Tables in the C++ STL & Elsewhere continued
The Rest of the Course
External Data

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, November 23, 2020

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Some material contributed by Chris Hartman

Review

Where Are We? — The Big Challenge

Our problem for most of the rest of the semester:

- Store: A collection of data items, all of the same type.
- Operations:
  - Access items [single item: retrieve/find, all items: traverse].
  - Add new item [insert].
  - Eliminate existing item [delete].
- Time & space efficiency are desirable.

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

In a **generic container**, client code can specify the value type.

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 self-balancing search trees

  } Idea #2: Keep a tree balanced

- ✓ Hash Tables

  } Idea #3: Magic functions

- ✓ Prefix Trees ← A special-purpose implementation: "the Radix Sort of Table implementations"
- (part) Tables in the C++ STL & Elsewhere

Overview of Advanced Table Implementations

We cover the following advanced Table implementations.

- Self-balancing search trees
  - To make things easier, allow more 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 balance up and down, for a strongly balanced Binary Search Tree again:
    - ✓ **AVL Tree**

  Idea #2: Keep a tree balanced

- Alternatively, forget about trees entirely:
  - ✓ **Hash Table**

  Idea #3: Magic functions

- Finally, "the Radix Sort of Table implementations":
  - ✓ **Prefix Tree**

DONE

Later, we will cover other self-balancing search trees: B-Trees, B+ Trees.

Tables in the C++ STL & Elsewhere — Overview

Table Implementations in Practice

- C++ STL
  - ✓ Set: `std::set`
  - ✓ Key-value structure: `std::map`
  - ✓ Hash Table versions: `std::unordered_set`, `std::unordered_map`
  - ✓ Tables allowing duplicate keys
- Other Programming Languages
  - Python
  - Perl
  - JavaScript
- Other Data Structures
  - Hash Trees
  - Skip Lists

Tables in the C++ STL & Elsewhere — std::set, std::map

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

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

Two STL Table implementations

- `set`: item is just a key. `map`: item is a key-value pair.
- Duplicate keys not allowed.
- Implementation: self-balancing search tree.

Operations

- Table Insert: member `insert`, takes item.
- Table Delete: member `erase`, takes key or iterator.
- Table Retrieve: member `find` or `count`, each takes key.
- Traverse: range-based for-loop. Keys accessed in sorted order.
- `map` has bracket op (`m[`*`key`*`] = `*`val`*`;`). I rarely use `insert`, `find`.
  - The bracket operator calls `insert`, so there is no const version. Do not use it to test whether a key lies in the `map`. Use member function `count`.

The same for `set`, but different for `map`.

Tables in the C++ STL & Elsewhere — Hash Tables

`std::unordered_set<`*`valuetype`*`> us;`

`std::unordered_map<`*`keytype`*`, `*`datatype`*`> um;`

Like `set`, `map`, respectively, but Hash-Table based.

- Efficiency issues are as for Hash Tables.
- When traversing, keys do not appear in sorted order.
- Requirements on key types are different.
- We can specify a custom hash function and equality comparison.

Tables in the C++ STL & Elsewhere — Tables Allowing Dup. Keys

```
std::multiset<valuetype> ms;
std::multimap<keytype, datatype> mm;
std::unordered_multiset<valuetype> ums;
std::unordered_multimap<keytype, datatype> umm;
```

Like `set`, `map`, `unordered_set`, and `unordered_map`, respectively, but allowing duplicate keys.

- The `count` member functions may return values greater than `1`.
- `multimap` & `unordered_multimap` have no `operator[]`.
- When using these containers, we often deal with a range of items having equivalent/equal keys.

Tables in the C++ STL & Elsewhere

continued

Tables in the C++ STL & Elsewhere
Other Programming Languages — Python [1/2]

Python has several standard Table types. The main two:

- **Dictionary**: `dict`. Hash Table of key-value pairs.
- **Set**: `set`. Hash Table of keys.

```
dd = { 1:"one", "hi":"ho", "two":2 }  # dd is a dict
x = dd[1]  # x should now be "one"
if 1 in dd:
    print("1 was found")
for k in dd:                          # Loop over keys
    print("Key:", k, "value:", dd[k])
ss = { 34, "hello" }                  # ss is a set
```

Note that different key types can be included in a single Table.

Dictionaries are used for *many* things in Python, including function & member look-up, which occurs at runtime.

Tables in the C++ STL & Elsewhere
Other Programming Languages — Python [2/2]

Several Python implementations exist. The standard: **CPython**.

- The CPython built-in Hash Tables use closed hashing.
- The array size is a power of 2. The load factor is kept under 2/3.
- The probe sequence is illustrated by the following C code.

Simplified version of part of the source code for CPython 3.10.0a2, file `dictobject.c`.

```
size_t hash_code, array_size;  // Hash code, # of slots

size_t perturb = hash_code;
size_t i = hash_code % array_size;  // A slot number
while (probe(i)) // Probe @ i; true: different key found
{
    i = (5*i + 1 + perturb) % array_size;
    perturb >>= 5;
}
// Now i is the slot where the key is (to be) stored.
```

Tables in the C++ STL & Elsewhere
Other Programming Languages — Perl

A Perl Table implementation is called a **hash**.

- These are Hash Tables. The implementation uses open hashing.
  - One can optionally switch to a Red-Black Tree implementation.
- Once again, different key types can be included in a single Table.

```
$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"
}
```

This is Perl 5.
Perl 6 (a.k.a. *Raku*) uses different syntax.

Tables in the C++ STL & Elsewhere
Other Programming Languages — JavaScript

In JavaScript an object *is* a Hash Table.

- Keys are strings. Numbers may be used as keys, but they are converted to strings. Associated values may be of any type.
- Implementations vary, as each web browser has its own.
- Different associated-value types may be included in a single Table.

```
var ob = { 1:"one", "hi":"ho", "two":2 };
```

Lookup by key uses the bracket operator. When a key looks like an identifier, the dot operator may be used.

```
var a = ob["two"];  // a is 2
var b = ob.two;     // b is 2
var c = ob[1];      // c is "one"
var d = ob["1"];    // d is "one"
```

Other Data Structures — Introduction

There are a number other data structures that might be used to store a Table. We look briefly at two of these.

Hash Trees. Hash Tables and Prefix Trees combined.

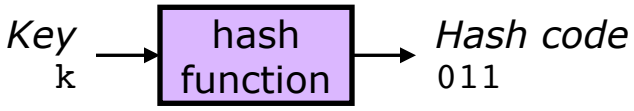
Skip Lists. A kind of binary-searchable Linked List.

Prefix Trees are rather special-purpose: they work only for keys that can be treated as *strings*.

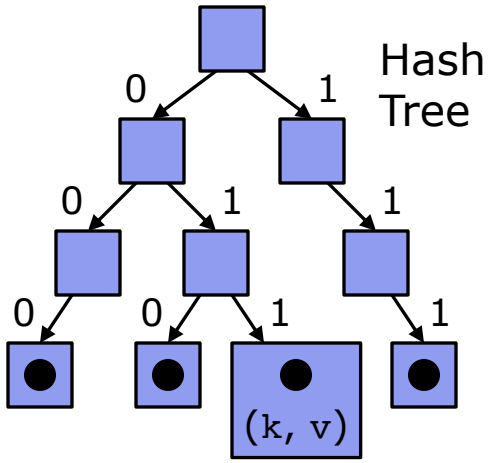
Idea. Using a hash function, hash each key. The hash code, treated as a string of bits (a **bit** is a 0 or 1), is the key for a Prefix Tree. Associated data = original key + the usual data.

This data structure is a **Hash Tree**.

Below is an illustration of how the key-value pair (`k`, `v`) would be stored in a Hash Tree. Note that different keys can have the same hash code, so a tree node must be able to store multiple key-value pairs.



This is a toy example.
In practice, hash codes are usually 32 or 64 bits.

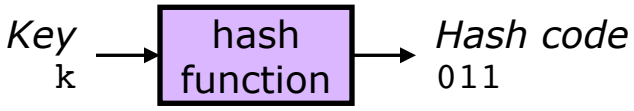


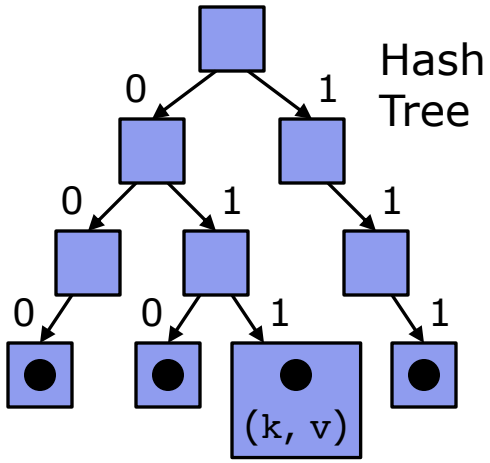
Tables in the C++ STL & Elsewhere
Other Data Structures — Hash Trees [2/2]

A number of variations on Hash Trees exist, with names like HAMT (Hash Array Mapped Trie) and CHAMP (Compressed Hash Array Mapped Prefix tree).

Hash Tree variants are used to implement *immutable* Tables in Java Virtual Machine languages Clojure (Lisp dialect) and Scala.

- **Immutable**: not modifiable.
- But we want efficient creation of a modified copy.
- In functional programming, data structures that support this are said to be **persistent**. Implementation of persistent Tables is an active research area.

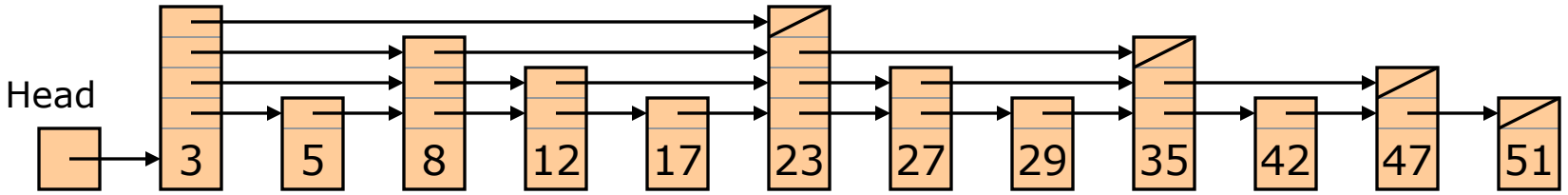




A Table stored in a sorted Linked List is SLOW.

However, an idea:

- Add a second pointer to each list node. Using these, make a secondary Liked List using *some* of the nodes in the original list.
- When searching, follow the secondary Linked List first. Once the position of the desired key is narrowed down, move to the full list.
- That might give us a small speed-up. How to get a *big* speed-up?
- Repeat. Add another pointer. Make another, even shorter, list. Etc.
- This binary-searchable Linked List is a **Skip List** [W. Pugh 1989].



Other Data Structures — Skip Lists [2/2]

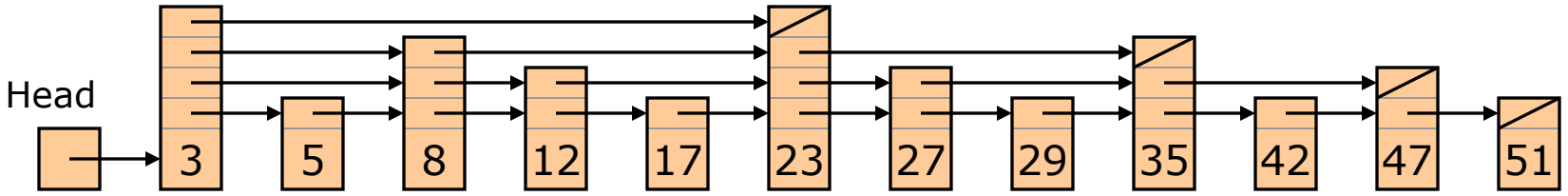
Worst case for Table insert/delete/retrieve is $\Theta(n)$.

Average case: $\Theta(\log n)$.

- Faster than self-balancing search trees!
- But—so what? We have Hash Tables.

However, Skip Lists are designed to support efficient *simultaneous* modification in different parts of the dataset.

So they may have great advantages in a multi-threaded context.



The Rest of the Course

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 kinds)
  - Priority Queues
  - Tables

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.

- **Briefly: external data, graph algorithms.**

The Rest of the Course
Overview

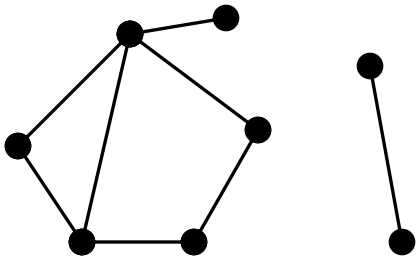
In the time remaining, we look briefly at two more topics.

External Data

- Previously, we dealt only with data stored in memory.
- Suppose, instead, that we wish to deal with data stored on an external device, accessed via a relatively slow connection and available in sizable chunks (data on a disk, for example).
- How does this affect the design of algorithms and data structures?

Graph Algorithms

- A **graph** models relationships between pairs of objects.
- This is a very general notion. Algorithms for graphs often have very general applicability.



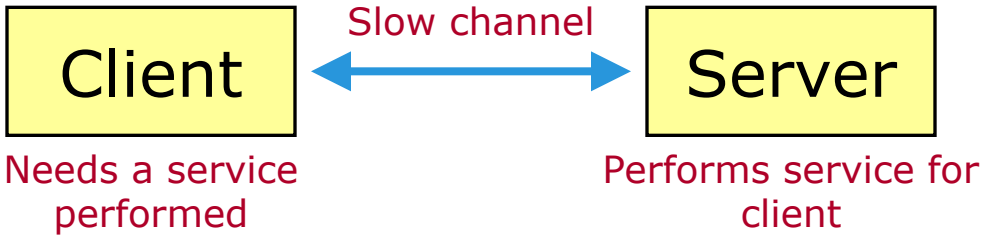
Drawing of a Graph

This usage of "graph" has nothing to do with the graph of a function. It is a different definition of the word.

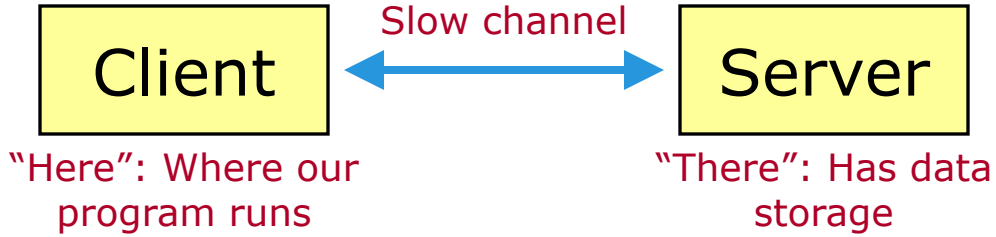
External Data

It is common for computing resources to be joined by a relatively poor (slow, perhaps unreliable) communication channel.

It can be helpful 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.

Compared with main memory, external storage is typically:

- More permanent.
- Larger.
- Much slower.

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

- We will refer to these chunks as **blocks**.
- A disk is a kind of **block-access device**.
- It can be expensive to access a data item, but there is little additional cost to access other data items stored in the same block.
- An appropriate basic operation: a single block read/write.

In our discussion, we will:

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

In practice, external storage read & write operations are significantly less reliable than those to memory.

Q. What happens if the communications channel to an external storage device fails in the middle of some algorithm?

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

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

A. The intermediate state of data should be either:

- A **valid** state,
- Or, if that is not possible, a state that can easily be **fixed**.

In particular:

When writing the equivalent of a pointer to data in external storage, write the data first, then the pointer.

Introduction — Two Tasks

Two tasks have occupied much of our time this semester. We consider these in the context of data on a mass-storage device.

Sorting. Sort a file—perhaps line by line.

Table Implementation. Store a large Table externally.

We are interested in time efficiency of sorting, as well as the various Table operations—traverse, retrieve, insert, delete. It will be helpful to change our model of computation, making our basic operation the **block access** (read or write a single block).

External Data
Sorting

We can do a reasonably efficient **Stable Merge** on two files.

- Stable Merge works well with *sequential-access* data.
  - Files *can* be random-access, but sequential access is an efficient way to handle a file as a whole, since consecutive read/write operations tend to deal with the same block.
- Recall: general-purpose Stable Merge uses additional temporary storage. We can use temporary files for this, if necessary.
- Since we access data in order, a Stable Merge operation should not read/write any single block more than once.

This idea gives us a reasonably efficient **external Merge Sort**.
Note that this will be stable.

External Data
Table Implementation — Options

Options for implementing an external Table are basically as before: Hash Tables and self-balancing search trees. But details vary.

Hash Tables

- If there are no collisions, then the hash function tells us where an item is. Retrieve it with a single block access.
- Collision resolution is cheap, if the key's spot is in that same block.
- This works well: open hashing, with each bucket stored in a small, fixed number of blocks (1 block? 2 blocks?).

Self-Balancing Search Trees

- Red-Black Trees are optimized for in-memory work. For external data, they require too many block accesses.
- Idea: Make nodes *large*. (How would this help?)

This leads to the structures we look at next: *B-Trees*.

External Data
Table Implementation — B-Trees [1/4]

We can generalize 2-3 Trees for block-access storage by **making the nodes large**. (Perhaps we store one node per block?)

Q. Think: why are 2-3 Trees *nice*?

A. An over-full node splits into 2 small nodes + 1 item to move up.

Can we make a *nice* self-balancing search tree with larger nodes?

- Consider a 2-3 Tree (max # of items in a node: 3–1 = 2).
  Over-full (3 items) splits 1 + 1 + 1 to move up.
- If max # of items in a node = 4:
  over-full (5 items) splits 2 + 2 + 1 to move up: a 3-4-5 Tree.
- If max # of items in a node = 6:
  over-full (7 items) splits 3 + 3 + 1 to move up: a 4-5-6-7 Tree.
- And so on …

These are **B-Trees**. Max # of children is the **degree** of the B-Tree.

For $m \geq 3$, a **B-Tree of degree *m*** is a ceiling(m/2) … m Tree.
[R. Bayer & E. McCreight 1970]

- A node has ceiling(m/2)–1 … m–1 items.
- Except: The root can have 1 … m–1 items.
- The order property holds, as for 2-3 Trees.
- All leaves are at the same level.
- Each node is either a leaf or has its maximum number of children.
- Degree = max # of children = # of items in an over-full node.

Why "B"?

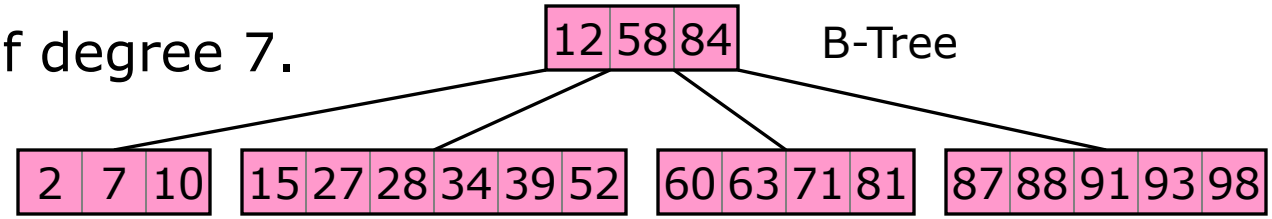
We don't know. Perhaps *balanced*, *broad*, *bushy*, *Bayer*, or *Boeing*—where Bayer & McCreight worked.

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

A B-Tree of degree 4 is a 2-3-4 Tree.

A B-Tree of degree 5 is a 3-4-5 Tree.

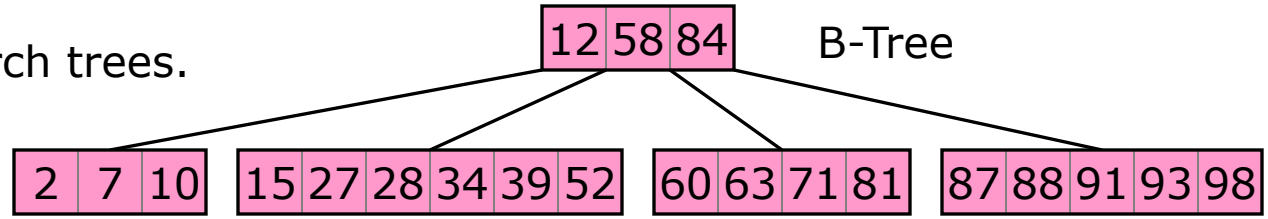
Shown is a B-Tree of degree 7.



In practice, the degree may be much higher (for example, 50).

How B-Tree Algorithms Work

- Traverse
  - Like other search trees (generalize inorder traversal).
  - If we have in-memory storage for *h* nodes, where *h* is the height of the tree, then we only need to read each node once.
- Retrieve
  - Like other search trees.

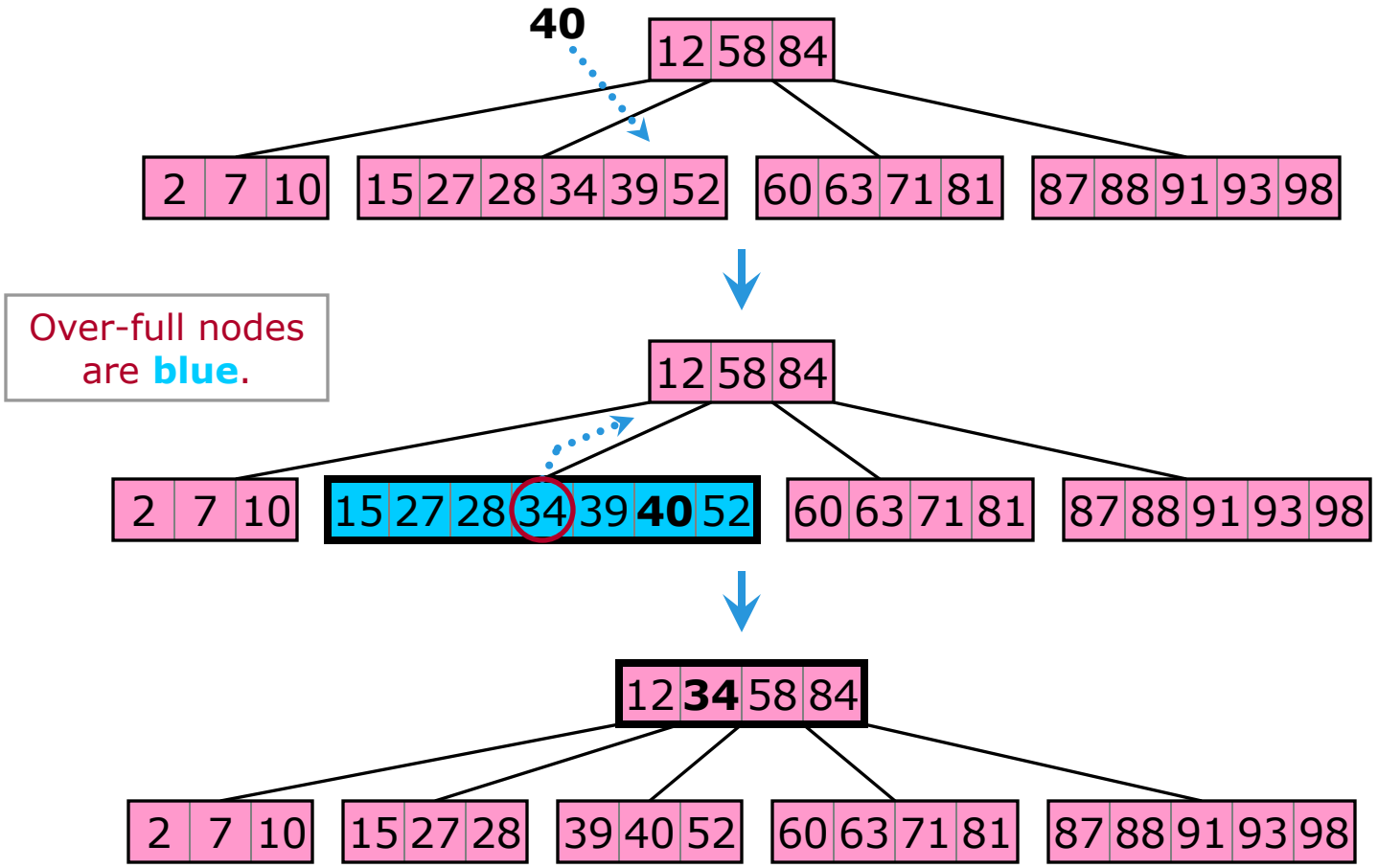


- Insert
  - Generalizes 2-3 Tree Insert algorithm:
    - Find the leaf that an item *should* go in.
    - Insert into this leaf.
    - If over-full, then split the node, with **middle** item moving up. Recursively insert this item into the parent node.
    - If the root becomes over-full, then split and create a new root.
- Delete
  - Generalizes 2-3 Tree Delete algorithm. We will not cover the details.

Illustration of B-Tree insert.

- Insert 40 into this B-Tree of degree 7.

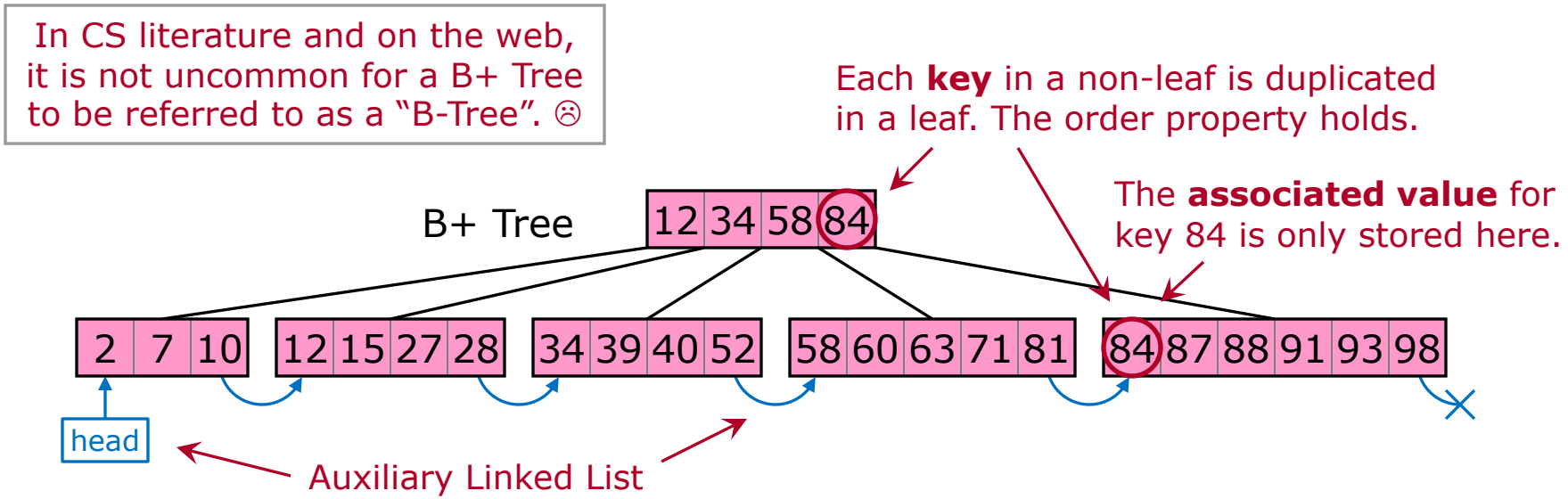
An **over-full** node has 7 items.



External Data
Table Implementation — B+ Trees [1/2]

There are a number of B-Tree variations. Probably the most used are **B+ Trees**. These are just like B-Trees, except:

- Keys in non-leaf nodes are duplicated in the leaves, while maintaining the order property.
- Associated values are stored only in the leaves.
- Leaves are joined into an auxiliary Linked List. This minimizes the number of block accesses required for a traversal.



External Data
Table Implementation — B+ Trees [2/2]

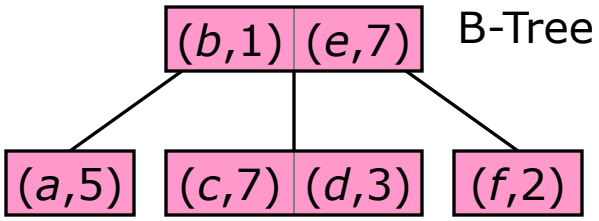
To fully illustrate the differences between B-Trees and B+ Trees, we picture two such trees holding the same dataset, with both keys and associated values shown.

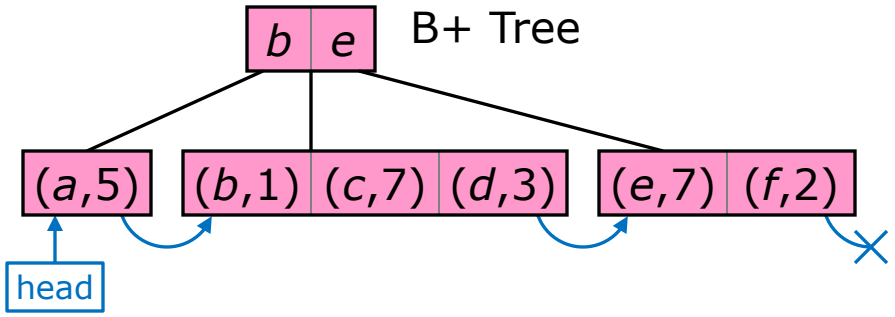
To the right is a Table dataset.

Below-left is a B-Tree holding this dataset.

Below-right is the corresponding B+ Tree.

Key	Associated Value
a	5
b	1
c	7
d	3
e	7
f	2





External Data
Table Implementation — Notes

For each of the covered external Table implementations, order of operations is the same as for the in-memory versions.

In particular, for a B-Tree or a B+ Tree, retrieve/insert/delete by key are Θ(log *n*), and traverse is Θ(*n*).

Modern filesystems typically involve a B-Tree or variant internally.

- Only major exceptions that I know of: Microsoft's FAT filesystems.
- The B+ Tree is a particularly common variant.

These trees are also used in relational-database implementation.