Datenstrukturen und Effiziente Algorithmen

Vorlesung *Datenstrukturen und Effiziente Algorithmen* vom 22.5.2012

Mario Stanke
Institut für Mathematik und Informatik
Universität Greifswald
Binary Search Tree

**Binary search tree (BST)**

- dynamic set data structure that supports operations: **SEARCH, INSERTION, DELETION, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR**
- **binary tree** that holds keys from a totally ordered set
- each node has attributes: `key`, `left`, `right`, `p`arent

**Binary search tree property**

If node $y$ is in the left subtree of node $x$, then $y.key \leq x.key$.
If node $y$ is in the right subtree of node $x$, then $y.key \geq x.key$. 
Example 1 (A binary search tree)

missing childs and parents (root) are NULL and denoted with “/”
Example 2 (BST in simple format)
## Binary Search Tree Operations

### Search \((x, k)\)

1. while \(x \neq \text{NULL}\) and \(k \neq x\.\text{key}\) do
2. if \(k < x\.\text{key}\) then
3. \(x \leftarrow x\.\text{left}\)
4. else
5. \(x \leftarrow x\.\text{right}\)
6. return \(x\)

### Successor \((x)\)

1. if \(x\.\text{right} \neq \text{NULL}\) then
2. return \(\text{Minimum}(x\.\text{right})\)
3. \(y \leftarrow x\.p\)
4. while \(y \neq \text{NULL}\) and \(x = y\.\text{right}\) do
5. \(x \leftarrow y\)
6. \(y \leftarrow y\.p\)
7. return \(y\)

### Minimum \((x)\)

1. while \(x\.\text{left} \neq \text{NULL}\) do
2. \(x \leftarrow x\.\text{left}\)
3. return \(x\)

### Maximum \((x)\)

1. while \(x\.\text{right} \neq \text{NULL}\) do
2. \(x \leftarrow x\.\text{right}\)
3. return \(x\)

### Predecessor \((x)\)

1. if \(x\.\text{left} \neq \text{NULL}\) then
2. return \(\text{Maximum}(x\.\text{left})\)
3. \(y \leftarrow x\.p\)
4. while \(y \neq \text{NULL}\) and \(x = y\.\text{left}\) do
5. \(x \leftarrow y\)
6. \(y \leftarrow y\.p\)
7. return \(y\)
Binary Search Tree Operations

**INSERT** \((T, k)\)

1: \(y = \text{NULL} \) // father of new node
2: \(x = T.\text{root} \)
3: while \(x \neq \text{NULL} \) do
4: \(y \leftarrow x \)
5: if \(k < x.\text{key} \) then
6: \(x \leftarrow x.\text{left} \)
7: else
8: \(x \leftarrow x.\text{right} \)
9: \(z \leftarrow \text{new node with } z.\text{key} = k, \)
\(z.\text{p} = y, z.\text{left} = z.\text{right} = \text{NULL} \)
10: if \(y = \text{NULL} \) then
11: \(T.\text{root} \leftarrow z \) // tree was empty
12: else if \(k < y.\text{key} \) then
13: \(y.\text{left} = z \)
14: else
15: \(y.\text{right} = z \)

**TRANSPLANT** \((T, u, v)\)

1: if \(u.\text{p} = \text{NULL} \) then
2: \(T.\text{root} \leftarrow v \)
3: else if \(u = u.\text{p}.\text{left} \) then
4: \(u.\text{p}.\text{left} \leftarrow v \)
5: else
6: \(u.\text{p}.\text{right} \leftarrow v \)
7: if \(v \neq \text{NULL} \) then
8: \(v.\text{p} \leftarrow u.\text{p} \)

**DELETE** \((T, z)\)

1: if \(z.\text{left} = \text{NULL} \) then
2: \(\text{TRANSPLANT}(T, z, z.\text{right}) \)
3: else if \(z.\text{right} = \text{NULL} \) then
4: \(\text{TRANSPLANT}(T, z, z.\text{left}) \)
5: else
6: \(y \leftarrow \text{MINIMUM}(z.\text{right}) \)
7: if \(y.\text{p} \neq z \) then
8: \(\text{TRANSPLANT}(T, y, y.\text{right}) \)
9: \(y.\text{right} \leftarrow z.\text{right} \)
10: \(y.\text{right}.\text{p} \leftarrow y \)
11: \(\text{TRANSPLANT}(T, z, y) \)
12: \(y.\text{left} \leftarrow z.\text{left} \)
13: \(y.\text{left}.\text{p} \leftarrow y \)
Binary Search Tree

Running time of operations

$h$: height of tree = number of edges on longest path from root to a leaf

<table>
<thead>
<tr>
<th>operation</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEARCH</td>
<td>$O(h)$</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>$O(h)$</td>
</tr>
<tr>
<td>MAXIMUM</td>
<td>$O(h)$</td>
</tr>
<tr>
<td>PREDECESSOR</td>
<td>$O(h)$</td>
</tr>
<tr>
<td>SUCCESSOR</td>
<td>$O(h)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$O(h)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$O(h)$</td>
</tr>
</tbody>
</table>

Tree height

best case: $h = \Omega(\log n)$
worst case: $h = O(n)$

*(chalk board)*
Why red-black trees?

- binary search tree, that is **approximately balanced**
- height $h$ of red-black tree with $n$ nodes is $O(\log n)$
- operations that change tree (INSERT and DELETE) **maintain** that property
- each node now has a fifth attribute: $color \in \{\text{red, black}\}$.
- leaves contain no data (convenience assumption)
## Red-Black Trees

### Why red-black trees?

- binary search tree, that is **approximately balanced**
- height $h$ of red-black tree with $n$ nodes is $O(\log n)$
- operations that change tree (**INSERT** and **DELETE**) **maintain** that property
- each node now has a fifth attribute: $color \in \{\text{red, black}\}$.
- leaves contain no data (convenience assumption)

### Definition 3 (Red-black tree)

A red-black tree is a binary search tree that satisfies the following **red-black properties**:

1. The root is black.
2. Every leaf is black.
3. If a node is red then both(!) its children are black.
4. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.
Each node in this graph without a left (right) son has actually a left (right) son (here not drawn), which is black and contains no keys. Only internal nodes are drawn here. All internal nodes have two children.
Lemma 5

The height $h$ of red-black tree with $n$ internal nodes satisfies $h \leq 2 \log_2(n + 1)$.

Proof.

Let $bh(x)$ be the **black height** of node $x$: The number of black nodes on any(!) path from, but excluding, $x$ down to a leaf.

**Claim:** For any node $x$, the subtree rooted at $x$ contains at least $2^{bh(x)} - 1$ internal nodes.

Proof by induction on the height of $x$. Initial case: $x$ is a leaf and has height 0. Then $bh(x) = 0$ and the subtree at $x$ contains $2^0 - 1 = 0$ internal nodes. Inductive step: $x$ has height >0 and has therefore two sons of height 1 less than $x$. The black heights of the two sons are at most 1 less than the black height of $x$. By inductive hypothesis, the subtrees rooted at the children of $x$ contain together at least

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

internal nodes. Adding 1 for the internal node $x$ this concludes the inductive step.

Let $h$ be the height of the tree. Then the black height of the root must be at least $h/2$ as at least half of the nodes on any path from the root to a leaf must be black (red-black property 3). Therefore,

$$n \geq 2^{h/2} - 1$$

or equivalently,

$$h \leq 2 \log_2(n + 1).$$
Red-Black Trees

Red-black trees

- Because a red-black tree is a BST, the operations
  - SEARCH
  - MINIMUM, MAXIMUM
  - PREDECESSOR, SUCCESSOR
  take time $O(h)$, which is now $O(\log n)$.
- INSERT, and DELETE now require extra work to keep red-black properties, but can also be done in $O(\log n)$ time.
- A basic manipulation step to fix violated red-black properties after INSERT and DELETE is the rotation.
A rotation maintains the BST property and changes the pointer structure only locally (on at most 4 nodes).
The right tree is the result of operation **RIGHT-ROTATE** on the left tree on node $y$. The left tree is the result of operation **LEFT-ROTATE** on the right tree on node $x$. 

---

**Example 6 (Rotation)**
**Binary Search Tree Operations**

**RB-INSERT**(\(T, k\))

1. \(y = NULL\) // father of new node
2. \(x = T.root\)
3. **while** \(x \neq NULL\) **do**
4. \(y \leftarrow x\)
5. **if** \(k < x.key\) **then**
6. \(x \leftarrow x.left\)
7. **else**
8. \(x \leftarrow x.right\)
9. \(z \leftarrow\) new node with \(z.key = k\),
   \(z.p = y, z.left = z.right = NULL,\)
   \(z.color = red\)
10. **if** \(y = NULL\) **then**
11. \(T.root \leftarrow z\) // tree was empty
12. **else if** \(k < y.key\) **then**
13. \(y.left = z\)
14. **else**
15. \(y.right = z\)
16. **RB-INSERT-FIXUP**(\(T, z\))

**RB-INSERT-FIXUP**(\(T, z\))

1. **while** \(z.p.color = red\) **do**
2. **if** \(z.p = z.p.p.left\) **then**
3. \(y \leftarrow z.p.p.right\) // \(y\) is uncle of \(z\)
4. **if** \(y.color = red\) **then** // case 1
5. \(z.p.color \leftarrow y.color \leftarrow black\)
6. \(z.p.p.color \leftarrow red\)
7. \(z \leftarrow z.p.p\)
8. **else**
9. **if** \(z = z.p.right\) **then** // case 2
10. \(z \leftarrow z.p\)
11. **LEFT-ROTATE**(\(T, z\))
   // case 3: \(z = z.p.left\)
12. \(z.p.color \leftarrow black\)
13. \(z.p.p.color \leftarrow red\)
14. **RIGHT-ROTATE**(\(T, z.p.p\))
15. **else**
16. // same as above with 'right' and 'left' exchanged
17. \(T.root.color = black\)
Example 7 (Fixing red-black properties after inserting key 4)
Red-Black Trees

**RB-INSERT**

The running time of RB-INSERT is $O(h) = O(\log n)$ as the depth of $z$ decreases with each execution of the loop in line 1 of RB-INSERT-FIXUP.

**RB-DELETE($T, z$)**

Can be defined as a modification of DELETE with a fixup that maintains the red-black properties. The running time of RB-DELETE is then also $O(h) = O(\log n)$. 