

Data Structures and Algorithms

Exam

Prof. Dr. M. Böhlen

February 16, 2007

14:00 - 16:00

The exam consists of 4 exercises. For each exercise you can get 20 points. It is important that you argue for your answers and that you present your solutions in a readable form. As auxiliary material you may use 1 A4 sheet with notes.

1 Recurrences

Calculate the tight upper bounds for the following recurrences.

1. $T(n) = 2T(\frac{n}{2}) + n$
2. $T(n) = T(n - 1) + n^2$
3. $T(n) = 4T(\frac{n}{3}) + 2n$
4. $T(n) = 4T(\frac{n}{3}) + n^2$
5. $T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + T(\frac{n}{8}) + n$

2 Persistent Binary Trees

We sometimes find that we need to maintain past versions of a binary tree as it is updated. Such a binary tree is called persistent. One way to implement a persistent binary tree is to copy the entire tree whenever it is modified. This approach is expensive in terms of time and space complexity. A better approach is to maintain a separate root for every version of the tree and copy only the nodes that have to be modified in order to reconstruct the current and previous versions.

Consider a persistent binary search tree with a node structure defined as follows.

```
struct Tree {  
    int key;  
    struct Tree * lChild;  
    struct Tree * rChild;  
};
```

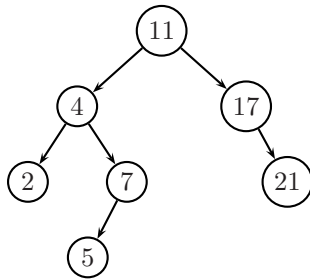


Figure 1: Binary search tree

1. For the persistent binary search tree in Figure 1 draw the complete persistent binary search tree after the persistent insertion of element “8”.
2. Consider the persistent binary search tree that resulted from the above insertion. Draw the complete persistent binary search tree after the persistent deletion of element “11”.
3. Write a function `struct Tree* persistentInsert(struct Tree * T, int k)` that, given a persistent tree T and a key k to insert, returns the root of a new persistent tree T' that is the result of inserting k into T .
4. What is the time complexity of persistent insertion? Explain your answer.
5. What is the time complexity of persistent deletion? Explain your answer.

3 Directed Acyclic Graphs

The packages that are installed on a Linux system form a directed acyclic graph, where $A \rightarrow B$ means that package A depends on package B. When package B is removed, all dependent package should also be removed.

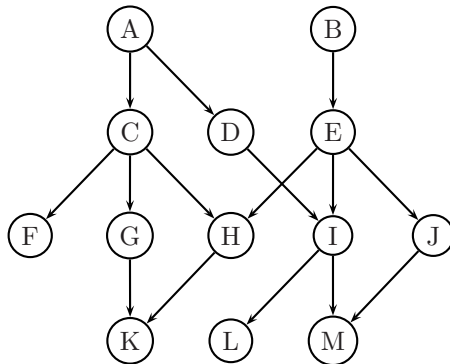


Figure 2: Package dependency graph

1. List all packages that are deleted when package G is deleted.
2. Write an algorithm for package deletion.
3. Estimate the asymptotic complexity of your algorithm.

4 Dynamic Programming

Consider a checkerboard with $n \times n$ squares and a cost-function $c(i, j)$ that returns a cost associated with square i, j (i being the row, j being the column). For instance, on the following 5×5 checkerboard $c(1, 3) = 5$.

	.-----.										
5		5		7		4		8		2	
	-----+-----+-----+-----										
4		7		6		1		1		4	
	-----+-----+-----+-----										
3		3		5		7		9		2	
	-----+-----+-----+-----										
2		1		6		8		0		2	
	-----+-----+-----+-----										
1		7		3		5		6		1	
	,-----,										
		1		2		3		4		5	

Figure 3: Checkerboard

Assume a checker can start on any square on the first row and you want to know the shortest path (sum of the costs of the visited squares are minimal) to get to the last row, assuming the checker can move only forward or diagonally left or right forward. Thus, a checker on $(1, 3)$ can move to $(2, 2)$, $(2, 3)$ or $(2, 4)$.

1. Show the cheapest path for the checkerboard in Figure 3.
2. Write a recursive algorithm that calculates the cheapest path through a checkerboard.
3. Write a version of the cheapest path algorithm that uses dynamic programming.
4. Compare the complexity of the two algorithms.