Data Structures, Spring 2003
PROBLEM SET 1 - School Solution.

Given on Tuesday March 4, 2003. Due: 19:00 Monday March 17, 2003.

NOTE: These solutions are sometimes more detailed than what you are requested to do. For example, we did not expect you to prove that $\theta(\theta(g)) = \theta(g)$ as we did in 1c. We add those details here so that you can check that you fully understand.

1. (The $O, \Omega, \Theta, \circ$ notation)

(a) Let $h(n) = \circ(f(n))$ (little O), $g(n) = \Theta(f(n) + h(n))$. Prove $g(n) = \Theta(f(n))$.

Solution: You may have assumed here that h, f are positive. In this case the proof is:

We first show that $g(n) = O(f(n))$. We know that $g(n) = O(f(n) + h(n))$, which means that there exists a constant C and n_0 such that $g(n) \leq C(f(n) + h(n))$ for $n \geq n_0$. Since $h(n) = \circ(f(n))$, there exists some n_1 such that $h(n) \leq f(n)$ for all $n \geq n_1$. Hence, for $n \geq max\{n_0, n_1\}$, $g(n) \leq C(f(n) + h(n)) \leq C(f(n) + f(n)) = 2Cf(n)$ which satisfies the definition of $g(n) = O(f(n))$.

We now show that $g(n) = \Omega(f(n))$. We know that $g(n) = \Omega(f(n) + h(n))$, which means that there exists a constant C and n_0 such that $g(n) \geq C(f(n) + h(n))$ for $n \geq n_0$. But $g(n) \geq C(f(n) + h(n)) \geq Cf(n)$ if h is positive, so we are done.

The claim is also true for f positive and h negative.

To prove $g(n) = O(f(n))$: We have $g(n) = O(f(n) + h(n))$, which means that there exists a constant C and n_0 such that $g(n) \leq C(f(n) + h(n))$ for $n \geq n_0$, but if h is negative, this implies $g(n) \leq C(f(n) + h(n)) \leq Cf(n)$ for $n \geq n_0$, which is the definition of $g(n) = O(f(n))$.

To prove $g(n) = \Omega(f(n))$: We have $g(n) = \Omega(f(n) + h(n))$, which means that there exists a constant C and n_0 such that $g(n) \geq C(f(n) + h(n))$ for $n \geq n_0$. Since $h(n) = \circ(f(n))$, there exists some n_1 such that $|h(n)| \leq f(n)/2$ for all $n \geq n_1$. Hence, for $n \geq max\{n_0, n_1\}$, $g(n) \geq C(f(n)+h(n)) \leq C(f(n)-f(n)/2) \geq C/2f(n)$ which satisfies the definition of $g(n) = \Omega(f(n))$.

(b) Prove that if $f(n) = h(n)p(n)$ where $p(n) = \Theta(g(n))$ then $f(n) = \Theta(h(n)g(n))$.

Solution: Since $p(n) = \Omega(g(n))$ there exists C, n_0 such that $p(n) \geq C(g(n))$ for $n \geq n_0$. Then $f(n) = h(n)p(n) \geq Ch(n)g(n)$ for $n \geq n_0$, so $f(n) = \Omega(h(n)g(n))$.

Since $p(n) = O(g(n))$, there exists C', n_1 such that $p(n) \leq C'(g(n))$ for $n \geq n_1$. Then $f(n) = h(n)p(n) \leq C'h(n)g(n)$ for $n \geq n_1$ which is the definition for $f(n) = O(h(n)g(n))$.

Together $f(n) = \Omega(h(n)g(n)), f(n) = O(h(n)g(n))$ imply $f(n) = \Theta(h(n)g(n))$.

(c) Prove: If $q > 1$ is a constant, then $\sum_{i=1}^{h(n)} q^i = \theta(q^{h(n)})$. (This completes the part that was missing in class in the proof of case c of the restricted Master theorem.)

Solution: We assume $h(n)$ is positive integer, as is implicitly assumed by the fact it appears in the sum.

Define $f(n) = \sum_{i=1}^{h(n)} q^i$. By the formula for a geometric sum, $f(n) = \sum_{i=1}^{h(n)} q^i = \frac{q^{h(n)+1}-q}{q-1}$.

$\frac{1}{q-1}$ is a constant, so it is $\theta(1)$. By (b), $f(n) = (q^{h(n)+1} - q)\Theta(1) = \Theta(q^{h(n)+1} - q)$.

Now, if you assumed that $h(n)$ grows to infinity, so $q^{h(n)+1}$ grows to infinity as well, since $q > 1$, and so q is negligible compared to $q^{h(n)+1}$, $q = o(q^{h(n)+1})$. Now by (a), we have $q^{h(n)+1} - q = \Theta(q^{h(n)+1})$.

[If you do not assume that $h(n)$ diverges, it is slightly more complicated to show $q^{h(n)+1} - q = \Theta(q^{h(n)+1})$: Since $q \leq \frac{1}{q}q^{h(n)+1}$, we have $q^{h(n)+1} - q \geq q^{h(n)+1} - \frac{1}{q}q^{h(n)+1} = (1 - \frac{1}{q})q^{h(n)+1} = \Theta(q^{h(n)+1})$ where the last equality follows from (b) since $1 - 1/q = \Theta(1)$. $q^{h(n)+1} - q \geq \Theta(q^{h(n)+1})$ implies $q^{h(n)+1} - q = \Omega(q^{h(n)+1})$. but since also $q^{h(n)+1} - q \leq q^{h(n)+1}$ we have $q^{h(n)+1} - q = O(q^{h(n)+1})$ so $q^{h(n)+1} - 1 = \Theta(q^{h(n)+1})$.]

We now want to get rid of the final 1 in the exponent. This is easy: $q^{h(i)+1} = q(q^{h(i)}) = \Theta(q^{h(i)})$ again by (b), since $q = \Theta(1)$.

Finally, we collect everything and we have proved together: $f(n) = \Theta(q^{h(n)+1} - q) = \Theta(\Theta(q^{h(n)+1})) = \Theta(\Theta(\Theta(q^{h(n)})))$. But $\Theta(\Theta(k(n)) = \Theta(k(n))$ for any function $k(n)$ (it follows trivially from the definition) so we get that $\Theta(\Theta(\Theta(q^{h(n)}))) = \Theta(q^{h(n)})$, the desired result.

2. (Recurrences) Give asymptotic upper and lower bounds for the following recurrences, or if possible solve exactly. You can consult the general Master theorem which appears on the slides and in CLR's book. Consider only n's for which the recurrences are well defined, and specify which are those n's.

   (a) $T(n) = 2T(n/4) + \sqrt{n}, \quad T(1) = 3$

   Solution:

   The recurrence is only defined for powers of 4, $n = 4^i$ for $i \geq 0$. We use the master equation, the case $a/b^c = 1$, since $a = 2, b = 4, c = 1/2$ to get $T = \theta(\sqrt{n}logn)$.

   (b) $T(n) = 3T(n/2) + nlog(n), \quad T(1) = 2$

   solution

   This recurrence is defined for power of 2, $n = 2^i, i \geq 1$. We use here the general Master theorem which deals with $T(n) = aT(n/b) + f(n)$. In this case $f(n) = nlog(n)$.

   We claim that $nlog(n) = O(n^{1.5})$. To see this, recall that $log(m) = O(m)$. This implies that for any constant $c > 0$, $log(n) = O(n^c)$, because just substitute $m = n^c$, and get $log(n^c) = clog(n) = O(n^c)$, or $log(n) = 1/cO(n^c) = O(n^c)$ (by 1.a). Now set $c = 1/2$ and we get that $log(n) = O(n^{1/2})$. If we multiply by n we get $nlog(n) = n \cdot O(n^{1/2}) = O(n^{1.5})$.

   We now see that the first case in CLR applies: $f(n) = O(n^{1.5}) = O(n^{log_2(3)-\epsilon})$ for some $\epsilon > 0$, because $log_2(3) > 1.5$.

   Using the Master theorem we have $T(n) = \Theta(n^{log_2(3)})$.

   (c) $T(n) = 4n + 2 + T(n - 5), \quad T(0) = 2.$

   Solution

   In this case the solution is defined only for multiples of 5, $n = 5k$ for $k \geq 0$.

   Here comes a nice trick (you can do without it but it is more elegant to use it): Substitute $n = 5k$. We get $T(5k) = 20k + 2 + T(5(k - 1))$. Now define a new function of k: $T'(k) = T(5k)$. We have the recurrence relation for T': $T'(k) = 20k + 2 + T'(k - 1), T'(0) = 2$.

   Since this looks simple, let us just try to open this up:

   $$T'(k) = 2 + (20 + 2) + (2 \cdot 20 + 2) + (3 \cdot 20 + 2) + ... + (k \cdot 20 + 2)$$

We see that the "2" terms are just summed up $k+1$ times, so they contribute $(k+1)2$. The "20" terms contribute:

$$20\sum_{i=1}^{k} k = 20(k(k+1)/2) = 10k(k+1)$$

Together

$$T'(k) = 2(k+1) + 10k(k+1) = (2+10k)(k+1).$$

We can now substitute back in $T(n)$, using $n = 5k$:

$$T(n) = T(5k) = T'(k) = (2+10k)(k+1) = (2+10(n/5))(n/5+1) = (2+2n)(n/5+1).$$

(d) $T(n) = T(n/2) + T(n/4) + n, \quad T(1) = 4, T(2) = 8.$

Solution Done in tirgul 6

3. (Finding the missing integer) All the integers from 0 to n except one, are stored in an array $A[1..n]$. We want to find the missing integer. We could easily do this in time $\Theta(n)$ by using an extra array $B[1..n]$ to record which numbers appear in A. In this problem, however, we cannot access an entire integer in A with a single operation. The elements in A are represented as binary strings, and the only operation we can use to access them is to ask for the ith bit of $A[j]$. Each such query takes constant time. Give a pseudocode for an algorithm that finds the missing integer using $\Theta(n)$ such queries. Prove your claim about the running time of your suggested algorithm.

   Solution Done in Tirgul 6

4. (Insertion and merge sort combined) Despite the fact that merge sort runs in $\Theta(nlog(n))$ and insertion sort in $\Theta(n^2)$, the constants are such that it is preferable to use insertion sort for small inputs. Consider the modification of merge sort

   modified-merge-sort[A,p,r]
   If $0 < r - p \leq k - 1$ then insertion-sort[A,p.r]
   If $r - p > k - 1$
   then $q = \lfloor (p+r)/2 \rfloor$
   modified-merge-sort(A,p,q)
   modified-merge-sort(A,q+1,r)
   Merge(A,p,q,r)

   This corresponds to sorting n/k sublists, each of length k, by insertion sort, and then merging them using the standard recursive merge sort mechanism. k is a value to be determined, and you can assume for simplicity that n/k is a power of 2.

   (a) Show that the running time of modified-merge-sort is $\Theta(nk + nlog(n/k))$.

   Solution
   We will consider the recursion tree. Let us denote its height by h. At each node of the tree, we perform a merge, except at the leaves. If j is the depth of a node, then the size of the problem it deals with is $n/2^j$. The two edges going down from this node lead to problems of half the size, except if the size s is $s \leq k$, we stop (and perform insertion sort.) So n is divided by 2, h times, until it reaches a problem size equal to k. Hence, $k = \frac{n}{2^h}$, or, equivalently the height of the tree is $h = log(n/k)$.
   Let us calculate the amount of work that is being done at each node of the tree: At an internal node of depth j, i.e. a node where merge takes place, the amount of

work is $O(n/2^j)$. At a leaf, where the size of the problem is k, we perform insertion sort, which takes k^2.

We now sum over the work in all the nodes at one level (not the bottom one). At the jth level (counting from the top) we have 2^j nodes, each performs $O(n/2^j)$ operations, altogether $2^j O(n/2^j) = O(n)$ for the jth level.

There are $log(n/k)$ levels, so we get $O(nlog(n/k))$ operations due to the merging in the recursion tree.

Now we need to add to this the work done at the lowest level, i.e. at the leaves: The work for the insertion sort. We have n/k leaves, and each one involves $O(k^2)$ operations; Together this gives $n/kO(k^2) = O(nk)$ operations.

Adding the work from the merge and from the insertion sort together we get the desired result.

(b) What is the largest k (as a function of n in Θ notation) for which the modified algorithm has the same asymptotic running time as standard merge sort?

Solution

If $K = \Theta(log(n))$ we get that the work is $\Theta(nk + nlog(n/k)) = \Theta(nlog(n) + nlog(n/log(n))) = \Theta(nlog(n) - nloglog(n))) = \Theta(nlog(n))$, where the last equality follows from (1.a) since $loglog(n) = \circ(log(n))$, (because if $log(n) = m$, this is equivalent to $log(m) = \circ(m)$.)

So $K = \Theta(log(n))$ gives us performance no worse than standard merge sort.

For k which is smaller than that, we of course still get performance which is $O(nlog(n))$ since nk will just be smaller, and the second term $nlog(n/k)$ is at most $nlog(n)$.

What happens if k is asymptotically larger than $log(n)$? i.e. that $log(n)$ is negligible with respect to k? Then $nlog(n)$ will be negligible compared to nk, so the first term nk, will be much larger than the running time for standard merge sort.

hence the answer is $k = \Theta(log(n))$.

(c) How should k be chosen in practice?

Solution

We can write the running time as: $\Theta(nk + nlog(n/k)) = Cnk + Dnlog(n/k)$ For constants C, D which depend on the performance of the specific implementations. Opening this, we have $Cnk+Dnlog(n)-D(nlogk) = Dnlog(n)+n(CK-D(log(k))$. To find the best k, We thus want to find the minimum of $Ck - D(log(k)$. To do this, take the derivative of $Ck - D(log(k)$ with respect to k: it is $C - D/k$. To find the minimum, we need that this derivative is 0, which gives $k = D/C$. We can run insertion sort on small inputs to find C, and run merge sort on small inputs to find D, and pick k to be D/C.

Data Structures, Spring 2003
PROBLEM SET 2

Given on Tuesday March 16, 2003. Due: 19:00 Monday April 7, 2003.
Justify every step rigorously. Please staple multiple pages and do not submit in plastic bags.

1. During the running time of the procedure Randomized-QuickSort, How many calls are made to the random number generator in the worst case? How about in the best case? Give your answer in the Θ notation.

   Solution: Solution given in Tirgul 10.

2. Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input, and then answers any query about how many of the n integers fall into range $[a...b]$ in $O(1)$ time. Your algorithm should use $\Theta(n+k)$ preprocessing time.

   Solution

   Pseudo Code for Preprocessing

   For i=1 to k B(i)=0 // Allocate an array B of length k with integers, initialized by 0 everywhere.

   For j=1 to length A,

   B(A(j))++

   For i=2 to k

   B(i)=B(i)+B(i-1)

   At the end of this code, $B[j]$ contains the number of elements if A which are at most j. Given a query $[a,...b]$ we just output $B[b]-B[a-1]$.

   This takes $\theta(n+k)$ time (and also $\theta(n+k)$ space.)

3. Prove that there is no comparison based sort whose running time is linear for at least half of the $n!$ inputs of length n. What about a fraction of $1/n$ of the inputs? What about a fraction of $1/2^n$?

   Solution

   (a) A comparison based sort can be described by a decision tree. Assume by contradiction that there exists a comparison based sort whose running time is linear (less than cn for some constant n) for half the inputs.

   The running time in a decision tree is the depth of the node that leads to the correct leaf. So this means that at least $n!/2$ leaves are of depth $\leq cn$.

   We will show a contradiction: The number of nodes in any binary tree of depth at most than cn is less than 2^{cn+1}, so this means that

   $$2^{cn+1} \geq n!/2. \tag{1}$$

We derive a contradiction: if we take log we get $cn + 1 \geq log(n!/2) = log(n!) - 1 = \Omega(nlog(n)) - 1$, where we have used $log(n!) = \Omega(nlog(n))$ which we have shown in class.

In other words we get: $cn \geq \Omega(nlog(n)) - 2$.

By ex1 we can omit the 2 since it is negligible with respect to $nlog(n)$. We get $cn = \Omega(nlog(n)$. We prove that this is wrong. This follows from the fact that if $cn = \Omega(nlogn)$, there exists a constant such that $cn \geq dnlogn$ or equivalently, $c \geq dlogn$ but since $log(n) \to \infty$, this is false.

(b) Now suppose the fraction of the inputs is $1/n$. The argument starts like before, but replacing $n!/2$ by $\frac{1}{n}(n!) = (n-1)!$, until we get to equation 1, which in this case states $2^{cn+1} \geq (n-1)!$. Taking log we get that $cn + 1 \geq log((n-1)!) = \Omega((n-1)log(n-1))$. We get the expression $cn + 1 \geq \Omega((n-1)log(n-1))$ or equivalently $cn \geq \Omega((n-1)log(n-1)) - 1$. We first want to simplify this expression, by getting rid of the negligible 1. For this we use a little algebra $(n-1)log(n-1) - 1 = nlog(n-1) - log(n-1) - 1 \geq nlog(n/2) - log(n-1) - 1 = n[log(n) - log(2)] - log(n-1) - 1] = nlog(n) - nlog(2) - log(n-1) - 1 = nlog(n) - n - log(n-1) - 1$. Since n and $log(n-1)$ and 1 are negligible with respect to $nlog(n)$, it follows that $\Omega((n-1)log(n-1)) - 1 \geq \Omega(nlog(n))$. So again we get $cn \geq \Omega(nlog(n))$ which is a contradiction as before. So even for a fraction of $1/n$ of the inputs the algorithm cannot find the solution in linear time.

(c) Finally, suppose the fraction is $1/2^n$. We substitute $n!/2^n$ instead of $n!/2$ in the first case, and again we get $cn + 1 \geq log(n!/2^n) = log(n!) - log(2^n) = log(n!) - n = \Omega(nlog(n)) - n$. again, since n and 1 are negligible relative to $nlog(n)$, we get $cn \geq \Omega(nlog(n))$ which is a contradiction as before.

So no comparison based algorithm can give the right answer in linear time even for $1/2^n$ fraction of the input!

4. Use a min-heap to give an $O(nlog(k))-$time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists.

Solution:

We use a min heap Q, implemented by an array A of length k, which will maintain a heap data structure of the k minimal elements in the k sorted lists. Each element x in the heap Q has a key ($key(x)$) and an index ($index(x)$) denoting which list it belonged to.

Initialization: We first insert the first k elements to Q, one by one (not in a heap structure) by deleting from each list its first (minimal) element and storing it in the array, together with an index of which list it is coming from. We then apply Build-Heap which takes $O(k)$.

We allocate an array B of length n in which the merged lists will appear, i.e. which will contain the sorted elements. Now we perform the following loop: For $j = 1$ *to* n, We extract the minimal element in Q (by $Extract - min(Q)$). If its key was key and its index was i, we put $A(j) = key$ and insert to Q (by $Heap - Insert(Q)$) the first element from the list L_i, deleting it from L_i. We also update the index field of the new element in Q to be L_i.

correctness:

B is sorted at the end, because it contains all the elements, and each element that enters B is the minimum of the remaining elements in the k lists, which only increases each step.

Complexity: The initialization takes $O(k)$ steps. The loop contains n iteration, each iteration involves a constant number of Heap operations, each takes $O(log(k))$ time, and one $List - delete$ operation which takes $O(1)$ time.

The overall complexity is thus $O(k)$ for initialization plus $n(O(log(k)) + O(1))$ which all together gives $O(nlog(k))$.

5. Recall: the max heap property is that every node is larger or equal to all its descendents, (sons, grandsons, etc.) Argue the correctness of Heap-Increase-Key using the following loop invariant:

   At the start of each iteration of the while loop, the array A(1,...HeapSize[A]) satisfies the max heap property, except that there may be one violation, which occurs at the heap that is rooted by Parent(i): that A[i] is larger than A[parent[i]].

Solution As it is written, the loop invariant is not correct exactly because even at the beginning, $A[i]$ might be larger than all its ancestors, not only from its parent. We slightly change the loop invariant to make it precise:

At the start of each iteration of the while loop, the array A(1,...HeapSize[A]) satisfies the max heap property, except that there may be a violation when comparing i to any of its ancestors y.

When we call $Heap - Increase - Key(A, i, key)$ the heap satisfies the heap property.

After that we substitute $A[i] = key$. We now enter the loop of $Heap - Increase - key$.

Let us now prove that $Heap - Increase - Key$ results in a heap, using the loop invariant.

Initialization: At the start of the first iteration, i is the only node which violates the heap property since we entered $Heap - Increase - Key$ with A satisfying the heap property at all nodes, and we changed after that only node i, to increase its key. The comparisons between i and his descendents are not violated since we only increased the key at i. So the only violations are between i and his ancestors.

Invariance: Suppose that the loop invariant holds up to the J'th entrance to the loop. So the only violation when we enter the loop is between i and his ancestors. Let us denote $Parent[i]$ by x and i by y.

Inside the loop, we exchange $A[Parent[i]]$ with $A[i]$. So we exchange the values at x and y. Where can we now have violations of the heap property? We just need to look at comparisons between nodes we have changed and their descendents and ancestors, because the other nodes remain as before so the heap property doesn't change there. So we need to check the comparisons of x with his descendents and ancestors, and y with his descendents and ancestors.

To prove the loop invariance, we need to prove that the only possible violation is between x and his ancestors, since x (which is parent[i]) becomes the new i when we go into the next iteration of the loop. So we need to check that after the exchange the other three types of comparisons are OK: that there is no violation between x and his descendents, between y and his descendents, and between y and his ancestors.

We start with x and his descendents. One of the descendents is y, and we know there is no problem there because of the exchange we made. As for the other descendents, they remained as they were when we entered the loop; By the loop invariant, there was no violation between them and the node y=parent[i] when we entered the loop, so the value at those descendents is at most $A[parent[i]]$. Since what sits now in node x, which is $A[i]$, is even bigger than what was there before, $A[parent[i]]$, no new violation appeared.

We now consider y and his descendents. After the exchange y contains $A[parent[i]]$, which might be problematic since it is smaller than what was there before, $A[i]$. BUT, we know from the loop invariant, that when we entered the loop the loop-invariant was correct at all places except i. In particular, it was correct at the node x compared to its descendents (except for the descendent y.) This means that $A[parent[i]]$ is greater or equal than what sits in all the descendents of x except y. But this means that what sits in y, $A[parent[i]]$, is greater or equal than what sits in all the descendents of y, since they are also descendents of x. so there is no violation between y and his descendents.

It remains to show that there is no violation between y and his ancestors. Again, y now contains $A[parent[i]]$.

For one ancestor, x, we know there is no problem because after the exchange it contains $A[i]$ which is bigger than $A[parent[i]]$ (that's why we entered the loop).

Now, for the other ancestors of y, which are above x. Let us call one such node z, and we want to show that $A[z] \geq A[parent[i]]$ (for all such $z's$).

This follows from the loop invariant, as we now prove. we know that when we entered the loop, there WASN'T any problem between z, (which is an ancestor of x), and x, because the only problem was at y. But when we entered the loop, $A[parent[i]]$ was the key at x, and $A[z]$ was the key of z, so if there was no problem we have that $A[z] \geq A[parent[i]]$.

Termination: At termination, there are two cases. Either $i = 1$, at which case it has no ancestors so by the loop invariant the heap property is satisfied everywhere.

The second case is that we found a place where $A[i] \leq A[parent[i]]$ and so we do not enter the loop. By the loop invariant, at this point the heap property holds for all nodes except maybe in the comparisons between i and its ancestors. We want to check that these comparisons are also OK. For it first ancestor, $Parent[i]$ we know the comparison is OK since that's why we didn't enter the loop.

All its other ancestors are also ancestors of $parent[i]$, so by the loop invariant we know that their keys are greater than $A[parent[i]]$, and so they are also greater than $A[i]$.

hence at termination no heap property is violated.

Data Structures, Spring 2003
PROBLEM SET 3

Given on Friday April 11, 2003. Due: 19:00 Sunday May 4, 2003.
Justify every step rigorously. Please staple multiple pages and do not submit in plastic bags.

1. Suppose that another data structure contains a pointer to a node y in a binary search tree, and suppose that y's predecessor z is deleted from the tree by the procedure TREE-DELETE. What problem can arise? How can TREE-DELETE be rewritten to solve the problem?

2. The black height of a Red-Black tree is the black height of its root. What is the largest possible number of internal nodes (without the leaf NIL(T)) in a red black tree with black-height h? What is the smallest possible number?

3. Show the red-Black trees that result when starting with an empty RB tree and inserting the following keys in the following order: 50, 46, 40, 17, 25, 7, And then deleting them in the following order: 7, 17, 25, 40, 46, 50.

4. Consider the following binary search tree T: The root is a node x colored black, with key 30. x has one son, y, colored red, with key 25. y has one son z, colored black, with key 11. Draw the tree. Now, suppose we call $Red - Black - DELETE(T, z)$.

   (a) What will happen? Can you explain the problem?

   (b) Can you write a general simple property of red black trees which tells us that the above problem never happens when we call $Red - Black - DELETE$?

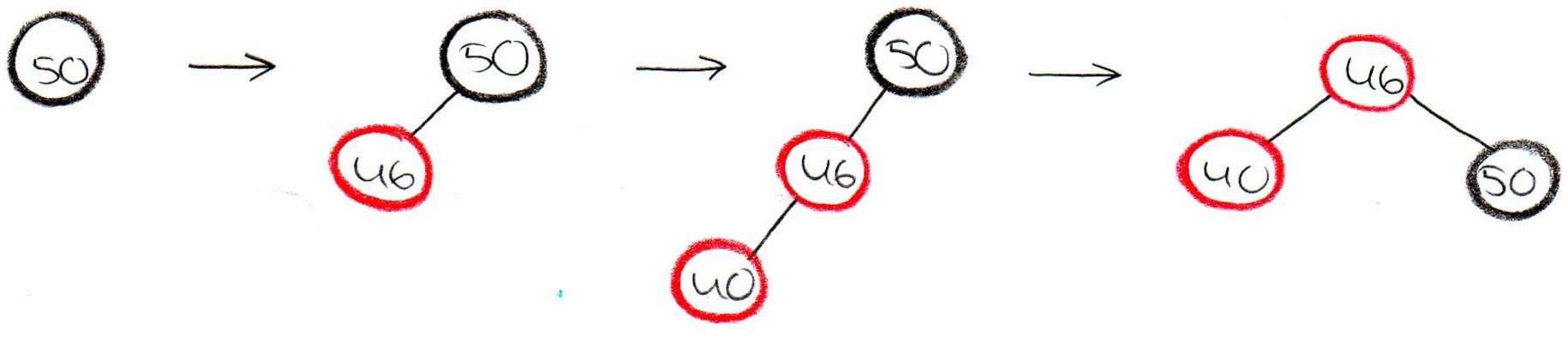
   (c) Prove that this property holds in any RB tree.

5. For the following problem you will need a simple fact about trees: A tree with n nodes has exactly $n-1$ edges. (It is simple to prove this by induction, but you are not requested to do this.) Let T be a binary search tree with n nodes. An in-order tree walk can be implemented by finding the minimum element of the tree using TREE-MINIMUM and then making $n-1$ calls to TREE-SUCCESSOR.

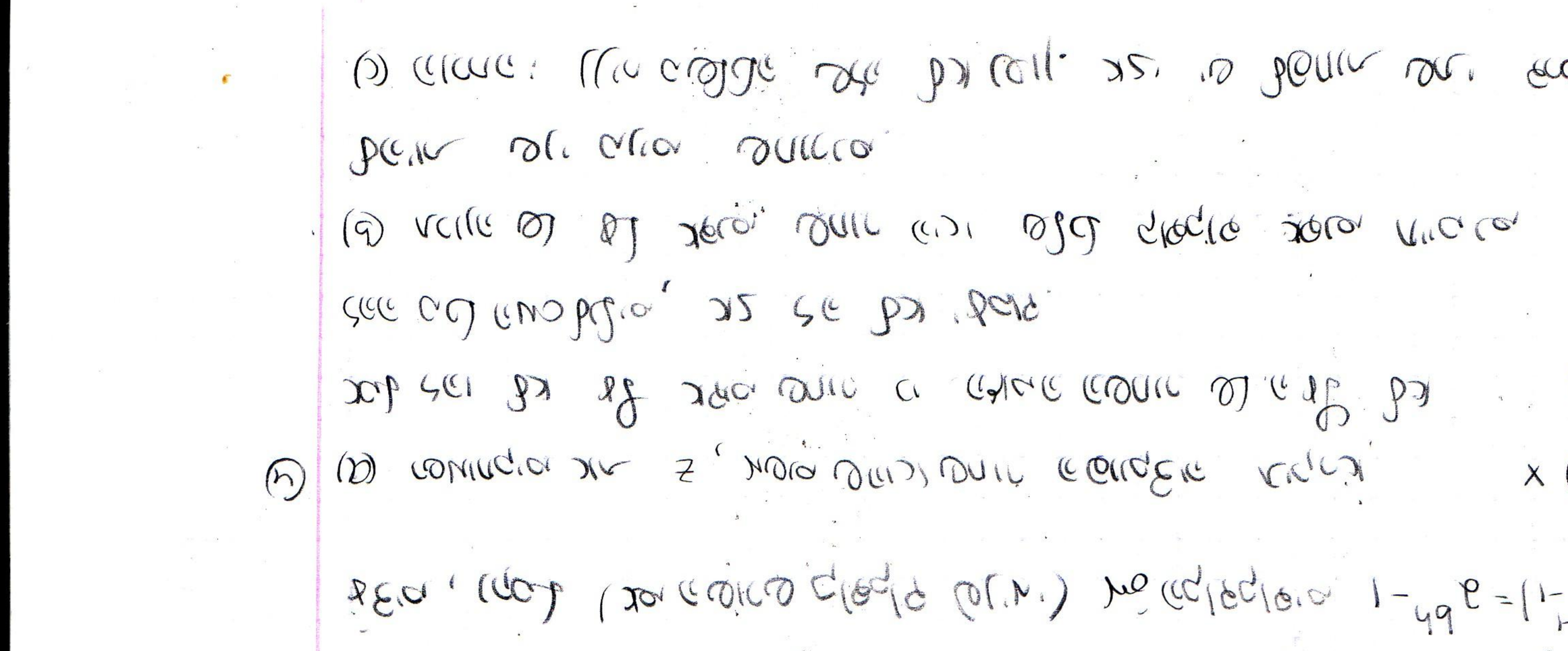
   (a) How many times (at most) do we pass through each edge in the tree, in the above in-order walk? Prove. (By "passing an edge", we mean this: inside algorithm TREE-SUCCESSOR, we move from one node to its parent or its child. We say we pass the edge connecting the two nodes.)

   (b) Prove that the above in-order walk takes $O(n)$ steps.

Problem set 3



ג) הוכיחו שלכל $h \geq 0$ מתקיים $2^{h+1}-1$ הוא מספר האיברים המקסימלי בעץ בינארי בגובה h. עץ כזה נקרא עץ בינארי מלא.



ד) הוכיחו באינדוקציה על h שלכל עץ בינארי מלא T בגובה h מתקיים $n(T) = 2^{h+1}-1$.

הוכיחו כי לכל $h \geq 0$ קיים עץ בינארי מלא T בגובה h עם $2^{h+1}-1$ צמתים.

ה) הוכיחו כי לכל $h \geq 0$ קיים עץ בינארי מלא T בגובה h עם $2^{h+1}-1$ צמתים.

ו) הוכיחו: בעץ בינארי מלא מספר העלים גדול ב-1 ממספר הצמתים הפנימיים.

ז) הוכיחו כי בעץ בינארי מלא מספר העלים הוא 2^h כאשר h הוא גובה העץ.

ח) הוכיחו כי בעץ בינארי מלא מספר הצמתים הפנימיים הוא $2^h - 1$ כאשר h הוא גובה העץ.

ט) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

י) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

יא) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

יב) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

יג) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

יד) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

טו) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

טז) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

יז) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

יח) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

יט) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

כ) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

כא) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

כב) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

כג) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

כד) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

כה) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

כו) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

כז) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

כח) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

כט) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

ל) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

לא) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

לב) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

לג) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

לד) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

לה) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

לו) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

לז) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

לח) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

לט) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

מ) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

מא) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

מב) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

מג) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

מד) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

מה) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

מו) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

מז) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

מח) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

מט) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

נ) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

נא) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

נב) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

נג) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

נד) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

נה) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

נו) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

נז) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

נח) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

נט) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

ס) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

סא) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

סב) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

סג) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

סד) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

סה) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

סו) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

סז) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

סח) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

סט) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

ע) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

עא) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

עב) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

עג) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

עד) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

עה) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

עו) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

עז) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

עח) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

עט) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

פ) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

פא) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

פב) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

פג) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

פד) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

פה) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

פו) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

פז) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

פח) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

פט) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

צ) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

צא) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

צב) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

צג) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

צד) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

צה) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

צו) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

צז) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

צח) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

צט) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

ק) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

קא) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

קב) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

קג) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

קד) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

קה) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

קו) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

קז) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

קח) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

קט) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

ר) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

רא) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

רב) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

רג) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

רד) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

רה) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

רו) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

רז) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

רח) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

רט) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

ש) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

שא) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

שב) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

שג) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

שד) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

שה) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

שו) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

שז) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

שח) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

שט) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

ת) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

תא) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

תב) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

תג) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

תד) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

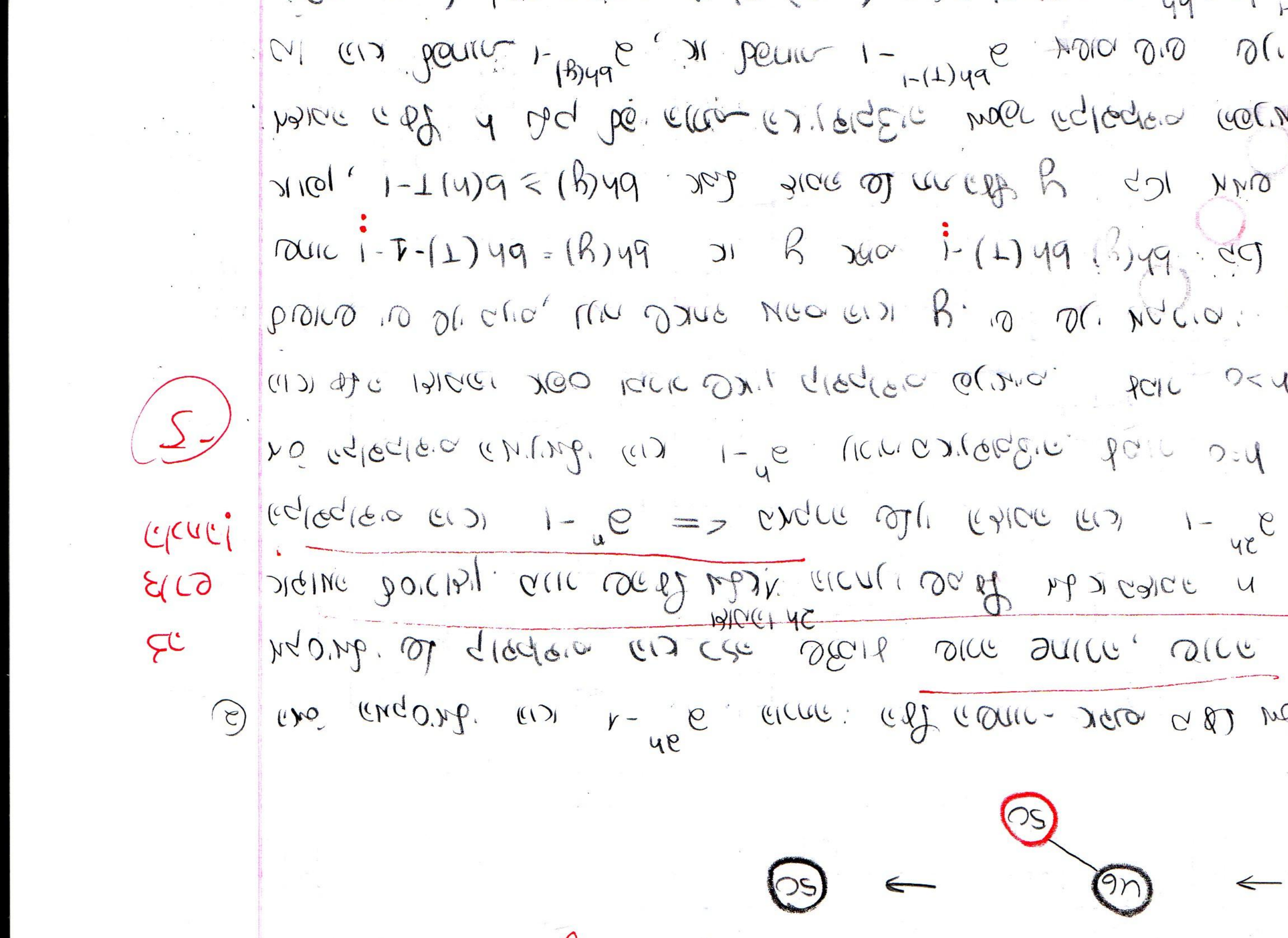
תה) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

תו) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

תז) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

תח) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.

תט) הוכיחו כי בעץ בינארי מלא מספר הצמתים הוא $2^{h+1}-1$ כאשר h הוא גובה העץ.



לחשב את הגובה השחור של העץ: הראשונה, ללכת לקודקוד הגבוה וממנו ל-NIL והשנייה, ללכת לקודקוד האדום, ממנו לשחור ואז ל NIL.
אבל אז מתקבלים שני גבהים שחורים שונים: h ו- h+1,
וזה סותר לתכונות העץ. מש"ל.

(ב) האלגוריתם עובר על כל צלע פעמיים, משום שבתחילת האלגוריתם מוצאים
את האיבר המינימלי ומשם עוברים על כל הצלעות בדרך אליו
בסוף האלגוריתם, אחרי שמצאנו את האיבר ~~המקסימלי~~ מינימלי? נחזור דרך כל הצלעות
בדרך אליו לשורש. => מספיק להוכיח שברגע שעברנו מקודקוד להורה,
לא נעבור שם שוב. צריך להוכיח את זה לכל תתי העצים של העץ.
נוכיח באינדוקציה על גובה העץ h. עבור h=0 אין צלעות. עבור h=1: אם
לשורש יש בן יחיד יש שני מקרים: אם הבן שמאלי נעבור על הצלע ~~פ~~ פעם
אחת כדי למצוא את המינימום ואז נעבור עליה שוב כדי למצוא את העוקב.
אם הבן ימני השורש הוא המינימום ואז נעבור על הצלע פעמיים: פעם אחת
כדי למצוא את העוקב ופעם אחת כדי לגלות שאין לבן הימני עוקב.
אם יש לעץ שני בנים נעבור על הצלע השמאלית פעם למציאת המינימום
ופעם למציאת העוקב ואז נעבור על הצלע הימנית פעם למציאת
העוקב ופעם בדרך חזרה לשורש.

(באינדוקציה מלאה אין צורך בבסיס הצעד מכסה (הוא מתקיים באופן ריק))

נניח שהטענה נכונה לכל n>h ונוכיח עבור h. לפי הנחת האינדוקציה
הטענה נכונה לכל הצלעות חוץ מאלו הצלעות שבין השורש לבנים שלו.
אם הבן השמאלי של השורש אינו NIL הוא בהכרח קטן יותר ובגלל זה
מגיעים אליו לפני ~~שמ~~ שמגיעים לשורש או לבן הימני שלו. אחרי
שעוברים מהבן השמאלי לשורש עוברים דרך הבן הימני (כי העוקב
של השורש נמצא בתת העץ הימני) ומרגע שלא חוזרים לשורש כי
האיברים בתת העץ הימני גדולים ממנו. לכן לא נעבור בכל צלע יותר
מפעמיים.

(ג) בעץ עם n קודקודים ו- n-1 צלעות, בכל קודקוד מבצעים מס'
סופי של פעולות d ועוברים דרך לכל היותר $2(n-1)$ צלעות.
=> זמן הריצה הוא $O(d \cdot 2(n-1)) = O(n)$.

(1) אם ל-z יש שני בנים מעבירים את הזוג של העוקב של z
למקום של z ומוחקים את y. כלומר, המצביע החיצוני
ישאר לבי מצביע על מה שהיה המצביע קודם

<u>fixed-tree-delete(T, z)</u>

```
if left(z) = NIL or right(z) = NIL then
    y ← z
else
    y ← tree-successor(z)
if left(y) ≠ NIL then
    x ← right(y)
if x ≠ NIL then
    p(x) ← p(y)
if p(y) = NIL then
    root(T) ← x
else
    if y = left(p(y)) then
        left(p(y)) ← x
    else
        right(p(y) ← x
if y ≠ z then
    left(y) ← left(z)
    right(y) ← right(z)
    p(y) ← p(z)
    if z = left(p(z)) then
        left(p(z)) ← y
    else
        right(p(z)) ← y
    return(z)
else return(y)
```

Data Structures, Spring 2003
PROBLEM SET 4

Given on Monday May 12, 2003. Due: 19:00 Monday May 26, 2003.
Justify every step rigorously. Please staple multiple pages and do not submit in plastic bags.

1. (a) What is the optimal code for the following set of frequencies: a: 1, b:1, c:2, d:3, e: 5, f: 8, g: 13, h:21 (the frequencies are relative: if you want to look at them as percentage, divide by their sum.)

   (b) Suppose you want to send a message of k characters, using only the characters a,b,c,d,e,f,g,h. How many bits will you need to send on average using the standard fixed length code? How many bits will you need to send on average using the code you generated?

   (c) Can you generalize your answer to the first item to find the optimal code when the frequencies are the first n frequencies of the Fibonacci numbers?

2. You are stuck in a square maze of n by n squares, and from each square you can go to some or all directions: north, east, west or south. You really want to get out of the maze, but you don't know how! Luckily, you have many coins in your pocket. How will you use the coins to get out of the maze? Describe your algorithm exactly. Do not use recursion in your description; Give exact instructions that you could follow if you are inside a Maze...

3. Your car has a very small gas tank: It can only drive 30 km and then you need to fill it again. The problem: You live in a country were there are no gas stations on the roads; just inside the cities... You are given a list of n cities, including your city, and the distances from any city to any other city. Give an efficient algorithm to find a list of all the cities you can reach from your home city. Give an upper bound on the complexity of your algorithm (the smaller the better.)

4. (a) Prove, assuming the lemma we proved in class about Dijkstra, that the nodes are put in the set S by the Dijkstra algorithm according to their distance from the source s. In other words, if the (correct) distance of a node v from the source s is shorter than the correct distance of the node w from s, then v will be put in S first. Do not use induction in your proof. (Hint: Assume in contradiction that the node w which was last inserted to S is farther from s than some other node y out of S.)

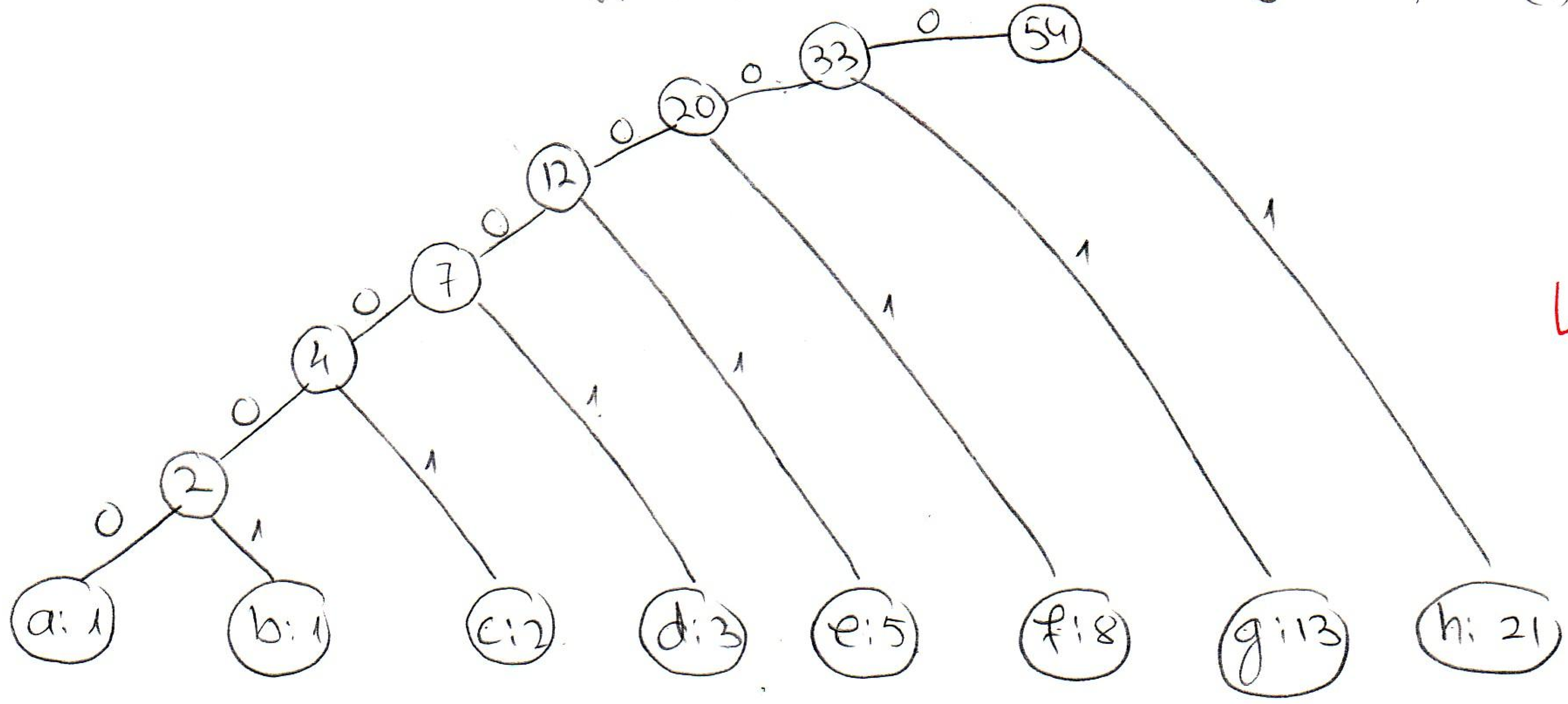
   (b) Assume that the lengths of the edges in a graph G are positive integers, denoted by $c(v, w)$ for an edge (v, w). We build an unweighted graph H from G, by replacing each edge (v,w) in G by a series of c(v,w) edges of length 1. This gives us H. (we also add c(v,w)-1 nodes per edge- we call these new nodes "dummy" nodes, and the old nodes are "real".) Show that BFS on H discovers the real nodes exactly in the same order that Dijkstra discovers them (i.e. puts them in S) when we run Dijkstra on the graph G. (You may assume for simplicity that any two different nodes have different distances from s. You may assume anything that is proven in the handouts about BFS.)

(1)

96

Problem Set 4

1 (a) הקוד האופטימלי הוא Huffman Code:



(b) יש 8 אותיות אז אם משתמשים בקוד קבוע באורך (3 בדרך) 3 ביטים לכל אות, בשביל לשלוח K תווים נשתמש ב 3K ביטים.
אם נשתמש בקוד שיוצר בסעיף (a) נשתמש בפחות:

$$\frac{K(1\cdot7+1\cdot7+2\cdot6+3\cdot5+5\cdot4+8\cdot3+13\cdot2+21\cdot1)}{1+1+2+3+5+8+13+21} \approx 2.44K$$

(c) נסתכל על הקוד שיוצר:

(1) a 0000000
(2) b 0000001
(3) c 000001
(4) d 00001
(5) e 0001
⋮
(8) h 1

נשים לב שאם מספר האותיות הוא n אז הקוד הוא כזה: עבור האות הנדירה ביותר (n-1) פעמים 0 עבור האות הבאה: ((n-1)-1) פעמים 0 ואז 1 וכן הלאה. עבור האות במקום ה-K ((n-1)-(K-1)) פעמים 0 ואז 1. עבור האות השכיחה ביותר הקוד הוא פשוט 1.
נטען שלכל n מתקיים $S_n = a_{n+2}+1$. נוכיח באינדוקציה. ב (a) ניתן

להראות את נכונות הטענה לכל $n>7$. נניח עבור k ונוכיח ל: $k+1$

$$S_{k+1} = S_k + a_{k+1} \underset{\downarrow}{=} a_{k+2} + 1 + a_{k+1} \underset{\downarrow}{=} a_{k+3} + 1$$

משל   לפי הנחת האינדוקציה   לפי הגדרת הסדרה

מהטענה נובע ש-$S_n < a_{n+2}$ לכן בהגדר קוד Huffman נחבר ראשית את שתי האותיות הראשונות ולאחר מכן נוסיף את האות השלישית, הרביעית וכן הלאה עד האות האחרונה. בכל שלב יווצר לנו מקודקוד אחד הסכום S_n ואליו נחבר את האות ה a_{n+1} משום ש $S_n < a_{n+2}$ $a_{n+1} < a_{n+2}$. ככה לכל אות הקוד יהיה רצף של 0 ו-1 בסוף.

② ✓ אלגוריתם:

1) כל עוד לא נמצאים ביעד הבא

1.1) הנח מטבע במעבר שבו הכי מעט מטבעות. אם יש כמה כאלה הנח את המטבע במעבר הראשון לפי כיוון השעון

1.2) עבור דרך המעבר שהנחת בו מטבע

2) צא

לא נדרשת נאין מומלץ לציין (ומבחינה אין חובה) מדוע אלגוריתם זה נכון. (DFS). נ״ב מחמין ה.ספק.

③ ✓ מטריצה $dist(i,j)$ מייצגת את המרחק בין העיר ה-i לעיר ה-j.

אתחול מערך בגודל n boolean[1..n] hmm = false

cities(dist, 0, hmm)

[illegible]

```
cities(double[][] dist, int i, boolean[] hmm)
1)  for j=1 to (n-1) do
1.1)  if dist(i,j) ≤ 30 and hmm(j) = false then
1.1.1)  hmm(j) = true
1.1.2)  print(j)
1.1.3)  cities(dist, j, hmm)
```

-2 בעצם שיפור של BFS $O(|V|+|E|)$

[illegible]

במקרה הגרוע ביותר זמן הריצה הוא $O(n^2)$ כי אז המרחק בין כל הערים קטן מ-30 וקוראים ל-cities $n-1$ פעמים והלולאה שבפנים גם מתבצעת $n-1$ פעמים

ג) (a) נניח שבאיזה שלב w [אתרגש?] ל-S. $d(w) > d(s)$ כי y מחוץ ל-S
⇐ שלב קודם $d(w)$ הכי קטן מכל הקודקודים שמחוץ ל-S (כי ככה
עובד האלגוריתם). אבל לפי מה שהוכחנו $dist(y)$ הוא אורך המסלול
המיוחד הקצר ביותר מ-v ל-y. $d(y) < d(w) < dist(w)$. צ"ל קיים מסלול
מ-v ל-y שקטן מ $dist(v)$ (מה? מיקוד?). ואם x הוא הקודקוד הראשון במסלול
שמחוץ ל-S, כל האיברים חיוביים ולכן $d(x) < d(y)$ והמסלול הלה
עובר רק דרך S. ⇐ $dist(w) > d(y) > dist(x)$ סתירה ☺

(b) נגדיר $z(u)$ הוא מספר הקודקודים שבהם עוברים מוקדם המוצא
s ל-u. צריך להוכיח שלכל u, v שעבורם $dist(u) < dist(v)$
מתקיים $z(u) < z(v)$. הוכחה: הגרף לא ממושקל כל יחידה שווה
ערך למעבר בקודקוד יחיד ⇐ $z(u) = dist(u) < dist(v) = z(v)$.
⇐ $z(u) < z(v) \iff dist(u) < dist(v)$

$r(u) \Leftarrow z(u)$

בלי קשר 3 [illegible] [illegible]

Data Structures, Spring 2003
PROBLEM SET 5

Given on Monday May 26, 2003. Due: 19:00 Monday June 9, 2003.
Justify every step rigorously. Please staple multiple pages and do not submit in plastic bags.

You need to solve only 4 out of the 5 questions- each question is worth 25 points. (you can get up to 125 points.)

1. Let p be a large prime, larger than the universe of keys U. Let m be a much smaller integer. Let the function h_a for $a \in Z_p$ be defined by

$$h_a(k) = ((k + a) mod\ p) mod\ m).$$

Consider the family of hash functions

$$H'_{p,m} = \{h_a | a \in Z_p\}$$

Prove that this is not a universal family of Hash functions by finding a bad input, for which the properties of universal families of hash functions do not hold. Show what happens with your input.

Solution: Consider the input containing the two keys $k_1 = 0, k_2 = m$. The properties of universal hash functions say that the probability for the two keys to be hashed to the same slot are less than or equal to $1/m^2$. We will show that for all $a \in \{0, ..., p-m-1\}$ the two keys are hashed to the same slot, so the probability for collision is very high for these two keys.

First, for $a \in \{0, ..., p-m-1\}$

$$h_a(k_1) = h_a(0) = (a\ mod\ p)\ mod\ m = a\ mod\ m$$

where we used $a\ mod\ p = a$. For the second key,

$$h_a(k_2) = h_a(m) = ((a+m)\ mod\ p)\ mod\ m = (a+m)\ mod\ m = a\ mod\ m$$

where we used the fact that for $a \in \{0, ..., p-m-1\}$, $(a+m)$ is smaller than p so $(a+m)\ mod\ p = a+m$.

Hence for all $a \in \{0, ..., p-m-1\}$ the two keys are hashed to the same slot. Hence the probability for a collision is at least $(p-m)/p$ which is close to 1 because m is much smaller than p, and is much larger than the desired $1/m^2$, which is less than a quarter. (we can assume $m > 1$, since other wise there is not point in hashing.)

2. Consider Perfect Hashing, where instead of picking m=n, choose m=cn. You get to pick the constant c.

   (a) Can you find a constant c for which the expected performance in terms of worst case access to the data will be faster?

Solution: No, the access time to the data in perfect hashing does not depend on c. We use double hashing, and the time to access any input takes exactly the time to calculate the first hash function and then the second.

For those of you who tried to find c such that we never need to use double hashing, so the worst case behavior is just one calculation of the hash function. First, this is not what was meant in the question- the question referred to perfect hashing as we learnt it, with double hashing. But even if you try to do it, this is impossible. The average number of collisions is n/c, and however large c you pick, once you fix it, the average number of collisions goes to infinity as n becomes larger... You might try to find a specific hash function from the family for which there are no collisions at all, but we cannot guarantee that such a function exists in our family.

(b) Can you find c such that the average space is smaller (in terms of constants) than the average for m=n which we saw in class?

(c) If you can find the optimal c, (c*) you get a factor of $1/(c^*)$ to your grade for this question.

Solution:

We will solve $b), c)$ together. As was explained in class, we assume that each secondary hash table has three extra slots, in which its parameters are written (a_j, b_j, m_j). To avoid complications, we ignored the fact that some slots might be empty, and said that we assign three such spaces regardless of whether the slot is empty or not. The total space that the perfect hashing occupies is thus m for the main table, plus $3m$ for these parameters, plus $\sum_{j=1}^{m} n_j^2$, the total length of the secondary hash tables. This gives:

$$4m + \sum_{j=1}^{m} n_j^2$$

In class the first linear term was ignored, because the point was to prove that the average of the total space is linear, and the first term is linear in any case, so we needed to consider only the second term. But in this question constants are important, and we cannot ignore the first term. We carry it along and proceed as we did in class:

$$= 4cn + \sum_{j=1}^{m} n_j + 2\sum_{j=1}^{m} \frac{n_j(n_j-1)}{2} =$$

$$4cn + n + 2\sum_{j=1}^{m} \frac{n_j(n_j-1)}{2}$$

The last sum is as we saw in class, exactly the number of collisions, and its average, exactly as we saw in class, is $n(n-1)/2m$, which is the number of pairs in n keys times the probability for collision for each pair, which is $1/m$. Substituting $m = cn$ we get for this average $(n-1)/2c$, and ignoring the negligible 1 we get that the average over the sum is $n/2c$. Thus the over all average space is

$$4cn + n + n/c$$

to optimize over c we take the derivative with respect to c, which gives

$$4n - n/c^2$$

and requiring that it is equal to 0 we get $c* = 1/2$.

The total average space is then $3n$ instead of $4n + n + n = 6n$, so we saved a factor of 2.

3. Give an $O(|V| + |E|)$ algorithm that takes as input a directed acyclic (no cycles) graph and two vertices s,t and returns the number of paths from s to t in G. Your algorithm needs only to count the paths, not list them. Hint: Use topological Sort as part of your algorithm.

Solution We note that after a topological sort, if the nodes are put in a linked list, then there are only edges from a node to nodes further down the list.

The idea of the algorithm is that the number of ways to reach a node v from s, is the sum over all nodes w with edges (w,v), of the number of ways to reach w from s.

If we first use Topological sort, then we can go through the nodes in topological order, and each time we reach a node we add the number of ways to reach it, to the number of ways to reach any of its neighbors down the sorted list. This way when we reach

the neighbor, we have already counted all the ways to reach it because we already went through all nodes from which we can reach that node (because all edges go forward in a topological sort.)

What we do is we first sort the graph by topological sort, and initialize an array called Paths of length n with 0's, and we set Paths[s]=1. This array will eventually contain the number of paths from s to each node. We then go through the nodes from the beginning of the list, and each time we are at a node v, we look at all edges going out of that node, and for an edge (v,u) we add the value of Paths[v] to the value of Paths[u].

As always, we denote $n = |V|$.

Here is the Pseudo code for this algorithm:

(a) Algorithm-count-paths$(G = (V, E), s, t)$
(b) Topological-sort(G) // The sorted graph sits now in a linked list L.
(c) //$L[i]$ is the ith node in the list
(d) For $i = 1$ *to* n do $Paths[i] = 0$ // Initialize an array of length n with 0.
(e) Paths[s]=1 // Initialize Paths[s] so that we can start counting paths from s.
(f) For $i = 1$ *to* n
(g) For all edges $(L[i], v)$ out of $L[i]$ do
(h) $Number[v] = Number[v] + Number[L[i]]$.
(i) Output Paths[t]

The complexity of this algorithm is $O(n + |E|)$: The topological sort takes $O(n + |E|)$ (as we saw in the tirgul, just like DFS). The initialization takes $O(n)$. The FOR loop of lines (f)-(h) takes $O(n + |E|)$ [since the loop has n steps, which contribute O(n), and inside it we go through each edge exactly once and operate a constant amount of work in line (h) for it, which gives $O(|E|)$ when counting the total work over all edges.]

4. Recall the doctor running the surgery room using the system of difference constraints. Suppose that **in addition** to difference constraints, the doctor also has equality constraints of the form $x_j = x_i + c_{i,j}$, saying, for example, that this patient must get into the surgery room exactly some number of hours after he entered it before. Give an algorithm that the doctor can use to solve this mixed system of equality and inequality constraints. What is its running time?

Solution

An equality $x_j = x_i + c_{i,j}$ can be replaced by two inequalities:

$$x_j \leq x_i + c_{i,j}$$

$$x_j \geq x_i + c_{i,j}$$

we need to write the inequalities such that they fit the format of the difference constraint system of inequalities, which uses only less than or equal, and not greater or equal:

$$x_j - x_i \leq b_{ij}$$

so the first inequality becomes

$$x_j - x_i \leq c_{i,j},$$

and the second:

$$x_i - x_j \leq -c_{i,j}$$

Hence for each equality we need to add the above two inequalities to the system of difference constraints, and solve as before, using Bellman Ford.

If $|E|$ is the number of difference constraints and equalities, and n is the number of variables, we get using Bellman Ford a running time of $O(n|E|)$.

5. Suppose the government decides to completely change the telephone system in Israel, and use a new kind of telephone wires. Suppose that you are chosen to design the net of phone wires, which needs to connect n given locations in Israel. It suffices that there is some path connecting any two locations to allow all people to talk with each other. You know the distances between any two locations, and your main goal is to save the total length of wires in the net, since implementing every meter of wire costs a lot of money. When you start programming the algorithm, your boss tells you casually that some wise guy had already gone and started to implement the wire between two locations in your list, Rosh-Pina and Haifa.

   (a) How would you decide, given your data, if it is OK that they have started to implement the wire, or whether you could have designed a cheaper net in which that wire would not appear?

   (b) Suppose that you decide that it was a smart move. How would you modify Prim's algorithm to design the cheapest net with that Rosh-Pina-Haifa wire included?

Solution

We answer both items together. The mission of finding a net of wires of minimal total length is actually exactly to find a MST. The question is whether there exists a MST of the graph in which the edge (Rosh-Pina,Haifa) exists. To do this, first run an MST algorithm, and then find the weight of a MST by running over the tree and adding the weights of the edges in the tree one by one. Suppose the weight turns out to be w. We then want to see if there exists a ST (spanning tree) with weight w which contains a specific edge. There are two ways to do it: a simple easier way which requires an observation about MST's, and a more straight forward way.

Solution I: (more elegant)

We observe

claim *if e is a unique edge of minimal weight in a graph, than any MST must contain it.*

To prove this, suppose there is an MST T that does not contain it. As usual, add e to T, it creates a cycle; if you remove any edge from the cycle you still have a connected graph (same proofs as in the class for all these cases), it has n-1 edges, so it is a spanning tree, but it is less heavy than T, in contradiction to the fact that T is MST. This proves the claim.

What we can do is give the graph as input to Prim, but change the weight of the edge Haifa-Rosh-Pina to 0, so that we are sure it is included in the MST of the new graph (which is the same graph but with that weight changed.) Then we apply prim's algorithm as usual, and find an MST T. T will contain the edge e by the claim.

We claim that T is a spanning tree of minimum weight among all those that contain e (we use the original weights to calculate the weight.) This is true because if there was a lighter tree which contained e, T', than if we reduce the weight of e from both trees, we would have gotten a contradiction to the fact that Prim finds a MST for the graph G with the modified weight (i.e. weight of e equals 0.)

Now, we want to calculate the weight of the tree we found; We just run through the tree we found, and calculate the sum of its weights (using the original weights.) We finally compare the result to w to see if it is bigger or not.

If not, then there exists an MST which contains e! Otherwise there isn't...

During the process we found a MST containing Haifa-RoshPina, if one exists: It is the output of Prim's algorithm when we gave it the weight of e =0.

Solution II

If you missed the point of reducing the weight of w to 0, you could also simply modify Prim's algorithm by Brute-Force to make sure the edge Haifa-Rosh-Pina is included.

Essentially, we want to start Prim's algorithm not as we defined it, but with an initial edge. We modify Prim's algorithm to an algorithm that finds an MST which contains one edge, (r,s), if such an MST exists. We do this by dealing with one of the cities separately, say s, not inserting it to the queue. We start the algorithm treating r like it was treated in the original algorithm, but s (Haifa, for example) will not enter the queue, and we will update all the dist of its neighbors separately before we start the main loop. We will point from s to r in the tree we form.

i. Algorithm-Modified-Prim (G,s,r)
ii. Q=V-s
iii. For each $v \in Q$ $dist[v] = \infty$
iv. dist[r]=0
v. Parent[r]=Nil.
vi. Parent[s]=r
vii. For all neighbors v of s except r,
viii. Parent[v]=s
ix. dist[v]=w[s,v]
x. while Q not empty
xi. u=ExtractMin(Q)
xii. for each v adjacent to u,
xiii. if ($v \in Q$ and $w(u,v) < dist[v]$)
xiv. parent[v]=u
xv. dist[v]=w(u,v)

We claim that if the weights are all of positive weights, then if there is an MST that contains (s,r), then the above algorithm finds such an MST, and if there isn't, the algorithm finds a tree with larger weight.

This is true because after the first step of extracting r, the state is that Dist contains the distance of the rest of the nodes from the small tree we already have (the edge (r,s)). So if there is an MST that contains this edge, the algorithm will find it and this follows from the correctness of the MST property, that if the subtree is part of an MST, adding the minimal edge connecting U and V-U will still be a subtree of an MST.

We also note that the algorithm outputs a spanning tree, since each node except r is connected at the end to some parent, so we have n-1 edges, and there are no cycles because each time we connect a new node.

Once we found this MST, we can run through it and find its weight, and compare it to w.

If the weight is the same, the answer is that it was OK to start working on the wire Haifa-Rosh-Pina.

During this process we already found the MST which uses this wire.