**Project One**

Kyle Dale

CS 300 DSA Analysis and Design

Professor Ostrowski

February 12, 2023

**Function Signatures:**

**// Vector pseudocode**

// defines various necessary libraries
#include <?????>

// defines standard namespace;
using namespace std;

// **defines main function which will read file**
int main() {

// call ifstream to read file into vector
ifstream is ("*filenamehere.filetype*");

// defines how vector will iterate
ifstream_iterator<*VARIABLE TYPE*> *filenamehere*(start, end);

// initialize and define vector
vector<*VARIABLE TYPE*> *filenamehere*(start, end);

// output vector information to user
cout << "Read " << *filenamehere*.size() << " *VARIABLE TYPE"* <<
endl;

// print variables read from file to std::out
cout << "variables read in:\n";
copy(*filenamehere*.begin(), *filenamehere*.end(),
ostream_iterator<*VARIABLE TYPE*>*(cout, " "));*
cout << endl;

// print menu (load menu excluded it is done automatically in this example design)
cout << "1: Print Course List" << endl;
cout << "2: Print Specific Course" << endl;
cout << "9: "Exit" << endl;

cin >> choice;

switch(choice) {

case 1: printCourseList(courses); break;

case 2: printCourse(courses); break;

```
case 9: cout << “Good Bye!” << endl; break;

default: cout << “Invalid Selection, please try again.” << endl;

}

    while (choice != 9) {

    return 0;

    }

}

}
```

// **define function to print specific course**

```
// name of variables and functions subject to change in code
void printCourse(Course course) {
    // define variables for function
    string courseNumber = course.courseNumber;
    string name = course.name;

    // define search function for course
    void searchCourse(vector<Courses> courses) {
    int x = courses.size();
    int y = 0;

    // prompt user for course number
    cout << “Enter course number for desired course”;
    cin >> courseNumber;

    for(int i = 0, i < x, i++) {
        // if course is found, print it
        if (courses[i].courseNumbe r== courseNumber) {
            y = 1;
            printCourse(courses[i]);
            break;
            }
        }
    }

    // define a vector of prereq courses to output information
    vector<string> prereq = course.prerequisite;
    cout << “Prerequisites: “;
    for (int i = 0, i < prerequisites.size(), i++) {
```

```
        cout << prerequisites[i] << endl;
        }


}
```

**// function to print alphabetized list of courses**

```
void printCourseList(vector<Course> courses) {
        // use bubble sort
        int x = courses.size();
        for (int i = 0, i < x-1, i++;) {
                for (int z = 0, z < x - i - 1, j++) {
                        if (courses[j].courseNumber > courses[j+1].courseNumber) {
                        swap(courses[j+1], courses[j]);
                        cout << course.courseId << ": " << course.courseName << endl;
                        }
                }
        }
        // traverse course list to print all course now that they are ordered
        for (int i = 0, i< x, i++) {
        printCourse(courses[i]);
        }
}
```

**Runtime Analysis Vector Worst Case:**

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
|---|---|---|---|
| // call ifstream to read file into vector<br>ifstream is ("*filenamehere.filetype*"); | 1 | 1 | 1 |
| // defines how vector will iterate<br>ifstream_iterator<*VARIABLE TYPE*><br>*filenamehere*(start, end); | 1 | 1 | 1 |
| // initialize and define vector<br>vector<*VARIABLE TYPE*><br>*filenamehere*(start, end); | 1 | n | n |
| // output vector information to user<br>cout << "Read " << *filenamehere*.size()<br><< " *VARIABLE TYPE"* << endl; | 1 | n | n |

| Code | Line Cost | # Times Executes | Total Cost |
|---|---|---|---|
| **// function to print alphabetized list of courses**<br>void printCourseList(vector<Course> courses) {<br>// use bubble sort<br>int x = courses.size();<br>for (int i = 0, i < x-1, i++;) {<br>for (int z = 0, z < x - i - 1, j++) {<br>if (courses[j].courseNumber > courses[j+1].courseNumber) {<br>swap(courses[j+1], courses[j]);<br>cout << course.courseId << ": " << course.courseName << endl;<br>}<br>}<br>}<br>// traverse course list to print all course now that they are ordered<br>for (int i = 0, i< x, i++) {<br>printCourse(courses[i]);<br>}<br>} | 1 | n | n |
| | | **Total Cost** | 3n + n |
| | | **Runtime** | O(n) |

```
// Hashtable pseudocode

// defines various necessary libraries
#include <?????>

// defines standard namespace;
using namespace std;
// defines the desired default size of hashtable and unit max
const unsigned int DEFUALT_SIZE = ??
const unsigned int UNIT_MAX = ??

// defines structure to store course information
struct Course {
	string courseId;
	string courseName;
}

// defines the hash table structure
class HashTable {
// define structure to hold course
Private:
	Course course;
	unsigned int key;
	Node *next;

	// default constructor
	Node() {
		// define default key to unit max defined elsewhere
		key = UNIT_MAX
		next = nullptr;


	// initialize with course
	Node(Course acourse) : Node() {
		course = acourse;
	}

	// initialize with course and a key
	Node(Course acourse, unsigned int aKey) : Node(acourse) {
		key = aKey;
	}
	};
	// set default hashtable size
	unsigned int tableSize = DEFUALT_SIZE;
	unsigned int hash(int key);
```

```
// Define various public variables for function
public:
        ……

};
// default constructor to hold courses
HashTable::Hashtable() {
        nodes.resize(tableSize);
}

// constructor to define size of table and reduce collisions
HashTable::Hashtable(unsigned int size) {
        this->tableSize = size;
        nodes.resize(tableSize);
}

// Destructor
HashTable::~HashTable() {
        nodes.erase(nodes.begin());
}

// calculate key value
unsigned int HashTable::hash(int key) {
        return key % tableSize;
}

// function to print all courses

void HashTable::PrintAll() {
        // use bubble sort
        unsigned int x = courses.size();
        for (int i = 0, i < x-1, i++;) {
                for (int z = 0, z < x - i - 1, j++) {
                        if (course[j].courseNumber > course[j+1].courseNumber) {
                        swap(course[j+1], courses[j]);
                        cout << course.courseId << ": " << course.courseName << endl;
                        }
                }
        }
}

// function to print specific course
Course HashTable::Search(string courseId) {
        Course course;
        // create key for given bid
```

```
	unsigned key = hash(atoi(courseId.c_str()));
	// if entry is found for the key
	Course* course = &(nodes.at(key));


	// if no entry is found for the key
	if (node == nullptr || node->key = UNIT_MAX) {
		cout << "The desired course was not found";
	}
	// while node is not equal to null
	while (node != nullptr) {
		//if the current node matches, return it
		if (node->key != UNIT_MAX && node->course.courseId.compare(courseId) == 0) {
			cout  << " Course found.";
			return node->course;
		}
	}
	// define a vector of prereq courses to output information
	vector<string> prereq = course.prerequisite;
	cout << "Prerequisites: ";
	for (int i = 0, i < prerequisites.size(), i++) {
	cout << prerequisites[course] << endl;
	}
}

// method to load file (example uses .csv)
void loadCourses (string csvPath, HashTable* hashTable) {
	cout << "Loading File " << csvPath << endl;
	// initialize parser
	csv::Parser file = csv::Parser(csvPath);
	try {
		//reads rows of file
		for (unsigned int i = 0, i < file.rowCount(), i++) {
			Course course;
			course.courseId = file[i][?];
			course.courseName = file[i][?];
			hashTable->Insert(course);
		}
	}
}

int main(int argc, char* argv[]) {
	// process command line arguments
	string csvPath, bidkey;
	switch (argc) {
```

```
default:
        csvPath = "file name here";
        bidKey = "bidKey here";
}
// print menu (load menu excluded it is done automatically in this example design)
cout << "1: Print Course List" << endl;
cout << "2: Print Specific Course" << endl;
cout << "9: "Exit" << endl;

cin >> choice;

switch(choice) {

case 1: printPrintAll(courses); break;

case 2: printSearch(courses); break;

case 9: cout << "Good Bye!" << endl; break;

default: cout << "Invalid Selection, please try again." << endl;

}

        while (choice != 9) {

        return 0;

        }

}

}
```

**Runtime Analysis Hash Table Worst Case:**

| Code | Line Cost | # Times Executes | Total Cost |
|---|---|---|---|
| // defines structure to store course information<br>struct Course {<br>string courseId;<br>string courseName;<br>} | 1 | 1 | 1 |
| **// defines the hash table structure**<br>class HashTable {<br>……….<br>} | 1 | 1 | 1 |
| // method to load file (example uses .csv)<br>void loadCourses (string csvPath, HashTable* hashTable) {<br>cout << "Loading File " << csvPath << endl;<br>// initialize parser<br>csv::Parser file = csv::Parser(csvPath);<br>try {<br>//reads rows of file<br>for (unsigned int i = 0, i < file.rowCount(), i++) {<br>Course course;<br>course.courseId = file[i][?];<br>course.courseName = file[i][?];<br>hashTable->Insert(course);<br>}<br>}<br>} | | | |

<table>
<tr><th>Code</th><th>Line Cost</th><th># Times Executes</th><th>Total Cost</th></tr>
<tr><td>// function to print specific course<br>Course HashTable::Search(string courseId) {<br>Course course;<br>// create key for given bid<br>unsigned key = hash(atoi(courseId.c_str()));<br>// if entry is found for the key<br>Course* course = &(nodes.at(key));<br><br>// if no entry is found for the key<br>if (node == nullptr || node->key = UNIT_MAX) {<br>cout << “The desired course was not found”;<br>}<br>// while node is not equal to null<br>while (node != nullptr) {<br>//if the current node matches, return it<br>if (node->key != UNIT_MAX && node->course.courseId.compare(courseId) == 0) {<br>cout << “ Course found.”;<br>return node->course;<br>}<br>}<br>// define a vector of prereq courses to output information<br>vector<string> prereq = course.prerequisite;<br>cout << “Prerequisites: “;<br>for (int i = 0, i < prerequisites.size(), i++) {<br>cout << prerequisites[course] << endl;<br>}<br>}</td><td>1</td><td>n</td><td>n</td></tr>
<tr><td colspan="3" align="right">Total Cost</td><td>n + 2</td></tr>
<tr><td colspan="3" align="right">Runtime</td><td>O(n)</td></tr>
</table>

**// Tree pseudocode**

```
// defines various necessary libraries
#include <?????>

// defines standard namespace;
using namespace std;

// define structure to hold course info
struct Course {
	string courseId;
	string courseName;
}

// internal structure for tree node
struct Node {
	Course course;
	Node *left;
	Node *right;
	// default constructor
	Node() {
		left = nullptr;
		right = nullptr;
	}
	// initialize with a course
	Node(Course aCourse) :
		Node() {
			course = aCourse;
		}
}

// implements binary tree
class BinarySearchTree {
Private:
	Node* root;

	//define methods of the BST
	// ex. void addNode(Node* node, Course course);
	………

public:
	BinarySearchTree();
	………
};
```

```
//default constructor
BinarySearchTree::BinarySearchTree() {
root = nullptr;
}

// Destructor
BinarySearchTree::~BinarySearchTree() {
	if(root = nullptr) {
		delete(root->left);
		delete(root->right);
		delete root;
	}
}
```

**// search for and print specific bid - will print prereqs in function of search automatically**

```
Course BinarySearchTree::Search(string courseId) {
	// set current node equal to root
	Node* node = root;
	// loop down until bottom reached or matching course found
	while(node != nullptr) {
		if(node->course.courseID == courseID) {
			return node->course;
		}
		// if course ID is smaller traverse left
		if (courseId < node->course.courseId) {
			node = node->left
			return node->course;
		}
		// if course ID is larger traverse right
		else (courseId > node->course.courseId) {
			node = node->right
			return node->course;
		}
	Course course;
	return course;
}

// function to sort and print all nodes
void BinarySearchTree::inOrder(Node* node) {
	if(node != nullptr) {
		inOrder(node->left);
		cout << node->course.courseID << ": " << node->course.courseName << endl;
		inOrder(node->right);
		cout << node->course.courseID << ": " << node->course.courseName << endl;
	}
```

```
}

// function to load file
void loadCourses (string csvPath, HashTable* hashTable) {
        cout << “Loading File ” << csvPath << endl;
        // initialize parser
        csv::Parser file = csv::Parser(csvPath);
        try {
                //reads rows of file
                for (unsigned int i = 0, i < file.rowCount(), i++) {
                        Course course;
                        course.courseId = file[i][?];
                        course.courseName = file[i][?];
                        bst->Insert(course);
                }
        }
}


int main(int argc, char* argv[]) {
        // process command line arguments
        string csvPath, bidkey;
        switch (argc) {
        default:
                csvPath = “file name here”;
                bidKey = “bidKey here”;
        }

        // Define bst to hold all courses
        BinarySearchTree* bst;
        bst = new BinarySearchTree();
        Course course;

        // print menu (load menu excluded it is done automatically in this example design)
        cout << “1: Print Course List” << endl;
        cout << “2: Print Specific Course” << endl;
        cout << “9: “Exit” << endl;

        cin >> choice;

        switch(choice) {

        case 1: bst->InOrder(); break;

        case 2: bst->Search(); break;
```

```
case 9: cout << “Good Bye!” << endl; break;

default: cout << “Invalid Selection, please try again.” << endl;

}

        while (choice != 9) {

        return 0;

        }

}

}
```

**Runtime Analysis BST Worst Case:**

| Code | Line Cost | # Times Executes | Total Cost |
|---|---|---|---|
| // define structure to hold course info<br>struct Course {<br>    string courseId;<br>    string courseName;<br>} | 1 | 1 | 1 |
| // internal structure for tree node<br>struct Node {<br>    Course course;<br>    Node *left;<br>    Node *right;<br>    // default constructor<br>    Node() {<br>        left = nullptr;<br>        right = nullptr;<br>    }<br>    // initialize with a course<br>    Node(Course aCourse) :<br>        Node() {<br>            course = aCourse;<br>        }<br>} | 1 | 1 | 1 |
| // implements binary tree<br>class BinarySearchTree {<br>    …………<br>**}** | 1 | 1 | 1 |

<table>
<tr><th>Code</th><th>Line Cost</th><th># Times Executes</th><th>Total Cost</th></tr>
<tr><td><pre>// function to load file
void loadCourses (string csvPath, HashTable*
hashTable) {
	cout << “Loading File ” << csvPath <<
endl;
	// initialize parser
	csv::Parser file = csv::Parser(csvPath);
	try {
		//reads rows of file
		for (unsigned int i = 0, i <
file.rowCount(), i++) {
			Course course;
			course.courseId =
file[i][?];
			course.courseName =
file[i][?];
			bst->Insert(course);
		}
	}
}</pre></td><td>1</td><td>1</td><td>1</td></tr>
<tr><td><pre>// search for and print specific bid - will print
prereqs in function of search automatically
Course BinarySearchTree::Search(string courseId) {
	// set current node equal to root
	Node* node = root;
	// loop down until bottom reached or
matching course found
	while(node != nullptr) {
		if(node->course.courseID ==
courseID) {
			return node->course;
		}
		// if course ID is smaller traverse
left
		if (courseId <
node->course.courseId) {
			node = node->left
			return node->course;
		}
		// if course ID is larger traverse
right
		else (courseId >
node->course.courseId) {
			node = node->right
			return node->course;
		}
	Course course;
	return course;
}</pre></td><td>1</td><td>h</td><td>h</td></tr>
<tr><td colspan="3" align="right">Total Cost</td><td>h + 4</td></tr>
<tr><td colspan="3" align="right">Runtime</td><td>O(h)</td></tr>
</table>

**Analysis:**

When comparing the various data structures and their runtime efficiency, I would conclude that a Binary Search Tree is the best data structure for our purposes. I came to this conclusion for a variety of reasons, the first being the very nature of a BST will allow us to keep track of course prerequisites without additional code. This is because the courses with the lower "value" will be stored on the left of a "higher" value course, so printing a specific course's prerequisites requires no additional formatting, merely printing all the nodes to the left. Aside from the inherent benefit contained within a BST's structure, the worst runtime efficiency of a BST is O(h) as the most iterations will be equal to the height of the tree, not the number of courses (n). When compared to a Hash Table or Vector, the worst runtime would be O(n). Although Hash Tables can achieve an efficiency of O(log n) or O(1), when dealing with things like courses, and storing them by course ID, we could generate multiple courses ending with the same digit, and thus being hashed to the same bucket. The potential for collision, and added complexity of implementation as a result lead me to conclude a BST would be a better data structure to employ. When comparing a BST to a Vector, I believe the worst case efficiency improvements and the inherent sorting of prerequisites make for a compelling case why a BST would be more beneficial in this scenario, as long as the structure and root are implemented thoughtfully.