

Data Structures & Algorithms II Course Assignment 2016

B.Sc. Information Technology (AI)

Edward Fleri Soler

Table of Contents

1. Attempts.....	3
2. Introduction	4
2.1 Skip Lists.....	4
2.2 Red Black Trees.....	4
3. Skip Lists.....	5
3.1 Main Menu	5
3.2 Initialisation	5
3.3 Inserting an Integer	6
3.4 Searching by value.....	7
3.5 Searching by index	7
3.6 Deleting by value.....	7
3.7 Deleting by Index	8
3.8 List Size	8
3.9 Number of steps to search	9
3.10 Height of Skip List.....	9
3.11 Empty Skip List.....	9
3.12 Print List.....	10
4 Red Black Trees.....	11
4.1 Main Menu	11
4.2 Initialisation	11
4.3 Inserting a Value	12
4.4 Deleting a Value.....	14
4.5 Number of items.....	16
4.6 Print Height of tree	16
4.7 Empty the tree.....	17
4.8 Pre-order traversal	17
5 Discussion.....	18
5.1 Skip Lists.....	18
5.2 Red Black Trees.....	18
6 References.....	20

1. Attempts

Task	Completed	Comments
SL: Development of assignment as specified	Yes	Program functions properly and as defined
SL: Initialise	Yes	Functions properly
SL: Inserting an Item	Yes	Functions properly
SL: Finding an Item	Yes	Functions properly
SL: Delete by value	Yes	Functions properly
SL: Delete by index	Yes	Functions properly however was tricky due to data structure implemented
SL: Get item by index	Yes	Functions properly however was tricky due to data structure implemented
SL: Count items	Yes	Functions properly
SL: Number of steps to find value	Yes	Functions properly
SL: Print height	Yes	Functions properly
SL: Empty	Yes	Functions properly
SL: Overall evaluation	Yes	The program as a whole functions properly and efficiently
RBT: Development of assignment as specified	Yes	Most of the program specifications have been met with fully functional software
RBT: Initialise	Yes	Functions properly
RBT: Insert value	Yes	Functions properly
RBT: Find value	Yes	Functions properly
RBT: Delete by value	Partially	Deletion attempts on nodes in certain arrangements fail
RBT: Count	Yes	Functions properly
RBT: Print height	Yes	Functions properly
RBT: Empty the RBT	Yes	Functions properly
RBT: Pre-order traverse	Yes	Functions properly
RBT: Overall evaluation	Yes	The program as a whole has been thoroughly tested, and all parts function properly except for value deletion

2. Introduction

In this assignment, two data structures were implemented in the C programming language. Each program involved a number of different functions which could be performed on the data structures. These data structures are Skip Lists and Red Black Trees.

2.1 Skip Lists

A skip list is a linked list data structure with added features to reduce search time to that of $O(\log(n))$ with a high degree of probability. This data structure may be compared to a subway line or bus route which involves the most popular stations as well as the other, less popular yet equally important to be reachable,

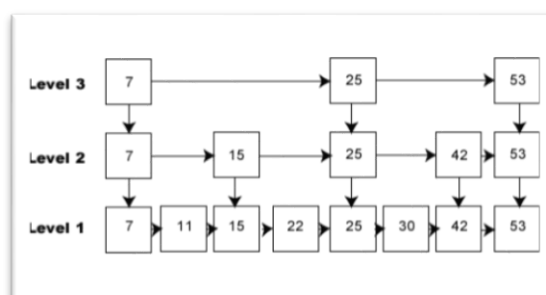


Figure 1: Example of a skip list

stations. As is the case with subway lines, express routes exist which only visit the most popular stations, skipping less popular stations and decreasing travelling time.

This idea has been translated into the digital domain, bringing about a new way of representing and handling data so as to increase access time. By having multiple lists on different levels, with each level holding less and less items, by starting a search from the top level, unnecessary items can be bypassed, reaching the required item in a shorter amount of time than sequential search.

2.2 Red Black Trees

Red Black Trees are a member of the tree family, and serve a similar purpose to other self-balancing tree structures such as the AVL tree. This data structure, however, does not require backtracking, as it may be balanced in a single top down pass. The principle behind this data structure is that nodes in a tree are coloured

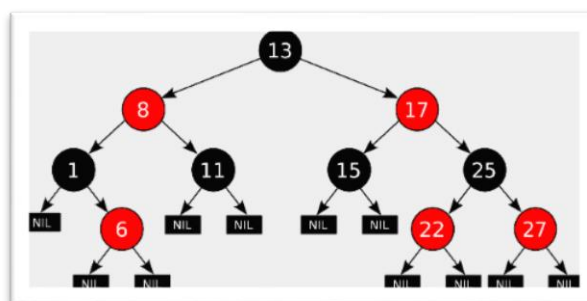


Figure 2: An example of a red black tree

black or red, and each depth first traversal must contain the same number of black nodes, while not having any two consecutive red nodes after each other.

By maintaining these two properties, the tree is self-balanced, reducing access and traversal time to $O(\log(n))$.

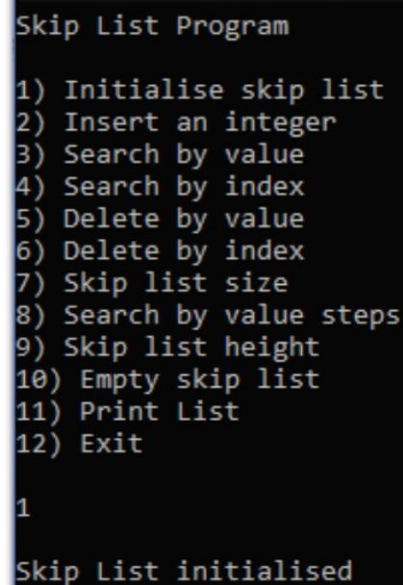
3. Skip Lists

In this section of the report, we shall discuss the manner in which the different functionalities of this data structure were implemented, including the decisions taken during the design phase and the procedures involved in each function.

3.1 Main Menu

The main menu of the program involves a basic command line interface whereby the user enters a number selecting the function that he/she would like to run. Basic input validation was performed, to ensure that the input value is indeed an integer and within range of the choices. A switch statement was then used to divert flow of the program to the respective function.

This code was placed within a loop which only exited once the exit option was selected by the user.



```
Skip List Program
1) Initialise skip list
2) Insert an integer
3) Search by value
4) Search by index
5) Delete by value
6) Delete by index
7) Skip list size
8) Search by value steps
9) Skip list height
10) Empty skip list
11) Print List
12) Exit
1
Skip List initialised
```

Figure 3: Screen shot of the program main menu

3.2 Initialisation

For purposes of efficiency, modularity and convenience, a new data type was defined via the 'typedef' command, so that new objects of this data type could easily be declared and initialised. The new 'Node' data type was that of a struct holding 7 different variables, with 4 being pointers to other Node data structures (up, down, left, right) and the remaining 3 being integer variable to store the value of each node, the height and the index. All elements within the skip list will therefore be of type node.

This approach was taken as opposed to the inbuilt linked list implementation so as to avoid relying on further complex data structures and to allow customisation of the data structure in future implementations. In this implementation, all functions were

performed on the custom designed integers and pointers. Although this decision posed somewhat of a challenge when tackling problems (such as the storage of an index number for each node on different levels) I believe that the outcome was successful and efficient.

The **initialiseList** function was called to initialise a new skip list. Tests were put in place to ensure that a list is initialised before any other functions may run, and that a list cannot be re-initialised once it has already been initialised; instead it could be emptied. Within this function, memory was dynamically allocated to a Node pointer, which will act as the head of the list on the lowest level. All pointers in this Node were initialised to NULL, and the value, height and index of the Node were initialised to the greatest negative integer number possible, 1 and -1 respectively. This initialised node will therefore never store a value, but will serve as the starting point for all searches.

3.3 Inserting an Integer

Inserting an integer into a skip list is slightly more complex than inserting one into a basic linked list. Integers must first be added to the correct position in the lowest level of the list. Within the **insertInt** function, once this position was found, and the pointers of the neighbouring nodes were changed, a probabilistic test was made to see whether to push the current node up to a higher level. This test involved generating a random number and performing modulus 2 on it, to return a binary result. In the case of a 0, the **getHeadHeight** and **insertIntAtHeight** functions were called, to respectively find the head of the next level, and insert the integer into the list at the next level. The **getheadHeight** function returns a pointer to the head of the given level. If the head for a level does not yet exist, which is the case every time a new level is reached, then the initialised function is used to generate a head, and the pointers for this head are set to point down to the head of the level below, and right to the first integer node. It is the job of the **insertIntAtHeight** function to add the new node to the right position in the new level, and to set the down/up pointers for the nodes with the same value on different levels. The probabilistic test procedure is then repeated to see whether to push the node up to yet another level.

A problem encountered at this point was ensuring that each node with the same value but on different levels stored the same index value. This was done by initially setting the index of a node upon insertion to the value of the node on the left hand side, plus

one. The indices of all nodes to the right of the newly inserted node were then incremented by means of the **incrementIndices** function, to make sure that a correct value is always stored. Similarly, the **decrementIndices** function is used in the same fashion when a node is deleted. Any node which is not on the lowest level then inherits its index value from the node below it.

3.4 Searching by value

The process of searching for a value in a skip list involves starting at the head of the highest level and traversing the list to the right, and possibly downwards, until the item is found, or an item greater is found on the lowest level, meaning that the item is not in the list. The **search** function performs this search, by accepting the head of the skip list (the head of the lowest level), and the value to search for.

Firstly, a function called **getHighestHead** returns a pointer to the head of the highest level list, which shall serve as the starting point of the search. This pointer is then passed onto the **findVal** function, which traverses the list towards the right, testing each encountered node with the search value. If a node with the same value is found, then the search is successful and the index of this node is returned. However, if the next value in the current level is seen to be greater than the required value, we go down one level, in the hope that an intermediate node storing our required value exists. This method is implemented recursively, until the item is found, or the lowest level is reached. If the lowest level is reached, and the item is still not found, then the item is not in the list and '-1' is returned.

3.5 Searching by index

Searching for a value by index is very similar to the above implementation. The search still starts from the highest level list and makes its way rightwards and downwards. However, at each point, the index values of each node are compared rather than the integer values stored in each node. Once a node with the same index as that given is found, the value is returned, otherwise if the search fails, -1 is returned.

3.6 Deleting by value

The delete function involves two main parts, searching for the node to be deleted, and arranging the pointers of the surrounding nodes. The **searchNode** function is called to search for the node to be deleted. This function takes as arguments, the head of

the lowest level of the skip list, and the value to search for. The result, being a pointer to a Node, is then passed onto the **deleteNode** function, together with the head of the skip list which is repeatedly called by means of a loop until the returned result of this function is null.

Within this function, the pointers of the nodes to the left and right are rearranged to skip the node being deleted. A test is then carried out to see whether any other nodes, other than the head of the list, exist on this level. If no other nodes exist on the level, and the list is on a level greater than the lowest, the head of the list is deleted too, by freeing its dynamically allocated memory and setting the head of the level below to point upwards to null. The node originally intended to be deleted is then freed so as to release the dynamically allocated memory. Finally, the **decrementIndices** function is called to decrement the index of each node to the right of the deleted node, if other nodes also existed on that level.

The **deleteNode** function then returns a pointer to the exact same node on the level below. This, together with the loop discussed above, deletes all occurrences of the value from the list by deleting the node from each level, completely removing it from the list.

3.7 Deleting by Index

Deleting a node by index involves the exact same procedure as was discussed above, with the slight change of searching for the node to be deleted via the **searchIndex** function discussed in 3.5 above rather than the **search** function discussed in 3.4. Therefore, once the node with the desired index is found, the exact same deletion process is undertaken.

3.8 List Size

The **listSize** function counts the number of unique items in the list. In a simple sequential manner, the head of the list acts as the starting point for the traversal, where a counter stores the number of nodes visited until a null point is reached. This value is then returned as the count of the list.

3.9 Number of steps to search

The **searchNodeSteps** function performs a search for a given value, just as the **search** function does. However, with each traversal downwards or sideways, a counter is incremented, so as to count the number of steps involved in searching for a value. As was the case with the **search** function, this function will return '-1' if the given value is not in the list. Observing the results of this function highlights the efficiency in searching that this data structure possesses over ordinary linked list, with the greater majority of searches being performed within $\log(n)$ steps, where n is the index number of the node storing that value.

3.10 Height of Skip List

The **getHeight** function was used to compute the height of the skip list. This simple function calls the **getHighestHead** function, which starting from the head of the skip list recursively traverses upwards till a null point is found, to retrieve the head of the highest list. The height value for this node is then returned as it is the height of the skip list.

3.11 Empty Skip List

Emptying the skip list involves traversing each and every node, freeing up its memory, while initialising the head of the skip list. The **emptyList** function takes as arguments a pointer to the head of the list. The head of the highest level is then retrieved by means of the **getHighestHead** function. A loop is then entered, which traverses each level to the end (until a null point is reached), freeing up each node found. Once this is done for all levels, the head of the lowest level is initialised by setting all of its pointers to null. The list is now empty.

3.12 Print List

For extra functionality and for testing purposes, an extra function was designed to print the contents of the list out on screen. This function, **printList**, takes the head of the skip list as arguments, and in a similar fashion to the **emptyList** function, starts a top down traversal of the lists, starting at the highest level. The value of each node is printed out on screen, showing the levels which a given value is present on.

```
Skip List Program
1) Initialise skip list
2) Insert an integer
3) Search by value
4) Search by index
5) Delete by value
6) Delete by index
7) Skip list size
8) Search by value steps
9) Skip list height
10) Empty skip list
11) Print List
12) Exit

11

3 | 19 104
2 | -10 19 104
1 | -10 1 12 19 104
```

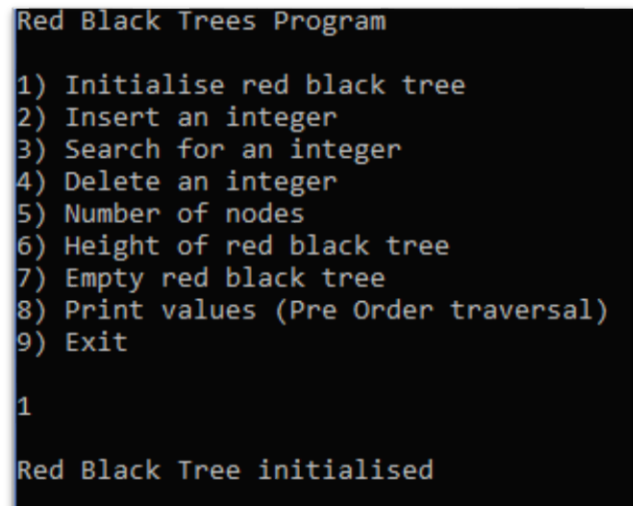
Figure 4: Print List functionality

4 Red Black Trees

In this section of the report, we shall discuss the functionality of the Red Black Trees program, any implementations and decisions taken during the design phase.

4.1 Main Menu

The main function of this program consisted of a menu where the user may select the function to perform regarding red black trees. A basic command line interface offering the user 9 possible functions was set up. Basic input validation was implemented to ensure the input value was within the menu range, and the inputted value was used to run the appropriate function by means of a switch statement.



```
Red Black Trees Program
1) Initialise red black tree
2) Insert an integer
3) Search for an integer
4) Delete an integer
5) Number of nodes
6) Height of red black tree
7) Empty red black tree
8) Print values (Pre Order traversal)
9) Exit

1

Red Black Tree initialised
```

Figure 5: Main Menu for Red Black Tree Program

This whole setup was placed within a while loop, which looped until the user chose to exit the program via the menu. As was the case with the Skip Lists program, no functions are executed until the tree is initialised, and once the tree is initialised, it cannot be re-initialised, but it can be emptied.

4.2 Initialisation

For purposes of convenience, efficiency and modularity, a new data type was defined for this program via the typedef function. This data type, called Node, was a struct containing three pointers to other nodes and two integer values. The three pointers are pointers to the left child node, to the right child node and to the parent node. Meanwhile, the integer variables store the value of the current node, and the colour of the current node. The colour value for each node could be 0 or 1, where 0 signifies a black node, while 1 signifies a red node. By defining this data type, creation of red black tree nodes may be done much more efficiently.

Initialisation of the red black tree was handled by the **initialiseRBT** function. This function dynamically allocates memory to a node pointer, which serves as the root of

the red black tree. The values of this node are then initialised to black, 0 and null respectively for the colour, value, and pointers. This pointer is then returned from the function, and stored within a variable named 'root' within the main function. The majority of functions require a pointer to the root node and therefore, this variable will be heavily used.

4.3 Inserting a Value

Insertion of a node into a red black tree is one of the more complex procedures on this data type, as upon addition of the new node, any new violations encountered must be handled. The **insertNode** function takes as arguments a pointer to the root node and the value to be inserted. This function will then return a Boolean value, describing the outcome of the insertion operation.

This function first tests to see whether any nodes have been added to the tree since it has been initialised. If this is the case, then the value of the root node is changed, rather than inserting a new node. If this isn't the case, then the procedure of insertion begins.

Firstly, the **newNode** function is called to generate a new node with the given value. This function works in the same way as the **initialiseRBT**, by dynamically allocating memory to a node pointer. However, this pointer has its colour value set to red and its integer value set to the value to be inserted. This newly created node is returned back to the insertion function, to be passed on to the **BSTSearch** function together with the root node pointer. This function performs a binary search by traversing the tree downwards, comparing the node to be inserted with the currently visited node. If the node to be inserted is greater than the current node, visit its right child; otherwise visit its left child. If the node being visited is found to be NULL, then the new node is placed there. This function therefore finds the correct location to connect the node to the tree. However, this may result in a violation of the red black tree rules; namely a red-red violation, due to the fact that the newly inserted node is red, and its parent may also be red. The handling of these violations is done by the **violationsTest** function, which accepts a pointer to the new node, as well as the root node.

Helper functions such as the **uncle** function and the **grandparent** function are used within this function. These mentioned functions perform a simple traversal to find the

uncle and grand parent of a given node respectively. If the uncle/grandparent doesn't exist, then null is returned.

The first test performed in the violation handling function, is to see whether the provided node is the root node of the tree. If this is the case, then its colour is set to black and the test is complete.

The second test to be performed checks whether the parent of the newly inserted node is black. If this is the case, no violations are present as the newly inserted node is red, leaving the tree balanced without breaching the red-red rule.

The third test is to check whether the uncle of the current node is red (as its parent is definitely red). If the uncle is black (or NULL if the uncle doesn't exist), then the insertion can be handled by means of a rotation and recolouring of nodes. The **relativePos** function is called to find the position of the current node with respect to its grandparent. A test is carried out to deduce whether the current node is the left or right child of its parent, and whether its parent is the left or right child of its grandparent. This test will therefore return a value ranging from 0 to 3, where 0 = LL Rotation, 1 = LR Rotation, 2 = RL Rotation and 3 = RR Rotation. According to the returned value, the appropriate rotation function is selected. Each rotation function performs the exact same process but with different nodes ending up in different positions. However, each function involves the altering of node pointer values, together with colour changes for the three nodes involved. All rotations result in one node becoming the parent of the other two nodes involved in the rotation. This node (parent) is coloured black while the other two nodes are coloured red. In this way, each red child has a parent which is black, fixing the red-red violation and maintaining balance within the tree.

Following any of the rotation functions, the **getRoot** function is called. This function takes the node currently pointed to as the root, and tests whether it has any parent nodes; disqualifying it as the root and renaming the new root. This test must be implemented as rotations within any of the 4 functions could have resulted in the old root being rotated out of its position and being replaced by a new root. This function therefore ensures that the correct root is found. One final step, following the rotation functions and the **getRoot** function, is to ensure that the new root is coloured black so as not to violate any rules.

The final possibility which must be handled by the **violationsTest** function, is the red uncle case. This violation occurs when the newly inserted root (which is red) has a red parent and a red uncle. Rotations and colour changing cannot overcome this violation, and therefore the **redUncle** method is defined to handle a problem like this.

The **redUncle** function generates pointers to the grandparent, parent and uncle nodes so that they may be recoloured. The grandparent node is recoloured from black to red, while the uncle and parent are recoloured to black to fix the violation. However, further tests must now be conducted on the grandparent, as the change of colour from black to red could have caused a new violation. Therefore, the **violationsTest** function is called again, passing a pointer to the grandparent node as an argument. This recursively traverses the tree upwards, fixing all violations until all properties are satisfied.

Once all violations have been handled, the insertion procedure is complete and the new node has been added. In this case, the function will return a Boolean value of true.

4.4 Deleting a Value

Deleting a node from a red black tree is the most complex task brought about by this data structure. This is due to the fact that multiple different cases, with their respective solutions must be handled so as to ensure that the tree is not violated upon deletion. The **deleteNode** function is responsible for the deleting of nodes from the tree. This function accepts as arguments a pointer to the root node and the integer value to be deleted.

A new helper function is implemented in the deletion process. This function is called **sibling** and, similarly to the **uncle** and **grandparent** functions, it traverses the tree to locate the sibling of a given node, returning a pointer to this sibling, or null in the case where the sibling doesn't exist. The first step in the deletion process begins at the root. Firstly, the root is coloured red. Next, if the two children of the root are coloured black, we traverse the child whose value is closer to the node to be deleted (BST traversal), calling a function called **case2** which shall handle such a case. This function takes the current node 'x' (which is either the left or right child of the root node) and the value to be deleted as arguments. We will take a look at this function later on.

If one, or both, of the two children of the root are red then case 2B must be handled by the function **case2B**. This function is passed a pointer to the root node and the value to be deleted.

The **case2** function tests whether the left and right child of the current node are both black. If this is the case, we move on to case 2A by passing the pointer to the current node, together with the value, to function **case2A**. Otherwise, if at least one of the two children is red, we move on to function **case2B**, with the same arguments.

Case 2A involves testing the colours of the children nodes of the sibling node. We shall refer to the sibling node as node 't'. If both of t's children are black, we move forward to case 2A1. If the left child, or both children of 't' are red, then we move on to case 2A2, otherwise (only right child is red) we move on to case 2A3.

case2A1 function involves the current node 'x', its sibling 't' and their parent 'p'. The colours of the three nodes are toggled so that the parent node is set to black while the children are set to red. Another level is traversed (again in BST fashion), passing on the pointer of the new node back to the **case2** function, to recursively move down the tree by selecting the appropriate case.

case2A2 function considers the current node 'x', its parent 'p', its sibling 't' and the left child of 't' called 'l'. This function calls **RLRotation** on the node 'l', which performs a rotation as well as recolouring, discussed previously. Following this rotation, the parent node is set to black and the current node 'x' is set to red. Finally, another BST traversal is performed on 'x', followed by the calling of **case2** again.

case2A3 function once again considers pointers to 'x', 'p', 't', 'l' and an extra 'r', which is the right child of 't' and the sibling of 'l'. A rotation is performed, bringing 't' to become the new parent of 'p' and 'r'. The nodes are once again recoloured, setting 'x' and 't' to red, while 'p' and 'r' are set to black. Once again, a new level is traversed and the new node is passed on to the **case2** function.

The **case2B** function is the final function to be called before the node is deleted. If the current node being visited holds to value to be deleted, then the node is freed via the **removeNode** function, which frees the node's memory and rearranges the pointers of its parent node and possible child node. If this is not the case, further tests are required. If the current node is red, then another level of the tree is traversed, and the

function recursively calls itself with the new node. Otherwise, (current node is black), a rotation is performed, bringing 't' to become the parent of node 'x', and recolouring 't' to black and 'x' to red. Node 'x' is then passed onto **case2** again, to recursively traverse the tree until the node is found.

Once the node has been removed by either of these cases, the root is reset to its original colour of black, and a Boolean value of true is returned if the procedure was successful.

4.5 Number of items

The **numNodes** function is responsible for counting the number of nodes within the red black tree. This function makes use of a static counter which is initialised to 0. Starting with the root, the counter is incremented if the current node is not null, and a recursive call is made to the same function with a pointer to the left and right children respectively. This function then returns the value of the static variable count.

Following the complete traversal of the tree, this variable will hold the number of nodes in the tree. However, since the static variable will never be reset to 0, successive calls of this function would lead to an incorrect count. Therefore, an added Boolean variable is passed as an argument. When this argument is true, the counter is reset to 0, otherwise it isn't. Then when the function is called from the main menu, the Boolean value is set to true to re-initialise the counter, however all successive recursive calls pass a value of false to stop the counter from being reinitialised.

4.6 Print Height of tree

In a similar fashion to the **numNodes** function, the **height** function calculates the height of the tree via a static variable. This variable, called 'max', is initialised to 0 by means of the exact same Boolean argument (when true). A counter called 'h' is also passed as an argument, and is set to 0 when the **height** function is called from the main menu. Starting from the root, if the node passed as an argument is not null, the height 'h' is incremented. If the height 'h' is greater than 'max', 'max' is updated. Recursive calls are then made to the same function, passing the left and right children of the current node, and the Boolean value false to stop the counter from being initialised to 0. On complete traversal of the tree, this function will return the depth of the deepest sub-tree, which is the height of the tree.

4.7 Empty the tree

The **empty** function is responsible for emptying the tree of all nodes, returning it to the state it was in when it was initialised. This function takes a node pointer and a pointer to the root as arguments. The given node pointer is tested to see whether it has a left child. If so, the **empty** function is recursively called upon the left child. The same process is performed on the right child. Next, the current node is compared to the root. If the current node is indeed the root, then its value is set to 0, its colour to black, and its pointers all to null. Otherwise (current node is not the root), the node's memory is released via the free function. Therefore, by means of recursion, all nodes in the tree are deleted and the root node is initialised. When calling this function from the main function, both arguments given are a pointer to the root function.

4.8 Pre-order traversal

The **preOrder** function is responsible for traversing the tree in pre-order fashion, outputting the value at each visited node. This function takes a pointer to a node as an argument. If the current node is not null, then its value is output on screen. If the current node is red, then the value is output onscreen inside of square brackets ([]). Finally, tail recursion is implemented to perform the exact same procedure on the left child, and then the right child of the current node, resulting in the pre-order traversal of the tree.

```
Red Black Trees Program
1) Initialise red black tree
2) Insert an integer
3) Search for an integer
4) Delete an integer
5) Number of nodes
6) Height of red black tree
7) Empty red black tree
8) Print values (Pre Order traversal)
9) Exit
8
19 12 [-10][45]34 100 [77]
```

Figure 6: Result of pre-order traversal on tree containing 7 nodes

5 Discussion

In this section of the report, we shall discuss the final outcome and performance of each program, mentioning any shortcomings and possible improvements for future implementations.

5.1 Skip Lists

Following debugging and testing of the program, it was found that all parts of the program work well and efficiently, with no major errors being encountered during testing. I believe that the generation and declaration of a new custom data type was a good decision during the design process as in future implementations, nodes can be customised to hold different amounts and types of data without changing much of the inner workings of the program. I also believe, that although the creation of a new node on every level rather than adjusting pointers to point to the same node, was a good idea, as although more memory is required (as each value may result in multiple nodes being created), the program was much easier to understand, implement and test.

On the whole, I believe that the outcome of the program is satisfactory, and no great improvements could be made which would have a great effect on the efficiency and functionality of the program.

5.2 Red Black Trees

Following debugging and testing of the program, it was understood that all parts of the program meet their functionality requirement, with exception to the delete function. It was found that this part of the program did not work for all cases of deletion. On some occasions, the given value was successfully deleted and removed from the tree without violating any rules; however on other occasions, the given value was not removed, leaving the tree unaffected.

Given that this process is such a complicated one, involving multiple different cases and possibilities, an error in the work flow most likely exists. However, following rigorous testing, re-testing and changes to the code, this error was not overcome. All other parts of the program were found to function properly and efficiently, with the correct violation handling techniques being implemented to solve a given violation upon insertion.

As was mentioned with skip lists, I believe the decision to create a custom data type was a good one, as this allows for extra customisation and adaptation in future implementations. I also believe that the program as a whole works well and meets all requirements (other than the above mentioned delete feature), with few possible improvements to the program as a whole that I am aware of.

6 References

1. Figure 1: <https://de.wikipedia.org/wiki/Rot-Schwarz-Baum>
2. Figure 2: <http://www.php5dp.com/tag/linked-lists/>
3. Red Black Tree insertion inspiration: <http://www.geeksforgeeks.org/red-black-tree-set-2-insert/>