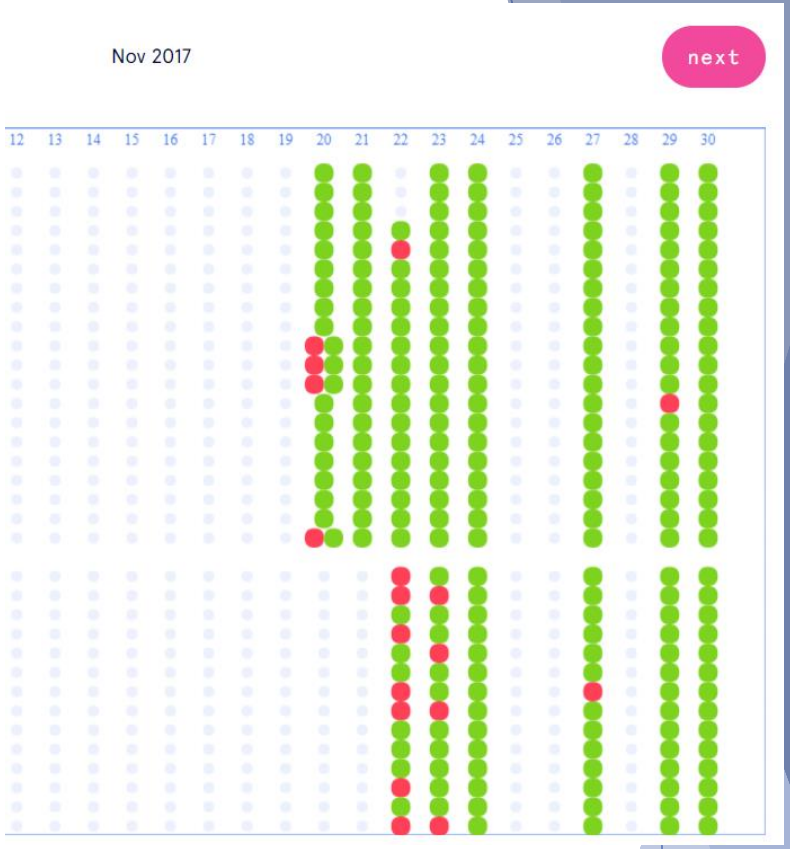
INFDEV036A - Algorithms Lesson Unit 3b

G. Costantini, F. Di Giacomo

costg@hr.nl, giacf@hr.nl – Office H4.206

Homework progress



- Have you completed all MC questions in GrandeOmega?

- Have you implemented all algorithms+data structures seen in the previous lessons?

Recap about exam

- We start with the written exam
  - **MC questions** *in Grande Omega*
    - Most likely, approx 20 questions with 6 answers each
  - After delivery you see how many questions were wrong

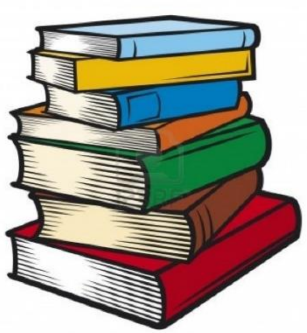
- Immediately followed by the practical assessment
  - **C# code** *in Visual Studio*
  - A few algorithms with some lines of code missing, to be completed by you
  - Not allowed to change the logic of the code (you have to fill in <u>specific lines</u>)
  - Grade is visible in real-time

Today

LIST

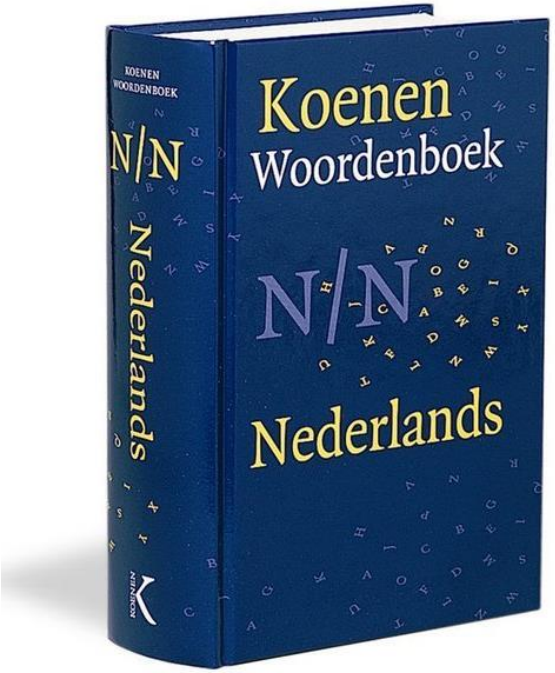
QUEUE

- ~~Why is my code slow?~~
  - ~~**Empirical and complexity analysis**~~
- ~~How do I order my data?~~
  - ~~**Sorting algorithms**~~
- How do I structure my data?
  - ~~**Linear**~~**, tabular, recursive data structures**
- How do I represent relationship networks?
  - **Graphs**



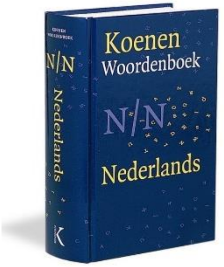
STACK

HASH TABLE


KOENEN WOORDENBOEK
N/N
Nederlands
Koenen
Woordenboek
N/N
Nederlands

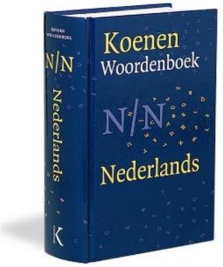
Hash table

Hash table - Definitions



- Hash table → data structure used to implement an *associative array*, a structure that can map keys to values
- Hash table → container that allows direct access by any index type: it works like an array except that the index variable does not need to be an integer
  - Also called: hash map, lookup table, associative array, dictionary
  - Analogy with the dictionary: index = word to look up; value indexed = dictionary definition

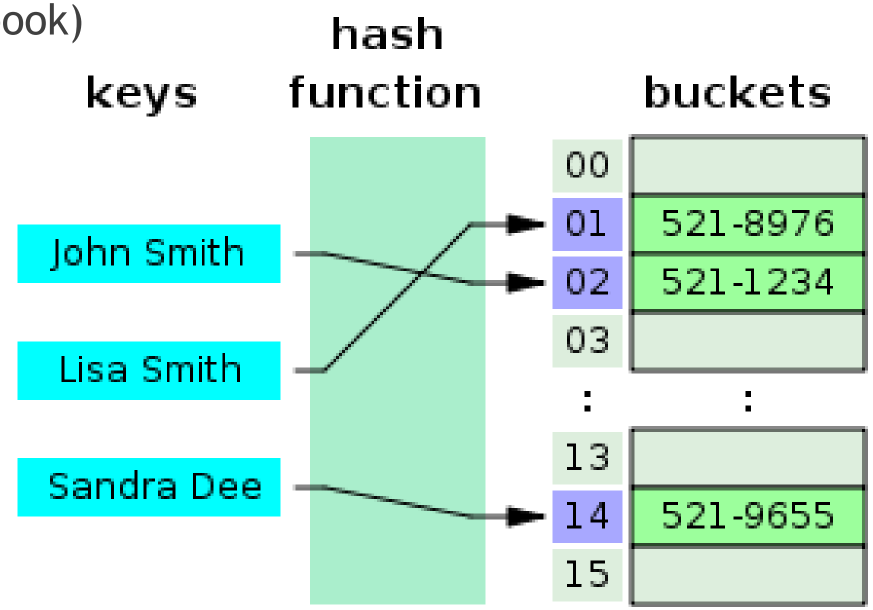
Hash table - Definitions



- Hash table → data structure used to implement an *associative array*, a structure that can map keys to values
- Hash table → container that allows direct access by any index type: it works like an array except that the index variable does not need to be an integer
  - Also called: hash map, lookup table, associative array, dictionary
  - Analogy with the dictionary: index = word to look up; value indexed = dictionary definition
- Entries of a hash table are called "**key-value**" pairs
  - *Key* → index into the table
  - *Value* → information being looked up

Hash table - Definition

- Hashing idea → distribute the entries (key/value pairs) across an array of *buckets* (also called *slots*)
- A **hash function** is used to compute the index in the buckets array, from which the correct value can be found
  - Example (phone book)



Hash table - Definition

- Given a **key**, the algorithm computes an **index** that suggests where the entry can be found:

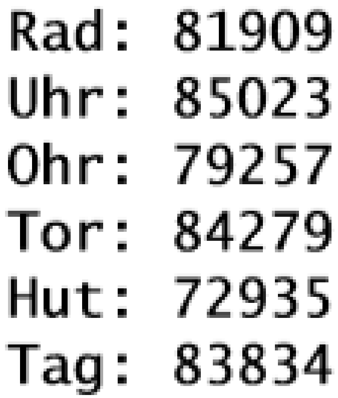
```
hash = hashfunc(key)
index = hash % array_size
```

- The hash is independent of the array size
  - it is then reduced to an index (a number between 0 and $array\_size - 1$) using the modulo operator (%)

Hash table - Definition

- In Java and .NET, every object is associated to a hash code (computed from the actual hard data stored in the object), accessible through the methods:
  - [Java] Object.hashCode()
  - [.NET] Object.GetHashCode()

- Example: hash codes of some strings made by three characters



```
Rad:  81909
Uhr:  85023
Ohr:  79257
Tor:  84279
Hut:  72935
Tag:  83834
```

```
hash = hashfunc(key)
```

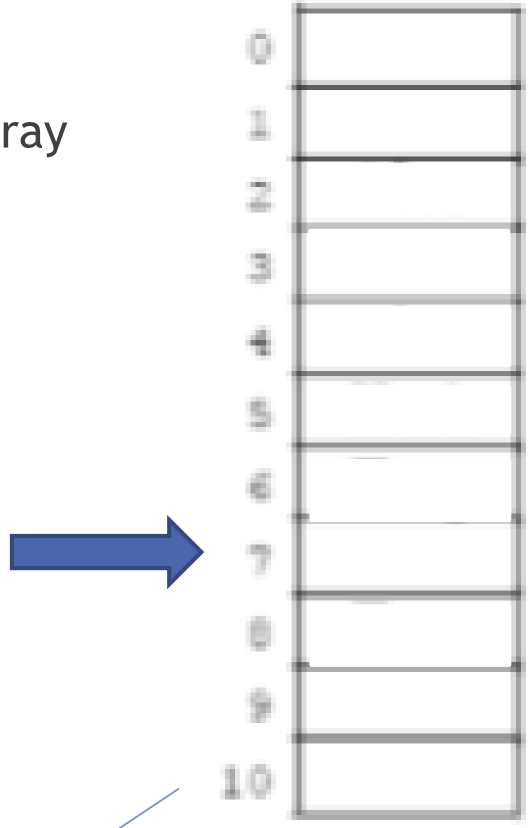
Hash table - Definition

- After computing the hash code, we must compute the index inside the array
  - Suppose that array_size = 11

```
index = hash % array_size
```

  - Index of Rad: 81901 % 11 = 3
  - Index of Uhr: 85023 % 11 = 4
  - Index of Ohr: 79257 % 11 = 2
  - Index of Tor: 84279 % 11 = 8
  - Index of Hut: 72935 % 11 = 5
  - Index of Tag: 83834 % 11 = 3 … same index as for the first string!!!

Rad:	81909
Uhr:	85023
Ohr:	79257
Tor:	84279
Hut:	72935
Tag:	83834



Hash table – Hash function

- You can also implement your *own hash function*
  - A good hash function and implementation algorithm are **essential** for good hash table performance, but may be difficult to achieve.
  - If all keys are known ahead of time, a *perfect hash function* can be used to create a perfect hash table that has no collisions.

- Basic requirement → the function should provide a *uniform* distribution of hash values (to avoid collisions as much as possible)
  - The hash function should also avoid *clustering* (= the mapping of two or more keys to consecutive slots) if the open addressing method is used to resolve collisions

Hash table – Load factor

- Load factor is a critical statistics for a hash table
  - Good performance depends a lot on it

$$\boldsymbol{loadFactor} = \frac{\#entries}{\#buckets}$$

  - *Entries* = actual number of elements inside the table
  - *Buckets* = capacity of the table (number of total available slots)
    - Example: 6 elements stored in a table with 101 slots → load factor = $\frac{6}{101} = 0.0594 \Rightarrow 5.9\%$

- If the load factor is too large, the hash table becomes slow
  - Possible way to solve the problem: resize the table when the load factor reaches a threshold (usually 75%)

Hash table - Collision resolution

- **Collision** → different keys are assigned by the hash function to the same bucket
    - Ideally, the hash function will assign each key to a unique bucket, but this situation is rarely achievable in practice → collisions are practically unavoidable when hashing a random subset of a large set of possible keys

- Most hash table implementations have some collision resolution strategy to handle such events (all requiring to store the key together with the value inside the table):
    - Separate chaining
    - Open addressing (linear probing, quadratic probing)
    - …

Hash table - Collision resolution with ***open addressing***

- Open addressing → when a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some probe sequence, until an unoccupied slot is found
  - The location ("address") of the item is not determined by its hash value (that's why is called *open addressing*)

- Probing sequences
  - Linear probing
  - Quadratic probing
  - …

Hash table - Collision resolution with open addressing and *linear probing*

- **Linear probing** → when a new item hashes to a table component that is already in use, the algorithm specifies to *increment the index* until an empty component is found

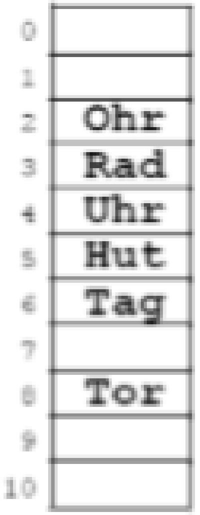
- Given the hash code H, the probing sequence is

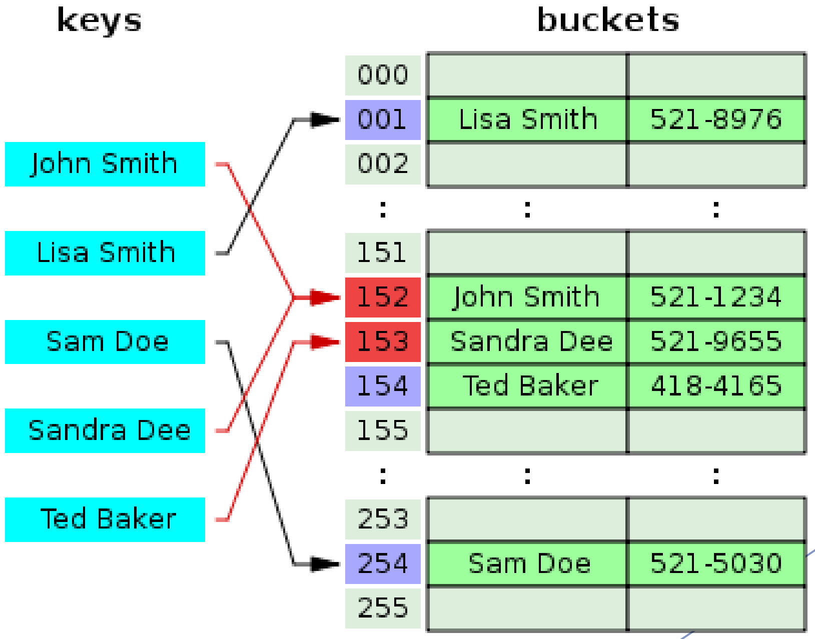
$$H + 1 \rightarrow H + 2 \rightarrow H + 3 \rightarrow H + 4 \rightarrow \cdots$$

  - NB: this may require a "wraparound" back to the beginning of the hash table

Hash table - Collision resolution with open addressing and *linear probing*

- Linear probing examples
  - Tag & Rad from a few slides earlier
  - Sandra Dee; Ted Baker in the phonebook





Hash table - Collision resolution with open addressing and *quadratic probing*

- **Quadratic probing** → taking the original hash index and adding successive values of an arbitrary *quadratic polynomial* until an open slot is found
  - Instead of searching linearly, it uses a squared increment
  - NB: this also may require a “wraparound” back to the beginning of the hash table

- Given the hash code H, a possible quadratic probing sequence is:

$$H + 1^2 \rightarrow H + 2^2 \rightarrow H + 3^2 \rightarrow H + 4^2 \rightarrow \cdots$$

- Improved performance with respect to linear probing, but it is also more likely to result in an infinite loop…

Hash table - Collision resolution with open addressing

- Open addressing methods drawbacks 👎
  1. the number of stored entries cannot exceed the number of slots in the bucket array
     - performance dramatically degrades when the load factor grows beyond 0.7 → dynamic resizing is mandatory
  2. more stringent requirements on the hash function
     - besides distributing the keys more uniformly over the buckets, the function must also minimize the clustering of hash values that are consecutive in the probe order

Hash table - Collision resolution with *separate chaining*

- Instead of *resolving* collisions, we can **avoid** them… how?
  - Allowing more than one item per bucket!
  - Method called “separate chaining” because it uses linked lists (“chains”) to hold the multiple items

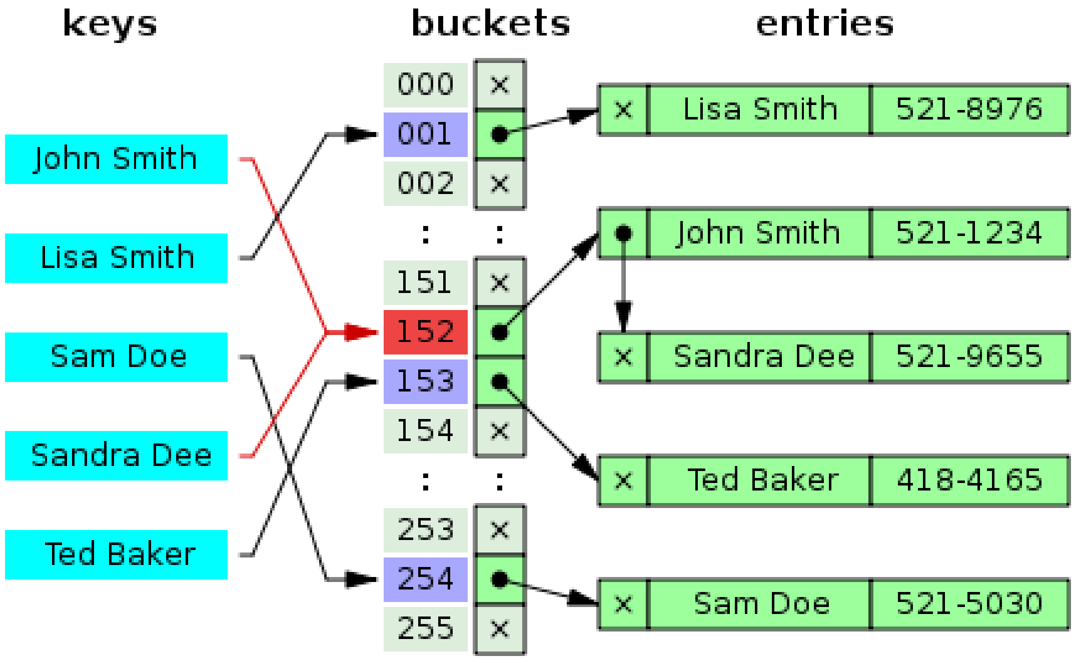
Hash table - Collision resolution with ***separate chaining***

- Instead of *resolving* collisions, we can **avoid** them… how?
  - Allowing more than one item per bucket!
  - Method called “separate chaining” because it uses linked lists (“chains”) to hold the multiple items

- In a good hash table, each bucket has zero or one entries, and sometimes two or three, but rarely more than that
  - Otherwise performance in hash table operations decreases because we have to add the time for the list operation

Hash table - Collision resolution with ***separate chaining***

- Example

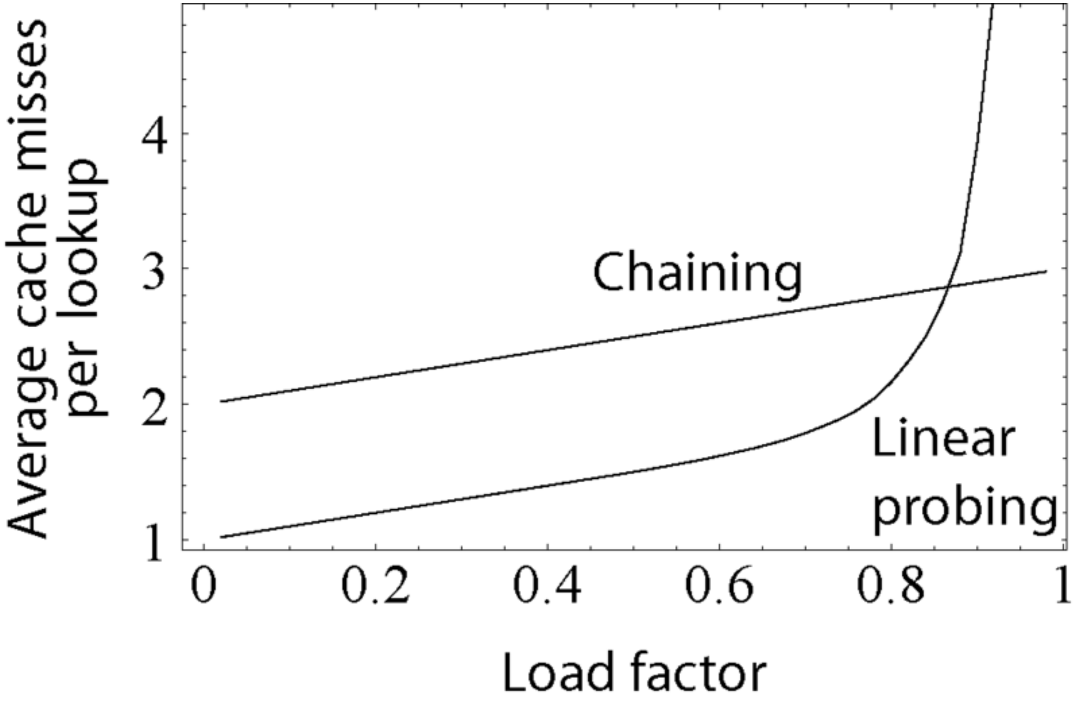


Hash table - Collision resolution with separate chaining

- Which data structure should we use to store the multiple items in each bucket?
  - **Linked lists**
    - Popular because it requires only basic data structures with simple algorithms
    - When storing small keys and values, the space overhead of the next pointer in each entry record can be significant
    - Traversing a linked list has poor cache performance, making the processor cache ineffective
  - Ordered lists, sorted by key field
  - Self-balancing search trees
    - Only worth the trouble and extra memory cost if long delays must be avoided at all costs (e.g. in a real-time application) or if one must guard against many entries hashed to the same slot (e.g. if one expects extremely non-uniform distributions, or in the case of web sites or other publicly accessible services, which are vulnerable to malicious key distributions in requests)
  - Dynamic arrays

Hash table - Collision resolution

- Comparison between the “performance” (seen as the average number of cache misses required to look up elements in tables) with separate chaining and linear probing
  - Linear probing's performance drastically degrades for load factors > 0.8



Hash table – Dynamic resizing

- A hash table functions well when the table size is proportional to the number of entries
- Practical problem: usually the number of entries is not known in advance
  - Very important to provide some method to resize the table in order to prevent the hash table from becoming too full
  - Resizing happens only when the load factor becomes too large
    - In Java the default load factor threshold for table expansion is 0.75; in Python's *dict* 2/3
  - Resizing is accompanied by a *full* or *incremental* table **rehash** whereby existing items are mapped to new bucket locations

Hash table – Dynamic resizing

- *Resizing by copying all entries*
  - Common approach → automatically trigger a complete resizing when the load factor exceeds some threshold
  - All the entries of the old table are removed and inserted into the new table
- *Incremental resizing*
  - Some hash table implementations (especially real-time systems), cannot pay the price of enlarging the hash table all at once: it may interrupt time-critical operations
  - Keep both the old and the new table; do lookups and deletions in both tables; new insertions only in the new one; at each insertion move some elements from the old to the new table until they are all removed (and then deallocate the old table)

Hash table – Performance analysis

- Average case
  - In a well-dimensioned hash table, the average cost (number of instructions) for each lookup is independent of the number of elements stored in the table
  - If the load factor is kept below some bound, the access functions are immediate, running in constant time → direct access, just like an array
- Worst case
  - Worst choice of hash function → every insertion causes a collision → hash tables degenerate to linear search

Operation	Average case	Worst case
Search	$O(1)$	$O(n)$
Insertion	$O(1)$	$O(n)$
Deletion	$O(1)$	$O(n)$

Hash table – Pros & Cons

- Main advantage
  - **Speed** → particularly efficient when the maximum number of entries can be predicted in advance (no resize)
- Disadvantages
  - The cost of a good hash function can be significantly higher than the inner loop of the lookup algorithm for a sequential list or search tree
    - hash tables not effective when the number of entries is very small
  - Entries can be enumerated only in pseudo-random order
    - no efficient way to locate an entry whose key is *nearest* to a given key → separate sorting step needed
  - With dynamic resizing, an insertion or deletion operation may occasionally take time proportional to the number of entries → problem in real-time or interactive applications
  - Quite inefficient when there are many collisions

Hash table – Applications

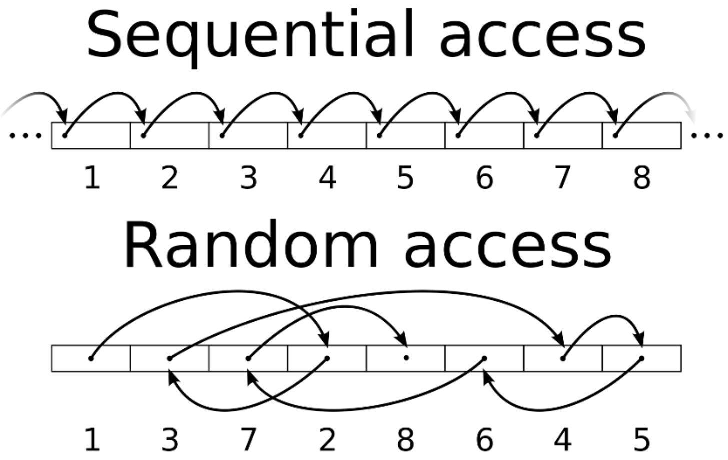
- In many situations, hash tables turn out to be more efficient than search trees or any other table lookup structure → widely used in many kinds of computer software
  - systems programming
  - primary building blocks of relational databases
  - associative arrays
  - caches
  - sets
  - …

Hash tables in C#

- **Dictionary** class
  - Generic with respect to the types of keys and values
  - http://www.dotnetperls.com/dictionary
  - http://msdn.microsoft.com/en-us/library/xfhwa508%28v=vs.110%29.aspx

- Live demo?

Summary



- **Array** and **Hash table**
  - *random access* data structures → each element can be accessed directly and in constant time
- **Linked list**
  - *sequential access* data structure → each element can be accessed only in a particular order
- **Stack & Queue**
  - *limited access* data structures (subcase of sequential data structures)

Homework

- Study the slides
- Answer the MC questions on GrandeOmega
- Implement $HashTable < K,V >$
  - with linear probing, and resizing by copying all entries

See you next week ☺