COMP 3760: Algorithm Analysis and Design

Lesson 8: Hashing

Rob Neilson

rneilson@bcit.ca

Homework (due 8:30 Monday next week)

- Reading for next week:
  - read chapters 6.4
- Exercises:
  - there is one question that you must sketch a solution for
  - you are expected to explain exactly how to solve the problem, providing pseudocode as appropriate
  - the actual question/problem is available as Homework 5 in webct
- Important:
  - for this week only all students in all sets are required to submit their solution to webct, *PRIOR TO 8:30AM MONDAY OCT 6*
  - late submissions will not be accepted; no exceptions

Fast Storage of Keyed Records

Goal: want some way to do fast storage/lookups/retrieval of information, based on an arbitrary key

eg: **`key = A00012667`**
`value = Robert Neilson`

Let's consider traditional data structures …

Array: How would you use an array (or arrays) to store this

- use either 2 1D arrays or 1 2D array or an array of objects
- store key in a sorted array (for fast retrieve)
- use the second array (or column) to store the record or a pointer to the record … or …
- create an object 'Employee', and store in an array of objects

Using Arrays to Store Keyed Records

2 1D Arrays …

1	A00012667
2	A00666666
3	
4	
	⋮
n-1	
n	

1	neilson
2	beelzebub
3	
4	
	⋮
n-1	
n	

1 2D Array …

1	A00012667	neilson
2	A00666666	beelzebub
3		
4		
	⋮	⋮
n-1		
n		

Using Arrays to Store Keyed Records (2)

Inserting a new element … eg: ***`insert(A00099999, “foo”)`***

1	A00012667	neilson
2	A00066666	beelzebub
3	A00100000	186A0
4	A00111111	jimmy
5	A00123456	n(n+1)/2
6	A00444444	bertcubed
7	A00666666	Beelzebub
8		
9		
10		

Using Arrays to Store Keyed Records (3)

Inserting a new element … eg: ***`insert(A00099999, "foo")`***

1	A00012667	neilson
2	A00066666	beelzebub
3	A00100000	186A0
4	A00111111	jimmy
5	A00123456	n(n+1)/2
6	A00444444	bertcubed
7	A00666666	Beelzebub
8		
9		
10		

`find location`

- **`(use binary search)`**
- **`O(logn) operation`**

Using Arrays to Store Keyed Records (4)

Inserting a new element … eg: ***`insert(A00099999, “foo”)`***

1	A00012667	neilson
2	A00066666	beelzebub
3		
4	A00100000	186A0
5	A00111111	jimmy
6	A00123456	n(n+1)/2
7	A00444444	bertcubed
8	A00666666	Beelzebub
9		
10		

```
find location
      - (use binary search)
      - O(logn) operation
```

```
create space
      - (move existing elements)
      - O(n) operation
```

Using Arrays to Store Keyed Records (5)

Inserting a new element ... eg: ***`insert(A00099999, “foo”)`***

1	A00012667	neilson
2	A00066666	beelzebub
3	A00099999	foo
4	A00100000	186A0
5	A00111111	jimmy
6	A00123456	n(n+1)/2
7	A00444444	bertcubed
8	A00666666	Beelzebub
9		
10		

```
find location
      - (use binary search)
      - O(logn) operation

create space
      - (move existing elements)
      - O(n) operation

put the new element
      - direct access to array
      - O(1) operation
```

Overall efficiency is:

$O(\log n)+O(n)+O(1) = O(n)$

Using Arrays to Store Keyed Records (6)

- using the same type of operations, we will find:
  - *search* operation is O(logn)
  - *retrieval* is O(logn)
  - *deletion* is O(n)

Note: we can improve insertion and deletion to O(logn) by using a balanced tree (a non-trivial data structure)

What if we use an unsorted Array:

- *insertion* will be much faster – O(1)
- *searching, retrieve* will be slower – O(n)
- *deletion* will be the same O(n)

*Will an **unsorted linked list** perform any better?*

- no – it will be the same as the unsorted array

*Will a **sorted linked list** perform any better?*

- we can't do binary search – so *searching, retrieve* will be O(n)
- *insert, delete* will also be O(n) as we have to first find the location
- so it will actually give us the worst performance

- *So how to get better performance … ?*

Observe ...

arrays are fast when we know which element we want

(ie: so maybe we can exploit this direct access capability)

Example:

- adding at end is O(1)
- retrieving a specific element (eg A[5]) is O(1)

Why?

- the key is the array index, so we can just store/retrieve directly

How can we exploit this idea?

- maybe we can make each key be an index into the array
- this way we could store it directly in one access

This is the idea behind Hashing!

- we use a function to map the items to be stored into the array locations

eg: an array uses f(index) = index

.... we can use f(key) = index

Hash Tables

- a hash table is a data structure that uses a function

  f(key) = index

  … to map the key into a storage location in an array

For example:

- assume ***f("dog") = 4***
- then we store dog at location 4

1	
2	
3	
4	dog
5	
6	

Hash Functions

hash function

- the function that maps a item to be stored to a storage location

- the most common hash function for numerical keys is simply:
  - ***K mod m*** - where ***K*** is the key and ***m*** is the size of the storage array

- ie: index = K mod m
  - this always results in a value between **0..m-1**

Example:

- assume m=5

0	10
1	6
2	
3	3
4	29

then the storage location for some keys are:

K	K mod m
3	3mod5 = 3
6	6mod5 = 1
10	10mod5 = 0
29	29mod5 = 4

Hash Functions (2)

- What do we do if our key is not a number?
  - *answer: map it to a number!*

  - *for example (from book):*

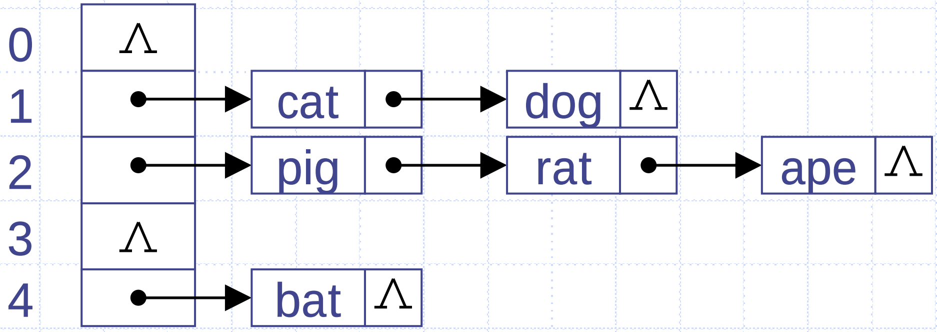
```
h ← 0                         // input is a string S of length s
for i ← 0 to s-1 do           // c_i is the char in i^th posn i of S
    h ← h + ord(c_i)          // ord(c_i) is the relative posn ...
                              // ... of c_i in the alphabet
code ← h mod numBuckets       // map sum of posns into range
```

  - *using the string "bob":*
    - ***h ← 2 + 15 + 2 = 19   // ord(b)=2, ord("o")=15***

  - *of course the actual hashcode depends on the number of buckets*
    - ***if numBuckets=10, the hashcode for "bob" is 9   (19 mod 10)***
    - ***ie: the string bob gets located in array element 9***

Collisions

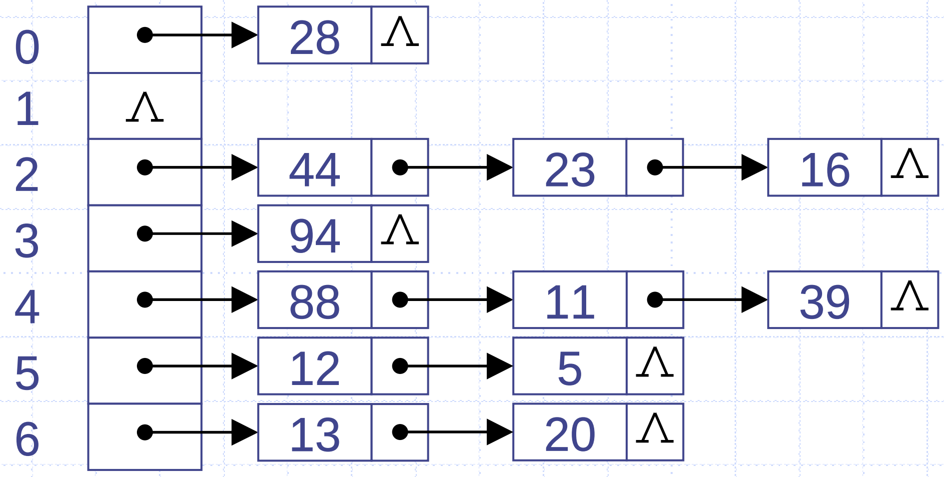
- It is possible that two keys could have the same index.
  - eg: numBuckets=25, key1=30, key2=105
    - index1 = 30 mod 25 = 5
    - index2 = 105 mod 25 = 5
- We may need to store more than one item in a single bucket
  - therefore each bucket must be a list or array or set or something
  - a typical implementation uses an array of lists …



Hashing Exercise 1

Draw the 7-element hash table resulting from hashing the keys:

- 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, 5, 28
- use the hash function **`h(i) = i mod 7`**
- Assume collisions are handled by chaining
  - ie: this means the table is an array of linked lists as shown on the preceding slide



Hashing Exercise 2

- Let's use hashing to implement a Map
- Consider the key:value pairs shown below
- Devise an algorithm (hash function) to map the keys to buckets
- Draw a 10-element hashmap resulting from hashing of the keys using your hash function

```
a8s:elvis
se3:weasil
22a:pepper
14c:chili
aba:pretzel
1s1:elvis
d6e:angus
```

Hashing Exercise 2 (solution part 1)

One possible algorithm is similar to the one discussed earlier for strings, but we don't take the ordinal value for integers (ie: the char "4" is just assigned the integer value 4)

For example: the string c7 is Ord("c") + 7 = 3+7 = 10

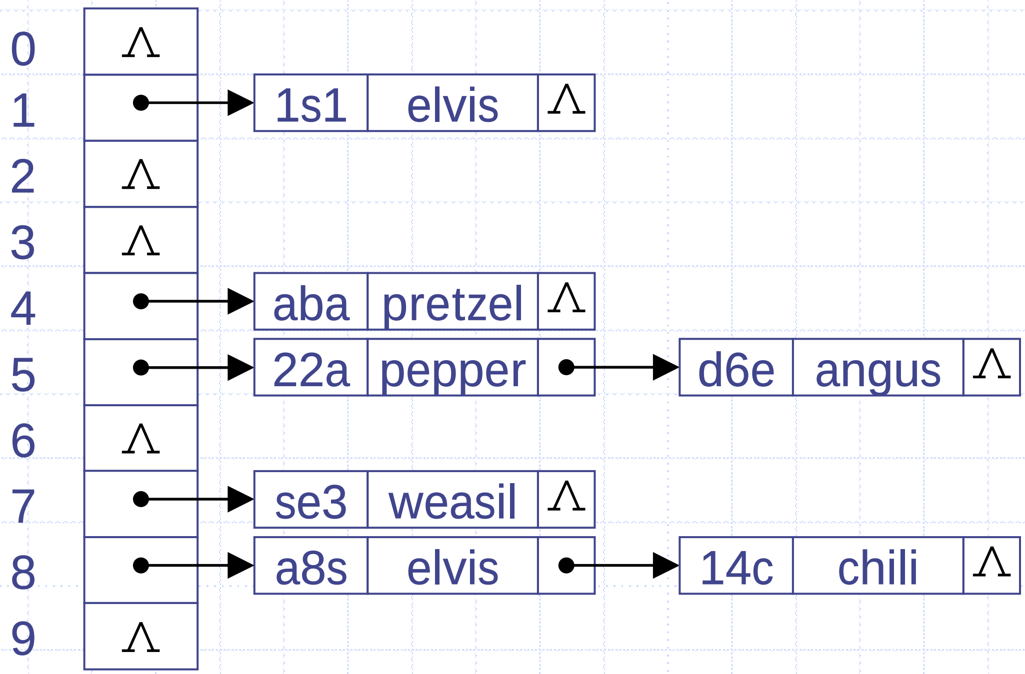
Using this algorithm we get:

KEY VALUE	ORD SUM	HASHCODE
a8s:elvis	1+8+19=28	28 mod 10 = 8
se3:weasil	19+5+3=27	27 mod 10 = 7
22a:pepper	2+2+1=5	5 mod 10 = 5
14c:chili	1+4+3=8	8 mod 10 = 8
aba:pretzel	1+2+1=4	4 mod 10 = 4
1s1:elvis	1+19+1=21	21 mod 10 = 1
d6e:angus	4+6+5=15	15 mod 10 = 5

a=1
b=2
c=3
d=4
e=5
f=6
g=7
h=8
i=9
j=10
k=11
l=12
m=13
n=14
o=15
p=16
q=17
r=18
s=19
t=20
u=21
v=22
w=23
x=24
y=25
z=26

Hashing Exercise 2 (solution part 2)

- now we draw the hashmap
  - we will need to store the keys as well as the values …



Efficiency of Hashing

What is the efficiency of the hashtable structure?

- assuming:
  - at least n buckets, where n > k keys
  - a uniform distribution of data to be hashed
  - a fast (constant time) hash function
- we get:
  - **add**(key, value) … is **O(1)**
  - value ← **get**(key) … is **O(1)**
  - **delete**(key) … is **O(1)**
- of course there could always be a degenerate case, where every insert causes a collision … in this case we would end up with O(n)

→ ***implementation of the hashing function is important***

→ ***it must distribute the keys evenly over the buckets***

The End