Imperial College London

Computer Lab Assignment Report

Building, Simplifying and Evaluating Binary Trees

Lei Wang (Wilson) - 01214043

Submitted: May 10, 2018

Contents

Part I

The struct array_aux

1 General Introduction

In the assignment, the students were asked to provide a piece of C++ coding, that would build a tree with the minimum number of nodes, and would evaluate the built trees. Simplification must take place to reduce the number of nodes in the tree. Two common approaches are to build first and simplify after, or to simplify first and build after.

I will be using the latter case, which is simplifying first and then building the tree.

The number of characters n in each string input is passed into the constructor of array_aux. The user inputs are stored in the *seq_2D_char_input* matrix, implemented using a 2D char array. The simplification will then take place. The 1D int array, *seq_1D_int*, is of length 2^n. In the 1D int array, the elements hold the default value of -1. If an input, for example, 1001, is passed in, the program will first convert 1001 to 9. The binary to decimal conversion is done outside the struct, using a finction called *binaryToInt*. The program will go to the $9th$ location in the *seq_1D_int* array and set the value at that location to a number and the number is the index in the *seq_2D_char_input* matrix where the input is stored. Every time an input is stored, the program will find the next location in the matrix where no valid records are stored. The index of that location is stored in the *seq_1D_int* array at the location that is equal to the decimal value of the input binary string. In other words, the int array is used as a mapping, to show the location of the record in the matrix. It is a way to reduce the size of the matrix. Suppose if the matrix is initialized with a dimension of $2^n * n$, while there are only 5 inputs. In this case, $2^n - 5$ rows are wasted, which is not RAM efficient. Hence the program uses a list of integers to show the index of the record in the record matrix. The size of the record matrix, *seq_2D_char_input*, can be reduced to the total number of inputs from the input vector. Also, *seq_1D_2D_available_space* is used to show whether there is a record at the index given in the *seq_2D_char_input* matrix, so that the program does not have to compare every element in a record in the matrix to determine whether the record is valid or not.

By hiding the complexity of storing the input binary strings, the data structure can be viewed in table 2. When the program wants to access a record whose decimal value is k, the program will first go to the kth location in the int array and retrieve the information stored at that location. If the retrieved number j is not -1, meaning it holds a valid record, the program will then go to the jth location in the char matrix to get the exact information in the record. If a record is to be deleted, the jth element in the *seq_1D_2D_available_space* is set to false. In the following explanation of the program, the way of describing data structure is in table 2. The actual implementaion is different from implementing a matrix like table 2.

Name	Type	Dimension	Explanation
input_string_length	int		number of characters in each input string
array_length_1D	int		total number of different possible inputs
vector_len	int		number of elements in the input vector
seq_1D_int	int array	2^n	array stores the indices of routes
seq_2D_char_input	char matrix	$vector\_len * n$	matrix stores the input
seq_1D_2D_available_space	bool array	*vector_len*	shows if a record exists in the 2D char table
seq_2D_int_sort	int matrix	$n * 6$	used in sorting

Table 1: Members of the struct *array_aux*

	bit 0	bit 1	bit 2	bit 3	....	bit n-1
row 0	$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	....	$a_{0,n-1}$
row 1	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	....	$a_{1,n-1}$
row 2	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	....	$a_{2,n-1}$
row 3	$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	....	$a_{3,n-1}$
....	....	....	....	....	....	....

Table 2: Data structure that stores the inputs with dimension $2^n * n$

	bit 0	bit 1	bit 2	bit 3
record 9	1	0	0	1

Table 3: Example: input = 1001 (9 in decimal), stored in the *seq_2D_char_input* matrix

In the *seq_2D_char_input* matrix, if a record exists, the corresponding bits will be set according to the binary strings input by the user. The matrix contains all the possible combinations of different bits. Each bit in the matrix will take one of the 3 possible values, 0, 1 or 2. To set the bits in the matrix, the binary string is converted to the corresponding decimal value, denoted as k. Then the program will go to the kth row in matrix. The row has the identical length as the length of each individual input string. Each bit in the row is set to either 1 or 0 based on the binary. Also, in the *seq_1D_2D_available_space* array, the element that links to the record is set to true.

The input binary strings are stored in a vector. The program will iterate through all the strings in the vector. For the rows in the *seq_2D_char_input* matrix that are not operated on, the default value of the bits in those rows is 2. During simplification, the bits that originally hold a 1 or 0 can be set to 2 as well. If a row is filled with 2, the row does not hold a record and does contribute to the structure of the simplified binary tree. The corresponding elements in the *seq_1D_2D_available_space* array will have values of false. To check whether a record exists at a given index m, the program does not have to check all the bits in the mth row in the matrix. Instead, using the value at the mth element in the *seq_1D_int* array as an index and using the index to check in the *seq_1D_2D_available_space* array is sufficient.

The *seq_2D_int_sort* matrix has a different dimension from the previously explained *seq_2D_char_input* matrix. The *seq_2D_int_sort* matrix holds information on certain bit position in any binary string, instead of each binary string from the input. A row number q in the matrix corresponds to the qth bit in any input string. The values in each row in the matrix are expianed in table 5. Note that the values in the *seq_2D_int_sort* matrix are set after simplification. Bit 2 and bit 3 in row q are set by counting the number of 1s and 0s in the *seq_2D_char_input* matrix at column q.

	bit 0	bit 1	bit 2	bit 3	bit 4	bit 5
row 0	$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
row 1	$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
row 2	$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$
row 3	$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$
....	....	....	....	....	....	....
row n	$a_{n,0}$	$a_{n,1}$	$a_{n,2}$	$a_{n,3}$	$a_{n,4}$	$a_{n,5}$

Table 4: Structure of the *seq_2D_int_sort* matrix with dimension $n * 6$

	bit 0	bit 1	bit 2	bit 3	bit 4	bit 5
row n	n	counter: bit $n = 0$	counter: bit $n = 1$	max of bit 1&2 at row n	sum of bit 1&2 at row n	weight of the bit n

Table 5: Explanation of each bit in the *seq_2D_int_sort* matrix

Function	Description
array_aux(int len, int len_of_vector)	constructor, initialize the members in the struct
∼array_aux()	destructor, release memory
int return_array_length_1D()	return the member array_length_1D
void seq_1D_int_set(int position, bool val)	change the element in the seq_1D_int array
int seq_1D_int_return(int position)	return one element in the seq_1D_int array
void seq_1D_int_test()	print the elements with index in the seq_1D_bool array
void seq_2D_char_input_set (int position, std::string binary_string)	set one row in the seq_2D_char_input matrix at given index using given string
void seq_2D_char_input_set_bit (int position, int bit, char info)	set certain bit in the seq_2D_char_input matrix using the given index
char seq_2D_char_input_return(int x, int y)	return the element at give index in the seq_2D_char_input matrix
int seq_2D_char_input_weight(int position)	treat 2 as 1 and calculate the decimal value at give index
void seq_2D_char_input_test()	print the seq_2D_char_input matrix
int check_available_routes()	check the number of routes that will be used in building the tree
void delete_route(int position)	delete a route at give index
void seq_2D_int_sort_bit_set()	update elements in the seq_2D_int_sort matrix using the seq_2D_char_input matrix
void seq_2D_int_sort_bit_sort(int bit_sort)	sort the seq_2D_int_sort matrix using a certain column
void seq_2D_int_sort_bit_run()	sort the seq_2D_int_sort matrix using column 4 first, followed by column 3
int seq_2D_int_sort_return(int x, int y)	return one element in the seq_2D_int_sort matrix
void seq_2D_int_sort_test()	print the seq_2D_int_sort matrix
void seq_2D_int_sort_priority_gen()	generate the priority of each bit and store them in the seq_2D_int_sort matrix

Table 6: Member functions in the struct *array_aux*

2 The Constructor

The function *array_aux()* is the constructor of the struct *array_aux* and takes in two arguments, which is the length of one individual input binary string (denoted as h), and the size of the input vector (denoted as v). The constructor first set the *input_string_length* member to h and *vector_len* to v. Then the constructor calculates *array_length_1D*, which is the total number of possible combinations, using the formula 2^h. *array_length_1D* and *input_string_length* are used to initialize the *seq_1D_2D_available_space* array, the *seq_2D_char_input* matrix and the *seq_2D_int_sort* matrix. The dimensions of those arrays and matrices can be found in table 1.

3 Check Available Routes

The function *check_available_routes()* is used to count how many elements that hold the value of true in the *seq_1D_bool* array. The returned result indicates the number of available routes that can be used to build the simplified the binary tree. The returned value will change when the function is called at different simplification cycles. Initially, the function is invoked before and after a simplification cycle. If the returned values are the same, then the whole simplification process has reached an end and no more simplification is possible. The program will then move onto the building stage.

4 Delete Routes

The function *delete_route(int position)* is used to delete a route at a given index. To delete the route, the function will set all the bits in the respective row in the *seq_2D_char_input* matrix to 2 and the respective element in the *seq_1D_2D_available_space* array to false.

5 Collect, Sort, Generate

The function *seq_2D_int_sort_bit_set()* will count the number of 1s and 0s at certain bits. In other words, the function counts the number of 1s at bit 0, bit 1, bit 2...until the last possible bit among all the available routes. And store the result in the third element at row 0, row 1, row 2...until the last row in the seq_2D_int_sort matrix. Then the program will do the same for counting 0s, but the result is stored in the second element in each row in the matrix. The fourth element in each row will store the bigger number by comparing the second and the third element in the same row. The fifth element will be the sum of the second and the third element in the same row. The sixth element in the row is the priority of the bit position associated with the row. The first element in the row of the matrix is the row number (counting from 0) plus 1, since the node numbering in the final built tree starts from 1, instead of 0.

The function *seq_2D_int_sort_bit_sort(int bit_sort)* will arrange the rows in the seq_2D_int_sort matrix. The principle of rearranging is comparing elements in a column of the mateix. The column number will be given when invoking the function. The algorithm used is bubble sort.

The function *seq_2D_int_sort_bit_run()* will run the function *seq_2D_int_sort_bit_sort(int bit_sort)* twice, but the parameters passed in are 4 first followed by 3. This means that the rows in the seq_2D_int_sort matrix will be sorted twice, first by the column showing the sum, followd by the column showing the maximum.

seq_2D_int_sort_priority_gen() is used to generate the priority of each bit position and fill in the sixth column in the seq_2D_int_sort matrix. The priority starts from 1 and increment by 1 every time a new and lower priority level is reached. The smaller the value of the priority level is, the higher the priority will be. Two bit positions, or two rows, will only have the same level of priority if and only if the max and the sum elements are the same. For instance, the max of row 4 equals that of row 5, and the sum of row 4 equals that of row 5, then row 4 and row 5 will have the same priority. If any of the two requirements are not satisfied, the priority generated will be different. Rows on the top of the matrix will have the highest priority.

Table 7 to Table 11 show the way of rearranging and priority generating.

	bit 0	bit 1	bit 2
row 0	2	2	2
row 1	0	0	1
row 2	2	2	2
row 3	2	2	2
row 4	1	0	0
row 5	2	2	2
row 6	2	2	0
row 7	1	1	1

Table 7: Demo: input table

	bit 0	bit 1	bit 2	bit 3	bit 4	bit 5
row 0	0	1	2	2	3	0
row 1	1	2	1	2	3	0
row 2	2	2	2	2	4	0

Table 8: Demo: seq_2D_int_sort matrix, count for 0s and 1s updated

	bit 0	bit 1	bit 2	bit 3	bit 4	bit 5
row 2	2	2	2	2	4	0
row 0	0	1	2	2	3	0
row 1	1	2	1	2	3	0

Table 9: Demo: seq_2D_int_sort matrix, sort using the sum (bit 4)

	bit 0	bit 1	bit 2	bit 3	bit 4	bit 5
row 2	2	2	2	2	4	0
row 0	0	1	2	2	3	0
row 1	1	2	1	2	3	0

Table 10: Demo: seq_2D_int_sort matrix, sort using the max (bit 3)

	bit 0	bit 1	bit 2	bit 3	bit 4	bit 5
row 2	2	2	2	2	4	1
row 0	0	1	2	2	3	2
row 1	1	2	1	2	3	2

Table 11: Demo: seq_2D_int_sort matrix, priority added

Part II

buildcompactbdt

1 Loading Inputs

In the first stage of building the tree, an *array_aux* object is instantiated and is called *array_gen*. The parameter passed in is the length of one individual input string. To load the binary strings, a For loop is used to iterate through the elements in the input string vector. Function *binaryToInt* is invoked to convert the binary string to a decimal integer. The input string will then be stored in *array_gen* at index that corresponds to its decimal value.

2 Extreme Cases

Two extreme cases exist after all the inputs are loaded. The first one is that the number of valid records is 0, meaning regardless of the inputs, the tree will always produce a 0 at the output. The solution to this extreme case is to build a tree with only one node that has the value of 0. The pointer of the node is returned and the *buildcompactbdt()* function is terminated.

The other extreme case is that the number of valid records equals the total number of possible inputs. In other words, the binary tree will always produce a 1 regardless of the inputs. The solution is to build a tree with only one node that has the value of 1. The pointer of the node is returned and the *buildcompactbdt()* function is terminated.

3 Calculating Weights and Simplification

After loading inputs comes the simplifying stage. The principle is simple Boolean algebra. If two records differ by only 1 bit while both of them give out a 1 as the output. Then the different bit can be treated as the *don't care* bit and can be removed in both records, resulting in a route that is 1 bit shorter than the orignal route. The second record in the comparison is deleted by calling the member function *delete_route* using the index of the second record.

Question on finding the index of the second record arises. Since the two records only differ by one bit. And the bit position is known. Hence technically the index of the second record can be calculated. If the second record does not exist, or it does not produce a output as 1, then the comparison does not produce further simplification at certain bit position with certain record. The program will move on to the next record that will show a 1 at the output.

Before simplification:

	bit 0	bit 1	bit 2
row 0	2	2	2
row 1	2	2	2
row 2	0	1	0
row 3	2	2	2
row 4	2	2	2
row 5	2	2	2
row 6	1	1	0
row 7	2	2	2

After simplification:

	bit 0	bit 1	bit 2
row 0	2	2	2
row 1	2	2	2
row 2	2	1	0
row 3	2	2	2
row 4	2	2	2
row 5	2	2	2
row 6	2	2	2
row 7	2	2	2

In the above example, row 2 and row 6 are being compared. The pointer for the records traverses from the top to the bottom of the matrix. In other words, the record pointer moves from smaller index to larger index in each cycle of simplification. And horizontal pointer for the bit moves from the left to the right. In this demonstration, the record pointer points to row 2 and the bit pointer points at bit 0. The second record for comparison should locate at distance of $2^2 = 4$ after the first record. The first record for comparison locates at index 2. Hence the second record for comparison locates at index 6. At index 6, a record exists that will produce a 1 at the output. Hence, for the first record, or the record at index 2 in the matrix, its bit 0 can be treated as *don't care case* and can be removed from the record at index 2. The record at index 6 is removed as well as a result of the simplification.

Each simplification operation can only simplify one bit in one record at one time. Once the record is modified, the program will move on to the next available record. Multiple operation on the same record is not allowed in the same simplification cycle. The program will start over, or start a new simplification cycle, from the very first record in the input matrix once it has reached the end of the matrix. To determine whether the table has reached its most simplified form, the program will do an available route counting before and after each simplification cycle. If the counts possess the same value, then the whole simplification process is done and the program will move to the stage of sorting the routes.

During simplification, some routes will be deleted and some nodes in the route will be set to 2. The bit labeled as 2 does not contribute to building the simplified tree. In the program, a 2 in a simplified route is treated as 1. Hence the weight, or the corresponding decimal value of that route does not equal to the index of the route. See an example below for explanation.

	bit 0	bit 1	bit 2
row 0	0	0	0
row 1	0	0	1
row 2	0	1	0
row 3	0	1	1
row 4	1	0	0
row 5	2	2	2
row 6	2	2	2
row 7	2	2	2

The number of valid Each cycle of simplification starts from row 0. The number of valid records is 5. At row 0 bit 0, a 0 is present. The weight of row 0 now is 0. The distance is $2^2 = 4$. The row for comparison locate at row $0 + 4 = 4$. Row 4 has a weight of 4, which is the same as the weight of row 1 plus the distance.

	bit 0	bit 1	bit 2
row 0	2	0	0
row 1	0	0	1
row 2	0	1	0
row 3	0	1	1
row 4	2	2	2
row 5	2	2	2
row 6	2	2	2
row 7	2	2	2

Simplification between row 0 and row 4 takes place. All elements in row 4 is deleted by setting them to 2. Also, at row 0 bit 0, it is set to 2 to indicate the bit is removed from the route.

	bit 0	bit 1	bit 2
row 0	2	0	0
row 1	0	0	1
row 2	0	1	0
row 3	0	1	1
row 4	2	2	2
row 5	2	2	2
row 6	2	2	2
row 7	2	2	2

Each row can only have one simplification in each simplification cycle. Hence the program will move to row 1. The weight of row 1 now is 1. The program now looks at row 1 bit 0, the distance is $2^2 = 4$. Hence the row for comparison is row $1 + 4 = 5$. However, row 5 is not a valid record since all its elements are 2.

	bit 0	bit 1	bit 2
row 0	2	0	0
row 1	0	0	1
row 2	0	1	0
row 3	0	1	1
row 4	2	2	2
row 5	2	2	2
row 6	2	2	2
row 7	2	2	2

The program now looks at row 1 bit 1. The corresponding distance is $2^1 = 2$. Hence the row for comparison is row $1+2 = 3$. The weight of row 1 now is 1. The weight of row 3 now is 3, which is the same as the weight of row 1 plus the distance.

	bit 0	bit 1	bit 2
row 0	2	0	0
row 1	0	2	1
row 2	0	1	0
row 3	2	2	2
row 4	2	2	2
row 5	2	2	2
row 6	2	2	2
row 7	2	2	2

Simplification takes place between row 1 and row 3. Row 1 bit 1 is set to 2 and row 3 is deleted by setting all elements to 2.

	bit 0	bit 1	bit 2
row 0	2	0	0
row 1	0	2	1
row 2	0	1	0
row 3	2	2	2
row 4	2	2	2
row 5	2	2	2
row 6	2	2	2
row 7	2	2	2

The program now moves to row 2. Row 2 bit 0 is 0 and the distance calculated is 4. Hence the row for comparison is row 6. However, row 6 is not a valid route. The program then moves to row 2 bit 1. But the matrix produces a 1 at this location. Hence this location is skipped for simplification.

	bit 0	bit 1	bit 2
row 0	2	0	0
row 1	0	2	1
row 2	0	1	0
row 3	2	2	2
row 4	2	2	2
row 5	2	2	2
row 6	2	2	2
row 7	2	2	2

The next bit for comparison is row 2 bit 2. The distance calculated is $2^0 = 1$. Hence the row for comparison is row 3. But row 3 is not a valid record. This cycle of simplification reaches an end since row 2 is the last row that produces a valid record. The number of valid records is 3, which is difference from the number of valid records at the start of this simplification cycle. Hence another cycle will start from the first valid record, which is row 0 in this case.

	bit 0	bit 1	bit 2
row 0	2	0	0
row 1	0	2	1
row 2	0	1	0
row 3	2	2	2
row 4	2	2	2
row 5	2	2	2
row 6	2	2	2
row 7	2	2	2

At the start of this simplification cycle, the number of valid records is 3. Since 2 is treated as 1 in this algorithm, the first position for comparison is row 0 bit 1. The corresponding distance is 2. The weight of row 0 now is 4. The row for comparison is row 2 with weight 2, which is not the same as the weight of row 0 plus the distance. Hence the simplification does not take place between row 0 and row 2 at bit position 1.

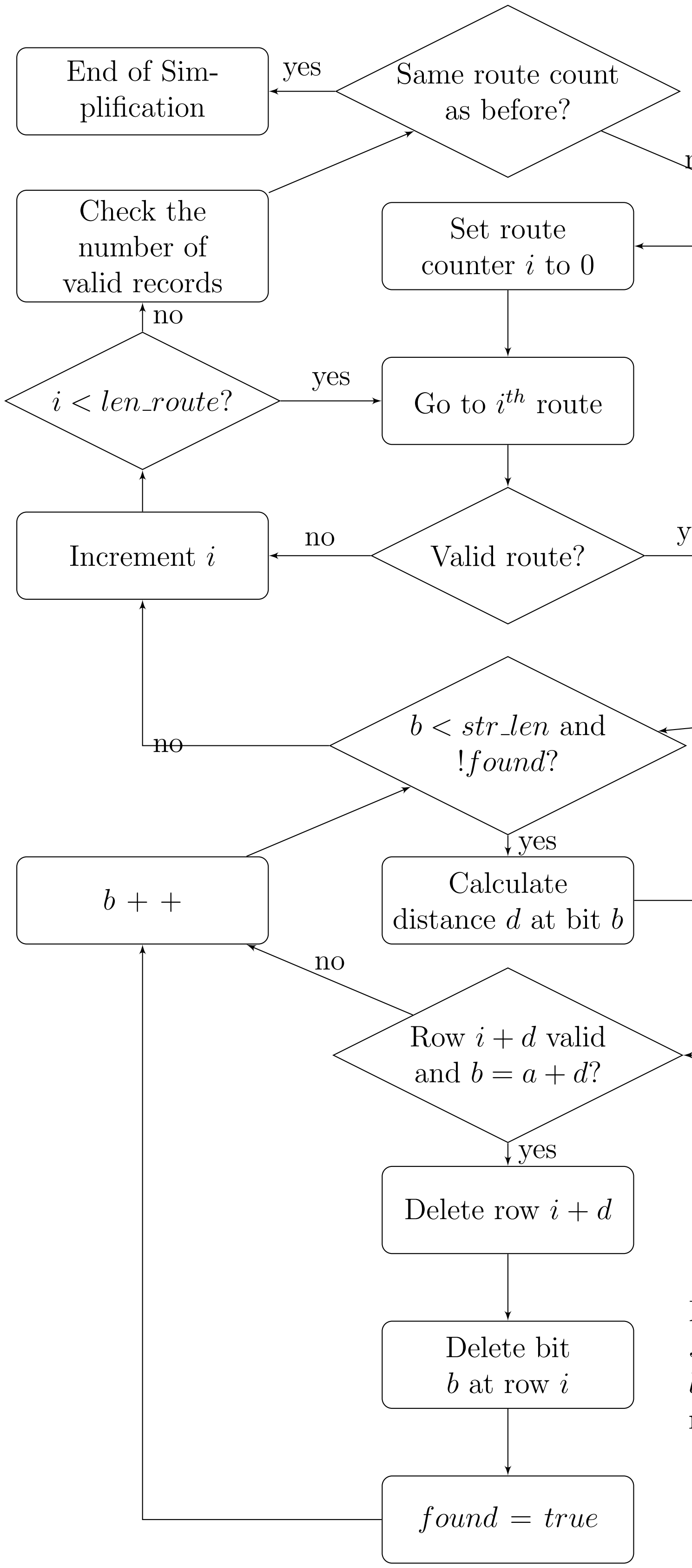
	bit 0	bit 1	bit 2
row 0	2	0	0
row 1	0	2	1
row 2	0	1	0
row 3	2	2	2
row 4	2	2	2
row 5	2	2	2
row 6	2	2	2
row 7	2	2	2

The next comparison is between row 0 and row 1 at bit 2. The weight of row 0 is 4. The distance is 1. The weight of row 1 is 3, which is not the same as the weight of row 0 plus the distance. Hence no simplification takes place.

The remaining discussion on whether the simplification will take place follows the same pattern. In this case, no other simplification cycle is possible. The final matrix will be the following:

	bit 0	bit 1	bit 2
row 0	2	0	0
row 1	0	2	1
row 2	0	1	0
row 3	2	2	2
row 4	2	2	2
row 5	2	2	2
row 6	2	2	2
row 7	2	2	2

Row 0, 1 and 2 will be used for building the binary tree.



Note:
str_len is the length of a binary string
len_route is the total number of all possible routes 2^{str_len}

Figure 1: Flowchart for simplfying the routes

4 Sorting Routes

Member functions *seq_2D_int_sort_bit_run()* and *seq_2D_int_sort_priority_gen()* will be invoked for generating the priority of each bit position.

The program will then initialize a $t*2$ matrix. t is the number of availe-ble routes, which can be obtained using the *check_available_routes()* member function. In each row of the matrix, the first element will be the route/valid record position and the second element will be the calculated weight of the route. The matrix does not hold the specific information on the route but the priority level and the index of the route. The priority is calculated using the priority generated for each bit position. Note than a 2 in the route record does not contribute to the weight of the route but a 1 or a 0 will contribute to the weight. In other words, when there is a 0 or a 1 in a certain bit position in a certain record, the weight or the priority of the bit position is added to the total weight of the route. Once the matrix that holds the route index and the route weights is generated, it will be sorted with bubble sort using the weight of each route as the sorting key.

In program's implementation, a low weight, or a small value in the priority indicates a high priority, meaning the route associated with the weight should be built prior to the remaining routes. The first route with the highest priority will be built first so that the other routes can have a place to build upon.

5 Building Routes and Filling 0s

Since there are nodes in a route that will be ignored, each route could have different lengths. In order to determining the end of a route, the number of nodes that will be used in building has to be available before building the route. The number of useful nodes in a route is determined by checking all the emelents in the route using a For loop. The counter is set to be equal to the length of a binary string (the length of each string from inputs). If the element is a 2, the counter will be decremented. Back to building the tree, every time a node is added to the route, the counter for the no-ignore nodes will decrement. The route will reach an end when the counter reaches 0 and in the end of the route, a 1 will be added to the label of the last non-leaf node. The sequence of the nodes in building each route, from the top to the bottom of the tree, will be following the node, or the bit priorities, which is determined previously by calling the member functions. The standard procedure of building a route will be getting the route index, getting the bit position in the bit priority sequence, and checking the corresponding bit in the route. If the result is a 1, then a new node will be generated and labeled with the bit position, by calling the *node-Namer()* function. Then the pointer will point to the right of the current node. If the result is 0, then the same procedure will apply other than the pointer will be pointing to the left of the current node. If the result is 2, then no new node will be added to the tree. The pointer will stay at the current position.

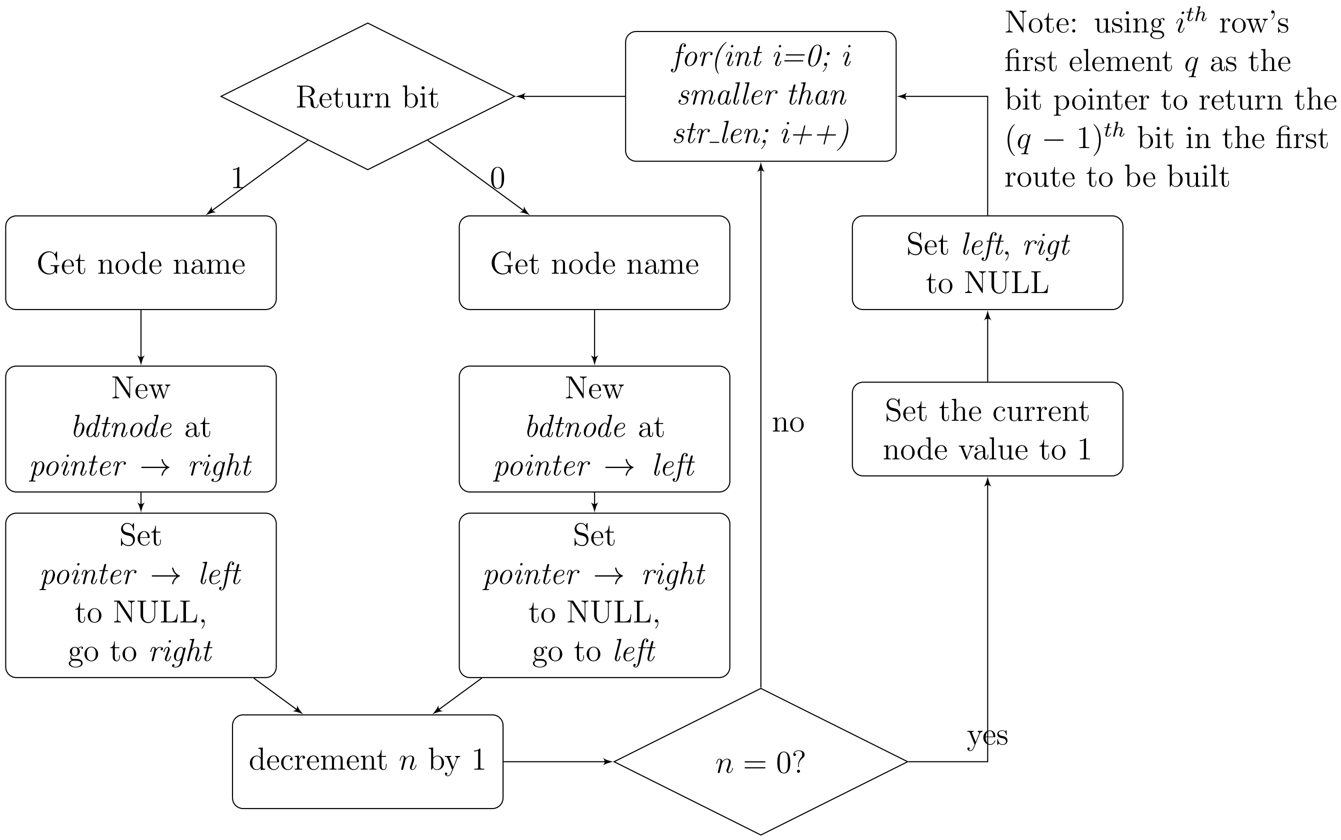


Figure 2: Flowchart of building the first route

After the first route is done, the program will move onto building other available routes. The difference between building the first route, and those that come after the first one, is the node name check. The node name, or the bit associated with the node name, will be used to guide the building process. If the bit position in the route is 1, then the pointer will point to the right of the current node and move on. If it is a 0, then the pointer will point to the left and move on. If it is a 2, the program will treat it as a 1, meaning the pointer will point to the right of the current node. A 2 indicates a *don't care case* and it does not matter which direction it will lead the pointer. However, the direction has to be consistent. It is not allowed that a 2 will lead the pointer to either the left or the right in the same tree.

In the building process, only 1s are filled at the leaf nodes. The missing 0s at the leaf nodes will be filled by calling the function *nodeSetZero()*. The root of the binary tree is returned as the output.

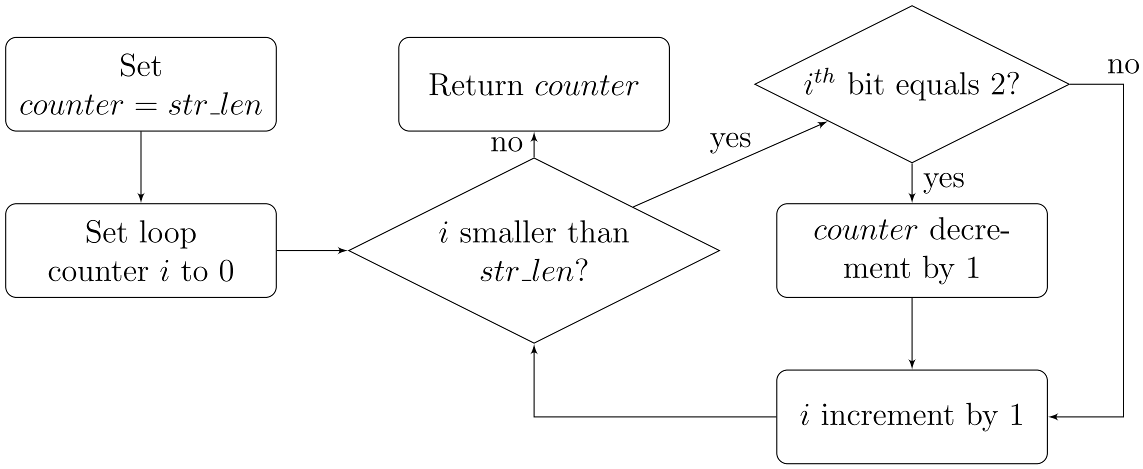


Figure 3: Flowchart of counting the number of nodes that contributes to building the tree

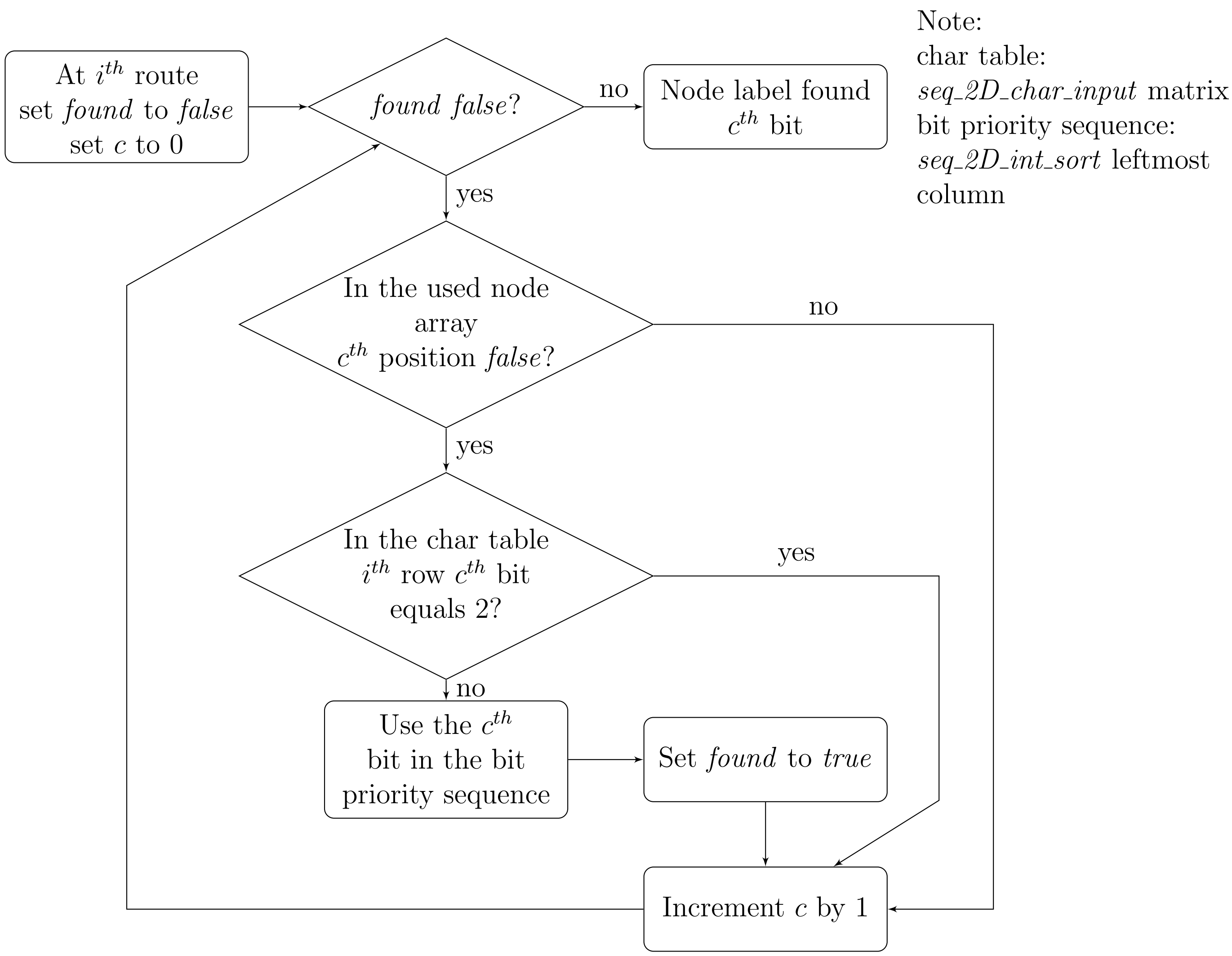


Figure 4: Flowchart of knowing the current node label, finding the next node label in building the route

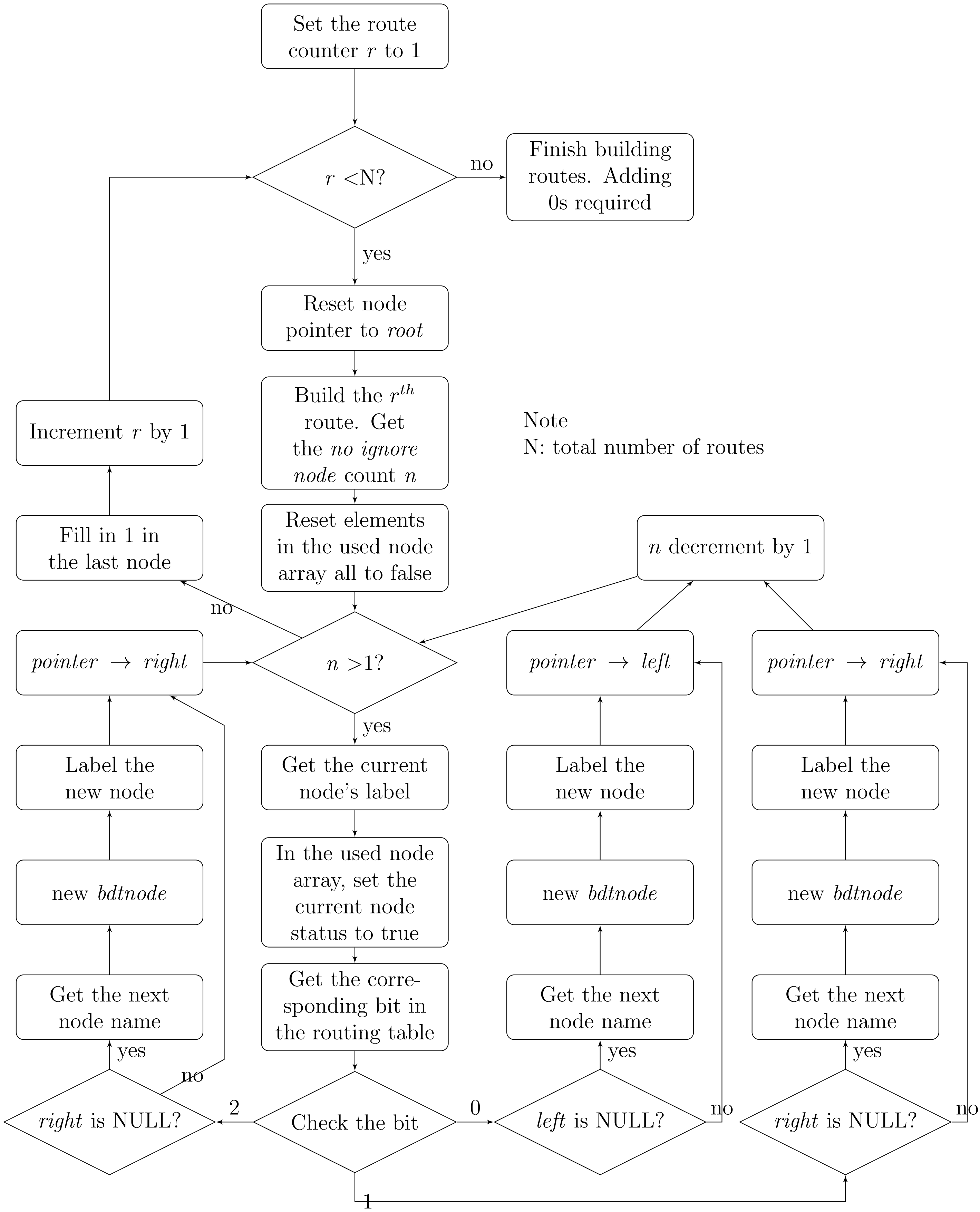


Figure 5: Flowchart of building the remaining routes

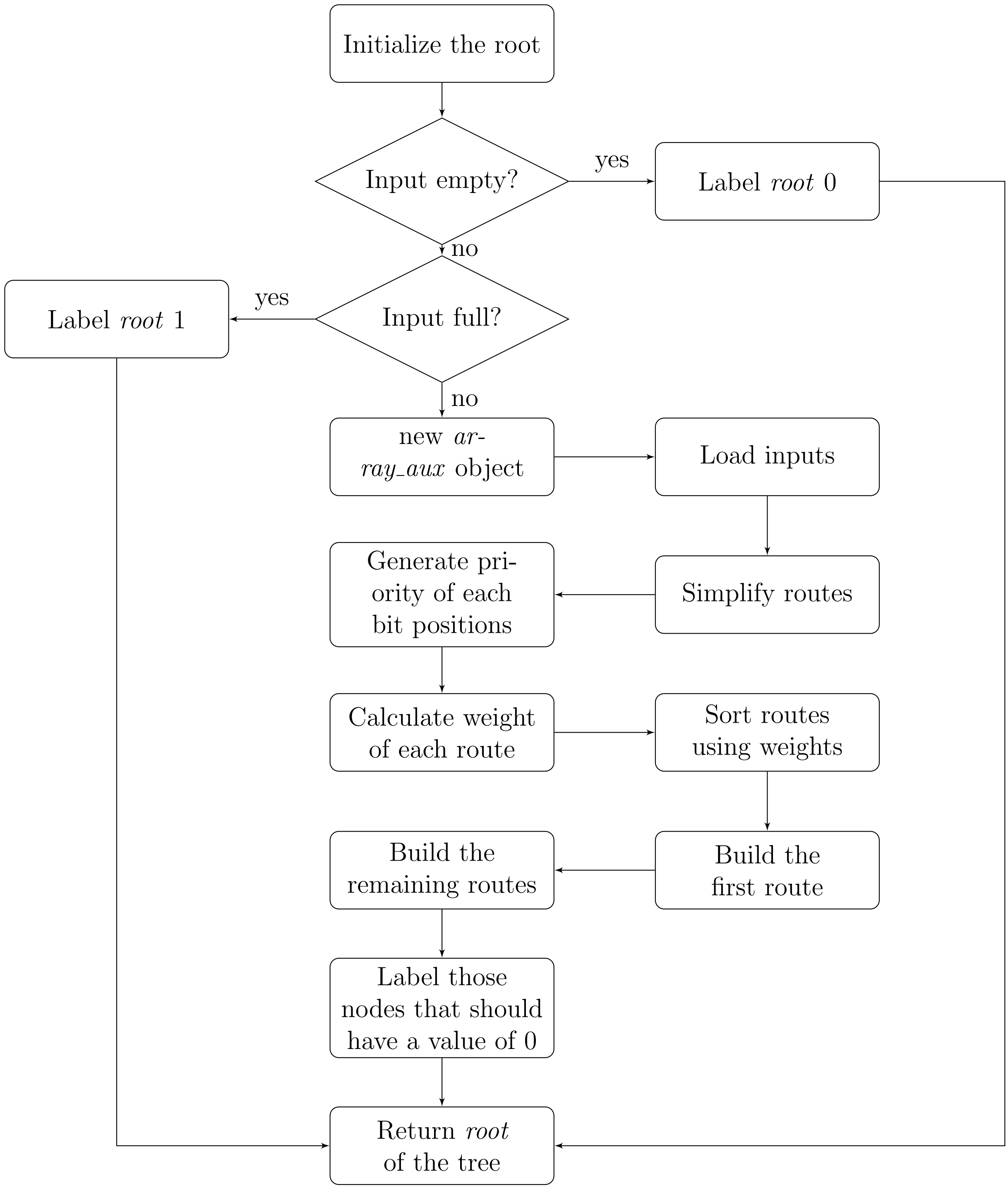
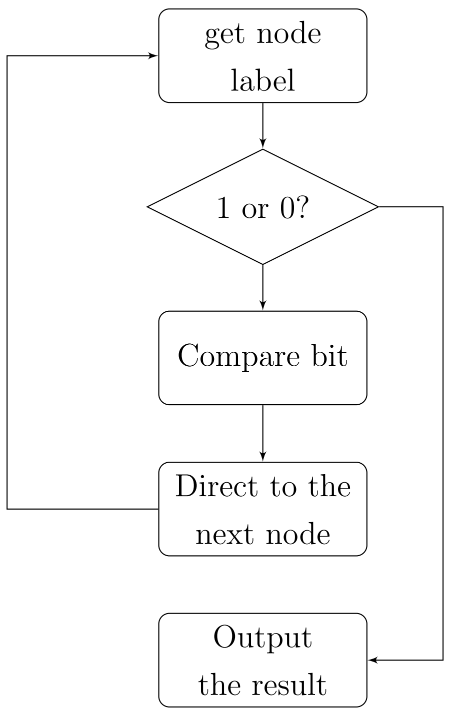


Figure 6: Flowchart of the function *buildcompactbdt*

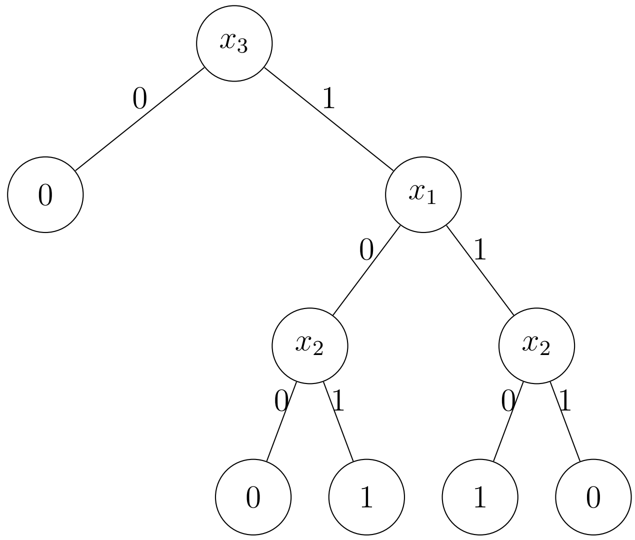
Part III

evalcompactbdt

The function *evalcompactbdt* will start from the root of the tree. In each iteration before getting a 1 or a 0, the function will get the number in the node name after the character x, using the *substr* method in the string library. There is a difference of 1 between the integer returned from trimming the node name and the bit it points to. Hence the returned integer number has to be deducted by 1. The integer will then be used as a pointer to find the corresponding value in the input string. If at that bit it is a 1, then move to the right. Else if it is a 0, then move to the left. The iteration is controlled using a while loop. When a 1 or a 0 is reached, the boolean variable that controls the while loop will be set to true and the loop will be terminated. The loop will be running if the boolean variable is false. The final value at the node will be returned as the output.



During the building process of the tree, the nodes are not always arranged in order. In other words, the nodes on the top does not always start from x1. The order is random while the rule of finding the output is fixed. The label of each node can lead to a bit in the input string. Hence a simple look up and check algorithm is implemented here.



An example here will be to use the binary string 101 and the above tree for evaluation. The label on the root of the tree is x3. The third bit, counting from the left, the character is 1. 1 indicates going to the right of the tree. The next node is labeled as x1. The first bit in the input string is 1. Hence the next step is to move to the right of the node. The next node is x2. Checking the second bit in the string outputs 0. Hence move to the left of the x2 node. A final value is shown here, which is 1. Therefore, using the binary string 101 and the above tree yield an output of 1.

Part IV

Other Functions

Function	Description
int binaryToInt(const std::string num)	convert a binary to an decimal integer
std::string nodeNamer(const int level_label)	output a string with x appended before the integer
void nodeList(bdt t)	list all the nodes in the tree
void nodeSetZero(bdt t)	set the 0 values in the simplified tree
int nodeCounter(bdt t)	count the number of leaf nodes
void nodeCountAux(bdt t, int& count)	auxiliary function in counting nodes that traverses the tree recursively

Table 12: Other functions outside the struct array_aux

1 binaryToInt

The algorithm is to calculate the weight of each bit in the binary string and sum the weights up to give the output. The function will iterate through all the bits in the binary string, starting from the right most bit. An int variable is initialized to be 0 and used to hold the result as the iteration continues.

	bit n	...	bit 2	bit 1	bit 0
weight	2^n	...	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$

The calculated result will be to use the bit at the corresponding position to multiply the weight at that bit position. And then add up all the weights to obtain the final result.

An example here will be to calculate the decimal value of the binary string 1011.

	bit 3	bit 2	bit 1	bit 0
weight	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$

The result will be $1{*}8{+}0{*}4{+}1{*}2{+}1{*}1 = 11$.

2 nodeNamer

The function is to add an x in front of the input integer and output the string. It initializes a stringstream variable and inputs the x followed by the integer. The stringstream variable is returned after calling the *str()* method.

3 nodeList

The function is recursive. Its main body is an If statement to determine whether the passed in pointer points to a NULL position or not. If the pointer does not points to a NULL position, the next statement will be to call itself using the pointer pointing to the right of the current node. Then a print statement is used to show the label on the current node. The last statement in the If statement block is to call the *nodeList* function using the pointer pointing to the left of the current node. The function is used to display all the nodes in the binary tree, from the rightmost to the leftmost.

4 nodeSetZero

The function is used in building the tree. As discussed preciously, the *buildcompactbdt* function will only fill in the 1s at the end of building each route. A lot of positions in the binary tree that remain as NULL while actually should be 0. The function *nodeSetZero* is used to change those locations to 0. The locations that should be labeled as 0 should possess the following properties. Its parent node's label starts with x. In other words, its parent is a non-leaf node in the tree. The information on the node is NULL. (Refer to Figure 1)

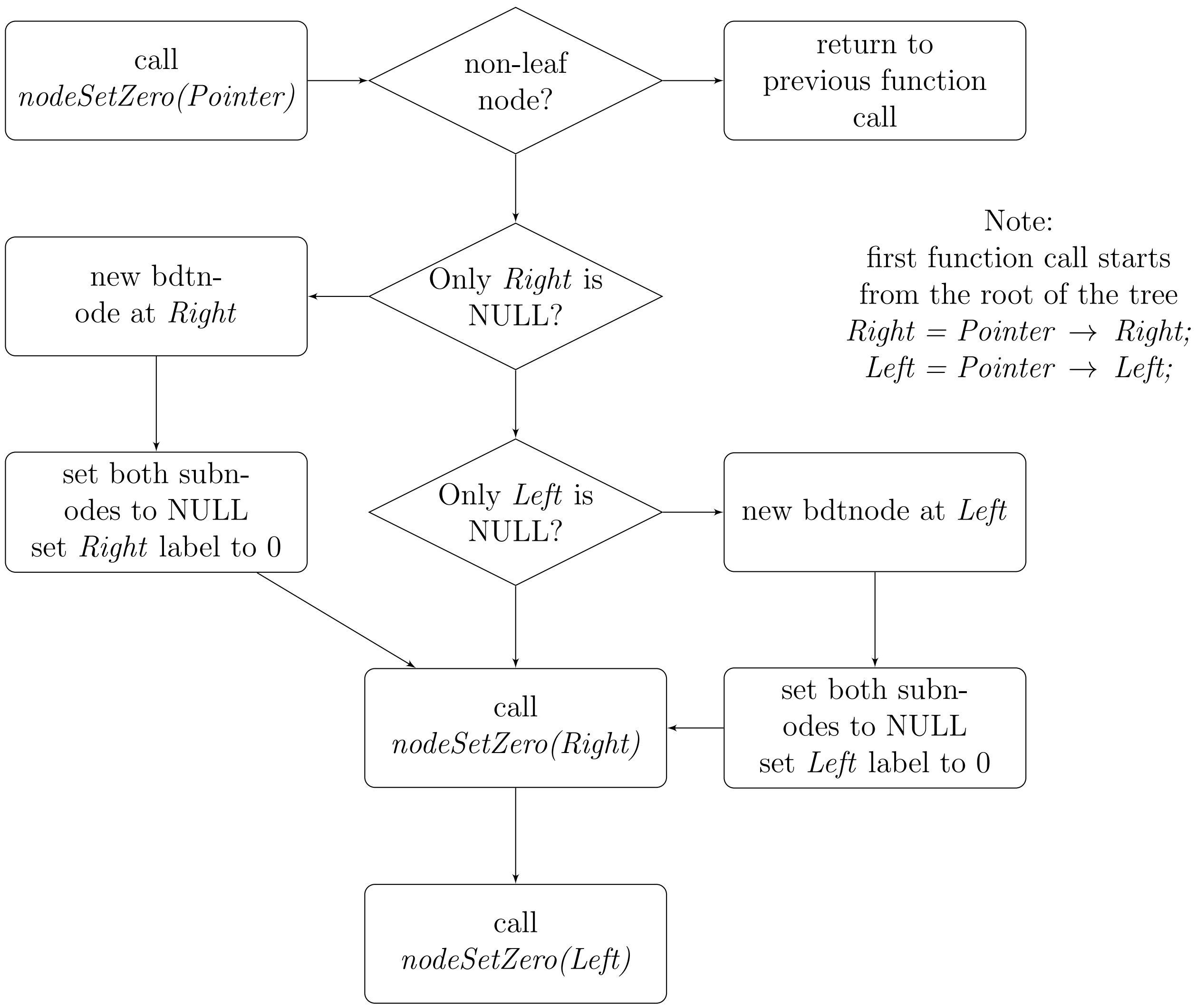


Figure 7: Flowchart for *nodeSetZero()*

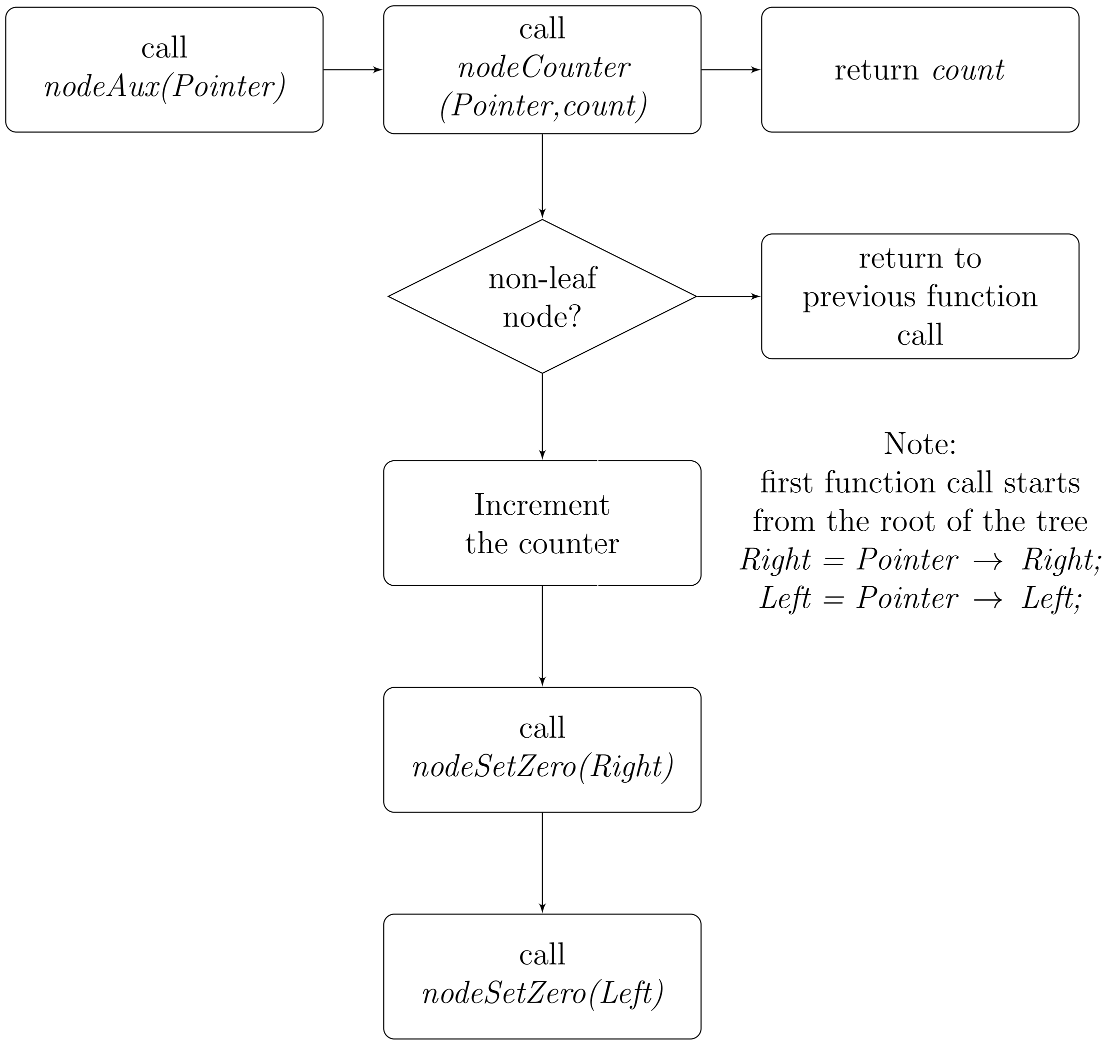


Figure 8: Flowchart for *nodeCounter()* and *nodeCountAux()*

5 nodeCounter and nodeCountAux

Those two functions are the last ones in the submitted code. The function *nodeCounter()* takes the root of a binary tree as the argument. Inside the implementation, it will initialize a int variable *count* to 0. It will then call another recursive function *nodeCountAux()* and pass in the root of the binary tree and the *count* int variable as the arguments.

Inside the implementation of the function *nodeCountAux()*, the function will only be counting the non-leaf nodes. In order to determine whether the node is non-leaf node, it will extract the first character of the label on the node. If the extracted character is x then the node is a non-leaf node. Once the non-leaf node is detected, the function *nodeCountAux()* will increment the passed in variable. Since the counter is passed in by reference, the effect in each recursive call will have effect on the variable initialized in the function *nodeCountAux()*.

The function *nodeCountAux()* returns the int variable *count*. (Refer to Figure 2)

Part V

Testing

To ensure at every stage of execution the program work correctly, the parts in the code that are labeled as uncomment to test are uncommented and extra outputs are produced other than normal evaluation outputs.

The folloiwng functions are called in order to producing the extra outputs:

seq_2D_char_input_test()

seq_2D_int_sort_test()

seq_2D_char_input_test()

A For loop is also used to test the matrix used for sorting routes using the weights.

The program is tested using the lab computer running Ubuntu. The computer is equiped with a i7 6700 CPU with 16GB of RAM. The maximum length of the binary string input is 30 characters. When I was testing 31-character input, I have to change the int arrays in the struct *array_aux* to long arrays. By doing so, the maximum length of the input strings is 31 characters long. Program uses about 4GB of the RAM while using the 30-bit binary string input with the arrays in the *array_aux* declared as int arrays. With a single 30-character long binary string input, the lab computer takes about 14 seconds to build the tree.

All of the following tests yield the correct results.

$x1$	$x2$	Output
0	0	0
0	1	1
1	0	0
1	1	1

set 1, testing 2 bits

testing input array before simplifying

testing the 2D char input array

1 1

0 1

after simplification

testing the 2D int sort array

2 0 1 1 1 1

1 0 0 0 0 2

testing the 2D char input array

2 2

2 1

testing the available route array

1 1

00 the result is: 0

01 the result is: 1

10 the result is: 0

11 the result is: 1

traversing the tree

root is: x2

1

x2

0

the number of non-leaf nodes is: 1

$x1$	$x2$	Output
0	0	0
0	1	1
1	0	1
1	1	1

set 2, testing 2 bits

testing input array before simplifying

testing the 2D char input array

1 0

1 1

0 1

after simplification

testing the 2D int sort array

2 1 1 1 2 1

1 0 1 1 1 2

testing the 2D char input array

1 0

2 2

2 1

testing the available route array

1 1

2 3

00 the result is: 0

01 the result is: 1

10 the result is: 1

11 the result is: 1

traversing the tree

root is: x2

1

x2

1

x1

0

the number of non-leaf nodes is: 2

$x1$	$x2$	$x3$	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

set 3, testing 3 bits

testing input array before simplifying

testing the 2D char input array

1 0 1

0 1 1

after simplification

testing the 2D int sort array

3 0 2 2 2 1

1 1 1 1 2 2

2 1 1 1 2 2

testing the 2D char input array

1 0 1

0 1 1

testing the available route array

3 5

5 5

000 the result is: 0

001 the result is: 0

010 the result is: 0

011 the result is: 1

100 the result is: 0

101 the result is: 1

110 the result is: 0

111 the result is: 0

traversing the tree

root is: x3

0

x2

1

x1

1

x2
0
x3
0
the number of non-leaf nodes is: 4

$x1$	$x2$	$x3$	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

set 4, testing 3 bits
testing input array before simplifying
testing the 2D char input array
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 1 0
1 1 1
after simplification
testing the 2D int sort array
3 1 2 2 3 1
1 1 1 1 2 2
2 0 1 1 1 3
testing the 2D char input array
2 2 0
0 2 1
2 2 2
2 2 2
2 2 2
2 2 2
1 1 1
testing the available route array
0 1
1 3
7 6
000 the result is: 1
001 the result is: 1
010 the result is: 1
011 the result is: 1

100 the result is: 1
101 the result is: 0
110 the result is: 1
111 the result is: 1
traversing the tree
root is: x3
1
x2
0
x1
1
x3
1
the number of non-leaf nodes is: 3

$x1$	$x2$	$x3$	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

set 5, testing 3 bits
testing input array before simplifying
testing the 2D char input array
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
000 the result is: 1
001 the result is: 1
010 the result is: 1
011 the result is: 1
100 the result is: 1
101 the result is: 1
110 the result is: 1
111 the result is: 1
traversing the tree root is: 1 1
the number of non-leaf nodes is: 0

$x1$	$x2$	$x3$	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

set 6, testing 3 bits

000 the result is: 0

001 the result is: 0

010 the result is: 0

011 the result is: 0

100 the result is: 0

101 the result is: 0

110 the result is: 0

111 the result is: 0

traversing the tree

root is: 0

0

the number of non-leaf nodes is: 0

$x1$	$x2$	$x3$	$x4$	Output
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	1

set 7, testing 4 bits

testing input array before simplifying

testing the 2D char input array

0 1 0 0

0 0 0 1

1 1 1 1

1 1 0 0

0 1 1 1

0 1 0 1

1 1 1 0

0 0 1 1

1 0 1 1

after simplification

testing the 2D int sort array

3 2 2 2 4 1

4 2 2 2 4 1

2 0 2 2 2 2

1 1 1 1 2 3

testing the 2D char input array

2 1 0 0

0 2 0 1

2 2 2 2

2 2 2 2

2 2 2 2
2 2 2 2
1 1 1 0
2 2 1 1
2 2 2 2
testing the available route array
3 2
4 4
1 5
14 7
0000 the result is: 0
0001 the result is: 1
0010 the result is: 0
0011 the result is: 1
0100 the result is: 1
0101 the result is: 1
0110 the result is: 0
0111 the result is: 1
1000 the result is: 0
1001 the result is: 0
1010 the result is: 0
1011 the result is: 1
1100 the result is: 1
1101 the result is: 0
1110 the result is: 1
1111 the result is: 1
traversing the tree
root is: x3
1
x4
1
x1
0
x2
0
x3
0
x1
1
x4
1
x2
0
the number of non-leaf nodes is: 7

$x1$	$x2$	$x3$	$x4$	Output
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

set 8, testing 4 bits
testing input array before simplifying
testing the 2D char input array
0 1 0 0
0 0 0 1
1 1 1 1
1 1 0 0
0 1 1 1
after simplification
testing the 2D int sort array
2 1 2 2 3 1
3 2 1 2 3 1
4 1 2 2 3 1
1 1 0 1 1 2
testing the 2D char input array
2 1 0 0
0 0 0 1
2 2 2 2
2 2 2 2
2 1 1 1
testing the available route array
4 3
7 3
1 5
0000 the result is: 0
0001 the result is: 1
0010 the result is: 0
0011 the result is: 0
0100 the result is: 1
0101 the result is: 0
0110 the result is: 0
0111 the result is: 1
1000 the result is: 0
1001 the result is: 0
1010 the result is: 0
1011 the result is: 0
1100 the result is: 1
1101 the result is: 0
1110 the result is: 0
1111 the result is: 1
traversing the tree
root is: x3
1
x4
1
x1
0
x2
0
x3
0
x1
1
x4
1
x2
0
the number of non-leaf nodes is: 7

set 9, testing 30 bits

testing input array before simplifying

testing the 2D char input array

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0

after simplification

testing the 2D int sort array

1 1 0 1 1 1

2 1 0 1 1 1

3 1 0 1 1 1

4 1 0 1 1 1

5 1 0 1 1 1

6 1 0 1 1 1

7 1 0 1 1 1

8 1 0 1 1 1

9 1 0 1 1 1

10 1 0 1 1 1

11 1 0 1 1 1

12 1 0 1 1 1

13 1 0 1 1 1

14 1 0 1 1 1

15 1 0 1 1 1

16 1 0 1 1 1

17 1 0 1 1 1

18 1 0 1 1 1

19 1 0 1 1 1

20 1 0 1 1 1

21 1 0 1 1 1

22 1 0 1 1 1

23 1 0 1 1 1

24 1 0 1 1 1

25 1 0 1 1 1

26 1 0 1 1 1

27 1 0 1 1 1

28 1 0 1 1 1

29 1 0 1 1 1

30 1 0 1 1 1

testing the 2D char input array

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0

testing the available route array

0 30

evaluating 000000000000000000000000000000

traversing the tree

root is: x1

0

x1

0

x2

0

x3

0

x4

0

x5

0

x6

0

x7

0

x8

0

x9

0

x10

0

x11

0

x12

0

x13

0

x14

0

x15

0

x16

0

x17

0

x18

0

x19

0

x20

0

x21

0

x22

0

x23

0

x24

0

x25

0

x26

0

x27

0

x28

0

x29

0

x30

1

the number of non-leaf nodes is: 30

Part VI

Code

```
/*
* Run.cpp
*
*  Created on: 15 April 2018
*      Author: lw2016
*/

#include<string>
#include<vector>
#include<iostream>
#include<sstream>
#include<math.h>

struct bdnode {
  std::string val;
  bdnode* left;
  bdnode* right;
};

typedef bdnode* bdt;

struct array_aux {

private:

  //number of digits in each input element
  int input_string_length;

  //length of array
  int array_length_1D;

  //length of the vector inpiut
  int vector_len;

  //int 1D array
  int *seq_1D_int;
```

```

    //n*2^n
    //2D char array
    char **seq_2D_char_input;

    //for accelerating the checking process
    bool *seq_1D_2D_available_space;

    //for sorting
    //bit 0: bit position
    //bit 1: bit = 0 counter
    //bit 2: bit = 1 counter
    //bit 3: max
    //bit 4: sum
    //bit 5: weight/priority
    int **seq_2D_int_sort;

  public:

    array_aux(int len, int len_of_vector) {
      input_string_length = len;
      array_length_1D = pow(2, input_string_length);
      vector_len = len_of_vector;

      //1D
      seq_1D_int = new int[array_length_1D];

      for (int i = 0; i < array_length_1D; i++) {
        seq_1D_int[i] = -1;
      }

      //2D & 1D
      seq_2D_char_input = new char*[vector_len];
      seq_1D_2D_available_space = new bool[vector_len];

      for (int i = 0; i < vector_len; i++) {
        seq_2D_char_input[i] = new char[input_string_length];
        seq_1D_2D_available_space[i] = false;
        for (int j = 0; j < input_string_length; j++) {
          seq_2D_char_input[i][j] = '2';
```

```
            }
        }

        //2D
        seq_2D_int_sort = new int*[input_string_length];

        for (int i = 0; i < input_string_length; i++) {
            seq_2D_int_sort[i] = new int[6];
            seq_2D_int_sort[i][0] = i + 1;
            for (int j = 1; j < 6; j++) {
                seq_2D_int_sort[i][j] = 0;
            }
        }
    }

    ~array_aux() {
        if (seq_1D_int) {
            delete[] seq_1D_int;
        }

        if (seq_2D_char_input) {
            for (int i = 0; i < vector_len; i++) {
                delete[] seq_2D_char_input[i];
            }
            delete[] seq_2D_char_input;
        }

        if (seq_1D_2D_available_space) {
            delete[] seq_1D_2D_available_space;
        }

        if (seq_2D_int_sort) {
            for (int i = 0; i < input_string_length; i++) {
                delete[] seq_2D_int_sort[i];
            }
            delete[] seq_2D_int_sort;
        }
    }

    int return_array_length_1D() {
```

```
      return array_length_1D;
    }

    void seq_1D_int_set(int position, int val) {
      seq_1D_int[position] = val;
    }

    int seq_1D_int_return(int position) {
      return seq_1D_int[position];
    }

    void seq_1D_int_test() {
      for (int i = 0; i < vector_len; i++) {
        std::cout << i << ' ' << seq_1D_int[i] << std::endl;
      }
      std::cout << std::endl;
    }

    void seq_1D_2D_char_input_set(int position, std::string binary_string) {

      if (seq_1D_int[position] == -1) {
        //find the location in the 2D char table
        //for inserting the new record
        bool found = false;
        int space_index = 0;

        while (!found && space_index < vector_len) {
          if (!seq_1D_2D_available_space[space_index]) {
            found = true;
            seq_1D_2D_available_space[space_index] = true;
          }
          else {
            space_index++;
          }
        }

        //set the value on the mapping list
        seq_1D_int[position] = space_index;

        //set the values in the 2D input table
```

```
        for (int i = 0; i < binary_string.length(); i++) {
          seq_2D_char_input[space_index][i] = binary_string[i];
        }
      }
    }

    void seq_2D_char_input_set_bit(int position, int bit, char info) {

      int index_hashed = seq_1D_int[position];

      seq_2D_char_input[index_hashed][bit] = info;
    }

    char seq_2D_char_input_return(int x, int y) {

      int index_hashed = seq_1D_int[x];

      return seq_2D_char_input[index_hashed][y];
    }

    int seq_2D_char_input_weight(int position) {

      int index_hashed = seq_1D_int[position];
      int weight = 0;

      if (index_hashed != -1) {
        for (int i = 0; i<input_string_length; i++) {
          if (seq_2D_char_input[index_hashed][i] != '0') {
            weight += pow(2, input_string_length - 1 - i);
          }
        }
        return weight;
      }
      return -1;

    }

    void seq_2D_char_input_test() {

      std::cout << "testing the 2D char input array" << std::endl;
```

```

    for (int i = 0; i < vector_len; i++) {
      for (int j = 0; j < input_string_length; j++) {
        std::cout << seq_2D_char_input[i][j] << ' ';
      }
      std::cout << std::endl;
    }
    std::cout << std::endl;
  }

  int check_available_routes() {
    int available_routes_count = 0;

    for (int i = 0; i < vector_len; i++) {
      if (seq_1D_2D_available_space[i]) {
        available_routes_count++;
      }
    }
    return available_routes_count;
  }

  void delete_route(int position) {

    int index_hashed = seq_1D_int[position];

    for (int i = 0; i < input_string_length; i++) {
      seq_2D_char_input[index_hashed][i] = '2';
    }

    seq_1D_int[position] = -1;
    seq_1D_2D_available_space[index_hashed] = false;
  }

  void seq_2D_int_sort_bit_set() {

    //fill in at position 1 and 2
    for (int i = 0; i < array_length_1D; i++) {
      if (seq_1D_int[i] != -1) {
        for (int j = 0; j < input_string_length; j++) {
          if (seq_2D_char_input[seq_1D_int[i]][j] == '0') {
```

```
                seq_2D_int_sort[j][1]++;
            }
            else if (seq_2D_char_input[seq_1D_int[i]][j] == '1') {
                seq_2D_int_sort[j][2]++;
            }
        }
    }
}

//fill in at position 3 and 4
for (int i = 0; i < input_string_length; i++) {
    if (seq_2D_int_sort[i][1] <= seq_2D_int_sort[i][2]) {
        seq_2D_int_sort[i][3] = seq_2D_int_sort[i][2];
    }
    else {
        seq_2D_int_sort[i][3] = seq_2D_int_sort[i][1];
    }
    seq_2D_int_sort[i][4] = seq_2D_int_sort[i][1]
        + seq_2D_int_sort[i][2];
}
}

void seq_2D_int_sort_bit_sort(int bit_sort) {
    int swap_tmpp_0;
    int swap_tmpp_1;
    int swap_tmpp_2;
    int swap_tmpp_3;
    int swap_tmpp_4;

    //bubble sort the 2D int order array
    //using the sum
    for (int i = 0; i < input_string_length - 1; i++) {
        for (int j = 0; j < input_string_length - 1; j++) {
            if (seq_2D_int_sort[j][bit_sort]
                < seq_2D_int_sort[j + 1][bit_sort]) {
                swap_tmpp_0 = seq_2D_int_sort[j + 1][0];
                swap_tmpp_1 = seq_2D_int_sort[j + 1][1];
                swap_tmpp_2 = seq_2D_int_sort[j + 1][2];
                swap_tmpp_3 = seq_2D_int_sort[j + 1][3];
                swap_tmpp_4 = seq_2D_int_sort[j + 1][4];
```

```
          seq_2D_int_sort[j + 1][0] = seq_2D_int_sort[j][0];
          seq_2D_int_sort[j + 1][1] = seq_2D_int_sort[j][1];
          seq_2D_int_sort[j + 1][2] = seq_2D_int_sort[j][2];
          seq_2D_int_sort[j + 1][3] = seq_2D_int_sort[j][3];
          seq_2D_int_sort[j + 1][4] = seq_2D_int_sort[j][4];

          seq_2D_int_sort[j][0] = swap_tmpp_0;
          seq_2D_int_sort[j][1] = swap_tmpp_1;
          seq_2D_int_sort[j][2] = swap_tmpp_2;
          seq_2D_int_sort[j][3] = swap_tmpp_3;
          seq_2D_int_sort[j][4] = swap_tmpp_4;
        }
      }
    }
  }

  //sort: sum first, max after
  void seq_2D_int_sort_bit_run() {
    seq_2D_int_sort_bit_sort(4);
    seq_2D_int_sort_bit_sort(3);
  }

  int seq_2D_int_sort_return(int x, int y) {
    return seq_2D_int_sort[x][y];
  }

  void seq_2D_int_sort_test() {

    std::cout << "testing the 2D int sort array" << std::endl;

    for (int i = 0; i < input_string_length; i++) {
      for (int j = 0; j < 6; j++) {
        std::cout << seq_2D_int_sort[i][j] << ' ';
      }
      std::cout << std::endl;
    }
    std::cout << std::endl;
  }

```

```
    void seq_2D_int_sort_priority_gen() {
      int priority_level = 1;

      seq_2D_int_sort[0][5] = priority_level;

      for (int i = 1; i < input_string_length; i++) {
        if (seq_2D_int_sort[i][3] == seq_2D_int_sort[i - 1][3]
          && seq_2D_int_sort[i][4] == seq_2D_int_sort[i - 1][4]) {
          seq_2D_int_sort[i][5] = priority_level;
        }
        else {
          priority_level++;
          seq_2D_int_sort[i][5] = priority_level;
        }
      }
    }
  };

  bdt buildcompactbdt(const std::vector<std::string>& fvalues);

  std::string evalcompactbdt(bdt t, const std::string& input);

  int binaryToInt(const std::string num);

  std::string nodeNamer(const int level_label);

  void nodeList(bdt t);

  void nodeSetZero(bdt t);

  int nodeCounter(bdt t);

  void nodeCountAux(bdt t, int &count);

  bdt buildcompactbdt(const std::vector<std::string>& fvalues) {

    bdt bdt_array_root = new bdnode;

    if (fvalues.size() == 0) {
      bdt_array_root->val = "0";
```

```
        bdt_array_root->right = NULL;
        bdt_array_root->left = NULL;
        return bdt_array_root;
      }

      //get the length of the input string
      int string_length = fvalues[0].length();

      int input_length = fvalues.size();

      array_aux *array_gen = new array_aux(string_length, input_length);

      //load values
      for (int i = 0; i < input_length; i++) {
        array_gen
        ->seq_1D_2D_char_input_set(binaryToInt(fvalues[i]), fvalues[i]);
      }

      //uncomment the following for testing
      //  std::cout << "testing input array before simplifying" << std::endl;
      //  array_gen->seq_2D_char_input_test();

      if (array_gen->check_available_routes()
      == array_gen->return_array_length_1D()) {
        bdt_array_root->val = "1";
        bdt_array_root->right = NULL;
        bdt_array_root->left = NULL;
        return bdt_array_root;
      }

      //simplify the routes
      int count_before = 0;
      int count_after = 0;
      bool match_found = false;
      int distance = 0;
      int bit_position = 0;
      int length_1D = array_gen->return_array_length_1D();
      int relative_position;
      int weight_i;
      int weight_relative;
```

```
    int difference;

    do {
      //check the termination condition, before simplification
      count_before = array_gen->check_available_routes();

      for (int i = 0; i < length_1D; i++) {
        if (array_gen->seq_1D_int_return(i) != -1) {

          match_found = false;
          bit_position = 0;

          while (bit_position < string_length && match_found == false) {

            if (array_gen->seq_2D_char_input_return(i, bit_position) == '0')

              distance = pow(2, (string_length - bit_position - 1));
              relative_position = i + distance;
              weight_i = array_gen->seq_2D_char_input_weight(i);
              weight_relative = array_gen
              ->seq_2D_char_input_weight(relative_position);
              difference = weight_relative - weight_i;

              if (array_gen->seq_1D_int_return(relative_position) != -1
                && difference == distance) {
                array_gen->delete_route(relative_position);
                array_gen->seq_2D_char_input_set_bit(i, bit_position, '2');
                match_found = true;
              }
            }
            bit_position++;
          }
        }
      }

      //check the termination condition, after simplification
      count_after = array_gen->check_available_routes();

    } while (count_before != count_after);

```

```
    array_gen->seq_2D_int_sort_bit_set();

    //sort: sum first, max after
    array_gen->seq_2D_int_sort_bit_run();

    array_gen->seq_2D_int_sort_priority_gen();

    //uncomment the following for testing
    //std::cout << "after simplification" << std::endl;
    //array_gen->seq_2D_int_sort_test();
    //array_gen->seq_2D_char_input_test();

    //create a table
    //record the available routes
    //calculate the corresponding weighting
    int **available_route = new int*[array_gen->check_available_routes()];

    for (int i = 0; i < array_gen->check_available_routes(); i++) {
      available_route[i] = new int[2];
      available_route[i][0] = 0;
      available_route[i][1] = 0;
    }

    int weighting = 0;
    int available_route_pointer = 0;

    for (int i = 0; i < array_gen->return_array_length_1D(); i++) {

      if (array_gen->seq_1D_int_return(i) != -1) {

        available_route[available_route_pointer][0] = i;
        bit_position = 0;
        weighting = 0;

        while (bit_position < string_length) {

          if (array_gen->seq_2D_char_input_return(i,
            array_gen->seq_2D_int_sort_return(bit_position, 0) - 1)
            != '2') {
            weighting += array_gen->seq_2D_int_sort_return(bit_position,
```

```
                    5);
              }
              bit_position++;
            }
            available_route[available_route_pointer][1] = weighting;
            available_route_pointer++;
          }
        }

        int swap_tmpp_0;
        int swap_tmpp_1;

        //bubble sort the route array
        //small first
        for (int i = 0; i < array_gen->check_available_routes() - 1; i++) {
          for (int j = 0; j < array_gen->check_available_routes() - 1; j++) {
            if (available_route[j][1] > available_route[j + 1][1]) {
              swap_tmpp_0 = available_route[j][0];
              swap_tmpp_1 = available_route[j][1];

              available_route[j][0] = available_route[j + 1][0];
              available_route[j][1] = available_route[j + 1][1];

              available_route[j + 1][0] = swap_tmpp_0;
              available_route[j + 1][1] = swap_tmpp_1;
            }
          }
        }

        //  testing
        //  std::cout << "testing the available route array" << std::endl;
        //  for (int i = 0; i < array_gen->check_available_routes(); i++) {
        //    std::cout << available_route[i][0] << ' ' << available_route[i][1]
        //                                    << std::endl;
        //  }
        //  std::cout << std::endl;

        //start building the tree
        bdt bdt_tmpp = bdt_array_root;
        int level_label = 0;
```

```
    int no_ignore_node_count = string_length;

    //set the node counter in the router
    for (int i = 0; i < string_length; i++) {
      if (array_gen
      ->seq_2D_char_input_return(available_route[0][0], i)
      == '2') {
        no_ignore_node_count--;
      }
    }

    //build the first route
    for (int i = 0; i < string_length; i++) {
      if (array_gen->seq_2D_char_input_return(available_route[0][0],
        array_gen->seq_2D_int_sort_return(i, 0) - 1) == '1') {
        level_label = array_gen->seq_2D_int_sort_return(i, 0);
        bdt_tmpp->val = nodeNamer(level_label);
        bdt_tmpp->right = new bdnode;
        bdt_tmpp->left = NULL;
        bdt_tmpp = bdt_tmpp->right;
        no_ignore_node_count--;
      }
      else if (array_gen->seq_2D_char_input_return(available_route[0][0],
        array_gen->seq_2D_int_sort_return(i, 0) - 1) == '0') {
        level_label = array_gen->seq_2D_int_sort_return(i, 0);
        bdt_tmpp->val = nodeNamer(level_label);
        bdt_tmpp->right = NULL;
        bdt_tmpp->left = new bdnode;
        bdt_tmpp = bdt_tmpp->left;
        no_ignore_node_count--;
      }
      if (no_ignore_node_count == 0) {
        bdt_tmpp->val = "1";
        bdt_tmpp->right = NULL;
        bdt_tmpp->left = NULL;
      }
    }

    //build the other routes
    //start from the root
```

```
    //going along the existing route
    //compare the current node name with the node required in the route
    //if necessary, build extra nodes with the correct node names
    int route_pointer = 1; //skip the first route, which is built
    std::string current_node_string;
    int current_node_int;
    char retrieved_bit_value;
    std::string new_node_name;
    bool name_found;
    int name_position_counter;

    //array for storing the used nodes
    bool *used_node = new bool[string_length];

    while (route_pointer < array_gen->check_available_routes()) {

      //reset the pointer
      //pointing the root
      bdt_tmpp = bdt_array_root;

      //reset the node counter in the route
      no_ignore_node_count = string_length;

      for (int i = 0; i < string_length; i++) {
        if (array_gen->seq_2D_char_input_return(
          available_route[route_pointer][0],
          i) == '2') {
          no_ignore_node_count--;
        }
      }

      //reset the used node array
      for (int i = 0; i<string_length; i++) {
        used_node[i] = false;
      }

      //keep building the tree
      //until reaching the last element in the route
      while (no_ignore_node_count > 1) {

```

```
      //get the current node/bit position/node name
      current_node_string = bdt_tmpp->val.substr(1,
        (bdt_tmpp->val.length() - 1));
      std::stringstream ss1;
      ss1 << current_node_string;
      ss1 >> current_node_int;

      //set the current node
      //used state to true
      for (int i = 0; i<string_length; i++) {
        if (array_gen
        ->seq_2D_int_sort_return(i, 0) == current_node_int) {
          used_node[i] = true;
        }
      }

      //retrieve the bit from the array
      //that stores the routing information
      retrieved_bit_value = array_gen->seq_2D_char_input_return(
        available_route[route_pointer][0], current_node_int - 1);

      //route the building sequence using the retrieved bit
      //add extra node if the pointer is pointing to a NULL position
      //skip adding new node if the node already exists
      if (retrieved_bit_value == '0') {
        if (bdt_tmpp->left == NULL) {

          //get the new node name
          name_found = false;
          name_position_counter = 0;
          while (name_found == false) {
            if (used_node[name_position_counter] == false && array_gen
            ->seq_2D_char_input_return(available_route[route_pointer][0],
            array_gen
            ->seq_2D_int_sort_return(name_position_counter, 0) - 1)
             != '2') {

              new_node_name = nodeNamer(
              array_gen
              ->seq_2D_int_sort_return(name_position_counter, 0));
```

```
              name_found = true;
            }
            name_position_counter++;
          }

          //create the node
          bdt_tmpp->left = new bdnode;

          //name the new node
          bdt_tmpp->left->val = new_node_name;

          //set the left and the right of the new node to NULL
          bdt_tmpp->left->right = NULL;
          bdt_tmpp->left->left = NULL;
        }
        //update the pointer
        bdt_tmpp = bdt_tmpp->left;
        no_ignore_node_count--;
      }
      else if (retrieved_bit_value == '1') {
        if (bdt_tmpp->right == NULL) {

          //get the new node name
          name_found = false;
          name_position_counter = 0;
          while (name_found == false) {
            if (used_node[name_position_counter] == false
            && array_gen
            ->seq_2D_char_input_return(available_route[route_pointer][0],
            array_gen
            ->seq_2D_int_sort_return(name_position_counter, 0) - 1) != '2')

              new_node_name = nodeNamer
              (array_gen->seq_2D_int_sort_return(name_position_counter, 0))
              name_found = true;
            }
            name_position_counter++;
          }

          //create the node
```

```
          bdt_tmpp->right = new bdnode;

          //name the new node
          bdt_tmpp->right->val = new_node_name;

          //set the left and the right of the new node to NULL
          bdt_tmpp->right->right = NULL;
          bdt_tmpp->right->left = NULL;
        }
        //update the pointer
        bdt_tmpp = bdt_tmpp->right;
        no_ignore_node_count--;
      }
      //retrieved bit = 2
      //route to the right
      //the same as retrieved bit = 1
      else {
        if (bdt_tmpp->right == NULL) {

          //get the new node name
          //get the new node name
          name_found = false;
          name_position_counter = 0;
          while (name_found == false) {
            if (used_node[name_position_counter] == false &&
            array_gen
            ->seq_2D_char_input_return(available_route[route_pointer][0],
            array_gen->
            seq_2D_int_sort_return(name_position_counter, 0) - 1)
            != '2') {

              new_node_name = nodeNamer
              (array_gen
              ->seq_2D_int_sort_return(name_position_counter, 0));
              name_found = true;
            }
            name_position_counter++;
          }

          //create the node
```

```
            bdt_tmpp->right = new bdnode;

            //name the new node
            bdt_tmpp->right->val = new_node_name;

            //set the left and the right of the new node to NULL
            bdt_tmpp->right->right = NULL;
            bdt_tmpp->right->left = NULL;
          }
          //update the pointer
          bdt_tmpp = bdt_tmpp->right;
        }
      }

      //fill in the value of the last element in the route
      current_node_string =
      bdt_tmpp->val.substr(1, (bdt_tmpp->val.length() - 1));
      std::stringstream ss;
      ss << current_node_string;
      ss >> current_node_int;

      retrieved_bit_value = array_gen->seq_2D_char_input_return(
        available_route[route_pointer][0], current_node_int - 1);

      if (retrieved_bit_value == '1') {
        bdt_tmpp->right = new bdnode;
        bdt_tmpp->right->val = "1";
        bdt_tmpp->right->right = NULL;
        bdt_tmpp->right->left = NULL;
      }
      else {
        bdt_tmpp->left = new bdnode;
        bdt_tmpp->left->val = "1";
        bdt_tmpp->left->right = NULL;
        bdt_tmpp->left->left = NULL;
      }
      route_pointer++;
    }

    nodeSetZero(bdt_array_root);
```

```
    delete array_gen;

    delete available_route;

    delete used_node;

    return bdt_array_root;
}

std::string evalcompactbdt(bdt t, const std::string& input) {

    bool found = false;
    std::string text;
    int number;

    if (t->val[0] != 'x') {
        return t->val;
    }

    while (found == false) {

        text = t->val;
        text = text.substr(1, (text.length() - 1));
        std::stringstream ss;
        ss << text;
        ss >> number;

        if (input[number - 1] == '1') {
            t = t->right;
            if (t->val == "1" || t->val == "0") {
                found = true;
            }
        }
        else if (input[number - 1] == '0') {
            t = t->left;
            if (t->val == "1" || t->val == "0") {
                found = true;
            }
        }
```

```
  }
  return t->val;
}

int binaryToInt(const std::string num) {

  int decimal = 0;
  int length = num.length();

  for (int i = 0; i < length; i++) {
    if (num[length - i - 1] != '0') {
      decimal = decimal + pow(2, i);
    }
  }
  return decimal;
}

std::string nodeNamer(const int level_label) {

  std::stringstream ss;
  ss << "x" << level_label;
  return ss.str();
}

void nodeList(bdt t) {

  if (t != NULL) {
    nodeList(t->right);
    std::cout << t->val << std::endl;
    nodeList(t->left);
  }
}

void nodeSetZero(bdt t) {

  if (t->val[0] == 'x') {
    if (t->right == NULL && t->left != NULL) {
      t->right = new bdnode;
      t->right->val = "0";
      t->right->right = NULL;
```

```
      t->right->left = NULL;
    }
    if (t->left == NULL && t->right != NULL) {
      t->left = new bdnode;
      t->left->val = "0";
      t->left->right = NULL;
      t->left->left = NULL;
    }
    nodeSetZero(t->right);
    nodeSetZero(t->left);
  }
}

int nodeCounter(bdt t) {
  int count = 0;
  nodeCountAux(t, count);
  return count;
}

void nodeCountAux(bdt t, int &count) {
  if (t->val[0] == 'x') {
    count++;
    nodeCountAux(t->right, count);
    nodeCountAux(t->left, count);
  }
}
```

References