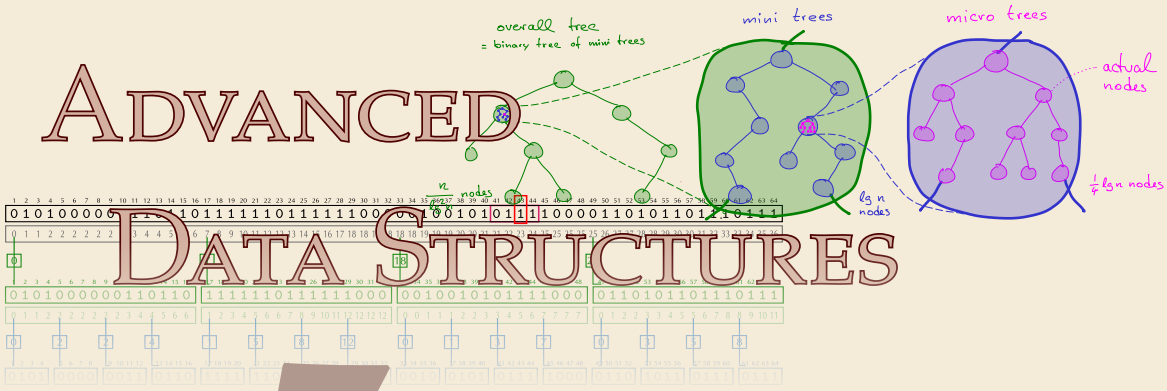


Advanced Data Structures

5 Integer Data Structures

Advanced Data Structures · Summer 2026

Prof. Dr. Sebastian Wild

CS650 (Summer 2026)
Philipps-Universität Marburg
version 2026-06-15 01:34 H

Outline

5 Integer Data Structures

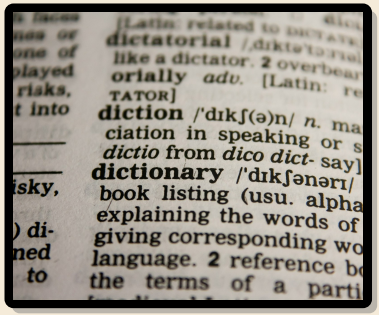
5.1 Integer Dictionaries

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()`

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

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 ↯

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

(b) $\mathcal{H}$ is called *strongly universal* or *pairwise independent* if
$$\forall x_1, x_2 \in U, y_1, y_2 \in R \;:\; x_1 \neq x_2 \implies \mathbb{P}\big[h(x_1) = y_1 \wedge h(x_2) = y_2\big] \leq \frac{1}{m^2}.$$ ◀

Examples of universal families

Universal families

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

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

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

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

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

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 $h(x_i) \neq h(x_j)$
- *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** $\mathcal{X}$
- further requirements
  1. Hash function must be fast to evaluate (ideally $O(1)$ time)
  2. Hash function must be small to store (ideally $O(n)$ space)
  3. should be fast to compute given $\mathcal{X}$ (ideally $O(n)$ time)
  4. Have small m (ideally $m = \Theta(n)$)

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 $\mathcal{H}$. ◀

Step 1: Simple, but space inefficient

Step 2: Two-tier solution

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.*

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

Surprisingly, the story shifts without `rank` and `select`!

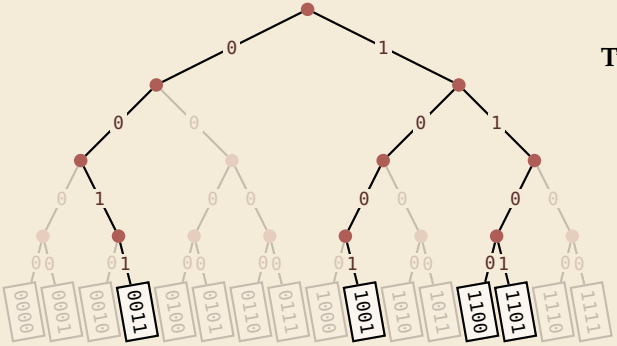
- Consider here updates and `floor` a.k.a. `predecessor`
  - Adding `ceiling` a.k.a. `successor` trivial (by symmetry)
- As we will see, adding `min` / `max` trivial for considered data structures

$\rightsquigarrow$ 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*



$\rightsquigarrow$ use a binary trie

Two simple additions

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

5.5 x-Fast Tries

Speeding up Lookup

x-Fast Tries

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. ◀

5.6 y-Fast Tries

“Indirection” a.k.a. Don’t Change Too Fast!

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 $O(nw)$ words of space. ◀

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!)

$\rightsquigarrow$ clearly, any word-RAM algorithm requires at least that much time

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`**; or

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 $n \leq u$, the bound holds with n instead of u, as well.

Two Classical Cell-Probe Bounds [2]

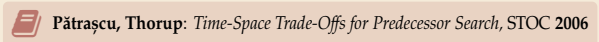
Theorem 5.10 (Predecessor Lower Bound)

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

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

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

Remarks



- Typical parameters:
  - space usage $S = n2^a$ is $\Theta(n)$ **words** of space $\rightsquigarrow$ $a = \lg\lg n$
  - $\ell \approx w = \lg u$

$\rightsquigarrow$ fusion trees optimal for huge w ($\lg w = \Omega(\sqrt{\lg n}\lg\lg n)$)

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

- last two terms less intuitive . . .

Fusion Trees – Overview Only

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