Advanced Algorithms – COMS31900

van Emde Boas trees

Raphaël Clifford

Slides by Benjamin Sach

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**

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...

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

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?

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,

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 amortised,

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

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 inflexibility?

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

Supporting more operations

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

Supporting more operations

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

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 $x \leqslant k$

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.

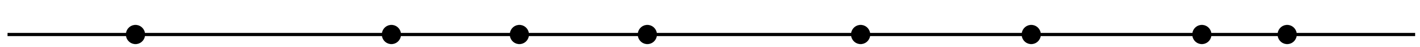
Three operations are supported:

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 $x \leqslant k$



Supporting more operations

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

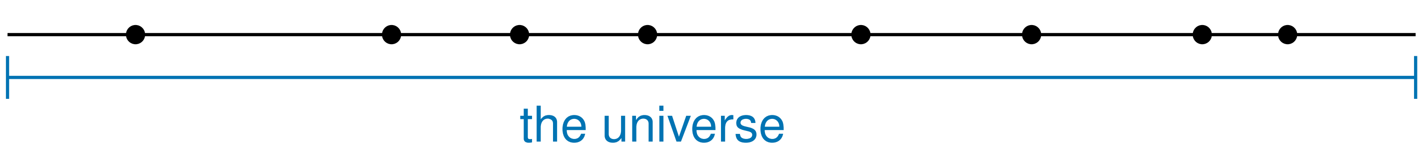
Three operations are supported:

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 $x \leqslant k$



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.

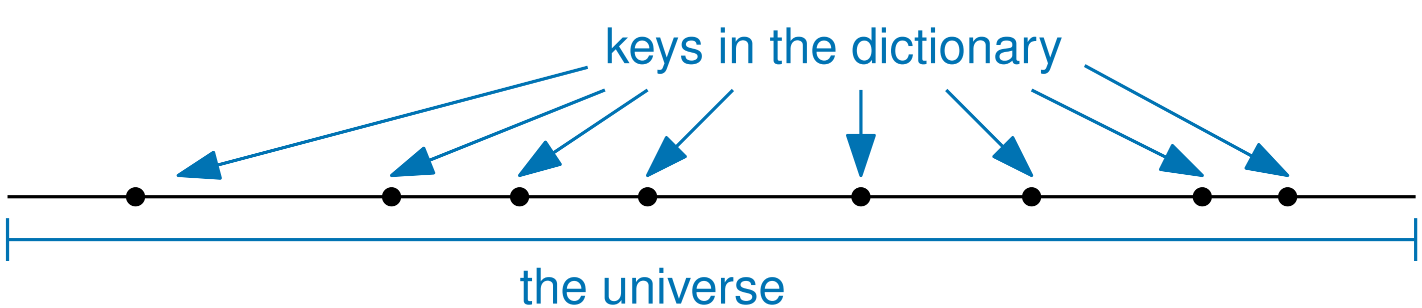
Three operations are supported:

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 $x \leqslant k$



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.

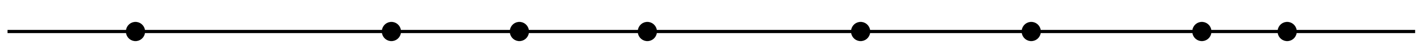
Three operations are supported:

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 $x \leqslant k$



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.

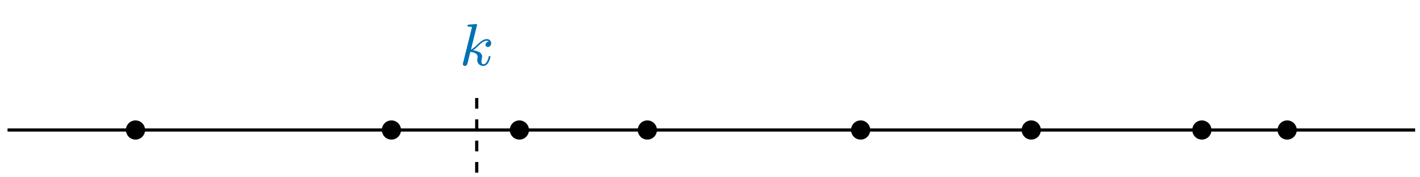
Three operations are supported:

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 $x \leqslant k$



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.

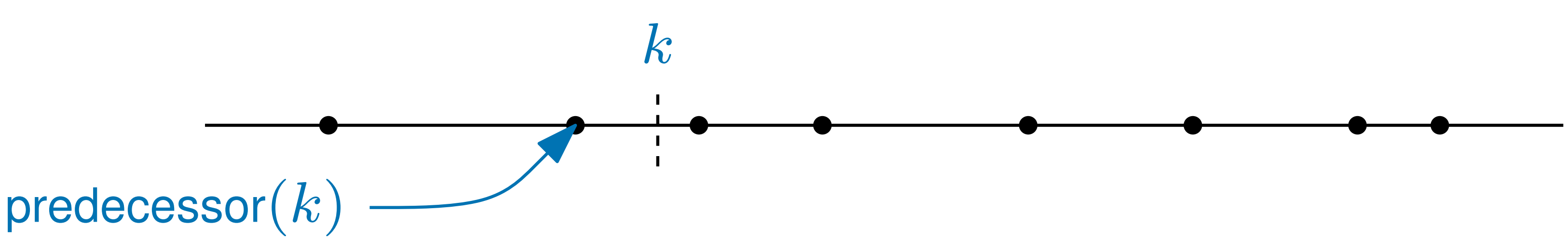
Three operations are supported:

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 $x \leqslant k$



Supporting more operations

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

Three operations are supported:

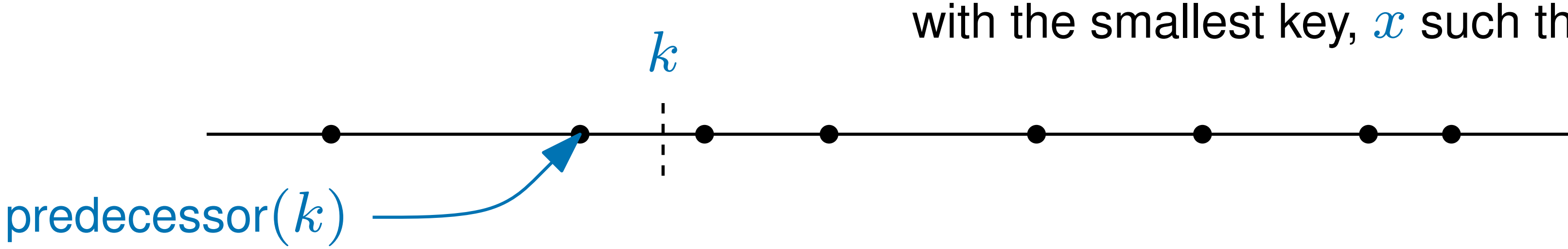
add(x, v)	Add the the pair (x, v) where $x \in U$ - the *universe*
lookup(x)	Return v if (x, v) is in dictionary, or NULL otherwise.
delete(x)	Remove pair (x, v) (assuming (x, v) is in the dictionary).

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 $x \leqslant k$

successor(k) - returns the (unique) element (x, v) in the dictionary with the smallest key, x such that $x \geqslant k$



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:

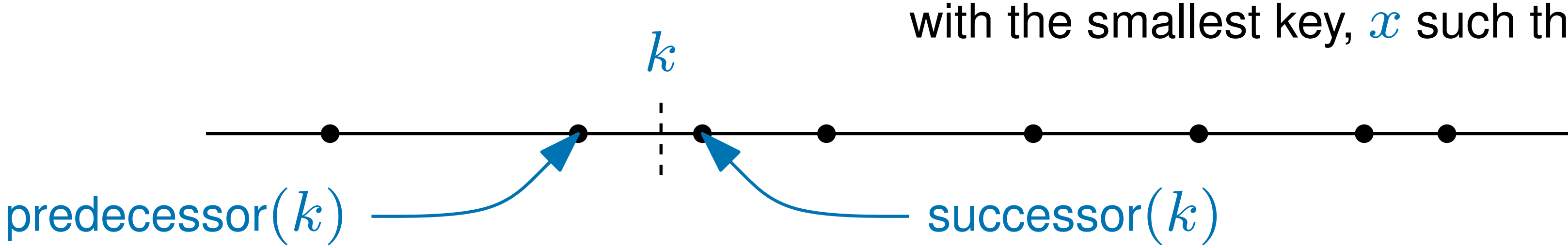
add(x, v)	Add the the pair (x, v) where $x \in U$ - the *universe*
lookup(x)	Return v if (x, v) is in dictionary, or NULL otherwise.
delete(x)	Remove pair (x, v) (assuming (x, v) is in the dictionary).

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 $x \leqslant k$

successor(k) - returns the (unique) element (x, v) in the dictionary with the smallest key, x such that $x \geqslant k$



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 $x \leqslant k$

successor(k) - returns the (unique) element (x, v) in the dictionary with the smallest key, x such that $x \geqslant k$

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:

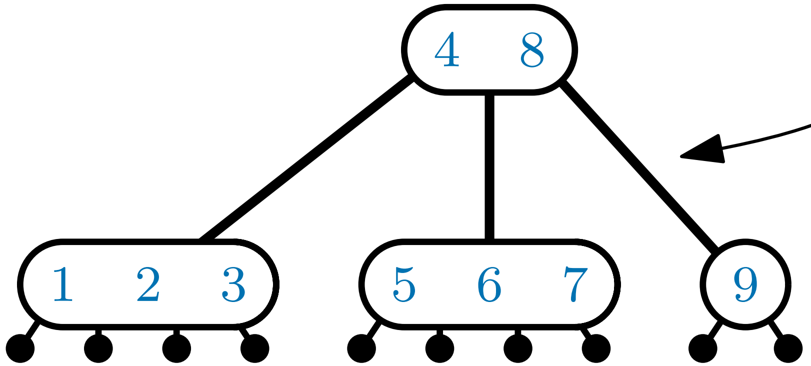
predecessor(k) - returns the (unique) element (x, v) in the dictionary
with the largest key, x such that $x \leqslant k$

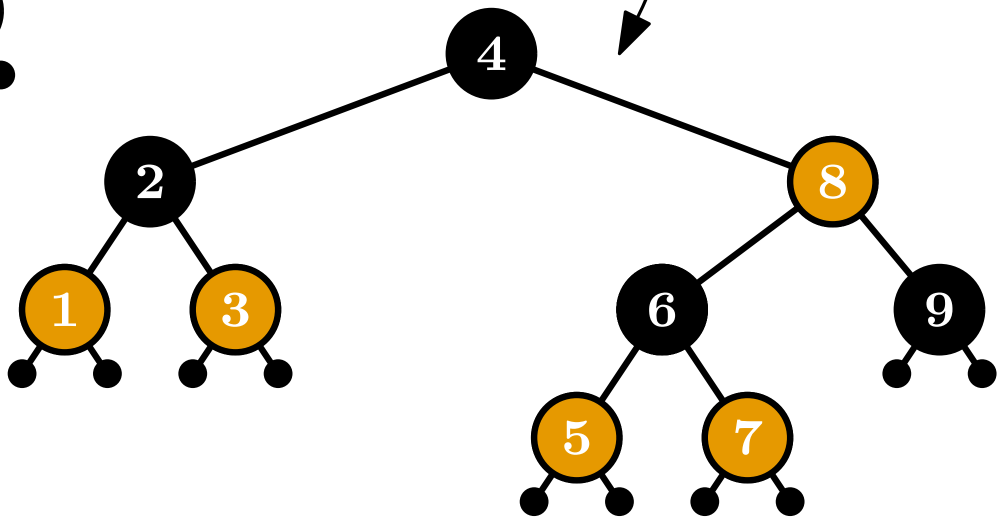
successor(k) - returns the (unique) element (x, v) in the dictionary
with the smallest key, x such that $x \geqslant k$

These are very natural operations that the **Hashing**-based solutions
that we have seen are very unsuited to

What could we use instead?

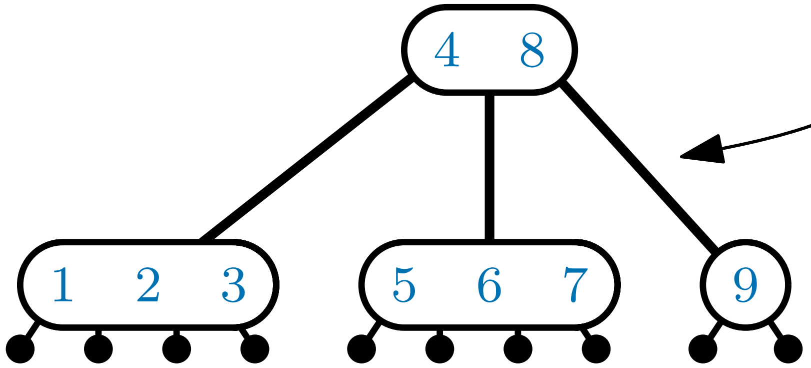
We could use a self-balancing binary search tree. . .
like a 2-3-4 tree, a red-**black** tree or an AVL tree

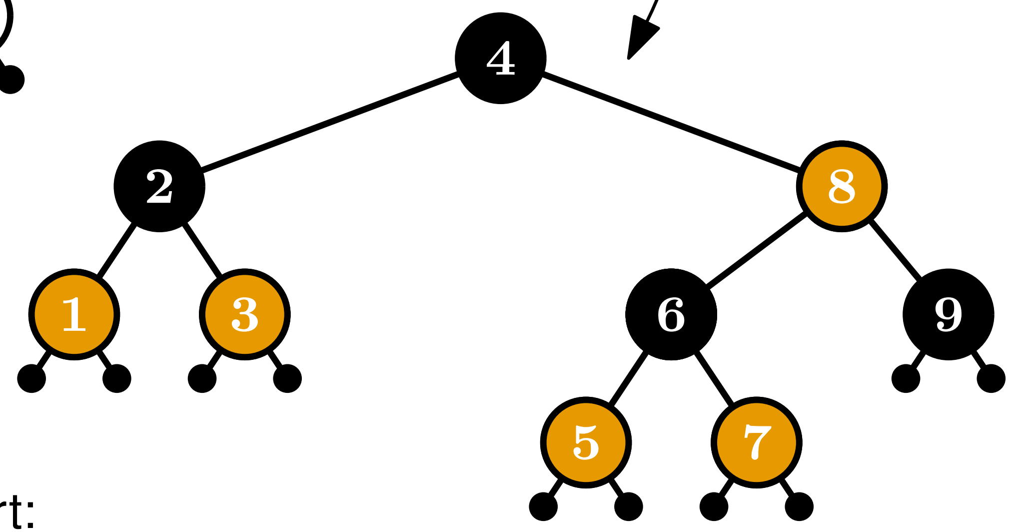




What could we use instead?

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





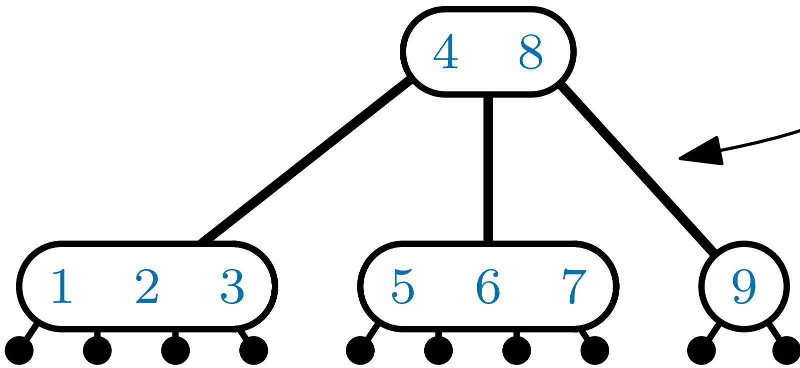
All three of these data structures support:

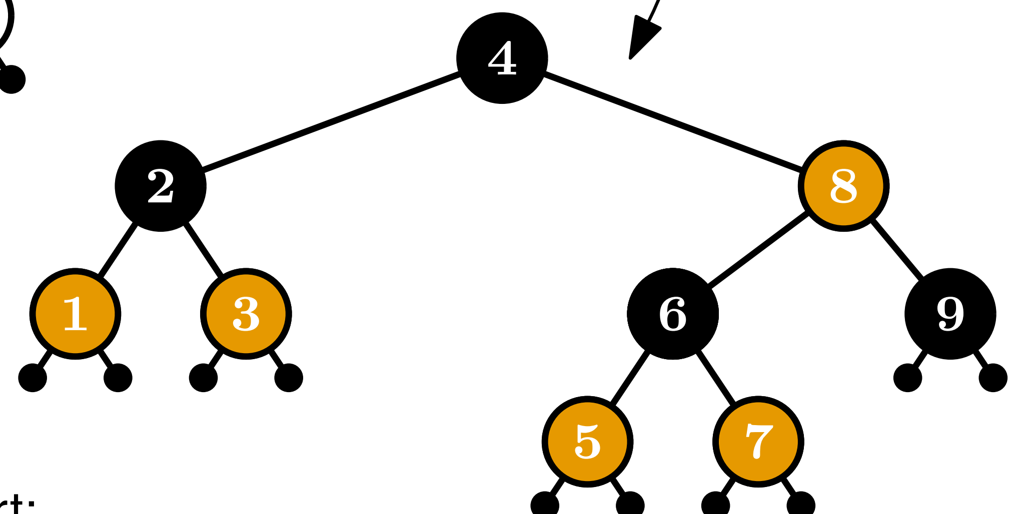
add(x, v),lookup(x), delete(x),predecessor(k) and successor(k)

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





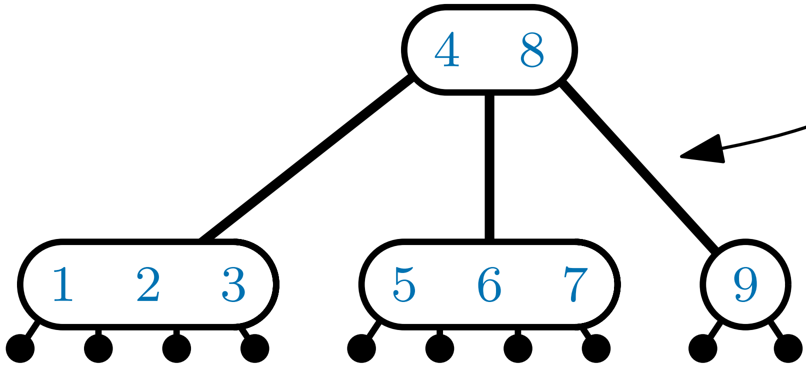
All three of these data structures support:

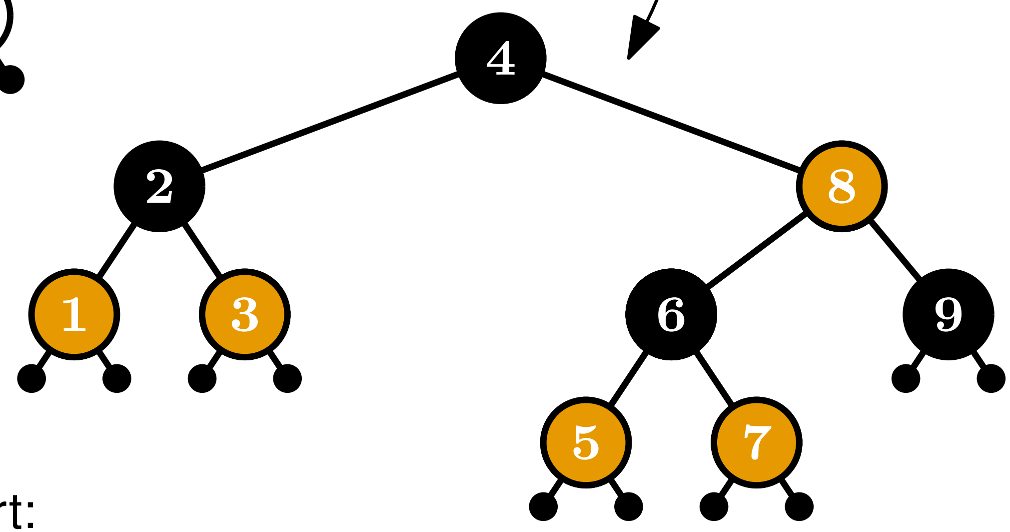
add(x, v),lookup(x), delete(x),predecessor(k) and successor(k)

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:

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

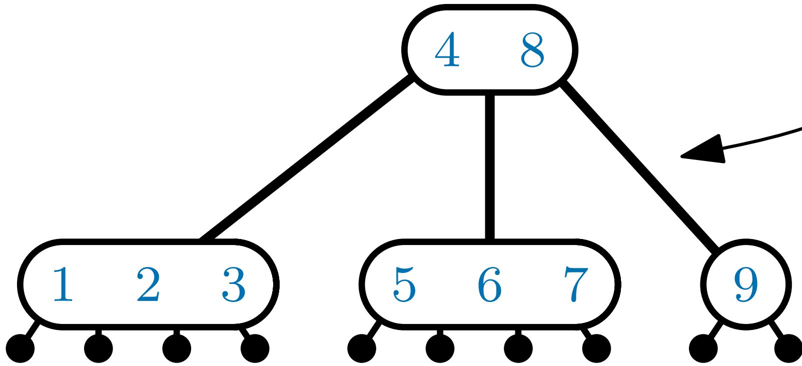
each in $O(\log n)$ worst case time and $O(n)$ space

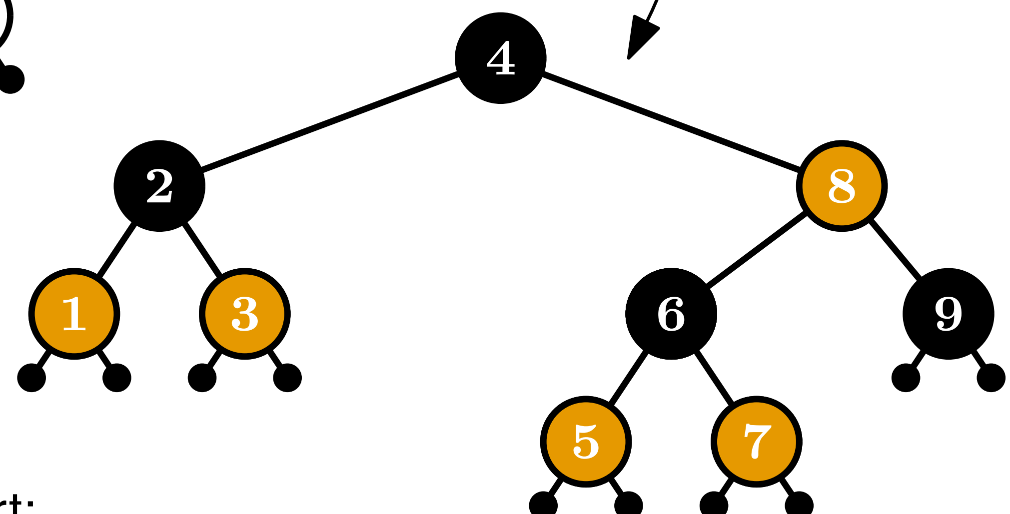
where n is the number of elements stored

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

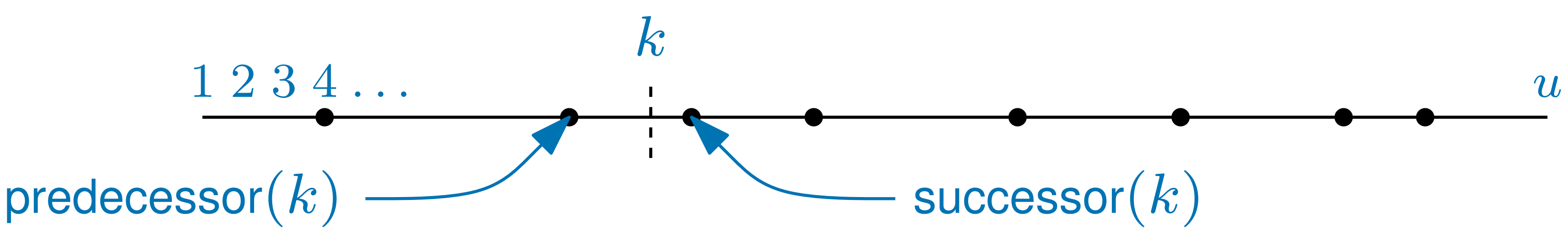
they are also *deterministic*

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 $U = \{1, 2, 3, 4 \dots u\}$ (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 $x \leqslant k$
successor(k)	Return the smallest integer x in S such that $x \geqslant k$

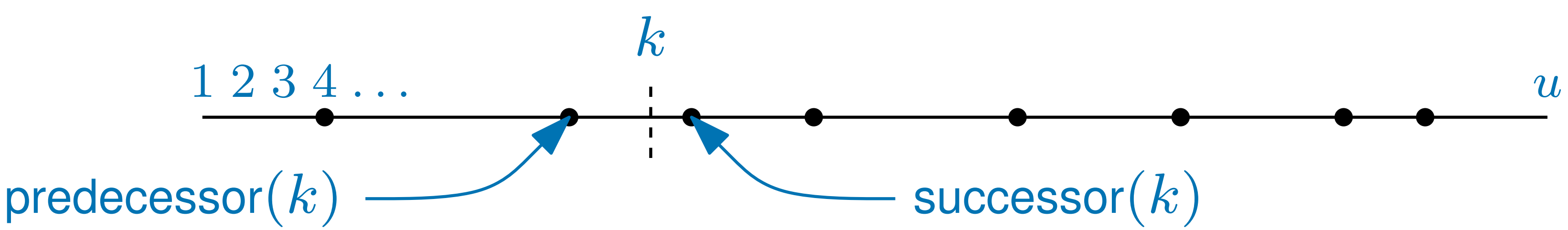


van Emde Boas Trees

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 $x \leqslant k$
successor(k)	Return the smallest integer x in S such that $x \geqslant k$



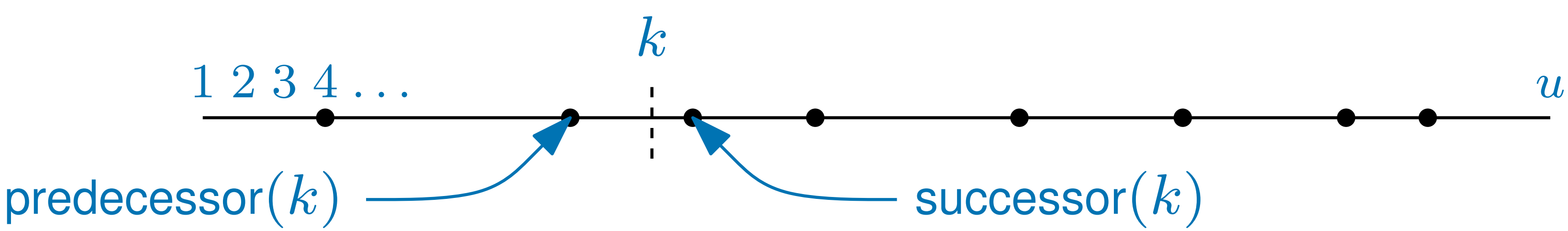
Warning: As stated the operations do not store any data (values) with the integers (keys)

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 $x \leqslant k$
successor(k)	Return the smallest integer x in S such that $x \geqslant k$



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

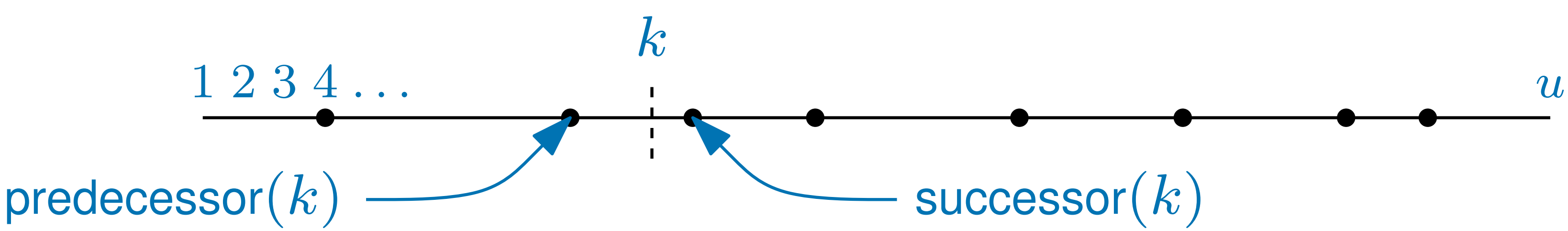
It is straightforward to extend the **van Emde Boas tree** to store (key, value) pairs
when the keys are integers from U

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 $U = \{1, 2, 3, 4 \dots u\}$ (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 $x \leqslant k$
successor(k)	Return the smallest integer x in S such that $x \geqslant k$



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

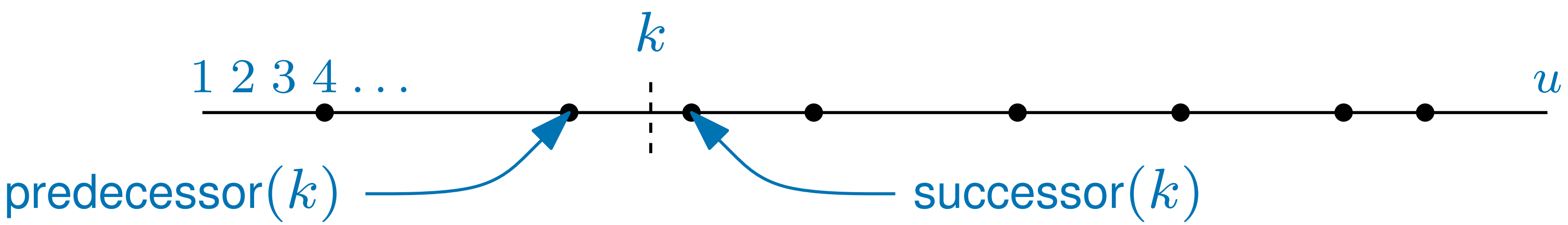
(but I think it's easier to think about like this)

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 $U = \{1, 2, 3, 4 \dots u\}$ (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 $x \leqslant k$
successor(k)	Return the smallest integer x in S such that $x \geqslant k$

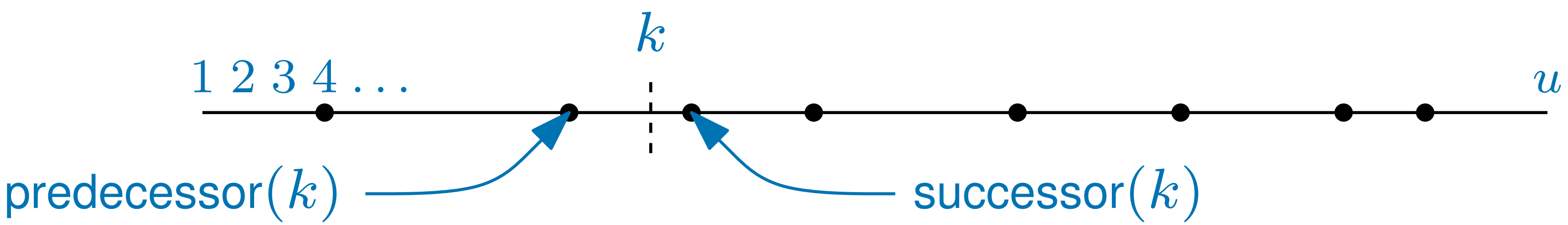


van Emde Boas Trees

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 $x \leqslant k$
successor(k)	Return the smallest integer x in S such that $x \geqslant k$



All operations will take $O(\log \log u)$ worst case time

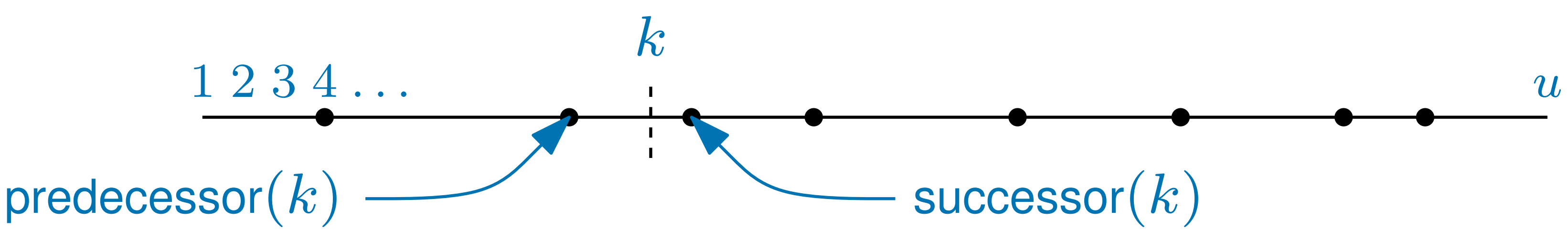
and the space used is $O(u)$

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 $U = \{1, 2, 3, 4 \dots u\}$ (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 $x \leqslant k$
successor(k)	Return the smallest integer x in S such that $x \geqslant k$



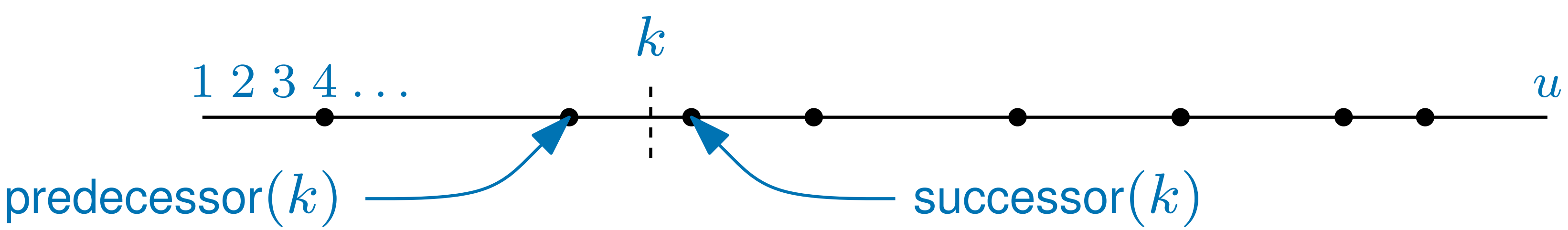
All operations will take $O(\log \log u)$ worst case time
and the space used is $O(u)$
and it is a deterministic data structure

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 $U = \{1, 2, 3, 4 \dots u\}$ (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 $x \leqslant k$
successor(k)	Return the smallest integer x in S such that $x \geqslant k$



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

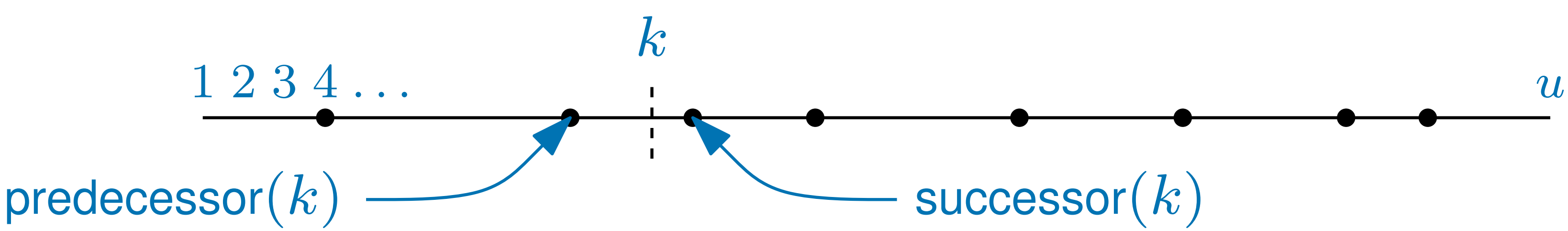
Example: If $U = \{1, 2, 3, 4 \dots 100 \cdot n\}$, you get $O(\log \log n)$ time and $O(n)$ space

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 $U = \{1, 2, 3, 4 \dots u\}$ (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 $x \leqslant k$
successor(k)	Return the smallest integer x in S such that $x \geqslant k$



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

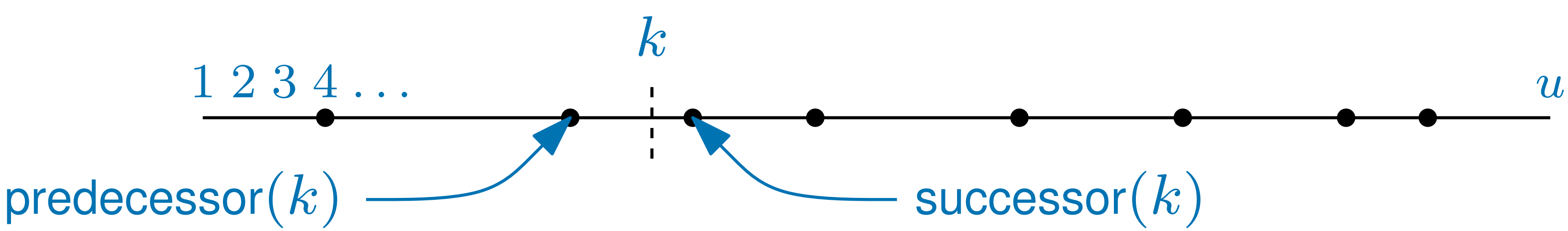
Example: If $U = \{1, 2, 3, 4 \dots n^2\}$, you get $O(\log \log n)$ time and $O(n^2)$ space

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 $x \leqslant k$
successor(k)	Return the smallest integer x in S such that $x \geqslant k$



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

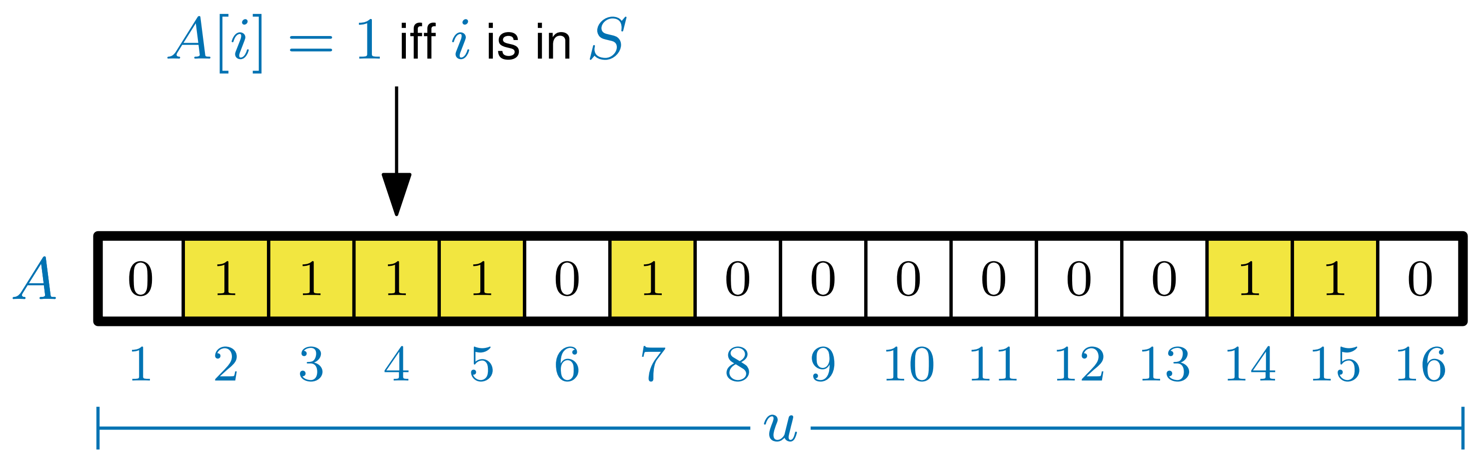
Build an array of length u...

A	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	16

$\underbrace{\hspace{10em}}_{u}$

Attempt 1: a big array

Build an array of length u...



Attempt 1: a big array

Build an array of length u...

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

A	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	16

$\underbrace{\qquad\qquad\qquad\qquad}_{u}$

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

Attempt 1: a big array

Build an array of length u...

add(12)

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

A	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	16

$\underbrace{\qquad\qquad\qquad\qquad}_{u}$

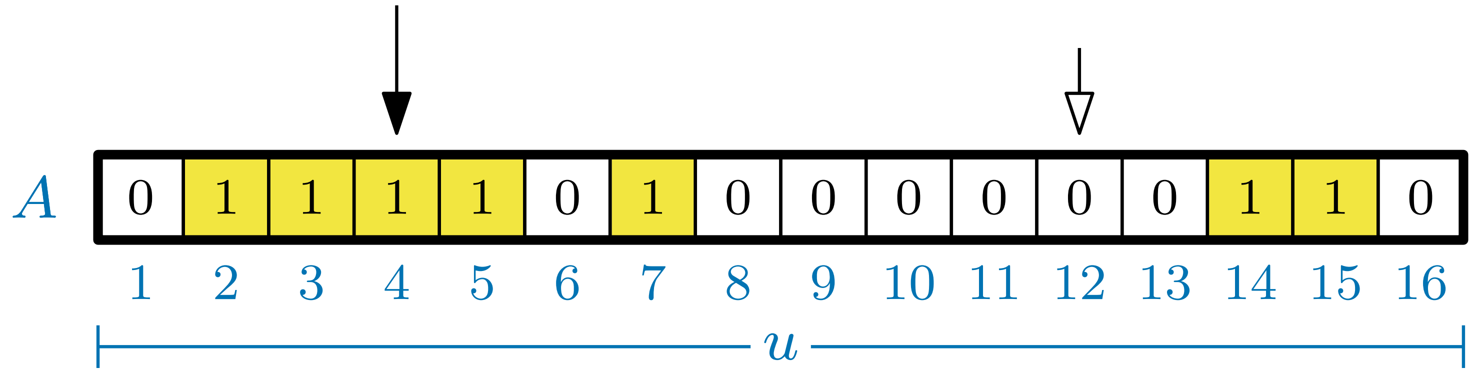
The operations add, delete and lookup all take $O(1)$ time.

Attempt 1: a big array

Build an array of length u...

add(12)

$A[i] = 1$ 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. . .

add(12)

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

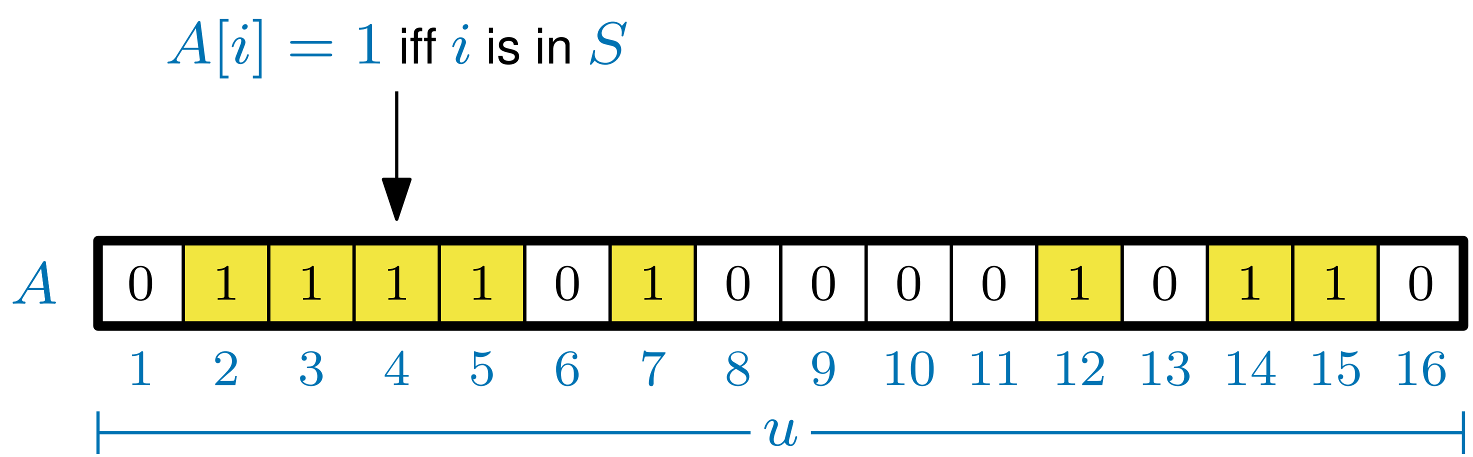
A	0	1	1	1	1	0	1	0	0	0	0	1	0	1	1	0
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

u

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

Attempt 1: a big array

Build an array of length u...



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

Attempt 1: a big array

Build an array of length u...

delete(14)

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

A	0	1	1	1	1	0	1	0	0	0	0	1	0	1	1	0
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

u

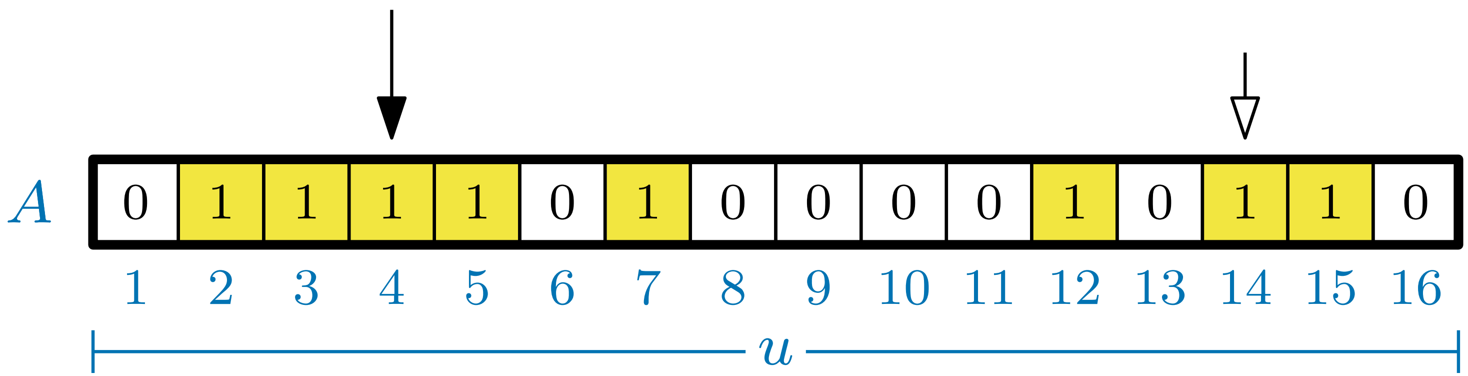
The operations add, delete and lookup all take $O(1)$ time.

Attempt 1: a big array

Build an array of length u...

delete(14)

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

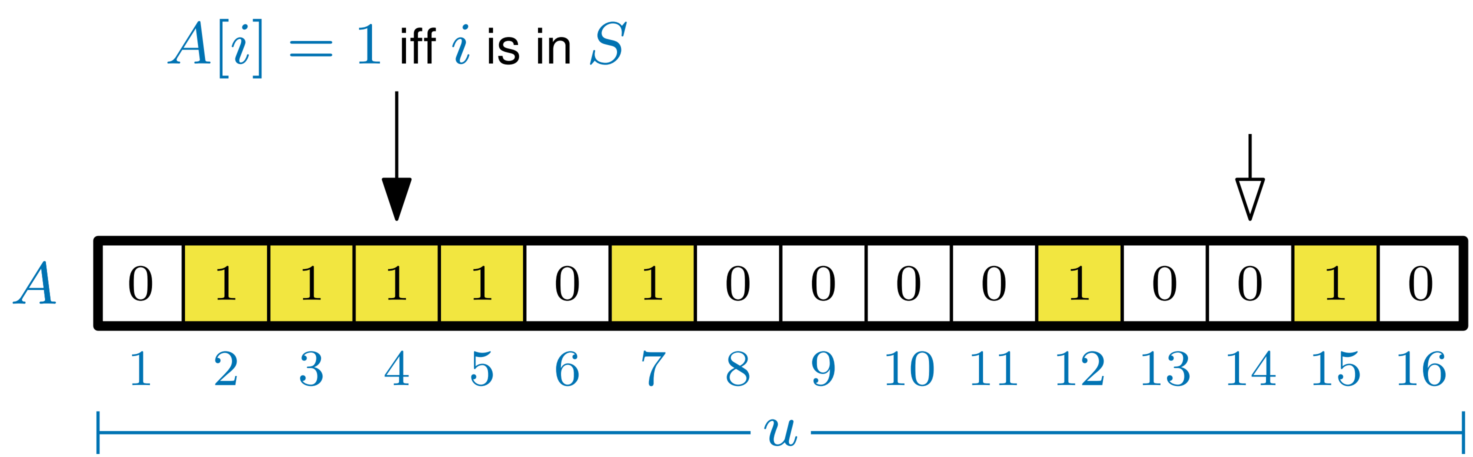


The operations add, delete and lookup all take $O(1)$ time.

Attempt 1: a big array

Build an array of length u...

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

A	0	1	1	1	1	0	1	0	0	0	0	1	0	0	1	0
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

u

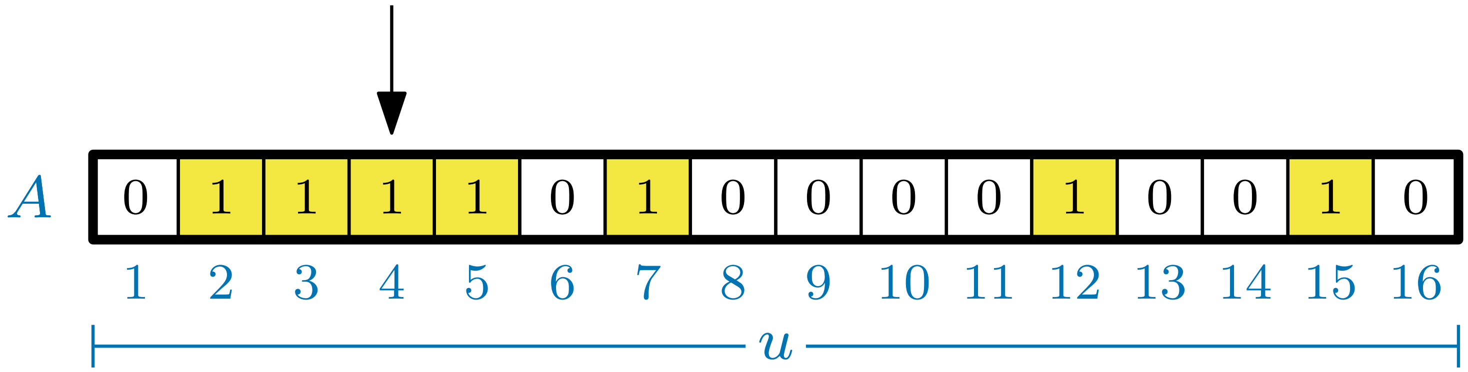
The operations add, delete and lookup all take $O(1)$ time.

Attempt 1: a big array

Build an array of length u...

lookup(11)

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



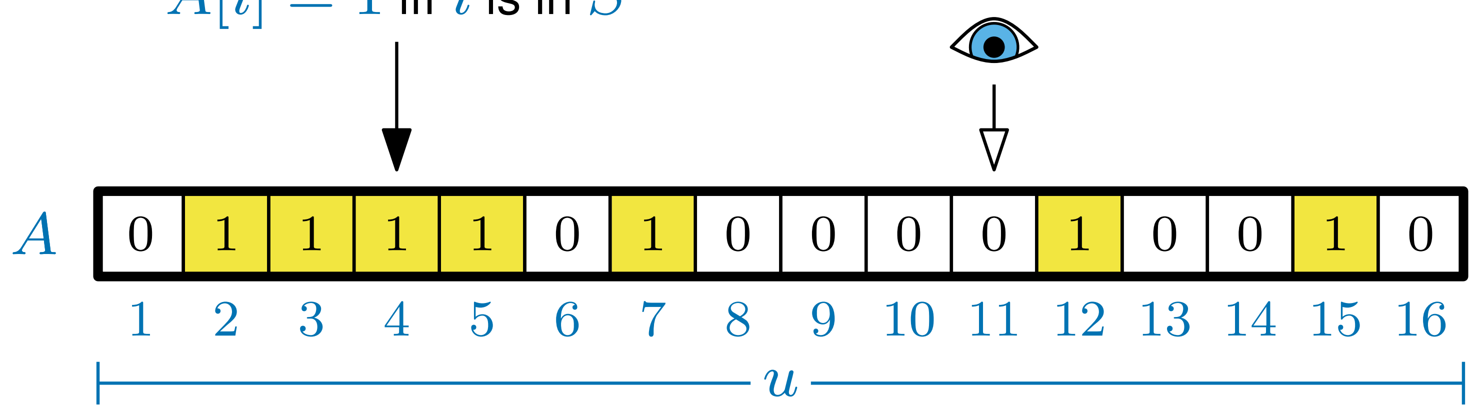
The operations add, delete and lookup all take $O(1)$ time.

Attempt 1: a big array

Build an array of length u...

lookup(11)

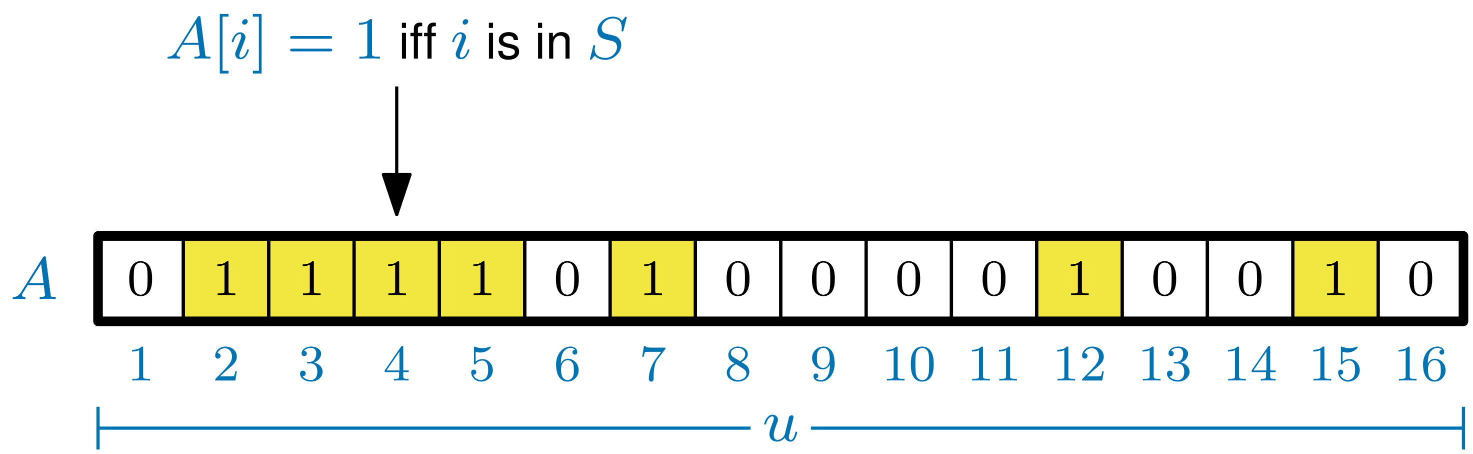
$A[i] = 1$ iff i is in S



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

Attempt 1: a big array

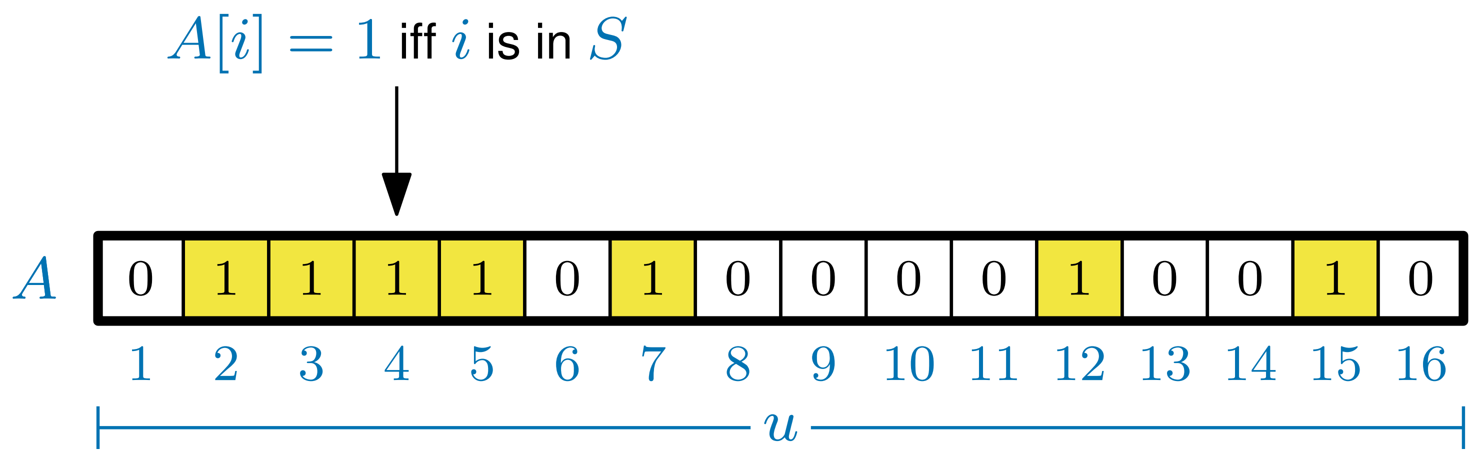
Build an array of length u...



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

Attempt 1: a big array

Build an array of length u...



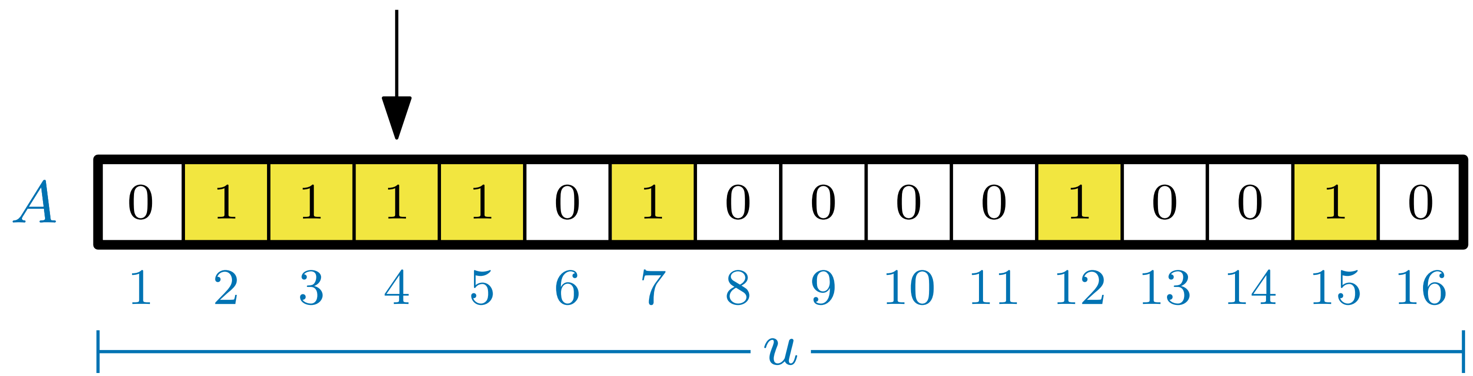
The operations add, delete and lookup all take $O(1)$ time.

...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!

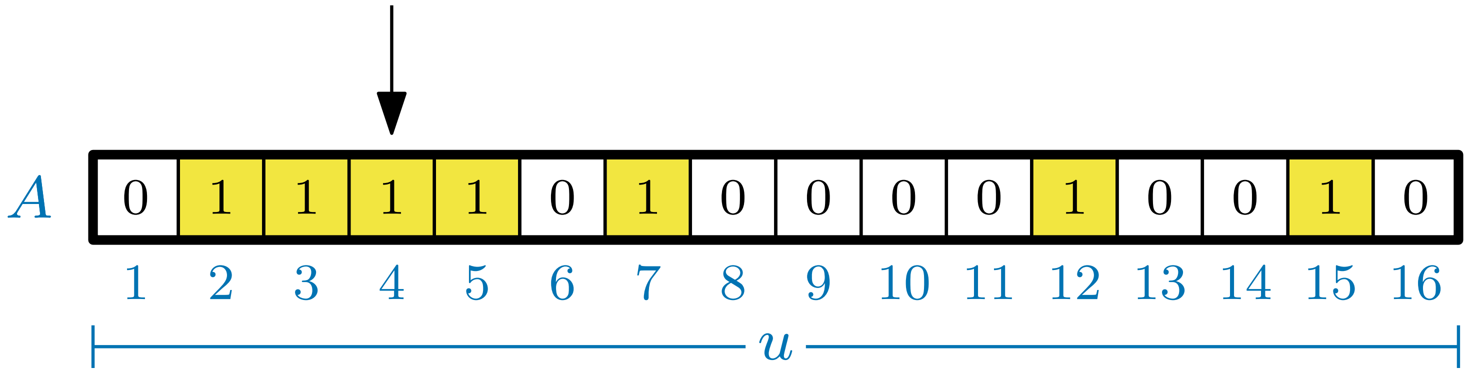
What about the predecessor operation?

Attempt 1: a big array

Build an array of length u. . .

predecessor(11)

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



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

. . . looks good so far!

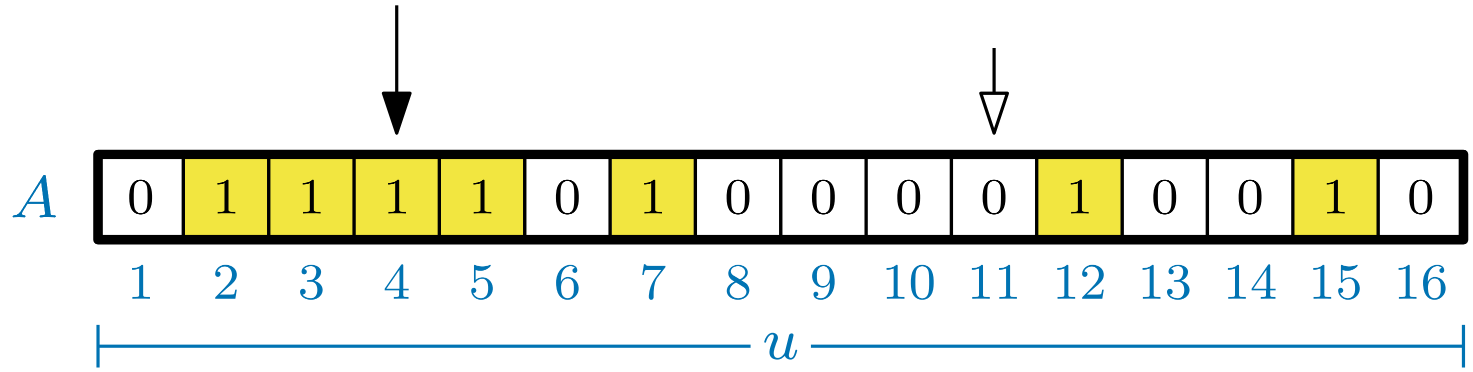
What about the predecessor operation?

Attempt 1: a big array

Build an array of length u...

predecessor(11)

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



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

...looks good so far!

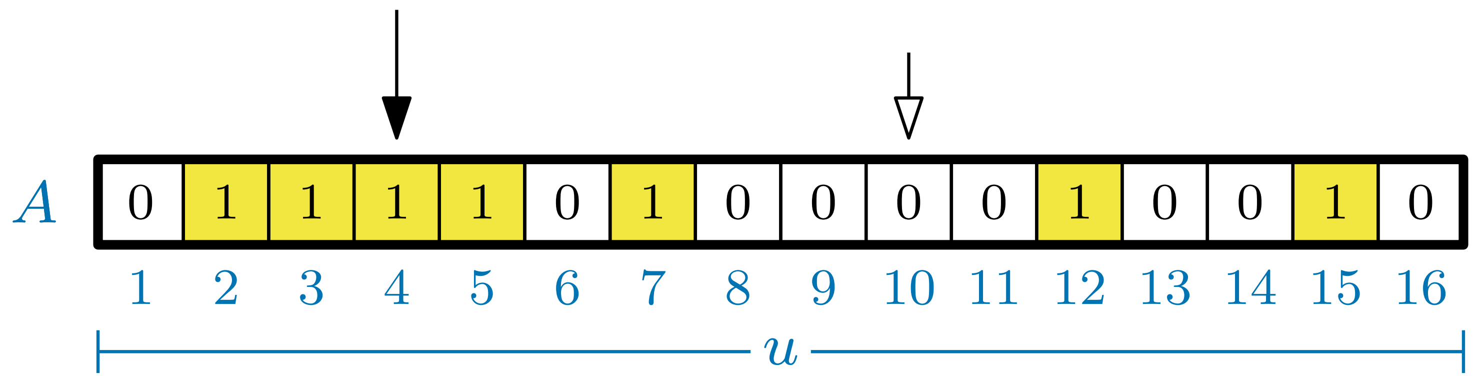
What about the predecessor operation?

Attempt 1: a big array

Build an array of length u...

predecessor(11)

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



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

...looks good so far!

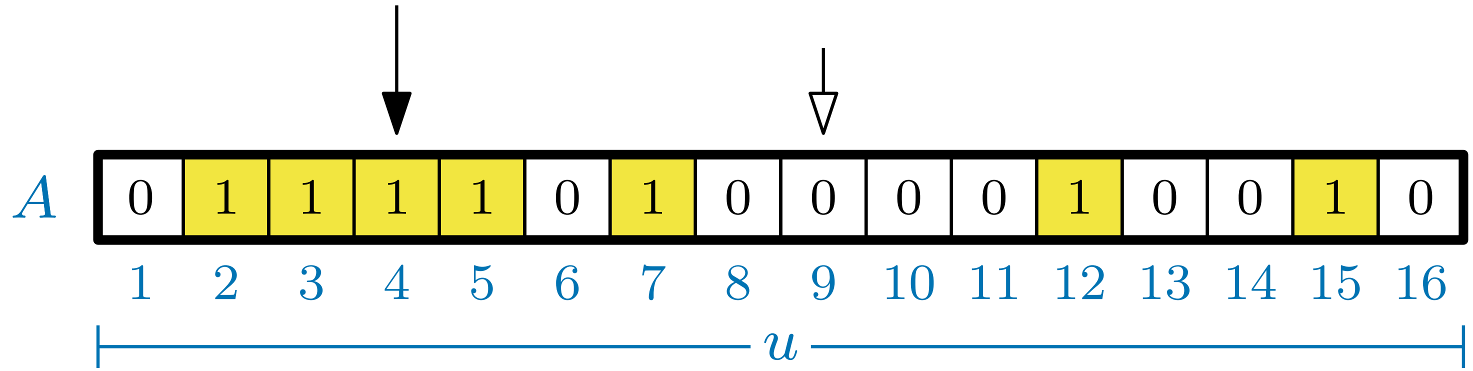
What about the predecessor operation?

Attempt 1: a big array

Build an array of length u...

predecessor(11)

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



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

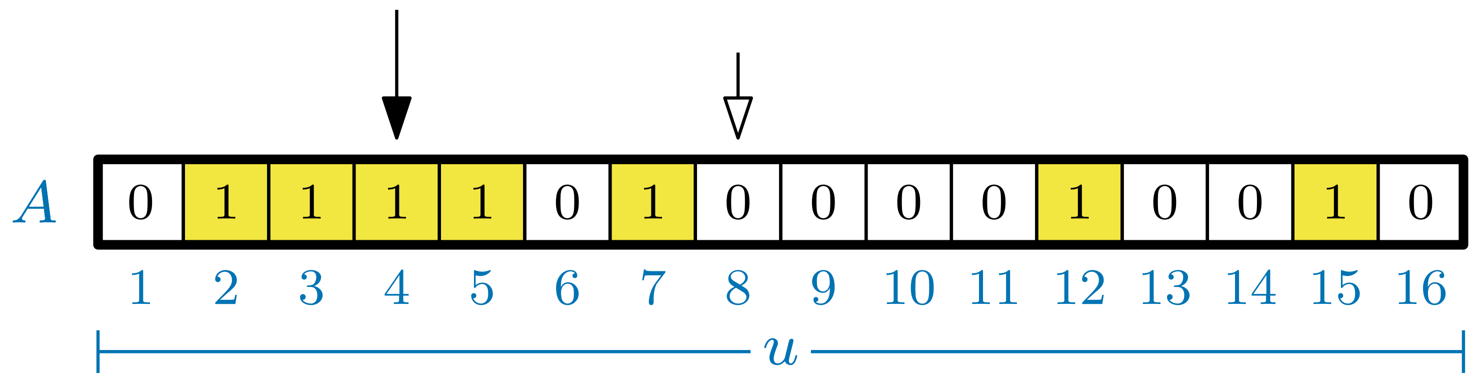
...looks good so far!

Attempt 1: a big array

Build an array of length u...

predecessor(11)

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



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

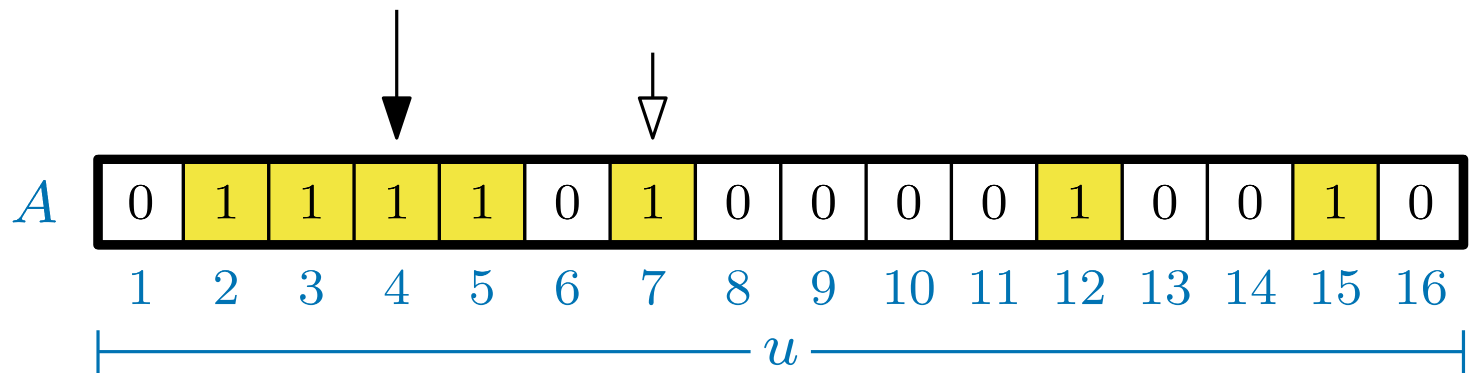
...looks good so far!

Attempt 1: a big array

Build an array of length u...

predecessor(11)

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



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

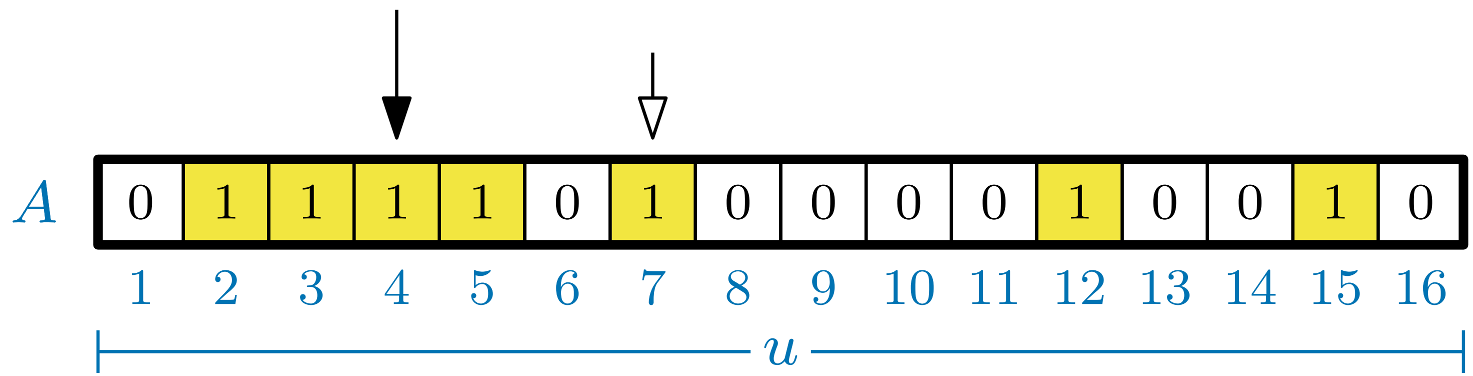
...looks good so far!

Attempt 1: a big array

Build an array of length u...

predecessor(11)

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



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

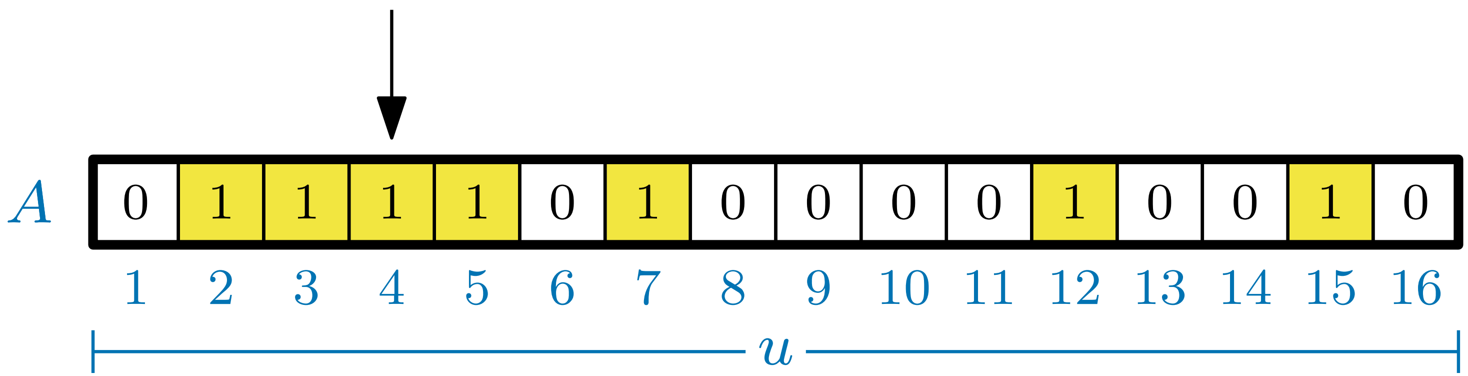
...looks good so far!

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.

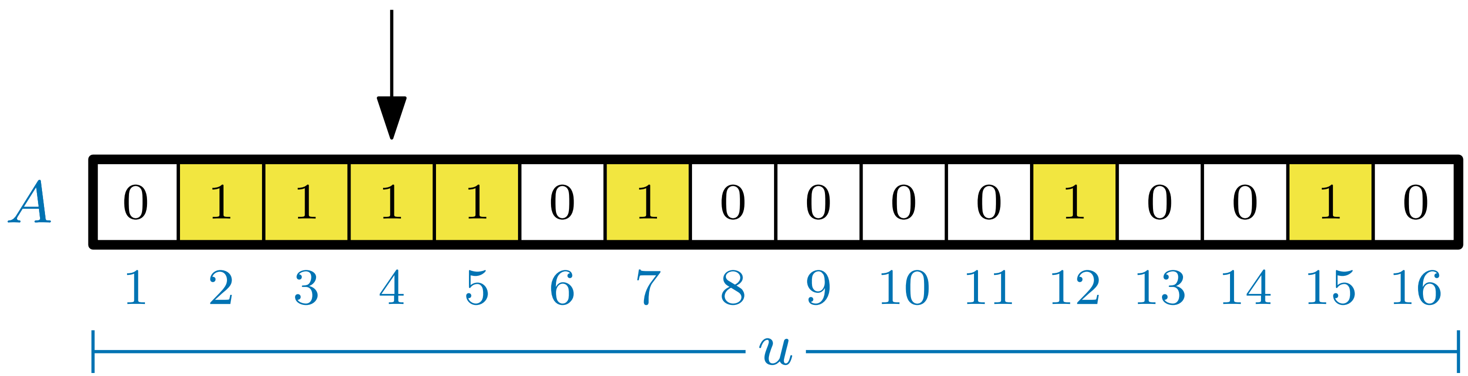
...looks good so far!

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

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 constant height tree

(on top of a big array)

A	0	1	1	1	1	0	1	0	0	0	0	1	0	0	1	0
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

u

Attempt 2: a constant height tree

(on top of a big array)

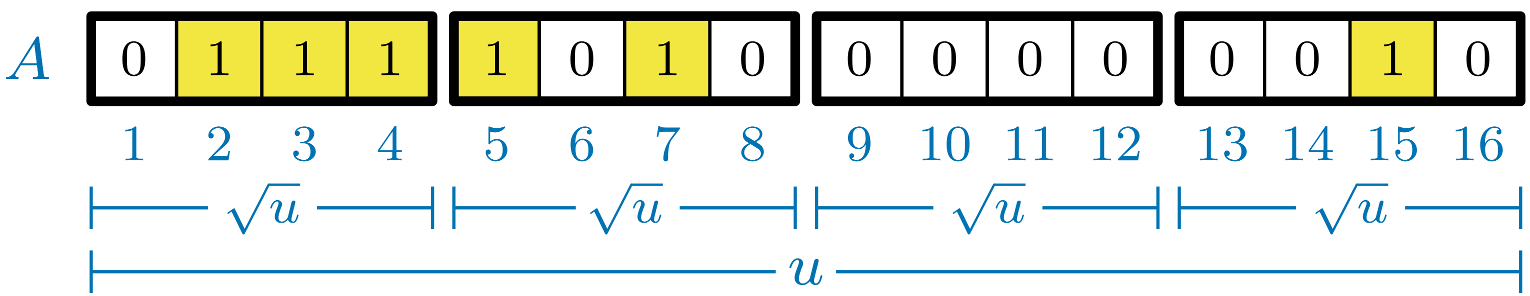
A	0	1	1	1	1	0	1	0	0	0	0	1	0	0	1	0
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

u

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

Attempt 2: a constant height tree

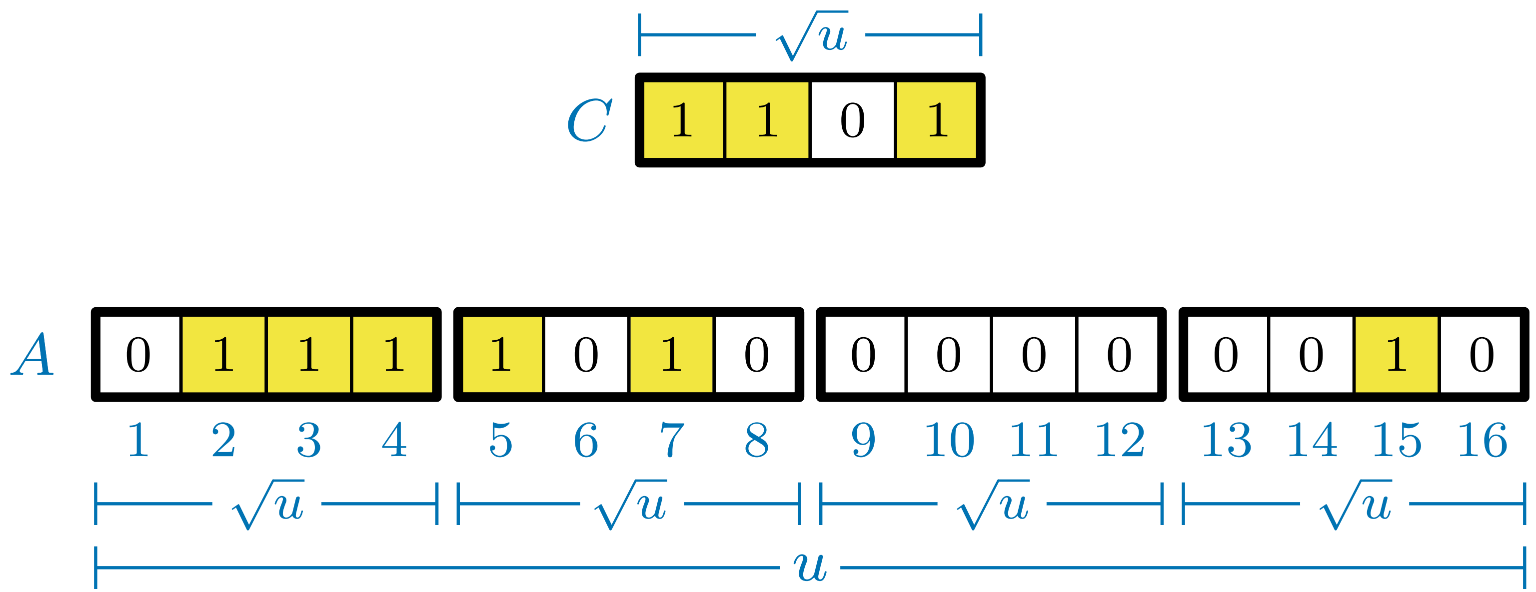
(on top of a big array)



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

Attempt 2: a constant height tree

(on top of a big array)

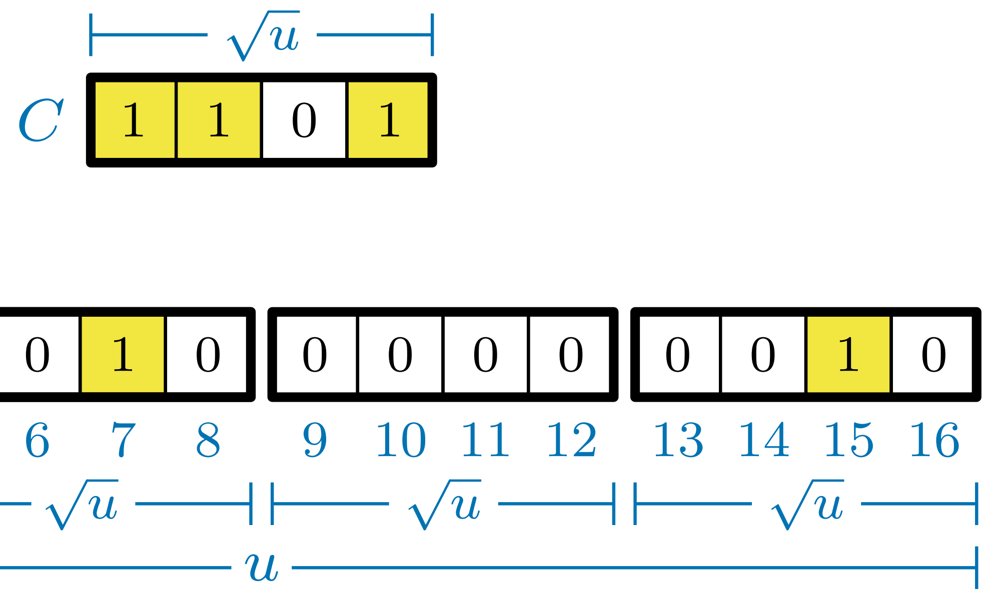


Split A into $\sqrt{u}$ *blocks* each containing $\sqrt{u}$ bits

Attempt 2: a constant height tree

(on top of a big array)

C is called the *summary* of A

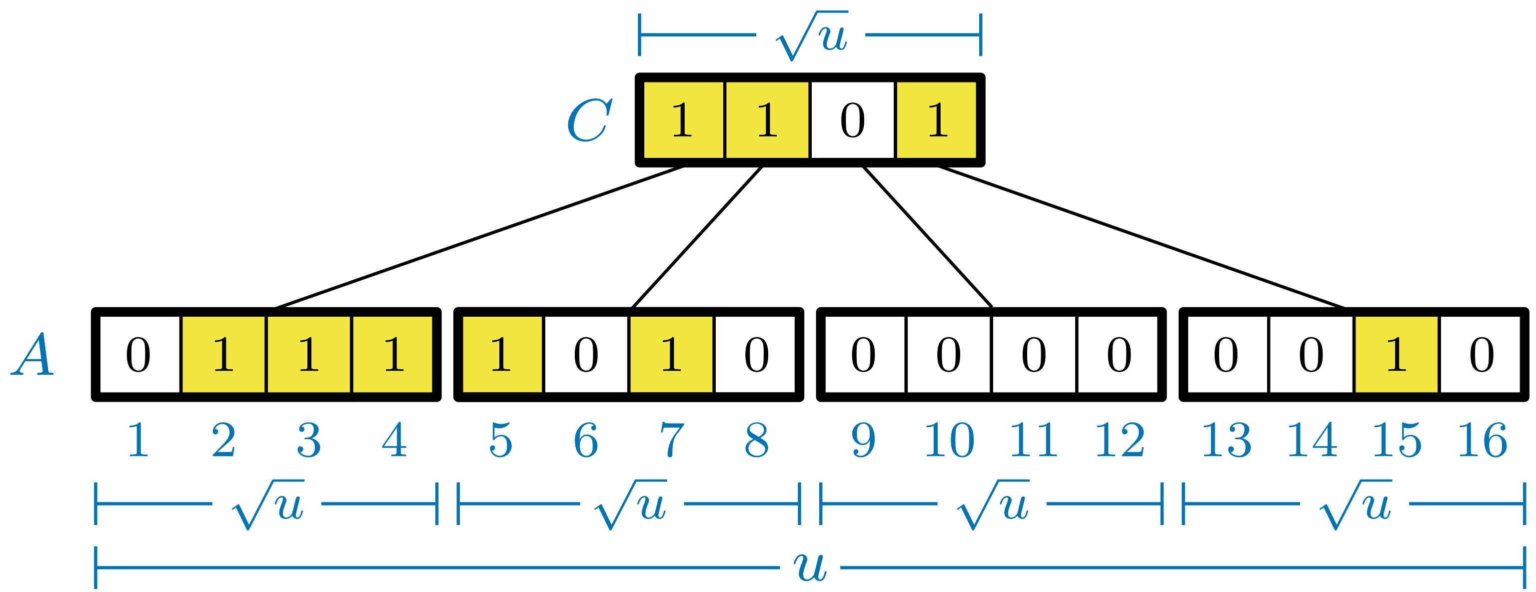


Split A into $\sqrt{u}$ *blocks* each containing $\sqrt{u}$ bits

Attempt 2: a constant height tree

(on top of a big array)

C is called the *summary* of A



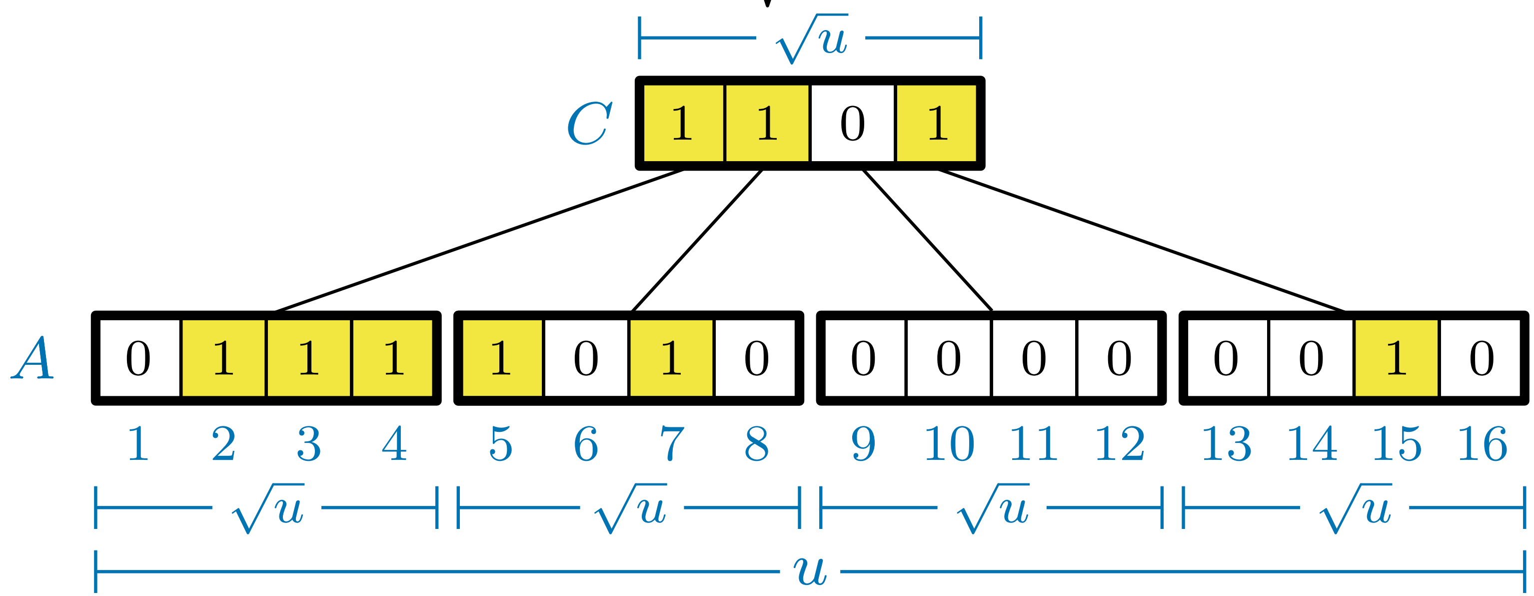
Split A into $\sqrt{u}$ *blocks* each containing $\sqrt{u}$ bits

Attempt 2: 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

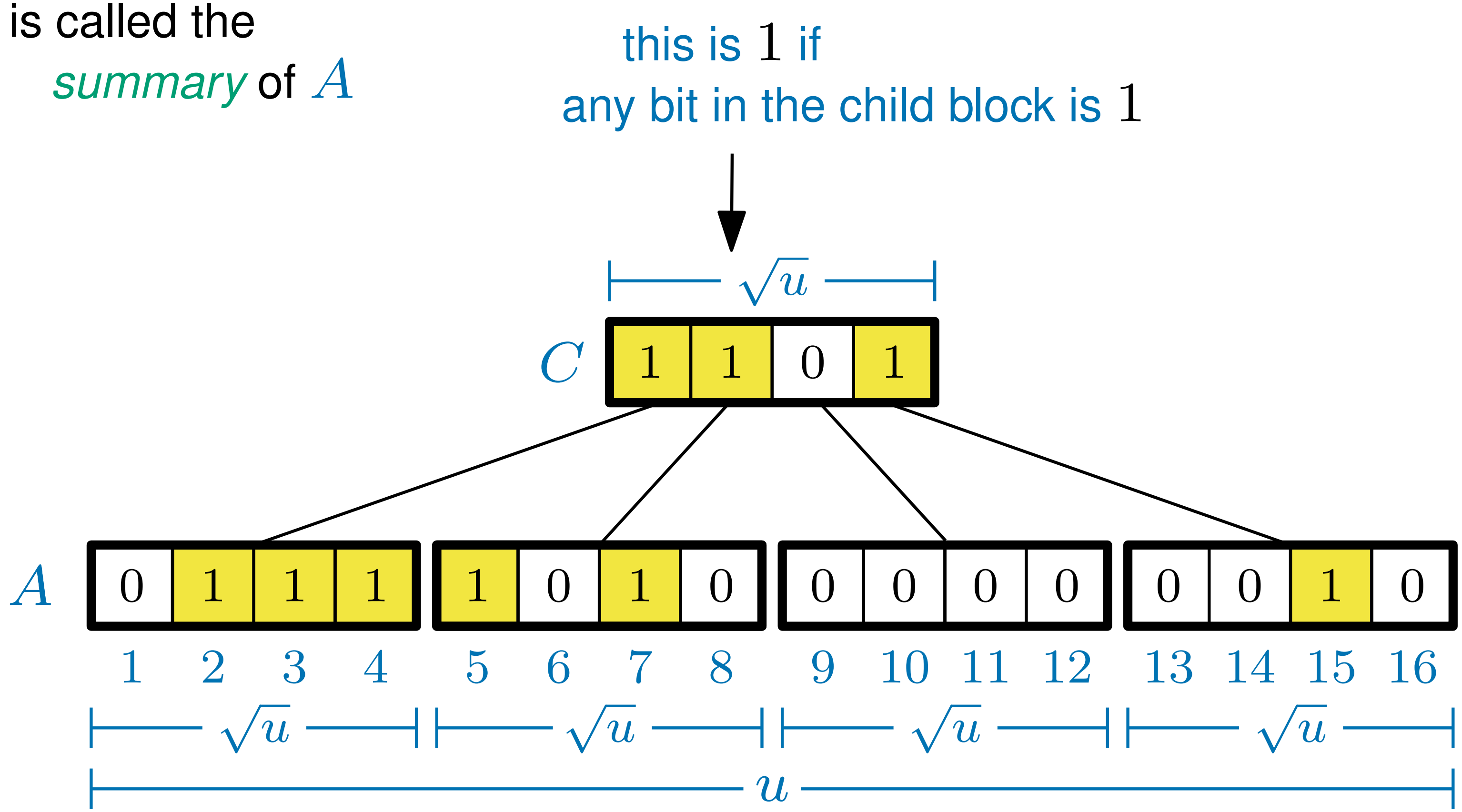


Split A into $\sqrt{u}$ *blocks* each containing $\sqrt{u}$ bits

Attempt 2: 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

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

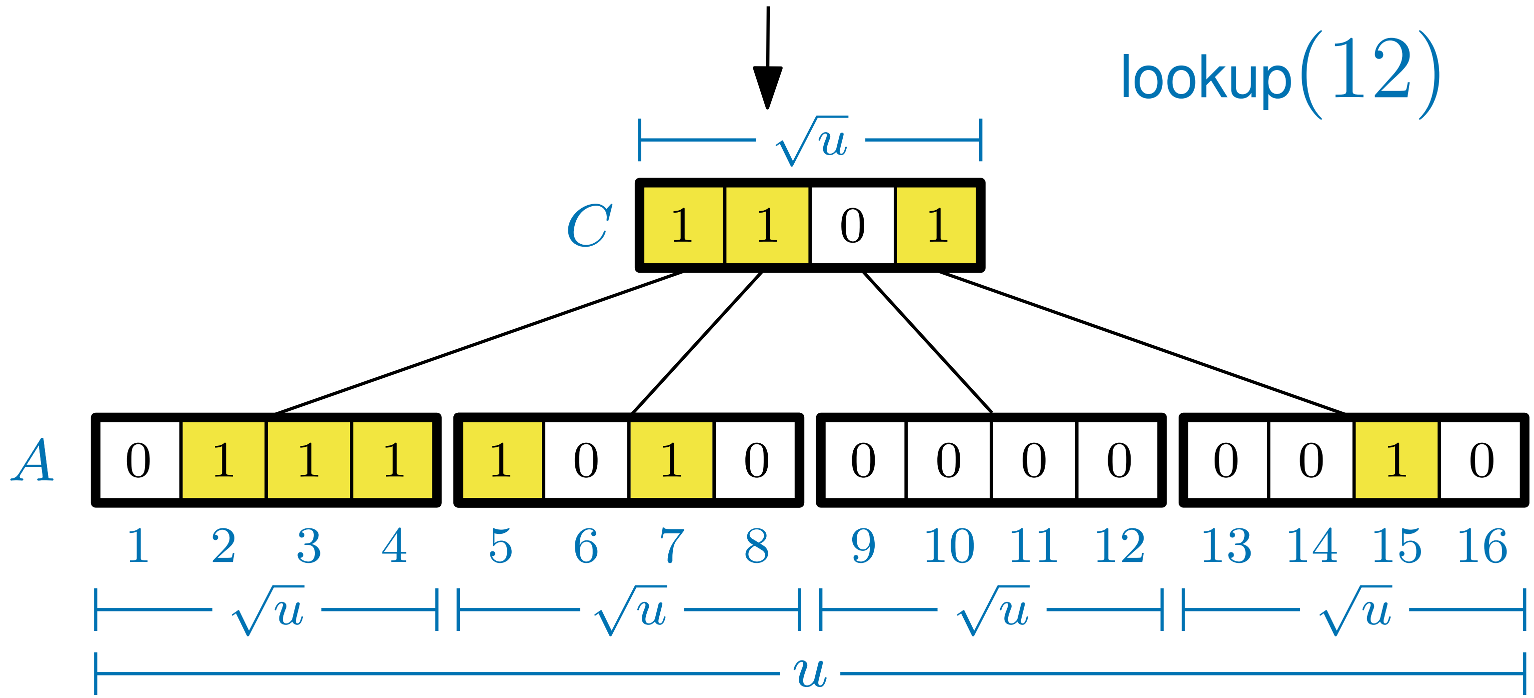
Attempt 2: a constant height tree

(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)



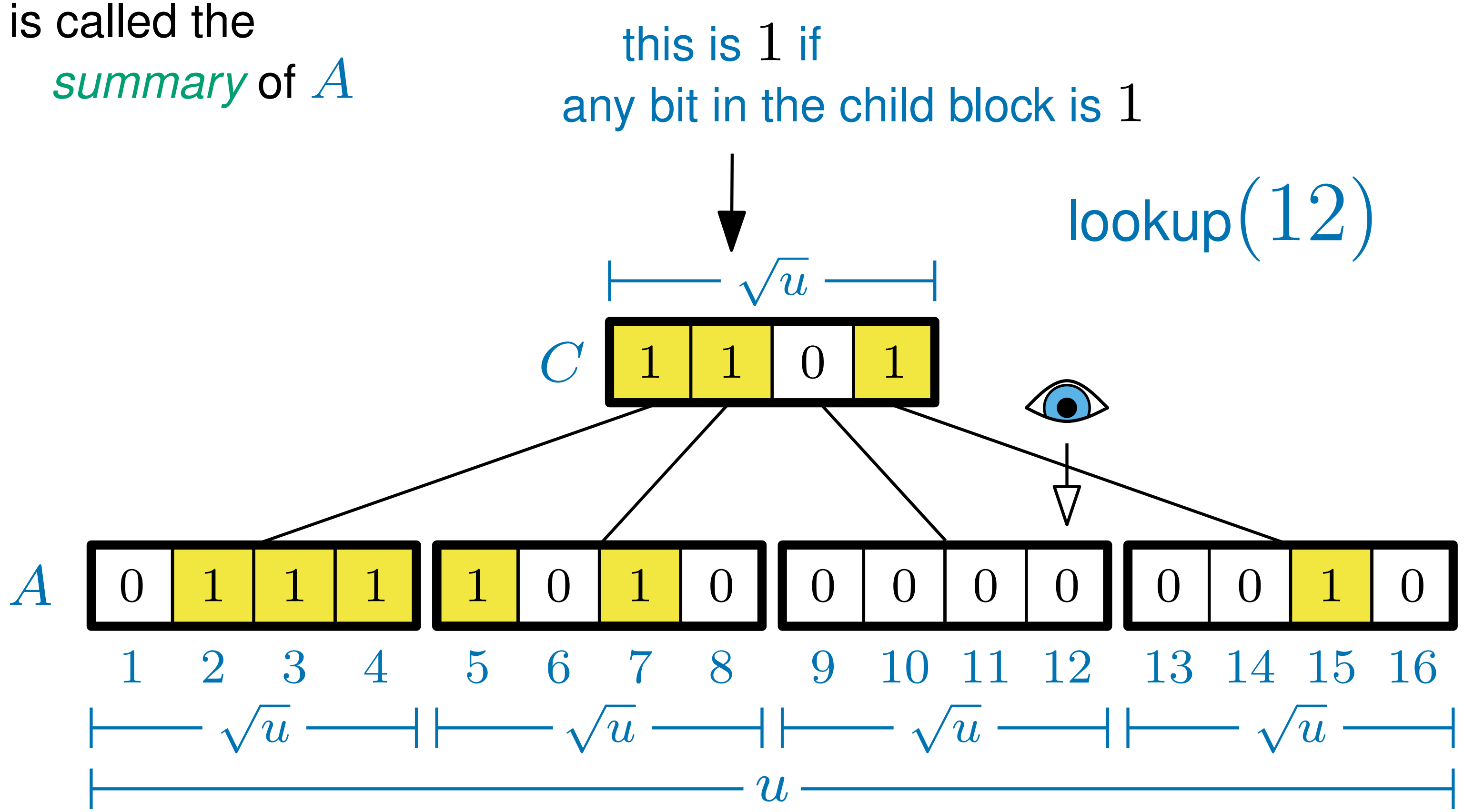
Split A into $\sqrt{u}$ *blocks* each containing $\sqrt{u}$ bits

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

Attempt 2: a constant height tree

(on top of a big array)

C is called the *summary* of A



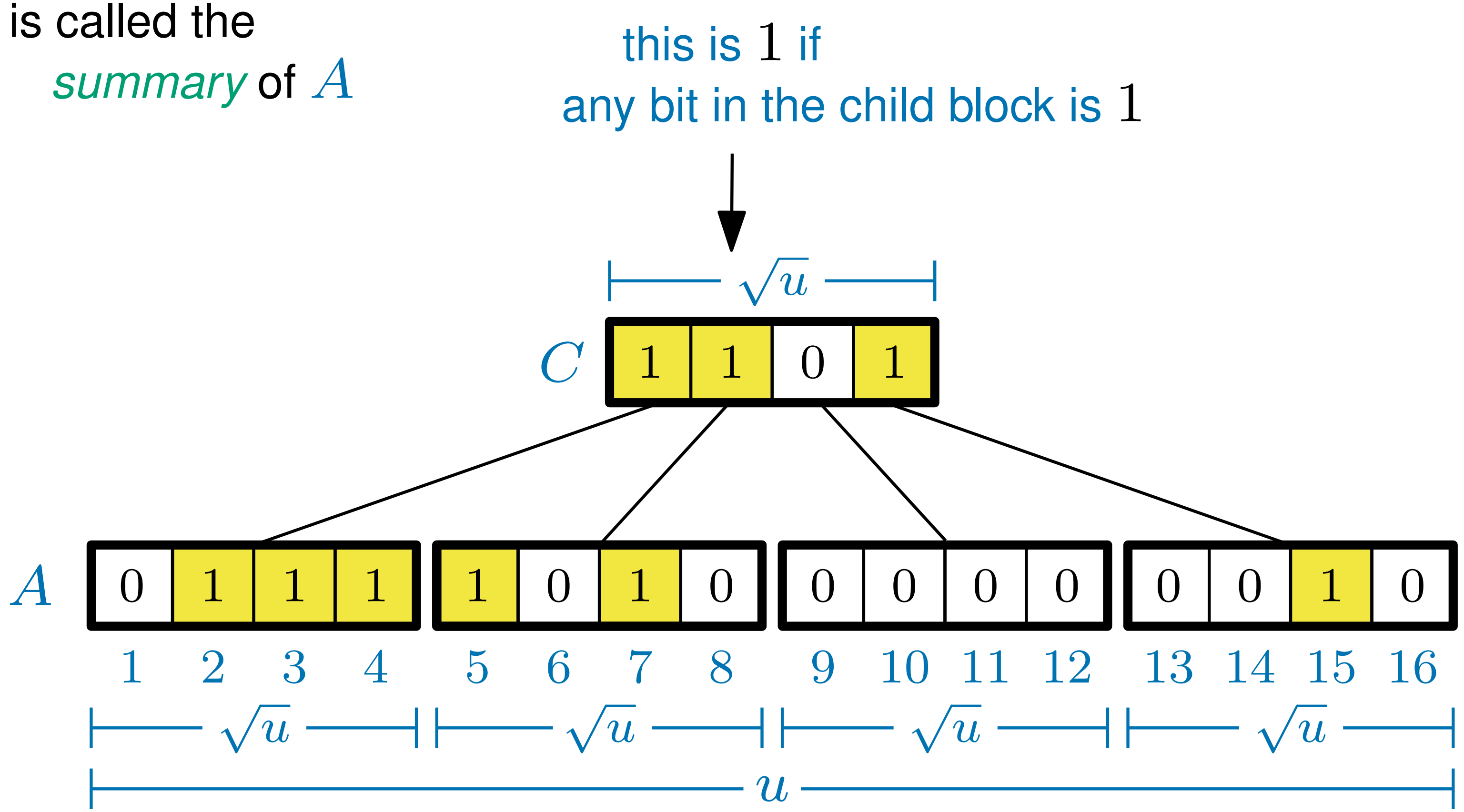
Split A into $\sqrt{u}$ *blocks* each containing $\sqrt{u}$ bits

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

Attempt 2: 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

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

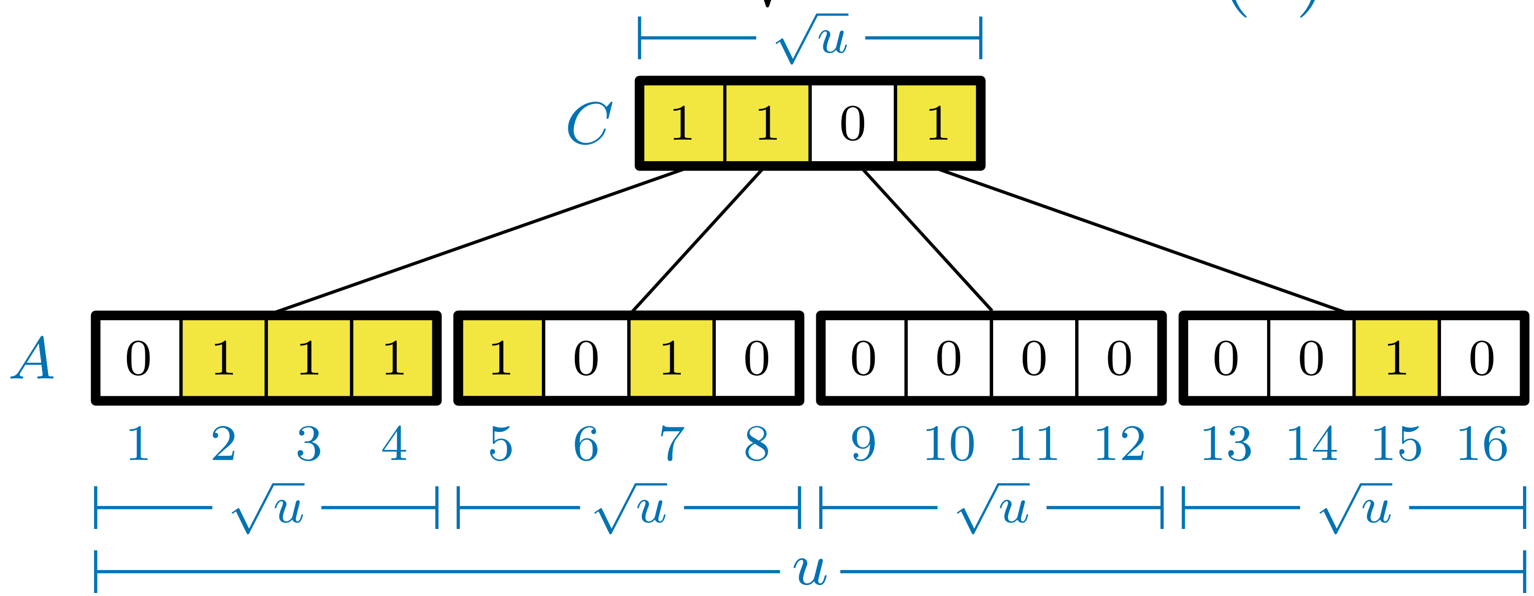
Attempt 2: a constant height tree

(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.

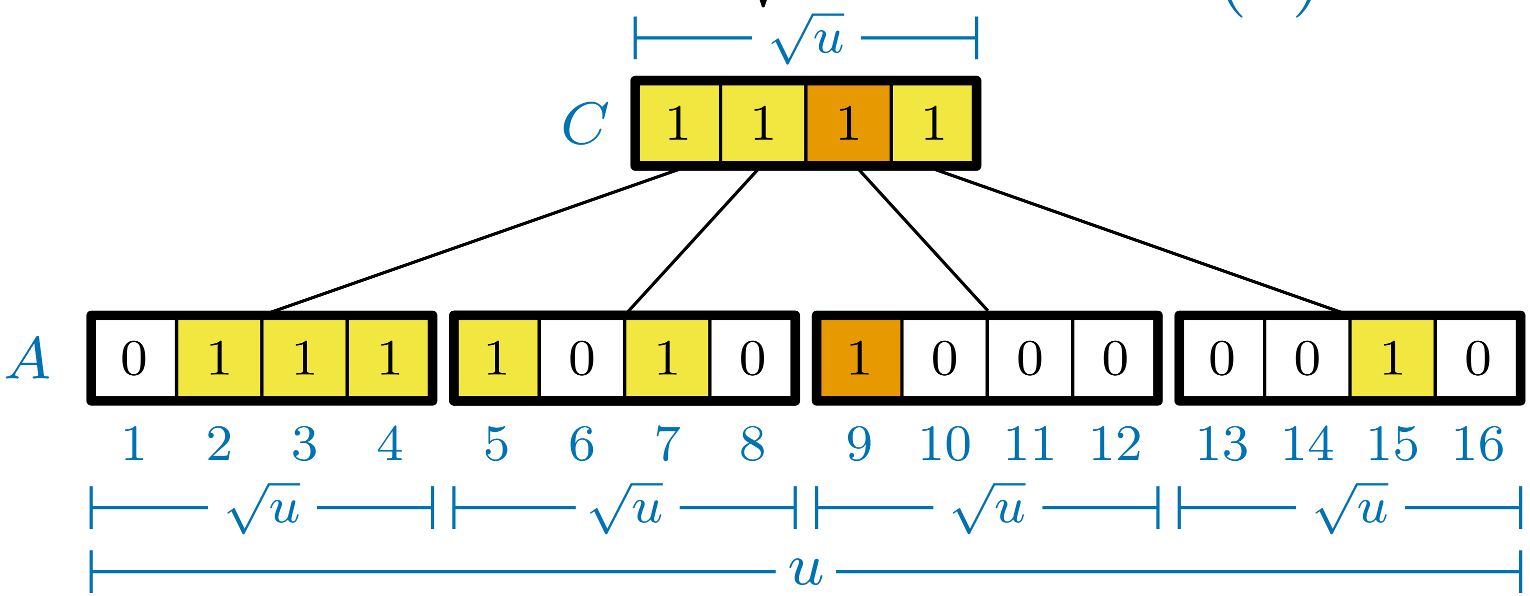
Attempt 2: a constant height tree

(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)



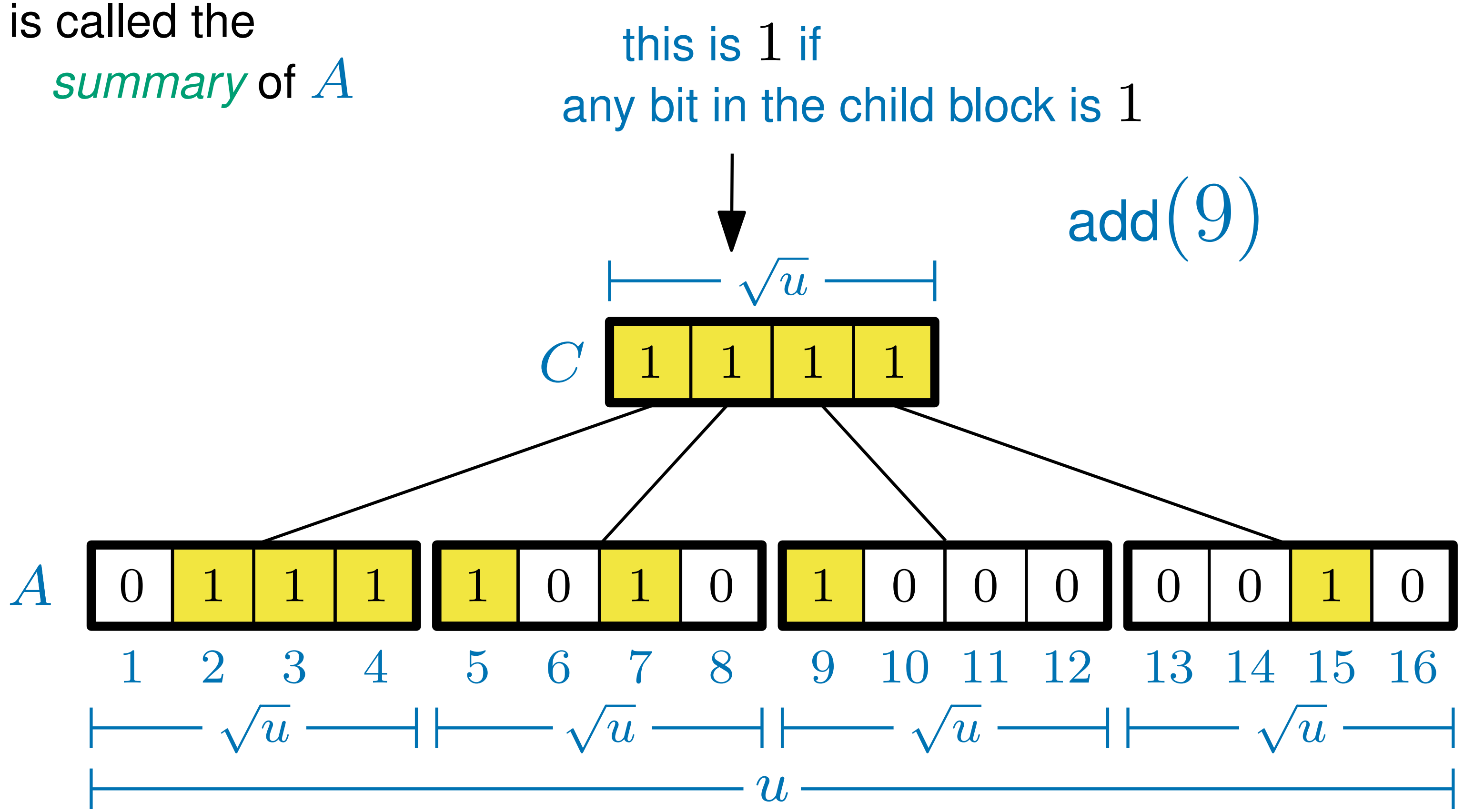
Split A into $\sqrt{u}$ *blocks* each containing $\sqrt{u}$ bits

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

Attempt 2: a constant height tree

(on top of a big array)

C is called the *summary* of A



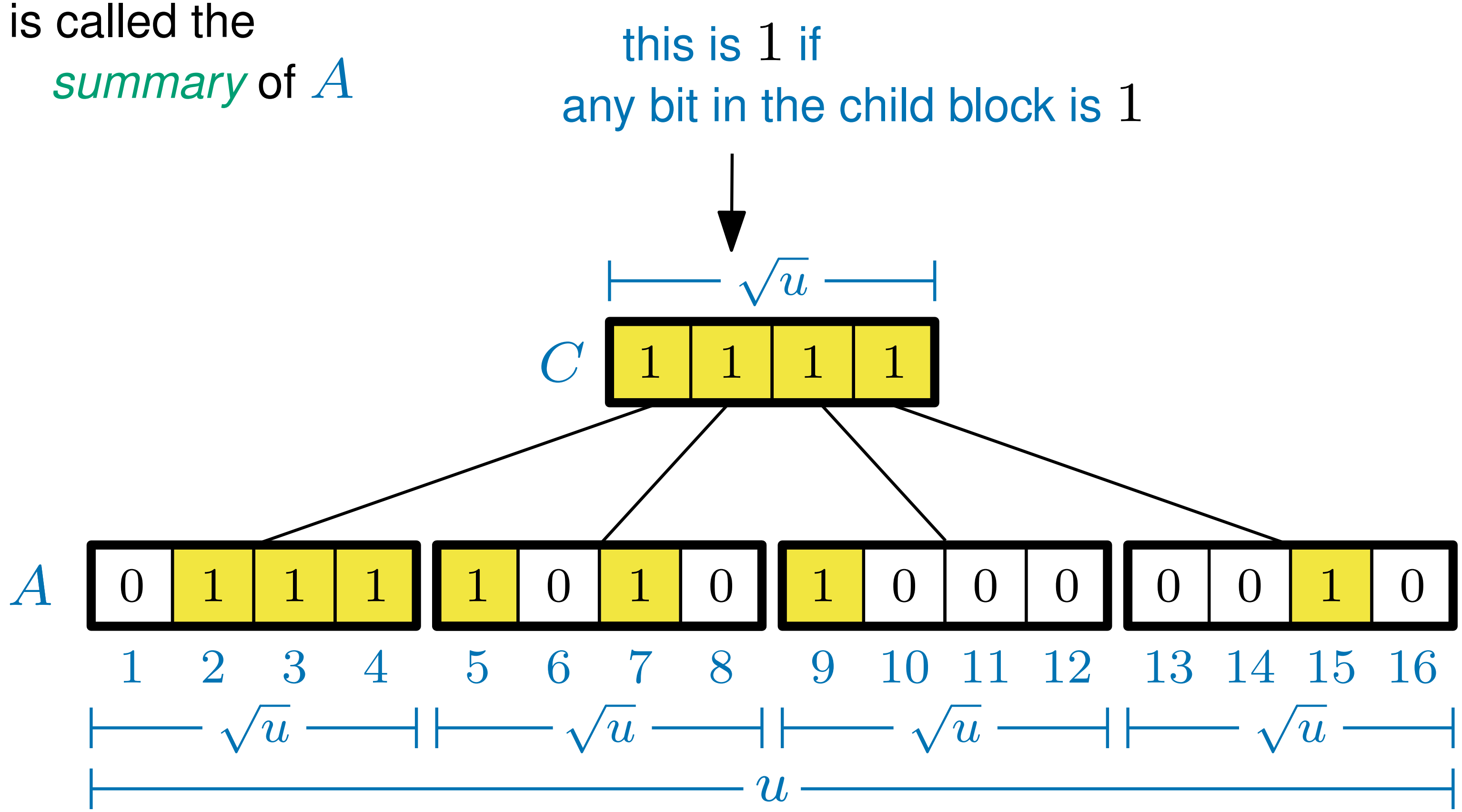
Split A into $\sqrt{u}$ *blocks* each containing $\sqrt{u}$ bits

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

Attempt 2: a constant height tree

(on top of a big array)

C is called the *summary* of A



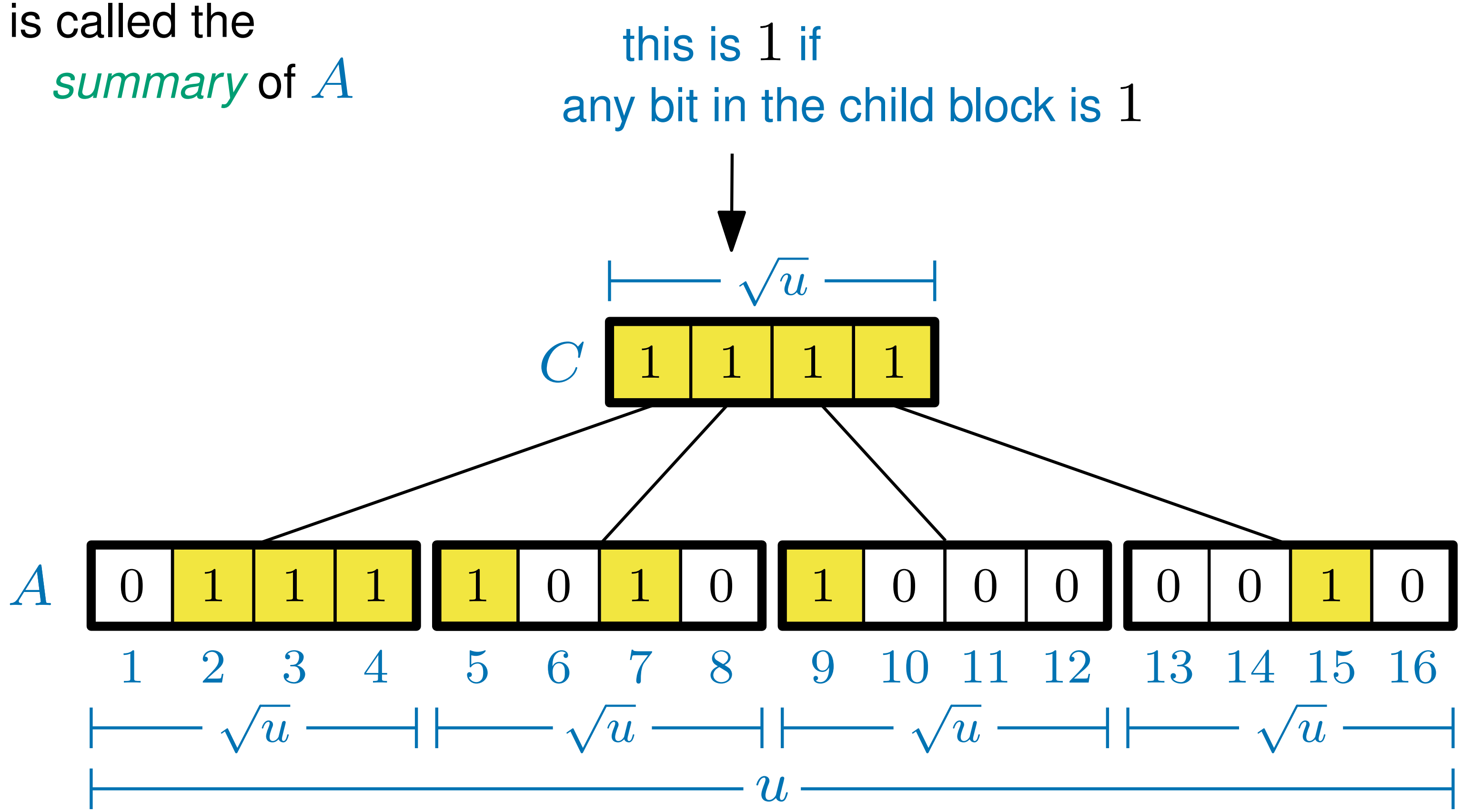
Split A into $\sqrt{u}$ *blocks* each containing $\sqrt{u}$ bits

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

Attempt 2: 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

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

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

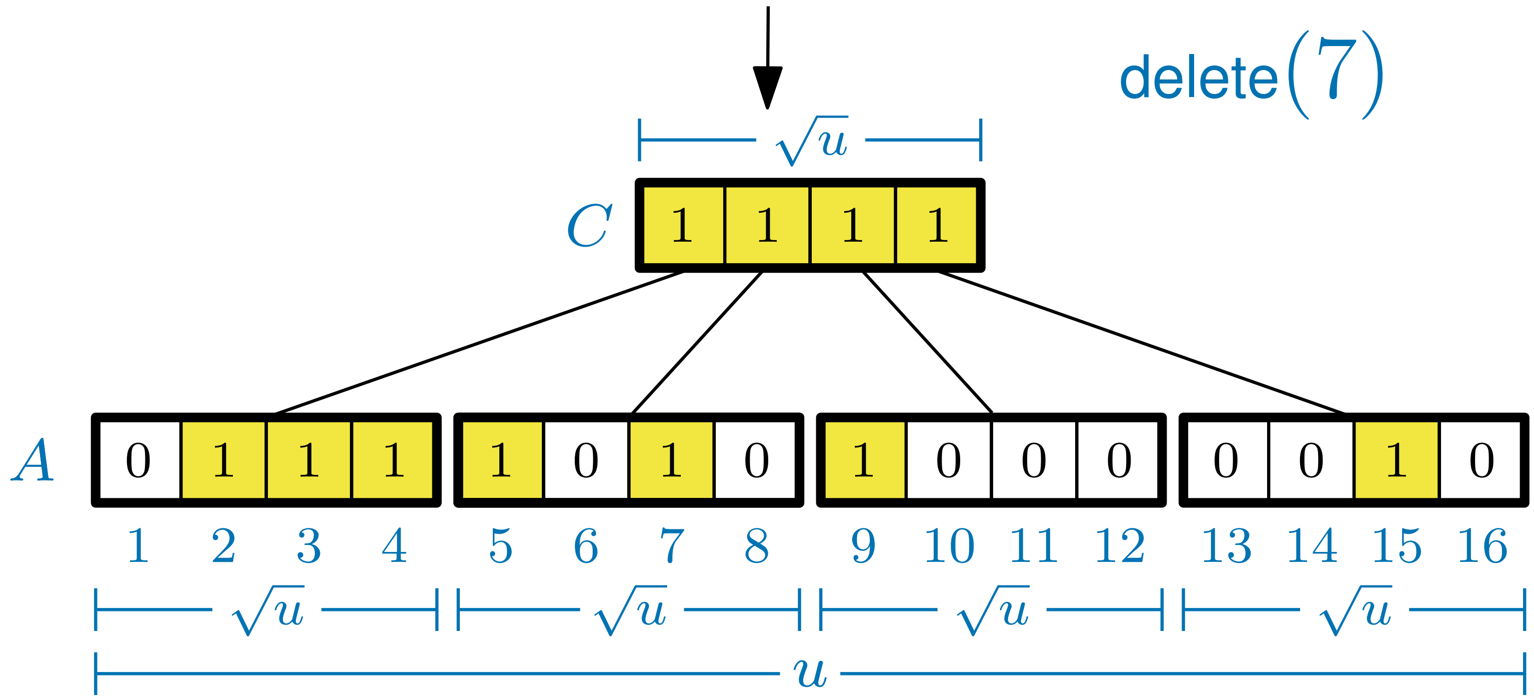
Attempt 2: a constant height tree

(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

delete(7)



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

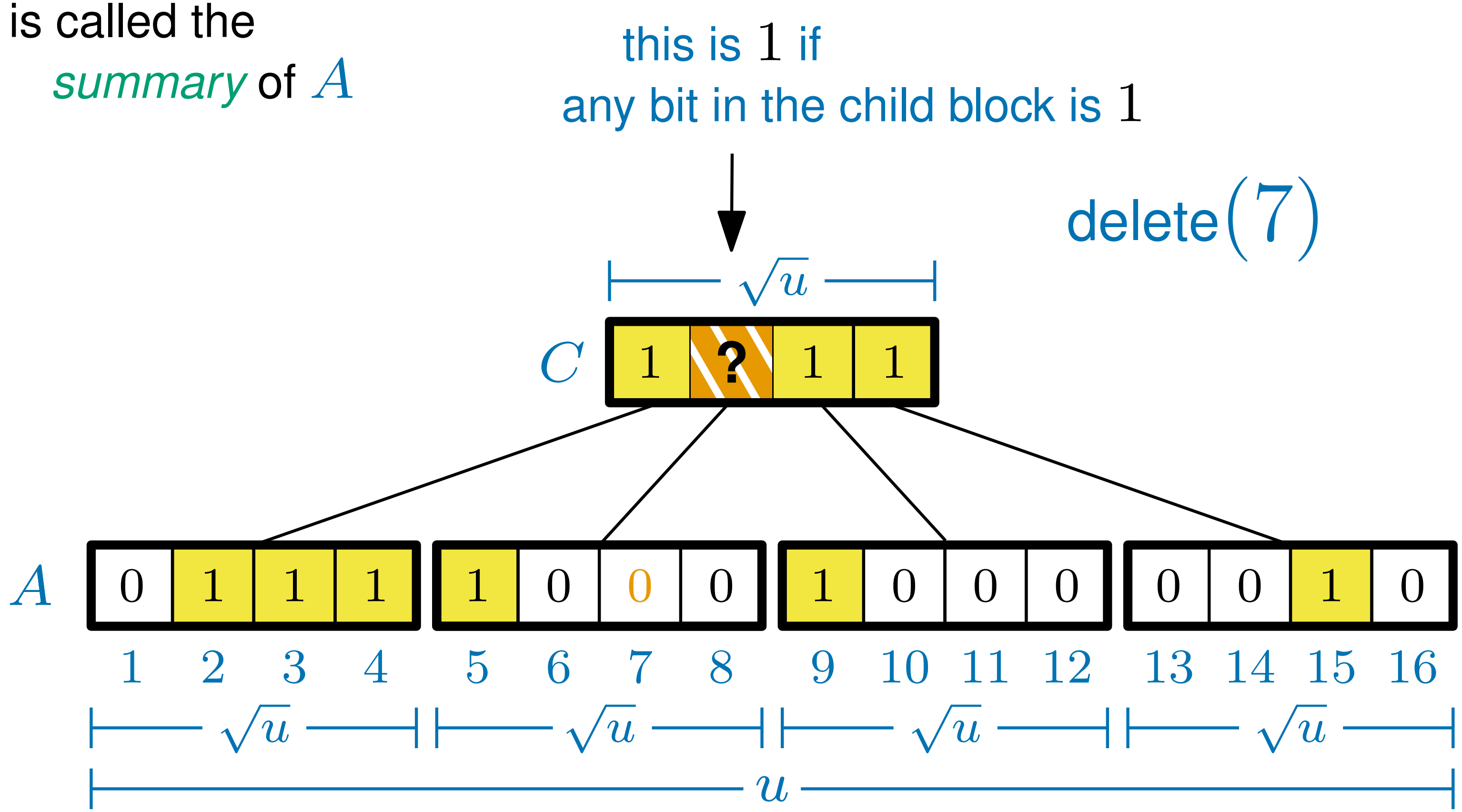
The lookup and add operations take $O(1)$ time.

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

Attempt 2: a constant height tree

(on top of a big array)

C is called the *summary* of A



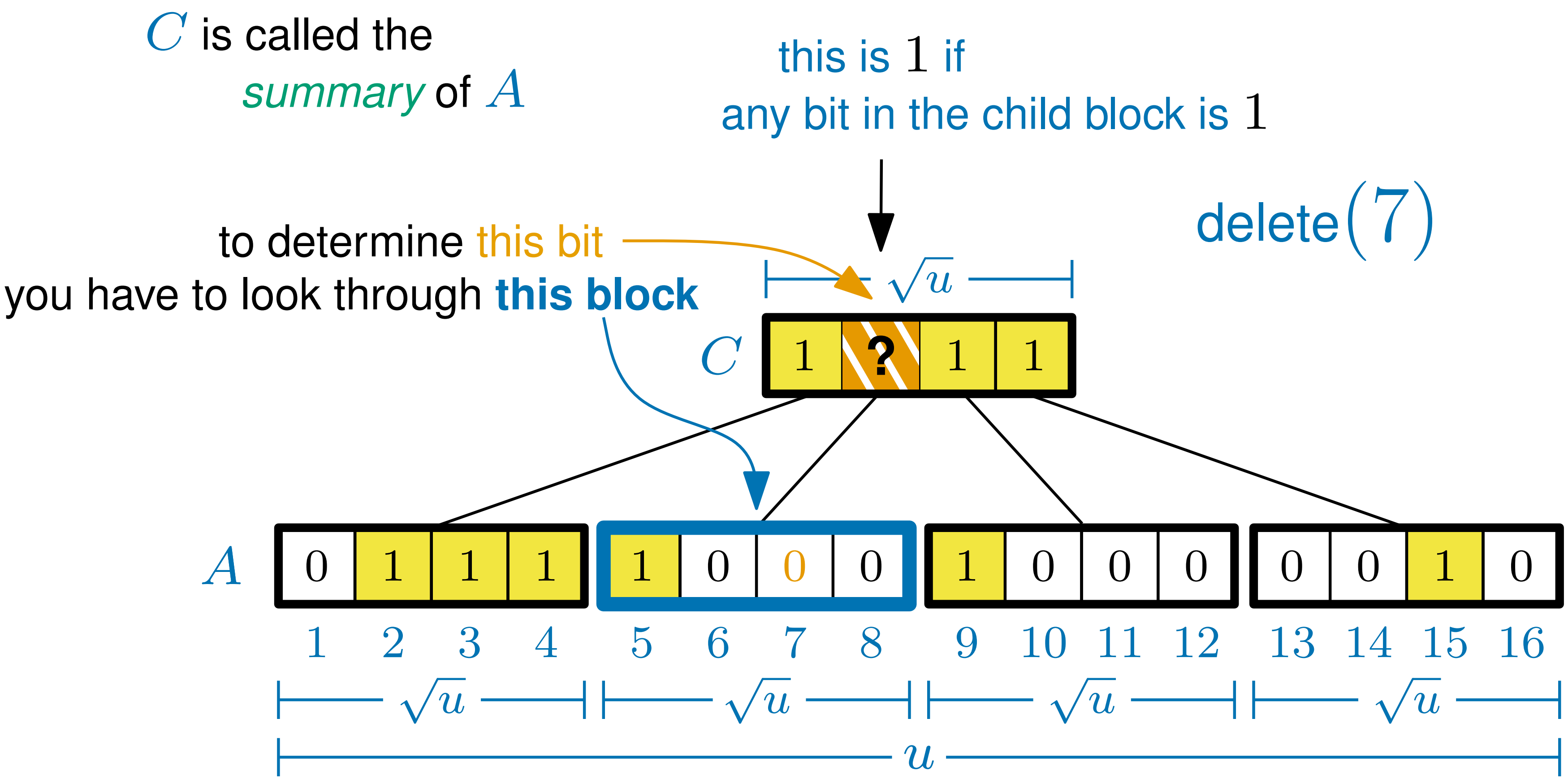
Split A into $\sqrt{u}$ *blocks* each containing $\sqrt{u}$ bits

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

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

Attempt 2: a constant height tree

(on top of a big 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.

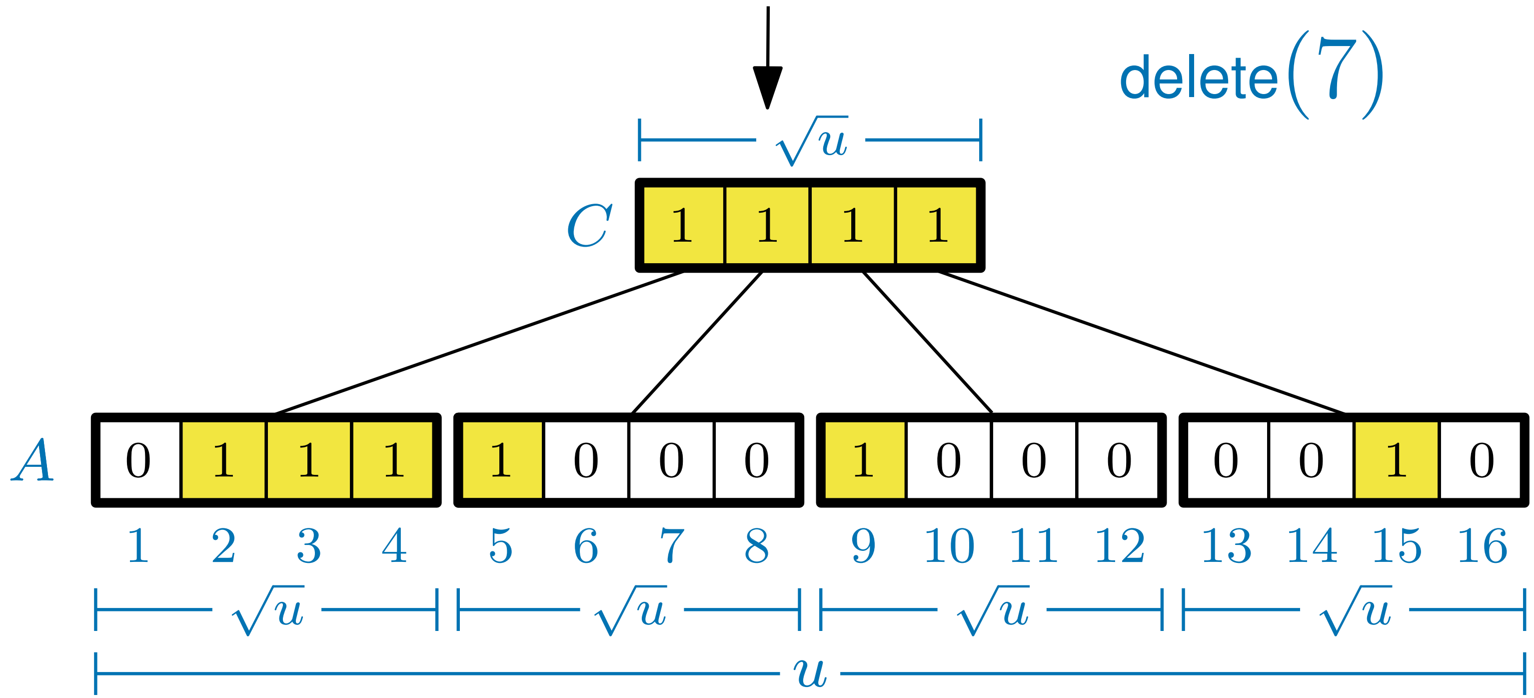
Attempt 2: a constant height tree

(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

delete(7)



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

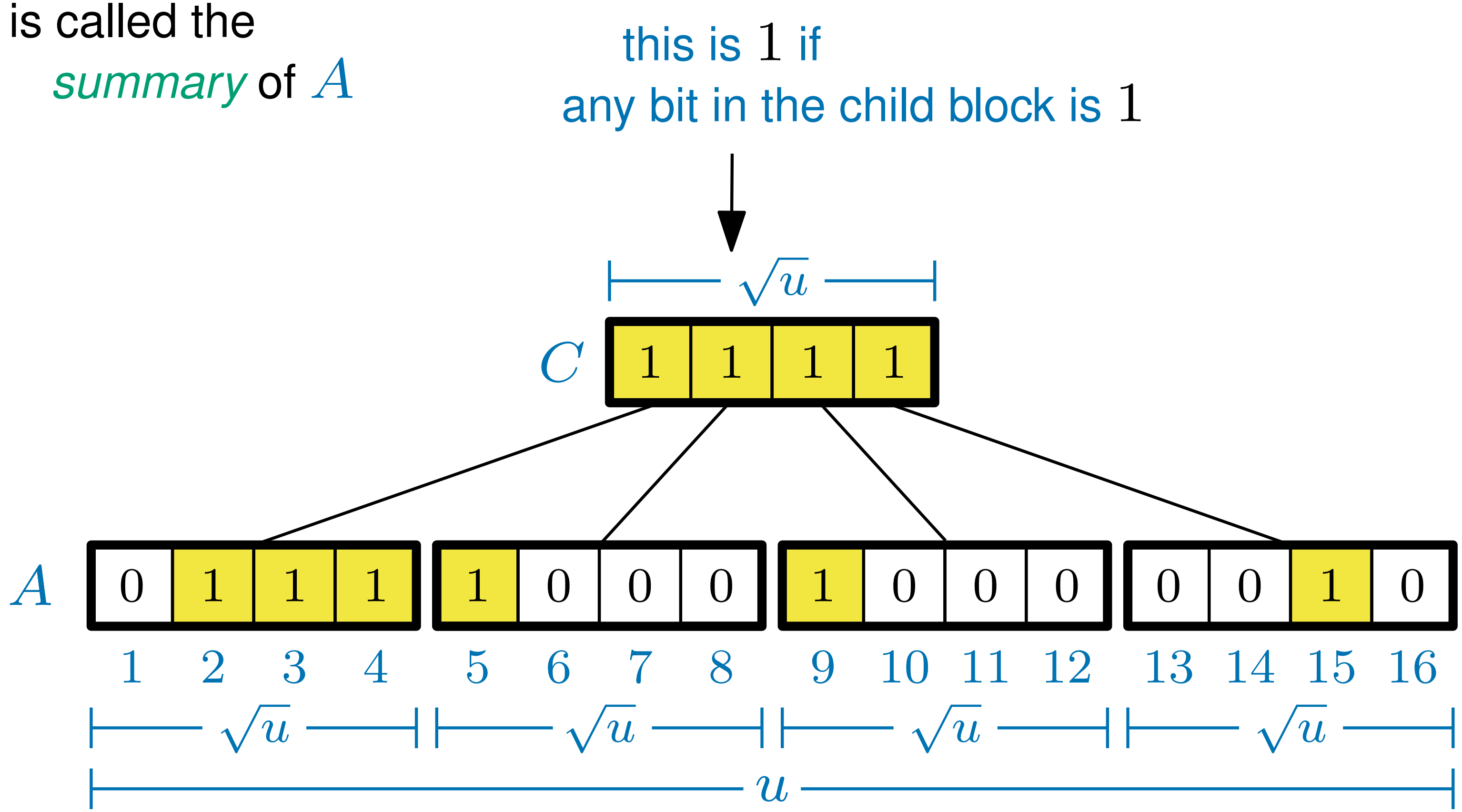
The lookup and add operations take $O(1)$ time.

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

Attempt 2: 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

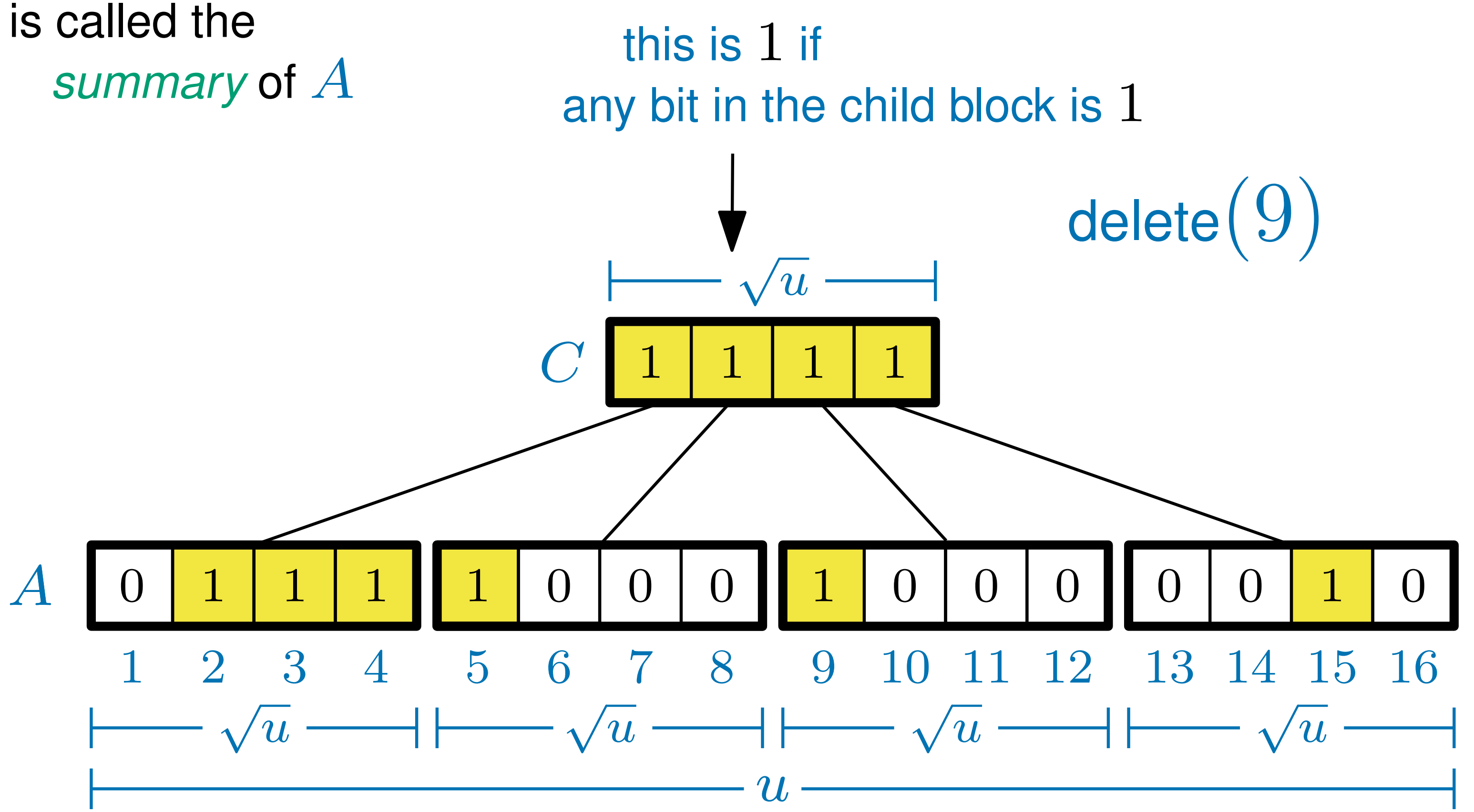
The lookup and add operations take $O(1)$ time.

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

Attempt 2: 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

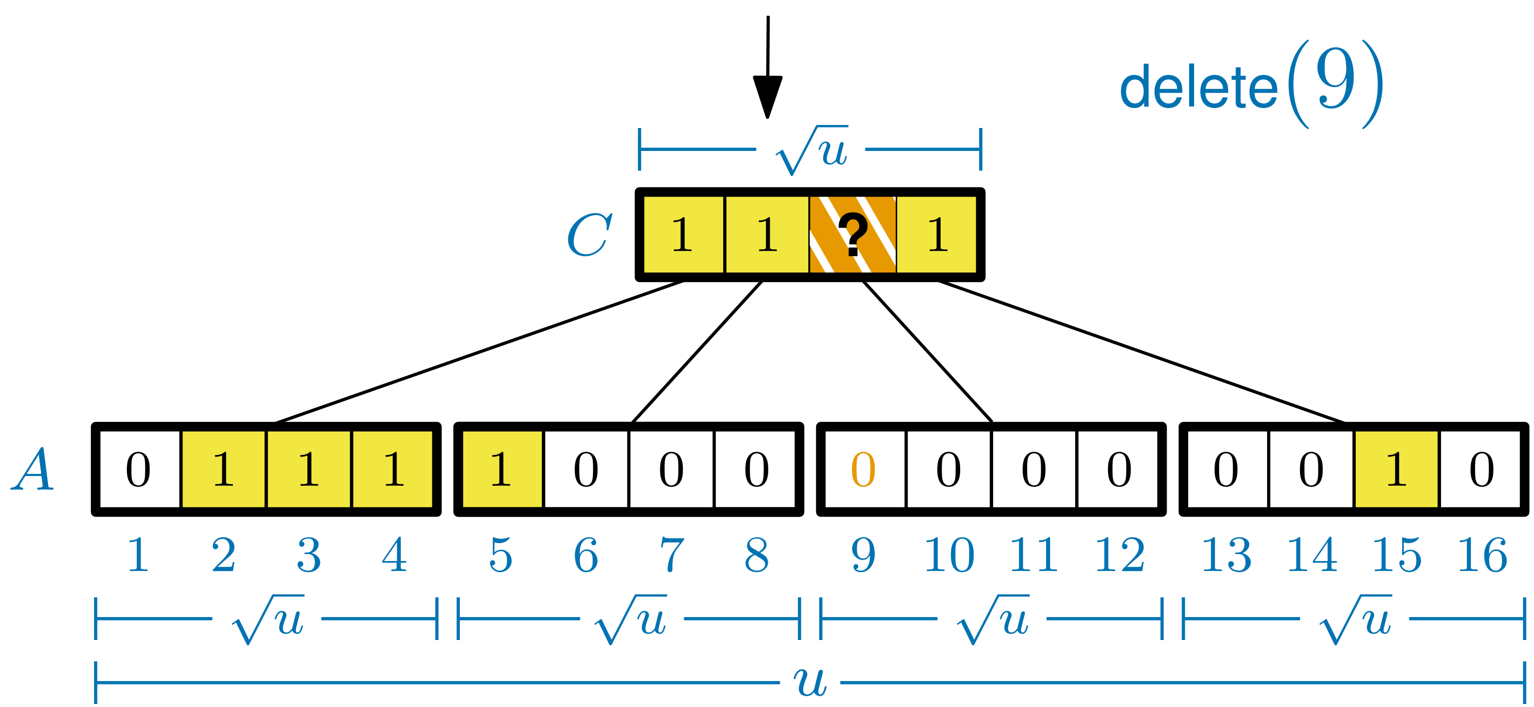
The lookup and add operations take $O(1)$ time.

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

Attempt 2: a constant height tree

(on top of a big array)

C is called the *summary* of A



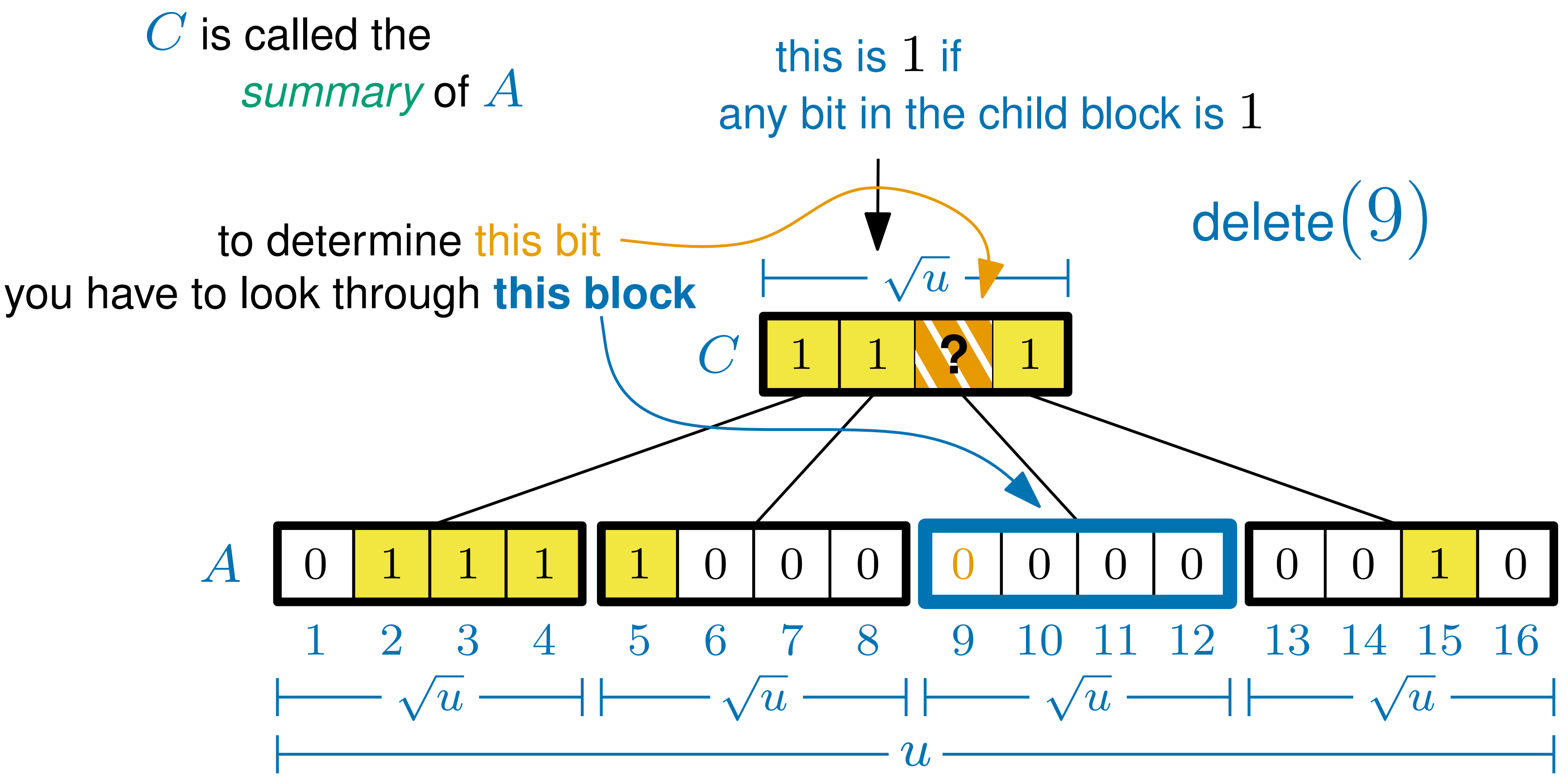
Split A into $\sqrt{u}$ *blocks* each containing $\sqrt{u}$ bits

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

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

Attempt 2: a constant height tree

(on top of a big array)



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

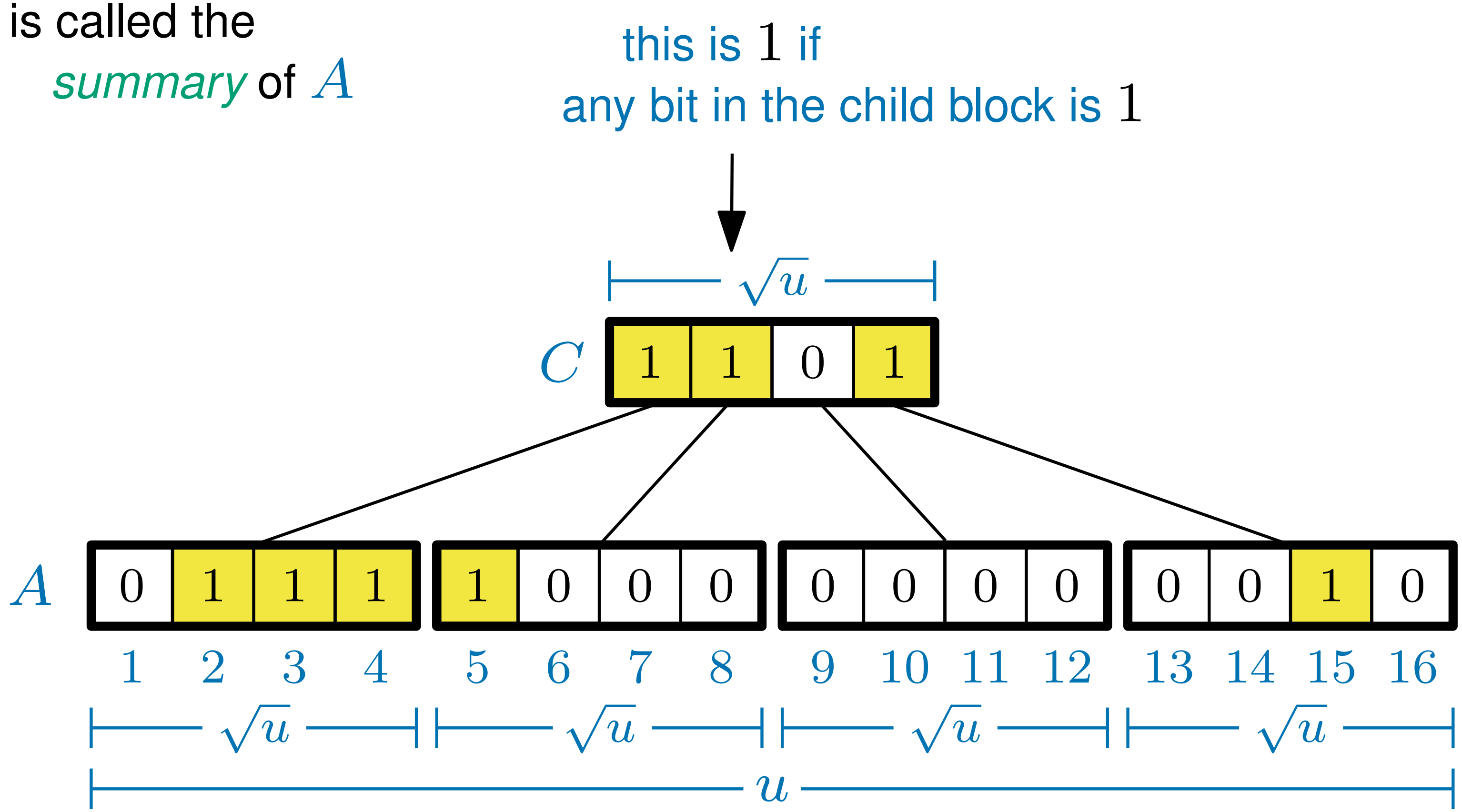
The lookup and add operations take $O(1)$ time.

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

Attempt 2: 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

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

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

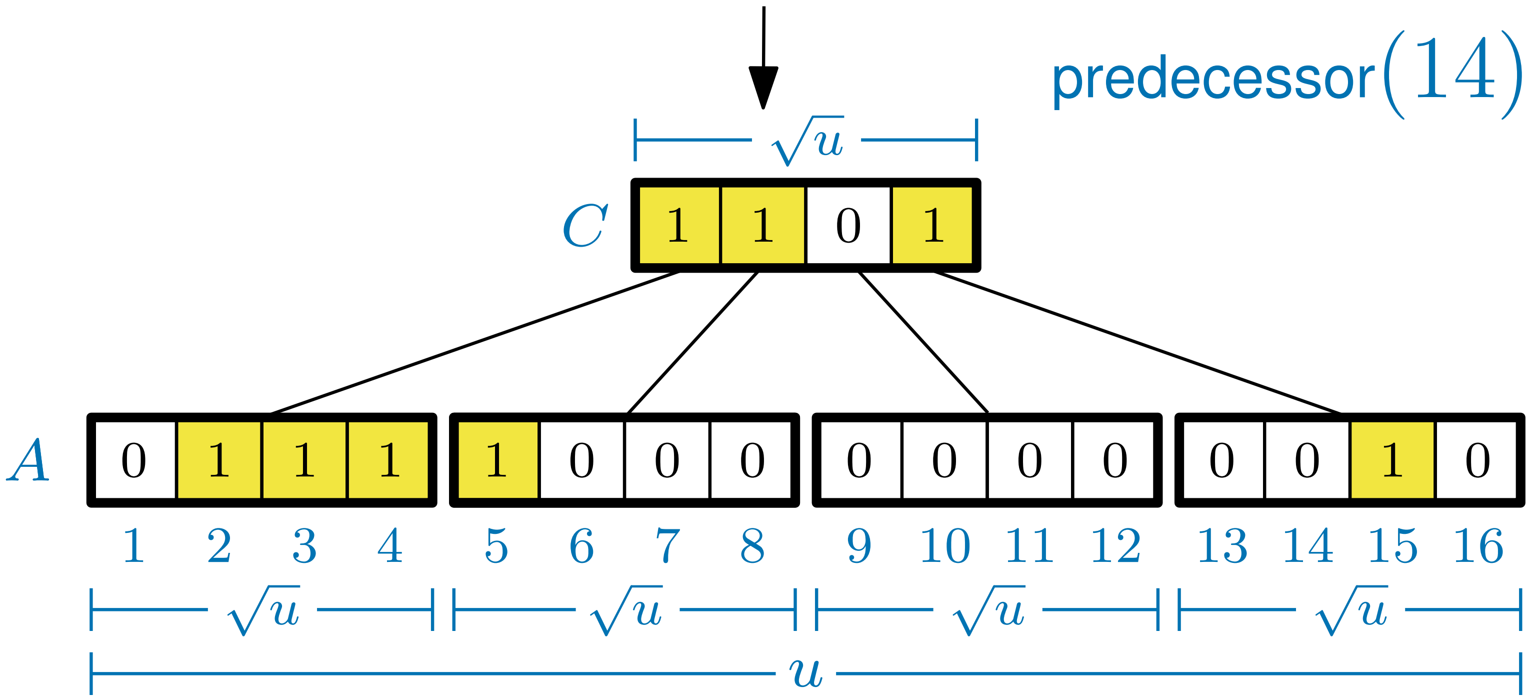
Attempt 2: a constant height tree

(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

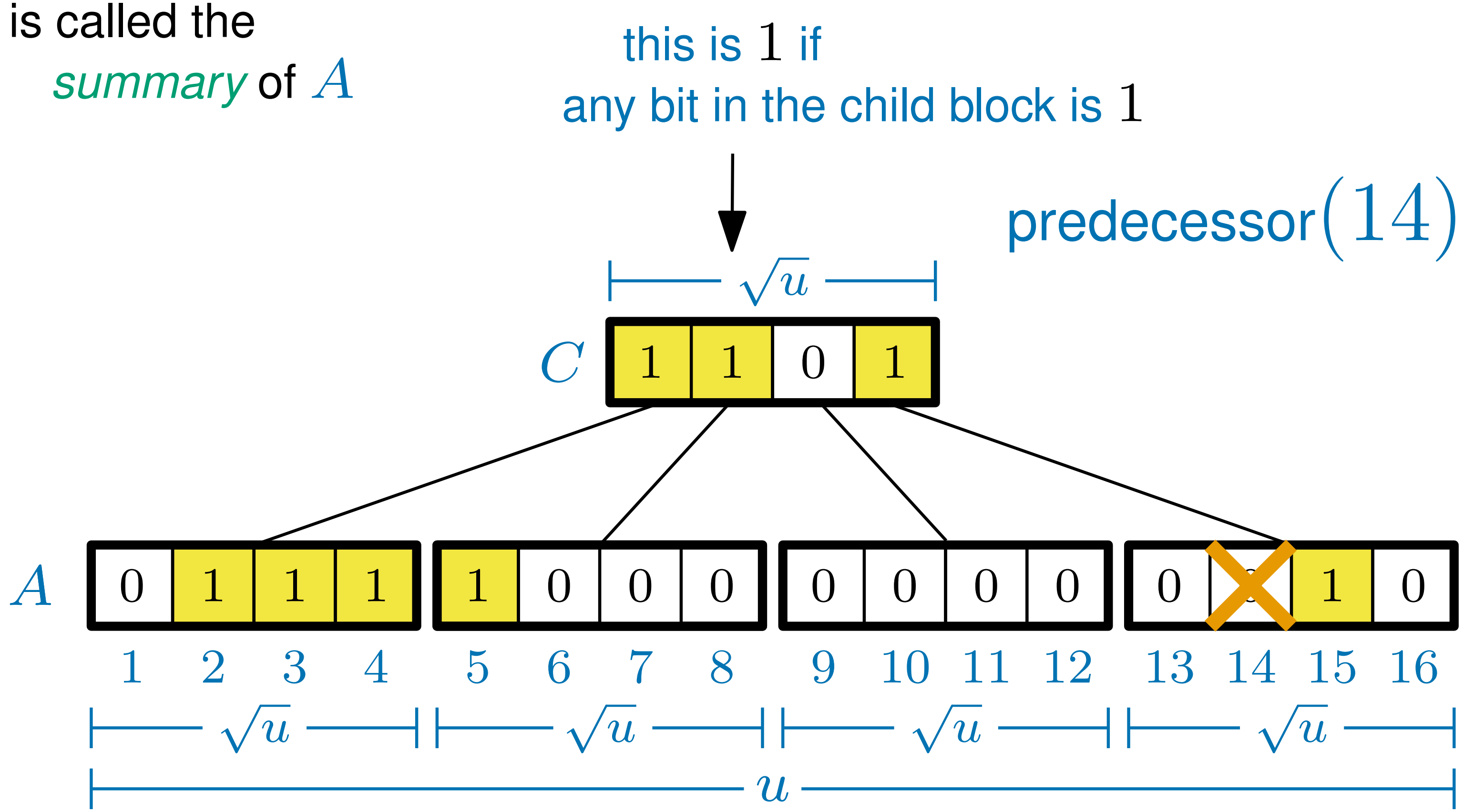
The lookup and add operations take $O(1)$ time.

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

Attempt 2: 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

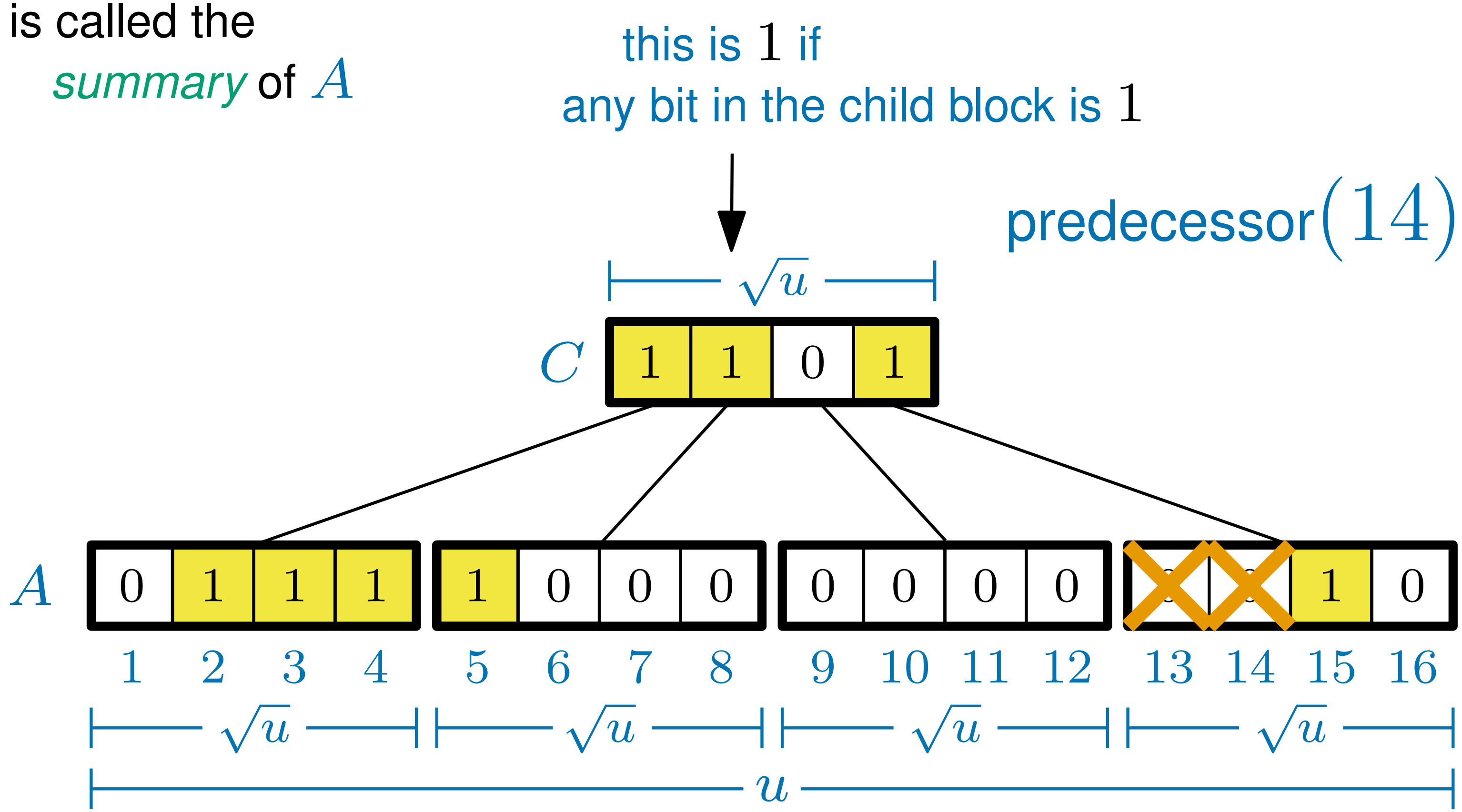
The lookup and add operations take $O(1)$ time.

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

Attempt 2: 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

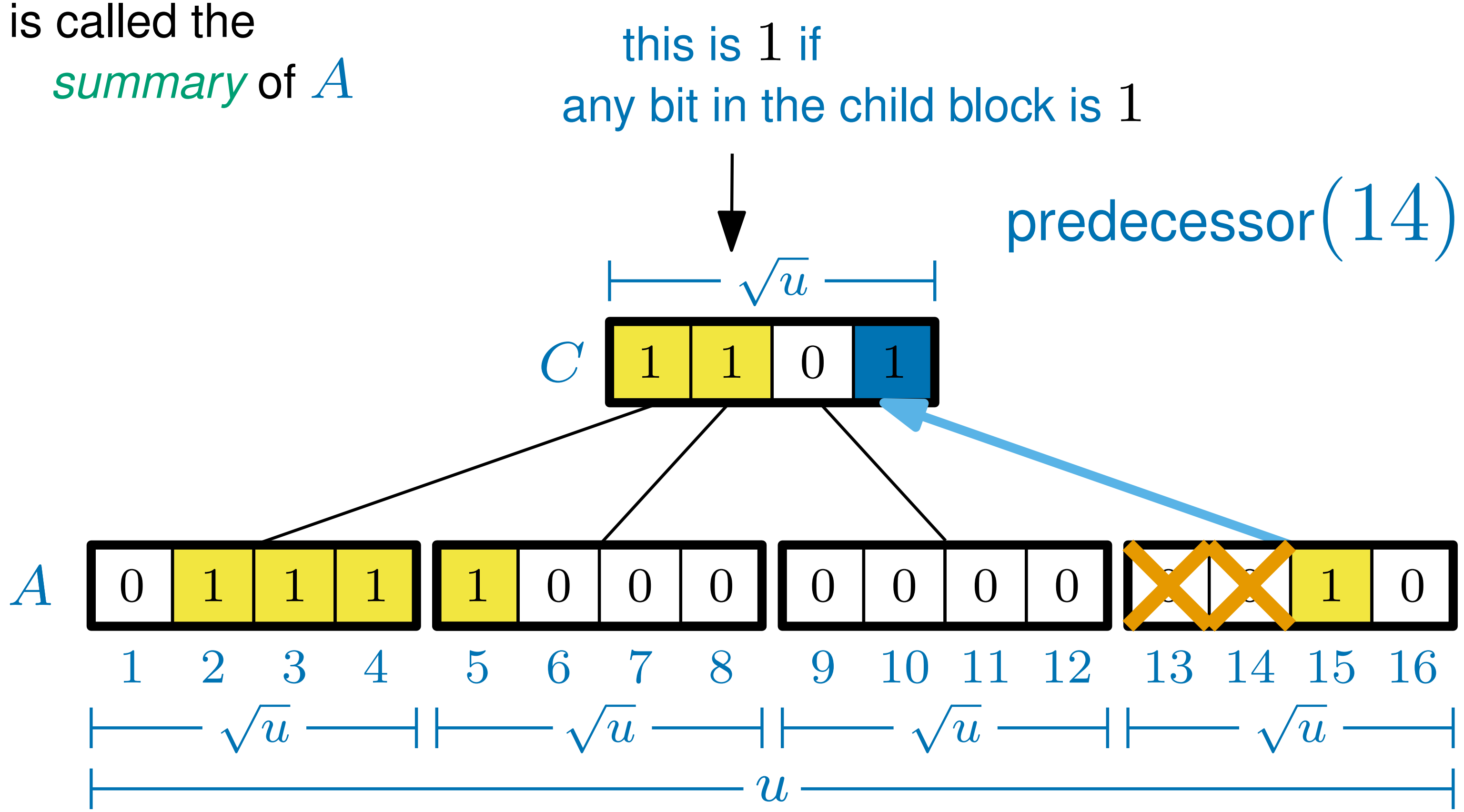
The lookup and add operations take $O(1)$ time.

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

Attempt 2: 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

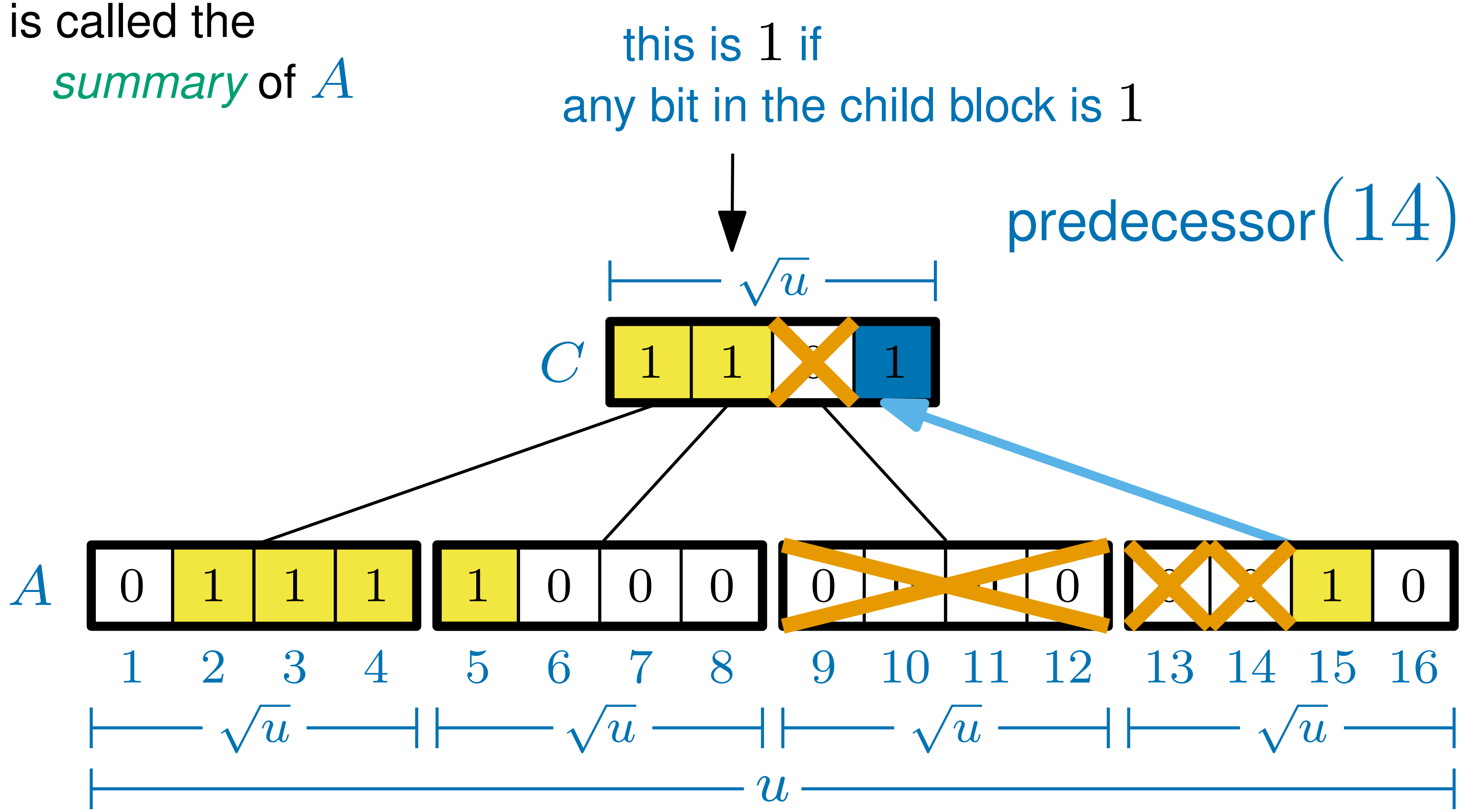
The lookup and add operations take $O(1)$ time.

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

Attempt 2: 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

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

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

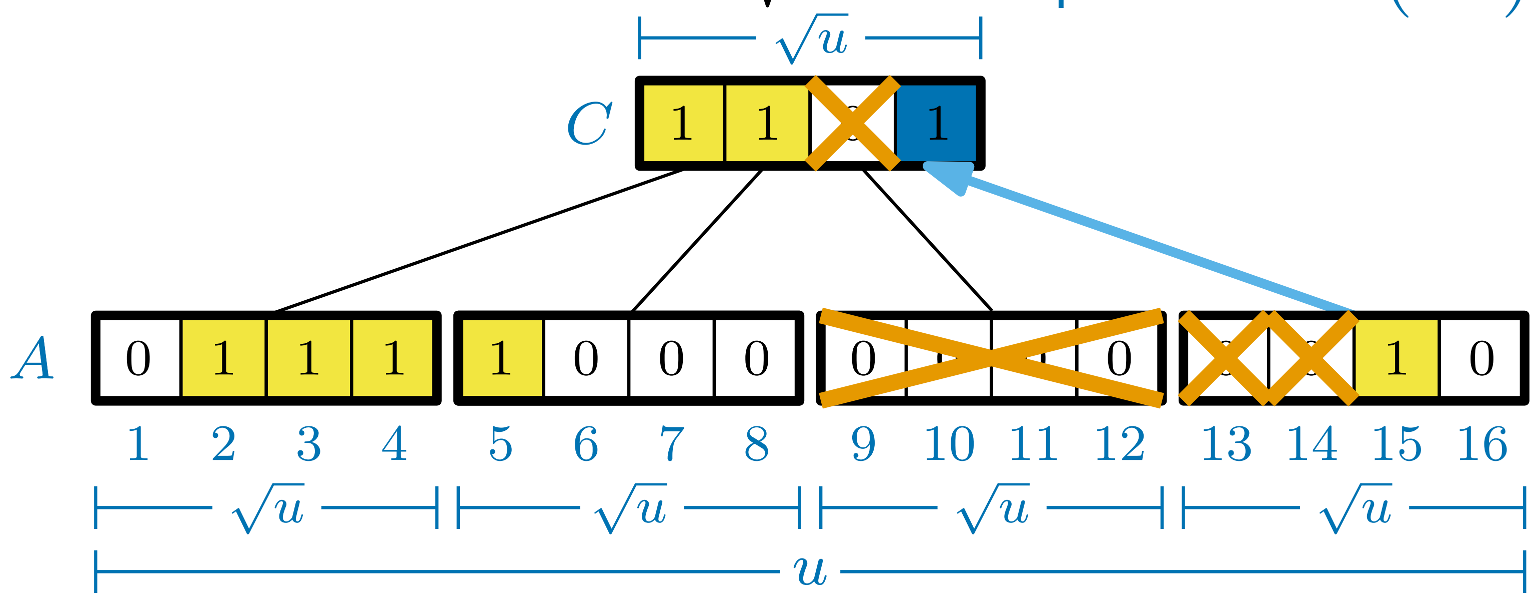
Attempt 2: a constant height tree

(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.

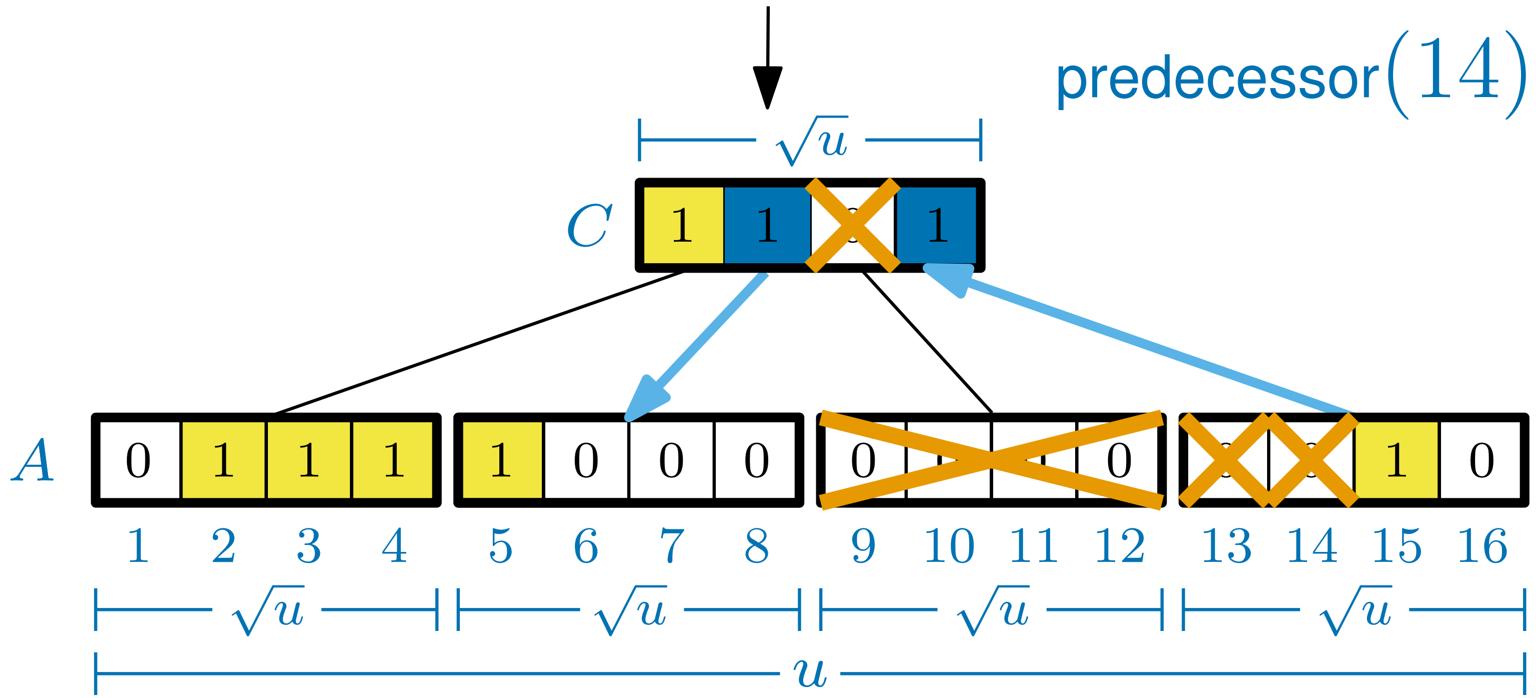
The operations delete, predecessor and successor take $O(\sqrt{u})$ time.

Attempt 2: 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

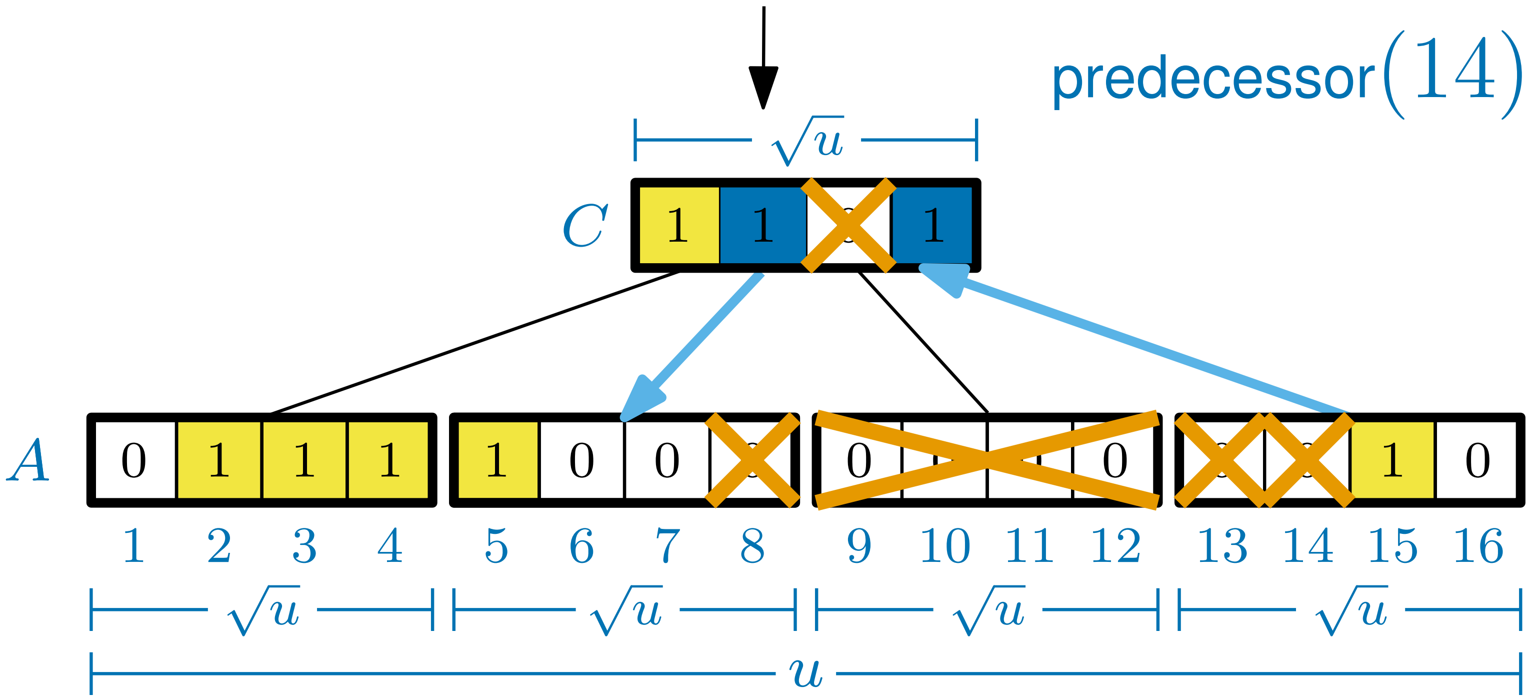
The lookup and add operations take $O(1)$ time.

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

Attempt 2: 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

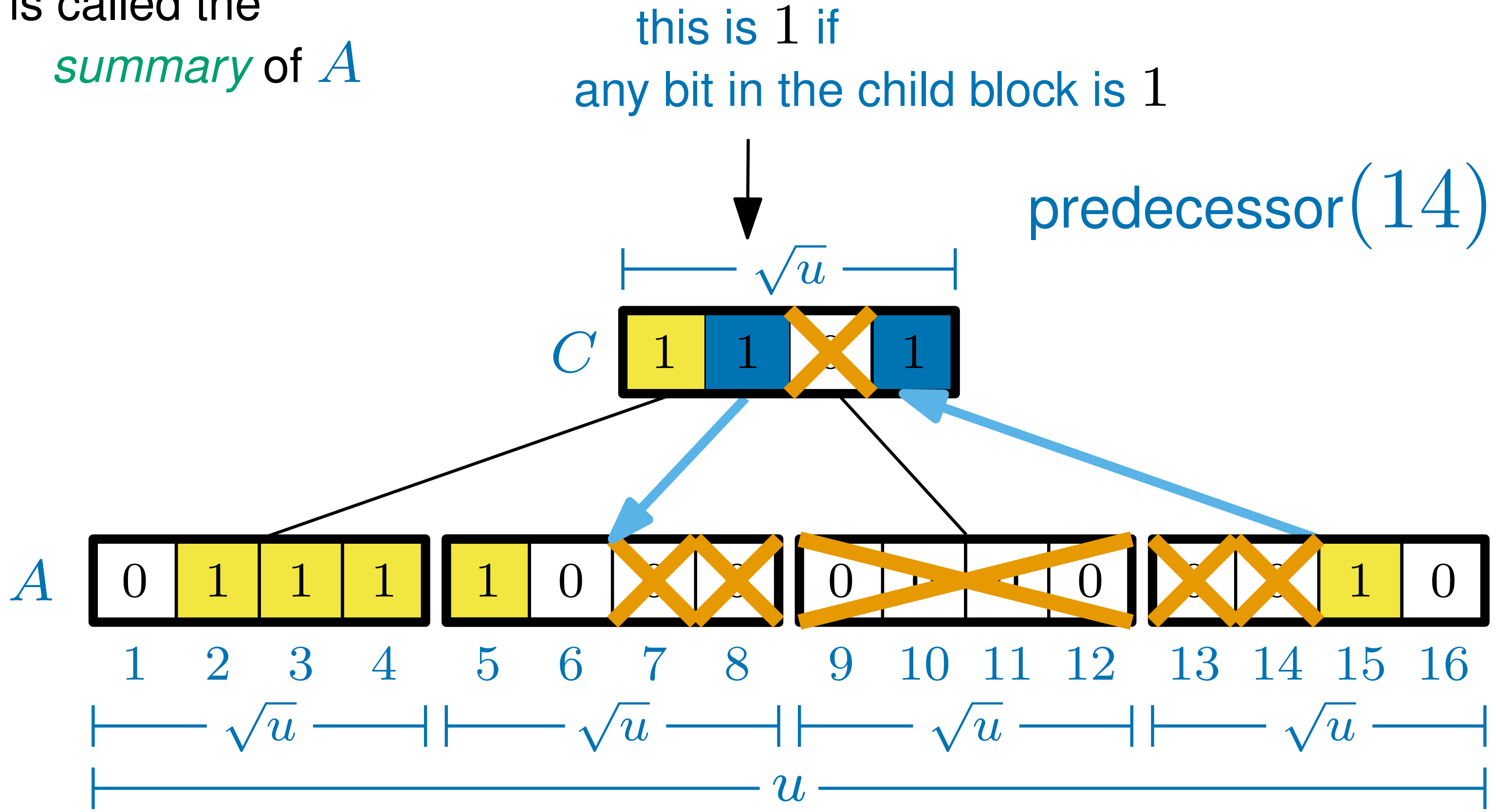
The lookup and add operations take $O(1)$ time.

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

Attempt 2: 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

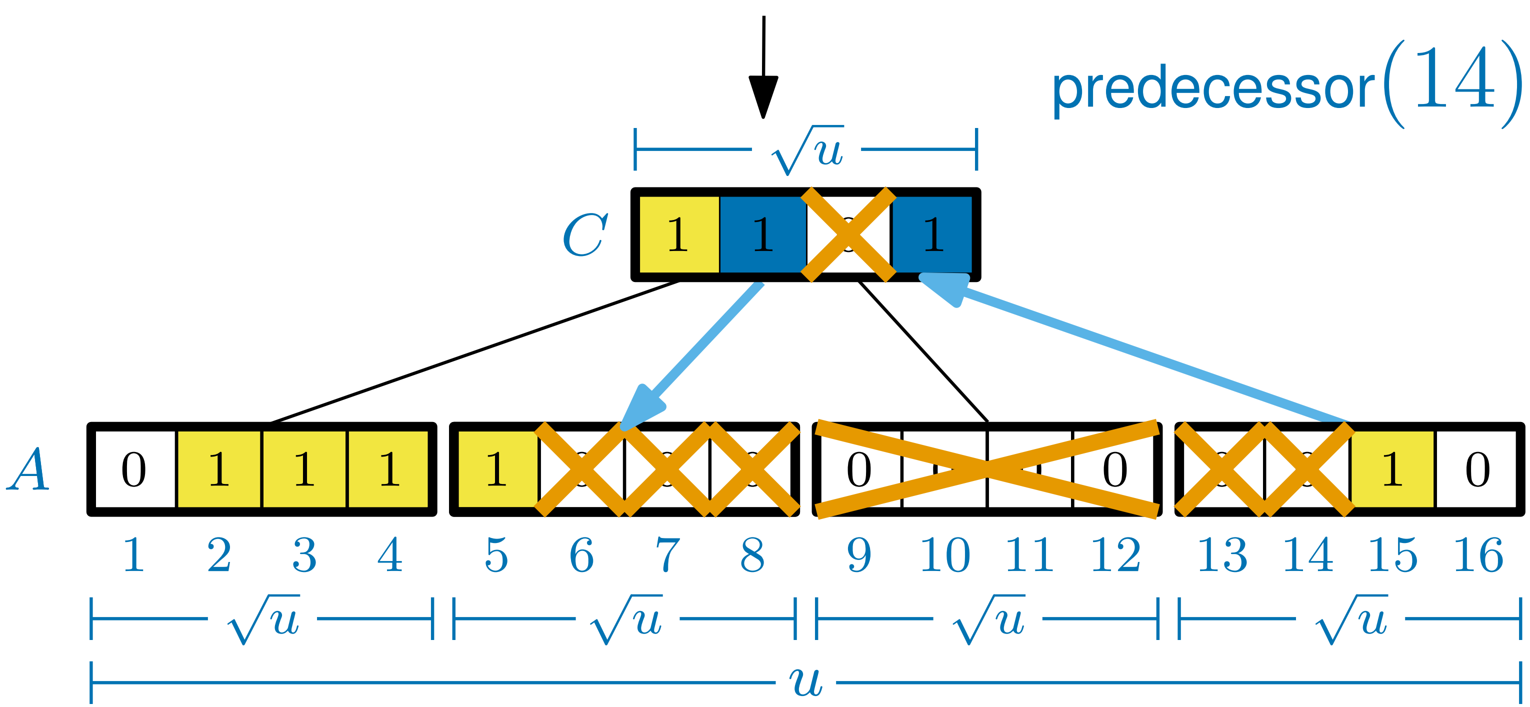
The lookup and add operations take $O(1)$ time.

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

Attempt 2: 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

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

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

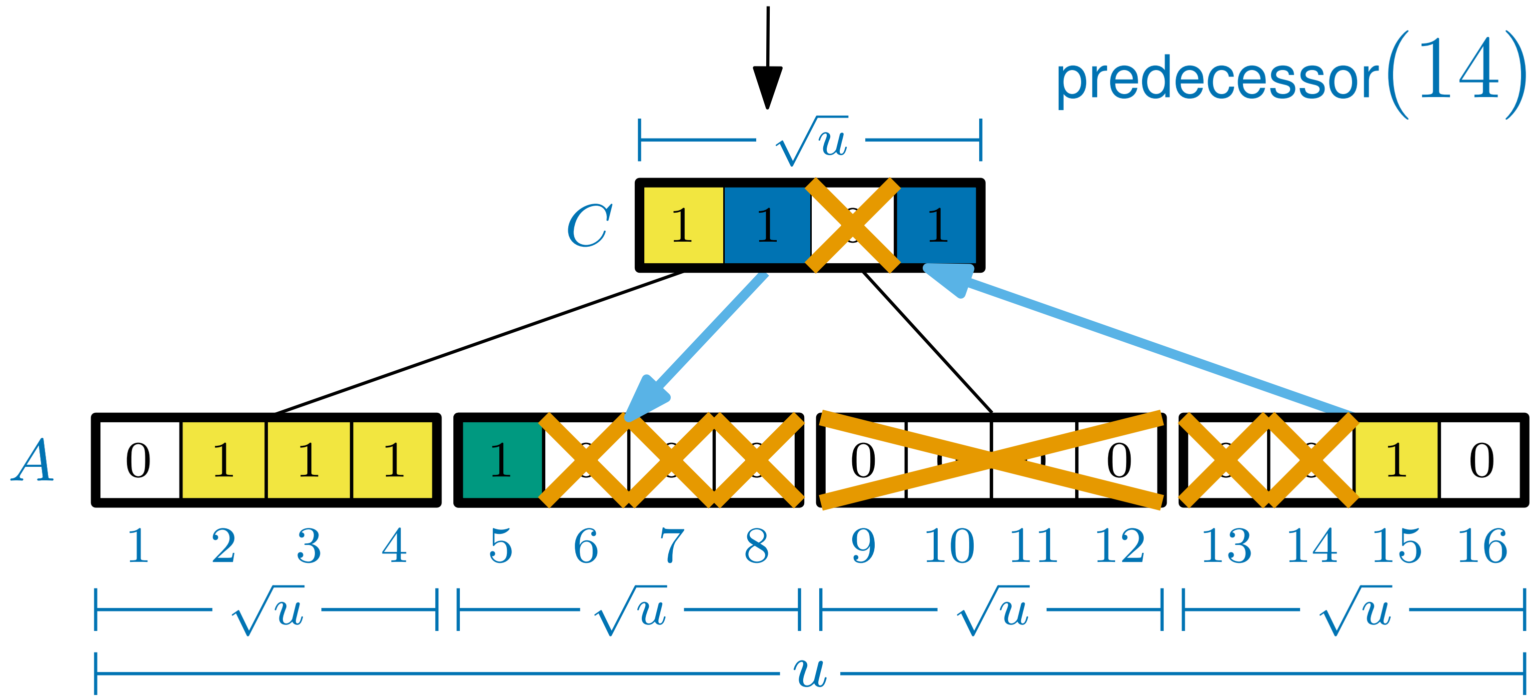
Attempt 2: a constant height tree

(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 2: a constant height tree

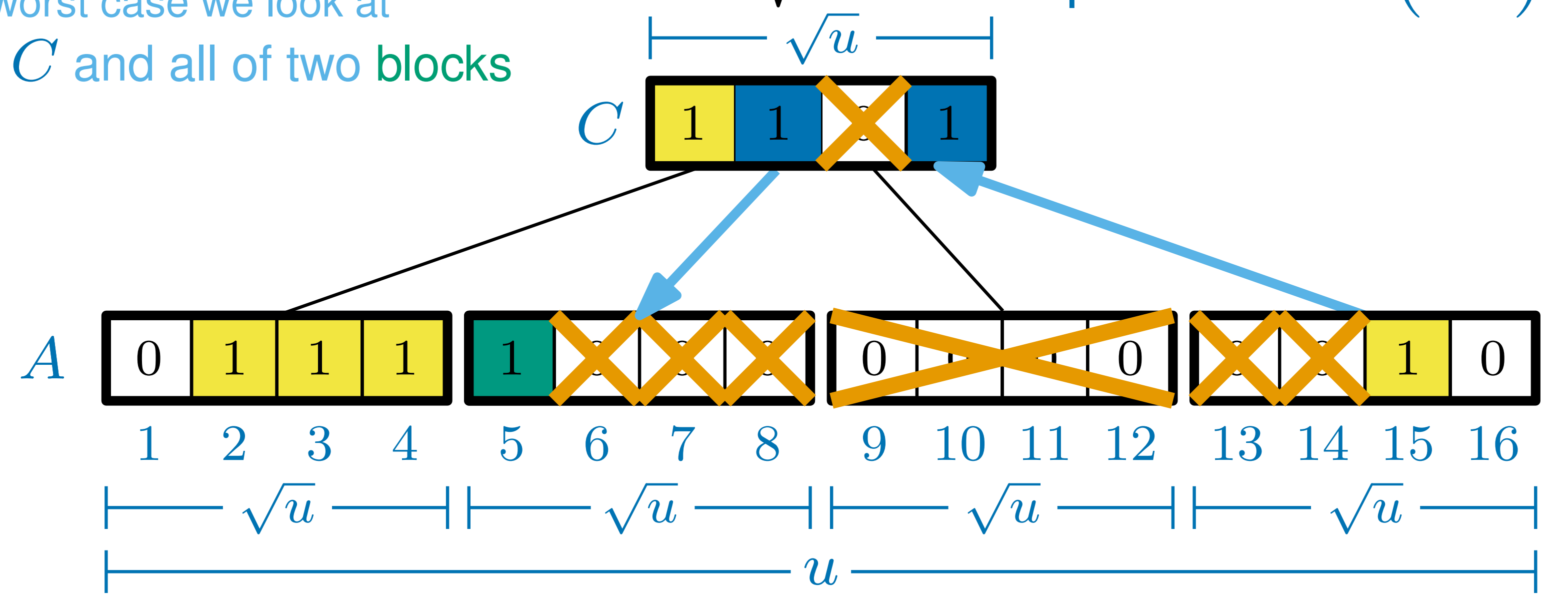
(on top of a big array)

C is called the *summary* of A

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

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.

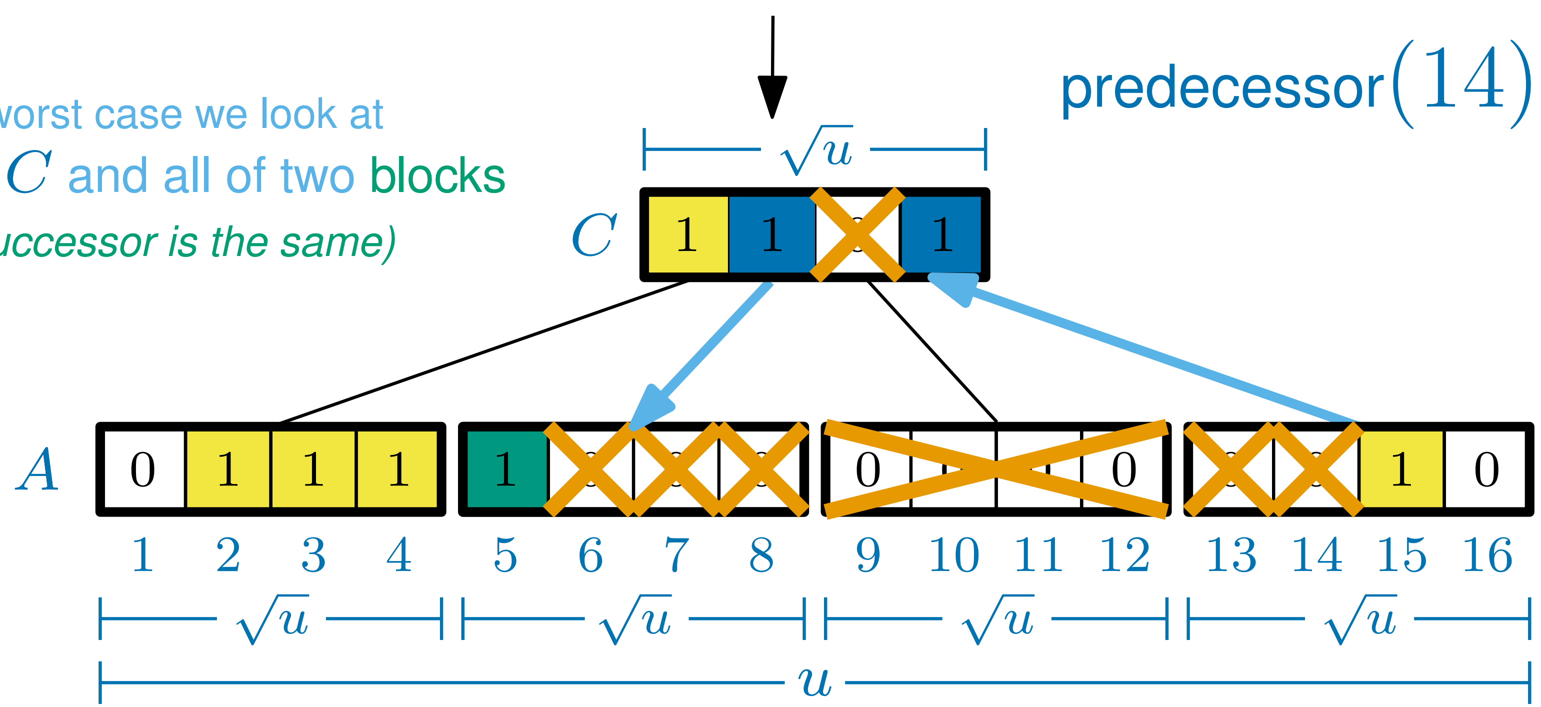
The operations delete, predecessor and successor take $O(\sqrt{u})$ time.

Attempt 2: 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)*



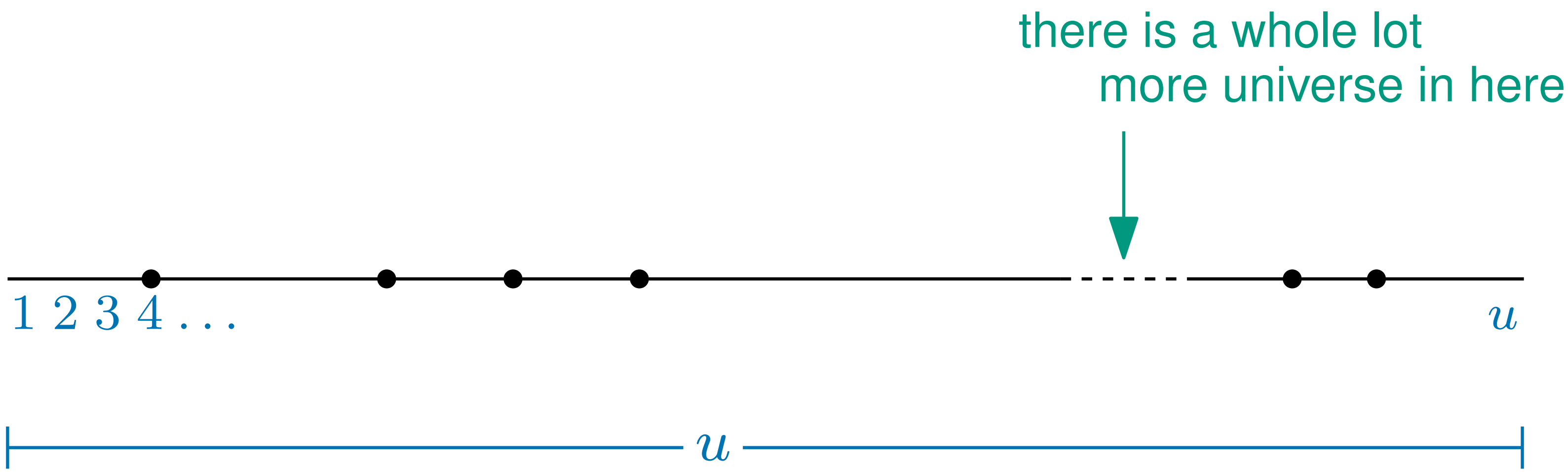
Split A into $\sqrt{u}$ *blocks* each containing $\sqrt{u}$ bits

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

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

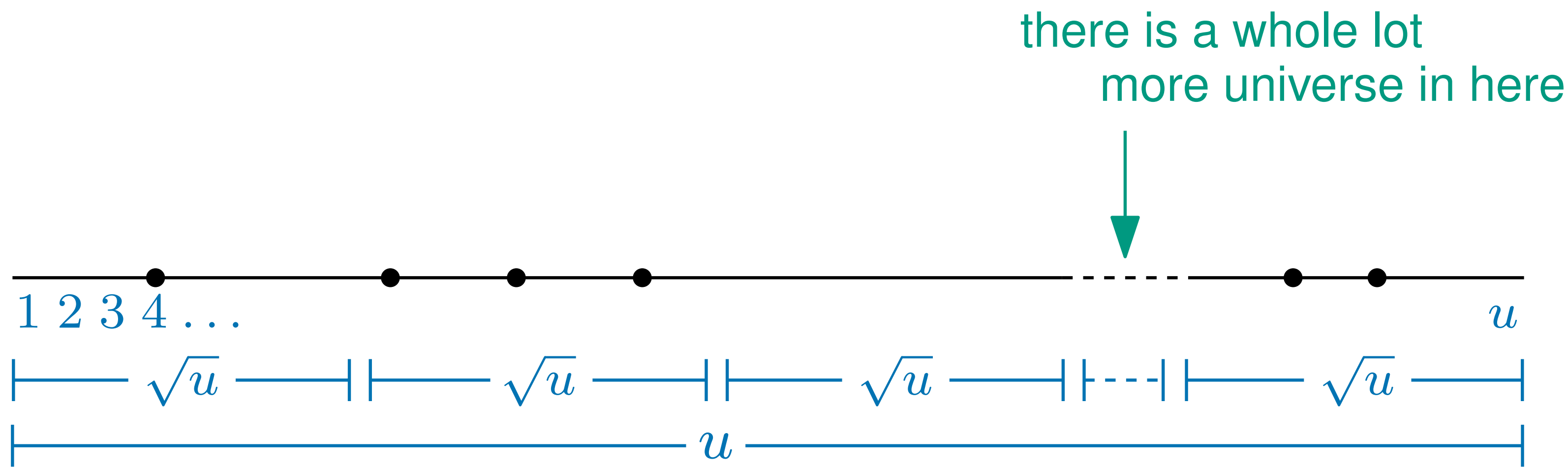
An abstract view

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



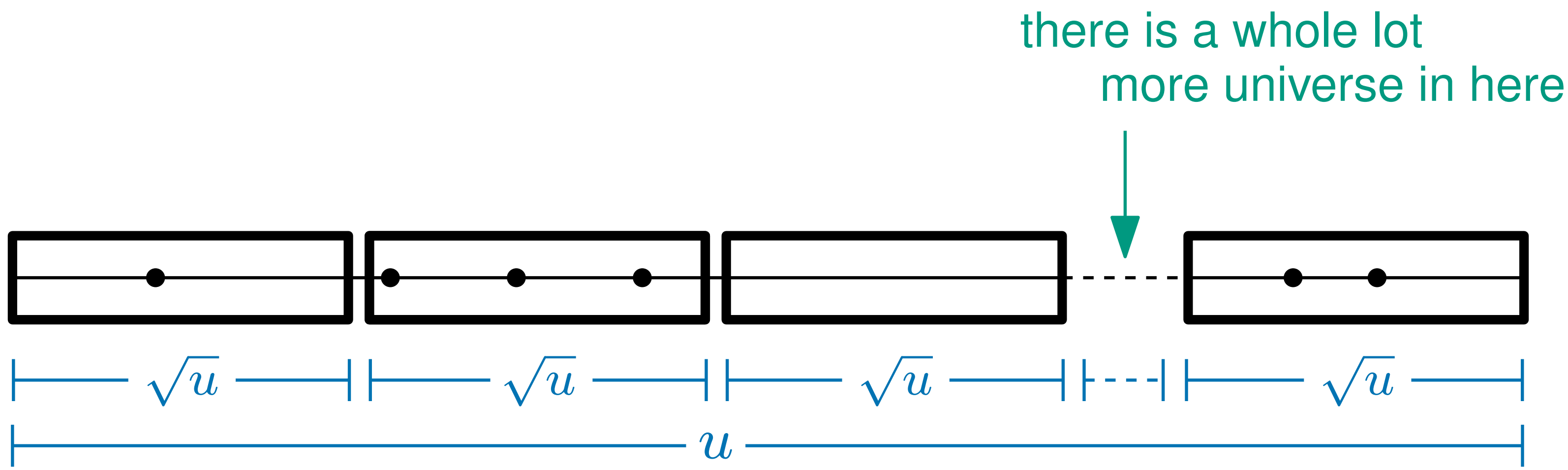
An abstract view

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

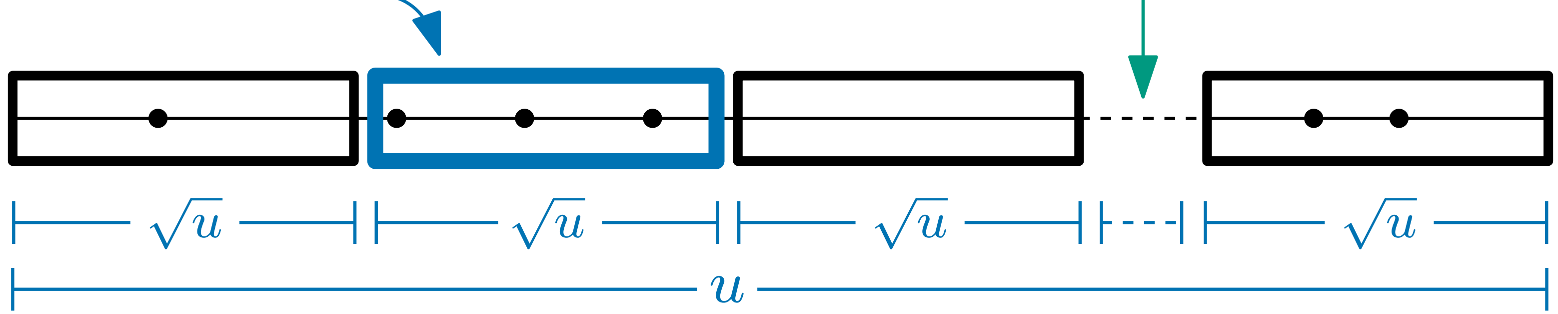


An abstract view

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 $\sqrt{u}$

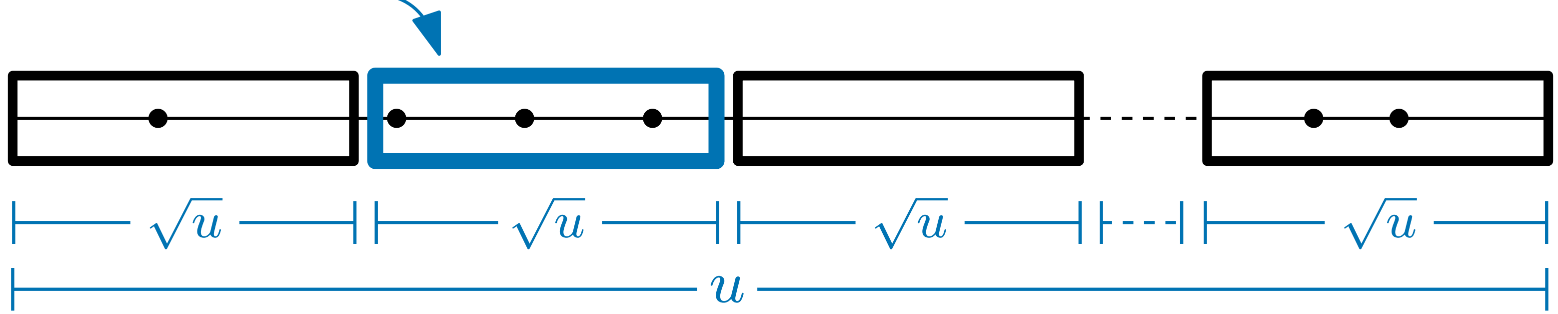
there is a whole lot
more universe in here



An abstract view

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

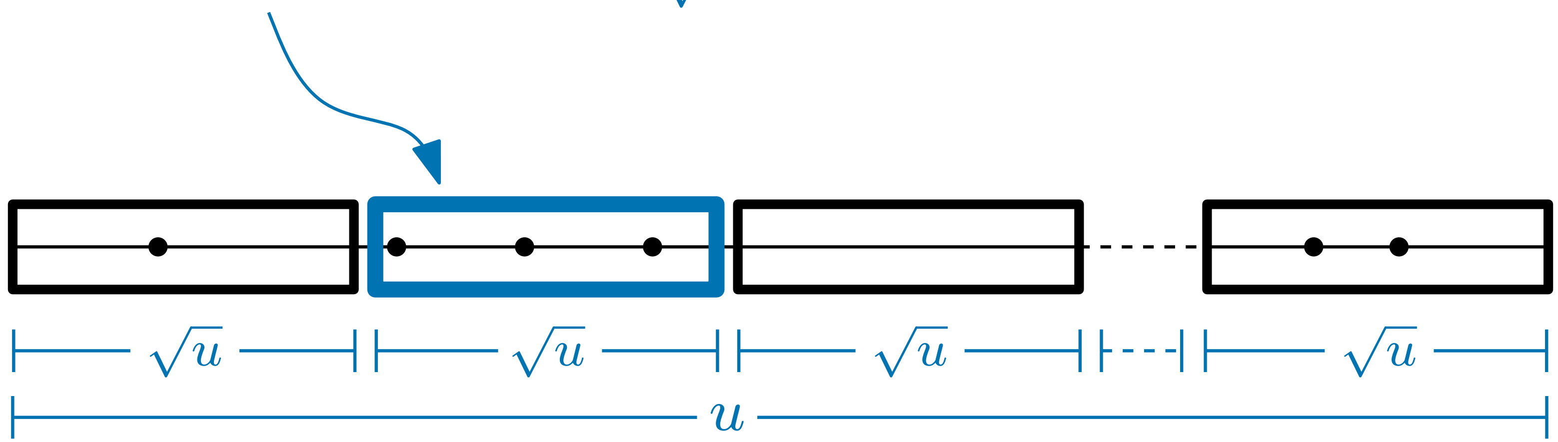
we can think of each block
as a *'little'* universe of size $\sqrt{u}$



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 $\sqrt{u}$



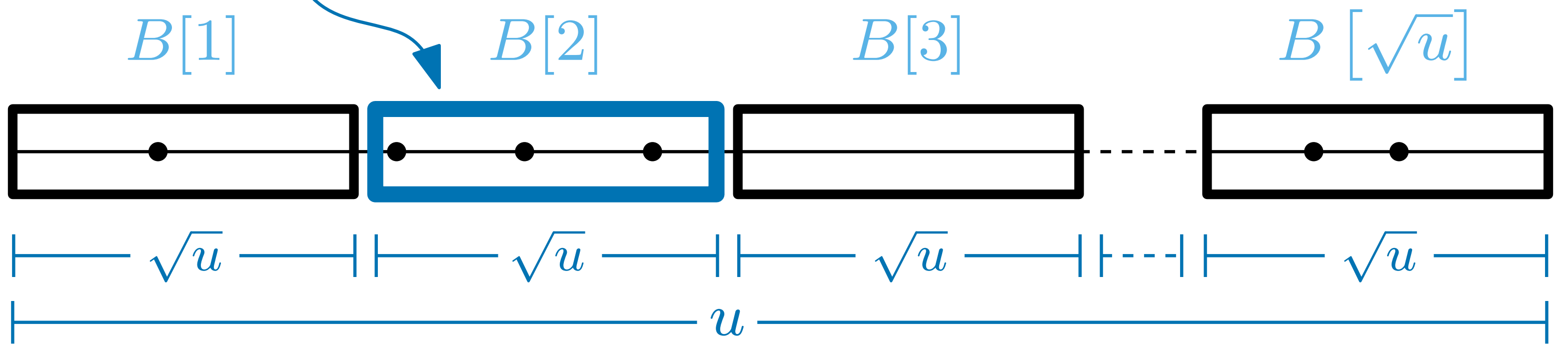
For block i, we build a data structure $B[i]$

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

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 $\sqrt{u}$

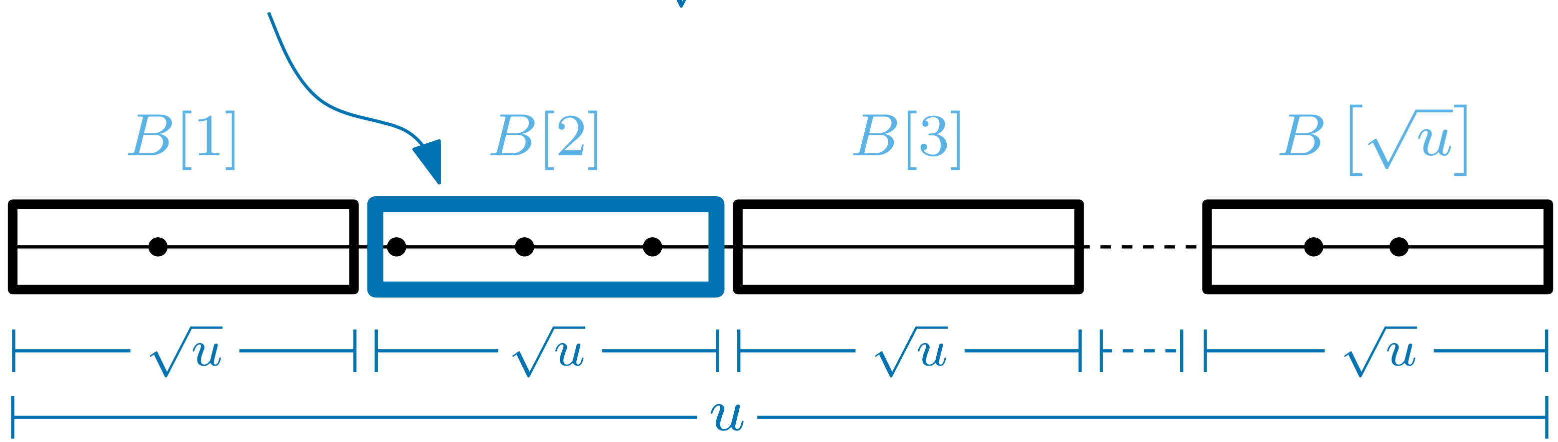


For block i, we build a data structure $B[i]$
which stores elements from $\{1, 2, 3, \ldots \sqrt{u}\}$

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 $\sqrt{u}$



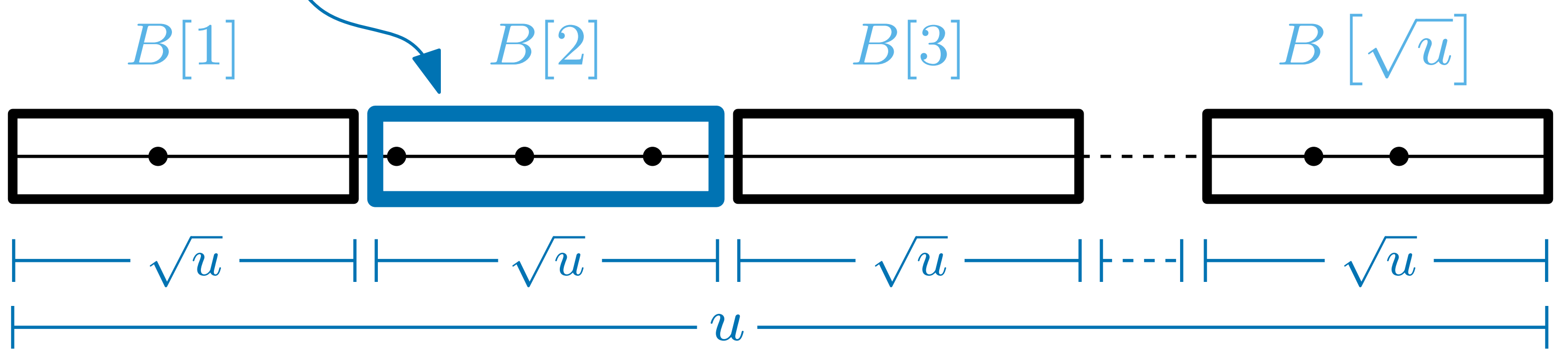
For block i, we build a data structure $B[i]$
which stores elements from $\{1, 2, 3, \ldots \sqrt{u}\}$

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

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 $\sqrt{u}$



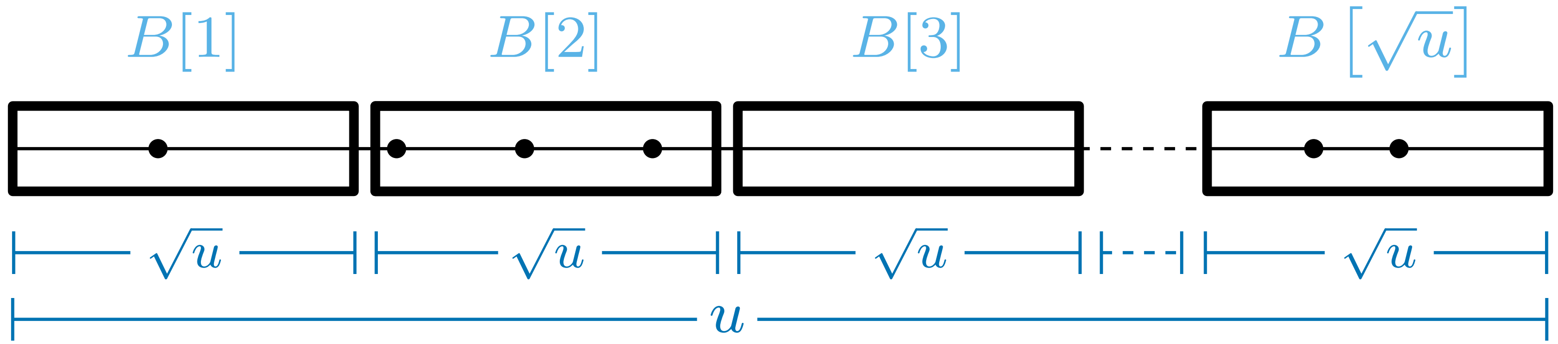
For block i, we build a data structure $B[i]$
which stores elements from $\{1, 2, 3, \ldots \sqrt{u}\}$

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

(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 $B[i]$

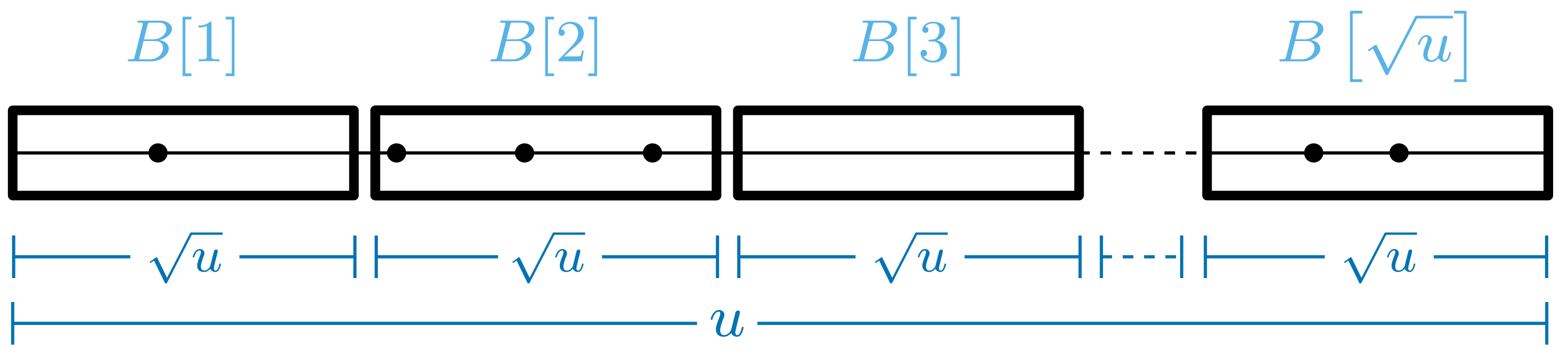
which stores elements from $\{1, 2, 3, \ldots \sqrt{u}\}$

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

(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



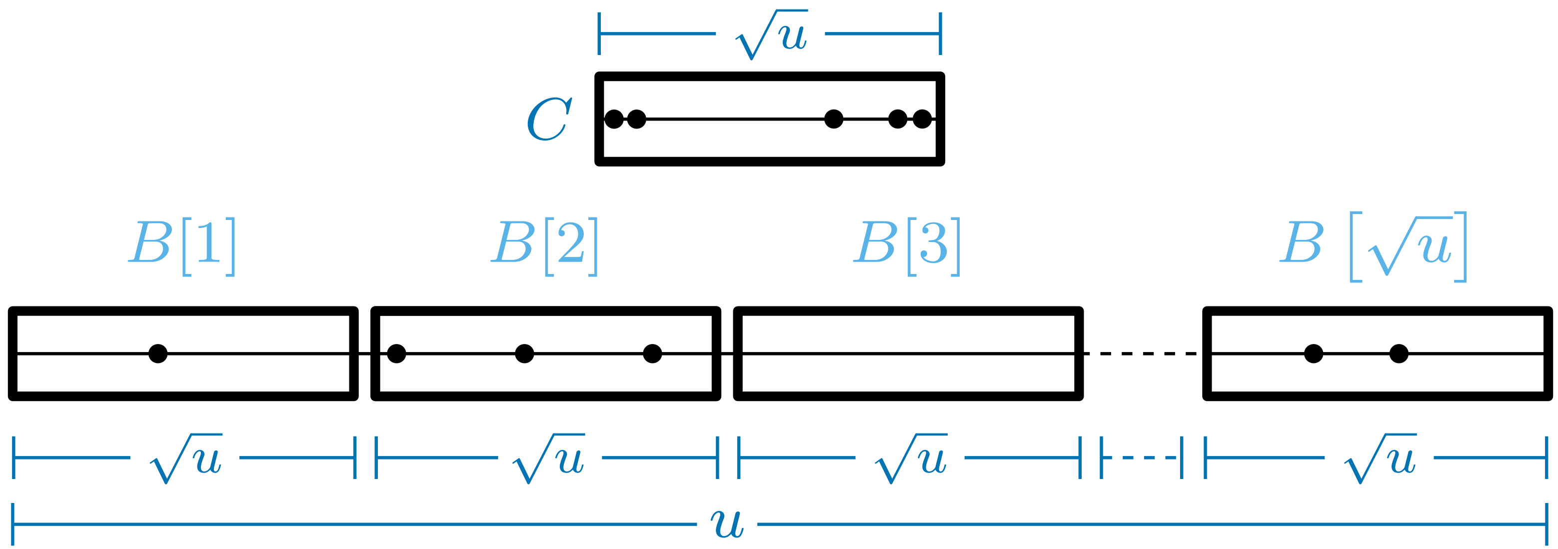
For block i, we build a data structure $B[i]$

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

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

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 $\left(x + (i-1)\sqrt{u}\right) \in S$

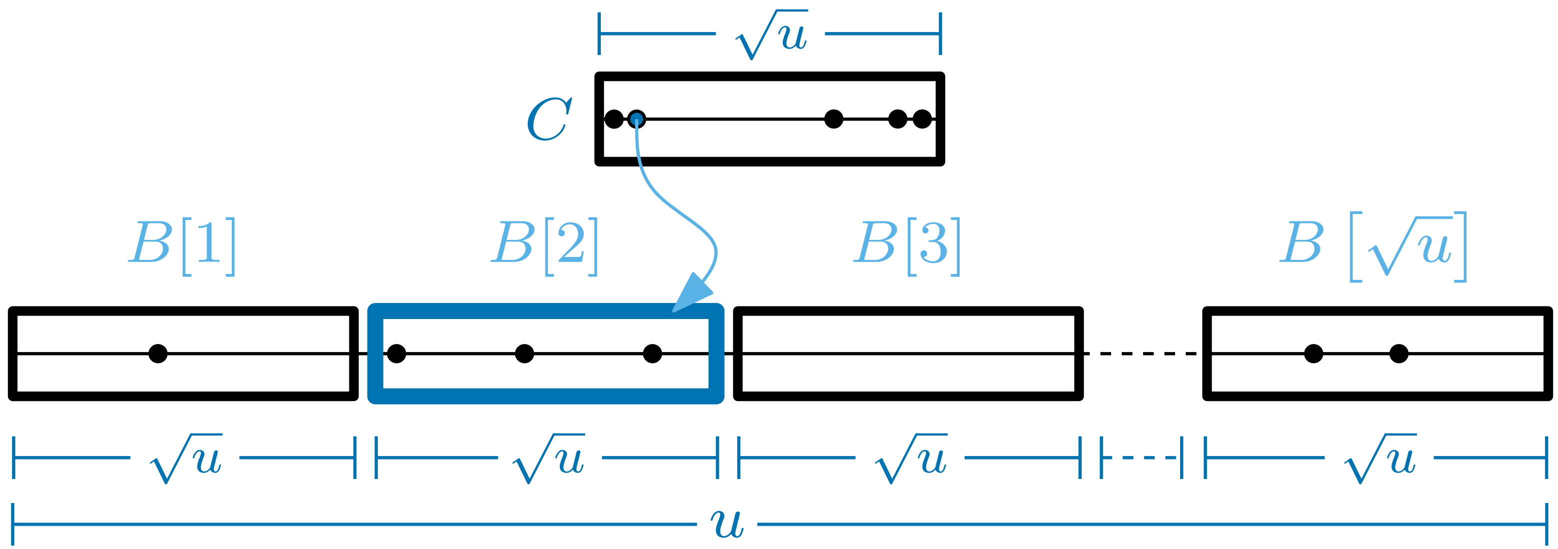
We also build a summary data structure C

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 $\left(x + (i-1)\sqrt{u}\right) \in S$

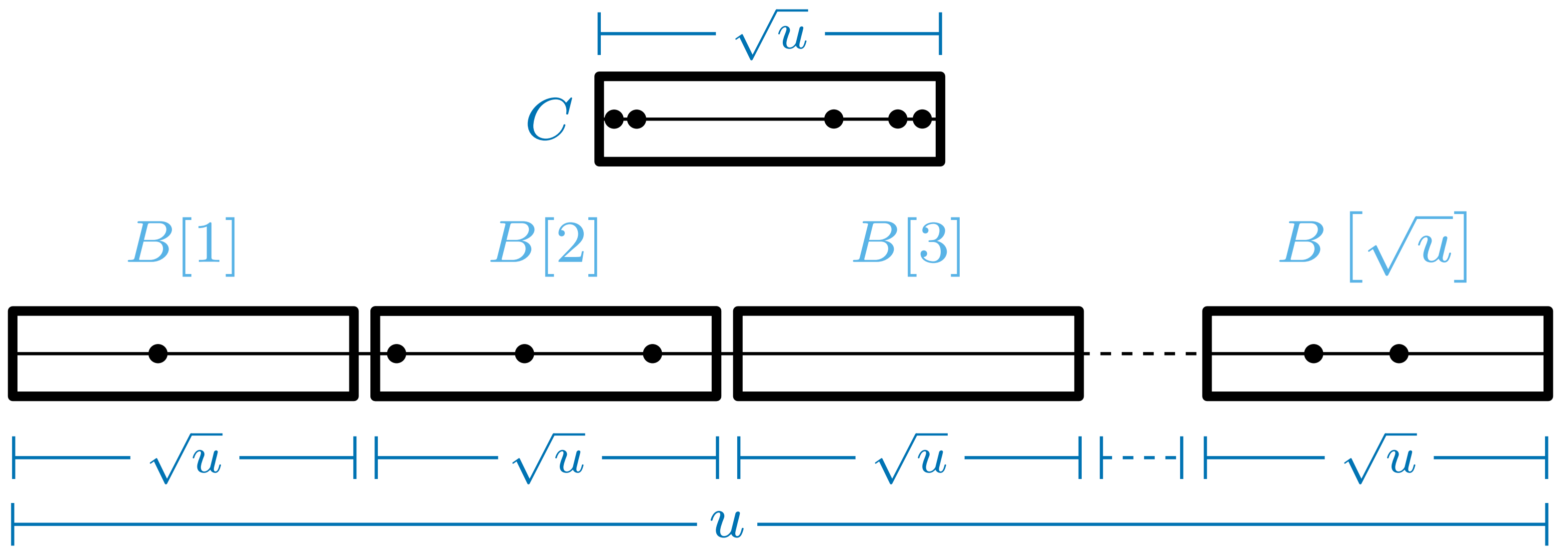
We also build a summary data structure C

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

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

An abstract view

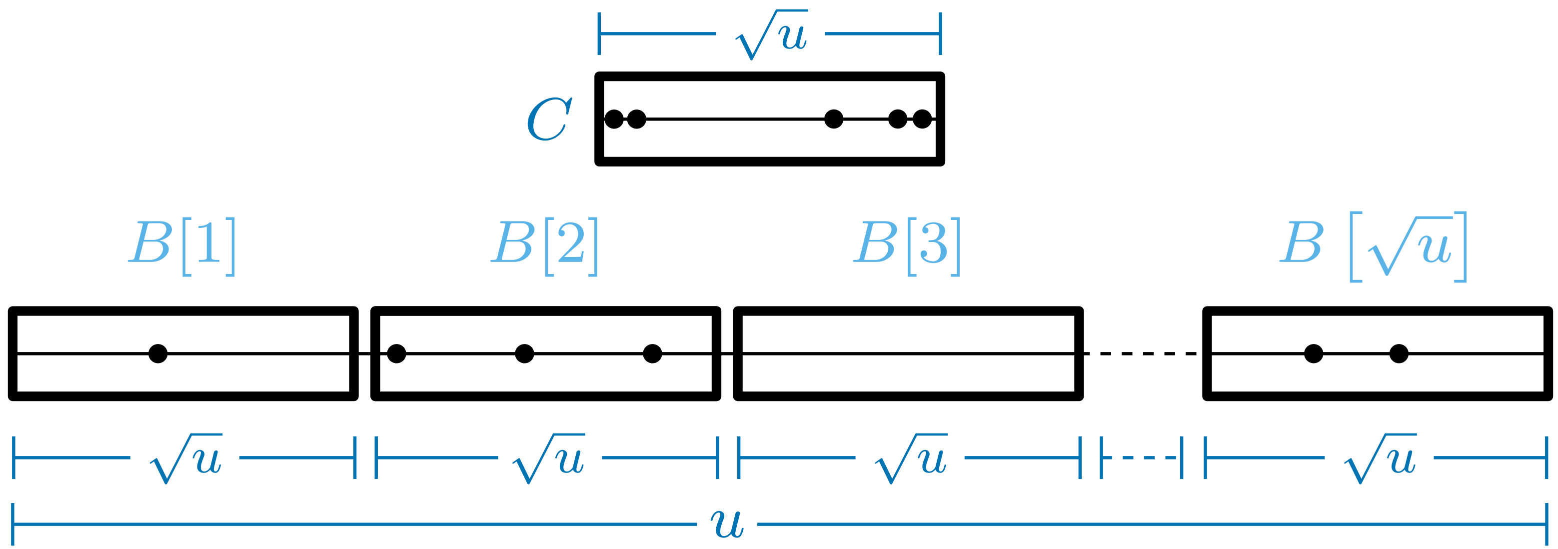
Split the universe U into $\sqrt{u}$ *blocks* each associated with $\sqrt{u}$ elements



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

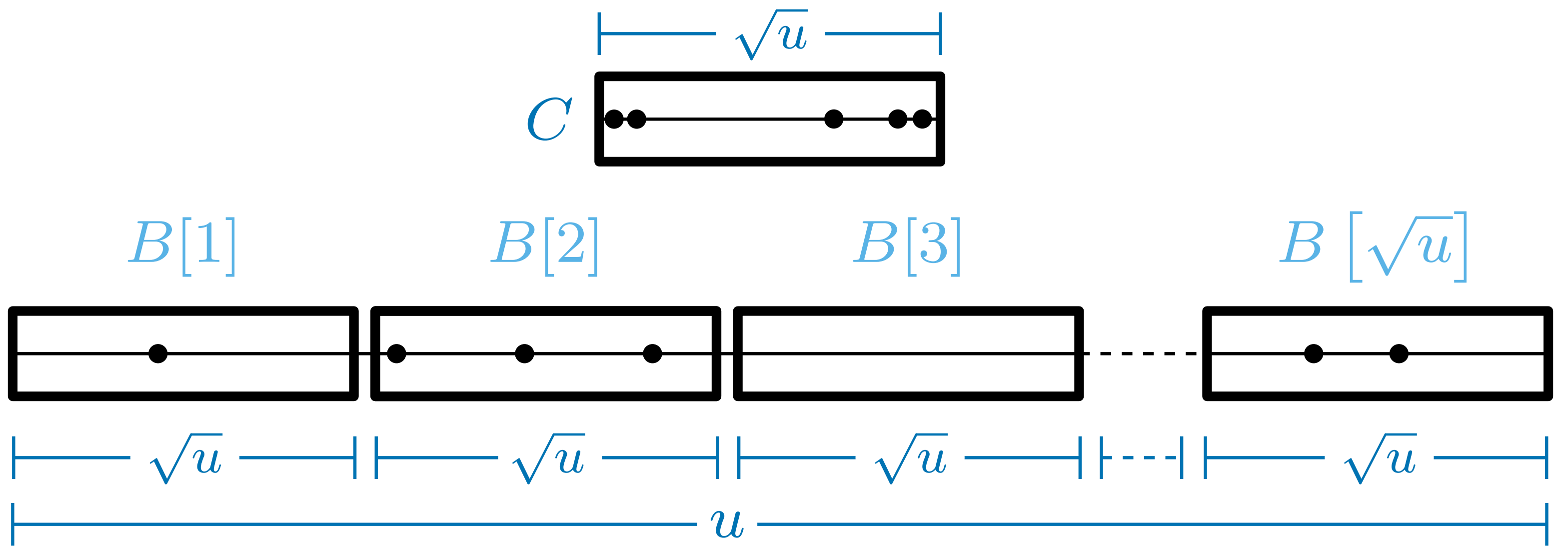


How should we build $B[1], B[2], \ldots B[\sqrt{u}]$ and C?

Recursion!

An abstract view

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



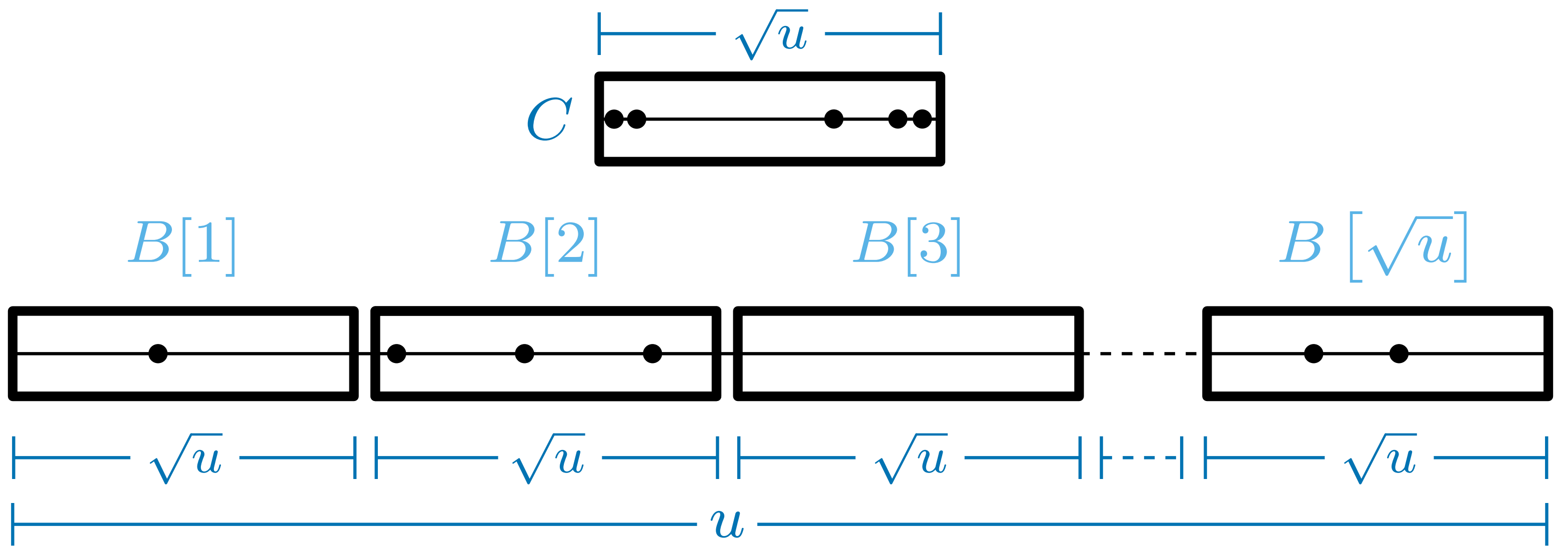
How should we build $B[1], B[2], \ldots B[\sqrt{u}]$ and C?

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



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

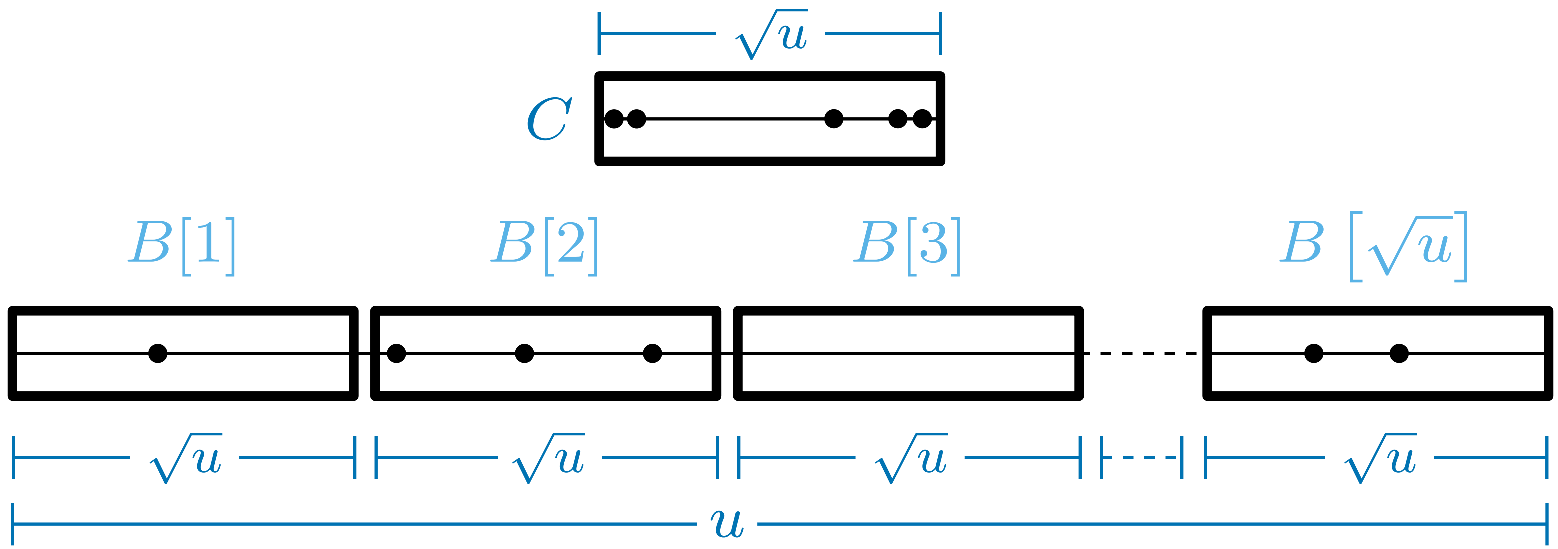
Recursion!

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!

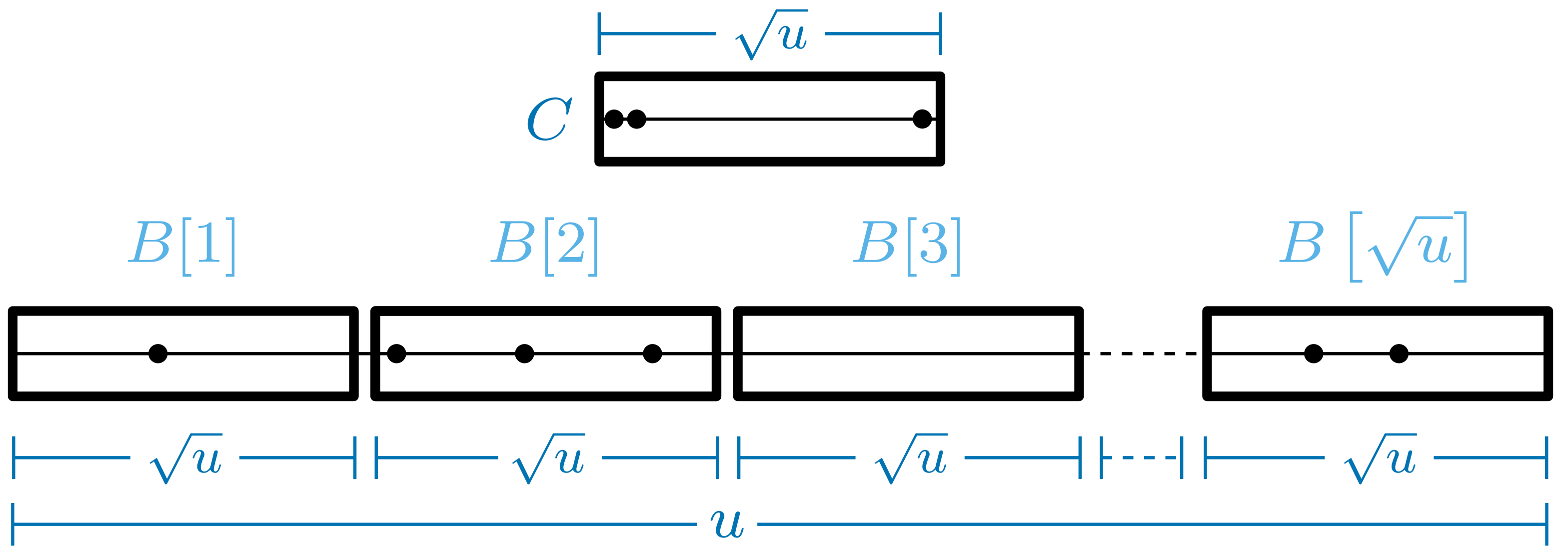
Each $B[i]$ has universe $\{1, 2, 3, \ldots \sqrt{u}\}$

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 $O(\log \log u)$ time solution

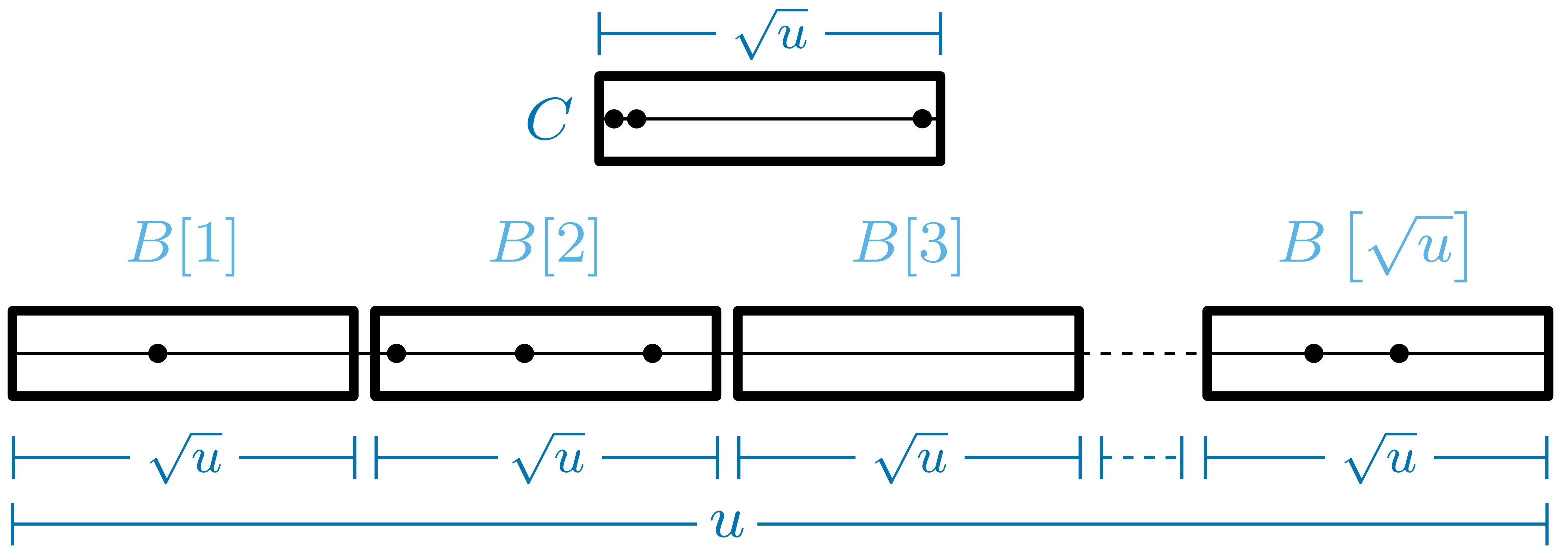
Attempt 3: Recursion

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



Attempt 3: Recursion

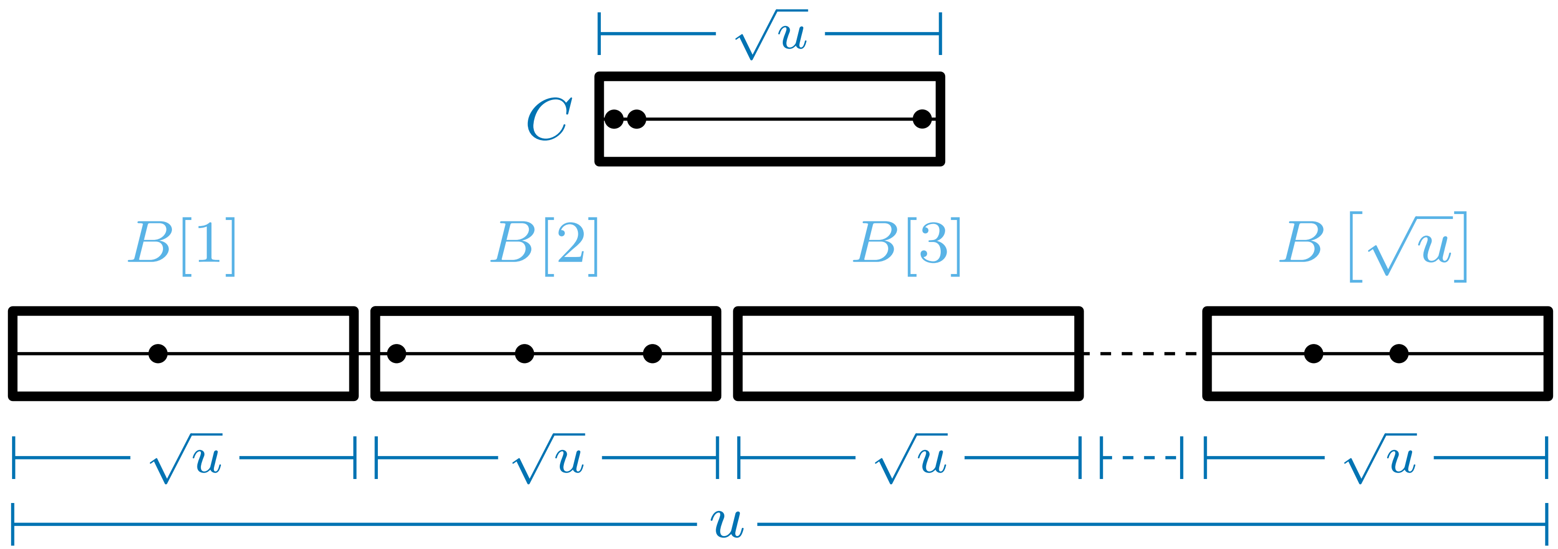
Split the universe U into $\sqrt{u}$ *blocks* each associated with $\sqrt{u}$ elements



How do we perform the operations?

Attempt 3: Recursion

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



To perform add(x):

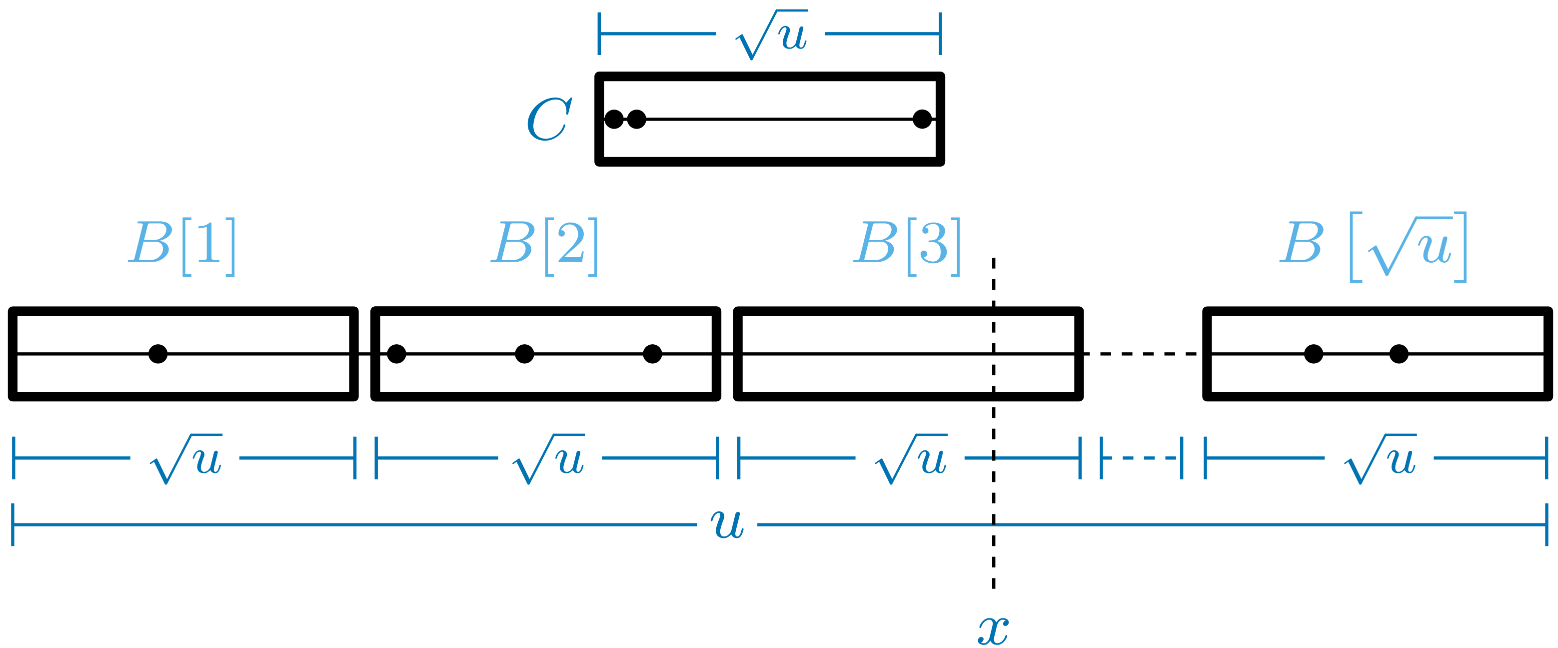
Step 1 Determine which $B[i]$ the element x belongs in
(this takes $O(1)$ time with a little bit twiddling)

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 3: Recursion

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



To perform add(x):

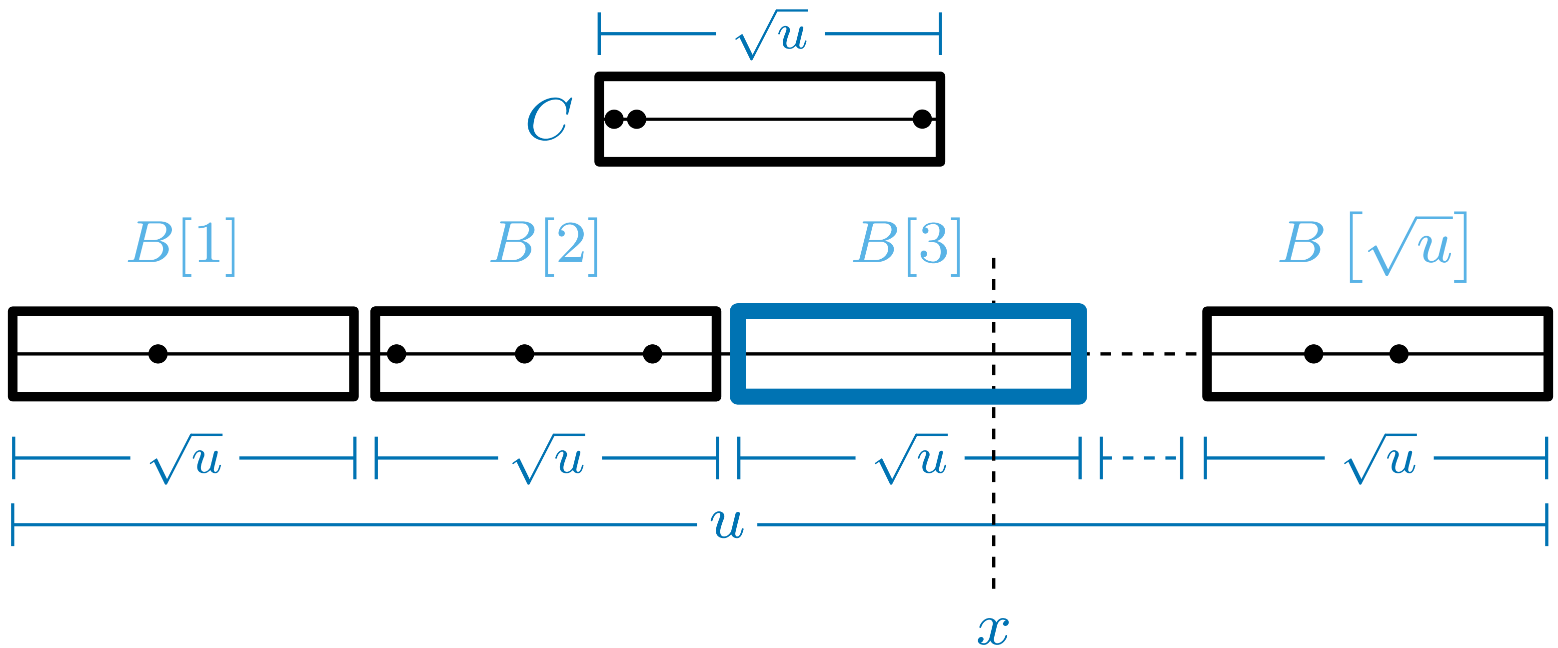
Step 1 Determine which $B[i]$ the element x belongs in
(this takes $O(1)$ time with a little bit twiddling)

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 3: 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

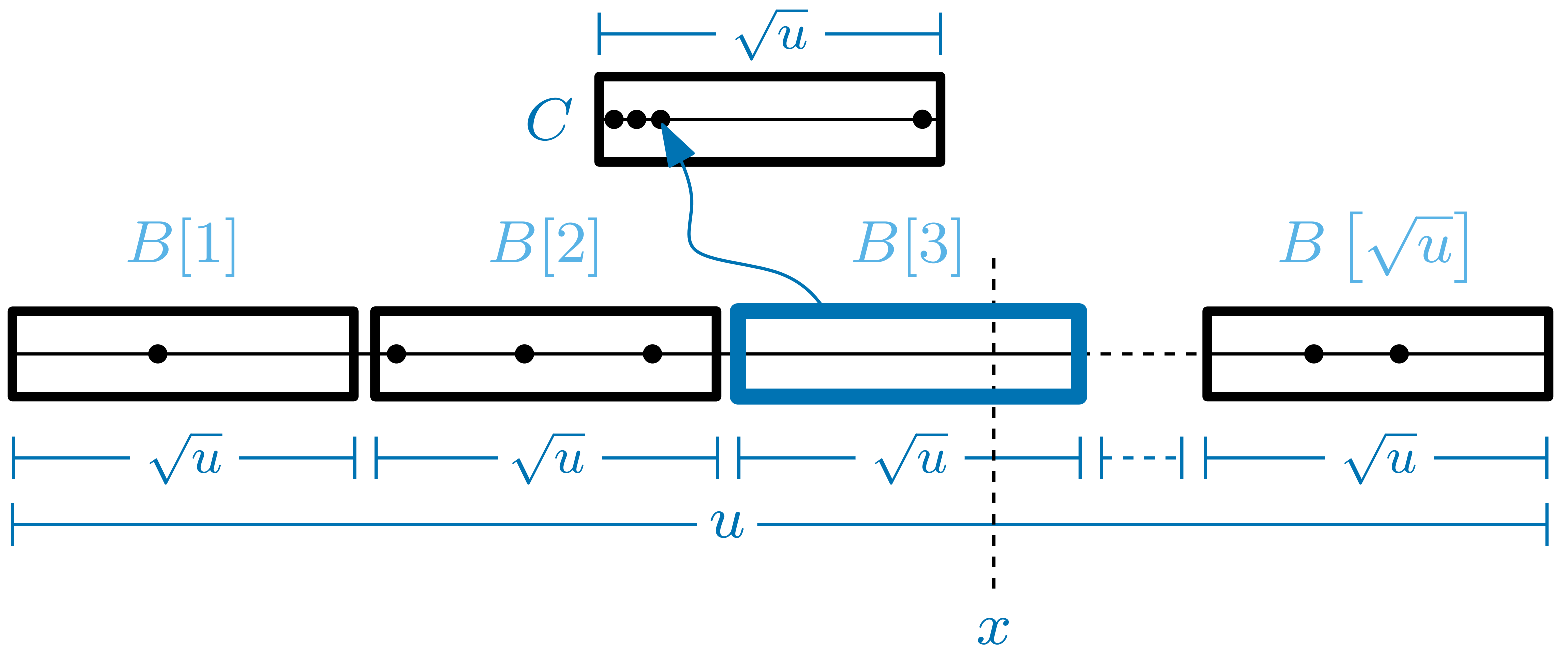
(this takes $O(1)$ time with a little bit twiddling)

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 3: Recursion

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



To perform add(x):

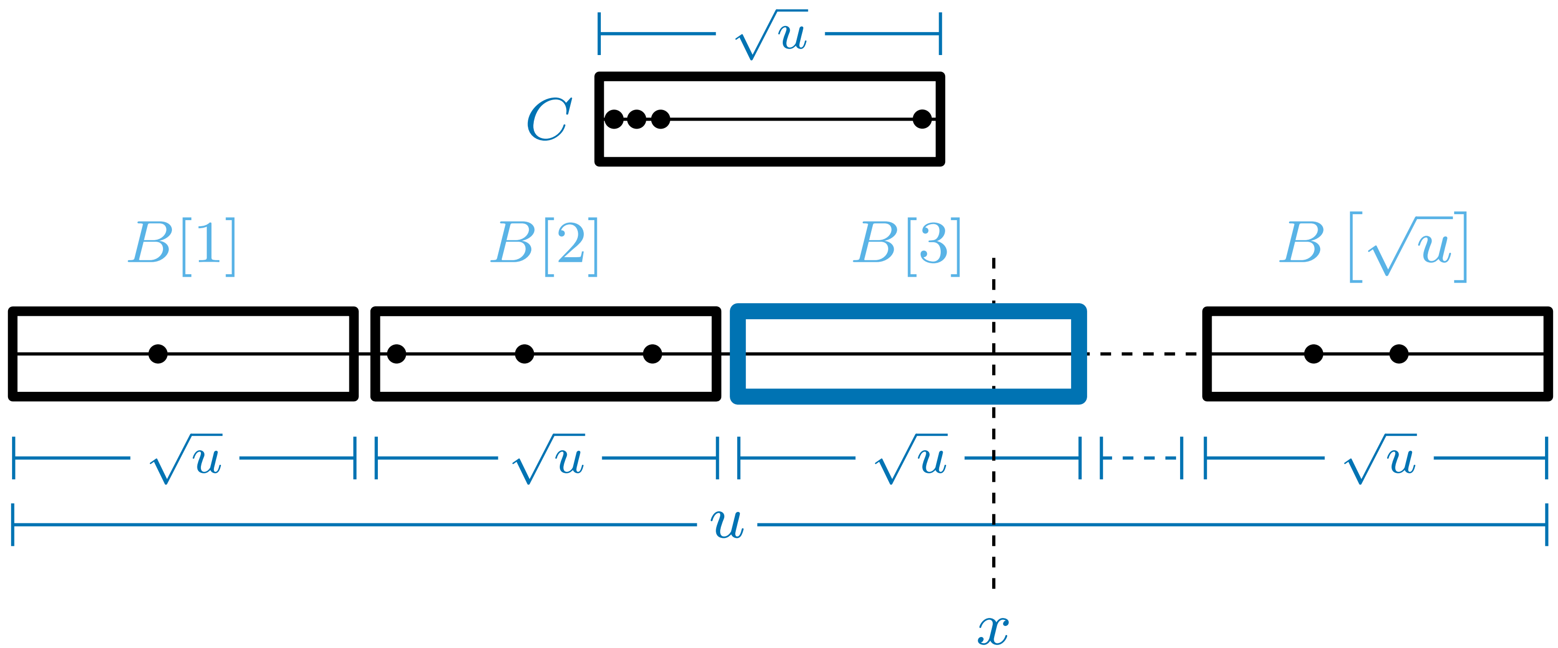
Step 1 Determine which $B[i]$ the element x belongs in
(this takes $O(1)$ time with a little bit twiddling)

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 3: 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

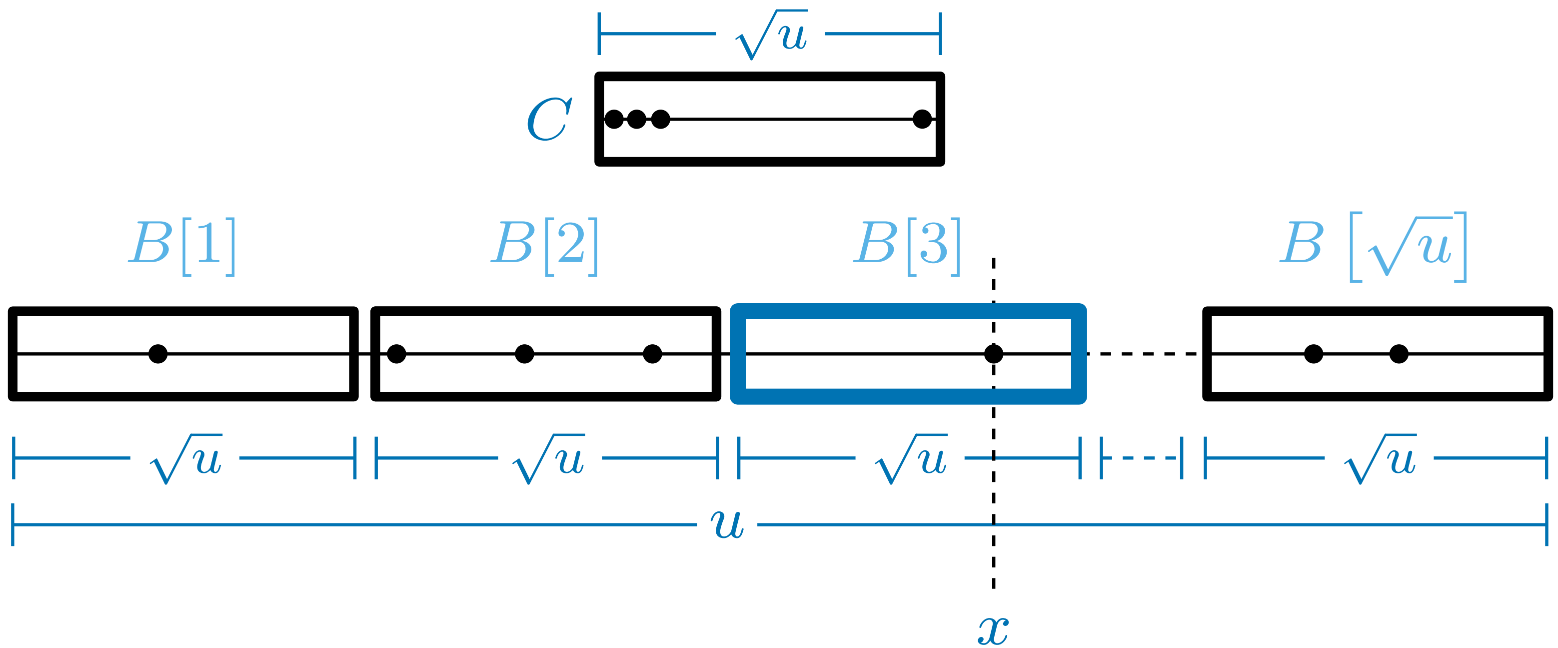
(this takes $O(1)$ time with a little bit twiddling)

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 3: Recursion

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



To perform add(x):

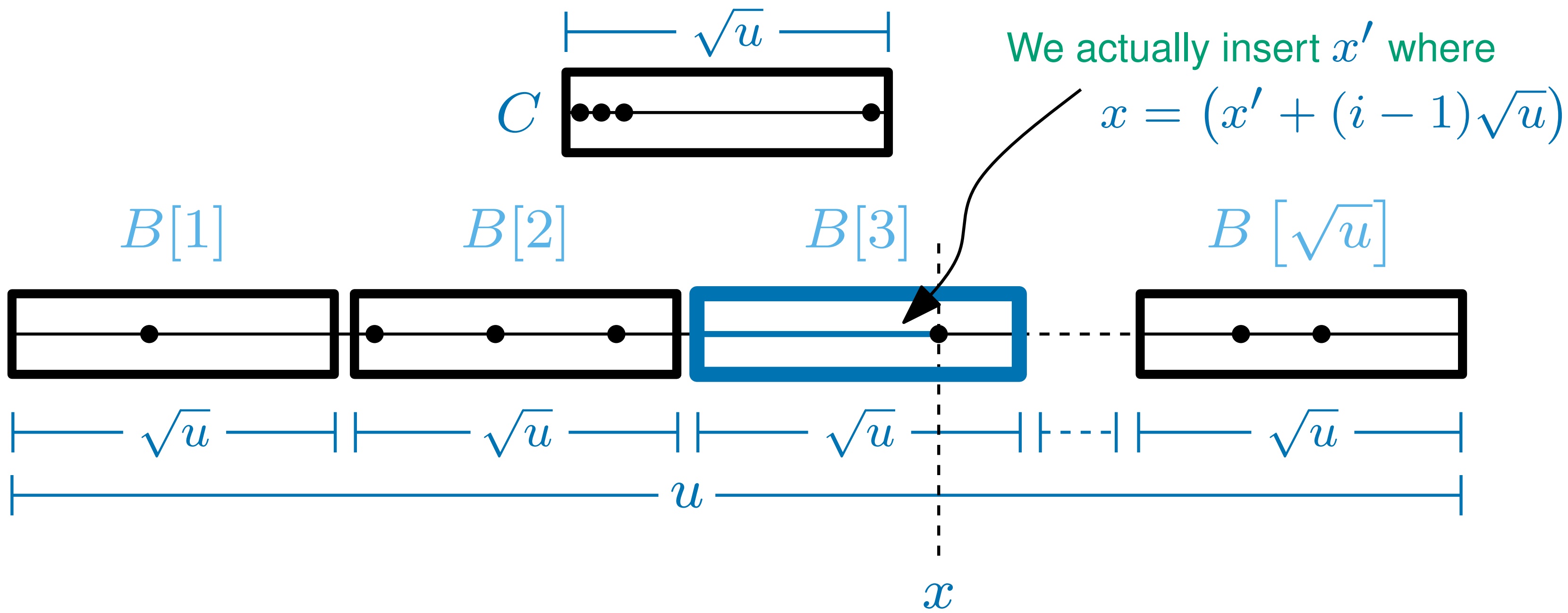
Step 1 Determine which $B[i]$ the element x belongs in
(this takes $O(1)$ time with a little bit twiddling)

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 3: 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

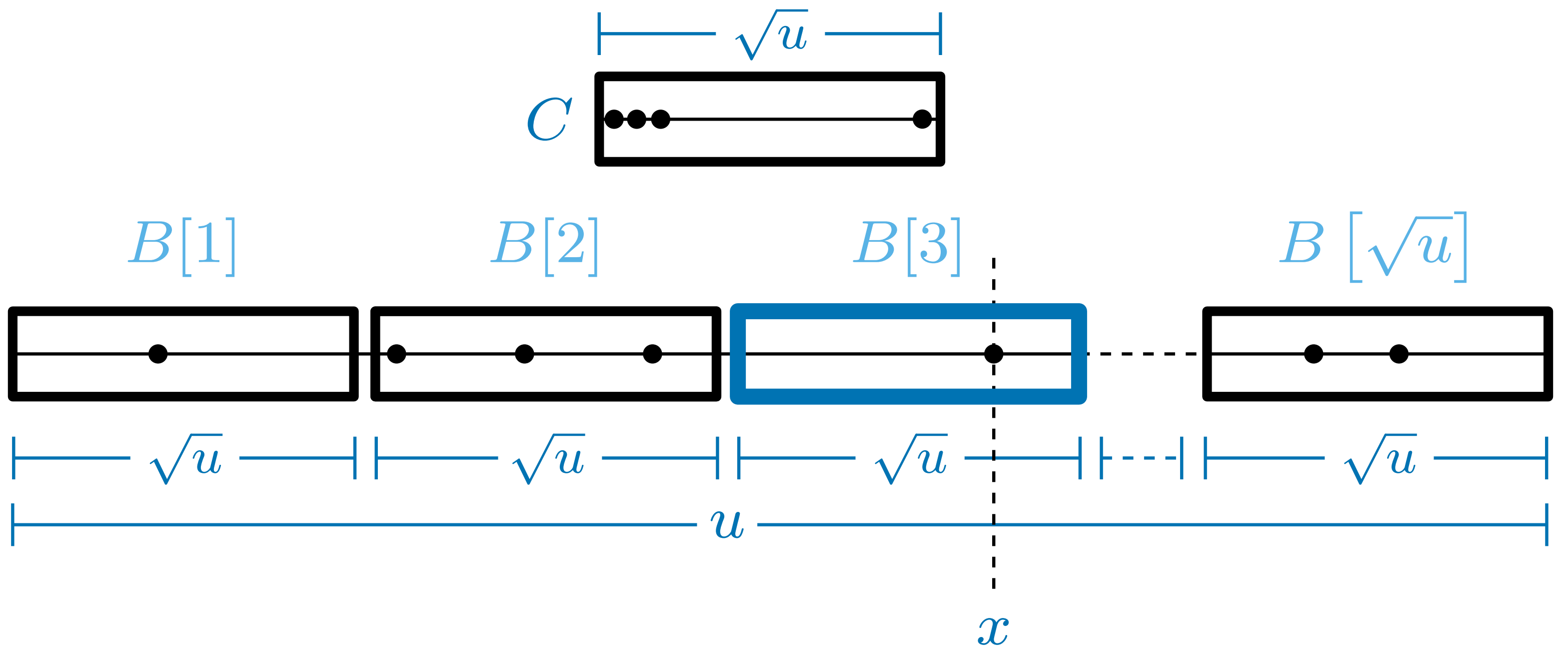
(this takes $O(1)$ time with a little bit twiddling)

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 3: Recursion

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



To perform add(x):

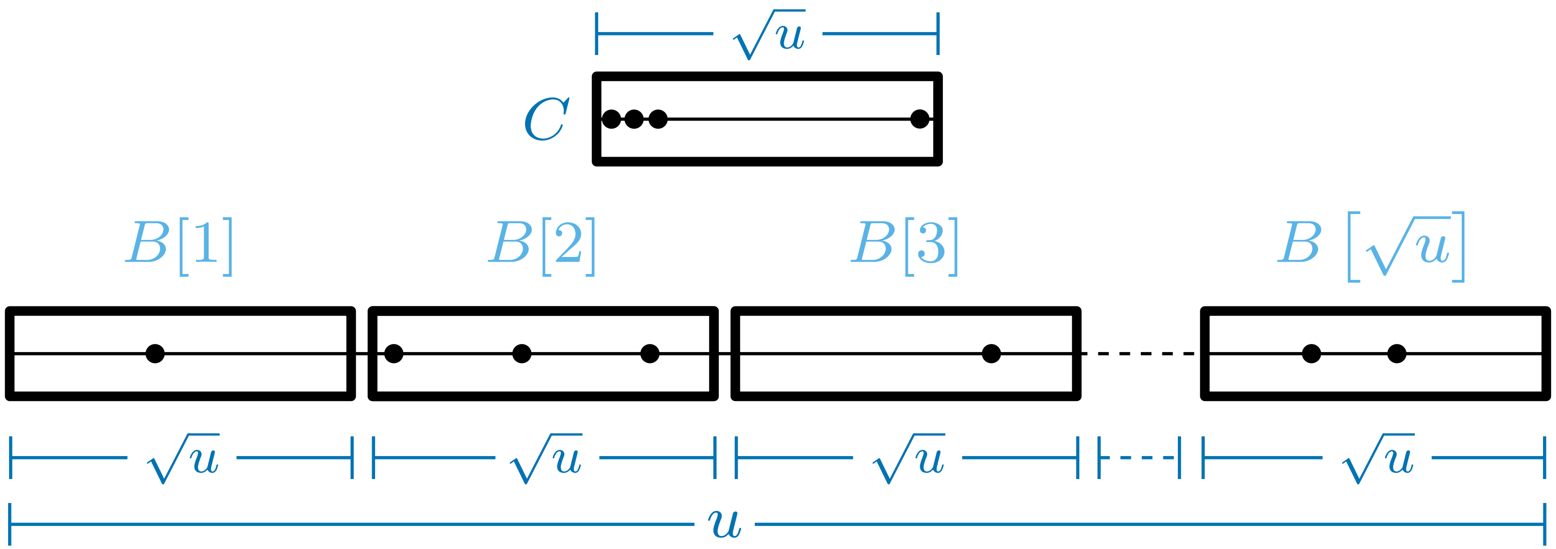
Step 1 Determine which $B[i]$ the element x belongs in
(this takes $O(1)$ time with a little bit twiddling)

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 3: 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

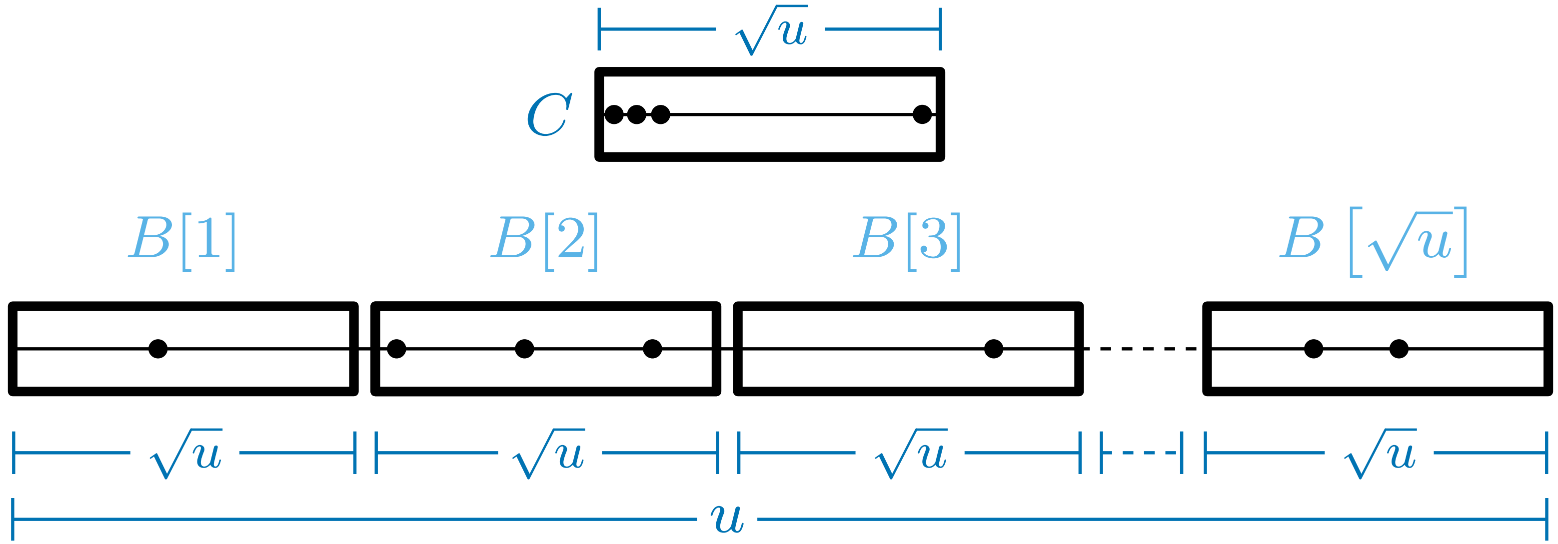
(this takes $O(1)$ time with a little bit twiddling)

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 $B[i]$)

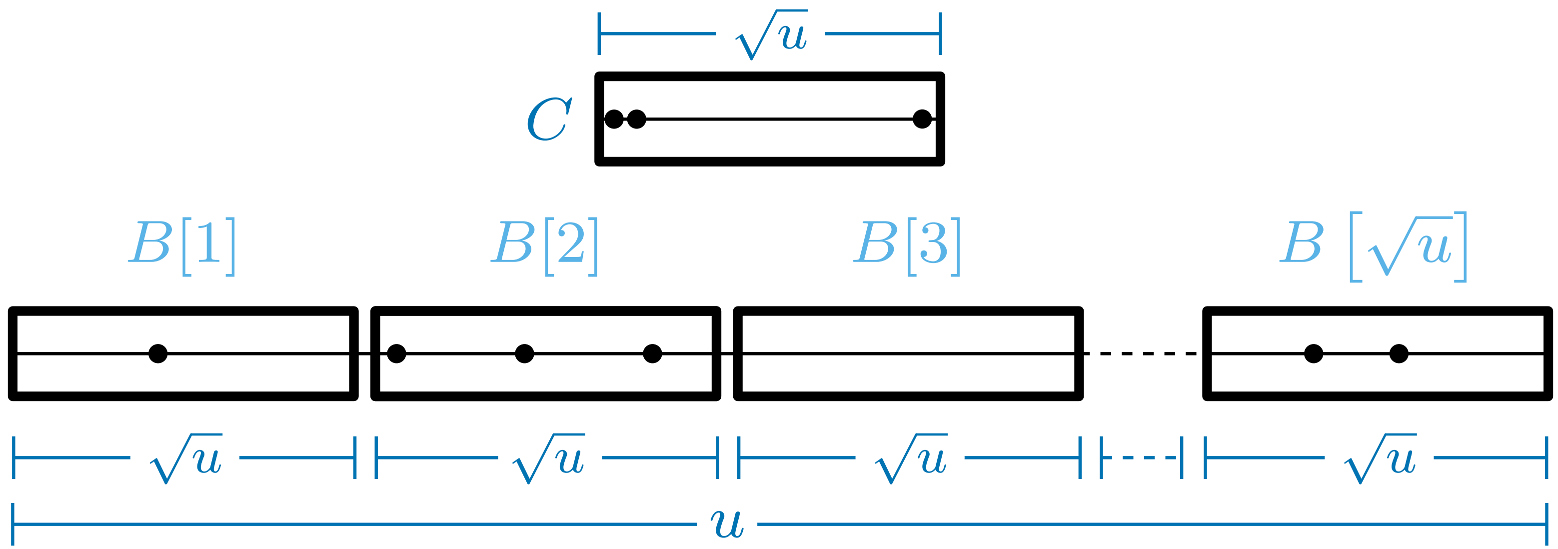
Attempt 3: Recursion

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



Attempt 3: Recursion

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



To perform 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 $B[i]$:

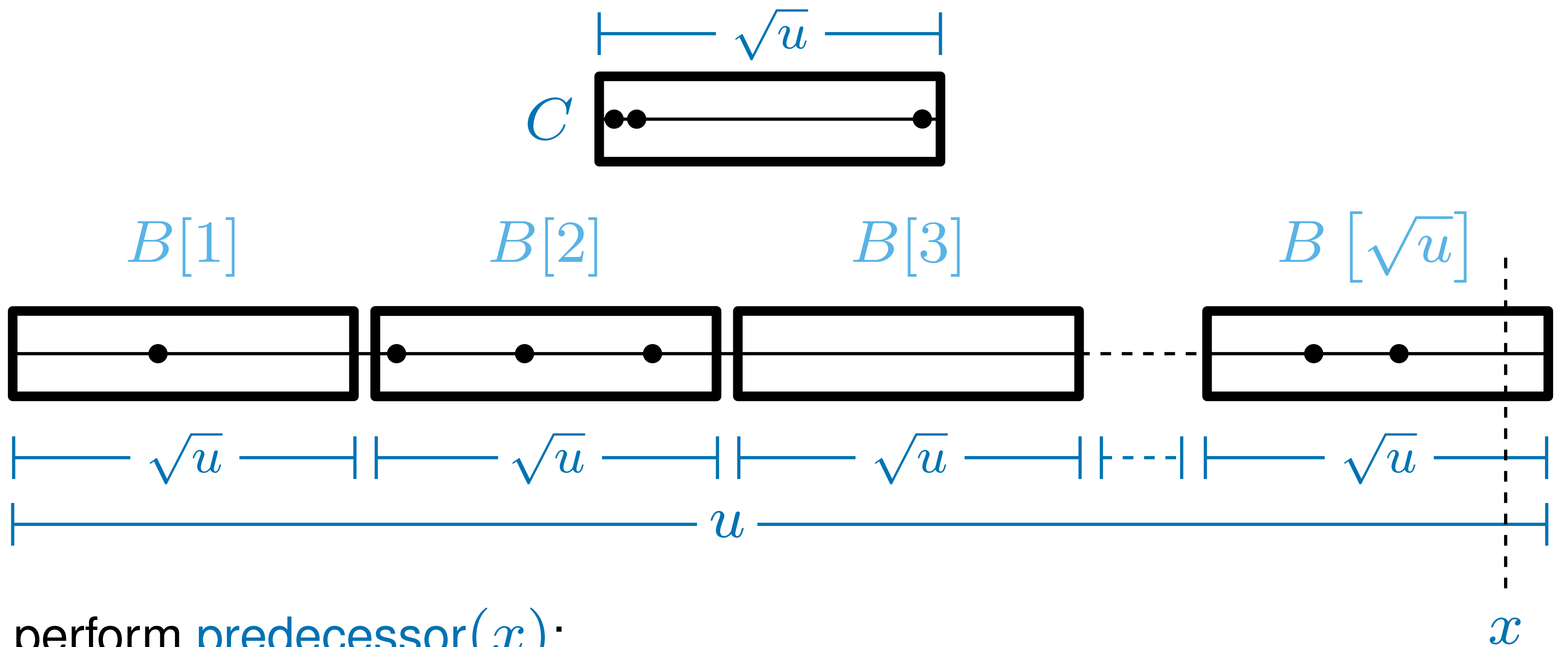
Compute $j =$ predecessor(i) in C

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

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

Attempt 3: Recursion

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



To perform 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 $B[i]$:

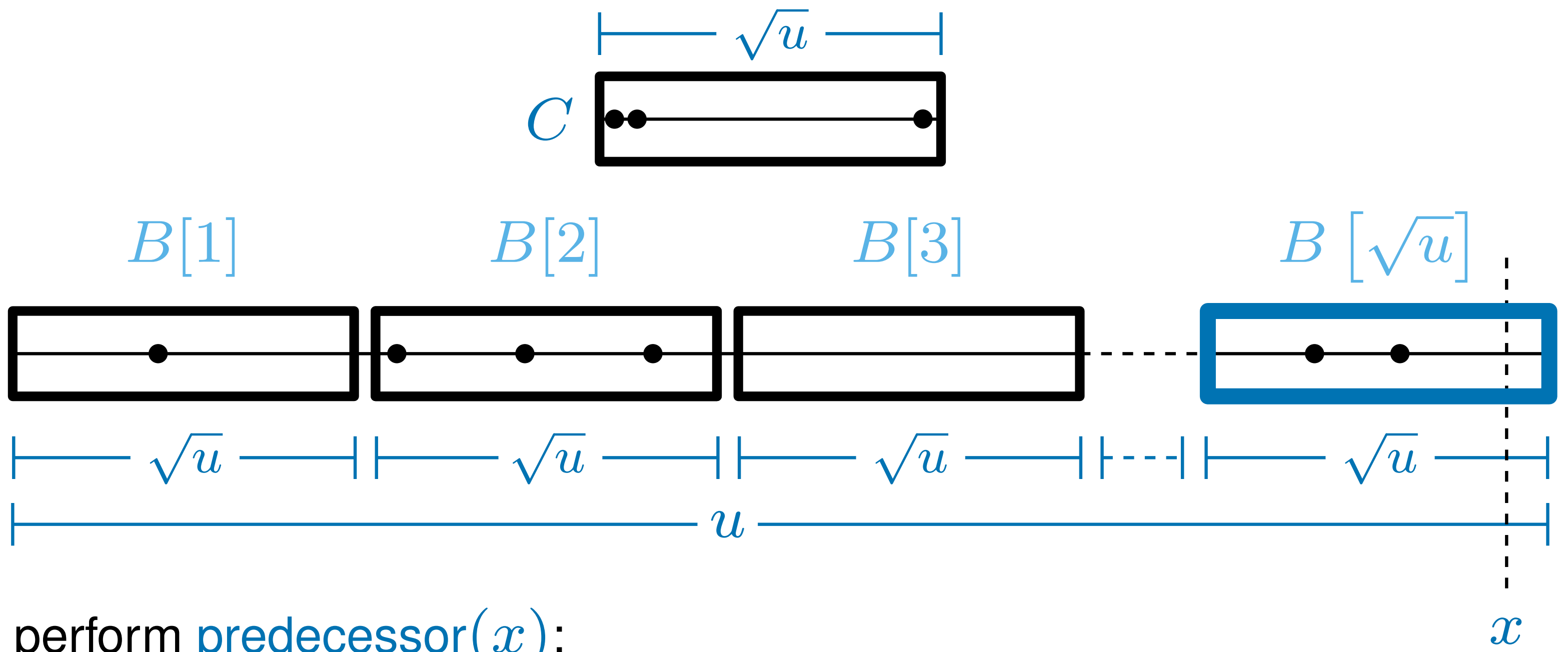
Compute $j =$ predecessor(i) in C

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

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

Attempt 3: Recursion

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



To perform 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 $B[i]$:

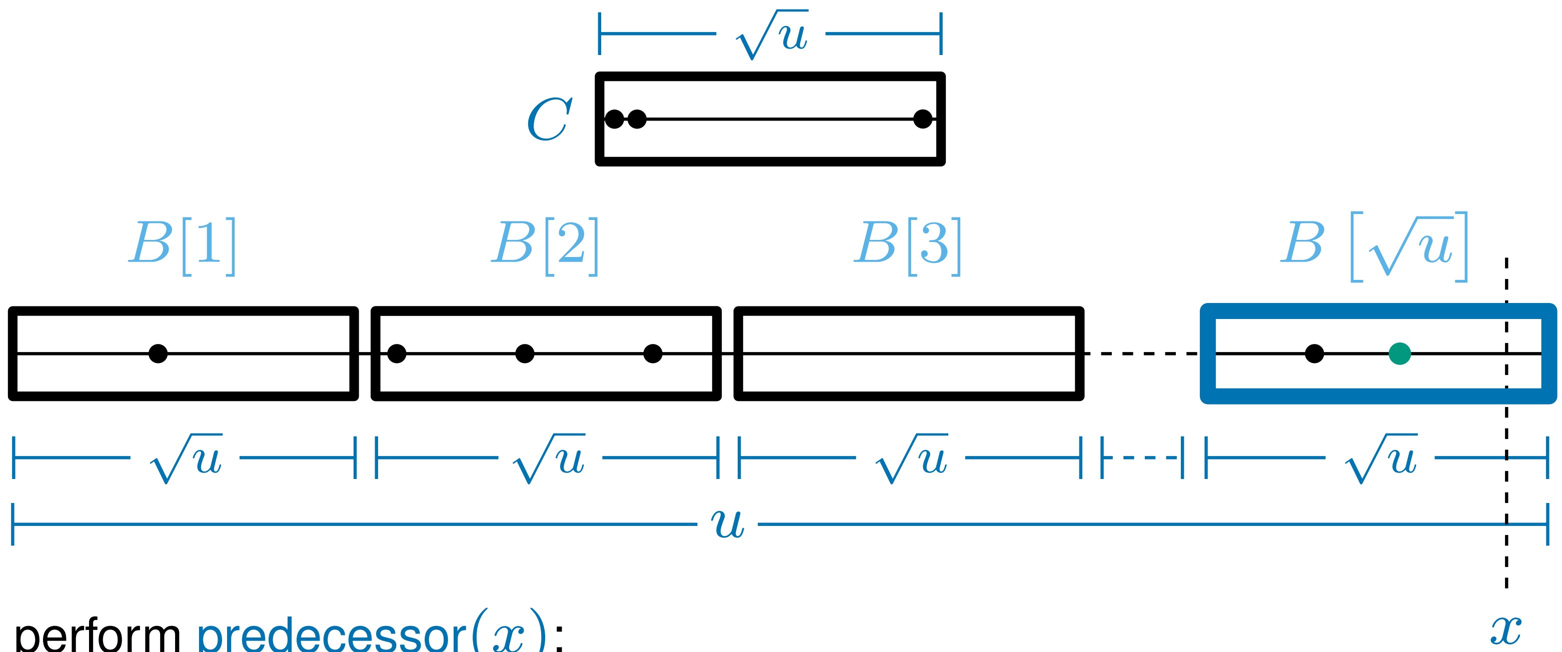
Compute $j =$ predecessor(i) in C

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

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

Attempt 3: Recursion

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



To perform 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 $B[i]$:

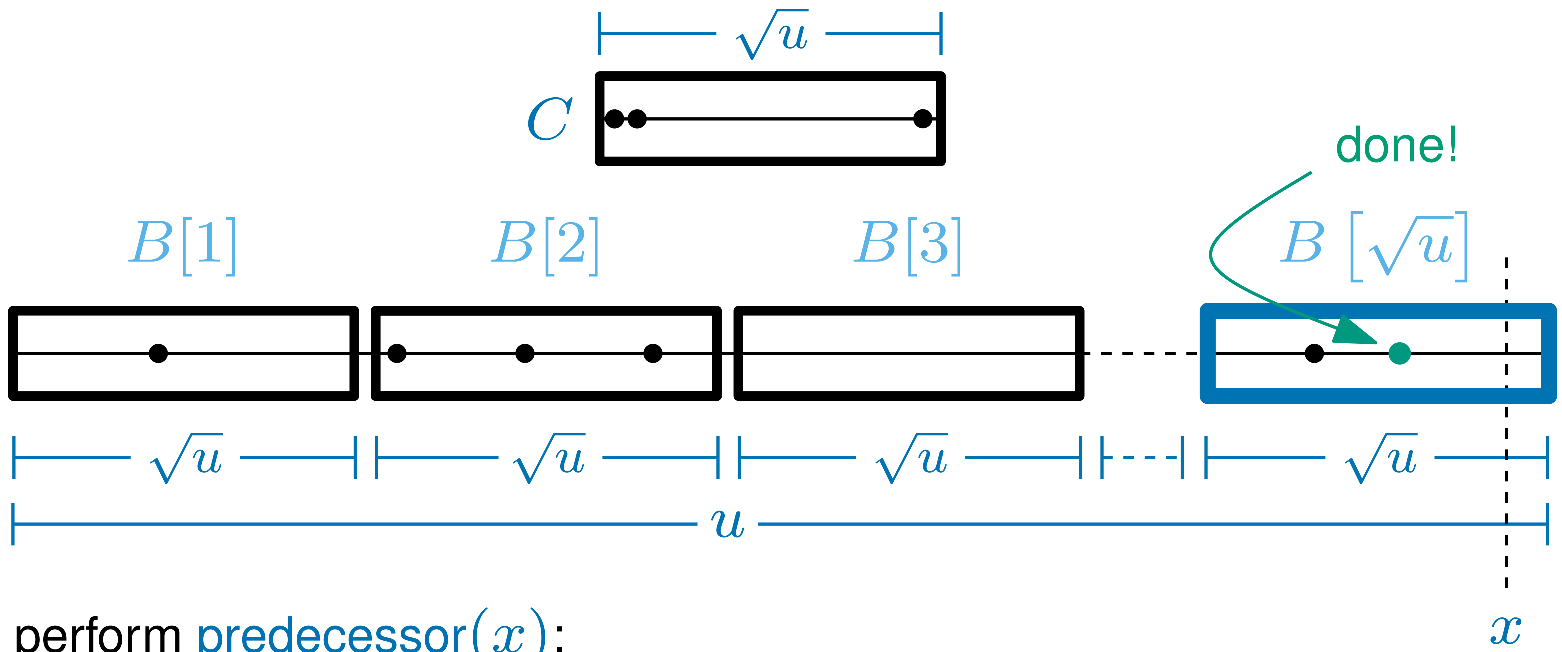
Compute $j =$ predecessor(i) in C

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

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

Attempt 3: Recursion

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



To perform 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 $B[i]$:

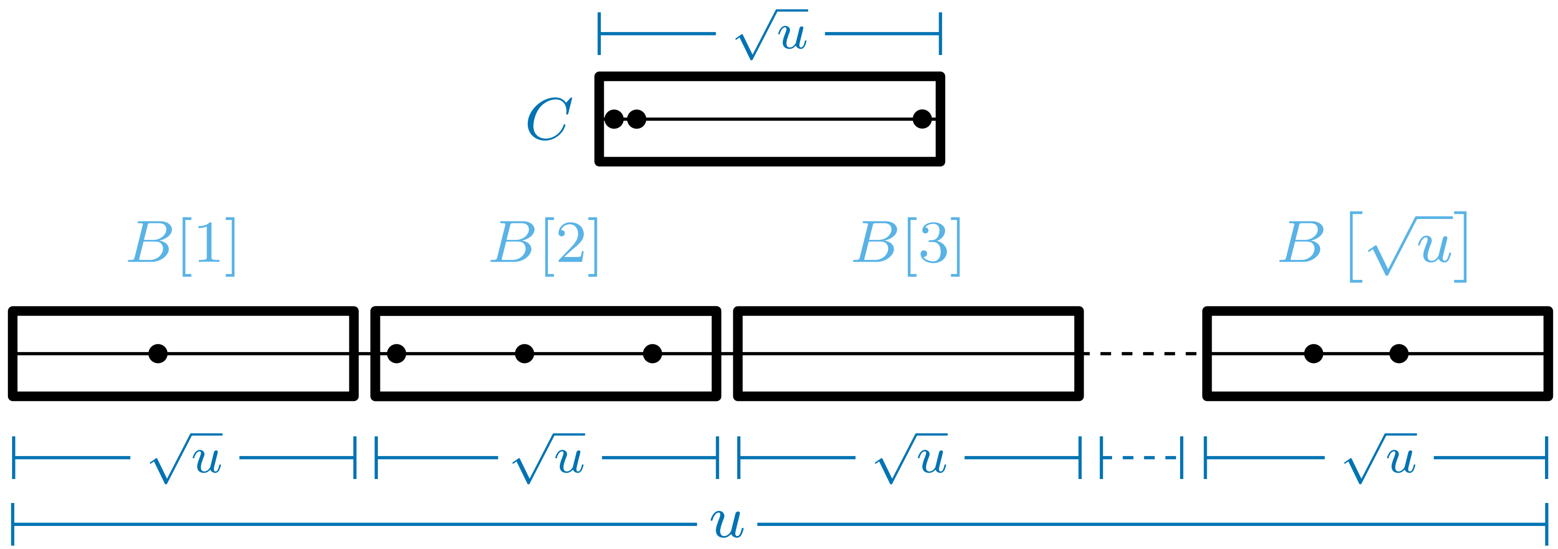
Compute $j =$ predecessor(i) in C

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

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

Attempt 3: Recursion

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



To perform 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 $B[i]$:

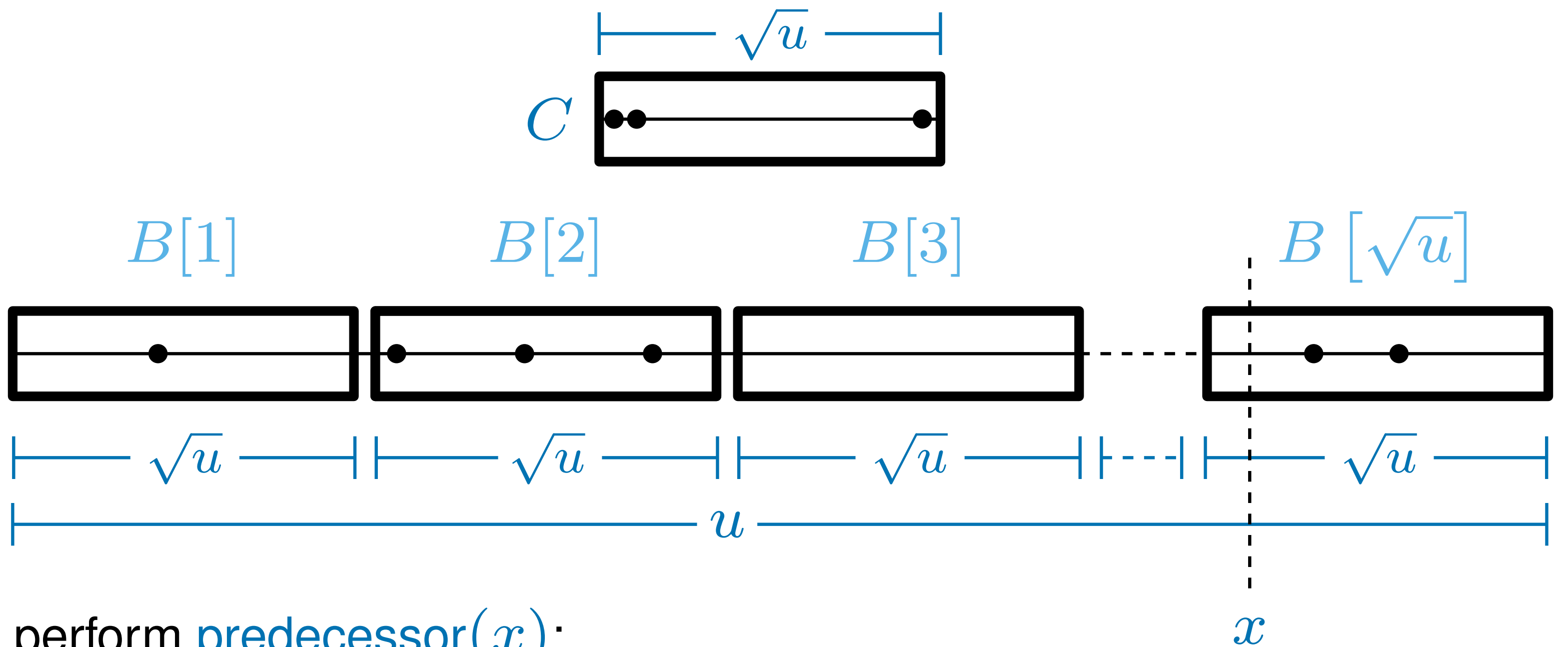
Compute $j =$ predecessor(i) in C

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

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

Attempt 3: Recursion

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



To perform 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 $B[i]$:

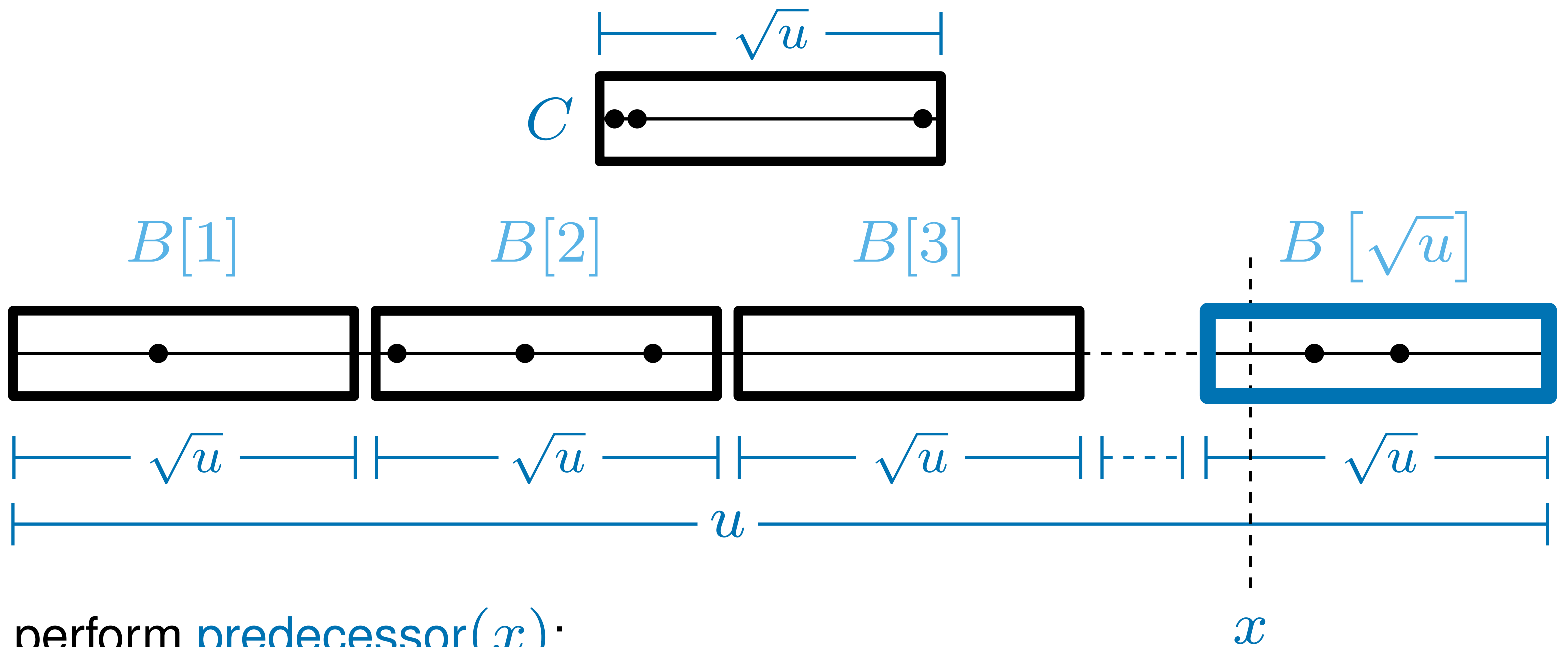
Compute $j =$ predecessor(i) in C

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

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

Attempt 3: Recursion

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



To perform 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 $B[i]$:

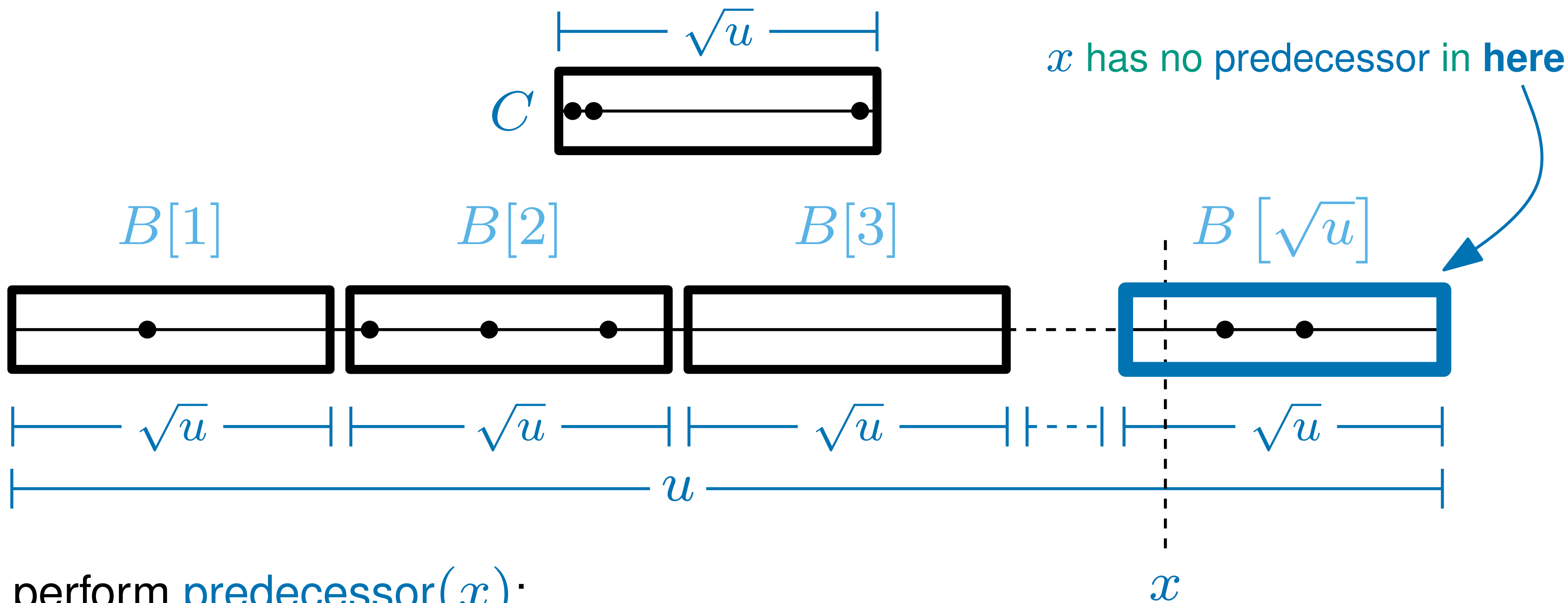
Compute $j =$ predecessor(i) in C

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

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

Attempt 3: Recursion

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



To perform 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 $B[i]$:

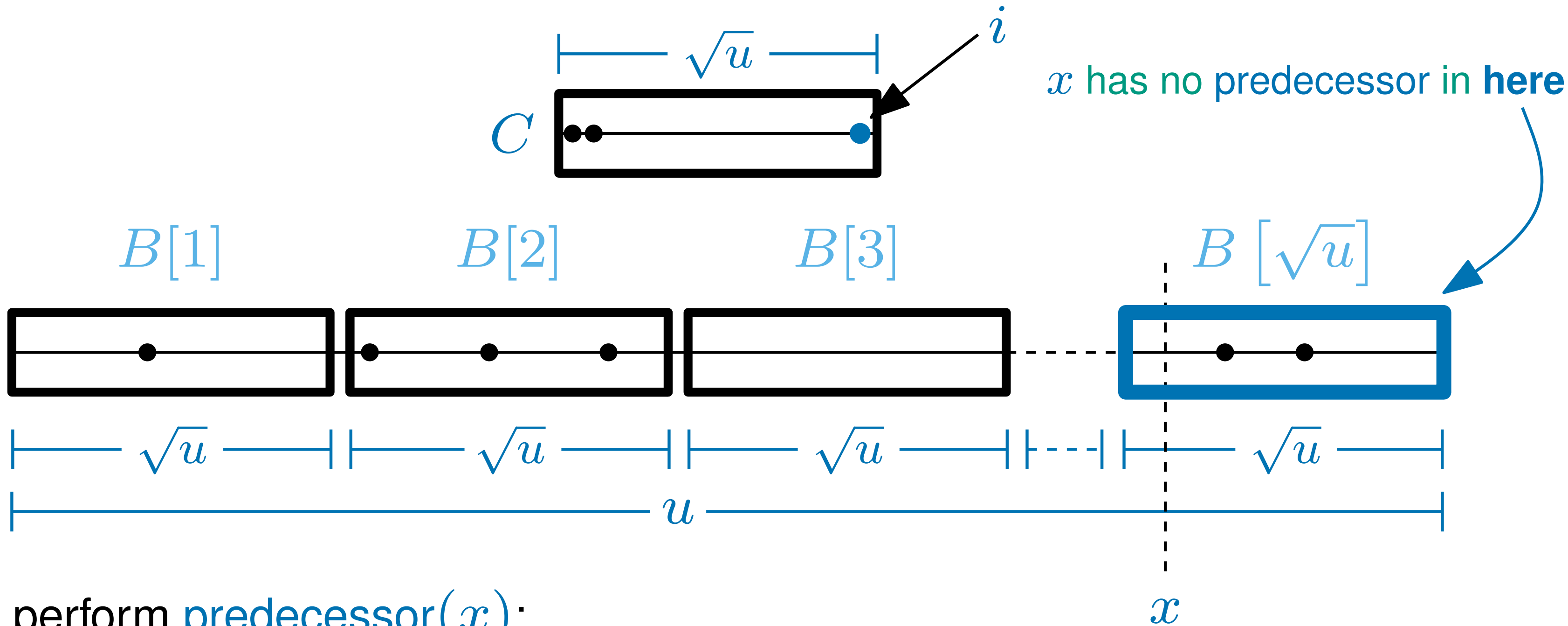
Compute $j =$ predecessor(i) in C

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

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

Attempt 3: Recursion

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



To perform 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 $B[i]$:

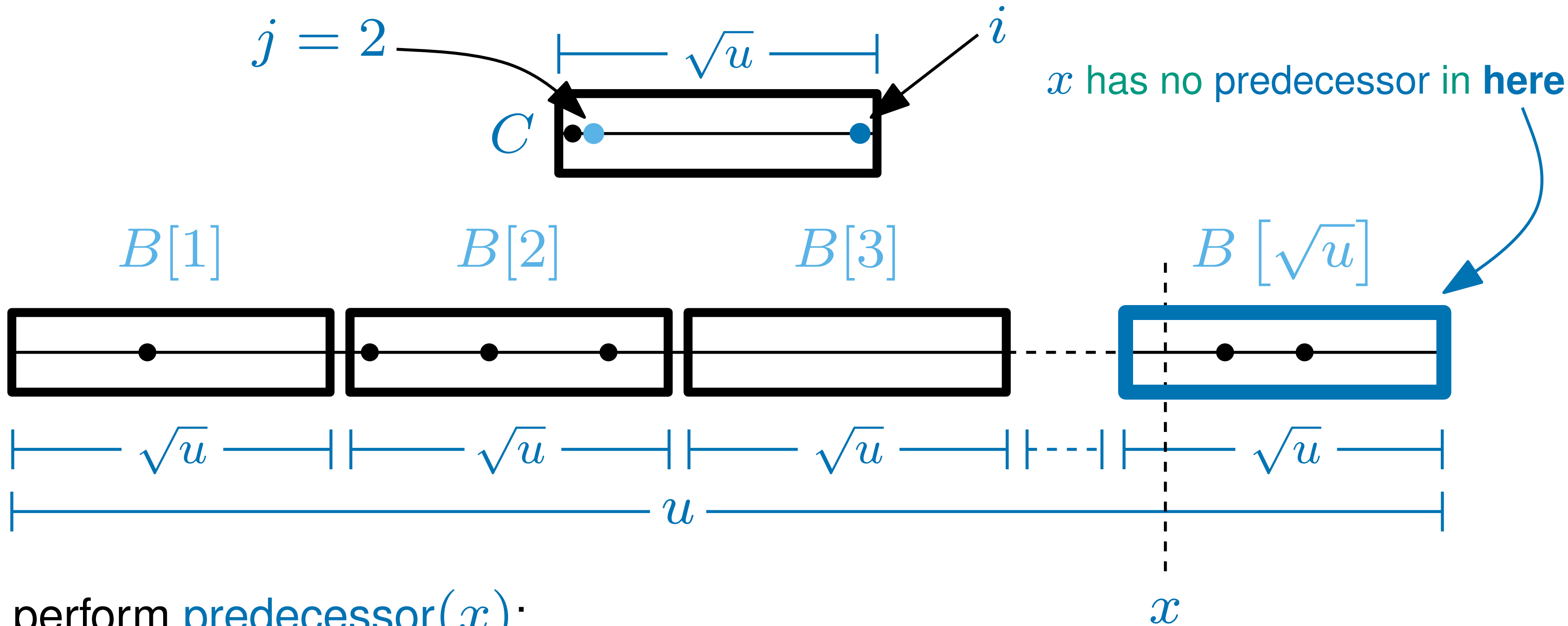
Compute $j =$ predecessor(i) in C

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

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

Attempt 3: Recursion

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



To perform 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 $B[i]$:

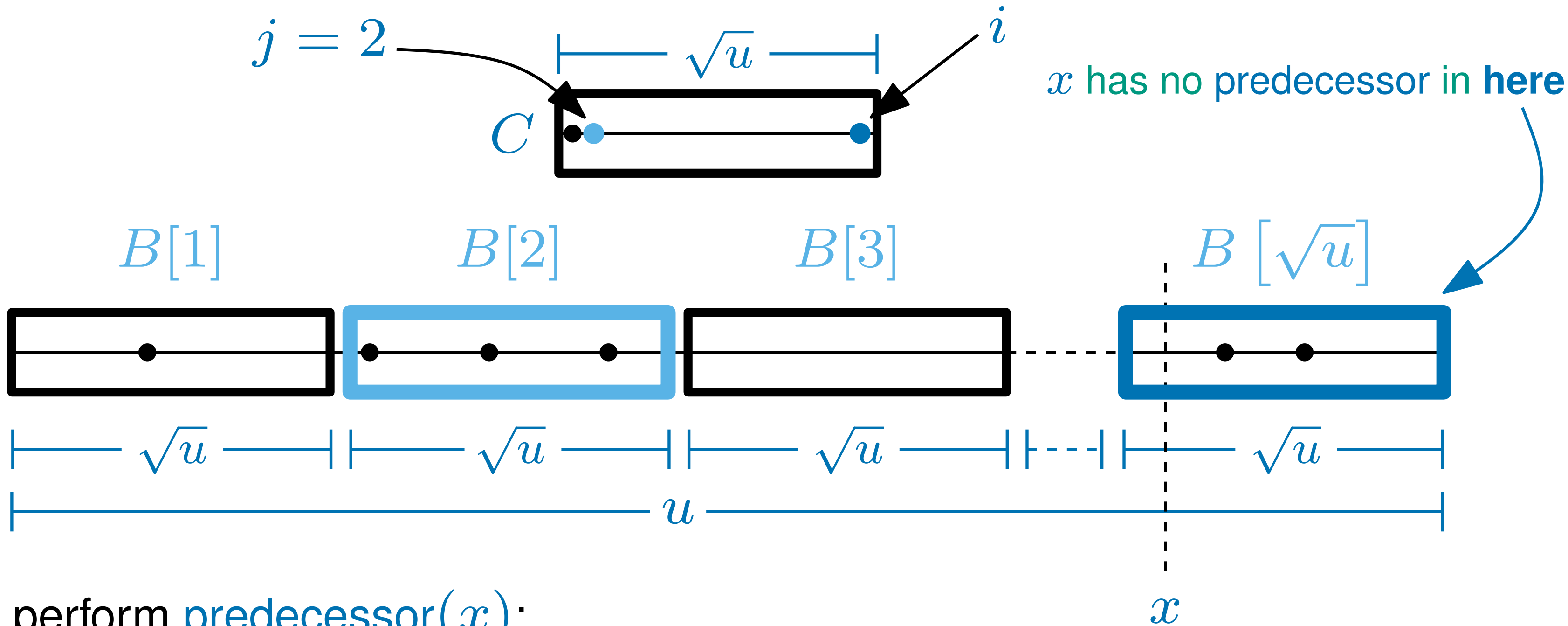
Compute $j =$ predecessor(i) in C

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

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

Attempt 3: Recursion

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



To perform 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 $B[i]$:

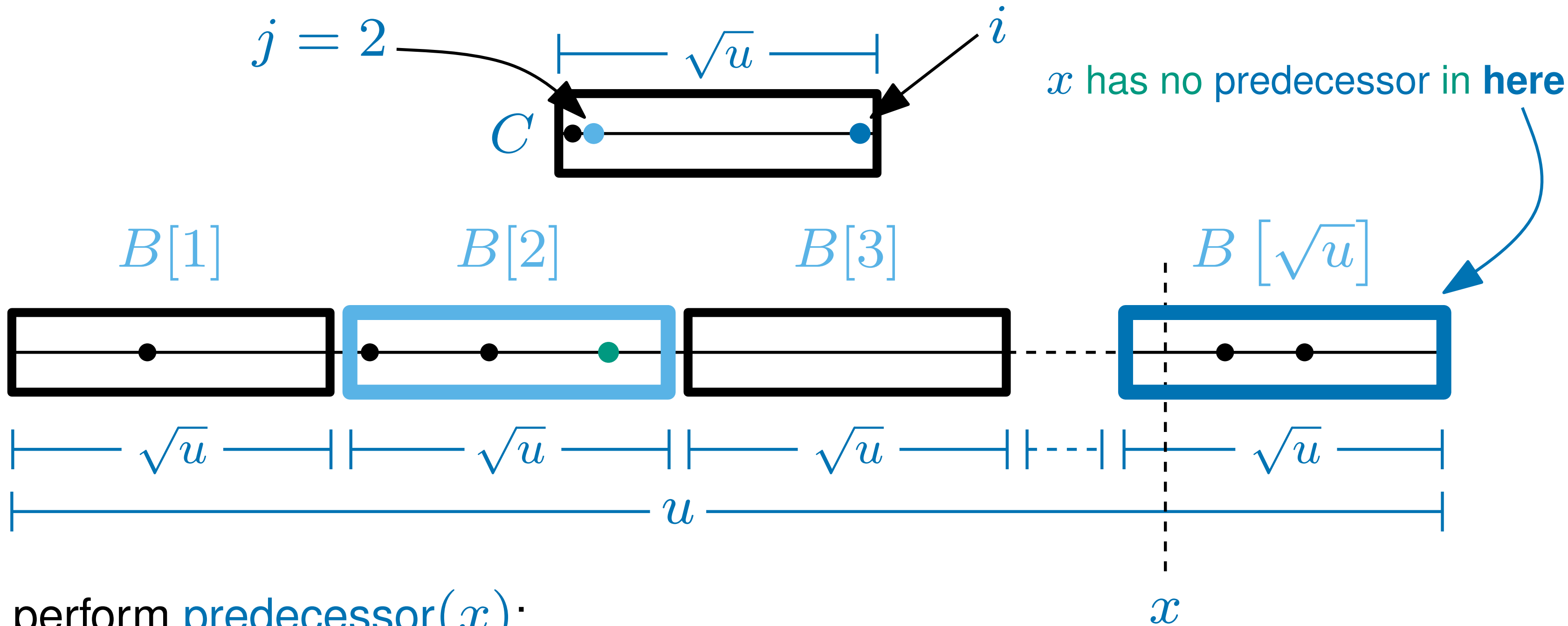
Compute $j =$ predecessor(i) in C

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

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

Attempt 3: Recursion

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



To perform 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 $B[i]$:

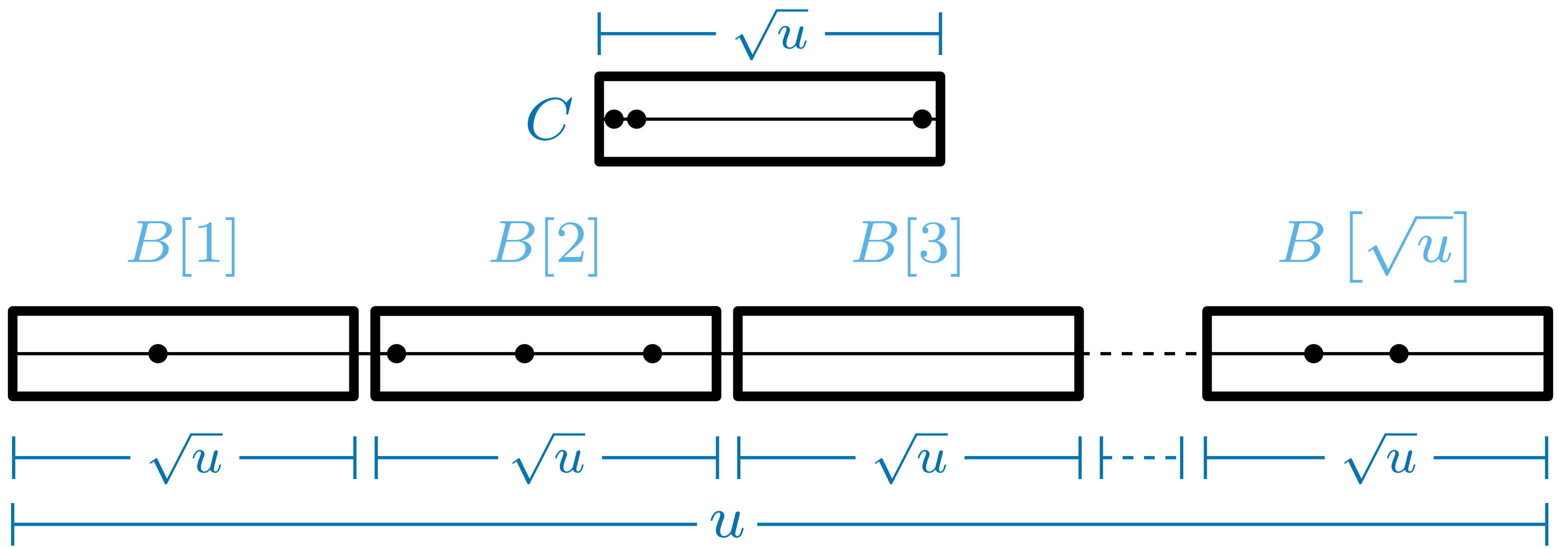
Compute $j =$ predecessor(i) in C

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

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

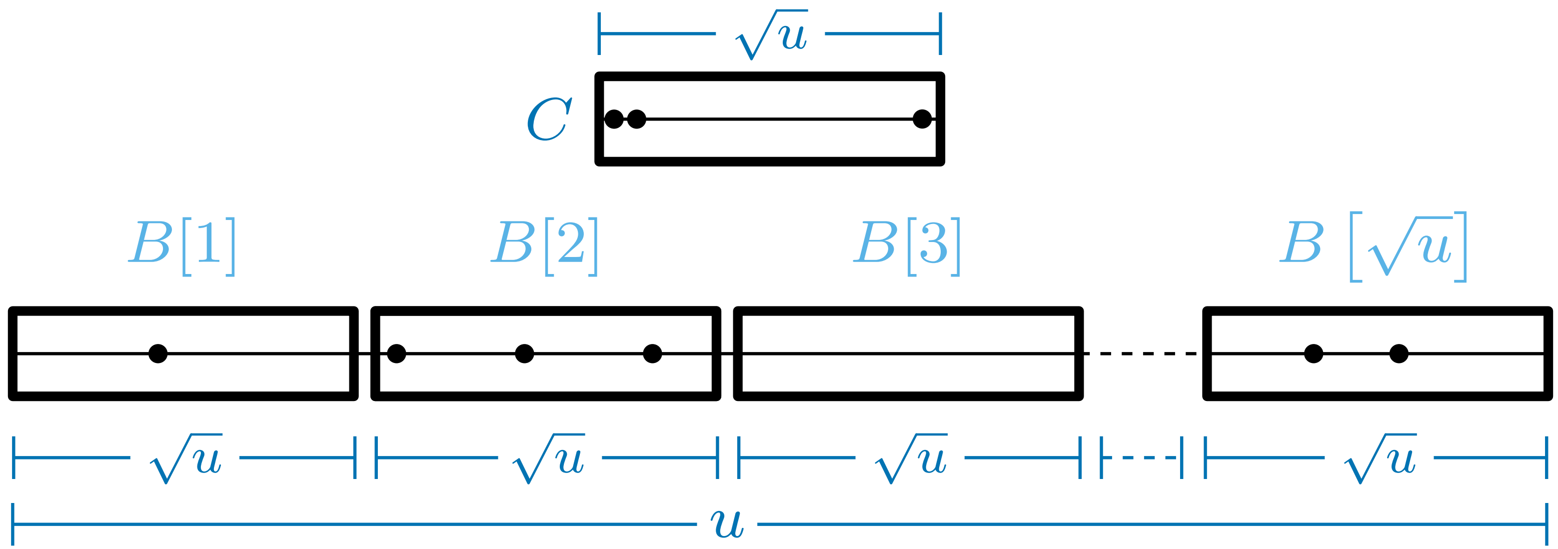
Attempt 3: Recursion

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



Attempt 3: Recursion

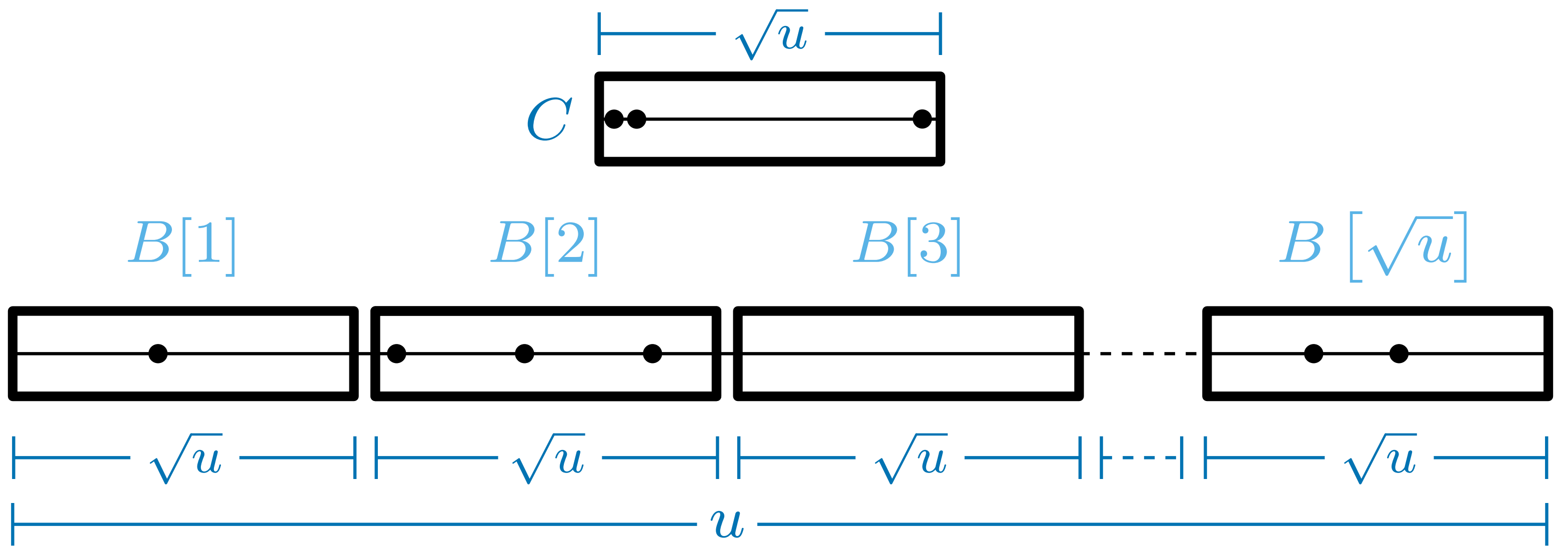
Split the universe U into $\sqrt{u}$ *blocks* each associated with $\sqrt{u}$ elements



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

Attempt 3: Recursion

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

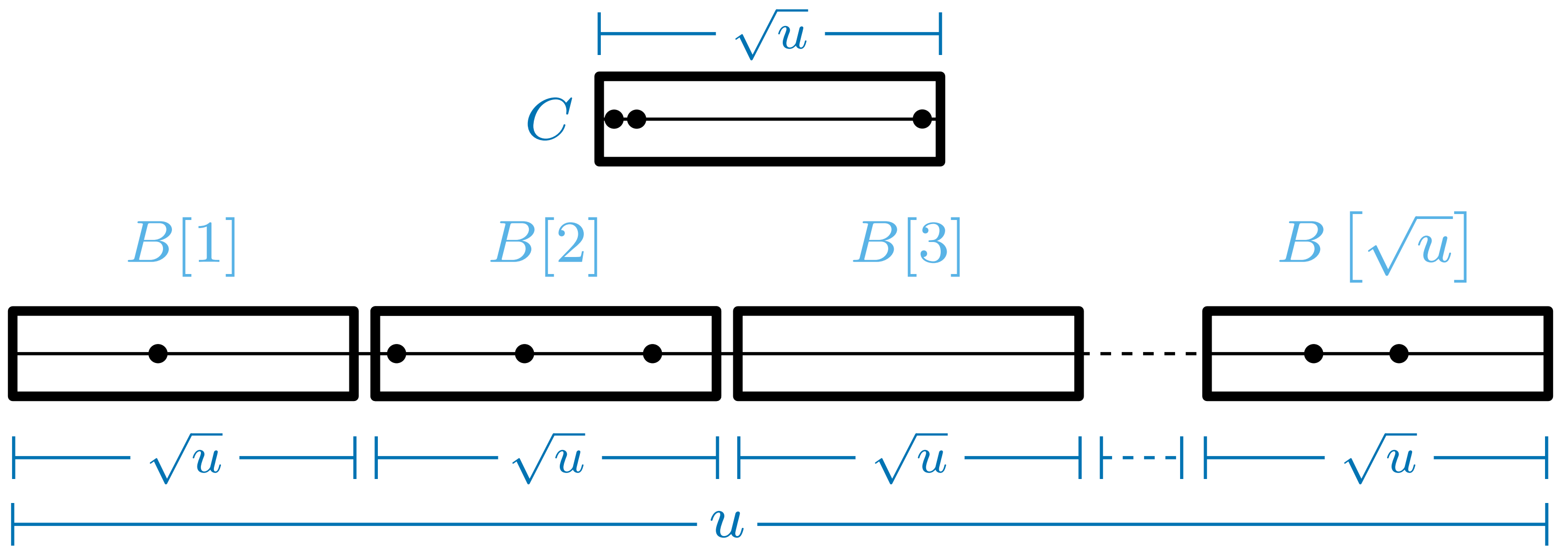


The operations lookup, delete and successor can all also be defined in a similar, recursive manner

How efficient are the operations?

Attempt 3: Recursion

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



To perform add(x):

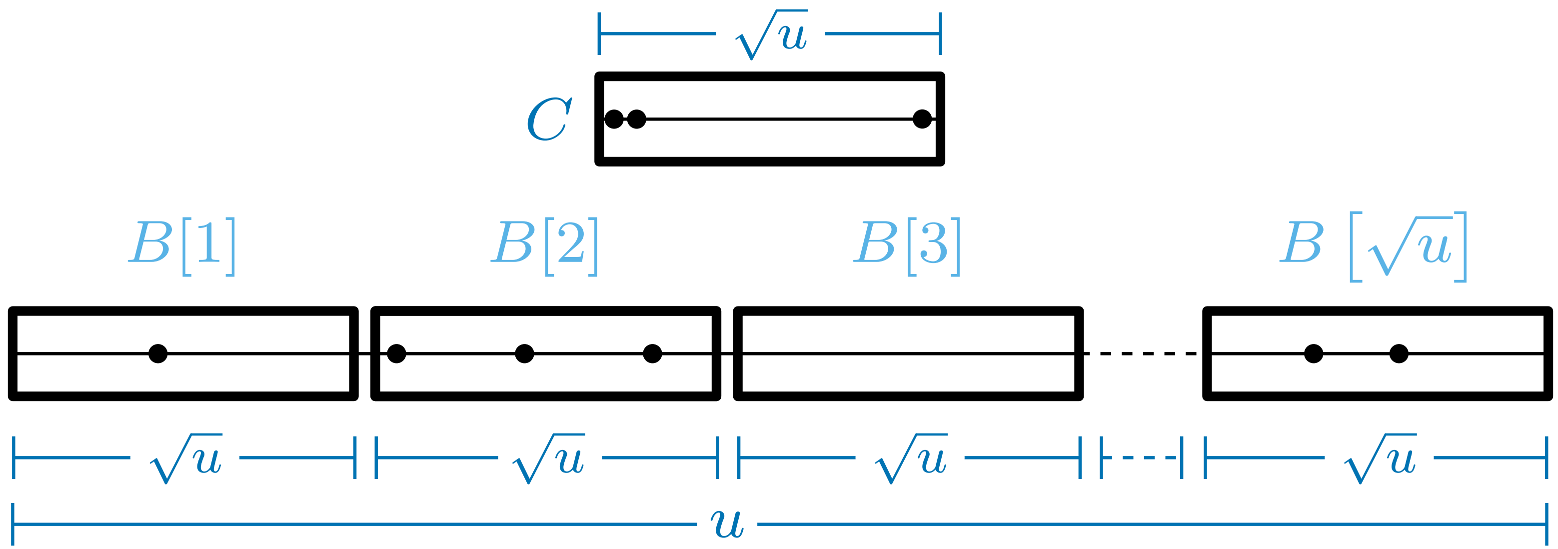
Step 1 Determine which $B[i]$ the element x belongs in
(this takes $O(1)$ time with a little bit twiddling)

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 3: Recursion

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



add makes up to two recursive calls

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)

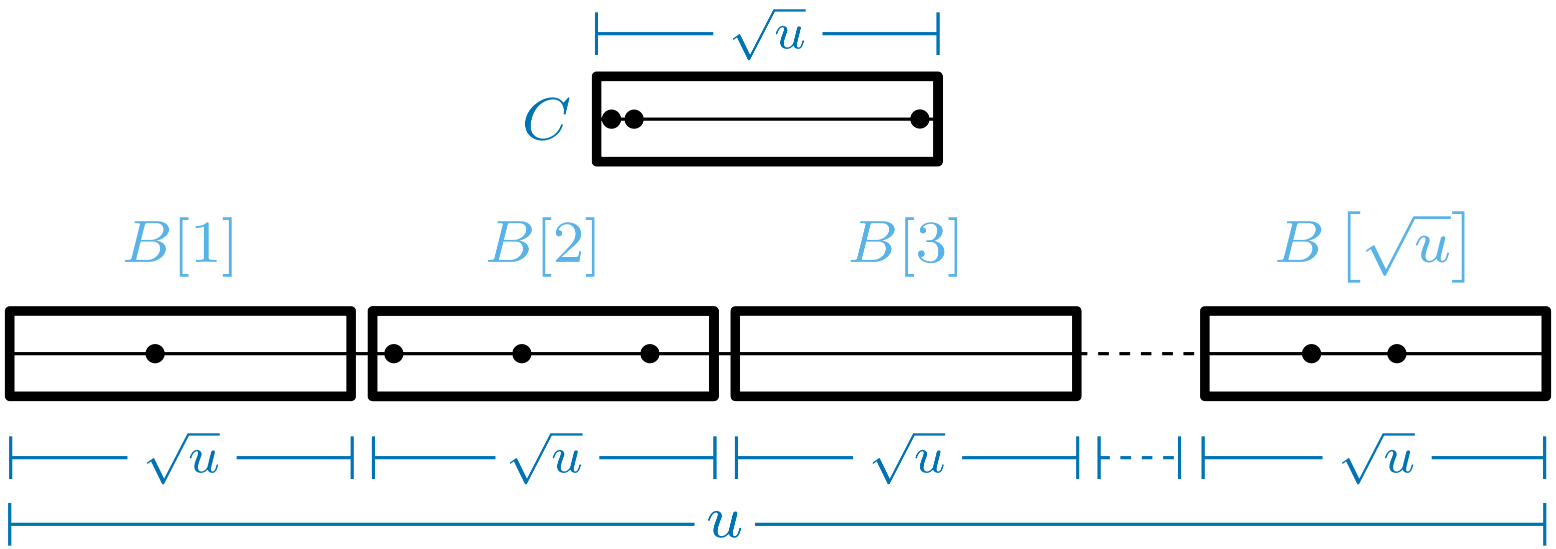
Step 2 If $B[i]$ is empty, add i to C

Step 3 add x to $B[i]$

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

Attempt 3: Recursion

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



add makes up to two recursive calls

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)

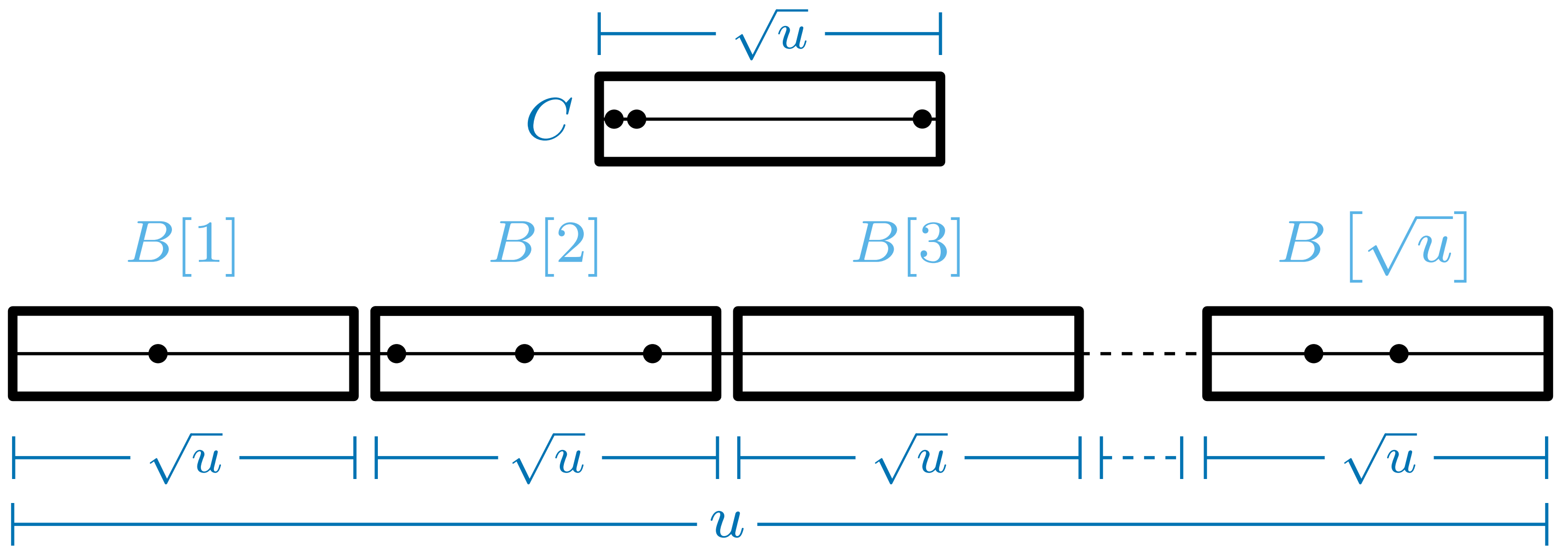
Step 2 If $B[i]$ is empty, add i to C

Step 3 add x to $B[i]$

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

Attempt 3: Recursion

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



To perform predecessor(x):

predecessor makes up to **three** recursive calls!

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 $B[i]$:

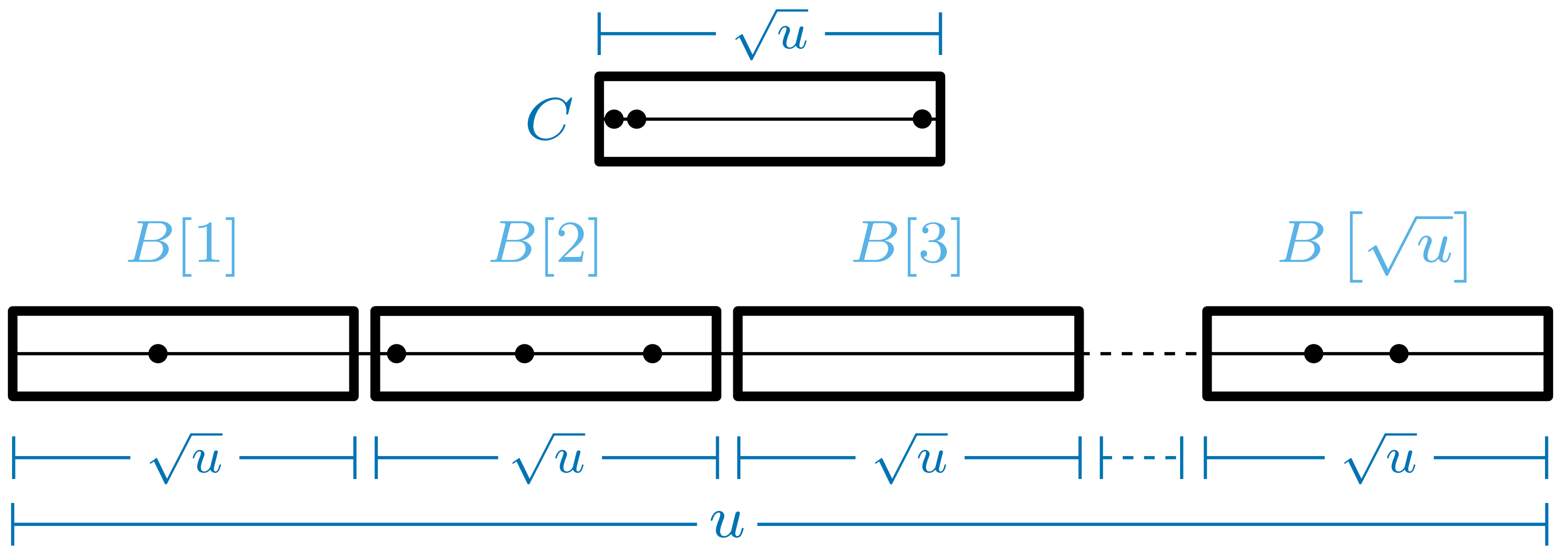
Compute $j =$ predecessor(i) in C

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

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

Attempt 3: Recursion

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



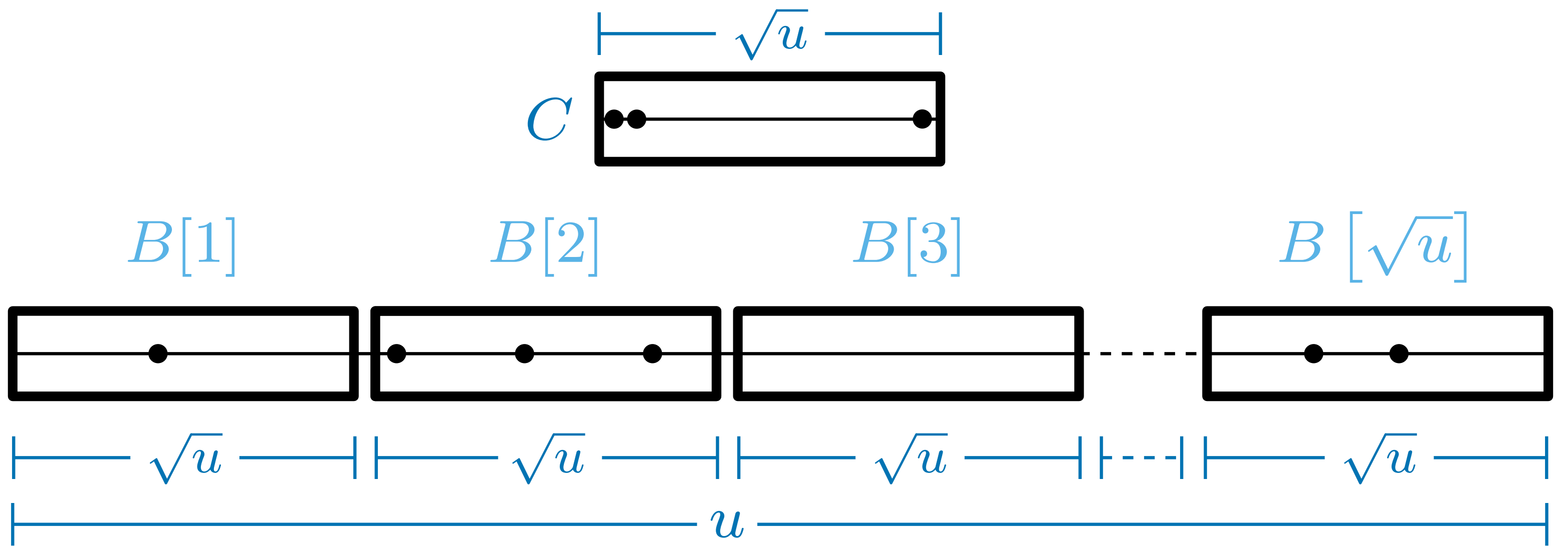
The operations lookup, delete and successor can all also be defined in a similar, recursive manner

How efficient are the operations?

University of BRISTOL

Attempt 3: Recursion

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



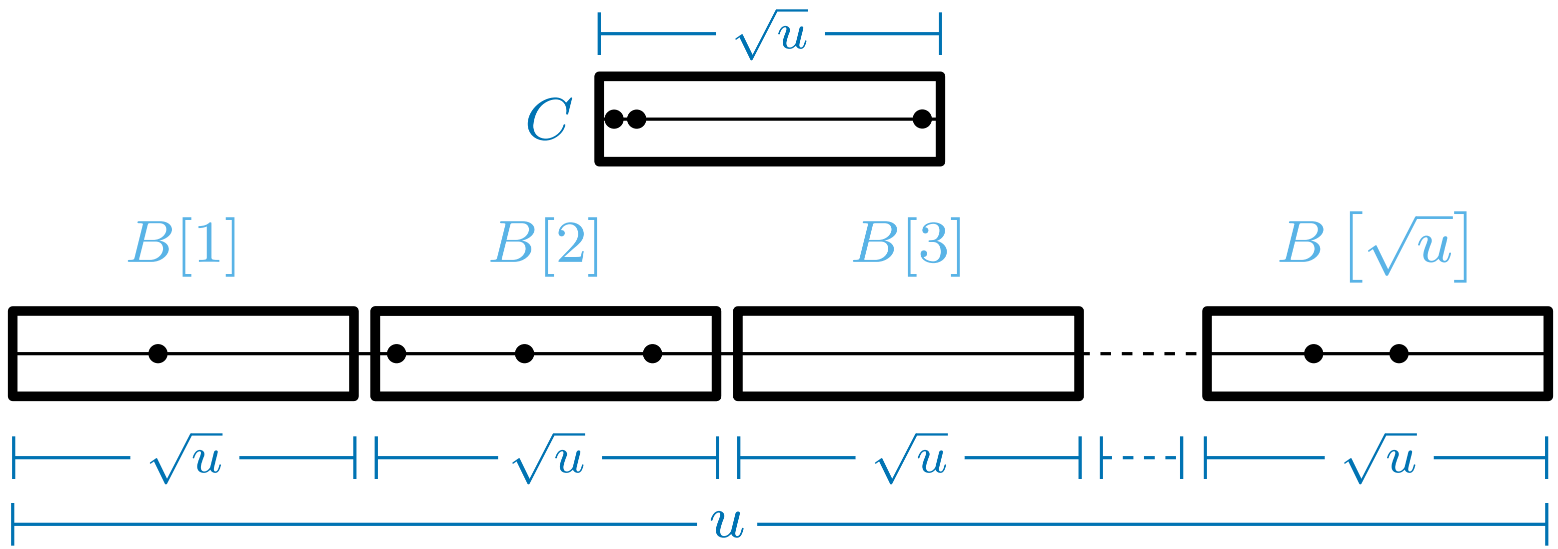
The operations lookup, delete and successor can all also be defined in a similar, recursive manner

How efficient are the operations?

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

Attempt 3: 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

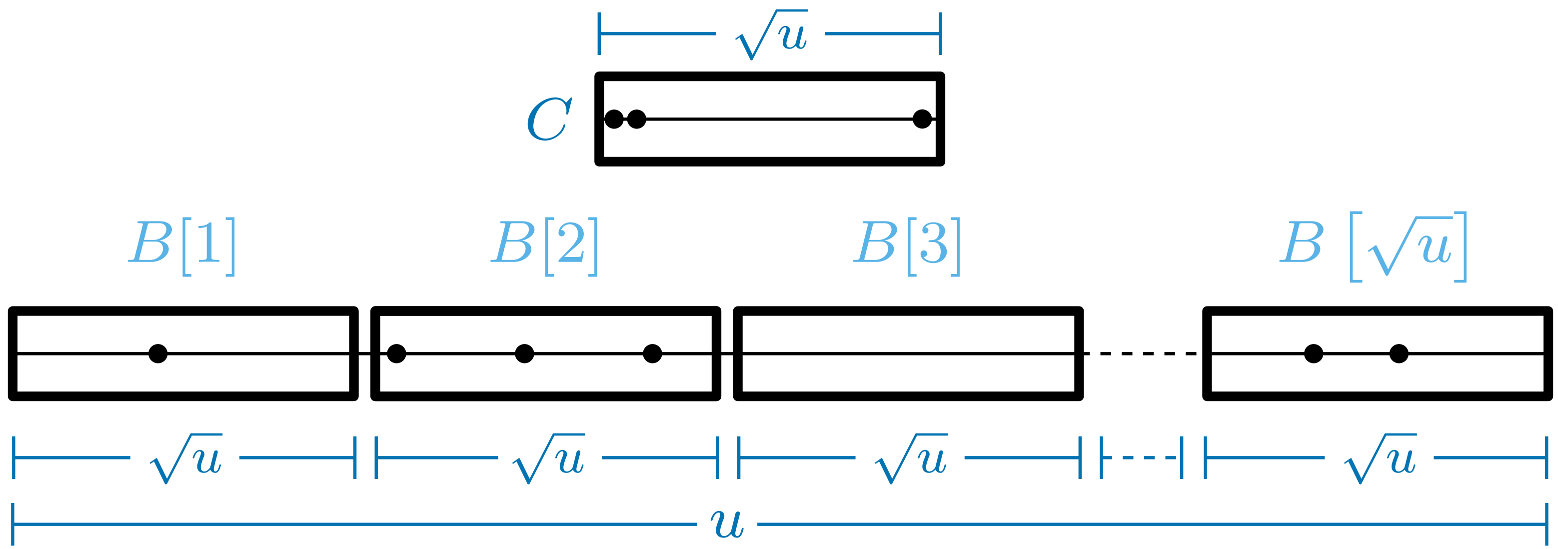
How efficient are the operations?

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 3: 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

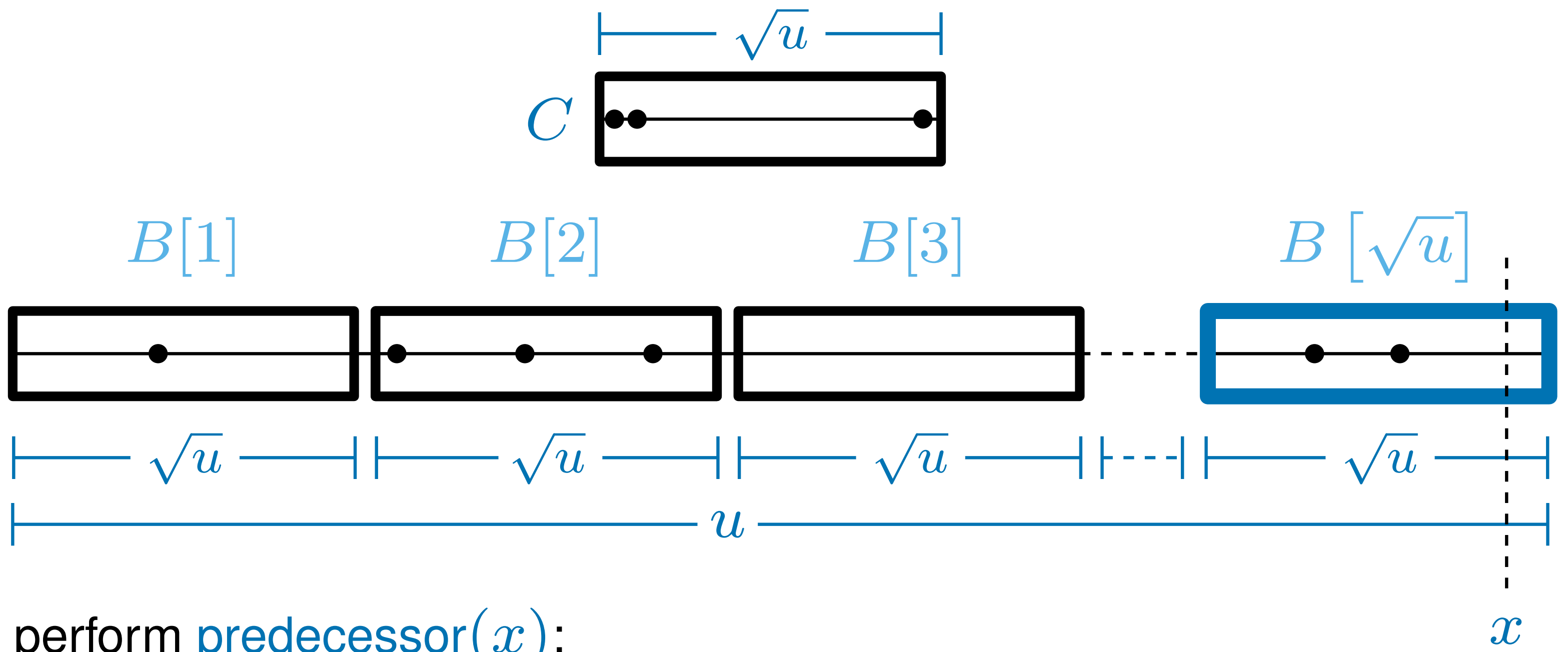
How efficient are the operations?

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 predecessor(x):

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

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

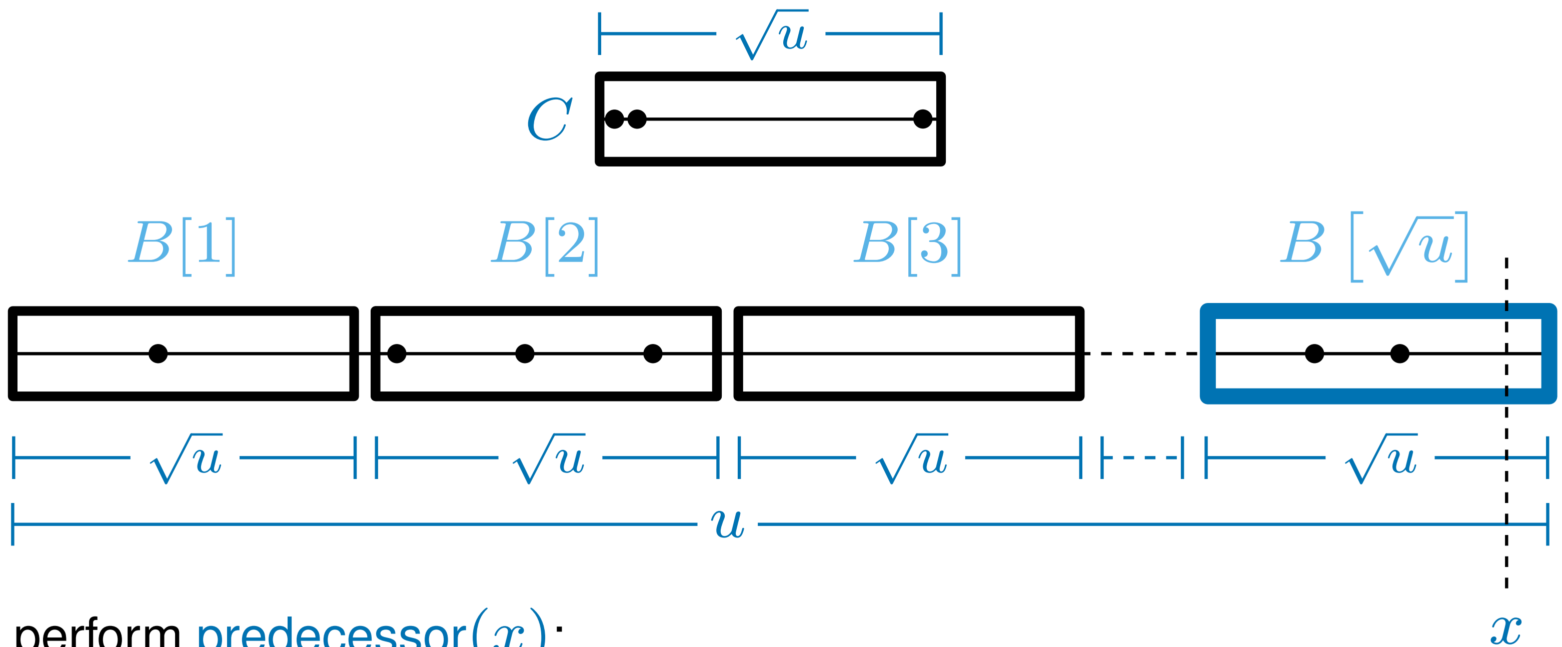
Step 3 If x has no predecessor in $B[i]$:

Compute $j = $ 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 predecessor(x):

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

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

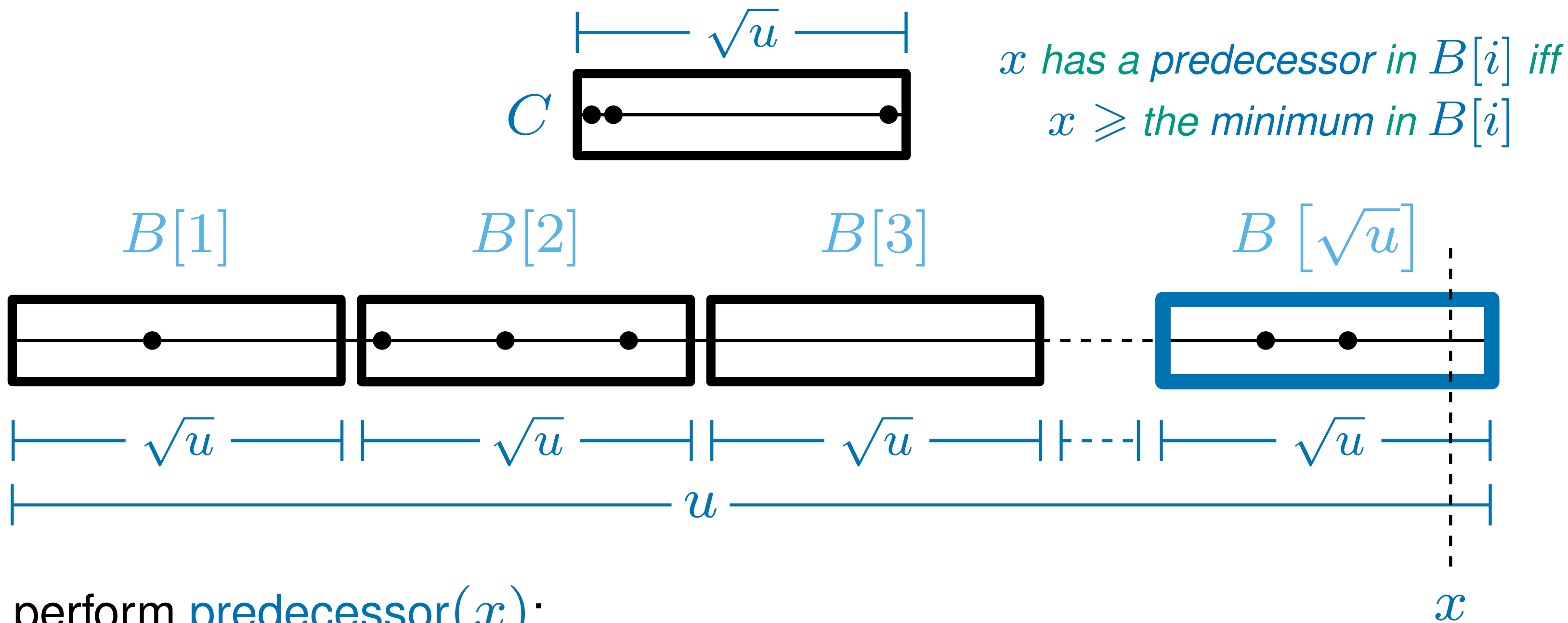
Step 3 If x has no predecessor in $B[i]$:

Compute $j =$ 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 predecessor(x):

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

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

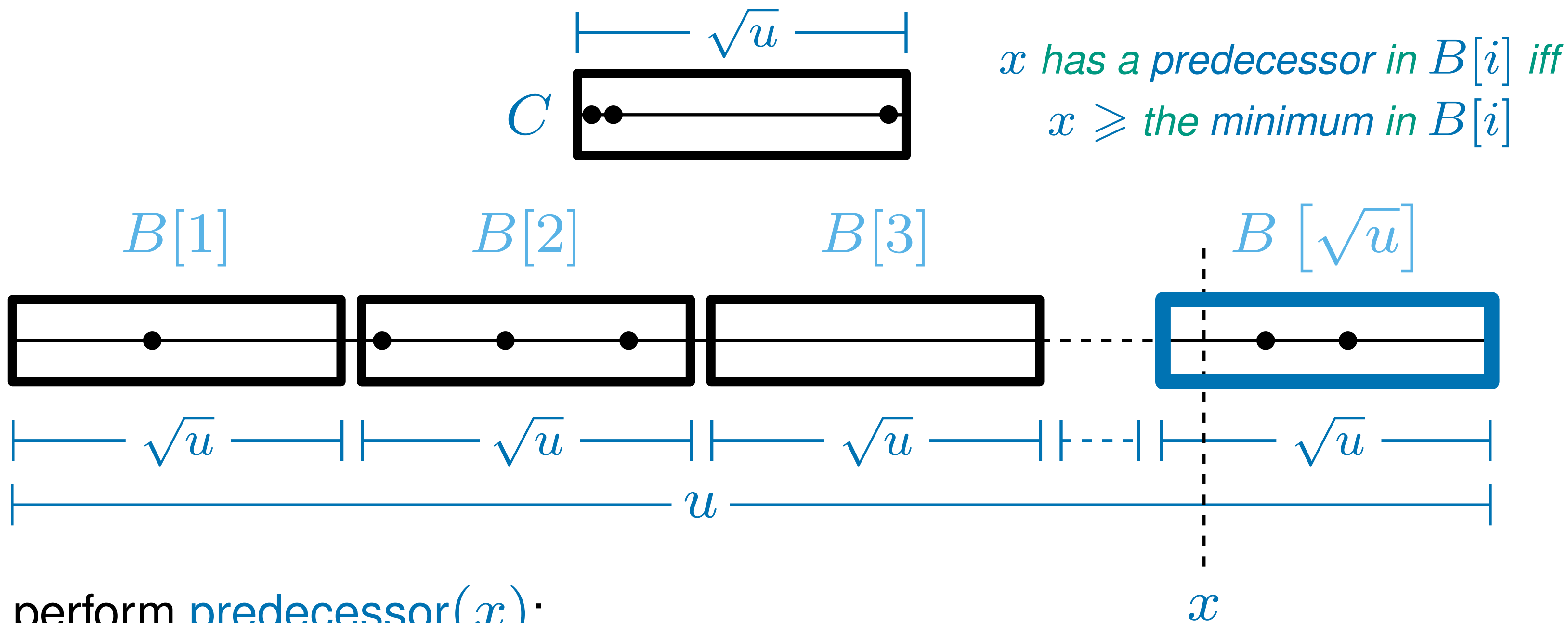
Step 3 If x has no predecessor in $B[i]$:

Compute $j =$ 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 predecessor(x):

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

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

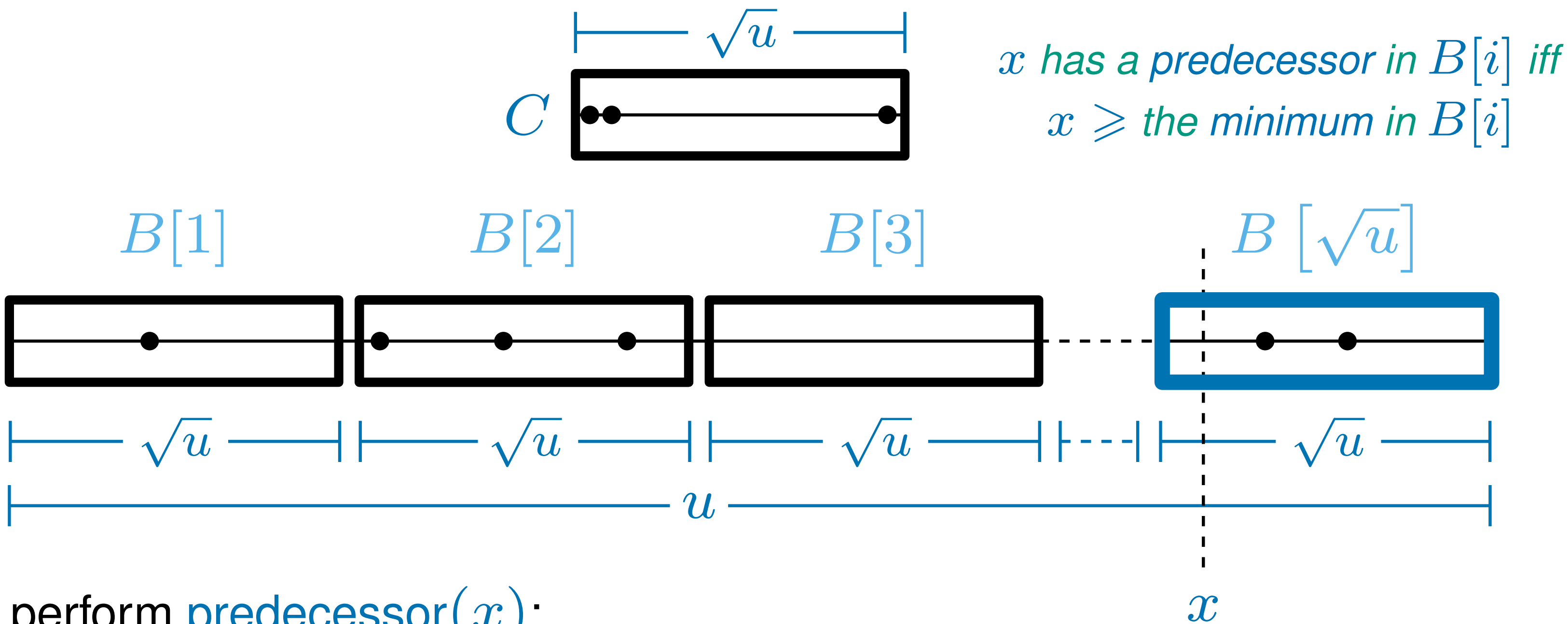
Step 3 If x has no predecessor in $B[i]$:

Compute $j =$ 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 predecessor(x):

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

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

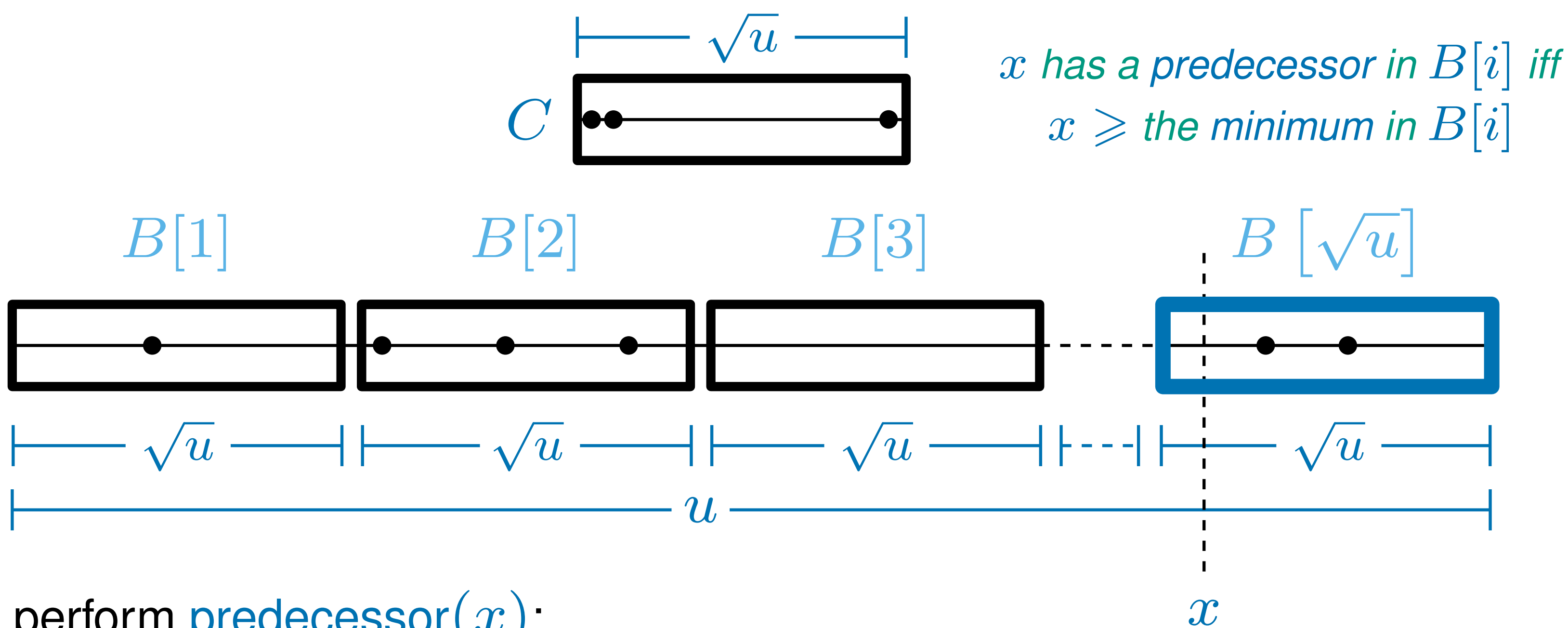
Step 3 If $x <$ the minimum in $B[i]$:

Compute $j =$ 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 predecessor(x):

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

Step 2 If $x \geqslant$ the minimum in $B[i]$:

Return the predecessor of x in $B[i]$

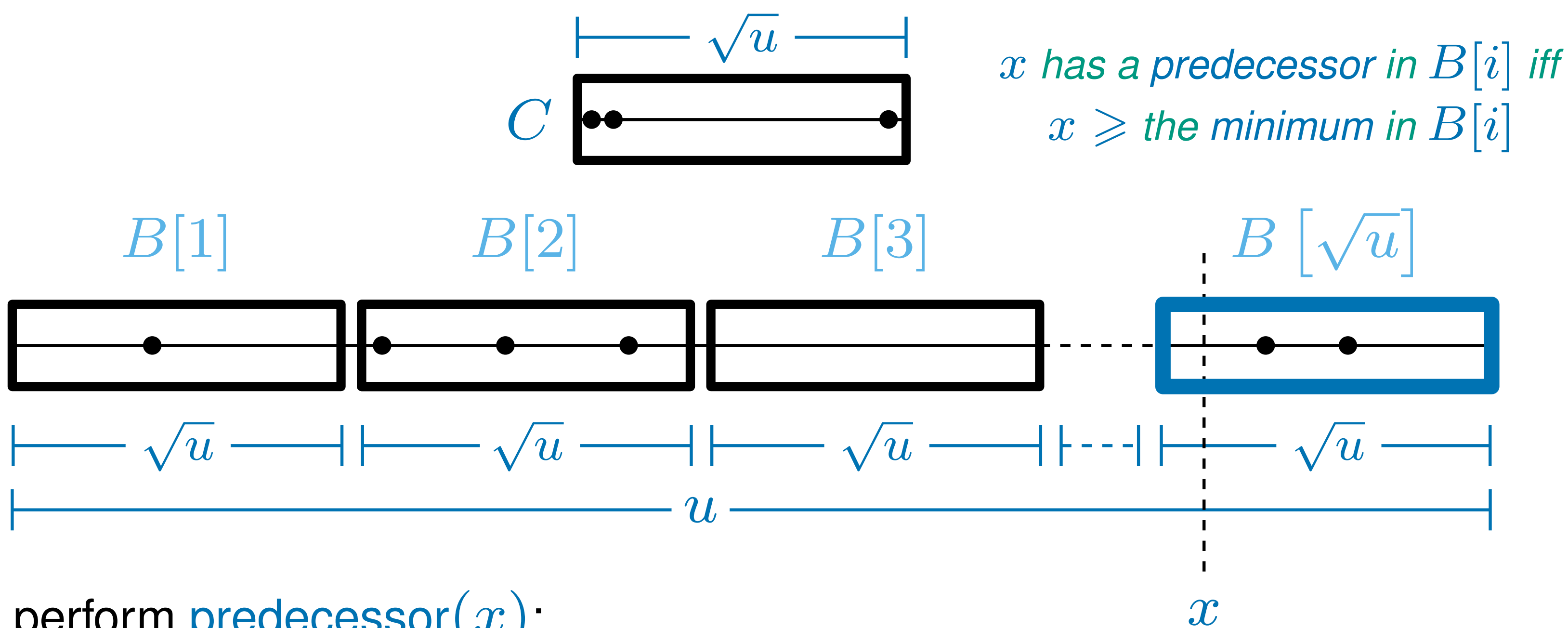
Step 3 If $x <$ the minimum in $B[i]$:

Compute $j =$ 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 predecessor(x):

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

Step 2 If $x \geqslant$ the minimum in $B[i]$:

Return the predecessor of x in $B[i]$

Step 3 If $x <$ the minimum in $B[i]$:

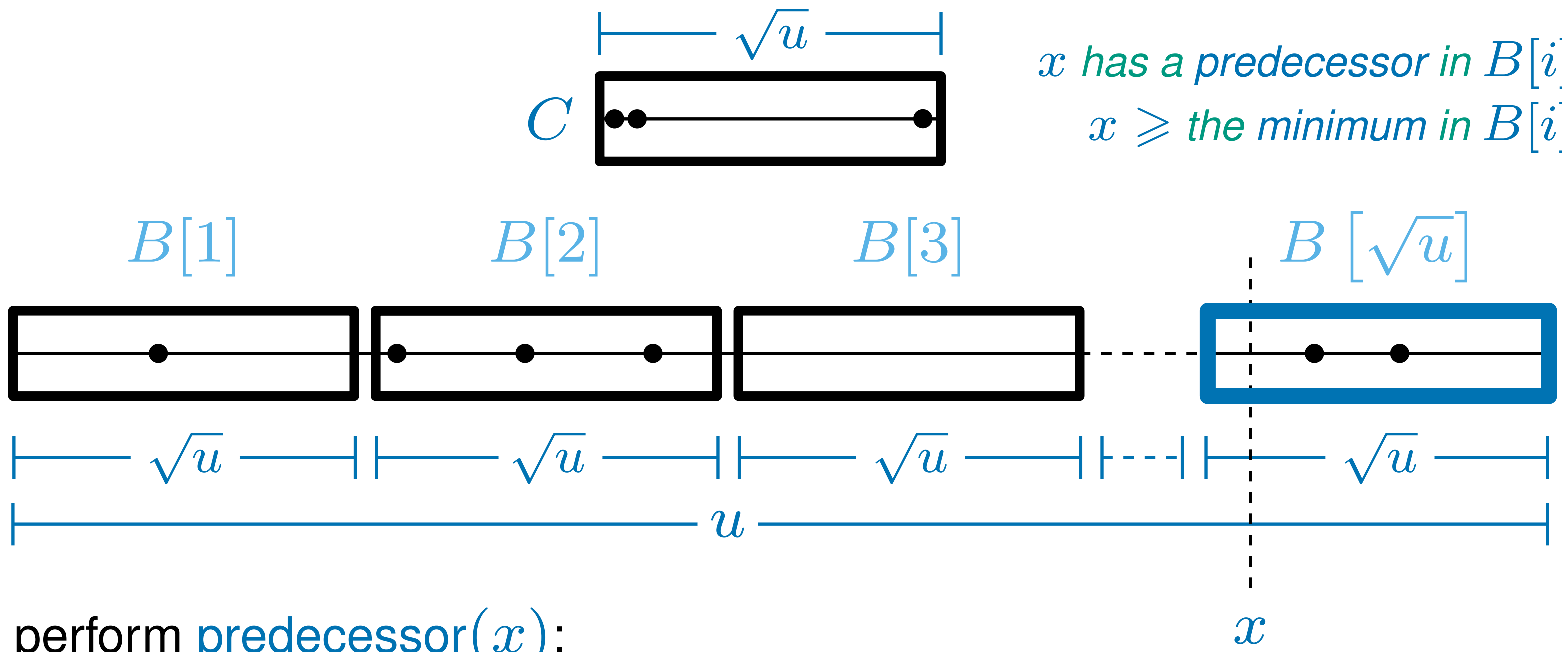
Compute $j =$ predecessor(i) in C

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 $x \geqslant$ the minimum in $B[i]$

To perform predecessor(x):

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

Step 2 If $x \geqslant$ the minimum in $B[i]$:

Return the predecessor of x in $B[i]$

Step 3 If $x <$ the minimum in $B[i]$:

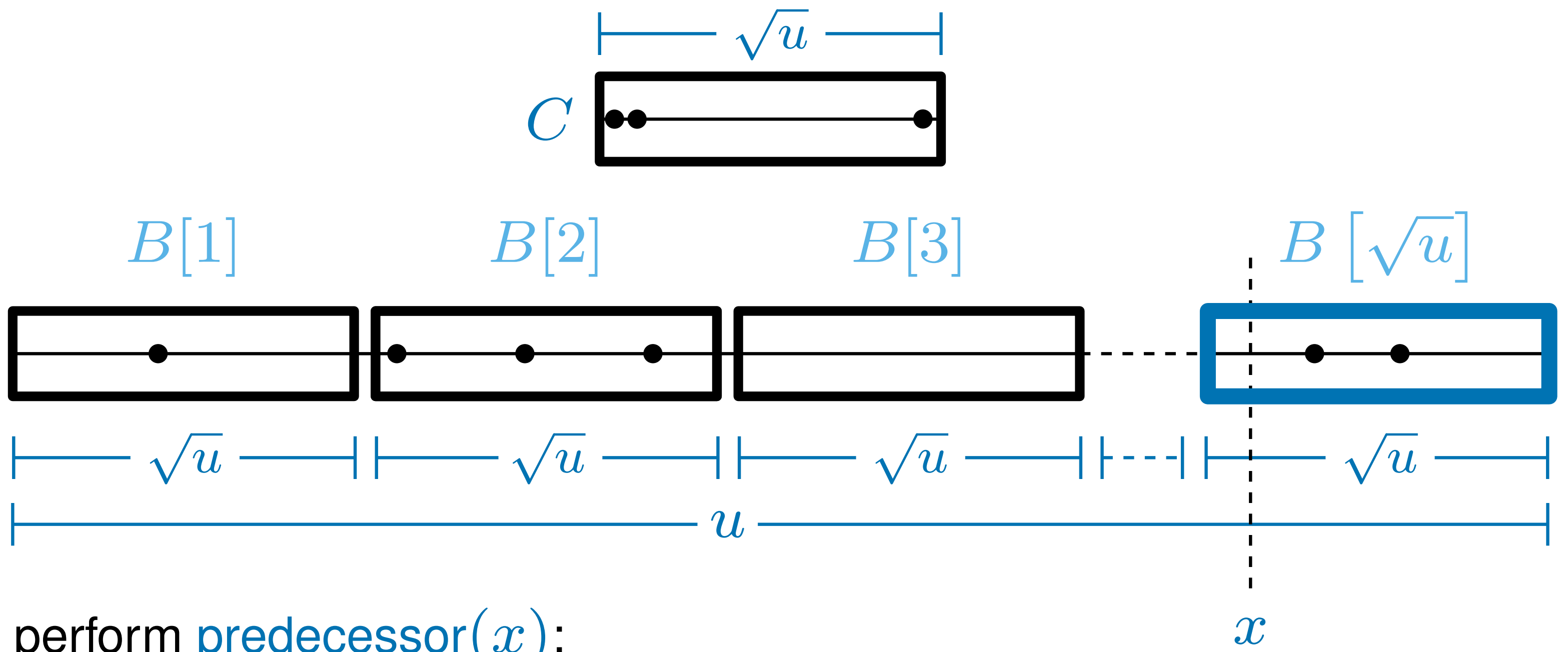
Compute $j =$ predecessor(i) in C

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

Now we make at most two recursive calls

(ignoring finding the minimum)

A closer look at predecessor



To perform predecessor(x):

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

Step 2 If $x \geqslant$ the minimum in $B[i]$:

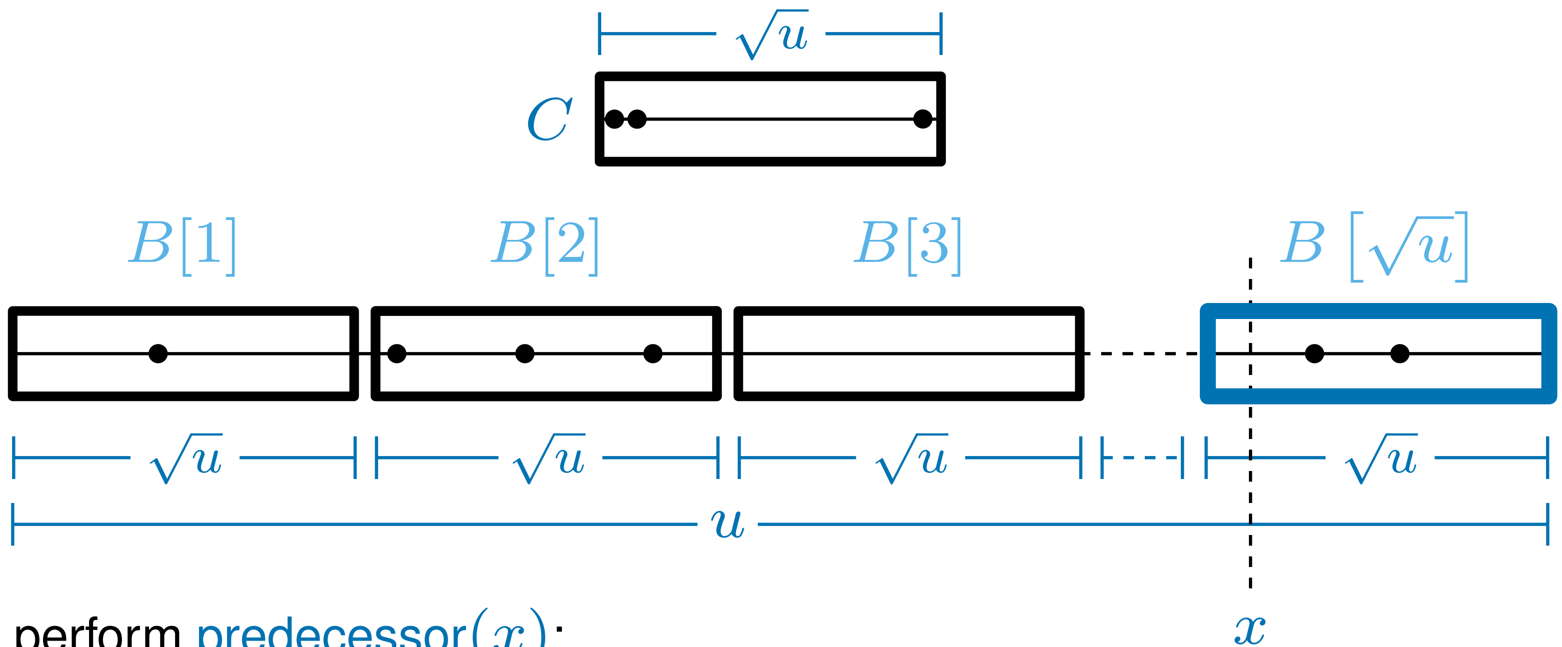
Return the predecessor of x in $B[i]$

Step 3 If $x <$ the minimum in $B[i]$:

Compute $j =$ predecessor(i) in C

Return the 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 $x \geqslant$ the minimum in $B[i]$:

Return the predecessor of x in $B[i]$

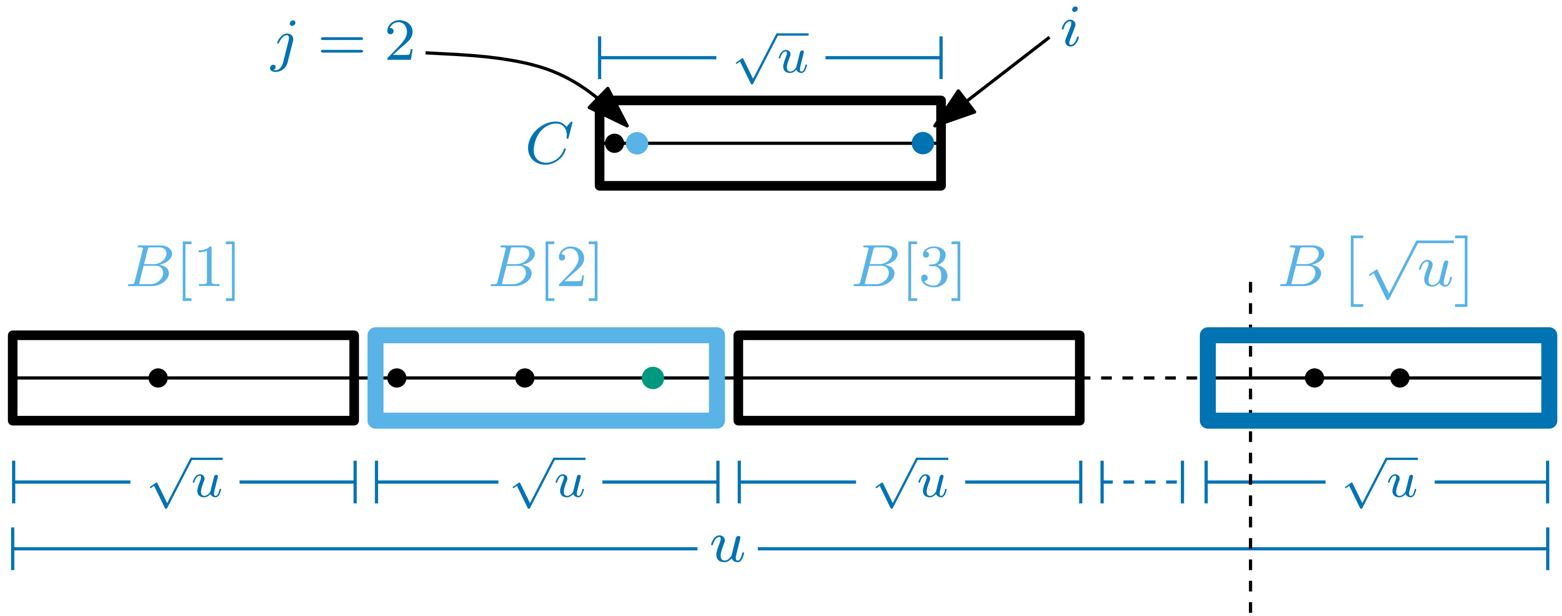
Step 3 If $x <$ the minimum in $B[i]$:

Compute $j =$ 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 predecessor(x):

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

Step 2 If $x \geqslant$ the minimum in $B[i]$:

Return the predecessor of x in $B[i]$

Step 3 If $x <$ the minimum in $B[i]$:

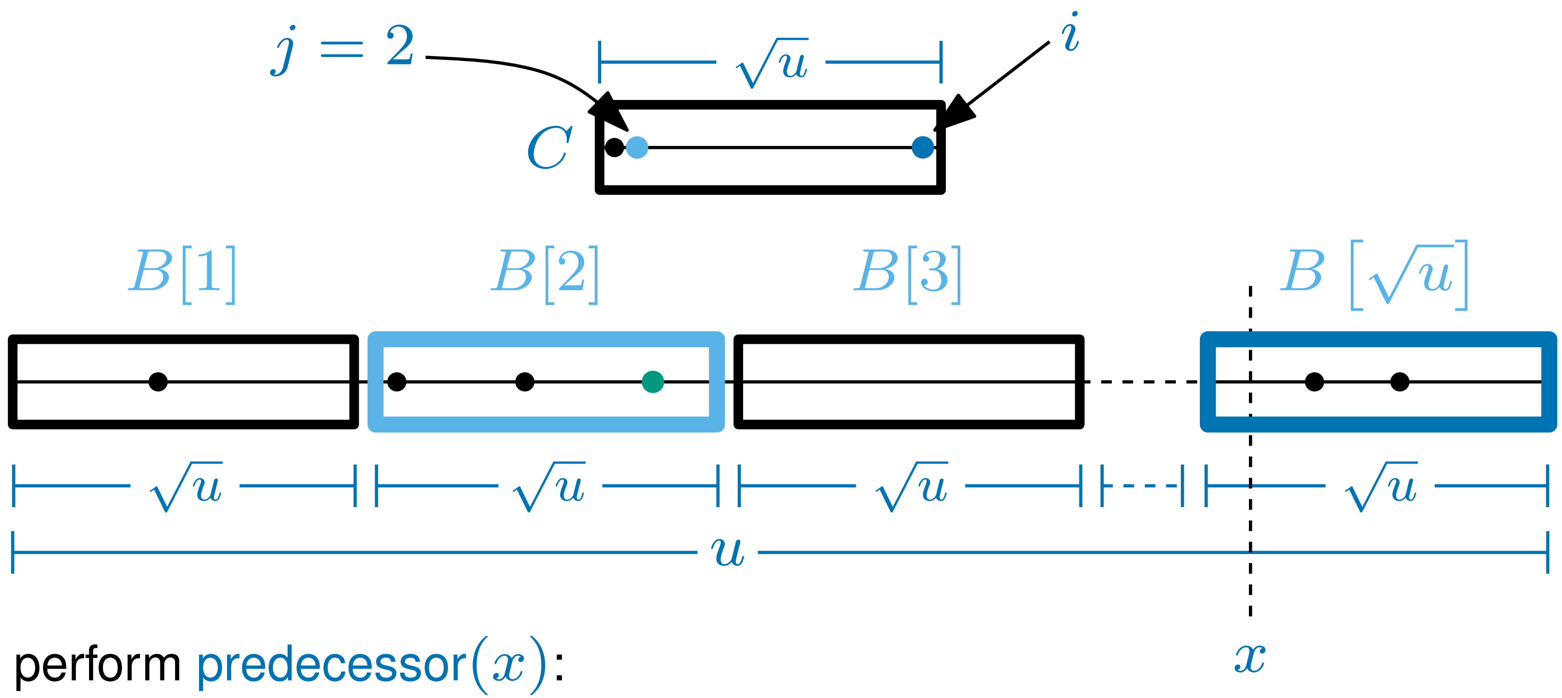
Compute $j =$ predecessor(i) in C

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

we need to get rid of one of these recursive calls

A closer look at predecessor

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



To perform predecessor(x):

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

Step 2 If $x \geqslant$ the minimum in $B[i]$:

Return the predecessor of x in $B[i]$

Step 3 If $x <$ the minimum in $B[i]$:

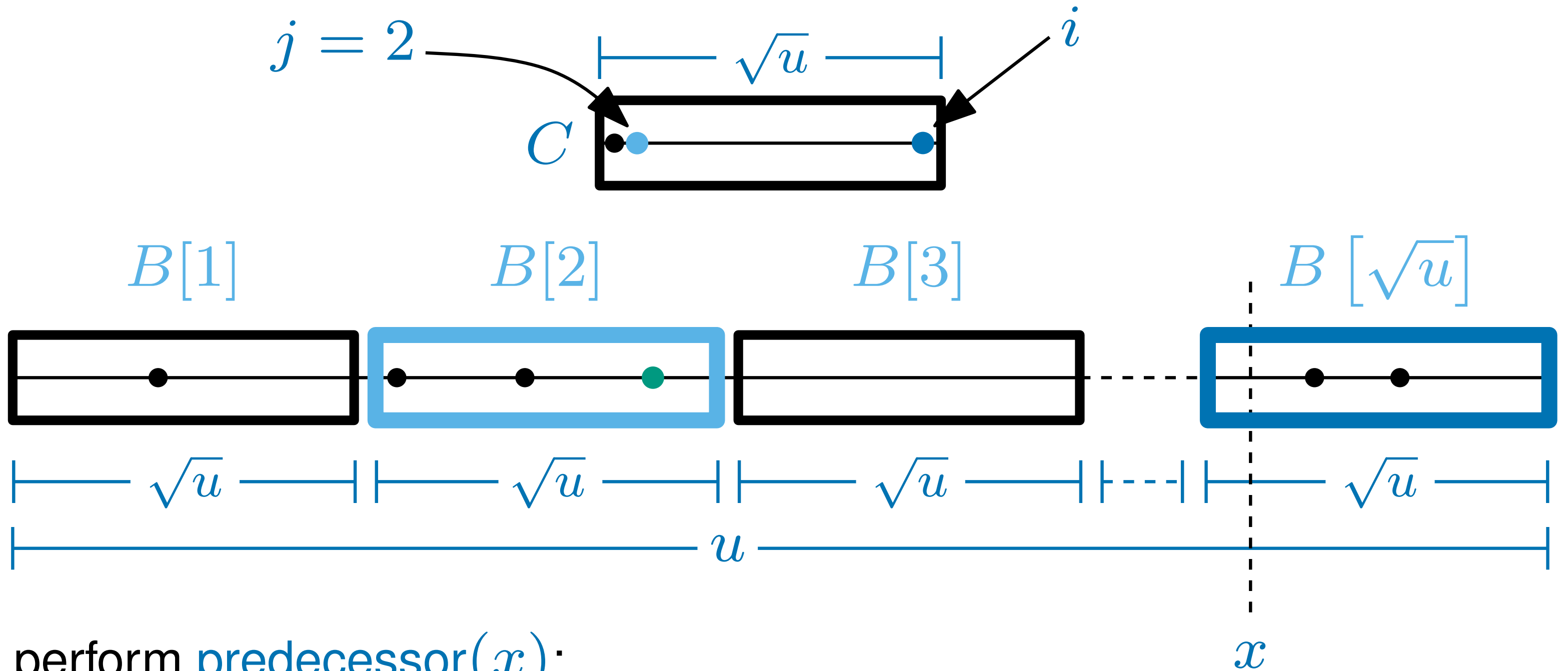
Compute $j =$ predecessor(i) in C

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

we need to get rid of one of these recursive calls

A closer look at predecessor

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



To perform predecessor(x):

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

Step 2 If $x \geqslant$ the minimum in $B[i]$:

Return the predecessor of x in $B[i]$

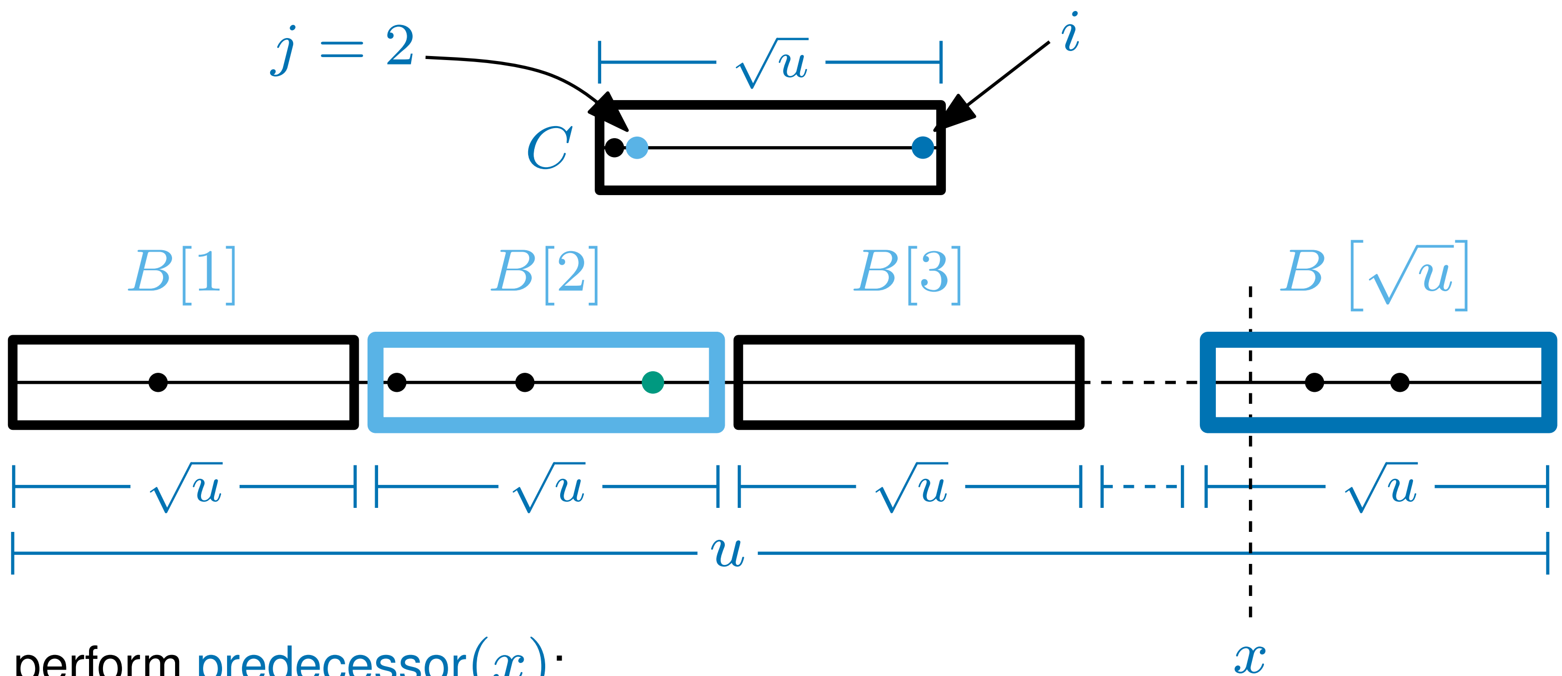
Step 3 If $x <$ the minimum in $B[i]$:

Compute $j =$ 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 predecessor(x):

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

Step 2 If $x \geqslant$ the minimum in $B[i]$:

Return the predecessor of x in $B[i]$

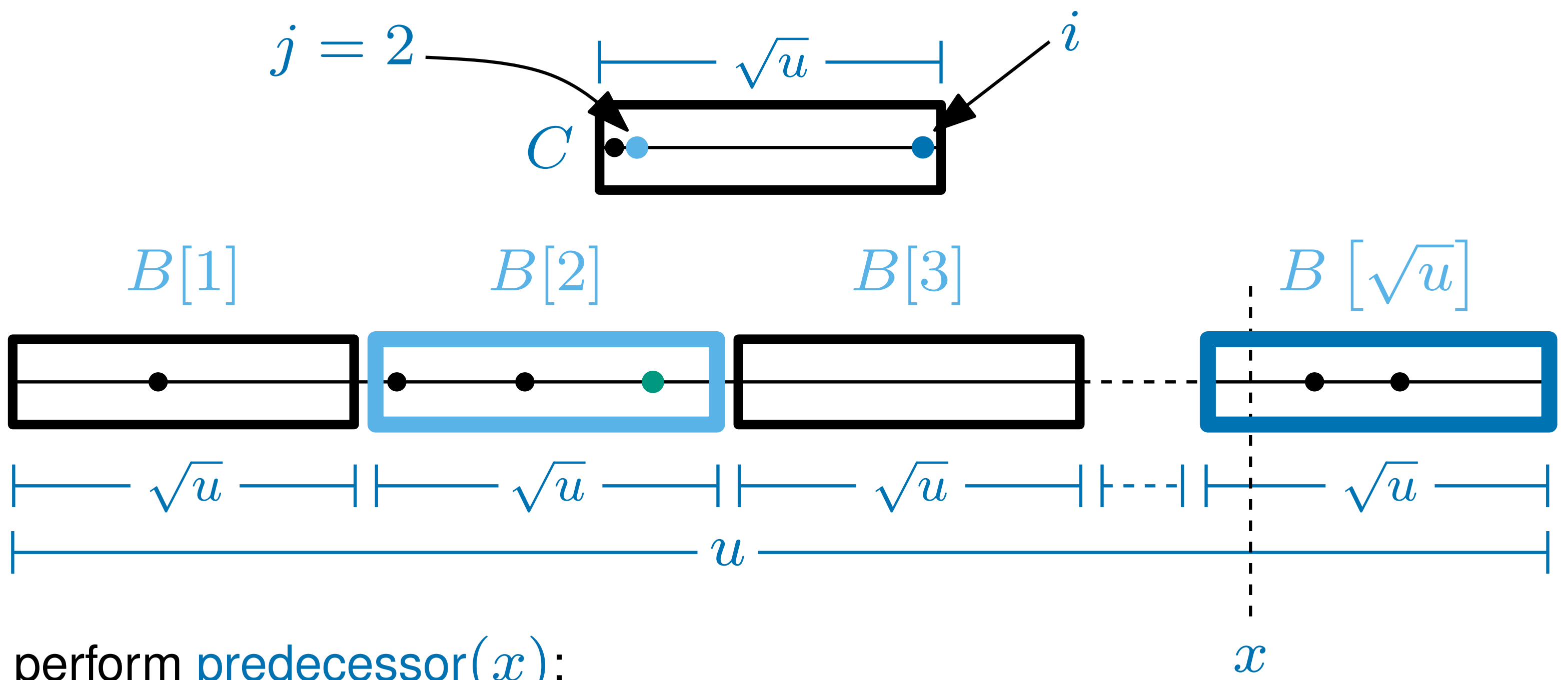
Step 3 If $x <$ the minimum in $B[i]$:

Compute $j =$ 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 predecessor(x):

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

Step 2 If $x \geqslant$ the minimum in $B[i]$:

Return the predecessor of x in $B[i]$

Step 3 If $x <$ the minimum in $B[i]$:

Compute $j =$ predecessor(i) in C

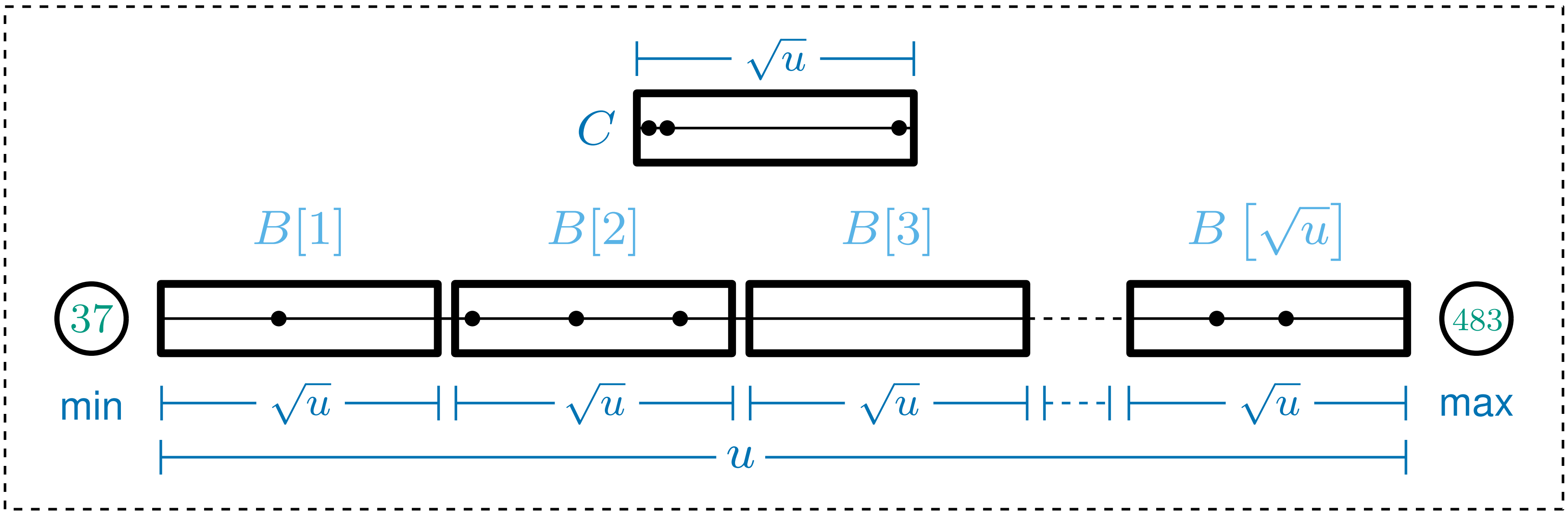
Return the maximum in $B[j]$

Now we make exactly one recursive call

(ignoring finding the min/max)

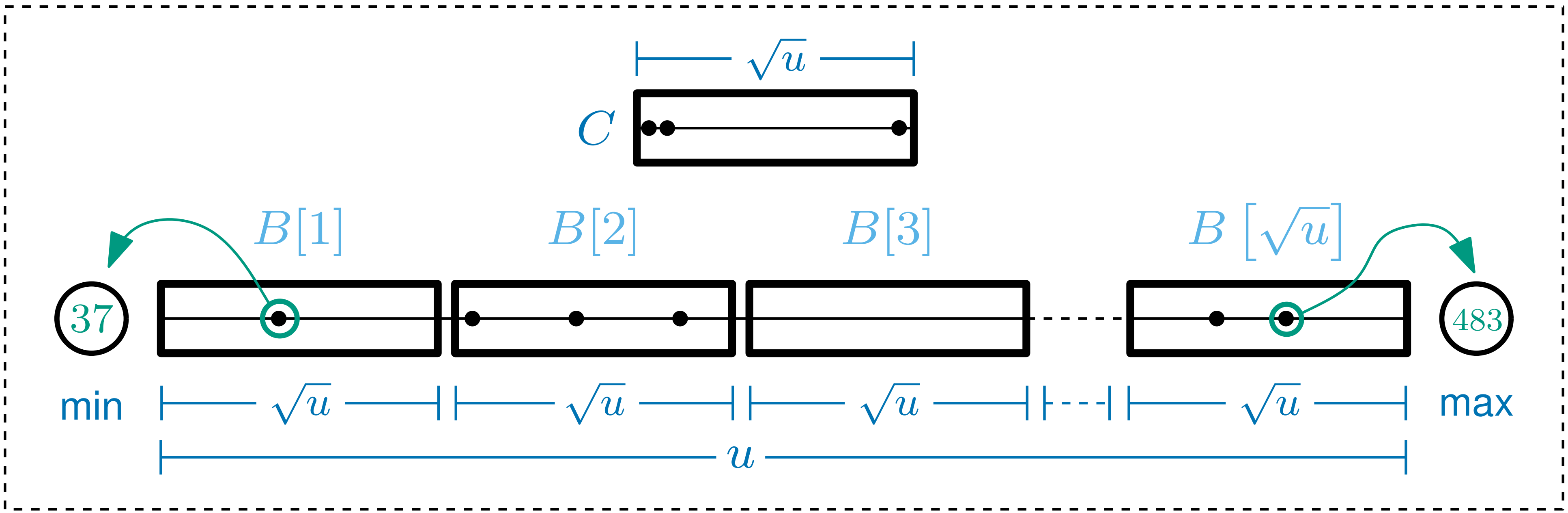
Finally: van Emde Boas Trees

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



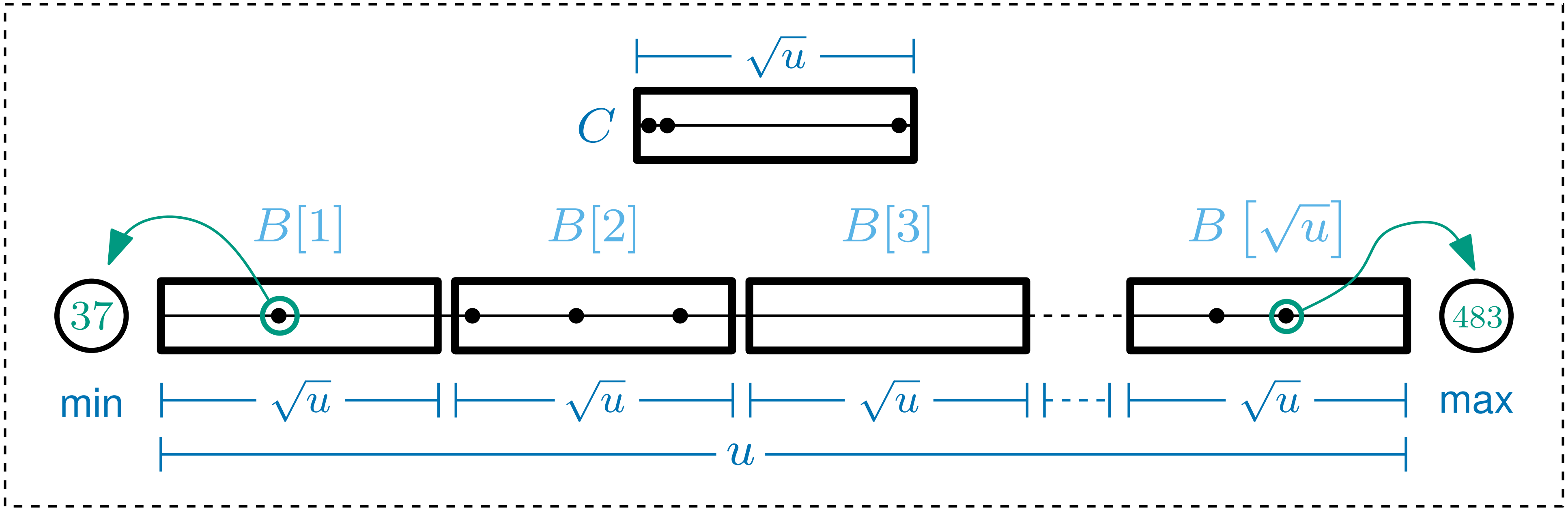
Finally: van Emde Boas Trees

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



Finally: van Emde Boas Trees

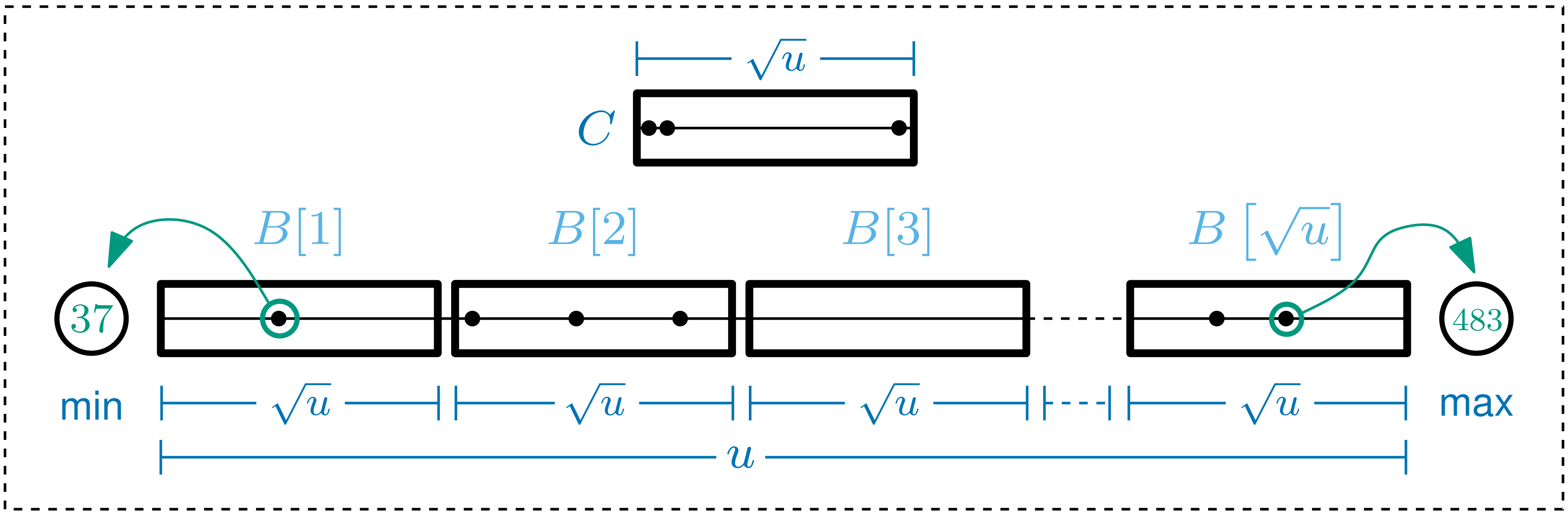
So that we can find the min/max quickly we store them seperately...



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 seperately...



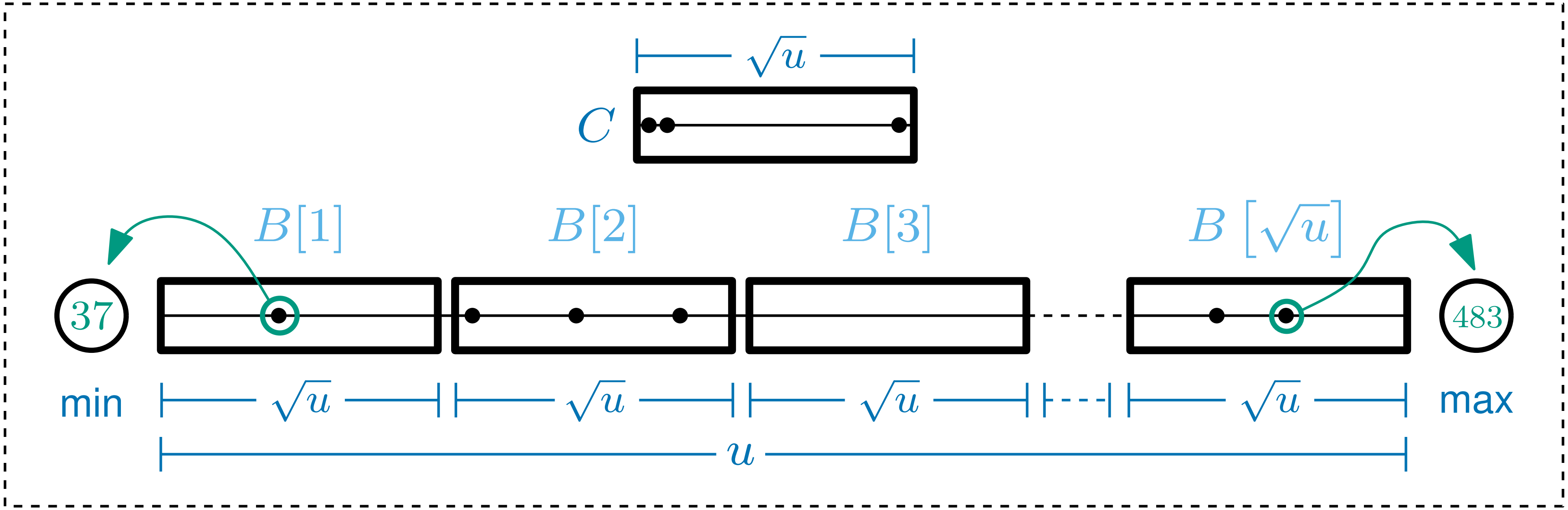
Remember that each $B[i]$ and C are also vEB (van Emde Boas) trees each over the universe $\{1, 2, 3, \ldots \sqrt{u}\}$

In particular $B[i]$ also stores it's min/max elements seperately

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 seperately...



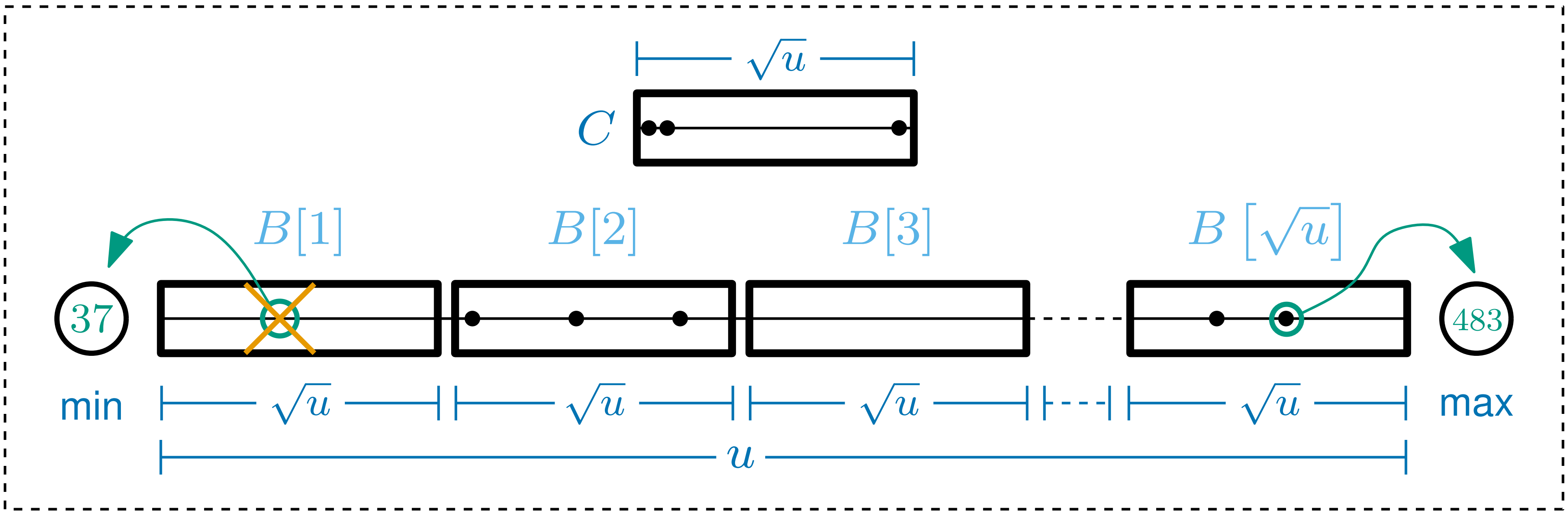
Remember that each $B[i]$ and C are also vEB (van Emde Boas) trees
each over the universe $\{1, 2, 3, \ldots \sqrt{u}\}$

In particular $B[i]$ also stores it's min/max elements seperately
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 seperately. . .



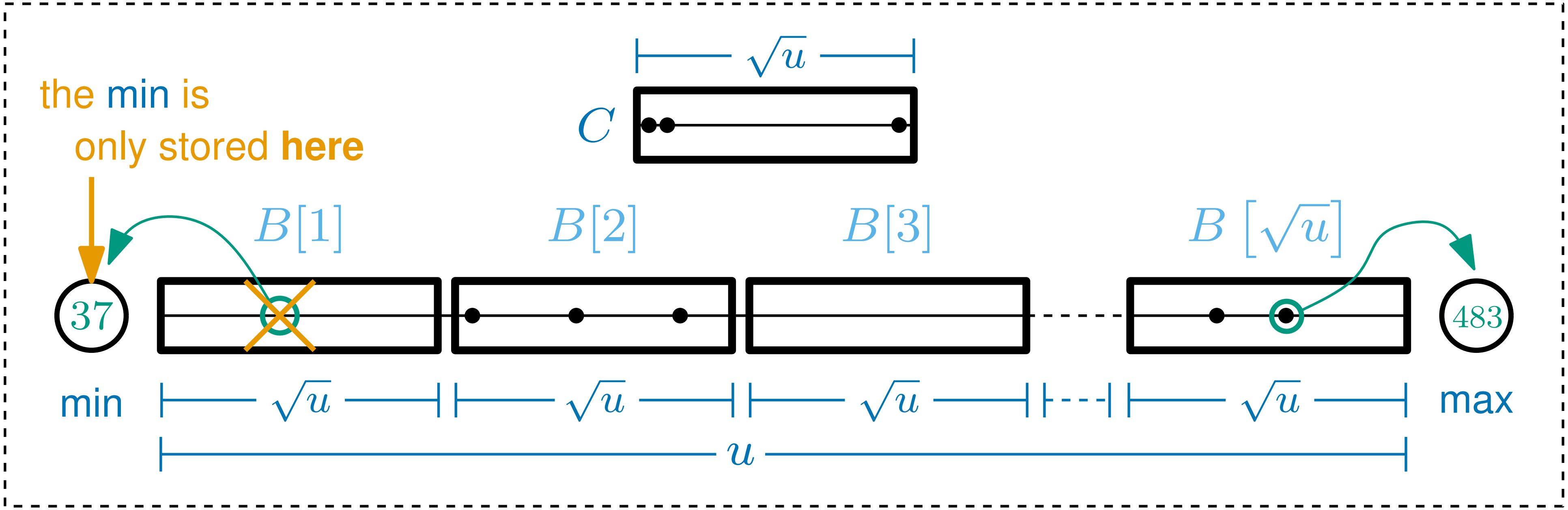
Remember that each $B[i]$ and C are also vEB (van Emde Boas) trees
each over the universe $\{1, 2, 3, \ldots \sqrt{u}\}$

In particular $B[i]$ also stores it's min/max elements seperately
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 seperately...



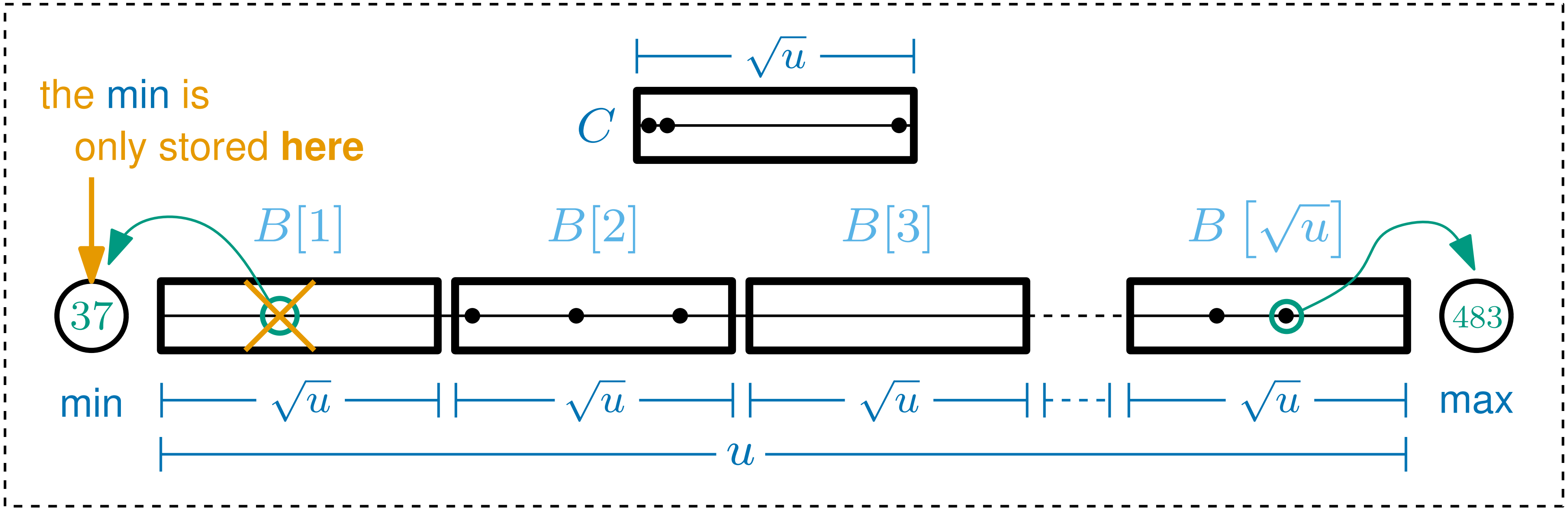
Remember that each $B[i]$ and C are also vEB (van Emde Boas) trees
each over the universe $\{1, 2, 3, \ldots \sqrt{u}\}$

In particular $B[i]$ also stores it's min/max elements seperately
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 seperately...

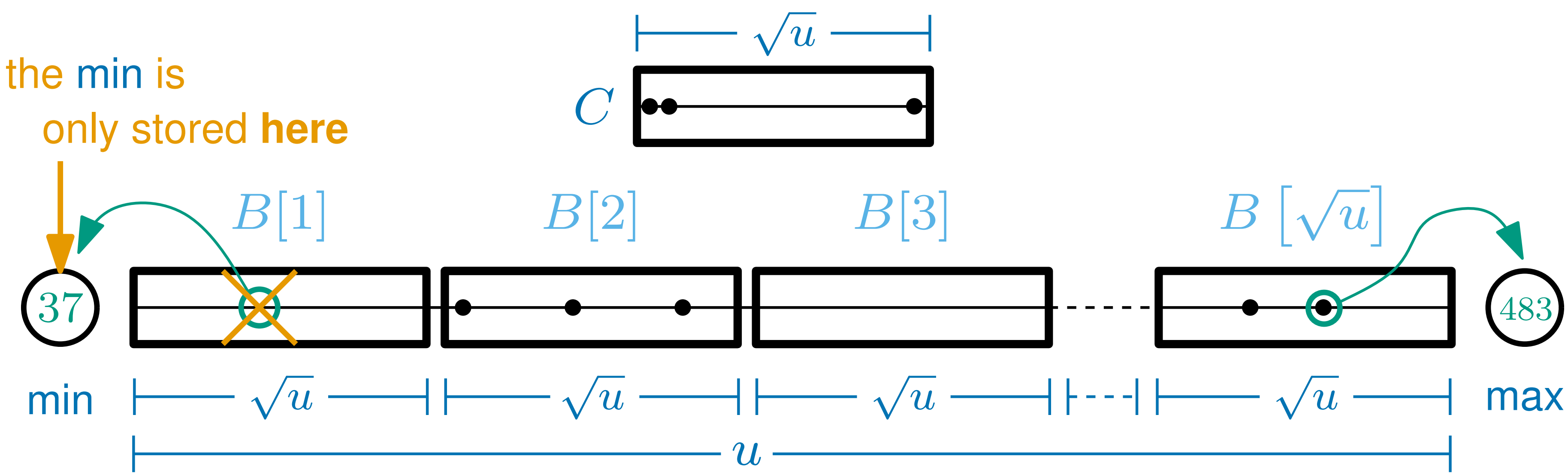


Remember that each $B[i]$ and C are also vEB (van Emde Boas) trees
each over the universe $\{1, 2, 3, \ldots \sqrt{u}\}$

In particular $B[i]$ also stores it's min/max elements seperately
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 add(x):

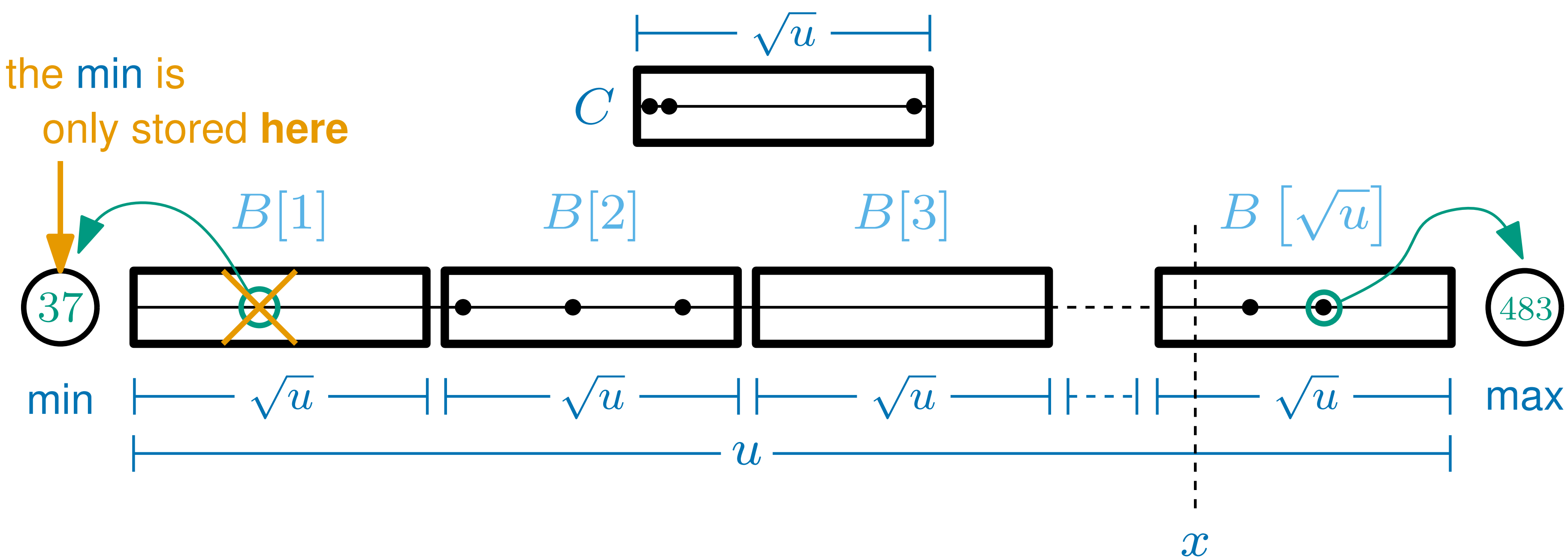
Step 1 Determine which $B[i]$ the element x belongs in

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):

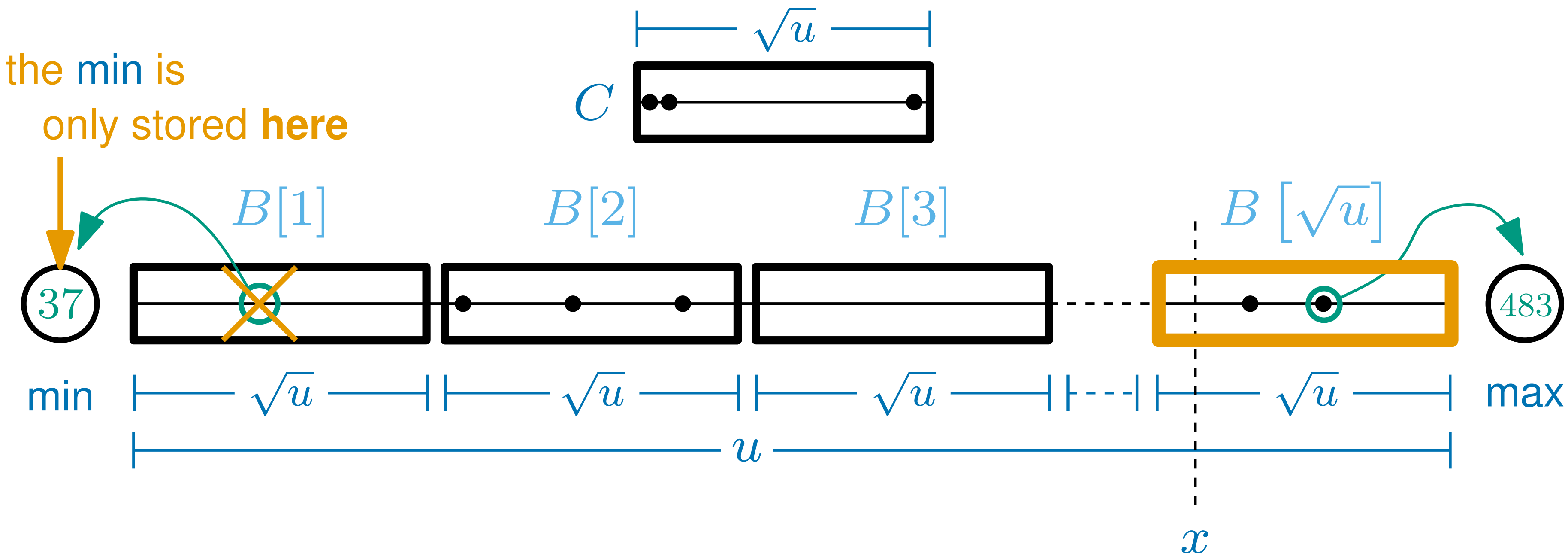
Step 1 Determine which $B[i]$ the element x belongs in

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 $B[i]$

Another look at add



To perform add(x):

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

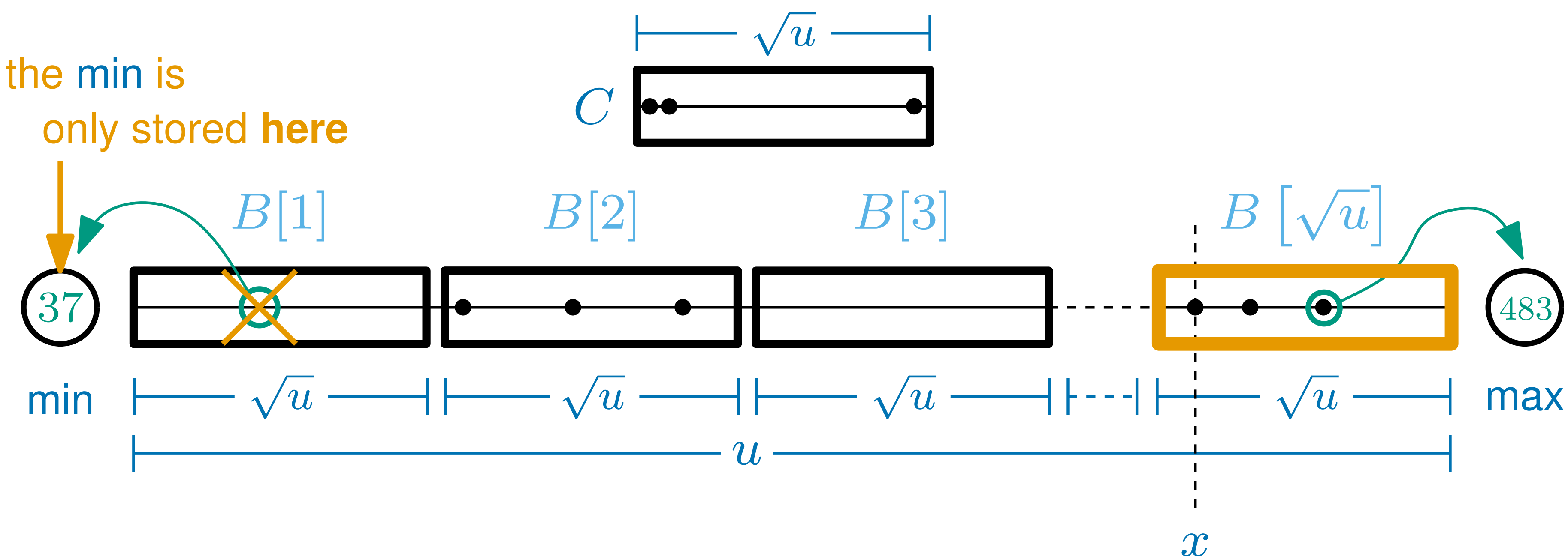
Step 2 If $B[i]$ is empty, add i to C

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

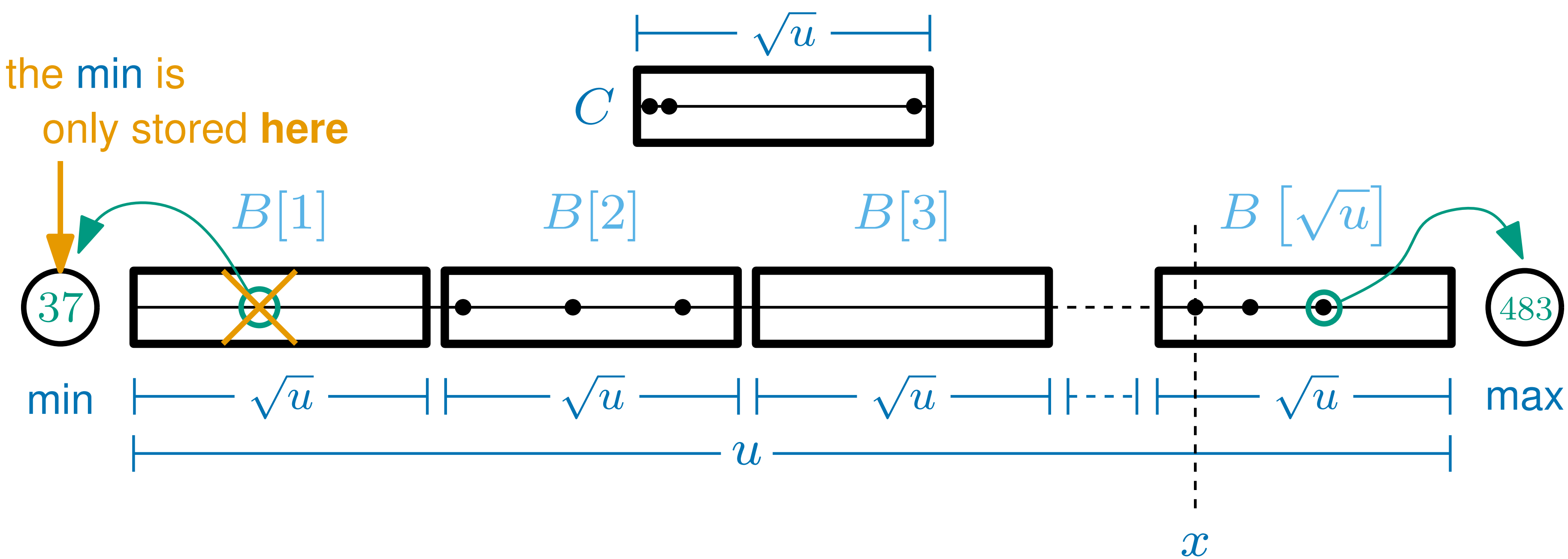
Step 2 If $B[i]$ is empty, add i to C

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 $B[i]$

we make
one recursive call

Another look at add



To perform add(x):

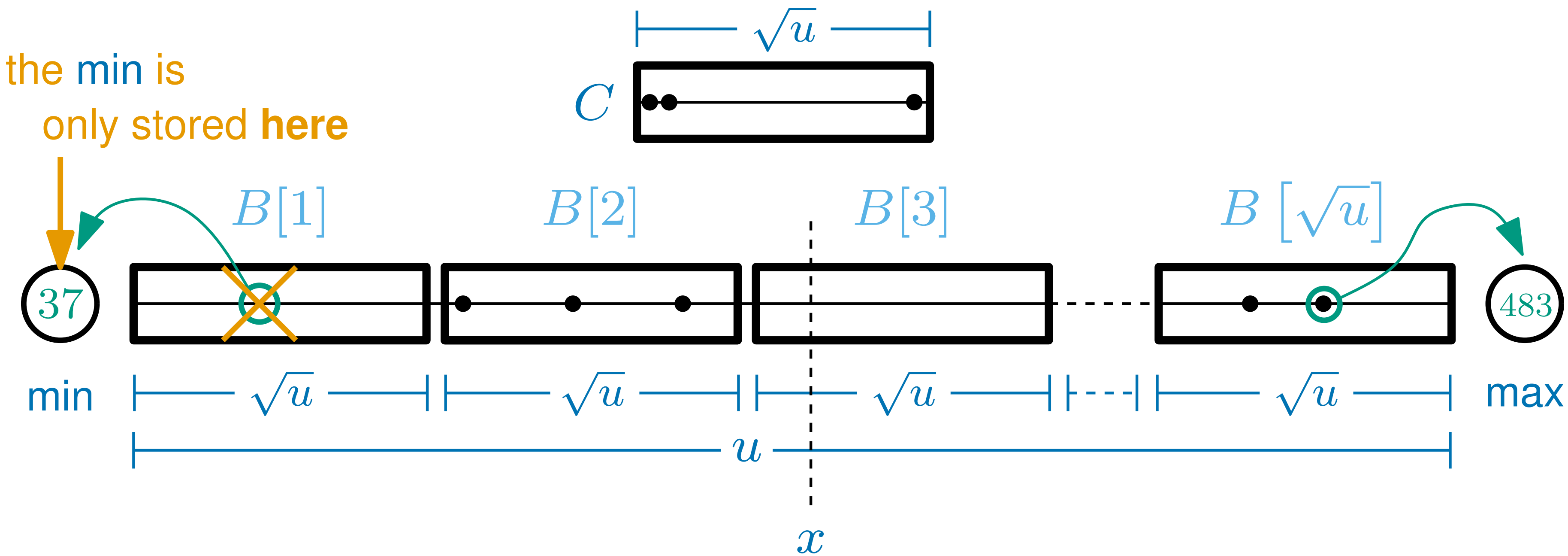
Step 1 Determine which $B[i]$ the element x belongs in

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 add(x):

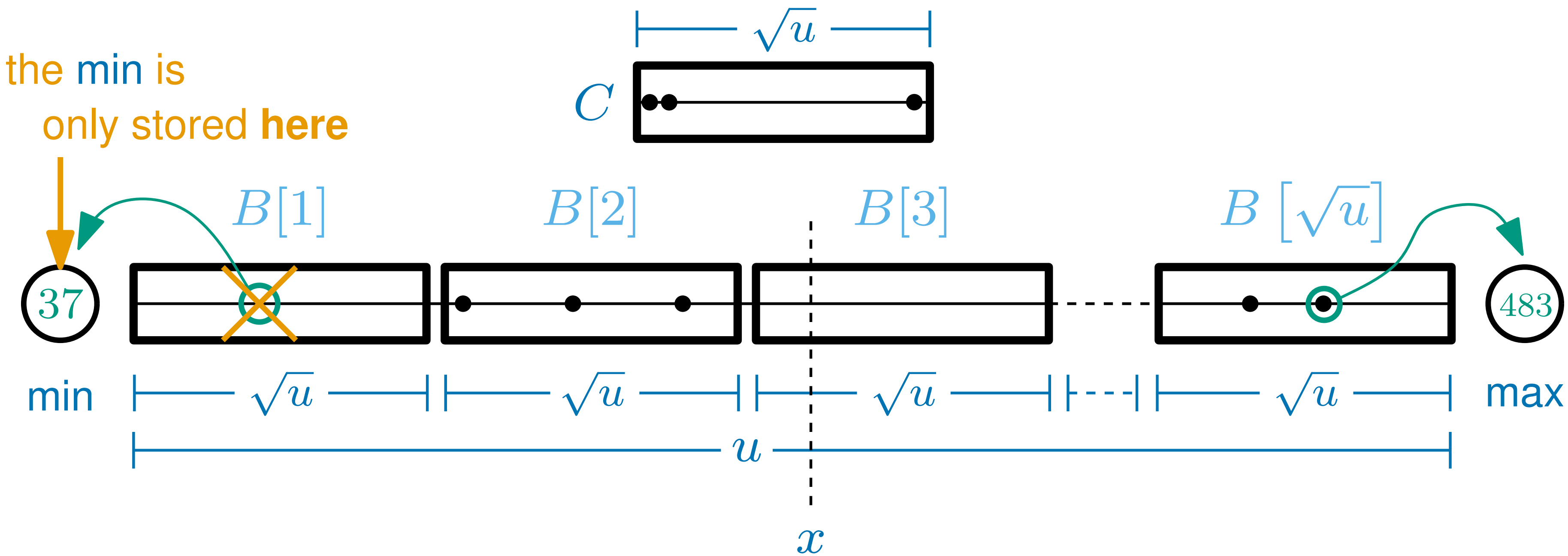
Step 1 Determine which $B[i]$ the element x belongs in

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

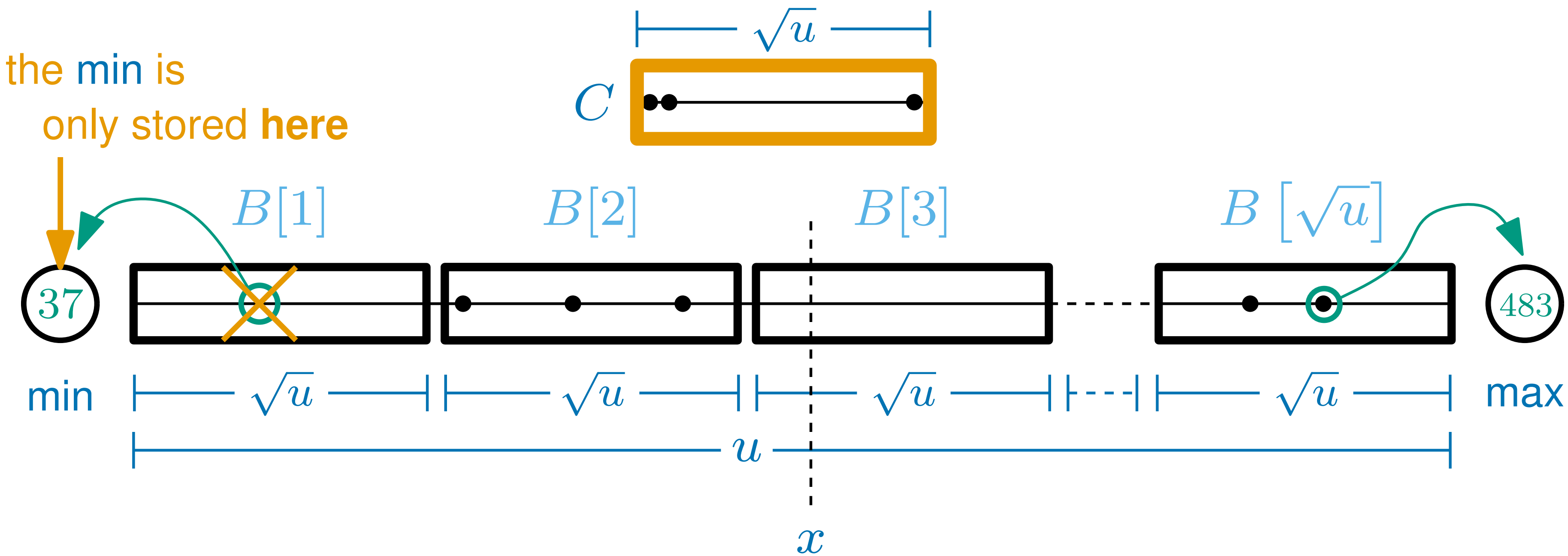
Step 2 If $B[i]$ is empty, add i to C

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 add(x):

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

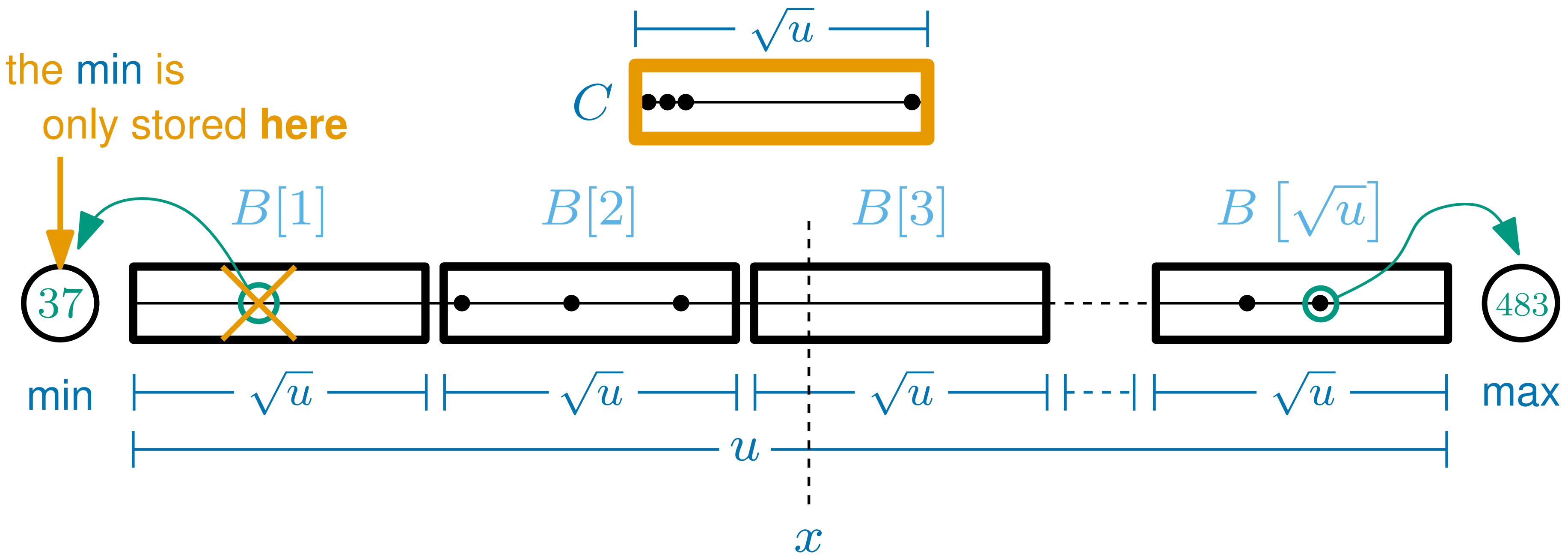
Step 2 If $B[i]$ is empty, add i to C

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

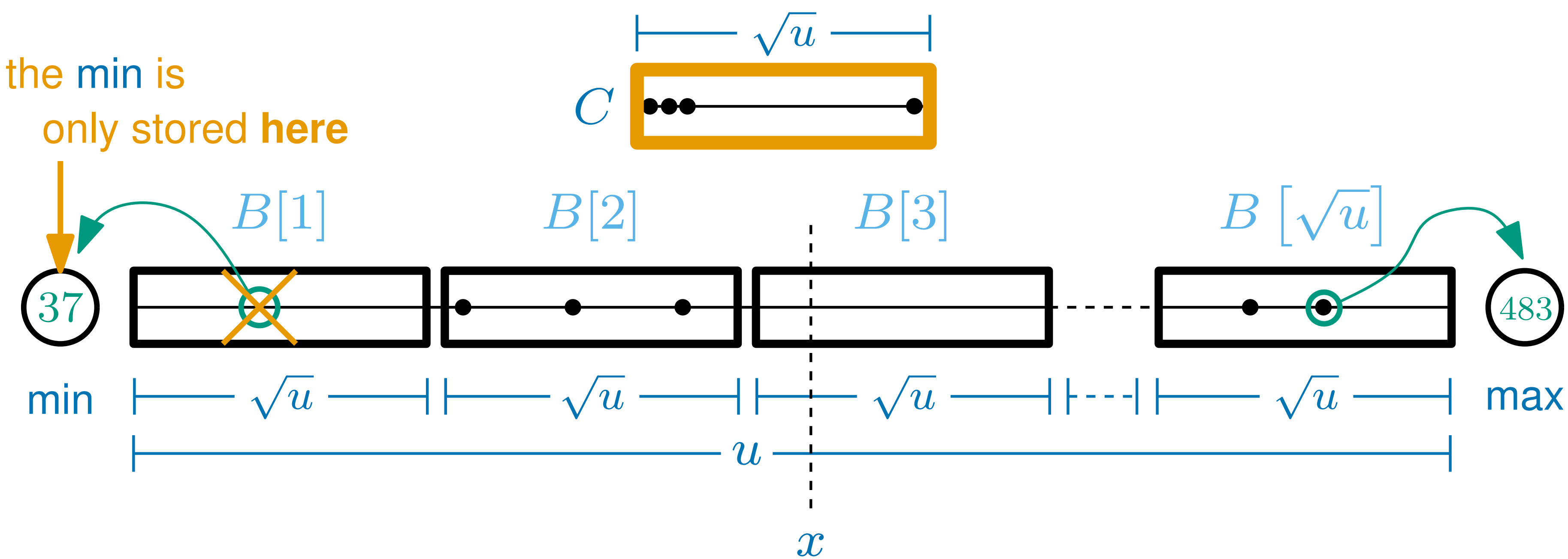
Step 2 If $B[i]$ is empty, add i to C

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 $B[i]$

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

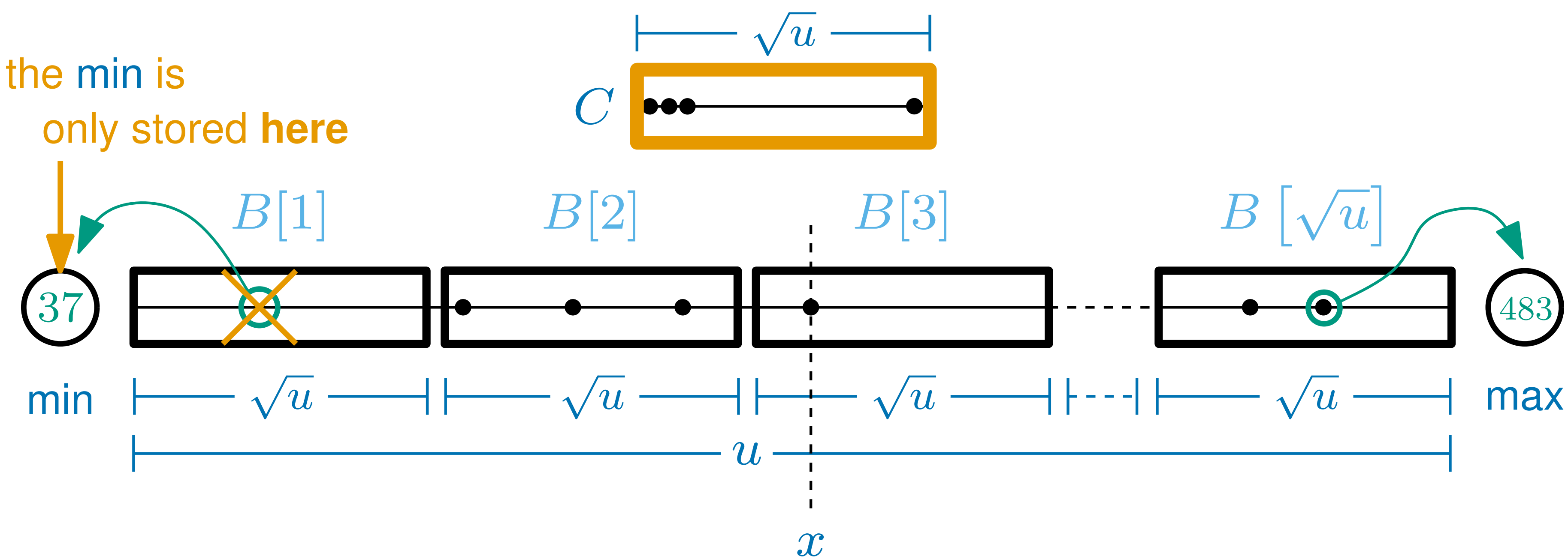
and set the min and max in $B[i]$ to x *(adjusting the offset)*

Step 3 If $B[i]$ is not empty, add x to $B[i]$

we make one recursive call

this is not recursive

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

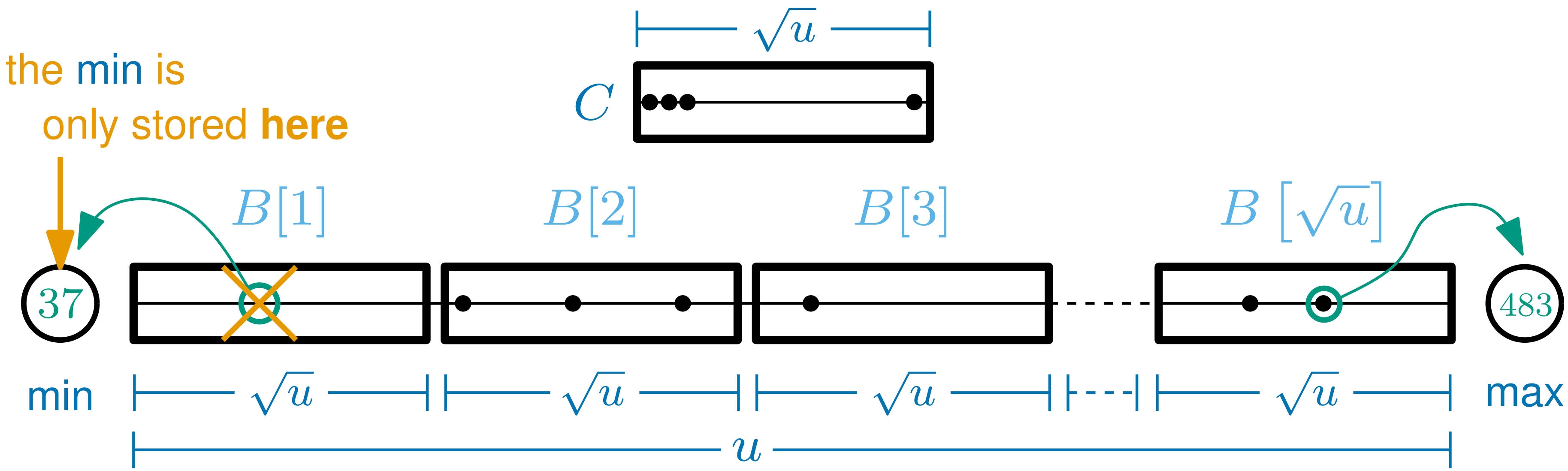
and set the min and max in $B[i]$ to x *(adjusting the offset)*

Step 3 If $B[i]$ is not empty, add x to $B[i]$

we make
one recursive call

this is not
recursive

Another look at add



Now we always make exactly one recursive call

To perform add(x):

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

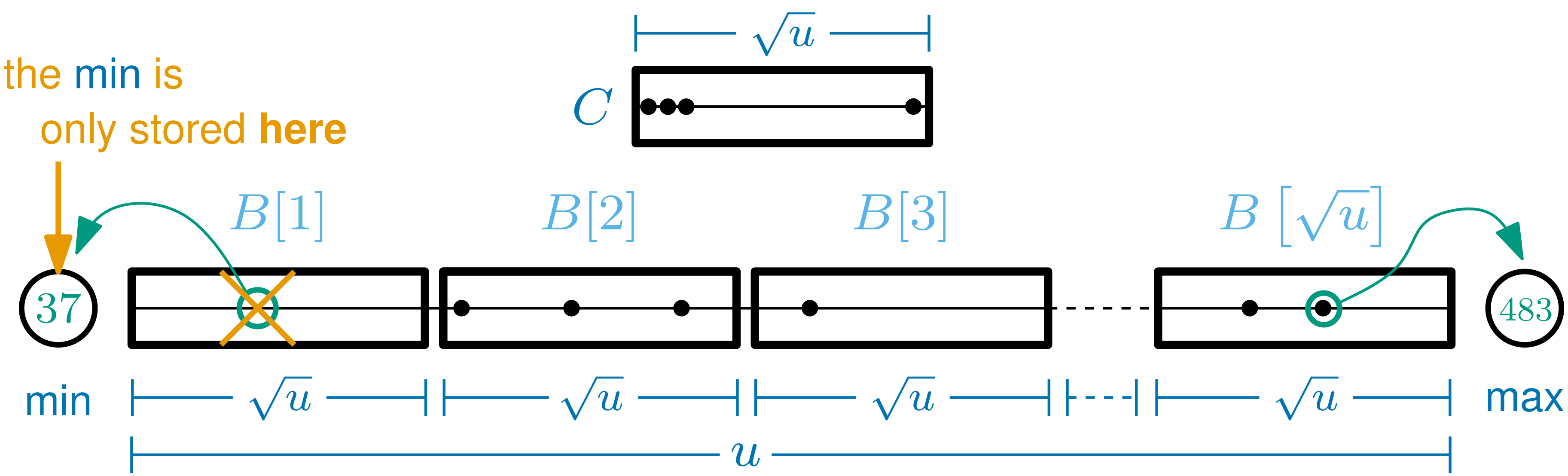
Step 2 If $B[i]$ is empty, add i to C

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

the min is
only stored **here**



Now we always make exactly one recursive call
but what happens when the min/max change?

To perform add(x):

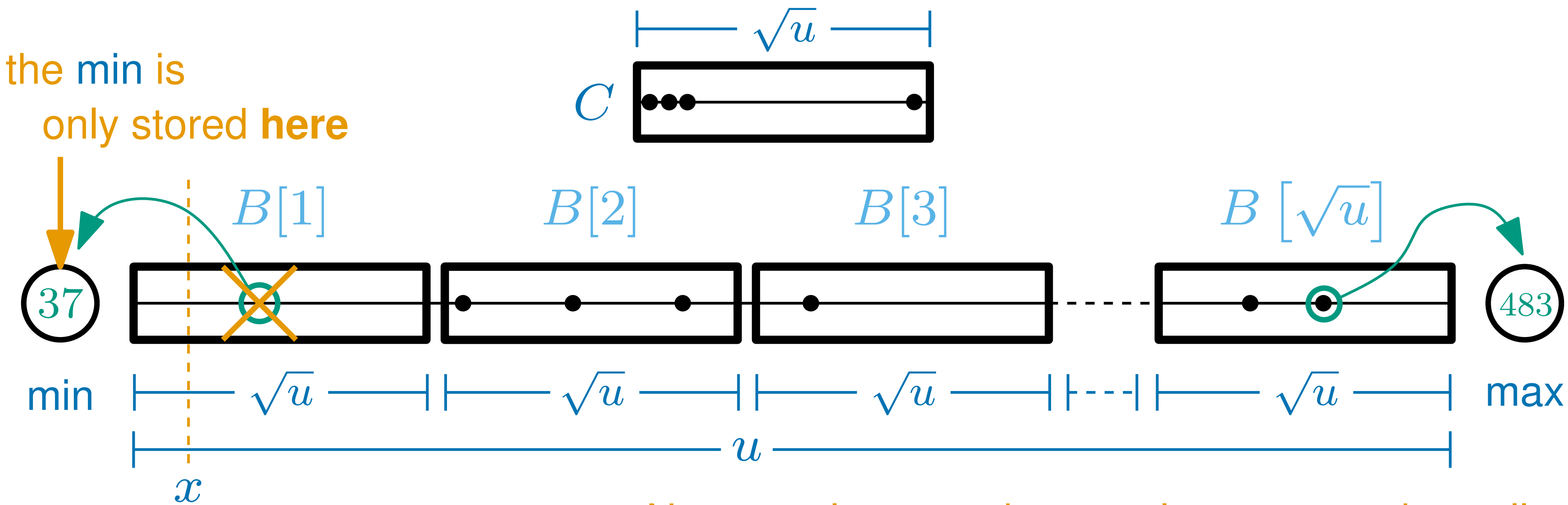
Step 1 Determine which $B[i]$ the element x belongs in

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



Now we always make exactly one recursive call
but what happens when the min/max change?

To perform add(x):

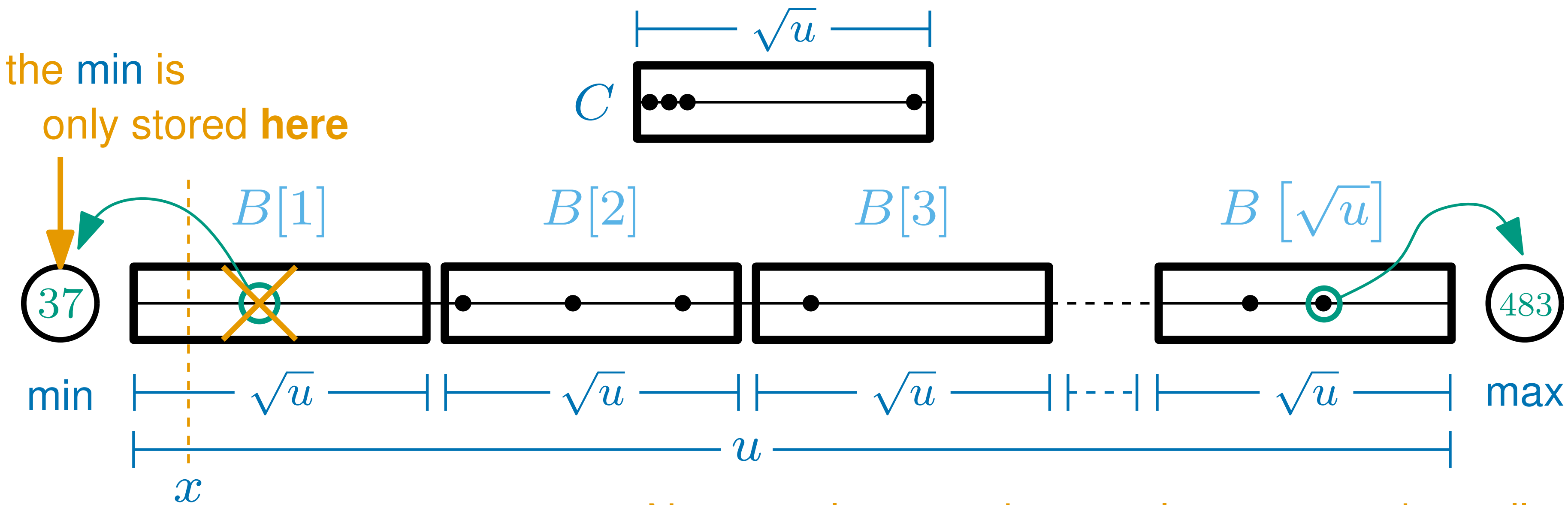
Step 1 Determine which $B[i]$ the element x belongs in

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



Now we always make exactly one recursive call
but what happens when the min/max change?

To perform add(x):

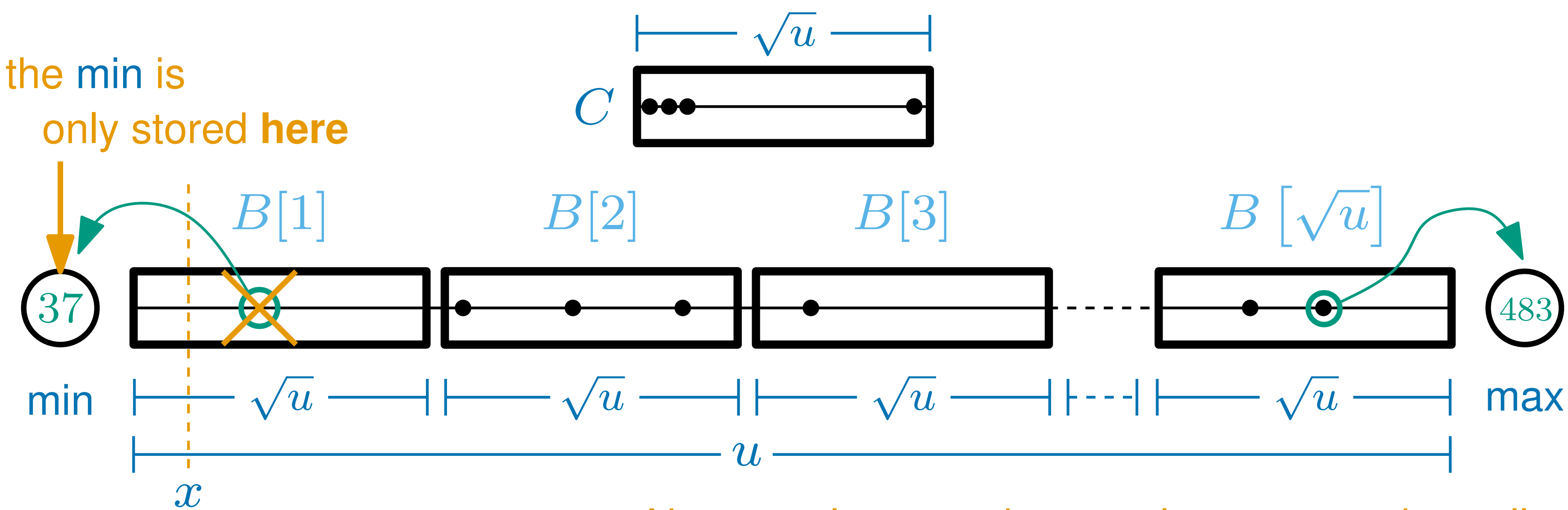
Step 1 Determine which $B[i]$ the element x belongs in

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



Now we always make exactly one recursive call

but what happens when the min/max change?

To perform add(x):

Step 0 If $x <$ min then swap x and min

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

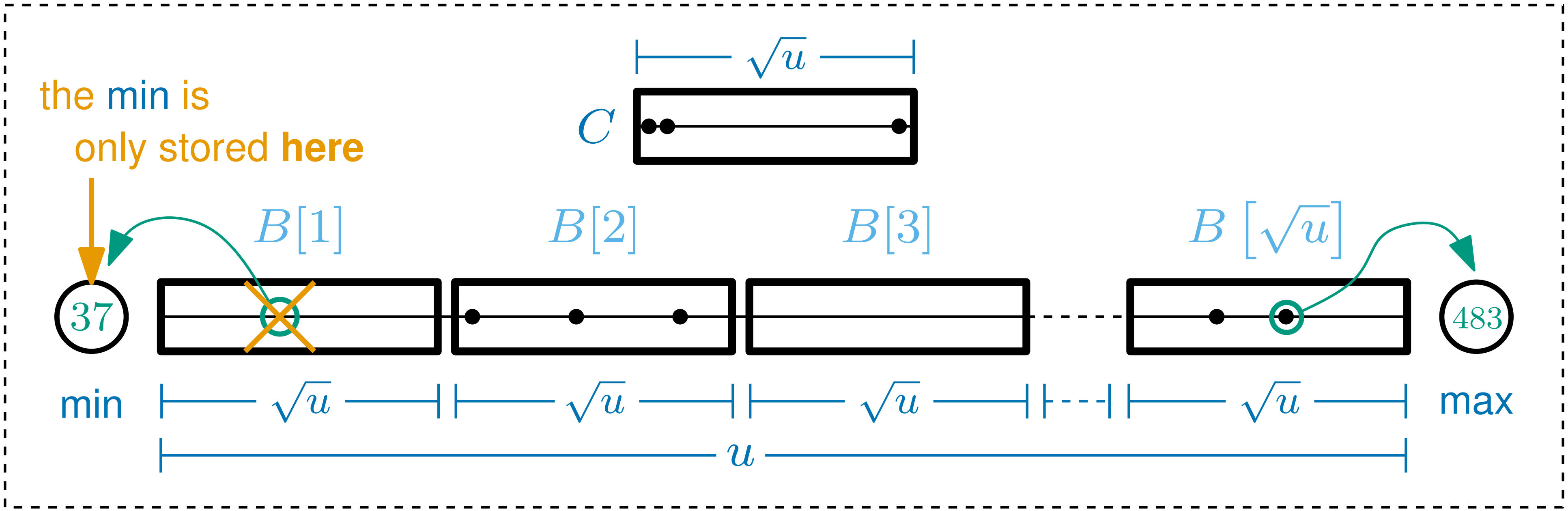
Step 2 If $B[i]$ is empty, add i to C

and set the min and max in $B[i]$ to x *(adjusting the offset)*

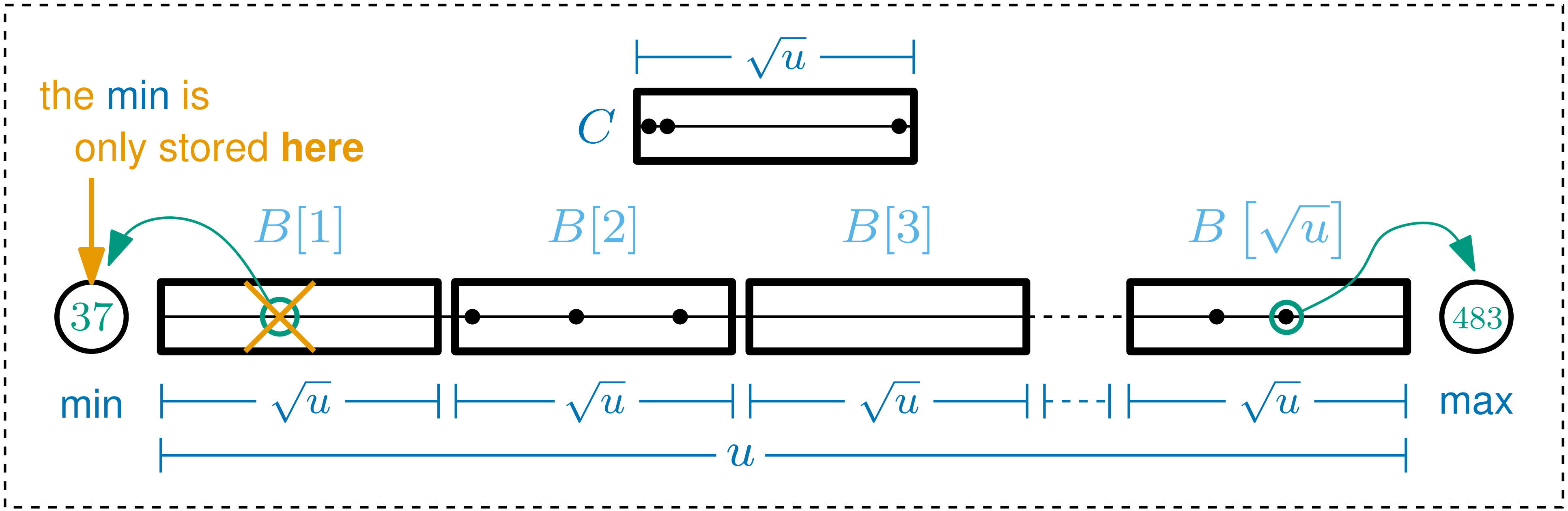
Step 3 If $B[i]$ is not empty, add x to $B[i]$

Step 4 Update the max

Time Complexity

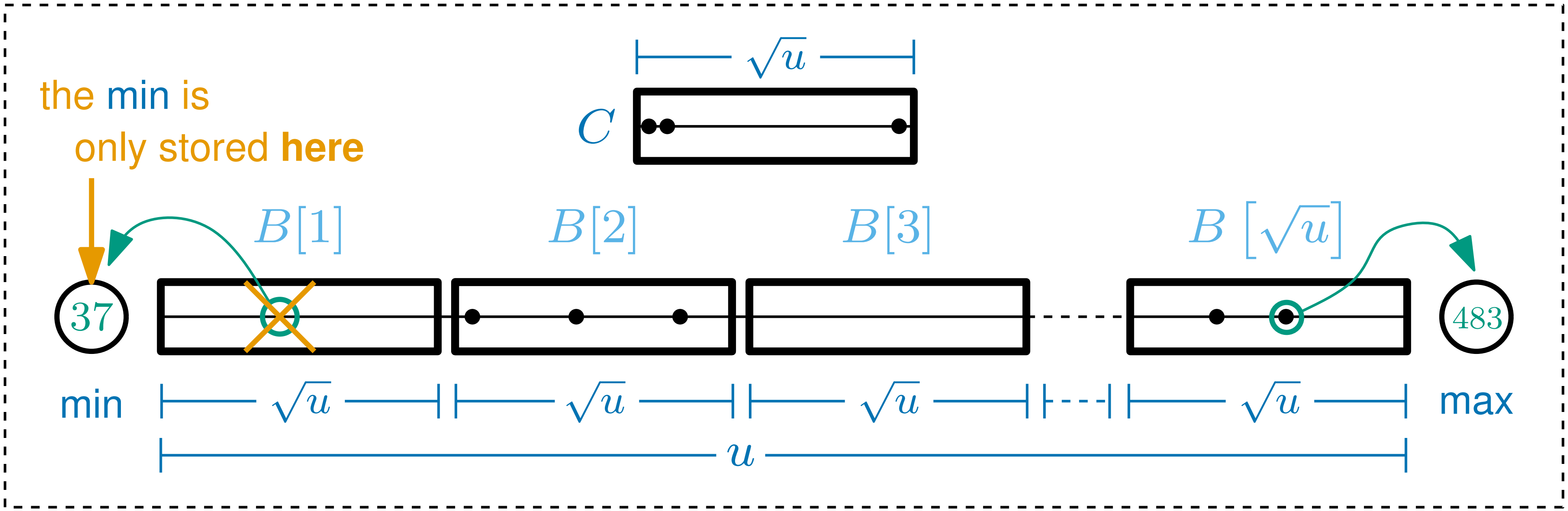


Time Complexity



We have seen that the operations add and predecessor can be defined so that they make only one recursive call

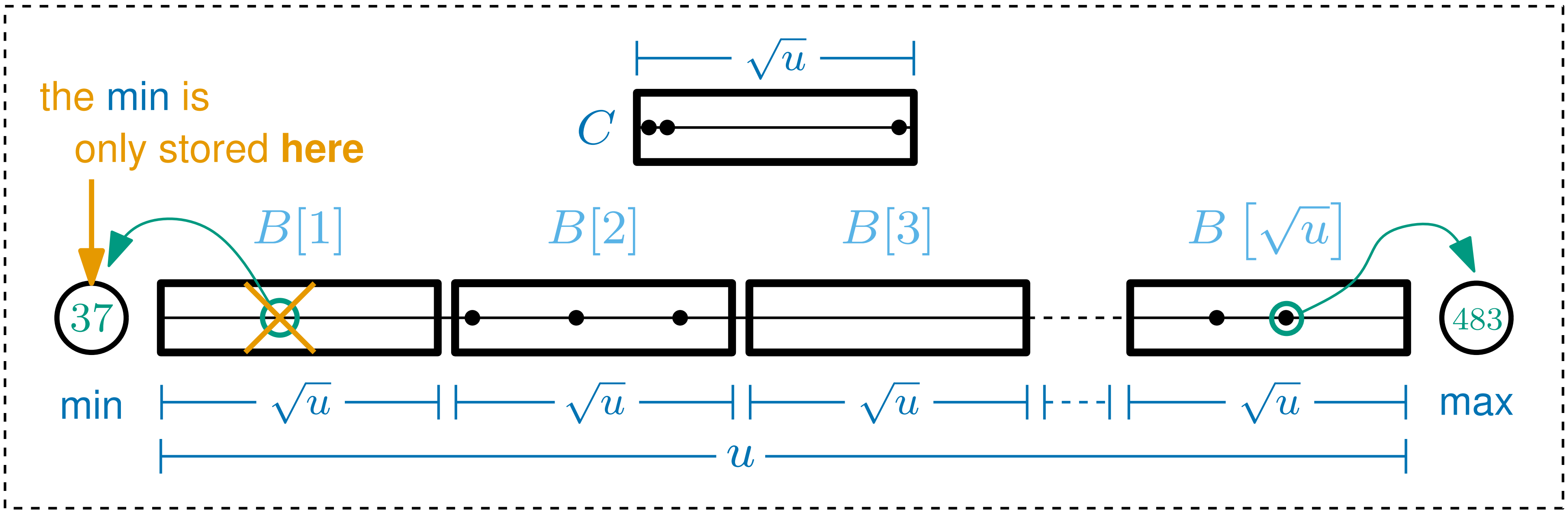
Time Complexity



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

Time Complexity

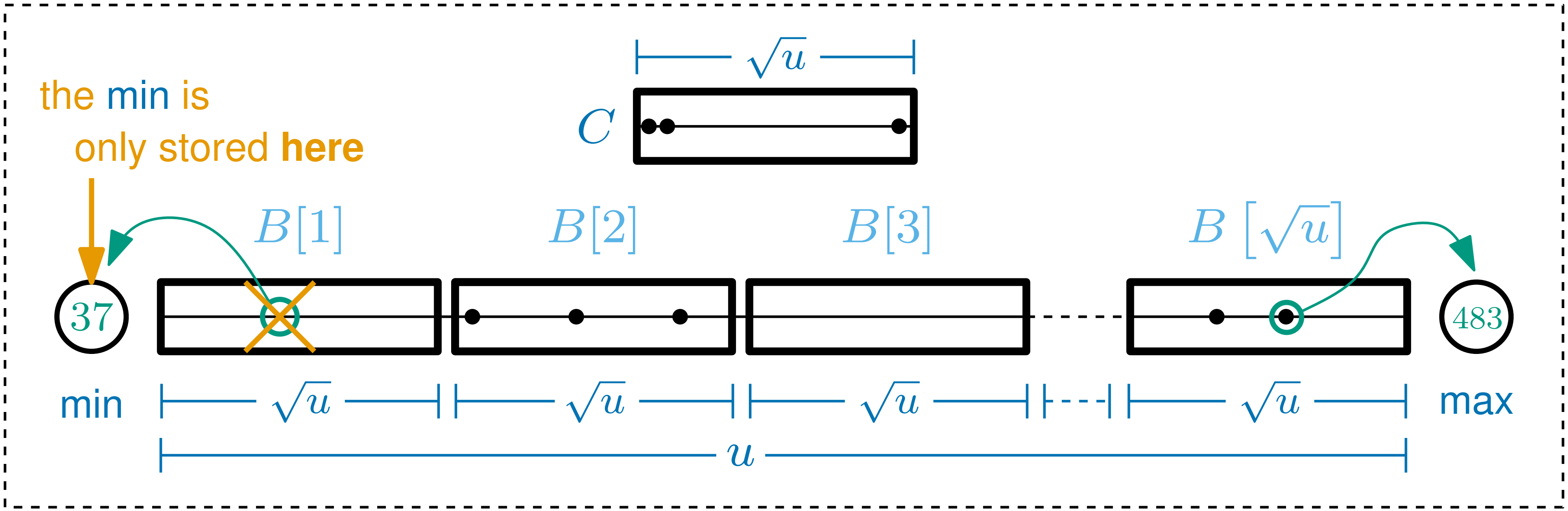


We have seen that the operations add and predecessor can be defined
so that they make only one recursive call

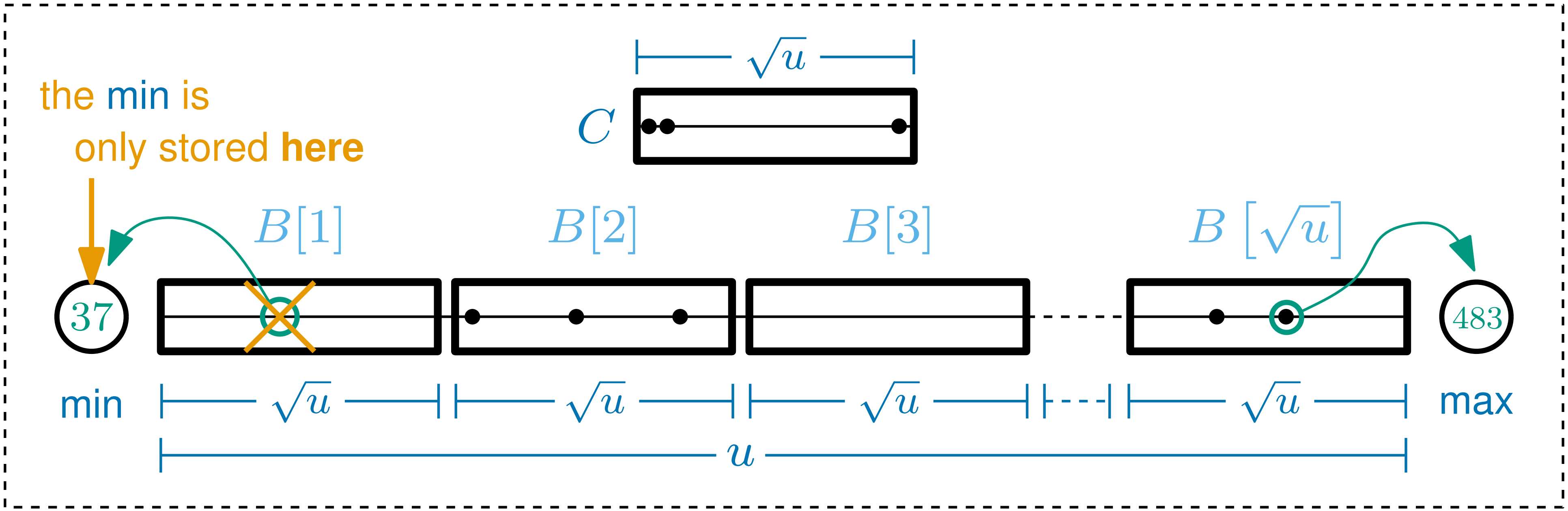
The operations lookup, delete and successor can
all also be defined in a similar, recursive manner
so that they make only one recursive call

How long do the operations take?

Time Complexity



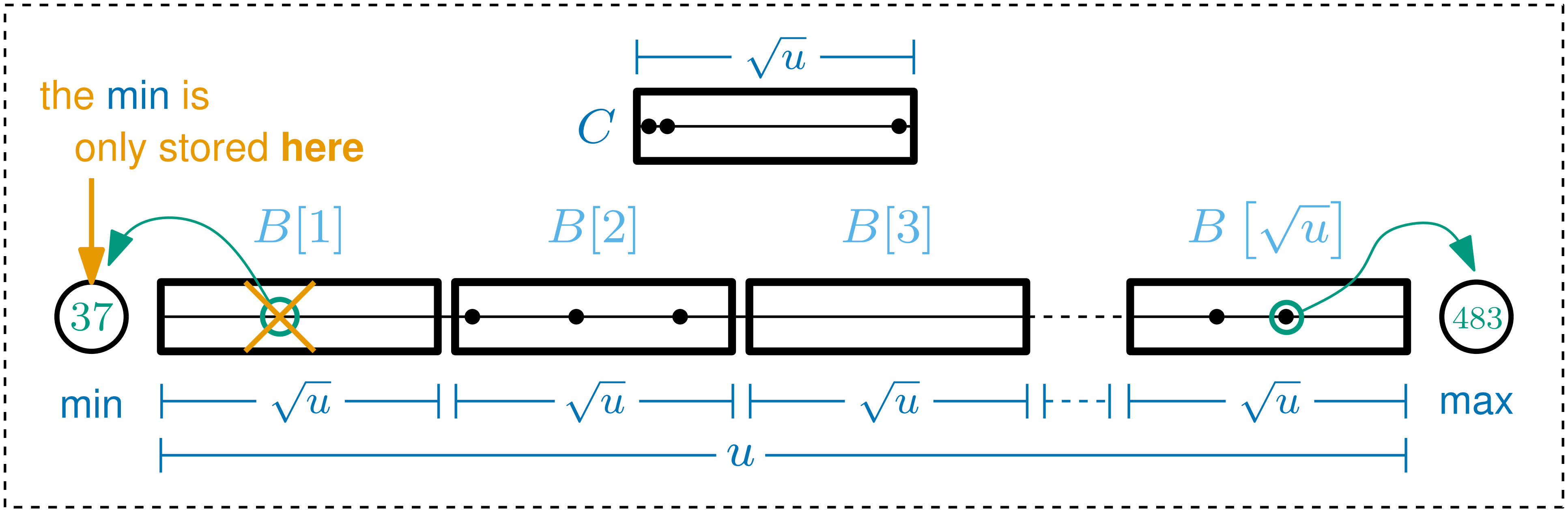
Time Complexity



Let $T(u)$ be the time complexity of the add operation

(where u is the universe size)

Time Complexity

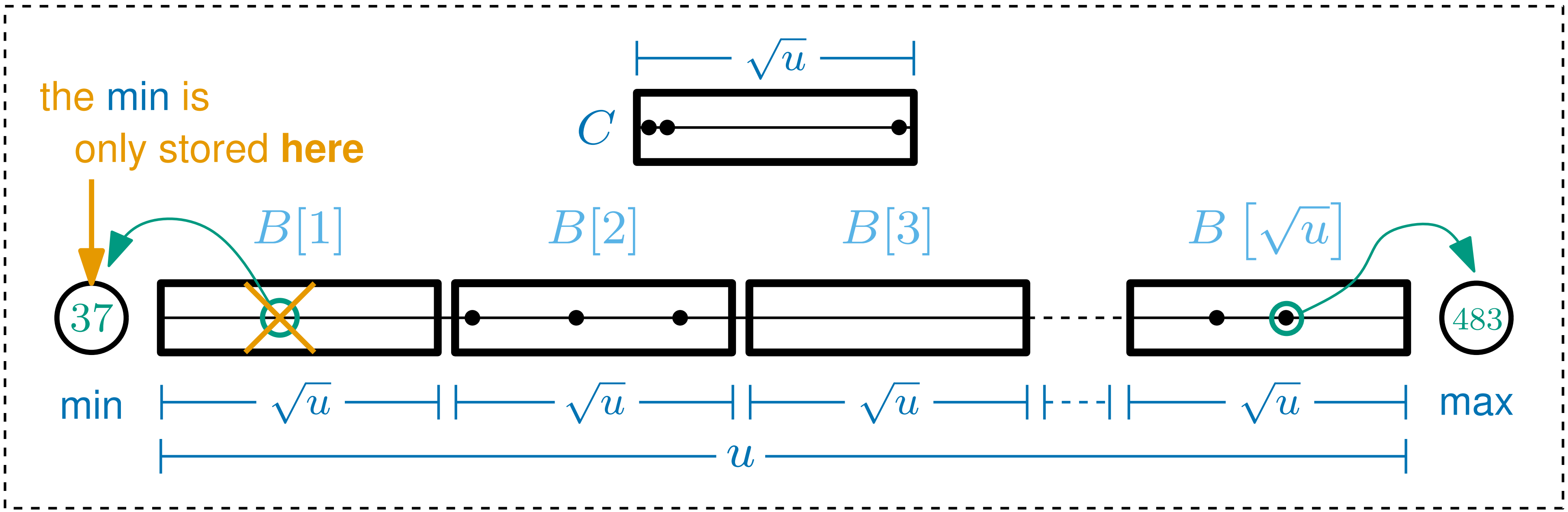


Let $T(u)$ be the time complexity of the add operation

(where u is the universe size)

We have that, $T(u) = T(\sqrt{u}) + O(1)$

Time Complexity



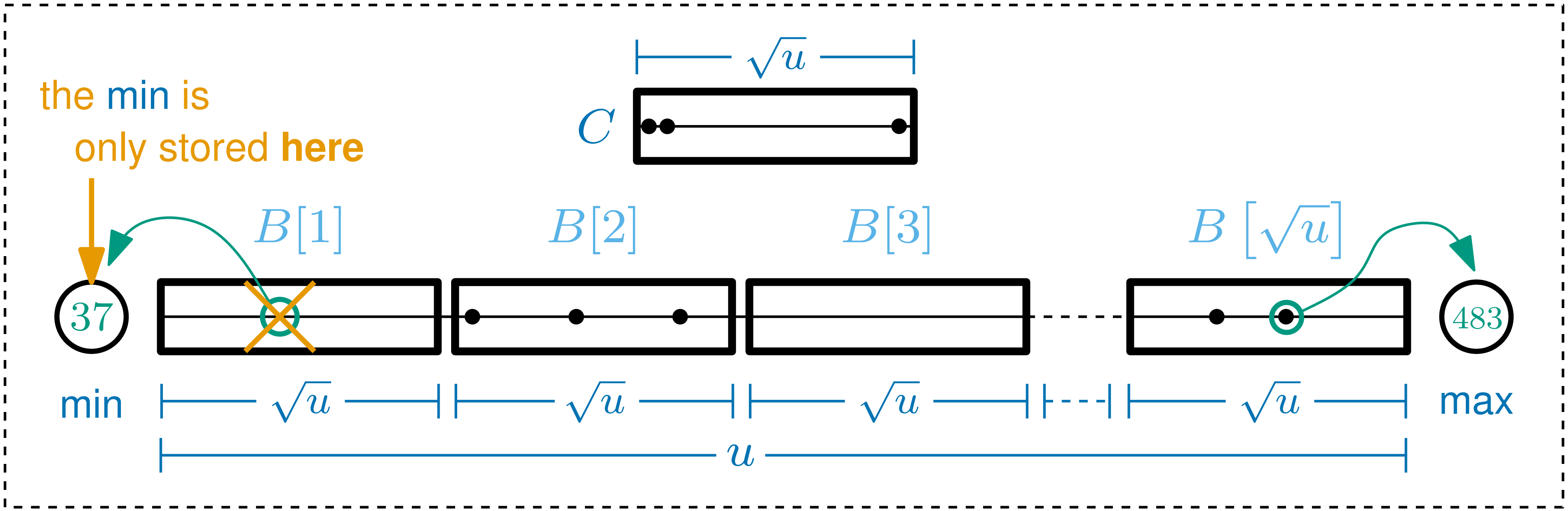
Let $T(u)$ be the time complexity of the add operation

(where u is the universe size)

We have that, $T(u) = T(\sqrt{u}) + O(1)$

Using substitution and the master method you can show that... $T(u) = O(\log \log u)$

Time Complexity



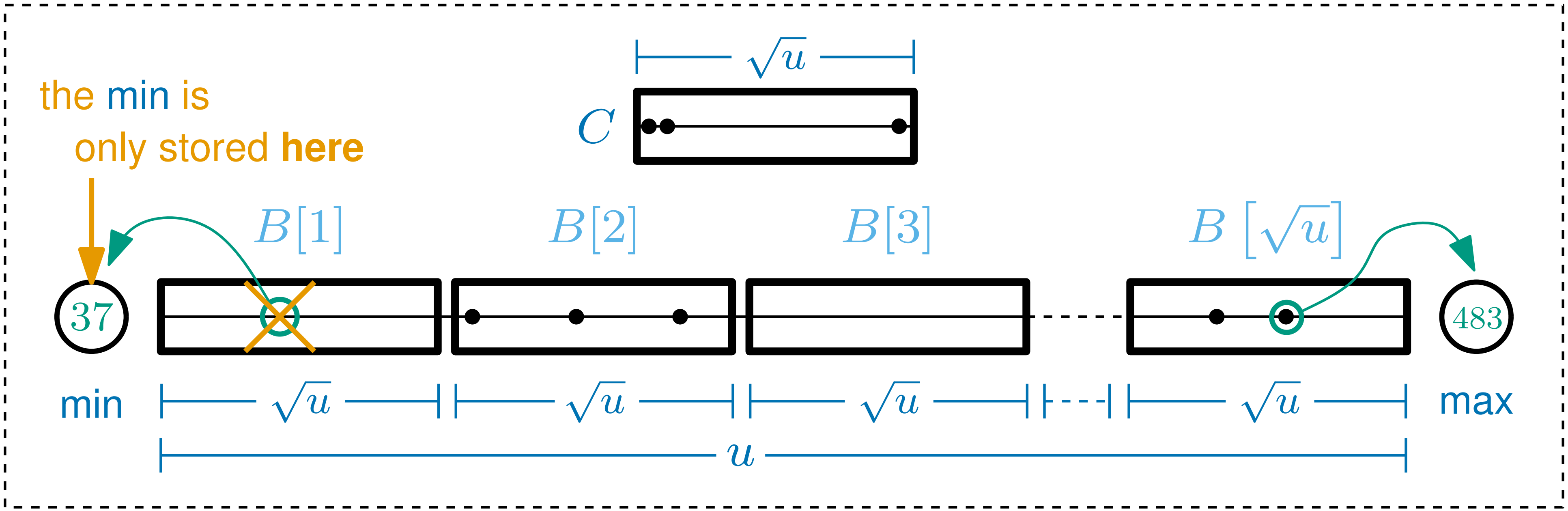
Let $T(u)$ be the time complexity of the predecessor operation

(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... $T(u) = O(\log \log u)$

Time Complexity



Let $T(u)$ be the time complexity of the predecessor operation

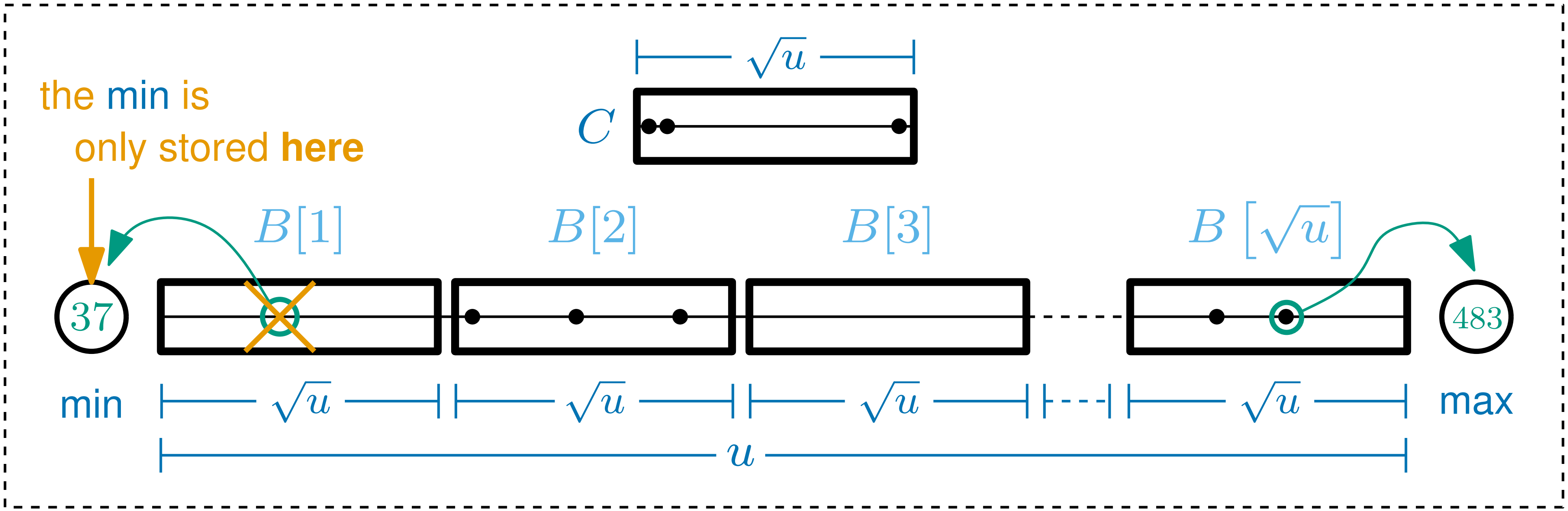
(where u is the universe size)

We have that, $T(u) = T(\sqrt{u}) + O(1)$

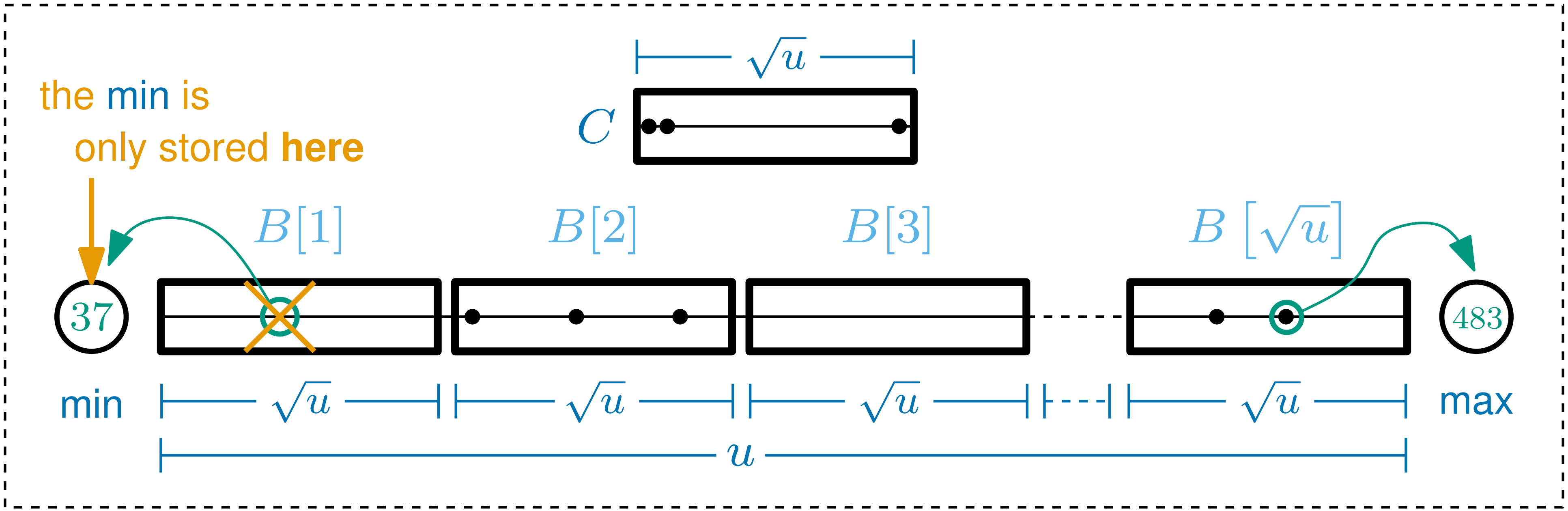
Using substitution and the master method you can show that... $T(u) = O(\log\log u)$

this holds for all the operations

Space Complexity

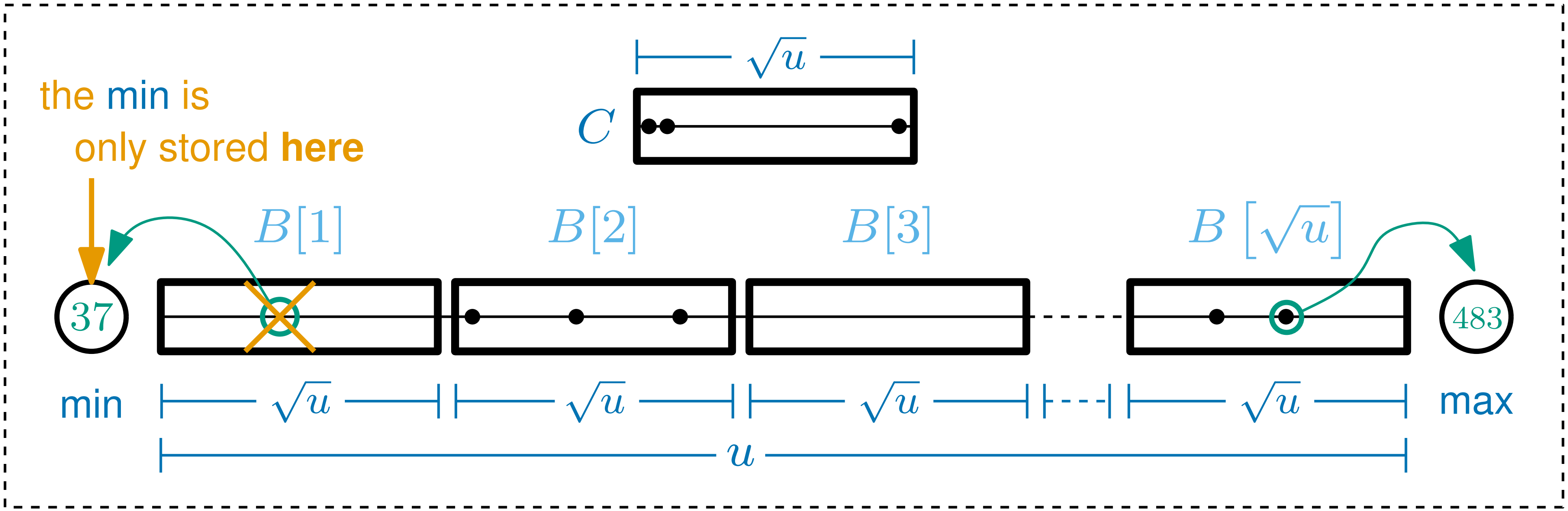


Space Complexity



Let $Z(u)$ be the space used by a vEB tree over a universe of size u

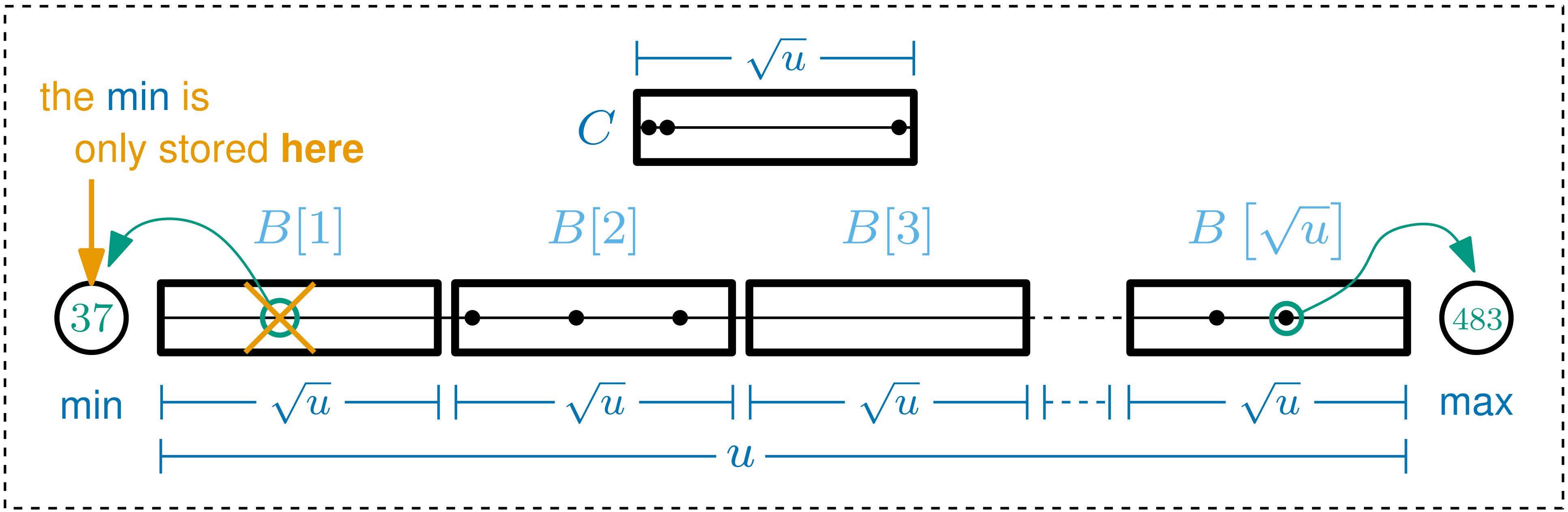
Space Complexity



Let $Z(u)$ be the space used by a vEB tree over a universe of size u

We have that, $Z(u) = (\sqrt{u} + 1) \cdot Z(\sqrt{u}) + O(1)$

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... $Z(u) = O(u)$

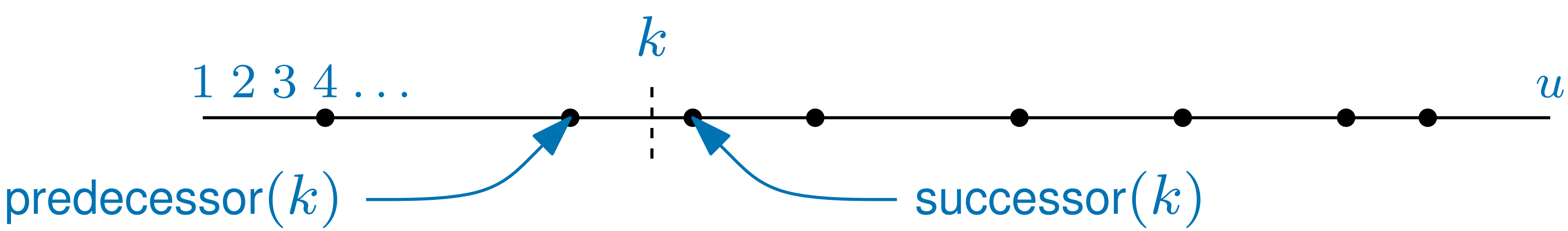
van Emde Boas Trees

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 $x \leqslant k$
successor(k)	Return the smallest integer x in S such that $x \geqslant k$



All operations take $O(\log \log u)$ worst case time

and the space used is $O(u)$

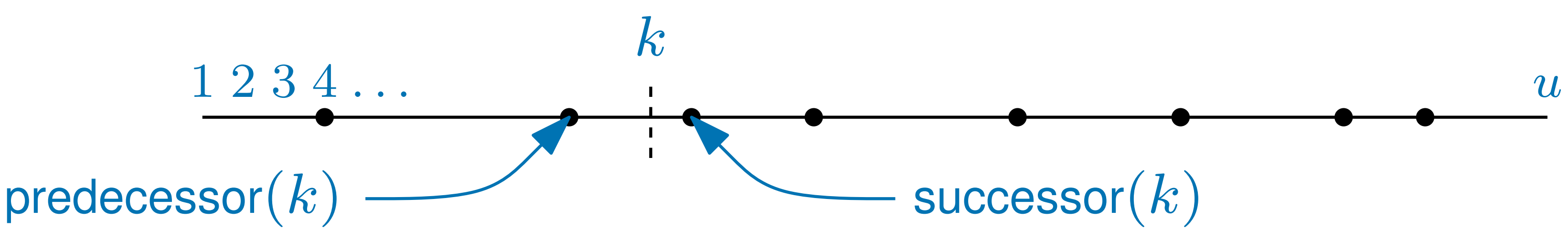
van Emde Boas Trees

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 $x \leqslant k$
successor(k)	Return the smallest integer x in S such that $x \geqslant k$



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)