**Project One – Data Structure Pseudocode**

Craig W. O’Loughlin

Southern New Hampshire University

CS300: DSA: Analysis and Design

Amy Burns, MBA

April 10, 2022

**Introduction 3**

**Basic Functions – Pseudocode and Runtime 4**

Main Menu 4

Parsing CSV File and Validation – Pseudocode 5

Parsing CSV File and Validation – Runtime 7

**Data Structures – Pseudocode and Runtime 9**

Vector – Pseudocode 9

Vector – Runtime 11

Hash Table – Pseudocode 13

Hash Table – Runtime 16

Binary Search Tree – Pseudocode 19

Binary Search Tree – Runtime 23

**Analysis 25**

Vector 25

Hash Table 26

Binary Search Tree 27

**Recommendation 28**

**Introduction**

In this project a layout of a software system to store details about courses in the Computer Science department at the fictional university ABCU is presented. Three separate structures to store and access the course data from memory are offered, and the advantages and disadvantages are analyzed. Finally, a recommendation for the most suitable data structure for this application is presented with rationale.

The course information is provided in CSV (comma-separated value) format in a text document. The planned software system presents a console window menu to the user and provides functions for loading, searching, and printing courses from the data structure in memory.

The system layout is written completely in language agnostic pseudocode. Specifications from the SNHU project one guidelines are included in *italics*.

**Basic Functions – Pseudocode and Runtime**

**Main Menu**

*2. Create pseudocode for a menu. The menu will need to perform the following:*

*a. Load Data Structure: Load the file data into the data structure. Note that before you can print the course information or the sorted list of courses, you must load the data into the data structure.*

*b. Print Course List: This will print an alphanumerically ordered list of all the courses in the Computer Science department.*

*c. Print Course: This will print the course title and the prerequisites for any individual course.*

*d. Exit: This will exit you out of the program.*

**Function Menu:**

Print menu options

While (not a valid int entered):

Get user choice as int

If choice = 1:

LoadDataStructure()

If choice = 2:

If already loaded file to data structure:

PrintCourseList()

Else:

Print error

If choice = 3:

If already loaded file to data structure:

PrintCourse()

Else:

Print error

If choice = 4:

Exit()

Else:

Print option not available

**Parsing CSV File and Validation**

*1. Design pseudocode to define how the program*

*a. Opens the file*

*b. Reads the data from the file*

*c. Parses each line*

*d. Checks for formatting errors*

**Function OpenFile(filepath):**

New input filestream

open file at file path using filestream

if file opened successfully:

return filestream

else:

Return error

**Function ReadLine(file):**

if not at end of file:

Get next line of file up to newline

Create new stringstream with line text

Return stringstream object

**Function NextTokenatLine(line):**

New string token

while chars exist in line:

Get nextChar = next char from stringstream

if nextChar is comma:

Return token

else:

Add char to end of token string

**Function Parse(file):**

while not at end of file:

Stringstream line = ReadLine(file)

WHILE not at end of line:

Token = NextTokenatLine(line)

**Function validateFormat(file):**

New hashmap validCourses

New vector prereqs

while not at end of file:

Line = ReadLine(file)

Add NextTokenatLine(line) to validCourses

if at end of line:

RETURN error “not enough tokens”

else:

Get NextTokenatLine(line) // “course name” not used for validation

while not at end of line:

Add NextTokenatLine to prereqs

FOR each prereq in prereqs:

If prereq not in validCourses:

return error “bad prerequisite”

return no error

**Runtime Analysis**

OpenFile(filepath)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| New input filestream | 1 | 1 | 1 |
| open file at file path using filestream | 1 | 1 | 1 |
| If file opened successfully | 1 | 1 | 1 |
| Return filestream | 1 | 1 | 1 |
| **Total Cost** | | | 4 |
| **Runtime** | | | O(1) |

ReadLine(file)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If not at end of file | 1 | 1 | 1 |
| Get next line of file up to newline | 1 | 1 | 1 |
| Create new stringstream with line text | 1 | 1 | 1 |
| Return stringstream object | 1 | 1 | 1 |
| **Total Cost** | | | 4 |
| **Runtime** | | | O(1) |

NextTokenatLine(line)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| New string token | 1 | 1 | 1 |
| While chars exist in line | 1 | n | n |
| Get nextChar = next char from stringstream | 1 | n | n |
| IF nextChar is comma: | 1 | 1 | 1 |
| Return token | 1 | 1 | 1 |
| **Total Cost** | | | 3 + 2n |
| **Runtime** | | | O(n) |

Parse(file)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| WHILE not at end of file: | 1 | n | n |
| Stringstream line = ReadLine(file) | 1 | n | n |
| WHILE not at end of line: | 1 | n | n |
| Token = NextTokenatLine(line) | n | n | n |
| **Total Cost** | | | 4n |
| **Runtime** | | | O(n) |

validateFormat(file)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| New hashmap validCourses | 1 | 1 | 1 |
| New vector prereqs | 1 | 1 | 1 |
| WHILE not at end of file: | 1 | n | n |
| Line = ReadLine(file) | 1 | n | n |
| Add NextTokenatLine(line) to validCourses | n | n | n |
| IF at end of line: | 1 | 1 | 1 |
| RETURN error “not enough tokens” | 1 | 1 | 1 |
| Else: |  |  |  |
| Get NextTokenatLine(line) | n | n | n |
| WHILE not at end of line: | 1 | n | n |
| Add NextTokenatLine to prereqs | n | n | n |
| FOR each prereq in prereqs: | 1 | n | n |
| If prereq not in validCourses: | 1 | n | n |
| RETURN error “bad prereq” | 1 | 1 | 1 |
| **Total Cost** | | | 8n + 5 |
| **Runtime** | | | O(n) |

**Data Structures – Pseudocode and Runtime**

**Vector - Pseudocode**

*1b. Show how to create course objects, so that one course object holds data from a single line from the input file.*

*1c. Design pseudocode that will print out course information and prerequisites.*

*3a. Sort the course information by alphanumeric course number from lowest to highest.*

*3b. Print the sorted list to a display.*

**Struct Course:**

Course ID

Course Name

Vector Prerequisites

**Function addCourses(fileValidated):**

New vector<course> courses

For each line of file:

New course object

Course.ID = NextTokenAtLine(line)

Course.Name = NextTokenAtLine(line)

While not at end of line:

Course.Prerequisites.add(NextTokenAtLine(line))

Add course to course objects vector

**Function SearchandPrint(ID):**

For each course in courses vector:

If course ID = search ID:

Print course name

For each prerequisite in course

Print prerequisite

**Function VectorPrintAll():**

For each course in courses vector:

Print course name

Print course ID

For each prerequisite in course

Print prerequisite

**Function partition(vector, begin, end):**

low = begin

high = end

pivot = middle element of vector

While true:

While element at low < pivot:

Increment low

While element at high > pivot:

Decrement high

If low >= high:

Return high

Swap elements at index low and high

Increment low

Decrement high

**Function quicksort(vector, begin, end):**

If begin >= end:

Return

middle = partition(vector, begin, end)

quicksort(vector, begin, middle)

quicksort(vector, middle + 1, end)

**Vector – Runtime Analysis**

addCourses(fileValidated)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| New vector<course> courses | 1 | 1 | 1 |
| For each line of file: | 1 | n | n |
| New course object | 1 | n | n |
| Course.ID = NextTokenAtLine(line) | n | n | n |
| Course.Name = NextTokenAtLine(line) | n | n | n |
| While not at end of line: | n | n | n |
| Course.Prerequisites.add(NextTokenAtLine(line)) | n | n | n |
| Add course to course objects vector | 1 | n | n |
| **Total Cost** | | | 7n + 1 |
| **Runtime** | | | O(n) |

SearchandPrint(ID)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| For each course in courses vector: | 1 | n | n |
| If course ID = search ID: | 1 | n | n |
| Print course name | 1 | n | n |
| For each prerequisite in course | 1 | n | n |
| Print prerequisite | 1 | n | n |
| **Total Cost** | | | 5n |
| **Runtime** | | | O(n) |

VectorPrintAll()

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| For each course in courses vector: | 1 | n | n |
| Print course name | 1 | n | n |
| Print course ID | 1 | n | n |
| For each prerequisite in course | 1 | n | n |
| Print prerequisite | 1 | n | n |
| **Total Cost** | | | 5n |
| **Runtime** | | | O(n) |

partition(vector, begin, end)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| low = begin | 1 | 1 | 1 |
| high = end | 1 | 1 | 1 |
| pivot = middle element of vector | 1 | 1 | 1 |
| While true: | 1 | n | n |
| While element at low < pivot: | 1 | n | n |
| Increment low | 1 | n | n |
| While element at high > pivot: | 1 | n | n |
| Decrement high | 1 | n | n |
| If low >= high: | 1 | n | n |
| Return high | 1 | 1 | 1 |
| Swap elements at index low and high | 1 | n | n |
| Increment low | 1 | n | n |
| Decrement high | 1 | n | n |
| **Total Cost** | | | 9n + 4 |
| **Runtime** | | | O(n) |

quicksort(vector, begin, end) – Worst Case

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If begin >= end: | 1 | n-1 | n-1 |
| Return |  |  |  |
| middle = partition(vector, begin, end) | n | n-1 | n(n-1) |
| quicksort(vector, begin, middle) | n | n-1 | n(n-1) |
| quicksort(vector, middle + 1, end) | n | n-1 | n(n-1) |
| **Total Cost** | | | 3n(n-1) + n - 1 |
| **Runtime** | | | O(n2) |

quicksort(vector, begin, end) – Average Case

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If begin >= end: | 1 | log(n) | log(n) |
| Return |  |  |  |
| middle = partition(vector, begin, end) | n | log(n) | n log(n) |
| quicksort(vector, begin, middle) | n | log(n) | n log(n) |
| quicksort(vector, middle + 1, end) | n | log(n) | n log(n) |
| **Total Cost** | | | 3 n log(n) + log(n) |
| **Runtime** | | | O(n log n) |

**Hash Table – Pseudocode**

*1b. Show how to create course objects, so that one course object holds data from a single line from the input file.*

*1c. Design pseudocode that will print out course information and prerequisites.*

*3a. Sort the course information by alphanumeric course number from lowest to highest.*

*3b. Print the sorted list to a display.*

**Struct Course:**

Course ID

Course Name

Vector Prerequisites

**Struct Node:**

Course course

Node pointer next

**Function HashTableAddItem(Course):**

New node(course)

Get hash key for this item by course ID

Insert object at hash key address

if hash key location is occupied

While node at location.next pointer is not null:

Move to node next pointer

Insert node address at node next pointer

**Function AddCourses(fileValidated):**

For each line of file:

New course object

Course.ID = NextTokenAtLine(line)

Course.Name = NextTokenAtLine(line)

While not at end of line:

Course.Prerequisites.add(NextTokenAtLine(line))

HashTableAddItem(object)

**Function SearchandPrint(courseID):**

Hash course ID

Get object at hash address

If object is not empty:

For each linked node:

If course ID = search ID

Print course name

For each prerequisite in course at node

Print prerequisite

**Function HashTableSort():**

New vector keys

For each course in hashtable:

Add course ID to keys vector

If item has chained nodes:

For each chained node:

Add course ID at node to vector

quicksort(keys, 0, last element)

Return keys

**Function HashTablePrintAll(keys):**

For each key in keys:

SearchandPrint(key)

**Hash Table – Runtime Analysis**

HashTableAddItem(Course) – Worst Case

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| New node(course) | 1 | 1 | 1 |
| Get hash key for this item by course ID | 1 | 1 | 1 |
| Insert object at hash key address | 1 | 1 | 1 |
| if hash key location is occupied | 1 | 1 | 1 |
| While node at location.next pointer is not null: | 1 | n | n |
| Move to node next pointer | 1 | n | n |
| Insert node address at node next pointer | 1 | 1 | 1 |
| **Total Cost** | | | 2n + 5 |
| **Runtime** | | | O(n) |

HashTableAddItem(Course) – Average Case

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| New node(course) | 1 | 1 | 1 |
| Get hash key for this item by course ID | 1 | 1 | 1 |
| Insert object at hash key address | 1 | 1 | 1 |
| if hash key location is occupied | 1 | 1 | 1 |
| While node at location.next pointer is not null: | 1 | c | c |
| Move to node next pointer | 1 | c | c |
| Insert node address at node next pointer | 1 | 1 | 1 |
| **Total Cost** | | | 5 + c |
| **Runtime** | | | O(1) |

addCourses(fileValidated)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| For each line of file: | 1 | n | n |
| New course object | 1 | n | n |
| Course.ID = NextTokenAtLine(line) | n | n | n |
| Course.Name = NextTokenAtLine(line) | n | n | n |
| While not at end of line: | 1 | n | n |
| Course.Prerequisites.add(NextTokenAtLine(line)) | n | n | n |
| HashTableAddItem(object) | 1 | n | n |
| **Total Cost** | | | 7n |
| **Runtime** | | | O(n) |

SearchandPrint(courseID)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| Hash course ID | 1 | 1 | 1 |
| Get object at hash address | 1 | 1 | 1 |
| If object is not empty: | 1 | 1 | 1 |
| For each linked node: | 1 | c | c |
| If course ID = search ID | 1 | c | c |
| Print course name | 1 | c | c |
| For each prerequisite in course at node | 1 | c | c |
| Print prerequisite | 1 | c | c |
| **Total Cost** | | | 5c + 3 |
| **Runtime** | | | O(1) |

HashTableSort()

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| New vector keys | 1 | 1 | 1 |
| For each course in hashtable: | 1 | n | n |
| Add course ID to keys vector | 1 | n | n |
| If item has chained nodes: | 1 | n | n |
| For each chained node: | 1 | n | n |
| Add course ID at node to vector | 1 | n | n |
| quicksort(keys, 0, last element) | n log n | 1 | n log n |
| Return keys | 1 | 1 | 1 |
| **Total Cost** | | | n log n + 5n + 2 |
| **Runtime** | | | O(n log n) |

HashTablePrintAll(keys)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| For each key in keys: | 1 | n | n |
| SearchandPrint(key) | 1 | n | n |
| **Total Cost** | | | 2n |
| **Runtime** | | | O(n) |

**Binary Search Tree – Pseudocode**

*1b. Show how to create course objects, so that one course object holds data from a single line from the input file.*

*1c. Design pseudocode that will print out course information and prerequisites.*

*3a. Sort the course information by alphanumeric course number from lowest to highest.*

*3b. Print the sorted list to a display.*

**Struct Course:**

Course ID

Course Name

Vector Prerequisites

**Struct Node:**

Course course

Node pointer left

Node pointer right

**Function AddCourses(fileValidated):**

For each line of file:

New course object

Course.ID = NextTokenAtLine(line)

Course.Name = NextTokenAtLine(line)

While not at end of line:

Course.Prerequisites.add(NextTokenAtLine(line))

BSTAddItem(object)

**Function BSTAddItem(Course):**

New node(course)

set currNode to root node

if currNode is null:

set root to object

return

while currNode is not null:

if object ID is less than currNode ID:

if currNode left side is empty:

set currNode left to new node with object

return

else:

set currNode to currNode left

else:

if currNode right side is empty:

set currNode right to new node with object

return

else:

Set currNode to currNode right

**Function SearchandPrint(courseID):**

recursiveSearchandPrint(BST root, courseID)

**Function recursiveSearchandPrint(node, courseID):**

if node is null:

Print not found

if node object ID is courseID:

Print course ID

Print course Name

For each prereq in prerequisites:

Print prereq

else:

if courseID is less than node object ID:

recursiveSearchandPrint(node left, courseID)

else:

recursiveSearchandPrint(node right, courseID)

**Function BSTPrintAllCoursesinOrder:**

recursivePrintAll(root node)

**Function recursivePrintAll(node):**

If node is empty return

recursivePrintAll(node left address)

Print course ID

Print course Name

For each prereq in prerequisites:

Print prereq

recursivePrintAll(node right address)

**Binary Search Tree – Runtime Analysis**

BSTAddItem(Course) – Average Case

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| SET currNode to root node | 1 | 1 | 1 |
| IF currNode is null: | 1 | 1 | 1 |
| SET root to object | 1 | 1 | 1 |
| RETURN | 1 | 1 | 1 |
| WHILE currNode is not null: | 1 | log n | log n |
| IF object ID is less than currNode ID: | 1 | log n | log n |
| IF currNode left side is empty: | 1 | log n | log n |
| SET currNode left to new node with object | 1 | log n | log n |
| RETURN | 1 | log n | log n |
| ELSE: | 1 | log n | log n |
| SET currNode to currNode left | 1 | log n | log n |
| ELSE: | 1 | log n | log n |
| IF currNode right side is empty: | 1 | log n | log n |
| SET currNode right to new node with object | 1 | log n | log n |
| RETURN | 1 | log n | log n |
| ELSE: | 1 | log n | log n |
| Set currNode to currNode right | 1 | log n | log n |
| **Total Cost** | | | 13log n + 4 |
| **Runtime** | | | O(log n) |

addCourses(fileValidated)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| For each line of file: | 1 | n | n |
| New course object | 1 | n | n |
| Course.ID = NextTokenAtLine(line) | n | n | n |
| Course.Name = NextTokenAtLine(line) | n | n | n |
| While not at end of line: | n | n | n |
| Course.Prerequisites.add(NextTokenAtLine(line)) | n | n | n |
| BSTAddItem(object) | log n | n | n log n |
| **Total Cost** | | | n log n + 6n |
| **Runtime** | | | O(n log n) |

recursiveSearchandPrint(node, courseID) – Average Case

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| IF node is null: | 1 | log n | log n |
| Print not found | 1 | log n | log n |
| IF node object ID is courseID: | 1 | log n | log n |
| Print course ID | 1 | log n | log n |
| Print course Name | 1 | log n | log n |
| For each prereq in prerequisites: | c | log n | log n |
| Print prereq | c | log n | log n |
| ELSE: |  |  |  |
| IF courseID is less than node object ID: | 1 | log n | log n |
| recursiveSearchandPrint(node left, courseID) | c | log (n – 1) | log n |
| ELSE: |  |  |  |
| recursiveSearchandPrint(node right, courseID) | c | log (n – 1) | log n |
| **Total Cost** | | | 10 log n |
| **Runtime** | | | O(log n) |

recursivePrintAll(node)

| **Code** | **Line Cost** | **# Times Executes** | **Total Cost** |
| --- | --- | --- | --- |
| If node is empty return | 1 | n | n |
| recursivePrintAll(node left address) | c | (n / 2) | (n / 2) |
| Print course ID | 1 | n | n |
| Print course Name | 1 | n | n |
| For each prereq in prerequisites: | c | n | n |
| Print prereq | c | n | n |
| recursivePrintAll(node right address) | c | (n / 2) | (n / 2) |
| **Total Cost** | | | 6n |
| **Runtime** | | | O(n) |

**Analysis**

**Vector Data Structure**

Overview of functions and associated runtime analysis:

|  |  |  |
| --- | --- | --- |
| Function | Average Case Time Complexity | Worst Case Time Complexity |
| Loading course objects into memory | O(n) | O(n2) |
| Accessing a specific object | O(n) | O(n) |
| Accessing all objects in alphanumeric order | O(n log n) | O(n2) |

The vector (or array) data structure offers linear time write to and access from memory and linearithmic access to in order items. Objects are stored in memory one after the other, in the order they are read, into the next index available in the array, thus taking as much time as there are objects to complete. Searching the unsorted array for a specific element involves visiting each element until a match is found (although faster ways to search a vector exist, they are not implemented in this example). To access each element in order, the vector must first be sorted. This process has on average a logarithmic time complexity, followed by a linear print of each element.

Linear time writing and access means that the time taken is directly proportional to the size of the input. In terms of this project, a vector can therefore provide very fast access times when working with a relatively small number of courses, especially considering speed of computation on modern hardware. It is also notably the simplest implementation to code. On the other hand, the access times can get quite long for very large inputs, and frequent access to specific objects involves excessive time due to searching the entirety of the array for each request.

**Hash Table Data Structure**

Overview of functions and associated runtime analysis:

|  |  |  |
| --- | --- | --- |
| Function | Average Case Time Complexity | Worst Case Time Complexity |
| Loading course objects into memory | O(n) | O(n2) |
| Accessing a specific object | O(1) | O(n) |
| Accessing all objects in alphanumeric order | O(n log n) | O(n2) |

One of the main advantages of the hash table is constant time access to any object stored in memory. This means that we could store as many course objects as we have space for, and the time taken to access would never increase. However, it should be noted that for very small input sizes it is still possible that the actual time taken to hash and access an object could be similar or even slower than a linear array access.

Accessing each stored element in order still has a linearithmic time complexity. This is because elements in a hash table are unordered, so a sort must be performed on a list of the keys, then each sorted key accessed from the hash table in constant time. Loading course files still takes linear time, increasing proportional only to the size of the input.

For this project, a key advantage to a hash table is in being able to handle frequent access requests in constant time. If the main functionality of the system is getting data for specific courses, this could be a very suitable data structure to implement. There are limited downsides as other functions remain fairly fast. The implementation itself, however, is likely the most difficult of the data structures considered here. Not only does a proper hashing algorithm need to be selected and implemented, but data collisions must also be addressed. In this example, chaining a linked list of objects at each address is used to handle collisions.

**Binary Search Tree Data Structure**

Overview of functions and associated runtime analysis:

|  |  |  |
| --- | --- | --- |
| Function | Average Case Time Complexity | Worst Case Time Complexity |
| Loading course objects into memory | O(n log n) | O(n2) |
| Accessing a specific object | O(log n) | O(n) |
| Accessing all objects in alphanumeric order | O(n) | O(n) |

The binary search tree offers a very balanced approach in time complexity for this project. A binary search tree acts as a presorted data set, where each element is inserted in its sorted location in memory during loading by chosen ID. This slows loading time but allows faster access times as a result.

Access to a specific object occurs in logarithmic time, which is marginally slower than constant time access. Access works by ‘ruling out’ half of the values in each iteration of a search. Accessing all objects in order is simply linear as the objects are already sorted in memory. This is the fastest possible in order access.

In this project, a binary search tree can provide great overall performance. The slowest operation is loading courses, which may be called least frequently in this system example. Access to all objects or specific objects will occur very fast even at large input sizes. Also, the implementation is not overly challenging, and use of recursive functions can make loop actions on the tree easier to code. The major downside is that creating very large trees will take noticeably longer than in the previous two data structures.

**Recommendation**

Considering the primary functions of the system and the type of input expected, a binary search tree is recommended as the system’s data structure.

A tree is chosen here as it offers the best balance of runtime complexity for needed functions vs. difficulty of implementation. A vector would be simpler to implement but requires a linear search each time a single data object is requested. A hash table offers constant time access to single objects but is more difficult to implement. A tree offers ‘not much slower’ access time to single objects, as well as the fastest in order access of the three data structures considered, with the only downside of taking more time to create the tree initially.