Capitolo 1 ~ Tipi di dato e strutture di dati

Introduzione. Il **dato** è un valore che una variabile può assumere, il **tipo di dato** è un modello matematico che consiste in una collezione di valori sui quali sono ammesse certe operazioni. Alcuni linguaggi di programmazione forniscono direttamente alcuni tipi di dato chiamati *tipi di dati primitivi*. È importante distinguere tra i dati stessi e la loro rappresentazione. In generale, *le proprietà che caratterizzano un tipo di dato dipendono esclusivamente dalla specifica del tipo di dato stesso mentre devono essere indipendenti dal modo in cui i dati sono rappresentati*. Si parla allora di un **tipo di dato astratto** dove l'astrazione è proprio rispetto alla rappresentazione del tipo di dato. Una corretta specifica equivale ad un buon manuale d'uso chiaro, conciso e non ambiguo, definendo le regole da seguire per l'utilizzo e la creazione del suddetto tipo di dato.

Strutture di dati: specifica e realizzazione. Accade spesso che i dati da elaborare siano riuniti in agglomerati detti strutture di dati. Una **struttura di dati** è uno speciale tipo di dato, caratterizzata più dall'organizzazione imposta agli elementi che la compongono, che dal tipo degli elementi stessi. Il tipo degli elementi di una struttura può essere assunto addirittura parametrico. Una struttura di dati consiste in un modo sistematico di organizzare i dati, in un insieme di operatori che permettono di manipolare elementi della struttura o di aggregare elementi. *È comodo fornire una classificazione delle strutture di dati in base alle caratteristiche presentate dalla disposizione dei dati e dal loro numero e dal loro tipo* (lineari, non lineari, a dimensione fissa o variabile, omogenee o non omogenee).
La **specifica sintattica** è la notazione con cui sono indicati sia i tipi di dato che le operazioni, fornisce l'elenco dei nomi dei tipi di dato utilizzati per definire la struttura delle operazioni specifiche della struttura stessa e delle costanti; la **specifica semantica** è il significato associato ai tipi di dato, associa un insieme matematico ad ogni nome di tipo introdotto nella specifica sintattica. La funzione è definita specificando esplicitamente una coppia di condizioni dette **precondizione** (*stabilisce quando l'operatore è applicabile, se la precondizione manca l'operatore è sempre applicabile*) e **postcondizione** (*indica come il risultato sia vincolato agli argomenti dell'operatore*) sui domini di partenza.

Rappresentazione in memoria. Nel valutare l'efficienza di procedure che utilizzino tipi di dati primitivi si prescinde solitamente dalle caratteristiche della macchina, o meglio, si assume una organizzazione abbastanza generica che è soddisfatta da ogni macchina ragionevole: i dati sono contenuti in memoria; la memoria è divisa in celle ognuna di uguale lunghezza; i dati elementari non eccedono la lunghezza di una cella; si accede alle celle mediante indirizzo; gli indirizzi sono interi consecutivi.

Puntatori. I **puntatori** permettono un accesso indiretto ad un dato mediante l'indirizzo della cella di memoria a partire dal quale il dato stesso è memorizzato. Con una dichiarazione di tipo puntatore è associata alla variabile una cella per contenere un indirizzo ma non è associata nessuna cella per contenere un dato del tipo puntato. Pertanto per utilizzare un dato di tipo puntato occorre acquisire successivamente una cella di memoria per un dato tipo e conservarne l'indirizzo nella variabile di tipo puntatore. I puntatori sono considerate variabili dinamiche poiché le celle di memoria indirizzate possono variare durante l'esecuzione.

Capitolo 2 ~ Le liste

Le liste. Una **lista** è una sequenza di elementi di un certo tipo in cui è possibile aggiungere o togliere elementi. Per fare questo occorre specificare la posizione relativa all'interno della sequenza nella quale il nuovo elemento va aggiunto o dalla quale il vecchio elemento va tolto. *È di dimensione variabile e si può accedere direttamente solo al primo ed ultimo elemento, per qualsiasi altro elemento occorre scandire tutta la lista.* Una lista ha lunghezza pari al numero dei suoi elementi, se la lista non ha nessun elemento è detta vuota. Ogni elemento ha una posizione nella lista, per comodità si prendono in considerazione anche una posizione antecedente il primo elemento ed una posteriore all'ultimo elemento. Non sempre la posizione corrisponde al numero dell'elemento (dipende dall'implementazione).

Specifica. Il tipo di dato lista comprende le operazioni di CREALISTA (per creare una nuova lista anche vuota), PRIMOLISTA ed ULTIMOLISTA (per selezionare il primo ed ultimo elemento della lista), SUCCLISTA e PREDLISTA (per scandire in avanti o indietro ogni elemento della lista), LISTAVUOTA o FINELISTA (per sapere se si è alla fine della sequenza o per sapere se la lista è vuota), LEGGILISTA (per

leggere il valore dell'elemento), SCRIVILISTA (per scrivere il valore di un elemento), INSLISTA (per inserire un nuovo elemento nella lista) e CANCLISTA (per eliminare un elemento della lista).

```
typedef struct cella * lista;
typedef lista          posizione;
typedef int            tipoelem;

struct cella {
  posizione precedente;
  int       elemento;
  posizione successivo;
};

lista CREALISTA () {
  lista L;
  L = (lista) malloc(sizeof (struct cella) );
  L->successivo = L;
  L->precedente = L;
  return L;
}

int LISTAVUOTA (lista L) {
  int listavuota;
  listavuota = ((L->successivo == L) && (L->precedente == L)) ? 1 : 0;
  return listavuota;
}

posizione PRIMOLISTA (lista L) {
  return L->successivo;
}

posizione ULTIMOLISTA (lista L) {
  return L->precedente;
}

posizione SUCCLISTA (posizione p) {
  return p->successivo;
}

posizione PREDLISTA (posizione p) {
  return p->precedente;
}

int FINELISTA (posizione p, lista L) {
  int finelista;
  finelista = (p == L) ? 1 : 0;
  return finelista;
}

tipoelem LEGGILISTA (posizione p) {
  return p->elemento;
}

void SCRIVILISTA (int a, posizione p) {
  p->elemento = a;
}

void INSLISTA (int a, posizione * p) {
  struct cella * tmp;

  tmp = (struct cella *) malloc(sizeof(struct cella));
```

```
  tmp->precedente = (*p)->precedente;
  tmp->successivo = (*p);
  tmp->elemento   = a;

  (*p)->precedente->successivo = tmp;
  (*p)->precedente             = tmp;

  (*p) = tmp;
}

void CANCLISTA (posizione * p) {
  posizione tmp;

  tmp = (*p);

  (*p)->precedente->successivo = (*p)->successivo;
  (*p)->successivo->precedente = (*p)->precedente;

  (*p) = (*p)->successivo;

  free(tmp);
}
```

In Pratica...

Data una lista di numeri interi, calcolare la somma di tutti gli elementi.

```
int sumlist (lista L) {
   posizione p = PRIMOLISTA (L);
   int sum = 0;
   while (!FINELISTA (p, L)) {
      sum +=LEGGILISTA (p, L);
      p = SUCCLISTA (p, L);
   }
   return sum;
}
```

Data una lista di numeri naturali costruire due liste contenenti rispettivamente i numeri pari e i numeri dispari

```
void pardis (lista L) {
   lista Pari, Dispari;
   lista Pari = CREALISTA ();
   lista Dispari = CREALISTA ();
   posizione pL = PRIMOLISTA (L);
   posizione pD = PRIMOLISTA (Dispari);
   posizione pP = PRIMOLISTA (Pari);
   while (!FINELISTA (pL, L)) {
      temp = LEGGILISTA (pL, L);
      if (temp %2 == 0) INSLISTA (temp, pP, Pari);
      else INSLISTA (temp, pD, Dispari);
      SUCCLISTA (pL, L);
   }
}
```

Realizzazione di una lista. Esistono diversi modi per ottenere una struttura di tipo lista. Il primo è mediante **vettore**, ma è da bocciare subito poiché il vettore è statico e limitato e nonostante la facilità intrinseca nell'accedere ad un qualsiasi suo elemento, cancellare ed aggiungerne di nuovi implica operazioni macchinose e complesse. È possibile quindi creare liste mediante:

- ✔ **Puntatori.** L'idea di base è di memorizzare una lista di elementi in altrettanti record usualmente detti celle, in modo che l'i-esima cella contenga il valore dell'i-esimo elemento della lista e l'indirizzo della cella (puntatore) contenente l'elemento successivo (nel caso *monodorezionale*) o della cella precedente e successiva (nel caso *bidirezionale*). Entrambe le scelte possono essere rese *circolari* facendo puntare l'ultima cella alla prima e viceversa.
- ✔ **Cursori.** Nei linguaggi che non forniscono puntatori è possibile simularsi utilizzando dei *cursori*, ovvero con variabili intere il cui valore è interpretato come un indice di un vettore. Il vettore simula la memoria disponibile per i puntatori che deve essere gestita esplicitamente.

Per rendere più leggibile il codice delle funzioni INSLISTA e CANCLISTA quando operano sugli elementi estremi della lista, è conveniente introdurre una ulteriore cella, che non contiene alcuna informazione valida, detta **sentinella**.

In Pratica...

Realizzazione di una lista bidimensionale con sentinella.

```
typedef int tipoelem;
typedef struct cella * lista;
typedef struct cella *posizione;

struct cella {
   posizione precedente;
   posizione successivo;
   tipoelem elemento;
}

lista CREALISTA () {
  lista L ;
  L = malloc ( sizeof ( struct cella )) ;
  L->successivo = L;
  L->precedente = L;
  return L ;
}

int LISTAVUOTA (lista L) {
  int listavuota ;
  listavuota = ((L->successivo == L) && (L->precedente == L)) ? 1 : 0;
  return listavuota ;
}

posizione PRIMOLISTA (lista L) {
   return L->successivo ;
}

posizione ULTIMOLISTA (lista L) {
   return L->precedente ;
}

posizione SUCCLISTA (posizione p) {
   return p->successivo ;
}

posizione PREDLISTA (posizione p) {
   return p->precedente ;
}

int FINELISTA (posizione p, lista L) {
   return ((p == L) ? 1 : 0);
}

int LEGGILISTA (posizione p) {
```

```
   return p->elemento ;
}

void SCRIVILISTA (int a, posizione p) {
    p->elemento = a;
}

void INSLISTA (int a, posizione * p) {
   posizione  tmp ;
   tmp = malloc(sizeof(struct cella ));
   tmp->elemento   = a;
   tmp->precedente = (*p)->precedente;
   tmp->successivo = (*p);
   ((*p)->precedente)->successivo = tmp;
   (*p)->precedente=tmp ;
   (*p)=tmp;
}

void CANCLISTA (posizione * p) {
   posizione  tmp ;
   tmp = *p;
   ((*p)->precedente)->successivo    = (*p)->successivo ;
   ((*p)->successivo)->precedente    = (*p)->precedente ;
    *p = (*p)->successivo;
   free(tmp);
}
```

Implementare la funzione APPARTIENELISTA.

```
int APPARTIENELISTA (int a, lista L) {
   posizione p ;
   int found;
   p = PRIMOLISTA(L);
   found = 0;
   while ( ! found && ! FINELISTA(p, L) ) {
      if ( p->elemento == a ) {
         found = 1;
      } else {
         p = SUCCLISTA(p);
      }
   }
return found ;
}
```

Capitolo 3 ~ Pile e Code

Pile e code. Una **pila** è una sequenza di elementi di un certo tipo in cui è possibile aggiungere o togliere elementi soltanto ad un estremo della sequenza, che di solito è detta *testa*. Può essere intesa come un caso speciale di lista in cui l'ultimo elemento inserito è anche il primo ad essere estratto e non è possibile accedere a nessun elemento che non sia quello di testa. Tale meccanismo è detto LIFO (Last In First Out). Una **coda** invece è una sequenza di elementi di un certo tipo in cui è possibile aggiungere elementi ad un estremo detto *fondo* e togliere elementi da un altro estremo detto *testa*. Può essere considerata come un caso particolare di lista in cui il primo elemento inserito è il primo elemento ad essere rimosso e non è possibile accedere a nessun elemento che non sia quello di fondo o quello di testa. Tale meccanismo è detto FIFO (First In First Out).

Specifiche. Le operazioni per il tipo di dato **pila** sono CREAPILA (per creare ed inizializzare la pila vuota), PILAVUOTA (per sapere se la pila è vuota), LEGGIPILA (per leggere il primo elemento della pila), FUORIPILA (per eliminare il primo elemento della pila) e INPILA (per inserire un elemento in

testa alla pila). Le operazioni per il tipo di dato **coda** sono analoghe: CREACODA, CODAVUOTA, LEGGICODA, FUORICODA e INCODA.

Realizzazione di una pila. Esistono diversi metodi per realizzare una struttura pila:

- ✔ **Vettore.** Realizzare una pila con un vettore consiste nel memorizzare tutti gli elementi della pila, in ordine inverso, nelle prime posizioni di un vettore lungo quanto sono numerosi gli elementi della pila (meglio se più lungo), mantenendo il cursore in testa alla pila. La pila vuota è individuata dal valore 0 contenuto nel cursore mentre quella satura riporta la lunghezza totale del vettore. Ogni operazione della pila richiede tempo costante. Per ottenere lo stesso effetto con una cosa si usa un vettore circolare.

Realizzazione di una coda. Esistono diversi modi di realizzare una struttura di dati a coda:

- ✔ **Vettore circolare.** Si suppone di avere un vettore di tanti elementi quanto è la sua lunghezza massima, calcolato da 0 al massimo meno uno. L'elemento 0 è il successore dell'ultimo elemento. La cosa è contenuta in tante locazioni consecutive del vettore quanti sono gli elementi.

Pile e procedure ricorsive. Una importante applicazione delle pile sta nella gestione dell'esecuzione di procedure ricorsive (che quindi chiamano se stesse al loro interno). Tale meccanismo prevede il salvataggio dei dati sui quali lavora la procedura al momento di una nuova chiamata ricorsiva interna e la loro estrazione quando la procedura interna termina.

Capitolo 4 ~ Alberi

Alberi. Un **albero ordinato** è formato da un insieme finito di elementi, detti *nodi*. Se tale insieme non è vuoto allora un particolare nodo è designato come *radice* ed i rimanenti nodi, se esistono, sono partizionati a loro volta in insiemi disgiunti ciascuno dei quali è un albero ordinato. Ogni albero è costituito da un insieme di sottoalberi aventi come radici i nodi figli della radice dell'albero principale. Ogni nodo che ha un nodo padre è detto nodo figlio di quel padre. Un nodo che non ha figli è un nodo foglia. Tutti i nodi hanno un padre tranne il nodo radice. Tutti i nodi che appartengono alla stessa "generazione" si dicono nodi dello stesso livello (i livelli si contano a partire da 0 che è quello del nodo radice). I nodi si disegnano con cerchietti e le parentele con linee che congiungono tali cerchietti. Gli alberi sono solitamente rappresentati rovesciati con la radice in alto e le foglie in basso.

Specifica. Una lista è una sottospecie di albero, per cui le operazioni che vigono sulle liste si adattano perfettamente a quelle necessarie per gli alberi: CREAALBERO (crea un nuovo albero vuoto), RADICE (accede direttamente alla radice), PADRE, PRIMOFIGLIO e SUCCFRATELLO (per accedere al nodo padre, al primo nodo figlio o al nodo fratello), ALBEROVUOTO (per sapere se l'albero è popolato), FOGLIA e FINEFRATELLI (per sapere se si è in un nodo foglia o in un nodo che è l'ultimo figlio di un altro nodo), INSRADICE (per inserire una nuova radice), CANCSOTTOALBERO e INSSOTTOALBERO (per cancellare ed inserire un nuovo sottoalbero).

Visite. La **visita di un albero** ordinato consiste nel seguire una rotta di viaggio che consenta di esaminare ogni nodo dell'albero esattamente una volta. La visita può essere eseguita in vari modi, detti *ordini*, tra cui i tre più importati sono:

- ✔ *Ordine anticipato* o **previsita**. Consiste nell'esaminare la radice e poi effettuare nell'ordine la previsita di ogni sottoalbero della radice.
- ✔ *Ordine differito* o **postvisita**. Consiste nell'effettuare nell'ordine la postvisita di ogni sottoalbero della radice per poi esaminare la radice per ultima.
- ✔ *Ordine simmetrico* o **invisita**. Consiste nell'effettuare nell'ordine la invisita di ogni sottoalbero fino ad un dato valore fissato, per poi passare a visitare la radice e riprendere l'invisita di ogni sottoalbero dal valore fissato+1 in avanti.

In tutte e tre le procedure viene effettuata una chiamata ricorsiva su ogni nodo.

Realizzazione di un albero. Si può realizzare una struttura ad albero in diverse maniere:

- ✔ **Vettore dei padri.** Si suppone che i vettori dei nodi di un albero siano numerati. Si usa quindi un vettore di lunghezza pari al numero dei nodi dell'albero contenete, per ogni nodo, il cursore al nodo padre del nodo rappresentato dalla locazione nel vettore. È facile allora

visitare i nodi di figlio in padre, ma è difficile fare l'inverso.

- ✔ **Liste di figli.** Si ha un albero con tutti i nodi numerati. Si può mantenere un vettore contenente per ogni nodo dell'albero, un puntatore od un cursore ad una lista di tutti i figli di quel nodo. È facile e veloce scorrere tutti i figli di un nodo ma è difficile risalire al padre di un nodo od al fratello successivo.
- ✔ **Puntatori padre/primo figlio/fratello.** Si fa in modo che ogni nodo dell'albero contenga le indicazioni (sotto forma di puntatori o cursori) su quali sono il proprio padre, il proprio fratello successivo ed il primo figlio. Sarà facile allora ricavare queste informazioni interrogando la struttura del nodo. Tale realizzazione fa assomigliare il più possibile gli alberi alle liste ordinate. Si aggiungono dunque nuove operazioni che sono LEGGINODO (per leggere le informazioni di un nodo) e SCRIVINODO (per scrivere in un nodo).

Alberi binari. Un **albero binario** è un particolare albero ordinato in cui ogni nodo ha al più due figli e si fa distinzione tra il figlio sinistro ed in figlio destro di un nodo (il fatto che i due figli si mescolino genera alberi diversi da quello di partenza). Per interrogarlo e manipolarlo esistono operazioni specifiche tra cui BINALBEROVUOTO (per sapere se è un albero vuoto), CREABINALBERO (per creare un nuovo albero), BINRADICE, BINPADRE, CANCSOTTOBINALBERO, LEGGINODO, SCRIVINODO, del tutto simili alle operazioni per gli alberi ordinati.

Capitolo 5 ~ Insiemi

Insiemi. Un **insieme** è una collezione o famiglia di elementi distinti, detti anche *componenti* o *membri*, dello stesso tipo. L'insieme è la struttura matematica fondamentale, e può essere descritto o elencandone tutti gli elementi o stabilendo una priorità che ne caratterizzi gli elementi. A differenza delle liste, gli elementi di un insieme non hanno posizioni che li renda riconoscibili.
La relazione fondamentale di un insieme è quella di APPARTENENZA, da questa deriva l'INCLUSIONE. Le operazioni principali degli insiemi sono UNIONE (gli elementi di tutti gli insiemi che vi partecipano diventano gli elementi di un insieme formato dall'unione di tali insiemi), INTERSEZIONE (appartengono all'intersezione solo gli elementi in comune di due o più insiemi) e DIFFERENZA (tolti elementi da un insieme, la differenza è data da quelli che restano).

Specifica. SI definisce una struttura dati insieme tramite le seguenti specifiche: CREAINSIEME (per creare un nuovo insieme), INSIEMEVUOTO (per determinare se un insieme è vuoto), APPARTIENE, UNIONE, INTERSEZIONE (rispettivamente per simulare le operazioni base di un insieme), INSERISCI (per inserire nuovi elementi nell'insieme), CANCELLA (per eliminare un elemento dall'insieme).

Realizzazione di un insieme. È possibile realizzare insiemi in differenti maniere:

- ✔ **Vettore booleano.** Un insieme può essere rappresentato da un vettore booleano di un valore massimo di bit in cui ogni bit risulta vero se il relativo elemento appartiene all'insieme, falso altrimenti. È semplice così verificare se un elemento appartiene o meno ad un dato insieme, oppure inserire o cancellare un elemento.
- ✔ **Liste non ordinate.** Si può realizzare un insieme tramite una lista i cui elementi sono quelli dell'insieme. In questo modo l'insieme non deve necessariamente essere composto da interi compresi tra 1 e N. inoltre l'occupazione di memoria dell'insieme cresce linearmente col numero di elementi effettivamente presenti nell'insieme.
- ✔ **Liste ordinate.** Se sugli elementi dell'insieme è definita una relazione di ordinamento totale, l'insieme stesso può essere rappresentato con una lista ordinata per valori crescenti degli elementi. Realizzando le liste ordinate in modo da poter accedere direttamente anche all'ultimo elemento anche la ricerca dell'elemento massimo è facilitata.

Capitolo 6 ~ Code con priorità

Code con priorità. Una coda con priorità è un particolare insieme, sugli elementi del quale è definita una relazione di "minore" di ordinamento totale, in cui è possibile inserire un nuovo elemento o estrarre un elemento "minimo". La coda di elementi deve rispettare sempre la priorità dei suoi elementi.

Specifica. La specifica è simile a quella del tipo di dato insieme in cui sono ammesse, oltre a CREA

(per creare una cosa), anche INSERISCI (per inserire un elemento), MIN (per individuare l'elemento minore) e CANCELLAMIN (per cancellare l'elemento minore).

Realizzazione di una coda con priorità. È possibile realizzare code con priorità di n elementi utilizzando anche liste ordinate o non ordinate.

- ✔ **Heap.** Gli elementi della coda con priorità possono essere disposti in un vettore che denominiamo heap che può essere interpretato come un albero binario. Ciascun nodo dell'albero corrisponde ad una posizione del vettore che memorizza un elemento della coda.

In Pratica...

23 Marzo 2010 :: *Dire le condizioni necessarie e sufficienti affiché un vettore V di n elementi rappresenti un heap binario.*

Gli elementi di una cosa con priorità possono essere memorizzati in un vettore detto **heap** che può essere interpretato come un albero binario in cui ciascun nodo rappresenta una posizione del vettore che a sua volta memorizza un elemento della cosa. Ogni nodo contiene un elemento maggiore o minore di quello contenuto nel padre. Contiene esattamente 2^h-1 nodi non foglia quindi l'albero è quasi perfettamente bilanciato fino al livello h-1. Tutte le foglie di livello h sono addossate a sinistra.

23 Marzo 2010 :: *Scrivere una funziona booleana VERIFICA che verifica in tempo ottimo se un vettore V rappresenta un heap binario. Definire l'ordine di complessità omicron della procedura.*

```
boolean verifica-heap (int v[], int n) {
   int flag = true;
   int i = 0;
   while (flag && (i<n/2)) {
      if ((2*i+2)>n) flag = (v[i] < v[2*i+1]);
      else flag = (v[i] < v[2*i+1]) && (v[i] < v[2*i+2]);
      i = i+1;
   }
   return flag;
}
```

ass

Capitolo 7 ~ Alberi bilanciati di ricerca

Alberi bilanciati di ricerca. Un **albero binario di ricerca** è una struttura dati che rispetta le seguenti proprietà: *per ogni nodo tutti gli elementi contenuti nel sottoalbero radicato nel figlio sinistro di tal nodo sono minori dell'elemento contenuto nel nodo* e viceversa *gli elementi del sottoalbero radicato nel figlio destro sono tutti maggiori dell'elemento del nodo di partenza.*

Capitolo 8 ~ Grafi

Grafi. Un **grafo orientato** (o *grafo diretto* o *digrafo*) è una coppia G=(N,A) dove N è un inseme finito di elementi detti **nodi** ed A è un insieme finito di coppie ordinate di nodi detti **archi**. I grafi orientati possono essere rappresentati graficamente disegnando ogni nodo con un cerchietto e ogni arco con una freccia che esce dal nodo di partenza verso il nodo di destinazione. Il numero di nodi e archi è indicato. In un grafo orientato un cammino è *una sequenza di nodi* che se non si ripetono danno luogo ad un cammino semplice, se il primo nodo coincide con l'ultimo il cammino è chiuso. Se gli archi di un grafo sono formati da coppie non ordinate il grafo è non orientato. Nel **grafo non orientato** il cammino corrisponde alla catena. Il grafo non orientato è un albero libero.

Specifica. Per realizzare un grafo ci serviamo delle operazioni CREAGRAFO (per creare il grafo), GRAFOVUOTO (per verificare che il grafo sia vuoto), INSNODO (per inserire un nuovo nodo), INSARCO (per inserire un nuovo arco), CANCNODO e CANCARCO (per cancellare nodi ed archi), ADIACENTI (per verificare che due nodi siano adiacenti).

Realizzazione di un grafo. Esistono diverse modalità di realizzazione di un grafo:

- ✔ **Matrici di incidenza.** La matrice di incidenza è molto utile per rappresentare un grafo in certi problemi di programmazione lineare nei quali si cerca un vettore di un certo numero di incognite associate agli archi che soddisfi un sistema di equazioni. La matrice di incidenza tra nodi ed archi è una matrice rettangolare dove ciascuna riga rappresenta un nodo e ciascuna colonna rappresenta un arco. Nella matrice, con -1 si segnala che l'arco esce dal nodo, con +1 che l'arco entra nel nodo e con 0 la non esistenza di un arco per quel nodo.
- ✔ **Matrice di adiacenza.** Simile alla precedente, la matrice di adiacenza è una matrice quadrata dove righe e colonne corrispondono ai nodi, ed in essa è segnato con 0 il caso che i due nodi non siano adiacenti (e che quindi non esista un arco che li congiunge), con 1 altrimenti.
- ✔ **Insiemi di adiacenza.** È conveniente mantenere degli insiemi di adiacenza per ogni nodo del grafo. Si possono usare le liste di adiacenza o dei vettori di adiacenza. Si usa una struttura per tenere a mente gli archi, l'altra per i nodi.

Esplorazione di un grafo. Esistono due modi principali per esplorare un grafo orientato fortemente connesso (o non orientato ma connesso).

- ✔ **Depth-First-Search.** Detto anche visita in profondità, diretta conseguenza della visita in ordine anticipato di un albero. Ci si allontana da un nodo di partenza il può possibile lungo un cammino od una catena visitando nodi ed archi fino a giungere in un vicolo cieco. Allora si torna indietro lungo l'ultimo arco visitato e si riprende la visita allontanandosi per un nuovo cammino.
- ✔ **Breadth-First-Search.** Detto anche visita in ampiezza, i nodi vengono visitati in ordine di distanza crescente dal nodo di partenza dove la distanza è calcolata dal numero di archi tra un nodo e quello di partenza.

In entrambi i modi bisogna mantenere una lista dei nodi già visitati i cui adiacenti non lo sono ancora stati.

In Pratica...

20 Luglio 2010 :: *Dato un grafo G=(N,A) realizzato con matrice di adiacenza, definire il tipo di dato grafo, il tipo di dato nodo, e una procedura C ottima che stampa i nodi del grafo (con lo schema DFS).*

23 Marzo 2010 :: *Dato un grafo G=(N,A) realizzato con matrice di adiacenza, definire il tipo di dato grafo, il tipo di dato nodo, e una procedura C ottima che stampa i nodi del grafo (con lo schema BFS).*

```
#define N n     // numero di nodi del grafo G
typedef int matrice_adiacenza[n][n];
typedef matrice_adiacenza grafo;
typedef int nodo;

void adiacenti (grafo G, nodo U) {
   nodo V;
   for (v=0; v<n; v++) {
      if (G[u][v] == 1) printf (“(%d) → (%d)\n”, u, v);
   }
}
```

8 Aprile 2010 :: *Si definiscano le condizioni necessarie e sufficienti affinché un grafo sia anche un albero.*
Un albero è un sottografo aciclico che include tutti i nodi del grafo di partenza.

8 Aprile 2010 :: *Definire una procedura C VERIFICA (...) che dato un grafo G non orientato e connesso, verifica l'assenza di cicli.*
Un grafo orientato è **aciclico** (non contiene cicli) se durante una visita DFS non si incontrano archi all'indietro cioè, se durante la visita del nodo u non si incontra un arco (u,v) tale che v sia un nodo visitato e appartenga al cammino corrente che va dal nodo iniziale al nodo u.

```
struct structgrafo {
   int nodi [n+1];
   int archi [m];
   boolean visitato [n];
```

```
   boolean cammino [n];
   boolean aciclico;
}
typedef struct structgrafo*grafo;

void VERIFICA (grafo G, nodo u) {
   int i;
   nodo v;
   G->cammino[u] = TRUE;
   G->visitato[u] = TRUE;
   for (i = G->nodi[u]; i<G->nodi[u+1]; i++) {
      v = G->archi[i];
      if (!G->visitato[v]) VERIFICA (G, v);
      else
          if (G->cammino[v] == TRUE) G->aciclico = FALSE;
   }
   G->cammino[u] = FALSE;
}
```

Capitolo 9 ~ Strutture di dati e progetto di algoritmi

Progetto di algoritmi. Nel progetto di algoritmi possono essere evidenziate quattro fasi comuni attraverso le quali occorre passare:

- ✔ **Classificazione del problema.** Si cerca di stabilire se il problema da risolvere fa parte di una classe più ampia di problemi aventi caratteristiche comuni:
  - ✗ **Problemi decisionali.** La cui risposta è sì o no.
  - ✗ **Problemi di ricerca.** Tra tutte le possibili soluzioni si vole trovarne una, detta soluzione ammissibile, che soddisfi una certa condizione.
  - ✗ **Problemi di ottimizzazione.** Nei quali alle soluzioni ammissibili è associata una misura (o costo, o obiettivo) e si vuole trovare una soluzione ottima.
- ✔ **Caratterizzazione della soluzione.** Si prova a caratterizzare matematicamente la soluzione del problema. Tale caratterizzazione di solito suggerisce un algoritmo di risoluzione semplice ma non efficiente.
- ✔ **Tecnica di progetto.** Si cerca di applicare certe tecniche di progetto di algoritmi per rendere gli algoritmi più veloci. Le principali tecniche sono divide et impera (partizionare il problema in sottoproblemi più piccoli risolubili indipendentemente l'unione delle sui soluzioni da la soluzione del problema di partenza), backtrack, greedy, programmazione dinamica e ricerca locale.
- ✔ **Strutture di dati.** Si vede se c'è un modo astuto di strutturare i dati elaborati dall'algoritmo che permetta di eseguire velocemente le operazioni utilizzate nell'algoritmo.

Cammini minimi. Ora si considera un noto problema di ottimizzazione sui grafi orientati pesati, quello dei **cammini minimi**. Dato un grafo orientato con pesi (o lunghezze) interi relativi ad ogni arco per tutti gli archi del grafo, dato un nodo appartenente al grafo si vuole trovare un cammino da tale nodo ad un secondo nodo, per ogni altro nodo appartenente al grafo, tale che la somma delle lunghezze degli archi del cammino sia il più piccolo possibile.

Algoritmo di Dijkstra. Se la struttura che utilizziamo è una coda con priorità realizzata con una lista non ordinata, si ottiene un algoritmo conosciuto fin dal 1959 ed attribuito a Dijkstra. Gli elementi della coda con priorità diventano nodi del grafo e le loro priorità sono le distanze dal nodo di partenza. La complessità dell'algoritmo si riduce se le lunghezze sono tutte positive.

Algoritmo di Johnson. Se la struttura è una coda con priorità realizzata con uno heap, si ottiene un algoritmo proposto da D.B. Johnson. Poiché gli elementi della coda con priorità non coincidono con le priorità stesse, la struttura semplificata dello heap va modificata.
Questo algoritmo è simile a quello di Dijkstra, e si ottiene dalla procedura dei cammini minimi sostituendo le operazioni sugli insiemi con quelle delle code con priorità.

Algoritmo di Bellman-Ford-Moore. Se la struttura è una coda si ottiene un algoritmo che ha complessità polinomiale anche se ci sono archi con lunghezza negativa. Tale algoritmo si ottiene dal prototipo sostituendo le operazioni sugli insiemi con quelle sulle code. La struttura dell'algoritmo in pratica è quella di una visita BFS in cui la marcatura di un nodo consiste nel diminuirne la distanza, quando un nodo può essere visitato più di una volta.

In Pratica...

20 luglio 2010 :: *Dire quali sono le condizioni necessarie e sufficienti perchè l'albero di copertura G, (G è orientato e pesato) sia di costo minimo.*
Il teorema di Bellman permette di definire un algoritmo per la risoluzione del problema dei cammini minimi su un grafo orientato basato sulla verifica arco per arco delle condizioni di Bellman e sostituzione degli archi che violano tali condizioni. Una soluzione ammissibile è corretta se e solo se valgono le seguenti condizioni per ogni arco (i,j) appartenente ad A:

- ✔ $d_j = d_i - c_{ij}$ se (i, j) appartiene a T_{min}
- ✔ $d_j <= d_i - c_{ij}$ se (i, j) non appartiene a T_{min}

Algoritmo di Pape-D'Esopo. In pratica l'algoritmo che risulta più veloce nei problemi reali impiega una struttura detta coda a doppia fine, cioè una sequenza modificabile ad entrambi gli estremi che combina le proprietà di una coda con quelle di una pila. L'algoritmo permette di estrarre solo dalla testa della struttura ma ammette l'inserimento da entrambi i lati.

Capitolo 10 ~ Divide et Impera

Divide et Impera. La tecnica più importante e più usata per progettare algoritmi efficienti si basa su una massima antica dei tiranni, enunciata anche da Machiavelli a proposito della politica dell'antico senato romano. Applicata alla risoluzione di un problema computazionale, questa tecnica consiste nel partizionare il problema in sottoproblemi più piccoli dello stesso tipo, risolverli, e successivamente ricombinare con poco sforzo le soluzioni ottenute per ottenere la soluzione del problema originale.
Se i dati sono partizionati in maniera bilanciata allora l'algoritmo può risultare particolarmente efficiente.

Mergesort. Un noto algoritmo di ordinamento è il **mergesort**. Consiste nel dividere un vettore in due sequenze di n/2 elementi ciascuno, ordinare ciascuna sequenza e poi fondere le due metà ordinate in un unica sequenza ordinata.
Algoritmi basati sulla procedura mergesort risultano ottimali solo se impiegati su elementi che risiedono nella memoria secondaria.

Quicksort. L'algoritmo praticamente più efficiente per ordinare gli elementi di un vettore è il **quicksort**. Questo algoritmo è basato sulla tecnica divide et impera ma è diverso dal mergesort nel modo in cui divide e ricombina i risultati. La strategia consiste nel selezionare un elemento del vettore e adibirlo a perno attorno al quale riarrangiare gli altri elementi. Tutti gli elementi più piccoli del perno vengono spostati in posizioni del vettore che precedono il perno mentre gli elementi più grandi sono spostati in posizioni successive.

In Pratica...

20 Luglio 2010 :: *Realizzare una funzione BUBBLESORT per ordinare un vettore in modo decrescente.*
L'algoritmo delle bolle, o BUBBLESORT, consiste nello scorrere il vettore effettuando uno scambio quando due elementi contigui non sono nell'ordine corretto. In questo modo gli elementi più piccoli vengono “a galla” salendo verso le posizioni più alte del vettore.

```
void bubblesort (int v[]) {
   int i, j, tmp;
   for (i = 1; i < SIZE; i++) {
      for (j = SIZE-1; j>=1; j-- ) {
        if (v[j-1] < v[j]){      // Scambio v[j] con v[j-1]
         tmp = v[j-1];
         v[j-1] = v[j];
         v[j] = tmp;
```

```
        }
      }
   }
}
```

8 Aprile 2010 :: *Si scriva in C l'algoritmo del torneo per ordinare un vettore V contenente interi positivi e negativi.*
È possibile determinare la permutazione dei dati di un vettore corrispondente a quella in cui i dati sono ordinati in senso crescente (o alternativamente in senso decrescente) effettuando un **torneo** tra tutti gli elementi di un vettore.

```
#define size n;

void TORNEO ( int src[], int dest[]) {
   int classifica [size];
   int i, j;
   for (i=0; i<size; i++) classifica [i] = 0;
   for (i=0; i<size; i++) {
      for (j=i+1; j<size; j++) {
          if (src[i] > src[j]) classifica[i]++;
          else classifica[j]++;
      }
   }
   for (i=0; i<size; i++) dest[classifica[i]]=src[i];
}
```

Capitolo 11 ~ Backtrack

Backtrack. Una tecnica non molto astuta, che però alle volte può risultare utile, si basa sulla considerazione che se non si riesce a risolvere un problema conviene tornare indietro e ricominciare da capo con un altra tecnica. Questa idea, detta **backtrack**, è alla base di molti algoritmi di visita di alberi e grafi ed è usata per generare sistematicamente tutte le soluzioni possibili di un problema.

Inviluppo convesso. Un poligono nel piano bidimensionale è convesso se ogni segmento di retta che congiunge due punti del poligono sta interamente nel poligono stesso incluso il bordo. L'inviluppo convesso di un poligono è dato dal più piccolo poligono che comprende tutti i vertici del poligono.

- ✔ **Algoritmo di Graham.** L'algoritmo appena illustrato non è molto efficiente perchè esamina gli n punti in modo troppo caotico. Vediamo come un esame sistematico dei punti, basato su un ordinamento ed un backtrack, permetta di progettare un algoritmo molto più efficiente. Innanzi tutto si osservi che in un insieme di punti, quello con ordinata minima è un vertice dell'inviluppo convesso. Inoltre tracciando una retta orizzontale in tale punto e facendola ruotare attorno al punto in senso antiorario si incontrano in sequenza tutti i punti rimanenti.