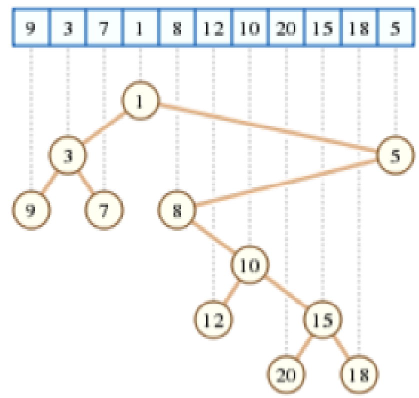
CS106X Handout 30
Autumn 2019 November 4th, 2019

Section Handout

Problem 1: Cartesian Trees



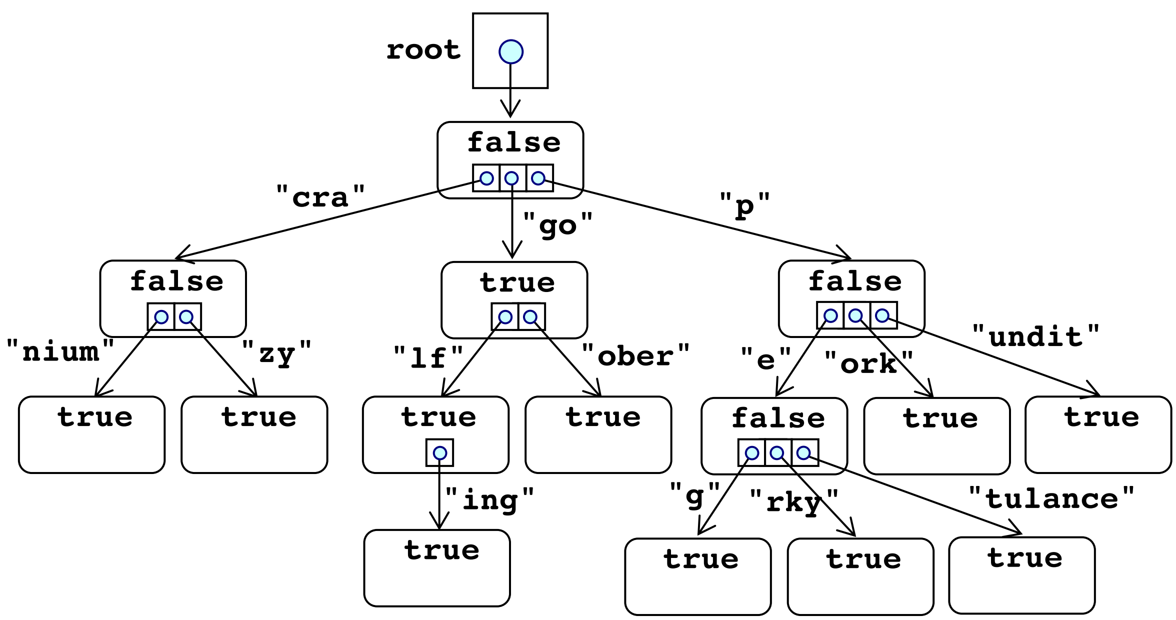
A Cartesian tree is a binary tree structure derived from an array of numbers such that the tree respects the min-heap property (value at the parent is less than the values of the two children) and an inorder traversal of the tree produces the original array sequence. The picture presented on the right (credit: Wikipedia) illustrates how an integer array and the corresponding Cartesian tree are related.

Write a function called **`arrayToCartesianTree`**, which accepts a reference to a constant **`Vector<int>`** of **unique** positive integers, synthesizes the corresponding Cartesian tree, and returns its root. The problem relies on the existence of the following type definition:

```
struct node {
   int value;
   node *left, *right;
};

static node *arrayToCartestianTree(const Vector<int>& inorder);
```

Problem 2: Patricia Trees

Consider the following illustration:



What's drawn above is an example of a **Patricia tree**—similar to a trie in that each node represents some prefix in a set of words. The child pointers, however, are more elaborate, in that they not only identify the sub-tree of interest, but they carry the substring of characters that should contribute to the running prefix along the way. Sibling pointers aren't allowed to carry substrings that have common prefixes, because the tree could be restructured so that the common prefix is merged into its own connection. By imposing that constraint, that means there's at most one path that needs to be explored when searching for any given word.

The children are lexicographically sorted, so that all strings can be easily reconstructed in alphabetical order. When a node contains a **`true`**, it means that the prefix it represents is also a word in the set of words being represented. [The root of the tree always represents the empty string.]

So, the tree above stores the following words:

> *cranium*, *crazy*, *go*, *golf*, *golfing*, *goober*, *peg*, *perky*, *petulance*, *pork*, and *pundit*.

These two type definitions can be used to manage such a tree.

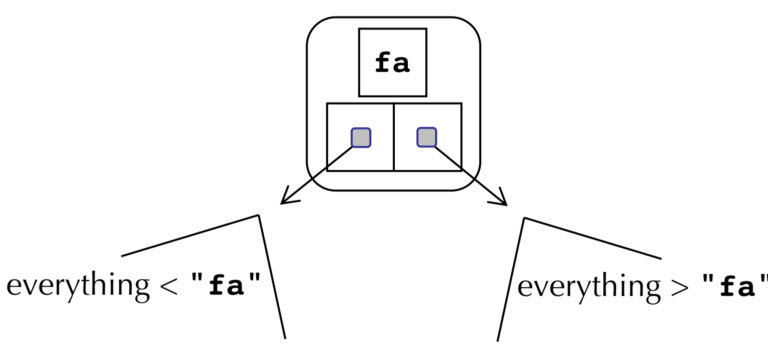
```
struct connection {
   string letters;
   struct node *subtree; // will never be NULL
};

struct node {
   bool isWord;
   Vector<connection> children;  // empty if no children
};
```

Implement the **`containsWord`** function, which accepts the root of a Patricia tree and a word, and returns **`true`** if and only if the supplied word is present. Even though the **`connection`**s descending from each node are sorted alphabetically, you should just do a **linear search** across them to see which one, if any, is relevant.

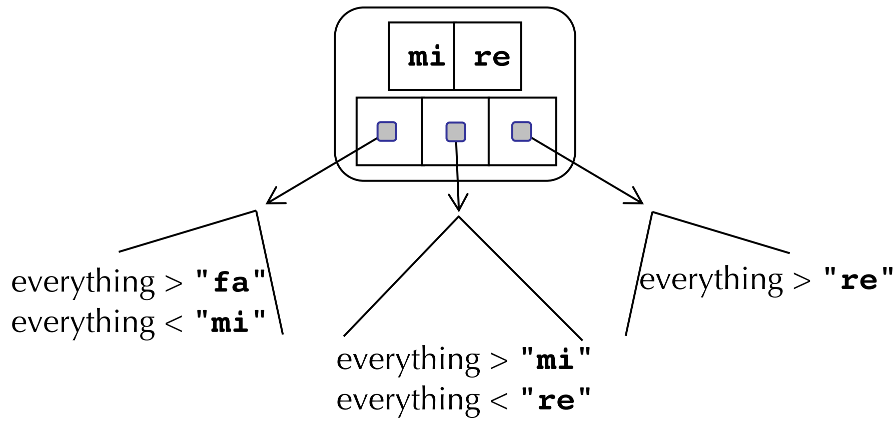
```
static bool containsWord(const node *root, const string& word);
```

Problem 3: Exponential Trees

Exponential trees are similar to binary search trees, except that the **depth** of the node in the tree dictates how many elements it can store. The root of the tree is at depth 1, so it contains 1 element and two children. The root of a tree storing strings might look like this:



If completely full, a node at depth 2—perhaps the right child of the above root node—might look like this:



Generally speaking, a node at depth **d** can accommodate up to **d** elements. Those **d** elements are stored in sorted order within a **Vector<string>**, and they also serve to distribute all child elements across the **d + 1** sub-trees.

We can use the following data structure to build up and manage an exponential tree:

```
struct expnode {
   int depth;              // depth of the node within the tree
   Vector<string> values;  // stores up to depth keys in sorted order
   expnode **children;     // set to NULL until node is saturated.
};
```

- Each node must keep track of its **depth**, because the depth alone decides how many elements it can hold, and how many sub-trees it can support.
- The string values are stored in the **values** vector, which maintains all of the strings it's storing in sorted order. We use a **Vector<string>** instead of an exposed array, because the number of elements stored can vary from **0** to **depth**.
- **children** is a dynamically allocated array of pointers to sub-trees. The **children** pointer is maintained to be **NULL** until the **values** vector is full, at which point the **children** pointer is set to be a dynamically allocated array of **depth + 1** pointers, all initially set to **NULL**. Any future insertions that pass through the node will actually result in an insertion into one of **depth + 1** sub-trees.

a. Draw the exponential tree that results from inserting the following strings in the specified left-to-right order:

"do" "re" "mi" "fa" "so" "la" "ti"

b. Implement the **expTreeContains** predicate function, which given the root of an exponential tree and a string, returns **true** if and only if the supplied string is present somewhere in the tree, and **false** otherwise. Your function should only visit nodes that lead to the string of interest. Your implementation can rely on the implementation of **find**, which accepts a sorted string vector and a new string **value** and returns the smallest index within the vector where **value** can be inserted while maintaining sorted order.

```
static int find(const Vector<string>& v, const string& value) {
   for (int pos = 0; pos < v.size(); pos++) {
      if (value <= v[pos]) {
         return pos;
      }
   }

   return v.size();
}

static bool expTreeContains(const expnode *root, const string& value);
```

c. Write the **expTreeInsert** function, which takes the root of an exponential tree [by reference] and the value to be inserted, and updates the tree to include the specified value, allocating and initializing new **expnode**s and arrays of **expnode ***s as needed. Ensure that you never extend a **values** vector beyond a length that matches the node's **depth**. Feel free to rely on **find** from part b.

```
static void expTreeInsert(expnode *& root, const string& value);
```

d. Finally, write the **expNodeDispose** function, which recursively disposes of the entire tree rooted at the specified address.

```
static void expTreeDispose(expnode *root);
```