cphbusiness

COPENHAGEN BUSINESS ACADEMY





Collections & efficiency

Topics / plan

- Measuring efficiency
- Classic algorithms
- Impact of implementation
- Hashing

- Selecting a suitable algorithm and/or collection

Efficiency of algorithms

- Comparison
  - Memory requirements
  - Execution time
    - absolute
    - relative (”time complexity”, ”growth rate”)
- Ex
  - Linear search, Binary Search
    - worst, best, average
- Big-O notation

```java
public static Comparable linearSearch (Comparable[] list,
                                      Comparable target)
  {
     int index = 0;
     boolean found = false;

     while (!found && index < list.length)
     {
        if (list[index].equals(target))
           found = true;
        else
           index++;
     }

     if (found)
        return list[index];
     else
        return null;
  }
```

```
public static Comparable binarySearch (Comparable[] list,
                                        Comparable target)
  {
    int min=0, max=list.length, mid=0;
    boolean found = false;

    while (!found && min <= max)
    {
      mid = (min+max) / 2;
      if (list[mid].equals(target))
        found = true;
      else
        if (target.compareTo(list[mid]) < 0)
          max = mid-1;
        else
          min = mid+1;
    }

    if (found)
      return list[mid];
    else
      return null;
  }
```

cphbusiness

Logarithms

How many times can we **half** N before we only have 1

- Log_2 - logarithm function with base 2
  - The inverse function to the exponential function with base 2:

    $f(x) = 2^x$
- Log_2
  - How does it look - graphically?

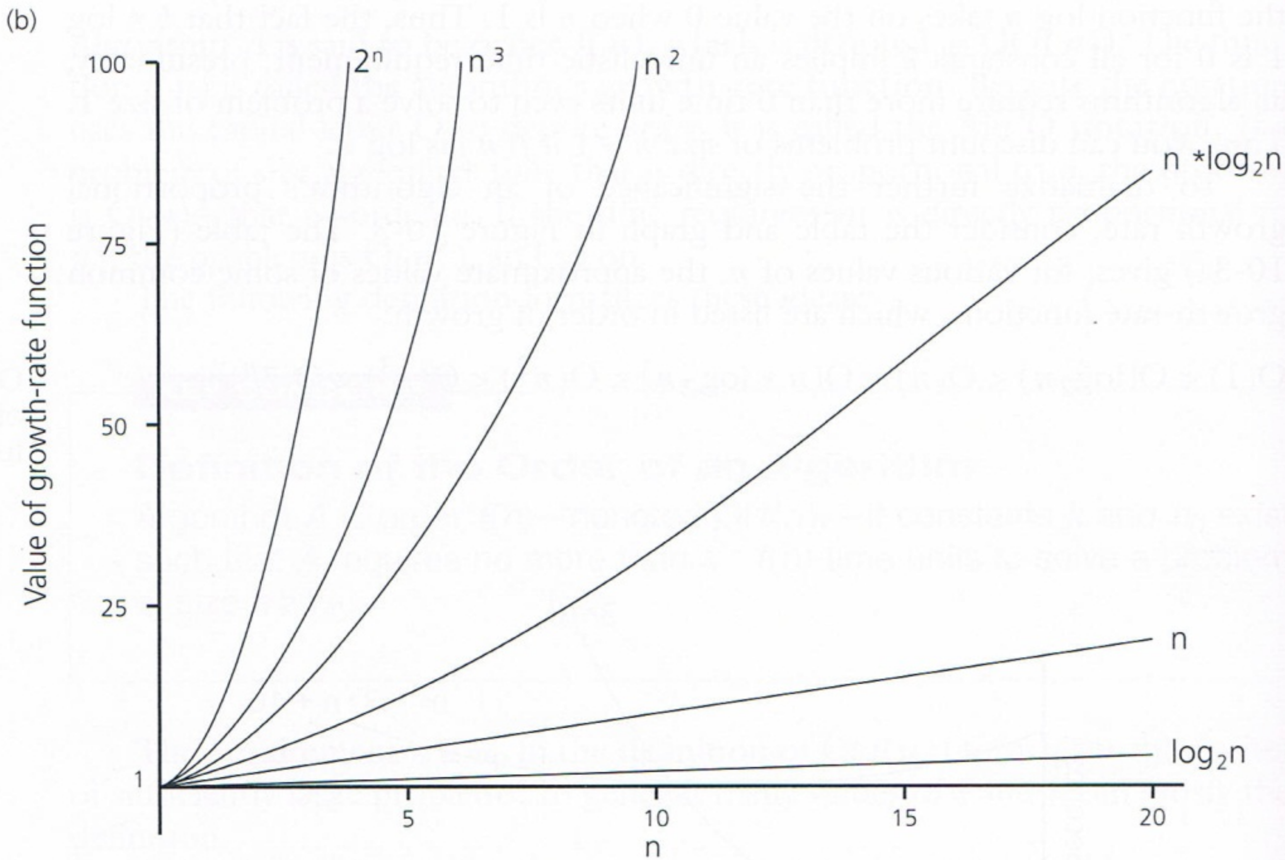


FIGURE 10-3

A comparison of growth-rate functions: (a) in tabular form; (b) in graphical form

3. The graph of $f(n) = 1$ is omitted because the scale of the figure makes it difficult to draw. It would, however, be a straight line parallel to the x axis through $y = 1$.

The table demonstrates the relative speed at which the values of the functions grow. (Figure 10-3b represents the growth-rate functions graphically.[3])

(a)

Function	10	100	1,000	10,000	100,000	1,000,000
1	1	1	1	1	1	1
$\log_2 n$	3	6	9	13	16	19
n	10	10^2	10^3	10^4	10^5	10^6
$n * \log_2 n$	30	664	9,965	10^5	10^6	10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{301}	$10^{3,010}$	$10^{30,103}$	$10^{301,030}$

cphbusiness

```java
public static void selectionSort (Comparable[] list)
  {
     int min;
     Comparable temp;

     for (int index = 0; index < list.length-1; index++)
     {
        min = index;
        for (int scan = index+1; scan < list.length; scan++)
           if (list[scan].compareTo(list[min]) < 0)
              min = scan;

        // Swap the values
        temp        = list[min];
        list[min]   = list[index];
        list[index] = temp;
     }
  }
```

Classic algorithms for manipulating a list

- Linear search: $O(n)$
- Binary search: $O(\log n)$

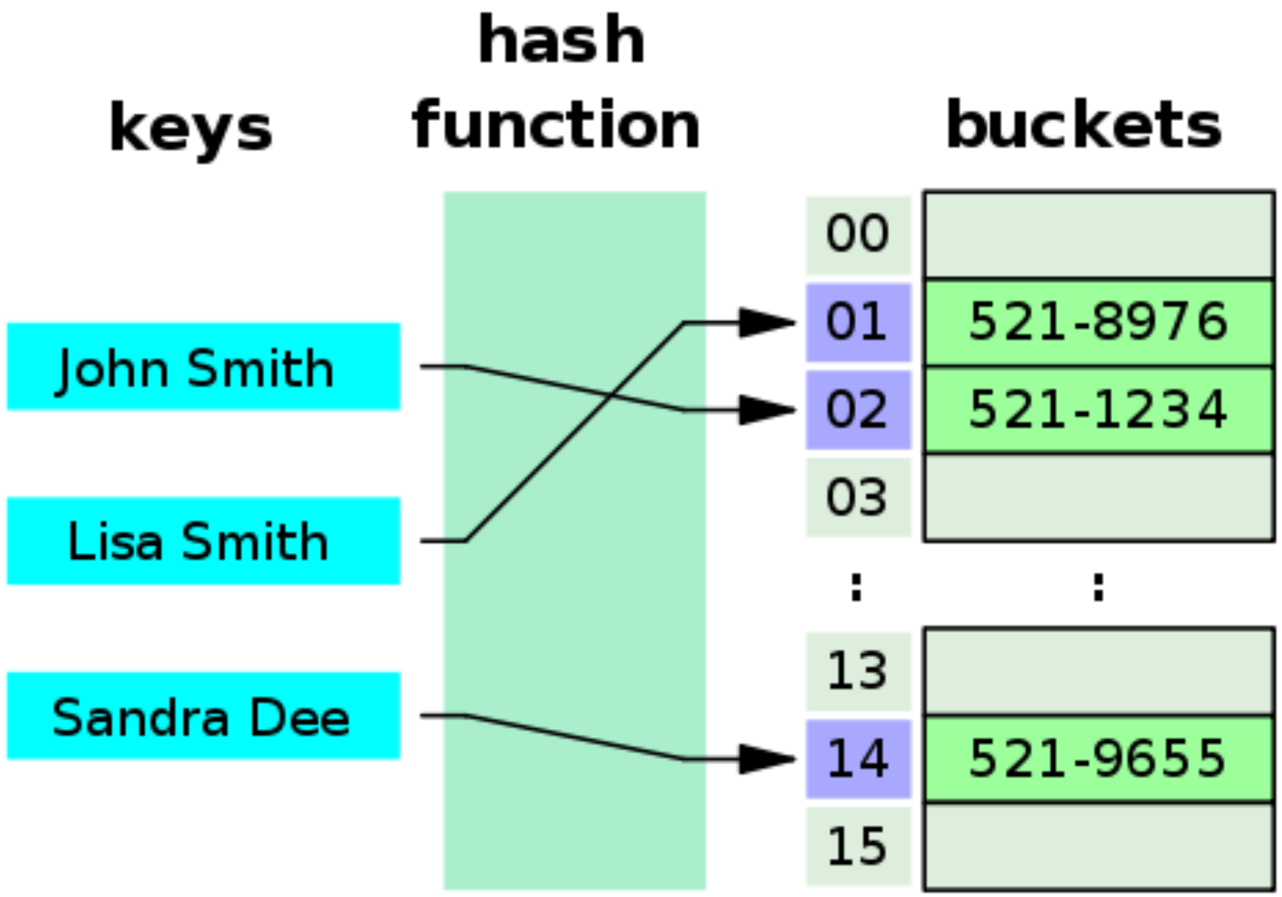
- Selection Sort: $O(n^2)$ (same for Insertion and Bubble Sort)
- Quick Sort: $O(n*\log n)$ (average)
  $O(n^2)$ (worst)

Hashing – Why another data structure?

	insert	**search**
Unsorted array	O (1)	**O (n)**
Unsorted linked list	O (1)	**O (n)**
Sorted array	O (n) *	**O (log n)**
Sorted linked list	O (n)	**O (n)**
Binary search tree	O(log n)	**O (log n)**

*) O(log n) + O (n)

Hash table


keys
hash function
buckets
John Smith
Lisa Smith
Sandra Dee
00
01
02
03
13
14
15
521-8976
521-1234
521-9655

Hashing – principle

- Data is stored in an **array**
- The index **is calculated** based on a key.
  - **index = hash-function (key)**
- Insert
  - put(key, value)
- ”Search”
  - get (key)
- The **hash-function** must
  - return an integer (index < size of table)
  - be easy to calculate (why?)
  - Minimize the number of collisions
  - distribute data elements evenly across the table (why?)

Hash function (ex):

key value modulo 11

(11 = size of table)

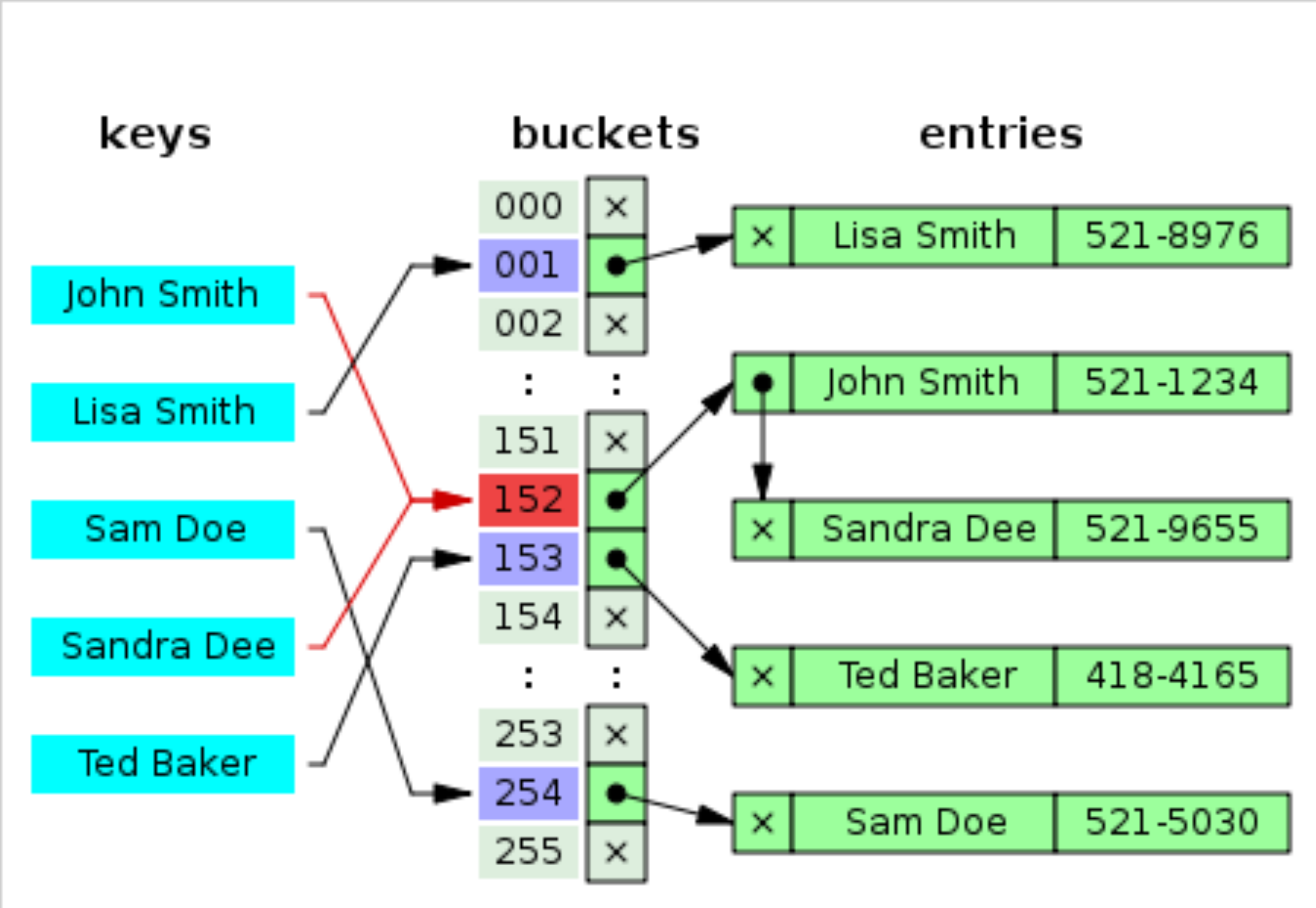
ex:

Key value: 13

hash (13) => 13 mod 11 => 2

0	11
1	1
2	13
3	
4	
5	
6	
7	
8	30
9	
10	10

Collisions, chaining



Hash collision resolved by separate chaining.

Efficiency and hash table

- Insert, delete and search is (nearly) independent of the number of elements (n)
  - O(1)
  - Load factor

- Table size << number of possible different key values

- Preferred when you require fast
  - search
  - Insert

  but not fast
  - Iterate sorted
  - Location of max /min

Choice of data structure (array/linked/hash table)?
Criterion: Frequency of operations

	1	2	3	4
Insert	rarely	often	often	often
Delete	rarely	often		
Search	often		often	
Iterate unsorted		often		
Iterate sorted	ofte			often