La Struttura Dati Lista

Introduzione

Nella definizione di un programma mirato alla risoluzione di un determinato problema, la scelta delle strutture dati è di fondamentale importanza per un'efficiente risoluzione del problema, almeno tanto quanto la scelta dell'algoritmo risolutore. Molto spesso, anzi, l'algoritmo in grado di risolvere il problema stesso dipende strettamente dalle strutture dati adottate. Ciò implica che il tempo di calcolo necessario per l'esecuzione del programma che realizza l'algoritmo risolutore è strettamente legato alla struttura dati adottata.

La lista rappresenta una struttura dati flessibile ed efficiente per la gestione delle informazioni. In questo documento introdurremo il concetto della struttura dati Lista, la sua rappresentazione collegata e l'implementazione delle funzioni necessarie alla gestione di una lista in linguaggio C.

1 Definizione di Lista

La lista rappresenta una struttura dati tra le più utilizzate per la memorizzazione e la gestione di informazioni. Il vantaggio del suo utilizzo è la semplicità della gestione. Di contro le prestazioni delle operazioni necessarie a gestire tale struttura dati non sono eccelse. Se n è il numero di informazioni contenute, la complessità computazionale (ossia i tempi di esecuzione) degli algoritmi necessari alla gestione di una lista è direttamente proporzionale al numero di informazioni da gestire, cioè n. Esistono altre strutture dati, come ad esempio alberi, che possono offrire prestazioni superiori (ad esempio proporzionale a $\log n$), ma a costo di una più difficile gestione.

Particolari liste che trovano un utilizzo molto frequente sono le PILE e le CODE. Le prime permettono di realizzare politiche di gestione delle informazioni del tipo Last In First Out (LIFO); una tipica applicazione di tale politica è quella relativa allo stack in un computer. La struttura CODA realizza una politica di gestione di tipo First In First Out (FIFO), che trova innumerevoli applicazioni quando si vogliano realizzare algoritmi che processano informazioni in base, ad esempio, ad un ordine temporale della loro generazione.

Una Lista è una sequenza di elementi omogenei, ossia appartengono allo stesso tipo. Un Lista viene solitamente rappresentato come sequenza degli elementi che la compongono. Si chiama lunghezza della lista il numero **n** di elementi che la compongono. Una lista è individuata dall'elemento in prima posizione. Una Lista Vuota non presenta alcun elemento.

2 Realizzazione di una Lista in Linguaggio C

La realizzazione della struttura dati lista, consiste nella sua rappresentazione attraverso i tipi di dati disponibili nei linguaggi di programmazione, ed in particolare in C. Esistono tra le altre, due modalità con cui realizzare 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: un elemento della lista e l'indicazione della locazione di memoria contenente l'elemento successivo. In tal caso le locazioni di memoria non sono per forza contigue in RAM, ma possono occupare spazi di memoria sparpagliati nella porzione RAM (tipicamente Heap) riservata al programma.

Nel seguito verrà presentata la sola rappresentazione collegata.

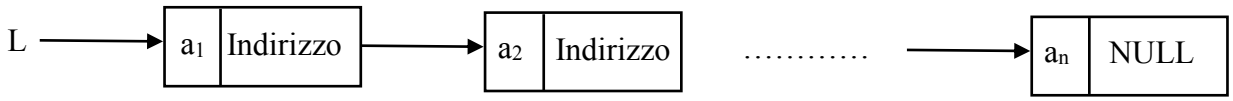
2.1.Rappresentazione Collegata

Come detto precedentemente, la rappresentazione collegata di una struttura dati 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 a_i, la posizione del successivo elemento a_{i+1}.

La rappresentazione collegata può essere realizzata molto bene tramite i 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.

Sia data la struttura dati lista composta dagli elementi: a_1, a_2, a_3, a_4, ……, a_n . Essa può essere rappresentata dalla seguente sequenza di elementi allocati in memoria RAM. L rappresenta l'indirizzo del primo elemento della lista.



Come si vede ogni elemento della lista ha due campi: l'elemento della lista e l'indirizzo (puntatore) all'elemento successivo della lista. L'ultimo elemento conterrà il valore NULL come indirizzo del successivo.

Nel seguito verrà mostrata la rappresentazione della Lista con puntatori in C.

Definiremo con ***tipobaseList*** il tipo di appartenenza di ciascun elemento della lista. Ad esempio se ciascun elemento fosse di tipo ***int***, l'implementazione del tipo ***tipobaseList*** risulterebbe:

```
typedef int tipobaseList;
```

Inoltre, la presenza di funzioni booleane ci suggerisce la possibilità di definire un tipo **boolean**, come alias del tipo short.

```
typedef short boolean;
```

Ciascun elemento della lista viene memorizzato in una locazione di memoria contente una struttura, composta da due campi: ***info*** che contiene il generico elemento della lista (di tipo **tipobaseList**), e il campo ***next***, che contiene l'indirizzo dell'elemento successivo. La lista, poi, è individuata dal puntatore alla locazione di memoria contenente il primo elemento della lista (il puntatore L nella figura precedente). Per quanto detto, l'implementazione di ciascun elemento della lista in C, è:

```
struct nodoList {
    tipobaseList info;
    struct nodoList *next;
};
```

Invece il tipo **list** è rappresentato dal puntatore all'elemento struct nodoList, visto che si è detto che la lista è individuata dal puntatore alla locazione di memoria contenente il primo elemento della lista (il puntatore L nella figura precedente). Dunque il tipo list è definito:

```
typedef struct nodoList * list;
```

3 Alcune operazioni sulla Struttura Dati Lista

Da ora in poi si consideri la seguente simbologia:

- sia ***l*** una lista,
- sia ***x*** un generico elemento dello stesso tipo degli elementi presenti nella lista. Il tipo di appartenenza verrà nel seguito indicato con ***TipobaseList***.

Come esempio, si consideri la seguente lista di numeri interi:

$$\boldsymbol{l}=\{12, 43, 21, 17, 19, 54, 3, 21\}$$

Le seguenti operazioni possono essere definite sulla struttura dati lista:

- ***ConfrontaList(tipobaseList, tipobaseList)***. Essa è definita come:

```
int ConfrontaList (tipobaseList a, tipobaseList b)
```

  Riceve in ingresso due parametri *tipobaseList* a e b e restituisce:

  - 0, se a e b sono identici
  - Un numero negativo se a è minore di b
  - Un numero positivo se a è maggiore di b

  Il motivo dell'uso di tale funzione è che nella gestione di una lista è necessario utilizzare espressioni che realizzano confronti logici (uguale, maggiore e minore) tra gli elementi della lista. Ad esempio in alcune operazioni si deve verificare se due elementi sono uguali tra loro. Come si sa, gli operatori ==, < e > non possono essere applicati a tutti i tipi di dati (ad esempio non si possono applicare agli struct). Dunque per avere delle funzioni che effettuano confronti logici e che valgano per qualunque *TipobaseList*, faremo uso della funzione *ConfrontaList* che implementeremo in modo diverso per ciascun *TipobaseList*. Nel caso in cui *TipobaseList* sia un intero, allora si potrebbe scrivere:

```
int ConfrontaList(tipobaseList a, tipobaseList b){
    return (a-b);
}
```

- ***VisualizzaElementoList(tipobaseList)***. Tale funzione permette di visualizzare ogni singolo elemento della lista. Essa dipende dal *TipobaseList*. Nel caso in cui esso fosse un intero, si potrebbe scivere:

```
void VisualizzaElementoList(tipobaseList x){
     printf("\n\nElemento = %d ", x);
}
```

- ***MakeNullList(list *)***: Restituisce una lista vuota. Attenzione, la funzione ***MakeNullList*** è intesa a inizializzare una lista, ponendo il suo valore a lista vuota. La funzione ***MakeNullList*** non è concepita per eliminare tutti gli elementi di una lista già esistente. Per far ciò si devono prima eliminare singolarmente tutti gli elementi della lista, con delle funzioni particolari che verranno spiegate successivamente. Una volta eliminati tutti gli elementi, si potrà procedere all'inizializzazione della lista tramite la funzione ***MakeNullList***.

```
void MakeNullList(list *l){
  *l=NULL;
}
```

- ***EmptyList(list)***: Restituisce il valore booleano vero se la lista ***l*** è vuota. Restituisce il valore falso altrimenti.

```
boolean EmptyList(list l){
  return(l==NULL);
}
```

- ***FullList(list)***:Restituisce il valore booleano vero se la lista è piena, ossia non è possibile inserire nuovi elementi. Restituisce il valore falso altrimenti. Per controllare che non sia più possibile inserire elementi, supporremo che allocare un nuovo elemento, che poi verrà subito dopo deallocato. Se la funzione di allocazione (malloc) fallisce, perché la memoria è satura, allora la funzione ***FullList*** torna il valore booleano vero. Se invece la funzione malloc riesce ad allocare, allora la funzione ***FullList*** tornerà il valore falso e l'elemento allocato per il test, verrà liberato con la funzione *free()*.

```
boolean FullList(list l){
  struct nodoList *temp;

  temp=(struct nodoList *)malloc(sizeof(struct nodoList));
  if(temp==NULL) return 1;
  free(temp);
  return 0;
}
```

- ***RetrieveList(list)***: Restituisce il ***tipobaseList*** contenuto nel primo elemento della lista l, se non è vuota.

```
tipobaseList RetrieveList(list l){
  if (!EmptyList(l)) return (l->info);
}
```

- ***InsertOrdList(list, tipobaseList)***: Inserisce l'elemento ***x*** nella lista l, in modo che x sia in ordine crescente. Se la lista è vuota, x viene inserito in testa, altrimenti la funzione scorre la lista fino a trovare la posizione giusta (in base all'ordinamento crescente) dell'elemento x. La funzione ritorna una lista ottenuta da quella iniziale inserendo l'elemento **x**.

```
list InsertOrdList(list l, tipobaseList x){
  struct nodoList * temp;
  list p;

  if (!FullList(l)) {
    temp=(struct nodoList *) malloc(sizeof(struct nodoList));
    temp->info=x;

    if (l==NULL) {
        temp->next=NULL;
        l=temp;
    } else {
        if (ConfrontaList(l->info,x)>0) {
           temp->next=l;
           l=temp;
        } else {
           p=l;
           while (p->next!=NULL) {
              if (ConfrontaList(p->next->info, x)<0) p=p->next;
              else break;
           }
           temp->next=p->next;
           p->next=temp;
       }
    }
  }
  return (l);
}
```

- ***DeleteList(list,tipobaseList)***: Elimina l'elemento x dalla lista, se lo trova e se la lista non è vuota. La funzione ritorna l'indirizzo della lista ottenuta da quella originaria, eliminando l'elemento x. Nel caso in cui la lista è vuota o l'elemento x non esiste, la funzione ritorna il valore NULL.

```
list DeleteList(list l, tipobaseList x){
  struct nodoList * temp;
  list p;

  if (!EmptyList(l))
      if (!ConfrontaList(l->info,x)) {
         temp=l->next;
         free (l);
         l=temp;
      } else {
         p=l;
         while (p->next!=NULL) {
                if (ConfrontaList(p->next->info, x)) p=p->next;
                else break;
         }
         if (p->next!=NULL){
                temp=p->next->next;
                free (p->next);
                p->next=temp;
         }
  }

  return (l);
}
```

- ***LocateList(list,tipobaseList)***: Ricerca un elemento **x** in una lista non vuota e ritorna l'indirizzo dell'elemento stesso o NULL se non lo trova o se la lista è vuota.

```
list LocateList(list l, tipobaseList x){

  if (!EmptyList(l))
          while (l!=NULL) {
                if (!ConfrontaList(l->info,x)) return(l);
                l=l->next;
          }

  return (NULL);
}
```

- ***VisitList(list)***: Visita la lista e visualizza sullo schermo ogni elemento della lista, se non è vuota.

```
void VisitList(list l){

  while (l!=NULL) {
    VisualizzaElementoList(l->info);
    l=l->next;
  }
}
```

3. Scrittura di un Programma di Gestione di Lista.

Il seguente programma si riferisce alla gestione delle funzioni principali di una struttura dati lista tramite un menu di comandi. Il programma presuppone che ogni informazione contenuta nella lista sia uno **struct tipobaseList** composto dai campi: cognome, nome, età.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

#define FFLUSH while(getchar()!='\n')

#define S 15

typedef short boolean;

typedef struct {
        char cognome[S], nome[S];
        unsigned short eta;
}tipobaseList;

void LeggiStringa(char s[],unsigned long dim){
  unsigned long i=0;

  for (i=0; i<dim-1;i++)
       if ((s[i]=getchar())=='\n') break;
  if (i==dim-1) FFLUSH;
  s[i]='\0';
}

int ConfrontaList(tipobaseList a, tipobaseList b){
   if (!strcmp(a.cognome,b.cognome))
                return strcmp(a.nome,b.nome);
   else return strcmp(a.cognome,b.cognome);
}

void VisualizzaElementoList(tipobaseList x){
     printf("\n\nElemento ");
     printf("\nCognome = %s ", x.cognome);
     printf("\nNome = %s ", x.nome);
     printf("\nEta' = %d \n\n",x.eta);
}

/*lista*/

struct nodoList {
   tipobaseList info;
   struct nodoList *next;
};

typedef struct nodoList * list;
```

```
void MakeNullList(list *l){
  *l=NULL;
}

boolean EmptyList(list l){
  return(l==NULL);
}

boolean FullList(list l){
  struct nodoList *temp;

  temp=(struct nodoList *)malloc(sizeof(struct nodoList));
  if(temp==NULL) return 1;
  free(temp);
  return 0;
}

tipobaseList RetrieveList(list l){
  if (!EmptyList(l)) return (l->info);
}

/*Inserimento Ordinato Crescente */
list InsertOrdList(list l, tipobaseList x){
  struct nodoList * temp;
  list p;

  if (!FullList(l)) {
    temp=(struct nodoList *) malloc(sizeof(struct nodoList));
    temp->info=x;

    if (l==NULL) {
        temp->next=NULL;
        l=temp;
    } else {
        if (ConfrontaList(l->info,x)>0) {
           temp->next=l;
           l=temp;
        } else {
           p=l;
           while (p->next!=NULL) {
              if (ConfrontaList(p->next->info, x)<0) p=p->next;
              else break;
           }
          temp->next=p->next;
          p->next=temp;
       }
    }
  }
  return (l);
}
```

```
/*Cancella un elemento dalla lista, se lo trova */
list DeleteList(list l, tipobaseList x){
  struct nodoList * temp;
  list p;

  if (!EmptyList(l))
      if (!ConfrontaList(l->info,x)) {
         temp=l->next;
         free (l);
         l=temp;
      } else {
         p=l;
         while (p->next!=NULL) {
                if (ConfrontaList(p->next->info, x)) p=p->next;
                else break;
         }
         if (p->next!=NULL){
                temp=p->next->next;
                free (p->next);
                p->next=temp;
         }
  }
  return (l);
}


/*Ricerca un elemento e ritorna l'indirizzo dell'elemento stesso
o NULL se non lo trova */
list LocateList(list l, tipobaseList x){

  if (!EmptyList(l))
      while (l!=NULL) {
            if (!ConfrontaList(l->info,x)) return(l);
            l=l->next;
      }

  return (NULL);
}

void VisitList(list l){

  while (l!=NULL) {
    VisualizzaElementoList(l->info);
    l=l->next;
  }
}

list lt, pos;
short s;
tipobaseList elem;
```

```
int main(void)
{

  MakeNullList(&lt);

  do {
      printf("\n\nMenu di Operazioni ");
      printf("\n1-Inserimento Ordinato ");
      printf("\n2-Ricerca ");
      printf("\n3-Cancellazione ");
      printf("\n4-Visita Lista ");
      printf("\n5-Fine ");
      printf("\nInserisci la scelta ---> ");
      scanf("%d",&s);
      FFLUSH;
      switch(s) {

          case 1 :
                  if (FullList(lt)) printf("\nLista piena ");
                  else {
                        printf("\nInserisci l'Elemento ");
                        printf("\nInserisci il Cognome ");
                        LeggiStringa(elem.cognome, S);
                        printf("\nInserisci il Nome ");
                        LeggiStringa(elem.nome, S);
                        printf("\nInserisci l'eta' ");
                        scanf("%hu",&elem.eta);
                        FFLUSH;
                        pos=LocateList(lt,elem);
                        if (pos!=NULL){
                                printf("\nElemento Trovato ");
                                printf("\nInserimento Impossibile ");
                        }else {
                                lt=InsertOrdList(lt,elem);
                                printf("\nElemento Inserito ");
                        }
                  }
                  break;

              case 2 :
                 if (EmptyList(lt)) printf("\nLista Vuota ");
                 else {
                        printf("\nRicerca per Cognome e Nome ");
                        printf("\nInserisci il Cognome ");
                        LeggiStringa(elem.cognome, S);
                        printf("\nInserisci il Nome ");
                        LeggiStringa(elem.nome, S);
                        pos=LocateList(lt,elem);
                        if (pos!=NULL){
                                 printf("\nElemento Trovato ");
                                 elem=RetrieveList(pos);
                                 VisualizzaElementoList(elem);
                        }else printf("\nElemento non Trovato ");
                  }
```

```
                break;

        case 3 :
                if (EmptyList(lt)) printf("\nLista Vuota ");
                else {
                    printf("\nRicerca per Cognome e Nome ");
                      printf("\nInserisci il Cognome ");
                      LeggiStringa(elem.cognome, S);
                      printf("\nInserisci il Nome ");
                      LeggiStringa(elem.nome, S);

                      pos=LocateList(lt,elem);
                        if (pos!=NULL) {
                                lt=DeleteList(lt,elem);
                                printf("\nElemento Eliminato ");
                        }
                      else printf("\nElemento non Trovato ");
                }
                break;

          case 4 :
                  if (EmptyList(lt)) printf("\nLista Vuota ");
                  else VisitList(lt);

        }
      } while (s<5);
}
```