CSC2001F: Data Structures II

Omowunmi Isafiade

Email: omowunmi.isafiade@uct.ac.za

Office: Room 306

Outline

- Methods of Creating Hash Functions
  - Division and Multiplication
- Linear Probing – A method of Collision Resolution
- Lazy Deletion
- Primary Clustering
- Summary

Hash Function: From Keys to Indices

- The mapping of keys to indices of a hash table is achieved using a hash function h(k), which usually comprises of two maps:
  - Hash code map: key $\rightarrow$ integer
  - Compression map: integer $\rightarrow$ [0, M-1]
- If your key is already an integer, no need for integer conversion (hash code map)
- A good hash function minimizes possibility of collisions
- M is the size of the array (so an index is a value between $0 \cdots M-1$)

Methods of Creating Hash Functions

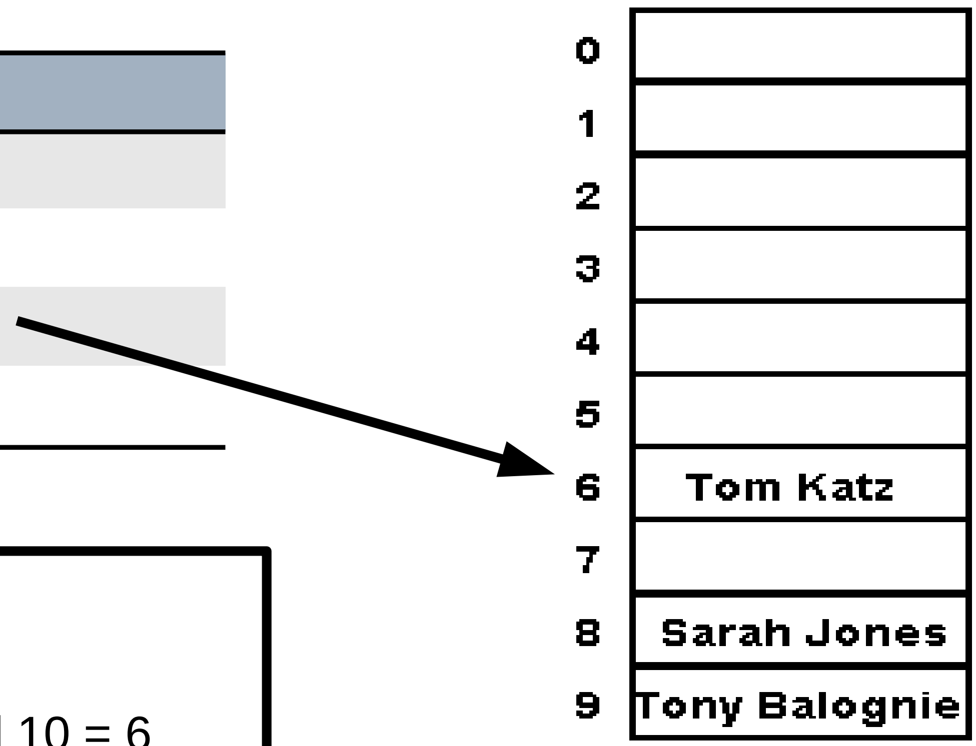
- **Division Method**
  - Map a key, *k,* into one of the slots m in the hash table by taking the remainder of *k* divided by m
  - So the hash function is:

    $h(k) = k \bmod m$
  - Note:
    - K is the key (integer value derived from the string conversion)
    - m is the size of the array (so an index is a value between 0 and m-1)

Division Method - Example

- Recall: hash code map (string conversion) and then h(k) evaluation

Name	Key
Sarah Jones	1038
Tony Balognie	1259
Tom Katz	746
John Smith	948



h (k) = k mod m

e.g. h (746) = 746 mod 10 = 6

Methods of Creating Hash Functions

- **Division Method**
  - $h(k) = k \bmod m$
  - **Choice of m is critical (division method)**
  - m prime is good
    - Ensure uniform distribution of keys
    - Prime is not too close to exact powers of 2
  - Note: m near power of 2 is not a good choice
    - Since h(k) gives the least significant bits of k
    - Many collisions result if the major difference in key values are in the ignored components (bits)

Methods of Creating Hash Functions

- **Multiplication Method**
  - Multiply the key *k* by a constant *A* in the range 0<A<1 and extract the fractional part of *KA*
  - Multiply this value by m (tableSize) and take the floor of the result
  - So the hash function is: $h(k) = \lfloor m(KA \bmod 1) \rfloor$
- **Advantage:** Value of m is not critical
- **Fibonacci Hashing** (Knuth's): $A = (\sqrt{5} - 1)/2$
  - multiplication hashing method in which the constant a is chosen uniquely as the **conjugate of the golden ratio**.

Multiplication Method - Example

- Multiplication Method
  - Suppose A = 0.62, insert the list of names in the table below into a hash table of size 10 using the function:

$$h(k) = \lfloor m(KA \bmod 1) \rfloor$$

Name	Key
Sarah Jones	1038
Tony Balognie	1259
Tom Katz	746
John Smith	948

h (746) = $\lfloor 10((746 * 0.62) \bmod 1) \rfloor$

10(0.52)= 5

→ h(1038) = 5

→ h(1259) = 5

→ h(746) = 5

→ h(948) = 7

A Case of Collision

- **Collision:** when two or more keys hash out to the same position.

$$h(k_i)=h(k_j)\ for\ k_i, k_j \in U, \wedge k_i \neq k_j$$

- $h(k)=\lfloor m(KA \bmod 1)\rfloor$

Name	Key
Sarah Jones	1038
Tony Balognie	1259
Tom Katz	746
John Smith	948

Results in collision

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

h (746) = $\lfloor 10((746 * 0.62) \bmod 1)\rfloor$

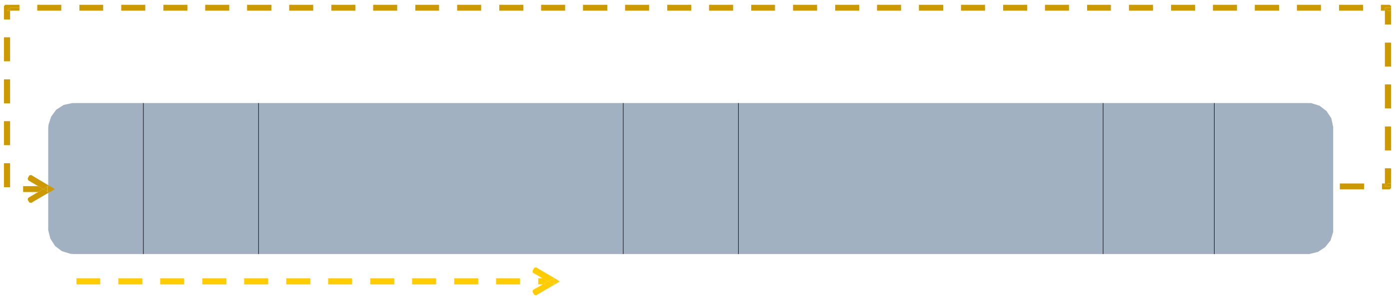
= 5

$\rightarrow$ h(1038) = 5
$\rightarrow$ h(1259) = 5
$\rightarrow$ h(746) = 5
$\rightarrow$ h(948) = 7

How do we resolve collision?

What is Linear Probing?

- Method of collision resolution
- A collision occurs, in a hash table, when two or more keys hash out to the same index (not one to one)
- **Linear probing:** resolves collision by scanning the array sequentially with wraparound until an empty slot is found.



Linear Probing - Method

- If current location is occupied, try the next location.

```
LinearProbingInsert( K )
If (table is full) throw an exception
    probe = h (k);
While (table [probe] occupied)
    probe = (probe + 1) mod m;
table[probe] = k;
```

- Use less memory than chaining, since one does not have to store those links (we will see chaining later).
- BUT! slower than chaining since one might have to search through the table for a long time.

Linear Probing – An Example

- **Collision resolution**

Name	Key
Sarah Jones	1038
Tony Balognie	1259
Tom Katz	746
John Smith	948

0	
1	
2	
3	
4	
5	Sarah Jones
6	
7	
8	
9	

What happens if we need to insert others at the same position?

Linear Probing – An Example

STEP 1

0	
1	
2	
3	
4	
5	Sarah Jones
6	
7	
8	
9	

STEP 2

0	
1	
2	
3	
4	
5	Sarah Jones
6	Tony Balognie
7	
8	
9	

STEP 3

0	
1	
2	
3	
4	
5	Sarah Jones
6	Tony Balognie
7	Tom Katz
8	
9	

STEP 4

0	
1	
2	
3	
4	
5	Sarah Jones
6	Tony Balognie
7	Tom Katz
8	John Smith
9	

Linear Probing – Search Operation

- To search for an item, we go to location "**h(k)**" and scan through **successive slots** until we find the item or encounter an empty location.
  - **Successful search:** to search for "Tom Katz" in the previous example, we go to **location 5** and continue scanning...(**location 7**)
  - **Unsuccessful search:** we compute the index and check sequentially, once we **encounter an empty slot, then the item we are searching for is not in the hash table ???**

Exercise – In Class

- Consider the input {437, 123, 617, 519, 456, 674, 199, 103, 93, 63}, a fixed table size of 10, and a hash function h(k) = k mod m
- Show the resulting linear probing hash table
- Now explain what the effect of deleting 123 from the hash table would be. For example what would happen if we were looking for 103, at a later stage?

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Lazy Deletion

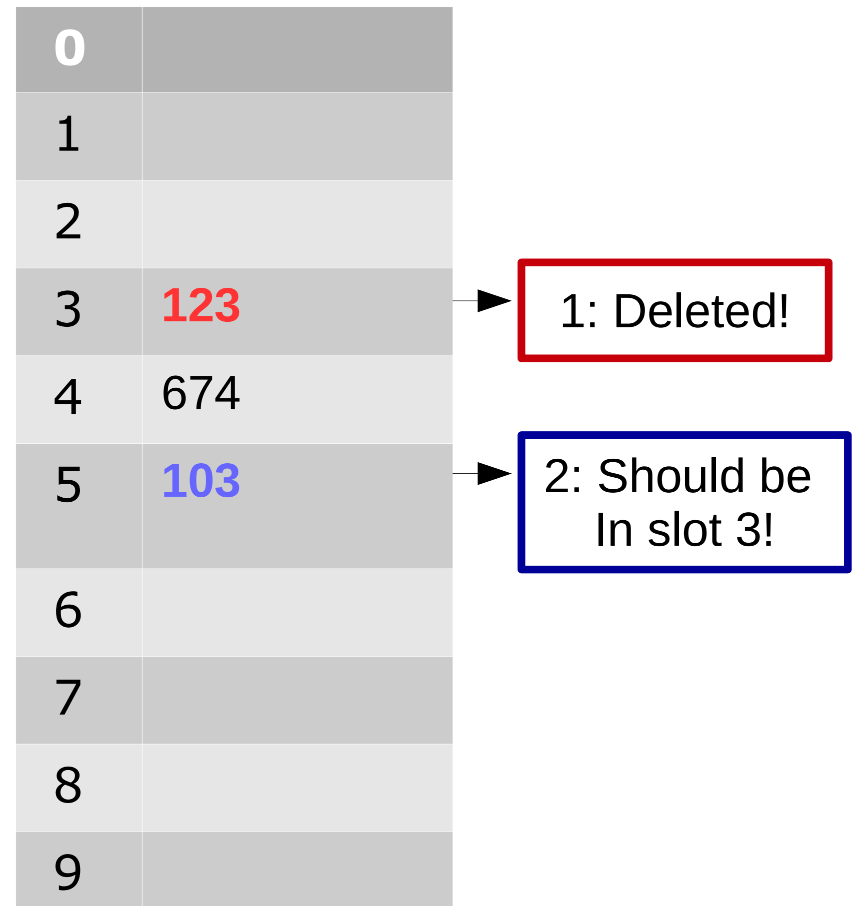
- What happened when an empty slot (cell) cannot be found? How do we avoid “crushing” already inserted values in the hash table?
  - Note: an item in the hash table not only represents itself, connects other items…

- **Definition:** “Lazy deletion is a technique that is used to **mark elements** in a hash table as deleted instead of physically removing them from the table”

- Mitigate the “problem” of (false) **unsuccessful search,** where an item is in the table but cannot be found.

Lazy Deletion - Usefulness

- Standard deletion – not possible because of the interconnections between hash table items.
- Hash table elements serve as place holders for collision resolution
- Example: removing 123 would cause all remaining “Search” operations to fail



Lazy Deletion - Usefulness

- Solution: Mark each slot as either active or deleted. Can use a binary marker – e.g. 1 = active; 0= deleted.

Index	State	Value
0	0	
1	0	
2	0	
3	**1**	**123**
4	1	674
5	1	103
6	0	
7	0	
8	0	
9	0	

Inactive/deleted

active

Primary Clustering

- Formation of large clusters of occupied cells causing expensive insertions

- Keys that hash into any of these large clusters of occupied cells require excessive attempts to resolve collisions

0	199
1	
2	
3	**123**
4	103
5	674
6	456
7	437
8	617
9	419

Primary cluster, example

Input: {337, 123, 617, 519, 456, 674, 199, 103}

Primary Clustering

- Case of inserting 505

  h(505, 10) = 505 mod 10 = 5

0	199
1	
2	
3	**123**
4	674
5	**675**
6	456
7	437
8	617
9	419

Impossible to insert!
Search for next available slot!

Primary cluster, example

Primary Clustering

- Items that collide because of identical hash function results in degenerate performance
  - Multiple attempts to find empty cells
  - Cluster increases in size for every collision resolution (insertion)

- Collision with alternative locations for other items cause poor performance

Load Factor (α)

- Number of elements in a hash table divided by the size of the hash table ($\alpha \in [0,1]$)
- That is the fraction of the hash table that is full
- The load factor ranges from 0 (empty) to 1(full)
- **Note:** can be greater than 1, we will see this later (separate chaining)

Expected Number of Probes (Naive Analysis)

- If independence of probes is assumed the average number of cells examined during an insertion using linear probing is $1/(1-\alpha)$
- A table with load factor of α has a probability of $1-\alpha$ of a cell being empty
- Example: in a table of size 10 with 5 slots filled we have a load factor of 0.5 (the number of probing trials is 1/0.5 =2

Expected Number of Probes (Naive Analysis)

- This is based on the fact that if the probability of an event is p then 1/p trials are needed till it occurs
- But what happens if we have a larger table say of size 100 and a load factor of 0.9
- Naive method indicates only 10 probes!
- Not exactly true because we have to be checking at least 90 slots in the worst case

Expected Number of Probes (A better estimate)

- A better way of getting an estimate for larger arrays is

$$\left(1+1/\left(1-\alpha\right)^{2}\right)/2$$

- In this case we see that at least 50 trials are needed for an insertion
- Note:
  - To avoid primary clustering, use a heuristic to ensure new insertions do not result in a table that is more than half full
  - That is the load factor is always < 0.5

Other Observations...

- Linear probing searches for a free slot sequentially
- Problematic for large tables with small groupings of clusters scattered arbitrarily (high load factors)
- Can take a long time to find a free slot. Too many markers degrade performance (rehash if necessary)
- Expanding table size to cope with higher load factors is impractical

Index	Value
0	199
1	
2	
3	**123**
4	674
5	
6	456
7	437
8	617
9	519