Overview

1. A Simple Spelling Checker
   Tree Nomenclature

2. Tree Traversal Orders
   Depth-First Traversal Orders
   Breadth-First Traversal Order
A Simple Spelling Checker

• A spelling checker highlights incorrectly spelled words as you type

\textcolor{red}{Thas is an example ov poorl}

• Note the incomplete word in red at the cursor
• Needs to compare against a large dictionary, say $\approx 100k$ words
• Should use minimal computation resources
• How can we store the dictionary for rapid access?
### A Small Sample Dictionary

<table>
<thead>
<tr>
<th>be</th>
<th>begin</th>
<th>weeks</th>
<th>ward</th>
</tr>
</thead>
<tbody>
<tr>
<td>bear</td>
<td>begins</td>
<td>word</td>
<td>warden</td>
</tr>
<tr>
<td>bears</td>
<td>we</td>
<td>words</td>
<td>was</td>
</tr>
<tr>
<td>bee</td>
<td>wear</td>
<td>work</td>
<td>wasp</td>
</tr>
<tr>
<td>bees</td>
<td>wearer</td>
<td>worker</td>
<td>wasps</td>
</tr>
<tr>
<td>beg</td>
<td>week</td>
<td>war</td>
<td>waste</td>
</tr>
</tbody>
</table>

- Many words have prefixes in common
- Want to check for each typed character if the string typed so far is a word
- Use a *rooted tree*
The Dictionary Tree
Dictionary Trees

What numbers does this dictionary tree store?

Pick one (Not Graded)

A: 1, 3, 5
B: 1, 2, 3, 4, 5
C: 1, 13, 25
D: 1, 3, 12, 45
E: 1, 2, 13, 24, 25
The Root

- A tree is *rooted* when one of its nodes is designated as the root
- Any node could be the root, put the root at the top

![Diagram of tree](image)

- [This tree is *binary*: it has a *branching factor* of at most 2]
- [It would be *strictly binary* if the b.f. were always either 2 or 0]
- The same tree, but rooted at node 2 (no longer binary!)

![Diagram of tree](image)

- These are different as *rooted trees*, the same as *free trees*
A Possible Implementation

- Each node in the dictionary tree has one child per letter of the alphabet
- The *depth* of a node is the number of edges between it and the root
- The *branching factor* of each node is the length of the alphabet
- All but two of the children of the root are empty subtrees
- We know the letter from the position among the siblings
- No need to store the letters anywhere
- All nodes look the same, except for which children are empty and whether the node *value* is True or False
What Letter Corresponds to Child n?

- It is convenient to have child 0 for “not a letter”
- Case insensitive: child 1 is either 'a' or 'A'
- Make a “lookup string”
  \[
  \text{alphabet} = ' ' + \text{.join(map(chr, range(ord('a'), ord('z') + 1)))}
  \]
- Does a given number correspond to a letter?
  \[
  \text{def isLetterNumber(n):}
  \]
  \[
  \quad \text{return } 1 \leq n \text{ and } n < \text{len(alphabet)}
  \]
- Convert integers to a lowercase character or the empty string
  \[
  \text{def character(n):}
  \]
  \[
  \quad \text{return alphabet[n] if isLetterNumber(n) } \text{ else ' '}
  \]
- Reverse map uses “ASCII arithmetic”
  \[
  \text{def number(c):}
  \]
  \[
  \quad n = \text{ord(c)} - (\text{ord('A')} \text{ if c.isupper() } \text{ else ord('a')}) + 1
  \]
  \[
  \quad \text{return n if isLetterNumber(n) else 0}
  \]
A Node in Python

class node:
    def __init__(self, isWord=False):
        self.isWord = isWord
        self.child = [None]*len(alphabet)

• Could have more information in a node
• The information in a node is the node’s value
• A tree is just a node with nodes as children
• Since children have children in turn, each child can be viewed as the root of a subtree
• Recursive definition of tree: A tree is a (possibly empty) value and a (possibly empty) list of trees
• The trees in the list are the subtrees rooted at the children
• Once you implement a node, you have the whole tree
The Spelling Checker: General Idea

• Assume that the dictionary tree is built, call it `tree`
• Set `current` to `tree` initially, so `current` points to the root
• If you read a white space
  (one of ' ', ' ', '; ! ? ( ) [ ] & \ - / @ # \n \r '''), reset `current` to `tree`
• [No numbers allowed, for simplicity]
• Otherwise, it’s a letter: descend to the corresponding child of `current`
• Return `current.isWord`
• [Additional code will recolor the current word appropriately]
A Complication

• “Descend to the corresponding child of current”
• What if the child is None?
• Messy solution: check for that condition and handle it appropriately
• More elegant solution:
  Replace None with an infinite subtree with isWord equal to False everywhere
A Very Small Infinite Tree

class terminal:
    def __init__(self, isWord=False):
        self.isWord = isWord
        self.child = [self]*length(alphabet)

Only works if all nodes are identical
Only One Infinite Tree

class terminal:    # Infinite tree
def __init__(self, isWord=False):
    self.isWord = isWord
    self.child = [self]*length(alphabet)

term = terminal()  # Single instantiation

class node:    # Regular node
def __init__(self, isWord=False):
    self.isWord = isWord
    self.child = [term]*length(alphabet)

def isLeaf(n):    # Not used during lookup!
    return all(c is term for c in n.child)
The Spelling Checker

whitespace = ' ,.:;!?()[]\"\'-/@#\n\r''

def isWhitespace(c): return whitespace.find(c) >= 0

current = tree

def check(c):
    global current
    current = tree if isWhitespace(c) \ 
        else current.child[number(c)]
    return current.isWord
Loading the Dictionary

# words is a list of words, perhaps from a file
def makeTree(words):

def insert(word, tree):
    nd = tree
    for letter in word:
        k = number(letter)
        if nd.child[k] is term:
            nd.child[k] = node()
            nd = nd.child[k]
            nd.isWord = True
    nd = node()
    for word in words: insert(word, tree)
    return tree
Tree Traversals

- *Traversing* a tree means visiting each node exactly once
- There are several ways to traverse a tree
  - Depth-first traversal: pre-order and post order
  - Breadth-first traversal
- Depth-first is most naturally done recursively
- Breadth-first is most naturally done iteratively
def printDepthFirst(tree, word=''):  
    if tree.isWord: print(word)  
    for k in range(1, len(tree.child) + 1):  
        if tree.child[k] is not term:  
            printDepthFirst(tree.child[k],\  
                word + character(k))

- **word** behaves like a *stack*:
  - Initially empty
  - *Push* one more character onto **word** when going one level deeper
  - **word** contains the letters on the *path* from the root to the current node
  - *Pop* a character from **word** when returning from printDepthFirst

- What does this print?
Alphabetical Order

This is a lexicographic sorting method
In what order does a depth-first, pre-order traversal of this tree visit the nodes?

Pick one  

A: 1, 2, 3, 4, 5, 6  
B: 1, 2, 4, 3, 5, 6  
C: 1, 2, 4, 5, 6, 3  
D: 4, 2, 1, 5, 6, 3  
E: 6, 5, 3, 4, 2, 1
Depth-First, Post-Order Traversal

```
def printDepthFirst2(tree, word=''):  
    for k in range(1, len(tree.child) + 1):  
        if tree.child[k] is not term:  
            printDepthFirst2(tree.child[k], \ 
                              word + character(k))  
        if tree.isWord: print(word)
```

- Only change: move the print from first to last line
- Prints on its way back up the tree, rather than on its way down
- What does this print?
- How to print in reverse alphabetical order?
In what order does a depth-first, post-order traversal of this tree visit the nodes?

**Pick one**

A: 1, 2, 3, 4, 5, 6  
B: 4, 2, 5, 3, 6, 1  
C: 4, 2, 1, 5, 6, 3  
D: 4, 2, 5, 6, 3, 1  
E: 4, 2, 1, 5, 3, 6
How to Print in Reverse Order?

def printDepthFirst3(tree, word=' '):
    for k in range(len(tree.child) - 1, 0, -1):
        if tree.child[k] is not term:
            printDepthFirst3(tree.child[k], \
            word + character(k))
        if tree.isWord: print(word)

• Do print last
  and visit children in reverse
• Prints on its way back up the tree and from right to left
• This is still a depth-first traversal
How to Print by Increasing Length?

- First print words at depth 1, if any, then at depth 2, ...
- Depth-first had a natural recursive implementation, because you first print the current root, then all the subtrees (smaller problems)
- Not so for printing one level at a time
- Best done iteratively
- Called \textit{breadth-first traversal}
- Idea
  - Put the root (if not terminal) in a queue
  - While the queue is not empty
    - pop the first node from the queue
    - print the corresponding word if appropriate
    - put all the children of that node at the end of the queue
Breadth-First Traversal

```python
def printBreadthFirst(tree):
    if tree is not term:
        queue = []
        queue.append((tree, ''))
    while len(queue) > 0:
        (node, word) = queue.pop(0)
        if node.isWord: print(word)
        for k in range(1, len(node.child)):
            child = node.child[k]
            if child is not term:
                queue.append((node.child[k],
                              word + character(k)))
```

• The queue ensures that any given level is printed first, then the children, left to right

• What does this print?

• Increasing lengths, and alphabetical for each length

be
we
bee
beg
war
was
bear
bees
ward
wasp
wear
week
word
work
bears
begin
wasp
waste
weeks
words
begins
warden
wearer
worker
In what order does a breadth-first traversal of this tree visit the nodes?

Pick one  (Not Graded)

A: 1, 2, 3, 4, 5, 6
B: 1, 2, 4, 3, 5, 6
C: 1, 2, 4, 5, 6, 3
D: 4, 5, 6, 2, 3, 1
E: 6, 5, 3, 4, 2, 1