Introduction to Hash Tables

Data Structures and Algorithms

Announcements

I’m not Ben – I’m Robbie

Your checkpoint is due tonight at midnight

Tag your commit with `SUBMIT-CHECKPOINT`

Right click on the project -> team ->advanced -> tag.

Look closely at the last screen, there may be multiple checkboxes to check.

On your projects, you should be pair programming

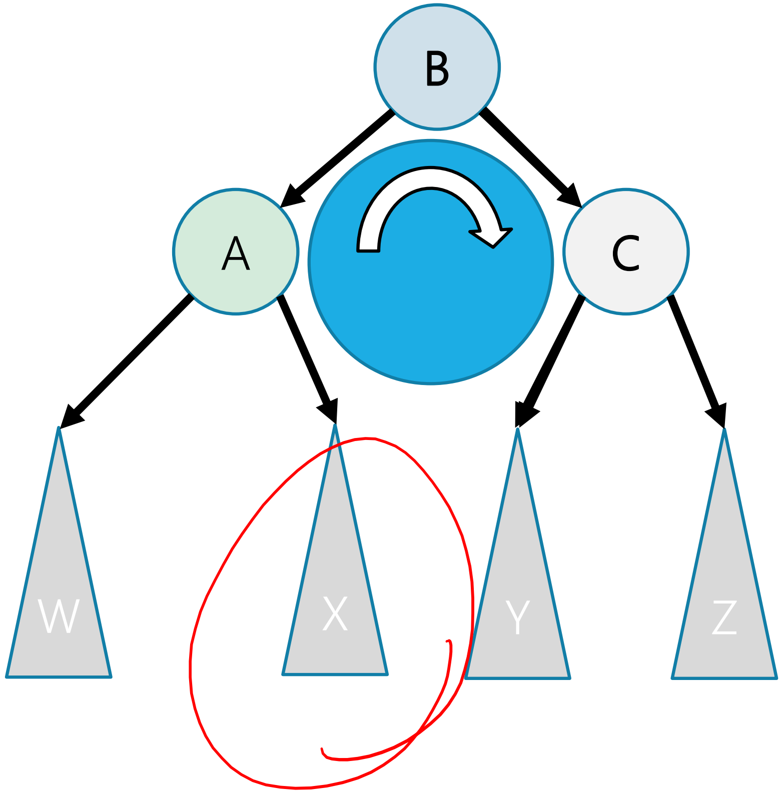
You are expected to understand every line of code your group submits.

Implementation details are fair game for the midterm!

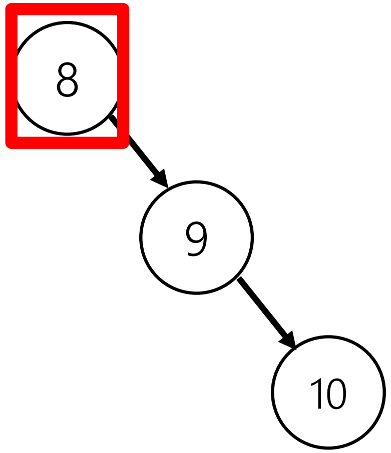
Warm Up Review

Draw the AVL tree that results from inserting 8, 9, 10, 12, 11.

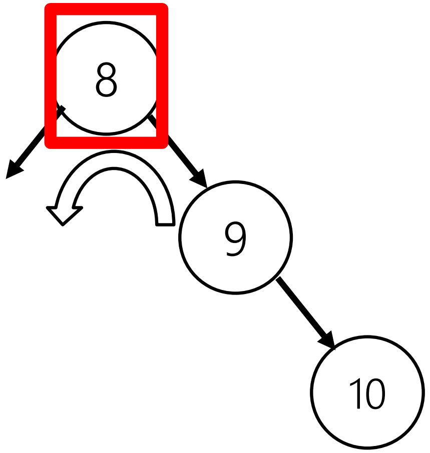
Tree Rotations



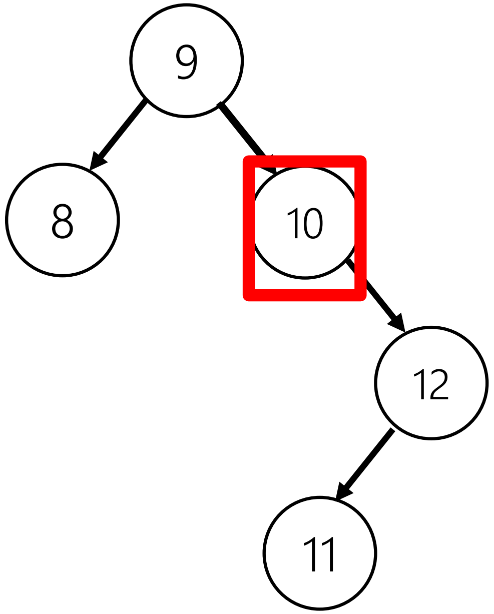
AVL Example: 8,9,10,12,11



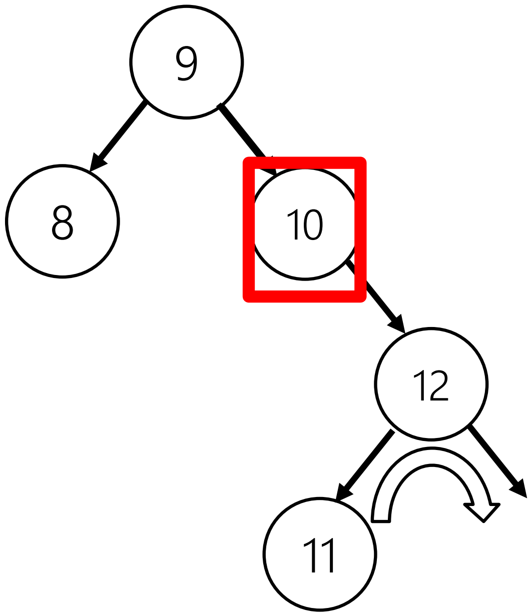
AVL Example: 8,9,10,12,11



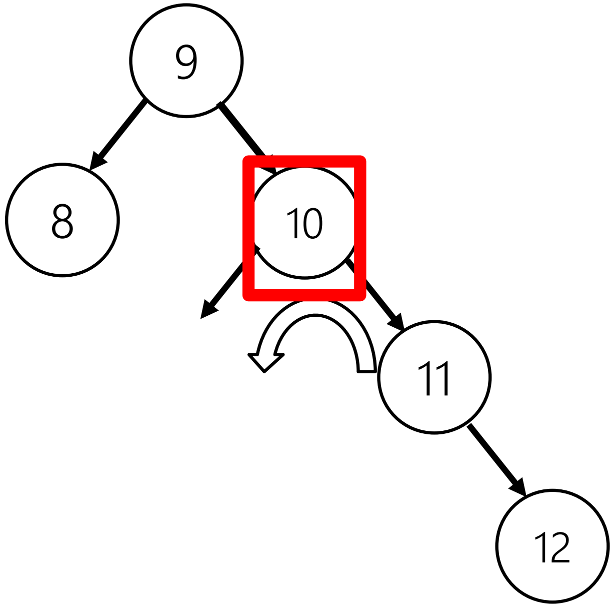
AVL Example: 8,9,10,12,11



AVL Example: 8,9,10,12,11



AVL Example: 8,9,10,12,11



Pros and Cons of AVL Trees

Arguments for AVL trees:

1. All operations logarithmic worst-case because trees are *always* balanced
2. Height balancing adds no more than a constant factor to the speed of `insert` and `delete`

Arguments against AVL trees:

1. Difficult to program & debug [but done once in a library!]
2. More space for height field
3. Asymptotically faster but rebalancing takes a little time
4. If *amortized* logarithmic time is enough, use splay trees (also in the text, not covered in this class)

Lots of cool Self-Balancing BSTs out there!

Popular self-balancing BSTs include:

AVL tree

Splay tree

2-3 tree

AA tree

Red-black tree

Scapegoat tree

Treap

(Not covered in this class, but several are in the textbook and all of them are online!)

(From https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree#Implementations)

The Story So Far…

Why are we so obsessed with Dictionaries? It's all about data baby!

When dealing with data:

- Adding data to your collection
- Getting data out of your collection
- Rearranging data in your collection

Operation		ArrayList	LinkedList	BST	AVLTree
put(key,value)	best	O(1)	O(1)	O(1)	O(1)
	average	O(n)	O(1)	O(logn)	O(logn)
	worst	O(n) + arrayExpansion	O(1)	O(n)	O(logn)
get(key)	best	O(1)	O(1)	O(1)	O(1)
	average	O(n), O(logn) if sorted	O(n)	O(logn)	O(logn)
	worst	O(n)	O(n)	O(n)	O(logn)
remove(key)	best	O(1)	O(1)	O(logn)	O(1)
	average	O(n)	O(n)	O(logn) + replace	O(logn)
	worst	O(n)	O(n)	O(n) + replace	O(logn) + rotation

Can we do better?

Implement a dictionary that accepts only integer keys between 0 and some value k

Leverage Array Indices! “Direct address map”

Operation		Array w/ indices as keys
put(key,value)	best	
	average	
	worst	
get(key)	best	
	average	
	worst	
remove(key)	best	
	average	
	worst	

Can we do better?

Implement a dictionary that accepts only integer keys between 0 and some value k

Leverage Array Indices! "Direct address map"

Operation		Array w/ indices as keys
put(key,value)	best	O(1)
	average	O(1)
	worst	O(1) + arrayExpansion
get(key)	best	O(1)
	average	O(1)
	worst	O(1)
remove(key)	best	O(1)
	average	O(1)
	worst	O(1)

Implement Direct Access Map

```
public V get(int key) {
   this.ensureIndexNotNull(key);
   return this.array[key].value;
}

public void put(int key, V value) {
    this.array[key] = value;
}

public void remove(int key) {
   this.entureIndexNotNull(key);
   this.array[key] = null;
}
```

Can we do this for any integer?

Idea 1:

Create a GIANT array with every possible integer as an index

Problems:

- Can we allocate an array big enough?
- Super wasteful

Idea 2:

Create a smaller array, but create a way to translate given integer keys into available indices

Problem:

- How can we pick a good translation?

Review: Integer remainder with %

The `%` operator computes the remainder from integer division.

```
14 % 4 is 2             218 % 5 is 3

     3                       43
4 ) 14                  5 ) 218
    12                      20
     2                       18
                             15
                              3
```

Applications of `%` operator:

- Obtain last digit of a number: `230857 % 10` is `7`
- See whether a number is odd: `7 % 2` is `1`, `42 % 2` is `0`
- Limit integers to specific range: `8 % 12` is `8`, `18 % 12` is `6`

First Hash Function: % table size

indices	0	1	2	3	4	5	6	7	8	9
elements	“poo”	“biz”				“bar”			“bop”	

```
put(0, “foo”);  0 % 10 = 0
put(5, “bar”);  5 % 10 = 5
put(11, “biz”)  11 % 10 = 1
put(18, “bop”); 18 % 10 = 8
put(20, “poo”); 20 % 10 = 0
```

→ Collision!

Implement First Hash Function

```
public V get(int key) {
   int newKey = key % this.array.length;
   this.ensureIndexNotNull(key);
   return this.array[key].value;
}

public void put(int key, V value) {
    this.array[key % this.array.length] = value;
}
public void remove(int key) {
   int newKey = key % this.array.length;
   this.entureIndexNotNull(key);
   this.array[key] = null;
}
```

Hash Obsession: Collisions

When multiple keys translate to the same location of the array

The fewer the collisions, the better the runtime!

Handling Collisions

Solution 1: Chaining

Each space holds a "bucket" that can store multiple values. Bucket is often implemented with a LinkedList

Operation		Array w/ indices as keys
put(key,value)	best	O(1)
	average	O( )
	worst	O(n)
get(key)	best	O(1)
	average	O( )
	worst	O(n)
remove(key)	best	O(1)
	average	O( )
	worst	O(n)

Handling Collisions

Solution 1: Chaining

Each space holds a "bucket" that can store multiple values. Bucket is often implemented with a LinkedList

Operation		Array w/ indices as keys
put(key,value)	best	O(1)
	average	O(1 + λ)
	worst	O(n)
get(key)	best	O(1)
	average	O(1 + λ)
	worst	O(n)
remove(key)	best	O(1)
	average	O(1 + λ)
	worst	O(n)

Average Case:
Depends on average number of elements per chain

Load Factor λ

If n is the total number of key-value pairs

Let c be the capacity of array

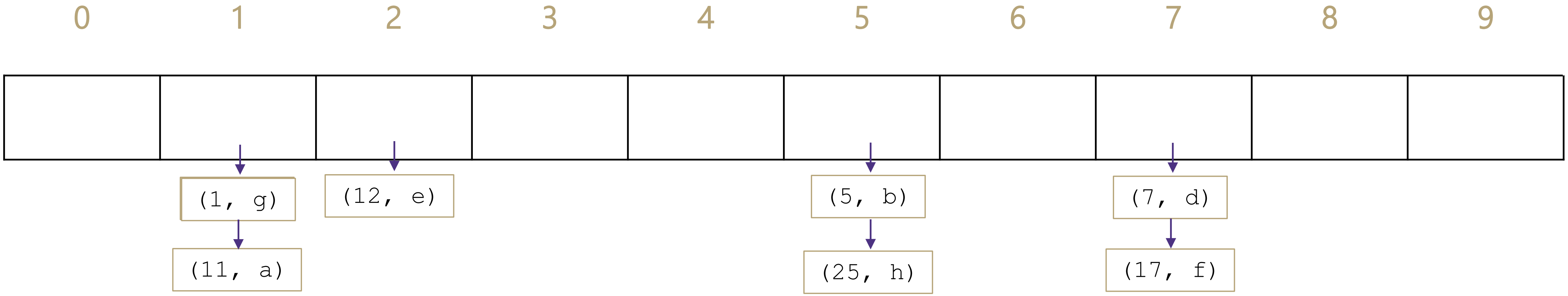
Load Factor λ = $\frac{n}{c}$

Practice

Consider an IntegerDictionary using separate chaining with an internal capacity of 10. Assume our buckets are implemented using a LinkedList where we append new key-value pairs to the end.

Now, suppose we insert the following key-value pairs. What does the dictionary internally look like?

(1, a) (5,b) (11,a) (7,d) (12,e) (17,f) (1,g) (25,h)



0 1 2 3 4 5 6 7 8 9

(1, g)

(11, a)

(12, e)

(5, b)

(25, h)

(7, d)

(17, f)

Can we do better?

Idea 1: Take in better keys

- Can't do anything about that right now

Idea 2: Optimize the bucket

- Use an AVL tree instead of a Linked List
- Java starts off as a linked list then converts to AVL tree when collisions get large

Idea 3: Modify the array's internal capacity

- When load factor gets too high, resize array
  - Double size of array
  - Increase array size to next prime number that's roughly double the array size
    - Prime numbers reduce collisions when using % because of divisors
  - Resize when $\lambda \approx 1.0$
  - When you resize, you have to rehash

What about non integer keys?

Hash Function

An algorithm that maps a given key to an integer representing the index in the array for where to store the associated value

Goals

Avoid collisions

- The more collisions, the further we move away from O(1)
- Produce a wide range of indices

Uniform distribution of outputs

- Optimize for memory usage

Low computational costs

- Hash function is called every time we want to interact with the data

How to Hash non Integer Keys

Implementation 1: Simple aspect of values

```
public int hashCode(String input) {
    return input.length();
}
```

Pro: super fast O(1)

Con: lots of collisions!

Implementation 2: More aspects of value

```
public int hashCode(String input) {
    int output = 0;
    for(char c : input) {
        out += (int)c;
    }
    return output;
}
```

Pro: fast O(n)

Con: some collisions

Implementation 3: Multiple aspects of value + math!

```
public int hashCode(String input) {
    int output = 1;
    for (char c : input) {
        int nextPrime = getNextPrime();
        out *= Math.pow(nextPrime, (int)c);
    }
    return Math.pow(nextPrime, input.length());
}
```

Pro: few collisions

Con: slow, gigantic integers

Balanced Hash Function

```
public int hashCode(String input) {
   int accum = 1;
   int output = 0;
   for (char c : input) {
      out += accum * (int)c;
      accum *= 31;
   }
   return output;
}
```

Pretty fast, O(n)
Uses both character values and positions, few collisions
Why 31? Magical research!

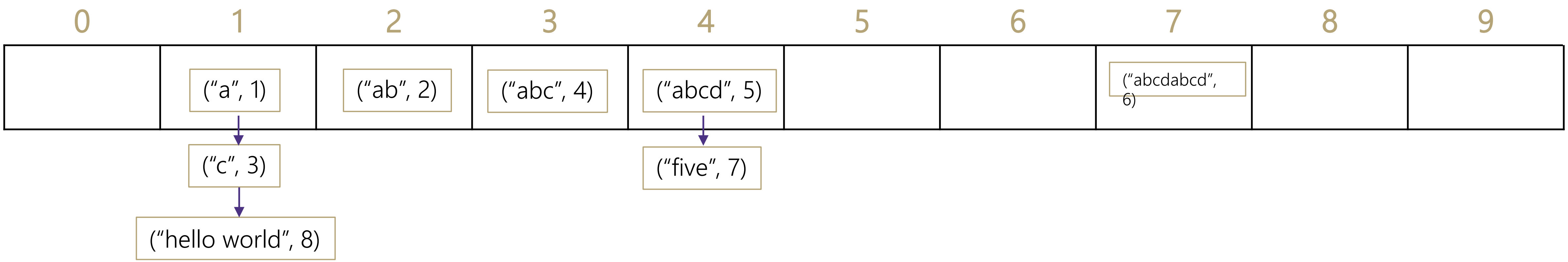
Practice

Consider a StringDictionary using separate chaining with an internal capacity of 10. Assume our buckets are implemented using a LinkedList. Use the following hash function:

```
public int hashCode(String input) {
    return input.length() % arr.length;
}
```

Now, insert the following key-value pairs. What does the dictionary internally look like?

("a", 1) ("ab", 2) ("c", 3) ("abc", 4) ("abcd", 5) ("abcdabcd", 6) ("five", 7) ("hello world", 8)



0 1 2 3 4 5 6 7 8 9

1: ("a", 1) → ("c", 3) → ("hello world", 8)

2: ("ab", 2)

3: ("abc", 4)

4: ("abcd", 5) → ("five", 7)

7: ("abcdabcd", 6)

Java and Hash Functions

Object class includes default functionality:

- equals
- hashCode

If you want to implement your own hashCode you MUST:

- Override BOTH hashCode() and equals()
- If a.equals(b) is true then a.hashCode() == b.hashCode() MUST also be true

Another Way to Deal With Collisions

Our strategy today:

- Find some way to store the new element where the hash function wanted to put it.

Alternative strategy:
just put it somewhere nearby

Two problems:

- What's nearby?
  How do we make sure we can find it later?

Ben will tell you on Monday!