Overview

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

Searching in a Linked List (1/3)

- Searching for an element in a linked list 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
- To find the $i$th element in an array or ArrayList is random access (which means $O(1)$), but searching for a particular element even with an index is still sequential $O(N)$
- And with NodeLists, even though they support indexing dictated by Java’s list interface, finding the $i$th element is also done (under the hood) by pointer-chasing and hence is $O(N)$

Searching in a Linked List (2/3)

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

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

Binary Search (1/5)

- Worst case for searching sorted linked list is checking every element i.e., sequential access
- We can do better with a sorted array which allows random access at any index
- Let’s demo binary search (“bisection”) for a sorted list of numbers
- Website: http://www.cs.armstrong.edu/liang/animation/web/BinarySearch.html

Binary Search (2/5)

- For N elements, search time is $O(\log N)$ (since we reduce number of elements to search by half each time), very efficient
  - start in the middle
  - keep bisecting the array by halving the index, deciding which half interval the search key lies in, until we land on that key or can’t subdivide further (not in array)

Binary Search (3/5)

- $\log N$ is considerably smaller than $N$, especially for large $N$. So doing things in time proportional to $\log N$ is better than doing them in proportion to $N$!
Binary Search (4/5)

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

Binary Search (5/5)

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

Trees vs Linked Lists (1/2)

- Singly linked list – collection of nodes where each node references only one neighbor, the node’s successor.

Trees vs Linked Lists (2/2)

- Tree – also collection of nodes, but each node may reference multiple successors/children.
- Trees can be used to model a hierarchical organization of data.
**Technical definition of a Tree**

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

**Inheritance Hierarchies as Trees**

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

**Graphical Containment Hierarchies as Trees**

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

**Tree Terminology**

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

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

Note: These nodes are similar to the linked list nodes, with one data and two child pointers.

Properties of Binary Trees (1/2)

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

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.
Binary Search Tree (2/2)

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

Binary Search Tree Class (1/3)

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

Nodes, data, and keys

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

Binary Search Tree Class (2/3)

```java
public class BinarySearchTree<Type extends Comparable<Type> { ...}
```

-BinarySearchTree

- _root_ = _null_

-BinarySearchTree

- _root_ = _null_

-BinarySearchTree

- _root_ = _null_

-BinarySearchTree

- _root_ = _null_

public void remove(Type dataToRemove) {
    //...}
Binary Search Tree Class (3/3)

- 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 `prev` and `cur` pointers and did the operations to search, insert and remove information
- Now we can have 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.
  - different from linear linked lists, stacks, queues: create specialized Node class that knows about its data, parent, and children

Binary Search Tree: Node Class (1/3)

- "Smart" Node includes the following methods:
  - //pass in entire data item, containing key, not just key, so compare() will work
  - public Node<Type> search(Type dataToFind);
  - public Node<Type> insert(Type newNode);
  
  /*Remove deletes Node pointing to dataToRemove, which contains key; removing Node also will remove the data instance unless there's another reference to it*/
  - public Node<Type> remove(Type dataToRemove);
  
  - public Node<Type> swapData();

- Plus setters and getters of instance variables, defined in the next slides ...

Binary Search Tree: 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
  - _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

Binary Search Tree: Node Class (3/3)

```java
public class Node<Type extends Comparable<Type>> {
    private Type _data;
    private Type _parent;
    private NodeType _left;
    private NodeType _right;
    public Node(Type data, NodeType parent) {
        _data = data;
        _parent = parent;
        //children start as null - their values are set when children nodes are created
        _left = null;
        _right = null;
    }
    //Will define other methods in next slides.
}
```
Aside: What about leaf Nodes?

- A leaf node has no descendants
  - We want to use a similar design to MyLinkedList, using null pointer
- Every Node starts as a leaf, as _left and _right start as null:

  ```java
  public Node(Type data, Node<TYPE> parent) {
    _data = data;
    _parent = parent;
    _left = null;
    _right = null;
  }
  ```

- We can simply reassign values of _left and _right using mutator methods when we insert new Nodes into the tree

---

Smart Node Approach

- BinarySearchTree is dumb so it delegates to root

  ```java
  public Node<TYPE> search(dataToFind) {
    return _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 list, but trees are more complicated, and recursion is easier

Recursion – Turtles all the Way Down

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

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

  - Stephen Hawking, A Brief History of Time (1988)

---

Searching a Binary Tree Recursively 1/2

```java
public Node<TYPE> search(TYPE dataToFind) {
    //if _data is the thing we're searching for
    if(_data.compareTo(dataToFind) == 0) {
        return this;
    } //if _data > dataToFind, can only be in left tree
    else if(_data.compareTo(dataToFind) > 0) {
        if(_left == null) {
            return _left.search(dataToFind);
        } //if _data < dataToFind, can only be in right tree
        else {
            if(_right == null) {
                return _right.search(dataToFind);
            } //only gets here if dataToFind isn't in tree, otherwise would've returned sooner
            return null;
        }
    }
} //search method for entire BinarySearchTree:
public Node<TYPE> search(TYPE dataToFind) {
    return _root.search(dataToFind);
}
```
Searching a Binary Tree Recursively 2/2

- Search path: start with root M and choose path to I (for reasonably balanced tree, M will be more or less “in the middle”)
  - The height of the tree with n nodes is $\log_2 n$
  - At most, we visit each level of the tree once
  - So, searching is $O(\log N)$

Searching Simulation

- 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. 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.”

Insertion into a BST(1/2)

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

Insertion into a BST(2/2)

- Example: Insert 115

Before:

100
/   \
150   200
/     \
125  175
/     \
85  140
/     \30
75 20

After:

200
/   \
100   150
/     \
125  175
/     \30
85  140
/     \30
75 20
/     \30
30
20
Insertion Code in BST

- Again, we use a "Smart Node" approach and delegate
  
  ```java
  public void insert(Type newData) {
    // if tree is empty, make first node. No traversal necessary!
    if (_root == null) {
      _root = new Node(newData, null);
    } else {
      _root.insert(newData); // delegate
    }
  }
  ```

Insertion Code in Node

```java
public Node<Type> insert(Type newData) {
  // if newData should be in left subtree
  if (_data.compareTo(newData) > 0) {
    if (_left == null) {
      // left child is null - we've found the place to insert!
      _left = new Node(newData, this);
      return _left;
    } else { // keep traversing down tree
      return _left.insert(newData);
    }
  } else { // newData should be in right subtree
    if (_right == null) {
      // right child is null - we've found the place to insert!
      _right = new Node(newData, this);
      return _right;
    } else { // keep traversing down tree
      return _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:

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

```
16
123
224
```

Insertion Simulation (2/4)

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

```
if (_data.compareTo(newData) > 0) {
  // code for inserting left child elided
  if (_right == null) {
    // code for inserting with null
    // right child elided
    return _right.insert(newData);
  } else {
    return _right.insert(newData);
  }
}
```

```
16
123
224
```
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 (_data.compareTo(newData) > 0) {
    if (_left == null) {
        _left = new Node(newData, this);
        return _left;
    } else {
        // Code for continuing traversal elided
    }
```
Remove: no child case

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

Remove: one-child case

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

Remove: two-children case (1/3)

- Harder case: node to remove has two internal children
  - brute force: just flag node for removal, and rewrite tree at a later time – bad idea, because now every operation requires checking that flag. Instead, do the work right away
  - this is tricky, because not immediately obvious which child should replace its parent
  - non-obvious solution: first swap the data in Node to be removed with data in a Node that doesn’t have two children, then remove Node using one of simpler remove cases

Remove: two-children case (2/3)

- Use an auxiliary method `swapData`
  - swaps data in node to be removed with the data in the right-most node in its left subtree
  - this child has a key value less than all Nodes in the to-be-removed Node’s right subtree, and greater than all other nodes in its left subtree
  - since it is a right-most Node, it has at most one child
  - this swap is temporary – we then remove the node in the right-most position using simpler remove
Remove: two-children case (3/3)

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

Remove: BST Code

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

```java
// in BinarySearchTree:
public void remove(Type dataToRemove) {
    Node<Type> toRemove = _root.search(dataToRemove);
    toRemove.remove();
}
```

Remove: Node Code (1/3)

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

```java
public Node<Type> remove() {
    // code for case 1 elided
    // in a one-child case, we replace the parent's reference to Node with the Node's child.
    if (_left != null && _right == null) {
        // Case 2.1 - Node only has left child
        if (_parent.getLeft() == this) {
            _parent.setLeft(_left);
        } else {
            _parent.setRight(_left);
        }
    } else if (_left == null && _right != null) {
        // Case 2.2 - Node only has right child
        if (_parent.getLeft() == this) {
            _parent.setLeft(_right);
        } else {
            _parent.setRight(_right);
        }
    } else { // Case 3 on next slide...
    }
}
```

Remove: Node Code (2/3)

```java
public Node<Type> remove() {
    // code for case 1 elided
    // in a one-child case, we replace the parent's reference to Node with the Node's child.
    if (_left != null && _right == null) {
        // Case 2.1 - Node only has left child
        if (_parent.getLeft() == this) {
            _parent.setLeft(_left);
        } else {
            _parent.setRight(_left);
        }
    } else if (_left == null && _right != null) {
        // Case 2.2 - Node only has right child
        if (_parent.getLeft() == this) {
            _parent.setLeft(_right);
        } else {
            _parent.setRight(_right);
        }
    } else { // Case 3 on next slide...
    }
}
```
Remove: Node Code (3/3)
- Successor is guaranteed to have at most one child, so we remove with simpler remove case

```java
public Node<T> remove() {
    // code for case 1 (no children) elided
    // code for case 2 (one child) elided
    else {  // case 3 - both children
        Node<T> toSwap = this.swapData(); // swap data with successor
        tidewp.remove(); // now remove tidewp, which holds original Node's data
        return tidewp; // return tidewp, since tidewp was data we removed
    }
    return this;  // return this if we didn't do any swapping since Node is removed
}
// swapData defined on next slide
```

Remove: swapData code
- We find the rightmost Node in left subtree, but we can also find the leftmost Node in right subtree

```java
public Node<T> swapData() {
    Node<T> curr = _left;  // first get left child
    while(curr.getRight() != null) {  // go right as far as possible
        curr = curr.getRight();
    }
    // swap data of this Node and successor
    Type template = _data;
    _data = curr.getData();
    curr.setData(template);
    return curr;
}
```

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!

```java
public void inOrder() {
    // Check for null children elided
    _left.inOrder();
    this.doSomething();
    _right.inOrder();
}
```
Preorder Traversal of BST

- **Preorder traversal**
  - "Preorder" because self is visited before ("pre") visiting children
  - Again, use recursion!

  ```java
  public void preorder() {
    // Check for null children elided
    this.doSomething();
    _left.preorder();
    _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
    _left.postOrder();
    _right.postOrder();
    this.doSomething();
  }
  ```

Prefix, Infix, Postfix Notation for Arithmetic Expressions

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

Using Prefix, Infix, Postfix Notation

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

To learn more about the exciting world of trees, take CS16 (CSCI0160): Introduction to Algorithms and Data Structures!
Dijkstra's infix-to-postfix Algorithm (1/2)

2 stack algorithm for single-pass Infix to Postfix conversion, using operator precedence

\[(a + (b \cdot (c ^ d))) \Rightarrow a \ b \ c \ d \ c^d \ a + \]

<table>
<thead>
<tr>
<th>Incoming Operator</th>
<th>Top of Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>( + )</td>
<td>A A A A C</td>
</tr>
<tr>
<td>( \cdot )</td>
<td>A B B B C</td>
</tr>
<tr>
<td>( ^ )</td>
<td>A A A B C</td>
</tr>
<tr>
<td></td>
<td>A A B C</td>
</tr>
<tr>
<td></td>
<td>A B C</td>
</tr>
<tr>
<td></td>
<td>A A A E</td>
</tr>
</tbody>
</table>

A) Push operands onto operand stack; push operators in precedence order onto the operator stack.

B) When precedence order would be disturbed, pop operator stack until order is restored, evaluating each pair of operands popped from the operand stack and pushing the result back onto the operand stack.

Note that equal precedence displaces. At the end of the statement (marked by ; or CR) all operators are popped.

C) "(" starts a new substack; ")" pops until its matching "(".

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

\[(a + (b \cdot (c ^ d))) \Rightarrow a \ b \ c \ d \ c^d \ a + \]

<table>
<thead>
<tr>
<th>Incoming Operator</th>
<th>Top of Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>( + )</td>
<td>A A A A C</td>
</tr>
<tr>
<td>( \cdot )</td>
<td>A B B B C</td>
</tr>
<tr>
<td>( ^ )</td>
<td>A A A B C</td>
</tr>
<tr>
<td></td>
<td>A A B C</td>
</tr>
<tr>
<td></td>
<td>A A E</td>
</tr>
</tbody>
</table>

Challenge Questions Solutions

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

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

Announcements

- Hours have been very empty since DJ was due.
  - This is a great time to come and ask conceptual questions.

- Tetris Help Session on Thursday from 6pm-8pm in Metcalf Research Building AUD

- Homework 3 is due Sunday at 2pm.
  - Remember: You may not be able to fully complete question 3 before Thursday’s lecture.