# 6.2 Project One

1. Resubmitted all three files and data & data structures.

**Data for the project:**

The course data is as follows:

CSCI100,Introduction to Computer Science

CSCI101,Introduction to Programming in C++,CSCI100

CSCI200,Data Structures,CSCI101

MATH201,Discrete Mathematics

CSCI300,Introduction to Algorithms,CSCI200,MATH201

CSCI301,Advanced Programming in C++,CSCI101

CSCI350,Operating Systems,CSCI300

CSCI400,Large Software Development,CSCI301,CSCI350

Data structures:

struct Course {

string courseId

string courseName

vector prerequisites

}

class HashTable {

private:

// Define structures to hold courses

struct Node {

Course course;

string key;

Node \*next;

// default constructor

Node() {

key = “”;

next = nullptr;

}

// initialize with a course

Node(Course aCourse) : Node() {

course = aCourse;

}

// initialize with a course and a key

Node(Course aCourse, string aKey) : Node(aCourse) {

key = aKey;

}

};

1. Pseudocode for Main Menu

int main(){

if input file is not specified as the argument

prompt the user to “Please provide the input file”.

Check the if file exists and readable, if not print error message and exit the program.

Read the input file and store it in “inputFile” variable.

HashTable\* courseList = new HashTable()

// display the Main Menu

While (choice is not 9) {

Print “1. Load Courses”

Print “2. Display Courses”

Print “3. Find Course”

Print “9. Exit”

Read the input to “choice”

Switch(choice) {

Case 1:

// read the file and store in course list

readInputFile(inputFile, courseList);

// validate course and remove invalid courses

validateCourse(courseList);

Case 2:

// Sort and print all the entries

printAll(courseList);

Case 3:

//search the courseId and print.

printCourse(courseId);

}

}

Print “Good bye.”

}

void readInputFile(String inputFile, HashTable \*courseList) {

#File exists

Open the file and read line by line till end of file.

Split the line by comma separator.

If we have less than 2 tokens then print error and

Skip the line. (we need to have at least two tokens).

If the line is valid store in the HashTable data structure

//First Create the Course object and assign the tokens

Course.courseId = first token course number.

Course.courseName = second token course name.

Course.prerequisites =

Split the third token with “,” separator and read all the token and store it in the vector assign.

Call courseList->Insert(Course)

Close the file and return

}

Void HashTable::Insert(Course){

Check the if Course.courseId exists

If exits

replace the course information

Else

Create a new node and add to the end.

}

unsigned int HashTable::hash(string key) {

// The course id “MATH201, CSCI201 or ITSI201”

// to avoid collision, get first 4 ASCII values

Loop through each char and add the ASCII values

int asciiSum = static\_cast<int>(chars 0 through 3)

// The course is “MATH201” split the string and read number

int value = atoi(key.substr(4,3).c\_str())

return (value+asciiSum) % tableSize;

}

Void HashTable::ValidateCourse() {

Loop through with first node in HashTable till node != nullptr

if the third value prerequisites vector size is

not empty

Loop the prerequisites vector and call HashTable::search(courseID) for the each entry in the vector.

if the prerequisites id not found

then delete node.

assign the node = node->next

}

Node HashTable::search(courseId) {

int key = HashTable::hash(courseId)

Node\* node = &(node.at(key));

If node matches the courseId

return node;

If node is nullptr or empty

return emptynode; // not found

Loop through the linked list and ≈if courseId matches

return node

Else

return emptynode // not found

}

Void printCourse(courseId) {

Call HashTable::search(courseId)

If node != nullptr

if prerequiriste vector is not empty

Loop through and create “prequiriste\_string”.

Print “CourseId: “ << node->course.courseId << “ Course Name: “ << node->course.courseName << “ Pre-requisites: “ << prequiriste\_string << endl;

Else // not found

Print “Course Id: “ << courseId << “ Not found !!!”

<< endl;

}

1. Pseudocode to print the list of courses in Computer Science program.

void printAll() {

// sort the course list

Create an empty vector<Course>.

Loop through the courseList and add the entry to vector<Course>

Sort the vector<Course>

Loop through the vector and print all the entries

if prerequiriste vector is not empty

Loop through and create “prequiriste\_string”.

Print “CourseId: “ << node->course.courseId << “ Course Name: “ << node->course.courseName << “ Pre-requisites: “ << prequiriste\_string << endl;

}

1. Runtime and memory of data structures:

Vector:

Search: O(n) – Linear search, need to iterate through each element to find a course.

Sort: O (n log n) – it needs to be sorted before printing all course list.

Insert: O(1) – appending at the end of the list is simple operation.

Memory usage: it uses contiguous memory. So, resizing is expensive.

Hash Table:

Search: O(1) – most of the time, O(n) if we have collisions.

Sort: It is not sorted, we need to store it in a vector data structure, sort and then print the course list.

Insert: O(1) – most of the time and O(n) with collisions.

Memory usage: it uses extra storage for Hash Key and needs to handle collision. So, it uses more memory.

Binary Search Tree:

Search: O(log n) is for well balanced tree

Sort: O (n) – well balanced tree is sort, we need to do in-order traversal.

Insert: O(log n) for well balanced tree and O(n) for unbalanced tree.

Memory usage: it uses more memory due to node structure. But this is structure is useful when we have frequent insertion and deletion.

1. Advantages and disadvantages of each structure:

Data Structure: Vector

Advantage: Simple and straightforward data structure to implement and iteration over elements is fast.

Disadvantage: Searching is slow O(n), it needs to be sorted before printing. Inserting or deleting elements in the middle of vector is expensive.

Data Structure: Hash Table

Advantage: Searching is faster and inserting and deleting is also faster O(1), large amout of data can be stored in small amount of memory.

Disadvantage: Data is not stored is not sorted. Collisions handling is needed which can impact performance.

Data Structure: Binary Search Tree

Advantage: The data is stored in ordered and has good balance. It is easier to search data O(log n). Inserting and deleting data is efficient.

Disadvantage: Implementing is complex task compared to Vector or Hash Table. If it is unbalanced it can degrade performance.

1. Recommendations:

For this project, I would like to recommend HashTable data structure. As we don’t have much insertion and deletion. This data structure has speed in searching and inserting course information. When printing all course list, we can temporarily, sort the data and print all courses. It is less complex to implement compared to Binary search tree.

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **for all courses** | 1 | n | n |
| **if the course is the same as courseNumber** | 1 | n | n |
| **for each prerequisite of the course** | 1 | 1 | 1 |
| **for each prerequisite of the course** | 1 | n | n |
| **print the prerequisite course information** | 1 | n | n |
| **Total Cost** | | | 4n + 1 |
| **Runtime** | | | O(n) |