Le Liste*

Maurizio Palesi

Sommario

La scelta delle appropriate strutture dati è di fondamentale importanza per la risoluzione di un certo problema almeno tanto quanto un buon programma di manipolazione. Le liste rappresentano una struttura dati flessibile ed efficiente per la gestione delle informazioni. In questo documento introdurremo il concetto di lista, la loro rappresentazione sia statica che dinamica e l'implementazione delle primitive di gestione in linguaggio C.

Indice

1 Definizione di Lista

Le liste rappresentano una tra le più semplici strutture dati utilizzate per la memorizzazione e la gestione di informazioni. Una lista è una sequenza di elementi, aventi le seguenti caratteristiche:

*Questo documento è stato redatto utilizzando come rifermiento l'ADT Lista realizzato dal Prof. Salvatore Cavalieri `http://www.diit.unict.it/users/scava`.

- Tutti gli elementi sono omogenei, ossia appartengono allo stesso tipo.
- Ciascun elemento della lista è contraddistinto da una posizione. In base alla posizione di ciascun elemento nella lista, è possibile individuare relazioni di precedenza e di successione tra gli elementi. Particolari elementi sono quello di testa che occupa la prima posizione e quello di coda che occupa l'ultima posizione. È ovvio che il primo elemento della lista non ha predecessore, e l'ultimo elemento non ha successore.

Una lista è un insieme finito e ordinato di oggetti

$$\langle x_0, x_1, x_2, \ldots, x_{k-1}, x_k, x_{k+1}, \ldots, x_{n-1} \rangle \tag{1}$$

messi in corrispondenza con l'insieme dei primi n numeri naturali in modo che ciascun elemento, ad eccezione del primo e dell'ultimo, abbia un *predecessore* ed un *successore* nella lista. Nel caso in cui la lista non contenga alcun elemento si parla di *lista vuota.*

2 Operazioni Primitive sulla Lista

Una lista è caratterizzata da alcune operazioni fondamentali. Da ora in poi si consideri la seguente simbologia: l sia una lista, p la generica posizione di un elemento della lista, x un generico elemento dello stesso tipo degli elementi presenti nella lista. Per fissare le idee si consideri la seguente lista di numeri interi:

$$l = \langle 12, 43, 21, 17, 19, 54, 3, 21 \rangle \tag{2}$$

Le seguenti operazioni primitive vengono definite:

- $End(l)$: Restituisce la posizione successiva alla posizione dell'ultimo elemento della lista. Tale posizione è quella in cui viene inserito un elemento che diverrà il successore dell'ultimo elemento della lista. Se la lista è vuota, $End(L)$ ritorna la prima posizione della lista. Per esempio, se l è la lista definita nella (2), $End(l)$ restituirà 8.
- $First(l)$: Restituisce la posizione del primo elemento della lista l. Nel caso in cui la lista è vuota ritorna $End(l)$. Per esempio, se l è la lista definita nella (2), $First(l)$ restituirà 0.
- $Insert(l, p, x)$: Inserisce l'elemento x in posizione p, spostando di una posizione più alta tutti gli elementi di posizione p e superiori. Per esempio, se l è la lista definita nella (2), $Insert(l, 1, -5)$ modificherà l in questo modo: $\langle 12, -\mathbf{5}, 43, 21, 17, 19, 54, 3, 21 \rangle$. La funzione non è definita quando p non esiste.
- $Delete(l, p)$: Elimina l'elemento della lista di posizione p, spostando di una posizione in basso tutti gli elementi di posizione superiore a p. Per esempio, se l è la lista definita nella (2), $Delete(l, 4)$ modificherà l in questo modo: $\langle 12, 43, 21, 17, 54, 3, 21 \rangle$. La funzione non è definita quando p non esiste o quando p coincide con $End(l)$.

- $Locate(l, x)$: Restituisce la posizione dell'elemento x. Se x compare nella lista più di una volta, allora viene restituita la posizione della prima occorrenza di x. Se x non compare nella lista allora viene restituita la posizione $End(l)$. Per esempio, se l è la lista definita nella (2), $Locate(l, 21)$ restituisce 2, $Locate(l, 52)$ restituisce 8.
- $Retrieve(l, p)$: Restituisce l'elemento della lista di posizione p. La funzione non è definita quando p non esiste o $p = End(l)$. Per esempio, se l è la lista definita nella (2), $Retrieve(l, 3)$ restituisce 17.
- $MakeNull(l)$: Restituisce una lista vuota.
- $Empty(l)$: Restituisce il valore booleano vero se la lista l è vuota. Restituisce il valore falso altrimenti.
- $Full(l)$: Restituisce il valore booleano vero se la lista è piena, ossia non è possibile inserire nuovi elementi. Restituisce il valore falso altrimenti.
- $Next(l, p)$: Restituisce la posizione dell'elemento successivo all'elemento di posizione p. Per esempio, se l è la lista definita nella (2), $Next(l, 2)$ restituisce 3. La funzione non è definita quando p non esiste o p coincide con $End(l)$.

3 Realizzazione di una Lista

La realizzazione di una lista, consiste nella sua rappresentazione attraverso i tipi di dati disponibili nei linguaggi di programmazione, ed in particolare in C.

Esistono due modalità realizzative di una lista: *sequenziale* e *collegata*. La realizzazione sequenziale prevede che la sequenzialità degli elementi della lista venga rappresentata dalla adiacenza delle locazioni di memoria utilizzate per contenere ciascun elemento della lista. Una tipica realizzazione sequenziale è attraverso un tipo di dato vettore. La realizzazione collegata si basa sull'idea di rappresentare la sequenzialità degli elementi della lista attraverso l'uso di locazioni di memoria ciascuna delle quali contiene due informazioni:

- L'elemento della lista;
- L'indicazione della locazione di memoria contenente l'elemento successivo.

Nel seguito verrà supposto che ciascun elemento della lista sia di tipo intero, ma nulla cambia per tipi di dato diversi.

3.1 Rappresentazione Sequenziale

Come detto prima, la più tipica rappresentazione sequenziale avviene tramite l'uso di un vettore. Nel seguito verrà mostrata la realizzazione sequenziale della lista tramite il tipo vettore reso disponibile dal linguaggio C.

```
#define N 10

typedef int BaseType;

struct List {
    BaseType info[N];
    int      last;
};
```

Allo scopo di rappresentare la sequenzialità degli elementi della lista, gli elementi della lista devono occupare componenti adiacenti del vettore. In questo modo, preso un qualunque elemento della lista, è possibile accedere al suo successore o predecessore considerando uno dei due componenti adiacenti (se esistono) del vettore. Utilizzando tale rappresentazione, l'elemento in testa alla lista viene memorizzato nel componente del vettore di indice 0 del vettore (ossia ha posizione 0), il secondo elemento della lista viene memorizzato nel componente di indice 1 (ossia la sua posizione è 1) e così via. In base a ciò, preso il generico elemento della lista in posizione i, il suo successore (se esiste) ha posizione $i + 1$, mentre il suo predecessore (se esiste) ha successore $i - 1$. L'elemento in testa ha posizione 0 e il suo successore ha posizione 1.

In questa rappresentazione, la posizione del primo elemento della lista è sempre la stessa ed è la posizione 0, mentre la posizione dell'ultimo elemento della lista dipende dalla lunghezza della lista. Allo scopo di individuare la posizione dell'ultimo elemento della lista, viene utilizzato una variabile che individua la posizione dell'ultimo elemento della lista. Tale variabile viene chiamata `last`. La struttura di nome `list`, dunque, contiene due campi. Il primo è il vettore a N componenti di tipo `BaseType`, metre il secondo è il puntatore `last`. Il campo `last` è di tipo intero visto che la posizione di ciascun elemento è un intero.

Le uniche due costanti da definire sono EMPTYLIST e UNDEFINED. La prima serve ad indicare che la lista è vuota. Visto che si è assunto di indicare con `last` la posizione dell'ultimo elemento della lista, la condizione di lista vuota , può essere rappresentata da un valore della variabile last inferiore a 0 (che indica la presenza di un solo elemento nella lista, in posizione 0). Ad esempio il valore `last=-1` può esere utilizzato per indicare la presenza di 0 elementi nella lista. La seconda a gestire eventuali condizioni di errore (per esempio la richiesta di accesso ad un indice non contenuto nella lista).

```
#define EMPTYLIST -1
#define UNDEFINED -2
```

Nelle sottosezioni successive implementeremo in linguaggio C le funzioni descritte nella Sezione 2.

3.1.1 End(l)

La funzione `End(l)` restituisce la posizione successiva alla posizione dell'ultimo elemento della lista. Tale posizione è quella in cui viene inserito un elemento che diverrà il successore dell'ultimo elemento della lista. Se la lista è vuota, $End(L)$ ritorna 0 cioè la prima posizione della lista.

```
int End(List *l)
{
    return(l->last+1);
}
```

3.1.2 First(l)

La funzione `First(l)` restituisce la posizione del primo elemento della lista. Se la lista è vuota restituisce `End(l)`.

```
int First(List *l)
{
    return(0);
}
```

3.1.3 MakeNull(l)

La funzione `MakeNull(l)` restituisce una lista vuota.

```
void First(List *l)
{
    l->last = EMPTYLIST;
}
```

3.1.4 Full(l)

La funzione `Full(l)` è definita per l'implementazione a vettore, visto che il numero di elementi della lista che possono essere memorizzati è limitato. Tale funzione fornisce 1 se non vi è più spazio per la memorizzazione di elementi. Ciò accade quando l'ultimo elemento occupa la posizione $N-1$.

```
int Full(List *l)
{
   return(l->last == N-1);
}
```

Cioè restituisce 1 se la lista è piena e non può contenere ulteriori elementi altrimenti restituisce 0.

3.1.5 Empty(l)

La funzione `Empty(l)` restituisce 1 se la lista è vuota.

```
int Empty(List *l)
{
  return(l->last == EMPTYLIST);
}
```

3.1.6 Insert(l,p,x)

La funzione `Insert(l,p,x)` inserisce l'elemento x in posizione p, spostando di una posizione più alta tutti gli elementi di posizione p e superiori. La funzione non è definita quando quando la lista è piena o p non esiste, ossia la posizione è negativa o è superiore alla posizione `End(l)`.

```
void Insert(List *l, int p, BaseType x)
{
  int i;

  if ( !Full(l) )
    {
      if ( (p >= 0) && (p <= l->last+1) )
        {
          for (i=l->last; i>=p; i--)
            l->info[i+1] = l->info[i];

          l->last++;
          l->info[p] = x;
        }
    }
}
```

3.1.7 Delete(l,p)

La funzione `Delete(l,p)` elimina l'elemento della lista di posizione p, spostando di una posizione in basso tutti gli elementi di posizione superiore a p. La funzione non è definita quando la lista è vuota o la posizione p non esiste (p esiste solo se è compreso tra 0 e $last$) o p coincide con `End(l)`.

```
void Delete(List *l, int p)
{
  int i;
```

```
  if ( !Empty(l) )
    {
      if ( (p >= 0) && (p <= l->last) )
        {
          for (i=p; i<l->last; i++)
            l->info[i] = l->info[i+1];

          l->last--;
        }
    }
}
```

3.1.8 Locate(l,x)

La funzione `Locate(l,x)` restituisce la posizione dell'elemento x. Se x compare nella lista più di una volta, allora viene restituita la posizione della prima occorrenza di x. Se x non compare nella lista o se la lista è vuota allora viene restituito la posizione `End(l)`.

```
int Locate(List *l, BaseType x)
{
  int i;

  if (! Empty(l) )
    for (i=0; i<=l->last; i++)
      if (l->info[i] == x)
        return(i);

  return(l->last+1);
}
```

3.1.9 Retrieve(l,p)

La funzione `Retrieve(l,p)` restituisce l'elemento di posizione p. La funzione non è definita quando la lista è vuota o p non esiste, ossia p è negativo o $p \geq$ `End(l)`.

```
BaseType Retrieve(List *l, int p)
{
  if ( !Empty(l) )
    if ( (p >= 0) && (p <= l->last) )
      return(l->info[p]);

  return(UNDEFINED);
```

```
}
```

3.1.10 Next(l,p)

La funzione `Next(l,p)` restituisce la posizione dell'elemento successivo all'elemento di posizione p. La funzione non è definita quando p non esiste o p coincide con `End(l)`.

```
int Next(List *l, int p)
{
  if ( !Empty(l) )
    if ( (p >= 0) && (p <= l.last) )
      {
        p++;
        return(p);
      }

  return(UNDEFINED);
}
```

3.2 Rappresentazione Collegata

Come detto precedentemente, la rappresentazione collegata della lista si basa sul principio di memorizzare gli elementi della lista in modo da non essere adiacenti, ossia da occupare posizioni in memoria RAM non contigue. La sequenzialità tipica della lista è mantenuta memorizzando accanto all'elemento generico x_i, la posizione del successivo elemento x_{i+1}.

La rappresentazione collegata può essere realizzata sia con vettori che con puntatori. Nel seguito verrà illustrata solo l'implementazione con puntatori. La rappresentazione collegata con puntatori si basa sull'idea di considerare elementi allocati dinamicamente in memoria RAM, ciascuno dei quali è rappresentato da due campi. Un campo memorizza il generico elemento della lista, mentre l'altro campo memorizza la posizione del successivo elemento, più precisamente, l'indirizzo della locazione di memoria a partire dalla quale si trove l'elemento successivo. Graficamente una lista collegata può essere rappresentata come in Figura 1.

La struttura dati in linguaggio C che implementa una lista collegata può essere definita come di seguito.

```
typedef struct Nodo
{
  BaseType     info;
  struct Nodo *next;
} *List;
```

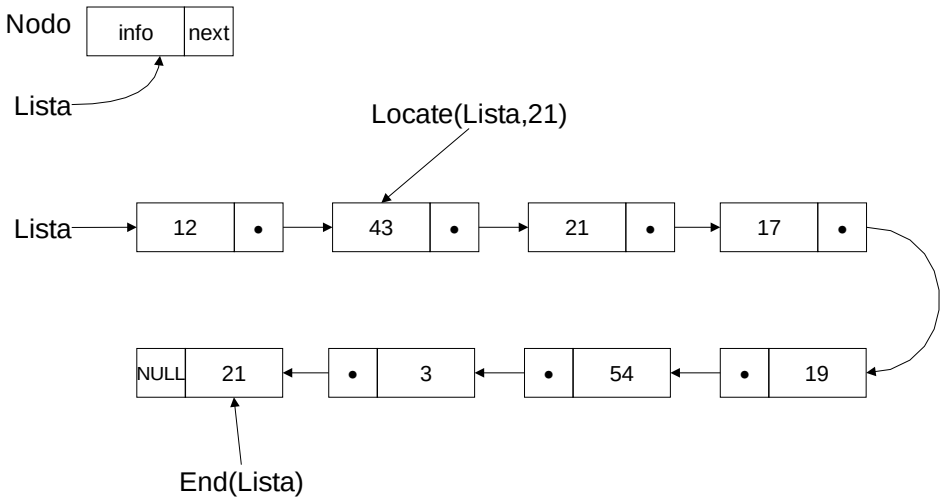


Figura 1: Una rappresentazione della lista collegata $l = \langle 12, 43, 21, 17, 19, 54, 3, 21 \rangle$.

Cioè il tipo `List` è un puntatore alla struttura `Nodo`.

L'unica costante da definire è EMPTYLIST che corrisponde ad una lista vuota. Se la lista è vuota il puntatore al primo elemento della lista è `NULL`. Dunque, la costante EMPTYLIST può essere rappresentata in C nella seguente maniera:

```
#define EMPTYLIST NULL
```

Nelle sottosezioni successive implementeremo in linguaggio C le funzioni descritte nella Sezione 2.

3.2.1 End(l)

Fornisce la posizione successiva alla posizione dell'ultimo elemento della lista. Per convenzione è pari al puntatore alla locazione di memoria contenente l'ultimo elemento della lista.

```
List End(List l)
{
  if (l == EMPTYLIST)
    return(EMPTYLIST);

  while (l->next != EMPTYLIST)
    l = l->next;

  return(l);
}
```

3.2.2 First(l)

Restituisce la posizione del primo elemento della lista. Per convenzione tale elemento è stato assunto pari a `NULL`. Se la lista è vuota restituisce `End(l)`.

```
List First(List l)
{
 return(EMPTYLIST);
}
```

3.2.3 MakeNull(l)

La funzione restituisce una lista vuota.

```
void MakeNull(List *l)
{
 *l = EMPTYLIST;
}
```

3.2.4 Full(l)

La funzione restituisce sempre 0, se si assume di avere sufficiente memoria RAM per la memorizzazione della lista.

```
int Full(List l)
{
 return(0);
}
```

3.2.5 Empty(l)

La funzione restituisce 1 se la lista è vuota.

```
int Empty(List l)
{
 return(l == EMPTYLIST);
}
```

3.2.6 Insert(l,p,x)

Questa funzione inserisce l'elemento x in posizione p, spostando di una posizione più alta tutti gli elementi di posizione p e superiori. La funzione `Insert(...)` distingue se l'inserimento deve essere fatto in testa (`p==NULL`) (Figura 2(a)) o in un'altra posizione (Figura 2(b)). Nel caso in cui l'inserimento è fatto in testa, allora il puntatore l iniziale è modificato.

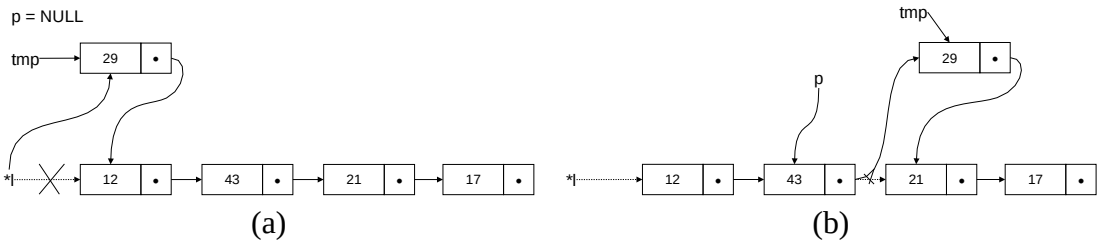


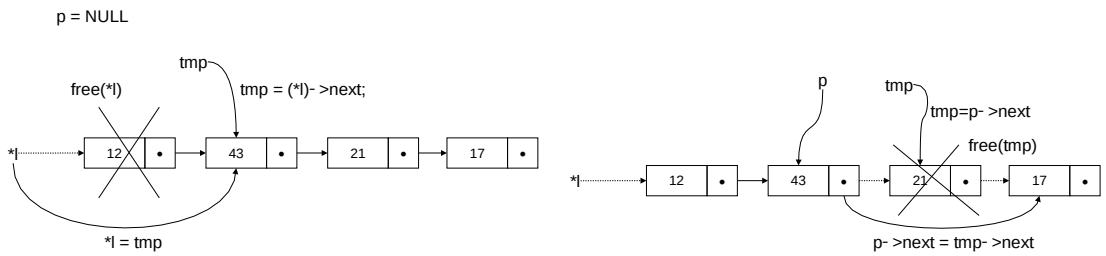
Figura 2: Inserimento in testa (a), in posizione generica (b).

```
void Insert(List *l, List p, BaseType x)
{
  List tmp;

  tmp = (List)malloc(sizeof(struct Nodo));

  tmp->info=x;

  if (p == EMPTYLIST)
    {
      tmp->next = *l;
      *l = tmp;
    }
  else
    {
      tmp->next = p->next;
      p->next=tmp;
    }
}
```

3.2.7 Delete(l,p)

Questa funzione elimina l'elemento della lista di posizione p, spostando di una posizione in basso tutti gli elementi di posizione superiore a p. La cancellazione viene differenziata in base al fatto che l'elemento da cancellare sia il primo (Figura 3(a)) o meno (Figura 3(b)).

```
void Delete(List *l, List p)
{
  List tmp;

  if ( !Empty(l) && p != End(l))
```



(a) (b)

Figura 3: Cancellazione in testa (a), in posizione generica (b).

```
    {
      if ( p == EMPTYLIST)
        {
          tmp = (*l)->next;
          free(*l);
          *l = tmp;
        }
      else
        {
          tmp = p->next;
          p->next = tmp->next;
          free(tmp);
        }
    }
}
```

3.2.8 Locate(l,x)

La funzione restituisce la posizione dell'elemento x. Se x compare nella lista più di una volta, allora viene restituita la posizione della prima occorrenza di x. Se x non compare nella lista allora viene restituito la posizione `End(l)`. Se l'elemento ricercato è il primo della lista, allora viene ritornato il valore `NULL`.

```
List Locate(List l, BaseType x)
{
  if (l->info == x)
    return(EMPTYLIST);

  while (l->next != EMPTYLIST)
    {
      if (l->next->info == x)
```

```
        return(l);
      l = l->next;
    }

  return(l);
}
```

3.2.9 Retrieve(l,p)

La funzione restituisce l'elemento di posizione p. Nel caso in cui `p == NULL`, viene restituito il primo elemento.

```
BaseType Retrieve(List l, List p)
{
  if ( (!Empty(l)) && (p != End(l)) )
    {
        if (p == EMPTYLIST)
          return(l->info);
        else
          return(p->next->info);
    }
}
```

3.2.10 Next(l,p)

La funzione restituisce la posizione dell'elemento successivo all'elemento di posizione p.

```
List Next(List l, List p)
{
 if (p != End(l))
   {
     if (p == EMPTYLIST)
       return(l);
     else
       return(p->next);
   }
}
```