Assignment 3 - CSC265

David Knott
Student #999817685

Celton McGrath
Student #999632303

November 14, 2013

1. Note that $z \in \{x_j + y_i : 1 \leq j, i \leq n\}$ if and only if there exists an $j, i \in [1, \ldots, n]$ such that $z = x_j + y_i$. Furthermore, this is equivalent to $z - y_i = x_j$ for some $j, i \in [1, \ldots, n]$. Using this fact to our advantage, we produce the following two step algorithm:

   Let $h : \mathbb{R} \to [1, \ldots, n]$ be a universal hash function. Let a be a sequence of n linked lists. The first step of our algorithm will iterate over all x and append it to $a[h(x)]$. This will take $O(n)$ time.

   Once this is done, to determine if a number $t = x_j$ for some $j \in [1, \ldots, n]$ we must simply check the entries of $a[h(t)]$ for equality. If $a[v]$ is, on average, constant for any $v \in [1, \ldots, n]$ then this will take expected time $O(1)$.

   The next step does exactly this by determining, for all $i \in [1, \ldots, n]$, if $z - y_i = x_j$ for some $j \in [1, \ldots, n]$. This is done by iterating over y and searching for $z - y$ in $a[h(z - y)]$. Assuming the expected time for the search is $O(1)$, the expected time to iterate over all y will be order $O(n)$.

   Finally, we must prove that if h is a universal hash function then $a[h(v)]$ will on average be constant. To do this we use the property that for any $i \in D(h)$ and $y \in R(h)$[1] we have that $\Pr_h(h(i) = y) = 1/n$. To find the estimated size of any $m \in a$ would be equivalent to counting the estimated number of $x \in [1, \ldots, n]$ that map to m. Suppose $a[y] = m$, then we have the following:

$$\begin{aligned} \mathrm{Ex}[|m|] &= \sum_{i=1}^{n} \Pr_h(h(i) = y) \\ &= \sum_{i=1}^{n} 1/n \end{aligned}$$

[1] Here the functions D and R denote the domain and range.

$$= 1$$

Therefore the expected size of any $m \in a$ will be constant, and hence a search over any list in a will be expected order $O(1)$.

The combination of these two steps will take $O(2n) = O(n)$ time.

2. (a) Suppose at the beginning H and T are empty. Suppose the first $m-1$ operations are enqueues. This means that T has a size $m-1$. If the mth operation is a dequeue, then we must copy all $m-1$ elements over to H, which will take $\Omega(m)$ time. This is also an upper bound on the worse case, since the only operation that takes more than constant time is dequeue, and it's runtime is proportional to the number of elements in T. Since the number of elements in T is bounded above by the number of enqueues that have added elements to it, and the number of enqueues is bounded by m, this means any dequeue operation after m operations by starting when empty stacks cannot take more than $O(m)$ time. Therefore the worst case is in $\Theta(m)$.

   (b) Suppose that pushing or popping elements of a stack costs us one unit of time, but our amortized cost will be three units of time. Also, suppose that every time we push an element onto T we associate the leftover 2 credits with that element. If T is empty at the beginning of the m operations, enqueuing an object will give T an element, as well as 2 extra credits to do a push or a pop later. If T has n elements and $2n$ extra credits, then enqueuing another object will leave T with $n+1$ elements and $2(n+1)$ extra credits. By induction, this implies that T will always have $2n$ extra credits, where n is the number of elements in T.

   When a dequeue call is done when H is non-empty we will simply throw out the extra credits.

   Consider now a dequeue call when H is empty. We must copy all elements of T over to H which will require a push and a pop for each element, totaling to $2n$ units of time. Thankfully, T has exactly that many extra credits, so this operation will take amortized constant time.

3. (a) Since *GRAFT* and *MAKE-TREE* are $O(1)$, the complexity would have to come from *FIND-DEPTH*, which is $O(k)$ where k is the length from the searched node to it's root. Suppose $T(f(x))$ is the runtime of f on x. Let $T(\mathit{GRAFT}(r, v)) = 1$, $T(\mathit{MAKE\text{-}TREE}(v)) = 1$ and $T(\mathit{FIND\text{-}DEPTH}(v)) = k$ where k is the number of nodes from v to it's root, including the root and itself. Note that $T(\mathit{FIND\text{-}DEPTH}(v))$ can be at most the number of nodes in the forest. The worst case for m operations would then be a scenario where *FIND-DEPTH* is run some multiple of m times, and each time it's run, say at step i, it takes some multiple of i times.

   Suppose we did the following operations: we initially create one tree in our forest, call it t, that contains the single node p. Then, for the next $m - 1$ operations we alternate between the following three operations:

   1. Creating a new tree called s.
   2. Grafting s onto t such that s is the root of t, then assigning s to t.
   3. Running *FIND-DEPTH* on p.

   Let $f(i)$ be the function that does the alternation, where i is the step number. Each call on *FIND-DEPTH* will take time proportional to $1/3$ of i, since it must iterate over ever node in the forest and after i steps there are $i/3$ nodes. From this, we have the following:

$$
\begin{aligned}
T(m \textit{ operations}) &= 1 + \sum_{i=1}^{m-1} T(f(i)) \\
&= 1 + \sum_{i=1}^{\frac{(m-1)}{3}} f(3i) + \sum_{i=1}^{\frac{(m-1)}{3}} f(3i+1) + \sum_{i=1}^{\frac{(m-1)}{3}} f(3i+2) \\
&= 1 + \sum_{i=1}^{\frac{(m-1)}{3}} T(\mathit{FIND\text{-}DEPTH}(p)) \\
&\quad + \sum_{i=1}^{\frac{(m-1)}{3}} T(\mathit{MAKE\text{-}TREE}(v)) \\
&\quad + \sum_{i=1}^{\frac{(m-1)}{3}} T(\mathit{GRAFT}(t, s)) \\
&= 1 + \sum_{i=1}^{\frac{(m-1)}{3}} T(\mathit{FIND\text{-}DEPTH}(p)) \\
&\quad + \sum_{i=1}^{\frac{(m-1)}{3}} 1 + \sum_{i=1}^{\frac{(m-1)}{3}} 1
\end{aligned}
$$

$$= 1 + \frac{2(m-1)}{3} + \sum_{i=1}^{\frac{(m-1)}{3}} T(\textit{FIND-DEPTH}(p))$$

$$= 1 + \frac{2(m-1)}{3} + \sum_{i=1}^{\frac{(m-1)}{3}} \frac{i}{3}$$

The last sum will be in $\Theta(m^2)$, so $T(m \textit{ operations}) \in \Theta(m^2)$.

(b) *MAKE-TREE*(v) will initialize two trees, both with one node, v at it's root. It will then put the first tree in the forest $\mathcal{F}$. The second tree's root node will get a second field which will contain it's pseudodistance, which is initialized to 1. This tree is then inserted into the forest $\mathcal{S}$.

(c) To modify the function *FIND-SET*(v) it will be beneficial to go over how it works. Though the book uses a recursive function, I find it may be easier to use an iterative one. In the iterative case, *FIND-SET* works like so:

1. Given the input node v from the forest $\mathcal{F}$, suppose $v.toS$ returns the corresponding node in $\mathcal{S}$. Let $w = v.toS$.
2. Initialize a linked list structure that we can use as a stack to push and pop pointers to nodes.
3. We push a reference to w into the linked list, then let $w = w.p$.
4. We repeat #3 until w is the root node. w will be our return value. We do not push a reference to the root node onto the list.
5. Finally we pop an element off the stack, call it x, and let $x.p = w$. We repeat this for all elements of the stack. This is the path compression step.
6. We return w.

To modify this, we make the following changes to each step:

After steps 1 and 2 we create an variable k that is initialized to 0. In step five, in addition to setting $x.p = w$ we also set $x.d = x.d + k$ and then $k = x.d$. Once we've processed all the elements on the stack, we return k as the number of nodes between the root and v, including the root.

(d)

(e)