Fortgeschrittene Datenstrukturen — Vorlesung 3

Florian Merz

February 2, 2012

1 Hashing

1.1 Perfect Hashing

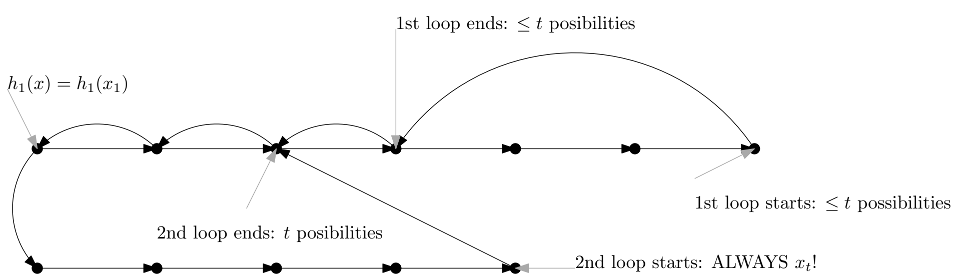
1.2 Cuckoo Hashing

1. ...

2. ...

3. We analyze this case by counting the number of 2-cycles subgraphs of the cuckoo graph, from which we derive the probability that this case occurs.
   Let again $h_1(x_1), \ldots, h_{1/2}(x_t)$ denote a walk of length t, this time containing exactly two cycles and stopping when the second loop occurs. First, what is the number of "topologies" for such walks?

   Look at the following picture:



   Hence, there are at most t^3 "topologies". For any of the t possibilities, apart from the first, we can choose one of the n elements from S (disregarding the fact that not every choice is valid). Hence, there are at most $t^3 n^{t-1}$ walks that start with x and contain exactly 2 cycles (and end with the 2nd cycle).

   In order to embed these walks in the two hash tables, we have to choose a hash value $h_.(x_i)$ for all $2 \leq i \leq t$; this can be done in m^{t-1} different ways. In total, there are at most

$$t^3 n^{t-1} m^{t-1}$$

different length-t walks in the cuckoo graph (starting at $h_1(x)$) containing 2 cycles and ending in the 2nd cycle.

Now the probability of *arbitrary but fixed* such walks on t elements from S is at most $\frac{1}{m^{2t}}$:

$$\begin{aligned}
&\mathbb{P}[\underbrace{h_1(x_1) = i_1 \wedge h_2(x_1) = j_1}_{\text{1st edge } (i_1, j_1)} \wedge \cdots \wedge \underbrace{h_1(x_t) = i_t \wedge h_2(x_t) = j_t}_{\text{last edge}(i_t, j_t)}] \\
=&\mathbb{P}[h_1(x_1) = i_1 \wedge \cdots \wedge h_1(x_t) = i_t] \cdot \mathbb{P}[h_2(x_1) = j_1 \wedge \cdots \wedge h_2(x_t) = j_t] \\
\leq&\frac{1}{m^t} \cdot \frac{1}{m^t} \text{ (by properties of } (1, \log n)\text{-universal hashing)} \\
=&\frac{1}{m^{2t}}
\end{aligned}$$

Hence, the probability of being in case 3 is, at most

$$\begin{aligned}
&\sum t = 3^{6 \log n} \frac{t^3 n^{t-1} m^{t-1}}{m^{2t}} \\
=&\sum t = 3^{6 \log n} \frac{t^3 n^{t-1}}{m^{t+1}} \\
=&\frac{1}{mn} \sum_{t=3}^{6 \log n} t^3 \left(\frac{n}{m}\right)^t \\
=&\frac{1}{mn} \underbrace{\sum_{t \geq 1} t^3 \left(\frac{n}{m}\right)^t}_{=O(1) \text{ since } \frac{n}{m} = \frac{1}{2}} \\
=&O\left(\frac{1}{n^2}\right).
\end{aligned}$$

This is the probability of a rehash in case 3.

Summarizing all three cases, we see that the probability that a *single* insertion causes a rehash is $O(\frac{1}{n^2})$. Therefore, the probability that n insertions cause a rehash is $O(\frac{1}{n})$, so a rehash (on elements) is successful with probability $1 - O(\frac{1}{n})$, almost always! So the expected number of trials is $O(1)$ until the rehash is successful (# trials = 1 + # unsuccessful trials, $\mathbb{E}$[unsuccessful trials] = $\sum_{t \geq 1} t \cdot \frac{1}{n^t} = \frac{n}{(n-1)^2} = O(\frac{1}{n})$), and the rehash takes $O(n)$ time in expectation.

In total, the *amortized* time for an insert-operation is

$$\underbrace{O(1)}_{\text{case } \frac{1}{2} \text{ expected time}} + O(\underbrace{\frac{1}{n}}_{\text{amortized over } n \text{ elements}} \cdot \underbrace{n}_{\text{cost of rebuild in expectation in cases } 1-3}) = O(1)$$

in *expectation*.

2 Predecessor Queries

If searching for an element $x \notin S$, hashing schemes only return the answer that x is not in the set. In some applications it might be interesting to know elements *closest* to x, either before or after. These are called *predecessors* and *successors*, respectively. They are formally defined by

$$\text{pred}(x) = \max\{y \in S \mid y \leq x\}, \text{ and}$$
$$\text{succ}(x) = \max\{y \in S \mid y \geq x\}.$$

In what follows, we assume again that S is a subset of a bounded universe $\mathcal{U} = \{0, 1, \ldots, u-1\}$. We also assume $u = 2^w$, where w is the bit length of the keys. Note that since $S \subseteq \mathcal{U}$ we have $n \leq u$ and therefore $\lg n \leq w$.

2.1 Static Predecessor Queries

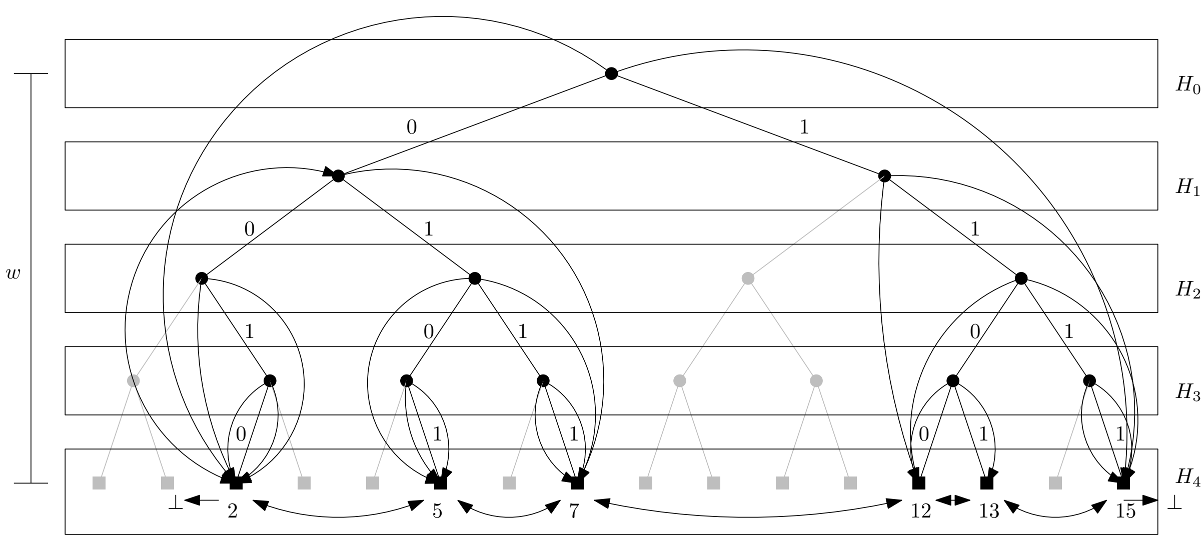
As with perfect hashing, assume first that S is static. We introduce a data structure called *y-fast tries* that answers predecessor (and successor) queries in $O(\lg \lg u) = O(\lg w)$ time.

Recommended reading:

- D.E. Willard: Log-logarithmic Worst-Case Range Queries are Possible in Space $\Theta(N)$. Inform. Process. Lett. 17(2): 81–84 (1983)
- Script from course "Advanced Data Structures" at MIT by Erik Demaine and Oren Weimann, Lecture 12, Spring 2007.
  Available online at http://courses.csail.mit.edu/6.851/spring07/scribe/lec12.pdf

The idea is to store the *binary representation* of the numbers $x \in S$ in a *binary trie* of height w.

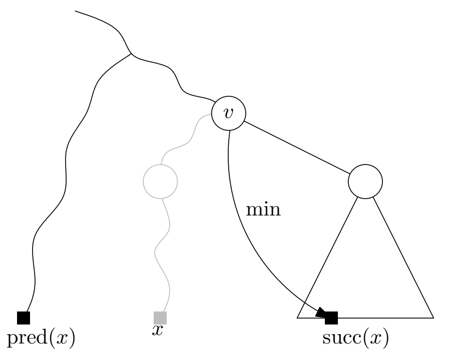
Example: Let $u = 16$ and $S = \{2, 5, 7, 12, 13, 15\}$



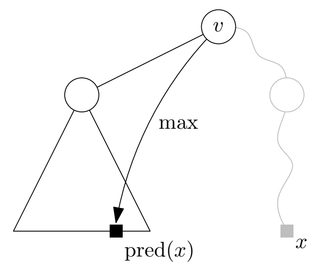
It is actually useful to imagine the trie as embedded into the *complete* binary trie over the full universe $\mathcal{U}$, as shown by the gray lines in the example above.

The trie is stored by w hash tables of size $O(n)$ each: on every level l of the trie, a hash table H_l stores the nodes that are present on that level. Formally, H_l stores all length-l *prefixes* of the numbers in S (H_0 stores the empty prefix ϵ). Each internal node stores a pointer to the minimum/maximum element in its subtree (we could also store these numbers *directly* at each node, but if satellite information is attached at the elements in S then a pointer is probably more useful). Finally, the leaves (= elements in S) are connected in a *doubly linked list.* If we use perfect hashing on each level, then the overall space if the data structure is $O(nw)$. This data structure is called *"x-fast trie"* in the literature.

To answer a query pred(x), in the *imaginary* complete trie we go to the leaf representing x and walk up until finding a node that is part of the *actual* trie. Then we have to distinguish between two cases:



(b) following min-pointer from v brings us to succ(x). We use the linked list to find pred(x).



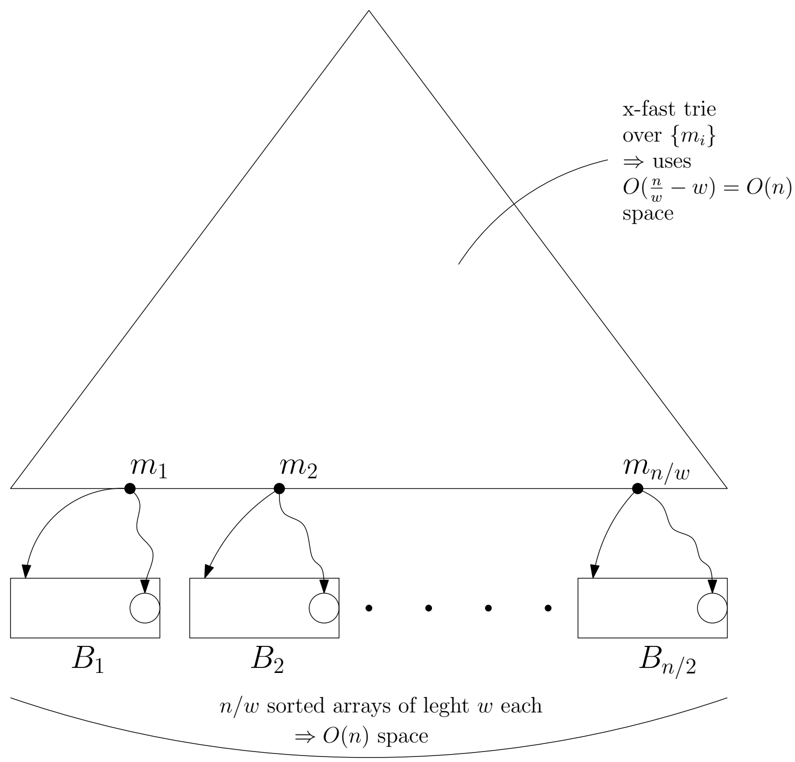
(a) following max-pointer from v gives pred(x) directly.

As described, the search of x would take $O(w)$ time. To bring this down to $O(\lg w)$, we use *binary search* on the levels of the trie: first set $l \leftarrow \lfloor \frac{w}{2} \rfloor$ and check if the length-l prefix of x is stored in H_l. Depending on the outcome of this composition, continue with $\lfloor \frac{w}{4} \rfloor$ or $\lfloor \frac{3w}{4} \rfloor$, and so on, until finding v in $O(\lg w)$ time.

So far we use $O(nw)$ space (for the w hash tables). To bring this down to $O(n)$, we do the following. Before building the x-fast trie, we group w consecutive elements from S into blocks $B_1, \ldots, B_{\lceil n/w \rceil}$. Formally,

$$S = \bigcup_{1 \leq i \leq \lceil n/w \rceil} B_i, |B_i| = w \text{ for } 1 \leq i \leq \frac{n}{w},$$

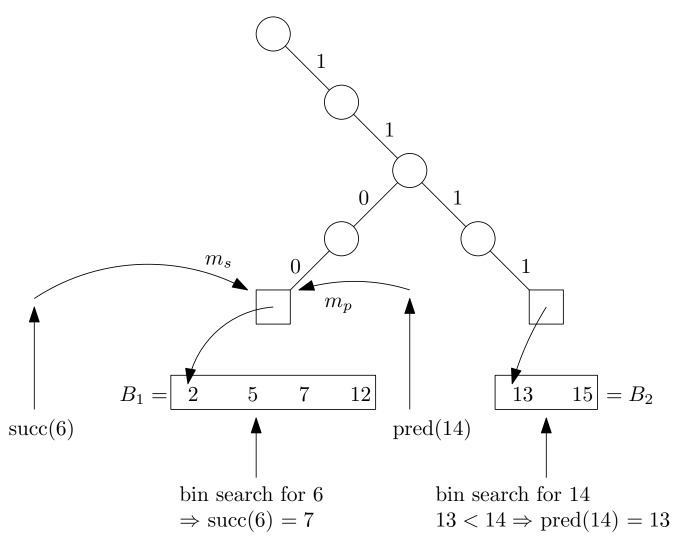
and if $x \in B_i$, $y \in B_j$ then $x < y$ iff $i < j$.
Let $m_i = \max\{x \in B_i\}$ be a *representative* of each block. We build the x-fast trie only on the set $\{m_1, m_2, \ldots, m_{\lceil n/w \rceil}\}$, and the B_i's are stored in sorted arrays.



To answer pred(x), we first use the x-fast trie to find the representative-predecessor m_p of x. Then pred(x) is either m_p itself, or it is in B_{p+1}. For the latter case, we need to *binary search* B_{p+1} for x in additional $O(\lg w)$ time.

To answer succ(x), we first use the x-fast trie to find the representative-successor m_s of x. Then succ(x) must be in B_s and can be found by a binary search over B_s.

Example: $B_1 = \{2, 5, 7, \underbrace{12}_{m_1}\}$, $B_2 = \{13, \underbrace{15}_{m_2}\}$



Note: The structure is called *"y-fast trie"* and can be made *dynamic* by

(a) using dynamic hashing (e.g. cuckoo hashing) for the x-fast trie,

(b) using balanced search trees of size between $\frac{1}{2}w$ and $2w$ instead of sorted arrays, and

(c) not requiring the representative elements be the maxima of the groups, but any element separating two consecutive groups.

Then a insertion/deletion first operates on the binary trees and only if the trees become too big/small we split/merge them and adjust the representatives in the x-fast trie (using $O(w)$ time). As this adjustment only happens every $\Theta(w)$ operations, we got *amortized & expected* $O(\lg w)$ time. The next section shows how to achieve such times in the worst case.

Summary:

y-fast tries	static	dynamic
pred(x)/succ(x)	$O(\lg\lg n)$ w.c.	$O(\lg\lg n)$ exp.
insert(x)/delete(x)	-	$O(\lg\lg n)$ exp. & am.
preprocessing	$O(n)$ exp.	-
space	$O(n)$ w.c.	$O(n)$ w.c.

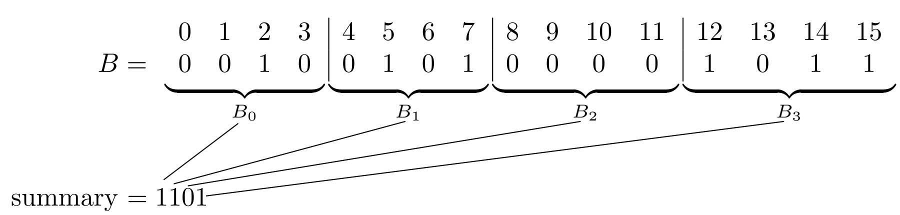
2.2 Dynamic Predecessors — van Emde Boas Trees

Recommended reading:

- P. van Emde Boas: Preserving Order in a Forest in less than Logarithmic Time. Proc. 16th anual Symposium on Foundations of Computer Science (FOCS), p. 75–84, 1975.
- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. Introduction to Algorithms (3rd ed.). MIT Press, 2009. Chapter 20.
- André Schulz: Skriptum zur Vorlesung "Datenstrukturen für Fortgeschrittene" d. Universität Münster (WS 10/11), VL 19. Available at http://wwwmath.uni-muenster.de/logik/Personen/Schulz/WS10/VL10.pdf.

We start by defining a bit-vector B of size u such that the i'th bit in B is set iff $i \in S$. We then divide B into $\sqrt{u}$ (conceptual) blocks $B_0, \ldots, B_{\sqrt{u}}$ of size $\sqrt{u}$ each. An additional bit-vector summary$[0, \sqrt{u}]$ marks those blocks that are nonempty: summary$[i] = 1$ iff B_i contains at least on 1.

Example: Let again $u = 16$ and $S = \{2, 5, 7, 12, 13, 15\}$



Searching predecessors/successors can be done by first finding the first nonempty block to the left of x by scanning summary, and then scanning the corresponding block. Both steps take $O(\sqrt{u})$ time. Insertions and deletions can be realized in $O(1)$ time: set/delete the corresponding bits in B and summary.

Observe that taking the square root of u corresponds to halving the number of bits:

$$\lg \sqrt{u} = \lg 2^{\frac{1}{2}w} = \frac{1}{2}w$$

$$\text{key } x = \overbrace{\underbrace{\boxed{\qquad\qquad\qquad}}_{\text{high}(x)=\text{ block nr.}}}^{\lfloor w/2 \rfloor}\overbrace{\underbrace{\boxed{\qquad\qquad\qquad}}_{\text{low}(x)=\text{ pos. in block}}}^{\lceil w/2 \rceil}$$

The numbers high(x) and low(x) can be efficiently computed by masking and shift operations (in $O(1)$ time).

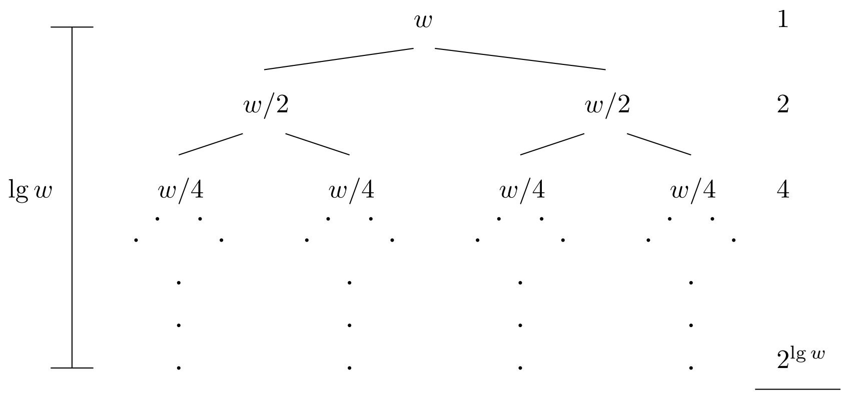
Now observe that finding the first nonempty block to the left of high(x) corresponds to a *predecessor search in the summary vector*. Likewise, the scanning of single blocks also corresponds to a predecessor search. This suggests the use of *recursion*, as the summary-vector and each block are only half the original size u.

```
function SUCC(B, x)
    inblock-succ ← succ(B_high(x), low(x))                    ▷ successor in block
    if inblock-succ ≠⊥ then
        return inblock-succ + high(x)·√|B|
    else
        succ-block ← succ(B.summary, high(x))
        if succ-block =⊥ then
            return ⊥
        else
            return min(B_succ-block) +succ-block · √|B|
                   ⇒store minimum with each block
        end if
    end if
end function
```

To analyze the running time, observe that there are at most two recursive calls on problems of size $\sqrt{|B|}$. Hence, the running time is described by the recursion

$$T(u) = 2T(\sqrt{u}) + O(1)$$

By using the Master Theorem or drawing the recursion tree ($w = \lg u$),



$$\sum = O(2^{\lg w}) = O(w) = O(\lg u)$$

this solves to $T(u) = \Theta(\lg u)$.
This is too slow! Our aim is to modify the algorithm such that only *one* recursive call is made, because the running time is

$$\begin{aligned} T'(u) &= T'(\sqrt{u}) + O(1) \\ &= \Theta(\lg \lg u). \end{aligned}$$