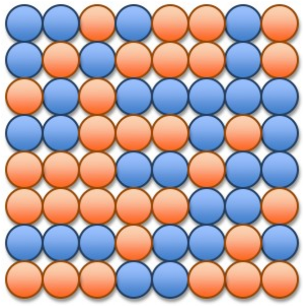


UNIT 6B
Organizing Data: Hash Tables

Quiz

Given: **`x = ["b", "c", "d"]`**

Write a short Ruby expression in terms of x to produce each of the following results:

```
["a", "b", "c", "d"]
["b", "c", "d", "e"]
[["b", "c", "d"]]
["a", ["b", "c", "d"]]
[["a","b"], ["a", "c"], ["a", "d"]]
```

Quiz

Analyze this recursive function:

```
def sum_elements (list)
  if list = [] then
    return 0
  else
    return list[0] +
      sum_elements(list[1..list.length-1])
  end
end
```

What is the base case? What is the base result?

What is the recursive case? How do you derive the result of this case from the result of the easier recursive call?

What Do We Know About Search?

- You can search an unordered list in O(*n*) time by "brute force" (simple linear search).
- You can search an ordered list in O(log *n*) time using binary search.
  - For large *n*, this is <u>much</u> faster than linear search.
  - But... sorting the list takes O(*n* log *n*) time.
  - Worth it if you're going to do lots of searches on the same list, since you only have to sort once.

Really Fast Searching

- Sorting has been proved to require O(n log n) time. There cannot be a faster algorithm.
  Life is hard.

- But you can *search* a list in O(1) time!

- How? Use a hash table.

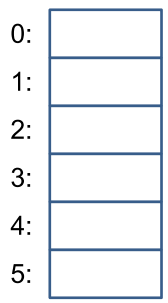
Requirements for Constant Time Search

- An hash table (array) T with *k* elements, called "buckets". The exact value of *k* doesn't matter but it must be of size comparable to n.
  In other words, *k* is of size O(*n*).

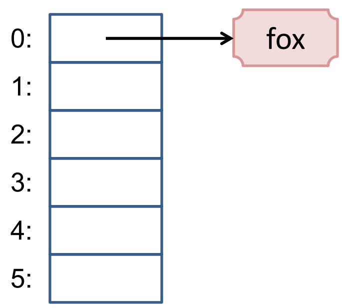
- A "hash function" h(*x*) that maps an item *x* to an array index in 0..*k*-1.

- To search the array T for item x, look in T[h(x)]

An Empty Hash Table

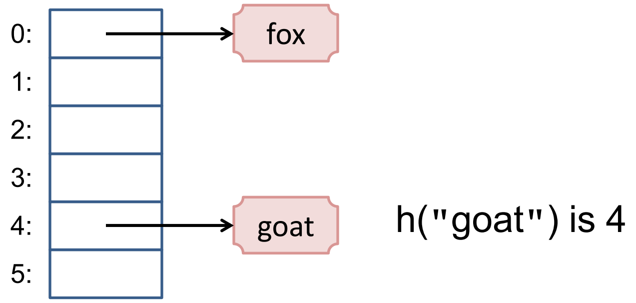


Add Element “fox”

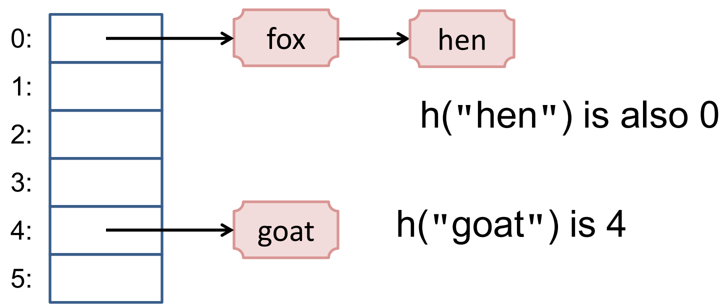


h("fox") is 0

Add Element “goat”



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.

Ruby Implementation

```
>> T = Array.new(6, [])
=> [[], [], [], [], [], []]

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

Testing Our Hash Table

```
>> hash_insert(T, "fox")
>> hash_insert(T, "goat")
>> hash_insert(T, "hen")

>> T
[["fox", "hen"], [], [], [], ["goat"], []]
```


Constant Time Search

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

>> hash_search(T, "fox")
⇒true

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

Review

- Why can the search be done in constant time?
  – Because the hash function is O(1), and …
  – A bucket contains only a few items.

- Why do buckets contain only a few items?
  – Because we have O(n) buckets, and …
  – Our hash function distributes items roughly uniformly throughout the array, so there are few collisions.

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.

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"$\times256^0$
$= 99\times256^2 + 97\times256^1 + 116\times256^0$
$= 6513012$

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 % 6
end
```

Number of buckets

```
>> h("goat")
=> 4
```

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?

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)