Problem list for the Programming Techniques project, Spring 2011

Alex Muscar

March 2, 2011

Textbook problems

1. *The change-making problem*: "How can an amount of money be made with the least number of coins of given denominations?"

2. *The rod-cutting problem*: "Given a rod of length n and a table of prices for rods of length from 1 to n, determine the maximum revenue obtainable by cutting up the rod and selling the pieces. Note that an optimal solution may require no cutting at all."

3. *The eight queens problem*: "Place eight queens on an 8×8 chessboard so that none of them can capture any other using the standard queen's moves. The queens must be placed in such a way that no two queens attack each other."

4. *The task-scheduling problem*: "Schedule several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities. Given that an activity has a start time and a finish time two activities are compatible if the intervals between theirs start and finish times don't overlap."

5. Implement two algorithms for minimum spanning trees (e.g. Prim's algorithm and Kruskal's algorithm).

Libraries

1. A library for *safe* string manipulation. The library will implement alternatives for the most common string manipulation functions: `strcat`, `strcpy`, `strlen` and `gets` for reading strings. The library functions *must* perform sanity checks to avoid buffer overruns and to ensure that all strings are null-terminated after each operation.

2. A library for linked lists. The library should implement singly and doubly linked lists. The operations provided by the library should be: initialization of an empty list, adding a value at the beginning and at the end, inserting an item at a specified position, removing an item at a specified position, computing the length of a list and appending two lists.

3. A library for binary search trees (*BST*). The library should provide operations for: creating an empty tree, inserting a node into a tree, deleting a node and at least two different traversal strategies — the traversal functions must accept a function which will be passed the current element.

4. A library for priority queues using *heaps*. The library should implement both min-priority queues and max-priority queues. The library should provide operations for inserting an element with a certain priority, finding the maximum/minimum element, extracting the maximum/minimum element and altering an element's priority.

5. A library for hash tables. Collisions should be resolved by chaining. The size of the hash table should be dynamically adjusted to accommodate as many keys as necessary.

6. A library which implements at least two efficient sorting algorithms (e.g. quick-sort, merge-sort). The sorting functions should have in a similar way to the standard `qsort` function (i.e. they should work for elements of any type as long as an ordering function is provided).

7. A library for operations with large numbers. The library should provide at least addition, subtraction, multiplication, division and taking the square root.

8. A library for operations with sparse matrices. The library should provide at least addition and multiplication.

9. A library for associative arrays implemented using prefix trees (*tries*). The array will accept string keys and values of any type and it will expose the following functionality: adding a new value indexed by a key, reassigning the value for a key and lookup for key values.

10. A library for representing oriented graphs. The library should allow to insert vertices, remove vertices, traverse the graph depth-first and breadth-first.

11. A library for disjoint sets. Given a set of elements it is often useful to break them up or partition them into a number of separate, non-overlapping sets. A disjoint-set data structure is a data structure that keeps track of such a partitioning. The library must allow to determine which set a particular element is in (also useful for determining if two elements are in the same set) and combine or merge two sets into a single set.

Applications

1. Since programming is such a sedentary job programmers need to watch their weight. The programmers in a company have access to all sorts of snacks in the cafeteria (with varying caloric intake), but they also have access to a gym and various activities that help them burn calories so that they maintain their weight. Given a list of activities and their caloric impact write a program that finds a combination of activities that keep the caloric intake to 0. Your program should take its input from a file. You can use any format for the input as long as it gives the name of the activity and the caloric intake (which should be a positive for snacks and negative for activities that burn calories). Print the list of activities to *stdout* if a solution is found or the message "no solution" otherwise.

2. Write an interpreter for a simple language that recognizes both negative and positive integer constants, variable assignment (e.g. `a = 1 * 2`) and the basic mathematical operators: `+`, `-`, `*`, `/` and `^` for exponentiation. Once read the expressions should be represented as binary trees which will be used to compute the result. The interpreter should present the user with a Read Eval Print Loop (*REPL*).

3. Write an interpreter for the BF language.

4. Write a simple shell with the following built-in functionality: listing files (sorted in natural order), showing the content of a file and showing the number of words and lines in a file. You can see how the linux `ls -l`, `cat` and `wc` commands behave for inspiration.

5. Write an interpreter for evaluating polynomials represented using linked lists. The interpreter should allow for addition, subtraction and evaluation of a polynomial. The interpreter should provide a syntax similar to that specified in problem 2.

6. Write a dictionary application. The dictionary should allow for inserting a word, updating the definition of an existing word and looking up definition of a word. Note that a word may have *multiple* definitions attached to it. The dictionary should use files for storing and loading its data.

7. Write a program that generates random text that reads well by using the *Markov chains*. The program will read text from a file. Start by reading the first two words (i.e. sequences of characters delimited by one or more spaces) from the file, say `w1` and `w2`. Using the pair `(w1, w2)` as a key insert the the word following them, `w3`, in a hash table. Replace the pair `(w1, w2)` with `(w2, w3)` and repeat until the end of the file. Note that each key made out of a pair of words will have a list of words as a value. In order to generate text start with the first key in the hash table `(w1, w2)`, print `w1` and `w2`, and then randomly pick one of he words in the

list associated to the key, `w3`, and replace the key by `(w2, w3)`. Repeat the process for a fixed number of steps.

8. Write a spell checker using the Levenshtein distance. The spell-checker will take an input file and it will output a list of misspelled words and the suggestions for them on *stdout*.