Inge Li Gørtz

Welcome

• Inge Li Gørtz.
• Reverse teaching and discussion of exercises:
  • 3 teaching assistants
  • 8.00-9.15 Group work
  • 9.15-9.45 Discussions of your solutions in class
  • 10.00-11.15 Lecture
  • 11.15-11.45 Work on exercises in the new material
  • 11.45-12.00 Round up
• Weekly assignments (You have to get 40 points + pass 2 implementation exercises in order to be able to attend the written exam).
• Prerequisites: 02105/02326 Algorithms and Data Structures I

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: 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>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</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. 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

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

Ex. Insert B

Ex. Insert X
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

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

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

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

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
Local transformations that work anywhere in the tree.

Ex. Splitting a 4-node attached to a 2-node

Insertion 2-3-4 trees

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

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

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

Invariant. Current node is not a 4-node.
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

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:
  - or
- 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
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
- 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.

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(1))</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(1))</td>
<td>(O(1))</td>
<td>(O(\log n))</td>
<td>(O(\log n))</td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>(O(n))</td>
<td>(O(1))</td>
<td>(O(1))</td>
<td>(O(1))</td>
<td>(O(1))</td>
<td>(O(1))</td>
<td>(O(1))</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:

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
- CLRS: All leaves (NIL) are black
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:

$F \approx F \approx CF$

Insert C

Insertion in 3-node:

$CF \approx F \approx CFH$

Insert H
Red-black tree: Splitting 4-nodes

Two easy cases:

\[
\begin{align*}
&\text{Example of hard case:} \\
&\text{Solution: Rotations!}
\end{align*}
\]
Rotations in red-black trees

Two types of rotations:

Insertion in red-black trees

Insertion in 3-node:

Insert H

Insert E

Insertion in red-black trees

Insertion in 3-node:
**Insertion in red-black tree**

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

Keep balancing with z

**Example**

- Insert G
- 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 list</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>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>BST</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</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

Red-black trees:
- Java: java.util.TreeMap, java.util.TreeSet.
- C++ STL: map, multimap, multiset.
- Linux kernel: linux/rbtree.h.

Exercise time

Start working on the exercises on weekplan BST.
11.45 talk about solutions to the first 2-3 exercises + info about the mandatory exercises.

Mandatory exercise

- To be handed in before 8PM the following Sunday on DTU Learn.
- This week: Binary tries from weekplan warmup.
Grading of mandatory assignments

You can get up to 20 points for an exercise:

- 5 points for the example (draw tree/run algorithm etc)
- 15 points for design an algorithm exercise:
  - 4 points for the description of your algorithm
  - 4 points for the quality of your algorithm
  - 4 points for the complexity analysis
  - 3 points for the correctness arguments