CSCI 136
Data Structures & Advanced Programming

Lecture 20
Fall 2017
Instructor: Bills
Administrative Details

• Lab 8 is available online
  • No partners this week
  • Review before lab; come to lab with design doc
  • We’ll give an overview shortly
Tonight

**COMPUTER SCIENCE PREREGRISTRATION INFO SESSION**

- Learn about Computer Science courses offered Spring 2017.
- Talk to professors about their classes.
- Meet other Computer Science students.
- Most importantly... **EAT PIZZA!**

**Monday, October 30**
**at 9:00 pm**
**Biology Lounge**
**TBL 211**
Last Time

- Trees
  - Implementation
  - Recursion/Induction on Trees
  - Applications: Decision Trees
Today

• Trees with more than 2 children
  • Representations
  • Application: Lab 8: Lexicon!

• Binary Trees
  • Traversals
    • As methods taking a BinaryTree parameter
    • With Iterators
Representing Arbitrary Trees

• What if nodes can have many children?
  • Example: Game trees

• Replace left/right node references with a list of children (Vector, SLL, etc)
  • Allows getting “i^{th}” child

• Should provide method for getting degree of a node

• Degree 0 ↔ Empty list ↔ No children ↔ Leaf
**Lab 8 : Lexicon**

- **Goal:** Build a data structure that can efficiently store and search a large set of words
- **A special kind of tree called a trie**

![Trie Diagram]
Lab 8 : Tries

• A trie is a tree that stores words where
  • Each node holds a letter
  • Some nodes are “word” nodes (dark circles)
  • Any path from the root to a word node describes one of the stored words
  • All paths from the root form prefixes of stored words (a word is considered a prefix of itself)
Even your program-final standard classes in our toolkit, such as a sized array which can grow and shrink as needed, a linked list of children pointers, or index. However, for most nodes within the trie, it's trivial to find the child for a given letter, you simply access for a given letter, (such as from Z to X) the array entry would be NULL. Array[0] is the child for A, array[1] refers to B, ... and array[25] refers to Z. When storing above, for the root to this node represents a word. Here's a conceptual picture of a small trie:

There are several different data structures you could choose from for managing the trie, including themselves smaller tries. You will be making assumption that there doesn't exist, so strings such as are, as, new, no, not, because strings like "a", "ab", "abc", "abcd", and so on, share that initial part of their paths. This saves space since there just those words that begin with the prefix, with a left or right turn, a search in a trie follows the letter for the search. Instead of just two children as in a binary tree, each node in the trie traces out a sequence of letters that represent a prefix or word in the lexicon.

For ease of writing and debugging the code, performance of add/remove, and so on. The root node indicates a word. Here's a conceptual picture of a small trie: Each entry in the array is a pointer to a child. The array is a static-sized 26-member array, indexed by the letter from A to Z. From the root node, one child has been taken: the child for A. From there, node A has children R, S, and E. From R, the child for E, W, and T are linked. From S, only the child for E is linked, while the child for E is the last child and thus has none. From E, the child for N is linked. The node for N has all its children: E, Z, O, and E. This is a trie for the words: "A", "R", "S", "E", "W", "O", "T", "E", "N", "D", "T", and "E", "N". Now add "dot" and "news".
The first step to removing a word is tracing out its path and turning off the

Removing common prefix nodes with words already in the trie.

A trie is an

is the sample trie after those three words have been added:

Adding the words doesn’t require any new nodes at all,

will add three new nodes, one for each letter. On

Adding the

necessitate adding a new node

the

searching. If any part of the path does not exist, the missing nodes must be added to

Adding

Therefore determining whether a string is a prefix of at least one

All

children leading from there, so the path for this string does not exist

prefix of other

isWord

Alternatively,

Searching

Searching

with our previous trie. This trie contains the words “not” and “zen”, even though

nor a prefix in this trie.

isWord

the

match

Searching

isWord

Searching

isWord

the

match

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord

Searching

isWord
Tries

A dead end. All paths in the trie must eventually lead to a word. If the word being removed was the only valid word along this path, the nodes along that path must be deleted from the trie along with the word. For example, if you removed the words zen and not from the trie shown previously, you should have the result below.

```
Start
  A
  | R
  S
  | O
  D
  | N
  E
  O
  T
  W
  S
```

As a general observation, there should never be a leaf node whose isWord field is false. If a node has no children and does not represent a valid word (i.e., isWord is false), then this node is not part of any path to a valid word in the trie and such nodes should be deleted when removing a word.

In some cases, removing a word from the trie may not require removing any nodes. For example, if we were to remove the word new from the above trie, it turns off isWord but all nodes along that path are still in use for other words.

Important note: when removing a word from the trie, the only nodes that may require deallocation are nodes on the path to the word that was removed. It would be extremely inefficient if you were to traverse the whole trie to check for deallocating nodes every time a word was removed, and you should not use such an inefficient strategy.

Other trie operations There are few remaining odds and ends to the trie implementation. Creating an iterator and writing the words to a file both involve a recursive exploration of all paths through the trie to find all of the contained words. Remember that in both cases it is only words (not prefixes) that you want to operate on and that these operations need to access the words in alphabetical order.

Once you have a working lexicon, you're ready to implement the snazzy spelling correction features. There are two additional Lexicon member functions, one for suggesting simple corrections and the second for regular expressions matching:

- `Set<string> *SuggestCorrections(string target, int maxDistance);`
- `Set<string> *MatchRegex(string pattern);`

Suggesting corrections First consider the member function `SuggestCorrections`. Given a (potentially misspelled) target string and a maximum distance, this function gathers the set of words from the lexicon that have a distance to the target string less than or equal to the given maxDistance. We define the distance between two equal-length strings to be the total number of character positions in which the strings differ. For example, "place" and "peace" have distance 1, "place" and "plank" have distance 2. The returned set contains all words in the lexicon that are the same length as the target string and are within the maximum distance.
Lab 8 : Lexicon

An interface that provides the methods

```java
public interface Lexicon {
    public boolean addWord(String word);
    public int addWordsFromFile(String filename);
    public boolean removeWord(String word);
    public int numWords();
    public boolean containsWord(String word);
    public boolean containsPrefix(String prefix);
    public Iterator<String> iterator();
    public Set<String> suggestCorrections(String target, int maxDistance);
    public Set<String> matchRegex(String pattern);
}
```
Lab 8

• Implement a program that creates, updates, and searches a Lexicon
  • Based on a LexiconTrie class
    • Each node of the Trie is a LexiconNode
    • Analogous to a SLL consisting of SLLNodes
  • LexiconTrie implements the Lexicon Interface
  • Supports
    • adding/removing words
    • searching for words and prefixes
    • reading words from files
    • Iterating over all words
Tree Traversals

• In linear structures, there are only a few basic ways to traverse the data structure
  • Start at one end and visit each element
  • Start at the other end and visit each element
• How do we traverse binary trees?
  • (At least) four reasonable mechanisms
Tree Traversals

In-order: Aria, Jacob, Kelsie, Lucas, Nambi, Tongyu
Pre-order: Lucas, Jacob, Aria, Kelsie, Nambi, Tongyu
Post-order: Aria, Kelsie, Jacob, Tongyu, Nambi, Lucas,
Level-order: Lucas, Jacob, Nambi, Aria, Kelsie, Tongyu
Tree Traversals

• Pre-order
  • Each node is visited before any children. Visit node, then each node in left subtree, then each node in right subtree. (node, left, right)
  • +*237

• In-order
  • Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree. (left, node, right)
  • 2*3+7

(“pseudocode”)
Tree Traversals

- Post-order
  - Each node is visited after its children are visited. Visit all nodes in left subtree, then all nodes in right subtree, then node itself. (left, right, node)
  - 23*7+

- Level-order (not obviously recursive!)
  - All nodes of level i are visited before nodes of level i+1. (visit nodes left to right on each level)
  - +*723

("pseudocode")
public void pre-order(BinaryTree t) {
    if(t.isEmpty()) return;
    touch(t); // some method
    preOrder(t.left());
    preOrder(t.right());
}

For in-order and post-order: just move touch(t)!

But what about level-order???
Level-Order Traversal

- Green
  - Blue
  - Violet
    - Orange
    - Yellow
      - Indigo
      - Red
Level-Order Traversal

Green

Blue  Violet

Orange  Yellow

Indigo  Red

G
Level-Order Traversal

Green
  Blue
  Violet
  Orange
  Yellow
  Indigo
  Red

G B
Level-Order Traversal

G B V
Level-Order Traversal

G B V O
Level-Order Traversal

G B V O Y I
Level-Order Traversal

G B V O Y I R
Level-Order Traversal

```
  Green
   /   \
  Blue  Violet
        /    /
   Orange  Yellow
          /    /
      Indigo  Red
```
Level-Order Traversal

Green

Blue

Violet

Orange  Yellow

Indigo  Red

todo queue
Level-Order Traversal

- Green
  - Blue
  - Violet
    - Orange
    - Yellow
      - Indigo
      - Red

Violet
Blue
todo queue

G
Level-Order Traversal

Green
  Blue
    violet
  Orange
    Indigo
    Red

Yellow

Violet

todo queue
Level-Order Traversal

G B V

Green
  Blue
  Violet
    Orange  Yellow
    Indigo  Red

Yellow
  Orange
todo queue
Level-Order Traversal

G B V O
Level-Order Traversal

G B V O Y
Level-Order Traversal

- Green
  - Blue
  - Violet
    - Orange
    - Yellow
      - Indigo
      - Red

Red

todo queue

G B V O Y I
Level-Order Traversal
public static <E> void levelOrder(BinaryTree<E> t) {
    if (t.isEmpty()) return;

    // The queue holds nodes for in-order processing
    Queue<BinaryTree<E>> q = new QueueList<BinaryTree<E>>();
    q.enqueue(t); // put root of tree in queue

    while(!q.isEmpty()) {
        BinaryTree<E> next = q.dequeue();
        touch(next);
        if(!next.left().isEmpty() ) q.enqueue( next.left() );
        if(!next.right().isEmpty() ) q.enqueue(next.right());
    }
}
Iterators

• Provide iterators that implement the different tree traversal algorithms

• Methods provided by BinaryTree class:
  • preorderIterator()
  • inorderIterator()
  • postorderIterator()
  • levelorderIterator()
Implementing the Iterators

- **Basic idea**
  - Should return elements in same order as corresponding traversal method shown
  - Recursive methods don’t convert as easily: must phrase in terms of next() and hasNext()
  - So, let’s start with levelOrder!
public BTLevelorderIterator(BinaryTree<E> root) {
    todo = new QueueList<BinaryTree<E>>();
    this.root = root; // needed for reset
    reset();
}

public void reset() {
    todo.clear();
    // empty queue, add root
    if (!root.isEmpty()) todo.enqueue(root);
}
public boolean hasNext() {
    return !todo.isEmpty();
}

public E next() {
    BinaryTree<E> current = todo.dequeue();
    E result = current.value();
    if (!current.left().isEmpty())
        todo.enqueue(current.left());
    if (!current.right().isEmpty())
        todo.enqueue(current.right());
    return result;
}
Pre-Order Iterator

• Basic idea
  • Should return elements in same order as processed by pre-order traversal method
  • Must phrase in terms of `next()` and `hasNext()`
  • We “simulate recursion” with stack
    • The stack holds “partially processed” nodes
Pre-Order Iterator

• Outline: node - left tree – right tree

1. Constructor: Push root onto todo stack
2. On call to next():
   • Pop node from stack
   • Push right and then left nodes of popped node onto stack
   • Return node’s value
3. On call to hasNext():
   • return !stack.isEmpty()
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.

- Green
  - Blue
  - Violet
    - Orange
    - Yellow
      - Indigo
      - Red
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.

```
Green
  Blue
  Violet
    Orange
    Yellow
      Indigo
      Red
```

todo stack
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.

G B
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.

G B V
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.

G B V O
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.

G B V O I
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.

G B V O I R
Pre-Order Iterator

Visit node, then each node in left subtree, then each node in right subtree.

```
G B V O I R Y
green
  blue
  violet
    orange
    yellow
      indigo
      red
todo stack
```
public BTPreorderIterator(BinaryTree<E> root)
{
    todo = new StackList<BinaryTree<E>>();
    this.root = root;
    reset();
}

public void reset()
{
    todo.clear(); // stack is empty; push on root
    if (!root.isEmpty()) todo.push(root);
}
Pre-Order Iterator

```java
public boolean hasNext() {
    return !todo.isEmpty();
}

public E next() {
    BinaryTree<E> old = todo.pop();
    E result = old.value();

    if (!old.right().isEmpty())
        todo.push(old.right());
    if (!old.left().isEmpty())
        todo.push(old.left());
    return result;
}
```
Tree Traversal Practice Problems

• Prove that levelOrder() is correct: that is, that it touches the nodes of the tree in the correct order (Hint: induction by level)
• Prove that levelOrder() takes $O(n)$ time, where $n$ is the size of the tree
• Prove that the PreOrder (LevelOrder) Iterator visits the nodes in the same order as the PreOrder (LevelOrder) traversal method
In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.
In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.

The diagram shows a tree structure with nodes labeled from light to dark: Blue, Green, Violet, Orange, Yellow, Indigo, Red. The order of visiting the nodes is Blue, Green, Orange, Yellow, Indigo, Red.
In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.
In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.
In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.
In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.
In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.

B G I O R

Violet

todo stack
In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.
In-Order Iterator

Each node is visited after all nodes in left subtree are visited and before any nodes in right subtree.

B G I O R V Y
In-Order Iterator

• Outline: left - node - right

  1. Push left children (as far as possible) onto stack

  2. On call to next():
     • Pop node from stack
     • Push right child and follow left children as far as possible
     • Return node’s value

  3. On call to hasNext():
     • return !stack.isEmpty()
Post-Order Iterator

- Left as an exercise...
Alternative Tree Representations

- Total # “slots” = 4n
  - Since each BinaryTree maintains a reference to left, right, parent, value
- 2-4x more overhead than vector, SLL, array, …
- But trees capture successor and predecessor relationships that other data structures don’t…
Array-Based Binary Trees

• Encode structure of tree in array indexes
  • Put root at index 0

• Where are children of node i?
  • Children of node i are at $2i+1$ and $2i+2$
  • Look at example

• Where is parent of node j?
  • Parent of node j is at $(j-1)/2$
ArrayTree Tradeoffs

• Why are ArrayTrees good?
  • Save space for links
  • No need for additional memory allocated/garbage collected
  • Works well for full or complete trees
    • Complete: All levels except last are full and all gaps are at right
    • “A complete binary tree of height h is a full binary tree with 0 or more of the rightmost leaves of level h removed”

• Why bad?
  • Could waste a lot of space
  • Tree of height of n requires $2^{n+1}-1$ array slots even if only $O(n)$ elements