

# ASSIGNMENT 2 REPORT

by UMEIGBO, Kosidinna Kingsley  
CID: 01201479

## ABSTRACT

This report contains all aspects considered for the Coursework Assignment 2. It first introduces the various function created and implemented and then goes on to describe what each function does and the algorithm behind each. The report then goes on to cover all experimental testing and evaluation done to ensure the program works fine.

Due to the fact that my Assignment 1 wasn't the best, this program is a combination of both assignments because the functions from the previous assignment were needed for this one to work properly.

## INTRODUCTION AND DESCRIPTION OF FUNCTIONS USED

The functions used in this assignment are listed below based on use:

1. **void** initial\_tree(**std::string** value, **bdt** &t, **int** i)
2. **int** node\_value\_int(**std::string** node\_value)
3. **void** fill\_tree(**bdt** &t, **std::string** number\_input)
4. **bdt** buildbdt(**const std::vector<std::string>&** fvalues)
5. **std::string** get\_eval\_value(**bdt** t, **const std::string&** number\_input)
6. **std::string** evalbdt(**bdt** t, **const std::string** &input)
7. **bool** check\_subtrees(**bdt** &t1, **bdt** &t2)
8. **bool** check\_tree(**bdt** &t)
9. **void** delete\_subtree(**bdt** &t)
10. **void** compact\_tree(**bdt** &t)
11. **bdt** buildcompactbdt(**const std::vector<std::string>&** fvalues)
12. **std::string** evalcompactbdt(**bdt** t, **const std::string&** input)

## DESCRIPTION OF FUNCTIONS USED

### 1. **void** initial\_tree(**std::string** value, **bdt** &t, **int** i)

This function takes in a created pointer **t** with value **NULL**, a string called **value** and an integer **i** which initially is zero. It is a recursive function. As long as **i** is less than the string size of **value**, a new node is formed with node value of "**x**(**i** + 1)". Then the left and right pointers are set to **NULL**. These are then recursively passed to the function, left tree first then right tree with **i** increasing by 1 each time the function is called. Therefore the root node has node value of "**x1**". When **i** gets to be the string size of value, a leaf node is then formed with node value of "**0**". This is has therefore created the initial tree we need based on the length of the input string.

### 2. **int** node\_value\_int(**std::string** node\_value)

This function takes in the value of a node called **node\_value**. The size of node\_value can either be equal to 1, in the case of a leaf node which is "**0**" or "**1**", or it can be greater than 1, in the case of a non-leaf node which has to begin with "**x**" followed by a number. This

function is only called if the size of a node value is greater than 1. What this function does is basically extract that string number and convert it to an integer value.

We first of all create a variable **string\_size** with type **int** which is the size of the node value. We then create an array of characters called **index** with size 1 less than **string\_size** because we want to get rid of the "x". A **for** loop is then created that copies the string values excluding "x" into **index** and this array is passed through a function called **atoi()** that converts this string number into a type **int** and this number, **x**, is returned.

### 3. **void fill\_tree(bdt &t, std::string number\_input)**

This function takes in input the pointer **t** to the binary tree and the input string of number, **number\_input** that makes the leaf node value to be "1". The function is a recursive function and only performs if the pointer **t** is not **NULL**. First of all if **t->val.size()** is greater than 1, it is not a leaf node and so we then go on to extract the integer value, **i** using the **node\_value\_int()** function and this forms the index used in the next part of the function. If **number\_input[i - 1]** is equal to '0', the function is recursively called with the pointer **t->left** and same input string. If **number\_input[i - 1]** is equal to '1', the function is recursively called with the pointer **t->right** and same input string.

### 4. **bdt buildbdt(const std::vector<std::string>& fvalues)**

This function takes in input a vector of strings called **fvalues** all of equal length and returns a pointer pointing to the root of the newly formed tree.

A variable **i** of type **int** is initially created to be equal to be zero. A pointer **t** of type **bdt** is initially created to be equal to **NULL**. Then **fvalues[i]**, **t**, and **i** form inputs to the function **initial\_tree()** to create the initial tree needed. Then we go on to create a **for** loop which starts with **int a** equal to zero, with condition **a** less than size of **fvalues** and normal increment by 1 each iteration. The code in the **for** loop is just the function **fill\_tree()** which takes in input the pointer **t** and **fvalues[a]** for each iteration until all items in the vector are considered.

### 5. **std::string get\_eval\_value(bdt t, const std::string& number\_input)**

This function has a similar algorithm and structure to the function **fill\_tree()** but this time, this function returns a string value which is the value in the required leaf node following the correct traversal of the tree.

The function first of all checks if the pointer points to a leaf node or not. If it isn't a leaf node, the integer value, **i**, of the node is gotten using the **node\_value\_int()** function and the index of the string **number\_input** is **[i - 1]**. If the **char** at that index is a '0', the function is recursively called with parameters **t->left** and **number\_input**. However, if the **char** at that index is a '1', the function is recursively called with parameters **t->right** and **number\_input**. In the case that it is a leaf node, we have now gotten to where we want to be and so the function returns the string value of the node, **t->val**.

### 6. **std::string evalbdt(bdt t, const std::string &input)**

This function basically just contains only the function **get\_eval\_value()** with the same parameters as those entered into the **evalbdt()** function. The **t** parameter is the pointer to the root of the filled tree and **input** is the string to be evaluated.

## 7. **bool** check\_subtrees(**bdt** &t1, **bdt** &t2)

The function takes in pointers **t1** and **t2** to the roots of two different binary trees and checks if the trees are identical. It is a recursive function and returns a **bool** variable type and returns **true** if both trees are identical and returns **false** otherwise.

First of all, if both **t1** and **t2** are **NULL**, the function returns **true** because they are obviously identical due to being **NULL**. In the next **if** statement, the recursive function call occurs. If the root node values of both trees, the left subtrees and the right subtrees are identical, then the function returns **true**. Anything else returns **false**.

## 8. **bool** check\_tree(**bdt** &t)

This function takes in a pointer **t** to a tree as a parameter. It returns a **bool** data type and makes use of the **check\_subtrees()** function. It returns **true** if the left and right subtrees are identical and **false** otherwise.

## 9. **void** delete\_subtree(**bdt** &t)

This function takes in a pointer **t** to a tree and deletes the left subtree if the left and right subtrees are identical. The function then makes the pointer **t** point to the other subtree, in this case the right subtree. This function uses the **check\_tree()** function.

First of all, we create two temporary pointers of type **bdt** called **temp1** and **temp2**. Pointer **temp1** points to the root of the right subtree and pointer **temp2** points to the root of the main tree entered into the function, that is **temp1 = t->right** and **temp2 = t**. What we want to accomplish is to delete the left subtree together with the root of the tree and then go on to make **t** point to the right subtree. Therefore we make **temp2->right** point to **temp2->left** in order to isolate the root and left subtree. We then make **t** point to **temp1** which is the root of the right subtree which now becomes the root of the main tree.

## 10. **void** compact\_tree(**bdt** &t)

This function is a recursive function and uses the **delete\_subtree()** function. It also traverses the tree in Post-Order form and at each visit of a node, performs the **delete\_subtree()** function. It takes in the pointer **t** to the root of a binary tree. At the end of this function you get the compact tree form required in the assignment.

## 11. **bdt** buildcompactbdt(**const std::vector<std::string>&** fvalues)

This function takes in a **vector** of type **std::string** and returns the pointer to the compact tree. It contains both the **buildbdt()** and **compact\_tree()** functions.

First of all, we pass the vector **fvalues** to the **buildbdt()** function and we get the pointer **t** to the root of the tree. We then pass this pointer **t** to the **compact\_tree()** function and we finally return the new pointer **t** which points to the root of the new compact tree.

## 12. **std::string** evalcompactbdt(**bdt** t, **const std::string&** input)

This function basically just contains only the function **get\_eval\_value()** with the same parameters as those entered into the **evalcompactbdt()** function. The **t** parameter is the pointer to the root of the compact tree and **input** is the string to be evaluated.

## TESTING OF PROGRAM

Tests were carried out to ensure the program does what it was supposed to do. The image directly below is the first section of the **main** function in order for the user to input the ones and zeros sequence required for the program:

```
int main(){
    std::string y_n, input;
    std::vector<std::string> fvalues;
    std::cout << "Please enter the number" << std::endl;
    std::cin >> input;
    fvalues.push_back(input);
    std::cout << "Continue? y/n" << std::endl;
    std::cin >> y_n;
    while(y_n == "y"){
        std::cout << "Please enter the number" << std::endl;
        std::cin >> input;
        fvalues.push_back(input);
        std::cout << "Continue? y/n" << std::endl;
        std::cin >> y_n;
    }

    return 0;
}
```

As can be seen, it is a loop which only continues if the user inputs "y" to indicate more numbers are going to be input. Also, the **printtree()** function used for testing prints the tree in In-Order traversal.


### I. TEST FOR **initial\_tree()**

In order to test the function, the next section of the **main** function looked like this:

```
    bdt t = NULL;
    int i = 0;
    initial_tree(fvalues[0], t, i);
    printtree(t);

    return 0;
}
```

We initialized **bdt t** to be **NULL** and had **int i** initialized to 0. Those made up the parameters for the function including the index 0 value of the **fvalues** vector. The result was:



```
Please enter the number
101
Continue? y/n
n
0
x3
0
x2
0
x3
0
x1
0
x3
0
x2
0
x3
0
Process returned 0 (0x0)   execution time : 5.092 s
Press any key to continue.
```

As can be seen, only one number was input for the testing. There were three digits in the number and so the tree that came out was the correct form because it went from **x1** to **x3**.

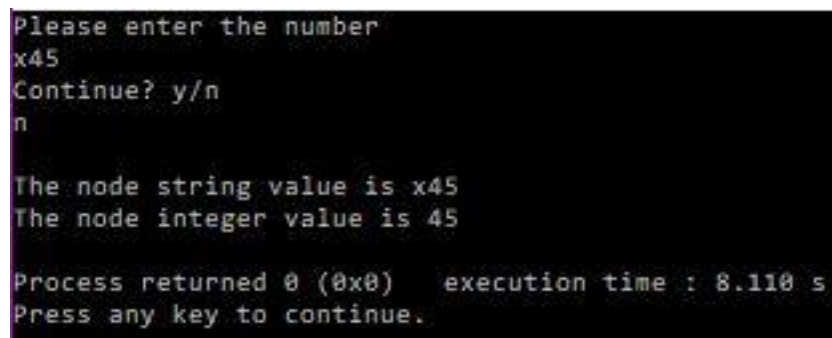
## II. TEST FOR **node\_value\_int()**

This function only works for non-leaf nodes on the tree. The **string** parameter entered begins with the letter x because all non-leaf nodes begin with 'x'. This section of the **main** function looked like this:

```
int k = node_value_int(fvalues[0]);
std::cout << std::endl;
std::cout << "The node string value is " << fvalues[0] << std::endl;
std::cout << "The node integer value is " << k << std::endl;

return 0;
}
```

In this test, the **fvalues** vector takes in a non-leaf node value string and converts it to an integer using the function and prints both node string value and the integer value. The result was as follows:



```
Please enter the number
x45
Continue? y/n
n

The node string value is x45
The node integer value is 45

Process returned 0 (0x0)   execution time : 8.110 s
Press any key to continue.
```

From the result, we get the correct value from **x45** to **45** and can say that the function performs its duties.

## III. TEST FOR **fill\_tree()**

For this test, the string value entered is index 0 of the **fvalues** vector. This section of the **main** function uses the **initial\_tree()** function. The **main** function looked like this:

```
bdt t = NULL;
int i = 0;
initial_tree(fvalues[0], t, i);
fill_tree(t, fvalues[0]);
printtree(t);
```

As can be seen from the picture above, we first had **bdt t** initialized to be **NULL**. Then we had the integer **i** initialized to be zero. This is for the **initial\_tree()** function. Then we had the **initial\_tree()** function with the **t**, **i** and **fvalues[0]** parameters. This is followed by the **fill\_tree()** function with the updated pointer **t** and **fvalues[0]** parameters.

The result of this test is as follows:

```
Please enter the number
000
Continue? y/n
n
1
x3
0
x2
0
x3
0
x1
0
x3
0
x2
0
x3
0
Process returned 0 (0x0)   execution time : 7.095 s
Press any key to continue.
```

The number "000" was input and this means that the most left leaf node will be filled with 1 and that was how the tree came out to look like. Therefore, this function works properly.

#### IV. TEST FOR **buildbdt()**

This function contains the **initial\_tree()** and **fill\_tree()** functions. The result of the test is shown below:

```
Please enter the number
000
Continue? y/n
y
Please enter the number
111
Continue? y/n
n
1
x3
0
x2
0
x3
0
x1
0
x3
0
x2
0
x3
1
1
```

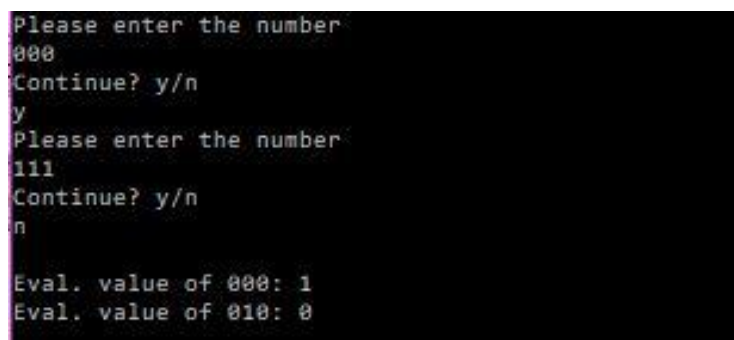
From the image above, the numbers 000 and 111 were input. This means that the most left and most right leaf nodes will have values of 1. This is shown and therefore this function does work properly because the number of nodes in the tree is correct and it gives the correct tree structure and node values.

## V. TEST FOR `get_eval_value()`

This function is used to evaluate the input into the program. We use the same input from the tree in the `buildbdt()` test. The `main` function is shown below:

```
bdt t = buildbdt(fvalues);
std::string right_value, wrong_value;
right_value = get_eval_value(t, fvalues[0]);
wrong_value = get_eval_value(t, "010");
std::cout << std::endl;
std::cout << "Eval. value of " << fvalues[0] << ": " << right_value << std::endl;
std::cout << "Eval. value of 010: " << wrong_value << std::endl;
```

The result is shown below:



```
Please enter the number
000
Continue? y/n
y
Please enter the number
111
Continue? y/n
n

Eval. value of 000: 1
Eval. value of 010: 0
```

As can be seen from the image above, the numbers that are input into the program are 000 and 111. Therefore only the leaf nodes corresponding to these numbers should contain the number 1. The evaluated value of 000 is clearly 0. However, 010 was not one of the input into the program and rightly so, its evaluated value is 0. I can therefore conclude, this function works properly.

## VI. TEST FOR `evalbdt()`

This function contains only the `get_eval_value()` function and they both have the same function parameters. Therefore, the `evalbdt()` test is identical to that of `get_eval_value()`. The structure of the `evalbdt()` function is shown below:

```
std::string evalbdt(bdt t, const std::string &input){
    return get_eval_value(t, input);
}
```

## VII. TEST FOR `check_subtrees()`

The `main` function for this test is shown below:

```
bdt t = buildbdt(fvalues);
if(check_subtrees(t->left, t->right)){
    std::cout << "This tree has identical subtrees" << std::endl;
}
else{
    std::cout << "This tree doesn't have identical subtrees" << std::endl;
}
```

The first result is shown below:

```
Please enter the number
00
Continue? y/n
y
Please enter the number
10
Continue? y/n
n
This tree has identical subtrees
```

As can be seen the two numbers used as input are 00 and 10. These will clearly have identical subtrees because the leaf nodes will have values of 1 – 0 – 1 – 0 from left to right. This test clearly shows that.

The second test result is shown below:

```
Please enter the number
00
Continue? y/n
y
Please enter the number
11
Continue? y/n
n
This tree doesn't have identical subtrees
```

As can be seen the two numbers used as input are 00 and 11. These will clearly not have identical subtrees because the leaf nodes will have values of 1 – 0 – 0 – 1 from left to right. This test clearly shows that.

I can therefore the function performs what it is supposed to do.

## VIII. TEST FOR **check\_tree()**

This function contains the **check\_subtrees()** function and takes in the pointer **t** to the tree. The test is similar to that of **check\_subtrees()** just by changing it to **check\_tree(t)** from **check\_subtrees(t->left, t->right)**. The **main** function for this test is:

```
bdt t = buildbdt(fvalues);
if(check_tree(t)){
    std::cout << "This tree has identical subtrees" << std::endl;
}
else{
    std::cout << "This tree doesn't have identical subtrees" << std::endl;
}
```

The structure of **check\_tree()** is shown below:

```
bool check_tree(bdt &t){
    if(check_subtrees(t->left, t->right)){
        return true;
    }
    else{
        return false;
    }
}
```



The test results for this function are identical to those from the **check\_subtrees()** function.

## IX. TEST FOR **delete\_subtree()**

The **main** for this test is:

```
bdt t = buildbdt(fvalues);  
delete_subtree(t);  
std::cout << std::endl;  
printtree(t);
```

The first result for this test is shown below:

```
Please enter the number  
00  
Continue? y/n  
y  
Please enter the number  
10  
Continue? y/n  
n  
  
1  
x2  
0
```

From the result above, the tree has identical subtrees and so it can be reduced to the form shown. It is also correct because from the number inputs, when the second digit in the number is 0, the leaf node is 1 which is seen from the reduced form.

The second result is shown below:

```
Please enter the number  
00  
Continue? y/n  
y  
Please enter the number  
11  
Continue? y/n  
n  
  
1  
x2  
0  
x1  
0  
x2  
1
```

From the result above, the trees do not have identical subtrees and so cannot be reduced anymore. Therefore the form shown, which is the original tree is the most-reduced form.

From the test results above, we can therefore conclude that the function performs what it is supposed to do.

## X. TEST FOR **compact\_tree()**

The **main** function for this test is shown below:

```
std::cout << std::endl;
bdt t = buildbdt(fvalues);
printtree(t);
std::cout << std::endl;
compact_tree(t);
std::cout << "The compact form of the tree is:" << std::endl;
printtree(t);
```

The results for the test are shown below:

```
0
x3
0
x2
1
x3
1
x1
0
x3
0
x2
1
x3
1
```

For the number inputs, we went with 010, 011, 110 and 111 which were used in the example for the assignment. The result shown above is the original tree after **buildbdt(t)** and before **compact\_tree(t)**. After the **compact\_tree(t)**, the result is shown below:

```
The compact form of the tree is:
0
x2
1
```

As can be seen, the compact tree is the correct form and therefore works.

## XI. TEST FOR **buildcompact()**

This function contains both **buildbdt()** and **compact\_tree()**. The main function is shown below:

```
bdt t = buildcompactbdt(fvalues);
std::cout << std::endl;
std::cout << "The compact form of the tree is:" << std::endl;
printtree(t);
```

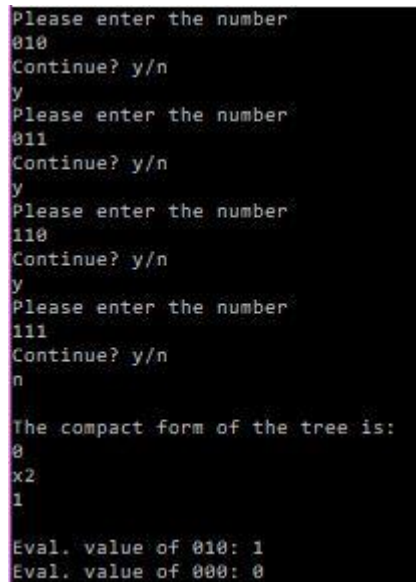
The result of the test is identical to that from Test X for **compact\_tree()**.

## XII. TEST FOR `evalcompactbdt()`

The `main` function for this test is shown below:

```
bdt t = buildcompactbdt(fvalues);
std::cout << std::endl;
std::cout << "The compact form of the tree is:" << std::endl;
printtree(t);
std::string right_value, wrong_value;
right_value = evalcompactbdt(t, fvalues[0]);
wrong_value = evalcompactbdt(t, "000");
std::cout << std::endl;
std::cout << "Eval. value of " << fvalues[0] << ": " << right_value << std::endl;
std::cout << "Eval. value of 000: " << wrong_value << std::endl;
```

The result is shown below:



```
Please enter the number
010
Continue? y/n
y
Please enter the number
011
Continue? y/n
y
Please enter the number
110
Continue? y/n
y
Please enter the number
111
Continue? y/n
n

The compact form of the tree is:
0
x2
1

Eval. value of 010: 1
Eval. value of 000: 0
```

The number inputs are the same used in the assignment example. The compact tree form is correct and the evaluated values are also correct because 010 was an input but 000 was not. I can therefore conclude this function works well.