Tree applications – Huffman Encoding and Binary Space Partition Trees

Professor Clark F. Olson (with some edits by Carol Zander)

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 be 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?

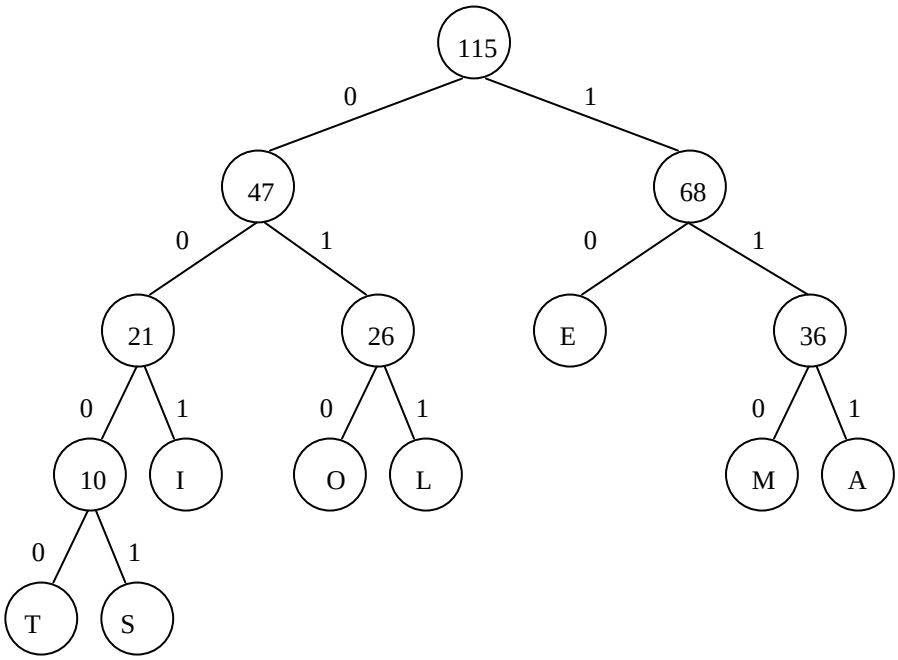
Steps:

```
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

```
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 (covered shortly). This method requires O(n·log n) time.

If the letters are already sorted, we can do even better. We read the letters out in order and each time we construct a new tree we put it at the end of a queue of trees. At each step, the two trees to merge must either be the first two individual letters, the first two trees in the queue, or one of each.

So, we would start with: `T S I O L M A E`
A second queue is constructed with the following sequences of states:

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

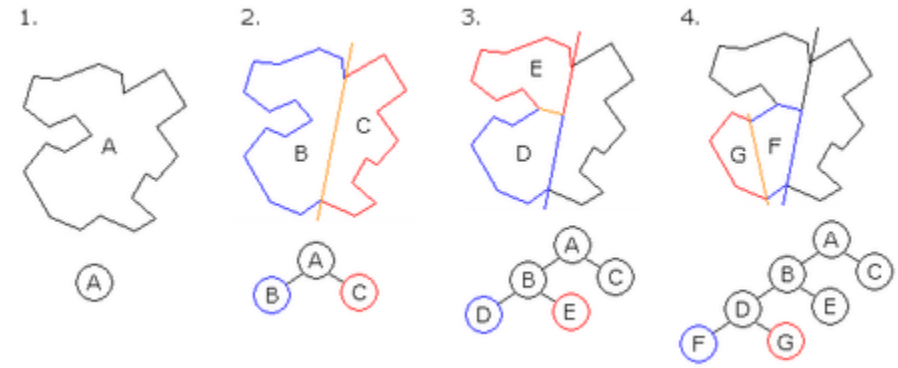
Binary space partitions

Information in the lecture was garnered from the following web pages:
http://en.wikipedia.org/wiki/Binary_space_partitioning
http://web.cs.wpi.edu/~matt/courses/cs563/talks/bsp/document.html

The demo can be found at:
http://symbolcraft.com/graphics/bsp/

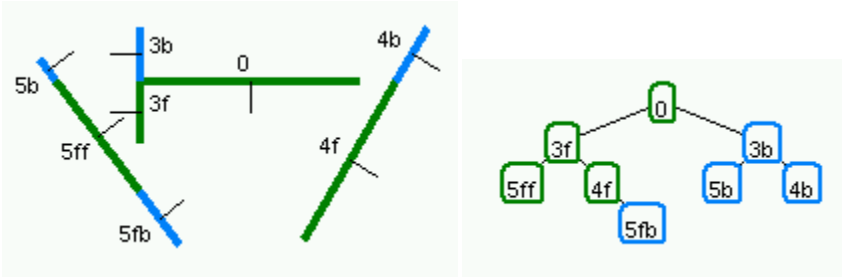
Binary space partitions are an interesting application of trees that has been used in rendering for computer graphics, collision detection for mobile robots, and computer games (it was used in the Doom engine). The basic idea is to recursively partition the space of interest (e.g., the game field) into smaller and simpler polygons (or polyhedra, in general) until each partition is empty of objects other than on the partitioning boundaries.

Here is a picture from wikipedia:



1. A is the root of the tree and the entire polygon.
2. A is split into B and C.
3. B is split into D and E.
4. D is split into F and G, which are convex and hence become leaves on the tree.

Here one that is a bit different from Matthew Ward's web site:



Modifying the code from that website some, we can see roughly how these work. I assume here that we have a Segment abstract data type already written with some useful methods.

```
struct BSPNode {
      vector<Segment *> partition;
      BSPNode *front, *back;
}

BSPNode* createBSPTree(vector<Segment *> s) {
      if(s.size() == 0) return NULL;

      BSPNode *root = new BSPNode;

      // assume select_partition written
      Segment *partition = select_partition(s);
      vector<Segment *> front, back;

      for(int cur = 0; cur < s.size(); cur++) {
            if (partition->coincident(s[cur])
                  root->partition.push_back(s[cur]);
            else if(partition->insect(s[cur]) {
                  Segment *sfront, *sback;
                  s[cur]->split(partition, sfront, sback);
                  front.push_back(sfront);
                  back.push_back(sback);
            }
            else if(partition->front(s[cur])
                  front.push_back(s[cur]);
            else
                  back.push_back(s[cur]);
      }
      root->front = createBSPTree(front);
      root->back = createBSPTree(back);
}
```

One interesting question is: How should we select the partition? We want to have a BSP tree with as few nodes as possible. In general, this means that the partition should intersect as few other segments as possible. However, owing to ties and other issues, we can't always predict which partition will be best. It is common to try multiple partitions are select which results in the smallest tree.

We can then traverse the BSP tree in order to render what can be seen from a particular location. We render the objects on the other side of each partition from us first, since the painter's algorithm relies on further away objects being drawn first and then the closer objects are drawn over them.

Summary of trees

Trees are a very useful data type for some collections of data. Examples include hierarchical data, such as mathematical expressions and data that must be accessed quickly that cannot be stored in an array (for example, due to the fixed array size), since linked lists are time consuming to traverse. We have examined three ways to traverse a tree (inorder, preorder, and postorder). Trees are typically implemented using pointers (although array implementations are possible). Binary search trees allow us to insert, delete, and locate items in a container efficiently, except for pathological insertion orders. Later we will examine techniques that guarantee efficiency no matter what the order of insertions is.