

## 10 van Emde Boas Trees

### Dynamic Set Data Structure $S$ :

- ▶  $S.insert(x)$
- ▶  $S.delete(x)$
- ▶  $S.search(x)$
- ▶  $S.min()$
- ▶  $S.max()$
- ▶  $S.succ(x)$
- ▶  $S.pred(x)$

## 10 van Emde Boas Trees

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

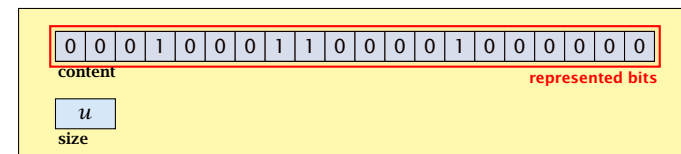
- ▶  $S.insert(x)$ : Inserts  $x$  into  $S$ .
- ▶  $S.delete(x)$ : Deletes  $x$  from  $S$ . Usually assumes that  $x \in S$ .
- ▶  $S.member(x)$ : Returns 1 if  $x \in S$  and 0 otherwise.
- ▶  $S.min()$ : Returns the value of the minimum element in  $S$ .
- ▶  $S.max()$ : Returns the value of the maximum element in  $S$ .
- ▶  $S.succ(x)$ : 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  $S$ .
- ▶  $S.pred(x)$ : 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  $\{0, 1, \dots, u-1\}$ , 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 17 array.insert( $x$ )

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

### Algorithm 18 array.delete( $x$ )

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

### Algorithm 19 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 20 array.max()

```
1: for ( $i = \text{size} - 1; i \geq 0; i--$ ) do  
2:   if content[ $i$ ] = 1 then return  $i$ ;  
3: return null;
```

### Algorithm 21 array.min()

```
1: for ( $i = 0; i < \text{size}; i++$ ) do  
2:   if content[ $i$ ] = 1 then return  $i$ ;  
3: return null;
```

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



## Implementation 1: Array

### Algorithm 22 array.succ( $x$ )

```
1: for ( $i = x + 1; i < \text{size}; i++$ ) do  
2:   if content[ $i$ ] = 1 then return  $i$ ;  
3: return null;
```

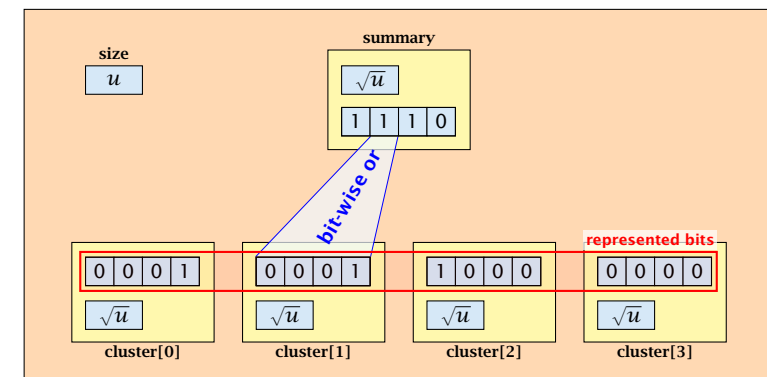
### Algorithm 23 array.pred( $x$ )

```
1: for ( $i = x - 1; i \geq 0; i--$ ) do  
2:   if content[ $i$ ] = 1 then return  $i$ ;  
3: 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  $\lfloor \frac{x}{\sqrt{u}} \rfloor$ .

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  $\text{high}(x)$  (the upper half of the dual representation of  $x$ ) and the position of  $x$  within its cluster as  $\text{low}(x)$  (the lower half of the dual representation).

## Implementation 2: Summary Array

### Algorithm 24 member( $x$ )

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

### Algorithm 25 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 26 delete( $x$ )

```
1: cluster[high( $x$ )].delete(low( $x$ ));  
2: if cluster[high( $x$ )].min() = null then  
3:   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 27 max()

```
1: maxcluster ← summary.max();  
2: if maxcluster = null return null;  
3: offs ← cluster[maxcluster].max();  
4: return maxcluster ◦ offs;
```

### Algorithm 28 min()

```
1: mincluster ← summary.min();  
2: if mincluster = null return null;  
3: offs ← cluster[mincluster].min();  
4: return mincluster ◦ offs;
```

The operator ◦ 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 29 succ(x)

```

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

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

## Implementation 2: Summary Array

### Algorithm 30 pred(x)

```

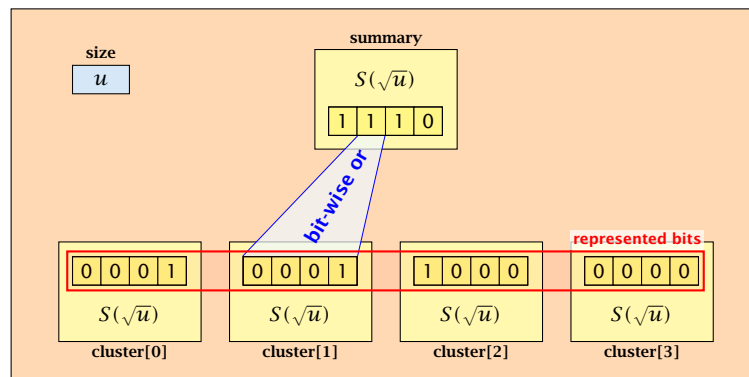
1:  $m \leftarrow \text{cluster}[\text{high}(x)].\text{pred}(\text{low}(x))$ 
2: if  $m \neq \text{null}$  then return  $\text{high}(x) \circ m$ ;
3:  $\text{predcluster} \leftarrow \text{summary}.\text{pred}(\text{high}(x))$ ;
4: if  $\text{predcluster} \neq \text{null}$  then
5:    $\text{offs} \leftarrow \text{cluster}[\text{predcluster}].\text{max}()$ ;
6:   return  $\text{predcluster} \circ \text{offs}$ ;
7: 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 `cluster[1].min()` from within the data-structure  $S(4)$  is **not** a recursive call as it will call the function `array.min()`.

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

## Implementation 3: Recursion

### Algorithm 31 `member(x)`

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

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

## Implementation 3: Recursion

### Algorithm 32 `insert(x)`

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

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

## Implementation 3: Recursion

### Algorithm 33 `delete(x)`

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

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

## Implementation 3: Recursion

### Algorithm 34 $\text{min}()$

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

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



## Implementation 3: Recursion

### Algorithm 35 $\text{succ}(x)$

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

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



## Implementation 3: Recursion

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

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

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

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



## Implementation 3: Recursion

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

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

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

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

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



## Implementation 3: Recursion

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

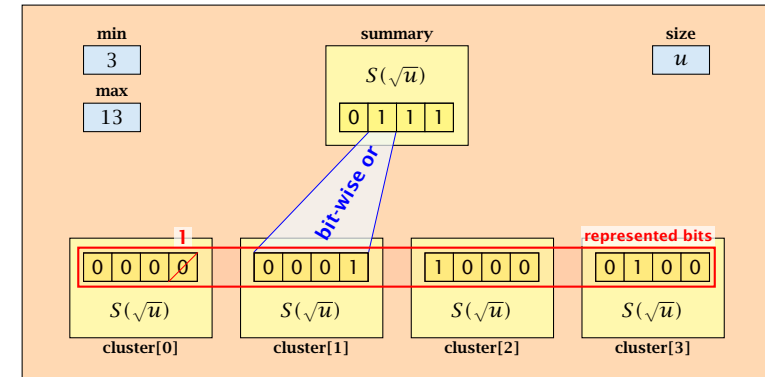
Set  $\ell := \log u$  and  $X(\ell) := T_{\text{del}}(2^\ell)$ . Then

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

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 36** max()

1: return max;

**Algorithm 37** min()

1: return min;

- ▶ Constant time.

## Implementation 4: van Emde Boas Trees

### Algorithm 38 member( $x$ )

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

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



## Implementation 4: van Emde Boas Trees

### Algorithm 39 succ( $x$ )

```
1: if  $\min \neq \text{null} \wedge x < \min$  then return  $\min$ ;
2:  $\text{maxincluster} \leftarrow \text{cluster}[\text{high}(x)].\text{max}()$ ;
3: if  $\text{maxincluster} \neq \text{null} \wedge \text{low}(x) < \text{maxincluster}$  then
4:    $\text{offs} \leftarrow \text{cluster}[\text{high}(x)].\text{succ}(\text{low}(x))$ ;
5:   return  $\text{high}(x) \circ \text{offs}$ ;
6: else
7:    $\text{succcluster} \leftarrow \text{summary}.\text{succ}(\text{high}(x))$ ;
8:   if  $\text{succcluster} = \text{null}$  then return  $\text{null}$ ;
9:    $\text{offs} \leftarrow \text{cluster}[\text{succcluster}].\text{min}()$ ;
10:  return  $\text{succcluster} \circ \text{offs}$ ;
```

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



## Implementation 4: van Emde Boas Trees

### Algorithm 40 insert( $x$ )

```
1: if  $\min = \text{null}$  then
2:    $\min = x$ ;  $\text{max} = x$ ;
3: else
4:   if  $x < \min$  then exchange  $x$  and  $\min$ ;
5:   if  $\text{cluster}[\text{high}(x)].\min = \text{null}$ ; then
6:      $\text{summary}.\text{insert}(\text{high}(x))$ ;
7:      $\text{cluster}[\text{high}(x)].\text{insert}(\text{low}(x))$ ;
8:   else
9:      $\text{cluster}[\text{high}(x)].\text{insert}(\text{low}(x))$ ;
10:  if  $x > \text{max}$  then  $\text{max} = x$ ;
```

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



## Implementation 4: van Emde Boas Trees

Note that the recursive 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 41 delete( $x$ )

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

## Implementation 4: van Emde Boas Trees

### Algorithm 41 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 recursive 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  $R(k) \leq k - 2$  for squares  $k \geq 4$ .
  - ▶ Obviously, this holds for  $k = 4$ .
  - ▶ For  $k = \ell^2 > 4$  with  $\ell$  integral we have

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

- ▶ 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].

