Assignments for Data Structures

Mark Allen Weiss
School of Computer Science
Florida International University
University Park
Miami, FL 33199

Abstract

A collection of assignments that I have used in my data structures classes is provided. In some cases a sketch of the necessary algorithm is provided, while in others, the instructor will need to spend time to describe the algorithm in detail.

1. **Basic Assignment**

   Write a program that solves the word search puzzle problem described in Chapter 1.

2. **Specifications**

   Your program should ask for the name of the puzzle file. The puzzle file will look exactly like the puzzle in the text but with no row or column numbers and no blanks. You may assume a limit of 64 letters per row or column, and you may convert everything to lower case. The exact puzzle dimensions are not given, but you can infer them from the puzzle. Read the puzzle into main memory.

   Your program should then ask for a list of words to search for. For the purpose of this assignment, you may assume that if there are r rows and c columns, then there are approximately $rc/7$ words (this is typical of most puzzles). You must find all occurrences of every word in the word list. Some words may appear more than once and some might not appear at all.

   It is guaranteed that the word list is sorted. You should use the second algorithm suggested in the book (on page 2); however, since the word list is sorted, you can use a binary search to check if a suggested word is in the dictionary.[1]

   Needless to say, you must check for all possible input errors, and do reasonable things when they are detected.

[1] An improvement to the algorithm is to check if the suggested word is a prefix of some word in the word list. If it is not, then searching in the current direction can be terminated. The prefix test can be incorporated into the binary search because in case of failure, the word is a prefix of a word in the word list if and only if it is a prefix of the last word examined by the binary search.

1. **Basic Assignment**

   Implement an infix expression evaluator.

2. **Specifications**

   Each infix expression is on a separate line. Read each line until the end of input is signaled (ctr-Z on VAX, ctr-D on UNIX) and output the result of the calculation.

   You must handle operations involving positive integers, `+`, `-`, `*`, `/`, `^`, `(`, `)`.

   You should write a package called `Infix` that has the single public function `EvaluateLine`. The function evaluates a string that is in infix form and returns its value.

   Your package will need to instantiate two stacks. You may use the stack implementation of your choice; if you use the array implementation then instantiate both stacks with a maximum stack size of 100.

   You must do reasonable error checking, and should not crash if the user gives you an illegal expression.

1. **Basic Assignment**

   You are given k unsorted files containing a total of n elements. Your assignment is to merge the files into one big sorted file, combining duplicates where appropriate. n will be about 100,000.

2. **The Input**

   The files are named *merge001.dat*, *merge002.dat*, ..., *merge300.dat*. Each file contains a list of ordered pairs. The first value is a `long_float`, and the second is an `Integer`. One or more files may be empty or may not exist at all, so you should check for error conditions. The output file is to be named *result.dat*.

3. **Strategy**

   Read the files into sorted linked lists. The sorted order of the linked list is based on the first value in the ordered pair.

   Next, collapse each sorted linked list so that consecutive entries (x, y_1), (x, y_2), ..., (x, y_i) are replaced by a single entry $(x, y_1 + y_2 + ... + y_i)$ (i.e. a linked list will not have duplicate first values).

   After this collapsing, each sorted linked list is logically put on a queue. Note carefully that what you are actually putting on the queue is a pointer to the sorted linked list, because sorted linked lists are limited private types. Thus you actually have a queue of pointers to sorted lists.

   While the queue has more then one list on it, take two lists off, merge them, and put the result back on the queue. If you find an entry (x, y_1) in one list and (x, y_2) in another, put $(x, y_1 + y_2)$ in the merged list. You should use a linked list queue implementation that supports constant time access (enqueues are at the rear, dequeues at the front of the list).

   To avoid memory leaks, you should be disposing of your linked lists after they have been merged into a larger one. Both the linked list and queue package should have a `Finalize` procedure defined; the queue package may also need an `Initialize` procedure defined.

1. **Basic Assignment**

   Write a program that reads an Ada program and lists all identifiers, and the line numbers on which they occur. Do not output any of Ada's reserved words. Also, skip over comments and string constants (but not character constants, because parsing the ' is tricky).

2. **Specifications**

   Since Ada is not case sensitive, convert characters to lower case as they are read. An identifier in Ada is a sequence of letter, digits, and the underscore character, but an identifier must start with a letter. When an identifier is seen, verify, via a binary search of a prestored array, that it is not a keyword.

   Otherwise, make an update to a binary search tree. The binary search tree stores items of the type `Word_Entry` shown below; you must also provide a comparison function that compares `Identifier` fields and a `Put` function for output. If the word is already in the tree, then simply enqueue into the queue. The queue can be accessed by a `Find`, `Retrieve`, and dereference combination. Otherwise, the word is new: in a `Word_Entry` record, place the word, dynamically allocate a new queue, place the current line in the queue, and then call the insertion routine.

   Here is an excerpt of some of the incantations you will need.

```
-- Prior to this point, we have code to instantiate a
-- Queue of integers. This includes the needed with statements.
-- Now we do some of the rest.

type Ptr_To_Queue is access Queue;

type Word_Entry is record
    Identifier : StringType;
    Lines      : Ptr_To_Queue;
end record;

function "<"( Lhs, Rhs: Word_Entry ) return Boolean;
procedure Put( Item: Word_Entry );

package My_Tree is new Search_Tree_Package( Word_Entry, "<", Put );
use My_Tree;
```

1. **Basic Assignment**

   Generate an index for a book.[2]

2. **The Input**

   The input file consists of a set of index entries. Each line consists of the string `IX:`, followed by an index entry name enclosed in braces, followed by a page number that is enclosed in braces. Each `!` in an index entry name represets a sub-level. A `|(` represents the start of a range and a `|)` represents the end of the range. Occasionally, this will be the same page, and in that case, only output a single page number. Otherwise, do not collapse or expand ranges on your own. As an example, here is a sample input:

```
IX: {Series|(}          {2}
IX: {Series!geometric|(}        {4}
IX: {Euler's constant}        {4}
IX: {Series!geometric|)}        {4}
IX: {Series!arithmetic|(}        {4}
IX: {Series!arithmetic|)}        {5}
IX: {Series!harmonic|(}        {5}
IX: {Euler's constant}        {5}
IX: {Series!harmonic|)}        {5}
IX: {Series|)}          {5}
```

   The corresponding output is

```
Euler's constant: 4, 5
Series: 2-5
   arithmetic: 4-5
   geometric: 4
   harmonic: 5
```

3. **Strategy**

   Each unique index entry will be represented as a record. These records will be stored in a binary search tree. An inorder traversal of the binary search tree will output the index in alphabetical order. The record that represents the index entry will contain a string and a pointer to a queue that stores information about the page numbers. The information on the queue is a page number and an indication as to whether this entry is a simple page, the start of a range, or the end of a range.

   When a word w on page p is read, we first check if it is in the binary search tree. If it is, we update the queue corresponding to w. If w is new, we add a record to the binary search tree. The inorder traversal of the tree will yield the words. The corresponding line numbers can be extracted by analyzing the sequence that results from emptying the queue.

   You may assume that every line of the input is properly formatted. You must verify that `|(` and `|)` are properly matched (this can be done when the line numbers are analyzed at the end of the program), and print a warning message if they are not. As usual, you must not leak memory.

[2]This is essentially the previous assignment with some extra goodies. Examine the discussion there for Ada hints.

1. **Basic Assignment**

   Given a set of required courses and a prerequisite list, you are to determine the minimum number of semesters required to graduate (assuming that you can take as many courses in a semester as you like, provided you do not violate a prerequisite requirement).

2. **Brief Description**

   The input file is of the following format: The first k lines contain one entry each, indicating a required course. Course names consist of letters and digits only, and are not case sensitive. **The first k lines are sorted.**[3] After that, all lines are of the form

   ```
   course:  prereq1 prereq2 ...  prereqn
   ```

   You must output the list of courses which can be taken in semester1, then semester2, etc., and you must use the minimum number of semesters. The answer is unique, though courses in a given semester may be output in any order.

3. **Basic Algorithm**

   Throughout the algorithm, refer to any course by its position in the list of required courses. You can find the position quickly by reading the courses into an array and applying binary search whenever necessary.

   Next, for each course, keep a linked list of all the prerequisites. Since, as per the previous paragraph, courses are represented by a natural number between 1 and k, what you have is an array of list of naturals. Also, for each course, attach a boolean, initially "marked" `False`, and the semester in which it can be taken, initially marked 1. Finally, keep an array which indicates, for each course, the number of *postrequisites* (the number of courses for which it is a prerequisite).

   Third, verify that the prerequisite structure is not bogus (that is, there is no cyclic chain of prerequisites). This can be done by a topological sort. See the start of Chapter 9. You will need a queue for this part. If the prerequisite listing is bogus, then stop.

   Finally, compute the answer by applying the recursive algorithm below. The recursive function is applied to all courses which are not prerequisites to any other course.

   ```
   function Compute_Semester( Course: Natural ) return Natural is
       How_Many : Natural := 1;
   begin
       if Course is Marked known then
           return the known information;
       else
           Mark Course as known;
           for each prerequisite course P loop
               How_Many := Max( Compute_Semester( P ) + 1, How_Many );
           end loop;
           return How_Many;
       end if;
   end Compute_Semester;
   ```

[3] A later assignment can remove this restriction and use a binary search tree.

1. **Basic Assignment**[4]

   Write a program that will repeatedly insert elements into an initially search empty tree and the print out a graphical representation of the tree.

2. **Specifications**

   Start with an empty search tree. Repeat until the user gives up: Ask for an integer $0 \leq i < 1000$ to insert into the tree. After each insertion, print out the tree. The root goes on one line. The nodes at depth one go on the next, depth two on the next, etc. You must make sure that any nodes in the left subtree of x are printed to the left of x and nodes in the right subtree are printed to the right of x.

   To do this, add two fields to your tree data structure. One is `Inorder_Num`, which is the inorder traversal number of any node. Calculate this each time after an insertion, prior to printing the tree. The other is the `Depth`, which is the depth of each node. This just tells you when to print newlines. Use a queue to perform the level-order traversal.

[4] Variations include: an AVL tree or splay tree.

1. **Basic Assignment**

   The Josephus problem is described in Exercise 3.10. Write an efficient program to solve the Josephus problem.

2. **Algorithm**

   Using a linked list is computationally inefficient when M and N are large because the running time would be $O(MN)$. Use the following alternative, which provides an $O(N \log N)$ algorithm:

   Initialize a binary search tree to have numbers 1 to N, inclusive. **Do not do this by successive insertions**, because that will give a very unbalanced tree. Instead, write a recursive routine.

   Each node in the binary search tree should store the size of the tree it roots. This will allow you to write the function

   ```
   function FindKth( T: Tree; K: Integer ) return Tree_Ptr;
   ```

   that returns the `Kth` smallest item in the tree.

   Each step of the Josephus problem then reduces to the following:

   - Find a new value of `K` (the previous value plus M, all taken MOD the number of remaining players).
   - Do a `FindKth` and delete from the tree the item that corresponds to it.

   Note that the deletion routine will now have to adjust the size fields of some nodes in the tree. It is up to you to figure out which ones are adjusted.

   Test your program for M = 40000 and N = 5000. Also test your program for M = 40000 and N = 1 (Answer is 14465) and M = 40000 and N = 0 (Answer is 40000, of course).

1. **Basic Assignment**

   Write a spelling checker. Any word that is not found in a dictionary file is to be considered misspelled. Ignore any case distinctions for the purposes of determining correct spelling, but output misspelled words exactly as they occur in the input file.

2. **Specifications**

   You will use a hash table to store the dictionary. Each misspelled word and the line numbers on which it occurs are to be placed into a binary search tree in the same manner as was done for the Ada cross reference generator or index program described in earlier places.

   Output all misspelled words and the line numbers on which they occur.

   A word is a maximal sequence of three or more alphabetic characters. Everything else should be treated as white space.

1. **Basic Assignment**

   You have a list of items $i_1, i_2, ..., i_n$ which have integer weights $w_1, w_2, ..., w_n$. You need to place the items into boxes but each box can hold at most 100 lbs. The *bin-packing* problem is to find an arrangement of the items such that the fewest number of boxes (bins) are used.

2. **Theory**

   Bin-packing is one of many problems that are $NP - complete$ (see the discussion in Section 9.7). Most researchers in the area believe that exponential running time is intrinsic to NP-complete problems. These problems still need to be solved, so algorithms have been developed to give good, fast approximations.

3. **An Approximation Algorithm**

   The algorithm you will use is simple and fast. When an item comes in, it is placed into the most empty box. If this box cannot hold the item, it is placed in a new box. Keep the boxes stored in a binary heap with the most empty box at the root.

4. **Specifications**

   Ask the user for the input file name. Ask the user if she/he wants the play-by-play description. If so, then as each item is processed, print out which box it goes in to.

   At the end of the program, print out the total number of boxes used and the total weight of all items in the boxes.

1. **Basic Algorithm**

   Write a program that reads a sequence of integers from a file and outputs three numbers in the sequence whose sum is exactly zero. Numbers may be used more than once. Example: if the input is 3, -2, 5, -4, -1, then the answer is yes, and the three numbers whose sum is zero are 3, -2, and -1. There are other answers for this input; you are only required to produce one.

2. **Specifications**

   The input items are contained in a file that is given as a single command line argument.

   The routine that determines if there are three numbers that sum to zero should return either TRUE or FALSE and should use out parameters to indicate what the three numbers are. The routine absolutely may not print anything!

   It happens that the number of items will be about 20,000. However, if you attempt to use a triple for loop to solve the problem, you will have an $O(N^3)$ algorithm that is too inefficient. Instead, use a quadratic algorithm. This algorithm will require that you sort the items. Use heapsort. After the items are sorted, use the following pseudocode:

```
for i in A'range loop
    j := i;
    k := A'last;
    Compute  A( i ) + A( j ) + A( k ), and
        if 0  => we have found a solution; return TRUE;
        if positive => decrease k
        if negative => increase j
    exit when k < j;
end loop;
return FALSE;
```

1. **Basic Assignment**

   This assignment is to compare the performance of median-of-three quicksort with immediate insertion sort of small subarrays vs. a single final insertion sort. Recent research suggests that on modern computers, the immediate insertion sort is more efficient than the single final insertion sort, contradicting previous beliefs.

2. **Specifications**

   Measure the average running time using both algorithms by running each algorithm ten times for arrays of input sizes 1000, 2000, 4000, 8000, ..., 64000 which contain integers. Summarize your results in a table or graph.

   Make sure that the `Swap` routine is inlined. Also write a routine to check that your array is actually sorted. Only include the time from the sorting algorithms, and not other time such as generating random inputs and checking output.

   Chapter 10 describes a random number generator that is sufficient for these purposes (since you need an integer, do not do the last division in that routine, but just return `Seed`). Initialize `Seed` to your social security number once, at the start of your program.

   After you have done tests for random input, repeat the tests for 64000 identical integers, and also 64000 integers which are already sorted (mostly to make sure your algorithms work in reasonable time).

1. **Basic Assignment**

   Convert one five letter word into another by means of single character substitutions. As an example, bleed can be converted to blood by the sequence bleed, blend, blond, blood. All words in the sequence must be in a dictionary of five letter words, which will be provided in a file.

2. **Basic Algorithm**

   This is just an unweighted shortest path problem. Each word is a vertex in the graph. If two words X and Y are identical except for one position, then there is an edge from X to Y and from Y to X.

   Read the words from the dictionary file into an array, and sort the array. Declare a record with the following fields: a word, distance (initialized to `Integer'last`), path (initialized to 0), and a list (initialized to empty). Then declare an array of records (the array should be indexed from 1 to the number of words read). The word field of the kth record should be a copy of the kth word. You should write a procedure `Get_Word_Pos` which takes a word and returns the index of the corresponding record.

   Next, find all pairs of words which agree except in one position. To do this run 5 partial sorts. Partial sort i treats a word as if the ith character was moved to the end of the word. Partial sort 5 is a normal sort. Partial sort i will guarantee that words that are identical except for the ith character are very close together. You will need to work out the specifics (think about where loose, moose, noose, goose, wind up among other words after partial sort 1). A partial sort can be done by instantiating a quicksort with an appropriate comparison function.

   When a pair X, Y is found, compute their positions in the array via a call to `Get_Word_Pos`, and insert two edges into the graph (in other words, add one entry to each of two adjacency lists).

   When all the pairs are found, the graph has been set up. Repeatedly prompt the user for two words, run the unweighted shortest path algorithm, and print a path if it exists. You must print the actual transformation.

1. **Basic Assignment** [5]

   Use a rehashable, generic, quadratic probing hash table to process a file containing the following commands:

`a word`	Adds `word` to the dictionary, assuming it is not already present. Does nothing if `word` is already in the dictionary.
`r word`	Prints the rank of `word` (i.e. the order in which it entered the dictionary), or a message that `word` is not in the dictionary.
`n`	Prints the number of words currently in the dictionary.

   A word is any sequence of letters, digits, and punctuation, but does not include white space.

[5] This is a first warmup for Dijkstra's algorithm.

1. **Basic Assignment** [6]

   Use a dynamically expanding, generic, priority queue to process a file containing the following commands:

`n numitems`	Set the number of items. Initialize an array of booleans to false.
`a item priority`	Insert an item with specified priority. This item might already be in the priority queue. If so, this is a duplicate that might have a lower priority.
`d`	Repeatedly extract items from the priority queue until an item never before seen is extracted. Set its entry in the boolean array to true.
`p`	Same as `d`, but print out the number and priority of the extracted item.

2. **Specifications**

   An item is an integer from 1 to the value specified in the initial `n` command. A priority is a `Natural`. You must, of course, flag any bad input lines as errors, skip the line, and continue. The number of insertions into the priority queue will be much larger than the number of different items.

   Example:

`n 5`	(Five items 1, 2, 3, 4, 5)
`a 1 4`	
`a 2 3`	(Two items are in the priority queue)
`d`	(Out comes item 2, priority 3)
`p`	(Print out item 1, priority 4)
`a 1 5`	
`a 3 6`	
`d`	(Out comes item 3, priority 6, after 1,5 is removed and ignored)
`d`	(Error: priority queue is empty)
`a 4 7`	
`a 5 7`	
`d`	(Either 4,7 or 5,7 comes out – it is a tie)
`d`	(The other one comes out now)

[6] This is a second warmup for Dijkstra's algorithm.

1. **Basic Assignment**

   You are to write a mail router for a computer. Your program will read a database of computer connections and decide (interactively) the shortest number of hops between any two vertices.

2. **The Input**

   Prompt the user for the name of an input file and then read it. Each line of the input file represents a connection between two computers. It contains the sending computer, the receiving computer, and the cost of transmitting a block of data.

3. **The Program**

   Ask the user for a file name. Read the file in. Then repeat the following steps: Ask the user for the name of the start vertex. Compute all the shortest paths from the start vertex using Dijkstra's algorithm. Then ask the user for the destination node.

   The output should be in UUCP mail format. Thus if the path from computer `x` to computer `a` goes from `x` to `y` to `z` to `a`, the mail address would read `x!y!z!a`. Print the total cost of transmitting a block of data.

4. **Data Structures**

   You will need four data structures to solve this problem:

   (a) A hash table to map the computer names into vertices
   (b) An adjacency list representation of the graph
   (c) A priority queue
   (d) A table to keep track of the answers and to convert the internal vertex numbers to computer names for output