van Emde Boas trees

Benjamin Sach
Dictionaries

In a dynamic dictionary data structure we store \((\text{key}, \text{value})\)-pairs such that for any key there is at most one pair \((\text{key}, \text{value})\) in the dictionary.

Three operations are supported:

- \(\text{add}(x, v)\) Add the the pair \((x, v)\) where \(x \in U\), the universe.
- \(\text{lookup}(x)\) Return \(v\) if \((x, v)\) is in dictionary, or \text{NULL} otherwise.
- \(\text{delete}(x)\) Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

*In previous lectures we have focussed on solutions using Hashing*
In a **dynamic dictionary** data structure we store \((key, value)\)-pairs such that for any \(key\) there is at most one pair \((key, value)\) in the dictionary.

Three operations are supported:

- **add**\((x, v)\)**  
  Add the pair \((x, v)\) where \(x \in U\), the universe

- **lookup**\((x)\)**  
  Return \(v\) if \((x, v)\) is in dictionary, or **NULL** otherwise.

- **delete**\((x)\)**  
  Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

*In previous lectures we have focussed on solutions using **Hashing** in particular.*
Dictionaries

In a **dynamic dictionary** data structure we store \((key, value)\)-pairs such that for any \(key\) there is at most one pair \((key, value)\) in the dictionary.

Three operations are supported:

- **add** \((x, v)\) Add the pair \((x, v)\) where \(x \in U\), the universe
- **lookup** \((x)\) Return \(v\) if \((x, v)\) is in dictionary, or **NULL** otherwise.
- **delete** \((x)\) Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

*In previous lectures we have focussed on solutions using **Hashing** in particular...*

**Theorem**

In the **Cuckoo hashing** scheme:

- Every **lookup** and every **delete** takes \(O(1)\) **worst-case** time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An **insert** takes **amortised expected** \(O(1)\) time
Dictionaries

In a **dynamic dictionary** data structure we store \((\text{key}, \text{value})\)-pairs such that for any key there is at most one pair \((\text{key}, \text{value})\) in the dictionary.

Three operations are supported:

- **add**\((x, v)\) Add the the pair \((x, v)\) where \(x \in U\), the **universe**
- **lookup**\((x)\) Return \(v\) if \((x, v)\) is in dictionary, or **NULL** otherwise.
- **delete**\((x)\) Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

*In previous lectures we have focussed on solutions using Hashing in particular.*

**Theorem**

In the **Cuckoo hashing** scheme:

- Every **lookup** and every **delete** takes \(O(1)\) **worst-case** time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An **insert** takes **amortised expected** \(O(1)\) time

What’s not to like?
Dictionaries

In a **dynamic dictionary** data structure we store *(key, value)*-pairs such that for any *key* there is at most one pair *(key, value)* in the dictionary.

Three operations are supported:

- **add**(*x, v*) Add the the pair *(x, v)* where *x* ∈ *U*, the *universe*
- **lookup**(*x*) Return *v* if *(x, v)* is in dictionary, or *NULL* otherwise.
- **delete**(*x*) Remove pair *(x, v)* (assuming *(x, v)* is in the dictionary).

*In previous lectures we have focussed on solutions using Hashing in particular...*

**Theorem**

In the **Cuckoo hashing** scheme:
- Every **lookup** and every **delete** takes *O(1)* worst-case time,
- The space is *O(n)* where *n* is the number of keys stored
- An **insert** takes *amortised expected* *O(1)* time

What’s not to like? Except the randomness,
Dictionaries

In a **dynamic dictionary** data structure we store \((key, value)\)-pairs such that for any \(key\) there is at most one pair \((key, value)\) in the dictionary.

Three operations are supported:

- **add\((x, v)\)**: Add the pair \((x, v)\) where \(x \in U\), the **universe**.
- **lookup\((x)\)**: Return \(v\) if \((x, v)\) is in dictionary, or **NULL** otherwise.
- **delete\((x)\)**: Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

*In previous lectures we have focussed on solutions using **Hashing** in particular...*

---

**Theorem**

In the **Cuckoo hashing** scheme:

- Every **lookup** and every **delete** takes \(O(1)\) **worst-case** time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An **insert** takes **amortised expected** \(O(1)\) time

What’s not to like? Except the randomness, the amortisation,
Dictionaries

In a **dynamic dictionary** data structure we store \((key, value)\)-pairs such that for any \(key\) there is at most one pair \((key, value)\) in the dictionary.

Three operations are supported:

- **add\((x, v)\)**: Add the the pair \((x, v)\) where \(x \in U\), the universe
- **lookup\((x)\)**: Return \(v\) if \((x, v)\) is in dictionary, or \texttt{NULL}\ otherwise.
- **delete\((x)\)**: Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

*In previous lectures we have focussed on solutions using Hashing in particular.*

**THEOREM**

In the **Cuckoo hashing** scheme:

- Every **lookup** and every **delete** takes \(O(1)\) worst-case time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An **insert** takes amortised expected \(O(1)\) time

What’s not to like? Except the randomness, the amortisation, and the inflexibility
Dictionaries

In a **dynamic dictionary** data structure we store \((key, value)\)-pairs such that for any \(key\) there is at most one pair \((key, value)\) in the dictionary.

Three operations are supported:

- **add\((x, v)\)**: Add the the pair \((x, v)\) where \(x \in U\), the *universe*
- **lookup\((x)\)**: Return \(v\) if \((x, v)\) is in dictionary, or **NULL** otherwise.
- **delete\((x)\)**: Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

*In previous lectures we have focussed on solutions using **Hashing** in particular.*

---

**THEOREM**

In the **Cuckoo hashing** scheme:

- Every **lookup** and every **delete** takes \(O(1)\) **worst-case** time,
- The space is \(O(n)\) where \(n\) is the number of keys stored
- An **insert** takes **amortised expected** \(O(1)\) time

---

**What's not to like?** Except the randomness, the amortisation, and the inflexibility
Supporting more operations

In a dynamic dictionary data structure we store \((key, value)\)-pairs such that for any key there is at most one pair \((key, value)\) in the dictionary.

Three operations are supported:

- **add**\((x, v)\)  Add the pair \((x, v)\) where \(x \in U\) - the universe
- **lookup**\((x)\)  Return \(v\) if \((x, v)\) is in dictionary, or **NULL** otherwise.
- **delete**\((x)\)  Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).
Supporting more operations

In a **dynamic dictionary** data structure we store \((key, value)\)-pairs such that for any \(key\) there is at most one pair \((key, value)\) in the dictionary.

Three operations are supported:

- \(\text{add}(x, v)\) Add the pair \((x, v)\) where \(x \in U\) - the universe
- \(\text{lookup}(x)\) Return \(v\) if \((x, v)\) is in dictionary, or \(\text{NULL}\) otherwise.
- \(\text{delete}(x)\) Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

What happens if we add more operations?
Supporting more operations

In a dynamic dictionary data structure we store (key, value)-pairs such that for any key there is at most one pair (key, value) in the dictionary.

Three operations are supported:

- **add**\((x, v)\) Add the pair \((x, v)\) where \(x \in U\) - the universe
- **lookup**\((x)\) Return \(v\) if \((x, v)\) is in dictionary, or **NULL** otherwise.
- **delete**\((x)\) Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

What happens if we add more operations?

We also want our data structure to support:

- **predecessor**\((k)\) - returns the (unique) element \((x, v)\) in the dictionary with the largest key, \(x\) such that \(x \leq k\)
Supporting more operations

In a **dynamic dictionary** data structure we store \((key, value)\)-pairs such that for any \(key\) there is at most one pair \((key, value)\) in the dictionary.

Three operations are supported:

- \(\text{add}(x, v)\) - Add the pair \((x, v)\) where \(x \in U\) - the *universe*
- \(\text{lookup}(x)\) - Return \(v\) if \((x, v)\) is in dictionary, or \(\text{NULL}\) otherwise.
- \(\text{delete}(x)\) - Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

What happens if we add more operations?

We also want our data structure to support:

- \(\text{predecessor}(k)\) - returns the (unique) element \((x, v)\) in the dictionary with the largest key, \(x\) such that \(x \leq k\)
Supporting more operations

In a dynamic dictionary data structure we store \((key, value)\)-pairs such that for any \(key\) there is at most one pair \((key, value)\) in the dictionary.

Three operations are supported:

- \(\text{add}(x, v)\) Add the pair \((x, v)\) where \(x \in U\) - the universe
- \(\text{lookup}(x)\) Return \(v\) if \((x, v)\) is in dictionary, or \(\text{NULL}\) otherwise.
- \(\text{delete}(x)\) Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

What happens if we add more operations?

We also want our data structure to support:

- \(\text{predecessor}(k)\) - returns the (unique) element \((x, v)\) in the dictionary with the largest key, \(x\) such that \(x \leq k\)
Supporting more operations

In a **dynamic dictionary** data structure we store \((key, value)\)-pairs such that for any \(key\) there is at most one pair \((key, value)\) in the dictionary.

Three operations are supported:

- **add\((x, v)\)** Add the the pair \((x, v)\) where \(x \in U\) - the **universe**
- **lookup\((x)\)** Return \(v\) if \((x, v)\) is in dictionary, or **NULL** otherwise.
- **delete\((x)\)** Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

What happens if we add more operations?

We also want our data structure to support:

**predecessor\((k)\)** - returns the (unique) element \((x, v)\) in the dictionary with the largest key, \(x\) such that \(x \leq k\)

![Diagram showing keys in the dictionary and their relationship to the universe](image-url)
Supporting more operations

In a **dynamic dictionary** data structure we store *(key, value)*-pairs such that for any key there is at most one pair *(key, value)* in the dictionary.

Three operations are supported:

- `add(x, v)` Add the pair *(x, v)* where \( x \in U \) - the universe
- `lookup(x)` Return \( v \) if *(x, v)* is in dictionary, or `NULL` otherwise.
- `delete(x)` Remove pair *(x, v)* (assuming *(x, v)* is in the dictionary).

What happens if we add more operations?

We also want our data structure to support:

`predecessor(k)` - returns the (unique) element *(x, v)* in the dictionary
with the largest key, \( x \) such that \( x \leq k \)
Supporting more operations

In a **dynamic dictionary** data structure we store \((key, value)\)-pairs such that for any \(key\) there is at most one pair \((key, value)\) in the dictionary.

Three operations are supported:

- \(add(x, v)\): Add the pair \((x, v)\) where \(x \in U\) - the universe
- \(lookup(x)\): Return \(v\) if \((x, v)\) is in dictionary, or \(NULL\) otherwise.
- \(delete(x)\): Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

What happens if we add more operations?

We also want our data structure to support:

- \(predecessor(k)\) - returns the (unique) element \((x, v)\) in the dictionary with the largest key, \(x\) such that \(x \leq k\).

\[\begin{array}{c}
\text{k} \\
| \\
\end{array}\]
Supporting more operations

In a dynamic dictionary data structure we store \((key, value)\)-pairs such that for any key there is at most one pair \((key, value)\) in the dictionary.

Three operations are supported:

- \(\text{add}(x, v)\): Add the pair \((x, v)\) where \(x \in U\) - the universe
- \(\text{lookup}(x)\): Return \(v\) if \((x, v)\) is in dictionary, or \(\text{NULL}\) otherwise.
- \(\text{delete}(x)\): Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

What happens if we add more operations?

We also want our data structure to support:

- \(\text{predecessor}(k)\) - returns the (unique) element \((x, v)\) in the dictionary with the largest key, \(x\) such that \(x \leq k\)
Supporting more operations

In a **dynamic dictionary** data structure we store \((\text{key}, \text{value})\)-pairs such that for any key there is at most one pair \((\text{key}, \text{value})\) in the dictionary.

Three operations are supported:

- \(\text{add}(x, v)\) Add the the pair \((x, v)\) where \(x \in U\) - the universe
- \(\text{lookup}(x)\) Return \(v\) if \((x, v)\) is in dictionary, or \(\text{NULL}\) otherwise.
- \(\text{delete}(x)\) Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

What happens if we add more operations?

We also want our data structure to support:

- \(\text{predecessor}(k)\) - returns the (unique) element \((x, v)\) in the dictionary with the largest key, \(x\) such that \(x \leq k\)
- \(\text{successor}(k)\) - returns the (unique) element \((x, v)\) in the dictionary with the smallest key, \(x\) such that \(x \geq k\)
Supporting more operations

In a **dynamic dictionary** data structure we store \((key, value)\)-pairs such that for any \(key\) there is at most one pair \((key, value)\) in the dictionary.

Three operations are supported:

- `add(x, v)` Add the the pair \((x, v)\) where \(x \in U\) - the *universe*
- `lookup(x)` Return \(v\) if \((x, v)\) is in dictionary, or `NULL` otherwise.
- `delete(x)` Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

What happens if we add more operations?

We also want our data structure to support:

- `predecessor(k)` - returns the (unique) element \((x, v)\) in the dictionary with the largest key, \(x\) such that \(x \leq k\)
- `successor(k)` - returns the (unique) element \((x, v)\) in the dictionary with the smallest key, \(x\) such that \(x \geq k\)
Supporting more operations

In a **dynamic dictionary** data structure we store \((key, value)\)-pairs such that for any \(key\) there is at most one pair \((key, value)\) in the dictionary.

Three operations are supported:

- **add**\((x, v)\) Add the the pair \((x, v)\) where \(x \in U\) - the universe
- **lookup**\((x)\) Return \(v\) if \((x, v)\) is in dictionary, or **NULL** otherwise.
- **delete**\((x)\) Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

What happens if we add more operations?

We also want our data structure to support:

- **predecessor**\((k)\) - returns the (unique) element \((x, v)\) in the dictionary with the largest key, \(x\) such that \(x \leq k\)
- **successor**\((k)\) - returns the (unique) element \((x, v)\) in the dictionary with the smallest key, \(x\) such that \(x \geq k\)
Supporting more operations

In a **dynamic dictionary** data structure we store \((key, value)\)-pairs such that for any *key* there is at most one pair \((key, value)\) in the dictionary.

Three operations are supported:

- \(\text{add}(x, v)\) Add the pair \((x, v)\) where \(x \in U\) - the *universe*
- \(\text{lookup}(x)\) Return \(v\) if \((x, v)\) is in dictionary, or *NULL* otherwise.
- \(\text{delete}(x)\) Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

What happens if we add more operations?

We also want our data structure to support:

- \(\text{predecessor}(k)\) - returns the (unique) element \((x, v)\) in the dictionary with the largest key, \(x\) such that \(x \leq k\)
- \(\text{successor}(k)\) - returns the (unique) element \((x, v)\) in the dictionary with the smallest key, \(x\) such that \(x \geq k\)

These are very natural operations that the **Hashing**-based solutions that we have seen are very unsuited to
What could we use instead?

We could use a self-balancing binary search tree... like a 2-3-4 tree, a red-black tree or an AVL tree.
What could we use instead?

We could use a self-balancing binary search tree... like a 2-3-4 tree, a red-black tree or an AVL tree.

All three of these data structures support:

\[ \text{add}(x, v), \text{lookup}(x), \text{delete}(x), \text{predecessor}(k) \text{ and } \text{successor}(k) \]
What could we use instead?

We could use a self-balancing binary search tree... like a 2-3-4 tree, a red-black tree or an AVL tree.

All three of these data structures support:

- add\((x, v)\), lookup\(x\), delete\(x\), predecessor\(k\) and successor\(k\)

  each in \(O(\log n)\) worst case time and \(O(n)\) space.
What could we use instead?

We could use a self-balancing binary search tree... like a 2-3-4 tree, a red-black tree or an AVL tree.

All three of these data structures support:

- \( \text{add}(x, v) \), \( \text{lookup}(x) \), \( \text{delete}(x) \), \( \text{predecessor}(k) \) and \( \text{successor}(k) \)

each in \( O(\log n) \) worst case time and \( O(n) \) space

where \( n \) is the number of elements stored.
What could we use instead?

We could use a self-balancing binary search tree... like a 2-3-4 tree, a red-black tree or an AVL tree.

All three of these data structures support:

- add($x, v$), lookup($x$), delete($x$), predecessor($k$) and successor($k$)

  each in $O(\log n)$ worst case time and $O(n)$ space

  where $n$ is the number of elements stored

  they are also deterministic
What could we use instead?

We could use a self-balancing binary search tree…

like a 2-3-4 tree, a red-black tree or an AVL tree

2-3-4 trees were covered in DSA this year in the “dynamic search structures” lectures (the slides are online if you are interested)

All three of these data structures support:

\begin{align*}
\text{add}(x, v), \text{lookup}(x), \text{delete}(x), \text{predecessor}(k) \text{ and } \text{successor}(k)
\end{align*}

each in $O(\log n)$ worst case time and $O(n)$ space

where $n$ is the number of elements stored

you are also deterministic
van Emde Boas Trees

In this lecture, we will see the van Emde Boas (vEB) tree which stores a set $S$ of integer keys from a universe $U = \{1, 2, 3, 4 \ldots u\}$ (i.e. $u = |U|$).

Five operations will be supported:

- **add**($x$) 
  Insert the integer $x$ into $S$ (where $x \in U$)

- **lookup**($x$) 
  Return yes if $x$ is in $S$, or no otherwise.

- **delete**($x$) 
  Remove $x$ from $S$

- **predecessor**($k$) 
  Return the largest integer $x$ in $S$ such that $x \leq k$

- **successor**($k$) 
  Return the smallest integer $x$ in $S$ such that $x \geq k$

\[1 \ 2 \ 3 \ 4 \ldots \ k \ u\]

\[\text{predecessor}(k) \quad \text{successor}(k)\]
van Emde Boas Trees

In this lecture, we will see the **van Emde Boas (vEB) tree**

which stores a set $S$ of **integer keys** from a universe $U = \{1, 2, 3, 4 \ldots u\}$ (i.e. $u = |U|$).

**Five** operations will be supported:

- `add(x)`: Insert the integer $x$ into $S$ (where $x \in U$)
- `lookup(x)`: Return **yes** if $x$ is in $S$, or **no** otherwise.
- `delete(x)`: Remove $x$ from $S$
- `predecessor(k)`: Return the **largest** integer $x$ in $S$ such that $x \leq k$
- `successor(k)`: Return the **smallest** integer $x$ in $S$ such that $x \geq k$

**Warning:** As stated the operations do not store any data (values) with the integers (keys)
van Emde Boas Trees

In this lecture, we will see the **van Emde Boas (vEB) tree**
which stores a set $S$ of integer keys from a universe $U = \{1, 2, 3, 4 \ldots u\}$ (i.e. $u = |U|$).

Five operations will be supported:

- **add($x$)**: Insert the integer $x$ into $S$ (where $x \in U$)
- **lookup($x$)**: Return *yes* if $x$ is in $S$, or *no* otherwise.
- **delete($x$)**: Remove $x$ from $S$
- **predecessor($k$)**: Return the largest integer $x$ in $S$ such that $x \leq k$
- **successor($k$)**: Return the smallest integer $x$ in $S$ such that $x \geq k$

**Warning:** As stated the operations do not store any data (values) with the integers (keys).

It is straightforward to extend the **van Emde Boas tree** to store (key, value) pairs when the keys are integers from $U$.
van Emde Boas Trees

In this lecture, we will see the **van Emde Boas (vEB) tree** which stores a set $S$ of integer keys from a universe $U = \{1, 2, 3, 4 \ldots u\}$ (i.e. $u = |U|$).

Five operations will be supported:

- **add($x$)**: Insert the integer $x$ into $S$ (where $x \in U$)
- **lookup($x$)**: Return yes if $x$ is in $S$, or no otherwise.
- **delete($x$)**: Remove $x$ from $S$
- **predecessor($k$)**: Return the largest integer $x$ in $S$ such that $x \leq k$
- **successor($k$)**: Return the smallest integer $x$ in $S$ such that $x \geq k$

**Warning:** As stated the operations do not store any data (values) with the integers (keys)

It is straightforward to extend the **van Emde Boas tree** to store (key, value) pairs when the keys are integers from $U$

*(but I think it’s easier to think about like this)*
van Emde Boas Trees

In this lecture, we will see the **van Emde Boas (vEB) tree** which stores a set $S$ of integer keys from a universe $U = \{1, 2, 3, 4 \ldots u\}$ (i.e. $u = |U|$).

**Five** operations will be supported:

- **add($x$)**: Insert the integer $x$ into $S$ (where $x \in U$)
- **lookup($x$)**: Return *yes* if $x$ is in $S$, or *no* otherwise.
- **delete($x$)**: Remove $x$ from $S$
- **predecessor($k$)**: Return the largest integer $x$ in $S$ such that $x \leq k$
- **successor($k$)**: Return the smallest integer $x$ in $S$ such that $x \geq k$

![Diagram of van Emde Boas Trees](image)
van Emde Boas Trees

In this lecture, we will see the **van Emde Boas (vEB) tree** which stores a set $S$ of integer keys from a universe $U = \{1, 2, 3, 4 \ldots u\}$ (i.e. $u = |U|$).

Five operations will be supported:

- **add**($x$) Insert the integer $x$ into $S$ (where $x \in U$)
- **lookup**($x$) Return **yes** if $x$ is in $S$, or **no** otherwise.
- **delete**($x$) Remove $x$ from $S$
- **predecessor**($k$) Return the largest integer $x$ in $S$ such that $x \leq k$
- **successor**($k$) Return the smallest integer $x$ in $S$ such that $x \geq k$

All operations will take $O(\log \log u)$ worst case time and the space used is $O(u)$.
van Emde Boas Trees

In this lecture, we will see the van Emde Boas (vEB) tree
which stores a set $S$ of integer keys from a universe $U = \{1, 2, 3, 4 \ldots u\}$ (i.e. $u = |U|$).

Five operations will be supported:

- **add($x$)**: Insert the integer $x$ into $S$ (where $x \in U$)
- **lookup($x$)**: Return yes if $x$ is in $S$, or no otherwise.
- **delete($x$)**: Remove $x$ from $S$
- **predecessor($k$)**: Return the largest integer $x$ in $S$ such that $x \leq k$
- **successor($k$)**: Return the smallest integer $x$ in $S$ such that $x \geq k$

All operations will take $O(\log \log u)$ worst case time
and the space used is $O(u)$
and it is a deterministic data structure
van Emde Boas Trees

In this lecture, we will see the van Emde Boas (vEB) tree which stores a set $S$ of integer keys from a universe $U = \{1, 2, 3, 4 \ldots u\}$ (i.e. $u = |U|$).

Five operations will be supported:

- **add**($x$) Insert the integer $x$ into $S$ (where $x \in U$)
- **lookup**($x$) Return yes if $x$ is in $S$, or no otherwise.
- **delete**($x$) Remove $x$ from $S$
- **predecessor**($k$) Return the largest integer $x$ in $S$ such that $x \leq k$
- **successor**($k$) Return the smallest integer $x$ in $S$ such that $x \geq k$

All operations will take $O(\log \log u)$ worst case time and the space used is $O(u)$

*and it is a deterministic data structure*

**Example:** If $U = \{1, 2, 3, 4 \ldots 100 \cdot n\}$, you get $O(\log \log n)$ time and $O(n)$ space
van Emde Boas Trees

In this lecture, we will see the van Emde Boas (vEB) tree, which stores a set $S$ of integer keys from a universe $U = \{1, 2, 3, 4 \ldots u\}$ (i.e. $u = |U|$).

Five operations will be supported:

- **add($x$)**: Insert the integer $x$ into $S$ (where $x \in U$)
- **lookup($x$)**: Return yes if $x$ is in $S$, or no otherwise.
- **delete($x$)**: Remove $x$ from $S$
- **predecessor($k$)**: Return the largest integer $x$ in $S$ such that $x \leq k$
- **successor($k$)**: Return the smallest integer $x$ in $S$ such that $x \geq k$

All operations will take $O(\log \log u)$ worst case time and the space used is $O(u)$ and it is a deterministic data structure.

**Example:** If $U = \{1, 2, 3, 4 \ldots n^2\}$, you get $O(\log \log n)$ time and $O(n^2)$ space.
van Emde Boas Trees

In this lecture, we will see the **van Emde Boas (vEB) tree** which stores a set $S$ of integer keys from a universe $U = \{1, 2, 3, 4 \ldots u\}$ (i.e. $u = |U|$).

Five operations will be supported:

- **add($x$)**: Insert the integer $x$ into $S$ (where $x \in U$)
- **lookup($x$)**: Return *yes* if $x$ is in $S$, or *no* otherwise.
- **delete($x$)**: Remove $x$ from $S$
- **predecessor($k$)**: Return the largest integer $x$ in $S$ such that $x \leq k$
- **successor($k$)**: Return the smallest integer $x$ in $S$ such that $x \geq k$

All operations will take $O(\log \log u)$ worst case time and the space used is $O(u)$ and it is a deterministic data structure.

**Example:** If $U = \{1, 2, 3, 4 \ldots n^3\}$, you get $O(\log \log n)$ time and $O(n^3)$ space.
Attempt 1: a big array

Build an array of length $u$...
Attempt 1: a big array

Build an array of length $u$ . . .

\[ A[i] = 1 \text{ iff } i \text{ is in } S \]
Attempt 1: a big array

Build an array of length $u$ ... 

$$A[i] = 1 \text{ iff } i \text{ is in } S$$

The operations add, delete and lookup all take $O(1)$ time.
**Attempt 1: a big array**

Build an array of length $u$...

$A[i] = 1$ iff $i$ is in $S$

The operations **add**, **delete** and **lookup** all take $O(1)$ time.
Attempt 1: a big array

Build an array of length $u$ ...

$A[i] = 1$ iff $i$ is in $S$

The operations add, delete and lookup all take $O(1)$ time.
**Attempt 1:** a big array

Build an array of length \( u \)...

\[
A[i] = 1 \text{ iff } i \text{ is in } S
\]

The operations \textit{add}, \textit{delete} and \textit{lookup} all take \( O(1) \) time.
**Attempt 1: a big array**

Build an array of length $u$ . . .

![Array representation with elements 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0]

$A[i] = 1$ iff $i$ is in $S$

The operations *add*, *delete* and *lookup* all take $O(1)$ time.
**Attempt 1: a big array**

Build an array of length $u$ . . .

$$A[i] = 1 \text{ iff } i \text{ is in } S$$

The operations **add**, **delete** and **lookup** all take $O(1)$ time.
**Attempt 1: a big array**

Build an array of length $u$...

$$A[i] = 1 \text{ iff } i \text{ is in } S$$

**delete(14)**

The operations **add**, **delete** and **lookup** all take $O(1)$ time.
**Attempt 1: a big array**

Build an array of length $u$...

$A[i] = 1$ iff $i$ is in $S$

The operations **add**, **delete** and **lookup** all take $O(1)$ time.
Attempt 1: a big array

Build an array of length $u$ . . .

$A[i] = 1$ iff $i$ is in $S$

The operations add, delete and lookup all take $O(1)$ time.
**Attempt 1: a big array**

Build an array of length $u$...

A $[i] = 1$ iff $i$ is in $S$

The operations add, delete and lookup all take $O(1)$ time.
Attempt 1: a big array

Build an array of length $u$...

$A[i] = 1$ iff $i$ is in $S$

The operations add, delete and lookup all take $O(1)$ time.
Attempt 1: a big array

Build an array of length $u$...

$$A[i] = 1 \text{ iff } i \text{ is in } S$$

The operations add, delete and lookup all take $O(1)$ time.
Attempt 1: a big array

Build an array of length $u$...

$A[i] = 1$ iff $i$ is in $S$

The operations add, delete and lookup all take $O(1)$ time.

...looks good so far!
Attempt 1: a big array

Build an array of length $u$...

$A[i] = 1$ iff $i$ is in $S$

The operations add, delete and lookup all take $O(1)$ time.

...looks good so far!

What about the predecessor operation?
Attempt 1: a big array

Build an array of length $u$...

\[ A[i] = 1 \text{ iff } i \text{ is in } S \]

The operations add, delete and lookup all take $O(1)$ time. 

...looks good so far!

What about the predecessor operation?
**Attempt 1: a big array**

Build an array of length $u$...

![Array Diagram]

$A[i] = 1$ iff $i$ is in $S$

predecessor(11)

The operations add, delete and lookup all take $O(1)$ time.  
...looks good so far!

What about the predecessor operation?
Attempt 1: a big array

Build an array of length \( u \) . . .

\[
A[i] = 1 \text{ iff } i \text{ is in } S
\]

predecessor(11)

The operations add, delete and lookup all take \( O(1) \) time.

. . . looks good so far!

What about the predecessor operation?
**Attempt 1: a big array**

Build an array of length $u$...

\[ A[i] = 1 \text{ iff } i \text{ is in } S \]

The operations add, delete and lookup all take $O(1)$ time.

...looks good so far!
Attempt 1: a big array

Build an array of length $u$...

$A[i] = 1$ if $i$ is in $S$

The operations add, delete and lookup all take $O(1)$ time.

...looks good so far!
Attempt 1: a big array

Build an array of length \( u \)...

\[ A[i] = 1 \text{ iff } i \text{ is in } S \]

The operations \text{add}, \text{delete} and \text{lookup} all take \( O(1) \) time.

\ldots looks good so far!

predecessor(11)
Attempt 1: a big array

Build an array of length \( u \)...

\[ A[i] = 1 \text{ iff } i \text{ is in } S \]

The operations add, delete and lookup all take \( O(1) \) time.

\( \text{predecessor}(11) \)

\( \ldots \text{looks good so far!} \)

The \textit{predecessor} and \textit{successor} operations take \( O(u) \) time
Attempt 1: a big array

Build an array of length $u$...

$A[i] = 1$ iff $i$ is in $S$

The operations add, delete and lookup all take $O(1)$ time.
...looks good so far!

The predecessor and successor operations take $O(u)$ time
Attempt 1: a big array

Build an array of length $u$...

$A[i] = 1$ iff $i$ is in $S$

The operations add, delete and lookup all take $O(1)$ time.

...looks good so far!

The predecessor and successor operations take $O(u)$ time

...not so good!
Attempt 2: a balanced binary tree

Build a balanced binary tree on top of the array...
Attempt 2: a balanced binary tree

Build a balanced binary tree on top of the array…

each node is 1 if either child is 1
**Attempt 2: a balanced binary tree**

Build a balanced binary tree on top of the array...

Each node is 1 if either child is 1

* i.e. if the subtree contains a 1
**Attempt 2: a balanced binary tree**

Build a balanced binary tree on top of the array...

Each node is 1 if either child is 1, i.e., if the subtree contains a 1.

The operations add and delete take $O(\log u)$ time.
Attempt 2: a balanced binary tree

Build a balanced binary tree on top of the array...

Each node is 1 if either child is 1

* i.e. if the subtree contains a 1

The operations add and delete take $O(\log u)$ time.
Attempt 2: a balanced binary tree

Build a balanced binary tree on top of the array... each node is 1 if either child is 1

\[ \text{add}(12) \]

The operations add and delete take \( O(\log u) \) time.
Attempt 2: a balanced binary tree

Build a balanced binary tree on top of the array...

each node is 1 if either child is 1

i.e if the subtree contains a 1

The operations add and delete take $O(\log u)$ time.
**Attempt 2: a balanced binary tree**

Build a balanced binary tree on top of the array... 

Each node is 1 if either child is 1, i.e., if the subtree contains a 1.

The operations add and delete take $O(\log u)$ time.
Attempt 2: a balanced binary tree

Build a balanced binary tree on top of the array...

each node is 1 if either child is 1

i.e if the subtree contains a 1

The operations add and delete take $O(\log u)$ time.
Attempt 2: a balanced binary tree

Build a balanced binary tree on top of the array...

each node is 1 if either child is 1

i.e if the subtree contains a 1

The operations add and delete take $O(\log u)$ time.
**Attempt 2: a balanced binary tree**

Build a balanced binary tree on top of the array... (on top of a big array)

Each node is 1 if either child is 1, i.e., if the subtree contains a 1.

The operations add and delete take $O(\log u)$ time.
Attempt 2: a balanced binary tree

Build a balanced binary tree on top of the array...

Each node is 1 if either child is 1

i.e if the subtree contains a 1

The operations add and delete take $O(\log u)$ time.

The lookup operation still takes $O(1)$ time (simply look at $A[x]$)
**Attempt 2: a balanced binary tree**

Build a balanced binary tree on top of the array... (on top of a big array)

Each node is 1 if either child is 1, i.e., if the subtree contains a 1.

The operations add and delete take $O(\log u)$ time.

The lookup operation still takes $O(1)$ time (simply look at $A[x]$).

How do we find a predecessor?
Attempt 2: a balanced binary tree

Build a balanced binary tree on top of the array...

Each node is 1 if either child is 1

i.e. if the subtree contains a 1

The operations add and delete take $O(\log u)$ time.

The lookup operation still takes $O(1)$ time (simply look at $A[x]$)

How do we find a predecessor?
Attempt 2: a balanced binary tree
(on top of a big array)

Build a balanced binary tree on top of the array...

Each node is 1 if either child is 1 i.e if the subtree contains a 1

The operations add and delete take $O(\log u)$ time.

The lookup operation still takes $O(1)$ time (simply look at $A[x]$)

How do we find a predecessor?
Attempt 2: a balanced binary tree

Build a balanced binary tree on top of the array...

each node is 1 if either child is 1

i.e if the subtree contains a 1

The operations add and delete take \( O(\log u) \) time.

The lookup operation still takes \( O(1) \) time (simply look at \( A[x] \))

How do we find a predecessor?
Attempt 2: a balanced binary tree

Build a balanced binary tree on top of the array...

Each node is 1 if either child is 1

i.e. if the subtree contains a 1

The operations add and delete take $O(\log u)$ time.

The lookup operation still takes $O(1)$ time (simply look at $A[x]$)

How do we find a predecessor?
Attempt 2: a balanced binary tree

Build a balanced binary tree on top of the array...

Each node is 1 if either child is 1 (i.e. if the subtree contains a 1)

The operations add and delete take $O(\log u)$ time.

The lookup operation still takes $O(1)$ time (simply look at $A[x]$)

How do we find a predecessor?
**Attempt 2:** a balanced binary tree

Build a balanced binary tree on top of the array...

Each node is 1 if either child is 1

* i.e if the subtree contains a 1

The operations **add** and **delete** take $O(\log u)$ time.

The **lookup** operation still takes $O(1)$ time (simply look at $A[x]$)

How do we find a predecessor?
Attempt 2: a balanced binary tree

Build a balanced binary tree on top of the array...

Each node is 1 if either child is 1

i.e. if the subtree contains a 1

The operations add and delete take $O(\log u)$ time.

The lookup operation still takes $O(1)$ time (simply look at $A[x]$)

How do we find a predecessor?
Attempt 2: a balanced binary tree

Build a balanced binary tree on top of the array...

Each node is 1 if either child is 1 (i.e., if the subtree contains a 1).

The operations add and delete take $O(\log u)$ time.

The lookup operation still takes $O(1)$ time (simply look at $A[x]$).

How do we find a predecessor?
Attempt 2: a balanced binary tree

Build a balanced binary tree on top of the array...

Each node is 1 if either child is 1 i.e. if the subtree contains a 1

The operations add and delete take $O(\log u)$ time.

The lookup operation still takes $O(1)$ time (simply look at $A[x]$)

How do we find a predecessor?
**Attempt 2: a balanced binary tree**

Build a balanced binary tree on top of the array...

Each node is 1 if either child is 1

\[ \text{i.e if the subtree contains a 1} \]

The operations **add** and **delete** take \( O(\log u) \) time.

The **lookup** operation still takes \( O(1) \) time (simply look at \( A[x] \))

**How do we find a predecessor?**
**Attempt 2: a balanced binary tree**

*on top of a big array*

Build a balanced binary tree on top of the array...

Each node is 1 if either child is 1

*If the subtree contains a 1*

The operations **add** and **delete** take \( O(\log u) \) time.

The **lookup** operation still takes \( O(1) \) time (simply look at \( A[x] \))

How do we find a predecessor?
Attempt 2: a balanced binary tree
(on top of a big array)

Build a balanced binary tree on top of the array...

Each node is 1 if either child is 1

i.e. if the subtree contains a 1

The operations add and delete take $O(\log u)$ time.

The lookup operation still takes $O(1)$ time (simply look at $A[x]$)

How do we find a predecessor?
**Attempt 2:** a balanced binary tree

Build a balanced binary tree on top of the array...

Each node is 1 if either child is 1

*i.e. if the subtree contains a 1*

The operations *add* and *delete* take $O(\log u)$ time.

The *lookup* operation still takes $O(1)$ time (simply look at $A[x]$)

How do we find a predecessor?
Attempt 2: a balanced binary tree

Build a balanced binary tree on top of the array...

Each node is 1 if either child is 1, i.e., if the subtree contains a 1.

The operations add and delete take $O(\log u)$ time.

The lookup operation still takes $O(1)$ time (simply look at $A[x]$).
Attempt 2: a balanced binary tree

(on top of a big array)

Build a balanced binary tree on top of the array...

The operations add and delete take $O(\log u)$ time.

The lookup operation still takes $O(1)$ time (simply look at $A[x]$).

The operations predecessor and successor take $O(\log u)$ time.
Attempt 3: a constant height tree

(on top of a big array)
**Attempt 3:** a constant height tree

*(on top of a big array)*

<table>
<thead>
<tr>
<th>A</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits.
Attempt 3: a constant height tree
(on top of a big array)

\[ A \]

Split \( A \) into \( \sqrt{u} \) blocks each containing \( \sqrt{u} \) bits
Attempt 3: a constant height tree
(on top of a big array)

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits
Attempt 3: a constant height tree

(on top of a big array)

\( C \) is called the summary of \( A \)

\[
\begin{array}{cccc}
1 & 1 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccc}
0 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
\end{array}
\]

Split \( A \) into \( \sqrt{u} \) blocks each containing \( \sqrt{u} \) bits
**Attempt 3:** a constant height tree

*on top of a big array*

\(C\) is called the **summary** of \(A\)

Split \(A\) into \(\sqrt{u}\) **blocks** each containing \(\sqrt{u}\) bits
**Attempt 3:** a constant height tree

*(on top of a big array)*

C is called the summary of A

this is 1 if any bit in the child block is 1

Split A into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits
Attempt 3: a constant height tree
(on top of a big array)

$C$ is called the summary of $A$

this is 1 if any bit in the child block is 1

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits
**Attempt 3:** a constant height tree

*(on top of a big array)*

\(C\) is called the *summary* of \(A\)

This is 1 if any bit in the child block is 1

Split \(A\) into \(\sqrt{u}\) blocks each containing \(\sqrt{u}\) bits

The *lookup* and *add* operations take \(O(1)\) time.
**Attempt 3:** a constant height tree

*(on top of a big array)*

$C$ is called the *summary* of $A$

this is 1 if any bit in the child block is 1

lookup(12)  

Split $A$ into $\sqrt{u}$ *blocks* each containing $\sqrt{u}$ bits

The lookup and add operations take $O(1)$ time.
Attempt 3: a constant height tree
(on top of a big array)

$C$ is called the summary of $A$

this is 1 if any bit in the child block is 1

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits

The lookup and add operations take $O(1)$ time.
**Attempt 3:** a constant height tree

*(on top of a big array)*

\(C\) is called the *summary* of \(A\)

This is 1 if any bit in the child block is 1

Split \(A\) into \(\sqrt{u}\) *blocks* each containing \(\sqrt{u}\) bits

The lookup and add operations take \(O(1)\) time.
Attempt 3: a constant height tree

(on top of a big array)

\( C \) is called the \textit{summary} of \( A \)

this is 1 if any bit in the child block is 1

Add(9)

Split \( A \) into \( \sqrt{u} \) \textit{blocks} each containing \( \sqrt{u} \) bits

The \textit{lookup} and \textit{add} operations take \( O(1) \) time.
**Attempt 3:** a constant height tree  
(on top of a big array)

$C$ is called the **summary** of $A$.

This is 1 if any bit in the child block is 1.

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits.

The **lookup** and **add** operations take $O(1)$ time.
Attempt 3: a constant height tree
(on top of a big array)

$C$ is called the summary of $A$

this is 1 if any bit in the child block is 1

add(9)

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits

The lookup and add operations take $O(1)$ time.
**Attempt 3:** a constant height tree

*(on top of a big array)*

$C$ is called the *summary* of $A$

this is 1 if any bit in the child block is 1

add(9)

Split $A$ into $\sqrt{u}$ *blocks* each containing $\sqrt{u}$ bits

The *lookup* and *add* operations take $O(1)$ time.
**Attempt 3:** a constant height tree
*(on top of a big array)*

\( C \) is called the **summary** of \( A \)

This is 1 if any bit in the child block is 1

Split \( A \) into \( \sqrt{u} \) blocks each containing \( \sqrt{u} \) bits

The lookup and add operations take \( O(1) \) time.
Attempt 3: a constant height tree
(on top of a big array)

$C$ is called the summary of $A$.

this is 1 if any bit in the child block is 1

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits

The lookup and add operations take $O(1)$ time.

The operations delete, predecessor and successor take $O(\sqrt{u})$ time.
**Attempt 3:** a constant height tree

*(on top of a big array)*

$C$ is called the \textit{summary} of $A$

this is 1 if any bit in the child block is 1

Split $A$ into $\sqrt{u}$ \textit{blocks} each containing $\sqrt{u}$ bits

The \textit{lookup} and \textit{add} operations take $O(1)$ time.

The operations \textit{delete}, \textit{predecessor} and \textit{successor} take $O(\sqrt{u})$ time.
Attempt 3: a constant height tree
(on top of a big array)

$C$ is called the summary of $A$

this is 1 if any bit in the child block is 1

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits

The lookup and add operations take $O(1)$ time.

The operations delete, predecessor and successor take $O(\sqrt{u})$ time.
Attempt 3: a constant height tree
(on top of a big array)

\( C \) is called the summary of \( A \)

this is 1 if any bit in the child block is 1

to determine this bit you have to look through this block

Split \( A \) into \( \sqrt{u} \) blocks each containing \( \sqrt{u} \) bits

The lookup and add operations take \( O(1) \) time.

The operations delete, predecessor and successor take \( O(\sqrt{u}) \) time.
Attempt 3: a constant height tree
(on top of a big array)

$C$ is called the summary of $A$

define this is 1 if any bit in the child block is 1

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits

The lookup and add operations take $O(1)$ time.

The operations delete, predecessor and successor take $O(\sqrt{u})$ time.
Attempt 3: a constant height tree  
(on top of a big array)

\(C\) is called the {
\textit{summary}} of \(A\)

this is 1 if
any bit in the child block is 1

\[\sqrt{u}\]

Split \(A\) into \(\sqrt{u}\) blocks each containing \(\sqrt{u}\) bits

The \textit{lookup} and \textit{add} operations take \(O(1)\) time.

The operations \textit{delete}, \textit{predecessor} and \textit{successor} take \(O(\sqrt{u})\) time.
Attempt 3: a constant height tree

(on top of a big array)

\(C\) is called the summary of \(A\)

this is 1 if any bit in the child block is 1

\[\text{delete}(9)\]

\(A\)

\[\begin{array}{cccc}
0 & 1 & 1 & 1 \\
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12 \\
13 & 14 & 15 & 16 \\
\end{array}\]

Split \(A\) into \(\sqrt{u}\) blocks each containing \(\sqrt{u}\) bits

The lookup and add operations take \(O(1)\) time.

The operations delete, predecessor and successor take \(O(\sqrt{u})\) time.
**Attempt 3:** a constant height tree

*(on top of a big array)*

\( C \) is called the *summary* of \( A \)

This is 1 if any bit in the child block is 1

\[ \sqrt{u} \]

\[ \text{height} \]

Split \( A \) into \( \sqrt{u} \) blocks each containing \( \sqrt{u} \) bits

The *lookup* and *add* operations take \( O(1) \) time.

The operations *delete*, *predecessor* and *successor* take \( O(\sqrt{u}) \) time.
**Attempt 3:** a constant height tree

*(on top of a big array)*

\(C\) is called the **summary** of \(A\)

this is 1 if any bit in the child block is 1

\(\sqrt{u}\)

\(C\)

\(A\)

split \(A\) into \(\sqrt{u}\) **blocks** each containing \(\sqrt{u}\) bits

split \(A\) into \(\sqrt{u}\) **blocks** each containing \(\sqrt{u}\) bits

- The lookup and add operations take \(O(1)\) time.
- The operations delete, predecessor and successor take \(O(\sqrt{u})\) time.
Attempt 3: a constant height tree
(on top of a big array)

$C$ is called the summary of $A$

this is 1 if any bit in the child block is 1

$C$ is called the summary of $A$

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits

The lookup and add operations take $O(1)$ time.

The operations delete, predecessor and successor take $O(\sqrt{u})$ time.
Attempt 3: a constant height tree
(on top of a big array)

$C$ is called the summary of $A$

this is 1 if any bit in the child block is 1

predecessor(14)

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits

The lookup and add operations take $O(1)$ time.

The operations delete, predecessor and successor take $O(\sqrt{u})$ time.
**Attempt 3:** a constant height tree

*(on top of a big array)*

$C$ is called the summary of $A$

this is 1 if any bit in the child block is 1

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits

The lookup and add operations take $O(1)$ time.

The operations delete, predecessor and successor take $O(\sqrt{u})$ time.
**Attempt 3:** a constant height tree  
(on top of a big array)

$C$ is called the summary of $A$.

This is 1 if any bit in the child block is 1.

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits.

The lookup and add operations take $O(1)$ time.

The operations delete, predecessor and successor take $O(\sqrt{u})$ time.
**Attempt 3:** a constant height tree  
*(on top of a big array)*

\(C\) is called the **summary** of \(A\)

This is 1 if any bit in the child block is 1

Split \(A\) into \(\sqrt{u}\) **blocks** each containing \(\sqrt{u}\) bits

The lookup and add operations take \(O(1)\) time.

The operations delete, predecessor and successor take \(O(\sqrt{u})\) time.
**Attempt 3: a constant height tree**

*(on top of a big array)*

\(C\) is called the *summary* of \(A\)

this is 1 if any bit in the child block is 1

\[\text{predecessor}(14)\]

Split \(A\) into \(\sqrt{u}\) blocks each containing \(\sqrt{u}\) bits

The *lookup* and *add* operations take \(O(1)\) time.

The operations *delete*, *predecessor* and *successor* take \(O(\sqrt{u})\) time.
**Attempt 3: a constant height tree**

*(on top of a big array)*

$C$ is called the **summary** of $A$

- this is $1$ if any bit in the child block is $1$

Split $A$ into $\sqrt{u}$ **blocks** each containing $\sqrt{u}$ bits

- The **lookup** and **add** operations take $O(1)$ time.
- The operations **delete**, **predecessor** and **successor** take $O(\sqrt{u})$ time.
Attempt 3: a constant height tree
(on top of a big array)

$C$ is called the summary of $A$

this is 1 if any bit in the child block is 1

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits

The lookup and add operations take $O(1)$ time.

The operations delete, predecessor and successor take $O(\sqrt{u})$ time.
**Attempt 3:** a constant height tree

*(on top of a big array)*

C is called the *summary* of A

this is 1 if any bit in the child block is 1

predecessor(14)

Split A into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits

The lookup and add operations take $O(1)$ time.

The operations delete, predecessor and successor take $O(\sqrt{u})$ time.
**Attempt 3:** a constant height tree

(on top of a big array)

$C$ is called the *summary* of $A$

this is 1 if any bit in the child block is 1

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits

The lookup and add operations take $O(1)$ time.

The operations delete, predecessor and successor take $O(\sqrt{u})$ time.
Attempt 3: a constant height tree
(on top of a big array)

$C$ is called the summary of $A$

this is 1 if any bit in the child block is 1

\[
p\text{predecessor}(14)\]

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits

The lookup and add operations take $O(1)$ time.

The operations delete, predecessor and successor take $O(\sqrt{u})$ time.
**Attempt 3:** a constant height tree

*(on top of a big array)*

$C$ is called the *summary* of $A$.

this is 1 if any bit in the child block is 1

Split $A$ into $\sqrt{u}$ *blocks* each containing $\sqrt{u}$ bits

The lookup and add operations take $O(1)$ time.

The operations delete, predecessor and successor take $O(\sqrt{u})$ time.
**Attempt 3:** a constant height tree

*(on top of a big array)*

$C$ is called the *summary* of $A$

In the worst case we look at all of $C$ and all of two blocks

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits

The lookup and add operations take $O(1)$ time.

The operations delete, predecessor and successor take $O(\sqrt{u})$ time.
Attempt 3: a constant height tree
(on top of a big array)

$C$ is called the summary of $A$

In the worst case we look at all of $C$ and all of two blocks
(successor is the same)

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits

The lookup and add operations take $O(1)$ time.

The operations delete, predecessor and successor take $O(\sqrt{u})$ time.
An abstract view

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

there is a whole lot more universe in here

1 2 3 4 ...
An abstract view

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements.
An abstract view

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

there is a whole lot more universe in here
An abstract view

Split the universe $\mathcal{U}$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements.

We can think of each block as a ‘little’ universe of size $\sqrt{u}$.

There is a whole lot more universe in here.

\[
\begin{array}{cccccc}
\text{\textbullet} & \text{\textbullet} & \text{\textbullet} & \text{\textbullet} & \text{\textbullet} \\
\hline
\sqrt{u} & \sqrt{u} & \sqrt{u} & \sqrt{u} & \sqrt{u} \\
\hline
u & u & u & u & u
\end{array}
\]
An abstract view

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements.
An abstract view

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

For block $i$, we build a data structure $B[i]$ which stores elements from $\{1, 2, 3, \ldots \sqrt{u}\}$
An abstract view

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

For block $i$, we build a data structure $B[i]$ which stores elements from $\{1, 2, 3, \ldots \sqrt{u}\}$
An abstract view

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

we can think of each block as a ‘little’ universe of size $\sqrt{u}$

For block $i$, we build a data structure $B[i]$ which stores elements from $\{1, 2, 3, \ldots \sqrt{u}\}$

$x$ is stored in $B[i]$ iff $(x + (i - 1)\sqrt{u}) \in S$
An abstract view

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

we can think of each block as a ‘little’ universe of size $\sqrt{u}$

For block $i$, we build a data structure $B[i]$

which stores elements from $\{1, 2, 3, \ldots \sqrt{u}\}$

$x$ is stored in $B[i]$ iff $(x + (i - 1)\sqrt{u}) \in S$

(this is just to deal with the offset from the start of the real universe)
An abstract view

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

For block $i$, we build a data structure $B[i]$ which stores elements from $\{1, 2, 3, \ldots \sqrt{u}\}$

$x$ is stored in $B[i]$ iff $(x + (i - 1)\sqrt{u}) \in S$
An abstract view

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

For block $i$, we build a data structure $B[i]$ which stores elements from $\{1, 2, 3, \ldots \sqrt{u}\}$

$x$ is stored in $B[i]$ iff $(x + (i - 1)\sqrt{u}) \in S$

We also build a summary data structure $C$ which stores elements from $\{1, 2, 3, \ldots \sqrt{u}\}$

$i$ is stored in $C$ iff $B[i]$ is non-empty
An abstract view

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

For block $i$, we build a data structure $B[i]$ which stores elements from $\{1, 2, 3, \ldots \sqrt{u}\}$

$x$ is stored in $B[i]$ iff $(x + (i - 1)\sqrt{u}) \in S$

We also build a summary data structure $C$ which stores elements from $\{1, 2, 3, \ldots \sqrt{u}\}$

$i$ is stored in $C$ iff $B[i]$ is non-empty
An abstract view

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

How should we build $B[1], B[2], \ldots$, $B[\sqrt{u}]$ and $C$?
An abstract view

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

$$\sqrt{u}$$

$C$


$$\sqrt{u} \quad \sqrt{u} \quad \sqrt{u} \quad \sqrt{u}$$

$u$

How should we build $B[1], B[2], \ldots B[\sqrt{u}]$ and $C$?

Recursion!
An abstract view

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

How should we build $B[1], B[2], \ldots B[\sqrt{u}]$ and $C$?

Recursion!

Each $B[i]$ has universe $\{1, 2, 3, \ldots \sqrt{u}\}$
An abstract view

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

\[ \begin{array}{c}
\sqrt{u} \\
\hline
\end{array} \]

$C$

\[ \begin{array}{c}
B[1] \\
\hline
\end{array} \quad \begin{array}{c}
B[2] \\
\hline
\end{array} \quad \begin{array}{c}
B[3] \\
\hline
\end{array} \quad \begin{array}{c}
B[\sqrt{u}] \\
\hline
\end{array} \]

How should we build $B[1], B[2], \ldots B[\sqrt{u}]$ and $C$?

Recursion!

Each $B[i]$ has universe $\{1, 2, 3, \ldots \sqrt{u}\}$

We recursively split this into $\sqrt[4]{u}$ blocks each associated with $\sqrt[4]{u}$ elements...
An abstract view

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

How should we build $B[1], B[2], \ldots B[\sqrt{u}]$ and $C$?

**Recursion!**

Each $B[i]$ has universe $\{1, 2, 3, \ldots \sqrt{u}\}$

We recursively split this into $\sqrt[4]{u}$ blocks each associated with $\sqrt[4]{u}$ elements...

eventually (after some more work), this will lead to an $O(\log \log n)$ time solution
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements.
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements.

How do we perform the operations?
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

To perform $\text{add}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in  
(this takes $O(1)$ time with a little bit twiddling)

**Step 2** If $B[i]$ is empty, add $i$ to $C$

**Step 3** add $x$ to $B[i]$ (suitably adjusting the offset from the start of $B[i]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

To perform $\text{add}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in
(this takes $O(1)$ time with a little bit twiddling)

**Step 2** If $B[i]$ is empty, add $i$ to $C$

**Step 3** Add $x$ to $B[i]$ (suitably adjusting the offset from the start of $B[i]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

To perform $\text{add}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in
(this takes $O(1)$ time with a little bit twiddling)

**Step 2** If $B[i]$ is empty, add $i$ to $C$

**Step 3** add $x$ to $B[i]$ (suitably adjusting the offset from the start of $B[i]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

To perform $\text{add}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in
(this takes $O(1)$ time with a little bit twiddling)

**Step 2** If $B[i]$ is empty, add $i$ to $C$

**Step 3** Add $x$ to $B[i]$ (suitably adjusting the offset from the start of $B[i]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements.

To perform $\text{add}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in
(this takes $O(1)$ time with a little bit twiddling)

**Step 2** If $B[i]$ is empty, add $i$ to $C$

**Step 3** add $x$ to $B[i]$ (suitably adjusting the offset from the start of $B[i]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements.

To perform $\text{add}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in (this takes $O(1)$ time with a little bit twiddling)

**Step 2** If $B[i]$ is empty, add $i$ to $C$

**Step 3** add $x$ to $B[i]$ (suitably adjusting the offset from the start of $B[i]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements.

To perform $\text{add}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in
(this takes $O(1)$ time with a little bit twiddling)

**Step 2** If $B[i]$ is empty, add $i$ to $C$

**Step 3** add $x$ to $B[i]$ (suitably adjusting the offset from the start of $B[i]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

To perform $\text{add}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in
(this takes $O(1)$ time with a little bit twiddling)

**Step 2** If $B[i]$ is empty, add $i$ to $C$

**Step 3** add $x$ to $B[i]$ (suitably adjusting the offset from the start of $B[i]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements.

To perform $\text{add}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in
(this takes $O(1)$ time with a little bit twiddling)

**Step 2** If $B[i]$ is empty, add $i$ to $C$

**Step 3** add $x$ to $B[i]$ (suitably adjusting the offset from the start of $B[i]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements.
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

To perform $\text{predecessor}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in

**Step 2** Compute the predecessor of $x$ in $B[i]

(suitably adjusting the offset from the start of $B[i]$)

**Step 3** If $x$ has no predecessor in $B[i]$:

Compute $j = \text{predecessor}(i)$ in $C$

Compute the predecessor of $x$ in $B[j]$

(suitably adjusting the offset from the start of $B[j]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

To perform $\text{predecessor}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in

**Step 2** Compute the predecessor of $x$ in $B[i]$

(suitably adjusting the offset from the start of $B[i]$)

**Step 3** If $x$ has no predecessor in $B[i]$:

- Compute $j = \text{predecessor}(i)$ in $C$
- Compute the predecessor of $x$ in $B[j]$

(suitably adjusting the offset from the start of $B[j]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

To perform $\text{predecessor}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in

**Step 2** Compute the predecessor of $x$ in $B[i]$ (suitably adjusting the offset from the start of $B[i]$)

**Step 3** If $x$ has no predecessor in $B[i]$

Compute $j = \text{predecessor}(i)$ in $C$

Compute the predecessor of $x$ in $B[j]$ (suitably adjusting the offset from the start of $B[j]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

To perform $\text{predecessor}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in

**Step 2** Compute the predecessor of $x$ in $B[i]$ \\
(suitably adjusting the offset from the start of $B[i]$)

**Step 3** If $x$ has no predecessor in $B[i]$: \\
Compute $j = \text{predecessor}(i)$ in $C$ \\
Compute the predecessor of $x$ in $B[j]$ \\
(suitably adjusting the offset from the start of $B[j]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements.

To perform $\text{predecessor}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in.

**Step 2** Compute the predecessor of $x$ in $B[i]$

(suitably adjusting the offset from the start of $B[i]$)

**Step 3** If $x$ has no predecessor in $B[i]$:

Compute $j = \text{predecessor}(i)$ in $C$

Compute the predecessor of $x$ in $B[j]$

(suitably adjusting the offset from the start of $B[j]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

To perform $\text{predecessor}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in

**Step 2** Compute the predecessor of $x$ in $B[i]$

(suitably adjusting the offset from the start of $B[i]$)

**Step 3** If $x$ has no predecessor in $B[i]$:  
Compute $j = \text{predecessor}(i)$ in $C$

Compute the predecessor of $x$ in $B[j]$

(suitably adjusting the offset from the start of $B[j]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

To perform $\text{predecessor}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in

**Step 2** Compute the predecessor of $x$ in $B[i]$

(suitably adjusting the offset from the start of $B[i]$)

**Step 3** If $x$ has no predecessor in $B[i]$:  
Compute $j = \text{predecessor}(i)$ in $C$
Compute the predecessor of $x$ in $B[j]$

(suitably adjusting the offset from the start of $B[j]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

To perform $\text{predecessor}(x)$:

Step 1 Determine which $B[i]$ the element $x$ belongs in

Step 2 Compute the predecessor of $x$ in $B[i]$

(suitably adjusting the offset from the start of $B[i]$)

Step 3 If $x$ has no predecessor in $B[i]$:  
Compute $j = \text{predecessor}(i)$ in $C$

Compute the predecessor of $x$ in $B[j]$

(suitably adjusting the offset from the start of $B[j]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements.

To perform $\text{predecessor}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in.

**Step 2** Compute the predecessor of $x$ in $B[i]$ (suitably adjusting the offset from the start of $B[i]$).

**Step 3** If $x$ has no predecessor in $B[i]$:
- Compute $j = \text{predecessor}(i)$ in $C$.
- Compute the predecessor of $x$ in $B[j]$ (suitably adjusting the offset from the start of $B[j]$).
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

To perform $\text{predecessor}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in

**Step 2** Compute the predecessor of $x$ in $B[i]$ 

(suitably adjusting the offset from the start of $B[i]$)

**Step 3** If $x$ has no predecessor in $B[i]$: 

Compute $j = \text{predecessor}(i)$ in $C$

Compute the predecessor of $x$ in $B[j]$

(suitably adjusting the offset from the start of $B[j]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

To perform $\text{predecessor}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in

**Step 2** Compute the predecessor of $x$ in $B[i]$

(suitably adjusting the offset from the start of $B[i]$)

**Step 3** If $x$ has no predecessor in $B[i]$:  
Compute $j = \text{predecessor}(i)$ in $C$
Compute the predecessor of $x$ in $B[j]$

(suitably adjusting the offset from the start of $B[j]$)
Attempt 4: Recursion

Split the universe \( U \) into \( \sqrt{u} \) blocks each associated with \( \sqrt{u} \) elements

To perform \texttt{predecessor}(x):

\textbf{Step 1} Determine which \( B[i] \) the element \( x \) belongs in

\textbf{Step 2} Compute the \texttt{predecessor} of \( x \) in \( B[i] \)

\hspace{1cm} (suitably adjusting the offset from the start of \( B[i] \))

\textbf{Step 3} If \( x \) has no \texttt{predecessor} in \( B[i] \):

\hspace{1cm} Compute \( j = \texttt{predecessor}(i) \) in \( C \)

\hspace{1cm} Compute the \texttt{predecessor} of \( x \) in \( B[j] \)

\hspace{1cm} (suitably adjusting the offset from the start of \( B[j] \))
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

To perform $\text{predecessor}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in

**Step 2** Compute the predecessor of $x$ in $B[i]$ (suitably adjusting the offset from the start of $B[i]$)

**Step 3** If $x$ has no predecessor in $B[i]$

Compute $j = \text{predecessor}(i)$ in $C$

Compute the predecessor of $x$ in $B[j]$ (suitably adjusting the offset from the start of $B[j]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

$C$


$\sqrt{u} \quad \sqrt{u} \quad \sqrt{u} \quad \ldots \quad \sqrt{u}$
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements.

The operations lookup, delete and successor can all also be defined in a similar, recursive manner.
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements.

The operations lookup, delete and successor can all also be defined in a similar, recursive manner.

How efficient are the operations?
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

To perform $\text{add}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in

(this takes $O(1)$ time with a little bit twiddling)

**Step 2** If $B[i]$ is empty, $\text{add} \ i$ to $C$

**Step 3** $\text{add} \ x$ to $B[i]$ (suitably adjusting the offset from the start of $B[i]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements.

To perform add($x$):

**Step 1** Determine which $B[i]$ the element $x$ belongs in (this takes $O(1)$ time with a little bit twiddling)

**Step 2** If $B[i]$ is empty, add $i$ to $C$

**Step 3** add $x$ to $B[i]$ (suitably adjusting the offset from the start of $B[i]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements.

To perform $\text{add}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in.

(\text{this takes } O(1) \text{ time with a little bit twiddling})

**Step 2** If $B[i]$ is empty, add $i$ to $C$

**Step 3** add $x$ to $B[i]$

(suitably adjusting the offset from the start of $B[i]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

To perform $\text{predecessor}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in

**Step 2** Compute the predecessor of $x$ in $B[i]$ (suitably adjusting the offset from the start of $B[i]$)

**Step 3** If $x$ has no predecessor in $B[i]$:
- Compute $j = \text{predecessor}(i)$ in $C$
- Compute the predecessor of $x$ in $B[j]$ (suitably adjusting the offset from the start of $B[j]$)
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements.

The operations lookup, delete and successor can all also be defined in a similar, recursive manner.

How efficient are the operations?
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

\[
\begin{array}{cccc}
\end{array}
\]

The operations lookup, delete and successor can all also be defined in a similar, recursive manner.

How efficient are the operations?

The add operation makes up to two recursive calls and the predecessor operation makes up to three.
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

The operations lookup, delete and successor can all also be defined in a similar, recursive manner

How efficient are the operations?

The add operation makes up to two recursive calls and the predecessor operation makes up to three

Each recursive call could in turn make multiple recursive calls...
Attempt 4: Recursion

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements.

The operations lookup, delete and successor can all also be defined in a similar, recursive manner.

How efficient are the operations?

The add operation makes up to two recursive calls and the predecessor operation makes up to three.

Each recursive call could in turn make multiple recursive calls…

This could get out of hand!
A closer look at predecessor

To perform \( \text{predecessor}(x) \):

**Step 1** Determine which \( B[i] \) the element \( x \) belongs in

**Step 2** Compute the \text{predecessor} of \( x \) in \( B[i] \)

**Step 3** If \( x \) has no \text{predecessor} in \( B[i] \):
- Compute \( j = \text{predecessor}(i) \) in \( C \)
- Return the \text{predecessor} of \( x \) in \( B[j] \)
A closer look at predecessor

**Observation 1:** if $x$ has a predecessor in $B[i]$ we only make one recursive call

To perform $\text{predecessor}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in

**Step 2** Compute the predecessor of $x$ in $B[i]$  

**Step 3** If $x$ has no predecessor in $B[i]$:
- Compute $j = \text{predecessor}(i)$ in $C$
- Return the predecessor of $x$ in $B[j]$
A closer look at predecessor

**Observation 1**: if $x$ has a predecessor in $B[i]$ we only make one recursive call

To perform $\text{predecessor}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in

**Step 2** Compute the predecessor of $x$ in $B[i]$

**Step 3** If $x$ has no predecessor in $B[i]$:  
Compute $j = \text{predecessor}(i)$ in $C$  
Return the predecessor of $x$ in $B[j]$
A closer look at predecessor

**Observation 1:** if $x$ has a predecessor in $B[i]$ we only make one recursive call

Let $x$ have a predecessor in $B[i]$ iff $x \geq$ the minimum in $B[i]$

To perform $\text{predecessor}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in
**Step 2** Compute the predecessor of $x$ in $B[i]$

**Step 3** If $x$ has no predecessor in $B[i]$:
- Compute $j = \text{predecessor}(i)$ in $C$
- Return the predecessor of $x$ in $B[j]$
A closer look at predecessor

**Observation 1:** if $x$ has a predecessor in $B[i]$ we only make one recursive call

To perform $\text{predecessor}(x)$:

**Step 1** Determine which $B[i]$ the element $x$ belongs in

**Step 2** Compute the predecessor of $x$ in $B[i]

**Step 3** If $x < \text{the minimum in } B[i]$:

- Compute $j = \text{predecessor}(i)$ in $C$
- Return the predecessor of $x$ in $B[j]$
A closer look at predecessor

**Observation 1:** if \(x\) has a predecessor in \(B[i]\) we only make one recursive call

\[ x \text{ has a predecessor in } B[i] \text{ iff } x \geq \text{ the minimum in } B[i] \]

To perform \(\text{predecessor}(x)\):

**Step 1** Determine which \(B[i]\) the element \(x\) belongs in

**Step 2** If \(x \geq \text{ the minimum in } B[i]\):
   - Return the predecessor of \(x\) in \(B[i]\)

**Step 3** If \(x < \text{ the minimum in } B[i]\):
   - Compute \(j = \text{predecessor}(i)\) in \(C\)
   - Return the predecessor of \(x\) in \(B[j]\)
A closer look at predecessor

Observation 1: if \( x \) has a predecessor in \( B[i] \) we only make one recursive call

\[
\text{\( x \) has a predecessor in \( B[i] \) iff}
\text{\( x \geq \text{the minimum in } B[i] \)}
\]

To perform \( \text{predecessor}(x) \):

**Step 1** Determine which \( B[i] \) the element \( x \) belongs in

**Step 2** If \( x \geq \text{the minimum in } B[i] \):

- Return the predecessor of \( x \) in \( B[i] \)

**Step 3** If \( x < \text{the minimum in } B[i] \):

- Compute \( j = \text{predecessor}(i) \) in \( C \)
- Return the predecessor of \( x \) in \( B[j] \)

Now we make at most two recursive calls
A closer look at predecessor

Observation 1: if $x$ has a predecessor in $B[i]$ we only make one recursive call

$x$ has a predecessor in $B[i]$ iff $x \geq$ the minimum in $B[i]$

To perform $\text{predecessor}(x)$:

Step 1 Determine which $B[i]$ the element $x$ belongs in

Step 2 If $x \geq$ the minimum in $B[i]$:  
Return the predecessor of $x$ in $B[i]$  
Now we make at most two recursive calls

Step 3 If $x <$ the minimum in $B[i]$:  
Compute $j = \text{predecessor}(i)$ in $C$  
(ignoring finding the minimum)

Return the predecessor of $x$ in $B[j]$
A closer look at predecessor

To perform \textit{predecessor}(x):

\textbf{Step 1} Determine which $B[i]$ the element $x$ belongs in

\textbf{Step 2} If $x \geq$ the minimum in $B[i]$:

Return the \textit{predecessor} of $x$ in $B[i]$

\textbf{Step 3} If $x <$ the minimum in $B[i]$:

Compute $j = \text{predecessor}(i)$ in $C$

Return the \textit{predecessor} of $x$ in $B[j]$
A closer look at predecessor

To perform predecessor($x$):

**Step 1** Determine which $B[i]$ the element $x$ belongs in

**Step 2** If $x \geq$ the minimum in $B[i]$:  
Return the predecessor of $x$ in $B[i]$

**Step 3** If $x <$ the minimum in $B[i]$:  
Compute $j = \text{predecessor}(i)$ in $C$  
Return the predecessor of $x$ in $B[j]$

we need to get rid of one of these recursive calls
A closer look at predecessor

To perform \( \text{predecessor}(x) \):

**Step 1** Determine which \( B[i] \) the element \( x \) belongs in

**Step 2** If \( x \geq \) the minimum in \( B[i] \):
- Return the predecessor of \( x \) in \( B[i] \)

**Step 3** If \( x < \) the minimum in \( B[i] \):
- Compute \( j = \text{predecessor}(i) \) in \( C \)
- Return the predecessor of \( x \) in \( B[j] \)

To perform \( \text{predecessor}(x) \):

1. Determine which \( B[i] \) the element \( x \) belongs in
2. If \( x \geq \) the minimum in \( B[i] \):
   - Return the predecessor of \( x \) in \( B[i] \)
3. If \( x < \) the minimum in \( B[i] \):
   - Compute \( j = \text{predecessor}(i) \) in \( C \)
   - Return the predecessor of \( x \) in \( B[j] \)
A closer look at predecessor

**Observation 2:** In **Step 3**, the predecessor of $x$ in $B[j]$ is the maximum in $B[j]$

![Diagram showing the predecessor concept]

To perform `predecessor(x)`:

**Step 1** Determine which $B[i]$ the element $x$ belongs in

**Step 2** If $x \geq$ the minimum in $B[i]$:  
Return the predecessor of $x$ in $B[i]$

**Step 3** If $x <$ the minimum in $B[i]$:  
Compute $j = \text{predecessor}(i)$ in $C$  
Return the predecessor of $x$ in $B[j]$
A closer look at predecessor

Observation 2: In Step 3, the predecessor of \( x \) in \( B[j] \) is the maximum in \( B[j] \)

To perform \( \text{predecessor}(x) \):

**Step 1** Determine which \( B[i] \) the element \( x \) belongs in

**Step 2** If \( x \geq \) the minimum in \( B[i] \):

Return the predecessor of \( x \) in \( B[i] \)

**Step 3** If \( x < \) the minimum in \( B[i] \):

Compute \( j = \text{predecessor}(i) \) in \( C \)

Return the predecessor of \( x \) in \( B[j] \)
A closer look at predecessor

Observation 2: In **Step 3**, the **predecessor** of \( x \) in \( B[j] \) is the maximum in \( B[j] \)

\[ j = 2 \]

\( i \)

\( C \)


To perform \( \text{predecessor}(x) \):

**Step 1** Determine which \( B[i] \) the element \( x \) belongs in

**Step 2** If \( x \geq \) the minimum in \( B[i] \):
   
   Return the **predecessor** of \( x \) in \( B[i] \)

**Step 3** If \( x < \) the minimum in \( B[i] \):
   
   Compute \( j = \text{predecessor}(i) \) in \( C \)
   
   Return the maximum in \( B[j] \)
A closer look at predecessor

**Observation 2:** In **Step 3**, the predecessor of \( x \) in \( B[j] \) is the maximum in \( B[j] \)

To perform \( \text{predecessor}(x) \):

**Step 1** Determine which \( B[i] \) the element \( x \) belongs in

**Step 2** If \( x \geq \) the minimum in \( B[i] \):

Return the predecessor of \( x \) in \( B[i] \)

**Step 3** If \( x < \) the minimum in \( B[i] \):

Compute \( j = \text{predecessor}(i) \) in \( C \)

Return the maximum in \( B[j] \)

Now we make exactly one recursive call (ignoring finding the min/max)
Finally: van Emde Boas Trees

So that we can find the min/max quickly we store them separately...
Finally: van Emde Boas Trees

So that we can find the min/max quickly we store them separately...
Finally: van Emde Boas Trees

So that we can find the min/max quickly we store them separately...

Remember that each $B[i]$ and $C$ are also vEB (van Emde Boas) trees each over the universe $\{1, 2, 3, \ldots \sqrt{u}\}$
Finally: van Emde Boas Trees

So that we can find the min/max quickly we store them separately...

Remember that each $B[i]$ and $C$ are also vEB (van Emde Boas) trees
each over the universe $\{1, 2, 3, \ldots \sqrt{u}\}$

In particular $B[i]$ also stores its min/max elements separately

so recovering the minimum or maximum in $B[i]$ (or $C$) takes $O(1)$ time.
Finally: van Emde Boas Trees

So that we can find the min/max quickly we store them separately...

Remember that each $B[i]$ and $C$ are also vEB (van Emde Boas) trees
each over the universe $\{1, 2, 3, \ldots \sqrt{u}\}$

In particular $B[i]$ also stores its min/max elements separately
so recovering the minimum or maximum in $B[i]$ (or $C$) takes $O(1)$ time

There is one more important thing, the minimum is not also stored in $B[i]$
Finally: van Emde Boas Trees

So that we can find the min/max quickly we store them separately...

Remember that each $B[i]$ and $C$ are also vEB (van Emde Boas) trees each over the universe $\{1, 2, 3, \ldots \sqrt{u}\}$

In particular $B[i]$ also stores its min/max elements separately

so recovering the minimum or maximum in $B[i]$ (or $C$) takes $O(1)$ time

There is one more important thing, the minimum is not also stored in $B[i]$
**Finally:** van Emde Boas Trees

So that we can find the min/max quickly we store them separately...

Remember that each $B[i]$ and $C$ are also vEB (van Emde Boas) trees each over the universe $\{1, 2, 3, \ldots \sqrt{u}\}$

In particular $B[i]$ also stores its min/max elements separately

*so recovering the minimum or maximum in $B[i]$ (or $C$) takes $O(1)$ time*

There is one more important thing, the minimum is **not** also stored in $B[i]$
**Finally: van Emde Boas Trees**

So that we can find the min/max quickly we store them separately...

Remember that each $B[i]$ and $C$ are also vEB (van Emde Boas) trees each over the universe \( \{1, 2, 3, \ldots, \sqrt{u}\} \)

In particular $B[i]$ also stores its min/max elements separately

so recovering the minimum or maximum in $B[i]$ (or $C$) takes $O(1)$ time

There is one more important thing, the minimum is **not** also stored in $B[i]$
this allows us to avoid making multiple recursive calls when adding an element
Another look at add

To perform \( \text{add}(x) \):

**Step 1** Determine which \( B[i] \) the element \( x \) belongs in

**Step 2** If \( B[i] \) is empty, add \( i \) to \( C \) and set the \( \text{min} \) and \( \text{max} \) in \( B[i] \) to \( x \) *adjusting the offset*

**Step 3** If \( B[i] \) is not empty, add \( x \) to \( B[i] \)
Another look at add

To perform \text{add}(x):

\textbf{Step 1} Determine which \(B[i]\) the element \(x\) belongs in

\textbf{Step 2} If \(B[i]\) is empty, \text{add} \(i\) to \(C\)
and set the min and max in \(B[i]\) to \(x\) (adjusting the offset)

\textbf{Step 3} If \(B[i]\) is not empty, \text{add} \(x\) to \(B[i]\)
Another look at add

To perform \text{add}(x):

\textbf{Step 1} Determine which \( B[i] \) the element \( x \) belongs in

\textbf{Step 2} If \( B[i] \) is empty, add \( i \) to \( C \) and set the \text{min} and \text{max} in \( B[i] \) to \( x \) \textit{(adjusting the offset)}

\textbf{Step 3} If \( B[i] \) is not empty, add \( x \) to \( B[i] \)
Another look at add

To perform add\( (x) \):

**Step 1** Determine which \( B[i] \) the element \( x \) belongs in

**Step 2** If \( B[i] \) is empty, add \( i \) to \( C \)

and set the min and max in \( B[i] \) to \( x \) (adjusting the offset)

**Step 3** If \( B[i] \) is not empty, add \( x \) to \( B[i] \)

we make one recursive call
Another look at add

To perform add($x$):

**Step 1** Determine which $B[i]$ the element $x$ belongs in

**Step 2** If $B[i]$ is empty, add $i$ to $C$

and set the min and max in $B[i]$ to $x$ (*adjusting the offset*)

**Step 3** If $B[i]$ is not empty, add $x$ to $B[i]$
Another look at add

To perform \( \text{add}(x) \):

**Step 1** Determine which \( B[i] \) the element \( x \) belongs in

**Step 2** If \( B[i] \) is empty, add \( i \) to \( C \)

and set the min and max in \( B[i] \) to \( x \) (*adjusting the offset*)

**Step 3** If \( B[i] \) is not empty, add \( x \) to \( B[i] \)
Another look at add

To perform add($x$):

**Step 1** Determine which $B[i]$ the element $x$ belongs in.

**Step 2** If $B[i]$ is empty, add $i$ to $C$ and set the min and max in $B[i]$ to $x$ (adjusting the offset).

**Step 3** If $B[i]$ is not empty, add $x$ to $B[i]$. 

we make one recursive call
Another look at add

To perform \( \text{add}(x) \):

**Step 1** Determine which \( B[i] \) the element \( x \) belongs in

**Step 2** If \( B[i] \) is empty, add \( i \) to \( C \) and set the min and max in \( B[i] \) to \( x \) (adjusting the offset)

**Step 3** If \( B[i] \) is not empty, add \( x \) to \( B[i] \)
Another look at add

To perform \( \text{add}(x) \):

**Step 1** Determine which \( B[i] \) the element \( x \) belongs in

**Step 2** If \( B[i] \) is empty, add \( i \) to \( C \)

and set the \( \text{min} \) and \( \text{max} \) in \( B[i] \) to \( x \) *(adjusting the offset)*

**Step 3** If \( B[i] \) is not empty, add \( x \) to \( B[i] \)
Another look at add

To perform \texttt{add}(x):

\begin{itemize}
  \item \textbf{Step 1} Determine which \(B[i]\) the element \(x\) belongs in.
  \item \textbf{Step 2} If \(B[i]\) is empty, add \(i\) to \(C\) and set the \texttt{min} and \texttt{max} in \(B[i]\) to \(x\) \textit{(adjusting the offset)}.
  \item \textbf{Step 3} If \(B[i]\) is not empty, add \(x\) to \(B[i]\).
\end{itemize}

\text{we make one recursive call}

\text{this is not recursive}
Another look at add

To perform \( \text{add}(x) \):

**Step 1** Determine which \( B[i] \) the element \( x \) belongs in

**Step 2** If \( B[i] \) is empty, add \( i \) to \( C \) and set the min and max in \( B[i] \) to \( x \) (*adjusting the offset*)

**Step 3** If \( B[i] \) is not empty, add \( x \) to \( B[i] \)

We make one recursive call

This is not recursive

The min is only stored here
Another look at add

To perform add(x):

Step 1 Determine which $B[i]$ the element $x$ belongs in

Step 2 If $B[i]$ is empty, add $i$ to $C$ and set the min and max in $B[i]$ to $x$ (adjusting the offset)

Step 3 If $B[i]$ is not empty, add $x$ to $B[i]$
Another look at add

To perform add\((x)\):

**Step 1** Determine which \(B[i]\) the element \(x\) belongs in.

**Step 2** If \(B[i]\) is empty, add \(i\) to \(C\) and set the min and max in \(B[i]\) to \(x\) *(adjusting the offset)*.

**Step 3** If \(B[i]\) is not empty, add \(x\) to \(B[i]\).

Now we always make exactly one recursive call but what happens when the min/max change?
Another look at add

To perform add($x$): 

**Step 1** Determine which $B[i]$ the element $x$ belongs in.

**Step 2** If $B[i]$ is empty, add $i$ to $C$ and set the min and max in $B[i]$ to $x$ (*adjusting the offset*).

**Step 3** If $B[i]$ is not empty, add $x$ to $B[i]$. 

Now we always make exactly one recursive call but what happens when the min/max change?
Another look at add

To perform add($x$):

Step 1 Determine which $B[i]$ the element $x$ belongs in

Step 2 If $B[i]$ is empty, add $i$ to $C$ and set the min and max in $B[i]$ to $x$ (adjusting the offset)

Step 3 If $B[i]$ is not empty, add $x$ to $B[i]$

Now we always make exactly one recursive call but what happens when the min/max change?

the min is only stored here
Another look at add

To perform add(x):

Step 0 If \(x < \text{min}\) then swap \(x\) and \(\text{min}\)

Step 1 Determine which \(B[i]\) the element \(x\) belongs in

Step 2 If \(B[i]\) is empty, add \(i\) to \(C\)

and set the min and max in \(B[i]\) to \(x\) (adjusting the offset)

Step 3 If \(B[i]\) is not empty, add \(x\) to \(B[i]\)

Step 4 Update the max

Now we always make exactly one recursive call

but what happens when the min/max change?
Time Complexity

The min is only stored here.

$\min$ $\sqrt{u}$ $\max$

The min is only stored here.
We have seen that the operations add and predecessor can be defined so that they make only one recursive call.
We have seen that the operations *add* and *predecessor* can be defined so that they make only one recursive call.

The operations *lookup*, *delete* and *successor* can all also be defined in a similar, recursive manner so that they make only one recursive call.
We have seen that the operations \texttt{add} and \texttt{predecessor} can be defined so that they make only one recursive call.

The operations \texttt{lookup}, \texttt{delete} and \texttt{successor} can all also be defined in a similar, recursive manner so that they make only one recursive call.

\textit{How long do the operations take?}
the min is only stored \textbf{here}

\text{min} \quad \sqrt{u} \quad \sqrt{u} \quad \sqrt{u} \quad \cdots \quad \sqrt{u} \quad u \quad \text{max}

\begin{itemize}
  \item $B[1]$
  \item $B[2]$
  \item $B[3]$
  \item $B[\sqrt{u}]$
\end{itemize}

$C$
Let $T(u)$ be the time complexity of the add operation (where $u$ is the universe size)
Let $T(u)$ be the time complexity of the add operation (where $u$ is the universe size)

We have that, $T(u) = T(\sqrt{u}) + O(1)$
Let $T(u)$ be the time complexity of the add operation

(\text{where } u \text{ is the universe size})

We have that, \[ T(u) = T(\sqrt{u}) + O(1) \]

Using substitution and the master method you can show that…

\[ T(u) = O(\log \log u) \]
Let $T(u)$ be the time complexity of the predecessor operation (where $u$ is the universe size).

We have that, $T(u) = T(\sqrt{u}) + O(1)$

Using substitution and the master method you can show that... $T(u) = O(\log \log u)$
Let $T(u)$ be the time complexity of the \textit{predecessor} operation (where $u$ is the universe size).

We have that, $T(u) = T(\sqrt{u}) + O(1)$

Using substitution and the master method you can show that… $T(u) = O(\log \log u)$

\textit{this holds for all the operations}
Space Complexity

the min is only stored here

C


min max

37 483
Let $Z(u)$ be the space used by a vEB tree over a universe of size $u$. 

The min is only stored here.
Let \( Z(u) \) be the space used by a vEB tree over a universe of size \( u \).

We have that, \( Z(u) = (\sqrt{u} + 1) \cdot Z(\sqrt{u}) + O(1) \).
Space Complexity

Let \( Z(u) \) be the space used by a vEB tree over a universe of size \( u \)

We have that, \( Z(u) = (\sqrt{u} + 1) \cdot Z(\sqrt{u}) + O(1) \)

If you solve this you get that\ldots \( Z(u) = O(u) \)
van Emde Boas Trees

The **van Emde Boas (vEB) tree**
stores a set $S$ of **integer keys** from a universe $U = \{1, 2, 3, 4 \ldots u\}$ (i.e. $u = |U|$).

**Five** operations are supported:

- **add($x$)**: Insert the integer $x$ into $S$ (where $x \in U$)
- **lookup($x$)**: Return **yes** if $x$ is in $S$, or **no** otherwise.
- **delete($x$)**: Remove $x$ from $S$
- **predecessor($k$)**: Return the **largest** integer $x$ in $S$ such that $x \leq k$
- **successor($k$)**: Return the **smallest** integer $x$ in $S$ such that $x \geq k$

![Diagram showing predecessor and successor operations]

All operations take $O(\log \log u)$ worst-case time
and the space used is $O(u)$
van Emde Boas Trees

The van Emde Boas (vEB) tree stores a set \( S \) of integer keys from a universe \( U = \{1, 2, 3, 4 \ldots u\} \) (i.e. \( u = |U| \)).

Five operations are supported:

- **add**\((x)\) Insert the integer \( x \) into \( S \) (where \( x \in U \))
- **lookup**\((x)\) Return yes if \( x \) is in \( S \), or no otherwise.
- **delete**\((x)\) Remove \( x \) from \( S \)
- **predecessor**\((k)\) Return the largest integer \( x \) in \( S \) such that \( x \leq k \)
- **successor**\((k)\) Return the smallest integer \( x \) in \( S \) such that \( x \geq k \)

All operations take \( O(\log \log u) \) worst case time and the space used is \( O(u) \)

The space can be improved to \( O(n) \) using hashing (see y-fast trees)