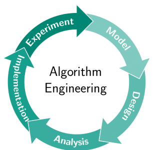
Predecessor Data Structures: Fusion Trees and van Emde Boas Trees

Stefan Walzer, Ragnar Groot Koerkamp, Stefan Hermann | compiled on 1. Juni 2026



KIT

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

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

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

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: Self-Balancing Binary Search Trees

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

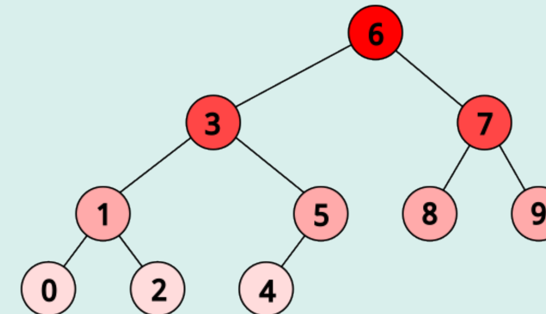
BST in Practice

- 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: 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 $\neq$ None
- $\text{succ}(i) = \text{select}(\text{rank}(i + 1))$ // assuming $\neq$ None

Compressed Storage using Elias Fano

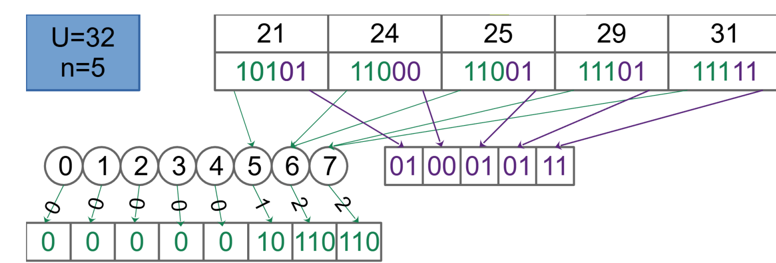


illustration by Wikipedia user Trobolt

https://commons.wikimedia.org/wiki/File:Elias-fano_1.svg

Theorem: Elias-Fano as Predecessor Structure

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

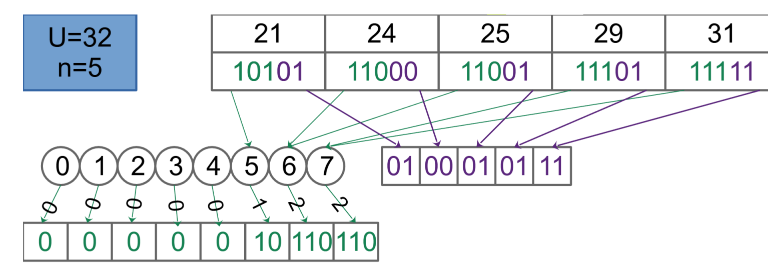
- at most $n(b+3) + o(n)$ bits of space
- $\mathcal{O}(1+b)$ time operations

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

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

Predecessor Search Implementation

Algorithm pred(x):

// split x into high and low bits
(high,low) $\leftarrow (\lfloor x/2^b \rfloor, x \bmod 2^b)$
// find range of elements with matching high bits
$(i', j') \leftarrow (B.\text{select}_0(\text{high} - 1), B.\text{select}_0(\text{high}))$ // select$(-1) := -1$
$(i, j) \leftarrow (i' - \text{high} + 1, j' - \text{high})$ // relevant range is $A[i..j)$
if $i = j$ *or* $A[i] \geq$ *low* **then**
  if $i = 0$ **then return** None
  high$' \leftarrow \text{select}_1(i-1) - (i-1)$
  return high$' \cdot 2^b + A[i-1]$
else
  low$' \leftarrow$ pred of low in $A[i..j)$ // binary search on up to 2^b elements
  return high $\cdot\, 2^b +$ low$'$

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

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

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

KIT

1. Problem Statement and Simple Solutions

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

B-Trees

Fusion Trees

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

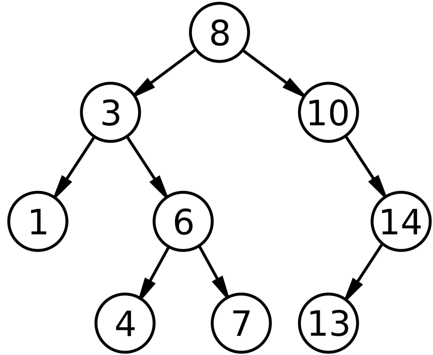
Van Emde Boas Tree

$x/y/z$-Fast-Tries

4. Conclusion

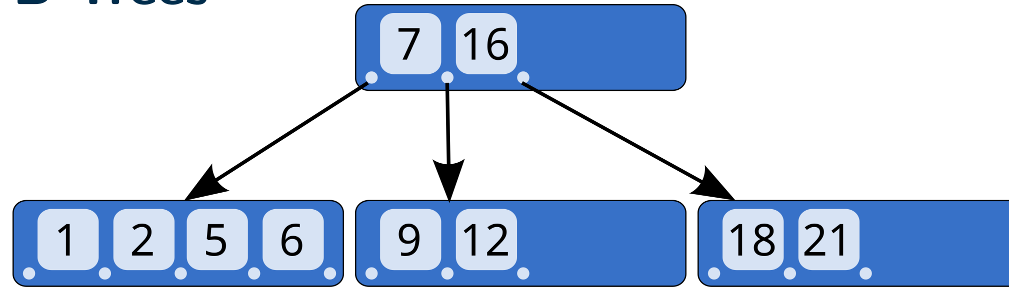
B-Trees

Binary Search Tree



- each node stores one splitter
- depth $\mathcal{O}(\log_2 n)$
  $\hookrightarrow$ 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 $\mathcal{O}(\log_B n)$
  $\hookrightarrow$ 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 static[1] 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(**00000**10101101111) = 5
    - pext(src = **0**1**0**1**0**1**0**1**1111**1**0**0**00**, mask = 1001011011110011) = 000000**0110111100**
  - similar behaviour can be simulated with c-like instructions // https://en.wikipedia.org/wiki/Fusion_tree

[1] Dynamic versions are also known.

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

Todo

- Given splitters $x_1, \ldots, x_k \in \{0,1\}^w$
- Given key $y \in \{0,1\}^w$
- Find: $0 \le i \le k$ with $x_i \le y < x_{i+1}$

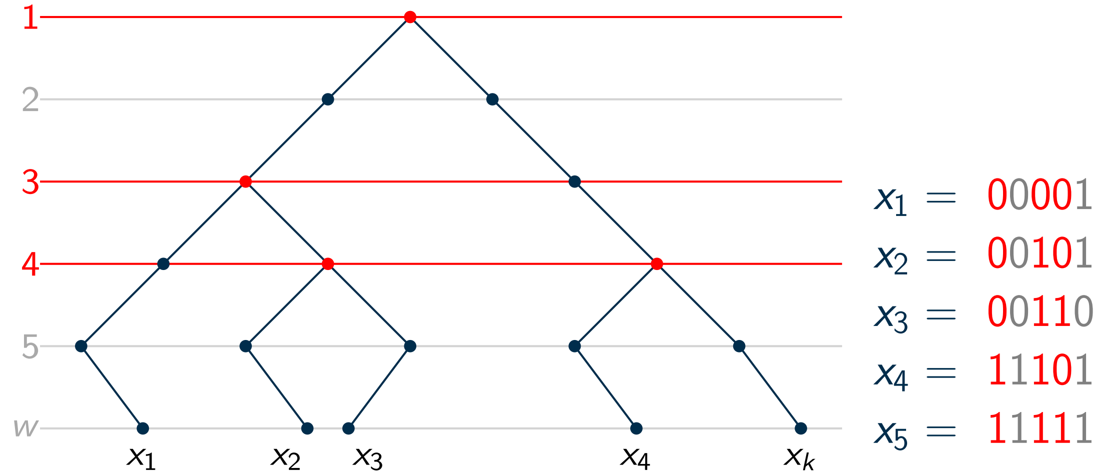
  // implicitly $x_0 = -\infty$, $x_{k+1} = \infty$

Challenge

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

Idea

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



- compute word indicating branching bits
  $\hookrightarrow$ branchBits $= 101100$ // note: $\le k-1$ bits set
- define sketch $\tilde{x} :=$ pext(branchBits, x)
  $\hookrightarrow$ $x_2 = 00101 \rightsquigarrow \tilde{x}_2 = 010$
  // pext = parallel bit extract
- concatenate sketches $\tilde{x}_1, \ldots, \tilde{x}_k$ into $\widetilde{X}$, separated by ones // ones relevant on next slide
  $\hookrightarrow$ $\widetilde{X} = 1\,\underbrace{000}_{\tilde{x}_1}\,1\,\underbrace{010}_{\tilde{x}_2}\,1\,\underbrace{011}_{\tilde{x}_3}\,1\,\underbrace{110}_{\tilde{x}_4}\,1\,\underbrace{111}_{\tilde{x}_5} \in \{0,1\}^{\mathcal{O}(w)}$

Parallel comparison of sketches

Algorithm rank($\tilde{y} \in \{0,1\}^k, \widetilde{X} = 1\tilde{x}_1 \dots 1\tilde{x}_k$)**:**

// goal: output $0 \le i \le k$ such that $\tilde{x}_i < \tilde{y} \le \tilde{x_{i+1}}$

$\tilde{Y} \leftarrow 0^k 1 0^k 1 \dots 0^k 1 \cdot y$ // $0\tilde{y}0\tilde{y}\dots0\tilde{y}$

$d \leftarrow (\widetilde{X} - \tilde{Y}) \& (10^k 10^k \dots 10^k)$ // "carry" bits indicating $\tilde{y} \ge \tilde{x}_i$

return $\mathrm{clz}(d)/(k+1)$ // clz = count leading zeroes

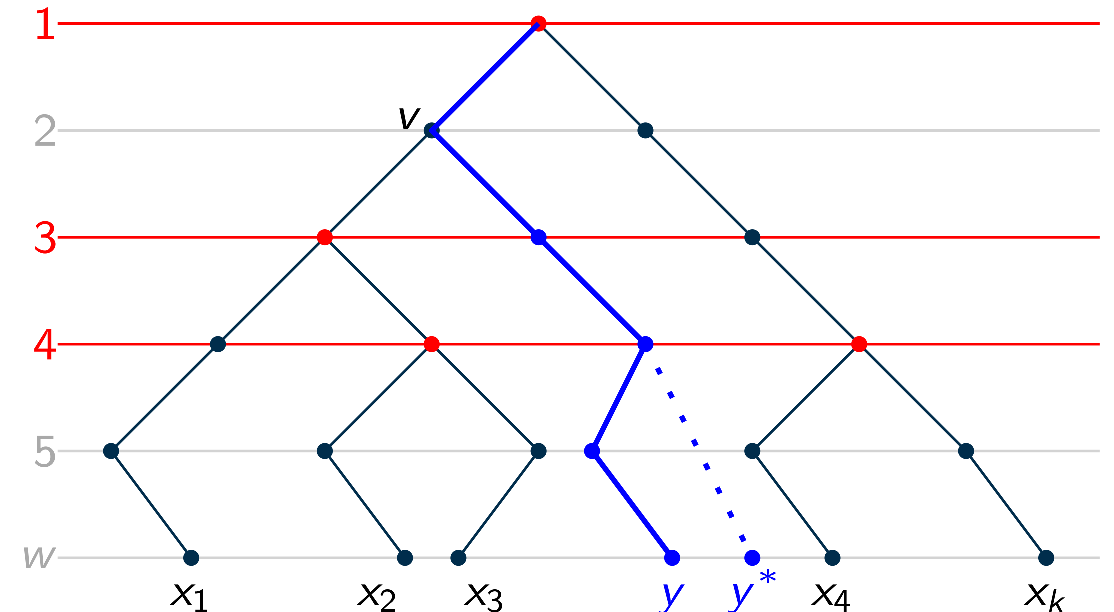
Intuition

variable		what it encodes (example)
$\tilde{y}$	$=$	7
$\widetilde{X}$	$=$	$(2,5,7,8,9)$
$\widetilde{Y}$	$=$	$(7,7,7,7,7)$
$\widetilde{X} - \widetilde{Y}$	$=$	$(-5,-2,+0,+1,+2)$
d	$=$	$(-,-,+,+,+)$
$\mathrm{clz}(d)/(k+1)$	$=$	2

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

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

```
// assume X̃ and branchBits precomputed
ỹ ← sketch(y)
r ← rank(ỹ, X̃) // r = 1 in examples
if x_{r+1} = y then return r + 1
// y "falls off" the tree at some node v of depth d
// in "sketch space" y navigates erratically below v
// but: x_r or x_{r+1} or both traverse v
// use longest common prefix lcp(x, y) := clz(x ⊕ y)
d ← max(lcp(y, x_r), lcp(y, x_{r+1})) + 1
if bit d of y is 1 then // y "falls off" to the right
    // predecessor of y: go to v, then right at all branch bits
    y* ← y[1...d] ∘ 1^{w−d}
    return rank(sketch(y*) + 1, X̃) // "+1" converts "≤" into "<"
else
    y* ← y[1...d] ∘ 0^{w−d} // analogous
    return rank(sketch(y*), X̃) // no "+1"
```



KIT

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 $S \subseteq \{0, \ldots, u-1\}$ with $\mathcal{O}(\log w)$-time operations. // assumes word size $w \geq \log u$

Idea

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

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

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

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

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

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

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

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 ≥ T.MAX then  return None
if x < T.MIN then  return T.MIN
⟨h, ℓ⟩ ← x // split x into w/2 high bits and w/2 low bits
if T.blocks[h].MIN = None or x ≥ T.blocks[h].MAX then
    h′ ← succ(T.summary, h)
    return ⟨h′, T.blocks[h′].MIN⟩
else
    return ⟨h, succ(T.blocks[h], ℓ)⟩
```

Algorithm pred(T, x):

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

Algorithm insert(T, x):

```
if base case then  [...] // todo
if T.MIN = None then
    T.MIN ← x, T.MAX ← x
    return
if T.MIN = x then return
if x < T.MIN then
    // next: insert old min
    swap(T.MIN,x)
T.MAX ← max(T.MAX, x)
⟨h, ℓ⟩ ← x // split
if T.blocks[h].MIN = None then
    insert(T.summary, h)
    // in this case: next insertion O(1)!
insert(T.blocks[h], x)
```

Algorithm delete(T, x):

```
if base case then  [...] // todo
if x = T.MIN = T.MAX then
    T.MIN ← None
    T.MAX ← None
    return
if x = T.MIN then
    T.MIN ← . . . // update in O(1)
    // delete T.MIN from substructures
    x ← T.MIN
⟨h, ℓ⟩ ← x
delete(T.blocks[h], ℓ)
if T.blocks.MIN = None then
    // note: previous call was O(1)
    delete(T.summary, h)
T.MAX ← . . . // update in O(1)
```

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

Space Usage of Van Emde Boas Trees

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

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

Proof: “don’t store empty structures”

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

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

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

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

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

$S(u) = \underbrace{\sqrt{u}}_{\text{array}} + \underbrace{(\sqrt{u}+1) \cdot S(\sqrt{u})}_{\text{substructures}} + \mathcal{O}(1)$ 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://infoweekly.blogspot.com/2010/09/van-emde-boas-and-its-space-complexity.html

[a] This is not trivial.

KIT

1. Problem Statement and Simple Solutions

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

B-Trees

Fusion Trees

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

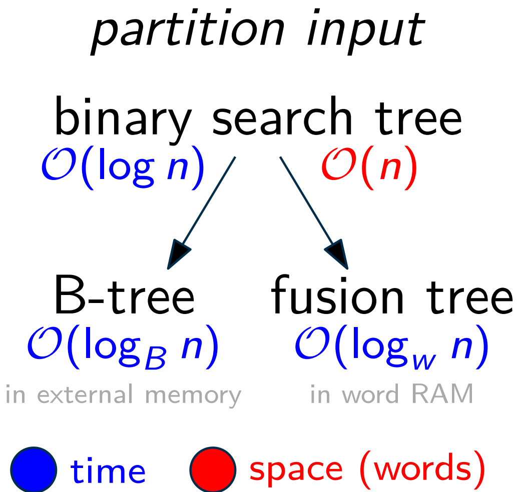
Van Emde Boas Tree

$x/y/z$-Fast-Tries

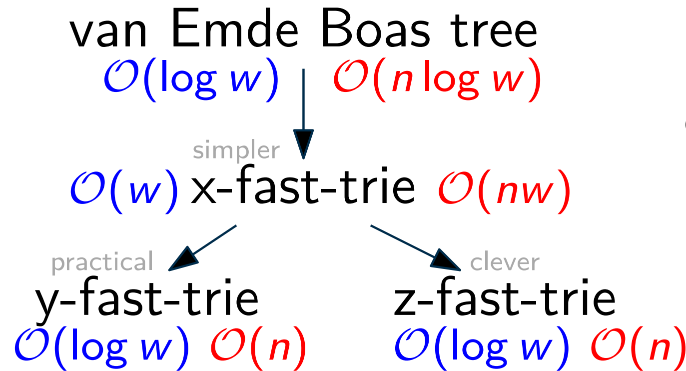
4. Conclusion

Predecessor Data Structures: Overview

worst case input



partition universe



structured input

random input

static: interpolation search
$\mathcal{O}(\log \log n)$ n

dynamic: hash table ($h = \text{id}$)
$\mathcal{O}(1)$ $\mathcal{O}(n)$

learnable input

PLA / PGM / ALEX / ...

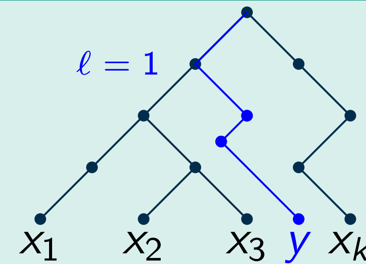
x-fast tries

Theorem

x-fast tries support

- space: $\mathcal{O}(nw)$
- queries: $\mathcal{O}(\log w)$ // expected
- insert/delete $\mathcal{O}(w)$ // 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	$\to$ MIN, MAX
ε	$\to$0000, 1101
0	$\to$0000, 0011
00	$\to$0000, 0011
000	$\to$0000, 0000
0000	$\to$0000, 0000
001	$\to$0010, 0011
0010	$\to$0010, 0010
0011	$\to$0011, 0011
1	$\to$1101, 1101
11	$\to$1101, 1101
110	$\to$1101, 1101
1101	$\to$1101, 1101

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

Algorithm locate(y):

```
if H contains y then
    return "y is in S"
// find using binary search on ℓ
p ← longest prefix of y in H
ℓ ← length of p
if bit ℓ + 1 of y is 1 then
    // y falls rightwards off tree
    return "pred(y) = H[p].MAX"
else
    // y falls leftwards off tree
    return "succ(y) = H[p].MIN"
```

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
```

y-fast tries

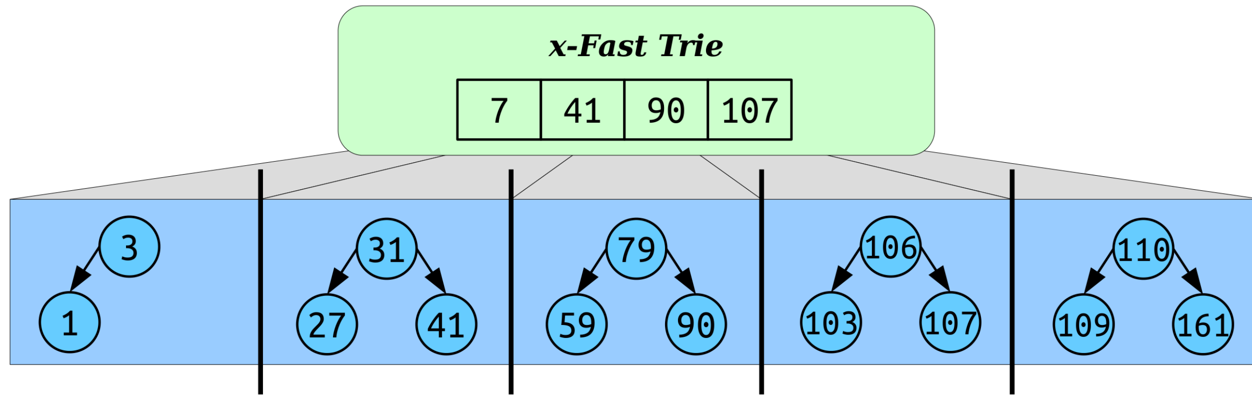


illustration from Stanford lecture CS 166

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

Idea

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

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

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

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

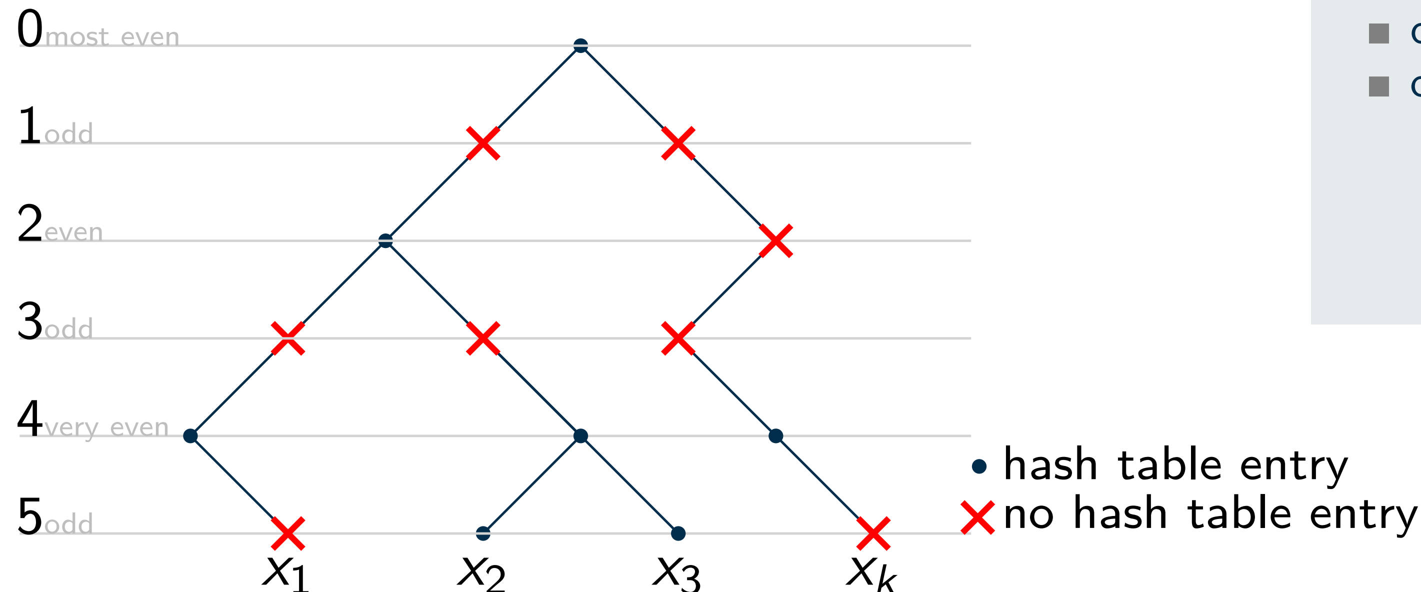
- pred/succ in $\mathcal{O}(\log w)$
  - $\mathcal{O}(\log w)$ for navigating to BST
  - $\mathcal{O}(\log w)$ for navigating within BST
- insert in *amortised* expected time $\mathcal{O}(\log w)$
  - $\mathcal{O}(\log w)$ for navigating to BST
  - $\mathcal{O}(\log w)$ for inserting into BST
  - if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

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

Theorem

z-fast tries support

- space: $\mathcal{O}(n)$
- queries: $\mathcal{O}(\log w)$ // expected
- insert/delete $\mathcal{O}(\log w)$ // 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	$\mathcal{O}(n)$	$\mathcal{O}(\log_B n)$	✓	highly practical
Fusion Tree	$\mathcal{O}(n)$	$\mathcal{O}(\log_w n)$	(✓)	$\sqrt{w}$-way splits
vEB Tree	$\mathcal{O}(u)$	$\mathcal{O}(\log w)$ // $= \mathcal{O}(\log \log u)$	✓	$\mathcal{O}(n \log w)$ space with hashing
↪ x-fast-trie	$\mathcal{O}(nw)$	$\mathcal{O}(\log w)$ // $\mathcal{O}(w)$ for insert	✓	one large hash table
↪ y-fast-trie	$\mathcal{O}(n)$	$\mathcal{O}(\log w)$	✓	x-fast-trie + BST
↪ z-fast-trie	$\mathcal{O}(n)$	$\mathcal{O}(\log w)$	(✓)	x-fast-trie but one hash table entry per trie branch

Possible Exam Questions

- What operations are supported by (dynamic) data structured 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 solutions?
- What performance does Elias-Fano offer? When is this a good solutions?
- 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 Emda 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?