COMP20003
Algorithms and Data Structures
Hash Tables

Nir Lipovetzky
Department of Computing and Information Systems
University of Melbourne
Semester 2

So far…

- Dictionary search has been based on key comparisons.
  - Linked list (sorted and unsorted)
  - Array (sorted and unsorted)
  - Binary Search Tree
  - Balanced Trees

This section

- A way of going directly to the desired item.

- Hash tables:
  - Search usually takes only 1 (or few) operations.
  - (on average)
  - (if managed well)
  - (but very bad worst case)
- Probabilistic data structure.

Textbook

- Skiena: Chapter 3, Section 3.7

Direct storage and search (a hypothetical fast method)

- Task: Store in dictionary items with keys within the range `0` to `RANGE-1`.
- Data structure: array:

```
itemtype    A[RANGE];
```

- Operations:

```
initialize(A)
   {for(i=0;i<RANGE;i++) A[i] = NULL;}
insert(item) {A[item->key] = item;}
search(key) {return A[key];}
```

Direct storage and search (a hypothetical fast method)

- Create table and initialize:

```
#define RANGE 10000
#define EXAMPLEKEY 8179
struct item{int key; char *info;} item;
item  *A[RANGE];
item  *newitem;

for(i=0;i<RANGE; i++) A[i] = NULL;
```

Direct storage and search (a hypothetical fast method)

- Insert item with key=**`EXAMPLEKEY`**:

```
newitem = (item *)malloc......
newitem->key = EXAMPLEKEY;
newitem->info = ...malloc....strcpy...
A[EXAMPLEKEY] = newitem;
```

- Search for item with key=**`EXAMPLEKEY`**:

```
return A[EXAMPLEKEY];
```

Limitations?

- Key range is often not known.
- **`RANGE`** might be too large to be a practical array size.
- We have assumed that keys are unique.

Use the idea: make it practical

- Size:
  - Squash the keys to fit into an array:
    - A[100]
    - Store key in A[key%100]
- Collisions
    - If key1 = 200 and key2 = 400, both map to A[0].
    - Collisions are *always* possible, so *must* have a plan.
- Patterns
    - Use more complicated mapping of keys to disrupt patterns.

Patterns Excercise

Key = Input % modulo

Input	Modulo 8	Modulo 7
0		
4		
8		
12		
16		
20		
24		
28		

Use the idea: make it practical

Input	Modulo 8	Modulo 7
0	0	0
4	4	4
8	0	1
12	4	5
16	0	2
20	4	6
24	0	3
28	4	0

Use more complicated mapping **and prime numbers to disrupt patterns**.

Hash Functions

- `hash(key)`
- Hash function maps the item's key to an array slot.
  - `A[hash(item->key)] = item;`
- Desirable features and requirements of a hash function:
  - Must output value within bounds of the array.
  - Should minimize collisions, as far as possible.
  - Should spread items throughout the table,

Hash Functions

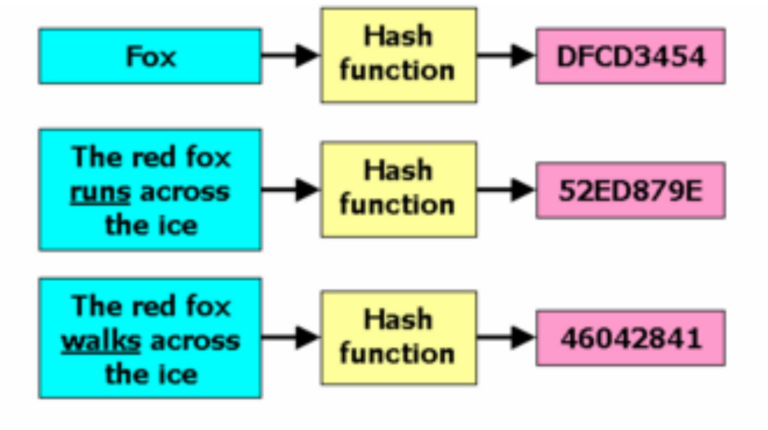
- Some bad hash functions
  - `A[100]; hash(key) = key%10`
  - `A[100]; hash(key) = key%100`
- Better:
  - `A[97]; hash(key) = (key*BIGPRIME)%97`
- Prime numbers help disrupt patterns in the data and spread it throughout the table.

Hash Functions

Student numbers example:

- **3 first numbers**
- **3 last numbers**
- **0-9 buckets**

Hash functions for strings



Record with string key *s*, array dictionary size **`SIZE`** might be stored in location:

`Example:`

`A[(17*s[0] + 37*s[2] +101*length(s))%SIZE]`

Hash functions for strings

- Skiena (p.89) shows mapping strings to number, base alphabet size:

$$H(S) = \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \times char(s_i)$$

- **`H("cat") = 26`^{2} `* 3 + 26 * 1 + 1 * 20`**
- Does this work for longer strings?
- Is this efficient?

Hash functions for strings

- More efficient:
- Use a power of 2 instead of alphabet size:
- **H(“cat”) = 32^2 * 3 + 32 * 1 + 1 * 20;**

```
hashcat = (('c')*(1<<10)) +
           (('a')* (1<<5)) +
            ('t'))%TABLESIZE;
hashcat = ((‘c’ << 10) +
            (‘a’ <<5)  +
            (‘t’))%TABLESIZE;
```

Hash functions for strings

- More efficient *and* prevent overflow:
- Use a power of 2 instead of alphabet size:
- **H("cat") = 32^2 * 3 + 32 * 1 + 1 * 20;**

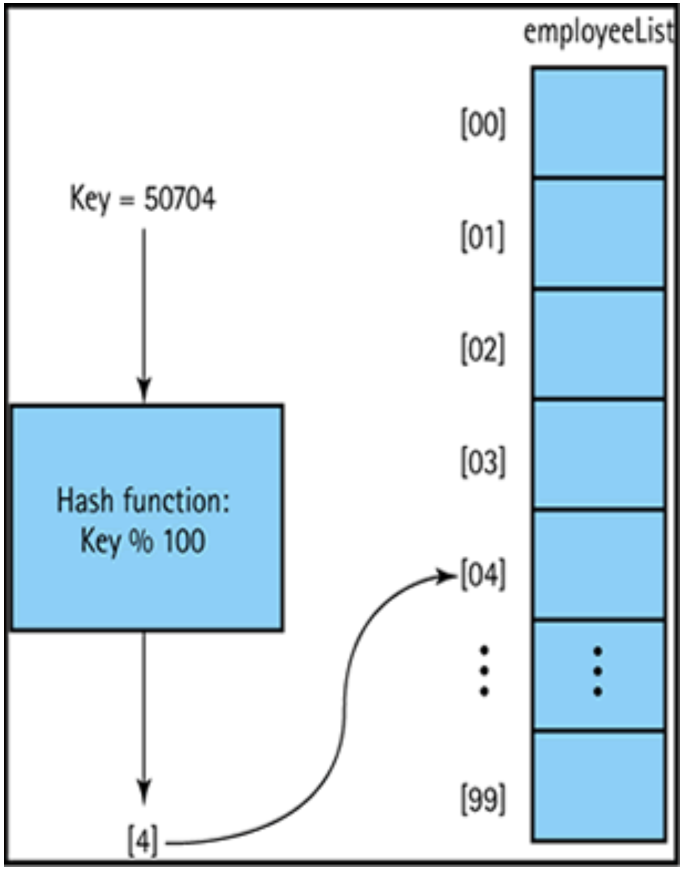
```
hashcat =((((('c'* 1<<10) %TABLESIZE)
          +('a' * 1 <<5)%TABLESIZE)
          +'c')%TABLESIZE;
```

Principle:

$$(a+b) \bmod n = ((a \bmod n) + (b \bmod n)) \bmod n$$

Hash tables: key idea

- Huge range of possible keys
  - e.g. space of possible surnames: 26^n
    - 26^{10} = 141,167,095,653,376
- Map to a smaller set of array indexes, 0..m-1
  - hash function: *h*
  - easily computed
  - even distribution



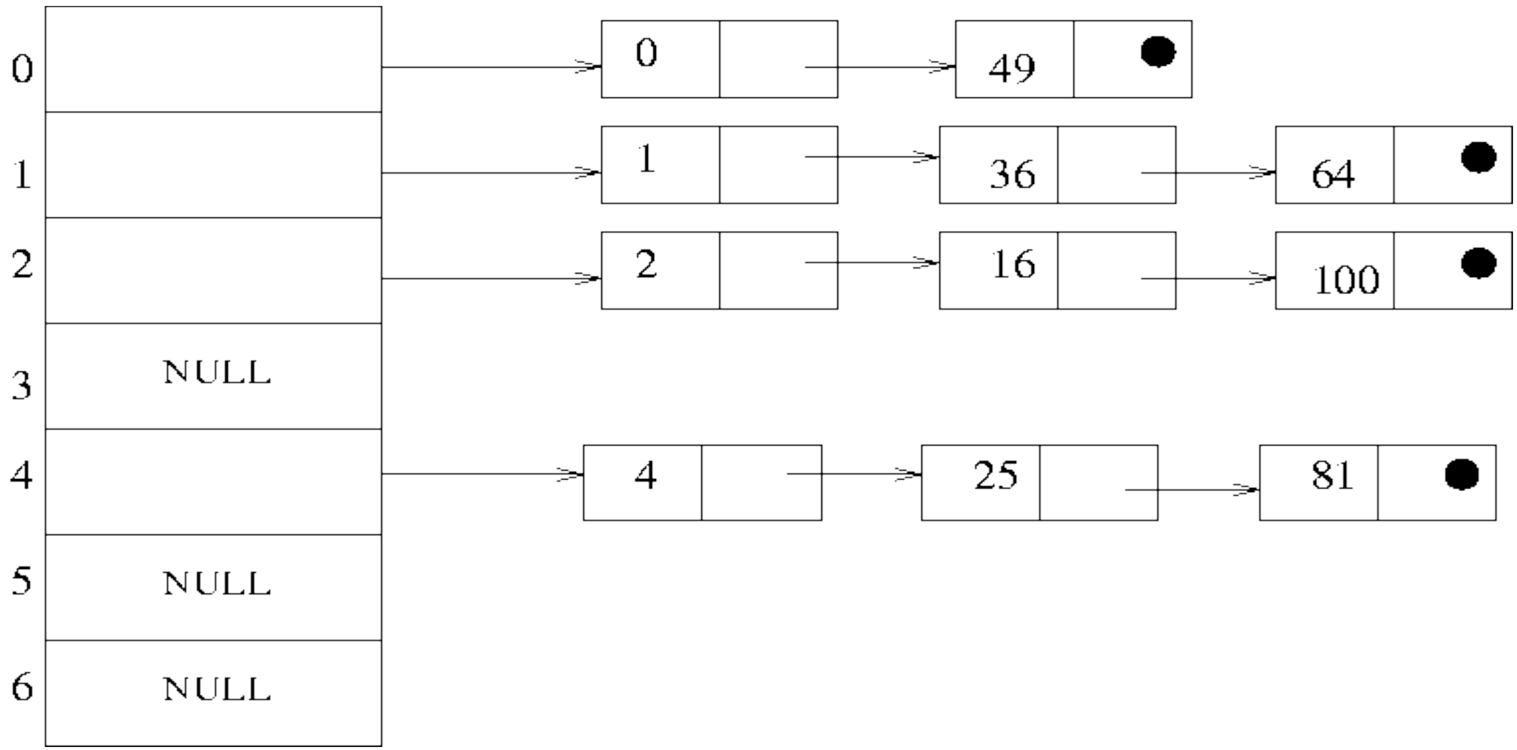
Collisions

- Collision: Two keys map to the same array index
  - $\mathtt{h(k_1)} == \mathtt{h(k_2)}$
- Where array **`SIZE`** < number of records, there will *definitely* be collisions.
- Where array **`SIZE`** > number of records, there will often still be collisions – and we *must* handle them.
- Good hash functions have fewer collisions
  - but *we can never assume there will be none*.

Collision Resolution Methods

- Chaining
- Open adressing methods
  - Linear probing
  - Double hashing

Chaining



Average length of the lists?

```
insert(HT,item)
{
   new newnode = /* make a list node */
    /* put item in the list node */
   index = hash(item->key);
   if(HT[index]==NULL) HT[index]=newnode;
   else
   {     newnode->next = HT[index]->node;
         HT[index] = newnode;
   }
}
```

Linear chaining

- What happens if you forget to null the table in the beginning?
- What happens if all the items hash to the same location?
- What happens if the number of items is much bigger than the table?

Chaining: analysis

- Insertion:
  - Best case
  - Worst case
  - Average Case
- Search
  - Best case
  - Worst case
  - Average Case

Chaining: analysis

- Average case: fast lookup when table is not heavily loaded.
- Performance degrades gracefully when table gets crowded.
- Eventually degenerates to a linked lists.

- Extra time and space for pointers.

Open addressing: Linear probing

- If there is a collision, put the item in the next available slot:

```
while (HT[index] != NULL)
      index= (index+1)%TABLESIZE
/* only get out of this loop when
  get to a vacant spot */
```

Open addressing: linear probing

m = 20, f(k) = k % m

Initial Situation

–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1	–1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

After inserting 34, 55, 12, 8, 45, 37, 32, 88, 98, 54

–1	–1	–1	–1	–1	45	–1	–1	8	88	–1	–1	12	32	34	55	54	37	98	–1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
					0			0	1			0	1	0	0	2	0	0	

After inserting 21, 42, 56, 74, 52, 33, 16

74	21	42	52	33	45	16	–1	8	88	–1	–1	12	32	34	55	54	37	98	56
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
6	0	0	11	11	0	10		0	1			0	1	0	0	2	0	0	3

Linear probing

- What happens when:
  - HT lightly loaded?
  - HT heavily loaded?
  - HT full?

Linear probing: Biggest problems

- Catastrophic failure when table full.
- Clustering. Once things start to go bad in part of the table…

Double hashing

- Choose a second hash function.
- Reduces clustering.

```
jumpnum = hash2(key);
while (HT[index] != NULL)
   index=(index+jumpnum)%TABLESIZE
```

Example **`hash2`** function:

```
hash2(key) = key%SMALLNUMBER + 1;
```

Open addressing: Analysis

- Consider load factor α
  - for n keys
  - in m cells
  - $\alpha = n/m$

Open Addressing: Analysis

- Average case:
- Under some simplifying assumptions, expected time for insertion is:
  - Double hashing: $1/(1-\alpha)$
  - Linear probing: $1/(1-\alpha)^2$
  - Example: $\alpha = 0.75$
    - Double hash insertion: 4 probes
    - Linear probing insertion: 16 probes
  - A nice explanation of the assumptions, by Tim Roughgarden:
  - https://class.coursera.org/algo-004/lecture/70?s=e

Open Addressing: Analysis

- Average case lookup:
  - Double hash ~ ½(1 + 1/(1-α))
  - Linear probing ~ ½(1 + 1/(1-α)^2)

	Double hash	Linear probe
α	$\frac{1}{2}(1+\frac{1}{1-\alpha})$	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$
50%	1.5	2.5
75%	2.5	8.5
90%	5.5	50.5

Open Addressing: Analysis

- Severely degraded performance as table nears full.

α	$\frac{1}{2}(1+\frac{1}{1-\alpha})$	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$
50%	1.5	2.5
75%	2.5	8.5
90%	5.5	50.5

- Catastrophic failure when table full.
- Performance depends on α (*n/m*), so choice of table size must be appropriate.

Open Addressing: Analysis

- Severely degraded performance as table nears full.

α	$\frac{1}{2}(1+\frac{1}{1-\alpha})$	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$
50%	1.5	2.5
75%	2.5	8.5
90%	5.5	50.5

- Catastrophic failure when table full.
- How and why do people use open addressing?

Open Addressing: Analysis

- Severely degraded performance as table nears full.

α	$\frac{1}{2}(1+\frac{1}{1-\alpha})$	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$
50%	1.5	2.5
75%	2.5	8.5
90%	5.5	50.5

- Catastrophic failure when table full.
- How might you prevent degraded performance?

Hash tables: Summary

- O(1) lookup!!
  - But only on average.
  - And only for small α
  - Some bad worst cases:
    - Table full (open addressing)
    - Table near full (open addressing)
    - Everything hashes to same/similar slot (all)

Hash tables: Summary

- Performance degrades:
  - For linear chaining, degrades gracefully.
  - For open address chaining, degrades, then can fail catastrophically.
- Cannot retrieve items in sorted order.
- A nice review of hashing, including some advanced topics:
- http://courses.csail.mit.edu/6.006/fall10/lectures/lecture5.pdf
- http://courses.csail.mit.edu/6.006/fall10/lectures/lecture6.pdf
- http://courses.csail.mit.edu/6.006/fall10/lectures/lecture7.pdf

Some notes about hash tables

- Hash tables show fast lookup.
  - O(1) lookup.
  - Better than log n.
- Used in non-time critical applications.
- Often used in CS applications.

Other uses of hashing

- Duplicate detection, *e.g.* for documents.
  - If hash signatures are different, documents can't be duplicates. Only have to thoroughly check a few documents.
- Plagiarism detection.