CSM 61B Spring 2017 — Hashing & Heaps

Week 10: March 20, 2017

1 Motivation

1.1 (a) In the worst case, how long does it take to index into a linked list?

$\Theta(N)$

(b) In the worst case, how long does it take to index into an array?

$\Theta(1)$

(c) In the worst case, how long does it take to insert into a linked list?

$\Theta(N)$, where N is the length of the linked list.

(d) Assuming there's space, how long does it take to put a element in an array?

$\Theta(1)$

(e) What if we assume there is no more space in the array?

$\Theta(N)$ to copy over N elements into the new array.

(f) Given what we know about linked lists and arrays, when would we choose to use one data structure over the other?

If you know in advance how large your data structure is, arrays are faster than linked lists in insertion, mutation, etc. However, if the array needs to expand frequently then things get expensive. But there are ways to amortize the cost of resizing with `ArrayLists`, for example.

- Objects with rows and columns (like a chessboard) where we wish to randomly index into exact position and where the board is of a fixed size
- Arguments to a java program: `String[] args`. Using a resizing `List` in this scenario doesn't necessarily make things better since the arguments to a program don't change once we start the program.

2 Hash Table Basics

2.1 Consider `BadHashMap`'s `put` implementation which assumes no collisions or negative hash codes.

```
public class BadHashMap<K, V> implements Map<K, V> {
    private V[] array;
    public void put(K key, V value) {
        this.array[key.hashCode() % this.array.length] = value;
    }
}
```

(a) Why do we use the `%` (modulo) operator?

Allows us to take a large `int` and turn it into an index into the `array`.

(b) What are collisions? What data structure can we use to address them?

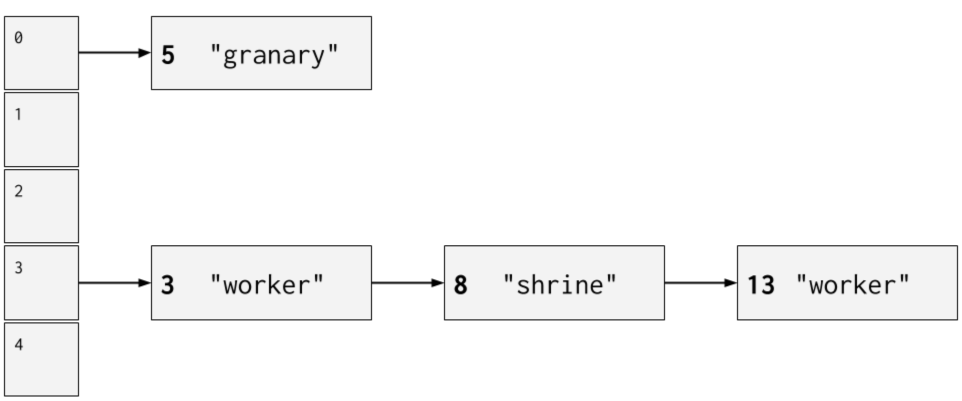
Collisions occur when two keys map to the same bucket, or index, in the `array`. We can use external chaining with a linked list, for example, to resolve the conflicts. Each bucket can be associated with more than one key.

(c) Describe an implementation for `get` and `containsKey` assuming you applied the changes for handling collisions.

An implementation for `get` might select the appropriate bucket, then iterate through the external chain, and return the value for the matching key if it exists. If no matching key exists, return null. Then, `containsKey` can just return whether `get(key) != null`.

2.2 Consider a `HashMap<Integer, String>` with an underlying array of size 5. Draw the resulting structure after the following operations. `Integer::hashCode` returns the integer's value itself.

```
put(3, "monument");
put(8, "shrine");
put(3, "worker");
put(5, "granary");
put(13, "worker");
```



`"worker"` replaces `"monument"` as their keys are the same. Each `put` must iterate through the entire external chain to ensure that a key-update is not necessary.

3 Hash Codes

3.1 Which of the following `hashCode`s are valid?

```
class Point {
    private int x;
    private int y;
    private static count = 0;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
        count += 1;
    }
```

(a)
```
public void hashCode() {
    System.out.print(this.x + this.y);
}
```

Invalid. Return type should be `int`.

(b)
```
public int hashCode() {
    Random random = new Random();
    return random.nextInt();
}
```

Invalid. Not deterministic.

(c)
```
public int hashCode() {
    return this.x + this.y;
}
```

Valid, but certain inputs may cause a significant number of collisions.

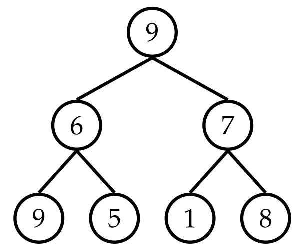
(d)
```
public int hashCode() {
    return count;
}
```

Invalid. Not consistent.

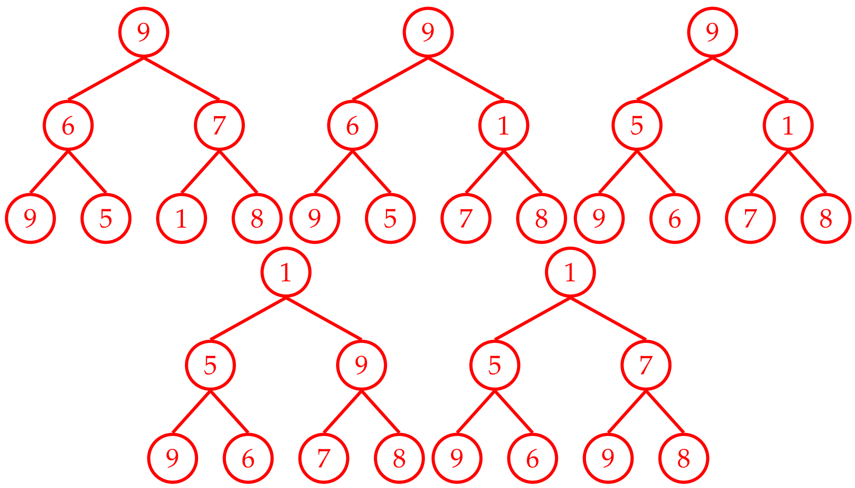
(e)
```
public int hashCode() {
    return 4;
}
```

Valid, but causes collisions on any input.

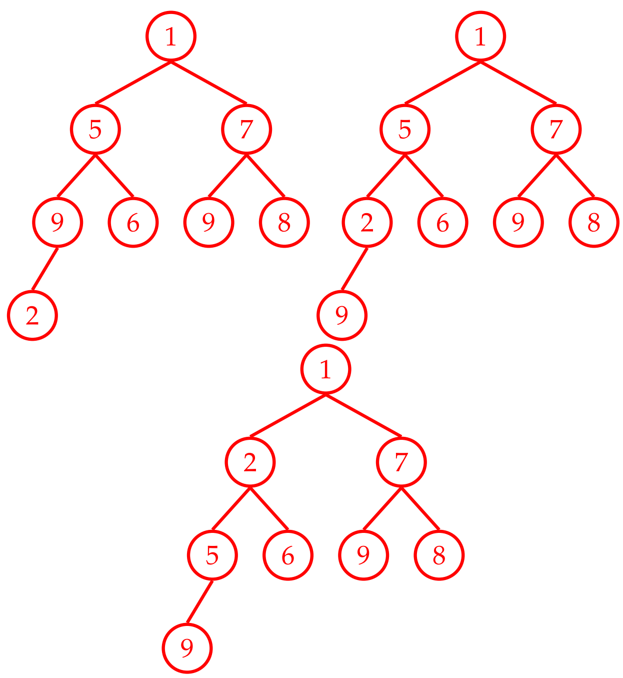
4 Min-Heapify This



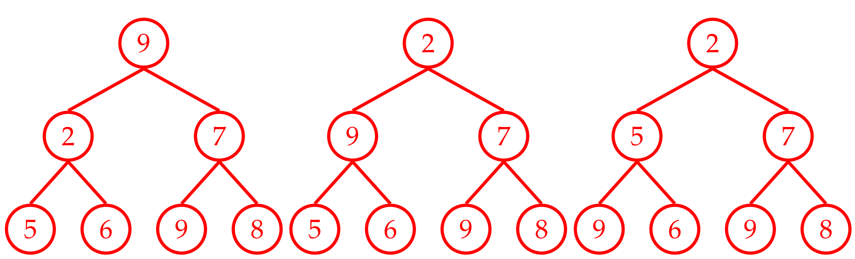
4.1 (a) Show the heapification of the tree.



(b) Now, insert the value 2.



(c) Finally, remove the value 1.



5 Merging Sorted Lists *Extra for Experts*

5.1 Given the following sorted lists as inputs, how we can efficiently return a sorted list containing all the elements?

```
inputs: {1, 2, 3, 7}, {3, 4, 5, 6}, {2, 4, 6, 8}, {0, 1, 2, 4}, {1, 1, 6, 9}
output: {0, 1, 1, 1, 1, 2, 2, 2, 3, 3, 4, 4, 4, 5, 6, 6, 6, 7, 8, 9}
```

Use a priority queue of iterators where the priority is given by `peek()`, or the value that would be returned from calling `next()`.

5.2 Write `merge`, a procedure that merges K sorted lists of length N into a single sorted list in $O(NK \log K)$ time.

```java
public class ListIterator<T extends Comparable<T>> implements Iterator<T>,
                                                               Comparable<ListIterator<T>> {
    private List<T> list;
    private int index;
    public boolean hasNext() {
        return index < list.size();
    }
    public T next() {
        T value = list.get(index);
        index += 1;
        return value;
    }
    public T peek() {
        return list.get(index);
    }
    public int compareTo(ListIterator<T> other) {
        return this.peek().compareTo(other.peek());
    }
    public static <T extends Comparable<T>> List<T> merge(List<T>[] lists) {
        List<T> result = new ArrayList<>();
        PriorityQueue<ListIterator<T>> remaining = new PriorityQueue<>();
        for (List<T> list : lists) {
            ListIterator<T> iterator = new ListIterator<>(list);
            if (iterator.hasNext()) {
                remaining.offer(iterator);
            }
        }
        while (!remaining.isEmpty()) {
            ListIterator<T> iterator = remaining.poll();
            result.add(iterator.next());
            if (iterator.hasNext()) {
                remaining.offer(iterator);
            }
        }
        return result;
    }
}
```

6 Caching *Extra for Experts*

6.1 You are a software engineer for a newspaper company! Your users are complaining about how slowly your website loads. After performing some performance profiling, you realize that the database queries are slowing the system down.

To fix the issue, you decide to implement a cache that contains the most recently accessed articles. The cache is only fast if it's small so you can only store a maximum of N articles. You want to keep only the N most recent articles that people have read. If a new, unique article is accessed, then the oldest article should be replaced.

Describe how you would implement this cache. What combinations of data structures would you use to build this efficiently?

Use a `HashMap` with references to nodes in a doubly linked list. The `HashMap` will map the name of the article to a tuple containing the article as well as a reference to a node in a doubly linked list. If the article we are trying to access is in the `HashMap`, we deliver the article, and then retrieve its respective node in the linked list and move it to the front. If the article is not in the `HashMap`, and the size of the `HashMap` is still less than N, we can fetch from the main server and create a new node to append to the front of the linked list. If the `HashMap` is at capacity, we look at the tail of the linked list and delete that article from both the linked list as well as the `HashMap`. We then add the new article into the `HashMap` and to the front of the linked list. In Java there's actually a data structure for this: `java.util.LinkedHashMap`!