CSCB63 A1

Saad Makrod

June 2022

Q1 Ranking Asymptotic Growth

Note the order below indicates that 1 is in O of 2 onward, 2 is in O of 3 onward and so on.

1. $4lg(lg(n))$
2. $4lg(n)$
3. $5n$
4. n^4
5. $lg(n)^{5lg(n)}$
6. $n^{lg(n)}$
7. $n^{n^{1/5}}$
8. 5^n
9. 5^{5n}
10. $(n/4)^{n/4}$
11. $n^{n/4}$
12. 4^{n^4}
13. 4^{4^n}
14. 5^{5^n}

Q2 Master Theorem Application

a) I will use repeated back substitution to solve this question since the recurrence relation $A(n) = n^2A(n-1)$ does not fit the form of the Generalized Master Theorem. We can observe the pattern in the relation below:

$A(n) = n^2A(n-1)$

$A(n-1) = (n-1)^2A(n-2)$

$A(n-2) = (n-2)^2A(n-3)$

$A(n-3) = (n-3)^2A(n-4)$

$\vdots$

$A(1) = 1$

I will perform back substitution and simplify the result to find the asymptotic bound:

$$
\begin{aligned}
A(n) &= n^2(n-1)^2(n-2)^2(n-3)^2 \dots 1^2 \\
&= (n)(n)(n-1)(n-1)(n-2)(n-2)(n-3)(n-3)\dots(1)(1) \\
&= (n)(n-1)(n-2)(n-3)\dots(1)(n)(n-1)(n-2)(n-3)\dots(1) \\
&= [(n)(n-1)(n-2)(n-3)\dots(1)][(n)(n-1)(n-2)(n-3)\dots(1)] \\
&= n!n! \\
&= (n!)^2
\end{aligned}
$$

Thus it is clear that $A(n) \in \theta((n!)^2)$.

b) I have the function $A(n) = 8A(\lfloor \frac{n}{2} \rfloor) + 2n^3 + 4n$ and $A(0) = 6$. This function has the form of the Generalized Master Theorem so I will use it to solve for the asymptomatic bound of this recurrence relation.

From the Generalized Master Theorem let $a = 8$, $b = 2$, $f(n) = 2n^3 + 4n$ Then since $f(x)$ is a function with the highest power being 3 I know that $\theta(f(x)) = 3$, so from the Generalized Master Theorem $c = 3$.

Now I will compute $log_b(a) = log_2(8) = 3$. Thus this falls into the class where $log_b(a) = c$ so by the Generalized Master Theorem $A(n) \in \theta(n^{log_b a} log_b n)$. Plugging in the values I get $A(n) \in \theta(n^3 log_2(n))$.

Thus $A(n) \in \theta(n^3 log_2(n))$.

Q3 Prove if $f(n) \in O(g(n))$ then $lg(f(n)) \in O(lg(g(n))$ given $f(n) \geq 1$ and $lg(g(n)) \geq 1$

Since $f(n) \in O(g(n))$ by definition I know $\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}$ such that $0 \leq f(n) \leq cg(n) \forall n \geq n_0$. However, it is given that $f(n) \geq 1$ so I can change the inequality to be $1 \leq f(n) \leq cg(n)$.

Since $1 \leq f(n) \leq g(n)$ is bounded from below by 1 I can lg the terms in the inequality without flipping the inequalities. Thus,

$1 \leq f(n) \leq cg(n)$

$lg(1) \leq lg(f(n)) \leq lg(cg(n))$

$lg(1) \leq lg(f(n)) \leq lg(c) + lg(g(n))$ By lg rules

Now I need to choose a c_2 such that $c_2 lg(g(n)) \geq lg(c) + lg(g(n))$ so that I can conclude that $lg(f(x)) \in O(lg(g(n)))$. I know that for any $x \in \mathbb{R}^+$ $lg(x) < x$. So I will choose $c_2 = 1 + c$. Now solving $c_2 lg(g(n))$,

$c_2 lg(g(n)) = (1 + c) lg(g(n)) = lg(g(n)) + clg(g(n))$

I know that $clg(g(n)) > lg(c)$ since it is given that $lg(g(n)) \geq 1$ and as stated earlier, $c \in \mathbb{R}^+$ and for any $x \in \mathbb{R}^+$ $lg(x) < x$. Thus since $c > lg(c)$ and $lg(g(n)) \geq 1$ the product of c and $lg(g(n))$ is guaranteed to be larger than $lg(c)$.

Thus,

$lg(1) \leq lg(f(n)) \leq lg(c) + lg(g(n))$

$lg(1) \leq lg(f(n)) \leq lg(c) + lg(g(n)) \leq lg(g(n)) + clg(g(n)) = c_2 lg(g(n))$

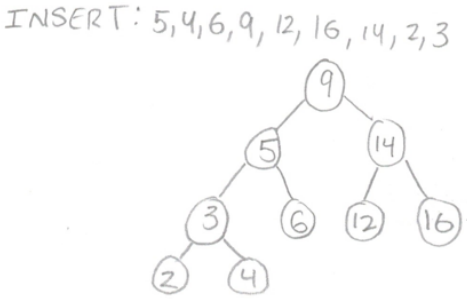
$0 \leq \lg(f(n)) \leq c_2 lg(g(n))$

Thus I have shown that $0 \leq \lg(f(n)) \leq c_2 lg(g(n))$ for $n \geq n_0$ so by definition $lg(f(n)) \in O(lg(g(n))$.

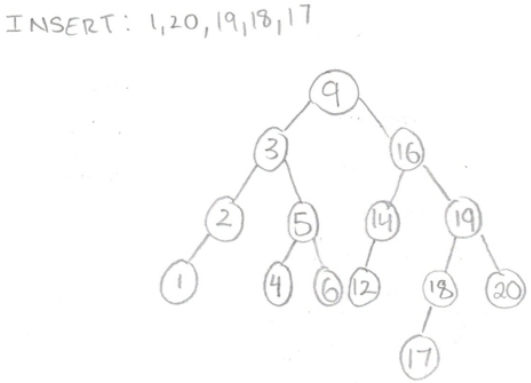
QED

Q4 AVL Tree Insertion and Deletion

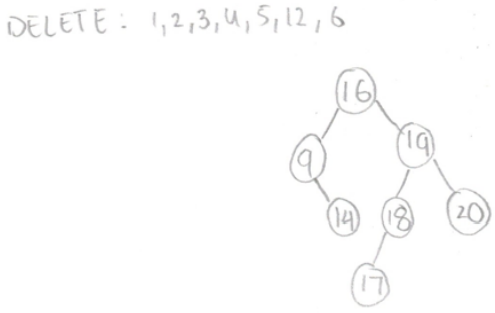
Part 1: Insertion



Part 2: Insertion



Part 3: Deletion



Q5 Analysis of Sorting Acceleration Attempt

I know that Insertion Sort requires cn^2 to sort n numbers (assume c is constant work). I also know that the time for merging in Merge Sort is $c(m-1)n$ assuming m is the number of subarrays, n is the number of items in all subarrays together, and c is the same constant work from Insertion Sort.

The following algorithm was proposed:

- The initial array of size n is split into m subarrays of size (n/m). Assume they are of equal sizes
- Each subarray of size (n/m) is sorted separately using Insertion Sort
- Sorted subarrays are merged into final sorted array

My goal is to tackle this prompt: If you are convinced that this acceleration is possible, find the optimum value of m and compare the resulting complexity with Insertion Sort and with Merge Sort

I can derive a generalized formula using the assumption in the first paragraph. If I sort m subarrays of size (n/m) using Insertion sort then it is clear that to sort one subarray it will take $c(n/m)^2$ work. So to sort m subarrays it will take $cm(n/m)^2 = c\frac{n^2}{m}$ work. Now to use merge the subarrays using the same method as Merge Sort is will take $c(m-1)n$ work since I have n number and m subarrays. Thus the total work of this algorithm in terms of m and n is $c\frac{n^2}{m} + c(m-1)n = cn(\frac{n}{m} + (m-1))$.

Now I need to choose a value for m in terms of n such that we minimize $(\frac{n}{m} + (m-1))$. I will list the properties of m before choosing a value:

1. $m < n$
2. n is divisible by m

In order to minimize $(\frac{n}{m} + (m-1))$ I must express m as the largest function of n such that the above two properties are met. This function is clearly $\sqrt{n}$. Thus the value of optimum value of m is $\sqrt{n}$ and plugging this into the expression I determined earlier,

$cn(\frac{n}{m} + (m-1)) = cn(\frac{n}{\sqrt{n}} + \sqrt{n} - 1) = nc(2\sqrt{n} - 1) = 2cn^{3/2} - nc$

Thus when the optimum value of m is chosen the algorithm is $O(n^{3/2})$. This is a slight improvement to Insertion Sort but it still is much worse than the complexity of Merge Sort.

I know that Insertion Sort has a complexity of n^2 and Merge Sort has a complexity of $nlog(n)$. Using this it is clear that $nlog(n) \in O(n^{3/2})$, $nlog(n) \in O(n^2)$ and $n^{3/2} \in O(n^2)$. Thus when comparing this new algorithm I can clearly see that in terms of complexity Merge Sort $<$ Proposed Algorithm $<$ Insertion Sort.

Thus an acceleration is indeed possible if I choose $m = \sqrt{n}$ however the new algorithm still has a higher complexity then Merge Sort with the overall comparison being Merge Sort $<$ Proposed Algorithm $<$ Insertion Sort.

Q6) Data Structure Proposal

Below is the description of the data structure I must propose:

Consider an abstract datatype RECENT that keeps track of recently accessed values. In this ADT, we consider a set $S = \{1, ..., n\}$ and a subset R of S, of size m. Initially, $R = \phi$. The only operation is access, which takes a parameter $i \in S$. It adds i to R and, if this causes the size of R to become bigger than m, removes the element from R that was least recently the parameter of an access operation. Give a data structure for this abstract data type that uses $O(mlog(n))$ space (measured in bits) and has worst case time complexity $O(log(m))$.

Proposal:

Consider a Queue as the data structure and an additional integer to record the size of the Queue.

Since the Queue will have at most m elements I know that the space complexity must be $m * space\ to\ store\ an\ element + space\ to\ store\ queue\ size$. An element in the Queue will simply store an element from S.

Note that S is a sequence of numbers from 1 to n. This means the largest number that will ever be stored in an element is n. To store n it will take $log(n)$ bits in binary representation (note log is base 2 here). Thus the maximum space required to store an element from S is $log(n)$. When considering the space required to store the size of the Queue, I know that the Queue will have at most m elements and $m \leq n$ since R is a subset of S. To store n it will take $log(n)$ bits in binary representation (note log is base 2 here). Thus the maximum space required to store the size of the Queue is $log(n)$.

Thus when returning to the original formula,

$m * space\ to\ store\ an\ element + space\ to\ store\ queue\ size = m * log(n) + log(n) = (m+1)log(n)$

I know that $(m+1)log(n) \leq 2mlog(n)$ since m is an integer and for any integer x where x is strictly greater than 0, it is clear that $2x \geq x+1$. Thus it is clear that $(m+1)log(n) \leq 2mlog(n) \in O(mlog(n))$.

Thus it is clear that a Queue uses $O(mlog(n))$ space (measured in bits).

Now we will consider the operation access. This operation has two phases I either insert an element into the Queue or I insert an element into the Queue and delete the oldest element in the Queue since the size has exceeded m.

I know that to insert an element into a Queue it is $O(1)$. I know this because insertion is just an enqueue operation which means that I just have to update the tail pointer. Additionally incrementing the counter for the size of the Queue is also an $O(1)$ operation. So I know that insertion has a worst case complexity of $O(1)$.

When having to insert and delete I will have to perform both an enqueue and dequeue operation since I need to remove the oldest element which access was performed on. I know enqueue will take $O(1)$ for the reasoning earlier. Dequeue will take also $O(1)$ since in this case all I have to do is update the head pointer. Additionally in this case there is no need to update the size counter since we added and removed an element so the size stays the same. Thus this is an $O(1)$ operation.

So I have shown that in the worst case the Queue will perform the access operation in $O(1)$ time. I know that for Big O anything in $O(1)$ is also in $O(log(m))$. Thus the access operation does indeed have a worst case time complexity of $O(log(m))$.

As a result the Queue with an integer to store the size of the Queue has both a space complexity of $mlog(n)$ and a worst case time complexity of $log(m)$ for the access operation. So the Queue with an integer to store the size of the Queue fulfills all the requirements.

Thus a Queue with an integer to store the size of the Queue is a possible data structure we can use.