7/15/2020 - Bitwise Op, Hash Tables

Wednesday, July 1, 2020 9:44 AM

What are bitwise operators?

They are operators, just like +, -, *, /, but instead of working on whole integers, they work on the individual BITS of the stored integers.

int x = 35;
// in the computer this is 00000000 00000000 00000000 00100011

In the computer, values are actually stored in two's complement. The key difference between this and regular binary is that the most significant bit has a negative value. So, a 1 in the first slot is worth -2^{31} not 2^{31}. In an integer, the max value is a 0 followed by all 1s, and this value is $2^{31} - 1$. The min value is a 1 followed by all 0s, which is -2^{31}. You'll learn more about this in CDA 3103…

List of bitwise operators:

&	notice this is one ampersand, not 2	this is bitwise AND
|	notice this is one vertical bar not 2	this is bitwise OR
^	this IS NOT EXPONENTIATION	this is bitwise XOR
~		this is bitwise NOT

XOR means exclusive or, here is the truth table for XOR

p	q	p XOR q
F	F	F
F	T	T
T	F	T
T	T	T

Note the difference with or in the last line.

35 & 41

100011
101001

100001 = 33

35 | 41

100011
101001

101011 = 43

35 ^ 41

100011
101001

001010 = 10

~35 (I am going to use 8 bits, two's complement)

35 = 00100011
~35 = 11011100 = $-2^7 + 2^6 + 2^4 + 2^3 + 2^2 = -36$
In general, ~x = -x-1. -x = ~x + 1.

Lowest One Bit - of all the bits set to one, what is the value of the least valuable bit? (This will always be a power of 2.) This is also the highest power of 2 you can factor out of x.

LowestOneBit(x) = x & (-x)

Reason is that, between the two representations, both have the same highest power of 2 that you can factor out, and none of the rest of the bits will both be set to 1.

Two more operators:

>>	Right Shift Operator
<<	Left Shift Operator

x >> 2, this means shift the binary value of x by 2 bits to the right.

Example:

113 >> 2

01110001 >> 2 = 011100 = 28

Note: This is equivalent to doing an integer division by 2^2, in all unsigned cases.

Example of left shift:

3 << 3

00000011 << 3 = 00011000 = 24

Note: For unsigned values, this is equivalent to multiplying by 2^3.

On a micro level, what can I use the shift operators for?

Ans: In a particular number, which bits are on and which bits are off. (I'll talk about bit 0 as being the least significant bit, bit 1 has being the next least signficant bit. In general, bit k is the bit value worth 2^k.

Let's say we have an integer x and we want to know which bits are on and off.

We can use the "flashlight" technique.

Any expression of the form 1 << i, where is is an integer, has a single 1 in its binary representation. If we do a bitwise and with a number that has only 1 bit 1 in its binary representation, the answer can have at most 1 bit 1 in its binary representation.

```
11110101      (some random number)
00010000 &    (number of the form 1 << i)
-----------
000?0000
```

The bitwise and must have 0s in all slots except for possibly, bit i. For bit I, the result will be 1 if the bit i in the other number is 1, and the result will be 0 if that bit is 0.

In general, what x & (1 << i) does, is that it shines a flashlight on bit i of x. If this bit is on, the value of the expression will be non-zero. If the bit is off, the value of the expression will be 0.

```
for (int i=0; i<31; i++) {
   if ((x & (1<<i)) != 0)
      printf("bit %d is on in %d\n", i, x);
}
```

What sorts of problems might this be useful in solving?

Answer: Pretty much anything where we deal with subsets. When we look at a single integer, it is basically a boolean array.

For example, n = 47

00101111, this represents the array:

T	T	T	T	F	T	F	F

Index 0 is true since bit 0 is 1
Index 1 is true since bit 1 is 1
…
Index 4 is false since bit 4 is 0.
…

A few lectures ago, in recursion, we printed out all subsets of some candy. Let's do that with bitwise operators…

My bitmask will do

00000 = 0 (all false)
00001 = 1 (index 0 is true, rest are false, etc.)
00010 = 2

…
11111 = 31

Correct Answer Recovery

Store each students' answers as a single integer bitmask

FFFTTTF convert to 14 (bitwise rep 0001110)

We don't know what the real answers were, but we could pretend that we knew and try each possible set. There are at most 2^q sets of answers and with q = 15, this is feasible.

When we try an answer key, we will use it to grade each test. Then, this will give us a grade distribution. We can check to see if this was the actual grade distribution. If it is, then this is a possible set of answers.

For each answer key k:

    Start a frequency array for answer key k.
    For each student s:
        score the exam s using key k, add to the freq array

    Check if this frequency array equals the known distribution.
    if so, add 1 to our answer.

How do I score an exam? Let's try out an example:

```
Student = 01101100
Ans     = 11110000  ^
          ----------
           10011100
```

All mismatched columns (wrong answers) have exactly 1 zero and 1 one, the xor of these bits is 1. (xor highlights the differences…)

Now, take this xor value and count the number of bits equal to 1. This is the number of wrong responses, so we can score the exam.

where I have res++, we could also do res = res | 1, since the lowest bit is always 0 right before this line.

I already showed you one strategy to count # of bits that are on. In my recovery code we have a different strategy, right shift the number and check the least significant bit each time.

HASH TABLES

We looked at linked lists, then binary trees…
We wanted to support inserting items, searching for items and deleting items.

Linked Lists did these things in O(n) time.
Binary Search Trees did these things in average of O(lg n) time.
AVL Trees did these in worst case O(lg n) time.

Can we do even better?

Answer: In the average case, yes!!! Hash Tables give us close to O(1) time for inserts, deletes searches.

The idea is as follows:

Make the table an array of size n, indexed from 0 to n-1.

We create a function, called a hash function. The property function is that it takes in any input that we want to add to the table (say a string or integer or some struct) and it returns an integer in between 0 and n-1. In theory, a good hash function returns each value equally likely and small changes in the input result in large random changes in the output.

Here is a not-so-good hash function to make this concrete:

table size = 26

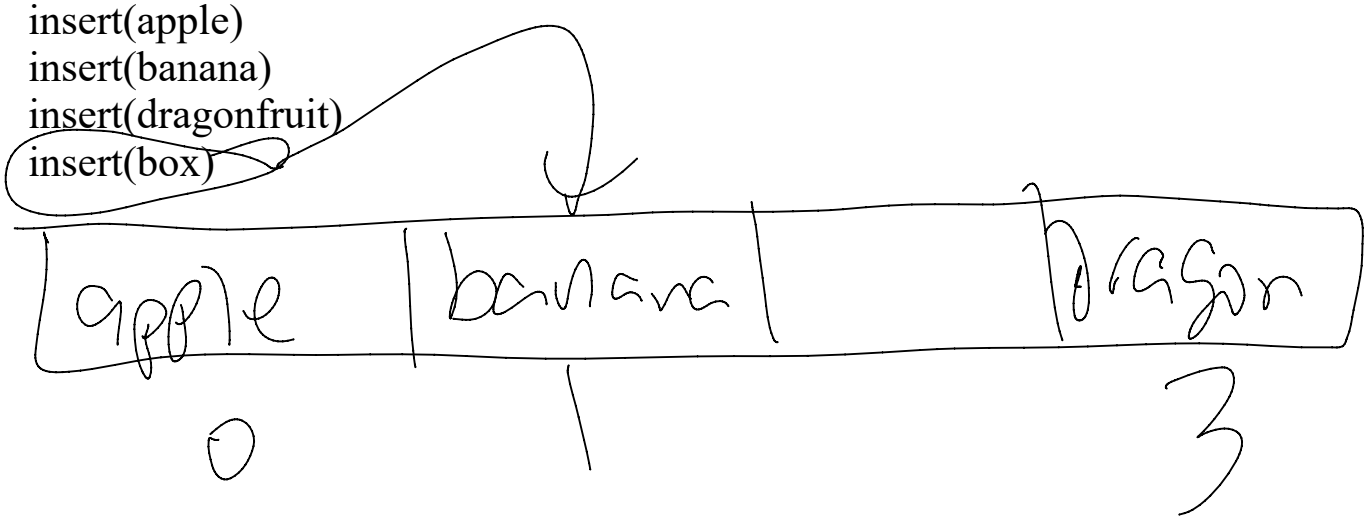
```
// Pre-condition: all words consist of lowercase letters only.
int hashfunc(char* word) {
```

```
    return word[0] - 'a';
}
```

When we get an item, x to insert, calculate hashfunc(x), and put the value in index hashfunc(x).

What's the problem with this?

insert(apple)
insert(banana)
insert(dragonfruit)
insert(box)



The problem with this situation is that hash functions by their very nature, are many to one functions, so you may have two different input values mapping to the same location in the table. This is called a collision.

What do we do with collisions?

1) Erase the old thing, put in the new thing (not dealing with it)
2) Linear Probing
3) Quadratic Probing
4) Linear Chaining Hashing

Option #1: Seems horrible, but there are applications (caching) where it's okay to lose data. (Strengths: super fast, Weakness: you lose stuff)

Option #2: If the initial location, k, is filled, go to k+1, then k+2,… until you find the first empty location, wrapping around the end of the array if necessary.

In linear probing, box would go in index 2:

In linear probing, box would go in index 2:

apple	banana	box	dragonfruit

Cluster

Good news is we didn't lose anything.
Bad news is…now searching takes longer. If I am looking for box and don't find it in index 1, I can't give up, I have to continue looking until I either hit box or an empty square.
Imagine looking for book, we would look in index 1, then index2, then index 3, and finally when we see index 4 is empty, we can conclude that book isn't in the table.

If the table is sparse, we'll hit an empty spot pretty soon, more than likely…but as the table fills up, it will get worse and worse (like traffic). There is a phenomenon called clustering…

After some time, clusters will grow where several spots in a row are used. The larger a cluster gets, the more likely a hit will occur somewhere in the cluster and then that next insertion will take longer AND make the cluster bigger.

Option #3: In an effort to avoid clustering, instead of trying indexes k, k+1, k+2, k+3, …We will try indexes in the following order:

k, k+1, k+4, k+9, k+16, etc.

Each time we get a hit, we go to index $k + i^2$, where i is how many times we've looked. In code, we actually do this:

```
int k = hashfunc(item);
int step = 1;
while (…) {

   k = k + step;
   step = step + 2;
}
```

We add 1 to k, then we add 3 to k, then we add 5 to 5, etc.

This idea comes from math $1 + 3 + 5 + .. + (2i-1) = i^2$

Example of tracing linear probing hashing

```
int hashfunc(int n) {
    return n%10;
}
```

insert(47)
insert(13)
insert(99)
insert(36)
insert(76)…collision until index 8
insert(30)
insert(16)…collision 6,7,8,9,wraparound0, okay 1

search 77…go to index 7, not there, but we must go to index 8, then 9, then 0, then 1 and at index 2 we can definitely say that 77 isn't in the table.

0	1	2	3	4	5	6	7	8	9
30	16		13			36	47	76	99

Example of tracing quadratic probing hashing

```
int hashfunc(int n) {
    return n%10;
}
```

insert(47)
insert(13)
insert(99)
insert(36)
insert(76)…try 6, 7, 10%10 = 0
insert(30)…try 0, insert at index 1
insert(16)…try 6, 7, 10%10 = 0, (6+9)%10 = 5

search 77…go to index 7, not there, but we must go to index 8, then 9, then 0, then 1 and at index 2 we can definitely say that 77 isn't in the

table.

0	1	2	3	4	5	6	7	8	9
76	30		13		16	36	47		99

General rule: Never fill either table more than 50%.

Last strategy: linear chaining hashing…

why go to a different index, just store multiple items at each index in a linked list:

Example of tracing linear chaining hashing

```
int hashfunc(int n) {
    return n%10;
}
```

insert(47)
insert(13)
insert(99)
insert(36)
insert(76)…insert at front of index 6
insert(30)…insert at front of index 0
insert(16)…insert at front of index 6

0	1	2	3	4	5	6	7	8	9
↓			↓			↓	↓		↓
30			13			16	47		99
						↓			
						76			
						↓			
						36			

when we search, we don't have to go to different indexes, but we do have to go through a linked list at our index.

Most solutions will probably use this last aspect. But one reason not to, is that array code is faster than having to deal with dyanmically allocated memory, so if you can make your table big enough for your application so that it never gets more than 50% filled, the Quadratic Probing could be faster that linear chaining hashing because the individual array operations are faster in practice.

New practice code I write:

Dictionary Look Up

regular binary search in an array of words.

hash table with quadratic probing

hash table with linear chaining hashing

Metrics - either do time, or count the number of simple operations.