SOMMARIO

- Funzioni hash e tabelle hash:
  - Strutture dati e operazioni di ricerca.
  - Funzioni hash:
    - Chiavi numeriche.
    - Chiavi non numeriche: approccio polinomiale.
  - Collisioni:
    - Distribuzione uniforme: numeri primi.
    - Concatenazioni separate.
- Applicazioni:
  - Correttore ortografico: `Hashtable`.
  - Funzioni hash: MD5 e SHA.

STRUTTURE DATI E OPERAZIONI DI RICERCA

- Si ricorda che la ricerca in un *array ordinato* ha un costo *O(log n)*, ma un inserimento/estrazione ha un costo lineare *O(n)*.

- Le operazioni su *alberi binari di ricerca* hanno costo medio *O(log n)*. Gli alberi sono strutture dati dinamiche: non è necessario sapere il numero di oggetti da inserire. Inoltre mantengono i dati ordinati.

STRUTTURE DATI E OPERAZIONI DI RICERCA

- Una *tabella hash* è una struttura dati che permette operazioni di ricerca e inserimento molto veloci: in pratica si ha un costo computazionale costante *O(1)*.
- Non è richiesto che gli oggetti memorizzati implementino le interfacce `IComparable` e `IComparer`, questo perché *una tabella hash non è basata sull'operazione di confronto*.
- Una tabella hash *non* mantiene ordinati i dati inseriti.

STRUTTURE DATI E OPERAZIONI DI RICERCA

- La realizzazione di una tabella hash è basata sull'uso di array.
- Uno svantaggio è il costo da pagare per *ridimensionare* la tabella.
- Un ulteriore svantaggio è che non c'è un modo conveniente per visitare gli elementi secondo un certo *ordine*, pertanto non è adatta a situazioni in cui sia importante un *ordinamento* dei dati.
- Tuttavia l'efficienza è tale che esempi di applicazioni includono la *tabella dei simboli* di un compilatore e la ricerca in un *dizionario* di parole chiave.

TABELLE HASH

- Si vogliono memorizzare degli oggetti in una tabella di dimensione `n` (con posizioni da `0` a `n-1`).
- Si memorizzano gli oggetti assegnando ad ogni oggetto un numero intero `i`, detto *codice hash*, compreso tra `0` e `n-1` e inserendo tale oggetto nell'i-esima posizione della tabella.
- Per la ricerca di un oggetto si calcola il suo codice hash e si accede alla corrispondente posizione nella tabella.

TABELLE HASH: FUNZIONE HASH

- La funzione per calcolare il valore hash di un oggetto, detta *funzione hash*, deve soddisfare le seguenti proprietà:
  - Essere semplice, per non influenzare il costo computazionale delle operazioni.
  - Trasformare (*mappare*, *to map*) oggetti uguali in numeri uguali. Il senso di uguale è quello del metodo `Equals()`.
- Una funzione hash può essere definita come *HF: K → [0, n-1]*. *HF* trasforma il *dominio* delle chiavi *K* nel *codominio* descritto dall'intervallo di interi da *0* a *n-1*.

FUNZIONE HASH: ESEMPIO

- Si supponga di avere un certo numero di impiegati, per esempio 100, ognuno ha un proprio numero identificativo `idnum` compreso nell'intervallo [0,99].
- Si può pensare di accedere alle informazioni di un impiegato per mezzo della chiave `idnum`. Se si memorizzano i dati in un array di 100 elementi, allora si può accedere alle informazioni di un impiegato direttamente attraverso l'indice dell'array.
- Quindi si ha un costo *O(1)*.

FUNZIONE HASH: ESEMPIO

- Tuttavia in pratica non è facile avere o mantenere una relazione perfetta tra il valore delle chiavi e gli indici di un array.
- Se l'intervallo del numero identificativo fosse più ampio, per esempio [0,99999], non sarebbe conveniente usare un array di 100000 elementi di cui solo 100 sono necessari.
- È necessario utilizzare un array della corretta dimensione e utilizzare solo due cifre della chiave per identificare un impiegato: per esempio l'impiegato `21374` è in `array[74]` e l'impiegato `32821` è in `array[21]`.

FUNZIONE HASH: ESEMPIO

- In tal modo gli elementi *non* risultano ordinati secondo il valore della chiave (l'impiegato 21374 dovrebbe precedere l'impiegato 32821), ma secondo una *certa funzione del valore della chiave*.

- Tale funzione è detta funzione hash (*hash function*).
- La tecnica di ricerca è detta *hashing*.
- La struttura dati è detta tabella hash (*hash table*).

FUNZIONE HASH: ESEMPIO

- Nell'esempio la funzione hash è `%` (*modulus operator*) :

  HF(key) → *key % n*

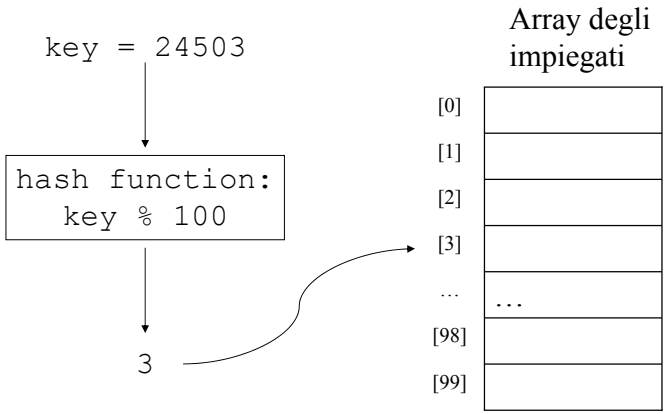
- A cui corrisponde il seguente codice:

```
hashValue = key % 100;
```

- La chiave è divisa per la dimensione della tabella e il resto della divisione è utilizzato come indice nell'array.

FUNZIONE HASH: ESEMPIO

- L'estrazione e l'inserimento dei dati sono implementati come semplici operazioni su array attraverso indici. Rappresentazione grafica.



ARRAY ASSOCIATIVO

- Le tabelle hash sono utili per implementare una struttura dati chiamata *array associativo*, i cui elementi sono indirizzati utilizzando le *chiavi* piuttosto che gli indici numerici [0, n-1].

- Pertanto, se la chiave *non è numerica* (per esempio una *stringa*, parole di un dizionario) si deve trasformare tale chiave in una rappresentazione numerica.

FUNZIONE HASH PER STRINGHE

- Vediamo il caso generale delle stringhe.

- Si può pensare di combinare i caratteri (il loro valore numerico) della stringa per avere una rappresentazione numerica.
- Supponiamo di avere un dizionario con 50000 vocaboli, ognuno composto da non più di 10 caratteri.

FUNZIONE HASH PER STRINGHE

- Un approccio semplice è quello di sommare il valore dei caratteri: per esempio la stringa `casa` produce il codice `3+1+19+1=24`. I valori dei caratteri sono `a=1`, `b=2`, `c=3` …
- Se si considera la stringa più corta (una `a`) e quella più lunga (dieci `z`) si ottiene che l'intervallo dei codici è [1,260], si suppongono 26 caratteri diversi.
- Con questo approccio si ottengono troppe collisioni, cioè *troppe parole associate allo stesso codice*. Si deve cercare una funzione che distribuisca meglio le parole.

FUNZIONE HASH PER STRINGHE

- Un modo differente di "mappare" stringhe in numeri è quello di fare in modo che ogni carattere contribuisca in modo unico al numero finale.
- Si segue un approccio polinomiale, utilizzando lo stesso *principio di rappresentazione in base* dieci dei numeri.
- Le lettere sono moltiplicate per un'appropriata potenza di 26: l'esempio precedente per la stringa `casa` produce il seguente codice $3*26^3+1*26^2+19*26^1+1*26^0=53899$.

FUNZIONE HASH PER STRINGHE

- Questa tecnica produce un numero unico per ogni potenziale parola. Tuttavia l'intervallo dei codici è troppo ampio, circa 26^{10} che vale ~$1.4*10^{14}$; *non è memorizzabile*.
- Inoltre in tal modo si assegnano codici a parole che non esistono (per esempio, abababababab).
- È necessario *comprimere* questo intervallo: si usa di nuovo l'operatore *modulo* (`%`) la dimensione della tabella.
- Quindi la funzione hash torna un valore a cui viene applicato l'operatore modulo per ottenere un indice della tabella.
- Tuttavia in tal modo si producono delle *collisioni*.

FUNZIONE HASH PER STRINGHE

- In generale, ogni classe che usa una funzione hash deve implementare il metodo `GetHashCode()`, che è ereditato dalla classe `object` e pertanto deve essere *overridden*.
- In particolare se due oggetti confrontati col metodo `Equals()` risultano uguali, allora il metodo `GetHashCode()` deve ritornare lo stesso valore (il valore hash deve dipendere dallo stato dell'oggetto).
- Scrivere una funzione hash corretta è "facile" (deve tornare lo stesso intero per oggetti uguali), è "difficile" scrivere una funzione efficiente (deve fornire una buona distribuzione degli oggetti).

VALUTAZIONE DI POLINOMI

- Una implementazione immediata per la valutazione dei polinomi implica un calcolo diretto di tutte le potenze x^n, questo produce un costo computazionale *quadratico*, $O(n^2)$.

- Si può utilizzare l'*algoritmo di Horner* che ha costo *lineare*, $O(n)$. Tale algoritmo si basa sull'uso delle parentesi:

$$a_4x^4+a_3x^3+a_2x^2+a_1x+a_0=(((a_4x+a_3)x+a_2)x+a_1)x+a_0$$

VALUTAZIONE DI POLINOMI

- Per evitare di avere numeri troppo grandi sarebbe necessario fornire alla funzione hash la dimensione della tabella.
- Altrimenti si lascia che la funzione hash torni numeri negativi e si gestiscono al momento dell'operazione modulo.

- Vediamo un esempio per le stringhe (in realtà la classe `string` implementa già il metodo, quindi si può usare quello predefinito).

VALUTAZIONE DI POLINOMI

Esempio di funzione hash (*algoritmo di Horner*) per una classe "stringa":

```
public override int GetHashCode(){
  int hashVal = 0;
  for (int i = 0; i < str.Length; i++)
      hashVal = 26 * hashVal + str[i];
  return hashVal;}
```

Nome di un campo stringa

Per usare il valore ritornato si può scrivere:

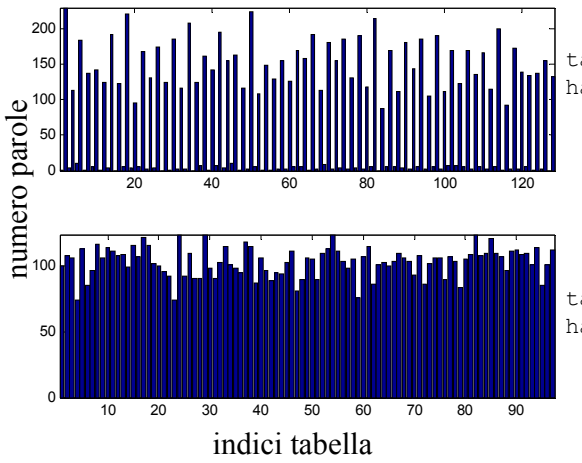
```
int tablesize = 100;
int index = "provaduetre".GetHashCode();
index = index % tablesize;
if (index < 0)
  index += tablesize;
```

Metodo predefinito

NUMERI PRIMI

- Per avere una buona (cioè uniforme) distribuzione degli oggetti nella tabella (e avere collisioni distribuite) è conveniente utilizzare dei *numeri primi* per la base delle potenze e per la dimensione delle tabelle.
- Se molte chiavi condividono un divisore con la dimensione della tabella, allora tendono ad utilizzare la stessa posizione.
- Vediamo un esempio: i grafici rappresentano la distribuzione di 10000 parole distinte con due diverse scelte per la dimensione della tabella e per la base del polinomio.

NUMERI PRIMI




```
tablesize = 128;
hashVal = 26*hashVal + …;
```

```
tablesize = 97;
hashVal = 31*hashVal + …;
```


NUMERI PRIMI

- Per trovare tutti i numeri primi *minori* di un dato numero si può utilizzare il *crivello di Eratostene*.

- L'idea di base è la seguente: dato un numero primo, tutti i suoi multipli non sono primi. Se si considera un insieme che contiene tutti i numeri maggiori o uguali a due, ogni volta che si individua un numero primo si devono escludere dall'insieme tutti i suoi multipli.

NUMERI PRIMI

```
static bool[] Primes(int n)
{
    bool[] p = new bool[n];
    for (int i = 2; i < n; i++)
        p[i] = true;
    for (int i = 2; i < n; i++)
    {
        if (p[i] != false)
        {
            for (int j = i; (long)j * i < n; j++)
                p[i * j] = false;
        }
    }
    return p;
}
```

NUMERI PRIMI

- Questo frammento di codice stampa tutti i numeri primi minori di 100:

```
bool[] vp = Primes(100);
for (int i = 0; i < vp.Length; i++)
    if (vp[i])
        Console.Write(i+" ");
```

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

COLLISIONI

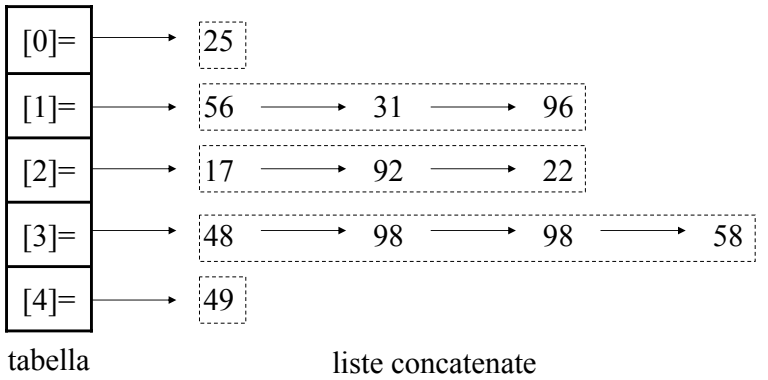
- Quando si "mappano" grandi insiemi di oggetti in interi di piccole dimensioni, le collisioni sono inevitabili. È importante che vi sia una *distribuzione uniforme*.
- Esistono diverse strategie di risoluzione delle collisioni:
  - Indirizzamento aperto (*open-addressing*).
    - Scansione lineare (*linear probing*).
  - Concatenazioni separate (*separate chaining*).

COLLISIONI: indirizzamento aperto

- Se si memorizzano `n` elementi in una tabella di dimensione `m>n` allora si può contare sul fatto di avere *spazio libero* per gestire le collisioni.
- Un semplice metodo ad *indirizzamento aperto* è la *scansione lineare*: quando si verifica una collisione è sufficiente sondare la successiva posizione della tabella per trovare una locazione libera ed eventualmente proseguire sino a tornare alla posizione iniziale, se si è giunti alla fine.
- Se il numero di elementi da inserire non è noto allora è preferibile usare la concatenazione separata.

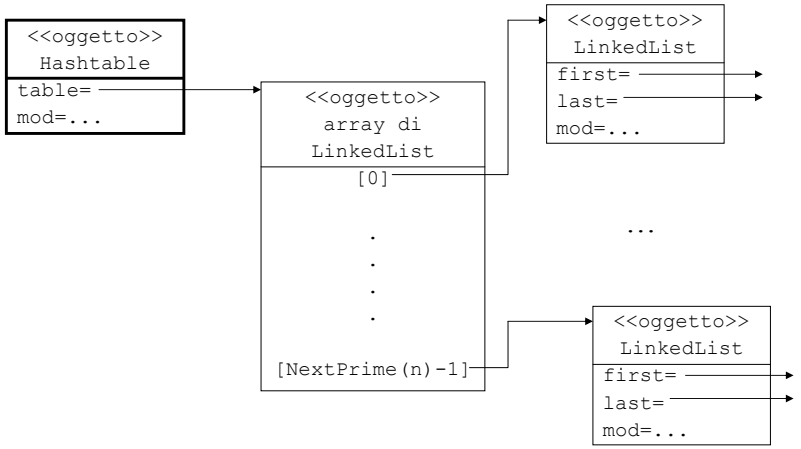
COLLISIONI: concatenazioni separate

- In questo approccio si costruisce una *lista concatenata* per ogni posizione della tabella. Per esempio, inserire gli interi `56 31 17 48 96 98 25 92 98 22 58 49` in una tabella di dimensione `5`:



CONCATENAZIONI SEPARATE: Hashtable

- Un possibile schema di implementazione di una Hashtable:



CONCATENAZIONI SEPARATE: Hashtable

- Si indica con *lf* il fattore di carico (*load factor*): il rapporto tra il numero di oggetti memorizzati nella tabella e la dimensione di tale tabella.
- Si può dimostrare che il costo computazionale dei metodi descritti è proporzionale a $1+lf$.
- Si può pensare che il costo per trovare una posizione nella tabella hash è 1, poi si deve scorrere la relativa lista per trovare/inserire l'oggetto. La lista è lunga (se la distribuzione delle chiavi è uniforme) proprio *lf*.

CONCATENAZIONI SEPARATE: Hashtable

- In media ogni lista contiene un numero di oggetti pari al rapporto tra il numero di oggetti memorizzati nella tabella hash e la dimensione di tale tabella, quindi *lf*.
- Se il fattore di carico *lf* cresce e quindi le liste concatenate contengono molti oggetti, allora è necessario aumentare la dimensione della tabella (al *successivo numero primo*) e inserire tutti gli oggetti nella nuova tabella.

APPLICAZIONI: correttore ortografico

- Vediamo come sviluppare un semplice correttore ortografico.
- Per prima cosa si carica da file (`It.dic`) un dizionario di 277434 vocaboli, poi si legge il file di testo (`es1.txt`) che si vuole correggere. Ogni parola letta è cercata nel dizionario: se tale parola non è trovata allora è visualizzata. Inoltre si incrementa un contatore degli errori.

APPLICAZIONI: correttore ortografico

```
Hashtable dizionario = new Hashtable(277434);

StreamReader fin = new StreamReader("It.dic");
string parola;
while ((parola = fin.ReadLine()) != null)
    dizionario.Add(parola, "");
fin.Close();

fin = new StreamReader("es1.txt");
string[] line;
string str;
int c = 0;
Console.WriteLine("Parole non trovate:");
```


APPLICAZIONI: correttore ortografico

```
while ((str = fin.ReadLine()) != null)
{
    line = str.Split(' ');
    for (int i = 0; i < line.Length; i++)
        if (!dizionario.Contains(line[i]))
        {
            Console.Write(line[i] + " ");
            c++;
        }
}
Console.WriteLine("\n\nErrori: {0}", c);
```


APPLICAZIONI: correttore ortografico

- Un breve file di prova (`es1.txt`):

introdurre i principali metodi utilizzati per organizzare e rappresentare le informazioni , le strutture dati , al finne di ottenerne una elaborazione efficiente , gli algoritmis

- L'output dell'applicazione:

Parole non trovate:
finne algoritmis

Errori: 2

APPLICAZIONI: correttore ortografico

- Dal punto di vista delle prestazioni si ottengono buoni risultati.
- Si è utilizzato un file di testo con circa 91000 parole e si è usato lo stesso dizionario dell'esempio precedente.

- I tempi di esecuzione sono i seguenti:
  - Circa 0.25 secondi per caricare il dizionario.
  - Circa 0.015 secondi per elaborare il file di 91000 parole.

APPLICAZIONI: correttore ortografico

- Vediamo un confronto con un correttore implementato con array di stringhe. Principali differenze nel codice:

```
string[] dizionario = new string[277434];
…
int ii = 0;
while ((parola = fin.ReadLine()) != null)
       dizionario[ii++]=parola;
…
Array.Sort(dizionario);
…
if (Array.BinarySearch(dizionario, line[i]) < 0)
…
```


APPLICAZIONI: correttore ortografico

- Nelle stesse condizioni precedenti (file di testo con circa 91000 parole) i tempi di esecuzione sono i seguenti:
  - Circa 1.20 secondi per *ordinare* il dizionario.
  - Circa 0.32 secondi per elaborare il file di 91000 parole.

APPLICAZIONI: funzioni hash

- Alcune funzioni hash sono sofisticate, in particolare quelle nate in ambito crittografico, come MD5 (*Message-Digest algorithm 5*) e SHA (*Secure Hash Algorithm*).

- Tali funzioni hash sono *iterative* e lavorano su blocchi di bit, applicandovi un'opportuna sequenza di operazioni logiche per manipolare i bit (per esempio, l'OR esclusivo o la rotazione dei bit).

APPLICAZIONI: funzioni hash iterative

- *Funzioni hash iterative*: dividono la chiave k in blocchi di sequenze binarie k_0, k_1, … k_{s-1} e lavorano su tali blocchi, per esempio attraverso l'OR esclusivo bit a bit, di fatto ripiegando (*folding*) la chiave su se stessa:

$$HF(k) \rightarrow k_0 \oplus k_1 \oplus \ldots \oplus k_{s-1}$$

Le sequenze k_i hanno la stessa lunghezza, valori in $[0, n-1]$ e il simbolo $\oplus$ indica l'OR esclusivo bitwise.

APPLICAZIONI: funzioni hash

- Poiché sono state verificate delle collisioni per MD5, tale algoritmo non viene più usato in ambito crittografico, ma solo per verificare l'integrità dei file.
- Le varianti dell'algoritmo SHA sono utilizzate in applicazioni nell'ambito della sicurezza e dei protocolli, come TLS, SSL, PGP e SSH.

- Le funzioni hash sono utilizzate anche nei sistemi distribuiti di condivisione dei file (*peer-to-peer,* P2P).

APPLICAZIONI: funzioni hash

- Nei sistemi distribuiti l'informazione è condivisa e distribuita tra tutti i *client* o *peer*, piuttosto che concentrata in pochi *server*: con un grande vantaggio in termini di *banda passante, capacità di calcolo* e *tolleranza ai guasti*.
- L'esempio più conosciuto di P2P è quello della condivisione dei file: quando si scarica un file i suoi blocchi sono recuperati da vari punti della rete.
- In tale scenario, lo stesso file può apparire con nomi diversi e file diversi possono apparire con lo stesso nome.

APPLICAZIONI: funzioni hash

- Quando i *peer* devono verificare quali file hanno in comune, non potendo affidarsi ai nomi dei file, devono verificarne il contenuto. Tuttavia, data la grande dimensione dei file, non possono scambiarsi direttamente il contenuto dei file.
- La soluzione è utilizzare delle funzioni hash: per ogni file viene calcolato il corrispondente valore hash (che dipende dal contenuto del file) e viene inviato solo tale valore di hash.
- Dal confronto di tali valori si può dedurre con certezza quali file sono diversi e, con altissima probabilità, quali file sono uguali.

APPLICAZIONI: funzioni hash

- Un numero sempre crescente di sistemi distribuiti si diffonde in diversi ambiti: per esempio, la telefonia via Internet e il trattamento di grandi quantità di dati (come quelli del genoma umano).

- Inoltre i sistemi P2P possono essere usati per eseguire programmi progettati per utilizzare un elevato numero di computer sparsi sulla rete: un esempio sono i programmi per verificare le potenzialità di nuovi farmaci.