ICS 691: Advanced Data Structures Spring 2016

Lecture 10

Prof. Nodari Sitchinava *Scribe: Kyle Berney*

1 Overview

In the last lecture we introduced the word-RAM model, the predecessor dictionary data structure, and the van Emde Boas Tree data structure [**?**], which we can use to implement a predecessor dictionary data structure. In the word-RAM model, each data cell can represent values up to w-bits, i.e., $w \geq \log n$. Hence, given a universe U containing all possible values in our predecessor dictionary data structure, we have that $|U| \leq 2^w$. For example: if $w = 2\log n$, then $|U| = n^2$ and $U = \{0, \ldots, n^2 - 1\}$. Furthermore, any single arithmetic operation defined by a C or Java programming language on these w-bit integers are free (i.e. constant time). Last time, our non-recursive van Emde Boas Tree contained a summary array of size $\sqrt{U}$ and $\sqrt{U}$ cluster arrays, each of size $\sqrt{U}$. Figure 1 below depicts an example of this non-recursive van Emde Boas Tree, with $|U| = 16$.

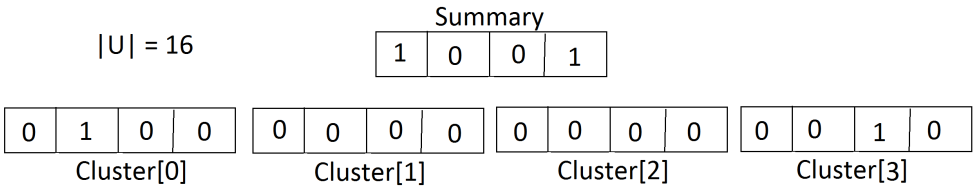

Figure 1: Depiction of the non-recursive van Emde Boas Tree.

Notice that any element $x \in U$ can be defined as $x = <c, i> = <\lfloor \frac{x}{\sqrt{U}} \rfloor, x\%\sqrt{U}> = c\sqrt{U} + i$, where c represents the cluster array the element x is in and i represents the index of the element x in the cluster array. For example, in Figure 1, we can represent element 14 as $14 = <3, 2>$, where $c = 3$ and $i = 2$.

In this lecture we will look at the recursively defined van Emde Boas Tree, and show that all predecessor dictionary operations have a runtime of $O(\log \log U)$.

2 van Emde Boas Tree

Let us define our recursive van Emde Boas Tree, denoted V, to be a van Emde Boas Tree such that each array is now a van Emde Boas Tree. In other words, our summary array and each cluster array will now be a van Emde Boas Tree with universe size of $\sqrt{U}$. Figure 2 below, depicts the structure of our recursive van Emde Boas Tree, V. Initially, we set $V.min = +\infty$ and $V.max = -\infty$. Our base case will be when the size of the universe is 2. Note that the element corresponding to $V.min$ is stored only in $V.min$ and nowhere else in the data structure, i.e., the element which is the minimum in V is not stored inside any of the clusters, instead it is stored only in $V.min$.

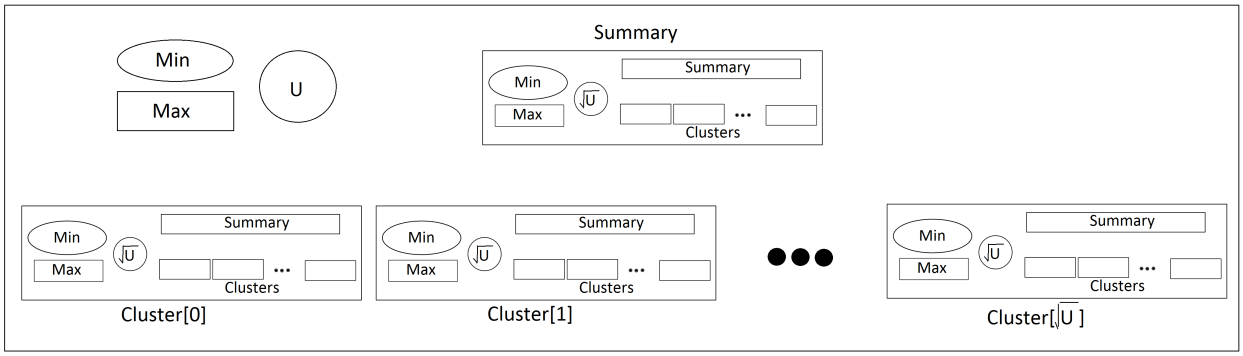

Figure 2: Depiction of the recursively defined van Emde Boas Tree.

Proposition 1. $T(U) = T(\sqrt{U}) + O(1) = O(\log \log U)$.

Proof. Let $m = \log U \Rightarrow U = 2^m$. Our recurrence relation is now: $T(2^m) = T(2^{\frac{m}{2}}) + O(1)$. Let $S(m) = T(2^m)$, then we have that: $S(m) = S(\frac{m}{2}) + O(1)$. By case 2 of the master method, $S(m) = O(\log m)$. Therefore, $T(U) = T(2^m) = S(m) = O(\log m) = O(\log \log U)$. [**?**] □

Thus, if we can implement our predecessor dictionary operations to have at most one recursive call on a problem size of $\sqrt{U}$, then the operation will run in $O(\log \log U)$ time.

2.1 Find(x, V)

Algorithm 1 Find the member $x = <c, i>$ in the data structure V

```
FIND(x, V)
return FIND(i, V.cluster[c])
```

To find an element x in V, recall from last lecture that we know that $x \in V$ if and only if $cluster[c][i] == 1$. For our recursively defined van Emde Boas Tree, this is equivalent to recursively calling our FIND function on the element i in $V.cluster[c]$. Clearly, we only have one recursive call on a problem size of $\sqrt{U}$ and thus, the runtime of FIND(x, V) is $O(\log \log U)$.

2.2 Predecessor(x, V)

Algorithm 2 Find the predecessor, if it exists, of the element $x = <c, i>$ in the data structure V

```
PREDECESSOR(x, V)
if x ≤ V.min then
  return NULL
end if
if x > V.max then
  return V.max
end if
if x > V.cluster[c].max then
  return V.cluster[c].max
else if i ≤ V.cluster[c].min then
  c' = PREDECESSOR(c, V.summary)
  return V.cluster[c'].max
else
  return PREDECESSOR(i, V.cluster[c])
end if
```

Recall that to find the predecessor on an element $x = <c, i>$, we look for the predecessor of x in the cluster c, if it exists, else we know that the predecessor of c is the maximum element in the first non-empty cluster with the largest index smaller than c. The general procedure is as follows:

1. If x is smaller than $V.min$, then we know that both x does not exist in V and the predecessor of x does not exist in V. Hence, we return NULL.
   $\Rightarrow O(1)$

2. If x is greater than $V.max$, then we know that the predecessor of x must be $V.max$. Hence, we return $V.max$.
   $\Rightarrow O(1)$

3. If x is greater than the maximum element in the cluster c, then we know that the predecessor is that maximum element in cluster c. Hence, we return $V.cluster[c].max$
   $\Rightarrow O(1)$

4. Else if $i \leq V.cluster[c].min$, then we know that the predecessor of x is not in the cluster c. Thus, we want to find the maximum element of the first non-empty cluster with the largest index smaller than c. To do this, we find the predecessor of c in the summary and then take the maximum element in that corresponding cluster. Hence, we first need one recursive call to find the predecessor of c in $V.summary$, then we return the maximum element in that cluster.
   $\Rightarrow$ One recursive call on a problem size of $\sqrt{U}$, plus $O(1)$ work

5. Else we know that the predecessor of x is in the cluster c. Hence, we only need one recursive call to PREDECESSOR$(i, V.cluster[c])$.
   $\Rightarrow$ One recursive call on a problem size of $\sqrt{U}$

Since steps 4 and 5 will never both be executed in the same function call, the recurrence for the runtime of PREDECESSOR(x, V) is $T(U) = T(\sqrt{U}) + O(1) = O(\log \log U)$.

2.3 Insert(x, V)

Algorithm 3 Insert the element $x = <c, i>$ into the data structure V

```
INSERT(x, V)
if V.min == +∞ then
  V.min = x
  return
else if x < V.min then
  SWAP(x, V.min)
end if
if x > V.max then
  V.max = x
end if
if V.cluster[c].min == +∞ then
  INSERT(i, V.cluster[c])
  INSERT(c, V.summary)
else
  INSERT(i, V.cluster[c])
end if
```

In order to insert $x = <c, i>$ into V, we need to insert i into $V.cluster[c]$ and insert c into $V.summary$ to mark that the cluster c is non-empty, if it is not already marked. We also need to make sure to update/maintain the maximum and minimum elements. The general procedure is as follows:

1. If V is empty, i.e., $V.min == +\infty$, then we only need to set $V.min = x$.
   $\Rightarrow O(1)$
2. Else if $x < V.min$, then x is the new minimum element in V. Hence, we swap x and $V.min$ and continue with the insertion with our new x value.
   $\Rightarrow O(1)$
3. If $x > V.max$, then x is the new maximum element in V. Unlike $V.min$, $V.max$ is also stored in the clusters, thus we simply set $V.max = x$ and proceed with the insertion.
   $\Rightarrow O(1)$
4. If $V.cluster[c]$ is empty, i.e., if $V.cluster[c].min == +\infty$, we need to insert i into $V.cluster[c]$ and also insert c into $V.summary$. Here we need to execute two recursive calls on a universe size of $\sqrt{U}$. However since $V.cluster[c]$ is empty, to insert i into $V.cluster[c]$ we only need to set $V.cluster[c].min = i$ which is $O(1)$ time.
   $\Rightarrow$ Two recursive calls on a problem size of $\sqrt{U}$, with one recursive call taking $O(1)$ time.
5. Else we just need to insert i into $V.cluster[c]$.
   $\Rightarrow$ One recursive call on a problem size of $\sqrt{U}$

Similar to PREDECESSOR(x, V), steps 4 and 5 will never both be executed in the same function call, therefore the recurrence for the runtime of INSERT(x, V) is $T(\sqrt{U}) + O(1) = O(\log \log U)$.

2.4 Delete(x, V)

Algorithm 4 Delete the element $x = <c, i>$ from the data structure V

```
DELETE(x, V)
if x == V.min then
  if V.max == −∞ then
    V.min = +∞
    return
  end if
else
  c_min = V.summary.min
  V.min = <c_min, V.cluster[c_min].min>
  x = <c, i> = <c_min, V.cluster[c_min].min>
end if
DELETE(i, V.cluster[c])
if V.cluster[c].min == +∞ then
  DELETE(c, V.summary)
end if
if V.summary.min == +∞ then
  V.max = −∞
else
  c_max = max{V.summary.max, V.summary.min}
  i_max = max{V.cluster[c_max].max, V.cluster[c_max].min}
  V.max = <c_max, i_max>
end if
```

In order to delete an element $x = <c, i>$ from V, we need to delete i from $V.cluster[c]$ and delete c from $V.summary$ if $V.cluster[c]$ is now empty after the deletion of i. We also have to make sure to update/maintain the minimum and maximum elements. The general procedure is as follows:

1. If x is $V.min$, then we have two cases:
   (a) If $V.max == -\infty$, then we know that there exists only 1 element in V. Hence, we want to set V back to its initial state and only need to set $V.min = +\infty$.
   $\Rightarrow O(1)$
   (b) Else, we need to find the new minimum element in V and delete it from its cluster. The deletion will be executed in step 3, hence, we only need to find this new minimum element set it to $V.min$ and continue on with the deletion procedure with the new minimum element as our x value.
   $\Rightarrow O(1)$
2. We now delete i from $V.cluster[c]$.
   $\Rightarrow$ One recursive call on a problem size of $\sqrt{U}$

3. If the cluster c is now empty, i.e., $V.cluster[c].min == +\infty$, then we need to mark cluster c as empty, i.e., we delete c from $V.summary$. This recursive call implies that the first recursive call in step 2 is $O(1)$ time, since to delete i from $V.cluster[c]$, we only need to set $V.cluster[c].min = +\infty$.
   $\Rightarrow$ One recursive call on a problem size of $\sqrt{U}$, and the recursive call from step 2 being $O(1)$ time

4. If $V.summary$ is now empty, i.e., if $V.summary.min == +\infty$, then similar to step 1(a) we want to set $V.summary$ back to its initial state and only need to set $V.max = -\infty$.
   $\Rightarrow O(1)$

5. Else we need to maintain/update $V.max$. Hence, we want to find the maximum element in V. To do this, we find the maximum cluster c and then the maximum element in the corresponding maximum cluster, and set that element as $V.max$.
   $\Rightarrow O(1)$

Thus, we have that the recurrence of the runtime of $\text{DELETE}(x, V)$ is $T(\sqrt{U}) + O(1) = O(\log \log U)$.

2.5 Sorting

In order to sort our items in the dictionary using our implementation of van Emde Boas Tree, we use the following procedure:

1. Initially start with the minimum element of V, i.e., $x = V.min$.

2. Append x onto the sorted list.

3. Set x to be the successor of the current element, i.e., $x = \text{SUCCESSOR}(x, V)$, and repeat step 2 until the successor of x is null.

Note that although we did not provide an implementation of the $\text{SUCCESSOR}(x, V)$ function, it is symmetric to the predecessor function, and thus runs in $O(\log \log U)$ time as well. Therefore, the runtime of the recurrence for $\text{SORT}(V)$ is $O(n) \cdot O(\text{SUCCESSOR}(x, V)) = O(n) \cdot O(\log \log U) = O(n \log \log U)$.

2.6 Reducing Space

Notice that our current implementation of the recursive van Emde Boas Tree takes up $O(|U|)$ memory space. In order to reduce this space, we can use a hash table instead of the array of clusters so that we do not need to store empty clusters. This results in $O(n)$ memory space, as we will only use space to store the items currently in the dictionary. [**?**]

References

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 3 edition, 2009.

[vEB75] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. *Proceedings of the 16th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 75–84, 1975.

[vEB77] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and space. *Information Processing Letters*, 6(3):80–82, 1977.