6-2 Project One

Jonathan C. Sanchez

August 25, 2024

|  |  |  |  |
| --- | --- | --- | --- |
| **Assignment** | **Date** | **Name** | **Comments** |
| 3-3 Project One Milestone One: Vector Data Structure | July 28, 2024 | Jonathan C. Sanchez | Vector - Milestone 1 |
| 4-3 Project One Milestone Two: Hash Table Data Structure | August 11, 2024 | Jonathan C. Sanchez | Hash Table - Milestone 2 |
| 5-3 Project One Milestone Three: Binary Search Tree Data Structure | August 23, 2024 | Jonathan C. Sanchez | Binary Search Tree – Milestone 3 |
| 6-2 Project One | August 25, 2024 | Jonathan C. Sanchez | Menu & Sorted List – Project One |

**Purpose:**

The purpose of Project One is to create Pseudocode that effectively illustrates the implementation of a Vector Data Structure, Hash Table Data Structure, Binary Search Tree Data Structure, and a Menu & Sorted List used for managing the computer science courses at ABCU. The pseudocode demonstrates how each data structure reads a file into the structure, stores course information, searches for course info, and returns related course information. Runtime analysis is also performed for each data structure in addition to the pseudocode. The pseudocode and analysis will help us determine the most efficient and suitable data structure to implement.

**Recommendation:**

Choosing the right data structure for any use case is a hard feat. The developer and customer must consider the needed functionality and efficiency of each data structure. After analyzing the three data structures, the hash table or HashMap stands out as the top contender for ABCU’s use case. One of the main benefits to a hash table is that it offers an average time complexity of O(1) for specific data lookups. This allows the user to access course details quickly. A second benefit is that hash tables automatically resize to accommodate the size of the data. If more data is added the table will dynamically resize to store the additional data. If less data is available, the opposite will occur, and the table will shrink. The ability to handle varying data sizes ensures that a system is efficient regardless of the volume of data. Considering ABCU’s large dataset of courses and the need for fast lookup of courses, a hash table is the better choice of the three data structures.

**Hash Tables:**

* Advantages
  + Fast Lookup/Access: Hash tables can provide constant time complexity O(1) for the lookup of elements due to the key-to-index mapping.
  + Dynamic Resizing: Hash tables dynamically resize to maintain effectiveness and load factor.
  + Versatile: Hash tables can be used in many scenarios due to its handling of key-value pairs.
* Disadvantages
  + Handling Collisions: Hash tables can potentially store different keys same index, causing a collision.
  + Memory: Hash tables use additional memory to buckets and hash functions.

**Binary Search Tree:**

* Advantages
  + Sorted Order: Elements are sorted natively.
  + Balancing: Trees are balanced to maintain logarithmic height to provide greater efficiency.
  + Recursive: Depending on the use case, this could be a disadvantage. However, it allows the user to loop effectively through the tree with the same logic.
* Disadvantages
  + Indirect Indexing: Requires traversal of each subtree.
  + Time Complexity: O(log n)

**Vector:**

* Advantages
  + Random Access: Elements can be accessed using an index.
  + Dynamic: Vectors can resize to accommodate data volume.
  + Cache: Elements are stored in contiguous memory locations. Improving the performance of the cache.
* Disadvantages
  + Costly: Inserts and deletions are O(n) and require shifting of elements.
  + Initial Capacity: Vectors have a fixed initial capacity. Causing the need for reallocation.

**Vector – Milestone 1**

**Read Input File:**

Ifstream input(“xyz.txt”);

If the file is not found{

print “File not Found”

}

While (!input.eof){

Read a line of the file

Parse the line into (courseNumber, courseTitle, prerequisiteNumbers)

}

**Return Courses:**

void fileManager(vector<Courses> courses, String courseNumber){

for (const Course course : courses){

if user inputs the course.number and it equals courseNumber{

return the Course Number and the Course Title

}

}

If user inputs the course.title and it equals courseTitle{

Return the Course Number and the Course Title}

If the course has prerequisites && each prerequisiteNumber has a corresponding courseNumber{

Print each perquisite’s course information

}

Else {

Print “Could Not Get Prerequisites”

}

Return;

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Vector** | Line Cost | #Times Executed | Total Cost |
| Read each string of file | 1 | N | N |
| For const course : courses | 1 | N | N |
| If course.number = courseNumber | 1 | 1 | 1 |
| Print course information | 1 | 1 | 1 |
| If course.title = courseTitle | 1 | 1 | 1 |
| Print course information | 1 | 1 | 1 |
| Check Prerequisites | 1 | 1 (best case) | 1 |
| Check Prerequisites | 1 | N | N |
| Print Each prerequisite | 1 | N | N |
| **Total Cost** | | | 6n + 3 |
| **Runtime** | | | O(n) |

**Hash Table – Milestone 2**

Create an empty Hash Table

//Define Course Structure

Struct Course {

courseNumber

courseTitle

prerequisiteNumbers

};

//Initialize hash table

Hashtable(){

For (int i = 0; i < tableSize; ++i)

Table[i] = nullptr;

}

//Load file

loadFile(filename){

ifstream file(filename);

if file is not found, return “file error”

}

While (!input.eof){

Read a line of the file

Parse the line into (courseNumber, courseTitle, prerequisiteNumbers)

}

Create new course = course;

Insert course into the hash table;

//Insert course method

Insert course{

Create hash key for courseNumber

Node \*newNode = course

If table[key] = nullptr{

Table[key] = newNode;

} else {

find the last node at table[key] & append newNode to the end;

}

}

//Return course by title

If course.title in courses{

Return the Course Number and the Course Title}

If the course has prerequisites && each prerequisiteNumber has a corresponding courseNumber{

Print each perquisite’s course information

}

Else {

Print “Could Not Get Prerequisites”

}

printAll(){

for (int i = 0; i < tableSize; i++) {

Node \*current = table[i];

While (current != nullptr){

Print

}

}

|  |  |  |  |
| --- | --- | --- | --- |
| **Hash Table** | Line Cost | #Times Executed | Total Cost |
| Create Hash Table | 1 | 1 | 1 |
| Course Structure | 1 | 1 | 1 |
| Initialize hash table | 1 | 1 | 1 |
| Load file | 1 | 1 | 1 |
| Check for file | 1 | 1 | 1 |
| End of file | 1 | N+1 | N+1 |
| Read file | 1 | N | N |
| Parse file | 1 | N | N |
| Create new course object | 1 | N | N |
| Insert course into hash table | 1 | N | N |
| Create Hash Key for course | 1 | N | N |
| Table key = newNode | 1 | N | N |
| If course.title in courses | 1 | M (max) | M (max) |
| Return course | 1 | N | N |
| Return Prerequisites | 1 | N | N |
| Print All | 1 | Size | Size |
| **Total Cost** | | | 6n + 3 |
| **Runtime** | | | O(n) |

**Binary Search Tree – Milestone 3**

Struct Course {

courseNumber

courseTitle

prerequisiteNumbers

};

//Create the methods to load and read the file

Function loadData(filename){

ifstream file(filename);

if file is not found, return “file error”

}

//Create Binary Search Tree

courseTree = Create BTS

While (!input.eof){

Read a line of the file

Parse the line into (courseNumber, courseTitle, prerequisiteNumbers)

}

If line is invalid{

Print “error: invalid format”

}

Course = new course

Insert course into BTS

Number = components

//Validate course data

Function validateCourseData(courseList){

Courses = Create empty set

ForEach course in courselist{

courseNumber = course.courseNumber

Courses.Add(coursenumber)

}

ForEach course in courseList{

forEach prereq in course.prerequisites{

if prerequisites Not in courseSet{

return “Error: prerequisite DNE”

}

}

}

Return “valid”

END

InsertCourseIntoTree(tree, course){

INSERT course INTO tree based on courseNumber

}

//Print Courses

printCourseInfo(tree, courseNumber){

course = (course, courseNumber)

If course is found{

print << “Course Number: “ + course.courseNumber , “|”, “Title: ” + course.courseTitle

If course.prerequisites not null{

Print << “Prerequisites: “

For each prerequisite in course course.prerequisites{

prerequisiteCourses = search tree for prerequisites

If prerequisiteCourse is found{

Print << prerequisiteCourse.courseNumber + “: “ + prerequisiteCourse.title

}

Else {

Print << prerequisite + “ (Prerequisite course not found)”

}

Else{

Print << “No Prerequisites”

}

Else{

Print << “Courses not valid”

}

}

END

|  |  |  |  |
| --- | --- | --- | --- |
| **Binary Search Tree** | Line Cost | #Times Executed | Total Cost |
| Open File | 1 | 1 | 1 |
| Read File | 1 | N | N |
| Parse File | 1 | N | N |
| Validate data | 1 | N | N |
| Create course object | 1 | N | N |
| Insert course into BST | Log(n) | N | N\*log(n) |
| Create Set of Course Numbers | 1 | 1 | 1 |
| Check Prequisites | Log(n) | N | N\*log(n) |
| Search for a course | Log(n) | 1 | Log(n) |
| Print course deatails | 1 | 1 | 1 |
| Search and print course prerequisites | Log(n) | N | N\*log(n) |
| **Total Cost** | | | 6n\*log(n)+5 |
| **Runtime** | | | O(n\*log(n)) |

**Menu & Sorted List – Project One:**

//Implement Main Menu loop

mainMenu() {

int option;

bool dataLoaded = false;

displayMenu() {

print("1. Load File Data")

print("2. Print All Courses")

print("3. Print Course Details")

print("9. Exit")

}

While(true){

DisplayMenu

option = getInput();

//Create cases for options

Switch (option){

case 1: Load File Data

print("Enter filename: ")

getinput;

loadFileData(courses, filename)

break;

case 2: Print All Courses

printAllCourses(courses)

break;

case 3: Print Course Details print(

"Enter course number: ")

Getinput;

courseNumber

printCourseInfo (course, courseNumber)

break;

case 9: Exit Program print("Exiting...");

return;

default: print("Invalid option. Please try again.")}

//Load Data

Void loadData(filename) {

ifstream file(filename);

if file is not found, return “file error”

}

//Print sortedList

Void printSortedList(){

<Course> courseList = getCourses();

//Sort Alphnumerically (CourseNumber)

Sort(courseList.begin(), courseList.end(),

[](const Course &a, const Course &b) { return a.courseNumber < b.courseNumber;

}

forEach(const Course & course in courseLIst){

print << “Course Number: “ + course.courseNumber << endl;

print << “Title: “ + course.title << endl;

print << “Prerequisites: “ + course.prerequisites << endl;

Void printCourseInfo(string courseNumer){

course = (course, courseNumber)

If course is found{

print << “Course Number: “ + course.courseNumber , “|”, “Title: ” + course.courseTitle

If course.prerequisites not null{

Print << “Prerequisites: “

For each prerequisite in course course.prerequisites{

prerequisiteCourses = search tree for prerequisites

If prerequisiteCourse is found{

Print << prerequisiteCourse.courseNumber + “: “ + prerequisiteCourse.title

}

Else {

Print << prerequisite + “ (Prerequisite course not found)”

}

Else{

Print << “No Prerequisites”

}

Else{

Print << “Courses not valid”

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Menu & Sorted List** | Line Cost | #Times Executed | | Total Cost |
| Display Menu | 1 | 1 | | 1 |
| Get Input (main menu loop) | 1 | 1 | | 1 |
| Load File Data (option 1) | 1 | 1 | | 1 |
| Get Input (filename) | 1 | 1 | | 1 |
| Load File Data (loadData) | 1 | 1 | | 1 |
| Read File (inside loadData) | 1 | N | | N |
| Parse File (inside loadData) | 1 | N | | N |
| Create Course Object | 1 | N | | N |
| Insert Course into Data Str. | 1 | N | | N |
| Print All Courses (option 2) | 1 | 1 | | 1 |
| Print Sorted List | 1 | 1 | | 1 |
| Sort Courses | N | 1 | | N |
| Print Course Details | 1 | 1 | | 1 |
| Get Input (course number) | 1 | 1 | | 1 |
| Print Course Info | 1 | 1 | | 1 |
| Search for Course | 1 | 1 | | 1 |
| Print Prerequisites | 1 | N | | N |
| **Total Cost** | | | 6n+12 | |
| **Runtime** | | | O(n) | |