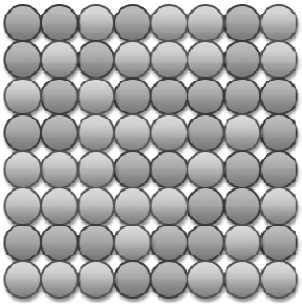

UNIT 6B
Organizing Data: Hash Tables

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: 100 buckets, each labeled with a value from 00 to 99. Use the student’s last two digits of their student ID number 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.

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

- 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]
```


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]
```

Hash function (cont'd)

- But 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:

```
>> 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]
```


Revised Hash table

- Let's make our hash table an array of arrays (an array of "buckets")
- Each bucket can hold more than one string.

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

Revised insert function

```
def insert(table, key)
  # find the bucket (array) in the table
  # array using the hash function h
  bucket = table[h(key)]
  # append the key string to the bucket
  # array
  bucket << key
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.
- 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.

First Try

```
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.

Second Try

```
def h(string)
  k = 0
  for i in 0..string.length-1 do
    k = string[i] + k*256
  end
  return k
end
h("hello") => 448378203247
h("olleh") => 478560413032
```

Better, but numbers are still high. We probably don't want to (or can't) create arrays that have indices this large.

Third Try

```
def h(string, tablesize)
  k = 0
  for i in 0..string.length-1 do
    k = string[i] + k*256
  end
  return k % tablesize
end
```

We can use the modulo operator to take the large values and map them to indices for a smaller array.

Revised insert function

```
def insert(table, key)
  # find the bucket (array) in the table
  # array using the hash function h
  bucket = table[h(key, table.length)]
  # append the key string to the bucket
  # array
  bucket << key
end
```


Final results

```
table3 = Array.new(13)
for i in 0..12 do table3[i] = [] end
=> [[], [], [], [], [], [], [], [], [], [], [], [],
   []]
>> insert(table3,"aardvark")
>> insert(table3,"bear")
>> insert(table3,"bunny")
>> insert(table3,"beaver")
>> insert(table3,"dog")
>> table3
=> [[], [], [], [], [], [], [], [], [], ["bunny"],
   ["aardvark", "bear"], ["dog"], ["beaver"]]
```

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

Searching in a hash table

To search for a key, use the hash function to find out which bucket it should be in, if it is in the table at all.

```
def contains?(table, key)
  bucket = table[h(key,table.length)]
  for entry in bucket do
    return true if entry == key
  end
  return false
end
```


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!