Hash Tables

CS 311 Data Structures and Algorithms
Lecture Slides
Wednesday, November 25, 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
- Tables in various languages

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]

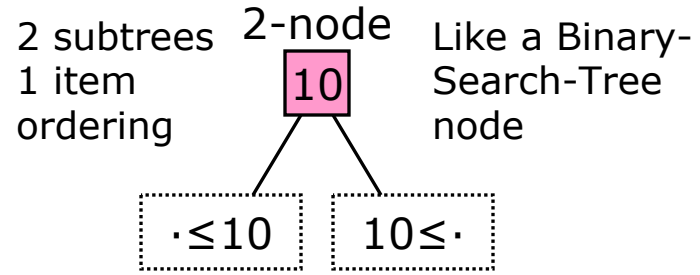
A Binary-Search-Tree style node is a **2-node**.

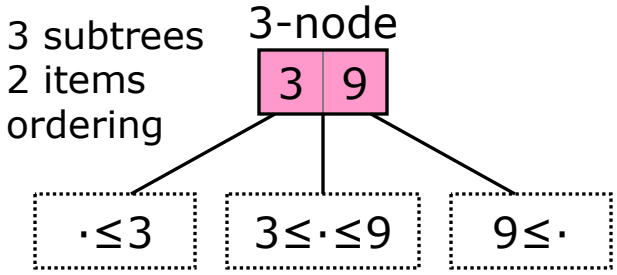
- This is a node with 2 subtrees and 1 data item.
- The item's value lies between the values in the two subtrees.

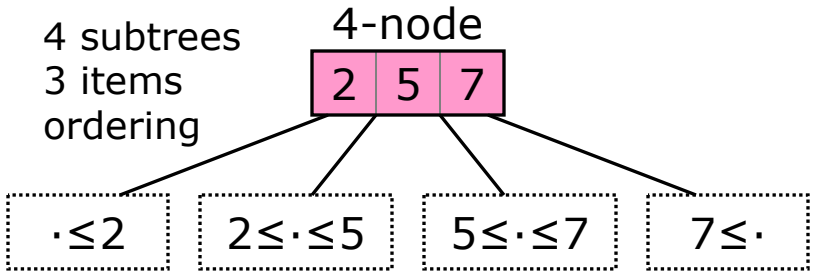
In a "2-3 Tree" we also allow a node to be a **3-node**.

- 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**.



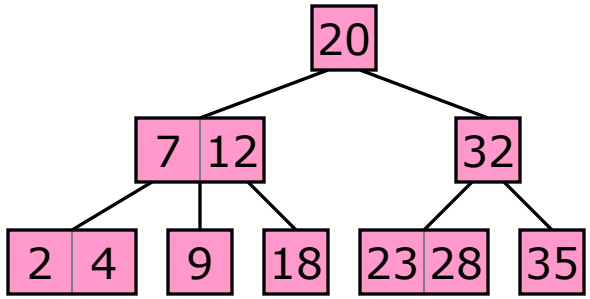




Review
2-3 Trees [2/4]

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.

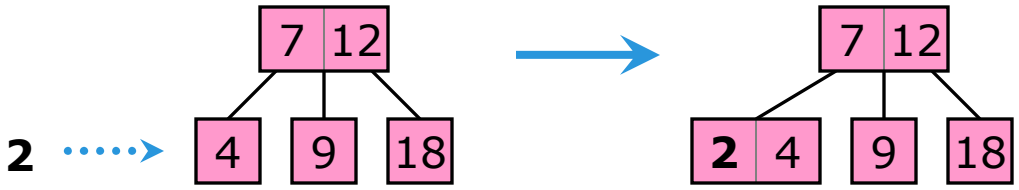
To **traverse** a 2-3 Tree:

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

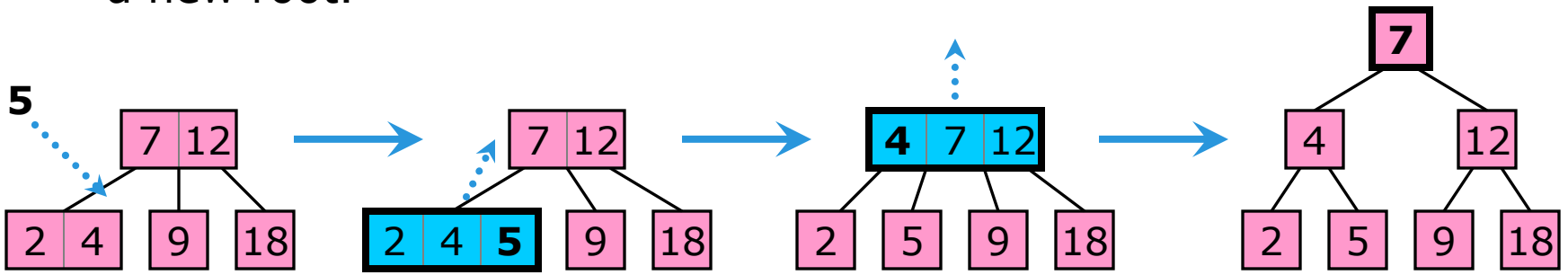
2-3 Trees [3/4]

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]

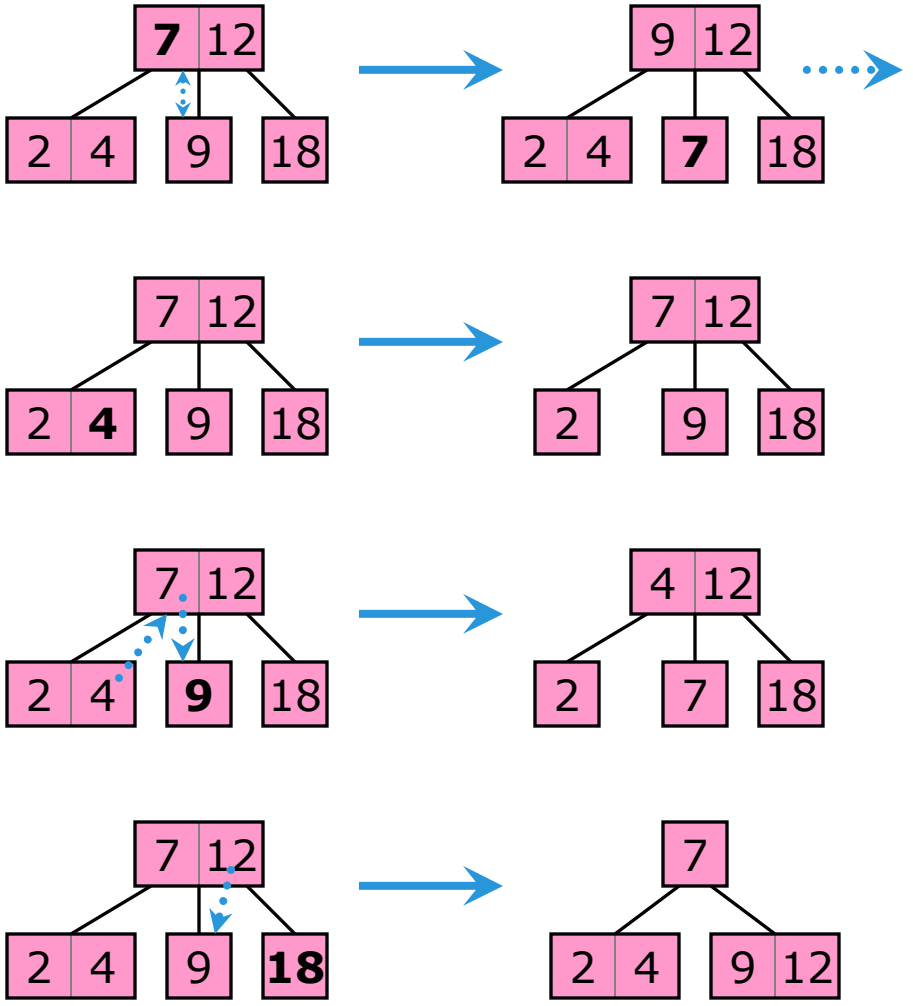
To **delete** in a 2-3 Tree:

- 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:

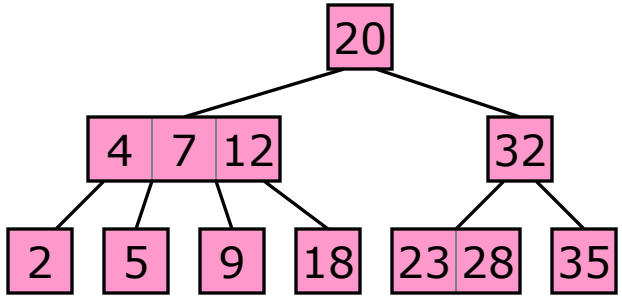
- **Easy Case**: If the item is in a node with another item, simply remove it.
- **Semi-Easy Case**: Otherwise, if the node has a consecutive sibling with two items, do a rotation with the parent.
- **Hard Case**: Otherwise, bring the parent down, combining it with a consecutive sibling.
  - Use recursive 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 [1/4]

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



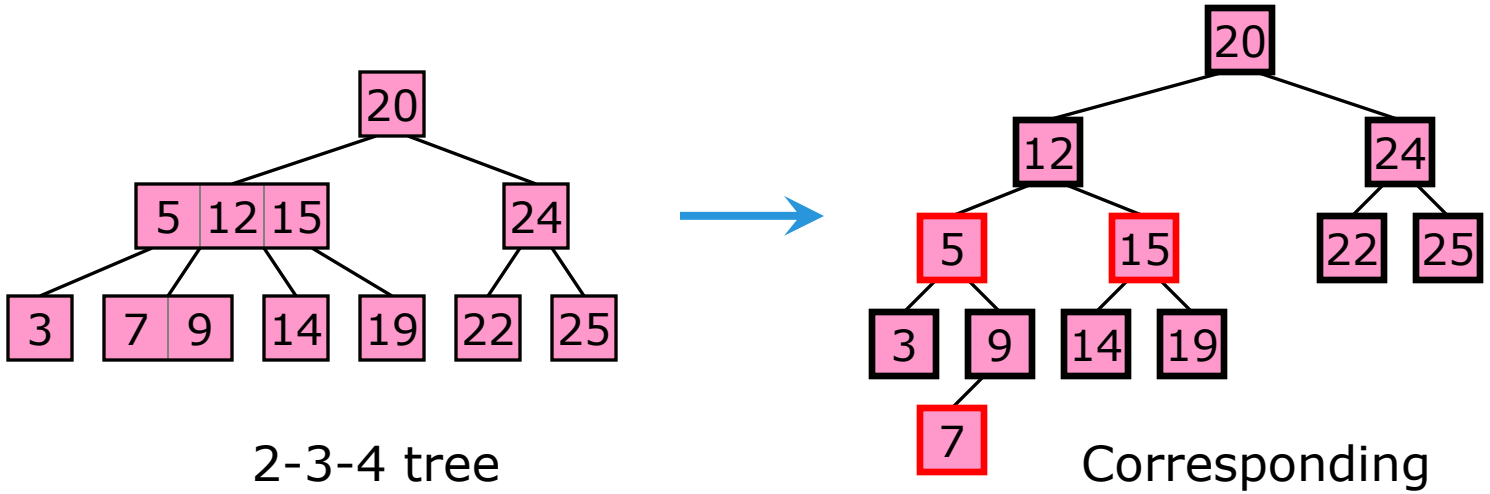
The insert and delete algorithms are not terribly different from those of a 2-3 Tree.

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

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**.



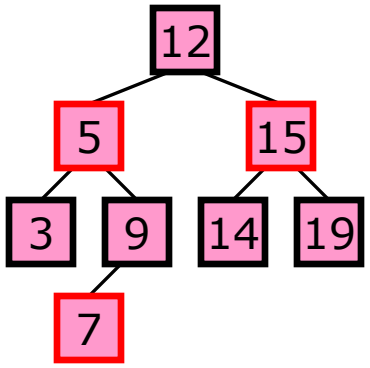
2-3-4 tree

Corresponding Red-Black Tree

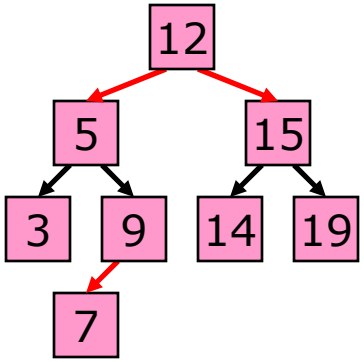
Other Balanced Search Trees [3/4]

Implementations of Red-Black Trees vary.

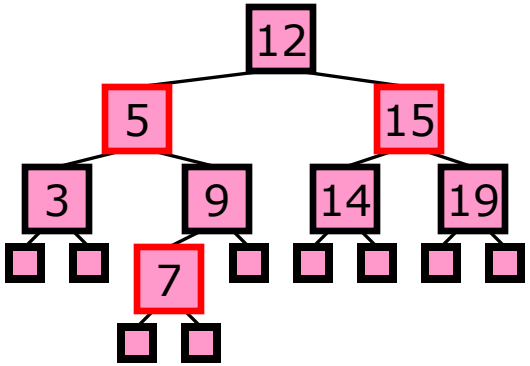
- I have presented them as having red and black **nodes**.



- 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.

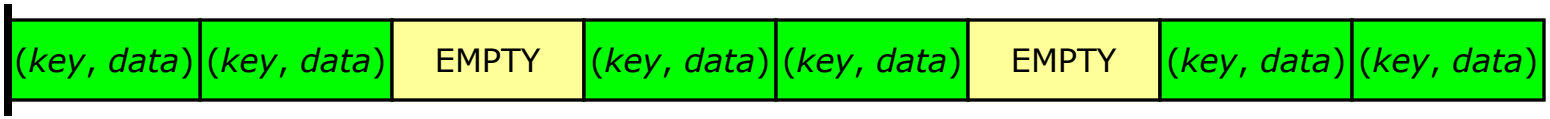
Hash Tables
Introduction [1/4]

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

- Kind of. It depends what we mean by "more efficient".

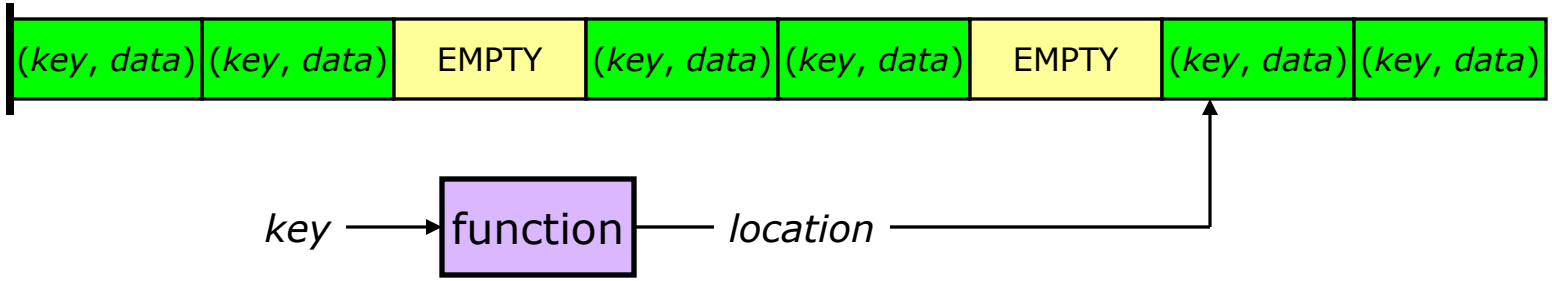
Idea: implement a Table as an unsorted array of key-data pairs.

- Delete-by-index is fast, if we allow **gaps** in our data.
  - To delete an item at a given index, set it to "EMPTY". Constant time.
  - However, to do *Table* delete (by key) we must find (retrieve) the item …
- Insert is fast, if we allow duplicate keys.
  - If no reallocation, constant time.
  - However, to insert when we do not allow duplicate keys, we need to find (retrieve) the item, so we can replace it. And, unfortunately …
- Retrieve is **slow**.
  - $O(n)$, in fact (Sequential Search).



Thus, **speeding up retrieve** might make *everything* fast …

Hash Tables
Introduction [2/4]



What if we had a magic function which, given a key, returned the index of the item?

- Then all three operations (insert, delete, retrieve) would be constant time!
  - Okay, maybe *amortized* constant time for insert.

Unfortunately, such a function is generally impractical. Given reasonable constraints it is often **impossible**.

- Think: What if there are more possible keys than array locations?

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 (key).
- 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.

Thus, a hash function does not need to be magic, only **consistent**.

- The reason it "knows" where an item is stored in the array, is that we asked the function where to put the item when it was inserted.
- As long as the function is **deterministic** (always gives same output for same input) it will work.

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:

- For a Hash Table to be worth using, it needs to have fewer locations in it than the number of possible keys.
  - Otherwise, just use an array indexed by key.
  - Typical key sets are large, however. Consider the number of possible 20-character strings (with the printable ASCII set, about 4.4×10^{39}).
- Thus, items with different keys *may* have the same hashed value.
- When this happens, we have a **collision**. Dealing with it is called **collision resolution**.

Later we will discuss various collision-resolution methods.

In general:

- Collisions are not a big problem as long as they are rare.
- But we cannot *guarantee* that they are rare.
- In fact, it is possible that, for *every* item we insert, the hash function has the same output. (Ick.)

Hash Tables
Good Hash Functions [1/4]

The problem of collisions is why we need hash functions to "mess up" their input, so that the output has no resemblance to the input.

- While patterns in the input are common, these should not lead to patterns in the output.
- An ideal hash function spreads the keys around, so that collisions are rare.

A hash function **must**:

- Take a valid key and return a nonnegative integer.
- Be **deterministic**.
  - Its value depends only on its input (the key). Using the same input multiple times results in the same output each time.

A **good** hash function:

- Can be computed quickly.
- Spreads out its results evenly over the possible output values.
- Turns patterns in its input into random-looking output.
  - For example, consecutive keys should generally not produce consecutive output values.

Hash Tables
Good Hash Functions [2/4]

The output of a hash function should be an integer. What if the input is not an integer?

- Strings can be converted to integers character by character. After that, the problem is similar to that of an integer with many digits.

What about client-specified key types?

- We cannot put client-specified key types into a Hash Table unless an appropriate hash function is also provided.
- Note: In a key-data pair, the key is the one we send to the hash function, and so client-specified data types are no problem at all.

What about different key types, using different hash functions, in the same Hash Table?

- No problem.
  - Assuming we can figure out how to get different types into one array. This is easy in (say) Python, and a bit tricky (but possible) in C++.
- On the other hand, we usually cannot **compare** different types.
- Thus, a Hash Table is often an appropriate implementation when multiple key types are to be stored in the same Table, while a balanced search tree might not work as well.

Hash Tables
Good Hash Functions [3/4]

Now we look at a few ideas involved in good hash functions. The actual design of a top-notch hash function is beyond the scope of this class. (If you need one, lots of good ones are freely available.)

Given an integer with many digits, one approach to hash function design is to form some function of the digits.

- For example, add them up.
- Better yet, do
  $1 \times (1^{st}\text{ digit}) + 2 \times (2^{nd}\text{ digit}) + \ldots$
- Bitwise operators (shift, xor, etc.) can be helpful, too.

Typically, client-provided hash functions output a large-ish integer (32-bit, say).

- If a Hash-Table implementation needs smaller values, then it can send the large values through its own function.

Hash Tables
Good Hash Functions [4/4]

One way to help get the evenly-spread-out property is to use modular arithmetic and give the Hash Table a prime number of locations.

- A prime number is an integer greater than 1 that is only divisible by itself and 1. The first few are 2, 3, 5, 7, …. Primes used as sizes of Hash Tables will be much larger.

```
int hash_func(const key_type & k)
{
   const int table_size = …; // Hash Table size
                             // (a prime number)
   unsigned long value = client_hash_func(k); // may be large
   return value % table_size;
}
```

However, some kinds of hash functions allow for other Table sizes.

- Powers of 2?

Recall that a **collision** is when the hash function produces the same output for different keys.

- We cannot guarantee that collisions will be rare.

How collisions are resolved is the primary design decision involved in a Hash Table.

- Different collision-resolution methods result in different Hash-Table implementations.

Two categories of collision-resolution methods:

- Open Addressing
  - The Hash Table is essentially an array of data items. Thus, each location can store **one data item**.
  - If we get a collision, we look for another spot.
- "Buckets"
  - Each location in the array is a data structure capable of storing **multiple data items**.
  - In this case, a location in the array is called a **bucket**.

Collision Resolution — Open Addressing [1/4]

In **open addressing**:

- The Hash Table is essentially an array of data items.
- Each location can be marked as "**empty**".

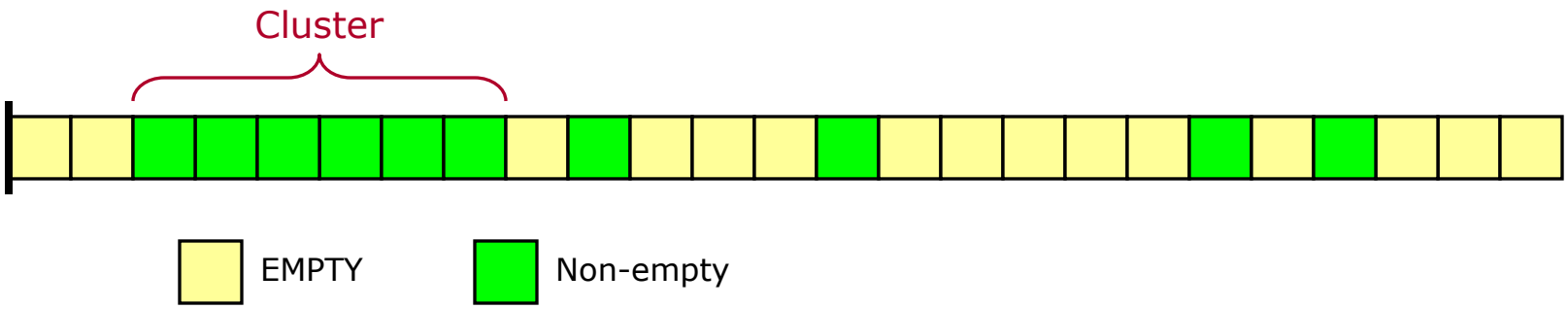
When inserting or retrieving (including the retrieve done as part of a delete), we look at a sequence of locations.

- The first is the location given by the hash function.
- We continue looking until we find the given key or we are sure it is not present.
- Each time we view a location, we are doing a **probe**. The entire sequence of locations to view is called the **probe sequence**.

Collision Resolution — Open Addressing [2/4]

The simplest probe sequence is the one in which we look at location t, then t+1, then t+2, etc. This is **linear probing**.

- Linear probing tends to form **clusters**, which slow things down.



To avoid clusters, we can use **quadratic probing**, which has the probe sequence t, $t+1^2$, $t+2^2$, $t+3^2$, etc.

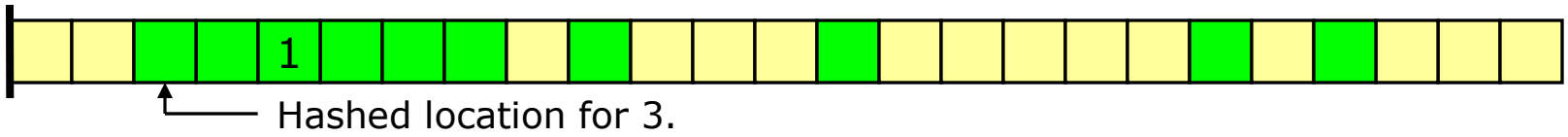
Collision Resolution — Open Addressing [3/4]

Fancier techniques involve using a secondary hash function when a collision occurs.

- These generally go under the name of **double hashing**.
- For example, do a variation of linear probing with a step size other than 1. The step size is given by the second hash function.
- Or, just use the second hash function to give a second location, after which one of the simpler probe sequences is used.

Hash Tables
Collision Resolution — Open Addressing [4/4]

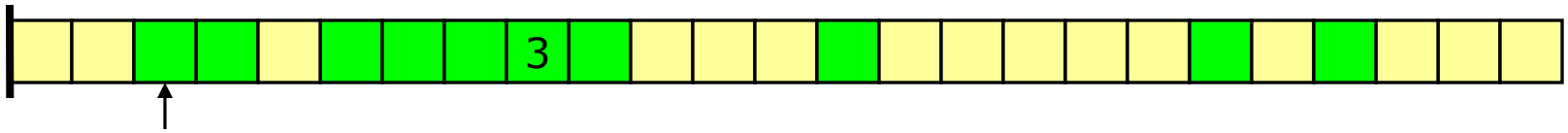
Open addressing leads to a problem: How to be sure a key is **not** present?



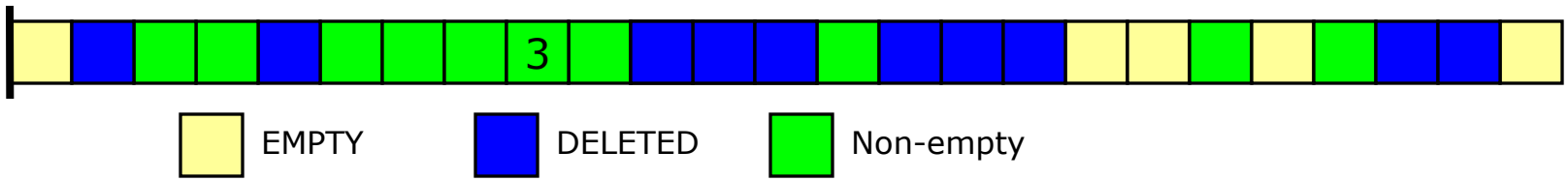
In the Hash Table above, retrieve 3 (not present). Use linear probing.

- Begin at the arrow, look until we find an "empty", return NOT FOUND.

Now insert 3 and then delete 1.



Again retrieve 3. Using the above strategy, we return NOT FOUND. ☹
Solution: Allow **DELETED** marks. Stop when we find given key or EMPTY.



But now, the array can fill up with "deleted" marks, which slows it down.

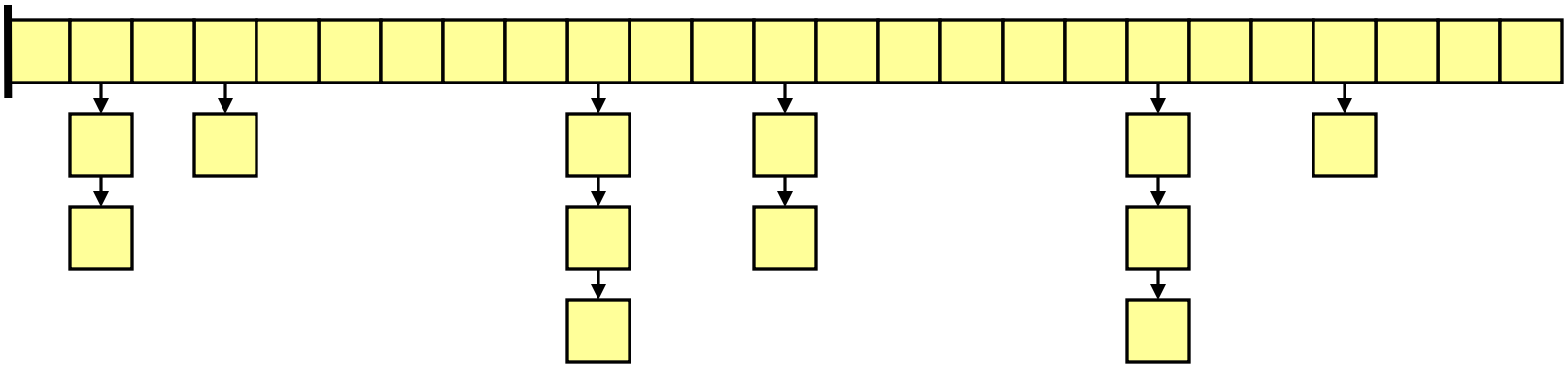
Hash Tables
Collision Resolution — "Buckets" [1/2]

Another collision-resolution idea is to make the Hash Table an array of data structures.

- Each structure can hold multiple data items.
- We call the locations in the Hash Table "buckets".

Very common: Make each bucket a Linked List. This is called **separate chaining**.

- Why do we *not* need (or even want) a Doubly Linked List?

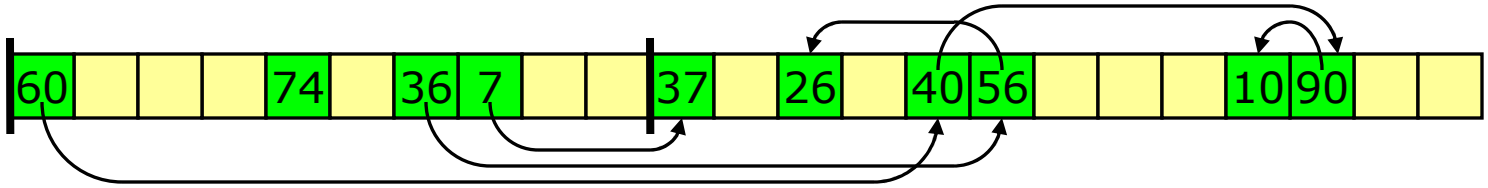


Why not make each bucket a Red-Black Tree?

Hash Tables
Collision Resolution — "Buckets" [2/2]

Idea: use an array-based Linked List for each bucket.

- The Hash Table can be a big array divided into two sections:
  - One section for the heads of the buckets.
    - This section is indexed using the output of the hash function.
  - One section for the rest of the nodes in the buckets.
- Each node is an array element.
- Pointers are replaced by array indices.
- Efficiency is much the same as for a pointer-based Linked List.
  - But memory-management overhead is reduced.



Hash Tables
Table-Remake

Sometimes it is necessary to remake the Hash Table.

- The Hash Table may fill up, requiring a larger array.
- All implementations have performance degradation as the number of data items rises.

In these cases, we need to do a reallocate-and-copy, as we did with smart arrays.

Here, however, the copy can be very time-consuming.

- We need to traverse the entire table, possibly including empty locations.
- We need to call the hash function for every key present.

This is one of the downsides of Hash Tables.

Hash Tables
TO BE CONTINUED …

Hash Tables will be continued next time.