Lecture 6
Van Emde Boas trees

Markus Jalsenius
Integer data structures

In the previous lectures we have used hashing to implement dictionaries that support the operations

- insert,
- delete,
- lookup (or a membership query if no satellite data).
Integer data structures

In the previous lectures we have used hashing to implement dictionaries that support the operations
- insert,
- delete,
- lookup (or a membership query if no satellite data).

In this lecture we will focus on a dictionary over the integer universe $U = \{0, \ldots, u - 1\}$. The operations are

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>insert(x)</code></td>
<td>Insert $x$.</td>
</tr>
<tr>
<td><code>delete(x)</code></td>
<td>Remove $x$.</td>
</tr>
<tr>
<td><code>member(x)</code></td>
<td>Returns <code>TRUE</code> if $x$ is in the dictionary, otherwise <code>FALSE</code>.</td>
</tr>
<tr>
<td><code>minimum</code></td>
<td>Returns the smallest element in the dictionary.</td>
</tr>
<tr>
<td><code>maximum</code></td>
<td>Returns the largest element in the dictionary.</td>
</tr>
<tr>
<td><code>successor(x)</code></td>
<td>Return the smallest $y$ in the dictionary such that $y &gt; x$.</td>
</tr>
<tr>
<td><code>predecessor(x)</code></td>
<td>Return the largest $y$ in the dictionary such that $y &lt; x$.</td>
</tr>
</tbody>
</table>
Deterministic

- Hashing does not allow operations like minimum and predecessor.
Deterministic

- Hashing does not allow operations like minimum and predecessor.
- Also, we want to operate deterministically.
  - No randomisation!
  - No errors!

Only...
We could use priority queues.

Recall, a priority queue is a data structure for maintaining a set of elements, each with an associated key – priority. Typically the following operations are efficient:

- \( \text{insert}(x, p) \) (insert element \( x \) which is associated with a key \( p \)).
- maximum or minimum (returns the element with the largest/smallest key),
- remove-maximum or remove-minimum (returns and removes the element with the largest/smallest key).
Priority queues

We could use *priority queues*. Recall, a priority queue is a data structure for maintaining a set of elements, each with an associated key – priority. Typically the following operations are efficient:

- **insert** \((x, p)\) (insert element \(x\) which is associated with a key \(p\)).
- **maximum or minimum** (returns the element with the largest/smallest key),
- **remove-maximum or remove-minimum** (returns and removes the element with the largest/smallest key).

Examples of data structures that implement priority queues are:

- Binary heaps.
- Red-black trees.
- Fibonacci heaps.
Priority queues

We could use *priority queues*.

Recall, a priority queue is a data structure for maintaining a set of elements, each with an associated key – priority. Typically the following operations are efficient:

- insert \((x, p)\) (insert element \(x\) which is associated with a key \(p\)).
- maximum or minimum (returns the element with the largest/smallest key),
- remove-maximum or remove-minimum (returns and removes the element with the largest/smallest key).

Examples of data structures that implement priority queues are:

- Binary heaps.
- Red-black trees.
- Fibonacci heaps.

**Observe**

These data structures base their decisions on comparing key values.
Priority queues

All these data structures must have an operation that takes $\Omega(\log n)$ time ($n$ is the number of inserted elements). Why?

Recall the meaning of $\Omega$. If you do not remember, look it up! See for example CLRS, page 48.
Priority queues

All these data structures must have an operation that takes $\Omega(\log n)$ time ($n$ is the number of inserted elements). Why?

Since they are comparison based, we could use a priority queue for comparison-based sorting;

insert all elements and then remove one by one using the remove-minimum operation.

Recall the meaning of $\Omega$. If you do not remember, look it up! See for example CLRS, page 48.
Priority queues

All these data structures must have an operation that takes $\Omega(\log n)$ time ($n$ is the number of inserted elements). Why?

Recall the meaning of $\Omega$. If you do not remember, look it up! See for example CLRS, page 48.

Since they are comparison based, we could use a priority queue for comparison-based sorting;

- insert all elements and then remove one by one using the remove-minimum operation.

Recall that $\Omega(n \log n)$ is a lower bound on the number of comparisons required when sorting $n$ elements. Thus, if say the remove-min operation was quicker than $\log n$, i.e. $o(\log n)$, we would be able to sort in time $o(n \log n)$. (See Section 8.1 of CLRS.)
Fast & furious

Today we will show how to solve all of the operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert((x))</td>
<td>Insert (x).</td>
</tr>
<tr>
<td>delete((x))</td>
<td>Remove (x).</td>
</tr>
<tr>
<td>member((x))</td>
<td>Returns \text{TRUE} if (x) is in the dictionary, otherwise \text{FALSE}.</td>
</tr>
<tr>
<td>minimum</td>
<td>Returns the smallest element in the dictionary.</td>
</tr>
<tr>
<td>maximum</td>
<td>Returns the largest element in the dictionary.</td>
</tr>
<tr>
<td>successor((x))</td>
<td>Return the smallest (y) in the dictionary such that (y &gt; x).</td>
</tr>
<tr>
<td>predecessor((x))</td>
<td>Return the largest (y) in the dictionary such that (y &lt; x).</td>
</tr>
</tbody>
</table>

in \(O(\log \log u)\) time (without randomisation) using \(O(u)\) space. Recall that the universe \(U = \{0, \ldots, u - 1\}\).
Today we will show how to solve all of the operations in $O(\log \log u)$ time (without randomisation) using $O(u)$ space. Recall that the universe $U = \{0, \ldots, u - 1\}$.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{insert}(x)$</td>
<td>Insert $x$.</td>
</tr>
<tr>
<td>$\text{delete}(x)$</td>
<td>Remove $x$.</td>
</tr>
<tr>
<td>$\text{member}(x)$</td>
<td>Returns TRUE if $x$ is in the dictionary, otherwise FALSE.</td>
</tr>
<tr>
<td>$\text{minimum}$</td>
<td>Returns the smallest element in the dictionary.</td>
</tr>
<tr>
<td>$\text{maximum}$</td>
<td>Returns the largest element in the dictionary.</td>
</tr>
<tr>
<td>$\text{successor}(x)$</td>
<td>Return the smallest $y$ in the dictionary such that $y &gt; x$.</td>
</tr>
<tr>
<td>$\text{predecessor}(x)$</td>
<td>Return the largest $y$ in the dictionary such that $y &lt; x$.</td>
</tr>
</tbody>
</table>

It can be shown that $\text{predecessor}(x)$ must in fact take $\Omega(\log \log u)$ time.

The solution will be given in the **RAM model**:  
- The memory is organised as words of $b$ bits.  
- A word can be accessed in $O(1)$ time.  
- The number of bits $b$ per word is $\Theta(\log u)$, i.e. large enough to hold the value of any element in the universe. If it were not, we would not be able to address all the data (i.e. there would not be enough number of bits in a word to store a pointer to a cell in the memory that holds the value of an element from the universe).
Computational model

- The solution will be given in the **RAM model**:
  - The memory is organised as words of $b$ bits.
  - A word can be accessed in $O(1)$ time.
  - The number of bits $b$ per word is $\Theta(\log u)$, i.e. large enough to hold the value of any element in the universe.
  If it were not, we would not be able to address all the data (i.e. there would not be enough number of bits in a word to store a pointer to a cell in the memory that holds the value of an element from the universe).

- We also assume that *standard* word operations take $O(1)$ time, e.g. addition, subtraction, multiplication, division, shifts and bitwise operations. (Think normal C programming.)
The solution will be given in the **RAM model**:

- The memory is organised as words of \( b \) bits.
- A word can be accessed in \( O(1) \) time.
- The number of bits \( b \) per word is \( \Theta(\log u) \), i.e. large enough to hold the value of any element in the universe. If it were not, we would not be able to address all the data (i.e. there would not be enough number of bits in a word to store a pointer to a cell in the memory that holds the value of an element from the universe).

We also assume that *standard* word operations take \( O(1) \) time, e.g. addition, subtraction, multiplication, division, shifts and bitwise operations. (Think normal C programming.)

In the RAM model we can beat the comparison-based lower bound of \( \Omega(n \log n) \) since we now operate on words.
Direct addressing

- Superimpose a binary tree over a bit array $A$ of size $u$.
- Element $A[i]$ is set to 1 iff the key $i$ is in the dictionary.
- An internal node is 1 if any of its children is 1.

Array $A$ of size $u$ ($u = 16$ here)

$$
\begin{array}{cccccccccccccccc}
0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15
\end{array}
$$

Height $\log u$
Direct addressing

- Superimpose a binary tree over a bit array $A$ of size $u$.
- Element $A[i]$ is set to 1 iff the key $i$ is in the dictionary.
- An internal node is 1 if any of its children is 1.

Array $A$ of size $u$

(u = 16 here)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Operation | Only array
--- | ---
insert($x$) | $O(1)$
delete($x$) | $O(1)$
member($x$) | $O(1)$
minimum/maximum | $O(u)$
successor($x$)/predecessor($x$) | $O(u)$
Direct addressing

- Superimpose a binary tree over a bit array $A$ of size $u$.
- Element $A[i]$ is set to 1 iff the key $i$ is in the dictionary.
- An internal node is 1 if any of its children is 1.

Array $A$ of size $u$

(height \( \log u \))

<table>
<thead>
<tr>
<th>Operation</th>
<th>Only array</th>
<th>Array with tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert($x$)</td>
<td>$O(1)$</td>
<td>$O(\log u)$</td>
</tr>
<tr>
<td>delete($x$)</td>
<td>$O(1)$</td>
<td>$O(\log u)$</td>
</tr>
<tr>
<td>member($x$)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>minimum/maximum</td>
<td>$O(u)$</td>
<td>$O(\log u)$</td>
</tr>
<tr>
<td>successor($x$)/predecessor($x$)</td>
<td>$O(u)$</td>
<td>$O(\log u)$</td>
</tr>
</tbody>
</table>
Direct addressing

- Superimpose a binary tree over a bit array $A$ of size $u$.
- Element $A[i]$ is set to 1 iff the key $i$ is in the dictionary.
- An internal node is 1 if any of its children is 1.

Array $A$ of size $u$
($u = 16$ here)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Only array</th>
<th>Array with tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert($x$)</td>
<td>$O(1)$</td>
<td>$O(\log u)$</td>
</tr>
<tr>
<td>delete($x$)</td>
<td>$O(1)$</td>
<td>$O(\log u)$</td>
</tr>
<tr>
<td>member($x$)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>minimum/maximum</td>
<td>$O(u)$</td>
<td>$O(\log u)$</td>
</tr>
<tr>
<td>successor($x$)</td>
<td>$O(u)$</td>
<td>$O(\log u)$</td>
</tr>
<tr>
<td>predecessor($x$)</td>
<td>$O(u)$</td>
<td>$O(\log u)$</td>
</tr>
</tbody>
</table>

Example: predecessor(14) gives 7: walk up, then down the tree.
Direct addressing

- Superimpose a binary tree over a bit array \( A \) of size \( u \).
- Element \( A[i] \) is set to 1 iff the key \( i \) is in the dictionary.
- An internal node is 1 if any of its children is 1.

**Example**

predecessor(14) gives 7: walk up, then down the tree.

**Observe**

If \( n \) is much smaller than \( u \), we could use a red-black tree: all operations take \( O(\log n) \) time.

Array \( A \) of size \( u \)

\[ u = 16 \text{ here} \]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Only array</th>
<th>Array with tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert(( x ))</td>
<td>( O(1) )</td>
<td>( O(\log u) )</td>
</tr>
<tr>
<td>delete(( x ))</td>
<td>( O(1) )</td>
<td>( O(\log u) )</td>
</tr>
<tr>
<td>member(( x ))</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>minimum/maximum</td>
<td>( O(u) )</td>
<td>( O(\log u) )</td>
</tr>
<tr>
<td>successor(( x ))/predecessor(( x ))</td>
<td>( O(u) )</td>
<td>( O(\log u) )</td>
</tr>
</tbody>
</table>
The bit array $A$ of size $u$ is partitioned into $\sqrt{u}$ blocks.
Each block has size $\sqrt{u}$ and are numbered from 0 to $\sqrt{u} - 1$.
There is another bit array $S$, the summary, of size $\sqrt{u}$.
Element $S[i]$ is set to 1 iff the block $i$ in $A$ contains at least one 1.

$(u = 16$ here) $A$

$S$

height 1
**Constant height tree**

- The bit array $A$ of size $u$ is partitioned into $\sqrt{u}$ blocks.
- Each block has size $\sqrt{u}$ and are numbered from 0 to $\sqrt{u} - 1$.
- There is another bit array $S$, the *summary*, of size $\sqrt{u}$.
- Element $S[i]$ is set to 1 iff the block $i$ in $A$ contains at least one 1.

---

### Operation | Time | Comment
--- | --- | ---
insert($x$) | $O(1)$ | Also set bit in $S$ if necessary.
delete($x$) | $O(\sqrt{u})$ | Check the block if it was the last 1.
member($x$) | $O(1)$ | Read off the bit in $A$.
minimum/maximum | $O(\sqrt{u})$ | Do min/max query on $S$, then its block.
successor($x$)/predecessor($x$) | $O(\sqrt{u})$ | Search the block, then $S$, then another block.
The bit array $A$ of size $u$ is partitioned into $\sqrt{u}$ blocks. Each block has size $\sqrt{u}$ and are numbered from 0 to $\sqrt{u} - 1$. There is another bit array $S$, the summary, of size $\sqrt{u}$. Element $S[i]$ is set to 1 iff the block $i$ in $A$ contains at least one 1.

$$u = 16$$ here

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert($x$)</td>
<td>$O(1)$</td>
<td>Also set bit in $S$ if necessary.</td>
</tr>
<tr>
<td>delete($x$)</td>
<td>$O(\sqrt{u})$</td>
<td>Check the block if it was the last 1.</td>
</tr>
<tr>
<td>member($x$)</td>
<td>$O(1)$</td>
<td>Read off the bit in $A$.</td>
</tr>
<tr>
<td>minimum/maximum</td>
<td>$O(\sqrt{u})$</td>
<td>Do min/max query on $S$, then its block.</td>
</tr>
<tr>
<td>successor($x$)/predecessor($x$)</td>
<td>$O(\sqrt{u})$</td>
<td>Search the block, then $S$, then another block.</td>
</tr>
</tbody>
</table>
Constant height tree

- The bit array $A$ of size $u$ is partitioned into $\sqrt{u}$ blocks.
- Each block has size $\sqrt{u}$ and are numbered from 0 to $\sqrt{u} - 1$.
- There is another bit array $S$, the *summary*, of size $\sqrt{u}$.
- Element $S[i]$ is set to 1 iff the block $i$ in $A$ contains at least one 1.

$(u = 16$ here$) \quad A$

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert$(x)$</td>
<td>$O(1)$</td>
<td>Also set bit in $S$ if necessary.</td>
</tr>
<tr>
<td>delete$(x)$</td>
<td>$O(\sqrt{u})$</td>
<td>Check the block if it was the last 1.</td>
</tr>
<tr>
<td>member$(x)$</td>
<td>$O(1)$</td>
<td>Read off the bit in $A$.</td>
</tr>
<tr>
<td>minimum/maximum</td>
<td>$O(\sqrt{u})$</td>
<td>Do min/max query on $S$, then its block.</td>
</tr>
<tr>
<td>successor$(x)$</td>
<td>$O(\sqrt{u})$</td>
<td>Search the block, then $S$, then another block.</td>
</tr>
</tbody>
</table>
Van Emde Boas Trees

- Recursively define the nodes of the **van Emde Boas tree** (vEB tree).
- We write $vEB(u)$ to denote the vEB tree on a universe of size $u$. 
Van Emde Boas Trees

- Recursively define the nodes of the **van Emde Boas tree** (vEB tree).
- We write $vEB(u)$ to denote the vEB tree on a universe of size $u$.

**Base case:** universe size $u = 2$, i.e. $U = \{0, 1\}$.

$vEB(2)$ has only one node that contains two variables:
- $\ast$ min, containing the minimum element.
- $\ast$ max, containing the maximum element.

---

**Example**

- If $\text{min} = \text{max} = 1$ then only element 1 is in the dictionary.
- If $\text{min} = 0$ and $\text{max} = 1$ then both 0 and 1 have been inserted.
- If $\text{min} = \text{max} = \text{NULL}$ then no element is in the dictionary.
Van Emde Boas Trees

**Recursive step:** the universe size \( u > 2 \), i.e. \( U = \{0, \ldots, u-1\} \).

Before we define \( vEB(u) \), we define

\[
\begin{align*}
  &\sqrt[\uparrow]{u} = 2^\left\lceil \frac{\log_2 u}{2} \right\rceil \\
  &\sqrt[\downarrow]{u} = 2^\left\lfloor \frac{\log_2 u}{2} \right\rfloor
\end{align*}
\]

We assume \( u \) is a power of 2.

**Example**

\[
\begin{align*}
u &= 32 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \\
\sqrt[\uparrow]{u} &= 8 \\
\sqrt[\downarrow]{u} &= 4
\end{align*}
\]

\( \sqrt[\uparrow]{u} \approx 5.7 \) (not an integer)
Van Emde Boas Trees

**Recursive step:** the universe size $u > 2$, i.e. $U = \{0, \ldots, u-1\}$.

Before we define $\text{vEB}(u)$, we define

- $\lceil \sqrt[\log_2 u]{u} \rceil = 2^{\lceil (\log_2 u)/2 \rceil}$
- $\lfloor \sqrt[\log_2 u]{u} \rfloor = 2^{\lfloor (\log_2 u)/2 \rfloor}$.

We assume $u$ is a power of 2.

**Example**

$u = 32 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$

$\sqrt[\log_2 u]{u} \approx 5.7$ (not an integer)

$\lceil \sqrt[\log_2 u]{u} \rceil = 8$

$\lfloor \sqrt[\log_2 u]{u} \rfloor = 4$

The *root* of $\text{vEB}(u)$ contains the following variables:

- min and max (the min/max elements).
- A pointer to the summary, which itself is a $\text{vEB}(\sqrt[\log_2 u]{u})$ tree.
- An array of size $\sqrt[\log_2 u]{u}$ that contains pointers to the $\sqrt[\log_2 u]{u}$ blocks, each of which is a $\text{vEB}(\sqrt[\log_2 u]{u})$ tree.
Example

Suppose for a second that we do not use recursion. Then we can use the previous constant height tree example.

The root of $\text{vEB}(16)$ contains:

- $\text{min} = 2$
- $\text{max} = 15$
- Summary pointer
- Block pointers

$(u = 16$ here)$

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

- Suppose for a second that we do not use recursion. Then we can use the previous constant height tree example.

The root of vEB(16) contains:

- \( \min = 2 \)
- \( \max = 15 \)

- Summary pointer
- Block pointers

Take \( S \) for example. On \( S \) we perform the same operations as on the dictionary of size \( u \) (a predecessor call in the previous example).

- \( S \) only has size \( \sqrt[4]{u} \) (here 4).

- So why not use a vEB(4) tree for \( S \) instead of a simple array?
Suppose for a second that we do not use recursion. Then we can use the previous constant height tree example.

The root of \( \text{vEB}(16) \) contains:

- \( \min = 2 \)
- \( \max = 15 \)

Take \( S \) for example. On \( S \) we perform the same operations as on the dictionary of size \( u \) (a predecessor call in the previous example).

\( S \) only has size \( \sqrt[4]{u} \) (here 4).

So why not use a \( \text{vEB}(4) \) tree for \( S \) instead of a simple array?

We can apply the same reasoning on each block as well.
Van Emde Boas Trees

**Example**

The root of vEB(16) contains:

- **Summary of S**
  - S: 1 1 0 1
  - Summary of S: 1 1

- **Block pointers**
  - min = 0, max = 3
  - Summary pointer
  - Block pointers

- **Subtrees**
  - min = 0, max = 1
    - 1 1
  - min = 0, max = 1
    - 1 1
  - min = 1, max = 1
    - 0 1
STOP!!! There is a caveat! The smallest element does not propagate down the tree.

Summary of $S$

$S$

The root of vEB(16) contains:

$\min = 2 \quad \max = 15$

Summary pointer

Block pointers

$\min = 0 \quad \max = 3$

Summary pointer

Block pointers

$\min = 0 \quad \max = 1$

$\min = 0 \quad \max = 1$

$\min = 1 \quad \max = 1$
Example

STOP!!! There is a caveat! The smallest element does not propagate down the tree.

Summary of $S$

\[
\begin{array}{c|c|c|c}
0 & 1 & 2 & 3 \\
\hline
1 & 1 & 0 & 1
\end{array}
\]

$S$

\[
\begin{array}{c|c|c|c}
0 & 1 & 2 & 3 \\
\hline
1 & 1 & 0 & 1
\end{array}
\]

The root of $vEB(16)$ contains:

- $\text{min} = 2$  $\text{max} = 15$
- Summary pointer
- Block pointers

\[
\begin{array}{c|c|c|c}
\text{min} = 2 & \text{max} = 15 \\
\hline
\text{Summary pointer} & \text{Block pointers}
\end{array}
\]

Minimum element is already stored in $\text{min}$.

STOP!!! There is a caveat! The smallest element does not propagate down the tree.
STOP!!! There is a caveat! The smallest element does not propagate down the tree.

The root of vEB\((16)\) contains:

- **Summary of** \(S\)
  - \(S = \begin{bmatrix} 1 & 1 & 0 & 1 \end{bmatrix}\)
  - **Summary of** \(S\)
  - \(\begin{bmatrix} 1 & 1 \end{bmatrix}\)

- **Summary of** \(S\)
  - \(\begin{bmatrix} 1 & 1 \end{bmatrix}\)

- **Summary of** \(S\)
  - \(\begin{bmatrix} 1 & 1 \end{bmatrix}\)

- **Summary of** \(S\)
  - \(\begin{bmatrix} 0 & 1 \end{bmatrix}\)

- **Summary of** \(S\)
  - \(\begin{bmatrix} 1 & 1 \end{bmatrix}\)

- **Summary of** \(S\)
  - \(\begin{bmatrix} 0 & 1 \end{bmatrix}\)

- **Summary of** \(S\)
  - \(\begin{bmatrix} 1 & 1 \end{bmatrix}\)

- **Summary of** \(S\)
  - \(\begin{bmatrix} 0 & 1 \end{bmatrix}\)

- **Summary of** \(S\)
  - \(\begin{bmatrix} 1 & 1 \end{bmatrix}\)

- **Summary of** \(S\)
  - \(\begin{bmatrix} 0 & 1 \end{bmatrix}\)

The smallest element does not propagate down the tree. The 1 does not propagate. It becomes 0.
The smallest element does not propagate

**Example**

The root of vEB(16) contains:

- $\text{min} = 2$
- $\text{max} = 15$

Summary pointer

Block pointers

Going back to the first example, the vEB(16) has a pointer to the vEB(4) tree representing the first block, which does not contain element 2, only element 3.
The smallest element does not propagate

**Example**

**Observe**
If the block becomes empty, the summary bit is set to 0. Here element 3 is still in.

The root of vEB(16) contains:

- min = 2
- max = 15

Summary pointer

Block pointers

Going back to the first example, the vEB(16) has a pointer to the vEB(4) tree representing the first block, which does not contain element 2, only element 3.
Suppose we want the maximum element of the second block.

We recursively use the maximum operator on the vEB(4).

The answer returned here is 3.

Now, this element corresponds to element 7 in the array of 16 elements.

Thus, when calling smaller vEB trees recursively, we need to translate the answers to match the corresponding index at the level of the tree we are calling from.
Indexing elements – a technicality

Let \( x \) be an element.

- Define
  
  \[
  \text{high}(x) = \left\lfloor \frac{x}{\sqrt{u}} \right\rfloor \\
  \text{low}(x) = x \mod \sqrt{u}
  \]

- \( \text{high}(x) \) is the block that contains \( x \).
- \( \text{low}(x) \) is the relative position of \( x \) within this block.
- Both \( \text{high}(x) \) and \( \text{low}(x) \) can be computed in constant time.

**Example**

- \( u = 16 \).
- \( x = 9 \).
- \( \sqrt{u} = 4 \).
- \( \text{high}(x) = \left\lfloor \frac{9}{4} \right\rfloor = 2 \).
- \( \text{low}(x) = 9 \mod 4 = 1 \).
Operations on vEB trees

- minimum/maximum:
  Simply return the value of \( \min/\max \). Done!
This takes \( O(1) \) time. Easy peasy!
Operations on vEB trees

- predecessor($x$):
  1. If $u = 2$, check min and max and return appropriately. **Done!**
  2. If $x > \text{max}$, return max. **Done!**
  3. Let $y = \text{high}(x)$ be the block containing $x$.
     If the min element in $y$ is less than $x$ (and not NULL), run predecessor(low($x$)) on $y$. **Done!**
  4. a) If not, run predecessor(high($x$)) on the summary vEB.
     b) If say block $z$ is returned, return the max element of $z$. **Done!**
     c) If NULL is return (no smaller element), return NULL. **Done!**
Operations on vEB trees

- **predecessor**($x$):
  1. If $u = 2$, check min and max and return appropriately. **Done!**
  2. If $x > \max$, return max. **Done!**
  3. Let $y = \text{high}(x)$ be the block containing $x$. If the min element in $y$ is less than $x$ (and not NULL), run predecessor(low($x$)) on $y$. **Done!**
  4. a) If not, run predecessor(high($x$)) on the summary vEB.
     b) If say block $z$ is returned, return the max element of $z$. **Done!**
     c) If NULL is return (no smaller element), return NULL. **Done!**

**Observe**
We can call predecessor with an index $x$ that is not necessarily in the dictionary, e.g., we can ask for the largest element less than 57 even if element 57 is not in the dictionary itself.
Operations on vEB trees

\textbf{predecessor}(x):

1. If \( u = 2 \), check min and max and return appropriately. \textbf{Done!}
2. If \( x > \text{max} \), return \text{max}. \textbf{Done!}
3. Let \( y = \text{high}(x) \) be the block containing \( x \).
   If the min element in \( y \) is less than \( x \) (and not NULL), run \predecessor(\text{low}(x)) \text{ on } y. \textbf{Done!}
4. a) If not, run \predecessor(\text{high}(x)) \text{ on the summary vEB.}
   b) If say block \( z \) is returned, return the max element of \( z \). \textbf{Done!}
   c) If NULL is return (no smaller element), return NULL. \textbf{Done!}

\textbf{Time analysis:} suppose the running time for predecessor is \( T(u) \).

\begin{itemize}
  \item \textbf{Steps 1 and 2} take \( O(1) \) time.
  \item Step 3 takes \( T(\sqrt[4]{u}) \) time.
  \item Step 4a takes \( T(\sqrt[4]{u}) \) time.
  \item Steps 4b and 4c take \( O(1) \) time.
\end{itemize}

Thus, total time is \( T(u) \leq T(\sqrt[4]{u}) + O(1) \).

\textbf{Observe:}

We can call \textbf{predecessor} with an index \( x \) that is not necessarily in the dictionary, e.g., we can ask for the largest element less than 57 even if element 57 is not in the dictionary itself.
Operations on vEB trees

- predecessor($x$):
  1. If $u = 2$, check min and max and return appropriately. **Done!**
  2. If $x > \text{max}$, return max. **Done!**
  3. Let $y = \text{high}(x)$ be the block containing $x$.
     If the min element in $y$ is less than $x$ (and not NULL), run predecessor(low($x$)) on $y$. **Done!**
  4. a) If not, run predecessor(high($x$)) on the summary vEB.
     b) If say block $z$ is returned, return the max element of $z$. **Done!**
     c) If NULL is return (no smaller element), return NULL. **Done!**

- Time analysis: suppose the running time for predecessor is $T(u)$.
  - Steps 1 and 2 take $O(1)$ time.
  - Step 3 takes $T(\sqrt[3]{u})$ time.
  - Step 4a takes $T(\sqrt[3]{u})$ time.
  - Steps 4b and 4c take $O(1)$ time.

Thus, total time is $T(u) \leq T(\sqrt[3]{u}) + O(1)$.
- Solving this recurrence gives $T(u) = O(\log \log u)$. 

**Observe**
We can call predecessor with an index $x$ that is not necessarily in the dictionary, e.g., we can ask for the largest element less than 57 even if element 57 is not in the dictionary itself.
Operations on vEB trees

- **insert**(x):
  1. If the dictionary is empty, set min and max to x. **Done!**
  2. If \( x < \min \), swap \( x \) and \( \min \). Now \( x \) is second smallest.
  3. Let \( y = \text{high}(x) \) be the block containing \( x \).
     - If block \( y \) is empty, insert \( y \) into the summary and insert \( x \) in \( y \).
  4. Otherwise, only insert \( x \) into block \( y \) (no need to touch the summary).
  5. Update max is necessary, i.e. if \( x > \max \) then \( \max \leftarrow x \). **Done!**
Operations on vEB trees

**insert**\(x\):

1. If the dictionary is empty, set \(\text{min}\) and \(\text{max}\) to \(x\). **Done!**
2. If \(x < \text{min}\), swap \(x\) and \(\text{min}\). Now \(x\) is second smallest.
3. Let \(y = \text{high}(x)\) be the block containing \(x\).
   If block \(y\) is empty, insert \(y\) into the summary and insert \(x\) in \(y\).
4. Otherwise, only insert \(x\) into block \(y\) (no need to touch the summary).
5. Update \(\text{max}\) is necessary, i.e. if \(x > \text{max}\) then \(\text{max} \leftarrow x\). **Done!**

**Time analysis:** suppose the running time for insert is \(T(u)\).

- Steps 1 takes \(O(1)\) time.
- Step 2 takes \(O(1)\) time.
- Step 3 takes \(T(\sqrt[3]{u})\) time (inserting \(y\) into the summary). Inserting \(x\) into the empty \(y\) takes \(O(1)\) time because of step 1.
- Step 4 takes \(T(\sqrt[3]{u})\) time.
- Steps 5 takes \(O(1)\) time.

Again, \(T(u) \leq T(\sqrt[3]{u}) + O(1)\), implying \(T(u) = O(\log \log u)\).
Space usage

- The other operations are left as an exercise.
Space usage

- The other operations are left as an exercise.
- How much space does the vEB tree use?
Space usage

The other operations are left as an exercise.

How much space does the vEB tree use?

Let $Z(u)$ be the size of a vEB tree with universe of size $u$.

- The root contains $\lceil \sqrt{u} \rceil + 1 = O(\sqrt{u})$ pointers.
- Each tree has size (roughly) $Z(\sqrt{u})$.
- Thus, total size is $Z(u) \leq \sqrt{u} \cdot Z(\sqrt{u}) + O(1)$.
- Solving this recurrence gives $Z(u) = O(u)$.

That is, the space usage of the whole vEB tree is linear in $u$. 