Tabelle hash

Un altro modo efficiente per rappresentare un insieme su cui occorre effettuare ricerche è quello di usare una tabella hash con concatenazione.

Teorema

> Supponiamo di rappresentare un insieme di n elementi mediante una tabella hash di dimensione m in cui le collisioni sono risolte per concatenazione.
>
> Nell'ipotesi di uniformità semplice della funzione hash, una ricerca richiede in media tempo $\Theta(1 + \frac{n}{m})$.

- Se la dimensione m della tabella è dello stesso ordine di grandezza del numero n di elementi in essa contenuti ($m = \Theta(n)$), la complessità della ricerca è $\Theta(1)$ (costante).
- Se la tabella è sottodimensionata (m è "piccolo" rispetto a n), la complessità della ricerca diventa $\Theta(n)$ (lineare) come nelle liste (situazione analoga a quanto avviene con gli alberi di ricerca degeneri).

Il rapporto $\frac{n}{m}$ è chiamato *fattore di carico* e rappresenta il numero medio di elementi in una lista di collisione.

Funzione hash

Per costruire una tabella hash di M elementi occorre definire una funzione hash

$$H : \mathbf{N} \rightarrow \{0, \ldots, M-1\}$$

dove $\mathbf{N} = \{0, 1, 2, \ldots\}$ è l'insieme dei *numeri naturali*.

Rappresentazione di una tabella hash

Una tabella hash si può rappresentare come un *array di liste*

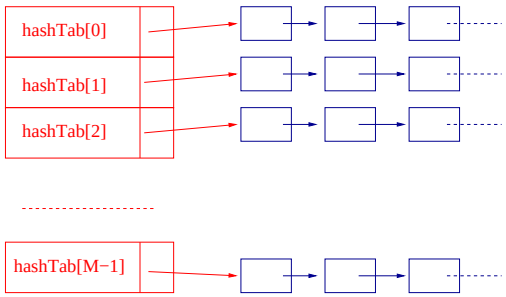
```
element*  hashTab[M];
```

dove:

- `element` è la struttura usata per rappresentare un elemento della lista (e dipende dal tipo degli elementi da inserire nella tabella)
- `M` è la dimensione della tabella.

Per ogni $0 \leq k \leq M-1$:

- la lista `hashTab[k]` contiene gli interi n tali che $H(n) = k$.



Principali operazioni

- Ricerca di un elemento

  L'elemento n, se è nella tabella, si trova nella lista $hashTab[k]$, dove $k = H(n)$.

- Inserimento di un elemento

  L'elemento n va inserito nella lista $hashTab[k]$, dove $k = H(n)$.

- Cancellazione di un elemento

  L'elemento n va cancellato dalla lista $hashTab[k]$, dove $k = H(n)$.

Osservazioni importanti

Come afferma il teorema precedente, per avere una tabella hash devono verificarsi le seguenti condizioni:

- La funzione hash deve distribuire in modo uniforme le chiavi nella tabella. Evitare di usare funzioni hash per cui tale requisito non è verificato, usare invece le funzioni sui libri di testo.
- La dimensione `M` della tabella deve essere dello stesso ordine di grandezza del numero n di elementi della tabella.

  Questo significa che `M` non può essere una costante, ma un parametro del programma modificabile a piacere.

  Tipicamente si pone nel codice una definizione del tipo

```
#define M  10000 // dimensione della tabella
```

  in modo che per cambiare la dimensione della tabella sia sufficiente modificare il valore di `M` e ricompilare il programma, senza però dover ridefinire la funzione hash.

Rappresentazione di elementi con numeri naturali

La funzione hash è definita come

$$H : \mathbf{N} \rightarrow \{0, \ldots, M-1\}$$

Se però gli elementi da inserire nella tabella non sono numeri naturali, prima di applicare H occorre definire una funzione R che permette di *rappresentare* un elemento con un numero naturale.

La funzione R deve essere *iniettiva*, ossia per ogni coppia di elementi e_1 e e_2, vale:

- se $e_1 \neq e_2$, allora $R(e_1) \neq R(e_2)$.

Il valore della funzione hash su un elemento e è dato da

$$H(R(e))$$

Esempio 1: rappresentazione di interi

Per rappresentare un numero intero con un naturale, si può usare la seguente funzione R:

$$R(k) = \begin{cases} 2 \cdot k & \text{se } k \geq 0 \\ 2 \cdot (-k) - 1 & \text{se } k < 0 \end{cases}$$

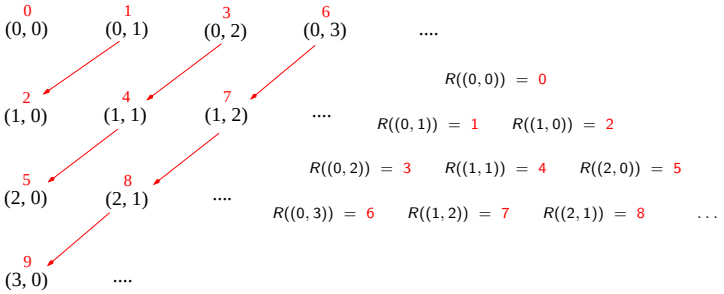
Quindi:

$$\begin{aligned} R(0) &= 0 \\ R(-1) &= 1 \\ R(1) &= 2 \\ R(-2) &= 3 \\ R(2) &= 4 \\ &\vdots \end{aligned}$$

È facile verificare che R è iniettiva.

Esempio 2: rappresentazione di coppie di naturali

Per rappresentare una coppia di naturali con un naturale si può immaginare di ordinare le coppie di naturali come nella figura, seguendo le frecce diagonali.



Per ogni $x \geq 0$ e $y \geq 0$, la coppia (x, y) è l'elemento di posto x sulla diagonale che inizia in $(0, x + y)$.

Indicando con $\Delta(k)$ il numero degli elementi sulla diagonale che inizia in $(0, k)$, si ha:

$$R((x, y)) \quad = \quad \Delta(0) \; + \; \Delta(1) \; + \; \cdots \; + \; \Delta(x + y - 1) \; + \; x$$

Poiché $\Delta(k) = k + 1$, vale:

$$R((x, y)) \quad = \quad 1 + 2 + \cdots + (x+y) + x \quad = \quad \frac{(x + y) \cdot (x + y + 1)}{2} + x$$

quindi

$$R((x, y)) \quad = \quad \frac{(x + y)^2 + 3x + y}{2}$$

Esempio 3: rappresentazione di coppie di interi

Basta combinare le rappresentazioni degli esempi 1 e 2.

Esempio 4: rappresentazione di parole

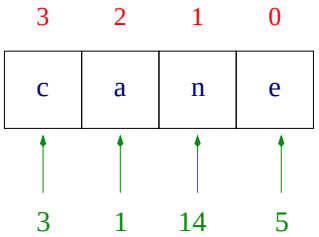
Supponiamo di voler rappresentare con un numero naturale una parola costruita sull'alfabeto di 26 lettere a, b, c,. . . , z.

Si può considerare una parola come un numero in base 27, dove il valore delle cifre è dato da:

Cifra	Valore
a	1
b	2
c	3
⋮	⋮
z	26

Esempio

La rappresentazione numerica della parola cane è:



$$R(\texttt{cane}) \quad = \quad 3 \cdot 27^3 \; + \; 1 \cdot 27^2 \; + \; 14 \cdot 27^1 \; + \; 5 \cdot 27^0 \quad = \quad 60161$$

Si noti che cane è l'*unica* parola w tale che $R(w) = 60161$.

Nota

Per avere l'unicità della rappresentazione, occorre porre a uguale a 1.

Se si pone a uguale a 0 (e, di conseguenza, b uguale a 1, c uguale a 2, ecc.) si ha:

$$R(\mathtt{a}) = 0 \cdot 27^0 = 0 \qquad R(\mathtt{aa}) = 0 \cdot 27^1 + 0 \cdot 27^0 = 0$$

Quindi R non è iniettiva.

Con la rappresentazione data si ha invece;

$$R(\mathtt{a}) = 1 \cdot 27^0 = 1 \qquad R(\mathtt{aa}) = 1 \cdot 27^1 + 1 \cdot 27^0 = 28$$

Calcolo di R

La funzione `charValue()` calcola il valore di $R(c)$, dove c è una lettera dell'alfabeto.

```
int charValue(char c){
  return (c-'a') + 1;
}
```

Il valore $R(w)$ di una parola `w` è calcolato dalla funzione:

```
int value(char* w){
  int val = 0;
  while(*w! = '\0'){
    val = 27 * val + charValue(*w);
    w++; // avanza di un carattere
  }
  return val;
}
```

Con parole lunghe si ottengono valori numerici molto alti, non rappresentabili nei tipi predefiniti del C.

Se la funzione hash H è definita con il metodo della divisione, si riesce a calcolare $H(R(w))$ evitando errori di overflow sfruttando le proprietà del resto della divisione.

Metodo della divisione

Supponiamo che la tabella abbia dimensione `M`.

La funzione hash

$$H : \mathbf{N} \to \{0, \ldots, M-1\}$$

è la funzione tale che, per ogni intero $n \geq 0$

$$H(n) = n \,\%\, M \qquad \text{resto della divisione intera fra } n \text{ e } M$$

Il valore hash di una parola w è dato da $H(R(w))$, che coincide con

```
value(w) % M
```


Proprietà del resto della divisione

$$(X + Y) \,\%\, M = (X \,\%\, M + Y \,\%\, M) \,\%\, M$$

Esempio

Supponiamo di poter usare solo numeri di due cifre e di dover calcolare il valore di

$$(90 + 12) \,\%\, 7$$

Non possiamo eseguire la somma, in quanto il risultato supera due cifre.

Applicando la proprietà del resto, non si hanno invece errori di overflow (i numeri non superano le due cifre):

$$\begin{aligned} (90 + 12) \,\%\, 7 &= (90 \,\%7 + 12 \,\%7) \,\%\, 7 \\ &\quad (6 + 5) \,\%\, 7 \\ &\quad 11 \,\%\, 7 \\ &\quad 4 \end{aligned}$$

La seguente funzione `hash()` calcola direttamente il valore $H(R(w))$ di una parola w (dove H e R sono definiti come prima) usando lo stesso principio.

```
int hash(char* w){
  int val = 0;
  while(*w != '\0'){
    val = (27*val + charValue(*w)) % M;
    w++;
  }
  return val;
}
```


Nota

Non si realizza una tabella hash definendo un array di 26 liste e usando una funzione hash che mette nella prima lista le parole che iniziano per a, nella seconda lista quelle che iniziano per b, ecc. Questo perché nessuno dei due requisiti prima esposti è rispettato. Infatti:

- La distribuzione delle parole nella tabella non è uniforme (una funzione hash dovrebbe tener conto di tutte le lettere che formano una parola, non solo della prima!).
- Cosa più grave, la dimensione della tabella non può essere modificata (occorrerebbe ridefinire la funzione hash), quindi se il numero di parole inserite è elevato, la tabella è inefficiente (è come avere un'unica lista di parole).

Esercizio

Con il metodo della divisione, per avere buone prestazioni il valore di `M` deve verificare certe proprietà (ad esempio, deve essere un numero primo; vedere il libro). Scrivere un programma che, dato un numero naturale n, determina il numero intero s più vicino a n che verifichi tali le proprietà, quindi costruisce la tabella hash di dimensione s.

Esempio: costruzione di un dizionario

Un dizionario è un insieme di parole costruite su un alfabeto (esempio, l'alfabeto inglese).

È noto che il tempo medio richiesto per costruire un insieme di n elementi, dove ogni inserimento richiede una *ricerca* per controllare se l'elemento da inserire è già presente nell'insieme, è:

- $O(n^2)$ se l'insieme è rappresentato da una *lista* (semplice, bidirezionale, ecc.).
- $O(n.\log n)$ se l'insieme è rappresentato da un *albero binario di ricerca*.
- $O(n.\frac{n}{m})$ se si usa una *tabella hash* di m elementi con liste di collisione.

Un utile esercizio è quello di costruire un dizionario usando le tre strutture nominate prima a partire dallo stesso insieme di parole, e confrontare sperimentalmente il tempo richiesto per la costruzione.

Il programma:

1. Genera un insieme $\mathcal{D}$ contenente n parole generate casualmente e le memorizza in un array.

   Il valore di n è letto in input; nel programma deve essere stabilita la lunghezza minima e massima di una parola.
2. Inserisce le parole contenute in $\mathcal{D}$ in albero binario di ricerca.
3. Inserisce le parole contenute in $\mathcal{D}$ in una tabella hash.
4. Inserisce le parole contenute in $\mathcal{D}$ in una lista.

Ogni parola va inserita una sola volta, quindi prima di inserirla va effettuata una ricerca.

Il programma deve stampare il tempo richiesto per costruire il dizionario.

Esempio di esecuzione

0utput ottenuto generando 20000 parole (17821 parole distinte).

```
ALBERO BINARIO DI RICERCA
TEMPO:  0.0700 sec.
Elementi inseriti: 17821
Altezza: 32

TABELLA HASH
TEMPO:  0.0500 sec.
Elementi inseriti: 17821
Dimensione tabella: 5000
Fattore di carico:  3.5642
Max lunghezza lista: 12 -  Media: 3.5642

LISTA
TEMPO:  39.1000 sec.
Elementi inseriti: 17821
```

Si noti la differenza fra il tempo impiegato con la lista e quello impiegato nei primi due casi! L'altezza dell'albero di ricerca e la massima lunghezza di una lista di collisione della tabella hash danno indicazioni sul caso peggiore della ricerca nelle rispettive strutture (confrontare con la lunghezza della lista!).

Altri tipi di ricerca

Vediamo un esempio di ricerca che può essere effettuata in modo efficiente con gli alberi binari di ricerca.

Ricerca di parole con un determinato prefisso

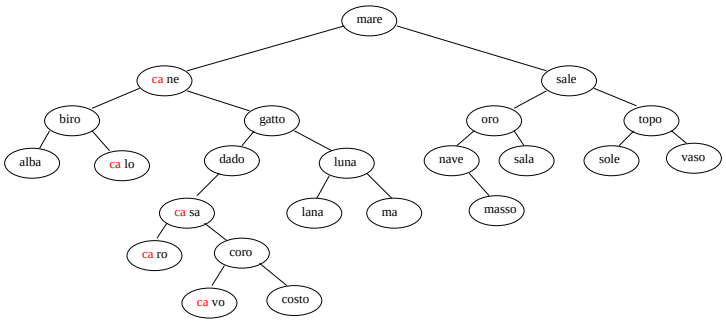
Dato un alfabeto Σ e due parole $v, w \in \Sigma^*$, v è un *prefisso* di w se e solo se esiste $z \in \Sigma^*$ tale che $w = vz$ (è compreso il caso in cui v è uguale a w).

Supponiamo di dover stampare le parole di un dizionario che hanno un certo prefisso v ($v \in \Sigma^*$).
La strategia più semplice è quella di leggere tutte le parole del dizionario e stampare quelle di prefisso v; la complessità è $O(n)$, dove n è il numero delle parole nel dizionario.

- Se il dizionario è rappresentato con una lista o con una tabella hash questa è l'*unica* strategia possibile.
- Se il dizionario è rappresentato con un albero binario di ricerca, si può adottare una strategia migliore che evita di esplorare i nodi in cui *sicuramente* non esistono parole con prefisso v.

Esempio

Consideriamo l'albero binario di ricerca (i cui nodi sono ordinati rispetto all'ordinamento lessicografico fra parole)



e supponiamo di voler stampare le parole con prefisso "ca".

Osserviamo che:

- Il nodo `cane` ha prefisso "ca"; possono esistere parole con prefisso "ca" sia nel sottoalbero sinistro (ad esempio, "calo") che nel sottoalbero destro (ad esempio, "casa").

  La ricerca deve continuare su entrambi i sottoalberi

- Consideriamo ora il nodo `biro`.
  Poiché "biro" è *minore* di "ca", nel sottoalbero sinistro non possono esserci parole con prefisso "ca", ma possono essercene nel sottoalbero destro (ad esempio, "calo").

  La ricerca va continuata *solo* sul sottoalbero destro.

- Discorso simmetrico per il nodo `gatto`.
  Essendo "gatto" maggiore di "ca", le parole di prefisso "ca" possono trovarsi solo nel sottoalbero sinistro.

  La ricerca deve continuare *solo* nel sottoalbero sinistro.

Esercizi

1. Applicando le osservazioni fatte, scrivere una funzione ricorsiva

```
void prefisso(char* pref, node* r)
```

che stampa le parole aventi prefisso `pref` dell'albero binario di ricerca di radice `r`.

2. Scrivere due funzioni ricorsive

```
void prefissoCresc(char* pref, node* r)
void prefissoDecr(char* pref,  node* r)
```

analoghe alla funzione dell'esercizio 1, in cui la stampa delle parole viene fatta rispettivamente in ordine crescente e in ordine decrescente.

Ad esempio, se `pref` è "ca" e `r` è l'albero della figura precedente, la prima funzione deve stampare

```
calo cane caro casa cavo
```

la seconda

```
cavo casa caro cane calo
```


Alberi lessicografici

Un albero lessicografico su un alfabeto $\mathcal{A}$ permette di rappresentare un insieme di parole costruite su $\mathcal{A}$.

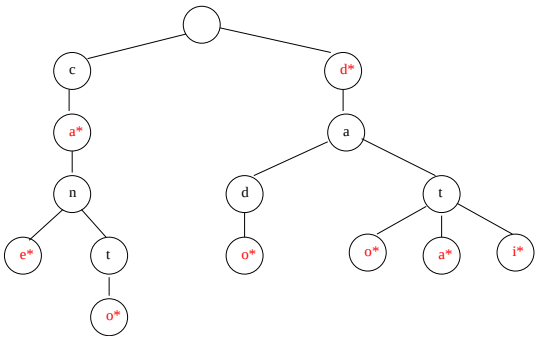
- Ogni nodo diverso dalla radice contiene un carattere di $\mathcal{A}$
- Ogni nodo può avere tanti figli quante sono le lettere di $\mathcal{A}$.
- I nodi sono distinti in nodi terminali (contrassegnati da $*$ nell'esempio) e non terminali.

Una parola dell'albero è rappresentata da un cammino che parte dalla radice a un nodo terminale. Si noti che:

- L'altezza dell'albero è data dalla massima lunghezza di una parola.
- Il numero di nodi presenti nell'albero è in genere inferiore al numero di parole rappresentate nell'albero.

Risultano particolarmente efficienti la ricerca di una parola e la ricerca per prefisso.

Esempio



Asuumendo $\mathcal{A}$ sia l'alfabeto italiano, l'albero in figura rappresenta l'insieme (dizionario) contenente le parole

`ca`, `cane`, `canto`, `d`, `dado`, `dato`, `data`, `dati`.

Si noti che, ad esempio, la parola `can` non è nel dizionario in quanto il nodo `n` non è terminale.

Per inserire `can` è sufficiente rendere terminale il nodo `n`.

Tipi enumerativi

In C è possibile definire un tipo *enumerativo*, ossia un insieme finito i cui elementi sono denotati da un nome (vedere i dettagli sul libro).

Esempio

Supponiamo di voler rappresentare una scacchiera (chessboard) in cui ogni casella (square) può contenere una regina (queen), un cavallo (horse) oppure può essere vuota (empty).

Per rappresentare una casella si può definire un nuovo tipo `enum square` i cui elementi sono *queen*, *horse* e *empty*.

```
enum square {queen, horse, empty};
```

Con

```
typedef enum  square  square;
```

viene definito il nuovo tipo `square` che è sinonimo di `enum square`

La funzione printSquare() stampa il contenuto della casella s passata come parametro.

```
void printSquare(square s){
  switch(s){
  case queen:
    putchar('X');  // simbolo per la regina
    break;
  case horse:
    putchar('#'); // simbolo per il cavallo
    break;
  case empty:
    putchar('-');  // simbolo per la casella vuota
    break;
  }
}
```


La funzione printChessBoard() stampa la scacchiera ch di dimensione row × col.

```
void printChessBoard(square **ch, int row, int col){
  int r,c;
  for(r=0 ; r<row ; r++){
     for(c=0; c<col ; c++)
        printSquare(ch[r][c]);
     putchar('\n');
  }
}
```


Esercizi

1. Scrivere il codice della funzione

```
square** newChessBoard(int row, int col)
```

che crea una scacchiera di dimensione row × col in cui tutte le caselle sono vuote.

2. Provare a eseguire le seguenti linee di codice

```
int main(){
  square** ch;
  ch = newChessBoard(3,5); // ch e' una scacchiera 3 X 5
  ch[0][3] = queen;
  ch[1][0] = horse;
  ch[1][4] = horse;
  ch[2][1] = queen;
  printChessBoard(ch,3,5);  // stampa ch
  return 0;
}
```