Motivation: Maintaining a Sorted Collection of Data

- A data dictionary is a sorted collection of data with the following key operations:
  - search for an item (and possibly delete it)
  - insert a new item
- If we use a list to implement a data dictionary, efficiency = $O(n)$.

<table>
<thead>
<tr>
<th>data structure</th>
<th>searching for an item</th>
<th>inserting an item</th>
</tr>
</thead>
<tbody>
<tr>
<td>a list implemented using an array</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a list implemented using a linked list</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- In the next few lectures, we'll look at data structures (trees and hash tables) that can be used for a more efficient data dictionary.
- We'll also look at other applications of trees.
What Is a Tree?

- A tree consists of:
  - a set of nodes
  - a set of edges, each of which connects a pair of nodes
- Each node may have one or more data items.
  - each data item consists of one or more fields
  - key field = the field used when searching for a data item
  - multiple data items with the same key are referred to as duplicates
- The node at the “top” of the tree is called the root of the tree.

Relationships Between Nodes

- If a node N is connected to other nodes that are directly below it in the tree, N is referred to as their parent and they are referred to as its children.
  - example: node 5 is the parent of nodes 10, 11, and 12
- Each node is the child of at most one parent.
- Other family-related terms are also used:
  - nodes with the same parent are siblings
  - a node’s ancestors are its parent, its parent’s parent, etc.
    - example: node 9’s ancestors are 3 and 1
  - a node’s descendants are its children, their children, etc.
    - example: node 1’s descendants are all of the other nodes
Types of Nodes

- A **leaf node** is a node without children.
- An **interior node** is a node with one or more children.

A Tree is a Recursive Data Structure

- Each node in the tree is the root of a smaller tree!
  - refer to such trees as **subtrees** to distinguish them from the tree as a whole
  - example: node 2 is the root of the subtree circled above
  - example: node 6 is the root of a subtree with only one node
- We’ll see that tree algorithms often lend themselves to recursive implementations.
Path, Depth, Level, and Height

- There is exactly one path (one sequence of edges) connecting each node to the root.
- depth of a node = # of edges on the path from it to the root
- Nodes with the same depth form a level of the tree.
- The height of a tree is the maximum depth of its nodes.
  - example: the tree above has a height of 2

Binary Trees

- In a binary tree, nodes have at most two children.
- Recursive definition: a binary tree is either:
  1) empty, or
  2) a node (the root of the tree) that has
     • one or more data fields
     • a left child, which is itself the root of a binary tree
     • a right child, which is itself the root of a binary tree
- Example:

  - How are the edges of the tree represented?
Representing a Binary Tree Using Linked Nodes

```java
public class LinkedTree {
    private class Node {
        private int key;
        private LLList data;  // list of data for that key
        private Node left; // reference to left child
        private Node right; // reference to right child
    }
    private Node root;
}
```

Traversing a Binary Tree

- Traversing a tree involves visiting all of the nodes in the tree.
  - visiting a node = processing its data in some way
    - example: print the key
- We will look at four types of traversals. Each of them visits the nodes in a different order.
- To understand traversals, it helps to remember the recursive definition of a binary tree, in which every node is the root of a subtree.
Preorder Traversal

- preorder traversal of the tree whose root is N:
  1) visit the root, N
  2) recursively perform a preorder traversal of N's left subtree
  3) recursively perform a preorder traversal of N's right subtree

![Tree Diagram]

- Preorder traversal of the tree above:
  7 5 2 4 6 9 8

Implementing Preorder Traversal

```java
public class LinkedTree {
    private Node root;

    public void preorderPrint() {
        if (root != null)
            preorderPrintTree(root);
    }

    private static void preorderPrintTree(Node root) {
        System.out.print(root.key + " ");
        if (root.left != null)
            preorderPrintTree(root.left);
        if (root.right != null)
            preorderPrintTree(root.right);
    }
}
```

- `preorderPrintTree()` is a static, recursive method that takes as a parameter the root of the tree/subtree that you want to print.
- `preorderPrint()` is a non-static method that makes the initial call. It passes in the root of the entire tree as the parameter.
Tracing Preorder Traversal

```java
void preorderPrintTree(Node root) {
    System.out.print(root.key + " ");
    if (root.left != null)
        preorderPrintTree(root.left);
    if (root.right != null)
        preorderPrintTree(root.right);
}
```

Postorder Traversal

- postorder traversal of the tree whose root is N:
  1) recursively perform a postorder traversal of N's left subtree
  2) recursively perform a postorder traversal of N's right subtree
  3) visit the root, N

- Postorder traversal of the tree above:
  4  2  6  5  8  9  7
Implementing Postorder Traversal

```java
public class LinkedTree {
    private Node root;

    public void postorderPrint() {
        if (root != null) {
            postorderPrintTree(root);
        }
    }

    private static void postorderPrintTree(Node root) {
        if (root.left != null) {
            postorderPrintTree(root.left);
        }
        if (root.right != null) {
            postorderPrintTree(root.right);
        }
        System.out.print(root.key + " ");
    }
}
```

- Note that the root is printed after the two recursive calls.

Tracing Postorder Traversal

```java
void postorderPrintTree(Node root) {
    if (root.left != null) {
        postorderPrintTree(root.left);
    }
    if (root.right != null) {
        postorderPrintTree(root.right);
    }
    System.out.print(root.key + " ");
}
```
Inorder Traversal

- inorder traversal of the tree whose root is N:
  1) recursively perform an inorder traversal of N's left subtree
  2) visit the root, N
  3) recursively perform an inorder traversal of N's right subtree

- Inorder traversal of the tree above:
  2 4 5 6 7 8 9

Implementing Inorder Traversal

```java
public class LinkedTree {
   private Node root;
   public void inorderPrint() {
      if (root != null)
         inorderPrintTree(root);
   }
   private static void inorderPrintTree(Node root) {
      if (root.left != null)
         inorderPrintTree(root.left);
      System.out.print(root.key + " ");
      if (root.right != null)
         inorderPrintTree(root.right);
   }
}
```

- Note that the root is printed between the two recursive calls.
**Tracing Inorder Traversal**

```java
void inorderPrintTree(Node root) {
    if (root.left != null)
        inorderPrintTree(root.left);
    System.out.print(root.key + "");
    if (root.right != null)
        inorderPrintTree(root.right);
}
```

**Level-Order Traversal**

- Visit the nodes one level at a time, from top to bottom and left to right.

- Level-order traversal of the tree above: **7 5 9 2 6 8 4**
- How could we implement this type of traversal?
**Tree-Traversals Summary**

- Preorder: root, left subtree, right subtree
- Postorder: left subtree, right subtree, root
- Inorder: left subtree, root, right subtree
- Level-order: top to bottom, left to right

- Perform each type of traversal on the tree below:

```
  9
 / \
15  7
 /   / \
23  8  10  5
 /   /   / \
12  6  35 26
   /   /       \
   2       10
```

**Using a Binary Tree for an Algebraic Expression**

- We'll restrict ourselves to fully parenthesized expressions and to the following binary operators: +, -, *, /

- Example expression: 
  \[
  ((a + (b * c)) - (d / e))
  \]

- Tree representation:

```
- 
  + 
    a * 
      b 
      c
    d 
    e
```

- Leaf nodes are variables or constants; interior nodes are operators.
- Because the operators are binary, either a node has two children or it has none.
Traversing an Algebraic-Expression Tree

- Inorder gives conventional algebraic notation.
  - print '(' before the recursive call on the left subtree
  - print ')' after the recursive call on the right subtree
  - for tree at right: \((a + (b + c)) - (d / e)\)

- Preorder gives functional notation.
  - print '('s and ')'s as for inorder, and commas after the recursive call on the left subtree
  - for tree above: \(\text{subtr} (\text{add}(a, \text{mul}(b, c)), \text{div}(d, e))\)

- Postorder gives the order in which the computation must be carried out on a stack/RPN calculator.
  - for tree above: \(\text{push } a, \text{push } b, \text{push } c, \text{multiply, add, ...}\)

Fixed-Length Character Encodings

- A character encoding maps each character to a number.
- Computers usually use fixed-length character encodings.
  - ASCII (American Standard Code for Information Interchange) uses 8 bits per character.
  - example: “bat” is stored in a text file as the following sequence of bits: 01100010 01100001 01110100
  - Unicode uses 16 bits per character to accommodate foreign-language characters. (ASCII codes are a subset.)
  - Fixed-length encodings are simple, because
    - all character encodings have the same length
    - a given character always has the same encoding
Variable-Length Character Encodings

- Problem: fixed-length encodings waste space.
- Solution: use a variable-length encoding.
  - use encodings of different lengths for different characters
  - assign shorter encodings to frequently occurring characters
- Example:

<table>
<thead>
<tr>
<th>Character</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>01</td>
</tr>
<tr>
<td>o</td>
<td>100</td>
</tr>
<tr>
<td>s</td>
<td>111</td>
</tr>
<tr>
<td>t</td>
<td>00</td>
</tr>
</tbody>
</table>

“test” would be encoded as $000111100 \rightarrow 00111100$

- Challenge: when decoding/decompressing an encoded document, how do we determine the boundaries between characters?
  - example: for the above encoding, how do we know whether the next character is 2 bits or 3 bits?
- One requirement: no character’s encoding can be the prefix of another character’s encoding (e.g., couldn’t have $00$ and $001$).

---

Huffman Encoding

- Huffman encoding is a type of variable-length encoding that is based on the actual character frequencies in a given document.
- Huffman encoding uses a binary tree:
  - to determine the encoding of each character
  - to decode an encoded file – i.e., to decompress a compressed file, putting it back into ASCII
- Example of a Huffman tree (for a text with only six chars):

Leaf nodes are characters.
Left branches are labeled with a 0, and right branches are labeled with a 1.
If you follow a path from root to leaf, you get the encoding of the character in the leaf.
example: $101 = 'i'$
Building a Huffman Tree

1) Begin by reading through the text to determine the frequencies.

2) Create a list of nodes that contain (character, frequency) pairs for each character that appears in the text.

3) Remove and “merge” the nodes with the two lowest frequencies, forming a new node that is their parent.
   - left child = lowest frequency node
   - right child = the other node
   - frequency of parent = sum of the frequencies of its children
   - in this case, 11 + 23 = 34

4) Add the parent to the list of nodes (maintaining sorted order):

5) Repeat steps 3 and 4 until there is only a single node in the list, which will be the root of the Huffman tree.
Completing the Huffman Tree Example I

- Merge the two remaining nodes with the lowest frequencies:

Completing the Huffman Tree Example II

- Merge the next two nodes:
Completing the Huffman Tree Example II

- Merge again:

Completing the Huffman Tree Example IV

- The next merge creates the final tree:

- Characters that appear more frequently end up higher in the tree, and thus their encodings are shorter.
The Shape of the Huffman Tree

• The tree on the last slide is fairly symmetric.
• This won’t always be the case!
  • depends on the frequencies of the characters in the document being compressed
• For example, changing the frequency of ‘o’ from 11 to 21 would produce the tree shown below:

![Huffman Tree Diagram]

• This is the tree that we’ll use in the remaining slides.

Using Huffman Encoding to Compress a File

1) Read through the input file and build its Huffman tree.
2) Write a file header for the output file.
   – include an array containing the frequencies so that the tree can be rebuilt when the file is decompressed.
3) Traverse the Huffman tree to create a table containing the encoding of each character:

<table>
<thead>
<tr>
<th>a</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>?</td>
</tr>
<tr>
<td>i</td>
<td>101</td>
</tr>
<tr>
<td>o</td>
<td>100</td>
</tr>
<tr>
<td>s</td>
<td>111</td>
</tr>
<tr>
<td>t</td>
<td>00</td>
</tr>
</tbody>
</table>

4) Read through the input file a second time, and write the Huffman code for each character to the output file.
Using Huffman Decoding to Decompress a File

1) Read the frequency table from the header and rebuild the tree.
2) Read one bit at a time and traverse the tree, starting from the root:
   - when you read a bit of 1, go to the right child
   - when you read a bit of 0, go to the left child
   - when you reach a leaf node, record the character, return to the root, and continue reading bits
   *The tree allows us to easily overcome the challenge of determining the character boundaries!*

example: 10111110000111100
101 = right,left,right = i
111 = right,right,right= s
110 = right,right,left = a
00 = left,left = t
01 = left,right = e
111 = right,right,right= s
00 = left,left = t

---

Binary Search Trees

- Search-tree property: for each node $k$:
  - all nodes in $k$’s left subtree are $< k$
  - all nodes in $k$’s right subtree are $\geq k$
- Our earlier binary-tree example is a search tree:
Searching for an Item in a Binary Search Tree

- Algorithm for searching for an item with a key \( k \):
  - if \( k == \) the root node’s key, you’re done
  - else if \( k < \) the root node’s key, search the left subtree
  - else search the right subtree

- Example: search for 7

```
public class LinkedTree {
    // Nodes have keys that are ints
    private Node root;

    public LLList search(int key) {
        Node n = searchTree(root, key);
        if (n == null)
            return null; // no such key
        else
            return n.data; // return list of values for key
    }

    private static Node searchTree(Node root, int key) {
        // write together
    }

    // Other methods and constructors
}
```
Inserting an Item in a Binary Search Tree

• We want to insert an item whose key is \( k \).

• We traverse the tree as if we were searching for \( k \).

• If we find a node with key \( k \), we add the data item to the list of items for that node.

• If we don’t find it, the last node we encounter will be the parent \( P \) of the new node.
  • if \( k < P’s \) key, make the new node \( P \)’s left child
  • else make the node \( P \)’s right child

• Special case: if the tree is empty, make the new node the root of the tree.

• The resulting tree is still a search tree.

Implementing Binary-Tree Insertion

• We’ll implement part of the \texttt{insert()} method together.

• We’ll use iteration rather than recursion.

• Our method will use two references/pointers:
  • \texttt{trav}: performs the traversal down to the point of insertion
  • \texttt{parent}: stays one behind \texttt{trav}
    • like the \texttt{trail} reference that we sometimes use when traversing a linked list
Implementing Binary-Tree Insertion

```java
public void insert(int key, Object data) {
    Node parent = null;
    Node trav = root;
    while (trav != null) {
        if (trav.key == key) {
            trav.data.addItem(data, 0);
            return;
        }
    }

    Node newNode = new Node(key, data);
    if (root == null)    // the tree was empty
        root = newNode;
    else if (key < parent.key)    // Note that in the while loop, we don't update the parent.
        parent.left = newNode;
    else
        parent.right = newNode;
}
```

Deleting Items from a Binary Search Tree

- Three cases for deleting a node x
- **Case 1:** x has no children.
  Remove x from the tree by setting its parent’s reference to null.

  ex: delete 4

- **Case 2:** x has one child.
  Take the parent’s reference to x and make it refer to x’s child.

  ex: delete 12
Deleting Items from a Binary Search Tree (cont.)

- **Case 3:** \( x \) has two children
  - we can't just delete \( x \). why?

  - instead, we replace \( x \) with a node from elsewhere in the tree
  - to maintain the search-tree property, we must choose the replacement carefully
    - example: what nodes could replace 26 below?

![Diagram of a binary search tree with nodes 26, 12, 32, 4, 18, 38, 7, 35.]

Deleting Items from a Binary Search Tree (cont.)

- **Case 3:** \( x \) has two children (continued):
  - replace \( x \) with the smallest node in \( x \)'s right subtree—call it \( y \)
    - \( y \) will either be a leaf node or will have one right child. why?
  - After copying \( y \)'s item into \( x \), we delete \( y \) using case 1 or 2.

ex:

delete 26

![Diagram showing the process of replacing 26 with the smallest node from its right subtree and deleting the smallest node.]

Implementing Binary-Tree Deletion

```java
public LLList delete(int key) {
    // Find the node and its parent.
    Node parent = null;
    Node trav = root;
    while (trav != null && trav.key != key) {
        parent = trav;
        if (key < trav.key)
            trav = trav.left;
        else
            trav = trav.right;
    }

    // Delete the node (if any) and return the removed items.
    if (trav == null)    // no such key
        return null;
    else {
        LLList removedData = trav.data;
        deleteNode(trav, parent);
        return removedData;
    }
}
```

- This method uses a helper method to delete the node.

Implementing Case 3

```java
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        // Find a replacement - and
        // the replacement's parent.
        Node replaceParent = toDelete;
        // Get the smallest item
        // in the right subtree.
        Node replace = toDelete.right;
        // What should go here?
        toDelete.key = replace.key;
        toDelete.data = replace.data;
        // Recursively delete the replacement
        // item's old node. It has at most one
        // child, so we don't have to
        // worry about infinite recursion.
        deleteNode(replace, replaceParent);
    } else {
        ...
    }
```

Implementing Cases 1 and 2

```java
private void deleteNode(Node toDelete, Node parent) {
    if (toDelete.left != null && toDelete.right != null) {
        // ...
    } else {
        Node toDeleteChild;
        if (toDelete.left != null)
            toDeleteChild = toDelete.left;
        else
            toDeleteChild = toDelete.right;
        // Note: in case 1, toDeleteChild
        // will have a value of null.
        if (toDelete == root)
            root = toDeleteChild;
        else if (toDelete.key < parent.key)
            parent.left = toDeleteChild;
        else
            parent.right = toDeleteChild;
    }
}
```

Efficiency of a Binary Search Tree

- The three key operations (search, insert, and delete) all have the same time complexity.
  - insert and delete both involve a search followed by a constant number of additional operations

- Time complexity of searching a binary search tree:
  - best case: \( O(1) \)
  - worst case: \( O(h) \), where \( h \) is the height of the tree
  - average case: \( O(h) \)

- What is the height of a tree containing \( n \) items?
  - it depends! why?
Balanced Trees

- A tree is *balanced* if, for each node, the node’s subtrees have the same height or have heights that differ by 1.

- For a balanced tree with $n$ nodes:
  - height = $O(\log_2 n)$.
  - gives a worst-case time complexity that is logarithmic ($O(\log_2 n)$)
    - the best worst-case time complexity for a binary tree

What If the Tree Isn't Balanced?

- Extreme case: the tree is equivalent to a linked list
  - height = $n \cdot 1$
  - worst-case time complexity = $O(n)$

- We’ll look next at search-tree variants that take special measures to ensure balance.