CS 600.226: Data Structures

Michael Schatz

Oct 24, 2018

Lecture 23. Heaps and Priority Queues

HW5

Assignment 5: Six Degrees of Awesome

Out on: October 17, 2018
Due by: October 26, 2018 before 10:00 pm
Collaboration: None
Grading:

- Packaging 10%,
- Style 10% (where applicable),
- Testing 10% (where applicable),
- Performance 10% (where applicable),
- Functionality 60% (where applicable)

Overview

The fifth assignment is all about graphs, specifically about graphs of movies and the actors and actresses who star in them. You'll implement a graph data structure following the interface we designed in lecture, and you'll implement it using the incidence list representation.

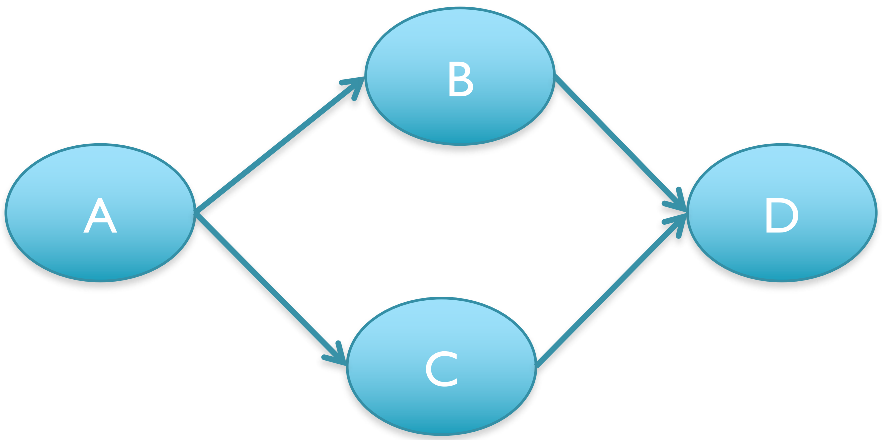
Turns out that this representation is way more memory-efficient for sparse graphs, something we'll need below. You'll then use your graph implementation to help you play a variant of the famous Six Degrees of Kevin Bacon game. Which variant? See below!

Agenda

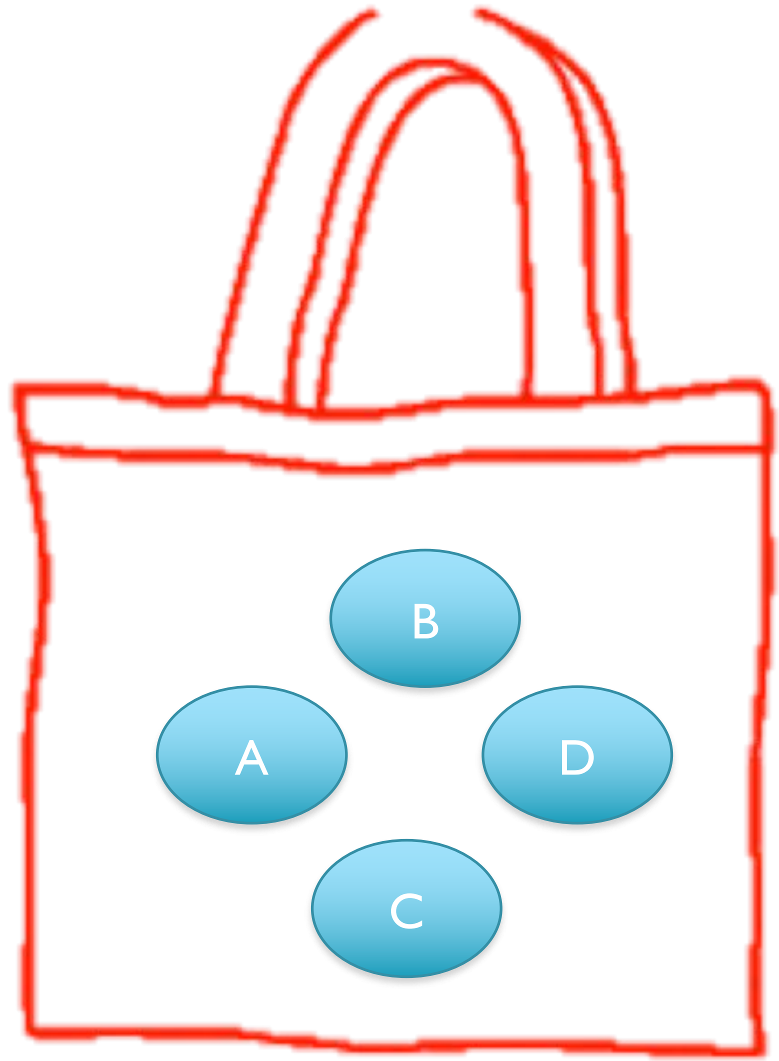
1. ***Recap on Sets, Self-Organizing Sets, and Ordered Sets***
2. ***Priority Queues***
3. ***Heaps***

Part 1.1:Sets

Graphs versus Sets



Position-Based



Value-Based

Set Interface

```
public interface Set<T> implements Iterable<T> {
    void insert(T t);
    void remove(T t);
    boolean has(T t);

    boolean empty();
    T any() throws EmptySetException;

    Iterator<T> iterator();
}
```

Now we can actually get all the values without destroying the set ☺

ArraySet

```
private int find(T t) {
    for (int i = 0; i < this.length; i++) {
        if (this.data[i].equals(t)) { return i; }
    }
    return -1;
}

public void remove(T t) {
    int position = this.find(t);
    if (position == -1) {return; }
    for (int i = position; i < this.length -1; i++) {
        this.data[i] = this.data[i+1];
    }
    this.length -= 1;
}

public boolean has(T t) {
    return this.find(t) != -1;
}

public void insert(T t) {
    if (this.has(t)) { return; }
    if (this.length == this.data.length) { this.grow(); }
    this.data[this.length] = t;
    this.length += 1;
}
```

Part 1.2: Self-Organizing Sets

Can we make these go faster?

Consider the input:

```
1 2 3 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 …
```

Can we change has() to speed it up?

```
class ArraySet
 private int find(T t) {
    for (int i = 0; i < this.length; i++) {
      if (this.data[i].equals(t)) {
        return i;
      }
    }
    return -1;
  }
  public boolean has(T t) {
    return this.find(t) != -1;
  }
  public void insert(T t) {
    if (this.has(t)) { return; }
  …
```

```
class ListSet
  private Node<T> find(T t) {
    for (Node<T> n = this.head;
         n != null;
         n = n.next) {
      if (n.data.equals(t)) { return n; }
    }
    return null;
  }
  public boolean has(T t) {
    return this.find(t) != null;
  }
  public void insert (T t) {
    if (this.has(t)) { return; }
  …
```

Can we change the internal array/list to speed it up?

Yes ☺

```
5 1 2 3 4
```

<= Will be about 5x faster for (very) long runs of 5

Performance Heuristics

Move-to-front Heuristic:

- If we are asked for find X and we do actually find it, we move that element to the front of the array or list so that it can be more quickly found next time
- **Example:**
  - If we start with [1 2 3 ... X ...],
  - After find(X) we shift the data to be [X 1 2 3 ...] instead.

How do you implement move-to-front on a ListSet()?

find() checks the data and if it matches the user data, stores a reference to that node in a new variable, and temporarily remove from the list. Then insert that node as the new beginning of the list.

What is the new complexity of find()?

Walk the monkey bars in O(n) to find the node, move to front in O(1) ☺

Any other issues?

ListSet iterator becomes very complex, don’t do it :-(

Performance Heuristics

Move-to-front Heuristic:

- If we are asked for find X and we do actually find it, we move that element to the front of the array or list so that it can be more quickly found next time
- **Example:**
  - If we start with [1 2 3 ... X ...],
  - After find(X) we shift the data to be [X 1 2 3 ...] instead.

How do you implement move-to-front on a ArraySet()?

find() checks the data in the node. If node has the user data do what?

Swapping to the front wont work because on the next round it may get swapped back to the end

Sliding to the front works correctly, but doubles the runtime for find() :-(

Any other ideas?

Like bubblesort, on a successful find() we can shift it forward by one slot

This is called a transpose

Performance Heuristics

Transpose Heuristic:

- If we are asked for find X and we do actually find it, we move that element up one closer to the front

- **Example:**
  - Start: [1 2 3 4 5 6]
  - Asked for 5, we swap 4 and 5: [1 2 3 5 4 6].
  - Asked for 5 agai
  - Asked for 2: [2 1

```
private int find(T t) {
  for (int i = 0; i < length; i++) {
    if (this.data[i].equals(t)) {
      if (i > 0) {
        T x = this.data[i];
        this.data[i] = this.data[i-1];
        this.data[i-1] = x;
        return i-1;
      }
    return i;
    }
  }
  return -1;
}
```

Over time, values that are “more popular” will take less time to find than values that are “less popular”.

UniqueArray vs UniqueTranspose

```
## Input the numbers 1 through 100,000
## Then 100,000 copies of 100,000

$ time ((seq 1 100000; jot -b 100000 100000)
 | java Unique > /dev/null)

real    0m36.439s
user    0m37.131s
sys     0m0.410s
```

```
$ time ((seq 1 100000; jot -b 100000 100000)
 | java UniqueTranspose > /dev/null)

real    0m25.987s
user    0m26.785s
sys     0m0.381s
```

Part 1.3: Ordered Sets

Set Interface

```
public interface Set<T> implements Iterable<T> {
    void insert(T t);
    void remove(T t);
    boolean has(T t);
}
```

Only operation is if t1.equals(t2)

Cannot make a faster Set other than some performance heuristics

```
public interface OrderedSet <T extends Comparable<T>>
                                       extends Iterable<T> {
    void insert(T t);
    void remove(T t);
    boolean has(T t);
}
```

We can compare if t1 < t2, t1 = t2, or t1 > t2

We can make OrderedSets go much faster! (how)?

Use binary search type techniques

OrderedArrayListSet

Lets extend ArrayListSet to an OrderedArrayListSet. find() will always return the correct index for the value, regardless of whether the value is in the set or not

What is the "correct index" for a value? Here are the possible cases:

1. The set was ***empty*** before this insertion. The "correct index" for the value is ***0*** in this case, the first spot in the array.
2. During our linear search, we find the ***first value that's greater than*** the value we're asked to insert. The "correct position" is "before that greater value" but because of the way the add(int index, E element) method works on ArrayList<E>, we want to use the index of that "greater value" itself.
3. Our linear ***search finishes without finding*** a greater value. The "correct index" is "the length of the array" => append the value at the end.

```
public class OrderedArrayListSet<T extends Comparable<T>>
                                        implements OrderedSet<T> {
private int find(T t) {
  for (int i = 0; i < this.data.size(); i++) {
    if (this.data.get(i).compareTo(t) >= 0) {
      return i;
    }
  }
  return this.data.size();
}
```

Testing!

```
private void printData() {
  for (int i = 0; i < this.data.size(); i++) {
    System.out.println("data[" + i + "]: " + this.data.get(i));
  }
}

public static void main(String[] args) {
  OrderedArrayListSet<Integer> s = new OrderedArrayListSet();
  s.insert(42);
  s.insert(100);
  s.insert(3);
  s.insert(200);
  s.insert(1);
  s.printData();
}
```

```
$ java OrderedArrayListSet
data[0]: 1
data[1]: 3
data[2]: 42
data[3]: 100
data[4]: 200
```

Data are in sorted order :-)

OrderedArrayListSetFast

We claimed OrderedSet was better because they could go much faster, but we are still using a linear scan to find anything. How can we do better?

```
public class OrderedArrayListSet<T extends Comparable<T>>
                                  implements OrderedSet<T> {
private int find(T t) {
  for (int i = 0; i < this.data.size(); i++) {
    if (this.data.get(i).compareTo(t) >= 0) {
      return i;
    }
  }
  return this.data.size();
}
```

With a binary search, find() will complete in O(lg n) instead of O(n)

If there are 1K items to search, will only take 10 steps to find ☺
If there are 1M items to search, will only take 20 steps to find ☺ ☺
If there are 1B items to search, will only take 30 steps to find ☺ ☺ ☺

Insert() will still need to slide things over in O(n) but at least we will find the correct position in O(lg n) time

Binary Search

Binary search is conceptually easy to understand, but notoriously difficult to implement correctly. Famously first described in 1946, but not correctly published until 1961

Invariant: always searching within [l,u]

```
private int find(T t) {
  int l = 0, u = this.data.size()-1;

  while(l <= u) {
    int m = (l + u) / 2;

    if (this.data.get(m).compareTo(t) > 0) {
      u = m - 1;

    } else if (this.data.get(m).compareTo(t) == 0) {
      return m;

    } else {
      l = m + 1;
    }
  }
  return l;
}
```

While non-empty range

Pick midpoint, integer arithmetic

m > t, check first half excluding m

Eureka!

Must be in the bottom, excluding m

Not found, l > u

Testing

$ seq 1 100000 | awk '{print int(rand()*100000)}' > rand100k.txt

```
$ time java UniqueArrayListSet < rand100k.txt > /dev/null

real    0m8.740s
user    0m9.053s
sys     0m0.369s
```

```
$ time java UniqueOrderedArrayListSetFast < rand100k.txt > /dev/null

real    0m1.199s
user    0m2.005s
sys     0m0.260s
```

Substantial speedups by replacing one function with another

Much more substantial than the move-to-front heuristics we saw

Part 2: Priority Queues

Queues

Whenever a resource is shared among multiple jobs:

- accessing the CPU
- accessing the disk
- Fair scheduling (ticketmaster, printing)

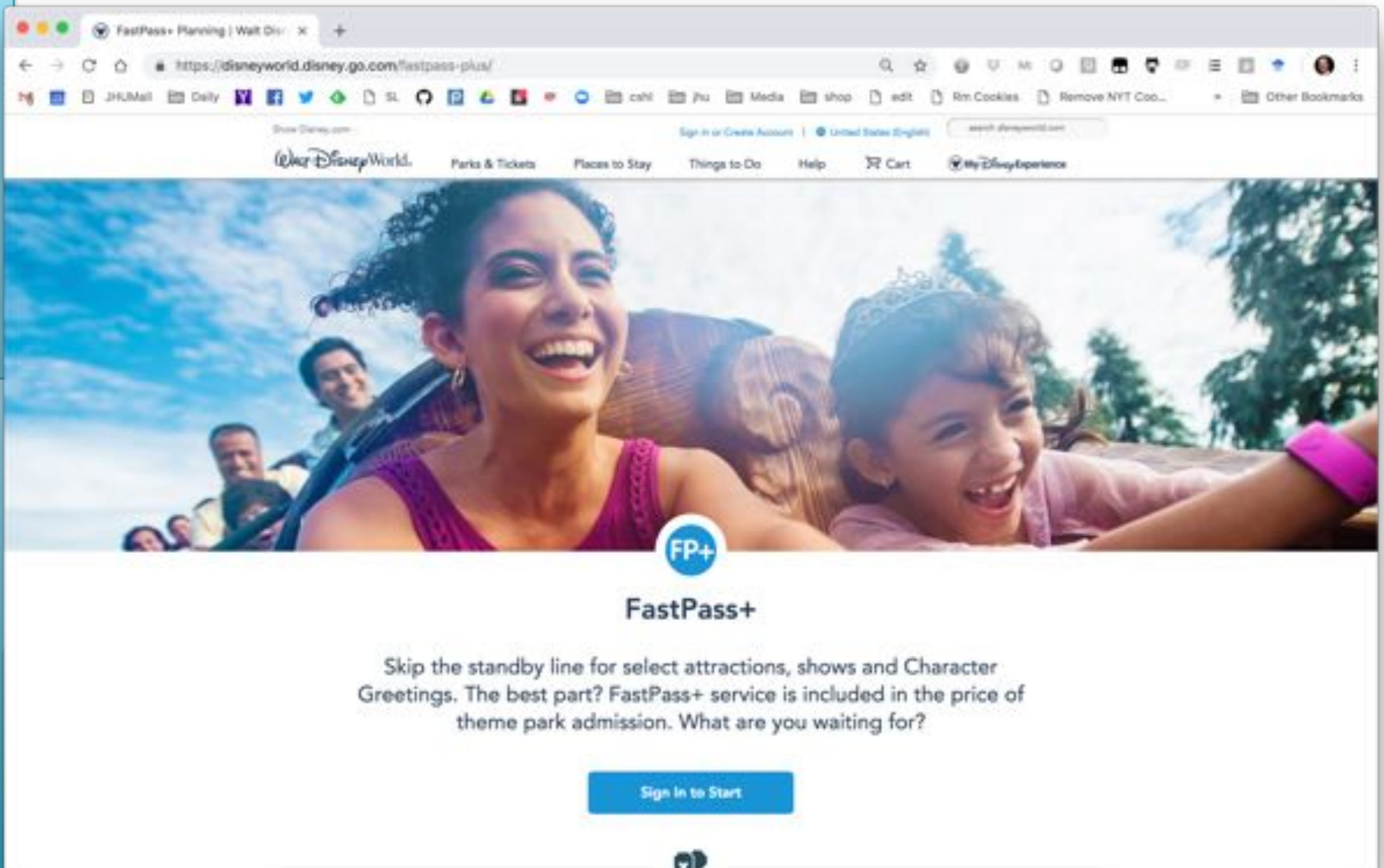
Whenever data is transferred asynchronously (data not necessarily received at same rate as it is sent):

- Sending data over the network
- Working with UNIX pipes:
    - ./slow | ./fast | ./medium

Also many applications to searching graphs (see 3-4 weeks)

FIFO: First-In-First-Out
Add to back +
Remove from front

Priority Queues



Priority Queue Interface

```
public interface PriorityQueue<T extends Comparable<T>> {
    void insert(T t);
    void remove() throws EmptyQueueException;
    T top() throwsEmptyQueueException;
    boolean empty();
}
```

Similar to a regular Queue, except the top() returns the ”largest” item rather than the first item inserted (top() instead of front())

```
pq.insert(42);
pq.insert(3);
pq.insert(100);
while (!pq.empty()){
  System.out.println(pq.top());
  pq.remove();
}
```

```
Prints:

100
42
3
```

What data structure should we use to implement a PQ?

An OrderedSet (using Binary Search :-))

Although we would allow for duplicates in a PQ

Priority Queue of Fruit

What if we wanted to use a Priority Queue of Fruit

```
PriorityQueue<Fruit> fpq = new PriorityQueue<Fruit>();
fpq.insert(apple);
fpq.insert(tomato);
fpq.insert(grape);
while (!pq.empty()){
  System.out.println(pq.top());
  pq.remove();
}
```

Prints:	**Value:**
tomato	$58B
grape	$39B
apple	$32B

How is the sort order defined?

Fruit class must implement/extend the Comparable interface by implementing the compareTo() method.

```
public class Fruit {
  int compareTo(Fruit other) {
    return this.globalValue < other.globalValue;
  }
}
```

Priority Queue Sort Order

What if we wanted to retrieve Integers sorted from smallest to largest?

1. Rewrite the priority queue: MinPriorityQueue, MaxPriorityQueue ¯_(ツ)_/¯

2. Change the comparison function ☺

Integers implement the compareTo() method:

Returns the value 0 if this Integer is equal to the argument Integer; a value less than 0 if this Integer is numerically less than the argument Integer; and a value greater than 0 if this Integer is numerically greater than the argument Integer (signed comparison).

https://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html

Extend the Priority Queue interface to accept a functor (function object) to establish the sort order

```
Interface Comparator<T> {
    int compare(T o1, T o2)
    boolean equals(Object obj)
}
```

```
class SortAscending<T>
        implements Comparator<T> {
    int compare(T o1, T o2) {
        //return o1.compareTo(o2)
        return o2.compareTo(o1);
    }
}
```

```
PriorityQueue<> p = new PriorityQueue<Integer>(new SortAscending());
```

Priority Queue Implementation

```
pq.insert(42);
pq.insert(3);
pq.insert(100);
while (!pq.empty()){
  System.out.println(pq.top());
  pq.remove();
}
```

```
f[]b
f[42]b
f[42,3]b
f[100,42,3]b
f[42,3]b
f[3]b
f[]b
```

PQ implemented with an OrderedArrayListSet has some hidden costs:
Insert: O(lg n + n) time to find() then slide into correct location
Remove: O(n) time: slide items over

What can we do to improve this?

```
b[]f
b[42]f
b[3,42]f
b[3,42,100]f
b[3,42]f
b[3]f
b[]f
```

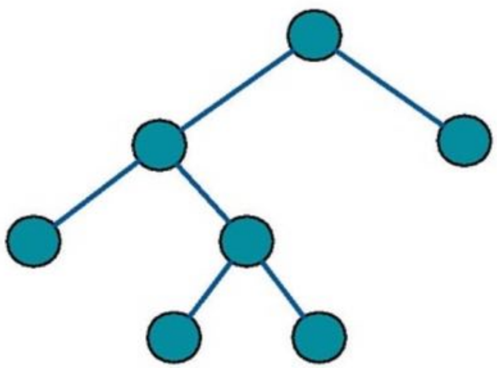
Ordering from back to front in the array allows for O(1) remove(), although insert() will remain at O(lg n + n)

What else can we do?

Do we need all the items sorted all the time?

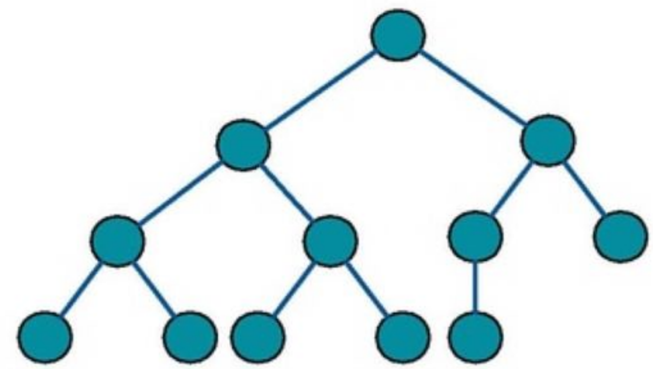
Part 3: (Binary) Heaps

Special Trees



Height of root
= 0

Total Height
= 3



Full Binary Tree
Every node has
0 or 2 children

Complete Binary Tree
Every level full, except
potentially the bottom level

What is the maximum number of leaf nodes in a complete binary tree? 2^h

What is the maximum number of nodes in a complete binary tree? $2^{h+1} - 1$

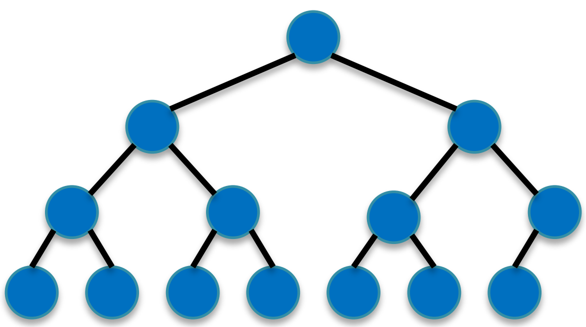
What fraction of the nodes in a complete binary tree are leaves? about half

What is the height of a balanced binary tree? lg n

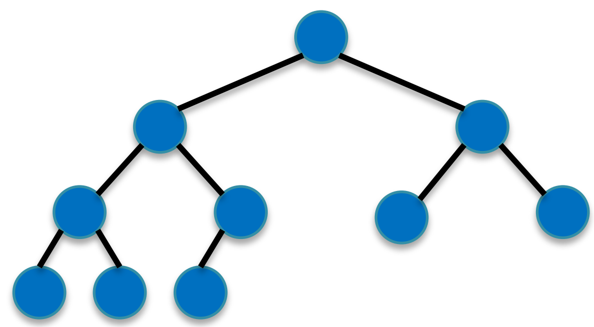
Binary Heaps

Shape Property:

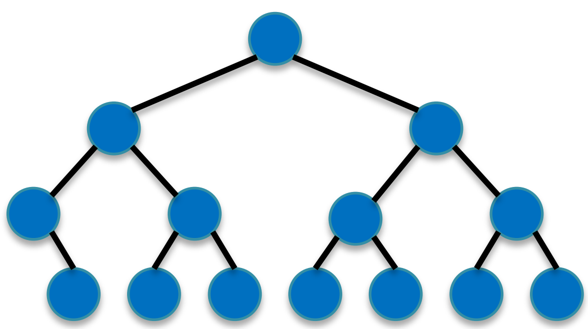
Complete binary tree with every level full, except potentially the bottom level, **AND** bottom level filled from left to right



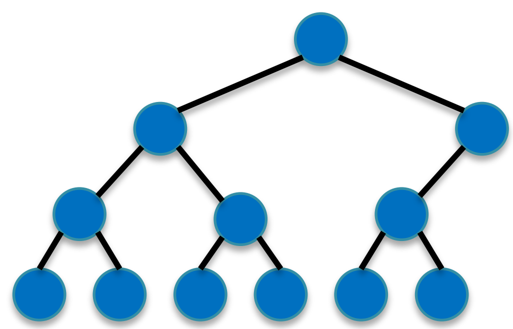
Valid



Valid



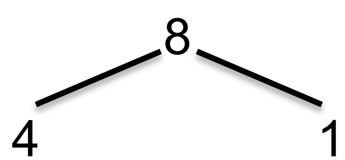
Invalid



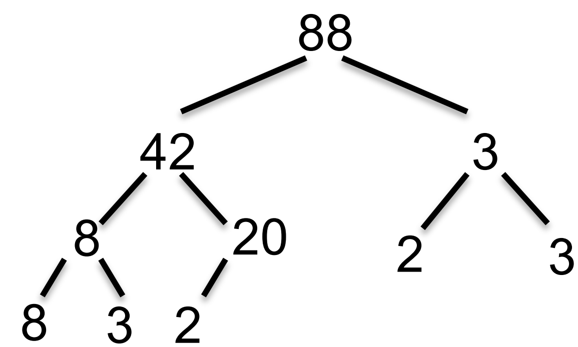
Invalid

Binary Heaps

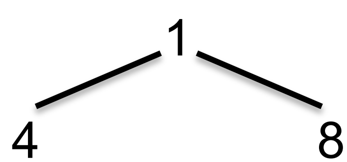
Ordering Property:
The value of each node is greater than or equal to the value of its children,
BUT there is no ordering between left and right children



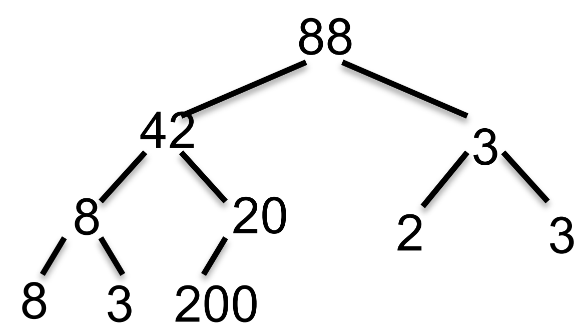
Valid



Valid

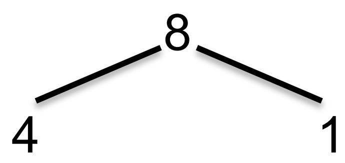


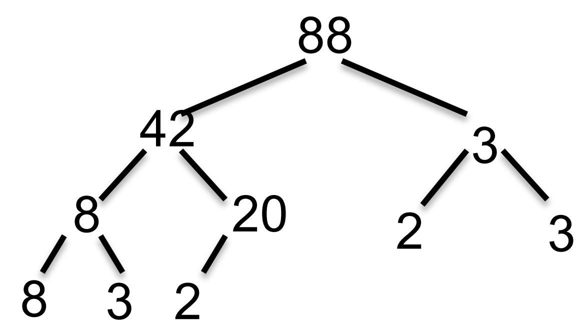
Invalid



Invalid

Binary Heaps





What does the shape property imply about the height of the tree?

Guaranteed to be lg n ☺

What does the ordering property imply about the top() of the tree?

Guaranteed max value will be in the root node

That's interesting, I wonder if we could use this for a priority queue…

… just need to efficiently insert() and removeTop()

Inserting into a binary heap

Insert the elements 8, 2, 7, 4

```
null    8       8        8          8          8
               /        / \        / \        / \
              2        2   7      2   7      4   7
                                 /          /
                                4          2

       i(8)    i(2)     i(7)             i(4)
```

The ***shape property*** tells us that we need to fill one level at a time, from left to right. So the ***number of elements*** in a heap ***uniquely determines where the next node*** has to be placed.

What about the ***ordering property***? When we insert 4, the parent 2 is not ≥ 4, so the ***ordering property is violated***. There's an ***easy fix*** however, just swap the values!

Note that in general, we ***may need to keep swapping "up the tree"*** as long as the ordering property is still violated. ***But since there are only log n levels, this can take at most O(log n) time in the worst case.***

Remove top from a binary heap

Remove the top

```
      8
     / \
    4   7
   /
  2
```

```
    4   7
   /
  2
```

ERROR:
2 trees

```
    4
   / \
  2   7
```

ERROR:
4 < 7

```
      7
     /
    4
   /
  2
```

ERROR:
Shape Violation

Any ideas?

1.Swap last

```
      2
     / \
    4   7
   /
  8
```

2. Remove last

```
    2
   / \
  4   7
```

3. Swap down from root with larger child

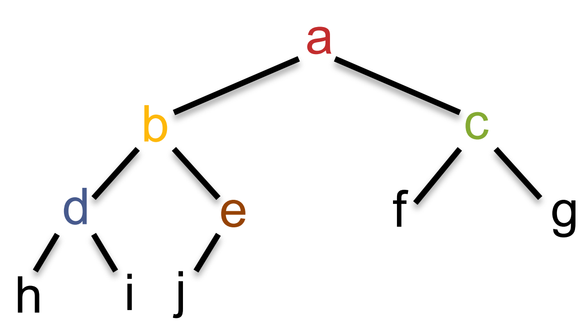
```
    7
   / \
  4   2
```

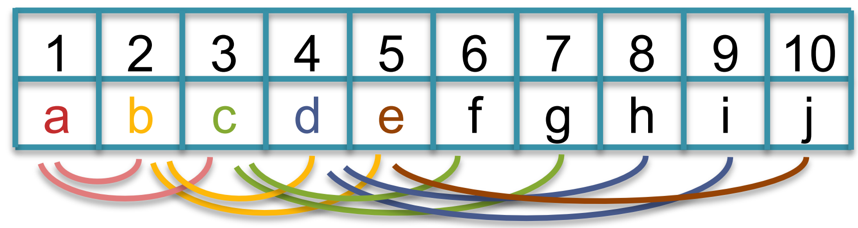
Note that in general, we ***may need to keep swapping "down the tree"*** as long as the ordering property is still violated. ***But since there are only log n levels, this can take at most O(log n) time in the worst case.***

Heap Implementation

We could implement a heap as a tree with references, but those references take up a lot of space and are relatively slow to resolve

Lets encode the tree inside an array!





Encoding a complete tree into the array in <u>level order</u>
puts the children and parent in predictable locations
(Math is easier if the array starts at 1 instead of 0)

Parent(i) = array[i/2]
Parent(f) = parent(6) = array[6/2] = array[3] = c

left(i) = array[i*2] & right(i) = array[i*2+1]
left(3) = array[3*2] = array[6] = f & right(3) = array[3*2+1] = array[7] = g

Heap-based Priority Queue

```
pq.insert(42);
pq.insert(3);
pq.insert(100);
while (!pq.empt
  System.out.pr
  pq.remove();
}
```

```
[]
            add 42 at end & upheap
[42]
            add 3 at end & upheap
[42,3]
            add 100 at end
[42,3,100]
            upheap 100
[100,3,42]
            remove top: swap root
[42,3,100]
            remove top: remove last & downheap
[42,3]
            remove top: swap root
[3,42]
            remove top: remove last & downheap
[3]
            remove top
[]
```

Heap-based Priority Queue

```
pq.insert(42);
pq.insert(3);
pq.insert(100);
while (!pq.empt
  System.out.pr
  pq.remove();
}
```

```
[]
                add 42 at end & upheap
[42]
                add 3 at end & upheap
[42,3]
                add 100 at end
[42,3,100]
                upheap 100
[100,3,42]
                remove top: swap root
[42,3,100]
                                                heap
                remove top: swap root
                remove top: remove last & downheap
                remove top
[]
```

Seems a little complicated, but each insert completes in O(lg n) and each remove completes in O(lg n) ☺

How could you use this for a general sort routine?

Add all elements in O(n lg n); remove in sorted order in O(n lg n)

Total time for HeapSort: O(n lg n) ☺

UniqueQueue

```java
import java.util.Scanner;

public final class UniqueQueue {
    private static PriorityQueue<Integer> data;
    private UniqueQueue() { }

    public static void main(String[] args) {
        data = new BinaryHeapPriorityQueue<Integer>();
        Scanner scanner = new Scanner(System.in);

        while (scanner.hasNextInt()) {
            int i = scanner.nextInt();
            data.insert(i);
        }

        Integer last = null;

        while (!data.empty()) {
            Integer i = data.remove();
            if (last == null || i != last) {
                System.out.println(i);
            }
            last = i;
        }
    }
}
```

Since data are in sorted ordered, just check to see current if different from last item

Testing

```
$ seq 1 1000000 | awk '{print int(rand()*1000000)}' > rand1000k.txt
```

```
$ time java UniqueOrderedArrayListSetFast < rand1000k.txt > /dev/null

real    0m18.386s
user    0m19.258s
sys      0m0.698s
```

```
$ time java UniqueQueue < rand1000k.txt > /dev/null

real    0m5.785s
user    0m6.912s
sys      0m1.023s
```

Substantial speedups replacing OrderedSet (with binary search but slow insert) with Heap-based Priority Queue (with O(n lg n) overall time) ☺ ☺ ☺

Next Steps

1. Work on HW5

2. Check on Piazza for tips & corrections!