Algorithms and Datastructures

Lecture 8

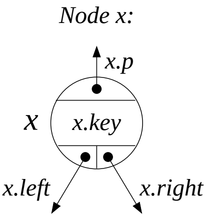
Manfred Jaeger

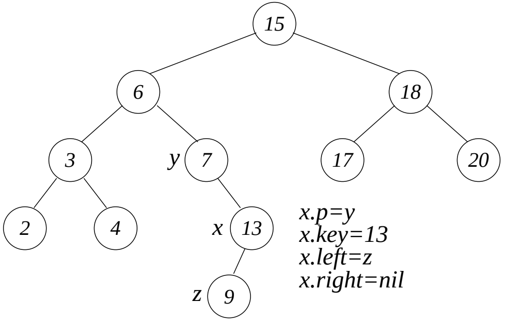
AALBORG UNIVERSITET

Binary Search Trees

Datastructure for storing a set of objects x with key attribute. Keys come from an ordered set (need not be integers).

A Binary Tree is composed of *node* elements:





A tree is a *node* object that defines the root of the tree.

Binary Search Tree Property

All keys in the *left subtree* of x are $\leq x.key$, and all keys in the *right subtree* of x are $\geq x.key$

BSTs support operations

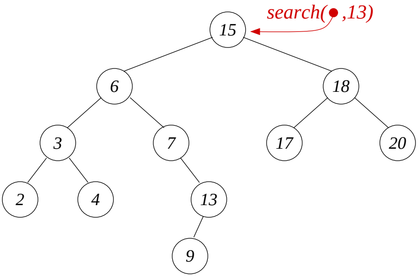
Object *search(Key k)*
void *insert(Object o)*
void *delete(Object o)*
Object *minimum()*
Object *maximum()*
Object *successor(Object o)*
Object *predecessor(Object o)*

☞ BSTs implement ADTs *Dictionary*, *Priority Queue*, . . .

input : Root node x of BST, Integer k
output: Node y in tree rooted at x with $y.key = k$ if such y exists, otherwise *nil*

TREESEARCH(x, k)

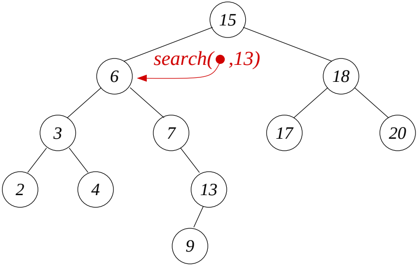
if $x == nil$ *or* $x.key == k$ **then**
    return (x)
if $k < x.key$ **then**
    return (TREESEARCH$(x.left, k)$)
else
    return (TREESEARCH$(x.right, k)$)



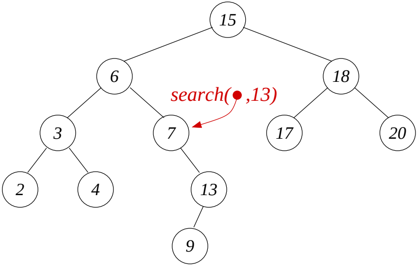
input : Root node x of BST, Integer k
output: Node y in tree rooted at x with $y.key = k$ if such y exists, otherwise *nil*

TREESEARCH(x, k)

```
if x == nil or x.key == k then
    return (x)
if k < x.key then
    return (TREESEARCH(x.left, k))
else
    return (TREESEARCH(x.right, k))
```



input : Root node x of BST, Integer k
output: Node y in tree rooted at x with $y.key = k$ if such y exists, otherwise *nil*



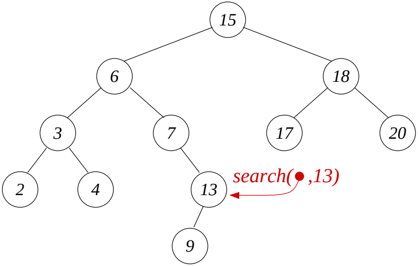
TREESEARCH(x, k)

if $x == nil$ *or* $x.key == k$ **then**
  return (x)
if $k < x.key$ **then**
  return (TREESEARCH($x.left, k$))
else
  return (TREESEARCH($x.right, k$))

input : Root node x of BST, Integer k
output: Node y in tree rooted at x with $y.key = k$ if such y exists, otherwise *nil*

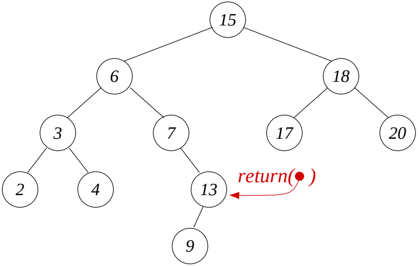
TREESEARCH(x, k)

if $x == nil$ *or* $x.key == k$ **then**
    return (x)
if $k < x.key$ **then**
    return (TREESEARCH($x.left, k$))
else
    return (TREESEARCH($x.right, k$))



input : Root node x of BST, Integer k
output: Node y in tree rooted at x with $y.key = k$ if such y exists, otherwise *nil*

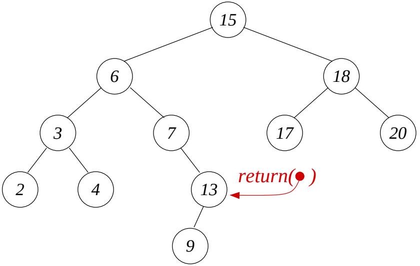
```
TREESEARCH(x, k)
if x == nil or x.key == k then
    return (x)
if k < x.key then
    return (TREESEARCH(x.left, k))
else
    return (TREESEARCH(x.right, k))
```



input : Root node x of BST, Integer k
output: Node y in tree rooted at x with $y.key = k$ if such y exists, otherwise *nil*

TREESEARCH(x, k)

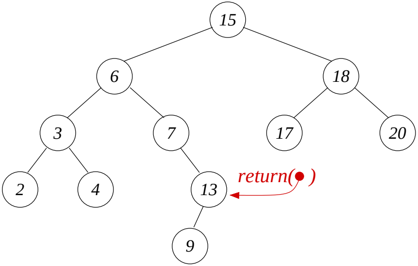
```
if x == nil or x.key == k then
    return (x)
if k < x.key then
    return (TREESEARCH(x.left, k))
else
    return (TREESEARCH(x.right, k))
```



Complexity: $O(\textit{height of tree})$

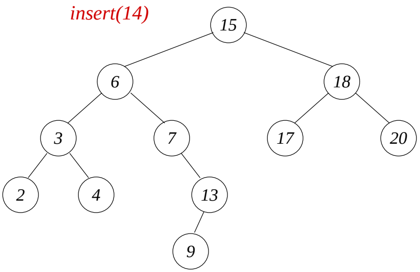
input : Root node x of BST, Integer k
output: Node y in tree rooted at x with $y.key = k$ if such y exists, otherwise *nil*

```
TreeSearch(x, k)
if x == nil or x.key == k then
    return (x)
if k < x.key then
    return (TreeSearch(x.left, k))
else
    return (TreeSearch(x.right, k))
```



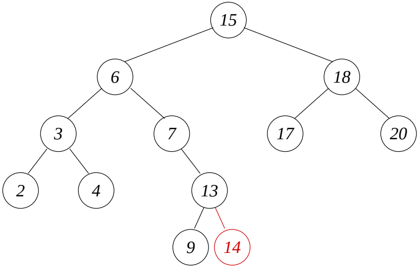
Complexity: O(*height of tree*)

Also in O(*height*): *minimum, maximum, successor, predecessor*



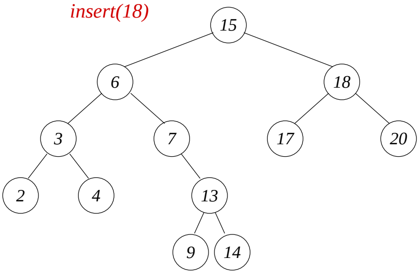
Inserting x:

- search for $x.key$ until *nil* encountered
- replace the *nil* with a new node for key x
- when $x.key$ already contained in the tree: continue “search” in right sub-tree



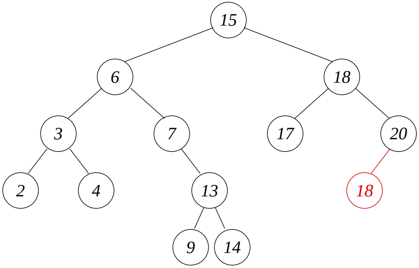
Inserting x:

- search for $x.key$ until *nil* encountered
- replace the *nil* with a new node for key x
- when $x.key$ already contained in the tree: continue “search” in right sub-tree



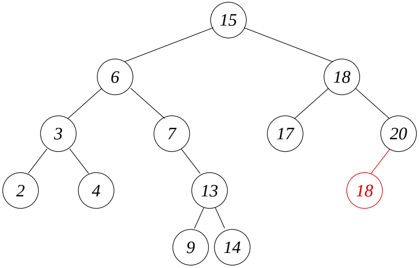
Inserting x:

- search for x.*key* until *nil* encountered
- replace the *nil* with a new node for key x
- when x.*key* already contained in the tree: continue "search" in right sub-tree



Inserting x:

- search for x.*key* until *nil* encountered
- replace the *nil* with a new node for key x
- when x.*key* already contained in the tree: continue “search” in right sub-tree

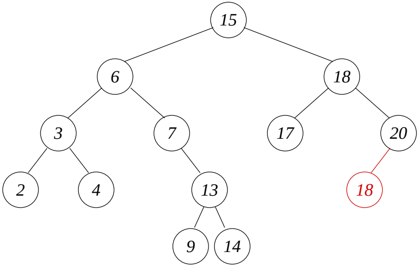


Inserting x:

- search for $x.key$ until *nil* encountered
- replace the *nil* with a new node for key x
- when $x.key$ already contained in the tree: continue "search" in right sub-tree

Alternative:

- Insertion (a.k.a. *put*) takes arguments *key* k and *value* v
- When node x with $x.key = k$ found in search, update $x.value$ with v.



Inserting x:

- search for $x.key$ until *nil* encountered
- replace the *nil* with a new node for key x
- when $x.key$ already contained in the tree: continue "search" in right sub-tree

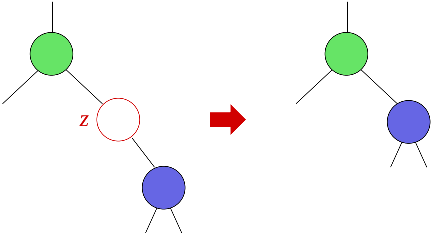
Alternative:

- Insertion (a.k.a. *put*) takes arguments *key* k and *value* v
- When node x with $x.key = k$ found in search, update $x.value$ with v.

Complexity: $O(h)$

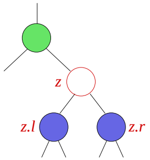
Operation: *delete*(z) (z a node)

Case 1: z has at most one child:

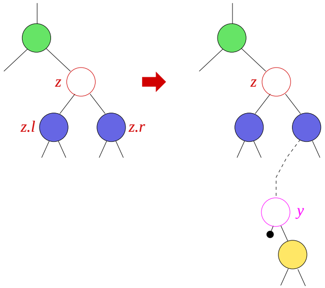


- z's child (if there is one) takes z's place in $z.p$

Case 2: z has two children:

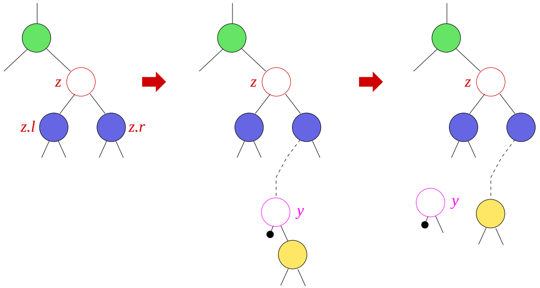


Case 2: z has two children:



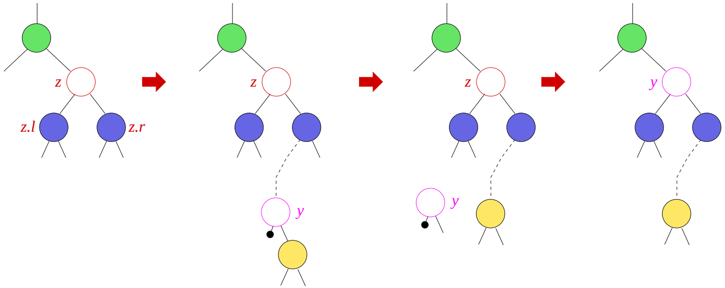
- Find *y=TreeMinimum(z.r)*. Then $y.l = nil$!

Case 2: z has two children:



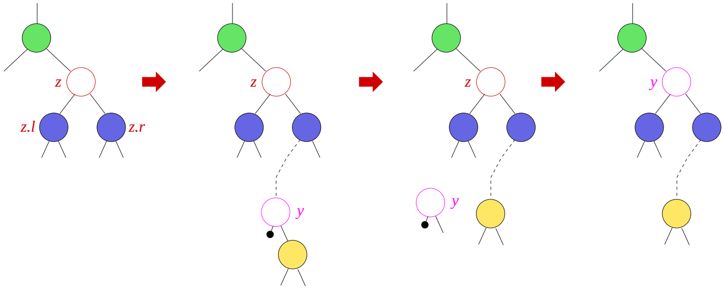
- Find y=*TreeMinimum(z.r)*. Then $y.l = nil$!
- "Delete y" (case 1!)

Case 2: z has two children:



- Find y=*TreeMinimum(z.r)*. Then $y.l = nil$!
- "Delete y" (case 1!)
- Substitute y for z

Case 2: *z* has two children:



- Find *y=TreeMinimum(z.r)*. Then $y.l = nil$!
- "Delete *y*" (case 1!)
- Substitute *y* for *z*

Complexity: $O(h)$ (finding *TreeMinimum*)

☞ All operations performed by re-directing *left*, *right* and *parent* references (no copying/editing of *node.key* or *node.value* fields)

The *height* of a binary tree containing n nodes is between

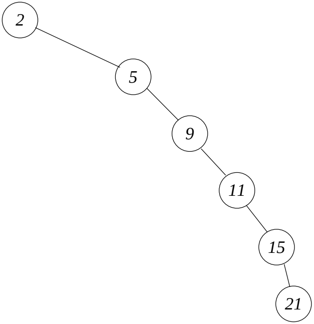
a 1 and n
b n and $n \lg n$
c $\lg n$ and $n \lg n$
d $\lg n$ and n
e 1 and $\lg n$

(select sharpest bounds!)

All operations *search*, *insert*, *delete* are $O(h)$. In the worst case $h = n$:

BST constructed from insertion sequence

$2, 5, 9, 11, 15, 21$:



Expected Height

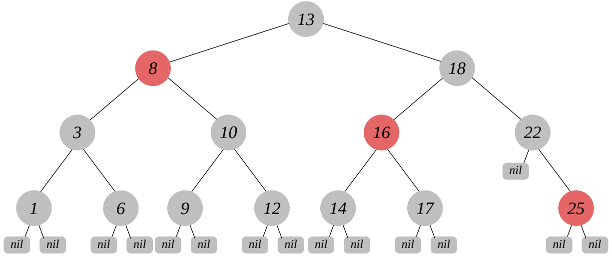
- When BST is constructed for n keys that arrive for insertion in random order, then the expected height of the tree is $O(\lg n)$.
- Compare: expected height of QUICKSORT recursion tree when input array is in random order

Red-Black Trees

Idea: ensure that binary search tree is approximately **balanced** regardless of the insertion order of keys

- Query operations *search*, *maximum*, *minimum*, *successor*, *predecessor* as before in time $O(h)$.
- Ensure that $h = \Theta(\lg n)$ by modifying *insert* and *delete*

Red-Black Trees are one (out of several) implementations of this idea. They are the basis for the Java `TreeMap` class.



- Only internal nodes are associated with keys. Leaves have value *nil*
- Nodes have a *color* attribute with values *red* or *black*
- The coloring satisfies the following conditions:
  - The root is black
  - All leaves are black
  - Both children of a red node are black
  - For all nodes *x*: all paths from *x* to a leaf contain the same number of black nodes

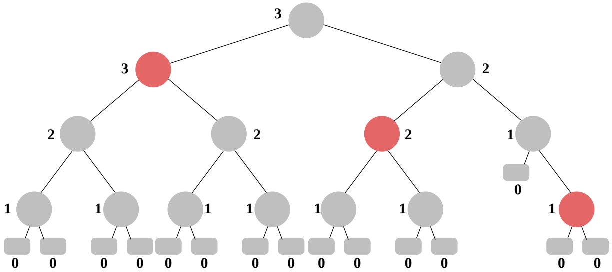
☞ the *nil* leaves are mainly a conceptual tool for simplifying the analysis; can be represented by a single *nil sentinel element* in implementation.

Lemma

A RB-tree with n internal nodes has height at most $2\lg(n+1)$.

Black-Height

For node x in RB-tree: $bh(x)$ = number of black nodes on a path from x to a leaf (not including x)



Nodes labeled with black-height

Claim: A subtree rooted at a node x contains at least $2^{bh(x)} - 1$ internal nodes.

Proof of claim by induction on the *height* (!) of x:

Base case: $h(x) = 0$: x is a leaf, $bh(x) = 0$, and $2^{bh(x)} - 1 = 0$.

Induction step: Let $h(x) > 0$. Induction hypothesis applied to $x.l$ and $x.r$ implies that the left and right sub-tree of x contain at least $2^{bh(x)-1} - 1$ internal nodes each. Thus, the total number of internal nodes of sub-tree rooted at x is at least

$$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$$

Claim proved! Now continue:

For the root r of the tree:

$$bh(r) \geq h(r)/2.$$

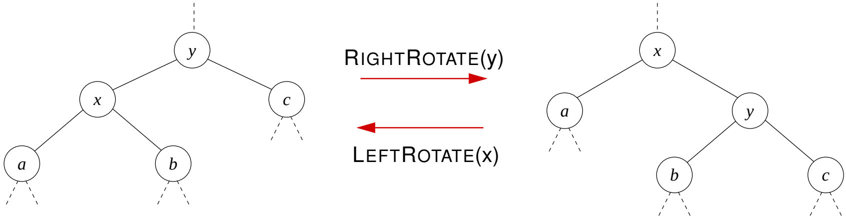
Therefore:

$$n \geq 2^{bh(r)} - 1 \geq 2^{h(r)/2} - 1,$$

and

$$\lg(n+1) \geq h(r)/2.$$

Rotations



- Constant time operations that locally redirect child/parent pointers
- Preserves binary search tree property for keys
- Does not take coloring into account: can destroy or establish RB tree conditions

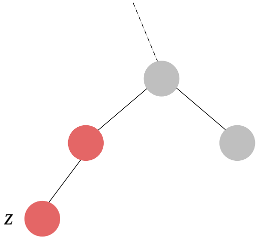
To insert new node z with key k:

- Perform a standard BST *insert*
- Color z red
- Call RB-INSERT-FIXUP(z) to re-establish RB tree properties

What can have gone wrong to require fixup?

- Root no longer black: only happens when z is the first node inserted.
- Leaves no longer black? Cannot happen!
- Red node has red child: can happen.
- Unequal number of black nodes on paths starting from a node x? Cannot happen!

The problem that needs to be fixed:



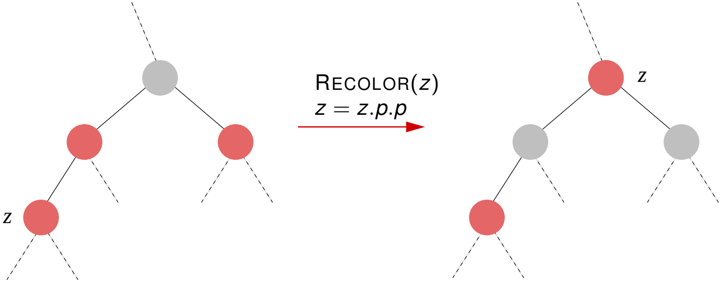
<u>FIXUP(z)</u>

while *z!= T.root & z.p.color == red* **do**
  if *z's uncle is red* **then**
    RECOLOR(z)
    $z = z.p.p$
  else
    if *linear-config(z)* **then**
      ROTATE-TO-FIX(z)
    if *zig-zag-config(z)* **then**
      ROTATE-TO-LINEAR-CONFIG(z)
$T.root.color = black$

Invariant maintained by the **while** loop:

- z is red
- the only possible violations of the RB tree properties are:
  - z is the root and red, or
  - z is not the root and $z.p$ is also red

Case: z has red uncle



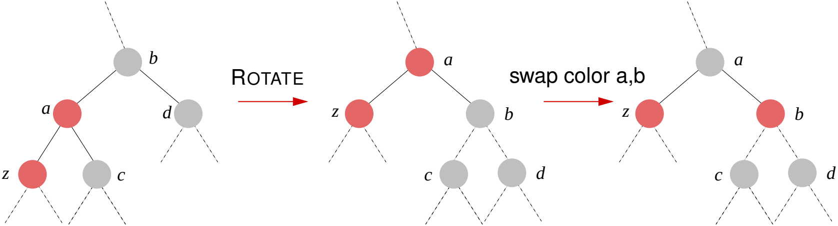
Invariant: maintained!

Progress: violating node z moves closer to the root!

Case: *linear-config(z)*

- z's uncle is black
- both z and $z.p$ are left (or both right) children

ROTATE-TO-FIX:



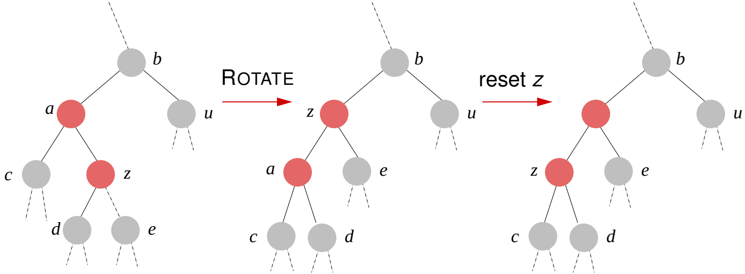
Invariant: no more violations!

Progress: done!

Case: *zig-zag-config(z)*

- z's uncle is black
- z is right, and $z.p$ left child (or vice-versa)

ROTATE-TO-LINEAR-CONFIG:



Invariant: maintained!

Progress: one step left to do!

Summary

```
FIXUP(z)
while z != T.root & z.p.color == red do
    if z's uncle is red then
        RECOLOR(z)
        z = z.p.p
    else
        if linear-config(z) then
            ROTATE-TO-FIX(z)
        if zig-zag-config(z) then
            ROTATE-TO-LINEAR-CONFIG(z)
T.root.color = black
```

Invariant maintained by the **while** loop:

- z is red
- the only possible violations of the RB tree properties are:
  - z is the root and red, or
  - z is not the root, and $z.p$ is also red

Progress: Each iteration of the **while** loop either moves z closer to the root, or leads to termination within at most one more iteration.

Complexity: One iteration of **while** loop: $O(1)$. At most $O(h)$ iterations.

Similar strategy as for *insert*:

- Perform standard BST *delete*
- Fix violations of RB tree properties that may have occurred

Possible violations now also include the property that all paths from a node contain the same number of black nodes.

Idea: Represent this as a local violation of RB tree properties by assigning a node a *double black* or *red and black* value.

Then same strategy as for *insert*:

- locally resolve the violation, and
- move violating node up the tree (or terminate)

Complexity: $O(h)$