

Other Balanced Search Trees

Hash Tables

CS 311 Data Structures and Algorithms

Lecture Slides

Wednesday, April 30, 2008

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

CHAPPELLG@member.ams.org

© 2005–2008 Glenn G. Chappell

Unit Overview

Tables & Priority Queues

Major Topics

- ✓ • Introduction to Tables
- ✓ • Priority Queues
- ✓ • Heap algorithms
- ✓ • Heaps and Priority Queues in practice
- ✓ • 2-3 Trees
 - Other balanced search trees
 - Hash Tables
 - Prefix Trees
- ✓ • Tables in practice

Review

Introduction to Tables

	Sorted Array	Unsorted Array	Sorted Linked List	Unsorted Linked List	Binary Search Tree	Balanced (how?) Binary Search Tree
Retrieve	Logarithmic	Linear	Linear	Linear	Linear	Logarithmic
Insert	Linear	Constant (?)	Linear	Constant	Linear	Logarithmic
Delete	Linear	Linear	Linear	Linear	Linear	Logarithmic

Idea #1: Restricted Table

- Perhaps we can do better if we do not implement a Table in its full generality.

Idea #2: Keep a Tree Balanced

- Balanced Binary Search Trees look good, but how to keep them balanced efficiently?

Idea #3: "Magic Functions"

- Use an unsorted array of key-data pairs. Allow array items to be marked as "empty".
- Have a "magic function" that tells the index of an item.
- Retrieve/insert/delete in constant time? (Actually no, but this is still a worthwhile idea.)

We will look at what results from these ideas:

- From idea #1: Priority Queues
- From idea #2: Balanced search trees (2-3 Trees, Red-Black Trees, B-Trees, etc.)
- From idea #3: Hash Tables

Advanced Table Implementations

Overview

We will cover the following advanced Table implementations.

- Balanced Search Trees
 - Binary Search Trees are hard to keep balanced, so to make things easier we allow more than 2 children:
 - ✓ • **2-3 Tree**
 - Up to 3 children
 - **2-3-4 Tree**
 - Up to 4 children
 - **Red-Black Tree**
 - Binary-tree representation of a 2-3-4 tree
 - Or back up and try a balanced Binary Tree again:
 - **AVL Tree**
- Alternatively, forget about trees entirely:
 - **Hash Tables**
- Finally, “the Radix Sort of Table implementations”:
 - **Prefix Tree**

Review

2-3 Trees [1/6]

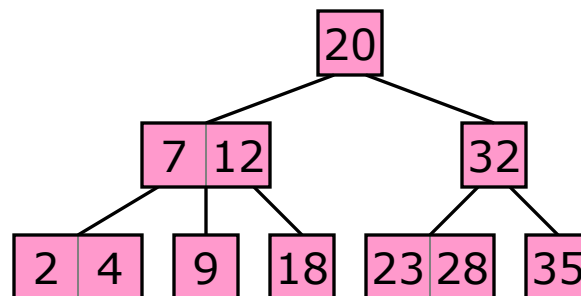
Obviously (?) a Binary Search Tree is a useful idea. The problem is keeping it balanced.

- Or at least keeping the height small.

It turns out that small height is much easier to maintain if we allow a node to have more than 2 children.

But if we do this, how do we maintain the “search tree” concept?

- We generalize the idea of an inorder traversal.
- For each pair of consecutive subtrees, a node has one data item lying between the values in these subtrees.



Review

2-3 Trees [2/6]

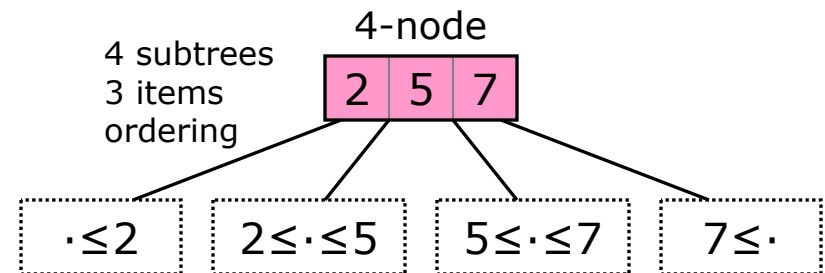
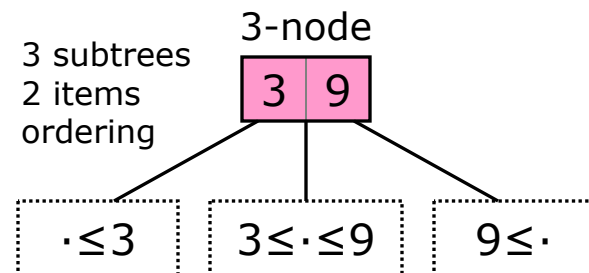
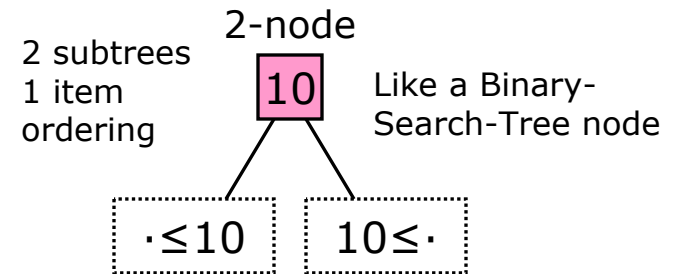
A Binary-Search-Tree style node is a **2-node**.

- This is a node with 2 subtrees and 1 data item.
- The item's value lies between the values in the two subtrees.

In a "2-3 Tree" we also allow a node to be a **3-node**.

- This is a node with 3 subtrees and 2 data items.
- Each of the 2 data items has a value that lies between the values in the corresponding pair of consecutive subtrees.

Later, we will look at "2-3-4 trees", which can also have **4-nodes**.

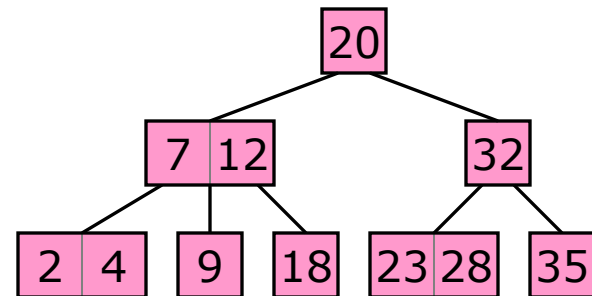


Review

2-3 Trees [3/6]

A **2-3 (Search) Tree** is a tree with the following properties.

- All nodes contain either 1 or 2 data items.
 - If 2 data items, then the first is \leq the second.
- All leaves are at the same level.
- All non-leaves are either *2-nodes* or *3-nodes*.
 - They must have the associated order properties.



To **retrieve** in a 2-3 Tree:

- Begin at the root, and go down, using the order properties, until the item is found, or clearly not in the tree.

To **traverse** a 2-3 Tree:

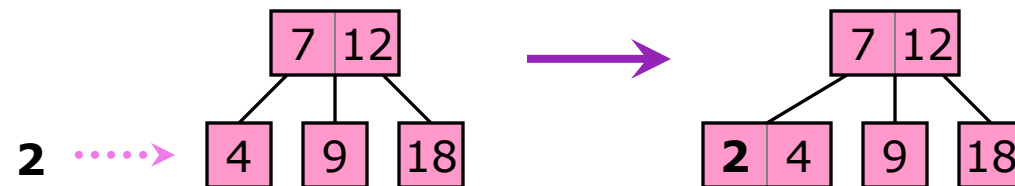
- Use the appropriate generalization of inorder traversal.
- Items are visited in sorted order.

Review

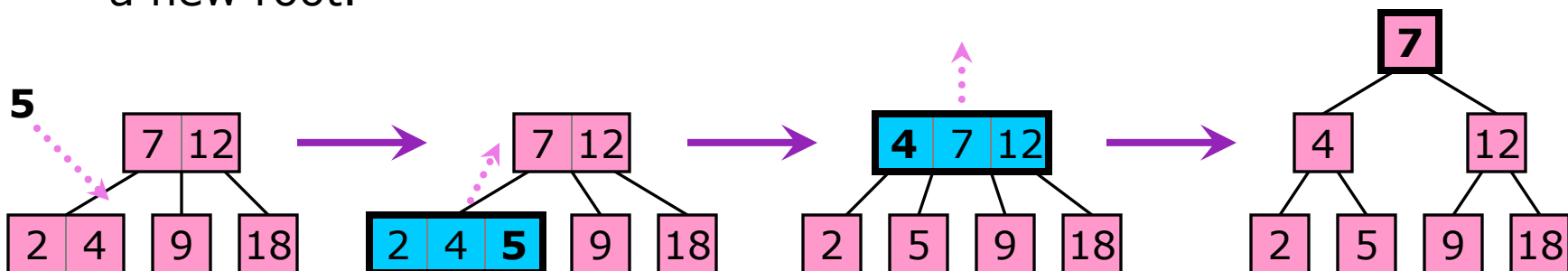
2-3 Trees [4/6]

To **insert** in a 2-3 Tree:

- Find the leaf that the new item should go in.
- If it fits, then simply put it in.



- Otherwise, there is an overfull node. Split it, and move the middle item up, either recursively inserting it in the parent, or else creating a new root.

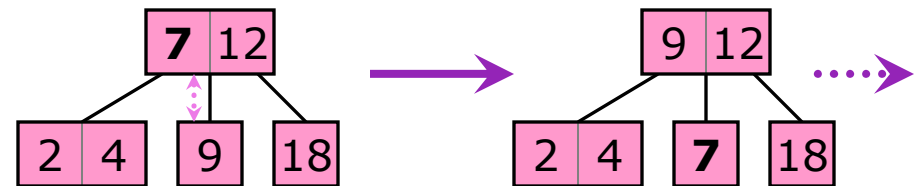


Review

2-3 Trees [5/6]

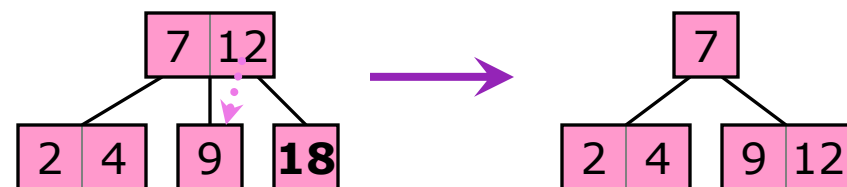
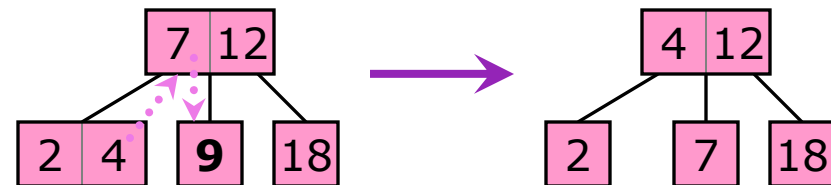
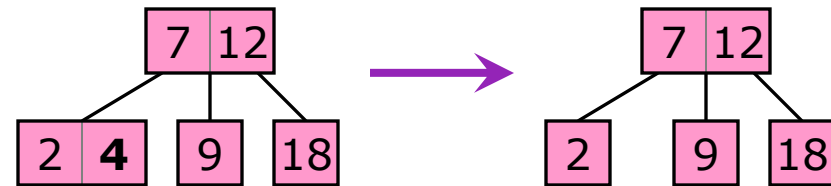
To **delete** in a 2-3 Tree:

- Find the item. If it is not a leaf, swap with its successor.
- Do the recursive delete-a-leaf procedure.



To delete-a-leaf:

- Easy Case: If the item is in a 3-node, then simply remove it.
- Semi-Easy Case: Otherwise, if the node has a consecutive sibling that is a 3-node, do a rotation with the parent.
- Hard Case: Otherwise, bring the parent down, combining it with a consecutive sibling.
 - Use recursive delete-a-leaf on the parent.



Review

2-3 Trees [6/6]

What is the order of the following operations for a 2-3 Tree?

- Traverse
 - $O(n)$ [as usual].
- Retrieve
 - $O(\log n)$.
 - The number of steps is roughly proportional to the height of the tree.
- Insert
 - $O(\log n)$.
 - Comments as for Retrieve.
- Delete
 - $O(\log n)$.
 - Comments as for Retrieve.

This is the **first** time we have seen a delete-by-key that handles *any* given key and is faster than linear time.

This is what we have been looking for.

A 2-3 Tree is a good basis for an implementation of a Table.

However, there are better bases.

- Not necessarily a **lot** better, but better.

Other Balanced Search Trees Better Than a 2-3 Tree?

Again, the Table operations retrieve, insert, and delete are all $O(\log n)$ for a 2-3 Tree implementation.

We do not know any structure in which all operations are $O(\log n)$ [worst case], and at least one is faster.

- Of course, we **can** make some operations better & some worse. For example, for an unsorted Linked List implementation, Table insert is $O(1)$, while Table retrieve & delete are $O(n)$.

However, we can make everything a **little** faster than a 2-3 Tree, although still $O(\log n)$.

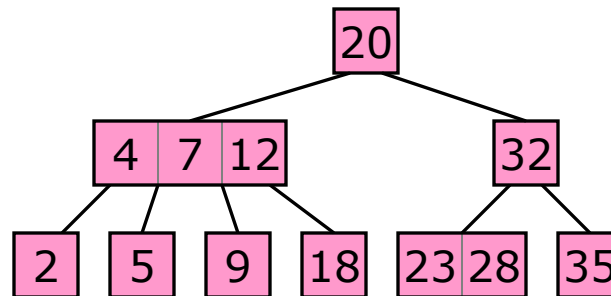
We do this with other kinds of balanced search trees.

- These are all similar to a 2-3 Tree.
- Thus, we will not look at them in great detail.
- See the text for details.

Other Balanced Search Trees

2-3-4 Trees

In a **2-3-4 Tree**, we also allow 4-nodes.



The insert and delete algorithms are not terribly different from those of a 2-3 Tree.

- They are a little more complex.
- And they tend to be a little faster.

Why not also allow 5-nodes (a "2-3-4-5 Tree")?

- Because the algorithms tend to be a little slower.

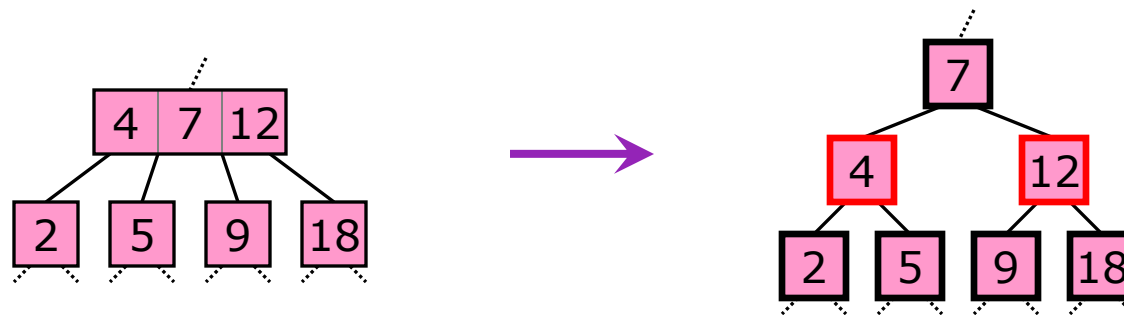
Other Balanced Search Trees

Red-Black Trees — Idea [1/3]

It turns out that we can increase the efficiency of 2-3-4 Tree operations by representing the tree using a Binary Search Tree plus a little more information.

- The representation we will discuss is called a **Red-Black Tree**.

Consider the 4-node below. We can represent this part of the 2-3-4 Tree using only 2-nodes if we add two new nodes (shown in red).

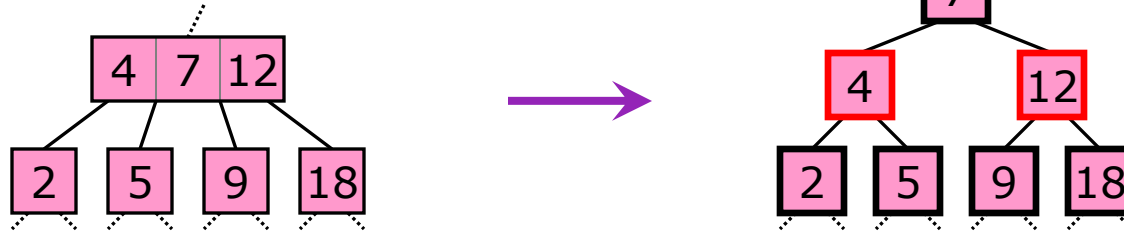


Note that the ordering property of the 2-3-4 Tree translates into the ordering property of a Binary Search Tree.

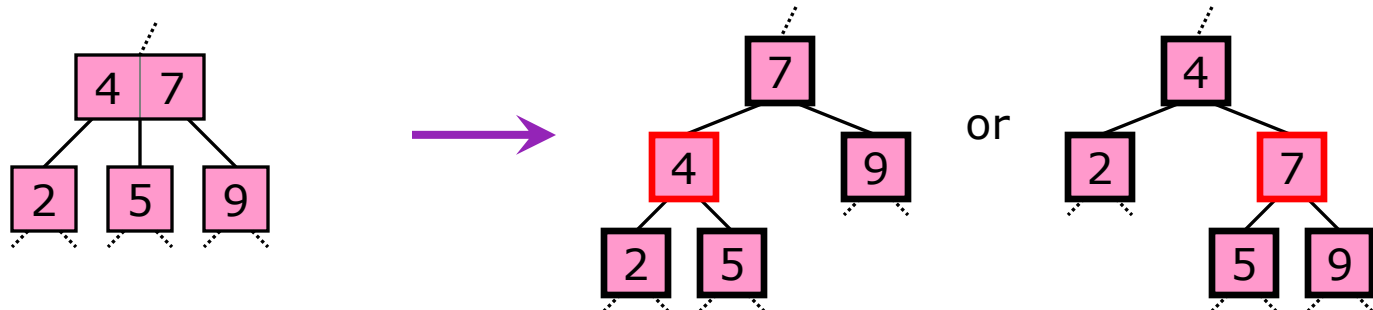
Other Balanced Search Trees

Red-Black Trees — Idea [2/3]

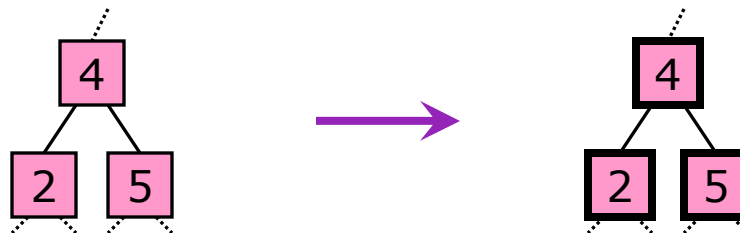
Here again is our transformed 4-node.



We can also apply this process to a 3-node (in two different ways).



2-nodes are essentially left alone.

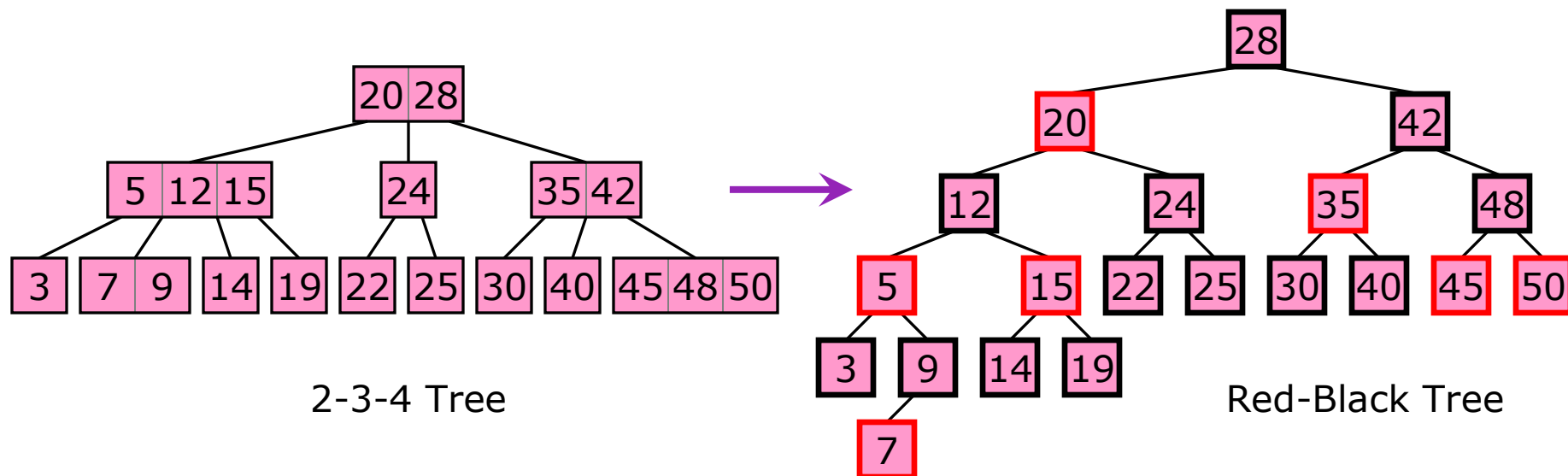


Other Balanced Search Trees

Red-Black Trees — Idea [3/3]

A **Red-Black Tree** is a Binary-Tree representation of a 2-3-4 Tree.

- A R.B.T. is a Binary Search Tree in which each node is “**red**” or “**black**”.
- Think of **black** nodes as representing 2-3-4 Tree nodes.
- Think of **red** nodes as being the extra ones required to make a Binary Tree out of the 2-3-4 Tree.



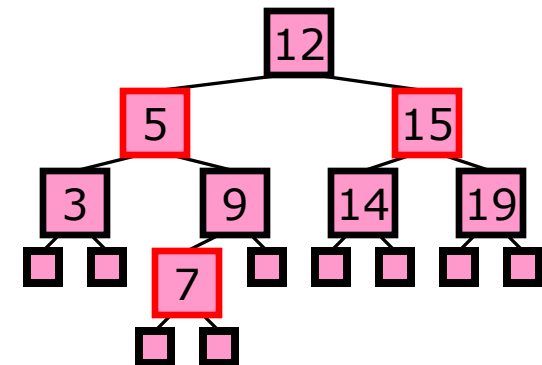
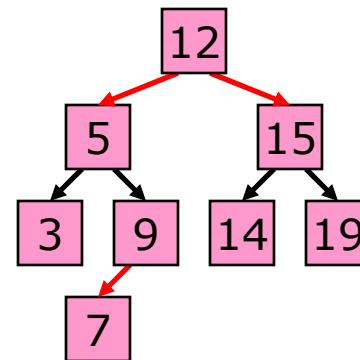
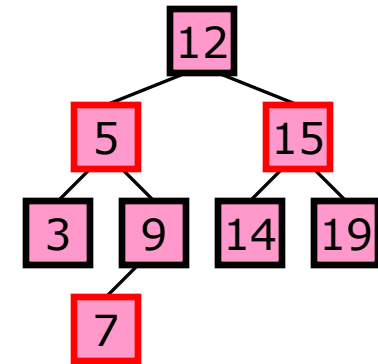
- It is no longer true that every leaf is at the same level. However, given a node, every path from it down to a leaf goes through the same number of **black** nodes.

Other Balanced Search Trees

Red-Black Trees — Variations

Implementations of Red-Black Trees vary.

- I have presented them as having red and black **nodes**.
- The text talks about red and black **pointers**.
 - Note that the root is always black, so it does not matter whether the root's color is stored somewhere.
- Some versions add "**null nodes**" at the bottom.
 - Null nodes are black and have no data.
 - All leaves are null nodes, and all null nodes are leaves.



Other Balanced Search Trees

Red-Black Trees — Usage

How do we use Red-Black Trees?

- Retrieve and traverse are exactly the same as for Binary Search Trees. Just ignore the color.
- Insert and delete are based on the algorithms for 2-3-4 Trees.

Why do we use Red-Black Trees?

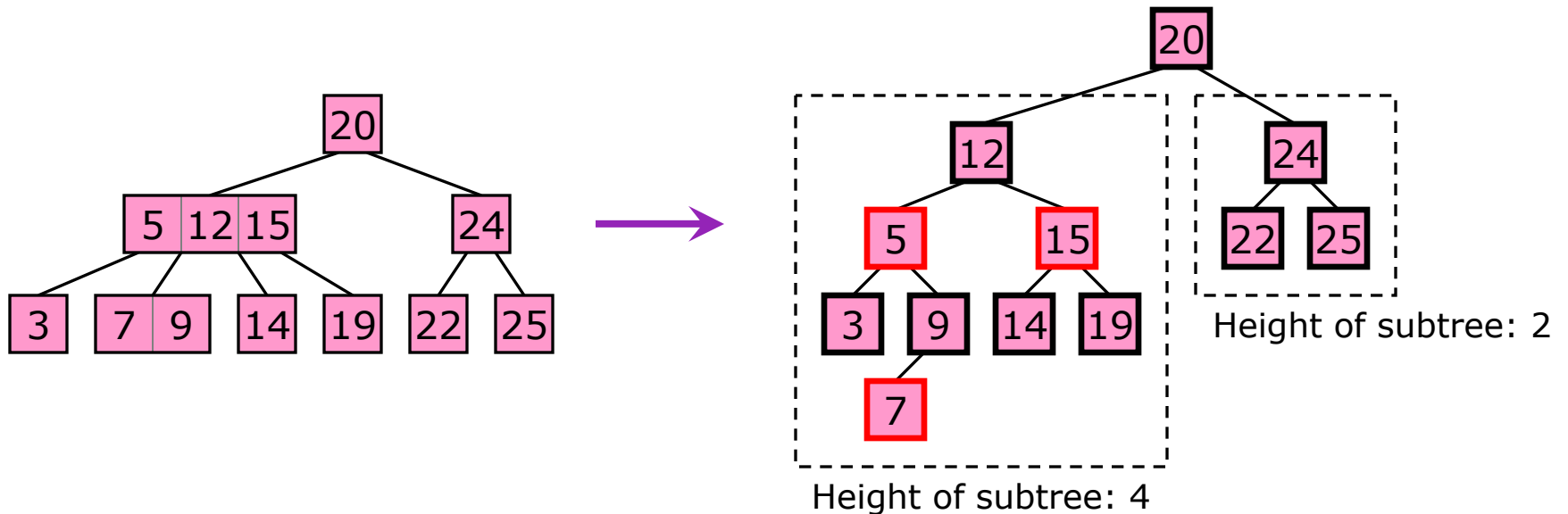
- Because they tend to be just a little more efficient than 2-3-4 Trees, which are just a little more efficient than 2-3 Trees.
- All three have $O(\log n)$ insert, delete, and retrieve.

Red-Black Trees are the most common basis for implementations of C++ STL Tables (`std::set`, `std::map`, etc.).

Other Balanced Search Trees

Red-Black Trees — Notes [1/2]

A Red-Black Tree is not necessarily a balanced Binary Tree, as we defined “balanced” earlier.



However, a Red-Black Tree with n nodes cannot have height more than $2 \log_2(n + 1)$.

Thus, the height is $O(\log n)$, which makes the retrieve, insert, and delete operations $O(\log n)$.

Other Balanced Search Trees

Red-Black Trees — Notes [2/2]

In practice, we *never* do the 2-3-4 Tree to Red-Black Tree conversion. Rather, we implement only a Red-Black Tree.

- The conversion was illustrated here in order to explain where Red-Black Trees come from and how they work.

If you need to implement a balanced search tree, use a Red-Black Tree.

- The insert & delete algorithms get rather complex. Look up the details.

Other Balanced Search Trees

AVL Trees — Definition

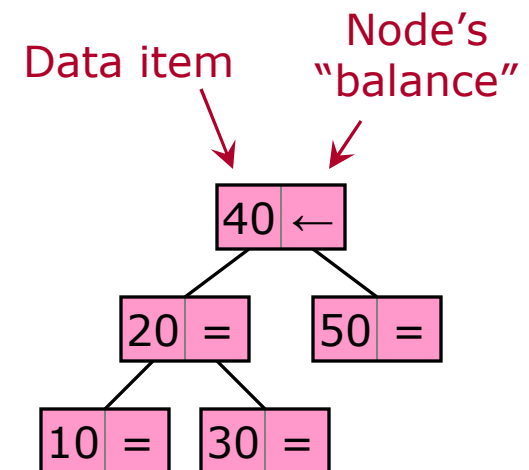
The first kind of self-balancing search tree was the “AVL Tree”.

- AVL trees are named after the authors of a 1962 paper describing them: Georgy Maximovich **A**delson-**V**elsky and Yevgeniy Mikhailovich **L**andis.
- These days, AVL Trees are mostly a historical curiosity.

An **AVL Tree** is a balanced (in our original, strict sense) Binary Search Tree in which each node has an extra piece of data: its “balance”: left high [\leftarrow], right high [\rightarrow], or even [=].

- Recall: a Binary Tree is *balanced*, if, for each node in the tree, its two subtrees have heights differing by at most 1.

A-V & L discovered logarithmic-time algorithms to do insert and delete while maintaining the balanced property.

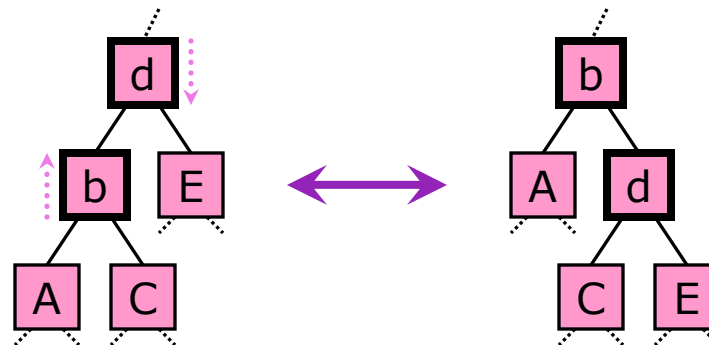


Other Balanced Search Trees

AVL Trees — Rotation

We will not cover all of the details of the AVL Tree algorithms.

- We note that they rest on an operation known as **rotation**.
- Rotation is pictured below. For nodes labeled A, C, E, the subtrees of which they are the roots are moved along with them.
- Note that we have seen something (roughly) like this before, in the “semi-easy case” of 2-3 Tree deletion.



When we allow rotations, we can insert or delete using at most $O(\log n)$ operations, while maintaining the balanced property.

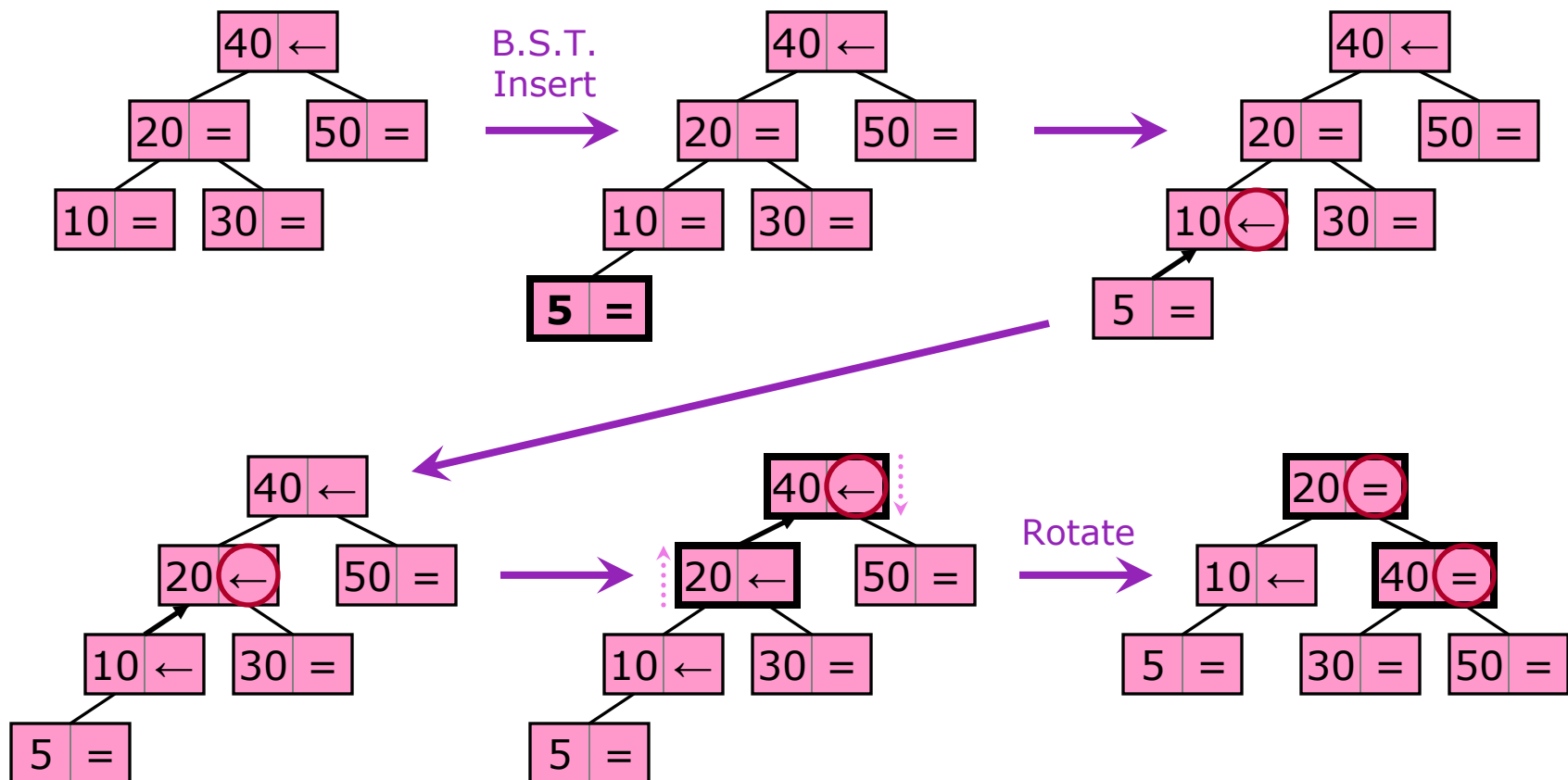
- Thus, insert and delete (and, by the balanced property, retrieve) are $O(\log n)$ operations for an AVL Tree.

Other Balanced Search Trees

AVL Trees — Example

Quick example of AVL Tree insert: Do Binary Search Tree insert, then proceed up to the root, adjusting “balances” and, if needed, rotating.

- Below we illustrate Insert 5.



Other Balanced Search Trees

Wrap-Up

All balanced search trees offer an implementation of the Table ADT in which the insert, delete, and retrieve operations are $O(\log n)$.

Generally, the Red-Black Tree is agreed to have best **overall** performance.

- It is the one that tends to be used to implement things like `std::map`.
- The word “overall” is important. For example, an AVL Tree has a faster retrieve operation than a Red-Black Tree. (But a sorted array has an even faster retrieve; no one uses AVL Trees.)

Implementation details may be changed due to various trade-off's.

- Space vs. time, etc.

Hash Tables

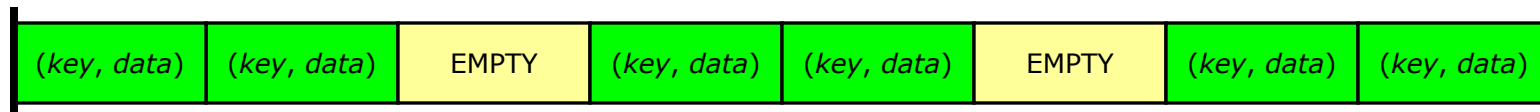
Introduction [1/4]

Balanced search trees allow the 3 primary Table operations to be $O(\log n)$. Is there an even more efficient implementation?

- Sort of.

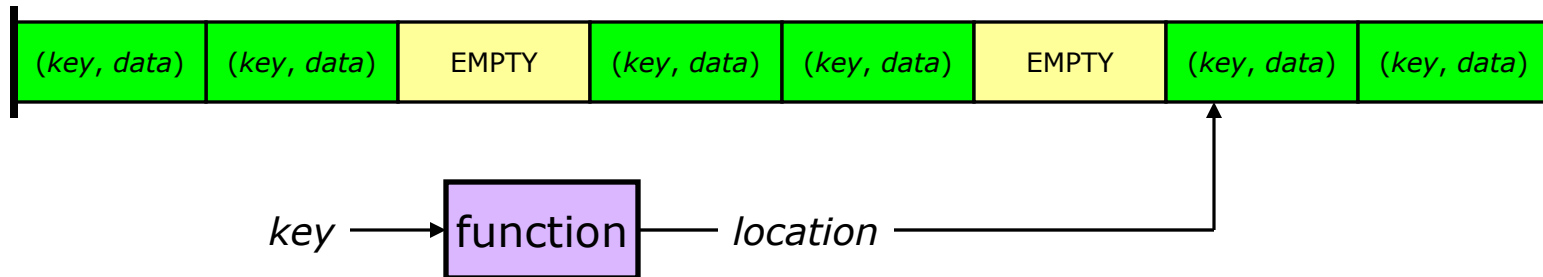
Suppose we want to implement a Table using an unsorted array.

- Insert is fast.
 - If no reallocation, constant time.
- Delete-by-position is fast **if** we allow **gaps** in our data.
 - To delete an item that is the result of a previous look-up, just set its value to "empty". Again: constant time.
 - However, to do a delete-by-key (Table delete) we need to find (retrieve) the proper item, and unfortunately ...
- Retrieve is **slow**.
 - $O(n)$, in fact (Sequential Search).



Hash Tables

Introduction [2/4]



What if we had a magic function which, given a key, returned the location of the item, if it is present?

- Then all three operations (insert, delete, retrieve) would be constant time!
- Note: Items might no longer be in sorted order.

Unfortunately, such a function is generally impractical. Given reasonable constraints (speed, etc.), it is usually **impossible**.

- But we can try.
- This is the basic idea behind a "Hash Table".

Hash Tables

Introduction [3/4]

A **hash function** is a function that “wants to be” our perfect location generator from the last slide.

- Typically, we take a key as input, and “mess it up”, so that the output (location) bears no resemblance to the input.
- Thus the name: “hash”.

A **Hash Table** is a data structure in which items are stored according to the location specified by a hash function.

- Typically the items are stored in an array.

A well-designed Hash Table gives us an implementation of the Table ADT which is very fast **most of the time**.

- Worst-case performance can be poor, however.

Hash Tables

Introduction [4/4]

The problem with Hash Tables is this:

- Allocating space for every possible key is often impractical.
 - For example, suppose our keys are strings.
- Therefore, we use a small-ish array (or similar structure). Items with different keys *may* have the same hashed value.
- When this happens, we have a **collision**. Dealing with it is called **collision resolution**.

We will discuss various collision-resolution methods.

In general:

- Collisions are not a big problem as long as there are not very many of them.
- But there might be lots of them.
- In fact, it is possible that, for *every* item we insert, the hash function has the same output. (Ick.)