CSCB63 A3

Saad Makrod

August 2022

Q1 Design FIFO Data Structure With Find Minimum Operation - All Operations $O(1)$ Amortized Time

a) A queue will be used as well as a modified doubly linked list to hold the minimum of the queue. The linked list will support regular linked list behaviour, however it will have a pointer to the tail of list as well as a pointer to the head.

The job of the linked list is to keep track of a list of minimums in descending order (meaning the smallest value is at the tail). The reason a list is needed is because once the real minimum is deleted the next one needs to be found. If there are a series of n deletions which result in the deletion of m minimums then the list needs to be at least $m + 1$ in length so that the next minimum is known.

Furthermore if the list needs to keep track of the minimums in the queue it is important to note that is does not need all of the queue elements. This is because due to the FIFO structure of a queue if the minimum is inserted any value that use to be in the list is irrelevant since it is guaranteed they will be deleted before the minimum. For example if 13, 7, and 12 were inserted the linked list would be $12- > 7$ there is no need to keep track of 13 since it will be deleted before 7. When I read the tail of the list I will find the current minimum. Additionally, when I delete twice then 12 will indeed be the next minimum. So when inserting an element in the linked list and element larger in value than it must be deleted.

The job of the queue is to keep track of the elements in FIFO order.

b) The three operations have explanations below:

<u>INSERT:</u>

Insert the element into the queue, then traverse through the linked list and delete any elements smaller than the value to be inserted. Then insert the value at the head of the list.

```
insert(q, list, value) {
    q.enqueue(value)
    curr = list.head
    while (curr != NULL && value < curr.value) {
        temp = curr
        curr = curr.next
        free(temp)
    }
    node = newNode(value)
    node.next = curr
    if (node.next == NULL) list.tail = node
    else node.next.prev = node
    list.head = node
```

```
}
```

DELETE:

Delete is simple, compare the value at the tail of the linked list with the value popped from the queue. If they are equal delete the value from the linked list:

```
delete(q, list) {
    val = queue.dequeue()
    if (val == list.tail.value) {
        temp = list.tail
        list.tail = list.tail.prev
        free(temp)
        list.tail.next = NULL
    }
}
```

FIND MINIMUM:

This is simple, just return the value at the tail of the linked list:

```
findMinimum(list) {
    return list.tail.value
}
```

c) The worst case and amortized complexity is analyzed below.

WORST CASE:

Insert: In the worst case the value inserted is the new minimum of the queue. This means that all the values in the list has to be deleted.

Inserting into the queue is $O(1)$ since enqueue just updates the tail pointer.

For the list this operation is $O(n)$ where n is the number of elements in the list, it is important to note that the number of elements in the list is not necessarily the same as the number of elements in the queue.

Thus insert in worst case is $O(n)$ where n is the number of elements in the list.

Delete: In worst case the minimum is deleted. In this case I just delete from the queue and update the tail pointer of the list. This is an $O(1)$ operation since enqueue just updates a pointer and updating the tail just updates a pointer. So overall worst case is $O(1)$.

Find Minimum: This is $O(1)$ worst case since I just return the value at the tail of the list.

AMORTIZED ANALYSIS:

Let the potential function $\phi(h) = n$ where n is the number of elements in the linked list. This is because all operations performed on the queue is $O(1)$ so the linked list is where bulk of the cost is. Note that $\phi(h) \geq 0$ since n is the size of the linked list so it is bounded from below by 0. Furthermore $\phi(h_0) = 0$ since in the initial state the linked list is empty. Thus this function satisfies all requirements for a potential function.

Insert (Aggregate):

If there are m insert operations, I know that worst case of insert is $O(n)$ where n is the number of elements greater than the value being inserted. Note that since there are m insertions at most there can only be $m-1$ deletions from the linked list. So the sequence cost is $m-1$ assuming we delete every element which was inserted except the new minimum. Thus,

$\frac{sequence\,cost}{number\,of\,operations} = \frac{m-1}{m} = O(1)$

Thus amortized cost is $O(1)$.

Insert (Potential):

The actual cost of this operation is $2 + k$ where k is the number of elements in the list that are greater than the value being inserted. This is because I also have to enqueue and insert the element into the list at the end. The size of the linked list after the value is inserted is $n - k + 1$ since you lose k elements and one element is inserted. The state of linked list before the insertion is n. Note that since this is a generalized expression there is no need to calculate the cost when the minimum is inserted and when it is not. Thus

$T_A(insert) = (2 + k) + \phi(h_{h+1}) - \phi(h_i) = 2 + k + (n - k + 1) - n = 3$

Thus this is $O(1)$.

Delete (Aggregate):

If there are m delete operations, I know that worst case of delete is $O(1)$ (see above) so:

$\frac{sequence\,cost}{number\,of\,operations} = \frac{m*O(1)}{m} = O(1)$

Thus amortized cost is $O(1)$.

Delete (Potential):

The actual cost of this operation is 2 since I have to delete from both the queue and the linked list in the worst case. The size of the linked list after the value is inserted is $n - 1$ since you lose 1 element. The state of linked list before the insertion is n. Thus

$T_A(delete) = 2 + \phi(h_{h+1}) - \phi(h_i) = 2 + (n - 1) - n = 1$

If the minimum was not deleted then cost is 1, size of lined list after operation is n, thus

$T_A(delete) = 1 + \phi(h_{h+1}) - \phi(h_i) = 1 + n - n = 1$

Thus this is $O(1)$.

Find Minimum (Aggregate):

If there are m find minimum operations, I know that worst case of find minimum is $O(1)$ (see above) so:

$\frac{sequence\,cost}{number\,of\,operations} = \frac{m*O(1)}{m} = O(1)$

Thus amortized cost is $O(1)$.

Find Minimum (Potential):

The actual cost of this operation is 1 since I just return the value at the tail pointer. The size of the linked list after the value is returned is still n. The state of linked list before the insertion is n. Note this is the same in all cases. Thus

$T_A(find\,minimum) = 1 + \phi(h_{h+1}) - \phi(h_i) = 1 + n - n = 1$

Thus this is $O(1)$.

Q2 Dynamic Array With 1.5x Expansion Analysis

I want the potential function ϕ to satisfy the following:

$\phi(h_0) = 0$

$\phi(h_i) \geq 0$

Where h_i is the ith state of the data structure. This function will keep track of the pre-charged cost of any computation. The amortized cost can be defined as $T_A(i) = c_i + \phi(h_{i+1}) - \phi(h_i)$ where c_i is the actual cost of an operation.

I will define n to be the number of elements in the array and m to be the size of the array. I want ϕ to get larger as n approaches m and be near 0 when m is increased by $\lceil 1.5n \rceil$. I know that when the array doubles $\lceil 1.5n \rceil = m$. Thus, I can use the potential function $\phi(h) = \lceil 1.5n \rceil - m$ however this will cause me to consider both an even and odd case. To skip this step I will scale the equation by a factor of 2 to get rid of the ceiling. This does not affect the results overall since the potential function is concerned with how

specific parts of a data structure is related with each other, scaling the equation does not change this since the ratio remains the same. So I will use the potential function $\phi(h) = 3n - 2m$.

This satisfies the two requirements of ϕ:

$\phi(h_0) = 3n - 2m = 3 * 0 - 2 * 0 = 0$

$\phi(h_i) = 3n - 2m \geq 0$ Since I know that at all times $3n \geq 2m$

Now I will consider two cases:

$\underline{n < m}$:

In this case the actual cost is 1 since the array has not increased in size. n increases by 1 and m stays the same since the array has not increased in size. So by the formula T_A,

$T_A(i) = c_i + \phi(h_{i+1}) - \phi(h_i) = 1 + (3(n+1) - 2m) - (3n - 2m) = 1 + 3 = 4$

So the amortized time in this case is 4.

$\underline{n = m}$:

In this case the actual cost is $n + 1$ since the array will increase in size. n increases by 1 and $m = 1.5n$. So by the formula T_A,

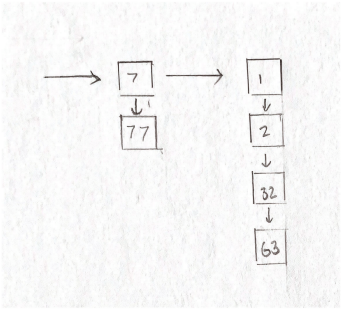
$T_A(i) = c_i + \phi(h_{i+1}) - \phi(h_i) = n + 1 + (3(n+1) - 2m) - (3n - 2m) = n + 1 + (3(n+1) - 2(1.5n)) - (3n - 2n) = 1 + 3 = 4$

So the amortized time in this case is 4.

Thus, in both cases the amortized complexity is $O(1)$. Therefore the amortized complexity of this variation is $O(1)$.

Q3 Set Data Structure Analysis

a) Draw the data structure that results after inserting the following sequence of elements into an initially empty set: 2, 63, 1, 32, 77, 7



b) Let I_n denote the set of bit positions in the binary representation of n that contain the value 1. Specifically, I_n contains the positions where a 1 occurs in the binary representation of n. The 0th position being the right most bit in a binary number. The size of I_n indicates the number of sorted arrays in a set of size n. This is because each 1 bit in the binary representation of n relates to a power of 2 needed to sum to n. Furthermore, each position where a 1 occurs indicates the size of each array.

To iterate this better consider an example where $n = 5$. The binary representation of 5 is 101, which means that the set I_5 is $\{0, 2\}$ where each element in I_5 is the position where a 1 occurred in 5. The size of I_5 is 2 which means that a set of 5 will contain two sorted arrays. Furthermore since the bits are in position 0 and

2 one array in the set will have $2^0 = 1$ elements and the other will have $2^2 = 4$ elements. So in general, the set I_n can give two pieces of information: the size will indicate the number of arrays in a set of size n, and 2 raised to each element in I_n indicates the number of elements in one of the arrays in the set.

In the worst case binary search must be performed in each array in the set. I know that binary search has a worst case complexity of $log(n)$. I will denote $|I_n|$ to be the size of I_n. Furthermore, I will denote $I_n[i]$ to be the ith element in I_n with indices starting at 1. The worst case complexity of search in a set of size n is as follows:

$\sum_{i=1}^{|I_n|} log(2^{I_n[i]}) = \sum_{i=1}^{|I_n|} I_n[i]$

This is because the arrays in the set are of size $2^{I_n[i]}$ so the complexity of binary search on any of the arrays is $log(2^{I_n[i]})$ (binary search worst case is $log(n)$). Since in the worst case I have to perform binary search on all of the arrays the complexity becomes $\sum_{i=1}^{|I_n|} log(2^{I_n[i]})$.

Thus the worst case complexity for search is the sum of the elements in I_n.

c) It is important to note that in a set of size n the worst insertion possible depends on the number of merges. In the worst case the algorithm will merge all the arrays. A set of size n can have at most $log(n)$ arrays which means that $log(n)$ merges are possible (log being rounded up to nearest integer). Note that this is an extreme exaggeration but it gives the worst case.

The worst case complexity is $\sum_{i=0}^{log(n)} 2^{i+1} - 1$. This is because to merge two arrays of size 2^i I need to create an array 2^{i+1} in size and I need to compare values $2^{i+1} - 1$ times to fill each spot (last spot is given since there is nothing to compare to). Evaluating the expression,

$\sum_{i=0}^{log(n)} 2^{i+1} - 1 = 2^{log(n)+2} - 2 - n = 4n - 2 - n = 3n - 2$

Therefore in worst case insertion is $O(n)$.

d) It is not reasonable to assume that each insertion costs $O(n)$. So the question becomes when do I merge and how often do I merge arrays of specific sizes? To determine this it is important to look at the binary representation of n. As a binary number increments the bit at position 0 takes two increments to becomes 0 (i.e. initially 0 then increment to 1 then increment to 10), the bit at position 1 takes 4 increments to become 0 (i.e initially 00 then 01 then 10, then 11, then 100). The general rule is the bit at position i takes 2^{i+1} increments to go from 0 to 1 and back to 0. This is important because every time a bit changes from 0 to 1 and back to 0 the algorithm performs a merge. Specifically the merge will cost $2^{i+1} - 1$ (as seen in part c) since when I want to merge two arrays of size 2^i to size 2^{i+1} there are $2^{i+1} - 1$ comparisons. This is important since it means that a merge that costs $2^{i+1} - 1$ occurs every 2^{i+1} increments.

How many times does an increment of 2^{i+1} occur in n insertions? It occurs $n/2^{i+1}$ times. Thus the sequence cost of the merges of n insertions is $\sum_{i=1}^{log(n)} \frac{n}{2^i}(2^i - 1)$, note that i starts at 1. However, I still have to take into account the insertions that do not cause merges, there are $n/2$ of them and since their cost is 1 the cost is overall $n/2$. Evaluating the expression,

$n/2 + \sum_{i=1}^{log(n)} \frac{n}{2^i}(2^i - 1) = n/2 + n\sum_{i=1}^{log(n)} \frac{2^i - 1}{2^i} < nlog(n) + n\sum_{i=1}^{log(n)} \frac{2^i}{2^i} = nlog(n) + n\sum_{i=1}^{log(n)} 1 = 2nlog(n)$

Thus the sequence cost is $O(nlog(n))$.

Following the aggregate method I can divide sequence cost by number of operations and get the following: $nlog(n)/n = log(n)$.

Thus the amortized cost by the aggregate method is $log(n)$.

e) I will add an additional charge upon insertion and charge nothing for the merging. First I need to determine a charge for insertion.

The sequence cost of the merges of n insertions is $\sum_{i=1}^{log(n)} \frac{n}{2^i}(2^i - 1)$, note the explanation below has been copied from part d.

It is not reasonable to assume that each insertion costs $O(n)$. So the question becomes when do I merge and how often do I merge arrays of specific sizes? To determine this it is important to look at the binary representation of n. As a binary number increments the bit at position 0 takes two increments to becomes 0 (i.e. initially 0 then increment to 1 then increment to 10), the bit at position 1 takes 4 increments to

become 0 (i.e initially 00 then 01 then 10, then 11, then 100). The general rule is the bit at position i takes $2^(i+1)$ increments to go from 0 to 1 and back to 0. This is important because every time a bit changes from 0 to 1 and back to 0 the algorithm performs a merge. Specifically the merge will cost $2^{i+1}-1$ (as seen in part c) since when I want to merge two arrays of size 2^i to size 2^{i+1} there are $2^{i+1}-1$ comparisons. This is important since it means that a merge that costs $2^{i+1}-1$ occurs every 2^{i+1} increments.

How many times does an increment of 2^{i+1} occur in n insertions? It occurs $n/2^{i+1}$ times. Thus the sequence cost of the merges of n insertions is $\sum_{i=1}^{log(n)} \frac{n}{2^i}(2^i-1)$, note that i starts at 1. Evaluating this expression,

$\sum_{i=1}^{log(n)} \frac{n}{2^i}(2^i-1) = n\sum_{i=1}^{log(n)} \frac{2^i-1}{2^i} < n\sum_{i=1}^{log(n)} 1 = nlog(n)$.

So I know merges will require an additional $nlog(n)$ in terms of cost. This means that over n insertions I must build up a credit of $nlog(n)$ to handle any merges that occur. Additionally, I know the cost of an insertion without a merge is 1. Thus I will let the total charge be $log(n)+1$ to handle n insertions.

Therefore the amortized cost is $log(n)$ since the charge $log(n)+1 \in O(log(n))$.

Q4 Building Disjoint Set From Undirected Graph

Iteration 0:

$\{l\}$ $\{m\}$ $\{n\}$ $\{o\}$ $\{p\}$ $\{q\}$ $\{r\}$ $\{s\}$ $\{t\}$ $\{u\}$ $\{v\}$

Iteration 1:

$\{l\}$ $\{m\}$ $\{n\}$ $\{o,t\}$ $\{p\}$ $\{q\}$ $\{r\}$ $\{s\}$ $\{u\}$ $\{v\}$

Iteration 2:

$\{l\}$ $\{m\}$ $\{n\}$ $\{o,t\}$ $\{p\}$ $\{q,v\}$ $\{r\}$ $\{s\}$ $\{u\}$

Iteration 3:

$\{l\}$ $\{m\}$ $\{n\}$ $\{o,t,r\}$ $\{p\}$ $\{q,v\}$ $\{s\}$ $\{u\}$

Iteration 4:

$\{l\}$ $\{n\}$ $\{o,t,r,m\}$ $\{p\}$ $\{q,v\}$ $\{s\}$ $\{u\}$

Iteration 5:

$\{l,s\}$ $\{n\}$ $\{o,t,r,m\}$ $\{p\}$ $\{q,v\}$ $\{u\}$

Iteration 6:

$\{l,s\}$ $\{n\}$ $\{o,t,r,m,u\}$ $\{p\}$ $\{q,v\}$

Iteration 7:

$\{l,s\}$ $\{n\}$ $\{o,t,r,m,u,q,v\}$ $\{p\}$

Iteration 8 (no change):

$\{l,s\}$ $\{n\}$ $\{o,t,r,m,u,q,v\}$ $\{p\}$

Iteration 9 (no change):

$\{l,s\}$ $\{n\}$ $\{o,t,r,m,u,q,v\}$ $\{p\}$

Iteration 10 (no change):

$\{l,s\}$ $\{n\}$ $\{o,t,r,m,u,q,v\}$ $\{p\}$

Iteration 11:

$\{l,s,p\}$ $\{n\}$ $\{o,t,r,m,u,q,v\}$

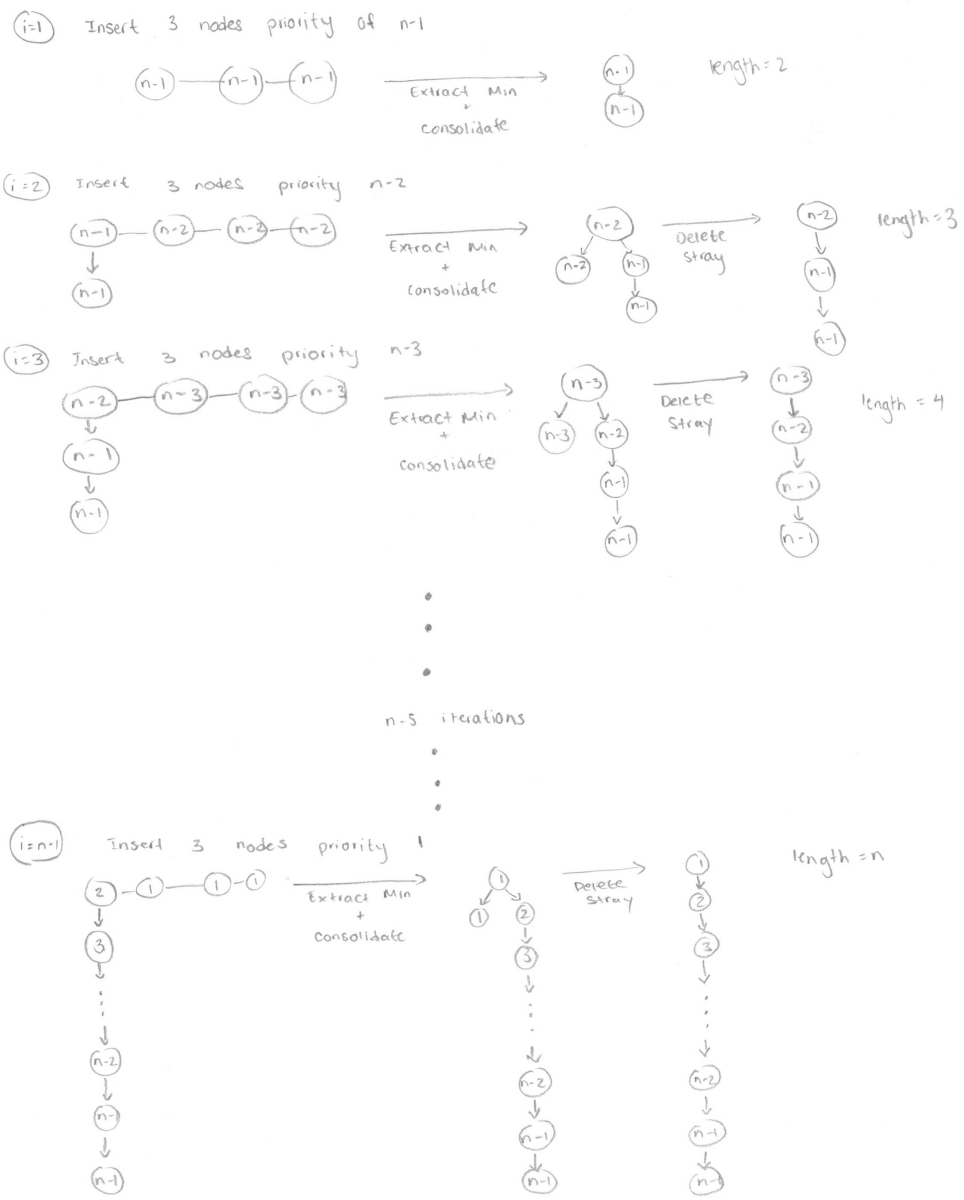
Q5 Disprove: The height of n node Fibonacci heap is always $O(log(n))$

a) Consider the following algorithm:

Insert 3 nodes into the heap, all with priority $n-i$, where i is a natural number less than n and refers to the ith iteration of this process. Perform extract min and consolidate. Delete the stray node that is not part of the chain (note this step is only applicable after the first iteration). This will result in a linear chain of length $i+1$. After this process is repeated $n-1$ times the Fibonacci heap will be a linear chain of n nodes.

This works since as nodes are inserting in decreasing priority the minimum will always change. Since the minimum changes when consolidate is performed the chain from the previous iteration will become a child of the new minimum resulting in the chain growing by one for each iteration.

Consider the illustration for a visual explanation:



b) Pseudocode for algorithm:

```
chainFibonacci(n):
    h = Heap()
    for i in range(1, n):
        H.insert(n-i)
        H.insert(n-i)
        H.insert(n-i)
        H.extractMin()
        if i > 1: H.delete(n-i, min=False) # second parameter to indicate not to touch the min
```

Q6 Modified Tree Implemented Disjoint Set To Retrieve All Set Members Linear Time

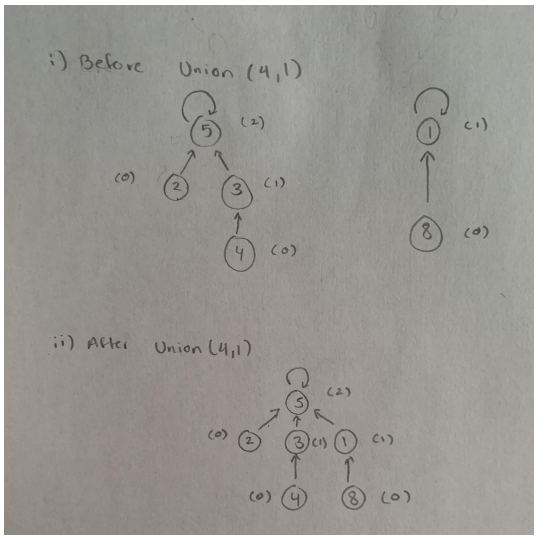
Consider adding a next pointer to each node in the set. The next pointer in each node will point to another node in the set and form a circular linked list. This would work since to get all the nodes in the set you can traverse the circular linked list and return the members of the set. Since the order is not important this pointer can be updated when a union occurs by swapping the next pointers of the two representatives. As a result a circular linked list structure will be retained. For example if the node is x and y are representatives of two sets, then if I want to union their sets all I have to do is swap their next pointers. This causes x.next to point to the beginning of y's list and y.next to point to node which x just disconnected from. Since I am just updating pointers this is an $O(1)$ operation.

This pointer will not affect any of the operations in the set since it takes constant time to manage the pointer. An explanation can be seen below.

A set has three main operations find, make-set, and union. The next pointer will not affect the find operation at all since the next pointer is not involved in this operation. All this operation does is find the set a node is part of so you only need to return the representative, since the next pointer is not involved in this operation it will not affect it. For the make-set, all I have to do is set the next pointer of the set to point to the representative node itself. This is an $O(1)$ operation so it will not affect the complexity of make-set. For the union operation the process was explained in the first paragraph, but to reiterate all I have to do is swap the next pointers of the two representatives so the operation will be completed in constant time. Thus this will not affect the union operation either.

Q7 Disjoint Set Analysis

a) Draw tree representation after certain operations (see handout)



b) It is possible to have $TT < TL$. Consider the following set of operations: $P_1 = make\text{-}set(1)$, $P_2 = make\text{-}set(2)$, $P_3 = make\text{-}set(3)$, $P_4 = make\text{-}set(4)$, $P_5 = find\text{-}set(1)$ $P_6 = union(1,2)$, $P_7 = union(3,4)$, $P_8 = union(1,3)$. So $P = \{P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8\}$.

In a tree representation of a disjoint set operations P_1 through P_4 will each result in one node being accessed. Similarly P_5 will only result in one node being accessed since the set only has one node. In the case of P_6 through P_8 each operation will result in two nodes being accessed since I just need to compare the ranks of trees and append the smaller tree to the larger one. As a result in total 11 nodes will be accessed so $TT = 11$.

In the case of the linked list representation of a disjoint set the operations P_1 through P_4 will each result in one node being accessed. Similarly P_5 will only result in one node being accessed since the set only has one node. Operations P_6 and P_7 will each result in two nodes being accessed since all sets are of size one so the union will not require the update of any additional representative pointers. The operations will only update the representative pointer of one of the nodes to attach it to the set of the other. In operation P_8 there will be 3 nodes accessed. This is because since both sets are of length 2 when we union one set to the other after comparing the size of the sets from the representatives, two representative pointers must be updated. SO in total we have 3 nodes accessed, 2 representatives, and one which belongs to the set of the representative which will become part of the other set. In total this causes 12 nodes to be accessed. As a result $TL = 12$.

From this set of operations I have $TL = 12$ and $TT = 11$, clearly $TT < TL$. Therefore it is possible to have $TT < TL$.

c) It is possible to have $TL < TT$. Consider the following set of operations: $P_1 = make\text{-}set(1)$, $P_2 = make\text{-}set(2)$, $P_3 = make\text{-}set(3)$, $P_4 = union(1,2)$, $P_5 = union(1,3)$, $P_6 = find\text{-}set(3)$. So $P = \{P_1, P_2, P_3, P_4, P_5, P_6\}$.

In a tree representation of a disjoint set each make-set operation results in one node being accessed. In the case of P_4 and P_5 two nodes will be accessed for each operation since after examining rank we just need to connect the smaller tree to the larger tree. Thus I only need to look at the head of each tree. In the case of P_6 3 nodes need to be accessed. Since this is the first find operation the tree has not had a chance to perform path compression, as a result all the nodes must be traversed to the root which is 1. As a result in total the tree representation will have 10 nodes accessed. So $TT = 10$.

In the linked list representation of a disjoint set each make-set operation results in one node being accessed. In the case of P_4 only two nodes will be accessed since 1 and 2 are both sets of length 1 currently. P_5 will also only require two nodes to be accessed since in this case after 1 and 3 are compared by length, since 3 is the smaller set it will be appended to 1's set. Furthermore, since 3's set only contains one element only one representative pointer must be updated, so in total two nodes are accessed. In the case of P_6 only 2 nodes need to be accessed. Since this is the linked list representation as soon as we call find-set on 3 wee return 3.rep() which will give 1. So there is no need for 2 to be visited which is different compared to the tree representation. As a result in total the linked list representation will have 9 nodes accessed. So $TL = 9$.

From this set of operations I have $TL = 9$ and $TT = 10$, clearly $TL < TT$. Therefore it is possible to have $TL < TT$.

Q8 Expected Value of Variable Analysis

The loop ends when $c = n$, in order for c to increase the expression $random(1, 2^c) == 1$ must be true (note the == indicates equality). In order to find the expected value to t I need to know how many attempts it will take on average for $random(1, 2^c) == 1$ to be true.

Since this is a randomized algorithm the probability of $random(1, 2^c)$ being any value in the range $[1, 2^c]$ is $1/2^c$ since there are 2^c possible values. So the probability that the expression $random(1, 2^c) == 1$ is true is $1/2^c$. Since the probability is $1/2^c$ this means that it will take 2^c attempts on average for the expression $random(1, 2^c) == 1$ to become true. This implies that on average t will increase by one 2^c times before c is increased.

Using this I can determine the value of t at the end of the randomized algorithm with the following expression: $\sum_{c=1}^{n} 2^c$. This is because t increases 2^c times each time c is incremented by 1 and c will go from 1 to n. I can evaluate the expression,

$\sum_{c=1}^{n} 2^c = 2^{n+1} - 2$

Therefore the expected final value of t is $2^{n+1} - 2$.

Q9 Binary Tree Search Expected Value

a) Assume (without loss of generality) that the 7 keys in the tree are 1, 2, 3, 4, 5, 6, 7, where 4 is at the root, 2 and 6 are at depth 1, and 1, 3, 5, 7 are at depth 2.

$search1(root, 1)$ results in 5 comparisons.

$search1(root, 2)$ results in 3 comparisons.

$search1(root, 3)$ results in 5 comparisons.

$search1(root, 4)$ results in 1 comparisons.

$search1(root, 5)$ results in 5 comparisons.

$search1(root, 6)$ results in 3 comparisons.

$search1(root, 7)$ results in 5 comparisons.

Since I have a random choice of k the probability of choosing any of 1 through 7 is 1/7. I have 3 distinct values for comparisons (1,3,5). The probability of having 1 comparison is 1/7, the probability of having 3 comparisons is 2/7, and the probability of having 5 comparisons is 4/7. Let X be the random variable such that X represents the probability of having n comparisons. Then the only possible values for X is 1, 3, and 5.

The expected value of X can be calculated as $\sum_x xP(X = x)$, where x iterates over all possible values of X.

Solving the expression,

$\sum_x xP(X = x) = 1 * 1/7 + 3 * 2/7 + 5 * 4/7 = 27/7$

Thus the expected value of X is 27/7 and as a result the expected number of comparisons when doing $search1$ is 27/7.

b) Assume (without loss of generality) that the 7 keys in the tree are 1, 2, 3, 4, 5, 6, 7, where 4 is at the root, 2 and 6 are at depth 1, and 1, 3, 5, 7 are at depth 2.

$search2(root, 1)$ results in 6 comparisons.

$search2(root, 2)$ results in 4 comparisons.

$search2(root, 3)$ results in 5 comparisons.

$search2(root, 4)$ results in 2 comparisons.

$search2(root, 5)$ results in 5 comparisons.

$search2(root, 6)$ results in 3 comparisons.

$search2(root, 7)$ results in 4 comparisons.

Since I have a random choice of k the probability of choosing any of 1 through 7 is 1/7. I have 5 distinct values for comparisons (2,3,4,5,6). The probability of having 2 comparisons is 1/7, the probability of having 3 comparisons is 1/7, the probability of having 4 comparisons is 2/7, the probability of having 5 comparisons is 2/7, and the probability of having 6 comparisons is 1/7. Let X be the random variable such that X represents the probability of having n comparisons. Then the only possible values for X is 2, 3, 4, 5, and 6.

The expected value of X can be calculated as $\sum_x xP(X = x)$, where x iterates over all possible values of X.

Solving the expression,

$\sum_x xP(X = x) = 2 * 1/7 + 3 * 1/7 + 4 * 2/7 + 5 * 2/7 + 6 * 1/7 = 29/7$

Thus the expected value of X is 29/7 and as a result the expected number of comparisons when doing $search2$ is 29/7.

c) In $search1$ the algorithm performs 2 comparisons to search left, 2 comparisons to search right and 1 comparison to return a node. In $search2$ the algorithm performs 2 comparisons to search left, 1 comparison to search right and 2 comparisons to return a node.

Thus I can express the number of comparisons in the search algorithms with the a formula. Let l_n denote the number of left traversals when searching for key n and r_n denote the number of right traversals when searching for a key n. Then the number of comparisons by each algorithm is as follows:

$search1$: $1 + 2l_n + 2r_n$

$search2$: $2 + 2l_n + r_n$

Note that there is no variable when referring to returning a node, this is because the algorithm can return a node only once. I can consider cases to see which formula will perform better.

$\underline{r_n = 0}$:

When r_n is 0 that means there are only left traversals to get the key n. This case applies to 11 keys since only 11 keys are along the leftmost chain in a tree with height 10. This formulas now become

$search1$: $1 + 2l_n$

$search2$: $2 + 2l_n$

Clearly in this case $search1$ always performs better by 1 comparison.

$\underline{l_n = 0}$:

When l_n is 0 that means there are only right traversals to get the key n. This case applies to 10 keys since only 10 keys are along the rightmost chain in a tree with height 10 (not including the root). This formulas now become

$search1$: $1 + 2r_n$

$search2$: $2 + r_n$

Clearly in this case $search2$ always performs at least as good as $search1$ or better. For this case when $r_n = 1$ then $search1$ and $search2$ perform the same however when $r_n > 1$ $search2$ performs better than $search1$.

$\underline{r_n \neq 0 \text{ and } l_n \neq 0}$:

When l_n and r_n are both non-zero this means the algorithms are searching for inner nodes. I have the following expressions:

$search1$: $1 + 2l_n + 2r_n$

$search2$: $2 + 2l_n + r_n$

Clearly for every left traversal both algorithms perform the same. So this can be ignored as it will not play a difference in how these algorithms perform with respect to each other. Thus I get the following,

$search1$: $1 + 2r_n$

$search2$: $2 + r_n$

So I am only interested in how $1 + 2r_n$ and $2 + r_n$ perform with respect to each other.

When graphing these two equations the point of intersection is at $r_n = 1$. For every $r_n > 1$ the equation $1 + 2r_n$ is greater than $2 + r_n$. Thus for any tree traversal to an inner node which requires one right traversal the algorithms perform the same, and for any which require more than one right traversal $search2$ performs better.

It is important to note that to get to any inner node at least one right traversal is required. In a perfect binary tree of height 10 there are $2047 - 21 = 2026$ inner nodes. So in this case for all 2026 nodes $search2$ will before at least as well or better than $search1$.

Many of these nodes will require more than one right traversal, for example if I look at the tree of the right child of the root's right child there are roughly 511 and nodes ((2047-3)/4), if I ignore the ones which are

part of the rightmost chain then there are 503 nodes (511-8). So in total there are at least 503 inner nodes that $search2$ performs better than $search1$.

So in this case for all 2026 nodes $search2$ will perform at least as well or better than $search1$ and I know that for at least 503 of these nodes $search2$ performs better than $search1$.

From the three cases above I know that in a perfect binary tree with 2047 nodes except for 11 nodes, $search2$ will perform as well as or better than $search2$ at finding 2035 nodes in terms of comparisons. Furthermore I know that there is a minimum of 512 nodes (sum of case 2 and 3) where $search2$ will definitely perform better than $search1$ in terms of comparisons. Therefore I can conclude that $search2$ will have a lower expected value than $search1$ in terms of comparisons.

Q10 Sequence Data Structure Analysis

a) When n=2 there are 4 possible combinations,

[PREPEND, PREPEND] with probability p^2

[PREPEND, ACCESS] with probability $p * (1 - p)$

[ACCESS, PREPEND] with probability $p * (1 - p)$

[ACCESS, ACCESS] with probability $(1 - p)^2$

b) Let X be the random variable such that $X = 1$ if the operation is PREPEND and $X = 0$ if the operation is ACCESS. Then $P(X = 1) = p$ and $P(X = 0) = 1 - p$.

To get the expected value of the length of the sequence after k operations I need to know what is the expected number of times for PREPEND to be called in a sequence of k operations.

If I had $k = 1$ then this would match the Bernoulli distribution. However since k is an unknown value and the probabilities of each trial is independent, this problem matches the Binomial distribution.

The expected value of a Binomial distribution is calculated by $number\ of\ events * probability\ of\ successful\ event$. The probability of the PREPEND operation occurring is p and since I have k trials, $number\ of\ events = k$. Therefore the expected length is kp.

c) On the kth operation there are two possibilities, either PREPEND is performed or ACCESS is performed. Since expected value is defined as $\sum_x xP(X = x)$ where x iterates over all possible values for X, the expected value of kth operation is $steps\ during\ PREPEND * probability\ of\ PREPEND + steps\ during\ ACCESS * probability\ of\ ACCESS$.

$steps\ during\ PREPEND * probability\ of\ PREPEND$:

This half is quite simple, PREPEND always requires one step so the expression is simply $probability\ of\ PREPEND$ which is p.

$steps\ during\ ACCESS * probability\ of\ ACCESS$:

This half is more complicated. I know that $probability\ of\ ACCESS$ is $1 - p$. However $steps\ during\ ACCESS$ is dependent on the value chosen to access (it could be the first or the last).

It is given that for each ACCESS operation, the value of the parameter i is chosen uniformly from $[1 \dots |S|]$. Depending on the i chosen ACCESS will take i steps. The expected value of a uniform distribution is calculated by $\frac{u+l}{2}$ where u and l are bounds on the range. In this case the expected value is $kp/2$ since there are expected to be kp values in the list from part b.

Thus $steps\ during\ ACCESS * probability\ of\ ACCESS$ is $(1 - p)(kp/2)$.

Therefore on the kth step the expected number of steps is $p + (1 - p)(kp/2)$.

d) I can use the formula from part c and calculate the sum of the expected values of each step since the steps are independent.

Thus the expected value can be calculated by $\sum_{k=1}^{n} p + (1 - p)(kp/2)$. Solving the expression below,

$\sum_{k=1}^{n} p + (1-p)(kp/2) = \sum_{k=1}^{n} p + \sum_{k=1}^{n}(1-p)(kp/2) = p\sum_{k=1}^{n} 1 + (1-p)(p/2)\sum_{k=1}^{n} k =$
$np + (1-p)(p/2)(n(n+1)/2)$

I can simplify the expression: $np + (1-p)(p/2)(n(n+1)/2) = np[1 + (1-p)(n+1)/4]$.

Therefore over all n steps the expected value is $np[1 + (1-p)(n+1)/4]$.

Q11 Modified Quicksort to Find k Smallest Element

a) The algorithm below performs quicksort but throws away one partition each time.

```
Smallest(k, S):
    pivot = S[random(1, S.size())]
    s1 = new Set()
    s2 = new Set()
    for element in S:
        if element <= pivot: s1.append(element)
        else: s2.append(element)
    if k < s1.size(): return Smallest(k, s1)
    elif k = s1.size() + 1: return pivot
    else: return Smallest(k-s1.size(), s2)
```

b) The worst case for this algorithm is the same as the worst case for a regular quicksort since the pivot can always be the max element and k can be 1. This results in $\sum_{i=1}^{S.size()} i$ iterations of the for loop. The summation evaluates to $S.size()(S.size()+1)/2$ which if S has n elements then this has a complexity of $O(n^2)$.

Therefore the worst case complexity is $O(n^2)$.

c) In the average case it is not reasonable to assume that the largest value in S is chosen as the pivot. The algorithm complexity depends on the size of the set passed into the function.

On average the two sets s1 and s2 will be relatively close in size since it is more likely to pick a pivot which is a middle value then one which is extremely small or large. I could assume that the arrays are split by 2 each time, however for the sake of calculations I will assume that the partitions are a $\frac{1}{3}/\frac{2}{3}$ split and k lies on the 2/3 portion of the partition. Note that it is an overestimation to assume that it is a $\frac{1}{3}/\frac{2}{3}$ split and k is always in the larger portion since as stated earlier on average the arrays will get closer to a $\frac{1}{2}/\frac{1}{2}$ split. I chose this split since all I know is that pivot is most likely to be a middle value not necessarily always a perfect $\frac{1}{2}/\frac{1}{2}$ pivot. Thus by choosing $\frac{1}{3}/\frac{2}{3}$ I am taking this into account. In this case the for loop executes n times on the first iteration, $2/3 * n$ times in the second iteration, $4/9 * n$ times in the third iteration and so on. So the expression is,

$\sum_{i=0}^{n}(\frac{2}{3})^i n = n(3 - 2(2/3)^n) < 3n$

Therefore the algorithm is $O(n)$.

Q12 Determine if Element Appears in Array more than n/2 Times

a) Use a similar algorithm to the modified quicksort from Q11. Pick a pivot at random and create three partitions one with less than values, one with equal values and the other with greater values. Once the partitions are created the algorithm does the following,

1. It checks the equal partition in length. If the equal partition is less than or equal to n/2 in length it is discarded, otherwise return true.

2. If the equal partition was discarded check the less than and the greater than partition. If any of these are less than or equal n/2 in length then discard them. If there are no partitions left return false, otherwise run the modified quicksort again.

This algorithm will continue until either the partitions are all less than or equal n/2 in length or the equal partition contains the majority element. The idea here is that if there is a majority element then it will be most likely to be picked as a pivot thus it is most likely for the equal portion to contain the majority element.

Pseudocode:

```
Majority(array, halfLength):
    pivot = array[random(1, array.length())]
    a1 = new Array()
    a2 = new Array()
    a3 = new Array()
    for element in array:
        if element < pivot: a1.append(element)
        if element = pivot: a2.append(element)
        else: a3.append(element)
    if a2.length() > halfLength: return True
    else:
        if a1.length() <= halfLength and a3.length() <= halfLength: return False
        elif a1.length() <= halfLength: return Majority(a3, halfLength)
        // at this point a3 must be less than or equal to half length
        else: return Majority(a1, halfLength)
```

b) There are two cases to consider, if the majority element exists and if the majority element does not exists.

Majority Element Exists:

In the average case it is not reasonable to assume that the largest value in the array is chosen as the pivot (assuming it is not the pivot). In fact in the average case if the majority exists it is most likely for it to be picked as the pivot since the pivot picking is random and it makes up most of the array. Meaning that if a majority element exists then in the average case this algorithm will run only once since the equal array will be greater than half length. As a result if the majority element exists this is definitely an $O(n)$ algorithm in the average case.

Majority Element Does Not Exists:

If the majority element does not exist the worst type of array this algorithm can receive is one with all distinct elements since there is no longer any benefit of the equal partition. However in this case this algorithm performs similarly to the algorithm from Q11 and it has the added benefit that it stops when the partitions are less than halfLength. So, on average the two arrays a1 and a3 will be relatively close in size since it is more likely to pick a pivot which is a middle value then one which is extremely small or large. I can assume again that the partitions are a $\frac{1}{3}/\frac{2}{3}$ split. In this case the for loop executes n times on the first iteration, and $2/3 * n$ times in the second iteration. There is no more iterations after this since picking another pivot will most likely result in the partitions being less than halfLength. So the expression is, $n + \frac{2}{3}n$.

Therefore the algorithm is $O(n)$ on average.

Q13 Industrial Power Grid Database Application

The combination of data structures I will use is as follows:

- Database: Hash table implemented as array to hold linked lists for the actual database (each linked list stores the businesses of a grid, insert at head)

- Capacity Manager: Max Heap to manage the capacity of the grids
- Business Lookup: Hash table implemented as a dynamic array to store a master list of all businesses (array indices are calculated by hashing)

Data Structure Specification:

Database:

Since I know that there are n grids, the array will have n indices. The way to calculate the indices of a specific grid will be through hashing. Note that the method of hashing will be close addressing. Once the key s_i is given a hash function will be executed to determine the index of the grid.

Furthermore at each index will be a pointer to a linked list which will hold all the businesses of a specific grid.

Capacity Manager:

A max heap will hold the grids, the priority is determined by the capacity of the grid. Each node in the max heap contains the following: the priority which is the grid capacity, and the s_i value of that grid.

Business Lookup:

A dynamic array will be used to keep track of all the businesses. The location of a business will be determined by using a hashing algorithm. Note that the method of hashing will be close addressing. Each node in the hash table will include the following: a pointer to the node that holds the business in the Database, and the capacity required to hold the business, the pointer to its grid's node in the max heap. Additionally, the Business Lookup will also keep track of the max capacity and the total number of elements in the dynamic array.

Operations Explanation:

initialize

When initialize is called the three data structures must be set up.

For the Database data structure, create an array of size n and then insert the power grids into the array using the hashing function on the s_i. For each insertion also initialize an empty linked list.

For the Capacity Manager, create the max heap by making an array and store the nodes into the array. Then call build heap to convert the array into a max heap.

For the Business Lookup just initialize an empty dynamic array.

power_on

When power_on is called look at the grid with the largest capacity from the max heap. Use the s_i to find the grid in the Database and then insert the business into the linked list of the grid. Then, insert the business into the Business Lookup, set a pointer to the node of the grid's linked list which holds the business, also set a pointer to the node in the max heap and update the counter for the number of elements in the array. If the number of elements is equal to the max capacity the array must be doubled in size and the elements copied over. Lastly, once the business is inserted update the max heap by adjusting the capacity of the grid and calling heapify.

power_off

For power off search for the business in the hash table from the Business Lookup. Once it is found then delete the item from the linked list by following the pointer. Now update the capacity of the grid in the max heap by following the pointer, then call percolate on the heap. Finally remove the business from the Business Lookup and decrement the counter for the number of elements in the heap.

businesses

Use the s_i to find the grid in the hash table, then traverse its linked list and return the names of all the businesses.

Complexity Analysis:

initialize

I will consider the worst case complexity to set up each of the data structures.

For the Database data structure, I first create an array of size. Then I repeat the following process n times: create a node which contains the s_i for the business and a NULL pointer for the linked list, use the hashing algorithm to insert the node into the array. This is clearly $O(n)$ since it is constant work to create a node all I have to do is set the NULL pointer and add the name to the node. It is also constant time to use a hashing function. So n constant work operations are $O(n)$ worst case.

For the Capacity Manager I create an array of size n and insert n nodes for each of the grids into it. Then I call build heap. To create n nodes for the heap and insert them into the array will be $O(n)$. This is because at this point I am not concerned about the index so I can use a for loop and insert each node into the fist empty slot. When I call build heap the array will be converted to a max heap. Per lecture we learned that calling build heap on an array has a complexity of $O(n)$. So overall I have two separate operations that cost $O(n)$. $O(n) + O(n) \in O(n)$, thus worst case this is $O(n)$.

For the Business Manager all I have to do is initialize an empty dynamic array and initialize two counters. This is $O(1)$.

So I have two operations which are $O(n)$ worst case and one which is $O(1)$ worst case thus overall initialize is $O(n)$ worst case.

power_on

For power_on I will consider the average complexity.

First I have to find the grid with the largest capacity which I can find by looking at the first element of the Capacity Manager. If the element cannot support the business I know no other can so it is safe to return. If it can I will use its s_i to find the grid in the Database and insert the business into the linked list. To search for the grid after given s_i is $O(1)$. This is because per lecture we learned that the complexity of search on average during closed addressing is the load factor which is $number\ of\ elements\ /\ number\ of\ buckets$. Since the size of the hash table is n and the number of elements is n I have $n/n = 1$ which means that on average a search will be $O(1)$. It is also $O(1)$ to insert into the linked list is $O(1)$ as well since I just insert at the head.

Next I insert the business into the Business Lookup. Inserting into a hash table is $O(1)$ with closed addressing so this will also be $O(1)$.

Lastly I need to update the max heap. Since I know that the grid is the max I just update the first node in the heap and call heapify. Since heapify is $log(n)$ in worst case complexity I know it will be at least as good as $log(n)$ in average case too. So in average case I can say this operation is $log(n)$.

Thus I have 3 operations, $O(1) + O(1) + O(log(n)) \in O(log(n))$. Therefore power_on is $O(log(n))$ on average.

power_off

For power_off I will consider the average case complexity.

To search for the business in the Business Lookup is $O(1)$. This is because per lecture we learned that the complexity of search during closed addressing on average is the load factor which is $number\ of\ elements\ /\ number\ of\ buckets$. Since I am using a dynamic array (implemented with a regular array) as the hash table, the hash table doubles in size every time the table becomes full. This means that I can guarantee that the $number\ of\ elements \leq number\ of\ buckets$ which means that the load factor < 1. Since this is the case the search will be $O(1)$ on average.

Now I can use the node at the Business Lookup to remove the node from the linked list by following its pointer. This is $O(1)$ worst case since if I where the element is in the linked list all I have to do is adjust two pointers to remove it from the list. This means on average to update the Database it will cost $O(1)$.

After this I can now update the max heap capacity. I can follow the pointer from the Business Lookup to the node in the max heap and update the capacity. Then I can call percolate which I know is $log(n)$ worst

case. This means on average to update the Capacity Manager it will cost $log(n)$.

Lastly I can delete the entry from the Business Lookup which is $O(1)$.

So I have 3 operations with average case $O(1)$ and one which is $O(log(n))$ on average. Thus $3O(1) + O(log(n)) \in O(log(n))$. Therefore power_off is $O(log(n))$ on average case.

businesses

I will consider average case for businesses.

To search for the grid after given s_i is $O(1)$. This is because per lecture we learned that the complexity of search on average during closed addressing is the load factor which is *number of elements / number of buckets*. Since the size of the hash table is n and the number of elements is n I have $n/n = 1$ which means that on average a search will be $O(1)$.

Once the grid is found I have to traverse the linked list. I know that worst case a linked list traversal is proportional to the number of elements so it is the same for the average case. Thus on average the linked list traversal to get all the business names in $O(k)$ where k is the number of businesses.

So on average I have $O(1) + O(k) \in O(k)$. Thus average complexity for businesses is $O(k)$.

Q14 Hashing: Chaining vs Linear Probing

a) For any insert on $T1$ there are no comparisons since after calculating the hash you just insert to the front of the list. Additionally, assuming that you know where the deletion should take place there are also no comparisons for $T1$. So overall, if the operations are just insertions and deletions $T1$ performs no comparisons.

$T2$ is similar to $T1$ for deletion. If you know where to delete then there are no comparisons. However, for insertion with linear probing you cannot always insert right away. If two elements hash to the same value you have to follow the probe sequence and then insert into the first empty slot you come across. As a result it is not always the case that insertion has no comparisons. For example, if I have the hash function $n \, mod \, 5$ where n is the integer being inserted and perform the following operations $Insert(1)$, $Insert(6)$. When I insert 6 I have to follow the probe sequence since the slot where I fist land is occupied by 1. Thus there is one comparison here. So for insertion $T1$ performs as well or better than $T2$.

Therefore if the sequence contains only insertion and delete operations it is guaranteed that $T1$ performs less or the same amount of comparisons as $T2$. Thus it is guaranteed that $C_{1,i} \leq C_{2,i}$.

b) This is no longer true because it is important to note that during insertion $T1$ inserts to the front of the list and $T2$ inserts to the "end" of a probe sequence (i.e when a space in the table is empty after executing the probe sequence). So I can search for an item which is effectively "first" in $T2$ but last in $T1$.

For example let the hashing function be determined by $n \, mod \, 5$ where n is the integer being inserted. Perform the following operations $Insert(1)$, $Insert(6)$. All of these evaluate to 1 by the hashing function so they go to the same bucket. When $T1$ performs chaining it inserts the new element to the head so I have $6 -> 1$. When $T2$ performs linear probing it goes to the first empty position after resolving the hashing function, so I have $1 -> 6$. Now if I perform $Search(1)$, I will get one comparison from $T2$ since the hashing function will resolve to 1 and two comparisons from $T1$ since I have to go through the chain to find the element.

Clearly, in this case the number of comparisons when performing search in $T2$ is less than the number of comparisons when performing search in $T1$. So when $Search$ is brought in the claim is no longer true.

c) Since S is the same as in part a consider $T1$ and $T2$ after S has completed.

It is given that $T1$ and $T2$ use the same hash function and the same number of buckets. Thus after S, $T1$ and $T2$ have the exact same number of items inserted and the exact same number of items in each bucket.

Let n be the number of items inserted into $T1$ and $T2$. Let X be the random variable representing the number of comparisons made when searching for an element in either $T1$ or $T2$. I want the expected value of X. The formula for expected value is $\sum_x xP(X = x)$ where x iterates over all possible values of X. Since the search operation is random on any of the elements, $P(X = x)$ is always $1/n$ since any of the elements

can be picked uniformly. Thus $\sum_x xP(X = x)$ becomes $\sum_x x/n = 1/n \sum_x x$.

Now I have to consider, what is the expected number of comparisons made during a search for a given element. From part b I know that it is not necessarily the case that $T1$ performs better than $T2$. For example, from b I did: $Insert(1)$, $Insert(6)$. In this case if I search for 6 then $T1$ performs better however if I search for 1 then $T2$ performs better. However consider the expected value of these searches. n is 2 and so the expected comparisons for $T1$ is $1/2 * 1 + 1/2 * 2 = 1.5$ and the expected comparisons for $T2$ is $1/2 * 1 + 1/2 * 2 = 1.5$. This means that I am not interested in the element I am searching for but the number of elements in a bucket. If the number of elements in a bucket are exactly the same intuitively the expected value should even out as it did in the above example. I can generalize this below,

I know that $T1$ and $T2$ use the same hash function and the same number of buckets, this means that the number of elements in any chain or probe sequence is exactly the same. Let m_i be the ith bucket $T1$ and $T2$.

Consider $T1$. Since m_i be the ith bucket for $T1$ and $T2$ I can say that $\sum_x x = \sum_{i=1}^{|m|} \sum_{j=1}^{|m_i|} x_{ij}$ where x_{ij} is the number of comparisons needed to find the jth element in the bucket m_i. This is true since instead of having a generalized sum which is $\sum_x x$ I just specify each element specifically in $\sum_{i=1}^{|m|} \sum_{j=1}^{|m_i|} x_{ij}$. Additionally, I know that to find the jth element in a linked list will take j comparisons. Since the chaining method using linked lists $x_{ij} = j$. Thus $\sum_{i=1}^{|m|} \sum_{j=1}^{|m_i|} x_{ij} = \sum_{i=1}^{|m|} \sum_{j=1}^{|m_i|} j$.

Now consider $T2$. I can use the same argument and arrive at $\sum_{i=1}^{|m|} \sum_{j=1}^{|m_i|} x_{ij} = \sum_{i=1}^{|m|} \sum_{j=1}^{|m_i|} j$. However, this is not always true. Consider the same hash function from earlier, $n \, mod \, 5$ with the probing sequence $(n \, mod \, 5) + i$ where the probing sequence executes i times. Now consider $Insert(1)$, $Insert(2)$, $Insert(6)$. Based on the probing sequence this will result in $1 -> 2 -> 6$. In this case bucket 1 has 2 elements (1 and 6) but it will take 3 comparisons to reach 6. This is because an element from another bucket can fill a slot before the item is inserted so the probing sequence can take more than j steps. Thus j is a lower bound on the number of searches performed. So for $T2$ the expression is actually $\sum_{i=1}^{|m|} \sum_{j=1}^{|m_i|} x_{ij_2} \geq \sum_{i=1}^{|m|} \sum_{j=1}^{|m_i|} j$. Note that x_{ij} is sub scripted with 2 for clarity that this refers to $T2$.

I can multiply $1/n$ on both sides of $\sum_{i=1}^{|m|} \sum_{j=1}^{|m_i|} x_{ij_2} \geq \sum_{i=1}^{|m|} \sum_{j=1}^{|m_i|} j$ to get $1/n \sum_{i=1}^{|m|} \sum_{j=1}^{|m_i|} x_{ij_2} \geq 1/n \sum_{i=1}^{|m|} \sum_{j=1}^{|m_i|} j$. This is equal to $E_2 \geq E_1$.

Therefore $E_2 \geq E_1$.