ricorsione versus iterazione

ricorsione su una lista

andata

lista

ritorno

passiamo su un nodo 2 volte

la ricorsione è terminale se al ritorno non si fa nulla

ricorsione terminale sulle liste: stampa

```
void stampa_ric(nodo * L)
{ if(L)
        {
        cout<<L→info<<‘ ‘;
        stampa(L→next);
        }
}
```

è facile da simulare con while

stampa iterativa di una lista:

```
void stampa(nodo *L)
{ while(L)
        {cout<<L→info<<‘ ‘; L=L→next; }

   cout<<endl;
}
```

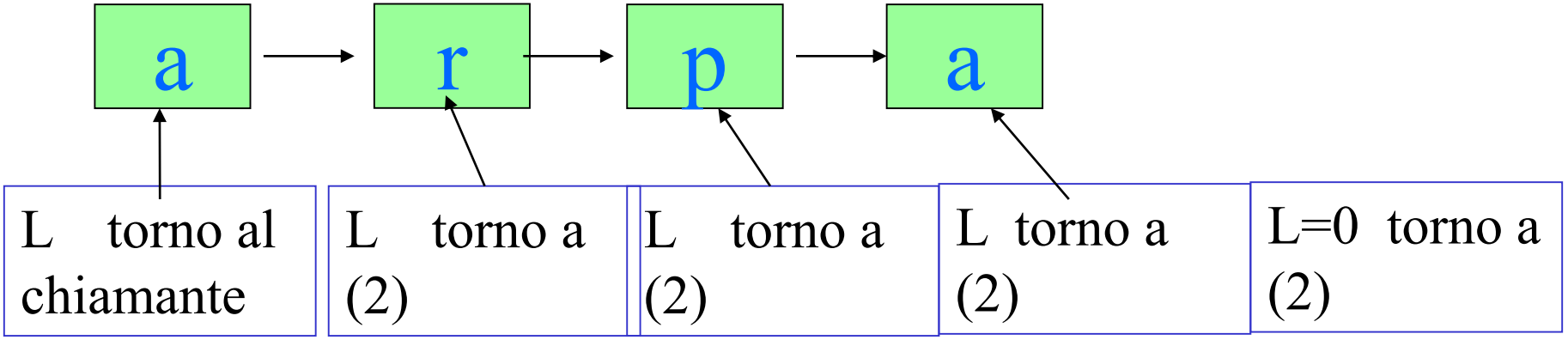
e la stampa a rovescio ?

la soluzione ricorsiva è facile !!

```
void stampa_ric(nodo * L)
{ if(L)
 {stampa(L→next);          ←(1)
  cout<<L→info<<‘ ‘;       ←(2)
  }
}
```

2 visite a L, 2 fasi

e quella iterativa ??



ogni invocazione ritorna al punto (2) della precedente, cioè alla stampa del campo info del nodo gestito da questa

l'operazione da fare al ritorno è sempre la stessa (stampa) quello che conta è cosa stampare (L->info)

il while simula l'attraversamento della lista, ma non ci da il ritorno indietro !!!

e in questo esempio ci serve,

dobbiamo essere capaci di considerare i nodi della lista 2 volte:

(1) per andare avanti

(2) per stampare il campo info

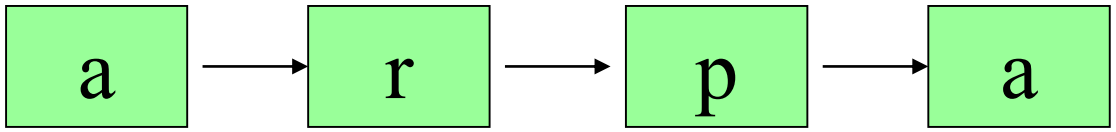
come fa la soluzione ricorsiva

facciamo così:

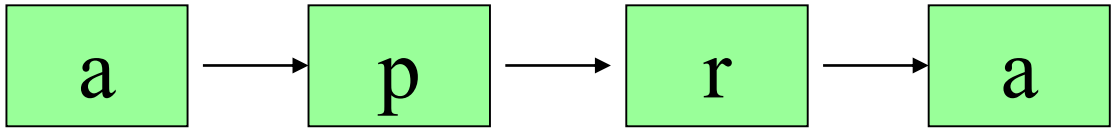
con un primo while costruiamo una copia rovesciata della lista da stampare

con un secondo while la stampiamo

input



con 1 while



col secondo while stampiamo

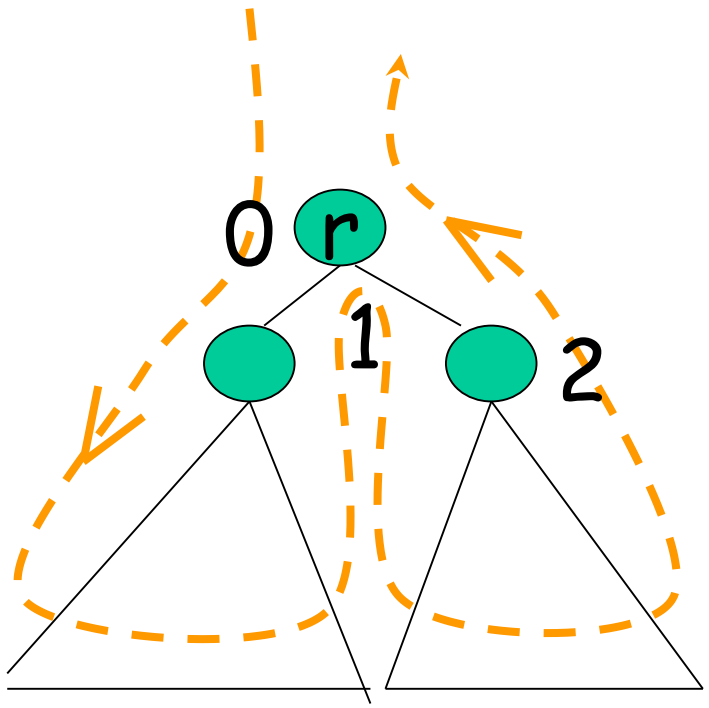
```
void stampa_iter(nodo *L)
{
    nodo* X=0;
    while(L)
        {X=new nodo(L->info,X); L=L->next;}

    while(X)
        {cout<< X->info << ' '; X=X->next;}
     cout<<endl;
}
```

e con gli alberi ?

stampa infissa di un albero:

```
void stampa(nodo *r)
{if(r)                                  ←(0)  test + left
        {
        stampa(r→left);
        cout<<r→info<<‘ ‘;              ←(1)  stampa + right
        stampa(r→right);
        }
}                                       ←(2) return
```

3 fasi con operazioni diverse da fare


0
r
1
2

```
struct elem{int fase; nodo *N;

elem(int a=0, nodo*b=0){fase=a; N=b;}};

fase = 0 / 1 / 2
```

rappresentiamo le 3 fasi:

0) elem(0, r)

stampa(r->left);

1) elem(1, r)

cout<<r->info;

stampa(r->right);

2) elem(2, r)

simuliamo la pila della ricorsione con una lista di

```
struct nodoM{elem info; nodoM*next;};
```

```
void stampa_iter(nodo *r)
{nodoM*X=new nodoM(elem(0,r));
 while(X)
{    switch(X->info.fase)  {
                    case 0:
                    { if (!X->info.N)
                              {nodoM* z=X; X=X->next;  delete z; }
                      else
                              {
                               X->info.fase++;
                               X=new nodoM(elem(0,X->info.N->left),X);
                              }
                       break;
                    }

                    case 1:……case 2: // ESERCIZIO
 } }
```

```
case 1:
{
cout<<X->info.N->info<<‘ ‘;
X->info.fase++;
X=new nodoM(elem(0, X->info.N->right),X);
break;
}
case 2:
{
   nodoM*z=X; X=X->next; delete z;

}
```

pattern match (non contiguo) iterativo sui cammini di un albero

usiamo:

struct elem{int fase, pm; nodo*N;
elem(int a = 0, int b = 0, nodo*c = 0){fase=a;
pm=b; N=c;}};

il campo pm serve per ricordare l'indice del prossimo elemento di P da matchare

quando pm=nP il match è completato

```
S[top]=el(0,0,R);
 top=1;
 bool success=false;
 while(top && !success)
  {
            switch(S[top-1].fase)
          {
          case 0:
           {
            if(S[top-1].pm==nP)
              success=true;
            else
              if(!S[top-1].N)
                      top--;
              else
                      {
                       if(S[top-1].N->info == P[S[top-1].pm])
                           (S[top-1].pm)++;
                       S[top-1].fase=1;
                       S[top]=el(0,S[top-1].pm,S[top-1].A->left);
                       top++;
                      }
             break;
           }
```

```
                case 1:
                  {
                    S[top-1].fase=2;
                    S[top]=el(0,S[top-1].pm,S[top-1].A->right);
                    top++;
                    break;
                  }
                case 2:
                  {
                    top--;
                  }
                }
    }
  if(success)
    cout<< "match c'è"<<endl;
  else
    cout<<"match non c'è"<<endl;
}
```