Lecture
8_8.2

Hashing

cini
Cybersecurity
National Lab

Paolo PRINETTO
Politecnico di Torino (Italy)
Univ. of Illinois at Chicago, IL (USA)
CINI Cybersecurity Nat. Lab. (Italy)

Paolo.Prinetto@polito.it

www.consorzio-cini.it
www.comitato-girotondo.org

License Information

Disclaimer

- We disclaim any warranties or representations as to the accuracy or completeness of this material.
- Materials are provided “as is” without warranty of any kind, either express or implied, including without limitation, warranties of merchantability, fitness for a particular purpose, and non-infringement.
- Under no circumstances shall we be liable for any loss, damage, liability or expense incurred or suffered which is claimed to have resulted from use of this material.

Rel. 07.04.2019

Goal

– This lecture aims at presenting Hash Tables as ADT, focusing first on some typical operations and then on possible implementation.

Rel. 07.04.2019

Prerequisites

- **Lectures:**
  - **8_1.1 Pointers & Dynamic Memory**
  - **8_2.1 Introduction to ADT**

Rel. 07.04.2019

Further readings

- **Students interested in a deeper look at the covered topics can refer, for instance, to the books listed at the end of the lecture.**

Rel. 07.04.2019

Outline

- Introduzione
- Hash Table come ADT
- Funzioni di Hash
- Collisione

Rel. 07.04.2019

Outline

- Introduzione
- Hash Table come ADT
- Funzioni di Hash
- Collisione

Rel. 07.04.2019

Searching

- Consider the problem of searching an array for a given value
  - If the array is not sorted, the search requires O(n) time
    - If the value isn't there, we need to search all n elements
    - If the value is there, we search n/2 elements on average
  - If the array is sorted, we can do a binary search
    - A binary search requires O(log n) time
    - About equally fast whether the element is found or not
  - It doesn't seem like we could do much better
    - How about an O(1), that is, constant time search?
    - We can do it *if* the array is organized in a particular way

Hashing

- Suppose we were to come up with a "magic function" that, given a value to search for, would tell us exactly where in the array to look
  - If it's in that location, it's in the array
  - If it's not in that location, it's not in the array
- This function would have no other purpose
- If we look at the function's inputs and outputs, they probably won't "make sense"
- This function is called a hash function because it "makes hash" of its inputs

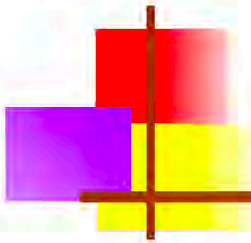
Example of an ideal Hash function

- Suppose our hash function gave us the following values:

```
hashCode("apple") = 5
hashCode("watermelon") = 3
hashCode("grapes") = 8
hashCode("cantaloupe") = 7
hashCode("kiwi") = 0
hashCode("strawberry") = 9
hashCode("mango") = 6
hashCode("banana") = 2
```

Hash Functions

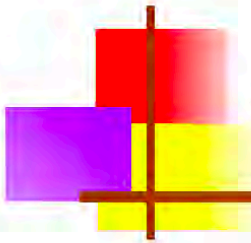
- A hash function is a function that:
  - When applied to an Object, returns a number
  - When applied to *equal* Objects, returns the *same* number for each
  - When applied to *unequal* Objects, is *very unlikely* to return the same number for each
- Hash functions turn out to be very important for searching, that is, looking things up fast



Sets and tables

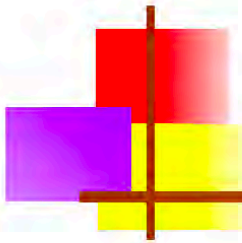
- Sometimes we just want a set of things—objects are either in it, or they are not in it
- Sometimes we want a map—a way of looking up one thing based on the value of another
  - We use a *key* to find a place in the map
  - The associated *value* is the information we are trying to look up
- Hashing works the same for both sets and maps
  - Most of our examples will be sets

. . .	*key*	*value*
141		
142	robin	robin info
143	sparrow	sparrow info
144	hawk	hawk info
145	seagull	seagull info
146		
147	bluejay	bluejay info
148	owl	owl info



Finding the hash function

- How can we come up with this magic function?
- In general, we cannot--there is no such magic function ☹
  - In a few specific cases, where all the possible values are known in advance, it has been possible to compute a perfect hash function
- What is the next best thing?
  - A perfect hash function would tell us exactly where to look
  - In general, the best we can do is a function that tells us where to *start* looking!

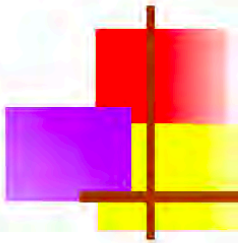


Example imperfect hash function

- Suppose our hash function gave us the following values:
  - hash("apple") = 5
    hash("watermelon") = 3
    hash("grapes") = 8
    hash("cantaloupe") = 7
    hash("kiwi") = 0
    hash("strawberry") = 9
    hash("mango") = 6
    hash("banana") = 2
    hash("honeydew") = 6

- Now what?

0	kiwi
1	
2	banana
3	watermelon
4	
5	apple
6	mango
7	cantaloupe
8	grapes
9	strawberry



Collisions

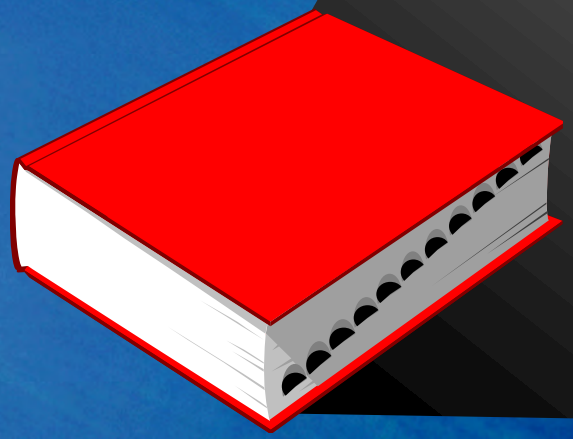
- When two values hash to the same array location, this is called a collision
- Collisions are normally treated as "first come, first served"—the first value that hashes to the location gets it
- We have to find something to do with the second and subsequent values that hash to this same location

Handling collisions

- What can we do when two different values attempt to occupy the same place in an array?
  - **Solution #1:** Search from there for an empty location
    - Can stop searching when we find the value *or* an empty location
    - Search must be end-around
  - **Solution #2:** Use a second hash function
    - ...and a third, and a fourth, and a fifth, ...
  - **Solution #3:** Use the array location as the header of a linked list of values that hash to this location
- All these solutions work, provided:
  - We use the same technique to *add* things to the array as we use to *search* for things in the array

Hash Table

ADT utilizzato per realizzare un vettore associativo



Vettore Associativo

- Un **vettore associativo** è un vettore in cui ciascun elemento è indirizzato dal suo contenuto piuttosto che dalla sua posizione.
- Ogni elemento contiene
  - Un dato secondo cui indirizzare (**chiave**)
  - Altri dati

Tecniche di Hashing

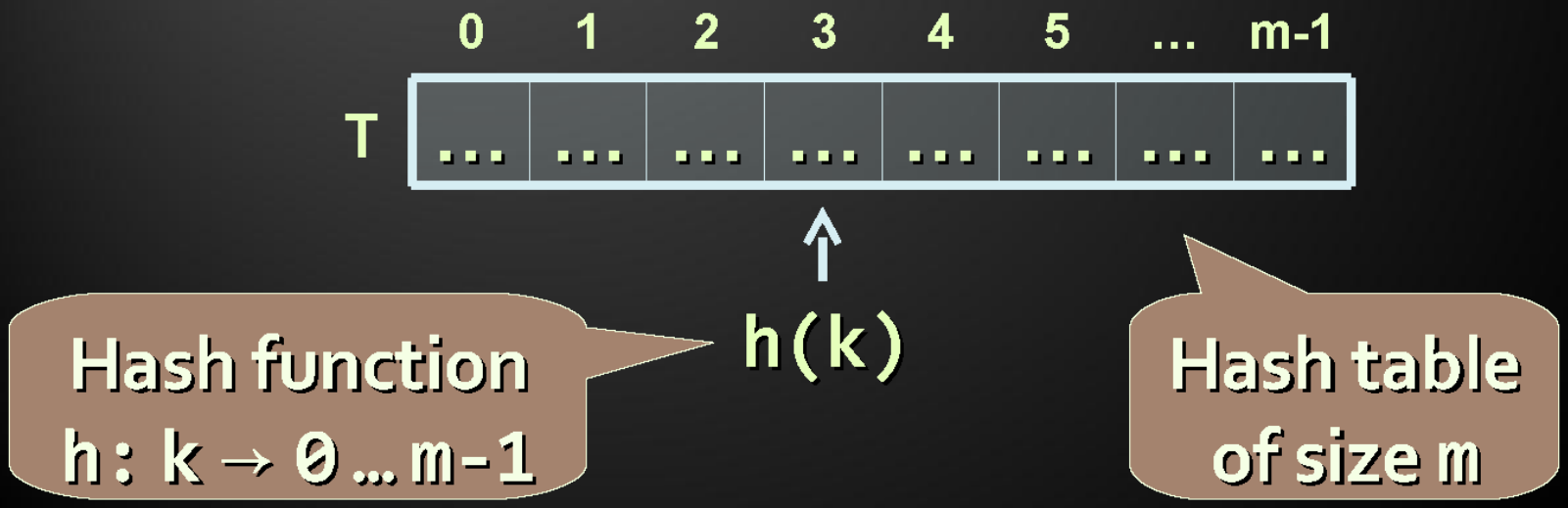
- Le tecniche di **hash** permettono la ricerca all'interno di un insieme di *n* chiavi in un tempo che, agli effetti pratici, è indipendente da *n* (ovvero *O(1)*).

- Sono basate sull'uso di una funzione *h(K)* detta **funzione di hash** che, operando su una chiave *K*, ritorna un intero compreso tra 0 e *m-1*.

- Tale valore viene poi usato come indice per accedere a un vettore di dimensione *m*, detto **tabella di hash (hashing table)**.

Hash Table

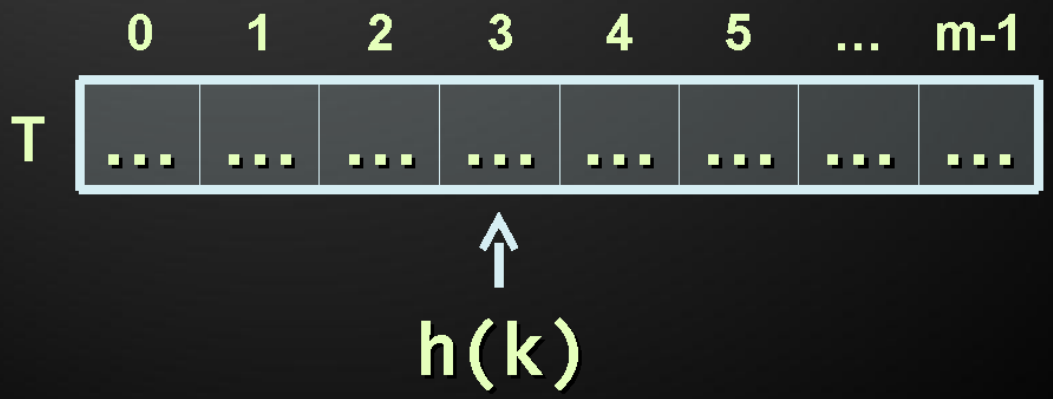
- A hash table is an array that holds a set of (key, value) pairs
- The process of mapping a key to a position in a table is called hashing



Rel. 07.04.2019

Hash Functions and Hashing

- A hash table has m slots, indexed from 0 to m-1
- A hash function h(k) maps keys to positions:
  - h: k → 0 … m-1
- For any value k in the key range and some hash function h we have h(k) = p and 0 ≤ p < m



Rel. 07.04.2019

Tecniche di Hashing *(cont'd)*

- **A seconda delle applicazioni, ciascuna cella della tabella può contenere un solo elemento o un numero massimo di elementi; in questo secondo caso prende il nome di bucket.**

- **È in genere richiesta la conoscenza del numero di elementi da memorizzare per dimensionare opportunamente la tabella.**

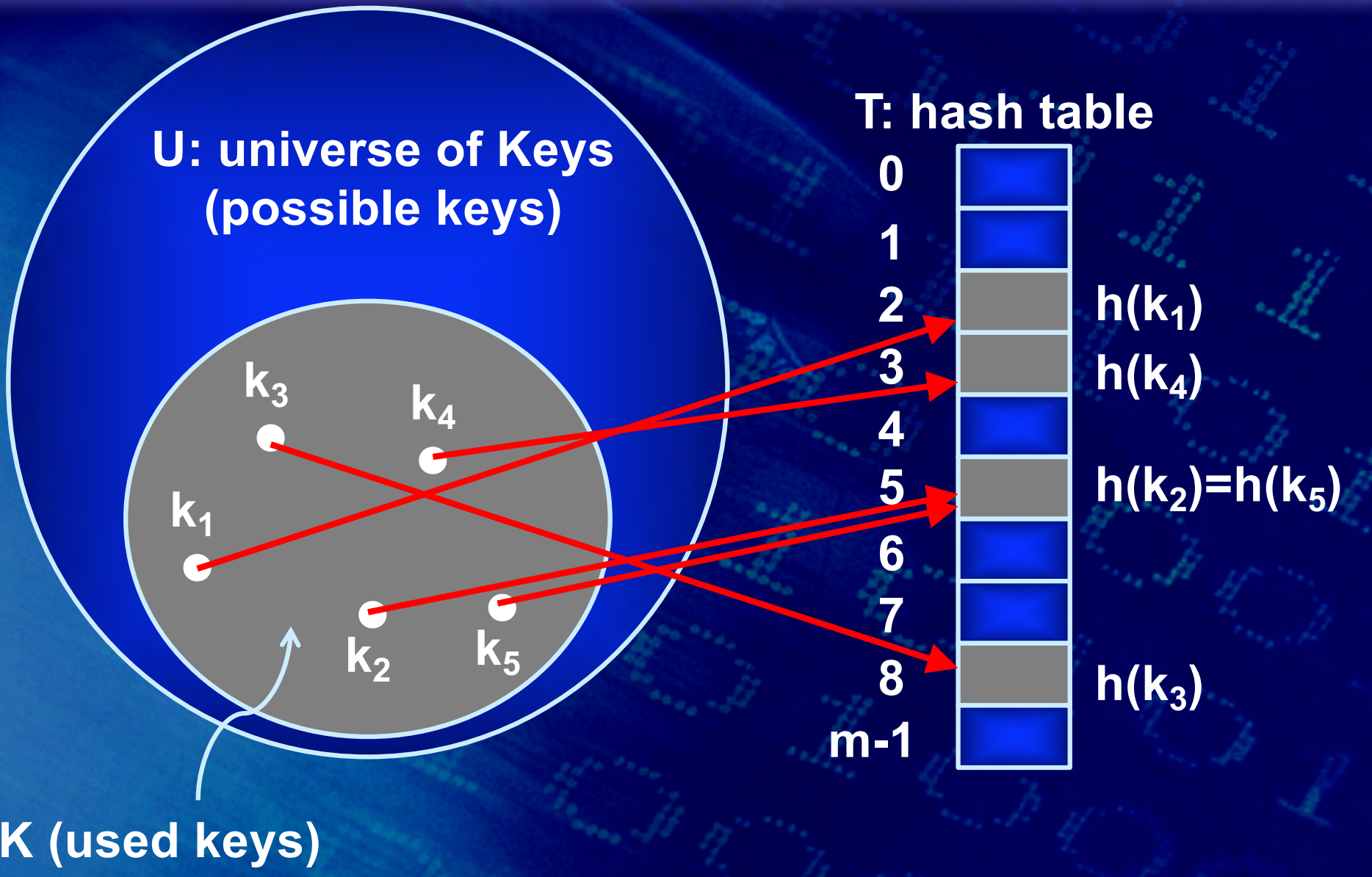
- **Le tecniche di hash non permettono una visita ordinata degli elementi presenti nella struttura di memorizzazione.**

Funzione di Hash

- **L'indirizzo di un elemento all'interno della tabella viene generato tramite la funzione di hash sia all'atto dell'inserimento sia nel momento della ricerca.**

- **La funzione *h(K)* deve ovviamente essere scelta in modo tale che i valori ritornati siano uniformemente distribuiti tra 0 e *m-1.***

Funzione di Hash



Rel. 07.04.2019

Collisione

- Si ha una **collisione** quando una funzione di hash ritorna lo stesso valore per due chiavi diverse, ovvero se:

$$h(k_i)=h(k_j) \text{ e } k_i \neq k_j$$

- È necessario:
  - Minimizzare il numero di collisioni (scegliendo opportunamente la funzione di hash)
  - Gestire le rimanenti collisioni

Rel. 07.04.2019

Collisions

- A collision is the situation when different keys have the same hash value

  $h(k_1) = h(k_2)$ for $k_1 \neq k_2$

- When the number of collisions is sufficiently small, the hash tables work quite well (fast)
- Several collisions resolution strategies exist
  - Chaining in a list
  - Using the neighboring slots (linear probing)
  - Re-hashing
  - …

Esempi di applicazione: Tabella dei Simboli di un Compilatore

- Un compilatore usa una **tabella dei simboli** per collegare i simboli ai dati associati
  - Simboli: nomi di variabili, nomi di funzioni, etc…
  - Dati associati: collocazione in memoria, grafo di chiamata, etc…
- Per una tabella dei simboli (anche chiamata **dizionario**) dobbiamo effettuare operazioni di ricerca, inserimento e cancellazione.
  - In genere non interessa che sia ordinata

Ulteriori esempi di applicazione

- **Rubrica telefonica**
  - **Chiave: Cognome**
  - **Numero di Chiavi possibili: molto grande**
  - ***m* : 21 (o 26)**
  - **Funzione di Hashing: Iniziale del cognome**

- **Internet Routing**
  - **Chiave: percorso del file**
  - **Informazioni aggiuntive: attributi e contenuto**

Rel. 07.04.2019

Ulteriori esempi di applicazione

- **Cybersecurity**
  - **Le funzioni di Hash sono ampiamente utilizzate per verificare l'integrità di messaggi/testi.**

Rel. 07.04.2019

Classificazione

Le tecniche di hashing possono essere classificate in base a caratteristiche diverse, quali:

- tipo di dizionario da memorizzare (statico o dinamico)
- funzione matematica usata nella trasformazione da chiave a indirizzo
- schema per il trattamento delle collisioni
- algoritmi per l'inserimento, la ricerca e la cancellazione
- algoritmi per il trattamento dell'overflow.

Outline

- Introduzione
- Hash Table come ADT
- Funzioni di Hash
- Collisione

Rel. 07.04.2019

Hash Table come ADT

Le operazioni che sono necessarie in una Hash Table, vista come ADT, sono le seguenti:

Rel. 07.04.2019

insert(T, x)

Inserisce un nuovo elemento *x*, nell'hash table *T*

search(T, k)

Cerca l'elemento con chiave *k* nell'hash table *T;*

se l'elemento esiste restituisce l'elemento *(x)*, altrimenti ERROR

delete(T, x)

Cancella l'elemento *x* dall'hash table *T;*

se l'elemento esiste restituisce OK, altrimenti ERROR

Outline

- Introduzione
- Hash Table come ADT
- Funzioni di Hash
- Collisione

Rel. 07.04.2019

Funzioni di hash

- Molti i modi proposti e impiegati per il calcolo della funzione di hash.
- Conviene classificarli in base al tipo di chiave trattata, vale a dire:
  - chiave alfanumerica
  - chiave intera.

Rel. 07.04.2019

Funzioni di hash per chiavi alfanumeriche

- **Nei casi in cui la chiave sia di tipo alfanumerico, la determinazione dell'indice avviene in due passi:**
  1. **conversione della chiave in un numero intero**
  2. **trasformazione di tale numero in indice.**

Rel. 07.04.2019

Funzioni di hash per chiavi alfanumeriche
(cont'd)

- **Se la chiave alfanumerica è corta può direttamente essere interpretata come un intero.**
- **In caso contrario, nella conversione da stringa a intero, occorre impiegare opportuni algoritmi di compressione.**
- **Tra i metodi più impiegati vanno ricordati:**
  - **metodo immediato**
  - **metodo della radice *K***

Rel. 07.04.2019

Metodo immediato

Usa come $h(K)$ la funzione:

$$h(K) = \left(\sum_{i=0}^{len-1} C_i \right) mod\, m$$

dove

- len **è la lunghezza della stringa**
- C_i **sono i valori della codifica ASCII dei caratteri**
- m **è la dimensione della tabella.**

Rel. 07.04.2019

Metodo immediato *(cont'd)*

Ad esempio, se:

- ***m*** **= 100**
- ***K*** **= "AGO"**

si ha:

$$h(K) = [65 + 71 + 79] \bmod 100 = 15$$

Il metodo immediato:

- **è un metodo molto rudimentale, che può generare numerose collisioni**
- **Un metodo sicuramente più efficiente è quello della "radice K".**

Metodo della radice K

- Converte la chiave in un numero in base K, dove K è il numero di possibili caratteri diversi.
- Ciascun carattere nella chiave viene trattato come una cifra di un numero in base K e successivamente riduce tale numero modulo un numero primo sufficientemente grande.

Rel. 07.04.2019

Metodo della radice K *(cont'd)*

- **Una possibile procedura di calcolo è la seguente.**

```
int radix_k(char s[]){
 int i,k,n,index;

 k = numero di caratteri distinti;
 m = dimensione della tabella;
 index = 0;

 for (i=0; i<strlen(s); i++)
    index = (k * index + s[i] - ' ') % m;

 return index;
}
```

Metodo della radice K: Esempio

Nel caso in cui sia:

- ***K*="AGO'**
- ***k*=26**
- ***m*=100**

si ha:

- **Passo 1:**
  index = (0+65-32) % 100 = 33
- **Passo 2:**
  index = (33+71-32) % 100 = 72
- **Passo 3:**
  index = (72+79-32) % 100 = 19

Rel. 07.04.2019

Funzioni di hash per chiavi intere

Tra i principali vanno ricordati:

- **metodo random**
- **metodo moltiplicativo (mid-square)**
- **metodo della codifica algebrica**
- **metodo delle divisioni**
- **metodo della ripiegatura.**

Rel. 07.04.2019

Metodo random

- usa come h(K) la funzione per la generazione di numeri pseudo-casuali disponibile nella maggior parte dei linguaggi di programmazione
- K viene usato come **seme** (seed)
- il valore di h(K) è il primo valore ritornato dalla funzione
- se necessario i valori successivi possono essere impiegati per la gestione delle eventuali collisioni
- in numerosi casi il metodo fornisce risultati inferiori a quelli ottimali.

Rel. 07.04.2019

Metodo moltiplicativo (mid-square)

- assume che la chiave *K* sia formata da *k* bit e che la tabella abbia dimensioni $m=2^p$, con $p < k$
- moltiplica la chiave o per se stessa o per una costante
- usa come indirizzo i *p* bit "centrali" del risultato ottenuto.

Rel. 07.04.2019

Metodo della codifica algebrica

- assume che la chiave *K* sia formata da *k* bit e che la tabella abbia dimensioni $m=2^p$, con $p < k$
- considera come valore della *h(K)* il resto della divisione del polinonmio *K(x)* per un polinomio di grado *p*
- per minimizzare la probabilità di aliasing è conveniente scegliere come divisori esclusivamente dei polinomi primitivi
- è particolarmente adatto a essere implementato via hardware.

Rel. 07.04.2019

Metodo delle divisioni

- è una variante del metodo precedente, in cui *h(K)* viene calcolata come:

$$h(K) = K \bmod m$$

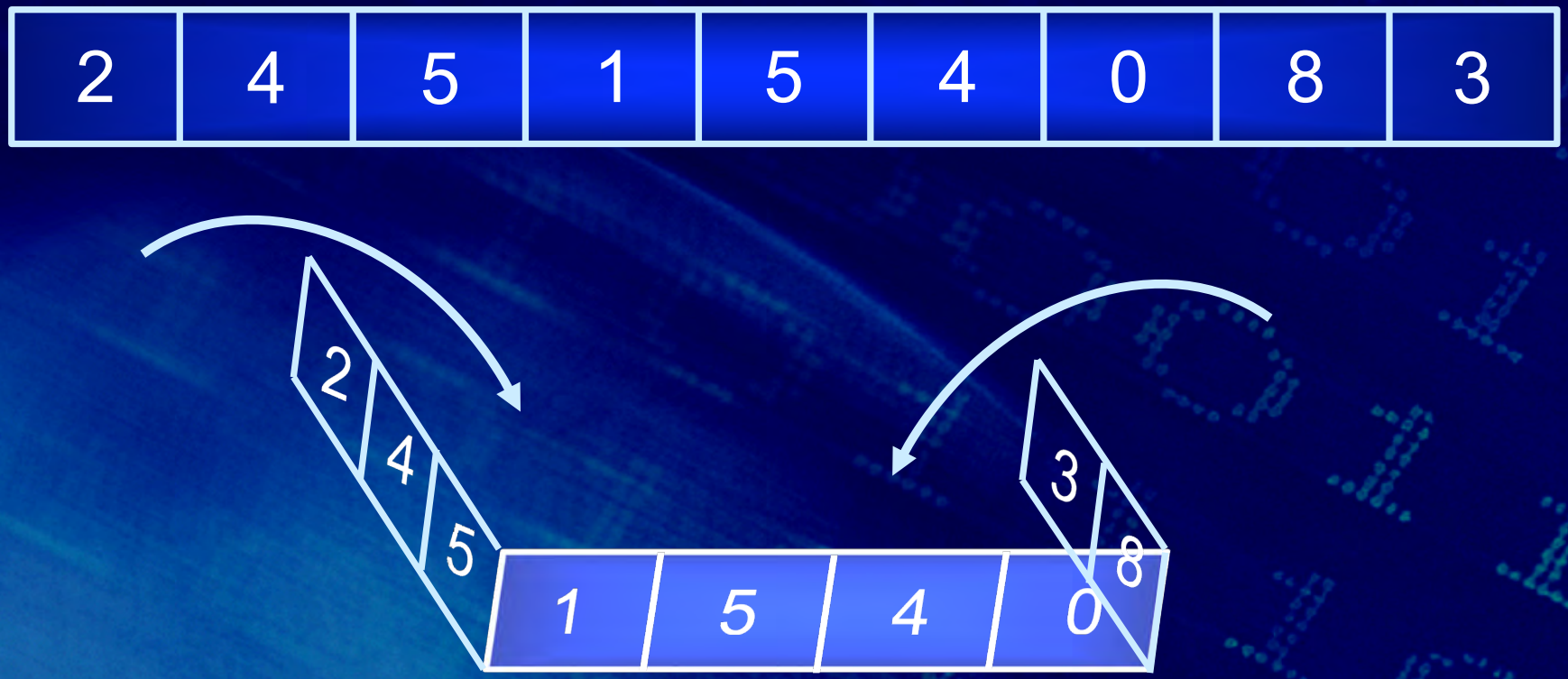
  essendo *m* la dimensione della tabella
- per minimizzare la probabilità di collisione è conveniente far sì che m sia primo.

Rel. 07.04.2019

Metodo della ripiegatura (folding)

- assume che la chiave *K* sia formata da *k* bit e che la tabella abbia dimensioni $m=2^p$, con $p < k$
- si suddividono i *k* bit della chiave *K* in gruppi di *p* bit
- si considerano coppie di gruppi e si eseguono delle operazioni tra i bit corrispondenti.
- Tipicamente si esegue:
  - la somma se i gruppi sono visti come cifre decimali
  - l'operazione di exor se i gruppi sono visti come sequenze di bit.

Rel. 07.04.2019



Rel. 07.04.2019

2	4	5	1	5	4	0	8	3

4 +

1 + 5 + 2 + 0 +

5 = 4 = 3 = 8 =

6 9 9 8

Rel. 07.04.2019

Outline

- Introduzione
- Hash Table come ADT
- Funzioni di Hash
- Collisione

Rel. 07.04.2019

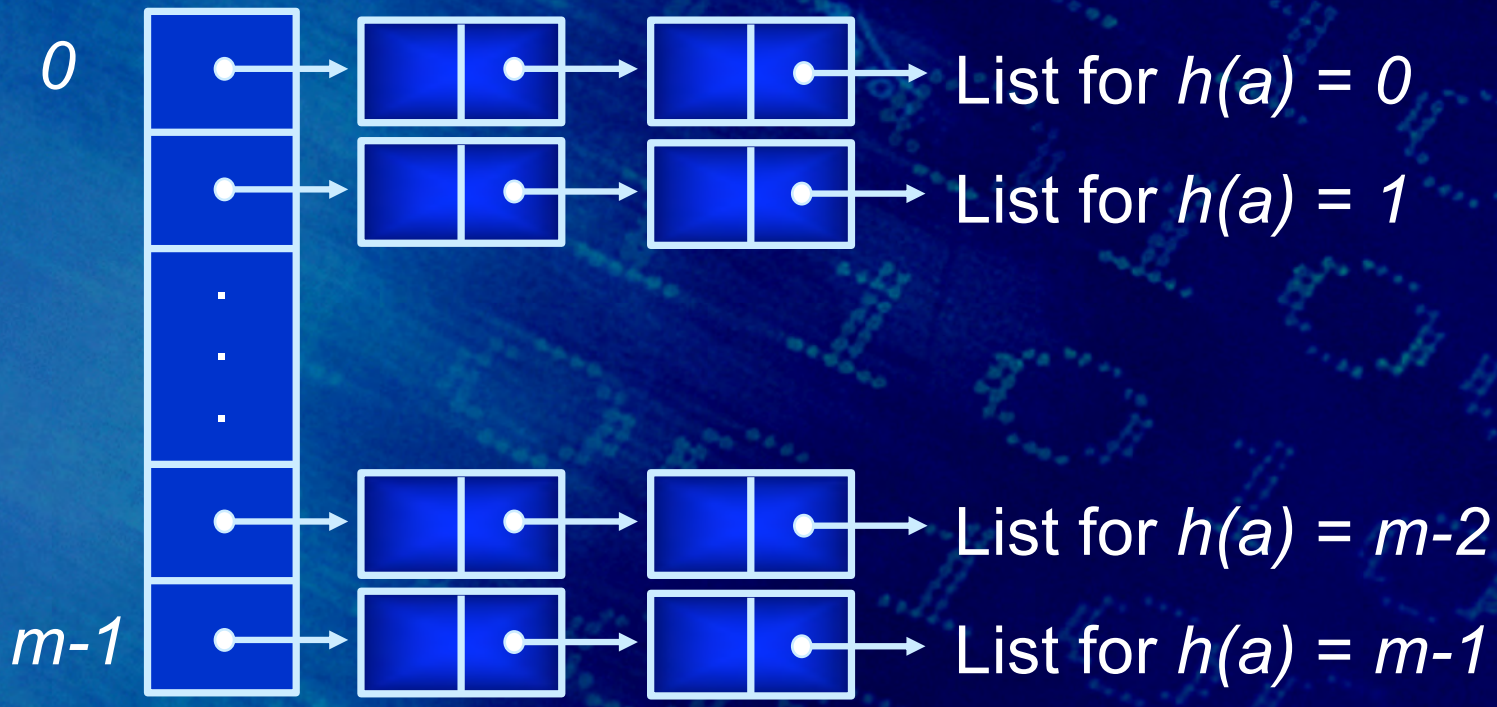
Trattamento delle collisioni

Due le soluzioni più comuni al problema della collisione:

- **chaining**
- **open addressing**

Chaining

- **Ciascuna cella della tabella non contiene un elemento, ma un puntatore a una lista di elementi aventi lo stesso valore di hash:**



Rel. 07.04.2019

Chaining *(cont'd)*

Vantaggi

- **il numero degli elementi da memorizzare non deve essere noto a priori**
- **è facile eseguire la cancellazione di un elemento**
- **non si ha mai overflow.**

Svantaggi

- **si devono gestire delle liste (alcuni linguaggi non lo permettono)**
- **si spreca spazio per i puntatori**
- **le liste possono diventare molto lunghe.**

Chaining: Analisi della complessità

Siano:

- ***n*** **il numero di elementi inseriti**
- ***m*** **il numero di celle nel vettore**

Se la funzione di hash distribuisce uniformemente le chiavi tra tutte le celle del vettore, la lunghezza media delle liste sarà *n/m*, e quindi si eseguiranno mediamente *O(n/m)* confronti per ogni richiesta.

Al crescere di *n/m* si può riallocare il vettore di dimensioni maggiori ed eseguire un re-hashing.

Rel. 07.04.2019

Open addressing

- In questo metodo, la tabella viene dimensionata in modo da poter contenere tutti gli elementi del dizionario.
- In caso di collisione, viene stabilita una sequenza di celle da visitare, nelle quali può trovarsi l'elemento cercato.
- Tra le diverse strategie possibili, le più comuni sono:
  - probing
  - double hashing
  - coalesced hashing

Probing

- **Il principio è illustrato dal programma seguente, in cui, al variare della funzione *G(K)*, varia la strategia di scelta delle celle da visitare successivamente.**

Rel. 07.04.2019

Probing (cont'd)

```
h = h(k);
i = 0;
do{
 if T(h).key == k
    found;
 else
    if T(h).key == free
       not present;
    else{ /* collisione */
       i++;
       h = h(k) + G(i);
    }
 } while(not found && not (not present)
                   && table not full);
```

Probing (cont'd)

Le tecniche di probing più impiegate sono le seguenti:

- **probing lineare**
- **probing quadratico**
- **probing random.**

Rel. 07.04.2019

Probing Lineare

Si provano le caselle adiacenti (in genere successive) a quelle fornite dalla funzione di hash.

In pratica la ricerca avviene esaminando le celle

$$(h(K) + i) \bmod m$$

con $1 \leq i \leq m-1$.

In questo modo le chiavi tendono ad ammucchiarsi intorno ad alcuni valori.

Rel. 07.04.2019

Probing Lineare *(cont'd)*

- **Il numero medio di celle cui si accede prima di poter rispondere alla interrogazione o prima di poter inserire un elemento nel vettore è dato da:**

$$E = \frac{1 - \alpha/2}{1 - \alpha}$$

dove

$$\alpha = \frac{n}{m + 1}$$

è il fattore di carico del vettore:

- α = 0 ➔ **vettore vuoto**
- α = $(m/m + 1)$ ➔ **vettore pieno.**

Probing Lineare *(cont'd)*

- **Il numero di tentativi è mediamente molto piccolo per α non prossimo a 1:**

α	E
0.1	1.06
0.25	1.17
0.5	1.50
0.75	2.50
0.9	5.50
0.95	10.50

Probing Lineare *(cont'd)*

- **Il numero di tentativi è mediamente molto piccolo per α n**

Quando il vettore è pieno al 70:80% (α = 0.7:0.8) occorre aumentare la dimensione del vettore stesso e procedere a un re-hash degli elementi già memorizzati

Probing quadratico

La ricerca avviene esaminando le celle

$$(h(K) + i^2) \bmod m$$

e

$$(h(K) - i^2) \bmod m$$

con $1 \leq i \leq (m-1)/2$.

Si può dimostrare che quando m è un numero primo del tipo $4j+3$, con j intero, il probing quadratico esamina tutte le celle comprese tra 0 e $m-1$.

Probing Random *(cont'd)*

La tabella seguente riporta alcuni numeri primi della forma 4j+3.

m	*j*
3	0
7	1
11	2
19	4
23	5
31	7
43	10
59	14

Probing Random *(cont'd)*

- **La ricerca avviene esaminando le celle i cui indirizzi sono forniti da una funzione generatrice di numeri pseudo-casuale, il cui seme è dato dal valore della chiave *K*.**

Rel. 07.04.2019

Double Hashing

- **Si sostituisce *G(i)* con una seconda funzione di hash, ad esempio:**

  $f_2(K) = n - (K \setminus n)$ **con** $n < m$

  $f_3(K) = 1 + K \setminus (m-2)$

- **Il numero di confronti richiesto in media è in questo caso minore che con il probing lineare, ma è necessario ogni volta calcolare una seconda funzione di hash.**

Rel. 07.04.2019

Coalesced hashing

- **Il vettore è composto da *m* + *c* celle, ma la funzione di hash ritorna valori tra *0* e *m - 1*.**
- **In caso di collisione, l'elemento viene messo (o cercato) nella cella libera con indirizzo minore all'interno della parte di vettore avente indice maggiore di *m - 1*, chiamata cellar.**
- **Risultati sperimentali hanno dimostrato che i migliori risultati si ottengono dimensionando il cellar pari a circa il 14% dell'intero vettore.**

Analisi comparativa

Le prestazioni delle varie tecniche di open addressing dipendono non tanto dalle dimensioni del dizionario, quanto dal fattore di carico della tabella.

Siano:

- **α fattore di carico della tabella**
- ***S* il numero medio di confronti richiesti per trovare un elemento**
- ***U* il numero previsto di confronti richiesti nel caso in cui l'elemento cercato non sia presente nella tabella.**

Rel. 07.04.2019

Analisi comparativa *(cont'd)*

Per il probing lineare si ha:

$$U \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right); \qquad S \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$$

Per il probing quadratico, per quello random e per il double hashing si ha:

$$U \approx \frac{1}{1-\alpha}; \qquad S \approx \frac{1}{\alpha}\log_e(1-\alpha)$$

Per il chaining si ha:

$$U \approx \alpha; \qquad S \approx 1 + \frac{\alpha}{2}$$

Problema delle cancellazioni

La cancellazione di elemento, mentre non pone problemi nel caso del chaining, richiede particolare attenzione nel caso dell'open addressing, in quanto un'eventuale cella vuota può interrompere la sequenza di ricerca in caso di collisione.

Rel. 07.04.2019

Problema delle cancellazioni (cont'd)

Conviene aggiungere a ciascun record della tabella un campo in grado di assumere i tre valori libero, occupato o cancellato.

I record cancellati vanno trattati:

- **in fase di ricerca, alla stessa stregua di quelli occupati**
- **in fase di inserimento, alla stessa stregua di quelli liberi.**

Rel. 07.04.2019

Tecniche di hash per dizionari statici

Nel caso di dizionari statici, può essere conveniente ricorrere al cosiddetto perfect hashing, vale a dire a funzioni di hash in grado di far corrispondere a ciascuna chiave una diversa posizione nella tabella.

Purtroppo:

- **le funzioni in grado di garantire un hashing perfetto sono molto rare**
- **gli algoritmi per calcolarle sono talmente complicati da essere applicabili a dizionari con poche decine di chiavi.**

Segmentazione

Una delle tecniche più utilizzate per superare queste difficoltà è la segmentazione, che, in pratica, adotta una tecnica di double hashing:

- l'insieme delle chiavi di partenza viene suddiviso in
- sottoinsiemi di cardinalità limitata, chiamati buckets,
- le cui chiavi ritornano tutte uno stesso valore della funzione di hash
- per determinare la posizione dell'elemento cercato all'interno del
- buckets, si impiega una funzione di hash perfetta.

Minimal Pefect Hashing

In alcuni casi è possibile ricavare funzioni di hashing che permettono il minimal perfect hashing: a ogni chiave corrisponde una diversa posizione nella tabella, e viceversa.

Rel. 07.04.2019

References

- A.V. Aho, J.E. Hopcroft, J.D. Ullman:
  "Data Structures and Algorithms,"
  Addison Wesley, Reading MA (USA), 1983
  pp 107-154
- G.H. Gonnet:
  "Handbook of Algorithms and Data Structures,"
  Addison Wesley, Reading MA (USA), 1984, pp. 37-67
- E. Horowitz, S. Sahni:
  "Fundamentals of Computer Algorithms,"
  Pittman, London (UK), 1978
  pp 452-473

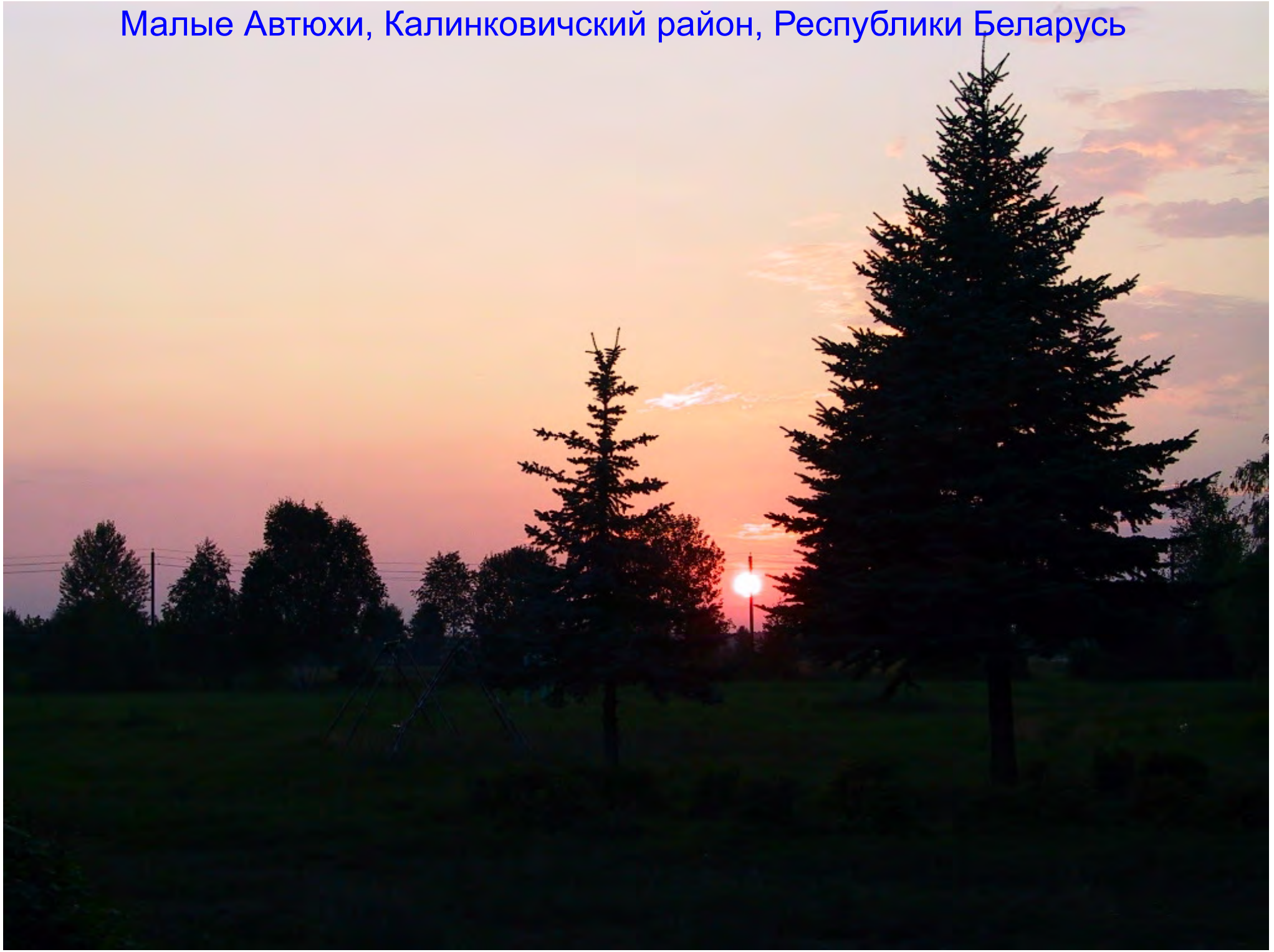
Rel. 07.04.2019

References

- T.G. Lewis, C.R. Cook:
  "Hashing for Dynamic and Static Internal Tables,"
  IEEE Computer, October 1988, pp. 45:56
- W.D. Maurer, T.G. Lewis:
  "Hash table methods,"
  Computing surveys, Vol. 7, n.1, March 1975, pp. 5-19
- R. Sedgewick:
  "Algorithms in C,"
  Addison Wesley, Reading MA (USA), 1990
  pp 177-192

References

- C.J. Van Wyk:
  "Data Structures and C Programs," Addison Wesley, Reading MA (USA), 1988
  pp 231-244


Малые Автюхи, Калинковичский район, Республики Беларусь