Edward Johnson

For this project, I have a series of data structures that I will be evaluating in terms of their efficiency, and which I believe would be the most practical to use for the application we will build. This application will be focused around processing a .csv file containing course information, loading that course information into individual course objects, and then storing those course objects into the chosen data structure. The three data structures that I have chosen to review for this application are, a hash table that implements collision chaining via node linked lists, a vector that implements a sorted linked list, and a tree that is implemented as an Adelson-Velsky and Landis self-balancing tree. Each of these data structures could be / are built using a template data structure that I have set up. A run-time analysis has been made for each of the functions that compose these applications at the end of this document. These structures will be reviewed based on the use case of these applications. Those use cases will be centered around loading a document of course objects, retrieving a single course object by ID, and retrieving all course objects in sorted alphanumerical order.

The first data structure that was considered for this application, is the hash table. The hash table provides excellent search and insertion times, and for the context of this program, would be a great choice. The overall time complexity for both insertions and searches into the hash table would be O(1) which is constant time. The worst case for these actions would be O(n). This worst case time complexity is really only the case if the hashing function leads to excess collisions and / or if the distribution of objects across the keys is highly unbalanced, i.e. there are long chains. The issue with using a hash table lies in it's relative inability to sort stored data. In order to retrieve all of the course data objects in sorted order, they would first need to be extracted from the hash table, inserted into a temporary data structure such as a vector, and then have a sorting algorithm run on the contents of the vector. This is where our use of the hash table as our data structure would go from average time complexity of constant O(1), to an average of O(n log n). The sorting algorithm that I have chosen to apply for the context of this hash table, is quick sort. The algorithm chosen for this quick sort should has a worst case time complexity of $O(n^2)$. Quicksort can technically have a worst case time complexity of $O(n^2)$ but the

implementation used for this scenario implements a median of three to assist in selecting the pivot value for the algorithm. This helps by reducing the chance of ending up in a worst case scenario by increasing the chances that the pivot value chosen is closer to the median value in the total set of objects. By having a pivot value that is closer to the median, we have a better chance of having the quicksort algorithm create more balanced partitions. The worst case scenario and time complexity with quicksort occurs when the algorithm creates partitions that are extremely unbalanced, like for instance, a partition with one element, and a partition with all of the rest of the elements. A strategy such as median of 3 helps to create a better balance, and pushes the average more towards O (n log n). That being said, the worst case of this sorting process would still be $O(n^2)$. This could technically be reduced down to a guaranteed O(n log n) by using an algorithm such as heap sort or merge sort, but it is my opinion that this would not be ideal for our use case. Because of the problematic nature that retrieving the data in a sorted format from the hash table would create, it is my recommendation that we do not use a hash table for the school's course catalog.

The next data structure that was considered for this application, is the vector. The way that the data is organized into this vector, will be in the form of a sorted linked list. This sort is achieved during the process of insertion, and ensures that the data structure remains sorted in the long term. As any objects are added to the list, they are inserted in their proper position into the list. The worst case time complexity in this data structure would come from placing an object at the very end of the list, or from retrieving a single object from the very end of the list. This is because linked lists do not typically contain index variables and to access an object at the end of the list, the list must first be traversed in full. This is fine for small datasets, and would work fine for the application of our course catalog, but with large datasets, this is a worst case time complexity of linear time or O(n). This means that the time it takes to access an item in the worst case increases linearly with the size of the dataset / elements. This linear time is the case for insertions, reads, and deletions from the list. This would be a solid alternative choice for data structures to use for this software application, but it would not be my first choice.

The last data structure that we have to review, and the selection for best candidate for our use case, is the AVL tree. This tree is a self-balancing binary search tree, and provides a guaranteed worst case time complexity of O(log n). This is the next best time complexity, after constant time or O(1). This data structure is more complicated than many simpler alternatives, but it provides long term stability and very efficient operations when implemented properly. This data structure is implemented similarly to a binary search tree, but with the addition of a height value to each tree node, a balance function, right and left rotation functions, and some additional balancing logic added to the insertion method. This additional logic includes calculating and updating the height value that is stored in each node along with the data, and triggering any rotations that are needed in order for the tree to be re-balanced when adding or removing nodes. What balance means in the context of this tree, is that the sub-trees of each node remain proportional on both sides. The balance occurs in that there is at most, a height difference of 1 in the height of the two sub trees within a given node. Whether this be from the root node, or a node further down one of the left or right sub-trees. This is known as the balance factor of the AVL tree. This balance factor can have one of three values, -1, 0, or 1. A balance factor of 0 represents a perfect balance of both left and right sub trees, and -1 or 1 indicates that the left or right side is heavier. The tree checks after each insertion whether the tree has become unbalanced, and if so, will begin performing rotations until the nodes again have reached a balance. There are four types of rotational actions that can occur in an AVL tree. These actions occur once the balance factor of a node reaches 2, or -2. If for example, a node is inserted into the left sub-tree, and the balance factor is calculated as being 2 of the node, then a right rotation would be needed as the left sub-tree is now unbalanced, this right rotation is also known as a right right rotation. If a right sided insertion caused a right side imbalance, then a left sided rotation would occur, or a left left rotation. The other two rotations that can occur are double rotations also known as left right, and right left rotations. These rotations occur when a child node and a sub-tree of that node become unbalanced due to an addition or removal from that sub-tree. How this would look visually, is like a zig-zag, where the unbalanced subtree branches down to the

left/right, and then the opposite direction for the next node. A combination of a right then a left, or a left then a right rotation is executed to return the tree to a balanced state. Say I have parent node C with a balance value of 2, left child of C, node A with a balance of 1, and right child of node A, node B with a balance of 0. I can return these nodes to a state of 0 by rotating node A down left to the position of the left child of node B. This is the first of two rotations in the double rotation, and leaves us with a balance of node B with 1 and node A now with 0. I then execute a right rotation on node B which rotates the parent node C to now be in the position of right child node of node B. With the double rotation executed, node A is now the left child of node B, and node C is now the right child of node B, each of these nodes now have the balance factor of 0. The balance is returned, and the nodes have still remained in ascending order of value. Previously, C was greater than and above A, and B was greater than, and below to the right of A. Now, A is to the left of (lower value than) B, and C is to the right of (higher value than) B.

The methods responsible for actions such as returning the balance, calculating / assigning the height value to a node, and rotating the nodes left and right all run in constant time, or O(1) . While these methods calls in themselves operate in constant time, when they are applied to the tree over the course of an insertion process, they happen in O(log n) time. This is because the process of calculating height and rebalancing the tree can occur as many times as O(log n) or, proportional to the height of the balanced AVL tree. Similar logic applies to why the search method of traversal of the tree takes at worst case, O(log n) time to complete. In the worst case, the search algorithm would have to traverse the tree a number of times that is proportional to the height of the tree, which is again, O(log n). The height of the AVL tree cannot exceed log(N), where N is the total number of nodes in the tree.

Of the three data structures covered in this analysis, the AVL tree offers the best performance, and will be the easiest to maintain in the long-term due to it’s ability to self-balance / regulate. It will offer the same time complexity for all operations that it is used to process, and that time complexity is at worst O(log n).

Additionally, the fact that AVL trees maintain an order to the data kept within the nodes of the tree are the primary reason that I would recommend this structure over the others. As we will regularly need to access the data of courses stored within the tree, and we will need to output these courses in a sorted format, it would be more beneficial to use a system that utilizes sorted storing. On top of this, to use a structure such as the hash table and then to extract and sort the data within, you are still looking at a worst case run time of O(log n) with heap or merge sort. So we are getting the same worst case sort times while using a system and structure that in my opinion, is less burdensome to maintain / implement. This is why I have selected this type of binary search tree as the logical structure around to base our catalog system. The reason why each of these algorithms will be at the least O(n) to print, is because printing is an operation that increases linearly with the amount of objects in the structure. Regardless of how fast you can reach individual data nodes in the structure, to output all nodes you must travels to each node.

Overall run-time analysis per data structure (worst case only):

Data Structure	**Insertion**	**Search / Retrieval**	**Sorted Print All**	**Total / Overall complexity**
Hash Table (w/ vector and sort alg. for sorted output)	O(n)	O(n)	$O(n^2)$ for Quick or O(n log n) merge	O(n log n) heap/ merge or $O(n^2)$ if utilizing quicksort for output
Vector - Ordered Linked List	O(n)	O(n)	O(n)	O(n)
AVL (binary search) Tree	O(log n)	O(log n)	O(n)	O(n)

GENERAL INPUT FUNCTIONS: – Total complexity worst case : O(n)

1. objectHandler()

Code	Line Cost	# Times Executed	Total Cost
DEFINE Course “course” objectHandler() method	1	1	1
DEFINE string vector "prerequisites"	1	1	1
For integer variable 'i' equals 2, while 'i' is less than the size of vector "courseData"	1	n	n
ASSIGN the string value at index 'i' to corresponding "prerequisites" location using push_back AND courseData.at('i')	1	n	n
IF "prerequisites" is empty, AND size of "courseData" is equal to 2,	1	1	1
ADD / push_back the string value "N/A" to "prerequisites" vector	1	1	1
DEFINE new struct object Course “course”	1	1	1
ASSIGN “course” string variable "courseId"	1	1	1
ASSIGN “course” string variable "courseTitle"	1	1	1
ASSIGN “course” string vector variable "prerequisiteCourses"	1	1	1
RETURN finished Course object "course"	1	1	1
		Total Cost:	2n + 9
		Runtime worst-case:	O(n)

2. courseInputHandler()

Code	Line Cost	# Times Executed	Total Cost
DEFINE void function courseInputHandler() ,	1	1	1
DEFINE string variable "headerLineInput"	1	1	1
IF (getline ("fileInputStream", "headerLineInput") fails	1	1	1
DEFINE string variable “nextInputLine”	1	1	1
WHILE getline ("fileInputStream", "nextInputLine") is successful)	1	n	n
DEFINE string vector "courseStrings"	1	1	1
ASSIGN "courseStrings" the value from function CALL to vectorParser()	2n + 5	1	2n + 5
IF the size of "courseStrings" is greater than OR equal to 2	1	1	1
DEFINE Course struct “nextCourseToInsert”	1	1	1
ASSIGN “nextCourseToInsert” the value of CALL to objectHandler()	2n+9	1	2n+9
CALL “courseContainer”→Insert() method with object “	? / 1 (going with 1 but depends on structure used)	1	n
		Total Cost:	6n+22
		Runtime worst-case:	O(n)

3. vectorParser()

Code	Line Cost	# Times Executed	Total Cost
DEFINE vector<string> vectorParser() method,	1	1	1
DEFINE stringstream ss(nextInputLine) using the function	1	1	1
DEFINE string vector "courseData"	1	1	1
DEFINE local string variable "tempCourseValue"	1	1	1
WHILE getline has more input to process / getline	1	n	n
ADD the next parsed "tempCourseValue" value to the array courseData	1	n	n
RETURN the complete string vector containing each value	1	1	1
		Total Cost:	2n+5
		Runtime worst-case:	O(n)

4. openFile()

Code	Line Cost	# Times Executed	Total Cost
DEFINE void openFile() method params string “filePath”	1	1	1
DEFINE ifstream “fileInputStream”	1	1	1
DEFINE boolean variable "isInputOpen",	1	1	1
WHILE "isInputOpen" is false	1	n	n
OPEN "fileInputStream" with the file	1	1	1
IF (fileInputStream.is_open() is false / or the ELSE condition	1	1	1
ASSIGN "isInputOpen" the value true	1	1	1
IF (peek character of fileinputstream	1	1	1
CALL the function courseInputHandler()	5n+23	1	5n+23
CLOSE the ifstream “file	1	1	1
RETURN	1	1	1
		Total Cost:	6n+32
		Runtime worst-case:	O(n)

AVL TREE FUNCTIONS – Total complexity worst case :

1. Insert()

Code	Line Cost	# Times Executed	Total Cost
DEFINE <class T> override void method Insert()	1	1	1
CALL insert() method with args “root” and “tData	1	1	1
		Total Cost:	2
		Runtime worst-case:	O(1)

2. Search()

Code	Line Cost	# Times Executed	Total Cost
DEFINE <class T> override T* ptr method Search()	1	1	1
DEFINE Node<T>* pointer “node” ASSIGN the value of CALL to search()		log n	log n
IF “node” does not equal nullptr	1	1	1
RETURN reference to the &“tData” object contained in the node	1	1	1
		Total Cost:	log n+3
		Runtime worst-case:	O(log n)

3. height()

Code	Line Cost	# Times Executed	Total Cost
DEFINE <class T> method int height()	1	1	1
IF “currentNode” equals nullptr	1	1	1
RETURN 0	1	1	1
		Total Cost:	3
		Runtime worst-case:	O(1)

4. returnBalance()

Code	Line Cost	# Times Executed	Total Cost
DEFINE <class T> method int returnBalance()	1	1	1
ELSE	1	1	1
RETURN the value of CALL to (height with arg (currentNode→left)) minus (height with arg (currentNode→right)) so height() * 2	7	1	7
		Total Cost:	9
		Runtime worst-case:	O(1)

5. rightRotate()

Code	Line Cost	# Times Executed	Total Cost
DEFINE <class T> method Node<T>* ptr rightRotate()	1	1	1
DEFINE Node<T>* ptr “childNode”, ASSIGN value as the “left” node	1	1	1
DEFINE Node<T>* ptr “tempSubTree” ASSIGN value as the “right” node	1	1	1
ASSIGN the “right” node of “childNode” the value of param “parentNode”	1	1	1
ASSIGN the “left” node of parentNode the value of “tempSubTree”	1	1	1
ASSIGN the value of parentNode’s height as CALL to max method with args ((height method called on left child node of “parentNode”), and (height method called on right child node of “parentNode”)) 2X max() & 2X height()	7	1	7
ASSIGN the value of childNode’s height as CALL to max method with args ((height method called on left child node of “childNode”), and (height method called on right child node of “childNode”))	7	1	7
RETURN new root of subtree, “childNode”	1	1	1
		Total Cost:	20
		Runtime worst-case:	O(1)

6. leftRotate()

Code	Line Cost	# Times Executed	Total Cost
DEFINE <class T> method Node<T>* ptr leftRotate()	1	1	1
DEFINE Node<T>* ptr “parentNode”, ASSIGN value as the	1	1	1
DEFINE Node<T>* ptr “tempSubTree” ASSIGN value as	1	1	1
ASSIGN the “right” node of “parentNode” the value of param “childNode”	1	1	1
ASSIGN the “left” node of “childNode” the value	1	1	1
ASSIGN the value of childNode’s height as CALL to max method with args ((height method called on left child node of “childNode”), and (height method called on right child node of “childNode”))	7	1	7
ASSIGN the value of childNode’s height as CALL to max method with args ((height method called on left child node of “childNode”), and (height method called on right child node of “childNode”))	7	1	7
RETURN new root , “parentNode”	1	1	1
		Total Cost:	20
		Runtime worst-case:	O(1)

7. insert() - potentially must rebalance / call height / max X height of the tree or (log n) times

Code	Line Cost	# Times Executed	Total Cost
DEFINE <class T> method Node<T>* ptr insert()	1	1	1
IF “nodeData” is less than the “nodeData” of currentNode	1	log n	log n
ASSIGN the left child of “currentNode” the value of CALL to insert()	1	log n	log n
ASSIGN the height var of current node the value of CALL to max method with args ((height method called on left child node of “currentNode”), and (height method called on right child node of “currentNode”)	7 (O(1))	log n	log n +7
DEFINE int var “balance” ASSIGN the value of CALL to returnBalance()	9 (O(1))	log n	log n +9
IF balance greater than 1 AND “nodeData” less than currentNode’s left child’s nodeData	1	1	1
RETURN the value of CALL to rightRotate() with arg “currentNode” (perform the rotations only as needed)	1	1	1
RETURN “currentNode”	1	1	1
		Total Cost:	4(log n)+4 (or +20?)
		Runtime worst-case:	O(log n)

8. search()

Code	Line Cost	# Times Executed	Total Cost
DEFINE <class T> method Node<T>* ptr search()	1	1	1
IF “tKey” is less than the “tKey” value (or courseId) of currentNode’s “nodeData” (	1	1	1
RETURN value of the CALL to Search() with args (left child node of “currentNode” and “tKey”		log n	log n
		Total Cost:	log n+2
		Runtime worst-case:	O(log n)

9. inOrder() - traverses all nodes

Code	Line Cost	# Times Executed	Total Cost
DEFINE <class T> method void inOrder()	1	1	1
IF “currentNode” does not equal nullptr	1	n	n
CALL inOrder() with the left child node of “currentNode”		n	n
CALL printANode() with argument “currentNode”	1n+7	n	1n+7
CALL inOrder() with the right child node of “currentNode”		n	n
		Total Cost:	4n 8
		Runtime worst-case:	O(n)

10. removeTree() - removes all nodes

Code	Line Cost	# Times Executed	Total Cost
DEFINE <class T> method void removeTree()	1	1	1
CALL purgeTree() with args: Node<T>* pointer “root”		n	n
ASSIGN “root” the value nullptr	1	1	1
		Total Cost:	n+2
		Runtime worst-case:	O(n)

11. printInOrder()

Code	Line Cost	# Times Executed	Total Cost
DEFINE <class T> method void printInOrder()	1	1	1
ELSE:	1	1	1
CALL inOrder() with the args: Node<T>* pointer “root”		n	n
		Total Cost:	n+2
		Runtime worst-case:	O(n)

12. isTreeEmpty()

Code	Line Cost	# Times Executed	Total Cost
DEFINE <class T> method bool isTreeEmpty()	1	1	1
RETURN root == nullptr	1	1	1
		Total Cost:	2
		Runtime worst-case:	O(1)

VECTOR FUNCTIONS (Ordered Linked List) – Total complexity worst case : O(n)

1. printInOrder()

Code	Line Cost	# Times Executed	Total Cost
DEFINE void template <class T> function printInOrder()	1	1	1
DEFINE Node<T>* ptr “currentNode”, ASSIGN it the value of “head” Node<T>*	1	1	1
DEFINE Node<T>* ptr “currentNode”, ASSIGN it the value of “head” Node<T>*	1	1	1
CALL printACourse() method with the tData object of “currentNode”	1n+7	1	1n+7
ASSIGN “currentNode” the value of the “next” Node<T>* after “currentNode”	1	1	1
		Total Cost:	n+11
		Runtime worst-case:	O(n)

2. Insert(),

Code	Line Cost	# Times Executed	Total Cost
DEFINE method that overrides template <class T> Insert(),	1	1	1
DEFINE Node<T>* ptr “newNode”, ASSIGN value of new call	1	1	1
DEFINE Node<T>* ptr “currentNode” ASSIGN the value of “head” Node	1	1	1
DEFINE Note<T>* ptr “prevNode” ASSIGN the value nullptr	1	1	1
WHILE “currentNode” not equal to nullptr AND ‘tData” of “currentNode”is less than param “tData”	1	n	n
ASSIGN “prevNode” the value “currentNode”	1	1	1
ASSIGN “currentNode” the value of the “next” Node after “currentNode”	1	1	1
		Total Cost:	n+6
		Runtime worst-case:	O(n)

3. Search(),

Code	Line Cost	# Times Executed	Total Cost
DEFINE method that overrides template <class T> Search(),	1	1	1
DEFINE Node<T>* ptr “currentNode”, ASSIGN it the value of Node<T>* “head”	1	1	1
WHILE “currentNode” is not equal to nullptr	1	n	n
IF value of “tKey” (courseId) in tData object of “currentNode” matches the “tKey”param value	1	1	1
ASSIGN “currentNode” the value of the “next” Node after “currentNode”	1	1	1
RETURN nullptr (no match) once currentNode→next equals nullptr	1	1	1
		Total Cost:	n+5
		Runtime worst-case:	O(n)

QUICKSORT – Total complexity worst case : $O(n^2)$

1. medOfThree(),

Code	Line Cost	# Times Executed	Total Cost
DEFINE template function<class T> size_t medOfThree(),	1	1	1
IF “tDataList”[firstIndex] is less than or equal to“tDataList” [middleIndex]	1	1	1
IF “tDataList”[1astIndex] is less than or equal to tDataList[firstIndex]	1	1	1
		Total Cost:	3
		Runtime worst-case:	O(1)

2. selectPivIndex(),

Code	Line Cost	# Times Executed	Total Cost
DEFINE size_t template <class T> function selectPivIndex(),	1	1	1
DEFINE size_t variable “mid” for the middle index	1	1	1
ASSIGN “mid” the value of (“low” + (“high” - “low”) / 2)	1	1	1
RETURN the value of CALL to medOfThree() method with args: “ tDataList”, “low”, “mid” and “high”	3	1	3
		Total Cost:	6
		Runtime worst-case:	O(1)

3. quickSort(),

Code	Line Cost	# Times Executed	Total Cost
DEFINE size_t template <class T> function quickSort(),	1	1	1
IF “low” is less than “high”	1	n	n
DEFINE size_t index “partIndex”,	1	1	1
ASSIGN “partIndex” the value returned from quickParttition() with args: “tDataList”, “low”, “high”	5n+18	n	5n+18
IF “partIndex” is greater than 0 AND “partIndex” greater than “low	1	1	1
CALL this func recursively with args: (“tDataList”, “low”, “pivotIndex”- 1)		n	n^2
IF “pivotIndex” less than high	1	1	1
CALL this func recursively with args: (“tDataList”, “partIndex” + 1, “high”		n	n^2
		Total Cost:	$2(n^2)+6n+22$
		Runtime worst-case:	$O(n^2)$

4. quickParttition(),

Code	Line Cost	# Times Executed	Total Cost
DEFINE size_t template <class T> function quickParttition(),	1	1	1
DEFINE size_t pivot index “pivot” , ASSIGN value of selectPivIndex() call args: “ tDataList”,	7	1	7
DEFINE type <T> variable “pivotValue” ASSIGN it the value of the tdata object at tDataList[pivot]	1	1	1
DEFINE size_t pointer position var, “ptrLeft”, ASSIGN it the value of “low”	1	1	1
DEFINE size_t pointer position var, “ptrRight”, ASSIGN it the value of “high” - 1	1	1	1
WHILE the “ptrLeft” is less than “ptrRight”	1	n	n
WHILE tDataList[ptrLeft] is less than “pivotValue” AND “ptrLeft” less than “ptrRight”	1	n	n
INCREMENT ptrLeft by +1	1	1	1
WHILE tDataList[ptrRight] greater than “pivotValue” AND “ptrRight” greater than “ptrLeft”	1	n	n
CALL SWAP on the object at tDataList[ptrLeft], with object at tDataList[ptrRight]	1	n	n
IF ptrLeft equals the “pivot” index	1	1	1
ASSIGN “pivot” the value of “ptrRight”	1	1	1
INCREMENT ptrLeft by +1	1	1,or n in loop?	1
DECREMENT ptrRight by -1	1	1,or n in loop?	1
IF “ptrLeft” is less than “high” AND tDataList[ptrLeft] is greater than “pivotValue”	1	1	1
CALL SWAP on the object at tDataList[ptrLeft], with object at tDataList[high]	1	n	n
RETURN “ptrLeft”	1	1	1
		Total Cost:	5n+18
		Runtime worst-case:	O(n)

HASH TABLE FUNCTIONS: – Total complexity worst case : O(n) alone, but with sorting becomes $O(n^2)$

1. hash()

Code	Line Cost	# Times Executed	Total Cost
CLASS METHOD DEFINITION: unsigned integer method <class T> hash()	1	1	1
RETURN the calculated modulus value of (“key” % “tableSize”)	1	1	1
		Total Cost:	2
		Runtime worst-case:	O(1)

2. remove()

Code	Line Cost	# Times Executed	Total Cost
CLASS METHOD DEFINITION: the string method <class T> remove()	1	1	1
CALL method remove() a node / object from the HashMap	n	n	n
RETURN the T& string “tKey” as confirmation upon deletion	1	1	1
		Total Cost:	n+1
		Runtime worst-case:	O(n)

3. Search()

Code	Line Cost	# Times Executed	Total Cost
CLASS METHOD DEFINITION: the T* type method <class T> Search()	1	1	1
IMPLEMENT the method to search through the hashmap using a T& string “tKey”	n	n	n
IF: no match found,	1	1	1
RETURN empty “tData” object	1	1	1
		Total Cost:	n+3
		Runtime worst-case:	O(n)

4. createNewKey()

Code	Line Cost	# Times Executed	Total Cost
CLASS METHOD DEFINITION: createNewKey()	1	1	1
RETURN the value of CALL to hash method with argument (atoi(courseId.c_str))	2	1	2
		Total Cost:	3
		Runtime worst-case:	O(1)

5. insert()

Code	Line Cost	# Times Executed	Total Cost
CLASS METHOD DEFINITION: void method insert()	1	1	1
DEFINE unsigned int variable “hKey” ASSIGN value from calling createNewKey() with argument string tData.tKey	1	1	3
DEFINE Node* pointer “insertNode” ASSIGN value from calling createNodePointer() with arg “hKey”	1	1	4
ELSE	1	1	1
ELSE	1	1	1
WHILE the value of this “insertNode” “next” is not nullpointer	1	n	n
UPDATE this “insertNode” T& object “tData”, ASSIGN the value of “tData”	1	1	1
ASSIGN the pointer of this “insertNode” the value “insertNode” ->“next”	1	1	1
WHEN loop ends and last position reached, ASSIGN this “insertNode”->	1	1	1
		Total Cost:	n+13
		Runtime worst-case:	O(n)

6. createNodePointer()

Code	Line Cost	# Times Executed	Total Cost
CLASS METHOD DEFINITION: createNodePointer() with parameter unsigned integer “hKey”	1	1	1
IF the key is less than the value of calling the size() method on tableList array	1	1	1
DEFINE Node<T>* pointer “newNodePointer” the value of t	1	1	1
RETURN the newly created Node<T>* pointer	1	1	1
		Total Cost:	4
		Runtime worst-case:	O(1)

7. sortTableData()

Code	Line Cost	# Times Executed	Total Cost
CLASS METHOD DEFINITION: <class T> vector <tData> method sortTableData()	1	1	1
DEFINE type T vector<T>& of tdata “tDataList”	1	1	1
DEFINE Node<T>* ptr “currentNode”	1	1	1
FOR EACH “hKey” in HashTable<T>	1	n	n
WHILE “currentNode” is not nullptr	1	n	n
CALL tDataList.push_back method on the currentNode’s “tData” object	1	n	n
ASSIGN currentNode ptr the value of “next” Node<T>* after “currentNode”	1	n	n
CALL single argument quickSort() with the finished reference to type T vector<T>& of tdata “tDataList”	1	1	n^2
CALL printInOrder() reference to the sorted type T vector<T>& of tdata “tDataList”	1	1	n
		Total Cost:	$n^2 + 5n + 3$
		Runtime worst-case:	$O(n^2)$

Sources:

- Erkiö, H. (1984). The worst case permutation for Median-of-Three Quicksort. *The Computer Journal*, *27*(3), 276–277. https://doi.org/10.1093/comjnl/27.3.276
- GfG. (2023, November 2). *AVL Tree Data Structure*. GeeksforGeeks. https://www.geeksforgeeks.org/introduction-to-avl-tree/
- Subedi, B. (n.d.-b). *Calculating the running time of Algorithms*. Algorithm Tutor. https://algorithmtutor.com/Analysis-of-Algorithm/Calculating-Running-time-of-Algorithms/
- *AVL Trees explained*. (n.d.). https://oramasearch.com/blog/avl-trees-explained
- Tóth, Š. (2023, March 29). The curious case of the always O(n*logn) QuickSort - ITNEXT. *Medium*. https://itnext.io/the-curious-case-of-the-always-o-n-logn-quicksort-603b56230e6
- *AVL Trees*. (n.d.). https://pages.cs.wisc.edu/~deppeler/cs400/readings/AVL-Trees/index.html