Which of the following is true about Topological Sort and Reductions?

a) Any given graph can be topologically sorted
b) The standard BFS algorithm can be used to topologically sort a graph
c) A reduction is a problem-solving strategy of reducing a problem into smaller chunks
d) Seam carving can be reduced to the BFS algorithm
e) None of the above
Announcements

• **EX4** (MSTs & Sorting) due tonight 11:59 PM PDT
  - Late cutoff Thursday, August 20th

• **P4** (Mazes) due Wednesday 11:59 PM PDT
  - Late cutoff Saturday, August 22nd (day until eternal mastery of CSE 373)

• All extra credit due Saturday, August 22nd

• **EXAM 2** Logistics & Information
  - Released Friday 08/21 12:01 AM PDT, due Saturday 08/22 11:59 PM PDT
  - No submissions accepted after Saturday deadline
  - See Exams page for more detailed logistics and relevant review materials

• Optional Exam II Office Hours during Friday’s lecture
  - For clarifying or logistical questions
  - We’ll also be actively monitoring Piazza for questions
Announcements

• Please fill out course evaluations!
  - We do read your feedback and take it into consideration
  - Aaron and your TAs would be so appreciative!
Learning Objectives

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

1. Identify when a Trie can and should be used, and describe the useful properties a Trie provides

2. Describe and implement the abstract Trie and argue how they are more efficient than using Hash Tables for storing Strings

3. Compare and contrast more advanced Trie designs and explain their differences in runtime and space complexity

4. Implement Trie prefix algorithms and explain how autocomplete algorithms are designed
Lecture Outline

• Tries Introduction
  - When does using a Trie make sense?

• Implementing a Trie using an array
  - How do we find the next child?

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

• Prefix Operations and Autocomplete
  - Find the keys associated with a 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: \( \Theta(l) \)
Lecture Outline

• Tries Introduction
  - When does using a Trie make sense?

• Implementing a Trie using an array
  - How do we find the next child?

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

• Prefix Operations
  - Find keys with a given prefix
**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>&quot;</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>$</td>
<td>36</td>
<td>24</td>
<td>44</td>
<td>$</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>%</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>&amp;</td>
<td>38</td>
<td>26</td>
<td>46</td>
<td>&amp;</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>10</td>
<td>(</td>
<td>40</td>
<td>28</td>
<td>50</td>
<td>)</td>
<td>72</td>
<td>48</td>
<td>108</td>
<td>H</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>11</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>10</td>
<td>20</td>
<td>0</td>
<td>48</td>
<td>30</td>
<td>60</td>
<td>0</td>
<td>80</td>
<td>50</td>
<td>120</td>
<td>P</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>1</td>
<td>81</td>
<td>51</td>
<td>121</td>
<td>Q</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>2</td>
<td>82</td>
<td>52</td>
<td>122</td>
<td>R</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>3</td>
<td>83</td>
<td>53</td>
<td>123</td>
<td>S</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>4</td>
<td>84</td>
<td>54</td>
<td>124</td>
<td>T</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>5</td>
<td>85</td>
<td>55</td>
<td>125</td>
<td>U</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>6</td>
<td>86</td>
<td>56</td>
<td>126</td>
<td>V</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>7</td>
<td>87</td>
<td>57</td>
<td>127</td>
<td>W</td>
</tr>
<tr>
<td>24</td>
<td>18</td>
<td>30</td>
<td>8</td>
<td>56</td>
<td>38</td>
<td>70</td>
<td>8</td>
<td>88</td>
<td>58</td>
<td>130</td>
<td>X</td>
</tr>
<tr>
<td>25</td>
<td>19</td>
<td>31</td>
<td>9</td>
<td>57</td>
<td>39</td>
<td>71</td>
<td>9</td>
<td>89</td>
<td>59</td>
<td>131</td>
<td>Y</td>
</tr>
<tr>
<td>26</td>
<td>1A</td>
<td>32</td>
<td>10</td>
<td>58</td>
<td>3A</td>
<td>72</td>
<td>11</td>
<td>90</td>
<td>5A</td>
<td>132</td>
<td>Z</td>
</tr>
<tr>
<td>27</td>
<td>1B</td>
<td>33</td>
<td>11</td>
<td>59</td>
<td>3B</td>
<td>73</td>
<td>12</td>
<td>91</td>
<td>5B</td>
<td>133</td>
<td>I</td>
</tr>
<tr>
<td>28</td>
<td>1C</td>
<td>34</td>
<td>12</td>
<td>60</td>
<td>3C</td>
<td>74</td>
<td>13</td>
<td>92</td>
<td>5C</td>
<td>134</td>
<td>J</td>
</tr>
<tr>
<td>29</td>
<td>1D</td>
<td>35</td>
<td>13</td>
<td>61</td>
<td>3D</td>
<td>75</td>
<td>14</td>
<td>93</td>
<td>5D</td>
<td>135</td>
<td>K</td>
</tr>
<tr>
<td>30</td>
<td>1E</td>
<td>36</td>
<td>14</td>
<td>62</td>
<td>3E</td>
<td>76</td>
<td>15</td>
<td>94</td>
<td>5E</td>
<td>136</td>
<td>L</td>
</tr>
<tr>
<td>31</td>
<td>1F</td>
<td>37</td>
<td>15</td>
<td>63</td>
<td>3F</td>
<td>77</td>
<td>16</td>
<td>95</td>
<td>5F</td>
<td>137</td>
<td>M</td>
</tr>
</tbody>
</table>

---

**Notes:**
- Dec: Decimal
- Hex: Hexadecimal
- Oct: Octal
- Char: Character
- ASCII Table provides a mapping between decimal, hexadecimal, octal, and character values.
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?
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-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>
</tr>
</thead>
<tbody>
<tr>
<td>Balanced BST</td>
<td>Comparable</td>
<td>$\Theta(\log(n))$</td>
<td>$\Theta(\log(n))$</td>
</tr>
<tr>
<td>Hash Map</td>
<td>Hashable</td>
<td>$\Theta(1)^*$</td>
<td>$\Theta(1)^*$</td>
</tr>
<tr>
<td>Trie (Data-Indexed Array)</td>
<td>String (Character)</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

* In-practice runtime

- **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
  - When does using a Trie make sense?

• Implementing a Trie using an array
  - How do we find the next child?

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

• Prefix Operations
  - Find keys with a given prefix
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

```
Abstract Trie

Index 0
  a
  u
  p

Index 1

Ternary Search Tree (TST)

Index 0
  u

Index 1
  a
  p

“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"?

Tries in COS 226 (Sedgewick, Wayne/Princeton)
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

Keys: [a, u, p]
Index: 0 0 1

Abstract Trie

Ternary Search Tree (TST)
Lecture Outline

• Tries Introduction
  - When does using a Trie make sense?

• Implementing a Trie using an array
  - How do we find the next child?

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

• Prefix Operations
  - Find keys with a given prefix
Prefix Operations with Tries

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

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

- **Longest Prefix From Trie**: given a String, retrieve longest prefix of that String that exists in the Trie
  
  ```java
  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()`)

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

```
List collect() {
    List keys;
    for (char ch : root.next.keys()) {
        collectHelper(ch, keys, root.next.get(ch));
    }
    return keys;
}

void collectHelper(String str, List keys, Node n) {
    if (n.isKey()) {
        keys.add(s);
    }
    for (char ch : n.next.keys()) {
        collectHelper(str + ch, keys, n.next.get(ch));
    }
}
```
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 (char ch : root.next.keySet()) {
        collectHelper(prefix + c, keys, node.next.get(ch));
    }
}

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

One approach to find top 3 matches with prefix "s":

1. Call `keysWithPrefix("s")`  
   
   "say", "smog", "spin", "spine", "spy"

2. Return the 3 keys with highest relevance  
   "spine", "spin", "say"

Q: This algorithm is slow — why? How can we optimize?
Improving Autocomplete with Tries

- **A**: short queries, such as "s", require checking the relevance for billions of matching Strings
  - We only need to keep the top 10

- **Solution**: prune the search space
  - Each node stores its own relevance and maximum relevance of descendants
  - Check that maximum relevance of a subtree is greater than top 10 Strings collected so far before exploring
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`