CS F364: Design & Analysis of Algorithms

Lecture-kt10: Binary Search Tree (contd.) + Red-Black Trees

Dr. Kamlesh Tiwari,
Assistant Professor,
Department of Computer Science and Information Systems,
BITS Pilani, Rajasthan-333031 INDIA

Feb 11, 2017 (Campus @ BITS-Pilani Jan-May 2017)

Recap: Binary Search Tree

- Search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE

Recap: Binary Search Tree

- Search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE
- Can be used as a dictionary and as a priority queue

Recap: Binary Search Tree

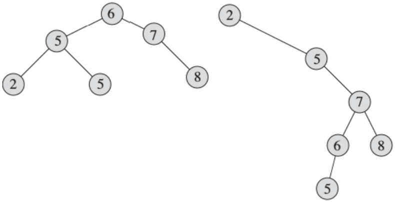
- Search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE
- Can be used as a dictionary and as a priority queue
- Basic operations on a binary search tree take time proportional to the height of the tree

Recap: Binary Search Tree

- Search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE
- Can be used as a dictionary and as a priority queue
- Basic operations on a binary search tree take time proportional to the height of the tree
- **Binary-search-tree property:** Let x be a node in a binary search tree. If y is a node in the
  - **left** subtree of x, then $y.key \leq x.key$
  - **right** subtree of x, then $y.key \geq x.key$.

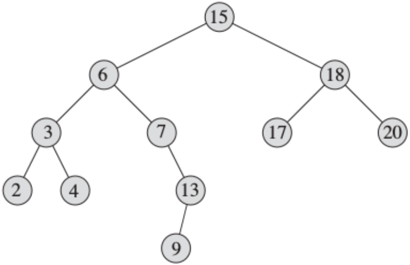
Recap: Binary Search Tree

- Search tree data structure supports many dynamic-set operations, including SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, and DELETE
- Can be used as a dictionary and as a priority queue
- Basic operations on a binary search tree take time proportional to the height of the tree
- **Binary-search-tree property:** Let x be a node in a binary search tree. If y is a node in the
  - **left** subtree of x, then $y.key \leq x.key$
  - **right** subtree of x, then $y.key \geq x.key$.

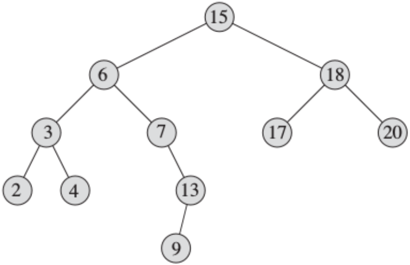


- We may safely assume all items to be different
- In order traversal is monotonous

Search in BST



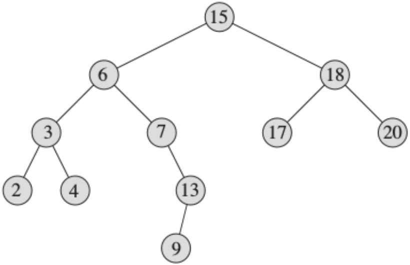
Search in BST



Algorithm 2: BST-Search(x,k)

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

Search in BST



Algorithm 3: BST-Search(x,k)

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

While searching for 363, which can not be a search sequence

a. 2, 252, 401, 398, 330, 344, 397, 363.

b. 924, 220, 911, 244, 898, 258, 362, 363.

c. 925, 202, 911, 240, 912, 245, 363.

d. 2, 399, 387, 219, 266, 382, 381, 278, 363.

e. 935, 278, 347, 621, 299, 392, 358, 363.

Minimum Maximum and Successor

Algorithm 4: BST-Minimum(x)

while $x.left \neq nil$ **do**
  $x = x.left$
return x

Minimum Maximum and Successor

Algorithm 7: BST-Minimum(x)

while $x.left \neq nil$ **do**
  $x = x.left$
return x

Algorithm 8: BST-Maximum(x)

while $x.right \neq nil$ **do**
  $x = x.right$
return x

Minimum Maximum and Successor

Algorithm 10: BST-Minimum(x)

```
while x.left ≠ nil do
    x = x.left
return x
```

Algorithm 11: BST-Maximum(x)

```
while x.right ≠ nil do
    x = x.right
return x
```

Successor is next smallest number

Algorithm 12: BST-Successor(x)

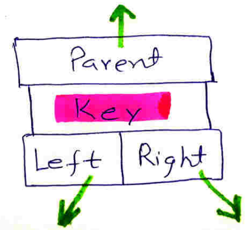
```
if x.right ≠ nil then
    return BST-Minimum(x.right)
y = x.parent
while y ≠ nil and x = y.right do
    x = y
    y = x.parent
return y
```

Insertion in BST

- Tree has
  - root

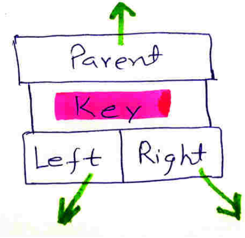
Insertion in BST

- Tree has
  - root
- Every node has
  - key
  - left
  - right
  - parent or p



Insertion in BST

- Tree has
  - root
- Every node has
  - key
  - left
  - right
  - parent or p

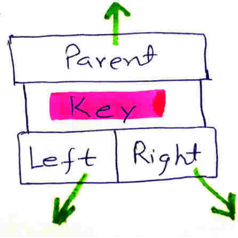


Algorithm 15: BST-Insert(T,z)

```
y = nil, x = T.root, z.left=nil, z.right=nil
while x ≠ nil do
    y=x
    if z.key < x.key then
        x=x.left
    else
        return x=x.right
z.parent = y
if y == nil then
    T.root = z
else if z.key < y.key then
    y.left=z
else
    y.right = z
```

Insertion in BST

- Tree has
  - root
- Every node has
  - key
  - left
  - right
  - parent or p



Algorithm 16: BST-Insert(T,z)

y = nil, x = T.root, z.left=nil, z.right=nil
while $x \neq nil$ **do**
    y=x
    if $z.key < x.key$ **then**
        x=x.left
    else
        return x=x.right
z.parent = y
if $y == nil$ **then**
    T.root = z
else if $z.key < y.key$ **then**
    y.left=z
else
    y.right = z

Time taken is proportional to height of tree

Deletion in BST

Algorithm 17: BST-Transplant(T,u,v)

if $u.parent = nil$ **then**
    $T.root = v$
else if $u = u.parent.left$ **then**
    $u.parent.left = v$
else
    $u.parent.right = v$
if $v \neq nil$ **then**
    $v.parent = u.parent$

Deletion in BST

Algorithm 19: BST-Transplant(T,u,v)

if $u.parent = nil$ **then**
  $T.root = v$
else if $u = u.parent.left$ **then**
  $u.parent.left = v$
else
  $u.parent.right = v$
if $v \neq nil$ **then**
  $v.parent = u.parent$

Algorithm 20: BST-Delete(T,z)

if $z.left = nil$ **then**
  BST-Transplant(T, z, z.right)
else if $z.right = nil$ **then**
  BST-Transplant(T, z, z.left)
else
  y = BST-Minimum(z.right) **if** $y.parent \neq z$ **then**
    BST-Transplant(T, y, z.right)
    y.right=z.right
    y.right.parent = y
  BST-Transplant(T, z, y)
  y.left=z.left
  y.left.parent = y

Time is proportional to tree height, (is $O(\log n)$ when build randomly)

Red-Black Trees

Red black tree is **approximately balanced** binary search tree that stores one extra bit of storage per node called *color*.

Red-Black Trees

Red black tree is **approximately balanced** binary search tree that stores one extra bit of storage per node called *color*.It satisfies following red-black properties:

1. Every node is either red or black

Red-Black Trees

Red black tree is **approximately balanced** binary search tree that stores one extra bit of storage per node called *color*.It satisfies following red-black properties:

1. Every node is either red or black
2. The root is black

Red-Black Trees

Red black tree is **approximately balanced** binary search tree that stores one extra bit of storage per node called *color*.It satisfies following red-black properties:

1. Every node is either red or black
2. The root is black
3. Every leaf (NIL) is black.

Red-Black Trees

Red black tree is **approximately balanced** binary search tree that stores one extra bit of storage per node called *color*.It satisfies following 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

Red-Black Trees

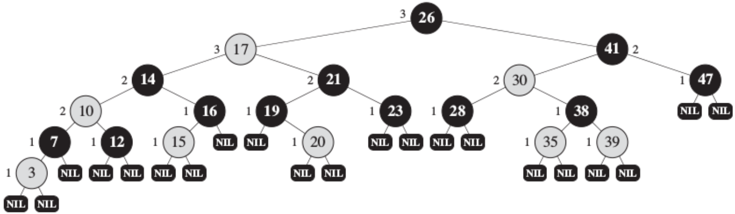
Red black tree is **approximately balanced** binary search tree that stores one extra bit of storage per node called *color*.It satisfies following red-black properties:

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 simple paths from the node to descendant leaves contain the same number of black nodes.

Red-Black Trees

Red black tree is **approximately balanced** binary search tree that stores one extra bit of storage per node called *color*.It satisfies following 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 simple paths from the node to descendant leaves contain the same number of black nodes.



Height of Red-Black Trees

- The number of black nodes on any path from, but not including, a node x to a leaf is called **black-height** of the node $bh(x)$

Height of Red-Black Trees

- The number of black nodes on any path from, but not including, a node x to a leaf is called **black-height** of the node $bh(x)$

A red-black tree with n internal nodes has height at most $2\log(n+1)$

Height of Red-Black Trees

- The number of black nodes on any path from, but not including, a node x to a leaf is called **black-height** of the node $bh(x)$

A red-black tree with n internal nodes has height at most $2\log(n+1)$

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

Height of Red-Black Trees

- The number of black nodes on any path from, but not including, a node x to a leaf is called **black-height** of the node $bh(x)$

A red-black tree with n internal nodes has height at most $2\log(n+1)$

1. A subtree rooted at x contains at least $2^{bh(x)} - 1$ internal nodes
   - If the height of x is 0, then x must be a leaf, and the subtree rooted at x indeed contains $2^{bh(x)} - 1 = 2^0 - 1 = 1 - 1 = 0$ internal nodes.

Height of Red-Black Trees

- The number of black nodes on any path from, but not including, a node x to a leaf is called **black-height** of the node $bh(x)$

A red-black tree with n internal nodes has height at most $2\log(n+1)$

1. A subtree rooted at x contains at least $2^{bh(x)} - 1$ internal nodes
   - If the height of x is 0, then x must be a leaf, and the subtree rooted at x indeed contains $2^{bh(x)} - 1 = 2^0 - 1 = 1 - 1 = 0$ internal nodes.
   - Consider a node x that has positive height and is an internal node with two children. Each child has black-height either $bh(x)$ or $bh(x) - 1$ depending on whether its color is red or black

Height of Red-Black Trees

- The number of black nodes on any path from, but not including, a node x to a leaf is called **black-height** of the node $bh(x)$

A red-black tree with n internal nodes has height at most $2\log(n+1)$

1. A subtree rooted at x contains at least $2^{bh(x)} - 1$ internal nodes
   - If the height of x is 0, then x must be a leaf, and the subtree rooted at x indeed contains $2^{bh(x)} - 1 = 2^0 - 1 = 1 - 1 = 0$ internal nodes.
   - Consider a node x that has positive height and is an internal node with two children. Each child has black-height either $bh(x)$ or $bh(x) - 1$ depending on whether its color is red or black
   - Assume each child has at least $2^{bh(x)-1} - 1$ number of internal nodes. Thus the subtree rooted at x has $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) - 1 = 2^{bh(x)} - 1$ internal nodes.

Height of Red-Black Trees

- The number of black nodes on any path from, but not including, a node x to a leaf is called **black-height** of the node $bh(x)$

A red-black tree with n internal nodes has height at most $2\log(n+1)$

1. A subtree rooted at x contains at least $2^{bh(x)} - 1$ internal nodes
   - If the height of x is 0, then x must be a leaf, and the subtree rooted at x indeed contains $2^{bh(x)} - 1 = 2^0 - 1 = 1 - 1 = 0$ internal nodes.
   - Consider a node x that has positive height and is an internal node with two children. Each child has black-height either $bh(x)$ or $bh(x) - 1$ depending on whether its color is red or black
   - Assume each child has at least $2^{bh(x)-1} - 1$ number of internal nodes. Thus the subtree rooted at x has $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) - 1 = 2^{bh(x)} - 1$ internal nodes.
2. At least half the nodes on any simple path from the root to a leaf, not including the root, must be black.

Height of Red-Black Trees

- The number of black nodes on any path from, but not including, a node x to a leaf is called **black-height** of the node $bh(x)$

A red-black tree with n internal nodes has height at most $2\log(n+1)$

1. A subtree rooted at x contains at least $2^{bh(x)} - 1$ internal nodes
   - If the height of x is 0, then x must be a leaf, and the subtree rooted at x indeed contains $2^{bh(x)} - 1 = 2^0 - 1 = 1 - 1 = 0$ internal nodes.
   - Consider a node x that has positive height and is an internal node with two children. Each child has black-height either $bh(x)$ or $bh(x) - 1$ depending on whether its color is red or black
   - Assume each child has at least $2^{bh(x)-1} - 1$ number of internal nodes. Thus the subtree rooted at x has $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) - 1 = 2^{bh(x)} - 1$ internal nodes.
2. At least half the nodes on any simple path from the root to a leaf, not including the root, must be black.
3. Black-height of the root must be at least $h/2$; thus $n \geq 2^{h/2} - 1$ that gets the same.

Thank You!

Thank you very much for your attention!

Queries ?