Tables in Various Languages

CS 311 Data Structures and Algorithms
Lecture Slides
Wednesday, November 28, 2012

Chris Hartman
Department of Computer Science
University of Alaska Fairbanks
cmhartman@alaska.edu
Based on material by Glenn G. Chappell

Review
Where Are We? — The Big Problem

Our problem for much of the rest of the semester:

- Store: a collection of data items, all of the same type.
- Operations:
  - Access items [one item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- All this needs to be efficient in both time and space.

A solution to this problem is a **container**.

Generic containers: those in which client code can specify the type of data stored.

Unit Overview
Tables & Priority Queues

Major Topics

- ✓ Introduction to Tables ← Lots of lousy implementations
- ✓ Priority Queues
- ✓ Binary Heap algorithms
- ✓ Heaps & Priority Queues in the C++ STL

  } Idea #1: Restricted Table

- ✓ 2-3 Trees
- ✓ Other balanced search trees

  } Idea #2: Keep a Tree Balanced

- Hash Tables

  } Idea #3: "Magic Functions"

- Prefix Trees

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

Overview of Advanced Table Implementations

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 [1/4]

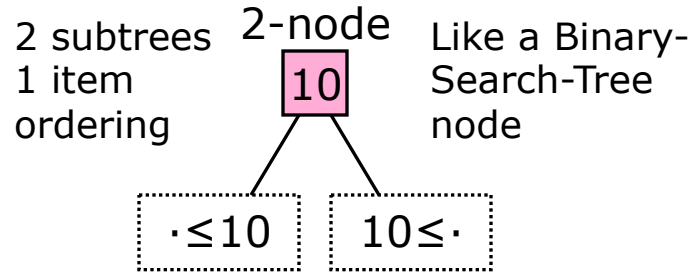
A Binary-Search-Tree style node is a **2-node**.

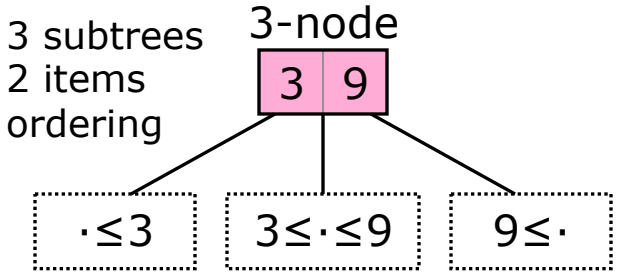
- This is a node with 2 subtrees and 1 data item.
- The item's value lies between the values in the two subtrees.

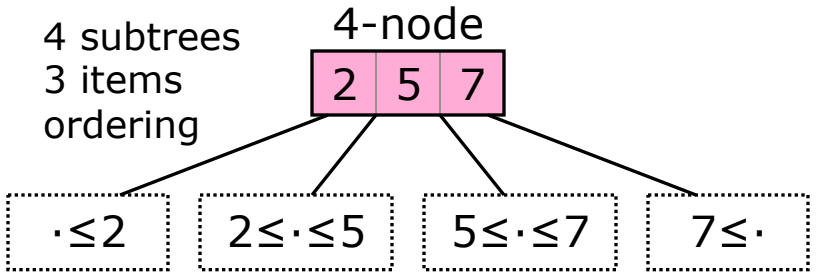
In a "2-3 Tree" we also allow a node to be a **3-node**.

- 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**.



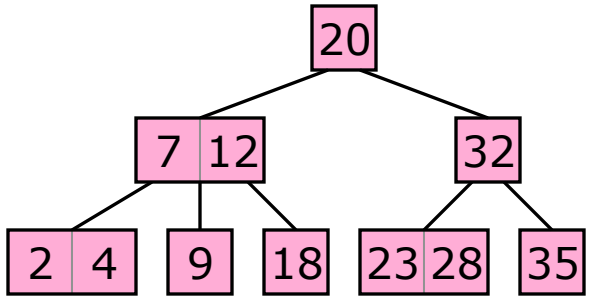




Review
2-3 Trees [2/4]

A **2-3 Search Tree** (generally we just say **2-3 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 ≤ 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.

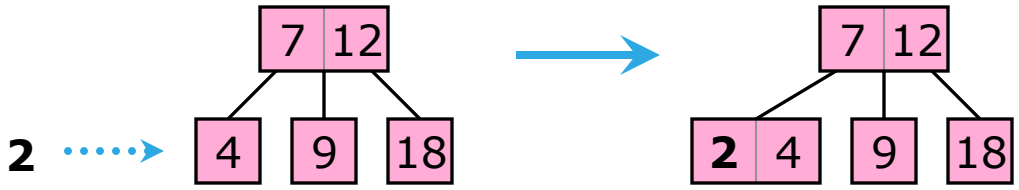
To **traverse** a 2-3 Tree:

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

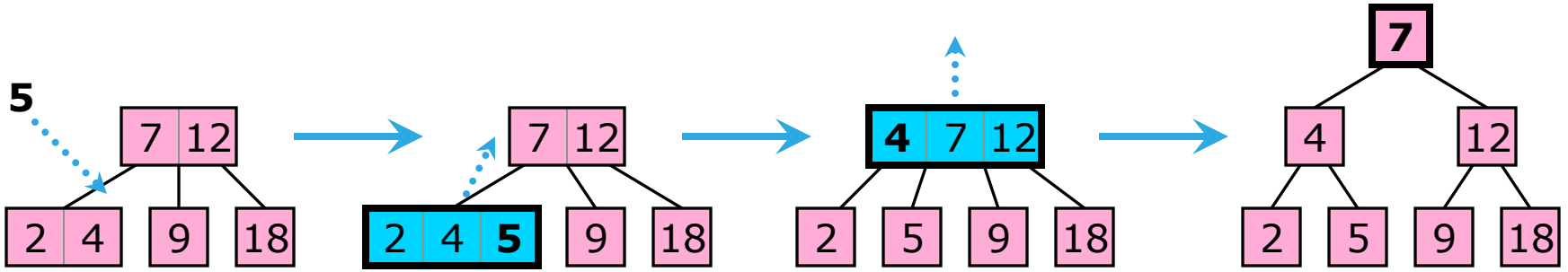
Review
2-3 Trees [3/4]

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 [4/4]

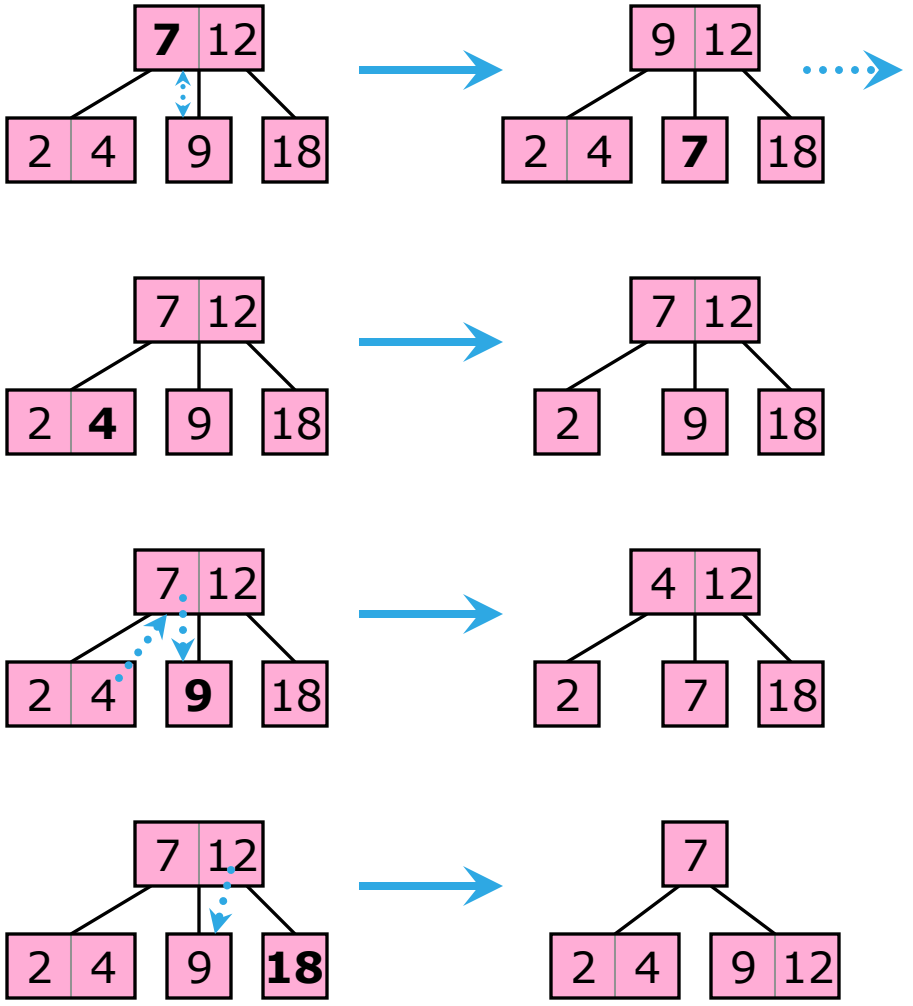
To **delete** in a 2-3 Tree:

- Find the item. If it is not in 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 node with another item, simply remove it.
- **Semi-Easy Case**: Otherwise, if the node has a consecutive sibling with two items, 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.

When doing a recursive "delete-a-leaf" on a non-leaf node, drag along subtrees.



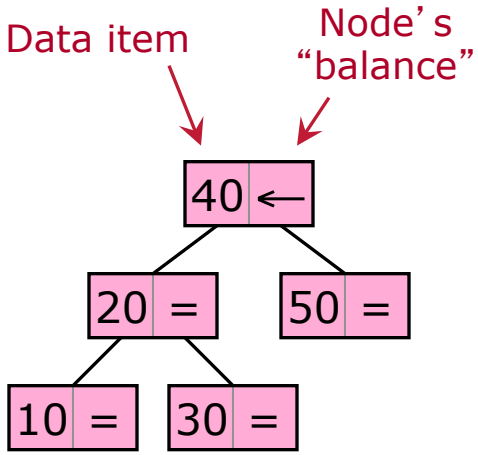
Other Balanced Search Trees
AVL Trees — Definition

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 [←], right high [→], 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.

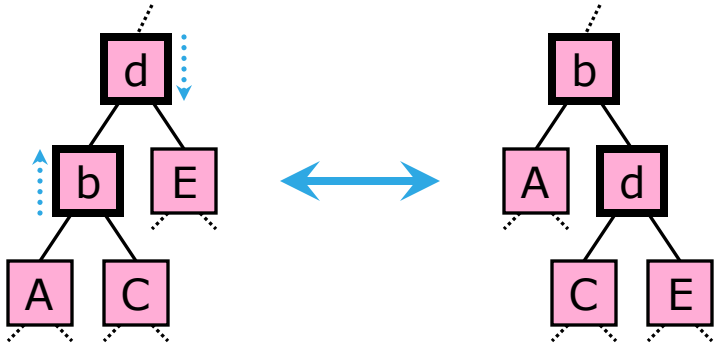


A-V & L discovered logarithmic-time algorithms to do insert and delete while maintaining the balanced property.

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.



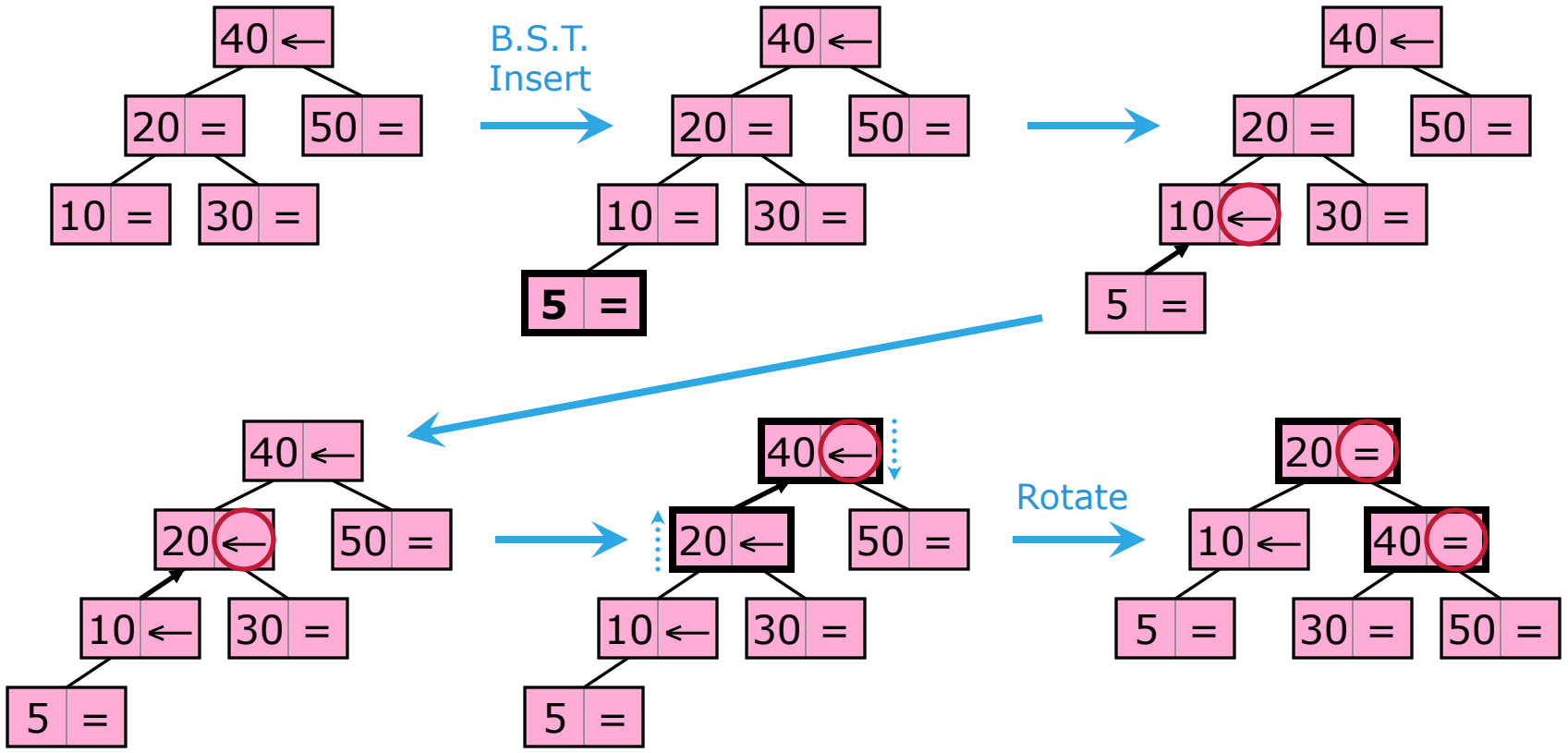
When we allow rotations, we can insert or delete using at most $O(\log n)$ operations, while maintaining the balanced property.

- 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.

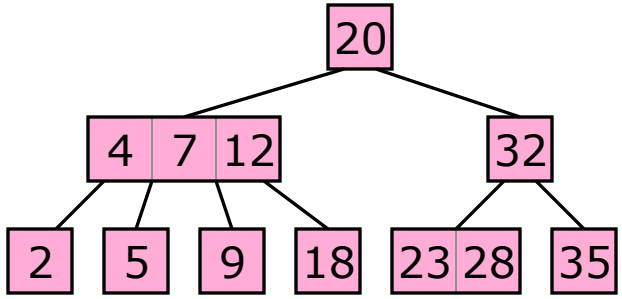
- It is the one that tends to be used to implement things like **`std::map`**.
- The word "overall" is important. For example, an AVL Tree has a faster retrieve operation than a Red-Black Tree.
  - But a sorted array has an even faster retrieve; no one uses AVL Trees.

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

- Space vs. time, etc.

Other Balanced Search Trees [1/4]

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



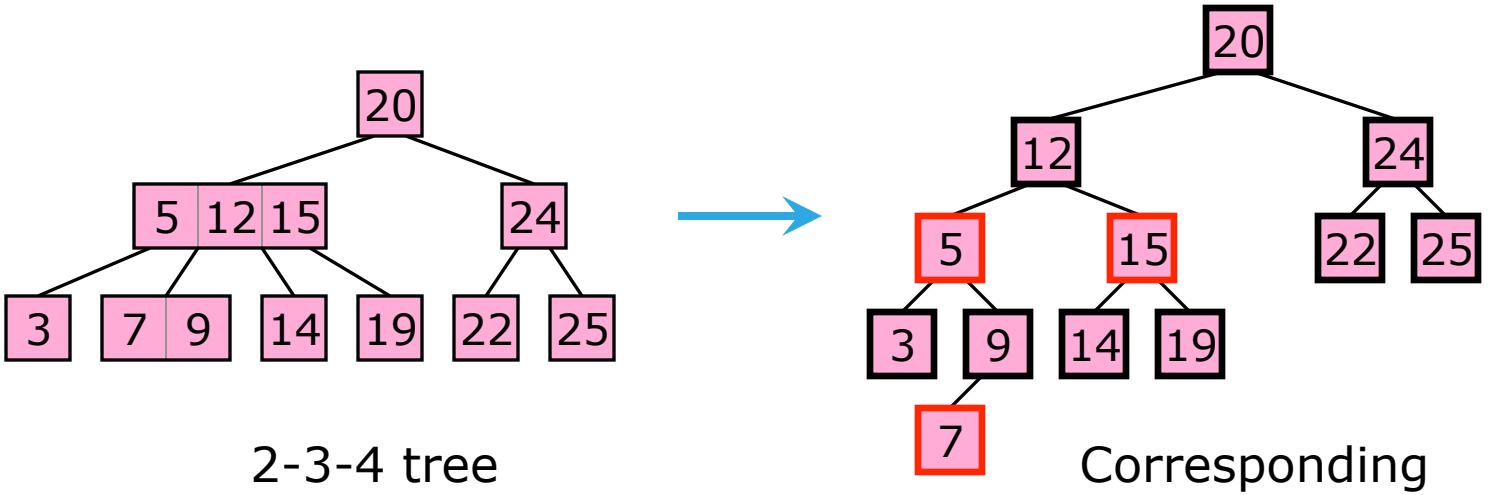
The insert and delete algorithms are not terribly different from those of a 2-3 Tree.

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

Review
Other Balanced Search Trees [2/4]

A very efficient kind of balanced search tree is a **Red-Black Tree**.

- This is a Binary-Search Tree representation of a 2-3-4 tree.
- Each node in a Red-Black Tree is either **red** or **black**.
- Each node in the 2-3-4 Tree corresponds to a **black node**.
- The **red nodes** are the extra ones we need to add.
- Red-Black Trees may not be balanced (in the strict sense). However, each path from the root to a leaf must pass through the same number of **black nodes**.



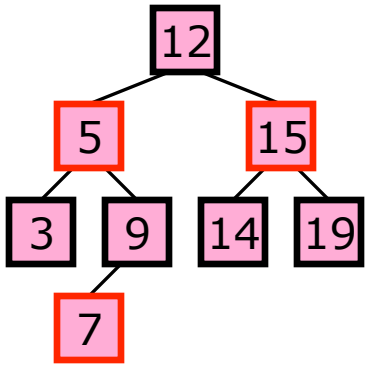
2-3-4 tree

Corresponding Red-Black Tree

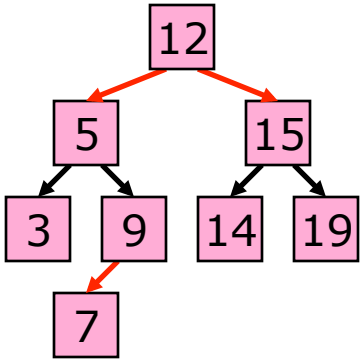
Review
Other Balanced Search Trees [3/4]

Implementations of Red-Black Trees vary.

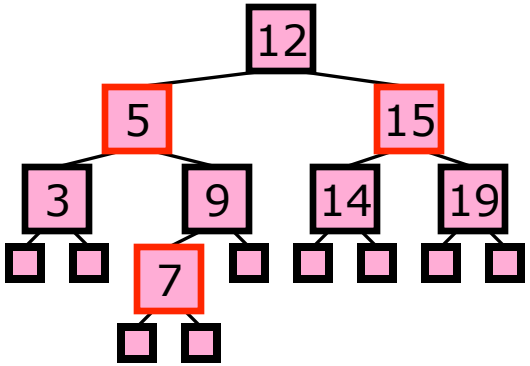
- I have presented them as having red and black **nodes**.



- 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 (most?) 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.



Review
Other Balanced Search Trees [4/4]

All balanced search trees (2-3 Trees, 2-3-4 Trees, Red-Black Trees, AVL Trees, etc.) have:

- $O(\log n)$ retrieve, insert, delete.
- $O(n)$ traverse (sorted).

Best **overall** performance for **in-memory** data, when we mix up retrieves, inserts, and deletes.

Retrieve & Sorted Traverse

- For Red-Black Trees and AVL Trees, use the B.S.T. algorithms (traverse = inorder traverse).
- For 2-3 Trees and 2-3-4 Trees, use the obvious generalization of the B.S.T. algorithms.

Insert & Delete

- These are more complicated.
- For 2-3 Trees, we looked at the algorithms in some detail.
- The 2-3-4 Tree algorithms are similar, generally requiring fewer operations.
- For Red-Black Trees, do something similar, but fancier.

Notes on Assignment 7
Introduction

In Assignment 7, you are to write a simple Binary Search Tree (class **`BSTree`**), with copying, retrieve, insert (but not delete!), and the three standard traversals.

Here are some ideas about writing the code.

This is a **node-based structure**.

- The general organization should be much like the Linked List you wrote: a class for the data structure as a whole, and a node class that the client code never sees.
- The node class should have three data members: a data item, and two child pointers: left & right. Each of these pointers is NULL if there is no child.
- Class **`BSTree`** should have two data members: root pointer (pointer to a node) and size.

Notes on Assignment 7
Helper Functions 1

Some useful private helper functions to write:

`copy`

- Copying is easy to do recursively. But your copy constructor cannot be recursive. So write a private recursive helper function **`copy`**.
- Function **`copy`** takes a pointer to a node, copies the tree rooted at that node, and then returns a pointer to the root of the new tree.
- Do this recursively:
  - If the given pointer is NULL, return NULL (base case). Otherwise, return a new node whose data is the same as the data in the given node, and whose left and right child pointers are gotten by recursive calls to **`copy`** on the given node's left & right child pointers.
- Make sure to think about exception safety – no memory leaks!

`swap`

- As usual: call **`std::swap`** on each data member.

Notes on Assignment 7
Helper Functions 2

Some useful private helper functions to write:

`find`

- Finding is easy to do recursively, but what should we pass and return?
- Pass: The root of a subtree to search, and a key to find.
- Return: A pointer pointing at the node where the data is (or should be). But **`insert`** needs to change this. Returning a reference to this pointer will make everything easy.
- Function **`find`** takes a pointer to the root of a (sub)tree and a key, and returns a reference to a pointer to a node.
- We need const and non-const versions

```
Node * & find(Node * & root, const Keytype& key)
const Node * & find(const Node * & root, const Keytype & key) const
```

Notes on Assignment 7
Ideas for Writing Functions [1/2]

Copy constructor

- Do everything in an initializer list. Set the root pointer to the result of calling helper function **`copy`** on the other object's root pointer. Set size to the other object's size.

Copy assignment operator

- Use the swap trick.

`retrieve`

- Call helper function **`find`** with the root of your tree and the given key. Return **`true`** if **`find`** returns a non-NULL value. Otherwise, return **`false`**.

`insert`

- Call helper function **`find`** with the given key. Return **`false`** if **`find`** returns a non-NULL value. Otherwise, set the given pointer to point to a new node holding the given key, increment size, and return **`true`**. (be careful to use the reference find() returns, not a copy of it. In particular "**`Node *here = find(…); here = new Node(…);`**" would not work.)

Notes on Assignment 7
Ideas for Writing Functions [2/2]

`preorderTraverse`

- Take the iterator **by value**. Then call a private helper function with the given iterator and the head pointer.
- Private helper function: takes iterator **by reference** (since it modifies the iterator) and a pointer to a node. If the pointer is NULL, return. Otherwise, do ***iterator`++ =` pointer`->data_`**, and then make two recursive calls:
  - One taking *iterator* and *pointer*`->left`.
  - One taking *iterator* and *pointer*`->right`.

`inorderTraverse`, `postorderTraverse`

- Write the same way as function **`preorderTraverse`**, but do the ***iterator`++ =` pointer`->data_`** operation in the appropriate place.
- Note that these will require *different* helper functions.
  - Or possibly one helper function that takes a parameter to know when to do the "visit".

Tables in Various Languages
Overview

We now take a brief look at Table usage in various languages, beginning with C++.

- C++ STL
  - Sets: **`std::set`**.
  - Maps: **`std::map`**.
  - Other Tables.
  - Set algorithms.
- Other Languages
  - Python.
  - Perl.
  - Lisp.

C++ STL: Refresher — `std::pair` [1/2]

The C++ STL contains a "pair" template: **`std::pair`**, in **`<utility>`**. It acts as if it is declared roughly like this:

- Note the public data members.

```
template<typename T, typename U>
struct pair {
    T first;
    U second;
};
```

Other members, including **`operator<`** and **`operator==`**, exist.

- So you can put a **`std::pair`** into a sorted container, as long as types **`T`** and **`U`** have **`operator<`**.
- You can also do **`std::find`** (Sequential Search) to look for a **`std::pair`**, as long as types **`T`** and **`U`** have **`operator==`**.

C++ STL: Aside — `std::pair` [2/2]

Example:

```
std::pair<int, double> p;
p.first = 3;
p.second = 4.5;
// Or we can do the above using a ctor:
std::pair<int, double> p2(3, 4.5);
```

What do we use a **`std::pair`** for?

- To return more than one value from a function.
  - Remember **`fibo3.cpp`**?
- To specify a range, using two iterators.
  - As in the return value of **`std::equal_range`**, a Binary Search variation.
- To store a key-data pair.
  - For use in a Table implementation.

Tables in Various Languages
C++ STL: `std::set` — Introduction

The simplest STL Table implementation is **`std::set`**, in **`<set>`**.

- The key type and value type are the same.
  - That is, the key is the whole data item.
- Duplicate (equivalent) keys are not allowed.
  - Thus, all you can say about a value is whether or not it is in the structure.
  - In short, it is just what it says it is: a set.
- The specification was put together with a balanced search tree in mind. Most implementations use a Red-Black Tree.

Declare a set as follows:

```
std::set<valuetype> s;
```

An optional template parameter specifies the comparison used.

- This is done just as for sorting, Heap algorithms, etc.
- The default is to use **`operator<`**.

```
std::set<valuetype, comparison> s;
```

Tables in Various Languages
C++ STL: `std::set` — Iterators

A **`std::set::iterator`** is a bidirectional iterator.

- Items appear in sorted order.
- So a **`std::set`** is pretty nearly a SortedSequence, right?

A **`std::set::iterator`** is not a **mutable iterator**, that is, one cannot do "**`*iter = v;`**".

- Why not?
  - Because items are stored in sorted order. Changing an item might break this invariant.

Iterators and references are valid until the item is erased.

- What does this tell us about the implementation?
  - Clearly, *references* stay valid, because this is a node-based structure.
  - A Red-Black tree can be reorganized by an insertion or deletion. Thus, *iterators* must *not* store information about the structure of the tree (as they do in a **`std::deque`**).
  - So we must be able to find our way around the tree starting at a leaf. This means the tree must have **parent pointers**.
  - Conclusion: Give the tree parent pointers, and make the iterator a wrapper around a pointer.

C++ STL: `std::set` — Major Operations [1/2]

Table Insert (**`insert`**)

- Given an item.
- Inserts given item into the set.
- **Does nothing** if an equivalent item (key) is already in the set.
- Returns a **`std::pair<`***iterator*`, bool>`**. The iterator points to the inserted item or the already present item. The **`bool`** is **`true`** if the insertion happened.

```
std::set<int> s;
s.insert(3);
if (!s.insert(4).second)
   cout << "4 was already present" << endl;
```

Tables in Various Languages
C++ STL: `std::set` — Major Operations [2/2]

Table Delete (**`erase`**)

- Given a key **or** an iterator.
- Deletes the proper item (if any) from the set.

```
s.erase(3);
s.erase(s.begin());
```

Table Retrieve (**`find`**)

- Given a key.
- Retrieves: returns an iterator, which either points to the item or is **`end()`**.

```
If (s.find(3) != s.end())
   cout << "3 was found" << endl;
```

- Why not just use **`std::find`** or **`std::binary_search`** (or a variation)?
  - They both work.
  - Both are linear time, the former because it always is, the latter because this is not random-access data. However, **`set<K>::find`** is logarithmic time.

Tables in Various Languages
C++ STL: `std::set` — Other Operations

There are many other members in **`std::set`**, including range insert & erase, etc.

One interesting member function is **`insert`** with "hint".

- This works like regular **`insert`**, but it is given a second parameter: an iterator. It returns an iterator to the item.
- The second parameter (iterator) is a "hint" as to where the item should be inserted.
- The code *may* ignore the hint, but it probably uses it.
- How do you think this is typically implemented?
  - Probably the inorder traversal property of a Red-Black Tree is used to look for locations "near" the given one.
- What is a good hint to give?
  - If you are inserting items in sorted order, a good hint is an iterator to the last item inserted.
- What is the likely effect of giving a bad hint?
  - Slower behavior [but still $O(\log n)$, as the Standard requires].

Tables in Various Languages
C++ STL: `std::map` — Introduction

The other main Table available in C++ is **`std::map`**, in **`<map>`**.

- The key and data types are specified separately.
- The value type is a pair: **`std::pair`**<*keytype*, *datatype*>.
- As with **`std::set`**:
  - Duplicate (equivalent) keys are not allowed.
  - The specification was put together with a balanced search tree in mind. The implementation is usually a Red-Black Tree.
  - An optional comparison can be specified. It defaults to using **`operator<`**.

Declaration:

```
std::map<keytype, datatype> m;
std::map<keytype, datatype, comparison> m2;
```

Major operations in **`std::map`** are much the same as for **`std::set`**.

- Insert operation: member function **`insert`**, given item. ← Different!
- Delete operation: member function **`erase`**, given key or iterator.
- Retrieve operation: member function **`find`**, given key.

Tables in Various Languages
C++ STL: `std::map` — Bracket Operator [1/3]

A very convenient operation is: *datatype* **`& operator[](`***key***`)`**

- This allows a map to be used like an array. Examples:

```
std::map<std::string, int> m;
m["abc"] = 7;
cout << m["abc"] << endl;
m["abc"] += 2;
```

- This can be defined as follows ("`k`" is the given key):

```
(*((m.insert(make_pair(k, data_type()))).first)).second
```

  - "Make sure key `k` is in the map, and give me the associated data."

C++ STL: `std::map` — Bracket Operator [2/3]

More `operator[]` examples:

```
std::map<int, int> m2;
m2[0] = 34;
m2[123456789] = 28;  // Very little memory used!

std::map<std::string, std::string> id;
id["Hubert Gump"] = "abc";
cout << id["Fred Smurg"] << endl;
// The above line inserts
//     std::pair<std::string, std::string>
//                   ("Fred Smurg", std::string())
// into the map. (Right?)
```

Tables in Various Languages
C++ STL: `std::map` — Bracket Operator [3/3]

A `map`'s `operator[]` is very convenient and useful. However …

- This `operator[]` **always inserts**. Thus, it has no `const` version.

```
void printEntry(const std::map<std::string, int> & m1)
{
    cout << m1["abc"] << endl;  // DOES NOT COMPILE!
}
```

- Because of this insertion, `operator[]` is generally *not* a good way to check whether a given key is already in the map. Instead, use `map<K,D>::find`.

```
std::map<Foo, Bar> m2;
Foo theKey;
// I want to test whether theKey lies in m2

if (m2.find(theKey) != m2.end())  // GOOD way to test
if (m2[theKey] == …)              // BAD way to test
```