Binary Trees and Huffman Encoding

Computer Science S-111
Harvard University
David G. Sullivan, Ph.D.

Motivation: Implementing a Dictionary

- A data dictionary is a collection of data with two main operations:
  - search for an item (and possibly delete it)
  - insert a new item
- If we use a sorted list to implement it, 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>$O(\log n)$ using binary search</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 how we can use a tree for a data dictionary, and we'll try to get better efficiency.
- 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.
  - distinguish between them using the direction left or right
- Example:

  ![Binary Tree Diagram]

  - Recursive definition: a binary tree is either:
    1) empty, or
    2) a node (the root of the tree) that has:
      - one or more pieces of data (the key, and possibly others)
      - a left subtree, which is itself a binary tree
      - a right subtree, which is itself a binary tree
Which of the following is/are not true?

A. This tree has a height of 4.
B. There are 3 leaf nodes.
C. The 38 node is the right child of the 32 node.
D. The 12 node has 3 children.
E. more than one of the above are not true (which ones?)

Representing a Binary Tree Using Linked Nodes

```java
public class LinkedTree {
    private class Node {
        private int key;      // limit ourselves to int keys
        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;...
}
```
Representing a Binary Tree Using Linked Nodes

public class LinkedTree {
    private class Node {
        private int key;
        private LLList data;
        private Node left;
        private Node right;
    }
    private Node root;
}

26
 12
 32
 4
 18
 38

7

root
LinkedTree object

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 that every node is the root of a subtree.

12 is the root of 26’s left subtree
4 is the root of 12’s left subtree
32 is the root of 26’s right subtree
1: 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

- preorder because a node is visited \textit{before} its subtrees
- The root of the tree as a whole is visited first.

Implementing Preorder Traversal

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

    public void preorderPrint() {
        if (root != null) {
            preorderPrintTree(root);
        }
        System.out.println();
    }

    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);
        }
    }
}
```

- \texttt{preorderPrintTree()} is a static, recursive method that takes the root of the tree/subtree that you want to print.
- \texttt{preorderPrint()} is a non-static "wrapper" method that makes the initial call. It passes in the root of the entire tree.

Not always the same as the root of the entire tree.
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);
    }
}
```

Using Recursion for Traversals

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

- Using recursion allows us to easily go back up the tree.
- Using a loop would be harder. Why?
2: 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 because a node is visited after its subtrees
- The root of the tree as a whole is visited last.

Implementing Postorder Traversal

```java
public class LinkedTree {
    private Node root;
    public void postorderPrint() {
        if (root != null) {
            postorderPrintTree(root);
        }
        System.out.println();
    }
    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.
3: 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

- The root of the tree as a whole is visited between its subtrees.
- We'll see later why this is called inorder traversal!
Implementing Inorder Traversal

public class LinkedTree {
    private Node root;
    public void inorderPrint() {
        if (root != null) {
            inorderPrintTree(root);
        }
        System.out.println();
    }
    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

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 9 5 8 6 2 4
- We can implement this type of traversal using a queue.

Tree-Traversal 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:
Tree Traversal Puzzle

- preorder traversal: A M P K L D H T
- inorder traversal: P M L K A H T D
- Draw the tree!
- What's one fact that we can easily determine from one of the traversals?

Using a Binary Tree for an Algebraic Expression

- We'll restrict ourselves to fully parenthesized expressions and to the following binary operators: $+, -, \ast, /$
- Example expression: $((a + (b \ast c)) - (d \div e))$
- Tree representation:

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

- 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: 
    \[(a + (b \times c)) - (d / e)\]

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

• Postorder gives the order in which the computation must be carried out on a stack/RPN calculator.
  • for tree above: push a, push b, push c, multiply, add,…

Fixed-Length Character Encodings

• A character encoding maps each character to a number.

• Computers usually use fixed-length character encodings.
  • ASCII - 8 bits per character
    
    | char | Dec | binary     |
    |------|-----|------------|
    | 'a'  | 97  | 01100001   |
    | 'b'  | 98  | 01100010   |
    | 'c'  | 99  | 01100011   |
    | ...  | ... | ...        |
    | 't'  | 116 | 01110100   |

  example: "bat" is stored in a text file as the following sequence of bits:
  01100010 01100001 01110100

• Unicode - 16 bits per character
  (allows for foreign-language characters; ASCII is a subset)

• Fixed-length encodings are simple, because:
  • all encodings have the same length
  • a given character always has the same encoding
A Problem with Fixed-Length Encodings

- They tend to waste space.

- Example: an English newspaper article with only:
  - upper and lower-case letters (52 characters)
  - spaces and newlines (2 characters)
  - common punctuation (approx. 10 characters)
  - total of 64 unique characters \(\Rightarrow\) only need ___ bits

- We could gain even more space if we:
  - gave the most common letters shorter encodings (3 or 4 bits)
  - gave less frequent letters longer encodings (> 6 bits)

Variable-Length Character Encodings

- Variable-length encodings:
  - use encodings of different lengths for different characters
  - assign shorter encodings to frequently occurring characters

- Example: if we had only four characters

<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
\[
00\ 01\ 111\ 00 \Rightarrow 000111100
\]

- 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

- 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 containing (character, frequency) pairs for each character in the text – sorted by frequency.
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

means null
Building a Huffman Tree (cont.)

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

```
'a' 25
's' 26
't' 27
'e' 40
```

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 1

- Merge the two remaining nodes with the lowest frequencies:

```
'a' 25
's' 26
't' 27
'e' 40
'o' 11
'i' 23
```
Completing the Huffman Tree Example II

- Merge the next two nodes:

```
  'e'  40
  34    51
  'o'  11  23
  25  26
  't'  27  34
```

- Merge again:

```
  'o'  11  23
  34    51
  't'  27  34
  25  26
  't'  27  34
  25  26
```
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:
- This is the tree that we'll use in the remaining slides.
**Huffman Encoding: Compressing 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:

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

---

**Huffman Decoding: Decompressing 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

First character = i
What are the next three characters?

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: 101111110000111100
first character = i (101)
Huffman Decoding: Decompressing 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