Final Exam in Algorithms and Data Structures 1 (1DL210)

Department of Information Technology
Uppsala University
February 30th, 2012
Lecturers: Parosh Aziz Abdulla, Jonathan Cederberg and Jari Stenman
Location: Polacksbacken
Time: $8 - 13$
No books or calculator allowed

Directions:

1. Answer only one problem on each sheet of paper
2. Do not write on the back of the paper
3. Write your name on each sheet of paper
4. **Important** Unless explicitly stated otherwise, justify you answer carefully!! Answers without justification do not give any credits.

Good Luck!

Problem 1 (20 p)

a) List the following functions by increasing asymptotic growth rate. If two functions have the same asymptotic growth rate, state that fact. No justification is needed.

$$\lg n \qquad 1.1^n \qquad n \lg n \qquad n(\lg n)^2 \qquad 3 \lg n \qquad 2^5 \qquad n^{34}$$

b) State whether the following statements are true or false. No explanation is needed.

(i) If $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$, then we have $(f(n))^2 = \Theta((g(n))^2)$

(ii) If $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$, then we have $f(n) = g(n)$

(iii) $2^n + n^2 = \mathcal{O}(3^n)$.

(iv) $2^n + n^2 = \mathcal{O}(2^n)$.

Solution for Problem 1

a)

$$2^5 \qquad \underbrace{\lg n \qquad 3\lg n}_{\text{Same asymptotic behaviour}} \qquad n\lg n \qquad n(\lg n)^2 \qquad n^{34} \qquad 1.1^n$$

By discarding constant coefficients we see that $3\lg n = \Theta(3\lg n) = \Theta(\lg n)$.

b) (i) The statement is true. To see this we will use the definitions of $\mathcal{O}$, Ω and Θ. The assumptions $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$ tell us that there are some constants $0 < n_0^{\mathcal{O}}, c^{\mathcal{O}}, n_0^{\Omega}, c^{\Omega}$ such that

$$\forall n > n_0^{\mathcal{O}}.0 < f(n) \leq c^{\mathcal{O}} g(n)$$
$$\forall n > n_0^{\Omega}.0 < c^{\Omega} g(n) \leq f(n)$$

We need to show that there are constants $0 < n_0, c_0, c_1$ such that

$$\forall n > n_0.0 < c_0(g(n))^2 \leq (f(n))^2 \leq c_1(g(n))^2$$

Since both assumptions hold for n which are greater than both $n_0^{\mathcal{O}}$ and n_0^{Ω} we guess that $n_0 = max(n_0^{\mathcal{O}}, n_0^{\Omega})$ would be a good choice. For $n > n_0$ we now have from the assumptions that

$$\begin{array}{cc} 0 < f(n) & 0 < g(n) \\ f(n) \leq c^{\mathcal{O}} g(n) & c^{\Omega} g(n) \leq f(n) \end{array}$$

Hence we have $(f(n))^2 \leq (c^{\mathcal{O}})^2 (g(n))^2$ and $(c^{\Omega})^2 (g(n))^2 \leq (f(n))^2$. So we see that if we chose $c_0 = (c^{\Omega})^2$ and $c_1 = (c^{\mathcal{O}})^2$, the sought inequality is satisfied.

(ii) The statement is false. To see this consider the functions $f(n) = n$ and $g(n) = 2n$. It is certainly true that $g(n)$ is both an asymptotic upper and lower bound for $f(n)$. But $f(n)$ and $g(n)$ are not the same function.

(iii) The statement is true. By the rule of discarding lower order terms we have $2^n + n^2 = \mathcal{O}(2^n)$. Furthermore we have $2^n = \mathcal{O}(3^n)$ since $2^n < 3^n$ for all $n > 0$.

(iv) The statement is true, as noted above.

Problem 2 (15p) Consider INSERTION-SORT and MERGE-SORT. For each algorithm, what will be the worst case asymptotic upper bound on the running time if you know additionally that

a) the input is already sorted?

b) the input is reversely sorted?

c) the input is a list containing n copies of the same number?

For each case and each sorting algorithm, state your answer and justify it in one sentence.

Solutions for Problem 2 Let n be the number of elements in the array that should be sorted. We know that MERGE-SORT runs in $\Theta(n \lg n)$ regardless of the content of the input array. Hence MERGE-SORT runs in $\mathcal{O}(n \lg n)$ in all three cases. For INSERTION-SORT, we need to reason about what happens during the execution of the sorting algorithm:

a) When the input array is already sorted, INSERTION-SORT performs very well. The outer loop will go through the entire array from left to right. But the inner loop will never execute, since every time an element is compared with a preceding element, the elements will already be in the correct order. Therefore INSERTION-SORT will run in $\mathcal{O}(n)$ in this case.

   Suggested answer:
   INSERTION-SORT: $\mathcal{O}(n)$
   MERGE-SORT: $\mathcal{O}(n \lg n)$
   Because MERGE-SORT always runs in $\mathcal{O}(n \lg n)$, and the body of the inner loop of INSERTION-SORT will never execute.

b) Here INSERTION-SORT encounters a worst case. The outer loop will go through the entire array from left to right. Each encountered element will be less than all previously encountered elements. So the inner loop will have to go through the array element by element all the way back to the first element, in order to find the correct position for the new element. Hence INSERTION-SORT will run in quadratic time.

Suggested answer:
INSERTION-SORT: $\mathcal{O}(n^2)$
MERGE-SORT: $\mathcal{O}(n \lg n)$
Because the inner loop in INSERTION-SORT will always have to go through all of the previously sorted elements.

c) In this case INSERTION-SORT performs very well again. When the loop condition of the inner loop compares two equal elements, it will realize that they are already in the correct order. Therefore the inner loop will never execute. As before, INSERTION-SORT runs in $\mathcal{O}(n)$ time.

Suggested answer:
INSERTION-SORT: $\mathcal{O}(n)$
MERGE-SORT: $\mathcal{O}(n \lg n)$
Because the body of the inner loop of INSERTION-SORT will never execute.

Problem 3 (20p) Suppose you have a set of numbers where you over some time period do a constant number of insertions and a linear number of lookups of the maximum. To maintain your set of numbers, you you can choose between using either a heap or a binary search tree.

a) Assume you are interested in minimizing the average total runtime over the time period. For each structure, what is the asymptotic upper bound on the average case runtime, and which of the two structures should you choose?

b) Suppose you realize that you are actually interested in the worst case. For each structure, what is the asymptotic upper bound on the worst case runtime, and which of the two structures should you choose now?

Each justification should be no longer than two sentences.

Solution for Problem 3 Let k be the number of insertions to be made. Let n be the number of elements in the set. By assumption there will be $\Theta(n)$ lookups of the maximum element.

a) Insertion into a heap with n elements amounts to bubbling a leaf up towards the root in a balanced tree with n elements. This is done in $\mathcal{O}(\lg n)$ time. Insertion into a binary search tree is done by traversing the tree from root to a leaf. The time consumption of this traversal depends on the shape of the tree. In the average case, we assume the tree to be somewhat balanced, so the height of the tree would be $\mathcal{O}(\lg n)$. The time consumption of the traversal is proportional to the height of the tree, and so also $\mathcal{O}(\lg n)$. In order to lookup the maximum element in a heap, we only have to look at the root element. This is done in $\mathcal{O}(1)$ time. To lookup the maximum element in a binary search tree we have to find the rightmost node in the tree. This can be done by traversing the tree from the root, always chosing the path to the right until this is no longer possible. In a somewhat balanced tree, the length of such a traversal is likely to be close to the height of the tree: $\mathcal{O}(\lg n)$. So the time consumption is also $\mathcal{O}(\lg n)$.

Hence, the total time consumption in the average case is $k \cdot \mathcal{O}(\lg n) + \Theta(n)\mathcal{O}(1) = \mathcal{O}(n)$ when we use a heap, and $k \cdot \mathcal{O}(\lg n) + \Theta(n)\mathcal{O}(\lg n) = \mathcal{O}(n \lg n)$ when we use a tree.

Suggested answer:
Heap: $\mathcal{O}(n)$
Tree: $\mathcal{O}(n \lg n)$
For the heap, the time consumption is dominated by a linear number of lookups, each performed in $\mathcal{O}(1)$ time. For the tree, the time consumption is dominated by a linear number of lookups, each performed in $\mathcal{O}(\lg n)$ time.

b) For the heap, lookup is always done in constant time, and insertion is always a traversal from leaf to root in a balanced tree. So the time consumption remains bounded by $\mathcal{O}(n)$ even in the worst case. For the tree, we must consider the degenerate case when no node has a left child. Then the tree is effectively a list rooted in the smallest element, descending rightwards through ever increasing elements towards the greatest element. In the worst case, we have to insert elements that are even greater than the elements already present in the tree. Then we have to traverse the entire tree, going through all $\Theta(n)$ nodes before reaching the correct place to insert the new element. Such an insertion runs in time bounded by $\mathcal{O}(n)$. Lookup of the maximal element in such

a degenerate tree is equally bad. Since the maximal element is kept in the rightmost node, in order to find it we have to traverse all $\Theta(n)$ nodes again. This is done in $\mathcal{O}(n)$ time.

So the total time consumption is $\mathcal{O}(n)$ when we use a heap, and $k \cdot \mathcal{O}(n) + \Theta(n)\mathcal{O}(n) = \mathcal{O}(n^2)$ when we use a tree.

Suggested answer:
Heap: $\mathcal{O}(n)$
Tree: $\mathcal{O}(n^2)$
The heap behaves asymptotically the same in the worst case as in the average case. In a tree where no node has a left child, both insertion and lookup of maximal element is done in $\mathcal{O}(n)$ time.

Problem 4 (10p) We know that hash tables are very common in practice when you have a lot of insertions, deletions and search operations. We have seen that under the assumption of uniform hashing, collision resolution by chaining, and constant time computable hash function, deletion and search are both $\mathcal{O}(1 + \frac{n}{m})$ in the average case.
We have also seen that this can in fact be reduced to $\mathcal{O}(1)$ under one additional assumption. Describe this additional assumption in one sentence.

Solution for Problem 4
Intuitively, deletion and search will run in average case constant time if the average number of elements in each bucket is constant. This is the case precisely when the fraction $\frac{n}{m}$ is bounded by some constant. I.e. when $\frac{n}{m} \leq c$ for some constant c. We recognize that this condition is elegantly expressed as $n = \mathcal{O}(m)$.

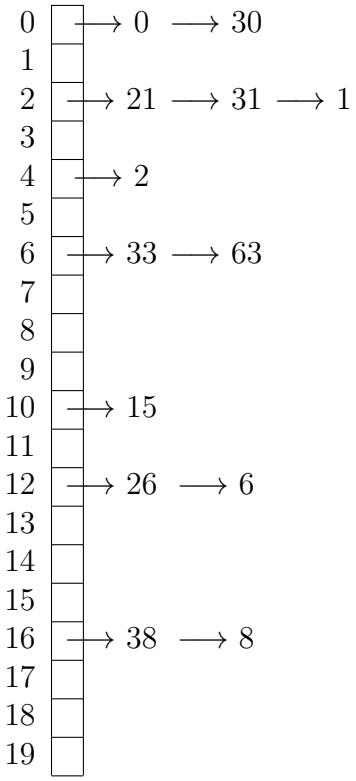
Problem 5 (15p) Suppose you have a hash table and have inserted some elements. When you inspect it you see that the result looks as the picture below. Being a good student, you realize this is a problem.

a) What is the problem here?

b) Give an example of a hash function that could give rise to this behavior.

c) What would be a better hash function?

Answer each question in one sentence.



Solution for Problem 5

a) The problem is that the elements are unevenly distributed over the hash table indices. There are multiple conflicts, yet only about a third of the indices are used.

b) We note that all elements that are hashed to the same index have the same last digit. Furthermore we note that odd indices are never occupied, and that elements with last digit d are hashed to index $2d$. This leads us to suggest the following hash function: $f(n) = 2n \mod 20$.

c) Simply removing the coefficient 2 from the above function will give us a more even distribution: $f'(n) = n \mod 20$.

Problem 6 (10p)

Suppose that we have numbers between 1 and 100 in a binary search tree and want to search for the number 45. Which (possibly multiple) of the following sequences could be the sequence of nodes examined?

- 5, 2, 1, 10, 39, 34, 77, 63.
- 1, 2, 3, 4, 5, 6, 7, 8.
- 9, 8, 63, 0, 4, 3, 2, 1.
- 8, 7, 6, 5, 4, 3, 2, 1.
- 50, 25, 26, 27, 40, 44, 42.
- 50, 25, 26, 27, 40, 44.

Solution for Problem 6

- "5, 2, 1, 10, 39, 34, 77, 63" is not a possible sequence to investigate during a search for 45 in a binary search tree. The first element in the sequence must be the root element. When we compare the root key 5 with the sought key 45, we see that $5 < 45$, and so 45, if present in the tree, must be in the right child tree of the root node. The second element in the sequence (2) should therefore be the root of the right subtree of the root node. But all keys in the right subtree are strictly greater than 5, and so 2 cannot be the key root of the right subtree.
- "1, 2, 3, 4, 5, 6, 7, 8" is a possible sequence to investigate. It is the sequence that is investigated in the degenerate tree consisting of elements $\{1, 2, 3, 4, 5, 6, 7, 8\}$, where no node has a left child.
- "9, 8, 63, 0, 4, 3, 2, 1" is not possible. If we go to 8 from 9, then we are going in the wrong direction to find 45.
- "8, 7, 6, 5, 4, 3, 2, 1" is similarly not possible.

- "50, 25, 26, 27, 40, 44, 42" is not possible. All steps are in the right direction except the last one. From 44 we should check the right child, which has a key greater than 44. Instead we go to 42.
- "50, 25, 26, 27, 40, 44" is a possible sequence.

Problem 7 (10p) Suppose that we first insert an element x into a binary search tree that does not already contain x. Suppose that we then immediately delete x from the tree. Will the new tree be identical to the original one? If *yes* give the reason in no more than 3 sentences. If *no* give a counter-example. Draw pictures if you necessary.

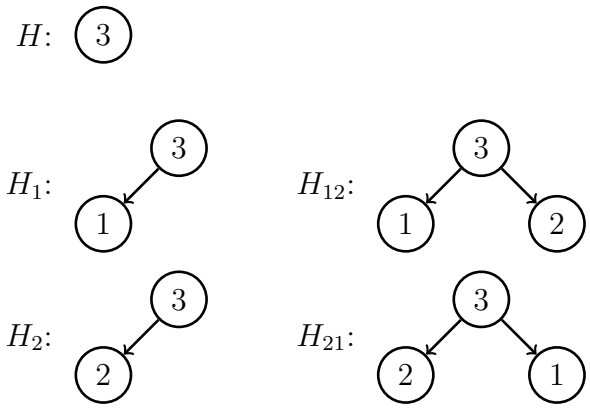
Solutions for Problem 7
In order to reason about this problem, we need to consider the different cases of node deletion in a binary search tree. When we delete a node x from a binary search tree, if x has no children, then we simply remove x. If x has only one child, then we replace the subtree rooted in x by the subtree that is the child of x. If x has two children we extract either the successor or the predecessor of x from its children, and replace x with that element. While the last two cases perform rather extensive changes to the tree, the first case only removes x while leaving the rest of the tree unchanged.
The element x was the last element to be inserted into the tree, and was not in the tree before. Hence the insertion of x into the tree will have changed the tree *only* in that x has appeared as a leaf. A leaf has no children, so removing x from the tree will leave the rest of the tree unchanged. Thus the tree after removing x will be the same as it was before x was inserted.
Suggested answer:
Yes.
x will be inserted as a leaf, with no further changes to the tree. A leaf has no children. When we remove a node without children, the rest of the tree remains unchanged.

Problem 8 (10p) Suppose we have a heap H and two values v_1 and v_2, such that all values are distinct. Let H_{12} be the heap you get if you insert v_1 and

then v_2 into H, and H_{21} be the heap you get if you insert v_2 and then v_1 into H. Give an example of H, v_1 and v_2 such that $H_{12} \neq H_{21}$. No justification needed, just draw the heaps H, H_{12} and H_{21}.

Solution for Problem 8

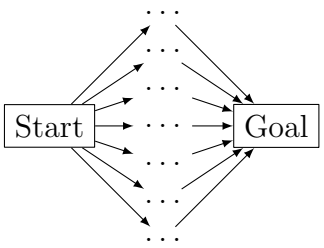
We first note that the heaps H_{12} and H_{21} will contain the same set of elements. So we need to get the elements in different order in H_{12} and H_{21}. When we insert an element into a heap, we first add it at the leftmost vacant space at the highest non-full tree layer. Then we let the element bubble upwards as necessary to reinstate the heap property. So if neither v_1 nor v_2 needs to bubble upwards, then they will remain at the bottom layer of the heap, and in different order in H_{12} and H_{21}. Based on this observation we suggest the heap $H = [3]$ and values $v_1 = 1$ and $v_2 = 2$. Inserting 1 (or 2) into H produces the heap $H_1 = [3, 1]$ (respectively $H_2 = [3, 2]$) where the heap property already holds and no bubbling is necessary. Then inserting 2 (respectively 1) into that heap gives the heap $H_{12} = [3, 1, 2]$ (respectively $H_{21} = [3, 2, 1]$) where again the heap property holds and no bubbling is required. $H_{12} \neq H_{21}$.



Problem 9 (10p)

Suppose you have a graph like the one below. The dots signify some part of the graph that you don't know exactly, *not* a straight link. You know however, that there are *many* paths from the start to the goal. You also know that they tend to be rather long. Suppose that you want to implement a program that searches for a path and returns the first one it can find. You have no need for finding the optimal path in any sense, just any path will do.

Would you want to use BFS or DFS as the basis for you program? Justify your answer in at most two sentences.



Solution for Problem 9

We consider first what will happen when we perform a BFS on the graph. The search will first explore all nodes at distance 1 from Start. Then all nodes at distance 2 from Start, and so forth. When the search reaches Goal, we will have explored all nodes at a distance from Start which is less than the length of the shortest path from Start to Goal. Since we know that there are many paths from Start to Goal, and that the paths are long, we will have explored a large number of nodes unnecessarily.

By contrast, since there are many routes from Start to Goal, DFS is likely to start along some such route before doing too much useless explaration. Once DFS has started along a route, it will follow it to the end, and so the search is likely to find Goal before long.

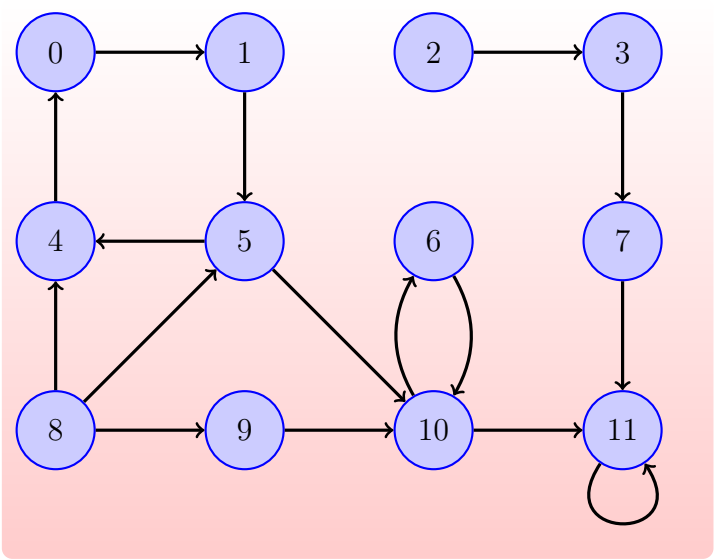
Suggested answer:

DFS

BFS will search all paths from the Start up to the length of the shortest path to the Goal, which is likely to be a large part of the entire search space. DFS

is likely to find one of the many paths to the Goal before doing too much useless exploring.

Problem 10 (10p) Give one possible DFS traversal starting from node 0 of the graph below, printing the nodes both as they are discovered and finished. Note that the graph is **directed**; for instance, there is an edge from 8 to 4 but not from 4 to 8.



Solution of Problem 10

Below, the answer consists of the left part of each line. The text within braces `{}` is only comments, and should not be included in the answer to the exam question. This is one DFS traversal, but not the only one. In the places where we arbitrarily pick a successor, we might as well pick another one.

```
Discover 0  {Go to the only successor 1.}
Discover 1  {Go to the only successor 5.}
Discover 5  {Successors: 4 and 10. Arbitrarily pick 10.}
Discover 10 {Successors: 6 and 11. Arbitrarily pick 6.}
```

```
Discover 6  {Only successor: 10. But 10 is visited already.}
Finish 6    {Backtrack to 10, go to the next successor 11.}
Discover 11 {Only successor: 11. But 11 is visited already.}
Finish 11   {Backtrack to 10}
Finish 10   {Backtrack to 5, go to the next successor 4}
Discover 4  {Only successor: 0. But 0 is visited already.}
Finish 4    {Backtrack to 5}
Finish 5    {Backtrack to 1}
Finish 1    {Backtrack to 0}
Finish 0    {We are done!}
```