Administrative Details

- Lab 8 today!
  - No partners this week
  - Review before lab; come to lab with design doc
    - Read over the supplied resources!
Last Time

• Trees with more than 2 children
  • Representations
  • Application: Lab 8: Hex-a-pawn!

• Binary Trees
  • Traversals
    • As methods taking a BinaryTree parameter
    • With Iterators
Today

• Wrap up Binary Tree Iterators
• Breadth-First and Depth-First Search
• Array Representations of (Binary) Trees
• Application: Huffman Encoding
Recall from last class:

- **In-order**: “left, node, right”
- **Pre-order**: “node, left, right”
- **Post-order**: “left, right, node”
- **Level-order**: visit all nodes at depth $i$ before depth $i + 1$
public BTPostorderIterator(BinaryTree<E> root) {
    todo = new StackList<BinaryTree<E>>();
    this.root = root;
    reset();
}

public void reset() {
    todo.clear();
    BinaryTree<E> current = root;
    while (!current.isEmpty()) {
        todo.push(current);
        if (!current.left().isEmpty())
            current = current.left();
        else
            current = current.right();
    }  // Top of stack is now left-most unvisited leaf
}
Post-Order Iterator

```java
public E next() {
    BinaryTree<E> current = todo.pop();
    E result = current.value();
    if (!todo.isEmpty()) {
        BinaryTree<E> parent = todo.get();
        if (current == parent.left()) { // current was left child
            current = parent.right();
            while (!current.isEmpty()) {
                todo.push(current);
                if (!current.left().isEmpty())
                    current = current.left();
                else current = current.right();
            }
        }
    }
    return result;
}
```
Traversals & Searching

- We can use traversals for searching trees.
- How might we search a tree for a value?
  - Breadth-First: Explore nodes near the root before nodes far away (level order traversal)
    - Nearest gas station
  - Depth-First: Explore nodes deep in the tree first (post-order traversal)
    - Solution to a maze
Loose Ends – Really Big Trees!

• In some situations, the tree we need might be too big or expensive to build completely
  • Or parts of it might not be needed at all times

• Example: Game Trees
  • Chess: you wouldn’t build the entire tree, you would grow portions of it as needed (with some combination of depth/breadth first searching)

• Example: File system tree
  • Active and recently used files in memory, fetch files from disk on demand
Lexicon Lab Demo

Tasks (not in order of implementation!):

• Read a file of words using a scanner
• Generate a trie by adding each word
• Search the trie for a word (e.g., spell check)
• Search for a set of words that match a pattern (e.g., cheat at crossword puzzles)
Alternative Tree Representations

- Total # “slots” = 4n
  - Since each BinaryTree maintains a reference to left, right, parent, value
- Similar tradeoff between lists and arrays
  - Pointers consume memory,
  - But only pay for what we use

Diagram:

```
              Green
             /    |
            Blue  Violet
           /  |
         Orange  Yellow
            /  |
           Indigo  Red
```
Array-Based Binary Trees

- Encode structure of tree in array indexes
  - Put root at index 0
- Where are children of node $i$?
  - Children of node $i$ are at $2i+1$ and $2i+2$
  - Look at example
- Where is parent of node $j$?
  - Parent of node $j$ is at $(j-1)/2$
ArrayTree Tradeoffs

• Why are ArrayTrees good?
  • Save space for links
  • No need for additional memory allocated/garbage collected
  • Works well for full or complete trees
    • Complete: All levels except last are full and all gaps are at right
    • “A complete binary tree of height h is a full binary tree with 0 or more of the rightmost leaves of level h removed”

• Why bad?
  • Could waste a lot of space
  • Tree of height of n requires $2^{n+1} - 1$ array slots even if only $O(n)$ elements
Next up: Huffman Codes

- Computers encode a text as a sequence of bits

### ASCII TABLE

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
<th>Decimal</th>
<th>Hex</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>[NULL]</td>
<td>32</td>
<td>20</td>
<td>(SPACE)</td>
<td>64</td>
<td>40</td>
<td>@</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>[START OF HEADING]</td>
<td>33</td>
<td>21</td>
<td>!</td>
<td>65</td>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>[START OF TEXT]</td>
<td>34</td>
<td>22</td>
<td>&quot;</td>
<td>66</td>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>[END OF TEXT]</td>
<td>35</td>
<td>23</td>
<td>#</td>
<td>67</td>
<td>43</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>[END OF TRANSMISSION]</td>
<td>36</td>
<td>24</td>
<td>$</td>
<td>68</td>
<td>44</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>[ENQUIRY]</td>
<td>37</td>
<td>25</td>
<td>%</td>
<td>69</td>
<td>45</td>
<td>E</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>[ACKNOWLEDGE]</td>
<td>38</td>
<td>26</td>
<td>&amp;</td>
<td>70</td>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>[BEL]</td>
<td>39</td>
<td>27</td>
<td>`</td>
<td>71</td>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>[BACKSPACE]</td>
<td>40</td>
<td>28</td>
<td>(</td>
<td>72</td>
<td>48</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>[HORIZONTAL TAB]</td>
<td>41</td>
<td>29</td>
<td>)</td>
<td>73</td>
<td>49</td>
<td>I</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>[LINE FEED]</td>
<td>42</td>
<td>2A</td>
<td>*</td>
<td>74</td>
<td>4A</td>
<td>J</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>[VERTICAL TAB]</td>
<td>43</td>
<td>2B</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>K</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>[FORM FEED]</td>
<td>44</td>
<td>2C</td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>L</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>[CARRIAGE RETURN]</td>
<td>45</td>
<td>2D</td>
<td>-</td>
<td>77</td>
<td>4D</td>
<td>M</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>[SHIFT OUT]</td>
<td>46</td>
<td>2E</td>
<td>.</td>
<td>78</td>
<td>4E</td>
<td>N</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>[SHIFT IN]</td>
<td>47</td>
<td>2F</td>
<td>/</td>
<td>79</td>
<td>4F</td>
<td>O</td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>[DATA LINK ESCAPE]</td>
<td>48</td>
<td>30</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>P</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>[DEVICE CONTROL 1]</td>
<td>49</td>
<td>31</td>
<td>1</td>
<td>81</td>
<td>51</td>
<td>Q</td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>[DEVICE CONTROL 2]</td>
<td>50</td>
<td>32</td>
<td>2</td>
<td>82</td>
<td>52</td>
<td>R</td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>[DEVICE CONTROL 3]</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>83</td>
<td>53</td>
<td>S</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>[DEVICE CONTROL 4]</td>
<td>52</td>
<td>34</td>
<td>4</td>
<td>84</td>
<td>54</td>
<td>T</td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>[NEGATIVE ACKNOWLEDGE]</td>
<td>53</td>
<td>35</td>
<td>5</td>
<td>85</td>
<td>55</td>
<td>U</td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>[SYNCHRONOUS IDLE]</td>
<td>54</td>
<td>36</td>
<td>6</td>
<td>86</td>
<td>56</td>
<td>V</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>[ENG OF TRANS. BLOCK]</td>
<td>55</td>
<td>37</td>
<td>7</td>
<td>87</td>
<td>57</td>
<td>W</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>[CANCEL]</td>
<td>56</td>
<td>38</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>X</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>[END OF MEDIUM]</td>
<td>57</td>
<td>39</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>Y</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>[SUBSTITUTE]</td>
<td>58</td>
<td>3A</td>
<td>:</td>
<td>90</td>
<td>5A</td>
<td>Z</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>[ESCAPE]</td>
<td>59</td>
<td>3B</td>
<td>;</td>
<td>91</td>
<td>5B</td>
<td>{</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>[FILE SEPARATOR]</td>
<td>60</td>
<td>3C</td>
<td>&lt;</td>
<td>92</td>
<td>5C</td>
<td>}</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>[GROUP SEPARATOR]</td>
<td>61</td>
<td>3D</td>
<td>=</td>
<td>93</td>
<td>5D</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>[RECORD SEPARATOR]</td>
<td>62</td>
<td>3E</td>
<td>&gt;</td>
<td>94</td>
<td>5E</td>
<td>^</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>[UNIT SEPARATOR]</td>
<td>63</td>
<td>3F</td>
<td>?</td>
<td>95</td>
<td>5F</td>
<td>_</td>
</tr>
</tbody>
</table>
Huffman Codes

- In ASCII: 1 character = 8 bits (1 byte)
  - Allows for $2^8 = 256$ different characters
- ‘A’ = 01000001, ‘B’ = 01000010
- Space to store “AN_ANTARCTIC_PENGUIN”
  - 20 characters -> 20*8 bits = 160 bits

- Is there a better way?
  - Only 11 symbols are used (ANTRCIPEGU_)
  - “ASCII-lite” only needs 4 bits per symbol (since $2^4 > 11$)!
    - 20*4 = 80 bits instead of 160!

- Can we still do better??
Huffman Codes

• Example
  • AN_ANTARCTIC_PENGUIN
  • Compute letter frequencies

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>E</th>
<th>G</th>
<th>I</th>
<th>N</th>
<th>P</th>
<th>R</th>
<th>T</th>
<th>U</th>
<th>_</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

• **Key Idea:** Use fewer bits for most common letters

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>E</th>
<th>G</th>
<th>I</th>
<th>N</th>
<th>P</th>
<th>R</th>
<th>T</th>
<th>U</th>
<th>_</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>110</td>
<td>111</td>
<td>1011</td>
<td>1000</td>
<td>000</td>
<td>001</td>
<td>1001</td>
<td>1010</td>
<td>0101</td>
<td>0100</td>
<td>011</td>
</tr>
</tbody>
</table>

• Uses 67 bits to encode entire string
## Huffman Codes

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>E</th>
<th>G</th>
<th>I</th>
<th>N</th>
<th>P</th>
<th>R</th>
<th>T</th>
<th>U</th>
<th>_</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>110</td>
<td>111</td>
<td>1011</td>
<td>1000</td>
<td>000</td>
<td>001</td>
<td>1001</td>
<td>1010</td>
<td>0101</td>
<td>0100</td>
<td>011</td>
</tr>
</tbody>
</table>

- Uses 67 bits to encode entire string
- Can we do better?

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>E</th>
<th>G</th>
<th>I</th>
<th>N</th>
<th>P</th>
<th>R</th>
<th>T</th>
<th>U</th>
<th>_</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>010</td>
<td>1100</td>
<td>1101</td>
<td>011</td>
<td>101</td>
<td>0001</td>
<td>0000</td>
<td>001</td>
<td>1110</td>
<td>1111</td>
</tr>
</tbody>
</table>

- Uses 67 bits to encode entire string
The Encoding Tree

Left = 0; Right = 1
Features of Good Encoding

- Prefix property: No encoding is a prefix of another encoding (letters appear at leaves)
- No internal node has a single child
- Nodes with lower frequency have greater depth
- All optimal length unambiguous encodings have these features