Assignment 2 - CSC265

David Knott
Student #999817685

October 21, 2013

1.

2. a) Note that for any $h \in \mathcal{H}$ for all distinct $x_1, x_2, x_3 \in U$ and all $y_1, y_2, y_3 \in \{0, \ldots, m-1\}$ we have that:

$$\begin{aligned}\text{Prob}_{h\in\mathcal{H}}[h(x_1) = y_1 \wedge h(x_2) = y_2 \wedge h(x_3) = y_3] = \\ \text{Prob}_{h\in\mathcal{H}}[h(x_1) = y_1] \times \\ \text{Prob}_{h\in\mathcal{H}}[h(x_2) = y_2] \times \\ \text{Prob}_{h\in\mathcal{H}}[h(x_3) = y_3] = \frac{1}{m^3}\end{aligned}$$

Since the xs are arbitrary, all of these probabilities must be the same. Solving for each probability yields $1/m$. So for for all distinct $x_1, x_2, x_3 \in U$ and all $y_1, y_2, y_3 \in \{0, \ldots, m-1\}$ we have that

$$\begin{aligned}\text{Prob}_{h\in\mathcal{H}}[h(x_1) = y_1] = \frac{1}{m} \\ \text{Prob}_{h\in\mathcal{H}}[h(x_2) = y_2] = \frac{1}{m} \\ \text{Prob}_{h\in\mathcal{H}}[h(x_3) = y_3] = \frac{1}{m}\end{aligned}$$

If we let $y_1 = h(x_2)$, since $h(x_2) \in \{0, \ldots, m-1\}$ we have:

$$\text{Prob}_{h\in\mathcal{H}}[h(x_1) = h(x_2)] = \frac{1}{m}$$

Which implies that a 3-universal family is universal.

3. a) At each node we store the number of nodes and the average of the subtree rooted at that node.

   b) While we're inserting the item we traverse down the tree to find a spot to put the new node. At every node we pass, we add one to the subtree total and recompute the average with the formula $(a + nc)/(n + 1)$ where a is the value we're inserting, c was the original average and n was the original number of nodes in the subtree. During rotations involved nodes recompute their subtree averages with similar formulas. The amount of nodes that need recomputing is constant, since it only happens when a node's children move out of their subtree. For any rotation the number of nodes that have this happen is at most 2. This means the running time is unaffected.

   c)

4. a) Since each insert increases the size of the list by 1, we must find the expected number of prepends after $k-1$ operations. Using the indicator random variable I_n associated with the event that the nth operation was a prepend. If I is the random variable for the number of prepends after $k-1$ operations then we have:

$$\begin{aligned}
I &= \sum_{n=1}^{k-1} I_n \implies \\
\mathrm{E}\big[I\big] &= \mathrm{E}\left[\sum_{n=1}^{k-1} I_n\right] \implies \\
\mathrm{E}\big[I\big] &= \sum_{n=1}^{k-1} \mathrm{E}\big[I_n\big] \implies \\
\mathrm{E}\big[I\big] &= \sum_{n=1}^{k-1} p \implies \\
\mathrm{E}\big[I\big] &= (k-1)p
\end{aligned}$$

By way of the linearity of expectation an the equality $\mathrm{E}[I\{A\}] = \Pr\{A\}$. The expected length of the list after $k-1$ operations is $(k-1)p$.

b) At this point the expected size of the list is $(k-1)p$. Assuming that the kth operation is an access, the number of expected number of steps would be:

$$\begin{aligned}
\mathrm{E}\big[I\big] &= \sum_{n=1}^{|S|} \frac{1}{|S|} n \implies \\
\mathrm{E}\big[I\big] &= \sum_{n=1}^{(k-1)p} \frac{n}{(k-1)p} \implies \\
\mathrm{E}\big[I\big] &= \frac{1}{2}((k-1)p+1)
\end{aligned}$$

Where I is the random variable for the number of steps given the kth operation is an access.

If the kth operation is a prepend then the number of steps is 1. Let A be the event that the operation is a prepend and B be the event that the operation is an access. If J_k is the random variable for the number of steps at the kth operation we have:

$$\begin{aligned}
\mathrm{E}\big[J_k\big] &= \Pr(A) + \Pr(B)\frac{1}{2}((k-1)p+1) \implies \\
\mathrm{E}\big[J_k\big] &= p + (1-p)\frac{1}{2}((k-1)p+1) \implies \\
\mathrm{E}\big[J_k\big] &= \frac{1}{2}(-kp^2 + kp + p^2 + 1)
\end{aligned}$$

So the expected number of steps is $\frac{1}{2}(-kp^2 + kp + p^2 + 1)$

c) If J is the random variable for the number of steps for the entire series of operations, then we have:

$$
\begin{aligned}
J &= \sum_{k=1}^{n} J_k \implies \\
\mathrm{E}\big[J\big] &= \mathrm{E}\left[\sum_{k=1}^{n} J_k\right] \implies \\
\mathrm{E}\big[J\big] &= \sum_{k=1}^{n} \mathrm{E}\big[J_k\big] \implies \\
\mathrm{E}\big[J\big] &= \sum_{k=1}^{n} \frac{1}{2}(-kp^2 + kp + p^2 + 1) \implies \\
\mathrm{E}\big[J\big] &= \frac{1}{4}n(2 + p + np + p^2 - np^2)
\end{aligned}
$$

So n operations is expected to take $\frac{1}{4}n(2 + p + np + p^2 - np^2)$ steps.

5.

6. For a list of lists with these restrictions, the number of lists and the maximum size of the largest list will be:

$$\lceil \frac{1}{2}(\sqrt{8n+1}-1)\rceil \in O(\sqrt{n})$$

In addition to this data structure, each group will keep track of how "rotated" their list is. This way we can find the maximum and minimum elements of an array in constant time. Keeping this information updated will not add anything to the runtime of any of the algorithms. For convenience, if g is a subgroup of S, then $g.\max$ and $g.\min$ will yield the maximum and minimum elements.

a) For insertion, for each step in the algorithm we maintain the property that the k^{th} group has at most k elements. This means every time we insert an element into a full group we must take one out. Taking advantage of this fact, consider the following sub algorithm:

```
function INSERT-INTO-GROUP($g$, $x$)
    if $x \geq g.\max$ then
        return $x$
    end if
    $r \leftarrow g.\max$
    Put $x$ where $g.\max$ was.
    Subtract one to the group's rotation amount.
    Bubble $x$ up the list until it reaches equilibrium.
    return $r$
end function
```

Since all the elements of a group are smaller than any of the elements in it's successor, the return value of this algorithm will be the minimum of the next group if we inserted it there. This is precisely how the algorithm will work:

```
function INSERT($S$, $x$)
    $i \leftarrow 1$
    $r \leftarrow x$
    while $i < |S|$ do
        $r \leftarrow$ INSERT-INTO-GROUP($S[i], r$)
    end while
    if $S[|S|]$ is full then
        $S[|S|+1] \leftarrow [r]$
    else
        Put $r$ at the end of $S[|S|]$ and bubble until equilibrium.
    end if
end function
```

The subfunction INSERT-INTO-GROUP has two behaviors: the first will return x without affecting g, the second will return the maximum value of g, having inserted x into g by preforming at most $|g|$ shifting operations. If x was already the minimum of g then there would be only a constant amount of shifting.

Through the execution of the algorithm, once calls to INSERT-INTO-GROUP start exhibiting the second behavior, they will continue to do so for the rest of the algorithm. This is because the max of one group is the min of the next. Additionally, INSERT-INTO-GROUP will take $O(|g|)$ time once, and $O(1)$ time anytime else. Since INSERT INTO GROUP is called $\sqrt{n}$ times, this means INSERT will take $O(\sqrt{n})$ time.

b) For deletions we assume that the input x is the position in the list of all lists where the element we're removing is. The algorithm is constructed as such:

```
function DELETE($S$, $x$)
    if $x \in S[|S|]$ then
        Remove $x$ from $S[|S|]$ and shift everything backward one space.
        Subtract one from the rotation amount if $x$ was a minimum.
        return
    else
        $r \leftarrow S[|S|].\min$
        Remove the minimum from $S[|S|]$, shift everything backward.
        $i \leftarrow |S| - 1$
        while $i > 0$ do
            if $x \in S[i]$ then
                Replace $x$ with $r$.
                Bubble $r$ it until it reaches it's place.
                Change the rotation amount of $S[i]$ accordingly.
                return
            else
                $m \leftarrow r$
                $r \leftarrow S[i].\min$
                Replace $S[i].\min$ with $m$.
                Subtract one from $S[i]$'s rotation amount.
            end if
        end while
    end if
end function
```

What the algorithm essentially does is remove the element at position x and replace it with the next group's minimum, doing necessary shifting to keep the current group sorted. It then recurses to the next group and replaces it's missing space with the next group's minimum. This takes constant time. It continues to do this until it reaches the bottom. The pseudo code actually does all of this in reverse, but explaining is better this way.

This algorithm does at most $\sqrt{n}$ iterations of it's while loop. Only two iterations of this loop will take more than constant time. These iterations take $O(\sqrt{n})$ time. This means the algorithm takes $O(3\sqrt{n}) =$

$O(\sqrt{n})$ time.

c) Searching can be done by doing a binary search over each group's max and min. If the search key "falls between the cracks" of two groups (i.e. for two successive groups g_1 and g_2 we find $g_1.\max < x < g_2.\min$) then it does not exist in the data structure. If $x \in [g.\min, g.\max]$ for some $g \in S$ then we do a binary search over the elements of g, keeping in mind that it's rotated by a set amount.

Let g.offset be the group g's rotation amount.

```
function SEARCH($S$, $x$)
    $intervalmin \leftarrow 0$
    $intervalmax \leftarrow 2|S| - 1$
    while $intervalmin < intervalmax$ do
        $midpoint \leftarrow \lfloor (intervalmin + intervalmax)/2 \rfloor$
        $smid \leftarrow \lfloor midpoint/2 \rfloor$
        $smidval \leftarrow S[smid + 1].\max$
        if $midpoint \mod 2 = 0$ then
            $smidval \leftarrow S[smid + 1].\min$
        end if
        if $x > smidval$ then
            $intervalmin \leftarrow midpoint + 1$
        else
            $intervalmax \leftarrow midpoint$
        end if
    end while
    if $intervalmin \neq intervalmax$ then
        return No Key
    end if
    $r \leftarrow intervalmin$
    if $r \mod 2 == 0$ then
        return No Key
    end if
    $r \leftarrow \lfloor r/2 \rfloor$
    $g \leftarrow S[r]$
    $intervalmin \leftarrow 0$
    $intervalmax \leftarrow |g| - 1$
    while $intervalmin < intervalmax$ do
        $midpoint \leftarrow \lfloor (intervalmin + intervalmax)/2 \rfloor$
        $gmidval \leftarrow g[((midpoint + g.\text{offset}) \mod |g|) + 1]$
        if $x > gmidval$ then
            $intervalmin \leftarrow midpoint + 1$
        else
            $intervalmax \leftarrow midpoint$
        end if
    end while
    if $intervalmin = intervalmax$ and then
```

```
      return $x$'s data/position
    end if
    return No Key
end function
```

Since there are at most $\sqrt{n}$ groups and at most $\sqrt{n}$ elements in each group, this algorithm takes $O(\log(\sqrt{n}))+O(\log(\sqrt{n})) = O(1/2\log(n)) = O(\log(n))$ time.

d) Implementing this structure as a dictionary is as simple as making each element of each array a pointer to a key-value pair. In comparison to other data structures, consider the following table:

	Unsorted Ary	Sorted Ary	Red-Black	Rotated Lists
Insert	$O(1)$	$O(n)$	$O(\log(n))$	$O(\sqrt{n})$
Delete	$O(1)$	$O(n)$	$O(\log(n))$	$O(\sqrt{n})$
Search	$O(n)$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

Running time for search trumps all of these algorithms. However, insert and delete, although faster than the sorted array, is slower than Red-Black trees. This structure would be a good fit for situations where insertions and deletions are less common than searches.

It would also be good for large datasets, because in practice the algorithm searches over a sorted list of size $\sqrt{n}$ and then a sorted list of size $n \leq \sqrt{n}$. If keys are ranked in such a way that lower values are searched more often, then n would be small, and the amount of comparisons would be closer to $\log(\sqrt{n}) + \log(c)$ for some constant c.

Red-Black trees don't have this advantage, since low valued keys would be closer to the leftmost side of the tree, and possibly near the bottom. This would make the practical number of comparisons around $\log(n)$ for low valued keys.