Lecture 12: Tries  
CSE 373: Data Structures and Algorithms
Announcements

Practice Midterms Posted
https://courses.cs.washington.edu/courses/cse373/23sp/#04-28

Project 2 Due Wednesday
Exercise 3 Due Monday
Exercise 4 Releases Monday
2–3 Insertions  Insert 12 and 13 into the following 2–3 tree
2–3 Trees

**PROS**
- All operations on 2–3 Tree have a logarithmic worst case
  - Because these trees are always balanced!
- Maintaining balance doesn’t require complex rotations
- Storing multiple values per node improves runtime constants because of memory locality

**CONS**
- No height triggered balancing means 2–3 trees stay a little less balanced than AVLs on average
- Multiple node types cause implementation complexity
  - Make all nodes 2 nodes and you have more unused space
2-3 insert() code

class Node {
    int[] keys;
    Node[] children;
    int numKeys;
    boolean isLeaf;
    ...
}

public void splitChild(int i, Node y) {
    Node z = new Node(y.order, y.isLeaf);
    z.numKeys = order - 1;
    for (int j = 0; j < order - 1; j++) {
        z.keys[j] = y.keys[j + order];
    }
    if (!y.isLeaf) {
        for (int j = 0; j < order; j++) {
            z.children[j] = y.children[j + order];
        }
    }
    y.numKeys = order - 1;
    for (int j = numKeys; j >= i + 1; j--) {
        children[j + 1] = children[j];
    }
    children[i + 1] = z;
    for (int j = numKeys - 1; j >= i; j--) {
        keys[j + 1] = keys[j];
    }
    keys[i] = y.keys[order - 1];
    numKeys++;
}

public void insertNonFull(int key) {
    int i = numKeys - 1;
    if (isLeaf) {
        while (i >= 0 && keys[i] > key) {
            keys[i + 1] = keys[i];
            i--;
        }
        keys[i + 1] = key;
        numKeys++;
    } else {
        while (i >= 0 && keys[i] > key) {
            i--;
        }
        if (children[i + 1].numKeys == 2 * order - 1) {
            splitChild(i + 1, children[i + 1]);
            if (keys[i + 1] < key) {
                i++;
            }
        }
        children[i + 1].insertNonFull(key);
    }
}
2-3 Trees

**PROS**
- All operations on 2-3 Tree have a logarithmic worst case
  - Because these trees are always balanced!
- Maintaining balance doesn’t require complex rotations
- Storing multiple values per node improves runtime constants because of memory locality

**CONS**
- No height triggered balancing means 2-3 trees stay a little less balanced than AVLs on average
- Multiple node types cause implementation complexity
  - Make all nodes 2 nodes and you have more unused space
Meet Red Black Trees

1. Every node has a color either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.
5. Every leaf (i.e. NULL node) must be colored BLACK.

All paths from a node to the NULL descendants contain the same number of black nodes

Black-Height = 2
Red Black Insertions

Insertion cases:

0. Node is the root
   a. Color node black

1. Node’s uncle is red
   b. recolor

2. Node’s uncle is black (Triangle)
   c. Rotate node’s parent

3. Node’s uncle is black (line)
   d. Rotate nodes’ grandparent & recolor

Red Black Tree Insertions (Video 5min)
Node’s uncle is red

Recolor parent, uncle and grandparent

case 1: Z.uncle = red
Uncle is black (triangle)

Rotate inserted Nodes parent in opposite direction of inserted node

**case 2**: $Z.\text{uncle} = \text{black (triangle)}$
Uncle is black (line)

Rotate node’s grandparent, then recolor

**case 3**: $Z.\text{uncle} = \text{black}$(line)
AVL vs Red Black Trees

Red Black Trees:
- Easier to implement than AVL
  - Left Leaning Red Black trees are even easier to implement
- Better performance for insertion and deletion because the balancing mechanism is less strict than AVL

AVL Trees:
- Have better look up performance because of their strict balance requirements
Left Leaning Red Black Trees

A translation of 2 3 trees using nodes with only 1 value
- Red links connect two nodes that would exist within the same node in a 2-3 tree
- Black links are “standard” connections
- Red links are always on the left
- A “balanced” LLRB has the same number of black links to leaf
  - Red links don’t count towards path length

A proposed improvement to the Red Black tree from its original designer Robert Sedgewick

Valid Left Leaning Red Black Tree?

- Right red link
- Different length paths
- Sequential Red nodes
public class LLRB<Key extends Comparable<Key>, Value> {
    private static final boolean RED = true;
    private static final boolean BLACK = false;
    private Node root;
    private class Node {
        private Key key;
        private Value val;
        private Node left, right;
        private boolean color;
        Node(Key key, Value val) {
            this.key = key;
            this.val = val;
            this.color = RED;
        }
    }
    public Value search(Key key) {
        Node x = root;
        while (x != null) {
            int cmp = key.compareTo(x.key);
            if (cmp == 0) return x.val;
            else if (cmp < 0) x = x.left;
            else if (cmp > 0) x = x.right;
        }
        return null;
    }
    public void insert(Key key, Value value) {
        root = insert(root, key, value);
        root.color = BLACK;
    }
    private Node insert(Node h, Key key, Value value) {
        if (h == null) return new Node(key, value);
        if (isRed(h.left) && isRed(h.right)) colorFlip(h);
        int cmp = key.compareTo(h.key);
        if (cmp == 0) h.val = value;
        else if (cmp < 0) h.left = insert(h.left, key, value);
        else h.right = insert(h.right, key, value);
        if (isRed(h.right) && !isRed(h.left)) h = rotateLeft(h);
        if (isRed(h.left) && isRed(h.left.left)) h = rotateRight(h);
        return h;
    }
}

public void insert(Key key, Value value) {
    root = insert(root, key, value);
    root.color = BLACK;
}
Lots of cool Self-Balancing BSTs out there!

Popular self-balancing BSTs include:

- AVL tree
- Splay tree
- 2-3 tree
- AA tree
- Red-black tree
- Scapegoat tree
- Treap

(From https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree#Implementations)

(Not covered in this class, but several are in the textbook and all of them are online!)
Trie Introduction
Implementation
Prefix Matching
Interview Question Prep
The Trie: A Specialized Data Structure

- Tries view its keys as:
  - a sequence of characters
  - some (hopefully many!) sequences share common prefixes

- sap
- sad
- awls
- a
- same
- sam

Set ADT

Trie

Diagram of Trie structure:

```
  a
 / \ 
 s   s
 /   / \
 w   a
 / \ /   / \
 l  d  m  p
 / \ /   / \
 s  e s  e
```
Trie: An Introduction

- Each level of the tree represents an index in the string
  - Children at that level represent possible characters at that index
- This abstract trie stores the set of strings:
  - awls, a, sad, same, sap, sam
- How to deal with a and awls?
  - Mark which nodes complete a string (shown in purple)
Searching in Tries

Two ways to fail a `contains()` check:

1. If we fall off the tree
2. If the final node isn’t purple (not a key)

<table>
<thead>
<tr>
<th>Input String</th>
<th>Fall Off? / Is Key?</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>contains(&quot;sam&quot;)</code></td>
<td>hit / purple</td>
<td>True</td>
</tr>
<tr>
<td><code>contains(&quot;sa&quot;)</code></td>
<td>hit / white</td>
<td>False</td>
</tr>
<tr>
<td><code>contains(&quot;a&quot;)</code></td>
<td>hit / purple</td>
<td>True</td>
</tr>
<tr>
<td><code>contains(&quot;saq&quot;)</code></td>
<td>fell off / n/a</td>
<td>False</td>
</tr>
</tbody>
</table>
Keys as “a sequence of characters” (1 of 2)

- Most dictionaries treat their keys as an “atomic blob”: you can’t disassemble the key into smaller components
- Tries take the opposite view: keys are a sequence of characters
  - Strings are made of Characters
- But “characters” don’t have to come from the Latin alphabet
  - Character includes most Unicode codepoints (eg, 蛋糕)
  - List<E>
  - byte[]
But “characters” don’t have to come from the Latin alphabet
- Character includes most Unicode codepoints (e.g., 蛋糕)
- List<E>
- byte[]

Tries are defined by 3 types instead of 2:
- An “alphabet”: the domain of the characters
- A “key”: a sequence of “characters” from the alphabet
- A “value”: the usual Dictionary value
### ASCII TABLE

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>Binary</th>
<th>Octal Char</th>
<th>ASCII</th>
<th>Decimal</th>
<th>Hexadecimal</th>
<th>Binary</th>
<th>Octal Char</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NULL</td>
<td>48</td>
<td>30</td>
<td>0000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>START OF RECORD</td>
<td>49</td>
<td>31</td>
<td>0000001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>10</td>
<td>2</td>
<td>END OF RECORD</td>
<td>50</td>
<td>32</td>
<td>0000010</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>11</td>
<td>3</td>
<td>END OF TEXT</td>
<td>51</td>
<td>33</td>
<td>0000011</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>100</td>
<td>4</td>
<td>END OF TRANSMISSION</td>
<td>52</td>
<td>34</td>
<td>0000100</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>101</td>
<td>5</td>
<td>ENQUIRY</td>
<td>53</td>
<td>35</td>
<td>0000101</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>110</td>
<td>6</td>
<td>ACKNOWLEDGE</td>
<td>54</td>
<td>36</td>
<td>0000110</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>111</td>
<td>7</td>
<td>BEL</td>
<td>55</td>
<td>37</td>
<td>0000111</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>1000</td>
<td>8</td>
<td>BACKSPACE</td>
<td>56</td>
<td>38</td>
<td>0100000</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>1001</td>
<td>9</td>
<td>HT</td>
<td>57</td>
<td>39</td>
<td>0100001</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>1010</td>
<td>A</td>
<td>LF</td>
<td>58</td>
<td>40</td>
<td>0100010</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>1011</td>
<td>B</td>
<td>VT</td>
<td>59</td>
<td>41</td>
<td>0100011</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>1100</td>
<td>C</td>
<td>FF</td>
<td>60</td>
<td>42</td>
<td>0100100</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>1101</td>
<td>D</td>
<td>CR</td>
<td>61</td>
<td>43</td>
<td>0100101</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>1110</td>
<td>E</td>
<td>SUB</td>
<td>62</td>
<td>44</td>
<td>0100110</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>1111</td>
<td>F</td>
<td>HT</td>
<td>63</td>
<td>45</td>
<td>0100111</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10000</td>
<td>10</td>
<td>DATA LINK ESCAPE</td>
<td>64</td>
<td>46</td>
<td>0100100</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>10001</td>
<td>11</td>
<td>START OF TEXT</td>
<td>65</td>
<td>47</td>
<td>0100101</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>10010</td>
<td>12</td>
<td>END OF RECORD</td>
<td>66</td>
<td>48</td>
<td>0100110</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>13</td>
<td>10011</td>
<td>13</td>
<td>ENQUIRY</td>
<td>67</td>
<td>49</td>
<td>0100111</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>10100</td>
<td>14</td>
<td>ACKNOWLEDGE</td>
<td>68</td>
<td>50</td>
<td>0101000</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>10101</td>
<td>15</td>
<td>BEL</td>
<td>69</td>
<td>51</td>
<td>0101001</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>10110</td>
<td>16</td>
<td>BS</td>
<td>70</td>
<td>52</td>
<td>0101010</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>10111</td>
<td>17</td>
<td>HT</td>
<td>71</td>
<td>53</td>
<td>0101011</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>18</td>
<td>11000</td>
<td>18</td>
<td>CR</td>
<td>72</td>
<td>54</td>
<td>0101100</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td>11001</td>
<td>19</td>
<td>LF</td>
<td>73</td>
<td>55</td>
<td>0101101</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>11010</td>
<td>20</td>
<td>VT</td>
<td>74</td>
<td>56</td>
<td>0101110</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>21</td>
<td>21</td>
<td>11011</td>
<td>21</td>
<td>FF</td>
<td>75</td>
<td>57</td>
<td>0101111</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>22</td>
<td>11100</td>
<td>22</td>
<td>ESCAPE</td>
<td>76</td>
<td>58</td>
<td>0110000</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>23</td>
<td>23</td>
<td>11101</td>
<td>23</td>
<td>SUBSTITUTE</td>
<td>77</td>
<td>59</td>
<td>0110001</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>24</td>
<td>11110</td>
<td>24</td>
<td>GROUP SEPARATOR</td>
<td>78</td>
<td>60</td>
<td>0110010</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>11111</td>
<td>25</td>
<td>RECORD SEPARATOR</td>
<td>79</td>
<td>61</td>
<td>0110011</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>26</td>
<td>26</td>
<td>100000</td>
<td>26</td>
<td>START OF TRANS BLOCK</td>
<td>80</td>
<td>62</td>
<td>0110100</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>27</td>
<td>27</td>
<td>100001</td>
<td>27</td>
<td>CANCEL</td>
<td>81</td>
<td>63</td>
<td>0110101</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>28</td>
<td>100010</td>
<td>28</td>
<td>END OF MEDIUM</td>
<td>82</td>
<td>64</td>
<td>0110110</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>29</td>
<td>29</td>
<td>100011</td>
<td>29</td>
<td>SUBSCRIBE</td>
<td>83</td>
<td>65</td>
<td>0110111</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>100100</td>
<td>30</td>
<td>SUBSCRIBE</td>
<td>84</td>
<td>66</td>
<td>0111000</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>31</td>
<td>31</td>
<td>100101</td>
<td>31</td>
<td>FILE SEPARATOR</td>
<td>85</td>
<td>67</td>
<td>0111001</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>100110</td>
<td>32</td>
<td>GROUP SEPARATOR</td>
<td>86</td>
<td>68</td>
<td>0111010</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>33</td>
<td>33</td>
<td>100111</td>
<td>33</td>
<td>RECORD SEPARATOR</td>
<td>87</td>
<td>69</td>
<td>0111011</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>34</td>
<td>34</td>
<td>101000</td>
<td>34</td>
<td>SPACE</td>
<td>88</td>
<td>70</td>
<td>0111100</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>35</td>
<td>35</td>
<td>101001</td>
<td>35</td>
<td>WIRE</td>
<td>89</td>
<td>71</td>
<td>0111101</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>36</td>
<td>36</td>
<td>101010</td>
<td>36</td>
<td>END OF TRANSM</td>
<td>90</td>
<td>72</td>
<td>0111110</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>37</td>
<td>37</td>
<td>101011</td>
<td>37</td>
<td>CLEAR</td>
<td>91</td>
<td>73</td>
<td>0111111</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>38</td>
<td>38</td>
<td>101100</td>
<td>38</td>
<td>START OF TEXT</td>
<td>92</td>
<td>74</td>
<td>0110000</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>39</td>
<td>39</td>
<td>101101</td>
<td>39</td>
<td>END OF RECORD</td>
<td>93</td>
<td>75</td>
<td>0110001</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
<td>101110</td>
<td>40</td>
<td>ENQUIRY</td>
<td>94</td>
<td>76</td>
<td>0110010</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>41</td>
<td>41</td>
<td>101111</td>
<td>41</td>
<td>ACKNOWLEDGE</td>
<td>95</td>
<td>77</td>
<td>0110011</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>42</td>
<td>42</td>
<td>100000</td>
<td>42</td>
<td>BEL</td>
<td>96</td>
<td>78</td>
<td>0110100</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>43</td>
<td>43</td>
<td>100001</td>
<td>43</td>
<td>BS</td>
<td>97</td>
<td>79</td>
<td>0110101</td>
<td>43</td>
<td>43</td>
</tr>
<tr>
<td>44</td>
<td>44</td>
<td>100010</td>
<td>44</td>
<td>CR</td>
<td>98</td>
<td>80</td>
<td>0110110</td>
<td>44</td>
<td>44</td>
</tr>
<tr>
<td>45</td>
<td>45</td>
<td>100011</td>
<td>45</td>
<td>LF</td>
<td>99</td>
<td>81</td>
<td>0110111</td>
<td>45</td>
<td>45</td>
</tr>
<tr>
<td>46</td>
<td>46</td>
<td>100100</td>
<td>46</td>
<td>VT</td>
<td>100</td>
<td>82</td>
<td>0111000</td>
<td>46</td>
<td>46</td>
</tr>
<tr>
<td>47</td>
<td>47</td>
<td>100101</td>
<td>47</td>
<td>FF</td>
<td>101</td>
<td>83</td>
<td>0111001</td>
<td>47</td>
<td>47</td>
</tr>
</tbody>
</table>

**Notes:**
- **NULL:** No character, used for the null character in strings.
- **START OF RECORD:** Marks the start of a new record.
- **END OF RECORD:** Marks the end of a record.
- **END OF TEXT:** Marks the end of a text block.
- **ENQUIRY:** Used for inquiries or questions.
- **ACKNOWLEDGE:** Acknowledges receipt of data.
- **BEL:** Bell character, used to signal a line feed.
- **BACKSPACE:** Backspaces the cursor.
- **HT:** Horizontal tabulator.
- **VT:** Vertical tabulator.
- **FF:** Form feed character.
- **CR:** Carriage return.
- **SUB:** Substitute.
- **SUBSCRIBE:** Magazine subscription.
- **FILE SEPARATOR:** Separates files.
- **GROUP SEPARATOR:** Separates groups.
- **RECORD SEPARATOR:** Separates records.
- **SPACE:** Spaces the text.
- **WIRE:** Used for wire transmission.
- **END OF TRANSM:** Marks the end of a transmission block.
- **CLEAR:** Clears the screen.
- **START OF TEXT:** Marks the start of text.
Simple Trie Implementation*

public class TrieSet {
    private Node root;

    private static class Node {
        private char ch;
        private boolean isKey;
        private Map<Character, Node> next;
        private Node(char c, boolean b) {
            ch = c;
            isKey = b;
            next = new HashMap();
        }
    }
}

![Trie Diagram]
Simple Trie Node Implementation

private static class Node {
    private char ch;
    private boolean isKey;
    private Map<char, Node> next;
    ...
}

Node

<table>
<thead>
<tr>
<th>ch</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>isKey</td>
<td>true</td>
</tr>
<tr>
<td>next</td>
<td></td>
</tr>
</tbody>
</table>

Map

| y     |

Node

<table>
<thead>
<tr>
<th>ch</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>isKey</td>
<td>false</td>
</tr>
<tr>
<td>next</td>
<td></td>
</tr>
</tbody>
</table>

a

y
Simple Trie Implementation

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

    private static class Node {
        private char ch;
        private boolean isKey;
        private Map<Character, Node> next;
        private Node(char c, boolean b) {
            ch = c;
            isKey = b;
            next = new HashMap();
        }
    }
}
```
Trie-Specific Operations

- The main appeal of tries is prefix matching!
  - Why? Because they view their keys as sequences that can have prefixes
- Longest prefix
  - `longestPrefixOf("sample")`
  - Want: `{"sam"}`
- Prefix match
  - `findPrefix("sa")`
  - Want: `{"sad", "sam", "same", "sap"}`
Related Problem: Collecting Trie Keys

- Imagine an algorithm that collects all the keys in a trie:
  - `collect()`:
    ```
    ["a","awls","sad","sam","same","sap"]
    ```
- It could be implemented as follows:

```markdown
Create an empty list of results x
For each character c in root.next.keys():
  call collectHelper(c, x, root.next.get(c))
return x
```
Summary

- A trie data structure implements the Dictionary and Set ADTs
- Tries have many different implementations
  - Could store HashMap/TreeMap/any-dictionary within nodes
  - Much more exotic variants change the trie's representation, such as the Ternary Search Trie
- Tries store sequential keys
  - … which enables very efficient prefix operations like findPrefix
Trie Introduction
Implementation
Prefix Matching

Interview Question Prep
Interview Prep

- Any time you see word/letter parsing!
  - fast run time with tries (quick word / letter lookup)
  - not just for interview, but real-world applications
- Interviewer’s favorite “gimmick” question
  - came up for me
- Example Problem:
  **Find first ‘k’ maximum occurring words in a given set of strings**
  - see if you can do this problem on your own
Questions?
Your toolbox so far...

ADT
○ List – flexibility, easy movement of elements within structure
○ Stack – optimized for first in last out ordering
○ Queue – optimized for first in first out ordering
○ Dictionary (Map) – stores two pieces of data at each entry

Data Structure Implementation
○ Array – easy look up, hard to rearrange
○ Linked Nodes – hard to look up, easy to rearrange
○ Hash Table – constant time look up, no ordering of data
○ BST – efficient look up, possibility of bad worst case
○ AVL Tree – efficient look up, protects against bad worst case, hard to implement

<- It’s all about data baby!
SUPER common in comp sci
- Databases
- Network router tables
- Compilers and Interpreters
Review: Dictionaries

Why are we so obsessed with Dictionaries?

When dealing with data:
- Adding data to your collection
- Getting data out of your collection
- Rearranging data in your collection

**Dictionary ADT**

<table>
<thead>
<tr>
<th>state</th>
<th>behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of items &amp; keys</td>
<td>put(key, item) add item to collection indexed with key</td>
</tr>
<tr>
<td>Count of items</td>
<td>get(key) return item associated with key</td>
</tr>
<tr>
<td></td>
<td>containsKey(key) return if key already in use</td>
</tr>
<tr>
<td></td>
<td>remove(key) remove item and associated key</td>
</tr>
<tr>
<td></td>
<td>size() return count of items</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>ArrayList</th>
<th>LinkedList</th>
<th>HashTable</th>
<th>BST</th>
<th>AVLTTree</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key, value)</td>
<td>best</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(log n)</td>
</tr>
<tr>
<td>get(key)</td>
<td>best</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(log n)</td>
</tr>
<tr>
<td>remove(key)</td>
<td>best</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(n)</td>
<td>Θ(log n)</td>
</tr>
</tbody>
</table>
Design Decisions

Before coding can begin engineers must carefully consider the design of their code will organize and manage data.

Things to consider:

- What functionality is needed?
  - What operations need to be supported?
  - Which operations should be prioritized?

- What type of data will you have?
  - What are the relationships within the data?
  - How much data will you have?
  - Will your data set grow?
  - Will your data set shrink?

- How do you think things will play out?
  - How likely are best cases?
  - How likely are worst cases?
Example: Class Gradebook

You have been asked to create a new system for organizing students in a course and their accompanying grades

What type of data will you have?
What are the relationships within the data?
How much data will you have?
Will your data set grow?
Will your data set shrink?
How do you think things will play out?
How likely are best cases?
How likely are worst cases?

What functionality is needed?
What operations need to be supported?
Add students to course
Add grade to student's record
Update grade already in student's record
Remove student from course
Check if student is in course
Find specific grade for student
Which operations should be prioritized?

Add students to course
Add grade to student's record
Update grade already in student's record
Remove student from course
Check if student is in course
Find specific grade for student

A couple hundred students, < 20 grades per student
A lot at the beginning, Not much after that
Lots of add and drops?
Lots of grade updates?
Students with similar identifiers?
Example: Class Gradebook

What data should we use to identify students? (keys)
○ Student IDs – unique to each student, no confusion (or collisions)
○ Names – easy to use, support easy to produce sorted by name

How should we store each student’s grades? (values)
○ Array List – easy to access, keeps order of assignments
○ Hash Table – super efficient access, no order maintained

Which data structure is the best fit to store students and their grades?
○ Hash Table – student IDs as keys will make access very efficient
○ AVL Tree – student names as keys will maintain alphabetical order
Practice: Music Storage

You have been asked to create a new system for organizing songs in a music service. For each song you need to store the artist and how many plays that song has.

What functionality is needed?

• What operations need to be supported?
  • Update number of plays for a song
  • Add a new song to an artist’s collection
  • Add a new artist and their songs to the service
  • Find an artist’s most popular song
  • Find service’s most popular artist

• What operations should be prioritized?
  • More…

What type of data will you have?

• What are the relationships within the data?
  • Artists need to be associated with their songs,
  • Songs need to be associated with their play counts

• How much data will you have?
  • Play counts will get updated a lot
  • New songs will get added regularly

• Will your data set grow?

• Will your data set shrink?

How do you think things will play out?

• How likely are best cases?
  • Some artists and songs will need to be accessed a lot more than others

• How likely are worst cases?
  • Artist and song names can be very similar
Practice: Music Storage

● How should we store songs and their play counts?
  ○ Hash Table – song titles as keys, play count as values, quick access for updates
  ○ Array List – song titles as keys, play counts as values, maintain order of addition to system

● How should we store artists with their associated songs?
  ○ Hash Table – artist as key,
    ■ Hash Table of their (songs, play counts) as values
    ■ AVL Tree of their songs as values
  ○ AVL Tree – artists as key, hash tables of songs and counts as values
Questions?
That’s all!