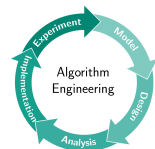




Predecessor Data Structures: Fusion Trees and van Emde Boas Trees

Stefan Walzer, Ragnar Groot Koerkamp, Stefan Hermann | compiled on 8. Juni 2026



1. Problem Statement and Simple Solutions

2. Search Trees of High Degree: “ $\log_w n$ ”

B-Trees

Fusion Trees

3. Splitting the Universe: “ $\log \log u$ ”

Van Emde Boas Tree

x/y/z-Fast-Tries

4. Conclusion

Sorted Sequences / The Predecessor Problem

Setting

- universe $\{0, \dots, u - 1\}$ of size $u = 2^w$
- input set $S \subseteq \{0, \dots, u - 1\}$ of size $n = |S|$
- model: word RAM with word size w

Operations

- $\text{pred}(x) = \max\{y \in S \mid y < x\}$ // or None
- $\text{succ}(x) = \min\{y \in S \mid y > x\}$ // or None
convenience: $\text{pred}(\text{None}) = \max S$, $\text{succ}(\text{None}) = \min S$
- $\text{insert}(x)$, $\text{delete}(x)$ // missing for static data structures

Example ($u = 256, w = 8, n = 4$)

$$S = \{5, 17, 25, 255\}$$

- $\text{pred}(20) = 17$,
- $\text{succ}(17) = 25$,
- $\text{pred}(1) = \text{None}$.

Sorted Sequences / The Predecessor Problem

Setting

- universe $\{0, \dots, u - 1\}$ of size $u = 2^w$
- input set $S \subseteq \{0, \dots, u - 1\}$ of size $n = |S|$
- model: word RAM with word size w

Operations

- $\text{pred}(x) = \max\{y \in S \mid y < x\}$ // or None
- $\text{succ}(x) = \min\{y \in S \mid y > x\}$ // or None
convenience: $\text{pred}(\text{None}) = \max S$, $\text{succ}(\text{None}) = \min S$
- $\text{insert}(x)$, $\text{delete}(x)$ // missing for static data structures

Example ($u = 256, w = 8, n = 4$)

$$S = \{5, 17, 25, 255\}$$

- $\text{pred}(20) = 17$,
- $\text{succ}(17) = 25$,
- $\text{pred}(1) = \text{None}$.

Note: pred and succ imply membership

Sorted Sequences / The Predecessor Problem

Setting

- universe $\{0, \dots, u - 1\}$ of size $u = 2^w$
- input set $S \subseteq \{0, \dots, u - 1\}$ of size $n = |S|$
- model: word RAM with word size w

Operations

- $\text{pred}(x) = \max\{y \in S \mid y < x\}$ // or None
- $\text{succ}(x) = \min\{y \in S \mid y > x\}$ // or None
convenience: $\text{pred}(\text{None}) = \max S$, $\text{succ}(\text{None}) = \min S$
- $\text{insert}(x)$, $\text{delete}(x)$ // missing for static data structures

Example ($u = 256, w = 8, n = 4$)

$$S = \{5, 17, 25, 255\}$$

- $\text{pred}(20) = 17$,
- $\text{succ}(17) = 25$,
- $\text{pred}(1) = \text{None}$.

Note: pred and succ imply membership

$$\text{isMember}(x) = [x \in S] = [\text{succ}(\text{pred}(x)) = x].$$

Solutions with $\mathcal{O}(\log(n))$ time operations

Dynamic:

Problem Statement and Simple Solutions
●○○○○

Search Trees of High Degree: " $\log_w n$ "
○○○○○○○○○

Splitting the Universe: " $\log \log u$ "
○○○○○○○○○

Conclusion
○○

Solutions with $\mathcal{O}(\log(n))$ time operations

Dynamic: Self-Balancing Binary Search Trees

- AVL-Trees
- Red-Black Trees
- 2-4-trees
- ...

BST in Practice

- C++: `std::set`:
`std::prev(M.lower_bound(x)); // if \neq None`
- java: `java.util.TreeSet<K>`
`M.lowerKey(x)`
- rust: `std::collections::BTreeSet<K>`
`M.range(..x).last()`

Solutions with $\mathcal{O}(\log(n))$ time operations

Static:

Dynamic: Self-Balancing Binary Search Trees

- AVL-Trees
- Red-Black Trees
- 2-4-trees
- ...

BST in Practice

- C++: `std::set`:
`std::prev(M.lower_bound(x)); // if \neq None`
- java: `java.util.TreeSet<K>`
`M.lowerKey(x)`
- rust: `std::collections::BTreeSet<K>`
`M.range(..x).last()`

Solutions with $\mathcal{O}(\log(n))$ time operations

Dynamic: Self-Balancing Binary Search Trees

- AVL-Trees
- Red-Black Trees
- 2-4-trees
- ...

BST in Practice

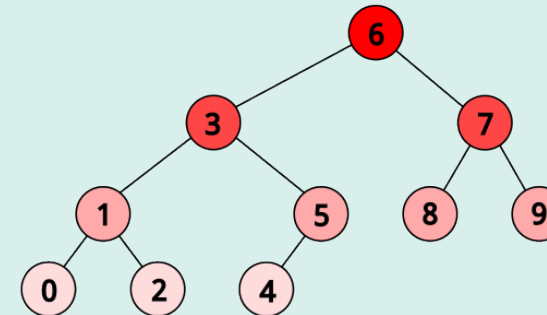
- C++: `std::set`:
`std::prev(M.lower_bound(x)); // if \neq None`
- java: `java.util.TreeSet<K>`
`M.lowerKey(x)`
- rust: `std::collections::BTreeSet<K>`
`M.range(..x).last()`

Static: Array + Binary Search

Binary Search in Practice



- Eytzinger layout for cache efficiency // known from heaps



picture taken from <https://en.algorithmica.org/hpc/data-structures/binary-search/>

- replace branches: $i' \leftarrow 2i + (A[i] < x)$
- use prefetching // next address very predicatable

Exercise: $\mathcal{O}(1)$ time, $\mathcal{O}(u)$ bits, static

Theorem: Succinct Select (see Ragnar's lecture)

There exists a succinct data structure answering rank/select queries on bitvectors in time $\mathcal{O}(1)$.

Corollary:

Exercise: $\mathcal{O}(1)$ time, $\mathcal{O}(u)$ bits, static

Theorem: Succinct Select (see Ragnar's lecture)

There exists a succinct data structure answering rank/select queries on bitvectors in time $\mathcal{O}(1)$.

Corollary: Predecessor data structure using bitvectors

A bitvector v with rank/select data structure yields a *static* predecessor data structure with $\mathcal{O}(1)$ time operations.

- space = $u + o(u)$ bits
- $v[i] = [i \in S]$
- $\text{pred}(i) =$

Exercise: $\mathcal{O}(1)$ time, $\mathcal{O}(u)$ bits, static

Theorem: Succinct Select (see Ragnar's lecture)

There exists a succinct data structure answering rank/select queries on bitvectors in time $\mathcal{O}(1)$.

Corollary: Predecessor data structure using bitvectors

A bitvector v with rank/select data structure yields a *static* predecessor data structure with $\mathcal{O}(1)$ time operations.

- space = $u + o(u)$ bits
- $v[i] = [i \in S]$
- $\text{pred}(i) = \text{select}(\text{rank}(i) - 1)$ // assuming $\neq \text{None}$
- $\text{succ}(i) =$

Exercise: $\mathcal{O}(1)$ time, $\mathcal{O}(u)$ bits, static

Theorem: Succinct Select (see Ragnar's lecture)

There exists a succinct data structure answering rank/select queries on bitvectors in time $\mathcal{O}(1)$.

Corollary: Predecessor data structure using bitvectors

A bitvector v with rank/select data structure yields a *static* predecessor data structure with $\mathcal{O}(1)$ time operations.

- space = $u + o(u)$ bits
- $v[i] = [i \in S]$
- $\text{pred}(i) = \text{select}(\text{rank}(i) - 1)$ // assuming $\neq \text{None}$
- $\text{succ}(i) = \text{select}(\text{rank}(i + 1))$ // assuming $\neq \text{None}$

Compressed Storage using Elias Fano

Theorem: Elias-Fano as Predecessor Structure

Let $b = \lfloor \log(u/n) \rfloor$. Elias-Fano encoding of S yields a static predecessor structure with

- at most $n(b + 3) + o(n)$ bits of space
- $\mathcal{O}(1 + b)$ time operations

Sanity Check

setting	space in bits	time
$u = n^2$		
$u = \Theta(n)$		
$u = n \log n$		

* bitvector for high bits, binary search for low bits

Compressed Storage using Elias Fano

Theorem: Elias-Fano as Predecessor Structure

Let $b = \lfloor \log(u/n) \rfloor$. Elias-Fano encoding of S yields a static predecessor structure with

- at most $n(b + 3) + o(n)$ bits of space
- $\mathcal{O}(1 + b)$ time operations

Sanity Check

setting	space in bits	time	resembles
$u = n^2$	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	binary search
$u = \Theta(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	bitvector
$u = n \log n$	$\mathcal{O}(n \log \log n)$	$\Theta(\log \log n)$	both*

* bitvector for high bits, binary search for low bits

Compressed Storage using Elias Fano

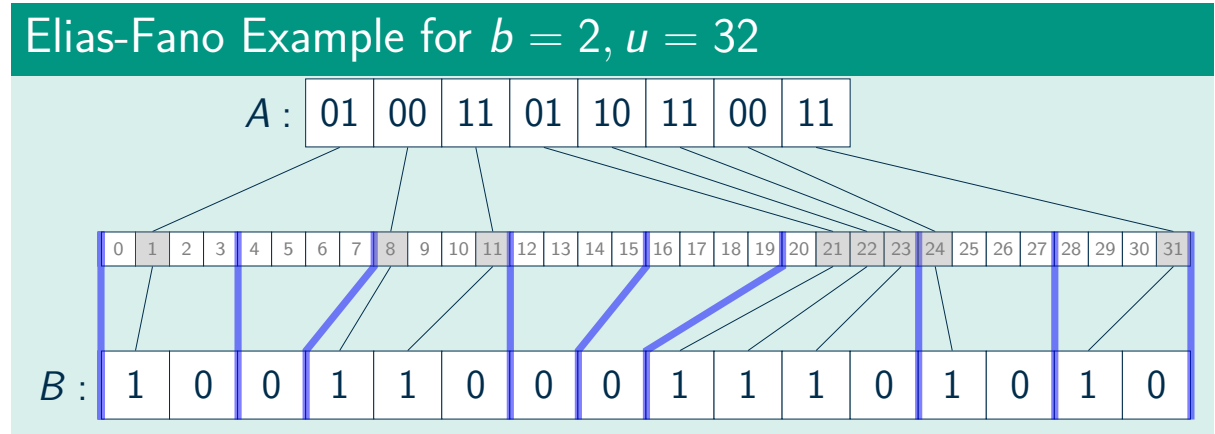
Theorem: Elias-Fano as Predecessor Structure

Let $b = \lfloor \log(u/n) \rfloor$. Elias-Fano encoding of S yields a static predecessor structure with

- at most $n(b + 3) + o(n)$ bits of space
- $\mathcal{O}(1 + b)$ time operations

Space Requirement (mostly reminder)

- for all elements: store low bits explicitly
 \hookrightarrow array A with nb bits
- for high $\in \{0, \dots, \frac{u}{2^b} - 1\}$: unary encoding of counts of element with that high part
 \hookrightarrow bitvector B of $\#0 + \#1 = \frac{u}{2^b} + n = \frac{u}{2^{\lfloor \log u/n \rfloor}} + n \leq 3n$ bits
- select support on B costing $o(n)$ bits



Compressed Storage using Elias Fano

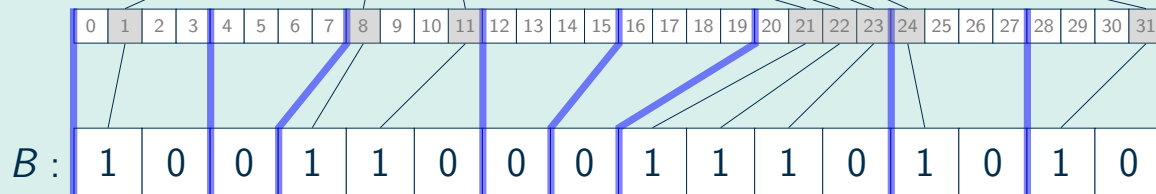
Theorem: Elias-Fano as Predecessor Structure

Let $b = \lfloor \log(u/n) \rfloor$. Elias-Fano encoding of S yields a static predecessor structure with

- at most $n(b + 3) + o(n)$ bits of space
- $\mathcal{O}(1 + b)$ time operations

Elias-Fano Example for $b = 2, u = 32$

A : 01 00 11 01 10 11 00 11



Predecessor Search Implementation

Algorithm pred(x):

```
// split x into high and low bits
⟨high,low⟩ ← x
// find range of elements with matching high bits
(i',j') ← (B.select0(high - 1), B.select0(high)) // select(-1) := 0
(i,j) ← (B.rank1(i'), B.rank1(j')) // relevant range in A is [i..j]
if i = j or low ≤ A[i] then
    // predecessor not in red box ⇒ answer is the (i - 1)th number
    if i = 0 then return None
    return EliasFanoAccess(i - 1) // = ⟨select1(i - 1) - (i - 1), A[i - 1]⟩
else
    low' ← pred of low in A[i..j] using binary search // ≤ 2b elements
    return ⟨high, low'⟩
```

↪ $\mathcal{O}(1 + b)$ time // similar for succ and isMember

Remark: Can simplify using $\text{rank}_1(\text{select}_0(i)) = \text{select}_0(i) - i$.

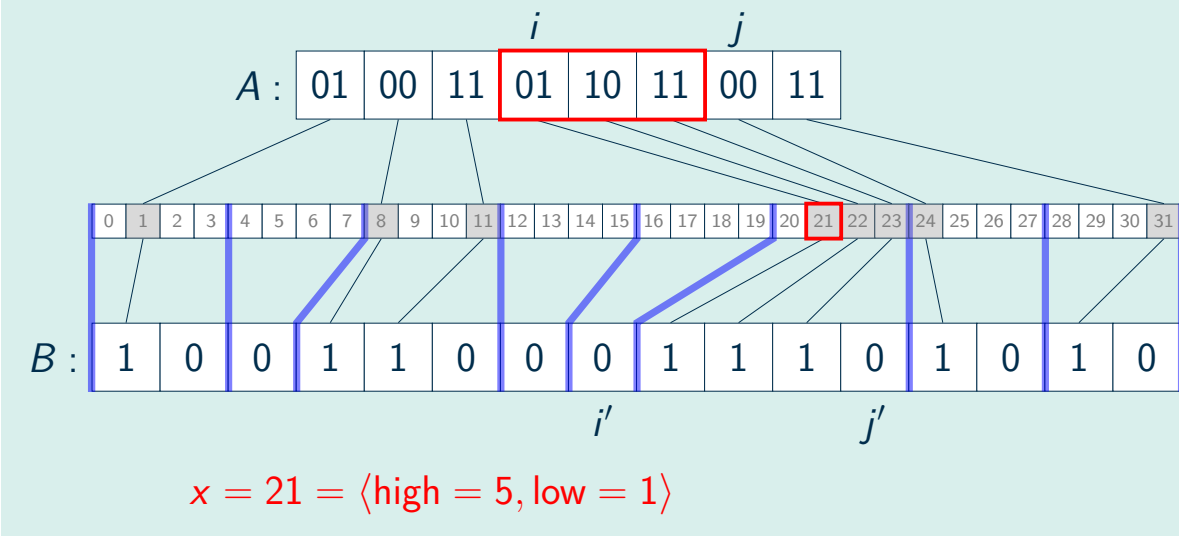
Compressed Storage using Elias Fano

Theorem: Elias-Fano as Predecessor Structure

Let $b = \lfloor \log(u/n) \rfloor$. Elias-Fano encoding of S yields a static predecessor structure with

- at most $n(b + 3) + o(n)$ bits of space
- $\mathcal{O}(1 + b)$ time operations

Elias-Fano Example for $b = 2, u = 32$



Predecessor Search Implementation

Algorithm $\text{pred}(x)$:

```

// split x into high and low bits
⟨high,low⟩ ← x
// find range of elements with matching high bits
(i',j') ← (B.select0(high - 1), B.select0(high)) // select(-1) := 0
(i,j) ← (B.rank1(i'), B.rank1(j')) // relevant range in A is [i..j]
if i = j or low ≤ A[i] then
    // predecessor not in red box ⇒ answer is the (i - 1)th number
    if i = 0 then return None
    return EliasFanoAccess(i - 1) // = ⟨select1(i - 1) - (i - 1), A[i - 1]⟩
else
    low' ← pred of low in A[i..j] using binary search // ≤ 2b elements
    return ⟨high, low'⟩

```

↪ $\mathcal{O}(1 + b)$ time // similar for succ and isMember

Remark: Can simplify using $\text{rank}_1(\text{select}_0(i)) = \text{select}_0(i) - i$.

From now on: Assume large $u \rightsquigarrow$ bitvector useless

Problem Statement and Simple Solutions
○○○○●

Search Trees of High Degree: " $\log_w n$ "
○○○○○○○○○

Splitting the Universe: " $\log \log u$ "
○○○○○○○○○

Conclusion
○○

1. Problem Statement and Simple Solutions

2. Search Trees of High Degree: “ $\log_w n$ ”

B-Trees

Fusion Trees

3. Splitting the Universe: “ $\log \log u$ ”

Van Emde Boas Tree

x/y/z-Fast-Tries

4. Conclusion

Search Trees of High Degree

Observation:

Search time in BST dominated by IOs / cache misses.

Response:

Multi-way split based on a single RAM access.

B-Trees

- $\mathcal{O}(B)$ -way split loading one **block** of size B .
- optimal in *external memory model*
- highly practical

Fusion Trees

- $w^{\Omega(1)}$ -way split in $\mathcal{O}(1)$ time
- model: word RAM with word size w
- space *and* time optimal in theory
- complicated and mostly ignored in practice

1. Problem Statement and Simple Solutions

2. Search Trees of High Degree: “ $\log_w n$ ”

B-Trees

Fusion Trees

3. Splitting the Universe: “ $\log \log u$ ”

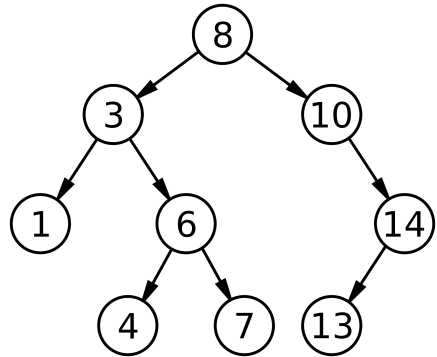
Van Emde Boas Tree

x/y/z-Fast-Tries

4. Conclusion

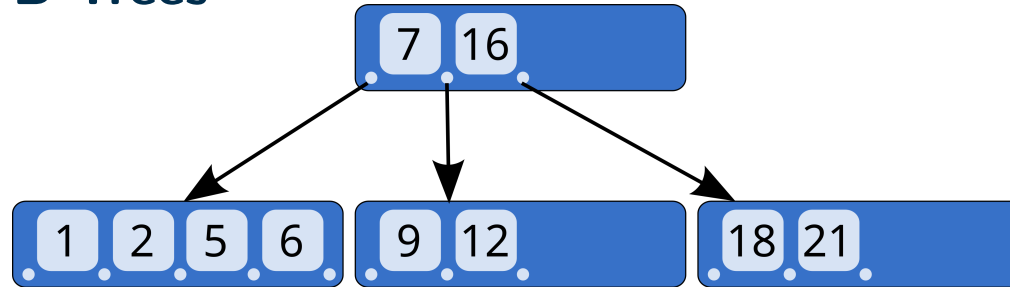
B-Trees

Binary Search Tree



- each node stores one splitter
- depth $\mathcal{O}(\log_2 n)$
↪ search in $\mathcal{O}(\log_2 n)$ IOs

B-Trees



<https://en.wikipedia.org/wiki/B-tree#/media/File:B-tree.svg> / Wikipedia User: Cyhawk

- each node stores $\Theta(B)$ splitters
- depth $\mathcal{O}(\log_B n)$
↪ search in $\mathcal{O}(\log_B n)$ IOs

Implementation Notes

- Enforcing balance: all leaves on same level
- B^+ trees: only leafs store values, leafs are linked
- use SIMD to compare against all splitters of node
- $B \geq 100$ not uncommon in data base contexts

1. Problem Statement and Simple Solutions

2. Search Trees of High Degree: “ $\log_w n$ ”

B-Trees

Fusion Trees

3. Splitting the Universe: “ $\log \log u$ ”

Van Emde Boas Tree

x/y/z-Fast-Tries

4. Conclusion

Fusion Trees

Theorem

There exists a static¹ predecessor data structure with $\mathcal{O}(\log_w n)$ -time operations in the word RAM model.

Proof plan:

- implement a tree node for w -way splitting
 - $\mathcal{O}(1)$ -time routing to correct subtree
 - height: $\log_w(n)$

¹Dynamic versions are known, e.g. doi:10.1109/FOCS.2014.26.

Fusion Trees

Theorem

There exists a static¹ predecessor data structure with $\mathcal{O}(\log_w n)$ -time operations in the word RAM model.

Proof plan:

- implement a tree node for w -way $\mathcal{O}(\sqrt{w})$ -way splitting
 - $\mathcal{O}(1)$ -time routing to correct subtree
 - height: ~~$\log_w(n)$~~
 - height: $\log_{\sqrt{w}} n = \frac{\log n}{\log \sqrt{w}} = 2 \frac{\log n}{\log w} = \mathcal{O}(\log_w n)$ ✓ // still strong enough

¹Dynamic versions are known, e.g. doi:10.1109/FOCS.2014.26.

Fusion Trees

Theorem

There exists a static¹ predecessor data structure with $\mathcal{O}(\log_w n)$ -time operations in the word RAM model.

Proof plan:

- implement a tree node for w -way $\mathcal{O}(\sqrt{w})$ -way splitting
 - $\mathcal{O}(1)$ -time routing to correct subtree
 - height: ~~$\log_w(n)$~~
 - height: $\log_{\sqrt{w}} n = \frac{\log n}{\log \sqrt{w}} = 2 \frac{\log n}{\log w} = \mathcal{O}(\log_w n)$ ✓ // still strong enough
- shamelessly use modern instructions // doubtful in word RAM model
 - we use `clz`, `pext` // BMI2 instruction set, 2013
 - `clz(0000010101101111) = 5`
 - `pext(src = 0101010111110000, mask = 1001011011110011) = 0000000110111100`
 - similar behaviour can be simulated with c-like instructions // https://en.wikipedia.org/wiki/Fusion_tree

¹Dynamic versions are known, e.g. doi:10.1109/FOCS.2014.26.

Implementation k -way splits for $k = \mathcal{O}(\sqrt{w})$

Todo

Given splitters $x_1, \dots, x_k \in \{0, 1\}^w$, construct a data structure that can answer the following in $\mathcal{O}(1)$ time:
Given a key $y \in \{0, 1\}^w$, compute its *rank* $0 \leq r \leq k$, i.e. how many splitters are less than y .

Challenge

x_1, \dots, x_k involve $kw = \Omega(w^{3/2})$ bits
 \hookrightarrow cannot be read in $\mathcal{O}(1)$ word operations

Implementation k -way splits for $k = \mathcal{O}(\sqrt{w})$

Todo

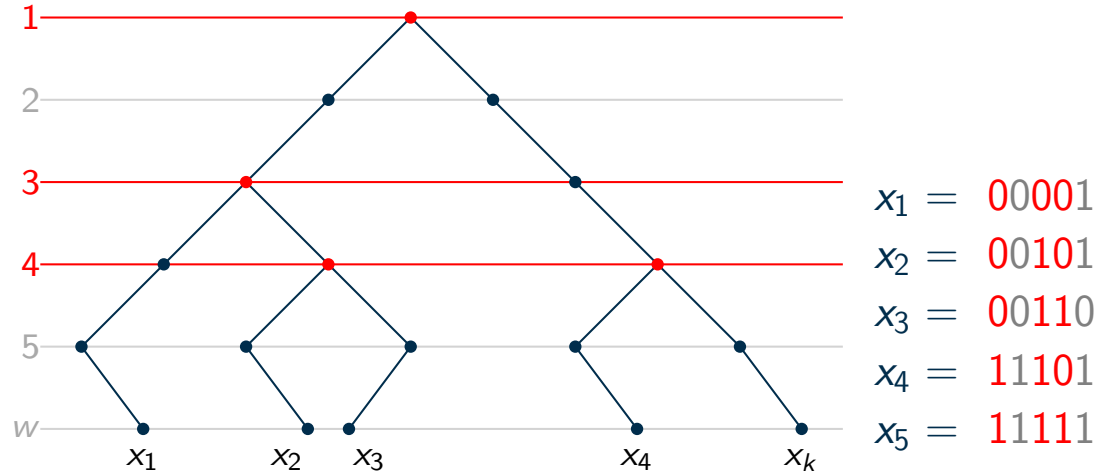
Given splitters $x_1, \dots, x_k \in \{0, 1\}^w$, construct a data structure that can answer the following in $\mathcal{O}(1)$ time:
 Given a key $y \in \{0, 1\}^w$, compute its *rank* $0 \leq r \leq k$, i.e. how many splitters are less than y .

Challenge

x_1, \dots, x_k involve $kw = \Omega(w^{3/2})$ bits
 \hookrightarrow cannot be read in $\mathcal{O}(1)$ word operations

Idea

Preprocess x_1, \dots, x_k to extract $\mathcal{O}(w)$ relevant bits.



- compute word indicating **branching bits**
 \hookrightarrow branchBits = 10110 // note: $\leq k - 1$ bits set
- define sketch $\tilde{x} := \text{pext}(x, \text{branchBits})$
 $\hookrightarrow x_2 = 00101 \rightsquigarrow \tilde{x}_2 = 010$
 // pext = parallel bit extract
- concatenate sketches $\tilde{x}_1, \dots, \tilde{x}_k$ into \tilde{X} , separated by ones // ones relevant on next slide
 $\hookrightarrow \tilde{X} = 1 \underbrace{000}_{\tilde{x}_1} 1 \underbrace{010}_{\tilde{x}_2} 1 \underbrace{011}_{\tilde{x}_3} 1 \underbrace{110}_{\tilde{x}_4} 1 \underbrace{111}_{\tilde{x}_5} \in \{0, 1\}^{\mathcal{O}(w)}$

Parallel comparison of sketches

Algorithm rank($\tilde{y}, \tilde{X} = 1\tilde{x}_1 \dots 1\tilde{x}_k$):

```
// goal: compute the rank  $0 \leq r \leq k$  of  $\tilde{y}$  with respect to  $\tilde{x}_1, \dots, \tilde{x}_k$ 
 $\tilde{Y} \leftarrow (0^k 1)^k \cdot \tilde{y}$  //  $0\tilde{y}0\tilde{y} \dots 0\tilde{y}$ 
 $d \leftarrow (\tilde{X} - \tilde{Y}) \& (10^k 10^k \dots 10^k)$  // “carry” bits indicating  $\tilde{y} > \tilde{x}_i$ 
return clz( $d$ )/( $k + 1$ ) // clz = count leading zeroes
```

Example ($k = 5$)

variable	what is encoded	binary representation
$\tilde{y} = 7$		= 00111
$\tilde{X} = (2, 5, 7, 8, 9)$		= 1 00010 1 00101 1 00111 1 01000 1 01001
$\tilde{Y} = (7, 7, 7, 7, 7)$		= 0 00111 0 00111 0 00111 0 00111 0 00111
$\tilde{X} - \tilde{Y} = (-5, -2, +0, +1, +2)$		= 0 11011 0 11110 1 00000 1 00001 1 00010
$d = (-, -, +, +, +)$		= 0 00000 0 00000 1 00000 1 00000 1 00000
$\text{clz}(d)/(k + 1) = 2$		

Implementation k -way splits for $k = \mathcal{O}(\sqrt{w})$

Algorithm routeQuery($y \in \{0, 1\}^w$):

```
// assume  $\tilde{X}$  and branchBits precomputed
 $\tilde{y} \leftarrow \text{sketch}(y)$ 
 $r \leftarrow \text{rank}(\tilde{y}, \tilde{X})$ 
if  $r < k$  and  $x_{r+1} = y$  then return  $r$ 
// use longest common prefix  $\text{lcp}(x, y) := \text{clz}(x \oplus y)$ 
 $\ell \leftarrow \max(\text{lcp}(y, x_r), \text{lcp}(y, x_{r+1}))$ 
if  $y[\ell + 1] = 1$  then //  $y$  “falls off” to the right
     $y^* \leftarrow y[1 \dots \ell] \circ 1^{w-\ell}$ 
    // “+1” converts “ $\leq$ ” into “ $<$ ”:
    return  $\text{rank}(\text{sketch}(y^*) + 1, \tilde{X})$ 
else
     $y^* \leftarrow y[1 \dots \ell] \circ 0^{w-\ell}$  // analogous
    return  $\text{rank}(\text{sketch}(y^*), \tilde{X})$  // no “+1”
```

Case 1: $y = x_{r+1}$ for some $0 \leq r < k$

Since $\tilde{x}_{r+1} = \tilde{y}$ and $\text{rank}(\tilde{x}_{r+1}, \tilde{X}) = r$ this case is correctly recognised.

Case 2: y is not one of the splitters

- Let p be the longest prefix of y that is a prefix of a splitter, let $\ell = |p|$ and $b = y[\ell + 1]$.
- Let $x_s < \dots < x_t$ be the range of splitters starting with p .
- Case 2.1: $b = 1$ ($b = 0$ is similar):
 - Note: y belongs after x_t . If b is missing from \tilde{y} , then $r = \text{rank}(\tilde{y}, \tilde{X})$ satisfies only $s - 1 \leq r \leq t$.
 - We may compute $\ell = \max(\text{lcp}(y, x_r), \text{lcp}(y, x_{r+1}))$.
 - After replacing all bits in y after p with 1-bits, the modified y^* satisfies $\tilde{x}_t < \text{sketch}(y^*) + 1 \leq \tilde{x}_{t+1}$, so we correctly return t .

$x_1 = 0010110$
 $x_2 = 0011010$
 $x_3 = \mathbf{01}00111 \leftarrow s = 3$
 $x_4 = \mathbf{01}01000$
 $x_5 = \mathbf{01}01001 \leftarrow t = 5$
 $x_6 = 1101011$

$y = \overbrace{\mathbf{01}}^p \overbrace{10100}^b$
 $\hookrightarrow \ell = 2$
 $\hookrightarrow y^* = 0111111$

1. Problem Statement and Simple Solutions

2. Search Trees of High Degree: “ $\log_w n$ ”

B-Trees

Fusion Trees

3. Splitting the Universe: “ $\log \log u$ ”

Van Emde Boas Tree

$x/y/z$ -Fast-Tries

4. Conclusion

Van-Emde-Boas Tree

Theorem

There exists a dynamic predecessor data structure for $S \subseteq \{0, \dots, u - 1\}$ with $\mathcal{O}(\log \log u)$ -time operations. // assumes word size $w \geq \log u$

Recursive Case: van Emde Boas tree for $u > w$

- $\text{MIN}, \text{MAX} \in \{\text{None}\} \cup \{0, \dots, u - 1\}$ // MIN is *not* stored recursively
- $\text{blocks}[0..\sqrt{u} - 1]$: vEB trees of size \sqrt{u}
- summary: vEB tree of size \sqrt{u} indicating non-empty blocks

Example for $u = 16$: $\text{vEB}(0110\ 1110\ 0100\ 0000) = \text{vEB}(S = \{1, 2, 4, 5, 6, 9\})$

- $\text{MIN} = 1, \text{MAX} = 9$
- $\text{blocks} = [\text{vEB}(0010), \text{vEB}(1110), \text{vEB}(0100), \text{vEB}(0000)]$
- $\text{summary} = \text{vEB}(1110)$

Idea

Recursively partition u into \sqrt{u} blocks of size \sqrt{u} .
 $\hookrightarrow \log \log u$ levels of recursion

Base Case: van Emde Boas “tree” for $u \leq w$

- represent S using a single word
- supports insert, delete, pred, succ in $\mathcal{O}(1)$

Van-Emde-Boas Tree

Theorem

There exists a dynamic predecessor data structure for $S \subseteq \{0, \dots, u - 1\}$ with $\mathcal{O}(\log \log u)$ -time operations. // assumes word size $w \geq \log u$

Recursive Case: van Emde Boas tree for $u > w$

- $\text{MIN}, \text{MAX} \in \{\text{None}\} \cup \{0, \dots, u - 1\}$ // MIN is *not* stored recursively
- $\text{blocks}[0..\sqrt{u} - 1]$: vEB trees of size \sqrt{u}
- summary: vEB tree of size \sqrt{u} indicating non-empty blocks

Example for $u = 16$: $\text{vEB}(0\mathbf{1}10\ 1110\ 0100\ 0000) = \text{vEB}(S = \{1, 2, 4, 5, 6, 9\})$

- $\text{MIN} = 1, \text{MAX} = 9$
- $\text{blocks} = [\text{vEB}(0\mathbf{0}10), \text{vEB}(1110), \text{vEB}(0100), \text{vEB}(0000)]$
- $\text{summary} = \text{vEB}(1110)$

Idea

Recursively partition u into \sqrt{u} blocks of size \sqrt{u} .
 $\hookrightarrow \log \log u$ levels of recursion

Base Case: van Emde Boas “tree” for $u \leq w$

- represent S using a single word
- supports insert, delete, pred, succ in $\mathcal{O}(1)$

Operations on vEB Trees (assume for convenience that w is power-of-2)

vEB data structure

- MIN, MAX // min not stored recursively
- blocks[0.. $\sqrt{u} - 1$]: vEB trees of size \sqrt{u}
- summary: vEB tree of size \sqrt{u}

Algorithm succ(T, x):



Operations on vEB Trees (assume for convenience that w is power-of-2)

vEB data structure

- MIN, MAX // min not stored recursively
- $\text{blocks}[0..\sqrt{u}-1]$: vEB trees of size \sqrt{u}
- summary: vEB tree of size \sqrt{u}

Algorithm $\text{succ}(T, x)$:

```
if base case then return [...] // todo
if T.MIN = None or  $x \geq T.MAX$  then return None
if  $x < T.MIN$  then return T.MIN
 $\langle h, \ell \rangle \leftarrow x$  // split  $x$  into  $w/2$  high bits and  $w/2$  low bits
if T.blocks[h].MIN = None or  $\ell \geq T.blocks[h].MAX$  then
    |  $h' \leftarrow \text{succ}(T.summary, h)$ 
    | return  $\langle h', T.blocks[h'].MIN \rangle$ 
else
    | return  $\langle h, \text{succ}(T.blocks[h], \ell) \rangle$ 
```

Algorithm $\text{pred}(T, x)$:

```
└ ... // similar
```

Operations on vEB Trees (assume for convenience that w is power-of-2)

vEB data structure

- MIN, MAX // min not stored recursively
- $\text{blocks}[0..\sqrt{u}-1]$: vEB trees of size \sqrt{u}
- summary: vEB tree of size \sqrt{u}

Algorithm $\text{succ}(T, x)$:

```
if base case then return [...] // todo
if T.MIN = None or  $x \geq T.MAX$  then return None
if  $x < T.MIN$  then return T.MIN
 $\langle h, \ell \rangle \leftarrow x$  // split  $x$  into  $w/2$  high bits and  $w/2$  low bits
if T.blocks[h].MIN = None or  $\ell \geq T.blocks[h].MAX$  then
    |  $h' \leftarrow \text{succ}(T.summary, h)$ 
    | return  $\langle h', T.blocks[h'].MIN \rangle$ 
else
    | return  $\langle h, \text{succ}(T.blocks[h], \ell) \rangle$ 
```

Algorithm $\text{pred}(T, x)$:

```
| ... // similar
```

Algorithm $\text{insert}(T, x)$:

Operations on vEB Trees (assume for convenience that w is power-of-2)

vEB data structure

- MIN, MAX // min not stored recursively
- blocks[0.. $\sqrt{u} - 1$]: vEB trees of size \sqrt{u}
- summary: vEB tree of size \sqrt{u}

Algorithm succ(T, x):

```
if base case then return [...] // todo
if T.MIN = None or  $x \geq T.MAX$  then return None
if  $x < T.MIN$  then return T.MIN
 $\langle h, \ell \rangle \leftarrow x$  // split  $x$  into  $w/2$  high bits and  $w/2$  low bits
if T.blocks[h].MIN = None or  $\ell \geq T.blocks[h].MAX$  then
     $h' \leftarrow \text{succ}(T.summary, h)$ 
    return  $\langle h', T.blocks[h'].MIN \rangle$ 
else
    return  $\langle h, \text{succ}(T.blocks[h], \ell) \rangle$ 
```

Algorithm pred(T, x):

```
... // similar
```

Algorithm insert(T, x):

```
if base case then [...] // todo
if T.MIN = None then
    T.MIN  $\leftarrow x$ , T.MAX  $\leftarrow x$ 
    return
if T.MIN =  $x$  then return
if  $x < T.MIN$  then
    // next: insert old min
    swap(T.MIN, x)
T.MAX  $\leftarrow \max(T.MAX, x)$ 
 $\langle h, \ell \rangle \leftarrow x$  // split
if T.blocks[h].MIN = None then
    insert(T.summary, h)
insert(T.blocks[h],  $\ell$ )
```

Operations on vEB Trees (assume for convenience that w is power-of-2)

vEB data structure

- MIN, MAX // min not stored recursively
- blocks[0.. $\sqrt{u} - 1$]: vEB trees of size \sqrt{u}
- summary: vEB tree of size \sqrt{u}

Algorithm succ(T, x):

```
if base case then return [...] // todo
if T.MIN = None or  $x \geq T.MAX$  then return None
if  $x < T.MIN$  then return T.MIN
 $\langle h, \ell \rangle \leftarrow x$  // split  $x$  into  $w/2$  high bits and  $w/2$  low bits
if T.blocks[h].MIN = None or  $\ell \geq T.blocks[h].MAX$  then
     $h' \leftarrow \text{succ}(T.summary, h)$ 
    return  $\langle h', T.blocks[h'].MIN \rangle$ 
else
    return  $\langle h, \text{succ}(T.blocks[h], \ell) \rangle$ 
```

Algorithm pred(T, x):

```
... // similar
```

Algorithm insert(T, x):

```
if base case then [...] // todo
if T.MIN = None then
    T.MIN  $\leftarrow x$ , T.MAX  $\leftarrow x$ 
    return
if T.MIN =  $x$  then return
if  $x < T.MIN$  then
    // next: insert old min
    swap(T.MIN, x)
T.MAX  $\leftarrow \max(T.MAX, x)$ 
 $\langle h, \ell \rangle \leftarrow x$  // split
if T.blocks[h].MIN = None then
    insert(T.summary, h)
insert(T.blocks[h],  $\ell$ )
```

Algorithm delete(T, x):

Operations on vEB Trees (assume for convenience that w is power-of-2)

vEB data structure

- MIN, MAX // min not stored recursively
- blocks[0.. $\sqrt{u} - 1$]: vEB trees of size \sqrt{u}
- summary: vEB tree of size \sqrt{u}

Algorithm succ(T, x):

```

if base case then return [...] // todo
if  $T.MIN = None$  or  $x \geq T.MAX$  then return None
if  $x < T.MIN$  then return  $T.MIN$ 
 $\langle h, \ell \rangle \leftarrow x$  // split  $x$  into  $w/2$  high bits and  $w/2$  low bits
if  $T.blocks[h].MIN = None$  or  $\ell \geq T.blocks[h].MAX$  then
     $h' \leftarrow succ(T.summary, h)$ 
    return  $\langle h', T.blocks[h'].MIN \rangle$ 
else
    return  $\langle h, succ(T.blocks[h], \ell) \rangle$ 

```

Algorithm pred(T, x):

```

    ... // similar

```

Algorithm insert(T, x):

```

if base case then [...] // todo
if  $T.MIN = None$  then
     $T.MIN \leftarrow x, T.MAX \leftarrow x$ 
    return
if  $T.MIN = x$  then return
if  $x < T.MIN$  then
    // next: insert old min
    swap( $T.MIN, x$ )
     $T.MAX \leftarrow max(T.MAX, x)$ 
     $\langle h, \ell \rangle \leftarrow x$  // split
    if  $T.blocks[h].MIN = None$  then
        insert( $T.summary, h$ )
    insert( $T.blocks[h], \ell$ )

```

Algorithm delete(T, x):

```

if base case then [...] // todo
if  $x = T.MIN = T.MAX$  then
     $T.MIN \leftarrow None$ 
     $T.MAX \leftarrow None$ 
    return
if  $x = T.MIN$  then
     $T.MIN \leftarrow \dots$  // update in  $\mathcal{O}(1)$ 
    // delete  $T.MIN$  from substructures
     $x \leftarrow T.MIN$ 
     $\langle h, \ell \rangle \leftarrow x$ 
    delete( $T.blocks[h], \ell$ )
    if  $T.blocks[h].MIN = None$  then
        delete( $T.summary, h$ )
     $T.MAX \leftarrow \dots$  // update in  $\mathcal{O}(1)$ 

```

Running Time Analysis:

Problem Statement and Simple Solutions
○○○○○

Search Trees of High Degree: " $\log_w n$ "
○○○○○○○○○

Splitting the Universe: " $\log \log u$ "
○○●○○○○○

Conclusion
○○

Operations on vEB Trees (assume for convenience that w is power-of-2)

vEB data structure

- MIN, MAX // min not stored recursively
- blocks[0.. $\sqrt{u} - 1$]: vEB trees of size \sqrt{u}
- summary: vEB tree of size \sqrt{u}

Algorithm succ(T, x):

```

if base case then return [...] // todo
if  $T.MIN = None$  or  $x \geq T.MAX$  then return None
if  $x < T.MIN$  then return  $T.MIN$ 
 $\langle h, \ell \rangle \leftarrow x$  // split  $x$  into  $w/2$  high bits and  $w/2$  low bits
if  $T.blocks[h].MIN = None$  or  $\ell \geq T.blocks[h].MAX$  then
     $h' \leftarrow succ(T.summary, h)$ 
    return  $\langle h', T.blocks[h'].MIN \rangle$ 
else
    return  $\langle h, succ(T.blocks[h], \ell) \rangle$ 
    
```

Algorithm pred(T, x):

```

    ... // similar
    
```

Algorithm insert(T, x):

```

if base case then [...] // todo
if  $T.MIN = None$  then
     $T.MIN \leftarrow x, T.MAX \leftarrow x$ 
    return
if  $T.MIN = x$  then return
if  $x < T.MIN$  then
    // next: insert old min
    swap( $T.MIN, x$ )
     $T.MAX \leftarrow \max(T.MAX, x)$ 
     $\langle h, \ell \rangle \leftarrow x$  // split
    if  $T.blocks[h].MIN = None$  then
        insert( $T.summary, h$ )
    insert( $T.blocks[h], \ell$ )
    
```

Algorithm delete(T, x):

```

if base case then [...] // todo
if  $x = T.MIN = T.MAX$  then
     $T.MIN \leftarrow None$ 
     $T.MAX \leftarrow None$ 
    return
if  $x = T.MIN$  then
     $T.MIN \leftarrow \dots$  // update in  $\mathcal{O}(1)$ 
    // delete  $T.MIN$  from substructures
     $x \leftarrow T.MIN$ 
     $\langle h, \ell \rangle \leftarrow x$ 
    delete( $T.blocks[h], \ell$ )
    if  $T.blocks[h].MIN = None$  then
        delete( $T.summary, h$ )
     $T.MAX \leftarrow \dots$  // update in  $\mathcal{O}(1)$ 
    
```

Running Time Analysis: only one

recursive call: $\rightsquigarrow t(u) = \mathcal{O}(1) + t(\sqrt{u}) \rightsquigarrow t(u) = \log \log u.$

Operations on vEB Trees (assume for convenience that w is power-of-2)

vEB data structure

- MIN, MAX // min not stored recursively
- blocks[0.. $\sqrt{u} - 1$]: vEB trees of size \sqrt{u}
- summary: vEB tree of size \sqrt{u}

Algorithm succ(T, x):

```

if base case then return [...] // todo
if  $T.MIN = None$  or  $x \geq T.MAX$  then return None
if  $x < T.MIN$  then return  $T.MIN$ 
 $\langle h, \ell \rangle \leftarrow x$  // split  $x$  into  $w/2$  high bits and  $w/2$  low bits
if  $T.blocks[h].MIN = None$  or  $\ell \geq T.blocks[h].MAX$  then
     $h' \leftarrow succ(T.summary, h)$ 
    return  $\langle h', T.blocks[h'].MIN \rangle$ 
else
    return  $\langle h, succ(T.blocks[h], \ell) \rangle$ 
    
```

Algorithm pred(T, x):

```

    ... // similar
    
```

Algorithm insert(T, x):

```

if base case then [...] // todo
if  $T.MIN = None$  then
     $T.MIN \leftarrow x, T.MAX \leftarrow x$ 
    return
if  $T.MIN = x$  then return
if  $x < T.MIN$  then
    // next: insert old min
    swap( $T.MIN, x$ )
     $T.MAX \leftarrow max(T.MAX, x)$ 
     $\langle h, \ell \rangle \leftarrow x$  // split
    if  $T.blocks[h].MIN = None$  then
        insert( $T.summary, h$ )
        // in this case: next insertion  $\mathcal{O}(1)$ !
    insert( $T.blocks[h], \ell$ )
    
```

Algorithm delete(T, x):

```

if base case then [...] // todo
if  $x = T.MIN = T.MAX$  then
     $T.MIN \leftarrow None$ 
     $T.MAX \leftarrow None$ 
    return
if  $x = T.MIN$  then
     $T.MIN \leftarrow \dots$  // update in  $\mathcal{O}(1)$ 
    // delete  $T.MIN$  from substructures
     $x \leftarrow T.MIN$ 
     $\langle h, \ell \rangle \leftarrow x$ 
    delete( $T.blocks[h], \ell$ )
    if  $T.blocks[h].MIN = None$  then
        // note: previous call was  $\mathcal{O}(1)$ 
        delete( $T.summary, h$ )
     $T.MAX \leftarrow \dots$  // update in  $\mathcal{O}(1)$ 
    
```

Running Time Analysis: only one non-trivial recursive call: $\rightsquigarrow t(u) = \mathcal{O}(1) + t(\sqrt{u}) \rightsquigarrow t(u) = \log \log u$.

Space Usage of Van Emde Boas Trees

Observation: Space $\mathcal{O}(u)$ (even naively)

$$S(u) = \underbrace{\sqrt{u}}_{\text{array}} + \underbrace{(\sqrt{u} + 1) \cdot S(\sqrt{u})}_{\text{substructures}} + \mathcal{O}(1) \text{ which gives}^a S(u) = \mathcal{O}(u).$$

^aThis is not trivial. Intuition: $S(u) = \sqrt{u}S(\sqrt{u})$ clearly gives $S(u) = u$. The factors $\frac{\sqrt{u}+1}{\sqrt{u}}$ make a total difference of at most $\prod_{k \geq 0} \frac{2^k+1}{2^k} = \mathcal{O}(1)$.

Space Usage of Van Emde Boas Trees

Theorem: Space $\mathcal{O}(n \log \log u)$

Van Emde Boas trees admit an implementation with *expected* time $\mathcal{O}(\log \log u)$ per operation and $\mathcal{O}(n \log \log u)$ words of space.

Observation: Space $\mathcal{O}(u)$ (even naively)

$$S(u) = \underbrace{\sqrt{u}}_{\text{array}} + \underbrace{(\sqrt{u} + 1) \cdot S(\sqrt{u})}_{\text{substructures}} + \mathcal{O}(1) \text{ which gives}^a S(u) = \mathcal{O}(u).$$

^aThis is not trivial. Intuition: $S(u) = \sqrt{u}S(\sqrt{u})$ clearly gives $S(u) = u$. The factors $\frac{\sqrt{u}+1}{\sqrt{u}}$ make a total difference of at most $\prod_{k \geq 0} \frac{2^k+1}{2^k} = \mathcal{O}(1)$.

Space Usage of Van Emde Boas Trees

Theorem: Space $\mathcal{O}(n \log \log u)$

Van Emde Boas trees admit an implementation with *expected* time $\mathcal{O}(\log \log u)$ per operation and $\mathcal{O}(n \log \log u)$ words of space.

Proof: “don’t store empty structures”

Idea: Make T .blocks a hash table (e.g. using linear probing)

space = $\mathcal{O}(\# \text{non-empty structures} + \# \text{hash table entries})$

Observation: Space $\mathcal{O}(u)$ (even naively)

$$S(u) = \underbrace{\sqrt{u}}_{\text{array}} + \underbrace{(\sqrt{u} + 1) \cdot S(\sqrt{u})}_{\text{substructures}} + \mathcal{O}(1) \text{ which gives}^a S(u) = \mathcal{O}(u).$$

^aThis is not trivial. Intuition: $S(u) = \sqrt{u}S(\sqrt{u})$ clearly gives $S(u) = u$. The factors $\frac{\sqrt{u}+1}{\sqrt{u}}$ make a total difference of at most $\prod_{k \geq 0} \frac{2^k+1}{2^k} = \mathcal{O}(1)$.

Space Usage of Van Emde Boas Trees

Theorem: Space $\mathcal{O}(n \log \log u)$

Van Emde Boas trees admit an implementation with *expected* time $\mathcal{O}(\log \log u)$ per operation and $\mathcal{O}(n \log \log u)$ words of space.

Proof: “don’t store empty structures”

Idea: Make T .blocks a hash table (e.g. using linear probing)

space = $\mathcal{O}(\# \text{non-empty structures} + \# \text{hash table entries})$

Note: insert takes $\mathcal{O}(\log \log u)$ time

\Rightarrow insertion contributes at most $\mathcal{O}(\log \log u)$ to space

Observation: Space $\mathcal{O}(u)$ (even naively)

$$S(u) = \underbrace{\sqrt{u}}_{\text{array}} + \underbrace{(\sqrt{u} + 1) \cdot S(\sqrt{u})}_{\text{substructures}} + \mathcal{O}(1) \text{ which gives}^a S(u) = \mathcal{O}(u).$$

^aThis is not trivial. Intuition: $S(u) = \sqrt{u}S(\sqrt{u})$ clearly gives $S(u) = u$. The factors $\frac{\sqrt{u}+1}{\sqrt{u}}$ make a total difference of at most $\prod_{k \geq 0} \frac{2^k+1}{2^k} = \mathcal{O}(1)$.

Space Usage of Van Emde Boas Trees

Theorem: Space $\mathcal{O}(n \log \log u)$

Van Emde Boas trees admit an implementation with *expected* time $\mathcal{O}(\log \log u)$ per operation and $\mathcal{O}(n \log \log u)$ words of space.

Proof: “don’t store empty structures”

Idea: Make T .blocks a hash table (e.g. using linear probing)

space = $\mathcal{O}(\# \text{non-empty structures} + \# \text{hash table entries})$

Note: insert takes $\mathcal{O}(\log \log u)$ time

\Rightarrow insertion contributes at most $\mathcal{O}(\log \log u)$ to space

Observation: Space $\mathcal{O}(u)$ (even naively)

$$S(u) = \underbrace{\sqrt{u}}_{\text{array}} + \underbrace{(\sqrt{u} + 1) \cdot S(\sqrt{u})}_{\text{substructures}} + \mathcal{O}(1) \text{ which gives}^a S(u) = \mathcal{O}(u).$$

Theorem: Space $\mathcal{O}(n)$

- refined version of “don’t store empty structures”
 - insert allocates $\mathcal{O}(1)$ words per recursion level
 - *but*: effective word size decreases geometrically
- \Rightarrow only $\mathcal{O}(1)$ words added overall!

⚠ pointers must also shrink geometrically

- see blog post by Mihai Pătrașcu
<https://infowebly.blogspot.com/2010/09/van-emde-boas-and-its-space-complexity.html>

^aThis is not trivial. Intuition: $S(u) = \sqrt{u}S(\sqrt{u})$ clearly gives $S(u) = u$. The factors $\frac{\sqrt{u}+1}{\sqrt{u}}$ make a total difference of at most $\prod_{k \geq 0} \frac{2^k+1}{2^k} = \mathcal{O}(1)$.

1. Problem Statement and Simple Solutions

2. Search Trees of High Degree: “ $\log_w n$ ”

B-Trees

Fusion Trees

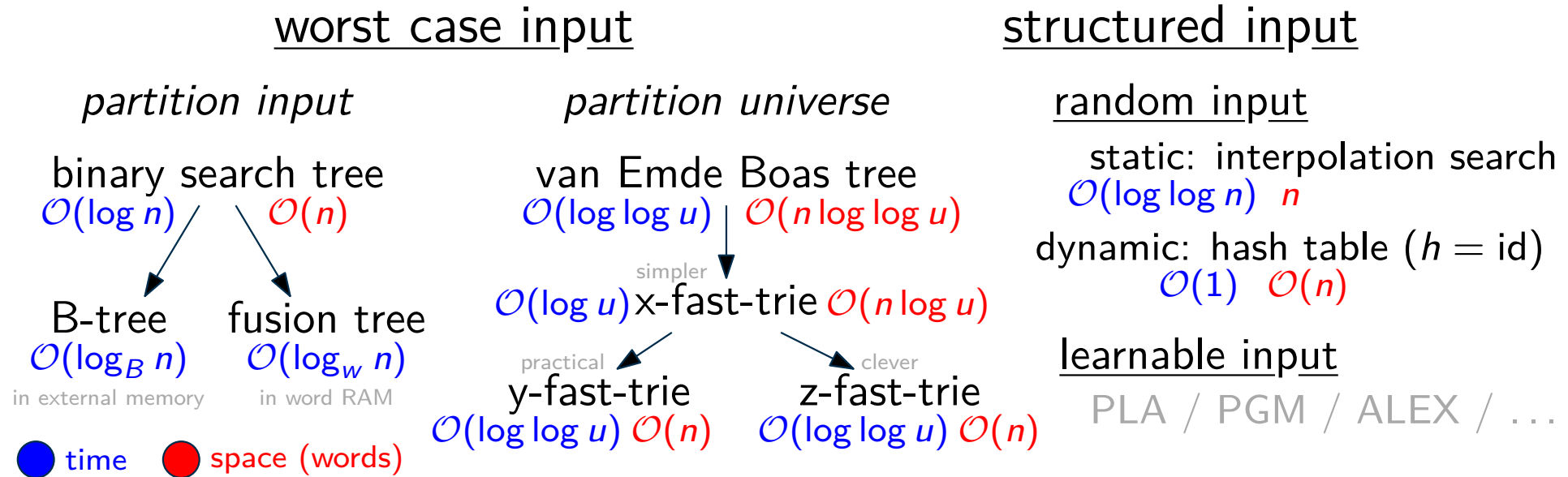
3. Splitting the Universe: “ $\log \log u$ ”

Van Emde Boas Tree

x/y/z-Fast-Tries

4. Conclusion

Predecessor Data Structures: Overview



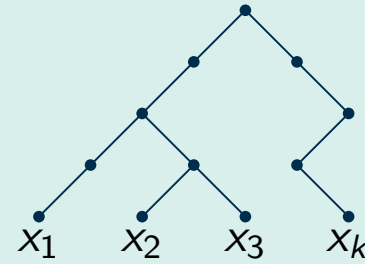
x-fast tries

Theorem

x-fast tries support

- space: $\mathcal{O}(n \log u)$
- queries: $\mathcal{O}(\log \log u)$ // expected
- insert/delete $\mathcal{O}(\log u)$ // expected

Idea: trie of input set S



x_1 0000
 x_2 0010
 x_3 0011
 x_4 1101

Data structure: hash table H

- one entry for every node of the trie
- maps p to MIN/MAX in subtree

prefix \rightarrow MIN, MAX

ε	\rightarrow 0000, 1101
0	\rightarrow 0000, 0011
00	\rightarrow 0000, 0011
000	\rightarrow 0000, 0000
0000	\rightarrow 0000, 0000
001	\rightarrow 0010, 0011
0010	\rightarrow 0010, 0010
0011	\rightarrow 0011, 0011
1	\rightarrow 1101, 1101
11	\rightarrow 1101, 1101
110	\rightarrow 1101, 1101
1101	\rightarrow 1101, 1101

- also: linked list of elements of S
- actually: suffices to store MIN or MAX

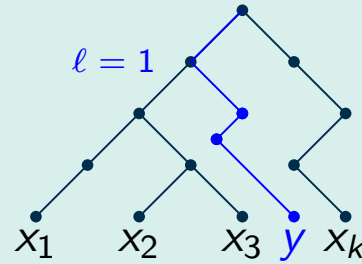
x-fast tries

Theorem

x-fast tries support

- space: $\mathcal{O}(n \log u)$
- queries: $\mathcal{O}(\log \log u)$ // expected
- insert/delete $\mathcal{O}(\log u)$ // expected

Idea: trie of input set S



x_1 0000
 x_2 0010
 x_3 0011
 x_4 1101

Data structure: hash table H

- one entry for every node of the trie
- maps p to MIN/MAX in subtree

prefix \rightarrow MIN, MAX

ε	\rightarrow 0000, 1101
0	\rightarrow 0000, 0011
00	\rightarrow 0000, 0011
000	\rightarrow 0000, 0000
0000	\rightarrow 0000, 0000
001	\rightarrow 0010, 0011
0010	\rightarrow 0010, 0010
0011	\rightarrow 0011, 0011
1	\rightarrow 1101, 1101
11	\rightarrow 1101, 1101
110	\rightarrow 1101, 1101
1101	\rightarrow 1101, 1101

- also: linked list of elements of S
- actually: suffices to store MIN or MAX

x-fast tries

Theorem

x-fast tries support

- space: $\mathcal{O}(n \log u)$
- queries: $\mathcal{O}(\log \log u)$ // expected
- insert/delete $\mathcal{O}(\log u)$ // expected

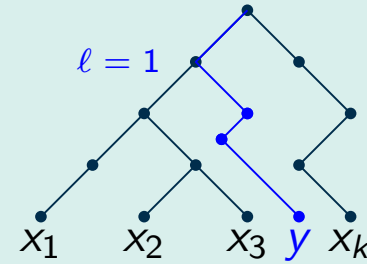
Algorithm locate(y):

```

if  $H$  contains  $y$  then
  _return " $y$  is in  $S$ "
binary search for the largest  $\ell$ 
  with  $y[1..\ell] \in \text{keys}(H)$ 
 $p \leftarrow y[1..\ell]$ 
if  $y[\ell + 1] = 1$  then
  //  $y$  falls rightwards off tree
  return " $\text{pred}(y) = H[p].\text{MAX}$ "
else
  //  $y$  falls leftwards off tree
  return " $\text{succ}(y) = H[p].\text{MIN}$ "
  
```

Problem Statement and Simple Solutions
○○○○○

Idea: trie of input set S



x_1 0000
 x_2 0010
 x_3 0011
 x_4 1101

Algorithm pred(y):

```

/* answer using locate( $y$ ) and linked list
   on elements of  $S$  */
  
```

Search Trees of High Degree: " $\log_w n$ "
○○○○○○○○○

Data structure: hash table H

- one entry for every node of the trie
- maps p to MIN/MAX in subtree

prefix \rightarrow MIN, MAX

ϵ	\rightarrow 0000, 1101
0	\rightarrow 0000, 0011
00	\rightarrow 0000, 0011
000	\rightarrow 0000, 0000
0000	\rightarrow 0000, 0000
001	\rightarrow 0010, 0011
0010	\rightarrow 0010, 0010
0011	\rightarrow 0011, 0011
1	\rightarrow 1101, 1101
11	\rightarrow 1101, 1101
110	\rightarrow 1101, 1101
1101	\rightarrow 1101, 1101

- also: linked list of elements of S
- actually: suffices to store MIN or MAX

Splitting the Universe: " $\log \log u$ "
○○○○○●○○

Conclusion
○○

x-fast tries

Theorem

x-fast tries support

- space: $\mathcal{O}(n \log u)$
- queries: $\mathcal{O}(\log \log u)$ // expected
- insert/delete $\mathcal{O}(\log u)$ // expected

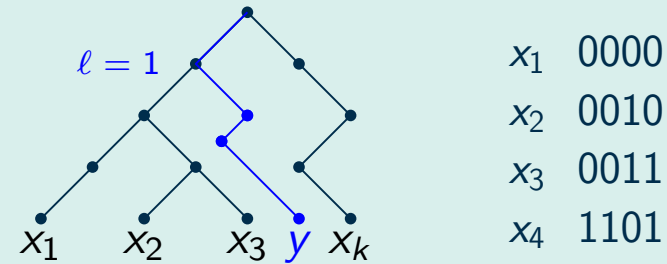
Algorithm locate(y):

```

if  $H$  contains  $y$  then
  _return " $y$  is in  $S$ "
binary search for the largest  $\ell$ 
  with  $y[1..\ell] \in \text{keys}(H)$ 
 $p \leftarrow y[1..\ell]$ 
if  $y[\ell + 1] = 1$  then
  //  $y$  falls rightwards off tree
  return " $\text{pred}(y) = H[p].\text{MAX}$ "
else
  //  $y$  falls leftwards off tree
  return " $\text{succ}(y) = H[p].\text{MIN}$ "
  
```

Problem Statement and Simple Solutions
○○○○○

Idea: trie of input set S



Algorithm pred(y):

```

/* answer using locate( $y$ ) and linked list
   on elements of  $S$  */
  
```

Algorithm insert(y):

```

locate( $y$ )
add  $y$  to linked list
foreach prefix  $p$  of  $y$  do
  _create/update entry for  $p$  in  $H$ 
  
```

Algorithm delete(y):

```

... // reverse of insert
  
```

Search Trees of High Degree: " $\log_w n$ "
○○○○○○○○○

Data structure: hash table H

- one entry for every node of the trie
- maps p to MIN/MAX in subtree

prefix \rightarrow MIN, MAX

ϵ	\rightarrow 0000, 1101
0	\rightarrow 0000, 0011
00	\rightarrow 0000, 0011
000	\rightarrow 0000, 0000
0000	\rightarrow 0000, 0000
001	\rightarrow 0010, 0011
0010	\rightarrow 0010, 0010
0011	\rightarrow 0011, 0011
1	\rightarrow 1101, 1101
11	\rightarrow 1101, 1101
110	\rightarrow 1101, 1101
1101	\rightarrow 1101, 1101

- also: linked list of elements of S
- actually: suffices to store MIN or MAX

Splitting the Universe: " $\log \log u$ "
○○○○○○●○○

Conclusion
○○

y-fast tries

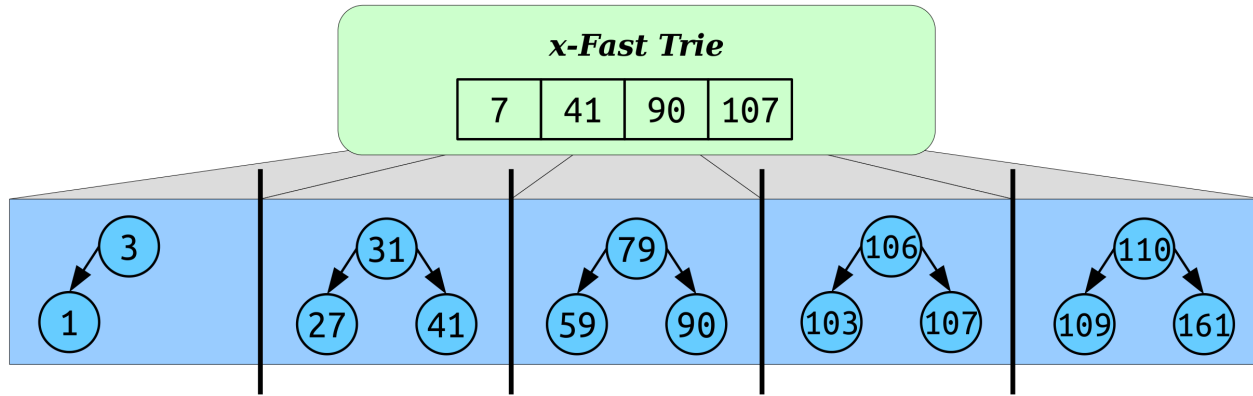


illustration from Stanford lecture CS 166

<https://web.stanford.edu/class/archive/cs/cs166/cs166.1206/lectures/15/Small15.pdf>

Total Space:

Idea

- elements stored in $\mathcal{O}(n/\log u)$ BSTs
 - ↪ between $\frac{1}{2} \log u$ and $2 \log u$ elements per BST
- x-fast trie stores $n' = \mathcal{O}(n/\log u)$ splitters

y-fast tries

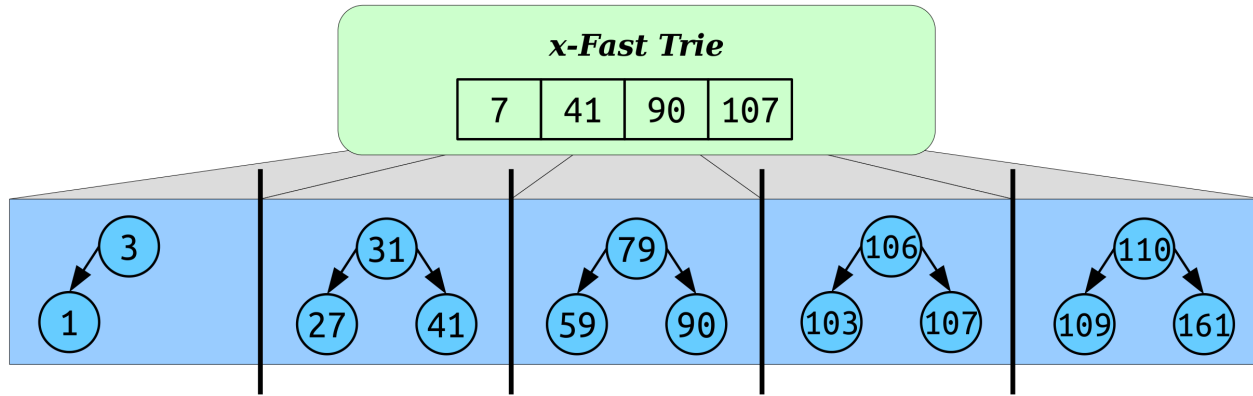


illustration from Stanford lecture CS 166

<https://web.stanford.edu/class/archive/cs/cs166/cs166.1206/lectures/15/Small15.pdf>

Idea

- elements stored in $\mathcal{O}(n/\log u)$ BSTs
 \hookrightarrow between $\frac{1}{2} \log u$ and $2 \log u$ elements per BST
- x-fast trie stores $n' = \mathcal{O}(n/\log u)$ splitters

Total Space: $\mathcal{O}(n)$

$\mathcal{O}(n' \log u)$ for x-fast trie and $\mathcal{O}(n)$ for BSTs

Running Times: $\mathcal{O}(\log \log u)$

- pred/succ in $\mathcal{O}(\log \log u)$

y-fast tries

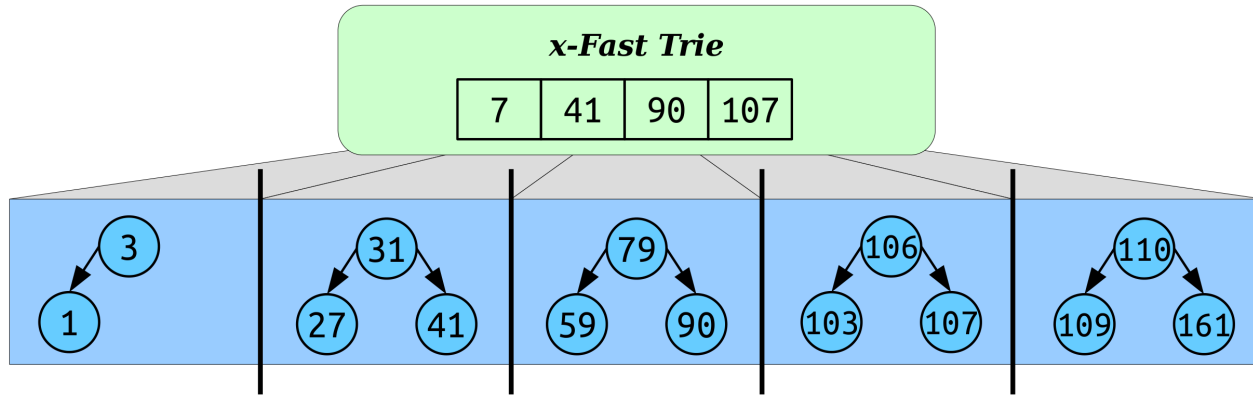


illustration from Stanford lecture CS 166

<https://web.stanford.edu/class/archive/cs/cs166/cs166.1206/lectures/15/Small15.pdf>

Idea

- elements stored in $\mathcal{O}(n/\log u)$ BSTs
 - ↪ between $\frac{1}{2} \log u$ and $2 \log u$ elements per BST
- x-fast trie stores $n' = \mathcal{O}(n/\log u)$ *splitters*

Total Space: $\mathcal{O}(n)$

$\mathcal{O}(n' \log u)$ for x-fast trie and $\mathcal{O}(n)$ for BSTs

Running Times: $\mathcal{O}(\log \log u)$

- pred/succ in $\mathcal{O}(\log \log u)$
 - $\mathcal{O}(\log \log u)$ for navigating in x-fast trie to BST
 - $\mathcal{O}(\log \log u)$ for navigating within BST
- insert in *amortised* expected time $\mathcal{O}(\log \log u)$

y-fast tries

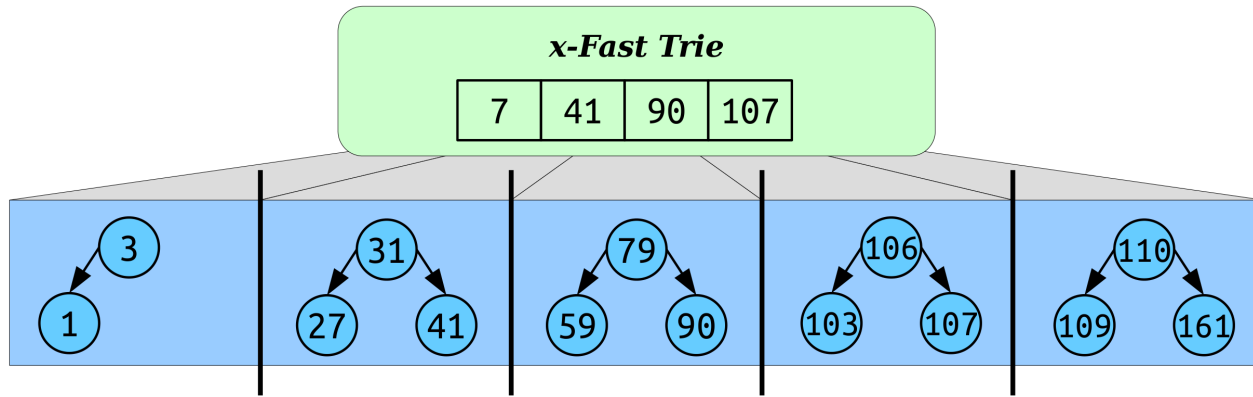


illustration from Stanford lecture CS 166

<https://web.stanford.edu/class/archive/cs/cs166/cs166.1206/lectures/15/Small15.pdf>

Idea

- elements stored in $\mathcal{O}(n/\log u)$ BSTs
 - ↔ between $\frac{1}{2} \log u$ and $2 \log u$ elements per BST
- x-fast trie stores $n' = \mathcal{O}(n/\log u)$ *splitters*

Total Space: $\mathcal{O}(n)$

$\mathcal{O}(n' \log u)$ for x-fast trie and $\mathcal{O}(n)$ for BSTs

Running Times: $\mathcal{O}(\log \log u)$

- pred/succ in $\mathcal{O}(\log \log u)$
 - $\mathcal{O}(\log \log u)$ for navigating in x-fast trie to BST
 - $\mathcal{O}(\log \log u)$ for navigating within BST
- insert in *amortised* expected time $\mathcal{O}(\log \log u)$
 - $\mathcal{O}(\log \log u)$ for navigating to BST
 - $\mathcal{O}(\log \log u)$ for inserting into BST
 - if BST full amortised $\mathcal{O}(1)$
 - split BST $\mathcal{O}(\log u)$ time
 - insert into x-fast trie $\mathcal{O}(\log u)$ time

y-fast tries

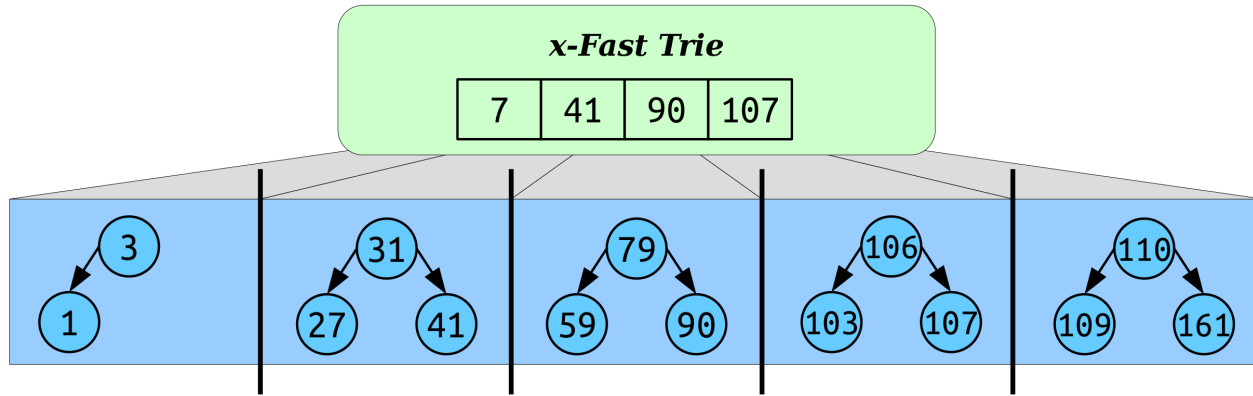


illustration from Stanford lecture CS 166

<https://web.stanford.edu/class/archive/cs/cs166/cs166.1206/lectures/15/Small15.pdf>

Idea

- elements stored in $\mathcal{O}(n/\log u)$ BSTs
 - ↔ between $\frac{1}{2} \log u$ and $2 \log u$ elements per BST
- x-fast trie stores $n' = \mathcal{O}(n/\log u)$ *splitters*

Total Space: $\mathcal{O}(n)$

$\mathcal{O}(n' \log u)$ for x-fast trie and $\mathcal{O}(n)$ for BSTs

Running Times: $\mathcal{O}(\log \log u)$

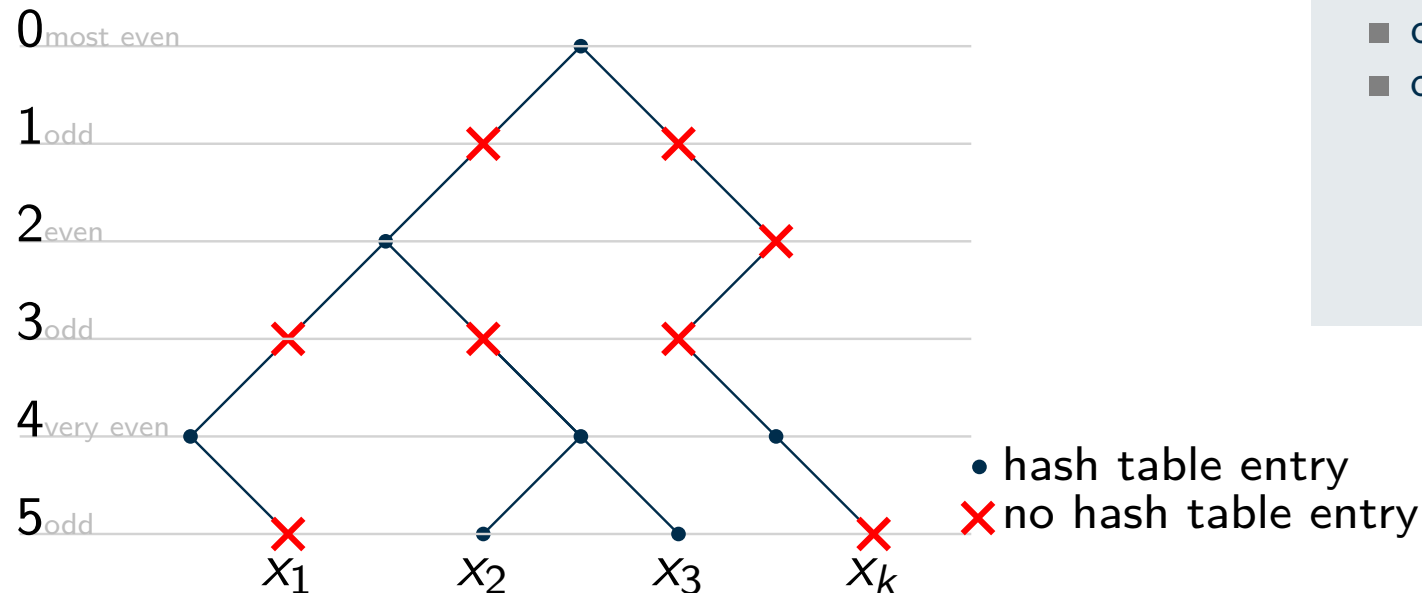
- pred/succ in $\mathcal{O}(\log \log u)$
 - $\mathcal{O}(\log \log u)$ for navigating in x-fast trie to BST
 - $\mathcal{O}(\log \log u)$ for navigating within BST
- insert in *amortised* expected time $\mathcal{O}(\log \log u)$
 - $\mathcal{O}(\log \log u)$ for navigating to BST
 - $\mathcal{O}(\log \log u)$ for inserting into BST
 - if BST full amortised $\mathcal{O}(1)$
 - split BST $\mathcal{O}(\log u)$ time
 - insert into x-fast trie $\mathcal{O}(\log u)$ time
- delete in *amortised* expected time $\mathcal{O}(\log \log u)$ // similar

Remark: z-fast tries (for details see [BBV2010])

Theorem

z-fast tries support

- space: $\mathcal{O}(n)$
- queries: $\mathcal{O}(\log \log u)$ // expected
- insert/delete $\mathcal{O}(\log \log u)$ // expected



Main idea

- contract non-branching paths
- one hash table entry per remaining edge e
 - key: the prefix p along e where $|p|$ is “most even”
// i.e. divisible by the highest power of 2
 - values: various pointers for jumping around
// MIN/MAX too expensive to update and not used

Summary: Data Structures for Sorted Sequences

data structure	space (* = bits)	time for pred	dynamic?	remarks
balanced BST	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	✓	e.g. red-black-trees, AVL-Trees, etc
binary search	just the array	$\mathcal{O}(\log n)$	✗	Eytzinger layout
bitvector	$\mathcal{O}(u)^*$	$\mathcal{O}(1)$	✗	rank/select
Elias-Fano	$\mathcal{O}(n(1 + \log u/n))^*$	$\mathcal{O}(1 + \log u/n)$	✗	compressed bitvector + binary search
B-Tree	$\mathcal{O}(n)$	$\mathcal{O}(\log_B n)$ // IO-model	✓	highly practical
Fusion Tree	$\mathcal{O}(n)$	$\mathcal{O}(\log_w n)$	(✓)	\sqrt{w} -way splits
vEB Tree	$\mathcal{O}(u)$	$\mathcal{O}(\log \log u)$	✓	$\mathcal{O}(n \log \log u)$ space with hashing
↪ x-fast-trie	$\mathcal{O}(n \log u)$	$\mathcal{O}(\log \log u)$ // $\mathcal{O}(\log u)$ for insert	✓	one large hash table
↪ y-fast-trie	$\mathcal{O}(n)$	$\mathcal{O}(\log \log u)$	✓	x-fast-trie + BST
↪ z-fast-trie	$\mathcal{O}(n)$	$\mathcal{O}(\log \log u)$	(✓)	x-fast-trie but one hash table entry per trie branch

Possible Exam Questions

- What operations are supported by (dynamic) data structures for sorted sequences?
- What base-line data structures for the predecessor problem would you expect in any standard library?
- What performance does a bitvector offer? When is it a good solution?
- What performance does Elias-Fano offer? When is this a good solution?
- B-Trees:
 - What is the idea?
 - What makes them faster than BSTs in practice?
 - In what model of computation are they faster than BSTs in theory?
- Fusion Trees:
 - In what way are they like B-Trees? In what way are the goals different?
 - What is the search time in Fusion Trees? What does an individual Fusion Tree node support?
 - Explain the idea of sketching. Explain the “routeQuery” algorithm in detail.
- van Emde Boas Trees:
 - Explain the recursive structure of a van Emde Boas Tree node.
 - Describe (one of) the operations pred/succ/insert/delete.
 - Derive the operation’s running time.
 - What would go wrong if MIN were stored recursively?
 - What is the space requirement (at first)?
 - How can it be reduced? Does that change the running times?
- x/y/z-fast-tries:
 - Explain their structure, space requirement and running times.
 - What are their advantages and disadvantages compared to each other and to vEB trees?