Algorithms and Data Structures for Mathematicians

Lecture 4: Trees

Peter Kostolányi
`kostolanyi at fmph and so on`
Room M-258

19 October 2017

Binary Trees and Their Representation

From the last lecture:

Binary Trees and Their Representation

From the last lecture:

- Heaps: "almost-complete" binary trees satisfying some property

Binary Trees and Their Representation

From the last lecture:

- Heaps: "almost-complete" binary trees satisfying some property
- "Almost-completeness" of heaps makes arrays a convenient choice for their representation

Binary Trees and Their Representation

From the last lecture:

- Heaps: "almost-complete" binary trees satisfying some property
- "Almost-completeness" of heaps makes arrays a convenient choice for their representation

For general binary trees:

Binary Trees and Their Representation

From the last lecture:

- Heaps: "almost-complete" binary trees satisfying some property
- "Almost-completeness" of heaps makes arrays a convenient choice for their representation

For general binary trees:

- Representation by arrays would not be so simple

Binary Trees and Their Representation

From the last lecture:

- Heaps: "almost-complete" binary trees satisfying some property
- "Almost-completeness" of heaps makes arrays a convenient choice for their representation

For general binary trees:

- Representation by arrays would not be so simple
- We shall use an approach similar to linked lists instead

Binary Trees and Their Representation

From the last lecture:

- Heaps: "almost-complete" binary trees satisfying some property
- "Almost-completeness" of heaps makes arrays a convenient choice for their representation

For general binary trees:

- Representation by arrays would not be so simple
- We shall use an approach similar to linked lists instead
- Binary trees will simply be "branching" linked lists

Binary Trees and Their Representation

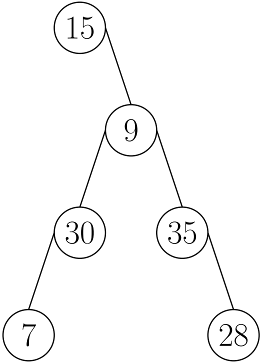
From the last lecture:

- Heaps: "almost-complete" binary trees satisfying some property
- "Almost-completeness" of heaps makes arrays a convenient choice for their representation

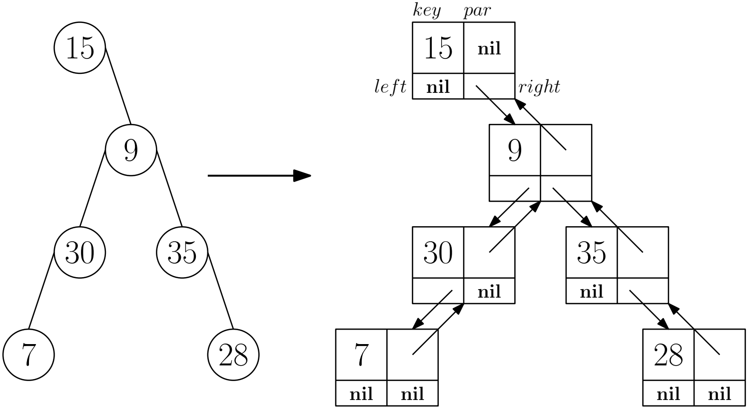
For general binary trees:

- Representation by arrays would not be so simple
- We shall use an approach similar to linked lists instead
- Binary trees will simply be "branching" linked lists
- Generalisation to k-ary trees is straightforward

Binary Trees and Their Representation



Binary Trees and Their Representation



Binary Search Trees

- Binary trees satisfying some condition (other than for heaps)

Binary Search Trees

- Binary trees satisfying some condition (other than for heaps)
- In general neither complete, nor “almost-complete”

Binary Search Trees

- Binary trees satisfying some condition (other than for heaps)
- In general neither complete, nor "almost-complete"
- Keys are taken from a totally ordered set

Binary Search Trees

- Binary trees satisfying some condition (other than for heaps)
- In general neither complete, nor "almost-complete"
- Keys are taken from a totally ordered set

The following operations on dynamic sets can be done in time proportional to the height of a tree ($\Theta(n)$ worst-case, but usually better):

Binary Search Trees

- Binary trees satisfying some condition (other than for heaps)
- In general neither complete, nor "almost-complete"
- Keys are taken from a totally ordered set

The following operations on dynamic sets can be done in time proportional to the height of a tree ($\Theta(n)$ worst-case, but usually better):

- $\mathrm{SEARCH}(X, k)$: returns some element x of X such that $x.key = k$ (if there is at least one such element)

Binary Search Trees

- Binary trees satisfying some condition (other than for heaps)
- In general neither complete, nor "almost-complete"
- Keys are taken from a totally ordered set

The following operations on dynamic sets can be done in time proportional to the height of a tree ($\Theta(n)$ worst-case, but usually better):

- $\mathrm{SEARCH}(X, k)$: returns some element x of X such that $x.key = k$ (if there is at least one such element)
- $\mathrm{INSERT}(X, x)$: inserts x into a dynamic set X

Binary Search Trees

- Binary trees satisfying some condition (other than for heaps)
- In general neither complete, nor "almost-complete"
- Keys are taken from a totally ordered set

The following operations on dynamic sets can be done in time proportional to the height of a tree ($\Theta(n)$ worst-case, but usually better):

- $\mathrm{SEARCH}(X, k)$: returns some element x of X such that $x.key = k$ (if there is at least one such element)
- $\mathrm{INSERT}(X, x)$: inserts x into a dynamic set X
- $\mathrm{DELETE}(X, x)$: deletes x from X

Binary Search Trees

- Binary trees satisfying some condition (other than for heaps)
- In general neither complete, nor "almost-complete"
- Keys are taken from a totally ordered set

The following operations on dynamic sets can be done in time proportional to the height of a tree ($\Theta(n)$ worst-case, but usually better):

- $\mathrm{SEARCH}(X, k)$: returns some element x of X such that $x.key = k$ (if there is at least one such element)
- $\mathrm{INSERT}(X, x)$: inserts x into a dynamic set X
- $\mathrm{DELETE}(X, x)$: deletes x from X
- $\mathrm{MIN}(X)$: returns some element x of X with minimal $x.key$

Binary Search Trees

- Binary trees satisfying some condition (other than for heaps)
- In general neither complete, nor “almost-complete”
- Keys are taken from a totally ordered set

The following operations on dynamic sets can be done in time proportional to the height of a tree ($\Theta(n)$ worst-case, but usually better):

- $\mathrm{SEARCH}(X, k)$: returns some element x of X such that $x.key = k$ (if there is at least one such element)
- $\mathrm{INSERT}(X, x)$: inserts x into a dynamic set X
- $\mathrm{DELETE}(X, x)$: deletes x from X
- $\mathrm{MIN}(X)$: returns some element x of X with minimal $x.key$
- $\mathrm{MAX}(X)$: returns some element x of X with maximal $x.key$

Binary Search Trees

- Binary trees satisfying some condition (other than for heaps)
- In general neither complete, nor "almost-complete"
- Keys are taken from a totally ordered set

The following operations on dynamic sets can be done in time proportional to the height of a tree ($\Theta(n)$ worst-case, but usually better):

- $\mathrm{SEARCH}(X, k)$: returns some element x of X such that $x.key = k$ (if there is at least one such element)
- $\mathrm{INSERT}(X, x)$: inserts x into a dynamic set X
- $\mathrm{DELETE}(X, x)$: deletes x from X
- $\mathrm{MIN}(X)$: returns some element x of X with minimal $x.key$
- $\mathrm{MAX}(X)$: returns some element x of X with maximal $x.key$
- $\mathrm{SUCC}(X, x)$: returns a successor x in the total ordering of keys

Binary Search Trees

- Binary trees satisfying some condition (other than for heaps)
- In general neither complete, nor "almost-complete"
- Keys are taken from a totally ordered set

The following operations on dynamic sets can be done in time proportional to the height of a tree ($\Theta(n)$ worst-case, but usually better):

- $\mathrm{SEARCH}(X, k)$: returns some element x of X such that $x.key = k$ (if there is at least one such element)
- $\mathrm{INSERT}(X, x)$: inserts x into a dynamic set X
- $\mathrm{DELETE}(X, x)$: deletes x from X
- $\mathrm{MIN}(X)$: returns some element x of X with minimal $x.key$
- $\mathrm{MAX}(X)$: returns some element x of X with maximal $x.key$
- $\mathrm{SUCC}(X, x)$: returns a successor x in the total ordering of keys
- $\mathrm{PRED}(X, x)$: returns a predecessor x in the total ordering of keys

Binary Search Trees

- Binary trees satisfying some condition (other than for heaps)
- In general neither complete, nor "almost-complete"
- Keys are taken from a totally ordered set

The following operations on dynamic sets can be done in time proportional to the height of a tree ($\Theta(n)$ worst-case, but usually better):

- SEARCH(X, k): returns some element x of X such that $x.key = k$ (if there is at least one such element)
- INSERT(X, x): inserts x into a dynamic set X
- DELETE(X, x): deletes x from X
- MIN(X): returns some element x of X with minimal $x.key$
- MAX(X): returns some element x of X with maximal $x.key$
- SUCC(X, x): returns a successor x in the total ordering of keys
- PRED(X, x): returns a predecessor x in the total ordering of keys

These can be used to realise, e.g., dictionaries or priority queues

Binary Search Trees

Assume that the keys are taken from a totally ordered set $(S, \preceq)$

Binary Search Trees

Assume that the keys are taken from a totally ordered set $(S, \preceq)$

The Binary Search Tree Property:

Binary Search Trees

Assume that the keys are taken from a totally ordered set $(S, \preceq)$

The Binary Search Tree Property:

Let x be a node of a binary search tree. Then:

Binary Search Trees

Assume that the keys are taken from a totally ordered set $(S, \preceq)$

The Binary Search Tree Property:

Let x be a node of a binary search tree. Then:

- For all y in a subtree rooted at $x.\mathit{left}$, we have $y.\mathit{key} \preceq x.\mathit{key}$

Binary Search Trees

Assume that the keys are taken from a totally ordered set $(S, \preceq)$

The Binary Search Tree Property:

Let x be a node of a binary search tree. Then:

- For all y in a subtree rooted at $x.left$, we have $y.key \preceq x.key$
- For all y in a subtree rooted at $x.right$, we have $y.key \succeq x.key$

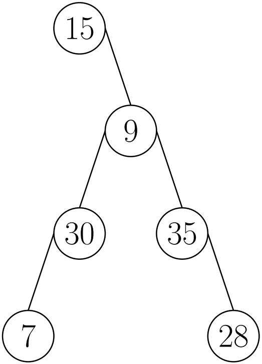
Binary Search Trees

Assume that the keys are taken from a totally ordered set $(S, \preceq)$

The Binary Search Tree Property:

Let x be a node of a binary search tree. Then:

- For all y in a subtree rooted at $x.left$, we have $y.key \preceq x.key$
- For all y in a subtree rooted at $x.right$, we have $y.key \succeq x.key$



Not a binary search tree

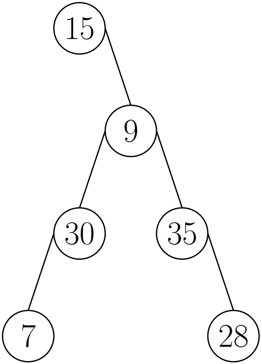
Binary Search Trees

Assume that the keys are taken from a totally ordered set $(S, \preceq)$

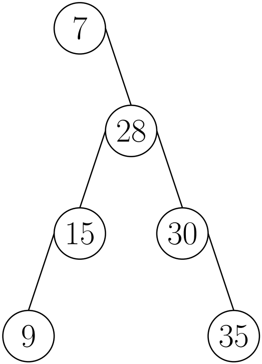
The Binary Search Tree Property:

Let x be a node of a binary search tree. Then:

- For all y in a subtree rooted at $x.left$, we have $y.key \preceq x.key$
- For all y in a subtree rooted at $x.right$, we have $y.key \succeq x.key$



Not a binary search tree



A binary search tree

Binary Search Trees

- Let x be a node in a binary search tree T

Binary Search Trees

- Let x be a node in a binary search tree T
- $\text{BSTSEARCH}(x, k)$ will search for a node with key k in a subtree rooted at x

Binary Search Trees

- Let x be a node in a binary search tree T
- $\text{BSTSEARCH}(x, k)$ will search for a node with key k in a subtree rooted at x
- The call $\text{BSTSEARCH}(T.root, k)$ searches the whole tree

Binary Search Trees

- Let x be a node in a binary search tree T
- $\text{BSTSEARCH}(x, k)$ will search for a node with key k in a subtree rooted at x
- The call $\text{BSTSEARCH}(T.root, k)$ searches the whole tree

$\text{BSTSEARCH}(x, k)$:

Binary Search Trees

- Let x be a node in a binary search tree T
- $\text{BSTSEARCH}(x, k)$ will search for a node with key k in a subtree rooted at x
- The call $\text{BSTSEARCH}(T.root, k)$ searches the whole tree

$\text{BSTSEARCH}(x, k)$:

if $x =$ **nil** *or* $k = x.key$ **then return** x;
if $x \neq$ **nil** *and* $k \prec x.key$ **then return** $\text{BSTSEARCH}(x.left, k)$;
if $x \neq$ **nil** *and* $k \succ x.key$ **then return** $\text{BSTSEARCH}(x.right, k)$;

Binary Search Trees

- Let x be a node in a binary search tree T
- $\text{BSTSEARCH}(x, k)$ will search for a node with key k in a subtree rooted at x
- The call $\text{BSTSEARCH}(T.root, k)$ searches the whole tree

$\text{BSTSEARCH}(x, k)$:

if $x = $ **nil** *or* $k = x.key$ **then return** x;
if $x \neq$ **nil** *and* $k \prec x.key$ **then return** $\text{BSTSEARCH}(x.left, k)$;
if $x \neq$ **nil** *and* $k \succ x.key$ **then return** $\text{BSTSEARCH}(x.right, k)$;

- Time complexity: $\Theta(h_x)$, where h_x is the height of x in T

Binary Search Trees

- Let x be a node in a binary search tree T

Binary Search Trees

- Let x be a node in a binary search tree T
- $\mathrm{BSTMIN}(x)$ resp. $\mathrm{BSTMAX}(x)$ will find a minimum resp. a maximum in a subtree rooted at x

Binary Search Trees

- Let x be a node in a binary search tree T
- $\text{BSTMIN}(x)$ resp. $\text{BSTMAX}(x)$ will find a minimum resp. a maximum in a subtree rooted at x

$\text{BSTMIN}(x)$:

Binary Search Trees

- Let x be a node in a binary search tree T
- $\mathrm{BSTMIN}(x)$ resp. $\mathrm{BSTMAX}(x)$ will find a minimum resp. a maximum in a subtree rooted at x

$\mathrm{BSTMIN}(x)$:

while $x.left \neq$ **nil do** $x \leftarrow x.left$;
return x;

Binary Search Trees

- Let x be a node in a binary search tree T
- BSTMIN(x) resp. BSTMAX(x) will find a minimum resp. a maximum in a subtree rooted at x

BSTMIN(x):

while $x.left \neq$ **nil do** $x \leftarrow x.left$;
return x;

BSTMAX(x):

Binary Search Trees

- Let x be a node in a binary search tree T
- $\mathrm{BSTMIN}(x)$ resp. $\mathrm{BSTMAX}(x)$ will find a minimum resp. a maximum in a subtree rooted at x

$\mathrm{BSTMIN}(x)$:

while $x.\mathit{left} \neq$ **nil do** $x \leftarrow x.\mathit{left}$;
return x;

$\mathrm{BSTMAX}(x)$:

while $x.\mathit{right} \neq$ **nil do** $x \leftarrow x.\mathit{right}$;
return x;

Binary Search Trees

- Let x be a node in a binary search tree T
- $\mathrm{BSTMIN}(x)$ resp. $\mathrm{BSTMAX}(x)$ will find a minimum resp. a maximum in a subtree rooted at x

$\mathrm{BSTMIN}(x)$:

while $x.\mathit{left} \neq$ **nil do** $x \leftarrow x.\mathit{left}$;
return x;

$\mathrm{BSTMAX}(x)$:

while $x.\mathit{right} \neq$ **nil do** $x \leftarrow x.\mathit{right}$;
return x;

- Time complexity: $\Theta(h_x)$, where h_x is the height of x in T

Binary Search Trees

- Let x be a node in a binary search tree T

Binary Search Trees

- Let x be a node in a binary search tree T
- $\text{BSTSUCC}(x)$ will find a successor of x in T w.r.t. the total ordering of keys

Binary Search Trees

- Let x be a node in a binary search tree T
- $\text{BSTSUCC}(x)$ will find a successor of x in T w.r.t. the total ordering of keys

$\text{BSTSUCC}(x)$:

Binary Search Trees

- Let x be a node in a binary search tree T
- BSTSUCC(x) will find a successor of x in T w.r.t. the total ordering of keys

BSTSUCC(x):

```
if x.right ≠ nil then
|   return BSTMIN(x.right)
end
else
|   y ← y.par;
|   while y ≠ nil and x = y.right do
|   |   x = y;
|   |   y = y.par;
|   end
|   return y;
end
```

Binary Search Trees

- Let x be a node in a binary search tree T
- $\text{BSTSUCC}(x)$ will find a successor of x in T w.r.t. the total ordering of keys

$\text{BSTSUCC}(x)$:

```
if x.right ≠ nil then
|   return BSTMIN(x.right)
end
else
|   y ← y.par;
|   while y ≠ nil and x = y.right do
|   |   x = y;
|   |   y = y.par;
|   end
|   return y;
end
```

- Time complexity: $\Theta(h)$, where h is the height of the tree T

Binary Search Trees

- Let x be a node in a binary search tree T
- $\mathrm{BSTSUCC}(x)$ will find a successor of x in T w.r.t. the total ordering of keys

$\mathrm{BSTSUCC}(x)$:

```
if x.right ≠ nil then
|   return BSTMIN(x.right)
end
else
|   y ← y.par;
|   while y ≠ nil and x = y.right do
|   |   x = y;
|   |   y = y.par;
|   end
|   return y;
end
```

- Time complexity: $\Theta(h)$, where h is the height of the tree T
- Predecessors can be found in a symmetric way

Binary Search Trees

- Let x be a node not in a binary search tree T

Binary Search Trees

- Let x be a node not in a binary search tree T
- That is, initially $x.par = x.left = x.right = \textbf{nil}$

Binary Search Trees

- Let x be a node not in a binary search tree T
- That is, initially $x.par = x.left = x.right =$ **nil**
- $\text{BSTINSERT}(T, x)$ will insert x into T

Binary Search Trees

- Let x be a node not in a binary search tree T
- That is, initially $x.par = x.left = x.right = \textbf{nil}$
- $\text{BSTINSERT}(T, x)$ will insert x into T

$\text{BSTINSERT}(T, x)$:

Binary Search Trees

- Let x be a node not in a binary search tree T
- That is, initially $x.par = x.left = x.right = \textbf{nil}$
- $\text{BSTINSERT}(T, x)$ will insert x into T

$\text{BSTINSERT}(T, x)$:

```
par ← nil;
y ← T.root;
while y ≠ nil do
    par ← y;
    if x.key ≺ par.key then y ← par.left else y ← par.right;
end
x.par ← par;
if par = nil then T.root ← x
else
    if x.key ≺ par.key then par.left ← x
    if x.key ⪰ par.key then par.right ← x
end
```

Binary Search Trees

- Let x be a node not in a binary search tree T
- That is, initially $x.par = x.left = x.right = \textbf{nil}$
- $\text{BSTINSERT}(T, x)$ will insert x into T

$\text{BSTINSERT}(T, x)$:

```
par ← nil;
y ← T.root;
while y ≠ nil do
    par ← y;
    if x.key ≺ par.key then y ← par.left else y ← par.right;
end
x.par ← par;
if par = nil then T.root ← x
else
    if x.key ≺ par.key then par.left ← x
    if x.key ⪰ par.key then par.right ← x
end
```

- Time complexity: $\Theta(h)$, where h is the height of T

Binary Search Trees

- Let x be a node in a binary search tree T

Binary Search Trees

- Let x be a node in a binary search tree T
- $\text{BSTDELETE}(T, x)$ will delete x from T

Binary Search Trees

- Let x be a node in a binary search tree T
- $\text{BSTDELETE}(T, x)$ will delete x from T

Idea:

Binary Search Trees

- Let x be a node in a binary search tree T
- $\mathrm{BSTDELETE}(T, x)$ will delete x from T

Idea:

- If x has no left child, then replace x by its right child (or by **nil** if there is none)

Binary Search Trees

- Let x be a node in a binary search tree T
- $\text{BSTDELETE}(T, x)$ will delete x from T

Idea:

- If x has no left child, then replace x by its right child (or by **nil** if there is none)
- If x has a left child but no right child, then replace x by its left child

Binary Search Trees

- Let x be a node in a binary search tree T
- $\text{BSTDELETE}(T, x)$ will delete x from T

Idea:

- If x has no left child, then replace x by its right child (or by **nil** if there is none)
- If x has a left child but no right child, then replace x by its left child
- If x has both children, then:

Binary Search Trees

- Let x be a node in a binary search tree T
- $\text{BSTDELETE}(T, x)$ will delete x from T

Idea:

- If x has no left child, then replace x by its right child (or by **nil** if there is none)
- If x has a left child but no right child, then replace x by its left child
- If x has both children, then:
  - Let y be the successor of x (i.e., the minimal element of the subtree rooted at the right child of x)

Binary Search Trees

- Let x be a node in a binary search tree T
- $\text{BSTDELETE}(T, x)$ will delete x from T

Idea:

- If x has no left child, then replace x by its right child (or by **nil** if there is none)
- If x has a left child but no right child, then replace x by its left child
- If x has both children, then:
  - Let y be the successor of x (i.e., the minimal element of the subtree rooted at the right child of x)
  - If y is the right child of x, then replace x by y

Binary Search Trees

- Let x be a node in a binary search tree T
- $\text{BSTDELETE}(T, x)$ will delete x from T

Idea:

- If x has no left child, then replace x by its right child (or by **nil** if there is none)
- If x has a left child but no right child, then replace x by its left child
- If x has both children, then:
  - Let y be the successor of x (i.e., the minimal element of the subtree rooted at the right child of x)
  - If y is the right child of x, then replace x by y
  - Otherwise replace y by its right child and then x by y

Binary Search Trees

- Let x be a node in a binary search tree T
- $\text{BSTDELETE}(T, x)$ will delete x from T

Idea:

- If x has no left child, then replace x by its right child (or by **nil** if there is none)
- If x has a left child but no right child, then replace x by its left child
- If x has both children, then:
  - Let y be the successor of x (i.e., the minimal element of the subtree rooted at the right child of x)
  - If y is the right child of x, then replace x by y
  - Otherwise replace y by its right child and then x by y
- Replacements shall be done via $\text{BSTTRANSPLANT}(T, u, v)$

Binary Search Trees

- $\text{BSTTRANSPLANT}(T, u, v)$ replaces a subtree rooted at u by a subtree rooted at v

Binary Search Trees

- $\text{BSTTRANSPLANT}(T, u, v)$ replaces a subtree rooted at u by a subtree rooted at v

$\text{BSTTRANSPLANT}(T, u, v)$:

Binary Search Trees

- $\mathrm{BSTTRANSPLANT}(T, u, v)$ replaces a subtree rooted at u by a subtree rooted at v

$\mathrm{BSTTRANSPLANT}(T, u, v)$:

if $u.par = $ **nil then** $T.root \leftarrow v$;
else
  if $u = u.par.left$ **then** $u.par.left \leftarrow v$;
  if $u = u.par.right$ **then** $u.par.right \leftarrow v$;
end
if $v \neq$ **nil then** $v.par \leftarrow u.par$;

Binary Search Trees

- $\text{BSTTRANSPLANT}(T, u, v)$ replaces a subtree rooted at u by a subtree rooted at v

$\text{BSTTRANSPLANT}(T, u, v)$:

if $u.par = $ **nil then** $T.root \leftarrow v$;
else
  if $u = u.par.left$ **then** $u.par.left \leftarrow v$;
  if $u = u.par.right$ **then** $u.par.right \leftarrow v$;
end
if $v \neq$ **nil then** $v.par \leftarrow u.par$;

- Time complexity: $\Theta(1)$

Binary Search Trees

$\text{BSTDELETE}(T, x)$:

Binary Search Trees

BSTDELETE(T, x):

```
if x.left = nil then
|   BSTTRANSPLANT(T, x, x.right)
else if x.right = nil then
|   BSTTRANSPLANT(T, x, x.left)
else
|   y ← BSTMIN(x.right);
|   if y.par ≠ x then
|   |   BSTTRANSPLANT(T, y, y.right);
|   |   y.right ← x.right;
|   |   y.right.par ← y;
|   end
|   BSTTRANSPLANT(T, x, y);
|   y.left ← x.left;
|   y.left.par ← y;
end
```

Binary Search Trees

```
BSTDELETE(T, x):
if x.left = nil then
|   BSTTRANSPLANT(T, x, x.right)
else if x.right = nil then
|   BSTTRANSPLANT(T, x, x.left)
else
|   y ← BSTMIN(x.right);
|   if y.par ≠ x then
|   |   BSTTRANSPLANT(T, y, y.right);
|   |   y.right ← x.right;
|   |   y.right.par ← y;
|   end
|   BSTTRANSPLANT(T, x, y);
|   y.left ← x.left;
|   y.left.par ← y;
end
```

- Time complexity: $\Theta(h_x)$, where h_x is the height of x in T

Balanced Binary Search Trees

For most operations on binary search trees:

Balanced Binary Search Trees

For most operations on binary search trees:

- The worst-case complexity is $\Theta(h)$, where h is the height of the tree

Balanced Binary Search Trees

For most operations on binary search trees:

- The worst-case complexity is $\Theta(h)$, where h is the height of the tree
- The worst case over all trees with n nodes: $\Theta(n)$

Balanced Binary Search Trees

For most operations on binary search trees:

- The worst-case complexity is $\Theta(h)$, where h is the height of the tree
- The worst case over all trees with n nodes: $\Theta(n)$

It makes sense to keep the tree balanced, so that:

Balanced Binary Search Trees

For most operations on binary search trees:

- The worst-case complexity is $\Theta(h)$, where h is the height of the tree
- The worst case over all trees with n nodes: $\Theta(n)$

It makes sense to keep the tree balanced, so that:

- The height of the tree is $\Theta(\log n)$

Balanced Binary Search Trees

For most operations on binary search trees:

- The worst-case complexity is $\Theta(h)$, where h is the height of the tree
- The worst case over all trees with n nodes: $\Theta(n)$

It makes sense to keep the tree balanced, so that:

- The height of the tree is $\Theta(\log n)$
- The operations on the tree take $\Theta(\log n)$ in worst-case

Balanced Binary Search Trees

For most operations on binary search trees:

- The worst-case complexity is $\Theta(h)$, where h is the height of the tree
- The worst case over all trees with n nodes: $\Theta(n)$

It makes sense to keep the tree balanced, so that:

- The height of the tree is $\Theta(\log n)$
- The operations on the tree take $\Theta(\log n)$ in worst-case

Solutions:

Balanced Binary Search Trees

For most operations on binary search trees:

- The worst-case complexity is $\Theta(h)$, where h is the height of the tree
- The worst case over all trees with n nodes: $\Theta(n)$

It makes sense to keep the tree balanced, so that:

- The height of the tree is $\Theta(\log n)$
- The operations on the tree take $\Theta(\log n)$ in worst-case

Solutions:

- **Red-black trees**

Balanced Binary Search Trees

For most operations on binary search trees:

- The worst-case complexity is $\Theta(h)$, where h is the height of the tree
- The worst case over all trees with n nodes: $\Theta(n)$

It makes sense to keep the tree balanced, so that:

- The height of the tree is $\Theta(\log n)$
- The operations on the tree take $\Theta(\log n)$ in worst-case

Solutions:

- **Red-black trees**
- AVL trees

Balanced Binary Search Trees

For most operations on binary search trees:

- The worst-case complexity is $\Theta(h)$, where h is the height of the tree
- The worst case over all trees with n nodes: $\Theta(n)$

It makes sense to keep the tree balanced, so that:

- The height of the tree is $\Theta(\log n)$
- The operations on the tree take $\Theta(\log n)$ in worst-case

Solutions:

- **Red-black trees**
- AVL trees
- ...

Rotations

Elementary transformations retaining the binary search tree property

Rotations

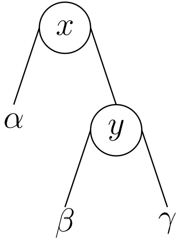
Elementary transformations retaining the binary search tree property

$\text{LEFTROTATE}(x):$

Rotations

Elementary transformations retaining the binary search tree property

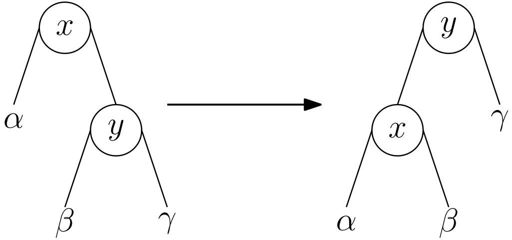
LEFTROTATE(x) :



Rotations

Elementary transformations retaining the binary search tree property

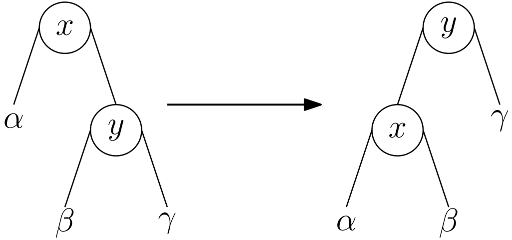
$\text{LEFTROTATE}(x):$



Rotations

Elementary transformations retaining the binary search tree property

LEFTROTATE(x) :

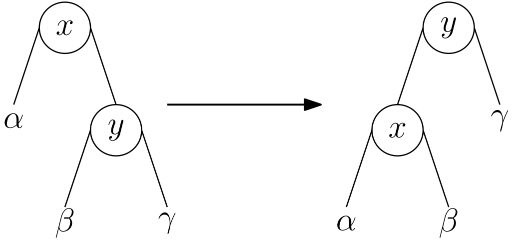


RIGHTROTATE(x) :

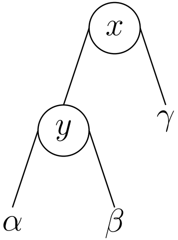
Rotations

Elementary transformations retaining the binary search tree property

LEFTROTATE(x) :



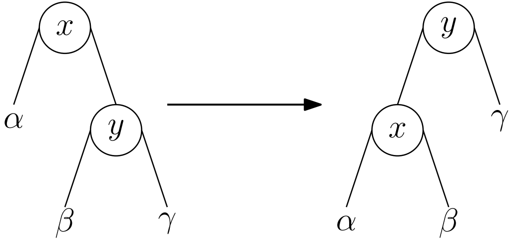
RIGHTROTATE(x) :



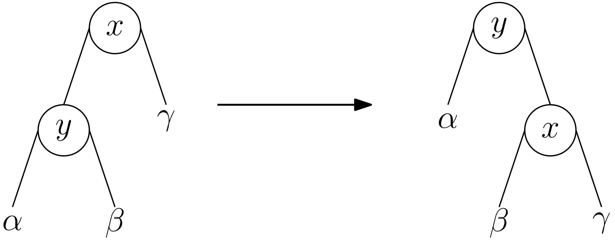
Rotations

Elementary transformations retaining the binary search tree property

LEFTROTATE(x) :



RIGHTROTATE(x) :



Rotations

- The following pseudocode assumes that $x.right \neq$ **nil**

Rotations

- The following pseudocode assumes that $x.right \neq \textbf{nil}$

LEFTROTATE(T, x):

Rotations

- The following pseudocode assumes that $x.right \neq$ **nil**

LEFTROTATE(T, x):

$y \leftarrow x.right$;
$x.right \leftarrow y.left$;
if $x.right \neq$ **nil then** $x.right.par \leftarrow x$;
$y.par \leftarrow x.par$;
if $y.par =$ **nil then** $T.root \leftarrow y$;
else
    if $x = y.par.left$ **then** $y.par.left \leftarrow y$;
    if $x = y.par.right$ **then** $y.par.right \leftarrow y$;
end
$y.left \leftarrow x$;
$x.par \leftarrow y$;

Rotations

- The following pseudocode assumes that $x.right \neq$ **nil**

LEFTROTATE(T, x):

$y \leftarrow x.right$;
$x.right \leftarrow y.left$;
if $x.right \neq$ **nil then** $x.right.par \leftarrow x$;
$y.par \leftarrow x.par$;
if $y.par =$ **nil then** $T.root \leftarrow y$;
else
  if $x = y.par.left$ **then** $y.par.left \leftarrow y$;
  if $x = y.par.right$ **then** $y.par.right \leftarrow y$;
end
$y.left \leftarrow x$;
$x.par \leftarrow y$;

- Time complexity: $\Theta(1)$

Rotations

- The following pseudocode assumes that $x.right \neq$ **nil**

LEFTROTATE(T, x):

```
y ← x.right;
x.right ← y.left;
if x.right ≠ nil then x.right.par ← x;
y.par ← x.par;
if y.par = nil then T.root ← y;
else
    if x = y.par.left then y.par.left ← y;
    if x = y.par.right then y.par.right ← y;
end
y.left ← x;
x.par ← y;
```

- Time complexity: $\Theta(1)$
- RIGHTROTATE(T, x) can be done symmetrically

Red-Black Trees

Binary search trees such that:

Red-Black Trees

Binary search trees such that:

- Each node has an additional attribute – a colour

Red-Black Trees

Binary search trees such that:

- Each node has an additional attribute – a colour
- The colour of a node can be red or black

Red-Black Trees

Binary search trees such that:

- Each node has an additional attribute – a colour
- The colour of a node can be red or black
- We shall regard **nil** as being black

Red-Black Trees

Binary search trees such that:

- Each node has an additional attribute – a colour
- The colour of a node can be red or black
- We shall regard **nil** as being black

A binary search tree T coloured like this is a red-black tree if:

Red-Black Trees

Binary search trees such that:

- Each node has an additional attribute – a colour
- The colour of a node can be red or black
- We shall regard **nil** as being black

A binary search tree T coloured like this is a red-black tree if:

- The root of T is black

Red-Black Trees

Binary search trees such that:

- Each node has an additional attribute – a colour
- The colour of a node can be red or black
- We shall regard **nil** as being black

A binary search tree T coloured like this is a red-black tree if:

- The root of T is black
- If a node in T is red, then both its children are black

Red-Black Trees

Binary search trees such that:

- Each node has an additional attribute – a colour
- The colour of a node can be red or black
- We shall regard **nil** as being black

A binary search tree T coloured like this is a red-black tree if:

- The root of T is black
- If a node in T is red, then both its children are black
- For each x in T: each path from x to its descendant **nil**s contains the same number of black nodes

Red-Black Trees

Binary search trees such that:

- Each node has an additional attribute – a colour
- The colour of a node can be red or black
- We shall regard **nil** as being black

A binary search tree T coloured like this is a red-black tree if:

- The root of T is black
- If a node in T is red, then both its children are black
- For each x in T: each path from x to its descendant **nil**s contains the same number of black nodes

The idea behind this definition:

Red-Black Trees

Binary search trees such that:

- Each node has an additional attribute – a colour
- The colour of a node can be red or black
- We shall regard **nil** as being black

A binary search tree T coloured like this is a red-black tree if:

- The root of T is black
- If a node in T is red, then both its children are black
- For each x in T: each path from x to its descendant **nil**s contains the same number of black nodes

The idea behind this definition:

- If T contains black nodes only, then T is complete

Red-Black Trees

Binary search trees such that:

- Each node has an additional attribute – a colour
- The colour of a node can be red or black
- We shall regard **nil** as being black

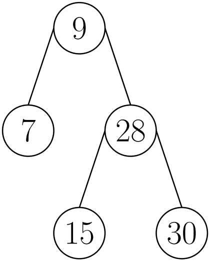
A binary search tree T coloured like this is a red-black tree if:

- The root of T is black
- If a node in T is red, then both its children are black
- For each x in T: each path from x to its descendant **nil**s contains the same number of black nodes

The idea behind this definition:

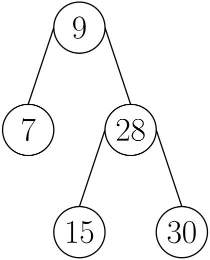
- If T contains black nodes only, then T is complete
- There are not so many red nodes...

Red-Black Trees

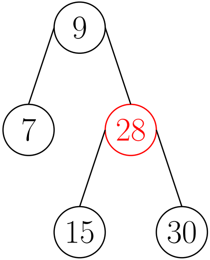


Not a red-black tree

Red-Black Trees



Not a red-black tree



A red-black tree

Red-Black Trees

Lemma

Let T be a red-black tree with n nodes. Then the height of each node in T is at most $\Theta(\log n)$.

Red-Black Trees

Lemma

Let T be a red-black tree with n nodes. Then the height of each node in T is at most $\Theta(\log n)$.

Proof sketch

Red-Black Trees

Lemma

Let T be a red-black tree with n nodes. Then the height of each node in T is at most $\Theta(\log n)$.

Proof sketch

- A tree of height h has at most $\Theta(2^h)$ nodes

Red-Black Trees

Lemma

Let T be a red-black tree with n nodes. Then the height of each node in T is at most $\Theta(\log n)$.

Proof sketch

- A tree of height h has at most $\Theta(2^h)$ nodes
- In a red-black tree: given a path from the root to a leaf, each other such path is at most twice as long

Red-Black Trees

Lemma

Let T be a red-black tree with n nodes. Then the height of each node in T is at most $\Theta(\log n)$.

Proof sketch

- A tree of height h has at most $\Theta(2^h)$ nodes
- In a red-black tree: given a path from the root to a leaf, each other such path is at most twice as long
- A tree of height h has between $\Theta(2^{h/2})$ and $\Theta(2^h)$ nodes

Red-Black Trees

Lemma

Let T be a red-black tree with n nodes. Then the height of each node in T is at most $\Theta(\log n)$.

Proof sketch

- A tree of height h has at most $\Theta(2^h)$ nodes
- In a red-black tree: given a path from the root to a leaf, each other such path is at most twice as long
- A tree of height h has between $\Theta(2^{h/2})$ and $\Theta(2^h)$ nodes
- A tree with n nodes is of height $\Theta(\log n)$

Red-Black Trees

Insertion and deletion can both be written so that they:

Red-Black Trees

Insertion and deletion can both be written so that they:

- Retain the red-black properties

Red-Black Trees

Insertion and deletion can both be written so that they:

- Retain the red-black properties
- Have time complexity $\Theta(\log n)$

Red-Black Trees

Insertion and deletion can both be written so that they:

- Retain the red-black properties
- Have time complexity $\Theta(\log n)$

Idea:

Red-Black Trees

Insertion and deletion can both be written so that they:

- Retain the red-black properties
- Have time complexity $\Theta(\log n)$

Idea:

- First insert or delete a node as in a binary search tree

Red-Black Trees

Insertion and deletion can both be written so that they:

- Retain the red-black properties
- Have time complexity $\Theta(\log n)$

Idea:

- First insert or delete a node as in a binary search tree
- When inserting a node, colour it red

Red-Black Trees

Insertion and deletion can both be written so that they:

- Retain the red-black properties
- Have time complexity $\Theta(\log n)$

Idea:

- First insert or delete a node as in a binary search tree
- When inserting a node, colour it red
- Run fixup procedures that recover the red-black properties via some rotations

Red-Black Trees

Insertion and deletion can both be written so that they:

- Retain the red-black properties
- Have time complexity $\Theta(\log n)$

Idea:

- First insert or delete a node as in a binary search tree
- When inserting a node, colour it red
- Run fixup procedures that recover the red-black properties via some rotations

See Cormen et al. for details

Tree Sort

Idea:

Tree Sort

Idea:

- Build a search tree containing elements of an array a

Tree Sort

Idea:

- Build a search tree containing elements of an array *a*
- Traverse the tree in inorder

Tree Sort

Idea:

- Build a search tree containing elements of an array a
- Traverse the tree in inorder

TRAVERSE(x):

Tree Sort

Idea:

- Build a search tree containing elements of an array a
- Traverse the tree in inorder

$\mathrm{TRAVERSE}(x)$:

if $x.\mathit{left} \neq$ **nil then** $a_l \leftarrow \mathrm{TRAVERSE}(x.\mathit{left})$;
if $x.\mathit{right} \neq$ **nil then** $a_r \leftarrow \mathrm{TRAVERSE}(x.\mathit{right})$;
return $a_l \cdot \langle x \rangle \cdot a_r$;

Tree Sort

Idea:

- Build a search tree containing elements of an array a
- Traverse the tree in inorder

$\mathrm{TRAVERSE}(x)$:

if $x.\mathit{left} \neq$ **nil then** $a_l \leftarrow \mathrm{TRAVERSE}(x.\mathit{left})$;
if $x.\mathit{right} \neq$ **nil then** $a_r \leftarrow \mathrm{TRAVERSE}(x.\mathit{right})$;
return $a_l \cdot \langle x \rangle \cdot a_r$;

$\mathrm{TREESORT}(a)$:

Tree Sort

Idea:

- Build a search tree containing elements of an array a
- Traverse the tree in inorder

$\text{TRAVERSE}(x)$:

if $x.\mathit{left} \neq$ **nil then** $a_l \leftarrow \text{TRAVERSE}(x.\mathit{left})$;
if $x.\mathit{right} \neq$ **nil then** $a_r \leftarrow \text{TRAVERSE}(x.\mathit{right})$;
return $a_l \cdot \langle x \rangle \cdot a_r$;

$\text{TREESORT}(a)$:

$T \leftarrow$ **nil**;
for $i \leftarrow 1$ **to** $a.\mathit{length}$ **do**
| $\text{BSTINSERT}(T, a[i])$
end
$a \leftarrow \text{TRAVERSE}(T.\mathit{root})$;

Tree Sort

Idea:

- Build a search tree containing elements of an array a
- Traverse the tree in inorder

TRAVERSE(x):

if $x.left \neq$ **nil then** $a_l \leftarrow \mathrm{TRAVERSE}(x.left)$;
if $x.right \neq$ **nil then** $a_r \leftarrow \mathrm{TRAVERSE}(x.right)$;
return $a_l \cdot \langle x \rangle \cdot a_r$;

TREESORT(a):

$T \leftarrow$ **nil**;
for $i \leftarrow 1$ **to** $a.length$ **do**
| $\mathrm{BSTINSERT}(T, a[i])$
end
$a \leftarrow \mathrm{TRAVERSE}(T.root)$;

- Worst-case time complexity: $\Theta(n^2)$

Tree Sort

Idea:

- Build a search tree containing elements of an array *a*
- Traverse the tree in inorder

TRAVERSE(x):

if $x.left \neq$ **nil then** $a_l \leftarrow \mathrm{TRAVERSE}(x.left)$;
if $x.right \neq$ **nil then** $a_r \leftarrow \mathrm{TRAVERSE}(x.right)$;
return $a_l \cdot \langle x \rangle \cdot a_r$;

TREESORT(a):

$T \leftarrow$ **nil**;
for $i \leftarrow 1$ **to** $a.length$ **do**
| $\mathrm{BSTINSERT}(T, a[i])$
end
$a \leftarrow \mathrm{TRAVERSE}(T.root)$;

- Worst-case time complexity: $\Theta(n^2)$
- Using balanced search trees: $\Theta(n \log n)$