CORE EXAMINATION
Department of Computer Science
New York University
September 28, 2007

This is the common examination for the M.S. program in CS. It covers core computer science topics: Languages and Compilers, Operating Systems, and Algorithms. The exam has two parts. The first part lasts three hours and covers the first two topics. The second part, given this afternoon, lasts one and one-half hours, and covers algorithms.

Use the proper booklet or answer sheet for each question. Each booklet is marked with the Area and Question number in the form PL&C1, PL&C2, PLC&C3, OS1, OS2, ALGS1, ALGS2. However, there is an answer sheet for the Algorithms problem 3 instead of a booklet. Use the appropriate booklet/answer sheet for each question. DO NOT put your name on the exam booklet. Instead, your exam number must be on every booklet.

You will be graded according to your exam number, shown on the envelope containing the booklets. Remember your exam number: when grades are given out, they will be published according to this number, not by name.

Make sure your name and signature are on the envelope. This is the only place where your name appears. Please include all the booklets inside the envelope and the answer sheet. You can keep the exam.

Good luck!

Algorithms

Question 1

This question deals with the problem of finding the intersection of two **sorted arrays** A and B. For simplicity, please assume that the elements of A are distinct, and likewise the elements of B are distinct. You can use any standard algorithm, provided you state exactly what the algorithm does.

1. [4 points] In this question, we assume that the two arrays are relatively short, e.g., $|A| = |B| = 1000$. Describe an algorithm to print the intersection and write the pseudocode. What is the complexity of your algorithm as a function of $|A|$, when $|A|$ and $|B|$ are approximately equal?

   Solution A.

   Merge the two sets and scan them for consecutive values that are equal. Both steps take $\Theta(|A| + |B|)$ time.

   Solution B.

   For specificity, let the indices of the arrays begin at 0, so that the A-index goes from 0 to $|A|-1$. For simplicity, let $A[|A|] = B[|B|] = \infty$. An algorithm that performs a linear scan of the keys in increasing order is as follows:

   $adex \leftarrow 0$;
   $bdex \leftarrow 0$;
   repeat
   if $A[adex] < B[bdex]$ **then** $adex \leftarrow adex + 1$
   elseif $A[adex] > B[bdex]$ **then** $bdex \leftarrow bdex + 1$
   else
   Print($A[adex]$);
   $adex \leftarrow adex + 1$;
   $bdex \leftarrow bdex + 1$
   endif
   until $adex = |A|$ and $bdex = |B|$;

   The operation count is $\Theta(|A| + |B|)$ because each iteration increases an index of A or B, and each iteration does constant work.

2. [6 points] Now assume that B is much larger than A. For example, let $|A| = 100$ and $|B| = 10^9$. Give a better-suited algorithm for this case. What is the complexity of your algorithm as a function of $|A|$ and $|B|$ in the case $|A| < |B|$?

   Solution A.

   Let $Bsearch(B, key)$ be a Boolean function that uses binary search to return true if key is in the array B, and false if not.

   One solution reads:

   for $j \leftarrow 0$ **to** $|A| - 1$ **do**
   if Bsearch$(B, A[j])$ **then** Print$(A[j])$ **endif**
   endfor

   The operation count is $\Theta(|A| \log |B|)$ because each key in A is processed just once, and the processing requires a binary search over B.

   A somewhat better solution (which is NOT required) can be devised by, for example, using binary search for the middle value $A[\frac{|A|}{2}]$. The search will split B into a sequence of values that are less than $A[\frac{|A|}{2}]$ and a sequence that is larger. Then the search for common elements

is applied recursively for $A[0..\frac{|A|}{2}-1]$ and the subsequence of B that is less than $A[\frac{|A|}{2}]$ and for $A[\frac{|A|}{2}+1..|A|-1]$ and the subsequence of B that is greater than $A[\frac{|A|}{2}]$.

The operation count is subtle, and is $\Theta(|A|\log(\frac{|A|+|B|}{|A|}))$. An unreasonably sophisticated solution might use some version of interpolation search, and other considerations would be appropriate for extremely large data sets.

Question 2

Let T be a balanced search tree such as a 2-3 Tree (or a balanced binary search tree) so that $find(x)$, $delete(x)$, and $insert(x)$ can execute in $\Theta(\log n)$ operations when T stores n records.

1. [5 points] Let $rank(x)$ be the number of records in T that have key values which are less than or equal to x. For simplicity, you can assume that no two records have the same key value. Give a high-level description of the enhancements necessary so that $rank(x)$, $find(x)$, $insert(x)$ and $delete(x)$ can all run in $\Theta(\log n)$ operations. This description should include:
   One sentence to describe the enhancement(s).
   One sentence to explain the changes when a new element is inserted into T. It in NOT necessary to explain what happens when a node splits (for a 2-3 Tree) or a rotation occurs (for some other balanced search tree.)
   One short sentence about why the operation count is still $\Theta(\log n)$.

   Solution.

   Use a 2-3 Tree. In each internal node v, maintain a count of the number of data leaves in the subtree rooted by v.

   These leaf counts are easy to update during an insertion or deletion with constant work per node because the nodes with values that change are along the path from the root of T to the inserted or deleted leaf. Splits and joins are easily processed, since the leaf count in any v is the sum of the counts for its children (or the number of children if they are leaves).

   The rank of a key x is one plus the sum of the leaf counts in the roots of the subtrees that hang off of (and to the left of) the path from the root of T to x. That is, for each v on the path through to x, the leaf counts for v's zero, 1, or 2 left-siblings are added to compute a running sum, which, when increased by 1, equals $rank(x)$. The count for each leaf that is a left-sibling of the leaf x is one (leaf). Since there are just $O(\log n)$ values that are added together, the total operation count is as requested.

2. [5 points] Now suppose we also wish to search by rank as well as by key value. So let $GetR(r)$ return the key with rank r.

   (a) Does your solution to part 1 contain enough information to compute $GetR(r)$ in $O(\log n)$ operations? If not, briefly present additional enhancements that would allow $GetR(r)$ to have this performance while maintaining the $O(\log n)$ operation counts for $rank$, $insert$, $delete$, and $find$.
   Solution.
   Yes. See below.

   (b) Briefly explain why $GetR(r)$ can be computed efficiently with your final structure. No algorithm is requested; all that is needed is a brief explanation about how $GetR(r)$ would work.
   Solution.
   Just as the leaf counts for the subtrees rooted by the internal nodes of T can be used to determine the rank of a key x, so can they be used in search for the key with a specified

rank r. The nicest solution is to search for the key with a relative rank r in a subtree rooted by the internal node w. Initially, w is T, and the relative rank is r. Upon descending from w to the correct child of w, r is decreased by the leaf counts for the children of w that are to the left of the child that is on the path. The child of w that stores the key being sought is the right-most child where the adjusted rank is positive.

Question 3

Let T be a tree. For each of the problems listed on the next page, you are given a standard DFS traversal of T. But the code is incomplete, and there are blank lines in the code where you can insert additional instructions as needed. For each of the three parts on the next page, insert the necessary instructions so that for each vertex v in T, $v.val$ will be set equal to the value that is requested in the problem. There are only six different instructions you can write in a blank line:

"$T.val \leftarrow 0$;"	"$Temp \leftarrow Temp + 1$;"
"$T.val \leftarrow 1$;"	"$Temp \leftarrow 0$;"
"$T.val \leftarrow Temp$;"	"$Temp \leftarrow Temp - 1$;"

You will need just two or three instructions for each problem. You might decide to write two instructions in a blank line. That is okay. These problems are elementary, but you do need to think clearly. To avoid mistakes, think abstractly. As you read each question, think about this. As I walk down the tree, what do I need to do at first entry to a descendant vertex? What do I do just before or just after a last exit from the vertex?

A sample problem and sample solution with the `solution written like this` is shown below.

Sample problem zero points
z) For each v, $v.val$ should be the preorder number of v in T.

global $Temp$;
$Temp \leftarrow 1$;
procedure DFS(T);

foreach child v of T **do**

DFS(v);

endfor;

end_procedure;

Sample solution
z) For each v, $v.val$ should be the preorder number of v in T.

global $Temp$;
$Temp \leftarrow 1$;
procedure DFS(T);
`T.val←Temp;`
`Temp←Temp+1;`

foreach child v of T **do**

DFS(v);

endfor;

end_procedure;

Question 3 – PLEASE WRITE YOUR ANSWERS ON THIS PAGE

Available Commands

exam number _____

"$T.val \leftarrow 0$;"	"$Temp \leftarrow Temp + 1$;"
"$T.val \leftarrow 1$;"	"$Temp \leftarrow 0$;"
"$T.val \leftarrow Temp$;"	"$Temp \leftarrow Temp - 1$;"

Insert the two or three commands needed to solve each problem as specified.

1. [3 points] For each v, $v.val$ should be the postorder number of v in T.

global $Temp$;
$Temp \leftarrow 1$;
procedure DFS(T);

  foreach child v of T **do**

    DFS(v);

  endfor;
  $\boxed{T.val \leftarrow Temp;}$
  $\boxed{Temp \leftarrow Temp + 1}$
end_procedure;

2. [3 points] For each v, $v.val$ should be the depth of v in T as measured in edges, where the root has depth 1, and its children have depth 2, etc.

global $Temp$;
$Temp \leftarrow 1$;

procedure DFS(T);
  $\boxed{T.val \leftarrow Temp;}$
  $\boxed{Temp \leftarrow Temp + 1;}$
  foreach child v of T **do**

    DFS(v);

  endfor;

  $\boxed{Temp \leftarrow Temp - 1;}$

end_procedure;

3. [4 points] For each v, $v.val$ should be zero if v is a leaf and one if v is not a leaf. No $Temp$ is needed. Reminder: $\begin{cases} leaves \leftarrow 0; \\ others \leftarrow 1. \end{cases}$

procedure DFS(T);

  $\boxed{Temp \leftarrow 0;}$

  if T has children **then**

    foreach child v of T **do**

      DFS(v);
      $\boxed{Temp \leftarrow 1}$
    endfor
  endif;

  $\boxed{T.val \leftarrow Temp}$
end_procedure;