CS300 – DSA

Final Project Part 1

Nathan Lebel

Travis Williamson

# Milestone Pseudocode

**Resubmit pseudocode from previous pseudocode assignments and update as necessary**. In the previous assignments, you created pseudocode for each of the three data structures: vector, hash table, and tree. Be sure to resubmit the following pseudocode for each data structure:

## Vector Pseudocode

### Open / Validate file

Vector<Course> courses;

bool validateFile(ifstream file) {  
 Vector<string> courseNumbers;  
  
 FOR each line in file  
 parse into tokens  
 put token 1 into courseNumbers  
  
 IF line has less than 2 tokens

RETURN false  
END FOR  
//O(3\*N)

FOR each line in file

parse into tokens

for tokens 3 to last token

IF token not in courseNumbers  
 RETURN false  
 END FOR

RETURN true  
 //O(3\*N)

}

void loadCourses(Vector<Course>& courses) {

string fileName  
ifstream file

DO  
 prompt “Enter the name of the .csv file containing the course information.”  
 input file name fileName  
  
 IF fileName substring (length-4, length) is not equal to .csv  
 Prompt for corrected file name

WHILE fileName.substring(length-4, length) is not equal to .csv

open file(fileName)

IF validateFile(file)

storeCourses(courses, fileStream)

ELSE

Print data formatting error

close fileName

//O(5 + 3\*n + 3\*n) -> O(5 + 6n) -> O(n)

}

### Create Course objects and store them in the Vector

void storeCourses(Vector<Course>& courses, ifstream& fileStream) {

string line

string cell

stringStream lineStream(line)

Course course

WHILE fileStream has next line

get line from fileStream into line  
getline(lineStream,cell, ‘,’)  
course.courseNumber = cell  
  
getline(lineStream,cell,’,’)  
course.courseName = cell

WHILE std::getline(lineStream, cell, ‘,’)  
 add prerequisite(s) cell to course  
  
 add course to Courses

END WHILE

close lineStream  
//O(6\*n + k\*n) where k is the # of prerequisites per course, overall O(n)

}

### Print course information for a given course

void searchCourse(Vector<Course> courses, String courseNumber) {

**for all courses**

**if the course is the same as courseNumber**

**print out the course information**

**for each prerequisite of the course**

**print the prerequisite course information**

//O(n + 1 + m), where m is the # of prerequisites, n comparisons + 1 print (if found) + m prints, overall O(n)

## Hash Table Pseudocode

### Open / Validate file

HashTable<Course> courses;

bool validateFile(ifstream file) {  
 Vector<string> courseNumbers;  
  
 FOR each line in file  
 parse into tokens  
 put token 1 into courseNumbers   
 IF line has less than 2 tokens

RETURN false  
END FOR

FOR each line in file

parse into tokens  
 for token 3 to last token

IF token not in courseNumbers  
 RETURN false  
 END FOR

RETURN true

}

void loadCourses(HashTable<Course>& courses) {

string fileName  
ifstream file

DO  
 prompt “Enter the name of the .csv file containing the course information.”  
 input file name fileName  
  
 IF fileName substring (length-4, length) is not equal to .csv  
 Prompt for corrected file name

WHILE fileName.substring(length-4, length) is not equal to .csv

open file(fileName)

IF validateFile(file)

storeCourses(courses, fileStream)

ELSE

Print data formatting error

close fileName

}

### Create Course objects and store them in the Hash Table

struct Course {

string courseNumber;  
 string name;  
 vector<string> prerequisites;

}

struct Node {

Course course;  
 unsigned int key;

Node\* next;

Node() {  
 key = UINT\_MAX;  
 next = nullptr;  
 }  
  
 Node(Course course) : Node() {  
 this->course = course;  
 }  
  
 Node(Course course, unsigned int key) : Node(course) {  
 this->key = key;  
 }

}

vector<Node> nodes;  
unsigned int tableSize = 11;  
  
void storeCourses(HashTable<Course> courses, ifstream& inFile) {

string nextLine  
 string cell  
 stringStream lineStream(nextLine)

int key;  
 Course course  
   
 WHILE inFile has next line  
   
 get line from inFile into nextLine  
 getline(lineStream,cell,’,’)  
 course.courseNumber = cell  
   
 generate hash from courseNumber

getline(lineStream,cell,’,’)  
 course.courseName = cell

WHILE getline(lineStream, cell, ‘,’)

Push cell to course.prerequisites  
END WHILE  
add course to courses

END WHILE  
close lineStream

}

### Print course information for a given course

void searchCourse(HashTable<Course> courses, String courseNumber) {

FOR Course course in courses

IF (course.courseNumber == courseNumber) {  
 PRINT node.course.courseNumber

PRINT node.course.name

FOR i = 0 to node.course.prerequisites.size()  
 PRINT node.course.prerequisites.at(i)

}

}

## Binary Search Tree Pseudocode

### Open / Validate file

BinarySearchTree<Course> courses;

bool validateFile(ifstream file) {  
 Vector<string> courseNumbers;

FOR each line in file  
 parse into tokens  
 put token 1 into courseNumbers   
 IF line has less than 2 tokens  
 RETURN false  
END FOR

FOR each line in file  
 parse into tokens  
 FOR token 3 to last token   
 IF token not in courseNumbers  
 RETURN false   
 END FOR  
 RETURN true  
 END FOR  
}  
  
void loadCourses(BinarySearchTree<Course>& courses) {

string fileName

ifstream file

DO

prompt “Enter the name of the .csv file containing the course information.”

input file name fileName

IF fileName substring (length-4, length) is not equal to .csv  
 Prompt for corrected file name

WHILE fileName.substring(length-4, length) is not equal to .csv

open file(fileName)

IF file.is\_open() and validateFile(file)

storeCourses(courses, fileStream)

ELSE

Print data formatting error

close fileName

}

### Create Course objects and Store them in the BST

struct Course {

string courseNumber;  
 string name;  
 vector<string> prerequisites;

}

struct Node {

Course course;  
 Node \*left;  
 Node \*right;  
   
 Node() {  
 left = nullptr;  
 right = nullptr;  
 }  
  
 Node(Bid aBid) : Node() {

Bid = aBid;

}

}

class BinarySearchTree {

private:   
 Node \*root;  
  
 addNode(Node\* node, Course course) {

IF node is null

node equals new Node(course)  
 ELSE IF node.course.courseNumber > course.courseNumber  
 addNode(node.left,course)  
 ELSE

addNode(node.right,course)

}

}

void storeCourses(BinarySearchTree<Course> courses, ifstream& inFile) {

string nextLine

string cell

stringStream lineStream(nextLine)

Course course

WHILE inFile has next line

get line from inFile into nextLine

getline(lineStream,cell,’,’)

course.courseNumber = cell

getline(lineStream,cell,’,’)

course.courseName = cell

WHILE getline(lineStream, cell, ‘,’)

Push cell to course.prerequisites

END WHILE

add course to courses from courses.root

END WHILE

close lineStream

}

### Print course information for a given course

Course searchTree(Node\* node, String courseNumber) {

IF node.course.courseNumber == courseNumber

RETURN node.course  
 ELSE IF node.course.courseNumber < courseNumber  
 searchTree(node.left, courseNumber)  
 ELSE  
 searchTree(node.right, courseNumber)  
 END IF  
  
 RETURN new course //No course found  
}  
  
void searchCourse(BinarySearchTree<Course> courses, String courseNumber) {

Course locatedCourse = searchTree(courses.root, courseNumber)  
  
IF locatedCourse.courseNumber == courseNumber  
 PRINT course information for locatedCourse  
END IF

# Pseudocode – Menu and Print Sorted List

**Create pseudocode for a menu**. The menu will need to perform the following actions:  
void MainMenu(\*vector / BinarySearchTree / HashTable\*<Course>& courses) {  
 int userOption, courseNumber  
 bool sorted;  
   
 switch userOption  
 case 1: loadCourses(courses)

sorted = false  
 break  
 case 2: if not sorted  
 sort courses  
 sorted = true  
 end if  
 printAllCourses(courses)   
 break  
 case 3: print enter a course number  
 user input to courseNumber  
 printCourse(courseNumber)  
 break  
 case 9:   
 default: exit program  
 end switch

}

**Design pseudocode that will print out the list of the courses in the Computer Science program in alphanumeric order.** Continue working with the Pseudocode Document linked in the Supporting Materials section. Note that you will design for the same three data structures that you have been using in your previous pseudocode milestones: vector, hash table, and tree. This time, you will create the final pieces of pseudocode that you will need for ABCU’s advising program. To complete this part of the process, do the following actions:

* 1. Sort the course information by alphanumeric course number from lowest to highest.
  2. Print the sorted list to a display.

|  |  |
| --- | --- |
| Vector | vector<Course> courses; |
| Vector Sort (Quicksort) | int partition(vector<Course>& courses, int begin, int end) {  int low = begin – 1  int mid = begin + (end – begin) / 2  int high = end + 1   string pivot = courses[mid].courseNumber  while true  while courses[low].courseNumber < pivot  increment low  end while   while courses[high].courseNumber > pivot  decrement high  end while  If low >= high  return high  swap(bids[low],bids[high])  end while  return high  }  void quickSort(vector<Course>& courses, int begin, int end) {  if end – begin < 2  return;   if begin >= 0 and end >= 0 and begin < end  int mid = partition(courses,begin,end)  quickSort(courses,begin,mid)  quickSort(courses,mid+1,end)  end if }  void Vector::Sort(Vector<Course>& courses) {  quickSort(courses, 0, courses.size())  } |
| Vector Print | void PrintAllCourses(vector<Course>& courses) {  for each Course c in courses  print course information  } |
| Hash Table | HashTable<Course> courses |
| Hash Table – Sort | void HashTable::Sort() {  //Using (courseNumber.substr(4,courseNumber.length()) / 50) - 2 as the hash value / key during creation allows us to sort the top level nodes at insertion time i.e. all 100-149, all 150 – 199. We then quicksort the linked list of nodes at each head node.   for each node in nodes  node = quickSort(node)  }  Node\* quickSort(Node\* head) {  Node\* tail = getTail(head)   quickSortHelper(head,tail)  return head  }  Node\* getTail(Node\* currentNode) {  while currentNode != nullptr and currentNode->next != nullptr  currentNode = currentNode->next   return currentNode }  Node\* partition(Node\* head, Node\* tail) {  Node\* pivot = head;   Node\* previousNode = head;  Node\* currentNode = head;   while currentNode != tail->next {  if currentNode.courseNumber < pivot.courseNumber  swap(currentNode.course, previousNode->next.course)  currentNode = currentNode->next  }    swap(pivot.course,previousNode.course)  return previousNode }  void quickSortHelper(Node\* head, Node\* tail) {  if head == nullptr or head == tail  return   Node\* pivot = partition(head,tail)    quickSortHelper(head,pivot)   quickSortHelper(pivot->next, tail)  } |
| Hash Table – Print | void PrintAllCourses() {  for node in nodes  if node.key is valid key  print course information  while node.next != nullptr  print course information  node = node.next  end while  end if  end for  } |
| Binary Search Tree | BinarySearchTree<Course> courses; |
| BST – Sort | //Printing the BST using in order traversal prints the elements in order |
| BST - Print | void InOrderPrint(Node\* node) {  inOrderPrint(node->left)  print course information  inOrderPrint(node->right)  } |

# Evaluation

1. **Evaluate the run time and memory of data structures that could be used to address the requirements**. In previous assignments, you created pseudocode to define how the program opens the file, reads the data from the file, parses each line, and checks for formatting errors and to show how to create course objects so that one course object holds data from a single line from the input file.
   1. Using the pseudocode you wrote for the previous assignments, analyze the worst-case running time of each, reading the file and creating course objects, which will be the Big O value. This analysis should not include the pseudocode written for the menu or the search/print functions Print Course List (Option 2) above. To complete this part of the project, do the following actions:  
      1. Specify the cost per line of code and the number of times the line will execute. Assume there are n courses stored in the data structure.
      2. Assume the cost for a line to execute is 1 unless it is calling a function, in which case the cost will be the running time of that function.

|  |  |  |  |
| --- | --- | --- | --- |
| Worst Case Time Complexity | Vector | Hash Table | Binary Search Tree |
| Read/Validate File | O(5 + 6n) -> O(n) | O(n) | O(h) |
| Create/Store Course Objects | O(6n + k\*n) -> O(n) | O(n) | O(h) |
| Print Course Information | O(n + k\*n) -> O(n) | O(n) | O(h) |

1. Based on the advisor’s requirements, analyze each of the vector, hash table, and tree data structures. **Explain the advantages and disadvantages of each structure in your evaluation.**Vector advantages include ease of use and built in functionality, O(1) insert at end and direct access times, however vectors / arrays suffer from the slowest average search / sort time when compared to a hash table with a well-designed hashing function or binary search tree.  
     
   Hash table advantages include O(1) average case time complexity for insert / search / deletion of elements, given no collisions. Disadvantages include a higher storage complexity. However, depending on how collisions are handled, a hash table can have a worst case time complexity of O(n). An additional disadvantage is the unsorted nature of a hash table, without specific handling and a familiarity with the data being inserted, printing a hash tables content sorted carries additional overhead. In this case, each linked list would need to be sorted using quicksort / mergesort for each root node.  
     
     
   Binary Search Tree advantages include O(h) average case insertion time and seek time, while removing element from a BST is a more complex process than removing an element from a vector or a hash table. Disadvantages include the inability to directly access an element by index, however this disadvantage is shared by the hash table.
2. Now that you have analyzed all three data structures, **make a recommendation for which data structure you plan to use in your code**. Provide justification for your recommendation based on the Big O analysis results and your analysis of the three data structures.

I plan to use a Binary Search Tree in my code, as while a BST has a worst case time complexity of O(n) for insertion, search, and deletion the average time complexity is O(h) where h is the height of the tree. Most importantly, printing a BST sorted in order does not carry the additional overhead of vector / linked list based hash table sorting, assuming collisions are handled such that they would prevent the hash table from being unsortable. While a hash table would be an easy choice for the average case time complexity of O(1) for insertion/deletion/search options, sorting the table requires an additional O(log(n)) time for each unsorted bucket.