CMSC 641, Design and Analysis of Algorithms, Spring 2010

Sandipan Dey,
Homework Assignment - 2

February 23, 2010

Problem 1 Solution

Part (a)

In the worst case, one needs to search all the sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where $k = \lceil lg(n+1) \rceil = \theta(lg\ n)$, with $n = \sum_{i=0}^{k-1} size(A_i) = \sum_{i=0}^{k-1} 2^i = 2^k - 1$.
The worst case time to perform binary search on the i^{th} sorted array
$= T(size(A_i)) = \lceil lg(size(A_i)) \rceil = \lceil lg(2^i) \rceil = i$

Hence, the worst case total SEARCH time
$= T(n) = T\left(\sum_{i=0}^{k-1} size(A_i)\right) = \sum_{i=0}^{k-1} T(size(A_i)) = \sum_{i=0}^{k-1} T(2^i)$
$= \sum_{i=0}^{k-1} i = \frac{k(k-1)}{2} = \theta(k^2) = \theta(lg^2 n)$.

Algorithm 1 Search element e from sorted k-array-set $A = \{A_0, \ldots, A_{k-1}\}$

SEARCH(A, e)
1: $i \leftarrow not\_found$. {array index}
2: **for** $r \leftarrow 0$ to $k-1$ **do**
3: $j \leftarrow$ BINSEARCH(A_r, e) {element index}
4: **if** $j \neq not\_found$ **then** {found!}
5: $i \leftarrow r$ {array index}
6: break
7: **end if**
8: **end for**
9: return $\{i, j\}$. {j^{th} element in i^{th} array}

Part (b)

INSERT Algorithm

Let's first establish a 1-1 correspondence between the INSERT in the set of arrays and INCREMENT in the binary counter problem. We must have
$size(A_i) = \left\{ \begin{array}{ll} 2^i & if\ n_i = 1 \\ 0 & if\ n_i = 0 \end{array} \right\}$

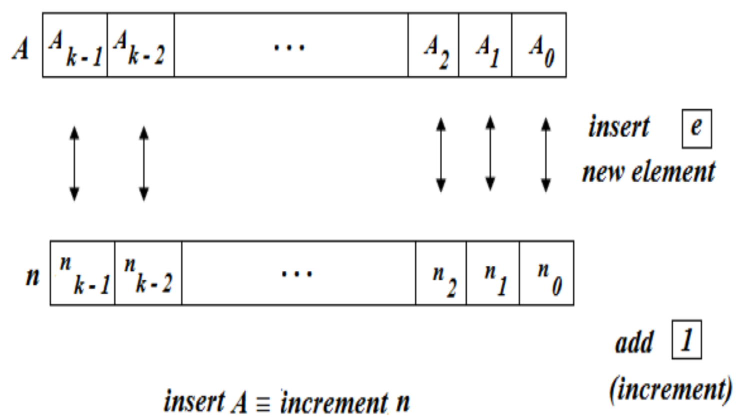

Figure 1: Equivalence of INSERT and INCREMENT for binary counter

The INSERT algorithm is described (with comparison to binary counter INCREMENT) in the following figure. It will insert element e in an array if $n < 2^k - 1$, i.e., all the k arrays are not already full. It starts by copying the element to an auxiliary array B and then replaces the array A_i by B if empty, otherwise merges sorted arrays A_i and B into B and goes to the next array A_{i+1} and repeats the same thing until it finds an empty array, staring from A_0.

Worstcase running time

Line $1 - 2$ of the INSERT algorithm is $\theta(1)$.
Merging two sorted arrays on line 5 of the algorithm takes $\theta(2^i + 2^i) = \theta(2^{i+1})$ time. Line 6 takes $\theta(1)$ time if we maintain an extra bit *empty* for each array A_i, which will be true if the array is empty, false otherwise. Line 7 is again $\theta(1)$.
Hence, running time lines $5 - 7$ is $\theta(2^{i+1})$ for each $i = 0, 1, \ldots k-2$.
In the worst case (when $n = 2^{k-1} - 1$), all the arrays $A_0, A_1, \ldots A_{k-2}$ will be full, hence $k-1$ merges will be needed (while loop $3-7$ will execute $k-1$ times), with the worstcase total merging time $= \sum_{i=0}^{k-2} \theta(2^{i+1}) = \theta\left(\sum_{i=0}^{k-2} 2^{i+1}\right) = \theta(2^k - 2)$.
Finally, line 9 involves copying / replacing the empty array A_i, in the worst

INCREMENT(n, k)	INSERT(A, e, k)
	$B \leftarrow \{e\}$ ▷ *auxiliary array B*
$i \leftarrow 0$	$i \leftarrow 0$
while $i < k$ and $n_i = 1$	**while** $i < k$ and A_i is *full* ▷ *if size*$(A_i) = 2^i$
do	**do**
	$B \leftarrow Merge(A_i, B)$ ▷ $size(B) \leftarrow 2^{i+1}$
$n_i \leftarrow 0$	*empty* A_i ▷ $size(A_i) \leftarrow 0$
$i \leftarrow i + 1$	$i \leftarrow i + 1$
if $i < k$	**if** $i < k$
then $n_i \leftarrow 1$	**then** $A_i \leftarrow B$ ▷ *replace* A_i, $size(A_i) \leftarrow 2^i$

Figure 2: Binary Counter INCREMENT vs Array INSERT algorithm

case the array A_{k-1} needs to be replaced, with running time $= \theta(2^{k-1})$. Hence, the worstcase total running time $= \theta(2^k - 2 + 2^{k-1}) = \theta(3.2^{k-1} - 2) = \theta(3.n - 2) = \theta(n)$, with $n = 2^{k-1} - 1$.

Also, we notice how the worstcase scenario for SEARCH differs from the same for INSERT.

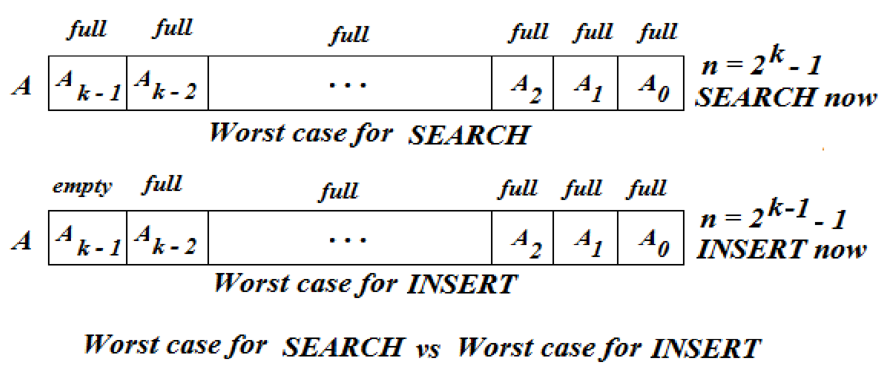

Figure 3: The Worst Case Scenarios

Amortized running time

Aggregate method

Let's consider a sequence of n INSERT operations, starting from all the k arrays empty, with $k = \lceil lg(n + 1) \rceil$.

First consider the merges inside the while loop. As we can see from the algorithm, the temporary array B has to be merged with the array $A_i \Leftrightarrow n_i$ flips from $1 \rightarrow 0$, $\forall i = 0, 1, \ldots, k-2$. But n_i flips from $1 \rightarrow 0$ only for $\left\lfloor \frac{n}{2^{i+1}} \right\rfloor$ times, $\forall i = 0, 1, \ldots, k-1$.

Since, merging of B and A_i takes $\theta(2^{i+1})$ time, the total merging ($B \leftarrow Merge(A_i, B)$) time for the sequence of n INSERT operations

$$= \sum_{i=0}^{k-2} \left\lfloor \frac{n}{2^{i+1}} \right\rfloor .\theta(2^{i+1}) = \sum_{i=0}^{k-2} n.\theta(1) = \theta(n.(k-1))$$
$$= \theta(n.(\lceil lg(n+1) \rceil - 1)) = \theta(nlg\ n).$$

Also, consider the copying of the temporary array B into A_i. This has to happen only when n_i flips from $0 \rightarrow 1$, which happens at most $\left\lceil \frac{n}{2^{i+1}} \right\rceil$ times and each copying operation takes $\theta(2^i)$ time.

Hence, the total copying ($A_i \leftarrow B$) time for the sequence of n INSERT operations

$$= \sum_{i=0}^{k-1} \left\lceil \frac{n}{2^{i+1}} \right\rceil .\theta(2^i) = \sum_{i=0}^{k-1} \frac{n}{2}.\theta(1) = \theta\left(\frac{n.k}{2}\right) = \theta(n.(\lceil lg(n+1) \rceil)) = \theta(nlg\ n).$$

Hence, the total amortized cost for sequence of all INSERT operations $= \theta(nlg\ n)$. The amortized cost per INSERT operation $= \theta(nlg\ n)/n = \theta(lg\ n)$.

Accounting method

We notice that during the sequence of operations an element can move on to array with higher index while merging and can never come back. Since there are k arrays, this transition from array with lower index to array with higher index for a given element can happen only for $k-1$ times. Hence the accounting analysis is as follows:

- Charge $k = \lceil lg(n+1) \rceil$ \$ to INSERT an element.
- Pay 1\$ for insertion immediately.
- Store rest of the $k-1$ charges to the element itself, so that it can always pay for future merges and transition from array A_i to A_{i+1}. But since such transition can happen at most for $k-1$ times, it always can pay for it.
- Hence amortized cost for each INSERT $= k = \theta(lg\ n)$.

Part (c)

DELETE Algorithm: Delete element e from k-array-set $A_0, \ldots, A_{k-1}$

- Call $SEARCH(A, e)$. Suppose it returns A_i, i.e., $e \in A_i$.
- $n \leftarrow size(A)$. Find the first non-zero bit j from right in n, i.e., find $j | n_j = 1$, $n_r = 0, \forall r < j$. It gives the first full array index. Let $e\prime \leftarrow$ the last element of A_j.
- $A_i \leftarrow A_i - \{e\} \cup \{e\prime\}$, i.e., remove e from A_i and put $e\prime$ into A_i . Then move $e\prime$ to its correct place in A_i.
- A_j is supposed to be with empty (since in $n_j = 1$, $n_r = 0, \forall r < j$, in $n-1$, j^{th} bit from the right will be 0 and all the following bits on the right will be 1, by binary counter DECREMENT) and all A_r with $r \leq j$ will be full. Hence, divide A_j (with $2^j - 1$ elements left): the 1st element goes into array A_0, the next 2 elements go into array A_1, the next 4 elements go into array A_2, and so forth. Mark array A_j as empty. The new arrays are created already sorted.

Runtime of DELETE

The worstcase running time of DELETE
$= \theta(lg^2 n)$ {SEARCH A_i}
$+ \theta(lg\ n)$ {Find 1st NonZero Bit j}
$+ \theta(n)$ {INSERT in sorted A_i in proper positon, linear time in $size(A_i) = 2^i$, worst case $2^{k-1} = \theta(n)$}
$+ \theta(n)$ {Copy A_j to lower index arrays, total number of elements to copy $= 2^j$, worst case $2^{k-1} = \theta(n)$}
$= \theta(n)$.

Problem 2 Solution

Part (a)

- Perform an IN-ORDER-WALK (call IN-ORDER-WALK(A, x, 0)) starting from node x, the output will be sorted (since output for all node n in IN-ORDER-WALK is by definition in the order $n_L \rightarrow n \rightarrow n_R$ and for a binary search tree $n_L.val < n.val < n_R.val$ by definition, here n_L and n_R denotes left-child and right child of node n respectively).
- Store the sorted output in the auxiliary storage (e.g., array A with size $\theta(size(x))$).
- Recursively find the MEDIAN of each interval and assign it to be the root of the current subtree using the construct_balanced_tree (divide and conquer) algorithm. Call construct_balanced_tree(A, x, 0, $size(x) - 1$).

Algorithm 2 IN-ORDER-WALK on a binary (search) tree rooted at *node*

IN-ORDER-WALK(A, $node$, i)

if node $\neq$ NULL **then**
    IN-ORDER-WALK(A, $node.left$, i)
    $A[i] \leftarrow node.val$
    $i \leftarrow i + 1$
    IN-ORDER-WALK(A, $node.right$, i)
end if

Algorithm 3 Constructs the $\frac{1}{2}$ balanced tree rooted at *node*

construct_balanced_tree(A, $node$, i, j)

if $i \leq j$ **then**
    $m \leftarrow \frac{i+j}{2}$.
    if $node = NULL$ **then**
        $node \leftarrow allocate\_node$
    end if
    $node.val \leftarrow A[m]$
    node.left $\leftarrow$ construct_balanced_tree(A, $node.left$, i, $m - 1$)
    node.right $\leftarrow$ construct_balanced_tree(A, $node.right$, $m + 1$, j)
end if
return node

Runtime of construct_balanced_tree

Let $n = size(A) = size(x)$. Then
$T(n) = 2T(n/2) + \theta(1) \Rightarrow T(n) = \theta(n^{lg_2 2}) = \theta(n)$, by Master theorem.
Hence, running time of construct_balanced_tree $= \theta(size(x))$. Similarly, runnig time of IN-ORDER-WALK is also $= \theta(size(x))$. Hence the runtime of the algorithm $= \theta(size(x))$.

Part (b)

For the worst case search time analysis in α-balanced binary search tree, we have the following:

$$n_L + n_R + 1 = n \ \wedge n_L \leq \alpha.n \ \wedge n_R \leq \alpha.n \Rightarrow max(n_L, n_R) \leq \alpha.n$$
$$T(n) \leq T(max(n_L, n_R)) + \theta(1) \leq T(\alpha.n) + \theta(1)$$

Hence, in the worst case, we have,

$T(n) = T(\alpha.n) + \theta(1) = T(\alpha.\alpha.n) + \theta(1) + \theta(1)$
$= \ldots = T(\alpha^k.n) + k.\theta(1)$, if $\alpha^k.n = 1 \Rightarrow n = \left(\frac{1}{\alpha}\right)^k \Rightarrow k = lg_{\frac{1}{\alpha}} n$.

$T(1) = 1 \Rightarrow T(n) = 1 + theta(k) = \theta\left(lg_{\frac{1}{\alpha}} n\right) = \theta\left(\frac{lg\ n}{lg\ \frac{1}{\alpha}}\right) = \theta(lg\ n)$.

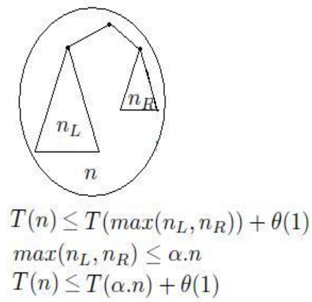

Figure 4: Runtime for SEARCH in α-balanced binary search tree

Part (c)

Define $\Delta(x) = |size(left(x)) - size(right(x))|$ and
the potential $\phi(T) = c. \sum_{x \in T:\Delta(x) \geq 2} \Delta(x)$, c sufficiently large.

By definition of potential since c is sufficiently large, $c > 0$, we have $\Delta(x) \geq 0 \Rightarrow \phi(T) \geq 0$ (mod function non-negative by definition).

For $\alpha = \frac{1}{2}$, we have the following:

$$
\begin{aligned}
size(left(x)) &\leq \frac{1}{2}.size(x),\ \forall x \in T \\
size(right(x)) &\leq \frac{1}{2}.size(x),\ \forall x \in T \\
size(left(x)) + size(right(x)) + 1 &= size(x),\ \forall x \in T \\
\Rightarrow size(left(x)) = size(x) - size(right(x)) - 1 &\geq size(right(x)) - 1 \\
\Rightarrow size(right(x)) - size(left(x)) &\leq 1 \\
\text{Similarly } size(left(x)) - size(right(x)) &\leq 1 \\
\Rightarrow |size(left(x)) - size(right(x))| &\leq 1 \\
\Rightarrow \Delta(x) &\leq 1,\ \forall x \in T \\
\Rightarrow |x \in T : \Delta(x) \geq 2| &= 0 \\
\Rightarrow \phi(T) &= 0
\end{aligned}
$$

Part (d)

Let's figure out the minimum possible potential in the tree that would cause us to rebuild a subtree of size-m rooted at x.

Now, x must not be α-balanced, otherwise we wouldnt need to rebuild the subtree. Let's say the left subtree is larger. Then, to violate the α-balanced criteria, we must have:

$size(left(x)) > \alpha m$.

Also, $size(left(x)) + size(right(x)) = m - 1$
$\Rightarrow size(right(x)) = m - 1 - size(left(x)) < m - 1 - \alpha.m = (1 - \alpha)m - 1$
$\Rightarrow \Delta(x) = |size(left(x)) - size(right(x))| > \alpha.m - ((1-\alpha)m-1) = (2\alpha-1)m+1$
$\Rightarrow \phi(T) = c. \sum_{x \in T : \Delta(x) \geq 2} \Delta(x) > c. ((2\alpha - 1)m + 1)$
(assuming $m \geq \frac{1}{2\alpha-1}$, we have, $\Delta(x) \geq 2$).

This potential must be at least equal to m units to pay for rebuilding the m node subtree (since we must have amortized cost providing an upper bound over the actual cost, i.e., $\hat{c_{rebuild}} \geq c_{rebuild} + \phi_i - \phi_{i-1}$, with actual cost m and amortized cost $O(1)$, we have, $O(1) \geq m + \phi_i - \phi_{i-1}$, with $\phi_{i-1} \geq m$, since the end potential ϕ_i is always greater than zero). Hence, we have
$\phi(T) > c. ((2\alpha - 1)m + 1) \geq m \Rightarrow c \geq \frac{m}{(2\alpha-1)m+1} = \frac{1}{(2\alpha-1)+\frac{1}{m}} > \frac{1}{2\alpha}$.

Hence if $c > \frac{1}{2\alpha}$, we can rebuild the subtree of size m in amortized cost $O(1)$.

Part (e)

- The amortized cost of the insert or delete operation in an n-node α- balanced tree is the actual cost plus the difference in potential between the two states.

- Search takes $O(lgn)$ time (as shown in part (b)) in an α-balanced tree, so the actual time to insert or delete will be $O(lgn)$.

- When we insert or delete a node x, we can only change the$\Delta(i)$ for nodes i that are on the path from the node x to the root. All other $\Delta(i)$ will remain the same since we dont change their subtree sizes. At worst, we will increase each of the $\Delta(i)$ for i in the path by 1 since we may add the node x to the larger subtree in every case. Again, as shown in part (b), there are $O(lgn)$ such nodes.

- The potential $\phi(T)$ can therefore increase by at most $c. \sum_{i \in path} 1 = O(clgn) =$ $O(lgn)$.

- So, the amortized cost for insertion and deletion is $O(lgn) + O(lgn) = O(lgn)$.

Problem 3 Solution

Out of total $m = 2n - 1$ operations, # of MAKE-SET operations $= n$ (since # of objects $= n$ to start with). Rest $n - 1$ operations can be arbitrary combinations of UNION and FIND-SET. Let's assume # of UNION operations =

k. ($0 \leq k \leq n-1$, since there are n objects to start with and each UNION decreases # of objects by exactly 1, hence there can be at most $n-1$ UNION operations). $\Rightarrow$ # of FIND-SET operations $= n-1-k = m-n-k$. Also, # of sets after sequence of k UNION operations are applied on n objects $= n-k$, with the largest possible set size $= k$.

Now, MAKE-SET and FIND-SET are $O(1)$ operations (since FIND-SET only needs to follow the representative pointer) and since each object can at most update its representative pointer for at most $O(lg\ k)$ times for sequence of k UNION operations with weighted union heuristics, total time for the entire sequence of operations
$= n.O(1) + (m-n-k).O(1) + k.O(lg\ k) = O(n+m-n-k+k\ lg\ k)$
$= O(m-k+k\ lg\ k) = O(m+k\ lg\ k)$
$\Rightarrow \exists\ constant\ c > 0$: total time for the sequence $\leq c\ (m+k\ lg\ k)$.

Operations	#	$Time/Operation$	$TotalTime$
MAKE-SET	n	$O(1)$	$n * O(1)$
UNION	k	$O(lg\ k)$	$k * O(lg\ k)$
FIND-SET	$m-n-k$	$O(1)$	$(m-n-k) * O(1)$
Total	m		$O(m+k\ lg\ k)$

Table 1: Set Operations

Now, let's assign the following charges and calculate the total amortized time for the entire sequence of m operations:

- MAKE-SET: $C\$$
- UNION: $C(log\ n\ +1)\$$
- FIND-SET: $C\$$

where C is a +ve constant and choose $C > c$.

Note that if we can show that the total amortized cost (time) provides an upper bound to the total actual time $\left(\sum_i \hat{c}_i \geq \sum_i c_i\right)$, we are done.

operations	#	charge per operation	total amortized cost
MAKE-SET	n	$C\$$	$n\$$
UNION	k	$C(lg\ n+1)\$$	$k(lg\ n+1)\$$
FIND-SET	$m-n-k$	$C\$$	$(m-n-k)\$$
Total	m		$C(m+k\ lg\ n)\$$

Table 2: Set Operations Amortized Costs

Now $n-1 \geq k \Rightarrow n > k \wedge C > c$
$\Rightarrow$ total amortized cost $= C(m+k\ lg\ n) > c(m+k\ lg\ k) \geq$ total actual cost,

which indeed provides an upper bound over the acutal cost.

Hence, the amortized costs for different operations are as claimed:

- MAKE-SET: $C\$ = O(1)$
- FIND-SET: $C\$ = O(1)$
- UNION: $C(log\ n\ + 1)\$ = O(lg\ n)$