

ADVANCED

overall tree
= binary tree of mini trees

mini trees

micro trees

actual nodes

n nodes

$\lg n$ nodes

$\frac{1}{4} \lg n$ nodes



DATA STRUCTURES


5

Integer Data Structures

Advanced Data Structures · Summer 2026

Prof. Dr. Sebastian Wild

5 Integer Data Structures

- 5.1 Integer Dictionaries
 - 5.2 Perfect Hashing
 - 5.3 Dynamic Perfect Hashing
 - 5.4 Binary Tries
 - 5.5 x-Fast Tries
 - 5.6 y-Fast Tries
 - 5.7 Lower Bounds
- 

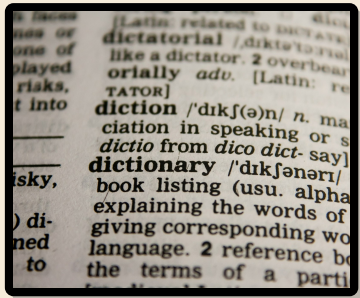
5.1 Integer Dictionaries

Recall: Symbol table ADT

Java: `java.util.Map<K,V>`, C++: `std::(unordered)_map`

Symbol table / Dictionary / Map / Associative array / key-value store:

Python dict {k:v}



- ▶ `put(k, v)` Python dict: `d[k] = v`
Put key-value pair (k, v) into table
- ▶ `get(k)` Python dict: `d[k]`
Return value associated with key k
- ▶ `delete(k)` Python dict: `del d[k]`
Remove key k (any associated value) from table
- ▶ `contains(k)` Python dict: `k in d`
Returns whether the table has a value for key k
- ▶ `isEmpty(), size(), create()`

Unlike before: Focus here on unordered case

- ▶ no rank-based queries
- ▶ values uninteresting
- ↪ summarize get and put as `lookup(k)`

Integer Dictionaries

For unsorted dictionaries, best solutions based on *hashing*.

- ▶ Whatever the actual objects we store, always apply a hash function first

↪ Consider here directly (dynamic) sets of integers

Hash Tables

Notation

- ▶ **universe** $U = [0..u)$ of allowed values
- ▶ **universe size** $u = |U|$, usually $u = 2^w$
- ▶ $S \subseteq U$ dynamic set of integers stored in our dictionary
- ▶ $n = |S|$ current **size** of set, $S = \{x_1, \dots, x_n\}$

Hash Tables

Notation

- ▶ **universe** $U = [0..u)$ of allowed values
 - ▶ **universe size** $u = |U|$, usually $u = 2^w$
 - ▶ $S \subseteq U$ dynamic **set** of integers stored in our dictionary
 - ▶ $n = |S|$ current **size** of set, $S = \{x_1, \dots, x_n\}$
 - ▶ use **hash table** $T[0..m)$ with m buckets
 - ▶ **hash function** $h : U \rightarrow [0..m)$
- ↪ element $x \in U$ stored in $T[h(x)]$

Hash Tables

Notation

- ▶ **universe** $U = [0..u)$ of allowed values
- ▶ **universe size** $u = |U|$, usually $u = 2^w$
- ▶ $S \subseteq U$ dynamic **set** of integers stored in our dictionary
- ▶ $n = |S|$ current **size** of set, $S = \{x_1, \dots, x_n\}$
- ▶ use **hash table** $T[0..m)$ with m **buckets**
- ▶ **hash function** $h : U \rightarrow [0..m)$
- ↔ element $x \in U$ stored in $T[h(x)]$
- ▶ in general: $T[h(x)]$ may be a **secondary data structure**
 - ▶ linked list ↔ *chaining hashing*
 - ▶ sequence of positions in T ↔ *open-addressing hashing*

Hash Tables

Notation

- ▶ **universe** $U = [0..u)$ of allowed values
- ▶ **universe size** $u = |U|$, usually $u = 2^w$
- ▶ $S \subseteq U$ dynamic **set** of integers stored in our dictionary
- ▶ $n = |S|$ current **size** of set, $S = \{x_1, \dots, x_n\}$
- ▶ use **hash table** $T[0..m)$ with m **buckets**
- ▶ **hash function** $h : U \rightarrow [0..m)$
- ↔ element $x \in U$ stored in $T[h(x)]$
- ▶ in general: $T[h(x)]$ may be a **secondary data structure**
 - ▶ linked list ↔ *chaining hashing*
 - ▶ sequence of positions in T ↔ *open-addressing hashing*
- ▶ if $h(x) = h(y)$, we have a **collision** (between x and y)

Hashing must be randomized

Collisions are inevitable

- ▶ usually want S represented with $O(n)$ space

↪ $m = O(n)$

- ▶ typically thus $|U| = 2^w \gg m$

↪ h must have many collisions

↪ For any *fixed* hash function h , worst-case set S has many collisions

↪ cannot have meaningful worst-case time bounds (even if typical case good)

365 \approx 25 people to get 50%
collision for birthdays

$n \approx 2^{2w}$

Hashing must be randomized

Collisions are inevitable

- ▶ usually want S represented with $O(n)$ space

↪ $m = O(n)$

- ▶ typically thus $|U| = 2^w \gg m$

↪ h must have many collisions

↪ For any *fixed* hash function h , worst-case set S has many collisions

↪ cannot have meaningful worst-case time bounds (even if typical case good)

Randomized Hashing

- ▶ draw *random* $h \in \mathcal{H}$

↪ running time expected over random choice, worst-case w.r.t. S

- ▶ allowing all $\mathcal{H} = \{h \mid h : U \rightarrow [0..m)\}$ requires m^u space ⚡

$$\log_2(m^u) = u \cdot \log_2 m$$

↪ must choose sufficiently small \mathcal{H}

Universal Hashing

Many guarantees of varying strength studied for \mathcal{H} ... here only simplest needed

Definition 5.1 (Universal Family)

Let \mathcal{H} be a set of hash functions from U to $[m]$ and $|U| \geq m$.

Assume $h \in \mathcal{H}$ is chosen uniformly at random.

for fully random h

$$h(x_1) \stackrel{D}{=} \mathcal{U}([m])$$

$$h(x_2) \stackrel{D}{=} \mathcal{U}([m])$$

$$\Rightarrow \mathbb{P}[h(x_1) = h(x_2)] = \frac{1}{m}$$

(a) Then \mathcal{H} is called a *c-universal* if

$$\forall x_1, x_2 \in U : x_1 \neq x_2 \implies \mathbb{P}[h(x_1) = h(x_2)] \leq \frac{c}{m}.$$

(b) \mathcal{H} is called *strongly universal* or *pairwise independent* if

$$\forall x_1, x_2 \in U, y_1, y_2 \in R : x_1 \neq x_2 \implies \mathbb{P}[h(x_1) = y_1 \wedge h(x_2) = y_2] \leq \frac{1}{m^2}. \quad \blacktriangleleft$$

Examples of universal families

Universal families

$$\blacktriangleright \mathcal{H}_1 = \{h_{ab} : a \in [1..p), b \in [0..p)\} \quad (c = 2)$$

$$\blacktriangleright \mathcal{H}_0 = \{h_{ab} : a \in [0..p), b \in [0..p)\} \quad (c = 4)$$

$$\blacktriangleright \mathcal{H}_2 = \{h_a : a \in [1..2^k), a \text{ odd}\} \quad (c = 2)$$

$$h_{ab}(x) = (a \cdot x + b \bmod p) \bmod m \quad p \text{ prime}, p \geq m$$

$$h_a(x) = (a \cdot x \bmod 2^k) \operatorname{div} 2^{k-\ell} \quad u = 2^k, m = 2^\ell$$

What does universality buy us?

Lemma 5.2 (Universal collisions)

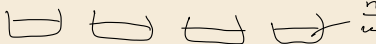
Consider hashing n keys x_1, \dots, x_n into m buckets using a random hash function h chosen from a c -universal family of hash functions \mathcal{H} . Let C denote the number of pairwise collisions, i. e., $C = |\{(i, j) : 1 \leq i < j \leq n \wedge h(x_i) = h(x_j)\}|$.

$$\text{Then } \mathbb{E}[C] < \frac{cn^2}{2m}.$$

$$C \leq \binom{n}{2}$$

$$C \geq \binom{\frac{n}{m}}{2} \cdot m \approx \frac{n^2}{2m} \cdot c \quad \blacktriangleleft$$

Proof
$$\mathbb{E}_h[C] = \sum_{1 \leq i < j \leq n} \mathbb{E}_h[C_{i,j}] \leq \binom{n}{2} \frac{c}{m} < \frac{cn^2}{2m}$$



$$C_{i,j} = [h(x_i) = h(x_j)]$$

$$\mathbb{E}_h[C_{i,j}] = \mathbb{P}_h[h(x_i) = h(x_j)] \leq \frac{c}{m}$$

What does universality buy us?

Lemma 5.2 (Universal collisions)

Consider hashing n keys x_1, \dots, x_n into m buckets using a random hash function h chosen from a c -universal family of hash functions \mathcal{H} . Let C denote the number of pairwise collisions, i. e., $C = |\{(i, j) : 1 \leq i < j \leq n \wedge h(x_i) = h(x_j)\}|$.

Then $\mathbb{E}[C] < \frac{cn^2}{2m}$.

Lemma 5.3 (Universal bin size)

Consider the scenario of Lemma 5.2. Let $X_j = |T[j]|$ be the number of elements in bucket j .

(a) $\mathbb{E}[X_{h(x_i)}] \leq 1 + c \cdot \frac{n}{m}$

(b) Then $\mathbb{P}\left[\max X_j > \sqrt{2c} \cdot \frac{n}{\sqrt{m}}\right] \leq \frac{1}{2}$

(a) $\mathbb{E}[X_{h(x_i)}] = 1 + \sum_{j \neq i} \mathbb{E}[C_{ij}] \leq 1 + (n-1) \frac{c}{m}$

(b) Any bucket w/ \hat{X} keys implies $\binom{\hat{X}}{2}$ pairwise collisions

$$\Rightarrow C \geq \binom{\hat{X}}{2} \geq \frac{(\hat{X}-1)^2}{2}$$

Note, $\hat{X} \geq \sqrt{\frac{2cn^2}{m}} + 1$ implies

$$\frac{(\hat{X}-1)^2}{2} \geq \frac{cn^2}{m} \quad \text{which implies} \quad C \geq \frac{cn^2}{m} \geq 2 \mathbb{E}[C]$$

\Rightarrow by Markov inequality $\boxed{\mathbb{P}[X \geq a \mathbb{E}[X]] \leq \frac{1}{a} \quad X \geq 0}$

$$\mathbb{P}\left\{\max X_i > \sqrt{2c} \cdot \frac{n}{\sqrt{m}}\right\} \leq \frac{1}{2}$$

5.2 Perfect Hashing

Perfect Hashing

A hash function $h : [u] \rightarrow [m]$ is called

- ▶ *perfect* for a set $S = \{x_1, \dots, x_n\} \subset [u]$ if $i \neq j$ implies $h(x_i) \neq h(x_j)$ $n \leq m$
- ▶ *minimal* for set $S = \{x_1, \dots, x_n\} \subset [u]$ if $m = n$

Perfect Hashing

A hash function $h : [u] \rightarrow [m]$ is called

- ▶ *perfect* for a set $S = \{x_1, \dots, x_n\} \subset [u]$ if $i \neq j$ implies $h(x_i) \neq h(x_j)$
- ▶ *minimal* for set $S = \{x_1, \dots, x_n\} \subset [u]$ if $m = n$

Perfect Hashing

- ▶ only possible for $n \leq m$
- ▶ stringent requirement \rightsquigarrow first focus on **static** \mathcal{S}

Perfect Hashing

A hash function $h : [u] \rightarrow [m]$ is called

- ▶ *perfect* for a set $S = \{x_1, \dots, x_n\} \subset [u]$ if $i \neq j$ implies $h(x_i) \neq h(x_j)$
- ▶ *minimal* for set $S = \{x_1, \dots, x_n\} \subset [u]$ if $m = n$

Perfect Hashing

- ▶ only possible for $n \leq m$
- ▶ stringent requirement \rightsquigarrow first focus on **static** $\& \mathcal{S}$
- ▶ further requirements
 1. Hash function must be fast to evaluate (ideally $O(1)$ time)
 2. Hash function must be small to store (ideally $O(1)$ words of space) plus, function allow: $O(1)$ words
 3. Have small m (ideally $m = \Theta(n)$)
 4. should be fast to compute given S (ideally $O(n)$ time)

Fredman-Komlós-Szemerédi Scheme

Theorem 5.4 (FKS Static Perfect Hashing)

Given a c -universal family of hash functions \mathcal{H} and a static set S , we can construct in $O(n)$ expected time a perfect hash function $h : S \rightarrow [m]$ with $m = O(n)$ and evaluation time $O(t)$ for t the time to evaluate functions in \mathcal{H} . ◀

(with universal families from before, all good properties fulfilled!)

Step 1: Simple, but space inefficient

What # buckets do we need to make random $h \in \mathcal{H}$ perfect with good prob.?
w/p $\geq \frac{1}{2}$

n fixed $m = \# \text{ buckets} = m(n)$

birthday paradox $n \approx \sqrt{2 \ln 2} \approx \sqrt{m}$

Let's try $m = n^2$

$$\mathbb{P}[\hat{X} \geq 2] \leq \mathbb{P}[C \geq 1] = \mathbb{P}\left[C \geq \frac{n^2}{m}\right] = \mathbb{P}\left[C \geq 2 \mid \mathbb{E}[C]\right] \leq \frac{1}{2}$$

\uparrow
not perfect

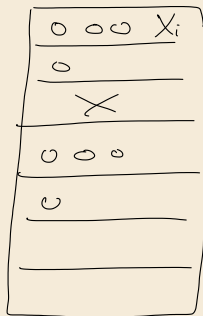
$$\hat{X} \geq 2 \Rightarrow C \geq 1 = \frac{n^2}{m}$$

\Rightarrow w/p $\geq \frac{1}{2}$ h is perfect!

but w/ $m = n^2$ \notin

Step 2: Two-tier solution

pretend $c=1$ universal



bucket i



$h_2^{(i)}$

choose these with $m_i = X_i^2$
after $O(1)$ tries, each $h_2^{(i)}$ perfect

$$s = 2n$$

$$h_2: \mathcal{U} \rightarrow [0..m)$$

$$h_2^{(k)}(x)$$

overall hash function

evaluation time $2 \cdot t + O(1)$

$$\underline{\text{Space}} \quad \# \text{ buckets} = m = \sum_{i=1}^s m_i = \sum_{i=1}^s X_i^2$$

(total space $O(n+m)$)

$$\begin{aligned}
 \mathbb{E}[m] &= \sum_{i=1}^s \mathbb{E}[X_i^2] = \sum_{i=1}^s \mathbb{E}[X_i(X_i-1)] + \underbrace{\sum_{i=1}^s \mathbb{E}[X_i]}_{=n} \\
 &\quad X^2 = X(X-1) + X \\
 &= n + 2 \underbrace{\sum_{i=1}^s \mathbb{E}\left[\binom{X_i}{2}\right]}_{= \mathbb{E}[C]} = n + 2 \frac{n^2}{2s} = n + \frac{1}{2}n = O(n) \\
 &= \mathbb{E}[C] = \frac{n^2}{2s}
 \end{aligned}$$

After $O(1)$ expected tries, find h_1 with $m \leq 3 \cdot n$

For each bucket, $O(1)$ expected tries finds perfect $h_2^{(i)}$ with $m_i = X_i^2$ slots

5.3 Dynamic Perfect Hashing

Dietzfelbinger et al. Scheme

Theorem 5.5 (Dynamic Perfect Hashing)

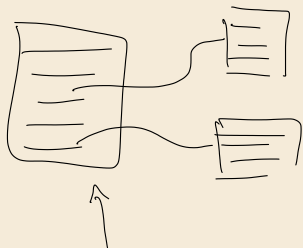
Dynamic perfect hashing stores a subset $S \subset [0..2^w)$ of n integers in a randomized data structure supporting $O(1)$ worst-case lookup, $O(1)$ expected amortized time insert and delete and uses $O(n)$ words of space. ◀



Dietzfelbinger, Karlin, Mehlhorn, Meyer auf der Heide, Rohnert, Tarjan:
Dynamic Perfect Hashing: Upper and Lower Bounds, SICOMP 1994

Note: This assumes that w is the word size, so that there are universal hash families with $O(1)$ evaluation time for the integers we store.

Conceptually: FKS but w/ doubling arrays throughout.



top-level hash table $\stackrel{8}{15} \sqrt{30} M$

has s slots $s = s(M) = \Theta(M)$

$c \in (0, 1)$ think $c = \frac{1}{2}$

target upper bound for # elements $M = (1+c) \cdot n$ initially

$$\boxed{\text{Invariant } M \leq \frac{1+c}{1-c} \cdot n}$$

$$\frac{3/2}{3/4} = 2n$$

$h \in \mathcal{H}_s$ universal hash function

$$h: U \rightarrow [0..s)$$

secondary HT

$$W_j := \{x \in S \cdot h(x) = j\} \quad b_j = |W_j|$$


use #slots $s_j := 2 \cdot m_j (m_j - 1)$

m_j = bound for b_j (since last rebuild)

$$h_j \in \mathcal{H}_{s_j}$$

$$(*) \quad \text{Invariant } \sum_{j=0}^{s(M)-1} s_j \leq \frac{32 M^2}{s(M)} + 4M = \Theta(M)$$

↑
if (*) violated \Rightarrow global rebuild
also, after cn updates ↑

deletion: use tombstones, mark as deleted, but leave in table 
overwrite upon insert
remove (only) at rebuilding HT

local rebuilding triggered upon collision
 $O(1)$ tries to get new & perfect h_j

INSERT: find slot for x

- global rebuild if needed (by count)
- local rehash if collision

if new local size violates (*) \rightarrow global rebuild
otherwise local rehash

o insert x

```
procedure Insert( $x$ );  
   $count \leftarrow count + 1$ ;  
  if  $count > M$   
  then  
    RehashAll( $x$ );  
  else  
     $j \leftarrow h(x)$ ;  
    if position  $h_j(x)$  of subtable  $T_j$  contains  $x$   
    then  
      if  $x$  is marked "deleted" then remove this tag;  
    else (*  $x$  is new for  $W_j$  *)  
       $b_j \leftarrow b_j + 1$ ;  
      if  $b_j \leq m_j$   
      then (* size of  $T_j$  sufficient *)  
        if position  $h_j(x)$  of  $T_j$  is empty  
        then  
          store  $x$  in position  $h_j(x)$  of  $T_j$ ;  
        else  
          go through the subtable  $T_j$ , put all elements  
          not marked "deleted" into a list  $L_j$ , and  
          mark all positions of  $T_j$  empty;  
          append  $x$  to list  $L_j$ ;  $b_j \leftarrow$  length of  $L_j$ ;  
          repeat  $h_j \leftarrow$  randomly chosen function in  $\mathcal{H}_{s_j}$   
          until  $h_j$  is injective on the elements of list  $L_j$ ;  
          for all  $y$  on list  $L_j$  store  $y$  in position  $h_j(y)$  of  $T_j$ ;  
      else (*  $T_j$  is too small *)  
         $m_j \leftarrow 2 \cdot \max\{1, m_j\}$ ;  $s_j \leftarrow 2m_j(m_j - 1)$ ;  
        if condition (2.3) is still satisfied  
        then (* double capacity of  $T_j$  *)  
          allocate new space, namely  $s_j$  cells, for new subtable  $T_j$ ;  
          go through old subtable  $T_j$ , put all elements  
          not marked "deleted" into a list  $L_j$ ,  
          and mark all positions empty;  
          append  $x$  to list  $L_j$ ;  $b_j \leftarrow$  length of  $L_j$ ;  
          repeat  $h_j \leftarrow$  randomly chosen function in  $\mathcal{H}_{s_j}$   
          until  $h_j$  is injective on the elements of list  $L_j$ ;  
          for all  $y$  on list  $L_j$  store  $y$  in position  $h_j(y)$  of  $T_j$ ;  
      else (* level-1-function  $h$  "bad" *)  
        RehashAll( $x$ );
```

```
procedure RehashAll( $x$ );  
  (* RehashAll is either called by Insert with a parameter  $x \in U$ ,  
  or by Delete or Initialize without parameters. RehashAll builds a  
  new table for all elements currently in the table (plus  $x$ , if given). *)  
  go through the whole table  $T$ , put all elements not tagged "deleted"  
  into a list  $L$ , count them, and mark all positions in  $T$  "empty";  
  if  $x \in U$  then append  $x$  to  $L$ ;  
   $count \leftarrow$  length of list  $L$ ;  
   $M \leftarrow (1 + c) \cdot \max\{count, 4\}$ ;  
  repeat  $h \leftarrow$  randomly chosen function in  $\mathcal{H}_{s(M)}$ ;  
  for all  $j, 0 \leq j < s(M)$ , do form a list  $L_j$  of all  $x \in L$  with  $h(x) = j$ ;  
  for all  $j, 0 \leq j < s(M)$ , do  
     $b_j \leftarrow$  length of list  $L_j$ ;  $m_j \leftarrow 2 \cdot b_j$ ;  $s_j \leftarrow 2m_j(m_j - 1)$ ;  
  until condition (2.3) is satisfied;  
  for all  $j, 0 \leq j < s(M)$ , do  
    allocate space  $s_j$  for subtable  $T_j$ ;  
    repeat  $h_j \leftarrow$  randomly chosen function in  $\mathcal{H}_{s_j}$   
    until  $h_j$  is injective on the elements of list  $L_j$ ;  
    for all  $x$  on list  $L_j$  do store  $x$  in position  $h_j(x)$  of  $T_j$ ;  
  
procedure Delete( $x$ );  
   $count \leftarrow count + 1$ ;  
   $j \leftarrow h(x)$ ;  
  if position  $h_j(x)$  of subtable  $T_j$  contains  $x$   
  then mark  $x$  as "deleted"  
  else return( $x$  is not a member of  $S$ );  
  if  $count \geq M$   
  then (* start new phase *)  
    RehashAll();  
  
procedure Lookup( $x$ );  
   $j \leftarrow h(x)$ ;  
  if position  $h_j(x)$  of subtable  $T_j$  contains  $x$  (not marked "deleted")  
  then return(" $x$  is a member of  $S$ ")  
  else return(" $x$  is not a member of  $S$ ");  
  
procedure Initialize;  
   $T \leftarrow$  an empty table;  
  RehashAll();
```

Analysis sketch

Lemma 1: Expected time before global rehash $\Theta(n)$

Lemma 2: Entire phase starting w/ global rehash w/ n elements takes $O(n)$ time total.

Local rehash: since all tables built so that twice the current # elements would have $P\{\text{no collisions}\} \geq \frac{1}{2}$
 \Rightarrow hash tables last until capacity exceeded (forced rebuild)
 \Rightarrow E cost $O(1)$ amortized per update

Global rehash: similar calculation as for C shows that (*) holds for cn updates w/ $p \geq \frac{1}{2}$

□.

Dynamic Perfect Hashing – Discussion



Strongest possible guarantee for lookup



All operations in $O(1)$ is clearly optimal (even though *expected amortized*)

↪ (almost) perfect dictionary?

Dynamic Perfect Hashing – Discussion



Strongest possible guarantee for lookup



All operations in $O(1)$ is clearly optimal (even though *expected amortized*)

↪ (almost) perfect dictionary?



Main downside (of hashing generally): No sorted-dictionary operations

5.4 Binary Tries

Recall: Ordered symbol tables

On top of updates and lookup, we have

- ▶ $\text{min}()$, $\text{max}()$
Return the smallest resp. largest key in the ST
- ▶ $\text{floor}(x)$ (a.k.a. $\text{predecessor}(x)$) $\lfloor x \rfloor = \mathbb{Z}.\text{floor}(x)$
Return largest key k in ST with $k \leq x$.
- ▶ $\text{ceiling}(x)$ (a.k.a. $\text{successor}(x)$)
Return smallest key k in ST with $k \geq x$.
- ▶ $\text{rank}(x)$
Return the number of keys k in ST $k < x$.
- ▶ $\text{select}(i)$
Return the i th smallest key in ST (zero-based, i. e., $i \in [0..n)$)

Sorted Dictionaries of Integers

As we will see at the end of this section, not much can be done for sorted integer dictionaries!

Fredman-Saks bound (informal version):

Maintaining a dynamic set of integers with rank or select queries requires $\Omega(\lg n / \lg \lg n)$ time per operation.

\rightsquigarrow Up to the $\lg \lg n$ factor, BSTs are best possible even for integers

Sorted Dictionaries of Integers

As we will see at the end of this section, not much can be done for sorted integer dictionaries!

Fredman-Saks bound (informal version):

Maintaining a dynamic set of integers with rank or select queries requires $\Omega(\lg n / \lg \lg n)$ time per operation.

↪ Up to the $\lg \lg n$ factor, BSTs are best possible even for integers

Surprisingly, the story shifts without rank and select!

- ▶ Consider here updates and floor a.k.a. predecessor
 - ▶ Adding ceiling a.k.a. successor trivial (by symmetry)

Sorted Dictionaries of Integers

As we will see at the end of this section, not much can be done for sorted integer dictionaries!

Fredman-Saks bound (informal version):

Maintaining a dynamic set of integers with rank or select queries requires $\Omega(\lg n / \lg \lg n)$ time per operation.

↪ Up to the $\lg \lg n$ factor, BSTs are best possible even for integers

Surprisingly, the story shifts without rank and select!

▶ Consider here updates and floor a.k.a. predecessor

▶ Adding ceiling a.k.a. successor trivial (by symmetry)

▶ As we will see, adding min / max trivial for considered data structures

↪ Our following ideas also work as priority queues for integers!

▶ decreaseKey possible via $\text{delete}(x)$ and $\text{insert}(x')$

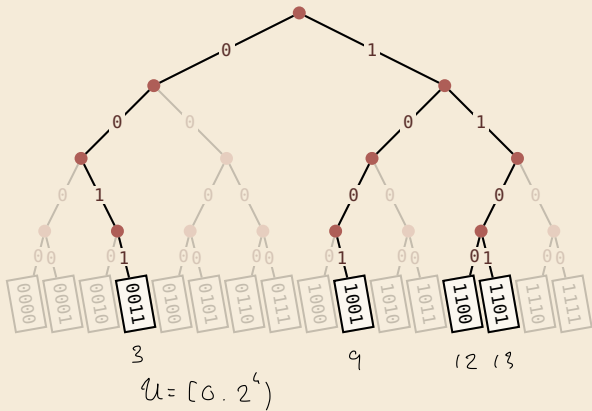
Tries

Integers are nothing but bit sequences \rightsquigarrow *can use string data structures such as **tries***

Tries

Integers are nothing but bit sequences \rightsquigarrow can use string data structures such as *tries*

\rightsquigarrow use a binary trie



Tries

Integers are nothing but bit sequences \rightsquigarrow can use string data structures such as *tries*

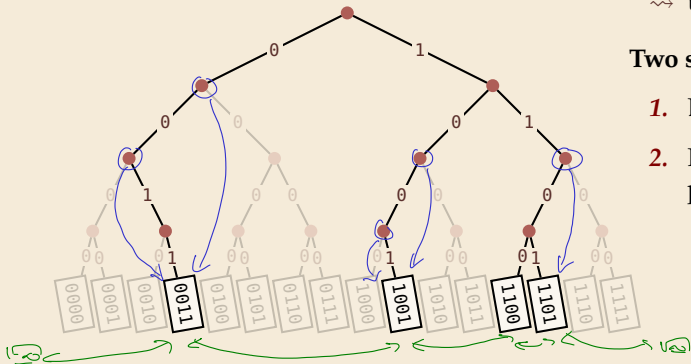
\rightsquigarrow use a binary trie

Two simple additions

1. Leaves doubly linked in sorted order

2. Each nonbinary node stores jump pointer to extremal leaf in subtree

- ▶ if node is missing its left child, *jump* is leftmost leaf in subtree
- ▶ if node is missing right child, *jump* is rightmost leaf in subtree



Binary Trie – Result

Theorem 5.6

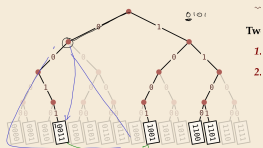
A binary trie can store a subset $S \subseteq [0..2^w)$ of n numbers supports updates and predecessor / successor queries all in $O(w)$ time per operation for a set of w -bit integers, using $O(nw)$ bits of space.

1
height of trie

$$n \leq 2^w$$

$$l_{S,n} \leq w$$

(worse than BST)



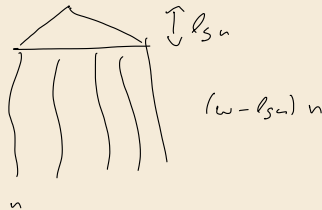
Predecessor(x)

① Find longest common prefix between x and any key $y \in S$

$O(w)$

② Follow one jump pointer and maybe one predecessor/successor pointer.

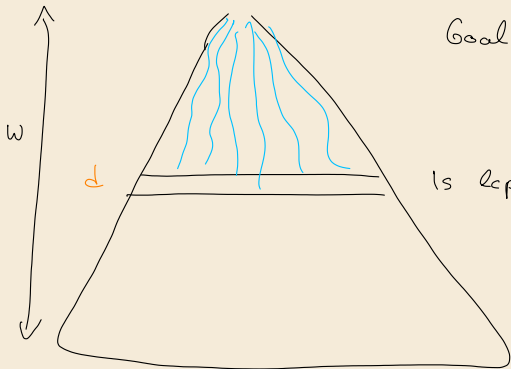
$O(1)$



5.5 x-Fast Tries

Speeding up Lookup

① Find lcp between x and S



Goal: Binary search over levels

↓ need to answer these questions.

Is $lcp \geq d$?

For each level d store set of prefixes
on root to node at depth d paths
 $\leq n$ d -bit integers

store these using dynamic perfect hashing!

\Rightarrow can find lcp in $O(\lg w)$ time!

② same $\Rightarrow O(\lg w)$ predecessor search

x-Fast Tries

Willard

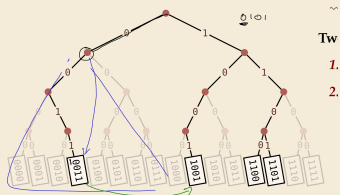
Theorem 5.7

An x-fast trie can store a subset $S \subseteq [0..2^w)$ of n integers in a randomized data structure supporting predecessor in $O(\lg w)$ worst case time and updates in $O(w)$ amortized expected time, using $O(nw)$ words of space.

updates: add up to w nodes
add prefix to up to
 w dictionaries

$\Theta(w)$ time

space: need to store w dictionaries w/ up to n integers each

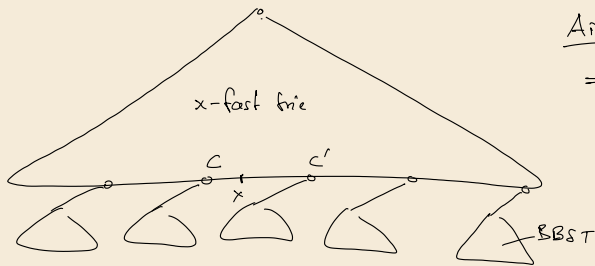


5.6 y-Fast Tries

"Indirection" a.k.a. Don't Change Too Fast!

Idea: cluster keys into locally searchable groups

use only representatives in previous (slow-update) data structure



Aim keep pred. search in $\Theta(\lg w)$
 $\Rightarrow \Theta(w)$ key per cluster
allow local search in

Pred(x): find cluster pred. C

\Rightarrow search for pred in C's BST and
C's successors (C')s BST

Space: clusters $O(u)$

x-fast trie $n' = \Theta(\frac{u}{w})$
 \Rightarrow space $O(n'w) = O(u)$

Need to select representatives (pivots) for clusters

Option 1: deterministic, split at $\geq 2w$ keys

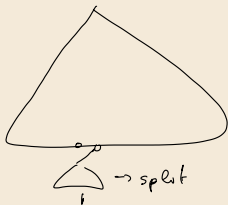
$a_i < \frac{1}{2}w$, merge w/ neighbor cluster, re-split

$\Rightarrow O(1)$ amortized

Option 2: every element chosen w/p $\frac{1}{w}$ as pivot (upon insert)

Represent clusters as treaps

Insert: Flap biased coin, w/p $\frac{1}{w}$ insert element into x-fast trie $O(w)$ time



split cluster

otherwise, only insert in correct cluster (found via successor)

Deletion, remove from cluster

if it was a representative, replace by predecessor from cluster $O(w)$ time

$\hookrightarrow w/p \frac{1}{w}$

Updates cost $O(\lg w) + O(1)$ expected cost for x-fast trie updates

y-Fast Tries

Theorem 5.8 (y-fast tries)

A y-fast trie can store a subset $S \subseteq [0..2^w)$ of n integers in a randomized data structure supporting predecessor in $O(\lg w)$ ~~worst case~~ time and updates in $O(\lg w)$ amortized expected time, using $O(n)$ words of space.

expected



Summary

Typical assumption: word size $w = \Theta(\log n)$.

Summary

Typical assumption: word size $w = O(\log n)$.

Consequences: (under above assumption)

- ▶ Can maintain $S \subseteq [0..2^w)$ with the following operations in $O(\log \log n)$ amortized expected time
 - ▶ insert, delete, lookup
 - ▶ predecessor, successor
 - ▶ min, max
 - ▶ decreaseKey, increaseKey

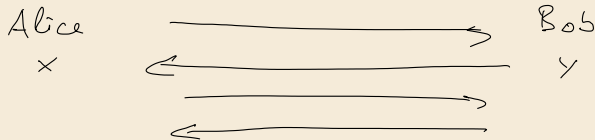
5.7 Lower Bounds

The Cell-Probe Model

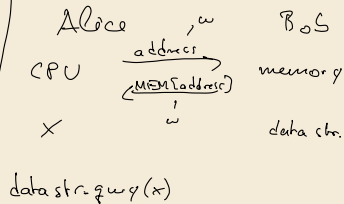
- ▶ model of computation mostly useful for lower bounds
- ▶ memory arranged in words of size w
- ▶ **only** cost in model is number of words of memory “touched” (read or written) (all computation for free!)

⇒ clearly, any word-RAM algorithm requires at least as much time as cell probes

Impossibility results from communication complexity



$f(x, y)$



Two Classical Cell-Probe Bounds [1]

Theorem 5.9 (Fredman-Saks Dynamic Rank Bound)

Maintaining a **dynamic** set $S \subseteq [0..u)$ of n integers in the cell-probe model with word size $w \leq \lg^c(u)$ (for constant c) and either operations

- (a) insert, delete, and **rank**; or
- (b) insert, delete, and **select**

requires **amortized** $\Omega(\log u / \log \log u)$ cell probes per operation. ◀



Fredman, Saks: *The Cell Probe Complexity of Dynamic Data Structures*, STOC 1989

Note: since $n \leq u$, lower bound of $\Omega(\log n / \log \log n)$ holds as well.

A special case

General proof of Theorem 5.9 is technical, but let's prove (substantially) weaker version

Theorem 5.10 (Fredman Dynamic Rank Bound for $w = 1$)

Maintaining a **dynamic** set $S \subseteq [0..u)$ of n integers in the cell-probe model with word size $w = 1$ and operations insert, delete, and rank requires $\Omega(\log u / \log \log u)$ cell probes per operation in the worst-case.

Two problems in between

Dynamic Prefix Parity

maintain bitvector $A[0..u)$ initially all 0

Update: $A[i] := a$

$$\text{Rank-Parity}(i) \quad A[0] \oplus A[1] \oplus \dots \oplus A[i] = \sum_{j=0}^i A[j] \pmod 2$$

To reduce to dynamic rank: maintain set $S = \{i : A[i] = 1\}$

to answer rank-parity compute $\text{rank}(i) \pmod 2$

Show dynamic prefix parity needs $\Omega(\log u / \log \log u)$ bit probes (per op. w.c.)

Consider yet another problem

Which-side problem maintain value $x \in [0..u)$

Set(y) $x := y$

Ask(y) is $y < x$?

obvious upper bound : $O(\lg u)$ bit probes Set store y in binary
Ask read y and compare

Lemma: Any which-side algorithm needs $\Omega(\lg u / \lg \lg u)$ bit probes w.c. per op.

Proof: seq. SET(i) followed only by Ask queries

Given solution with space s bits and operation t (w.c. per op.)

$x_i :=$ state of memory $x_i \in \{0,1\}^s$

Hamming weight of x_i : $\|x_i\|_1 \leq t$ (only t cells touched)

Use lemma on decision tree size memory formed from $\lg u$ bits
 (binary search) together $\Rightarrow d \leq \lg u \cdot t$

$$\Rightarrow u \leq \sum_{i=0}^t \binom{\lg u \cdot t}{i} \quad \text{and} \quad t = \Omega\left(\frac{\lg u}{\lg \lg u}\right)$$

Trick $\sum_{i=0}^k \binom{n}{i} \leq \left(\frac{en}{k}\right)^k$

"
 # subsets of $[n]$
 of size $\leq k$

$$\leq \left(\frac{e t \lg u}{t}\right)^t$$

$$\lg u \leq t \left(\lg e + \lg t + \frac{\lg \lg u}{t} - \lg t \right)$$

$$t \geq \frac{\lg u}{\lg \lg u + \lg e} = \Omega\left(\frac{\lg u}{\lg \lg u}\right)$$

From binomial theorem $(1+x)^n = \sum_{i=0}^n \binom{n}{i} x^i 1^{n-i}$

$$\boxed{x \leq 1} \quad \geq \sum_{i=0}^k \binom{n}{i} x^i = x^k \sum_{i=0}^k \binom{n}{i} \quad \boxed{x := \frac{k}{n}}$$

$$\sum_{i=0}^k \binom{n}{i} \leq \left(1 + \frac{k}{n}\right)^n \cdot \left(\frac{k}{n}\right)^{-k} \leq e^k \left(\frac{n}{k}\right)^k$$

□

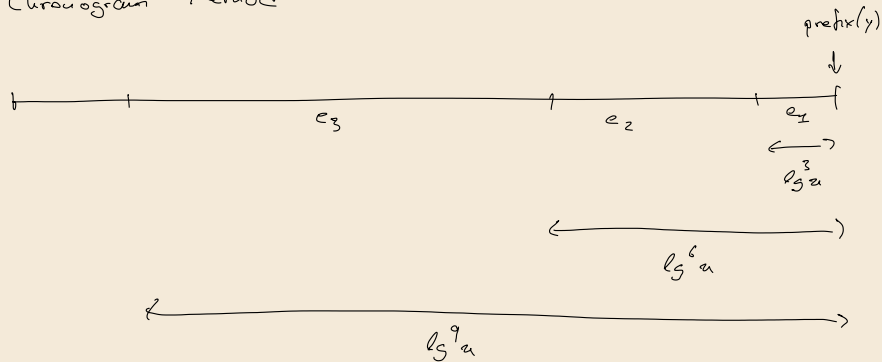
Remains to show: dynamic prefix parity can simulate which-side

SET(i) $A[i] := 1$

ASK(j) we output $A[0] \oplus \dots \oplus A[j]$

How to proof original Fredman-Saks $w = \lg u$

Chronogram Method



\Rightarrow # epochs r s.t. $\lg^{3r}(u) = \sqrt{u} = k$

$$r = \frac{\lg u}{6 \lg \lg u}$$

Show that query needs to read a word written in each epoch



Two Classical Cell-Probe Bounds [2]

Theorem 5.11 (Predecessor Lower Bound)

Given a static set of $S \subseteq [0..u)$ of n integers with $\ell = \lg u$ bits each, represented in a data structure using $S \geq n\ell$ bits of space. Set $a = \lg(S/n)$. Answering predecessor (a.k.a. floor) queries costs

#bits per number

$$\Omega \left(\min \left\{ \log_w(n), \lg \left(\frac{\ell - \lg n}{a} \right), \frac{\lg(\frac{\ell}{a})}{\lg(\frac{a}{\lg n} \cdot \lg(\frac{\ell}{a}))}, \frac{\lg(\frac{\ell}{a})}{\lg(\lg(\frac{\ell}{a}) / \lg(\frac{\lg n}{a}))} \right\} \right)$$

cell probes on a word-RAM with word size w . ◀



Pătraşcu, Thorup: *Time-Space Trade-Offs for Predecessor Search*, STOC 2006

Two Classical Cell-Probe Bounds [2]

Theorem 5.11 (Predecessor Lower Bound)

Given a **static** set of $S \subseteq [0..u)$ of n integers with $\ell = \lg u$ bits each, represented in a data structure using $S \geq n\ell$ bits of space. Set $a = \lg(S/n)$. Answering predecessor (a.k.a. floor) queries costs

$$\Omega \left(\min \left\{ \log_w(n), \lg \left(\frac{\ell - \lg n}{a} \right), \frac{\lg(\frac{\ell}{a})}{\lg(\frac{a}{\lg n} \cdot \lg(\frac{\ell}{a}))}, \frac{\lg(\frac{\ell}{a})}{\lg(\lg(\frac{\ell}{a}) / \lg(\frac{\lg n}{a}))} \right\} \right)$$

cell probes on a word-RAM with word size w . ◀

Remarks

 Pătrașcu, Thorup: *Time-Space Trade-Offs for Predecessor Search*, STOC 2006

▶ Typical parameters:

▶ space usage $S = n2^a$ is $\Theta(n)$ **words** of space $\rightsquigarrow a = \lg \lg n$ $a = \lg w$

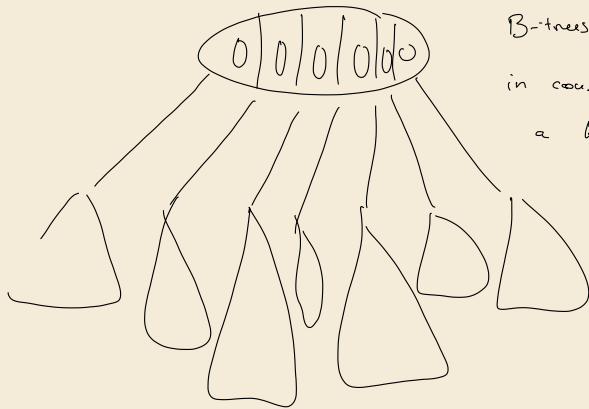
▶ $\ell \approx w = \lg u$

\rightsquigarrow **fusion trees** optimal for huge w , i. e., when $\lg w = \Omega(\sqrt{\lg n} \lg \lg n)$

\rightsquigarrow **y-fast tries** optimal for moderate w , i. e., any $w = O(\text{polylog } n)$

▶ last two terms less intuitive ...

Fusion Trees – Overview Only



B-trees with fanout $\sqrt[w]{w}$.

in constant time, can compare
a key with $\sqrt[w]{w}$ keys!

$$\begin{aligned} \text{height } \log_{\sqrt[w]{w}}(n) &= \frac{\log n}{\log(\sqrt[w]{w})} \\ &= \Theta(\log_w(n)) \end{aligned}$$

☞ exam

Integer Dictionaries

Upper Bounds:

- ▶ **Unsorted** dictionary: dynamic perfect hashing does it!
- ▶ **Sorted** dictionary: plain old BBST almost optimal (gap not worth the hassle in practice)
fusion tree closes gap with $O\left(\frac{\lg n}{\lg \lg n}\right)$ operations
- ▶ **Predecessor only / priority queue**: y-fast tries give $O(\lg \lg n)$ operations
seems to require randomization

Integer Dictionaries

Upper Bounds:

- ▶ **Unsorted** dictionary: dynamic perfect hashing does it!
- ▶ **Sorted** dictionary: plain old BBST almost optimal (gap not worth the hassle in practice)
fusion tree closes gap with $O\left(\frac{\lg n}{\lg \lg n}\right)$ operations
- ▶ **Predecessor** only / **priority queue**: y-fast tries give $O(\lg \lg n)$ operations
seems to require randomization

Lower Bounds:

- ▶ complicated results (skipped proofs), many-parameter trade-offs
- ▶ largely matching upper and lower bounds for predecessor and dynamic rank
- ▶ predecessor bound applies even to static case; needs to take space into account
- ▶ dynamic rank/select lower bound holds irrespective of space usage, exponentially higher