CLRS Chapter 20 van Emde Boas Trees Solutions

Contact: David Zhang – dzhang [at] cs [dot] toronto [dot] edu

20.1-1 Modify the data structures in this section to support duplicate keys.

Store a counter in each array position. On insert (or delete), increment (or decrement) the corresponding counter. All other operations differ only in checking whether a key is present in the dynamic set; k keys x are present iff $A(x) = k$.

20.1-2 Modify the data structures in this section to support keys that have associated satellite data.

Bundle satellite data into objects with a $.key$ field. Modify the operations to use $.key$, and store pointers to objects in array entries.

20.1-3 Observe that, using the structures in this section, the way we find the successor and predecessor of a value x does not depend on whether x is in the set at the time. Show how to find the successor of x in a binary search tree when x is not stored in the tree.

Search for x in the search tree and return the node at which the search path takes its final left (into the node's left child). If no such node exists, then x has no successor.

20.1-4 Suppose that instead of superimposing a tree of degree $\sqrt{u}$, we were to superimpose a tree of degree $u^{\frac{1}{k}}$, where $k > 1$ is a constant. What would be the height of such a tree, and how long would each of the operations take?

The tree will have k levels.

- **insert** $O(k)$ to update k levels.
- **delete** $O(ku^{\frac{1}{k}})$ for cascading deletes. We rise to the first non-empty summary block, which requires scanning $O(k)$ empty levels. The cost of 'rising' is insignificant compared to the cost of the scans, but is nevertheless paid for by the key's insert.
- **member** $O(1)$ to direct address.
- **min/max** $O(ku^{\frac{1}{k}})$ to take the leftmost (or rightmost) path, performing scans at each level to identify blocks to recurse into.
- **successor/predecessor** $O(ku^{\frac{1}{k}})$ to rise to the first non-empty summary block, and then descend the path to its min/max key.

20.2-1 Write pseudocode for the procedures Proto-vEB-Maximum and Proto-vEB-Predecessor.

```
Proto-vEB-Maximum($V$)
1  if $V.u = 2$
2      if $V.A(1) = 1$
3          return 1
4      elseif $V.A(0) = 1$
5          return 0
6      else return Nil
   else
7      $max - cls$ = Proto-vEB-Maximum($V.summary$)
8      if $max - cls$ is Nil
9          return Nil
       else
10         $ofst$ = Proto-vEB-Maximum($V.cluster(max - cls)$)
11         return $max - cls \cdot \sqrt{V.u} + ofst$
```

$$\begin{aligned} T(u) &= 2T(\sqrt{u}) + O(1) \\ &= T(2^m), \quad m = \log u \\ &= S(m), \quad S(m) = T(2^m) \\ &= 2S(m/2) + O(1) \\ &= \Theta(m) = \Theta(\log u) \end{aligned}$$

```
PROTO-vEB-PREDECESSOR($V, x$)
 1  if $V.u = 2$
 2      if $x = 1$ and $V.A(0) = 1$
 3          return 0
 4      else return NIL
    else
 5      $pred-ofst$ = PROTO-vEB-PREDECESSOR($V.cluster(\lfloor x/\sqrt{V.u}\rfloor),\ x \mod \sqrt{V.u}$)
 6      if $pred-ofst$ is not NIL
 7          return $\lfloor x/\sqrt{V.u}\rfloor \cdot \sqrt{V.u} + pred-ofst$
        else
 8          $pred-cls$ = PROTO-vEB-PREDECESSOR($V.summary, \lfloor x/\sqrt{V.u}\rfloor$)
 9          if $pred-cls$ is NIL
10              return NIL
            else
11              return $pred-cls \cdot \sqrt{V.u}$ + PROTO-vEB-MAXIMUM($V.cluster(pred-cls)$)
```

$T(u) = 2T(\sqrt{u}) + O(\log u) = \Theta(\log u \log\log u)$.

20.2-2 Write pseudocode for PROTO-vEB-DELETE. It should update the appropriate summary bit by scanning the related bits within the cluster. What is the worst-case running time of your procedure?

```
PROTO-vEB-DELETE($V, x$)
1  if $V.u = 2$
2      $V.A(x) = 0$
   else
3      PROTO-vEB-DELETE($V.cluster(\lfloor x/\sqrt{V.u}\rfloor),\ x \mod \sqrt{V.u}$)
4      if PROTO-vEB-MINIMUM($V.cluster(\lfloor x/\sqrt{V.u}\rfloor)$) is NIL
5          PROTO-vEB-DELETE($V.summary, \lfloor x/\sqrt{V.u}\rfloor$)
```

$T(u) = 2T(\sqrt{u}) + O(\log u) = \Theta(\log u \log\log u)$.

20.2-3 Add the attribute n to each proto-vEB structure, giving the number of elements currently in the set it represents, and write pseudocode for PROTO-vEB-DELETE that uses the attribute n to decide when to reset summary bits to 0. What is the worst-case running time of your procedure? What other procedures need to change because of the new attribute? Do these changes affect their running times?

```
PROTO-vEB-DELETE($V, x$)
1  if $V.u = 2$
2      $V.A(x), V.n = 0, V.A(0) + V.A(1)$
   else
3      $p = V.cluster(\lfloor x/\sqrt{V.u}\rfloor).n$
4      PROTO-vEB-DELETE($V.cluster(\lfloor x/\sqrt{V.u}\rfloor),\ x \mod \sqrt{V.u}$)
5      $V.n = V.n - (p - V.cluster(\lfloor x/\sqrt{V.u}\rfloor).n)$
6      if $V.cluster(\lfloor x/\sqrt{V.u}\rfloor).n = 0$
7          PROTO-vEB-DELETE($V.summary, \lfloor x/\sqrt{V.u}\rfloor$)
```

$T(u) = 2T(\sqrt{u}) + O(1) = \Theta(\log u)$. Insert will need to update the $.n$ field in proto-vEBs along the path to the newly inserted element. This incurs a constant overhead at each level of the recursion. Hence, insert retains its $O(\log u)$ performance. All other operations remain unchanged.

20.2-4 Modify the proto-vEB structure to support duplicate keys.

In each entry of A in each proto-vEB(2), store how many of that key the dynamic set currently contains instead of a boolean flag. Only the base cases of operations will need to change at no additional cost.

20.2-5 Modify the proto-vEB structure to support keys that have associated satellite data.

In each entry of A in each proto-vEB(2), store a pointer to the object containing the satellite data as well as a $.key$ field. Operations will need to change to use object keys but will otherwise function identically.

20.2-6 Write pseudocode for a procedure that creates a proto-vEB(u) structure.

```
1  $V.u = u$
2  if $V.u = 2$
3      $V.A = [0\ 0]$
   else
4      $V.summary$ = PROTO-vEB($\sqrt{u}$)
5      $V.cluster = array[0..\sqrt{u} - 1]$
6      for $i = 0$ to $\sqrt{u} - 1$
7          $V.cluster(i)$ = PROTO-vEB($\sqrt{u}$)
```

20.2-7 Argue that if line 9 of Proto-vEB-Minimum is executed, then the proto-vEB structure is empty.

If the set contains keys, then it will have a minimum-index cluster. No such cluster exists, so the set does not have keys.

20.2-8 Suppose that we designed a proto-vEB structure in which each cluster array had only $u^{\frac{1}{4}}$ elements. What would the running times of each operation be?

Each cluster array entry will point to a proto-vEB($u^{\frac{3}{4}}$) structure.

- **insert** $T(u) = T(u^{\frac{1}{4}}) + T(u^{\frac{3}{4}}) + O(1) \leq 2T(u^{\frac{3}{4}}) + O(1) = O((\log u)^{\log_{4/3} 2})$ for inserts into the summary and cluster proto-vEBs.
- **member** $T(u) = T(u^{\frac{3}{4}}) + O(1) = \Theta(\log\log u)$.
- **min/max** $T(u) \leq 2T(u^{\frac{3}{4}}) + O(1) = O((\log u)^{\log_{4/3} 2})$ for min/max operations in the summary and cluster proto-vEBs.
- **successor/predecessor** $T(u) \leq 2T(u^{\frac{3}{4}}) + O((\log u)^{\log_{4/3} 2}) = O((\log u)^{\log_{4/3} 2} \log\log u)$ for successor /predecessor operations in the cluster and summary proto-vEBs, plus 1 min/max operation in a cluster proto-vEB.
- **delete** $T(u) \leq 2T(u^{\frac{3}{4}}) + O((\log u)^{\log_{4/3} 2}) = O((\log u)^{\log_{4/3} 2} \log\log u)$ for deletes in the summary and cluster proto-vEBs, plus by 1 min/max operation in a cluster proto-vEB.

20.3-1 Modify vEB trees to support duplicate keys.

Modify the base vEB(2) structures to store the number of keys of $.min$ or $.max$ value that the structure currently contains in separate $.nmin$, $.nmax$ fields. Add a $.nmin$ field to vEB(u) structures, for $u > 2$. The insert and delete operations will need to increment or decrement these counts, passing $.nmin$ and $.nmax$ values up the recursion chain. Only when count(s) reach 0 will an update to the $.min/.max$ fields be necessary. We store a $.nmin$ field in all vEBs because the key stored in the $.min$ field is not recursively stored.

20.3-2 Modify vEB trees to support keys that have associated satellite data.

Use the $.key$ field in all operations. Store a pointer to the object in vEB(2) structures. For larger vEBs, store pointers to the objects in both the $.min$ and $.max$ fields. This is particularly important for the $.min$ field since the minimum key is not recursively stored.

20.3-3 Write pseudocode for a procedure that creates an empty van Emde Boas tree

```
$V.u, V.min, V.max = u$, NIL, NIL
if $V.u > 2$
    $V.summary$ = vEB($^{\uparrow}\sqrt{u}$)
    $V.cluster = array[0..^{\uparrow}\sqrt{u} - 1]$
    for $i = 0$ to $^{\uparrow}\sqrt{u} - 1$
        $V.cluster(i)$ = vEB($^{\downarrow}\sqrt{u}$)
```

20.3-4 What happens if you call vEB-Tree-Insert with an element that is already in the vEB tree? What happens if you call vEB-Tree-Delete with an element that is not in the vEB tree? Explain why the procedures exhibit the behavior that they do. Show how to modify vEB trees and their operations so that we can check in constant time whether an element is present.

Inserting a duplicate element will have no effect unless the element is the minimum. If the element is the minimum, then subsequent inserts will store copies at various levels of the tree. As a result, deleting the minimum key from the set requires at least as many deletes as there are copies of the minimum. Deleting an element that is not in the tree will have no effect. At the cost of $O(u)$ space, we can maintain a boolean array A, where $A(x) = 1$ iff key x is currently present in the dynamic set. Only insert x if $A(x) = 0$, and only delete x if $A(x) = 1$.

20.3-5 Suppose that instead of $^{\uparrow}\sqrt{u}$ clusters, each with universe size $^{\downarrow}\sqrt{u}$, we constructed vEB trees to have $u^{\frac{1}{k}}$ clusters, each with universe size $u^{1-\frac{1}{k}}$, where $k > 1$ is a constant. If we were to modify the operations appropriately, what would be their running times? For the purpose of analysis, assume that $u^{\frac{1}{k}}$ and $u^{1-\frac{1}{k}}$ are always integers.

$$
\begin{aligned}
& T(u) \le T(u^{1-\frac{1}{k}}) + O(1) \\
\implies & T(2^m) \le T(2^{m-\frac{m}{k}}) + O(1),\ m = \log u \\
\implies & S(m) \le S\left(\frac{m}{\frac{k}{k-1}}\right) + O(1),\ S(m) = T(2^m) \\
\implies & S(m) = O(\log m) = O(\log\log u)
\end{aligned}
$$

20.3-6 Creating a vEB tree with universe size u requires $O(u)$ time. Suppose we wish to explicitly account for that time. What is the smallest number of operations n for which the amortized time of each operation in a vEB tree is $O(\lg\lg u)$?

The total cost of initialization and performing n operations is $O(u + n\log\log u)$. We want to spread $O(u)$ across the n operations. That is, we want $\frac{u}{n} = \log\log u$. Having $n = \Omega\left(\frac{u}{\log\log u}\right)$ operations achieves this.

20.1 a) Explain why the following recurrence characterizes the space requirement $P(u)$ of a van Emde Boas tree with universe size u: $P(u) = (\sqrt{u} + 1)P(\sqrt{u}) + \Theta(\sqrt{u})$ (20.5).

Each vEB(u) contains vEB($\sqrt{u}$) structures for $\sqrt{u}$ clusters and 1 summary block. Each also stores a cluster array of size $\sqrt{u}$.

20.1 b) Prove that recurrence (20.5) has the solution $P(u) = O(u)$.

Proceed via substitutions. Eventually, we derive the following expression:

$$\begin{aligned} P(u) &= (\sqrt{u}+1)P(\sqrt{u}) + \Theta(\sqrt{u}) \\ &= \prod_{k=1}^{i}(u^{\frac{1}{2^k}}+1)P(u^{\frac{1}{2^i}}) + \sum_{k=1}^{i}\Theta(u^{\frac{1}{2^k}})\cdot\prod_{j=1}^{k-1}(u^{\frac{1}{2^j}}+1) \end{aligned}$$

We enter our base case when the universe size of our vEBs is $u = 2$. That is, when $u^{\frac{1}{2^i}} = 2$. Hence, $i = \log_2\log_2 u$ and $P(2) = O(1)$.

$$= \prod_{k=1}^{\log_2\log_2 u}(u^{\frac{1}{2^k}}+1) + \sum_{k=1}^{\log_2\log_2 u}\Theta(u^{\frac{1}{2^k}})\cdot\prod_{j=1}^{k-1}(u^{\frac{1}{2^j}}+1)$$

We will solve for each term separately using sums of geometric series.

$$\begin{aligned} \prod_{k=1}^{\log_2\log_2 u}(u^{\frac{1}{2^k}}+1) &= \sum_{k=0}^{2^{\log_2\log_2 u}-2}(u^{\frac{1}{2^{\log_2\log_2 u}}})^k + u^{\frac{2^{\log_2\log_2 u}-1}{2^{\log_2\log_2 u}}} \\ &= \frac{1-(u^{\frac{1}{2^{\log_2\log_2 u}}})^{2^{\log_2\log_2 u}-1}}{1-(u^{\frac{1}{2^{\log_2\log_2 u}}})} + u^{\frac{2^{\log_2\log_2 u}-1}{2^{\log_2\log_2 u}}} \\ &= u-1 \end{aligned}$$

We will approximate the following term by exploiting log rules, the largeness of u, and the fact that $k \le \log_2\log_2 u$. In particular, we make use of $u^{\frac{1}{2^j}} \gg 1$, and as a corollary, the fact that $\log(u^{\frac{1}{2^j}}+1) \approx \log u^{\frac{1}{2^j}} = \frac{1}{2^j}\log u$.

$$\begin{aligned} \log\prod_{j=1}^{k-1}(u^{\frac{1}{2^j}}+1) &= \sum_{j=1}^{k-1}\log(u^{\frac{1}{2^j}}+1) \\ &\approx \log u\sum_{j=1}^{k-1}\frac{1}{2^j} \\ &= (1-\frac{1}{2^{k-1}})\cdot\log u \\ \implies \prod_{j=1}^{k-1}(u^{\frac{1}{2^j}}+1) &\approx u^{1-\frac{1}{2^{k-1}}} \end{aligned}$$

We will use this approximation to simplify the second summand.

$$\sum_{k=1}^{\log_2\log_2 u}\Theta(u^{\frac{1}{2^k}})\cdot\prod_{j=1}^{k-1}(u^{\frac{1}{2^j}}+1) = \sum_{k=1}^{\log_2\log_2 u}\Theta(u^{\frac{1}{2^k}})\cdot u^{1-\frac{1}{2^{k-1}}}$$

$\log_2\log_2 u = o(u^{1-\frac{1}{2^{k-1}}})$, and the terms in our sum asymptotically increase.
Thus, the dominant bound (at $k = \log_2\log_2 u$) will dominate the whole sum.

$$\begin{aligned} &= \Theta(u^{1-\frac{1}{\log_2 u}}) \\ &= \Theta(u) \end{aligned}$$

Hence, $P(u) = u - 1 + \Theta(u) = \Theta(u)$.

20.1 c) Modify the vEB-Tree-Insert procedure to produce pseudocode for the procedure RS-vEB-Tree-Insert(V, x), which inserts x into the RS-vEB tree V, calling Create-New-RS-vEB-Tree as appropriate.

```
RS-vEB-Tree-Insert($V, x$)
 1  if $V.u = 2$
 2      brute force
    else
 3      if $V.min$ is Nil
 4          $V.min = V.max = x$
        else
 5          if $x < V.min$
 6              $x, V.min = V.min, x$
 7          if $x > V.max$
 8              $V.max = x$
 9          if $V.cluster(\lfloor x/^{\downarrow}\sqrt{u} \rfloor)$ is Nil
10              if $V.summary$ is Nil
11                  $V.summary$ = Create-New-RS-vEB-Tree($^{\uparrow}\sqrt{u}$)
12              RS-vEB-Tree-Insert($V.summary, \lfloor x/^{\downarrow}\sqrt{u} \rfloor$)
13              $V.cluster(\lfloor x/^{\downarrow}\sqrt{u} \rfloor)$ = Create-New-RS-vEB-Tree($^{\downarrow}\sqrt{u}$)
14              $V.cluster(\lfloor x/^{\downarrow}\sqrt{u} \rfloor).max = V.cluster(\lfloor x/^{\downarrow}\sqrt{u} \rfloor).min = x \mod {}^{\downarrow}\sqrt{u}$
          else
15              RS-vEB-Tree-Insert($V.cluster(\lfloor x/^{\downarrow}\sqrt{u} \rfloor), x \mod {}^{\downarrow}\sqrt{u}$)
```

20.1 d) Modify the vEB-Tree-Successor procedure to produce pseudocode for the procedure RS-vEB-Tree-Successor(V, x), which returns the successor of x in the RS-vEB tree V, or Nil if x has no successor in V.

```
RS-vEB-Tree-Successor($V, x$)
 1  if $V.u = 2$
 2      brute force
    else
 3      if $V.min$ is Nil or $x \geq V.max$
 4          return Nil
 5      if $x < V.min$
 6          return $V.min$
 7      if $V.cluster(\lfloor x/^{\downarrow}\sqrt{u} \rfloor)$ is not Nil and $x \mod {}^{\downarrow}\sqrt{u} < V.cluster(\lfloor x/^{\downarrow}\sqrt{u} \rfloor).max$
 8          return $\lfloor x/^{\downarrow}\sqrt{u} \rfloor \cdot^{\downarrow} \sqrt{u}$ + RS-vEB-Tree-Successor($V.cluster(\lfloor x/^{\downarrow}\sqrt{u} \rfloor), x \mod {}^{\downarrow}\sqrt{u}$)
      else
 9          $suc - cls$ = RS-vEB-Tree-Successor($V.summary, \lfloor x/^{\downarrow}\sqrt{u} \rfloor$)
10          if $suc - cls$ is Nil
11              return Nil
            else
12              return $suc - cls \cdot^{\downarrow} \sqrt{u} + V.cluster(suc - cls).min$
```

20.1 e) Prove that, under the assumption of simple uniform hashing, your RS-vEB-Tree-Insert and RS-vEB-Tree-Successor procedures run in $O(\lg \lg u)$ expected time.

Hash table operations take $O(1)$ in expectation and our recurrences are as before: $T(u) \leq T(^{\uparrow}\sqrt{u}) + O(1)$. Hence, $T(u) = O(\log \log u)$ in expectation.

20.1 f) Assuming that elements are never deleted from a vEB tree, prove that the space requirement for the RS-vEB tree structure is $O(n)$, where n is the number of elements actually stored in the RS-vEB tree.[1]

Notice that each key is either the minimum of the dynamic set and is stored at the top level or is not and will have its bits recursively stored in a cluster. The number of levels that a key occupies obeys the recurrence: $L(u) \leq L(u^{\frac{1}{2}}) + O(1)$. As we have seen, this resolves to $L(u) = O(\log\log u)$, so each key is present in $O(\log\log u)$ levels of the tree. Hence, the space requirement to store n keys is $O(n \log\log u)$.

We will prove a matching lower bound by providing a family of inputs[2], on which each key occupies $\Omega(\log\log u)$ levels, amortized – thus requiring $\Omega(n \log\log u)$ space.

Intuition: The problem with an amortization argument is that we do not properly account for the summary 'block.' It is insufficient to charge each summary pointer to the key stored in the $.min$ field because the summary *tree* can be much larger than the cluster tree if we use many clusters, but each cluster is sparse. If we have sparse clusters, then it turns out that the number of nodes in the summary tree that we use to represent each cluster, on average, will be $\approx n$. If we desire an amortized cost of $\Omega(\log\log u)$ per key, then we want to use $\Omega(\log\log u)$ clusters in the root. Then, our total space requirement will be $\Omega((\log\log u)^2) = \Omega(n \log\log u)$, as required.

Define a family of inputs $S_u = \{2^{\log_2 u - 2^i} : i \in \{0, \ldots, \log_2 \log_2 u\}\}$. This family preserves 3 key properties over all $u = 2^{2^k}$:

- $1 \in S_u$;
- each key in S_u belongs to a distinct cluster in RS-vEB(u);
- taking the top $\frac{1}{2}\log_2 u$ bits from each element produces $S_{u^{1/2}}$.

Writing both the bit and decimal representations of elements is useful for developing intuition. Notice that an immediate corollary is that $|S_u| = |S_{u^{1/2}}| + 1$, since 1 will be stored in $.min$ of RS-vEB(u) and each of the remaining keys is partitioned to a distinct cluster, therefore requiring its high bits to be stored in the summary tree. The resultant tree is a caterpillar, which we traverse through by following summary pointers. Each body segment requires Ω(# clusters) words, hence the total space requirement is $\sum_{i=0}^{\log_2\log_2 u - 1} \Omega(\log\log u - i) = \sum_{i=1}^{\log_2\log_2 u} \Omega(i) = \Omega((\log\log u)^2) = \Omega(n \log\log u)$. Thus, our $\Theta(n \log\log u)$ bound is tight.

20.1 g) RS-vEB trees have another advantage over vEB trees: they require less time to create. How long does it take to create an empty RS-vEB tree?

$O(1)$.

20.2 a) How much space does this structure require?

$\Theta(n \log u)$ for the hash table of keys and bitstrings.

20.2 b) Show how to perform the Minimum and Maximum operations in $O(1)$ time; the Member, Predecessor, and Successor operations in $O(\lg\lg u)$ time; and the Insert and Delete operations in $O(\lg u)$ time.

Intuition: Suppose that we have a u-bit bitmap, where the bit in each position from 0 to $u - 1$ is 1 iff the key corresponding to its position is currently present. Build a binary tree over bitmap entries, such that each leaf is a bit in the bitmap, and each inner node is the logical OR of its children. This tree will have height $O(\log u)$. Additionally, store all currently present keys in sorted order in a doubly linked list. This gives us the following:

- **member** $O(1)$ to direct address.
- **min/max** $O(1)$ to access the head/tail of the linked list.
- **insert/delete** $O(\log u)$ to flip bits in the path from the root to the key's leaf.
- **successor/predecessor** $O(\log\log u)$. If the key is in the set, then return the next (or previous) key in the linked list in $O(1)$; otherwise, binary search for the lowest 1 in the path from the root to the key. If the key is in 1's left subtree, then the minimum of 1's right subtree is the successor. If the key is in the right subtree, then the maximum of 1's left subtree is its predecessor. If we are seeking the key's successor but have its predecessor, then find the successor by taking the next element following the predecessor in the linked list. The same applies to finding the predecessor, given the successor.

[1]In the errata, it is mentioned that this bound should instead be $\Theta(n \lg n)$. We will prove a universally adopted result by Vladimír Čunát: $\Theta(n \log\log u)$.

[2]Note that the order of insertion does not matter, i.e. the structure of the tree is determined entirely by the keys that it stores.

This is **not** the structure that we are given. However, the structure that we *are* given is a space-optimized version of this data structure – in particular, it only stores '1' entries. Have the hash table map each base 10 number to its full binary representation. Map each full binary representation to a pointer to its linked list entry. Map each partial binary representation to the full binary representations of the maximum and minimum elements in its subtree.

- **member** $O(1)$ expected to search the hash table.
- **min/max** $O(1)$ to access the head/tail of the linked list.
- **successor/predecessor** $O(\log\log u)$. If the key is in the set, then follow the pointers to the linked list in $O(1)$. Otherwise, binary search via bitshifts on the key's bitstring. Then, having obtained the successor or predecessor, follow the same procedure as outlined previously.
- **insert/delete** $O(\log u)$ to add/update/delete binary prefixes to maintain the hash table invariants. If the minimum and maximum of a partial bit representation's subtree are the same, then it only contains that key and can be deleted. It is possible to do deletes efficiently by using the maximum (or minimum) of subtrees. Call successor or predecessor to obtain a pointer to the linked list and insert/delete in $O(\log\log u)$. Then, add or remove the key from the hash table.

20.2 c) Show that a y-fast trie requires only $O(n)$ space to store n elements.

The x-fast trie is over $\Theta(\frac{n}{\log u})$ elements and the binary search trees cumulatively store $\Theta(n)$ elements. Hence, the total space requirement is $\Theta(\frac{n}{\log u}\log u + n) = \Theta(n)$.

20.2 d) Show how to perform the Minimum and Maximum operations in $O(\lg\lg u)$ time with a y-fast trie.

Call min (or max) in the x-fast trie to retrieve the representative that stores the min (or max) key. Then, traverse the leftmost (or rightmost) path in the balanced binary search tree in $O(\log\log u)$. Operations in the x-fast trie require $O(1)$ time so the total time requirement is $O(\log\log u)$.

20.2 e) Show how to perform the Member operation in $O(\lg\lg u)$ time.

Call successor in the x-fast trie to find the smallest representative $\geq$ key. If the representative is Nil, then the key is not in the trie. Otherwise, the key belongs to the corresponding binary search tree. The call to obtain the predecessor in the x-fast trie, and searching in the binary search tree takes $O(\log\log u)$.

20.2 f) Show how to perform the Predecessor, and Successor operations in $O(\lg\lg u)$ time.

As before, with $O(1)$ calls into the x-fast trie. To find the predecessor of a key, find the maximum representative $\leq$ key and search for the maximum inside its binary search tree. If the maximum is the key, then find the predecessor in the binary search tree. If no such predecessor exists, then move to the next smallest representative and return the maximum of its binary search tree. This takes time $O(\log\log u)$. Finding the successor is analogous.

20.2 g) Explain why the Insert and Delete operations take $\Omega(\lg\lg u)$ time.

An insert/delete to a balanced binary search tree of $\Theta(\log u)$ items will take $\Omega(\log\log u)$.

20.2 h) Show how to relax the requirement that each group in a y-fast trie has exactly $\lg u$ elements to allow Insert and Delete to run in $O(\lg\lg u)$ amortized time without affecting the asymptotic running times of the other operations.

Allow each balanced binary search tree to contain between $\frac{1}{2}\log u$ and $2\log u$ keys, and set the representative of the first binary search tree to be $u-1$.

- If an insert causes a tree to exceed $2\log u$ keys, then split it. Take the maximums of the binary trees as their representatives unless the tree's previous representative was $u-1$. In that case, take the maximum of the tree with smaller keys as its representative while keeping the representative of the larger-keyed tree as $u-1$. Thus, an additional insert to the x-fast trie is required, taking time $O(\log u)$. But since $\Theta(\log u)$ inserts were needed to split the tree, we only need to charge each insert an additional $O(1)$ to account for the overhead of adding a representative and splitting the tree.
- If a delete causes a tree to shrink below $\frac{1}{2}\log u$ keys, then merge it with a neighbouring tree and delete the smaller representative from the x-fast trie. If the merged tree has $> 2\log u$ keys, then split it once more and add a representative to the x-fast trie as previously described. Delete performs at most 2 $O(\log u)$ operations in the x-fast trie, but only occurs after $\Theta(\log u)$ operations. Hence, charging each delete an additional $O(1)$ is sufficient to pay for splits, joins and x-fast trie operations.