CSC 212: Data Structures and Abstractions

Balanced trees (part 2)

Prof. Marco Alvarez

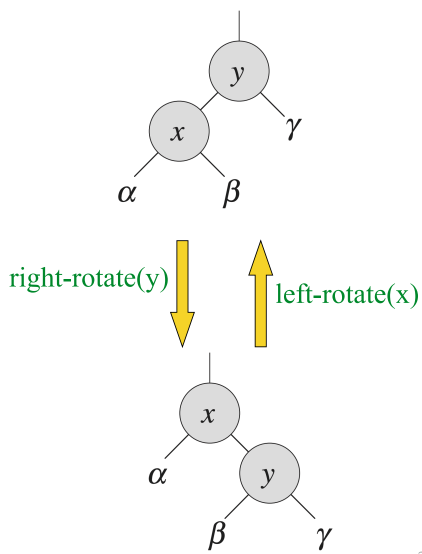
Department of Computer Science and Statistics
University of Rhode Island

Fall 2025

Rotations

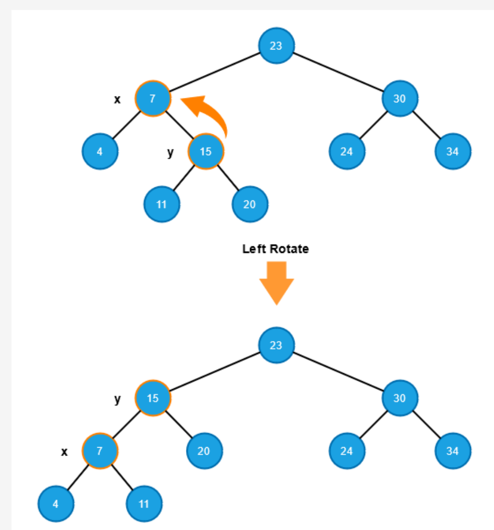
BST Rotations

- A **rotation** is a O(1)-time local operation that preserves the BST order property while changing the tree's structure
- **Right rotation** at node y
  - requires y's left child x to be *non-null*
  - elevates x to become the subtree root
  - y becomes x's right child
  - x's original right child becomes y's left child
- **Left rotation** at node x
  - requires x's right child y to be *non-null*
  - elevates y to become the subtree root
  - x becomes y's left child
  - y's original left child becomes x's right child



Example: left rotation

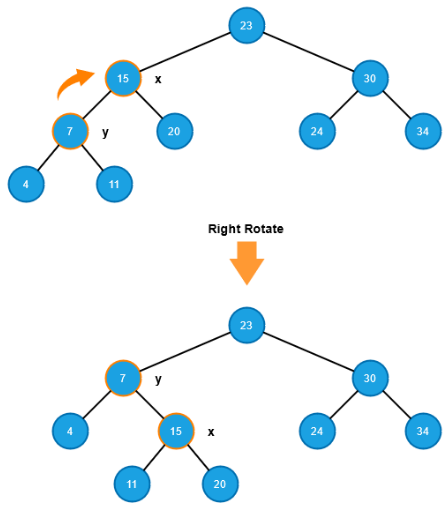
left-rotate(x)



https://www.formosa1544.com/2021/04/30/build-the-forest-in-python-series-red-black-tree/

Example: right rotation

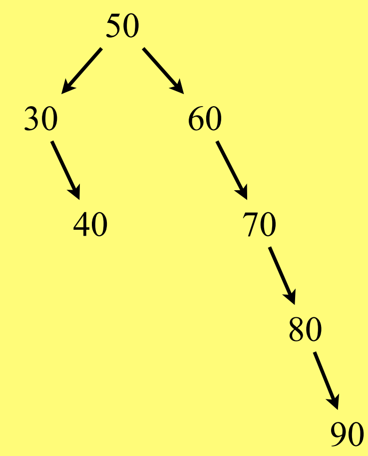
right-rotate(x)



https://www.formosa1544.com/2021/04/30/build-the-forest-in-python-series-red-black-tree/

Practice

- Perform the following operations in sequence
  - ✓ rotate-left(70)
  - ✓ rotate-left(50)
  - ✓ rotate-left(30)
  - ✓ rotate-right(50)



Red-black tree operations

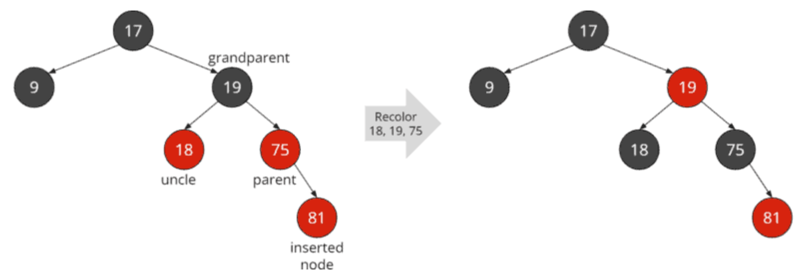
Insertion (overview)

- Steps
  - ✓ insert the new node following standard BST rules
  - ✓ color the new node **red** (to avoid violating the *root-to-null* rule)
  - ✓ if parent is **black**, terminate (forms 3-node or 4-node)
  - ✓ if parent is **red**, resolve the *red-red* violation using case-based rebalancing
  - ✓ finally, ensure the root is **black**
- Violation resolution
  - ✓ apply **recoloring** (preserves structure) and/or **rotations** (preserves order)
    - rotations are used when recoloring alone cannot restore properties

Insertion (detailed steps)

- Initial insertion
  - ✓ insert the new node following standard BST rules
  - ✓ color the new node **red**
  - ✓ apply the appropriate case resolution based on parent and uncle colors
- **Case 1**: parent is **black**
  - ✓ no *red-red* violation occurs
  - ✓ all properties satisfied, terminate
- **Case 2**: parent and uncle are both **red**
  - ✓ recolor parent and uncle to black
  - ✓ recolor grandparent to red
  - ✓ recursively apply case analysis to grandparent

Example: case 2

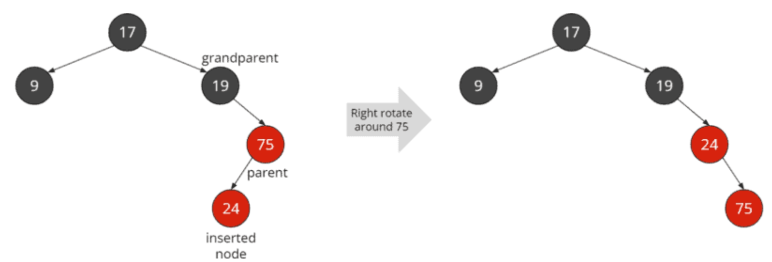


consider 3 other analogous (sub)cases

Insertion (detailed steps)

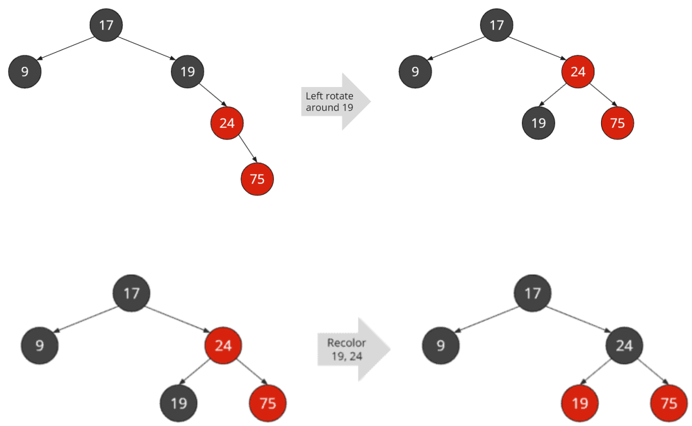
- **Case 3**: triangle formation — parent is **red**, uncle is **black** (or null)
  - ✓ rotate at parent to transform triangle into to line formation
    - left rotation at parent if new node is right child of left parent
    - right rotation at parent if new node is left child of right parent
  - ✓ proceed to case 4
- **Case 4**: line formation — parent is **red**, uncle is **black** (or null)
  - ✓ rotate at grandparent opposite to the line direction
    - right rotation at grandparent if line extends left
    - left rotation at grandparent if line extends right
  - ✓ swap colors of parent and grandparent
  - ✓ this case resolves the violation locally, no further propagation
- Final step
  - ✓ after all case resolutions, ensure the root node is colored **black**

Example: case 3 (triangle)



consider 1 other analogous (sub)case

Example: case 4 (line)



`consider 1 other analogous (sub)case`

https://www.happycoders.eu/algorithms/red-black-tree-java/

Practice

- Insert the following keys into a RB-tree
  - ✓ 10, 20, 30, 40, 50, 15, 25, 35, 45

Final remarks

- Theoretical Equivalence
  - ✓ due to the correspondence between red-black trees and 2-3-4 trees, the maximum height of a red-black tree with n nodes is $O(\log n)$, specifically at most $2\log_2(n+1)$
- Other operations
  - ✓ **search**: identical to standard BST search (colors are ignored, $O(\log n)$ time)
  - ✓ **delete**: follows BST deletion, then applies case-based rebalancing (not covered in lecture; more complex than insertion)
- C++ Implementation
  - ✓ STL containers `std::set` and `std::map` are typically implemented using red-black trees (though the standard doesn't mandate the specific data structure)

Analysis

Data Structure	Worst-case			Average-case			Ordered?
	insert at	delete	search	insert at	delete	search	
sequential (unordered)	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)	No
sequential (ordered) binary search	O(n)	O(n)	O(log n)	O(n)	O(n)	O(log n)	Yes
BST	O(n)	O(n)	O(n)	O(log n)	O(log n)	O(log n)	Yes
2-3-4	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	Yes
Red-Black	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	Yes

STL (ordered) containers

Ordered associative containers (STL)

Ordered associative containers implement sorted data structures that can be quickly searched – $O(\log n)$ complexity

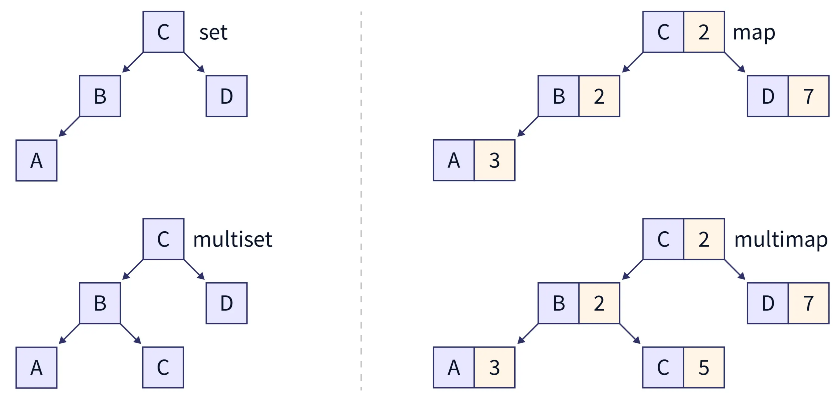


Image credit: https://www.scaler.com/topics/cpp/containers-in-cpp/

What is the output? — set

```
#include <iostream>
#include <set>

int main() {
    std::set<int> numbers;

    numbers.insert(5);
    numbers.insert(2);
    numbers.insert(8);
    numbers.insert(2);  // duplicate
    numbers.insert(5);  // duplicate
    numbers.insert(2);  // duplicate

    std::cout << "set elements: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    std::cout << "Count of 2 in set: " << numbers.count(2) << "\n";

    return 0;
}
```


What is the output? — multiset

```
#include <iostream>
#include <set>

int main() {
    std::multiset<int> numbers;

    numbers.insert(5);
    numbers.insert(2);
    numbers.insert(8);
    numbers.insert(2);  // duplicate
    numbers.insert(5);  // duplicate
    numbers.insert(2);  // duplicate

    std::cout << "multiset elements: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << "\n";

    std::cout << "Count of 2 in multiset: " << numbers.count(2) << "\n";

    return 0;
}
```

What is the output? — map

```cpp
#include <map>
#include <string>
#include <iostream>

int main() {
    std::map<std::string, int> freq_table;

    std::string words[] = {"apple", "banana", "apple", "cherry", "banana", "apple"};

    for (const auto& word : words) {
        freq_table[word]++;  // creates entry with value 0 if key doesn't exist
    }

    if (freq_table.find("date") == freq_table.end()) {
        std::cout << "'date' not found in map\n";
    }

    std::cout << "Frequency of 'apple': " << freq_table["apple"] << "\n";

    std::cout << "Word frequencies (sorted by key):\n";
    for (const auto& pair : freq_table) {
        std::cout << pair.first << ": " << pair.second << "   ";
    }
    std::cout << "\n";

    return 0;
}
```


What is the output? — multimap

```cpp
#include <map>
#include <string>
#include <iostream>

int main() {
    std::multimap<std::string, std::string> phone_book;

    // Insert multiple numbers for same person
    phone_book.insert({"Alice", "555-1234"});
    phone_book.insert({"Alice", "555-5678"});
    phone_book.insert({"Bob", "555-9999"});
    phone_book.insert({"Alice", "555-0000"});

    std::cout << "Alice has " << phone_book.count("Alice") << " numbers\n";

    std::cout << "Phone Directory:\n";
    for (const auto& entry : phone_book) {
        std::cout << " " << entry.first << ": " << entry.second << "\n";
    }

    std::cout << "Alice's numbers:\n";
    auto range = phone_book.equal_range("Alice");
    for (auto it = range.first ; it != range.second ; ++it) {
        std::cout << "  " << it->second << "\n";
    }

    return 0;
}
```