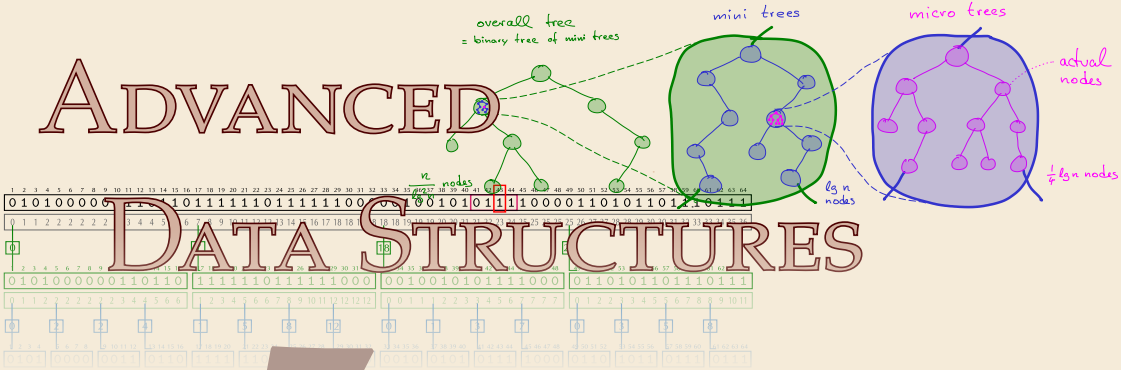


Advanced Data Structures

5 Integer Data Structures

Advanced Data Structures · Summer 2026

Prof. Dr. Sebastian Wild

CS650 (Summer 2026)
Philipps-Universität Marburg
version **2026-06-16 10:06**

Outline

5 Integer Data Structures

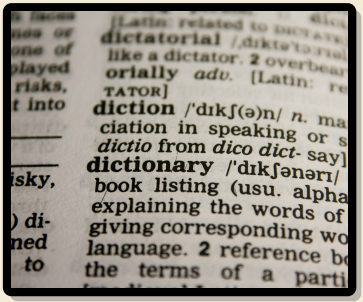
5.1 Integer Dictionaries

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}`



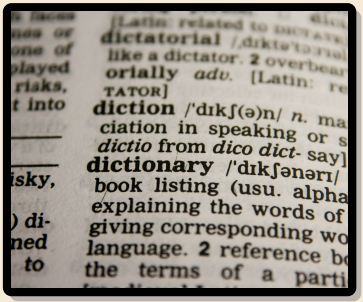
- `put(`k`,`v`)` *Python* dict: `d[`k`] = `v
  Put key-value pair (k, v) into table
- `get(`k`)` *Python* dict: `d[`k`]`
  Return value associated with key k
- `delete(`k`)` *Python* dict: `del d[`k`]`
  Remove key k (any associated value) form table
- `contains(`k`)` *Python* dict: k `in d`
  Returns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

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) form 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, $S = \{x_1, \ldots, x_n\}$

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, $S = \{x_1, \ldots, x_n\}$
- use **hash table** $T[0..m)$ with m **buckets**
- **hash function** $h : U \to [0..m)$

$\rightsquigarrow$ 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, $S = \{x_1, \ldots, x_n\}$
- use **hash table** $T[0..m)$ with m **buckets**
- **hash function** $h : U \to [0..m)$

$\rightsquigarrow$ element $x \in U$ stored in $T[h(x)]$

- in general: $T[h(x)]$ may be a **secondary data structure**
  - linked list $\rightsquigarrow$ *chaining hashing*
  - sequence of positions in T $\rightsquigarrow$ *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, $S = \{x_1, \ldots, x_n\}$
- use **hash table** $T[0..m)$ with m **buckets**
- **hash function** $h : U \to [0..m)$

$\rightsquigarrow$ element $x \in U$ stored in $T[h(x)]$

- in general: $T[h(x)]$ may be a **secondary data structure**
  - linked list $\rightsquigarrow$ *chaining hashing*
  - sequence of positions in T $\rightsquigarrow$ *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
  - $\rightsquigarrow$ $m = O(n)$
- typically thus $|U| = 2^w \gg m$
  - $\rightsquigarrow$ h must have many collisions

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

$\rightsquigarrow$ cannot have meaningful worst-case time bounds (even if typical case good)

365 ≈ 25 people to get 50% collision for birthdays

$n \approx 2\sqrt{m}$

Hashing must be randomized

Collisions are inevitable

- usually want S represented with $O(n)$ space
  - $\rightsquigarrow$ $m = O(n)$
- typically thus $|U| = 2^w \gg m$
  - $\rightsquigarrow$ h must have many collisions

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

$\rightsquigarrow$ cannot have meaningful worst-case time bounds (even if typical case good)

Randomized Hashing

- draw *random* $h \in \mathcal{H}$
  - $\rightsquigarrow$ running time expected over random choice, worst-case w.r.t. S
- allowing all $\mathcal{H} = \{h \mid h : U \to [0..m)\}$ requires ~~m^u~~ space $\log(m^u) = u \cdot \log m$
- $\rightsquigarrow$ 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.

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

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

for fully random h

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

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

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

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

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

◀

Examples of universal families

Universal families

- $\mathcal{H}_1 = \{h_{ab} : a \in [1..p), b \in [0..p)\}$ $(c = 2)$
- $\mathcal{H}_0 = \{h_{ab} : a \in [0..p), b \in [0..p)\}$ $(c = 4)$
- $\mathcal{H}_2 = \{h_a : a \in [1..2^k), a \text{ odd}\}$ $(c = 2)$

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

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

What does universality buy us?

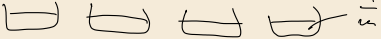
Lemma 5.2 (Universal collisions)

Consider hashing n keys $x_1, \ldots, x_n$ 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., $C = \left|\{(i,j) : 1 \le i < j \le n \wedge h(x_i) = h(x_j)\}\right|$.

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

$C \le \binom{n}{2}$ $\quad C \ge \binom{\frac{n}{m}}{2} \cdot m \simeq \frac{n^2}{m^2} \cdot m$

Proof $\mathop{\mathbb{E}}_{h}[C] = \sum_{1 \le i < j \le n} \mathop{\mathbb{E}}_{h}[C_{ij}] \le \binom{n}{2} \frac{c}{m} < \frac{cn^2}{2m}$



$C_{ij} = [h(x_i) = h(x_j)]$ $\quad \mathop{\mathbb{E}}_{h}[C_{ij}] = \mathop{\mathbb{P}}_{h}[h(x_i) = h(x_j)] \le \frac{c}{m}$

What does universality buy us?

Lemma 5.2 (Universal collisions)

Consider hashing n keys $x_1, \ldots, x_n$ 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., $C = \left|\{(i,j) : 1 \le i < j \le n \wedge h(x_i) = h(x_j)\}\right|$.

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

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

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

(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 $\boxed{\mathbb{P}\{X \geq a\,\mathbb{E}[X]\} \leq \frac{1}{a} \quad X \geq 0}$

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

5.2 Perfect Hashing

Perfect Hashing

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

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

Perfect Hashing

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

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

Perfect Hashing

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

Perfect Hashing

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

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

Perfect Hashing

- only possible for $n \leq m$
- stringent requirement $\rightsquigarrow$ first focus on **static** ~~$\mathcal{S}$~~ 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)
  3. Have small m (ideally $m = \Theta(n)$)
  4. should be fast to compute given S (ideally $O(n)$ time)

Fredman-Komlós-Szemeredi 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 $h : S \to [m]$ 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 $\quad m = \#\text{buckets} = m(n)$

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

Let's try $m = n^2$

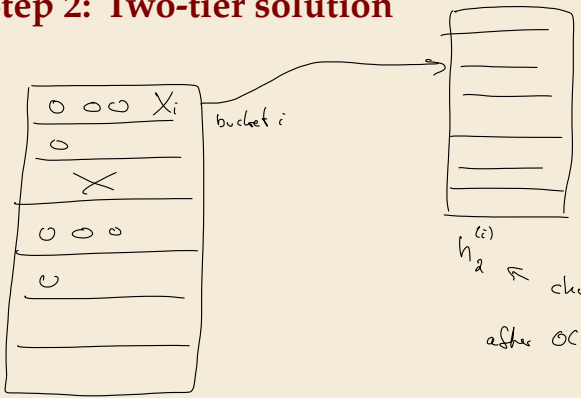
$$\mathbb{P}[\hat{X} \geq 2] \leq \mathbb{P}[C \geq 1] = \mathbb{P}\left[C \geq \frac{n^2}{m}\right] = \mathbb{P}\left[C \geq 2\,\mathbb{E}[C]\right] \leq \frac{1}{2}$$

$\hat{X}$ 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/ $m = n^2$ 😢

Step 2: Two-tier solution



pretend $c = 1$ universal

choose these with $m_i = X_i^2$

after $O(1)$ tries, each $h_2^{(i)}$ perfect

$h_1 : \mathcal{U} \to [0..m)$

$h_2^{(h_1(x))}(x)$ overall hash function

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

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

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

$$\mathbb{E}[m] = \sum_{i=1}^{s} \mathbb{E}[X_i^2] = \sum_{i=1}^{s} \mathbb{E}[X_i(X_i-1)] + \underbrace{\sum_{i=1}^{s} \mathbb{E}[X_i]}_{=n}$$

$x^2 = x(x-1) + x$

$$= n + 2 \underbrace{\sum_{i=1}^{s} \mathbb{E}\binom{X_i}{2}}_{= \mathbb{E}[C] = \frac{n^2}{2s}} = n + \cancel{2} \frac{n^2}{\cancel{2}s} = n + \frac{1}{2}n = O(n)$$

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

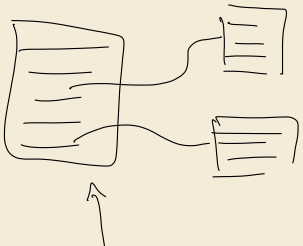
Theorem 5.5 (Dynamic Perfect Hashing)

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 $\quad \frac{8}{15}\lceil 30 \cdot M \rceil$

has s slots $\quad s = s(M) = \Theta(M)$ $\quad\quad$ $c \in (0,1)$ $\quad$ think $c = \frac{1}{2}$

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

$$\boxed{\text{Invariant} \quad M \leq \frac{1+c}{1-c} \cdot n} \qquad \frac{3/2}{3/4} = 2n$$

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

$h : \mathcal{U} \to [0..s)$

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

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

m_j = **bound for** b_j (since last rebuild)

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

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

if (*) violated ⇒ global rebuild

also: after c_N 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 (*) ⇒ global rebuild

  otherwise local rehash

o insert x

```
procedure Insert(x);
  count ← count + 1;
  if count > M
  then
    RehashAll(x);
  else
    j ← 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 ← b_j + 1;
        if b_j ≤ 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 ← length of L_j;
                repeat h_j ← randomly chosen function in 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 ← 2 · max{1, m_j}; s_j ← 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 ← length of L_j;
                repeat h_j ← randomly chosen function in 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 ∈ 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 ∈ U then append x to L;
  count ← length of list L;
  M ← (1 + c) · max{count, 4};
  repeat h ← randomly chosen function in H_{s(M)};
    for all j, 0 ≤ j < s(M), do form a list L_j of all x ∈ L with h(x) = j;
    for all j, 0 ≤ j < s(M), do
      b_j ← length of list L_j; m_j ← 2 · b_j; s_j ← 2m_j(m_j − 1);
  until condition (2.3) is satisfied;
  for all j, 0 ≤ j < s(M), do
    allocate space s_j for subtable T_j;
    repeat h_j ← randomly chosen function in 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 ← count + 1;
  j ← 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 ≥ M
    then (* start new phase *)
      RehashAll();


procedure Lookup(x);
  j ← 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 ← 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 $\mathbb{P}[\text{no collisions}] \geq \frac{1}{2}$

$\Rightarrow$ hash tables last until capacity exceeded (forced rebuild)

$\Rightarrow$ $\mathbb{E}$ cost $O(1)$ amortized per update

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

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

- `min()`, `max()`
  Return the smallest resp. largest key in the ST
- `floor(`x`)` (a.k.a. `predecessor(`x`)`) $\lfloor x \rfloor = \mathbb{Z}.$`floor(`$x$`)`
  Return largest key k in ST with $k \leq x$.
- `ceiling(`x`)` (a.k.a. `successor(`x`)`)
  Return smallest key k in ST with $k \geq x$.
- `rank(`x`)`
  Return the number of keys k in ST $k < x$.
- `select(`i`)`
  Return the ith 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.

$\rightsquigarrow$ 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.

$\rightsquigarrow$ 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

$\rightsquigarrow$ Our following ideas also work as priority queues for integers!

- `decreaseKey` possible via `delete(`x`)` and `insert(`x'`)`

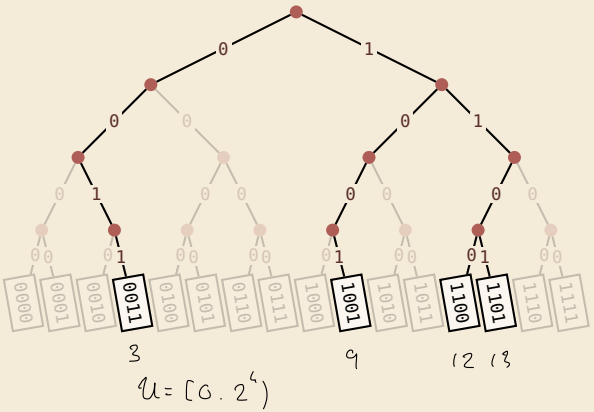
Tries

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

Tries

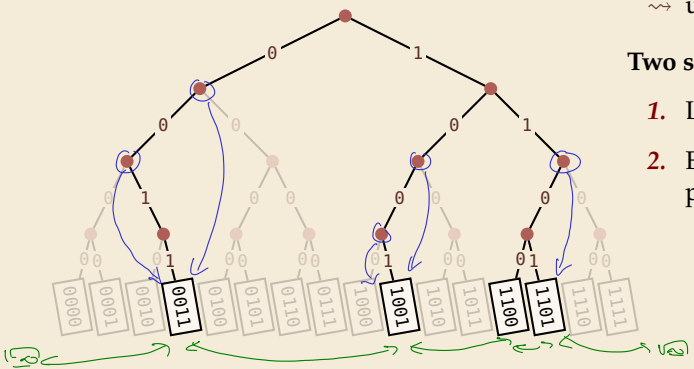
Integers are nothing but bit sequences ⇝ *can use string data structures such as tries*

⇝ use a binary trie



Tries

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



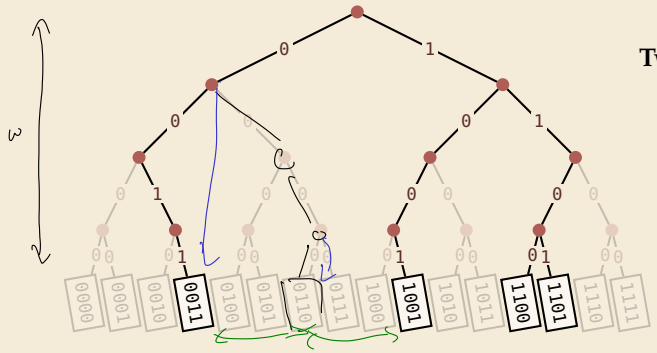
⇝ use a binary trie

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

Tries

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



⇝ use a binary trie

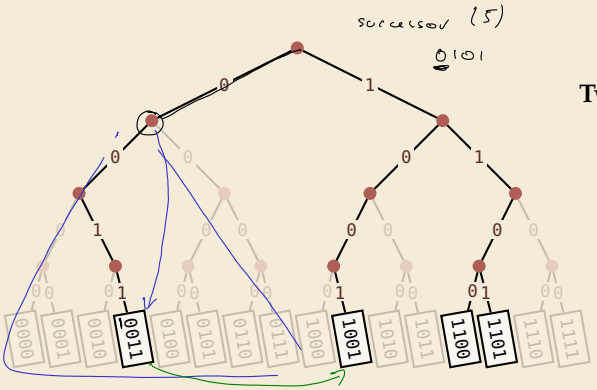
Two simple additions

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

- Additions are easy to maintain upon updates operations $O(w)$ time
- By traversing the trie an following pointers, we can implement successor/predecessor queries

Tries

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



⇝ use a binary trie

Two simple additions

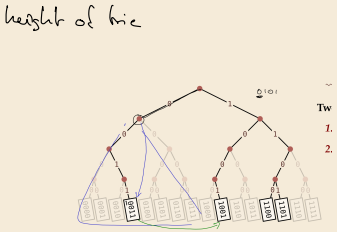
1. Leaves doubly linked in sorted order
2. Each nonbinary node stores *jump* pointer to extremal leaf in subtree
   - if node is missing its left child, *jump* is leftmost leaf in subtree
   - if node is missing right child, *jump* is rightmost leaf

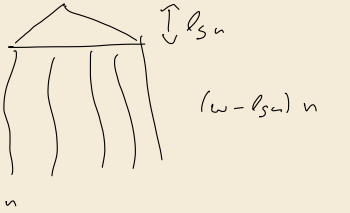
- Additions are easy to maintain upon updates
- By traversing the trie an following pointers, we can implement successor/predecessor queries

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.





$n \leq 2^w$

$\ell g n \leq w$

(worse than BST)

Predecessor(x)

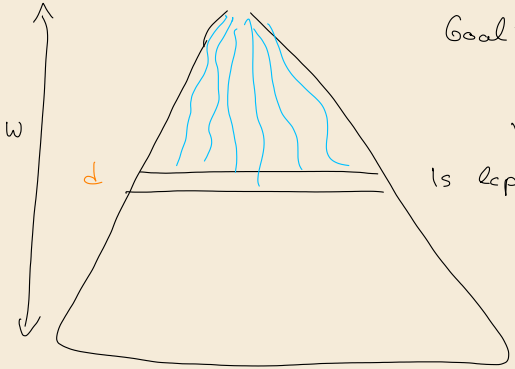
① Find longest common prefix between x and any key $y \in S$ — $O(w)$

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 $O(\log w)$ time!

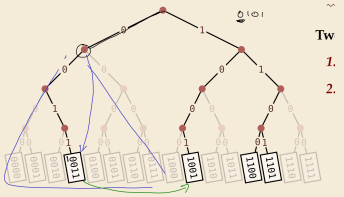
② same $\Rightarrow$ $O(\log w)$ predecessor search

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

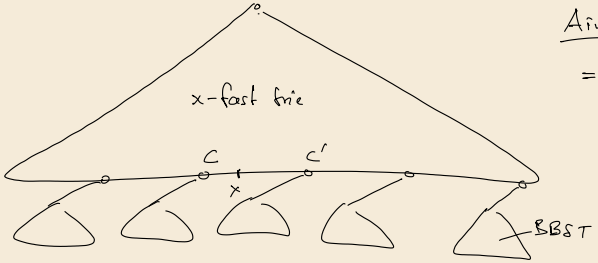
space: need to store w dictionaries w/ up to n integers each

5.6 y-Fast Tries

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

Idea: cluster keys into locally scalable groups

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



Aim: keep pred. search in $\Theta(\log w)$

$\Rightarrow$ $\Theta(w)$ keys 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 $O(n)$

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

$\Rightarrow$ space $O(n'w) = O(n)$

Need to select representatives / pivots for clusters

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

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

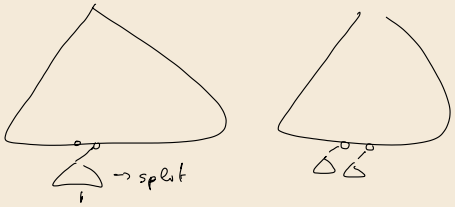
$\Rightarrow O(1)$ amortized

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

Represent clusters as treaps

Insert: Flip biased coin, w/ $\frac{1}{w}$ 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

↳ w/p $\frac{1}{w}$

Updates: cost $O(\log w) + O(1)$ 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 $\boxed{w = \Theta(\log n)}$.

Summary

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

Consequences: (under above assumption)

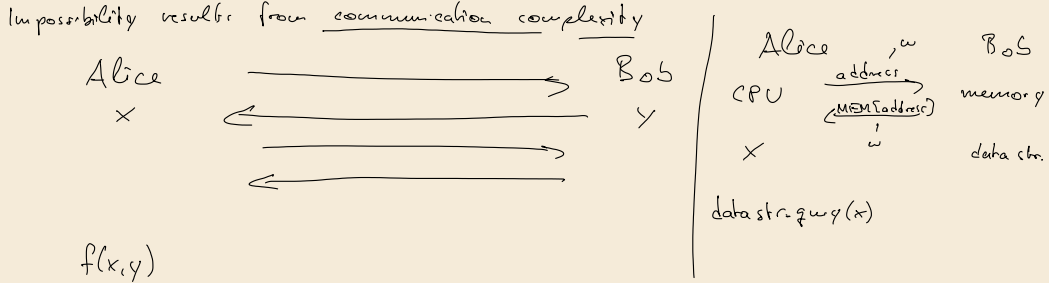
- Can maintain $S \subseteq [0..2^w)$ with the following operations in $O(\log \log n)$ amortized expected time
  - `insert`, `delete`, `lookup`
  - `predecessor`, `successor`
  - `min`, `max`
  - `decreaseKey`, `increaseKey`

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



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 : $A[i] := a$

Rank-Parity(i) $A[0] \oplus A[1] \oplus \cdots \oplus A[i] = \sum_{j=0}^{i} A[j] \bmod 2$

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

to answer rank-parity compute rank(i) mod 2

Show dynamic prefix parity need $\Omega(\lg u / \lg\lg u)$ 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 $y < x$?

obivious upper bound : $O(\lg u)$ bit probes Set store y in binary

Ask read y and compare

Lemma: Any which-side algorithm needs $\Omega(\lg u / \lg \lg_{s/t} a)$ 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 $\|x_i\|_1 \le t$ (only t cells touched)

Use $\log n$ ASK calls to binary search for i

$i \neq j \quad K_i \neq K_j$

$\Rightarrow$ lower bound for t

Lemma: Given bit $x_1, \dots, x_n \in \{0,1\}^s \quad \|x_i\|_1 \leq t$

Suppose a binary decision tree of depth d distinguishes them.

Then $n \leq \sum_{i=0}^{t} \binom{d}{i}$

$\rightsquigarrow \left| \{ x \in \{0,1\}^d : \|x\|_1 \leq t \} \right|$