Use the struct to store each information of the node.

```
struct node {
    //establish the node to contain symbol, frequence, and pointer to point to children
    char symbol;
    int frequence;
    node *leftNode;
    node *rightNode;
};
```

Use “createHeapNode” function to create a new node with the given information.

```
node createHeapNode(char sym, int freq) {
    //allocate a new memory for the new node
    node *newNode = (node*)malloc(sizeof(node));
    newNode->leftNode = newNode->rightNode = NULL;
    newNode->frequence = freq;
    newNode->symbol = sym;
    return *newNode;
}
```

Use “sawpNode” to exchange two nodes.

```
void swapNode(node &a, node &b) {
    //use the tmp to be the media to swap node
    node tmp = a;
    a = b;
    b = tmp;
}
```

Use “”heapify” function to implement the minheap tree rule.

```
void heapify(vector<node> &minheap, int i) {
    //let the parent node to be smallest
    int smallest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    //compare the left child first
    if (left < (signed)minheap.size() && minheap[left].frequence < minheap[smallest].frequence)
        smallest = left;
    else
        smallest = i;

    //compare the right child
    if (right < (signed)minheap.size() && minheap[right].frequence < minheap[smallest].frequence)
        smallest = right;

    //if the smallest one changes then swap the node, and heapify the subtree
    if (smallest != i) {
        swapNode(minheap[smallest], minheap[i]);
        heapify(minheap, smallest);
    }
}
```

Extrac the top node from the heap and return the node to the place it calls.

```
node* extractMin(vector<node>&minHeap)
{
    //extract the top of the heap tree
    node *top = new node;
    top->frequence = minHeap[0].frequence;
    top->symbol = minHeap[0].symbol;
    top->rightNode = minHeap[0].rightNode;
    top->leftNode = minHeap[0].leftNode;

    //turn the last one to be the top of the heap
    minHeap[0] = minHeap[minHeap.size() - 1];
    minHeap.pop_back();

    //renew the heap size and heapify again
    heapify(minHeap, 0);

    return top;
}
```

Use this function below to insert a new node into the heap tree

```
void insertMinHeap(vector<node> &minHeap, node node)
{
    int i = minHeap.size();
    //check the value starting from last node to choose the proper place
    while (i && node.frequence < minHeap[(i - 1) / 2].frequence) {
        if (i >= (signed)minHeap.size())
            minHeap.push_back(minHeap[(i - 1) / 2]);
        else
            minHeap[i] = minHeap[(i - 1) / 2];
        i = (i - 1) / 2;
    }
    //insert the node into the heap tree
    if(i >= (signed)minHeap.size())
        minHeap.push_back(nc size_type std::vector::size()
    else
        minHeap[i] = node;
}
```

Use the function below to create the Huffman tree which follows the rule to create a new node by adding the frequency of two top node extracted from the heap tree.

```
node* buildHuffmanTree(vector<node> &minHeap)
{
    // new memeory to store the node pointer
    node *leftChild = new node;
    node *rightChild = new node;
    node *parent = new node;
    while (minHeap.size() != 1) {
        //extract the node from the top of the heap tree
        rightChild = extractMin(minHeap);
        leftChild = extractMin(minHeap);
        //use the node extracted to create the new node
        *parent = createHeapNode('$', leftChild->frequence + rightChild->frequence);
        parent->leftNode = leftChild;
        parent->rightNode = rightChild;
        //insert the new node into the heap tree
        insertMinHeap(minHeap, *parent);
    }
    //after there is only one node in the heap tree return the root
    return extractMin(minHeap);
}
```

Use the function below to return a int vector which store the results after preorder traversal

```
vector<int> preorder(node* root) {
    stack<node*> s;
    vector<int> traversal;
    s.push(root);
    while (s.size() != 0)
    {
        //get the node on the top of the stack
        node *top = s.top();
        //pop and then push the right node first and then right node
        s.pop();
        if (top->rightNode) s.push(top->rightNode);
        if (top->leftNode) s.push(top->leftNode);
        //push the data into vector
        traversal.push_back(top->frequence);
    }
    return traversal;
}
```

Use the function below to do the same thing as above but change the traversal way to inorder traversal

```
vector<int> inorder(node* root) {
    stack<node*> s;
    vector<int> traversal;
    node *current = root;
    while (current || !s.empty())
    {
        //keep push into the stack while the left branch doesn't traverse over
        if (current) {
            s.push(current);
            current = current->leftNode;
        }
        else {
            //while there is no left node then pop the top on the stack and then start to check right branch
            node* top = s.top();
            s.pop();
            traversal.push_back(top->frequence);
            current = top->rightNode;
        }
    }
    return traversal;
}
```

Use the function to get the max level of the Huffman tree by get the longest string of the Huffman code.

```
void getLevel(vector<string> code, int &levelCount) {
    int counter = 0;
    levelCount = 0;
    //by getting the longest code string in the vector to set as maxlevel
    for (unsigned int i = 0; i < code.size() / 2; i++)
    {
        counter = code[i * 2 + 1].length();
        if (counter > levelCount)
            levelCount = counter;
    }
}
```

According to the rule that when traverse to the left branch then add “1”, the right branch then add “0”, build the code string of each symbol. And since the recursive method is used, the last char must be removed after the string returned.

```
void huffmanEncode(node* root,vector<string> &code, string codeString) {
    // traverse to the left branch add 1 into the code string
    if (root->leftNode)
    {
        codeString += "1";
        huffmanEncode(root->leftNode, code, codeString);
        codeString.erase(codeString.end()-1);
    }
    // traverse to the right branch add 0 into the code string
    if (root->rightNode)
    {
        codeString += "0";
        huffmanEncode(root->rightNode, code, codeString);
        codeString.erase(codeString.end()-1);
    }
    //when get to the leaf then recode the code string and the symbol
    if (!root->leftNode && !root->rightNode) {
        char *data = &(root->symbol);
        code.push_back(data);
        code.push_back(codeString);
    }
}
```

Use the function below to decode the string input by traverse to left if the input char equals to “1” and to right if to “0”, and add the symbol got from the traversal to the decode string.

```
void huffmanDecode(node* root, string encode, string &decode) {

    node *current = root;
    // traverse through the huffman tree by the 1 and 0 in the string
    for (unsigned int i = 0; i < encode.length(); i++)
    {
        if (encode[i] == '1')
            current = current->leftNode;

        else if(encode[i] == '0')
            current = current->rightNode;

        if (!current->leftNode && !current->rightNode)
        {
            decode += current->symbol;
            current = root;
        }
    }
}
```

Preprocess the data input:

Get different chars and store in the “symbolString”

```
for (unsigned int i = 0; i < input.length(); i++)
{
    char newChar = input[i];
    if (!i || symbolString.find(newChar, 0) == string::npos)
        symbolString += newChar;
}
size = symbolString.size();
cout << "Your input size[int]: "<< size << endl;
```

Reorder the symbol and print in proper order

```
for (unsigned int j = symbolString.length(); j > 0; j--) {
    for (unsigned int i = 0; i < symbolString.length() - 1; i++)
    {
        if (symbolString[i] > symbolString[i + 1]) {
            char mask = symbolString[i];
            symbolString[i] = symbolString[i + 1];
            symbolString[i + 1] = mask;
        }
    }
}
//print the symbol
cout << "Your input symbol[chat]: ";
for (unsigned int i = 0; i < symbolString.length(); i++)
    cout << symbolString[i] << " ";
cout << endl;
```

Count the frequency of each symbol and then print out

```
//count the frequency of each symbol
for (unsigned int i = 0; i < symbolString.length(); i++)
{
    int freq = 0;
    for (unsigned int j = 0; j < input.length(); j++)
    {
        if (input[j] == symbolString[i])
            ++freq;      string main::symbolString
    }
    inputFrequence.push_back(freq);
}
cout << "Your input frequency[chat]: ";
for (unsigned int i = 0; i < inputFrequence.size(); i++)
    cout << inputFrequence[i] << " ";
cout << "\n==========================================";
```

Build the min heap tree by creating new nodes and putting the nodes into the function to heapify.

```
/*build the min heap tree by the given data*/
for (int i = 0; i < size; i++)
{
    node newNode;
    newNode = createHeapNode(symbolString[i], inputFrequence[i]);
    minHeapTree.push_back(newNode);
    for (int j = minHeapTree.size() / 2 - 1; j >= 0; j--)
    {
        if (j < 0)
            j = 0;
        heapify(minHeapTree, j);
    }
}
```

Print the minheap tree by level

```
//display the min heap tree in correct format
cout << "=========================================";
cout << "\nMinHeap Tree:\n";
for (int i = 0;; i++)
{
    int j;
    /*
            0     <- the first one in each row equals to "2 ^ i - 1", Ex: 2 ^ 0 - 1 = 0, 2 ^ 1 - 1 = 1
          / \
         1   2   <- the limit in each row must equals to "first one + 2 ^ level", Ex: 0 + 2 ^ 0 = 1, 1 + 2 ^ 1 = 3
        / \ / \
       3  4 5  6 <- only the index is less than size will be printed
    */
    for (j = (int)pow(2, i) - 1; j < (int)(pow(2, i) - 1 + pow(2, levelCount)); j++)
    {
        if(j < size)
            cout << minHeapTree[j].frequence << "__";
        else break;
    }
    cout << endl;
    levelCount++;
    if (j >= size - 1)
        break;
}
```

Build the Huffman tree and print out the tree by inorder and preorder traversal only pass the parameter of the root pointer, and create the encoded string to put into the function to get the max level of the tree.

```
cout << "=========================================";
cout << "\nHuffman Tree:\n";
cout << "Preorder: ";

//print the huffman tree by preorder traversal
vector<int> preorderTraversal = preorder(huffmanTreeRoot);
for (unsigned int i = 0; i < preorderTraversal.size(); i++)
    cout << preorderTraversal[i] << " ";
cout << endl;
//print the huffman tree by inorder traversal
cout << "Inorder: ";
vector<int> inorderTraversal = inorder(huffmanTreeRoot);
//encode the symbol by huffman tree
huffmanEncode(huffmanTreeRoot, HuffmanCode, "");
//use the encoded string to get the max level
getLevel(HuffmanCode, levelCount);

for (unsigned int i = 0; i < preorderTraversal.size(); i++)
    cout << inorderTraversal[i] << " ";
cout << endl;
cout << "Max Level: " << levelCount << endl;
cout << "Number of node: " << inorderTraversal.size() << endl;
//print the information of coding
cout << "=========================================";
```

Get the decoded string by pass the encoded string as parameter, and print the string out.

```
cout << "\nInput sequence for decode:";
//ask the user to input the encoded string
cin >> encodedString;
//decode the encoded string
huffmanDecode(huffmanTreeRoot, encodedString, decodedString);
cout << "\nDecode Huffman Data:\n";
cout << decodedString << endl;
```