Hacettepe University

Computer Engineering Department

BMM203 Data Structures Lab - 2020 Fall

Assignment 4 - Trees

January 3, 2021

Student name:
Deniz Gönenç

Student Number:
b21946146

1 Encoding Algorithm and Code

My encoding algorithm uses two main functions: convert() and encode(). convert() is a recursive function that saves the binary codes of the characters in a vector.

1.1 convert()

convert() uses Inorder traversal to visit all the leaves which have all the characters. Other nodes are only internal nodes. If the current node has a left child, it appends a "0" to the current string and calls convert() for the left child. If the current node has a right child, it appends a "1" to the current string and calls convert() for the right child. If all possible recursive calls were called for the current node, then we know that it is a leaf node and push this character to the characters vector and push its binary code to the huffmans vector.

```
void Tree::convert(Node* node, string s){
    if (node->left != nullptr){
        convert(node->left, s + "0");
    }
    if(node->right != nullptr){
        convert(node->right, s + "1");
    }

    if(node->isInternal != true){
        characters.push_back(node->ch);
        huffmans.push_back(s);
    }
}
```

1.2 encode()

encode() starts the convert function by passing the root as node and an empty string as s. Then we begin building our result string which is our encoded string. The for loop iterates over the initial input string character by character. In each iteration it finds the current character's index in the characters vector(which we initialized in convert()). Then it finds the string in that index in the huffmans vector(which we initialized in convert()). This string is the binary code of the current character. We append this string to our resulting string. The loop continues until the initial input string ends.

```
void Tree::encode(){
    convert(root, "");
    result = "";

    for(int i = 0; i < input.length(); i++){
        auto it = find(characters.begin(), characters.end(), tolower(input[i]));
        if (it != characters.end()){
            int index = it - characters.begin();
            result += huffmans[index];
        }
        else{
            cout << "char␣not␣in␣vector" << endl;
        }
    }
    cout << result << endl;
}
```

2 Decoding Algorithm

My decoding algorithm uses one function: decode(). It takes one string argument which is the string to be decoded, and it returns a string which is the decoded string.

We begin by setting an empty string to append our characters to and setting a temp Node pointer to root. We iterate over the argument string character by character. We check if the character is a "0" or "1".
If it is "0" and temp has a left child, we set temp to temp's left child. If the character is "0" but temp does not have a left child then we have found a leaf, so we append the character of this leaf to our result string, then we set temp to root's left child(since we already read this "0" character). If the character is "1" and temp has a right child, we set temp to temp's right child. If the character is "1", but temp does not have a right child, then we have found a leaf, so we append the character of this leaf to our result string, then we set temp to root's right child(since we already read this "1" character).

In the end there will be an unread character left in temp, so we append that character to our result string, then return our result string.

```
string Tree::decode(string h, vector<char> chars, vector<string> huff){
    string res = "";
    string s = "";
    for (int i = 0; i < h.length(); i++){
        s += h[i];
        auto it = find(huff.begin(), huff.end(), s);
        if (it != huff.end()){
            int index = it - huff.begin();
            res += chars[index];
            s = "";
        }
    }
    return res;
}
}
```

3 List Tree Command

I use the listTree() function to list the tree and a small utility function multiplyString().

3.1 listTree()

listTree() is a recursive function. It has a node argument, a rec argument to keep the depth of recursions, a check argument for the "|" character and a checkrec argument to keep the recursion level of where check was set to true. We use Preorder traversal to print the tree. If the current node is an internal node we print it as "IN", if it is not an internal node then it is a leaf node and it has a character. So we print it as that character.

Then we look for a left child. If there is a left child and a right child, we set check to true, so that we can print "|"s accordingly. We call listTree for the left child, setting node to node→left, increasing the recursion depth, setting check to true and setting the checkrec to the current recursion level. After we come out of the left child's recursive calls, we check if the current recursion level is equal to checkrec. If it is we set check to false. Because we are done with the left child branch of that recursion level and we don't need check to be true anymore.

If there is only a left child, we don't have to worry about check for the current recursion level. We call listTree for the left child, setting node to node→left, increasing the recursion depth, setting check to false and setting checkrec to -1. If there is a right child, we call listTree for the right child, setting node to node→right, increasing the recursion depth, and keeping the check and checkrec values.

```
//node = root, rec = 0, check = false, checkrec = -1 in initial function
void Tree::listTree(Node* node, int rec, bool check, int checkrec){
    cout << multiplyString(rec, check) << "+-␣";
    if(node->isInternal){
        cout << "IN" << endl;
    }
    else{
        cout << node->ch << endl;
    }

    if(node->left != nullptr){
        if(node->right != nullptr){
            listTree(node->left, rec+1, true, rec);
            if (checkrec == rec){
                check = false;
                checkrec = -1;
            }
        }
        else listTree(node->left, rec+1, false, -1);
    }

    if(node->right != nullptr){
        listTree(node->right, rec+1, check, checkrec);
    }
}
```

3.2 multiplyString()

multiplyString builds up a string according to rec, the recursion depth. For each recursion level it adds a tab character. If check is true, it also adds "|" between the tabs. When it is done it return the built string.

```
string multiplyString(int a, bool check){
    string res = "";
    if(a == 0) return res;
    if(a == 1) return "\t";
    while(a > 0){
        if (check){
            res += "\t";
            if (a != 1) res += "|";
        }
        else{
            res += "\t";
        }
        a--;
    }
    return res;
}
```

4 Notes

I have not used my 2-day extension (without penalty) in previous assignments and I would like to use it on this one.