Tries

Instructor: Hunter Schafer

TAs:
- Ken Aragon
- Khushi Chaudhari
- Joyce Elauria
- Santino Iannone
- Leona Kazi
- Nathan Liparski
- Sam Long
- Amanda Park

Paul Pham
- Mitchell Szeto
- Batina Shikhalieva
- Ryan Siu
- Elena Spasova
- Alex Teng
- Blarry Wang
- Aileen Zeng
Learning Objectives

After this lecture, you should be able to...

Today is an optional, “cool” data structure day. Since it’s not part of our core course, we don’t outline learning objectives.

The class session for today will just be Q&A since we don’t have practice for this material.
Lecture Outline

• Tries Introduction

• Implementing a Trie using arrays

• Advanced Implementations: dealing with sparsity
  - Hash Tables, BSTs, and Ternary Search Trees
Autocomplete

• Search Engines support autocomplete.
• How do you efficiently implement autocomplete with the ADTs we know so far?
• Formal Problem: Given a “prefix” of a string, find all strings in a set of possible strings that have the given prefix.
Tries: A *Specialized* Data Structure

- Tries are a character-by-character set-of-Strings implementation
- Nodes store *parts of keys* instead of *keys*
Abstract Trie

• Each level represents an index
  - Children represent next possible characters at that index

• This Trie stores the following set of Strings:
  
  a, aqua, dad,
  
  data, day, days

• How do we deal with a and aqua?
  - Mark complete Strings with a boolean (shown in blue)
  - Complete string: a String that belongs in our set
Searching in Tries

Search hit: the final node is a key (colored blue)

Search miss: caused in one of two ways
1. The final node is not a key (not colored blue)
2. We “fall” off the Trie

contains("data") // hit, l = 4
contains("da") // miss, l = 2
contains("a") // hit, l = 1
contains("dubs") // miss, l = 4

contains runtime given key of length l with n keys in Trie: Θ(l)
Prefix Operations with Tries

• The main appeal of Tries is its efficient prefix matching!

• **Prefix**: find set of keys associated with given prefix
  
  ```
  keysWithPrefix("day") returns ["day", "days"]
  ```

• **Longest Prefix From Trie**: given a String, retrieve longest prefix of that String that exists in the Trie
  
  ```
  longestPrefixOf("aquarium") returns "aqua"
  longestPrefixOf("aqueous") returns "aqu"
  longestPrefixOf("dawgs") returns "da"
  ```
Collecting Trie Keys

**Collect:** return set of all keys in the Trie (like `keySet()`)  

\[
\text{collect(trie)} = \{ "a", "aqua", "dad", "data", "day", "days" \}
\]

```java
List collect() {
    List keys;
    for (Node c : root.children) {
        collectHelper(n.char, keys, c);
    }
    return keys;
}

void collectHelper(String str, List keys, Node n) {
    if (n.isKey()) {
        keys.add(s);
    }
    for (Node c : n.children) {
        collectHelper(str + c.char, keys, c);
    }
}
```
keysWithPrefix Implementation

- **keysWithPrefix(String prefix)**
  - Find all the keys that corresponds to the given prefix

```java
List keysWithPrefix(String prefix) {
    Node root;  // Node corresponding to given prefix
    List keys;  // Empty list to store keys
    for (Node n : root.children) {
        collectHelper(prefix + n.char, keys, c);
    }
}

void collectHelper(String str, List keys, Node n) {
    if (n.isKey()) {
        keys.add(s);
    }
    for (Node c : n.children) {
        collectHelper(str + c.char, keys, c);
    }
}
```
Autocomplete with Tries

• Autocomplete should return the **most relevant results**

• One method: a Trie-based `Map<String, Relevance>`
  - When a user types in a string "hello", call `keysWithPrefix("hello")`
  - Return the 10 Strings with the highest relevance
Lecture Outline

• Tries Introduction

• Implementing a Trie using arrays

• Advanced Implementations: dealing with sparsity
  - Hash Tables, BSTs, and Ternary Search Trees
# Trie Implementation Idea: Encoding

## ASCII Table

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>Oct</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Oct</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Oct</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Oct</th>
<th>Char</th>
<th>Dec</th>
<th>Hex</th>
<th>Oct</th>
<th>Char</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>[space]</td>
<td>32</td>
<td>20</td>
<td>40</td>
<td>@</td>
<td>64</td>
<td>40</td>
<td>100</td>
<td>@</td>
<td>96</td>
<td>60</td>
<td>140</td>
<td>`</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>!</td>
<td>33</td>
<td>21</td>
<td>41</td>
<td>A</td>
<td>65</td>
<td>41</td>
<td>101</td>
<td>A</td>
<td>97</td>
<td>61</td>
<td>141</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>&quot;</td>
<td>34</td>
<td>22</td>
<td>42</td>
<td>B</td>
<td>66</td>
<td>42</td>
<td>102</td>
<td>B</td>
<td>98</td>
<td>62</td>
<td>142</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>#</td>
<td>35</td>
<td>23</td>
<td>43</td>
<td>C</td>
<td>67</td>
<td>43</td>
<td>103</td>
<td>C</td>
<td>99</td>
<td>63</td>
<td>143</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>$</td>
<td>36</td>
<td>24</td>
<td>44</td>
<td>D</td>
<td>68</td>
<td>44</td>
<td>104</td>
<td>D</td>
<td>100</td>
<td>64</td>
<td>144</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>%</td>
<td>37</td>
<td>25</td>
<td>45</td>
<td>E</td>
<td>69</td>
<td>45</td>
<td>105</td>
<td>E</td>
<td>101</td>
<td>65</td>
<td>145</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>&amp;</td>
<td>38</td>
<td>26</td>
<td>46</td>
<td>F</td>
<td>70</td>
<td>46</td>
<td>106</td>
<td>F</td>
<td>102</td>
<td>66</td>
<td>146</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>'</td>
<td>39</td>
<td>27</td>
<td>47</td>
<td>G</td>
<td>71</td>
<td>47</td>
<td>107</td>
<td>G</td>
<td>103</td>
<td>67</td>
<td>147</td>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>(</td>
<td>40</td>
<td>28</td>
<td>48</td>
<td>H</td>
<td>72</td>
<td>48</td>
<td>108</td>
<td>H</td>
<td>104</td>
<td>68</td>
<td>150</td>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td>)</td>
<td>41</td>
<td>29</td>
<td>51</td>
<td>I</td>
<td>73</td>
<td>49</td>
<td>111</td>
<td>I</td>
<td>105</td>
<td>69</td>
<td>151</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>12</td>
<td>*</td>
<td>42</td>
<td>2A</td>
<td>52</td>
<td>J</td>
<td>74</td>
<td>4A</td>
<td>112</td>
<td>J</td>
<td>106</td>
<td>6A</td>
<td>152</td>
<td>j</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>13</td>
<td>+</td>
<td>43</td>
<td>2B</td>
<td>53</td>
<td>K</td>
<td>75</td>
<td>4B</td>
<td>113</td>
<td>K</td>
<td>107</td>
<td>6B</td>
<td>153</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>14</td>
<td>,</td>
<td>44</td>
<td>2C</td>
<td>54</td>
<td>L</td>
<td>76</td>
<td>4C</td>
<td>114</td>
<td>L</td>
<td>108</td>
<td>6C</td>
<td>154</td>
<td>l</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>15</td>
<td>-</td>
<td>45</td>
<td>2D</td>
<td>55</td>
<td>M</td>
<td>77</td>
<td>4D</td>
<td>115</td>
<td>M</td>
<td>109</td>
<td>6D</td>
<td>155</td>
<td>m</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>16</td>
<td>.</td>
<td>46</td>
<td>2E</td>
<td>56</td>
<td>N</td>
<td>78</td>
<td>4E</td>
<td>116</td>
<td>N</td>
<td>110</td>
<td>6E</td>
<td>156</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>17</td>
<td>/</td>
<td>47</td>
<td>2F</td>
<td>57</td>
<td>O</td>
<td>79</td>
<td>4F</td>
<td>117</td>
<td>O</td>
<td>111</td>
<td>6F</td>
<td>157</td>
<td>o</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>20</td>
<td>0</td>
<td>48</td>
<td>30</td>
<td>60</td>
<td>P</td>
<td>80</td>
<td>50</td>
<td>120</td>
<td>P</td>
<td>112</td>
<td>70</td>
<td>160</td>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>21</td>
<td>1</td>
<td>49</td>
<td>31</td>
<td>61</td>
<td>Q</td>
<td>81</td>
<td>51</td>
<td>121</td>
<td>Q</td>
<td>113</td>
<td>71</td>
<td>161</td>
<td>q</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>12</td>
<td>22</td>
<td>2</td>
<td>50</td>
<td>32</td>
<td>62</td>
<td>R</td>
<td>82</td>
<td>52</td>
<td>122</td>
<td>R</td>
<td>114</td>
<td>72</td>
<td>162</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>13</td>
<td>23</td>
<td>3</td>
<td>51</td>
<td>33</td>
<td>63</td>
<td>S</td>
<td>83</td>
<td>53</td>
<td>123</td>
<td>S</td>
<td>115</td>
<td>73</td>
<td>163</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>24</td>
<td>4</td>
<td>52</td>
<td>34</td>
<td>64</td>
<td>T</td>
<td>84</td>
<td>54</td>
<td>124</td>
<td>T</td>
<td>116</td>
<td>74</td>
<td>164</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>15</td>
<td>25</td>
<td>5</td>
<td>53</td>
<td>35</td>
<td>65</td>
<td>U</td>
<td>85</td>
<td>55</td>
<td>125</td>
<td>U</td>
<td>117</td>
<td>75</td>
<td>165</td>
<td>u</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>16</td>
<td>26</td>
<td>6</td>
<td>54</td>
<td>36</td>
<td>66</td>
<td>V</td>
<td>86</td>
<td>56</td>
<td>126</td>
<td>V</td>
<td>118</td>
<td>76</td>
<td>166</td>
<td>v</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>27</td>
<td>7</td>
<td>55</td>
<td>37</td>
<td>67</td>
<td>W</td>
<td>87</td>
<td>57</td>
<td>127</td>
<td>W</td>
<td>119</td>
<td>77</td>
<td>167</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>28</td>
<td>8</td>
<td>56</td>
<td>38</td>
<td>70</td>
<td>X</td>
<td>88</td>
<td>58</td>
<td>130</td>
<td>X</td>
<td>120</td>
<td>78</td>
<td>170</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>29</td>
<td>9</td>
<td>57</td>
<td>39</td>
<td>71</td>
<td>Y</td>
<td>89</td>
<td>59</td>
<td>131</td>
<td>Y</td>
<td>121</td>
<td>79</td>
<td>171</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>30</td>
<td>a</td>
<td>58</td>
<td>3A</td>
<td>72</td>
<td>Z</td>
<td>90</td>
<td>5A</td>
<td>132</td>
<td>Z</td>
<td>122</td>
<td>7A</td>
<td>172</td>
<td>z</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>31</td>
<td>b</td>
<td>59</td>
<td>3B</td>
<td>73</td>
<td></td>
<td>91</td>
<td>5B</td>
<td>133</td>
<td></td>
<td>123</td>
<td>7B</td>
<td>173</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>32</td>
<td>c</td>
<td>60</td>
<td>3C</td>
<td>74</td>
<td></td>
<td>92</td>
<td>5C</td>
<td>134</td>
<td></td>
<td>124</td>
<td>7C</td>
<td>174</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>33</td>
<td>d</td>
<td>61</td>
<td>3D</td>
<td>75</td>
<td></td>
<td>93</td>
<td>5D</td>
<td>135</td>
<td></td>
<td>125</td>
<td>7D</td>
<td>175</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>34</td>
<td>e</td>
<td>62</td>
<td>3E</td>
<td>76</td>
<td></td>
<td>94</td>
<td>5E</td>
<td>136</td>
<td></td>
<td>126</td>
<td>7E</td>
<td>176</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>35</td>
<td>f</td>
<td>63</td>
<td>3F</td>
<td>77</td>
<td></td>
<td>95</td>
<td>5F</td>
<td>137</td>
<td></td>
<td>127</td>
<td>7F</td>
<td>177</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
class TrieSet {
    final int R = 128;  // # of ASCII encodings
    Node overallRoot;

    // Private internal class
    class Node {
        // Field declarations
        char ch;
        boolean isKey;
        DataIndexedCharMap<Node> next;  // array encoding

        // Constructor
        Node(char c, boolean b, int R) {
            ch = c;
            isKey = b;
            next = new DataIndexedCharMap<Node>(R);
        }
    }
}
Data Structure for Trie Implementation

• Think of a Binary Tree
  - Instead of two children, we have 128 possible children
  - Each child represents a possible next character of our Trie

• How could we store these 128 children?
Data-Indexed Array Visualization

// Private internal class
class Node {
   // Field declarations
   char ch;
   boolean isKey;
   DataIndexedCharMap<Node> next;
}

R = 128 links, 127 null
Removing Redundancy

class TrieSet {
    final int R = 128;
    Node overallRoot;

    // Private internal class
class Node {
        // Field declarations
        char ch;
        boolean isKey;
        DataIndexedCharMap<Node> next;

        // Constructor
        Node(char c, boolean b, int R) {
            ch = c;
            isKey = b;
            next = new DataIndexedCharMap<Node>(R);
        }
    }
}
Does the structure of a Trie depend on the order of insertion?

a) Yes
b) No
c) I’m not sure…
## Runtime Comparison

- Typical runtime when treating length \( l \) of keys as a constant:

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Key Type</th>
<th>contains</th>
<th>add</th>
<th>keysWithPrefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balanced BST</td>
<td>Comparable</td>
<td>( \Theta(\log(n)) )</td>
<td>( \Theta(\log(n)) )</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>Hash Map</td>
<td>Hashable</td>
<td>( \Theta(1)^* )</td>
<td>( \Theta(1)^* )</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>Trie (Data-Indexed Array)</td>
<td>String (Character)</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
<td>( \Theta(p)^{**} )</td>
</tr>
</tbody>
</table>

* In-practice runtime

** Where \( p \) is the number of strings with the given prefix. Usually \( p \ll n \).

### Takeaways:

- When keys are Strings, Tries give us a better \texttt{add} and \texttt{contains} runtime
- \texttt{DataIndexedCharMap} takes up a lot of space by storing \( R \) links per node

\[ R = 128 \text{ links, 127 null (unused)} \]
Lecture Outline

• Tries Introduction

• Implementing a Trie using arrays

• Advanced Implementations: dealing with sparsity
  - Hash Tables, BSTs, and Ternary Search Trees
DataIndexedCharMap Implementation

Abstract Trie

Data-Indexed Array Trie
Hash Table-based Implementation

- Use Hash Table to find character at a given index
BST-based Implementation

• Use Binary Search Tree to find character at a given index

• Two types of children:
  1) “Trie” child: advance a character (index)
  2) “Internal” child: another character option at current character (index)

• Both are essentially child references
  - Could we simplify this design?
Ternary Search Tree (TST) Implementation

• Combines character mapping with Trie itself

“Internal” left child (smaller character at same index)

“Trie” child: advance to next String index

“Internal” right child (greater character at same index)
Which node is associated with the key "CAC"?
Searching in a TST

- Searching in a TST
  - If smaller, take left link
  - If greater, take right link
  - If equal, take the middle link and move to next character

- **Search hit:** final node yields a key that belongs in our set
- **Search miss:** reach null link or final node is yields a key not in our set
Trie Takeaways

• Tries can be used for storing Strings (or any sequential data)
• Real-world performance often better than Hash Table or Search Tree
• Many different implementations: DataIndexedCharMap, Hash Tables, BSTs (and more possible data structures within nodes), and TSTs
• Tries enable efficient prefix operations like keysWithPrefix

Binary Search Tree

Trie

Hash Table