B-Trees, LLRBs, Hashing

Exam Prep 07

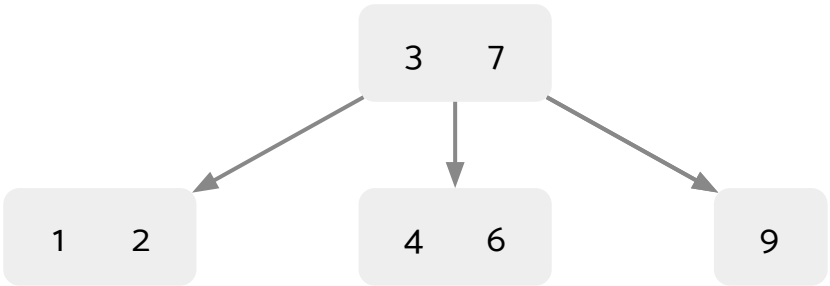
Announcements

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
			3/6 Project 2A Due		3/8 Lab 7 Due	
					3/15 Lab 8 Due	

Content Review

B-Trees

B-Trees are trees that serve a similar function to binary trees while ensuring a bushy structure (check: why don't BSTs/binary trees generally?). In this class, we'll often use B-Tree interchangeably with 2-3 Trees.

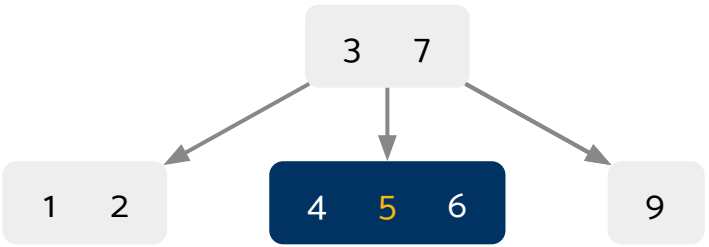


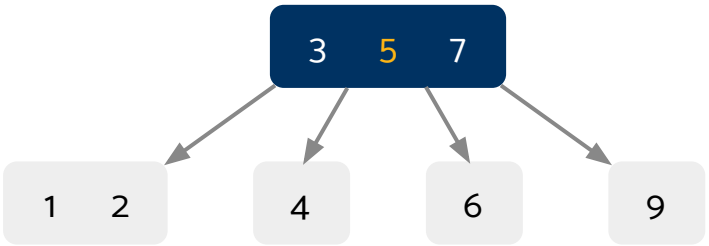
Each node can have up to 2 items and 3 children. There are variations where these values are higher, known as 2-3-4 trees (nodes can have up to 3 items and 4 children).

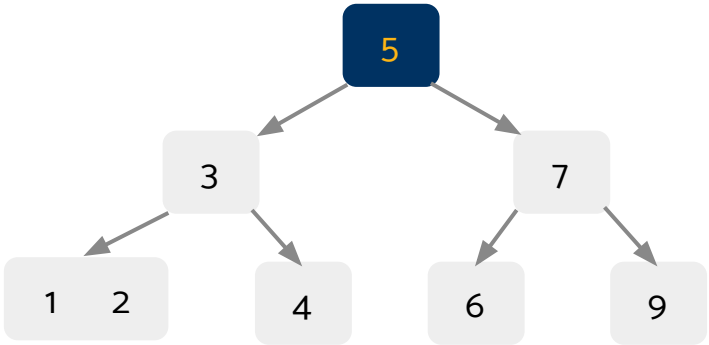
All leaves are the same distance from the root, which makes getting take Θ(log N) time.

B-Trees

When adding to a B-Tree, you first start by adding to a leaf node, and then pushing the excess items (typically the middle element) up the tree until it follows the rules (max 2 elements per node, max 3 children per node).

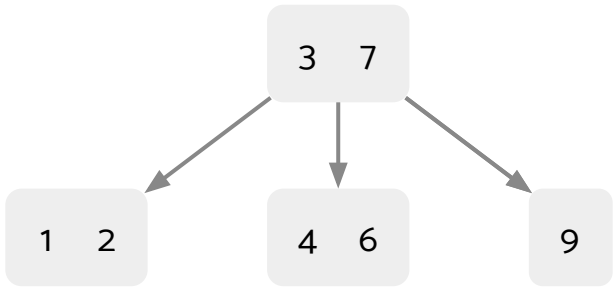


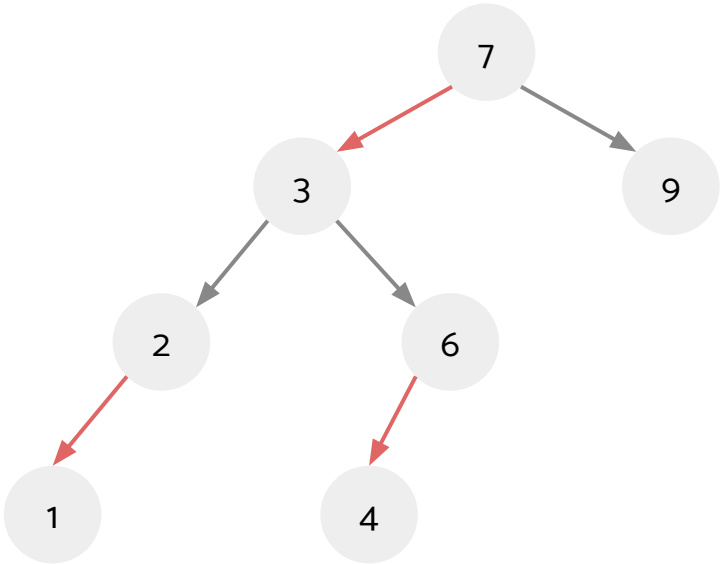




Left Leaning Red Black Trees

LLRBs are a representation of B-trees that we use because it is easier to work with in code. In an LLRB, each multi-node in a 2-3 tree is represented using a red connection on the left side.





LLRB Rules

Each 2-3 tree corresponds to a (unique) LLRB*. This implies that:

1. **The LLRB must have the same number of black links in all paths from root to null (not root to leaf!)**
2. **A node may not have two red children**
3. **All red links should be left-leaning**
4. **Height cannot be more than ~2x the height of its corresponding 2-3 tree**
5. **Additionally, we insert elements as leaves with red links to their parent node**

All these invariants mean that sometimes our LLRB becomes unbalanced (ie. it violates a rule), so we need some way to fix that.

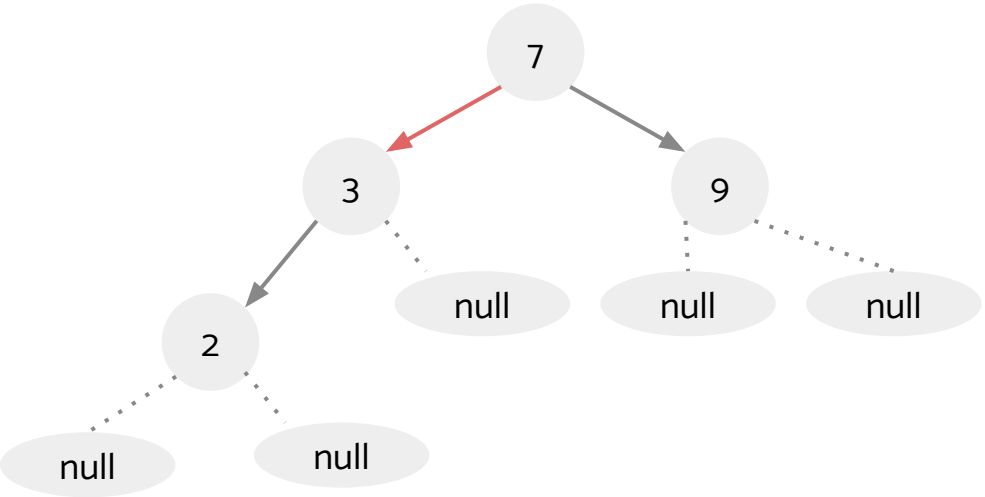
*2-3-4 trees correspond more generally to regular Red Black Trees, but our focus in 61B is on LLRBs.

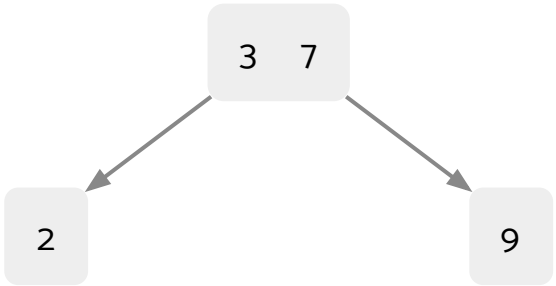
Why root to null?

Consider the left image below: each leaf is 1 black link away from the root, but it's not a valid LLRB!

This is because when we convert it to a B-tree, the [3, 7] node will only have 2 children, not 3!

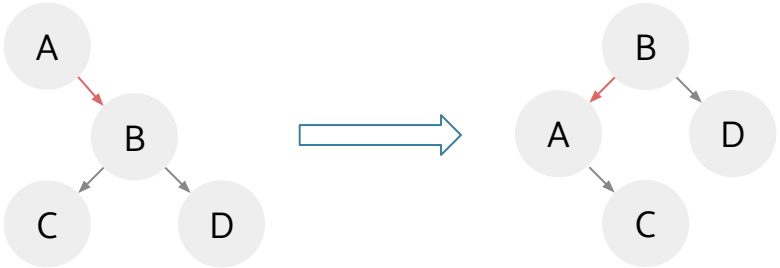
If we check for root-to-null: the right of 3 is 0 black link away, but the other nulls are 1 black link away.





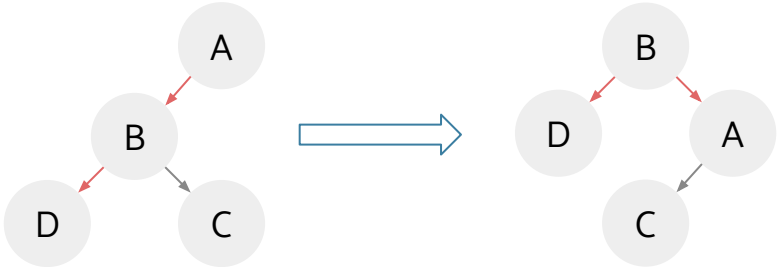
LLRB Balancing Operations

`rotateLeft(A);`



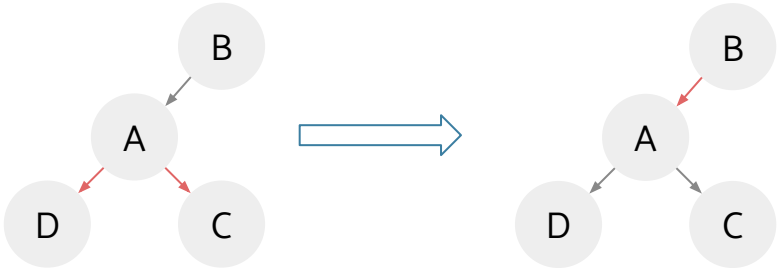
we can't have a right red link

`rotateRight(A);`



we can't have 2+ consecutive (left) red links

`colorFlip(A);`



we can't have both child links of a node be red

Hashing

Hash functions are functions that represent objects as integers so we can efficiently use data structures like HashSet and HashMap for fast operations (ie. get, put/add).

Once we have a hash for our object, we use modulo to find out which "bucket" it goes into. For example, we can create a hash function for the Dog class by overriding Object's hashCode():

```
@Override
public int hashCode() {
    return 37 * this.size + 42;
}
```

Then, when we try to put the dog into a HashSet, the HashSet code might look something like this:

```
int targetBucket = dog.hashCode() % numBuckets;
addToTargetBucket(dog, targetBucket);
```

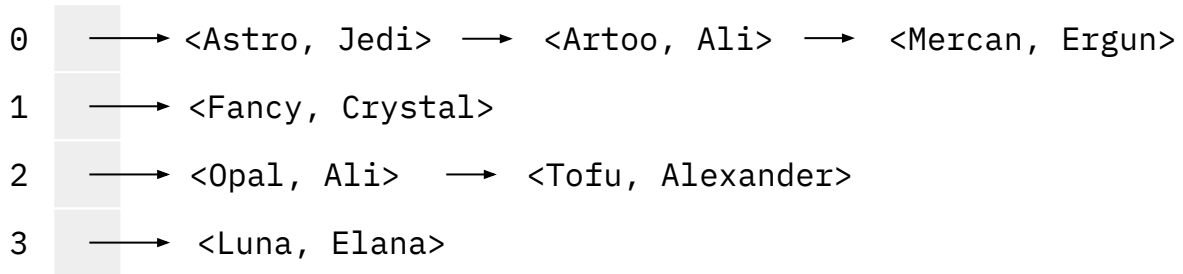


5 6 7 8 9
0 1 2 3 4


尾号0
尾号1
尾号2
尾号3
尾号4
尾号5
尾号6
尾号7
尾号8
尾号9
八大权益 全新升级
新品尝鲜
免配送费
35090-1
35082-1
35062-1
35073-1
35053-1
35094-1
35006-1
35067-1
35987-1
35087-1
35098-1
35099-1

Hashing

In each bucket, we deal with having lots of items by chaining the items and using .equals to find what we are looking for. In a HashMap, we're specifically concerned with equality of keys in key-value pairs (in HashSet, we only have a value to compare to).



Therefore, it is important that your .equals() function matches the result of comparing hashcodes - if two items are equal, they must also have the same hashcode

Hashing

The **load factor** tells us when we should resize. We calculate it by dividing the total number of elements added by the number of buckets we currently have. When resizing up, if the load factor exceeds some threshold, we increase the number of buckets we use in the data structure.

Because all elements were initially placed into buckets based on how many buckets were previously available, we also need to rehash all elements into a potentially new destination bucket when resizing, or else subsequent calls to get() may fail.*

* Resizing sounds like a linear-time operation…how does that affect the runtimes of our operations?

Valid vs. Good Hashcodes

Properties of a valid hashcode:

1) Must be an integer
2) The hashcode for the same object should always be the same
3) If two objects are “equal”, they have the same hashcode
    - Check! What about the reverse?

Properties of a good hashcode:

1) Distributes elements evenly
    - What does this even mean?

Worksheet

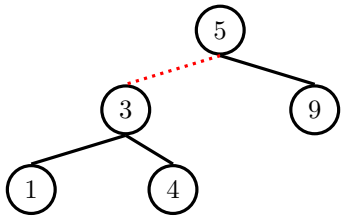
CS 61B Spring 2024

B-Trees, LLRBs, Hashing

Exam-Level 07: March 4, 2024

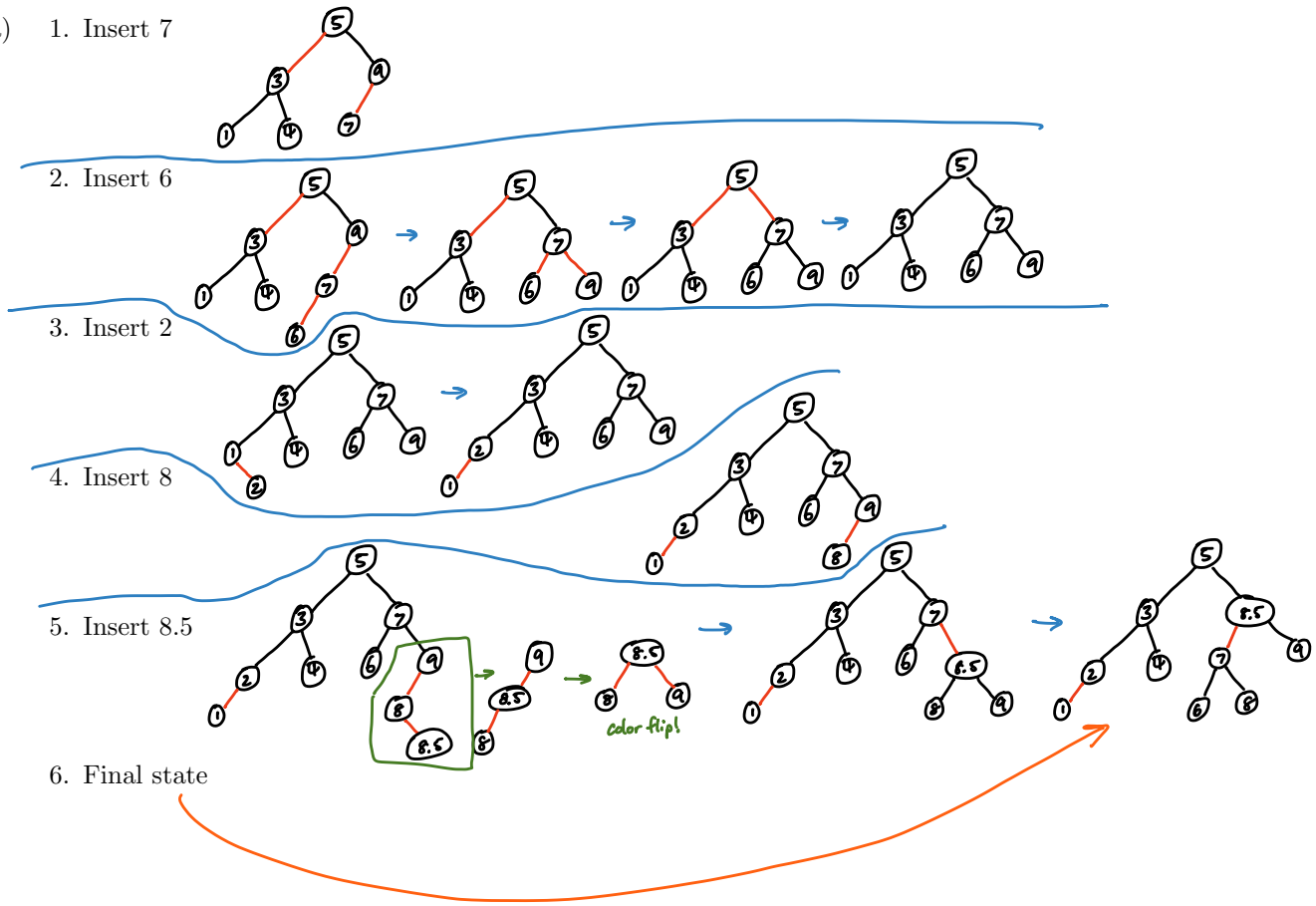
1 LLRB Insertions

Given the LLRB below, perform the following insertions and draw the final state of the LLRB. In addition, for each insertion, write the balancing operations needed in the correct order (rotate right, rotate left, or color flip). If no balancing operations are needed, write "Nothing". Assume that the link between 5 and 3 is red and all other links are black at the start.



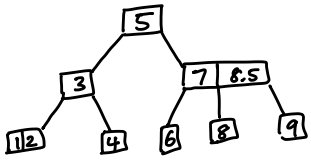
(a)

1. Insert 7
2. Insert 6
3. Insert 2
4. Insert 8
5. Insert 8.5
6. Final state



(b) Convert the final LLRB to its corresponding 2-3 Tree.

1:1 mapping since left-leaning



2 Hashing Gone Crazy

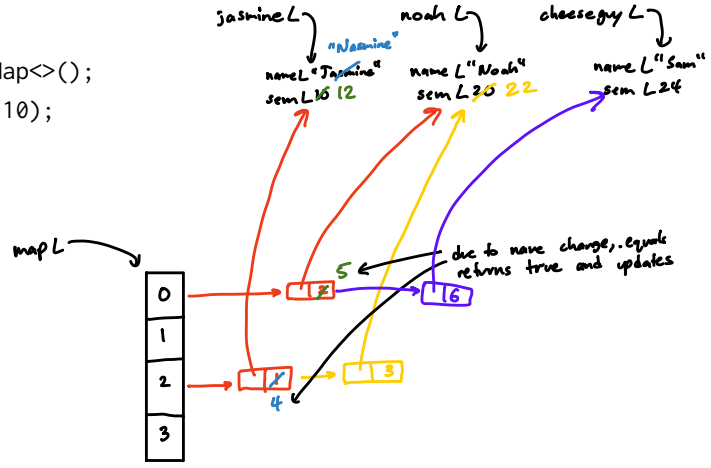
For this question, use the following `TA` class for reference.

```
1   public class TA {
2       int semester;
3       String name;
4       TA(String name, int semester) {
5           this.name = name;
6           this.semester = semester;
7       }
8       @Override
9       public boolean equals(Object o) {
10          TA other = (TA) o;
11          return other.name.charAt(0) == this.name.charAt(0);
12      }
13      @Override
14      public int hashCode() {
15          return semester;
16      }
17  }
```

Assume that the `hashCode` of a `TA` object returns `semester`, and the `equals` method returns `true` if and only if two `TA` objects have the same first letter in their `name`.

Assume that the `ECHashMap` is a `HashMap` implemented with external chaining as depicted in lecture. The `ECHashMap` instance begins at size 4 and, for simplicity, does not resize. Draw the contents of `map` after the executing the insertions below:

```
1   ECHashMap<TA, Integer> map = new ECHashMap<>();
2   TA jasmine = new TA("Jasmine the GOAT", 10);
3   TA noah = new TA("Noah", 20);
4   map.put(jasmine, 1);
5   map.put(noah, 2);
6
7   noah.semester += 2;
8   map.put(noah, 3);
9
10  jasmine.name = "Nasmine";
11  map.put(noah, 4);
12
13  jasmine.semester += 2;
14  map.put(jasmine, 5);
15
16  jasmine.name = "Jasmine";
17  TA cheeseguy = new TA("Sam", 24);
18  map.put(cheeseguy, 6);
```



3 Buggy Hash

The following classes may contain a bug in one of its methods. Identify those errors and briefly explain why they are incorrect and in which situations would the bug cause problems.

```
1   class Timezone {
2       String timeZone; // "PST", "EST" etc.
3       boolean dayLight;
4       String location;
5       ...
6       public  int currentTime() {
7           // return the current time in that time zone
8       }
9       public int hashCode() {
10          return currentTime();
11      }
12      public boolean equals(Object o) {
13          Timezone tz = (Timezone) o;
14          return tz.timeZone.equals(timeZone);
15      }
16  }
```

not deterministic! hashcode changes randomly with respect to object

```
1   class Course {
2       int courseCode;
3       int yearOffered;
4       String[] staff;
5       ...
6       public int hashCode() {
7           return yearOffered + courseCode;
8       }
9       public boolean equals(Object o) {
10          Course c = (Course) o;
11          return c.courseCode == courseCode;
12      }
13  }
```

equality must imply matching hashcode
vice versa is not true (collisions)