Algorithms HW3

Problem 1

a) Lookup x_4 $x_1 x_2 x_3 x_4$ Paid 4
Lookup x_2 $x_4 x_1 x_2 x_3$ Paid 3
lookup x_4 $x_2 x_4 x_1 x_3$ Paid 2
lookup x_2 $x_4 x_2 x_1 x_3$ Paid 2

final list $x_2 x_4 x_1 x_3$
total cost paid = 11.

b) Let L be the list of elements that behave with the MTF heuristic. Let L^* be the list of elements that does not move.

Both list start with the elements in the same order. Now lets define the following sets. in reference to the element x we want to access. in the ith access.

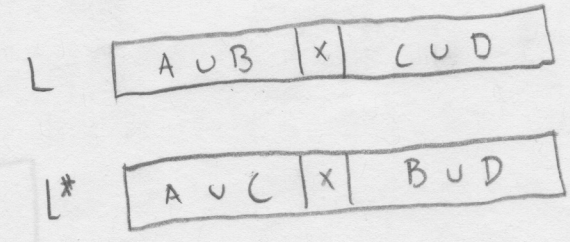
A: consists of elements that precede x in both L and L^*

B: consists of elements that precede x in L but are after x in L^*

C: consists of elements that follow x in L but precede x in L^*

D: consists of elements that follow x in both L and L^*.

Therefore before the ith access the two lists look like this



in the ith access the cost of accessing x in either array is the amount of elements that one needs to traverse in order to get to X + 1. That is

$$c_i = |A| + |B| + 1$$

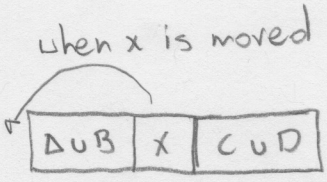
$$c^*_i = |A| + |C| + 1.$$

For the Move to front heuristic one can define the amortized cost of accessing element x as the following.

$$\hat{C}_i = C_i + \Delta\Phi$$

and we can define Φ as the number of inversions in each step thus here $\Delta\Phi = \Phi_i - \Phi_{i-1}$.

So if we had



Because the elements in A precede x in L^* then we are creating at most $|A|$ more inversions in L and because the elements of B follow x in L^* we are getting rid of atmost $|B|$ inversions in L by moving x to the front of the list. We know nothing about the relative organization of A and B within themselves and with one another, thus we need to place these bounds.

Therefore $\Phi_{L_i} - \Phi_{L_{i-1}} \leq |A| - |B|$

So now we can say that the amortized cost of a single lookup in L is

$$\hat{C}_i = C_i + \Delta\Phi$$

$$\hat{C}_i \leq C_i + |A| - |B|$$

$$\hat{C}_i \leq |A| + |B| + 1 + |A| - |B|$$

$$\hat{C}_i \leq 2|A| + 1.$$

Now remembering that the cost of the same access in L^* $C^*_i = |A| + |C| + 1$ then we know that for a single access the amortized cost for an access in L $\hat{C}_i \leq 2|A| + 1$ which in turn is certainly less than twice the cost of the same access in L^* $2C^*_i = 2|A| + 2|C| + 2$

That is the cost for a single operation

$$\hat{c}_i \leq 2c^*_i$$

because

$$\hat{c}_i \leq 2|A| + 1 \leq 2|A| + 2|C| + 2 = c^*_i$$

And because the inequality holds for every single access it must be that it holds for the sum of the costs over m accesses.

c) An interesting insight is that we made no assumptions about the relative organizations of the sets in the above proof, up to the point of the total cost of access. In the proof above we used as a particular fixed list "hit" but there is no reason we can't generalize our list L^* to any fixed list. Therefore it is still possible to use the inequality above for the cost of a single access in which $c_{i\,MTF} \leq 2c^*_i$ since the "static list in this problem certainly qualifies as a fixed list L^*. And lets assume further that we can calculate the number of inversions in the MTF list by having as reference the position that they have in the static list.

Thus we can say that the amortized cost of 1 lookup in the MTF list is $\hat{c}_i = c_i + \Delta\Phi$ where Φ is the number of inversions with respect to positions in the static list.

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Delta\Phi$$

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i) + \Phi_{final} - \Phi_{initial}$$

$$\downarrow$$

$$\text{So } C_{MTF} \leq \sum_{i=0}^{n} \hat{c}_i - (\Phi_{final} - \Phi_{initial}).$$

Finally we know that relative to the positions in C_{STATIC} the maximum difference of inversions in MTF has to be the maximum possible number of inversions or $\binom{n}{2}$ given there are that many ways to choose a pair in a list of size n. So

$$C_{MTF} \leq \sum_{i=0}^{n} \hat{c}_i + O(n^2) \leq 2\sum_{i=1}^{n} c^*_i$$

And because we know from b and keeping the rules described above that total cost of a move to front list is at most 2 times the cost of any given fixed list then it follows that

then

$$C_{MTF} \leq 2C^*_i + n^2$$

or $C_{MTF} \leq 2C_{STATIC} + n^2$.

Problem 2a

let our counter increase up 2^k times thus increasing to number 2^k; that is in binary this number would be represented by a 1 trailed by $\log(2^k)-1$ 0. If we decrease this number once then we need to flip exactly $\log(2^k)$ bits. If we proceed to then increase 2^k-1 to 2^k again then we would again have to flip $\log(2^k)$ bytes.

If we then proceed to decrease 2^k, and then increase 2^k-1 2^k times then all of these operations would all cost $\log(2^k)$ each as everytime we would be flipping $\log(2^k)$ bytes.

The 2^k increases that took us from 0 to 2^k cost each an amortized cost $O(1)$ as we discussed in class.

Therefore, we can say that we need to account for the 2^k increase/ decreases. Because the cost of doing each add every operation at this point is $\log(2k)$ and we are doing this 2^k time the lower bound for the total cost of this operations is $2^k \log(2^k)$ thus we have achieved a sequence of n operations for which the amortized cost per operation is $\Omega(\log(n))$ for a total cost of $\Omega(n \log(n))$

2b For this problem lets consider the binary number to be stored in an Array (which is big enough to hold any number that we care about). So each element of A starts out as 0 and is always -1, 0, or 1. Let A[0] be the least significant bit then A[1] and so on.

lets think of each bit (each element of A) having its own "bank". When we do an operation that flips A[i] from

0 to 1 or from 0 to -1 we charge 1 token for the operation and an extra 1 token to store in the bank. When we flip from 1 to 0 or from -1 to 0 we will use a token from the bank to pay for the operation.

Now lets analyze how we would increment or decrement our binary counter. In any given operation we add 1 or subtract 1 from the least significant trit. There are 4 easy cases to consider and 2 trichier ones.

The four easy cases are as follows; if the least significant trit is 1 and we subtract 1 then the trit flips to 0 charging from the bank and we are done. Likewise, if the least significant trit is -1 and we add 1 we take a token from the bank and flip the trit to 0 and we are done. In the next two cases if the least significant trit is 0 and we either add or subtract we charge 2 and then flip the trit from 0 to 1 or from 0 to -1.
In any of this cases it is easy to see that at most, any operation costs 2.

Now in the two trichier cases we have that the least significant bit is 1 and we add, producing a carry to the next least significant bit but not charging (charge from 1 to 0 takes a token from the bank) or we have -1 and subtract, producing a borrow but not charging (as charge from -1 to 0 takes a token from the bank).

However, from the four cases above we see that as soon as we flip a trit from 1 to 0 by subtracting and from -1 to 0 by adding the propagation stops (an no charge is incurred). More importantly though as soon as a trit is flipped from 0 to 1 by adding or from 0 to -1 by subtracting, the propagation stops only having charged 2.

Therefore any given operation might have to flip a different amount of trits but only at most one of those trits is going to flip from 0 to 1 or from 0 to -1 making the charge of any operation at most 2.

that is, for any given increment or debrement we pay at most 2 and thus for n operations we pay at most 2n. Thus the amortized cost per operation is O(1) and for n operations it is O(n).

Problem 3

a) For our algorithm the only two functions that create new sets are make set and union set.

For make set we return a node which has as it parent itself. In other word Find (Make set x) returns x; for this case $u = v$ so ranks are the same. However, we must assume that for this problem $u \neq v$.

In the base case we union two singletons u and v. Based on the union by rank rules because the ranks of u and v are equal, in this case, then we'll make v point to u and increase the rank of u to 1. Then in this case it is true that u is the parent of v and u's rank is strictly larger than v' rank.

Now lets assume we have two trees such that if u is the parent of v then u's rank is strictly larger than that of v. for all nodes in both of the trees.

Now if we proceed to union these two trees by the rules of union by rank one of three things will happen: the root of the first tree has a rank less than the root of the second tree, the root of the first tree has a rank greater than the root of the second tree, or both roots have equal ranks. In the first two cases the lower ranked root will now point to the greater ranked root. In these two cases the higher ranked root is the parent of the lower ranked root in which case it holds that if u is the parent of v then the rank of u is greater than the rank of v. In the third cases we will pick one of the two nodes and make it point to the other node, increasing the rank of the other node. Therefore the node that is now the parent has a rank higher than the other node.

Because we previously had two trees for which the rule held throughout all the nodes and the only new child to parent connection we formed still enforces the rule, then it must be true that for all nodes in this new tree, if u is the parent of v then the rank of u is strictly larger than the rank of v.

Problem 3

b) In the base case let all nodes be singletons. Then it is the case that all nodes have rank 0. Moreover it is trivially true that for any node, it is the root of a tree with $2^0 = 1$ nodes in it.

Now the only way to increase r for any given singleton is to perform union of two of them. After this union we will have one node with rank 1 which is the root of a tree that has at least $2^1 = 2$ nodes and we have the child with rank 0 that used to be the root of a tree of size $2^0 = 1$

Now if we have any two trees with roots U V and U has rank r_1 and V has rank r_2 then by lemma 7.3.2 proved in class the root U has at least 2^{r_1} nodes and root V has at least 2^{r_2} nodes

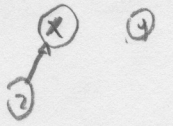
When we union this two trees it is the case that one of the roots will become a child to the other if $r_1 > r_2$ or $r_2 > r_1$ or if $r_1 = r_2$ then WLOG U will be children to V. In the first two cases the node that retains the root is still a root with rank r which has at least 2^r nodes and the node which is not a root anymore used to be a node with rank r that used to be the root of a tree with at least 2^r nodes. In the third case the node that is still a root with now rank r+1 is still a root with at least 2^{r+1} nodes in its tree (by lemma from class) but more importantly the node that is not a root anymore for which ranks didn't change used to be a root with at least 2^r nodes.

Therefore in any which way we union two trees any arbitrary node with rank r must have been at one point a root of a tree with at least 2^r nodes.

Problem claim: let u be an arbitrary node. Prove that for any rank r there is at most one node with rank r that is ancestor of u

c) For this problem again, the only two operations that make new sets are make set and union.

First lets consider when we can make a singleton set. For this case It has no ancestors so the claim is trivially true.

Base case: lets consider the union of a tree with two nodes and a singleton  and lets consider z union y then y will point to x and the ranks dont change. In this case only the ancestral line of y changes that is by getting parent x and thus for any given rank r y only has at most one ancestor with that rank, z had no ancestral change and neither did z, so the claim holds.

Now lets consider 2 trees with nodes for which the claim holds

When we union a node a of the first tree with a node b of the second tree then only the ancestry of one of the nodes will change: if the root of node a has lower rank(r) than that of node b then the root of b becomes the root of a, in which case it becomes an ancestor of a with a greater r than any other ancestor a has. b does not have its ancestry change. In the case that the root of node b has lower rank than that of node a then the root of a becomes the root of b, in which case it becomes an ancestor of b with a greater r than any other ancestor b has. a does not have its ancesty change. In the last case the root of a has the same r than that of the root of b. WLOG lets say we increase the r of the root at a and make the root of a the parent of the root of b. In this case the ancestry of b changed by adding an ancestor to b that has r greater than any ancestor of b. In any of the three cases union is changing the

ancestors of only one of the nodes and it is the case that when it does it, the newly added ancestor has rank larger than any previous ancestors.

So the claim still holds after we union 2 trees for which all the nodes had it so that for any rank r, there is at most one node of rank r that is an ancestor of said node.

Therefore in the resulting tree for any arbitrary node, for any rank r there is at most one node of rank r that is an ancestor of that node.

Problem 3

d)

By proof of b) we can state that any given node with rank r was at some point a root with rank r an at least 2^r nodes in its tree. Moreover by proof a) we know that any given child of this root must have rank lower than r. Moreover we know from prove c that there is only one ancestor of our node r that has an arbitrary rank. These two proofs allow us to assert that any two nodes with rank r must have come from disjoint sets (again of at least 2^r elements).

Altogether because there are only n elements the most number of these disjoint trees, with size at least 2^r, that is nodes with rank r that there can be is $n/2^r$.

e)

Based on the definition explained in e we could do the following table. And based on proof b we can say

bucket	rank	Max nodes
0	1	$n/2^1$
1	2	$n/2^2$
	3	$n/2^3$
2	4 (2^2)	$n/2^4$
	5	$n/2^5$
	6	$n/2^6$
	7	$n/2^7$
	⋮	⋮
	15 ($2^{(i+1)}-1$)	$n/2^{15}$
		($2^{(i+1)}$)

Now for instance for bucket 2 the sum of possible nodes in the bucket is at most $\left(\frac{n}{16} + \frac{n}{32} + \frac{n}{64} \cdots + \frac{n}{2^{15}}\right)$ and this sum can be

represented like $\frac{n}{16} + \left(\frac{n}{32} + \frac{n}{64} + \cdots \frac{n}{2^{15}}\right)$

↑ The remainder of this sum is less than $\frac{n}{16}$ by rules of power series

So the maximum number of nodes possible for bucket 2 is $2\left(\frac{n}{16}\right) = 2\left(\frac{n}{2^{2*2}}\right)$

from this we can generalize the sum of maximum possible nodes in a bukket to

$$\left(\frac{n}{2^{2*i}} + \frac{n}{2^{2*i+1}} + \frac{n}{2^{2*i+2}} + \cdots + \frac{n}{2^{2*(i+1)-1}}\right)$$

Now from this sum we know that the term that dominates the maximum possible nodes in a bucket is the maximum possible nodes in the lowest rank of said bucket and the rest of nodes in the bucket is less than the amount of nodes in that first rank by rules of the

Thus we can represent the ith bucket with lowest rank $2\uparrow i$ and thus the lowest rank can have at most $n/2^{2\uparrow i}$ nodes and the rest of the ranks combined can have at most $n/2^{2\uparrow i}$ nodes making the number of nodes in the ith bucket at most $2\left(\frac{n}{2^{2\uparrow i}}\right)$

f) In the worst case for a find operation we are going from a rank 0 node to the root node which is rank r.

now say rank r is in the ith bucket. (lowest rank in bucket r; highest rank in bucket i)

then based on g the rank is bound by $2\uparrow i \leq r \leq 2\uparrow(i+1)-1$

we also now from class that if we were to perform unions of equal ranked trees only we could perform $\leq \log(n)$ of such unions so the rank of any given root node can be at most $\log(n)$.

Therefore we can say that $2\uparrow i \leq \log(n) \leq 2\uparrow(i+1)-1$

now we are interested in the times that we might need to go accross buckets, in that case then we can cross at most to

i buckets thus. from

$$2\uparrow i \leq \log(n) \leq 2\uparrow(i+1) - 1$$

this inequality gives us the maximum number of buckets i that we can have with n elements and thus the maximum number of crossings we need to make.

$$2\uparrow i \leq \log(n) \qquad \log^*$$

$$\log^*(2\uparrow i) \leq \log^*(\log(n)) \leq \log^*(n)$$

$$i \leq \log^*(n) = O(\log^*(n))$$

So in any find path there are at most i edges that go between buckets which is at most $O(\log^*(n))$ so in m finds there are at most $O(m \log^*(n))$ of such traversals.