CSC 212: Data Structures and Abstractions

Balanced trees (part 2)

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2025

THINK BIG WE DO℠

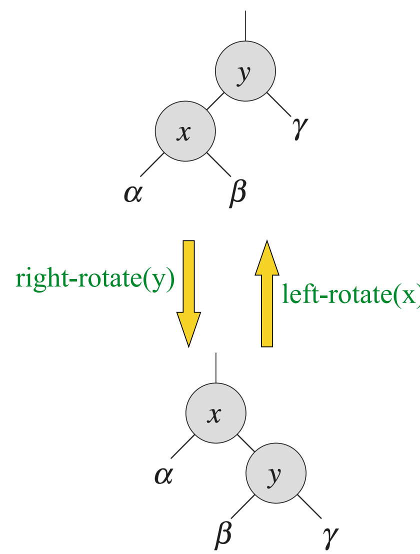
Red-black trees

Insertion

- Steps
  - ✓ insert the new node following standard BST rules
  - ✓ color the new node **red** (required to maintain 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 by propagating fixes upward
- Violation resolution
  - ✓ apply **recoloring** (preserves structure) and/or **rotations** (preserves order)
    - rotations are used when recoloring alone cannot restore properties

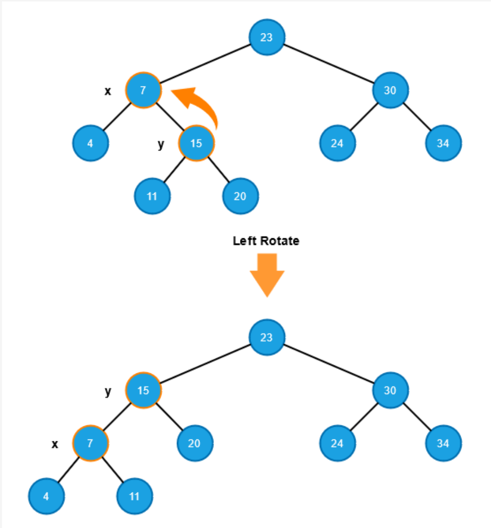
Rotations

- A **rotation** is a O(1)-time local operation that preserves the BST order property while modifying 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

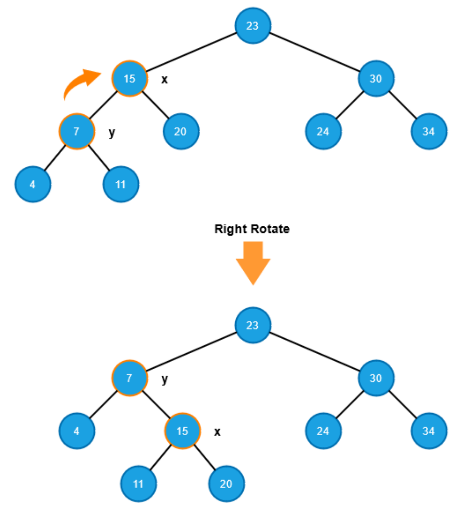
left-rotate(x)



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

Example: right rotation

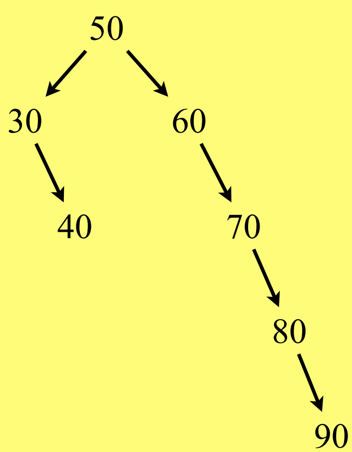
right-rotate(x)



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)

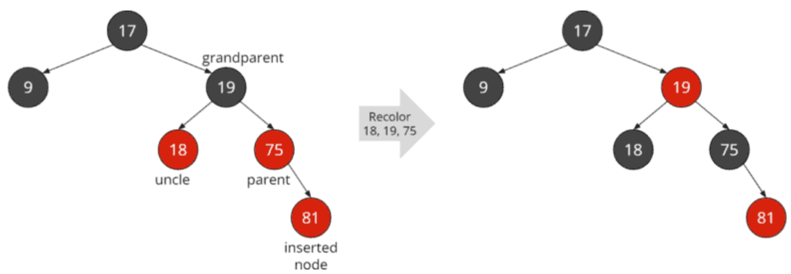


Insertion (detailed steps)

- Insert the new node following standard BST rules
  - ✓ color the new node **red**
  - ✓ apply the appropriate case resolution

- Case resolution
  - ✓ **Case 1**: parent of new node is **black**
    - no *red-red* violation occurs
    - the tree remains valid without modification
  - ✓ **Case 2**: parent and uncle of new node are **red**
    - recolor parent and uncle to black
    - recolor grandparent to red
    - recursively validate the grandparent as if newly inserted

Example: case 2

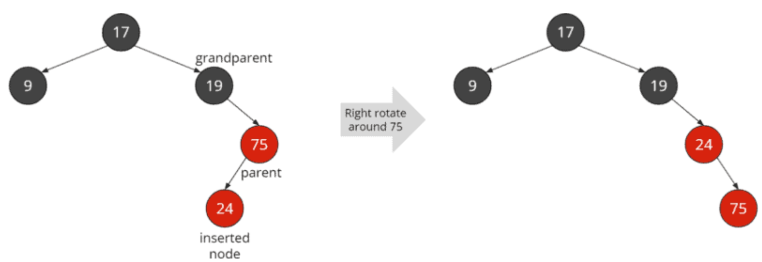


consider 3 other analogous (sub)cases

Insertion (detailed steps)

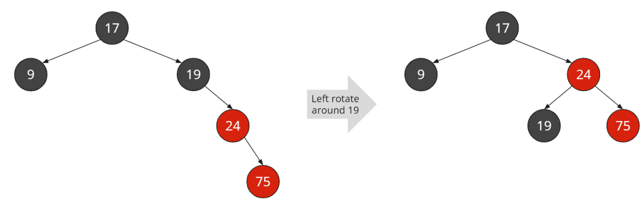
- Case resolution (cont.)
  - ✓ **Case 3**: triangle formation
    - condition: <u>parent</u> is **red**, <u>uncle</u> is **black** (or null)
    - action: apply rotation at parent to convert to line formation
      - if new node is right child of left parent, perform left rotation at parent
      - if new node is left child of right parent, perform right rotation at parent
    - proceed to case 4
  - ✓ **Case 4**: line formation
    - condition: <u>parent</u> is **red**, <u>uncle</u> is **black** (or null)
    - action: rotate at grandparent and then swap colors of original parent and grandparent
      - right rotation if line extends left
      - left rotation if line extends right
    - this case terminates the rebalancing process
- Final step
  - ✓ after all case resolutions, ensure the root node is colored **black**

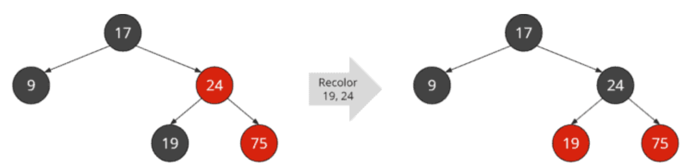
Example: case 3 (triangle)



consider 1 other analogous (sub)case

Example: case 4 (line)



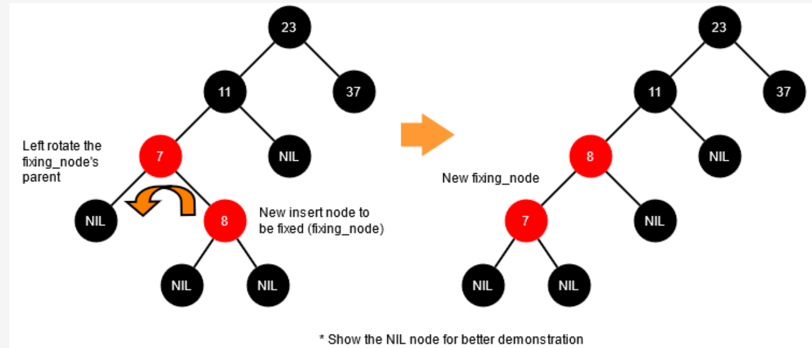


consider 1 other analogous (sub)case

Practice

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

Example: case 2



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

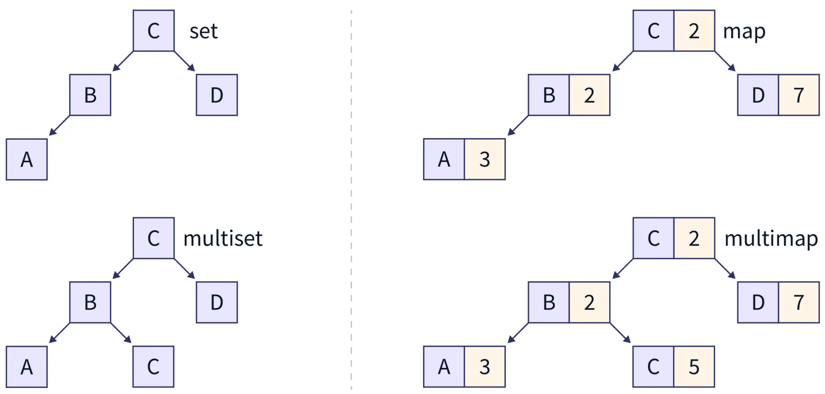
Final remarks

- Theoretical Equivalence
  - ✓ due to the isometry between red-black trees and 2-3-4 trees, the maximum height of a red-black tree with n nodes: $2\log_2(n+1)$
- Other operations
  - ✓ **search**: identical to binary search trees
  - ✓ **delete**: similar to insert (not covered in lecture, standard libraries provide optimized implementations)
- C++ Implementation
  - ✓ STL containers `std::map` and `std::set` typically use red-black trees

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

Ordered associative containers (STL)



std::set

Defined in header <set>

```
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;
```
(1)

```
namespace pmr {
    template<
        class Key,
        class Compare = std::less<Key>
    > using set = std::set<Key, Compare, std::pmr::polymorphic_allocator<Key>>;
}
```
(2) (since C++17)

`std::set` is an associative container that contains a sorted set of unique objects of type Key. Sorting is done using the key comparison function *Compare*. Search, removal, and insertion operations have logarithmic complexity. Sets are usually implemented as Red-black trees.

Everywhere the standard library uses the *Compare* requirements, uniqueness is determined by using the equivalence relation. In imprecise terms, two objects `a` and `b` are considered equivalent if neither compares less than the other: `!comp(a, b) && !comp(b, a)`.

`std::set` meets the requirements of *Container*, *AllocatorAwareContainer*, *AssociativeContainer* and *ReversibleContainer*.

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

int main() {
    std::set<int> example{1, 2, 3, 4};

    for (int x : {2, 5}) {
        if (example.contains(x))
            std::cout << x << ": Found\n";
        else
            std::cout << x << ": Not found\n";
    }
}
```


std::map

Defined in header <map>

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T>>
> class map;
```
(1)

```
namespace pmr {
    template<
        class Key,
        class T,
        class Compare = std::less<Key>
    > using map = std::map<Key, T, Compare,
                           std::pmr::polymorphic_allocator<std::pair<const Key, T>>>;
}
```
(2) (since C++17)

`std::map` is a sorted associative container that contains key-value pairs with unique keys. Keys are sorted by using the comparison function `Compare`. Search, removal, and insertion operations have logarithmic complexity. Maps are usually implemented as Red-black trees.

Iterators of `std::map` iterate in ascending order of keys, where ascending is defined by the comparison that was used for construction. That is, given

- `m`, a `std::map`
- `it_l` and `it_r`, dereferenceable iterators to `m`, with `it_l < it_r`.

`m.value_comp()(*it_l, *it_r) == true` (least to greatest if using the default comparison).

Everywhere the standard library uses the *Compare* requirements, uniqueness is determined by using the equivalence relation. In imprecise terms, two objects `a` and `b` are considered equivalent (not unique) if neither compares less than the other: `!comp(a, b) && !comp(b, a)`.

`std::map` meets the requirements of *Container*, *AllocatorAwareContainer*, *AssociativeContainer* and *ReversibleContainer*.