Linked Data Structures

Assigned: Sunday, November 1
Due: Tuesday, November 10, before midnight
Value: 20 points (for successfully submitting a functionally correct program to your Assembla repo before the deadline). Late submissions will receive a 0.

Executive Summary

In this lab, you will be designing a linked data structure, specifically a binary search tree.

Purpose

- Increase your understanding of linked data structures
- Improve your pointer manipulation skills

Deliverables

In this assignment, you'll be designing a binary search tree data structure to store urls and their counts. You will be writing `populate_tree`, `add_to_tree`, `lookup` and `treeprint` functions for your binary search tree.

Download files: `main.h`, `main.c`, `assignment-6.c`, `small_url`, `long_url`

The file `main.c` calls your binary search tree functions that you will write in `assignment-6.c` Two files `"small_url"` and `"long_url"` containing repeated urls have also been provided.

Context

User Story: Self-referential Structures

Suppose you are designing a brand new shiny browser coming out of UT called **Longhorn_Surf**. We want it to show the most frequently visited urls based on user history. Since the list of urls is not known in advance, we can't conveniently sort it and use a binary search. Yet we can't do a linear search for each url as it arrives, to see if it's already been seen; the program would take too long. How can we organize the data to cope efficiently with a list of arbitrary urls?

One solution is to keep the set of urls seen so far sorted at all times, by placing each url into its proper position in the order as it arrives. This shouldn't be done by shifting urls in a linear array

or linked list, since that also takes too long. Instead we will use a data structure called a *binary search tree.*

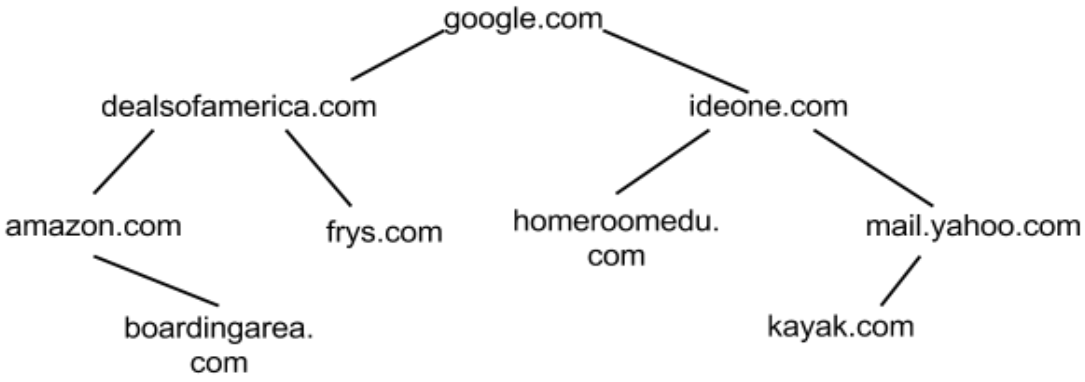
Binary Search Tree

Your binary search tree contains one “node” per distinct url; each node contains:

- a pointer to the text containing the url
- a count of the number of occurrences of the url
- a pointer to the left child node
- a pointer to the right child node

No node may have more than two child nodes; it may have 1) no child nodes or 2) one left or one right child node or 3) one left child and one right child node.

The nodes are maintained so that at any node in the left subtree contains only urls that are lexicographically less than the url at the node, and the right subtree contains only urls that are greater. This is a tree for the set of urls: {“google.com”, “ideone.com”, “mail.yahoo.com”, “dealsofamerica.com”, “frys.com”, “amazon.com”, “homeroomedu.com”, “kayak.com”, ”boardingarea.com”}, as built by inserting each url as it is encountered:



URL Binary Tree

To find out whether a new url is already in the tree, start at the root and compare the new url to the url stored at that node. If they match, the question is answered affirmatively. If the new url is less than the url at the root, continue searching at the left child, otherwise at the right child. If there is no child in the required direction, the new url is not in the tree, and in fact the empty slot is the proper place to add the new url. This process is recursive, since the search from any node uses a search from one of its children. Accordingly, recursive routines for insertion and printing will be most natural.

Going back to the description of a node, it is most conveniently represented as a structure with four components:

```
struct tnode {            /* the tree node: */
    char *key;            /* points to the url string */
    int count;            /* number of occurrences */
    struct tnode *left;   /* left child */
    struct tnode *right;  /* right child */
};
```

The main routine reads urls from a given file with function `populate_tree` and adds them to the tree with `add_to_tree`. It later calls `test_print`, which calls `treeprint` to do an inorder traversal of the binary search tree

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAX_URL_LENGTH 100

void populate_tree(char *url_file, struct tnode**);
struct tnode* add_to_tree(struct tnode *, char *);
int lookup(struct tnode *, char *);

void test_print(struct tnode*, int, char [][MAX_URL_LENGTH],int *);

int treeprint(int size, struct tnode* p, char
URL_array[][MAX_URL_LENGTH], int *);

main()
{
   struct tnode *root;
   char word[MAX_URL_LENGTH];
   root = NULL;
   populate_tree("small_url", &root);
   test_print(root, 7, Expected_URLs_small, Expected_Url_Freq_small)

   root = NULL;
   populate_tree("long_url", &root);
   test_print(root, 10, Expected_URLs, Expected_Url_Freq);

  return 0;
}
```

The function `add_to_tree` is recursive. A url is presented by main to the top level (the root) of the tree. At each stage, that url is compared to the url already stored at the node, and is

percolated down to either the left or right subtree by a recursive call to `add_to_tree`. Eventually, the url either matches something already in the tree (in which case the count is incremented), or a null pointer is encountered, indicating that a node must be created and added to the tree.

If a new node is created (i.e., the url was not previously in the tree), `add_to_tree` returns a pointer to the new node, which is also pointed to by the parent node (pointed to by the left or right pointer).

```
/* add_to_tree:  add a node with w, at or below p */
struct tnode* add_to_tree(struct tnode *p, char *w)
{

}
```

You need to allocate storage for the new node if it is not already present in the tree. The new url is copied into the new space. The `count` is initialized, and the two children are made null. This part of the code is executed only at the leaves of the tree, when a new node is being added. (Hint: You can use `malloc`, `strcpy` for these operations)

`treeprint` stores *at most* 10 urls in a string array in alphabetical order. It traverses from the root of the tree in sorted order; at each node, it copies the left subtree (all the urls less than this url), then the url itself, then the right subtree (all the urls greater). (Inorder traversal)
The function also stores the frequency of each url in a corresponding integer array. It returns the number of unique urls it could find with a max of 10.

```
/* treeprint:  in-order print of tree p */
int treeprint(int size, struct tnode *p, char [][MAX_URL_LENGTH],
int*)
{

}
```

Requirements

Setup

Either use the provided solution project that pushed to your repo, or follow these steps to manually create the project:

1. Create a Project named `Lab6` in Visual Studio, following the same setup as in previous labs.
2. Download the starter files, **place them under the Lab6 folder that was created**

From the Solution Explorer, add all the downloaded files to your project, so your file structure will be `<repo>/Lab6/<individual .c or .h files>`

Implementation Notes:

1. Use strcpy to create new strings. You need to allocate sufficient space for the destination string beforehand.

FAQs

Q. What is done for unbalanced trees?

- A practical note: if the tree becomes ‘’unbalanced'' because the urls don't arrive in random order, the running time of the program can grow too much. As a worst case, if the urls are already in order, this program does an expensive simulation of linear search. There are generalizations of the binary tree that do not suffer from this worst-case behavior, but we will not describe them here.

Q. Is “www.google.com” treated as a different url from “google.com”?

- Yes

Q. Are URLs case-sensitive?

- No. But we will not provide any URLs containing capital letters.