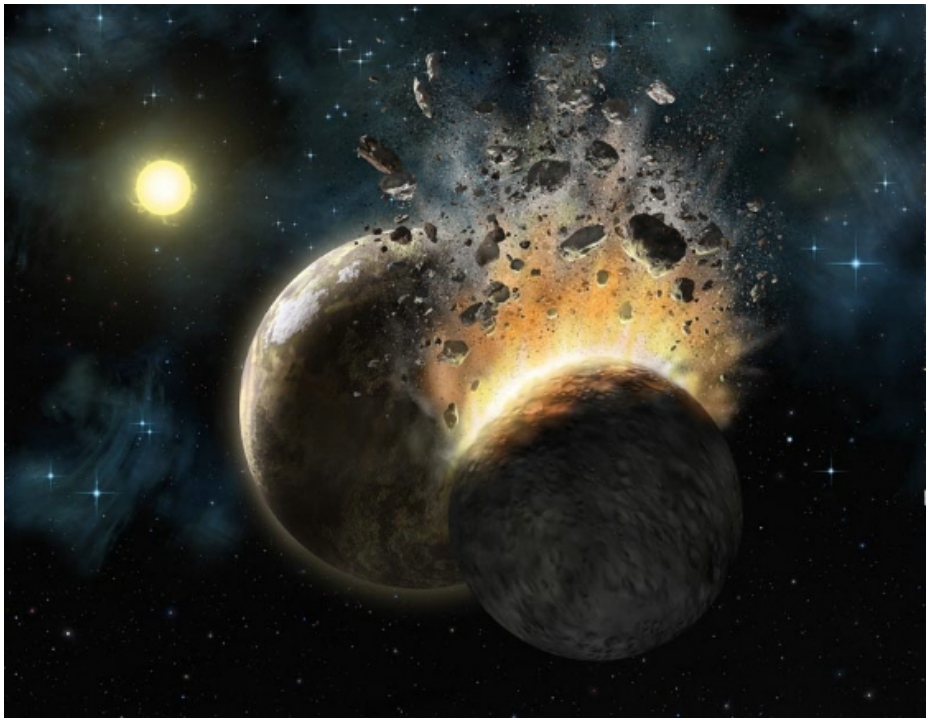
Lecture 16
Hash Collision Resolution



EECS 281: Data Structures & Algorithms

Collision Resolution

Data Structures & Algorithms

Collision Resolution

Needed to handle insert, search, and removal, when multiple keys hash to same table index.

Methods of Collision Resolution

- Separate Chaining
- Linear Probing
- Quadratic Probing
- Double Hashing

} Open addressing methods

Separate Chaining

Resolve collisions by maintaining *M* linked lists, one for each table index

1. Quickly find the aisle you want
2. Perform linear search on the items in that aisle

Separate Chaining Properties

- Separate chaining reduces the number of comparisons for sequential search by a factor of *M* (on average), using extra space for *M* links
- In a table with *M* lists (table indices) and *N* keys, the probability that the number of keys in each list is within a small constant factor of *N*/*M* is extremely close to 1, *if the hash function is good*

Load Factor (α)

- α = *N*/*M*, where *N* keys are placed in an *M*-sized table
- Separate Chaining
  - α is average number of items per list
  - α can be > 1
- Open Addressing
  - α is percentage of table indices occupied
  - α must be <= 1

Separate Chaining Complexity

- **Insert**: constant time
  - ***O*(1)** if duplicate keys are allowed, ***O*(α)** if not
- **Search**: time proportional to *N*/*M*
  - ***O*(α)**
- **Remove**: dependent upon Search
  - ***O*(α)**

This is why we choose $M \approx N$: $\boldsymbol{O(\alpha) = O(1)}$

Speeding up the Worst Case

Use something other than linked lists

- Use *M* vectors
  - Could keep contents sorted, which is $\boldsymbol{O(\alpha)}$ insert but $\boldsymbol{O(\log \alpha)}$ search
- Use *M* binary search trees, which is $\boldsymbol{O(\log \alpha)}$ insert and search (as long as tree is not a stick)
  - Large memory overhead

Open Addressing

Resolve collisions by using empty locations in the table

- Parking: if spot is full, go to the next open spot
- Returning to car: remember general area, linear search the spots for your car; if a spot is empty, keep searching (someone left)

Probing

- Check a location, by index, in the hash table
- Used to search for data or find available locations
- Possible probe outcomes (basic)
  - **Empty**: no data found at probed location
  - **Hit**: probe finds occupied location that contains an item whose key matches the search key
  - **Full**: probe finds occupied location but item key doesn't match search key
- If probe results in *full*, then probe table at "next higher" cell until *hit* (search ends successfully) or *empty* (search ends unsuccessfully)

Linear Probing

1. Hash search key to get initial table index
2. Probe buckets at sequential indices, starting at table index, until search key or an open address is found

Collision resolution

- $t(key) \bmod M \Rightarrow$ table index
- If bucket at table index is full, then try
  - $(t(key) + j) \bmod M$ for $j = 1, 2, \ldots$

Clusters

- A contiguous group of occupied table cells
- Consider a table that is half-full ($N = M / 2$)
  - What is best-case distribution of half-full table?
  - What is worst-case distribution of half-full table?

- What is the *average* cost (in terms of N) to obtain a *miss* (find an empty cell) given the best-case distribution?
- What is the *average* cost (in terms of N) to obtain a *miss* (find an empty cell) given the worst-case distribution?
- What can be done to favor the best case distribution?

Open Addressing Deletions

How can an item be deleted from a table built with linear probing? Why is this hard?

- Option 1: remove it, re-hash rest of cluster
- Option 2: use a “deleted” item
  - Not a data item, not empty either
  - We’ll call this “deleted”

“Deleted” Items

New probing outcome

- **Deleted**: probe finds a location that once held an item, but is not currently holding one

When a probe finds an item marked deleted

- ***Insert*** considers it an empty spot (*miss*) and thus usable (if the key does not exist elsewhere)
- ***Search*** considers it occupied (*full*) and keeps searching

Maintaining Bucket Status

- To determine whether a bucket is used or unused requires a bool for each bucket
- When a third state is added: used, unused, or deleted
  - Option 1: Use two bools
  - Option 2: Declare an `enum`

  `enum`: A variable type that can only take on certain “enumerated” values

enum Example

```
1  enum class BucketType {
2      Empty,  // bucket never contained an item
3      Occupied,  // currently contains an item
4      Deleted  // is a deleted element
5  };  // BucketType{}
6
7    if (buckets[i].type == BucketType::Empty) ⋯
8
9    switch (buckets[i].type) {
10   case BucketType::Occupied:
11     ⋯
12   }  // switch
```

Counting Probes

When collisions are resolved with linear probing, the average number of probes required to search in a hash table of size *M* that contains $N = \alpha M$ keys is about

$$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$ for hits

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$ for misses

Effect of Load on # of Probes Linear Probing

α	**Average Probes: Successful Search**	**Average Probes: Unsuccessful Search**
0.1	1.1	1.1
0.2	1.1	1.3
0.3	1.2	1.5
0.4	1.3	1.9
0.5	1.5	2.5
0.6	1.8	3.6
0.7	2.2	6.1
0.8	3.0	13.0
0.9	5.5	50.5

Quadratic Probing

1. Hash search key to get initial table index
2. Probe buckets at increasing ‘distance’ from the table index, until search key or an open address is found

Collision resolution

- $t(key) \bmod M \Rightarrow$ table index
- If bucket at table index is full, then try
  - $(t(key) + j^2) \bmod M$ for $j = 1, 2, \ldots$

Counting Probes

When collisions are resolved with quadratic probing, the average number of probes required to search in a hash table of size *M* that contains *N* = αM keys is about

$$1 - \frac{\alpha}{2} + ln\left(\frac{1}{1-\alpha}\right) \qquad \text{for hits}$$

$$\frac{1}{1-\alpha} - \alpha + ln\left(\frac{1}{1-\alpha}\right) \qquad \text{for misses}$$

Effect of Load on # of Probes Quadratic Probing

α	**Average Probes: Successful Search**	**Average Probes: Unsuccessful Search**
0.1	1.06	1.12
0.2	1.12	1.27
0.3	1.21	1.49
0.4	1.31	1.78
0.5	1.44	2.19
0.6	1.62	2.82
0.7	1.85	3.84
0.8	2.21	5.81
0.9	2.85	11.40

Double Hashing

1. Hash search key to get initial table index
2. Use a second hash function to calculate an interval used to probe buckets after the table index, until search key or an open address is found

Collision resolution

- $t(key) \bmod M \Rightarrow$ table index
- If bucket at table index is full, then try
  - $(t(key) + j * t'(key)) \bmod M$ for $j = 1, 2, 3\ldots$
  - $t'(key) = q - (t(key) \bmod q)$ for some prime number $q < M$

Collision Resolution

Data Structures & Algorithms

Dynamic Hashing

Data Structures & Algorithms

Dynamic Hashing

- As the number of keys in a hash table increases, search performance degrades
- Separate Chaining
  - Search time increases gradually
  - Doubling # of keys means doubling (on average) list length at each of *M* table indices
- Linear Probing
  - Search time increases dramatically as table fills
  - May reach point when no more keys can be inserted

Simple Dynamic Hashing

- Double the size of table when it “fills up”
- Choose a load factor of 0.5
- Each item must be rehashed into the table
- Expensive, but infrequent

Amortized Analysis

- Cannot guarantee that each and every operation will be fast, but can guarantee that average cost per operation will be low
- Total cost is low, but performance profile is erratic
- Most operations are extremely fast, but some operations require much more time to rebuild the table

Amortized Analysis: Concept

- Each insert
  - Pays (small constant) cost to actually insert
  - Deposits other small constant (“balance”) in a bank
- First $M/2 - 1$: build up “balance”
- $(M/2)$th insertion
  - Faced with a big (not small constant) bill
  - Finds a big (not small constant) balance
- Net result
  - Each insert charged small constant costs
  - Some costs deferred

Amortized Analysis: Applied

- Assume linear probing (see table below)
- Start with table of size *M*
- Each insertion in a table ≤ ½ full
  - Average cost is ≤ 2.5 probes
- Insert *M*/2 - 1 keys
  - 2.5 * (*M*/2 - 1) = ***O*(*M*)**

α	Average Probes: Successful Search	Average Probes: Unsuccessful Search
0.1	1.1	1.1
0.2	1.1	1.3
0.3	1.2	1.5
0.4	1.3	1.9
0.5	1.5	2.5
0.6	1.8	3.6
0.7	2.2	6.1
0.8	3.0	13.0
0.9	5.5	50.5

Amortized Analysis: Applied

- Insert $(M/2)^{th}$ key
- Build new table, size $2M$
  - Remove keys from old table, insert in new
  - Each insert ≤ ¼ full, with average cost less than 1.5 probes (from table)
  - $1.5 * M/2 = \boldsymbol{O(M)}$
- $\boldsymbol{O(M) + O(M) = O(M)}$
  - Linear time to insert $M/2$ keys, but last one is at a higher cost than the previous inserts

Dynamic Hashing

Data Structures & Algorithms

Hash Template Demo

From a web browser:

`bit.ly/eecs281-hash-template-demo`

From a terminal:

```
wget bit.ly/eecs281-hash-template-demo -O htdemo.tgz
```

At the command line:

```
tar xvzf htdemo.tgz
g++ -std=c++1z hash-template-demo.cpp -o htdemo
./htdemo
```

Summary: Hashing

- Collision Resolution
  - **Separate Chaining** creates a linked list for each table index
  - **Linear Probing** uses empty places in table to resolve collisions
  - **Quadratic Probing** looks for empty table index at increasing distance from original hash
  - **Double Hashing** applies additional hash function to original hash
- **Dynamic Hashing** increases the table size when it reaches some pre-determined load factor