Specification

This telegraph program demonstrates how morse code is built using a binary tree. The program will read in morse code symbols and build a tree by going left for '.' and right for '-'. The program will demonstrate that it works by taking in a string and outputing its morse code translation. Then it will read the morse code back in and translate it back into letters.

Analysis

- Inputs: The program reads in a list of morse code defined in the TELEGRAPH::table. It will also get an user inputted string from the console.
- Process: The morse code tree is first built by reading the table. In the code section of the MORSECODE object, the program will read each character, building the tree left or right based on if it is a dot or a dash. Then the program will prompt the user for an input. When the input is recieved, the program will encode it into morse code and display it. Then it will read the morse code back into the program and decode it into letters.
- Outputs: This program will output the message encrypted into morse code, and the message after it has been decrypted.

Design

- morse.h: Header file for struct MORSECODE, TNODE and class TELEGRAPH
- morse.cpp: This contains the morse code table
- buildtree.cpp: Reads from the MORSECODE table and build the binary tree
- destroytree.cpp: Public function calls the overloaded private function which deletes each node recursively
- encode.cpp: Takes in two references to char arrays, the first being the string to translate, and the 2nd is used to store the encoded translation
- decode.cpp: Takes in two references to char arrays, the first being the morse code string to decode, and the second is to store the decoded message

Implementation: morse.h File

```
#include <iostream>
static const char DOT = '.';
static const char DASH = '-';
struct MORSECODE {
  char symbol;
  char code[10];
};

struct TNODE {
  char symbol;
  TNODE *left;
  TNODE *right;

  TNODE() {
    symbol = '*';
    left = right = 0;
  }
};
```

DOT and DASH are constants to refer to '.' and '-'. The struct MORSECODE has a char element symbol which will be the letter, number, or punctuation marks. It also has a char array to store its morse code translation.

The struct TNODE also has a char element named symbol. The node only needs to know the english translation of the character. As a tree node, TNODE also as a left and a right pointer.

Implementation: morse.h File

```
class TELEGRAPH {
  enum {N=40};
  private:
    static MORSECODE table[N];
    static TNODE *root;
    static void recursiveDisplay(TNODE* n){
        if (!n) return;
        recursiveDisplay(n->left);
        std::cout << n->symbol << "\t";
        recursiveDisplay(n->right);
    }
    static void DestroyTree(TNODE *node);
```

This is the private section of the TELEGRAPH class. It contains the morse code table that will be used to create the tree. It contains a TNODE pointer as its root. There is also a recursive display function that is passed a pointer to a node and traverses the tree in order. It also declares a private member function for DestroyTree which takes a pointer to a TNODE as its parameter.

Implementation: morse.h File

```
  public:
    static void buildTree();
    static void displayTree() {recursiveDisplay(root);}
    static void DestroyTree();
    void Encode(char text[], char morse[]);
    void Decode(char morse[], char text[]);
};
```

These are the declarations for the public member functions of the TELEGRAPH class. The static void function displayTree will call the private recursiveDisplay function.

Implementation: morse.cpp File

```
#include <ctype.h>
#include "morse.h"
MORSECODE TELEGRAPH::table[40] = {
 {'A', ".-"}, {'B', "-..."}, {'C', "-.-."}, {'D', "-.."},
 {'E', "."}, {'F', "..-."}, {'G', "--."}, {'H', "...."},
 {'I', ".."}, {'J', ".---"}, {'K', "-.-"}, {'L', ".-.."},
 {'M', "--"}, {'N', "-."}, {'O', "---"}, {'P', ".--."},
 {'Q', "--.-"}, {'R', ".-."}, {'S', "..."}, {'T', "-"},
 {'U', "..-"}, {'V', "...-"}, {'W', ".--"}, {'X', "-..-"},
 {'Y', "-.--"}, {'Z', "--.."},
 {'0', "-----"}, {'1', ".----"}, {'2', "..---"}, {'3', "...--"},
 {'4', "....-"}, {'5', "....."}, {'6', "-...."}, {'7', "--..."},
 {'8', "---.."}, {'9', "----."},
 {'.', ".-.-.-"}, {',', "--..--"}, {'?', "..--.."},
 {'\0', "END"}
};
```

This is the morse code table that will be read in to build the tree.

Implementation: buildtree.cpp File

```
#include "morse.h"
TNODE *TELEGRAPH::root = 0;
void TELEGRAPH::buildTree(){
  TNODE *node, *nextNode;
  int i;
  char *dd;

  root = new TNODE;
  if (!root) return;
  root->symbol = ' ';

  for(i = 0; table[i].symbol; i++){
```

The variable i is used to traverse the table. The char pointer dd is used to traverse the MORSECODE code elemenent. There is no tree when this function is called so the root is set to a new TNODE.

Implementation: buildtree.cpp File

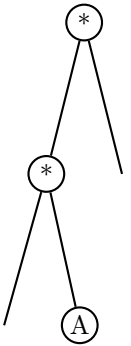
```
for(i = 0; table[i].symbol; i++){
  node = root;
  for(dd = table[i].code; *dd; dd++){
      if(*dd == '.'){
          if(!node->left){
            TNODE* newnode = new TNODE;
            node->left = newnode;
          }
          node = node->left;
      }
```

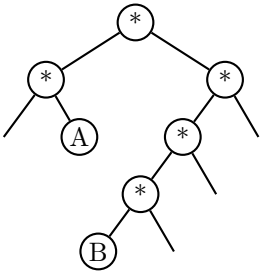
In the outer for loop, i iterates through the MORSECODE table, stopping at each element. The pointer node is set to the root at the beginning of each iteration. The inner for loop has dd iterating through the code element which are the dots and dashes. When i = 0, and dd = table[i], it checks if *dd is a dot. Because it is a dot, it checks if there is an node to the left of the current node. Because there is no node to the left, it creates a new node on the left. Then it sets node equal to the new node, moving our tracker down the tree. The inner loop moves onto the next iteration.

Implementation: buildtree.cpp File

```
        }
        if(*dd == '-'){
            if(!node->right){
              TNODE* newnode = new TNODE;
              node->right = newnode;
            }
            node = node->right;
        }
    }
    node->symbol = table[i].symbol;
  }
}
```

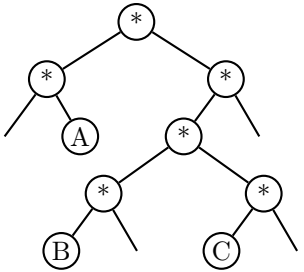
The next character is a DASH so it checks if there is a node to the right. Because there is no node to the right, it creates a node. Now that the inner for loop is complete, it sets the symbol into the node. Then it continues onto the next MORSECODE.



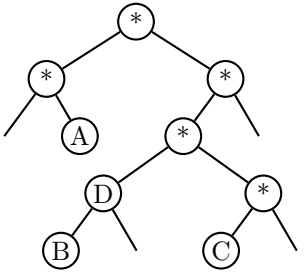


The next MORSECODE is "-..." so it will go to the right once, then left three times. It creates a new node each time if there is none.

"C" is DASH DOT DASH DOT.



"D" is DASH DOT DOT which already has a node there, so it will replace the "*" with "D".



Implementation: destroytree.cpp File

```
#include "morse.h"

void TELEGRAPH::DestroyTree(TNODE *node){

    if(node && !node->left && !node->right){
        delete node;
    } else {
    if(node->left)
    DestroyTree(node->left);
    if(node->right)
    DestroyTree(node->right);
    delete node;
    }
}

void TELEGRAPH::DestroyTree(){
  DestroyTree(root);
  root = 0;
}
```

When the public function DestroyTree is called, the overloaded private member function DestroyTree(TNODE) is called. The private DestroyTree function traverses down the tree until it finds a leaf node and deletes it. As the function returns up the call stack, using a post oder traversal method, it deletes the node on the left if there is one, then right if there is one, and finally deleting itself.

Implementation: encode.cpp File

```
void TELEGRAPH::Encode(char text[], char morse[]){
  int i; char c, *t, *dd;

  for(t=text; *t; t++){
      c = toupper(*t);
      if(c == ' '){
        *morse++ = ' ';
        continue;
      }
      for (i=0; table[i].symbol; i++)
        if (table[i].symbol == c ) break;

      if(!table[i].symbol) continue;
      dd = table[i].code;
      while (*dd) *morse++ = *dd++;
      *morse++ = ' ';
  }
  *morse = '\0';
}
```

The string inputed by the console is read as a char array. It first converts all letters to uppercase. Then it looks through the table to find the symbol. Once the symbol is matched, it uses the pointer dd to iterate the morse code and add it to the morse array.

Implementation: decode.cpp File

```
void TELEGRAPH::Decode(char morse[], char text[]){
  char *dd;
  TNODE *node;
  char *i;

  node = root;
  for (dd = morse; *dd; dd++) {
      if(*dd == '.') node = node->left;
      else if(*dd == '-') node = node->right;
      else if(*dd == ' '){
        *text = node->symbol;
        text++;
        node = root;
      }
  }
  *text = '\0';
}
```

Decode takes in two pointers to arrays as parameters. The first is the morse array. The pointer dd will iterate through this array. We also have a TNODE pointer called node. This pointer will be our runner. The outer for loop will start reading each charater in the morse array. When it reads a DOT, the node runner will move to the left. When it reads a DASH, the node runner will move to the left. When it reads a " ", the symbol is saved into the *text and text is incremented by one charater space. Then the node runner is set back to the root. When the whole morse array is read, a "
0" is added to signify the end of a string.

Implementation: tstmorse.cpp File

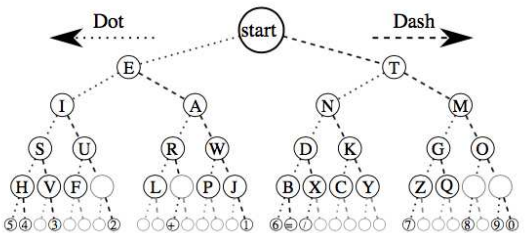
```
int main(){
    TELEGRAPH station;
    char text[80], morse[600];

    TELEGRAPH::buildTree();
    TELEGRAPH::displayTree();

    std::cout << "\nEnter telegram ==> ";
    std::cin.getline(text,80);
    std::cout << "\nSending >>> ";
    station.Encode(text, morse);
    std::cout << morse;
    std::cout << " >>> Received\n\n";
    station.Decode(morse, text);
    std::cout << text << std::endl;
    TELEGRAPH::DestroyTree();
    std::cout << "tree destroyed\n";
    TELEGRAPH::displayTree();

}
```

This is the main function. It first creates a TELEGRAPH object station and two char arrays. TELEGRAPH calls its static member function buildTree to build the morse code tree. displayTree is called to verify the tree is built properly. Then the program asks for the user to input a message. When the message is inputted, the program will call Encode which will turn the message into morse code. This message is printed out. Then the program will take the morse code and pass it to the Decode function which will recreate the original string. Finally the program calls DestroyTree which will free up the memory by deleting each node recursively

Test

This is a test of the displayTree function. The displayTree function prints in order of smallest to largest or left, root, right. When compared to the morse code tree, we can tell the program has built the tree correctly.

```
debian@debian:~/lab6$ ./lab
5       H       4       S       V       3       I
F       U       ?       *       *       2       E
L       R       *       .       *       A       P
W       J       1               6       B       D
X       N       C       K       Y       T       7
Z       *       ,       G       Q       M       8
*       0       9       *       0
```



Test

This is a test of the encoding and decoding. When we test for a word "hello" we should get DOT DOT DOT DOT DOT DOT DASH DASH DASH DOT DASH DASH DASH DASH DASH DASH. This corresponds to the result shown in the test. The decoded message should be "HELLO" in all caps which the test proves.

```
Enter telegram ==> hello

Sending >>> .... . .-.. .-.. ---  >>> Received

HELLO
tree destroyed
debian@debian:~/lab6$
```