Overview

- Definitions, Terminology, and Properties
- Binary Trees
- Search Trees: Improving Search Speed
- Traversing Binary Search Trees

Trees

Searching in a Linked List (1/3)
- Searching for element in LinkedList involves pointer-chasing and checking consecutive Nodes to find it (or not)
  - It is sequential access
  - O(N) – can stop sooner for element not found if list is sorted
- Finding i-th element in array or ArrayList is random access (which means O(1)), but searching for particular element (even with index) remains sequential O(N)
- Even though NodeLists support indexing (dictated by Java’s list interface), finding the i-th element is also done (under the hood) by pointer-chasing and hence is O(N)

Searching in a Linked List (2/3)
- Searching for E:
  - start at A, beginning of list
  - but A is not E – continue to node B
  - but B is not E, continue to node C (and so on...)
  - till... E is E, found it!
  - or it isn’t in list – exit on null (unsorted) or first element greater (sorted)

Searching in a Linked List (3/3)
- For N elements, search time is O(N)
  - unsorted: sequentially check every node in list till element (“search key”) being searched for is found, or end of list is reached
  - if in list, for a uniform distribution of keys, average search time is N/2
  - if not in list, it is N
  - sorted: average search time is N/2 if found, N/2 if not found (the win!)
  - we ignore issue of duplicates
- No efficient way to access N-th node in list
- Insert and remove similarly have average search time of N/2 to find the right place
- Is there another data structure that provides faster search time and still fast updating of the data structure?

Binary Search (1/5)
- Worst case for searching sorted linked list is checking every element, i.e., sequential access
- We can do better with a sorted array which allows random access at any index
- Let’s demo binary search (“bisection”) for a sorted list of numbers
- Website:
Binary Search (2/5)
- For N elements, search time is \(O(\log_2 N)\) (since we reduce number of elements to search by half each time), very efficient!
  - Start in the middle
  - Keep bisecting the array by halving the index, deciding which half interval the search key lies in, until we land on that key or can't subdivide further (not in array)

Binary Search (3/5)
- \(\log_2 N\) is considerably smaller than N, especially for large N. So doing things in time proportional to \(\log_2 N\) is better than doing them in proportion to N!

Binary Search (4/5)
- A sorted array can be searched quickly using bisection because arrays are indexed
- Java ArrayLists are indexed too, so they share this advantage! But inserting and removing from ArrayLists is slow!
- Inserting into or deleting from middle of ArrayList causes all successor elements to be shifted over to make room. For arrays, you manage this yourself. Both have same worst-case run time – \(O(N)\)
- Advantage of linked lists is insert/remove by manipulating pointer chain is faster \([O(1)]\) than shifting elements \([O(N)]\), but search can't be done with bisection \(\circ\), a real downside if search is done frequently

Binary Search (5/5)
- Is there a data structure/Abstract Data Type that provides both search speed of sorted arrays and ArrayLists and insertion/deletion efficiency of linked lists?
  - Yes, indeed! Trees!

Trees vs Linked Lists (1/2)
- Singly linked list – collection of nodes where each node references only one neighbor, the node's successor:

Trees vs Linked Lists (2/2)
- Tree – also collection of nodes, but each node may reference multiple successors/children
- Trees can be used to model a hierarchical organization of data
Technical definition of a Tree

- Finite set, $T$, of one or more nodes such that:
  - $T$ has one designated root node
  - remaining nodes partitioned into disjoint sets $T_1, T_2, \ldots, T_n$
  - each $T_i$ is also a tree, called subtree of $T$
- Look at the image on the right - where have we seen such hierarchies like this before?

Inheritance Hierarchies as Trees

- Higher in inheritance hierarchy, more generic
  - Animal is most generic
- Lower in inheritance hierarchy, more specific
  - Dog is more specific than Mammal, which in turn is more specific than Animal

Graphical Containment Hierarchies as Trees

- Levels of containment of GUI components
- Higher levels contain more components
- Lower levels contained by all above them
  - Panes contained by Root Pane of PaneOrganizer, which is contained by Scene

Tree Structure

- Note that the tree structure has meaning
  - Any subtree of $T$, $T_i$, is also a tree with specific values
  - Can be useful to only examine specific subtrees of $T$

Tree Terminology

- A is the root node
- B is the parent of D and E
- D and E are children of B
- (C ---- F) is an edge
- D, E, F, G, and I are external nodes or leaves (i.e., nodes with no children)
- A, B, C, and H are internal nodes
- depth (level) of E is 2 (number of edges to root)
- height of the tree is 3 (max number of edges in path from root)
- degree of node B is 2 (number of children)

Binary Trees

- Each internal node has a maximum of 2 successors, called children
  - So, each internal node has degree 2 at most
- Recursive definition of binary tree: A binary tree is either an:
  - External node (leaf), or
  - Internal node (root) with two binary trees as children (left subtree and right subtree)
  - Empty tree (represented by a null pointer)

Note: These nodes are similar to the linked list nodes, with one data and two child pointers
Properties of Binary Trees (1/2)
- A Binary tree is full when each node has exactly zero or two children
- Binary tree is perfect when for every level \( i \), there are \( 2^i \) nodes (i.e., each level contains a complete set of nodes)
  - Thus, adding anything to the tree would increase its height

Properties of Binary Trees (2/2)
- In a full Binary Tree: \( \# \text{leaf nodes} = (\# \text{internal nodes}) + 1 \)
- In a perfect Binary Tree: \( \# \text{nodes at level } i \leq 2^i \)
- In a perfect Binary Tree: \( \# \text{leaf nodes} \leq 2^{\text{height}} \)
- In a perfect Binary Tree \( \text{height} \geq \log_2(\# \text{nodes}) - 1 \)

Clicker Question
Which of the following are true about internal nodes and external nodes for binary trees?
A. Internal nodes are nodes located between the root of the tree and leaves
B. External nodes cannot be leaves of a tree
C. The root node is often an external node
D. Internal nodes must have one or two children, while external nodes have none

Binary Search Tree a.k.a BST (1/2)
- Binary search tree stores keys in its nodes such that, for every node, keys in left subtree are smaller, and keys in right subtree are larger

Binary Search Tree (2/2)
- Below is also binary search tree but much less balanced. Gee, it looks like a linked list!
- The shape of the trees is determined by the order in which elements are inserted

Binary Search Tree Class (1/3)
- What do binary search trees know how to do?
  - much the same as sorted linked lists: insert, remove, size
  - BSTs also have a special search method, since searching is more complicated than simply iterating through its nodes
- Let’s see what an implementation of a binary search tree class would look like...
  - you’ll learn more about implementing classes of data structures in CS16!
Nodes, data, and keys

- _data_ is a composite that can contain many properties, one of which is a key that _Nodes_ are sorted by (here, ISBN number).

- _left_ and _right_ children that hold data implementing `Comparable<T>`
  - Four instance variables: _data, parent, left, and right_, with each having a get and set method.
  - _data_ represents the data that _Node_ stores. It also contains the key attribute that _Nodes_ are sorted by – we’ll use a tree that stores books.
  - _parent_ represents the direct parent (another _Node_) of _Node_ – only used in remove method.
  - _left_ represents _Node_’s left child and contains a subtree, all of whose data is less than _Node_’s data.
  - _right_ represents _Node_’s right child and contains a subtree, all of whose data is greater than _Node_’s data.
  - Arbitrarily select which child should contain data _equal_ to _Node_’s data _number_.

**Binary Search Tree: Node Class (2/3)**

```java
public class Node<T extends Comparable<T>> {
    Comparable<T> _data;
    private Node<T> _left;
    private Node<T> _right;
    private Node<T> _parent;

    //constructor
    public Node(T data, Node<T> parent) {
        _data = data;
        _parent = parent;
    }
    //constructor
    public Node(T data, Node<T> parent) {
        _data = data;
        _parent = parent;
    }
    //constructor
    public Node(T data, Node<T> left, Node<T> right) {
        _data = data;
        _left = left;
        _right = right;
    }
    //constructor
    public Node(T data, Node<T> parent, Node<T> left, Node<T> right) {
        _data = data;
        _parent = parent;
        _left = left;
        _right = right;
    }
    //constructor
    public Node(T data, T key) {
        _data = data;
        _key = key;
    }
    //constructor
    public Node(T data, Node<T> parent, T key) {
        _data = data;
        _parent = parent;
        _key = key;
    }
    //constructor
    public Node(T data) {
        _data = data;
    }
    //constructor
    public Node() {
    }
    public int compareTo(Node<T> node) {
        return _data.compareTo(node._data);
    }
    //getters
    public T getData() {
        return _data;
    }
    public Node<T> getLeft() {
        return _left;
    }
    public Node<T> getRight() {
        return _right;
    }
    public Node<T> getParent() {
        return _parent;
    }
    //setters
    public void setData(T data) {
        _data = data;
    }
    public void setLeft(Node<T> left) {
        _left = left;
    }
    public void setRight(Node<T> right) {
        _right = right;
    }
    public void setParent(Node<T> parent) {
        _parent = parent;
    }
    //... other methods...
}
```

**Binary Search Tree: Node Class (3/3)**

- “Smart” _Node_ includes the following methods:
  - _swapData_;
  - _remove_;
  - _removeWithParent_;
  - _nodeRemove_;
  - _nodeInsert_;
  - _nodeSearch_;
  - _nodeCount_;
  - _nodeMax_;
  - _nodeMin_;
  - _nodeHeight_;
  - _nodeBalance_;
  - _nodeNLR_;
  - _nodeRNL_;
  - _nodeNLR_;
  - _nodeRNL_;
  - _nodeNLR_;
  - _nodeRNL_;
  - _nodeNLR_;
  - _nodeRNL_;

**Binary Search Tree: Node Class (1/3)**

- _Nodes_ have a maximum of two non-null children that hold data implementing `Comparable<T>`.

**Binary Search Tree Class (2/3)**

```java
public class BinarySearchTree<T extends Comparable<T>> {
    public Node<T> _root;
    public void insert(T newData) {
        Node<T> temp = _root;
        Node<T> prev = null;
        while (temp != null) {
            if (newData.compareTo(temp._data) < 0) {
                prev = temp;
                temp = temp._left;
            } else if (newData.compareTo(temp._data) > 0) {
                prev = temp;
                temp = temp._right;
            } else {
                throw new RuntimeException("Duplicate key!");
            }
        }
        prev._left = new Node<T>(newData);
    }
    //... other methods...
}
```

**Binary Search Tree Class (3/3)**

```java
public class BinarySearchTree<T extends Comparable<T>> {
    public Node<T> _root;
    public void insert(T newData) {
        Node<T> temp = _root;
        Node<T> prev = null;
        while (temp != null) {
            if (newData.compareTo(temp._data) < 0) {
                prev = temp;
                temp = temp._left;
            } else if (newData.compareTo(temp._data) > 0) {
                prev = temp;
                temp = temp._right;
            } else {
                throw new RuntimeException("Duplicate key!");
            }
        }
        prev._left = new Node<T>(newData);
    }
    //... other methods...
}
```
Recursion – Turtles all the Way Down

- "A well-known scientist (some say it was Bertrand Russell) once gave a public lecture on astronomy. He described how the earth orbits around the sun and how the sun, in turn, orbits around the center of vast collection of stars called our galaxy. At the end of the lecture, a little old lady at the back of the room got up and said: "What you have told us is rubbish. The world is flat! It is supported on the back of a giant turtle."

  "The scientist gave a superior smile before replying. "What is the turtle standing on?" "You're very clever, young man," said the old lady, "but it's turtles all the way down!"

  - Stephen Hawking, A Brief History of Time (1988)
Searching a Binary Tree Recursively

```java
public Node<Type> search(Type dataToFind) {
    // if _data is the thing we're searching for
    if (_data.compareTo(dataToFind) == 0) {
        return this;
    }
    // if _data > dataToFind, can only be in left tree
    else if (_data.compareTo(dataToFind) > 0) {
        if (_left != null) {
            return _left.search(dataToFind);
        }
    // if _data < dataToFind, can only be in right tree
    } else {
        if (_right != null) {
            return _right.search(dataToFind);
        }
    }
    // only gets here if dataToFind isn't in tree, otherwise would've returned sooner
    return null;
}
```

Binary Search: Nearest Value

- What if we wanted to find the closest possible value, rather than the exact value?
  - search(H) would return H
  - search(D) would return B
- Can use binary search with one modification
  - Instead of immediately returning the element if you find it, have a variable that keeps track of the closest neighbor

Nearest Value Pseudocode

```java
search (type dataToFind, Node currNode):
// method in BinaryTree class
if currNode is null:
    // base case
    return _closestNode
// initialized elsewhere in BinaryTree class
// see how close current data is from target
data
\[\text{closestNode} = \text{currNode} \]
// if this is closer than our current assumption of closest node, update
if abs(currNode.data - dataToFind) < _closestData:
    _closestData = abs(currNode.data - dataToFind)
    _closestNode = currNode
// if currNode.data equals target data, return immediately (can't get closer)
if currNode.data == dataToFind:
    return currNode
// if currNode.data is equal to desired data
else if currNode.data < dataToFind:
    // otherwise, recurse
    return this.search(dataToFind, currNode.left)
else:
    return this.search(dataToFind, currNode.right)
```

Binary Search: Nearest Values

- What if we wanted to find the closest x possible values, rather than just the closest value?
  - search(H, 1) would return {H}
  - search(D, 2) would return {A, B}
- Can use method similar to binary search
  - Find closest neighbors to point, then find closest neighbors to immediate neighbors
  - Won't go into detail, but this is a powerful advantage of the BST

Insertion into a BST(1/2)

- Search BST starting at root until we find where the data to insert belongs
  - Insert data when we reach a Node whose appropriate child is null
- We make a new Node, set the new Node's _data to the data to insert, and set the parent's child reference to this Node.
- Runtime is O(log N), yay!
  - O(log N) to search the tree to find the place to insert
  - Constant time operations to make new Node

Insertion into a BST(2/2)

Example: Insert 115

Before:

After:
Insertion Code in BST

- Again, we use a "Smart Node" approach and delegate

```java
public void insert(Typed newData) {
    // If tree is empty, make first node. No traversal necessary!
    if (_root == null){
        _root = new Node(newData, null);
    } else{
        _root.insert(newData); // delegate
    }
}
```

Insertion Code in Node

```java
public Node<Typed> insert(Typed newData) {
    if (_data.compareTo(newData) > 0) {
        // Code for inserting left side
        if (_left == null){
            _left = new Node(newData, this);
            return _left;
        } else{
            return _left.insert(newData);
        }
    } else {
        // Code for inserting right side
        if (_right == null){
            _right = new Node(newData, this);
            return _right;
        } else{
            return _right.insert(newData);
        }
    }
}
```

Insertion Simulation (1/4)

- Insert: 224
- First call `insert` in BST:

```
_root = _root.insert(newData);
```

Insertion Simulation (2/4)

- 123 says: "I am less than 224. I'll let my right child deal with it.

```
if (_data.compareTo(newData) > 0) {
    if (_left == null){
        _left = new Node(newData, this);
        return _left;
    } else{
        return _left.insert(newData);
    }
}
```

Insertion Simulation (3/4)

- 252 says: "I am greater than 224. I'll pass it on to my left child — but my left child is null!"

```
if (_data.compareTo(newData) > 0) {
    if (_left == null){
        _left = new Node(newData, this);
        return _left;
    } else{
        // Code for continuing traversal elided
    }
}
```

Insertion Simulation (4/4)

- 252 says: "You belong as my left child, 224. Let me make a node for you, make this new node your home, and set that node as my left child."

```
_left = new Node(newData, this);
return _left;
```
Notes on Trees (1/2)

- Different insertion order of nodes results in different trees
  - If you insert a node referencing data value of 18 into empty tree, that node will become root.
  - If you then insert a node referencing data value of 12, it will become left child of root.
  - However, if you insert node referencing 12 into an empty tree, it will become root.
  - Then, if you insert one referencing 18, that node will become right child of root.
  - Even with same nodes, different insertion order makes different trees.
  - On average, for reasonably random (unsorted) arrival order, trees will look similar in depth, so order doesn’t really matter.

Notes on Trees (2/2)

- When searching for a value, reaching another value that is greater than the one being searched for does not mean that the value being searched for is not present in tree (whereas it does in linked lists!)
  - It may well still be contained in left subtree of node of greater value that has just been encountered.
  - Thus, where you might have given up in linked lists, you can’t give up here until you reach a leaf (but depth is roughly log_N which is much smaller than N/2!)

Clicker Question

Which tree does the code at the top right represent below?

A.  

B.  

C.  

Remove: no child case

- Node to remove has no children (is a leaf).
  - Just set the parent’s reference to this Node to null — no more references means the Node is garbage collected!
  - Example: Remove P
  - Set O’s right child to null, and P is gone!

Remove: one-child case

- Harder case: Node to delete has one child.
  - Replace Node child.
  - Example: Remove O
    - O has one child.
    - O replaces O by replacing its left child, previously O, with P.

Remove: two-children case (1/3)

- Hard case: node to remove has two internal children.
  - Brute force: Just flag node for removal, and rewrite tree at a later time — bad idea, because now every operation requires checking that flag.
  - Instead, do the work right away.
  - This is tricky, because not immediately obvious which child should replace its parent.
  - Non-obvious solution: First swap the data in Node to be removed with data in a Node that doesn’t have two children, then remove Node using one of simpler remove cases.
Remove: two-children case (2/3)

- Use an auxiliary method, swapData
  - swaps data in node to be removed with the data in the right-most node in its left subtree
  - this child has a key value less than all Nodes in the to-be-removed Node's right subtree, and greater than all other nodes in its left subtree
  - since it is a right-most Node, it has at most one child
  - this swap is temporary — we then remove the node in the right-most position using simpler remove

Remove: BST Code

- Starts as usual with delegating to root
- Need to first find the Node to remove, then we remove it
- Nodes are “smart,” so they can remove themselves
- O(\log N) because of searching

// in BinarySearchTree:
public void remove(Node dataToRemove) {
    Node root = _root.search(dataToRemove);
    root.remove();
}

Remove: two-children case (3/3)

- Remove R
  - R has two children
  - swap R with the right-most Node in the left subtree, Q
    - Children in R's left subtree are smaller than Q
    - Children in R's right subtree are larger than Q
    - R is in the wrong place but...
  - remove R (in its new position) using the one-child case

Remove: Node Code (1/3)

- In the Node class, remove method allows Node to remove itself

public Node remove() {
    // case 1 - Node to remove is a leaf node
    if (parent.getLeft() == null) {
        // set its parent's reference that originally refers to this node to null
        if (parent.getRight() == null) {
            parent.setLeft(null);
        } else {
            parent.setRight(null);
        }
    } // case for other cases on next slide...
}

Remove: Node Code (2/3)

public Node remove() {
    // case for case 1 (one child)
    if (left != null && right == null) {
        // case 2.1 - Node only has left child
        _parent.setLeft(_left);
        if (_parent.getLeft() != null) {
            _parent.setLeft(_left);
        } else {
            _parent.setRight(_right);
        }
    } else if (left == null && right != null) {
        // case 2.2 - Node only has right child
        _parent.setLeft(_right);
        if (_parent.getRight() != null) {
            _parent.setRight(_right);
        } else {
            _parent.setLeft(_left);
        }
    } // case 3 on next slide ...
}

Remove: Node Code (3/3)

- Successor is guaranteed to have at most one child, so we remove with simpler remove case

public Node remove() {
    // case for case 1 (no children) aliased
    // case for case 2 (one child) aliased
    // case 3 - both children
    Node successor = this.swapData(); // swap data with successor
    Node toSwapRemove = _right.remove(); // remove successor, which holds original Node's data
    _parent.setRight(toSwapRemove); // return toSwap, since toSwap was data we removed
    return successor; // return this if we didn't do any swapping since Node is removed
}

// swapData defined on next slide...
**Remove: swapData code**

- We find the right-most Node in left subtree, but we can also find the left-most Node in right subtree.
- Use recursion!

```java
public Node<T> swapData() {
    Node<T> curr = _left; // first get left child
    while (curr.getRight() != null) // go right as far as possible
        curr = curr.getRight();

    // swap data of this Node and successor
    type template _data;
    _data = curr.getData();
    curr.setData(tempData);
    return curr;
}
```

**Tree Runtime**

- Binary Search Tree has a search of $O(\log n)$ → can we make it faster?
  - Or a ternary tree with $O(\log n)$?
- Let’s try the runtime for a search with 1,000,000 nodes
  - $\log(1,000,000) = 20$, a shallower but broader tree
- Analysis: the logs are not sufficiently different and the comparison (basically an n-way nested if-else) is far more time consuming, hence not worth it
- Furthermore, binary tree makes it easy to produce an ordered list (see slide 64)

**Hash Tables vs. Trees**

- You might be asking “why use trees when hash tables have $O(1)$ insert and remove?”
  - Hash Tables and Trees are different data structures used for different kinds of problems
  - If you’re only concerned with finding exact values, a hash table will be faster
    - You know the exact key to search for
    - Ex. Find every person in the class that has a birthday on 05/08
    - Use Hash Table where key is birthday, and value is CS15Student
  - If you’re trying to solve a nearest neighbors problem, a tree will be faster
    - You do not know the exact key to search for
    - Ex. Find 4 people closest to my height
    - Use a BST where key is height, value is CS15Student
  - Can produce an already sorted list of data items by traversing the tree

**Traversing a Binary Tree**

- We often want to access every Node in tree
  - so far, we have only searched for a single element
  - so we can use a traversal algorithm to perform some arbitrary operation on every Node in tree
- Many ways to traverse Nodes in tree
  - order children are visited is important
  - three traversal types: Inorder, preorder, postorder
- Exploit recursion!
  - subtree has same structure as tree

**Inorder Traversal of BST**

- Considered “in order” because Nodes are visited in sorted order
- Traverse left subtree first, then visit self, then traverse right subtree
- Use recursion!

```java
public void inOrder() {
    // check for all children visited
    _left.inOrder();
    this.displayNode();
    _right.inOrder();
}
```

**N-ary tree example**

- Use the first character of last name as the start of a binary tree for all names with that initial character
- Shove down right away
- Disadvantages
  - Some trees will be very small, eg. “q” some will be much larger than average, eg. “t, v”
  - Dividing by 26 doesn’t really get you first much (high and logarithms aren’t that different)

**Inorder Traversal of BST**

- Considered “in order” because Nodes are visited in sorted order
- Traverse left subtree first, then visit self, then traverse right subtree
- Use recursion!
Use rule matrix to implement strategy

2 stack algorithm for single-pass Infix to Postfix

(a + (b * (c ^ d))) → a b c d ^ * +

Use rule matrix to implement strategy

A) Push operands onto operand stack; push operators in precedence order onto the operator stack.
B) When precedence order would be disturbed, pop operator stack until order is restored, evaluating each pair of operands popped from the operand stack and pushing the result back onto the operand stack.
C) "(" starts a new substack;
   ")" pops until its matching "("

Dijkstra's infix-to-postfix Algorithm (2/2)

(a + (b * (c ^ d))) → a b c d ^ * +

Operator Stack

Precedence Checker

Top of Stack

Incoming Operator

(  +  *)

Top of Stack

Incoming Operator

(  +  *)

Note: our Stack implementation uses a list, allowing the top element without popping it. Java implementation has a pop method.

Using Prefix, Infix, Postfix Notation

When you type an equation into a spreadsheet, you use Infix; when you type an equation into many Hewlett-Packard calculators, you use Postfix, also known as "Reverse Polish Notation," or "RPN," after its inventor Polish Logician Jan Lukasiewicz (1924)

Easier to evaluate Postfix because it has no parenthesis and evaluates in a single left-to-right pass

Use Dijkstra's 3-stack shunting yard algorithm to convert from user-entered Infix to easy-to-handle Postfix – compile or interpret it on the fly
**Challenge Questions Solutions**

- **Q:** How would you print the elements of a Binary Search Tree in increasing order?
  - **A:** You would traverse the BST in-order.

- **Q:** How would you find the 'successor' (i.e., next greatest number) of a node in a Binary Search Tree?
  - **A:** The pseudo-code for the solution is as follows:
    ```java
    if node != null
        node = node.right()
    while(node != null)
        node = node.left()
    return node
    ```

**Announcements**

- **Lab 7** is due today at lab or Sunday 11/13 at TA Hours
- **Lab 8** is due today at lab or Tuesday 11/15 at TA Hours
  - You know the exact key to search for
  - Ex. Find every person in the class that has a birthday on 05/08
  - Use a Hash Table where key is birthday, and value is CS15Student
- **Data Structures and Algorithms discussion will happen next week**
- **Tetris deadlines**
  - Early: 11/18, 10:00pm
  - Online: 11/20, 11:59pm
  - Late: 11/22, 11:59pm
  - If you missed your design discussion on Wednesday, there will be a make up design discussion **tomorrow at 3pm in CIT 368**

- **Next week’s lectures are very very important**
  - Tuesday: Final Project demos
  - Thursday: Final Project help sessions
- **There will be skits and super cool demos- come to class!**