Lecture 19
More Data Structures for Collections: Part 2
Social Good in CS Survey

www.tinyurl.com/cs15nov11survey
Stack Trace

- When an exception is thrown in a program, get a long list of methods and line numbers known as a stack trace

```
Exception in thread "main" java.lang.NullPointerException
  at DoodleJump.scroll(DoodleJump.java:94)
  at DoodleJump.updateGame(DoodleJump.java:44)
  ...
```

- A stack trace prints out all methods currently on execution stack

- If exception is thrown during execution of recursive method, prints all calls to recursive method
Bootstrapping Data Structures

- This implementation of the stack data structure uses a wrapper of a contained MyLinkedList, but user has no knowledge of that

- Could also implement it with an Array or ArrayList
  - Array implementation could be less efficient as we would have to expand our Array as we push more objects onto the Stack.
  - User’s code would not be affected if the implementation of Stack changed (as is true for methods as well, if their semantics isn’t changed) – loose coupling!

- We’ll use the same technique to implement a Queue
What are Queues?

- Similar to stacks, but elements are removed in different order
  - information retrieved in the same order it was stored
  - **FIFO**: First In, First Out (as opposed to stacks, which are **LIFO**: Last In, First Out)

- Examples:
  - standing in line at the checkout counter or movie theater
  - waitlist for TA hours after randomization

![Server at Seattle restaurant reminding herself what order customers get served in](image)
Methods of a Queue

- Add element to end of queue
- Remove element from beginning of queue
- Returns whether queue has any elements
- Returns number of elements in queue

void enqueue(Type el)
Type dequeue()
boolean isEmpty()
int size()
Enqueuing and Dequeuing

- Enqueuing: adds a node
- Dequeuing: removes a node

Before Enqueuing:

1 2 3

head of queue  tail of queue  student to add

After Enqueuing:

1 2 3 4

head of queue  tail of queue
Enqueuing and Dequeuing

- Enqueuing: adds a node
- Dequeuing: removes a node

Before Dequeuing:

1 2 3 4

head of queue

tail of queue

After Dequeuing:

1 2 3 4

dequeued student

head of queue
tail of queue
Our Queue

- Again use a wrapper for a contained `MyLinkedList`. As with Stack, we’ll hide most of MLL’s functionality and provide special methods that delegate the actual work to the MLL.

- Contain a `MyLinkedList` within `Queue` class
  - `enqueue` will add to the end of `MyLinkedList`
  - `dequeue` will remove the first element in `MyLinkedList`

```java
public class Queue<Type> {
    private MyLinkedList<Type> list;

    public Queue() {
        this.list = new MyLinkedList<>();
    }
    // Other methods elided
}
```
enqueue

- Just call list’s `addLast` method – delegation

- This will add node to end of `list`

```java
public void enqueue(Type newNode) {
    this.list.addLast(newNode);
}
```
**dequeue**

- We want first node in `list`
- Use `list`'s `removeFirst` method – delegation
  
  ```java
  public Type dequeue() {
      return this.list.removeFirst();
  }
  ```

- What if `list` is empty? There will be nothing to dequeue!
- Our `MyLinkedList` class's `removeFirst()` method returns `null` in this case, so `dequeue` does as well
isEmpty() and size()

- As with Stacks, very simple methods; just delegate to MyLinkedList

```java
public int size() {
    return this.list.size();
}

public boolean isEmpty() {
    return this.list.isEmpty();
}
```
TopHat Question

In order from head to tail, a queue contains the following: bostonRob, russell, parvati, ozzy. We remove each person from the queue by calling dequeue() and then immediately push() each dequeued person onto a stack.

At the end of the process, what is the order of the stack from top to bottom?

A. bostonRob, russell, parvati, ozzy
B. bostonRob, ozzy, russell, parvati
C. ozzy, parvati, russell, bostonRob
D. It's random every time.
Trees
Searching in a Linked List (1/2)

- 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

- Getting $N^{th}$ element in an `Array` or `ArrayList` by index is random access (which means $O(1)$), but (content-based) searching for particular element, even with index, remains sequential $O(N)$

- Even though `LinkedLists` support indexing (dictated by Java’s `List` interface), getting the $i$th element is also done (under the hood) by pointer chasing and hence is $O(N)$
Searching in a Linked List (2/2)

- For N elements, search time is $O(N)$
  - **unsorted**: sequentially check every node in list until 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 for a random element 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 (via index)

- Insert and remove similarly have average search time of $N/2$ to find the right place

*Actually more complicated than this – depends on distribution of keys*
## Searching, Inserting, Removing

<table>
<thead>
<tr>
<th></th>
<th>Search if unsorted</th>
<th>Search if sorted</th>
<th>Insert/remove after search</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked list</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Array</td>
<td>O(N)</td>
<td>O(log N) [coming next]</td>
<td>O(N)</td>
</tr>
</tbody>
</table>
Binary Search (1/4)

- Searching **sorted linked list** is sequential access
- We can do better with a **sorted array** that allows random access at any index to obviate sequential search
- Remember merge sort with search $O(\log_2 N)$ where we did “bisection” on the array at each pass
- If we had a sorted array, we could do the same thing (like “20 questions”)
  - start in the middle
  - keep bisecting array, deciding which portion of the sub-array the search key lies in, until we find that key or can’t subdivide further (not in array)
  - For $N$ elements, search time is $O(\log_2 N)$ (since we reduce number of elements to search by half each time), very efficient!
Binary Search (2/4)

- $\log_2 N$ grows much more slowly than $N$, especially for large $N$

*relatively small $n$ in this graph, but imagine how large the difference is as $n$ increases*

<table>
<thead>
<tr>
<th>$N$</th>
<th>(int) $\log(N)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
</tr>
<tr>
<td>10,000</td>
<td>13</td>
</tr>
<tr>
<td>1,000,000</td>
<td>17</td>
</tr>
<tr>
<td>10,000,000</td>
<td>20</td>
</tr>
<tr>
<td>100,000,000</td>
<td>23</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>27</td>
</tr>
</tbody>
</table>
Binary Search (3/4)

- A sorted array can be searched quickly using bisection because arrays are indexed.

- ArrayLists (implemented in Java using arrays) are indexed too, so a sorted ArrayList shares this advantage! But inserting and removing from ArrayLists is slow (except for insertion and removal at end):
  - Inserting into or deleting from an arbitrary index in ArrayList causes all successor elements shift over. Thus insertion and deletion have same worst-case run time $O(N)$.

- Advantage of linkedLists is insert/remove by manipulating pointer chain is faster $[O(1)]$ than shifting elements $[O(N)]$, but search can’t be done with bisection 😞, a real downside if search is done frequently.
Binary Search (4/4)

- Is there a data structure that provides both search speed of sorted arrays and ArrayLists and insertion/deletion efficiency of linked lists?
- Yes, indeed! Trees! They provide much faster searching than linked lists and much faster insertions than arrays!
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: $T_1$, $T_2$, … $T_n$
  - each $T_i$ is also a self-contained tree, called subtree of $T$

- Look at the image on the right—where have we seen such hierarchies like this before?
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, 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 \textit{children}
  - i.e., each internal node has degree 2 at most
- Recursive definition of binary tree: A binary tree is either an:
  - external node (\textit{leaf}), or
  - internal node (\textit{root}) with one or two binary trees as children (\textit{left subtree, right subtree})
  - empty tree (represented by a null pointer)
  - \textit{Note}: These nodes are similar to the linked list nodes, with one data and two child pointers – we show the data element inside the circle
Properties of Binary Trees (1/2)

- A binary tree is **full** when each node has exactly zero or two children.
- A 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: (number of leaf nodes) = (number of internal nodes) + 1
- In a perfect Binary Tree: (number of nodes at level $i$) = $2^i$
- In a perfect Binary Tree: (number of leaf nodes) $\leq 2^{(\text{height})}$
- In a perfect Binary Tree: (height) $\geq \log_2(\text{number of nodes}) - 1$
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.

Note: the keys here are sorted alphabetically!
Below is also BST 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.
BST Class (1/4)

● What do BSTs know how to do?
  o much the same as sorted linked lists: insert, remove, size, empty
  o BSTs also have their own search method – a bit more complicated than simply iterating through its nodes

● What would an implementation of a BST class look like…
  o in addition to data, left, and right child pointers, we’ll add a parent “back” pointer for ease of implementation (for the remove method – analogous to the previous pointer in doubly-linked lists!)
  o you’ll learn more about implementing data structures in CS200!
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 #) – but how do we compare **Nodes** to sort them?

```
BinarySearchTree
  root
    data
      parent
        right
          left
            Node<Book>
              data
                parent
                  right
                    left
                      ... Node<Book>
                      ...
                      ...
                      ...
                      ...
                      Book
                        isbn = 9783245206
                        pubDate = ...
                        title = ...
                        author = ...
```

Andries van Dam © 2021 11/11/21
Java’s \texttt{Comparable\langle Type\rangle} interface (1/3)

- Previously we used \texttt{==} to check if two things are equal
  - this only works correctly for primitive data types (e.g., \texttt{int}), or when we are comparing two variables referencing the exact same object
  - to compare \texttt{Strings}, need a different way to compare things

- We can implement the \texttt{Comparable\langle Type\rangle} generic interface provided by Java

- It specifies the \texttt{compareTo} method, which returns an \texttt{int}

- Why don’t we just use \texttt{==}, even when using something like ISBN, which is an \texttt{int}?
  - can treat ISBNs as \texttt{ints} and compare them directly, but more generally we implement the \texttt{Comparable\langle Type\rangle} interface, which could easily accommodate comparing \texttt{Strings}, such as author or title, or any other property
Java’s **Comparable<Type>** interface (2/3)

- The **Comparable<Type>** interface is specialized (think of it as parameterized) using generics

  ```java
  public interface Comparable<Type> {
    int compareTo(Type toCompare);
  }
  ```

- Call `compareTo` on a variable of same type as specified in implementator of interface (**Book**, in our case)
  - `currentBook.compareTo(bookToFind);`
Java’s `Comparable<Type>` interface (3/3)

- `compareTo` method must return an `int`
  - negative if element on which `compareTo` is called is less than element passed in as the parameter of the search
  - 0 if element is equal to element passed in
  - positive if element is greater than element passed in
  - sign of `int` returned is all-important, magnitude is not and is implementation dependent

- `compareTo` not only used for numerical comparisons—it could be used for alphabetical or geometric comparisons as well—depends on how you implement `compareTo`
“Comparable” Book Class

- Recall format for `compareTo`:
  - `elementA.compareTo(elementB)`

- Book class now implements `Comparable<Book>`
  - this means we can compare books, using `bookA.compareTo(bookB)`

- `compareTo` is defined according to these specifications
  - returns number that is `<0, 0` or `>0`, depending on the ISBN numbers
  - `<0` if stored `this.isbn < toCompare`
**BST Class (2/4)**

- **Using keyword `extends` in this way ensures that `Type` implements `Comparable<Type>`**
  - note nested `<>
  - nested `<>` to show it modifies `Type` and not the class

- **All elements stored in `MyLinkedList` must now have `compareTo` method for `Type`; thus restricts generic

```java
public class BinarySearchTree<Type extends Comparable<Type>> {
    private Node<Type> root;

    public BinarySearchTree(Type data) {
        //Root of the tree
        this.root = new Node(data, null);
    }

    // other methods shown next slide
}
```

In our example use `Book` as `Type`
public class BinarySearchTree<Type extends Comparable<Type>> {

    private Node<Type> root;

    public BinarySearchTree(Type data) {
        //Root of the tree
        this.root = new Node(data, null);
    }

    public void insert(Type newData) {
        // . . .
    }

    public Node<Type> search(Type dataToFind) {
        // . . .
    }

    public int size() {
        // . . .
    }

} // end of class

//class continued
public void remove(Type dataToRemove) {
    // . . .
}

public Node<Type> search(Type dataToFind) {
    // . . .
}

public int size() {
    // . . .
}

} // end of class
Our implementations of **Linked Lists**, **Stacks**, and **Queues** are “smart” data structures that chain “dumb” nodes together:

- the lists did all the work by maintaining previous and current pointers and did the operations to search for, insert, and remove information – thus, nodes were essentially data containers

Now we will use a “dumb” tree with “smart” nodes that will delegate using **recursion**:

- tree will delegate action (such as searching, inserting, etc.) to its root, which will then delegate to its appropriate child, and so on
- creates specialized **Node** class that stores its data, parent, and children, and can do operations such as **insert** and **remove**
“Smart” Node includes the following methods:

```java
// pass in entire data item, containing key, so compareTo() will work
public Node<Type> search(Type dataToFind);
public Node<Type> insert(Type newData);

/* remove deletes Node pointing to dataToRemove, which contains key;
removing Node also will remove the matched data element instance unless
there’s another reference to it */
public Node<Type> remove(Type dataToRemove);
```

- Plus setters and getters of instance variables, defined in the next slides ...
BST: Node Class (2/3)

- **Nodes** have a maximum of two non-null children that hold data implementing `Comparable<Type>`
  - 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 make 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.
public class Node<Type implements Comparable<Type>> {
    private Type data;
    private Type parent;
    private Node<Type> left;
    private Node<Type> right;

    public Node(Type data, Node<Type> parent) {
        this.data = data;
        this.parent = parent;
        //child ptrs null for leaf nodes; set for internal nodes when child is created
        this.left = null;
        this.right = null;
    }

    // will define other methods in next slides...
}
Smart Node Approach

- **BinarySearchTree** is “dumb,” so it delegates to root, which in turn will delegate recursively to its left or right child, as appropriate

  ```java
  // search method for entire BinarySearchTree:
  public Node<Type> search(dataToFind) {
      return this.root.search(dataToFind);
  }
  ```

- Smart node approach makes our code clean, simple and elegant
  - non-recursive method is much messier, involving explicit bookkeeping of which node in the tree we are currently processing
    - we used the non-recursive method for sorted linked lists, but trees are more complicated, and recursion is easier – a tree is composed of subtrees!
Searching Simulation (animated)

- What if we want to know if 224 is in Tree?
- Tree says:

"Hey Root! Ya got 224?"

123 says:

"Let's see. I'm not 224. But if 224 is in tree, since it's larger, it would be to my right. I'll ask my right child and return its answer."
Searching Simulation (animated)

- What if we want to know if 224 is in Tree?

252 says:

“I’m not 224. I better ask my left child and return its answer.”
Searching Simulation (animated)

- What if we want to know if 224 is in Tree?

224 says:

“224? That’s me! Hey, caller (252) here’s your answer.”

Answer: 224 is in the Tree!
Searching Simulation (animated)

- What if we want to know if 224 is in Tree?

Answer: 224 is in the Tree!

252 says:
“Hey, caller (123)! Here’s your answer.”

Answer: 224 is in the Tree!
Searching Simulation (animated)

- What if we want to know if 224 is in Tree?

Answer: 224 is in the Tree!

123 says: “Hey, Tree! Here’s your answer”
Searching Simulation - Recap

- What if we want to know if 224 is in Tree?
- Tree says “Hey Root! Ya got 224?”
- 123 says: “Let’s see. I’m not 224. But if 224 is in tree, it would be to my right. I’ll ask my right child and return its answer.”
- 252 says: “I’m not 224, it’s smaller than me. I better ask my left child and return its answer.”
- 224 says: “224? That’s me! Hey, caller (252) here’s your answer.” (returning node indicates that query is in tree)
- 252 says: “Hey, caller (123)! Here’s your answer.”
- 123 says: “Hey, Tree! Here’s your answer.”
Searching a BST Recursively Is $O(\log_2 N)$

- Search path: start with root M and choose path to I (for a reasonably balanced tree, M will be more or less “in the middle,” and left and right subtrees will be roughly the same size)
  - structurally, the height of a reasonably balanced tree with $n$ nodes is about $\log_2 n$
  - at most, we visit each level of the tree once
  - so, runtime performance of searching is $O(\log_2 N)$ as long as tree is reasonably balanced, which will be true if entry order is reasonably random (slide 87)
TopHAT Question

What's the runtime of (recursive) search in a BST and why?

• A. $O(n)$ – because you only iterate once
• B. $O(2n)$ – because you go visit both the left and right subtrees
• C. $O(n/2)$ – because you incorporate the idea of “bisection” to mean half the nodes
• D. $O(\log_2 n)$ - because you incorporate the idea of “bisection” to eliminate half the number of nodes to search at each recursion
• E. $O(n^2)$ – because recursion makes your runtime quadratic
Searching a BST Recursively

```java
public Node<Type> search(Type dataToFind) {
    // if data is the thing we’re searching for
    if (this.data.compareTo(dataToFind) == 0) {
        return this;
    // if data > dataToFind, can only be in left tree
    } else if (data.compareTo(dataToFind) > 0) {
        if (this.left != null) {
            return this.left.search(dataToFind);
        }
    // if data < dataToFind, can only be in right tree
    } else if (this.right != null) {
        return this.right.search(dataToFind);
    }
    // Only get here if dataToFind isn’t in tree, otherwise would’ve returned sooner
    return null;
}
```
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 L or R child is null
● That Node makes a new Node, sets the new Node’s data to the data to insert, and sets child reference to this new Node
● Runtime is $O(\log_2 N)$, yay!
  ○ $O(\log_2 N)$ to search the nearly balanced tree to find the place to insert
  ○ constant time operations to make new Node and link it in
Insertion into a BST (2/2)

- Example: Insert 115

Before:

```
      100
     /   
   125    200
  /  
 50  150
```

After:

```
      100
     /   
   125    200
  /  
 50  150
    /  
   115  175
```
Insertion Code in BST

Again, we use a “Smart Node” approach and delegate

```java
// Tree’s insert delegates to root
public Node<Type> insert(Type newData) {
    // if tree is empty, make first node. No traversal necessary!
    if (this.root == null) {
        this.root = new Node(newData, null); // root’s parent is null
        return this.root;
    } else {
        // delegate to Node’s insert() method
        return this.root.insert(newData);
    }
}
```
public Node<Type> insert(Type newData) {
    //insert method continued!
    if (this.data.compareTo(newData) > 0) {
        //newData should be in left subtree
        if (this.left == null) {
            //left child is null - we’ve found the place to insert!
            this.left = new Node(newData, this);
            return this.left;
        } else {
            //keep traversing down tree
            return this.left.insert(newData);
        }
    }
    else {  //newData should be in right subtree
        if (this.right == null) {
            //right child is null – we’ve found the place to insert!
            this.right = new Node(newData, this);
            return this.right;
        } else {
            //keep traversing down tree
            return this.right.insert(newData);
        }
    }
}

Reference to the new Node is passed up the tree so it can be returned by the tree
Insertion Simulation (1/4)

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

  ```java
  this.root = this.root.insert(newData);
  ```

![Diagram showing insertion into a binary search tree]
Insertion Simulation (2/4)

- **123** says: “I am less than **224**. I’ll let my right child deal with it.”

```java
if (this.data.compareTo(newData) > 0) {
    //code for inserting left elided
} else {
    if (this.right == null) {
        //code for inserting with null right child elided
    } else {
        return this.right.insert(newData);
    }
}
```
• 252 says: “I am greater than 224. I’ll pass it on to my left child – but my left child is null!”

```java
if (this.data.compareTo(newData) > 0) {
    if(this.left == null) {
        this.left = new Node(newData, this);
        return this.left;
    } else {
        //code for continuing traversal elided
    }
}
```
• 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. Lastly, I will return a pointer to the new left node”. (And each node, as its recursive invocation ends, passes the pointer to the new 224 node up to its parent, eventually up to whatever method called on the tree’s search)

```java
this.left = new Node(newData, this);
return this.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 $\log_2 N$ for a nearly balanced tree, which is much smaller than $N/2$!)
Postorder Traversal of BST

- Postorder traversal
  - “post-order” because self is visited after (“post-”) visiting children
  - again, use recursion!

```java
public void postOrder() {
    //Check for null children elided
    this.left.postOrder();
    this.right.postOrder();
    System.out.println(curr.data);
}
```

To learn more about the exciting world of trees, take CS200 (CSCI0200): Program Design with Data Structures and Algorithms!
Tree Runtime

- Binary Search Tree has a search of $O(\log_2 n)$ runtime → can we make it faster?
- Could make a ternary tree! (each node has at least 3 children)
  - $O(\log_3 n)$ runtime
- Or a 10-way tree with $O(\log_{10} n)$ runtime
- Let’s try the runtime for a search with 1,000,000 nodes
  - $\log_{10} 1,000,000 = 6$
  - $\log_2 1,000,000 < 20$, so shallower but broader tree
- Analysis: the logs are not sufficiently different and the comparison (basically an $n$-way nested if-else-if) is far more time consuming, hence not worth it
- Furthermore, binary tree makes it easy to produce an ordered list
Sets and Maps
Introducing… Sets

• A set is a collection of unique, unordered elements
  o no duplicates
  o \[A = \{2,3,5\} = \{5,3,2\}\]
  o \(A, B\) can be single elements or sets of multiple elements

• Basic operations of the Set data structure:
  o add element to set
  o remove element from set
  o check if element is in set
  o **Union**: merge two sets together
    ▪ ex: Union set contains students who are CS15 students or graduate students (or both – inclusive or)
  o **Intersection**: Intersection set contains only elements in two sets that are in both
    ▪ ex: students who are both CS15 students and graduate students
Set Data Structure (1/2)

- Sets can be implemented using arrays, lists, hashing (coming up), etc.
- No indices, no random access
- Useful for:
  - checking if elements of one collection are also a part of another collection (e.g., finding all students in CS15 who are also taking ECON0100). Since there is no explicit intersection operator in Java, we must loop through the elements of the smaller set, and check membership in the larger set
  - prevent an array from storing duplicates by checking an element to be inserted against a set of previously encountered names: if it is already in the set, it is a duplicate, if not, enter it into array and set. The win is in the efficiency of checking if an element is in a set (O(1)) versus having to search for it in the array (O(N))
Set Data Structure (2/2)

- Because there is no order/index, Sets can be implemented differently than Lists and other data structures we have shown so far.
- Java has a class `java.util.HashSet<Type>` specialized for set operations. This class implements the Set interface and is backed by a Hash Table.
HashSet Methods (1/2)

/* Constructor returns new HashSet capable of holding elements of type Type. 
* Java will let us create non-homogeneous sets, but we rarely want this, so 
* specify use the generic Type to enforce homogeneity */
public HashSet<Type>()

/* adds element e to HashSet, if not already present (returns false if 
* element is already present) */
public boolean add(Type e)

/* returns true if this set contains the specified element. 
* note on parameter type: Java accepts any Object since the elements of 
* your set could be any object, but you should supply one of type Type 
* for good programming practices */
public boolean contains(Object o)
HashSet Methods (2/2)

//removes all elements from this set
public void clear()

//returns true if this set contains no elements
public boolean isEmpty()

/*removes specified element from this set if present
 *note on parameter type: Java accepts any Object since the elements of
 *your set could be any object, but you should supply one of type Type*/
public boolean remove(Object o)

//returns the number of elements in this set
public int size()

//see JavaDocs for more methods, including set union and intersection
Iteration over a HashSet

- You can also iterate over elements stored in a HashSet by using a `for-each` loop.
  - as it is a set, there is no guaranteed order of processing elements

```java
HashSet<String> strings = new HashSet<String>();

//elided adding elements to the set

for (String s: strings) { //in HashSet strings, of type String, for each element s
    System.out.println(s); //prints all Strings in HashSet
}
```
HashSet Example

//somewhere in your app
HashSet<String> springCourses = new HashSet<String>();
springCourses.add(“BIOL0200”);
springCourses.add(“ECON0110”);
//elided adding rest of Banner

//in another part of your program
if (springCourses.contains(“CS0200”)){
    System.out.println(“I can take cs200 next semester!”);
}
//elided checking for other classes

As we will see, each check for set membership takes just $O(1)$! i.e., no actual searching!
Introducing… Maps (1/3)

• Maps are used to store (key, value) pairs.
  o a key is used to lookup its corresponding value
• (Word, Definition) in a dictionary
• (Brown ID, Person) in Banner
• (Name, Phone #) in a contacts list
• Think of a map as discrete function that maps from domain to co-domain
Introducing… Maps (2/3)

• Java provides `java.util.HashMap<K,V>` class
• Often called a “hash table”
• Other structures that provide maps include TreeMap, Hashtable, LinkedHashMap, and more
  o each has its own advantages and drawbacks
  o we will focus on HashMap
• HashMaps have constant time insert, removal, and search!—explained shortly
HashMap Syntax (1/2)

- Like other data structures, need to specify type of elements it holds
- This time need to specify type of both key AND value
- Key and value can be instances of any class

```java
new HashMap<KeyClass, ValueClass>();
```
- Only one entry for a given key - no duplicates
HashMap Syntax (2/2)

- If we wanted to map an Integer to its String representation
  ```java
  HashMap<Integer, String> intTable = new HashMap<>();
  ```

- If we wanted to map a TA to their Birthday
  ```java
  HashMap<CS15TA, Date> birthdayTable = new HashMap<>();
  ```

- In all cases, both key and value types must resolve to a type (e.g., class, interface)

- Note: Can’t use `int` or `boolean` as a type because they are *primitives*, not classes
  - so use a built-in class that is equivalent to that primitive, `Integer` or `Boolean` respectively
java.util.HashMap Methods (1/2)

 KD refers to type of Key, V to type of value. 
 * Adds specified key, value pair to the table, returns value. 
 * If there already was an entry for this key, it is replaced */
 public V put(K key, V value)

 /* returns value to which the specified key is mapped, or null */ 
 * if map contains no mapping for the key. 
 * note on parameter type: Java accepts any Object, but you should 
 * supply the same type as the key*/
 public V get(Object key)

 //returns the number of keys in this hashtable 
 public int size()
java.util.HashMap Methods (2/2)

/*note on parameter type: Java accepts any Object, but you
 *should supply the same type as the key.
 *Predicate tests if specified object is a key in this hash table*/

public boolean containsKey(Object key)

//returns true if hash table maps at least one key to this value
public boolean containsValue(Object Value)

/*removes key and its corresponding value from hash table,
 *returns value which the key mapped to or null if key had no mapping */

public V remove(Object key)

//more methods in JavaDocs
Finding out your friends’ logins (1/4)

• Given an array of CS students who have the properties “csLogin” and “real name”, how might you efficiently find out your friends’ logins?

• Givens
  o String[] friends, an array of your friends’ names
  o CSStudent[] students, an array of students with a “csLogin” and a “real name”
Finding out your friends’ logins (2/4)

• Old Approach:

```java
for (int i=0; i < friends.length; i++) { // for all friends
    for (int j=0; j < students.length; j++) { // for all students
        if (friends[i].equals(students[j].getName())) { // getName() code elided
            String login = students[j].getLogin(); // getLogin() code elided
            System.out.println(friends[i] + "'s login is " + login + "!");
        }
    }
}
```

• Note: Use `String` class’ `equals()` method because “==” checks for equality of reference, not of content

• This is $O(n^2)$—far from optimal
Finding out your friends’ logins (3/4)

- Better solution: use a `HashMap` to store students instead of an array:
  - key is name
  - value is login
  - use name to look up login!
Finding out your friends’ logins (4/4)

- Using a HashMap

```
HashMap<String, String> myTable = new HashMap<>();
for (CSStudent student : students){ //same array of students
    //getName() and getLogin() code elided
    myTable.put(student.getName(), student.getLogin()); //build HashMap
}
for (String friendName : friends){ //same array of friends
    String login = myTable.get(friendName); //look up friend’s login
    if (login == null){
        System.out.println("No login found for " + friendName);
        continue;
    }
    System.out.println(friendName + "'s login is " + login + "!");
}
```

- Each insert and search in HashMap is only O(1)!
Map Implementation (1/4)

• How do we implement a Map with constant-time insertion, removal, and search?

• In essence, we are searching through a data structure for value associated with key
  o similar to searching problem we have been trying to optimize

• Searching in an array:
  o unsorted array is $O(n)$
  o sorted array is $O(\log n)$, as is tree
    - remember binary partitioning of array in merge sort where tree depicting passes had depth of $\log_2 n$?; same for binary search tree?
  o can we do even better than $\log_2 n$?!? That would be $O(1)$!!!
    - yes: with hashing, but has limitations
Map Implementation (2/4)

- Try a radically different approach, using an array.
- What if we could directly use the key as an index to access appropriate spot in the array?
- Remember: digits, alphanumerics, symbols, even control characters are all stored as bit strings—“it’s bits all the way down…”
  - see ASCII table
  - bit strings can be interpreted as numbers in binary that can be used to index into an array to get oct or hex equivalent
  - $O(1)$ to find the key in array at given index!!!
Map Implementation (3/4)

- But creating an array to look up CS15 students (value) based on Banner ID # (key) would be a tremendous waste of space
  - if ID number is one letter followed by eight digits (e.g., B00011111), there are $10^8$ combinations!
  - do not want to allocate 100,000,000 words for no more than 400 students
  - (1 word = 4 bytes)
  - array would be terribly sparse…

- What about using social security number?
  - would need to allocate $10^9$ words, about 4 gigabytes, for no more than 400 students! And think about arbitrary names <30 chars: need $26^{30}$ !!
Map Implementation (4/4)

- Thus, two major problems:
  - how can we deal with arbitrarily long keys, both numeric \textit{and} alphanumeric?
  - how can we build a small, dense (i.e., space-efficient) array that we can index into to find keys and values?

- Impossible?
  - No, we approximate
Hashing

- How do we approximate?
  - we use hashing
  - hashing refers to computing an array index from an arbitrarily large key using a hash function
  - hash function takes in key and returns index in array

- Index leads to a simple value or an entire object
- Therefore, a two-step process:
  - hash to create index
  - use index to get value
Hashing

- Array used in hashing typically holds several hundred to several thousand entries; size typically a prime (e.g., 1051)
  - array of links to instances of the class HTA

Hash('Harriet')=0
Hash('Lila')=4
Hash('UV')=8
Hash('Will')=10
Hash('Daniel')=95
Hash Functions (1/4)

• An example of a hash function for alphanumerical keys
  o ASCII is a bit representation that lets us represent all alphanumerical symbols as integers
  o take each character in key, convert to integer, sum integers—sum is index
  o but what if index is greater than array size?
  o use mod, i.e. (index % arrayLength) to ensure final index is in bounds
    • think as if index is being “wrapped around”
  o note: hash functions are non-reversible, meaning can’t get original data from output of hash function
Hash Functions (2/4)

- Almost any reasonable function that uses all bits will do, so choose a fast one, and one that distributes more or less uniformly (randomly) in the array to minimize holes!

- A better hash function
  - take a string, chop it into sections of 4 letters each, then take value of 32 bits that make up each 4-letter section and XOR (exclusive OR) them together, then % (mod) that result by table size

- Will cover this more in CS200!
Hash Functions (3/4)

• We want to turn “harriet muutu” into an integer index for an array of size 101
  
  o Group into 4 character substrings
    o “harr” “ietm” “uutu”
  
  o Turn each character into ASCII
    o 110 111 97 104 | 107 111 114 111 | 116 122 101 114
  
  o Turn each ASCII character into binary
    o 01101110 01101111 01100001 01101000 | 01101011 01101111 01110010 01101111 | 01110100 01111010 01100101 01110010
Hash Functions (4/4)

• We want to turn “harriet muutu” into an integer index for an array of size 101
  
  - Turn each group into one value by mashing bits together
    - “harr” → 0110111001101111011000101101000
    - “ietm” → 0110101101101111011101001101111
    - “uutu” → 0111010001111010110010110110010
  
  - XOR the 3 groups together
    - 0110111001101111011000101101000 ^ 0110101101101111011101001101111 ^
      0111010001111010110010110110010 = 1110001011110100111011001110101
    - 1110001011110101100111011001110101 (binary) → 1903851125
  
  - Mod by size of list to ensure it’s within the array
    - Index = 1903851125 % 101 = 14
Collisions (1/2)

• If we have 6,000 Brown student names that we are mapping to Banner IDs using an array of size 1051, clearly, we are going to get “collisions” where different keys will hash to the same index.

• Does that kill the idea? No!

• Instead of having an array of type Value, we instead have each entry in the array be a head pointer to an overflow “bucket” for all keys that hash to that index. The bucket can be, e.g., our perennial favorite, the unsorted singly linked list, or an array, whatever…

• So, if we get a collision, the linked list will hold all values with keys associated to that bucket.
Collisions (2/2)

• Since collisions are frequent, for methods like `get(key)` and `remove(key)`, HashMap will have to iterate through all items in the hashed bucket to `get` or `remove` the right object

• This is $O(k)$, where $k$ is the length of a bucket – it will be small, so brute force search is fine

• The best hash functions minimize collisions

• Java has its own efficient hash function, covered in CS16

• A way to think about hashing: a fast, large initial division (e.g., 1051-way), followed by a brute force search over a small bucket—even bucket size 100 is fast!
HashMap Pseudocode

table = array of lists of some size
h = some hash function

public put(K key, V val):
    int index = h(key)
    table[index].addFirst(key, val)

public V get(K key):
    int index = h(key)
    /*search through (key, val) pairs in bucket at table[index] */
    for each (k, v) in table[index]:
        if k == key:
            return v
    return null //if not found, return null

O(1), if h() runs in O(1) time

Indexing with hash is $O(1)$, and buckets are usually well under 100, so linear search time is trivial, $O(1)$

Note: LinkedLists only hold one element per node, so in actual code, use instance of a class that holds key and value
HashMaps… efficiency for free?

- Not quite
- While `put()` and `get()` methods on average run in $O(1)$ time, each takes more time than inserting at the end of an `ArrayList`, for example
- A bit more memory expensive (array + buckets)
- Inefficient when many collisions occur (array too small)
- But it is likely the best solution overall, if you don’t need order
  - (key, value) pairs are stored in random order based on hash. The best hash is random to minimize collisions.
  - Trees can answer certain types of questions far more efficiently than a random hashmap (e.g., what is the value closest to a given value)
Hash Tables vs. Trees

- Hash Tables and Trees are different data structures used for different kinds of problems

- For just searching, insert/remove, a Hash Table will be faster
  - you know the exact key to search for
  - find a student’s Banner ID given their name, key is name and value is Banner ID

- If you’re trying to solve a nearest neighbors problem, a BST will be faster
  - find 4 people closest to a 95 in the class, key is grade and value is student name
  - if you’re trying to find the min and max in an array of numbers, a BST will be faster

- Can produce an already sorted list of data items by traversing the tree
Announcements

• Tetris deadlines
  o early handin: Saturday 11/13
  o on-time handin: Monday 11/15
  o late handin: Wednesday 11/17

• HTA Hours Friday 3-4pm (as always!) in Friedman 101

• Tetris check-ins will happen after Thanksgiving
Topics in Socially-Responsible Computing

CS for Social Good: Part II
Ameelio

- Currently: communication with incarcerated people is extremely expensive, sometimes a dollar per minute on video calls
- Expensive because private prison companies have monopoly on prison communication
- FCC commissioner called the prison communication system “the clearest, most glaring type of market failure I’ve ever seen as a regulator.” (TechCrunch)
- Can send messages for free to incarcerated people using their app
  - hard to figure out mailing addresses of prisons – Ameelio aggregates them and keeps updated
  - works with a service that will print out messages inputted in app and send them as physical letters
- From their site:
  - 113 million people in America have had an immediate family member incarcerated!
  - 400k+ lives impacted, 200k+ happy users, 3,000+ facilities served, 1m+ messages sent
- Plans to take on phone/video calling in future

Credit: TechCrunch
Upsolve

• It can cost $1,500 in court filing + attorney fees to declare bankruptcy!
• Totally different legal system for people without lawyers! No public defender given for civil suits
• Incubated by famed startup incubator Y Combinator as a nonprofit
• Upsolve automates filing process to file for bankruptcy without hiring a lawyer—take advantage of tech’s ability to scale
• Has helped relieve more than $250 million in total debt nationwide (TIME)
The Drivers Cooperative*

- Rideshare (Lyft/Uber)
  - drivers are underpaid / don’t have paid time off, regular salary, etc.
  - Prop 22: California Ballot Measure cos spent $200 million on to stop new labor legislation from applying to “app-based drivers”

- Drivers Co-Op: worker-owned alternative to Lyft/Uber
  - Wages 8-10% higher, all profits go back to drivers, rides designed to be 5% cheaper (and no surge pricing) because smaller overhead (FastCompany, NYMag)
  - Members of driver council shape how the app is developed + help hire tech team working on it
  - Co-Op ride (their app), currently only available in NYC
  - Early days!

*I am a (small-time) investor!

Credit: AppAdvice
Signal

- Encrypted instant messaging app
  - facilitates private communication with journalists (for whistleblowers!) / evades surveillance with tech tools (my bias!)
  - privacy/security tradeoff, concerns this also facilitates illicit activity outside of eye of the govt
- Nonprofit, advocacy work in addition to development
  - they made the banned Instagram Ads you saw in section!
  - “The subpoena requested a wide variety of information that fell into this nonexistent category, including the addresses of the users, their correspondence, and the name associated with each account…Just like last time, we couldn’t provide any of that. It’s impossible to turn over data that we never had access to in the first place.” — Signal blog, in response to government subpoena
- Funded initially by $105 million loan from founder of WhatsApp (Brian Acton) who has spoken out against Facebook after WhatsApp’s acquisition
- “[Founder Moxie Marlinspike] is acutely aware that the reason encryption did not catch on in the nineties was that the cypherpunks expected users to adopt the conventions of software engineers, rather than the other way around.” — New Yorker
Some news….

• “[Meta] planned to eliminate advertisers’ ability to target people with promotions based on their interactions with content related to health, race and ethnicity, political affiliation, religion, sexual orientation and thousands of other topics.”

• Takes effect Jan 19
More reading that may be of interest!

- Ameelio: Transforming corrections with Technology
- “Ameelio wants to take on for-profit, prison-calling rackets after starting with free letters to inmates” — TechCrunch
- “Upsolve is helping Americans understand that bankruptcy is a core part of their identity” — Yahoo
- “⚡ Putting Workers in the Driver's Seat” — Reboot
- “A ridehailing platform owned by workers, not billionaire founders and venture capital” — WeFunder
- “A Worker-Owned Cooperative Tries to Compete With Uber and Lyft” — NYTimes
- “Why Your Uber Ride Is Suddenly Costing a Fortune” — NY Mag
- “Grand jury subpoena for Signal user data, Central District of California” — Signal
- “Taking Back Our Privacy” — New Yorker
- “Meta plans to remove thousands of sensitive ad-targeting categories.” — NYTimes
Extra Optional Material

A Sneak-Peak Toward CS200!
Building a Node List
How To Build A Node List

• Now that we have a building block, there are a number of methods we can implement to make a higher-level NodeList that implements Java’s List interface (like ArrayList does)
  o note: List interface is very general…

• Main addition List mandates is to support indexing into the NodeList. Let’s write one of the simpler ones:
  o get(int i) method that returns element (Type) at that index
**search** Private Helper Method

- First, define a `search` helper method to return node at a particular index

- Want to use this helper method in the class, but don’t want to expose found nodes publicly; that would violate encapsulation - make helper `private`

- If a provided index is out of bounds, return `null` (print line is an optional error message)

- Otherwise, iterate through list until node at desired index is reached and return that node

```java
class NodeList<Type> {  
    //constructor elided  
    private Node<Type> search(int i) {  
        if(i < 0 || i >= this.size) {  
            System.out.println("Invalid index");  
            return null;  
        }  
        Node<Type> curr = this.head;  
        //for loop stops at i; pointer-chase to i  
        for (int counter = 0; counter < i; counter++) {  
            curr = curr.getNext();  
        }  
        return curr;  
    }
}
```
Private Helper Method Runtime

```java
private Node<Type> search(int i) {
    if (i >= this.size || i < 0) {
        // 1 op
        System.out.println("Invalid index");
        // 1 op
        return null;
        // 1 op
    }

    Node<Type> curr = this.head;
    // 1 op
    for (int counter = 0; counter < i; counter++) {
        // n ops
        curr = curr.getNext();
    }

    return curr;  // 1 op
}
```

→ \(\text{search}(\text{int } i) \text{ is } O(n)\)
Public Wrapper Method

• Write the publicly accessible wrapper code for the NodeList’s `get` method
  
  o this shows a very common pattern of “thin wrappers” over private code

```java
//inside NodeList
public Type get(int i) {
    return this.search(i).getElement();
}
```
Reversing a Linked List
An Exercise (common job interview question)

• Write a method that reverses the order of a `MyLinkedList`.
Solution A

• If list is empty or has 1 node, return list

• Otherwise, create a new list of same type as input list

• Iterate through input list, removing first element each time and adding it as first element of new list

```java
public MyLinkedList<Type> reverse(MyLinkedList<Type> toReverse) {
    if (toReverse.size() < 2) {
        return toReverse;
    }

    MyLinkedList<Type> newList = new MyLinkedList<Type>();
    int origSize = toReverse.size();

    while (newList.size() < origSize) {
        newList.addFirst(toReverse.removeFirst());
    }

    return newList;
}
```
Solution B (1/2)

• Is there a better way?

• First algorithm reversed in $O(n)$ time
  o but it wasn’t “in-place” – (had to create a new list)
  o memory use is also $O(n)$

• Can write a method within `MyLinkedList` that reverses itself without creating new nodes
  o still $O(n)$ but in-place and therefore more efficient
Solution B (2/2)

- Keep track of previous, current, and next node
- While current node isn’t null, iterate through nodes, resetting node pointers in reverse
- In doing so, must be careful not to delete any references further on in the list
- Finally, set the head pointer to what had been the last node (held in the prev variable)

- If the list is empty curr will be null, so the loop will never begin and head will continue to point to null

```java
public void reverse() {
    Node<Type> prev = null;
    Node<Type> curr = this.head;
    Node<Type> next = null;
    //set tail to head
    this.tail = this.head;

    while (curr != null) {
        next = curr.getNext();
        curr.setNext(prev);
        prev = curr;
        curr = next;
    }

    this.head = prev;
}
```
Solution B Walkthrough (1/15)

```
prev = null
curr
next = null
```

```
head

A

B

C
curr
tail

head
```
Solution B Walkthrough (2/15)

prev = null
curr = null
next = null

this.tail = this.head;
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}

prev = null

Solution B Walkthrough (3/15)
Solution B Walkthrough (4/15)

```
prev = null
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
```
Solution B Walkthrough (5/15)

```java
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
```
Solution B Walkthrough (6/15)

while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}

A

B

C

null

head

tail

prev

next

curr

Andries van Dam © 2021 11/11/21
Solution B Walkthrough (7/15)

```java
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
```

![Diagram showing linked list with nodes A, B, and C, and the loop for traversing the list.](image)
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
Solution B Walkthrough (9/15)

```java
while (curr != null) {
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
```
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}
Solution B Walkthrough (11/15)

while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}

A -> B

next = null
Solution B Walkthrough (12/15)

while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}

tail
head
A
B
C
prev
curr
null
next = null
null

while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}

head

while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}

curr

next = null

Solution B Walkthrough (13/15)
while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}

head

null

tail

while(curr!=null){
    next = curr.getNext();
    curr.setNext(prev);
    prev = curr;
    curr = next;
}

curr = null
next = null

A

B

C

prev

Andries van Dam © 2021 11/11/21
null

null = null

this.head = this.prev;

head

tail

prev

curr = null

got it, next = null
Using a Stack to Reverse a Linked List
Reverse a Linked List with a Stack (1/4)

● How can we use a **Stack** to reverse a Linked List?

● Linked List: Montana, Jessica, Camilla, Amber

● Note: user wouldn’t see **head** and **tail** – implementation detail
Reverse a Linked List with a Stack (2/4)

● Solution:
  
  o while Linked List is not empty, remove from Linked List and push elements onto Stack

  o then, while Stack is not empty, pop elements from Stack and add to Linked List
Reverse a Linked List with a Stack (3/4)

while(!this.list.isEmpty()) {
    stack.push(this.list.removeFirst());
}

Stack

head
Montana  Jessica  Camilla  Amber
tail
Reverse a Linked List with a Stack (4/4)

while(!this.list.isEmpty()) {
    stack.push(this.list.removeFirst());
}

head tail   tail
            head tail
            Null

stack: [Amber, Camilla, Jessica, Montana]
Check for Balanced Parentheses
Check for Balanced Parentheses (1/2)

● Check for balanced parentheses in a given string

● Balanced: [()()]{[()]} }

● Not balanced: [()]
Check for Balanced Parentheses (1/2)

- Go through every character, if it is a starting bracket, push it onto the stack
- If it is a closing bracket, pop from the stack
  - if stack is empty, return false
- The bracket you pop should be the opening bracket that corresponds to the closing bracket you are looking at
  - if it is not, return false
- If you get through every character and you haven’t returned false, check if stack is empty
- If it is, the brackets are balanced!
Check for Balanced Parentheses Pseudocode

for each bracket in string:
    if it is a starting bracket:
        push it onto stack
    if it is a closing bracket:
        pop from the stack
        if the popped character is not the matching opening bracket:
            return false
    if stack is empty:
        return true
for each bracket in string:
  if it is a starting bracket:
    push it onto stack
  if it is a closing bracket:
    pop from the stack
    if the popped character is not the matching opening bracket:
      return false
  if stack is empty
    return true
for each bracket in string:
  if it is a starting bracket:
    push it onto stack
  if it is a closing bracket:
    pop from the stack
    if the popped character is not the matching opening bracket:
      return false
  if stack is empty
    return true
for each bracket in string:
    if it is a starting bracket:
        push it onto stack
    if it is a closing bracket:
        pop from the stack
        if the popped character is not the matching opening bracket:
            return false
if stack is empty
    return true

[ ( ) ]

Match! Keep going…

Stack
for each bracket in string:
  if it is a starting bracket:
    push it onto stack
  if it is a closing bracket:
    pop from the stack
    if the popped character is not the matching opening bracket:
      return false
if stack is empty
  return true

[ ( ) ]

Match! Keep going…

Stack
for each bracket in string:
  if it is a starting bracket:
    push it onto stack
  if it is a closing bracket:
    pop from the stack
    if the popped character is not the matching opening bracket:
      return false
if stack is empty
  return true

[ ( ) ]
for each bracket in string:
    if it is a starting bracket:
        push it onto stack
    if it is a closing bracket:
        pop from the stack
        if the popped character is not the matching opening bracket:
            return false
if stack is empty
    return true
Exercise 2 Actual Code

```java
for (int i=0; i<parenthesesArray.length; i++) {
    // If the element at this index is either starting bracket, push onto stack
    if (parenthesesArray[i].equals("[") || parenthesesArray[i].equals("(") ) {
        myStack.push(parenthesesArray[i]);
    }
    // If the element at this index is either closing bracket, pop off of stack
    // Note use of built-in equals() method to compare Strings - returns a boolean
    if (parenthesesArray[i].equals("]") || parenthesesArray.equals(")") ) {
        String popped = myStack.pop();
        if (parenthesesArray[i].equals("") && !popped.equals("(") ) {
            return false;
        } else if (parenthesesArray[i].equals("]") && !popped.equals("[" ) {
            return false;
        }
    }
}
if (myStack.isEmpty()) {
    return true;
}
```
Model TA Hours Line
TA Hours Line (1/2)

• Let’s model the TA hours line
• Because it is FIFO, we need to use a queue!
• What functionality do we need?
  o a method for students to be added to the line
  o a method for TAs to help the line until it is empty
TA Hours Line (2/2)

- Start by initializing `queue` and `ta`
- Define a method for adding to hours line
  - this can be used before hours or during hours to sign up
- Define a method for seeing a student – uses `CS15TA`'s `help()`
- Define a method for emptying the queue
  - useful after the cutoff is set

```java
public class TAHoursLine{
    private Queue<Student> queue;
    private CS15TA ta;

    public TAHoursLine(CS15TA ta){
        this.queue = new Queue<Student>();
        this.ta = ta;
    }

    public Student addToLine(Student s){
        return this.queue.enqueue(s);
    }

    public void seeStudent(){
        this.ta.help(this.queue.dequeue());
    }

    public void holdHoursUntilCutoff(){
        while(!this.queue.isEmpty()){
            this.seeStudent();
        }
    }
}
```
Traversing Trees
Traversing a Binary Tree

- We often want to access every Node in tree
  - so far, we have only searched for a single element
  - 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!
- If we print our current node’s data, this will print an alphabetical list!

```java
public void inOrder() {
    //Check for null children elided
    this.left.inOrder();
    System.out.println(this.data);
    this.right.inOrder();
}
```
Preorder Traversal of BST

- **Preorder traversal**
  - “preorder” because self is visited before (“pre”) visiting children
  - again, use recursion!
  - note that we can recover the tree structure using the preorder result

```java
public void preOrder() {
    //Check for null children elided
    System.out.println(this.data);
    this.left.preOrder();
    this.right.preOrder();
}
```
Postorder Traversal of BST

- **Postorder traversal**
  - “post-order” because self is visited after (“post-”) visiting children
  - again, use recursion!

```java
public void postOrder() {
    //Check for null children elided
    this.left.postOrder();
    this.right.postOrder();
    System.out.println(curr.data);
}
```

To learn more about the exciting world of trees, take CS200 (CSCI0200): Program Design with Data Structures and Algorithms!
Counting Frequency in an Array
Counting frequency in an Array (1/4)

• How many times does a given word show up in a given string?
• Consider a book as one long String. That’s too hard to search, so let’s chop the string into individual words using punctuation as a separator and put each word in an array

• Givens
  o String[] _book, an array of Strings, each an individual word
  o String searchTerm, the word you’re looking for
public void frequency(String searchTerm) {
    int wordCounter = 0; // frequency of single term
    for (String word : _book){
        if (word.equals(searchTerm)){
            wordCounter++;
        }
    }
    System.out.println(searchTerm + " appears " + wordCounter + " times");
}
Counting frequency in an Array (3/4)

- When tracking one word, code is simple
- But what if we wanted to keep track of 5 words? 100?
- Should we make instance variables to count the frequency of each word? For each term in the book?
  - should we iterate through book for each of the search terms? Sounds like $O(n^2)$...
HashMap<String, Integer> countMap = new HashMap<String, Integer>();
/* _book is an array of words.
 * If currWord in _book matches a search term,
 * put currWord back with updated count. By using
 * put(), we replace current entry in hashMap.
 * Note use of Integer rather than int because you
 * can’t use base types as generics */
for (String currWord : _book){
    if (countMap.containsKey(currWord)){
        Integer count = countMap.get(currWord);
        count++;
        countMap.put(currWord, count);
    }
    else{
        //First time seeing word
        countMap.put(currWord, 1);
    }
}
/*separate method: searchTerms is now an array of Strings we’re counting */
public void frequencies(String[] searchTerms) {
    for (String word : searchTerms){
        Integer freq = 0;
        if (countMap.get(word) != null){
            freq = countMap.get(word);
        }
        System.out.println(word + “ shows up ” + freq + “ times!”);
    }
}

Despite increase in search terms, still $O(n)$