Last Time

• Trees
  • Expression Trees
    • Recursive evaluation
  • Implementation
Today

• Recursion/Induction on Trees
• Applications: Decision Trees
• Trees with more than 2 children
  • Representations
• Traversing Binary Trees
  • As methods taking a BinaryTree parameter
  • With Iterators
• Prove
  • The number of nodes at depth $n$ is at most $2^n$.
  • The number of nodes in tree of height $n$ is at most $2^{(n+1)}-1$.
  • A tree with $n$ nodes has exactly $n-1$ edges
  • The size() method works correctly
  • The height() method works correctly
  • The isFull() method works correctly
Prove: Number of nodes at depth \( d \geq 0 \) is at most \( 2^d \).

Idea: Induction on depth \( d \) of nodes of tree

Base case: \( d = 0 \): 1 node. \( 1 = 2^0 \) √

Induction Hyp.: For some \( d \geq 0 \), there are at most \( 2^d \) nodes at depth \( d \).

Induction Step: Consider depth \( d+1 \). There are at most 2 nodes at depth \( d+1 \) for every node at depth \( d \).
Therefore it has at most \( 2 \times 2^d = 2^{d+1} \) nodes √
Prove that any tree on \( n \geq 1 \) nodes has \( n-1 \) edges

Idea: Induction on number of nodes

Base case: \( n = 1 \). There are no edges

Induction Hyp: Assume that, for some \( n \geq 1 \), every tree on \( n \) nodes has exactly \( n-1 \) edges.

Induction Step: Let \( T \) have \( n+1 \) nodes. Show it has exactly \( n \) edges.

- Remove a leaf \( v \) (and its single edge) from \( T \)
- Now \( T \) has \( n \) nodes, so it has \( n-1 \) edges
- Now add \( v \) (and its single edge) back, giving \( n+1 \) nodes and \( n \) edges.
BT Questions/Proofs

Prove that BinaryTree method size() is correct.

- Let n be the number of nodes in the tree T
- Alert: Strong Induction Ahead...

Base case: n = 0. T is empty---size() returns 0✓

Induction Hyp: Assume size() is correct for all trees having at most n nodes.

Induction Step: Assume T has n+1 nodes

- Then left/right subtrees each have at most n nodes
- So size() returns correct value for each subtree
- And the size of T is 1 + size of left subtree + size of right subtree✓
Representing Knowledge

• Trees can be used to represent knowledge
  • Example: InfiniteQuestions game
    • Let’s play!

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

• Two methods: play() and learn()
  • Play: Move down decision tree until we reach a leaf
    • Check to see if the leaf is correct
  • Learn: If not correct, add 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

---

Fruit images: banana, lemon, orange, green beans, plum, grapefruit (yellow)
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 Empty list No children Leaf
Lab 9 Preview: 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
Lab 9 Preview : 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”
node. However, your work is not yet done because you need to find a common prefix node with words already in the trie.

Instead, consider the sample trie after those three words have been added:

![Sample Trie Diagram]

Adding the words "not" and "zen" doesn't require any new nodes at all, so the trie stays the same. Now remove "not" and "zen".

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.

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.
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)
  • $23 \times 7 +$

• 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)
  • $\ast 723$

(“pseudocode”)
Tree Traversals

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???