CS 111 Design Problem

Laboratory 1 – Tab Completion

Mark Vismonte
mark.vismonte@gmail.com
503765196

Timothy Wang
timzwang@gmail.com
Tim's ID number

Professor Kohler
UCLA Winter Quarter - 2011
Saturday, January 29, 2011

Vismonte 503765196
Wang #########

Introduction

For our design problem, we decided to explore how to further emulate the *bash* shell. We will observe that behavior of tab completion in order to define a plan for extending our own *ospsh* shell. The feature we will be observing tab completion.

We will also explore the functionality of the GNU readline library. According to the description, this library can be used to display a prompt and receive the entered string from the user. In addition, the library provides emacs style bindings to the user. This allows the user to use shortcuts such as ctrl + e to get to the end of a line and ctrl + u to delete a line.

Tab Completion

One of the most widely used features of *bash* is tab completion. This feature allows the user to press the tab key to automatically display and complete a list of commands that match the current inputted string. For example, a user would start typing in “ssh” if they wanted to execute the command “ssh-keygen”. After initially typing this in, the user would then be able to type the tab key. On the line below the current line, a list of matched commands are printed out. For this example, “ssh”, “ssh-agent”, and “ssh-keygen” are all commands that would be printed out.

This also allows the user to finish

Binary Search Tree

While other data structures could be used in our implementation, we decided to implement a binary search tree. A binary search tree is a dynamically allocate structure that is guaranteed to be sorted. It is easy to insert new entries into it, and does not require shifting of items. All it requires is allocation of a new node and inserting it into its proper place.

While an array or a linked list were choices four our data structure, we ultimately decided to implement a binary search tree.

Feature	Dynamic Array	Linked List	Binary Search Tree
Sorting	Algorithm would include shifting elements over	Sorting built in to the algorithm used to insert; would have to traverse the whole linked list; Worse case O(N)	Sorting built in to the algorithm used to insert; Ideal worse case O(NlogN), real worse case O(N)
Overhead (extra bytes besides string)	None; the string would be the only variable stored	4 bytes; we need a pointer to the next node	8 bytes; we need pointers to the left and right children

Talk about big O.

Design / Implementation Details

Adding Readline

Related Functions:

```
char *readline (const char *prompt);
char *fgets (char *str, int num, FILE *stream);
```

Implementation:

The first thing we need to do to implement tab completion is to introduce the *readline* function into our existing main. Currently, the function *fgets* is being used to retrieve input from the keyboard. In addition, the way *readline* works is different compared to how *fgets* works. The *readline* function allocates memory for a string and returns it while *fgets* takes the pointer to an existing buffer as well as its size and writes into it.

To implement this, we need to create a `char *` variable. We can then set this variable to the return value of *readline*. If the return value is equal to NULL, then we should exit because then EOF was returned. Otherwise, we should use the return value and put it through the existing operations. Also, it is important to free the string at the end of the iteration.

Storing commands

Related Variable:

```
pathcommand_t *HEAD;
```

Related Struct:

```
typedef struct pathcommand pathcommand_t;
//Define a node for our BST
struct pathcommand {
    char *cmd;
    pathcommand_t *left;
    pathcommand_t *right;
};
```

Related Functions:

```
pathcommand_t *pathcommand_alloc(void); //alloc
void pathcommand_free(pathcommand_t *pathcmd); // free
void add_pathcommand(char *cmd);
void print_tree(void); // only used for debugging
```

Implementation:

First, we will strictly explain how we will build our data structure. If we want to build a binary search tree, then we will have to create a node struct. That is precisely what `pathcommand_t` is. The data value for our struct is `char *cmd`. In addition, we will create a `pathcommand_t *HEAD` as a global variable since we will only have one tree. We can reference this as our head pointer for the binary search tree.

Our design requires us to create a function to add `pathcommand_t` ‘s, but not to remove. As a result, we should create a command that adds a pathcommand to the tree. We can call a recursive helper function to add the new command to the tree.

Finding commands

Related Struct:

```
struct dirent { // d_name is the only important member of this struct
     char d_name[255 + 1];/* name must be no longer than this */
};
```

Related Functions:

```
char *getenv(const char *name);
DIR *opendir(const char *dirname);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);
void initialize_path_tree(void);
```

Implementation:

Our approach here is to populate a list of all the commands when our program starts. We will write the function `initialize_path_tree` to accomplish this task. Our algorithm will be simple:

Replace with a diagram?

1. Add all our special commands to the BST (e.g. cd, exit, makeq…)
2. Go through each directory in your PATH
    a. Open the directory
    b. Iterate through the items in the directory
        i. If the item is not “.” Or “..”
            1. Add it to the BST
    c. Close the directory

Integrating with readline

Related Functions:

Implementation:

The first thing we need to do to implement tab completion is to introduce the readline function into our

Results

Conclusion

Tab Completion

Other Random Features