memoria dinamica

e

liste

finora: in A[10][5] limiti sono costanti

ma in C++ esistono array con limiti variabili:

```
int a, b;
int A[a][b];
cout<< sizeof(A);  //??
int a, b;
cin>>a>>b;
int A[a][b];
cout<< sizeof(A);  //ok, ma non si possono più
                                cambiare, né deallocare
```

meglio usare la memoria dinamica che vedremo adesso

il C++ permette di chiedere da programma al sistema operativo l'allocazione di memoria da usare nel programma:

memoria per ospitare valori di qualsiasi tipo, intero, double, enum, struct, array …

new è la funzione che fa la richiesta, essa restituisce il puntatore alla memoria allocata:

int *p=new int;

alloca spazio Ram per un intero, p punta al primo byte dei 4 allocati, ma non sono nella pila

la memoria richiesta con la new viene allocata sullo HEAP che è diverso dalla pila

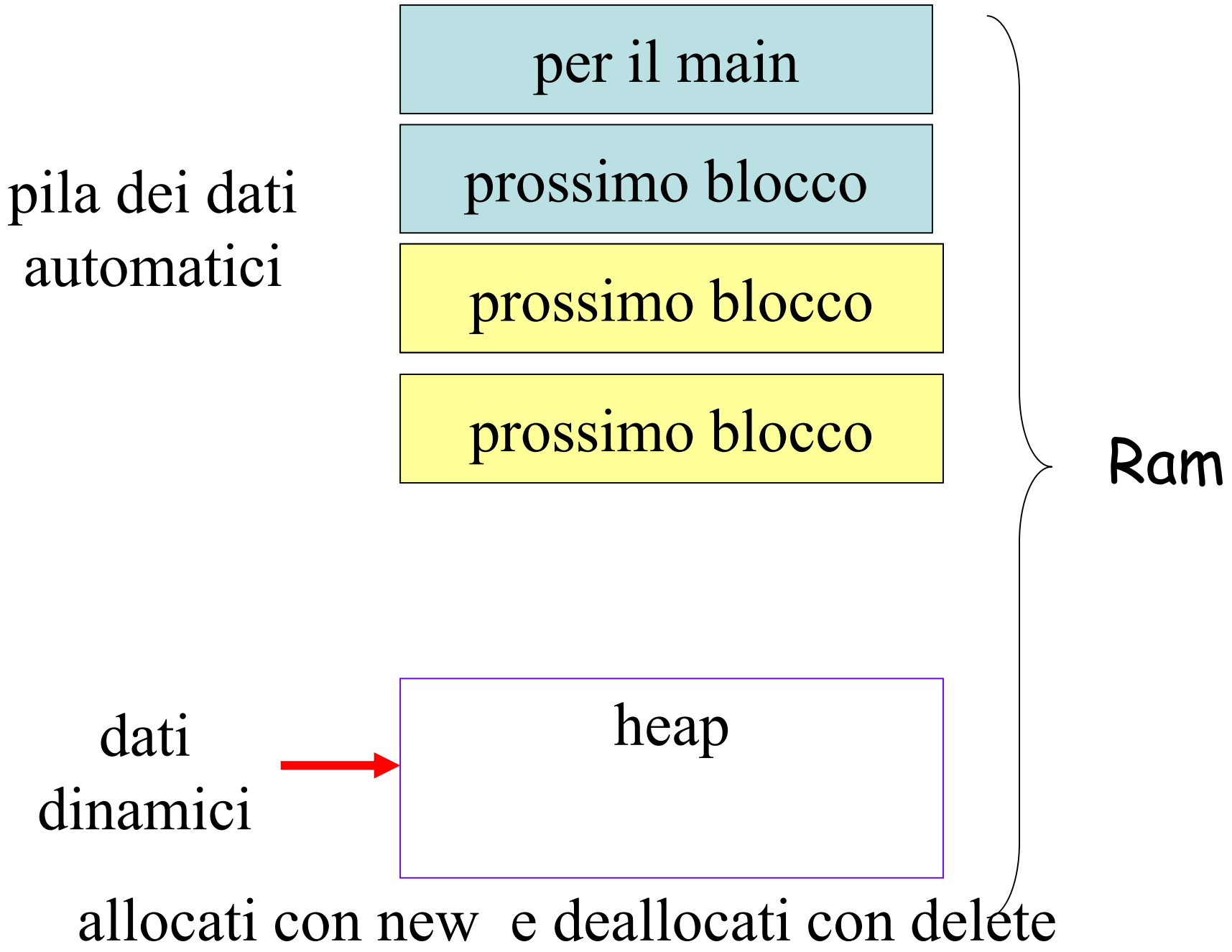
la deallocazione può essere fatta esplicitamente da programma con la funzione delete:

delete p;

dove p punta all'oggetto da deallocare

se il programma non fa la delete, lo spazio resta occupato fino alla fine del programma → ERRORE

memory leak


per il main
prossimo blocco
prossimo blocco
prossimo blocco
pila dei dati automatici
Ram
heap
dati dinamici
allocati con new e deallocati con delete

allocazione e deallocazione di array;

```
int * p= new int[10];

delete[] p;
```

anche a più dimensioni:

```
int (*p)[10]=new int [5][10];

delete[] p;

int (*p)[8][10]=new int[5][8][10];

delete[] p;
```

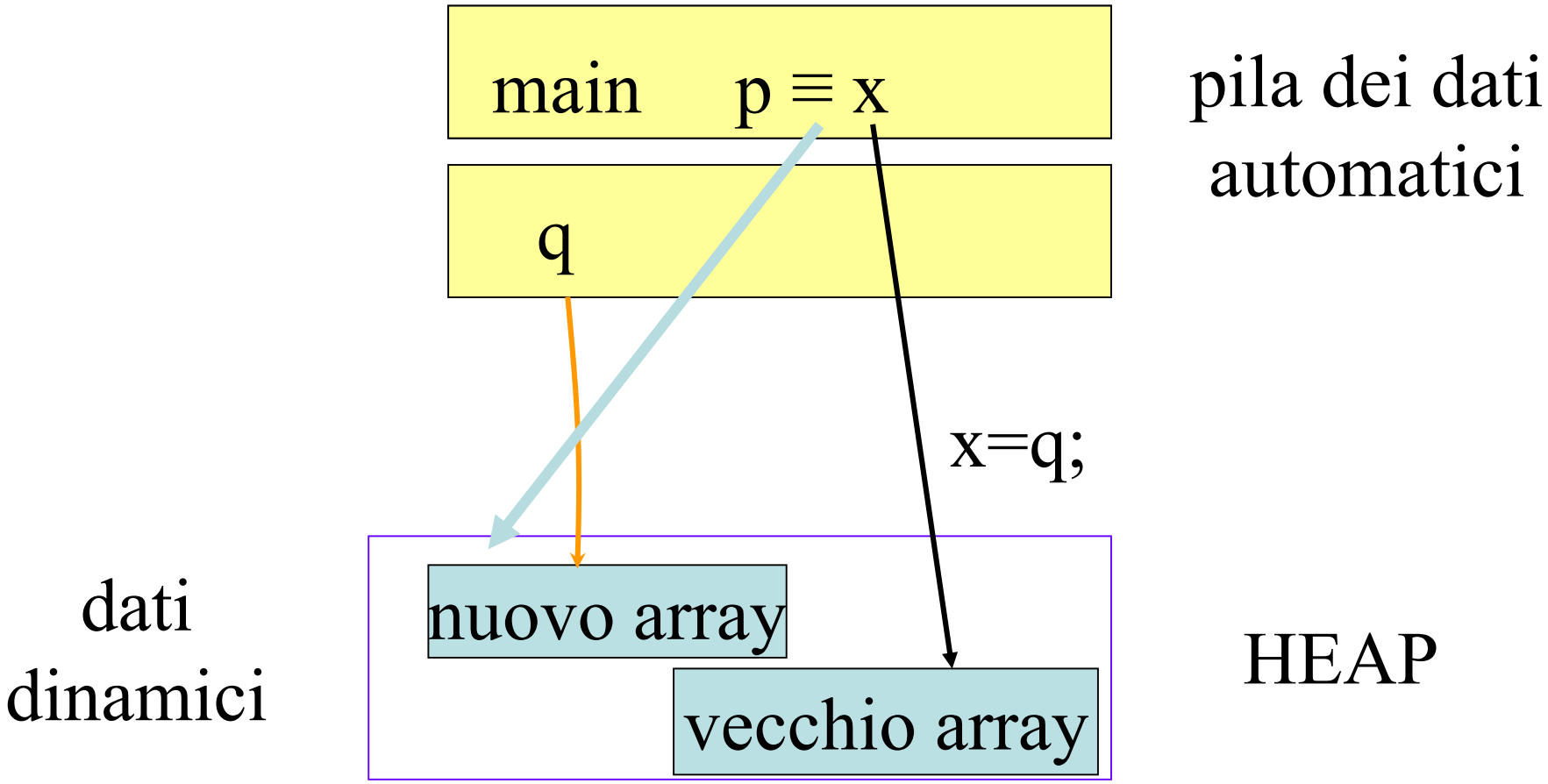
possiamo adattare dinamicamente gli array al bisogno in ogni momento dell'esecuzione del programma

supponiamo di avere un programma che legge da cin in un array fino a che trova la sentinella -1 e che deve potersi adattare a quanti interi vengono inseriti

```
main()                  si parte con 10 posizioni
{int dim=10, *p=new int[dim], i=0;
bool sentinella=false;
while(!sentinella)
        {if(i==dim)  allunga(p,dim);
        cin>>p[i];

        if(p[i]==-1)
                sentinella=true;
        else
                i++;
        } // i valori letti, senza -1

}  // p ha dim posizioni
```

```
void allunga(int *& x, int & dim)

{int * q= new int[dim*2]; //creo nuovo array
for(int i=0; i<dim; i++)
q[i]=x[i];     // ricopio il vecchio nel nuovo
delete [] x;  // elimino il vecchio
x=q;    // x punta al nuovo array
dim=dim*2; // dim è nuova dimensione
}
```

notare: q non va deallocata è variabile locale di allunga e quindi viene deallocata automaticamente (sta sulla pila)


main
p ≡ x
q
x=q;
nuovo array
vecchio array
pila dei dati automatici
dati dinamici
HEAP

con new e delete possiamo costruire strutture dati capaci di cambiare dinamicamente a seconda del bisogno

liste e alberi

le liste possono allungarsi e accorciarsi e gli alberi possono crescere o venire potati

una lista è una struttura dati ricorsiva, infatti è definita ricorsivamente con

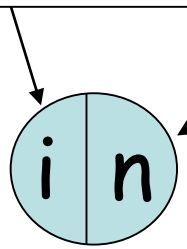
caso base: una lista vuota

caso ricorsivo: un elemento seguito da una lista di elementi (detta il resto della lista)

realizzazione di liste concatenate in C++

ogni nodo della lista ha 2 campi:

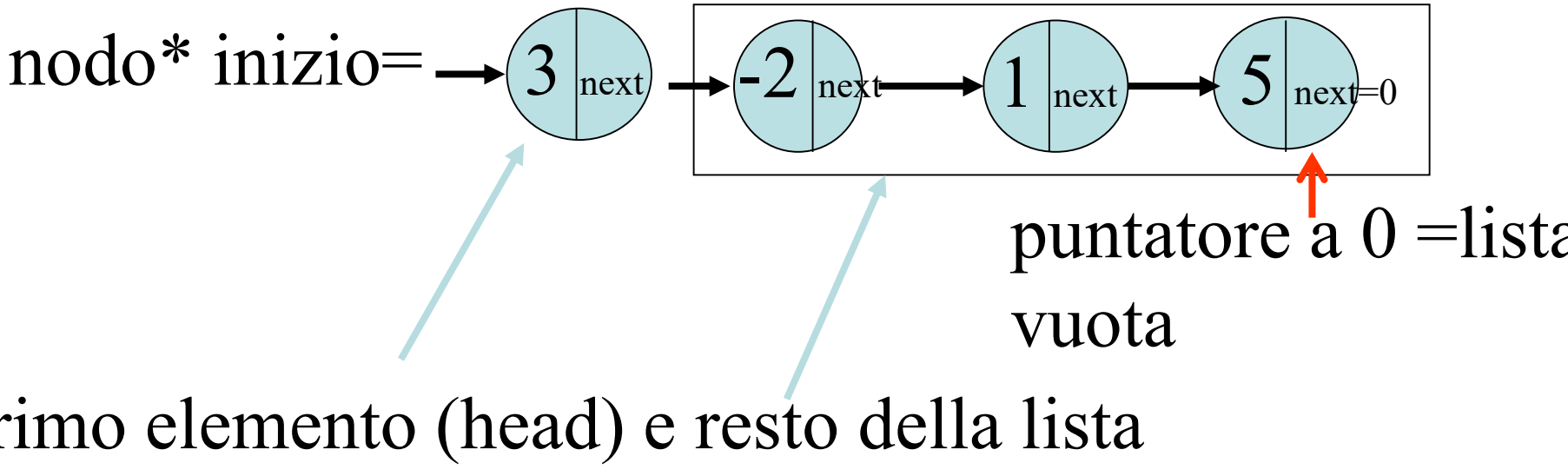
struct nodo {int info; nodo* next;};

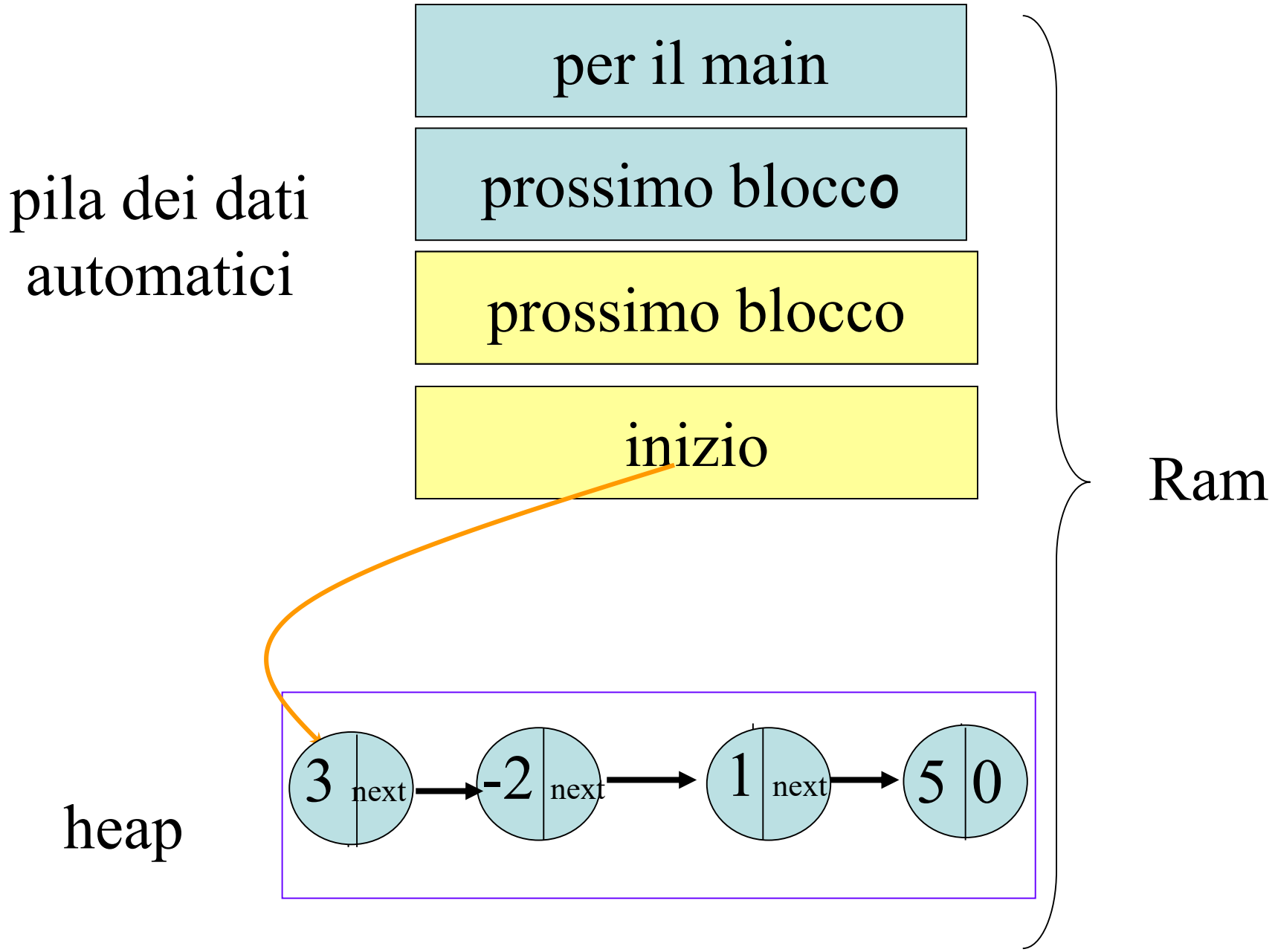


lista vuota:

nodo * inizio = 0;

lista con 4 nodi:




per il main
prossimo blocco
prossimo blocco
inizio
pila dei dati automatici
Ram
heap
3
next
-2
next
1
next
5
0

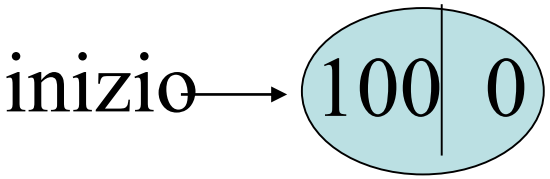
allocando i nodi dinamicamente con new ed eliminandoli con delete possiamo avere liste che crescono (si aggiungono nodi) e diminuiscono (si eliminano nodi) dinamicamente

lista vuota: nodo * inizio=NULL; //const=0

inizio=new nodo;

(*inizio).info=100;

(*inizio).next=0;

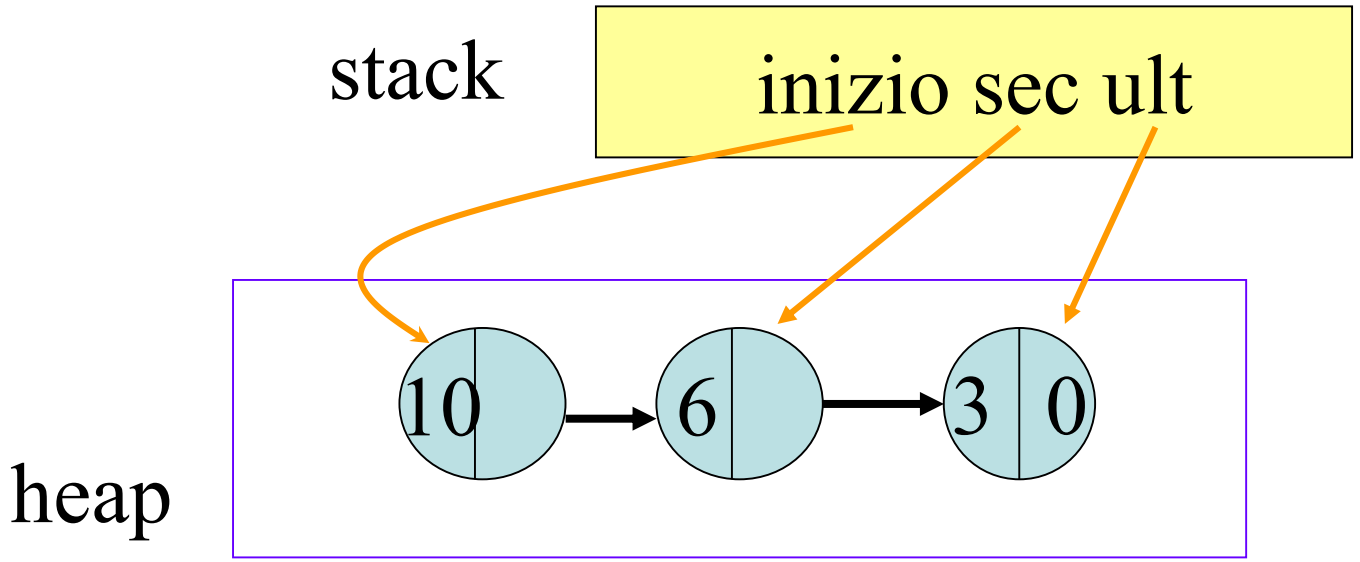


disponibili 2 notazioni :

(*inizio).info=100; inizio→info=100;

(*inizio).next=0; inizio→next=0;

```
struct nodo{int info; nodo* next;

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

nodo * ult=new nodo(3,0);

nodo* sec=new nodo(6,ult);
nodo * inizio=new nodo(10,sec);
```



una lista è un valore di tipo nodo*

una lista si dice corretta quando:

-è 0 (o NULL)

-punta a un nodo il cui campo next è una lista corretta

in pratica, è una sequenza possibilmente vuota di nodi, in cui ciascun nodo ha campo next che punta al prossimo fino all'ultimo che ha next=0

stampa di liste concatenate

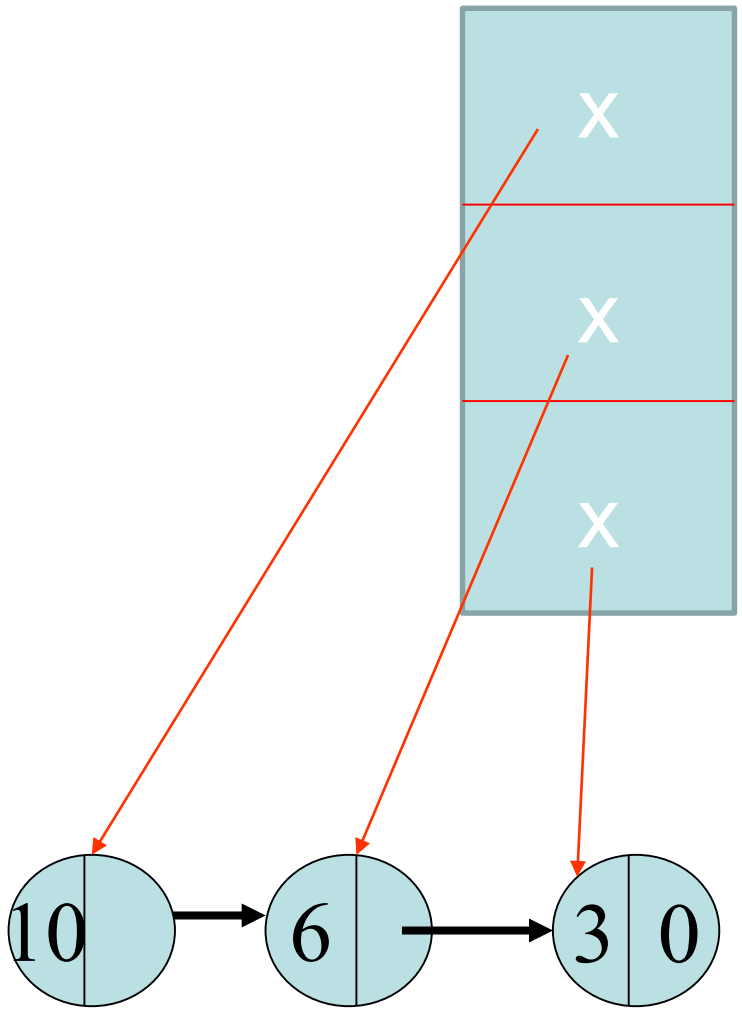
-stampa dal primo all'ultimo

-in ordine inverso: dall'ultimo al primo

le facciamo ricorsivamente

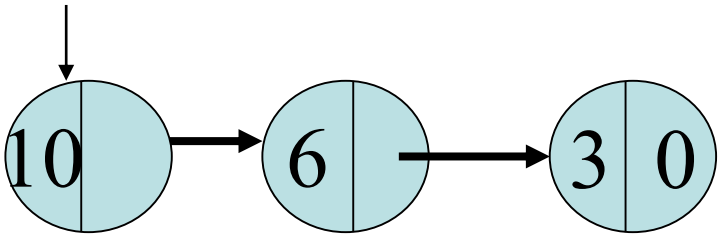
```
void stampa(nodo *x)
{
 if(x)
        {
          cout<< x->info;
          stampa(x->next);
        }
}
```

ogni invocazione ha una x
che punta ad un nodo
ma non serve!!
è ricorsione terminale



si può fare facilmente anche col while

```
nodo *x=inizio;
while(x!=0){
                cout<< x->info<<endl;
                 x=x->next;
              }
```

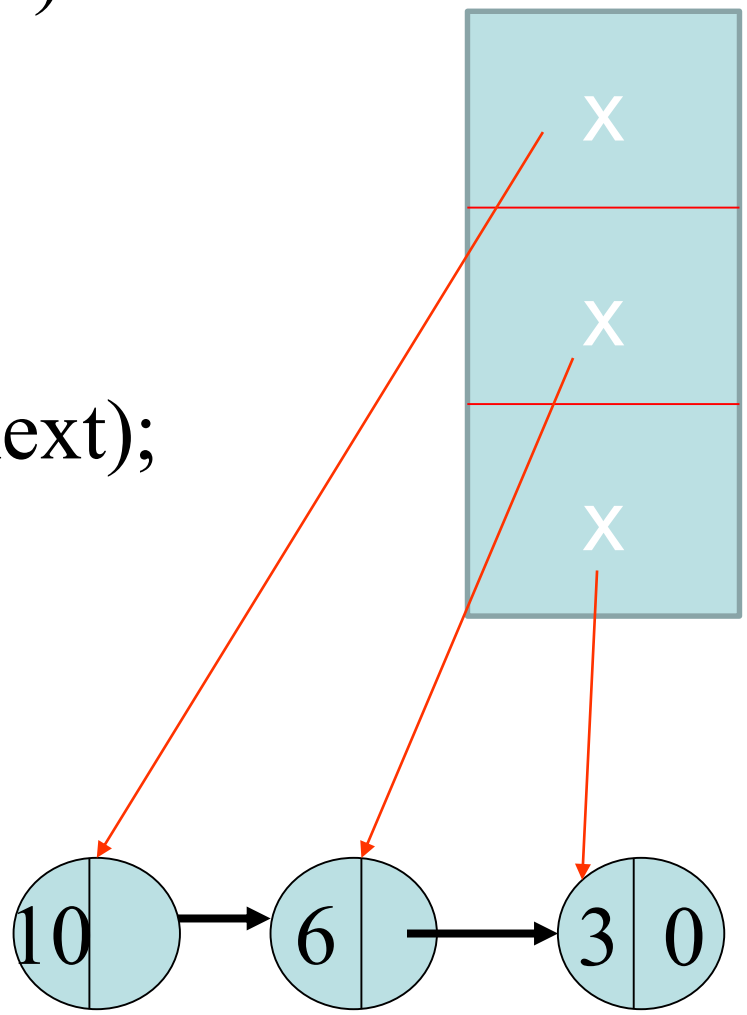


la variabile x scorre i 3 nodi, il ciclo si ferma quando x=0. cioè quando «esce» dalla lista

stampa dal fondo

```
void stampa_rov(nodo *x)
{
        if(x)
        {
          stampa_rov(x->next);
          cout<< x->info;
        }
}
```

servono le 3 x !!
non è terminale !!



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

ricorsione terminale
equivalente a while

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

non terminale
più complicato da
simulare col while

IMPORTANTE: la new può fallire !!

potrebbe non esserci memoria Ram sufficiente

```
int *  x= new int[1000000];
if(x==NULL)   //  la new è fallita
throw(..);
```

la costante predefinita NULL ha valore 0 ed è da usare solo con puntatori, altrimenti warning !