Donovan Brown
10/9/22
CS 300

Project One: Data Structure Pseudocode and Evaluation

Pseudocode

// Define Course structure used by all data structures

```
class Course {
    String courseNumber;
    String courseName;
    Vector<string> preReqs;
    Void Print() {
        Output this courseNumber and courseName
        For each preReq in preReqs output preReq
    }
}
```

Vector

// Search schedule for a given course using courseNumber

```
courseSearch(vector<course> courses, string courseNum){
    Create empty course
    For each course in courses
        If current course courseNumber matches courseNum
            return course
    Return empty
}
```

// Prints whole schedule

```
Void Print(vector<course> courses, string courseNum){
    For each course in courses
        Invoke course print()
}

Void selectionSort(vector<node> & courses){
    Initialize int min
    For loop from int i = 0 to courses - 1 {
        If courseNumber at j is < courseNumber at min
```

```
                Set min to j
            }
            Swap node at i with min
        }
    }

    Void loadCourses(string fileName, vector<course> & courses){
        Initialize fstream fileStream to get contents of file
        Initialize string line to hold a single line in file
        Initialize stringstream lineStream to get contents of each line
        Initialize string token to hold a single word in line

        Open fileName with file Stream
        Initialize int count to hold the token count per line in file
        Get line from fileStream until none left
            Fill lineStream with current line
            Set count to 1
            Create course aCourse for each line in file
            Get token from lineStream up to ‘,’ until none left
                If (count == 1){
                    Set aCourse courseNumber to token
                    Increment count
                }
                Else if (count == 2){
                    Set aCourse courseName to token
                    Increment count
                }
                Else{
                    If (token exists in courses as a course) add token to aCourse’s preReqs
                    Else output file format error
                    Increment count
                }
            If (count < 2){
                Output “Error in file format, each course must have course # and name.”
            }
        Push aCourse to back of courses
        Clear lineStream for next time
    }
```

Hashtable

```
const unsigned int DEFAULT_SIZE = 8;
```

```
class Hashtable {
    struct Node {
        Course course;
        Node* next;
        unsigned int key;

        Node() {
            key = UINT_MAX;
            next = nullptr;
        }
        Node(course aCourse): Node() {
            course = aCourse;
        }
        Node(course aCourse, unsigned int akey) : Node(acourse) {
            key = akey;
        }
    };

    unsigned int size = DEFAULT_SIZE;
    vector<Node> table;

};

// Hashes digits in course number string, ex. CSCI100 - return 100 % size
unsigned int Hashtable::Hash(string courseNum){
    return atoi(courseNum.substr(4).c_str()) % size;
}

void Hashtable::add(course aCourse){
    Create key for aCourse by hashing acourse's courseNumber
    Create Node* node to retrieve node using key
    If (node == nullptr){
        Create new node newCourse with aCourse and key
        Insert contents of newCourse into table at position[key]
    }
    else if (node's key == UINT_MAX){
        Update node's key to key
        Update node's course to aCourse
        Update node's next to nullptr
    }
    Else{
        While (node's next != nullptr) set node to node's next
            Create new node newCourse with acourse and key
            Set node's next to newCourse
    }
}

void Hashtable::Print(){
    Initialize vector of Nodes sortedTable
    For each node in table
        If (node's key != UINT_MAX) {
            push node to back of sortedTable
        }
    Create Node* listNode and set to node's next
    While (listNode != nullptr) {
        push listNode to back of sortedTable
        set listNode to listNode's next
    }
    SelectionSort(sortedTable)
    For each node in sortedTable
        Invoke node's course print()
}
void Hashtable::SelectionSort(vector<Node> &sortedTable) {
    initialize int min
    For loop from int i = 0 to sortedTable - 1{
        set min to i
        For loop from int j = i + 1 to end of sortedTable {
            If courseNumber at j is < courseNumber at min
                set min to j
            }
            Swap node at i with min
        }
    }
Course Hashtable::Search(string courseNum){
    Create empty course obj
    For each node in this table
        If (node's course's courseNumber == courseNum)
            return node's course
        Create Node* listNode and set to node's next
        While (listNode != nullptr) {
            If (listNode's course's courseNumber == courseNum) return listNode's
Course
            listNode = listNode's next
        }
    return empty obj
}

void LoadCourses(string fileName, Hashtable &Htable){
    Initialize fstream fileStream to get contents of file
    Initialize string line to hold a single line in file
    Initialize stringstream lineStream to get contents of each line
    Initialize string token to hold a single word in line

    Open fileName with fileStream
    Initialize int count to hold the token count per line in file
    Get line from fileStream until none left
        Fill lineStream with current line
        Set count to 1
        Create Course aCourse for each line in file
        Get token from lineStream up to ',' until none left
            If (count == 1) {
                set acourse's courseNumber to token
                increment count
            }
            else if (count == 2) {
                set acourse's courseName to token
                increment count
            }
            Else {
                If (token exists in Hashtable as a course) add token to aCourse's PreReqs
                Else output file format error
                increment count
            }
        If (count < 2) {
            output "Error in file format, each course must have course # and name."
        }
    Add aCourse to Htable
    Clear lineStream for next line
}
```

Tree

```
class BST {
    struct Node {
        Course course;
        Node* left;
        Node* right;
        Node() {
            left = nullptr;
            right = nullptr;
        }
        Node(Course aCourse) : Node() {
            course = aCourse;
        }
        ~Node() {
            delete left;
            delete right;
        }
    };
    Node* root;
};

void BST::InOrder(){
    inOrder(root)
}

void BST::inOrder(Node* node){
    If (node is not empty) {
        recursively traverse node's left sub-tree
        invoke node's course print()
        recursively traverse node's right sub-tree
    }
}

void BST::Insert(Course aCourse){
    If (root is empty) set root to new node with aCourse
    Else addNode(root, acourse)
}

void BST::addNode(Node* node, Course aCourse){
    If (acourse's courseNumber < current node's courseNumber) {
        If (node's left child is empty) add new Node with course at node's left child
        Else recursively traverse node's left sub-tree
    }
    Else{
        If (node's right child is empty) add new Node with course at node's right child
        Else recursively traverse node's right sub-tree
    }
}

void BST::Remove(string courseNum){
    removeNode(root, nullptr, courseNum)
}
```

```
}

void BST::removeNode(Node* node, Node* par, string courseNum){
    If (node's course courseNumber matches courseNum ) {
        // remove leaf
        If (node's left is nullptr AND node's right is nullptr) set node to nullptr
        // remove node with left child
        Else if (node's left is not nullptr) {
            If (par is nullptr) set root to root's left
            Else if (par's left is node) set par's left to node's left
            Else set par's right to node's left
        }
        // remove node with right child
        Else if (node's right is not nullptr) {
            If (par is nullptr) set root to root's right
            Else if (par's left is node) set par's left to node's right
            Else set par's right to node's right
        }
        // remove node with two children
        Else {
            set Node pointer suc to node's right
            While (suc's left is not nullptr) {
                set par to suc
                set suc to suc's left
            }
            Set Node pointer temp to suc
            removeNode(suc, par, courseNum)
            set node to temp
        }
    }
    else if (node's course courseNumber > courseNum) removeNode(node's left, node, courseNum)
    Else removeNode(node's right, node, courseNum)
}

Course BST::Search(string courseNum){
    set Node pointer current to root
    while(current is not nullptr) {
        If (current's course courseNumber matches courseNum) return current's course
        If (current's course courseNumber > courseNum) set current to current's left
        Else set current to current's right
    }
    create empty course
    return empty course
}
```

```
void LoadCourses(string fileName, BST &bst) {
    Initialize fstream fileStream to get contents of file
    Initialize string line to hold a single line in file
    Initialize stringstream lineStream to get contents of each line
    Initialize string token to hold a single word in line

    Open fileName with fileStream
    Initialize int count to hold the token count per line in file
    Get line from fileStream until none left
        Fill lineStream with current line
        Set count to 1
        Create course aCourse for each line in file
        Get token from lineStream up to ',' until none left
            If (count == 1) {
                set aCourse's courseNumber to token
                increment count
            }
            else if (count == 2) {
                set aCourse's courseName to token
                increment count
            }
            Else {
                if (token exists in bst as a course) add token to acourse's PreReqs
                else output file format error
                increment count
            }
        If (count < 2) {
            output "Error in file format, each course must have course # and name."
        }
    Insert aCourse into bst
    clear lineStream for next line
}
```

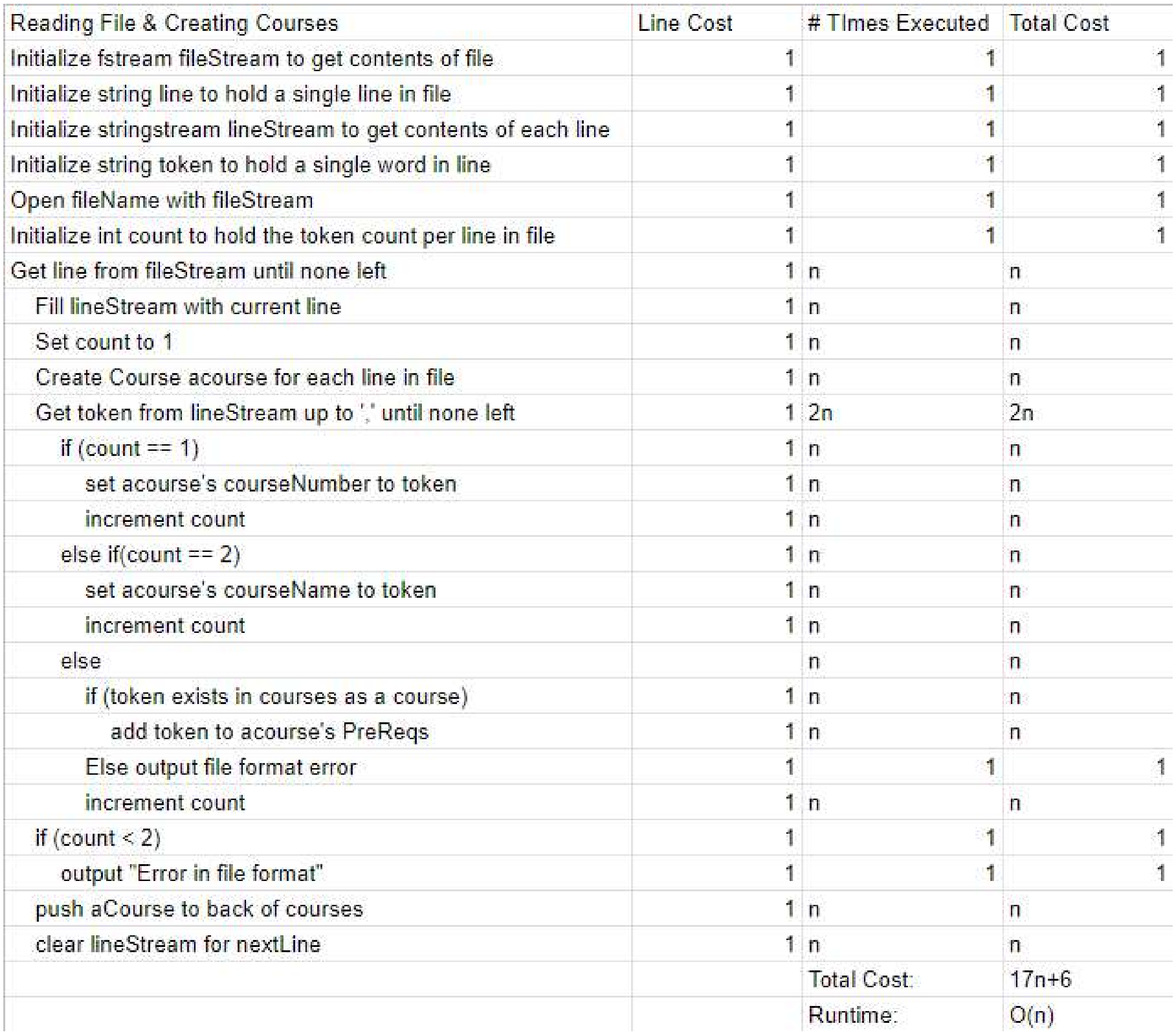
Menu

```
Create schedule object to hold courses
Initialize string coursekey
Initialize Course aCourse
Initialize int choice to 0
Initialize int choice2 to 0
While (choice != 9) {
    Output "Menu:"
    Output "1. Load Schedule\n"
    Output "2. Display\n"
    Output "3. Remove Course\n"
    Output "9. Exit\n"
    Output "Enter choice: "
    wait for input and store in choice

    Switch (choice) {
    Case 1:
        LoadCourses(fileName, schedule)
        Break
    Case 2:
        While (choice2 == 0) {
            Output "1 ). Display Schedule\n"
            Output "2 ). Display Course\n"
            Output "Enter choice: "
            wait for input and store in choice2

            Switch (choice2) {
            Case 1:
                print schedule
                Break
            Case 2:
                Output "Enter course number: "
                wait for input and store in courseKey
                set aCourse to schedule.Search(courseKey)
                If (aCourse is empty) output "Course is not in schedule.\n"
                Else print aCourse
                Break
            }
        }
        Set choice2 to 0
        Break
    Case 3:
        Output "Enter course number: "
        wait for input and store in courseKey
        If (coursekey is not found in schedule) {
            Output "Course does not exist.\n"
            Break
        }
        Else remove courseKey from schedule
        output courseKey " removed.\n"
        Break
    }
}
Output "goodbye.\n"
```

Evaluation

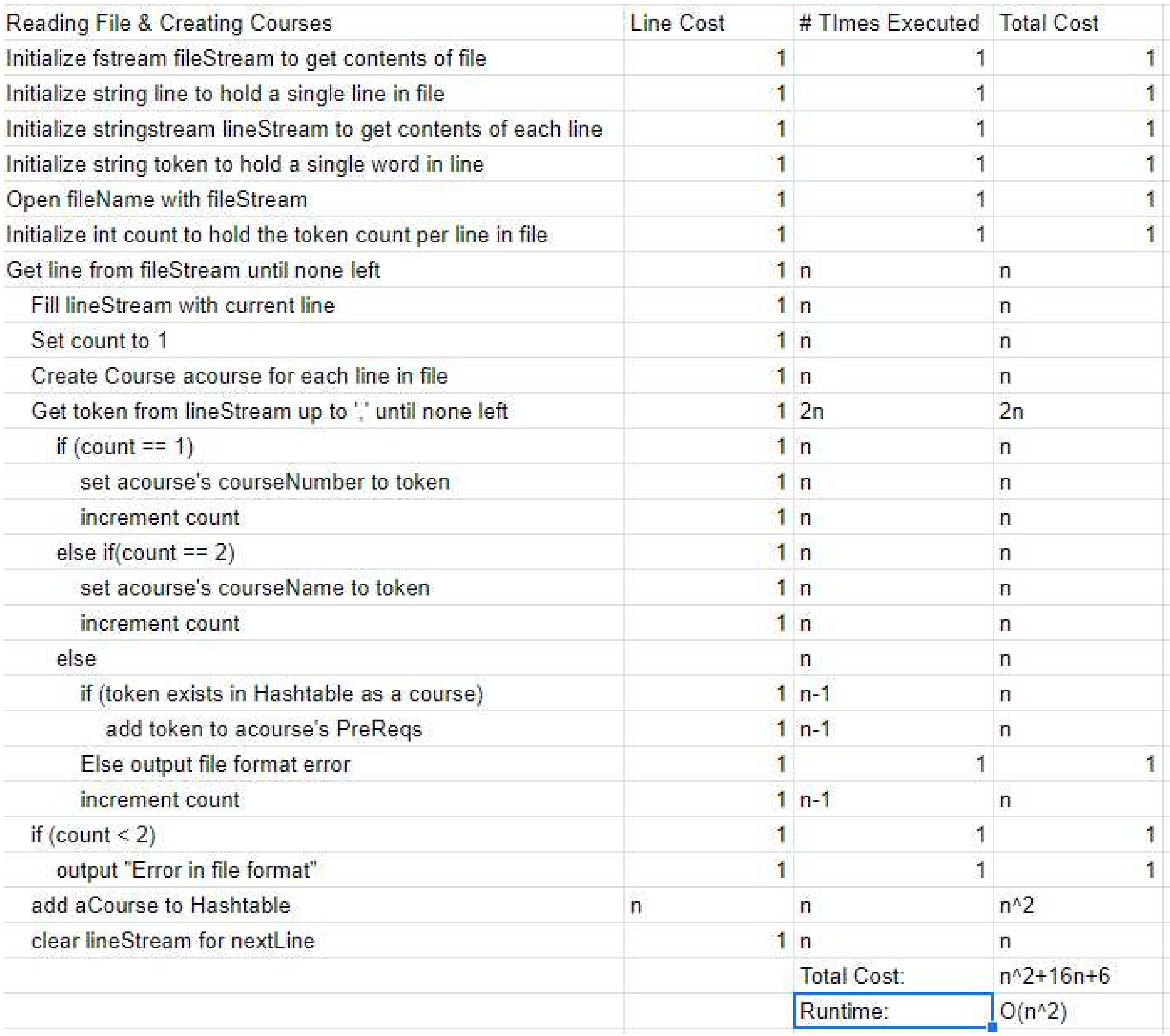
Big-O Analysis

Vector

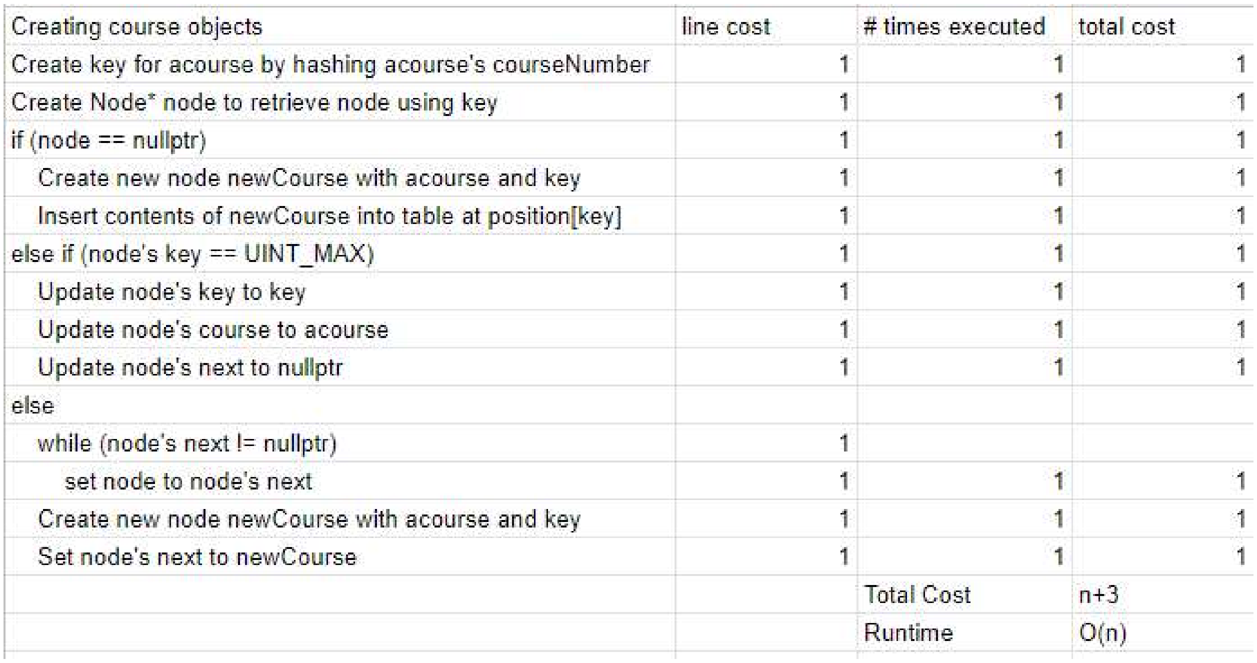


Reading File & Creating Courses	Line Cost	# TImes Executed	Total Cost
Initialize fstream fileStream to get contents of file	1	1	1
Initialize string line to hold a single line in file	1	1	1
Initialize stringstream lineStream to get contents of each line	1	1	1
Initialize string token to hold a single word in line	1	1	1
Open fileName with fileStream	1	1	1
Initialize int count to hold the token count per line in file	1	1	1
Get line from fileStream until none left	1	n	n
Fill lineStream with current line	1	n	n
Set count to 1	1	n	n
Create Course acourse for each line in file	1	n	n
Get token from lineStream up to ',' until none left	1	2n	2n
if (count == 1)	1	n	n
set acourse's courseNumber to token	1	n	n
increment count	1	n	n
else if(count == 2)	1	n	n
set acourse's courseName to token	1	n	n
increment count	1	n	n
else		n	n
if (token exists in courses as a course)	1	n	n
add token to acourse's PreReqs	1	n	n
Else output file format error	1	1	1
increment count	1	n	n
if (count < 2)	1	1	1
output "Error in file format"	1	1	1
push aCourse to back of courses	1	n	n
clear lineStream for nextLine	1	n	n
		Total Cost:	17n+6
		Runtime:	O(n)

Hashtable

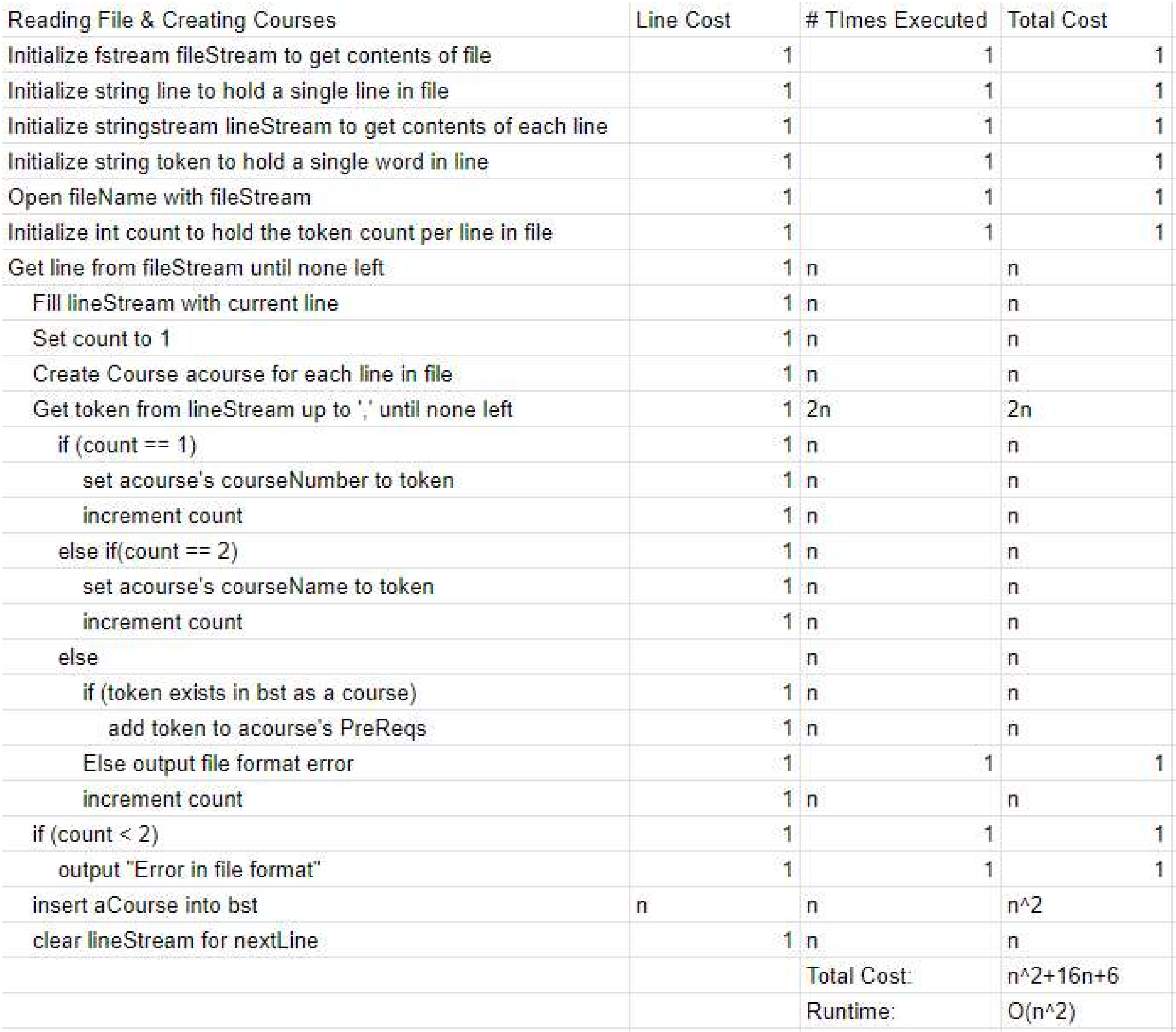


Reading File & Creating Courses	Line Cost	# TImes Executed	Total Cost
Initialize fstream fileStream to get contents of file	1	1	1
Initialize string line to hold a single line in file	1	1	1
Initialize stringstream lineStream to get contents of each line	1	1	1
Initialize string token to hold a single word in line	1	1	1
Open fileName with fileStream	1	1	1
Initialize int count to hold the token count per line in file	1	1	1
Get line from fileStream until none left	1	n	n
Fill lineStream with current line	1	n	n
Set count to 1	1	n	n
Create Course acourse for each line in file	1	n	n
Get token from lineStream up to ',' until none left	1	2n	2n
if (count == 1)	1	n	n
set acourse's courseNumber to token	1	n	n
increment count	1	n	n
else if(count == 2)	1	n	n
set acourse's courseName to token	1	n	n
increment count	1	n	n
else		n	n
if (token exists in Hashtable as a course)	1	n-1	n
add token to acourse's PreReqs	1	n-1	n
Else output file format error	1	1	1
increment count	1	n-1	n
if (count < 2)	1	1	1
output "Error in file format"	1	1	1
add aCourse to Hashtable	n	n	n^2
clear lineStream for nextLine	1	n	n
		Total Cost:	n^2+16n+6
		Runtime:	O(n^2)

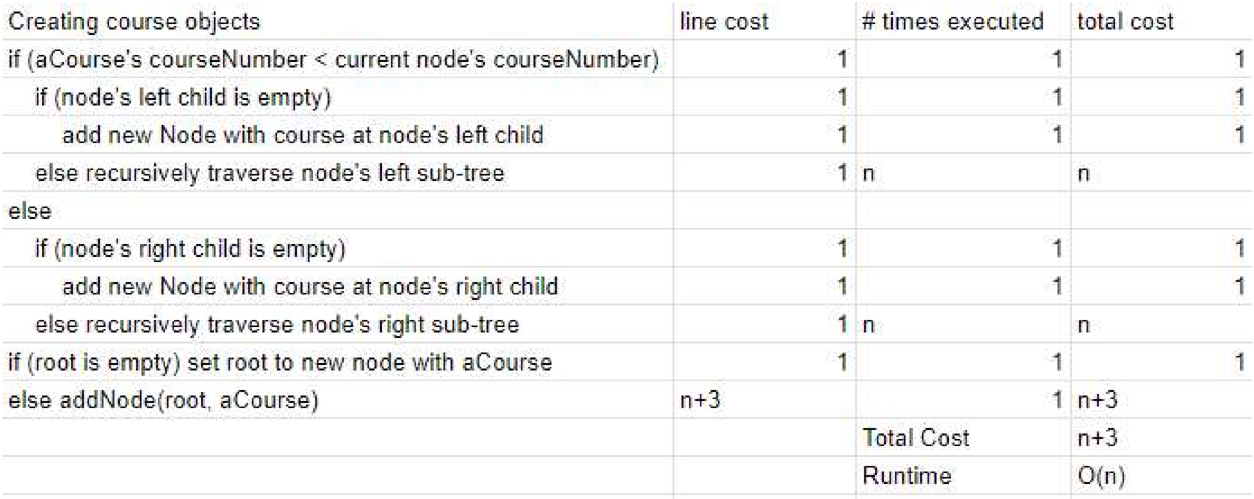


Creating course objects	line cost	# times executed	total cost
Create key for acourse by hashing acourse's courseNumber	1	1	1
Create Node* node to retrieve node using key	1	1	1
if (node == nullptr)	1	1	1
Create new node newCourse with acourse and key	1	1	1
Insert contents of newCourse into table at position[key]	1	1	1
else if (node's key == UINT_MAX)	1	1	1
Update node's key to key	1	1	1
Update node's course to acourse	1	1	1
Update node's next to nullptr	1	1	1
else			
while (node's next != nullptr)	1		
set node to node's next	1	1	1
Create new node newCourse with acourse and key	1	1	1
Set node's next to newCourse	1	1	1
		Total Cost	n+3
		Runtime	O(n)

Tree



Reading File & Creating Courses	Line Cost	# TImes Executed	Total Cost
Initialize fstream fileStream to get contents of file	1	1	1
Initialize string line to hold a single line in file	1	1	1
Initialize stringstream lineStream to get contents of each line	1	1	1
Initialize string token to hold a single word in line	1	1	1
Open fileName with fileStream	1	1	1
Initialize int count to hold the token count per line in file	1	1	1
Get line from fileStream until none left	1	n	n
Fill lineStream with current line	1	n	n
Set count to 1	1	n	n
Create Course acourse for each line in file	1	n	n
Get token from lineStream up to ',' until none left	1	2n	2n
if (count == 1)	1	n	n
set acourse's courseNumber to token	1	n	n
increment count	1	n	n
else if(count == 2)	1	n	n
set acourse's courseName to token	1	n	n
increment count	1	n	n
else		n	n
if (token exists in bst as a course)	1	n	n
add token to acourse's PreReqs	1	n	n
Else output file format error	1	1	1
increment count	1	n	n
if (count < 2)	1	1	1
output "Error in file format"	1	1	1
insert aCourse into bst	n	n	n^2
clear lineStream for nextLine	1	n	n
		Total Cost:	n^2+16n+6
		Runtime:	O(n^2)



Creating course objects	line cost	# times executed	total cost
if (aCourse's courseNumber < current node's courseNumber)	1	1	1
if (node's left child is empty)	1	1	1
add new Node with course at node's left child	1	1	1
else recursively traverse node's left sub-tree	1	n	n
else			
if (node's right child is empty)	1	1	1
add new Node with course at node's right child	1	1	1
else recursively traverse node's right sub-tree	1	n	n
if (root is empty) set root to new node with aCourse	1	1	1
else addNode(root, aCourse)	n+3	1	n+3
		Total Cost	n+3
		Runtime	O(n)

Pros/Cons

Vector

Pros:

1. Easy implementation
2. Can be searched in O(logn) time if sorted with binary search
3. Insertion at the back is constant

Cons:

1. Must be sorted to take full advantage of search capabilities
2. Removing items from front takes linear time because of shifting
3. Depending on the compiler used reallocation of vector may take up more space than needed

HashTable

Pros:

1. Direct access to items
2. Insert and delete in constant time no matter size of table
3. When implemented correctly, hash tables are the best data structures for speed

Cons:

1. Takes up more space than what is needed
2. Retrieval of elements does not preserve order
3. Randomly stores elements in memory which can cause cache misses resulting in long delays.

Tree

Pros:

1. Retrieves items in order
2. Insert and delete in O(logn) time
3. Access speed

Cons:

1. Must maintain balance for best performance
2. Can quickly cause stack overflow when using recursion
3. Shape depends on first item inserted

Recommendation

For this assignment, I recommend the use of the binary search tree, as it is the best option to store course objects. When displaying courses alphabetically, a BST will do better than a Vector or Hash Table due to ordered traversal, thus no sorting needs to be done. Vectors and Hash Tables in comparison must have sorting abilities to efficiently display courses alphabetically. The BST will (on average) take O(logn) time to complete. This is almost as good as the Hash Table's constant time, but again for the Hash Table to compete there has to be a good hash function and complete knowledge of the data that is to be stored.