Tanner Hunt

CS300 DSA: Analysis and Design

April 9, 2025

Project One: Data Structure Pseudocode

Contents

[1. Menu Pseudocode 2](#_Toc195124027)

[1.1 main method 2](#_Toc195124028)

[1.2 Helper method: print Menu 3](#_Toc195124029)

[1.3 Helper method: string split 3](#_Toc195124030)

[2. Vector Pseudocode 3](#_Toc195124031)

[2.1 File reading 3](#_Toc195124032)

[2.2 Course and Node creation 3](#_Toc195124033)

[2.3 course creation 3](#_Toc195124034)

[2.4 course deletion 3](#_Toc195124035)

[2.5 Course Search 3](#_Toc195124036)

[2.6 course printing 4](#_Toc195124037)

[3. Hash Table Pseudocode 4](#_Toc195124038)

[3.1 File reading 4](#_Toc195124039)

[3.2 Course and Node structures, init 5](#_Toc195124040)

[3.3 Insert/ course creation 5](#_Toc195124041)

[3.4 Hash Key calculation 5](#_Toc195124042)

[3.5 Course Search 5](#_Toc195124043)

[3.6 Course deletion 6](#_Toc195124044)

[3.7 Print all courses (sorted and unsorted) 6](#_Toc195124045)

[4. Tree Pseudocode 6](#_Toc195124046)

[4.1 File Reading 6](#_Toc195124047)

[4.2 course and node structure 7](#_Toc195124048)

[4.3 BST insert 7](#_Toc195124049)

[4.4 BST delete 8](#_Toc195124050)

[4.5 BST search 8](#_Toc195124051)

[4.6 BST print (ordered) 8](#_Toc195124052)

[5. Evaluation 8](#_Toc195124053)

[5.1 Vector Big O 8](#_Toc195124054)

[5.1.1 File Reading -- O(n^3) 8](#_Toc195124055)

[5.1.2 Insert -- O(n) 10](#_Toc195124056)

[5.1.3 Delete – O(n) 10](#_Toc195124057)

[5.2 Hash\_map Big O 11](#_Toc195124058)

[5.2.1 Loading data -- O(n) 11](#_Toc195124059)

[5.2.2 Insert -- O(1) 12](#_Toc195124060)

[5.2.3 delete – O(1) 12](#_Toc195124061)

[5.2.4 calculate Hash O(1) 13](#_Toc195124062)

[5.3 Tree Big O 13](#_Toc195124063)

[5.3.1 Loading data O(nlog(n)) 13](#_Toc195124064)

[5.3.2 Insert O(log(n)) 14](#_Toc195124065)

[5.3.3 delete O(log(n)) 14](#_Toc195124066)

[5.4 Pros and Cons 15](#_Toc195124067)

[5.4.1 Worst time 15](#_Toc195124068)

[5.4.2 Expected time 16](#_Toc195124069)

[5.5 Recommendation 16](#_Toc195124070)

# 1. Menu Pseudocode

## 1.1 main method

1. Create Parser object

2. String input = “”

3. While input doesn’t equal “9”

4. PrintMenu();

5. String[] inputs = splitInputIntoTwo(input)

6. If inputs[0] == “1”

7. Parser. Load data file

8. Else if inputs[0] == “2”

9. Parser.datastructure.print()

10. Else if inputs[0] == “3”

11. String coursId = inputs[1]

12. Parser.search(courseId)

13. PrintCourse(courseId)

14. Input = getLine()

15.

## 1.2 Helper method: print Menu

1. Void printMenu(){

2. Print “type 1 to load the data structure”

3. Print “type 2 to print all courses in order

4. Print “type 3 and the coursId to print an individual courses details

5. Print “type 9 to exit this program”

6. }

7.

## 1.3 Helper method: string split

1. &String[] SplitInputIntoTwo(String input){

2. Int index = 0

3. String1 = “”

4. String2 = ””

5.

6. //copy substring from input into string one until space is found

7. Char character = input.at(index)

8. While(character != ‘ ‘ and index < input.length){

9. String1 = String1 + character

10. Index += 1

11. Character = input.at(index)

12. }

13.

14. //copy substring from input into string two

15. Index = index + 1

16. While(index < input.length){

17. String2 = String2 + character

18. Index += 1

19. Character = input.at(index)

20. }

21.

22. String[] stringsToReturn = {String1, String2}

23. Return &stringsToReturn;

24.

25. }

26.

# 2. Vector Pseudocode

## 2.1 File reading

1. Class Parser

2. Void LoadData(){

3. Define filePath;

4. Define queue of classes to be validated

5. Create a linked\_list of courses using <string, course>

6.

7. Open input file stream(ifstream) with filePath

8. If ifstream is open

9. //read each file, read the course ID, course name, then prerequisites

10. While getline returns a new line

11. Split the line into tokens deliminited by the comma

12. If line has less than 2 parameters, skip line

13. Create a new Course

14. Set the first token as the course ID

15. Set the second token as the course name

16.

17. //if the course has prerequisites, load them into the course struct.

18. //then add it to a queue to validate its prerequisites exist after all

19. //the courses have been loaded into memory

20. If there are more tokens

21. push all additional tokens into the courses vector of prereq’s

22. add the new course to the queue of courses to validate prereq’s

23. add this course to the vector of courses

24.

25. Close file

26.

27. //delete courses whose prerequisite courses does not exist

28. For each course in queue of courses to be validated

29. For each prerequisite course in course

30. If prerequisite course ID is not in vector of courses

31. Delete courseID from vector

32.

33. }

34.

## 2.2 Course and Node creation

1. Course Struct and storage

2. Struct Course{

3. String course ID

4. String course name

5. Vector<String> prerequisites course ID’s

6. }

7.

## 2.3 course creation

1. //insert a course in alphanumeric order

2. Public void AddCourse(Node\* node){

3. Node\* curNode = head

4.

5. //special case where node inserts to head of list

6. If(node.courseId < head.courseId){

7. Node.next = head

8. Head = node

9. Return;

10. }

11. While curNode is not null

12. If node < curNode.next

13. Node.next = curNode.next

14. curNode.next = node

15. return

16. else curNode = curNode.next

17. }

18.

## 2.4 course deletion

1. Public void Delete(Course course){

2. Node\* curNode = head

3. Node\* prevNode = nullptr

4. Node\* sucNode = nullptr

5. While curNode not null

6. If curNode.course.id == course.id

7. sucNode = curNode.next

8. If prevNode is not nullptr

9. prevNode.next = sucNode

10. delete curNode

11. return

12. else

13. prevNode = curNode

14. curNode = curNode.next

15. }

16.

## 2.5 Course Search

1. Public Boolean hasCourse(Course course){

2. Node\* curNode = head

3. While curNode is not null

4. If curNode.course.id equals course.id

5. Return true

6. Else curNode = curNode.next

7. //if not found

8. Return false

9. }

10.

## 2.6 course printing

9. Public Course getCourses(string courseID){

10. Node\* curNode = head

11. While curNode is not null

12. printCourseDetails(curNode.course)

13. curNode = curNode.next

14.

15. }

1. Public void PrintCourseDetails(Course course){

2. Print course id, course name, prerequisites

3. }

4.

# 3. Hash Table Pseudocode

## 3.1 File reading

1. Class Parser

2. LoadData(){

3. Define filePath;

4. Define queue of classes to be validated

5. Create a hashmap of courses using <string, courseNode> as key, value pairs

6.

7. Open input file stream(ifstream) with filePath

8. If ifstream is open

9. //read each file, read the course ID, course name, then prerequisites

10. While getline returns a new line

11. Split the line into tokens deliminited by the comma

12. If line has less than 2 parameters, skip line

13. Create a new Course

14. Set the first token as the course ID

15. Set the second token as the course name

16.

17. //if the course has prerequisites, load them into the course struct.

18. //then add it to a queue to validate its prerequisites exist after all

19. //the courses have been loaded into memory

20. If there are more tokens

21. push all additional tokens into the courses vector of prereq’s

22. add the new course to the queue of courses to validate prereq’s

23. add this course to the hashmap of courses using its courseID as a key

24.

25. Close file

26.

27. //delete courses whose prerequisite courses does not exist

28. For each course in queue of courses to be validated

29. For each prerequisite course in course

30. If prerequisite course ID is not in hashmap of courses

31. Delete courseID from hashmap

32. Delete course

33. }

34.

## 3.2 Course and Node structures, init

1. Course struct

2. String key

3. String course name

4. Vector<Course> prerequisites

5.

6. CourseNode

7. Value Course

8. Node\* nextNode

9.

10. Initializing HashMap

11. Create vector of course nodes

12.

## 3.3 Insert/ course creation

**Creating course objects and putting in appropriate structure**

1. Void insertCourse(Course course){

2. If(course\_key exists in courses.keys)

3. Update value of course.key

4. Return;

5.

6. Set hashkey to CalculateHashKey()

7. Get the bucket for the hashkey in vector of course nodes

8. Set the next node for the course to the dummy nodes next course

9. Set the dummy nodes next course to this course

10.

11. }

12.

## 3.4 Hash Key calculation

1. calculateHashKey(string key){

2. hashvalue = 0

3. hashmultiplier = 2

4. for each character in string

5. hashvalue = hashvalue \* multiplier + character\_ascii\_code

6.

7. return hashvalue % tablesize

8. }

9.

## 3.5 Course Search

1. SearchForCourse(Course course){

2. get the hashkey from calculateHashKey(course key)

3. start curnode at the bucket for that hash key in course nodes

4. while the curnode is not null

5. check if curnodes course == course

6. return node if matches

7. else, curnode = next

8. if no node is found return nullptr

9. }

10.

## 3.6 Course deletion

1. Void Delete(Course course){

2. Get the hashkey from calculateHashKey(course key)

3. Start curNode at the bucket for that hashkey in course nodes

4. While curNode is not null

5. If curNode.next == key to delete

6. Node deleteNode = curNode.next

7. curNode.next = deleteNode.next

8. delete deleteNode

9. Return

10. }

11.

## 3.7 Print all courses (sorted and unsorted)

1. //hashtables loses the ordering of a set. Load the hashtable contents into an ordered set to //print

2. PrintSorted(){

3. Create BST

4. For each bucket in course nodes

5. For each node in this buckets linked list

6. Insert into BST

7. //BST is sorted by default, so printing by preorder prints the ordered list

8. BST.PrintPreorder()

9. }

10.

**Printing out code (unsorted)**

1. For each linked-list of course nodes in vector of linked lists

2. Start at first node

3. While node is not null

4. Print node id, course name

5. Foreach prerequisite course

6. Print course id, course name

7. Move to next node

8.

# 4. Tree Pseudocode

## 4.1 File Reading

1. Class Parser

2. LoadData(){

3. Define filePath;

4. Define queue of classes to be validated

5. Create a BST of courses

6.

7. Open input file stream(ifstream) with filePath

8. If ifstream is open

9. //read each file, read the course ID, course name, then prerequisites

10. While getline returns a new line

11. Split the line into tokens deliminited by the comma

12. If line has less than 2 parameters, skip line

13. Create a new Course

14. Set the first token as the course ID

15. Set the second token as the course name

16.

17. //if the course has prerequisites, load them into the course struct.

18. //then add it to a queue to validate its prerequisites exist after all

19. //the courses have been loaded into memory

20. If there are more tokens

21. push all additional tokens into the courses vector of prereq’s

22. add the new course to the queue of courses to validate prereq’s

23. add this course to the BST of courses

24.

25. Close file

26.

27. //delete courses whose prerequisite courses does not exist

28. For each course in queue of courses to be validated

29. For each prerequisite course in course

30. If BST.Search(courseID) is null

31. BST.Delete(courseID)

32. }

33.

## 4.2 course and node structure

1. Course struct

2. String key

3. String course name

4. Vector<Course> prerequisites

5.

6. Class BST

7. CourseNode

8. Value Course

9. Node\* nextNode

10.

## 4.3 BST insert

1. InsertNodeIntoBST()

2. currentNode = root

3. If correntNode is null

4. Root = node

5. While currentNode is not null

6. If node <= currentNode and currentNode.left is not null

7. Current node = currentNode.left

8. Else if node > currentNode and currentNode.right is not null

9. Current node = currentNode.right

10. Else if node<= currentNode and currentNode.left IS null

11. Set left node to the new node

12. Return

13. Else

14. Set the right node to the new node

15. return

16.

## 4.4 BST delete

1. Delete()

2. Set the current node to the root

3. Using the search algorithm, find the node to be deleted

4. If the node does not exist, exit

5. If the node has one child

6. Set the nodes parent to the child

7. Delete this node

8. return

9. If the node has one child and is the root

10. Set the root to the child

11. Delete this node

12. Return

13. If the node has two children

14. Find the successor node

15. Go to the current nodes right child

16. Recursively go each nodes left child after that until the end

17. If the node has a parent, set its pointer to the successor node

18. Set the successor nodes pointer to the other child

19.

## 4.5 BST search

1. Search(val)

2. Set the current node to the root

3. While current node is not null

4. If current node and val match

5. Return current node

6. Else if val < current node

7. Current node = left

8. Else

9. Current node = right

10.

## 4.6 BST print (ordered)

1. PrintBST(Node)

2. If this node is null, exit

3. PrintNode(curNode)

4. Recursively call printPreorder( left node)

5. Recursively call printPreorder(right node)

6.

# 5. Evaluation

## 5.1 Vector Big O

### 5.1.1 File Reading -- O(n^3)

|  |  |  |
| --- | --- | --- |
| Code | O() | comment |
| Define filePath | 1 |  |
| Define queue of classes to be validated | 1 |  |
| Create a linked\_list of courses using <string, course> | 1 |  |
| Open input file stream(ifstream) with filePath | 1 |  |
| If ifstream is open | 1 |  |
| While getline returns a new line | n |  |
| Split the line into tokens delimited by the comma | 1 |  |
| If line has less than 2 parameters, skip line | 1 |  |
| Create a new Course | 1 |  |
| Set the first token as the course ID | 1 |  |
| Set the second token as the course name | 1 |  |
| If there are more tokens, push all additional tokens into the courses vector of prereq’s | 1 |  |
| Add the new course to the queue of courses to validate prereq’s | 1 |  |
| Add this course to the vector of courses | n | New courses are inserted in order. This could be improved by an append function, but would require implementing a sort() method which has an expensive time complexity in a linked list. |
| Close file | 1 |  |
| For each course in queue of courses to be validated | n | Assuming nearly every course has a prerequisite |
| For each prerequisite course in course | 1 | Assuming the number of prerequisite courses is small for an individual course |
| If prerequisite course ID is not in vector of courses | n | Cost of searching for a matching prereq. Course |
| Delete courseID from vector | n | Must read through list to delete a node and patch next pointers. This could be improved by adding a previous pointer to every node and allowing delete() to directly delete a node |
| Delete course from queue | 1 |  |

### 5.1.2 Insert -- O(n)

|  |  |  |
| --- | --- | --- |
| Code | O() | comment |
| Node\* curNode = head | 1 |  |
| If(node.courseId < head.courseId){ | 1 |  |
| Node.next = head | 1 |  |
| Node.next = head | 1 |  |
| Head = node | 1 |  |
| Return; | 1 |  |
| While curNode is not null | N | Possibly searching through every node before adding. Best case O(1), worst case O(n) |
| If node < curNode.next | 1 |  |
| Node.next = curNode.next | 1 |  |
| curNode.next = node | 1 |  |
| else curNode = curNode.next | 1 |  |
|  |  |  |

### 5.1.3 Delete – O(n)

|  |  |  |
| --- | --- | --- |
| Code | O() | Comment |
| Node\* curNode = head | 1 |  |
| Node\* prevNode = nullptr | 1 |  |
| Node\* sucNode = nullptr | 1 |  |
| While curNode not null | N | Possibly read through all nodes. Best case runtime complexity is O(1). |
| If curNode.course.id == course.id | 1 |  |
| sucNode = curNode.next | 1 |  |
| If prevNode is not nullptr | 1 |  |
| prevNode.next = sucNode | 1 |  |
| delete curNode | 1 |  |
| Return | 1 |  |
| Else | 1 |  |
| prevNode = curNode | 1 |  |
| curNode = curNode.next | 1 |  |

## 5.2 Hash\_map Big O

### 5.2.1 Loading data -- O(n)

|  |  |  |
| --- | --- | --- |
| Code | O() | Comment |
| Define filePath | 1 |  |
| Define queue of classes to be validated | 1 |  |
| Create a hashmap of courses using <string, courseNode> as key, value pairs | 1 |  |
| Open input file stream (ifstream) with filePath | 1 |  |
| If ifstream is open | 1 |  |
| While getline returns a new line | N | Read every course from file |
| Split the line into tokens delimited by the comma | 1 |  |
| If line has less than 2 parameters, skip line | 1 |  |
| Create a new Course | 1 |  |
| Set the first token as the course ID | 1 |  |
| Set the second token as the course name | 1 |  |
| If there are more tokens, push all additional tokens into the courses vector of prereq’s | 1 | Assumed that courses have a finite number of prerequisites |
| Add the new course to the queue of courses to validate prereq’s | 1 |  |
| Add this course to the hashmap of courses using its courseID as a key | 1 | Hashmap.insert() has expected complexity O(1). It could be O(n) for a hashmap with n collisions, but that is a failure in design and not expected |
| Close file | 1 |  |
| For each course in queue of courses to be validated | N | Assuming every course has prerequisites |
| For each prerequisite course in course | 1 |  |
| If prerequisite course ID is not in hashmap of courses | 1 | Search has an expected complexity of O(1). A table with only collisions could have a complexity of O(n), but that is not expected. |
| Delete courseID from hashmap | 1 | Delete also has an expected complexity of O(1), but could be up to O(n) in a hashmap using linked lists with many collisions. |
| Delete course | 1 |  |

### 5.2.2 Insert -- O(1)

|  |  |  |
| --- | --- | --- |
| Code | O() | Comment |
| If(course\_key exists in courses.keys) | 1 | Assuming there are few collisions. Worst case runtime is O(n) |
| Update value of course.key | 1 |  |
| Return | 1 |  |
| Set hashkey to CalculateHashKey() | 1 |  |
| Get the bucket for the hashkey in vector of course nodes | 1 |  |
| Set the next node for the course to the dummy nodes next course | 1 |  |
| Set the dummy nodes next course to this course | 1 |  |

### 5.2.3 delete – O(1)

|  |  |  |
| --- | --- | --- |
| Code | O() | comment |
| Get the hashkey from calculateHashKey(course key) | 1 |  |
| Start curNode at the bucket for that hashkey in course nodes | 1 |  |
| While curNode is not null | 1 | Assuming small bucket sizes. A hash table with many collisions has a worst-case complexity of O(n) |
| If curNode.next == key to delete | 1 |  |
| Node deleteNode = curNode.next | 1 |  |
| curNode.next = deleteNode.next | 1 |  |
| delete deleteNode | 1 |  |
| Return | 1 |  |

### 5.2.4 calculate Hash O(1)

|  |  |  |
| --- | --- | --- |
| Code | O() | Comment |
| hashvalue = 0 | 1 |  |
| hashmultiplier = 2 | 1 |  |
| for each character in string | 1 |  |
| hashvalue = hashvalue \* multiplier + character\_ascii\_code | 1 |  |
| return hashvalue % tablesize | 1 |  |

## 5.3 Tree Big O

### 5.3.1 Loading data O(nlog(n))

|  |  |  |
| --- | --- | --- |
| Code | O() | Comment |
| Define filePath; | 1 |  |
| Define queue of classes to be validated | 1 |  |
| Create a BST of courses | 1 |  |
| Open input file stream(ifstream) with filePath | 1 |  |
| If ifstream is open | 1 |  |
| While getline returns a new line | N |  |
| If line has less than 2 parameters, skip line | 1 |  |
| Create a new Course | 1 |  |
| If there are more tokens | 1 |  |
| push all additional tokens into the courses vector of prereq’s | 1 |  |
| add the new course to the queue of courses to validate prereq’s | 1 |  |
| add this course to the BST of courses | Log(n) | Must search down the length of the tree to insert. An unbalanced tree may have a complexity of O(n) |
| Close file | 1 |  |
| For each course in queue of courses to be validated | N | Assuming all courses have prerequisites |
| For each prerequisite course in course | 1 |  |
| If BST.Search(courseID) is null | Log(n) | Searching for a course may span the height of the tree, log(n) |
| BST.Delete(courseID) | Log(n) | Course must be searched for again |

### 5.3.2 Insert O(log(n))

|  |  |  |
| --- | --- | --- |
| **Code** | **O()** | **Comment** |
| currentNode = root | 1 |  |
| If correntNode is null | 1 |  |
| Root = node | 1 |  |
| While currentNode is not null | Log(n) | Height is expected to be log(n), but could be n in an unbalanced tree |
| If node <= currentNode and currentNode.left is not null | 1 |  |
| Current node = currentNode.left | 1 |  |
| Else if node > currentNode and currentNode.right is not null | 1 |  |
| Current node = currentNode.right | 1 |  |
| Else if node<= currentNode and currentNode.left IS null | 1 |  |
| Set left node to the new node | 1 |  |
| Return | 1 |  |
| Else | 1 |  |
| Set the right node to the new node | 1 |  |
| Return | 1 |  |

### 5.3.3 delete O(log(n))

|  |  |  |
| --- | --- | --- |
| **Code** | **O()** | **Comment** |
| Set the current node to the root | 1 |  |
| Using the search algorithm, find the node to be deleted | Log(n) | Assuming a balanced tree. An unbalanced tree may have height n |
| If the node does not exist, exit | 1 |  |
| If the node has one child | 1 |  |
| Set the nodes parent to the child | 1 |  |
| Delete this node | 1 |  |
| Return | 1 |  |
| If the node has one child and is the root | 1 |  |
| Set the root to the child | 1 |  |
| Delete this node | 1 |  |
| Return | 1 |  |
| If the node has two children | 1 |  |
| Find the successor node | 1 | We have already assumed this node is at the bottom of the tree for the longest lookup time, so there are no more nodes to search for. The time complexity of this row and the second row could easily be swapped because their complexity is dependent on one another |
| Go to the current nodes right child | 1 |  |
| Recursively go each nodes left child after that until the end | 1 |  |
| If the node has a parent, set its pointer to the successor node | 1 |  |
| Set the successor nodes pointer to the other child | 1 |  |

## 5.4 Pros and Cons

The pros and cons of each data structure can be summarized in the tables below:

### 5.4.1 Worst time

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Data structure** | **File loading** | **insert** | **delete** | **search** | **Ordered?** |
| Vector | n^3 | N | n | n | Yes |
| Hashmap | n^3 | N | n | N | No |
| BST | n^3 | N | n | n | Yes |

### 5.4.2 Expected time

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Data structure** | **Total runtime** | **Insert** | **delete** | **search** | **Ordered?** |
| Vector | N^2 | N | n | n | Yes |
| Hashmap | N | 1 | 1 | 1 | No |
| BST | Nlog(n) | Log(n) | Log(n) | Log(n) | yes |

## 5.5 Recommendation

The best data structure for this project is a binary search tree. Vectors have the benefit of being easy to implement. While the best case for a vector has fast insertion and lookup times (if elements are found early on the list), the expected time complexity is much higher due to insertions being in a random order. The worst case time complexity could also be improved if the vector used a doubly linked list. The current time complexity estimates are done using a singly linked list. Hashmaps have the benefit of fast insertion, searches, and deletions – which would reduce the time complexity of the program – but has the downside of losing the ordering of elements. A direct access hashmap could sidestep this, but at the cost of possibly outrageous memory complexity. Seeing as printing an alpha-numerically ordered course list is a project requirement, hashmaps are not a valid data structure for this project. Binary search trees provide the most stable expected time complexity with fast insertion, deletions, and searches. The worst-case time complexity for a binary search tree matches that of the vector if elements are inserted in either ascending or descending order. This outcome is unexpected if elements are added in random order. A balanced tree could also be guaranteed with a red-black tree or AVL tree at the cost of a more complex implementation.