Week 10: Binary Trees

CS2123 Data Structures
Binary Trees

Dr. Weining Zhang
Department of Computer Science
University of Texas at San Antonio
August 28, 2009

Outline

1. Basic Concepts
   - Binary Trees
   - Tree Traversals
2. Implementation
   - Linked Binary Tree
   - Array Binary Tree
3. Huffman Encoding
   - Huffman Encoding Tree
   - Decoding
   - Encoding
   - Implementation

Tasks of the Week

- Discuss basic tree terminology and representations
- Case Study: Develop a linked implementation of BinaryTreeADT.
- Case Study: Huffman Coding
Basic Concepts

Terminology of Tree

- Node (or vertex), edge (downward link)
- Child and parent nodes
- Root: the node that has no parent
- Leaf (or external) node: a node that has no child
- Internal node: a node that has at least one child
- Fan-out: number of child nodes
- Level: root is at level 0
- Height: number of levels

Binary Trees

Definition
A tree in which each internal node may have at most two child nodes, the left and the right child

Example

Binary Trees

Topology of Binary Tree

Definition
A binary tree is complete if every internal node has two child nodes; sparse otherwise.

Definition
A binary tree is balanced if all leaf nodes are at the same level.

Binary Tree Traversals

Definition
A traversal of a tree is to visit every node in the tree exactly once

Traversal algorithms:
- Preorder: visit a node before visiting its left and right child
- Inorder: visit a node after visiting its left child before visiting its right child
- Postorder: visit a node after visiting its left and right child
- Level: visit nodes topdown, level by level, and in each level from left to right
Traversals

Example

- Inorder: DBAECGF
- Preorder: ABDCEF
- Postorder: DBEGFC
- Level: ABCDEFG

Linked Binary Tree

Array Binary Tree

- Store tree nodes in an array of element type
- The parent-child relationships among nodes are determined by the indexes of the nodes.
  - Root is at node 0
  - For node at position \(i\), the left child is position \(2i\), and the right child is at position \(2i + 1\)
A method to compress data by encoding bytes each with less than 8 bits, according to frequencies of the bytes in the data.

Represent encoding/decoding mapping as a Huffman coding tree, a binary tree constructed according to frequencies of bytes.

### Example

**Data:**
```
bcbbbbbaacaabbcede
```

<table>
<thead>
<tr>
<th>letter</th>
<th>freq</th>
<th>ASCII</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>5</td>
<td>01100001</td>
<td>11</td>
</tr>
<tr>
<td>b</td>
<td>9</td>
<td>01100010</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>01100011</td>
<td>101</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td>01100100</td>
<td>1001</td>
</tr>
<tr>
<td>e</td>
<td>1</td>
<td>01100101</td>
<td>1000</td>
</tr>
</tbody>
</table>

Construct Huffman Tree

- scan data to create leaf nodes and set their weights;
- put all leaf nodes in a priority queue on ascending weights;
- while priority queue has more than one node do
  - delete the first two nodes from the priority queue;
  - create a new node with these deleted nodes as children;
  - set weight of new node to the sum of that of the deleted nodes;
  - add new node to the priority queue;
- take the last node in the priority queue as the root;
Decoding Algorithm

- Set the current node to the root of the tree;
- While the input is not empty, read the next bit in the input
  - If the input is a 0, go to the left child of the current node;
  - If the bit is a 1, go to the right child of the current node;
  - If the current node is a leaf, output the byte in the node and reset the current node to the root;

Encoding Algorithm

- For each character in the input file,
  - Find the leaf node corresponding to that character
  - Traverse the tree to its root, prepending the bits for that character.
    - If current node is a left child, prepend a 0.
    - If current node is a right child, prepend a 1.
    - If current node has no parent, return the string for this character. Otherwise, set the parent the current node.

Decoding Example

- Input bit string: 0001101011000

![Huffman Encoding Tree]

- Output: bbabce

Encoding Example

- Input bit string: bbabce

![Huffman Encoding Tree]

- Output: 0001101011000
### Array As Encoding Table

<table>
<thead>
<tr>
<th>value</th>
<th>index</th>
<th>weight</th>
<th>parent</th>
<th>isRightChild</th>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td>98 (b)</td>
<td>0</td>
<td>9</td>
<td>8</td>
<td>false</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>99 (c)</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>true</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>97 (a)</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>true</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>100 (d)</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>true</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>101 (e)</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>false</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>128 (T1)</td>
<td>5</td>
<td>2</td>
<td>6</td>
<td>false</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>129 (T2)</td>
<td>6</td>
<td>5</td>
<td>7</td>
<td>false</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>130 (T3)</td>
<td>6</td>
<td>10</td>
<td>8</td>
<td>true</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>131 (T4)</td>
<td>8</td>
<td>19</td>
<td>-1</td>
<td>false</td>
<td>0</td>
<td>7</td>
</tr>
</tbody>
</table>

### Inverted Index

<table>
<thead>
<tr>
<th>Character</th>
<th>Index to Huffman Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>97 (a)</td>
<td>2</td>
</tr>
<tr>
<td>98 (b)</td>
<td>0</td>
</tr>
<tr>
<td>99 (c)</td>
<td>1</td>
</tr>
<tr>
<td>100 (d)</td>
<td>3</td>
</tr>
<tr>
<td>101 (e)</td>
<td>4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>127</td>
<td>-1</td>
</tr>
</tbody>
</table>