CSS 343 – Data Structures, Algorithms, and Discrete Mathematics II
Professor Clark F. Olson
Lecture Notes 2 – Huffman coding and binary search trees

Reading: Carrano, Chapter 16

Huffman coding

An important application of trees is coding letters (or other items, such as pixels) in the minimum possible space using Huffman coding. Each letter (or item, in general) is represented using a string of bits. For example, we could choose to represent the letter *t* as 0, letter *e* as 1, and letter *s* as 01. What is 01010? It could be "test," but it could also be "stet," or a number of other words (or nonsense.)

It is important to choose the bit strings such that the coding results in a unique word. If we choose *t* to be 0, *e* to 10, and *s* to be 11, then 010110 can only represent the word "test." These are called prefix codes. No letter can have a code that is a prefix for another letter's code. This idea is used in Huffman coding to represent strings using the minimum possible number of bits, assuming that we know the frequency of each letter before we start. The algorithm works like this:

At the start, each letter is a small tree with a single node and has an associated weight (the frequency.)
Repeat until all nodes form a single tree:
  Select the two trees with the smallest weights.
  Merge these trees into a new tree by adding a node that is the parent of both.
  The weight of the new tree is the sum of the weights of the two previous trees.
  Assign a *0* to one branch of the tree and a *1* to the other branch of the tree.

When finished the sequence of *0*s and *1*s necessary to get from the root to any letter is the representation for the letter. Letters with higher frequency will have shorter representations.

Note that a set of nodes in multiple (unconnected) trees is called a **forest**. This is the situation we have until the final iteration when the last two trees are connected.

Example: Let's say we have 8 letters with the following frequencies:

A	20
E	32
L	14
I	11
M	16
O	12
S	7
T	3

What tree do we get?

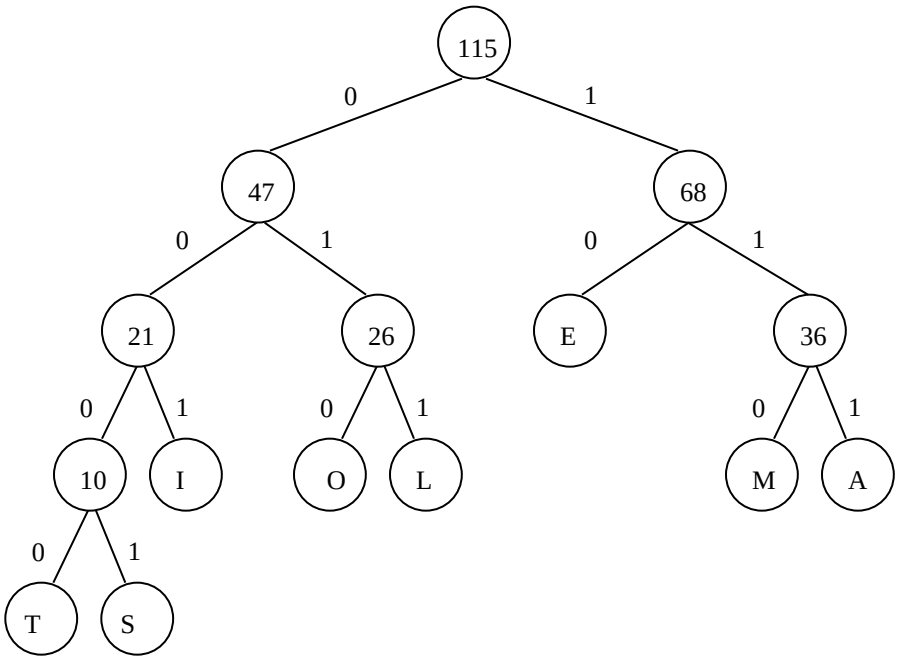
Steps:

Merge TS:	weight 10
Merge (TS)I:	weight 21
Merge OL:	weight 26
Merge MA:	weight 36
Merge (TSI)(OL):	weight 47
Merge E(MA):	weight 68
Merge (TSIOL)(MAE):	weight 115

Final representation (if lower frequency is 0 each time)

A	111
E	10
L	011
I	001
M	110
O	010
S	0001
T	0000

Here is the final resulting tree:



What is the code for TEST? 00001000010000
What is the code for LEAST? 0111011100010000

Although we won't prove it, this is guaranteed to be the shortest binary representation for the input letters (with the appropriate frequency.) It would require 64 bits for the 32 E's, 60 bits for the 20 A's, etc. (323 total bits for 115 letters.) Improvement can sometimes be achieved with other techniques (e.g., encoding multiple letters at a time, run-length encoding, lossy compression.)

What is the efficiency of determining the codes? At each step, we need to find the two trees with the smallest weight. A naïve algorithm would require O(n) time to find the two smallest and, thus, $O(n^2)$ overall. A better algorithm can be developed using a priority queue (which we'll cover shortly). This method requires $O(n \cdot \log n)$ time.

If the letters are already sorted by frequency, we can do even better. We maintain two queues, one contains trees with individual letters (in order of weight) and one contains larger trees (also in order of weight). The queue storing larger trees is initially empty. At each step, we must consider only the first two elements in each queue in order to decide which two trees to merge. The new tree is then put at the end of the queue storing larger trees. Each loop iteration can be performed in O(1) time and the entire algorithm now requires O(n) time.

So, we would start with T S I O L M A E in the letter queue and an empty tree queue. The sequence of queue states would be:

Letter queue	Tree queue
TSIOLMAE	<empty>
IOLMAE	(TS)
OLMAE	(TSI)
MAE	(TSI) (OL)
E	(TSI) (OL) (MA)
E	(MA) (TSIOL)
	(TSIOL) (EMA)
	(TSIOLEMA)

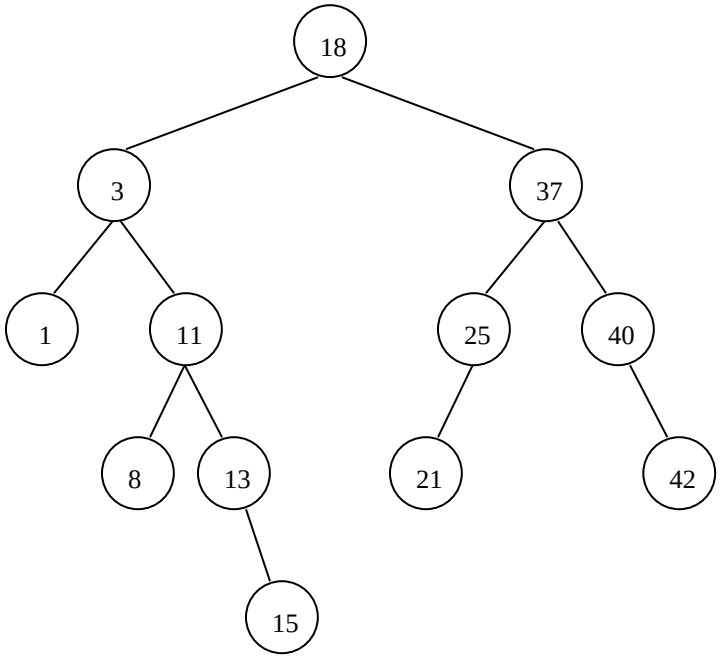
Binary search trees

A *binary search tree* is a special type of binary tree. For each node *n* in a binary search tree, the value of the item stored by *n* is greater than all of the values in the left subtree and less than all of the values in the right subtree. This implies that **all subtrees must be binary search trees**. This type of tree is useful for performing fast lookup (like binary search) and can be used to sort arrays. The primary advantage of this structure over an array is that you can efficiently insert and delete items. One disadvantage is that you don't have direct access to the *i*th element in the sorted list. Note that items that are equal can be put on the left or right or discarded, depending on the application.

In a general binary tree, it is time consuming to find a particular item, since we may need to traverse the entire tree to find the item that we are looking for. This is much easier in a binary search tree, see we know which subtree the item must be in for any ancestor node. This is similar to the difference between sequential search and binary search, except that a general binary search tree does not guarantee that the problem size is halved (one subtree may be much bigger than the other one).

Creating a binary search tree

Let's create a binary search tree storing the following values: 18 3 37 11 25 21 40 8 13 1 42 15



Programming a binary search tree

Once we have a binary search tree, we can find an item efficiently (usually) by taking the appropriate branch of the tree at each step:

```
const TreeData * retrieve(BinaryTreeNode *root, const TreeData &key) {
        if (root == NULL) return NULL;
        else if (key == *root->item)
                return root->item;
        else if (key < *root->item)
                return retrieve(root->leftChild, key);
        else
                return retrieve(root->rightChild, key);
}
```

Inserting an item is straightforward:

```
void insert(BinaryTreeNode *&root, TreeData *item) {
        if (root == NULL) {
                root = new BinaryTreeNode;
                root->item = item;
                root->leftChild = NULL;
                root->rightChild = NULL;
        }
        else if (*item < *root->item)
                insert(root->leftChild, item);
        else
                insert(root->rightChild, item);
        // What if they are equal?
}
```

This can be done non-recursively, but the code is messier:

```
void insert(BinaryTreeNode *&root, TreeData *item) {
        BinaryTreeNode *node = new BinaryTreeNode;
        node->item = item;
        node->leftChild = NULL;
        node->rightChild = NULL;

        if  (root == NULL)
                root = node;
        else {
                BinaryTreeNode *cur = root;
                while (true) {
                        if (*item < *cur->item)
                                if  (cur->leftChild == NULL) {
                                        cur->leftChild = node;
                                        return;
                                }
                                else cur = cur->leftChild;
                        else
                                if (cur->rightChild == NULL) {
                                        cur->rightChild = node;
                                        return;
                                }
                                else cur = cur->rightChild;
                }
        }
}
```

When you want to remove an item from a binary search tree, finding the item is easy, but deleting it is a little trickier if the node has two children. If the node has zero or one children, we can delete the node easily and replace the pointer to it (with the child if one exists.) If the node containing the item has two children, we must find a replacement item to place in the node. This replacement item is either the largest descendant of the left child or the smallest descendant of the right child (which are the next smaller and next larger item in the tree.)

```
bool deleteNode(BinaryTreeNode *&root, const TreeData &item) {
        if (root == NULL)
                return false;
        else if (item == *root->item) {
                deleteRoot(root);
                return true;
        }
        else if (item < *root->item)
                return deleteNode(root->leftChild, item);
        else
                return deleteNode(root->rightChild, item);
}

void deleteRoot(BinaryTreeNode *&root) {
        if (root->leftChild == NULL && root->rightChild == NULL) {
                delete root->item;
                delete root;
                root = NULL;
        }
        else if (root->leftChild == NULL) {
                BinaryTreeNode *tmp = root;
                root = root->rightChild;
                delete tmp->item;
                delete tmp;
        }
        else if (root->rightChild == NULL) {
                BinaryTreeNode *tmp = root;
                root = root->leftChild;
                delete tmp->item;
                delete tmp;
        }
        else {
                delete root->item;
                root->item = findAndDeleteSmallest(root->rightChild);
        }
}


// Pre-condition: root is not NULL
TreeData *findAndDeleteSmallest(BinaryTreeNode *&root) {
        if (root->leftChild == NULL) {
                TreeData  *item = root->item;
                BinaryTreeNode *tmp = root;
                root = root->rightChild;
                delete tmp;
                return item;
        }
        else {
                return findAndDeleteSmallest(root->leftChild);
        }
}
```

Binary search tree efficiency

The efficiency of operations on a binary search depends on how balanced the tree is when it is built. Like quick sort, if a value that is smaller (or larger) than all of the current elements is inserted at each step, then we will get poor efficiency. We will discuss balanced binary tree structures that deal with this problem in a few weeks. In the worst case, insertion, deletion, and retrieval take O(n) time, where n is the number of nodes in the tree. In the best case, it is possible to perform the operations in O(1) for an unbalanced tree, if we get lucky. Ideally, the tree will be perfectly balanced (for each node, the two subtrees will never have a height difference greater than one). In this case, the height of the tree is O(log n) and insertion, deletion, and retrieval can all be performed in O(log n) time, since, at worst, they require traversing to a leaf of the tree. It turns out that the average case (if it is assumed that each relative position in sorted order is equally likely at each step), is also O(log n). However, special cases, such as arrays that are already sorted, can cause problems. Traversal of the tree is always O(n), since you have to visit each node.

Sorting is easy with the help of a binary search tree. Simply insert each item into the tree and read them back out using an inorder traverse. How long does this take? In the worst case, it will require O(n^2) to insert them all and O(n) to read them back out. It the best/average case, it requires O(log n) to insert each one and O(n) to read them back out for a total of O(n log n). Tree sort is very similar to quick sort. The array is essentially partitioned by the side of the root on which that each element lies.

Practice problems (optional):

Carrano, Chapter 15:

#9, #15

Carrano, Chapter 16:

#4, #18