CS106X
Autumn 2019

Handout 30S
November 6th and 7th, 2019

Section Solution

Problem 1 Solution: Cartesian Trees

I gave this question as an exam question about ten years ago, and most did very well on it. There are several approaches, but the most straightforward is one which scans the sequence of interest and identifies the minimum element (and its location), establishes that as the root, and then recurs on either side, as with:

```
static int findIndexOfMinimum(const Vector<int>& inorder, int low, int high) {
   int index = low;
   for (int i = low + 1; i <= high; i++) {
      if (inorder[i] < inorder[index]) {
         index = i;
      }
   }
   return index;
}

static node *arrayToCartesianTree(const Vector<int>& inorder, int low, int high) {
   if (low > high) return NULL;
   int index = findIndexOfMinimum(inorder, low, high);
   node *root = new node;
   root->value = inorder[index];
   root->left = arrayToCartesianTree(inorder, low, index - 1);
   root->right = arrayToCartesianTree(inorder, index + 1, high);
   return root;
}

static node *arrayToCartesianTree(const Vector<int>& inorder) {
   return arrayToCartesianTree(inorder, 0, inorder.size() - 1);
}
```

This O(nlgn) algorithm is the most straightforward divide-and-conquer algorithm I can think of, since it exploits the recursive substructure of trees. There are, not surprisingly, other approaches to building the tree. One very clever algorithm—an O(n) one that makes use of a stack—requires just a single pass over the entire array. It requires a much more careful implementation than the one needed for the O(nlgn) algorithm above.

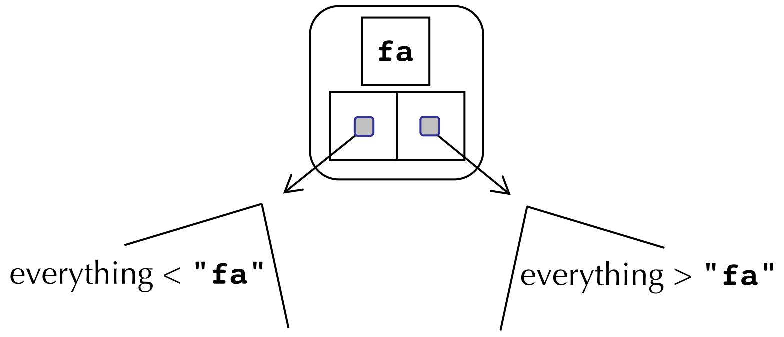
Problem 2 Solution: Patricia Trees

The **`containsWord`** routine is nontrivial, because it's as much about trees as it is about advanced string manipulation. It's complicated by the fact that the letters in the **`connection`** may be longer than the remaining portion of the word.

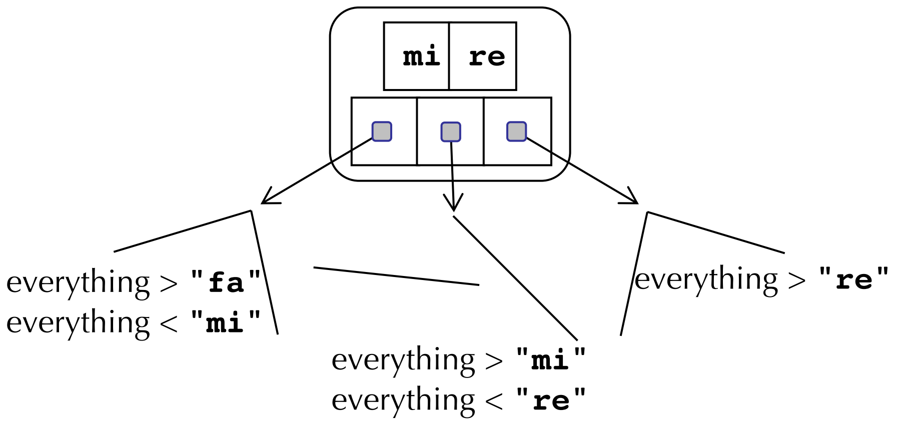
```
static int findConnection(const Vector<connection>& children,
                          const string& word) {
   for (int i = 0; i < children.size(); i++) {
      string prefix = word.substr(0, children[i].letters.size());
      int cmp = prefix.compare(children[i].letters);
      if (cmp == 0) return i;
      if (cmp < 0) break;
   }

   return -1;
}

static bool containsWord(const node *root, const string& word) {
   const node *curr = root;
   string clone = word;
   while (!clone.empty()) {
      int index = findConnection(curr->children, clone);
      if (index == -1) return false;
      clone = clone.substr(curr->children[index].letters.size());
      curr = curr->children[index].subtree;
   }

   return curr->isWord;
}
```

Problem 3 Solution: 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.

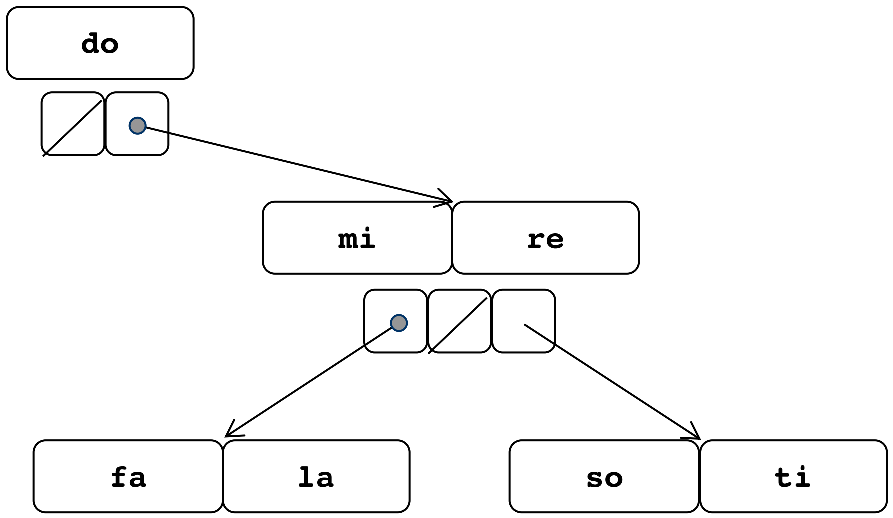
We can use the following data structure to build up and manage an exponential tree:

```
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. You may implement this either iteratively or recursively.

```
static bool expTreeContains(const expnode *root, const string& value) {
   const expnode *curr = root;
   while (curr != NULL) {
      int pos = find(curr->values, value); // provided in section handout
      if (pos < curr->values.size() && curr->values[pos] == value)
         return true;
      curr = curr->children == NULL ? NULL : curr->children[pos];
   }
   return false;
}
```

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 expnode *createExpNode(int depth) {
   expnode *node = new expnode;
   node->depth = depth;
   node->children = NULL;
   return node;
}

static void allocateChildPointers(expnode *node) {
   node->children = new expnode *[node->depth + 1];
   for (int i = 0; i < node->depth + 1; i++) node->children[i] = NULL;
}

static void expTreeInsert(expnode *& root, const string& value) {
   int depth = 1;
   expnode **currp = &root;
   while (true) {
      if (*currp == NULL) *currp = createExpNode(depth);
      expnode *curr = *currp;
      assert(curr->values.size() <= depth);
      int pos = find(curr->values, value);
      if (curr->values.size() < depth) {
         curr->values.insert(pos, value);
         if (curr->values.size() == depth) allocateChildPointers(curr);
         return;
      }
      currp = &curr->children[pos];
      depth++;
   }
}
```

There's also a recursive solution that avoids the double pointer calisthenics, and it adopts the same approach your textbook uses to insert a new value into a binary search tree. In practice, however, unary recursion is frowned upon if there's an equally time-efficient iterative alternative, since the iterative approach can do all the work within a single function and a constant amount of local memory.

```
static void expTreeInsert(expnode *& root, const string& value, int depth) {
   if (root == NULL) root = createExpNode(depth);
   assert(root->values.size() <= depth);
   int pos = find(root->values, value);
   if (root->values.size() < depth) {
      root->values.insert(pos, value);
      if (root->values.size() == depth) allocateChildPointers(root);
      return;
   }
   expTreeInsert(root->children[pos], value, depth + 1);
}

static void expTreeInsert(expnode *& root, const string& value) {
    expTreeInsert(root, value, 1);
}
```

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

```
static void expTreeDispose(expnode *root) {
   if (root == NULL) return;
   if (root->children != NULL) {
      for (int i = 0; i < root->depth + 1; i++)
         expTreeDispose(root->children[i]);
      delete[] root->children;
   }
   delete root;
}
```