Introduction to algorithms and data structures

The IT University of Copenhagen

January 11, 2006, 9.00-13.00

This examination assignment consists of 4 problems with a total of 12 questions. Questions are given equal weight in the grading. You have 4 hours to answer all 12 questions. Remember to write the page number, your name and your CPR.-number on each page of your written answer. The complete assignment consists of 12 numbered pages.

CLRS refers to "Introduction to Algorithms" by Cormen, Leiserson, Rivest and Stein, Second Edition, 2001.

For problems that ask for effecient algorithms the asymptotic complexity of the specified solution will be taken into account when grading. Problems asking for time complexity must be answered using O-notation with least possible asymptotic growth.

1 Searching and Shortest Paths

This problem considers choosing the most efficient algorithm or data structure in a specific setting. The first two questions deal with set data structures with predecessor queries. The last question considers shortest path algorithms.

In the course you have seen different ways to store a set of elements. Here we consider two different implementations of the same abstract data structure. We denote the two DS1 and DS2. We are interested in the abstract data structure that supports three different operations, insert, look-up, and predecessor. INSERT(x) inserts element x into the data structure, LOOK-UP(x) searches for element x in the data structure and returns a pointer to the element or NIL if it is not in the data structure, and PRED(x) returns a pointer to the largest element in the data structure, that is smaller than x, or NIL if no such element is in the data structure. The following table shows the times for the operations for DS1 and DS2. For DS2 the times for insert and look-up are expected. One can see that DS2 does not support predecessor queries that well, but on the other hand it is faster for the other two operations. Which one to choose will depend on how many operations of each type the application performs.

	INSERT(x)	LOOK-UP(x)	PRED(x)
DS1	$O(\log n)$	$O(\log n)$	$O(\log n)$
DS2	$O(1)$ expected	$O(1)$ expected	$O(n)$

Question 1.1 Which two data structures, dealt with in the course, support operations in the above time bounds? Specify which one corresponds to DS1 and to DS2, respectively, and describe a situation where DS2 is expected to be asymptotically more efficient than DS1, when considering the total time for a sequence of operations.

Answer
A balanced search tree can do all operations in logarithmic time, so DS1 is e.g. an AVL tree.
A hash table have constant expected time insertions and look-up, but it does not support predecessor queries. All data in the hash table has to be checked to answer a predecessor query, which takes $O(n)$ time, so DS2 is a hash table.
When few or no predecessor queries are performed on the data structure DS2 is more efficient than DS1.

Note that for the following question, the answer to question 1.1 is not required.

Consider the following combined data structure. Instead of choosing between DS1 and DS2, the combined data structure keeps all data in two data structures, one of type DS1 and one of type DS2. The space usage will increase, but it may

give some advantages as well.

Question 1.2 What are the time bounds for INSERT(x), LOOK-UP(x), and PRED(x), respectively, in the combined data structure, as described above? Is this combined data structure more efficient in any situation? If yes, describe such a situation. If no, argue why not.

Answer
Insert has to be done in both DS1 and DS2, which gives the time $O(\log n)$. Look-up and predecessor can be done in the data structure which is most efficient for the operation, i.e. DS2 for look-up and DS1 for predecessor, hence the time for look-up is $O(1)$ and the time for predecessor is $O(\log n)$.
The combined data structure is more efficient for example when there are few insertions and many look-up and predecessor queries.

The last question considers choosing between Dijsktra's algorithm and the Bellman-Ford algorithm for computing the shortest path in a weighted graph. We denote by V the number of vertices in the graph and by E the number of edges in the graph. We assume that the graph has no edges of negative weight, hence both algorithms can be used. For the general case Dijkstra's algorithm is the natural choice, since it has a lower running time, however the Bellman-Ford algorithm can be modified to be faster than $O(VE)$ when all shortest paths have few edges. We denote by k the maximum number of edges in a shortest path from s, the starting vertex, to any other vertex in the graph. (Note that this question is unrelated to questions 1.1 and 1.2.)

Question 1.3 Describe a modification of the Bellman-Ford algorithm, such that it is more efficient when all shortest paths have few edges. Give the time complexity in terms of V, E and k. For which k is your algorithm more efficient than Dijkstra's algorithm?

Answer In the original form in CLRS Bellman-Ford loops V times, and ends with a test of whether any relaxation improves a d-value. This test can be done in each iteration of the loop. When no d-value improved during an iteration of the loop, all shortest paths have been found. According to the Path Relaxation Lemma, if for all paths, the edges have been relaxed in the same order as they appear in the path, the shortest paths have been found. Since all shortest paths have at most k edges, k iterations of the loop is enough to find all shortest paths. In the $(k+1)$th iteration no d-value will improve and the algorithm ends. The total time is therefor $O(kE+V)$. This is better than Dijkstra's algorithm when $k < (V \log V)/E$.

2 Around Sorting

Question 2.1 Remember the RADIX-SORT algorithm of CLRS p. 172. Which $n \log n$ comparison sorting algorithm (out of those presented in the course) would be suitable as a basic working algorithm of RADIX-SORT, so that the sorting algorithm obtained is correct? Why? What would be the running time of such algorithm? Express your answer in terms of n—the number of integers to sort, d—the number of digits in each of the integers, and k—the number of values that each digit can take.

Answer Any stable sorting algorithm would do, in particular MERGE-SORT. The obtained algorithm would be $O(dn \log n)$.

Question 2.2 Propose a fast and stable sorting algorithm for sorting a sequence of rational numbers of the form $\frac{p}{10^q}$, where $1 \leq p \leq 9$ and $0 \leq q \leq 10$. Both p and q range over integers. The numbers are represented as an array of pairs (p, q). Describe your solution in English or in pseudocode. Then state whether your algorithm is still correct if we relax the assumption about p to $0 \leq p \leq 11$. Motivate your answer rigorously.
Hint: Your algorithm does not need to work correctly under the relaxed condition about p in order to qualify for the full points. It is required though that you can judge whether it will work or not, and why.

Answer Treat rationals as pairs of numbers and use a variant of RADIX-SORT to sort them. The algorithm should sort the numbers in increasing order of p values, and then sort the result by q values in decreasing order using a stable sort.
This algorithm will not work correctly if p can go up to 11, or down to 0. For example the following sequence of numbers: $\frac{11}{10^2}, \frac{9}{10^2}, \frac{1}{10^1}$ would be sorted to $\frac{9}{10^2}, \frac{11}{10^2}, \frac{1}{10^1}$, which is incorrect.

Remember the MERGE procedure that is described in CLRS p. 29. Briefly speaking a call to MERGE(A, p, q, r) merges two sorted subarrays $A[p \,..\, q]$ and $A[q + 1 \,..\, r]$ into a sorted subarray $A[p \,..\, r]$, using linear time. We shall use this procedure in the following sorting algorithm:

Sort(A : integer array, p : integer, r : integer)
if $p \geq r$
then return
$m \leftarrow \lfloor \frac{r-p-1}{3} \rfloor$
Sort($A, p, p+m$)
Sort($A, p+m+1, r-m-1$)
Sort($A, r-m, r$)
Merge($A, p, p+m, r-m-1$)
Merge($A, p, r-m-1, r$)

Question 2.3 Formulate a recurrence describing the asymptotic running time of Sort and solve it using the master theorem (CLRS p. 73). You can assume that n is a power of 3.

Answer The following recurrence relation describes the running time of Sort for inputs of size being a power of 3:

$$\begin{cases} T(1) = O(1) \\ T(n) = 3 * T(\frac{n}{3}) + O(n) \end{cases}$$

From the second case of the master theorem T is $O(n \log n)$.

3 Search Trees

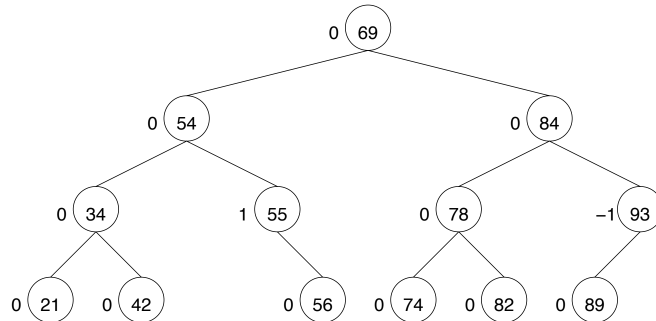
This problem considers balanced search trees, in particular AVL trees and a variant of balanced search trees called *finger search trees*, described below. It also considers the use of finger search trees for sorting.

Question 3.1 Draw an AVL tree for the following 13 elements, include the balance for each vertex.

$$21, 34, 42, 54, 55, 56, 69, 74, 78, 82, 84, 89, 93$$

Answer

The following shows one possible AVL-tree on the 13 elements, there are several other correct answers.



A *finger search tree* is a binary search tree. It has logarithmic depth and searching for an *arbitrary* key takes time $O(\log n)$ in a tree holding n keys. It has an additional feature, that makes some operations considerably faster:

Let S be the set of the keys stored in the finger search tree. Let k_1 be the key involved in the last operation (insert, delete, or find) on the tree. The time to search for, insert or delete a key k_2 depends on the distance between k_1 and k_2 in the sorted order of the keys in S. Let d denote the distance between k_1 and k_2. d is defined as the number of keys in S between k_1 and k_2, or more precisely the number of keys $k \in S$ such that $\min(k_1, k_2) \leq k \leq \max(k_1, k_2)$. The time to search for, insert, or delete k_2 is $O(\log d)$. Note that d is at most n so no operation takes longer than in e.g. an AVL tree, but it can be much faster if k_1 and k_2 are close in the sorted order.

The name, finger search tree, comes from the metaphor finger for a pointer to a place in the tree visited before. A new operation starts from this finger, instead of from the root of the tree. In this problem you only need to know what the complexities of the operations are, and that a finger search tree is a search tree augmented with some extra information in the nodes.

Question 3.2 Given a large finger search tree containing n elements. You have a set of concrete operations (i.e. insertions, deletions, and find operations with known keys) you want to perform on the finger search tree. They can be performed in any order. The total number of operations is m, where m is much smaller than n. In what order should you choose to perform the operations in order to minimize the total time spent.

Answer
By sorting the operations on the keys and perform them in that order, the total time will be minimized, since elements with keys close to each other will be performed after each other, hence minimizing the time it takes.
Note that the above is not the standard way of describing finger search trees, it is a simplified version for this exam.

Sorting an almost sorted sequence can be made more efficient than sorting in general. The sequence 2, 5, 7, 13, 3, 15, 18, 21, 32, 24, 29, 31 is almost sorted, by only moving two elements, 3 and 32, the entire sequence get sorted. In this question you are asked to come up with an algorithm that is fast when it is possible to sort the sequence by moving few elements.

Question 3.3 You want to sort a sequence of n numbers. You know that the sequence is almost sorted, in the sense that it is enough to move k elements to make the sequence sorted. Use a finger search tree to sort the sequence faster than $O(n \log n)$, when k is sufficiently small. State and argue for the time complexity of your algorithm. (*Hint:* It can be done in $O(n + k \log n)$ time.)

Answer

Insert the elements in the input sequence into the finger search tree, one by one in the order they appear in the input sequence. Since a finger search tree is a search tree, a traversal of the tree will return the sorted sequence.

Denote the input sequence $i_1, i_2, \ldots, i_n$. The time to insert element i_j depends on the distance between i_j and element inserted just before it, i.e. i_{j-1}. If i_{j-1} and i_j are consecutive in the output, then the time to insert i_j is $O(1)$, according to the description of the finger search tree. The k misplaced numbers in the input sequence will cause some elements to take longer to insert. It takes time $O(\log n)$ to insert an arbitrary number. The number of elements in the output that do not precced the same number in the input is at most $3k$: Let i_x be a misplaced number, then i_x do not preceed the same number in the input, and the next number i_{x+1} also do not, and the number after i_x in the output will also have another preceeding number in the input. In total, each misplaced number give rise to 3 insertions that may take $O(\log n)$, in total $O(k \log n)$ time. All other numbers take time $O(1)$ to insert. The total time is $O(n + k \log n)$. (This is $O(n)$ if at most $O(n/\log n)$ numbers are misplaced in the input sequence, and it is never worse than $O(n \log n)$.)

4 The House of Bernadotte

Figure 1 presents a *genealogical tree*. The nodes in the tree represent descendants of king Gustaf VI of Sweden. Edges represent parent/child relationships. The nodes deeper in the tree are younger: Ingrid is a daughter of Gustaf VI, and Victoria is a daughter of Carl XVI.

The genealogical tree is an *ordered tree*. Each node x has an attribute $children(x)$, which is an array of children. This array is sorted in the decreasing order of age, so that $children(x)[1]$ is the oldest child of its parent $parent(x)$ and $children(x)[length(children(x))]$ is the youngest child. We assume that all children in the array have distinct ages. In our example Carl XVI is the youngest son of Gustaf Adolf, while Victoria is the oldest child of Carl XVI.

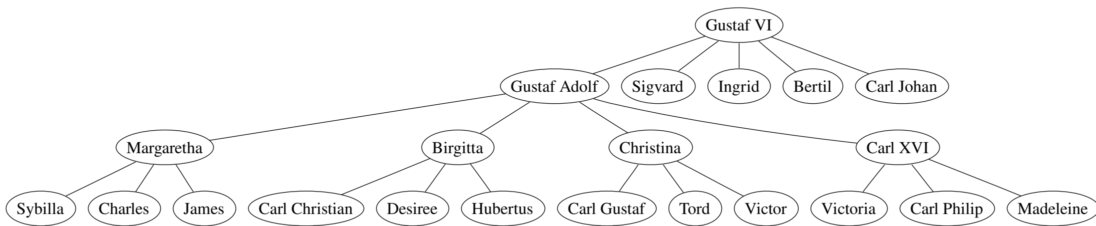


Figure 1: An example of a genealogical tree.

Observe that the genealogical tree may happen to be neither binary nor complete. We write $parent(x)$ in order to access the unique parent node of x. If x is the root, then $parent(x)$ returns NIL. Given a node x, $next(x)$ denotes its oldest sibling younger than x, or NIL, if x is the youngest child in its family. The kth *generation* of the tree is the set of all its nodes of depth k.

The chamberlain of the Royal House is planning the 200th anniversary of the house of Bernadotte on the Swedish throne. Each generation should be commemorated with a number of contemporary art items. In order to plan the exhibition he needs precise information on gender distribution across generations: the number of male members and the number of female members in each generation.

Question 4.1 Design a fast algorithm solving the GENDER-DISTRIBUTION problem: given a genealogical tree rooted in T return two arrays M and F, such that for each generation k, $M[k]$ contains the number of male members of the generation, and $F[k]$ contains the number of its female members. Check the gender of the node x using the *gender* attribute, evaluating to M (male) or F (female). Write your algorithm in pseudocode. What is the time and space complexity of your algorithm? Explain your answers.

Answer global arrays F, M

```
GENDER-DISTRIBUTION(T : node)
    d ← DEPTH(T)
    F ← new array of d cells
    M ← new array of d cells
    for i = 1 to d
          do F[i] ← 0
             M[i] ← 0
    COUNT(T, 1)
    return F, M


DEPTH(x : node)
    d ← 0
    if x ≠ NIL
       then for i = 1 to length(children(x))
                d ← max(DEPTH(children(x)[i]), d)
    return d + 1


COUNT(x : node, d : integer)
    if gender(x) = F
       then F[d] ← F[d] + 1
       else  M[d] ← M[d] + 1
    for i = 1 to length(children(x))
          do COUNT(children(x)[i], d + 1)
```

This algorithm uses space linear in the depth of the tree (for arrays F, M and for the recursion stack). The algorithm is linear in the number of nodes ($O(n)$), as it visits each node exactly twice during execution. The first time during calculating depth and the second time during counting.

The chamberlain would like to pay you for your program. However, since he is spending public money, he would like to judge how good is your solution first.

Question 4.2 Give an asymptotic lower bound for the running time of any algorithm solving the GENDER-DISTRIBUTION problem. Motivate your reply.

Answer In order to count the number of female and male in each generation any algorithm solving GENDER-DISTRIBUTION problem needs to visit each node of the tree at least once. So the lower bound for the problem is $\Omega(n)$. Above we have shown an $O(n)$ algorithm for the problem, so we know that this bound is tight.

In 1980 the Kingdom of Sweden has adopted a full and equal *primogeniture*: the oldest child of the current monarch inherits the throne, regardless of gender. In simplified terms the queue of heirs of x is constructed as follows. If x is alive then x is the first in the queue. Then the queue of heirs of the oldest child of x is appended to the end, after which the queue of heirs of the second oldest child, until all descendants of x are considered. After that the same procedure is applied to the younger siblings of x, then to younger siblings of parent of x, until all possible heirs are placed in the queue.

According to the the new rule the most likely successor to the Swedish throne is now princess Victoria, and not prince Carl Philip (see Fig. 1), who follows her in the queue. Should that bill be passed earlier, it would have a tremendous effect on the history of Sweden. Carl XVI, the present king of Sweden, most likely would have never become a king. Such speculations are of great interest for journalists and novel writers, and they would certainly be happy to have a web service quickly computing successors for various scenarios.

Question 4.3 Design an algorithm HEIR-TRAVERSAL(x) that for a given node x of a genealogical tree prints the queue of heirs of x. You can test whether a node y is dead by using the *died*(y) predicate. Give the worst case asymptotic running time for your algorithm and argue for it.

Answer

This a recursive solution. It is certainly possible to write a similar iterative one.

```
HEIR-TRAVERSAL(x : node)
    if x ≠ NIL
        then VISIT(x, parent(x))


VISIT(x : node, p :node)
    if x = NIL
        then if parent(x) = NIL
                 then return
                 else  VISIT(p, parent(p))
        else  if not dead(x)
                 then PRINT name of x
              VISIT-BELOW(x)
              VISIT(next(x), p)
    return


VISIT-BELOW(x : node)
    if x has no children
        then return
    Let y be the oldest child of x
    while y ≠ NIL
          do if not dead(y)
                 then PRINT name of y
             y ← next(y) return
```

This algorithm visits each node in the tree at most once. In the worst case for the algorithm is when it will have to traverse the entire tree (if the tree is a list), so it is linear in the number of nodes in the tree.