Lecture 26: Tries

CSE 373: Data Structures and Algorithms
Memory & B Trees Takeaways

Memory in your computer is organized into layers spanning out from the CPU > L1 Cache > L2 Cache > RAM > Disk
- cache: a place to store some memory that’s smaller and closer to the CPU compared to RAM
  - Analogy: refrigerator keeps food closer to kitchen than store
- Getting data from cache to CPU is a lot quicker than from RAM to CPU.
- the more memory a layer can store, the slower it is (generally)
- accessing the disk is very slow

Operating system is “the memory boss”
- pulls data from outer layers closer to CPU based on usage
- When do we “evict” memory to make room? How does the OS minimize disk accesses?
  - Temporal locality: Once we load something into RAM or cache, keep it around or a while
    - Analogy: top layer of clothing stays on floor, don’t wash that knife you just used to cut a sandwich just yet (I am a good adult I swear)
    - EX: if you accessed one index of an array in the cache, keep that array around for a while
  - Spatial Locality: Computers try to partition memory you are likely to use close by
    - Analogy: if you’re going to your storage unit, might as well fill up your car
    - EX: if you ask for one index of an array from RAM, pull the whole array down
Announcements

P4 due a week from today
- please get started soon

Interview practice tomorrow in section

TA Career Panel

Memorial day next week
- no class, no office hours
- Office hours end on Friday June 4th

Grades posted to canvas
- late penalties have yet to be applied
- please let us know if there are any issues
- regrade requests live on gradescope

Course Evals are out
- please fill out the official one
- supplemental one – if we get over 90.5% everyone gets 1-point ec!

TA Lead Final Review
Final Exam Topics

**Sorting**
- Sorting algorithm properties (stable, in-place)
- Quadratic sorts: insertion sort, selection sort
- Faster sorts: heap sort, merge sort, quick sort
- Runtimes of all of the above (in the best and worst case)

**Graphs**
- Definitions (e.g., directed, undirected, weighted, unweighted, walks, paths, cycles, self-loops, parallel edges, trees, DAGs, etc.)
- Implementations (Adjacency list, Adjacency matrix, and their pros and cons)
- Traversals (BFS and DFS)
- Single-source shortest-path algorithms: Dijkstra’s algorithm
- Topological sort
- MST algorithms: Prim and Kruskal
- Disjoint sets
- Framing/modeling problems with graphs

**Coding Projects**
- Implementation of each data structure
- Best / average / worst case runtime for each method of each data structure

**Design Decisions**
- Given a scenario, what ADT, data structure implementation and/or algorithm is best optimized for your goals?
  - What is unique or specialized about your chosen tool?
  - How do the specialized features of your chosen tool contribute to solving the given problem scenario?
  - How expensive is this tool and its features in terms of runtime and memory?
- Given a scenario, what changes might you make to a design to better serve your goals?

**Memory and Locality**
- How to leverage caching

**Pre-midterm topics**
- All ADTs and data structures
- Asymptotic analysis
  - Code Modeling (including recurrences)
  - Complexity Classes
  - Big O, Omega, Theta
- BSTs, AVL Trees
- Hashing
- Heaps
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*.

---

Binary Search Tree

<table>
<thead>
<tr>
<th>a</th>
<th>paw</th>
<th>par</th>
<th>parts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Trie

<table>
<thead>
<tr>
<th>a</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>q</td>
<td>a</td>
</tr>
<tr>
<td>u</td>
<td>d</td>
</tr>
<tr>
<td>t</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hash Table

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>aqua</td>
<td>a</td>
<td>paw</td>
<td>part</td>
</tr>
<tr>
<td>parts</td>
<td>par</td>
<td>pair</td>
<td>part</td>
</tr>
</tbody>
</table>
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: $\Theta(l)$
Prefix Operations with Tries

The main appeal of Tries is its efficient prefix matching!

**Prefix:** find set of keys associated with given prefix

\[
\text{keysWithPrefix}("day") \text{ returns } ["day", "days"]
\]

**Longest Prefix From Trie:** given a String, retrieve longest prefix of that String that exists in the Trie

\[
\text{longestPrefixOf("aquarium") returns "aqua"}
\]
\[
\text{longestPrefixOf("aqueous") returns "aqu"}
\]
\[
\text{longestPrefixOf("dawgs") returns "da"}
\]
Collecting Trie Keys

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

`collect(trie) = ["a", "aqua", "dad", "data", "day", "days"]`

```
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);
    }
}
```

![Tree Diagram](attachment:tree.png)

```
root
  d
  a
    q
    u
    y
  a
  a
  s
```
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>
</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>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>!</td>
<td>33</td>
<td>21</td>
<td>41</td>
<td>#</td>
<td>65</td>
<td>41</td>
<td>101</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>#</td>
<td>34</td>
<td>22</td>
<td>42</td>
<td>$</td>
<td>66</td>
<td>42</td>
<td>102</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>$</td>
<td>35</td>
<td>23</td>
<td>43</td>
<td>%</td>
<td>67</td>
<td>43</td>
<td>103</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>%</td>
<td>36</td>
<td>24</td>
<td>44</td>
<td>&amp;</td>
<td>68</td>
<td>44</td>
<td>104</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>&amp;</td>
<td>37</td>
<td>25</td>
<td>45</td>
<td>'</td>
<td>69</td>
<td>45</td>
<td>105</td>
<td>E</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>'</td>
<td>38</td>
<td>26</td>
<td>46</td>
<td>(</td>
<td>70</td>
<td>46</td>
<td>106</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>(</td>
<td>39</td>
<td>27</td>
<td>47</td>
<td>)</td>
<td>71</td>
<td>47</td>
<td>107</td>
<td>G</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>)</td>
<td>40</td>
<td>28</td>
<td>50</td>
<td></td>
<td>72</td>
<td>48</td>
<td>110</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>9</td>
<td></td>
<td>41</td>
<td>29</td>
<td>51</td>
<td></td>
<td>73</td>
<td>49</td>
<td>111</td>
<td>I</td>
</tr>
<tr>
<td>10</td>
<td>A</td>
<td>12</td>
<td></td>
<td>42</td>
<td>2A</td>
<td>52</td>
<td>*</td>
<td>74</td>
<td>4A</td>
<td>112</td>
<td>J</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>13</td>
<td>*</td>
<td>43</td>
<td>2B</td>
<td>53</td>
<td>+</td>
<td>75</td>
<td>4B</td>
<td>113</td>
<td>K</td>
</tr>
<tr>
<td>12</td>
<td>C</td>
<td>14</td>
<td>+</td>
<td>44</td>
<td>2C</td>
<td>54</td>
<td>,</td>
<td>76</td>
<td>4C</td>
<td>114</td>
<td>L</td>
</tr>
<tr>
<td>13</td>
<td>D</td>
<td>15</td>
<td>,</td>
<td>45</td>
<td>2D</td>
<td>55</td>
<td></td>
<td>77</td>
<td>4D</td>
<td>115</td>
<td>M</td>
</tr>
<tr>
<td>14</td>
<td>E</td>
<td>16</td>
<td></td>
<td>46</td>
<td>2E</td>
<td>56</td>
<td></td>
<td>78</td>
<td>4E</td>
<td>116</td>
<td>N</td>
</tr>
<tr>
<td>15</td>
<td>F</td>
<td>17</td>
<td></td>
<td>47</td>
<td>2F</td>
<td>57</td>
<td></td>
<td>79</td>
<td>4F</td>
<td>117</td>
<td>O</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>20</td>
<td></td>
<td>48</td>
<td>30</td>
<td>60</td>
<td></td>
<td>80</td>
<td>50</td>
<td>120</td>
<td>P</td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>21</td>
<td></td>
<td>49</td>
<td>31</td>
<td>61</td>
<td></td>
<td>81</td>
<td>51</td>
<td>121</td>
<td>Q</td>
</tr>
<tr>
<td>18</td>
<td>3</td>
<td>22</td>
<td></td>
<td>50</td>
<td>32</td>
<td>62</td>
<td></td>
<td>82</td>
<td>52</td>
<td>122</td>
<td>R</td>
</tr>
<tr>
<td>19</td>
<td>4</td>
<td>23</td>
<td></td>
<td>51</td>
<td>33</td>
<td>63</td>
<td></td>
<td>83</td>
<td>53</td>
<td>123</td>
<td>S</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>24</td>
<td></td>
<td>52</td>
<td>34</td>
<td>64</td>
<td></td>
<td>84</td>
<td>54</td>
<td>124</td>
<td>T</td>
</tr>
<tr>
<td>21</td>
<td>6</td>
<td>25</td>
<td></td>
<td>53</td>
<td>35</td>
<td>65</td>
<td></td>
<td>85</td>
<td>55</td>
<td>125</td>
<td>U</td>
</tr>
<tr>
<td>22</td>
<td>7</td>
<td>26</td>
<td></td>
<td>54</td>
<td>36</td>
<td>66</td>
<td></td>
<td>86</td>
<td>56</td>
<td>126</td>
<td>V</td>
</tr>
<tr>
<td>23</td>
<td>8</td>
<td>27</td>
<td></td>
<td>55</td>
<td>37</td>
<td>67</td>
<td></td>
<td>87</td>
<td>57</td>
<td>127</td>
<td>W</td>
</tr>
<tr>
<td>24</td>
<td>9</td>
<td>30</td>
<td></td>
<td>56</td>
<td>38</td>
<td>70</td>
<td></td>
<td>88</td>
<td>58</td>
<td>130</td>
<td>X</td>
</tr>
<tr>
<td>25</td>
<td>10</td>
<td>31</td>
<td></td>
<td>57</td>
<td>39</td>
<td>71</td>
<td></td>
<td>89</td>
<td>59</td>
<td>131</td>
<td>Y</td>
</tr>
<tr>
<td>26</td>
<td>11</td>
<td>32</td>
<td></td>
<td>58</td>
<td>3A</td>
<td>72</td>
<td></td>
<td>90</td>
<td>5A</td>
<td>132</td>
<td>Z</td>
</tr>
<tr>
<td>27</td>
<td>12</td>
<td>33</td>
<td></td>
<td>59</td>
<td>3B</td>
<td>73</td>
<td></td>
<td>91</td>
<td>5B</td>
<td>133</td>
<td>I</td>
</tr>
<tr>
<td>28</td>
<td>13</td>
<td>34</td>
<td></td>
<td>60</td>
<td>3C</td>
<td>74</td>
<td></td>
<td>92</td>
<td>5C</td>
<td>134</td>
<td>J</td>
</tr>
<tr>
<td>29</td>
<td>14</td>
<td>35</td>
<td></td>
<td>61</td>
<td>3D</td>
<td>75</td>
<td></td>
<td>93</td>
<td>5D</td>
<td>135</td>
<td>K</td>
</tr>
<tr>
<td>30</td>
<td>15</td>
<td>36</td>
<td></td>
<td>62</td>
<td>3E</td>
<td>76</td>
<td></td>
<td>94</td>
<td>5E</td>
<td>136</td>
<td>L</td>
</tr>
<tr>
<td>31</td>
<td>16</td>
<td>37</td>
<td></td>
<td>63</td>
<td>3F</td>
<td>77</td>
<td></td>
<td>95</td>
<td>5F</td>
<td>137</td>
<td>M</td>
</tr>
</tbody>
</table>
DataIndexedCharMap Pseudocode

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

```java
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 add and contains runtime
  − DataIndexedCharMap takes up a lot of space by storing $R$ links per node
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

isKey = false

isKey = true

key = "a"

key = "up"

isKey = false

isKey = true

key = "a"

key = "up"
Hash Table-based Implementation

- Use Hash Table to find character at a given index

Abstract Trie

Hash Table-based Trie
**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

Abstract Trie

"Internal" left child
(smaller character at same index)

"Trie" child:
advance to next String index

"Internal" right child
(greater character at same index)

Ternary Search Tree (TST)
Which node is associated with the key "CAC"?
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

### Abstract Trie

```
<table>
<thead>
<tr>
<th>[a, u, p]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1</td>
</tr>
</tbody>
</table>
```

### Ternary Search Tree (TST)

```
<table>
<thead>
<tr>
<th>a</th>
<th>u</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Index 0

Index 1

null link
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`