CSC 212: Data Structures and Abstractions

Balanced trees

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2025

THINK BIG WE DO℠

From lab session

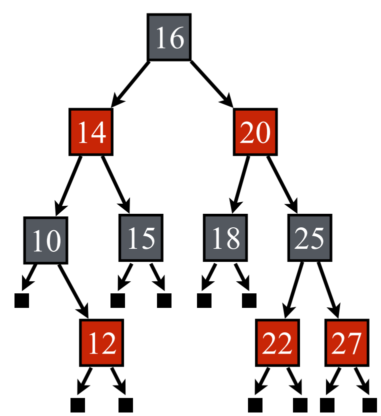
- Assume the dictionary has n keys, and the book has m words (tokens)
  - ✓ what is the computational cost of finding all words in the book that are not in the dictionary?
    - dictionary is represented as a BST and assume that $h = O(\log n)$

Balanced search trees

- **Balanced search trees** are a type of search trees that maintain a balanced structure to ensure that the height of the tree is logarithmic in the number of nodes
  - ✓ among the most useful data structures in computer science
  - ✓ many programming languages have built-in support: e.g. Java's `TreeSet` and `TreeMap`, C++'s `std::set` and `std::map`
- Examples of balanced trees:
  - ✓ AVL trees, **Red-Black trees**, B-trees, Splay trees, Treaps, etc.

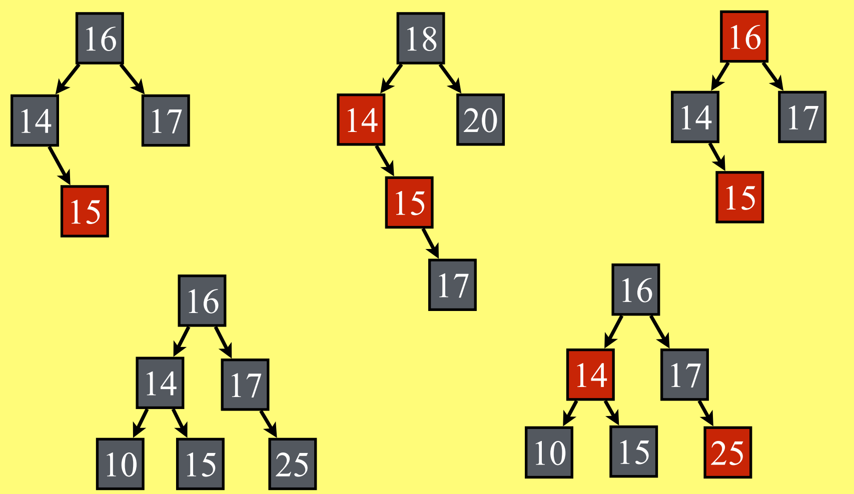
Red-black trees

- Red-black trees are BSTs that maintain a balanced structure by enforcing the following properties on the nodes:
  - ✓ each node is colored either **red** or **black**
  - ✓ the root node is always **black**
  - ✓ **red** nodes cannot have **red** children (no two red nodes can be adjacent)
  - ✓ *null* nodes are considered **black**
  - ✓ every *root-to-null* path must have the same number of **black** nodes



Practice

- Are these valid red-black trees? — (null nodes not shown)



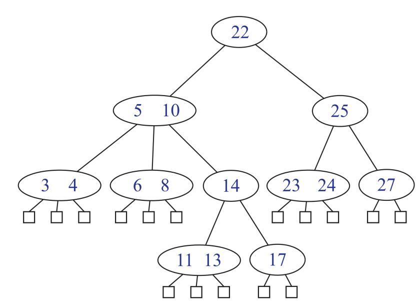
Analysis

- A red-black tree on n nodes has $h = O(\log n)$
  - ✓ after performing an insertion or deletion the tree may become unbalanced
  - ✓ to restore balance, we efficiently modify the tree to satisfy the red-black tree properties
    - done by performing a sequence of **rotations** and **recoloring** nodes
- Equivalence to **B-trees**
  - ✓ red-black trees are equivalent to B-trees of order 4
  - ✓ it is easier to understand the complexity analysis and rebalancing operations of red-black trees by thinking of them as B-trees

B-Trees (interlude)

Multi-way search trees

- A multi-way search tree is a generalization of a BST that allows nodes to have more keys and more than two children
  - ✓ the keys in each node are **sorted** in increasing order
  - ✓ the keys in the left subtree of a key k are less than k, and the keys in the right subtree are greater than k

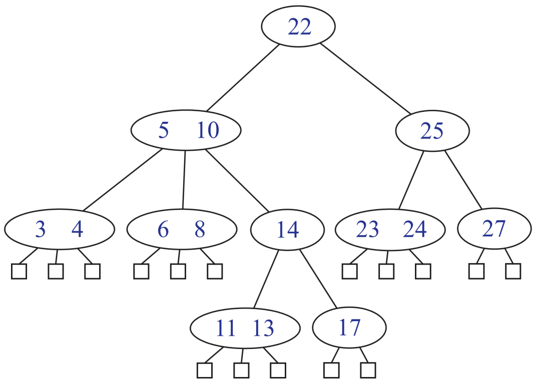


note that null pointers are illustrated as external nodes

Image credit: Data Structures and Algorithms in C++ 2e

Search on a multi-way search tree

- Perform **search** for 12, 17, 24, and 50 on the following tree
  - ✓ note that null pointers are illustrated as external nodes



Assume d denotes the maximum number of children of any node of T, and h denotes the height of T. What is the cost of search?

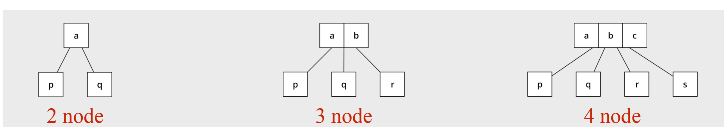
Image credit: Data Structures and Algorithms in C++ 2e

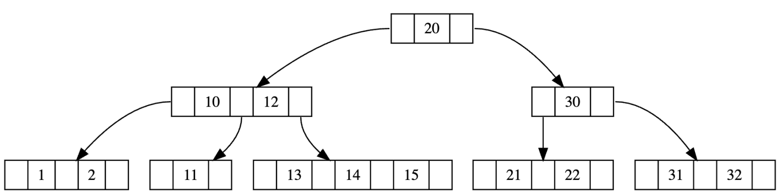
Balanced multi-way search trees

- Balanced multi-way search tree
  - ✓ cap the number of children to a fixed number and keep the leaf nodes at the same depth
  - ✓ the tree is **always balanced**
    - all leave nodes have the same depth
    - search, insertion, and deletion can be performed in $O(\log n)$ time
- **B-tree:** specific type of a balanced multi-way search tree
  - ✓ on a B-tree of order m, each node, except the root, must have between $\lceil b/2 \rceil$ and b children
    - note there are differences in terminology (multiple “order” definitions)
  - ✓ heavily used in databases and file systems to store large amounts of data (common orders: 1024, 2048, 4096, …)

2-3-4 tree

- A 2-3-4 tree (a.k.a. 2-4 tree) is a B-tree of order 4
  - ✓ each node can have 2, 3, or 4 children
  - ✓ i.e. all nodes must have at least 1 key and at most 3 keys, except the root node that can have 0 keys when the tree is empty

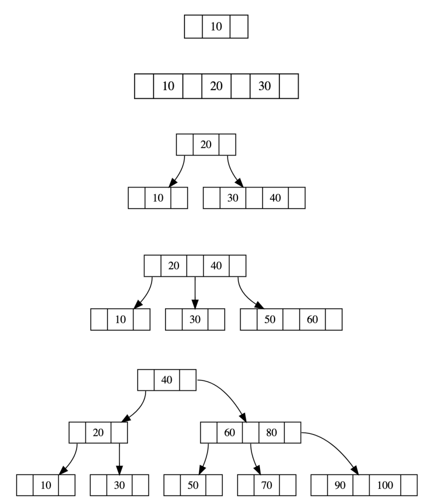




Insertion (2-3-4 tree)

- Steps
  - ✓ start at the root and traverse down the tree to find the appropriate leaf node
  - ✓ if the leaf node has less than 3 keys, insert the new key in sorted order
  - ✓ if the leaf node has 3 keys, split it into two nodes and promote the middle key to the parent node
    - insert the new key in the appropriate child node
    - if the parent node also has 3 keys, repeat the splitting process up to the root
- Tree remains balanced after each insertion
  - ✓ all leaf nodes are at the same level

Insert 10, 20, 30, 40, 50, 60, 70, 80, 90, 100



http://ysangkok.github.io/js-clrs-btree/btree.html

Practice

- Insert the following sequence into a 2-3-4 tree
  - ✓ 15, 10, 25, 5, 1, 30, 45, 60, 100, 70, 80, 40, 35, 90

Practice

- What is the max h of a 2-3-4 tree with n nodes?
  - ✓ to maximize the height, we want to minimize the number of keys per node (instance of a worst-case)
  - ✓ draw an example tree and express h in terms of n

Practice

- What is the cost of search and insert on a 2-3-4 tree?
  - ✓ worst-case scenario

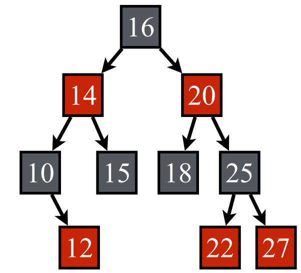
So far …

- The cost of operations in a B-tree of order b is $O(b \log_b n)$
  - ✓ insert, search, remove
  - ✓ small values of b make this cost optimal
- In practice …
  - ✓ B-trees are widely used in databases and file systems to manage large amounts of data efficiently
  - ✓ useful for systems that read and write large blocks of data
    - B-trees can minimize the number of disk accesses required (much larger order values)

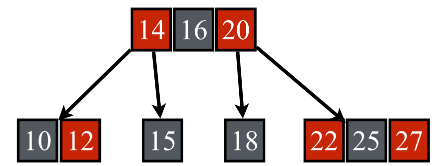
Red-black trees

Red-black trees <=> 2-3-4 trees

- Red-black trees are isometric to 2-3-4 trees
  - ✓ the number of black nodes between any *root-to-null* path in the red-black tree matches the depth of the corresponding 2-3-4 tree



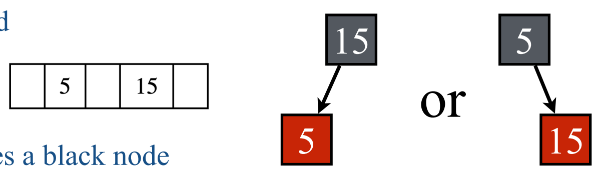
- Every red-black tree can be transformed into an equivalent 2-3-4 tree and vice versa



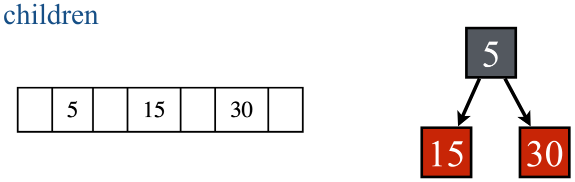
Red-black trees <=> 2-3-4 trees

- A 2-node in a 2-3-4 tree becomes a black node in a red-black tree

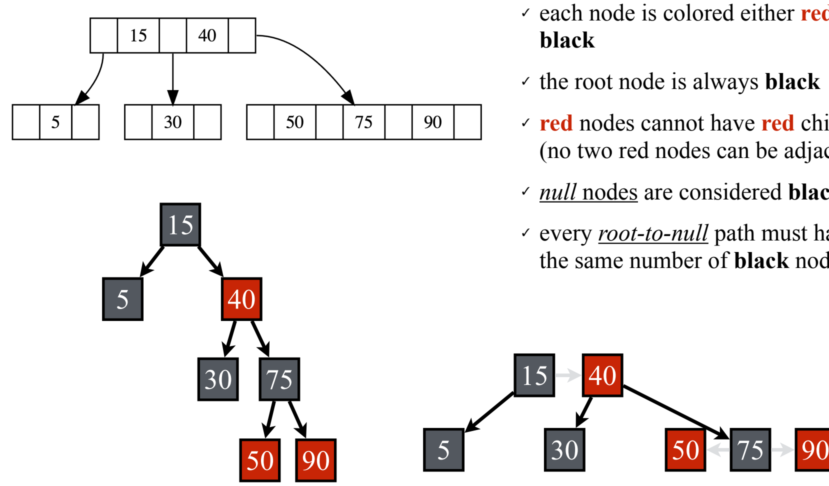
- A 3-node becomes a black node with one red child



- A 4-node becomes a black node with two red children



Example



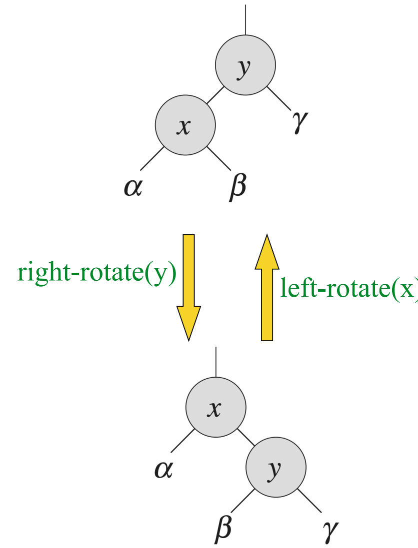
- ✓ each node is colored either **red** or **black**
- ✓ the root node is always **black**
- ✓ **red** nodes cannot have **red** children (no two red nodes can be adjacent)
- ✓ <u>*null* nodes</u> are considered **black**
- ✓ every <u>*root-to-null*</u> path must have the same number of **black** nodes

Insertion

- Steps
  - insert the new node as you would in a regular BST
  - color the new node **red**
    - (can't color it black otherwise it violates the "root-to-null" rule)
  - if the parent is **black**, you are done
    - (becomes 3-node or 4-node)
  - if the parent is **red**, you have a violation of the red-black tree properties
    - (need to fix the violation all the way up to the root)
- Fixing the violation
  - involves applying **recoloring** and/or **rotation** operations
  - recoloring is trivial and does not change the structure (BST order property)

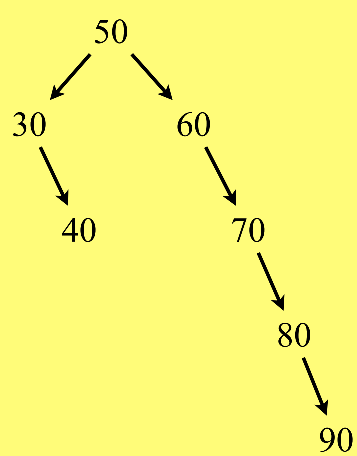
Rotations

- A rotation is a local operation that preserves the BST order property
  - two kinds: left and right rotation
  - can be done in O(1) time
- Left rotation at node x
  - assume that its right child y is not null
  - make y the new root of the subtree, with x as y's left child and y's left child as x's right child
- Right rotation at node y
  - assume that its left child x is not null
  - make x the new root of the subtree, with y as x's right child and x's right child as y's left child



Practice

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



Insertion (all cases)

- Insert the new node using the standard BST insertion
  - color the new node as **red**
  - apply one of the cases below
- **Case 1**: the parent of the new node is **black**
  - no violation occurs - the tree remains valid
- **Case 2**: the parent and uncle of the new node are both **red**
  - change the parent and uncle to black
  - change the grandparent to red
  - recursively check the grandparent as if it were newly inserted

Insertion (all cases)

- **Case 3**: the parent is **red**, the uncle is **black** (or null), and the tree forms a triangle
  - ✓ if the new node is a right child of its parent and the parent is a left child of the grandparent (or vice versa), perform a left rotation on the parent (or right rotation in the other case)
  - ✓ then apply Case 4
- **Case 4**: the parent is **red**, the uncle is **black** (or null), and the tree forms a line
  - ✓ if the new node is a left child of its parent and the parent is a left child of the grandparent (or both are right children), perform a right rotation on the grandparent (or left rotation in the other case), then swap the colors of the original parent and grandparent
  - ✓ case 4 is terminal (no further work is necessary)
- After applying all cases, the root of the tree is set to **black**