Binary Search Tree – Course Project

Software development with the C-programming language for international programs

Bianca-Ioana Stefanescu

March - April 2025

Table of Contents

1 AI Declaration

I used AI (Chat GPT) in making this project. I used it to:

>> make sure I tackle all situations that might throw errors
>> to fix small issues in the code (but no generated solution was copy-pasted and I asked the AI to “help me identify the issue without giving the solution, only hints”)
>> to refine some of the documentation as my ways of explaining were unclear and messy
>> it helped me by suggesting using “**`while(getchar() != '\n');`**” to manage input buffer

2 Implementation

2.0 libraries and structures

The libraries included are:

Line / Block of code	Description
`#include <stdio.h>`	because I use basic predefined functions such as **`printf()`** and **`scanf()`**
`#include <string.h>`	because I use predefined functions for strings such as **`strcmp()`**
`#include <stdlib.h>`	because I use predefined functions such as **`malloc()`** and **`realloc()`**

The structures defined are the ones provided in the project description:

Line / Block of code	Description
`typedef struct node {` `  char key[30];` `  char description[200];` `  struct node *left;` `  struct node *right;` `  } Node;`	Definition for Node type of structure
`typedef struct {` `  Node *root;` `  } Tree;`	Definition for Tree type of structure

2.1 createNode

Functionality: **`createNode()`** creates a new node. It takes the key (the identifier) and the description as parameters. If either of them is missing (NULL), the function asks the user to enter them manually. Their validity is checked using **`isValidKey()`** (section 2.1.1) and **`isValidDescription()`** (section 2.1.2)

Line / Block of code	Description
`Node* createNode(const char* key, const char* description) {`	The function takes pointers to the key and description as parameters, and returns a pointer to the new node.
`Node* newNode = (Node*)malloc(sizeof(Node));`	It dynamically allocates memory to the new node using **`malloc()`**.
`if (newNode == NULL) {` `  printf("Error 1.1: Memory allocation failed =<\n");` `  return NULL;` `  } else {` `  printf ("Memory allocation successful =D\n");`	If memory allocation fails, an error message is printed. If memory allocation is successful, it prints: >> “Memory allocation successful =D”

`    }`	
`        strcpy (newNode->key, key);` `        strcpy (newNode->description, description);`	The key and the description are copied into the new node.
`    newNode->left = NULL;` `    newNode->right = NULL;`	The function sets the pointers to the left and right children to NULL, ensuring the node starts as a leaf.
`    return newNode;` `}`	Lastly, it returns the pointer to the newly created node.

2.1.1 isValidKey

Functionality: **`isValidKey()`** checks is a key does not exceed the character limit, and has a valid format (is a single word, has no spaces, no ':' or '.')

Line / Block of code	Description
`int isValidKey(const char* key) {`	The function takes a pointer to a given key and integer (0 for false, not valid, and 1 for true, valid)
`if (key == NULL || strlen(key) == 0) {` `        printf("Error 1.2.4: Key is empty =< Try again\n");` `        return 0;` `    }`	If the key is NULL or empty, an error message is printed and the function returns 0 (not valid).
`if (strlen(key) > 29) {` `        printf("Error 1.2.1: Key has more than 29 characters =< Try again\n");` `        return 0;` `    }`	If the key exceeds the character limit, an error message is printed and the function returns 0 (not valid).
`for (int i = 0; key[i] != '\0'; i++) {` `        if (key[i] == ' ') {` `            printf("Error 1.2.2: Key contains space(s) =< Try again\n");` `            return 0;` `        } else if (key[i] == ':' || key[i] == '.') {` `            printf("Error 1.2.3: Key contains invalid character(s) =< Try again\n");` `            return 0;` `        }` `}`	The function goes through all the characters in the key. If one of them is a space, a colon, or a dot, an error message is printed and the function returns 0 (not valid).
`   return 1;` `}`	If all checks are passed, the function returns 1 (valid).

2.1.2 isValidDescription

Functionality: **`isValidDescription()`** checks is a description does not exceed the character limit, and has a valid format (has no ':' or '.')

Line / Block of code	Description
`int isValidDescription(const char* description)` `{`	The function takes a pointer to a given description and integer (0 for false, not valid, and 1 for true, valid)
`if (description == NULL || strlen(description) == 0) {` `        printf("Error 1.3.3: Description is empty =< Try again\n");` `        return 0;` `    }`	If the description is NULL or empty, an error message is printed and the function returns 0 (not valid).
`if (strlen(description) > 199) {` `        printf("Error 1.3.1: Description has more than 199 characters =< Try again\n");`	If the description exceeds the character limit, an error message is printed and the function returns 0 (not valid).

`        return 0;` `    }`	
`for (int i = 0; description[i] != '\0'; i++) {` `        if (description[i] ==':' ||` `description[i] == '.') {` `            printf("Error 1.3.2: Description` `contains invalid character(s) =< Try again\n");` `            return 0;` `        }` `    }`	The function goes through all the characters in the description. If one of them is a colon, or a dot, an error message is printed and the function returns 0 (not valid).
`   return 1;` `}`	If all checks are passed, the function returns 1 (valid).

2.2 printNode

Functionality: **`printNode()`** prints the key and description for a given node.

Line / Block of code	**Description**
`void printNode(Node* node) {`	The function takes a pointer to the node as parameter and does not return anything.
`if (node == NULL) {` `        printf("Error 2.1: The node is NULL` `=<\n");` `    } else {` `        printf("%s:%s.\n", node->key, node-` `>description);` `    }` `}`	If the node is NULL, an error message is printed. If it is not, it prints the key and description.

2.3 createTree

Functionality: **`createTree()`** creates a new, empty tree and initializes its root to NULL. It does not take any parameters as none are needed to create the new tree.

Line / Block of code	**Description**
`Tree* createTree() {`	The function returns a pointer to the new tree.
`   Tree* newTree = (Tree*)malloc(sizeof(Tree));`	It dynamically allocates memory to the new tree using **`malloc().`**
`   if (newTree == NULL) {` `        printf("Error 3.1: Memory allocation` `failed =<\n");` `        return NULL;` `    }`	If memory allocation fails, an error message is printed and the function returns NULL to prevent memory issues.
`    newTree->root = NULL;`	The function sets the root to NULL, ensuring the tree starts empty.
`    return newTree;` `}`	Lastly, it returns the pointer to the newly created tree.

2.4 treeEmpty

Functionality: **`treeEmpty()`** checks whether a given tree is empty or not.

Line / Block of code	**Description**
`int treeEmpty(Tree* tree) {`	The function takes a pointer to the tree as parameter and returns an integer (either 0 or 1).
`  if (tree == NULL || tree->root == NULL) {` `        return 1;` `    }` `    return 0;` `}`	If the tree's root is NULL (the tree is empty), the function returns 1 (true). Otherwise, it returns 0 (false).

2.5 insertNode

Functionality: **`insertNode()`** inserts a new node into a given tree. The new node either becomes the root if the tree is empty, or is placed in the correct position based on alphabetical order. If the new node is a duplicate, it is discarded.

0.	Line / Block of code	Description
	`void insertNode(Tree* tree, Node* newNode) {`	The function takes pointers to the tree and new node to be inserted, and does not return anything.
	`if (tree->root == NULL){` `tree->root = newNode;` `return;` `}`	If the root of the tree is NULL, meaning the tree is empty, the new node becomes the root. The function then terminates.
	`Node* current = tree->root;` `Node* parent = NULL;`	The function uses two pointers: >> *current,* which keeps track of the node being compared as it searches for a place to insert the new node, starting from the root of the tree >> *parent,* which keeps track of the previously visited node as it searches for a place to insert the new node
	`while (current != NULL){` `parent = current;`	The function loops until the current node is NULL, meaning a free space is available for the new node. The parent is updated to remember the last node before moving left or right.
	`if (strcmp(newNode->key, current->key) < 0){` `current = current->left;` `if (current == NULL){` `parent->left = newNode;` `return;` `}`	If the new node's key is smaller than the current node's key (since they are strings, it is using **`strcmp()`** to compare the ASCII value of the letters, in other words checks which one is first in alphabetical order), the left child becomes the current node. If the left child is NULL, the function puts the new node there. If the node is placed, the function terminates.
	`} else if (strcmp(newNode->key, current->key) > 0){` `current = current->right;` `if (current == NULL){` `parent->right = newNode;` `return;` `}`	Similarly, if the new node's key is greater than the current node's key, it moves to the right and places the new node there if the right child is empty. If the node is placed, the function terminates.
	`} else {` `return;` `}` `}` `}`	If the two keys are identical, it means the node already exists in the tree and the function terminates.

2.6 search

Functionality: **`search()`** looks for a node with the given key within the tree. It goes from node to node and checks if its key is the same as the given one. If the key is found, the function returns the node. Otherwise, it returns NULL.

Line / Block of code	Description
`Node* search(Tree* tree, char* key) {`	The function takes one pointer for the tree and one for the key to be searched, and returns the node with the key if found.
`Node* current = tree->root;`	The function uses the pointer current to traverse the nodes of the tree and starts from the root.
`while (current != NULL){` `if (strcmp(key, current->key) < 0) {` `current = current->left;`	Then, the function loops while the current node is not NULL, comparing the key and the current node's key,

`        } else if (strcmp(key, current->key) > 0) {` `            current = current->right;` `        } else {` `            return current;` `        }` `    }`	using **`strcmp()`**. If the key is smaller, it means it is in the left subtree of the current node, so the function moves to the left. If the key is greater, it means it is in the right subtree of the current node, so the function moves to the right. If the two keys are the same, it means the node was found and it is returned.
`return NULL;` `}`	If the loop terminates and no node has the same key, the function returns NULL as the key was not found.

2.7 delete

Functionality: **`delete()`** eliminates a node with the given key and re-arranges the nodes in the tree if necessary. The function tackles 6 possible situations:

1. delete root node with no children >> set the tree to empty
2. delete root node with one child >> set the node to its child
3. delete root node with two children >> find the inorder successor, copy it to the root, adjust tree structure, and delete the inorder successor
4. delete leaf node >> set the node's parent left/right child to NULL
5. delete node with left/right child >> set the node's parent left/right child to the node's left/right child
6. delete node with two children >> find the inorder successor, copy it to the node, adjust tree structure, and delete the inorder successor

Line / Block of code	**Description**
`void delete(Tree* tree, char* key) {`	The function takes one pointer for the tree and one for the key of the node to be deleted.
`   Node* byeNode = search(tree, key);`	The function uses the **`search()`** function (section 2.7) to look for the node with the given key. The output of the search function is passed to byeNode.
`   if (byeNode == NULL){` `        printf("Error 7.1: Key not found =<\n");` `        return;` `    }`	If byeNode is NULL, it means the key was not found and there is no node to delete. The function terminates here in this case.
`    Node* parent = NULL;` `    Node* current = tree->root;`	The function uses parent to track the parent of the node to be deleted and current to traverse through the tree while looking for the parent. The traversal starts with current, from the root-tree, and parent starts as NULL since the root has no parents.
`    while (current != byeNode && current != NULL){` `        parent = current;` `        if (strcmp(key, current->key) < 0){` `            current = current->left;` `        } else {` `            current = current->right;` `        }` `    }`	The function loops through the nodes searching for the key, with the scope of getting the parent of the node. As long as the current node is not the one to be deleted and is not NULL, the parent becomes the current, and current moves either to its left or right child, depending on its position relative to the key.
`    if (byeNode->left == NULL && byeNode->right == NULL){` `        if (parent == NULL){` `            tree->root = NULL;` `        } else if (parent->left == byeNode){` `            parent->left = NULL;` `        } else {` `            parent->right = NULL;` `        }` `        free(byeNode);`	(case 1) If the node to be deleted has no children and its parent is also NULL, it means the node is the root and it just needs to be set as NULL. (case 4) If the node to deleted is a leaf and is the left child of its parent, the left child of the parent is set to NULL. Similarly, if the node is a right child, the right child of the parent is set to NULL. Memory is then freed of the deleted node.

`} else if (byeNode->left != NULL && byeNode->right == NULL){` `if (parent == NULL){` `tree->root = byeNode->left;` `} else if (parent->left == byeNode){` `parent->left = byeNode->left;` `} else {` `parent->right = byeNode->left;` `}` `free(byeNode);`	(case 2) If the node to be deleted has only a left child, and its parent is null, it means the node is the root and it just needs to be set as the left child. (case 5) If the node to be deleted has a parent and is the left child, its left child becomes the left child of the parent. Similarly, if the node is the right child, its left child becomes the right child of its parent. Memory is then freed of the deleted node.
`} else if (byeNode->left == NULL && byeNode->right != NULL){` `if (parent == NULL){` `tree->root = byeNode->right;` `} else if (parent->left == byeNode){` `parent->left = byeNode->right;` `} else {` `parent->right = byeNode->right;` `}` `free(byeNode);`	(case 2) If the node to be deleted has only a right child, and its parent is null, it means the node is the root and it just needs to be set as the right child. (case 5) If the node to be deleted has a parent and is the left child, its right child becomes the left child of the parent. Similarly, if the node is the right child, its right child becomes the right child of its parent. Memory is then freed of the deleted node.
`} else {` `Node* nextParent = byeNode;` `Node* next = byeNode->right;` `while (next->left != NULL){` `nextParent = next;` `next = next->left;` `}`	If the node has two children, it needs to be replaced by its inorder successor (leftmost descendent in the node's right subtree). For this, the function needs to traverse the right subtree and keep track of the next parent and the next current node. It starts from the right child of the node to be deleted. It then goes to the leftmost descendent possible.
`strcpy(byeNode->key, next->key);` `strcpy(byeNode->description, next->description);`	(case 3) Then the function copies the key and description of the leftmost descendent to the node to be deleted. In other words, the node is replaced by the descendent.
`if (nextParent->left == next){` `nextParent->left = next->right;` `} else {` `nextParent->right = next->right;` `}` `free(next);` `}` `}`	(case 6) If the node is not the root, its parent's pointer to it is set to the leftmost descendent, which is then deleted. Memory is then freed of the deleted node.

2.8 inorderPrint

Functionality: **`inorderPrint()`** prints at most a given number (n) of nodes from the tree in alphabetical order. It does this using an aiding function, **`inorderPrintHelper()`** (section 2.8.1), which does an inorder traversal of the tree.

Line / Block of code	**Description**
`void inorderPrint(Tree* tree, int n){`	The function takes a pointer to the tree and an integer n, which is the maximum number of nodes to be printed.
`int count = 0;`	The function needs a counter that will start from 0 to keep track of the number of printed nodes. This way it can stop when the maximum number of nodes is reached.
`if (treeEmpty(tree)) {` `printf("Error 8.1: The tree is empty =<\n");` `return;` `}`	If the tree is empty (**`treeEmpty()`** function (see section 2.4) returns 1), the function terminates as there is nothing to traverse or print.
`inorderPrintHelper(tree->root, &count, n);` `}`	Lastly, it calls the helper function which takes three parameters:

	>> the root of the tree, to start the traversal >> a pointer to the counter so it can be updated across the recursive calls >> n for making sure the printing stops when the limit is reached

2.8.1 inorderPrintHelper

Functionality: **`inorderPrintHelper()`** uses recursion to traverse the tree and print a given number (n) of nodes in alphabetical order.

Line / Block of code	Description
`void inorderPrintHelper(Node* node, int* count, int n){`	The function takes a pointer to a given node, a pointer to the counter set in the **`inorderPrint()`** function (section 2.8), and n, the number of nodes to be printed, also given in the **`inorderPrint()`** function (section 2.8).
`if (node == NULL || *count == n){` `return;` `}`	If the node is NULL, or if the counter reached the limit, n, meaning n nodes have already been printed, the function terminates.
`inorderPrintHelper(node->left, count, n);`	First, the function calls itself for the left child of the node. Here it can either go further to the children of the current left child and so on, or come back and go to the next step.
`if (*count < n){` `printNode(node);` `(*count)++;` `}`	Then, in the next step, it checks if more nodes can be printed. If so, it prints the current node, increases the counter by one, and goes to the last step. Otherwise, it goes directly to the last step.
`if (*count < n) {` `inorderPrintHelper(node->right, count, n);` `}` `}`	Ultimately, in the last step, if more nodes can be printed, the function calls itself for the right child. Here it can either go further to the children of the current right child and so on, or terminate.

2.9 loadTextFile

Functionality: **`loadTextFile()`** reads the contents of a given file, line by line, extracts the key and description from each line (if the line follows the format “Key:Description.”), uses them to create a node, and inserts the node in the given tree. The function tackles 8 possible error than can occur due to the line format:

1. the line has no key / description
2. the line has an empty key / description
3. the line does not have a ‘:’ / ‘.’/ both
4. the line has more than one ‘:’ / ‘.’ / both
5. the line is blank
6. the line does not end with a ‘.’
7. the line has a space before / after / before and after the ‘:’
8. the line has a key that contains a space (is not a single word)

Line / Block of code	Description
`void loadTextFile(Tree* tree, char* filename) {`	The function takes a pointer to the tree and the name of the file to be read.
`FILE* fp;` `fp = fopen(filename, "r");` `if (fp == NULL) {`	The function uses a pointer to the file which takes the file opened in reading mode. If the pointer is NULL, no file has been passed to it so the file was not found and an error message is printed. The function terminates.

`printf("Error 9.1: File not found =<\n");` `return;` `}`	
`char line[500];` `while (fgets(line, sizeof(line), fp)){` `line[strcspn(line, "\n")] = '\0';`	The function takes a variable for a line to navigate through the lines of the file (I chose size 500 to make sure there is enough for any line). It loops through the lines of the file and removes the newline character at their end.
`if (strlen(line) == 0) {` `printf("Error 9.2: Blank line =<\n");` `continue;` `}`	(case 5) If a line has no length, it means it is a blank line, which is not allowed so the function skips it and moves on.
`int colonCount = 0;` `int dotCount = 0;` `for (int i = 0; line[i] != '\0'; i++){` `if (line[i] == ':'){` `colonCount++;` `}` `if (line[i] == '.'){` `dotCount++;` `}` `}` `if (colonCount != 1 || dotCount != 1) {` `printf("Error 9.3.1: Invalid format --> Incorrect number of `:` or `.` =<\n");` `continue;` `}`	The function takes 2 counters, one for the ":" and one for the "." in the line. It goes through each element in the line and counts how many colons and dots are. (case 4) If there are more than one ":" or more than one ".", the format is not valid, an error message is printed, and the function moves on to the next line.
`char *colonPosition = strchr(line, ':');` `char *dotPosition = strchr(line, '.');`	The function takes two pointers, one for the position of the ':' in the line, and one for the '.' which will be used to check the format of the line and its validity.
`if (!colonPosition || !dotPosition || dotPosition != &line[strlen(line) - 1]) {` `printf("Error 9.3.2: Invalid format --> Missing or misplaced `:` or `.` =<\n");` `continue;` `}`	(case 3 and 6) If either the ':' or the '.' is missing, or if the '.' is not the last element in the line, the format is not valid, an error message is printed, and the function moves on to the next line.
`if (colonPosition == line || *(colonPosition - 1) == ' ' || *(colonPosition + 1) == ' ') {` `printf("Error 9.3.3: Invalid format --> Spaces around `:` =<\n");` `continue;` `}`	(case 7) If the ':' exists but there is at least one space either before or after it, the format is not valid, an error message is printed, and the function moves on to the next line.
`char *key, *description;` `key = strtok(line, ":");` `description = strtok(NULL, ".");`	The function then extracts the key and description of the node from the line. The key should be everything from the start of the line until the ':', and the description should be everything after the ':' and before the '.' (which is supposed to be the end of the line).
`if (!key || !description || strlen(key) == 0 || strlen(description) == 0 || strchr(key, ' ')) {` `printf("Error 9.3.4: Invalid format --> Missing or invalid key or description =<\n");` `continue;` `}`	(case 1, 2, and 8) If, for either the description or the key, nothing is passed, or their length is 0, or if the key contains a space, meaning it is not a single word, the format is not valid, an error message is printed, and the function moves on to the next line.
`Node* newNode = createNode(key, description);`	The function creates a node using the **`createNode()`** function (section 2.1) which takes the key and node as parameters.

`if (newNode == NULL) {` `printf("Error 9.4: Memory allocation failed =<\n");` `continue;` `}`	If the newly created node is NULL, there was an issue with memory allocation and an error message is printed.
`insertNode(tree, newNode);` `}`	The node is inserted into the tree using the **`insertNode()`** function (section 2.5).
`fclose(fp);` `}`	The function closes the file.

2.10 storeTextFile

Functionality: **`storeTextFile()`** writes, in a given file, the given tree in preorder. It follows the same line format as the one in **`loadTextFile`** function (section 2.9). It also uses a helper function, **`writeInPreorder`** (section 2.10.1) which uses recursion to traverse the tree in preorder.

Line / Block of code	Description
`void storeTextFile(Tree* tree, char* filename) {`	The function takes a pointer to the tree and the name of the file to be written to.
`FILE* fp;` `fp = fopen(filename, "w");` `if (fp == NULL){` `printf("Error 10.1: File unable to open =<\n");` `return;` `}`	The function uses a pointer to the file which takes the file opened in writing mode. If the pointer is NULL, no file has been passed to it so the file was not found and an error message is printed. The function terminates.
`writeInPreorder(tree->root, fp);`	The function calls the helper function, **`writeInPreorder()`** (section 2.10.1) to write the tree in preorder traversal starting from the root.
`fclose(fp);` `}`	The function closes the file.

2.10.1 writeInPreorder

Functionality: **`writeInPreorder()`** uses recursion to write (in the given file) the given node and its children in preorder traversal (node, left subtree, right subtree).

Line / Block of code	Description
`void writeInPreorder(Node* node, FILE* fp){`	The function takes a pointer to the node being visited and the name of the file to be written to.
`if (node == NULL || fp == NULL){` `return;` `}`	If the node or the file is NULL, it means there is no starting point for this traversal or there is no file to write to, so the function terminates.
`fprintf(fp, "%s:%s.\n", node->key, node->description);`	The function prints the key and description of the node following the format “Key:Description.” from the **`loadTextFile`** function (section 2.9) .
`writeInPreorder(node->left, fp);` `writeInPreorder(node->right, fp);` `}`	Then, the function calls itself for the left child, then the right child. This way, the traversal will visit the node, left child, then right child for each node, performing the preorder traversal.

2.11 freeTree

Functionality: **`freeTree()`** prevents memory leaks by freeing the tree of all existing nodes. It uses an aiding function **`freeTreeHelper()`** (section 2.11.1).

Line / Block of code	Description
`void freeTree(Tree* tree) {`	The function takes a pointer to the tree to be freed.
`    if (treeEmpty(tree)){` `        return;` `    }`	If the tree is empty (**`treeEmpty()`** function (see section 2.4) returns 1), there is nothing to free and the function terminates.
`    freeTreeHelper(tree->root);` `    tree->root = NULL;` `}`	The function calls the helper function **`freeTreeHelper()`** (section 2.11.1) to start the freeing from the root. Then it resets the tree as empty, making sure the root is NULL.

2.11.1 freeTreeHelper

Functionality: **`freeTreeHelper()`** uses recursion to free each given node and its descendants to prevent memory leaks.

Line / Block of code	Description
`void freeTreeHelper(Node* node){`	The function takes a pointer to the node to be freed.
`    if (node == NULL){` `        return;` `    }`	If the node is NULL, it means there is nothing to free and the function terminates.
`    freeTreeHelper(node->left);` `    freeTreeHelper(node->right);`	The function calls itself for the node's left child, then right child. This way the all children are freed before their parents so no nodes are lost.
`    free(node);` `}`	Lastly, the function frees the node.

2.12 main

Functionality: **`main()`** is menu based and allows the user to perform operations on nodes or tree until they exit the program (or a fatal error occurs, like not enough space to allocate). It also handles errors (section 3.1.8 for details).

Line / Block of code	Description
`int main() {` `    Tree* newTree = createTree();` `    int choice = -1;` `        printf("Welcome to the Binary Search Tree program =D\n");` `    while(1){` `        printf("\nMenu:\n");` `        printf("1: CREATE NODE\n");` `        printf("2: PRINT NODE\n");` `        printf("4: TREE EMPTY\n");` `        printf("6: SEARCH\n");` `        printf("7: DELETE\n");` `        printf("8: INORDER PRINT\n");` `        printf("9: LOAD TEXT FILE\n");` `        printf("10: STORE TEXT FILE\n");` `        printf("11: FREE TREE\n");` `        printf("0: EXIT\n");`	The function takes: Creates the tree and takes an integer, **`choice,`** which is -1 by default and with which the user's option for the operation to be performed is handled. The function prints the welcome message at the beginning. Then, it prints the menu and asks the user what would they like to do and for a choice each time an operation ends (unless a fatal error is thrown). The function uses **`while(getchar() != '\n')`** to clear the input buffer. Then, based on the choice provided by the user, the function enters a case.

```
        printf("\nWhat would you like to do? Please enter your
choice:\n");
        scanf("%d", &choice);

        while(getchar() != '\n');

        switch(choice)
        {
```

```
case 1:
                char case1Key[30];
                char case1Description[200];

                printf("Please enter the key(a single word, no
spaces, no ':' or '.', max 29 characters):\n");

                while(1) {
                    fgets(case1Key, sizeof(case1Key), stdin);
                    case1Key[strcspn(case1Key, "\n")] = '\0';
                    if (isValidKey(case1Key)) {
                        break;
                    }
                }

                if (search(newTree, case1Key) != NULL) {
                    printf("Error 12.1.1: Node with %s key
already exists =<\n", case1Key);
                    break;
                }

                printf("Please enter the description (no ':'
or '.', max 199 characters):\n");

                while(1) {
                    fgets(case1Description,
sizeof(case1Description), stdin);
                    case1Description[strcspn(case1Description,
"\n")] = '\0';
                    if (isValidDescription(case1Description))
{
                        break;
                    }
                }

                Node* newNode = createNode(case1Key,
case1Description);
                insertNode(newTree, newNode);

                break;
```

Case 1 is responsible for **createNode** function (section 2.1).

It takes two char variables, one for the key and one for the description.

Then it prints a message asking the user to input a key. If the key is not valid (checked using the **isValidKey** function (section 2.1.1), it keeps asking until a valid one is entered.

Then, it checks if a node with the same key already exists in the tree using **search** function (section 2.6). If it does, it prints an error message and exits.

If no node with the key exists yet, it asks the user to enter a description. Just like with the key, it keeps asking until the input is valid (checked using **isValidDescription** function (section 2.1.2)).

Finally, if both inputs are valid, the new node is created using **createNode** function (section 2.1) and inserted using **insertNode** function (section 2.5) into the tree.

```
case 2:
                char case2Key[30];

                printf("Please enter the key of the node to be
printed:\n");
                fgets(case2Key, sizeof(case2Key), stdin);
                case2Key[strcspn(case2Key, "\n")] = '\0';

                Node* case2Node = search(newTree, case2Key);
                if (case2Node == NULL) {
                    printf("Error 12.2.1: No node with '%s'
key was found =<\n", case2Key);
                    break;
                } else {
                    printNode(case2Node);
                    break;
                }

                break;
```

Case 2 is responsible for **printNode** function (section 2.2).

It takes a char variable for the key. Then it prints a message asking the user to input the key of the node to be printed.

It searches for the node with the given key in the tree using **search** function (section 2.6). If no such node is found, an error message is printed. If the node is found, it is printed using the **printNode** function (section 2.2).

```
case 4:
                if (treeEmpty(newTree)) {
                    printf("The tree is empty.\n");
                } else {
                    printf("The tree is not empty.\n");
                }

                break;
```

Case 4 is responsible for **treeEmpty** function (section 2.4).

The appropriate message is printed depending on what the function returns.

```
case 6:
                if (treeEmpty(newTree)) {
                    printf("Error 12.6.1: The tree is empty
=<\n");
                    break;
                }

                char case6Key[30];
                Node* case6Node = NULL;

                printf("Please enter the key of the node to be
searched:\n");
                fgets(case6Key, sizeof(case6Key), stdin);
                case6Key[strcspn(case6Key, "\n")] = '\0';

                case6Node = search(newTree, case6Key);

                if (case6Node != NULL) {
                    printNode(case6Node);
                } else {
                    printf("Error 12.6.2: Node with key '%s'
does not exist in the tree =<\n", case6Key);
                }

                break;
```

Case 6 is responsible for **search** function (section 2.6).

It checks if the tree is empty using **treeEmpty** function (section 2.4). If it is, an error message is printed and the function exits as there is nothing to search.

It takes a node and a char variable for the key of the node to be searched for. Then it asks the user to input the key.

It searches for the node with the given key in tree using the **search** function (section 2.6) and stores the result in the node variable. If it exists, the node is printed using the **printNode** function (section 2.2). If no such node exists in the tree, an error message is printed.

```
case 7:
                if (treeEmpty(newTree)) {
                    printf("Error 12.7.1: The tree is empty
=<\n");
                    break;
                }

                char case7Key[30];

                printf("Please enter the key of the node to be
deleted:\n");
                fgets(case7Key, sizeof(case7Key), stdin);
                case7Key[strcspn(case7Key, "\n")] = '\0';

                if (search(newTree, case7Key) != NULL) {
                    delete(newTree, case7Key);
                    printf("Node successfully deleted =D\n");
                } else {
                    printf("Error 12.7.2: Node with key '%s'
does not exist in the tree =<\n", case7Key);
                }

                break;
```

Case 7 is responsible for **delete** function (section 2.7).

It checks if the tree is empty using **treeEmpty** function (section 2.4). If it is, an error message is printed and the function exits as there is nothing to delete.

It takes a char variable for the key of the node to be deleted. Then it asks the user to input the key.

It searches for the node with the given key in tree using the **search** function (section 2.6) and if it exists, it deletes the node using the **delete** function (section 2.7). If no such node exists in the tree, and error message is printed.

```
case 8:
                if (treeEmpty(newTree)) {
                    printf("Error 12.8.1: The tree is empty
=<\n");
                    break;
                }

                int n;

                printf("Please enter the maximum number of
nodes from the tree to be printed inorder:\n");
```

Case 8 is responsible for inorderPrint function (section 2.8).

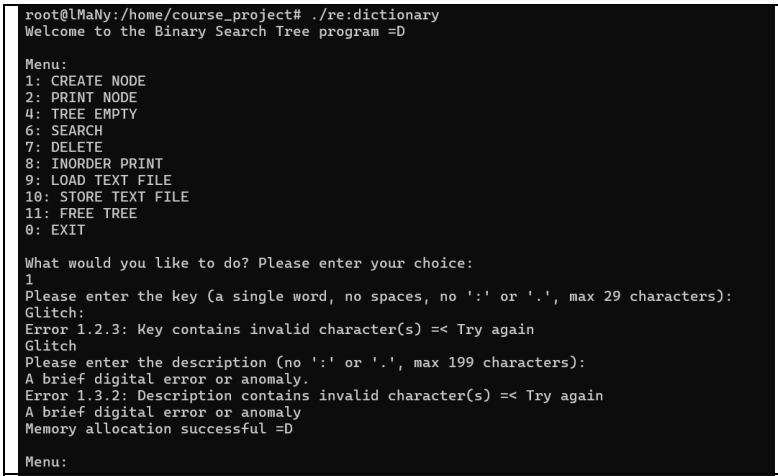
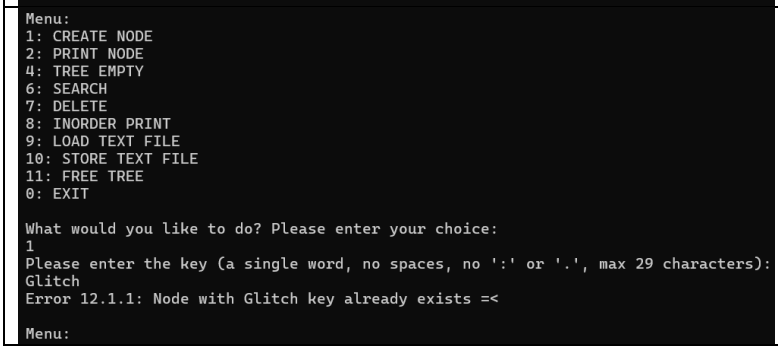
It checks if the tree is empty using **treeEmpty** function (section 2.4). If it is, an error message is printed and the function exits as there is nothing to print.

`` scanf("%d", &n); inorderPrint(newTree, n); break; ```	It takes an int variable for the maximum number of nodes to be printed. Then it asks the user to input the number. It prints the nodes using the **inorderPrint** function (section 2.8).
`` case 9: char case9File[100]; printf("Please enter the .txt file to be loaded:\n"); fgets(case9File, sizeof(case9File), stdin); case9File[strcspn(case9File, "\n")] = '\0'; loadTextFile(newTree, case9File); break; ```	Case 9 is responsible for the **loadTextFile** function (section 2.9). It takes a char variable for the name of the file to be loaded. Then it asks the user to input the file name. It calls the **loadTextFile** function (section 2.9) which tries to load the data from the file into the tree.
`` case 10: if (treeEmpty(newTree)) { printf("Error 12.10.1: The tree is empty =<\n"); break; } char case10File[] = "storage.txt"; storeTextFile(newTree, case10File); printf("Tree successfully stored =D\n"); break; ```	Case 10 is responsible for the **storeTextFile** function (section 2.10). It checks if the tree is empty using **treeEmpty** function (section 2.4). If it is, an error message is printed and the function exits as there is nothing to store. It takes a char variable for the name of the file where the data from the tree will be stored. It calls the **storeTextFile** function (section 2.10) which tries to store the data from the tree into the storage.txt file.
`` case 11: if (treeEmpty(newTree)) { printf("Error 12.11.1: The tree is empty =<\n"); break; } freeTree(newTree); printf("Tree successfully freed =D\n"); break; ```	Case 11 is responsible for the **freeTree** function (section 2.11). It checks if the tree is empty using **treeEmpty** function (section 2.4). If it is, an error message is printed and the function exits as there is nothing to free the tree from. It calls the **freeTree** function (section 2.11) which frees the tree of all nodes. Then an appropriate message is printed.
`` case 0: free(newTree); printf("Thank you for using the program. Bye!\n"); return 0; ```	Case 0 is responsible for exiting. It frees the tree using the **freeTree** function (section 2.11) to release the used memory. Then it thanks the user for using the program.
`` default: printf("Error 12.0: Invalid choice =< Please choose a valid number\n"); break; } } return 0; } ```	Default is responsible for handling the situation in which the input choice is not valid (anything other than 1, 2, 4, 6, 7, 8, 9, 10, 11, 0).

3 Testing

For testing, I started by running each function right after implementing it (and its necessary helper function(s) where it was the case) and trying to make sure it worked well. After that, I tried to catch all the identified errors (section 3.7) (except those that would be thrown in case of failed memory allocation). Then, I tested the entire program and focused on the 6 cases described in sections 3.1 – 3.6. Naturally, I also did some small tests here and there when fixing issues along the way.

3.1 Inserting nodes

	Inserting a node test (and invalid input): In this test, I chose option 1 for creating a node. Then I typed an invalid key (I put ":" at the end), got an error message and had to try again. I retyped the key correctly, then typed an invalid description (put "." at the end). Again, an error message was shown and I retyped the description correctly. Once the two were valid, the new node was created and inserted into the tree, proof being the "Memory allocation successful" message.
	Inserting a duplicate node test: After inserting the node with the "Glitch" key, I attempted to do it again. This time, an error message was shown saying that a node with the same key already exists in the tree, and the function exited. So, the program successfully prevents the user from inserting duplicate nodes in the tree.

3.2 Searching for nodes

```
Menu:
1: CREATE NODE
2: PRINT NODE
4: TREE EMPTY
6: SEARCH
7: DELETE
8: INORDER PRINT
9: LOAD TEXT FILE
10: STORE TEXT FILE
11: FREE TREE
0: EXIT

What would you like to do? Please enter your choice:
6
Please enter the key of the node to be searched:
Glitch
Glitch:A brief digital error or anomaly.

Menu:
1: CREATE NODE
2: PRINT NODE
4: TREE EMPTY
6: SEARCH
7: DELETE
8: INORDER PRINT
9: LOAD TEXT FILE
10: STORE TEXT FILE
11: FREE TREE
0: EXIT

What would you like to do? Please enter your choice:
6
Please enter the key of the node to be searched:
Cyberware
Error 12.6.2: Node with key 'Cyberware' does not exist in the tree =<
```

Searching for an existing node:
In this test, I chose option 6 for searching for a node in the tree. Then I typed the key of a node I previously created, “Glitch”. The key was a valid one and the node was found in the tree, so the function printed the node with the key and description.

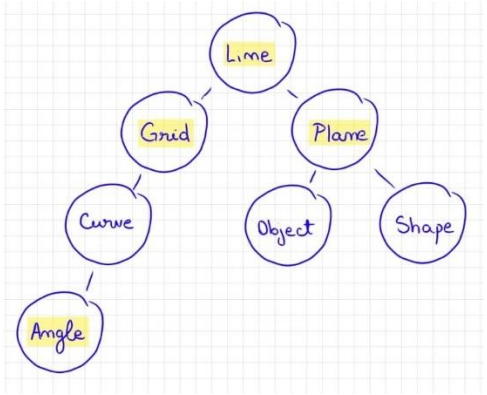
Searching for a non-existing node:
Then, I typed a key that does not belong to any node in the tree, “Cyberware”, and an error message was printed showing that no such node exists.

3.3 Deleting nodes

For this test I used the storeTextFile function and the fact that it stores nodes in preorder. This way I can see how nodes are stored, draw the tree, and test deletion with different kinds of nodes. Also, it is easy to see if nodes are stored correctly after deleting one.

```
1   Line:A straight path between two points.
2   Grid:A pattern of intersecting lines forming squares or rectangles.
3   Curve:A smooth, bending line.
4   Angle:The space between two intersecting lines.
5   Plane:A flat, two-dimensional surface.
6   Object:A three-dimensional form or item.
7   Shape:A two-dimensional form or item.
8
```

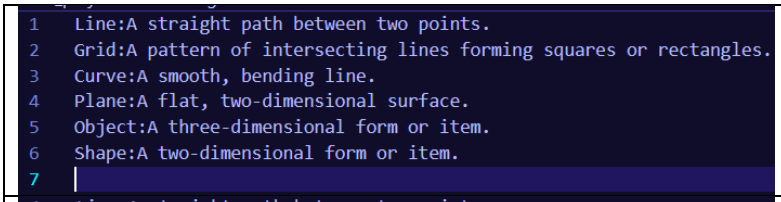
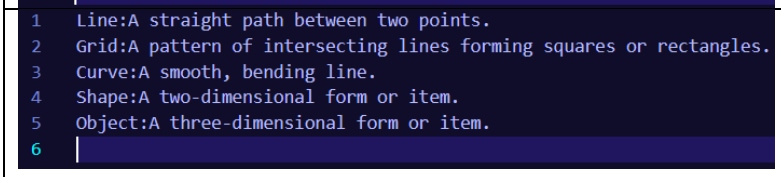
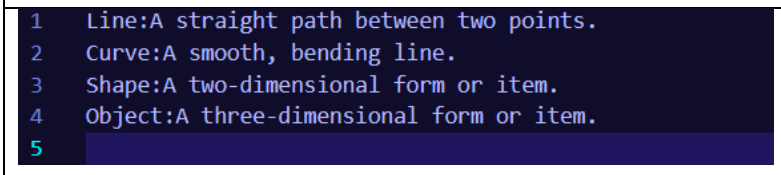
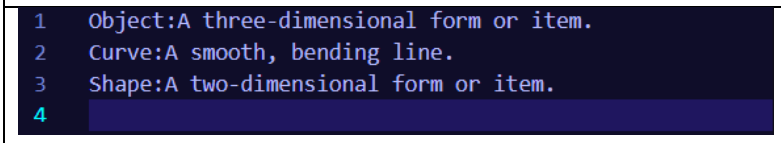
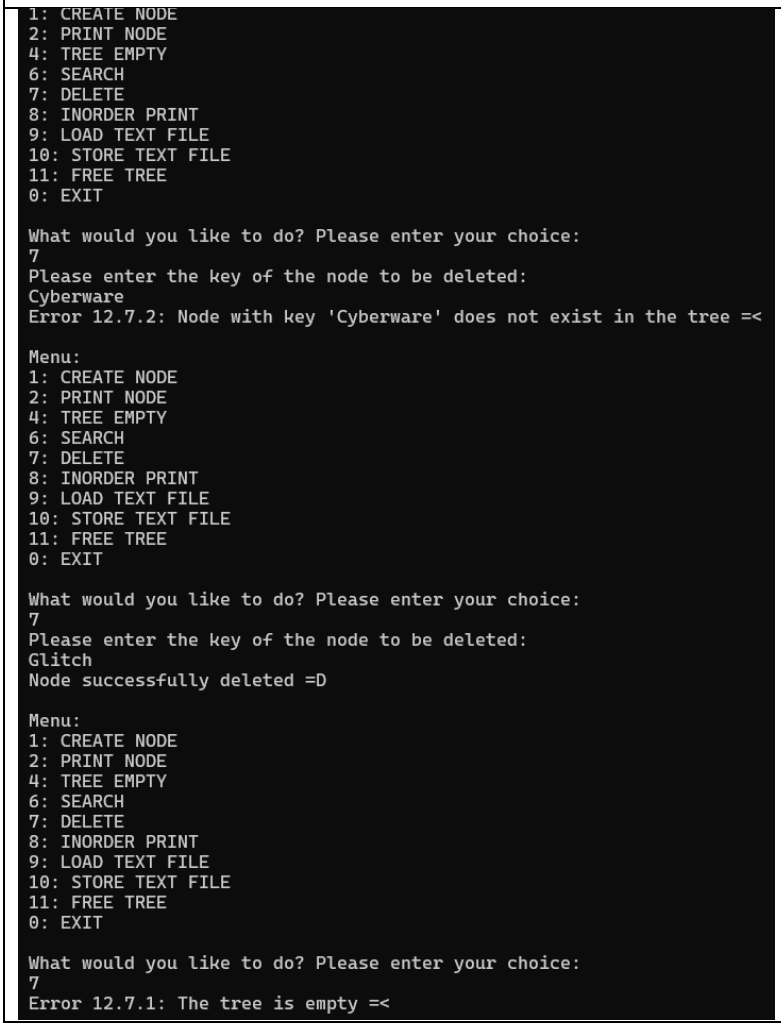
^ Picture 1



^ Picture 2

I started by inserting a bunch of nodes into the tree. After that I stored them in the text file. The preorder looked like this (see picture 1), meaning the tree would look like this (see picture 2)

The nodes I targeted for deletion are:
>> Angle (leaf node)
>> Plane (node with 2 children)
>> Grid (node with one child)
>> Line (root node)

	Deleting a leaf node: After deleting the leaf node, the tree says the same as no rearrangements are needed. This can be seen as the preorder only eliminated the node, the rest of them staying the same.
	Deleting a node with two children: After deleting the node with two children, the smallest node on the right subtree of the deleted one (in this example node with key “Shape”) takes the place of the deleted node. As proof, the preorder has changed, “Shape” being now the right child of “Line” (the root).
	Deleting a node with one child: After deleting the node with one child, the child takes its place in the tree (in this example node with key “Grid” is replaced by node “Curve”, which becomes the left child of root).
	Deleting the root: After deleting the root, the smallest node in its right subtree (in this example node “Object”) becomes the root. In this case, it was similar to deleting a node with 2 children, because the root had two children.
	Deleting a non-existing node: In this test, I chose option 7 for deleting a node in the tree. Then, I typed a key that does not belong to any node in the tree, “Cyberware”, and an error message was printed showing that no such node exists. Deleting an existing node: Then I typed the key of a node I previously created, “Glitch”. The key was a valid one and the node was found in the tree, so the function deleted the node. Trying to delete when try is empty: Then, I tried again to delete a node, but as the tree is empty and there is nothing to delete, an error message was shown.

3.4 Loading and storing files

```
Menu:
1: CREATE NODE
2: PRINT NODE
4: TREE EMPTY
6: SEARCH
7: DELETE
8: INORDER PRINT
9: LOAD TEXT FILE
10: STORE TEXT FILE
11: FREE TREE
0: EXIT

What would you like to do? Please enter your choice:
9
Please enter the .txt file to be loaded:
somefile.txt
Error 9.1: File not found =<

Menu:
1: CREATE NODE
2: PRINT NODE
4: TREE EMPTY
6: SEARCH
7: DELETE
8: INORDER PRINT
9: LOAD TEXT FILE
10: STORE TEXT FILE
11: FREE TREE
0: EXIT

What would you like to do? Please enter your choice:
9
Please enter the .txt file to be loaded:
dictionary.txt
Memory allocation successful =D
Memory allocation successful =D
Memory allocation successful =D
Memory allocation successful =D
```

...

```
Memory allocation successful =D
Error 9.3.4: Invalid format --> Missing or invalid key or description =<
Memory allocation successful =D
Memory allocation successful =D
```

...

Loading from a file:
In this test, I chose option 9 for loading data from a file.

Then, I typed the name of a file that does not exist, “somefile.txt”, and an error message was printed showing that no such file was found.

Then I typed the name of the provided file for the project, “dictinary.txt”. It was a valid one, so the function read all lines and printed an appropriate message for each one. All had valid keys and descriptions, so the “Memory allocation successful” message was printed to show that the node was created and inserted in the tree. There was only one line that had an invalid key (“Ice cream” which is not a single word as a key should), so an error message was printed for that line.

In total there were 118 lines in the file, and 117 successful + 1 error messages were printed, so the loading works well.

```
Memory allocation successful =D

Menu:
1: CREATE NODE
2: PRINT NODE
4: TREE EMPTY
6: SEARCH
7: DELETE
8: INORDER PRINT
9: LOAD TEXT FILE
10: STORE TEXT FILE
11: FREE TREE
0: EXIT

What would you like to do? Please enter your choice:
10
Tree successfully stored =D
```

...

```
       recreation.
115    Yak:A long-haired ox domesticated in the Tibetan
       plateau.
116    Zipper:A fastening device consisting of two strips
       of fabric with interlocking metal or plastic teeth.
117    Zucchini:A type of summer squash, often green in
       color.
```

Storing to a file:
Then, I chose option 10 for storing the data to a file, “storage.txt”.

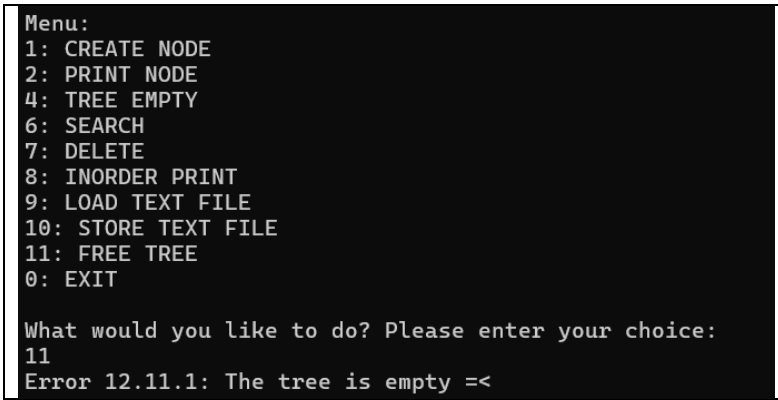
The file has 117 lines, as the one that threw an error was ignored. All stored lines have the correct format: “key:definition.”

Menu: 1: CREATE NODE 2: PRINT NODE 4: TREE EMPTY 6: SEARCH 7: DELETE 8: INORDER PRINT 9: LOAD TEXT FILE 10: STORE TEXT FILE 11: FREE TREE 0: EXIT What would you like to do? Please enter your choice: 11 Tree successfully freed =D Menu: 1: CREATE NODE 2: PRINT NODE 4: TREE EMPTY 6: SEARCH 7: DELETE 8: INORDER PRINT 9: LOAD TEXT FILE 10: STORE TEXT FILE 11: FREE TREE 0: EXIT What would you like to do? Please enter your choice: 4 The tree is empty. Menu: 1: CREATE NODE 2: PRINT NODE 4: TREE EMPTY 6: SEARCH 7: DELETE 8: INORDER PRINT 9: LOAD TEXT FILE 10: STORE TEXT FILE 11: FREE TREE 0: EXIT What would you like to do? Please enter your choice: 10 Error 12.10.1: The tree is empty =<	Trying to store when the tree is empty: Then, I freed the tree, checked that it is empty, then tried to store to the file again. Because the tree is empty, an error message was printed as there is nothing to store.

3.5 Printing the tree

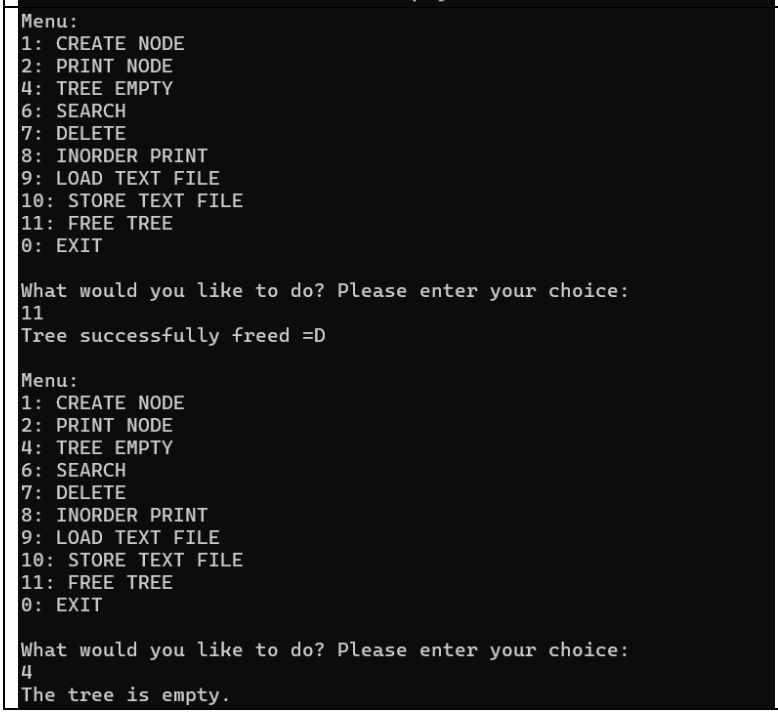
Menu: 1: CREATE NODE 2: PRINT NODE 4: TREE EMPTY 6: SEARCH 7: DELETE 8: INORDER PRINT 9: LOAD TEXT FILE 10: STORE TEXT FILE 11: FREE TREE 0: EXIT What would you like to do? Please enter your choice: 8 Error 12.8.1: The tree is empty =<	In this test, I chose option 8 for printing nodes in order. Trying to print when the tree is empty: At first, I tried doing it with the tree empty, which threw an error as there is nothing to print.
Menu: 1: CREATE NODE 2: PRINT NODE 4: TREE EMPTY 6: SEARCH 7: DELETE 8: INORDER PRINT 9: LOAD TEXT FILE 10: STORE TEXT FILE 11: FREE TREE 0: EXIT What would you like to do? Please enter your choice: 8 Please enter the maximum number of nodes from the tree to be printed inorder : 10 Airplane:A powered flying vehicle with fixed wings. Albatross:A large seabird with long wings, known for its gliding flight. Ant:A small insect that lives in colonies. Apple:A crisp, juicy fruit that comes in various colors. Astronaut:A person trained to travel in spacecraft. Banana:A long, yellow fruit that is sweet and soft inside. Bicycle:A vehicle with two wheels, powered by pedaling. Book:A set of written or printed pages bound together. Bridge:A structure spanning a physical obstacle, such as a river or road. Butterfly:An insect with colorful wings and a slender body.	Printing nodes: Then, I loaded the data from the “dictionary.txt” file into the tree to have node. I entered the number of maximum nodes to be printed (I chose 10 as an example), and the function printed the first inorder 10 nodes (proof, the first 10 nodes in alphabetical order).

3.6 Memory leak detection

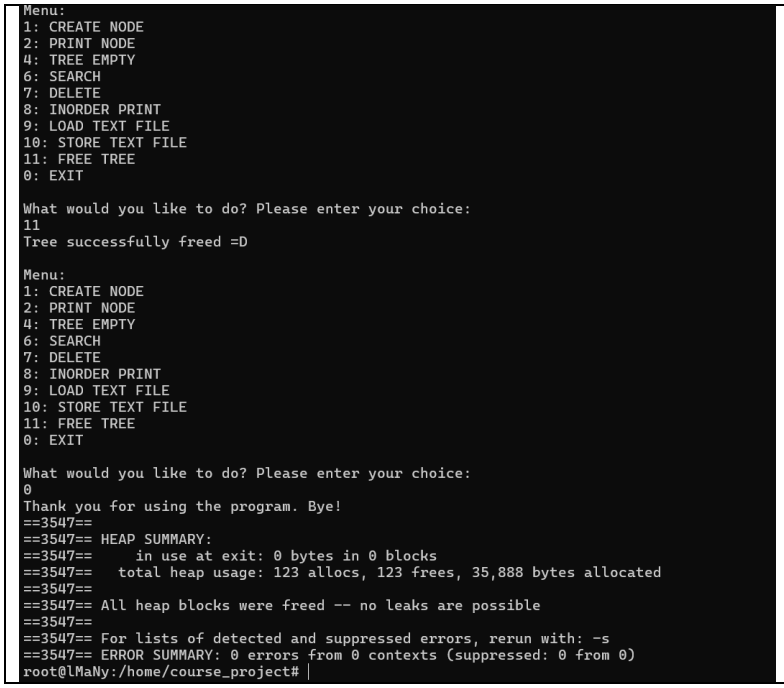


In this test, I chose option 11 for freeing the tree.

Trying to free the tree when it is empty:
At first, I tried doing it with the tree empty, which threw an error as there is nothing to free the tree from.



Freeing the tree:
Then, I loaded the data from the "dictionary.txt" file into the tree to have node.
I then freed the tree and a message saying the operation was successful was printed.
To further checks, I tried the isEmpty function, which also confirmed the tree is empty.



Then, I ran the program with valgrind, added nodes to the tree, freed the tree, then exited. In the end, all heap block were freed.

3.7 Error inventory

3.7.1 createNode

Index	Message	Description
Error 1.1	Memory allocation failed =<	Occurs: if malloc is unable to allocate memory for the new node (for example if there is not enough memory) Recovery: function returns NULL and terminates (no node is created) and user is sent back to the menu

3.7.1.1 isValidKey

Index	Message	Description
Error 1.2.1	Key has more than 29 characters =< Try again	Occurs: if the key exceeds the set limit of characters (29) Recovery: function asks the user to re-enter the key
Error 1.2.2	Key contains space(s) =< Try again	Occurs: if the key is not a single word without space Recovery: function asks the user to re-enter the key
Error 1.2.3	Key contains invalid character(s) =< Try again	Occurs: if the key contains ':' or '.' Recovery: function asks the user to re-enter the key
Error 1.2.4	Key is empty =< Try again	Occurs: if the key is empty Recovery: function asks the user to re-enter the key

3.7.1.2 isValidDescription

Index	Message	Description

Error 1.3.1	Description has more than 199 characters =< Try again	Occurs: if the description exceeds the set limit of characters (199) Recovery: function asks the user to re-enter the description
Error 1.3.2	Description contains invalid character(s) =< Try again	Occurs: if the key contains ':' or '.' Recovery: function asks the user to re-enter the key
Error 1.3.3	Description is empty =< Try again	Occurs: if the description is empty Recovery: function asks the user to re-enter the description

3.7.2 printNode

Index	Message	Description
Error 2.1	The node is NULL =<	Occurs: if the node passed as a parameter is NULL (no existing node was actually passed) Recovery: function terminates and user is sent back to the menu

3.7.3 createTree

Index	Message	Description
Error 3.1	Memory allocation failed =<	Occurs: malloc is unable to allocate memory for the new tree (for example if there is not enough memory) Recovery: function returns NULL and terminates (no tree is created) and user is sent back to the menu

3.7.4 delete

Index	Message	Description
Error 7.1	Key not found =<	Occurs: if the key passed as a parameter to be deleted does not exist in the tree Recovery: function terminates (no node is deleted) and user is sent back to the menu

3.7.5 loadTextFile

Index	Message	Description
Error 9.1	File not found =<	Occurs: if fopen fails (for example if the file with the provided name does not exist) Recovery: function terminates and user is sent back to the menu
Error 9.2	Blank line =<	Occurs: if there is a blank line in the read file (fgets catches an empty line) Recovery: function continues to read the next line
Error 9.3.1	Invalid format --> Incorrect number of `:` or `.` =<	Occurs: if the line does not have exactly one ':' and one '.' Recovery: function continues to read the next line
Error 9.3.2	Invalid format --> Missing or misplaced `:` or `.` =<	Occurs: if the line does not have a ':' and a '.', or if the '.' Is not at the end of the line Recovery: function continues to read the next line

Error 9.3.3	Invalid format --> Spaces around `:` =<	Occurs: if there is a space before, after or both before and after ':' Recovery: function continues to read the next line
Error 9.3.4	Invalid format --> Missing or invalid key or description =<	Occurs: if the line does not have a valid key (either is missing, empty, or is not a single word and contains spaces) or description (either is missing, or empty) Recovery: function continues to read the next line
Error 9.4	Memory allocation failed =<	Occurs: if creating a node using the createNode function (section 2.1) fails (section 3.1.1 -> Error 1.1) Recovery: function continues to read the next line

3.7.6 storeTextFile

Index	Message	Description
Error 10.1	File unable to open =<	Occurs: if fopen fails (for example if the file with the provided name does not exist) Recovery: function terminates and user is sent back to the menu

3.7.7 main

Index	Message	Description
Error 12.0	Invalid choice =< Please choose a valid number (0 - 11)	Occurs: when the user enters an invalid menu option (anything other than an integer from 0 to 11) Recovery: function asks the user to enter a valid option
Error 12.1.1	Node with %s key already exists =<	Occurs: when trying to create a node with a key that already exists (either a previously created and not inserted node or a node in the tree) Recovery: function terminates (no node is created), discards of the given key (potential duplicate) and user is sent back to the menu
Error 12.1.2	Memory allocation failed =<	Occurs: when realloc fails to resize the list of uninserted nodes (for example if there is not enough memory) Recovery: the program terminates as there is no space and nothing can be done
Error 12.2.1	No node with '%s' key was found =<	Occurs: when trying to print a node that does not exist (either in the list of uninserted nodes or in the tree) Recovery: function terminates and user is sent back to the menu
Error 12.6.1	The tree is empty =<	Occurs: when trying to search for a node but the tree is empty (there is nothing to search) Recovery: function terminates and user is sent back to the menu
Error 12.6.2	Node with key '%s' does not exist in the tree =<	Occurs: when trying to search for a node that does not exist in the tree (the tree is not empty) Recovery: function terminates and user is sent back to the menu
Error 12.7.1	The tree is empty =<	Occurs: when trying to delete a node but the tree is empty (there is nothing to delete)

		Recovery: function terminates and user is sent back to the menu
Error 12.7.2	Node with key '%s' does not exist in the tree =<	Occurs: when trying to delete a node that does not exist in the tree (the tree is not empty) Recovery: function terminates and user is sent back to the menu
Error 12.8.1	The tree is empty =<	Occurs: when trying to traverse the tree and print its nodes inorder but the tree is empty (there is nothing to traverse or print) Recovery: function terminates and user is sent back to the menu
Error 12.10.1	The tree is empty =<	Occurs: when trying to store the tree to a file but the tree is empty (there is nothing to store) Recovery: function terminates and user is sent back to the menu
Error 12.11.1	The tree is empty =<	Occurs: when trying to free the tree but it is already empty (there is nothing to free) Recovery: function terminates and user is sent back to the menu