ECE-250 – Algorithms and Data Structures (Winter 2012)

Tutorial 3 (2012-02-02) – Suggested solutions

1 - Playing with binary arithmetic: Write (reasonably efficient) C++ functions operating on `unsigned int` values that:

(a) Check if a given number is a power of 2.
(b) Round a number up to the next power of 2 (e.g., values between 17 and 32 would be rounded to 32).
(c) Multiply times 34 (recall that multiplying times 2 or times a power of 2 is fast).
(d) Swap the lower and upper bytes in a 16-bit value — that is, produce a value that is the given value with the swapped bytes (you may assume either an `unsigned short int` / `uint16_t` value, or simply assume that the value stored in an `unsigned int` is a 16-bit value).
(e) Reverse the bits.
(f) Given four `unsigned int` values in an array, obtain a single value that contains, at the corresponding position of the byte, the lower byte of the value at position 0, the second lower byte of the value at position 1, and so on for the four values.

Example: if the values are {`0x11111180, 0x22224022, 0x33203333, 0x10444444`}, the function should produce `0x10204080` as the result.

Solution:

(a) Without proof (thought it would be an interesting exercise to do — induction, perhaps?):

A power of 2 has just one bit set (as in, set to 1), and the rest are zeros. If we subtract one, we get all ones in the bits lower than the bit that was one (2^n is a 1 followed by n zeros. $2^n - 1$ is a zero followed by n ones). Thus, the positions of the ones are disjoint. This only happens if the original value had only one bit set (this is the part that would be interesting to prove).

Thus, we compute the bitwise AND of the value (say, x) and $x - 1$. If we obtain zero, it means the ones are disjoint, and thus x had only one bit set.

The one problem is that for $x = 0$, this operation returns true; so, we explicitly exclude that one:

```
bool is_pow2 (unsigned int x)
{
    return x != 0 && (x & (x - 1)) == 0;
}
```

(b) The idea is that we want to take the highest bit that is one, and return a value that has the following bit as the only one (unless it is already a power of 2, in which case we return the same value).

To this end, we can shift-right and bitwise OR (so that every bit that is one propagates to the right filling up every bit with a value of 1). We could do this 32 times to ensure that the propagation fills all the bits. However, if we keep shifting the value, the only way that the result of the OR operation won't change is when all of the right-most bits are already one. Once we have all ones in the bits at and below the highest bit originally set, we just add one:

```
unsigned int round_pow2 (unsigned int x)
{
    if (is_pow2(x))  return x;

    unsigned int combined = x, shifted = x >> 1;
    while ((combined | shifted) != combined)
    {
        combined = combined | shifted;
            // we trust that the compiler will optimize this
            // and will avoid the duplicate calculation (otherwise,
            // we require an extra variable)
        shifted >>= 1;
    }
    return combined + 1;
}
```

(c) Multiplying times 32 would be really easy — just shift-left five times; then, we'd need to add twice the original value. And twice the original value is just the value shifted once. A more compact way to do this (though probably the same in terms of the assembly-level code that the compiler will generate) could be multiplying times 17 first (since this is multiplying times 16 — easy — and then add the original value). Once we have this result, we shift it left one position:

```
unsigned int times34 (unsigned int x)
{
    return ((x << 4) + x) << 1;
}
```

(d) This is much less tricky than the previous ones — just mask each part, shift them to the right target positions, then bitwise OR them:

```
unsigned int swap_words (unsigned int x)
{
    return ((x & 0xFFFF) << 16) | ((x & 0xFFFF0000) >> 16);
}
```

(e) Reversing the bits requires a loop (well, we could do it like (d), but writing up the 32 sub-expressions and leaving the compiler do the dirty work. But that's ugly (yes, I know — *but that's efficient!*).

The solution with the loop is just extracting the lower bit (AND with 1), then add it to the result as the LSB, and shift that result (to open up space for the next bit):

```
unsigned int reversed_bits (unsigned int x)
{
    unsigned int reversed = 0;
    for (int i = 0; i < 32; i++)
    {
        reversed |= (x & 1);
        reversed <<= 1;
        x >>= 1;
    }
    return reversed;
}
```

(f) Not much to say about this, other than clarifying that in the expression `*ptr++`, where `ptr` is a pointer, the precedence of operators works as expected: we dereference `ptr` and get the pointed value, and `ptr` is incremented (but we get the pointed value *before* the pointer gets incremented):

```
unsigned int bytes (unsigned int * x)
{
    unsigned int result = 0, mask = 0xFF;

    result |= (mask & *result++);
    mask <<= 8;
    result |= (mask & *result++);
    mask <<= 8;
    result |= (mask & *result++);
    mask <<= 8;
    result |= (mask & *result);

    return result;
}
```

2 - Hash tables – linear probing: We want to store values between 0 and 9999 in a hash table of size 10.

The hash function operates as follows: given a value x, add the four digits of x and take the last (right-most) digit. For example, the hash of 3276 would be 8 ($3 + 2 + 7 + 6 = 18$).

(a) Insert, in the given order, the values 3836, 7209, 2373, 9412, 6950, 471, 5569, 9703.

(b) Then, remove, in the given order, 3836, 9412, and 5569.

Solution:

(a) Adding elements

h(3836) = 0 $(3 + 8 + 3 + 6 = 20)$, so it goes at bin 0
h(7209) = 7 $(7 + 2 + 0 + 9 = 18)$ — bin 8
h(2373) = 5 $(2 + 3 + 7 + 3 = 15)$ — bin 5
h(9412) = 6 $(9 + 4 + 1 + 2 = 16)$ — bin 6

At this point, the array's contents is:

0	3836
1	
2	
3	
4	
5	2373
6	9412
7	
8	7209
9	

h(6950) = 0 $(6 + 9 + 5 + 0 = 20)$ — collision at bin 0; probe the following bins, to find bin 1 empty:

0	3836
1	6950
2	
3	
4	
5	2373
6	9412
7	
8	7209
9	

h(471) = 2 $(0 + 4 + 7 + 1 = 12)$ — bin 2
h(5569) = 5 $(5 + 5 + 6 + 9 = 25)$ — collision at bin 5; probe the following bins, to find bin 5 and 6 taken, then 7 available:

0	3836
1	6950
2	471
3	
4	
5	2373
6	9412
7	5569
8	7209
9	

h(9703) = 9 $(9 + 7 + 0 + 3 = 19)$ — bin 9

0	3836
1	6950
2	471
3	
4	
5	2373
6	9412
7	5569
8	7209
9	9703

(b) Removing elements:

h(3836) = 0, and this value is at that position; remove it, and scan forward to look for elements that could (and should) be moved back to fill this bin: 6950 hashes to 0, so it can be moved:

0	6950
1	
2	471
3	
4	

5	2373
6	9412
7	5569
8	7209
9	9703

Now, repeat for hole at position 1, scanning forward: 471 hashes to 2, so it should NOT be moved; continue searching, and we hit an empty bin, so we're done with this removal.

h(9412) = 6, and this value is at bin 6; remove it, and scan forward for elements that can be moved to fill the empty bin: 5569 hashes to 5 (before the hole), so this one should be moved. Now hole is at position 7; 7209 hashes to 8, so it should not be moved, 9703 hashes to 9, so it should not be moved; then cycle back to position 0, to check 6950, which hashes to 0, so it should not be moved, next one is an empty bin, so we're done with this removal:

0	6950
1	
2	471
3	
4	
5	2373
6	5569
7	
8	7209
9	9703

Removing 5569 causes nothing else to move (we compute its hash, probe next bin to find it, remove it, and there is an empty bin in the next position, so no elements can be affected by this removal).

3 - Hash tables – double hashing (you may need a calculator for this one [1]): with a hash table of size 8 to store values between 0 and 9999, we use the following hash functions: given a value $x = d_3d_2d_1d_0$, we compute the value $(d_3+1)\times(d_2+1)\times(d_1+1)\times(d_0+1)$. This result takes no more than 13 or 14 bits (right? *why?*). So, the primary hash function (the one to determine the bin) is the value $b_4b_3b_2$, where b_n is the bit n of the result, with the convention that b_0 is the least-significant bit. The jump size (the secondary hash function) is given by the value b_6b_51.

(a) Write C++ functions (as efficient as possible) implementing these hash functions — you want to avoid repeating the product of the digits. You may assume that the given value is between 0 and 9999.

(b) Insert, in the given order, the values 3836, 7209, 2373, 9412, 6950.

Solution:

(a) The product of digits is straightforward: we isolate the digits by taking modulo 10 (not a particularly efficient operation, but we have no choice, given the description of the hash functions), then we divide (integer division) by 10, to shift-right one position in the decimal representation (so that we

[1] If you don't, good for you!

can then extract the next digit):

If we want to be fancy, we could create a class to do both hash functions, so that the class does the operation and holds on to the result, so that now the two hash functions can be obtained without repeating the computation:

```
class Double_hash
{
public:
    Double_hash (int x)
    {
        d_scrambled = (x % 10) + 1;
        x /= 10;
        d_scrambled *= (x % 10) + 1;
        x /= 10;
        d_scrambled *= (x % 10) + 1;
        x /= 10;
        d_scrambled *= (x % 10) + 1;
    }

    unsigned int bin() const
    {
        return (d_scrambled >> 2) & 7;
    }

    unsigned int jump_size() const
    {
        return ((d_scrambled >> 4) & 7) | 1;
    }

private:
    unsigned int d_scrambled;
};
```

(b) Inserting values:

3836 goes into bin 4: $4 \times 9 \times 4 \times 7 = 1008 = 1111110000_2$ — $\mathrm{h}(3836) = 100_2 \;=\; 4$

7209 produces a collision at bin 4: $8 \times 3 \times 1 \times 10 = 240 = 11110000_2$ — $\mathrm{h}(7209) = 100_2 \;=\; 4$. The jump size for this one is 111_2, or 7 (which modulo 8 is -1 — right?). So, we probe bin 3 ($4 + (-1)$, or equivalently, $4 + 7 \bmod 8 = 3$.

Thus, 7209 goes into bin 3.

2373 goes into bin 0: $3 \times 4 \times 8 \times 4 = 384 = 110000000_2$ — $\mathrm{h}(2373) = 0$.

9412 produces a collision at bin 3: $10 \times 5 \times 2 \times 3 = 300 = 100101100_2$ — $\mathrm{h}(9412) = 011_2 \;=\; 3$. The jump size for this one is $001_2 = 1$. We then probe bin 4 (taken), and bin 5 (available).

Thus, 9412 goes into bin 5.

Side note: it may look like now we're getting a cluster (which double hashing was supposed to avoid, right?). But no: this is *not* a cluster — different values would have different jump sizes, so being one after the other one does not mean that they constitute a cluster in the bad sense of deteriorating performance.

Finally, 6950 goes into bin 1: $7 \times 10 \times 6 \times 1 = 420 = 110100100_2$ — h(6950) = 1. No collision — the jump size, had it been needed, is $101_2 = 5$ (so this example supports the argument that those three consecutive bins do not constitute a cluster — if this value had caused a collision there, just one probe would have gotten us out of the alleged cluster).