5.8 van Emde Boas Trees

Dynamic Set Data Structure S:

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

5.8 van Emde Boas Trees

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

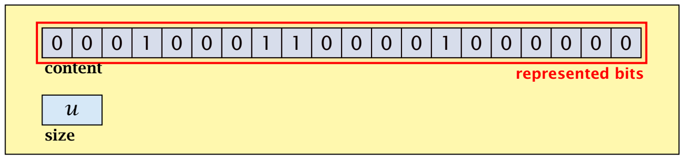
- $\boldsymbol{S}.\mathbf{insert}(\boldsymbol{x})$**:** Inserts x into S.
- $\boldsymbol{S}.\mathbf{delete}(\boldsymbol{x})$**:** Deletes x from S. Usually assumes that $x \in S$.
- $\boldsymbol{S}.\mathbf{member}(\boldsymbol{x})$**:** Returns 1 if $x \in S$ and 0 otw.
- $\boldsymbol{S}.\mathbf{min}()$**:** Returns the value of the minimum element in S.
- $\boldsymbol{S}.\mathbf{max}()$**:** Returns the value of the maximum element in S.
- $\boldsymbol{S}.\mathbf{succ}(\boldsymbol{x})$**:** 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.
- $\boldsymbol{S}.\mathbf{pred}(\boldsymbol{x})$**:** 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.

5.8 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)

content[x] $\leftarrow 1$;

Algorithm 2 array.delete(x)

content[x] $\leftarrow 0$;

Algorithm 3 array.member(x)

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; $i \geq 0$; 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)

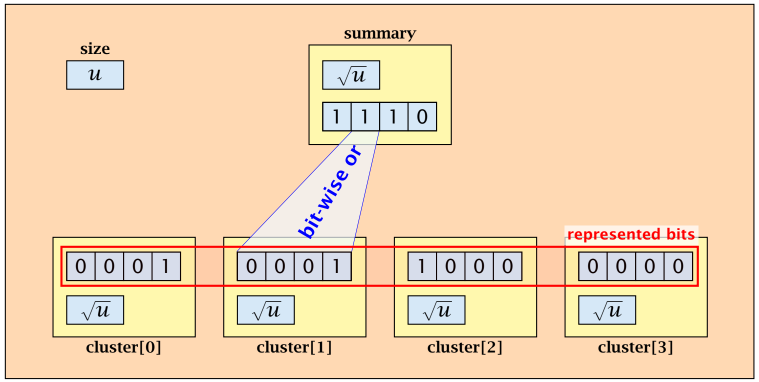
for ($i = x + 1$; $i <$ size; i++) **do**
    if content[i] $= 1$ **then return** i;
return null;

Algorithm 7 array.pred(x)

for ($i = x - 1$; $i \geq 0$; 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

Algorithm 8 member(x)

return $\mathrm{cluster}[\mathrm{high}(x)].\mathrm{member}(\mathrm{low}(x))$;

Algorithm 9 insert(x)

$\mathrm{cluster}[\mathrm{high}(x)].\mathrm{insert}(\mathrm{low}(x))$;
$\mathrm{summary}.\mathrm{insert}(\mathrm{high}(x))$;

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

Implementation 2: Summary Array

Algorithm 10 delete(x)

$\mathrm{cluster}[\mathrm{high}(x)].\mathrm{delete}(\mathrm{low}(x))$;
if $\mathrm{cluster}[\mathrm{high}(x)].\mathrm{min}() = \mathrm{null}$ **then**
    $\mathrm{summary}.\mathrm{delete}(\mathrm{high}(x))$;

- 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()

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

Algorithm 12 min()

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

The operator $\circ$ stands for the concatenation of two bitstrings. This means if $x = 0111_2$ and $y = 0001_2$ then $x \circ y = 01110001_2$.

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

Implementation 2: Summary Array

Algorithm 13 succ(x)

```
m ← cluster[high(x)].succ(low(x))
if m ≠ null then return high(x) ∘ m;
succcluster ← summary.succ(high(x));
if succcluster ≠ null then
    offs ← cluster[succcluster].min();
    return succcluster ∘ offs;
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)

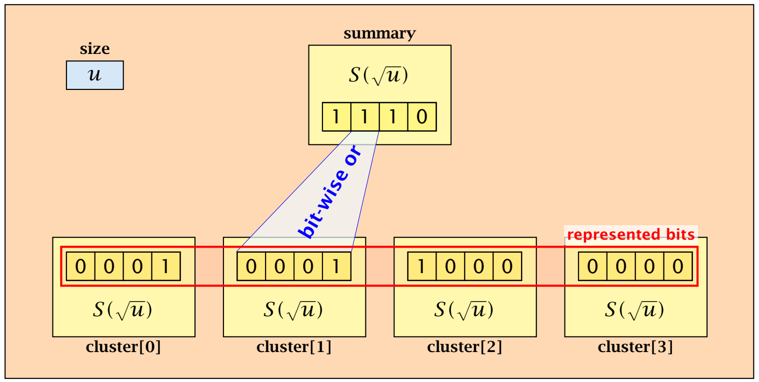
```
m ← cluster[high(x)].pred(low(x))
if m ≠ null then return high(x) ∘ m;
predcluster ← summary.pred(high(x));
if predcluster ≠ null then
    offs ← cluster[predcluster].max();
    return predcluster ∘ offs;
return null;
```

- 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

Algorithm 15 member(x)

1: **return** $\text{cluster}[\text{high}(x)].\text{member}(\text{low}(x))$;

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

Implementation 3: Recursion

Algorithm 16 insert(x)

1: $\text{cluster}[\text{high}(x)].\text{insert}(\text{low}(x))$;
2: $\text{summary}.\text{insert}(\text{high}(x))$;

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

Implementation 3: Recursion

Algorithm 17 delete(x)

1: $\text{cluster}[\text{high}(x)].\text{delete}(\text{low}(x))$;
2: **if** $\text{cluster}[\text{high}(x)].\text{min}() = \text{null}$ **then**
3: $\quad$ $\text{summary}.\text{delete}(\text{high}(x))$;

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

Implementation 3: Recursion

Algorithm 18 min()

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

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

Implementation 3: Recursion

Algorithm 19 succ(x)

```
m ← cluster[high(x)].succ(low(x))
if m ≠ null then return high(x) ∘ m;
succcluster ← summary.succ(high(x));
if succcluster ≠ null then
    offs ← cluster[succcluster].min();
    return succcluster ∘ offs;
return null;
```

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

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

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

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

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

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

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

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

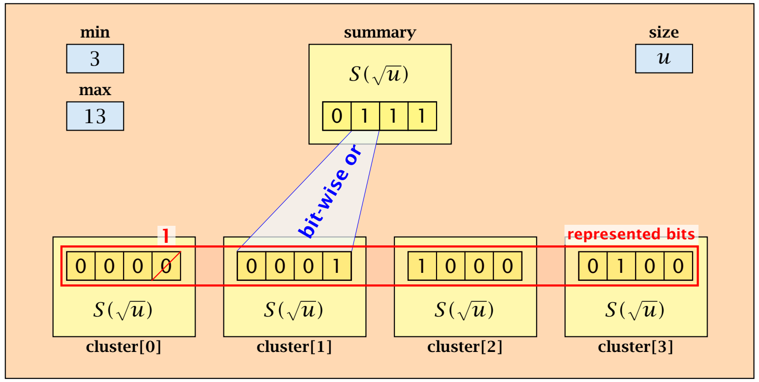
Set $\ell := \log u$ and $X(\ell) := T_{\text{del}}(2^{\ell})$. Then

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

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.
- min = null means that the data-structure is empty.
- min = max $\neq$ 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 = null.

Implementation 4: van Emde Boas Trees

Algorithm 20 max()
1: **return** max;

Algorithm 21 min()
1: **return** min;

- Constant time.

Implementation 4: van Emde Boas Trees

Algorithm 22 member(x)

```
if x = min then return 1; // TRUE
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 min ≠ null ∧ x < min then return min;
maxincluster ← cluster[high(x)].max();
if maxincluster ≠ null ∧ low(x) < maxincluster then
    offs ← cluster[high(x)].succ(low(x));
    return high(x) ∘ offs;
else
    succcluster ← summary.succ(high(x));
    if succcluster = null then return null;
    offs ← cluster[succcluster].min();
    return succcluster ∘ 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 35 insert(x)

```
1: if min = null then
2:     min = x; max = x;
3: else
4:     if x < min then exchange x and min;
5:     if x > max then max = x;
6:     if cluster[high(x)].min = null; then
7:         summary.insert(high(x));
8:         cluster[high(x)].insert(low(x));
9:     else
10:        cluster[high(x)].insert(low(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 8 takes constant time as the if-condition in Line 6 ensures that we are inserting in an empty sub-tree.

The only non-constant recursive calls are the call in Line 7 and in Line 10. 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 36 delete(x)

```
1: if min = max then
2:     min = max = null;
3: else
4:     if x = min then                              find new minimum
5:         firstcluster ← summary.min();
6:         offs ← cluster[firstcluster].min();
7:         x ← firstcluster ∘ offs;
8:         min ← x;
9:     cluster[high(x)].delete(low(x));             delete
                      continued...
```


Implementation 4: van Emde Boas Trees

Algorithm 36 delete(x)

```
                      ...continued                  fix maximum
10:     if cluster[high(x)].min() = null then
11:         summary.delete(high(x));
12:         if x = max then
13:             summax ← summary.max();
14:             if summax = null then max ← min;
15:             else
16:                 offs ← cluster[summax].max();
17:                 max ← summax ∘ offs
18:     else
19:         if x = max then
20:             offs ← cluster[high(x)].max();
21:             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 $\mathrm{cluster}[\mathrm{high}(x)]$. Such a call only takes constant time.

Hence, we get a recurrence of the form

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

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

5.8 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 $R(k^2) \leq k^2 - 2$ for $k^2 \geq 4$.
  - Obviously, this holds for $k^2 = 4$.
  - For $k^2 > 4$ we have

$$\begin{aligned} R(k^2) &= (1+k)R(k) + ck \\ &\leq (1+k)(k-2) + k \leq k^2 - 2 \end{aligned}$$

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

van Emde Boas Trees

Bibliography

[CLRS90] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein:
Introduction to Algorithms (3rd ed.),
MIT Press and McGraw-Hill, 2009

See Chapter 20 of [CLRS90].