Summary previous lectures

Implementations

Methods

find
insert
delete
Traverse (in-order, pre-order, post-order)
show
Transform a list into a binary search tree (using successive insertion):
10, 4, 7, 15, 3, 18, 16

- Random list → balanced tree
- Search $O(\log N)$, insert $O(\log N)$, etc.

If we now suppose that the data are already sorted or inversely sorted
3, 4, 7, 10, 15, 16, 18

- Tree is unbalanced!
- Search $O(N)$
Tree becomes unbalanced or partially unbalanced if:

- the data are (almost) sorted or inversely (almost) sorted
- A small or large key value at a given node prevents insertion in the left or right subtree (for example a root of 3 allows only 2 nodes to be inserted on the left)

The search procedure can degenerate to O(N)

To guarantee the quick O(\log N) search, we need to ensure that the tree is always balanced (each node has roughly the same number of descendents on its left side as it has on its right side).

Solution → Red-Black Tree

- It is a BST with some added features (a node can be black or red, for example using the boolean field isRed)
- Using a Red-Black Tree balance is achieved during insertion (or deletion)
- Red-Black are not trivial to understand and very complex to implement
Red-Black Tree rules

1- Every node is either red or black
2- The root is always black
3- If a node is red, its children must be black
4- Every path from the root to a leaf, or to a null child, must contain the same number of black nodes (black height is the same)

Examples of rule violations:
Red-Black Tree- basics

- Color of inserted node is always red by default
- How to fix rule violations?
  - You can change the colors of the nodes
  - You can perform rotation (re-structuration of the tree)
- Examples of basic manipulations

To do: Test Java applet RBTree.html

- Experiment 1- Insert 50, 25, 75
- Experiment 2- Rotate right, then rotate Left
- Experiment 3- Insert 12...need a color flip first
The word rotation is misleading, it is only relationship between nodes that changes.

- **Simple rotation**

- **Crossover rotation**

The inside grandchild (here 4) of the node that leads the rotation (here 10), is always disconnected from its parents (here 3) and reconnected to its grandparent (10).
Red-Black Tree- more on rotations

- It is possible to rotate entire subtree
- The relations of the nodes within each subtree are not affected
Red-Black Tree- inserting a new node

- Color of inserted node is always **red** by default
- The insertion is first similar to the one for BST, find the position where the node should be inserted
- Three main stages to ”automatically” fix violations of the rules
  - a- **Color flips on the way down**
    Every time the insertion routine encounters a **black** node that has **two red** children, it must change the children to black and the parent to red (unless the latter is the root)- example:

- **b- Rotations after the node is inserted**
- **c- Rotations on the way down**
Red-Black Tree- inserting a new node

- b- Rotation after the node is inserted
  Three main 3 post-insertion possibilities after inserting X

1- If P (parent) is **black** → here everything works fine, just insert X
2- If P is **red** and X is an outside grandchild of G
3- If P is **red** and X is an inside grandchild of G
1. Rotation after the node is inserted

2. If P is red and X is an outside grandchild of G
   - i. switch the color of G (Grandparent of X)
   - ii. switch the color of P (parent of X)
   - iii. rotate with G at the top, in the direction that raises X

Example (after inserting 1)
b- Rotation after the node is inserted

3- If P is red and X is an inside grandchild of G
   - i- switch the color of G (Grandparent of X)
   - ii- switch the color of X
   - iii- rotate with P at the top, in the direction that raises X
   - iv- rotate with G at the top, in the direction that raises X

Example (After inserting 3)
Red-Black Tree- inserting a new node

- **c- Rotation on the way down**
  - Takes place before the node is inserted on the way down of the search
  - An offending node may happen (after a color flip) causing a red-red conflict
- **First Possibility (outside grandchild)**
  - i- switch the color of G (Grandparent of X- offending node)
  - ii- switch the color of P
  - iii- rotate with G at the top, in the direction that raises X

Example: insert 3

Tree is balanced!
Red-Black Tree- inserting a new node

- **c- Rotation on the way down**
- **Second Possibility (inside grandchild)**
  - i- switch the color of G (Grandparent of X- offending node)
  - ii- switch the color of X
  - iii- rotate with P at the top, in the direction that raises X
  - iv- rotate with G at the top, in the direction that raises X

Example: insert 28

*a)*

*b)*

*c)*

**Rq:** there is another color flip 25,50 before insertion
Tree is balanced!
Red-Black Tree - Example

**Insert**: 1, 2, 3, 4, 5, 6, 7

1. Insert 1:

```
1
```

2. Insert 2:

```
1
  2
```

3. Insert 3:

```
1
  3
  2
```

4. Insert 4 with recoloring:

```
1
  3
  2
  4
```

5. Insert 5 with Left rotation:

```
1
  3
  2
  4
  5
```
Red-Black Tree Example

**Insert:** 1, 2, 3, 4, 5, 6, 7
Red-Black Tree- Example

- Tree is balanced $O(\log N)$ for search, insert, delete!
- Try the Java applet RBTree.html
- Another one available on-line: http://gauss.ececs.uc.edu/RedBlack/redblack.html