2IL50 Data Structures

2023-24 Q3

Lecture 7: Binary Search Trees

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Announcements

Practice version of Segment 1 Interim Test on Ans

Remaining tests will be published after the results are known

Segment 2 Interim Test this Thursday

- New room distribution, finalized tomorrow
- No new rooms
- Additional time students now in Neuron 0.242

- Bring your own scrap paper – loose sheets!
- No electronic devices, also not once you are done – a book?
- External mice are fine, external keyboards not.
- The keyboard is not your enemy, don’t bash it loudly!
- Respect your fellow students: stay seated and quiet until the test is over!
- *The test auto-submits in Ans after 45 (55) min*

Dynamic Sets

Dynamic sets

Dynamic sets
Sets that can grow, shrink, or otherwise change over time.

Two types of operations:

■ queries	return information about the set
■ modifying operations	change the set

Common queries
Search, Minimum, Maximum, Successor, Predecessor

Common modifying operations
Insert, Delete

Dictionary

Dictionary

stores a set S of elements, each with an associated key (integer value)

Operations

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

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

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

Implementing a dictionary

	Search	Insert	Delete
linked list	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
sorted array	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
hash table	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Hash table

Running times are average times and assume indepedent uniform hashing and a large enough table (for example, of size $2n$)

Today Binary search trees

Binary Search Trees

Binary search trees

Binary search trees are an important data structure for dynamic sets.

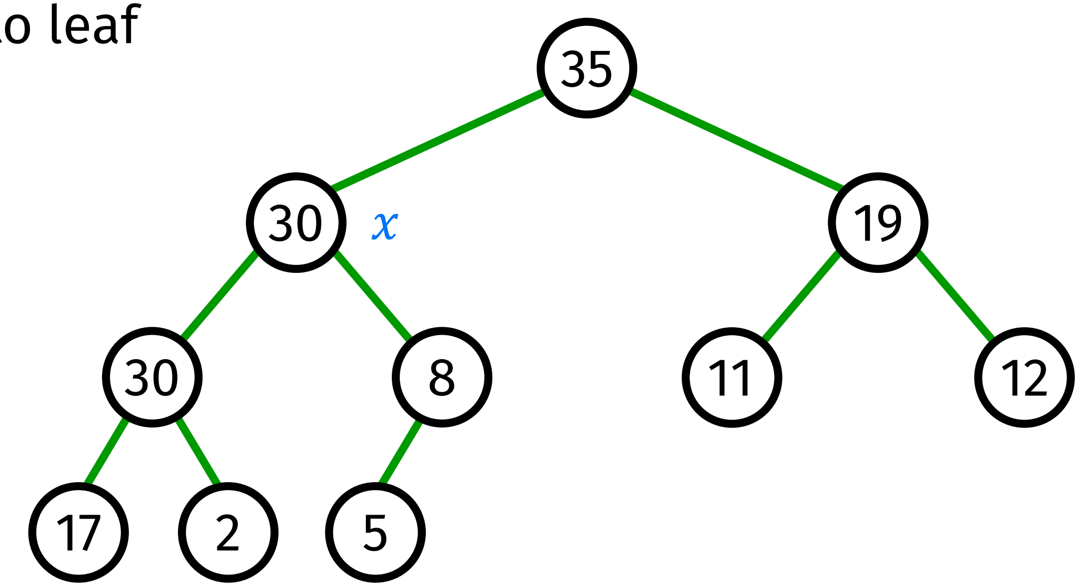
They can be used as both a dictionary and a priority queue.

They accomplish many dynamic-set operations in $O(h)$ time,
where $h =$ height of tree.

Tree terminology

Binary tree: every node has 0, 1, or 2 children
Root: top node (no parent)
Leaf: node without children
Subtree rooted at node x: all nodes below and including x
Depth of node x: length of path from root to x
Depth of tree: max. depth over all nodes
Height of node x: length of longest path from x to leaf
Height of tree: height of root
Level: set of nodes with same depth

Family tree terminology
Left/right child
Parent
Grandparent ...

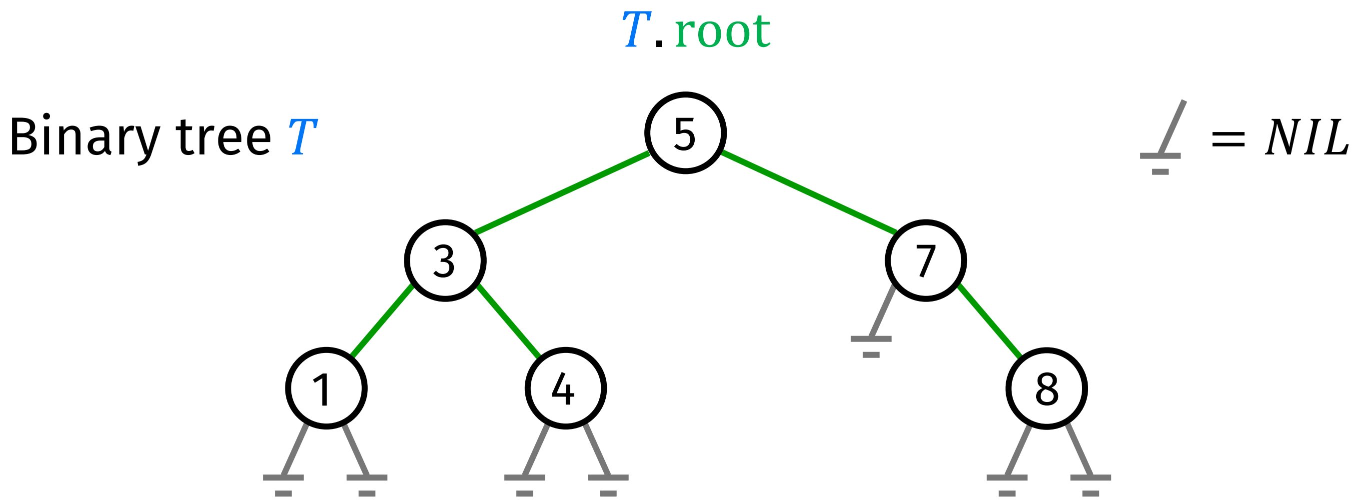


Binary search trees

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. $T.\mathrm{root}.p = NIL$



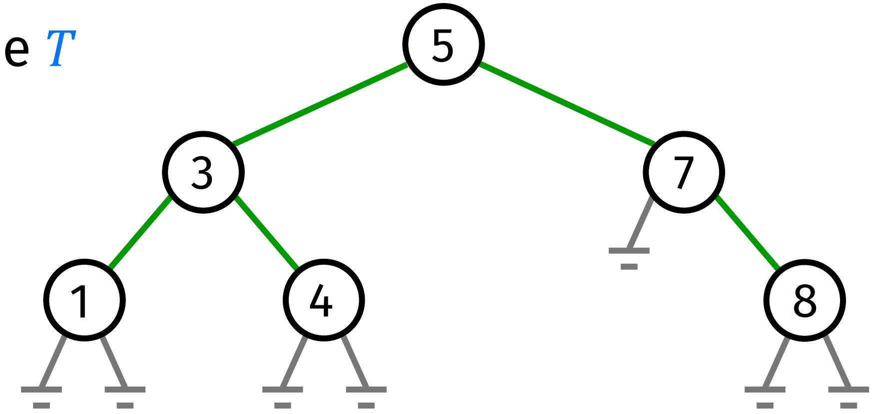
Binary search trees

Keys are stored only in internal nodes!

There are binary search trees which store keys only in the leaves ...

$T.\mathrm{root}$

Binary tree T



$= NIL$

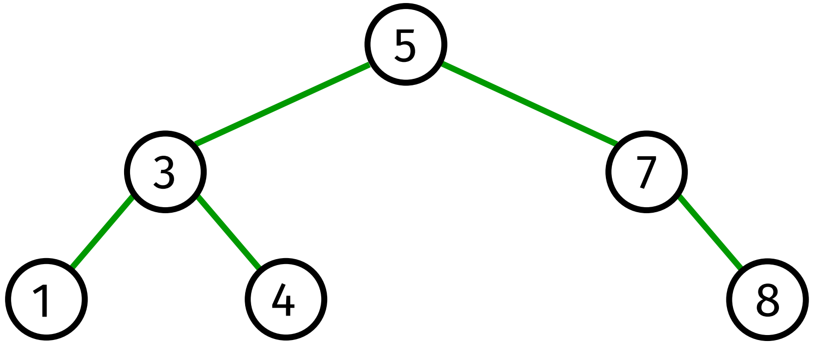
Binary search trees

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

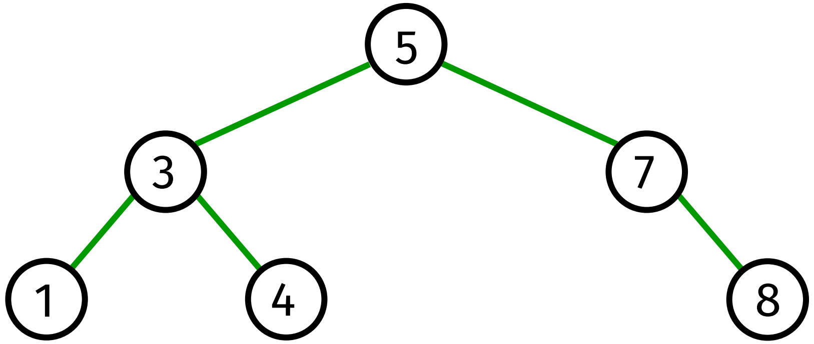
- if y is in the left subtree of x, then $y.\text{key} \leq x.\text{key}$
- if y is in the right subtree of x, then $y.\text{key} \geq x.\text{key}$



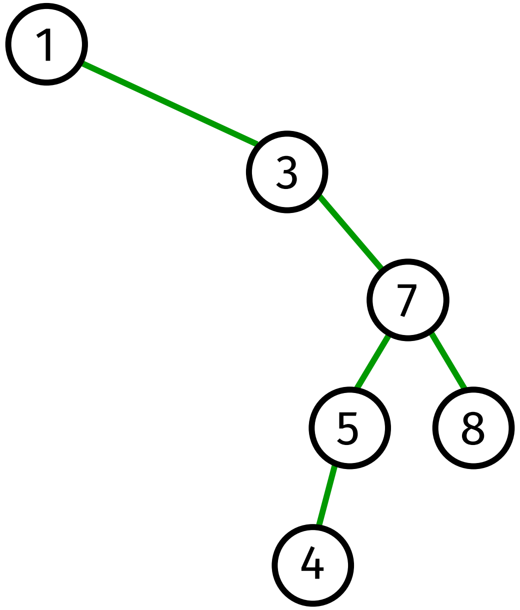
Keys don't have to be unique …

can use stricter property, take care when balancing

Binary search trees



height $h = 2$



height $h = 4$

Binary-search-tree property

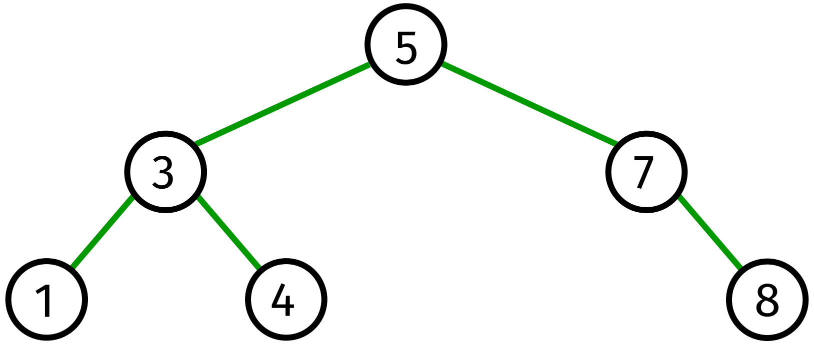
- if y is in the left subtree of x, then $y.\mathrm{key} \leq x.\mathrm{key}$
- if y is in the right subtree of x, then $y.\mathrm{key} \geq x.\mathrm{key}$

Tree walks

Binary search trees are recursive structures
➡ recursive algorithms often work well

TreeWalk(x)

RecurseLeft()
RecurseRight()
Do something



PreorderTreeWalk

Do something
RecurseLeft()
RecurseRight()

InorderTreeWalk

RecurseLeft()
Do something
RecurseRight()

PostorderTreeWalk

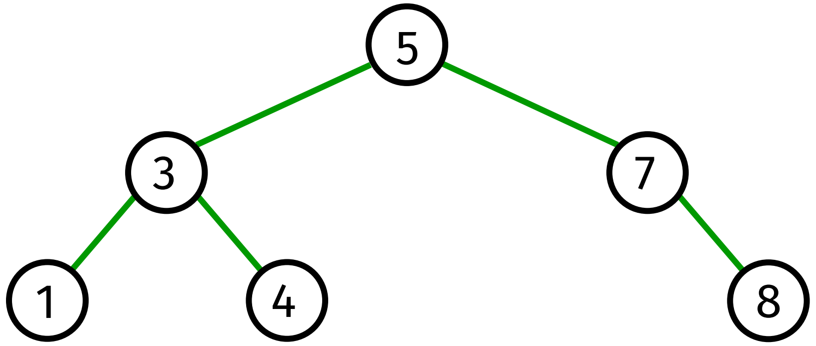
RecurseLeft()
RecurseRight()
Do something

Inorder tree walk

Example: print all keys in order using an inorder tree walk

InorderTreeWalk(x)

```
if x ≠ NIL
    InorderTreeWalk(x.left)
    print x.key
    InorderTreeWalk(x.right)
```



Correctness: follows by induction from the binary search tree property

Running time?

Intuitively, $O(n)$ time for a tree with n nodes, since we visit and print each node once.

Inorder tree walk

InorderTreeWalk(x)

```
if x ≠ NIL
    InorderTreeWalk(x.left)
    print x.key
    InorderTreeWalk(x.right)
```

Theorem

If x is the root of an n-node subtree, then the call InorderTreeWalk(x) takes $\Theta(n)$ time.

Proof:

- $T(n)$ takes small, constant amount of time on empty subtree

  $T(0) = c$ for some positive constant c

- for $n > 0$ assume that left subtree has k nodes, right subtree $n - k - 1$

  $T(n) = T(k) + T(n - k - 1) + d$ for some positive constant d

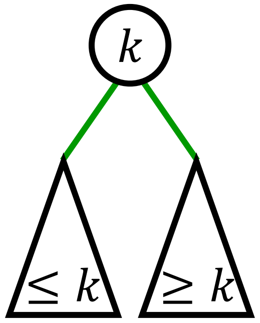
use substitution method … to show: $T(n) = (c + d)n + c$

Querying a binary search tree

TreeSearch(x, k)

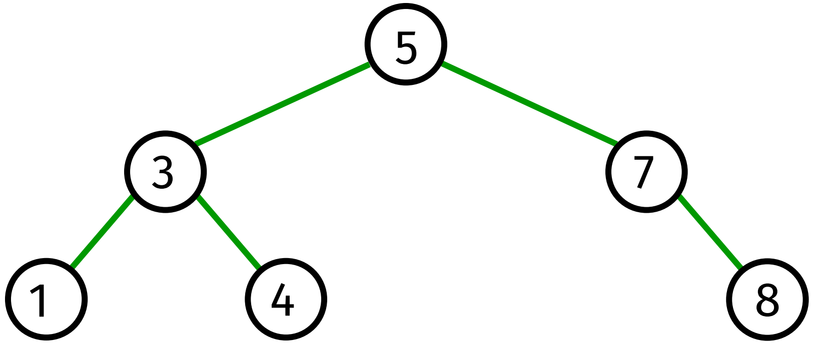
if $x == NIL$ or $k = x.\mathrm{key}$: **return** x
if $k < x.\mathrm{key}$
  return TreeSearch($x.\mathrm{left}$, k)
else
  return TreeSearch($x.\mathrm{right}$, k)

Binary-search-tree property



Initial call: TreeSearch($T.\mathrm{root}$, k)

- TreeSearch($T.\mathrm{root}$, 4)
- TreeSearch($T.\mathrm{root}$, 2)



Running time:

Θ(length of search path)

worst case $\Theta(h)$

Querying a binary search tree – iterative

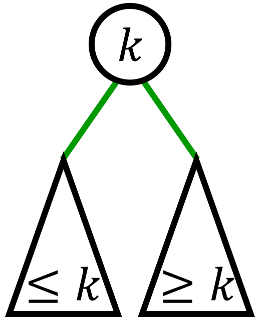
TreeSearch(x, k)

if $x == NIL$ or $k = x.\mathrm{key}$: **return** x
if $k < x.\mathrm{key}$
    return TreeSearch($x.\mathrm{left}$, k)
else
    return TreeSearch($x.\mathrm{right}$, k)

IterativeTreeSearch(x, k)

while $x \neq NIL$ and $k \neq x.\mathrm{key}$
    if $k < x.\mathrm{key}$: $x = x.\mathrm{left}$
    else: $x = x.\mathrm{right}$
return x

Binary-search-tree property



the iterative version is more efficient on most computers

Minimum and maximum

Binary-search-tree property guarantees that

- the minimum key is located in the leftmost node
- the maximum key is located in the rightmost node

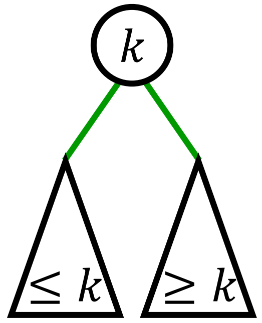
TreeMinimum(x)

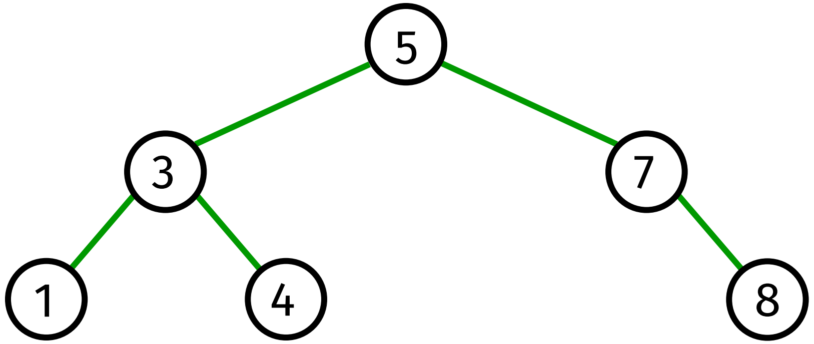
while $x.\text{left} \neq NIL$
  $x = x.\text{left}$
return x

TreeMaximum(x)

while $x.\text{right} \neq NIL$
  $x = x.\text{right}$
return x

Binary-search-tree property



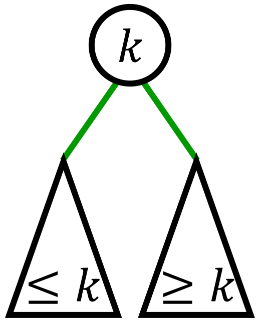


Minimum and maximum

Binary-search-tree property guarantees that

- the minimum key is located in the leftmost node
- the maximum key is located in the rightmost node

Binary-search-tree property



TreeMinimum(x)

while $x.\mathrm{left} \neq NIL$
    $x = x.\mathrm{left}$
return x

TreeMaximum(x)

while $x.\mathrm{right} \neq NIL$
    $x = x.\mathrm{right}$
return x

Running time?

Both procedures visit nodes on a downward path from the root
➡ $O(h)$ time

Successor and predecessor

Assume that all keys are distinct

Successor of a node x:
node y such that $y.\mathrm{key}$ is the smallest key $> x.\mathrm{key}$
(if x has the largest key, then we say x's successor is NIL)

We can find y based entirely on the tree structure, no key comparisons are necessary …

Successor and predecessor

Successor of a node x

node y such that $y.\mathrm{key}$ is the smallest key $> x.\mathrm{key}$

Two cases:

1. x has a non-empty right subtree
   ➡ x's successor is the minimum in x's right subtree

2. x has an empty right subtree
   ➡ x's successor y is the node of which x is the predecessor
   (x is the maximum in y's left subtree)

   as long as we move to the left up the tree (move up through right children), we're visiting smaller keys …

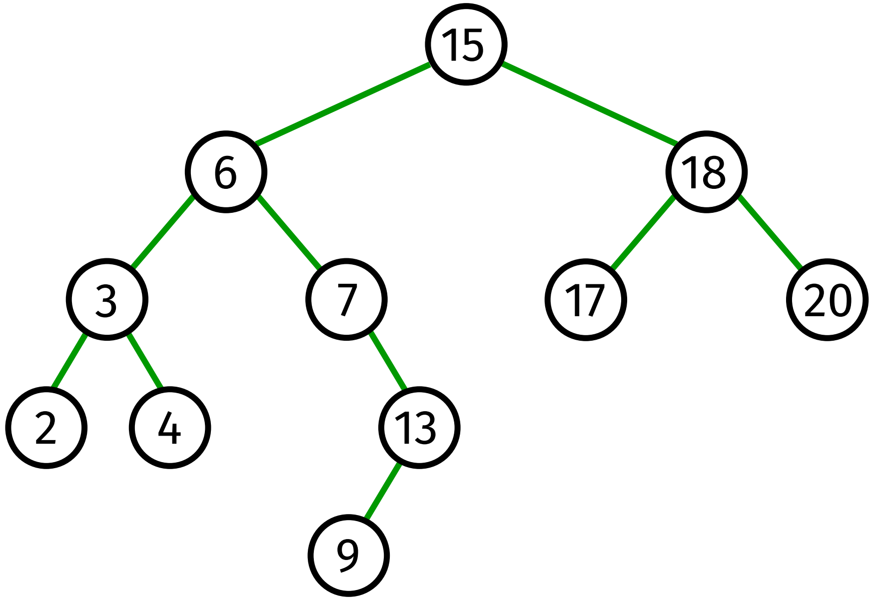
Successor and predecessor

TreeSuccessor(x)

if $x.\text{right} \neq NIL$
    return TreeMinimum($x.\text{right}$)
$y = x.p$
while $y \neq NIL$ and $x = y.\text{right}$
    $x = y$
    $y = x.p$
return y

TreePredecessor is symmetric

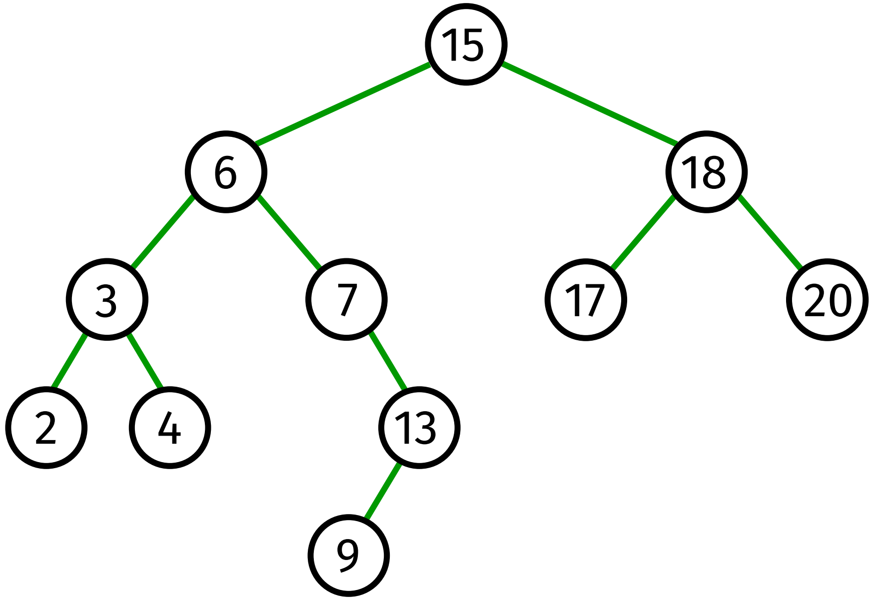
- Successor of 15?
- Successor of 6?
- Successor of 4?
- Predecessor of 6?



Successor and predecessor

TreeSuccessor(x)

if $x.\mathrm{right} \neq NIL$
    return TreeMinimum($x.\mathrm{right}$)
$y = x.p$
while $y \neq NIL$ and $x = y.\mathrm{right}$
    $x = y$
    $y = x.p$
return y



TreePredecessor is symmetric

Running time? $O(h)$

Insertion

TreeInsert(T, z)

$y = NIL$
$x = T.\mathrm{root}$
while $x \neq NIL$
    $y = x$
    if $z.\mathrm{key} < x.\mathrm{key}$: $x = x.\mathrm{left}$
    else: $x = x.\mathrm{right}$
$z.p = y$
if $y == NIL$
    $T.\mathrm{root} = z$
else
    if $z.\mathrm{key} < y.\mathrm{key}$: $y.\mathrm{left} = z$
    else: $y.\mathrm{right} = z$

to insert value v, insert node z with $z.\mathrm{key} = v$, $z.\mathrm{left} = NIL$, and $z.\mathrm{right} = NIL$

traverse tree down to find correct position for z

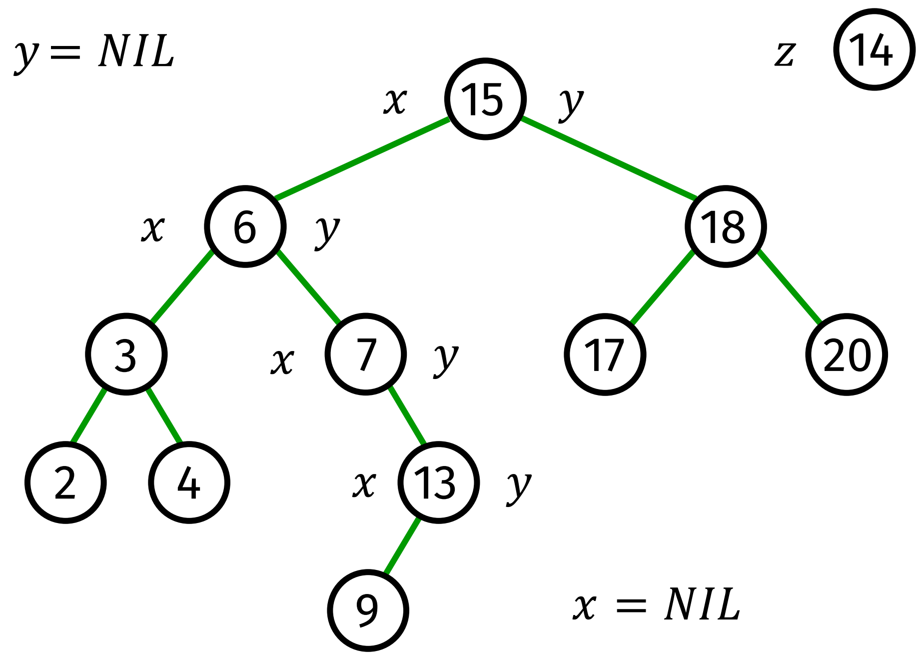
Insertion

TreeInsert(T, z)

$y = NIL$
$x = T.\mathrm{root}$
while $x \neq NIL$
    $y = x$
    if $z.\mathrm{key} < x.\mathrm{key}$: $x = x.\mathrm{left}$
    else: $x = x.\mathrm{right}$
$z.p = y$
if $y == NIL$
    $T.\mathrm{root} = z$
else
    if $z.\mathrm{key} < y.\mathrm{key}$: $y.\mathrm{left} = z$
    else: $y.\mathrm{right} = z$

to insert value v, insert node z with $z.\mathrm{key} = v$, $z.\mathrm{left} = NIL$, and $z.\mathrm{right} = NIL$

traverse tree down to find correct position for z



Insertion

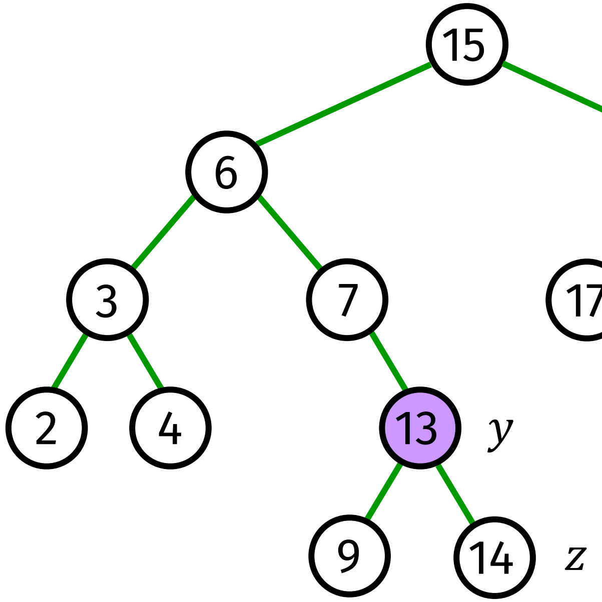
TreeInsert(T, z)

$y = NIL$
$x = T.\text{root}$
while $x \neq NIL$
    $y = x$
    if $z.\text{key} < x.\text{key}$: $x = x.\text{left}$
    else: $x = x.\text{right}$
$z.p = y$
if $y == NIL$
    $T.\text{root} = z$
else
    if $z.\text{key} < y.\text{key}$: $y.\text{left} = z$
    else: $y.\text{right} = z$

Running time? $O(h)$

to insert value v, insert node z with $z.\text{key} = v$, $z.\text{left} = NIL$, and $z.\text{right} = NIL$

traverse tree down to find correct position for z



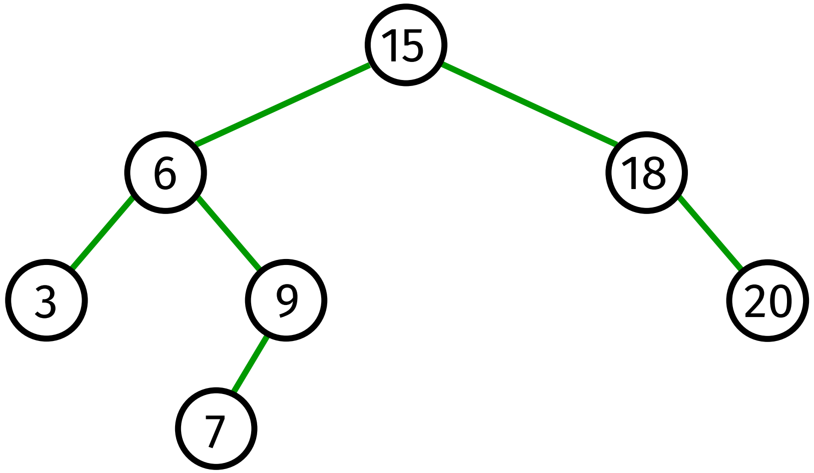
Deletion

We want to delete node z

TreeDelete has three cases:

z has no children

- delete z by having z's parent point to NIL, instead of to z



Deletion

We want to delete node z

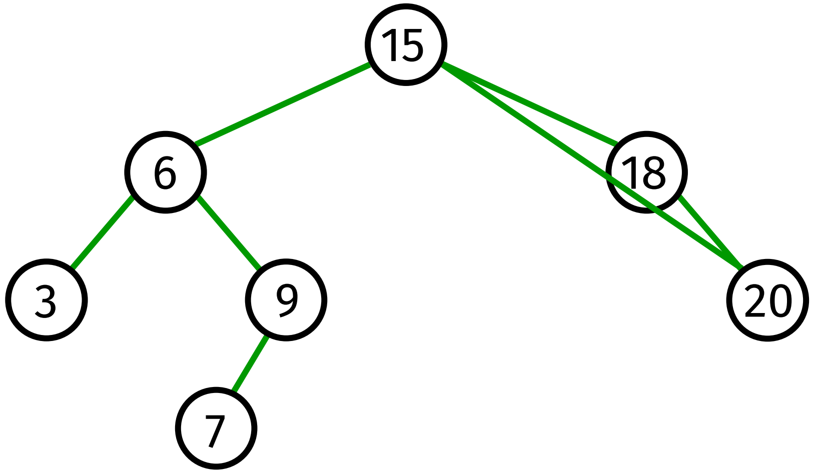
TreeDelete has three cases:

z has no children

- delete z by having z's parent point to NIL, instead of to z

z has one child

- delete z by having z's parent point to z's child, instead of to z



Deletion

We want to delete node z

TreeDelete has three cases:

z has no children

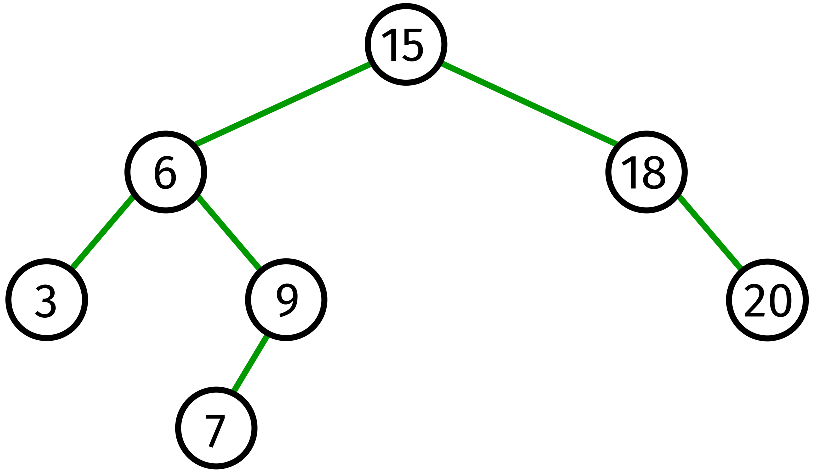
- delete z by having z's parent point to NIL, instead of to z

z has one child

- delete z by having z's parent point to z's child, instead of to z

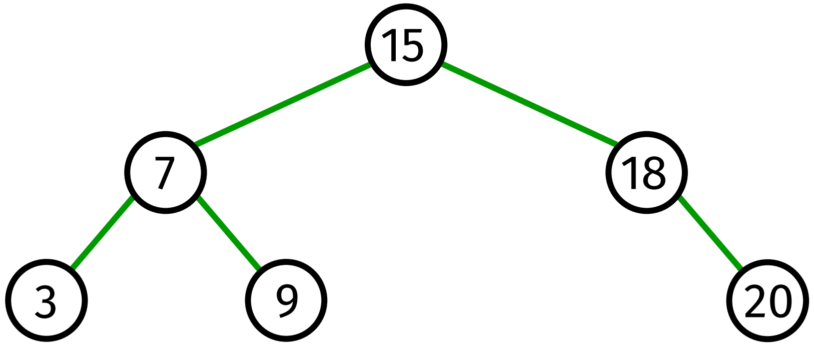
z has two children

- z's successor y has either no or one child
  delete y from the tree and replace z's key and satellite data with y's



Deletion

We want to delete node z



TreeDelete has three cases:

z has no children

- delete z by having z's parent point to NIL, instead of to z

z has one child

- delete z by having z's parent point to z's child, instead of to z

z has two children

- z's successor y has either no or one child
  delete y from the tree and replace z's key and satellite data with y's

Running time?

$O(h)$

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)* $\Theta(\log n)$

Balanced Search Trees

Balanced Search trees

There are many methods to balance a search tree.

by weight
for each node the number of nodes in the left and the right subtree
is approximately equal

by height
for each node the height of the left and the right subtree
is approximately equal

by degree
all leaves have the same depth, but the degree of the nodes differs
(hence not a binary search tree)

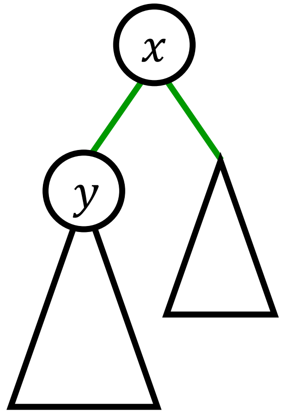
Weight-balanced search trees

BB[α]-tree

binary search tree where for each pair of nodes x, y, with y being a child of x we have

$$\alpha \leq \frac{\text{number of leaves in subtree rooted at } y}{\text{number of leaves in subtree rooted at } x} \leq 1 - \alpha$$

where α is a positive constant with $\alpha \leq 1/3$



For the height $h(n)$ it holds that $h(n) \leq h((1-\alpha)n) + 1$

Master theorem: $h(n) = \Theta(\log n)$

Ideally: α as close as possible to $1/3$

But: $\alpha = 1/3$ gives too little flexibility for updates

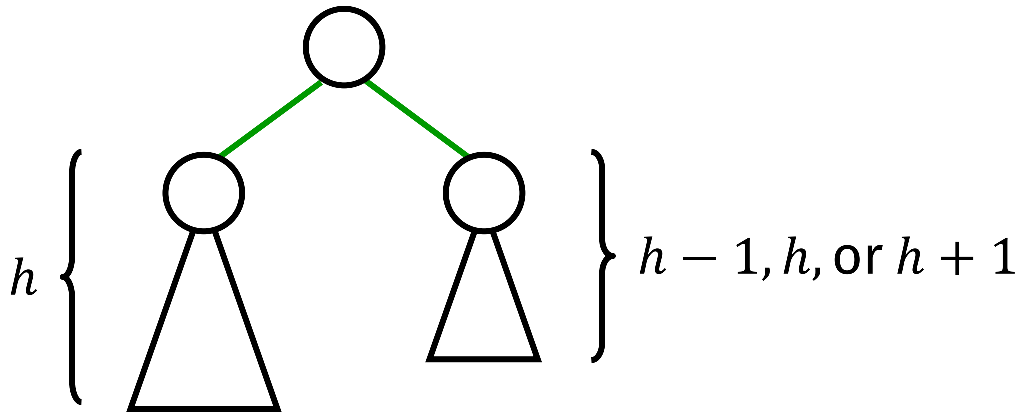
α just smaller than $1/3$ works fine

Height-balanced search trees

AVL-tree

binary search tree where for each node

| height left subtree – height right subtree | ≤ 1



Theorem

An AVL-tree with n nodes has height $\Theta(\log n)$.

Height-balanced search trees

Theorem

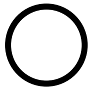
An AVL-tree with n nodes has height $\Theta(\log n)$.

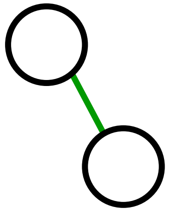
Proof

Let $n(h) =$ minimum number of nodes in an AVL-tree of height h

Claim: $n(h) \geq 2^{h/2}$

Proof of Claim: induction on h

$h = 0$  $n(0) = 1$ ✓

$h = 1$  $n(1) = 2$ ✓

Height-balanced search trees

Theorem

An AVL-tree with n nodes has height $\Theta(\log n)$.

Proof

Let $n(h)$ = minimum number of nodes in an AVL-tree of height h

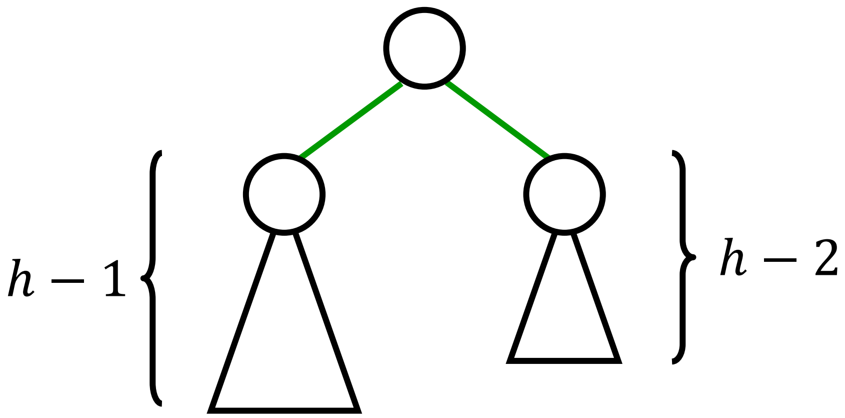
Claim: $n(h) \geq 2^{h/2}$

Proof of Claim: induction on h

$h \geq 2$

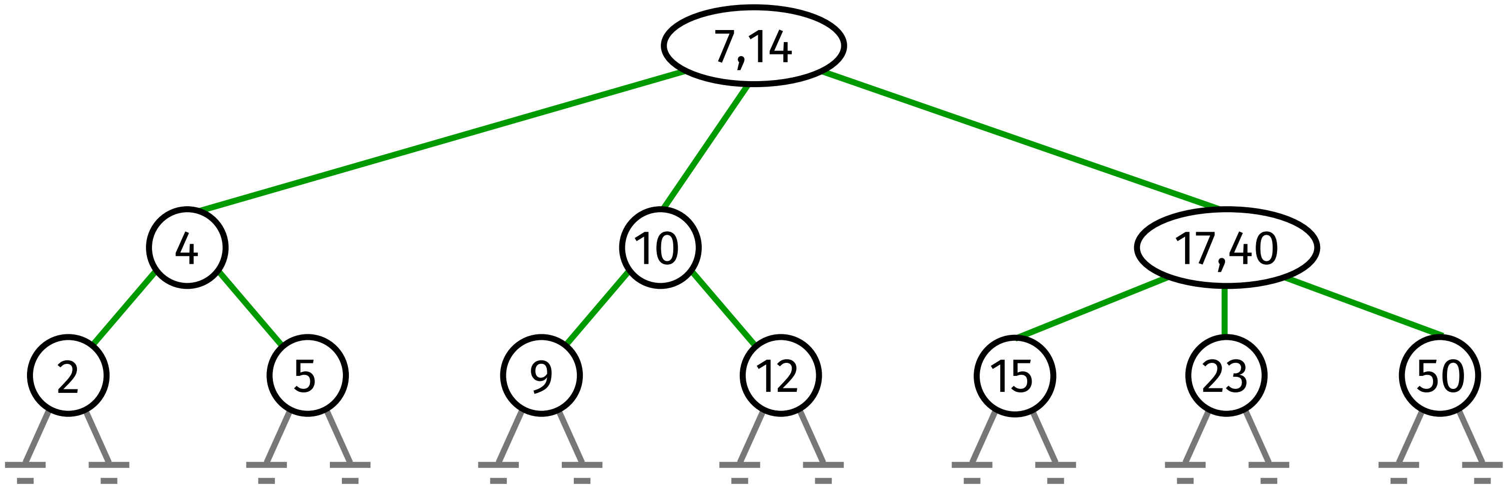
$$\begin{aligned} n(h) &\geq 1 + n(h-1) + n(h-2) \\ &\geq 2n(h-2) \\ &\geq 2 \cdot 2^{(h-2)/2} \\ &= 2^{h/2} \end{aligned}$$

■



Degree-balanced trees

(2, 3)-tree
: search tree where all leaves have the same depth and internal nodes have degree 2 or 3



Theorem
: A (2, 3)-tree with n nodes has height $\Theta(\log n)$.

Red-black Trees

Another height-balanced search tree

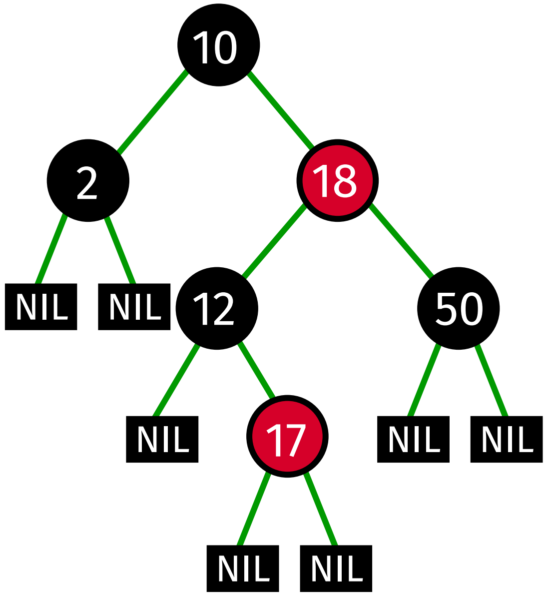
Red-black trees

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.



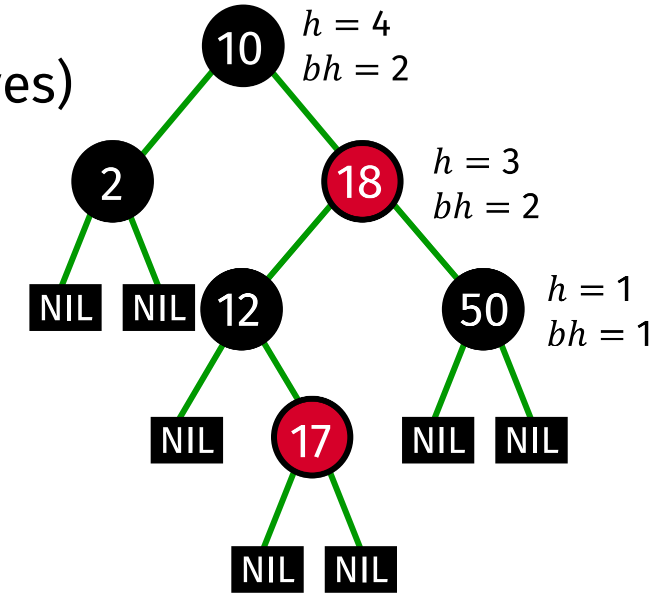
Red-black trees: height

height of a node

number of edges on a longest path to a leaf

black-height of a node x

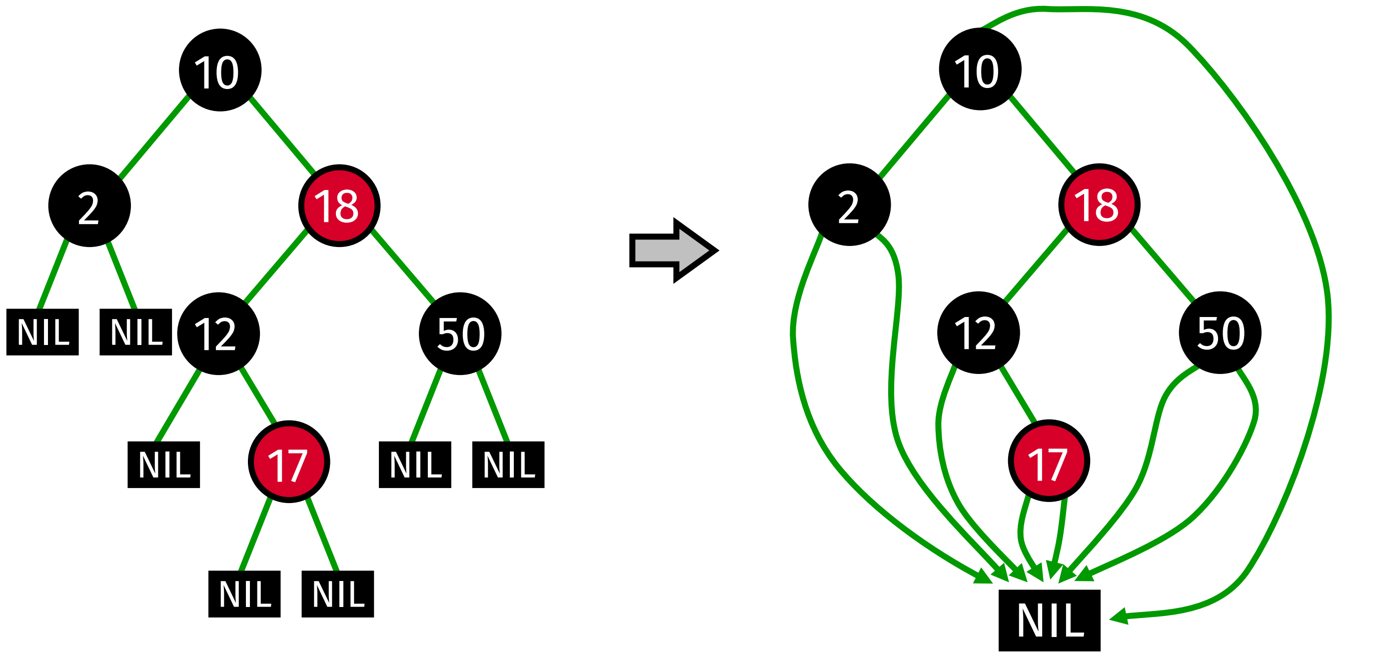
$bh(x)$ is the number of black nodes (including NIL leaves) on the path from x to leaf, not counting x.



Red-black properties

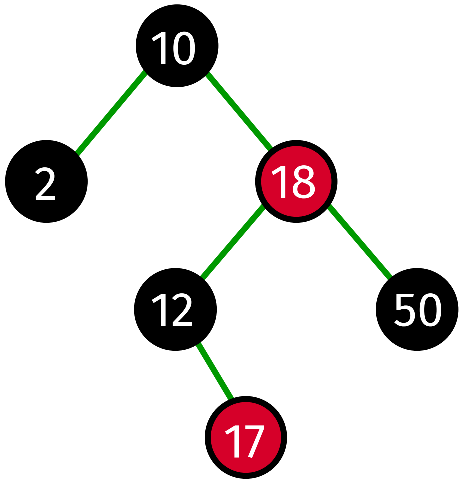
1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
   (Hence no two reds in a row on a simple path from the root to a leaf)
5. For each node, all paths from the node to descendant leaves contain the same number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

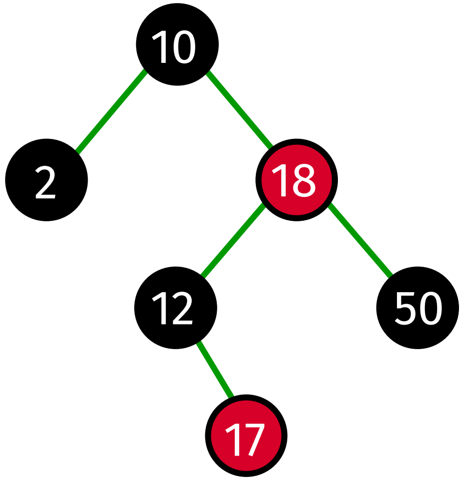
Red-black trees: implementation detail



It is useful to replace each NIL by a single sentinel $T.\mathrm{nil}$ which is always black.
The root's parent is also the sentinel.

Red-black trees: implementation detail





It is useful to replace each NIL by a single sentinel $T.\mathrm{nil}$ which is always black.
The root's parent is also the sentinel.

NIL will not (always) be drawn on the following slides

Red-black trees

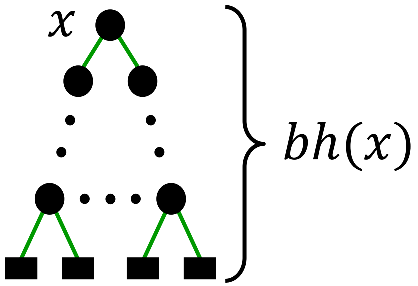
Lemma

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

Proof

the subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes

“smallest” subtree with black-height $bh(x)$



the complete tree has n internal nodes

➡ $2^{bh(T.\text{root})} - 1 \leq n$

➡ $bh(T.\text{root}) \leq \log(n+1)$

$$\begin{aligned}\text{height}(T) &= \text{number of edges on a longest path to a leaf} \\ &\leq 2 \cdot bh(T.\text{root}) \\ &\leq 2\log(n+1) \qquad \blacksquare\end{aligned}$$

Balanced binary search trees

Advantages of balanced binary search trees

over linked lists

efficient search in $\Theta(\log n)$ time

over sorted arrays

efficient insertion and deletion in $\Theta(\log n)$ time

over hash tables

can find successor and predecessor efficiently in $\Theta(\log n)$ time