HO #17B
Hash supplementary
Fall 2022
Gary Chan

Hashing

(N:12)

Open Addressing via Linear Probing: Insertion

- Compute the home bucket $L = h(K)$
- if $T[L]$ is not empty, consider the hash table as circular:

```
for( i = 0; i < m; i++ )
    compute L = ( h(K) + i ) % m;
    if T[L] is empty, put K there and stop
```

- If we can't find a position (the table is completely full), return an error message

An Example

- *D* = 11
- Add 58: Collision with 80
- Add 24
- Add 35: Collision with 24
- Insertion of 13 will join two clusters

			80				40			65
0	1	2	3	4	5	6	7	8	9	10

		24	80	58			40			65
0	1	2	3	4	5	6	7	8	9	10

		24	80	58	35		40			65
0	1	2	3	4	5	6	7	8	9	10

Searching for linear probing

- Search begins at the home bucket $h(k)$ for the key k
- Continues by examining successive buckets in the table by regarding the table as circular until one of the following happens
  1. A bucket containing the element with key k is reached; in this case, we found the element;
  2. An empty bucket is reached; in which case, the element is not found
  3. We return to the home bucket; in which case, the element is not found

Clustering in Linear Probing

- We call a block of contiguously occupied table entries a cluster
- Linear probing becomes slow when large clusters begin to form (primary clustering)
- For example:
  - Once $h(K)$ falls into a cluster, the cluster will definitely grow in size by 1
  - Larger clusters are easier targets for collision in the future
  - If two clusters are only separated by one entry, then inserting one key into a cluster can merge the two clusters

Deletion

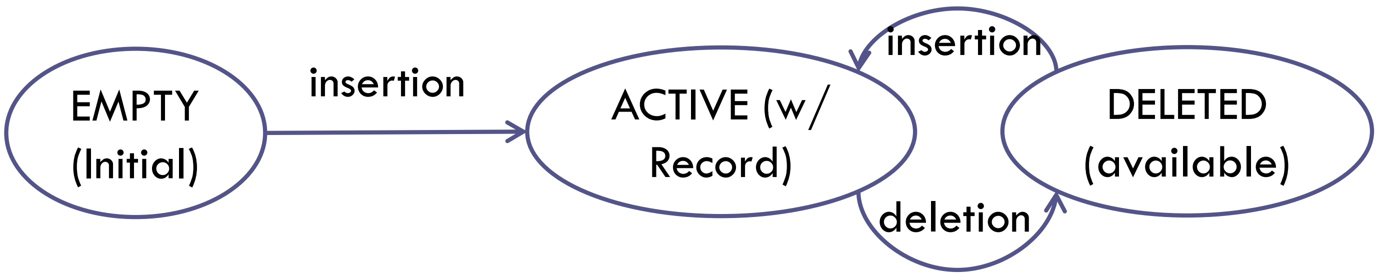
- May require several movement: we cannot simply make the position empty. E.g., consider deleting 58

		24	80	58	35		40			65
0	1	2	3	4	5	6	7	8	9	10

- Move begins just after the bucket vacated by the deleted element. We need to *rehash* buckets one by one in the remainder of the cluster.
- Clearly, it may lead to a lot of movements involving at worst the whole table (of O(m), where m is the size of the table)
- To reduce rehashing overhead, we can simply mark the entry "deleted," which means treating it belonging to the cluster but skipping it in a cluster inspection
  - We need to distinguish it from the empty bucket at the cluster boundary

Introduction of a State Field in a Bucket

- A new state variable is inserted into each bucket



- The condition for unsuccessful search is that an *EMPTY* bucket is reached (not *DELETED* because it belongs to the cluster)
- After a while, almost all buckets have this status field set to *ACTIVE* or *DELETED*, and unsuccessful searches examine all buckets
- To improve performance, we must reorganize the table by, for example, rehashing into a new fresh table

Linear Probing Performance

- Let m be the number of buckets in the hash table
- n elements are present in the table
- Worst case search and insert time is $\Theta(n)$ (all n keys have the same home bucket)
- Average performance: U_n and S_n be the average number of buckets examined during an unsuccessful and successful search, respectively, and $\alpha = n/m$ be the load factor:

$$\mathrm{U_n} \approx \frac{1}{2}\left[1 + \frac{1}{(1-\alpha)^2}\right] \qquad \mathrm{S_n} \approx \frac{1}{2}\left[1 + \frac{1}{1-\alpha}\right]$$

An Application Example

- A hash table is to store up to 1,000 elements. Need to find its hash table size.
- Successful searches should require no more than 4 bucket examination on average and unsuccessful searches should examine no more than 50.5 buckets on average
  - i.e., S_n <= 4 ➔ α <= 6/7
  - i.e., U_n <= 50.5 ➔ α <= 0.9
- We hence require $\alpha = min(6/7, 0.9) = 6/7$ and therefore b >= 1167
- We choose D to be $37 \times 37 = 1369$ (no prime factors less than 20)

Quadratic Probing

- Insertion
- Compute $L = h(K)$
- Quadric jump away from its home bucket
- If $T[L]$ is not empty:

```
for( i = 0; i < m; i++ )
    compute L = ( h(K) + i*i ) % m;
    if T[L] is empty, put K there and stop
```

- Helps to eliminate primary clustering
- However, if the table gets too full, this approach is not guaranteed of finding an empty slot!
- It may also never visit the home bucket again.

Double Hashing

- To alleviate the problem of primary clustering
- Use a second hash function h_2 when collision occurs
- Resolve collision by choosing the subsequent positions with a constant offset independent of the primary position
- Incrementally jump away from its home bucket in constant step size depending on the key
  - $H(K_i, 0) = h(\mathrm{Ki})$
  - $H(K_i, 1) = (H(K_i, 0) + h_2(K_i)) \bmod m$
  - $H(K_i, 2) = (H(K_i, 1) + h_2(K_i)) \bmod m$
  - . . .
  - $H(K_i,m) = (H(K_i,m-1) + h_2(K_i)) \bmod m$

Choice of h_2

- For any key K, $h_2(K)$ must be relatively prime to the table size m
- Otherwise, we will only be able to examine a fraction of the table entries
- For example, if $h_2(K) = m/2$ (not relatively prime to m), then for $h(K) = 0$ we can only examine the entries $T[0]$ and $T[m/2]$ and nothing else!
- The only solution is to make m prime, and choose r to be a prime smaller than m, and set $h_2(K) = r - (K \bmod r)$
  - E.g., if m = 37, we may pick r = 23 and hence $h_2(K) = 23 - K \bmod 23$
  - We may as well use r = 11, and hence $h_2(K) = 11 - K \bmod 11$

Hash Performance

Strategies for improved performance:

1. Increase table capacity (less collisions)
2. Use a different hash function

- Hash table capacity
  - Size of table is better to be at least1.5 to 2 times the size of the number of items to be stored
  - Otherwise probability of collisions would be very high

Comparison between BST and hash tables

BST	Hash tables
Comparison-based	Non-comparison-based
$O(\log n)$ time per operation	$O(1)$ time per operation
$O(n)$ space	$O(n)$ space
Keys stored in sorted order	Keys stored in arbitrary order
More operations are supported: min, max, neighbor, traversal	Only search, insert, delete
Can be augmented to support range queries	Do not support range queries
In C++: `std::map`	In C++: `std::unordered_map`

Q&A