Balanced Search Trees

2-3-4 trees
red-black trees

References: Algorithms in Java (handout)
Balanced search trees

Dynamic sets

- Search
- Insert
- Delete
- Maximum
- Minimum
- Successor(x)  (find minimum element ≥ x)
- Predecessor(x)  (find maximum element ≤ x)

This lecture: 2-3-4 trees, red-black trees

Next time: Tiered vektor (not a binary search tree, but maintains a dynamic set).

In two weeks time: Splay trees
Dynamic set implementations

Worst case running times

<table>
<thead>
<tr>
<th>Implementation</th>
<th>search</th>
<th>insert</th>
<th>delete</th>
<th>minimum</th>
<th>maximum</th>
<th>successor</th>
<th>predecessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>linked lists</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>ordered array</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>BST</td>
<td>O(h)</td>
<td>O(h)</td>
<td>O(h)</td>
<td>O(h)</td>
<td>O(h)</td>
<td>O(h)</td>
<td>O(h)</td>
</tr>
</tbody>
</table>

In worst case \( h=n \).

In best case \( h= \log n \) (fully balanced binary tree)

**Today:** How to keep the trees balanced.
2-3-4 trees
2-3-4 trees

2-3-4 trees. Allow nodes to have multiple keys.

**Perfect balance.** Every path from root to leaf has same length.

Allow 1, 2, or 3 keys per node

- 2-node: one key, 2 children
- 3-node: 2 keys, 3 children
- 4-node: 3 keys, 4 children
Searching in a 2-3-4 tree

Search.

- Compare search key against keys in node.
- Find interval containing search key
- Follow associated link (recursively)

Ex. Search for L
Insertion in a 2-3-4 tree
Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.

Ex. Insert B
Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node

Ex. Insert B

B fits here
Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node

Ex. Insert X

```
A  D  F  G  J  L  N  Q  S  T  Y  Z
   C  E
      K  R
```

larger than R
larger than U
X not found
Insertion in a 2-3-4 tree

Insert.

• Search to bottom for key.
• 2-node at bottom: convert to 3-node
• 3-node at bottom: convert to 4-node

Ex. Insert X

- X fits here
- Larger than R
- Larger than W
Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node

Ex. Insert H
Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node
- 4-node at bottom: ??

Ex. Insert H

H does not fit here!
Splitting a 4-node in a 2-3-4 tree

Idea: split the 4-node to make room

Problem: Doesn’t work if parent is a 4-node

Solution 1: Split the parent (and continue splitting while necessary).

Solution 2: Split 4-nodes on the way down.
Splitting 4-nodes in a 2-3-4 tree

**Idea:** split 4-nodes on the way down the tree.
- Ensures last node is not a 4-node.
- Transformations to split 4-nodes:

**Invariant.** Current node is not a 4-node.

**Consequence.** Insertion at bottom is easy since it's not a 4-node.
Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node
- 4-node at bottom: ??

Ex. Insert H

![Diagram of 2-3-4 tree with nodes labeled A to Y, Z. The node with key H is inserted, causing a 4-node to be converted to a 3-node, with a 3-node becoming a 4-node at the bottom.](image-url)
Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node
- 4-node at bottom: ??

Ex. Insert H

```
A  D  F  J  L  N  Q  S  V  Y  Z
```

```
X
```

```
KR
```

```
CEG
```

```
M  O
```

```
`
Insertion in a 2-3-4 tree

Insert.

- Search to bottom for key.
- 2-node at bottom: convert to 3-node
- 3-node at bottom: convert to 4-node
- 4-node at bottom: ??

Ex. Insert H
Splitting 4-nodes in a 2-3-4 tree

Local transformations that work anywhere in the tree.

Ex. Splitting a 4-node attached to a 2-node
Splitting 4-nodes in a 2-3-4 tree

Local transformations that work anywhere in the tree

Ex. Splitting a 4-node attached to a 3-node

could be huge

unchanged
Splitting 4-nodes in a 2-3-4 tree

Local transformations that work anywhere in the tree.

Splitting a 4-node attached to a 4-node never happens when we split nodes on the way down the tree.

Invariant. Current node is not a 4-node.
Insertion 2-3-4 trees

1. Insert U
2. Insert G
3. Insert G
4. Insert U
5. Insert U
6. Insert T
7. Insert T
Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key

Ex. Delete minimum
Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key

Ex. Delete minimum
Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key
- 2-node??

Ex. Delete minimum

Delete B?
Deletions in 2-3-4 trees

Idea: On the way down maintain the invariant that current node is not a 2-node.

- Child of root and root is a 2-node:

- on the way down:
Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key
- 2-node: split/merge on way down.

Ex. Delete minimum
Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key
- 2-node: split/merge on way down.

Ex. Delete minimum

```
28
S  V
F  G  J
K  R
E
M  O
X
B C D
L
N
Q
S V
Y Z
```
Deletions in 2-3-4 trees

Delete minimum:

- minimum always in leftmost leaf
- If 3- or 4-node: delete key
- 2-node: split/merge on way down.

Ex. Delete minimum
Deletions in 2-3-4 trees

Delete:

```
B  D  F  G  J
C  E
L  N  Q
M  O
X
K  R
```

30
Deletions in 2-3-4 trees

Delete:

• During search maintain invariant that current node is not a 2-node
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K

- Find successor
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K

- Find successor
Deletions in 2-3-4 trees

Delete:

• During search maintain invariant that current node is not a 2-node

• If key is in a leaf: delete key

• Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K

• Find successor
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K
- Find successor
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K

- Find successor
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K

- Find successor
- Delete L from leaf
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K

- Find successor
- Delete L from leaf
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K

- Find successor
- Delete L from leaf
- Replace K with L
Deletions in 2-3-4 trees

Delete:

- During search maintain invariant that current node is not a 2-node
- If key is in a leaf: delete key
- Key not in leaf: replace with successor (always leaf in subtree) and delete successor from leaf.

Ex. Delete K

- Find successor
- Delete L from leaf
- Replace K with L
2-3-4 Tree: Balance

Property. All paths from root to leaf have same length.

Tree height.

Worst case: \( \lg N \) [all 2-nodes]

Best case: \( \log_4 N = 1/2 \lg N \) [all 4-nodes]

Between 10 and 20 for a million nodes.

Between 15 and 30 for a billion nodes.
# Dynamic set implementations

## Worst case running times

<table>
<thead>
<tr>
<th>Implementation</th>
<th>search</th>
<th>insert</th>
<th>delete</th>
<th>minimum</th>
<th>maximum</th>
<th>successor</th>
<th>predecessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>linked lists</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>ordered array</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>BST</td>
<td>$O(h)$</td>
<td>$O(h)$</td>
<td>$O(h)$</td>
<td>$O(h)$</td>
<td>$O(h)$</td>
<td>$O(h)$</td>
<td>$O(h)$</td>
</tr>
<tr>
<td>2-3-4 tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>
Red-black trees
Red-black tree (Guibas-Sedgewick, 1979)

Represent 2-3-4 tree as a binary search tree

• Use colors on nodes to represent 3- and 4-nodes.
Red-black tree (Guibas-Sedgewick, 1979)

Represent 2-3-4 tree as a binary search tree

• Use colors on nodes to represent 3- and 4-nodes.

• Connection between 2-3-4 trees and red-black trees:
Represent 2-3-4 tree as a binary search tree

- Use colors on nodes to represent 3- and 4-nodes.

- Connection between 2-3-4 trees and red-black trees:
Red-black tree

Properties of red-black trees:

- The root is always black
- All root-to-leaf paths have the same number of black nodes.
- Red nodes do not have red children
Red-black tree

Connection between 2-3-4 trees and red-black trees:
Red-black tree

Connection between 2-3-4 trees and red-black trees:
Red-black tree

Connection between 2-3-4 trees and red-black trees:
Insertion in red-black trees

Insertion in 2-node:

Insertion in 3-node:
Red-black tree: Splitting 4-nodes

Two easy cases:

\[
\begin{align*}
&\begin{array}{c}
\text{B} \\
\text{DIM}
\end{array} & \approx & \begin{array}{c}
\text{B} \\
\text{I} \\
\text{D} \\
\text{M}
\end{array} & \rightarrow & \begin{array}{c}
\text{B} \\
\text{I} \\
\text{D} \\
\text{M}
\end{array} & \approx & \begin{array}{c}
\text{B} \\
\text{I} \\
\text{D} \\
\text{M}
\end{array} \\
&\begin{array}{c}
\text{BD} \\
\text{EIM}
\end{array} & \approx & \begin{array}{c}
\text{B} \\
\text{I} \\
\text{D} \\
\text{M}
\end{array} & \rightarrow & \begin{array}{c}
\text{B} \\
\text{I} \\
\text{D} \\
\text{M}
\end{array} & \approx & \begin{array}{c}
\text{B} \\
\text{I} \\
\text{D} \\
\text{M}
\end{array}
\end{align*}
\]
Red-black trees: Splitting of 4-nodes

Example of hard case:

\[
\begin{array}{c}
\text{BL} \\
\text{DHK}
\end{array}
\approx
\begin{array}{c}
\text{B} \\
\text{L}
\end{array}
\rightarrow
\begin{array}{c}
\text{B} \\
\text{H}
\end{array}
\approx
\begin{array}{c}
\text{B} \\
\text{H}
\end{array}
\approx
??
\]

Solution: Rotations!
Rotations in red-black trees

Two types of rotations
Rotations in red-black trees

Two types of rotations:

1. **Right Rotation** (R):
   - A → B
   - B → C

2. **Left Rotation** (L):
   - C → B
   - B → A
Insertion in red-black trees

Insertion in 3-node:

Insert $H$: $\approx \quad \approx \quad \approx \quad ??$

Insert $E$: $\approx \quad \approx \quad \approx \quad ??$
Insertion in red-black trees

Insertion in 3-node:

\[
C F \quad \approx \quad \begin{array}{c}
\quad C \\
\quad \quad F
\end{array} \quad \rightarrow \quad \begin{array}{c}
\quad C \\
\quad \quad F \\
\quad \quad \quad H
\end{array} \quad \rightarrow \quad \begin{array}{c}
\quad F \\
\quad \quad C \\
\quad \quad \quad H
\end{array} \quad \approx \quad \begin{array}{c}
\quad C \\
\quad \quad F \\
\quad \quad \quad H
\end{array}
\]

Insert H

\[
C F \quad \approx \quad \begin{array}{c}
\quad C \\
\quad \quad F
\end{array} \quad \rightarrow \quad \begin{array}{c}
\quad C \\
\quad \quad F \\
\quad \quad \quad E
\end{array} \quad \rightarrow \quad \begin{array}{c}
\quad F \\
\quad \quad E
\end{array} \quad \approx \quad ??
\]

Insert E
Insertion in red-black trees

Insertion in 3-node:

1. Insertion of H:
   - Original: CF
   - Insert H: C F H

2. Insertion of E:
   - Original: CF
   - Insert E: C E F
Insertion in red-black tree

Insert x:
   Search to bottom after key (x)
   Insert red leaf
   Balance: 3 cases (+ symmetric)

Keep balancing with z
Eksempel

1. Insert U
2. Insert V
3. Rotate U
4. Rotate U
Example

Insert G

Rotate I

Rotate I

Rotate I
Running times in red-black trees

• Time for insertion:
  • Search to bottom after key: \(O(h)\)
  • Insert red leaf: \(O(1)\)
  • Perform recoloring and rotations on way up: \(O(h)\)
    • Can recolor many times (but at most \(h\))
    • At most 2 rotations.
  • Total \(O(h)\).

• Time for search
  • Same as BST: \(O(h)\)

• Height: \(O(\log n)\)
## Dynamic set implementations

### Worst case running times

<table>
<thead>
<tr>
<th>Implementation</th>
<th>search</th>
<th>insert</th>
<th>delete</th>
<th>minimum</th>
<th>maximum</th>
<th>successor</th>
<th>predecessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>linked lists</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>ordered array</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>BST</td>
<td>O(h)</td>
<td>O(h)</td>
<td>O(h)</td>
<td>O(h)</td>
<td>O(h)</td>
<td>O(h)</td>
<td>O(h)</td>
</tr>
<tr>
<td>2-3-4 tree</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>red-black tree</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>
Balanced trees: implementations

Redblack trees:

Java: `java.util.TreeMap, java.util.TreeSet`.

C++ STL: `map, multimap, multiset`.

Linux kernel: `linux/rbtree.h`.