CS 6968: Data Str & Alg Scalable Comp	Spring 2023
Lecture 1 — Jan. 11, 2023	
Prof. Prashant Pandey	*Scribe: Hunter McCoy*

1 Overview

In the last lecture we covered the syllabus.

In this lecture we will cover **Van Emde Boas Trees**, a data structure that offers $O(\log \log u)$ performance for insertions, queries, deletions, and successor and predecessor scans.

2 Terms and Setup

The data structures considered today contain a set of n items pulled from a universe U, so $|S| = \text{n}$.

The Universe is the total set of items possible in the current representation. For example, the universe of integers is of size $2^32 = 4,294,967,296$, as that is how many different integers are possible. In the same vein, the universe of uint64_ts has size $|U| = 2^64$ and for chars it has size $2^8 = 256$.

The operations we will look at today are:

add(x): $S = S \Cup x$

delete(x): $S = S \backslash x$

successor (x): successor search returns smallest item in the set larger or equal to the input.

predecessor (x): predecessor search returns the largest item smaller or equal to the input.

find(x): return whether $x \in S$

2.0.1 Some Relevant Terms

Amortized: Average amount of work per item - Used mainly in cases where for most items we do very little work, but occasionally an item must do a lot of work. One example is the updating of a Fibonacci tree.

Expectation: Average case of a random variable E[x]. This is a different bound from W.H.P. as it doesn't account for variance or skew.

2.1 Comparing some data structures

Data Struct	Successor/Predeccesor complexity	Space Usage
BST	$\log(n)$	$O(n)$
Heap	$\log(n)$	$O(n)$
Skip List	$\log(n)$ in expectation	$O(n)$
Hash Table	$O(n)$	$O(n)$

3 Van Emde Boas Tree

The Van Emde Boas Tree (vEB Tree) is a tree structure built on an associative array.

3.1 Performance Bounds

The main benefit of the vEB tree is its fast theoretical speed. All of the above operations occur in $O(\log \log U)$ - This is good when good when $|S| \approx |U|$, or n is close to the size of the universe.

More recent versions like the x-fast trie boast space usage $O(n \log \log U)$ or even $O(n)$ space usage. The version we will go over uses $O(U)$ space.

3.2 Equations

if $|U| = n^{o(1)}$ or $U = n^{(\log n)^{o(1)}}$ then $\log \log U = \mathrm{O}(\log \log n)$.

Binary search: $T(k) = T(k/2)+o(1)$ - remember the master theorem! this has $\log k$ time complexity.

let $k = \log U$, then $\mathrm{T}(\log u) = T(\frac{logU}{2}) + o(1)$.

$T(u) = T(\sqrt{U}) + o(1)$ - need to shrink by an exponential amount on every recursive step to get $\mathrm{O}(\log \log U)$ performance.

4 Solution attempts

4.1 Attempt 1: Bitvectors

Create a bit vector with U bits - for every item in the universe you have a bit for present or not

This gives us Add/Delete in O(1), but successor/predicate search is O(U) in the worst case. This is because we need to potentially scan the entire bitvector to find the prev/next item.

4.1.1 Splitting the bitvector into chunks

To improve the successor/predecessor search, we will build a tree on top of the structure. To start, the associative array at the bottom level will be partitioned into chunks of size $\sqrt{U}$. Each chunk is known as a **cluster**. Every layer above the clusters is known as a **summary** layer. Each bit in

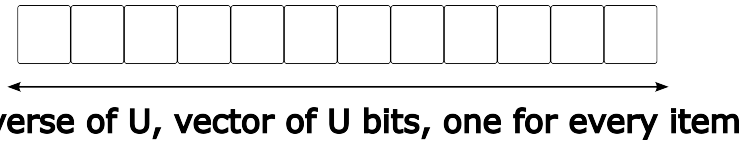

Figure 1: A diagram of the associative array bitmap. Each bit corresponds to one item in the Universe U.

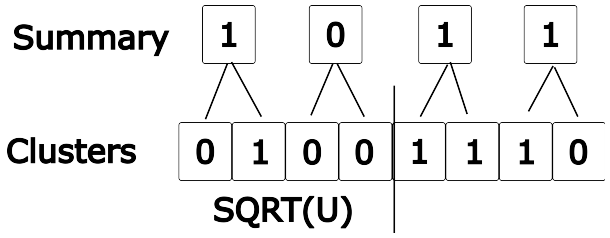

Figure 2: Diagram of the Summary and Cluster layers of the new data structure.

a summary layer corresponds with a cluster in the layer below it. If the cluster corresponding to a summary bit is empty, the bit is 0. Otherwise, it is set to 1.

For a successor search, you first check for a successor inside of your block. If it is now found, float up on Miss and check the summary bits to find the next non-empty block. If the summary blocks are empty, recurse to the next level up. When you have found a bit set to 1 you float down until you are back at the cluster level and have identified the next item.

The new recurrence relation is $T(U) = 3T(\sqrt{U}) + O(1)$.

New Notation: Consider an item as a mix of its cluster id and its index into the cluster. For any item $x \in U$, we can express x as

$$x = i * \sqrt{u} + j$$

, where i is the cluster ID and j is the index inside of the cluster. As i is the upper bits of the item and j are the lower bits, we also refer to I as the quotient and j as the remainder. Since the index size of a cluster is bounded by $\sqrt{U}$, $0 \leq j \leq \sqrt{U}$

Calculating i and j are expensive, as i requires division and j requires modulus. These operations are expensive but are very cheap when the operands are powers of two.

Division is a left shift: $9/2 = 0b1001 >> 1 == 0b0100 == 4$.

Modulus in powers of two is a bitmask: 9 % 2. $9 == 0b1001$, 0b1001 & 0b1 == 0b1 == 1.

Within a Universe where $\sqrt{U} == 2$, $9 = 2 * 4 + 1$.

Note: These structures only make sense in quantizable, manageable universes. Floats, strings, and big universes like 2^64 aren't feasible.

For the rest of the lecture, the quotient bits i are referred to as the *high* bits, and the remainder bits j are the *lower* bits.

At the moment, space usage is still $O(U)$.

4.2 Operations on this new data structure

Call the data structure V, with an input item x.

Algorithm 1 Insert(V,x)

```
1: int i = high(x)
2: int j = low(x)
3: V.clusters[i].bits |= (1 ≪ j)                ▷ set the bit at this level
4: Insert(V.summary, high)                      ▷ recurse and set flags in summary levels
```

Algorithm 2 Successor(V,x)

```
1: int i = high(x)
2: int j = successor(V.clusters[i], low(x))     ▷ check in my cluster
3: if j! = ∞ then
4:     return j                                 ▷ Valid item in my cluster, return that
5: end if
6: i = successor(V.summary, i)                  ▷ find next valid cluster
7: j = successor(v.clusters[i], −∞)             ▷ and search in that cluster
8: return j
```

5 Improving the successor call

At the moment, Successor(n) = $O(\sqrt{U})$. This isn't the $O(\log \log n)$ bound promised, so we need to do better! T(U) = 3T($\sqrt{U}$) is the current bound, and the constant scaling is the problem: $T(U) = T(\sqrt{U}) = O(\log \log U)$, so if we can remove the constant scaling then we are done!

This scaling factor of 3 stems from 3 recursive calls - One in my block, one in my summary, and one in the next cluster.

5.1 Improving the bound: Storing extra metadata

if j = ∞, then the successor is not found in the cluster. This takes up one of the recursive calls to successor, and is overkill. What if we could determine if a larger item existed in $O(1)$?

Idea: Store the highest/lowest bit set in each cluster - the max avoids successor search in the local block. The min avoids a successor search in the next valid block!

Comparing to the maximum bit set has two cases. If we are the maximum bit, we saved a successor search but need to find the next valid cluster. If we are not the maximum bit, then we know a larger bit exists in our cluster, so one call to successor() is guaranteed to find us a valid solution.

Once we have a valid cluster, the minimum bit saves us another call to successor(), as we no longer need to probe into that cluster.

The minimum always cuts off one call to successor, and the max always cuts off exactly 1 of the two remaining successor calls. This means our recurrence is $T(U) = T(\sqrt{U}) + O(1)$, so $T(U) =$

$O(\log \log U)$!

We've hit our performance bound, but space usage is bad: $O(U)$ space. In practice, we use hash tables and avoid storing empty blocks to achieve $O(n)$ space.

5.2 After lecture comments

Updating max and min is constant work per cluster.

These updates to the max and min can be delayed, and using amortized analysis the original paper shows that inserts and deletes can be done in $O(\log \log U)$ time as well.

y-fast tries and x-fast tries use hash tables and tries to convert the space cost to $O(n \log \log U)$, or even $O(n)$ space.

The original paper for the vEB tree is paper 1 on the course website. Papers 2 and 3 supplement it with more advanced implementations of the data structure.