Project one run-time analysis

| **Code loadCourses for vectors** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Open filesName** | **1** | **1** | **1** |
| **If checkFile(filename != 1)** | **1** | **1** | **1** |
| **Close filename** | **1** | **1** | **1** |
| **Exit program** | **1** | **1** | **1** |
| **Else** | **1** | **1** | **1** |
| **Close filename** | **1** | **1** | **1** |
| **Open filename** | **1** | **1** | **1** |
| **While not at end of file** | **1** | **N** | **N** |
| **Course tempCourse** | **1** | **N** | **N** |
| **Parse string up to ‘,’** | 1 | n | n |
| **Set tempCourse courseNum to string** | 1 | n | n |
| **Parse string up to next ‘,’** | 1 | N | N |
| **Set tempcourse courseName** | 1 | n | n |
| **Parse string to next ‘,’ or newline** | 1 | N | N |
| **While string != newline** | 1 | N | N |
| pushback string to prereq vector of tempCourse | 1 | N | N |
| **Parse string to next ‘,’ or newline** | 1 | N | N |
| Pushback tempCourse to courses | 1 | n | n |
| Close fileName | 1 | 1 | 1 |
| **Total Cost** | | | 11n + 8 |
| **Runtime** | | | O(n) |

| **Code loadCourses for hashTable**  **Same as above plus insert function** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| **Int key = hash(courseNum)** | **1** | **N** | **N** |
| If(nodes.at(key) is empty){ | **1** | **N** | **N** |
| Set nodes.at(key) course to equal course | **1** | **n** | **n** |
| Set nodes.at(key) key to equal key | **1** | **n** | **n** |
| Set nodes.at(key) next to null | **1** | **N** | **N** |
| **Else** | **1** | **N** | **N** |
| Set tempPtr to nodes.at(key) | **1** | **N** | **N** |
| While(tempPtr next != null) | **N** | **N** | **N^2** |
| Set tempPtr to tempPtr next | **N** | **N** | **N^2** |
| Set tempPtr next to point to new node | **1** | **N** | **N** |
| Set new node course to course | **1** | **N** | **N** |
| **Set new node key to key** | **1** | **N** | **N** |
| **Set new node next to null** | **1** | **N** | **N** |
|  |  | **Total Cost** | **11N = 2N^2** |
|  |  | **Runtime** | **O(N^2)** |

| **Code loadCourses for hashTable**  **Same as above plus insert function** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If BST root is empty | **1** | **n** | **N** |
| Point root to a new Node | **1** | **N** | **N** |
| Add course data to node | **1** | **N** | **N** |
| **Else** | **1** | **N** | **N** |
| **Call addNode(rootNode, course)** | **N^N** | **N** | **N(N^N)** |
|  |  | **Total Cost** | **4N + N(N^N)** |
|  |  | **Runtime** | **O(N^N)** |

**Evaluations**

**Vectors**

Advantages:

* Faster course loading
* Easier to code for
* Least storagespace

Disadvantages:

* Slowest search, sort, and removal algorithms

**Hash Tables**

Advantages:

* Faster course loading than BST
* Can have faster searches than BST dependent on hash algorithm and how collisions are handled
* Faster searches, sorting, and removal as compared to vectors

Disadvantages:

* Slower course loading than vectors
* Possibly hardest to code well
* Posibbly max storagespace requirements

**BST**

Advantages:

* Faster removal than Hash Table
* Easier coding than Hash Table
* Can have faster searching than Hash Table
* Automatically sorts when created
* Faster searches, sorting, and removal than vectors

Disadvantages:

* Slowest course loading
* Larger storage space requirements than vectors
* Removals can become complicated if several children exist for node to be removed

Project 2 sort and print course function

//Partition function used by the quickSort

int partition(vector<Course>& courses, int left, int right) {

//Set pivot index to the midpoint between left and right indexes

int pivotIndex = left + (right - left) / 2;

//Set pivot value to the courseNum at the pivotIndex

string pivotValue = courses.at(pivotIndex).getCourseNum();

//Initialize tempValue to later hold course to be moved

Course tempValue;

//While loop to go until left index equals or surpases right index

while (left <= right) {

//increase left index until the courseNum at that index is larger than pivotValue

while (courses.at(left).getCourseNum().compare(pivotValue) < 0) {

++left;

}

//Decrease right index until the courseNum at that index is less than pivotValue

while (courses.at(right).getCourseNum().compare(pivotValue) > 0) {

--right;

}

//check to make sure that left index is still less than right

if (left <= right) {

//set tempValue to the course at the left index

tempValue = courses.at(left);

//set the left index to hold the course currently at the right index

courses.at(left) = courses.at(right);

//set right index to hold course in tempValue

courses.at(right) = tempValue;

//Increase left and decrease right index to keep process moving

left++;

right--;

}

}

//return left index as pivot index for quickSort

return left;

}

//quicksort function for vector of courses

void quickSort(vector<Course>& courses, int left, int right) {

//Check if left index is smaller than right. if it is not then vector section is considered sorted

//and exit back to calling function

if (left < right) {

//call partition to sort vector section and use the return as the new pivotIndex

int pivotIndex = partition(courses, left, right);

//call recursive quicksort for left and right half of the current vector section

quickSort(courses, left, pivotIndex - 1);

quickSort(courses, pivotIndex, right);

}

}

//function to print all courses held in the vector courses

void printCourseList(vector<Course> courses) {

//First call quicksort to put courses in alphanumeric order least to greatest

quickSort(courses, 0, courses.size() - 1);

//for loop to go through every course and call the basic print function

for (int i = 0; i < courses.size(); i++) {

courses.at(i).printCourseBasic();

}

}