Trie

Remember about Binary Search Tree (BST)

- We have seen binary trees, where each node has at most two children

```
Struct node{
        int data;
       struct node *left, *right;
};
```

- If you need to store strings in your binary search tree, the node structure would look like this:

```
Struct node{
        char name[100];
       struct node *left, *right;
};
```

- Can you think of an application scenario of storing string in a BST?
  - Dictionaries, spell checking

Storing words

- We can use BST for dictionary by supporting insertion and look-up
    - Also as you are working on your lab submission, you can insert items and lookup items from the BST.
    - Can you guess what kind of weakness it has?
    - Weakness: there are lot of string comparison happens while traversing throughout the tree (both in insertion and search). Depending on the length of the strings in each node, it can be time consuming.
- We could also use binary search:
    - By storing the words into an array and then sort it with an O(n long n) sorting algorithm
    - Then use binary search
    - The problem is , we will need to add words while program is running and we will need to expand our array. It can be time consuming too.
    - Adding words can be frequent operation like people add words in their web browser and cell phone spell checkers all the time

Storing words

- If we want to know how many times a word occurs in a corpus (a body of text), how would you implement it using BST?
    - Maybe inserting words into a BST multiple times, which would take up extra space and also require us to write a slower function to determine how many times a word occurs in the BST
    - we could modify our BST node struct to have a word count field

```
Struct node{
        char word[100];
        int count
       struct node *left, *right;
};
```

Storing words

- All the approaches mentioned so far suffer tremendously if we have a dictionary full of fake words that share a long prefix.
- For example, suppose I have a dictionary with 200 million words that all begin with "spaghettiHasNothingToDoWithAnything", such as,
    - "spaghettiHasNothingToDoWithAnythingA",
    - "spaghettiHasNothingToDoWithAnythingB",
    - "spaghettiHasNothingToDoWithAnythingAndThisCouldGoOnForever", and so on.)
    - Suddenly, looking up those strings is a bit more expensive, because we have to go through 35 characters every single time we perform a comparison.
    - That wasn't the case when we were dealing with arrays of integers, and we often sweep that issue under the rug when talking about runtime, because we assume string length is bounded by some sort of reasonable constant, but it's a good thing to keep in mind, because it does have a real impact on runtime, especially if you're processing a ton of data.

Trie

- Trie is another data structure
- Came from the word *re**trie**val*. Pronounced as “tree” . We pronounce it as “try” to distinguish it from the tree data structure.
- There are two super amazing characteristics of Tries:
- First:
  - A trie is a tree in which every node has 26 children (it means 26 child pointers!).
  - Some or all pointers maybe NULL out of this 26 pointers.
  - Why 26? Because we have 26 letters (a to z)
  - Each pointer is for each character (starting from 0 to 25). We will see more in next slides.

```
Struct TrieNode{
        int count;
       struct TriNode *children[26]; //an array of 26 TrieNode pointers
};
```

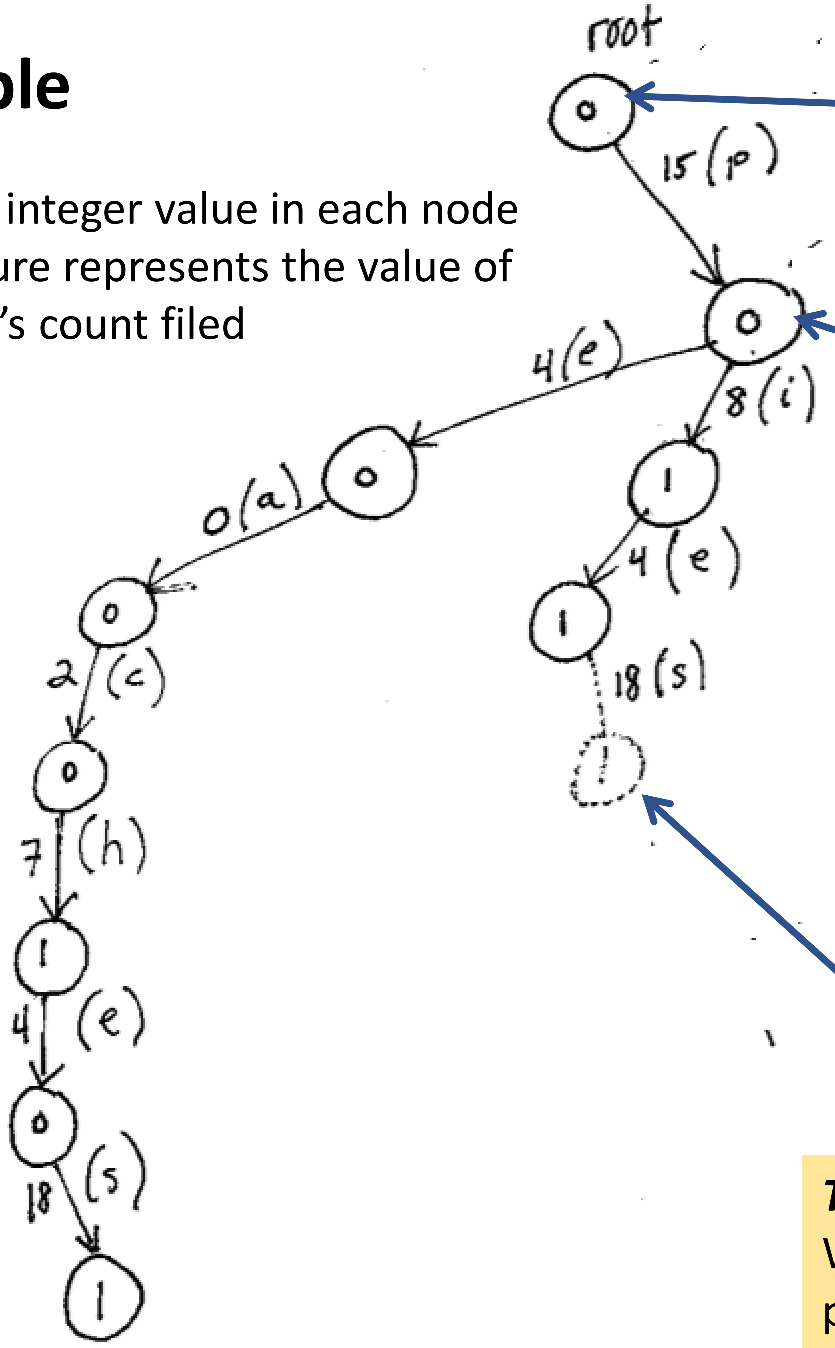
Trie

- Secondly:
  - Although we will use tries to represent strings, the strings we insert into a trie are not stored as data <u>inside</u> our nodes.
  - Instead, the string that a node represents is based on the path you take to reach that node.

- Example will clarify.

Example

Note: the integer value in each node in this figure represents the value of the struct's count filed



If the trie contained the empty string (""), the value here would be non-zero

This node is at ***root->children[15].*** It actually represents "p" because 'p' is the 16th letter of the alphabet.
# if you start counting form zero (for letter 'a'), then 'p' is integer #15.

Note: every node has many children that are not shown in this figure for the sake of clarity and simplicity. We assume those children are NULL.

Insertion of "pies"

This trie represents:
We have words:
pi 1 time, pie 1 time, pies one time, peach 1 time, peaches 1 time

Run time

- The big-oh runtime for ________ into a trie is:

Operation	Worst	Best
Insertion	O(k)	O(k)
Look-up	O(k)	O(1)
Deletion	O(k)	O(1)

- Where K us the length of the string we are inserting/deleting/looking

Applications

- Dictionary
- Spell check
- Document word count/ word frequency
- Find all words beginning with same prefix
- Word prediction for typing/texting/voice recognition (speech to text processing)

Accessing the count of a word

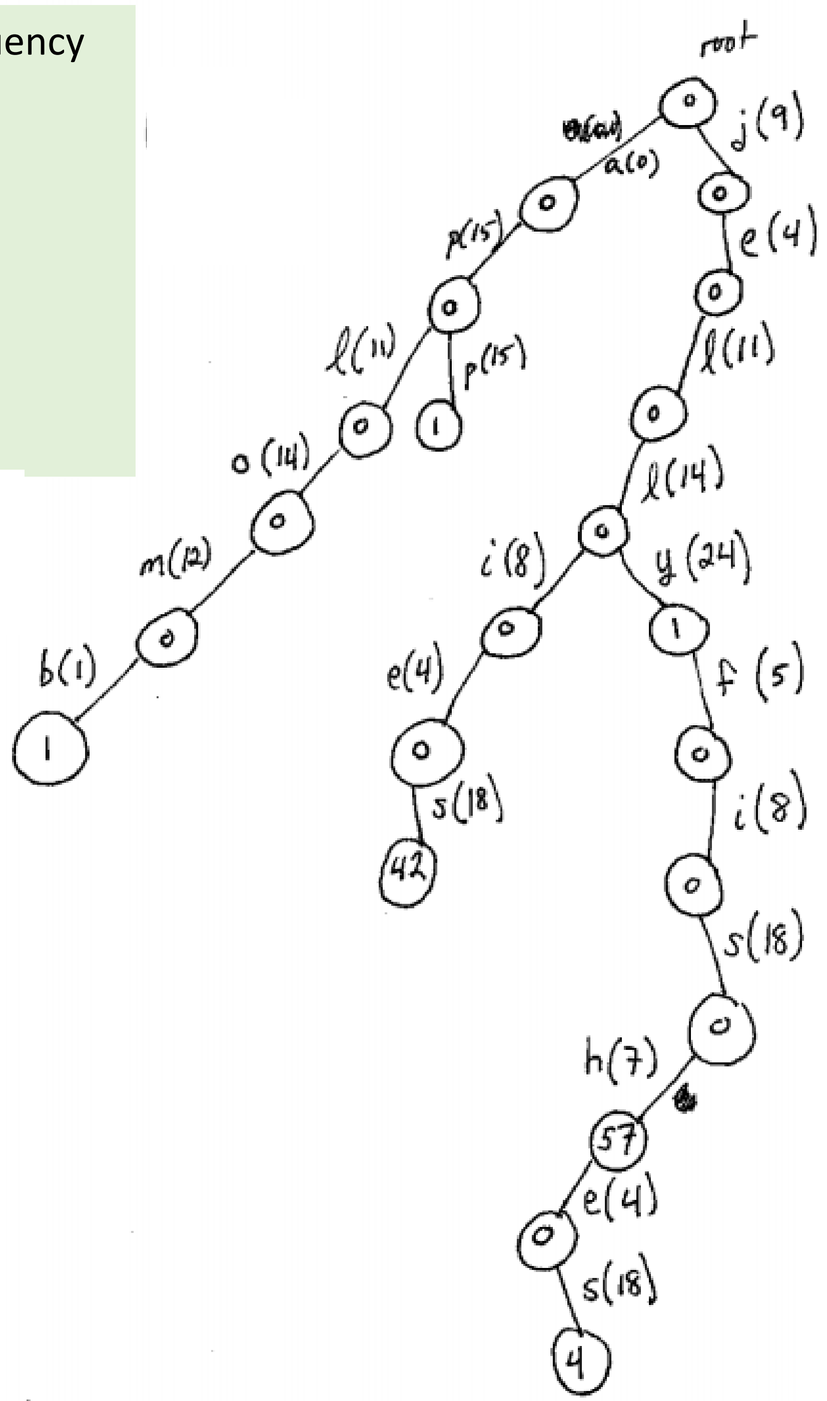
- If the word “apple” has been inserted into your trie, how can you access its count field.
- The letter:
  - ‘a’ corresponds to index 0 in the children arrays.
  - ‘p’ corresponds to index 15.
  - ‘l’ corresponds to index 11,
  - and ‘e’ corresponds to index 4.
- So, we could access that field like:
  - *root->children[0]->children[15]->children[15]->children[11]->children[4]->count*
  - Note that, we assumes “apple” is in the tree, so we don’t have to worry about segfaults.
- What if we don’t know the index of ’p’ off the top of our heads, though? How can you generate an index from a character?
- ‘p’ – ‘a’ can actually give you the index of ‘p’
- *root->children[‘a’-’a’]->children[‘p’-’a’]->children[‘p’-’a’]->children[‘l’-’a’]->children[‘e’-’a’]->count*
- More generally, if a char variable is c, we can get its index by c – ‘a’. *Thus, ->children[c-’a’]*

Deleting Strings from Trie

Suppose we have a word frequency trie with the following words:

jelly (x 1)
jellies (x 42)
jellyfish (x 57)
jellyfishes(x 4)
aplomp (x 1)
app (x1)

- To delete "aplomb":
  - We can decrement the count at its terminal node to zero,
  - Then delete all the nodes up to (and including) the 'l' since they have no children.
- To delete one occurrence of "jelly", we simply decrement the count value at its terminal node
  - We cannot delete its terminal node even though its count is now zero. Why?
- To delete one occurrence of "jellyfishes", we decrement the count filed at its terminal node, but no nodes get deleted (why not?)



Reference

- Sean's Trie Notes
- CMU Notes on Tries
- **Code link (we will go though it in the class):** http://www.cs.ucf.edu/~dmarino/ucf/transparency/cop3502/sampleprogs/mytrie.c
- More code: http://www.cs.ucf.edu/~dmarino/ucf/transparency/cop3502/sampleprogs/mytrie2.c