Hashing (Κατακερματισμός)

The Symbol Table ADT (Reminder)

- A **symbol table** T is an abstract storage that contains table entries that are either empty or are pairs of the form (K, I) where K is a **key** and I is some information associated with the key which we call **value**.
- We also make the assumption that **distinct table entries have distinct keys,** so the symbol tables to be studied in this lecture are **maps**.

Operations for the Symbol Table ADT

- **Initialize** the table T to be the **empty table**. The empty table is filled with **empty table entries** (K_0, I_0) where K_0 is a special **empty key**, distinct from all other nonempty keys.
- Determine whether or not the table T is **full**.
- **Insert** a new table entry (K, I) into the table provided that T is not already full.
- **Delete** the table entry (K, I) from table T.
- Given a search key K, **retrieve** the information I from the table entry (K, I) in table T.
- **Update** the table entry (K, I) in table T by replacing it with a new table entry (K, I').
- **Enumerate** the table entries (K, I) in table T in increasing order of their keys.

Possible Representations for the Symbol Table ADT

- We have already discussed the following data structures for symbol tables:
  - Arrays of structs sorted in ascending order of their keys
  - Linked lists of structs
  - Binary search trees
  - AVL trees
  - (2,4) trees
  - Red-black trees
  - Skip lists

Hashing

- We will introduce a new method to implement a symbol table called **hashing**.

- Hashing differs from the representations based on searching by key comparisons because we are trying to **refer directly to elements of the table by transforming keys into addresses in the table.**

Introducing Hashing by Example

- We will use as keys **letters of the alphabet** having as subscripts their order in the alphabet. For example, A_1, C_3, R_{18}.
- We will use a small table T of 7 positions as storage. We call it **hash table (πίνακας κατακερματισμού)**.
- We will find the location to store a key L_n by using the following **hash function (συνάρτηση κατακερματισμού)**:

$$h(L_n) = n \,\%\, 7$$

- $x \,\%\, y$ computes the **remainder** of the integer division of x with y.

An Empty Hash Table

	Table T
0	
1	
2	
3	
4	
5	
6	

Table T after Inserting keys $B_2, J_{10}, S_{19}, N_{14}$

	Table T
0	N_{14}
1	
2	B_2
3	J_{10}
4	
5	S_{19}
6	

Keys are stored in their **hash addresses.** The cells of the table are often called **buckets (κάδοι).**

Insert X_{24}

	Table T
0	N_{14}
1	
2	B_2
3	J_{10}
4	
5	S_{19}
6	

$h(X_{24}) = 3$

Now we have a **collision (σύγκρουση)**. We will use the **collision resolution policy (πολιτική επίλυσης συγκρούσεων)** of looking at lower locations of the table to find a place for the key.

Insert X_{24}

	Table T
0	N_{14}
1	X_{24}
2	B_2
3	J_{10}
4	
5	S_{19}
6	

3rd probe

2nd probe

$h(X_{24}) = 3$

1st **probe**

Insert W_{23}

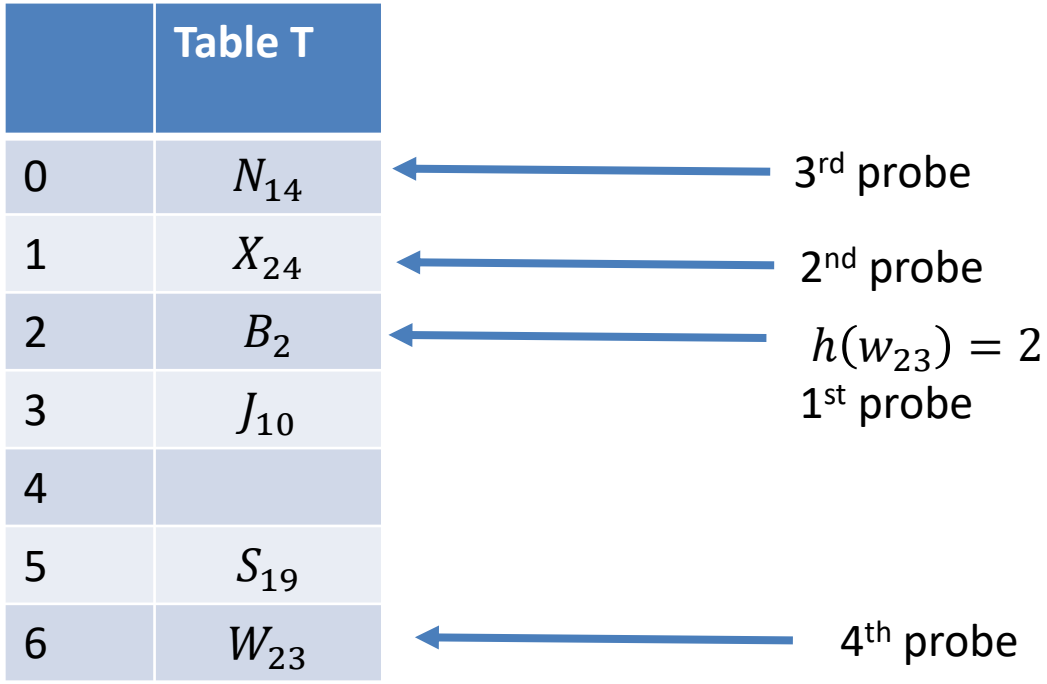


Table T
0
N_{14}
1
X_{24}
2
B_2
3
J_{10}
4
5
S_{19}
6
W_{23}
3rd probe
2nd probe
$h(w_{23}) = 2$
1st probe
4th probe

Open Addressing

- The method of inserting colliding keys into empty locations of the table is called **open addressing (ανοικτή διευθυνσιοδότηση).**
- The inspection of each location is called a **probe (διερεύνηση)**.
- The locations we examined are called a **probe sequence**.
- The probing process we followed is called **linear probing (γραμμική διερεύνηση).**
- So our hashing technique is called **open addressing with linear probing.**

Double Hashing

- **Double hashing** is another open addressing technique, which uses non-linear probing by computing **different probe decrements** for different keys using a **second hash function** $p(L_n)$.
- Let us define the following **probe decrement function**:

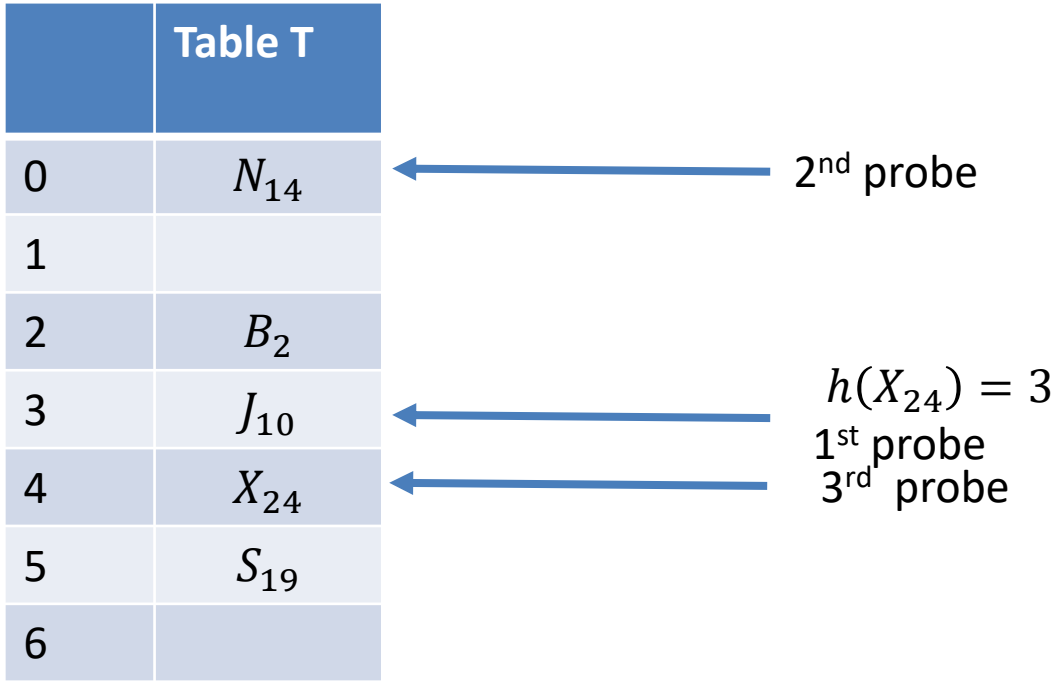
$$p(L_n) = \max(1, n/7)$$

- In the above equation:
  - The term $n/7$ denotes the quotient of the integer division of n by 7.
  - The operator $\max(a, b)$ returns the maximum of a and b.

Table T after Inserting keys $B_2, J_{10}, S_{19}, N_{14}$

	Table T
0	N_{14}
1	
2	B_2
3	J_{10}
4	
5	S_{19}
6	

Insert X_{24}



We use a probe decrement of $p(X_{24}) = 3$.

Insert W_{23}

	Table T
0	N_{14}
1	
2	B_2
3	J_{10}
4	X_{24}
5	S_{19}
6	W_{23}

$h(W_{23}) = 2$
1st probe

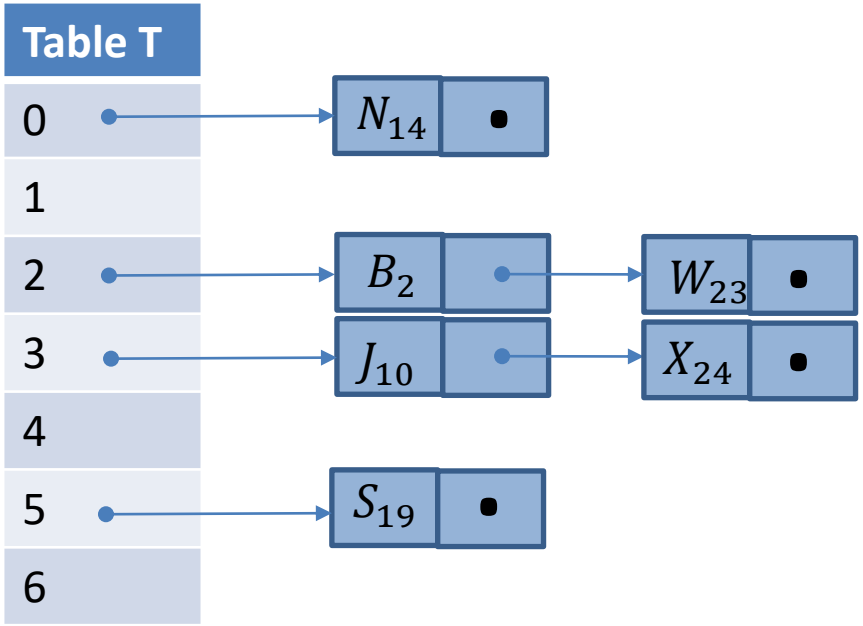
2nd probe

We use a probe decrement of $p(W_{23}) = 3$.
In general probe decrements will be **different** for different keys. Try inserting key P_{16} on your own.

Collision Resolution by Separate Chaining

- The method of **collision resolution by separate chaining (χωριστή αλυσίδωση)** uses a linked list to store keys at each table entry.
- This method should not be chosen if space is at a premium, for example, if we are implementing a hash table for a mobile device.

Example



Good Hash Functions

- Suppose T is a hash table having M entries whose addresses lie in the range 0 to $M - 1$.
- An **ideal hashing function** $h(K)$ maps keys onto table addresses in a **uniform and random** fashion.
- In other words, for any arbitrarily chosen key, any of the possible table addresses is equally likely to be chosen (the probability of being chosen is $\frac{1}{M}$).
- Also, the computation of a hash function should be **very fast.**

Collisions

- A **collision** between two keys K and K' happens if, when we try to store both keys in a hash table T, both keys have the same hash address $h(K) = h(K')$.
- **Collisions are relatively frequent** even in sparsely occupied hash tables.
- The **von Mises paradox**: if there are more than 23 people in a room, there is a greater than 50% chance that two of them will have the same birthday ($M = 365$). See the proof in the book by Standish.
- A good hash function should **minimize collisions**.

Primary clustering

- Linear probing suffers from what we call **primary clustering (πρωταρχική συσταδοποίηση)**.
- A **cluster** (**συστάδα**) is a sequence of adjacent occupied entries in a hash table.
- In open addressing with linear probing **such clusters are formed and then grow bigger and bigger**. This happens because all keys colliding in the same initial location trace out **identical search paths** when looking for an empty table entry.
- Double hashing does not suffer from primary clustering because initially colliding keys search for empty locations along **separate probe sequence paths**.

Ensuring that Probe Sequences Cover the Table

- In order for the open addressing hash insertion and hash searching algorithms to work properly, we have to guarantee that every probe sequence used can probe **all locations** of the hash table.
- This is obvious for linear probing.
- Is it true for double hashing?

Choosing Table Sizes and Probe Decrements

- If we choose the table size to be a **prime number (πρώτος αριθμός) M and probe decrements to be positive integers $p(K)$ in the range $1 \leq p(K) \leq M$** then we can ensure that the probe sequences cover all table addresses in the range 0 to $M - 1$ exactly once.
- Proof?

Proof

- For a key K for which there is a collision, the locations visited by the probe sequence are of the form

$$[h(K) - i * p(K)] \% M$$

for i in the range $0 \leq i \leq M - 1$.

- These are $M - 1$ positions and they cover the whole table.
- Let us assume by contradiction that they do not.

Proof (cont'd)

- Let us suppose that there are two distinct integers j and k in the range 0 to $M-1$ which generate the same probe location.
- In this case

$$[h(K) - j * p(K)]\%M = [h(K) - k * p(K)]\%M.$$

- This can be written in **number-theoretic terms** as

$$\{[h(K) - j * p(K)] \equiv [h(K) - k * p(K)]\} \, (modulo \; M).$$

Terminology and Notation

- Let us explain, the previous number-theoretic terminology and notation.
- We will say that two integers $\boldsymbol{a}$ **and** $\boldsymbol{b}$ **are congruent (ισοδύναμοι) modulo** $\boldsymbol{n}$ and denote this by

$$\boldsymbol{a \equiv b \ (mod \ n)}$$

if and only if $a - b$ is an integer multiple of n i.e., n divides $a - b$.

- **Examples**:

$$38 \equiv 14 \ (mod \ 12), 8 \equiv -7 \ (mod \ 5)$$

- More details in the course Number Theory.

Proof (cont'd)

- We can subtract $h(K)$ from both sides of this congruence and we can multiply by -1 to get

$$[j * p(K) \equiv k * p(K)]\ (modulo\ M).$$

- Because $p(K)$ and M are **relatively prime (σχετικώς πρώτοι)**, we can divide by $p(K)$ to arrive at

$$j \equiv k\ (modulo\ M).$$

- Using the definition of congruence, the above is equivalent to

$$(j\%M) = (k\%M).$$

Proof (cont'd)

- Now recall that j and k are integers in the range 0 to $M - 1$.
- Therefore, the above is equivalent to $j = k$.
- But this contradicts our initial assumption that j and k were distinct integers.
- Thus, we have proven that probe sequences cover all the addresses of the entire hash table exactly once.

Terminology

- Two integers a and b are **relatively prime** if they have no other common divisor than 1.
- **Examples**:
  - 7 and 5 are relatively prime.
  - 8 and 3 are relatively prime.
  - 8 and 4 are not relatively prime.

Proof (cont'd)

- In our proof, since M is prime, its divisors are 1 and M itself. Thus, it is relatively prime with $p(K)$ such that $1 \leq p(K) \leq M - 1$.

Good Double Hashing Choices

- Choose the table size M to be a **prime number**, and choose probe decrements, any integer in the range 1 to $M - 1$.
- Choose the table size M to be a **power of 2** and choose as probe decrements any **odd integer** in the range 1 to $M - 1$.
- In other words, it is good to **choose probe decrements to be relatively prime with M**.

Open Addressing with Double Hashing Implemented in C

- We will use the following two constants:

```
#define M 997   /* 997 is prime */
#define EmptyKey 0
```

Open Addressing (cont'd)

- We will define the following types:

```
typedef int KeyType;
typedef struct {
  /* some members of various types giving */
  /* information associated with search keys */
        } InfoType;
typedef struct {
           KeyType Key;
           InfoType Info;
         } TableEntry;
typedef TableEntry Table[M];
```

Initialization

```
/* global variable T for the hash table */
Table T;

void Initialize(void)
{
   int i;

   for (i=0; i<M; i++)
      T[i].Key=EmptyKey

}
```

Insertion

```
void HashInsert(KeyType K, InfoType I)
{
   int i;
   int ProbeDecrement;

   i=h(K);
   ProbeDecrement=p(K);

   while (T[i].Key != EmptyKey){
      i-=ProbeDecrement;
      if (i<0)
        i+=M;
    }
    T[i].Key=K;
    T[i].Info=I;
}
```

Search

```
int HashSearch(KeyType K)
{
   int I;
   int ProbeDecrement;
   KeyType ProbeKey;

   /*Initializations */
   i=h(K);
   ProbeDecrement=p(K);
   ProbeKey=T[i].Key;

   /* Search loop */
   while ((K!=ProbeKey) && (ProbeKey!=EmptyKey)){
      i-=ProbeDecrement;
      if (i<0)
         i+=M;
      ProbeKey=T[i].Key;
   }

   /* Determine success or failure */
   if (ProbeKey==EmptyKey)
      return -1;
   else
      return i;

}
```

Deletion

- The function for deletion from a hash table is left as an exercise.
- But notice that **deletion poses some problems**.
- If we delete an entry and leave a table entry with an empty key in its place then **we destroy the validity of subsequent search operations** because a search terminates when an empty key is encountered.
- As a solution, we can leave the deleted entry in its place and **mark it as deleted (or substitute it by a special entry "available").** Then search algorithms can treat these entries as not deleted while insert algorithms can treat them as deleted and insert other entries in their place.
- However, in this case, if we have many deletions, **the hash table can easily become clogged with entries marked as deleted.**

Code for Separate Chaining

```
typedef struct STnode* link;
struct STnode
   { Item item; link next; };

static link *heads, z;
static int N, M;
```

Code for Separate Chaining (cont'd)

```
static link NEW(Item item, link next)
  { link x = malloc(sizeof *x);
    x->item = item;
    x->next = next;
    return x;
  }

void STinit(int max)
  { int i;
    N = 0; M = max/5;
    heads = malloc(M*sizeof(link));
    z = NEW(NULLitem, NULL);
    for (i = 0; i < M; i++) heads[i] = z;
  }
```

Notes

- `N` is the **number of keys/elements in the hash table.**
- `M` is the **size of the hash table**.
- **We maintain `M` lists**, with head links in array `heads`, using a hash function to choose among the lists.
- The `STinit` function sets `M` such that **we expect the lists to have about 5 items each**; therefore, the other operations require just a few probes.

Code for Separate Chaining (cont'd)

```
Item searchR(link t, Key v)
  {
    if (t == z) return NULLitem;
    if (eq(key(t->item), v)) return t->item;
    return searchR(t->next, v);
  }


Item STsearch(Key v)
{ return searchR(heads[hash(v, M)], v); }
```

Code for Separate Chaining (cont'd)

```
void STinsert(Item item)
  { int i = hash(key(item), M);
    heads[i] = NEW(item, heads[i]);
    N++;
  }

void STdelete(Item item)
  { int i = hash(key(item), M);
    heads[i] = deleteR(heads[i], item);
  }
```

Hash Function

- The previous code assumes that we have defined an appropriate hash function.
- For example, for integers keys, we might have:

```
#define hash(v, M) (v % M)
```

Exercise

- Write the code for `deleteR`.

Load Factor

- The **load factor (συντελεστής πλήρωσης)** α of a hash table of size M with N occupied entries is defined by

$$\alpha = \frac{N}{M}.$$

- The load factor is an important parameter in characterizing the performance of hashing techniques.

Performance Formulas

- Let T be a hash table of size M with exactly N occupied entries i.e., with load factor $\alpha = \frac{N}{M}$.
- Let C_N be the average number of probe addresses examined during a **successful search.**
- Let C'_N be the average number of probe addresses examined during an **unsuccessful search (or insertion).**
- For the following results, we assume that our hash functions h and p are **uniform and random**.

Efficiency of Linear Probing

- For **open addressing with linear probing**, we have the following performance formulas:

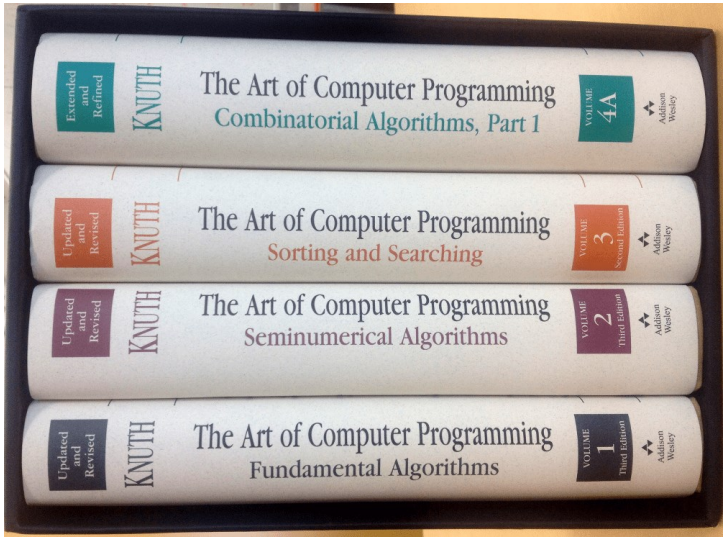
$$C_N = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

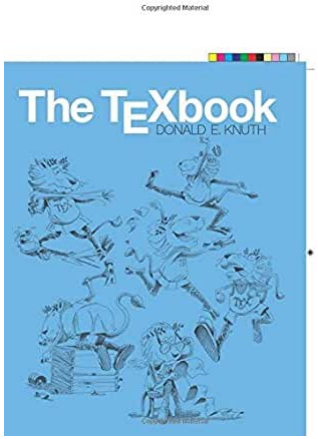
$$C'_N = \frac{1}{2}\left(1 + \left(\frac{1}{1-\alpha}\right)^2\right)$$

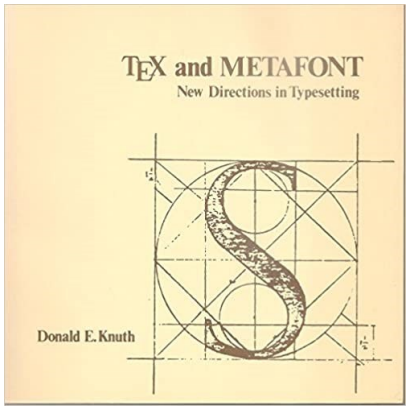
- The formulas are known to apply when the table T is up to 70% full (i.e., when $\alpha \leq 0.7$).
- This is an important result proved by Donald Knuth in 1962.

Donald Knuth

- **Donald Knuth** is one of the most important theoretical computer scientists (https://en.wikipedia.org/wiki/Donald_Knuth).
- He won the Turing Award in 1974.







Efficiency of Double Hashing

- For **open addressing with double hashing**, we have the following performance formulas:

$$C_N = \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$$

$$C'_N = \frac{1}{1-\alpha}$$

- This important result was proven by Guibas and Szeredi in 1976.

- **Notation**: $\ln(x)$ denotes the natural logarithm of x**.**

Efficiency of Separate Chaining

- For **separate chaining**, we have the following performance formulas:

$$C_N = 1 + \frac{1}{2}\alpha$$

$$C'_N = \alpha$$

- In a separate-chaining hash table of size M and N keys, the probability that the number of keys in each list is within a small constant factor of $\frac{N}{M}$ is extremely close to 1.

Proof

- Let us give only the easy proof that $C'_N = \alpha$ for the case of separate chaining.
- When we hash a key K into table T, we find a chain at location $h(K)$ which contains, let us say, j keys. It is possible that $j = 0$, if the chain at location $h(K)$ is empty.
- **Under the assumption that the hash function is uniform and random**, all of the keys N previously stored in table T are distributed uniformly over all of the M table locations, forming M separate chains (some of which are empty, and some of which contain one or more keys).

Proof (cont'd)

- Thus, the average chain length equals the number of keys N in all of the chains divided by the total number of chains M. In other words, the average chain length is $\alpha = \frac{N}{M}$.

- Since on the average, an unsuccessful search must compare the search key K with all keys in the chain, the average number of comparisons C'_N, for unsuccessful searching is just α.

Important

- Note that the previous formulas show that the **performance of a hash table depends only on the load factor** and not on any other parameter.

Theoretical Results: Apply the Formulas

- Let us now compare the performance of the techniques we have seen for different load factors using the formulas we presented.
- We apply the formulas to a table with size 997.

- Experimental results obtained after implementing the algorithms are similar.

Successful Search

	Load Factors					
	0.10	**0.25**	**0.50**	**0.75**	**0.90**	**0.99**
Separate chaining	1.05	1.12	1.25	1.37	1.45	1.49
Open/linear probing	1.06	1.17	1.50	2.50	5.50	50.5
Open/double hashing	1.05	1.15	1.39	1.85	2.56	4.65

- The cells of the table give C_N, the average number of probe addresses, for each of the hashing techniques.
- Notice that C_N increases when the load factor increases.
- Open addressing with linear probing performs worse than the other two techniques for big load factor.
- The best performing technique is separate chaining.

Unsuccessful Search

	Load Factors					
	0.10	**0.25**	**0.50**	**0.75**	**0.90**	**0.99**
Separate chaining	0.10	0.25	0.50	0.75	0.90	0.99
Open/linear probing	1.12	1.39	2.50	8.50	50.5	5000.5
Open/double hashing	1.11	1.33	2.50	4.00	10.0	100.0

- The cells of the table give C'_N, the average number of probe addresses, for each of the hashing techniques.
- Notice that C'_N increases when the load factor increases.
- Open addressing with linear probing performs worse than the other two techniques for big load factor.
- The best performing technique is separate chaining.

Complexity of Hashing

- Let us assume that we will use a hash table that is never more than **half-full** ($\alpha \leq 0.50$).
- If the table becomes more than half-full, we can expand the table by choosing a new table twice as big and by **rehashing** the entries in the new table.
- Suppose also that we use one of the hashing methods we presented.
- Then the previous tables show that **successful search can never take more than $\mathbf{1.50}$ key comparisons** and **unsuccessful search can never take more than $\mathbf{2.50}$ key comparisons.**
- So, the behaviour of hash tables is independent of the size of the table or the number of keys, hence **the complexity of searching is $\boldsymbol{O(1)}$**.

Complexity of Hashing (cont'd)

- **Insertion** takes the same number of comparisons as an unsuccessful search, so it has complexity $\boldsymbol{O(1)}$ as well.
- **Retrieving and updating** also take $\boldsymbol{O(1)}$ time**.**
- **Deletion** also takes $\boldsymbol{O(1)}$ time**.**
- To **enumerate the entries** of a hash table, we must first sort the entries into ascending order of their keys. This requires time $\boldsymbol{O(n \log n)}$ using a good sorting algorithm.

Load Factors and Rehashing

- In all the hash table schemes we discussed, the load factor a should be kept below 1.
- Experiments and average case analysis suggest that **we should maintain $a < 0.5$ for open addressing schemes and $a < 0.9$ for separate chaining.**
- With **open addressing**, as the load factor grows beyond 0.5 and starts approaching 1, clusters of items in the table start to grow as well.
- **At the limit, when a is close to 1, all table operations have linear expected running times** since, in this case, we expect to encounter a linear number of occupied cells before finding one of the few remaining empty cells.

Load Factors and Rehashing (cont'd)

- If the load factor of a hash table goes significantly above a specified threshold, then it is common to require the table to be resized to regain the specified load factor.
- This process is called **rehashing (ανακατακερματισμός)** or **dynamic hashing (δυναμικός κατακερματισμός).**
- When rehashing to a new table, a good requirement is having the new array's size be **at least double** the previous size.

Summary: Open Addressing or Separate Chaining?

- Open addressing schemes save space but they are not faster.
- As you can see in the above theoretical results (and corresponding experimental results), the separate chaining method is either competitive or faster than the other methods depending on the load factor of the table.
- So, **if memory is not a major issue, the collision handling method of choice is separate chaining**.

Comparing the Performance of Some Table ADT Representations

	Initialize	Determine if full	Search Retrieve Update	Insert	Delete	Enumerate
Sorted array of structs	$O(n)$	$O(1)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
AVL tree of structs (or (2,4) tree or red-black tree)	$O(1)$	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Hashing	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(n \log n)$

Choosing a Good Hash Function

- Ideally, a hash function will map keys **uniformly and randomly** onto the entire range of the hash table locations with each location being equally likely to be the target of the function for a randomly chosen key.

Example of a Bad Choice

- Suppose our **keys are variables up to three characters in a particular assembly language** using 8-bit ASCII characters.
- Thus, **we can represent each key by a 24-bit integer** divided into three equal 8-bit sections each representing an ASCII character.
- Suppose we use open addressing with double hashing.
- Suppose we select a table size $M = 2^8 = 256$.
- Suppose we define our hashing function as $h(K) = K\%256$.

Example (cont'd)

- **This hash function is a poor one** because it **selects the low-order character of the three-character key** as the value of $h(K)$.
- If the key is $C_3C_2C_1$, when considered as a 24-bit integer, it has the numerical value

$$C_3 * 256^2 + C_2 * 256^1 + C_1 * 256^0.$$

- Thus, when we take the remainder of the integer division with 256, we get the value C_1.

Example (cont'd)

- Now consider what happens when we hash the keys $X1, X2, X3, Y1, Y2, Y3$ using this hash function:

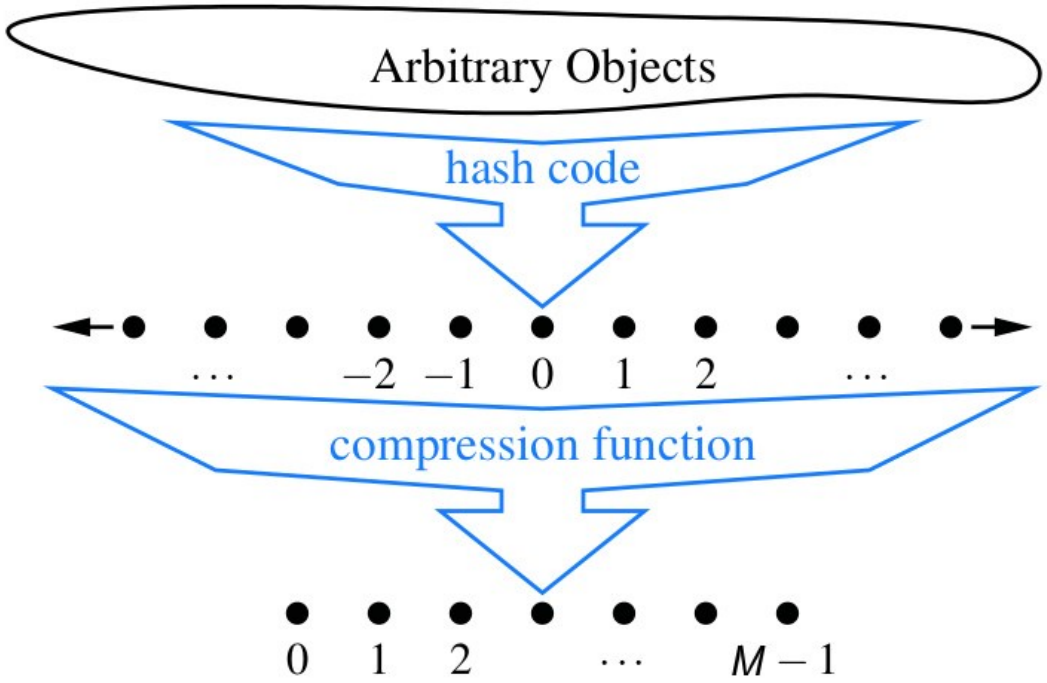
$$h(X1) = h(Y1) = 1$$
$$h(X2) = h(Y2) = 2$$
$$h(X3) = h(Y3) = 3$$

- So, the six keys will be mapped into a common cluster of three contiguous table addresses.
- Thus, this hash function will create and preserve clusters instead of **spreading** them as a good hash function will do.
- Hash functions should take into account **all the bits of a key**, not just some of them.

Designing Hash Functions

- Let M be the size of the hash table.
- We can view the evaluation of a hash function $h(K)$ as consisting of two actions:
  - Mapping the key K to an integer, called the **hash code**, and
  - Mapping the hash code to an integer within the range of indices 0 to $M-1$. This is called the **compression function (συνάρτηση συμπίεσης).**

Designing Hash Functions (cont'd)



Hash Codes

- The first action that a hash function performs is to take an arbitrary key K and map it into an integer value.
- This integer need not be in the range 0 to $M - 1$ and may even be negative, but we want the set of hash codes to **avoid collisions**.
- **If the hash codes of our keys cause collisions, then there is no hope for the compression function to avoid them.**

Hash Codes for Elements of C Datatypes

- The hash codes for elements of C datatypes described below are based on the assumption that the number of bits of each data type is known.

Converting to an Integer

- For any data type D that is represented using at most as many bits as our integer hash codes, we can simply **take an integer interpretation of the bits as a hash code** for elements of D.
- Thus, for the C basic types `char`, `short int` and `int`, we can achieve a good hash code simply by **casting** this type to `int`.

Converting to an Integer (cont'd)

- On many machines, the type `long int` has a bit representation that is twice as long as type `int`.
- One possible hash code for a `long int` element is to simply cast it down to an `int`. The result is implementation-dependent; typically, the higher order bits are ignored.
- But notice that this hash code **ignores half of the information** present in the original value. So, if many of the keys differ only in these bits, they will **collide** using this simple hash code.
- A **better hash code**, which takes all the original bits into consideration, **sums** an integer representation of the high-order bits with an integer representation of the low-order bits.

Converting to an Integer (cont'd)

- In general, if we have **an object x whose binary representation can be viewed as a k-tuple** of integers $(x_0, x_1, \ldots, x_{k-1})$, we can form a hash code for x as $\sum_{i=0}^{k-1} x_i$ ignoring overflows.
- **Example**: Given any floating-point number mEe, we can sum its mantissa m and exponent e as long integers and then apply a hash code for long integers to the result.
- Another technique would be to take the **exclusive-or** of the components.

Summation Hash Codes

- The summation hash code, described above, is **not a good choice for character strings or other variable-length objects that can be viewed as tuples of the form $(x_0, x_1, \ldots, x_{k-1})$ where the order of the x_i's is significant.**
- **Example**: Consider a hash code for a string s that sums the ASCII values of the characters in s. This hash code produces lots of unwanted collisions for common groups of strings e.g., `temp01` and `temp10`.
- A better hash code should take the order of the x_i's into account.

Polynomial Hash Codes

- Let a be an integer constant such that $a \neq 1$.

- We can use the polynomial
  $$x_0 a^{k-1} + x_1 a^{k-2} + \cdots + x_{k-2} a + x_{k-1}$$
  as a hash code for $(x_0, x_1, \ldots, x_{k-1})$.

- This is called a **polynomial hash code**.

Polynomial Hash Codes (cont'd)

- To evaluate the polynomial, we should use the efficient **Horner's method.**
- The idea behind this method is the following identity:

$$x_0 a^{k-1} + x_1 a^{k-2} + \cdots + x_{k-2} a + x_{k-1} =$$
$$x_{k-1} + a(x_{k-2} + a\big(x_{k-3} + \cdots + a(x_1 + a x_0)\big) \cdots))$$

- This allows us to evaluate the polynomial **with just $k-1$ additions and $k-1$ multiplications.**
- In general, Horner's rule allows the evaluation of a polynomial using an **optimal number of operations and numerical stability!**

Polynomial Hash Codes (cont'd)

- Intuitively, a polynomial hash code uses multiplication by different powers as a way to **spread out the influence of each component across the resulting hash code.**

- Of course, on a typical computer, evaluating a polynomial will be done using the finite bit representation for a hash code; hence, the value will periodically **overflow** the bits used for an integer.

- Since we are more interested in a good spread of the object $(x_0, x_1, \ldots, x_{k-1})$ with respect to other keys, **we simply ignore such overflows**.
- Still, we should be mindful that such overflows are occurring and **choose the constant a so that it has some nonzero, low-order bits**, which will serve to preserve some of the information content even when we are in an overflow situation.

Polynomial Hash Codes (cont'd)

- Experiments show that in a list of over 50,000 English words, if we choose $a = 33, 37, 39$ or 41, we produce **less than seven collisions** in each case.
- For the sake of speed, we can apply the hash code to only a fraction of the characters in a long string.

Polynomial Hash Codes (cont'd)

```
/* This is a complete hash function, not just a
hash code. */

int hash(char *K)
{
  int h=0, a=33;

  for (; *K!='\0'; K++)
    h=(a*h + *K) % M;

  return h;
}
```

Comments

- The previous function takes as input a **pointer to a character array**. This pointer points to a null-terminated string `K` and computes a hash function for it.
- The statement

```
h=(a*h + *K) % M
```

updates the value of `h` using its previous value, the value of base `a` and the ASCII value of the current character in string `*K`.

Polynomial Hash Codes (cont'd)

- In theory, we first compute a polynomial hash code and then apply the compression function *modulo* M (another way of saying that we compute the remainder of the integer division with M).

- The previous hash function takes the modulo M **at each step.**
- The two approaches are the same because the following equality holds for all a, b, x, M that are nonnegative integers:

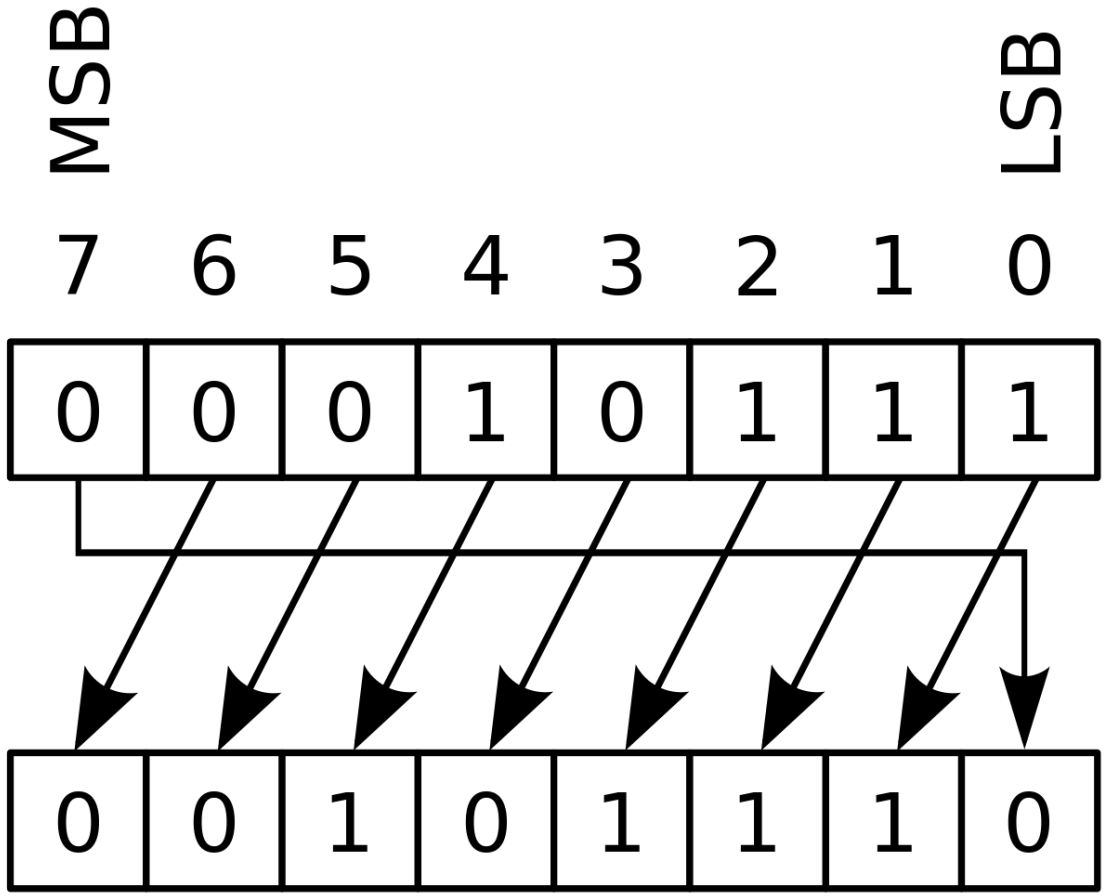
$$\Big(\big((ax)\ mod\ M\big) + b\Big)\ mod\ M = (ax + b)\ mod\ M$$

- **Note**: the mod operator is the same as $\%$
- The approach of the previous function is preferable because, otherwise, **we get errors with long strings** when the polynomial computation produces overflows (try it!).

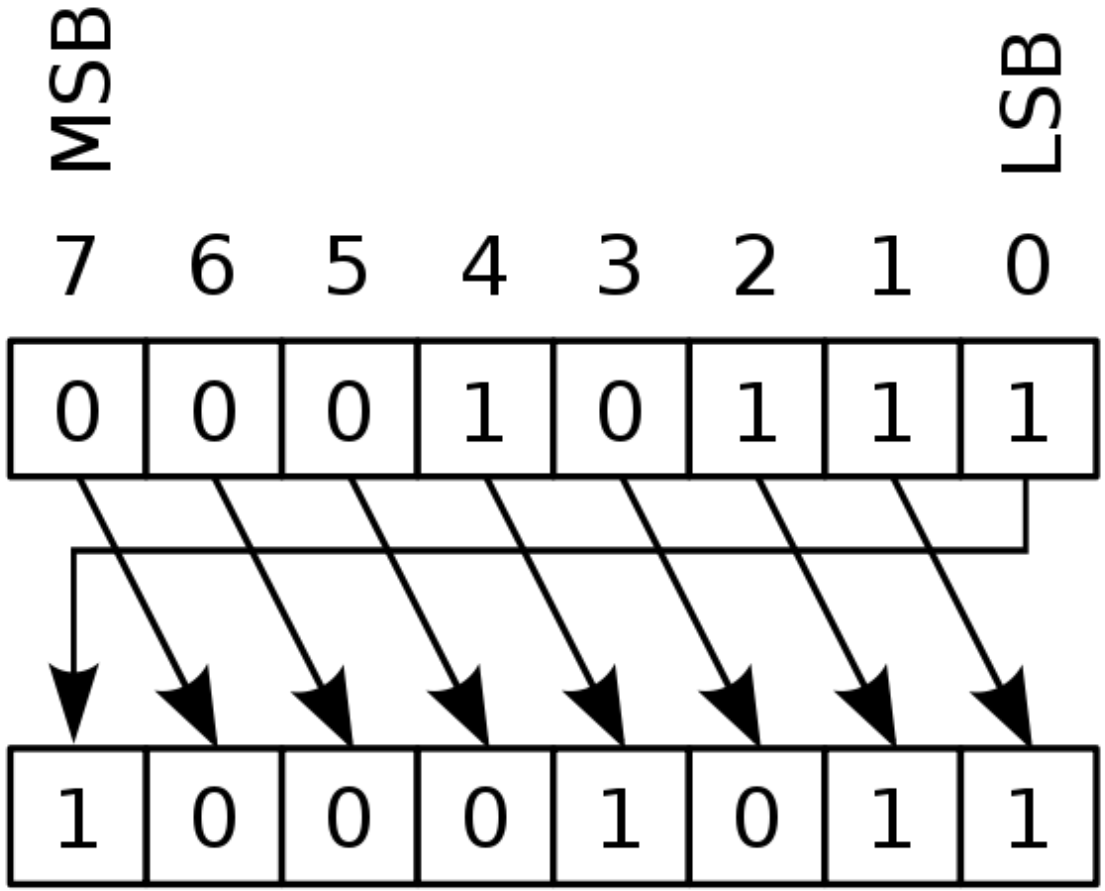
Cyclic Shift Hash Codes

- A variant of the polynomial hash code replaces multiplication by a with a **cyclic shift of a partial sum by a certain number of bits**.
- While this operation has little natural meaning in terms of arithmetic, **it accomplishes the goal of varying the bits of the calculation of the hash code.**

1-Bit Left Cyclic Shift



1-Bit Right Cyclic Shift



Cyclic Shift Hash Codes (cont'd)

- In C, a cyclic shift of bits can be accomplished through careful use of the **bitwise shift operators.**
- **Example**: the following function achieves a **5-bit cyclic shift** by taking the bitwise inclusive OR of a 5-bit left shift and a 27-bit right shift. The bitwise operators << and >> of C are used.

Cyclic Shift Hash Codes (cont'd)

```
int hashCode(const char *p, int len){
    unsigned int h=0;
    int i;

    for (i=0; i<len; i++){
        h=(h << 5) | (h >> 27); /* 5-bit cyclic shift of
                                   the running sum */
        h+=(unsigned int) p[i]; /* add in next character */
    }
    return h;
}
```

Comments

- Operation `h << 5`: The current hash value is left-shifted by 5 bits. This effectively "mixes in" the bits from the left side of `h` into the right side.
- `|`: The bitwise OR operator combines the left-shifted value with the right-shifted value (`h >> 27`). This ensures that all bits in `h` are eventually incorporated into the hash code.
- The result is a **5-bit cyclic shift** of the partial sum.

Experiment

Shift	Collisions	
	Total	Max
0	234735	623
1	165076	43
2	38471	13
3	7174	5
4	1379	3
5	190	3
6	502	2
7	560	2
8	5546	4
9	393	3
10	5194	5
11	11559	5
12	822	2
13	900	4
14	2001	4
15	19251	8
16	211781	37

Experiment (cont'd)

- The previous table presents a comparison of collision behavior for the cyclic-shift hash code as applied to a list of 230,000 English words.
- The "Total" column records the total number of words that collide with at least one other, and the "Max" column records the maximum number of words colliding at any one hash code.
- This experiment motivated the choice of 5 in our code since this is the case where we have the fewest total collisions.
- Note that with a cyclic shift of 0, this hash code reverts to the one that simply sums all the characters.

Hash Codes for Floating Point Numbers

- We can achieve a better hash code for floating point numbers than casting them down to `int` as follows.
- Assuming that a `char` is stored as an 8-bit byte, we could **interpret a 32-bit `float` as a four-element character array** and use the hash code functions we discussed for strings.

Compression Functions

- Once we have determined an integer hash code for a key K, there is still the issue of mapping that integer into the range 0 to $M - 1$.
- This can be done using the following methods.

The Division Method

- One good method for choosing a compression function is the **division method** we have demonstrated already in the context of double hashing:
  - M is chosen to be a **prime number (important!)**
  - $h(K) = K\%M$
  - $p(K) = \max(1, K/M)$

Why M Should Be a Prime Number?

- If we take M to be a prime number, then the modulo compression function helps "spread out" the distribution of hashed values.

- If M is not prime, then there is greater risk that patterns in the distribution of hash codes will be repeated in the distribution of hash values, thereby causing collisions.

- **Example**: If we insert keys with hash codes { 200, 205, 210, 215, 220, . . . , 600 } into a hash table of **size 100**, then **each hash code will collide with three others**. But if we use a bucket array of **size 101**, then there will be **no collisions**.

Precautions

- If there is a repeated pattern of key values of the form $iM + j$ for several different i, then there are still **collisions**.

Precautions (cont'd)

- If r is the radix (or base) of the character set for the keys and k and a are small integers, then we should **not** choose a prime of the form

$$M = r^k \pm a.$$

- **Radix r above refers to the size of the alphabet from which the characters in the keys are drawn**, treated as the base for a numerical interpretation of the key string.
- If our keys consist of 8-bit ASCII characters then $\boldsymbol{r = 256}$.

Example

- Let us suppose we again consider three character keys $C_3C_2C_1$ in 8-bit ASCII where the radix of the character set is again $2^8 = 256$.
- Suppose we choose $M = 2^{16} + 1 = 65537$.
- Then $h(C_3C_2C_1) = (C_2C_1 - C_3)_{256}$.
- Generally, if $M = r^k \pm a$, the value of $h(K)$ will tend to be simple sums and differences of products of the characters C_i.
- Such a hash function **will not spread clusters of keys, nor will it map keys uniformly and randomly onto the space of hash table locations**.

Double Hashing

- Another good way to choose $p(K)$ in double hashing is the following:

$$p(K) = Q - K\%Q$$

  where Q is prime and $Q < M$.
- The values of $p(K)$ are $1, \ldots, Q$.

The MAD Method

- A more sophisticated compression function which helps eliminate repeated patterns in a set of integer hash codes is the **multiply and divide** (or **MAD**) method.

- In this case, the compression function maps integer i to
$$h(i) = ((ai + b) \% p) \% M$$
where M is the size of the hash table, p is a prime number greater than M, a and b are nonnegative integers randomly chosen in the interval $[0, p-1]$ and $a > 0$.

Some Applications of Hash Tables

- Databases (including hashing for external storage).
- Cryptography
- Symbol tables in compilers
- Browser caches
- Peer-to-peer systems and torrents (distributed hash tables)
- …

Readings

- The material in the present slides comes (often verbatim) from the following sources:
  - T. A. Standish. *Data Structures, Algorithms and Software Principles in C*.
    - Chapter 11
  - M. T. Goodrich, R. Tamassia and Michael H. Goldwasser. *Data Structures and Algorithms in Java. 6th* edition. John Wiley and Sons, 2014.
    - Chapter 10
  - M. T. Goodrich, R. Tamassia. *Δομές Δεδομένων και Αλγόριθμοι σε Java*. 5η έκδοση. Εκδόσεις Δίαυλος.
    - Chapter 9
  - R. Sedgewick. *Αλγόριθμοι σε C*. 3η Αμερικανική Έκδοση. Εκδόσεις Κλειδάριθμος.
    - Κεφάλαιο 14