Parte Seconda: Tipi di Dato

Premessa

La presente dispensa, purtroppo è largamente incompleta: la parte relativa agli alberi in particolare, contiene quasi esclusivamente il codice dei programmi visti a lezione.

1 Tipi di dato: astrazione e implementazione

Un famoso slogan coniato dal Turing Award[1] Nicklaus Wirth[2] recita: "Algoritmi + Strutture Dati = Programmi"[3]. Lo slogan evidenzia il principio secondo cui la corretta progettazione delle strutture dati manipolate da un programma è essenziale per rendere i programmi facili da scrivere e soprattutto da leggere, e quindi modificare. In questa sezione ci occuperemo di vedere cosa si intende per nozioni quali *struttura dati*, o *tipo (astratto) di dato* e vedremo come la corretta progettazione delle strutture dati, renda semplici problemi a prima vista non banali e la progettazione di strutture dati più complesse.

1.1 Esempio: Labirinto

Supponiamo di voler scrivere una procedura di visita di un labirinto, che trovi sempre, se esiste, una strada per uscire. Ora, indipendentemente da quale sia la *rappresentazione* del labirinto come struttura dati, quali sono le *azioni* che dobbiamo saper compiere per uscire dal labirinto? Innanzitutto ci serviranno le operazioni di percorrere un corridoio e riconoscere se una stanza sia un'uscita o meno. Ci servirà inoltre, un meccanismo per poter marcare le strade percorse e quindi uno per riconoscere quelle già percorse (il nostro filo di Arianna); sarà necessario inoltre saper tornare indietro quando è stabilito che da una certa stanza non c'è via d'uscita. Avendo a disposizione questo tipo di operazioni, e supponendo che il progettista della struttura dati "labirinto" le abbia messe a nostra disposizione, possiamo scrivere un algoritmo per uscire dal labirinto come illustrato in Fig.**??**. Osserviamo che non è necessario conoscere l'*implementazione* dei tipi `labirinto, stanza` o `cammino`, ma solo la *specifica* delle operazioni definite su oggetti di questi tipi. Non solo, ma **non** conoscendo la struttura dati usata per implementare questi tipi, siamo *costretti* ad usare le operazioni rese disponibili da chi ha definito questi tipi di dato: questa perdita di libertà, tipicamente si rivela un vantaggio, in quanto assicura che l'utilizzatore acceda in modo coerente alla struttura dati che, sotto la responsabilità del suo progettista, rispetterà un *invariante di tipo di dato*: accessi incoerenti, infatti, potrebbero generare strutture dati con rappresentazioni non previste dal progettista della struttura dati e quindi minare la correttezza delle operazioni definite sulla struttura dati. Discuteremo più a fondo questa questione presentando il tipo di dato *insieme*.

Un secondo importante vantaggio di questo approccio è legato al cosidetto *riuso del software*: se il progettista della struttura dati labirinto volesse cambiare la *rappresentazione* del tipo di dato labirinto, ad esempio per implementare in modo efficiente una nuova operazione

[1] Una sorta di premio Nobel per l'Informatica.

[2] Nicklaus Wirth, è stato uno dei grandi studiosi dei linguaggi di programmazione. E' noto soprattutto in quanto ha definito e implementato, tra gli altri, il linguaggio Pascal ed il Modula 2.

[3] è anche il titolo di un celeberrimo libro che appare nella lista dei testi consigliati nella bibliografia.

non prevista all'inizio, avendo cura di salvaguardare la specifica delle operazioni già definite, non renderebbe necessario riscrivere i programmi che usano la struttura dati, per i quali la modifica della rappresentazione sottostante sarebbe del tutto *trasparente*.

Tipicamente ragioni di efficienza, potrebbero sporadicamente suggerire di violare questa gerarchia di *astrazioni*.

```
int esciLabirinto(labirinto L, cammino *p)
/* PREC: ??
/* POST: ritorna la sequenza di stanze attraversata se
/* il labirinto ha soluzione, la sequenza vuota altrimenti
{ stanza s = entrata(L);
  corridoio c;

  *p = camminoVuoto();

  while(1)
  { if (uscita(s)) return 1;        /* FINE: sono arrivato all'uscita */
    if (c=prossimoCorridoio(s))     /* se c'e' un corridoio uscente NON percorso */
       { marca(c);                  /* marca il passaggio in C */
         s = percorriAvanti(c);     /* vado nella prossima stanza */
         *p = aggiungi(c, *p);      /* aggiorna cammino percorso */
        }
   else /* non ci sono nuove strade da esplorare da s*/
     { if (nonVuoto(p))             /* se non e' una stanza iniziale ...*/
          { c = ultimoPercorso(*p); /* seleziona l'ultimo corridoio percorso */
            s = percorriIndietro(c);/* torna indietra */
            togli(c, *p);           /* aggiorna cammino percorso */
            }
       else return 0;               /* FINE: non c'e' soluzione */
      }
   }
}
```

Figure 1: Programma per uscire da un labirinto

2 Liste

In questa sezione rivedremo il tipo di dato *lista concatenata*. Iniziamo col dare una definizione formale, *induttiva* di cosa sia una lista:

Definizione 2.1 Una lista di elementi di tipo T è:

1. la lista vuota (cioè la lista che non contiene nessun elemento);
2. oppure un elemento di tipo T, seguito da una lista.

Osserviamo subito che la definizione di cosa sia una lista viene data in termini di ciò che è una lista. Il lettore non dovrebbe essere particolarmente sorpreso da questo modo di dare definizioni, pervasivo in matematica, e già incontrato nella definizione di funzioni attraverso equazioni ricorsive. Volendo definire nel linguaggio `C` un tipo di dato *lista*, occorre aggirare l'intrinseca ricorsività della definizione induttiva di lista, usando l'escamotage dei *puntatori*. Un puntatore, altro non è che l'indirizzo di una cella di memoria. In `C` non è possibile che la definizione di tipo contenga un campo dello stesso tipo che si sta definendo (in effetti questo definirebbe una struttura dati che occupa uno spazio infinito): tuttavia è possibile che contenga un campo puntatore ad oggetti dello stesso tipo. L'idea è quella di vedere una lista come una sequenza di lunghezza illimitata a priori di *nodi*[4]. Ciascun nodo della lista contiene un campo per memorizzare l'informazione di tipo T ed un campo per mantenere un puntatore al resto della lista (più concretamente al nodo successivo). Il tipo `lista` sarà poi definito come un tipo puntatore ad un nodo. In questo modo la lista vuota sarà rappresentata dalla costante pointer `NULL`.

```
typedef struct L
   { int elem;
     struct L * next;
    } listanode;

typedef listanode* lista.
```

Ovviamente la rappresentazione scelta, anche se è la più comune in letteratura, non è la sola possibile. Negli esercizi, verrà chiesto di trovare rappresentazioni alternative per implementare in modo efficiente alcune operazioni.

Osserviamo che già in questo esempio, appaiono evidenti due aspetti:

1. un aspetto *logico*, che a che fare con cosa sia una lista, quali siano gli oggetti di tipo lista (la lista vuota, distinta dalle liste non vuote, formate da un elemento (*testa*) seguito dal resto della lista (*coda*));

2. un aspetto *implementativo*, legato ad una *rappresentazione* nel linguaggio di programmazione scelto[5] (nel nostro caso il `C`).

Potremmo per la verità scrivere in `C` una serie di funzioni sulle liste che implementano le operazioni elementari (come aggiungere o togliere un elemento), e quindi dimenticare operazioni troppo a basso livello (come allocare o disallocare memoria o spostare un puntatore) che non appartengono alla specifica logica del tipo di dato, ma alla sua rappresentazione: nella scrittura di funzioni più complesse potremmo riferirci alle funzioni definite come *azioni elementari*, scrivendo programmi più legati all'aspetto logico che a quello implementativo e quindi più vicino alle specifiche e al nostro modo di concettualizzare i problemi e di conseguenza più leggibili.

Il problema rimane che il linguaggio `C` a differenza dei suoi discendenti ad oggetti (ad esempio il `C++`), non offre meccanismi di *mascheramento*, che rendano inacessibile all'utilizzatore di un tipo di dato la sua rappresentazione sottostante. Tuttavia ci imporremo una autodisciplina, imponendoci di non violare la gerarchia di astrazioni costruita.

[4] l'unica limitazione sarà data dalla disponibilità di memoria da allocare ad eventuali nuovi nodi.

[5] osserviamo a questo proposito che esistono linguaggi di programmazione in cui i tipi di dato vengono specificati semplicemente attraverso la loro struttura logica, lasciando al compilatore il fardello di costruire una adeguata rappresentazione interna inaccessibile

Notazione 2.2 Per parlare di proprietà logiche dei programmi, abbaimo necessità che il nostro *linguaggio logico* contenga degli operatori sulle liste. Ci basteranno i seguenti operatori:

1. una costante per la lista vuota: nil;
2. un operatore per attaccare un elemento in testa ad una lista: .;
3. un operatore per concatenare due liste $\cdot$.

Useremo inoltre, nello scrivere le asserzioni logiche, la convenzione che una variabile di tipo `lista L` (che concrettamente è semplicemente un puntatore) rappresenti la lista formata dalla sequenza di elementi incontrati seguendo i puntatori `next`. Qualora fosse conveniente indicare una lista, elencando tutti i suoi elementi $x_1, \ldots, x_n$, sarà usata la notazione $\langle x_1, \ldots, x_n \rangle$.

Esempio 2.3 Le seguenti saranno notazioni che indicano la stessa lista:

1. $\langle 1, 5, 7 \rangle$
2. $1.\langle 5, 7 \rangle$
3. $\langle 1 \rangle \cdot \langle 5, 7 \rangle$

□

Cominciamo col vedere brevemente la definizione `C` di alcune operazioni elementari: inserimento in testa, test di lista vuota, e selezione della testa e della coda di una lista e verifica della presenza di un elemento.

```
lista addHead(lista L, int el)
/* POST: ritorna el.L */
{ lista lAux = (lista) malloc(sizeof (listanode));
     /* alloco memoria per un nuovo nodo */
  lAux->elem=el;          /* lAux = <el> */
  lAux->next= L;          /* lAux = el.L */
  return lAux;
  }
```

```
int isEmpty(lista L)
/* ritorna 1 se L e' vuota. 0 altrimenti */
{ if (L==NULL) return 1
     else return 0;
}
```

```
int isIn(lista L, int elem)
/* ritorna 1 se elem e' in L
{ if (!L) return 0
     else return isIn(L->next, elem);
  }
```

```
int head(lista L)
/* PREC: L non vuota */
{ return L->elem;}
```

```
int tail(lista L);
/* PREC: L != nil */
{return L->next;}
```

A titolo di ripasso del linguaggio `C` è interessante vedere due possibili codici alternativi per la semplice funzione `isEmpty`:

```
int isEmpty(lista L)        int isEmpty(lista L)
{ if (L) return 1;          { return !L;}
  return 0;
}
```

Osserviamo che già queste semplici funzioni ci permetterebbero di scrivere tutte le funzioni calcolabili sulle liste senza scrivere nessun puntatore e nessun operatore su di essi! A titolo di esempio vediamo come scrivere una funzione che aggiunge un elemento in coda ad una lista. Questa funzione si può facilmente scrivere osservando che aggiungere un elemento in testa o in coda alla lista vuota produce lo stesso effetto. Quindi ci aspettiamo che siano verificate le seguenti equazioni:

$$\begin{array}{rcll} \mathsf{addTail}(\mathsf{nil}, e) & = & \langle e \rangle & = \mathsf{addHead}(\mathsf{nil}, e) \\ \mathsf{addTail}(h.T, e) & = & h.\mathsf{addTail}(e, T) & = \mathsf{addHead}(\mathsf{addTail}(e, T), h) \end{array}$$

da cui si deriva facilmente il seguente codice C:

```
lista addTail(lista L, int el)
/* POST: restituiste  L@<el>*/
{ if (isEmpty(L)) return addHead(L, el);
     else return addHead(addTail(tail(L), elem), head(L));
}
```

Probabilmente il lettore non è familiare con questo stile di programmazione, fatto solo di composizione di chiamate di funzione, eventualmente annidate, tipico dei linguaggi detti appunto *funzionali*. Come abbiamo già detto la chiamata di funzione è un'operazione costosa. In un linguaggio imperativo, disponendo dell'assegnazione, un codice più ragionevole potrebbe essere il seguente:

```
lista addTail(lista L, int el)
/* Aggiunge el in coda alla lista L */
{ if (isEmpty(L)) return addHead(L, el);
     else
         { L->next= addTail(L->next, elem);
           return L; }
}
```

Allo scopo di far familiarizzare il lettore con diversi stili di programmazione, alterneremo in maniera arbitraria lo stile scelto per scrivere i programmi. Veniamo ora ad un altro tipico problema sulle liste: date due liste, appendere la seconda lista alla prima. Ancora una volta, possiamo specificare mediante equazioni ricorsive le relazioni logiche verificate dalla funzione append:

$$\begin{array}{rcl} \mathsf{append}(\mathsf{nil}, M) & = & M \\ \mathsf{append}(h.T, M) & = & h.\mathsf{append}(T, M) \end{array}$$

e scrivere il corrispondente codice C:

```
lista append(lista L, lista M)
/* restituisce L@M
 */
```

```
{ if (isEmpty(L)) return M;
      else return addHead(head(L), append(tail(L), M));
}
```

Per quanto riguarda la correttezza di questa funzioni, è banale verificare che esse calcolano quanto prescritto dalle specifiche mediante equazioni ricorsive. La terminazione invece dipende solo dal fatto che le chiamate ricorsive vengono attivate con parametri che contengono liste "più corte": una ovvia funzione di terminazione per `addTail` o `append` sarà la lunghezza della lista passata come primo parametro.

Possiamo scrivere una funzione `appendIt`, facendo ragionamenti più concreti, pensando a quali operazioni occorre fare per appendere una lista in coda ad un'altra: bisogna scorrere tutta la prima lista per trovare la sua fine, e, arrivati in fondo, modificare il puntatore che segnala la fine della lista, facendolo puntare all'inizio della seconda. Vediamo l'applicazione di questa idea al programma iterativo che calcola la funzione `append`:

```
lista appendIt(lista L, lista M)
/* ritorna L@M
 */
{ lista lAux = L;

  if (lAux)
     { while (lAux->next)
       /* INV: esiste L'. L=L'@lAux & lAux != nil */
         { lAux=lAux->next;}
       lAux->next = M;
       return L;}
    else
       return M;
  }
```

L'invariante del ciclo semplicemente dice che la variabile `lAux` rappresenta una lista che è un *suffisso* del primo parametro e che durante il ciclo tale lista è non vuota (condizione garantita dalla guardia). Questo invariante e la negazione della guardia garantiscono che all'uscita dal ciclo, `lAux` punterà all'ultimo nodo della lista originaria: sarà sufficiente modificare il puntatore al successivo, in modo che punti all'inizio della lista `M` per ottenere l'effetto desiderato.

2.1 La funzione Reverse

Proponiamoci ora di scrivere una funzione che rovescia gli elementi di una lista. E' relativamente scrivere semplice la specifica logica, attraverso equazioni ricorsive. Chiaramente la lista vuota rovesciata è ancora la lista vuota, mentre la testa della lista va appeso in coda al rovesciamento della coda della lista (il primo elemento diventa l'ultimo!):

$$\begin{aligned} \mathsf{reverse}(\mathsf{nil}) &= \mathsf{nil} \\ \mathsf{reverse}(h.t) &= \mathsf{addTail}(\mathsf{reverse}(t), h) \end{aligned}$$

Ancora una volta, possiamo facilmente trasformare queste equazioni ricorsive in un programma `C` ricorsivo:

```
lista reverse(lista L)
/* ritorna la lista L rovesciata
 */
{ if (L==NULL) return NULL;
  else return (addTail (reverse(L->next), L->elem));
}
```

Osserviamo due cose:

1. il programma allocherà nuova memoria per creare una nuova lista (ciò è dovuto al fatto che questo viene fatto dalla funzione `addTail`), mantenendo immutata la lista di ingresso;

2. chiaramente la funzione scorrerà la lista di ingresso (quindi eseguendo un numero di chiamate ricorsive pari al numero di elementi della lista di ingresso).

Tuttavia osserviamo che la chiamata ad `addTail` non ha complessità costante, ma proporzionale alla lunghezza della lista a cui attaccare in coda l'elemento; quindi la complessità della funzione, essendo n la lunghezza della lista, sarà $1 + 2 + ... + (n-1) = \frac{n \times (n-1)}{2}$, cioè $\mathcal{O}(n^2)$. E' questa la complessità intrinseca del problema o è possibile fare meglio? Se riuscissimo a costruire la nuova lista facendo n inserimenti in testa, invece che in coda, otterremo una funzione di complessità *lineare*, cioè proporzionale ad n. Per non dimenticare l'iterazione, vediamo l'algoritmo iterativo lineare:

```
lista reverseIt(lista L)
/* POST: ritorna M = reverse(L)
{ lista M = NULL;
  lista N = L;    /* variabile ausiliaria per scorrere L */

  while (N)
  /* INV: L=N@reverse(M) */
  { M = addHead(M, N->elem);
    N = N->next;
   }
  return M;
}
```

Osservate l'invariante e osservate che all'uscita del ciclo (che avviene quando il pointer `N` assume il valore `NULL`, cioè quando $N = \mathsf{nil}$), ho che l'asserzione finale è verificata (infatti ho che: $\mathsf{reverse}(L) = M$ se e solo se $L = \mathsf{reverse}(M)$. Ricordando l'esempio di fibonacci, non è difficile pensare ad un programma ricorsivo che emula il comportamento del programma ricorsivo: consideriamo una funzione ausiliaria con due paramentri, in cui il secondo parametro emula la variabile `M` della funzione iterativa:

```
lista reverseEffAux(lista L, lista M)
{ if (isEmpty(L)) return M;
     else return reverseEffAux(tail(L), addHead(M, head(L));
}
```

```
lista reverseEff(lista L)
/* ritorna la lista L rovesciata
 */
{ return reverseAux(L, NULL);
}
```

La funzione `reverseEffAux` usa un trucco che si può rivelare spesso utile: l'osservazione cruciale è che scorrendo una lista ricorsivamente, la si percorre di fatto due volte: una all'"andata", allocando le chiamate ricorsive e una al "ritorno" (in ordine rovesciato!), al rientro delle chiamate ricorsive. Per convincersi "sperimentalmente" di questo fatto, il lettore è invitato a verificare l'output del seguente programma:

```
void printList(lista L)
{ if (L==NULL) {printf("\n");}
     else
        { printf("%3d", L->elem);
          printList(L->next);
          printf("%3d", L->elem);
     }
  }
```

E' necessario il parametro ausiliario nella funzione `reverseEffAux`, perché ovviamente disponiamo dei risultati calcolati ricorsivamente, solo durante il rientro delle chiamate ricorsive. Gli inserimenti in testa alla lista ausiliaria `M` avvengono infatti all'andata nello scorrimento della lista. Concludiamo il discorso su questa funzione, ponendoci il problema di effettuare il rovesciamento della lista *senza allocare nuova memoria*, detta anche *in place*. E' necessario rovesciare i puntatori. Chiaramente in questo caso distruggeremo la lista originaria. Sfruttando sempre l'idea dell'"andata" e "ritorno", la versione ricorsiva, non è particolarmente più complicata:

```
lista reversePlaceAux(lista L)
/* PREC: L != nil
 */
 { lista M;

   if (L->next==NULL) return L;
      else
        { M = reversePlaceAux(L->next);
          (L->next)->next = L;
          return M;
          }
}

lista reversePlace(lista L)
/* rovescia la lista L rovesciando i puntatori
 */
 { lista M;
```

```
   if (L==NULL) return L;
   else
     { M=reversePlaceAux(L);
       L->next = NULL;
       return M;
     }
}
```

La procedura ausiliaria si limita all'"andata" delle chiamate ricorsive a trovare il nuovo primo elemento della lista (che è l'ultimo della lista in ingresso) e al "ritorno" sposta opportunamente i puntatori. Osserviamo che il caso base della ricorsione, per una volta, non è la lista vuota, ma la lista che contiene un solo elemento (anche sulla lista che contiene un solo elemento, il rovesciamento non produce effetti). La procedura `reversePlace` invece tratta correttamente il caso di lista vuota, invoca la funzione ausiliaria che rovescia i puntatori di tutti gli elementi, tranne quello del primo nodo, che essendo diventato l'ultimo, dovrà diventare il pointer `NULL`, il consueto segnale di fine lista.

2.2 Rimozione di Elementi

Vediamo in questa sottosezione, alcuni programmi che rimuovono elementi da una lista. Cominciamo con il programma che rimuove tutte le occorrenze di un elemento da una lista: sarà sufficiente scorrere la lista e quando il nodo contiene l'elemento fornito in input eliminarlo, liberando la memoria e avendo cura di risistemare opportunamente i puntatori. Vediamo ancora una volta sia la versione ricorsiva che iterativa (in fig. **??**).

Osservate che prima di invocare la funzione `free` è necessario salvare *il contenuto* della memoria deallocata, nel caso in cui questo sia utile (tipicamente questo è sempre necessario per sistemare opportunamente i puntatori dopo la cancellazione). La deallocazione in genere non implica la *cancellazione* del contenuto delle celle liberate, ma *non* è prudente speculare sul fatto che queste celle non siano già state riallocate da altre procedure e quindi già sovrascritte con altri dati. (ricordate che nei calcolatori moderni, tipicamente molti programmi sono contemporaneamente in esecuzione).

```
lista removeElementRec(lista L, int el)
/* rimuove tutte le occorrenze di el nella lista L
 */
{ lista M;

  if (L==NULL) return NULL;
  else if (L->elem == el)
            { M = L->next;
              free(L);
              return(removeElement(M, el));
              }
  else
    { L->next = removeElementRec(L->next, el);
      return L;}
 }
```

```
lista removeElementIt(lista L, int el)
/* POST: restituisco L', el not in L', forall e<>el in L, e in L'
 */
{ lista inizioL = L;
  lista currL;
  lista precL;
  lista tmp;

 /* si rimuovono  tutte le eventuali occorrenze iniziali di el
  * ottenendo il puntatore al primo nodo che contiene un valore
  * diverso da el. Questo sara' il pointer da tornare al chiamante
  */
  while (inizioL && inizioL->elem == el)
    { tmp = inizioL->next;
      free(inizioL);
      inizioL = tmp;
     }
  precL = inizioL;

  /* scorro la lista con due puntatori ad elementi
   * consecutivi per permettere le cancellazioni
   * OSS: alla prima iterazione precPTR->elem != el
   */
  while (precL)
  { currL = precL->next;
    if (currL)
        if (currL->elem ==el)
          { precL->next = currL->next;
            free(currL);
            }
    precL = currL;
    }
  return inizioL;
}
```

Figure 2: Rimozione Iterativa di un Elemento

Vediamo ora come usando la funzione che rimuove tutte le occorrenze di un elemento sia possibile scrivere facilmente una funzione che prese due liste, `L` ed `M`, restituisce tutti gli elementi che appartengono ad `L`, ma non ad `M`. L'idea è semplicemente quella di scorrere `M` ed eliminare da `L` gli elementi trovati percorrendo `M`.

```
lista diffListaPlace(lista L,lista M)
/* modifica la lista L togliendo tutti gli elementi che compaiono in M
 */
{ if (M == NULL) return L;
     else return diffLista(removeElementRec(L, M->elem), M->next);
}
```

Qualora si sia interessati a mantenere immutate le liste di ingresso, occorre seguire un'altra strategia, e cioé ricopiare in una nuova lista gli elementi che appartengono a `L` ma non a `M`, aggiungendoli in testa, previo il controllo che non stiano in `M`.

```
lista diffLista(lista L,lista M)
/* crea una lista con tutti gli elementi di L che non stanno in M
 */
{ if (L == NULL) return NULL;
  else if (!isIn(M, L->elem))
            return addHead(diffLista(L->next, M), L->elem);
  else return diffLista(L->next, M);
}
```

Concludiamo questa sezione, ponendoci un ultimo problema, simile ai precedenti: data una lista, restituirne una che contiene gli stessi elementi, ma in cui sono stati eliminati eventuali occorrenze ripetute di ciascun elemento. L'algoritmo è semplice: è sufficiente percorrere la lista e, per ogni elemento, cancellare eventuali altre occorrenze di quell'elemento nel resto della lista.

```
lista elimCopie(lista L)
/* ritorna la lista cancellando eventuali ripetizioni di ciascun elemento
 */
{ if (!L) return NULL;
     else
       { L->next = elimCopie(removeElement(L->next, L->elem));
         return L;}
}
```

2.3 Confronto tra Liste e Array

Siamo pronti per affrontare brevemente il confronto tra la struttura dati lista concatenata e la struttura dati array. La lista ha una natura essenzialmente *sequenziale* e quindi per accedere ad un elemento è necessario scorrere sequenzialmente la lista; gli elementi dell'array sono invece accessibili direttamente specificando un indice (provare a scrivere la ricerca binaria in una lista ordinata!). Per contro, la lista risulta più flessibile nella soluzione di molti problemi, perchè non è necessario specificarne a priori la lunghezza e finchè c'è memoria libera è possibile allocare nuovi elementi. La lista presenta un certo overhead di occupazione

di memoria dovuto alla necessità di memorizzare oltre ai campi informazione anche i campi puntatore. Un ultimo vantaggio della lista dipende dal fatto che alcune operazioni (come rimuovere o inserire un elemento) necessitano solo lo spostamento di alcuni puntatori, mentre in un array le stesse operazioni necessitano di costosi spostamenti di tutti gli elementi che stanno a destra del punto in cui ho eliminato o inserito l'elemento. Occorrerà sempre tenere ben presenti questi elementi al fine di utilizzare la giusta struttura dati, tenendo conto della natura del problema e degli obiettivi prioritari (efficienza, occupazione di memoria, flessibilità ...).

2.4 Liste Ordinate

Concludiamo l'analisi delle liste, vedendo alcune funzioni che trattano liste ordinate: alcune operazioni trarranno vantaggio dall'ordine, al prezzo di dover mantenere ordinata la lista. Ad esempio, dovendo rimuovere un elemento, non sarà necessario scorrere tutta lista, ma sarà sufficiente fermarsi non appena si incontra un elemento maggiore di quello ricevuto in input. Cominciamo a vedere una funzione per costruire liste ordinate, che inserisce un elemento al posto giusto in una lista ordinata per mantenerla tale:

```
lista addOrd(lista L, int el)
/* PREC: ord(L)
 * POST: ritorna L', ord(L'),  esiste M,N, L=M@N, L'=M@(el.N)
 */
{ if (isEmpty(L) || L->elem >= el) return addHead(L, el);
    else
      { L->next = addOrd(L->next, el);
        return L;}
}
```

Rispetto all'analogo programma su liste non ordinate (`addTail`) cambia solamente la condizione sotto la quale andiamo ad inserire l'elemento. Osserviamo che la condizione dell'`if` non genera errori di *null pointer exception*, tipici di quando si va ad applicare l'operatore `->` ad un pointer `NULL`: questo perché non appena la condizione `isEmpty(L)` è valutata vera, l'or viene valutato vero, senza verificare la seconda condizione. Analogamente, sotto l'ipotesi che la lista di ingresso sia ordinata, anche nelle funzioni che rimuovono elementi, potremmo fermarci non appena incontriamo memorizzati nella lista elementi maggiori dell'elemento da cancellare. Vediamo ad esempio le funzioni `removeElementOrd`:

```
lista removeElementOrd(lista L, int el)
/* PREC: ord(L)
 * POST: ritorna M@N, L=M@(el....el.N)
 */
{ lista M;

  if (L == NULL) return NULL;
  else if (L->elem == el)
            { M = L->next;
              free(L);
              return(removeElementOrd(M, el));
```

```
            }
   else if (L->elem > el) return L;
   else
        { L->next = removeElementOrd(L->next, el);
          return L;}
 }
```

Sostituendo questa funzione nella definizione di `elimCopie` alla funzione `removeElement`, otteniamo una funzione `elimCopieOrd`, che cancella le occorrenze ripetute di un elemento, sotto le ipotesi che la lista di intresso sia ordinata.

Più interessanti sono le modifiche da farsi alla funzione `diffLista`, per ottenere il massimo profitto dall'assunzione che i parametri di ingresso siano liste ordinate. L'idea è di scorrere le due liste, `L` e `M` parallelamente: se la testa di `L` è maggiore di quella di `M` si fa avanzare `M`, e viceversa. Se i due elementi in testa sono uguali, allora significa che posso eliminare questo elemento dal risultato.

```
lista diffListaOrdPlace(lista L,lista M)
/* PREC: ord(L) & ord(M)
 * return L', forall el in L', el occorre in L, ma non in M
 */
{ if (!L) return NULL;
  else if (!M) return L;
  else if (L->elem == M->elem)
            return diffListaOrd(L->next, M->next);
  else if (L->elem < M->elem)
            return addHead(diffListaOrd(L->next, M), L->elem);
  else return diffListaOrd(L, M->next);
 }
```

In questo caso, chiamando $l = \mathsf{lenght}(L)$ ed $m = \mathsf{lenght}(M)$, la funzione di terminazione è $l + m$. Ogni attivazione ricorsiva della funzione, sarà invocata sulla coda di L o di M (o entrambe) e quindi la funzione di terminazione decresce almeno di 1 (ed eventualmente 2). Di consenguenza la complessità dell'algoritmo è lineare nella lunghezza delle due liste, cioè $l + m$. La funzione `diffLista`, invece, che non fa ipotesi sull'ordinamento delle liste di ingresso, necessita di scorrere la seconda lista *per ogni* elemento della prima e quindi ha complessità $l \times m$.

L'applicazione della tecnica di scorrere parallelamente due liste, funziona egregiamente nella soluzione di un tipico problema su liste (e vettori) ordinati: la fusione ordinata di liste ordinate (*merge*). Anche in questo caso, la complessità dell'algoritmo sarà dell'ordine di $l + m$.

```
lista mergePlace(lista L, lista M)
/* PREC: ord(L) & ord(M)
 * POST: restituisce L', ord(L'), forall elem in L o in M, el in L'
 */
{ if (!L) return M;
  else if (!M) return L;
  else if (L->elem < M->elem)
```

```
          { L->next = mergePlace(L->next, M);
            return L;
            }
 else
     { M->next = mergePlace(L, M->next);
       return M;}
       }
}
```

Esercizi

1. Scrivere una procedura iterativa che rovescia una lista in place.

2. Modificare le funzioni `removeElementIt` e `removeElementRec`, scrivendo due funzioni `removeFirstIt` e `removeFirstRec` che in modo che eliminano solo la *prima* occorrenza dell'elemento.

3. Modificare le funzioni `removeElementIt` e `removeElementRec`, scrivendo due funzioni `removeLastIt` e `removeLastRec` che eliminano solo la *ultima* occorrenza dell'elemento.

4. Considerate le seguenti equazioni ricorsive per specificare il comportamento della funzione append:
$$\begin{aligned} \mathsf{append}(M, \mathsf{nil}) &= M \\ \mathsf{append}(M, h.T) &= \mathsf{append}(\mathsf{addTail}(M, h), T) \end{aligned}$$
Scrivere il programma `C` associato. Perchè il programma scritto nella sezione **??** ha da ritenersi preferibile?

5. ♣Dimostrare per induzione che le equazioni ricorsive dell'esercizio precedente specificano la stessa funzione append definita precedentemente nella sezione **??**.

6. Modificare la funzione `mergePlace` in modo che la funzione generi una nuova lista. [**Sugg**: usare la funzione `addHead` e una funzione ausiliaria `copia` che presa una lista genera una copia di essa]

7. Modificare la funzione `mergePlace` *ad hoc* per calcolare l'unione tra insiemi, che eviti di creare ripetizioni nella lista risultato, rendendo inutile quindi la chiamata di `elimCopie`.

8. Scrivere una funzione `C int sottoSequenza(lista L, lista M)` che restituisca 1 se la lista `L` è sottolista di M. Formalmente, se esistono due liste N, N' eventualmente vuote tali che:
$$M = N \cdot L \cdot N'$$

9. Scrivere una funzione `C int sottoLista(lista L, lista M)` che restituisca 1 se gli elementi della lista `L` compaiono nello stesso ordine in `M`, non necessariamente consecutivamente.

10. Implementare il tipo di dato lista concatenata in modo che l'operazione di *concatenazione* di liste abbia complessità costante, non necessiti cioè dello scorrimento di tutta la prima lista.

11. Implementare il tipo di dato *numero naturale*, in modo che il numero naturale n sia rappresentato da una lista con n nodi (i nodi contengono solo il campo puntatore). Scrivere prima le funzione base (successore, predecessore, test di zero), poi le usuali funzioni aritmetiche (somma, sottrazione, moltiplicazione, divisione e resto intero) ed infine esponenziale e fattoriale.

3 Il tipo di Dato Insieme

In questa sezione, vediamo come definire un tipo di dato *insieme*. Dobbiamo essenzialmente porci due questioni:

1. quali operazioni dobbiamo considerare e implementare;
2. quale struttura dati può convenientemente memorizzare insiemi in modo che le operazioni siano implementate in modo efficiente.

In quanto alla prima domanda, dovremmo scrivere le operazioni: test di insieme vuoto, unione, intersezione, differenza, inserimento di un elemento, test di appartenenza. Per quanto riguarda la seconda, osserviamo subito che le operazioni sugli insiemi assomigliano molto alle operazioni sulle liste, tuttavia saltano subito agli occhi alcune differenze:

1. le liste $\langle 1, 2, 3\rangle$ e $\langle 2, 3, 1\rangle$ sono due liste diverse, mentre, nella usuale rappresentazione degli insiemi, $\{1, 2, 3\}$ e $\{2, 3, 1\}$ denotano lo stesso insieme;
2. analogamente le liste $\langle 1, 2, 3\rangle$ e $\langle 1, 1, 2, 3\rangle$ sono due liste diverse, mentre gli insiemi $\{1, 2, 3\}$ e $\{1, 1, 2, 3\}$ denotano lo stesso insieme;

In sostanza, volendo rappresentare gli insiemi come liste, dobbiamo confrontarci col problema che molte liste diverse rappresentano lo stesso insieme. Questo fenomeno risulta essere molto fastidioso, sia nelle funzioni di input/output, sia nell'implementazione di funzioni come l'uguaglianza di due insiemi.

Da un punto di vista matematico il tipo di dato insieme è costituito dalle *classi di equivalenza* delle liste rispetto all'ordine e alle ripetizioni. Come forse il lettore saprà, in questi casi è conveniente, per ciascuna classe di equivalenza trovare un elemento *canonico* che rappresenti l'intera classe.

Una scelta naturale è quella di rappresentare un insieme con una *lista ordinata senza ripetizioni*. Questa scelta, è avallata anche dalla maggiore efficienza che le operazioni sulle liste ordinate hanno rispetto alle stesse opearazioni su liste qualsiasi. Il fatto che la rappresentazione di un insieme sia una lista ordinata senza ripetizioni, costituisce l'*invariante del tipo di dato* insieme: quindi le varie operazioni non si dovranno limitare ad assumere questa proprietà, ma qualora sia necessario fornire in output un insieme, questo dovrà essere ancora rappresentato con un insieme ordinato senza ripetizioni.

Ad esempio la funzione che unisce due insiemi potrà essere implementata usando la funzione `merge` tra liste ordinate (che produce come risultato una lista ordinata. Sarebbe sconveniente usare la più semplice funzione `append`). Tuttavia il risultato potrebbe contenere più copie dello stesso elemento: possiamo ripristinare l'invariante del tipo di dato semplicemente applicando la funzione `elimCopieOrd` prima di restituire il risultato.

La specifica completa del tipo di dato insieme di interi viene data in fig. **??**. Ancora una volta osserviamo che la manipolazione di insiemi attraverso queste funzioni garantisce

```
typedef lista set;

set isEmptySet(set A)
{ return isEmpty(A);}

set makeEmptySet()
{ return NULL;}

int belongsTo(int a, set A)
{ return isInOrd(A, a);}

set insert(int a, set A)
{ if !(isInOrd(A, a)) return addOrd(A, a);
  return A;
}

set union(set A, set B)
{ return elimCopieOrd(merge(A, B));}

set intersec(set A, set B)
{ return filtraOrd(A, B);}

set diff(set A, set B)
{ return diffListaOrd(A, B);}

set diffSimm(set A, set B)
{ return unione(diff (A, B), diff (B, A));}

set addSet(int a, set A)
{ return addElemOrd(A, a);}
```

Figure 3: Definizione della Struttura Dati Insieme

che venga mantenuto l'invariante del tipo di dato insieme, mentre accessi che sfruttano maldestramente l'interscambiabilità dei tipi `lista` e `set` (ad esempio usando la funzione `append` su due insiemi), distruggerebbero questo invariante, minando le precondizioni che garantiscono la correttezza delle funzioni su insiemi.

3.1 Detour: Side Effects

Finiamo con una importante osservazione legata ad uno sgradevole fenomeno della famiglia di linguaggi a cui appartiene il `C`, i *linguaggi imperativi*[6], che è noto col nome di *side effect*: cosa accadrebbe se definissimo la funzione `diff`, usando la versione "distruttiva" in place `diffListaOrdPlace`? La funzione `diffSimm` smetterebbe di funzionare. Questo perché la prima chiamata `diff(A,B)` distruggerebbe in modo irreversebile la lista che rappresenta

[6] per altro di gran lunga i più diffusi!

l'insieme `A`, rendendo il risultato della seconda chiamata `diff(B,A)` assolutamente imprevedibile. Supponete ad esempio che $A = \{1, 2\}$ e $B = \{2, 3\}$: noi ci aspettiamo che la differenza simmetrica, definita algebricamente come segue:

$$A \nabla B = (A \backslash B) \cup (B \backslash A)$$

dia come risultato l'insieme $\{1, 3\}$. Tuttavia dopo la prima chiamata la lista che rappresenta A, sarebbe corrotta e conterrebbe solo l'elemento 1. Per cui la seconda chiamata a `diff`, produrrebbe erroneamente l'insieme $\{2, 3\}$ e il risultato finale sarebbe $\{1, 2, 3\}$. Questo mostra una volta di più che le variabili dei linguaggi di programmazione imperativi **non** sono le variabili immutabili della matematica. E nella valutazione della correttezza dei programmi occorre stare molto attenti a possibili effetti collaterali, nascosti a prima vista.

Il fenomeno è particolarmente perverso nel caso di strutture dinamiche come liste o alberi, rappresentati con puntatori: in effetti il parametro lista viene passato *per valore*, passando il puntatore al primo elemento della lista. Infatti la seguente funzione:

```
void tail1(lista L)
{ L = L->next;}
```

non ha alcun effetto, quando invocata! Per ottenere una modifica dello stato visibile dal chiamante, dovremmo passare la lista *per indirizzo*, passando un puntatore (che in effetti è un puntatore a puntatore):

```
void tail2(lista *L)
{ *L = *L->next;}
```

L'altra tecnica che abbiamo già visto ampiamente quella di far tornare un valore alla funzione per comunicare con il chiamante:

```
lista tail3(lista L)
{ return L->next;}
```

per cui l'istruzione `tail2(&M)` risulta equivalente all'assegnazione `M=tail3(M)`. Tuttavia **tutta** la memoria puntata raggiungibile da `L` è di fatto *passata implicitamente per indirizzo*. Per fare un altro esempio minimale, la funzione:

```
void twiceFirst(lista L)
{ L->elem*=2;}
```

effettivamente raddopia l'informazione contenuta nel primo nodo di `L`, anche se la lista è passata per valore; analogamente anche una eventuale modifica del pointer `L->next`, non è affatto innocua come la modifica di `L`. Infatti una invocazione alla funzione:

```
lista cutTail(lista L)
{ L->next=NULL;}
```

stacca la coda della lista! Quindi, massima attenzione!

3.2 Equivalenza di Tipi

La definizione di nuovi tipi attraverso l'istruzione `typedef`, pone una questione importante relativa all'intercambiabilità di variabili di tipo diverso, ossia alla *compatibilità* tra tipi. Il linguaggio `C` ha un atteggiamento molto liberale rispetto ai tipi ed il lettore dovrebbe esserne cosciente, avendo osservato fenomeni come il seguente: assegnando un valore di tipo `char` ad una variabile dichiarata di tipo `int` (o passando un valore di tipo `char` come parametro attuale ad una funzione in cui è dichiarato un parametro formale di tipo `int`), viene automaticamente effettuata una *conversione di tipo* e la variabile di tipo intero (o il paramentro formale) assumeranno come valore il codice ASCII del carattere.

Abbiamo anche abbondantemente approfittato di questo fatto nella definizione del tipo di dato insieme, passando a funzioni sulle liste, valori di tipo insieme. Questo atteggiamento sui tipi detto *equivalenza per struttura*, ha l'indubbio vantaggio di essere molto flessibile e permette di riusare pezzi di codice che trattano tipi di dato diversi, ogni qualvolta i tipi di dato hanno una struttura compatibile.

Lo svantaggio è che non c'è nessuna difesa contro accessi incoerenti alla struttura dati: infatti, nessuno vieta di fare l'`append` tra due insiemi, ottenendo una lista in generale nè ordinata, nè senza ripetizioni. Altri linguaggi, seguono un principio, l'*equivalenza per nome*, in cui vengono ritenuti equivalenti solo tipi che hanno lo stesso nome: l'idea è che se il programmatore ha chiamato in modo diverso la stessa struttura dati, non vuole che vengano mescolati valori strutturalmente uguali, ma logicamente appartenenti a mondi diversi: si tratta probabilmente di dati che hanno la stessa *rappresentazione*, ma che devono essere *interpretati* in modo diverso.

Per chiarire il concetto di rappresentazione ed interpretazione, ricordiamo che le celle di memoria di un calcolatore elettronico contengono sempre sequenze binarie: a seconda di come queste sequenze vengono interpretate, vi possiamo leggere la codifica di un numero intero o di un carattere o di un puntatore.

4 Tipi di Dato Generici

Alcuni moderni linguaggi di programmazione permettono di scrivere programmi che sono parametrici rispetto ad un tipo di dato. Funzioni come `addHead` o `append` sulle liste non dipendono in nessun modo dal fatto che le liste considerate siano di interi. Sarebbe interessante poter scrivere il codice di queste funzioni e poterlo utilizzare su liste che contengono elementi di qualsiasi tipo: funzioni con queste caratteristiche vengono dette *polimorfe*. Il linguaggio `C` non supporta il polimorfismo, ma permette ugualmente di scrivere funzioni polimorfe usando uno stratagemma, la compatibilità del tipo `void *` con qualsiasi altro tipo puntatore.

Osserviamo tuttavia, che le funzioni sulle liste ordinate necessitano di utilizzare l'operatore di minore, definito sugli interi (e su quasi tutti i tipi base), ma non necessariamente definito per un tipo definito dall'utente. Inoltre, se volessimo scrivere le funzioni per liste ordinate che contengano campi informazione di tipo `void *` per valutare il test di minoranza, è necessario *dereferenziare* le variabili di tipo `void *` (`*p < *q`), ma questo è proibito: infatti il compilatore, non conoscendo quanta memoria sia necessaria per memorizzare gli oggetti in questione, non è in grado di ricostruire il valore puntato, e quindi non è mai possibile applicare l'operatore di dereferenziazione `*` a variabili di tipo `void *`.

Sarà necessario, per poter scrivere il tipo di dato "lista ordinata generica", parametrizzare

le funzioni con una funzione `minore`, e passare opportunamente una funzione come ulteriore parametro che esegue il test di minoranza tra gli oggetti che in quel momento sono contenuti nella lista.

4.1 Pile generiche

In questa sezione vedremo, come definire un tipo di dato *pila* o *stack*. Una pila è una struttura dati sequenziale su cui si impone il vincolo che inserimenti, cancellazioni e letture possano avvenire *solo* in testa. Il nome è chiaramente reminiscente di una pila di fogli o libri, di cui possiamo vedere o prendere solo il foglio in cima alla pila. La pila segue in sostanza una disciplina LIFO (last in, first out), tipica tra l'altro dei linguaggi di programmazione: la prima parentesi chiusa si riferisce all'ultima parentesi aperta e la prima attivazione ricorsiva di una funzione sarà l'ultima da cui si rientrerà. Vedremo infatti, come applicazione delle pile, l'implementazione della ricorsione e una tecnica per trasformare qualsiasi programma ricorsivo in iterativo (sez. **??**).

Insistiamo sul fatto che la specifica del tipo di dato pila, non consiste di in una struttura dati adatta a memorizzare pile, ma in un preciso *comportamento*, definito dalle operazioni che permettono di accedere ad una pila. E sarà importante che le operazioni seguano la disciplina di accesso fissata e non ci siano accessi incoerenti alla struttura dati. Vediamo quali sono le operazioni che caraterrizzano il tipo di dato pila:

```
stack createStack();
/* restituisce una pila vuota  */

int isEmpty(stack);
/* testa se una pila e' vuota */

void pop(stack);
/* rimuove l'elemento in cima alla pila */

void* top(stack);
/* restituisce l'elemento in cima alla pila
 * (senza rimuoverlo!)
 */

void push(stack, void*);
/* inserisce un elemento in cima alla pila */
```

Si osservi che gli elementi che vengono memorizzati nella pila sono di tipo `void *`. Quindi, se vi vogliamo inserire interi, dovremmo passare puntatori a interi. Ma grazie alla compatibilità del tipo `void *` con ogni altro puntatore, potremmo memorizzare dati di qualsiasi tipo (e dimensione), semplicemente passando un puntatore a questi dati, e ciò permetterà di usare il tipo pila senza dover riscrivere il codice a seconda del tipo di dati che vogliamo mettere nella pila. Infine osserviamo che le operazioni di estrazione, lettura, cancellazione e inserimento non dipendono in nessun modo dal tipo di dato che memorizzo nella pila. In effetti, le variabili di tipo `void *` non si possono *dereferenziare*, cioè non si può applicare l'operatore `*`. Vedremo nel seguito come sia possibile generalizzare questo approccio al caso in cui sia necessario usare degli operatori (immaginate di voler scrivere funzioni sulle liste

ordinate generiche: servirà parametrizzare tutte le funzioni rispetto alla funzione `minore`). Scegliamo di rappresentare una pila come una lista concatenata, accessoriata di un nodo speciale che mantiene un puntatore alla cima della pila ed altri dati, come il numero di elementi (opzionale). Negli esercizi verrà chiesto al lettore di fornire implementazioni che usano array. Un significativo esercizio sperimentale è quello di verificare che se l'interfaccia delle operazioni rimane immutata, i programmi che usano le pile continuanao a funzionare correttamente, indipendentemente da quale sia l'implementazione sottostante. Vediamo ora l'implementazione della struttura dati, mentre l'implementazione delle operazioni è in figura ??:

```
typedef struct S
   { void *elem;
     struct S *next;
     } Snode;

typedef struct
   { Snode* top;
     int dimstackMax;
     int numElem;
     } stackDescriptor;

typedef stackDescriptor *stack;
```

4.2 Code generiche

Accenniamo brevemente al fatto che, modificando la disciplina di ingresso/uscita, e considerando una disciplina FIFO (*first-in-first-out*) otteniamo un altro tipo interessante; le *code*. Daremo solo la specifica del tipo di dato (cioè le operazioni che definiscono le pile) e invitiamo il lettore a completare l'implementazione in analogia con l'implementazione delle pile. Useremo le code per implementare le visite per livelli degli alberi.

```
queue createQueue();
/* restituisce una coda vuota  */

int isEmptyQ(queue);
/* testa se una coda e' vuota */

void dequeue(queue);
/* rimuove il primo elemento della coda */

void* first(queue);
/* restituisce il primo elemento della coda
 * (senza rimuoverlo!)
 */

void enqueue(queue, void*);
/* inserisce un elemento in fondo alla coda */
```

```
stack createStack(int dimMax)
/* crea una pila vuota; */
{ stack Sptr;
   Sptr = malloc(sizeof(stackDescriptor)); /* alloca la memoria per il descrittore della pila
   Sptr->top = NULL;                       /* cima della pila a NULL  */
   Sptr->dimstackMax = dimMax;
   Sptr->numElem = 0;                      /* numero di elementi a 0  */
   return Sptr;
 }

int isEmpty(stack S)
/* testa se la pila e' vuota
{ return S->numElem==0;}
/* alternativamente si poteva scrivere: return !S->top; e' equivalente se e' mantenuto un giusto
*/

int isFull(stack S)
/* testa se una pila e' piena */
{ return S->numElem == S->dimstackMax;}

void pop(stack S)
/* estrae la cima della pila, SENZA tornare il valore */
{ Snode *tmp;
  S->numElem--;  /* decrementa numElem mantenendo l'invariante del tipo di dato */
  tmp = S->top;            /* salvo il putatore al primo elemento */
  S->top = (S->top)->next; /* modifico il puntatore al top della pila */
  free(tmp);               /* libero la memoria occupata dal nodo rimosso */
 }

void* top(stack S)
/* ritorna l'elemento memorizzato in cima alla pila */
{ return (S->top)->elem;
}

void push(stack S, void* el)
{ node *tmp;
  tmp = malloc(sizeof(node));     /* alloco memoria per inserire un nuovo nodo
  tmp->next = S->top; /* il link del nuovo nodo punta al vecchio top */
  tmp->elem = el;     /* carico il campo informazione */
  S->top = tmp;       /* modifico il puntatore al top della pila */
  S->numElem++;       /* incremento il numero di elementi della pila */
  }

int howManyStack(stack S)
{ return S->numElem;}
```

Figure 4: Definizione del tipo di dato Pile Generiche

Osservate che la definizione della struttura dati è molto simile a quella delle pile, con la differenza che per motivi di efficienza, conviene tenere due puntatori, uno alla testa ed uno al fondo della coda.

```
typedef struct Q
   { void *elem;
     struct Q *next;
     } Qnode;

typedef struct
   { Qnode* first;
     Qnode* last;
     int dimqueueMax;
     int numElem;
     } queueDescriptor;

typedef queueDescriptor *queue;
```

Esercizi

1. Completare la rappresentazione del tipo di dato *insieme di interi* attraverso liste ordinate.

2. Completare la rappresentazione del tipo di dato *coda*.

3. Implementare pile e code usando gli array come struttura dati sottostante. Valutare pregi e difetti delle due implementazioni. Scrivere un programma che usa le code e verificare che sostituendo il modulo che implementa le code, il programma continua a funzionare correttamente (ovviamente dovete aver mantenuto le stesse operazioni con gli stessi nomi, ossia la stessa *interfaccia*).

5 Alberi Binari

Vediamo ora un'altra struttura dati estremamente importante in informatica, gli alberi.

Definizione 5.1 Una *albero binario* con nodi etichettati con elementi di tipo T è:

1. l'albero vuoto (cioè l'albero senza alcun nodo);

2. oppure una radice etichettata con un elemento di tipo T, con due sottoalberi chiamati rispettivamente sottoalbero sinistro e destro.

Ancora una volta, troviamo una rappresentazione adeguata a rappresentare questo tipo di dato, usando puntatori laddove la definizione del tipo albero riferisce a se stessa:

```
typedef struct tNode
{ struct tNode *left;
  int info;
  struct tNode *right;
```

```
  } treeNode;

typedef treeNode * tree;
```

Vediamo innanzitutto quali sono le funzioni base che permettono di costruire e decomporre un albero binario. Chiaramente dovremmo avere:

1. un modo per accedere alle componenti di un albero: radice, sottoalbero destro e sinistro;
2. un modo per verificare se un albero sia vuoto o meno;
3. un modo per costruire un albero, fornendo due alberi e un intero.

Esistono molti modi per definire funzioni che accedono alle sottocomponenti di un albero. Vediamone uno un po' originale: scriviamo un'unica funzione `isEmptyTree` con 4 parametri, che restituisce 1 se l'albero dato in ingresso (sul primo parametro) è l'albero vuoto, e 0 altrimenti: nel caso la risposta sia 0 (quindi l'albero NON è vuoto) userà i rimanenti tre paramentri (passati per indirizzo!) per comunicare al chiamante radice, albero sinistro e destro:

```
int isEmptyTree(tree T, int * r, tree * L, tree * R)
{ if (!T) return 0
     else
       { *L = T->left;
         *R = T->right;
         *r = T->info;
         return 1;
        }
 }
```

Per costruire tutti gli alberi binari etichettati con interi, devo saper costruire l'albero vuoto e dati un intero r e due alberi L ed S, devo saper costruire l'albero con radice r ed alberi sinistro e destro L ed R rispettivamente:

```
tree makeTree(int r, tree L, tree R)
{ tree tmp = malloc(sizeof(treeNode));

  tmp->info = r;
  tmp->left = L;
  tmp->right = R;
  return tmp;
}

tree makeEmptyTree()
{ return NULL;}
```

Come al solito non saremo mai così integralisti da accedere alla struttura dati solo attraverso queste funzioni. Vediamo tuttavia qualche esempio di semplice funzione che usa i costruttori e i distruttori definiti sopra:

```
int isLeaf(tree T)
/* POST: restituisce 1 se T e' una foglia
 */
{ int radix;
  tree left, right;

  if (isEmptyTree(T, &radix, &left, &right)) return 0;
     else return (!left && !right);
}

int depth (tree T)
/* POST: restituisce la profondita' di T
 */
{ int radix;
  tree left, right;

  if (isEmptyTree(T,&radix, &left, &right)) return -1;
  else return 1 + max(depth(left), depth(right));
}

int howManyNodes (tree T)
/* POST: restituisce il numero di nodi di T
 */
{ int radix;
  tree left, right;

  if (isEmptyTree(T,&radix, &left, &right)) return 0;
  else return 1 + howManyNodes(left) + howManyNodes(right));
}
```

5.1 Visite in Profondità

Un tipico problema sugli alberi, consiste nel visitare tutti i nodi dell'albero. Vedremo brevemente, come stampare in diversi ordini i nodi dell'albero. Le visite in profondità che vengono considerate in letteratura sono:

1. visita *in-order* o *simmetrica*: si visita prima il sottoalbero sinistro, quindi la radice ed infine il sottoalbero destro;
2. visita *pre-order* o *anticipata*: si visita prima la radice, poi il sottoalbero sinistro ed infine il sottoalbero destro;
3. visita *post-order* o *differita*: si visitano prima i sottoalberi ed infine la radice.

Come già osservato sulle liste, dipende se la stampa del nodo viene fatta all'"andata delle chiamate ricorsive che percorrono l'albero, o invece al ritorno. Osserviamo che le funzioni `depth` e `howManyNodes` sono funzioni che adottano una strategia analoga alla visita differita:

prima computano il loro valore sui sottoalberi e poi combinano questi valori (ed eventualmente quello contenuto nella radice). Vedremo nel seguito funzioni che traggono invece vantaggio dal trasmettere valori "in avanti" alle attivazioni della funzione sui sottoalberi.

```
void printInOrder(tree T)
{ if (T)
    { printInOrder(T->left);
      printf("%3d", T->info);
      printInOrder(T->right);
   }
}
```

```
void printPreOrder(tree T)
{ if (T)
    { printf("%3d", T->info);
      printPreOrder(T->left);
      printPreOrder(T->right);
     }
 }
```

```
void printPostOrder(tree T)
{ if (T)
    { printPostOrder(T->left);
      printPostOrder(T->right);
      printf("%3d", T->info);
     }
 }
```

Risulta interessante vedere anche come sia possibile preso in ingresso un albero, restituire come output una lista che contiene una visita in profondità:

```
lista inOrder(tree T)
{ int radix;
  tree left, right;

  if (isEmptyTree(T,&radix, &left, &right)) return NULL;
  else return append(preorder(left), addHead(preorder(right), radix));
 }

lista preOrder(tree T)
{ int radix;
  tree left, right;

  if (isEmptyTree(T,&radix, &left, &right)) return NULL;
  else return addHead(append(preorder(left), preorder(right)), radix);
 }

lista postOrder(tree T)
{ int radix;
  tree left, right;

  if (isEmptyTree(T,&radix, &left, &right)) return NULL;
  else return addTail(append(preorder(left), preorder(right)), radix);
 }
```

Concludiamo la sezione con un ultimo problema, sempre legato alle visite: dato un albero, restituire la lista che contiene tutte le sue foglie, da destra a sinistra:

```
lista frontier(tree T)
/* POST: restituisce una lista con tutte le foglie di T */
{ int radix;
  tree left, right;

  if (isEmptyTree(T,&radix, &left, &right)) return NULL;
  else if isLeaf(T) return addHead(NULL, radix);
  else return append(frontier(left), frontier(right));
 }
```

5.2 Visita per Livelli

Visitare un albero per livelli significa visitare i nodi dell'albero, cominciando dalla radice, e poi visitare tutto il primo livello, poi il secondo e così via. La difficoltà sta appunto nel fatto che l'ordine in cui si visitano i nodi non ha alcuna relazione con la rappresentazione degli alberi: infatti ogni nodo è collegato col padre e i figli, ma non con gli altri nodi che stanno sullo stesso livello.

La tecnica consiste nel mantenere una coda coi sottoalberi ancora da visitare. Ad ogni iterazione, prendere il primo sottoalbero della coda, visitare la radice e mettere i sottoalberi in fondo alla coda. Così facendo, sottoalberi che hanno la radice allo stesso livello, saranno sempre memorizzati in nodi adiacenti nella coda.

Osservate che la lunghezza della coda di norma, cresce durante l'algoritmo (finchè non si arriva ai sottoalberi vuoti) e quindi non è una buona funzione di terminazione. La misura che decresce è il numero complessivo dei nodi dei sottoalberi memorizzati nella coda.

Osservate che tutti i nodi degli alberi messi sulla coda, infatti, sono esattamente tutti i nodi ancora da visitare. Questa è anche la proprietà invariante del ciclo.

```
lista visitaLivelli(tree T)
{ coda C = createQueue();
  lista L = NULL;
  tree L, R, U;

  enqueue(C, T); /* metto nella coda l'albero da visitare
  while (!isEmptyQueue(C))
  { U = first(C);
    if (!isEmptyTree(U, &r, &L, &R)
{ L = addTail(r, L);
  dequeue(C);
  enqueue(C, L);
  enqueue(C, R);
  }
    }
  return L;
}
```

5.3 Bilanciamento

Definizione 5.2 Un albero è *bilanciato in altezza* se:

1. è l'albero vuoto;

2. i sottoalberi destro e sinistro sono bilanciati in altezza e la differenza delle loro altezze differisce al più di 1.

Definizione 5.3 Un albero è *bilanciato nel numero dei nodi* se:

1. è l'albero vuoto;

2. i sottoalberi destro e sinistro sono bilanciati nel numero dei nodi e la differenza dei numeri dei nodi differisce al più di 1.

Molte applicazioni degli alberi generano algoritmi la cui complessità è proporzionale alla *profondità* dell'albero. E' facile rendersi conto che, presi due alberi con lo stesso numero di nodi, sarà più profondo l'albero meno bilanciato.

Occupiamoci ora di scrivere due funzioni che verificano il bilanciamento di un albero, `depthBalanced` per il bilanciamento in altezza e `NNbalanced` per il bilanciamento nel numero dei nodi. Forti delle funzioni `depth` e `howManyNodes` possiamo scrivere due funzioni molto eleganti che seguono sostanzialmente lo schema di programmazione già abbondantemente visto in precedenza, e cioè, stabilire il valore della funzione sul caso base e calcolare il valore della funzione su alberi non vuoti, componendo in modo opportuno i valori calcolati componendo opportunamente i valori computati dalle chiamate ricorsive sui sottoalberi:

```
int depthBalanced(tree T)
/* POST: restituisce 1 se l'albero e' bilanciato in profondita'
 * 0 altrimenti
 */
{ int r;
  tree L, R;

  if (isEmpty(T, &r, &L, &R) return 1;
  else if (depthBalanced(L)
          if (depthBalanced(R))
             if (abs(depth(L)-depth(R))<=1) return 1;
  /* se tutti i controlli precedenti danno esito negativo ... */
  return 0;
}

int NNBalanced(tree T)
/* POST: restituisce 1 se l'albero e' bilanciato nel numero dei nodi
 * 0 altrimenti
 */
{ int r;
  tree L, R;

  if (isEmpty(T, &r, &L, &R) return 1;
  else if (NNBalanced(L)
          if (NNBalanced(R))
             if (abs(howManyNodes(L)-howManyNodes(R))<=1) return 1;
```

```
  /* se tutti i controlli precedenti danno esito negativo ... */
  return 0;
}
```

Le due funzioni precedenti sono molto eleganti, ma hanno il difetto di eseguire numerose visite inutili di sottoalberi. E' infatti possibile scrivere una funzione ricorsiva che verifica il bilanciamento, attraversando una sola volta tutti i nodi dell'albero. In entrambi i casi, si osservi che l'informazione sintetica restituita dalle funzioni (bilanciato/non bilanciato) costringe a riesaminare i sottoalberi con le funzioni che calcolano profondità e numero dei nodi.

Quindi, riscriveremo le funzioni in modo che contemporaneamente, in un unica scansione, la funzione verifichi il bilanciamento e calcoli rispettivamente la profondità e il numero dei nodi. Vedremo due tecniche per "restituire" più informazioni:

1. in `depthBalancedEff` useremo l'uso di un parametro ausiliario passato per *indirizzo*: nel caso in cui un albero sia bilanciato, questo paramentro viene opportunamente aggiornato con l'altezza dell'albero;

2. in `NNBalancedEff` useremo il valore ritornato per *codificare* entrambe le informazioni: se la funzione ritorna un numero maggiore o uguale a 0, significa che l'albero è bilanciato e il valore ritornato è il numero dei nodi dell'albero. Altrimenti viene ritornato il valore -1 (osservare che questo valore è diverso da qualsiasi possibile numero di nodi dell'albero).

```
int depthBalancedEff(tree T, int *h)
{ int hr, hl;
  int r;
  tree L, R;

  if (isEmpty(T, &r, &L, &R)
    { *h = -1;
      return 1;
     }
  else if (depthBalancedEff(T->left,&hl))
          if (depthBalancedEff(T->right,&hr))
             if (abs(hr - hl)<=1)
                { *h = 1+max(hl,hr);
                  return 1;
                 }
return 0;
}

int NNbalancedEffAux(tree T)
 { int nnr, nnl;
   int r;
   tree L, R;

   if (isEmpty(T, &r, &L, &R) return 0;
   else if ((nnl=NNbalancedAux(L))>=0)
```

```
            if ((nnr=NNbalancedAux(R))>=0)
                if (abs(nnr-nnl)<=1) return 1+nnl+nnr;
   return -1;
}
```

Affinché il chiamante abbia l'interfaccia che si aspetta (1 se l'albero e' bilanciato e zero altrimenti) possiamo definire una funzione che si limita a trasformare i risultati della funzione `NNbalancedEffAux` in modo opportuno:

```
int NNbalancedEff(tree T)
{ if (NNbalancedEffAux(T)>=0) return 1;
  else return 0;
 }
```

5.4 Relazioni di Uguaglianza e SottoAlbero

Concludiamo questa sezione, con tre problemini: verificare se due alberi binari sono uguali, se uno e' sottoalbero dell'altro

```
int eqTree(tree T, tree U)
/* verifica se gli alberi T ed U sono uguali, cioe'
 * se hanno la stessa struttura e gli stessi campi
 * informazione
 */
{ if (!T && !U) return 1;
else if (!T || !U) return 0;
     /* siccome la condizione !T && !U e' falsa nel
      * ramo else, se la condizione !T || !U
      * e' verificata allora UNO SOLO tra T ed U
      * e' un albero NON VUOTO
      */
else if (T->info==U->info)
 if (eqTree(T->left, U->left))
  return eqTree(T->right, U->right);
     else return 0;
 else return 0;
}
```

```
int subTree(tree T, tree U)
/* verifica se T e' sottoalbero di U
 * OSSERVARE che questa funzione sospende le chiamate
 * ricorsive non appena viene trovato in U un sottoalbero
 * uguale ad T
 */
{ if (!U) return 0;
  else if (eqTree(T, U)) return 1;
  else if (subTree(T, U->left)) return 1;
  else if (subTree(T, U->right)) return 1;
  else return 0;
}
```

5.5 Alberi n-ari e tipi di dato mutuamente ricorsivi

Poniamoci ora il problema di generalizzare il concetto di albero, prevedendo una struttura in cui ciascuno nodo può avere un numero arbitrario di figli. Una definizione molto elegante è la seguente, che definisce in modo *mutuamente induttivo* il concetto di albero e di foresta[7]:

Definizione 5.4 Una *foresta* è:

1. la foresta vuota;
2. oppure un albero, seguito da una foresta.

Una *albero n-ario* con nodi etichettati con elementi di tipo T è una radice con una foresta di sottoalberi:

Il lettore più smaliziato osserverà che una foresta non è nient'altro che una lista di alberi, e quindi è possibile evitare la mutua induzione semplicemente riferendosi al già definito concetto di lista. Tuttavia è interessante osservare che è possibile nel linguaggio `C` implementare definizioni di tipo mutuamente induttive, che generalmente vengono manipolate attraverso funzioni mutuamente ricorsive. L'unica avvertenza da seguire è quella di informare preventivamente il compilatore che un certo nome rappresenta il nome di un tipo (nel nostro caso di una struttura), per poterlo usare nella definizione dell'altro tipo e solo dopo darne la definizione completa (nel nostro caso si informa il compilatore che `fNode` è il nome di una struttura, mentre si da la definizione della struttura `fNode`, solo dopo aver dato la definizone del tipo `nTreeNode`). Osservate anche che nella definizione di `nTreeNode` non sarebbe legale riferie alla struttura `fNode`, perché il compilatore non saprebbe quanto spazio allocare. E' invece legale dichiarare un campo che è un puntatore alla struttura `fNode`.

```
struct fNode;

typedef struct nTNode
{ int rad;
  struct fNode *subT;
 } nTreeNode;

typedef struct fNode
{ nTreeNode first;
  struct fNode* next;
  } forestNode;

typedef nTreeNode nTree;
typedef forestNode * forest;
```

Vediamo innanzitutto quali siano le funzioni base che permettono di costruire e decomporre un albero n-ario. Come sempre sarà necessario avere delle funzioni per costruire alberi e foreste e funzioni che permettono di accedere alle sottocomponenti.

[7] Osservare che rispetto alle definizioni date a lezione, l'albero vuoto è rappresentato dalla foresta vuota. Questo implica che nella definizione `C` gli alberi non siano pointer a `nTreeNode`, ma semplicemente degli `nTreeNode`

```
nTree makeTree(int r, forest F)
{ nTree * tmp;
  tmp = malloc(sizeof(nTreeNode));
  tmp->rad = r;
  tmp->subT = F;
  return *tmp;
}

forest createEmptyForest()
{ return NULL;}

forest addTree(nTree T, forest F)
{ forest tmp;
  tmp = malloc(sizeof(forestNode));
  tmp->first = T;
  tmp->next = F;
  return tmp;
  }

forest subtrees(nTree T)
{ return T.subT;}

nTree firstTree(forest F)
/*PREC: Foresta F non vuota */
{ if (F) return F->first;
    else printf("ERRORE!! foresta vuota");
}

forest tailForest(forest F)
/*PREC: Foresta F non vuota */
{ if (F) return F->next;
    else printf("ERRORE!! foresta vuota");
}

int isEmptyForest(forest F, nTree *firstTree, forest* tailForest)
{ if (!F) return 1;
else
   { *firstTree = F->first;
     *tailForest = F->next;
     return 0;
    }
}

int radix(nTree T)
{ return T.rad;}
```

Come abbiamo detto, la mutua ricorsione della definizione del tipo albero n-ario si traduce nella mutua ricorsione delle funzioni che devono analizzare la struttura di un albero

n-ario. Un esempio paradigmatico è dato dalle visite: vediamo la visita anticipata, lasciando al lettore la scrittura della procedura di visita posticipata. La visita simmetrica invece, non ha un senso preciso per gli alberi n-ari.

```
void preOrderTree(nTree T)
{   printf("%d.", radix(T));
    preOrderForest(subtrees(T));
  }

void preOrderForest(forest F)
{ nTree first;
  forest tail;

  if (!isEmptyForest(F, &first, &tail))
  { preOrderTree(first);
    preOrderForest(tail);
    }
 }
```

Ricordando la struttura di lista di una foresta, possiamo scrivere un algoritmo che evita la mutua ricorsione scrivendo una funzione che scorre iterativamente la foresta, esattamente come si scorre una lista:

```
void preOrderIter(nTree T)
{  forest F = subtrees(T);
   nTree first;
   forest tail;

   printf("%d", radix(T));
   while (!isEmptyForest(F, &first, &tail))
   { postOrderIter(first);
     F = tail;
     }
}
```

Anche altre funzioni, come la profondità, possono essere elegantemente scritte usando funzioni mutuamente ricorsive:

```
int depthTree(nTree T)
{ return 1+depthForest(subtrees(T), -1);}

int depthForest(forest F, int h)
{ nTree first;
  forest tail;

  if (isEmptyForest(F, &first, &tail)) return h;
  else return depthForest(tail, max(depthTree(first),h));
}
```

Il lettore è invitato a scrivere la versione iterativa `depthTree`.

Esercizi

1. Considerare la seguente definizione di albero binario:

   Definizione 5.5 Una *albero binario* con nodi etichettati con elementi di tipo T è:

   (a) una foglia etichettata con un elemento di tipo T;

   (b) oppure una radice etichettata con un elemento di tipo T, con due sottoalberi chiamati rispettivamente sottoalbero sinistro e destro.

   Riflettere su quali alberi vengono definiti e confrontare con l'insieme degli alberi definiti dalla definizione **??**. Scrivere le opportune funzioni costruttori e distruttori che garantiscano che gli alberi ottenuti appartengano a questa famiglia di alberi.

2. ♣Dimostrare per induzione che se un albero è bilanciato nel numero dei nodi, allora è bilanciato in profondità.

3. Definire una struttura dati per memorizzare un labirinto ed implementare le funzioni usate dall'algoritmo nella sottosezione **??**.

4. Considerare il tipo di dato *data*: riflettere su quali sono le operazioni elementari ed implementarle in `C`.

5. Considerare il tipo polinomi a coefficienti interi. Descrivere quali sono le operazioni e trovare una struttura dati adeguata per implementarle in modo efficiente.