Binary Trees
Binary Search Trees

1. Code Walkthroughs
2. Last year’s assignment on Huffman coding
3. Some problems related to BST
   a. Kth Smallest element in BST
   b. Is Binary tree a BST?
   c. Merging BST

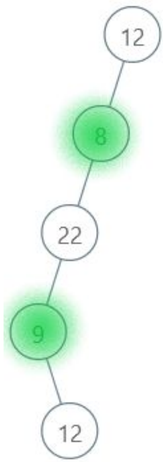
Node definition

```
#include <stdio.h>
#include <stdlib.h>
struct node
{
   int data; //node will store an integer
   struct node *right; // right child
   struct node *left; // left child
};
```

Creating a new node

```
struct node* newNode(int data) {
	struct node* temp = (struct node*) malloc(sizeof( struct node ));
	temp->data = data;
	temp->left = temp->right = NULL;
	return temp;
}
```

Creating a binary tree



[12, 8, null, 22 , null, 9, null, null, 12]

Rules:

1. Whenever new node is added to binary tree, node is pushed into queue
2. Node will stay in queue until both it's children are not filled
3. One both children are filled, node is popped from queue.

Steps:

1. Create a queue. Create node from first array element and push to queue.
2. One by one create a node from next element in array and if queue.peek element has left child, insert there, else insert as right child.
3. If both children are filled, pop node from queue.

Creating a binary tree

```
void insert(struct node **root, int data, struct Queue* queue) {
        struct node *temp = newNode(data);            // Create a new node for given data
        if (!*root)  {*root = temp;}                  // If the tree is empty, initialize the root with new node.
        else    {
        struct node* front = getFront(queue);         // get the front node of the queue.
         while (!front) {
                Dequeue(queue);
                front = getFront(queue);
         }
        if (!front->left){ front->left = temp;}       // If no left child, set left as temp
        else if (!front->right){
                front->right = temp;
                Dequeue(queue);}                      // If no right child, set right as temp
        Enqueue(temp, queue);                         // Enqueue() the new node for later insertions
}
```

Creating a binary search tree

```
struct node* insert(struct node *root, int x) {
   //searching for the place to insert
   if(root==NULL)
      return new_node(x);
   else if(x>root->data) // x is greater. Should be inserted to right
      root->right_child = insert(root->right_child, x);
   else // x is smaller should be inserted to left
      root->left_child = insert(root->left_child,x);
   return root;
}
```

Search for a new node in BST

```
struct node* search(struct node *root, int x) {
   if(root==NULL || root->data==x) // Element is found
           return root;
   else if(x>root->data) // x is greater, so we will search the right subtree
           return search(root->right_child, x);
   else //x is smaller than the data, so we will search the left subtree
           return search(root->left_child,x);
}
```

Find Predecessor of BST

Steps:

1. If root is NULL; return;
2. If key is found;
   a. If left subtree is not NULL; predecessor = right most child of left subtree or left child
   b. If left subtree is NULL; predecessor = ancestor. Move up to root until node is right child of its parents.
   c. If still not found; predecessor does not exist

Find Predecessor of BST

```
Node *findPredecessor(Node* root, int key) {
        Node *predecessor = NULL;
        Node *curr = root;
        If (!root) return NULL;
        While (curr && curr->data != key) {
                If (curr->data > key) { curr = curr->left;}
                Else {
                        Predecessor = curr;
                        Curr = curr->right;
        }
        If (curr & curr->left) { Predecessor = findMaximum(curr->left); }
        Return predecessor;
}
```

Find Predecessor of BST (Recursive)

```
void findPredecessor(Node* root, Node*& prec, int key) {
  if (root == nullptr)   {
    prec = NULL;
    return;
  }
  if (root->data == key) {
    if (root->left) {prec = findMaximum(root->left); }          // Predecessor is max value in left subtree
  }
  else if (key < root->data) {
    findPredecessor(root->left, prec, key);          // If curr node != key and key < curr node
  }
  else {
    prec = root;                                      // Need to find ancestor.
    findPredecessor(root->right, prec, key);
  }
}
```

Find Maximum Helper Function

```
struct Node* findMaximum(struct Node* root) {
        If (!root) {return NULL;}
        while (root->right) {        root = root->right;          }
        return root;
}
```

Find Minimum

```
//function to find the minimum value in a node
struct node* find_minimum(struct node *root)
{
   if(root == NULL)
          return NULL;
   else if(root->left_child != NULL) // node with minimum value will have no left child
          return find_minimum(root->left_child); // left most element will be minimum
   return root;
}
```

Practise Yourself

- In-order traversal
- Pre-order traversal
- Post-order traversal
- Delete a node

Last Year Assignment - Huffman Coding

For communicating through digital mediums, we must encode messages into bit streams first. In such a system, the sender needs an encoder for encoding text messages into bit streams, and the recipient, on receiving this bit stream, uses a decoder for reconstructing the original message. Today we will see such an algorithm called Huffman Coding, which can encode text messages into bit streams of minimal length. The algorithm is based on a binary-tree frequency-sorting method that allows to encode any message sequence into a bit stream and reassemble any encoded message into its original version without losing any data. When you compare between fixed-length binary encoding and Huffman coding, the later has the least possible expected message length (under certain constraints).

Part I - Observing Frequency Distribution

As a first step towards building an efficient prefix code for communication, you would need to gather symbol probabilities first. Thankfully, you have access to a log of previously communicated messages as a text file.

The -> t:1, h:1, e:1
Over -> o:1, v:1, e:2, r:1

FILE: *log.txt*

```
16
the
quick
brown
fox
jumps
over
a
lazy
dog
the
five
boxing
wizards
jump
quickly
ten10
```

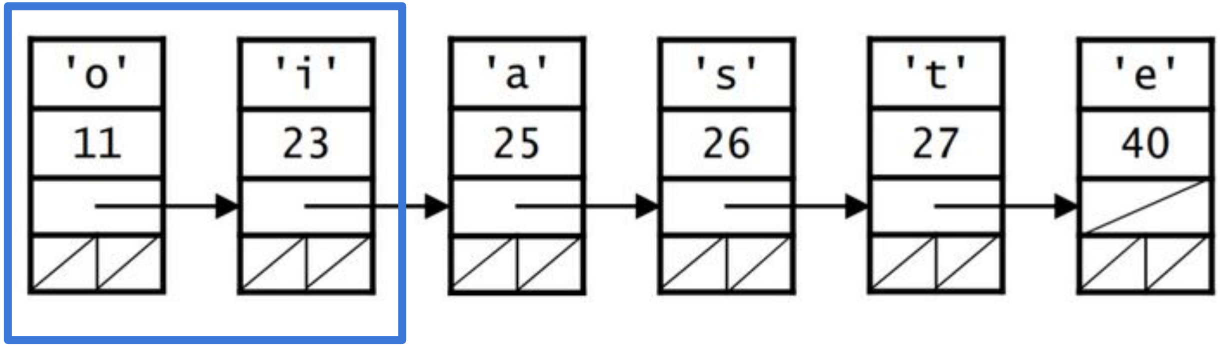
Last Year Assignment - Huffman Coding

character	Frequency
o	11
i	23
a	25
s	26
t	27
e	40

character	code-word
e	0
a	100
s	101
o	1100
i	1101
t	111

Part II - Finding the Huffman Coding

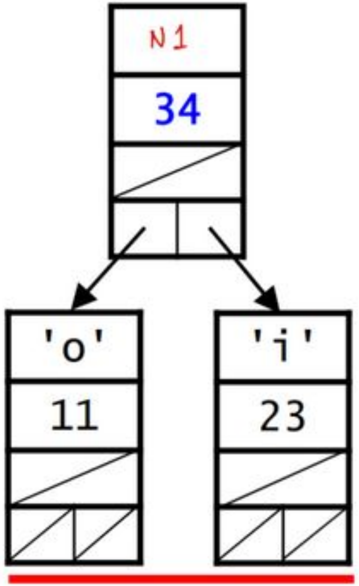
Build the huffman tree:
1. Create a node for each character. Build a sorted linked list from these individual nodes



Part II - Finding the Huffman Coding

Build the huffman tree:
2.
Take least frequency items and merge them. Add the two nodes as children of merged node.

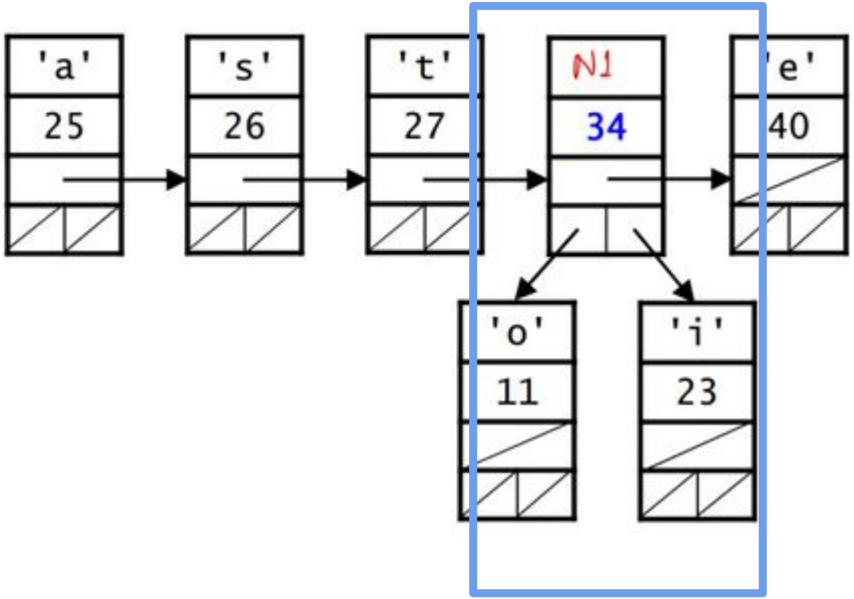
Least frequency node is left child



Part II - Finding the Huffman Coding

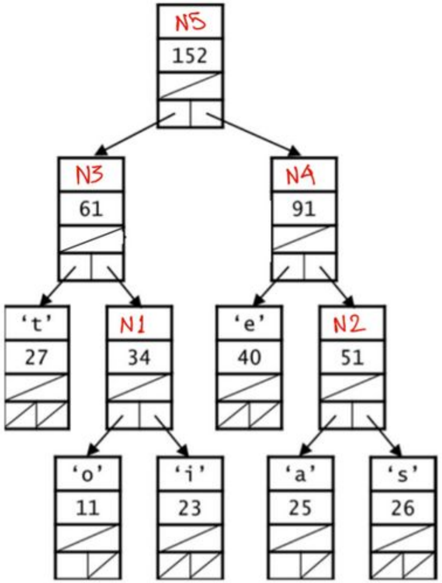
Build the huffman tree:

3. Add merged node back to sorted linked list.



Part II - Finding the Huffman Coding

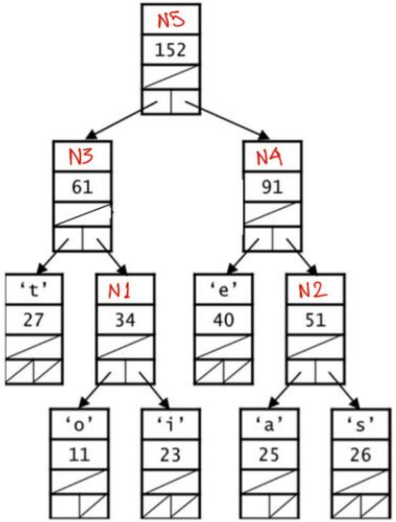
Build the huffman tree:
4. Repeat the previous 2 steps until tree is entire created.

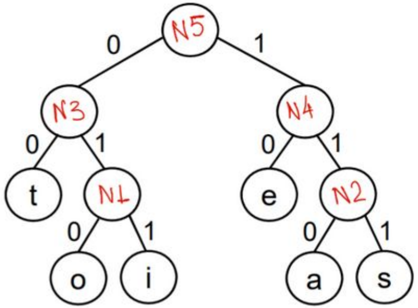


Part II - Finding the Huffman Coding

Obtaining the huffman encoding:

Left child edge gets 0 value,
Right child edge gets 1 value





Code Templates

```
class Node {
        char * symbol ;
        int frequency ;
        Node * next ;
        Node * left ;
        Node * right ;
}
```

Algorithm 1: TreeTraverse

Input: T: Root Node of Tree, C: prefix code, H: Array for storing all Huffman Codes

if *T is a leaf* **then**

    H[T.symbol] = C;

else

    TreeTraverse(N.left, concat(C,0));

    TreeTraverse(N.right, concat(C,1));

end

Code Templates

Algorithm 2: Encode a message

```
Input: H: Huffman Codes, encodedMessage
Output: R: Encoded Message
R = "";
for c in encodedMessage do
    R = concat(R, H[c])
end
```

Algorithm 3: Decode an encoded message

```
Input: S: Symbol Alphabet, H: Huffman Codes, encodedMessage
Output: R: Decoded message
cache = "";
R = "";
for char b in encodedMessage do
    // b can be '0' or '1' only;
    cache = concat(cache, b);
    for s ∈ S do
        if cache == H[s] then
            R = concat(R,s);
            reset cache;
        end
    end
end
```

Solution Walk through

Practice Problems

1. Kth smallest element in BST
2. Is binary tree a BST?
3. Merging BST

Kth Smallest element in BST

Inorder travel and return kth element of array

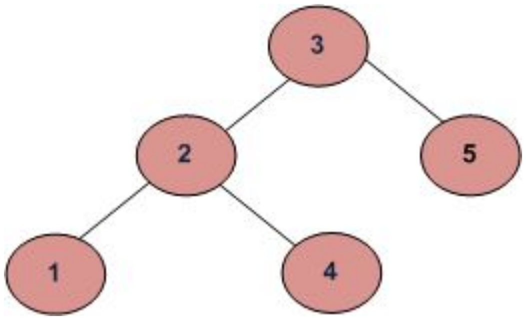
Time Complexity: O(n)
Space Complexity: O(h)

Is Binary Tree BST?

Check if left child is smaller than self and if right node is larger than self

Is Binary Tree BST?

Check if left child is smaller than self and if right node is larger than self



Is Binary Tree BST?

Recursive Method 1:

Check if max value in left subtree is less than current node value
Check if min value in right subtree is more than current node value

Is Binary Tree BST?

Method 2:

Check if Inorder traversal is sorted

Time Complexity: O(n)
Space Complexity: O(n)

Is Binary Tree BST?

Method 2:

Keep track of range allowed for a particular node

Time Complexity: O(n)
Space Complexity: O(1)

Merging BST

```
Input:      3            4
           / \          / \
          2   5        1   6
Output: [1, 2, 3, 4, 5, 6]
```

```
Input:      3            6
             \          / \
              4        2   10
Output:[2, 3, 4, 6, 10]
```

Merging BST

Naive Method:

Inorder Traversal on both trees and store them in arrays

A = [2,3,5]

B = [1,4,6]

Merge the two arrays into a resultant array by appending and sorting

Time Complexity: O(n log n)   Sorting takes O n log n

Space Complexity: O(n)

```
Input:       3              4
            / \            / \
           2   5          1   6
Output: [1, 2, 3, 4, 5, 6]
```

Merging BST

Can we perform faster merging of arrays?

Merging BST

Sorted Merge:

Inorder Traversal on both trees and store them in arrays

A = [2,3,5]

B = [1,4,6]

A, B will always be sorted. Merge them

Time Complexity: O(n)   Merging takes O n

Space Complexity: O(n)

```
Input:       3              4
            / \            / \
           2   5          1   6
Output: [1, 2, 3, 4, 5, 6]
```

Merging BST

Can we improve time complexity?

Can we improve space complexity?

Merging BST

Stacks:

Inorder Traversal on both trees and store them in arrays

A = [2,3,5]

B = [1,4,6]

A, B will always be sorted. Merge them

Time Complexity: O(n)   Merging takes O n

Space Complexity: O(n)

```
Input:       3              4
            / \            / \
           2   5          1   6
Output: [1, 2, 3, 4, 5, 6]
```