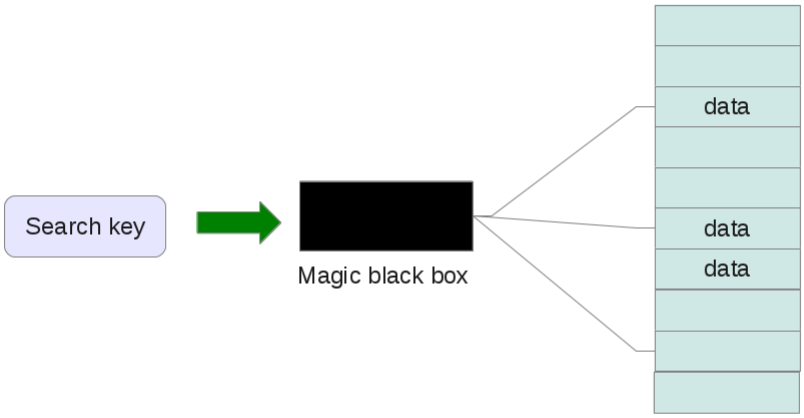
HASH TABLE

CS 200 RECITATION 10 (WHICH MEANS IT'S WEEK 11, WHERE DOES THE TIME GO??!)

Hash Table / Map

Abstract data type, usually based on an array. **Very different from the trees we have been working with.**



Main idea is to have a function (the magic box above) which computes the correct array index to store something in the array based on a search key, hopefully every search key will have a unique index. This function is called the Hash Function.

- Hash Function computes array index from search key
- Hash Function should run in constant time
- Hash Function should return a unique index for each unique search key (though, this is not always practical as discussed below)
- Hash tables generally do not have traversal algorithms, you are supposed to get single elements by their keys.
- Data is usually stored as a <key, value> pair

- Advantages:
  - Fast insert, search, delete. These usually happen in constant time.
- Disadvantages:
  - space inefficient, as the array has to be larger than the total number of data elements (it has to maintain some blank space), discussed below.
  - not sorted.

Hash Functions

Can be very simple. Any technique to convert a search key into a array index will work (more or less). Signs of a good hash function:

- Easy and fast to compute
- spreads data evenly through the table. Both random and non-random should spread evenly
- low rate of collisions

Examples:

- if search key is an integer:
  - pick 3 digits, make new number, that is hash
  - add up all the digits
- if search key is a string:
  - assign each character a number, add

Collisions

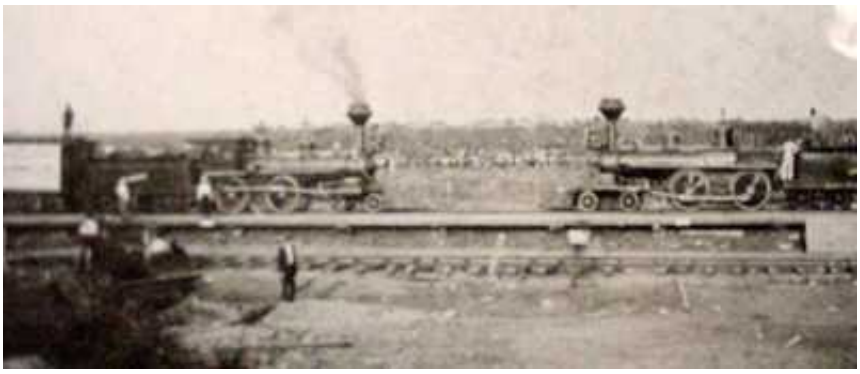


Figure 1: Image credit: Wikimedia Commons, public domain, http://commons.wikimedia.org/wiki/File:CrushTxBefore.jpg

A collision happens when two different search keys hash to the same index in the table. This is actually pretty common, as most hash functions are imperfect. So we must have a way to resolve collisions. There are several ways to do this, but they all boil down to either finding another location in the table for the second entry or making one table entry able to hold multiple elements. Common approaches:

- Scan the table for an empty slot, increment the location one at a time

- Scan the table for an empty slot, increment the location by some other hash function
- Convert a table entry into a multiple entry structure like a linked list or array

A high rate of collisions is the Achilles heel of the hash map. The rate of collisions generally increases as the underlying array fills up. For this reason, The array is usually maintained to have significant empty space.

Hash Table Operations

- Hash Function
  - converts search key into table (array) index.
- Insert
  - compute hash from search key
  - if table[hash] is empty, store <key, value> at that location
  - resolve collision if necessary
- Retrieve (search)
  - compute hash from search key
  - if table[hash] is empty, object is not in the table
  - if table[hash] has correct object, return it
  - if table[hash] has incorrect object, there was a collision. Apply same technique used in Insert to search for the correct object.
- Delete
  - Very similar to search, except that if/when the correct object is found, it is deleted instead of being returned.

Similar to other data structures we have studied: Insert, search and retrieve are all very similar in their operation. They use the same technique to find things, and the same technique to resolve collisions

Exercise

I've given out skeleton code for a Hash Table based on an array. This table will store data similar to a compiler's symbol table, the search keys will be strings (a.k.a variable names) and the values will be memory addresses (a.k.a. ints). This is loosely based on Prichard pp 774, programming problem 4.

- I've made an object called table_entry to store our <key, value> (<String, int>) pairs.
- The hash table will be built on an array of table_entry objects
  - table_entry table[] = new table_entry[size]
- The hash function will be:
  - Convert the string into an integer N (see below)
  - Hash = N % table_size
    - % is the modulus operator, divides it's operands and returns the remainder.
- Collisions will be resolved with linear probing:

  - Scan every table index looking for an empty spot
  - check table[hash+1], then table[hash+2], etc...
  - if it reaches the end of the table, restart from the beginning.
  - don't let it loop infinitely, stop when you have found an empty space or have checked the entire table once.
  - while(table[index] != null) //might work

- You must implement:
  - hash_function(String key)
    - convert the string into an array index
  - insert(String key, int value)
    - get hash from hash_function()
    - check for and resolve collision if necessary
    - set table[some_index] = new table_entry(key, value)
  - retrieve(String key)
    - get hash from hash_function()
    - if table[hash] is not null and it's object has same search key, return table[hash]
    - if table[hash] is not null and it has wrong object, there was a collision. Scan table in same manner as insert, looking for correct key
    - if table[hash] is null, key is not in the table
  - delete(String key)
    - get hash from hash_function()
    - same behavior as retrieve, but deletes instead of returning (set table[hash] = null)

- Given code:
  - main_class
    - main method, testing
  - Randoms
    - used by main to generate random 5 character strings and random memory addresses
  - hash_table
    - the hash table ADT, where you will be implementing stuff
    - hash_string: convert string to an integer, see below
    - number_of_char: convert a character into it's alphabet location. A is 1, B is 2, ... Z is 26
  - table_entry
    - stores a <key, value> pair
    - should not need modification

String to int

First, each character is converted into it's location in the alphabet. Next, we want to convert them to binary, and list them all next to each other as one int.

```
key = "note", so
n = 14 is 01110
o = 15 is 01111
t = 20 is 10100
e = 5  is 00101
all togeather: 01110011111010000101 is 474,757 in decimal
```

However the following equation make the computation much easier:
$14 \times 32^3 + 15 \times 32^2 + 20 \times 32^1 + 5 \times 32^0 = 474,757$
See prichard pp 743 for more details

Grading

- Sign the attendance sheet
- Checkin your code as a tar file like so:
  - Monday: ~cs200/bin/checkin R10L01 R10L01.tar
  - Tuesday: ~cs200/bin/checkin R10L02 R10L02.tar
  - Wednesday: ~cs200/bin/checkin R10L03 R10L03.tar
  - Thursday: ~cs200/bin/checkin R10L04 R10L04.tar

Example Run

here is my version running with table size 23, inserting 14 entries:

```
Adding: kovst 0x330C54B0
Adding: albpy 0x615DB05E
Adding: cbkxe 0x3F71DFDB
Adding: lybie 0x751FE880
Adding: rgsxa 0x1B061852
Adding: miluu 0x22BF94F5
Adding: ilgag 0x28135087
Adding: wxqgk 0x10FE491E
Adding: vehvw 0x0F2331CC
Adding: psccy 0x35D50D1B
Adding: lkhjp 0x4A5F6121
Adding: awgjl 0x7967CC6A
Adding: owwzo 0x3441E914
Adding: wxbap 0x31476094

Hash Table:
 0: <albpy, 0x615DB05E>
 1: <miluu, 0x22BF94F5>
 2: <kovst, 0x330C54B0>
 3: <ilgag, 0x28135087>
 4: <vehvw, 0x0F2331CC>
 5: <psccy, 0x35D50D1B>
 6:    ...[empty]...
 7: <lybie, 0x751FE880>
 8: <awgjl, 0x7967CC6A>
 9:    ...[empty]...
10:    ...[empty]...
11:    ...[empty]...
12: <cbkxe, 0x3F71DFDB>
13: <lkhjp, 0x4A5F6121>
14:    ...[empty]...
15:    ...[empty]...
16:    ...[empty]...
17: <owwzo, 0x3441E914>
```

```
18: <wxqgk, 0x10FE491E>
19: <wxbap, 0x31476094>
20:    ...[empty]...
21: <rgsxa, 0x1B061852>
22:    ...[empty]...
retrieve 4th addition:
<rgsxa, 0x1B061852>
retrieve 'tom', not in the table:
null
Delete 4th addition, print resulting table:
Hash Table:
 0: <albpy, 0x615DB05E>
 1: <miluu, 0x22BF94F5>
 2: <kovst, 0x330C54B0>
 3: <ilgag, 0x28135087>
 4: <vehvw, 0x0F2331CC>
 5: <psccy, 0x35D50D1B>
 6:    ...[empty]...
 7: <lybie, 0x751FE880>
 8: <awgjl, 0x7967CC6A>
 9:    ...[empty]...
10:    ...[empty]...
11:    ...[empty]...
12: <cbkxe, 0x3F71DFDB>
13: <lkhjp, 0x4A5F6121>
14:    ...[empty]...
15:    ...[empty]...
16:    ...[empty]...
17: <owwzo, 0x3441E914>
18: <wxqgk, 0x10FE491E>
19: <wxbap, 0x31476094>
20:    ...[empty]...
21:    ...[empty]...
22:    ...[empty]...
```