

## 7.4 Augmenting Data Structures

Suppose you want to develop a data structure with:

- ▶ **Insert( $x$ )**: insert element  $x$ .
- ▶ **Search( $k$ )**: search for element with key  $k$ .
- ▶ **Delete( $x$ )**: delete element referenced by pointer  $x$ .
- ▶ **find-by-rank( $\ell$ )**: return the  $\ell$ -th element; return “error” if the data-structure contains less than  $\ell$  elements.

Augment an existing data-structure instead of developing a new one.

## 7.4 Augmenting Data Structures

Suppose you want to develop a data structure with:

- ▶ **Insert( $x$ )**: insert element  $x$ .
- ▶ **Search( $k$ )**: search for element with key  $k$ .
- ▶ **Delete( $x$ )**: delete element referenced by pointer  $x$ .
- ▶ **find-by-rank( $\ell$ )**: return the  $\ell$ -th element; return “error” if the data-structure contains less than  $\ell$  elements.

**Augment an existing data-structure instead of developing a new one.**

## 7.4 Augmenting Data Structures

### How to augment a data-structure

1. choose an underlying data-structure
2. determine additional information to be stored in the underlying structure
3. verify/show how the additional information can be maintained for the basic modifying operations on the underlying structure.
4. develop the new operations

## 7.4 Augmenting Data Structures

### How to augment a data-structure

1. choose an underlying data-structure
2. determine additional information to be stored in the underlying structure
3. verify/show how the additional information can be maintained for the basic modifying operations on the underlying structure.
4. develop the new operations

## 7.4 Augmenting Data Structures

### How to augment a data-structure

1. choose an underlying data-structure
2. determine additional information to be stored in the underlying structure
3. verify/show how the additional information can be maintained for the basic modifying operations on the underlying structure.
4. develop the new operations

## 7.4 Augmenting Data Structures

### How to augment a data-structure

1. choose an underlying data-structure
2. determine additional information to be stored in the underlying structure
3. verify/show how the additional information can be maintained for the basic modifying operations on the underlying structure.
4. develop the new operations

## 7.4 Augmenting Data Structures

**Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time  $\mathcal{O}(\log n)$ .**

1. We choose a red-black tree as the underlying data-structure.
2. We store in each node  $v$  the size of the sub-tree rooted at  $v$ .
3. We need to be able to update the size-field in each node without asymptotically affecting the running time of insert, delete, and search. We come back to this step later...

## 7.4 Augmenting Data Structures

**Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time  $\mathcal{O}(\log n)$ .**

1. We choose a red-black tree as the underlying data-structure.
2. We store in each node  $v$  the size of the sub-tree rooted at  $v$ .
3. We need to be able to update the size-field in each node without asymptotically affecting the running time of insert, delete, and search. We come back to this step later...



## 7.4 Augmenting Data Structures

**Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time  $\mathcal{O}(\log n)$ .**

1. We choose a red-black tree as the underlying data-structure.
2. We store in each node  $v$  the size of the sub-tree rooted at  $v$ .
3. We need to be able to update the size-field in each node without asymptotically affecting the running time of insert, delete, and search. We come back to this step later...

## 7.4 Augmenting Data Structures

**Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time  $\mathcal{O}(\log n)$ .**

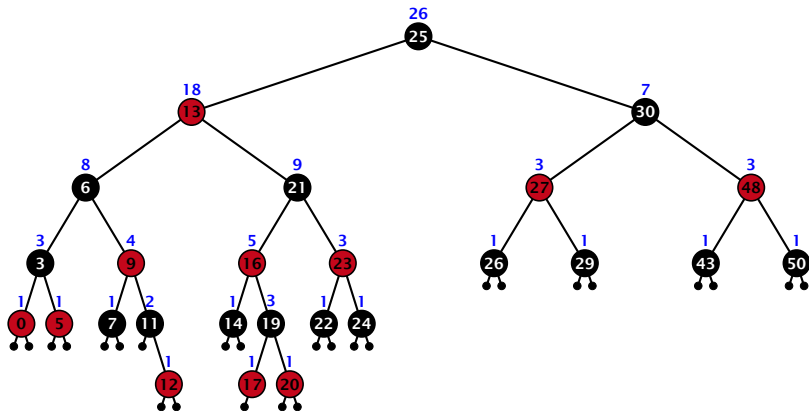
4. How does find-by-rank work?

Find-by-rank( $k$ ) := Select( $\text{root}, k$ ) with

**Algorithm 11** Select( $x, i$ )

```
1: if  $x = \text{null}$  then return error
2: if  $\text{left}[x] \neq \text{null}$  then  $r \leftarrow \text{left}[x].\text{size} + 1$  else  $r \leftarrow 1$ 
3: if  $i = r$  then return  $x$ 
4: if  $i < r$  then
5:     return Select( $\text{left}[x], i$ )
6: else
7:     return Select( $\text{right}[x], i - r$ )
```

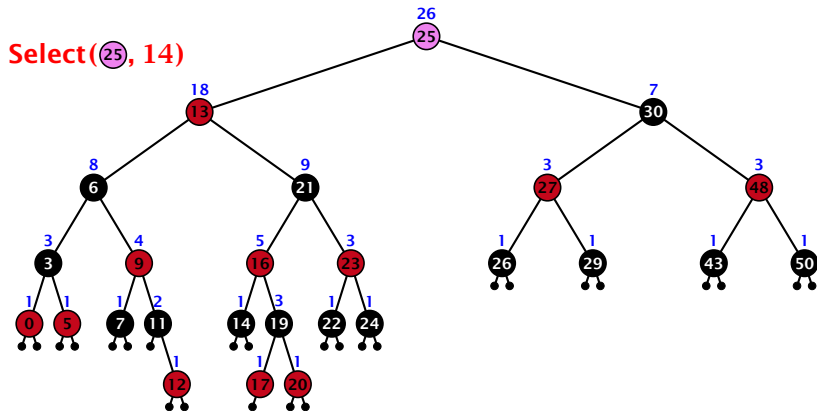
## Select( $x, i$ )



### Find-by-rank:

- ▶ decide whether you have to proceed into the left or right sub-tree
- ▶ adjust the rank that you are searching for if you go right

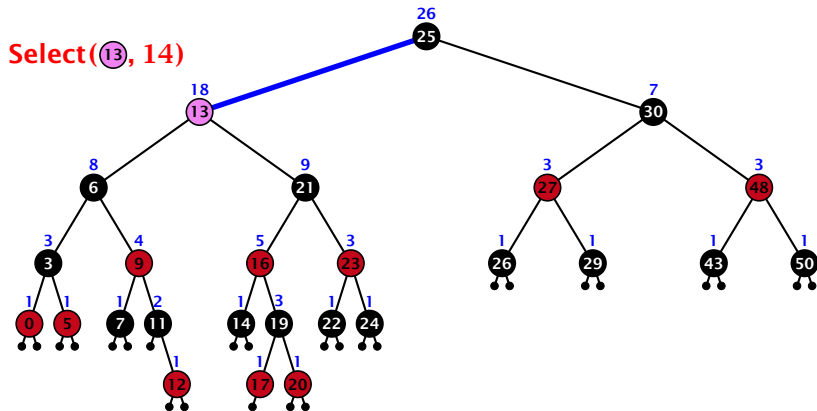
# Select( $x, i$ )



## Find-by-rank:

- ▶ decide whether you have to proceed into the left or right sub-tree
- ▶ adjust the rank that you are searching for if you go right

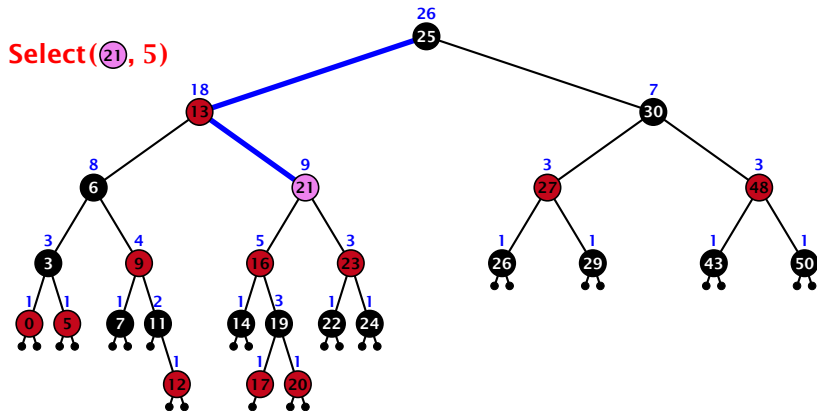
# Select( $x, i$ )



## Find-by-rank:

- ▶ decide whether you have to proceed into the left or right sub-tree
- ▶ adjust the rank that you are searching for if you go right

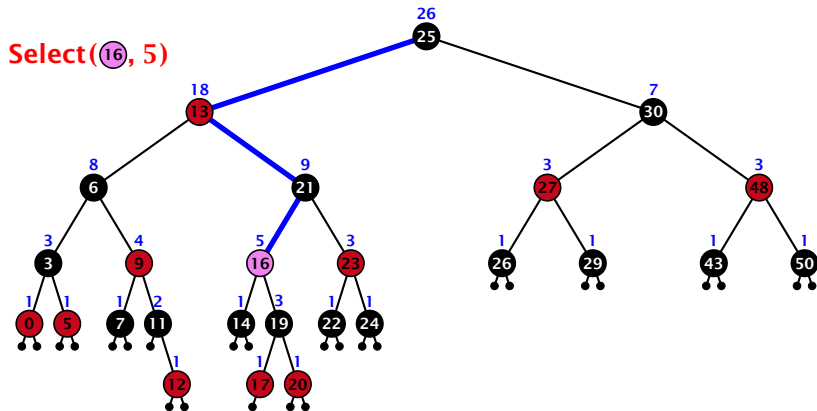
# Select( $x, i$ )



## Find-by-rank:

- ▶ decide whether you have to proceed into the left or right sub-tree
- ▶ adjust the rank that you are searching for if you go right

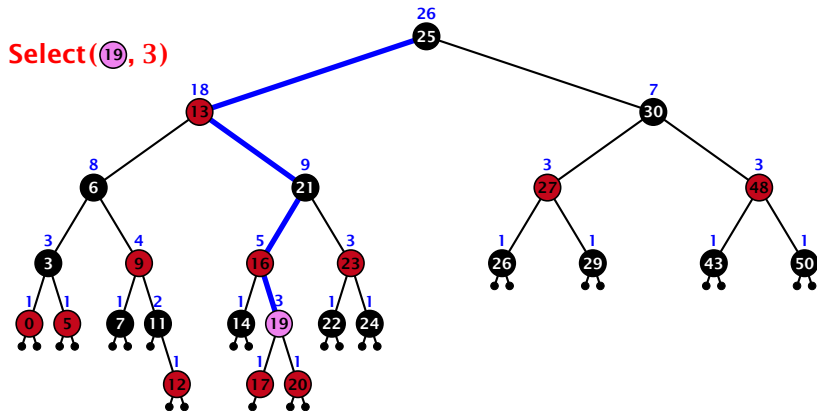
# Select( $x, i$ )



## Find-by-rank:

- ▶ decide whether you have to proceed into the left or right sub-tree
- ▶ adjust the rank that you are searching for if you go right

# Select( $x, i$ )

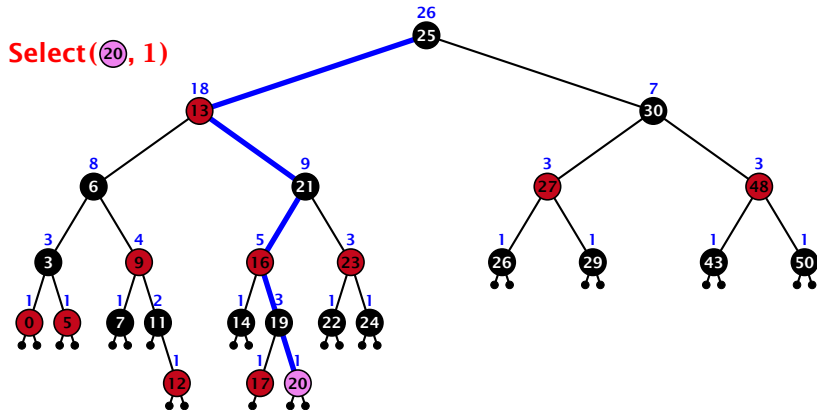


## Find-by-rank:

- ▶ decide whether you have to proceed into the left or right sub-tree
- ▶ adjust the rank that you are searching for if you go right



# Select( $x, i$ )



## Find-by-rank:

- ▶ decide whether you have to proceed into the left or right sub-tree
- ▶ adjust the rank that you are searching for if you go right

## 7.4 Augmenting Data Structures

**Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time  $\mathcal{O}(\log n)$ .**

3. How do we maintain information?

**Search( $k$ ):** Nothing to do.

**Insert( $x$ ):** When going down the search path increase the size field for each visited node. **Maintain the size field during rotations.**

**Delete( $x$ ):** Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. **Maintain the size field during rotations.**

## 7.4 Augmenting Data Structures

**Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time  $\mathcal{O}(\log n)$ .**

3. How do we maintain information?

**Search( $k$ ):** Nothing to do.

**Insert( $x$ ):** When going down the search path increase the size field for each visited node. **Maintain the size field during rotations.**

**Delete( $x$ ):** Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. **Maintain the size field during rotations.**

## 7.4 Augmenting Data Structures

**Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time  $\mathcal{O}(\log n)$ .**

3. How do we maintain information?

**Search( $k$ ):** Nothing to do.

**Insert( $x$ ):** When going down the search path increase the size field for each visited node. **Maintain the size field during rotations.**

**Delete( $x$ ):** Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. **Maintain the size field during rotations.**

## 7.4 Augmenting Data Structures

**Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time  $\mathcal{O}(\log n)$ .**

3. How do we maintain information?

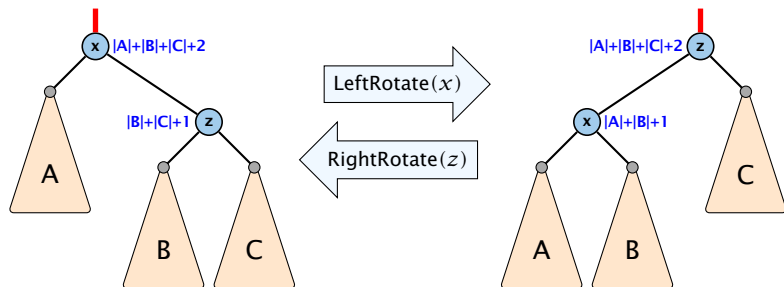
**Search( $k$ ):** Nothing to do.

**Insert( $x$ ):** When going down the search path increase the size field for each visited node. **Maintain the size field during rotations.**

**Delete( $x$ ):** Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. **Maintain the size field during rotations.**

# Rotations

The only operation during the fix-up procedure that alters the tree and requires an update of the size-field:



The nodes  $x$  and  $z$  are the only nodes changing their size-fields.

The new size-fields can be computed **locally** from the size-fields of the children.