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
Monday, January 31, 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 feature is important to users because it allows them to discover their path and save time by simply pressing tab. It is one that is noticeable and missed if no present. For example, we found ourselves frequently pressing tab while testing and completing the main assignment for lab 1b, hoping that our desired command would miraculously show up.

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. A dynamic array and a linked list were among also among our choices when deciding on which data structure we were going to use.

Feature	Dynamic Array	Linked List	Binary Search Tree
Adding (allocation)	Would only need to use realloc if not enough space. Otherwise just insert element	Need to allocate enough memory for a new node	Need to allocate enough memory for a new node
Add (sorting)	The keep elements in order, we would need to shift elements in the array 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) if the tree is balancing
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

Searching algorithm	Use binary search to find first matching command and go down the array until there are no more matching commands	Must traverse from the head of the array list and first the item that matches, until there are no more matching commands	Use binary search to find matching commands

Design / Implementation Details

In our implementation, we will be expanding off the current lab 1. We will be adding two files to our project: tab_completion.h and tab_completion.c. The tab_completion files will handle the data structure and its functions, as well as the requirements for *readline*. We will discuss the details of our implementation plan in this section.

Adding Readline

Related files:

main-b.c

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 files:

tab_completion.h / tab_completion.c

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. We should also remember not to add duplicates to the tree. If we want our algorithm to have O(NlogN) search time, then we need to make sure that its insertion algorithm is autobalancing. The follow diagram shows how we intend to make our data structure auto balance:

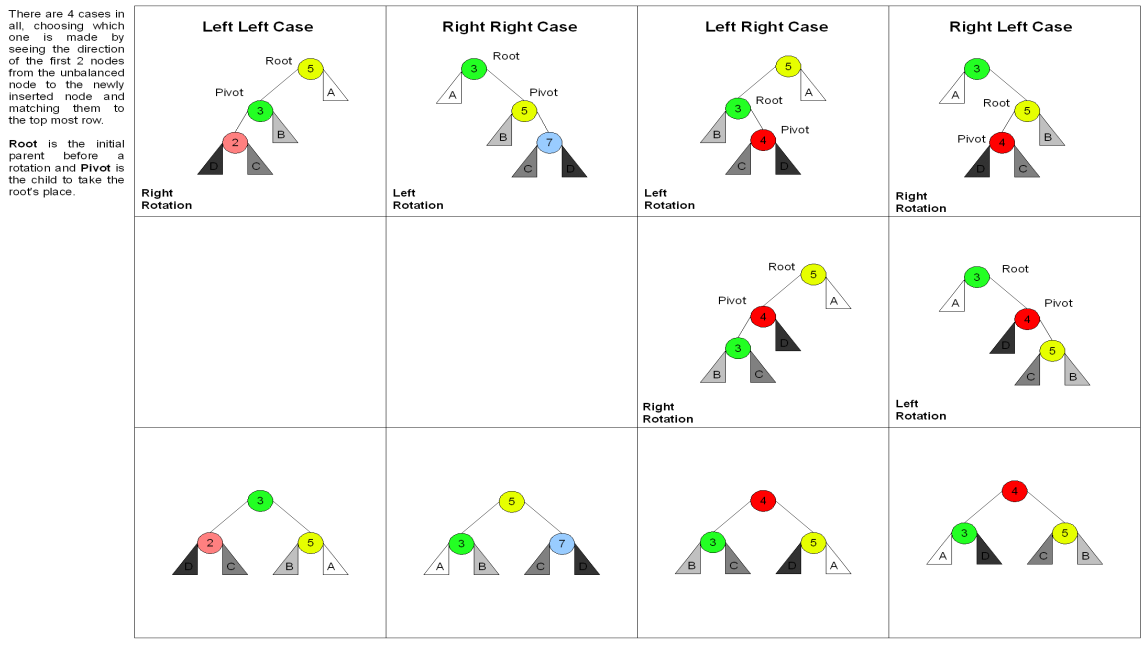

Figure 1 Rebalancing Algorithm from Wikipedia

Finding commands

Related files:

tab_completion.h / tab_completion.c

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:

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

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:

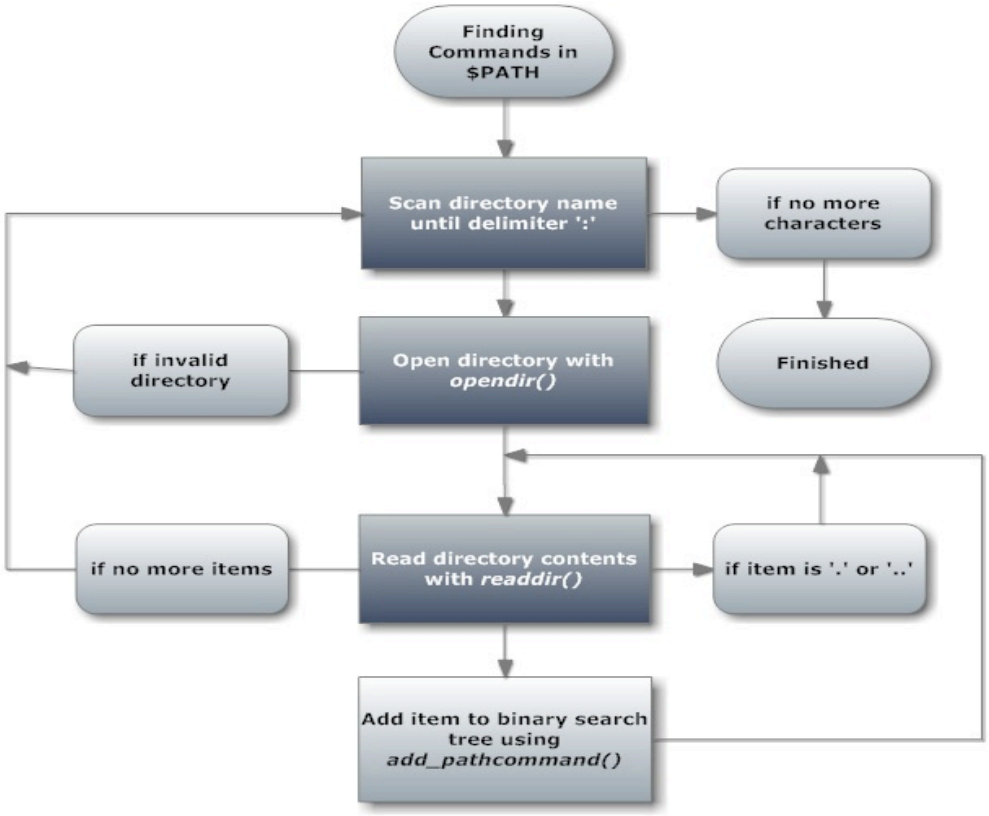

Figure 2 Finding all commands in PATH

At the beginning of the function, we should also remember to add the special commands to our tree. We should insert “makeq” first since it’s roughly close to the middle of the alphabet, but we should not forget to add “cd”, “exit”, “q”, and “waitq” to our structure.

Integrating with readline

Related files:

tab_completion.h / tab_completion.c / main-b.c

Related Variables:

```
rl_attempted_completion_function //from readline library
```

Related Functions:

```
char **find_matches(pathcommand_t *cur, char *str, size_t len,
    char **ret, int *index, int *size);
char *command_generator(char *str, int state);
char **command_completion(char *str, int start, int end);
char **rl_completion_matches(char *str, char * (function)(char, int));
```

Implementation:

After reading the *readline* documentation, we discover that we need to implement a few functions in order to use our library. First, we need to create the function *command_completion()*. This function will be called when tab completion is attempted; however, it currently maps to the default function. In order to override this, we must set the *rl_attempted_completion_function* variable to *command_completion*. After this is set in our main, the function should be called when tab is pressed.

Our *command_completion* method currently doesn’t have anything in it. The *readline* documentation says that we should the function *rl_completion_matches* to generate the list of strings that are returned. That function accepts a pointer to another function. This function is supposed to be called multiple times until it returns null. This is when we know we have no more strings that match.

This function will be called *command_generator()*. It will have a static array of strings, which will be initialized to an array of all the commands when state equals 0. State is only equal to zero the first time it is being called with a certain string. The function *find_matches()* is used to find all the matches. It uses a modified search algorithm to find all the commands. This is because instead of just looking for one command, it is trying to add all the commands that have the first part equal. For example, a normal search would only look to the left node if the current string were lexicographically less than the one of the node. Instead we will use a partial compare and also look left if the nodes are equal.

We can check our completeness by trying to command complete a command in our shell and then trying to command complete it in bash or another shell. If the results are equal, then our algorithm should be correct.

Results

This should be a summary of the results. What were the successes of our design. What were the failures? How could we have made it better? Did we complete the implementation specified above? What obstacles did we encounter? What there anything that was easier than expected? What did we learn?

Conclusion

How did we divide the work? Who did what? What did we learn? What other implementations could we have tried? What was the hardest part of implementation?