Computer Science Midterm 2 Cheat Sheet

by Paloma via cheatography.com/55343/cs/15271/

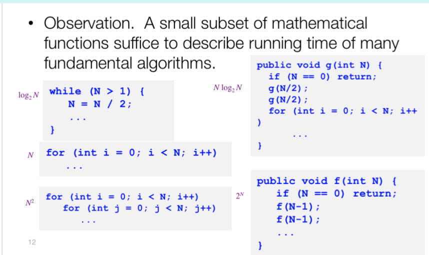
Some Helpful Big Oh Analysis

Expansion	Summation	Big Oh
1+2+3+4+....+ N	N(N+1)/2	O(N^2)
N+N+N+....+N	N*N	O(N^2)
N+N+N+....+N+....+N+. ...+N	3N*N	O(N^2)
1+2+4+....+	2^(N+1)-1	O(2^N)

2^10 = 1024 ~~ 1000

O- notation is an upper bound so N is O(N) but it is also O(N^2)

Order of Growth Classifications

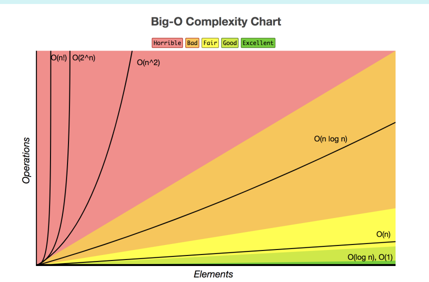


Usually, nested for loops have a big O(N^2) because each of them runs n times. However, sometimes they can run less than n times.

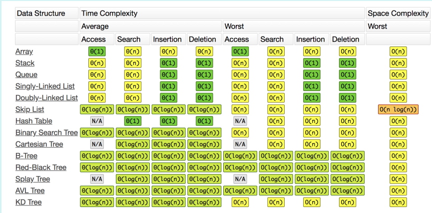
for (int i =0; i<N; i++) ---> N times
for (int j =1; j<n; j - j*2)

Big O is n* log n times

Big Oh Complexity



Common Data Structure Operations

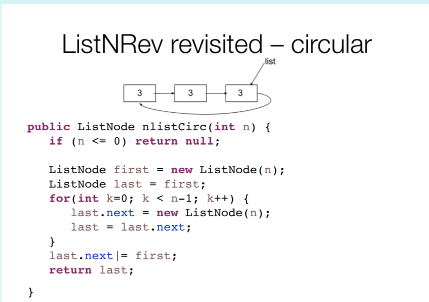


Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

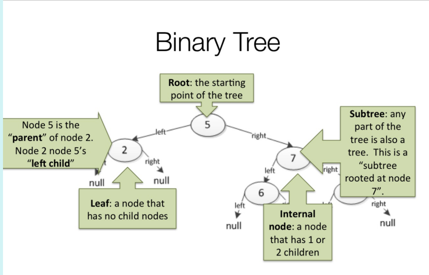
Arrays vs ArrayLists

Arrays	ArrayLists
They have a fixed size	Size can change
Much faster to add to	Adding to an arraylist is usually N
	But when you reach the max, the computer doubles the limit every time you hit the limit so it takes O(N) times --> This is why it takes longer

Circular Linked Lists

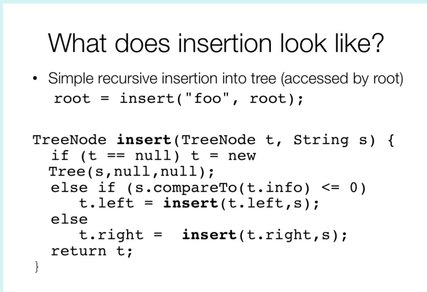


Binary Search Tree

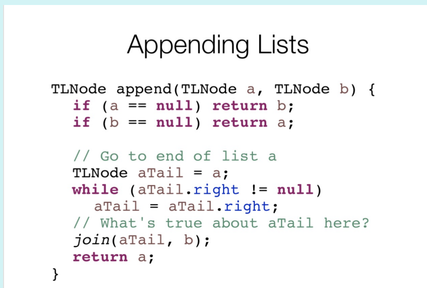


- Each node has a value
- Nodes with the values less than their parent are in the left
-Nodes with values greater than their parent are in the right subtree
- If equal, choose a side and stay consistent
- Insert from top of binary search tree and move down

Binary Tree Insertion



Appending Lists



BSTS to Lists

Tree to List

- How do do we update

```
public static TLNode treeToList(TLNode root) {
      // base case
      if (root == null)
         return null;
      TLNode beforeMe = treeToList(root.left);
      TLNode afterMe = treeToList(root.right);

      // TODO What do you need to do here?

      return root;
   }
```

-**Trees:** are nodes with two pointers

-**Doubly linked lists:** also nodes with two pointers (allows for constant time access with one pointing to front and one pointing to back)

Complete Binary Tree

- Every non leaf node has two children
- All the leaves are at the same level
- There are 2^N -1 or O(2^N) nodes with N levels
- There are 2^{N-1} leaves with n levels

Priority Queues

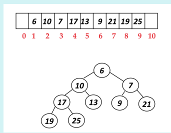
```
PriorityQueue<Integer> pq = new
    PriorityQueue<Integer>();
// add all elements from list to pq
for (int elem : list)
    pq.add(elem);
for (int index = 0; pq.size() > 0; index++)
    // remove minimum remaining element
    list[index] = pq.poll();
```

- Minimum is first out
-Poll means remove the minimum each time
-List [0] will be smallest
-List [1] is smallest of all the ones that remain
-While a queue is first in first out, a priority queue is minium out first
-Shortest path

Heaps

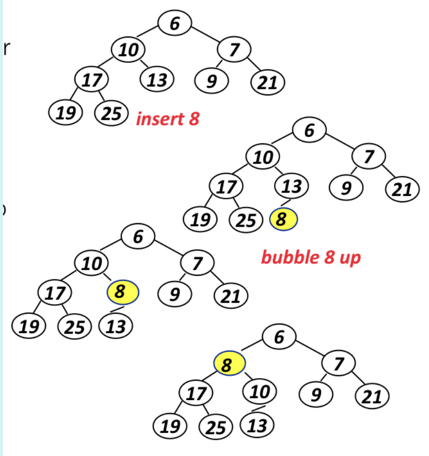
- Heap is an array-based implementation of a binary tree used for implementing priority queues and supports: insert, findMin, and deleteMin
-Using array minimizes storage (no explicit pointers) , faster too because children are locatd by index/position in array
Deletion: remove root and replace with right most child and then bubble down filling left to right
-**Properties:**
- **shape:** tree filled at all levels (except perhaps last) and filled in left-to-right (complete binary tree
- **value:** each node has value smaller than both children
Min Heap:
- Minimal element is at root, index 1
-Maximal element has to be a leaf, because can't be greater than child
-Complexity of finding maximal elements, half nodes are trees --> O(n/2) so O(n)
- Second smallest element must be one level below root

Using An Array For a Heap



- Store node values in array starting at index 1
- For node with index k:
- **left child:** index 2*k
-**right child:** index 2*k +1
- **parent:** index k/2

Adding Values to Heap



- To maintain heap shape, must add new value in left-to-right order of last level
- This could violate heap property
- move value "up" if too small
- Change places with parent if heap property is violated and stop when parent is smaller and stop when root is reached
-Pull parent down

Heap Add Implementation

```
void add(ArrayList<Integer> list,
           int elt) {
  // add elt to heap in myList
  list.add(elt);
  int loc = list.size();

  while (1 < loc &&
           elt < list.get(loc / 2)) {
     list.set(loc, list.get(loc/2))
     // go to parent
     loc = loc / 2;
  }
  // What's true here?
  list.set(loc, elt);
}
```

Tries

- Tries support add, contains, delete in O(w) time for words of length w
- Each node in a trie has a subtrie for every valid letter than can follow
-

Priority Queue Implementations

	Insert average	Getmin (peek)	Insert worst	Getmin (delete)
Unsorted ArrayList	O(1)	O(n)	O(1)	O(n)
Sorted ArrayList	O(n)	O(1)	O(n)	O(1)/ O(n)
Heap	O(log n)	O(1)	O(log n)	O(log n)
Balanced binary search tree	O(log n)	O(log n)/ O(1)	O(log n)	O(log n)

- Heap has O(1) find-min (no delete) and O(n) build heap

Operations: O(log n)
add - add element to last spot and bubble up
remove/poll - remove root.min and take last element and bubble down

Graphs Vocabulary

- A collection of vertices and edges
- Edge connections two vertices
- Direction can be imported, directed edge, directed graph
- Edge may have associated weight/cost
- A vertex sequence is a path where vk and vk+1 are connected by an edge
- If some vertex is repeated, the path is a cycle
- A graph is connected if there is a path between any pair of vertices
-Articulation Point breaks graph in two

Graphs DFS

Depth-first search on Graphs

```
public Set<Graph.Vertex> dfs(Graph.Vertex start){
    Set<Graph.Vertex> visited = new TreeSet<Graph.Vertex>();
    Stack<Graph.Vertex> qu = new Stack<Graph.Vertex>();
    visited.add(start);
    qu.push(start);

    while (qu.size() > 0){
        Graph.Vertex v = qu.pop();
        for(Graph.Vertex adj : myGraph.getAdjacent(v)){
            if (! visited.contains(adj)) {
                visited.add(adj);

                qu.push(adj);
            }
        }
    }
    return visited;
}
```

Envision each vertex as a room
Go into a room, mark the room, choose an unused door, exit
Don't go into room you've already been in--> explore every vertex one time
qu is where we're going, visited is where we've been

Adjacency Lists and Matrix

For example, consider the following graph:

```
  0
 / \
1---2
 \ /
  3
```

The adjacency list is:

```
{[1,2],
[0,2,3],
[0,1,3],
[1,2]}
```

And the adjacency matrix is:

```
[[FTTF],
 [TFTT],
 [TTFT],
 [FTTF]]
```

Where F and T represent boolean variables.

Adjacency List: V+E spaces
Adjacency Matrix: V*E

Sorting

- Simple, $O(n^2)$ sorts --- for sorting n elements
  - Selection sort --- n^2 comparisons, n swaps, easy to code
  - Insertion sort --- n^2 comparisons, n^2 moves, stable, fast
  - Bubble sort --- n^2 everything, slow, slower, and ugly
- Divide and conquer faster sorts: $O(n \log n)$ for n elements
  - Quick sort: fast in practice, $O(n^2)$ worst case
  - Merge sort: good worst case, great for linked lists, uses extra storage for vectors/arrays
- Other sorts:
  - Heap sort, basically priority queue sorting, Big-Oh?
  - Radix sort: doesnt compare keys, uses digits/characters $O(dn+kd)$
  - Shell sort: quasi-insertion, fast in practice, non-recursive $O(n^{1.5})$

Creating Adjacency Matrix

```
public int howLong(String [] connects, String [] costs){
    int [] [] adjMatrix = new int [connects.length] [connects.length]'
    for (int i =0; i<connects.length; i++){
        String [] edges - connects[i[.split(" ");
        String [] weights = costs[i].split(" ");
            for (int j =0; j<edges.length; j++){
                adjMatrix[i][Integer.partseInt(edges[j])) = Integer.parseInt(weights[j]);
```

Analysis: Empirical vs. Mathematical

Empirical Analysis	Mathematical Analysis
Measure running times, plot, and fit curve	Analyze algorithm to estimate # ops as a function of input size
Easy to perform experiments	May require advanced mathematics
Model useful for predicting, but not for explaining	Model useful for predicting and explaining
	Mathematical analysis is independent of a particular machine or compiler; applies to machines not yet built.

Comparators

Let's assume that the natural ordering of Employee instances is Name ordering (as defined in the previous example) on employee name. Unfortunately, the boss has asked for a list of employees in order of seniority. This means we have to do some work, but not much. The following program will produce the required list.

```
import java.util.*;
public class EmpSort {
    static final Comparator<Employee> SENIORITY_ORDER =
                                        new Comparator<Employee>() {
            public int compare(Employee e1, Employee e2) {
                return e2.hireDate().compareTo(e1.hireDate());
            }
    };

    // Employee database
    static final Collection<Employee> employees = ... ;

    public static void main(String[] args) {
        List<Employee> e = new ArrayList<Employee>(employees);
        Collections.sort(e, SENIORITY_ORDER);
        System.out.println(e);
    }
}
```

Comparators

- Can't always access comparable method (implements .compareTo and uses Collections.sort and Arrays.sort)
-Sometimes must implement comparators in which you pass two objects
- Must implement .compare(T a, T b)
- Return <0 when a<b
- Return ==0 when a ==b
Return >0 when a>b

Comparator Example

```
public class SortByFreqs {
    public class Fruits {
        String fruit;
        int count;
        public Fruits(String k, Integer v) {
            fruit = k;
            count = v;
        }



    public String getFruit() {
        return fruit;
    }

    public int getCount() {
        return count;
    }
    }
    public String[] sort(String[] data) {
        Map<String, Integer> myMap = new TreeMap<String, Integer>();
       for (int i= 0; i<data.length; i++) {
            if(!myMap.containsKey(data[i])) {
                myMap.put(data[i], 0);
            }if (myMap.containsKey(data[i])) {
                myMap.put(data[i], myMap.get(data[i])+1);
            }

       }
       int counter =0;
       Fruits [] fruit = new Fruits [myMap.size()];
       for (String k: myMap.keySet()) {
            fruit[counter] = new Fruits (k, myMap.get(k)) ;
            counter +=1 ;

       }

//              Comparator<Fruits> comp = Comparator.comparing(Fruits::getFruit).reversed();
//              comp = comp.thenComparing(Fruits::getCount);
                Comparator<Fruits>comp = Comparator.comparing(Fruits::getCount).reversed();
                comp=comp.thenComparing(Fruits::getFruit);
                Arrays.sort(fruit, comp);

            String [] array = new String[myMap.size()];
            for (int i=0; i<fruit.length; i++) {
                array[i]= fruit[i].getFruit();
    }
    return array;

}
```

Comparator Example

```
public class Dirsort {
    public class Directories{
        String folderName;
        Integer depth;
        public Directories(String k) {
            folderName=k;
            depth=counter(k);
        }
        public int counter(String k) {
            int counter=0;
            for (int i=0; i<k.length(); i++){
                if (k.charAt(i)=='/') {
                    counter+=1;
                }

            }return counter;

        }

        public String getFolderName() {
            return folderName;
        }
        public int getDepth() {
            return depth;
        }
    }
    public String[] sort(String[] dirs) {
        int counter =0;
        Directories [] directoriesOrdered= new Directories [dirs.length];
        for (String k: dirs) {
            directoriesOrdered[counter]= new Directories(k);
            counter +=1;
        }
        Comparator<Directories> comp = Comparator.comparing(Directories::getDepth);
        comp = comp.thenComparing(Directories::getFolderName);
        Arrays.sort (directoriesOrdered, comp);
        String [] array = new String[dirs.length];
        for (int i=0; i<directoriesOrdered.length; i++) {
            array[i]= directoriesOrdered[i].getFolderName();
        }
        return array;

    }
```

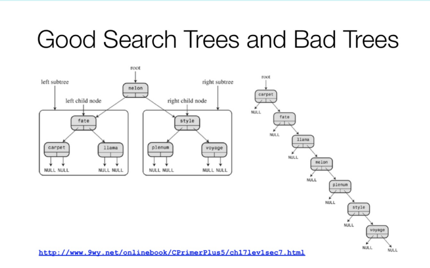
Linked List vs. ArrayList

Linked List	ArrayList	Both
Separate elements in memory that all have pointers to each other	A collection of elements in order in memory	Collection of elements
		Add, remove, for loops, sort themselves, clear

Linked List vs. ArrayList (cont)

Remove First Element:	N Time because you don't have to shift something when it is in the front of the list	N^2 Time because everything stores sequentially so when you take something out you have to shift everything by 1
Remove Middle Index	Has a higher coefficient and thus is slower: To get there takes time but to remove it is instantaneous :O(N)	Faster: To get to middle element is instantaneous but to remove it you have to shift it: O(N)
	Best for adding/removing front	Best for adding/removing something from back/middle

Trees



If you have N nodes, height is asking how many times you can divide by 2 --> expressed as the log base 2 of n

Good search tree is height is log n

Bad search tree is n

Balanced if left and right subtrees are height balanced and left and right heights differ by at most 1

Autocomplete

-**BruteAutocomplete:** stores data as a Term array and finds the top k matches by iterating through the array and pushes all terms starting with the prefix into a max priority queue sorted by weight and returns the top k terms from that priority queue

- not compared by weight and o organization

-topMatches: O(n+mlogm)

- topMatch: O(n)

-**Improving BruteAutocomplete:**

- had to iterate through every single term in the array because it did not know where the terms starting with the prefix were located aka array was unsorted.

- If we sort the array lexicographically, then all the terms with the same prefix will be adjacent (Sorting takes O(n log n)

Autocomplete (cont)

- **Term:** encapsulates a word/term and its corresponding weight

-**BinarySearchAutocomplete:** finds Terms with a given prefix by performing a binary search on a sorted array of Terms

-**TrieAutocomplete:** finds Terms with a given prefix by building a trie to store the Terms. To be efficient should only look at words whos maxsubtree weight is greater than the minimum

Autocomplete: Term class

- The term class encapsulates a comparable word weight pair

- **WeightOrder:** sorts in ascending weight order

-**ReverseWeightOrder:** sorts in descending weight order

-**PrefixOrder:** which sorts by the first r characters

-If one or both words are shorter than r, we just use normal lexicographical sorting

- compare method must take O(r)

Autocomplete: Binary Search

- Find all the range of all the terms comparator considers equal to key

- Quickly return the first and last index respectively of an element in the input array which the comparator considers to be equal to key

Autocomplete: Binary Search (cont)

- We specify the first and last index because there could be multiple Terms in which the comparator consider to be equal to key

- Collections.binary search does not guarantee first index of terms that match key, it gives an index

Autocomplete: Tries

- To completely eliminate terms which don't start with prefix, store in trie

- Navigate to the node representing the string. The trie rooted at this node only contains nodes starting with this trie

- No matter how many words are in our trie, navigating this node takes the same amount of time

Autocomplete: Big-Oh

Class	TopMatch	TopMatches
BruteAutocomplete	O(n)	O(mlogm + n)
BinarySearchAutocomplete	O(log(n) + m)	O(log(n) + (m + k)log(k))
TrieAutocomplete	O(w)	O(w)

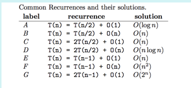
n: number of terms in total

m: number of terms that match the prefix

k: desired number of terms

w: number of letters in the word

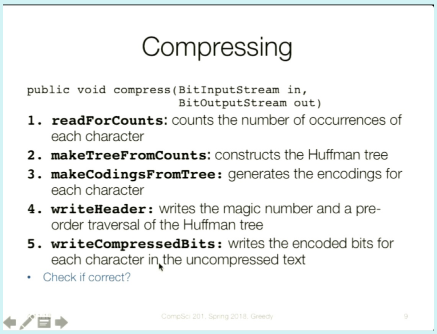
Common Recurrences and Their Solutions



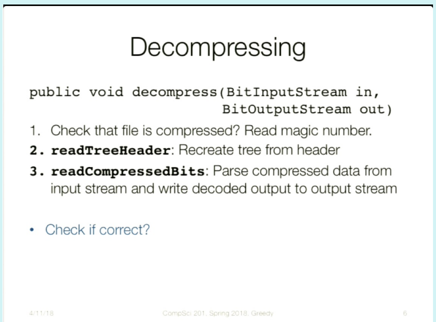
Common Recurrences and their solutions.

label	recurrence	solution
A	T(n) = T(n/2) + O(1)	$O(\log n)$
B	T(n) = T(n/2) + O(n)	$O(n)$
C	T(n) = 2T(n/2) + O(1)	$O(n)$
D	T(n) = 2T(n/2) + O(n)	$O(n \log n)$
E	T(n) = T(n-1) + O(1)	$O(n)$
F	T(n) = T(n-1) + O(n)	$O(n^2)$
G	T(n) = 2T(n-1) + O(1)	$O(2^n)$

Huffman: Compressing



Huffman: Decompressing



Graphs BFS

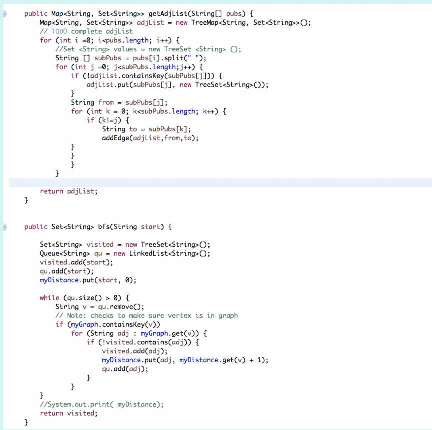
Visit everything that is one away, then everything that is two away...
Used to find shortest distance
takes a lot of space --> B^d

Bacon Number

Good Center - Has the most people closest to them
Chooses the best path, lowest number of edges
Actor Actor Representation//Vertices: actors or actresses//Edges: Two actors are adjacent (joined by a graph edge) if and only if they appear in the same movie
Movie Movie Representation// Vertices: Movies//Edges: Two movies are adjacent if they share a cast member
Actor Movie Representation//Vertices: Actors, actresses, and movies// Edges: An actor is connected to a movie if he or she appeared in that movie

1. Most vertices: Actor to movie
a. All of the vertices you had in actor to actor and all vertices in movie
to movie
2. Most edges: Actor to Actor

Erdos Number Part 2



Stacks

LIFO

Queue

FIFO

Recursion

Efficient sorting algorithms are usually recursive
Base Case: does not make a recursive call
For Linked Lists: Base case is always empty list or singular node/Recursive calls make progress toward base case (list.next as argument)

Percolation Overview

- System percolates if top and bottom are connected by open sites
- Given a NxN grid, where each is site is open with probability p*, what is the probability that the system percolates?
- if p>p*, system most likely percolates
- if p< p*, system does not percolate
-All simulations, whether using PercolationDFS, PercolationDFSFast, or Percolation UF with any implementation of union-find will be at least O(N^2)
- **Finding the threshold**
- Initialize NxN grid of sites as blocked
- Randomly open sites until system percolates
- Percentage of pen sites gives an approximation of p*
How do you get random cells to open and not open same shell more than once:
- Make points out of the cells

Percolation Overview (cont)

- Shuffle them gets a random ordering of all the point where each one
occurs once time
- Go through and repeatedly open each

Percolation Solution 1: Depth First Search

- Try searching from all of the open spots on the top row
- Search from all legal adjacent spots you have not visited
- Recurse until you can't search any further or have reached the bottom row
- Try all the legal adjacent spots (what makes it recursive is that we do the
same problem but at a different place)
- Base Cases:
- Out of bounds
- Blocked
- Already full/visited

PercolationDFS sets each grid cell to OPEN and runs a DFS from each open cell in the top (index-zero) row to mark the cells reachable from them as FULL. In the new model PercolationDFSFast, you'll make this implementation more efficient by only checking the cell being opened to see if it results in more FULL cells, rather than checking every cell reachable from the top row.

-Percolation DFS and DFSFast run in O(N) because it iterates through only the bottom row to check if it is full

Percolation DFSFast

- **Why is this an Improvement**: An improvement because we don't have to search from the top:
- Don't have to start from the top and go down
- For the cells that are adjacent, now search from that spot
- If one of my neighbors is full, I am full

-Don't have to redfs things you've already seen

Methodology:

Percolation DFS Fast

1. Create a grid
2. Set them all to blocked
3. Protected void updateOnOpen
4. Clear everything from being full
5. Dfs checks base cases

a. If not in bounds, return

b. If cell is full or not open, return

Otherwise try all neighbors recursively

Percolation Solution 2: Union- Find

- Create an object for each site (each cell) (Vtop as N*N, Vbottom as N^2 +1)
- Percolates if vtop is connected to vbottom
- One call that you have to make --> union find
- For every cell, give it an index
- Becomes problematic when n is too long

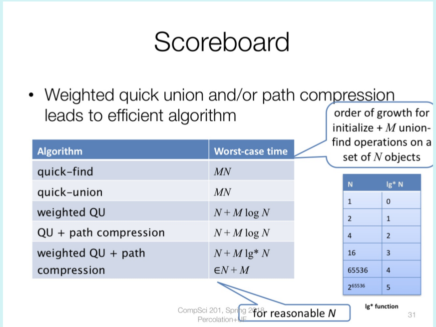
QuickUWPC:

- Look at ultimate parent making path short to find parent at constant time
- Run time is O(1) because we simply check if vtop and vbottom are in the same set

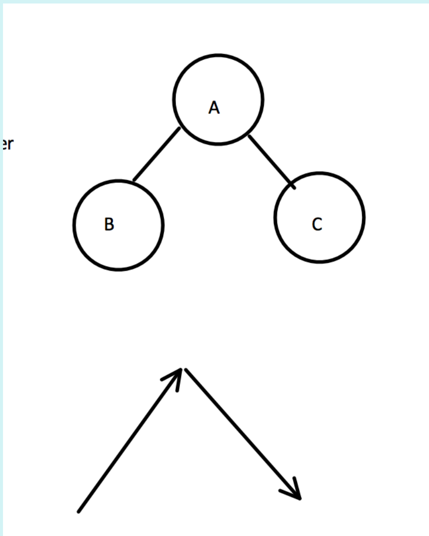
Percolation Solution 2: Union- Find (cont)

-IUnionFInd.find is called from both connected and union to find sets that p and q belong to

Percolation Method Score Board



Tree Traversals InOrder

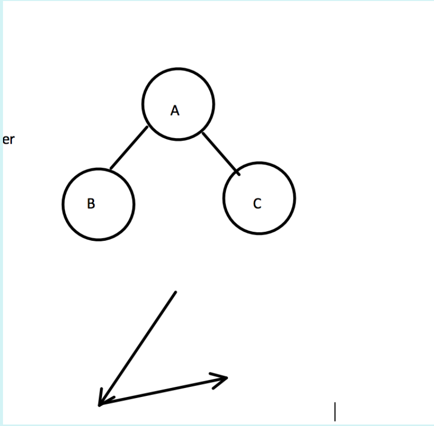


Visit left sub-tree, process root, visit right subtree

Increasing order

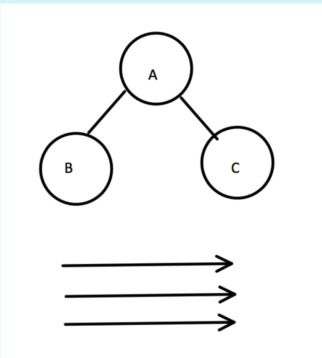
- Follow path and In order is when you do outline and you hit it the second time

Tree Traversals PreOrder



Process root, then visit left subtree, then visit right subtree
Good for reading/writing trees
- When you follow the outline and preorder is just when you hit it for the first time

Tree Traversals: PostOrder



Visit left subtree, right subtree, process root
Good for deleting trees
When you follow the outline and postorder is when you hit it going up

Recursion with ListNodes in return statement

```
public ListNode<Integer> convertRec
(ListNode<String> list):
if (list == null) return null;
return new ListNode<Integer>
(list.info.length,
convertRev(list.next);
```

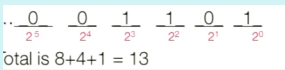
Doubly Linked Lists

```
List Node first = new ListNode
<"cherry", null, null>;
List Node fig = new ListNode
<"fig", first, null>;
List Node mango = new ListNode
<"mango", fig, null>;
first.right = fig;
fig.right = mango;
```

Data Compression

Types:
Lossless: Can recover exact data
Lossy: Can recover approximate data
- Use when you don't care, photos can't tell the difference, can compress it more

Bytes

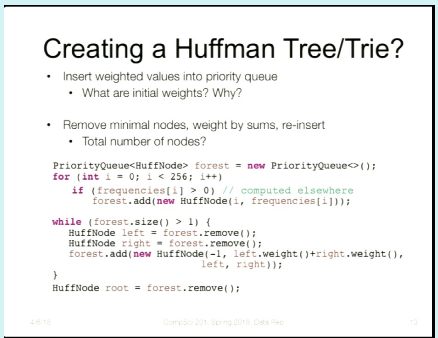


Huffman: Text Compression

In the trie, 0 is left, 1 is right
Make the ones that occur most often the shortest path
Ones that rarely occur can be long
Ones that never occur can be as long as we want
Look at it 8 bits at a time
Building: Combine minimally weighted trees --> Greedy
Bad Huffman Tree: when different character occurs once
Good Hufman Tree: One character occurs multiple times

Alphabet size and run time and compression rate:
- Alphabet size has a big impact on run time because alph size tell syou how big the tree will be
- The number of leaves is equal to the size of your allphabet, so you have 2^k nodes in your tree
- Amount of compression is frequency that it occurs
○ 256 characters that occur the same amount of time is bad compression
○ Huffman takes advantage of the fact that some characters occur more often than others

Creating Huffman Tree



Huffman: TreeTighten APT

```
public class TreeTighten
{
    public TreeNode tighten(TreeNode t)
    {
        if(t == null || (t.left == null && t.right == null))
            return t;
        if(t.left == null)
            t = tighten(t.right);
        else if(t.right == null)
            t = tighten(t.left);
        else
        {
            t.left = tighten(t.left);
            t.right = tighten(t.right);
        }
        return t;
    }
}
```

Big Oh for Huffman Encodings

Encoding

- Given a file with:
  - n characters
  - an alphabet of k distinct characters,
  - r is the compression rate (bits/character)

1. Count occurrence of all occurring characters $O(n)$
2. Build priority queue $O(k \lg k)$
3. Build Huffman tree $O(k \lg k)$
4. Create Table of encodings from tree $O(k)$
5. Write Huffman tree and coded data to file $O(n)$

4/6/18 CompSci 201, Spring 2018, Data Rep 34

Read in tree data: O(k)

Decode bit string with tree: O(n)

Actor Actor Graph

```
public void createActorGraph() {
    for (Actor a : myActors.values())
        for (Actor b : a.coStars().keySet())
            myG.addEdge(a.getName(), b.getName());
}
```

Movie Movie Graph

```
public void createMovieGraph() {
    for (Movie m : myMovies.values())
        for (Actor a : m.getActors())
            for (Movie otherMoov : a.getMovies())
                if (otherMoov != m)
                    myG.addEdge(m.name, otherMoov.name);
}
```

Actor Movie Graph

```
public void createActorMovieGraph() {
    for (Movie m : myMovies.values())
        for (Actor a : m.getActors())
            myG.addEdge(m.name, a.name);
}
```

DFS Recursion

```
Public void recursive(Graph.Vertex start,
                      TreeSet<Graph.Vertex> visited) {

    visited.add(start);

    for(Graph.Vertex adj: myGraph.getAdjacent(start)) {
        if (! visited.contains(adj)) {
            recursive(adj,visited);
        }
    }
}
```

Erdos Number Part 1

```
public class ErdosNumber {
    Map<String, Set<String>> myGraph = new TreeMap<String, Set<String>>();
    Map <String, Integer> myDistance = new TreeMap <String, Integer> ();
    public String[] calculateNumbers(String[] pubs) {
        // TODO complete calculateNumbers
                myGraph = getAdjList(pubs);

                 //for debugging your getAdjList
                //printMap(myGraph);

                // TODO initialize distances for every author


                //  Traverse the graph starting at Erdos
                bfs("ERDOS");

                // TODO construct answer array
                int count = 0;
                String [] ret = new String [myGraph.keySet().size()];
                int i= 0;
                for (String k: myGraph.keySet()) {
                    if(myDistance.containsKey(k)) {
                        ret[i]= k+ " " + myDistance.get(k);
                    }
                    else {
                        ret[i] = k;
                    }
                    i++;

                }
                return ret;
    }
    private void addEdge(Map<String, Set<String>> adjList, String from, String to) {
        if (!adjList.containsKey(from)) {
            adjList.put(from, new TreeSet<String>());
        }
        adjList.get(from).add(to);

        if (!adjList.containsKey(to)) {
            adjList.put(to, new TreeSet<String>());
        }
        adjList.get(to).add(from);

    }
```

Creating Adjacency Matrix

```
public int howLong(String []
connects, String [] costs){
    int [] [] adjMatrix = new int
[connects.length]
[connects.length]'
   for (int i =0;
i<connects.length; i++){
      String [] edges -
connects[i[.split(" ");
      String [] weights =
costs[i].split(" ");
           for (int j =0;
j<edges.length; j++){
                 adjMatrix[i][Inte
ger.partseInt(edges[j])) =
Integer.parseInt(weights[j]);
```