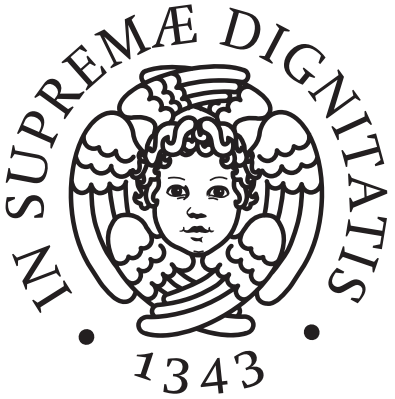

Universitá di Pisa - Dipartimento di Matematica

Funzioni su Alberi e Liste in C

Simmaco Di Lillo
dsimmaco@gmail.com

Capitolo 1

Ringraziamenti

Per la parte riguardanti le prove pratiche degli anni passati, ringrazio Giuseppe De Pasquale.

Indice

Capitolo 2

Da copiare in ogni codice

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

struct El{
  int info;
  struct El *next;
};

typedef struct El ElementoDiLista;
typedef ElementoDiLista* ListaDiElementi;
```

Capitolo 3

Aggiungere

3.1 Aggiungere in testa

```
void AggiungiTesta(ListaDiElementi *lis, int val ){
        ListaDiElementi aux;
         aux=malloc(sizeof(ElementoDiLista));
        aux->info=val;
        aux->next=*lis;
        *lis=aux;
}
```

3.2 Aggiungere in coda

```
void AggiungiCoda(ListaDiElementi *lis, int val){
  if (*lis==NULL)
    {
      ListaDiElementi aux;
      aux=malloc(sizeof(ElementoDiLista));
      aux->info=val;
      aux->next=NULL;
      *lis=aux;
    }
  else
    AggiungiCoda(&((*lis)->next),val);
}
```

3.3 Aggiungere in ordine crescente

Serve la funzione Aggiungi Testa

```
void swap(ListaDiElementi *lis, int val)
{
    if((*lis==NULL) || ((*lis)->info>val) )
    AggiungiTesta(lis, val);
  else
    swap(&((*lis)->next),val);
}
```

3.4 Aggiungere -1 quando si verifica una condizione

```
void Modifica(ListaDiElementi *lis){
  if (*lis!=NULL){
    if (condizione){
      ListaDiElementi aux;
      aux=malloc(sizeof(ElementoDiLista));
      aux->info=-1;
      aux->next=*lis;
      *lis=aux;
      Modifica(&((*lis)->next->next));}
    else
      Modifica(&((*lis)->next));
  }}
```

Capitolo 4

Eliminare

4.1 Eliminare la testa

```
void EliminaTesta(ListaDiElementi *lis){
  if (*lis!=NULL)
    {
      ListaDiElementi el=*lis;
        *lis=(*lis)->next;
      free(el);
    }
}
```

4.2 Eliminare la coda

```
void EliminaCoda(ListaDiElementi *lis){
  if (*lis!=NULL)
    {
      if ((*lis)->next==NULL)
        *lis=NULL;
      else
        EliminaCoda(&((*lis)->next));
    }
}
```

4.3 Eliminare la prima occorrenza

```
 void EliminaPrimaOcc(ListaDiElementi *lis, int val){
  if(*lis!=NULL)
    {
      if((*lis)->info==val)
        {
          ListaDiElementi el=*lis;
          *lis=(*lis)->next;
          free(el);
        }
      else
        EliminaPrimaOcc(&((*lis)->next),val);
    }
}
```

4.4 Eliminare tutte le occorrenze

```
void EliminaOcc(ListaDiElementi *lis, int val)
{
  if(*lis!=NULL)
    {
      if ((*lis)->info==val)
        {
          ListaDiElementi el=*lis;
          *lis=(*lis)->next;
          free(el);
          EliminaOcc(lis,val);  //2 istruzioni prima siamo andati avanti
        }
      else
        EliminaOcc(&((*lis)->next),val);
    }
}
```

4.5 Eliminare l'ultima occorrenza

Serve la funzione per eliminare la testa e la coda.

```
void EliminaUltima(ListaDiElementi *lis, int val)
{
  if (*lis!=NULL)
    {
      ListaDiElementi prec=NULL, occ=*lis,ultimo;
// ultimo  salva il prec di quello da eliminare (se ultimo=NULL va eliminata
      int trovato=0;
      while( occ->next!=NULL)
        {
          if (occ->info==val)
            {
              trovato=1;
              ultimo=prec;
            }
          prec=occ;
          occ=occ->next;
        }
      if(occ->info==val)
        EliminaCoda(lis);
      else
        {
          if (trovato)
            {
              if(ultimo==NULL)
                EliminaTesta(lis);
              else
                {
                  ListaDiElementi el=ultimo->next;
                  ultimo->next=(ultimo->next)->next;
                  free(el);
                }
            }
        }
    }
}
```

Capitolo 5

Altre funzioni

5.1 Stampare

```
void PrintList(ListaDiElementi lis)
{
  if (lis==NULL)
    printf("NULL\n");
  else
    {
      printf("%d -> ", lis->info);
      PrintList(lis->next);
    }
}
```

5.2 Costruire una lista invertita

Serve la funzione per agginugere in testa

```
ListaDiElementi ListaInversa(ListaDiElementi lis){
  ListaDiElementi ris=NULL;
  if(lis!=NULL)
    {
      ListaDiElementi occ=lis;
      while(occ!=NULL)
        {
        AggiungiTesta(&ris,occ->info);
        occ=occ->next;
        } }
  return ris;}
```

5.3 Verificare la presenza di un valore

Da aggiungere questa linea di codice

```
  typedef enum{false, true} boolean;


boolean Trova(ListaDiElementi lis, int val ){
    boolean ris=false;
```

```
    if(lis!=NULL)
    {
        if (lis->info==val)
            ris=true;
        else
            ris=Trova(lis->next,val);

    }
    return ris;
}
```

5.4 Contare le occorrenze di un valore

```
int ContaOcc(ListaDiElementi lis,int val)
{
  int ris=0;
  if(lis!=NULL)
    {
      if (lis->info==val)
        ris=1+ContaOcc(lis->next, val);
      else
        ris=ContaOcc(lis->next,val);
    }
  return ris;
}
```

5.5 Contare il numero di elementi

```
int Lunghezza(ListaDiElementi lis){
  int ris=0;
  if (lis!=NULL)
    ris=1+Lunghezza(lis->next);
  return ris;
}
```

Parte I

Alberi

Capitolo 6

Da copiare in ogni programma

```
#include<stdio.h>
#include<stdlib.h>
struct nodoAlberoBinario{
  int label;
  struct nodoAlberoBinario *left;
  struct nodoAlberoBinario *right;
};

typedef struct nodoAlberoBinario NodoAlbero;

typedef NodoAlbero *AlberoBinario;
```

Capitolo 7

Visite

7.1 Simmetrica

La visita procede nel seguente ordine

1. Ramo di sinistra
2. Radice
3. Ramo di destra

Se si visita un albero di ricerca in questo modo si ottiene una lista di elementi in ordine crescente

```
void VisitaSimmetrica(AlberoBinario bt){
  if (bt !=NULL){
    VisitaSimmetrica(bt->left);
    printf("%d->", bt->label);
    // Modifica( &(bt->label));
    // modifica ha come prototipo Modifica (int * x)
    VisitaSimmetrica(bt->right);
  }
}
```

7.2 Anticipata

La visita procede nel seguente ordine

1. Radice
2. Ramo di sinistra
3. Ramo di destra

```
 void VisitaAnticipata(AlberoBinario bt){
if (bt!=NULL)
  {
    printf("%d", bt->label);
    // qui si modifica qualcosa
    VisitaAnticipata(bt->left);
    VisitaAnticipata(bt->right);
  }}
```

7.3 Posticipata

La visita procede nel seguente ordine

1. Ramo di destra
2. Ramo di sinistra
3. Radice

```
 void VisitaPosticipata(AlberoBinario bt){
if (bt!=NULL){
  VisitaPosticipata(bt->right);
  VisitaPosticipata(bt->left);
  printf("%d", bt->label);
  // qui si modifica
}}
```

Capitolo 8

Trovare elementi

8.1 Controllare la presenza di un elemento

Da aggiungere questa linea di codice

```
  typedef enum{false, true} boolean;

 boolean Controlla( AlberoBinario bt, int val){
    boolean ris=false;
    if(bt!=NULL)
        ris=( (bt->label== val) || Controlla( bt->left ,val) || Controlla(bt->

    return ris;
}
```

8.2 Trovare profonditá di un elemento

Nel caso che l'elemento non si trovi nell'albero la funzione restituisce -1

```
 int Profondita(AlberoBinario bt , int val)
{    int ris=-1;
     if(trova(bt,val))
     {    ris=0;
          if (bt->label==val)
               ris=0;
          else
          {
               if ((bt->label)>val)
                    ris=1+prof(bt->left ,val);
               else
                    ris=1+prof(bt->right ,val);
           }
     }
     return ris;
}
```

8.3 Contare le occorrenze di un elemento

```
int ContaOcc(AlberoBinario bt, int val){
  int ris=0;
  if (bt !=NULL){
    if ((bt->label)==val)
      ris++;
    ris=ris+ContaOcc(bt->left ,val)+ContaOcc(bt->right ,val);
  }
  return ris;
}
```

Capitolo 9

Altre funzioni

9.1 Aggiungere elementi ad un albero di ricerca

```
    void Aggiungi( AlberoBinario *bt, int val){
     if((*bt)==NULL){
          AlberoBinario aux;
          aux=malloc(sizeof(NodoAlbero));
          aux->label=val;
          aux->left=NULL;
          aux->right=NULL;
          *bt=aux;}
     else
     {
          if (((*bt)->label)>val)
               Aggiungi(&((*bt)->left),val);
          else
               Aggiungi(&((*bt)->right),val);
     }
}
```

9.2 Calcolare l'altezza di un albero

Per la funzione principale occorre la seguente funzione che restituisce il massimo tra 2 numeri

```
   int max (int a,int b)
{
    int ris=a;
    if(a<b)
        ris=b;
    return ris;
}
```

La funzione principale

```
int Altezza(AlberoBinario bt)
{
    int ris;
    if (bt==NULL)
        ris=0;
    else
        ris=max(Altezza(bt->left),Altezza(bt->right))+1;
    return ris;
}
```

Parte II

Funzioni presenti in vecchie prove pratiche o esercitazioni

Capitolo 10

Funzioni Gennaio 2018

10.1 ReadList

readList: Legge dallo standard input una sequenza di numeri interi ordinati in maniera strettamente crescente e termina automaticamente l'acquisizione alla prima occorrenza di un numero che non rispetta l'ordinamento (l'intero che viola l'ordinamento non va inserito nella lista). Gli interi devono essere memorizzati, nell'ordine di acquisizione, in una lista concatenata opportunamente allocata. La funzione readList restituisce un intero pari al numero di elementi presenti nella lista creata. Per esempio, supponendo che venga acquisita la sequenza (5,8,15,9) la lista dovrebbe essere la seguente (e sarebbe la stessa se si acquisisse la sequenza (5,8,15,15)), e la funzione restituirebbe il valore 3:

```
int readList(ListaDiElementi *lista)
{
        ListaDiElementi li=malloc(sizeof(ElementoLista));
        int a;
        int b;
        int c=1;
        scanf("%d", &a);
        AggiungiCoda(lista ,a);
        b=a;
        scanf("%d", &a);
        while(a>b)
                {
                c=c+1;
                b=a;
                AggiungiCoda(lista ,a);
                scanf("%d", &a);
                }
        return c;
}
```

10.2 filterLists

filterList: date due liste list1 e list2 di lunghezza qualsiasi ordinate e che non contengono duplicati, la funzione elimina dalla lista list1 tutti gli elementi che sono presenti anche nella lista list2.

```
void filterLists(ListaDiElementi *lista1, ListaDiElementi lista2)
```

```
{
        if((*lista1)!=NULL&&lista2!=NULL)
                {
                if(((*lista1)->info)>(lista2->info))
                        {
                filterLists(&((*lista1)->next),lista2);
                        }
                else
                        {
                        if(((*lista1)->info)>(lista2->info))
                                {
                        filterLists(lista1,(lista2->next));
                                }
                        else
                                {
                        ListaDiElementi aux=(*lista1);
                        (*lista1)=((*lista1)->next);
                        filterLists(lista1,(lista2->next));
                                }
                        }
                }
}
```

Capitolo 11

Funzioni Febbraio 2018

11.1 ReadList

ReadList: legge una sequenza di numeri interi, finch 'e vale la condizione

che questa alterni numeri pari e numeri dispari (cominciando indifferentemente per uno o l'altro), e termina l'acquisizione quando legge il primo elemento che viola tale alternanza. I numeri letti devono essere memorizzati in una lista nell'ordine inverso a quello di acquisizione (dunque con inserimento in testa): l'ultimo numero letto, che fa terminare l'acquisizione, non va inserito nella lista. Per esempio, supponendo che la sequenza immessa sia (5,0,15,10,12),al termine di ReadList avremmo la lista (10,15,0,5). Il puntatore alla testa della lista creata deve essere restituito come valore di ritorno della funzione.

```
ListaDiElementi readList()
{
        ListaDiElementi occ=malloc(sizeof(ElementoLista));
        int a;
        int b;
        scanf("%d", &a);
        (occ->info)=a;
        (occ->next)=NULL;
        b=a;
        scanf("%d", &a);
        while((b+a)%2==1)
                {
                b=a;
                ListaDiElementi lis=malloc(sizeof(ElementoLista));
                (lis->info)=a;
                (lis->next)=occ;
                occ=lis;
                scanf("%d", &a);
                }
        return occ;
}
```

11.2 OddEven

OddEven: Questa funzione prende una lista di interi e restituisce 1 se essa contiene lo stesso numero di elementi pari ed elementi dispari. Restituisce 0 altrimenti. Per esempio, per la lista

(10,15,0,5),OddEven restituirá 1. Per la lista(1,8,3),OddEven restituirá 0.

```
int OddEven(ListaDiElementi lista)
{
        int a=0;
        if(lista!=NULL)
                {
                a=1;
                while(lista->next!=NULL)
                        {
                        lista=lista->next;
                        a=a+1;
                        }
                }
        a=(a+1)%2;
        return a;
}
```

11.3 Equality

Equality: data una lista di interi, Equality deve cancellare dalla coda della lista il minimo numero (che pu'o anche essere = 0) di elementi affinché la lista risultante contenga tanti numeri pari quanti numeri dispari. Per esempio, se la lista fosse (1,8,3). Equality restituirebbe la lista (1,8). Il puntatore alla testa della lista creata deve essere restituito come valore di ritorno della funzione.

```
void filterLists(ListaDiElementi *lista1, ListaDiElementi lista2)
{
        if((*lista1)!=NULL&&lista2!=NULL)
                {
                if(((*lista1)->info)>(lista2->info))
                        {
                        filterLists(&((*lista1)->next),lista2);
                        }
                else
                        {
                        if(((*lista1)->info)>(lista2->info))
                                {
                                filterLists(lista1,(lista2->next));
                                }
                        else
                                {
                                ListaDiElementi aux=(*lista1);
                                (*lista1)=((*lista1)->next);
                                filterLists(lista1,(lista2->next));
                                }
                        }
                }
}
```

Capitolo 12

Esercitazione 2018

12.1 removemin

removeMin: data una lista list di lunghezza maggiore o uguale a uno e con possibili elementi duplicati, la funzione elimina dalla lista l'elemento minimo. Nel caso in cui ci siano elementi minimi ripetuti, la funzione elimina quello piú a destra (ossia l'ultimo incontrato navigando la lista a partire da list).

```
int removeMin(ListaDiElementi* lista)
{
        int a;
        int b;
        ListaDiElementi prec=NULL;
        ListaDiElementi occ=*lista;
        ListaDiElementi min=NULL;
        if(occ!=NULL)
                {
                a=((*lista)->info);
                if((*lista)->next!=NULL)
                        {
                        prec=occ;
                        occ=occ->next;
                        }
                while((occ->next)!=NULL)
                        {
                        b=occ->info;
                        if(b<=a)
                                {
                                a=b;
                                min=prec;
                                }
                        prec=occ;
                        occ=occ->next;
                        }
                b=occ->info;
                if(b<=a)
                        {
                        while(((*lista)->next)!=NULL)
                                {
```

```
                        lista=(&(*lista)->next);
                        }
                (*lista)=NULL;
                a=b;
                }
        else
                {
                if(min==NULL)
                        {
                        ListaDiElementi el=*lista;
                        *lista=((*lista)->next);
                        free(el);
                        }
                else
                        {
                        ListaDiElementi el=(min->next);
                        (min->next)=(min->next)->next;
                        free(el);
                        }
                }
        return a;
        }
}
```

12.2 pushback

pushBack: data una lista list di lunghezza maggiore o uguale a zero e un intero x, inserisce x in fondo alla lista.

```
void pushBack(ListaDiElementi* lista, int x)
{
        if(*lista==NULL)
                {
                ListaDiElementi aux;
                aux=malloc(sizeof(ElementoLista));
                aux->info=x;
                aux->next=NULL;
                *lista=aux;
                }
        else
                {
                pushBack(&((*lista)->next),x);
                }
}
```

Capitolo 13

Gennaio 2019

13.1 read

read: La funzione read legge dallo standard input una sequenza di numeri interi ordinati in maniera strettamente crescente e termina automaticamente l'acquisizione alla prima occorrenza di un numero che non rispetta l'ordinamento (l'intero che viola l'ordinamento non va inserito nella lista). Gli interi devono essere memorizzati, nell'ordine di acquisizione, in una lista concatenata opportunamente allocata (Nota: Il puntatore l passato come argomento alla procedura— deve essere impostato in modo che punti alla lista creata). La funzione read restituisce un intero pari al numero di elementi presenti nella lista creata.

```
int read(List *l)
{
        int a;
        int b;
        scanf("%d", &a);
        if((*l)!=NULL)
                {
                if(((*l)->info)<a)
                        {
                        List aux;
                        aux=malloc(sizeof(Element));
                        aux->info=a;
                        aux->next=NULL;
                        *l=aux;
                        b=1+read(&aux);
                        ((*l)->next)=aux;
                        return b;
                        }
                else
                        {
                        *l=NULL;
                        return 0;
                        }
                }
        else
                {
                List aux;
                aux=malloc(sizeof(Element));
```

```
                aux->info=a;
                aux->next=NULL;
                *l=aux;
                b=1+read(&aux);
                ((*l)->next)=aux;
                return b;
                }
}
```

13.2 intersection

intersection: Date due liste l1 e l2 di lunghezza qualsiasi, i cui elementi sono in ordine strettamente crescente e che non contengono duplicati, la procedura intersection deve modificare la lista l1 in modo che mantenga solo gli elementi comuni a entrambe le liste (ovvero eliminando tutti gli elementi della lista l1 che non sono presenti nella lista l2).

```
void  intersection(List *l1, List l2)
{
if((*l1)!=NULL&&l2!=NULL)
        {
        if((*l1)->next!=NULL&&(l2->next)!=NULL)
                {
                if((l2->info)<(*l1)->info)
                        {
                        intersection(l1,(l2->next));
                        }
                else
                        {
                        if((l2->info)>(*l1)->info)
                                {
                                *l1=((*l1)->next);
                                intersection(l1,l2);
                                }
                        else
                                {
                                intersection(&((*l1)->next),l2);
                                }
                        }
                }
        else
                {
                if((l2->next)==NULL)
                        {
                        if((*l1)!=NULL&&((*l1)->info)==(l2->info))
                                {
                                ((*l1)->next)=NULL;
                                }
                        else
                                {
                                if((*l1)!=NULL)
```

```
                                                {
                                                *l1=NULL;
                                                }
                                        }
                                }
                        }
                }
        }
}
```

Capitolo 14

Gennaio 2019 primo appello

14.1 insert

La procedura insert deve prendere in ingresso un albero binario di ricerca e un valore intero val. Se val non 'e contenuto nell'albero, insert deve creare un nuovo nodo con val come valore, inserirlo nell'albero (mantenendo l'ordine dell'albero binario di ricerca). Altrimenti, insert non deve effettuare alcun inserimento nell'albero ricevuto in ingresso.

```
        void insert( AlberoBinario *radice, int val ){
  if((*radice)==NULL){
    AlberoBinario aux;
    aux=malloc(sizeof(NodoAlbero));
    aux->label=val;
    aux->left=NULL;
    aux->right=NULL;
    *radice=aux;}
  else
    {
      if (((*radice)->label)>val)
        insert(&((*radice)->left),val);
      if (((*radice)->label)<val)
        insert(&((*radice)->right),val);
    }
}
```

14.2 kth

La funzione kth deve prendere in input la radice di un albero binario di ricerca ed un valore k. La funzione kth deve restituire il k-esimo valore tra quelli memorizzati nell'albero di ricerca la cui radice 'e radice. Si assuma che il numero n di interi memorizzati nell'albero sia non inferiore a due e che k $\leq$ n.
Attenzione serve la funzione ContaNodi

```
int ContaNodi(AlberoBinario radice)
{
    int ris=0,dx,sx;
    if ( radice!=NULL )
    {
```

```
        return 1+ContaNodi(radice->left)+ ContaNodi(radice->right) ;
    }
    return ris;
}

int kth( AlberoBinario radice, int k){
    int ris;
    int n=ContaNodi(radice->left);
    if ( n==k-1)
       ris=radice->label;
    else
      {
          if ( n>k-1 )       ris=kth(radice->left,k);

          else               ris=kth(radice->right, k-(n+1));

      }
    return ris;

}
```