1 Hashing

1.1 Perfect Hashing

1.2 Cuckoo Hashing

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 $h_i(x_i)$ for all $2 \leq i \leq t$; this can be done in $m^{t-1}$ different ways. In total, there are at most

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

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 \textit{arbitrary but fixed} such walks on \( t \) elements from \( S \) is at most \( \frac{1}{m^t} \):

\[
\begin{align*}
\mathbb{P}[h_1(x_1) &= i_1 \land h_2(x_1) = j_1 \land \cdots \land h_1(x_t) = i_t \land h_2(x_t) = j_t] \\
&= \mathbb{P}[h_1(x_1) = i_1 \land \cdots \land h_1(x_t) = i_t] \cdot \mathbb{P}[h_2(x_1) = j_1 \land \cdots \land h_2(x_t) = j_t] \\
&\leq \frac{1}{m^t} \cdot \frac{1}{m^t} \text{(by properties of (1, log } n)\text{-universal hashing)} \\
&= \frac{1}{m^{2t}}
\end{align*}
\]

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

\[
\sum_{t=3}^{t} = 3^6 \log n \frac{t^3 n^{t-1} m^{t-1}}{m^{2t}} = \sum_{t=3}^{t} = 3^6 \log n \frac{t^3 n^{t-1} m^{t-1}}{m^{t+1}}
\]

\[
= \frac{1}{mn} \sum_{t=3}^{t} t^3 \left( \frac{n}{m} \right)^t = \frac{1}{mn} \sum_{t=1}^{t} t^3 \left( \frac{n}{m} \right)^t
\]

\[
= \mathcal{O}(1) \text{ since } \frac{n}{m} = \frac{1}{2}
\]

\[
= \mathcal{O}\left( \frac{1}{n^2} \right).
\]

This is the probability of a rehash in case 3.

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

In total, the \textit{amortized} time for an insert-operation is

\[
\mathcal{O}(1) + \mathcal{O}\left( \frac{1}{n^2} \right) \cdot \mathcal{O}\left( \frac{n}{m} \right) = \mathcal{O}(1)
\]

in \textit{expectation}. 

2
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:


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 $\epsilon$). 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 $\text{pred}(x)$, 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:

![Diagram showing two cases for finding pred(x)](image)

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

(a) following max-pointer from $v$ gives $\text{pred}(x)$ 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 $l \leftarrow \lfloor \frac{w}{4} \rfloor$ 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 \( \text{pred}(x) \), we first use the x-fast trie to find the representative-predecessor \( m_p \) of \( x \). Then \( \text{pred}(x) \) 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 \( \text{succ}(x) \), we first use the x-fast trie to find the representative-successor \( m_s \) of \( x \). Then \( \text{succ}(x) \) must be in \( B_s \) and can be found by a binary search over \( B_s \).

**Example:** \( B_1 = \{2, 5, 7, 12\} \), \( B_2 = \{13, 15\} \)
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:
\[
\begin{array}{c|c|c}
\text{y-fast tries} & \text{static} & \text{dynamic} \\
\hline
\text{pred}(x)/\text{succ}(x) & O(lg lg n) \text{ w.c.} & O(lg lg n) \text{ exp.} \\
\text{insert}(x)/\text{delete}(x) & - & O(lg lg n) \text{ exp. & am.} \\
\text{preprocessing} & O(n) \text{ exp.} & - \\
\text{space} & O(n) \text{ w.c.} & O(n) \text{ w.c.} \\
\end{array}
\]

2.2 Dynamic Predecessors — van Emde Boas Trees

Recommended reading:


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 $S = \{2, 5, 7, 12, 13, 15\}$

\[
B = \begin{bmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
\end{bmatrix}
\]

summary = 1101
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:

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

![Diagram showing key \( x \) with high and low components](https://example.com/diagram.png)

The numbers \( \text{high}(x) \) and \( \text{low}(x) \) 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 \( \text{high}(x) \) 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 \( u \).

```plaintext
1: function succ(B, x)
2:     inblock-succ ← succ(B_{high(x)}, low(x))          \(\triangleright\) successor in block
3:     if inblock-succ ≠ ⊥ then
4:         return inblock-succ + high(x) \cdot \sqrt{|B|}
5:     else
6:         succ-block ← succ(B.summary, high(x))
7:         if succ-block = ⊥ then
8:             return ⊥
9:         else
10:            return \(\min(B_{\text{succ-block}}) + \text{succ-block} \cdot \sqrt{|B|}\)
11:               \(\Rightarrow\) store minimum with each block
12:     end if
13: 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 \( (w = \lg u) \),
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

$$T'(u) = T'\left(\sqrt{u}\right) + O(1)$$

$$= \Theta(\lg \lg u).$$