CSC265 Fall 20120 Homework Assignment 1

The list of people with whom I discussed this homework assignment: Kunal Chawla, Raymond Liu.

1. For $1 \leq k \leq g$, what is the location, $f(k)$, of the first element in group k? Briefly justify your answer.

Solution

To find $f(k)$, we count the number of elements before k, which is $\sum_{i=1}^{k-1} i = \frac{i(i-1)}{2}$. Note that for $k = 1$, this value is 0.

Then, since the first element of group k is the next element, we obtain the location to be $f(k) = \frac{i(i-1)}{2} + 1$.

2. Give an algorithm (in pseudocode) for performing SEARCH(A, x) in this data structure. It should return the location i of x in $A[1..n]$ or 0, if x is not in $A[1..n]$. If it improves clarity, you can break up your algorithm into subprogams. Briefly explain how your algorithm works. Give the high level idea, NOT a line by line description of the pseudocode.

Solution

We first present SEARCH and then its subprograms.

SEARCH

Our SEARCH algorithm first applies a binary search on the groups using FIND-GROUP. This will return 0 if x does not fall in the range of any group, otherwise it returns the group index. After obtaining the group, SEARCH applies another binary search (SEARCH-ROTATED), this time inside the group returned. The details of specification is as follows.

- A: a list of rotated lists with n elements
- x: the key to search for
- return value: the location of x in A, or 0 if x is not in A

SEARCH(A, x)

```
1  group = FIND-GROUP(A[1..n], x)
2  if group == 0
3       return 0
4  return SEARCH-ROTATED(GROUP(A[1..n], group), x)
```

When performing binary searches on groups, we need to know at the final step the maximum and minimum of the final candidates. This is done by the following few subprograms. MAX-INDEX works by keeping track of the range of candidates with *left* and *right*.

MAX-INDEX

- R: a nonempty rotated list with attribute $R.length$
- $left, right$: the range of indices to consider ($1 \le left \le right \le R.length$), inclusive
- return value: the index of the maximum element in $R[left..right]$

MAX-INDEX$(R, left, right)$

```
 1  if left == right
 2      return R[1]
 3  if right − left == 1
 4      if R[left] > R[right]
 5          return left
 6      else
 7          return right
 8  center = ⌈(left + right)/2⌉
 9  if R[1] > R[center]
10      return MAX-INDEX(R, left, center)
11  else
12      return MAX-INDEX(R, center, right)
```

MIN-INDEX

- R: a nonempty rotated list with attribute $R.length$
- return value: the index of the minimum element in R

MIN-INDEX(R)

```
return R[(MAX-INDEX(R, 1, R.length) mod R.length) + 1]
```

For a nonempty rotated list R, let MIN(R) and MAX(R) be abbreviations for R[MIN-INDEX(R)] and R[MAX-INDEX$(R, 1, R.length)$], respectively.

GROUP

For easy access to groups given group index, we have the following short cut.

- k: the group index
- A: a list of rotated list that has at least k groups
- return value: group k of A as an array

GROUP(A, k)

```
1  if k(k+1)/2 > A.length
2        return A[k(k-1)/2+1...A.length]
3  return A[k(k-1)/2+1...k(k+1)/2]
```

SEARCH-ROTATED

- R: a rotated list with attribute $R.length$
- x: a key to search for
- $left, right$: the range (inclusive) of this search ($1 \leq left \leq right \leq R.length$)
- return value: the index of x in R if found, 0 otherwise

SEARCH-ROTATED$(R, left, right, x)$

```
 1  if right − left ≤ 1
 2        if R[left] == x
 3              return left
 4        elseif R[right] == x
 5              return right
 6        else
 7              return 0
 8  center = ⌊(left + right)/2⌋
 9  if R[left] < R[right]
10        if R[left] ≤ x ≤ R[center]
11              return SEARCH-ROTATED(R, left, center, x)
12        else
13              return SEARCH-ROTATED(R, center, right, x)
14  else
15        if R[center] ≤ x ≤ R[right]
16              return SEARCH-ROTATED(R, center, right, x)
17        else
18              return SEARCH-ROTATED(R, left, center, x)
```

FIND-GROUP

- A: a list of rotated list with attribute $A.length$ storing n and $A.g$ storing g.
- x: the value we need to find a group for
- return value: the index of the group where x can be found, or 0 if x cannot be found in any group

```
FIND-GROUP(A, x)
 1  if A.length == 0
 2       return 0
 3  left = 1
 4  right = A.g
 5  while right − left > 1
 6       center = ⌊(right + left)/2⌋
 7       if A[center(center − 1)/2 + 1] ≤ x
 8            left = center
 9       else
10            right = center
11  if MIN(GROUP(A, left)) ≤ x ≤ MAX(GROUP(A, left))
12       return left
13  elseif MIN(GROUP(A, right)) ≤ x ≤ MAX(GROUP(A, right))
14       return right
15  else return 0
```

3. Prove that the worst case time complexity of your algorithm is $\Theta(\log n)$.

Solution

Let integer comparisons (i.e $<, >, ==, \leq, \geq, \min, \max$) be the elementary operations we concern.

To show the worst case of our algorithm is $\Theta(\log n)$, it is enough to show 1) $T_{\text{SEARCH}}(n) \in O(\log n)$ and 2) $T_{\text{SEARCH}}(n) \in \Omega(\log n)$, where $T_{\text{SEARCH}}(n)$ is the maximum number of elementary operations SEARCH uses for input of length n.

1) We show $T_{\text{SEARCH}}(n) \in O(\log n)$ by showing every input of size n result in less than $c \log n$ steps for large enough $n \in \mathbb{N}$ and some $c \in \mathbb{R}^+$.

   First, we look at the two subprograms of SEARCH: FIND-GROUP and SEARCH-ROTATED.

   Lemma 1. MIN-INDEX, MAX-INDEX, MIN, *and* MAX *have worst case complexity* $O(\log n)$.

   Proof. Notice that MIN, MAX, and MIN-INDEX all use MAX-INDEX without any additional comparison calls, meaning that it is enough to prove MAX-INDEX have worst case complexity $O(\log(n))$, where $n = right - left$. We do so by finding an upperbound $U(n)$ of $T_{\text{MAX-INDEX}}$ and showing that $U \in O(\log(n))$.

   Lines 1 to 7 of MAX-INDEX show the base cases are handled with only one

comparison: $T_{\text{MAX-INDEX}}(0) = T_{\text{MAX-INDEX}}(1) = 1$. Then, from line 8 we know that

$$\max\{center - left, right - center\} = center - left = \lceil (right - left)/2 \rceil.$$

We define $U : \mathbb{N} \to \mathbb{N}$ so that U agrees with $T_{\text{MAX-INDEX}}$ on the base cases and $U(n) = U(\lceil n/2 \rceil) + 3 \geq T_{\text{MAX-INDEX}}(n)$ (+3 here to account for the comparisons on lines 1, 3, and 9). By Master Theorem, we have $U \in \Omega(\log n)$. Hence $T_{\text{MAX-INDEX}} \in O(\log n)$. □

Lemma 2. $T_{\text{FIND-GROUP}} \in O(\log n)$.

Proof. Notice that before line 3 our code handles the case of A being empty in one comparison. Assume now that $A.length > 0$, we show FIND-GROUP terminates in less than some constant multiple of $\log n$ comparisons. We observe that the while loop starting on line 5 runs at most $\log_{3/2}(A.g) + 1$ times. To see this, notice that on the ith iteration(for $i \geq 1$, $i = 0$ means before the loop), lines 8 and 10 ensure that

$$right_i - left_i \leq \left\lceil \frac{right_{i-1} - left_{i-1}}{2} \right\rceil = \frac{right_{i-1} - left_{i-1} + 1}{2} \leq \frac{right_{i-1} - left_{i-1}}{3/2}$$

when $right_{i-1} - left_{i-1} \geq 3$, and when $right - left < 3$, the loop runs for at most one time more. Hence, the while loop runs for at most $\log_{3/2}(A.g) + 1$ times, resulting in at most $2(\log_{3/2}(A.g) + 1)$ comparison calls. Since $g \leq n$, we know $T_{\text{FIND-GROUP}} \in O(\log n)$.

Lastly, lines 11 to 14 handle the final output in $c \log n$ number of operations (since MIN and MAX run in $\log n$ ti[illegible] by lemma above).

Combining everything, we can conclude that $T_{\text{FIND-GROUP}} \in O(\log n)$. □

Lemma 3. $T_{\text{SEARCH-ROTATED}} \in O(\log n)$.

Proof. Notice that SEARCH-ROTATED handles base cases in constant number of comparisons and makes only one recursive call of size at most $n/2$ ($\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ are equal for master theorem by our textbook). Therefore, this lemma can be proved using the same method as that used in lemma 1.

□

By lemmas, $T_{\text{FIND-GROUP}}, T_{\text{SEARCH-ROTATED}} \in O(\log n)$. We know that $T_{\text{SEARCH}} \leq T_{\text{FIND-GROUP}} + T_{\text{SEARCH-ROTATED}} + 1$ by the definition of SEARCH (the +1 accounts for the comparison on line 2). Thus, it is clear that $T_{\text{SEARCH}}(n) \in O(\log n)$.

2) We now prove $T_{\text{SEARCH}} \in \Omega(\log n)$ by showing that for large enough n, there is always some input with $A.length = n$ resulting in more than $\log_3(n) - 1$ comparison calls in the subprogram FIND-GROUP.

For $n \geq 3$, consider the input $A = (1, \ldots, n)$ and $x = n + 1$. On the ith iteration of the while loop (line 5),

$$right_i - left_i \geq \left\lfloor \frac{right_{i-1} - left_{i-1}}{2} \right\rfloor = \frac{right_{i-1} - left_{i-1} - 1}{2} \geq \frac{right_{i-1} - left_{i-1}}{3}$$

when $right_{i-1} - left_{i-1} \geq 3$. From this, we know that the loop runs for at least $\log_3(n) - 1$ steps (-1 to account for $right - left \geq 3$). Since every loop contains at least one comparison calls, we know that FIND-GROUP makes at least $\log_3(n) - 1$ comparison calls.

Since SEARCH uses FIND-GROUP, we know that $T_{\text{SEARCH}} \geq T_{\text{FIND-GROUP}}$. Hence, $T_{\text{SEARCH}} \in \Omega(\log n)$.

Combining 1) and 2) gives the result that $T_{\text{SEARCH}} \in \Theta(\log n)$.

4. Prove that your SEARCH algorithm is correct. Note that you may need to state and prove some additional lemmas.

Solution

Here, we will give proofs of partial correctness and termination together, first for subprograms, then for SEARCH.

Lemma 4. MAX-INDEX, MIN-INDEX, MIN, *and* MAX *terminate and return correct outputs.*

Proof. (a) We know that MIN, MAX are each simply a direct call to MIN-INDEX, which is a direct call to MAX-INDEX. Thus, to show termination, it is enough that we show it for MAX-INDEX.

Suppose the preconditions of MAX-INDEX are satisfied. We show this by complete induction on the size of $n = right - left$. For $n = 0, 1$, note that lines 2, 5, 7 terminate the program. Now, suppose $n \in \mathbb{N}$ with $n > 1$ and for all natural number $k < n$, MAX-INDEX terminates when $right - left = k$. Since $n > 1$, line 8 must execute. Note that by line 8, $\max\{center - left, right - center\} < n$, because $n > 1$. Thus, by induction hypothesis, both of the recursive calls on lines 10, 12 terminate. Therefore, MAX-INDEX terminates.

(b) Similarly, we first show partial correctness for MAX-INDEX. We do so by complete induction on $n = right - left$. For $n = 0$, there is exactly one key in $A[left..right]$. So line 2 indeed returns the correct result. Similarly, for $n = 1$, lines 5, 7 returns the index with maximum key. Now, suppose for $n > 1$ and for every $k < n$, MAX-INDEX returns the correct result. Since $n > 1$, we know that $right - center \leq$

center − *left* < n. We now show the two branches on resulting in lines 10, 12 are correct. First, if $R[1] > R[center]$, this means the maximum is in $A[1..center]$ (otherwise the sequence of keys from 1 to *center* should be increasing). So the call on line 10 is correct, and by hypothesis this returns the correct value. Similarly, if $R[1] < R[center]$, then the maximum key should be on the second half of the array. Hence, MAX-INDEX is partially correct.

For MIN-INDEX, the subprogram is simply a call to MAX-INDEX where 1 is added to the final index. This is correct since R is a rotated list: the minimum element follows the maximum element. The MIN and MAX, these are automatically correct since MIN-INDEX and MAX-INDEX are. □

Lemma 5. GROUP *is correct.*

Proof. This is a directly application of question 1, where we are simply slicing groups out from the array. □

Lemma 6. SEARCH-ROTATED *terminates and returns the correct output.*

Proof. (a) We first prove termination by complete induction on $n = R.length$. For $n = 0, 1$, termination is obvious from lines 1 to 7. Suppose now that $n > 1$ and SEARCH-ROTATED terminates on every size $k < n$ for $R.length$. By $n > 1$ and line 8, we know $center - left, right - center < n$. This ensures the recursive cases terminates (by induction hypothesis), showing the SEARCH-ROTATED terminates on size n inputs. By induction, we know SEARCH-ROTATED terminates always.

(b) We now show the correctness of SEARCH-ROTATED—again by complete induction $n = R.length$. The case where $n \leq 1$ is handled by lines 1 to 7 where we simply check if x is present. For $n > 1$, assume that SEARCH-ROTATED outputs correctly for all inputs of size $k < n$. We consider two cases. Case 1: $R[left] < R[right]$. Since R is a rotated list, we should search for x in the left half if x falls in between *left* and *right*: line 11 does this; we should search in the right half if $x <$ *left* or $x >$ *right*: line 13 does this. Case 2: $R[left] > R[right]$. This case almost exactly the same with the case 1, with recurrences on lines 16 and 18. By part a), we know $center - left, right - center < n$. This allows us to use our induction hypothesis that SEARCH-ROTATED outputs correctly for all inputs of size $k < n$. Therefore, SEARCH-ROTATED terminates correctly on inputs with size n. By induction, we are done.

Combining parts a) and b), we have shown SEARCH-ROTATED is correct. □

5. Give an algorithm (in pseudocode) for performing INSERT in this data structure in $O(\sqrt{n} \log n)$

time. Give precise specifications for your algorithm. Briefly explain how your algorithm works, why it is correct, and why it runs in $O(\sqrt{n}\log n)$ time.

Solution

The description of this algorithm will come after the pseudocode.

NEW-FIND-GROUP

- A: a nonempty list of rotated lists with property $A.length = n$ and has $A.g = g$ groups in it
- x: a key not in A that we are planning to insert
- return value: the group where x should be inserted

NEW-FIND-GROUP(A, x)

```
1  left = 1
2  right = A.g
3  while right − left > 1
4  center = ⌊(right + left)/2⌋
5  if GROUP(A, center)[0] < x
6        left = center
7  else
8        right = center
9  if x < MAX(GROUP(A, left))
10       return left
11 return right
```

INSERT

- A: a nonempty list of rotated lists with properties $A.n = n$, $A.s = s$, and has $A.g = g$ groups in it
- x: a key that we are planning to insert

```
INSERT(A, x)
 1  if A.n == 0 < A.s
 2       A[0] = 1
 3       A.n = A.n + 1
 4       A.g = 1
 5       return
 6  elseif A.n == A.s
 7       return
 8  if search(A, x) ≠ 0
 9       return
10  group-to-insert = NEW-FIND-GROUP(A, x)
11  prev-max = x
12  curr = group-to-insert
13  while curr ≤ g
14       curr-max-index = MAX-INDEX(GROUP(A, curr))
15       curr-max = GROUP(A, curr)[curr-max-index]
16       GROUP(A, curr)[curr-max-index] = prev-max
17       prev-max = curr-max
18       curr = curr + 1
19  A[A.n] = prev-max
20  A.n = A.n + 1
21  if A.n = A.g(A.g + 1)/2
22       A.g = A.g + 1
23       HEAPSORT(GROUP(A, group-to-insert))
24       HEAPSORT(GROUP(A, A.g))
```

The algorithm works by first eliminating trivial cases including: when A is empty/full, and when x is already in A. This is done in lines 1 to 9, and in $O(\log n)$ because SEARCH runs in $O(\log n)$. Then, the algorithm finds the group where x should be inserted by calling the subprogram NEW-FIND-GROUP which is a modified version of FIND-GROUP (so NEW-FIND-GROUP has worst case $\log n$ too). After this, INSERT replaces the maximum key in the group *group-to-insert* with x, and iteratively replaces the maximum key in group $k+1$ by the maximum in group k for *group-to-insert* $< k < g-1$. This is valid since every group contains elements strictly less than those in the next group, so the maximum of group k should be the minimum of group $k+1$.

We now compute time complexity of this process of shifting maximum keys. We know k full groups consist $k(k+1)/2$ elements. Setting this to n' and solving for positive k gives $k = \frac{1}{2} + \sqrt{-1/4 + 2n'}$. To account for n' where the last group is not full length, we ceil the expression to get $k = \lceil \frac{1}{2} + \sqrt{-1/4 + 2n'} \rceil \approx \sqrt{2n'}$ (we can do this because they give the same class of functions in big O). This means there are around $\sqrt{2n}$ groups, where the last group is of size $\sqrt{2n}$. For each of the $\sqrt{2n}$ groups, MAX-INDEX takes at most $O(\log \sqrt{2n})$ steps by lemma above, resulting in $O(\sqrt{2n} \log(\sqrt{2n}))$ steps.

Finally, before termination, INSERT sorts using HEAPSORT the first and last groups involved in this operation to ensure that the lists satisfy the data structure, specification. This takes $\sqrt{2n}O(\log(\sqrt{2n}))$ steps by CLRS.

Overall, the algorithm runs in

$$\log n + \sqrt{2n}\log\sqrt{2n} + 2\sqrt{2n}\log\sqrt{2n} \in O(\sqrt{n}\log n)$$

steps.

6. Give an algorithm (in pseudocode) for performing DELETE in this data structure in $O(\sqrt{n}\log n)$ time. Give precise specifications for your algorithm. Briefly explain how your algorithm works, why it is correct, and why it runs in $O(\sqrt{n}\log n)$ time.

Solution

- A: a list of rotated lists with property $A.length = n$ and has $A.g = g$ groups in it
- x: a key that we are planning to delete

```
DELETE(A, x)
 1  pos = SEARCH(A, x)
 2  gp = FIND-GROUP(A[1 : A.n], x)
 3  if pos == 0
 4      return
 5  prev = gp
 6  prev-min-index = pos − gp(gp − 1)/2
 7  while prev ≤ A.g − 1:
 8      curr = prev + 1
 9      curr-min-index = MIN-INDEX(GROUP(A, curr))
10      GROUP(A, prev)[prev-min-index] = GROUP(A, curr)[curr-min-index]
11      prev-min-index = curr-min-index
12      prev = curr
13  HEAPSORT(GROUP(A, gp))
14  pos = prev-min-index
15  while pos < A.n
16      next = pos + 1
17      GROUP(A, A.g)[pos] = GROUP(A, A.g)[next]
18      GROUP(A, A.g)[next] = NIL
19      pos = pos + 1
20  A.n = A.n − 1
21  if A.n < A.g(A.g − 1)/2:
22      A.g = A.g − 1
23  return
```

This algorithm works in a way similar to the INSERT algorithm.

First, the trivial case where x is not in A is handled by lines 1 to 4. This takes $O(2 \log n)$ steps because the use of SEARCH and FIND-GROUP.

Second, we obtain the precise location of x and iteratively replace the minimum key of group k by the minimum key of group $k+1$, for $gp < k < g-1$ (notice that in the first iteration, we deleted x and put the minimum of group $gp+1$ there). We can do this since every key in a group is less than that of the next group and they are arranged in a rotated list. Using the exact same reasoning as in the analysis of INSERT, we can see that this takes at most $O(\sqrt{2n} \log(\sqrt{2n}))$ steps.

Finally, the algorithm applies HEAPSORT to group gp to make sure the group is sorted (since x might not be the maximum of group gp). To make sure that there is no gap in the array, lines 15 to 19 shifts every key after the minimum key of group g forward. This takes $O(\sqrt{2n})$ steps since the size of the last group is at most size $\lceil \frac{1}{2} + \sqrt{-1/4 + 2n'} \rceil$ (recall from last question). And lastly, attributes n and g are updated appropriately.

Overall, the time complexity of DELETE is

$$2\log n + \sqrt{2n}\log(\sqrt{2n}) + \sqrt{2n} \in O(\sqrt{n}\log n).$$

7. What are the advantages and disadvantages of using this data structure to implement a DICTIONARY as compared to using a sorted array and an unsorted array?

Solution

We will compare the data structures one by one:

- **Unsorted Array**: For unsorted array, the advantage is at INSERT. Since the array is unsorted, it's enough that we directly append it to the end, taking $O(1)$ time. However, SEARCH must take linear time since there is no efficient way to eliminate candidates other than going through possibly the entire array. DELETE is similar since it requires SEARCH.
  In summary, unsorted array is better at insertion but worse at the other two operations as a DICTIONARY.
- **Sorted Array**: It is well-known that searching a sorted array takes $\Omega(\log n)$ time using binary search, which is equal to the time complexity of SEARCH we implemented above. However, for INSERT and DELETE, a sorted array takes $O(n)$ because one could insert to/delete from the first element in the sorted array, causing n shifts.
  In summary, sorted array is overall less efficient than our data structure.