Introduction to Algorithms and Data Structures

IT University of Copenhagen

June 11, 2004

This exam consists of 4 problems containing 13 subproblems. Each problem is marked with the weight in percent, it is given in the evaluation. You have 4 hours to complete the exam. Remember to number the pages and write your name and CPR number on every page. The exam consists of 6 numbered pages.

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

For problems in which algorithms must be specified, the asymptotic time complexity of the specified solution will be taken into account when grading. Problems asking for time complexity must be answered using O-notation. It is weighted in the evaluation that growth rates in the O-notation are expressed with least possible asymptotic growth.

Problem 1 (35%)

Let $A[1..n]$ be an array of integers. For $1 \le k \le n$, the procedure MININDEX(A, k) returns the index of a minimum element of the subarray $A[k..n]$.

```
MinIndex(A, k)
1   m ← k
2   for i ← k + 1 to n
3       do if A[i] < A[m]
4             then m ← i
5   return m
```

a) How many times is line 3 of MININDEX executed as a function of n and k?

The procedure MINSORT uses MININDEX as a subroutine for sorting an array of integers in non-decreasing order.

```
MinSort(A)
1   for i ← 1 to n − 1
2       do exchange A[i] ↔ A[MinIndex(A, i)]
```

b) Consider a single call to procedure MINSORT. What is the total number of times line 3 of MININDEX is executed as a function of n?

c) What is the asymptotic time complexity of MINSORT?

Consider the following loop invariant of the **for** loop of procedure MININDEX:

> *at the start of each iteration of the for loop of lines 2-4, $A[m]$ is a minimum element of the subarray $A[k..i-1]$.*

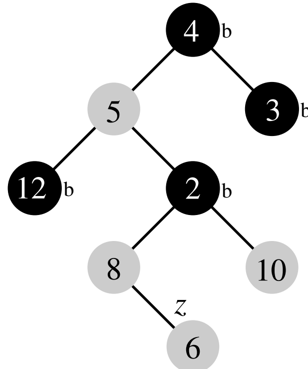
d) Use the loop invariant defined above to carefully prove correctness of procedure MININDEX.

Assume that an integer array A has been sorted in non-decreasing order using MINSORT or some other sorting procedure.

e) Write a procedure FIND(A, a, b, e) that returns *true* if $A[i] = e$ for some $1 \le a \le i \le b \le n$ and otherwise *false*. The asymptotic time complexity of FIND(A, a, b, e) should be $O(\lg n)$.

Problem 2 (20%)

Consider the invalid Red-Black Tree T_{RB} shown below.



a) Without changing the structure of T_{RB}, rearrange the numbers such that it satisfies the binary-search-tree property. Name the resulting tree T'_{RB}.

b) Which of the properties 1-5 on page 273 of CLRS are not satisfied by T'_{RB}?

c) Illustrate the operations of RB-INSERT-FIXUP(T'_{RB}, z) using the approach shown in Figure 13.4 on page 282 of CLRS. Make sure to clearly mark black nodes (e.g., with a small 'b' as shown for T_{RB}). Indicate for each rearrangement of the nodes, whether it was caused by a right or left rotation.

Problem 3 (20%)

Let $G = (V, E)$ be a directed graph with a source vertex s, a goal vertex g, and a weight function $w : E \to \mathbf{R}^+$ where all edge weights are positive. That is, $w(p, q) > 0$ for each edge $(p, q) \in E$.

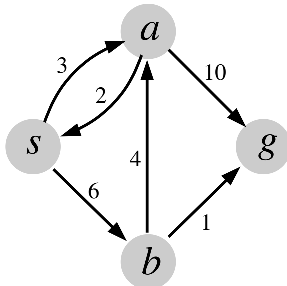
The procedure SHORTESTPATH(G, w, s, g) can be used to find a shortest path from s to g.

```
SHORTESTPATH(G, w, s, g)
1   Q ← ∅
2   INSERT(Q,MKNODE(s, 0, NIL))
3   while Q ≠ ∅
4       do x ← EXTRACT-MIN(Q)
5           if v[x] = g
6               then return x
7               else for each vertex q ∈ Adj[v[x]]
8                           do INSERT(Q,MKNODE(q, d[x] + w(v[x], q), x))
9   return NIL
```

The procedure MKNODE(v, d, π) returns a data object x called a *search node*. It has three attributes: a vertex $v[x] = v$, a cost $d[x] = d$, and a predecessor node $\pi[x] = \pi$. Q is a min-priority queue of search nodes. The key of Q is the cost of the search nodes.

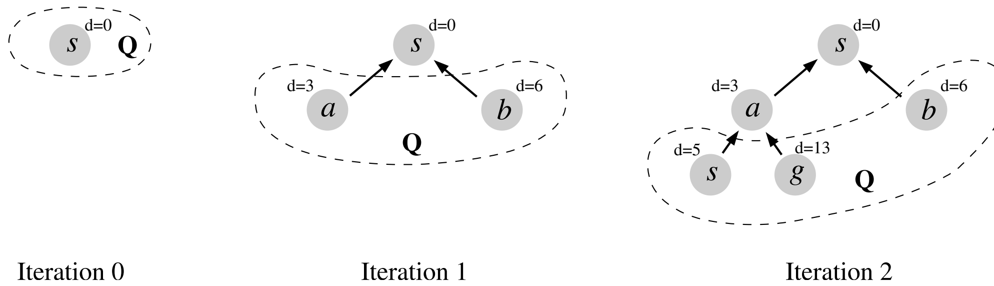
If the goal vertex g is reachable from s, SHORTESTPATH(G, w, s, g) terminates and returns a search node x where $v[x] = g$ and $d[x]$ equals the weight of a shortest path from s to g $(d[x] = \delta(s, g))$. A shortest path from s to g can be extracted by traversing the predecessor nodes of x.

Consider the following graph G_0. The weights are as shown on the edges.



During execution, SHORTESTPATH builds a *search tree* rooted at a search node associated with vertex s. After each iteration of the while loop, the leaves of this search tree are the nodes stored in Q.

Given G_0 as input, the structure of the search tree of ShortestPath after iteration 0, 1, and 2 of the while loop is as follows.



a) In the same style as above, draw the search tree after each of the remaining iterations of the while loop. Point out the node returned by ShortestPath on the last search tree and mark the search nodes corresponding to the shortest path from s to g found by ShortestPath.

The introduction of 0-weight edges may cause ShortestPath to loop forever even though the goal vertex is reachable from the source vertex.

b) Give an example showing this.

It can be shown that it is unnecessary to insert a node x in Q if a node y associated with the same vertex ($v[x] = v[y]$) previously has been inserted in Q and the cost of x is greater than or equal to the cost of y ($d[x] \geq d[y]$).

c) Consider a version of the ShortestPath procedure employing this pruning rule. Draw the search tree after each iteration of this procedure given G_0 as input. Point out the node returned by the procedure on the last search tree and mark the search nodes corresponding to the shortest path from s to g found by the procedure.

Problem 4 (25%)

The computer game *You Are Never Alone* is played on an undirected graph G and involves a user-controlled player and a variable number of evildoers in pursuit of the player. The evildoers currently in play are represented by objects stored in a doubly-linked list.

At each moment in the play, each evildoer e finds him/herself at some vertex v of G, so that position is stored as one of the fields of the object e. During the play, the evildoers change their position within G in such a way that if two evildoers have the misfortune of sharing the same vertex, they immediately mutually annihilate and are removed from the playfield (which implies that no vertex of G ever houses more than one evildoer). Furthermore, there may be bombs situated at vertices of G (perhaps left by the player to inconvenience his pursuers). If a bomb at vertex v goes off, then all evildoers who at that time find themselves at v or any vertex adjacent to v are taken out of the play as well. (We assume that the player never engages in suicide bombing.)

Among the challenges involved in programming this game, one obviously faces those of implementing the following operations:

$\textsc{Move}(e, v)$ — the evildoer e changes his/her position to vertex v. Observe that this may necessitate the removal of e as well as of another evildoer;

$\textsc{Explode}(v)$ — a bomb explodes at vertex v. According to the rules above, this may entail the removal of a number of evildoers.

Recall that by $\deg v$ one denotes the number of vertices adjacent to v in a graph.

a) Design a data structure (involving G and the list of evildoers) together with implementations of the two operations so that $\textsc{Move}(e, v)$ takes $O(1)$, and $\textsc{Explode}(v)$ takes $O(\deg v)$ worst-case time.

b) Suppose that explosions at v affect not just the evildoers, but that the vertex v itself together with all the edges incident on it disappear from the game. Adjust your data structure and algorithms so that $\textsc{Move}(e, v)$ and the new version of $\textsc{Explode}(v)$ still only take $O(1)$ and $O(\deg v)$ time respectively.