Bachelor-Seminar “Ausgewählte Kapitel der Informatik”

Search Trees: Traversal and Maintenance

Kevin Straßmeir
7. Februar 2013
Agenda

1. **Things to know about a tree**
2. Binary tree (BT)
3. Binary search tree (BST)
4. Self-balancing BSTs
   - 4.1. Red-Black-trees
5. Generalized STs
   - 5.1. B-trees
   - 5.2. 2-3-4 trees
6. Relation between red-black trees and 2-3-4 trees
Tree as a data structure

- Used to represent relationships and hierarchy
  - Parent-child, ancestor-descendant, siblings
- Consists of nodes and edges
- Has a root
- Is recursive
  - A tree is a node, possibly empty, with a list of nodes, each of which is a tree
Important terms: level, height, size, n-ary tree

- **Level of a node**
  Number of nodes on path from root to the node (root level 1)

- **Height of a tree**
  Maximum level of nodes

- **Size of a tree**
  Number of nodes in a tree

- **N-ary trees**
  Trees whose nodes cannot have more than n children
Agenda

1. Things to know about a tree
2. **Binary tree (BT)**
3. Binary search tree (BST)
4. Self-balancing BSTs
   4.1. Red-Black-trees
5. Generalized STs
   5.1. B-trees
   5.2. 2-3-4 trees
6. Relation between red-black trees and 2-3-4 trees
Important terms for binary trees (BTs)

- **Balanced**
  
  Difference in height between right and left subtree of any node $\leq 1$

- **Full**
  
  All nodes at a level $< \text{height}$ have 2 children

- **Complete**
  
  Full to height-1, then filled from left to right
Java implementation of BT

- Array based

```java
public class BinaryTreeArray {
    int root;

    // free entries
    int free;

    // No. of entries used
    int size;

    // data of nodes
    char data[];

    // subtrees
    int left[];
    int right[];
}
```

<table>
<thead>
<tr>
<th>data</th>
<th>M</th>
<th>I</th>
<th>L</th>
<th>U</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>right</td>
<td>-1</td>
<td>5</td>
<td>3</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

- Reference/pointer based

```java
public class BinaryTreePointer<E> implements Serializable {
    protected static class Node<E> implements Serializable {
        protected E data;
        protected Node<E> left;
        protected Node<E> right;
        public Node(E data) {
            this.data = data;
            left = null;
            right = null;
        }
    }
}
```
Four ways to traverse a BT

1. **Pre-order**: data → left → right
2. **In-order**: left → data → right
3. **Post-order**: left → right → data
4. **Level-order**: Level by level from left to right
Four ways to traverse a BT

1. **Pre-order**: data → left → right
   
   9, 3, 1, 4, 5, 6, 9, 2

2. **In-order**: left → data → right
   
   4, 1, 3, 5, 9, 9, 6, 2

3. **Post-order**: left → right → data
   
   4, 1, 5, 3, 9, 2, 6, 9

4. **Level-order**: Level by level from left to right
   
   9, 3, 6, 1, 5, 9, 2, 4
Agenda

1. Things to know about a tree
2. Binary tree (BT)
3. **Binary search tree (BST)**
4. Self-balancing BSTs
   4.1. Red-Black-trees
5. Generalized STs
   5.1. B-trees
   5.2. 2-3-4 trees
6. Relation between red-black trees and 2-3-4 trees
Binary Search Tree (BST) property

- All values/keys smaller than the root are stored in left subtree and bigger ones in the right subtree
Searching in BST

- Recursive

```java
search(node n, BST t) {
    if(t == null)
        return ERROR;
    if(n == t.root)
        return SUCCESS;
    if(n < t.root)
        search(n, t.leftSubT);
    else
        search(n, t.rightSubT);
}
```

- Iterative

```java
search(node n, BST t) {
    while(t != null) {
        if(n == t.root)
            return SUCCESS;
        if(n < t.root)
            t = t.leftSubT;
        else
            t = t.rightSubT;
    } // t == null
    return ERROR;
}
```
Inserting node into BST

```java
void insert(node n, BST t) {
    if (t == null) {
        // Construct node
        t.root = new Node(n);
        return SUCCESS;
    }
    if (n < t.root)
        insert(n, t.leftSubT);
    else if (n > t.root)
        insert(n, t.rightSubT);
    else
        // n already in tree
        return ERROR;
}
```
Deleting from BST: three cases

```java
int delete(node n, BST t) {
    if (n == t.root) {
        // case 1: no children
        if (isLeaf(t.root()))
            t = null;
        // case 2: one child
        else if (t.leftSubT != null && t.rightSubT == null)
            t = t.leftSubT;
        else if (t.leftSubT == null && t.rightSubT != null)
            t = t.rightSubT;
        // case 3: two children
        else {
            node newRoot = findMin(t.rightSubT);
            t.root = newRoot;
            t.rightSubT = delete(newRoot, t.rightSubT);
        }
        return SUCCESS;
    }
}
```

Diagram:
```
  5
 /   \\
 3     8
|     /  \
1     4   7
     |   /  \\n  2   9
```
Deleting from BST: three cases

```c
int delete(node n, BST t) {
    if(n == t.root) {
        // case 1: no children
        if(isLeaf(t.root))
            t = null;
        // case 2: one child
        else if(t.leftSubT != null
            && t.rightSubT == null)
            t = t.leftSubT;
        else if(t.leftSubT == null
            && t.rightSubT != null)
            t = t.rightSubT;
        // case 3: two children
        else {
            node newRoot = findMin(t.rightSubT);
            t.root = newRoot;
            t.rightSubT = delete(newRoot, t.rightSubT);
        }
    return SUCCESS;
}
```
Deleting from BST: three cases

```java
delete(node n, BST t) {
    if (n == t.root) {
        // case 1: no children
        if (isLeaf(t.root))
            t = null;
        // case 2: one child
        else if (t.leftSubT != null
            && t.rightSubT == null)
            t = t.leftSubT;
        else if (t.leftSubT == null
            && t.rightSubT != null)
            t = t.rightSubT;
        // case 3: two children
        else {
            node newRoot = findMin(t.rightSubT);
            t.root = newRoot;
            t.rightSubT = delete(newRoot, t.rightSubT);
        }
        return SUCCESS;
    }
}
```
Deleting from BST: three cases

```java
void delete(node n, BST t) {
    if (n == t.root) {
        // case 1: no children
        if (isLeaf(t.root)) {
            t = null;
        }
        // case 2: one child
        else if (n.leftSubTree != null && n.rightSubTree == null) {
            t = n.leftSubTree;
        }
        else if (n.leftSubTree == null && n.rightSubTree != null) {
            t = n.rightSubTree;
        }
        // case 3: two children
        else {
            node newRoot = findMin(t.rightSubTree);
            t.root = newRoot;
            t.rightSubTree = delete(newRoot, t.rightSubTree);
        }
    }
    return SUCCESS;
}
```
Running time of BST operations: $O(\text{height})$

- **Best case:** $\text{height} = \log_2(\text{size} + 1)$

- **Worst case:** $\text{height} = \text{size}$
  \[\rightarrow\text{skewed tree}\]
Agenda

1. Things to know about a tree
2. Binary tree (BT)
3. Binary search tree (BST)
4. **Self-balancing BSTs**
   4.1. Red-Black-trees
5. Generalized STs
   5.1. B-trees
   5.2. 2-3-4 trees
6. Relation between red-black trees and 2-3-4 trees
Motivation for self-balancing BST

• **Issue**
  
  Performance of BST is proportional to its height

• **Approach**
  
  Keep height of tree as short as possible

• **Solution**
  
  Rotate around nodes for

  – keeping heights of left and right subtrees (nearly) equal

  – maintaining BST property
Red-black tree

1. Nodes either red or black
2. Root always black
3. Red node always has black children (null reference considered to refer to black node)
4. Number of black nodes in any path from root to leaf is the same
   - Only black nodes determine height
     → Red-black tree must be balanced
Inserting into red-black tree

- Algorithm follows same recursive search used for BST to reach insertion point
- If initial tree is empty, black root node is constructed
- If leaf is found, new node is inserted as red node
  - If the parent is black → done
  - Otherwise there are three cases to consider; more than one can occur after inserting
Inserting case 1: Red parent has red sibling

**Invariants**

1. Nodes either red or black
2. Root always black
3. Red node always has black children
4. Number of black nodes in any path from root to leaf is the same
Inserting case 1: Red parent has red sibling

Invariants

1. Nodes either red or black
2. Root always black
3. Red node always has black children
4. Number of black nodes in any path from root to leaf is the same
Inserting case 1: Red parent has red sibling

1. Change color of parent and its sibling to black
2. Change color of grandparent to red

**Invariants**

1. Nodes either red or black
2. **Root always black**
3. Red node always has black children
4. Number of black nodes in any path from root to leaf is the same
Inserting case 1: Red parent has red sibling

1. Change color of parent and its sibling to black
2. Change color of grandparent to red
3. Change color of root to black

**Invariants**

1. Nodes either red or black
2. Root always black
3. Red node always has black children
4. Number of black nodes in any path from root to leaf is the same
Case 2: Red parent, no sibling, same direction

Invariants

1. Nodes either red or black
2. Root always black
3. Red node always has black children
4. Number of black nodes in any path from root to leaf is the same
Case 2: Red parent, no sibling, same direction

Invariants

1. Nodes either red or black
2. Root always black
3. Red node always has black children
4. Number of black nodes in any path from root to leaf is the same
Case 2: Red parent, no sibling, same direction

1. Change color of parent to black
2. Change color of grandparent to red

Invariants

1. Nodes either red or black
2. Root always black
3. Red node always has black children
4. Number of black nodes in any path from root to leaf is the same
Case 2: Red parent, no sibling, same direction

1. Change color of parent to black
2. Change color of grandparent to red
3. Rotate left/right around grandparent

**Invariants**

1. Nodes either red or black
2. Root always black
3. Red node always has black children
4. Number of black nodes in any path from root to leaf is the same
Case 2: Red parent, no sibling, same direction

1. Change color of parent to black
2. Change color of grandparent to red
3. Rotate left/right around grandparent

**Invariants**

1. Nodes either red or black
2. Root always black
3. Red node always has black children
4. Number of black nodes in any path from root to leaf is the same
Case 3: Red parent, no sibling, other direction

**Invariants**

1. Nodes either red or black
2. Root always black
3. Red node always has black children
4. Number of black nodes in any path from root to leaf is the same
Case 3: Red parent, no sibling, other direction

**Invariants**

1. Nodes either red or black
2. Root always black
3. **Red node always has black children**
4. Number of black nodes in any path from root to leaf is the same
Case 3: Red parent, no sibling, other direction

1. Rotate left/\textbf{right} around the parent

\textbf{Invariants}

1. Nodes either red or black
2. Root always black
3. \textbf{Red node always has black children}
4. Number of black nodes in any path from root to leaf is the same
Case 3: Red parent, no sibling, other direction

1. Rotate left/right around the parent
2. Continue with case 2

Invariants

1. Nodes either red or black
2. Root always black
3. Red node always has black children
4. Number of black nodes in any path from root to leaf is the same
Case 3: Red parent, no sibling, other direction

1. Rotate left/right around the parent
2. Continue with case 2
   1. Change color of parent to black
   2. Change color of grandparent to red
   3. Rotate left/right around grandparent

Invariants

1. Nodes either red or black
2. Root always black
3. Red node always has black children
4. Number of black nodes in any path from root to leaf is the same
Performance of red-black tree

- Average performance is significantly better than worst-case performance and close to that of complete BT
- Complexity for searching, inserting, deleting $O(\log n)$
- Superior to AVL tree, because red-black tree can be more 'unbalanced' and thus, needs to be re-balanced less frequently
Applications

• For BSTs
  – Tree sort

• For self-balancing BSTs
  – E.g., Priority queues
  – Associative arrays
  – Java API: TreeMap class is Red-Black tree based
Agenda

1. Things to know about a tree
2. Binary tree (BT)
3. Binary search tree (BST)
4. Self-balancing BSTs
   4.1. Red-Black-trees
5. Generalized STs
   5.1. B-trees
   5.2. 2-3-4 trees
6. Relation between red-black trees and 2-3-4 trees
B-tree

- Designed for building indexes to large databases stored on hard disks → minimizing number of disk reads
- Complexity for searching, inserting, deleting $O(\log n)$
- Max number of children is order of B-tree
- Keeps properties with splitting/merging nodes
2-3-4 tree

- B-tree of order 4
- Node can be 2-, 3- or 4-node
- 4-node has space for three data items and four children
Inserting into 2-3-4 tree

- Search for point of insertion
- If 4-node is encountered, split it in advance → guarantees there is space for inserting when reaching a leaf
Inserting into 2-3-4 tree, example

- **Search for point of insertion**

- If 4-node is encountered, split it in advance $\rightarrow$ guarantees there is space for inserting when reaching a leaf
Inserting into 2-3-4 tree, example

- Search for point of insertion

- If 4-node is encountered, split it
Inserting into 2-3-4 tree, example

- **Split:**
  - Move middle data item to parent
  - Adjust edges
Inserting into 2-3-4 tree, example

- Split:
  - Move middle data item to parent
  - Adjust edges
Inserting into 2-3-4 tree, example

- **Search for point of insertion**

- If 4-node is encountered, split it in advance
Agenda

1. Things to know about a tree
2. Binary tree (BT)
3. Binary search tree (BST)
4. Self-balancing BSTs
   4.1. Red-Black trees
5. Generalized STs
   5.1. B-trees
   5.2. 2-3-4 trees
6. Relation between red-black trees and 2-3-4 trees
Relation between red-black tree and 2-3-4 tree

• A Red-Black tree is a binary-tree equivalent of a 2-3-4 tree

• 2-node is a black node

• 3-node can be represented as either a black node with a left red child or a black node with a right red child

• 4-node is a black node with two red children
Sources

- Ang, Chuan Heng, Lecture notes to CS20120: Data Structures and Algorithms II, Semester 2, AY 2011/2012, NUS

- Kriegel, Hans-Peter, Skript zur Vorlesung Algorithmen und Datenstrukturen, SoSe 2012, LMU