a vertex.

Non-Linear Data Structures
Data structures, which are not arranged regularly in an order, are called non-linear data structures. Examples are trees, graphs, heaps, etc.

N-Stacks
N-stack or multiple-stack is a single array having more than one stack in it. The memory in this case can be divided either equally to all stacks or with respect to individual sizes if these are known.

Null graphs
A null graph is a graph whose edge set is empty. Each vertex of a null graph is isolated.

Open hashing or Separate chaining
A strategy to resolve collision commonly known as either open hashing or separate chaining, is to keep a list of all elements that hash to the same value.

Parent
Parent of the node is the immediate predecessor of a node.

Partially Dynamic Arrays
The array for which the size can be specified when an object of the class is created. For example the following is a partially dynamic array. Array A(10); Where A is an object of the class array.

Path
A path is a walk in which no vertex is repeated. In figure 4, PàTàSàR is a path.

Path/Walk Length
The number of edges in a path/walk is called its length

Performance analysis
Performance analysis refers to the process of estimating the complexity of a program. It is often done as part of the design phase of software development, especially as a way to choose best among competing approaches.

Performance Evaluation
Performance evaluation can be divided into two major phases. A priori estimate, and A posteriori testing These are referred to as performance analysis and performance measurement, respectively.

Performance measurement
Performance measurement refers to doing actual testing to measure the programs performance.

Polynomial
Arrays can be used to implement the polynomial class. A polynomial is of the form, nå ai xii=1 Where ‘a’is the co-efficient, ‘x’ is the variable and ‘i’ is the exponent

Post Order Traversal
At each node, First, perform a post-order traversal of the left sub tree, Then perform a post-order traversal of the right sub tree. Then visit the node.Also called an LRN traversal (Left - Right – Node).

Postorder Binary Threaded Tree
In and postorder Binary threaded tree is an binary three If any node N has its right (left) link empty then this field is be initialized by the postorder successor (predecessor) of N.”

Preorder Binary Threaded Tree
In and preorder Binary threaded tree is an binary three If any node N has its right (left) link empty then this field is be initialized by the preorder successor (predecessor) of N.”

Preorder Traversal
In an pre order traversal, the node is visited first, next its left child is visited and then the right child. In other words At each node, Then visit the node, First, perform an in-order traversal of the left sub tree, Then perform an in-order traversal of the right sub tree. Also called an NLR traversal (Node - Left – Right)

Primary Clustering
Primary clustering occurs when many items hash into the same slot and long runs of slots are filled up. This results in increased search times. The key that hashes into the cluster will require several attempts to resolve the collision, and then it will add to the cluster.

Priority queue
A queue where elements are removed, not in order of arrival, but in order of priority as specified in a key data member, where elements are

Program
A set of instruction that are executed on a set of values in a specific sequence to achieve desired task.

Queue
Queues are linear and ordered data structures based on the following principle. The organization of data in queues follows the ‘First In First Out’ (FIFO) order. The values are inserted in the queue at the head and removed at the tail.

Quick Sort
Choose a pivot element, effectively place it in its correct position in the sorted list by sorting all other elements to either side appropriately. o This is accomplished as follows: move a left pointer from the second element towards the right until it reaches an element

with a key larger than the pivot element's key. Similarly, move a right pointer from the end towards the left until it reaches an element with a key as small as the pivot element's key. Swap the two elements pointed to by the pointers. Repeat until the left pointer moves passed the right pointer, then swap the pivot element with the element pointed to by the left pointer. Finally, recursively sort the sublists on either side of the pivot element.

Recursion
Recursion is a process in which a function calls itself directly or indirectly (through another function).

Recursive Definition
a definition in which an object is defined in terms of a simpler case of itself is called recursive definition.

Recursive Lists
If a node in a list points to a preceding node in the list, such type of a list is said to be recursive in nature.

Re-hashing
Re-hashing schemes use a second hashing operation when there is a collision. If there is a further collision, we re-hash until an empty "slot" in the table is found.

Root
This is a specially designated node, which has no parent.

Selection sort
Selection sort works by finding the item that goes at the end of the list (or the beginning), moving that item to the correct location, and then sorting the rest of the items (either iteratively or recursively).

Sibling
The nodes that have the same parent are called siblings.

Simple Graph
simple path – path where all vertices, except, possibly, the first and last, are distinct. We often call it a simple directed path if G is directed.

Single Linked List
Each item has a pointer to the next item in the list, and last item point to NULL is called a singly linked list.

Space Complexity
The amount of memory a program needs to run to completion.

Spanning tree

A spanning tree is a subgraph G' = (V,E') of a connected graph G = (V,E) such that E' Í E and G forms a tree rooted at a designated vertex in V. Intuitively, G' is a minimal connected subgraph of G that includes all the nodes in G.

Sparse matrix
Sparse matrices are matrices with many zero entries. Large, sparse matrices are very common in many applications e.g., the matrix representing a large image taken by a NASA spacecraft.

Stack
A Stack is an ordered (linear) data structure, in which the insertions (adding elements to the stack) and deletions (removing elements from the stack) are made at only one end i.e., the top. This property of stacks corresponds to the Last In First Out strategy.

Static arrays
The array which has a fixed size and its size cannot change at run time is called a static array. For example, the array, int A [10], is a static array of size 10.

STL
STL stands for the Standard Template Library. The fundamental components of this library are defined as, Containers ─ contain a collection of objects of various classes. Algorithms ─ procedures/logic involved in solving a problemIterators ─ help to iterate through the containers

String
A string is a sequence of characters S=s0,…,sn-1 where n is its length.

Strongly connected (directed) graph
strongly connected (directed) graph G – there exists a directed path from u to v and v to u for every pair of distinct vertices u and v.

Sub graph
The sub graph of a graph G is a graph whose every vertex is also vertex of G and whose every edge is also a vertex of G. A sub graph is obtained by deleting edges and vertices from a graph.

The Big-Oh Notation (O)
It's useful to be able to estimate the CPU or memory resources an algorithm requires. This "complexity analysis" attempts to characterize the relationship between the size of the data and resource usage with a simple formula. The "O" stands for "order of". Big-Oh gives the upper bound.

Time Complexity
The amount of computer time a program needs to run to completion.

Trees

Trees are non-linear, two-dimensional data structures consisting of a finite set of nodes in which,There is a specially designated node called the root. The remaining nodes are partitioned into n ≥0 disjoint sets T1,…,Tn, where each of the sets is a tree. T1,…,Tn are called the sub trees of the root.

Walk

A walk is a very general term and there are many types of walks in graph theory. A walk is a 'way of getting from one vertex to another'. It consists of a sequence of edges one follows one after the other.

Weighted graph

Weighted graph (aka a network) – graph where each edge is associated with a weight.

Worst-case

It is the step count in which the maximum number of steps that can be executed for the given parameters are calculated.