UNIT 6B
Organizing Data: Hash Tables

Announcements

- Online assignment due Wednesday 27th
- Lab Exam 1 Thursday 28th
  - Write simple programs during recitation

Last Lecture

- Arrays, lists, stacks, queues

This Lecture

- Hash tables

Comparing Algorithms

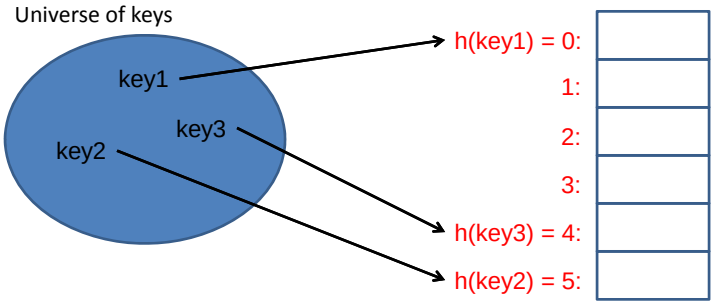
- You are a professor and you want to find an exam in a large pile of n exams.
- Search the pile using linear search.
  - Per student: O(n)
  - Total for n students: $O(n^2)$
- Have an assistant sort the exams first by last name.
  - Assistant's work: O(n log n) using merge sort
  - Professor:
    - Search for one student: O(log n) using binary search
    - Total for n students: O(n log n)

Another way

- Set up a large number of "buckets".
- Place each exam into a bucket based on some function.
  - Example: 26 buckets, each labeled with a letter. Use the first letter of student's andrew ID to choose the bucket.
- Ideally, if the exams get distributed evenly, there will be only a few exams per bucket.
  - Assistant: O(n) putting n exams into the buckets
  - Professor: O(1) search for an exam by going directly to the relevant bucket and searching through a few exams.

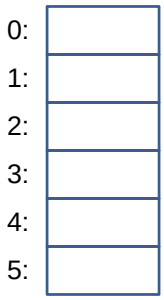
Hashing

- A "hash function" h(key) that maps a key to an array index in 0..*k*-1.
- To search the array Table for that key, look in Table[h(key)]

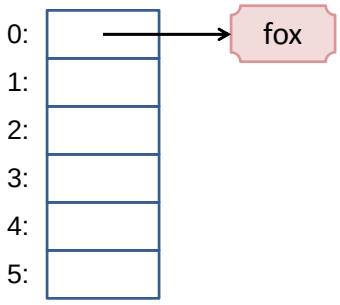


A hash function h is used to map keys to hash-table slots. In our example, keys were names and the hash function was getting the first letter of the name.

An Empty Hash Table



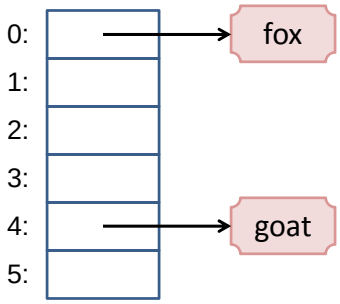
Add Element “fox”



h("fox") is 0

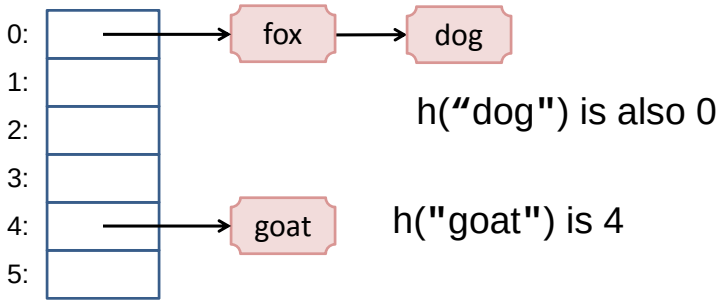
Suppose some function h gives these results. We did not specify what it is.

Add Element “goat”



h("goat") is 4

Add Element “hen”



Requirements for the Hash Function h(x)

- Must be fast: O(1)
- Must distribute items roughly uniformly throughout the array, so everything doesn’t end up in the same bucket.

Hash table

- Let's assume that we are going to store only lower case strings into an array (**hash table**).

```
table1 = Array.new(26)
=> [nil, nil, nil, nil, nil, nil, nil, nil,
  nil, nil, nil, nil, nil, nil, nil, nil,
  nil, nil, nil, nil, nil, nil, nil, nil,
  nil, nil]
```


Strings and ASCII codes

```
s = "hello"
for i in 0..s.length-1 do
  print s[i], "\n"
end
```

```
104
101
108
108
111
```

You can treat a string like an array in Ruby.
If you access the ith character, you get the ASCII code for that character.

Hash table

- We could pick the array position where each string is stored based on the first letter of the string using this hash function:

```
def h(string)
  return string[0] - 97
end
```

The ASCII values of lowercase letters are:

"a" -> 97, "b" -> 98, "c" -> 99, "d" -> 100, etc.

Inserting into Hash Table

- To insert into the hash table, we simply use the hash function h to determine which index ("bucket") to store the element.

```
def insert(table, name)
  table[h(name)] = name
end

insert(table1, "aardvark")
insert(table1, "beaver")  ...
```

Hash function (cont'd)

- Using the hash function h:
  - "aardvark" would be stored in an array at index 0
  - "beaver" would be stored in an array at index 1
  - "kangaroo" would be stored in an array at index 10
  - "whale" would be stored in an array at index 22

```
table1
=> ["aardvark", "beaver", nil, nil, nil,
  nil, nil, nil, nil, nil, "kangaroo", nil,
  nil, nil, nil, nil, nil, nil, nil, nil,
  nil, nil, "whale", nil, nil, nil]
```


Constant Time Search

```
def hash_search(table, item)
  return table[h(item)].include?(item)
end

>> hash_search(table1, "kangaroo")
⇒true

>> hash_search(table2, "armadillo")
⇒false
```

Hash function (cont'd)

```
>> insert(table1,"bunny")
>> insert(table1,"bear")
>> table1
⇒["aardvark", "bear", nil, nil, nil, nil,
  nil, nil, nil, nil, "kangaroo", nil, nil,
  nil, nil, nil, nil, nil, nil, nil, nil,
  nil, "whale", nil, nil, nil]
```

If we try to insert "bunny" and "bear" into the hash table, each word overwrites the previous word since they all hash to index 1.

Revised Ruby Implementation

```
>> table2 = Array.new(26)
>> for i in 0 .. 25 do
     table2[i] = []
   end
⇒[[], [], [], [], [], [], [], [], [], [], [],
  [], [], [], [], [], [], [], [], [], [], [],
  [], [], [], []]
```

Inserting into New Table

```
def insert(table, item)
  index = h(item)
  if not table[index].include?(item) then
    table[index] << item
  end
  return nil
end
```


Inserting into new hash table

```
insert(table2, "aardvark")
>> insert(table2, "beaver")
>> insert(table2, "kangaroo")
>> insert(table2, "whale")
>> insert(table2, "bunny")
>> insert(table2, "bear")
>> table2
=> [["aardvark"], ["beaver", "bunny",
  "bear"], [], [], [], [], [], [], [],
  ["kangaroo"], [], [], [], [], [], [], [],
  [], [], [], [], ["whale"], [], [], []]
```

Collisions

- “beaver”, “bunny” and “bear” all end up in the same bucket.
- These are collisions in a hash table.
- Why do we want to minimize collisions?

Collisions

- The more collisions you have in a bucket, the more you have to search in the bucket to find the desired element.
- We want to try to minimize the collisions by creating a hash function that distribute the keys (strings) into different buckets as evenly as possible.

A Poor Attempt

```
def h(string)
  k = 0
  for i in 0..string.length-1 do
    k = string[i] + k
  end
  return k
end
h("hello") => 532
h("olleh") => 532
```

Permutations still give same index (collision) and numbers are high.

What's A Good Hash Function?

- For strings:
  - Treat the characters in the string like digits in a base-256 number.
  - Divide this quantity by the number of buckets, *k*.
  - Take the remainder, which will be an integer in the range 0..*k*-1.

Hash Function For Strings

```
def h(s)
  sum = 0
  for i in 0..s.length-1 do
    sum = 256*sum + s[i]
  end
  return sum % 10
end
```

Number of buckets, could be anything

```
>> h("goat")
=> 2
```


Treating Characters As Numbers

```
>> "a"[0]
⇒97
>> "A"[0]
⇒65
>> s = "cat"
⇒"cat"
>> s[0]
⇒99
>> s[1]
⇒97
>> s[2]
⇒116
```

Base 10:
"573" is $5\times10^2 + 7\times10^1 + 3\times10^0 = 573$

Base 256:
"cat" is "c"$\times256^2$ + "a"$\times256^1$ + "t"$\times256^0$
$= 99\times256^2 + 97\times256^1 + 116\times256^0$
$= 6513012$

Final results

```
>> table3 = Array.new(10)
>> for i in 0 .. 9 do
     table2[i] = []
   end
⇒ [[], [], [], [], [], [], [], [], [], []]
>> insert(table3,"aardvark")
>> insert(table3,"bear")
>> insert(table3,"bunny")
>> insert(table3,"beaver")
>> insert(table3,"dog")
>> table3
=> [[“bear”], [“bunny”], [], [], [“beaver”], [], [], [],
   [],[“aardvark”, “dog”]]
```

Still have one collision, but b-words are distributed better.

Fancier Hash Functions

- How would you hash an integer i?
  - Perhaps i % k would work well.
- How would you hash a list?
  - Sum the hashes of the list elements.
- How would you hash a floating point number?
  - Maybe look at its binary representation and treat that as an integer?

Efficiency

- If the keys (strings) are distributed well throughout the table, then each bucket will only have a few keys and the search should take O(1) time.
- Example:
  If we have a table of size 1000 and we hash 4000 keys into the table and each bucket has approximately the same number of keys (approx. 4), then a search will only require us to look at approx. 4 keys => O(1)
  - But, the distribution of keys is dependent on the keys and the hash function we use!

Summary of Search Techniques

Technique	Setup Cost	Search Cost
Linear search	0, since we're given the list	O(n)
Binary search	O(n log n) to sort the list	O(log n)
Hash table	O(n) to fill the buckets	O(1)

Hash Tables in Ruby

- So far, we looked at hash tables as a means of determining whether a key is in a list in O(1) time.
- We can generalize this idea to associate a key with a value.
- Examples:
  - Employee name => Employee number
  - Product code => Price
  - Name in contacts list => Email address

Hashes (Associate Arrays) in Ruby

```
>> h
{"Mercedes" => 50000,
 "Bentley" => 120000}

>> h["Mercedes"]
⇒50000
```

Hash in Ruby (continued)

```
>> h2 = {:apple => :red,
         :banana => :yellow,
         :cherry => :red}
>> h2[:banana]
⇒:yellow
>> h2.invert
⇒{:red => :cherry,
   :yellow => :banana}
```


Next Week

- Monday: Finish data structures unit with trees and graphs
- Wednesday and Friday: New unit in data representation