Algoritmi e basi di dati – modulo Algoritmi e Strutture dati – a.a. 2010/2011

22 luglio 2011

Esercizio 1

Dato l'array **`A=[64 -1 -2 18 28 -1 14 22]`** (dove **`-1`** rappresenta una posizione disponibile e **`-2`** una cancellata), la funzione hash ***h(x) = x mod 8*** e la legge di scansione lineare, indicare l'array dopo l'inserimento dell'elemento **`15`** e dopo l'estrazione dell'elemento **`18`** secondo il metodo hash. Indicare gli indici delle posizioni lette durante le operazioni di inserimento ed estrazione.

	Array:	Posizioni lette:
Dopo l'inserimento di **`15`**.	**`[64 15 -2 18 28 -1 14 22]`**	**`7,0,1`**
Dopo l'estrazione di **`18`**.	**`[64 15 -2 -2 28 -1 14 22]`**	**`2,3`**

Esercizio 2

Dato l'array **`A=[7 1 8 4 1 5 9]`**, indicare tutte le chiamate a quicksort con il relativo perno generate dalla chiamata **`quicksort(A,0,6)`**.

Array:	Inf:	Sup:	Perno:
`7 1 8 4 1 5 9`	**`0`**	**`6`**	**`4`**
`1 1 4 8 7 5 9`	**`0`**	**`2`**	**`1`**
`1 1 4 8 7 5 9`	**`1`**	**`2`**	**`1`**
`1 1 4 8 7 5 9`	**`3`**	**`6`**	**`7`**
`1 1 4 5 7 8 9`	**`5`**	**`6`**	**`8`**
`1 1 4 5 7 8 9`			

Esercizio 3

Calcolare la complessità del **`for`** in funzione di **`n>0`**.

```
for (int i=0; i <= g(f(n)); i++) cout << i;
```

con le funzioni **`f`** e **`g`** definite come segue. Indicare per esteso le relazioni di ricorrenza e, per ogni comando ripetitivo, il numero di iterazioni e la complessità della singola iterazione.

```
int g(int x) {
  if (x<=0) return 1;
  int a = 0;
  for (int i=0; i <= x; i++)
    a += i;
  cout << g(x-1);
  return g(x-1); }
```

```
int f(int x) {
  if (x<=0) return 1;
  int b = 3+f(x/2);
  cout << b;
  return 1+f(x/2);}
```

$T_g(0)= d$

$T_g(n)= cn + 2T_g(n-1)$ $T_g(n)$ è $O(2^n)$

$R_g(n)= 1$ $R_g(n)$ è $O(1)$

Funzione f

$T_f(0)= d$
$T_f(n)= c+ 2T_f(n/2)$ $T_f(n)$ è $O(n)$

$R_f(0)= d$
$R_f(n)= c+ R_f(n/2)$ $R_f(n)$ è $O(\log n)$

Calcolo for del blocco:
numero iterazioni: O(1)
Complessità della singola iterazione: $T_f(n) + T_g(\log n) = O(n) + O(n) = O(n)$
Complessità del for: =O(n)

Esercizio 4

Scrivere una funzione ricorsiva che, dato un albero binario **t** con etichette intere e una etichetta **x**, conta i nodi che hanno più nodi con etichetta **x** nel sottoalbero sinistro di nodi con etichetta **x** nel sottoalbero destro.

```
int piu_x_sx(Node* t, int x, int& num) {
  if (!t) { num = 0; return 0; }
  int sx, dx, num_sx, num_dx;
  sx = piu_x_sx(t->left, x, num_sx);
  dx = piu_x_sx(t->right, x, num_dx);
  num = num_sx + num_dx + (t->info == x);
  return sx + dx + (num_sx > num_dx);
}
```

Esercizio 5

Scrivere una funzione che, dato un albero generico **t** con etichette tutte diverse fra di loro, e due etichette **x** e **y**, scambia il primo sottoalbero del nodo con etichetta **x** con il primo sottoalbero del nodo con etichetta **y**. Se **x** o **y** non compaiono nell'albero, o se **x** o **y** non hanno sottoalberi, la funzione lascia l'albero inalterato.

```
Node* cerca(Node* t, int x) {
  if (!t) return NULL;
  if (t->info == x) return t;
  Node* a = cerca(t->left, x);
  if (a) return a;
  a = cerca(t->right, x);
  return a;
}
```

```
void scambia(Node* t, int x, int y) {
  Node* nodex = cerca(t, x);
  if (!nodex) return;
  if (!nodex->left) return;
  Node* nodey = cerca(t, y);
  if (!nodey) return;
  if (!nodey->left) return;

  Node* a1, a2, b1, b2;
  a1 = nodex->left;
  a2 = a1->right;
  b = nodey->left;
  nodex->left = b;
  nodey->left = a1;
  a1->right = b->right;
  b->right = a2;
}
```