2IL50 Data Structures

2023-24 Q3

Lecture 8: Augmenting Data Structures

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Announcement

Next Monday March 11 only two lecture rooms: AUD 3 and AUD 6

Binary Search Trees

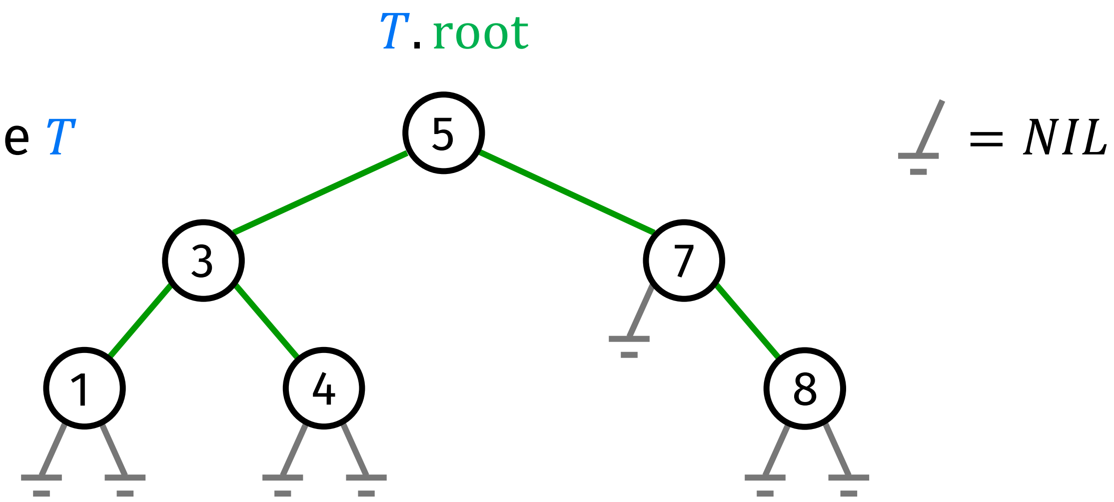
Binary search trees

root of T denoted by $T.\mathrm{root}$

internal nodes have four fields:

- key (and possible other satellite data)
- left: points to left child
- right: points to right child
- p: points to parent. $T.\mathrm{root}.p = NIL$

Binary tree T



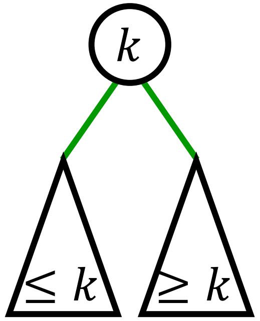
Binary search trees

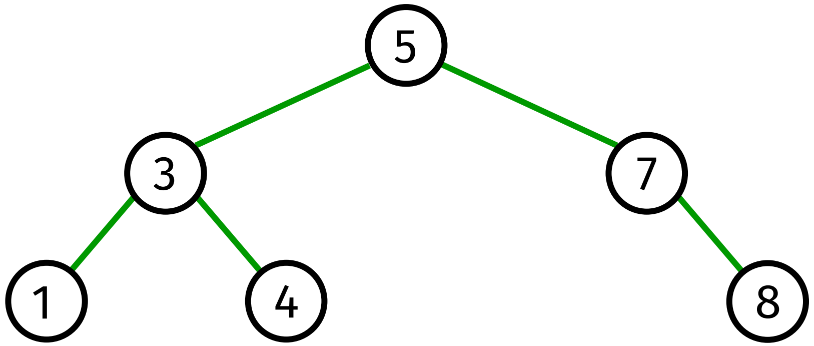
A binary tree is

- a leaf or
- a root node x with a binary tree as its left and/or right child

Binary-search-tree property

- if y is in the left subtree of x, then $y.\mathrm{key} \leq x.\mathrm{key}$
- if y is in the right subtree of x, then $y.\mathrm{key} \geq x.\mathrm{key}$





Minimizing the running time

All operations can be executed in time proportional to the height h of the tree
(instead of proportional to the number n of nodes in the tree)

Worst case: $\Theta(n)$

Solution: guarantee small height *(balance the tree)*

➡ $h = \Theta(\log n)$

Balanced binary search trees

Search, Minimum, Maximum, Predecessor, Successor, Insert, and Delete can be executed in time $\Theta(\log n)$.

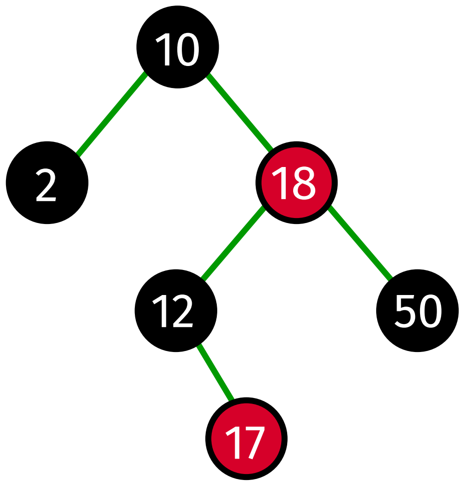
Red-black Trees

Red-black trees

Red-black tree

binary search tree where each node has a color attribute which is either red or black

Red-black properties



1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
   (Hence no two reds in a row on a simple path from the root to a leaf)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Lemma

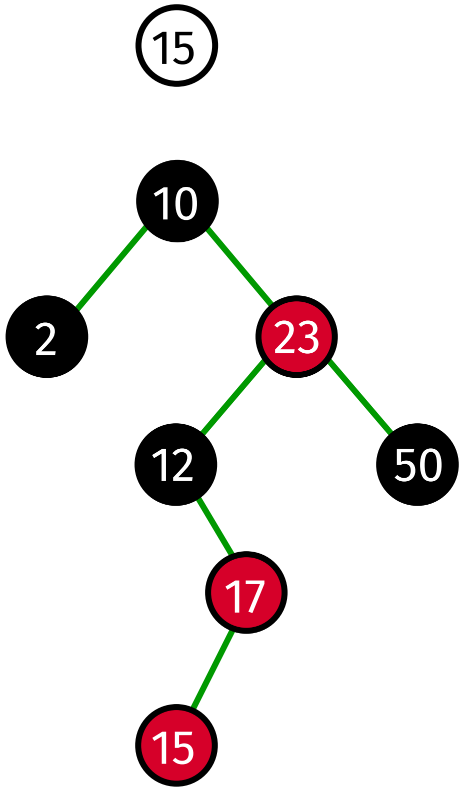
A red-black tree with n nodes has height $\leq 2\log(n+1)$.

Red-black trees: Insertion

1. Do a regular binary search tree insertion
2. Fix the red-black properties

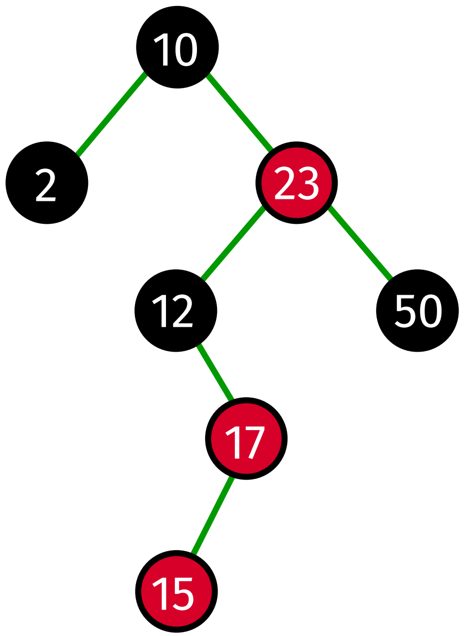
Step 1

- find the leaf where the node should be inserted
- replace the leaf by a red node that contains the key to be inserted



Red-black trees: Insertion

1. Do a regular binary search tree insertion
2. Fix the red-black properties

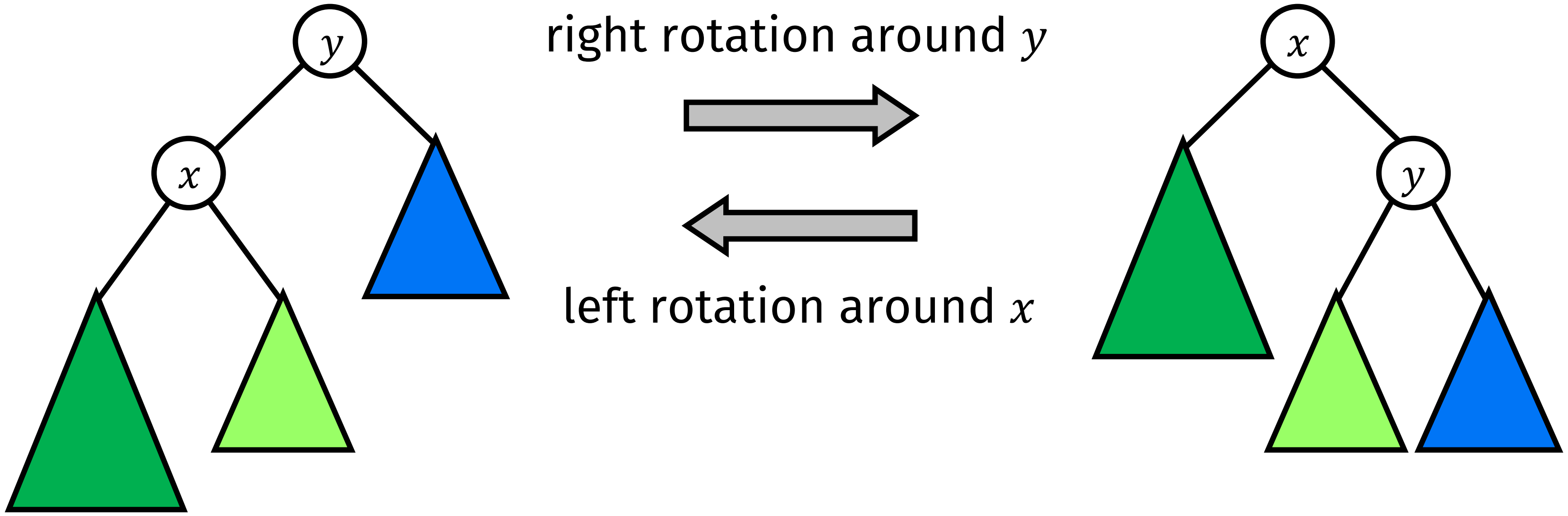


Red-black properties

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

The new node is red ➡ Property 2 or 4 can be violated.
Remove the violation by rotations and recoloring.

Rotation



Step 2: Fixing the red-black properties

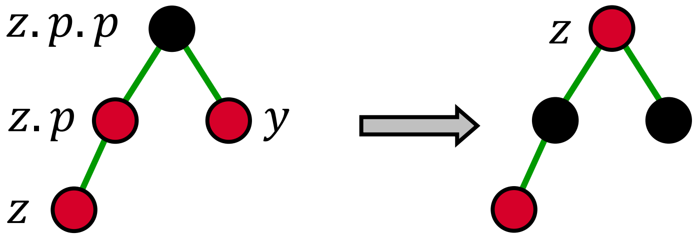
Invariant: z and $z.p$ are both red and this is the only red-black violation
(or z is a red root ➡ just recolor and terminate)

while $z \neq T.\text{root}$ and $z.p$ is red ⋯→ hence $z.p \neq T.\text{root}$

if $z.p == z.p.p.\text{left}$

$y = z.p.p.\text{right}$

case i: y is red



color $z.p$ and y black, color $z.p.p$ red

$z = z.p.p$

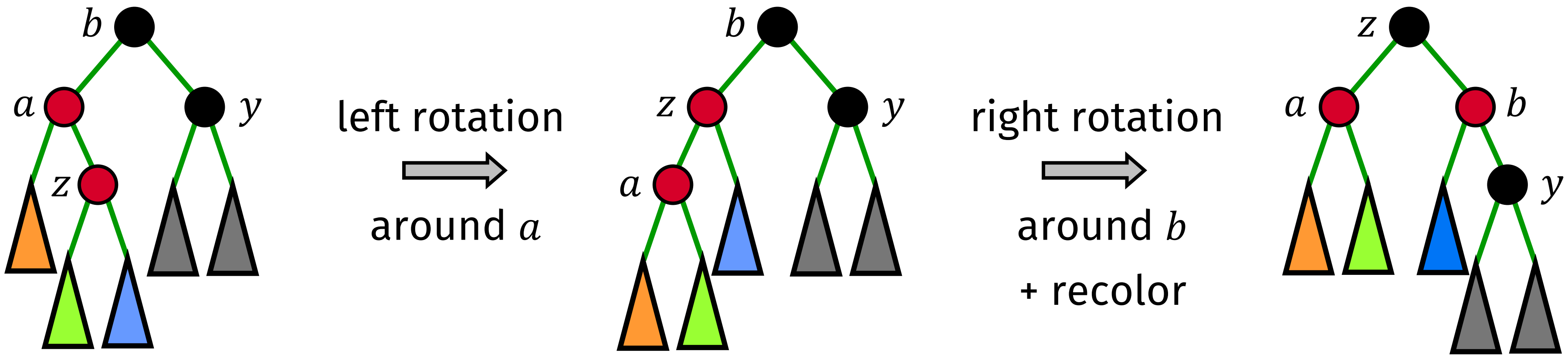
continue up the tree

else … // symmetric case

Step 2: Fixing the red-black properties

Invariant: z and $z.p$ are both red and this is the only red-black violation
(or z is a red root ➡ just recolor and terminate)

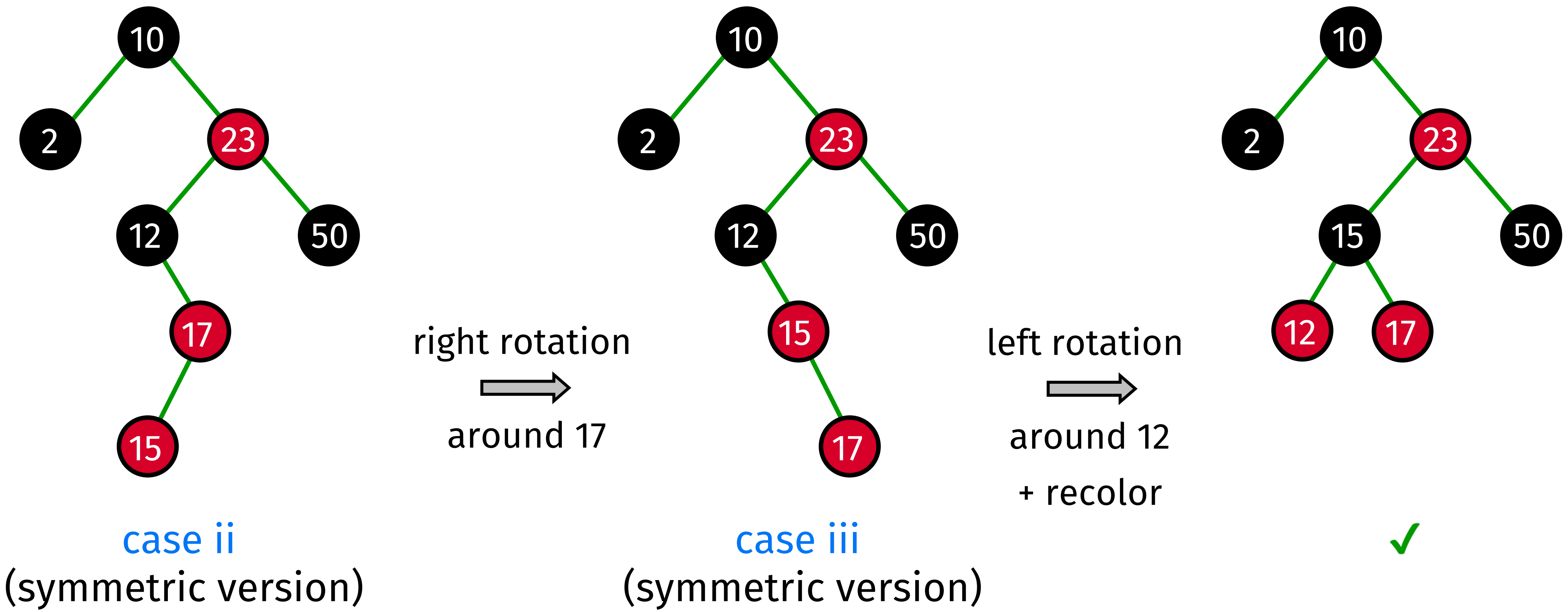
case ii and iii: y is black



case ii: $z == z.p.\mathrm{right}$ case iii: $a == a.p.\mathrm{left}$ ✓

Red-black trees: Insertion

1. Do a regular binary search tree insertion
2. Fix the red-black properties



Red-black trees: Insertion

1. Do a regular binary search tree insertion
2. Fix the red-black properties
   - move up the tree as long as case i occurs and recolor accordingly
   - as soon as case ii or iii occurs
     at most two rotations and two recolorings ✓
   - if you reach the root or the parent of z is black ✓

Running time? $O(\text{height of the tree}) = O(\log n)$

Red-black trees: Deletion

1. Do a regular binary search tree deletion
2. Fix the red-black properties
   - Slightly more complicated case distinction than insertion
     see book for details
   - can be done by recoloring as before and at most three rotations

Search, insert, and delete can be executed with a red-black tree in $O(\log n)$ time.

Augmenting Data Structures

Data structures

Data structures are used in many applications

- directly: the user repeatedly queries the data structure
- indirectly: to make algorithms run faster

In most cases a standard data structure is sufficient
(possibly provided by a software library)

But sometimes one needs additional operations
that are not supported by any standard data structure

➡ need to design new data structure?

Not always: often augmenting an existing structure is sufficient

Example

S set of elements, each with a unique key.

Operations

Search(S, k): return a pointer to an element x in S with $x.\mathrm{key} = k$,
or NIL if such an element does not exist.

OS-Select(S, i): return a pointer to an element x in S with the i^{th} smallest key
(the key with rank i)

Solution: sorted array

A	2	5	6	9	10	11	24	27	31	35	41	43	54	55	73

the key with rank i is stored in $A[i]$

Example

S set of elements, each with a unique key.

Operations

Search(S, k): return a pointer to an element x in S with $x.\mathrm{key} = k$,
or NIL if such an element does not exist.

OS-Select(S, i): return a pointer to an element x in S with the i^{th} smallest key
(the key with rank i)

Insert(S, x): inserts element x into S, that is, $S \leftarrow S \cup \{x\}$

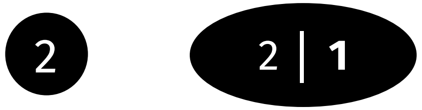
Delete(S, x): remove element x from S

Solution?

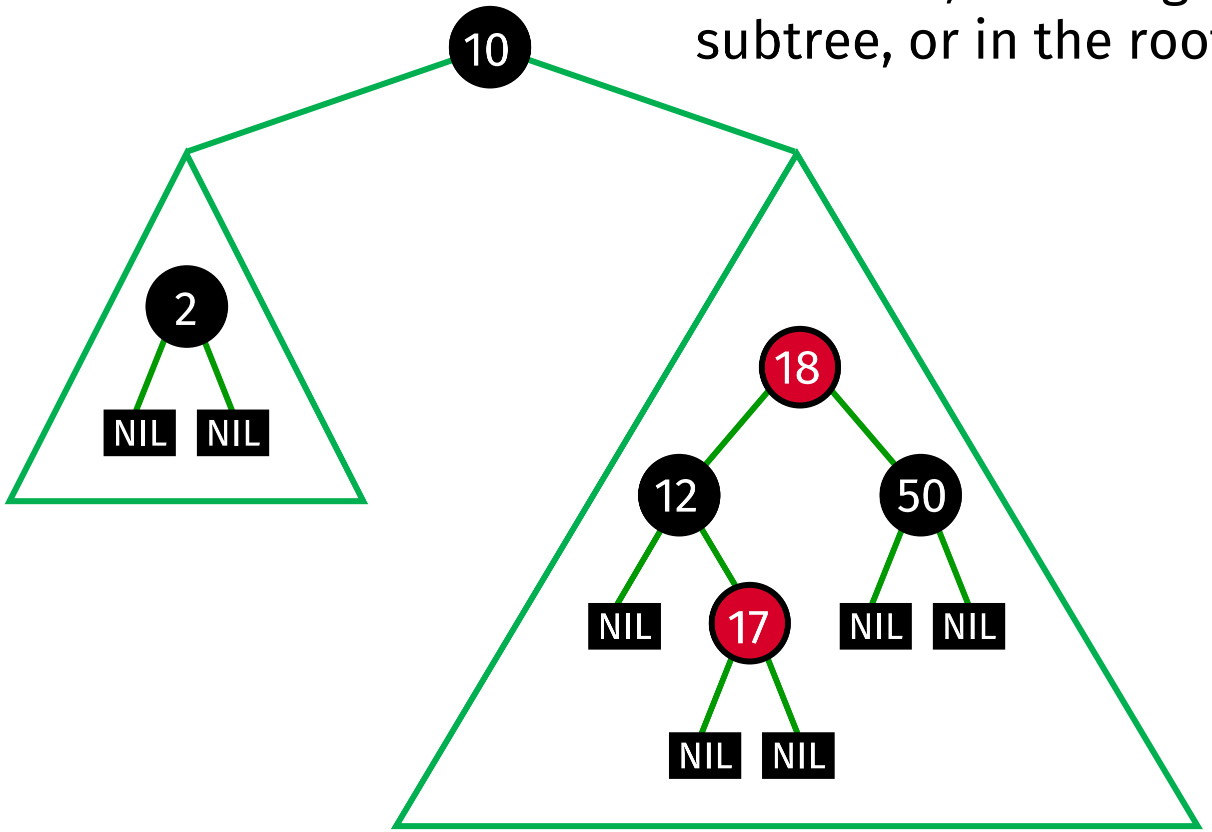
Use red-black trees

OS-Select(S, 3): report key with rank 3

Idea 1: store the rank of each node in the node



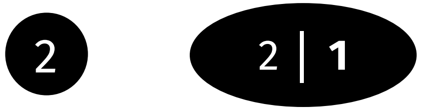
Is the key with rank 3 in the left subtree, in the right subtree, or in the root?



Use red-black trees

OS-Select(S, 3): report key with rank 3

Idea 1: store the rank of each node in the node

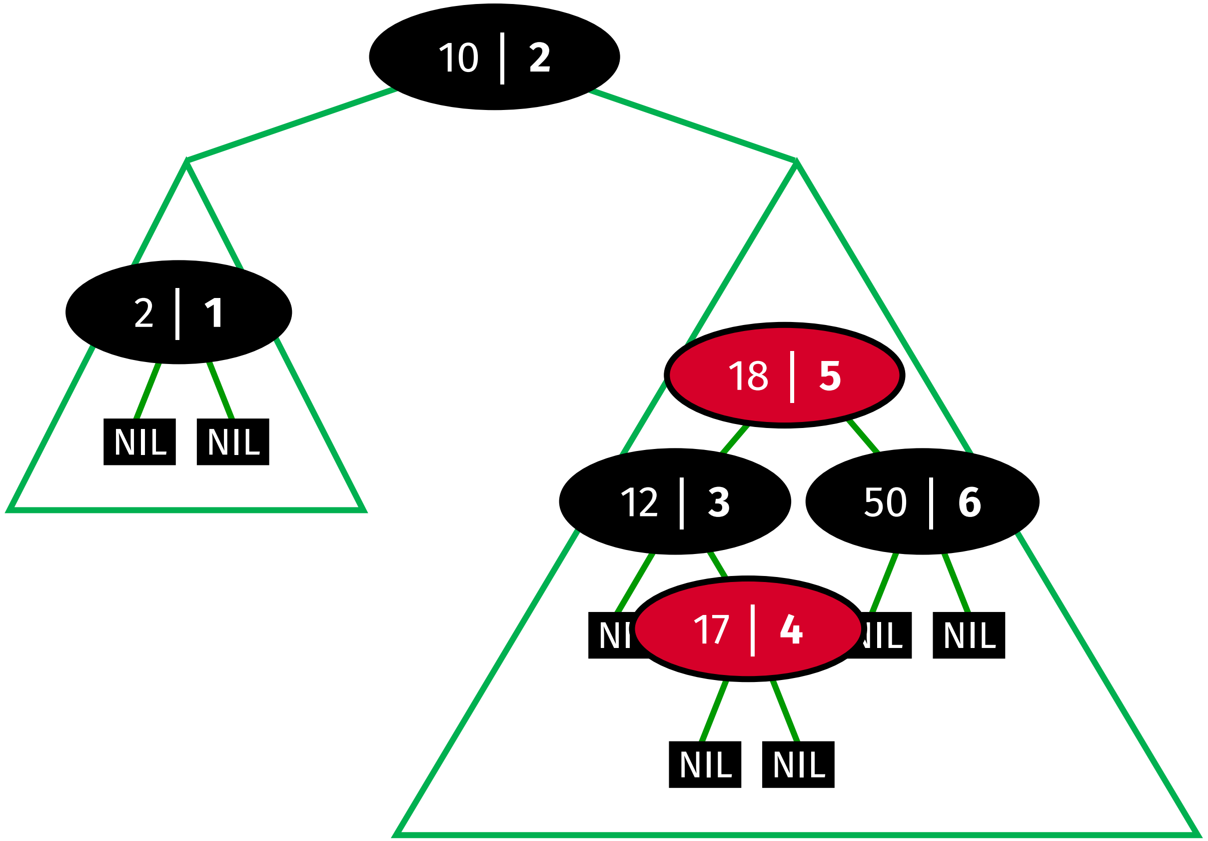


Problem:

Insertion can change the rank of every node!

Worst case $O(n)$

Idea 2: store the size of the subtree in each node



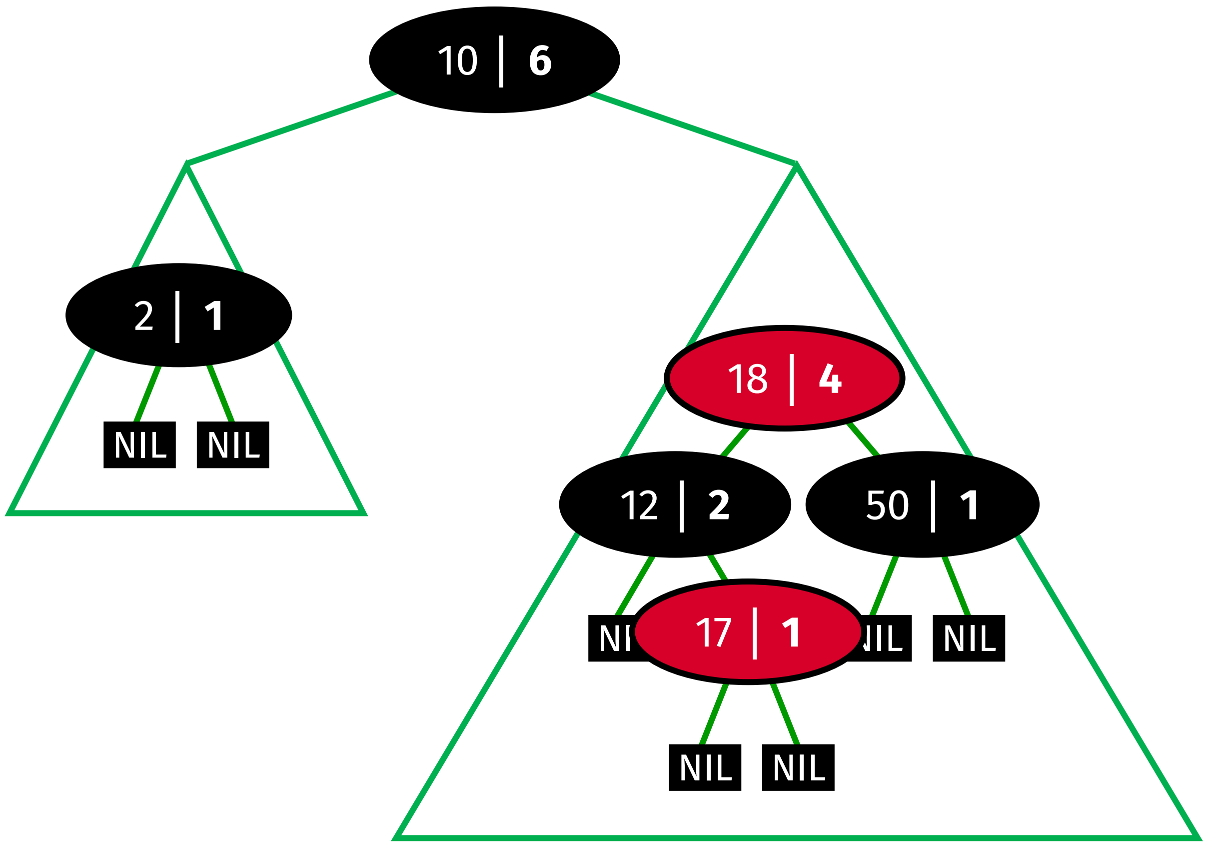
Idea 2: store the size of the subtree

Store in each node x:

- $x.\mathrm{left}$, $x.\mathrm{right}$
- $x.\mathrm{parent}$
- $x.\mathrm{key}$
- $x.\mathrm{color}$

- $x.\mathrm{size} =$ number of keys in subtree rooted at x $(NIL.\mathrm{size} = 0)$

Order-Statistic tree



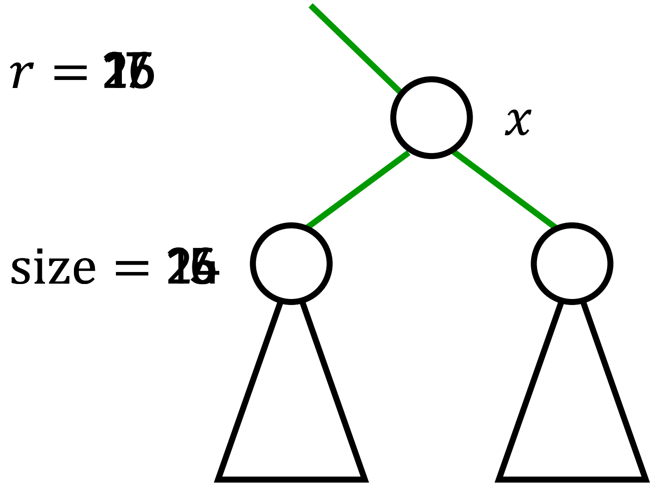
Order-statistic trees: OS-Select

OS-Select(x, i): return pointer to node containing the i^{th} smallest key of the subtree rooted at x

OS-Select(x, i)

```
r = x.left.size + 1
if i == r
        return x
elseif i < r
        return OS-Select(x.left, i)
else
        return OS-Select(x.right, i − r)
```

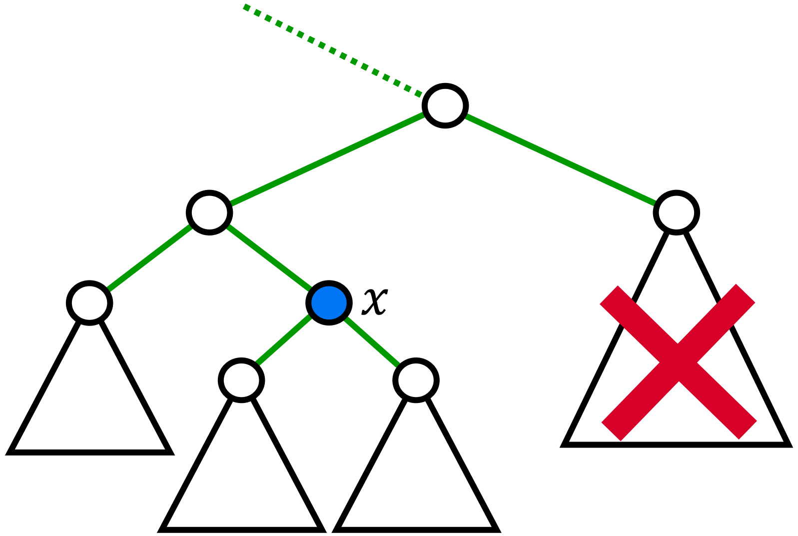
OS-Select(x, 17)



Running time? $O(\log n)$

Order-statistic trees: OS-Rank

OS-Rank(T, x): return the rank of x in the linear order determined by an inorder walk of T
$= 1 +$ number of keys smaller than x

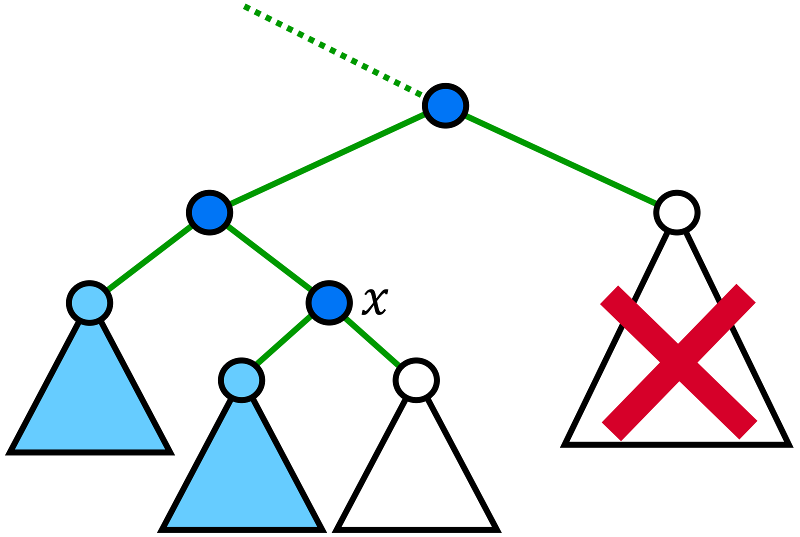


Order-statistic trees: OS-Rank

OS-Rank(T, x): return the rank of x in the linear order determined by an inorder walk of T
$= 1 +$ number of keys smaller than x

OS-Rank(T, x)

$r = x.\mathrm{left.size} + 1$
$y = x$
while $y \neq T.\mathrm{root}$
    if $y == y.p.\mathrm{right}$
        $r = r + y.p.\mathrm{left.size} + 1$
    $y = y.p$
return r



Running time? $O(\log n)$

OS-Rank: Correctness

OS-Rank(T, x)

$r = x.\mathrm{left.size} + 1$
$y = x$
while $y \neq T.\mathrm{root}$
  if $y == y.p.\mathrm{right}$
    $r = r + y.p.\mathrm{left.size} + 1$
  $y = y.p$
return r

Invariant

At the start of each iteration of the **while** loop,
r = rank of $x.\mathrm{key}$ in $\underbrace{T_y}_{\text{subtree with root } y}$

Initialization

r = rank of $x.\mathrm{key}$ in T_x $(y = x)$
= number of keys smaller than $x.\mathrm{key}$ in $T_x + 1$
$= x.\mathrm{left.size} + 1$ (binary-search-tree property)

OS-Rank: Correctness

OS-Rank(T, x)

$r = x.\mathrm{left.size} + 1$
$y = x$
while $y \neq T.\mathrm{root}$
  if $y == y.p.\mathrm{right}$
    $r = r + y.p.\mathrm{left.size} + 1$
  $y = y.p$
return r

Invariant

At the start of each iteration of the **while** loop,
$r =$ rank of $x.\mathrm{key}$ in T_y

Termination

loop terminates when $y = T.\mathrm{root}$

➡ subtree rooted at y is entire tree

➡ $r =$ rank of $x.\mathrm{key}$ in entire tree

OS-Rank: Correctness

OS-Rank(T, x)

$r = x.\text{left}.\text{size} + 1$
$y = x$
while $y \neq T.\text{root}$
  if $y == y.p.\text{right}$
    $r = r + y.p.\text{left}.\text{size} + 1$
  $y = y.p$
return r

Invariant

At the start of each iteration of the **while** loop,
$r =$ rank of $x.\text{key}$ in T_y

Maintenance

case i: $y = y.p.\text{right}$

➡ all keys in $T_{y.p.\text{left}}$ and $y.p.\text{key}$ are smaller than $x.\text{key}$

➡ rank $x.\text{key}$ in $T_{y.p}$ = rank $x.\text{key}$ in $T_y + y.p.\text{left}.\text{size} + 1$

case ii: $y = y.p.\text{left}$

➡ all keys in $T_{y.p.\text{right}}$ and $y.p.\text{key}$ are larger than $x.\text{key}$

➡ rank $x.\text{key}$ in $T_{y.p}$ = rank $x.\text{key}$ in T_y

Order-statistic trees: Insertion and deletion

Insertion and deletion
as in a regular red-black tree, but we have to update $x.\mathit{size}$ field

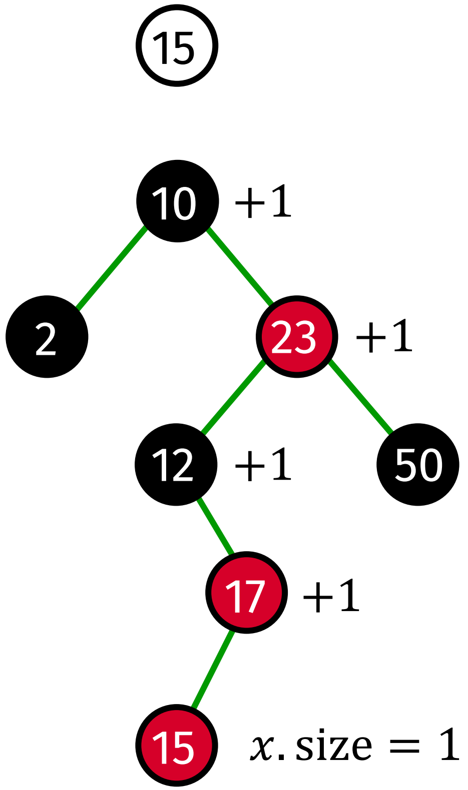
Red-black trees: Insertion

1. Do a regular binary search tree insertion
2. Fix the red-black properties

Step 1

- find the leaf where the node should be inserted
- replace the leaf by a red node that contains the key to be inserted

- size of the new node = 1
- increment size of each node on the search path

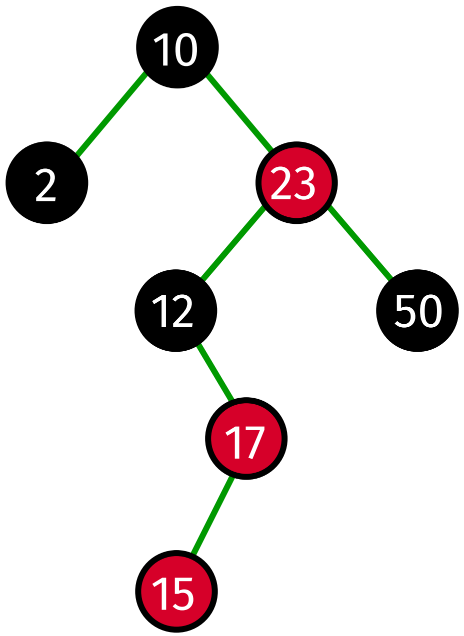


Red-black trees: Insertion

1. Do a regular binary search tree insertion
2. Fix the red-black properties

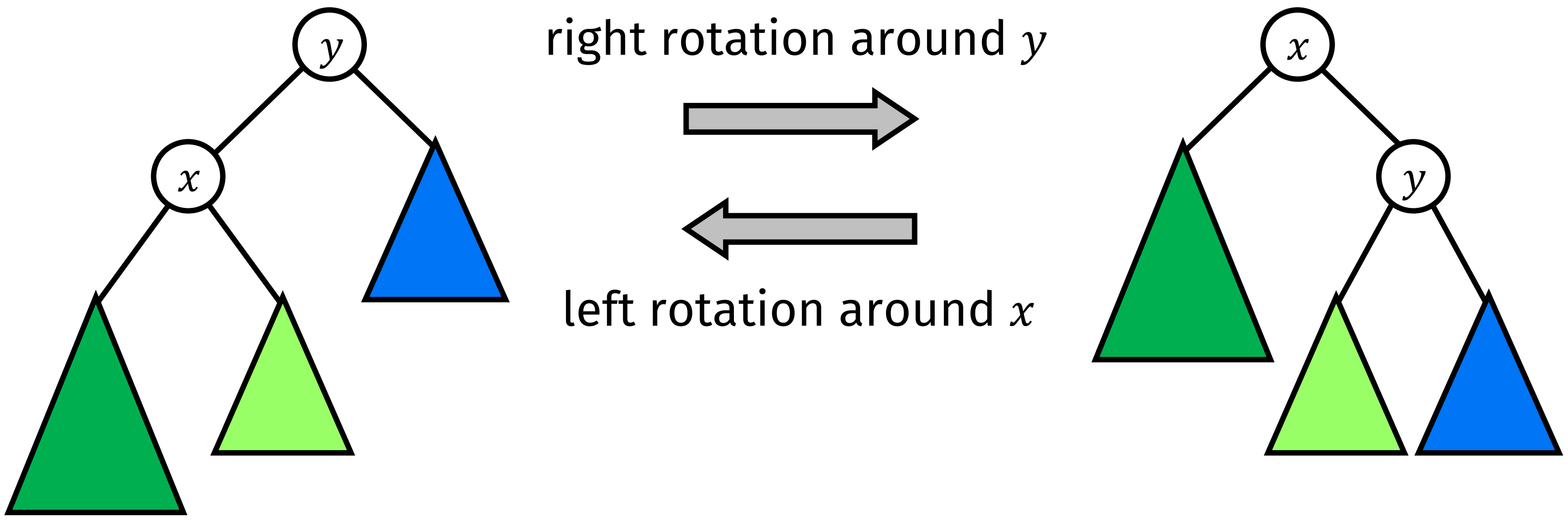
Red-black properties

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.



The new node is red ➡ Property 2 or 4 can be violated.
Remove the violation by rotations and recoloring.

Rotation



A rotation affects only $x.\mathrm{size}$ and $y.\mathrm{size}$

We can determine the new values based on the size of children:

$$x.\mathrm{size} = x.\mathrm{left}.\mathrm{size} + x.\mathrm{right}.\mathrm{size} + 1$$

and the same for y …

Order-statistic trees

The operations Insert, Delete, Search, OS-Select, and OS-Rank can be executed with an order-statistic tree in $O(\log n)$ time.

Augmenting data structures

Methodology for augmenting a data structure

1. Choose an underlying data structure.
2. Determine additional information to maintain.
3. Verify that we can maintain additional information for existing data structure operations.
4. Develop new operations.

OS tree

1. R-B tree
2. $x.\mathrm{size}$
3. maintain size during insert and delete
4. OS-Select and OS-Rank

You don’t need to do these steps in strict order!

Red-black trees are very well suited to augmentation …

Augmenting red-black trees

Theorem [RB-tree Augmentation]

Augment an RB-tree with field f, where $x.f$ depends only on information in x, $x.\mathrm{left}$, and $x.\mathrm{right}$ (including $x.\mathrm{left}.f$ and $x.\mathrm{right}.f$).
Then we can maintain values of f in all nodes during insert and delete without affecting $O(\log n)$ performance.

When we alter information in x, changes propagate only upward on the search path for x …

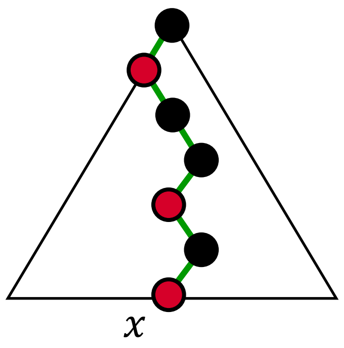
Augmenting red-black trees

Theorem [RB-tree Augmentation]

Augment an RB-tree with field f, where $x.f$ depends only on information in x, $x.\mathrm{left}$, and $x.\mathrm{right}$ (including $x.\mathrm{left}.f$ and $x.\mathrm{right}.f$).
Then we can maintain values of f in all nodes during insert and delete without affecting $O(\log n)$ performance.

Proof (insert)

Step 1 Do a regular binary search tree insertion



go up from inserted node and update f

additional time: $O(\log n)$

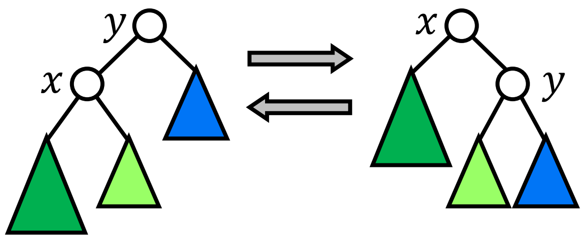
Augmenting red-black trees

Theorem [RB-tree Augmentation]

Augment an RB-tree with field f, where $x.f$ depends only on information in x, $x.\mathrm{left}$, and $x.\mathrm{right}$ (including $x.\mathrm{left}.f$ and $x.\mathrm{right}.f$). Then we can maintain values of f in all nodes during insert and delete without affecting $O(\log n)$ performance.

Proof (insert)

Step 2 Fix the red-black properties by rotations and recoloring



update f for x, y, and their ancestors

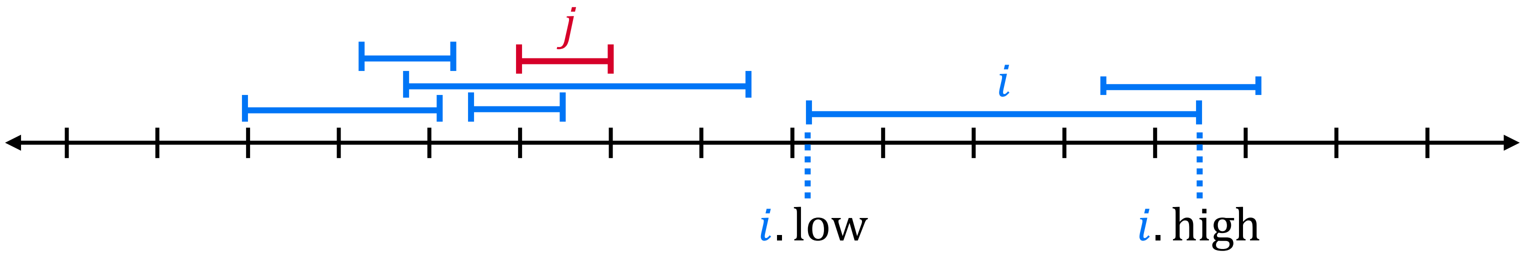
additional time per rotation: $O(\log n)$

Example: Interval Trees

Interval trees

S set of closed intervals

closed: endpoints are part of the interval



Operations

Interval-Insert(T, x): adds an interval x, whose int field is assumed to contain an interval, to the interval tree T.

Interval-Delete(T, x): removes the element x from the interval tree T.

Interval-Search(T, j): returns pointer to a node x in T such that $x.\text{int}$ overlaps j, or NIL if no such element exists.

Methodology

1. Choose an underlying data structure.
2. Determine additional information to maintain.
3. Verify that we can maintain additional information for existing data structure operations.
4. Develop new operations.

Methodology

1. Choose an underlying data structure.

   - use red-black trees
     - each node x contains interval $x.\mathrm{int}$
     - key is left endpoint $x.\mathrm{int}.\mathrm{low}$

   inorder walk would list intervals sorted by low endpoint

2. Determine additional information to maintain.

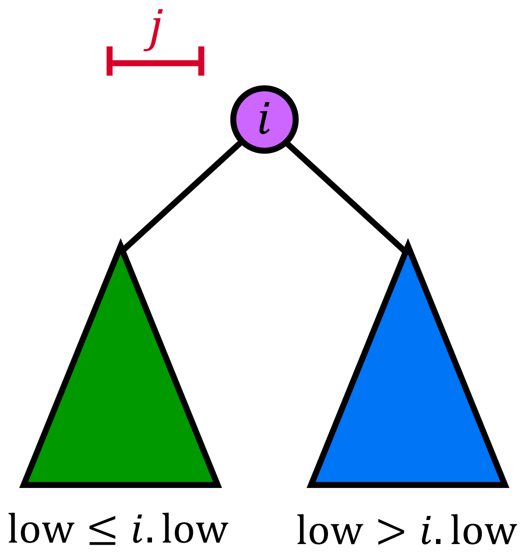
3. Verify that we can maintain additional information
   for existing data structure operations.

4. Develop new operations.

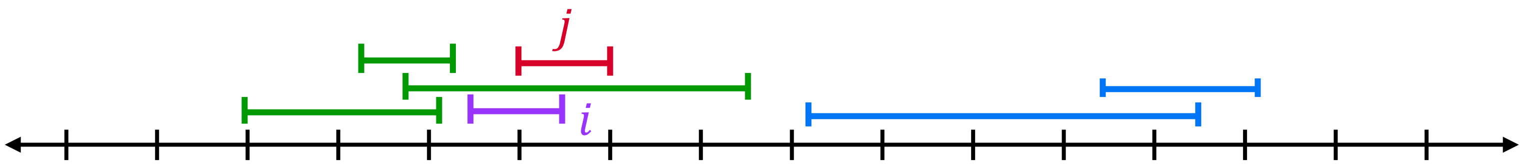
Methodology

1. Choose an underlying data structure. ✓
2. Determine additional information to maintain.
3. Verify that we can maintain additional information for existing data structure operations.
4. Develop new operations.

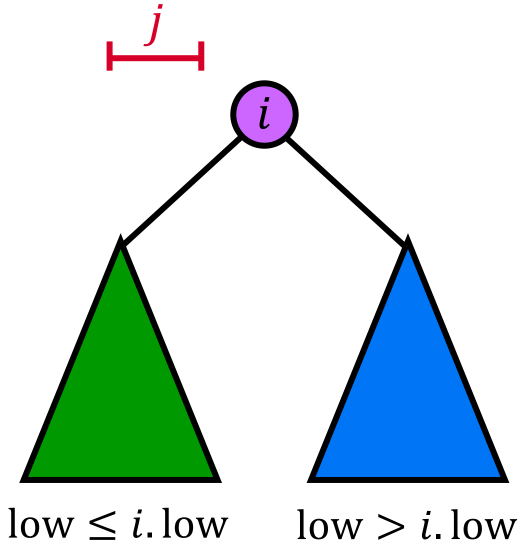
Additional information for Interval-Search



case 1: $i \cap j \neq \emptyset$ report i



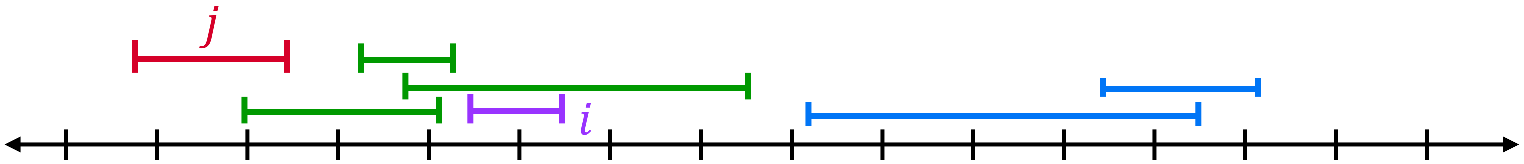
Additional information for Interval-Search



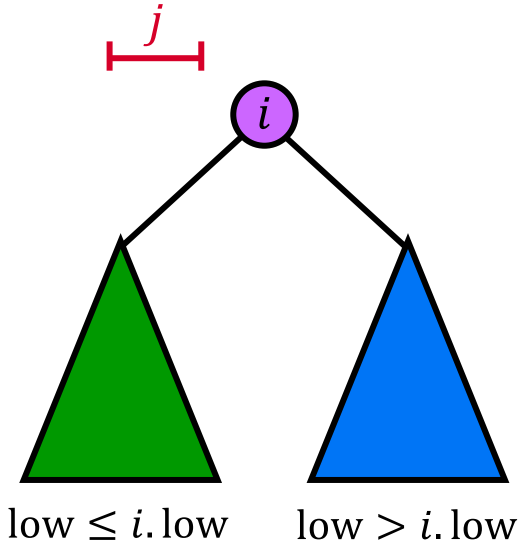
case 1: $i \cap j \neq \emptyset$ report i

case 2: j lies left of i

j cannot overlap any interval in the right subtree



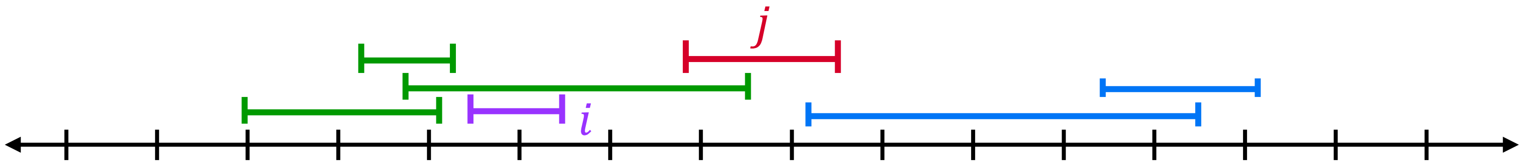
Additional information for Interval-Search



case 1: $i \cap j \neq \emptyset$ report i

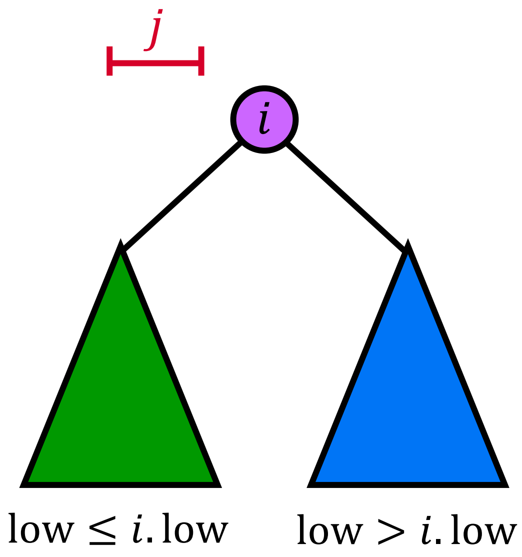
case 2: j lies left of i
j cannot overlap any interval in the right subtree

case 3: j lies right of i
need additional information!



$x.\text{max}$ = max endpoint value in subtree rooted at x
= $\max\{i.\text{high}$ where i is stored in the subtree rooted at $x\}$

Additional information for Interval-Search



case 1: $i \cap j \neq \emptyset$ report i

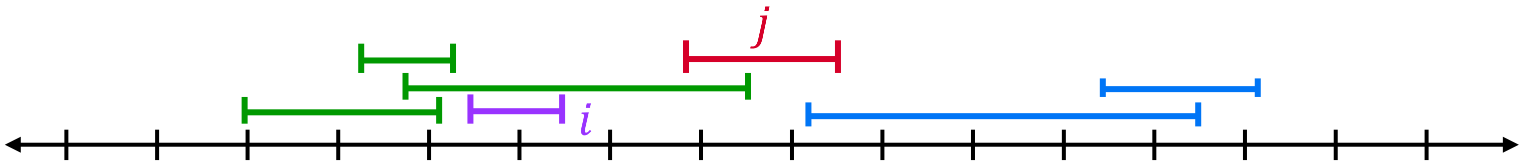
case 2: j lies left of i

j cannot overlap any interval in the right subtree

case 3: j lies right of i

j overlaps interval in left subtree if and only if

$j.\text{low} \le i.\text{left.max}$



$x.\text{max}$ = max endpoint value in subtree rooted at x

= $\max\{i.\text{high}$ where i is stored in the subtree rooted at $x\}$

Methodology

1. Choose an underlying data structure. ✓
2. Determine additional information to maintain. ✓
3. Verify that we can maintain additional information for existing data structure operations.
4. Develop new operations.

Interval-Search

Interval-Search(T, j)

```
x = T.root
while x ≠ T.nil and j does not overlap x.int
    if x.left ≠ NIL and x.left.max ≥ j.low
        x = x.left
    else
        x = x.right
return x
```

Correctness

Invariant If tree T contains an interval that overlaps j, then there is such an interval in the subtree rooted at x.

Running time? $O(\log n)$

Methodology

1. Choose an underlying data structure. ✓
2. Determine additional information to maintain. ✓
3. Verify that we can maintain additional information for existing data structure operations.
4. Develop new operations. ✓

Augmenting red-black trees

Theorem [RB-tree Augmentation]

Augment an RB-tree with field f, where $x.f$ depends only on information in x, $x.\mathrm{left}$, and $x.\mathrm{right}$ (including $x.\mathrm{left}.f$ and $x.\mathrm{right}.f$).
Then we can maintain values of f in all nodes during insert and delete without affecting $O(\log n)$ performance.

Additional information $x.\mathrm{max}$ = max endpoint value in subtree rooted at x

$x.\mathrm{max}$ depends only on

- information in x: $x.\mathrm{int}.\mathrm{high}$
- information in $x.\mathrm{left}$: $x.\mathrm{left}.\mathrm{max}$
- information in $x.\mathrm{right}$: $x.\mathrm{right}.\mathrm{max}$
- $x.\mathrm{max} = \max\{x.\mathrm{int}.\mathrm{high}, x.\mathrm{left}.\mathrm{max}, x.\mathrm{right}.\mathrm{max}\}$

➡ insert and delete still run in $O(\log n)$ time