CPSC 331

Assignment #4

June 24th/2018

Kenneth Sharman

ID# 00300185

#1a) To determine the upper bounds on the expected number of probes in a hash table I made use of the formulas developed by Donald Knuth (sourced from lecture slides). Rather than simply computing the math, I figured this was a chance to practice coding in Python (see attached for the file if you're interested). The results are displayed below. I rounded results up to an integer, as the question asked for an upper bound.

	Linear Probing	Quadratic Probing	Double Hashing
$\alpha = 3/4$			
successful	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) = 3$	$1 - \ln(1-\alpha) - \frac{\alpha}{2} = 3$	$\frac{1}{\alpha} ln\left(\frac{1}{1-\alpha}\right) = 2$
unsuccessful	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right) = 9$	$\frac{1}{1-\alpha} - \alpha - ln(1-\alpha) = 5$	$\frac{1}{1-\alpha} = 4$
$\alpha = 7/8$			
successful	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) = 5$	$1 - \ln(1-\alpha) - \frac{\alpha}{2} = 3$	$\frac{1}{\alpha} ln\left(\frac{1}{1-\alpha}\right) = 3$
unsuccessful	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right) = 33$	$\frac{1}{1-\alpha} - \alpha - ln(1-\alpha) = 10$	$\frac{1}{1-\alpha} = 8$

Our results indicate that linear probing is on average the most expensive, and Double Hashing is the least expensive. There isn't a massive difference between Quadratic and Double Hashing, at least with these load factors. The most significant spike in the number of probes occurrs with unsuccessful linear probes, with a large load factor.

All of these observations are consistent with the graphs that compare the strategies, which are presented in the lecture slides.

#1b) Theorem 11.1 and 11.2 (CLRS text) state that in a hash table which collisions are resolved by chaining, unsuccessful and successful searches take average-case time $\Theta(1+\alpha)$, assuming simple uniform hashing. Suppose we implement the hash table as an array of ArrayLists (as opposed to linked lists), and that each list is kept in sorted order. Similar to the unsorted version, we have constant time to determine hash value. We can make use of Binary Search, since the list is in sorted order. We have seen that Binary Searches are $\Theta(\lg n)$ in terms of number of comparisons. Here we have lists in a table, with average size α, so $\Theta(\lg \alpha)$ run time for both successful and unsuccessful binary searches. Combining hash and binary search operations we have

Successful and Unsuccessful searches take average-case time $\Theta(1+\lg \alpha)$.

Therefore, keeping this lists sorted reduces the cost for searching, but remember that it comes at the added cost of sorting the lists.
To determine the running time for insertions and deletions, recalling that we are implementing the lists using ArrayLists. To insert into, or delete from an ArrayList we have linear average running time, as all the elements past the insertion/deletion index must be shifted positions to accomodate this change. Further, we must locate the position where insertion/deletion is to occur. The running time to locate is simply that of a successful search, $\Theta(1+\lg \alpha)$. The linear run time for resizing the list (with average length α) dominates the $\lg \alpha$ term and we have:

Average-case time for insertions/deletions is $\Theta(1+\alpha)$.

#2. Loop invariant for heap sort: At an iteration with a given i, we know

1) $arr[i+1] \leq arr[i+2] \leq \ldots \leq arr[n-2] \leq arr[n-1]$
2) $arr[0,1,2,\ldots,i]$ is a heap
3) $arr[k] \geq arr[j]$, $i+1 \leq k \leq n-1$, $0 \leq j \leq i$

Assume that Heapify organizes the array such that it satisfies the heap property, and that the percolate down operation is correct.

Initialization The loop starts with $i = n-1$. The Heapify method is called on entire array, thus $arr[0 \ldots n-1]$ is a heap. Note:

$arr[i+1] = arr[n] \leq \ldots \leq arr[n-1]$ and

$arr[k] \geq arr[j]$, $n \leq k \leq n-1$, $0 \leq j \leq n-1$ are both both vacuously true, since they are both empty inequalities $\Rightarrow$ Initialization holds.

Maintenance: Assume L.I. holds up to an iteration with a given i. From assumption we know $arr[0 \ldots i]$ is a heap, which implies $arr[0]$ is the max element in $arr[0 \ldots i]$. In the loop $arr[0]$ and $arr[i]$ are swapped. Thus, $arr[j] \leq arr[i]$, $0 \leq j \leq i-1$ after the loop. From assumption we also know $arr[i] \leq arr[i+1] \leq \ldots \leq arr[n-1]$. Finally, $arr[0, \ldots, i-1]$ is a heap since the percolate method is correct.

$\therefore$ Maintenance step holds. $\Rightarrow$ Loop Invariant holds.

Precondition: array contains comparable elements.

Postcondition: array is sorted : $arr[0] \leq arr[1] \leq arr[2] \leq \ldots \leq arr[n-1]$

Correctness: The loop runs from $i = n-1$ down to 1, so when we exit the loop $i = 0$. The loop invariants says: $arr[0+1] \leq arr[0+2] \leq \ldots arr[n-1]$ Which means $arr[1 \ldots n-1]$ is sorted. Condition 3 in the loop invariant tells us $arr[0] \leq arr[1]$ therefore $arr[0 \ldots n-1]$ is sorted. When the precondition is met the postcondition holds which proves correctness.

#3 a) From the KQuick Sort pseudo-code we see that if $n < K$ then insertion sort is called. The worst case number of both comparisons and movements is $\Theta(n^2)$. Therefore, $T(n) = \Theta(n^2)$, $n < K$.
For $n \geq K$ we have three steps: KPartition (given as $\Theta(n)$), and 2 recursive calls on partitions of size $K-1$ and $n-K$.

$$T(n) \leq \begin{cases} c_1 n^2 & , \ n < K \\ c_2 n + T(K-1) + T(n-K) & , \ n \geq K \end{cases}$$

b) To find an upper bound on $T(n)$ we will turn inequality into an equality, ignore constants, and note that $K-1 < K$ so $T(K-1)$ can be expressed as $c_1(K-1)^2$. This yields an expression for upper bound.

$T(n) = n + (K-1)^2 + T(n-K)$ Using method of substitution:

$= n + (K-1)^2 + (n-K) + (K-1)^2 + T(n-2k)$

$= 2n + 2(K-1)^2 - K + T(n-2k)$

$T(n) = 2n + 2(K-1)^2 - k + (n-2k) + (K-1)^2 + T(n-3k)$

$= 3n + 3(K-1)^2 - k(1+2) + T(n-3k)$

$\vdots \qquad \vdots \qquad \vdots$

Assume n is multiple of K. Then we must subtract K from n, $\frac{n}{K}$ times in order to get a value less than K.

$$\Rightarrow T(n) = \frac{n}{K} n + \frac{n}{K}(K^2 - 2K + 1) - k(1+2+\ldots) + T(0)$$

Upperbound: $T(n) \leq c\left[\frac{n^2}{K} + n\left(K - 2 + \frac{1}{K}\right) - k\sum_{i=0}^{\frac{n}{k}-1} i + 0^2\right]$, where c is a constant

c) Worstcase running time is for partition size of $K=1 \Rightarrow T(n) = O(n^2)$

d) The worst case runtime of quicksort is when partition size is 1 and runtime is $\Theta(n^2)$. What we have done here is essentially equivalent.

#4. Prove that a tree with n vertices has $n-1$ edges.

Inductive Hypothesis: A tree with n vertices has $n-1$ edges.

Recall that all trees are acyclic.

Base Case: $n = 1$ vertex. Since there are no other vertices to connect to there are $1-1=0$ edges.

Inductive Step: Assume hypothesis holds for any tree with up to $V-1$ vertices.

Suppose that T is any tree with V vertices. Since it is a tree and not a forest, we know that there is a non-zero number of edges, as the vertices must be connected.

Now remove any one of the edges. Since T is acyclic, this will result in two smaller trees T_1 and T_2, with V_1 and V_2 vertices respectively.

Note $V_1, V_2 \in \mathbb{Z}^+$ and $V_1 + V_2 = V$ $\therefore V_1 < V$ and $V_2 < V$.

Since we have only removed an edge, the the sum of all vertices remains unchanged.

By assumption we know T_1 has $V_1 - 1$ vertices and T_2 has $V_2 - 1$ vertices.

Together they have $(V_1 - 1) + (V_2 - 1) = V_1 + V_2 - 2 = V - 2$ vertices.

Now connect T_1 and T_2 with the edge that was removed.

We once again have any tree with V vertices and

$$(V-2) + 1 = V - 1 \text{ edges.}$$

Q.E.D.