CS 163: Design Write-Up

By: Enis Inan

Date: June 10th, 2015

I. Main Program

The main program is organized as follows. First the user is given three options, two to select the data structure, either a 2-3 tree or a Hash Table, to use in implementing their own personal, text-based media library, and an extra option to exit the program. Once the user makes a valid choice, they are taken to another menu that indicates whether they'd like to open an existing media library (1), or create a new one from scratch (2). If they select (1), they are then prompted to enter a valid file name whose contents are used to recreate their media library while selecting (2) takes them directly to the library menu. Regardless, the user is transferred to the main library menu. Here, they are given five more options. (1) To add a new entry to their media library; (2) To retrieve an existing entry; (3) To remove an existing entry; (4) To display the contents of the library; and (99) to return to the main program menu to either exit the program or select another data structure to implement their library. Depending on the user choice, the relevant function is called to execute each of these options. Refer to the comments in the code "mainProgram.cpp" for more details on what these functions do.

If the user decides to exit the program, they are asked to type in the name of the file they wish to store their new media library in. The program stores the information in the file, and also some relevant statistics that are data structure dependent for them to view. For the 2-3 tree, these are: the tree height, the number of 2-Nodes, the number of 3-Nodes, and the total number of items in the library. For the hash table, these become: the table size, the number of collisions in the table, the maximum collision size of the table, defined as the number of nodes after the first node in the linked chain, the number of occupied table entries, and the total number of items in the library.

The program is divided into three main classes, each containing their own sub classes. These are: a class for the 2-3 Tree and Hash table each, and a class for the media library itself. Refer to Sections II, III and IV, respectively, for more details. Note: An exception class "NotFoundException" was used to handle exceptions resulting from attempting to access entries from an empty tree or table. Refer to the header and implementation files "NotFoundException.h" and "NotFoundException.cpp" for more details. Refer to Section V for a summary of the program statistics.

II. 2-3 Tree Design

The 2-3 tree implementation uses three classes: an abstract data type (ADT) called "Balanced Search Tree Interface" that contains the relevant data structure functions; a class "TriNode" for the nodes"; and, finally, a link-node based stack implementation called "Stack" to aid in traversing the tree during insertion and removal. Section II is divided into four sub-sections. IIa will discuss "Balanced Search Tree Interface"; IIb will present the design for "TriNode"; IIc for "Stack"; and, finally, IId will go over the implementation of the 2-3 tree class itself in more detail.

IIa. Balanced Search Tree Interface

This is a generic ADT template for binary search trees. The core functions that each implementation should have are: (1) to check if the tree is empty; (2) to get the height of the tree; (3) to get the number of items in the tree; (4) to add an entry into the tree; (5) to remove an entry from the tree; (6) to restore the tree back into an empty state; (7) to retrieve an existing entry; (8) to see if an entry is contained in the tree; and (9), to conduct an inorder traversal of the tree that executes a client defined function, visit, on each of the items in the nodes of the tree. Functions 1 – 9 are described in more detail in the header file "BalancedSearchTreeInterface.h"

IIb. TriNode

The class TriNode is designed as follows. It has five private data members smallItem, largeItem, leftChildPtr, midChildPtr, and rightChildPtr. The members smallItem and largeItem are pointers of a template type "ItemType;" pointers were used here to avoid using a dummy value to differentiate an empty node, 2-node, and 3-node from one another since these values are type dependent. Using this implementation, an empty node is defined as a TriNode member whose smallItem and largeItem pointers are the NULL pointer; a 2-Node is one that has its largeItem set as NULL; and a 3-Node has both smallItem and largeItem not equal to NULL. These are used in the definition of the corresponding member functions. Note: The pointer-based design does not violate encapsulation for the present project because TriNode is implemented privately within the 2-3 tree class and only in the 2-3 tree class, and hence the client cannot access the individual item pointers directly. Finally, leftChildPtr, midChildPtr, and rightChildPtr store pointers to the left, middle, and right child of the node respectively.

The class provides both accessor and mutator functions for each of these private data members. Refer to the comments in the header and implementation files "TriNode.h" and "TriNode.cpp" for more details on these functions and their definitions.

IIc. Stack

The class “Stack” has one private data member, a pointer, topPtr, to the top of the stack. Each item in the stack is comprised of linked nodes of type “SNode”, defined as a struct having member variables item, which stores the node’s item, and next, a pointer of SNode type that stores the pointer to the next node. Inside the “Stack” class are four functions: empty, push, pop and top. “empty” checks if the stack is empty, which occurs then topPtr points to NULL. “Push” places a new node at the front (“rear”) of the stack while “pop” removes the item at the top of the stack. Finally, “top” retrieves the item on the top of the stack. Refer to the header and implementation files, “Stack.h” and “Stack.cpp”, for more details on these functions and their definitions.

IId. Implementation of the 2-3 Tree

The 2-3 Tree class itself has one private data variable, a pointer of TriNode type called “rootPtr” that points to the root of the tree. Along with rootPtr, several private member functions were written to facilitate the implementation, and these complement the ADT functions specified in Section IIa. Two new, public functions were added to the class for the purposes of the present project only. These are writeToFile and displayStatistics. “writeToFile” outputs the contents of the tree to an external file while “displayStatistics” displays the statistics outlined in Section I of the tree. Table 1 below indicates the private member functions and which of the main, public functions they are connected to.

Table 1: Outline of the connections between the public methods and the private methods used to facilitate their implementation.

Public Method	Corresponding Private Methods
getHeight	-getHeightHelper
getNumberOfItems	-getNumberOfItemsHelper
add	-findInsertLoc -insertInLoc -getMiddleItem -split -reconnect
remove	-mergeTwoParent -mergeThreeParent -redistributeTwoParent -redistributeThreeParent -getSiblingPtr -removeValue -removeTwoNode

clear	postorderDelete
getEntry	-findItem
contains	
traverse	-inorderHelper
writeToFile	-inorderFileWrite
displayStatistics	-getNumNodes
copy constructor	-copyTree

The implementation details of the functions for all of the public and private methods in Table 1 save for those in the same row as add and remove are straightforward and described in detail in the comments of the header and implementation files "TwoThreeTree.h" and "TwoThreeTree.cpp." Of interest are tree insertion and removal, described in Sections IId – a and IId – b, respectively, below.

IId - a. Insertion into the 2-3 Tree

There are two special cases to consider when inserting into the 2-3 tree. (1) is the case of the empty tree, wherein a single node is created and becomes the root of the tree. (2) is the case of a nonempty tree and is more involved. For (2), a new memory location is created that stores the item to be inserted, and the rootPtr and the corresponding itemPtr are passed into the function "findInsertLoc." "findInsertLoc" traverses the tree and stores each of the node pointers it encounters in a stack until a leaf is found. Once a leaf is found, the pointer to the leaf-node, the pointer to the new item, and the stack are passed into the function "insertInLoc." "insertInLoc" inserts the new item into the tree. It has two cases – (1) insertion into a 2-node leaf, and (2) insertion into a 3-node leaf. The case of (1) is straightforward – the item is inserted directly into the 2-node leaf, which then becomes a 3-node leaf. (2) requires the 3-node leaf to be split into two nodes n1 and n2, where n1 houses the smaller item and n2 houses the larger item of three items: the small item in the 3-node, the passed item (stored in itemPtr), and the large item in the 3-node. The middle item gets passed up and becomes the new value of itemPtr. "getMiddleItem" rearranges the 3-node leaf prior to splitting and returns the new, modified itemPtr that stores the passed, middle item.

The 3-node leaf is now ready to be split. This is done via the function "split", where two new nodes, "n1" and "n2", are created that store the small and large item in the 3-node, respectively, while these same items are removed from the 3-node tree. n1 and n2 are then connected as the left and right child, respectively, of the now-empty 3-node to facilitate reconnecting the tree. A pointer to the empty node is returned as the connecting pointer so that,

as the passed item traverses up the tree, n1 and n2 are reconnected to it. Figure 1 shows a representative example of splitting a 3-node leaf:

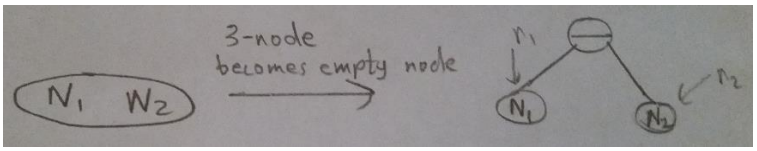


Figure 1: Splitting a 3-node leaf. Note that the original 3-node becomes the new empty node, and this pointer is returned to as the connecting pointer so that any other splits along the way can reconnect to n1 and n2.

Once the 3-node leaf is split, the top pointer stored in the stack is retrieved. There are two cases to consider. (1) We have a 2-node parent and (2), we have a 3-node parent. (1) means we are done rebuilding the tree – we convert the 2-node parent into a 3-node whose small or large item, depending on which is larger, takes the value of the passed item, and then reconnect the nodes n1 and n2 in Fig. 1 back to the tree. The latter is accomplished by the function "reconnect," described in more detail in the ensuing parts. For (2), we have another 3-node we need to split so, similar to the case of the 3-node leaf, the 3-node is reconfigured using the function "getMiddleItem," the new middle item is stored in itemPtr, and the 3-node is split again. However unlike the case of the leaf, we now have n1 and n2 from a previous split to reconnect – this is accomplished in the split function, which calls the function "reconnect" depending on if the configuration of the empty node satisfies a few select cases or reconnects the nodes itself for a special case. This, too, will be described below.

The function "reconnect" connects two nodes n1 and n2 from a previous split back to a 2-node or 3-node parent. The 2-node parent represents either an n1 or n2 from a current split while the 3-node parent is a 2-node parent converted to a 3-node parent (i.e. a node that satisfies case (1) from the previous paragraph). Figure 2 shows the four possible cases of "reconnect."

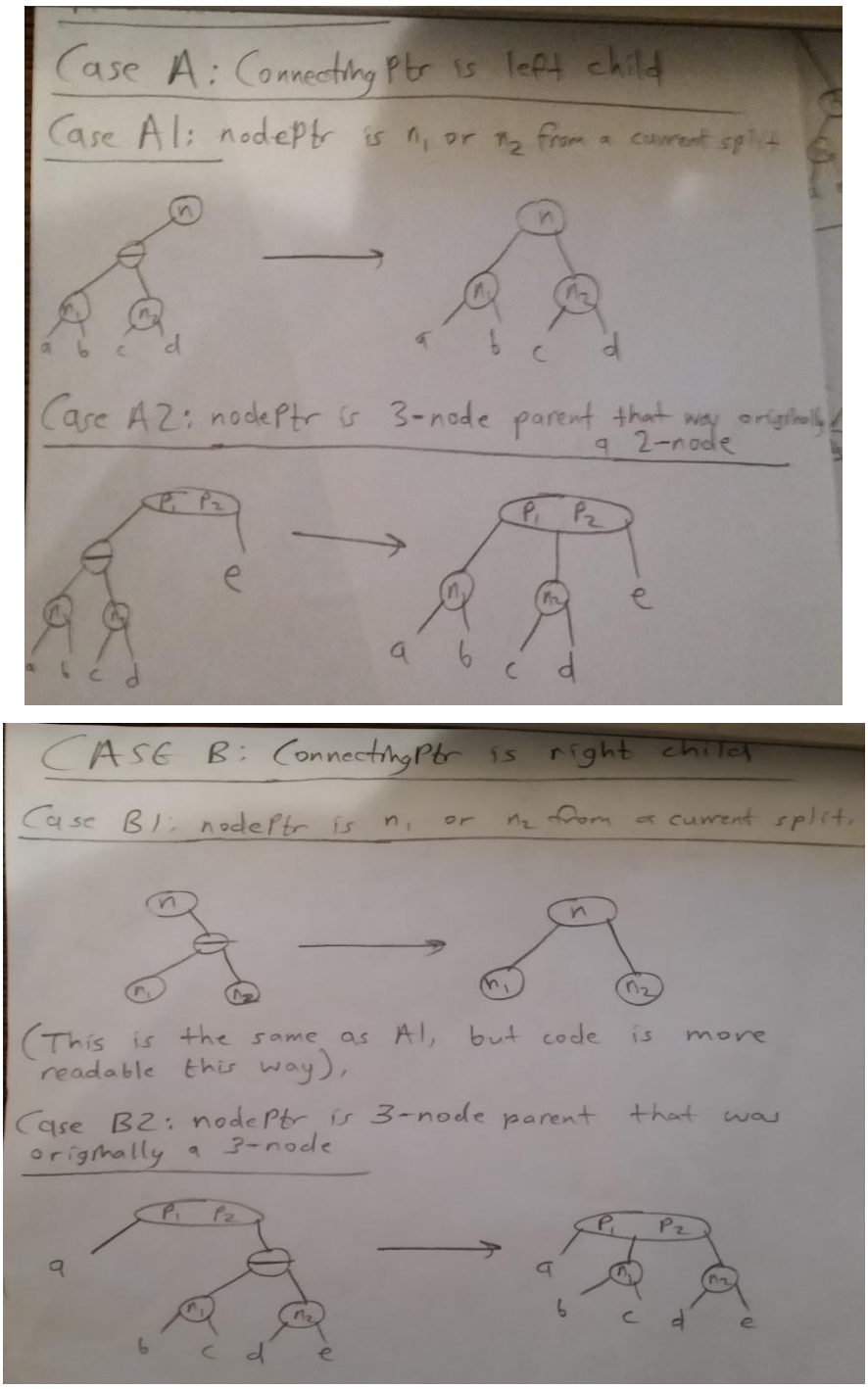


Figure 2: Reconnecting n1 and n2 from a previous split back to either n1 or n2 from a current split, or to the new 3-node parent that was originally a 2-node. From top to bottom – a) Case A, when the connecting pointer is the left child; b) Case B, when the connecting pointer is the right child.

Splitting an internal 3-node is slightly different than splitting a leaf since we also have to reconnect two nodes from an earlier split to either just n1, just n2, or both. These split cases are shown below in Fig. 3.

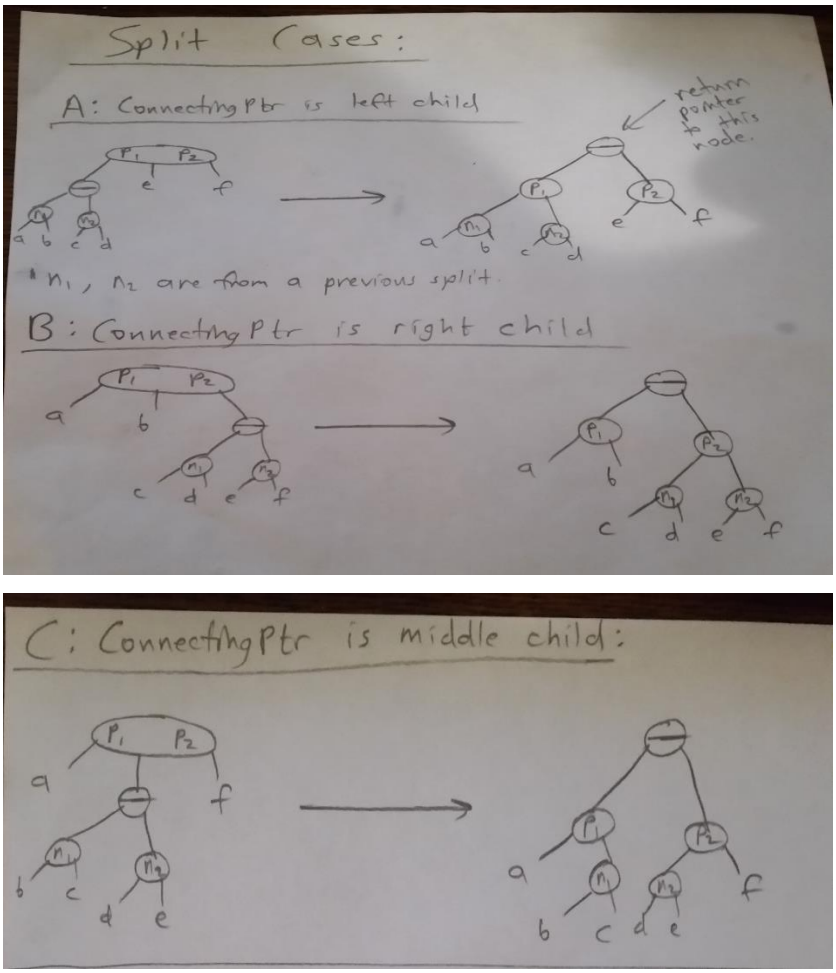


Figure 3: Cases for splitting an internal 3-node. Note that in cases A and B, the nodes n1 and n2 from a previous split are connected to p1 and p2, respectively – these correspond to cases A1 and B1 in Fig. 2, respectively. Case C is a special case, when the empty node is the middle child. Here, n1 and n2 are connected to both p1 and p2.

This process of splitting, reconnecting, and passing the middle item is repeated until either a 2-node internal parent is encountered (i.e. when there is no passed item), or the root is reached. When the root is reached, the passed item is passed back into the empty node and the root becomes this new 2-node. The insertion into the tree is then concluded.

IId - a. Removal from a 2-3 Tree

The remove function the 2-3 Tree class calls the function "removeValue," which locates the item in the tree and removes it (if it exists). As the tree is traversed, the pointers to the nodes are stored in a stack. If the item does not exist in the tree, the "removeValue" does nothing and a value of false is returned, indicating that the removal failed. However if the item is located, we have two cases to consider: (1) the node is a leaf, or (2) it is an internal node. For (1), we have two subcases: (1a) the node is a 3-node, and (1b) the node is a 2-node. For (1a), the removal is

simple – either the small or large item of the 3-node is removed, and the 3-node leaf is then converted back into a 2-node leaf: the removal process ends here. For (1b), the function "removeTwoNode" is called, with the pointer to the leaf and the stack passed as parameters. This function is described in more detail later on. Case (2) indicates that the node is an internal node and by definition of a 2-3 tree, the value in the node has an inorder successor since an internal node has to have either two children for a 2-Node, or three children for a 3-Node. The function proceeds to locate this inorder successor, again storing the pointer of each node encountered into the stack, and then swaps the two values with each other once the successor is found. After the swap, the removal of the item then follows the same two cases as that of a removal from a leaf, which were described above.

The function "removeTwoNode" has two cases, (1) the tree is a single 2-node or (2), the tree is a 2-node with a parent. For (1), the node is deleted and the rootPtr is set to the NULL pointer. For (2), the contents of the 2-Node leaf are deleted, and then its parent pointer is obtained from the top of the stack. A separate function, getSiblingPtr, returns the adjacent sibling of the node – this is described more in the comments in both the header and implementation files. We then have two cases to consider. (1) The parent is a 2-node and (2) the parent is a 3-node. For (1), we have two sub-cases: (1a) the adjacent sibling is a 2-node or (1b) the adjacent sibling is a 3-node. For (1a), we have to merge the sibling and parent into a single 3-node, accomplished via the function mergeTwoParent. The two cases for this are shown in Fig. 4.

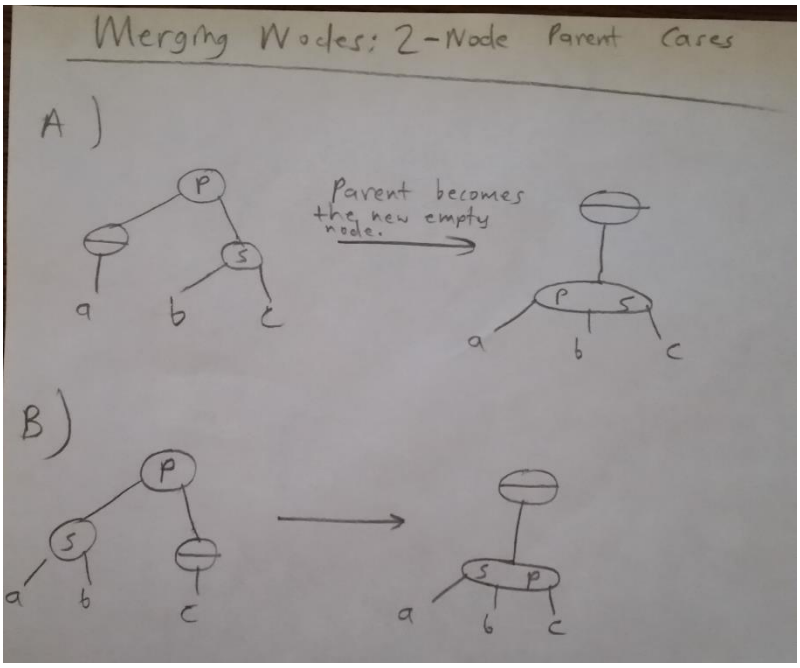


Figure 4: Cases for merging the sibling and parent nodes for a 2-node parent. Note that the empty node is removed from the tree, its child reconnected to the new, merged node, and that the parent now becomes the new empty node.

Note that merging the sibling and parent together for the case of a 2-Node parent means that the parent now becomes the empty node, so the removal is not yet complete. This is described in more detail later in the report.

For case (1b), we have to redistribute the 3-Node sibling's value across the parent and into the empty node – this is accomplished in the function redistributeTwoParent. The sibling then becomes a 2-Node, while the empty node becomes a 2-node again. There are two cases for redistribution for a 2-Node parent, outlined below in Fig. 5.

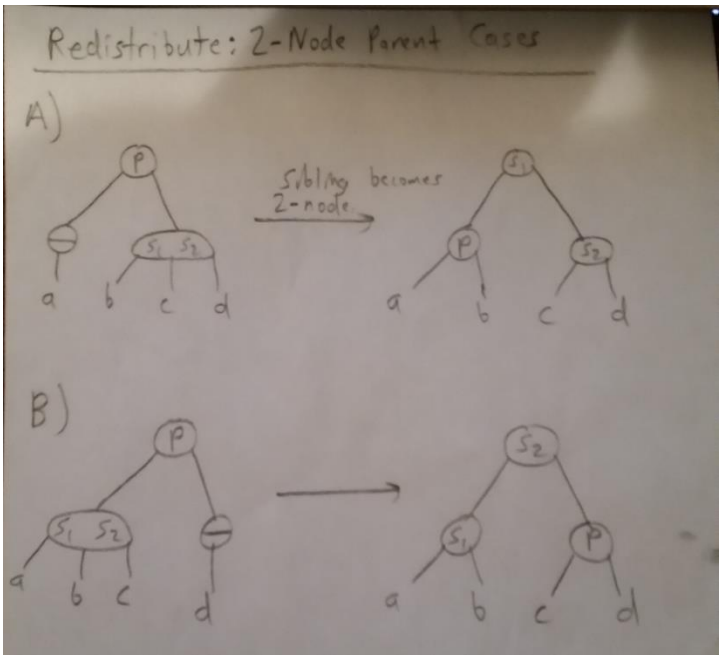


Figure 5: Cases for redistributing the value in the adjacent 3-node sibling for a 2-node parent. Note that the empty node is now a 2-node containing the parent's value, and the sibling becomes a 2-node.

Unlike Case (1a), Case (1b) concludes the removal process since the tree remains a 2-3 tree after the redistribution and requires no further traversal up the tree.

Similar to Case (1), Case (2) also has two subcases: (2a) the adjacent sibling is a 2-node, and (2b) the adjacent sibling is a 3-node. Unlike Case (1) both (2a) and (2b), after their execution, conclude the removal process. Case (2a) is similar to Case (1a) except that it converts the parent into a 2-node instead of leaving it empty and is executed in the function mergeThreeParent. The three possible cases for merging with a 3-node parent are shown below in Fig. 6.

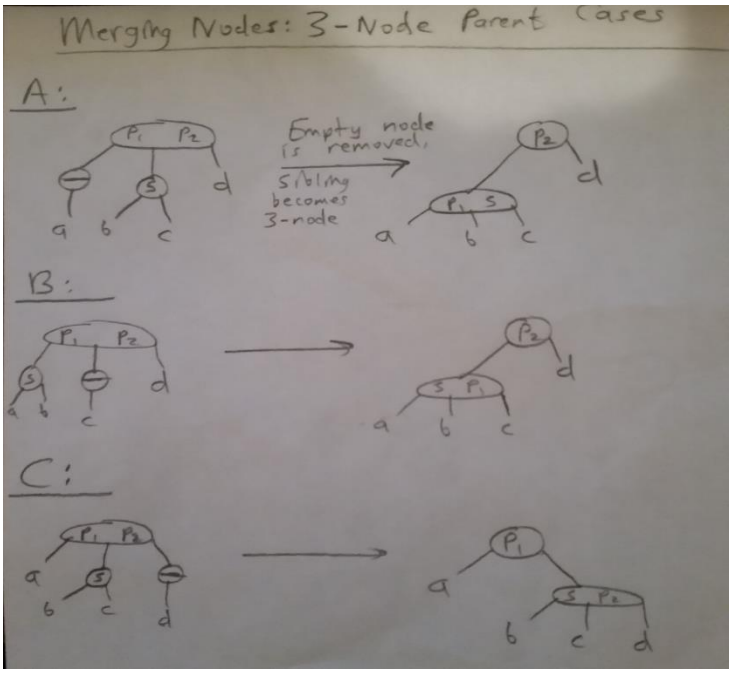
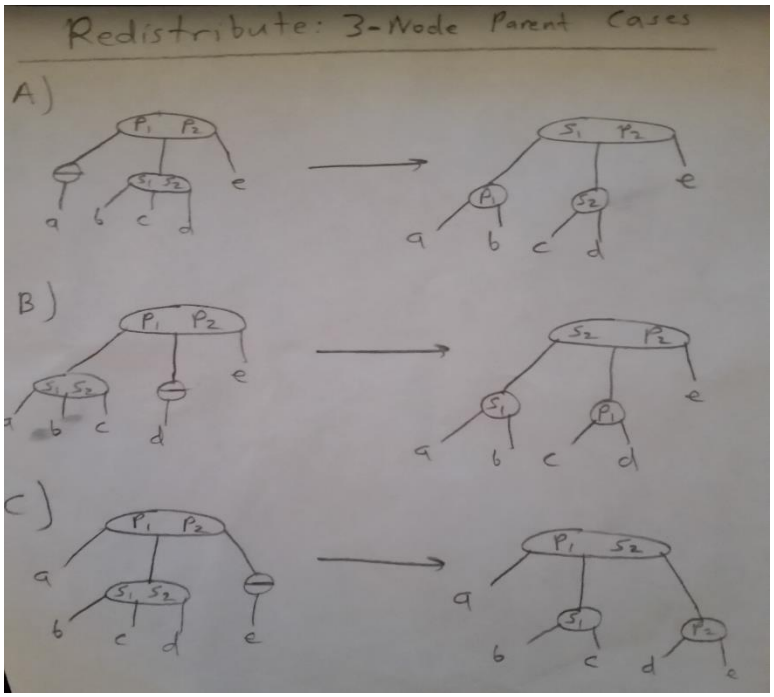


Figure 6: Cases for merging for a 3-node parent. Note that the empty node is removed, the parent node becomes a 2-node, and the sibling becomes a 3-node. Also the removal process is concluded here, unlike that in Fig. 4.

Case (2b) is also similar to Case (1b), except this time we redistribute the values of the 3-Node sibling across a 3-Node parent. There are four possible cases for this outlined in Fig. 7, with the additional fourth case arising as a result of the empty node being the middle child, where the adjacent 3-Node sibling could be either one of the parent's left or right child.



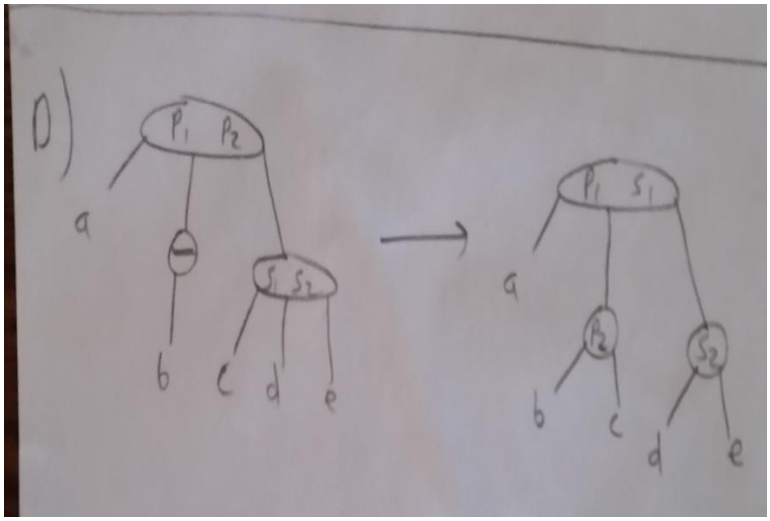


Figure 7: Cases for redistributing the value in an adjacent 3-node sibling for a 3-node parent. Note that the parent remains a 3-Node while the empty node and sibling become 2-Nodes.

Figures 4 – 7 represent all of the possible cases involved in the removal of a 2-Node leaf aside from the single node case. Note that out of all of them, only the cases in Fig. 4 leave an empty node that needs to be taken care of, which is done by traversing up the tree until either one of the cases outlined in Fig. 5 – 7 is reached. If none of these cases are reached, then we have a situation similar to the empty parent in Fig. 4, where this is now the root node connected to a 3-node child. This case is accounted for by making the root of the tree be its 3-Node child, and then subsequently deleting the empty node.

III. Hash Table Design

The hash table uses one class – an ADT "Table Interface" that details typical operations for a table. Section IIIa describes this ADT while Section IIIb discusses the implementation of the hash table itself in more detail.

IIIa. Table Interface

Table Interface is similar to the ADT Balanced Search Tree Interface, except it outlines the typical operations used in a table implementation. These are: (1) check to see if the table is empty; (2) get the number of items in the table; (3) add an item to the table; (4) remove an item from the table; (5) get the table size; (6) clear the table to the empty state; (7) retrieve an entry from the table; (8) check if an entry is in the table; and (9) traverse the table and execute some client-specified function on each item in the table. Functions 1 – 9 are described in more detail in the header file "TableInterface.h."

IIIb. Implementation of the Hash Table

The hash table uses separate chaining to resolve any collisions, with the struct "Node" used to represent the nodes in each linked chain. Inside this struct are two member variables: item of type ItemType, which stores the contents of the node and next, a pointer of Node type that points to the next node in the chain. There are four private member variables inside the class: a dynamic array of pointers of Node type; and three variables that store the table size, the total number of collisions in the table, and the number of entries in it, respectively. Along with the private member variables were several private functions written to facilitate the implementation. Finally, two new public methods were added to this implementation – the methods "writeToFile" and "displayStatistics" for the purposes of this project. "writeToFile" writes the contents of the table to an external file while "displayStatistics" outputs the relevant statistics for the hash table, described in Section I of the write-up. Finally, the table itself is dynamic – once a user specified collision size is reached, the function "expandTable" expands the contents of the table to a size that is at least 2 times the original size. The functions getNextPrime and isPrime ensure that the table size remains a prime number during this expansion, while the default constructor allows the user to specify an initial, preferably prime number for the table size.

The implementation details of the hash table are not as involved as the 2-3 tree, and are described in more detail in the comments contained in the implementation files "HashTable.h" and "HashTable.cpp". Of interest here is the hash function used to calculate the address of the table – this will be discussed below in Section IIIb – a.

IIIb – a. Hash Function

A modified version of the hash function described in Carrano and Henry (2013) was used. This function takes a string, s, of some length and converts it to an integer as follows:

1. A value between 1 – 51 is assigned to any character in the set S, which is the union of the English alphabet and the set{**(**, **)**, *****, **+**, **,** , **-**, **.** , **/**, **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**, **:** , **;** , **<**, **=**, **>**, **?** , **@**,}. The code, C, for any character x in S is computed by the following formula:

$$\boldsymbol{C = toupper(x) - 39}$$

"toupper" is used to avoid the necessity of precisely typing in the names of each media entry. The blank character is assigned the value "52," since its ASCII value is 155 and hence cannot be computed via the above formula. Note: It is unlikely that any media title will contain characters

such as (or), but including them makes it easier to write readable code without any switch statements. The function "getCharCode" in the Hash Table class computes the character code in this manner.

2. Once the character code for each character in the phrase is obtained, their 6-bit binary representations are concatenated together, and the modulus of this new binary number with the table size is taken to obtain the address.

Now, this new number is certainly going to be very large and hence will overflow the program. To facilitate calculations, a recursive version of Horner's rule was used. An example will be used to explain it. Consider the phrase "word." Let us compute the address of this phrase in our Hash Table. Using getCharCode, we obtain the following:

$$getCharCode('w') = 48 = 110000$$
$$getCharCode('o') = 40 = 101000$$
$$getCharCode('r') = 43 = 101011$$
$$getCharCode('d') = 29 = 011101$$

Concatenating these together yields the following binary number:

$$110000101000101011011101 = 12749533 \text{ in decimal}$$

This can also be computed as follows:

$$48(64)^3 + 40(64)^2 + 43(64) + 29 = 12749533$$

Using Horner's rule,

$$8(64)^3 + 40(64)^2 + 43(64) + 29 = \big((8 * 64 + 40) * 64 + 43\big) * 64 + 29$$

The modulus of this number yields the address or, letting h be the hash function and n the table size:

$$h(WORD) = \Big(\big((8 * 64 + 40) * 64 + 43\big) * 64 + 29\Big) \, mod \, n$$

Note that h(WORD) can be written as:

$$h(WORD) = \Big(\big((8 * 64 + 40) * 64 + 43\big) * 64 + 29\Big) \, mod \, n = \big((x * 64 + 29)\big) \, mod \, n$$

where x represents the highlighted region. Using the following properties for the modulus function:

$$(x + y) \, mod \, n = (x \, mod \, n + y \, mod \, n) \, mod \, n$$

$$(xy)\ mod\ n = (x\ mod\ n * y\ mod\ n)\ mod\ n$$

we have,

$$h(WORD) = (x * 64 + 29)\ mod\ n = \big((x * 64)\ mod\ n + 29\ mod\ n\big)\ mod\ n$$

$$h(WORD) = \big((x\ mod\ n * 64\ mod\ n)\ mod\ n + 29\ mod\ n\big)\ mod\ n$$

Notice that,

$$x\ mod\ n = \big((8 * 64 + 40) * 64 + 43\big)\ mod\ n = h(WOR)$$

so that,

$$h(WORD) = \big((h(WOR) * 64\ mod\ n)\ mod\ n + 29\ mod\ n\big)\ mod\ n$$

which shows the recursive nature of h(x). The base case is when the phrase is a single character of the form h(t), where the address is:

$$h(t) = \big(getCharCode(t)\big)\ mod\ n$$

Hence, we have the following recursive definition for the hash function, where t is a character and x is a string:

$$\boldsymbol{h(t) = s\ mod\ n}\ (Base\ case)$$

$$\boldsymbol{h(xt) = \big((h(x) * 64\ mod\ n)\ mod\ n + s\ mod\ n\big)\ mod\ n}\ (Recursive\ definition)$$

where s = getCharCode(t). This is what's used in the code.

IV. Media Library

There were two classes used to implement the media library: an ADT called Media Library Interface that details the typical operations of a text-based media library, and a class used to store an entry in the library called Media Entry. Section IVa describes the class "Media Library Interface"; Section IVb discusses the class "Media Entry" in detail, while Section IVc discusses the implementation details of the media library.

IVa. Media Library Interface

The ADT Media Library Interface outlines the typical operations in a text-based media library. These are: (1) add a new entry to the library; (2) remove an existing entry from the library; (3) get an entry from the library; (4) check if an entry is contained in the library; (5) display all of the movies in the library; (6) display all of the music in the library; (7) display all of the TV shows in the library; (8) display the entire contents of the library; (9) write the contents of the library to a file; (10) get the number of items in the library; and (11) display the relevant

statistics for the data structure used to implement the library. Functions 1-11 are described in more detail via the comments in the header file "MediaLibraryInterface.h."

IVb. Media Entry

The class "Media Entry" is an object that stores the important parameters of a text-based entry into a media library: its title (A), type (B), and title length (C). There are three possible types of media: movies (**M**), music/songs (**S**), and TV shows (**T**), with the character in the parenthesis being the representation of each in the program. A, B and C correspond to the private member variables in the class. Several private member functions are also defined to facilitate the implementation. Typical operations on a media entry include: (1) retrieving its title length; (2) retrieving its media type; (3) setting its title to some user-defined phrase; (4) setting the media type to some user defined type; (5) writing its contents to a file; and (6) comparing two media entries with one another. Functions 1 – 5 are straightforward and are described in the comments within the header and implementation files "MediaEntry.h" and "MediaEntry.cpp," respectively, while 6 is described in more detail in Section IVb – a.

IVb – a. Comparing Two Media Entries

Each of the boolean operators ==, <=, >=, <, >, and != are overloaded for comparison. Two media entries are identical if and only if their types and titles are the same. A media entry is less than another media entry in precedence if and only if its title comes alphabetically before the title of the other entry or, if the two titles are identical, its type has a lower precedence with songs having the highest precedence, followed by TV shows, and then movies. Because there will be non-alphabetical characters in some media titles, only the alphabetical characters in both titles are compared. This is done via the private function compareTitles with the first title being the class calling the function and the second entry being some other object of the same type. It returns a value of -1 if the title in the first entry is "alphabetically less" than the second entry; 0 if the titles are "alphabetically equal"; or 1 if the title of the first entry is "alphabetically greater" than the title of the second entry. These three terms are defined below:

Alphabetically less: A phrase, x, is alphabetically less than another phrase, y, if there exists an alphabetical character (i.e. a character that's a part of the English alphabet) in x such that that character comes before the same, corresponding character in y in the alphabet, with the upper case form of the characters being compared. If the characters in x and y are identical up to either

the length of x or the length of y, then the character with the smaller "alphabetical length", where "alphabetical length" is the number of characters in the phrase that are a part of the English alphabet, is less. The following phrases are examples of "alphabetically" less.

abc < bcd (1)

abc < B c d e f g h (2)

abc < a.-!@#b d g h j k (3)

abc < abc defg (4)

!@#@!%!%#@%#@%#@$@ < b (5)

(1) is because "A" comes before "B"; (2) is also because "A" comes before "B"; (3) is because "C" comes before "D"; (4) is because the alphabetical length of "abc," 3, is less than the alphabetical length of "abc defg," 7. Finally, (5) is because the alphabetical length of the left hand side is 0, while the alphabetical length of "b" is 1. Remember that only characters in the alphabet are compared with each other.

Alphabetically greater: The opposite of alphabetically less.

Alphabetically equal: Two phrases x and y are alphabetically equal if and only if they have the same alphabetical length and alphabetical, uppercase characters. The following are examples of alphabetical equality, denoted by =:

abcd = abcd (1)

aBC D E = abcde (2)

!@#@!#!@# = !@#@!#!$@!%!%! (3)

a. f w e -!@#@! = afwe (4)

(1) is because both have the same alphabetical length, 4, and the same characters in the uppercase form (**A, B, C and D**); (2) is the same as (1), except the alphabetical length here is 5; (3) is because both sides have an alphabetical length of 0 and no alphabetical characters; (4) is because they have the same alphabetical characters (**A, F, W and E**) as well as the same alphabetical length of 4.

Hence, the function compareTitles does the following. First it checks if one of three cases occur. Letting A be the first media entry and B be the second media entry, we have: (1) A is

empty and B is nonempty; (2) A is nonempty and B is empty; (3) A and B are either nonempty or both empty. For (1), compareTitles returns -1, since the empty string has the lowest precedence; (2) returns 1 and (3) returns the correct value for both the nonempty and empty cases. For 3, a while loop is executed. Specifically, every alphabetical character in both phrases is compared until one of the alphabetical characters in A is alphabetically less or greater than the corresponding one in B where the function returns -1 and 1, respectively, or the end of either A or B is reached. The function getNextChar returns the next alphabetical character in a phrase starting at a specified index, and it also increments the alphabetical length of this phrase via a reference parameter if an alphabetical character is found. If the end of A or B is reached, then the alphabetical lengths of both are compared. If the alphabetical length of A is less than that of B, -1 is returned and vice versa for the greater than case. If they are equal, then that means the titles are identical, and hence a value of 0 is returned. Note: For the case of both titles being empty, the while loop is not executed meaning that the function returns the default value of 0, which is correct.

V. Summary of Statistics

Twenty eight different movies, music and songs were entered into the main program; a default table size of 53 was assumed for the hash table. The data was saved in two files, 23tree.txt and hashTable.txt with the corresponding statistics of each data structure. The statistics are as follows. There were twenty two 2-Nodes and three 3-Nodes in the 2-3 tree; the height of the tree was 4. This indicates that the 2-3 tree data structure is quite efficient in its operations, as at most only four traversals are necessary to insert, locate, or remove an item from the tree. For the hash table, there were six total collisions with a maximum collision size of two, and 22 of the 53 available entries were occupied. However the table size of the hash table was relatively large with respect to the number of items in the table. Bringing the size down to 31, and then running the program again yields the following statistics: 10 collisions with a maximum collision size of 2, and 18 of the 31 available entries are occupied. So compared to last time, we see that there are more collisions in the table, and that the maximum length of any chain in the table is 3 when we include the head node. Hence compared to the four traversals in the 2-3 tree, at most three traversals are necessary to insert, locate, or remove an item in the table for both table sizes of 31 and 53. These results suggest that the hash table, overall, is a better implementation.

Given the program code, one can also test for themselves whether these statistics are true. Refer to the file my_library.txt attached with the program code to obtain all of the entries used in the present study and then run them in the main program to replicate these results. If you wish to enter more entries manually in a more efficient manner instead of typing them in the command prompt, open the file in any text editor and enter the media entry in the following format, similar to the items already in the file:

Title of the media entry

Type (either 'M', 'T', or 'S')

If the entry is the last entry in the file, ensure that there is at least one new line character in the file to facilitate program input.

VI. Conclusion

I would like to conclude this write-up by answering Question 1: Algorithms and Ethics. My stance on the ethical concerns of maintaining the customer's privacy in these databases is a bit conflicted, specifically on whom or what is responsible should a privacy breach occur. This is because, unlike the earlier periods in man's history where records were kept in a printed format, nearly anybody with a computer and internet connection has the potential to, should they possess the necessary skills, hack into and access these files for themselves no matter how well-thought the program in question is. In these types of situations, one could argue that it is entirely the hacker's fault for abusing their power and using it for a malicious purpose but, at the same time, it could be that they were able to execute such a breach because of the program itself. Specifically, that the program was poorly designed from a security standpoint. However this, too, is too simplistic because there might have been a variety of factors contributing to the poor design that were beyond the team members' control, such as the possibility that the project was rushed, that there was no coherency amongst management and the developers involved in creating the system, etc. Further, it could very well be that the hacker in question is an individual of considerable skill and brilliance that no well-designed security system could block. Regardless, these considerations indicate that it is imperative for any developer to carefully consider all possibilities of breach when the data in question is the customers' privacy but, at the same time, that in situations of breach the parties involved should understand that some aspects of a program are beyond the developers' control.