Funzioni di Programmazione I
Liste, code e alberi binari

10 settembre 2018

Indice

Capitolo 1

Liste e nodi

1.1 Struttura nodo

```
struct nodo{
        int info;
        nodo* next;
        nodo(int a=0, nodo*b=0){info=a; next=b;}
}
```

1.2 Creazione ricorsiva di una lista

1.2.1 Con sentinella

```
nodo* leggi() {
    int n;
    cin >> n;
    if(n==-1) return 0;
    else return new nodo(n, leggi());
}
```

1.2.2 Data la dimensione

```
nodo* crea(int dim){
   if(dim){
        int x;
        cin>>x;
        return new nodo(x,crea(dim-1));
   }
   return 0;
}
```

1.3 Ricerca di un valore in una lista (ritorna l'indice (intero) del nodo)

```
int ricerca(nodo* L, int x) {
    if(L==0) return -1;
    if(L->info==x) return 0;
    if(ricerca(L->next, x)==-1) return -1;
    else return 1+ricerca(L->next, x);
```

```
}
```

1.4 Inserimento di un nodo nel posto giusto (ordine crescente)

```
nodo* inserisci(nodo* L, int x) {
    if (L==0) return new nodo(x,0);
    if(L->info>x) return new nodo(x, L);
    else return new nodo(L->info, inserisci(L->next,x));
}
```

1.5 Stampa di una lista

1.5.1 Stampa semplice

```
void stampa(nodo* L) {
    if(L) {
        cout << L->info << " ";
        stampa(L->next);
    }
}
```

1.5.2 Stampa elaborata

```
void stampa_lista(nodo* L) {
    if(L==0) cout << "Lista vuota" << endl;
    else {
        cout <<L->info;
        if(L->next==0) cout << endl;
        else {
            cout << "->";
            stampa_lista(L->next);
        }
    }
}
```

1.6 Pattern matching

1.6.1 Funzione che ritorna true se c'è match

```
bool testa(nodo*& y, int* P, int dimP, nodo*& m){
if(!dimP) { //caso base: pattern vuoto => match vero
    m=y;
          y=0;
    return true;
}
if(!y) return false; //secondo caso base: lista vuota
if(y->info==*P) return testa(y->next, P+1, dimP-1, m); //primo match: ricorsione
return false; //altrimenti niente match
}
```

1.6.2 Return match e resto della lista per riferimento

```
//per riferimento il resto della lista e con il return la lista matchata
nodo* match(nodo* &L, int* P, int dimP) {
if(!L) return 0;
nodo* r=L, *m; //r e' il puntatore all'inizio (ricorsivo) della lista
if(testa(r, P, dimP, m)) { //se ha trovato il match...
// (dobbiamo costruire la lista restante)
    L=m; //"salta" il match e lo mette (riferimento) in L
    return r; //restituisce la lista del match
}
else return match(L->next, P, dimP);
}
```

1.6.3 Per riferimento il matche e il resto col return

```
//per riferimento la lista matchata e il resto col return
nodo* match(nodo* &L, int* P, int dimP) {
if(!L) return 0;
nodo *r=L, *m;
if(testa(r, P, dimP, m)) {
    L=r;
    return m;
}
else {
    L=L->next;
    r->next=match(L, P, dimP);
    return r;
}
}
```

1.7 Concatenazione

1.7.1 Concatenazione iterativa

```
    if(!L1) return L2;
    if(!L2) return L1;
    nodo* x=L1;
    while(x->next)
        x=x->next;
    x->next=L2;
    return L1;
}
```

Capitolo 2

Code e FIFO

2.1 Libreria che gestisce le code

2.1.1 -code.h

```
#include<iostream>
struct nodo {
    char chiave;
    nodo *next;
    nodo(char c='\0', nodo* n=NULL);
};
struct coda {
    nodo *inizio;
    nodo *fine;
    coda(nodo* i=NULL, nodo* f=NULL);
};

void push(char c, coda &Q);
char pop(coda &Q);
bool e_vuota(coda Q);
```

2.1.2 -code.cpp

```
#include<iostream>
#include "code.h"
nodo::nodo(char c, nodo* n) {chiave=c; next=n;}
coda::coda(nodo* i, nodo* f) {inizio=i; fine=f;}
void push(char c, coda &Q) { //push:aggiunge un nuovo elemento alla fine
// della coda
    if(!Q.inizio) {
        Q.inizio=new nodo(c, 0);
        Q.fine=Q.inizio;
    }
    else {
        Q.fine->next=new nodo(c,0);
        Q.fine=Q.fine->next;
    }
}
void pushList(nodo*L,coda & Q) { //pushList: inserisce una nuova lista alla
// fine della coda
    if(L) {
```

```
        nodo* t=L->next;
        push(L,Q);
        pusLlist(t, Q);
    }
}

char pop(coda &Q) { // pop: elimina il primo elemento della coda e lo ritorna
    char key=(Q.inizio)->chiave;
    nodo* del=Q.inizio;
    Q.inizio=(Q.inizio)->next;
    if(!Q.inizio) Q.fine=NULL;
    delete del;
    return key;
}
bool eVuota(coda Q) { // eVuota : ritorna true sse la coda e' vuota
    return(!Q.inizio);
}
```

2.2 Strutture di FIFO e nodi FIFO

```
    nodo* primo, *ultimo;
    int dim;
    FIFO(nodo*a=0,nodo*b=0,int c=0){
        primo=a; ultimo=b; dim=c;}
};

struct nodoFIFO{ //per creare liste di FIFO
    FIFO info;
    nodoFIFO* next;
    nodoFIFO(FIFO a=FIFO(), nodoFIFO*b=0){
        info=a; next=b;}
};
```

2.3 Inserimento di un nodo all'inizio della FIFO

```
    if(!a.primo){
        a.primo=a.ultimo=b;
        b->next=0; return a;
    }
    else{
        b->next=a.primo;
        a.primo=b;
        return a;
        }
}
```

2.4 Inserimento di un nodo alla fine della FIFO

```
        b->next=0;
        if(!a.primo)
        a.primo=a.ultimo=b;
    else{
        a.ultimo->next=b;
        a.ultimo=b;
        }
```

```
8         return a;
9   }
```

2.5 Concatenazione di due FIFO

```
1         a.ultimo->next = b.primo;
2         a.ultimo = b.ultimo;
3         return a;
4   }
```

Capitolo 3

Alberi binari

3.1 Costruzione di alberi

3.1.1 Struttura albero

```
struct nodo{
int info;
nodo* left,*right;
nodo(int a=0, nodo* b=0, nodo*c=0){info=a; left=b;right=c;}
};
```

3.2 Funzione buildtree per costruire l'albero

```
nodo* buildtree(nodo* r, int n) {
if(!n) return r;
int num;
cin>>num;
r = bilanciato(r,num);
return buildtree(r,n-1);
}
```

3.3 Funzione bilanciato per bilanciare l'albero(necessario)

```
nodo* bilanciato(nodo* r, int k) {
if(!r) return new nodo(k,0,0);
if(contanodi(r->left)<=contanodi(r->right)) r->left=bilanciato(r->left, k);
else r->right=bilanciato(r->right, k);
}
```

3.4 Funzione contanodi per contare il numero di nodi dell'albero

```
int contanodi(nodo* R) {
if(!R) return 0;
return 1+contanodi(R->left)+contanodi(R->right);
}
```

3.5 Stampa di un albero

3.5.1 Lineare

```
void stampa_l(nodo* r) {
//ricorsivo, stampa sottoalbero sx e sottoalbero dx
if(r){
    cout<<r->info<<'(';
    stampa(r->left);
    cout<<',';
    stampa(r->right);
    cout<<')';
}
else cout << '_';
}
```

3.5.2 Infissa

```
void infix(nodo *x) {
if(x) {
infix(x->left); //stampa albero sinistro
cout<<x->info<<' '; //stampa nodo
infix(x->right); //stampa albero destro
}
}
```

3.5.3 Prefissa a salti

```
int stampa_a_salti(nodo* r, int k, int n) { //stampa un nodo ogni k attraversati
if(!r) return n;
int salti;
if(n==1) {
    cout << r->info << " ";
    n=k;
}
else n--;
salti=stampa_a_salti(r->left, k, n);
return stampa_a_salti(r->right, k, salti);
}
```

3.6 Cercare cammino in un albero

```
bool cerca_cam(nodo* r, int k, int y, int* C) {
//ritorna true se c' e' e in tal caso si trova in C
if(r->info==y) k=k-1;
if(k<0) return false;
if(!r->left && !r->right) {
    if(k==0) {
        *C=-1;
        return true;
    }
    return false;
}
if(r->left) {
    if(cerca_cam(r->left, k, y, C+1)) {
        *C=0;
```

```
15          return true;
16      }
17  }
18  if(r->right) {
19      if(cerca_cam(r->right, k, y, C+1)) {
20          *C=1;
21          return true;
22      }
23  }
24  return false;
25  }
```

3.7 Restituire una lista ordinata

che consiste di un numero di nodi pari a quelli di un albero e i cui campi info sono gli stessi dell' albero

```
1  nodo* buildList(nodoA* r) {
2  if(!r) return NULL;
3  return fuse(new nodo(r->info), fuse(buildList(r->left),buildList(r->right)));
4  }
```

3.7.1 Restituire la lista ordinata costruita disponendo in ordine i nodi L1 e L2

(necessaria alla funzione sopra)

```
1  nodo* fuse(nodo* L1, nodo* L2) {
2  coda Q=coda();
3  while(L1||L2) {
4      if(L1 && L2) {
5          nodo* t;
6          if(L1->info<=L2->info) {
7              t=L1->next;
8              push(L1, Q);
9              L1=t;
10         } else {
11             t=L2->next;
12             push(L2,Q);
13             L2=t;
14         }
15     } else if(L1) {
16         push_list(L1,Q);
17         L1=NULL;
18     } else {
19         push_list(L2,Q);
20         L2=NULL;
21     }
22 }
23 return Q.inizio;
24 }
```

3.7.2 Inserire un valore in un albero in modo che sia ordinato

(e conseguente buildtree)

```
nodoA* insert(nodoA*r, char y) {
if(!r) return new nodoA(y);

if(conta_n(r->left)<=conta_n(r->right))
r->left=insert(r->left,y);
else
r->right=insert(r->right,y);
return r;
}
nodoA* buildtree(nodoA*r, int dim) {
if(dim)
{
char z;
cin>>z;
nodoA*x=insert(r,z);
return buildtree(x,dim-1);
}
return r;
}
```

3.8 Restituisce la lista concatenata i cui nodi puntano ai nodi dell'albero ordinati secondo l'ordine infisso

tali che il primo nodo nella lista punti al nodo n-esimo dell'albero secondo l'ordine infisso, e i successivi nodi puntano ad un nodo dell'albero ogni k nodi sempre secondo l'ordine infisso

```
nodo* B(nodoA* r, int k, int &n) {
if(!r) return 0;
nodo* sx, *dx, *c;
sx=B(r->left, k, n);
if(n==1) {
    c=new nodo(r, NULL);
    n=k;
    sx=fuse(sx, c);
} else n=n-1;
dx=B(r->right, k ,n);
return fuse(sx, dx);
}
```

3.9 Contare le foglie di un albero

```
int ContaFoglie(nodo* n)
{
if(!n) return(0);
else {
    if ((n->left==NULL) && (n->right==NULL)) return(1);
    else return ContaFoglie(n->left) + ContaFoglie(n->right);
}
}
```

3.10 Sapere se un albero è perfettamente bilanciato

```
bool PerfBil(nodo* n)
{
if (!n) return(true);
else {
if ((n->left==NULL) && (n->right==NULL)) // Se il nodo e' una foglia
  return(true);
else {
  if ((n->left!=NULL) && (n->right!=NULL))
    // Paraticamente: se il nodo ha tutti e due i figli..
    return( PerfBil(n->left) && PerfBil(n->right) );
  else return(false);
}
}
}
```

3.11 Trovare e restituire un nodo con campo info = y

```
nodo* trova(nodo* x, char y) {
if(!x) return 0; //se e' vuoto fallisce la ricerca
if(x->info==y) return x; //se e' quello restituisce il nodo
nodo* z=trova(x->left, y); //sottoalbero sx
if(z==0) return z;
return trova(x->right);
}
```

3.12 Trovare l'altezza di un albero

```
int altezza(nodo*x) {
if(!x->left && !x->right) return 0;
int a=-1, b=-1;
if(x->left) a=altezza(x->left);
if(x->right) b=altezza(x->right);
if(a>b) return a+1;
else return b+1;
}
```

3.13 Seguire un percorso dato in un albero

```
nodo* trova(nodo *x, int *C, int lung) {
if(!x) return 0; //fallito
if(lung==0) return x; //trovato
if(*C==0) return trova(x->left, C+1, lung-1); //C=0 vuol dire sx
else return trova(x->right, C+1, lung-1);
}
```

3.14 Da un albero, ricavare una lista dei nodi con campo info = y ricorsivamente

```
punt* conc(punt*a,punt*b){
if(!a) return b;
a->next=conc(a->next,b);
```

```
4   return a;
5   }
6   punt* buildList(nodo*r,int y){ //albero(r), y e' un intero uguale ad un campo info
7   if(!r) return 0;
8   punt*a=buildList(r->left,1);
9   punt*b=buildList(r->right,1);
10  if(r->info==y)
11      b=new punt(r,b);
12  return conc(a,b);
13  }
```