B-Trees/Red-Black Trees

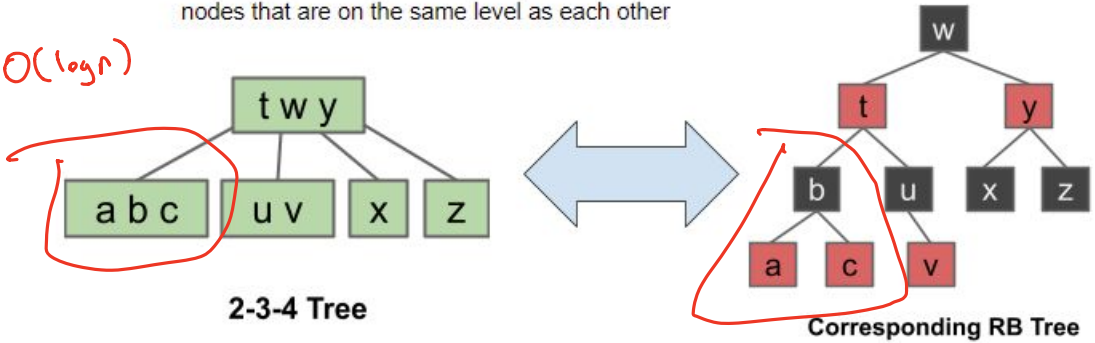
Hug's Slides: B-Trees, RB-Trees

What?

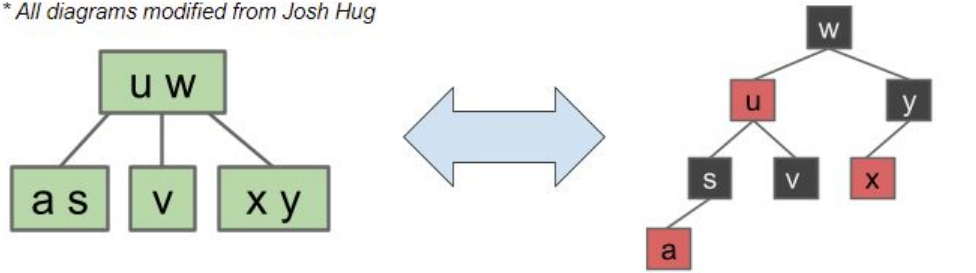
- B-Trees represent same idea as RB Trees, but the latter is easier to implement

B-Trees (2-3/2-3-4) and Red-Black Trees (LLRB/2-3-4):

- Insert into node until node overflows, then push middle (or left middle) element to the top
- B-Tree nodes **always** have the max number of children (# items in node + 1) or 0 children, all leaves are the same distance to the source
- Red-Black Trees are just a BST representation of a B-Tree, with red nodes to represent nodes that are on the same level as each other



** All diagrams modified from Josh Hug*



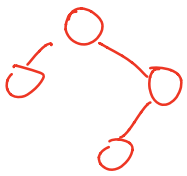
2-3 Tree **Corresponding LLRB Tree**

Left-Leaning Red-Black Tree Rotations:

- ALWAYS insert into LLRB with a red node, below are some common cases
  - If inserting on **right** with no siblings, **rotate left** on its parent
  - If inserting on **right** with a red node already to left, "**flip**" the two red nodes so their parent becomes the red node
  - If inserting on **left** with the parent as a red node, we have double left-leaning nodes. **Rotate right** on its parent then **color flip**
  - Inserting on **left** when the parent is not a red node is perfectly okay!

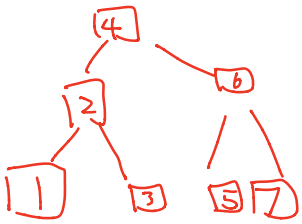
- Check these slides for visuals (Note: Professor Hug uses red links instead of nodes)

BST



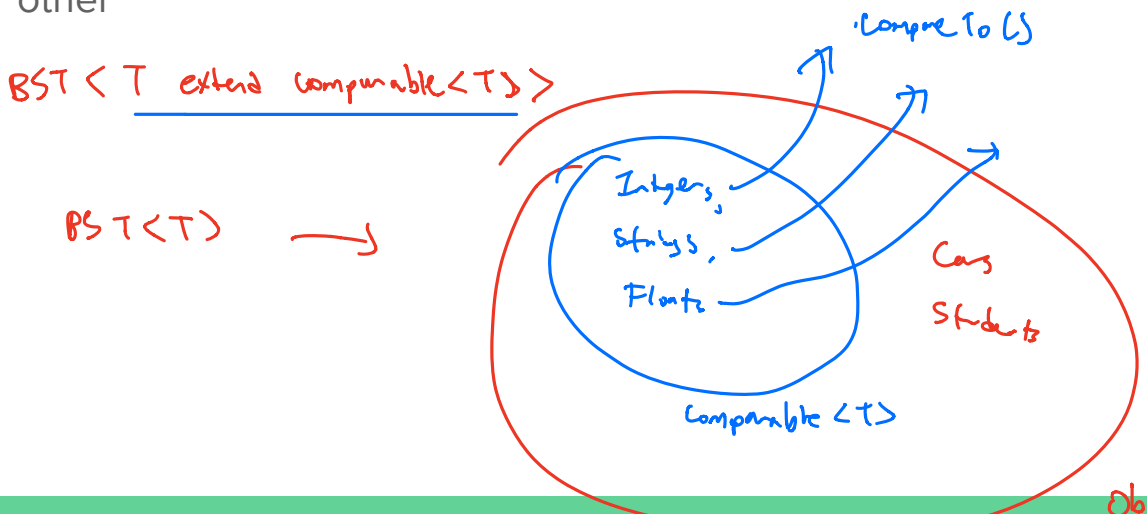
X, 2, 3, 4 ... 10

B-Tree



Why?

- Remember that BSTs can have a worst case runtime of O(n) if Spindly!
  - However, Red-Black Trees are harder to implement than BSTs
- Again - used as a storing mechanism for items that can be **compared** to each other



Examples

- Red-Black Java implementation

Hashing

Hash Tables → O(1) access

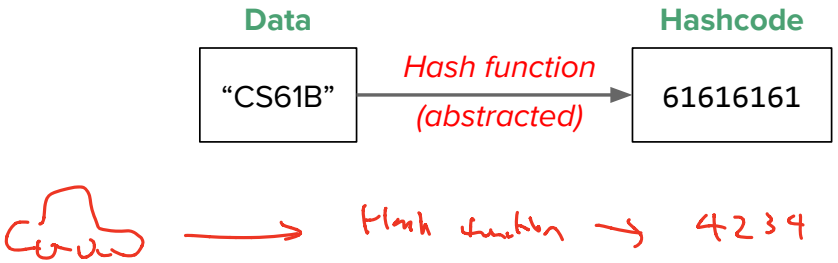
Hug's Slides

Overview

1. Hashing definition
2. Motivating example
3. Introducing hashing for real
4. The 3 steps of inserting into a hash table
5. Good hashcode properties!
6. Other behavior

What is Hashing?

- *Converts an object/data to an integer!*
- Integral in the **hash table** data structure, which we will introduce as motivation



Motivation

Background:

Facebook is a huge company, with billions of users

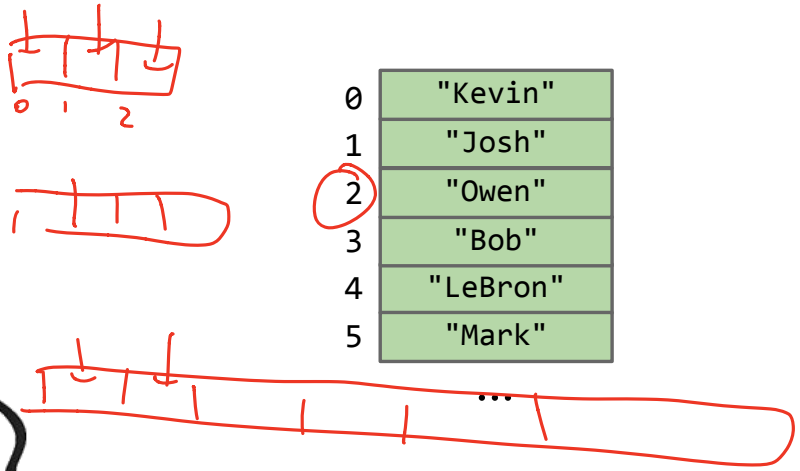
Goal:

Want to keep each user and their information in some database, but be able to access any user really quickly upon request

Motivation

What data structure can access things really fast?

An Array! Θ(1) access time

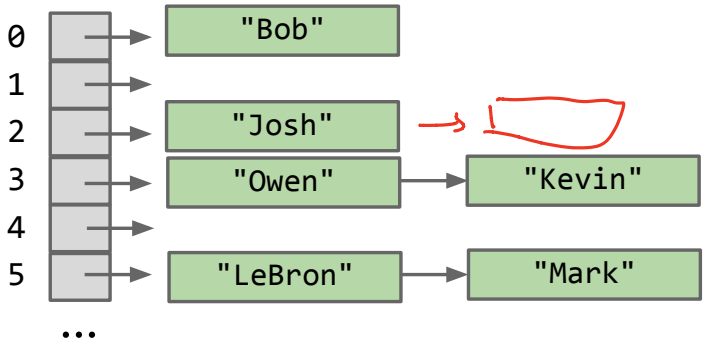


What happens as array grows large?

How do we know what size array to start with?

Motivation

This structure is called a *Hash Table*!



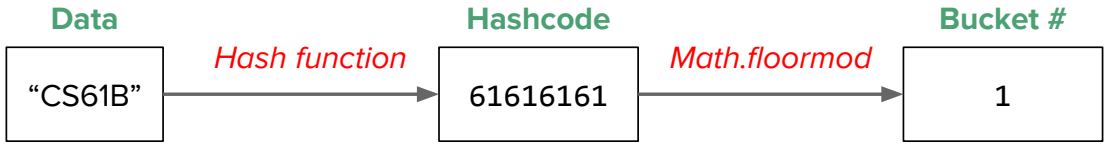
Still close to Θ(1) access if Linked List is small

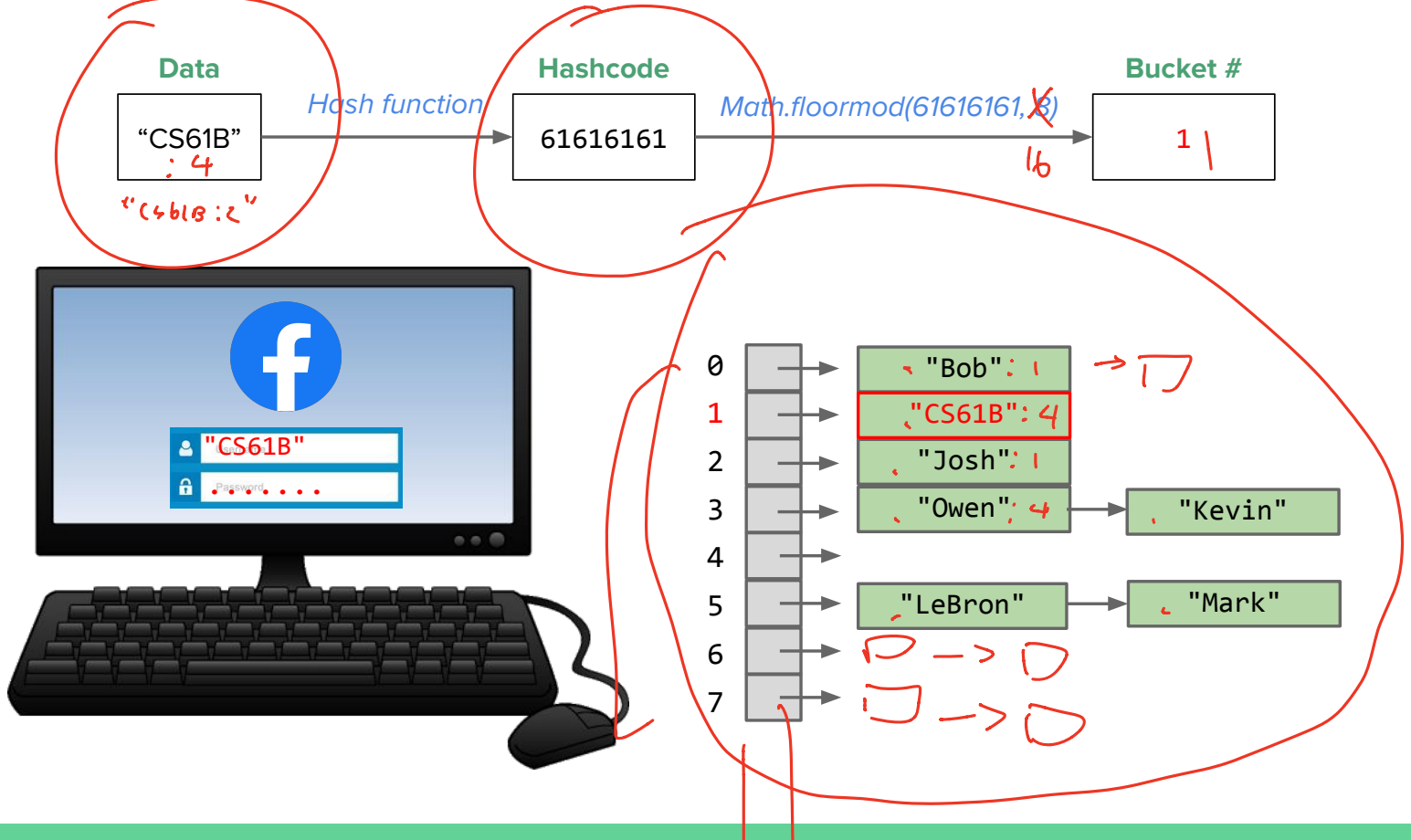
(Will discuss how to balance array size and # of links later)

Disclaimer: this is a very simplified example, probably not how Facebook actually does it!

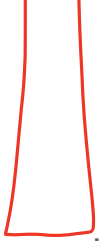
But how do we insert in the first place?

1. Calculate the object's **hashcode** from hash function (abstracted for now)
2. Take **hashcode** mod number of buckets to figure out which bucket to insert into
3. Insert object to the end of the bucket's Linked List

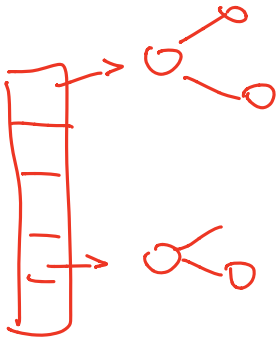



Data
"CS61B"
Hash function
Hashcode
61616161
Math.floormod(61616161,8)
Bucket #
1
"CS61B"
0
1
2
3
4
5
6
7
"Bob": 1
"CS61B": 4
"Josh": 1
"Owen": 4
"Kevin"
"LeBron"
"Mark"

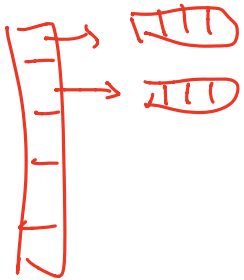
Question for You!



- Why not use other data structures for external chaining?
  - B-Trees/LLRB
  - Binary Search Trees
  - Arrays



Downside: Harder to implement, extend comparable



Downside: Fixed

Good Hashcode Properties

- If we have great hashcode, access time of hash table will be quick!
- **Deterministic**: repeated calls to hashcode() return the same thing (not Random)
- **Uniform**: Keys spread evenly across buckets
  - Don't use the first letter of a word as a hashcode! Only 26 letters possible
- **Quick**: Hashcode is relatively quick to compute

```
public MyClass{
    long a, b, c;

    @Override
    public int hashCode() {
     ...
    }
}
```

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + (int) (a ^ (a >>> 32));
    result = prime * result + (int) (b ^ (b >>> 32));
    result = prime * result + (int) (c ^ (c >>> 32));
    return result;
}
```

Hashcode functions usually based on small prime

Other Properties

- Hash Table can be used to implement both HashMaps and HashSets
- If exceed a **load factor**, resize array by a multiplicative factor, and REHASH all the items - many objects may be in a different bucket!
  - Load factor is a number, computed by (# objects / # buckets)
- When `<HashSet>.contains(<item>)` is called
  - Follow procedure of hashing item and calculating bucket
  - Then for every item in the linked list, **.equals()** is called until it reaches the end or finds the item.
    - .equals() is also unique for each object, similar to hashcode()

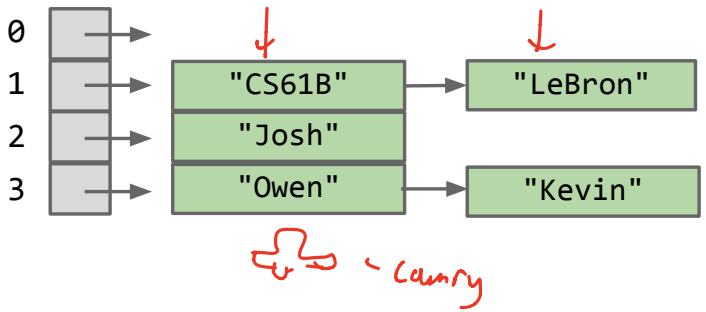
class Object

class Blob

public hashcode()

Resizing example

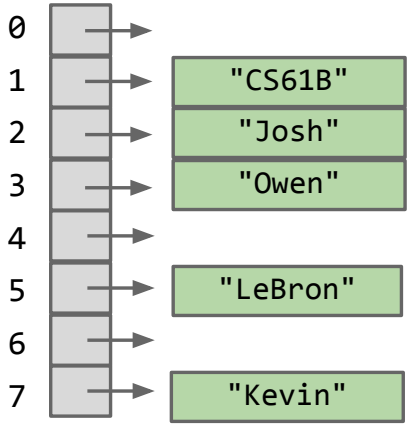
- Load factor = 1.5
  - When # elements / # buckets ≥ 1.5, double size of array!



"Mark"

Hashcode: 37

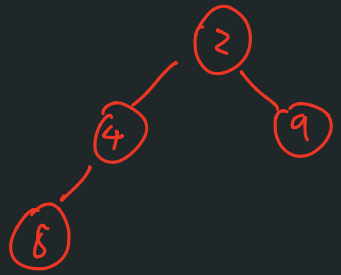
How to add?



Spring 2019 MT2

Warnings

- Mutating an object does NOT rehash it (can lead to mistakes in **.contains()**)
  - Ex. using custom object “Point”
- On exams, DO NOT assume a good hash function or amortized constant runtime!



Heaps/PQs

Hug's Slides

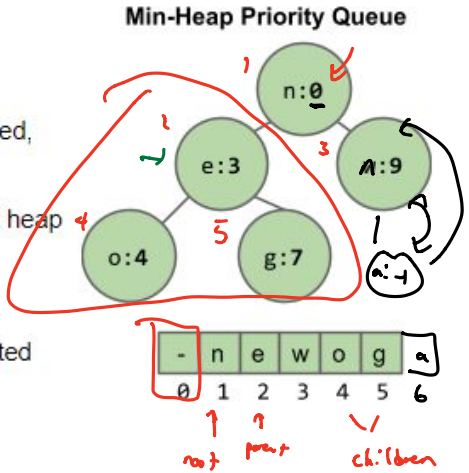
What?

- Heap is the data structure, Priority Queue is the Abstract Data Type (ADT)
- 2 major properties:

Heaps and Priority Queues:

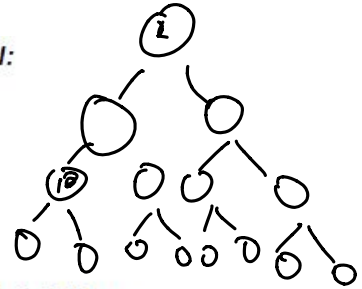
Heap Properties:

- **Complete:** Every level except last must be completely filled, nodes as far left as possible
- **Min-Heap:** Every node is less than or equal to child (Max heap - every node greater than or equal to child)
- 0th index nulled for easy child and parent indexing
- **Priority Queue** is an ABSTRACT DATA TYPE implemented with a heap (DATA STRUCTURE)!



Index References of Node in Heap with 0th index null:

- **Parent Index** = index / 2
- **Left Child Index** = index * 2
- **Right Child Index** = index * 2 + 1



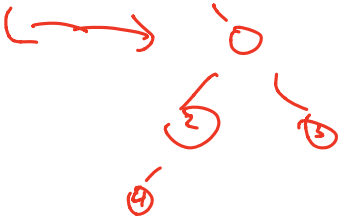
Methods:

- **Insert** - insert to as bottom and as left as possible, *bubble up*
- **deleteMin** - remove the top node and replace with bottom right, *bubble down*
- **changePriority (Θ(log(n))** - *bubble up* or *down* depending on if priority became higher or lower

Exam Tips:

- The only thing you know about a min-heap is that its children must be greater than or equal to it, and the **root node is the min value**. Thus, for a min-heap with unique values, any node that has **no children can be the largest value** and any node that doesn't have **more than half of the total number of nodes as children or descendants** can be the median value

Why?



- Used for a very specific purpose - to extract the highest/lowest priority node in O(1) time (wow!)
  - Priority can mean many things, from smallest number, largest number, longest word, however you want to define it in your structure
- Can “re-adjust” itself in O(logn) time
  - Aka once you extract the priority node, can modify itself so the next priority node is now on top

Examples

- Exam problems from Princeton