FONDAMENTI DI INFORMATICA II – modulo Algoritmi e Strutture dati

18 giugno 2009 - ANNO ACCADEMICO 2008/2009

NOME: COGNOME: MATRICOLA:
PROGETTO: SI NO (barrare SI se si intende svolgere lo scritto ridotto)

Esercizio 1
Scegliere la memorizzazione ottima di un insieme di elementi sapendo che:

- il numero massimo di elementi è 100
- ogni elemento occupa due celle di memoria
- il numero medio di elementi è 20 e l'insieme da memorizzare varia molto spesso

nei due casi seguenti:

a) Si vuole ottenere la minima occupazione di memoria, indipendentemente dal tempo di ricerca
b) Si vuole ottenere la massima velocità di ricerca, utilizzando non più di 300 celle di memoria

In ognuno dei due casi a) e b) elencare le varie possibilità e confrontarle come tempo di ricerca e occupazione di memoria.

a)

Con questi vincoli la migliore struttura dati è una lista semplice con occupazione 20*3=60 celle in media e tempo di ricerca O(n). L'array con ricerca lineare O(n) o il metodo hash con alpha=1 occupano comunque 200 celle e l'albero binario di ricerca occupa 20*4=80 celle in media con tempo di ricerca O(logn). Alternativamente si può usare una tabella hash con concatenazione, con occupazione media 60 + la memoria dell'array iniziale. In conclusione la lista semplice è la struttura dati che occupa la minor memoria.

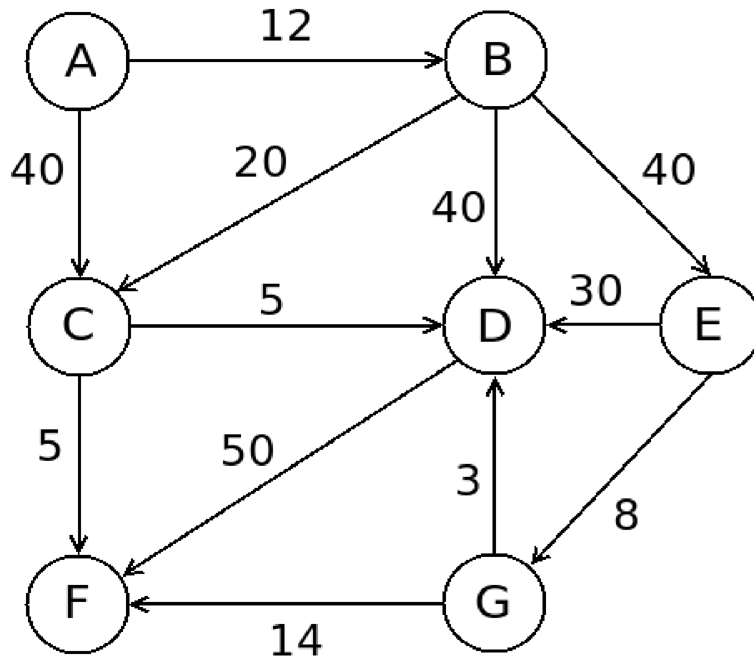
b)

Per tutti i metodi diversi dalla ricerca hash valgono i dati del caso a), perché la maggiore disponibilità di memoria non migliora le prestazioni di questi metodi. Si può utilizzare una tabella hash ad indirizzamento aperto con un array in cui ogni elemento occupa due celle di memoria. Data la disponibilità di 300 celle, possiamo dimensionare l'array a 150 elementi e utilizzare ad esempio come funzione hash il resto della divisione dell'elemento per k=150. Abbiamo alpha=n/k=100/150=2/3. Il tempo di accesso è praticamente costante.

Situazione	*Esercizi*	*Tempo*
Senza progetto	**1,2,3,4,5**	80 min
Con progetto	**1,2,3**	40 min

1	*2*	*3*	*4*	*5*

Esercizio 2
Dato il seguente grafo orientato:



a) Supponendo che gli archi siano memorizzati tramite liste di adiacenza (ordinate alfabeticamente), visitare il grafo mediante la visita in profondità e riportare l'ordine in cui vengono visitati i nodi.
b) applicare l'algoritmo di Dijkstra per trovare il cammino minimo da A agli altri nodi, **supponendo che il grafo sia non orientato**.

A)

A B C D F E G

B)

Q	distanze A	B	C	D	E	F	G	predecessori A	B	C	D	E	F	G
{A,B,C,D,E,F,G}	0	∞	∞	∞	∞	∞	∞	-	-	-	-	-	-	-
{B,C,D,E,F,G}	0	12	40	∞	∞	∞	∞	-	A	A	-	-	-	-
{C,D,E,F,G}	0	12	32	52	52	∞	∞	-	A	B	B	B	-	-
{D,E,F,G}	0	12	32	37	52	37	∞	-	A	B	C	B	C	-
{D,E,G}	0	12	32	37	52	37	51	-	A	B	C	B	C	F
{E,G}	0	12	32	37	52	37	40	-	A	B	C	B	C	D
{E}	0	12	32	37	48	37	40	-	A	B	C	G	C	D
{}	0	12	32	37	48	37	40	-	A	B	C	G	C	D

Esercizio alternativo. Trovare l'albero di copertura minimo utilizzando l'algoritmo di Kruskal.
Soluzione. Gli archi scelti sono, nell'ordine, i seguenti: (G,D)(C,D) (C,F) (E,G) (A,B) (B,C)

Esercizio 3

Calcolare la complessità del blocco (indicando le relazioni di ricorrenza di tempo e risultato per ogni funzione) in funzione di n:

```
{
  int a = 0;
  for (int i=0; i <= g(n); i++)
     a += f(n);
}
```

con le funzioni **f** e **g** definite come segue:

```
int f(int x) {
   if (x<=1)
     return 1;
   cout << f(x/2) + f(x/2);
   return f(x/2) + 1;
}
```

```
int g(int x) {
  int a=0;
  for (int i=0; i <= f(x); i++)
     a++;
  return a;
}
```

Indicare per esteso le relazioni di ricorrenza e, per ogni comando ripetitivo, il numero di iterazioni e la complessità dell'iterazione singola.

$T_f(0,1)= k_1$

$T_f(n)= k_2 + 3\ T_f(n/2)$ $T_f(n)$ è $O(n^{1,5})$

$R_f(0,1)= 1$

$R_f(n)= 1 + R_f(n/2)$ $R_f(n)$ è $O(\log n)$

Calcolo $T_g(n)$

for:

numero iterazioni: $O(\log n)$ Complessità singola iterazione: $O(n^{1,5})$

complessità del for: $O(n^{1,5} \log n)$

$T_g(n)$ è $O(n^{1,5} \log n)$

$R_g(n)$ è $O(\log n)$

Calcolo for del blocco:

numero iterazioni: $R_g(n) = O(\log n)$

Complessità singola iterazione: $T_g(n) = O(n^{1,5} \log n)$

Complessità totale blocco = $O(n^{1,5}\ (\log n)^2)$

Esercizio 5

Scrivere una funzione **`bool trefiglifoglie(Node* t)`** che ritorna **`true`** se tutti i nodi dell'albero generico di radice **`t`** hanno almeno **`3`** figli che sono foglie oppure sono foglie essi stessi, **`false`** altrimenti. Si supponga che l'albero generico sia non vuoto e memorizzato figlio fratello.

```
bool trefiglifoglie (Node* t) {
if (!t) return true;
if (!t->left) return trefiglifoglie(t->right);
return conta(t->left)&&
       trefiglifoglie(t->left)&& trefiglifoglie(t->right);
}

int conta (Node* t) {
  int a=0;
  while (t) {
    if (!t->left) a++;
    if (a == 3) return true;
    t=t->right;
  }
  return false;
}
```

Esercizio 5

Sia dato un albero binario ad etichette intere t. Si scriva una funzione che restituisce il numero dei sottoalberi (non vuoti) di t che hanno una media delle etichette superiore a 3. Calcolare la complessità della soluzione proposta in funzione del numero n di nodi dell'albero.

```
int contamedia(Node* t, T& nodi, T& somma) {
  if (!t) {
    nodi = 0;
    somma = 0;
    return 0;
  }
  int nodi_l, nodi_r, somma_l, somma_r;
  int conta_l = contamedia(t->left,nodi_l,somma_l);
  int conta_r = contamedia(t->right,nodi_r,somma_r);
  somma = somma_l + somma_r + t->label;
  nodi = nodi_l + nodi_r + 1;
  float media = (float) (somma) / nodi;
  return conta_l + conta_r + (media > 3.0);
};
```

Complessità O(n)