Scapegoat Trees
AAPP 2017 Project

Analysis, C implementation and testing, C++ benchmarks

Team members:
Crippa Mattia - 10397252
Vetere Alessandro - 10425802

Introduction

- Scapegoat trees are **self balancing binary search trees**
- No extra data in the tree nodes
- Scapegoat scheme:
  - Tree:
    - Root node pointer
    - Total size
    - Maximum total size since last completely rebuilt
  - Node:
    - Key value
    - Left and right child nodes pointers
- Rebalancing operation:
  - On INSERT
  - On DELETE

Fundamental properties

- α-weight-balanced:

  size(node.left) ≤ α*size(node) ∧ size(node.right) ≤ α*size(node)

- α-height-balanced, n = size(root):

  $$h(T) \leq h_\alpha(n) = \lfloor \log_{1/\alpha} n \rfloor$$

- If T is an α-weight-balanced binary search tree, then T is α-height-balanced

- Scapegoat trees are **loosely** α-height-balanced, n = size(root):

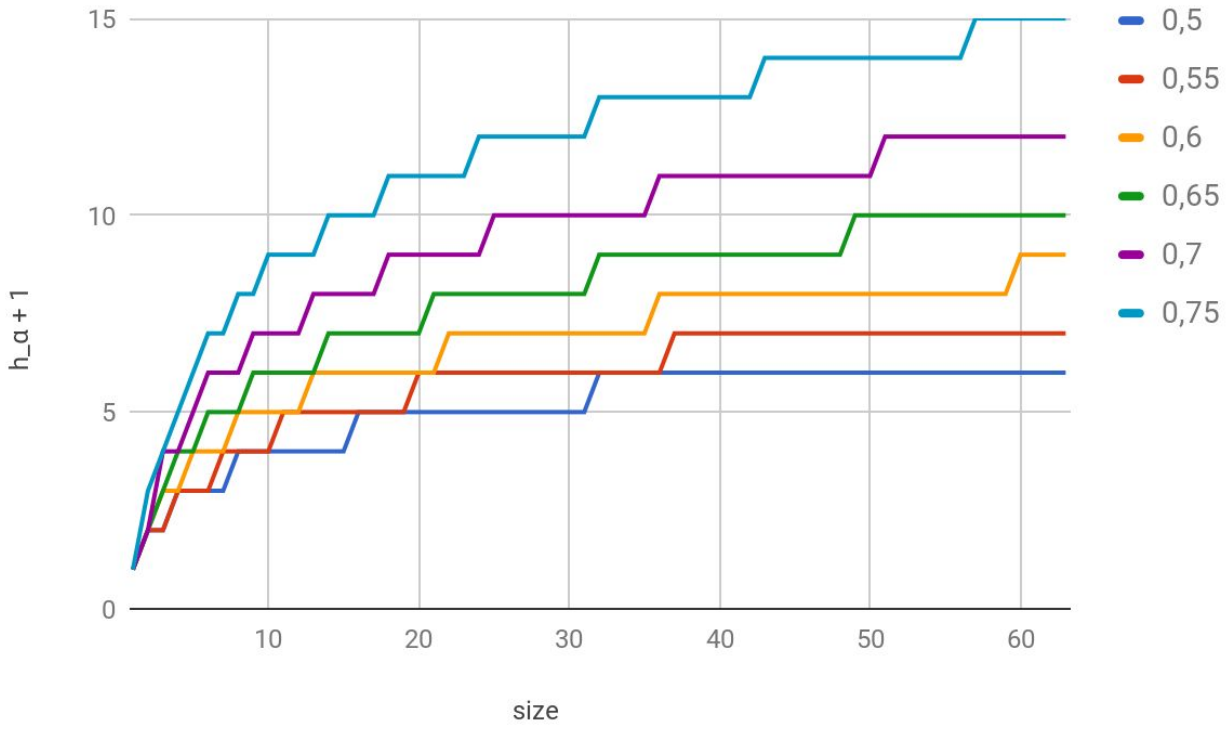
  $$h(T) \leq h_\alpha(n) + 1$$

- Fixed α, a node of depth greater than $h_\alpha(n)$ is a **deep** node

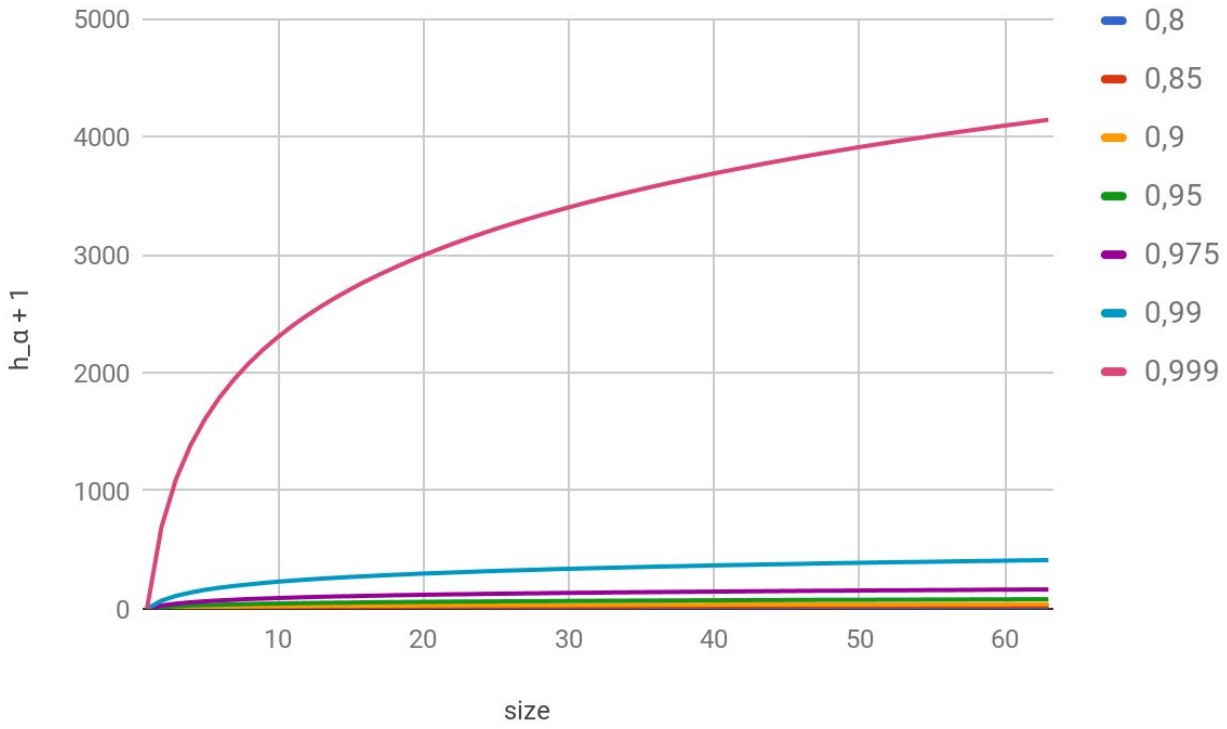
Meaning of α

- Scapegoat node is an α-weight-unbalanced node used to perform a **rebalance** operation
- Subtree rooted at the scapegoat node becomes an ½-weight-balanced tree
- Value of α should be in [0.5, 1):
  - High α → Few rebalance operations, quick INSERT, but slow SEARCH and DELETE
  - Low α → Many rebalance operations, quick SEARCH and DELETE, but slow INSERT
- Choose α depending on expected frequency of actions

Maximum tree depth given α



What happens as α approaches 1



SEARCH

- SEARCH proceeds as in a standard binary search tree

- No restructuring is performed

- Due to loosely α-height-balanced property, SEARCH has a worst case time of $O(h_\alpha(n))$ = **O(log n)**

INSERT

- INSERT a node as into an ordinary binary search tree, update size and max_size and record depth of new node
- If new node is deep, rebalance the tree:
  - Find scapegoat node → ancestor of the inserted node which isn't α-weight-balanced
  - Rebalance the subtree rooted at the scapegoat
- Rebalancing operation will restore the α-height-balanced (strict) property
- Worst case complexity of INSERT is O(size(scapegoat)), but it's amortized complexity for m operations is **O(log n)**

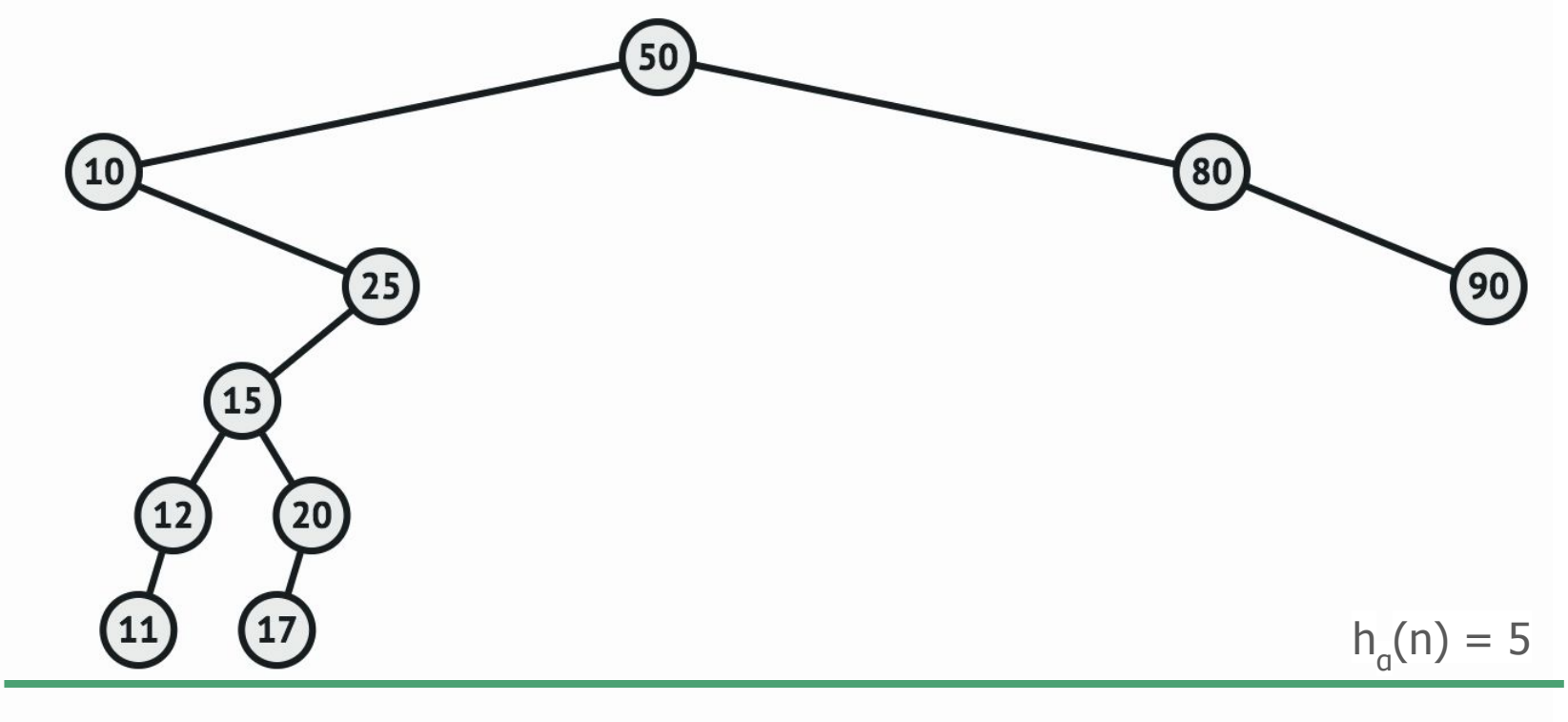
How to choose scapegoat node

- Climb the tree from the new node back up to the root and select the first node that isn't α-weight-balanced

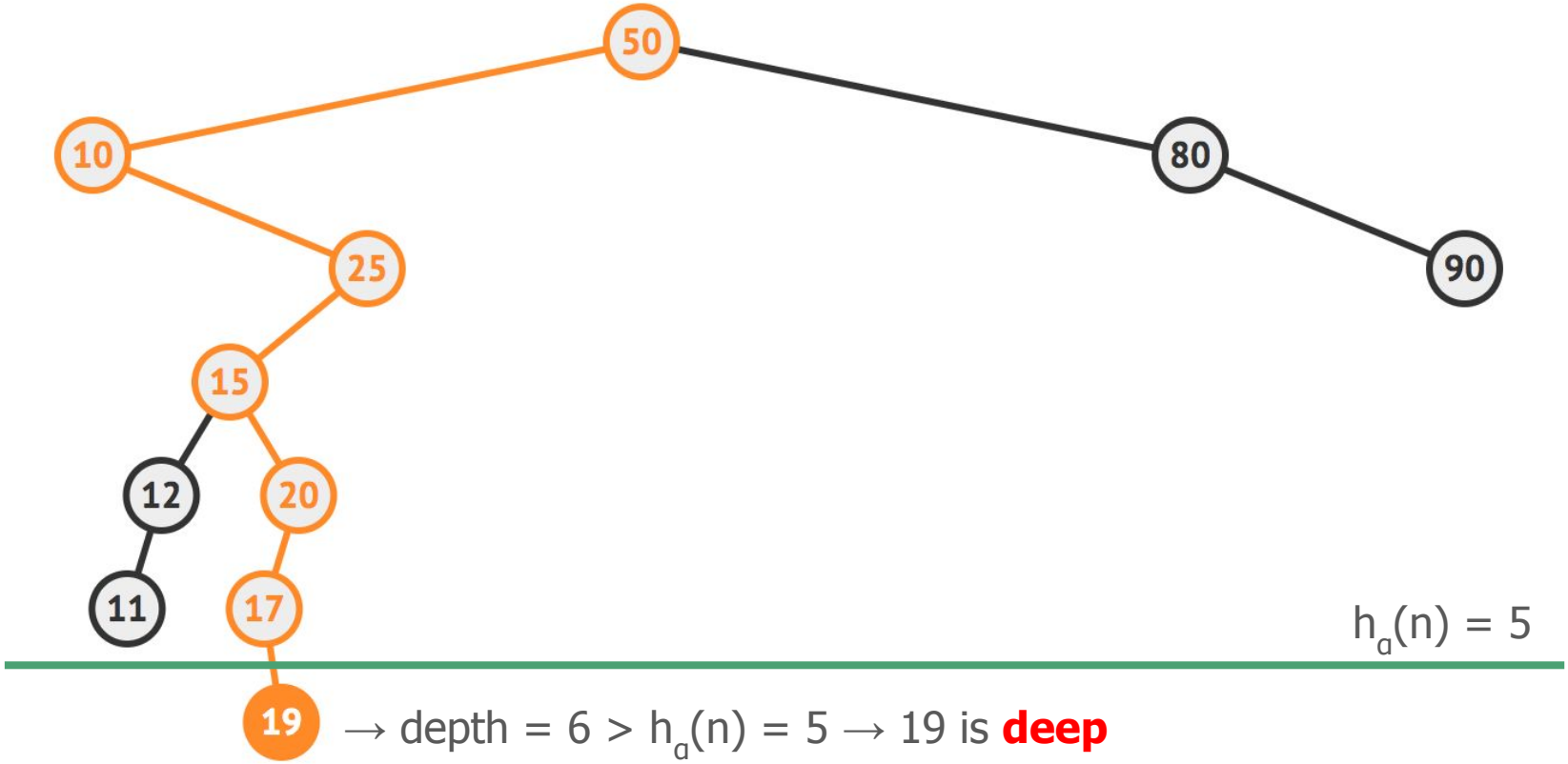
- A better heuristic is choose the first node x_i satisfying the following condition, being i the minimum distance of x_i from the new node:

$$i > h_\alpha(size(x_i))$$

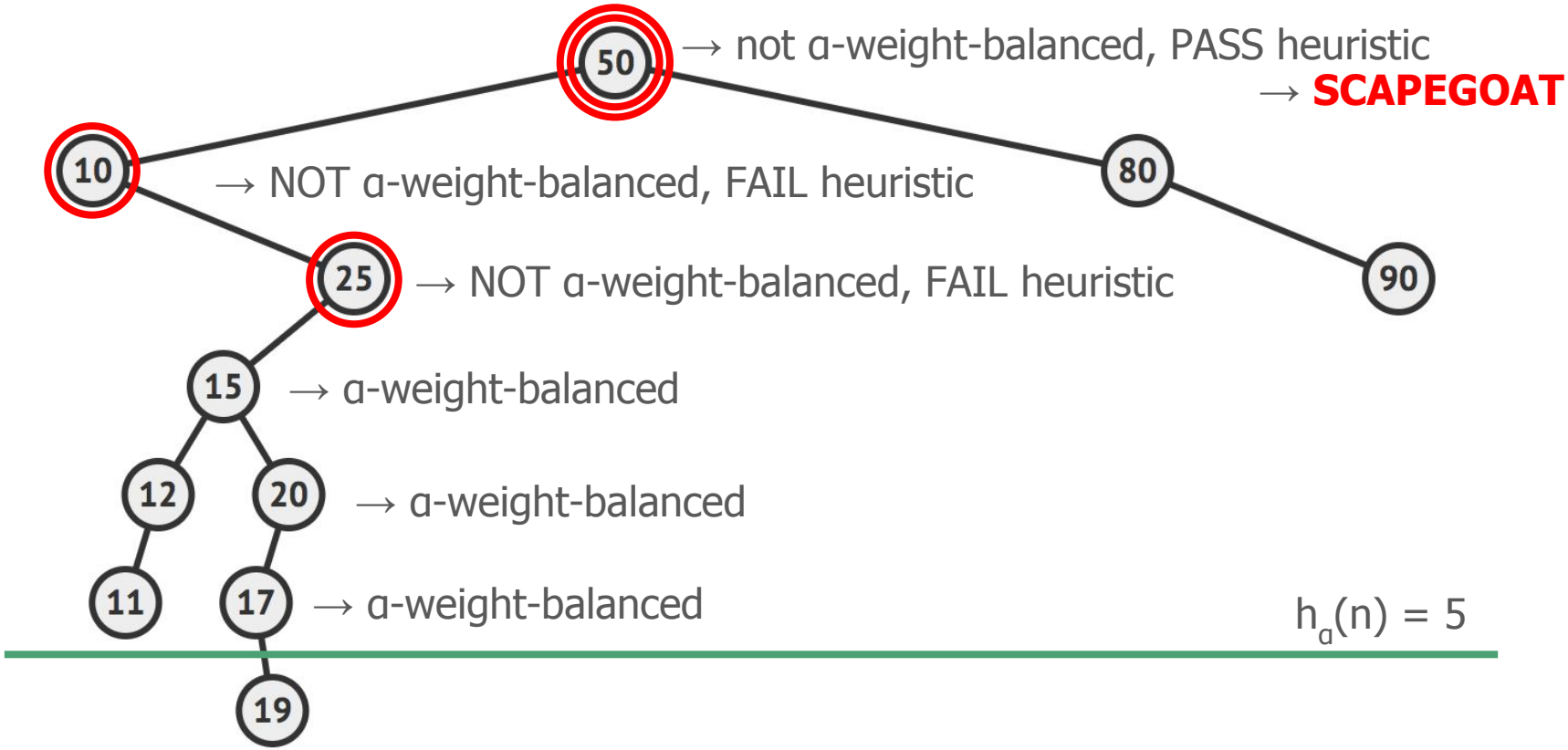
INSERT Example (1/5)



INSERT Example (2/5)

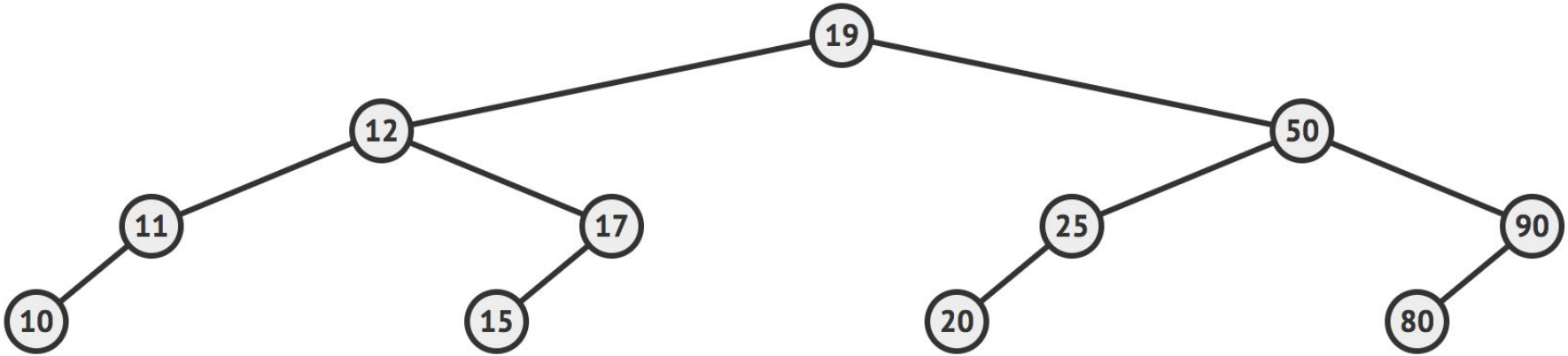


INSERT Example (3/5)



INSERT Example (4/5)

INSERT Example (5/5)



$h_a(n) = 5$

DELETE

- DELETE proceeds as in a standard binary search tree
- Use an additional value stored in the tree data structure
  - max_size which represents the highest achieved size since the tree was last rebuilt
- Rebalancing condition:

$$size < \alpha * max\_size$$

- Worst case complexity of DELETE is O(n), but it's amortized complexity for m operations is **O(log n)**

Pros

- SEARCH has worst case time complexity **O(log n)**
- INSERT, DELETE have an amortized time cost of **O(log n)**
- Scapegoat trees don't require any additional per-node information other than the key, the pointer to right child and the one to left child

Cons

- They do require two extra values per tree to see if a rebalance upon INSERT (size) or DELETE (size and max_size) is required
- α needs to be tuned for the expected frequency of the operations

Implementation in C

- Fast, robust and concise code
- Rich debug output at runtime if compiled using flag -DDEBUG
- Multiplatform, tested on Windows 10 64 bit and macOS Sierra
- Uses few external libraries:
  - stdio.h → printf
  - math.h → log, floor, ceil
  - stdlib.h → malloc, free
- Almost pedantic w.r.t. implementation described in the paper

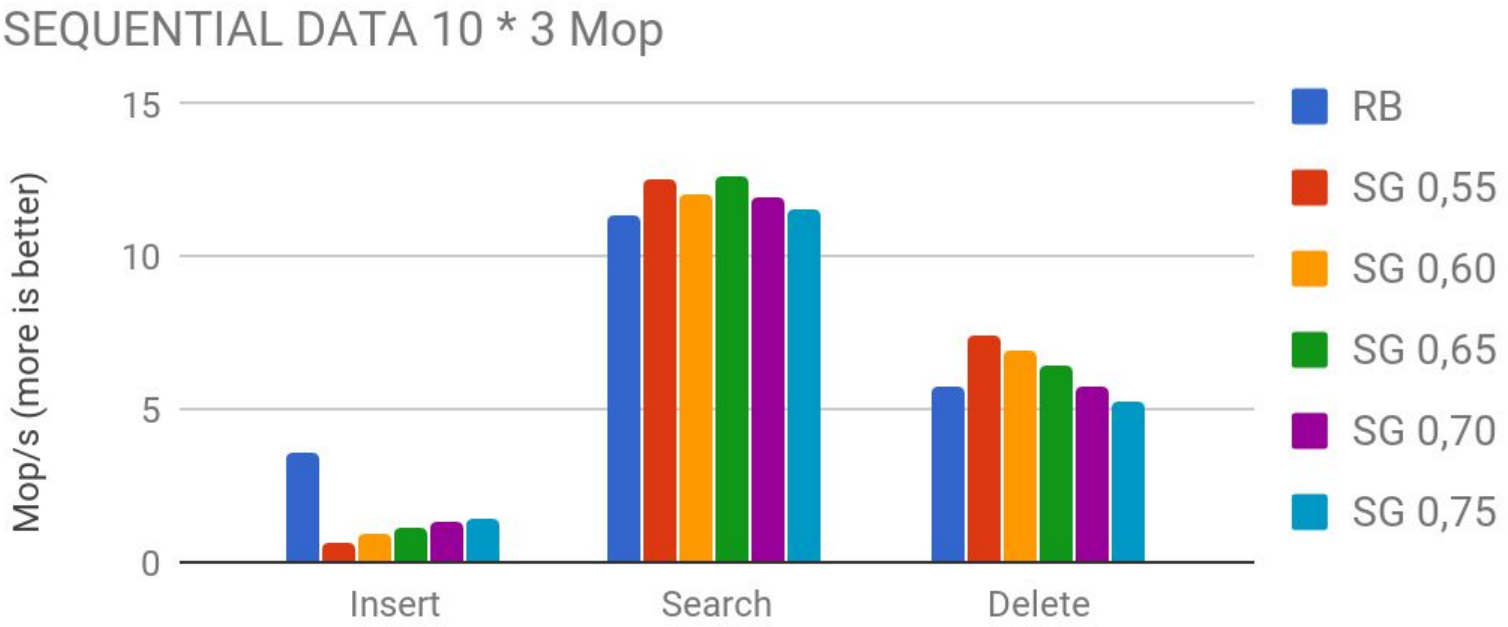
C programmatic API

```c
typedef struct sg_node {
      int key;
      struct sg_node* left;
      struct sg_node* right;
} t_sg_node;

typedef struct sg_tree {
      t_sg_node* root;
      unsigned int size;
      unsigned int max_size;
      double alpha;
      double log_one_over_alpha;
      unsigned int h_alpha;
} t_sg_tree;
```

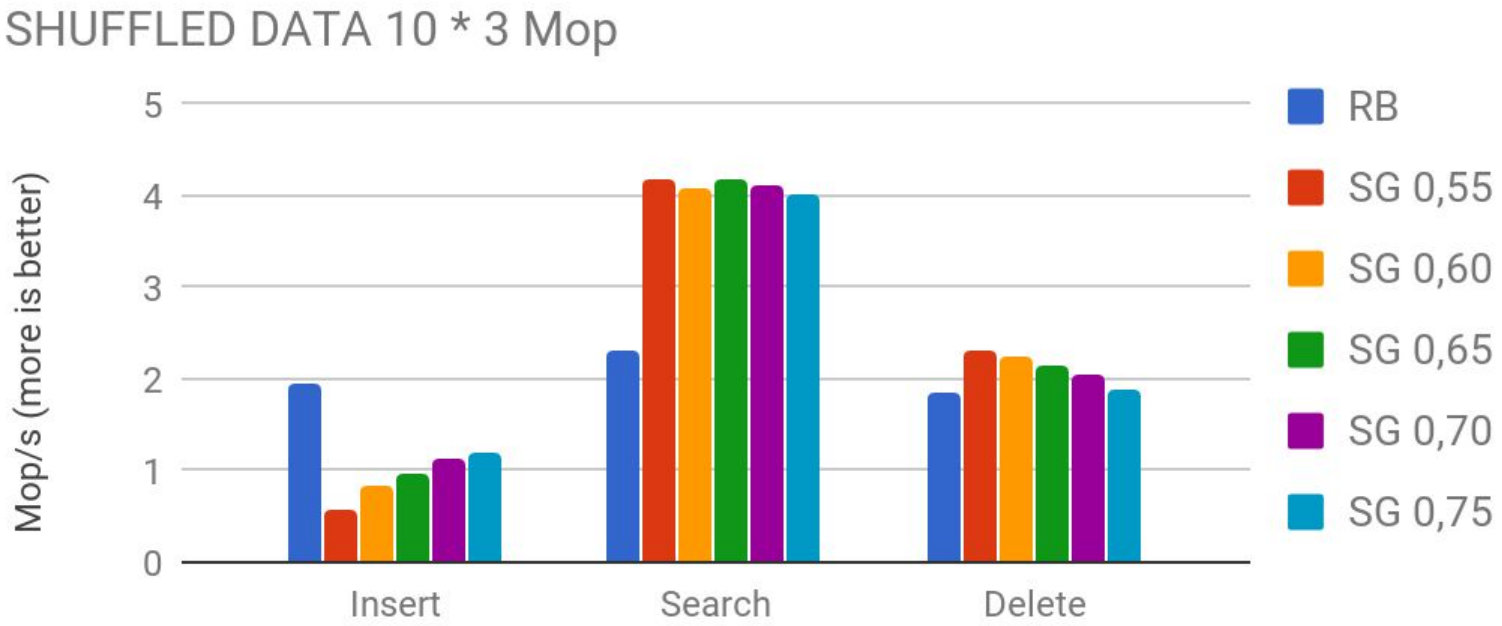
```c
t_sg_tree* sg_create_tree(double alpha);

void sg_delete_tree(t_sg_tree* tree);

t_sg_node* sg_search(t_sg_tree* tree, int key);

unsigned char sg_delete(t_sg_tree* tree, int key);

unsigned char sg_insert(t_sg_tree* tree, int key);

void sg_clear_tree(t_sg_tree* tree);
```

Benchmark in C++ (1/4)

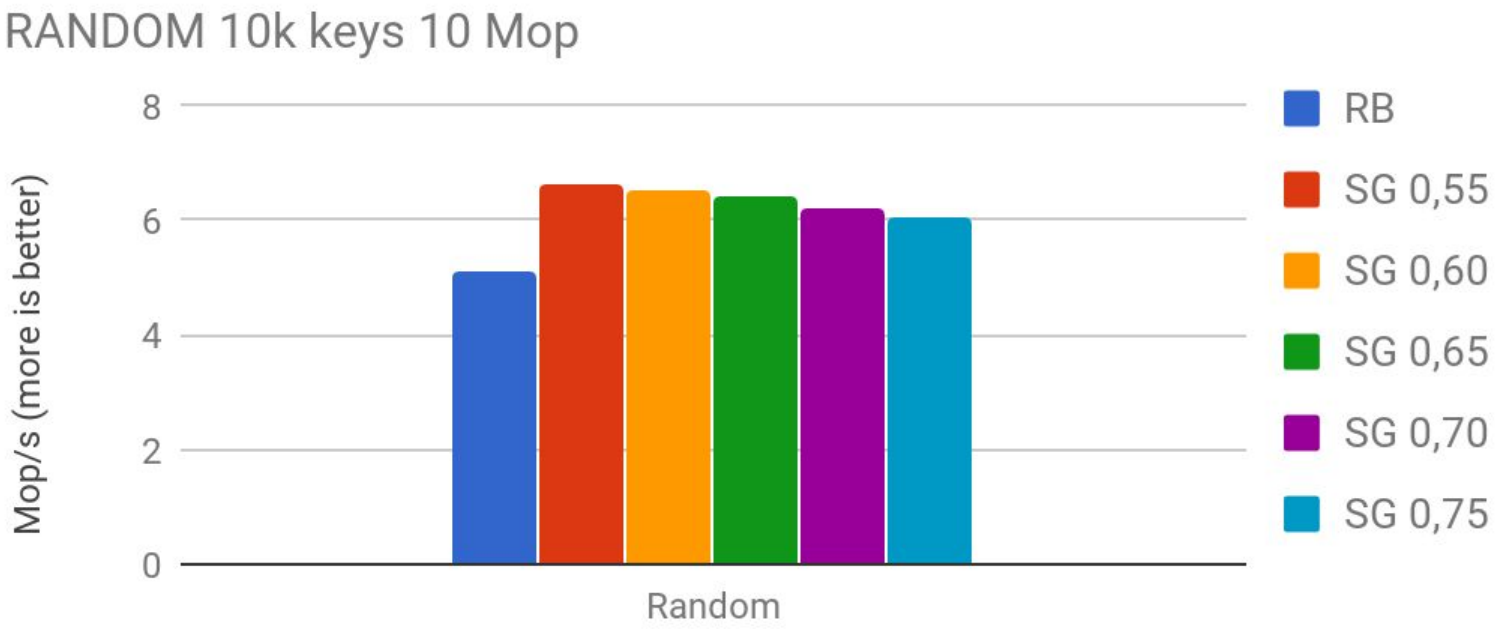
- Versus std::set<int> ➡ red-black tree



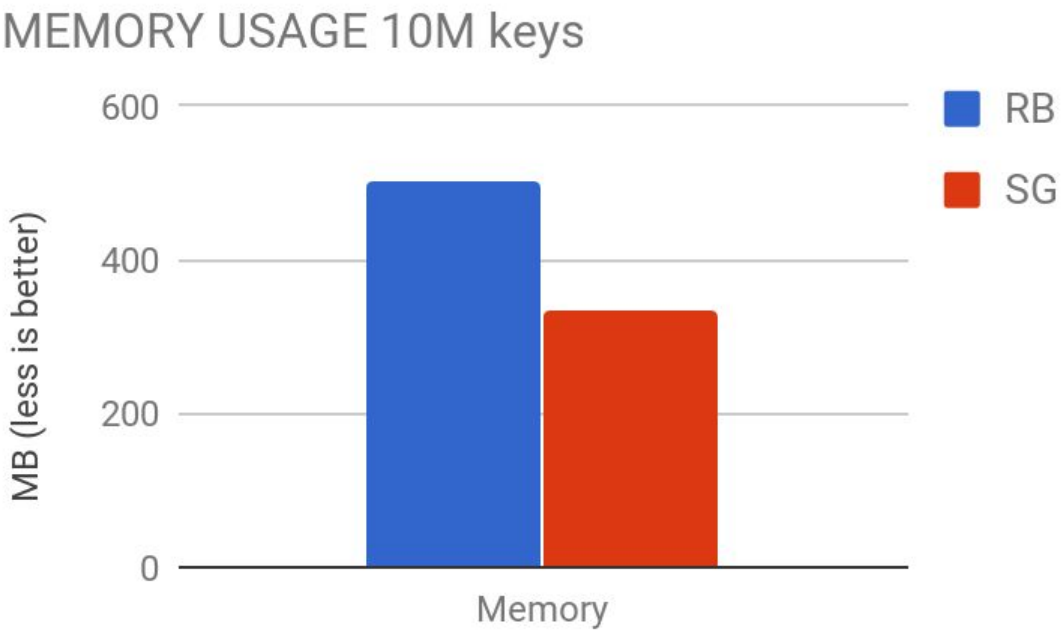
Benchmark in C++ (2/4)



Benchmark in C++ (3/4)



Benchmark in C++ (4/4)



Conclusions

- Sequential and shuffled INSERT faster on red-black tree, due to faster rebalancing operation
- Sequential SEARCH comparable, the two schemes yield balanced trees after sequential INSERTs
- Shuffled SEARCH much faster on scapegoat tree
- Sequential and shuffled DELETE comparable, a bit faster on scapegoat tree due to sporadic but complete rebalancing of the tree
- Random operations on random keys are faster on scapegoat tree, which demonstrates its usefulness as general purpose ordered data structure.
- Scapegoat tree is much more memory efficient than red-black