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
Last Time

- Trees
  - Implementation
  - Recursion/Induction on Trees
Today

• Application: Decision Trees
• Some Loose Ends
• Trees with more than 2 children
  • Representations
  • Application: Lab 8: Lexicon!
• Binary Trees
  • Traversals
    • As methods taking a BinaryTree parameter
    • With Iterators
Tonight

**COMPUTER SCIENCE Preregistration 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
Representing Knowledge

• Trees can be used to represent knowledge
  • Example: InfiniteQuestions game

• We often call these trees decision trees
  • Leaf: object
  • Internal node: question to distinguish objects

• Move down decision tree until we reach a leaf node

• Check to see if the leaf is correct
  • If not, add another question, make new and old objects children

• Let’s look at the code…
Building Decision Trees

- Gather/obtain data
- Analyze data
  - Make greedy choices: Find good questions that divide data into halves (or as close as possible)
- Construct tree with shortest height
- In general this is a *hard* problem!
- Example
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\text{th}” child

• Should provide method for getting degree of a node

• Degree 0 \leftrightarrow \text{Empty list} \leftrightarrow \text{No children} \leftrightarrow \text{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

A trie is a letter-tree that efficiently stores strings. A node in a trie represents a letter. A path from the root to a node indicates a prefix or word in the lexicon. A trie is a shared static-sized array of children pointers, where each child pointer can be NULL or point to another trie node. This allows nodes to be dynamically added and deleted in constant time. Additional data can be stored in each node, such as a boolean flag to indicate whether the node is a complete word or a prefix. A word is identified by traversing to a leaf node and checking the boolean flag. For example, a trie for the words as, new, no, not, and thick would have a leaf node for each word.

A trie can be implemented using a linked list, a binary search tree, a hashtable, and many others. Each of these offers tradeoffs in ease of writing and debugging the code, performance of add/remove, and so on. The choice depends on the specific requirements and constraints of the application.

For a given letter, you can simply access the array entry corresponding to that letter. For example, in a trie for the English alphabet, array[0] is the child for A, array[1] refers to B, ..., and array[25] refers to Z. When searching for a word, you start at the root and follow the path indicated by the letters in the word. If you reach a leaf node and the boolean flag is true, then the word is found. Otherwise, if any path is missing, the word is not in the trie.

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.

A trie is a letter-tree that efficiently stores strings. A node in a trie represents a letter. A path from the root to a node indicates a prefix or word in the lexicon. A trie is a shared static-sized array of children pointers, where each child pointer can be NULL or point to another trie node. This allows nodes to be dynamically added and deleted in constant time. Additional data can be stored in each node, such as a boolean flag to indicate whether the node is a complete word or a prefix. A word is identified by traversing to a leaf node and checking the boolean flag. For example, a trie for the words as, new, no, not, and thick would have a leaf node for each word.

A trie can be implemented using a linked list, a binary search tree, a hashtable, and many others. Each of these offers tradeoffs in ease of writing and debugging the code, performance of add/remove, and so on. The choice depends on the specific requirements and constraints of the application.

For a given letter, you can simply access the array entry corresponding to that letter. For example, in a trie for the English alphabet, array[0] is the child for A, array[1] refers to B, ..., and array[25] refers to Z. When searching for a word, you start at the root and follow the path indicated by the letters in the word. If you reach a leaf node and the boolean flag is true, then the word is found. Otherwise, if any path is missing, the word is not in the trie.
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)
Tries

Now add “dot” and “news”
Now remove “not” and “zen”
Tries

Now 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

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

- 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: “left, node, right”
   Aria, Jacob, Kelsie, Lucas, Nambi, Tongyu
Pre-order: “node, left, right”
   Lucas, Jacob, Aria, Kelsie, Nambi, Tongyu
Post-order: “left, right, node”
   Aria, Kelsie, Jacob, Tongyu, Nambi, Lucas,
Level-order: visit all nodes at depth i before depth i+1
   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)
  - $23 \times 7 +$

- **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)
  - $+ \times 723$

("pseudocode")
public void pre-order(BinaryTree t) {
    if (!t.isEmpty()) return;
    visit(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
Level-Order Traversal

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

G
Level-Order Traversal

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

G B
Level-Order Traversal

G B V
Level-Order Traversal

G B V O
Level-Order Traversal

G B V O Y
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

- Green
Level-Order Traversal

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

- todo queue
  - Violet
  - Blue

G
Level-Order Traversal

G B

Green
  Blue
    Violet
    Orange
      Indigo
      Red
    Yellow

todo queue
Level-Order Traversal

G B V

Green

Blue

Violet

Orange Yellow

Indigo Red

Todo queue

Yellow

Orange
Level-Order Traversal
Level-Order Traversal
Level-Order Traversal

G B V O Y I

Red

todo queue
Level-Order Traversal

G B V O Y I R
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();
        visit(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()
    • Similar to how we implemented SkipIterator: do some prep work before returning from next()
  • 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 children 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.
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.

(todo stack)
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
Pre-Order Iterator

class BTPreorderIterator {
    public BTPreorderIterator(BinaryTree<E> root) {
        todo = new StackList<BinaryTree<BinaryTree<E>>>(root);
        this.root = root;
        reset();
    }

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

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

• 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()
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.
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 V Y
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