Trie

1. What is a *trie*:

   You've probably already seen kinds of trees that store things more efficiently, such as a *binary search tree*. Here, we will examine another variant of a tree, called a *trie*.

   Aside: The name trie comes from its use for retrieval. It is pronounced like "try" by some, like "tree" (the pronunciation of "trie" in "retrieval") by others. Here, we will discuss a particular implementation of a trie, which may be somewhat different than how it is described elsewhere.

   We use a trie to store pieces of data that have a *key* (used to identify the data) and possibly a *value* (which holds any additional data associated with the key).

   Here, we will use data whose keys are *strings*.

   Suppose we want to store a bunch of *name/age* pairs for a set of people (we'll consider names to be a single string here).

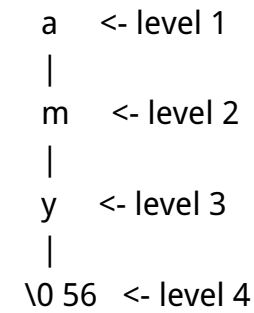
   Here are some pairs:

   ```
   amy 56
   ann 15
   emma    30
   rob 27
   roger   52
   ```

   Now, how will we store these name/value pairs in a trie? A trie allows us to share prefixes that are common among keys. Again, our keys are names, which are *strings*.

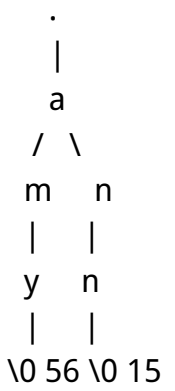
   Let's start off with amy. We'll build a tree with each character in her name in a separate node. There will also be one node under the last character in her name (i.e., under y). In this final node, we'll put the *nul character* (\0) to represent the end of the name. This last node is also a good place to store the age for amy.

   ```
   .    <- level 0 (root)
   |
   ```



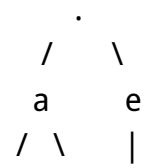
Note that each *level* in the trie holds a certain character in the string amy. The first character of a string key in the trie is always at level 1, the second character at level 2, etc.

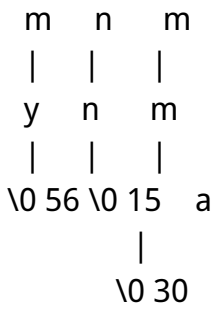
Now, when we go to add ann, we do the same thing; however, we already have stored the letter a at level 1, so we don't need to store it again, we just reuse that node with a as the first character. Under a (at level 1), however, there is only a second character of m...But, since ann has a second character of n, we'll have to add a new branch for the rest of a*nn*, giving:



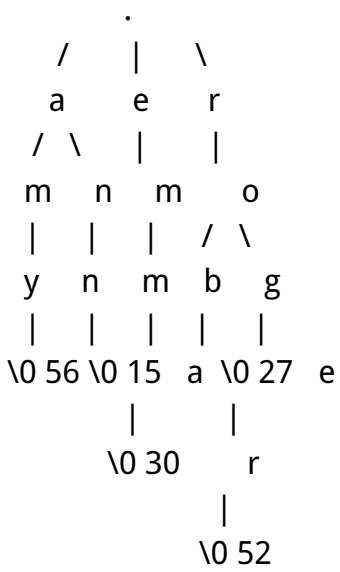
Note: Again, ann's data (an age of 15) is stored in her last node.

Now, let's add emma. Remember e is the first character and should go at level 1. Since there is no node with character e at level 1, we'll have to add it. In addition, we'll have to add nodes for all the other characters of e*mma* under the e. The first m will be a child of the e, the next m will be below the first m, etc., giving:





Now, let's add the last two names, namely rob and roger, giving:



Because the key for each piece of data is a sequence of characters, we will sometimes refer to that sequence as the *keys* (plural) for that data. For example, ann's data is referenced using the keys a, n, n (in that order).

To better understand how a trie works, answer the following questions.

- *What would the trie look like if we now added anne with age 67? How about ro with age 23?*
- *Would the trie look different if we added the names in a different order, say: rob, ann, emma, roger, amy?*
- *Is this a binary tree, tertiary tree or what? In other words, each node has at most how many children?*

2. Trie operations:

Here are the operations that we will concern ourselves with for this *trie*. You may

need others for a particular use of the trie.

- Add:

  We've already given examples of adding.

- IsMember:

  See if data with a certain string key is in the trie.

  For example, IsMember(trie, "amy") should report a true value and and IsMember(trie, "anna") should report a false value.

  We can imagine other variations where we do something with the *value* (like return it) once we find something with the *matching key*.

- Remove:

  Remove something from the trie, given its key.

We may want more operations depending on how we'll use the trie.

Since our trie holds data with string keys, which of the operations need a key and value, and which just need keys?

3. IsMember algorithm:

   Remember that a trie is a special kind of tree. Since a trie organizes its data via the keys (as specified above), it is easy to find whether a particular key is present.

   Finding a key can be done with iteration (looping).

   Here is an outline of such an algorithm. It looks in a particular *trie* and determines whether data with a particular string *key* is present.

   IsMember(*trie*, *key*) [iterative]

   1. Search top level for node that
      matches first character in key
   2. If none,
        return false
      Else,

3. If the matched character is \0?
   return true
  Else,
4. Move to subtrie that matched this character
5. Advance to next character in key*
6. Go to step 1

---

* I.e., the new search key becomes the old one without its first character.

The algorithm moves down the tree (to a subtree) at step 6. Thus, the *top level* in step 1 actually may refer to any level in the tree depending on what subtree the algorithm is currently at.

4. Trie implementation:

   Now, let's think about how to actually implement a trie of *name/age pairs* in C.

   As usual, we'll put the data structure in its own module by producing the source files trie.h and trie.c.

   The functions needed for our trie are the operations we mentioned:

   ```
   TrieAdd()
   TrieIsMember()
   TrieRemove()
   ```

   However, we also need additional functions for *setup* and *cleanup*:

   ```
   TrieCreate()
   TrieDestroy()
   ```

   Now, before we ponder the details of the trie functions, what must we decide on?

5. Organization of data types for a trie:

   Let's think about the data types for a trie and how to divide them between the *interface* (in trie.h) and the implementation (in trie.c) using ADTs and CDTs.

   We'll start with the type of a value. Since our values are ages, we have the following:

   ```
   typedef int trieValueT;
   ```

Since the type of values is something that people using the trie need to know, it goes in the interface (trie.h).

Next, we decided that keys will always be strings. However, we will not construct *elements* that are made up of *strings* and *values*. The reason is that we do not store entire string keys in nodes of the trie. Remember, we store only the individual characters of the string key in the nodes.

Thus, the type of a node begins as:

```
typedef struct trieNodeTag {
  char key;
  trieValueT value;
  ...
} trieNodeT;
```

Since it is only a detail of the implementation, it goes in trie.c.

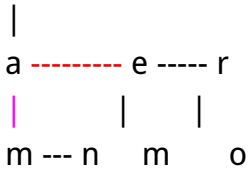
Note: We could make the trie more generic, by allowing it to handle keys that are any type of *array*, i.e., arrays of things other than characters. For other types of arrays, we'd have to determine how to represent the end-of-key, which we currently do with the *nul character* (\0).

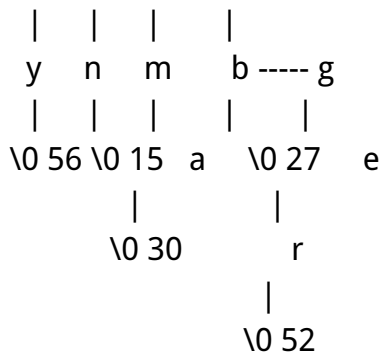
For now, we'll just hardcode the use *character* for the key stored at each node, and *string* (i.e., array of character) for the entire key (or sequence of keys) associated with each piece of data.

Now we need to complete the type of a node. *How will we construct a tree whose nodes can have several children?* One way is to have the children of a node be part of a linked list of nodes.

Structure

If we view siblings at a level as being linked in a list, then the trie we saw above now could be viewed structurally as:





First, the associated nodes at a given level form a linked list (e.g., a, e, r at level 1). Note, however, that each level may have more than one linked lists. For example, at the second level, m and n form their own list (as they are associated with a at the first level). Likewise, m (as it is associate with e at the first level) forms its own linked list. And finally, o, which is associated with r at the first level, forms its own list.

Thus, each node (e.g., a at level 1) has a link to the next node at that level and a link to a list of its children. To implement this structure, we will need two pointers in a node, giving:

```
typedef struct trieNodeTag {
  char key;
  trieValueT value;
  struct trieNodeTag *next, *children;
} trieNodeT;
```

Note: The *value* part of a node is unused in most cases since we only store the value in the node with the nul character (\0) as a key. If a value was something that was large, we would have to consider being smarter about our design.

The only types left are those that keep track of the trie. Based on our choice for the structure of the trie implementation, we see we'll need a pointer to the top level's first node.

Since this pointer has to do with the *implementation* of the trie, we put it in the *concrete type*, struct trieCDT:

```
typedef struct trieCDT {
  trieNodeT *root;
} trieCDT;
```

In the interface, we must fill in what the *abstract type* is as follows:

```
typedef struct trieCDT *trieADT;
```

Finally, we have:

```
trie.h                    trie.c
------                    ------
                          #include "trie.h"

                          typedef struct trieNodeTag {
                            char key;
                            trieValueT value;
typedef int trieValueT;           struct trieNodeTag *next,
                                    *children;
                          } trieNodeT;

typedef struct trieCDT          typedef struct trieCDT {
      *trieADT;           trieNodeT *root;
                          } trieCDT;
```

6. Using a trie:

   Now that we've decided on the data types for a trie, we can imagine how our trie will be used:

```
trieADT trie;

trie = TrieCreate();

TrieAdd(trie, "amy", 56);
TrieAdd(trie, "ann", 15);

if (TrieIsMember(trie, "amy"))
  ...
```

   When someone needs a trie, they define a trieADT variable and set it up with TrieCreate().

   Note: Since we don't store entire string keys and values together (per our discussion

above), you might pass a key and a value separately to TrieAdd().

7. Filling in trie functions:

   Let's now consider the prototype for our TrieIsMember() function:

   ```
   int TrieIsMember(trieADT trie, char keys[]);
   ```

   It must take the trie in which to look for data and the string key (i.e., a sequence of character *keys*) used to find that data. In addition, it needs to return a true or false value based on whether it finds the key or not.

   Here is an implementation based on the algorithm we already discussed:

   ```
   int TrieIsMember(trieADT trie, char keys[])
   {
     /* Start at the top level. */
     trieNodeT *level = trie->root;

     /* Start at beginning of key. */
     int i = 0;

     for (;;) {
       trieNodeT *found = NULL;
       trieNodeT *curr;

       for (curr = level; curr != NULL; curr = curr->next) {
         /*
          * Want a node at this level to match
          * the current character in the key.
          */
         if (curr->key == keys[i]) {
           found = curr;
           break;
         }
       }

       /*
        * If either no nodes at this level or none
        * with next character in key, then key not
   ```

```
     * present.
     */
    if (found == NULL)
      return 0;

    /* If we matched end of key, it's there! */
    if (keys[i] == '\0')
      return 1;

    /* Go to next level. */
    level = found->children;

    /* Advance in string key. */
    i++;
  }
}
```

Fill in the prototypes for the rest of the trie functions:

```
return-type TrieCreate(parameters);
return-type TrieDestroy(parameters);
return-type TrieAdd(parameters);
int         TrieIsMember(trieADT trie, char keys[]);
return-type TrieRemove(parameters);
...
```

and then implement them.

8. A more generic trie:

   We can easily redesign the trie so that it can use keys that are different kinds of arrays.
