Programming project: Web-indexer version

The semester programming project may be done in either of two versions: Web indexer and search engine, or language translater.

The main purpose of this project is for you to empirically test and compare the performance of four search algorithms, operating on three data structures, using real-world data.

You will build a Web indexer, which accomplishes part of what a Web crawler does, and a simplified Web search engine. The indexer will read Web pages that you have downloaded, and will generate an index from the text. A user of the search engine can enter a keyword and see the URLs and names of any sites containing the given keyword. The search engine will use the index to look up the URL and other information

You will use empirical results to determine which algorithms and data structures have the best search time efficiencies given various collection sizes.

Specifications

You will test the following different implementations of your index data structure:

- an unsorted array of objects,
- a sorted array, searched by the binary-search algorithm;
- a binary search tree;
- a hash table.

Your indexer should read all the text files named in your list of Web pages. For each word in each file, look up the word in your index and insert the web file's ID into the list of IDs contained in the word's entry. If the word is not yet in your index, insert an entry. Download enough Web data so that at least 2000 different words are indexed.

Your search engine should prompt the user for a word to look up. If the word is present in the index, display all the URLs and site names where that word is found. If it is not present, display a message to the user stating that the word was not found.

Design details

- Start with a text file that contains the following information for each Web page you are going to include in your index:
  - File name (on your system)
  - URL (on Web)
  - Site name

  Suggestion: Store each web page record on one line of your text file.
- You need a structure containing the above information read from file, plus for each URL an ID (Suggestion: use the number 1 for the first page you index, 2 for the second, etc.)
- Your index should be generated into memory; writing it to disk is an optional feature you may choose to implement
- The index is a collection of objects, whose type could be called *index-entry*, with the following members:
  - word spelling
  - linked list of web page IDs
- When storing a word, convert it to uppercase, and when searching, convert the search key to uppercase; also strip punctuation when storing or searching;
- You may wish to create a list of words that are *not* to be included in the index, such as "the";
- Your *main* should be a menu with options such as to read a file into an array, to sort an array, to read into a BST, to read into a hash table, to search, to display counts (important), and to display a collection (for debugging purposes);
- Write *main* and the source files for the array, BST, and hash table implementations as separately compiled *.java* files; use appropriate header files.

Documentation

Explain the specifications for the program (user view and inputs and outputs) in your own words in one or more well-written paragraphs. What is the purpose? Reader should know what program is *about* after reading the specifications.

Separately, document the *design* of your project. How does program go about accomplishing its purpose?

Document each class or structure type you use by describing its role in the application. Each method should have a comment at the top stating its purpose. Documentation should include diagrams of any relevant class relationships or *module hierarchies*. Use UML *class diagrams* to help document each class.

Testing and discussion of test results

Test a variety of different implementations of the index and algorithms to search it. Show test results, in time or in counts of memory accesses, for searching an index implemented in each of the four ways described in Specifications, above.

Use a table to present test results for indexes of different sizes (100 words, 200, 500, 1000, 2000, etc.). In paragraph form, relate your empirical data to the theoretical considerations presented in topic 3 of this course; i.e., for each algorithm, graph or otherwise relate search time to collection size. Discuss efficiencies of linear search, binary search, and BST search algorithms in light of your test data.

You have the option of showing running times in fractions of a second, instead of memory accesses. The purpose of this exercise is for you to assess how well our theoretical notions correlate with data found in real-world text files.

If two algorithms have the same complexity in big-O notation, explain discrepancies between your test results and the theoretical prediction.

Collaboration

When you submit your project with your name on it, you are claiming that it is your own work.

Collaboration may be at the level of discussion of the problem only. That is, you may share ideas, and show each other diagrams, pseudocode, and test results, but not program code or written comments. Any sharing of ideas or other sharables must be documented.

Grading criteria

	%
Specification of program	10
Documentation of design	10
Documentation and formatting of code	10
Presentation and discussion of test results	20
Correctness of code for:	
Web-page collection and index entry	10
Unsorted array of entries	10
Sorted array of entries	10
BST of entries	10
Hash table of entries	10

Submission

Preliminary: 10/20/08

Preliminary submission will include documentation and code and test results for the unsorted and sorted array implementations of the index.

Final: 12/1/08

Submit your project in a thin (not loose-leaf) binder with table of contents and separate sections for specifications, design, code, and test results. Do *not* submit raw data (text downloaded from web).

One way to organize the web-indexer project

1. Restate the project specs in your own words and make them a comment at the start of your program file.
2. Read steps 3-6 below and summarize them in a design documentation comment.
3. Declare an index-entry structure type or class, with a string member (denoting a word) and a linked-list member denoting the set of IDs for web pages that contain the word.
4. Declare an index structure type or class, with an array of index entries and an integer (number of entries in the index) as members.
5. Create a menu that lets you choose to read a text file into the index, search the index, or sort the index.
6. Declare and define three subprograms that are called from the menu to read a file, do a search for the user, and do a parameterized search.
   a. The read-file method opens the text file and inputs all strings. It calls the parameterized-search method to see if the word is in the index, and inserts the word if it is not found.
   b. The user-driven-search method prompts the user for a word and calls the parameterized-search method to see if the word is in the index.
   c. The parameterized-search method takes a string (a word) as a parameter and returns True or False depending on whether the word is in the index.
7. Test your methods with a text file.
8. Add code to determine the number of steps required for a search and test this code.
9. Gather data on search times for various words in and not in the index and for indexes of various sizes, e.g., 50, 100, 200, 500, 1000, 2000. To set the index at a particular size, limit the number of words the read-file method can add to the index or set the index size to a chosen value.
10. Implement automatic reading of several text files in a row, so that the index stores words from all the files.
11. Add a sort option to your menu. Have it invoke a sorting algorithm.
12. Write a user-binary-search and a parameterize-binary-search method, callable when the index is in a sorted state. Repeat step 9, getting test results for binary search.
13. Each time a word is searched for an found, prepend the ID of its file to that word's linked list of IDs. Each time a new word is inserted, insert its file's ID in the index entry's linked ID list.
14. Add options to your menu to read text files into a BST and to search the BST. Obtain search times for BST searches as in step 9.
15. Do the same for a hash-table implementation.
16. Summarize your test data and compare with predicted search times (big-O) for each algorithm.

Note: There are many correct ways to do this project: write separate programs for each index implementation, use nested menus, all-manual, all-automatic, etc.

Longer step-by-step procedure for Indexer project

Phase 1 (Documentation, test data, string collection)

Start the project by doing the following:

- Write short first drafts of specifications and design documentation for *entire* project in your own words – this will entail something equivalent to user instructions or user requirements, as well as a description of the module structure, including a module hierarchy diagram;
- Write a modular program that reads a text file into an array of *distinct* strings (no two identical, i.e., discarding all duplicates);
- As test data, download a web page with plenty of text in it and save it on your hard disk as a *.txt* file.
- Each time you retrieve a new string, search your array and insert the string into the array only if the search determines that the string is not present;
- Since "this" and "This" are the same word, for example, store your words as all-upper or all-lower case and convert any word read from the text file to that form before searching for it; also, strip any punctuation from the beginning or end of a word;
- Document preconditions of any methods where they are relevant;
- As test results, show a short input segment (50-100 words) and the result array.

Phase 2 (web-page and index-entry collections)

1. The web-page indexing project will have two collections: information about Web pages, and information about words found in them.
   - The collection of Web pages can be just a small array of objects. You will need to write a function that reads a text file of web-page records into your array. To start with, you need a set of such text files and a text file that you create, listing the IDs you assign, the names, the URLs of the web pages, and the file names of downloaded text;
   - Instead of collections of strings, as in Phase 1, your index will be a collection of structures or objects, each of which will have a string (a word found in a downloaded text file) and a linked list of integers (IDs of all those text files in which the word was found);
   - When your program is reading a file downloaded from the Web so as to index the words in the file, your program knows the ID of the file (an integer). Each time your program searches for a word unsuccessfully, insert the word into the index collection, and the word's list of ID numbers should contain just the ID of that file;
   - Each time the search is successful (the word is already in the index), no new index entry should be inserted; rather the ID for the file in which the word was found should be append-ed to the linked list for the given index entry;
   - Be sure that only one reference to a given page, i.e., only one copy of a given ID, is in any of the linked lists in your index;
   - The program should output a list of words together with a list of IDs of the sites where each word was found;
   - Label test results to show the meaning of your output.

2. Implement the index in two different ways: as an unsorted array of entries and as a sorted array. Allow user to choose which implementation to use, and other user options, by making your program *menu driven*. Submit code and one or two pages of test results.

   - For the unsorted-array version, use the linear search for lookup (including lookup upon insertion, to avoid duplicates); for the sorted-array version, use the binary search for lookup;
   - Note that at the end of the phase in which your program reads the web text into collections of entries, the sorted-array version should apply a sort. (Challenge: write two different sorts and compare performance.)
   - Use a counter or a timer to obtain figures (in steps or in fractions of a second) on how long each algorithm takes to perform searches, given various index sizes, and arrange these figures as a table as part of your test results;

Phase 3 (BST)

Add a binary search tree to your project, as a third way to implement your collection of words alongside sorted and unsorted arrays.

- Do this by adding a type definition for a BST of index entries and providing a separate menu option to build the index as a BST;
- Implement and test a user-driven search option, if this does not exist already;
- Get search times for the BST implementation;
- Begin writing your presentation of test data for indexes of different sizes and for different implementations (array, sorted array, BST);
- Explain how imbalanced your BST is at some size; i.e., find average node depth of a certain BST, and in your test results presentation compare this node depth with the predicted average node depth of a balanced BST;
- Test results should include a table that shows average search times for trees of various sizes;
- In your presentation of results, explain how imbalanced the BST is in at least one test case (i.e., find average node depth and compare with $\log_2 n$).

Phase 4 (hash table)

Add a hash-table implementation of the general collection ADT to the indexer project and test the results. Your hash table should be built from the raw data file as were the array and BST implementations. Write a *separate* hash-function method called by both your hash-table builder and your hash-table searcher methods. Suggestion: Use a hash table implemented as an array of pointers to linked lists of index entries. See how many memory accesses are required to search this hash table, given different sizes of indexes.

Option: Explore complexities of different array sizes for the hash table, and given some good and bad hash functions. A bad hash function might be the sum of the ASCII values of the first two letters of a word, modulo the hash-table size. (Why?)

Submit only code and tests related to or used by the hash-table aspect of the project.

Programming project: Language-translation version

The semester programming project may be done in either of two versions: Web indexer and search engine, or language translater.

The main purpose of this project is for you to implement a searchable collection of strings and to empirically test and compare the performance of four search algorithms, operating on three data structures, using real-world data. The three data structures are an array, a binary search tree, and a hash table.

A secondary purpose is for you to learn about data structures and the execution and translation of programs by working with an assembler, a processor simulator, a symbol table, and a parser/code generator.

Groups of up to three students may work on the project together. Each method, test data set, or lengthy comment must be signed by the author. It is assumed that all students who participate have reviewed all code, documentation, and test results.

Writing a compiler is difficult, is time consuming, and requires theoretical study beyond Data Structures. This project explores some beginner steps in compiler writing and introduces some concepts of machine architecture and programming-language syntax and semantics.

General specifications

You will begin with a Java application, called *P10*, that translates from a simple assembler language to the machine code for a simple ten-instruction processor and simulates execution of the machine-code program on the processor.

The assembler includes a symbol table that is an array of strings, searched linearly. You are to implement four new versions of the symbol table, testing the performance of the four search algorithms: linear search, binary search, BST search, hash-table search.

The simulator executes sequences, branches, and loops, but not subroutine calls. You are to extend it to handle subroutine calls using a stack structure.

The assembler language is inconvenient for humans. You are to implement a simple *compiler* that translates some Java statements into the language of the P10 processor. This will involve *lexical analyisis*, *parsing*, and *code generation*. Lexical analysis breaks down a Java program into *tokens*, such as literals, identifiers, keywords, operators, and punctuation. Parsing groups tokens into meaningful units such as expressions and statements, in the form of a *parse tree*. Code generation translates nodes of a parse tree into assembler language.

Documentation

Explain the specifications for the program (user view and inputs and outputs) in your own words in one or more well-written paragraphs. What is the purpose? Reader should know what program is *about* after reading the specifications.

Separately, document the *design* of your project. How does program go about accomplishing its purpose?

Document each class or structure type you use by describing its role in the application. Each method should have a comment at the top stating its purpose. Documentation should include diagrams of any relevant class relationships or *module hierarchies*. Use UML *class diagrams* to help document each class.

Symbol table and discussion of test results

For the symbol table, show test results, in time or in counts of memory accesses, for each of the four structures/algorithms described in Specifications, above.

Use a table to present test results for symbol tables of different sizes (100 words, 200, 500, 1000, 2000, etc.). Obtain test data by reading HTML files from the Web. In paragraph form, relate your empirical data to the theoretical considerations presented in topic 3 of this course; i.e., for each algorithm, graph or otherwise relate search time to collection size. Discuss efficiencies of linear search, binary search, and BST search algorithms in light of your test data.

You have the option of showing running times in fractions of a second, instead of memory accesses. The purpose of this exercise is for you to assess how well our theoretical notions correlate with data found in real-world text files.

If two algorithms have the same complexity in big-O notation, explain discrepancies between your test results and the theoretical prediction.

Lexical analyzer

To begin, write a program that separates a "hello world" Java program into tokens.

Later, allow numeric literals, parentheses, and arithmetic operators.

Advanced objectives could include assignments, variable declarations, arrays, and method definitions.

Parser

To begin, parse a method call, such as *System.out.println("Hello");*.

Later, parse arithmetic expressions such as (3 + 2 - 4), 3 + 2 * 4, and (3 + 2) * 4.

Then handle assignments

See advanced objectives above under "lexical analyzer."

Code generator

System.out.println(0); should translate to

```
        output zero
        stop
zero    data   0
```

*System.out.println(3+2);*should translate

```
        load    3
        add     2
        store   sum
        output  sum
```

Grading criteria

	%
Specification of program	10
Documentation of design	10
Documentation and formatting of code	10
Presentation and discussion of test results	20
Correctness of code for:	
Web-page collection and index entry	10
Unsorted array of entries	5
Sorted array of entries	5
BST of entries	5
Hash table of entries	5
Search engine	5
Implementation of methods	5
Parser	5
Code generator	5

Submission

Preliminary: 10/20/08

Preliminary submission will include documentation and code and test results for the unsorted and sorted array implementations of the index.

Final: 12/1/08

Submit your project in a thin (not loose-leaf) binder with table of contents and separate sections for specifications, design, code, and test results.

Group work 0 (CS II review)

1. (Individual, not group) At the Data Structures BlackBoard site, see Discussion Board forum, “Introduction.” Read the message in thread “Welcome” of and respond to it by leaving your own message.

For #2-5, use C, C++, or Java. Comments at the top of each program should include the program’s name and purpose, your name, the date, and the assignment number. Document the purpose of each method and document the process in some way (see syllabus). Show test results. Indent and otherwise format code clearly.

2. Using an editor, create a text file containing several integers. Write a program that reads the integers into an array from the text file and displays the message, “Dups,” if any two or more integers are duplicates, otherwise “No dups.” Your program should work regardless of what data is in the file or how large the file is. In your test output, show full contents of file.
   Sample I/O:
   ```
   1 2 3 4    No dups
   1 2 3 4 2  Dups
   ```

3. Write a program that prompts the user and inputs a string variable, declares a pointer to *char*, dynamically allocates just enough memory for the input string, assigns the address of the dynamic variable to your pointer to *char,* and stores the user’s input in the dynamically allocated string. Display the dynamically allocated string.

4. Write a program that inputs *one* string (a person’s name in two words), and copies the name into another string variable, last name first, with a comma and space between last and first. Display the second (last, first) string. To do this, define a method that takes two in-parameters and uses an out-parameter to return the name in (last, first) form.
   Sample I/O:
   ```
   George Kerry   Kerry, George
   ```

5. Write a method that converts a binary numeral, stored as a C-style string, to a decimal value. Your method will take a string parameter that will consist of ‘1’ and ‘0’ characters, and return an *int*. Test your method with a driver (your *main*) that prompts for a binary numeral as a string, calls your method, and displays the return value. Have your method display an error message if an invalid parameter is passed to it. For an algorithm, see over. (Save your code after 63.271 for possible use in 63.355.)
   Hint: Note that the string, “10”, consists of a character ‘1’ (ASCII code 49), followed by a ‘0’ (code 48), and a null (code 0). If the string variable *s* contains “10”, then *s*[0] – ‘0’ = 1; *s*[1] – ‘0’ = 0.

   Sample parameter/return value:
   ```
   10101     21
   ```

5.4. Binary-to-decimal conversion

It is also useful to be able to convert binary numerals to decimal equivalents. The next two examples illustrate a procedure for doing this.
Method:

1. Initialize the decimal value (*sum*) to 0.
2. Double the sum. [*Note:* if this seems nonsensical with 0, see step 4.]
3. Beginning with the most significant, or leftmost, digit, and proceeding from left to right, add the current digit of the binary number to the sum.
4. Repeat steps 2 and 3 until all digits of the binary numeral have been processed.

The sum will then contain the decimal equivalent of the original binary numeral.

Doubling the sum in step 2 accomplishes our objective because each *1* in a binary numeral represents a value twice as large as a *1* just to its right.

Example 1: Convert 10_2 to a decimal numeral.

1. $sum = 0$
2. $sum = 2 \times sum = 0$
3. $sum = sum + 1 = 1$
4. (back to step 2:) $sum = 2 \times sum = 2$
5. (to step 3:) $sum = sum + 0 = 2$, and exit, since there are no more binary digits to convert.

Digit	Sum	Sum × 2	Sum × 2 + Digit
1	0	0	1
0	1	2	2
	2		

2 ← The final SUM is the decimal equivalent of 10_2.

Example 2: Convert 1101_2 to a decimal numeral.

Digit	Sum	Sum × 2	Sum × 2 + Digit
1	0	0	1
1	1	2	3
0	3	6	6
1	6	12	13
	13		

13 ← This is the desired decimal equivalent of 1101_2.

Copyright © 1998 by David Keil and Richard Johnson. For use of students in D. Keil's classes.

Group work 1 (Numeric, class, and array data types)

For this and all group work, implement the data structures with your own classes, not using Java collection libraries. Please submit all code as Notepad-readable text files. All code should include commenting, including spec of program and of each method, name of coder, and date, plus a file containing test results.

A. Collections

All groups should solve this problem.

1. Write a short program consisting of *main* and several methods called from *main* as described below. (Later you will expand this skeleton program to be a test laboratory for your implementations of all data structures in the course.) In *main*, write a loop that shows a menu with at least the following options: *Insert, Display, Save, Retrieve, Find, Delete, Quit* (or equivalent names) The user's response should be the selector in a *switch* statement in the loop that calls methods, one per menu option (except *Quit*). The skeleton program needs only stub methods.

2. Using the skeleton of #1, implement a collection of integers using an array. Test it by using each of the menu items. The operations in detail are as follows: *Insert* allows an integer chosen by the user to be added to the collection; *Display* displays all the integers in the collection on the screen; *Save* writes all items to a file, with spaces or carriage returns between the values; *Retrieve* reads the contents of a file into the collection; *Find* searches the collection for a key value input by the user; *Delete* finds value input by the user and deletes it from the collection.

B. String search

Problem: for strings s_1 and s_2, find occurrences of s_1 in s_2. Code and find approximate minimum, maximum, and average numbers of steps for the following variants of this problem.

1. Number of occurrences of s_1 in s_2.
2. Number of consecutive occurrences.
3. Location of nth occurrence (0 if none).
4. First location of match to the *pattern* s_1 in s_2, where '?' in the pattern denotes a single-character wild card.
5. First location of match to the *pattern* s_1 in s_2, where '*' in the pattern denotes a 0, 1, 2, ... n character wild card.
6. Return s_2 with each occurrence of s_1 replaced by a third parameter, s_3.

C. Algorithm design

1. Write pseudocode or program code for an algorithm that takes as parameters an array of integers and its size, and that returns *true* if the array contains *three* or more identical elements. State its complexity in big-O notation and justify your analysis.

2. Code and test the solution to the Towers of Hanoi/Towers of Brahma problem. Program should input the number of disks to be used and should output detailed instructions, such as "Move disk from peg 1 to peg 3," etc., so that all disks will be moved from peg 1 to peg 3, one at a time, without putting a larger disk on a smaller one.

   Sample I/O:
   ```
   How many disks? 2
   Solution to Towers of Hanoi problem with 2
   disks:
   Move top disk on Left to Middle
   Move top disk on Left to Right
   Move top disk on Middle to Right
   ```

3. Code and test a recursive solution to the rabbits problem. Program should input a number of months and should output the number of rabbits that will be on hand after this number of months, assuming that we start with two rabbits and each pair of rabbits produces a new pair each month after a one-month childhood.
Sample I/O:
```
How many months? 12
There will be 144 pairs after 12 months
```
4. (a) Code the *power* method iteratively (do *not* use the standard *pow* method). This method takes two parameters, a *double* value and an integer, and returns the *double* value a^b if the parameters are a and b, respectively. Do *not* use the library method *pow*.
Sample I/O:
```
Enter 2 numbers: 2 3  2 ^ 3 = 8
Enter 2 numbers: 4 2  4 ^ 2 = 16
```
(b) Use your iterative method to calculate 2^{25}.
(c) Try using it to calculate 1.01^{50000} and 1.01^{100000} and explain your observations.
(d) Recode the method recursively and compare test results to those found in (b) and (c).

5. Write a program that declares and initializes an unsorted array of *int* and prompts the user for a single *int*. Define and call a method that takes an array, its size, and an integer *key* as parameters and returns the number of occurrences of *key* in the array. Your method may have this prototype:
```
int num_occur(int A[], int size, int key);
```
Sample I/O:
```
Search key: 6
The value 6 appears 2 times in the array
{2, 7, 3, 8, 9, 6, 3, 10, 2, 4, 6, 1}
```

D. Sorting

See Problem A, #2. Implement and test a sorting algorithm for your array. Discuss its time performance.

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Quick Sort
5. Merge Sort

Group work 2 (Linked structures)

A. Singly linked list

Using the collection skeleton from Group work 1 (#A-2), implement the collection as a singly linked list of integers. (See problem B on implementation.) Implement the following operation corresponding to your group number.

1. prepend
2. insert in ascending order
3. recursive search
4. reverse the list
5. recursive display
6. search
7. search and delete
8. circular list (see textbook)
9. selection sort
10. Bubble sort

B. Various linked-list implementations

Experiment with three ways to implement the linked list:

(a) two classes, a list class and a node class, where the list class has a header node;
(b) one class, for nodes, using a header node whose *next* points to the first list node that stores data;
(c) one class, for nodes, using a reference to *Node* to access the first node.

Show the class definitions including methods. You do not need to implement all operations for all classes.

Comment on these two ways to implement linked lists, saying which way you prefer initially.

C. Doubly linked list

Implement a doubly linked list, and code for DLLs the operation that is listed in Problem Set A *after the* operation you implemented for SLLs.

Group work 3 (Stacks, queues)

For problem sets A and C, each group should solve the problems corresponding to the group number. Post solutions at Discussion Board, Topic 3, listing with each solution the names of those who contributed significantly to it. For each method coded, write a comment stating the purpose of the method and the name of the student who did the main work writing the method.

A. RPN problems

Evaluate, showing operations. If expression is invalid, state reason.

1. 3 2 * 4 + 1 –
2. 4 3 2 1 – + *
3. 4 * 3 2 + 1 –
4. 4 3 * 2 1 +
5. 4 2 3 * + –
6. 1 4 + 2 3 * –

B. Implementing stack

Write a well-documented program that implements a stack of integers, using a linked list, with operations tested using a menu interface.

Write code to throw exceptions where appropriate. Test as described below.

See collection skeleton with menu from Group work 1, and see code you have already written for lists (Group Work 2). What you need to add will be a structure type or class definition for the stack data structure, and methods to implement the *is_empty, is_full, pop,* and *push* operations.

Call *push* and *pop* in response to the menu choices to *insert* and *delete-top*. Also implement non-destructive *display* and *search* menu options (note that these are not part of the interface of a stack).

Use these to test the stack ADT, and submit test results showing that insertions and deletions have the proper results.

C. Implementing queue

Do the same as in B, but define a queue of integers, replacing *pop* and *push* with *enqueue* and *dequeue* as defined in class and in the textbook. Show results.

D. Using stack

Using your stack class coded for Problem D, design, write and test code to:

1. Reverse a string
2. Check matching parentheses in a string
3. Syntax check an RPN expression
4. Match <i> and </i> tags in a line of an HTML file

E. RPN calculator (challenge; may replace D)

Design a reverse-Polish notation (postfix) calculator, using the stack class definition you created for B. Implement your design in Java, including a trace of stack operations in the output. This means just writing a new *main* that declares a stack and calls the *pop* and *push* methods for that stack. Use a loop to input strings, determine whether a string is a number or an operator, handle it appropriately, and terminate input on any non-number, non-operator. Test with a variety of valid and invalid input expressions.

Group work 4 (Heaps)

As you did for stacks in Group work 3, use your menu-driven skeleton program to implement a priority queue of integers using a *minimum*-heap structure. Use a *counter* to illustrate your complexity analysis; i.e., show time reuired for heap insertion or extraction operations given heaps of various sizes.

A. Heap class

(50%) Start by adding a class for the minimum heap, and write *heap_insert, extract_min,* and *heapify* methods that *main* will call. (See maximum heap of characters, coded in *charheap.cpp.*)

B. Testing

(50%) Test your code with some integers that could represent print-job-type priorities. Have your display option in *main* dequeue and display these values in priority order, by repeatedly extracting the minimum value that is on the heap. Append test results to your code.

C. Assertions (extra credit)

Choose one of your heap methods, *heap_insert* or *heapify*, to code *iteratively* (using *while*) and write assertions that argue for the method's correctness. Recall: *insert* adds to the end of the array, drags a value up the tree; *extract* moves the last element to the top and drags the value down; postcondition in both cases is that all root-to-leaf paths are ascending. Example of a loop invariant that could be used in the
Build-heap algorithm:

```
i ← ⌊size(A) ÷ 2 ⌋
while i ≥ 1
        // LI: Subtrees with root greater
        // than i are heaps
        Heapify(A, I )
        i ← i – 1
```

Group work 5 (Trees)

A. BST

As in Group works 2-4 for other kinds of collections, use your menu-driven skeleton program to implement a BST of integers. Implement standard general-collection operations or others:

1. insert
2. search
3. search-delete
4. display-all
5. find maximum depth of tree
6. count nodes

B. General tree (optional)

Modify the program *outline.c*, on handout, to permit updating and deletion of the current line of the outline and all its sublevels (but not deleting other lines at the same outline level). You may also wish to implement saving and retrieving outlines as text files. Test to show that your code deletes one node and its children, but not its siblings. (Note: your solution is *not* the same as *BST-delete* because the tree involved is not a binary search tree.)

Group work 6 (Hashing)

A. Implementations of hash table

Show results for larger and smaller tables:

1. Implement hash table of strings with linear probing. Show performance results for larger and smaller tables.
2. Implement hash table using double hashing.
3. Use an array of linked lists and resolve collisions by insertion of a record into a linked list (bucket) rather than by linear probing.

B. Variations of hash function

Test performance using hash table used for Problem A.

1. Test mid-squares hash function
2. Test ELF hash
3. Invent your own hash function, document the strategy.

Group work 6 (Hashing)

A. Implementations of hash table

Show results for larger and smaller tables:

4. Implement hash table of strings with linear probing. Show performance results for larger and smaller tables.
5. Implement hash table using double hashing.
6. Use an array of linked lists and resolve collisions by insertion of a record into a linked list (bucket) rather than by linear probing.

B. Variations of hash function

Test performance using hash table used for Problem A.

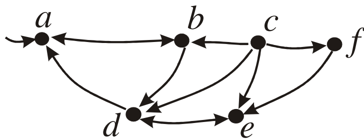
4. Test mid-squares hash function
5. Test ELF hash
6. Invent your own hash function, document the strategy.

Group work 7 (Graphs)

For the problems with numbered subproblems, each group should solve the subproblem corresponding to its group number. Submit solutions on paper if necessary, post at Discussion Board Topic 7 if in text form.

A. Graph representations

Draw a picture of two representations of the graph below, using a matrix of booleans and an array of lists.



B. Reachability

Use the algorithms specified below to determine which vertices are reachable from the specified source vertex.

1. From vertex *a* using Warshall's algorithm.
2. From vertex *c* using Warshall's algorithm.
3. From vertex *a* using depth-first search.
4. From vertex *c* using depth-first search.
5. From vertex *c* using breadth-first search.
6. From vertex *f* using breadth-first search.

C. Coding

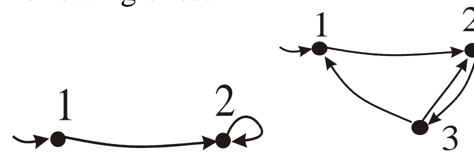
a. Define a Java class for a graph, represented as either an array of lists or as a matrix.

b. Write pseudocode or Java code for an algorithm that takes as parameters a graph (defined as in (a)) and a sequence of vertices, and returns *true* if the sequence of vertices defines a path in the graph, otherwise *false*. For example, in the graph above, *abda* and *cbde* define paths, but *bcf* and *ade* are not.

D. Kripke structures

A *Kripke structure* is a directed graph that models a process in which vertices represent states of the process and edges represent transitions that occur from state to state. Hence a Kripke structure can be considered as a triple (*S, m, r*), where *S* is a set of states, *m* is a set of ordered pairs of states denoting possible transitions, and *r* is a starting state.

1. Define a data type to represent Kripke structures. Initialize instances to represent the following ones:



   Define and test a method (for a given Kripke structure of the type you defined in (a)) that tells whether a given state (the method's parameter) is reachable from the initial stae by some series of transitions.
2. Define and test a method, with parameters *s* and *n*, that tells whether any paths of length *n* pass through state *s*. (Negation denotes *safety*.)
3. Define a method that tells whether *all* paths of length *n* pass through state *s*. (Denotes *liveness*)
4. Define a method that takes a path length *n* and a set of states as parameters, and tells whether *any* paths of length *n* pass *only* through states in a given set of states.

Group work 8 (Multithreading)

Group work 9 (Graphics programming)