“log log n has been proven to go to ∞ but has never been seen to do so.”

-Anonymous

7.1 Ordered Dictionary

We can have the following operations in an ordered dictionary:

- insert(x)
- delete(x)
- member(x)
- next(x)
- prev(x)
- max
- min

But we will not be focusing as much on the max and min operations. We can also note that all operations available to heaps are implementable.

Applications

- Priority queues and their applications
- Sorting
- Sorted key value store (by adding satellite data), which we will not discuss.

Examples with run time

<table>
<thead>
<tr>
<th></th>
<th>Balanced BST</th>
<th>Sorted Array</th>
<th>Bit Array</th>
<th>VEB Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(log log u)</td>
</tr>
<tr>
<td>delete</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(log log u)</td>
</tr>
<tr>
<td>member</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(1)</td>
<td>O(log log u)</td>
</tr>
<tr>
<td>next</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(U)</td>
<td>O(log log u)</td>
</tr>
<tr>
<td>prev</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(U)</td>
<td>O(log log u)</td>
</tr>
</tbody>
</table>
Today all elements we are dealing with are integers in the range \( \{1, 2, \ldots, U - 1\} \).

**Question** Too good to be true? How can we have something sort in \( O(n \log \log U) \) time when we have \( \Omega(n \log n) \) lower bound for sorting?

**Answer** This is NOT comparison based sorting and so the lower bound doesn’t apply! (Also \( \log \log U \) can be \( \Omega(\log n) \))

### 7.2 Bit Array

#### 7.2.1 Bit Array v1.0

![Figure 7.1: Bit Array](image)

**Idea**

\( A[i] = 1 \) if and only if \( i \) is in the set while initially \( A[i] = 0 \) for all \( i \).

\[
\begin{align*}
O(1) \text{ insert}(i): & \quad A[i] = 1 \\
O(1) \text{ delete}(i): & \quad A[i] = 0 \\
O(1) \text{ member}(i): & \quad \text{return } A[i] \\
O(U) \text{ next}(i): & \quad \text{for } j = i, i+1, \ldots, U \\
& \quad \text{if } A[j] == 1 \quad \text{return } j \text{ return } \text{nil} \\
O(U) \text{ prev}(i): & \quad \text{symmetric to above}
\end{align*}
\]

**Question** Both \( \text{prev} \) and \( \text{next} \) take \( O(U) \) time. How can we make this faster?

**Answer** We can break up the range into smaller pieces allowing us to search fewer pieces.

#### 7.2.2 Bit Array v2.0

No we can try a similar process but with two levels.

We have an array \( A \) of size \( \sqrt{U} \) of pointers to other arrays of size \( \sqrt{U} \).

**Idea**

\( A[0] \) corresponds to \( \{0, \ldots, \sqrt{U} - 1\} \)

\( A[1] \) corresponds to \( \{\sqrt{U}, \ldots, 2\sqrt{U} - 1\} \)

\( \vdots \)

\( A[\sqrt{U} - 1] \) corresponds to \( \{U - \sqrt{U}, \ldots, U - 1\} \)

Therefore element \( i \) is represented by \( i \mod \sqrt{U} \) in \( A[\lfloor \sqrt{U} \rfloor] \)
Lecture 7: Van Emde Boas Trees

7.2.3 Bit Array $vk.0$

The 2-level bit array can be extended to a k-level bit array. In this data structure, each array is of size $U^{\frac{1}{k}}$, and contains pointers to bit arrays of size $U^{\frac{1}{k}}$. Similar to the 2-level bit array, each element will also store a count field that tracks the total number of elements in its children arrays. All count fields are initialized to zero. The 1st level can be indexed by the top $\frac{\log U}{k}$ bits, the 2nd level by the next $\frac{\log U}{k}$ bits, and the last layer by the bottom $\frac{\log U}{k}$ bits. An element $i$, if in the set, can be found on the last layer.

**Insert, Delete, and Member**

Similar to the 2-level bit array, insert, delete and member can be done using $O(1)$ operations per level. This results in $O(k)$ time as there are $k$ levels.

**Question** How can we do better for prev and next?

**Answer** More levels!!

\[
O(1) \text{ insert}(i): B = A[\lfloor \frac{i}{\sqrt{U}} \rfloor] \\
B.\text{insert}(i \mod \sqrt{U})
\]

\[
O(1) \text{ delete and member - similar}
\]

\[
O(\sqrt{U}) \text{ next}(i):
B = A[\lfloor \frac{i}{\sqrt{U}} \rfloor] \\
B.\text{next}(i \mod \sqrt{U})
\]

if $j \neq \text{nil}$

\[
\text{return } j + \lfloor \frac{i}{\sqrt{U}} \rfloor \sqrt{U}
\]

for $k = \lfloor \frac{i}{\sqrt{U}} \rfloor + 1, \lfloor \frac{i}{\sqrt{U}} \rfloor + 2, \ldots, \sqrt{U} - 1$

if $A[k].\text{size} \neq 0$

\[
\text{return } A[k].\text{next}(0) + k \sqrt{U}
\]

\[
\text{return } \text{nil}
\]
Next and Previous

In the worst case for the \texttt{next} operation, there will be two scans per level (one going upwards and one downwards), each scan requiring $U^\frac{1}{k}$ operations. With a total of $k$ levels, we can bound the work as:

$$\leq O(1) \ast 2 \ast k \ast U^\frac{1}{k} = O(kU^\frac{1}{k})$$

Best Choice of $k$

The optimum choice of $k$ minimizes $kU^\frac{1}{k}$. Minimizing $g(k) = kU^\frac{1}{k}$, $k \geq 1$, is equivalent to minimizing $\ln(kU^\frac{1}{k})$, $k \geq 1$.

Let $f(k) := \ln(kU^\frac{1}{k}) = \ln(k) + \frac{1}{k} \ln(U)$

To minimize $f(k)$, we will take the derivative with respect to $k$ and equate it to 0.

$$f(k) = \ln(k) + \frac{1}{k} \ln(U)$$

$$f'(k) = \frac{1}{k} - \frac{1}{k^2} \ln(U) = 0$$

$$\frac{1}{k} = \frac{1}{k^2} \ln(U)$$

$$k = \ln(U)$$

Thus, our minimum for $k$ is $\ln(U)$.

$$g(k) = g(\ln U) = (\ln U) \ast U^\frac{1}{\ln U} = \ln U \ast e = O(\log U)$$
Another value of $k$ that achieves the $O(\log U)$ asymptotic bound for $g(k)$, is $k = \log_2 U$.

$$g(k) = g(\log_2 U) = (\log_2 U) \cdot U \cdot \frac{1}{\log_2 U} = \log_2 U \cdot 2 = O(\log U)$$

Taking $k = \log_2 U$, each array would be of size $U^{\frac{1}{k}} = U^{\log_2 2} = 2$.

Insert, delete, and member will take $O(k) = O(\log U)$ time. Next and previous will take $O(kU^{\frac{1}{k}}) = O(\log U)$ time. Thus, all operations will take $O(\log U)$ time.

Figure 7.4: Bit Array $V_{k.0}$ with optimal $k$

We have re-invented balanced search trees!

Consider the member operation. The run-time can be represented by the following recurrence:

$$T(U) = T\left(\frac{U}{2}\right) + 1 = \Theta(\log U)$$

This divides the universe size by constant 2 every recursive call, each of which costs 1. Hence, this recurrence is in $\Theta(\log U)$.

**Our Goal**: Recurrences of the form:

$$T(U) = T(\sqrt{U}) + 1 = \Theta(\log \log U)$$

This recurrence divides the exponent of the universe size by a constant 2 every recursive call, each of which costs 1. Hence, this recurrence is in $\Theta(\log \log U)$.

We can also prove this by the *substitution method*:

**Proof.** Let $m := \log_2 U$ and $S(m) := T(2^m)$. Then,

$$S(m) = T(2^m) = T(U) = T(\sqrt{U}) + 1 = T \left(2^{\frac{m}{2}}\right) + 1 = S \left(\frac{m}{2}\right) + 1$$

Thus, $S(m) = S(\frac{m}{2}) + 1 = \Theta(\log m)$.

$$\implies T(U) = T(2^m) = S(m) = \Theta(\log m) = \Theta(\log \log U)$$
7.3 Van Emde Boas Trees

7.3.1 Take 1

Takeaways from our target recurrence $T(U) = T(\sqrt{U}) + 1$:

1. Different Universe Size structures at each level: $(U, \sqrt{U}, \sqrt[4]{U}, \ldots)$.
2. Single recursive call.
3. Constant run-time per recursive call.

Let $VEB(U) \equiv$ Van Emde Boas Tree for universe of size $U$.

![Diagram of VEB(U) tree](image)

**Figure 7.5: VEB(U): Van Emde Boas Tree of size $U$**

Insert($i$):

$B = A[\lfloor \frac{i}{\sqrt{U}} \rfloor]$

$B$.insert($i \mod \sqrt{U}$)

The delete and member operations can be done similarly.

It can be seen that the run-time of these operations can be represented by our target recurrence:

$T(U) = T(\sqrt{U}) + 1 = \Theta(\log \log U)$.

Next($i$):

$B = A[\lfloor \frac{i}{\sqrt{U}} \rfloor]$

$j = B$.next($i \mod \sqrt{U}$)

if $j \neq \text{nil}$

return $j + \lfloor \frac{i}{\sqrt{U}} \rfloor \times \sqrt{U}$

for $k = \lfloor \frac{i}{\sqrt{U}} \rfloor + 1, \ldots, \sqrt{U} - 1$ (*)

if $A[k].\text{size} \neq 0$

return $A[k].\text{next}(0) + k \times \sqrt{U}$

return $\text{nil}$

The run-time for the next and prev operations can be described by the following recurrence:

$T(U) = 2T(\sqrt{U}) + \sqrt{U}$
The per-recursive-call cost of $\sqrt{U}$ is due to the scan loop at $(\ast)$.

**Fix:** We can maintain another VEB($\sqrt{U}$) of entries $k$ of array $A$ such that $A[k].size \neq 0$. We will call this VEB Top. This allows us to re-write the next operation, replacing the scan with a next call to Top.

\[
\text{next}(i):
\begin{align*}
B &= A[\lfloor \frac{i}{\sqrt{U}} \rfloor] \\
 j &= B.\text{next}(i \mod \sqrt{U})
\end{align*}
\]

if $j \neq \text{nil}$
\[
\text{return } j + \lfloor \frac{i}{\sqrt{U}} \rfloor \ast \sqrt{U}
\]

$k = \text{Top.}\text{next}(\lfloor \frac{i}{\sqrt{U}} \rfloor + 1)$

if $k == \text{nil}$
\[
\text{return } \text{nil}
\]

\[
\text{return } A[k].\text{next}.0 + k \ast \sqrt{U}
\]

We analyze this fix formally below.

### 7.3.2 Take 2

![Diagram of VEB(U) with Top](image)

To fix the additive $\sqrt{U}$ term in the recursion, we avoid the linear search across $A$ by adding yet another VEB($\sqrt{U}$) for entries $k$ of array $A$ with $A[k].size \neq 0$ (called Top).

**Proof.** Substitution method again, $m := \log U$, $S(m) := T(2^m)$

\[
S(m) = 2S(m/2) + 1 = \Theta(m) = \Theta(\log U)
\]

But next($i$) is even worse now as it needs 3 recursive calls

\[
T(U) = 3T(\sqrt{U}) + 1 = \Theta((\log U)^{\log_2 3}) \equiv \Theta((\log U)^{1.58})
\]
Proof. As above, this time also relying on the master theorem. Alternatively, one could analyze the recursion tree
\[ S(m) = 3S(m/2) + 1 = \Theta(m^{\log_2 3}) \]

So how do we decrease the number of recursive calls here? We can maintain min and max fields to decrease the number of recursive calls.

7.3.3 For real

Let us start by implementing \texttt{next}(i) with \texttt{min} and \texttt{max}.

```plaintext
\textbf{next}(i):
\begin{align*}
    B &= A\lfloor i/\sqrt{U} \rfloor \\
    \text{if } B.\text{max} &\geq i \mod \sqrt{U} \quad \text{// Previously recursed here} \\
    \text{return } B.\text{next}(i \mod \sqrt{U}) + \lfloor i/\sqrt{U} \rfloor \sqrt{U} \\
    k &= \text{Top.next}(\lfloor i/\sqrt{U} \rfloor + 1) \\
    \text{if } k &\neq \text{nil} \\
    \text{return } A[k].\text{min} + k\sqrt{U} \\
    \text{return nil} \quad \text{// Previously } A[k].\text{next}(0)
\end{align*}
```

Only one recursive call now, either in \(A[i/\sqrt{U}]\) or \texttt{Top}.

\[ T(U) = T(\sqrt{U}) + 1 = \Theta(\log \log U) \]

What about insert? The introduction of \texttt{Top} led to 2 recursive calls - one to \(A[i/\sqrt{U}]\) and one to \texttt{Top} in case \(A[i/\sqrt{U}]\) was empty before.

Idea: Save recursive call in \(A[i/\sqrt{U}]\) if \(A[i/\sqrt{U}]\) was empty before, by not inserting \texttt{min/max} into recursive structures!

```plaintext
\textbf{insert}(i):
\begin{align*}
    \text{if } \text{size} &= 0 \\
    \text{min} &= \text{max} = i \\
    \text{size} &= 1 \\
    \text{return}
\end{align*}
```
if \( i < \text{min} \) 
    swap(i, min)
if \( i > \text{max} \) 
    swap(i, max)

\[
B = A[[i/\sqrt{U}]]
\]

\[
B.\text{insert}(i \mod \sqrt{U})
\]

if \( B.\text{size} == 1 \) 
    Top.\text{insert}([i/\sqrt{U}])
size = size +1

Note: When we add a new element to the data structure, we don’t insert the new element and \( \text{min}/\text{max} \) recursively - but only insert the new element (or old \( \text{min}/\text{max} \) in case this new element becomes \( \text{min}/\text{max} \)).

If \( B.\text{size} == 1 \) after \( B.\text{insert}(i \mod \sqrt{U}) \), then \( B.\text{insert}(i \mod \sqrt{U}) \) takes \( O(1) \) time.

\[
T(U) = T(\sqrt{U}) + O(1) = \Theta(\log \log U)
\]

delete(i) is symmetric.
find(i) also requires only one recursive call.

\[
\text{find}(i):
\]
\[
\text{if } i == \text{min} \text{ or } i == \text{max}
\]
\[
\text{return true}
\]
\[
B = A[[i/\sqrt{U}]]
\]
\[
\text{return } B.\text{find}(i \mod \sqrt{U})
\]

7.4 Summary

We saw an ordered dictionary with \( \Theta(\log \log U) \) time for all operations. Main takeaways:

- Design and analysis go hand in hand. To get \( \Theta(\log \log U) \) time we aimed for the right recurrence and fixed design along the way to get there.
- Be suspicious of assumptions of lower bounds. \( \Omega(n \log n) \) only applies to comparison based bounds.
- If you need some information often, make it easily accessible. For example \( \text{Top} \) allowed us to quickly find next non-empty recursive VEB(\( \sqrt{U} \)) to look at. Similarly, \( \text{min} \) (resp. \( \text{max} \)) saved us some recursive calls, viz. calls which find no larger element and calls intended to output \( \text{min} \) (resp. \( \text{max} \)).