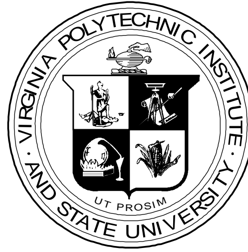


Instructions:

- Print your name in the space provided below.
- This examination is closed book and closed notes, aside from the permitted one-page formula/fact sheet. No calculators or other computing devices may be used.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are 7 questions, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- **Note that failure to return this test, or to discuss its content with a student who has not taken it, is a violation of the Honor Code.**

Do not start the test until instructed to do so!

Name Solution

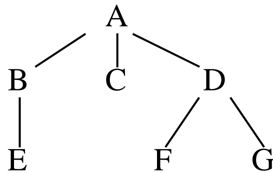
Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

____

1) (a) [8 points] Define the overhead fraction F for a data structure implementation.

Overhead is the amount of storage used for structural information (e.g., pointers, unused array cells, etc.) as opposed to actual logical data. The overhead fraction is the ratio of the overhead to the total storage.

(b) [8 points] Sketch the left child/right sibling data structure for representing the tree given below. Assume that parent pointers are used and list the nodes in alphabetical order.



Index	0	1	2	3	4	5	6
Left child	1	4	-1	5	-1	-1	-1
Data	A	B	C	D	E	F	G
Parent	-1	0	0	0	1	3	3
Right sibling	-1	2	3	-1	-1	6	-1

(Nulls are represented by –1.)

(c) [8 points] Calculate the overhead fraction F for the left child/right sibling general tree implementation, for a tree containing a total of N nodes. Assume parent pointers are used, that the data value stored in each node occupies D bytes, and that an integer index occupies X bytes.

Each "node" stores one data item and three integer indices, and there are N nodes, so the total storage required would be N(D + 3X).

The overhead in each node consists of the three integer indices, so the total overhead is N(3X).

Therefore the overhead fraction would be N(3X)/N(D + 3X) or 3X /(D + 3X).

2) [12 points] For each of the following sorting algorithms, give big-Theta notation for the best, average, and worst-case running times (swaps plus compares) on input of size n:

Algorithm	Best	Average	Worst
selection sort	n^2	n^2	n^2
quicksort	n log(n)	n log(n)	n^2
heap sort	n log(n)	n log(n)	n log(n)

3) [10 pts] Assume there is an algorithm `Split4( )` that can split a list `L` of `n` elements into 4 sublists, each containing one or more elements, such that sublist `i` contains only elements whose values are less than all elements in sublist `j` for `i < j <= 4`. If `n < 4`, then `4 - n` sublists are empty, and the rest are of length 1. Assume that `Split4( )` has time complexity O(n). Furthermore, assume that 4 lists can be concatenated together again in constant time. Consider the following sorting algorithm:

```
LIST Sort4(LIST L) {
    LIST sub[4];                                   // to hold the sublists
    if (LENGTH(L) > 1) {
        Split4(L, sub);                            // split the list
        for (int i = 0; i < 4; i++)                // sort the sublists
              sub[i] = Sort4(sub[i]);
        L = Concatenate(sub);                      // concatenate sublists
    }
    return L;
}
```

What is the best-case asymptotic running time for `Sort4( )`? Under what circumstances would the best case be obtained? (Hint: think about QuickSort.)

The best case occurs when Split4 partitions the list into equal sections. In that case there will be $\log_4(n)$ levels of recursion before the sublists reach length 1. The cost of applying Split4 at each level would be bounded by n.

Therefore, in the best case, the running time would be bounded by $n \log_4(n)$.

4) Consider the following "sorting" problem. You are given a collection of n bolts and n nuts. For each bolt in the collection, there is at least one nut of the same size (and so for each nut, there is at least one bolt on which it will fit). You do not know, initially, which nut matches which bolt. Furthermore, the differences in size between two nuts or two bolts may be too small to detect visually; so the only way to compare sizes is to attempt to screw a particular nut onto a particular bolt, which may reveal that the nut is too small for the bolt, just right, or too large. Attempting to screw a nut onto a bolt is a constant-time operation.

(a) [10 points] Describe an efficient algorithm for matching all the nuts with corresponding bolts. Your description does not have to be in pseudocode, but it must be clear and complete.

For convenience, label the bolts B_k and the nuts N_k, where $1 <= k <= n$.

Pick up bolt 1 and try each of the nuts against bolt 1.
  If the nut is the first to fit, screw it onto bolt 1.
  If the nut is too small, set it to the left.
  If the nut is too large, set it to the right.
  If the nut fits but is not the first to fit, set it to the right.

Now we have one nut/bolt pair and a partitioning of the nuts.

Pick up bolt 2 and try the nut that fit bolt 1.
  If it fits or is too small, repeat the process above with bolt 2, comparing it to the nuts on the right.
  If it is too large, repeat the process above with bolt 2, comparing it to the nuts on the left.

Now we have two nut/bolt pairs and have a three-way partitioning of the nuts.

Continue in this manner, comparing each bolt to the matched nuts to determine which set of nuts it should be compared to, and then comparing the bolt only to those nuts. Note: you can apply a "bisection" approach to this comparison, rather than a sequential search.

This is conceptually similar to quicksort in that the nuts are successively partitioned. This will terminate after n steps since one bolt is successfully matched at each step. How much does a step cost? Well, bolt k must be compared to up to k – 1 matched nuts, costing log(k – 1) if done right, and then to one "partitioned" set of nuts. If the partitions are even, there would be about $n/2^{k-1}$ nuts in that partition. This works out to n log(n) + log(n) cost.

(b) [10 points] For the best possible algorithm in part (a), what is big-O for the number of comparisons that would be required in the best case? Justify your answer.

The cheap answer is that it's essentially a sort of 2n items and comparisons must be used, so the theoretical minimum would be 2n log(2n). The algorithm above achieves this.

5) [12 points] Recall the physical geometry of a disk system. For each of the options listed below (which do not change the total capacity of the disk), determine the effect that option would have on the indicated timing (none, increase, decrease). Consider only the physical operation of the device, disregarding cost of logic operations in the disk controller.

Option	Seek time	Latency time	Transfer time
Double the number of platters and halve the number of tracks per platter.	decrease	no change	no change
Double the number of tracks per platter, keeping the same track spacing and halving the number of platters.	Increase	no change	no change
Double the rotational speed.	no change	decrease	arguable

6) [12 points] For each of the following protocols for a self-organizing list, state clearly one situation in which the protocol would be less effective than desirable, or even counterproductive, and explain why.

(a) move-to-front

This over-rewards a seldom-accessed record, possibly at the expense of frequently-accessed records.

(b) order by access count

This retains too much access history. A record may be accessed very frequently for a short time, building up a high count which keeps it near the front of the list, and then have very infrequent accesses.

7) [10 points] Recall that a selection sort works by finding the smallest value in the unsorted portion of the list and moving that element to the proper position.

Write a SelectionSort function for integer key values. However, there's a catch: the input is a stack (**not** an array), and the only local variables that your algorithm may use are a fixed number of integers and a fixed number of stacks. (No, your function may not use any global variables.) The function will also take a second stack parameter in which to return the result; you may assume this stack is initially empty. The returned result stack should contain the items from the original stack in sorted order, with the smallest value at the top of the stack. Your algorithm should be $\Theta(n^2)$ in the worst case. A partial `Stack` class declaration has been provided for your reference.

```
class Stack {
public:
   void Clear();                 // clear the stack
   void Push(int element);       // push element onto stack
   int  Pop();                   // pop and return top element
   int  topValue();              // retrieve value of top element
   bool isEmpty();               // check if stack is empty
}
```

```
void SelectionSort (Stack input, Stack& result) {

   int tVal, MinSoFar;
   Stack tStack;

   if (input.isEmpty()) return;            // null check

   while (! input.isEmpty()) {             // while there are unsorted values

      MinSoFar = input.Pop();              // initialize minimum

      while (! input.isEmpty) {            // find min value in input stack

         tVal = input.Pop();
         if (tVal < MaxSoFar)
            MinSoFar = tVal;

         tStack.Push(tVal);                // cycle input stack to local stack
      }

      while (! tStack.isEmpty()) {         // now cycle back to input stack

         tVal = tStack.Pop();

         if (tVal == MinSoFar)
            result.Push(tVal);             // moving minimum to result stack
         else
            input.Push(tVal);              // to input stack otherwise
      }

   }  // end of main loop

}  // end of SelectionSort()
```