Jeff Morton
Data Structures and Algorithms 2
Project 1
Functional Decomposition

User-defined data structures used as parameters in the functions:

```
typedef struct Node *NodeP;
typedef struct Node
{
      NodeP parent, left, right;
      void *info;
} Node;


typedef struct Tree
{
      NodeP root, curDir;
}Tree, *TreeP;


typedef struct Command *CommandP;
typedef struct Command
{
      char *args[3];
      int argCount; // number of arguments in the current
command.
}Command;


typedef struct Info
{
      char name[MAX_ARG_LEN+1];
      char type;
}Info, *InfoP;
```

Files and Functions in the Program:

```
/*************************************************************
 * Student Name: Jeff Morton
 * File Name: BinaryTree.c
 * Assignment Number: 1
 * Date Due: Jan 31, 2016
```

```
 *
 *  Created on: Jan 26, 2015
 *      Author: Jeff Morton (jhm14@students.uwf.edu)
 *
 * About this file:
 * this file provides an API used for creating and managing binary trees
 ****************************************************************/
```

BinaryTree.c (Use for BinaryTree.h as well, same functions)

```
/**
 * newTree
 * malloc's a new tree struct and returns a StructP pointer to it.
 * Remember, this pointer must be type-casted by the program utilizing the ADT
 * @return TreeP pointer to the new tree
 */
TreeP newTree();

/**
 * newNode
 * malloc's a new node struct and returns a NodeP pointer to it.
 * Remember, this pointer must be type-casted by the program utilizing the ADT
 * @return NodeP pointer to new node
 */
NodeP newNode();

/**
 * getInfo
 * Returns a void pointer to the info for the tree node passed to it.
 * Remember, this pointer must be type-casted by the program utilizing the ADT
 * @param {NodeP} node - pointer to the node to get the info of
 * @return void* void pointer to node's info data
 */
void *getInfo(NodeP node);

/**
 * setInfo
 * Sets the info of the node passed to the void info pointer passed.
 * @param {NodeP} node - pointer to the node to set the info of
 * @param {void*} info - the info to set the info pointer in the node to
 */
void setInfo(NodeP node, void *info);

/**
 * getLeft
```

```
 * returns a NodeP pointer to the left child node of the NodeP passed.
 * @param {NodeP} node - the node to get the left child of
 * @return NodeP the left child of the node
 */
NodeP getLeft(NodeP node);

/**
 * getRight
 * returns a NodeP pointer to the right child node of the NodeP passed.
 * @param {NodeP} node - the node to get the right child of
 * @return NodeP the right child of the node
 */
NodeP getRight(NodeP node);

/**
 * getParent
 * returns a NodeP pointer to the parent node of the NodeP passed.
 * @param {NodeP} node - the node to get the parent of
 * @return NodeP the parent of the node
 */
NodeP getParent(NodeP node);

/**
 * setLeft
 * sets the left pointer of the first NodeP to the second NodeP passed.
 * @param {NodeP} node - the node to set the child of
 * @param {NodeP} node - the left child node
 */
void setLeft(NodeP node, NodeP node2);

/**
 * setRight
 * sets the right pointer of the first NodeP to the second NodeP passed.
 * @param {NodeP} node - the node to set the child of
 * @param {NodeP} node - the right child node
 */
void setRight(NodeP node, NodeP node2);

/**
 * setParent
 * sets the parent pointer of the first NodeP to the second NodeP passed.
 * @param {NodeP} node - the node to set the parent of
 * @param {NodeP} node - the parent node
```

```
 */
void setParent(NodeP node, NodeP node2);

/*****************************************************************
 * Student Name: Jeff Morton
 * File Name: CommandProcessor.c
 * Assignment Number: 1
 * Date Due: Jan 31, 2016
 *
 *  Created on: Feb 1, 2015
 *      Author: Jeff Morton (jhm14@students.uwf.edu)
 *
 * About this file:
 * this file provides an API to handle the core functionality of all of the commands the filesystem supports
 *****************************************************************/
```

CommandProcessor.c

```
/**
 * doesFileExist
 * Recursively checks the current working directory for conflicting/duplicate file names
 * Returns TRUE if duplicate name is found, FALSE if there is no duplicate file name.
 * @param {char *} name - the name of the file/folder to search for
 * @param {NodeP} node - the first node to check in the directory
 * @return int returns TRUE if file exists, otherwise FALSE
 */
int doesFileExist(char *name, NodeP node);

/**
 * printParents
 * prints out the parent directories of the node passed (prints file location)
 * @params {NodeP} node - the node to print the parents of
 */
void printParents(NodeP node);

/**
 * insertNode
 * inserts the node into the directory, organized alphabetically
 * @params {NodeP} nodeSRC - the node to insert
 * @params {NodeP} nodeFS - specifies the directory to insert source node into
 */
void insertNode(NodeP nodeSRC, NodeP nodeFS);
```

```
/**
 * findNode
 * Recursively checks the current working directory for for the node specified
 * Returns NodeP if file/folder is found, NULL if not found
 * @param {char *} name - the name of the file/folder to search for
 * @param {NodeP} node - the first node to check in the directory
 */
NodeP findNode(char *name, NodeP node);

/**
 * clone
 * Recursively clones the node passed as well as all children and returns a pointer to the
node
 * @params {NodeP} nodeSRC - the source node to clone
 * @params {TreeP} tree - a pointer to the file system tree
 * @return NodeP the duplicate node with children created
 */
NodeP clone(NodeP nodeSRC, TreeP tree);

/**
 * recursiveFree
 * Recursively frees memory of the node passed, all children, and the info for the node
and children
 * @param {NodeP} node - the node free memory of
 * @return NodeP the remains of what was the node pointer. This is always NULL.
 */
NodeP recursiveFree(NodeP node);

/**
 * postOrderSearch
 * performs a post-order traversal of the file system for a file/folder with the name passed
 * @params {NodeP} node - the node to start the search at
 * @params {TreeP} tree - a pointer to the file system tree
 * @param {char *} name - the name of the file/folder to search for
 * @return NodeP the pointer to the first node found with the name specified. NULL if not
found.
 */
NodeP postOrderSearch(NodeP node, TreeP tree, char *name);

/**
 * newCommand()
 * makes a new Command struct and returns a pointer do it.
 * This is used to store command args.
```

```
 * @return CommandP a pointer to the new command struct
 */
CommandP newCommand();

/**
 * newInfo()
 * makes a new Info struct and returns a pointer do it.
 * This is used to store file/folder info.
 * @return InfoP a pointer to the new info struct
 */
InfoP newInfo();

/**
 * getCommand
 * takes command input from the user, processes and stores the
 * command args in arg1, arg2, and arg3 in the CommandP passed.
 * @param {CommandP} command - the command to store the user's command in
 * @return CommandP the command with user data
 */
CommandP getCommand(CommandP command)

/**
 * takes command input from commands.txt, processes and stores the
 * command args in arg1, arg2, and arg3 in the CommandP passed.
 * @param {CommandP} command - the command to store the user's command in
 * @param {FILE *} inFile - the file pointer from which to read commands from
 * @return CommandP the command with user data
 */
CommandP readCommand(CommandP command, FILE *inFile);

/** Functions:
        ls              // lists all files and directories in the current directory, indicating which
(file or directory) it is
        mkdir <dirname> // creates a new directory if it does not already exist
        cd <dirname>    // changes into specified directory if it exists
        cd ..           // changes to the parent directory
        pwd             // specifies the current directory as: <yourname>/root/nextdir/etc/
        addf <filename> // adds a file to the current directory
        mv <fname1> <fname2> // change the name of the file or directory to the new
name
        cp <fname1> <fname2> // copy file or folder to the new name
        rm <filename>   // locate and remove the file or directory
        bye             // ends the session
```

```
        whereis <filename>   // show path to first occurrence of file or directory if it exists
 */

/**
 * processCommand
 * Processes the CommandP passed and calls the appropriate functions to handle it.
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
 * @param {FILE *} inFile - the file pointer from which commands might be read. Needs
this in case it needs to be closed.
 */
void processCommand(CommandP command, TreeP tree, FILE *inFile);

/**
 * ls
 * lists all files and directories in the current directory, indicating which (file or directory) it
is
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
 */
void ls(CommandP command, TreeP tree);

/**
 * mkdir
 * creates a new directory if it does not already exist
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
 */
void mkdir(CommandP command, TreeP tree);

/**
 * cd
 * changes into specified directory if it exists or the parent directory
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
 */
void cd(CommandP command, TreeP tree);

/**
 * pwd
 * specifies the current directory as: <yourname>/root/nextdir/etc/
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
```

```
 */
void pwd(CommandP command, TreeP tree);

/**
 * addf
 * adds a file to the current directory
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
 */
void addf(CommandP command, TreeP tree);

/**
 * mv
 * change the name of the file or directory to the new name
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
 */
void mv(CommandP command, TreeP tree);

/**
 * cp
 * copy file or folder to the new name
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
 */
void cp(CommandP command, TreeP tree);

/**
 * rm
 * locate and remove the file or directory
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
 */
void rm(CommandP command, TreeP tree);

/**
 * whereis
 * show path to first occurrence of file or directory if it exists
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
 */
void whereis(CommandP command, TreeP tree);
```

```
/**
 * bye
 * exits the program and closes the input file
 * @param {FILE *} inFile - the input file to close
 */
void bye(FILE *inFile);


/* these are all utility functions */
/**
 * listArgs
 * lists the arguments of the current command
 * @param {CommandP} command - the command to list the arguments of
 */
void listArgs(CommandP command);
```

```
/*****************************************************************
 * Student Name: Jeff Morton
 * File Name: Terminal.c
 * Assignment Number:
 * Date Due: Jan 31, 2016
 *
 *  Created on: Feb 1, 2015
 *      Author: Jeff Morton (jhm14@students.uwf.edu)
 *
 * About this file:
 * this file is merely responsible for calling a few functions to set up the program and repeatedly call to get input
 *****************************************************************/
```

Terminal.c

```
/**
 * main
 * The main function of the entire program. Literally does nothing.
 * @return int returns an integer exit code.
 */
int main();
```

```
/*************************************************************
 * Student Name: Jeff Morton
 * File Name: BinaryTree.h
 * Assignment Number: 1
 * Date Due: Jan 31, 2016
 *
 *  Created on: Jan 26, 2015
 *      Author: Jeff Morton (jhm14@students.uwf.edu)
 *
 * About this file:
 * this file provides an interface used for creating and managing binary trees
 *************************************************************/
```

BinaryTree.h

```
typedef struct Node *NodeP;
typedef struct Node
{
      NodeP parent, left, right;
      void *info;
} Node;


typedef struct Tree
{
      NodeP root, curDir;
}Tree, *TreeP;


/**
 * newTree
 * malloc's a new tree struct and returns a StructP pointer to it.
 * Remember, this pointer must be type-casted by the program utilizing the ADT
 * @return TreeP pointer to the new tree
 */
TreeP newTree();

/**
 * newNode
 * malloc's a new node struct and returns a NodeP pointer to it.
 * Remember, this pointer must be type-casted by the program utilizing the ADT
 * @return NodeP pointer to new node
 */
```

```
NodeP newNode();

/**
 * getInfo
 * Returns a void pointer to the info for the tree node passed to it.
 * Remember, this pointer must be type-casted by the program utilizing the ADT
 * @param {NodeP} node - pointer to the node to get the info of
 * @return void* void pointer to node's info data
 */
void *getInfo(NodeP);

/**
 * setInfo
 * Sets the info of the node passed to the void info pointer passed.
 * @param {NodeP} node - pointer to the node to set the info of
 * @param {void*} info - the info to set the info pointer in the node to
 */
void setInfo(NodeP, void *);

/**
 * getLeft
 * returns a NodeP pointer to the left child node of the NodeP passed.
 * @param {NodeP} node - the node to get the left child of
 * @return NodeP the left child of the node
 */
NodeP getLeft(NodeP);

/**
 * getRight
 * returns a NodeP pointer to the right child node of the NodeP passed.
 * @param {NodeP} node - the node to get the right child of
 * @return NodeP the right child of the node
 */
NodeP getRight(NodeP);

/**
 * getParent
 * returns a NodeP pointer to the parent node of the NodeP passed.
 * @param {NodeP} node - the node to get the parent of
 * @return NodeP the parent of the node
 */
NodeP getParent(NodeP);
```

```
        /**
         * setLeft
         * sets the left pointer of the first NodeP to the second NodeP passed.
         * @param {NodeP} node - the node to set the child of
         * @param {NodeP} node - the left child node
         */
        void setLeft(NodeP, NodeP);

        /**
         * setRight
         * sets the right pointer of the first NodeP to the second NodeP passed.
         * @param {NodeP} node - the node to set the child of
         * @param {NodeP} node - the right child node
         */
        void setRight(NodeP, NodeP);

        /**
         * setParent
         * sets the parent pointer of the first NodeP to the second NodeP passed.
         * @param {NodeP} node - the node to set the parent of
         * @param {NodeP} node - the parent node
         */
        void setParent(NodeP, NodeP);
```

```
/*************************************************************
 * Student Name: Jeff Morton
 * File Name: CommandProcessor.h
 * Assignment Number: 1
 * Date Due: Jan 31, 2016
 *
 *  Created on: Feb 1, 2015
 *      Author: Jeff Morton (jhm14@students.uwf.edu)
 *
 * About this file:
 * this file provides an interface to handle the core functionality of all of the commands the
filesystem supports
 *************************************************************/
```

CommandProcessor.h

```
#define MAX_ARG_LEN 100 //max # of chars of any given command
argument

/**
 * /sets the max size of a command based on the MAX_ARG_LEN.
The +6 comes from:
 * +2 for each of the 3 args to include space or newline chars;
 */
#define MAX_CMD_LEN MAX_ARG_LEN*3 + 6

#define TRUE 1
#define FALSE 0


typedef struct Command *CommandP;
typedef struct Command
{
      char *args[3];
      int argCount; // number of arguments in the current
command.
}Command;//, *CommandP;


typedef struct Info
{
      char name[MAX_ARG_LEN+1];
      char type;
}Info, *InfoP;


/**
 * listArgs
 * lists the arguments of the current command
 * @param {CommandP} command - the command to list the arguments of
 */
void listArgs(CommandP command); //lists all the args of the current command

/**
 * newCommand()
 * makes a new Command struct and returns a pointer do it.
 * This is used to store command args.
 * @return CommandP a pointer to the new command struct
 */
```

```
CommandP newCommand();

/**
 * newInfo()
 * makes a new Info struct and returns a pointer do it.
 * This is used to store file/folder info.
 * @return InfoP a pointer to the new info struct
 */
InfoP newInfo();

/**
 * getCommand
 * takes command input from the user, processes and stores the
 * command args in arg1, arg2, and arg3 in the CommandP passed.
 * @param {CommandP} command - the command to store the user's command in
 * @return CommandP the command with user data
 */
CommandP getCommand(CommandP command);

/**
 * takes command input from commands.txt, processes and stores the
 * command args in arg1, arg2, and arg3 in the CommandP passed.
 * @param {CommandP} command - the command to store the user's command in
 * @param {FILE *} inFile - the file pointer from which to read commands from
 * @return CommandP the command with user data
 */
CommandP readCommand(CommandP command, FILE *inFile);

/**
 * processCommand
 * Processes the CommandP passed and calls the appropriate functions to handle it.
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
 * @param {FILE *} inFile - the file pointer from which commands might be read. Needs
this in case it needs to be closed.
 */
void processCommand(CommandP command, TreeP tree, FILE *inFile);

/**
 * ls
 * lists all files and directories in the current directory, indicating which (file or directory) it
is
 * @param {CommandP} command - the command to get parsed command from
```

```
 * @param {TreeP} tree - a pointer to the file system
 */
void ls(CommandP command, TreeP tree);

/**
 * mkdir
 * creates a new directory if it does not already exist
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
 */
void mkdir(CommandP command, TreeP tree);

/**
 * cd
 * changes into specified directory if it exists or the parent directory
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
 */
void cd(CommandP command, TreeP tree);

/**
 * pwd
 * specifies the current directory as: <yourname>/root/nextdir/etc/
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
 */
void pwd(CommandP command, TreeP tree);

/**
 * addf
 * adds a file to the current directory
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
 */
void addf(CommandP command, TreeP tree);

/**
 * mv
 * change the name of the file or directory to the new name
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
 */
void mv(CommandP command, TreeP tree);
```

```
/**
 * cp
 * copy file or folder to the new name
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
 */
void cp(CommandP command, TreeP tree);

/**
 * rm
 * locate and remove the file or directory
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
 */
void rm(CommandP command, TreeP tree);

/**
 * whereis
 * show path to first occurrence of file or directory if it exists
 * @param {CommandP} command - the command to get parsed command from
 * @param {TreeP} tree - a pointer to the file system
 */
void whereis(CommandP command, TreeP tree);

/**
 * bye
 * exits the program and closes the input file
 * @param {FILE *} inFile - the input file to close
 */
void bye(FILE *inFile);
```