Review: Accessing Data Items

- Variables: named by identifiers
  - local variables
  - parameters
  - instance variables
- Indexed items: named by index
  - in Arrays
  - in ArrayLists
- Referenced items: "named" by pointer
  - next and element in nodes
**Review: Accessing Nodes Via Pointers**

- **this.head.getNext();**
  - This does not get `next` field of `head`, which doesn’t have such a field, being just a pointer.
  - Instead, read this as “get `next` field of the node `head` points to.”
  - What does `this.tail.getNext()` produce?
  - What does `this.tail.getElement()` produce?
- Node we can access a variable by its unique name, index, contents, or here, via a pointer.

---

**remove Method**

- We have implemented methods to remove first and last elements of `MyLinkedList`.
- What if we want to remove any element from `MyLinkedList`?
- Let’s write a general `remove` method:
  - Think of it in 2 phases:
    - a search loop to find correct element (or end of list)
    - breaking the chain to jump over the element to be removed.

---

**remove Method**

- Search loop through nodes until an element matches `itemToRemove`.
- “Jump over” node by re-linking predecessor of node (using loop’s `prev` pointer) to successor of node (via its `next` reference).
- With no more reference to node, it is garbage collected at termination of method.
Andries van Dam
2023 11/07/23

Edge Case(s)
- again: can't delete from empty list
- if removing first item or last item, delegate to `removeFirst`/`removeLast`

General Case
- iterate over list until `itemToRemove` is found in `ptr`
- again: need `prev`, so we can re-link predecessor of `curr` node in GC on return

```java
public Node<Type> remove(Type itemToRemove) {
    if (this.isEmpty()) {
        System.out.println("List is empty");
        return null;
    }
    if (itemToRemove.equals(this.head.getElement())) {
        return this.removeFirst();
    }
    if (itemToRemove.equals(this.tail.getElement())) {
        return this.removeLast();
    }
    // advance to 2nd item
    Node<Type> curr = this.head.getNext();
    Node<Type> prev = this.head;
    while (curr != null) {
        // pointer-chasing loop to find ele.
        if (curr.getElement().equals(itemToRemove)) {
            prev.setNext(curr.getNext());
            this.size--;
            return curr;
        }
        prev = curr;
        curr = curr.getNext();
    }
    return null; // return null if itemToRemove is not found
}
```

Note: caller of `remove` can find out if `item` was successfully found (and removed) by testing for `!= null`

remove Method

remove Runtime

TopHat Question

Given that `animals` is a Singly Linked List of animals, `curr` points to the node with an animal to be removed from the list, that `prev` points to `curr`'s predecessor, and that `curr` is not the tail of the list, what will this code fragment do?

A. List is unchanged, prints out removed animal
B. List is unchanged, prints out the animal after the one that got removed
C. List loses an animal, prints out removed animal
D. List loses an animal, prints out the animal after the one that was removed
Insertion in a Sorted MyLinkedList

- We search a LinkedList on a "key," an alphanumeric
- Search for first node that nodeToInsert's key (7) < node's key
- Break chain by making predecessor's next link to nodeToInsert and have its next point to successor node

Doubly Linked List (1/3)

- Is there an easier/faster way to get to previous node while removing a node?
  - with Doubly Linked Lists, nodes have references to both next and previous nodes
  - can traverse list both backwards and forwards -- Linked List still stores reference to front of list with head and back of list with tail
  - modify Node class to have two pointers: next and prev
  - eliminates pointer chasing loop because prev points to predecessor of every node, at cost of second pointer
  - classic space-time tradeoff
Doubly Linked List (2/3)

- For Singly Linked List, processing typically goes from first to last node, e.g., search, finding place to insert or delete.
- Sometimes, particularly for sorted list, need to go in the opposite direction.
  - E.g., sort CS15 students on their final grades in ascending order. Find lowest numeric grade that will be recorded as an “A”. Then ask: who has a lower grade but is closer to the “A” cut-off, i.e., in the grey area, and therefore should be considered for “benefit of the doubt”?  

Doubly Linked List (3/3)

- This kind of backing-up can’t easily be done with the Singly Linked List implementation we have so far.
  - Could build our own specialized search method, which would scan from the head and be, at a minimum, O(n).
- It is simpler for Doubly Linked Lists:
  - Find student with lowest “A” using search
  - Use prev pointer, which points to the predecessor of a node (O(1)), and back up until hit end of B+/A grey area.
Stacks and Queues
Abstractions that are Wrappers for MyLinkedList

Outline
• Stacks and Queues
• Trees

Stacks
• Stack has special methods for insertion and deletion, and two others for size
  o push and pop
  o isEmpty, size
• Instead of being able to insert and delete nodes from anywhere in the list, can only add and delete nodes from top of Stack
  o LIFO (Last In, First Out)
• We’ll implement a stack with a linked list
Methods of a Stack

- Add element to top of stack
- Remove element from top of stack
- Returns whether stack has any elements
- Returns number of elements in stack

```
public void push(Type el)
public Type pop()
public boolean isEmpty()
public int size()
```

Stack Constructor

- When generic `Stack` is instantiated, it contains an empty `MyLinkedList`
- When using a stack, you will replace `Type` with type of object your `Stack` will hold – enforces homogeneity
- Note: `Stack` uses classic "wrapper" pattern to modify functionality of the data structure, `MyLinkedList`, and to add other methods

```
public class Stack<Type> {  
  private MyLinkedList<Type> list;
  public Stack() {
    this.list = new MyLinkedList<>();
  }
  /* other methods elided */
}
```
Implementing Push

Let's see behavior...
- When element is pushed, it is always added to front of list
- Thus, Stack delegates to the MyLinkedList, this.list to implement push

```java
//in the Stack<Type> class ...
public Node<Type> push(Type newData) {
    return this.list.addFirst(newData);
}
```

Implementing Pop

Let's see what this does...
- When popping element, it is always removed from top of Stack, so call removeFirst on MyLinkedList – again, delegation
- removeFirst returns element removed, and Stack in turn returns it
- Remember that removeFirst method of MyLinkedList first checks to see if list is empty

```java
//in the Stack<Type> class ...
public Type pop() {
    return this.list.removeFirst();
}
```

isEmpty

Stack will be empty if the MyLinkedList, list, is empty – delegation
- Returns true if Stack is empty, false otherwise

```java
//in the Stack<Type> class ...
public boolean isEmpty() {
    return this.list.isEmpty();
}
```
size

- Size of Stack will be number of elements that the MyLinkedList, list contains – delegation
- Size is updated whenever Node is added to or deleted from list during push and pop methods.

```java
//in the StackType class ...
public int size() {
    return this.list.size();
}
```

TopHat Question

Look over the following code:

Who’s left in the stack?

Stack mein fill: myStack = new Stack();
myStack.push(htaSarah);
myStack.push(htaAllie);
myStack.pop();
myStack.push(htaCannon);
myStack.pop();

A. htaSarah
B. htaAllie
C. htaCannon
D. none of them!

Example: Execution Stacks

- Each method has an Activation Record (AR) – recall recursion lecture
  - contains execution pointer to next instruction in method
  - contains all local variables and parameters used by method
- When methods execute and call other methods, Java uses a Stack to keep track of the order of execution: "stack trace"
  - when a method calls another method, Java adds activation record of called method to Stack
  - when new method is finished, its AR is removed from Stack, and previous method is continued
  - method could be different or a recursively called clone, when execution pointer points into same immutable code, but different values for variables/parameters
**Execution Stacks**

- A calls B
- B calls C
- ... etc.

When E finishes, its AR is popped. Then D's AR is popped, etc. Note this handles the tracking of invocations (clones) in recursion automatically.

---

**Stack Trace**

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

```plaintext
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 LinkedList, but user has no knowledge of that.
- Could also implement it with an Array or ArrayList:
  - Array implementation could be more difficult—Array's have fixed size, so would have to copy our Array into a larger one as we push more objects onto the Stack.
  - User's code should not be affected even if the implementation of Stack changes.
- 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 for merch at the Eras Tour
  - Waitlist for TA hours after randomization

Methods of a Queue

- Add element to end of queue: `public void enqueue(type e)`
- Remove element from beginning of queue: `public type dequeue()`
- Returns whether queue has any elements: `public boolean isEmpty()`
- Returns number of elements in queue: `public int size()`

Enqueuing and Dequeuing

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

Before Enqueuing: 1 2 3

<table>
<thead>
<tr>
<th>Before Enqueuing</th>
<th>After Enqueuing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head of queue</td>
<td>Head of queue</td>
</tr>
<tr>
<td>End of queue</td>
<td>End of queue</td>
</tr>
</tbody>
</table>

After Enqueuing: 1 2 3 4

<table>
<thead>
<tr>
<th>Before Enqueuing</th>
<th>After Enqueuing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head of queue</td>
<td>Head of queue</td>
</tr>
<tr>
<td>End of queue</td>
<td>End of queue</td>
</tr>
</tbody>
</table>
Enqueuing and Dequeuing

- Enqueuing: adds a node to the back
- Dequeuing: removes a node from the front

Before Dequeuing:
1  2  3  4

Head of queue

Tail of queue

After Dequeuing:
1

Head of queue

Tail of queue

● Enqueuing: adds a node to the back
● Dequeuing: removes a node from the front

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

public class Queue<Type> {
  private MyLinkedList<Type> list;
  public Queue() {
    this.list = new MyLinkedList<>();
  }
  // Other methods elided
}

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

enqueue

- Just call list’s addLast method – delegation

    public void enqueue(Type newNode) {
      this.list.addLast(newNode);
    }

- This will add newNode to end of list
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 our wrapped MyLinkedList
  ```java
  public int size() {
      return this.list.size();
  }
  ```
  ```java
  public boolean isEmpty() {
      return this.list.isEmpty();
  }
  ```

TopHat Question

In order from head to tail, a queue contains the following: katniss, gale, finnick, beetee. 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. katniss, gale, finnick, beetee  
B. katniss, beetee, gale, finnick  
C. beetee, finnick, gale, katniss  
D. It's random every time.
Outline
- Stacks and Queues
- 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)
  - 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 Nth 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 improve sequential search
- Remember merge sort with search O(logN) where we did “bisection” on the array at each pass
- If we had a sorted array, we could do the same thing
  - start in the middle
  - keep bisecting array, deciding which portion of the sub-array the search key lies in, until we find the key or can’t bisect further (not in array)
  - For N elements, search time is O(logN) (since we reduce number of elements to search by half each time), very efficient!
Binary Search (2/4)
- $\log N$ grows much more slowly than $N$, especially for large $N$.

<table>
<thead>
<tr>
<th>$N$ (int)</th>
<th>$\log(N)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>1,000</td>
<td>3</td>
</tr>
<tr>
<td>10,000</td>
<td>4</td>
</tr>
<tr>
<td>100,000</td>
<td>5</td>
</tr>
<tr>
<td>1,000,000</td>
<td>6</td>
</tr>
</tbody>
</table>

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

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 either 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 linked lists is insertion/deletion 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: T₁, T₂, …, Tₙ
  - each Tᵢ 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
    - Pane contained by root pane, which is contained by Scene

Tree Structure

- Note that the tree structure has meaning
  - any subtree of T, T, 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.
  - i.e., 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 one or two binary trees as children (left subtree, 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 – 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.
- 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.
  - Level

Properties of Binary Trees (2/2)

- In a full Binary Tree: (# leaf nodes) = (# internal nodes) + 1
- In a perfect Binary Tree: (# nodes at level i) = 2
- In a perfect Binary Tree: (# leaf nodes) <= 2^i
- In a perfect Binary Tree: (height) >= log(# 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!

BST (2/2)

- 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?
  - much the same as sorted linked lists: insert, remove, size, empty
  - 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...
  - 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)
  - you’ll learn more about implementing data structures in CS200!
Nodes, data items, and keys

- Item is a composite that can contain many properties,
- One of which is a key that Nodes are sorted by (here, ISBN #)

Comparable Book Class

```java
public class Book {
    // variable declarations, elided
    public Book(String author, String title, int isbn) {
        // variable initializations elided
    }
    public int getISBN() {
        return this.isbn;
    }
    // other methods elided
}
```

In our example, we use Book as Type
- Our BinarySearchTree stores objects of type Book, meaning we will be able to use all methods Book has within our BST

BinarySearchTree Class (2/4)

```java
public class BinarySearchTree<Book> {
    private Node<Book> root;
    public BinarySearchTree(Book item) {
        // Root of the tree
        this.root = new Node(item, null);
    }
    // other methods shown next slide

    class ComparableBook { // Placeholder for ComparableBook details
        public int compareBooks(Book toCompare) {
            return (this.isbn - toCompare.getISBN());
        }
    }
}
```

In our example, we use Book as Type
BST Class (3/4)

```java
public class BinarySearchTree {
    private Node<Book> root;

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

    public void insert(Book newItem) {
        // . . .
    }

    public void remove(Book itemToRemove) {
        // . . .
    }

    public Node<Book> search(Book itemToFind) {
        // . . .
    }

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

// End of class

---

BST Class (4/4)

● Our implementations of LinkedLists, 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
  - The 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 item, parent, and children, and can perform operations such as insert and remove

---

BST: Node Class (1/3)

● “Smart” Node includes the following methods:

  ```java
  public Node<Book> search(Book itemToFind);
  public Node<Book> insert(Book newItem);
  public Node<Book> remove(Book itemToRemove);
  ```

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

- Nodes have a maximum of two non-null children that hold data:
  - Four instance variables: `item, parent, left, and right`, with each having a `get` and `set` method
  - `item` represents the data item 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 items are less than `node`’s data item
  - `right` represents `node`’s right child and contains a subtree, all of whose data items are greater than `node`’s data item
  - Arbitrarily select which child should contain items equal to `node`’s data item

```java
public class Node<Book> {
    private Book item;
    private Book parent;
    private Node<Book> left;
    private Node<Book> right;

    public Node(Book myItem, Node<Book> parent) {
        // construct a leaf node as default
        this.item = myItem;
        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…
}
```

**BST: Node Class (3/3)**

- 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
    public Node<Book> search(Book itemToFind) {
        return this.root.search(itemToFind);
    }
    ```
  - 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
Let's Search a BST

For a detailed step-by-step walkthrough of this algorithm, see slide 100.

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 visit both the left and right subtrees
C. $O(n/2)$ – because you incorporate the idea of "bisection" to eliminate half the number of nodes to search at each recursion
D. $O(\log n)$ – 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 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.
- $O(\log_2 N)$ is much less than $N$, this is thus much more efficient.
Searching a BST Recursively

```java
public Node<Book> search(Book itemToFind) {
    // if itemToFind is the thing we're searching for
    if (this.item.compareBooks(itemToFind) == 0) {
        return this.item;
    // if item is > itemToFind, can only be in left tree
    } else if (this.item.compareBooks(itemToFind) > 0) {
        if (this.left != null) {
            return this.left.search(itemToFind);
        }
        // if item is < itemToFind, can only be in right tree
    } else if (this.right != null) {
        return this.right.search(itemToFind);
    }
    // only get here if itemToFind isn't in tree, otherwise would've returned sooner
    return null;
}
```

Let's Add to a BST (1/3)
For a step-by-step walkthrough of this algorithm, see slide 112

Let's Add to a BST (2/3)
Let's Add to a BST (3/3)

For a step-by-step walkthrough online, see slide 112

Insertion into a BST

- 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 $O(\log N)$, yay!

- $O(\log N)$ to search the nearly balanced tree to find the place to insert
- Constant time operations to make new Node and link it in

Insertion Code in BST

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

```java
public Node<Book> insert(Book newItem) {
  // if tree is empty, make first node. No traversal necessary!
  if (this.root == null) {
    this.root = new Node(newItem, null);
  } else {
    // delegate to node's insert() method
    return this.root.insert(newItem);
  }
}
```

- Smart Node delegation to root
**Insertion Code in Node**

```java
public Node<Book> insert(Book newItem) {
    if (this.item.compareBooks(newItem) > 0) {
        // newItem should be in left subtree
        if (this.left == null) {
            // left child is null
            // we've found the place to insert!
            this.left = new Node(newItem, this);
            return this.left;
        } else {
            // keep traversing down tree
            return this.left.insert(newItem);
        }
    } else {
        // newItem should be in right subtree
        if (this.right == null) {
            // right child is null
            // we've found the place to insert!
            this.right = new Node(newItem, this);
            return this.right;
        } else {
            // keep traversing down tree
            return this.right.insert(newItem);
        }
    }
}
```

Reference to the new node is passed up the tree so it can be returned by the tree.

---

**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 an 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.
  - With same nodes, different insertion order makes different tree.
  - An average, for reasonably random (unsorted) arrival order, trees will look similar in depth so order doesn't play a major role in runtime.

---

**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 log2N for a nearly balanced tree, which is much smaller than N/2).
Preorder Traversal of BST

- **Preorder traversal**
  - "pre-order" because self is visited before ("pre-") visiting children
  - again, use recursion!
  ```java
  public void preOrder() {
    if (curr != null) {
      System.out.println(curr.item);
      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() {
    if (curr != null) {
      this.left.postOrder();
      this.right.postOrder();
      System.out.println(curr.item);
    }
  }
  ```

Inorder Traversal of BST

- **Inorder traversal**
  - "in-order" because self is visited between ("in-") visiting children
  - again, use recursion!
  ```java
  public void inOrder() {
    if (curr != null) {
      this.left.inOrder();
      System.out.println(curr.item);
      this.right.inOrder();
    }
  }
  ```

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 $O(\log n)$ runtime, can we make it faster?
- Could make a ternary tree (each node has at least 3 children)
- 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

Prefix, Infix, Postfix Notation for Arithmetic Expressions (1/2)

- 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 parentheses and evaluates in a single left-to-right pass
- Use Dijkstra’s 2-stack shunting yard algorithm to convert from user-entered Infix to easy-to-handle Postfix — compile or interpret it on the fly (Covered in optional lecture Dec 6)

Prefix, Infix, Postfix Notation for Arithmetic Expressions (2/2)

- Infix, Prefix, and Postfix refer to where the operator goes relative to its operands:
  - Infix: (fully parenthesized)
    - $(1 \times 2 + (3 \times 4)) \times ((5 - 6) + (7 / 8))$
  - Prefix:
    - $- \times + 1 2 * 3 4 * + 5 6 7 8$
  - Postfix:
    - $1 2 3 4 * + 5 6 - 7 8 / + -$

Graphical representation for equation:
Announcements

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

- HTA Hours Friday 3-4pm (as always!) in CIT 210
  - come talk to us about which FP to do!
What’s the difference?

- Generative AI intelligence (GAI)
- Human intelligence
- Artificial general intelligence (AGI)

Comparing different types of intelligence

Singularity: Machine thinking is so advanced that it alters existing systems fundamentally.

Does the process matter?

Illustration of neural networks in our brain.

Source: ExtremeTech.com
The Turing Test

A. Turing

B. Human

C. Computer

John Searle – The Chinese Room

Chinese writing given to non-Chinese speaker

Detailed book of rules

Future of autonomy...

Increase in AI usage

Limited human decision-making
Wondering how to make a generic BST that can store more than just books?
Appendix

- Generic BST
- Searching Simulation
- Insertion Demonstration

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?

Nodes are sorted by ISBN:

```
BinarySearchTree root = null;
Node<Book>...
```

Java's Comparable<Type> interface (1/3)

- Previously we used == to check if two things are equal
  - this only works correctly for primitive data types (e.g., int), or when we are comparing two variables referencing the exact same object
  - to compare Strings, need a different way to compare things
- We can implement the Comparable<Type> generic interface provided by Java
  - It specifies the compareTo method, which returns an int
- Why don't we just use ==, even when using something like ISBN, which is an int?
  - can treat ISBN as int and compare them directly, but more generally we implement the Comparable<Type> interface, which could easily accommodate comparing 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 implementor of interface (`Book`, in our case)
  ```java
  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 unimportant, 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`:
  ```java
  public class Book implements Comparable<Book> {
    // variable declarations, e.g., isbn, elided
    public Book(String author, String title,
          int isbn){
      //variable initializations elided
    }
    public int getISBN(){
      return this.isbn;
    }
    //other methods elided
    //compare isbn of book passed in to stored one
    @Override
    public int compareTo(Book toCompare){
      return (this.isbn - toCompare.getISBN());
    }
  }
  ```
BST Class (2/4)

- Using keyword `extends` in this way ensures that `Type` implements `Comparable<Type>`
  - note nested `<`
  - nested `<` to show it modifies `Type`
- All elements stored in `MyLinkedList` must now have `compareTo` method for `Type`;
  - thus restricts generic

BST Class (3/4)

```java
class BinarySearchTree<Type extends Comparable<Type> {
    private Node<Type> root;
    public BinarySearchTree(Type data) {
        this.root = new Node(data, null);
    }
    public void insert(Type newData) {
        // . . .
    }
    public void remove(Type dataToRemove) {
        // . . .
    }
    public Node<Type> search(Type dataToFind) {
        // . . .
    }
    public int size() {
        // . . .
    }
    // end of class
```

BST Class (4/4)

- Our implementations of `LinkedList`, `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 perform operations such as `insert` and `remove`
BST: Node Class (1/3)

- "Smart" Node includes the following methods:
  - public Node<Type> search(Type dataToFind);
  - public Node<Type> insert(Type newData);
  - 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.

BST: Node Class (3/3)

```
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){
    // construct a leaf node as default
    this.data = data;
    this.parent = parent;
    this.left = null;
    this.right = null;
  }
}
```
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 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!

Appendix

- Generic BST
- Searching Simulation
- Insertion Demonstration

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!"
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.”

224 says: “That's me! Hey, caller (252) here's your answer.”

Answer: 224 is in the Tree!

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!

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(log₂N)

- Search path: start with root \textbf{M} and choose path to \textbf{I} (for a reasonably balanced tree, \textbf{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)
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.data;
    }
    // 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;
}
```

Appendix

- **Generic BST**
- **Searching Simulation**
- **Insertion Demonstration**

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 $O(\log N)$, yay!
  - $O(\log N)$ 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:

```
200
150
125
200
175
140
30
20
```

After:

```
200
150
125
175
140
85
75
20
30
80
115
```

Insertion Code in BST

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

```
// 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 {
        return this.root.insert(newData); // delegate to Node's insert() method
    }
}
```

Insertion Code in Node

```
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 {
            return this.left.insert(newData); // keep traversing down tree
        }
    } 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 {
            return this.right.insert(newData); // keep traversing down tree
        }
    }
}
```
Insertion Simulation (1/4)

- Insert: 224
- First call insert in BST:

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

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);
    }
}
```

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!"

```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
    }
}
```

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. Lastly, I will return a pointer to the new left node", yes 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;
```

Before

```
100
  /|
  | 123
  |
```

After

```
100
  /|
  | 123
  | 252
  /|
  | 16
  |
```