



A safe component with strong guarantees

AVL tree

Bo Simonsen
bo@geekworld.dk

Department of Computing
The University of Copenhagen

June 24, 2008



Introduction

The AVL tree

What I accomplished

First refactoring

Algorithmic improvements

CPHSTL requirements

Associated data structure

New problems

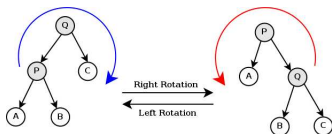
Exception safety

Bugs

Future work

The AVL tree

- Proposed by **A**delson-**V**elsky and **L**andis in 1962.
- Rebalancing is done on rotations and takes place in a bottom-up way.
- Rotations is performed on balance:
- $balance = H_{right} - H_{left}$.
- On insertion: If $balance \in [-1; 1]$, no rotations needed.
- Else rotations (single or double, single shown below):



- It's interesting since it maintains a height which is $\max 1.44 \lg n$, red-black trees has $2 \lg n$, and in CPHSTL we focus on constants.

What I accomplished

During this project I've made the following improvements to the component:

- Complexity ✓ (one problem left, without a solution)
- Strong exception safety ✓ (a few problems)
- Safe Iterators ✓

These properties was already there in the original code, but it's maintained during development:

- Linear Space ✓
- Iterator validity ✓



First refactoring

At the first sight at the original implementation:

- The component relied on BOOST.
- Which means it was implemented in boost namespace.
- No use of bridge classes (no iterators too, and a difference template interface).
- The implementation used property maps, boost proxy pattern for references.
- Methods missing `erase(v)` and `insert(I,I)`.
- **It didn't compile, even with all boost components installed.**

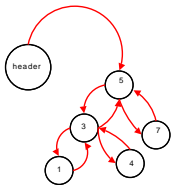
So the code we handed in is a complete rewrite of the interface.

CPHSTL requirements

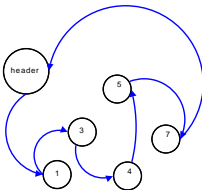
- Iterator operations ($++$, $--$) couldn't be verified.
- CPHSTL require iterator operations in $O(1)$, non-amortized.
- Finding successor/predecessor requires trivially $O(h)$ where $h = \lg n$ [Cormen].
- So we need a better solution.

The associated data structure

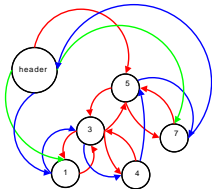
a) Normal tree structure



b) Associated structure



c) a+b, but with leftmost and rightmost



Exception safety

- copy constructor - just delete the tree an exception is thrown.
- assignment operator - keep the pointer to the old tree, start constructing the new tree. If something goes wrong use the old tree and destroy what's left of the new one. Else use the new and destroy the old.
- `insert(v)`, `insert(pos, v)` - all work is done in `_insert`.
Comparator fail: restore pred, succ pointers, Allocator fail: simply propagate the exception.
- `insert_sorted` - we first create a list with the nodes to be inserted, if an exception occurs, we will simply delete the list. If an error occurs while merging, delete the newly inserted elements.
- `erase(v)` - since we can have multiple elements, we simply find the elements to erase and store them in an array. And at last we traverse the array and delete them.

Bugs

Bugs found after the deadline (or close to the deadline, but no time for fixing it):

- **copy constructor + assignment operator:** Comparator and allocator should be set inside the try-catch block (since the copy constructor of the comparator/allocator can throw an exception), this was last minute changes, so that's why they are not there.
- **swap:** This method is not safe from the same reason as above. A solution could be to keep them as pointers, such that we should simply swap some pointers, which is an elementary operation which can't fail.
- The way of declaring arrays of dynamic size inside a method is not ansi compliant. This should be solved by allocating memory using the allocator. (g++ with -ansi will not work)
- Header should not have a data field (solution: inheritance).

Future work

A better multiset solution

