Appunti di Algoritmi e strutture dati

di Gianluca Mondini

1 Complessità di un algoritmo

1.1 Definizione

È una funzione (sempre positiva) che associa alla dimensione del problema il costo della sua risoluzione in base alla misura scelta

$T_P(n)$ = Complessità con costo = tempo del programma P al variare di n

1.2 Ordine

$g(n)$ è di ordine $O(f(n))$ se esistono un intero n_0 ed una costante $c > 0$ tali che per ogni $n > n_0$ si ha $g(n) \leqslant c\, f(n)$

1.3 Regole

1.3.1 Regola dei fattori costanti

Per ogni costante positiva k, $O(f(n)) = O(k\, f(n))$

1.3.2 Regola della somma

Se $f(n)$ è $O(g(n))$, allora $f(n) + g(n)$ è $O(g(n))$

1.3.3 Regola del prodotto

Se $f(n)$ è $O(f1(n))$ e $g(n)$ è $O(g1(n))$, allora $f(n)g(n)$ è $O(f1(n)\, g1(n))$

1.3.4 Altre regole

Se $f(n)$ è $O(g(n))$ e $g(n)$ è $O(h(n))$, allora $f(n)$ è $O(h(n))$

Per ogni costante k, k è $O(1)$

Per $m \leqslant p$, n^m è $O(n^p)$

Un polinomio di grado m è $O(n^m)$

1.4 Teorema

Per ogni k, $n^k \in O(a^n)$, per ogni $a > 1$

Una qualsiasi funzione polinomiale ha minore complessità di una qualsiasi funzione esponenziale

2 Alberi binari

- NULL è un albero binario
- Un nodo p più due alberi binari Bs e Bd forma un albero binario

2.1 Numero di foglie e di nodi

Un albero binario bilanciato con livello k ha

- $2^{k+1} - 1$ nodi
- 2^k foglie

2.2 Alcuni algoritmi

2.2.1 Contare il numero dei nodi

```
int nodes(Node* tree) {
  if (!tree) return 0;
  return 1 + nodes(tree -> left) + nodes(tree -> right);
}
```

2.2.2 Contare il numero delle foglie

```
int leaves(Node* tree) {
  if (!tree) return 0;
  if (!tree -> left && !tree -> right) return 1;
  return leaves(tree -> left) + leaves(tree -> right);
}
```

2.2.3 Cercare un etichetta e resistuire un puntatore

Se il nodo non compare nell'albero, viene restituito NULL. Se l'albero contiene più di un'etichetta, viene restituito un puntatore al primo

```
Node* findNode(Infotype n, Node* tree) {
  // L'albero è vuoto, l'etichetta non può esserci
  if (!tree) return NULL;
  // Trovata l'etichetta, restituisco il puntatore
  if (tree -> label == n) return tree;
  // Cerco a sinistra
  Node* a = findNode(n, tree -> left);
  // Controllo se il puntatore della ricerca "a sinistra"
  // a resistuito qualcosa di interessante, altrimenti cerco a destra
  if (a) return a;
  else return findNode(n, tree -> right);
}
```

2.2.4 Eliminare tutto l'abero

Alla fine il puntatore deve essere NULL

```
void delTree(Node* &tree) {
  if (tree) {
    delTree(tree -> left);
    delTree(tree -> right);
    delete tree;
    tree = NULL;
  }
}
```

2.2.5 Inserire un nodo

Inserisce un nodo (son) come figlio di father, sinistro se $c = 'l'$ oppure destro se $c = 'r'$.

Ritorna 1 se l'operazione ha successo, 0 altrimenti. Se l'albero è vuoto, inserisce il nodo come radice

```
int insertNode(Node* &tree, InfoType son, InfoType father, char c) {
  // Caso in cui l'albero sia vuoto
  if (!tree) {
    tree = new Node;
    tree -> label = son;
    tree -> left = tree -> right = NULL;
    return 1;
  }
  // Caso normale
  // Effettuo la ricerca di father con la funzione
  // di ricerca nodo (vedi sopra)
  Node * a = findNode(father, tree);
  // Se il nodo non è stato trovato, restituisco 0 e mi fermo
  if (!a) return 0;
  // Inserimento come figlio sinistro
  if (c == 'l' && !a -> left) {
    a -> left = new Node;
    a -> left -> label = son;
    a -> left -> left = a -> left -> right = NULL;
    return 1;
  }
  if (c == 'r' && !a -> right) {
    a -> right = new Node;
    a -> right -> label = son;
    a -> right -> left = a -> right -> right = NULL;
    return 1;
  }
}
```

3 Alberi generici

- Un nodo p è un albero
- Un nodo più una sequenza di alberi $A_1 ... A_n$ è un albero

3.1 Alcuni algoritmi

3.1.1 Contare il numero di nodi

Vedi l'algoritmo per gli alberi binari

3.1.2 Contare il numero di foglie

```
int leaves(Node* tree) {
  if (!tree) return 0;
  // Caso della foglia
  if (!tree -> left) return 1 + leaves(tree -> right);
  // "Non caso" della foglia
  return leaves(tree -> left) + leaves(tree -> right);
```

3.1.3 Inserire un nodo in fondo ad una lista di fratelli

```
void addSon(InfoType x, Node* &tree) {
  // Caso in cui la lista sia vuota
  if (!tree) {
    tree = new Node;
    tree -> label = x;
    tree -> left = tree -> right = NULL;
  }
  else {
    addSon(x, tree -> right);
  }
}
```

3.1.4 Inserire un nodo son come ultimo figlio di father

Se l'albero è vuoto, lo inserisce come radice

```
int insert(InfoType son, InfoType father, Node* &tree) {
  if (!tree) {
    tree = new Node;
    tree -> label = son;
    tree -> left = tree -> right = NULL;
    return 1;
  }
  Node* a = findNode(father, tree);
  if (!a) return 0;
  addSon(son, a -> left);
  return 1;
}
```

4 Alberi binari di ricerca

Un albero binario di ricerca è un albero binario tale che per ogni nodo p

- I nodi del sottoalbero sinistro di p hanno etichetta minore dell'etichetta di p
- I nodi del sottoalbero destro di p hanno etichetta maggiore dell'etichetta p

4.1 Proprietà

- Non ci sono doppioni
- La visita simmetrica elenca le etichette in ordine crescente

4.2 Alcuni algoritmi

4.2.1 Cercare un nodo

```
Node* findNode(InfoType n, Node* tree) {
   if (!tree) return 0;
   if (n == tree -> label) return tree;
   if (n < tree -> label) {
```

```
      return findNode(n, tree -> left);
    }
    return findNode(n, tree -> right);
}
```

4.2.2 Inserire un nodo

```
void insertNode(InfoType n, Node* &tree) {
  // Albero vuoto
  if (!tree) {
    tree = new Node;
    tree -> label = n;
    tree -> left = tree -> right = NULL;
    return;
  }
  // Caso n < radice
  if (n < tree -> label) {
    insertNode(n, tree -> left);
  }
  if (n > tree -> label) {
    insertNode(n, tree -> right);
  }
}
```

L'algoritmo ha complessità $O(\log(n))$

[proseguire da pagina 129]