# CSCI 2270

## Data Structures & Algorithms

Gabe Johnson

Lecture 19 Mar 1, 2013

# Standard Containers

# Lecture Goals

1. B-Tree: Impending Doom?
2. Container Types
3. Hash Functions

# Upcoming Homework Assignment

HW #5 **Due: Friday, Mar 1**

## B-Trees

B-Tree Assignment is due tonight if you want to get more than 15 points. After tonight you have the rest of the semester to get up to 15 points. RG unit tests are up for C++ and Java. You can still turn in Python code to me via email, and I'll score it manually.

# Last Minute B-Tree Help?

23 of you have more than 15 points already.

Python unit tests are still but an aspiration. If you are doing the Python version I will check it manually. I will put one up eventually, but for now I need to let my wrists take a break before I hurt myself.

*(carpal tunnel syndrome sucks. don't get it.)*

# Std. Containers and ADTs

- **Collection:** Generic word for any grouping of data
- **List:** Sequence supporting random access
- **Map:** Associates keys with values, access via key
- **Set:** Unordered group of unique objects
- **Bag (Multiset):** Like Set but duplicates allowed
- **Queue:** List w/ First In, First Out (FIFO) semantics
- **Stack:** List w/ Last In, First Out (LIFO) semantics

# Collection

A ‘Collection’ is a very generic word for any grouping of data. There are many *kinds* of collections, like lists, vectors, sets, queues, etc. You could even consider a struct or an array to be a collection.

# Collections May Be Typed

We may be able to restrict the collection to only allow items a given data type. Depends on the language.

In **C++** we *must* create a vector of items with the same type: e.g. integers, or strings.

In **Java** we can be specific, or allow a generic group.

In **Python** we *can't* create a list of items whose types are specified.

# Common Collection Methods

There is no central committee that decides which operations all collections must have. It varies by language. But in general, you can at least ***iterate*** over collections. Even in collection types that do not enforce ordering, you can access them in an order (that order may not be dependable among incarnations).

# Common Collection Methods

Operations that are usually (not always) available:

- **Iterate** over elements of the container
- Get the **size** of the container
- What's the **first item**?
- What's the **last item**?
- Is the container **empty**?
- **Copy** the container
- Does the container represent the **same data** as another container?
- Does it currently **contain** a given value?

# List

- Sequential Data
- Random Access via numeric index
- May support sorting
  - On insert
  - Or on demand

C++ vector<int>
Java ArrayList
Python Lists

Lists are sometimes called *dynamic arrays* which is a very fancy term. It means you don't have to worry about how the memory is laid out (nor can you depend on it working like an array).

# Map

A Map can be called a *hash*, an *associative array*, or a *dictionary*.

- Contains key / value pairs
- Access values using *key* rather than numeric index
- May support sorting
- By key
- By value
- Size refers to number of key / value pairs
- Keys are unique; pairs are not. See ‘set’.

# Uses of Maps

You can use a Map whenever you have a dynamic set of things (e.g. you don't know what they will be) but you need to relate them to values. Examples include:

- IP address to domain names;
- Usernames to password hashes;
- Colleges to mascots;

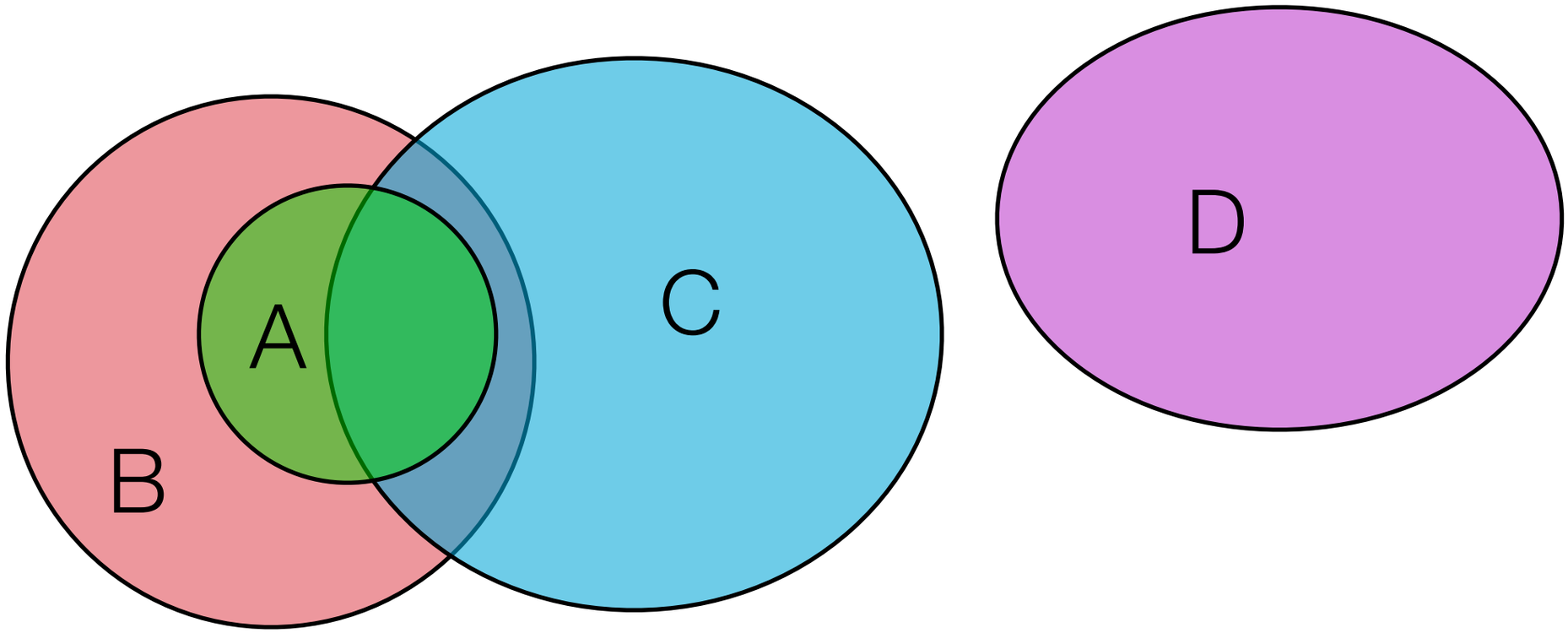
# Set

A Set contains *unique* items. While a List may contain [ 5, 8, 8, 5, 10 ], a set may not have redundant values. So if we convert that list to a set we are left with (5, 10, 8).

Sets are egalitarian: they are typically *unordered*. There is no dependable way to determine which one is first or last.

Some variants allow you to enforce a sort order when you iterate over the values.

# Sets Allow Interesting Math




Sets let us add, subtract, intersect, and exclude using *set semantics*. What's the intersection of A and C? What's the combined set of B and C?

# Sets and Databases

If I search for users in the RetroGrade database, I get a Set. If I look for everyone named ‘Nick’, I get five people. Some are in CS 1300, others are in 2270. So I can do set arithmetic to get the Nicks who are in one class:

students_in_2270 = some database query
dudes_named_nick = another query
nicks_in_2270 =
    students_in_2270 *intersected with* dudes_named_nick

# Bag (Multiset)

A *Bag* is an extended form of Set that removes the uniqueness constraint. Bags can contain multiple entries. There's two ways of going about this:

- Only store one of them in memory, and then retain a count of how many there are, or

- Store all of them in memory

# Removing from Bags

If I have a Bag with [5, 8, 8, 10, 5], and I want to remove 5, what should the container look like afterwards?

Two possible semantics:

- Remove *all* the fives
- Remove *one* five.

# Queue

We talked about queues last time, but I didn't mention the sacred text: Queues are *First In, First Out.* The first person in line is the first person to leave the line. This works just like waiting in line at the bank.

Remember that phrase: FIFO = First In First Out. You'll see it everywhere.

# Queue Quirks

A bare-bones queue doesn't need to give operations like random access by key or index. It doesn't even need to give the size.

All a Queue is required to do is remember which order things were put in so you can remove them in that order.

# Queue Quirks

There are special words for adding and removing from queues. I don't know why.

**Enqueue** is the fancy word for adding to a queue.

**Dequeue** is the fancy word for removing from a queue.

This was probably decided by the same person who thought it would be a neat idea to have nineteen different words for 'loan'.

# Stack

The inverse of a Queue (if there is such a thing) is a Stack. The canonical example is a stack of plates. You ***push*** a new plate onto the stack. When you need a plate you ***pop*** it off the stack.

This means that a Stack has *Last In, First Out* semantics, often abbreviated LIFO.

I don’t know why it isn’t First In, Last Out (FILO).

# Stacks are Very Useful

Stacks have many, many applications. Your C++ compiler uses stacks to parse your source files. When a program is running, it uses a stack to record what it was doing when you call a function (e.g. so it can return to the right place).

If you search a binary tree and you recursively look in a child node, it pushes state onto a stack. When that recursive call returns, it pops the stack.

# Stacks Beyond Programs

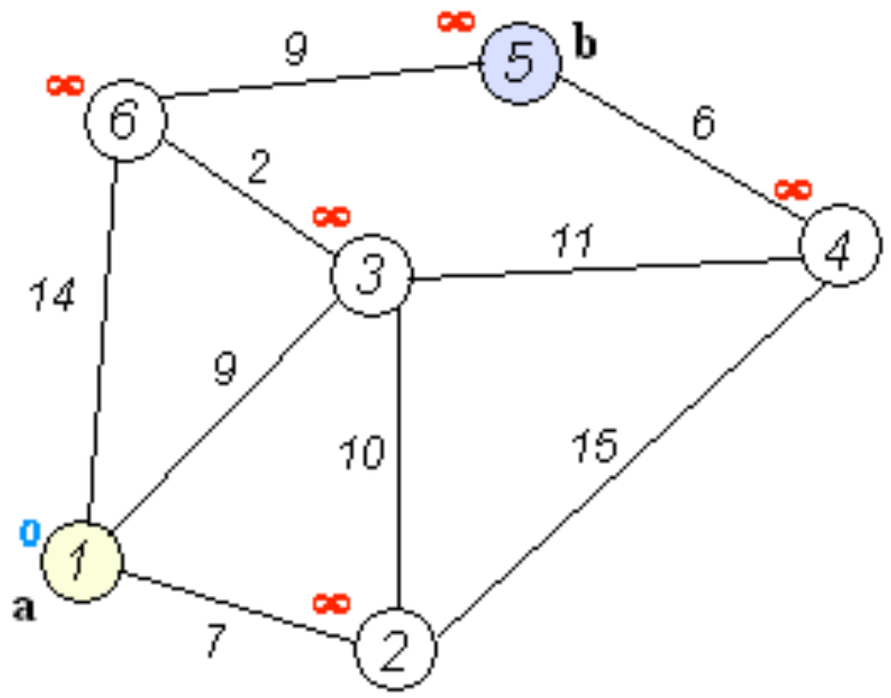
Stacks can be used to model human actions too. We talk about A, go off on a tangent about B, give some details about topic C, then pop the stack and talk about B some more. If we are done with B, we pop the stack again and return to topic A.

Only Nerds use these terms to talk about conversations, but that doesn't mean it is wrong.

# Stacks/Queues Used in Graphs

Even though stacks and queues are linear (they record an ordering based on when something showed up) they are very useful in traversing nonlinear structures like this.

Stacks/queues give us a way to record a path through some complex structure. E.g. recording the path of a B-Tree insert operation.



# Equivalence

How do we know if two things are equivalent? E.g. if I create two strings with the same character data and throw them into a set, should the set

A. Say yes, they are the same since they have the same character data, or

B. Say no, they are not the same since they are unique objects in memory?

# Could use ‘equal’

Answer: Depends on what you’re doing.

(Sorry)

If we want two different objects in memory to be treated as ‘equal’, we need to write code that determines equivalence. One way to do this is with an ‘equals’ function, or to override the ‘==’ operator.

# Equals Function

In Java:

```
public boolean equals(Thingy other) {
    return this.fooVal == other.fooVal &&
           this.barVal == other.barVal &&
           this.sillyFunction() == other.sillyFunction();
}
```

This can be pretty involved. Or not. Depends on how it is done.

# Equal is a Boolean Op.

If we have a set with lots of items and we want to see if an input item ***x*** is contained in the set, we could use an equals() function to test:

```
bool contains(x, the_set) {
   for (item in the_set) {
      if (item.equals(x)) {
         return true;
      }
   }
   return false;
}
```

What's the computational complexity of this operation, using the set size as n?

# Complexity using equals()

```
bool contains(x, the_set) {
   for (item in the_set) {
      if (item.equals(x)) {
         return true;
      }
   }
   return false;
}
```

An **O(n)** algorithm like this would be a terrible way to test membership, especially with large containers. We could use a hash function to do better, since it gives an integer, for which we have fast sort/find algorithms.

# Hash Function

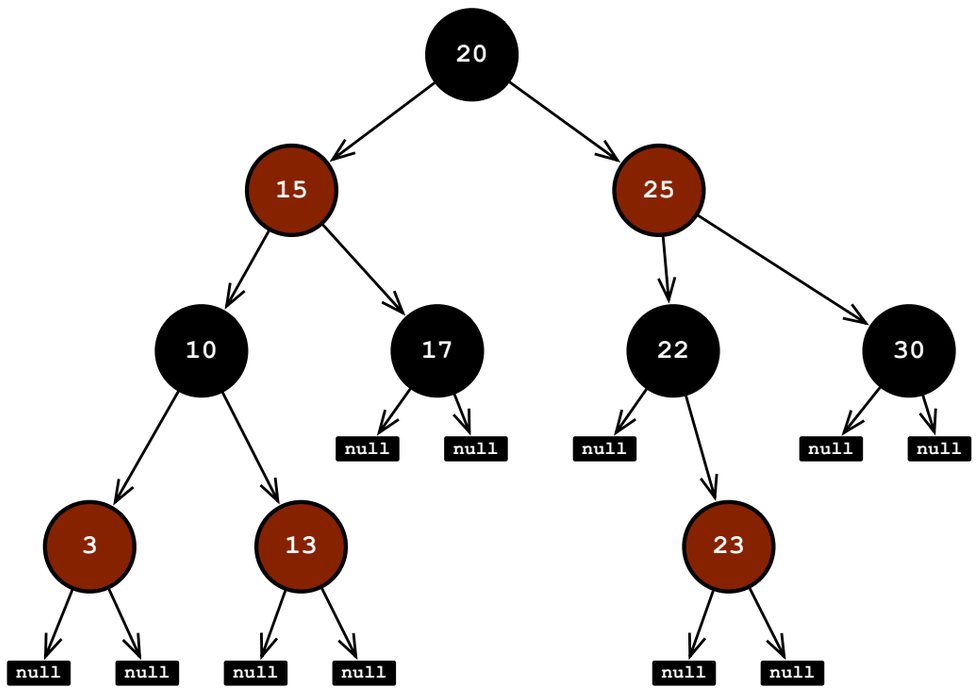
A very common alternative to testing equivalence is to use a *hash function*. A hash function is a bit of math that turns an arbitrary object into a number, called the *hash*. For any equivalent object, the same hash function will produce the same hash.

You want to use a hash function that gives you a good spread over the entire spectrum of numbers (e.g. positive numbers from 0 to $2^{32}$, for example). You also want to avoid *collisions* as much as possible.

# Complexity of Comparing Hashes

Since hashes give integers we can use straightforward algorithms (some of which we've done!) to keep things in order and find things quickly.

Example: use a red-black tree that uses *hash(obj)* to insert and find *obj*. What's the complexity for insert and remove on a red-black tree?




A: **$O(\log_2 n)$**

# Complexity *Can* be O(1)

If our hash function produces numbers in the range of [0, *n*], and we happen to have *size_of(n)* bytes of memory available, we can simply make a sparse array with *n* slots.

The insert and lookup for this is **O(1)**.

# Hashes are Integers

A hash function gives you an integer, so you have a way to keep objects ordered, even if that ordering has no meaning outside of your hash function's semantics.

With a hash, you can compare objects ***a*** and ***b*** using their hashes.

```
if (hash(a) == hash(b)) {
  // likely the same object. not 100% sure,
  // due to collisions.
}
```

# *Fast* Hash Function

A good hash function executes quickly: just a few instructions should do the trick. Here's a good one:

```
unsigned int djb_hash(string &s)
{
  int i;
  unsigned int h;

  h = 5381;

  for (i = 0; i < s.size(); i++) {
    h = (h << 5) + h + s[i];
  }
  return h;
}
```

What's the computational complexity of this, using the string size as *n*?

# Details about DJB Hash

```
unsigned int djb_hash(string &s)
{
  int i;
  unsigned int h;

  h = 5381;

  for (i = 0; i < s.size(); i++) {
    h = (h << 5) + h + s[i];
  }
  return h;
}
```

It computes the same hash reliably given the same input string.

This produces good coverage over the range of unsigned integers.

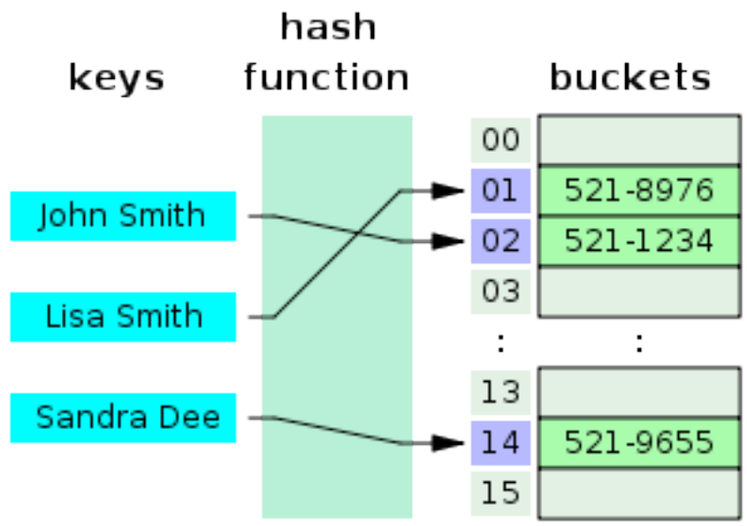
It is not likely to have collisions.

*h << 5* is a very fast way to compute h * $2^5$.

# Hash Collisions

Hash collisions are a favorite interview topic at famous companies whose names start with F, G or M.

Here's the graphic from Wikipedia's page on hash tables, which is a kind of Map that uses hashes as keys. Each object gets a *hash*, and this is used to find a *bucket* for the value. Sometimes two objects have the same hash, so the bucket needs to accommodate both of them.



# Collisions

If two objects hash to the same value, a container that uses hashes might then use an equality statement to see if they are identical.