10 van Emde Boas Trees

Dynamic Set Data Structure S:

- $S.\mathrm{insert}(x)$
- $S.\mathrm{delete}(x)$
- $S.\mathrm{search}(x)$
- $S.\mathrm{min}()$
- $S.\mathrm{max}()$
- $S.\mathrm{succ}(x)$
- $S.\mathrm{pred}(x)$

10 van Emde Boas Trees

For this chapter we ignore the problem of storing satellite data:

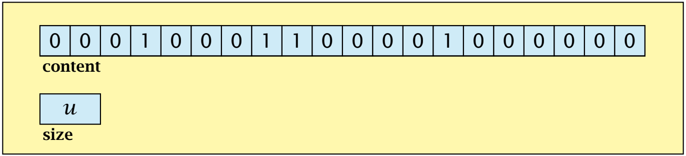
- $S.\text{insert}(x)$: Inserts x into S.
- $S.\text{delete}(x)$: Deletes x from S. Usually assumes that $x \in S$.
- $S.\text{member}(x)$: Returns 1 if $x \in S$ and 0 otw.
- $S.\text{min}()$: Returns the value of the minimum element in S.
- $S.\text{max}()$: Returns the value of the maximum element in S.
- $S.\text{succ}(x)$: Returns successor of x in S. Returns null if x is maximum or larger than any element in S. Note that x needs not to be in S.
- $S.\text{pred}(x)$: Returns the predecessor of x in S. Returns null if x is minimum or smaller than any element in S. Note that x needs not to be in S.

10 van Emde Boas Trees

Can we improve the existing algorithms when the keys are from a restricted set?

In the following we assume that the keys are from $\{0, 1, \ldots, u - 1\}$, where u denotes the size of the universe.

Implementation 1: Array



one array of u bits

Use an array that encodes the indicator function of the dynamic set.

Implementation 1: Array

Algorithm 1 array.insert(x)

1: content[x] $\leftarrow 1$;

Algorithm 2 array.delete(x)

1: content[x] $\leftarrow 0$;

Algorithm 3 array.member(x)

1: **return** content[x];

- Note that we assume that x is valid, i.e., it falls within the array boundaries.
- Obviously(?) the running time is constant.

Implementation 1: Array

Algorithm 4 array.max()

```
for (i = size − 1; i ≥ 0; i−−) do
    if content[i] = 1 then return i;
return null;
```

Algorithm 5 array.min()

```
for (i = 0; i < size; i++) do
    if content[i] = 1 then return i;
return null;
```

- Running time is $\mathcal{O}(u)$ in the worst case.

Implementation 1: Array

Algorithm 4 array.max()

```
for (i = size − 1; i ≥ 0; i−−) do
    if content[i] = 1 then return i;
return null;
```

Algorithm 5 array.min()

```
for (i = 0; i < size; i++) do
    if content[i] = 1 then return i;
return null;
```

- Running time is $\mathcal{O}(u)$ in the worst case.

Implementation 1: Array

Algorithm 4 array.max()

for (i = size -1; $i \geq 0$; i--) **do**
    if content[i] $= 1$ **then return** i;
return null;

Algorithm 5 array.min()

for ($i = 0$; $i <$ size; i++) **do**
    if content[i] $= 1$ **then return** i;
return null;

- Running time is $\mathcal{O}(u)$ in the worst case.

Implementation 1: Array

Algorithm 6 array.succ(x)

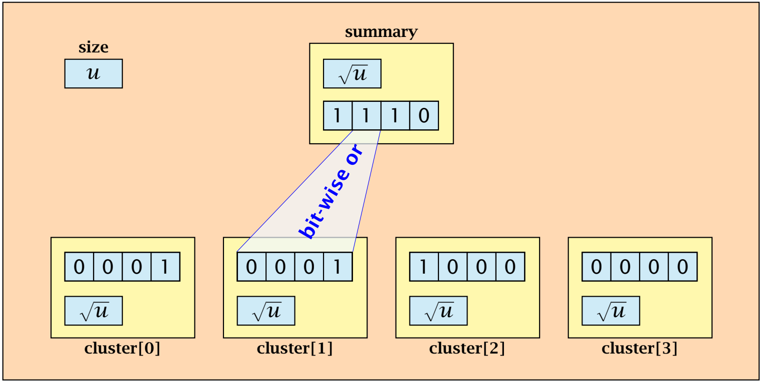
```
for (i = x + 1; i < size; i++) do
    if content[i] = 1 then return i;
return null;
```

Algorithm 7 array.pred(x)

```
for (i = x − 1; i ≥ 0; i−−) do
    if content[i] = 1 then return i;
return null;
```

- Running time is $\mathcal{O}(u)$ in the worst case.

Implementation 2: Summary Array



- $\sqrt{u}$ cluster-arrays of $\sqrt{u}$ bits.
- One summary-array of $\sqrt{u}$ bits. The i-th bit in the summary array stores the bit-wise or of the bits in the i-th cluster.

Implementation 2: Summary Array

The bit for a key x is contained in cluster number $\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$.

Within the cluster-array the bit is at position $x \bmod \sqrt{u}$.

For simplicity we assume that $u = 2^{2k}$ for some $k \geq 1$. Then we can compute the cluster-number for an entry x as $\mathrm{high}(x)$ (the upper half of the dual representation of x) and the position of x within its cluster as $\mathrm{low}(x)$ (the lower half of the dual representation).

Implementation 2: Summary Array

The bit for a key x is contained in cluster number $\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$.

Within the cluster-array the bit is at position $x \bmod \sqrt{u}$.

For simplicity we assume that $u = 2^{2k}$ for some $k \geq 1$. Then we can compute the cluster-number for an entry x as $\mathrm{high}(x)$ (the upper half of the dual representation of x) and the position of x within its cluster as $\mathrm{low}(x)$ (the lower half of the dual representation).

Implementation 2: Summary Array

The bit for a key x is contained in cluster number $\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$.

Within the cluster-array the bit is at position $x \bmod \sqrt{u}$.

For simplicity we assume that $u = 2^{2k}$ for some $k \geq 1$. Then we can compute the cluster-number for an entry x as $\mathrm{high}(x)$ (the upper half of the dual representation of x) and the position of x within its cluster as $\mathrm{low}(x)$ (the lower half of the dual representation).

Implementation 2: Summary Array

The bit for a key x is contained in cluster number $\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$.

Within the cluster-array the bit is at position $x \bmod \sqrt{u}$.

For simplicity we assume that $u = 2^{2k}$ for some $k \geq 1$. Then we can compute the cluster-number for an entry x as $\mathrm{high}(x)$ (the upper half of the dual representation of x) and the position of x within its cluster as $\mathrm{low}(x)$ (the lower half of the dual representation).

Implementation 2: Summary Array

Algorithm 8 member(x)

1: **return** cluster[high(x)]. member(low(x));

Algorithm 9 insert(x)

1: cluster[high(x)]. insert(low(x));
2: summary . insert(high(x));

- The running times are constant, because the corresponding array-functions have constant running times.

Implementation 2: Summary Array

Algorithm 8 member(x)

```
return cluster[high(x)].member(low(x));
```

Algorithm 9 insert(x)

```
cluster[high(x)].insert(low(x));
summary.insert(high(x));
```

- The running times are constant, because the corresponding array-functions have constant running times.

Implementation 2: Summary Array

Algorithm 8 member(x)

```
return cluster[high(x)].member(low(x));
```

Algorithm 9 insert(x)

```
cluster[high(x)].insert(low(x));
summary.insert(high(x));
```

- The running times are constant, because the corresponding array-functions have constant running times.

Implementation 2: Summary Array

Algorithm 10 delete(x)

```
cluster[high(x)].delete(low(x));
if cluster[high(x)].min() = null then
    summary.delete(high(x));
```

- The running time is dominated by the cost of a minimum computation on an array of size $\sqrt{u}$. Hence, $\mathcal{O}(\sqrt{u})$.

Implementation 2: Summary Array

Algorithm 10 delete(x)

```
if cluster[high(x)].min() = null then
    summary.delete(high(x));
```

- The running time is dominated by the cost of a minimum computation on an array of size $\sqrt{u}$. Hence, $\mathcal{O}(\sqrt{u})$.

Implementation 2: Summary Array

Algorithm 11 max()

$maxcluster \leftarrow$ summary.max();
if $maxcluster$ = null **return** null;
$offs \leftarrow$ cluster[$maxcluster$].max()
return $maxcluster \circ offs$;

Algorithm 12 min()

$mincluster \leftarrow$ summary.min();
if $mincluster$ = null **return** null;
$offs \leftarrow$ cluster[$mincluster$].min();
return $mincluster \circ offs$;

- Running time is roughly $2\sqrt{u} = \mathcal{O}(\sqrt{u})$ in the worst case.

Implementation 2: Summary Array

Algorithm 11 max()

```
maxcluster ← summary.max();
if maxcluster = null return null;
offs ← cluster[maxcluster].max()
return maxcluster ∘ offs;
```

Algorithm 12 min()

```
mincluster ← summary.min();
if mincluster = null return null;
offs ← cluster[mincluster].min();
return mincluster ∘ offs;
```

- Running time is roughly $2\sqrt{u} = \mathcal{O}(\sqrt{u})$ in the worst case.

Implementation 2: Summary Array

Algorithm 11 max()

```
maxcluster ← summary.max();
if maxcluster = null return null;
offs ← cluster[maxcluster].max()
return maxcluster ∘ offs;
```

Algorithm 12 min()

```
mincluster ← summary.min();
if mincluster = null return null;
offs ← cluster[mincluster].min();
return mincluster ∘ offs;
```

- Running time is roughly $2\sqrt{u} = \mathcal{O}(\sqrt{u})$ in the worst case.

Implementation 2: Summary Array

Algorithm 13 succ(x)

$m \leftarrow \text{cluster}[\text{high}(\text{x})].\text{succ}(\text{low}(x))$
if $m \neq \text{null}$ **then return** $\text{high}(x) \circ m$;
$\textit{succcluster} \leftarrow \text{summary}.\text{succ}(\text{high}(x))$;
if $\textit{succcluster} \neq \text{null}$ **then**
    $\textit{offs} \leftarrow \text{cluster}[\textit{succcluster}].\text{min}()$;
    return $\textit{succcluster} \circ \textit{offs}$;
return null;

- Running time is roughly $3\sqrt{u} = \mathcal{O}(\sqrt{u})$ in the worst case.

Implementation 2: Summary Array

Algorithm 13 succ(x)

1: $m \leftarrow$ cluster[high(x)]. succ(low(x))
2: **if** $m \neq$ null **then return** high(x) $\circ$ m;
3: *succcluster* $\leftarrow$ summary . succ(high(x));
4: **if** *succcluster* $\neq$ null **then**
5:     *offs* $\leftarrow$ cluster[*succcluster*]. min();
6:     **return** *succcluster* $\circ$ *offs*;
7: **return** null;

- Running time is roughly $3\sqrt{u} = \mathcal{O}(\sqrt{u})$ in the worst case.

Implementation 2: Summary Array

Algorithm 14 pred(x)

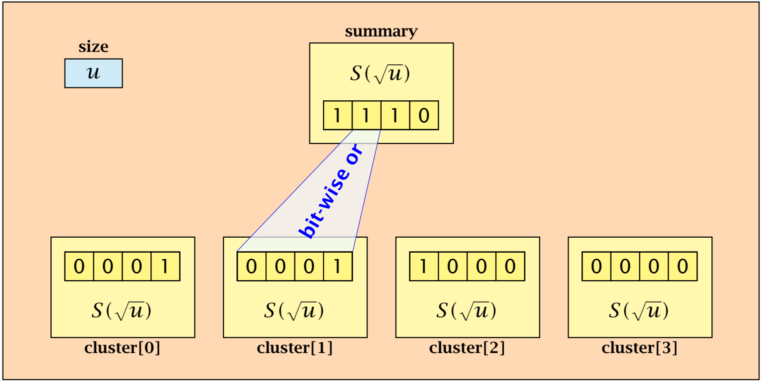
```
m ← cluster[high(x)].pred(low(x))
if m ≠ null then return high(x) ∘ m;
predcluster ← summary.pred(high(x));
if predcluster ≠ null then
        offs ← cluster[predcluster].max();
        return predcluster ∘ offs;
return null;
```

- Running time is roughly $3\sqrt{u} = \mathcal{O}(\sqrt{u})$ in the worst case.

Implementation 3: Recursion

Instead of using sub-arrays, we build a recursive data-structure.

$S(u)$ is a dynamic set data-structure representing u bits:



Implementation 3: Recursion

We assume that $u = 2^{2^k}$ for some k.

The data-structure $S(2)$ is defined as an array of 2-bits (end of the recursion).

Implementation 3: Recursion

The code from Implementation 2 can be used unchanged. We only need to redo the analysis of the running time.

Note that in the code we do not need to specifically address the non-recursive case. This is achieved by the fact that an $S(4)$ will contain $S(2)$'s as sub-datastructures, which are arrays. Hence, a call like $\text{cluster}[1].\text{min}()$ from within the data-structure $S(4)$ is not a recursive call as it will call the function $\text{array}.\text{min}()$.

This means that the non-recursive case is been dealt with while initializing the data-structure.

Implementation 3: Recursion

The code from Implementation 2 can be used unchanged. We only need to redo the analysis of the running time.

Note that in the code we do not need to specifically address the non-recursive case. This is achieved by the fact that an $S(4)$ will contain $S(2)$'s as sub-datastructures, which are arrays. Hence, a call like $\mathrm{cluster}[1].\mathrm{min}()$ from within the data-structure $S(4)$ is not a recursive call as it will call the function $\mathrm{array}.\mathrm{min}()$.

This means that the non-recursive case is been dealt with while initializing the data-structure.

Implementation 3: Recursion

The code from Implementation 2 can be used unchanged. We only need to redo the analysis of the running time.

Note that in the code we do not need to specifically address the non-recursive case. This is achieved by the fact that an $S(4)$ will contain $S(2)$'s as sub-datastructures, which are arrays. Hence, a call like $\text{cluster}[1].\text{min}()$ from within the data-structure $S(4)$ is not a recursive call as it will call the function $\text{array}.\text{min}()$.

This means that the non-recursive case is been dealt with while initializing the data-structure.

Implementation 3: Recursion

The code from Implementation 2 can be used unchanged. We only need to redo the analysis of the running time.

Note that in the code we do not need to specifically address the non-recursive case. This is achieved by the fact that an $S(4)$ will contain $S(2)$'s as sub-datastructures, which are arrays. Hence, a call like $\text{cluster}[1].\text{min}()$ from within the data-structure $S(4)$ is not a recursive call as it will call the function $\text{array}.\text{min}()$.

This means that the non-recursive case is been dealt with while initializing the data-structure.

Implementation 3: Recursion

Algorithm 15 member(x)

```
1: return cluster[high(x)].member(low(x));
```

- $T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1$.

Implementation 3: Recursion

Algorithm 16 insert(x)

```
cluster[high(x)].insert(low(x));
summary.insert(high(x));
```

- $T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1$.

Implementation 3: Recursion

Algorithm 17 delete(x)

```
cluster[high(x)].delete(low(x));
if cluster[high(x)].min() = null then
    summary.delete(high(x));
```

- $T_{\mathrm{del}}(u) = 2T_{\mathrm{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1$.

Implementation 3: Recursion

Algorithm 18 min()

1: *mincluster* $\leftarrow$ summary.min();
2: **if** *mincluster* $=$ null **return** null;
3: *offs* $\leftarrow$ cluster[*mincluster*].min();
4: **return** *mincluster* $\circ$ *offs*;

- $T_{\min}(u) = 2T_{\min}(\sqrt{u}) + 1$.

Implementation 3: Recursion

Algorithm 19 succ(x)

1: $m \leftarrow$ cluster[high(x)]. succ(low(x))
2: **if** $m \neq$ null **then return** high(x) $\circ$ m;
3: *succcluster* $\leftarrow$ summary . succ(high(x));
4: **if** *succcluster* $\neq$ null **then**
5:     *offs* $\leftarrow$ cluster[*succcluster*]. min();
6:     **return** *succcluster* $\circ$ *offs*;
7: **return** null;

- $T_{\text{succ}}(u) = 2T_{\text{succ}}(\sqrt{u}) + T_{\text{min}}(\sqrt{u}) + 1$.

Implementation 3: Recursion

$$T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1:$$

Implementation 3: Recursion

$\boldsymbol{T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1}$**:**

Set $\ell := \log u$ and $X(\ell) := T_{\text{mem}}(2^{\ell})$.

Implementation 3: Recursion

$\boldsymbol{T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1}$:

Set $\ell := \log u$ and $X(\ell) := T_{\text{mem}}(2^{\ell})$. Then

Implementation 3: Recursion

$\boldsymbol{T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1}$**:**

Set $\ell := \log u$ and $X(\ell) := T_{\text{mem}}(2^\ell)$. Then

$$X(\ell)$$

Implementation 3: Recursion

$\boldsymbol{T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1}$**:**

Set $\ell := \log u$ and $X(\ell) := T_{\text{mem}}(2^\ell)$. Then

$$X(\ell) = T_{\text{mem}}(2^\ell)$$

Implementation 3: Recursion

$\boldsymbol{T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1}$**:**

Set $\ell := \log u$ and $X(\ell) := T_{\text{mem}}(2^\ell)$. Then

$$X(\ell) = T_{\text{mem}}(2^\ell) = T_{\text{mem}}(u)$$

Implementation 3: Recursion

$\boldsymbol{T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1}$**:**

Set $\ell := \log u$ and $X(\ell) := T_{\text{mem}}(2^\ell)$.Then

$$X(\ell) = T_{\text{mem}}(2^\ell) = T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1$$

Implementation 3: Recursion

$\boldsymbol{T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1}$**:**

Set $\ell := \log u$ and $X(\ell) := T_{\text{mem}}(2^\ell)$. Then

$$\begin{aligned} X(\ell) = T_{\text{mem}}(2^\ell) = T_{\text{mem}}(u) &= T_{\text{mem}}(\sqrt{u}) + 1 \\ &= T_{\text{mem}}(2^{\frac{\ell}{2}}) + 1 \end{aligned}$$

Implementation 3: Recursion

$T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1$:

Set $\ell := \log u$ and $X(\ell) := T_{\text{mem}}(2^\ell)$. Then

$$\begin{aligned} X(\ell) = T_{\text{mem}}(2^\ell) = T_{\text{mem}}(u) &= T_{\text{mem}}(\sqrt{u}) + 1 \\ &= T_{\text{mem}}(2^{\frac{\ell}{2}}) + 1 = X(\tfrac{\ell}{2}) + 1 \enspace . \end{aligned}$$

Implementation 3: Recursion

$\boldsymbol{T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1}$:

Set $\ell := \log u$ and $X(\ell) := T_{\text{mem}}(2^\ell)$. Then

$$X(\ell) = T_{\text{mem}}(2^\ell) = T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1$$
$$= T_{\text{mem}}(2^{\frac{\ell}{2}}) + 1 = X(\tfrac{\ell}{2}) + 1 \ .$$

Using Master theorem gives $X(\ell) = \mathcal{O}(\log \ell)$, and hence $T_{\text{mem}}(u) = \mathcal{O}(\log\log u)$.

Implementation 3: Recursion

$$T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.$$

Implementation 3: Recursion

$$\boldsymbol{T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.}$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^{\ell})$.

Implementation 3: Recursion

$$T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^\ell)$. Then

Implementation 3: Recursion

$$T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1 .$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^\ell)$. Then

$$X(\ell)$$

Implementation 3: Recursion

$$T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^\ell)$. Then

$$X(\ell) = T_{\text{ins}}(2^\ell)$$

Implementation 3: Recursion

$$\boldsymbol{T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.}$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^\ell)$. Then

$$X(\ell) = T_{\text{ins}}(2^\ell) = T_{\text{ins}}(u)$$

Implementation 3: Recursion

$$\boldsymbol{T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.}$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^\ell)$. Then

$$X(\ell) = T_{\text{ins}}(2^\ell) = T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1$$

Implementation 3: Recursion

$$\boldsymbol{T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.}$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^{\ell})$. Then

$$\begin{aligned} X(\ell) = T_{\text{ins}}(2^{\ell}) = T_{\text{ins}}(u) &= 2T_{\text{ins}}(\sqrt{u}) + 1 \\ &= 2T_{\text{ins}}(2^{\frac{\ell}{2}}) + 1 \end{aligned}$$

Implementation 3: Recursion

$$T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^\ell)$. Then

$$\begin{aligned} X(\ell) = T_{\text{ins}}(2^\ell) = T_{\text{ins}}(u) &= 2T_{\text{ins}}(\sqrt{u}) + 1 \\ &= 2T_{\text{ins}}(2^{\frac{\ell}{2}}) + 1 = 2X(\tfrac{\ell}{2}) + 1 \ . \end{aligned}$$

Implementation 3: Recursion

$$\boldsymbol{T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.}$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^\ell)$. Then

$$\begin{aligned} X(\ell) = T_{\text{ins}}(2^\ell) = T_{\text{ins}}(u) &= 2T_{\text{ins}}(\sqrt{u}) + 1 \\ &= 2T_{\text{ins}}(2^{\frac{\ell}{2}}) + 1 = 2X(\tfrac{\ell}{2}) + 1 \ . \end{aligned}$$

Using Master theorem gives $X(\ell) = \mathcal{O}(\ell)$, and hence $T_{\text{ins}}(u) = \mathcal{O}(\log u)$.

Implementation 3: Recursion

$$T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^\ell)$. Then

$$\begin{aligned} X(\ell) = T_{\text{ins}}(2^\ell) = T_{\text{ins}}(u) &= 2T_{\text{ins}}(\sqrt{u}) + 1 \\ &= 2T_{\text{ins}}(2^{\frac{\ell}{2}}) + 1 = 2X(\tfrac{\ell}{2}) + 1 \ . \end{aligned}$$

Using Master theorem gives $X(\ell) = \mathcal{O}(\ell)$, and hence $T_{\text{ins}}(u) = \mathcal{O}(\log u)$.

The same holds for $T_{\max}(u)$ and $T_{\min}(u)$.

Implementation 3: Recursion

$$T_{\mathrm{del}}(u) = 2T_{\mathrm{del}}(\sqrt{u}) + T_{\mathrm{min}}(\sqrt{u}) + 1 \leq 2T_{\mathrm{del}}(\sqrt{u}) + c\log(u).$$

Implementation 3: Recursion

$$T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 \leq 2T_{\text{del}}(\sqrt{u}) + c\log(u).$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{del}}(2^{\ell})$.

Implementation 3: Recursion

$$T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 \leq 2T_{\text{del}}(\sqrt{u}) + c\log(u).$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{del}}(2^{\ell})$. Then

Implementation 3: Recursion

$$T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 \le 2T_{\text{del}}(\sqrt{u}) + c\log(u).$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{del}}(2^\ell)$. Then

$$X(\ell)$$

Implementation 3: Recursion

$$T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 \leq 2T_{\text{del}}(\sqrt{u}) + c\log(u).$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{del}}(2^\ell)$. Then

$$X(\ell) = T_{\text{del}}(2^\ell)$$

Implementation 3: Recursion

$$T_{\mathrm{del}}(u) = 2T_{\mathrm{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 \leq 2T_{\mathrm{del}}(\sqrt{u}) + c\log(u).$$

Set $\ell := \log u$ and $X(\ell) := T_{\mathrm{del}}(2^\ell)$. Then

$$X(\ell) = T_{\mathrm{del}}(2^\ell) = T_{\mathrm{del}}(u)$$

Implementation 3: Recursion

$$T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 \leq 2T_{\text{del}}(\sqrt{u}) + c\log(u).$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{del}}(2^\ell)$. Then

$$X(\ell) = T_{\text{del}}(2^\ell) = T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + c\log u$$

Implementation 3: Recursion

$$T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 \leq 2T_{\text{del}}(\sqrt{u}) + c\log(u).$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{del}}(2^\ell)$. Then

$$\begin{aligned} X(\ell) = T_{\text{del}}(2^\ell) = T_{\text{del}}(u) &= 2T_{\text{del}}(\sqrt{u}) + c\log u \\ &= 2T_{\text{del}}(2^{\frac{\ell}{2}}) + c\ell \end{aligned}$$

Implementation 3: Recursion

$$T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 \leq 2T_{\text{del}}(\sqrt{u}) + c\log(u).$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{del}}(2^\ell)$. Then

$$\begin{aligned} X(\ell) = T_{\text{del}}(2^\ell) = T_{\text{del}}(u) &= 2T_{\text{del}}(\sqrt{u}) + c\log u \\ &= 2T_{\text{del}}(2^{\frac{\ell}{2}}) + c\ell = 2X(\tfrac{\ell}{2}) + c\ell \ . \end{aligned}$$

Implementation 3: Recursion

$$\mathbf{T_{del}(u) = 2T_{del}(\sqrt{u}) + T_{min}(\sqrt{u}) + 1 \le 2T_{del}(\sqrt{u}) + c\log(u).}$$

Set $\ell := \log u$ and $X(\ell) := T_{del}(2^\ell)$. Then

$$\begin{aligned} X(\ell) = T_{del}(2^\ell) = T_{del}(u) &= 2T_{del}(\sqrt{u}) + c\log u \\ &= 2T_{del}(2^{\frac{\ell}{2}}) + c\ell = 2X(\tfrac{\ell}{2}) + c\ell \ . \end{aligned}$$

Using Master theorem gives $X(\ell) = \Theta(\ell \log \ell)$, and hence $T_{del}(u) = \mathcal{O}(\log u \log\log u)$.

Implementation 3: Recursion

$$T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 \leq 2T_{\text{del}}(\sqrt{u}) + c\log(u).$$

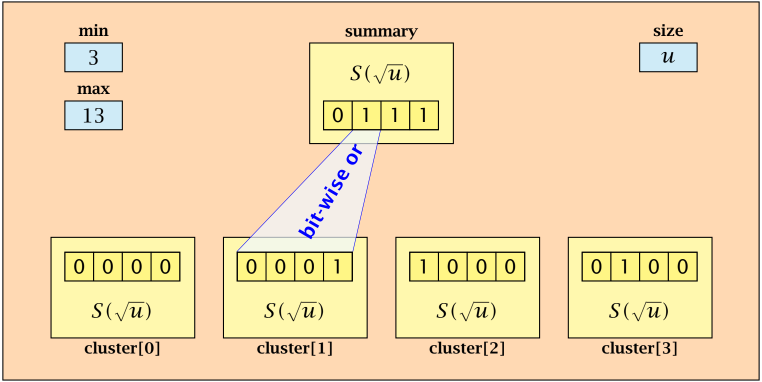
Set $\ell := \log u$ and $X(\ell) := T_{\text{del}}(2^\ell)$. Then

$$\begin{aligned} X(\ell) = T_{\text{del}}(2^\ell) = T_{\text{del}}(u) &= 2T_{\text{del}}(\sqrt{u}) + c\log u \\ &= 2T_{\text{del}}(2^{\frac{\ell}{2}}) + c\ell = 2X(\tfrac{\ell}{2}) + c\ell \ . \end{aligned}$$

Using Master theorem gives $X(\ell) = \Theta(\ell \log \ell)$, and hence $T_{\text{del}}(u) = \mathcal{O}(\log u \log\log u)$.

The same holds for $T_{\text{pred}}(u)$ and $T_{\text{succ}}(u)$.

Implementation 4: van Emde Boas Trees



- The bit referenced by min is not set within sub-datastructures.
- The bit referenced by max is set within sub-datastructures (if $\max \neq \min$).

Implementation 4: van Emde Boas Trees

Advantages of having max/min pointers:

- Recursive calls for min and max are constant time.
- $\mathrm{min} = \mathrm{null}$ means that the data-structure is empty.
- $\mathrm{min} = \mathrm{max} \neq \mathrm{null}$ means that the data-structure contains exactly one element.
- We can insert into an empty datastructure in constant time by only setting $\mathrm{min} = \mathrm{max} = x$.
- We can delete from a data-structure that just contains one element in constant time by setting $\mathrm{min} = \mathrm{max} = \mathrm{null}$.

Implementation 4: van Emde Boas Trees

Advantages of having max/min pointers:

- Recursive calls for $\min$ and $\max$ are constant time.
- $\min = \text{null}$ means that the data-structure is empty.
- $\min = \max \neq \text{null}$ means that the data-structure contains exactly one element.
- We can insert into an empty datastructure in constant time by only setting $\min = \max = x$.
- We can delete from a data-structure that just contains one element in constant time by setting $\min = \max = \text{null}$.

Implementation 4: van Emde Boas Trees

Advantages of having max/min pointers:

- Recursive calls for $\min$ and $\max$ are constant time.
- $\min = \text{null}$ means that the data-structure is empty.
- $\min = \max \neq \text{null}$ means that the data-structure contains exactly one element.
- We can insert into an empty datastructure in constant time by only setting $\min = \max = x$.
- We can delete from a data-structure that just contains one element in constant time by setting $\min = \max = \text{null}$.

Implementation 4: van Emde Boas Trees

Advantages of having max/min pointers:

- Recursive calls for $\min$ and $\max$ are constant time.
- $\min = \text{null}$ means that the data-structure is empty.
- $\min = \max \neq \text{null}$ means that the data-structure contains exactly one element.
- We can insert into an empty datastructure in constant time by only setting $\min = \max = x$.
- We can delete from a data-structure that just contains one element in constant time by setting $\min = \max = \text{null}$.

Implementation 4: van Emde Boas Trees

Advantages of having max/min pointers:

- Recursive calls for min and max are constant time.
- $\text{min} = \text{null}$ means that the data-structure is empty.
- $\text{min} = \text{max} \neq \text{null}$ means that the data-structure contains exactly one element.
- We can insert into an empty datastructure in constant time by only setting $\text{min} = \text{max} = x$.
- We can delete from a data-structure that just contains one element in constant time by setting $\text{min} = \text{max} = \text{null}$.

Implementation 4: van Emde Boas Trees

Advantages of having max/min pointers:

- Recursive calls for $\min$ and $\max$ are constant time.
- $\min = \text{null}$ means that the data-structure is empty.
- $\min = \max \neq \text{null}$ means that the data-structure contains exactly one element.
- We can insert into an empty datastructure in constant time by only setting $\min = \max = x$.
- We can delete from a data-structure that just contains one element in constant time by setting $\min = \max = \text{null}$.

Implementation 4: van Emde Boas Trees

Algorithm 20 max()

return max;

Algorithm 21 min()

return min;

- Constant time.

Implementation 4: van Emde Boas Trees

Algorithm 22 member(x)

```
1: if x = min then return 1; // TRUE
2: return cluster[high(x)].member(low(x));
```

- $T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1 \Longrightarrow T(u) = \mathcal{O}(\log\log u)$.

Implementation 4: van Emde Boas Trees

Algorithm 23 succ(x)

if $\text{min} \neq \text{null} \wedge x < \text{min}$ **then return** min;
maxincluster ← cluster[high(x)].max();
if *maxincluster* $\neq$ null $\wedge$ low(x) < *maxincluster* **then**
  offs ← cluster[high(x)].succ(low(x));
  return high$(x) \circ$ *offs*;
else
  succcluster ← summary.succ(high(x));
  if *succcluster* = null **then return** null;
  offs ← cluster[*succcluster*].min();
  return *succcluster* $\circ$ *offs*;

- $T_{\text{succ}}(u) = T_{\text{succ}}(\sqrt{u}) + 1 \Longrightarrow T_{\text{succ}}(u) = \mathcal{O}(\log\log u)$.

Implementation 4: van Emde Boas Trees

Algorithm 44 insert(x)

```
if min = null then
    min = x; max = x;
else
    if x < min then exchange x and min;
    if cluster[high(x)].min = null; then
        summary.insert(high(x));
        cluster[high(x)].insert(low(x));
    else
        cluster[high(x)].insert(low(x));
    if x > max then max = x;
```

- $T_{\text{ins}}(u) = T_{\text{ins}}(\sqrt{u}) + 1 \Longrightarrow T_{\text{ins}}(u) = \mathcal{O}(\log\log u)$.

Implementation 4: van Emde Boas Trees

Note that the recusive call in Line 7 takes constant time as the if-condition in Line 5 ensures that we are inserting in an empty sub-tree.

The only non-constant recursive calls are the call in Line 6 and in Line 9. These are mutually exclusive, i.e., only one of these calls will actually occur.

From this we get that $T_{\text{ins}}(u) = T_{\text{ins}}(\sqrt{u}) + 1$.

Implementation 4: van Emde Boas Trees

- **Assumes that x is contained in the structure.**

Algorithm 45 delete(x)

if min $=$ max **then**
  min $=$ null; max $=$ null;
else
  if $x =$ min **then**
    firstcluster $\leftarrow$ summary.min();
    offs $\leftarrow$ cluster[*firstcluster*].min();
    $x \leftarrow$ *firstcluster* $\circ$ *offs*;
    min $\leftarrow x$;
  cluster[high(x)].delete(low(x));

continued...

Implementation 4: van Emde Boas Trees

- **Assumes that x is contained in the structure.**

Algorithm 45 delete(x)

```
if min = max then
    min = null; max = null;
else
    if x = min then                                find new minimum
        firstcluster ← summary.min();
        offs ← cluster[firstcluster].min();
        x ← firstcluster ∘ offs;
        min ← x;
    cluster[high(x)].delete(low(x));
```

continued...

Implementation 4: van Emde Boas Trees

- Assumes that x is contained in the structure.

Algorithm 45 delete(x)

```
if min = max then
    min = null; max = null;
else
    if x = min then
        firstcluster ← summary.min();
        offs ← cluster[firstcluster].min();
        x ← firstcluster ∘ offs;
        min ← x;
    cluster[high(x)].delete(low(x));        delete
```

continued...

Implementation 4: van Emde Boas Trees

Algorithm 45 delete(x)

...continued

```
if cluster[high(x)].min() = null then
    summary.delete(high(x));
    if x = max then
        summax ← summary.max();
        if summax = null then max ← min;
        else
            offs ← cluster[summax].max();
            max ← summax ∘ offs
else
    if x = max then
        offs ← cluster[high(x)].max();
        max ← high(x) ∘ offs;
```

Implementation 4: van Emde Boas Trees

Algorithm 45 delete(x)

...continued — fix maximum

```
if cluster[high(x)].min() = null then
    summary.delete(high(x));
    if x = max then
        summax ← summary.max();
        if summax = null then max ← min;
        else
            offs ← cluster[summax].max();
            max ← summax ∘ offs
else
    if x = max then
        offs ← cluster[high(x)].max();
        max ← high(x) ∘ offs;
```

Implementation 4: van Emde Boas Trees

Note that only one of the possible recusive calls in Line 9 and Line 11 in the deletion-algorithm may take non-constant time.

To see this observe that the call in Line 11 only occurs if the cluster where x was deleted is now empty. But this means that the call in Line 9 deleted the last element in $\text{cluster}[\text{high}(x)]$. Such a call only takes constant time.

Hence, we get a recurrence of the form

$$T_{\text{del}}(u) = T_{\text{del}}(\sqrt{u}) + c \ .$$

This gives $T_{\text{del}}(u) = \mathcal{O}(\log\log u)$.

10 van Emde Boas Trees

Space requirements:

- The space requirement fulfills the recurrence

$$S(u) = (\sqrt{u} + 1)S(\sqrt{u}) + \mathcal{O}(\sqrt{u}) \ .$$

- Note that we cannot solve this recurrence by the Master theorem as the branching factor is not constant.
- One can show by induction that the space requirement is $S(u) = \mathcal{O}(u)$. Exercise.

- Let the “real” recurrence relation be

$$S(k^2) = (k+1)S(k) + c_1 \cdot k;\ S(4) = c_2$$

- Replacing $S(k)$ by $R(k) := S(k)/c_2$ gives the recurrence

$$R(k^2) = (k+1)R(k) + ck;\ R(4) = 1$$

  where $c = c_1/c_2 < 1$.
- Now, we show $R(k) \le k - 2$ for squares $k \ge 4$.
  - Obviously, this holds for $k = 4$.
  - For $k = \ell^2 > 4$ with ℓ integral we have

$$\begin{aligned} R(k) &= (1+\ell)R(\ell) + c\ell \\ &\le (1+\ell)(\ell-2) + \ell \le k-2 \end{aligned}$$

  - This shows that $R(k)$ and, hence, $S(k)$ grows linearly.