Advanced Data Structures

Based on the lectures of Prof. Moshe Lewinstein

notes by
G. Hoch & A. Aped

November 28, 2016

Contents

Chapter 1

The Dictionary Problem

Given a "world" U such that:

$$U := \{0, 1, \ldots, u-1\}$$

we want to store a sub-group $S \subseteq U$ such that $|S| = n$
and we want to support several queries over S:

1. Existence:
   - **input:** $x \in U$
   - **output:** 1 if $x \in S$, 0 otherwise.
2. Successor
   - **input:** $x \in U$
   - **output:** $y | y = min\{z \in S | x \leq z\}$
3. Predecessor
   - **input:** $x \in U$
   - **output:** $y | y = max\{z \in S | x \geq z\}$
4. Insert
   - **input:** $x \in U$
   - **output:** $S \leftarrow S \bigcup \{x\}$
5. Delete
   - **input:** $x \in U$
   - **output:** $S \leftarrow S \backslash \{x\}$

We will try to deal with the existence problem:

1.1 Universal Hashing, Perfect Hashing & FKS

On the following proposals, we will try to answer the questions:

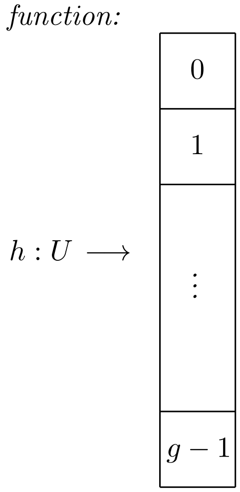
1. How long does the query take?
2. What is the size of the data structure for S?
3. How long does it take to build this data structure?
4. Does the data structure support changes (insert/delete)?

Possible solutions:

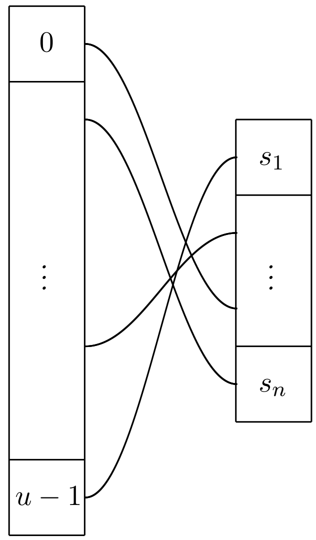
Proposition. 1.1.1 *(for fast queries)*
we will store an array A the size of u, such that $A[i]$
(where $0 \leq i \leq u-1$) contain true $\iff i \in S$
Space: $O(u)$
Time: $O(1)$

Proposition. 1.1.2 *(for space efficiency)*
a sorted array for S, for every query we can use binary search
Space: $O(n)$
Time: $O(log(n))$

Proposition. 1.1.3 *(for both space & time efficiency)*
Hash function:



we would want to ensure that $g = O(n)$, one way to ensure that, is to use chaining, but then, worst case scenario, if we have a lot of collisions, queries will take longer than $O(1)$. another option is to choose a "good" function $h : U \to [c \cdot n]$. if we take a random function, then the *number of collisions will be $O(1)$, but it is highly unlikely to find a compact representation for a random function. that is, the only way to represent the function, is with a table the of size u:*



so it didn't save us space, and we are better off with the naieve solution suggested in Proposition 1.1.1 (a boolean array the size of u).

Universal Hashing

Definition. 1.1.4 *a family $\mathcal{H}$ of hash functions $h : U \to [m]$*[1] *is called* universaly weak*, if for some $x, y \in U$ and for every $h \in \mathcal{H}$:*

$$Pr\Big[h(x) = h(y)\Big] \leq \frac{1}{m}$$

assuming we are using a universaly weak function h, what would be the expected number of collisions?

$$I_x(y) = \begin{cases} 1 & h(x) = h(y) \\ 0 & otherwise \end{cases}$$

well, the number of collisions on $h(x)$ is:

$$\sum_{y \in S} I_x(y)$$

so the expected number of collisions is:

$$E\Big(\sum_{y \in S} I_x(y)\Big) = \sum_{y \in S} E\Big(I_x(y)\Big) = \sum_{y \in S} Pr\Big[h(x) = h(y)\Big] = 1 + \sum_{y \neq x} Pr\Big[h(x) = h(y)\Big] \leq 1 + (n-1)\frac{1}{m}$$

So, if we choose $m = n$, we will get that the expected number of collisions $\leq 1 + 1 = 2$
OK, so how can we find such functions?
$h_{a,b}(x) = ((a \cdot x + b) \bmod p) \bmod m$, where p is prime, and $m < p$, also $a, b \in \{0, \ldots, p-1\}, a \neq 0$,
$\mathcal{H}_p = \{h_{a,b} | 0 < a \leq p-1, 0 \leq b \leq p-1\}$

[1] $[m] = \{0, \ldots, m-1\}$

Claim. 1.1.5 *$\mathcal{H}_p$ is universaly weak.*

Proof. Omitted. Can be found in [?]. ■

Definition. 1.1.6 *a hash function is* perfect *for $S \subseteq U$ if $\forall x, y \in S, x \neq y \Longrightarrow h(x) \neq h(y)$*

We will take a universal family $\mathcal{H}_m$, such that $m = n^2$, and pick $h \in \mathcal{H}_m$ at random.
$\mathcal{H}_m =$ [2] $\{h_1, \ldots, h_{m^2}\} = \{h_1, \ldots, h_{n^4}\}$, $S = \{x_1, \ldots, x_n\}$

$$\#\text{collisions} = \sum_{\substack{x \neq y \\ x,y \in S}} I_x(y)^{3}$$

$$E\Big(\sum_{\substack{x \neq y \\ x,y \in S}} I_x(y)\Big) = \sum_{\substack{x \neq y \\ x,y \in S}} E\Big(I_x(y)\Big) = \sum_{\substack{x \neq y \\ x,y \in S}} Pr\Big[h(x) = h(y)\Big] \leq \binom{n}{2} \cdot \frac{1}{m} = \frac{n(n-1)}{2} \cdot \frac{1}{n^2} \leq \frac{1}{2}$$

$$\Longrightarrow Pr\Big[\text{there's a collision}\Big] = {}^{4}Pr\Big[X^{5} \geq 1\Big] \leq \frac{\frac{1}{2}}{1} = \frac{1}{2}$$

So actually, not only the expectation for a collision $\leq \frac{1}{2}$, the chance that there will even be a collision is equal to $\frac{1}{2}$. checking for collisions = building the table: $h(x_1), h(x_2), \ldots, h(x_n)$
an algorithm to build a perfect hash function, when $m = n^2$:

Algorithm. 1.1.7

1. *take the universal family $\mathcal{H}_{n^2}$.*
2. *choose $h \in \mathcal{H}_{n^2}$ at random.*
   2.1. *if h is not perfect for S, go back to 2.*

$$\begin{aligned} E\Big(\text{the algorithm running time}\Big) &= \sum_{i=1}^{\infty} n \cdot Pr\Big[\text{the algorithm ran for } i \text{ iterations}\Big] \\ &= \sum_{i=1}^{\infty} n \cdot \frac{1}{2^i} = n \cdot \sum_{i=1}^{\infty} \frac{1}{2^i} = 2n \end{aligned}$$

at the end of the process, we were able to find a perfect hash function for S, such that the expected time for building the function is $2n(O(n))$, and query time is always $O(1)$. the downside of the solution is about the space complexity. that's because we pay $m = n^2$ in space for the table, when most of it stays empty.

Space: $O(n^2)$
Query Time: $O(1)$
Build Time: $E\big(O(n)\big)$

[2] there are m possibilities to choose the coefficient a, and m possibilities to choose b, so overall, the function family $\mathcal{H}_m$ contain $m \cdot m$ distinct functions.

[3] where $I_x(y)$ is same as before: $I_x(y) = \begin{cases} 1 & h(x) = h(y) \\ 0 & otherwise \end{cases}$

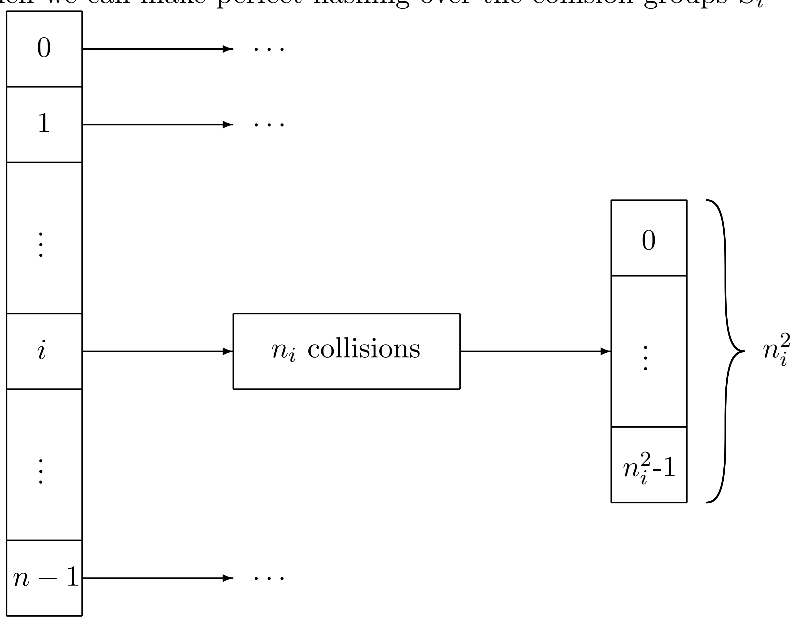
[4] **reminder.** Markov's inequality: $Pr\big[X \geq t\big] \leq \frac{E(X)}{t}$

[5] X is the random variable that stands for the number of collisions.

improvment: FKS[6] method.
we will choose $m = n$:

$$E\Big(\text{the number of collisions}\Big)^{7} \leq \binom{n}{2} \cdot \frac{1}{m} = \frac{n(n-1)}{2} \cdot \frac{1}{n} \leq \frac{n}{2}$$

and then we can make perfect hashing over the collision groups S_i



$S_i = \{x \in S | h(x) = i\}, |S_i| = n_i$
we will apply perfect hashing for S_i, so that the "main" hash table contains only a pointer to a secondary hash table for S_i

Algorithm. 1.1.8

1. *take the universal family $\mathcal{H}_n$.*
2. *choose $h \in \mathcal{H}_n$ at random.*
   2.1. *if the number of collisions in $h > n$, go back to 2.*
3. *create a table for $h : S \to [n]$.*
4. *for every index i in the table:*
   4.1 *choose $h_i \in \mathcal{H}_{n_i^2}$ at random.*
   4.2 *if h_i is not perfect hash for S_i, go back to 5.1.*

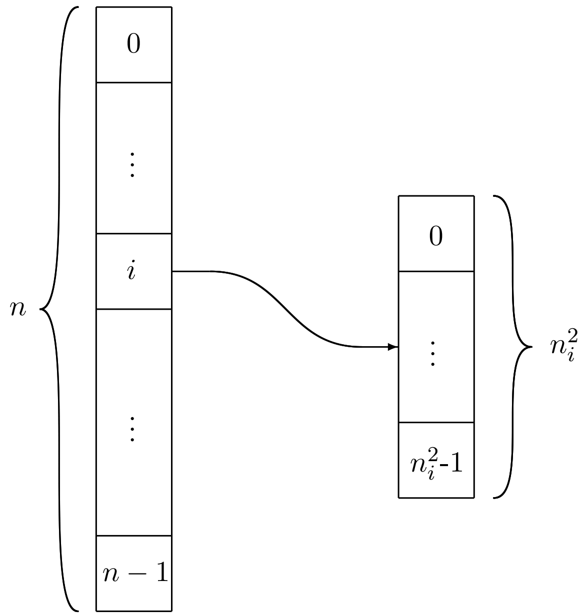
---

[6] FKS - Michael L. Fredman and János Komlós and Endre Szemerédi [?]

[7] according to Markov's inequality theorem: $Pr\Big[X \geq n\Big] \leq \frac{\frac{n}{2}}{n} = \frac{1}{2}$, and that's why finding a function h with n collisions at most, would take 2 iterations (in expectation).

But, we do have some open questions:

1. How much space is required for this data structure?
2. how long doe's it takes to build it?



The size of this data structure is:

$$O\Big(n + \sum_{i=0}^{n} n_i^2\Big) = {}^{8} O\big(n + (\#\text{ collisions})\big) = O(n)$$

$\implies$ the space complexity is
Space: $O(n)$
what about time complexity?
Build time: $O\big(n + \sum n_i\big) = O(n)$[9]
Query time: $O(1)$[10]

---

[8] when n_i elements, all colliding with each other, it means we have $\frac{n_i(n_i-1)}{2}$ collisions, and that is why $n_i \leq \sqrt{n}$, so $n_i^2 \leq n$, since all the collisions in S are less than n, i.e. $\sum n_i \leq n \Longrightarrow n = (\sqrt{n})^2 \geq \sum n_i^2$

[9] in expectation

[10] Query algorithm:

Algorithm. 1.1.9

1. *search the main table for a pointer to n_i, if null return false.*
2. *search n_i for the requested element and return the answer.*

this algorithm works in constant time, and this is why query time is always $O(1)$.

1.2 Van-Emde Boas

reminder. we have a world $U = \{0, \dots, u-1\}$, and a subset $S \subseteq U$. we want to support the actions: *insert,delete,successor,predecessor*, where *successor* & *predecessor* are defined by:

$$succ(x) = y | y = min\{z \in S | x \leq z\}$$

$$pred(x) = y | y = max\{z \in S | x \geq z\}$$

which (simple) data structure can we use to support these actions?[11]

Proposition. 1.2.1 *we can store S's elements in an array.*

Option 1. the array will be the size of $|S|$:

$S = \{x_1, \dots, x_n\}$

$S =$

x_1	x_2	$\cdots$	x_n

Successor: $O(log(n))$

Insert: $O(n)$

Option 2. the array will be the size of $|U|$:

$$A[x] = \begin{cases} 1 & x \in S \\ 0 & otherwise \end{cases}$$

$A =$

1	2		x		$u-1$
		$\cdots$		$\cdots$	

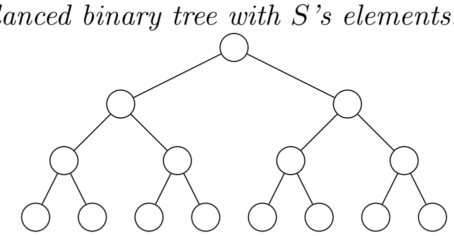
Successor: $O(u)$

Insert: $O(1)$

Proposition. 1.2.2 *a balanced binary tree with S's elements.*

Successor: $O(log(n))$

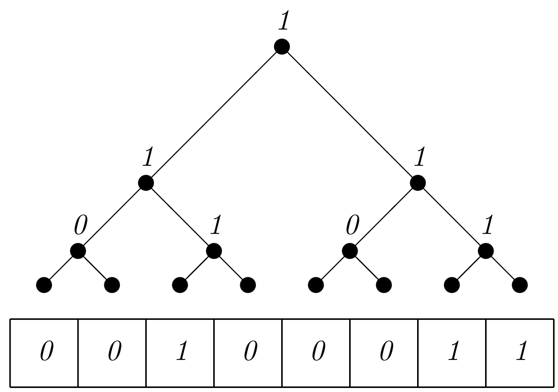
Insert: $O(log(n))$



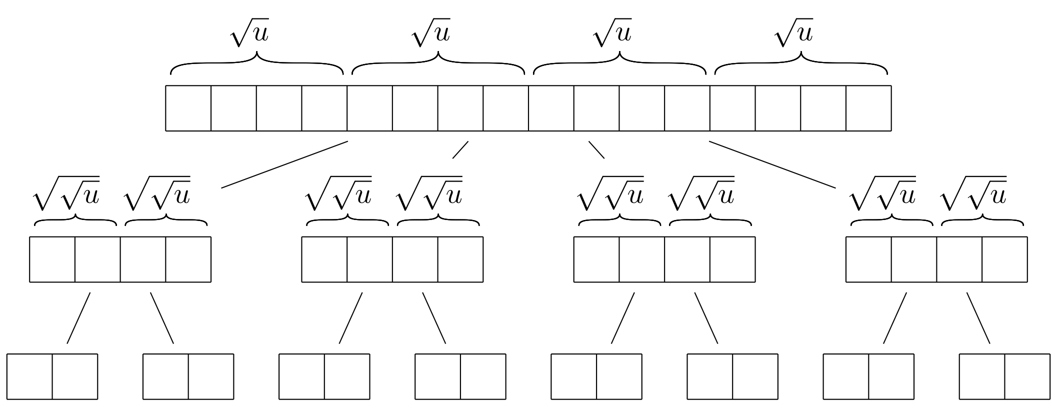
Well, $S \subseteq U$, and usally $S \subsetneq U$, so $|S| < |U|$, but we will assume that S is not "much" smaller than U. That is, u is polynomic in n (not exponential), so $u \in poly(n)$, for instance: $u = n^5$. in that case, $log(n)$ is too much time. Assuming there's a solution that takes $O(log(log(u))$ time, may be such a solution is preferable to us. i.e. n is very big, so:
$u \in poly(n) \Longrightarrow log(u) = c \cdot log(n) \Longrightarrow log(n) = O(log(u))$, and the balanced binary tree solution is not good enough for us.

[11] the *predecessor* is symetric to the *successor*, so from now on, without loss of generality, can talk only on *successor* (*predecessor* will be implemented much like *successor*). also, *delete* can be implemented with a "flag" indicating if the element was deleted or not, so we will not mention *delete* either.

Proposition. 1.2.3 *a tree that is built above U: on one hand, we improved the solution suggested in Proposition 1.2.1, such that* successor *takes $O(log(u))$, but* insert *also takes $O(log(u))$ instead of the previously $O(1)$ time. This is because now we need to update all the nodes while moving on the path towards the root. Searching for an element will be done as follows: assume we are looking for the successor of x, then we start from $U[x]$, and move up untill we reach a vertex that is flaged with 1 (of course, only in case that $x \in S$, and so $U[x] = 0$, otherwise x's parent will already be flages as 1). if we reached a node from left, we will search the successor on the sub-tree hanging on the right child, but if we reached the node from right, it is flaged 1 only because an element in the range that is smaller than x, and we should continue upwards on the path. but, from now on, on every node we reached to from the left child, we have to "peek" in the right child's flag, and check if it's 1.*



Overall, *successor* takes $O(log(u))$ time. we will try to improve:
We can divide the vector into $\sqrt{u}$ segments the size of $\sqrt{u}$ each. each of the segments, we can divide further into $\sqrt{\sqrt{u}}$ segments the size of $\sqrt{\sqrt{u}}$, and so on...



$$
\begin{aligned}
T(u) &= \text{time to do “something” in a structure of size } u \\
T(u) &= T(\sqrt{u}) + 1 \\
T(2) &= 1 \\
T(u) &= T(u^{\frac{1}{2}}) + 1 = T(u^{\frac{1}{2^2}}) + 2 = T(u^{\frac{1}{2^3}}) + 3 = \dots \\
&\Longrightarrow T(u^{\frac{1}{2^i}}) + i \\
u^{\frac{1}{2^i}} &= 2 \\
&\iff \left(2^{log(u)}\right)^{\frac{1}{2^i}} = 2 \iff log(u) \cdot \frac{1}{2^i} = 1 \iff 2^i = log(u) \\
&\iff i = log(log(u))
\end{aligned}
$$

Actually, we can look at the structure “recursively”, that is, in the lowest level there's $\sqrt{u}$ structures, each the size of $\sqrt{u}$, in the level above, there's $\sqrt{\sqrt{u}}$ structures, each the size of $\sqrt{\sqrt{u}}$. well, now we have $\sqrt{t}$ arrays at each level, where's t is the number of arrays in the preceding level.

Notation. 1.2.4

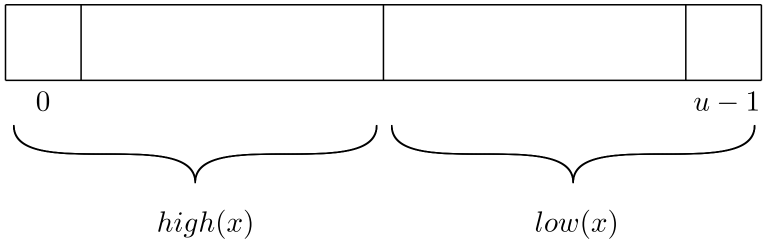
$$
\begin{aligned}
sub[0] &\text{ - } \textit{the structure (array) of the first } \sqrt{u} \textit{ elements} \\
sub[1] &\text{ - } \textit{the structure (array) of the next } \sqrt{u} \textit{ elements} \\
&\vdots \\
sub[u-1] &\text{ - } \textit{the structure (array) of the last } \sqrt{u} \textit{ elements}
\end{aligned}
$$

In general, $sub[i]$ is the structure of the $\sqrt{u}$ elements: $i \cdot \sqrt{u}, \dots, (i+1) \cdot \sqrt{u} - 1$
well, how would we implement $succ(x)$ or $pred(x)$ algorithms?

Proposition. 1.2.5 *we can look at a higher level, and check if the representative is* 0*, if so, we should continue the scan untill the end of the structure, if all were zeros, we should continue to the next structure, and so on. if we found* 1*, we should go down to it's sub-set, where it is guaranteed to have a* successor/predecessor*.*
Time: $O(\sqrt{u})$

Still not good enough. we can look at the binary representation of an element (number) x in our structure, when we assume (for convenience) that $u = 2^k, k \in \mathbb{N}$. if so, the length of x's representation is of size $log(u)$. we will divide the representation of x in half, and notate:

Notation. 1.2.6



Each part of the representation, is the length of $\frac{1}{2} \cdot log(u)$, and the number of elements we can represent with $\frac{1}{2} \cdot log(u)$ bits, is: $2^{\frac{1}{2} \cdot log(u)} = (2^{log(u)})^{\frac{1}{2}} = u^{\frac{1}{2}} = \sqrt{u}$.
in general, according to our notation:

$$x = high(x) \cdot 2^{\frac{log(u)}{2}} + low(x)$$

So, if we want to find the set/array $sub[i]$ of x, we can compute: $x \in sub[high(x)]$. we can see that all the first $\sqrt{u}$ elements has the same $high(x)$:

the first $\sqrt{u}$ elements	$0\ 0\ \cdots\ 0$	$0\ \cdots\ 0\ 0$
	$0\ 0\ \cdots\ 0$	$0\ \cdots\ 0\ 1$
	$\vdots$	$\vdots$
	$0\ 0\ \cdots\ 0$	$1\ \cdots\ 1\ 1$
the next $\sqrt{u}$ elements	$0\ 0\ \cdots\ 1$	$0\ \cdots\ 0\ 0$
	$\vdots$	$\vdots$
	$0\ 0\ \cdots\ 1$	$1\ \cdots\ 1\ 0$
	$0\ 0\ \cdots\ 1$	$1\ \cdots\ 1\ 1$
$\vdots$	$\vdots$	

Well then, $high(x)$ determines in which "sub-group" x is found, and $low(x)$ determines where exactly in the group it is found. if S, our structure, is of size $|S|$:
$sub[S][0], sub[S][1], sub[S][2], \ldots, sub[S][\sqrt{u} - 1]$, we would probably want to store an extra bit array the size of $\sqrt{u}$ to check quickly if there's an element in the array $sub[S][i]$. that is (we will call this extra array: $summary[S]$): $\big((summary\ AND\ 2^i) == 2^i\big) \Longrightarrow \exists\ x \in sub[S][i]$
so the size of $summary[S]$ is $\sqrt{|S|}$, and it help us determine if $sub[S][i]$ is empty or not, according to the i bit. so how do we perform $insert(x)$?

Algorithm. 1.2.7 *insert(x,S)*

1. *flag* $\leftarrow$ *isEmpty*($sub[S][high(x)]$)
2. $insert(low(x), sub[S][high(x)])$
3. *if(flag)*

   3.1. *then* $insert(high(x), summary[S])$

So, how much time does this takes? (assuming the size of the structure is u):

$$T(u) = 2T(\sqrt{u}) + 1 = 2^2 \cdot T(u^{\frac{1}{2^2}}) + 2 + 1 = 2^3 \cdot T(u^{\frac{1}{2^3}}) + 4 + 2 + 1 = \ldots = 2^i \cdot T(u^{\frac{1}{2^i}}) + \sum_{j=0}^{i-1} 2^j = log(u)$$

Not so good. we are back to $O(log(u))$ time.[12]
This happens because we have 2 recursive calls for *insert*.

[12] $|U| > |S| \Longrightarrow u > n \Longrightarrow log(u) > log(n)$, so we better off with a "regular" balanced tree (AVL/Red-Black/etc'...)

let's check *successor* also:

Algorithm. 1.2.8 *succ(x,S)*

1. $j \leftarrow succ\big(low(x), sub[S][high(x)]\big)$
2. *if* $j < \infty$
   2.1. *return* $high(x) \cdot \sqrt{|S|} + j$
3. $i \leftarrow succ\big(high(x)+1, summary[S]\big)$
4. *if* $i = \infty$
   4.1. *return* ∞
5. $j \leftarrow succ\big(0, sub[S][i]\big)$
6. *return* $i \cdot \sqrt{|S|} + j$

Well, this suggested algorithm takes: $T(u) = 3 \cdot T(\sqrt{u}) + 1 \Longrightarrow O\big(T(u)\big) \gg O\big(log(u)\big)$ which is even worse than what we had on *insert*. the reason for this is that we have 3 recursive calls for *succ*, which we invoke because we want to find the successor element when were in a structure, that queried for an element that is greater than the max element in this structure. this means we need to search the next structure. if we had known this in advanced, it would have saved us a lot of time. we will try to improve:

Proposition. 1.2.9 *we can add to each structure, an extra 2 elements indicating the minimum & maximum elements in the structure.*

$$S = \boxed{summary, sub[S][0], \ldots, sub[S][\sqrt{|S|}-1], min, max}$$

$$sub[S][0] = \boxed{summary, \ldots, min, max} \quad \cdots$$

Now, with the modified structure, how can we find the *successor* of some element x?

Algorithm. 1.2.10 *succ(x,S)*

1. *if* $low(x) < max\big(sub[S][high(x)]\big)$
   1.1. $j \leftarrow succ\big(low(x), sub[S][high(x)]\big)$
   1.2. *return* $high(x) \cdot \sqrt{|S|} + j$
2. *else*
   2.1. $i \leftarrow succ\big(high(x)+1, summary[S]\big)$
   2.2. *if* $i = \infty$
      2.2.1 *return* ∞
   2.3. *return* $i \cdot \sqrt{|S|} + min\big(sub[S][i]\big)$

So now, we only have 1 recursive call, so the overall time for *successor* is
$T(u) = T(\sqrt{u}) + 1 = \ldots = O\Big(log\big(log(n)\big)\Big)$. but we ignored *insert*, which will now need more updates of the new data (min, max) we entered, so, have we worsen the problem for *insert*?. Well, not only that we hav'nt made it worse, we actually made it better! we now only need to update *summary* once, and we don't realy need to update min, max every time. we will update only when we first put an element in, and from now on, we can simply switch our newly inserted element, in case it is bigger (smaller) than the *max* (*min*), and continue with the recursive insertion with the old *max* (*min*) element. let's see how the new algorithm will work:

Algorithm. 1.2.11 *insert(x,S)*

1. *if* $x < min[S]$
   1.1. $switch\big(min[S], x\big)$
2. *if* $isEmpty\big(sub[S][high(x)]\big)$
   2.1. $min\big(sub[S][high(x)]\big) \leftarrow low(x)$
   2.2. $insert\big(high(x), summary[S]\big)$
3. *else*
   3.1. $insert\big(low(x), sub[S][high(x)]\big)$
4. *if* $x > max[S]$
   4.1. $max[S] \leftarrow x$

now, *insert* has only 1 recursive call like *successor*, and so, it takes $O\Big(log\big(log(n)\big)\Big)$ as well. so overall, we improved the time complexity:

Successor: $O\Big(log\big(log(u)\big)\Big)$

Insert: $O\Big(log\big(log(u)\big)\Big)$

the space complexity is:

Space: $O(u)$

these are the final results.
and as the title states, this data structure is named after it's inventor: *Van-Emde Boas*

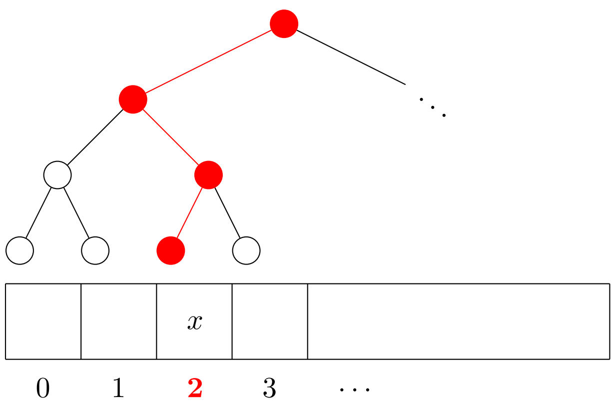
1.3 X-Fast Trie

we allready saw previously the *Van-Emde Boas* trees(1.2), and the space complexity for this data structure was: $S(u) = \sqrt{u}S(\sqrt{u}) + \sqrt{u}$[13], the "height" of the structure is:
$H(u) = H(\sqrt{u}) + 1 = log(log(u))$, so the total space capacity is:

$$S(u) = \sqrt{u} \cdot S(\sqrt{u}) + \sqrt{u} = \sqrt{u} + \sqrt{u}\sqrt{\sqrt{u}} + \sqrt{u}\sqrt{\sqrt{u}} \cdot S(\sqrt{\sqrt{u}}) = \dots$$
$$= u^{\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\dots+\frac{1}{2^{log(log(u))}}} + \sum_{i=0}^{2^{log(log(u))}} u^{\frac{1}{2^i}} = u^{1-\frac{1}{log(u)}} + \sum_{i=0}^{log(u)} u^{\frac{1}{2^i}} \approx O(u)$$

so the Van-Emde Boas data structure can support the actions *insert,delete,successor,predecessor* in $O\Big(log\big(log(u)\big)\Big)$ but the space complexity is linear in u. can it be improved?

$S = \{x_1, x_2, \dots, x_n\}, S \subseteq U$



actually, we don't realy need edges. in the above example,
$x = 2 \Longrightarrow$ the binary representation of x is: 010, and the path from x upwards is:
left$\rightarrow$ *right*$\rightarrow$ *left*, which confronts to $0 \rightarrow 1 \rightarrow 0$. in general, when looking into the bitwise representation of an element x, we can "travel" it's path to the root by starting from the *LSB*, and moving towards the *MSB*, while turning right whenever we encounter 0, and turning left when we encounter 1.

[13] $\sqrt{u}$ in every level for the summary.

1.4 Y-Fast Trie

some text to be inserted here

1.5 Cukoo Hashing

some text to be inserted here

Chapter 2

Data Structures for Strings

2.1 Pattern Matching

2.1.1 Suffix Tree

some text to be inserted here

2.1.2 Suffix Array

some text to be inserted here

2.2 LCP - Longest Common Prefix

2.2.1 Kasai's Algorithm

some text to be inserted here

2.3 RMQ - Range Minimum Query

2.3.1 Cartesian Tree

some text to be inserted here

2.4 Misc

2.4.1 General Problems

LCA,Palindrom,K-mistakes…

2.4.2 Karkaihner & Sanders Algorithm

some text to be inserted here

Chapter 3

Trees

3.1 Tree Decomposition

3.1.1 Centroid Path Decomposition

some text to be inserted here

3.1.2 Heavy Path Decomposition

some text to be inserted here

Chapter 4

Special Properties for Data Structures

4.1 Persistent Data Structures

4.2 Succint Data Structures

Bibliography