Tabelle Hash

Ricerca in Array/Lista

- Dato un array o lista di n elementi, vogliamo trovarne uno (e.g., un numero x in un array di interi)

Ricerca in Array/Lista

- Dato un array o lista di n elementi, vogliamo trovarne uno (e.g., un numero x in un array di interi)
- L’unico modo che conosciamo è scorrere tutto l’array/lista, e ritornare ‘true’ se incontriamo il valore.

Ricerca in Array/Lista

- Dato un array o lista di n elementi, vogliamo trovarne uno (e.g., un numero x in un array di interi)
- L’unico modo che conosciamo è scorrere tutto l’array/lista, e ritornare ‘true’ se incontriamo il valore.
- Costo: proporzionale a n.
- Scriveremo O(n) (‘ordine di n’) per dire che il costo, ovvero numero di operazioni svolte, è pari a n per una qualsiasi costante fissa. Lo vedremo meglio più avanti.

Operazioni su Array

- Se voglio *inserire* un elemento x (senza duplicazione!), in un array devo prima controllare che non sia già presente (costo O(n)), poi inserirlo in coda (costo 'O(1)' se c'è spazio per inserire, altrimenti O(n) per fare realloc()).
- Se voglio *eliminare* un elemento x da un array, devo prima trovarlo (costo O(n)), poi cancellarlo spostando di 1 cella indietro tutti i valori successivi (costo O(n)), e decrementare la lunghezza dell'array (costo O(1)).

Operazioni su Array

- In conclusione, i costi per operare su un array come se fosse un **set** di elementi sono:

- Ricerca : O(n)
- Inserimento : O(n)
- Rimozione : O(n)

E su una lista?

Operazioni su Array

- In conclusione, i costi per operare su un array come se fosse un **set** di elementi sono:

- Ricerca : O(n)
- Inserimento : O(n)
- Rimozione : O(n)

Si può fare di meglio?
Si! Ad esempio con le **tabelle hash**

Tabelle Hash

- Le tabelle hash permettono di memorizzare dati estremamente dinamici in modo da ottenere un accesso con tempi migliori a quelli di un array o lista
- Per poter creare una tabella hash, i dati devono essere strutture con almeno un campo (detto chiave, key) in cui ogni dato è garantito avere un valore diverso:
  - Ad esempio, nella struttura **`studente_t`** il campo matricola e' un campo chiave perche' identifica univocamente lo studente e quindi la sua rappresentazione

Tabelle Hash

- Le tabelle hash permettono di memorizzare dati estremamente dinamici in modo da ottenere un accesso con tempi migliori a quelli di un array o lista
- Per poter creare una tabella hash, i dati devono essere strutture con almeno un campo (detto chiave, key) in cui ogni dato è garantito avere un valore diverso:
  - Ad esempio, nelle astruttura studente il campo matricola e' un campo chiave perche' identifica univocamente lo studente e quindi la sua rappresentazione
  - Per numeri (e.g., int o double), il numero stesso è chiave

Tabelle Hash

- Dato un tipo di dato (studente, int, double ...) con una chiave, abbiamo bisogno di una funzione di hash
- La funzione di hash mappa ogni chiave ad un intero in un intervallo [0, ..., m-1].

Tabelle Hash

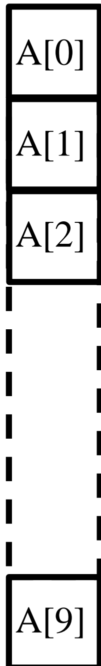
- Dato un tipo di dato (studente, int, double ...) con una chiave, abbiamo bisogno di una funzione di hash
- La funzione di hash mappa ogni chiave ad un intero in un intervallo [0, ..., m-1].

- Esempio funzione hash per chiavi intere: ‘modulo 10’

- Usiamo questi valori come indici in un array di m elementi

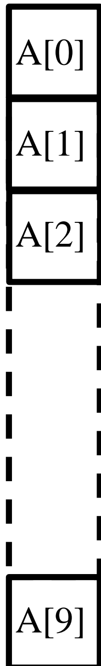
Esempio: tabella hash per interi

Funzione di hash: ‘modulo m’
Creo un array di m elementi
(in questo caso 10)



Esempio: tabella hash per interi

Funzione di hash: ‘modulo m’
Creo un array di m elementi
(in questo caso 10)

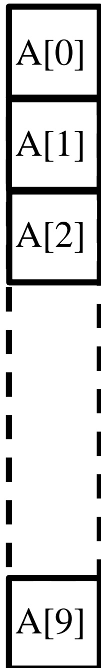


Voglio inserire 522

522

Esempio: tabella hash per interi

Funzione di hash: ‘modulo m’
Creo un array di m elementi
(in questo caso 10)



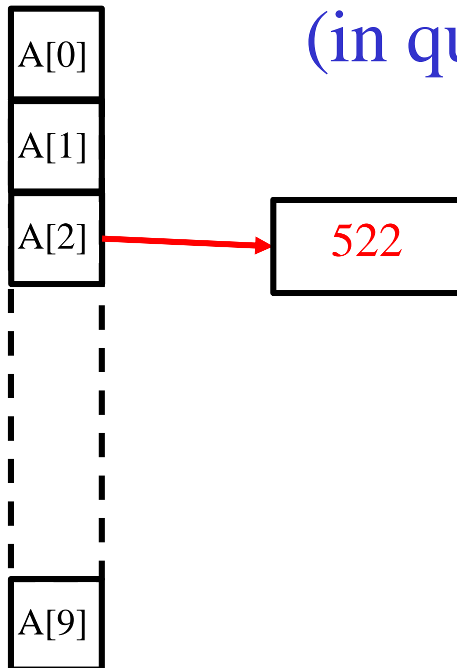
Voglio inserire 522

522

522 % 10 = 2

Esempio: tabella hash per interi

Funzione di hash: 'modulo m'
Creo un array di m elementi
(in questo caso 10)

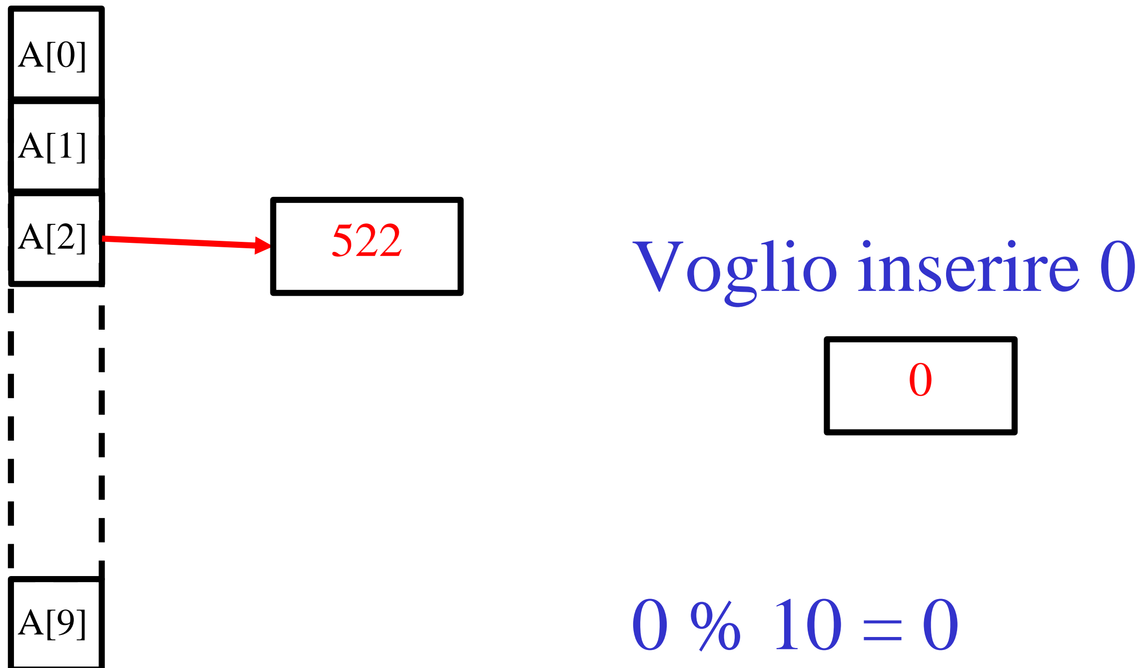


Voglio inserire 522

522 % 10 = 2

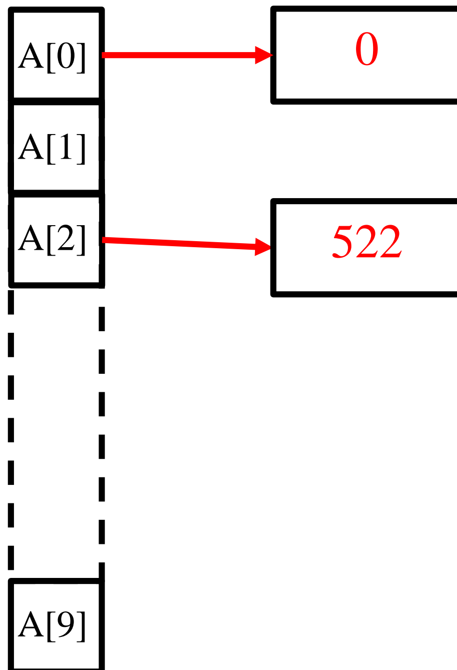
Esempio: tabella hash per interi

Funzione di hash: ‘modulo 10’



Esempio: tabella hash per interi

Funzione di hash: ‘modulo 10’

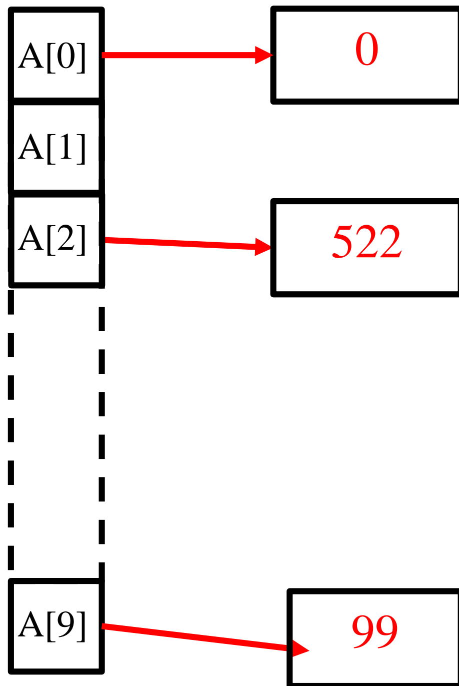


Voglio inserire 0

0 % 10 = 0

Esempio: tabella hash per interi

Funzione di hash: ‘modulo 10’



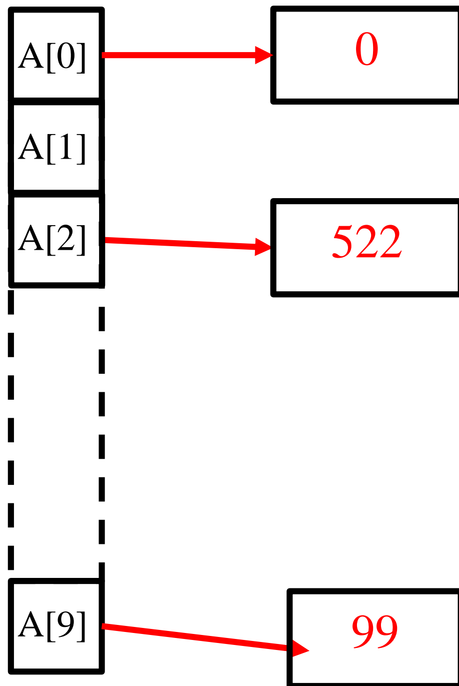
Costo di cercare x?

Costo di inserire x?

Costo di rimuovere x?

Esempio: tabella hash per interi

Funzione di hash: ‘modulo 10’



Costo di cercare x? O(1)

calcolo l’hash della chiave e controllo che il valore nella cella sia x

Costo di inserire x? O(1)

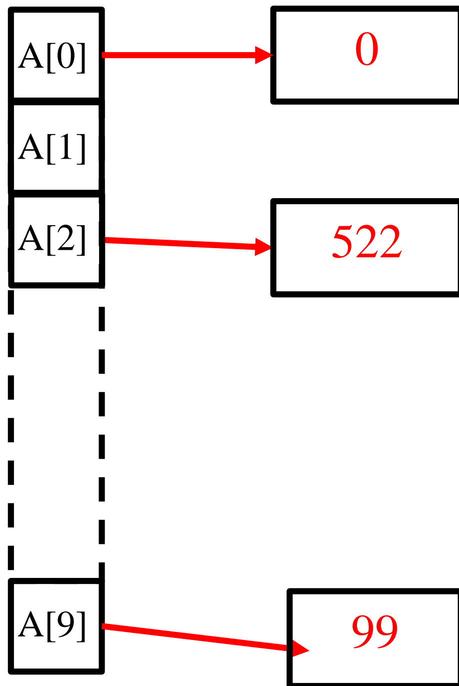
calcolo l’hash della chiave e inserisco nella cella

Costo di rimuovere x? O(1)

calcolo l’hash della chiave e se x è nella cella lo rimuovo

Esempio: tabella hash per interi

Funzione di hash: ‘modulo 10’



Problema: e se voglio inserire 22?

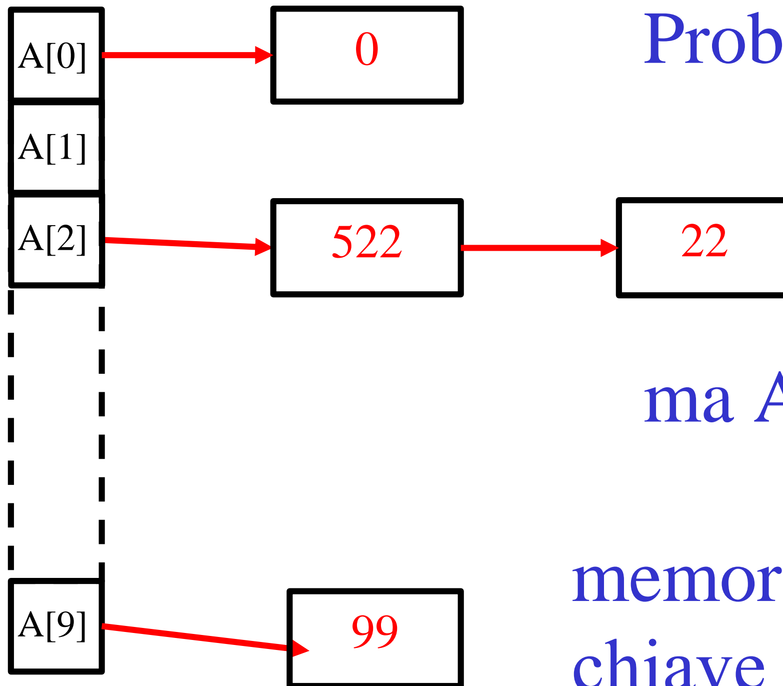
22

22 % 10 = 2

ma A[2] contiene già un valore!

Liste di trabocco

Funzione di hash: ‘modulo 10’



Problema: e se voglio inserire 22?

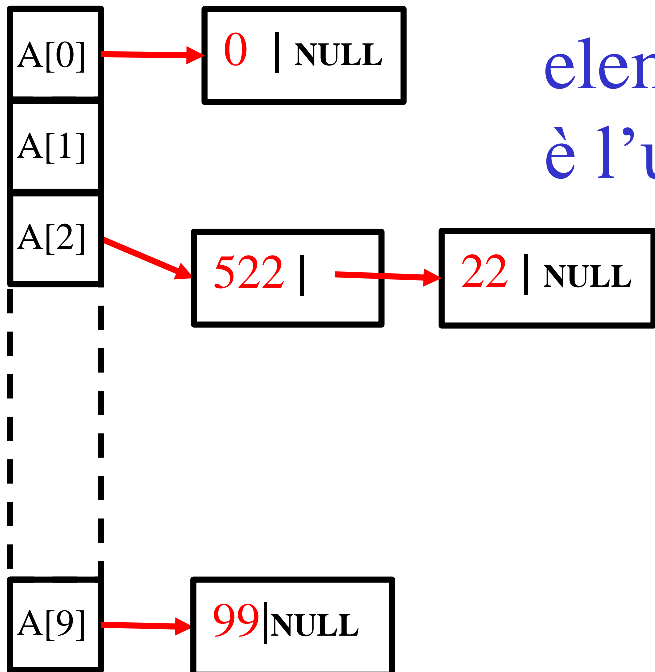
22 % 10 = 2

ma A[2] contiene già un valore!

memorizziamo i valori con stessa chiave in ‘liste di trabocco’

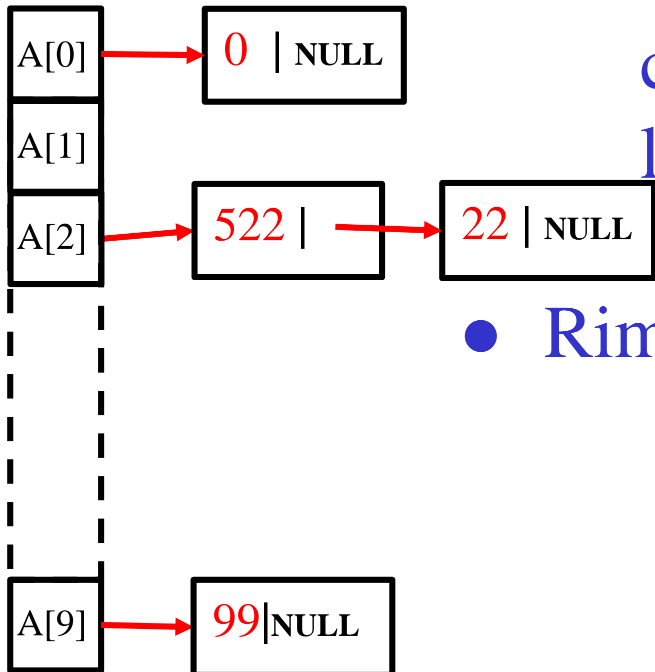
Liste di trabocco

- A[i] conterrà una lista (i.e., puntatore al primo elemento della lista)
- Se la lista è vuota, A[i] = **NULL**
- Ogni elemento della lista contiene un valore, ed il puntatore 'next' al prossimo elemento (che vale **NULL** se l'elemento è l'ultimo)



Operazioni su Tabelle Hash

- Ricerca di x:
  calcolo l’hash della chiave, scorro la lista A[i] relativa e ritorno ‘true’ se trovo x
- Inserimento di x:
  calcolo l’hash, inserisco x nella lista relativa (in testa o coda?)
- Rimozione di x:
  calcolo l’hash, scorro la lista relativa, se presente rimuovo x dalla lista



Operazioni su Tabelle Hash

- Ricerca di x: O(???)

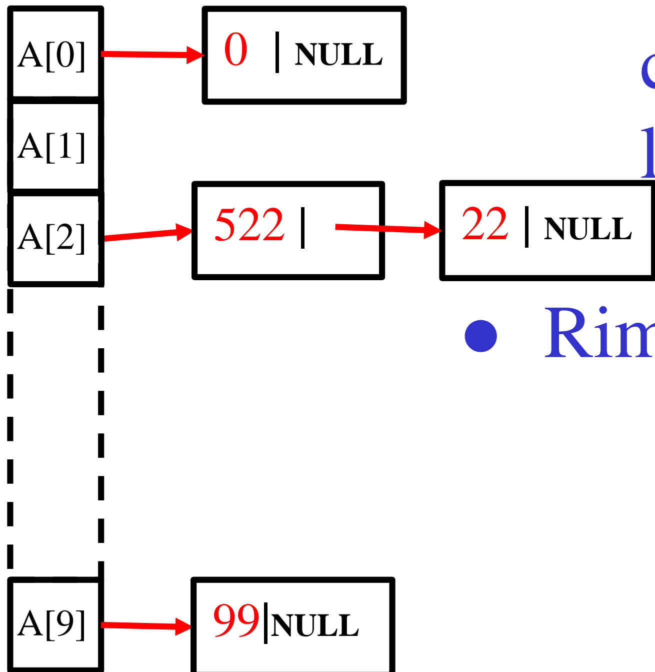
  calcolo l'hash della chiave, scorro la lista A[i] relativa e ritorno 'true' se trovo x

- Inserimento di x: O(1)

  calcolo l'hash, inserisco x nella lista relativa (in testa o coda?)

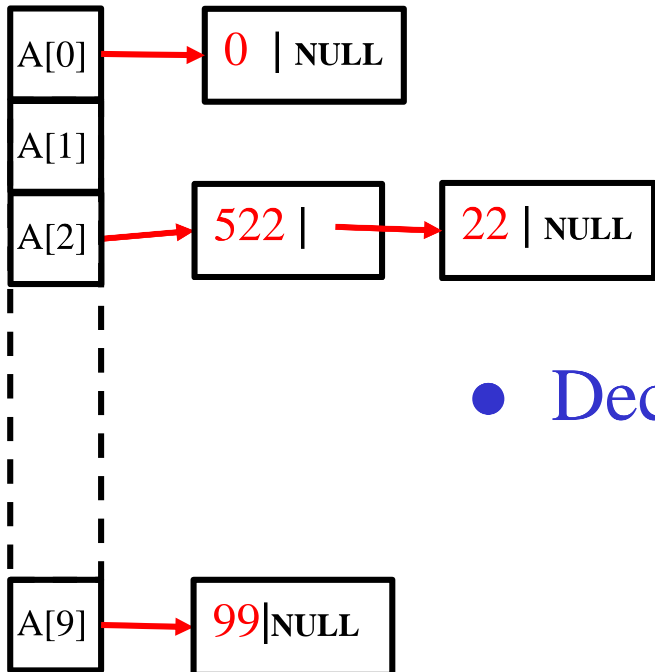
- Rimozione di x: O(???)

  calcolo l'hash, scorro la lista relativa, se presente rimuovo x dalla lista



Operazioni su Tabelle Hash

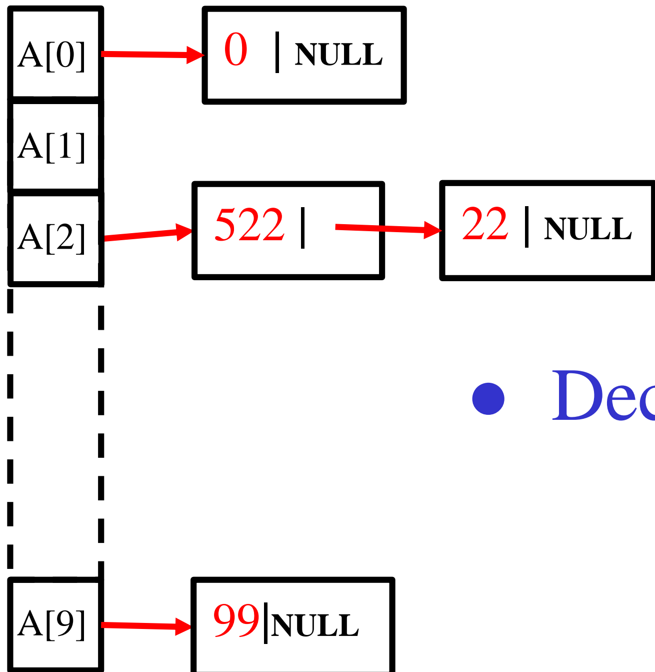
- Il costo dipende dalla lunghezza delle liste!
- In pratica, se ci sono pochi elementi nella tabella (rispetto a m), ci aspettiamo lunghezza costante quindi:
  - Ricerca di x: O(1)
  - Inserimento di x: O(1)
  - Rimozione di x: O(1)



- Decisamente meno di array e liste!

Operazioni su Tabelle Hash

- Il costo dipende dalla lunghezza delle liste!
- In pratica, se ci sono pochi elementi nella tabella (rispetto a m), ci aspettiamo lunghezza costante quindi:
  - Ricerca di x: O(1)
  - Inserimento di x: O(1)
  - Rimozione di x: O(1)



- Decisamente meno di array e liste!

Tabella Hash in C

- Come si realizza una tabella hash in C ?
  - Conosciamo il tipo di dato ‘tipo_t’, la funzione di hash, e m
  - La tabella T è un array di lunghezza m.
  - Ogni T[i] contiene una lista di ‘tipo_t’ (i.e., puntatore al primo elemento), inizialmente vuota (NULL).
  - Ogni lista è composta da nodi, con un ‘valore’ (tipo_t) e un ‘next’ (puntatore a nodo). Ad es se ‘tipo_t’ è ‘int’:

```
typedef struct lista {
            int val;
      struct lista * next;
    } lista_t;
```

Tabella Hash in C

- Inizializziamo la tabella di hash T:

```
typedef struct lista {
        int val;
        struct lista * next;
     } lista_t;

        lista_t ** T = malloc(sizeof(lista_t *)*m);

        for(int i=0; i<m; i++) T[i] = NULL;
```

Funzione Hash in C

- Per comodità e pulizia possiamo dichiarare la funzione hash separatamente (così non dobbiamo copiarla ogni volta)

```
/* restituisce un valore in [0,...,m-1] */
int hash(int val, int m){
        return val % m;
    }
/* un po' meno banale, con 'A' e 'B'
costanti globali*/
int hash(int val, int m){
        return (val*A + B) % m;
    }
```

Stampa Tabella Hash

- Stampiamo tutti gli elementi in T:

```
typedef struct lista {
                int val;
                struct lista * next;
    } lista_t;
        lista_t ** T = malloc(sizeof(lista_t *)*m);
        for(int i=0; i<m; i++) T[i] = NULL;

... //uso la tabella

for(int i=0; i<m; i++)
   if( T[i] != NULL ) stampa_lista(T[i]);
```

Esercizio

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 $m=2n$ utilizzando liste di trabocco per risolvere conflitti.

Utilizzare la funzione hash '$h(x) = ((ax + b) \% p) \% m$' 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, stampare (i) gli elementi nella tabella, (ii) il numero totale di conflitti, (iii) la lunghezza massima delle liste e (iv) il numero di elementi distinti nella tabella.

Provare lo stesso con funzione hash '$h(x) = x \% m$'
e osservare se c'è differenza nella performance.

Algoritmi di ordinamento

Il problema

- Vogliamo ordinare un array monodimensionale in modo crescente
  - per il caso decrescente valgono le stesse considerazioni
- Vari algoritmi possibili
  - Diverse caratteristiche di complessità computazionale

Un primo algoritmo: selection sort

- Abbiamo un array a di lunghezza N
  - Al primo passo scegliamo il valore minimo fra tutti quelli presenti nell'array e lo spostiamo nella posizione 0
  - Al passo 1 scegliamo il minimo fra tutti i valori in posizione $1, 2, N-1$ e lo scambiamo con il valore in posizione 1
  - Al passo i scegliamo il minimo fra tutte le posizioni

$$i,\ i+1,\ \ldots, N-1$$

e lo scambiamo con il valore in posizione i

Selection sort: esempio

```
5 2 4 6 1 3
1 2 4 6 5 3
1 2 4 6 5 3
1 2 3 6 5 4
1 2 3 4 5 6
1 2 3 4 5 6
1 2 3 4 5 6
```

Selection sort: codifica

```
/* calcola la posizione del minimo elemento di
v nella porzione [from,to] */
int minPos(int * v, int from, int to);
/* scambia il valore delle variabili puntate da p e q */
void swap(int *p, int *q);
void selectionSort(int * v, int dim) {
  int i, min;
  for(i=0; i<dim-1; i++){
    min = minPos(v, i, dim-1);
    swap(v+i, v+min);
  }
}
/* swap, minPos le implementiamo in laboratorio */
```

Algoritmo Quicksort

- Utilizza uno schema di soluzione *divide et impera*
  - *Decomposizione*: Si divide il problema in piu' sottoproblemi
  - *Ricorsione*: Si risolvono i sottoproblemi
  - *Combinazione*: Si combina la soluzione dei sottoproblemi nella soluzione complessiva del problema di partenza

Algoritmo Quicksort

- Per l’ordinamento di un array
  - *Decomposizione*: Si sceglie un elemento nell'array da ordinare (il pivot)
  - *Ricorsione*: Si crea l'array a degli elementi minori del pivot e l'array b degli elementi maggiori del pivot, e si ordinano ricorsivamente
  - *Combinazione*: Si si combinano gli array ottenuti nell'array complessivo ordinato

Quicksort esempio

Esempio array da ordinare :

2 8 7 1 3 5 6 4

Quicksort esempio

`Esempio array da ordinare :`

`2 8 7 1 3 5 6 4`

Pivot

Quicksort esempio

`Esempio array da ordinare :`

```
2    8    7    1    3    5    6    4
2    8    7    1    3    5    6    4
2    8    7    1    3    5    6    4
```

Array dei valori maggiori del pivot

Array dei valori minori del pivot

Quicksort esempio

Esempio array da ordinare :

```
2   8   7   1   3   5   6   4
2   8   7   1   3   5   6   4
2   8   7   1   3   5   6   4
```

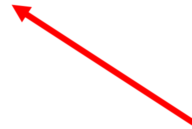
Array dei valori maggiori del pivot

Array dei valori minori del pivot

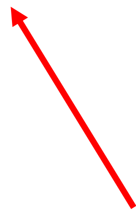
Quicksort esempio

Esempio array da ordinare :

```
2   8   7   1   3   5   6   4
2   8   7   1   3   5   6   4
2   1   7   8   3   5   6   4
```



Array dei valori maggiori del pivot

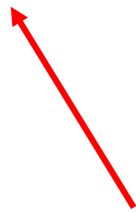


Array dei valori minori del pivot

Quicksort esempio

`Esempio array da ordinare :`

```
2   8   7   1   3   5   6   4
2   8   7   1   3   5   6   4
2   1   3   8   7   5   6   4
```



Array dei valori maggiori del pivot

Array dei valori minori del pivot

Quicksort esempio

Esempio array da ordinare :

```
        2   8   7   1   3   5   6   4
        2   8   7   1   3   5   6   4
        2   1   3   8   7   5   6   4
```

Array dei valori maggiori del pivot

Array dei valori minori del pivot

Quicksort esempio

`Esempio array da ordinare :`

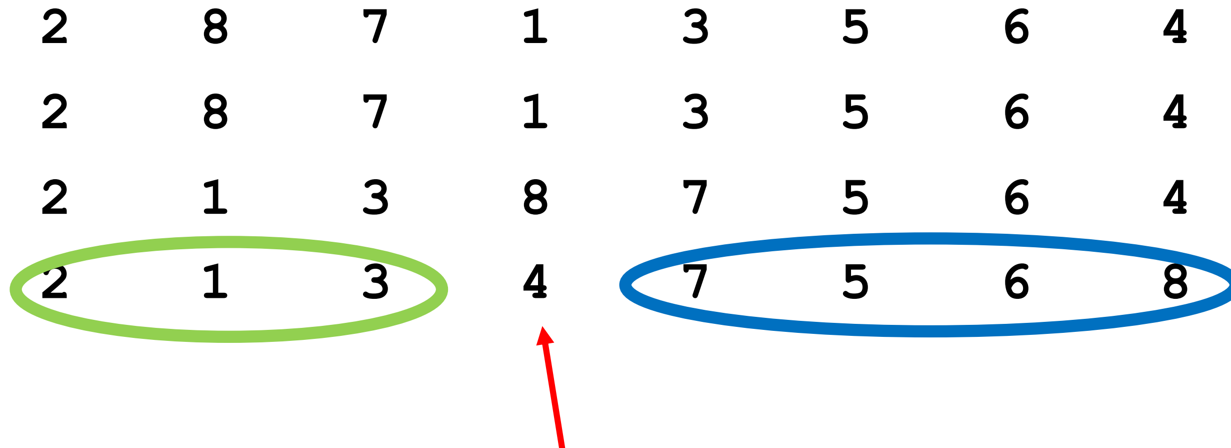
```
2   8   7   1   3   5   6   4
2   8   7   1   3   5   6   4
2   1   3   8   7   5   6   4
```

Array dei valori maggiori del pivot

Array dei valori minori del pivot

Quicksort esempio

Esempio array da ordinare :



Sposto il pivot scambiandolo con il primo dei valori maggiori

Quicksort esempio

Esempio array da ordinare :

Ordinati attraverso le chiamate ricorsive

```
/* struttura di quickSort (lo implementeremo in laboratorio */

void quickSort(int* a, int from, int to)
   int pivot; /* la posizione dell'array che farà da pivot */
   int perno;
   if ( from >= to ) return; /*caso base */
   /* scelta a caso fra gli estremi */
   pivot = random_choice(from,to);
   /* separa gli elementi minori di a[pivot] da quelli maggiori
      o uguali. perno è la posizione del pivot alla fine */
   perno = distribuisci(a,from,to,pivot);
   /* ordinano ricorsivamente i minori e i maggiori */
   quickSort(a,from, perno-1);
   quickSort(a, perno + 1, to);
} /* fine quickSort */
```

ESERCIZIO

Scrivere una funzione int* random_array(int n) che restituisce un array contenente n elementi casuali (allocato sullo heap!! altrimenti viene deallocato!)

Ordinare l'array ottenuto con selection sort, oppure quicksort.

Osservare le differenze di performance tra i due algoritmi al crescere di n (e.g., n > 1000000)

ESERCIZIO BONUS

Dato un array di interi ordinato, come posso verificare se contiene un intero x?

Devo davvero leggere ogni valore, o esiste un modo più veloce?

BONUS
Altri algoritmi famosi per il sort

Insertion Sort

- Assumiamo che la sequenza da ordinare sia partizionata in una sottosequenza già ordinata, all'inizio composta da un solo elemento, e una ancora da ordinare.
- Alla k-esima iterazione, la sequenza già ordinata contiene k elementi.
- Alla k-esima iterazione, viene rimosso un elemento dalla sottosequenza non ordinata e inserito nella posizione corretta della sottosequenza ordinata, estendendola così di un elemento.

Insertion Sort: esempio

```
3 7 4 9 5 2 6 1
3 7 4 9 5 2 6 1
3 7 4 9 5 2 6 1
3 4 7 9 5 2 6 1
3 4 7 9 5 2 6 1
3 4 5 7 9 2 6 1
2 3 4 5 7 9 6 1
2 3 4 5 6 7 9 1
1 2 3 4 5 6 7 9
```

Insertion Sort: Codifica

```
void insertionSort(int * A, int len) {
int i, j;
int key;
for(i = 1; i < len; i++) {
      key = A[i];
      j = i - 1;
      while (( j >= 0 ) && (A[j]> key)) {
            A[j+1] = A[j];
            j--;
      }
      A[j+1] = key;
   }
}
```

Algoritmo Merge-Sort

- Anche qua *divide et impera*
  - *Decomposizione*: Si divide l'array in due parti di lunghezza simile
  - *Ricorsione*: Si ordinano le due parti
  - *Combinazione*: Si combinano i due array ordinati

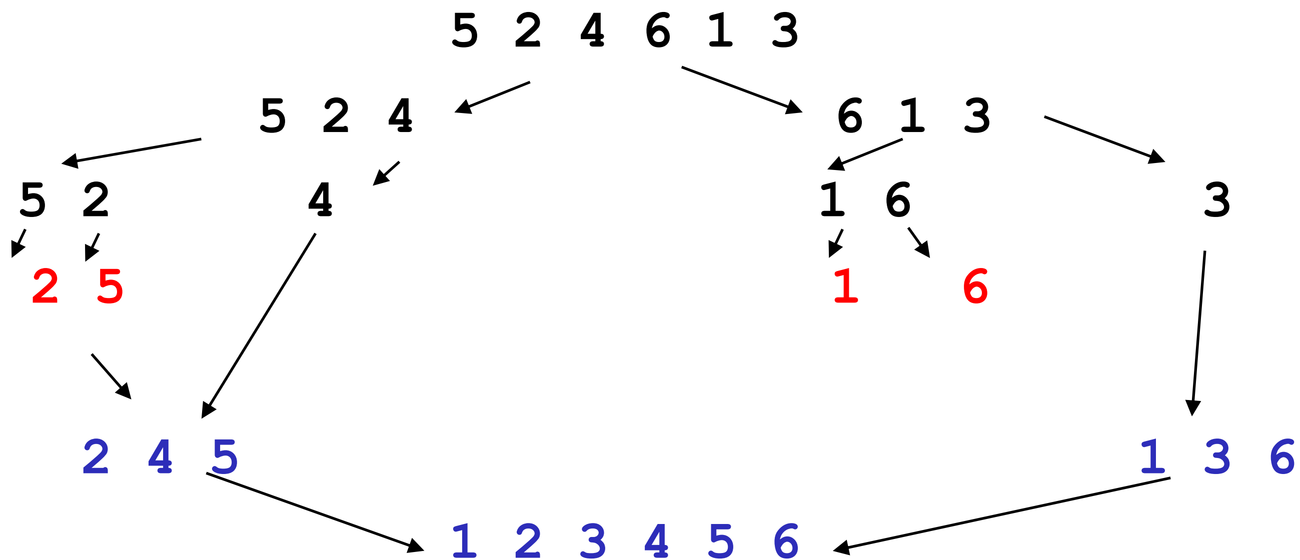
Merge sort: codifica

```
void mergeRicorsivo(int * a, int from, int to) {
  int mid;
  if (from < to) { /* l'intervallo da mid a to, estremi
    inclusi, comprende almeno due elementi */
    mid = (from + to) / 2;
    mergeRicorsivo(a, from, mid);
    mergeRicorsivo(a, mid+1, to);
    merge(a, from, mid, to); /* fonde le due porzioni
   ordinate [from, mid],[mid+1, to] nel sottovettore
   [from, to] */
  }
}
```

Merge sort: codifica

```
void sort(int * v, int dim) {
    mergeRicorsivo(v, 0, dim-1);
 }
```

Esempio:



```
void merge(int* a, int from, int mid, int to) {
  int b[LUNG]; /* vettore di appoggio */
  int i = from, j = mid + 1; /* scorrono la prima e la seconda
   parte di a */
  int k= from; /* scorre b */
/* copio il minore dalla prima o dalla seconda porzione */
while (i <= mid && j <= to) {
  if (a[i] <= a[j]) {
      b[k] = a[i];
      i++; /* copia dalla prima porzione ed avanza l'indice */
   }
   else {
   b[k] = a[j] ;
   j++; /* copia dalla seconda porzione ed avanza l'indice */
   }
  k++;
 } /* fine while */              /* ... Segue ... */
```

```
void merge(int* a, int from, int mid, int to) {
   .....
   /* controllo quale porzione è finita prima e copio quello
  che rimane dell'altra nel vettore di appoggio */
  if ( i > mid )
    for ( ; j <= to ; j++, k++ )
       b[k] = a[j] ;
  else
    for ( ; i <= mid ; i++, k++ )
       b[k] = a[i] ;

  /* ricopia tutti gli elementi ordinati da b ad a */
  for ( k = from ; k <= to ; k++ )
       a[k] = b[k] ;
} /* fine merge */
```

```
/* un possibile main  */
#define LUNG 10

int main (void) {
   int a[LUNG];
   leggi(a, LUNG);
   mergeSort(a, LUNG);
   stampa(a,LUNG);
   return 0;
} /* fine main */
```