Binary Trees

• Rooted, acyclic graphs
  – Each node organized into a hierarchy
    • 1 Parent node
    • 2 Child nodes

• Reading:
  – ODS Section 6.1
Binary Search Trees

• Binary Tree with an invariant:
  \[ \text{left.value} < \text{this.value} < \text{right.value} \]

• Reading:
  – ODS Section 6.2
BinarySearchTree

- `add()`/`remove()`/`contains()` operations depend on searching the tree.

- Length of the search path in a perfectly balanced tree is $O(\log n)$.

- Binary tree may be completely imbalanced, making the length $O(n)$. 
A **BinarySearchTree** implements the **SSet** interface and supports the operations:

- **find(x)**/**add(x)**/**remove(x)** in **O(n)** time per operation.
Random Binary Search Trees

• Random permutation of elements put into a Binary Search Tree
  – Achieve balance through randomness

• Reading:
  – ODS Section 7.1
Random Binary Search Trees

- 15! permutations of the elements {1...15}
  - 1 permutation leads to the tree on the left
  - 21964800 permutations lead to the tree on the right
- Balanced trees are much more likely
RandomBinarySearchTree

Lemma:
The expected length of the search path in a RandomBinarySearchTree is at most $2\ln(n)+O(1)$.

Theorem:
A RandomBinarySearchTree can be constructed in $O(n\log n)$ time and supports the operation:
- $\text{find}(x)$ in $O(\log n)$ expected time.
Treaps

• Randomized BinarySearchTree
  – Stores (key, priority) pairs
  – BST on keys
  – Heap on priorities

• Reading:
  – ODS Section 7.2
Treaps

Theorem:

A Treap implements the SSet interface and supports the operations:

- \texttt{find(x)}/\texttt{add(x)}/\texttt{remove(x)} in $O(\log n)$ expected time per operation.
Scapegoat Trees

• Deterministically balanced BST
  – Height maintained within $O(\log n)$
  – Rebuilding operations

• Reading:
  – ODS Chapter 8
# Binary Search Trees

<table>
<thead>
<tr>
<th>Operation</th>
<th>BST</th>
<th>RBST/Treaps</th>
<th>Scapegoat Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>find(x)</td>
<td>$O(n)$</td>
<td>$O(\log n)$ expected</td>
<td>$O(\log n)$ amortized</td>
</tr>
<tr>
<td>add(x)</td>
<td>$O(n)$</td>
<td>$O(\log n)$ expected</td>
<td>$O(\log n)$ amortized</td>
</tr>
<tr>
<td>remove(x)</td>
<td>$O(n)$</td>
<td>$O(\log n)$ expected</td>
<td>$O(\log n)$ amortized</td>
</tr>
</tbody>
</table>
Scapegoat Trees

Theorem:

A ScapegoatTree implements the SSet interface. Ignoring the cost of rebuilding() operations, a ScapegoatTree supports the operations:

- **find(x)/add(x)/remove(x)** in \(O(\log n)\) time per operation.

In addition, starting with an empty ScapegoatTree, any sequence of \(m\) add(x) and remove(x) operations results in a total of \(O(m \log n)\) time spent during all calls to rebuild().
2-4 Trees

• Each node has 2, 3, or 4 children
• Every leaf has the same depth
• Deterministic height bound

• Reading:
  – ODS Section 9.1
  – Data Struc. & Alg. Section 11.5
## Sorted Sets

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skiplist</td>
<td>$O(\log n)$ expected</td>
</tr>
<tr>
<td>Treaps</td>
<td>$O(\log n)$ expected</td>
</tr>
<tr>
<td>Scapegoat Trees</td>
<td>$O(\log n)$ amortized</td>
</tr>
<tr>
<td>2-4/RedBlack Trees</td>
<td>$O(\log n)$ worst case</td>
</tr>
</tbody>
</table>
RedBlack Trees

• Implementation of 2-4 Trees using only binary nodes

• Height is $\leq 2\log(n)$

• Reading:
  – ODS Section 9.2
  – Data Struc. & Alg. Section 11.6
Theorem:

A **RedBlackTree** implements the **SSet** interface. A RedBlackTree supports the operations:

- **find(x)**/**add(x)**/**remove(x)** in \(O(\log n)\) time per operation.
Appendix: RedBlack Tree Insertion

• add(x):
  – Find the external node to put x
  – Insert x, as a new red node
  – 3 cases...

• Case 0: x's parent is black
  – Complete.
Appendix: RedBlack Tree Insertion

• Case 1: red-red w/ black uncle
  – Label nodes:
    • x: new node (red)
    • y: x's parent (red)
    • z: x's grandparent (black)
  – 4 scenarios...

1. x is red
   - y is red
   - z is black

2. x is red
   - y is red
   - z is red

3. x is red
   - y is black
   - z is red

4. x is black
   - y is red
   - z is red
Appendix: RedBlack Tree Insertion

- Case 1 solution: restructure 3 nodes
  - Order x, y, z according to in-order traversal
  - Label them a, b, c
  - Structure/colour into tree as follows
Appendix: RedBlack Tree Insertion

- Case 2: **red-red w/ red uncle**
  - Corresponds to overflow in 2-4 tree node
    - Recolour accordingly
    - Repeat up the tree if necessary