Appunti per la prova pratica di Algoritmi

Gianluca Mondini

Non mi ritengo responsabile per eventuali errori contenuti nella dispensa.

1 Vettori

```
#include <vector>
```

1.1 Esempi di costruttori

```
// Creazione di un vettore vuoto
std::vector<int> first;

// Inserimento di due valori
std::vector<int> second (4,100);

// Copia di un pezzo di second
std::vector<int> third(second.begin(), second.end());

// Copia di third
std::vector<int> fourth (third);

// Creazione a partire da un array
int myints[] = {12, 4, 532, 1, 4};
std::vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int);
```

1.2 Iterare su di un vettore

```
// Scrivo il contenuto di un vettore
for (std::vector<int>::iterator it = fifth.begin(); it != fifth.end(); i++) {
        std::cout << ' ' << *it;
}
std::cout << std::endl;
```

1.3 Modificare un vettore

```
// Inserimento in coda
myvector.push_back(3);

// Rimozione dalla coda
myvector.pop_back();
```

2 Stringhe

```
#include <string>
```

2.1 Esempi di costruttori

```
std::string s0 ("Hello world!");
```

2.2 Iterare su una stringa

```
std::string str ("Ciao mondo!");
```

```
for (std::string::iterator it = str.begin(); it!=str.end(); i++) {
        std::cout << *it;
}
std::cout << std::endl;
```

2.3 Funzioni utili

```
// Lunghezza
str.length();

// Appendere in fondo ad una stringa
std::string str;
std::ifstream file ("test.txt", std::ios::in);
if (file) {
        while (!file.eof()) str.push_back(file.get());
}
std::cout << str << std::endl;

// Appendere una stringa ad un'altra stringa
str2.append(str1);

// Cercare un elemento in una stringa
std::size_t found = str.find(str2);
if (found != std::string::npos) {
        std::cout << found << std::endl;
}

// Generare sottostringhe
std::string str = "Sono una stringa di prova";
// Parto dal char 3 e ne estraggo 5
std::string str2 = str.substr(3, 5);
// Effettuo una ricerca
str::size_t pos = str.find("una");
// Genero una stringa contente "una stringa di prova"
std::string str3 = str.substr(pos);
```

2.4 Conversione da *string* ad altri tipi

```
#include <stdlib.h>     /* atoi */
string stringa;
int intero = atoi(stringa.c_str())
```

2.5 Conversione da *string* ad altri tipi - Solo *C++ 11*

```
using namespace std;
string str_dec = "2001, A Space Odyssey";
string str_hex = "40c3";
string str_bin = "-1010101110101";
string str_auto = "0x7f";

string::size_type sz;

int i_dec = stoi (str_dec, &sz);
int i_hex = stoi (str_hex, nullptr, 16);
int i_bin = stoi (str_bin, nullptr, 2);
```

```
int i_auto = stoi (str_auto, nullptr, 0);
```

2.6 Inversione di una stringa

```
#include <algorithm>
string str = "ciao";
string reversed_str = str;
reverse(reversed_str.begin(), reversed_str.end());
cout << reversed_str << endl; // $ oaic
```

3 Alberi binari

4 Alberi binari

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

4.1 Numero di foglie e di nodi

Un albero binario bilanciato con livello k ha

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

4.2 Alcuni algoritmi

4.2.1 Contare il numero dei nodi

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

4.2.2 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);
}
```

4.2.3 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);
}
```

4.2.4 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;
  }
}
```

4.2.5 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;
  }
}
```

4.2.6 Compito 2 - Esercizio 4

Scrivere una funzione che, dato un albero binario ad etichette di tipo string, con puntatore alla radice t, restituisca come risultato il numero di nodi che hanno numero dispari di discendenti. Si calcoli la complessità della soluzione proposta in funzione del numero di nodi dell'albero.

```
int dispari(Node* t, int & nodi) {
  if (!t) {
    nodi = 0;
    return 0;
  }
  int cs, cd, nodis, nodid;
  cs = dispari(t -> left, nodis);
  cd = dispari(t -> left, nodid);
  nodi = nodis + nodid + 1;
  return cs + cd + ((nodis + nodid) % 2);
}
```

La funzione è $O(n)$

4.2.7 Compito 1 - Esercizio 4

Scrivere una funzione ricorsiva che, dato un albero binario a etichette intere, conta il numero di nodi che hanno più foglie maggiori o uguali a zero che minori di zero tra i propri discendenti.

```
int conta(const Node * t, int & pos, int & neg) {
  if (!t) {
    pos = 0;
    neg = 0;
    return 0;
  }
  if (!t -> left && !t -> right) {
    pos = (t -> info >= 0) ? 1 : 0;
    neg = (t -> info < 0)  ? 1 : 0;
    return 0;
  }
  int pos_left, pos_right;
  int neg_left, neg_right;
  int conta_left = conta(t -> left, pos_left, neg_left);
  int conta_right = conta(t -> right, pos_right, neg_right);
  pos = pos_left + pos_right;
  neg = neg_left + neg_right;
  return (pos > neg) ? 1 : 0 + conta_left + conta_right;
}
```

5 Alberi generici

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

5.1 Alcuni algoritmi

5.1.1 Contare il numero di nodi

Vedi l'algoritmo per gli alberi binari

5.1.2 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);
```

5.1.3 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);
  }
}
```

5.1.4 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;
}
```

5.1.5 Compito 1 - Esercizio 5

Scrivere una funzione che, dato un albero generico a etichette intere e memorizzazione figlio-fratello, conta il numero di nodi che hanno più figli maggiori o uguali a zero che minori di zero.

```
int conta(const Node* t) {
  if (!t) return 0;
  const Node * n;
  int pos = 0, neg = 0;
  for (n = t -> left; n != NULL; n = n -> right) {
    if (n -> info >= 0) pos++;
    else neg++;
  return (pos > neg) ? 1 : 0 + conta(t -> left) + conta(t -> right);
}
```

6 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

6.1 Proprietà

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

6.2 Alcuni algoritmi

6.2.1 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);
}
```

6.2.2 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 $O(\log(n))$

6.2.3 Restituire l'etichetta del nodo più piccolo di un albero ed eliminare il nodo che la contiene

```
void deleteMin(Node *& tree, InfoType &m) {
  if (tree -> left) // C'è un nodo più piccolo
    deleteMin(tree -> left, m);
  else {
    m = tree -> label; // restituisco l'etichetta
    Node * a = tree;
    // connetto il sottoalbero destro di
    // m al padre di m
    tree = tree -> right;
    // elimino il nodo
    delete a;
  }
```

```
}
```

6.2.4 Cancellare un nodo ?

```
void deleteNode(InfoType n, Node *& tree) {
  if (tree) {
    // n è minore della radice
    if (n < tree -> label) {
      deleteNode(n, tree -> left);
      return;
    }
    // n è maggiore della radice
    if (n > tree -> label) {
      deleteNode(n, tree -> right);
      return;
    }
    // n non ha figlio sinistro
    if (!tree -> left) {
      Node * a = tree;
      tree = tree -> right;
      delete a;
      return;
    }
    // n non ha figlio destro
    if (!tree -> right) {
      Node * a = tree;
      tree = tree -> left;
      delete a;
      return;
    }
    // n ha entrambi i figli
    deleteMin(tree -> right, tree -> label);
}
```

Questo algoritmo ha complessità $O(\log(n))$

7 Heap

Un heap (binario) è una struttura dati composta da un array che possiamo considerare come un albero binario quasi completo. Ogni nodo dell'albero corrisponde a un elemento dell'array. Tutti i livelli dell'albero sono compeltamente riempiti, tranne eventualmente l'ultimo l'ultimo che può essere riempito da sinistra fino ad un certo punto.

Definizione alternativa

Un heap è un albero binario quasi bilanciato con le seguenti proprietà:

- i nodi dell'ultimo livello sono addossati a sinistra
- in ogni sottoalbero l'etichetta della radice è maggiore o uguale a quella di tutti i discendenti

L'heap viene memorizzato in un array

7.1 Calcolare le parentele

- Figlio sinistro di $i = 2\,i + 1$
- Figlio destro di $i = 2\,i + 2$
- Padre di $i = \frac{i-1}{2}$

7.2 Classe Heap

```
class Heap {
  private:
    int * h;
    int last;
    void up(int);
    void down(int);
    void exchange(int i, int j);
  public:
    Heap(int);
    ~Heap();
    void insert(int);
    int extract();
};
```

7.2.1 Costruttore

```
Heap::Heap(int n) {
  h = new int[n];
  last = -1;
}
```

7.2.2 Distruttore

```
Heap::~Heap() {
  delete h [n];
}
```

7.2.3 Inserimento

- Memorizza l'elemento nella prima posizione libera dell'array
- Fa risalire l'elemento tramite scambi figlio-padre per mantenere la proprietà dello heap

```
void Heap::insert(int x) {
  h[++last] = x;
  up(last);
}

// i è l'indice dell'elemento da far risalire
void Heap::up(int i) {
  // Se non sono sulla radice
  if (i > 0) {
    // Se l'elemento è maggiore del padre
    if (h[i] > h[(i-1)/2]) {
      // Scambia il figlio con il padre
      exchange(i, (i-1)/2);
      // chiama up sulla nuova posizione
      up((i-1)/2);
    }
  }
}
```

La funzione termina in due casi:

- viene chiamata con l'indice 0 (radice)
- L'elemento è inferiore al padre

La complessità è $O(\log(n))$ perché ogni chiamata risale di un livello

7.2.4 Estrazione

- Restituisce il primo elemento dell'array
- Mette l'ultimo elemento al posto della radice e decrementa last
- Fa scendere l'elemento tramite scambi padre-figlio per mantenere la proprietà dello heap

```
int Heap::extract() {
  int r = h[0];
  h[0] = h[last--];
  down(0);
  return r;
}

// i è l'indice dell'elemento da far scendere
void Heap::down(int i) {
  // son = indice del figlio sinistro (se esiste)
  int son = 2*i+1;
  // se i ha un solo figlio (è l'ultimo dell'array)
  if (son == last) {
    // se il figlio è maggiore del padre
    if (h[son] > h[i]) {
      // fai lo scambio, altrimenti termina
      exchange(i, last);
    }
  }
  // se i ha entrambi i figli
  else if (son < last) {
    // son = indice del figlio maggiore tra i due
    if (h[son] < h[son+1]) son++;
    // se il figlio è maggiore del padre
    if (h[son] > h[i]) {
      // fai lo scambio
      exchange(i, son);
      // chiama down sulla nuova posizione
      down(son);
      // altrimenti termina (termina anche se i non ha figli)
    }
  }
}
```

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

8 Ricerca hash

8.1 Alcuni algoritmi

8.1.1 Ricerca tramite hash

```
bool hashSearch (int* A, int k, int x) {
  int i = h(x);
  if (A[i] == 1) return true;
  else return false;
}
```

8.2 Metodo hash ad accesso non diretto

È possibile rilasciare l'iniettività e permettere che due elementi diversi abbiano lo stesso indirizzo hash. Si ha una collisione quando

$$h(x_1) = h(x_2)$$

Bisogno gestire le seguenti situazioni:

- Come cercare un elemento se il suo posto è occupato da un altro
- Come inserire gli elementi

8.2.1 Soluzione: hash modulare

Si scrive una funzione $h()$

$$h(x) = (x \ \% \ k)$$

In modo tale da essere sicuri di generare tutti e soli gli indici dell'array

Legge di scansione lineare Se non si trova l'elemento al suo posto, lo si cerca nelle posizioni successive fino a trovarlo o ad incontrare una posizione vuota.

L'inserimento è fatto con lo stesso criterio

Agglomerato Gruppo di elementi con indirizzi hash diversi (?)

La presenza di collisioni ed agglomerati aumenta il tempo di ricerca

Funzione di ricerca con scansione lineare

```
bool hashSearch(int *A, int k, int x) {
  int i = h(x);
  for (int j=0; j<k; j++) {
    int pos = (i+j) % k;
    if (A[pos] == -1) return false;
    if (A[pos] == x) return true;
  }
  return false;
}
```

Funzione di insierimento in presenza di cancellazioni

```
int hashInsert(int *A, int k, int x) {
  int i = h(x);
  for (int j=0; j < k; j++) {
    int pos = (i+j) % k;
    // -1: posizione vuota
    // -2: posizione disponibile
    if ((A[pos] == -1) || (A[pos] == -2)) {
      A[pos] = x;
      return 1;
    }
  }
  return 0;
}
```

9 Grafi

$$\text{Grafo orientato}=(N, A)$$

dove

- $N =$ insieme di nodi
- $A \subseteq N \times N =$ insieme di archi (coppie ordinate di nodi)

Un grafo orientato con n nodi ha al massimo n^2 archi

9.1 Rappresentazioni in memoria di grafi

9.1.1 Tramite liste di adiacenza

```
struct Node {
  int NodeNumber;
  Node * next;
};

Node* graph[N];
```

9.1.2 Tramite matrici di adiacenza

```
int graph[N][N];
```

9.2 Rappresentazioni in memoria di grafi con nodi ed archi etichettati

9.2.1 Tramite liste di adiacenza

```
struct Node {
  int NodeNumber;
  ArcType archLabel;
  Node * next;
};

Node * graph[N];

NodeType nodeLabels[N];
```

NodeType = char
ArcType = int

9.2.2 Tramite matrici di adiacenza

```
ArcType graph[N][N];
NodeType nodeLabels[N];
```

9.3 Cammino più breve

9.3.1 Algoritmo di Dijkstra

- Si applica ai grafi orientati
- Trova i cammini minimi **da un nodo a tutti gli altri**
- È basato sulla metodologia **greedy**

Vengono utilizzate due tabelle:

- **dist** (distanza)
- **pred** (predecessore)

con n elementi.

Vengono eseguiti n passi, e ad ogni passo:

1. Si sceglie il nodo con distanza minore in **dist**
2. Si aggiornano **pred** e **dist** per i suoi immediati successori

10 Templates

10.1 Esempi di definizione

```
template <class T1>
T1 somma(T1 a, T2 b) {
  return a + b;
}
```

10.2 Parametri costanti

10.3 Variabili statiche

Ogni "tipo di funzione" possiede la propria variabile statica indipendente

```
template<class Data>
void typeCounter(Data a) {
        static int c;
        c++;
        cout << "Il counter vale " << c << endl;
}
```

11 Derivazione

11.1 Specificatori d'accesso

- I campi **privati** di una classe non sono accessibili dalle sottoclassi nè dall'esterno
- I campi **protetti** di una classe sono accessibilit dalle sottoclassi, ma non dall'esterno
- I campi **pubblici** di una classe sono accessibili anche dall'esterno
- I campi privati, protetti e pubblici rimangono tali in tutta la gerarchia

11.2 Costruzione degli oggetti

Quando un oggetto viene costruito si costruisce prima la parte **BASE** ed in seguito quella **DERIVATA**.

Viene quindi prima chiamato il costruttore della classe base e poi quello della classe derivata.

Se la classe base ha dei costruttori, il costruttore di una classe derivata deve chiamarne uno nella lsita di inizializzazione. Può non chiamarlo esplicitamente se la classe base ha un costruttore di default, che in questo caso viene chiamato automaticamente.

12 Complessità di un algoritmo

12.1 Dalla relazione di ricorrenza alla complessità

- $T(0) = a$
- $T(n) = b + T(n-1)$

$$T(n) \in O(n)$$

- $T(1) = a$

- $T(n) = b\,n + T(n-1)$

$$T(n) \in O(n^2)$$

- $T(1) = a$
- $T(n) = b\,n + 2\,T(n/2)$

$$T(n) \in O(n \log(n))$$

- $T(0) = a$
- $T(n) = b + T(n-1)$

$$T(n) \in O(n)$$

- $T(0) = a$
- $T(n) = b + T(n/2)$

$$T(n) \in O(\log(n))$$

- $T(0) = a$
- $T(n) = b + 2\,T(n/2)$

$$T(n) \in O(n)$$

- $T(1) = a$
- $T(n) = b + 2\,T(n-1)$

$$T(n) \in O(2^n)$$

- $T(0) = d$
- $T(n) = b\,n^k + T(n-1)$

$$T(n) \in O(n^{k+1})$$

Esempio: fibonacci() ricorsivo

- $T(0) = T(1) = d$
- $T(n) = b + T(n-1) + T(n-2)$

$$T(n) \in O(2^n)$$

Esempio: fibonacci() non ricorsivo

- $T(0) = d$
- $T(n) = b + T(n-1)$

$$T(n) \in O(n)$$

Esempio: mergeSort()

- $T(0) = T(1) = d$
- $T(n) = b\,n + 2\,T(n/2)$

$$T(n) \in O(n \log(n))$$

Esempio: split()

- $T(0) = T(1) = d$
- $T(n) = b + T(n-2)$

$$T(n) \in O(n)$$

Esempio: merge()

- $T(0) = d$
- $T(n) = b + T(n-1)$

$$T(n) \in O(n)$$