Lezione 10

Tabelle Hash

Rossano Venturini

rossano.venturini@unipi.it

Pagina web del corso

http://didawiki.cli.di.unipi.it/doku.php/informatica/all-b/start

Esercizio 2

ABR: Visita

Scrivere un programma che legga da tastiera una sequenza di N interi distinti e li inserisca in un albero binario di ricerca (senza ribilanciamento). Il programma deve visitare opportunamente l'albero e restituire la sua altezza.

Esercizio 2

```
typedef struct _node {
  struct _node *left;
  struct _node *right;
  int value;
} node;

int maxdepth(node *n) {
  if (n == NULL) return 0;
  return 1 + max( maxdepth(n->left), maxdepth(n->right) );
}
```

Esercizio 5

Albero Ternario (prova del 01/02/2012)

Scrivere un programma che riceva in input una sequenza di N interi positivi e costruisca un albero **ternario** di ricerca **non** bilanciato. L'ordine di inserimento dei valori nell'albero deve coincidere con quello della sequenza.

Ogni nodo in un albero ternario di ricerca può avere fino a tre figli: figlio sinistro, figlio centrale e figlio destro. L'inserimento di un nuovo valore avviene partendo dalla radice dell'albero e utilizzando la seguente regola. Il valore da inserire viene confrontato con la chiave del nodo corrente. Ci sono tre possibili casi in base al risultato del confronto:

1. se il valore è minore della chiave del nodo corrente, esso viene inserito ricorsivamente nel sottoalbero radicato nel figlio sinistro;

2. se il valore è **divisibile** per la chiave del nodo corrente, esso viene inserito ricorsivamente nel sottoalbero radicato nel figlio centrale;

3. in ogni altro caso il valore viene inserito ricorsivamente nel sottoalbero radicato nel figlio destro.

Il programma deve stampare il numero di nodi dell'albero che hanno **tre** figli.

Esercizio 5

```
typedef struct _Nodo {
  int key;
  struct _Nodo* left;
  struct _Nodo* central;
  struct _Nodo* right;
} Nodo;
```

Esercizio 5

```
void insert(Nodo **t, int k) {
 Nodo *e, *p, *x;

 /* crea il nodo foglia da inserire contenente la chiave */
 e = (Nodo *) malloc(sizeof(Nodo));
 if (e == NULL) exit(-1);
 e->key = k;
 e->left = e->right = e->central = NULL;
 x = *t;

 /* se l'albero è vuoto imposta e come radice dell'albero */
 if (x == NULL) {
    *t = e;
    return;
}

continua ...
```

Esercizio 5

```
 /* altrimenti cerca la posizione della foglia nell'albero */
while (x != NULL) {
  p = x;
  if(k % x->key == 0) x = x->central;
  else {
    if (k < x->key) x = x->left;
    else x = x->right;
  }
 }

 /* ora p punta al padre del nuovo elemento da inserire in t quindi si procede a
collegare p ed e */
 if(k % p->key == 0) p->central = e;
 else {
   if (k < p->key) p->left = e;
   else p->right = e;
 }
```

Esercizio 5

```
int conta(Nodo * node) {
  int r, c, l, curr;

  if (node == NULL) return 0;
  r = c = l = curr = 0;

  if (node->right != NULL) { r = conta(node->right); curr++;}
  if (node->left != NULL) { l = conta(node->left); curr++;}
  if (node->central != NULL) { c = conta(node->central); curr++;}

  if (curr == 3) return r+l+c+1;
  else return r+l+c;
}
```

Tabelle Hash

Soluzione standard per il problema del dizionario

Tabelle Hash

Soluzione standard per il problema del dizionario

Mantenere un insieme $S \subseteq U$ e fornire le seguenti operazioni

Insert(x): inserisce x in S

Delete(x): rimuove x da S

Search(x): determina se x è in S

Tabelle Hash

Soluzione standard per il problema del dizionario

Mantenere un insieme $S \subseteq U$ e fornire le seguenti operazioni

Insert(x): inserisce x in S

Delete(x): rimuove x da S

Search(x): determina se x è in S

Vedremo Tabelle hash con gestione dei conflitti con liste di adiacenza

Tabelle Hash

Tabelle Hash

S={5, 10, 15, 20, 22, 25, 33}

Tabelle Hash

S={5, 10, 15, 20, 22, 25, 33}

h: S$\rightarrow$|T|

h(5)=1

h(10)=h(22)=h(33)=3

h(15)=h(20)=7

h(25)=9

Tabelle Hash

S={5, 10, 15, 20, 22, 25, 33}

h: S$\rightarrow$|T|

h(5)=1

h(10)=h(22)=h(33)=3

h(15)=h(20)=7

h(25)=9

T

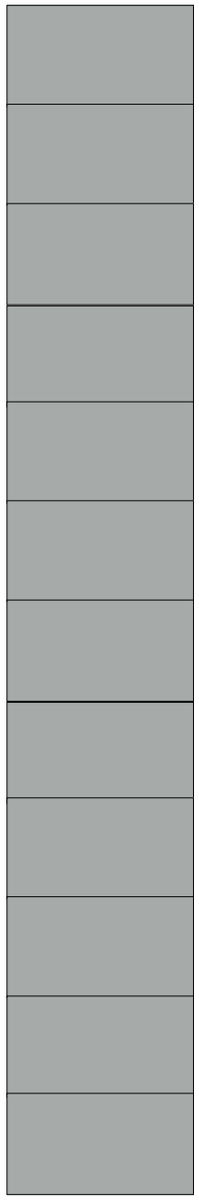


Tabelle Hash

S={5, 10, 15, 20, 22, 25, 33}

h: S$\rightarrow$|T|

h(5)=1

h(10)=h(22)=h(33)=3

h(15)=h(20)=7

h(25)=9

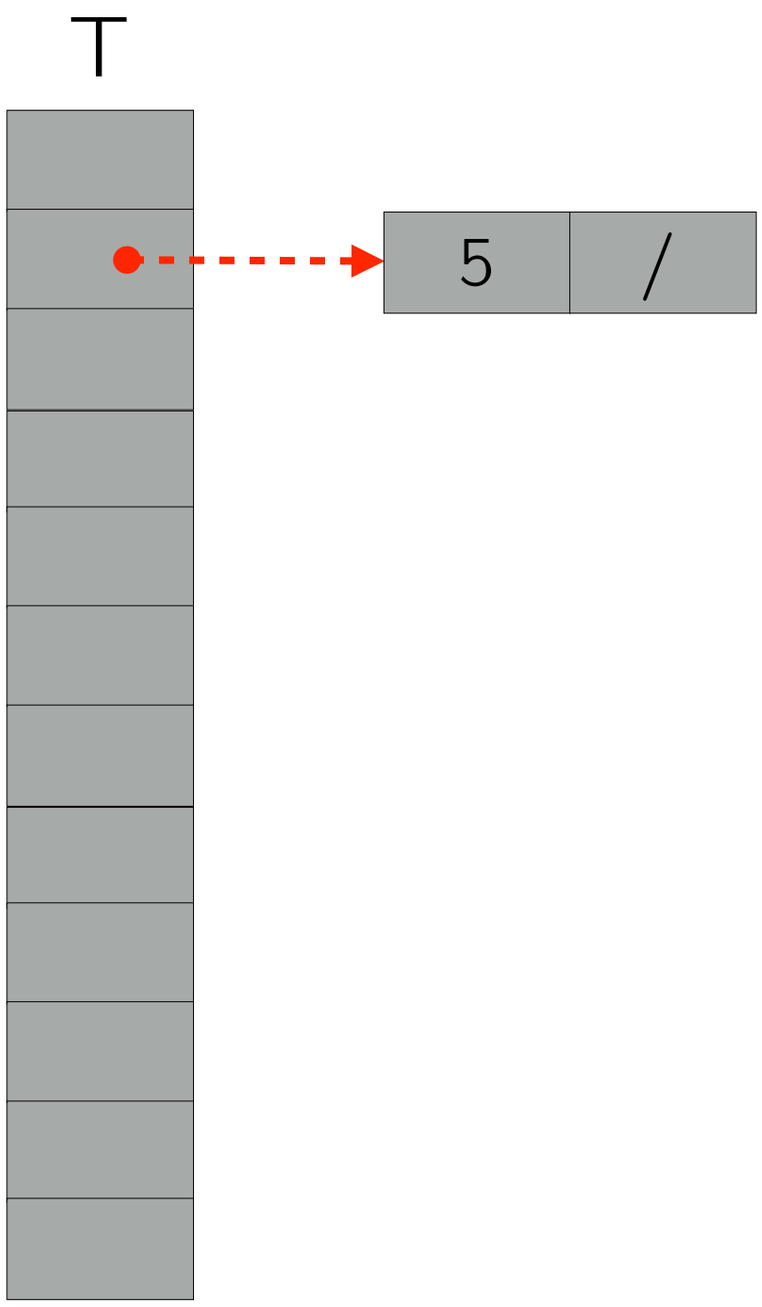


Tabelle Hash

S={5, 10, 15, 20, 22, 25, 33}

h: S$\rightarrow$|T|

h(5)=1
h(10)=h(22)=h(33)=3
h(15)=h(20)=7
h(25)=9

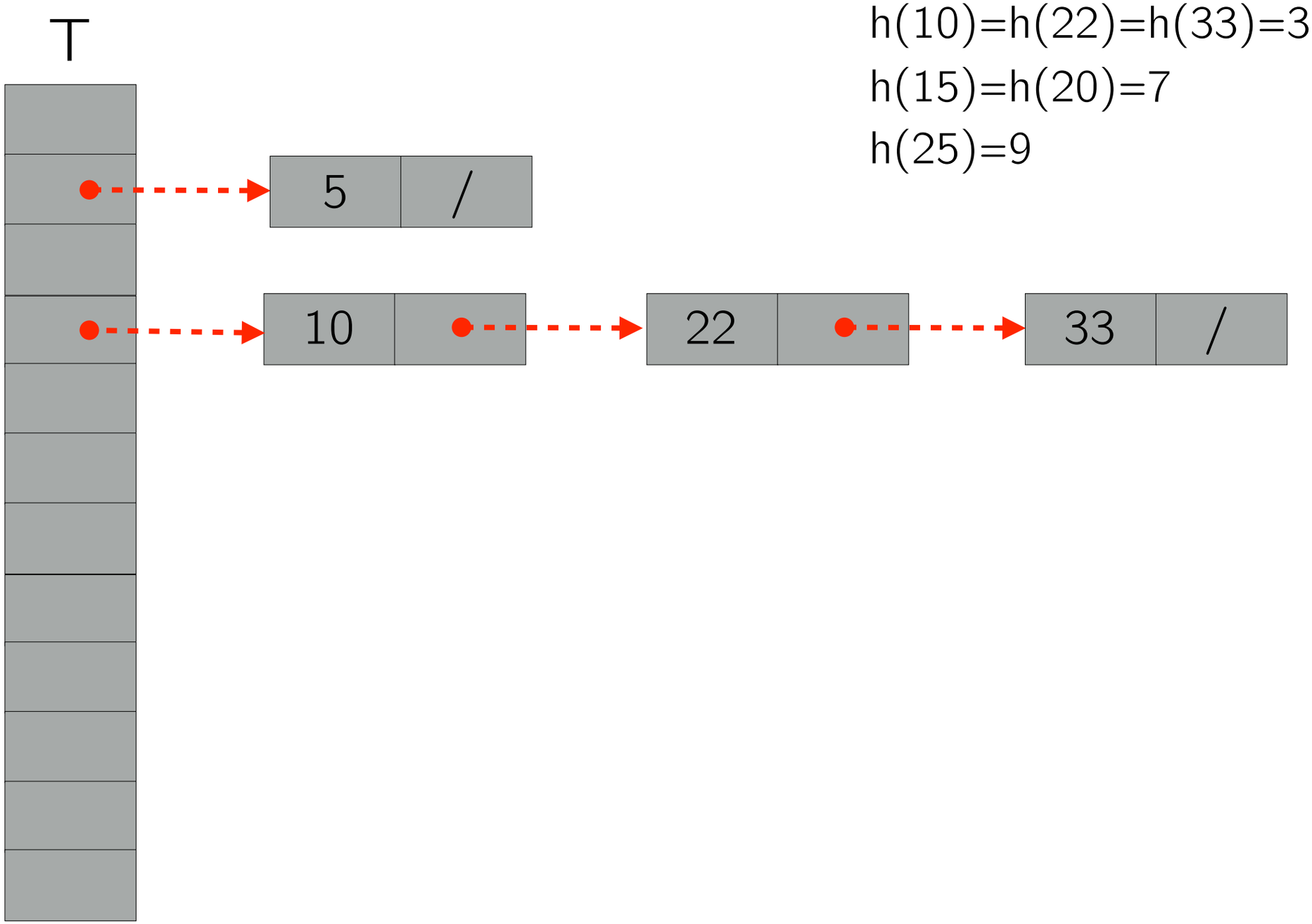


Tabelle Hash

S={5, 10, 15, 20, 22, 25, 33}

h: S$\rightarrow$|T|

h(5)=1
h(10)=h(22)=h(33)=3
h(15)=h(20)=7
h(25)=9

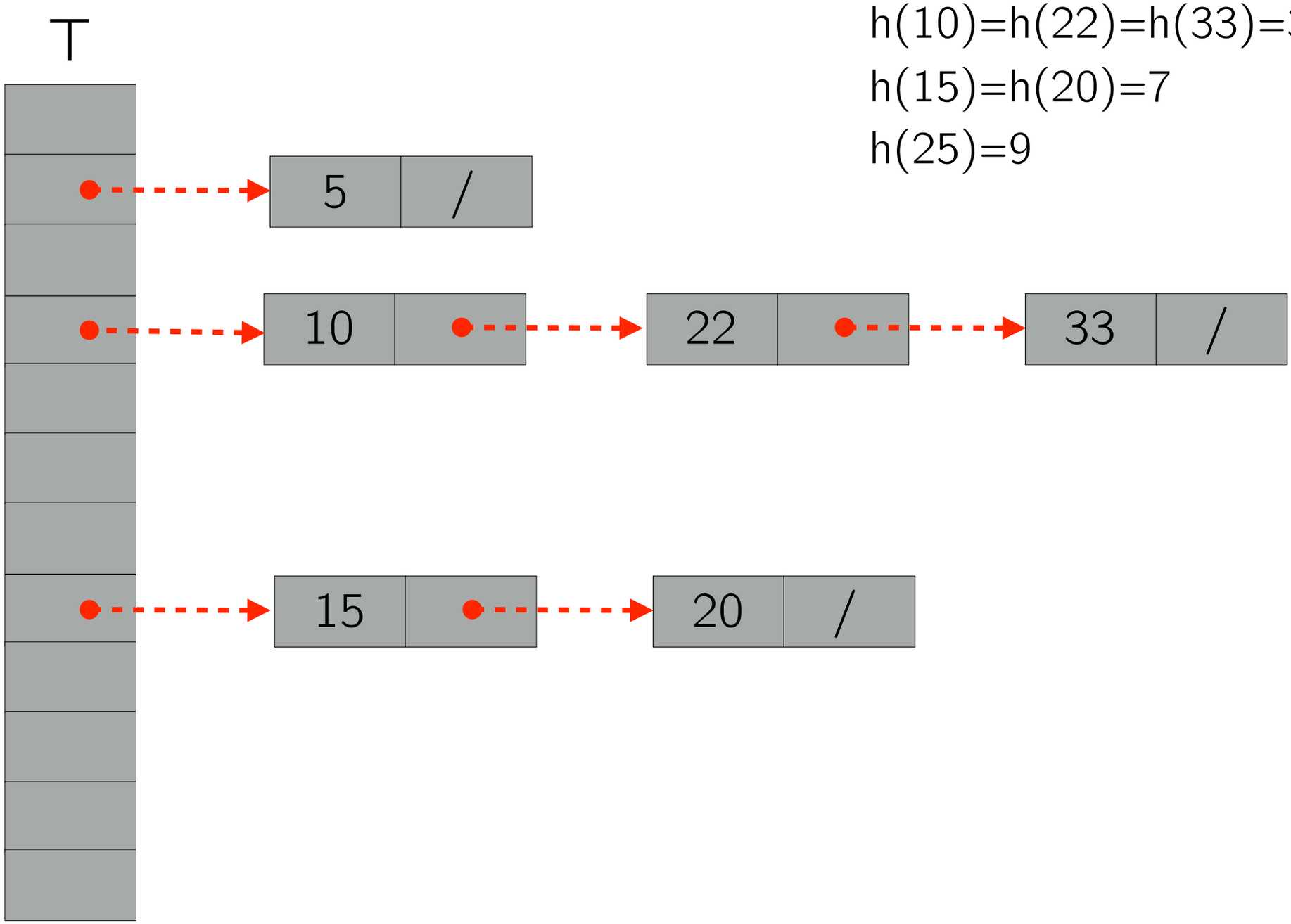


Tabelle Hash

S={5, 10, 15, 20, 22, 25, 33}

h: S$\rightarrow$|T|

h(5)=1
h(10)=h(22)=h(33)=3
h(15)=h(20)=7
h(25)=9

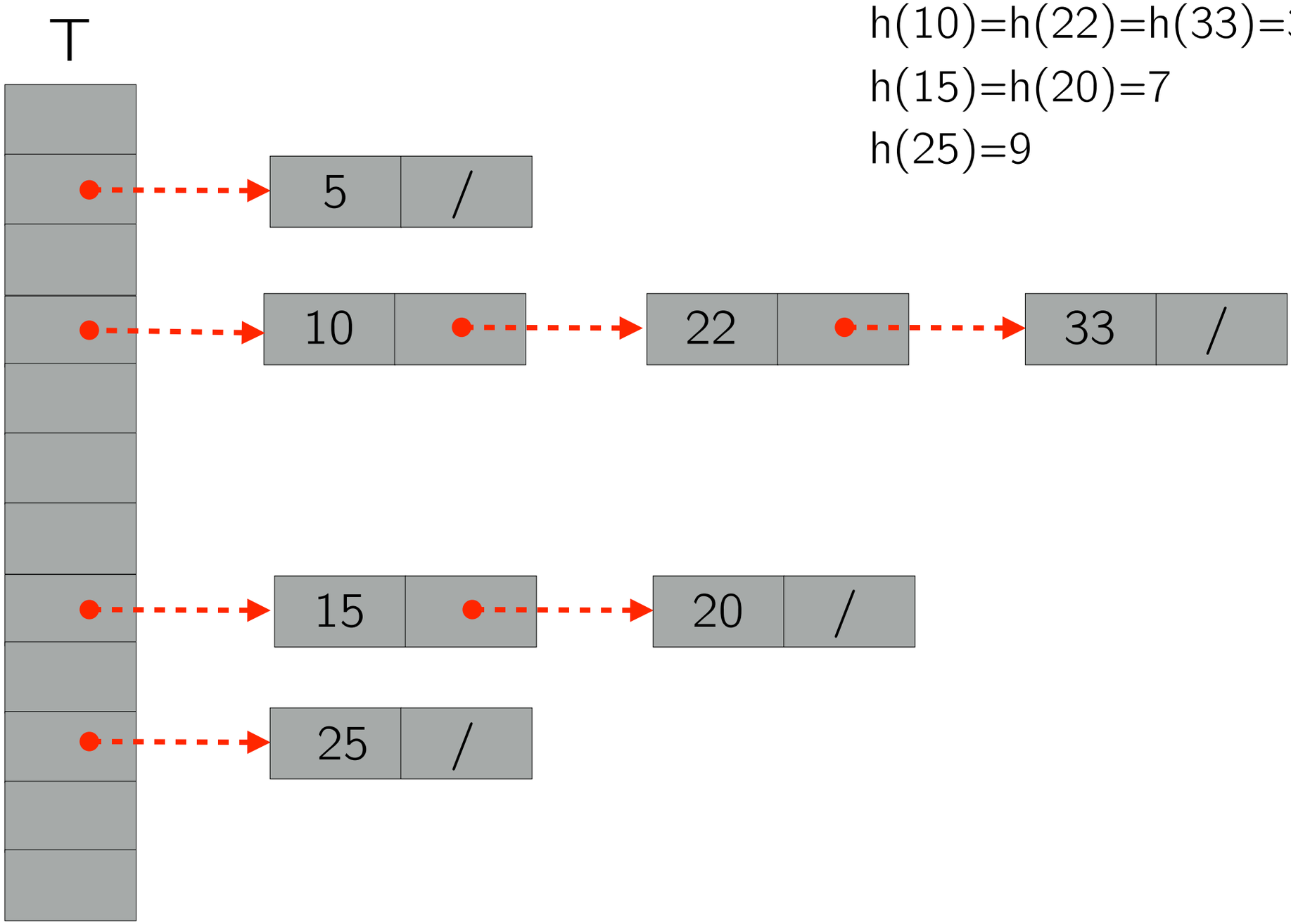
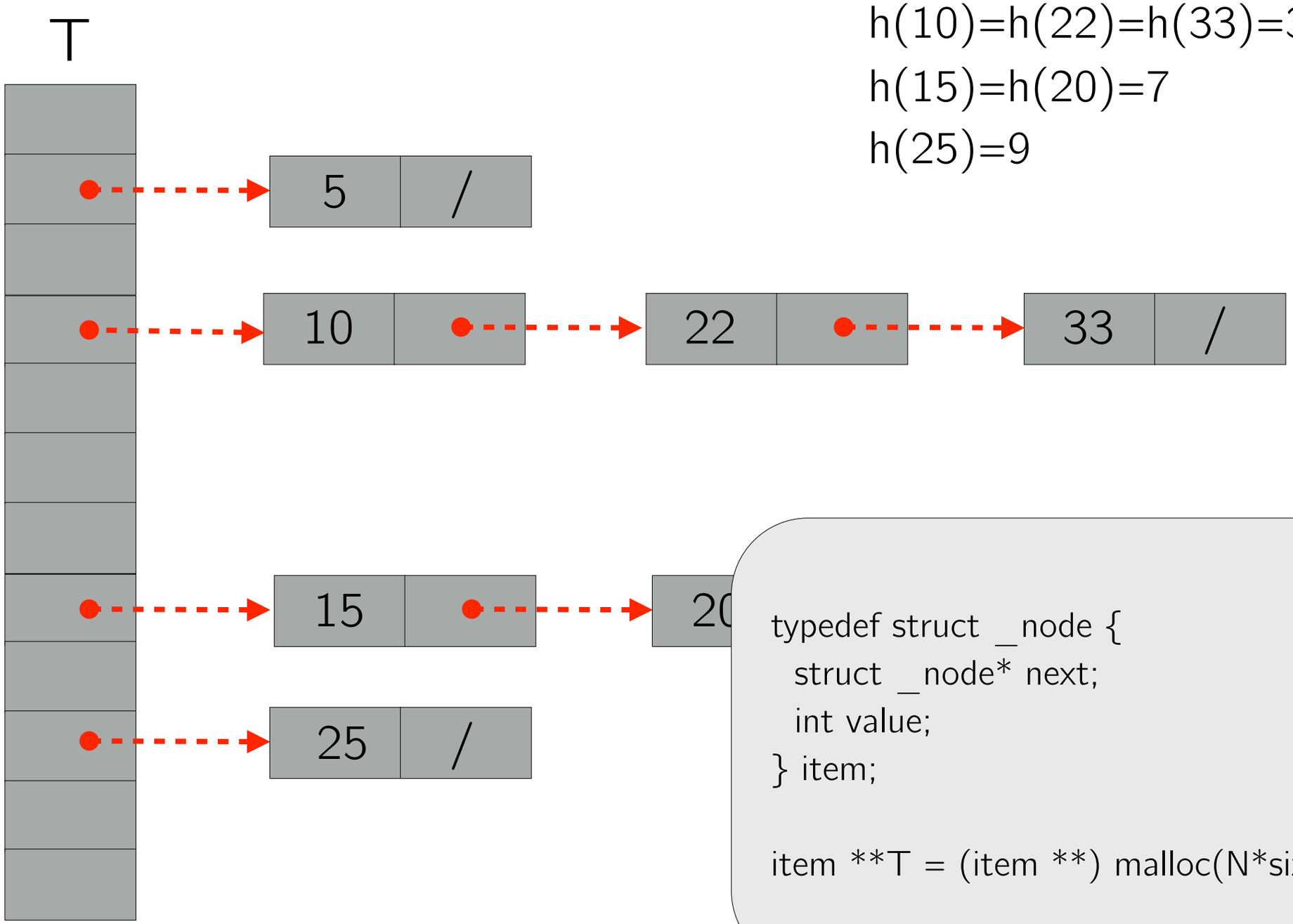


Tabelle Hash

S={5, 10, 15, 20, 22, 25, 33}

h: S$\rightarrow$|T|

h(5)=1
h(10)=h(22)=h(33)=3
h(15)=h(20)=7
h(25)=9




```
typedef struct _node {
  struct _node* next;
  int value;
} item;

item **T = (item **) malloc(N*sizeof(item *));
```

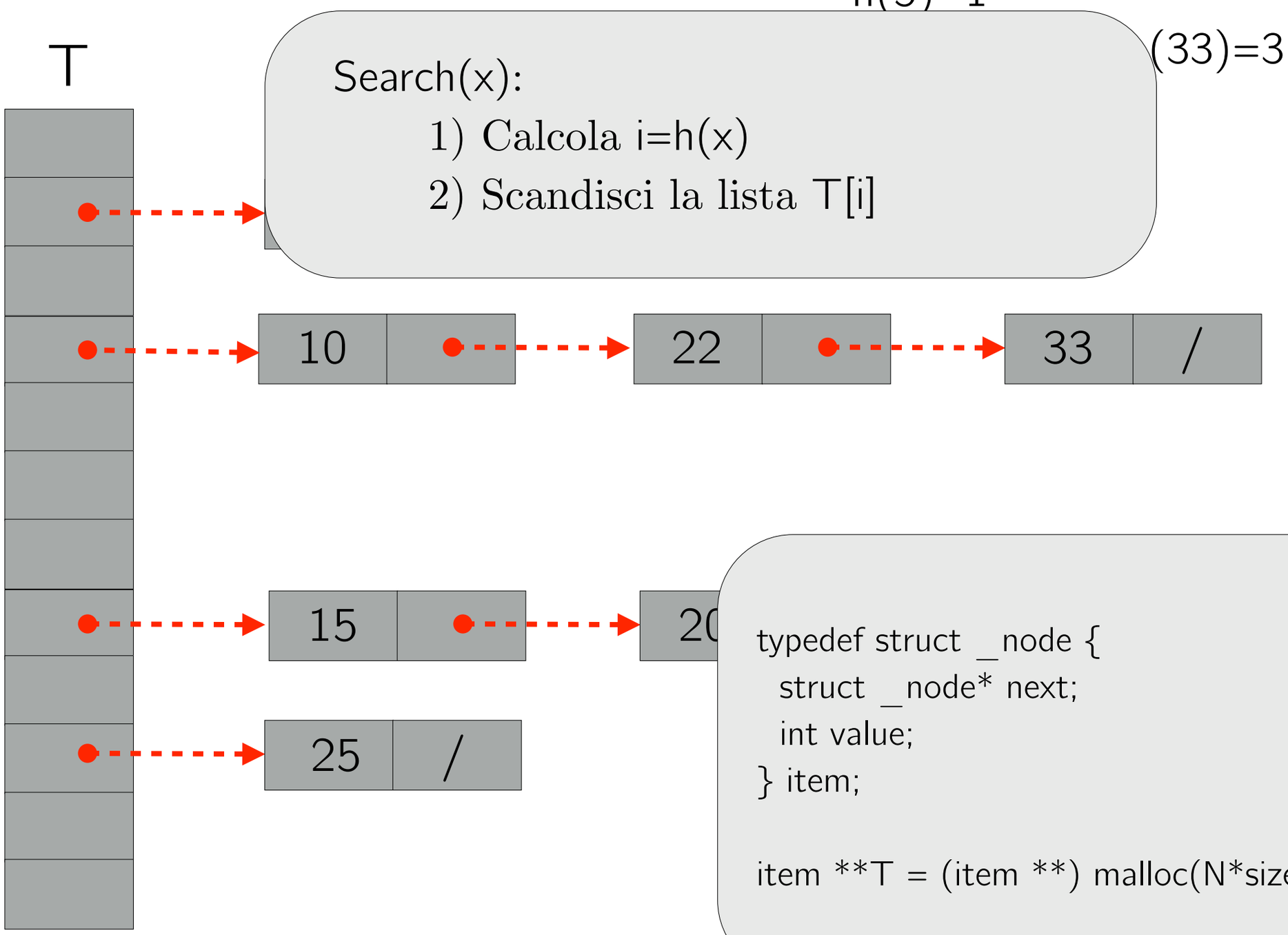
Tabelle Hash

S={5, 10, 15, 20, 22, 25, 33}

h: S→|T|

h(5)=1

(33)=3




```
typedef struct _node {
  struct _node* next;
  int value;
} item;

item **T = (item **) malloc(N*sizeof(item *));
```

Tabelle Hash

S={5, 10, 15, 20, 22, 25, 33}

h: S→|T|

h(5)=1

(33)=3

T

Search(x):

1) Calcola i=h(x)

2) Scandisci la lista T[i]

Insert(x):

1) Calcola i=h(x)

2) Inserisci x in testa/coda a T[i]

/

15

20

25

/

```
typedef struct _node {
  struct _node* next;
  int value;
} item;

item **T = (item **) malloc(N*sizeof(item *));
```

Tabelle Hash

S={5, 10, 15, 20, 22, 25, 33}

h: S→|T|

h(5)=1

(33)=3

T

Search(x):

1) Calcola i=h(x)
2) Scandisci la lista T[i]

Insert(x):

1) Calcola i=h(x)
2) Inserisci x in testa/coda a T[i]

/

Delete(x):

1) Calcola i=h(x)
2) Rimuovi x da T[i]

```
item **T = (item **) malloc(N*sizeof(item *));
```

Esercizio 1

Tabelle Hash: inserimento

Scrivere un programma che legga da tastiera una sequenza di n interi **distinti** e li inserisca in una tabella hash di dimensione $2n$ posizioni utilizzando liste monodirezionali per risolvere eventuali conflitti.

Utilizzare la funzione hash $h(x) = ((ax + b) \ \% \ p) \ \% \ 2n$ dove p è il numero primo 999149 e a e b sono interi positivi minori di 10.000 scelti casualmente.

Una volta inseriti tutti gli interi, il programma deve stampare la lunghezza massima delle liste e il numero totale di conflitti.

Prima di scrivere il programma chiedersi perché la tabella ha dimensione $2n$ e non n.

Esercizio 2

Tabelle Hash: inserimento con rimozione dei duplicati

Scrivere un programma che legga da tastiera una sequenza di n interi **NON distinti** e li inserisca senza duplicati in una tabella hash di dimensione $2n$ posizioni utilizzando liste monodirezionali per risolvere eventuali conflitti.

Utilizzare la funzione hash $h(x) = ((ax + b) \ \% \ p) \ \% \ 2n$ dove p è il numero primo 999149 e a e b sono interi positivi minori di 10.000 scelti casualmente.

Una volta inseriti tutti gli interi, il programma deve stampare il numero totale di conflitti, la lunghezza massima delle liste e il numero di elementi distinti.

Esercizio 3

Liste: cancellazione

Scrivere un programma che legga da tastiera una sequenza di n interi distinti e li inserisca in una lista monodirezionale. Successivamente il programma deve calcolare la media aritmetica dei valori della lista ed eliminare tutti gli elementi il cui valore è inferiore o uguale alla media, troncata all'intero inferiore. Ad esempio:

$$avg(1, 2, 4) = 7/3 = 2$$

IMPORTANTE: Si abbia cura di liberare la memoria dopo ogni cancellazione.

Esercizio 4

Liste: Somme suffisse

Scrivere un programma che legga da tastiera una sequenza di n interi maggiori di 0 e li inserisca in una lista nell'ordine di immissione. La lista deve essere monodirezionale.

Successivamente il programma deve sostituire il valore di ciascun elemento con la somma dei valori degli elementi che lo seguono nella lista.

Suggerimento: si utilizzi la ricorsione per ottenere la somma ad ogni passo.

Puzzle

Orco e hobbit

Un orco ha rapito N hobbit e gli propone: "Domani mattina metterò un cappello a ciascuno di voi. Ogni cappello sarà etichettato con un numero da 0 a $N-1$, doppioni sono possibili. Ogni hobbit potrà vedere il numero sui cappelli degli altri ma non il suo. Quando suonerò la campanella, tutti gli hobbit dovranno simultaneamente dire un numero (potenzialmente diverso per ogni hobbit). Se almeno un hobbit sarà in grado di indovinare il suo numero, tutti gli hobbit saranno liberati." La sera stessa gli hobbit si incontrano e individuano una strategia che è **sempre** vincente, quale?

Per $N = 2$ la strategia è relativamente facile mentre la soluzione per N arbitrario è decisamente più difficile.