Section Notes 7

Computer Science 50

October 22, 2009

1 Bit Strings

As you know, *everything* in your computer is represented as a string of 0s and 1s. Usually the machine interprets such *bit strings* as numbers, or addresses, or characters, or combinations of those that form more complicated data structures. But the machine also provides operations for treating integers as simple sequences of bits. It's kind of like treating an `int` as an array of 32 `bool` values, except that you can't access the `i`th element using the notation `s[i]`. But the advantage is that the bit string is packed much more efficiently than an actual array of `bool`s, and you can operate on the elements of bit strings in parallel using a single machine operation.

For example, in UNIX/Linux, the low-level system call for opening a file is named `open`, and you typically call it like this: `open("errors.log", O_CREAT | O_TRUNC | O_WRONLY)`. The first argument is the name of the file to be opened. The second argument is an integer that represents a set of *flags*, or Boolean properties. `O_CREAT` means create the file if it doesn't exist. `O_TRUNC` means truncate it to zero length if it does exist. And `O_WRONLY` means don't allow it to be read until it's closed. The *bitwise* operator "`|`" combines these properties into an integer that represents the set of all three flags, and passes that set as a single, compact parameter to `open`. The implementation of `open` uses other bitwise operators to test its `flags` argument. For example, the statement `if (flags & O_CREAT) { ... }` will test whether the file should be created. The result of the `&` expression is non-zero if `O_CREAT` is in the `flags` set, and zero otherwise.

Operations like "`|`" and "`&`" are called "bitwise" because the computer determines the value of each result bit only by looking at the corresponding pair of bits in the operands (the arguments of the operation). The system flags used in this example are defined like this:

```
#define O_WRONLY            01
#define O_CREAT           0100
#define O_TRUNC          01000
```

The representation of each one has a single "1" bit, and no two are equal. The combination of the three flags in the argument to `open` shown above has the value 01101. That is, it includes the bits for all three flags. So this bit string represents a set that contains those three properties and no others. To test for the presence of a particular flag such as `O_CREAT`, `open` uses the `&` operator to check whether that flag's bit is set in its `flags` argument.

The following subsections describe the bitwise operations provided in C. For simplicity, we'll use 8-bit binary integers (e.g. 00101001 or 10100101) as examples, but these operations work on all of the integer types. When programming with these operations, it's important to distinguish the *logical* operations that we've learned already from these new bitwise operators. The logical OR of two integers (operator "`||`") produces true if either operand is non-zero. The bitwise or of two integers (operator "`|`") produces a new integer, each of whose bits is the logical OR of the corresponding bits in the two operands. Be careful not to confuse these closely related operations.

1.1 Bitwise AND

Bitwise AND is written `a & b`. Each output bit is 1 if the corresponding bits of `a` and `b` are both 1, and 0 otherwise. For example:

```
  a      01011011                                  a      10110101
  b      10001101                                  b      01010101
a & b    00001001                                a & b    00010101
```

1.2 Bitwise OR

Bitwise OR is written `a | b`. Each output bit is 1 if either of the corresponding bits of `a` and `b` is 1, and 0 otherwise. For example:

```
  a      01011011                                  a      10110101
  b      10001101                                  b      01010101
a | b    11011111                                a | b    11110101
```

1.3 Bitwise NOT

Bitwise NOT is a unary operator, written `~a`. Each output bit is 1 if the corresponding bit of `a` is 1, and 0 otherwise. For example:

```
  a      01011011                                  a      10110101
 ~a      10100100                                 ~a      01001010
```

1.4 Bitwise XOR

Bitwise XOR (exclusive OR) is written `a ^ b`. Each output bit is 1 if exactly one of the corresponding bits of `a` and `b` is 1, and 0 otherwise. For example:

```
  a      01011011                                  a      10110101
  b      10001101                                  b      01010101
a ^ b    11010110                                a ^ b    11100000
```

1.5 Summary Table

a	b	a & b	a | b	~a	a ^ b
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	0	0

1.6 Shift Operators

Two other operations on bit strings in C are often used in conjunction with the bitwise operators above. These are the left shift (`<<`) and right shift (`>>`) operators. The expression `s << i` treats integer `s` as a bit string, and `i` as a number of bits to shift that string to the left. Zero bits are inserted at the right, and bits shifted beyond the left boundary of the bit string are discarded.

For an unsigned integer `s`, the right-shift operation is exactly the opposite, with zero bits entering at the left and discarded bits dropping off the right to form the result. However, if `s` is a signed integer that happens to be negative, the bits entering the left are normally ones, rather than zeros, so that the sign of the result retains the sign of `s`. (Surprisingly, the language definition does not guarantee this behavior!)

Just as adding zeros at the right of a decimal number is equivalent to multiplying by a power of 10, the left-shift operation `s << i` amounts to multiplication of `s` by $2^{\mathtt{i}}$. Similarly for right-shift and division.

2 Hash Tables

As we have seen previously, linked lists are data structures that support efficient *insertion* and *deletion* of elements. We can accomplish these operations in constant time–time that is unrelated to the number of elements in the data structure. However, on average, each *lookup* operation takes $\frac{n}{2}$ comparisons, where n is the number of items in the list. Thus, as the list grows longer, the average time to look up an element takes proportionally longer.

How can we build a lookup function that doesn't have to loop over all of the elements stored in our data structure? One option would be for us to change our data structure so that something about the structure of a datum tells us where to find it. A hash table does exactly this. A hash table is a data structure where insertion, deletion, and lookup of data can each be accomplished in nearly constant time (on average). The data structure isn't perfect, however. While insertion, deletion, and lookup are very fast, hash tables do not easily support ordering or sorting of the stored data. Still, a hash table is a good data structure for a spell checker, since we don't have any real reason to print out the entire list of words in our dictionary.

To build a spell checker, we will want our hash table to store string values. You should however realize that the data structure can be used for storing other types of data as well.

2.1 Hash Table Structure

Hash tables are basically arrays coupled with a function (called the *hash function*) for mapping the values of the stored objects to indices in the storage array (or table). For our task, we will define a hash function that maps a string to an integer (also called a *hash code*). Since we will use the hash code as an index into a C array, we want it to fall in the range `[0, TABLE_SIZE - 1]`, where `TABLE_SIZE` is the number of elements in the array representing our hash table. An easy way to do this is to calculate the initial hash code independently of this range, and then take the initial value modulo the table size.

2.2 Hash Functions

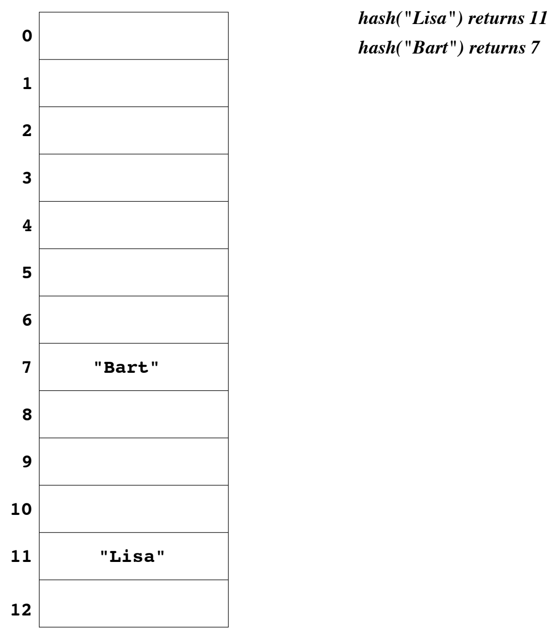
The simplest definition of a hash function is some function that maps a (not necessarily integral) data type onto integers.
There are infinitely many possible hash functions, but not all of them are necessarily useful or good.[1] An example of a very simple hash function on strings could be adding the ASCII values of all the letters in the string, and then mod'ing by the size of the hash table. Here it is, in pseudocode:

```
for(i goes from 0 to length(str)-1) {
    sum += str[i]
}
return sum % HASH_TABLE_SIZE
```

We hope that, by this point, you are able to write this function in real C very quickly.

To store a string in a hash table, one merely puts the string in the array at the index given by its hash code. For instance, if we had a hash table of size 13, and the string "Lisa" has hash code 11 and "Bart" has hash code 7, then the hash table would look as follows:

[1]The course staff encourages you to search the Internet for good (evenly and randomly distributing) hash functions, which you may use or modify in your code. If you choose to use a publicly available hash function, however, you must cite the location in a comment.



To see if the string “Bart” is in the hash table, one computes the hash code for “Bart” (7), and sees if “Bart” is stored at that index in the array. Note that neither insertion nor lookup involve iterating over the elements in the array in this example.

To have a good hash table, we’d like to have a large number of different possible hash codes, which are all generated by our hash function. A good hash function has several properties:

1. The hash code is fully determined by the string being hashed. (And nothing else.)
2. The hash code for two identical strings is the same. (No “random numbers” involved in the calculation.)
3. The hash function uses the whole string. If we only used some part of the string, then we wouldn’t be maximizing the number of possible hash codes and our table would not be very good.

   For example, if we had the two strings “happy” and “happiest”, we would like them to hash to different places in the table. But if we calculated the hash code based on only the first few letters of the strings, the hash code for both these strings would be the same, and the strings would occupy the same location in our hash table.
4. The hash function “uniformly” distributes the strings in the hash table.
5. The hash function can generate very different hash codes for similar (but not the same) strings.

2.3 Collisions

The prior section referred to an ideal hash table because, when the hash function is applied to different strings, the strings may happen to hash to the same value (especially with a table size of only 13). Obviously, a single array location can hold only a single value. When two values produce the same hash code, we have what is called a *collision*.

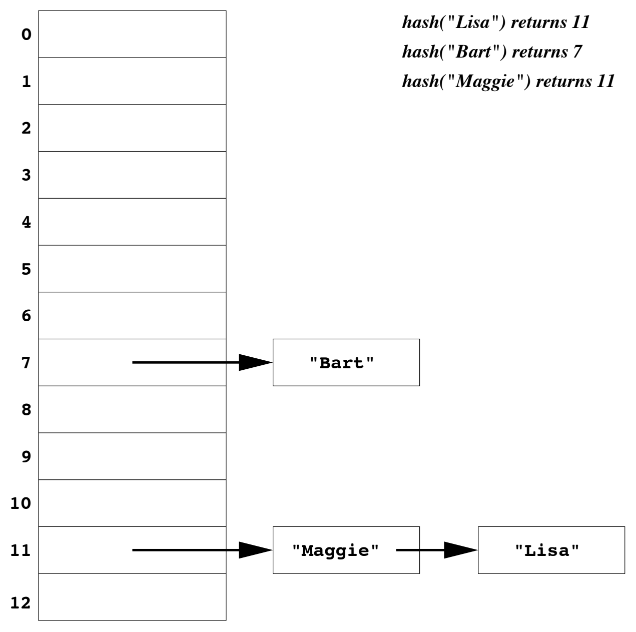
There are numerous methods for handling collisions in a hash table. Two of the most common methods are discussed here. The first method, called *separate chaining*, leverages what you have learned about linked lists. The second method, called *linear probing*, removes the need for dynamically allocated storage.

2.4 Implementation 1 — Separate Chaining

In *separate chaining*, each element of the hash table is a pointer to the head of a linked list.[2]

In the picture below, “Bart” has a hash code of 7, and “Lisa” and “Maggie” both have a hash code of 11. The hash table is of size 13.

[2]Combining data structures FTW!



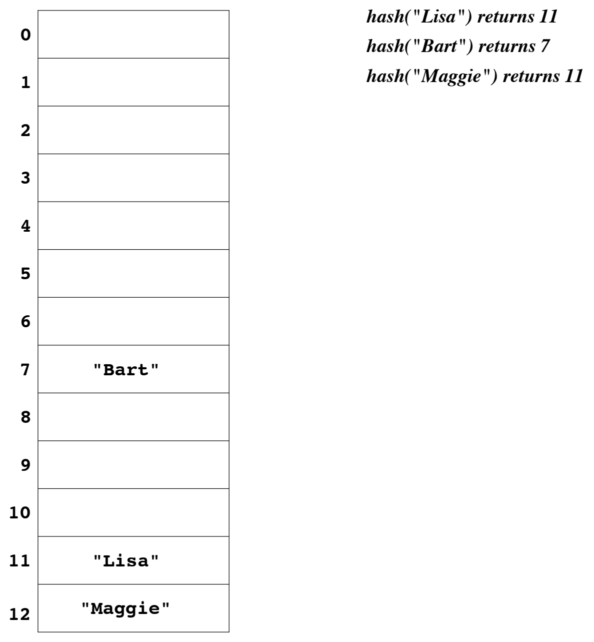
The implementation of insertion first hashes the string. The list at the computed hash code is checked to see if the word already exists in the table. If it doesn't, the word is inserted at the head of the linked list.

The lookup process first hashes the string, and then searches through the linked list starting at the table element associated with the resulting code. If it finds the string, lookup is successful; if not, the lookup fails.

2.5 Implementation 2 — Linear Probing

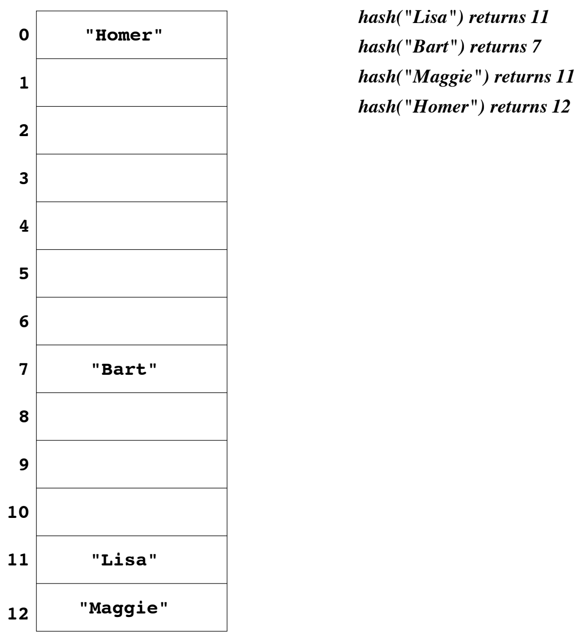
An alternative implementation for a hash table uses a technique called *linear probing*. In this method, the elements of the array hold single values, not linked lists of them. If a collision occurs, the algorithm tries array elements sequentially until an empty cell is found.

In the picture below, "Lisa" has been inserted first, with a hash code of 11, "Bart" was inserted with a hash code of 7, and finally "Maggie" was inserted with a hash code of 11.



This technique is subject to a problem called clustering. Once a collision occurs during insertion, there will be two table cells next to each other that are occupied. Afterwards, an insertion that collides with either of those cells will cause that cluster to grow. Now the probability is even larger that another collision will cause the cluster to grow.

The picture below shows what happens when "Homer" is inserted, with a hash code of 12.

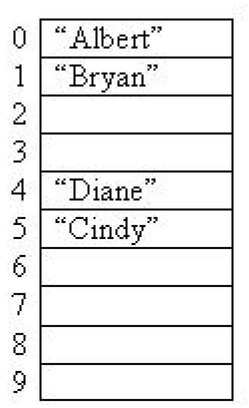


The notion of linear probing is only one method of "probing" through the list for the next available slot. Another common approach is *quadratic probing*. Instead of searching for the next available cell, it checks the cell just beyond the current cell. If that cell is not free, it then checks the cell 4 (2^2) beyond that, then 9, 16, 25, ...

There are a couple other caveats we should mention. If you apply the linear probing model to resolve conflicts, you will need to be careful that your program doesn't get into an infinite loop when the table fills up. (Why don't you have this problem with a chaining model?) With quadratic probing, you could even find yourself with a relatively sparsely populated hash table (i.e., there are empty cells), but still be unable to insert a string. How so? Consider a hash table of size 10, and suppose we have a *really* poor hash function, such as:

```
int
hash_function(char *word)
{
    return 0;
}
```

Initially, we have inserted the strings "Albert", "Bryan", "Cindy", and "Diane" (in that order) into the hash table, so that it looks like this. (Remember that we are using a quadratic probing model...so make sure you understand why the table looks like this. If you are confused, ask a TF.)



0	“Albert”
1	“Bryan”
2	
3	
4	“Diane”
5	“Cindy”
6	
7	
8	
9	

If we try and insert the string “Edward”, we have to wait until we are probing $1+4+9+16+25+36+49+64+81+100+121$ cells in advance (“Edward” will be in cell 6). “Francine” requires us to go $1+4+\ldots+169$ cells in advance (“Francine” will be in cell 9), and so on. Admittedly, we picked an unrealistically bad hash function to illustrate a point. Even so, the potential inefficiency is significant, and it’s not as easy to avoid an infinite loop as in the case of linear probing.

3 Trees

Trees are an extremely useful data structure that are a slight modification of something you’ve seen before, but in many ways are much more powerful and give us the ability to do a lot of really neat stuff.

3.1 Binary Trees and Search Trees

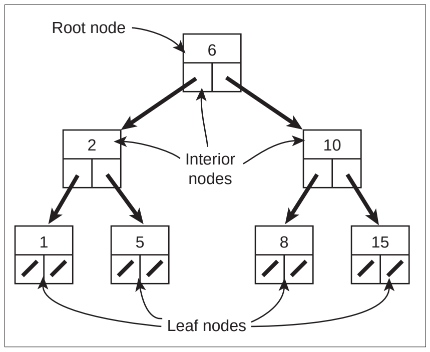


Figure 1: A Binary Search Tree

Note that, as we see in the above image, the structure of a *node* in a binary tree is very similar to an element of a doubly linked list: we have two node pointers and some data. As a *data structure*, however, the binary tree is quite different from a linked list. This difference arises from the meaning we assign to the fields in the nodes (`left` and `right` *children*, instead of `prev` and `next` elements), and by enforcing that meaning through the operations we are allowed to perform (i.e., a binary tree node may not point at its parent with the left or right pointers; it must point at its children or nothing at all, `NULL`). Here is an example of a binary tree node implementation:

```
typedef struct node
{
    value val;
    struct node *left;
    struct node *right;
} node;
```

Binary trees are great when you want to use binary search. If you build up the binary tree such that for each node, every value on its left child's side is less than the node's associated value of the node, and every value on the right side is greater, then we can easily perform a binary search for a given value in the tree. We call a tree with this ordering of the nodes a *binary search tree.* Not only the search algorithm itself, but also the operations for updating the tree run in time proportional to the height of the tree, i.e., the distance from the root of the tree to its deepest leaf node.

The code for searching a binary tree looks like:

```
bool
binary_search(node *root, value val)
{
    if (root == NULL)
        return false;
    if (root->val == val)
        return true;
    else if (val < root->val)
        return binary_search(root->left, val)
    else
        return binary_search(root->right, val)
}
```

3.2 Trying Out "Tries"

An *associative array* or *dictionary* is a generalized array that allows you to associate values with keys that are more descriptive than simple integers, as with ordinary arrays. For instance, suppose you need a data structure that allows you to look up the records for a student starting from his or her 8-digit university ID. A simple array would need too many entries. You could use a hash table, extracting hash codes from either the numerical or the string representation of the student IDs. Or you could build a tree, placing the student records at the leaves of the tree. In such a tree, the paths from the root to a leaf would be labeled with digits of the ID. Each interior (non-leaf) node could therefore have as many as ten children. To find a student record, you would start at the root and use successive digits of the student's ID to choose successive child nodes until you reach the desired record or you reach a dead end (meaning that the student hasn't yet been inserted in the tree).

Such a tree is called a *trie.* In general, there may be data stored at interior nodes, not just at leaf nodes. Figure 2 shows a dictionary trie in which the keys are sequences of lower-case letters. The node data type for such a trie might look like this:

```
typedef struct trie {
    char *data;
    struct trie *children[26];
} trie;
```

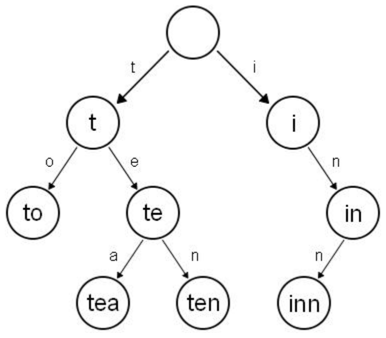


Figure 2: An trie that maps a word to itself.

Here the `data` field is `NULL` if the path to the node is not a valid word, and otherwise it represents the word in string form.

Tries can be quite efficient for lookup in terms of time, but they can also consume *lots* of space.

4 Data Compression

At some point, you've probably encountered a `.zip` file, more correctly termed a zip *archive*. What is an archive? An archive is a *compressed* version of some data. When you need to access the data, you can decompress the data and then you have access to it. In a world where our hard drives get bigger and bigger, you might think that it's not as necessary to zip things as it was before. But we argue that it's always important to be space efficient.

4.1 The Need for Data Compression

The world is full of information, lots and lots and lots of it. All of this information is stored on disk in the form of strings of 1's and 0's. While we often simply store these 1's and 0's unmodified on disk, this is not necessarily the most space efficient method for storing information. It turns out that we can often store an alternate representation which uses less disk space, and later recover our original data from this encoding.

One common and natural way to think of a file is as a sequence of characters. Since an 8-bit byte is customarily the smallest addressable storage unit in computers, it's natural to represent each character as an 8-bit value. That means a file of n characters will occupy $8n$ bits on disk. Alternatively, a file using the emerging Unicode encoding might use 16 bits to store each character, so the file will be twice as large. Since each of these representations uses a fixed number of bits to represent a character, each is an example of a *fixed-length encoding* of data. Files encoded under fixed-length encoding schemes are decoded by looking at file blocks of constant length (e.g., 8-bit or 16-bit blocks).

4.2 Huffman Encoding

Understanding how to Huffman encode and decode is a relevant skill, especially if you are interested in keeping all your CS50 projects and trying to remain under quota. This week we'll examine in detail the specific compression scheme that's used and the data structures that represent it effectively.

Unlike ASCII, Huffman encodings are *variable-length encodings*, meaning that each character can have a representation of different length. If we can somehow figure out which characters occur most often, and encode them with fewer than 8 bits, we might be able to save some space over ASCII. However, if we do this, then we will have to code some of the less frequently occurring characters with more than 8 bits. Ideally, we'd like to minimize the total number of bits used.

4.2.1 Prefix Property and Immediately Decodable Encodings

There are many ways to design a variable-sized encoding of data, but Huffman encoding has another property that will prove useful, the *prefix property.* This means that no bit representation for any of the characters is a prefix of any other character's representation. An encoding with the *prefix property* is said to be *immediately decodable.* Why is this fact so crucial? Consider the following example: If we use the coding in Figure 3 to

A	0
B	1
C	01
D	11

Figure 3: **Not** a prefix code

encode "CAB", we would get the bit–string 0101. However, presented with the bit–string 0101, there is no way to tell whether it represents "ABAB", "CAB", "ABC", or "CC". When decoding information encoded using this representation, it is unclear whether the first 0 should be interpreted as an 'A' or as the first part of a 'C'. The coding in Figure 4 is unambiguous. The bit–string 010100 can *only* represent "ABBA", and the bit–string 1110111 can *only* represent "DAD" using this encoding. (Confirm this yourself.)

A	0
B	10
C	110
D	111

Figure 4: A prefix code

4.2.2 Building the Huffman tree

We will use a binary tree to construct Huffman codes for each character. First, we must figure out how often each character occurs in the file by building a *frequency table*. Next, we create a forest of trees where each tree is a single node containing a character and the frequency of that character.

Then we loop, until the forest contains just one tree:

1. Remove the two trees from the forest that have the smallest frequencies in their roots.
2. Create a new node, to be the root of a new tree with the two trees in step 1 as its children. The frequency (actually, character count) of this parent node is set to the sum of the counts of the two children.
3. Insert this new tree back into the forest.

4.2.3 Properties of Huffman Trees

Trees created using the above algorithm have several interesting properties.

1. Characters are at the leaves of the tree. The path to each leaf is necessarily unique, so the paths from the root to a character can be used to code that character.
2. It can be proved that this method produces encodings that have the prefix property and that these encodings produce the smallest possible file size for this class of encodings. Such a proof is beyond the scope of this course, and interested students are encouraged to read Huffman's original paper on the topic.[3]
3. Since the frequency field of any internal node contains the sum of the frequencies of its two children, the frequency field of the root node will contain the total number of characters in the file.
4. One can find the final length of an encoded file in bits by summing the frequency fields of all internal nodes. Note that this length does not include the bits required to store the header structure.

4.2.4 Example of Building a Huffman Tree

As an example, suppose we had a file whose character frequencies are given by Figure 5. We encode it as

A	10
B	20
C	15
D	18
E	30
F	19
G	6
H	5

Figure 5: A frequency table

demonstrated in Figure 6. The Huffman codes we get by walking through the paths of the tree are given in Figure 7. On an intuitive level, we can think of the "combining the first two trees in the forest" operation

[3]You can find it here as of the writing of this document: http://compression.graphicon.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf

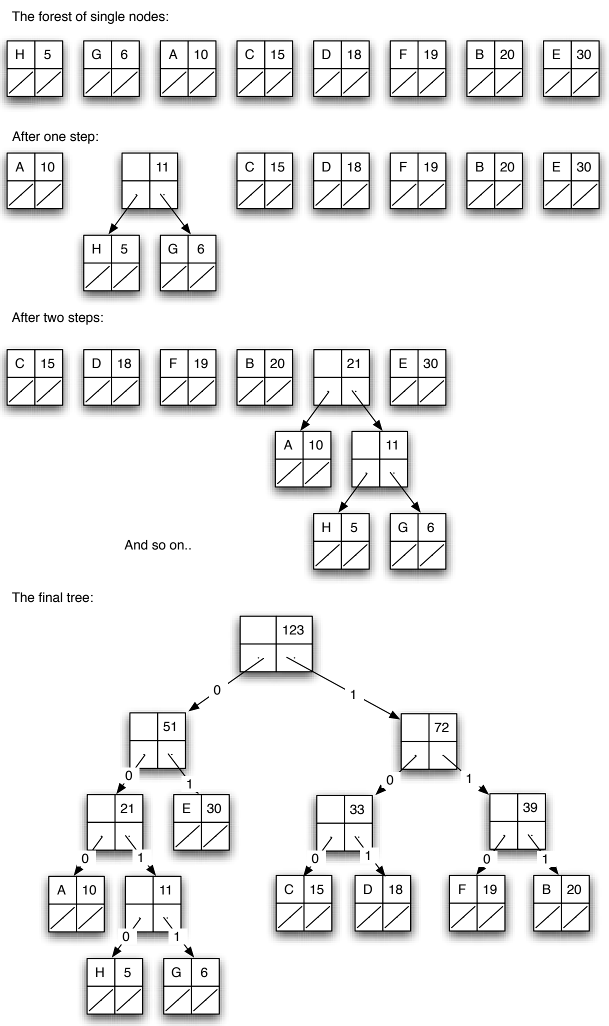


Figure 6: Huffman tree

as a penalty against those two trees — they were less frequent than the others, so they get an extra parent node added as a result. This pushes them further away from the root node and creates a longer path to their leaves, giving those (less frequent) characters a longer encoding. The rightmost trees are more frequent, and so should only have another level of path added in front of them as a last resort.

4.2.5 Decompressing using the Huffman tree

Since the encoding is simply the path that was taken through the tree to reach the required leaf node, we can start at the top and use the bits of the encoded stream to tell us whether to take the left or right branch. Thanks to the prefix property, we know we're done when we reach a leaf node (two NULL children), and we can start decoding the next character by returning to the root of the Huffman tree and repeating.

A	000
B	111
C	100
D	101
E	01
F	110
G	0011
H	0010

Figure 7: Huffman coding for the example