2-INF-237 Vybrané partie z dátových štruktúr

2-INF-237 Selected Topics in Data Structures

- Instructor: Broňa Brejová
- E-mail: brejova@fmph.uniba.sk
- Office: M163

- Course webpage: http://compbio.fmph.uniba.sk/vyuka/vpds/

Model of computation: word RAM

- Consider universe $\mathcal{U} = \{0, \ldots, 2^w - 1\}$
- Word: w-bit unsigned integer
- Input, output, queries, etc. are given in words
- Memory is an array of $\mathfrak{m}$ cells of size w
- Words can serve as pointers (indices)
- We need $w \geq \lg(\mathfrak{m})$, otherwise we cannot index whole memory
- If $\mathfrak{n}$ is the problem size, this implies $w \geq \lg(\mathfrak{n})$
- Program may use "C-style" operations in $O(1)$ on words, such as $+, -, *, /, \%, \&, |, \gg, \ll, <, >$

Predecessor problem

- Maintain a set words over universe U
- Operations insert, delete, predecessor, successor
- Predecessor of $x \in U$: $\max\{y \in S \mid y < x\}$

n: the number of elements in the set

w: size of word

$u = 2^w$: size of universe

Binary search trees: $O(\log n)$ time, $O(n)$ words

Rank/select (static only): $O(1)$ time, $O(2^w/w)$ words

Today vEB: $O(\log w)$ time, $\Omega(2^w)$ words

Improvements: $O(\log w)$ time, $O(n)$ words

Note: if $u = n^{O(1)}$, we have $O(\log w) = O(\log \log n)$

van Emde Boas tree (vEB) (1977)

Goal: running time governed by recurrence

$$T(w) = T(w/2) + O(1)$$

$$T(u) = T(\sqrt{u}) + O(1)$$

Result: $T(w) = O(\log w)$, $T(u) = O(\log \log u)$

Approach: Split U into blocks of size $\sqrt{u}$ or split words into halves

Hierarchical coordinates: $x = \langle b, i \rangle$

where b is block ID, i ID within block

$$x = b\sqrt{u} + i$$

$$b = x/\sqrt{u}$$

$$i = x\%\sqrt{u}$$

van Emde Boas data structure

Structure V for universe size u contains:

- Array V.blocks of size $\sqrt{u}$
  – each element pointer to vEB for universe of size $\sqrt{u}$
- Another vEB V.summary for universe of size $\sqrt{u}$
  – contains as elements IDs of non-empty blocks
- Integer V.max contains maximum element in V
- Integer V.min contains minimum element in V
  – this minimum not stored elsewhere (blocks/summary)
  – this is important for achieving running time

Example of a van Emde Boas tree

$w = 4, \{1 \mid 4, 5, 6, 7 \mid 9, 10 \mid 12, 13, 14\}$

Root structure

min 1, max 14

summary $\{1, 2, 3\}$

blocks $[\emptyset, \{0, 1, 2, 3\}, \{1, 2\}, \{0, 1, 2\}]$

Subtrees for $w = 2$

For set $\{1 \mid 2, 3\}$

min 1, max 3, summary $\{1\}$ blocks $[\emptyset, \{0, 1\}]$

For set $\emptyset$

min None, max None, summary $\emptyset$ blocks $[\emptyset, \emptyset]$

For set $\{0, 1 \mid 2, 3\}$

min 0, max 3, summary $\{0, 1\}$ blocks $[\{1\}, \{0, 1\}]$

...

Predecessor query

```
1  Predecessor(V, x = <b,i>) {
2      if (V.blocks[b].min != None && i > V.blocks[b].min) {
3          return <b, Predecessor(V.blocks[b], i)>;
4      } else {
5          bb = Predecessor(V.summary, b);
6          if(bb == None) {
7              if(x > V.min) return V.min;
8              else return None;
9          }
10         return <bb, V.blocks[bb].max>;
11     }
12 }
```

Insert operation

```
1  Insert(V, x = <b,i>) {
2      if (V.min == None) {
3          V.min = V.max = x;
4          return;
5      }
6      if (x < V.min) swap(x, V.min);
7      if (x > V.max) V.max = x;
8      if (V.blocks[b].min == None) {
9          Insert(V.summary, b);
10     }
11     Insert(V.blocks[b], i);
12 }
```

vEB trees via hash tables

Improving memory:

– do not store empty vEB structures

– array V.blocks replaced by a hash table with dynamic perfect hashing

The memory is proportional to the number of vEB structures:

– hash table proportional to the number of children

– charge space in hash table to children, the rest $O(1)$ per node

The number of vEB structures $O(n \log w)$:

– charge each vEB to the minimum element

– in summary represent each block by minimum

– each element minimum at most twice on one level

Total memory $O(n \log w)$ words,

can be improved to $O(n)$ by careful packing of shorter numbers

x-fast trie, Willard 1983

Each element of S a binary string of length w

Store all prefixes of all elements of S in a hash table

Overall $O(nw)$ words of space

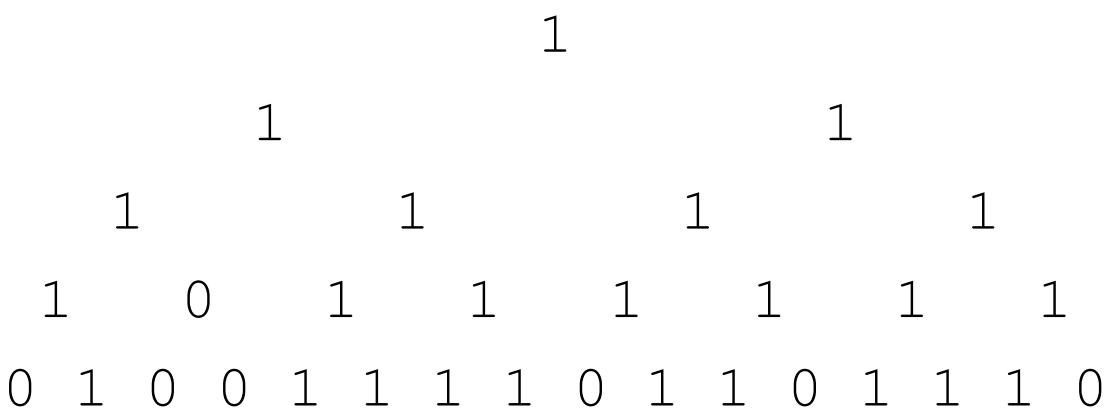
For each prefix store minimum and maximum element with this prefix

For each element of the set store its predecessor and successor

Example of x-fast trie

$w = 4, \{1, 4, 5, 6, 7, 9, 10, 12, 13, 14\}$

View as trie:



Prefixes:

$\epsilon, 0, 1, 00, 01, 10, 11, 000, 010, 011, 100, 101, 110, 111,$
$0001, 0100, 0101, 0110, 0111, 1001, 1010, 1100, 1101, 1110$

Predecessor in x-fast trie

```
1  Predecessor(H, x) {
2    if(x in H) { return H[x].predecessor }
3    y = longest prefix of x in H    // by binary search
4    if(x == y1u) { return H[y].max }
5    else return H[H[y].min].predecessor
6  }
```

Time $O(\log w)$ with perfect hashing

Insert to x-fast trie

- Need to insert/update all $w + 1$ prefixes of x
- For each prefix update min and max
- Update predecessor and successor for x and its neighbors
- $O(w)$ expected time with perfect hashing

y-fast trie, Willard 1983

Disadvantages of x-fast trie:

space $O(nw)$ rather than $O(n)$

insert time $O(w)$ rather than $O(\log w)$

Improve by bucketing:

Maintain buckets, each containing $\Theta(w)$ successive elements of the set

Within bucket store data in a balanced BST

Minimum of each bucket inserted to a x-fast trie or improved vEB

Size of this structure is $O(nw/w) = O(n)$

Combined sizes of trees $O(n)$

y-fast trie operations

Maintain buckets, each containing $\Theta(w)$ successive elements of the set

Within bucket store data in a balanced BST

Minimum of each bucket inserted to a x-fast trie or improved vEB

Predecessor query:

find correct bucket, search in BST, both $O(\log w)$

Insert:

find correct bucket, if enough space, add in $O(\log w)$,

otherwise split bucket into two, insert another element to main structure

Second case costs $O(w)$ but happens infrequently, amortize