Tables in Various Languages continued
External Data

CS 311 Data Structures and Algorithms

Lecture Slides

Friday, December 4, 2009

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
CHAPPELLG@member.ams.org

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
- (part) Tables in various languages

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.

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

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

- Table implementation. Key type & value type are the same.
- Duplicate keys not allowed.
- Implementation: balanced search tree.
- Iterators are bidirectional, not mutable (no "**`*iter = v;`**").
- Iterators & references valid until item is erased.

Operations

- Insert: member **`insert`**, takes item.
  - Returns **`std::pair<`***iterator*`, bool>`**.
  - Does nothing if key already present.
- Delete: Member **`erase`**, takes key or iterator.
- Retrieve: Member **`find`**, takes key.
  - Returns iterator, which is *container*`::end()` if not found.
- Also, **`insert`** with hint, takes item and "hint" iterator.
  - Returns iterator to item inserted.
  - Exists so that inserter iterators work with **`std::set`**.

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

`std::map<`*keytype*`, `*datatype*`> m;`

- Table implementation. Holds key-data pairs.
- Duplicate keys not allowed.
- Implementation: balanced search tree.
- Iterators are bidirectional, not mutable (no "**`*iter = v;`**").
  - But *can* do "**`(*iter).second = d;`**".
- Iterators & references valid until item is erased.

Operations

- As for **`std::set`**.
  - Note: For a **`map`**, items and keys are different. Member **`insert`** takes an item; members **`erase`** and **`find`** take keys.
- Also has bracket operator:
  - Say "**`m[key] = data;`**".
  - Inserts key into map if not present. Thus, can modify map; no const version. Do not use to determine whether a key is present; use **`find`**.

`std::multiset<`*`valuetype`*`> ms;`

`std::multimap<`*`keytype`*`, `*`datatype`*`> mm;`

- Like **`std::set`** & **`std::map`**, respectively, except that duplicate keys are allowed.
- Retrieve operations return either ranges or counts.
- Unlike **`std::map`**, **`std::multimap`** has no bracket operator.

Hash Tables

- Not found in 1998 C++ Standard.
- Nonstandard versions abound.
- Upcoming revised standard should have Hash Tables.

STL Set Algorithms

- Deal with sorted sequences specified with iterators.
- Names: **`includes`**, **`set_union`**, **`set_intersection`**, **`set_difference`**, **`set_symmetric_difference`**.
- To send output to a container (**`set`**?) use an **inserter**.

```
std::set_intersection(s1.begin(), s1.end(),
                      s2.begin(), s2.end(),
                      std::inserter(s3, s3.begin()));
```

See **`usesetalgs.cpp`**, *on the web page.*

Python includes a Table type, called a "dictionary" or "dict".

- Dictionaries are used for many things in Python.
  - They are used for function look-up, which is done at runtime.
- Dictionaries are Hash-Table based.
  - Built-in types have hash functions provided.
  - User-defined types can specify their own hash functions.
- Example:

```
d = { 1:"one", "hi":"ho", "two":2 }  # d is a dict
x = d[1]  # x should now be "one"
if 1 in d:
    print "1 was found"
for k in d:                          # Loop over keys
    print("key:", k, "data:", d[k])
```

Tables in Various Languages
Other Languages — Perl

Perl has had Tables for a long time.

- A Perl Table implementation is called a “hash”.
  - These are Hash-Table based, of course.
  - Example:

```
$H{1} = "one";          # H is a hash
$H{"hi"} = "ho";
$H{"two"} = 2;
print $H{"hi"}, "\n";   # Prints "ho"
@A = keys %H;           # Array of keys of hash H
foreach $K (keys %H)    # Loop over keys
{
    print "key: ", $K, " data: ", $H{$K}, "\n"
}
```

Note: The syntax is different in Perl version 6.

Tables in Various Languages
Other Languages — Lisp

Lisp has had Tables for a long, long time.

- Remember that Lisp uses Binary-Tree-based lists of lists for **everything**.
  - Thus, we cannot expect Lisp to have a special Table type.
  - There are, however conventions for expressing Tables as lists.
- One kind: **property list**.
  - Constructed as "**(** *key1 data1 key2 data2 key3 data3* **)**".
  - Example: "**(1 one hi ho two 2)**".
- Another kind : **association list**.
  - Something like this: "**((** *key1 data1* **) (** *key2 data2* **))**".
    - I'm told that the implementation uses memory a bit more efficiently, replacing pointers with values.
- Of course, these do not have very efficient operations.
  - But they also predate much of the intense research on key-based look-up in the 1960's. So we'll forgive them. **Maybe.**
  - Also note that, while there is no efficient built-in Table, one can certainly write one. And people have.

The Rest of the Course
That's All …

This ends the core material of CS 311.

The Rest of the Course
From the First Day of Class: Course Overview — Topics

The following topics will be covered, *roughly* in order:

- Advanced C++
- Software Engineering Concepts
- Recursion
- Searching
- Algorithmic Efficiency
- Sorting
- Data Abstraction
- Basic Abstract Data Types & Data Structures:
  - Smart Arrays & Strings
  - Linked Lists
  - Stacks & Queues
  - Trees (various types)
  - Priority Queues
  - Tables
- **Other, as time permits: graph algorithms, external methods.**

DONE

Goal: Practical generic containers

A **container** is a data structure holding multiple items, usually all the same type.

A **generic** container is one that can hold objects of client-specified type.

The Rest of the Course
Overview

As announced on the first day of class, other topics would be covered, if time permits.

- And it does. ☺

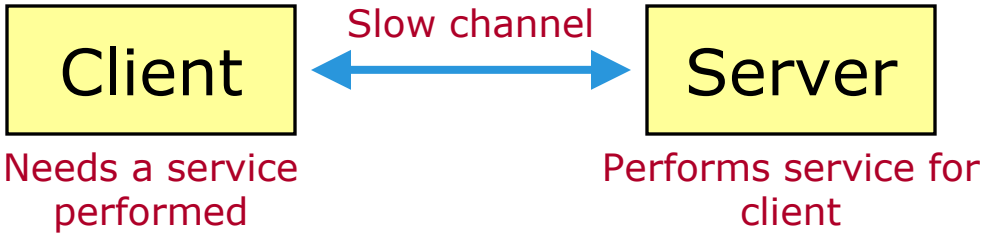
Two Topics

- External Data
  - Throughout this semester, we have dealt-with data stored in memory.
  - What if we store data on an external device, accessed via a (relatively) slow connection. How does this change the design of algorithms and data structures?
- Graph Algorithms
  - A **graph** is a way of modeling relationships between pairs of objects.
  - This is a very general notion; thus, algorithms for graphs often have very general applicability.

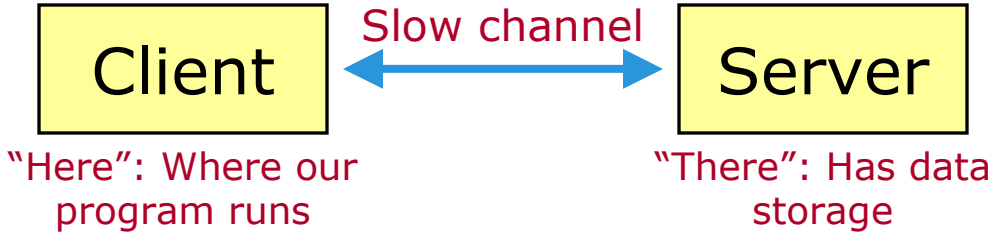
External Data
Introduction — Slow Channels

Often, we deal with computing resources joined by a relatively slow communication channel.

- It is often useful to think in terms of a **client/server** paradigm.



Now we consider **data** that are accessed via such a slow channel.



- Overriding concern: **Minimize use of the channel.**
- This has a significant impact on data structure & algorithm design.

External storage is storage that is not part of main memory.

As compared with main memory, external storage is usually:

- More permanent.
- Larger.
- Slower.

Due to slow communications with external storage, we usually access data in chunks: disk blocks, network packets, etc.

- Here, we will refer to these chunks as **blocks**.
- A key feature of block access is that, while it can be very expensive to retrieve (for example) a single data item, it is no more expensive to retrieve all other data items in the block.

In our discussion, we will:

- Do all processing on the client side of the channel.
- Usually not expect to hold all data in memory at once.
- Expect essentially unlimited storage to be available on the server.

External Data
Introduction — Two Problems to Solve

We consider two tasks dealing with data stored on a mass-storage device:

- **Sorting**
  - We have a file (essentially, an array stored externally) that we want to sort.
- **Table Implementation**
  - We want to store a very large Table externally.

In both cases, we are interested in time efficiency. In particular, we want to **minimize the number of block accesses** required for:

- Sorting.
- Table retrieve/insert/delete/traverse.

External Data Sorting

If data can be read entirely into memory, then: read, sort, write.

- But if it cannot …

We can do a reasonably efficient **Stable Merge** on data stored on an external mass-storage device.

- Stable Merge works well with **sequential-access** data.
  - Files *can* be random-access, but sequential access is a more efficient way to handle a file, since consecutive read/write operations tend to deal with the same block.
- The best Stable Merge requires **additional temporary storage**.
  - We can use **temporary files** for this.
- We write Stable Merge so that operations on data of block size or smaller all occur in memory.
  - Since we access data in order, during a single Stable Merge operation we will not read/write any single block more than once.

Using this, we get a reasonably efficient **external Merge Sort**.

- It is stable, and so we do not need more than one algorithm.

External Data
Tables — Introduction

Suppose we want to implement an external Table.

- We may assume the Table is too big to fit into memory.
- We might implement the Table as before (only stored on disk). However, this often results in too many block accesses.

Idea: Use an **index**.

Think: **index of a book**.
Not the same as the "index" of an item in a Sequence.

- Often, in a key-data pair, the data part is far larger than the key.
- Put all key-data pairs in some simple data structure stored externally (unsorted array or Linked List?). Also store the keys in a special data structure: the index. Each key gets a pointer to the proper spot in the larger structure.
  - Recall: All Table operations are constant-time for an unsorted Linked List if a "find" can somehow be done quickly.
- **If the index fits in memory**, then organize it as before, and we have a good external Table implementation.
- But if not …

External Data
Tables — Hash Table & Balanced Search Tree

Suppose the index does not fit in memory.

- We *might* still want to have an index, stored on disk.
- Regardless, the implementation options boil down to the same two as before: Hash Tables and balanced search trees.

Hash Tables

- These can be implemented similarly to the in-memory version.
- Remember that minimizing block accesses is the goal.
- If there are no collisions, then the hash function tells us exactly where an item is. We retrieve it with a single block access.
- Collision resolution is very low-cost, as long as we can still find our item **in the same block**.
  - Open addressing does not work well. Linear probing has its usual problem of cluster formation, while other probe sequences tend to put items far away, requiring multiple block accesses.
  - So: Make each block a bucket?

Balanced Search Trees

- Red-Black Trees are optimized for in-memory work.
- For external data, Red-Black Trees may require many block accesses.
- Idea: Make nodes large (one per block?). Minimize height as much as possible. Result: A **B-Tree** of high degree.

External Data
Tables — Better External Search Tree

We have already generalized 2-3 Trees to 2-3-4 Trees.

- The real reason that a 2-3-4 Tree is nice is that it has a convenient Binary-Tree representation (Red-Black Tree).
- Now we go in a different direction: **making nodes large**.

Question: Why are 2-3 Tree algorithms nice?

- Answer: Because (in the Insert algorithm) an overfull node splits exactly into 2 smallest nodes + 1 element that moves up.
- So generalize this.

2-3 Tree

- Max items = 3–1 = 2.
- Overfull (3 items) splits into 1 + 1 + 1 to move up.

Similarly, if max items = 4:

- Overfull (5 items) splits into 2 + 2 + 1 to move up.
- Call this a "3-4-5 Tree".

If max items = 6:

- Overfull (7 items) splits into 3 + 3 + 1 to move up.
- Call this a "4-5-6-7 Tree".

If max items = 8:

- Overfull (9 items) splits into 4 + 4 + 1 to move up.
- Call this a "5-6-7-8-9 Tree".

And so on …

External Data
Tables — B-Trees [1/3]

A **B-Tree of degree *m*** (*m* is odd) is essentially an (*m*+1)/2 … *m* Tree.

Terminology varies. This is what the text uses.

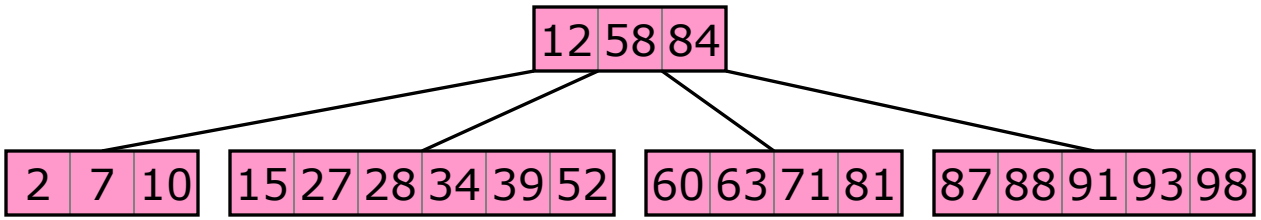
- Each node has (*m*–1)/2 up to *m*–1 items.
- Exception: The root can have 1 … *m*–1 items.
- All leaves are at the same level.
- All non-leaves have 1 more child than # of items.
- Order property holds, as for 2-3 Trees and 2-3-4 Trees.

A 2-3 Tree is precisely a B-Tree of degree 3.

- Degree = max # of children = # of items in an *overfull* node.

Below is an example of a B-Tree of degree 7.

- In practice, a B-Tree could have much higher degree than this.



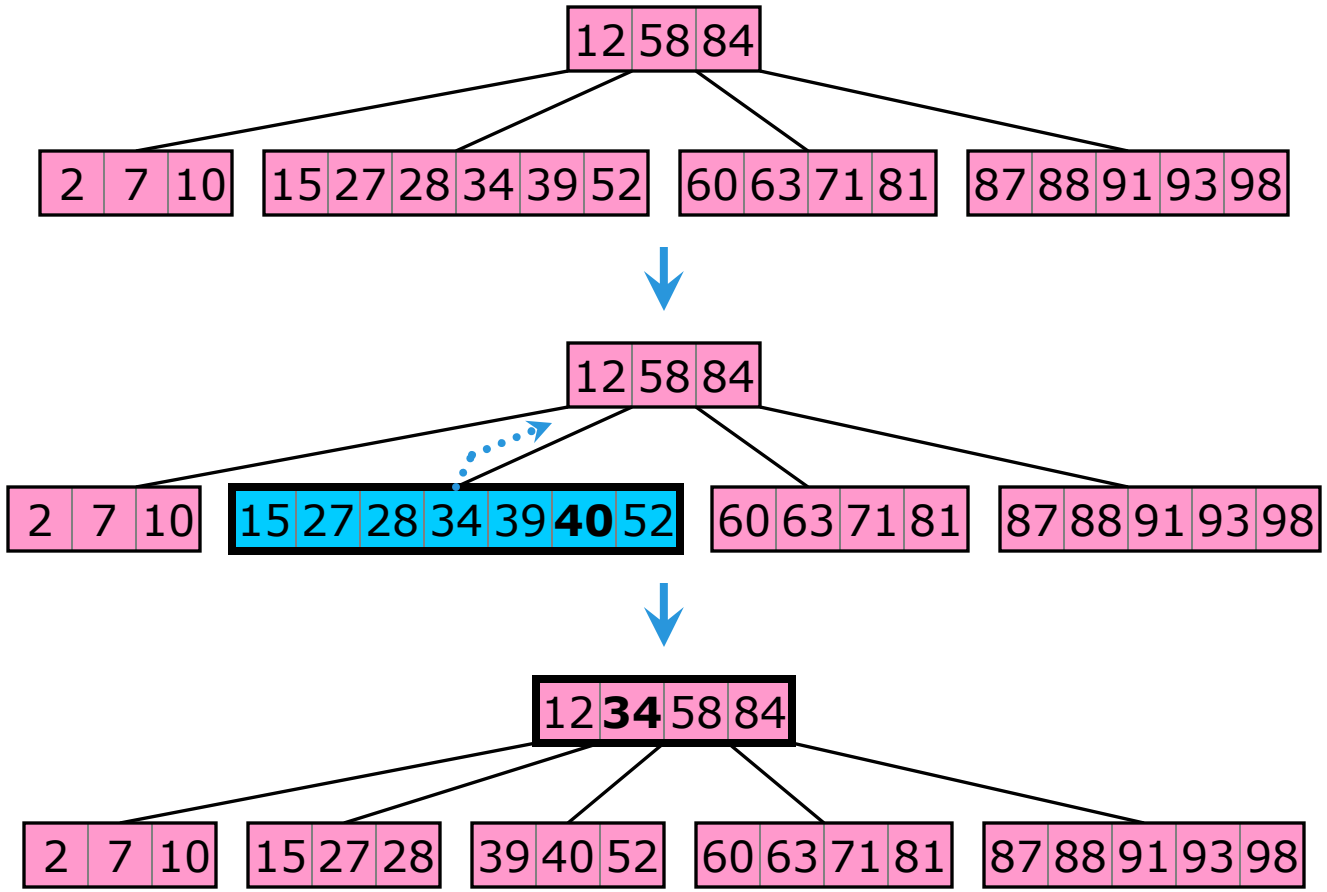
How B-Tree Algorithms Work

- Retrieve
  - Like other search trees.
- Traverse
  - Like other search trees (generalized inorder traversal).
  - Note that we need only read each block once, **if** we have in-memory storage for *h* blocks, where *h* is the height of the tree.
- Insert
  - Generalizes 2-3 Tree Insert algorithm:
    - Find the leaf that an item "should" go in.
    - Insert into this leaf.
    - If overfull, split it and move up the middle item, recursively inserting it in the parent node.
    - If the root becomes overfull, split and create a new root.
- Delete
  - Generalizes 2-3 Tree Delete algorithm.

External Data
Tables — B-Trees [3/3]

Here is an illustration of B-Tree insert.

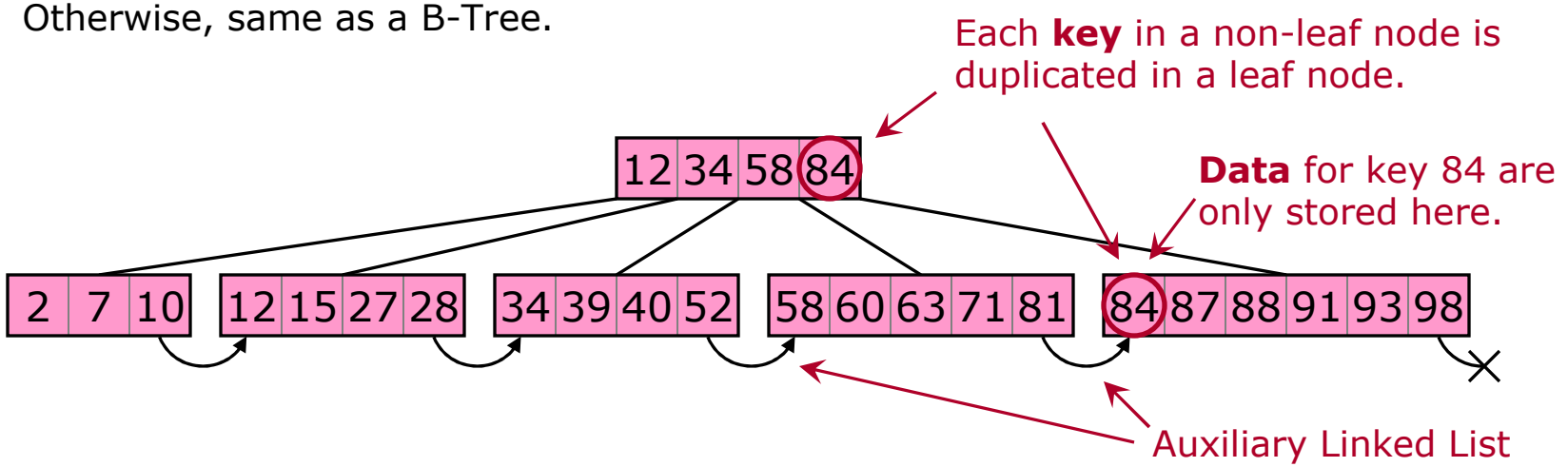
- We insert 40 into this B-Tree of degree 7.



External Data
Tables — B+ Trees

A common variation of a B-Tree is a **B+ Tree**.

- All data (the non-key portion of the value) is in the leaves.
- Keys in non-leaf nodes are duplicated in the leaves.
- Leaves are typically joined in an auxiliary Linked List. This minimizes the number of block accesses required for a traversal.
- Otherwise, same as a B-Tree.



From the Wikipedia "B+ Tree" article (4 Dec 2009):

> btrfs, NTFS, ReiserFS, NSS, XFS, and JFS filesystems all use this type of tree for metadata indexing. Relational database management systems such as IBM DB2, Informix, Microsoft SQL Server, Oracle 8, Sybase ASI, PostgreSQL, Firebird and MySQL support this type of tree for table indices. Key-value database management systems such as Tokyo Cabinet and Tokyo Tyrant support this type of tree for data access.

External Data
Reliability Issues

In practice, external storage is significantly less reliable than a computer's main memory.

Consider: What happens if the communications channel to external storage device fails in the middle of some algorithm?

- The data on the device may be left in an intermediate state.

How can we take this into account when designing algorithms that deal with data on external storage?

- As much as possible, the intermediate state of data should be either:
  - A **valid** state,
  - Or, if that is not possible, a state that can easily **fixed** (made valid).

In particular:

- **When writing the equivalent of a pointer to data on an external storage, write the data first, then the pointer.**