Midterm
Introduction to Algorithms
601.433/633

Tuesday, October 22rd, 9am-10:15am

Name:

Ethics Statement

I agree to complete this exam without unauthorized assistance from any person, materials, or device.

Signature: Date:

1 Problem 1: Short Problems (25 points)

For the following, you *do not* need to give proofs or counterexamples. If the problem is multiple-choice, please circle your answer(s).

(a) Let $T(n) = 3T(n-2)$, with $T(1) = T(2) = 1$. Circle whichever of the following are true (there may be more than one).
$\boldsymbol{T(n) = \Theta(3^{n/2})}$ $T(n) = \Theta(2^{n/3})$ $T(n) = \Theta(3^n)$ $T(n) = \Theta(2^n)$

(b) $\log \log n = O(\log^* n)$
true **false**

(c) On *every* input, the expected running time of randomized quickselect is $O(\log n)$
true false

(d) If we changed the median-of-median algorithm (BPFRT) to use groups of size 7 rather than 5, then its running time is still $O(n)$.
true false

(e) The sorting lower bound implies that no sorting algorithm can sort n integers between 1 and n^{10} in $o(n \log n)$ time (in the worst case).
true **false**

(f) If we do the exact same number of Extract-Mins as Inserts, then the total running time of a binary heap and a binomial heap are the same asymptotically (they are $\Theta(\cdot)$ of each other)
true false

(g) If we use trees with union-by-rank and path compression, the amortized cost of both Union and Find is $O(\log^* n)$.
true false

(h) Let H be a universal family of hash functions from U to $[M]$. Then for any sequence of L insert, delete, and lookup operation, the expectation of the total running time (when h is drawn uniformly from H) is $O(L)$.
true false

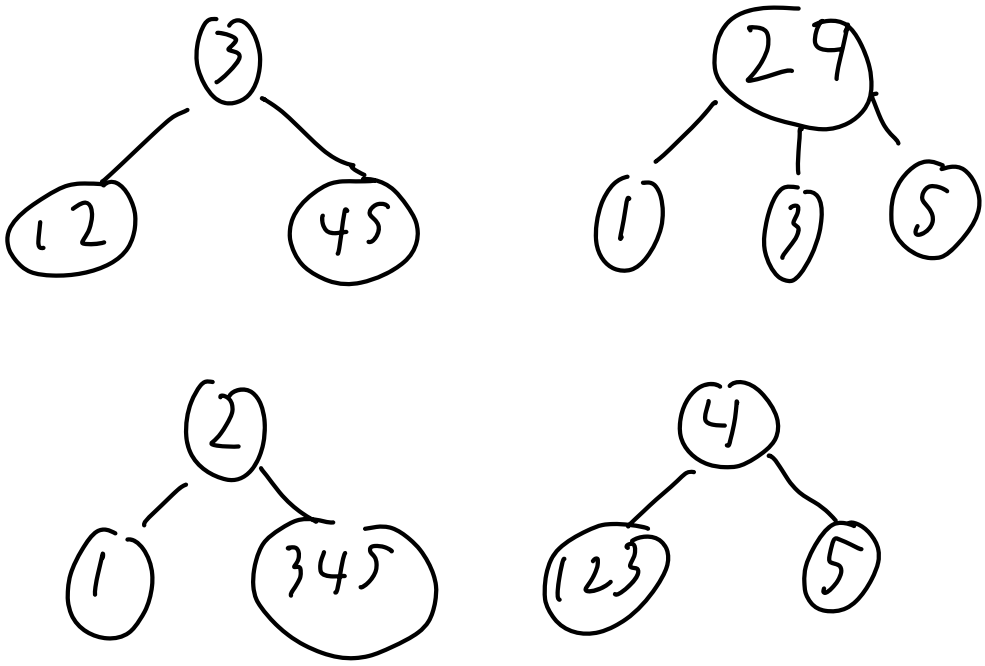
(i) Let H be a universal family of hash functions. Then for every $h \in H$, if we choose two elements $x, y \in U$ uniformly at random, the probability that $h(x) = h(y)$ is at most $1/M$
true **false**

(j) If we build a B-tree with $t = \Theta(\log n)$, then the running time of Insert is $O(\log n)$
true **false**

2 Problem 2 (25 points): Data Structures

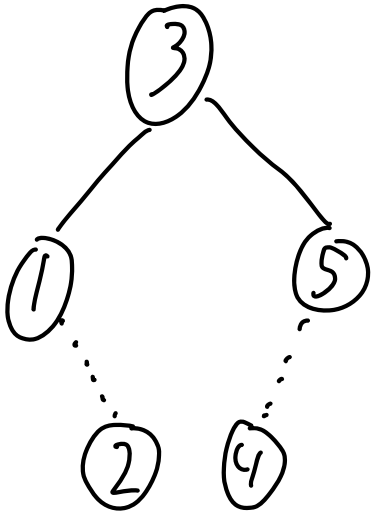
(a) (8 points) Draw all possible 2-3-4 trees on the keys $\{1, 2, 3, 4, 5\}$.

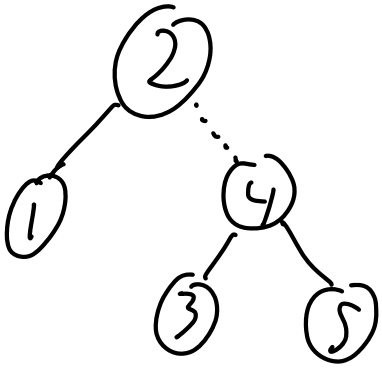
Solution:

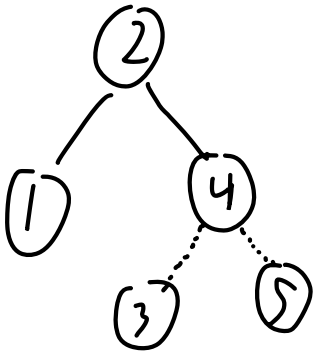


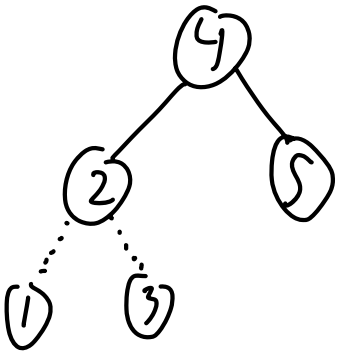
(b) (8 points) For each of the trees in the previous part, draw a corresponding red-black tree (use a dashed line for red and a solid line for black).

Solution:



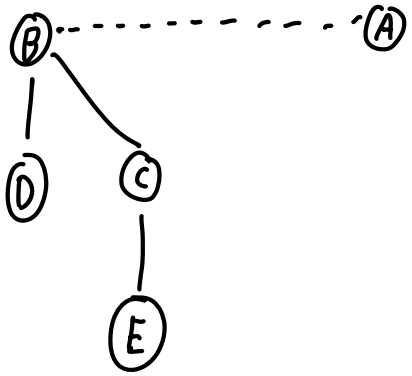






(c) (9 points) How many different binomial heaps are there on the keys $\{1, 2, 3, 4, 5\}$? Justify your answer. Hint: think about the structure of a binomial heap. How many ways are there to fill in that structure legally?

Solution: 15. A binomial heap on $\{1, 2, 3, 4, 5\}$ has the following structure:



There are 5 ways of filling in node A. For each choice, B is fixed (it must be the smallest of the remaining elements). Then there are three ways of filling in D, and for each such choice, C and E are fixed. Thus overall there are $5 \times 3 = 15$ possible binomial heaps.

3 Problem 3 (25 points): Amortized Analysis

Suppose that we want to implement a stack as an array, for example if we want to also have read-access to elements in the middle rather than just being able to pop out the element on top. We saw in class that if we double the size of the array when it is full, then the amortized cost of an insert is still only $O(1)$. In other words, n inserts only take time $O(n)$ even though a single insert might take $\Omega(n)$ time (to create a new array of twice the size and copy over all of the elements).

What if we add in the ability to pop from the stack, though? Pops can be implemented quickly, but we might end up in the undesirable situation of having an array that is much, much bigger than the size of the stack. For example, if we do n pushes and $n-1$ pops, then the size of the array will be $\Theta(n)$ even though there is only one element in the stack!

To fix this, consider the following algorithm. Let D be the current array, and suppose that α positions of the array have a stack element. So if $\alpha = |D|$ then the array is full, and if $\alpha = 0$ then the array is empty. As before, if the array is full then when we push a new element we double its size. Let's say that this takes time exactly $|D| + 1 = \alpha + 1$, since it takes $|D| = \alpha$ time to copy the elements and 1 to push the new element. On the other hand, if after a pop the array is less than $1/3$ full then we *contract* the array by making a new array of $2/3$ the size and copying all of the elements. This also takes time $\alpha + 1$ (to pop a single element and then copy the rest), but note that in this case $\alpha = |D|/3$ rather than $|D|$.

Consider the potential function $\Phi = |2\alpha - |D||$.

(a) (9 points) Prove that on any push, the amortized cost (when using the above potential function) is at most $O(1)$.

Solution: If we do a push and do not expand the array, the amortized cost is $1 + \Delta\Phi \leq 3$. (the change in potential will be 2 if α was initially at least $|D|/2$, and will be -2 is α was initially less than $|D|/2$). If we do a push that does cause the array to expand, then it must be the case that before the push the array was full, so $\alpha = |D|$ and thus $\Phi = \alpha$. Then since the expansion double the array, after the expansion we have that $\alpha = |D|/2$ and thus $\Phi = 0$. So the amortized cost is $\alpha + 1 + \Delta\Phi = 1$.

(b) (9 points) Prove that on any pop, the amortized cost (when using the above potential function) is at most $O(1)$.

Solution: If we do a pop but do not contract the array then the amortized cost will be $1 + \Delta\Phi \leq 3$. If we do a pop that causes the array to contract then initially $\alpha = |D|/3$ and thus $\Phi = \alpha$, and after the contraction $\alpha = |D|/2$ so $\Phi = 0$. Thus the amortized cost is $\alpha + 1 + \Delta\Phi = 1$.

(c) (7 points) Using the previous parts, prove that for every sequence of n operations (pushes and pops) the total running time is $O(n)$. You may assume initially $|D| = 2$ and there is one element in the array.

Solution: We know by definition that the total running time is equal to the total amortized time plus $\Phi_{init} - \Phi_{final}$. Since $\Phi_{init} = 0$ and $\Phi_{final} \geq 0$, this implies that total time of n operations is at most the total amortized time. By the previous two parts, the total amortized time is $O(n)$, and hence the total time is $O(n)$.

4 Problem 4 (25 points): Dynamic Programming

Let $G = (V, E)$ be a directed graph with vertices $v_1, v_2, \ldots, v_n$. Recall that $|E| = m$. We say that G is an *ordered graph* if it has the following two properties:

1. Every edge goes from a vertex with a lower index to a vertex with a higher index. That is, every edge has the form (v_i, v_j) with $i < j$.

2. Every vertex other than v_n has at least one outgoing edge.

The length of a path is the number of edges on it. In this problem you will design an efficient algorithm to find the length of the *longest* path which starts at v_1 and ends at v_n. Note that by the two properties of an ordered graph, every node has at least one path to v_n.

(a) Let $S(i)$ denote the length of the longest path that starts at v_i and ends at v_n. Write a (recursive) formula for $S(i)$, which in the next part will form the basis of a dynamic programming algorithm. You do not need a formal proof of correctness, but should justify your answer.

Solution: First, note that $S(i)$ is well-defined for all i because (by the second property of ordered graphs) every node has at least one path to v_n. Note that the longest path from v_i to v_n must have some first hop using edge (v_i, v_j) where $j > i$. Thus we have the formula

$$S(i) = \begin{cases} 0 & \text{if } i = n, \\ 1 + \max_{j:(v_i,v_j)\in E}\{S(j)\} & \text{otherwise} \end{cases}$$

(b) Design an $O(m)$-time algorithm to find the longest path which starts at v_1 and ends at v_n (using your solution from part (a)). Argue correctness and running time (again, you do not need a fully formal proof, but should provide justification).

Solution: Consider the following algorithm (which is just a bottom-up dynamic programming algorithm for the formula from part (a)), which returns the *length* of the longest path but not the longest path itself.

$M[n] = 0$
for $(i = n - 1$ downto $1)$ {
$M[i] = 1 + \max_{j:(v_i,v_j)\in E} M[j]$
}
return $M[1]$

For correctness, we know by construction that $M[n] = S(n)$, and then a simple induction (together with part (a)) implies that $M[i] = S(i)$ for all i. Hence $M[1] = S(1)$, which by definition is the value we are looking for.

For the running time, let d_i be the outdegree of v_i. Note that the time it takes to calculate $M[i]$ is $O(d_i)$. Hence the total running time is $\sum_{i=1}^{n} O(d_i) = O(m)$.

Once the above algorithm correctly fills out M, we can use the standard backtracking through the dynamic programming table to compute the longest path itself. Slightly more formally (not necessary that students actually write this out), we can use the following algorithm to give the actual path.

$i = 1$
Path $P = (i)$
while $(i \neq n)$ {
iterate through the edges out of v_i until we find the edge (i, j) where $M[i] == M[j] + 1$
Add j to the path.
set $i \leftarrow j$
}

The running time of this is clearly at most $O(m)$, and since the previous algorithm fills out M correctly we know that it will correctly find the longest path.