CS 300

Project One

Chaz Burkhart

**VECTOR**

1)

vector<Course> loadCourses (string csvPath) {

vector<Course> courses;

Parse file = csv Parser(csv Path)

try {

for each row in file

Course course

courseNumber

courseTitle

coursePrerequisite

Course pushback(course)

Catch Error

return courses

2)

Void addCourse(Vector<Course> courses, String courseNumber, String courseTitle, String coursePrerequisite)

Course course

courseNumber

courseTitle

coursePrerequisite

Course pushback(course)

3)

Void linearSearch(Vector<Course> courses, String courseNumber) {

For all courses

If the course is the same as courseNumber

Print coursePrerequisite(s)

Else

Print not found

**HASH TABLE**

1)

courseTable = new HastTable ()

HashTable::loadCourses(string csvPath, HashTable\* courseTable)

vector<Course> courses;

Parse file = csv Parser(csv Path)

try {

for each row in file

Course course

courseNumber

courseTitle

coursePrerequisite

couseTable->Insert(course)

Catch Error

2)

void HashTable::AddCourse(courseNumber)

key = hash(courseNumber)

node = node at key

if node is null

insert course

else if node point is null

insert course after existing node

else

while node pointer is not null

node = next node

insert course after existing node

3)

void HashTable::PrintCourse(courseNumber)

key = hash(courseNumber)

node = node at key

if node is null

Return

else if node is not null and course number doesn’t match and node pointer is null

return

else

while courseNumber doesn’t match and node pointer is not null

node = next node

if matching course number

print courseRequisites

**TREE**

1)

binarySearchTree = new BinarySearchTree();

void loadCourses(csvPath, binarySearchTree)

csv::Parser file = csv::Parser(csvPath);

Try{

For each row in file

Course course

courseNumber

courseTitle

coursePrerequisite

Course -> Insert(course)

Catch Error

return

2)

void BinarySearchTree::Insert(Course course)

if (root is nullpointer)

root = new Node(course)

else

this -> addNode(root, course)

void BinarySearchTree::addNode(Node node, Course course)

if (node->course.courseNumber > 0)

If (node -> left is null pointer)

node -> left = new Node(course)

else

this-> addNode(node->left, course)

else

if (node -> right is null pointer)

node -> right = new Node(course)

else

this-> addNode(node->right, course)

3)

void BinarySearchTree::inOrder(Node Course)

If (node is not equal to null pointer)

inOrder(node->left)

Print coursePrerequisites

inOrder(node->right)

**LOAD MENU**

int Choice = 0;

while (choice != 9)

print “Menu”

print “1) Load Courses”

print “2) Print Course List”

print “3) Print Course”

print “4) Exit”

input choice

switch (choice)

Case 1:

loadCourses(csvPath, courseList/courseTable/binarySearchTree)

Case 2:

printCourseList()

Case 3:

printCourse(courseNumber)

Case 4:

Return

**PRINT ALL**

void LinkedList::printCourseList()

Node\* temp = head

while (temp is not null pointer)

print Course

temp = temp -> next

return

void HashTable::printCourseList()

for (1 to number of courses)

if (nodes[i].key != UINT\_MAX)

print course

node \* current = nodes[i].next

while (current is not null pointer)

print course

current = current -> next

void BinarySearchTree::printCourseList()

if (node is not null pointer)

inOrder(node -> left)

print Course

inOrder (node -> right)

**EVALUATION**

The program initializes the CSV parser, then tries to loop the rows of the CSV file. For every row in the file the program initializes a course from the data on that row. Then the program adds it to the list, hash table, or tree. The program catches errors for each attempt.

Vector Add Course (Worst Case

| Code | Line Cost | # Times Executed | Total Cost |
| --- | --- | --- | --- |
| Course Append List | 1 | 1 | 1 |
| Total Cost | | | 1 |
| Runtime | | | O(1) |

Hash Table Add Course (Worst Case)

| Code | Line Cost | # Times Executed | Total Cost |
| --- | --- | --- | --- |
| Key = Hash(CourseNumber) | 1 | 1 | 1 |
| Node = node at key | 1 | 1 | 1 |
| else node point is not null | 1 | n | n |
| Node = next node | 1 | n | n |
| Insert course after existing node | 1 | 1 | 1 |
| Total Cost | | | 2n + 3 |
| Runtime | | | O(n) |

Binary Search Tree Add Course (Worst Case)

| Code | Line Cost | # Times Executed | Total Cost |
| --- | --- | --- | --- |
| If root is not null | 1 | 1 | 1 |
| if course number > current node | 1 | Log n | Log n |
| if node is null add pointer | 1 | 1 | 1 |
| Add node | 1 | 1 | 1 |
| Total Cost | | | Log n + 3 |
| Runtime | | | O(log n) |

Adding a course to the end of a list is simple. You simply append the course at the end of the list. Whereas for Hash Tables you could have to loop a few times to find the correct location. Binary Search Trees can cut down on time by comparing higher or lower on each node quickly finding the correct spot.

Each of these algorithms have their uses. Binary Search Trees are typically good for large data sets, because of their logarithmic properties. The Hash Table can be good in a similar situation especially if chaining in the buckets is used and the hash key distributes the data fairly evenly. Each course could have its own bucket which would speed up time significantly rather than looping around until it finds an empty bucket because of multiple collisions. Finally lists are good if the number of data points is small. Searching through 20 items is very quick, whereas searching through 200,000 items is very slow. Adding to the end of the line is also very quick..

For ABCU, I would recommend a hash table. The number of classes a single computer science department has is rather small, but the benefits of being able to scale efficiently while still having good speeds are smaller data sizes allows ABCU to still have a fast program.