Hash Tables

Hussein Suleman <hussein@cs.uct.ac.za>

Department of Computer Science
School of IT
University of Cape Town

2020

Hash Tables

- Data structure where items are stored in a location determined by their content.
  - Content-based index
    - vs.
  - Comparison-based index

- Every hash table is fundamentally composed of:
  - Array of items
  - Hash function: item->index

Trivial Hash Table

- Suppose our data is a subset of unique numbers from 0..n-1.
- Create an array A of integers.
- Insert(x) algorithm: A[x] = 1
- Delete(x) algorithm: A[x] = 0
- Find(x) algorithm: A[x]==1?

Data to insert
4 3 0 5

0	1	2	3	4	5	...	n-1
1	0	0	1	1	1		

Trivial Hash Table, with Duplicates

- Suppose our data is a subset of numbers from 0..n-1. Duplicates are allowed.
- Create an array A of integers.
- Insert(x) algorithm: A[x]++
- Delete(x) algorithm: A[x]--
- Find(x) algorithm: A[x]>0?

Data to insert
4 3 0 5 0 3 0

0	1	2	3	4	5	...	n-1
3	0	0	2	1	1		

Analysis of Trivial Hash Table

- Insert: O(1)
- Delete: O(1)
- Find: O(1)

Issues to resolve

- What if there are more keys than slots?
  - overflow
- What if key is not an integer?
  - map
- What if keys are the same, but values are different?
  - collision

Hash Function

- A hash function is a mapping from an item to an integer.
- Use modulus to solve the integer key size problem:
  - hash(x) = x % tableSize
- One-way function is fine (but may cause collisions).
- Produces values in the range 0..tableSize-1

Hash Function 1

- Hashing strings - add together Unicode values and mod tablesize.

```
public int hash1 ( String key )
{
   int hashVal = 0;

   for( int i = 0; i < key.length(); i++ )
      hashVal += key.charAt(i);

   return hashVal % tableSize;
}
```

Hash Function 1 Analyzed

- h("abc") = (97+98+99) % tableSize

Length	First String	Last String	Range of h(s) (before %)
1	a	z	97-122
2	aa	zz	194-244
3	aaa	zzz	291-366
4	aaaa	zzzz	388-488

- Poor hash function for large tableSize and small keys.
- Also, h("abc")=h("cab")=h("bac") causes collisions.

Hash Function 2

- Solve the uniqueness problem by multiplying/shifting each character by some value. This also increases hash values.

```
public int hash2 ( String key )
{
   int hashVal = 0;

   for( int i = 0; i < key.length(); i++ )
      hashVal = (37 * hashVal) + key.charAt(i);

   return hashVal % tableSize;
}
```

Hash Function 2 Analyzed

- h("abc") = (((97)*37+98)*37+99) % tableSize

Length	First String	Last String	Range of h(s) (before %)
1	a	z	97-122
2	aa	zz	3686-4636
3	aaa	zzz	136479-171654
4	aaaa	zzzz	5049820-6351320

- Larger hash values - better spread.
- Also, h("abc")!=h("cab")!=h("bac").
  - h(abc) =136518 % tableSize,
  - h(bac) =137850 % tableSize, etc.

Requirements of Good Hash Functions

- Fast to compute.
- Deterministic.
- Spread keys evenly in hash table.

- What other hash functions could we use?

Perfect Hash Function

- A perfect hash function maps every distinct key onto a distinct integer.
- For example:
  - If keys are 2-letter words (26*26 combinations),
  - If tableSize=1000,
  - h(x) = (x[0]-'a')*30+(x[1]-'a')
  - Still some gaps/holes.
- Minimal perfect hash function has no holes in the array.

Collision Resolution Approaches

- Open Addressing
  - Linear Probing
  - Quadratic Probing
- Closed Addressing
  - Chaining

Collision Resolution by Linear Probing

- Insertion algorithm:
  - Generate hashcode h=hash(key)
  - While A[h] contains a key
    - h=(h+1) % tableSize
  - A[h] = {key, value}
- Find algorithm:
  - Generate hashcode h=hash(key)
  - While A[h] contains a key
    - if (A[h]{key}==key) return A[h]{value}
    - h=(h+1) % tableSize
  - return Not Found

Linear Probing Example

- Using tableSize=10 and h(x)=x % 10
  - insert: 23, 56, 13, 93, 33, 36, 89, 99

A	A	A	A	A	A	A	A
							99
23	23	23	23	23	23	23	23
		13	13	13	13	13	13
			93	93	93	93	93
	56	56	56	56	56	56	56
				33	33	33	33
					36	36	36
						89	89

Linear Probing Issues

- Primary Clustering
  - Multiple adjacent items and slow performance.
- Uneven gaps in table.
- Table Full
  - What to do when table is full?
    - Throw an error?
    - Create new and larger table?

Linear Probing Analysis

- The load factor λ is the proportion of the table that is full. Proportion of empty table is (1-λ).
- Probability of a cell being empty is (1-λ).
- Ave. number of cells examined for insert (failed find):
  - $T(\lambda) = (1+1/(1-\lambda)^2)/2$
  - $T(0.5) = (1+1/0.5^2)/2 = 2.5$; $T(0.1) = (1+1/0.9^2)/2 = 1.117$
  - $T(0.9) = (1+1/0.1^2)/2 = 50.499$
- Ave. number of cells examined for successful find:
  - $T(\lambda) = (1+1/(1-\lambda)/2$
  - $T(0.5) = (1+1/0.5)/2 = 1.5$; $T(0.1) = (1+1/0.9)/2 = 1.056$
  - $T(0.9) = (1+1/0.1)/2 = 5.5$

Collision Resolution by Quadratic Probing

- Insertion algorithm:
  - Generate hashcode h=H=hash(key), i=1
  - While A[h] contains a key
    - h=(H+i*i) % tableSize
    - i++
  - A[h] = {key, value}
- Find algorithm:
  - Generate hashcode h=H=hash(key), i=1
  - While A[h] contains a key
    - if (A[h]{key}==key) return A[h]{value}
    - h=(H+i*i) % tableSize
    - i++
  - return Not Found

Quadratic Probing Example

- Using tableSize=10 and h(x)=x % 10
  - insert: 23, 56, 13, 93, 33, 36, 89, 99

A	A	A	A	A	A	A	A
					36	36	36
				33	33	33	33
23	23	23	23	23	23	23	23
		13	13	13	13	13	13
	56	56	56	56	56	56	56
			93	93	93	93	93
							99
						89	89

Quadratic Probing Analysis

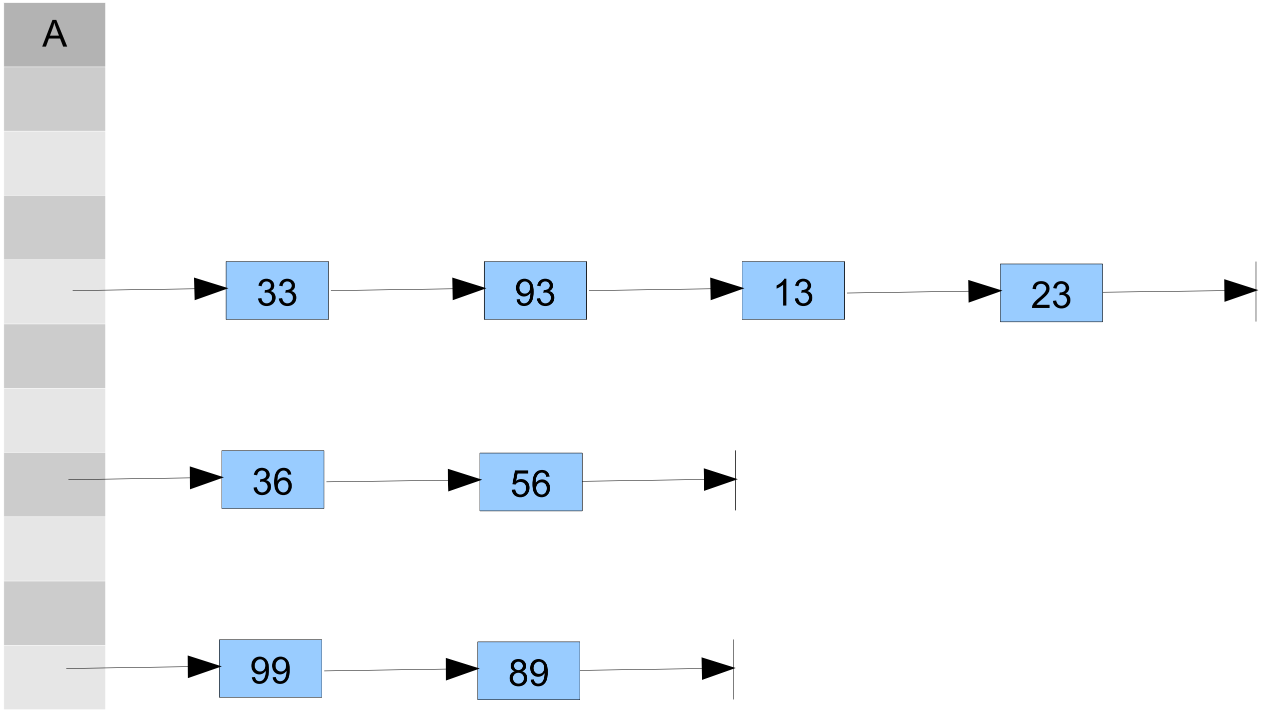
- If table size is prime and $\lambda<0.5$, then we never check same cell twice and new element can always be inserted.
- Rough proof sketch:
  - Choose a prime M=tableSize
  - Choose 2 distinct i and j values <M/2
  - Suppose $H+i^2 \equiv H+j^2 \pmod{M}$
  - then, $(i-j)(i+j) \equiv 0 \pmod{M}$ so M has a factor
  - By contradiction, first M/2 positions checked are distinct

Collision Resolution by Chaining

- Use a table of pointers/references.
- Each new item must be added to a linked list at that position in the table.
- Insertion algorithm:
  - Generate hashcode h=hash(key), p=new Node
  - p.data = {key, value}; p.next = A[h]
  - A[h] = p
- Find algorithm:
  - Generate hashcode h=hash(key), p=A[h]
  - While p!=null and p{key}!=key
    - p = p.next
  - return p

Chaining Example

- Using tableSize=10 and h(x)=x % 10
  - insert: 23, 56, 13, 93, 33, 36, 89, 99



Chaining Analysis

- Assume N items
- Assume tableSize=M
- Assume even distribution of hash values and use of all possible hash values
- Then, on average, each LL is of size N/M
- Average time to search=1/2 N/M
- Worst case O(N); Best case O(N/M)
  - It is all about the choice of M relative to N!

Handling deletions

- Open addressing:
  - Deletion is not easily possible.
  - Add a flag to an item to mark it as deleted.
    - Skip deleted items during search.
    - Unmark and overwrite deleted items on insert.
- Chaining:
  - Use linked list deletion operations.

Other Variations

- Double Hashing
  - Secondary clustering is where a key generates the same sequence of locations to check.
  - Maybe use a second hash function when there is a collision.
  - h = H1 + H2, where H2 is the rehashing function
  - A different sequence is checked for each key.

- Chaining using BSTs or other data structures.

that's all folks!