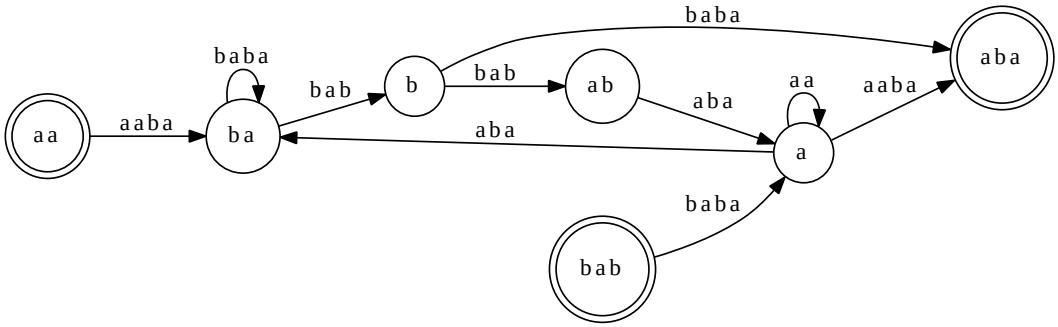
CMPS 340 File Processing
Test #2 Fall 2016 Sample Solutions to selected problems
Dr. McCloskey

1. Consider the set of codewords $C = \{aa, aaba, aba, bab, baba\}$, which is not uniquely decipherable.

(a) Draw a picture of $SP(C)$ (the Sardinas-Patterson graph for C).

Solution:



(b) Identify a path in the graph that begins in a codeword node and ends in a codeword node, and show the corresponding prime disagreeing pair of factorizations.

Solution: The shortest such path is $bab \xrightarrow{baba} a \xrightarrow{aaba} aba$, which gives rise to this prime disagreeing pair of factorizations: $bab|aaba$ vs. $baba|aba$.

Another short path going from one codeword node to another is $aa \xrightarrow{aaba} ba \xrightarrow{bab} b \xrightarrow{baba} aba$. It gives rise to this prime disagreeing pair of factorizations: $aa|bab|aba$ vs. $aaba|baba$

(c) Identify three strings x, y, and z such that, for all $i \geq 1$, xy^iz (i.e., x, followed by i occurrences of y, followed by z) has a prime disagreeing pair of factorizations.

Hint: Make use of a cycle in $SP(C)$.

Solution: Suppose that we follow any of the paths in the family

$$bab \xrightarrow{baba} \underbrace{a \xrightarrow{aa} a \xrightarrow{aa} \cdots \xrightarrow{aa} a \xrightarrow{aa}}_{2k \text{ times}} a \xrightarrow{aaba} aba$$

where $k \geq 0$. These paths gives rise to the prime diagreeing pairs of factorizations

$$\begin{array}{l} bab|\overbrace{aa|aa|\cdots|aa|}^{k\geq 0 \text{ times}}aaba \\ baba|\underbrace{aa|aa|\cdots|aa|}_{k\geq 0 \text{ times}}aba \end{array}$$

of the strings $bab(aa)^{k+1}ba$ ($k \geq 0$). So, to answer the question, we can choose $x = bab$, $y = aa$, and $z = ba$.

An alternative family of paths to use is

$$aa \xrightarrow{aaba} \underbrace{ba \xrightarrow{baba} ba \xrightarrow{baba} \cdots \xrightarrow{baba} ba \xrightarrow{baba}}_{2k \text{ times}} ba \xrightarrow{bab} b \xrightarrow{baba} aba$$

These give rise to the factorizations

$$aa|\overbrace{baba|\cdots|baba|}^{k\geq 0 \text{ times}}bab|abaaaba|\underbrace{baba|\cdots|baba|}_{k\geq 0 \text{ times}}baba$$

of the strings $aa(baba)^k bababa = aa(ba)^{2k+3}$ ($k \geq 0$). Here we can choose $x = aa$, $y = baba$, and $z = ba$.

In these two examples we exploited a so-called self-loop in the graph (i.e., an edge that connects a node to itself). Any cycle can be so exploited. For example, we could have made use of the cycle $ba \xrightarrow{bab} b \xrightarrow{bab} ab \xrightarrow{aba} a \xrightarrow{aba} ba$. That would have led to a much longer choice for y, however.

(d) There are two members of C such that, if we removed one or the other (but not both) of them, the set C' thereby obtained would still lack the property of unique decipherability. Identify at least one such member of C (but preferably both) and justify your choice.

Hint: A nontrivial path beginning and ending at codeword nodes induces a disagreeing pair of factorizations that involves only those codewords labeling the edges on the path, plus the labels of the nodes at the two ends of the path.

Solution: The premise of the question is incorrect, for which your instructor apologizes. Actually, there are *three* members of C —namely aa, $aaba$, and $baba$— such that removing any one of them results in a set of codewords that is still not uniquely decipherable. (The question incorrectly implied that there were only two such codewords.)

How can we tell that $C-\{aa\}$ is non-u.d.? The evidence comes from the path $bab \xrightarrow{baba} a \xrightarrow{aaba} aba$, which begins and ends in codeword nodes but does not involve aa at all, either as a node label or an edge label. It follows that we can construct a pair of disagreeing factorizations (which we did in answering (b) above) in which aa does not appear as a factor. Thus, $C - \{aa\}$ is not u.d.

Similarly, the path $aa \xrightarrow{aaba} ba \xrightarrow{bab} b \xrightarrow{bab} ab \xrightarrow{aba} a \xrightarrow{aaba} aba$ induces a pair of disagreeing factorizations in which $baba$ does not appear as a factor. Thus, $C - \{baba\}$ is not u.d.

Finally, the path $bab \xrightarrow{baba} a \xrightarrow{aba} ba \xrightarrow{bab} b \xrightarrow{baba} aba$ induces a pair of disagreeing factorizations in which $aaba$ does not appear as a factor. But wait! Neither does aa appear as a factor! Which means that $C-\{aa, aaba\}$ is not u.d., either! This contradicts the question, which implies that

removing any two codewords from C would yield a u.d. set of codewords. Again, apologies for that.

---

2. Consider the following hash table (where only keys are shown, with home addresses in parentheses), in which *linear probing* is used for resolving collisions. The table has buckets/cells of size one.

```
   0     1   2     3        4      5    6   7     8      9
+------+---+---+--------+------+------+---+---+------+-------+
| Gorn |   |   | Picard | Worf | Troi |   |   | Borg | Spock |
|  (9) |   |   |   (3)  |  (4) |  (3) |   |   |  (8) |  (8)  |
+------+---+---+--------+------+------+---+---+------+-------+
```

(a) What is the packing density (or load factor, if you prefer) α of this table?

Solution: The table has ten cells of which six are occupied, so the answer is $\frac{6}{10}$. In simplest form, this is $\frac{3}{5}$.

(b) Assuming that each key in this table is equally likely to be sought, what is the average number of probes (i.e., accesses to buckets) performed in a successful retrieval of a record?

Solution: For each key in the table, count the number of cells between its home address and the cell it occupies, inclusive. Add these up and divide by the number of keys in the table. Considering the keys occupying cells in the table from "top to bottom", we get $(2 + 1 + 1 + 3 + 1 + 2)/6$, which simplifies to $\frac{5}{3}$.

(c) Assuming that each of the ten addresses is equally likely to be the home address of a key that is sought, what is the average number of probes in an unsuccessful retrieval (i.e., a search for a key that is not present) in this table? Compare this figure with the number of probes we would expect (according to the analysis done in class) in a table having this one's packing density.

Solution: For each address in the table (that's zero through nine), count the number of cells between it and the next empty cell, inclusive. (Each of these cells would be "probed" during a search for a non-existent key with that particular home address.) Add these up and divide by the number of cells. Considering the addresses from "top to bottom", we get $(2 + 1 + 1 + 4 + 3 + 2 + 1 + 1 + 4 + 3)/10$, which is $\frac{22}{10}$, or 2.2.

The analysis carried out in class suggests that you could expect to make, on average, $\frac{1}{1-\alpha}$ probes in carrying out a search for a non-existent key, where α is the packing density of the table. For this table, $\alpha = \frac{3}{5}$. Plugging that in and doing the arithmetic, we end up with $\frac{5}{2}$, or 2.5, which is a bit more than the 2.2 we calculated for the table shown here.

(d) Show the table after `Horta`, with home address 9, is inserted.

Solution: `Horta` would occupy the cell at address 1 after being inserted. Her home address, 9, and the one following that, 0, are both occupied, so we would resort to putting her in cell 1.

(e) Suppose that the entry for `Worf` is to be deleted. Aside from removing "him" from bucket 4, what other change should be made in the table (to ensure that a subsequent search doesn't produce a false "key not present" result).

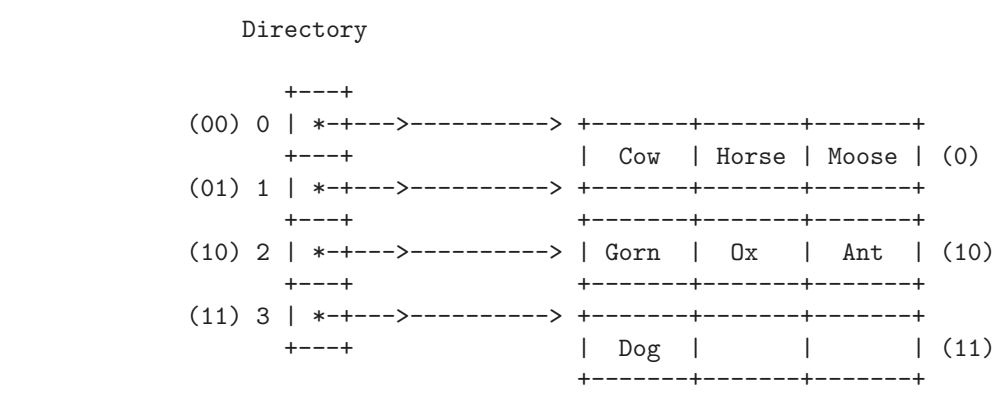
Solution: If we simply removed `Worf` from cell 4, it would be empty and so any search for a key (other than `Picard`) with home address 3 or 4 would end upon reaching (empty) cell 4. So a search for `Troi`, who occupies cell 5, would not find her, erroneously. To avoid such an error, we could take steps to ensure that the removal of a key does not result in there being an empty cell between any key's home address and the address it occupies. For the particular situation described here, we could move `Troi` into cell 4. A more complicated situation would have existed had, say, `Troi`'s home address been 5 and there had been a key (say `Horta`) with home address 4 but occupying cell 6. Then we would have to move `Horta` into cell 4 when `Worf` was removed. It's left to the reader to consider all the possibilities and how, algorithmically, they could be covered.

---

3. Suppose that we are using an extendible hash table having buckets large enough to fit three records. Our hash function H is such that

```
H(Ant)   = 1001...      H(Bat)   = 0001...      H(Cat)  = 1000...
H(Cow)   = 0000...      H(Dog)   = 1110...      H(Gorn) = 1010...
H(Horse) = 0010...      H(Moose) = 0011...      H(Ox)   = 1011...
H(Pig)   = 0110...
```

At the present time, the hash table is as follows:



For each operation described below, apply it to the original, unmodified hash table illustrated above. (Do not apply (c), for example, to the table obtained by first applying (a) and then (b)!) Show enough of the resulting table to make it clear what changes occurred. In particular, if the directory changes in size, redraw the entire picture. For the purposes of this question, two "sibling" buckets should be merged (as a result of a record deletion) if it is possible to do so. In real life, one would probably not want to merge two sibling buckets unless the resulting bucket had at least a little free space. **Explain why.**

(a) insert Cat
(b) delete Gorn
(c) insert Bat
(d) delete Horse
(e) insert Pig

Solutions:

(a) `Cat` cannot be inserted into full bucket (10), so we split it into buckets (100) and (101), with `Ant` going into the former and both `Gorn` and `Ox` going into the latter. Having split a depth-2 bucket into two depth-3 buckets requires that the depth-2 directory be doubled so as to be of depth 3. Once all this has taken place, `Cat` can be inserted into bucket (100).

(b) Having removed `Gorn` from bucket (10), and there remaining only three animals occupying it and its sibling bucket (11), we can merge them to form bucket (1). There no longer being any depth-2 buckets, the directory can be halved.

(c) Here a most unusual thing happens. We cannot insert `Bat` into bucket (0) because it is full. So we split it into buckets (00) and (01) with all its occupants going into the former (because each one is mapped by the hash function to a bit string that begins `00`). Attempting to insert `Bat` again results in a similar situation, because it, too, maps to bucket 00. So we split bucket (00) into buckets (000) and (001). `Cow` goes into the former and both `Horse` and `Moose` go into the latter. As in (a), the directory must be doubled. Then `Bat` can be put into bucket (000) without incident.

Having to split nodes two times during one insertion is very unusual, but made less so by the small scale of the example being considered. A consequence of double-splitting is that one of the new buckets introduced (namely, (01)) has no occupants.

(d) `Horse` is removed from bucket (0) without incident.

(e) `Pig` cannot be inserted into bucket (0) because it's full. So we split it (as in (c)) and all three occupants go into new bucket (00). Then we can insert `Pig` into bucket (01) without incident.

---

6. Develop an algorithm (expressed in Java-like pseudocode) that, given data that has been compressed via *run-length encoding*, decompresses it.

For simplicity, assume that the input is a sequence of nonnegative integers whose members are accessible via a method with signature `int readInt()`, each invocation of which returns the next element in the sequence. It returns -1 when there are no more elements. The output should be produced via invocations of a method with signature `void writeInt(int val)`, each invocation of which outputs the value of its parameter.

What follows is a precise description of (one way of doing) run-length encoding. Given a sequence, a *run* is a maximal subsequence all of whose members are the same. Thus, any sequence can be split up (uniquely) into runs. For example, consider this sequence:

7 0 0 5 5 5 5 5 5 9 9 0 0 0 8 2 2 2 0 6 6 6 6 7 5 1 1 2

(No particular significance should be attached to the fact that all elements in this example are one-digit numbers.)

Using parentheses (purely for illustrative purposes) to partition this sequence into its runs, we get

(7)(0 0)(5 5 5 5 5 5)(9 9)(0 0 0)(8)(2 2 2)(0)(6 6 6 6)(7)(5)(1 1)(2)

The rules by which we do run-length encoding are as follows:

(1) Any run of length three or less whose member is non-zero is left unaltered.
(2) Any run of length four or more whose member is non-zero is replaced by three numbers: zero (which signals that a run is being encoded), followed by the length of the run, followed by the member itself. For example, a run of 5's having length twelve is encoded as 0 12 5.
(3) Any run of length two or more whose member is zero is encoded as in (2). For example, a run of 0's having length three is encoded as 0 3 0.
(4) A run of length one whose member is zero is encoded as 0 0.

Following these rules, the example sequence above is encoded as

(7)(0 2 0)(0 6 5)(9 9)(0 3 0)(8)(2 2 2)(0 0)(0 4 6)(7)(5)(1 1)(2)

(Again, we used parentheses simply to highlight the runs.) For this particular example, very little compression was achieved. (The original sequence contains 28 values, the compressed sequence 26.) For some kinds of data (e.g., graphical images), however, it often happens that run-length encoding achieves good compression.

Solution:

```
k := readInt();
do while k != -1
   if k != 0 then   // it's not a run
      writeInt(k);
   else  // could be a run
      m := readInt();
      if m = 0 then   // double zero encodes zero
         writeInt(0);
      else  // it's a run!
         r := readInt();
         do m times        // write m copies of r
            writeInt(r);
         od
      fi
   fi
   k := readInt();
od
```