Programmazione 2

Gestione della memoria in C

Il linguaggio C permette l'allocazione dinamica della memoria, attraverso apposite funzioni.

Tipologia	Descrizione
Stack	Gestito da compiler e OS.
Empty	Non eseguibile, modificabile.
Heap	Gestita dal programmatore.
Variabili globali	Allocate a inizio programma.
Costanti, lettere	Allocato a inizio programma, read only non eseguibile.
Codice programma	Allocato a inizio programma, read only eseguibile.

Lo Stack

Lo stack è chiamato così perché è una pila di elementi, si possono inserire solo in cima così come si toglie sempre dalla cima (LIFO). Viene usato per immagazzinare i **record di attivazione**. Cos'è un record di attivazione? Si tratta della zona di memoria che contiene tutti i valori come variabili locali e parametri.

- Quando viene **invocata**: viene creato un record di attivazione in memoria nello stack (**push**).
- Quando viene **terminata**: viene cancellato il record di attivazione dalla memoria (**pop**).

Un record di attivazione ha la seguente struttura:

Tipologia	Descrizione
Indirizzo di ritorno	Indica dove bisogna riprendere il programma dopo la funzione.
Link dinamico	Indica chi ha chiamato la funzione tramite il record di attivazione del chiamante.
Parametri di input	Parametri passati nella dichiarazione.
Variabili locali	Variabili create nella funzione.
Valore di ritorno	Opzionale perché esiste “void” come valore di return.

Esempio con il seguente codice, una funzione `potenza()` chiamata dalla funzione `main()` e una variabile globale `int ris = 0`:

```
int potenza(int x, int n) {
    int out = 1;
    for(int i = 0; i < n; i++
        out = out * x;
    }
    return out;
}
```

```
int main() {
    int x = 3, n = 2;
    ris = potenza(x, n); // <
    printf("Potenza: %d", ris
}
```

Tipologia	Valore
Indirizzo return	α
Link dinamico	main()
Parametri input	x, n
Variabili locali	out, i
Valore return	9

Come possiamo vedere l’indirizzo di return ci dice che dopo l’esecuzione di `potenza()` dobbiamo tornare a quella riga. Il link dinamico ci dice chi ha chiamato la funzione.

La Heap

La memoria heap viene **gestita** in maniera **manuale** dal programmatore, ha una dimensione più grande dello stack, il problema della heap è che avendo questo libertà deve essere gestita in maniera intelligente perché a differenza dello stack **non è una sequenza** di elementi in ordine LIFO. L’allocazione viene fatta nel **primo spazio disponibile**.

Stack	Heap
La memoria è gestita in automatico	La memoria è gestita manualmente
Dimensioni ridotte	Dimensioni considerevoli
Non flessibile, la memoria allocata non può essere cambiata	Flessibile, la memoria allocata può essere cambiata
Accesso veloce, allocazione e de allocazione.	Accesso lento, allocazione e de allocazione.
La visibilità degli elementi è limitata al thread.	La visibilità degli elementi è globale al programma.
Il OS alloca lo stack quando il thread è creato.	Il OS è chiamato dal linguaggio per allocare la memoria heap.

Strutture dati

In C si possono creare delle strutture dati, per definire nuovi tipi di dati, esistono quelli di base e quelli definiti dall'utente. ad esempio un punto in un piano cartesiano verrà definito da:

```
// struct nome { tipo variabile };
struct punto {
    float x;
    float y;
};
```

Possiamo definire anche strutture di strutture come ad esempio una figura geometrica che è formata da punti:

```
struct rettangolo {
    struct punto p1; // a loro volta sono definiti da una x e u
    struct punto p2; // possiamo usare questa forma all'infinit
}
```

Il passaggio delle **strutture dati è eseguita per copia**, non per riferimento, quindi il loro valore è copiato in ram e viene modificata la copia dentro la funzione.

Definire una nuova struttura

Usiamo la parola chiave **typedef** per creare un nuovo tipo di dato, così da per rendere più leggibile il codice.

Complessità di un algoritmo

Bisogna riuscire a valutare l'utilizzo di spazio e tempo in base al tipo di algoritmo. Si calcolano complessità di **tempo** e di **spazio**. Cosa influenza la complessità? Tipo di **algoritmo**, **dimensione input** e **velocità della macchina** (anche se quest'ultima è trascurabile da noi).

Il fattore più importante è la dimensione dell'input ad esempio un ordinamento su 10 o 10000 elementi cambia la velocità di risoluzione.

Esistono due tipologie di analisi:

- **Empirica**: dati due algoritmi che risolvono lo stesso problema bisogna identificare qualche impiega più tempo. Si fa una stima su dati reali, il problema è capire se l'algoritmo è veloce davvero oppure abbiamo inserito dei dati "comodi".
- **Matematica**: si conta il numero di esecuzioni delle operazioni fondamentali degli algoritmi.

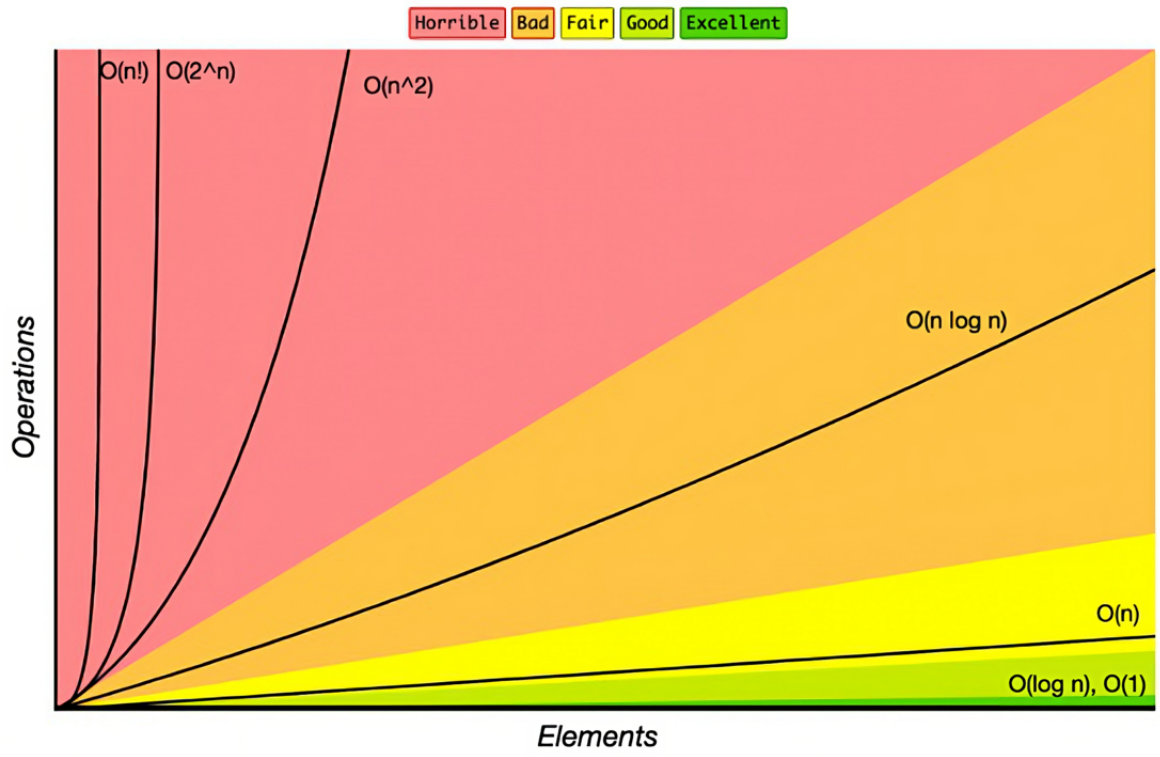
La **complessità asintotica** valuta il costo di esecuzione di un dato algoritmo in termini asintotici ovvero in input sufficientemente grandi potenzialmente tendenti a infinito. Input piccoli potrebbero mascherare la complessità.

Tipi di complessità:

- **Tempo:** numero di passi / istruzioni richiesti dall'algoritmo.
- **Spazio:** memoria richiesta dall'algoritmo ovvero il record di attivazioni richiesti contemporaneamente sullo stack.

La notazione O() serve per descrivere il comportamento asintotico delle funzioni matematiche.

In un grafico se è alto il numero delle operazioni e piccolo il numero di input allora l'algoritmo è fatto male. Al contrario se con poche operazioni si possono gestire molti input l'algoritmo sarà veloce e fatto bene.



Ci sono dei limiti alle implementazioni, sotto una certa soglia non si può scendere, neanche con la migliore delle implementazioni.

Esempi:

- 5 ha complessità $O(1)$.
- 5 + 2n ha complessità $O(n)$.
- 5 + $2n^2$ ha complessità $O(n^2)$.
- etc...

Serve conoscere la complessità per evitare situazioni in cui il nostro algoritmo ci impeghi troppo tempo.

Calcolo complessità

$\sum costo\ singole\ istruzioni$ da cui dobbiamo ricavare:

- **Istruzioni elementari:** aritmetiche, lettura / scrittura, condizioni logiche, operandi di input / output.
- **Istruzioni condizionali**: condizione di if, switch.

- **Istruzioni iterative**: cicli while, cicli for.

Complessità in spazio

Bisogna vedere **quanti record di attivazione sono caricati allo stesso tempo** e prendere il numero massimo. Ad esempio:

Stack
A

Stack
A
B

Stack
A
B
C

Stack
A
E

In questo caso sappiamo che il numero di picco è tre record di attivazioni attivati contemporaneamente. Per le **funzioni ricorsive** il valore di memoria è definito **quante volte viene chiamata** la funzione ricorsiva.

Allocazione dinamica della memoria

Andremo a lavorare con la memoria heap, **gestita da noi** programmatori, è **visibile globalmente** da tutto il programma, se allochiamo uno spazio di memoria non sparisce se termina la funzione, può essere un pregio e un difetto.

La prima funzione che vediamo è `malloc` ovvero una funzione che permette di allocare un **blocco continuo di memoria** di dimensione size.

In output viene fornito un puntatore alla zona di memoria, fornisce un `void` così che quando serve facciamo un cast sul tipo puntato.

La seconda funzione è `sizeof` è ottenere la dimensione di un tipo di dato in memoria.

```
float *p;
p = (float*) malloc(sizeof(float)); // Abbiamo assegnato un'area
p = (float*) malloc(sizeof(float) * 5);// Abbiamo assegnato
                                       // un'area di memoria grand
```

Per non perdere blocchi di memoria dobbiamo usare la funzione `free` la quale libera lo spazio grazie al puntatore dell'area.

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

int main() {
    int *a, *b;
    int n;
    printf("Dimensione di a: ");
    scanf("%d", &n);

    a = (int*) malloc(sizeof(int) * n);
    for(int i = 0; i < n; i++) {
        a[i] = 1000 + i;
        printf("a[%d] = %d\n",i+1, a[i]);
    }

    //free(a);
    n += 20;
    b = (int*) malloc(sizeof(int) * n);
    for(int i = 0; i < n; i++) {
        printf("b[%d] = %d\n",i+1, b[i]);
    }
}
```

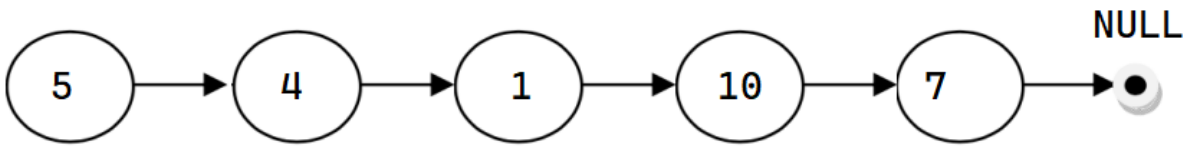
Il problema di `malloc` è che non inizializza una zona di memoria, se fosse già stata usata noi avremmo gli scarti, con `calloc` invece la zona di memoria viene inizializzata ovviamente a un **costo di performance**.

Un'altra funzione da conoscere è `realloc` che serve per riallocare lo spazio precedentemente allocato, serve anche a modificare la dimensione di un blocco di memoria, se non dovesse bastare lo spazio si cerca in un altro spazio.

Liste dinamiche

Le liste sono una **sequenza** ordinata di **elementi detti nodi**, i nodi possono contenere qualunque tipo di informazione, una **lista vuota** è comunque una **lista**, un **nodo seguito da una lista è una lista**.

Un tipo primitivo in C per le liste non esiste, va creato. Con le struct e con le typedef possiamo risolvere questo problema.



Ogni nodo punta allo spazio di memoria del successivo, l'ultimonodo punta a **NULL,** il tipo di dato può anche essere un'altra struct.

Esempio di nodo generico dove abbiamo un DATA e un LINK, il **primo è il dato** in sè mentre **LINK è il puntatore** al nodo successivo.

Visita lista - Pattern Generale (condizione)

```
… visita_con_condizione(LINK lis,…) {
   while (lis != NULL) {
      if(CONDIZIONE) {
         OPERAZIONE(I) SUL SINGOLO ELEMENTO
      }
      lis= lis->next;
   }
…
}
```

Se volessimo solo stampare basta togliere l'if e mettere `printf()`.

Visita condizionata con accumulatore

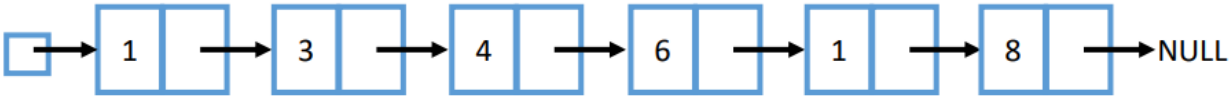
```
... visita_condizionata_var(LINK lis,…) {
   INIZIALIZZAZIONE VARIABILE
   while (lis != NULL) {
      if(CONDIZIONE SU VARIABILE) {
         OPERAZIONE(I) SU SINGOLO ELEMENTO
      }
      AGGIORNAMENTO VARIABILE
      lis= lis->next;
   }
…
}
```

Se volessimo svolgere l'operazione solo su n numeri della lista, aggiungiamo la condizione nel while, esempio: `while(lis != NULL && n < 10).`

Algoritmi con finestra

Sono quelli che analizzano più elementi della lista contemporaneamente, per questo bisogna avere i dovuti accorgimenti.

Esempio: contare quanti numeri sono minore del successivo.



Per farlo dobbiamo prima di tutto assicurarci che **ALMENO** i primi due elementi siano diversi da NULL e successivamente cambiare la condizione del while, abbiamo analizzato già l'lemento attuale, l'algoritmo continua fin quando il **nodo NEXT è diverso da NULL**.

```
int elementi_minori(LINK lis) {
    int cnt=0;
```

```
    if(lis == NULL) return 0;
    if(lis->next == NULL) return 0;

    while (lis->next != NULL) {
        if(lis->d < lis->next->d ) cnt++;
        lis= lis->next;
    }
    return cnt;
}
```

Otteniamo così il nostro pattern generale:

```
… funzione(LINK lis,…) {
   VERIFICA ESISTENZA FINESTRA
   while (FINO ALLA FINE DELLA FINESTRA) {
      OPERAZIONI CON ELEMENTI IN FINESTRA
   lis= lis->next;
   }
…
}
```

La finestra cambia a seconda del problema, può essere che ci sia una finestra anche su 3 elementi e dovremmo controllarli tutti prima difare il while. **In generale l'ultimo if da la condizione del while.**

Ricerca di un elemento

Ci possono essere due possibili scenari: ricerca **elemento** e ricerca **posizione.** In entrambi i casi **dobbiamo restituire il nodo** se esiste in questo modo:

```
LINK find_pos(int x, LINK p) {
    int pos=1;
    while((pos < x) && (p!= NULL)) {
        p=p->next;
```

```
        pos+=1;
    }
    return(p);
}
// Oppure
LINK find(int x, LINK p) {
    int found=0;
    while((p != NULL) && (!found)) {
        if (p->d==x) found=1;
        else p=p->next;
    }
    return(p);
}
```

Se volessimo il nodo precedente a un determinato x possiamousare questa soluzione:

```
LINK findpred(int x, LINK lis) {
    int trovato=0;
    if (lis == NULL) {
        printf("lista vuota\n");
        return(NULL);
    }
    else {
        if (lis->d == x) {
        printf("%d e' a inizio lista\n", x);
        return(NULL);
    }
    else {
        while ((lis->next != NULL) && (! trovato))
            if (lis->next->d == x) trovato=1;
            else lis=lis->next;
            if (trovato) return(lis);
            else return(NULL);
        }
```

```
    }
}
```

Inserimento in una lista

Per modificare una lista ci sono due vie: se modifichiamo i valori dei nodi per valore altrimenti se vogliamo aggiungere un nodo per riferimento.

```
void tailinsert(LINK *lis, int x) {
    LINK p,q;
    p=newnode();
    p-> d = x;
    p-> next = NULL;
    q = *lis;
    if (q == NULL) {
        *lis = p;
    }
    else {
        while (q->next != NULL) q = q->next;
        q-> next = p;
    }
}
```

Quando vogliamo inserire un nuovo nodo dobbiamo anche controllare che non sia NULL inizialmente, per inserire in coda bisogna fare questo controllo e poi scorrere la lista fino a che il successivo non è null, in quel caso si inserisce il nodo.

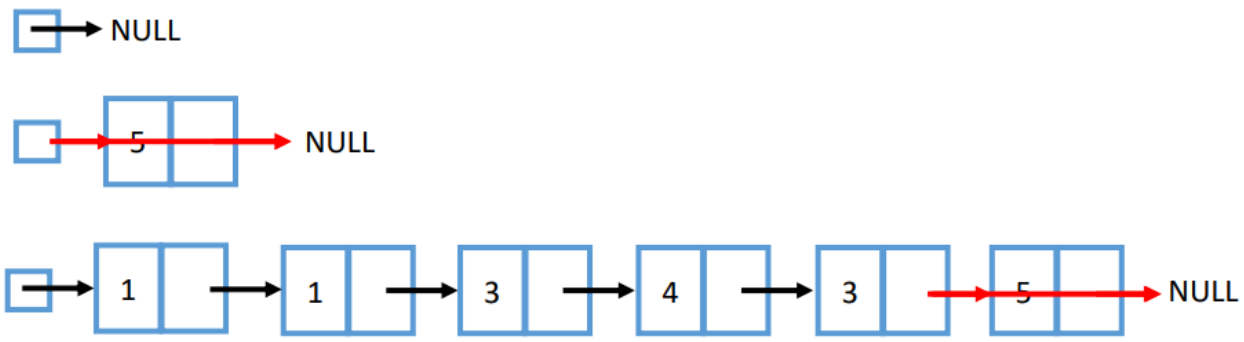
Cancellazione di un nodo

Per modificare una lista come abbiamo detto bisogna passare la lista per riferimento, dobbiamo fare in questo modo:

```
void removefirst(LINK *lis) {
    LINK p;
    if (*lis != NULL) {
```

```
        p=*lis;
        *lis=(*lis)->next;
        free(p);
    }
}
```

Se volessimo eliminare tutti i nodi dobbiamo sostituire l'if con il while.



Duplicazione di una lista

Quando vogliamo duplicare una lista dobbiamo creare una variabile di tipo list e creare un heap tanti spazi in memoria quanti necessari. Dobbiamo creare una testa che punta a NULL e da lì appendere i nodi uno dopo l'altro verificando che nella lista da duplicare l'elemento punti a un elemento diverso da NULL.

```
void duplist(LINK lis, LINK *new){
    LINK p, tail;
    while (lis != NULL ) {
        p = newnode();
        p->d = lis->d;
        p->next = NULL;
        if (*new == NULL ) {
            *new = p;
            tail = p;
        } else {
            tail->next = p;
            tail = p;
```

```
        }
        lis= lis->next;
    }
}
```

Operazioni su più liste

Ecco il pattern generale su operazioni su più liste

```
... f(LINK l1, LINK l2, ...) {
    while ((l1!=NULL)&&(l2!=NULL)) {
        OPERAZIONI
        l1 = l1->next;
        l2=l2->next;
     }
    while (l1!=NULL) {
        OPERAZIONI
        l1 = l1->next;
    }
    while (l2!=NULL) {
        OPERAZIONI
        l2 = l2->next;
    }
eventuale return
}
```

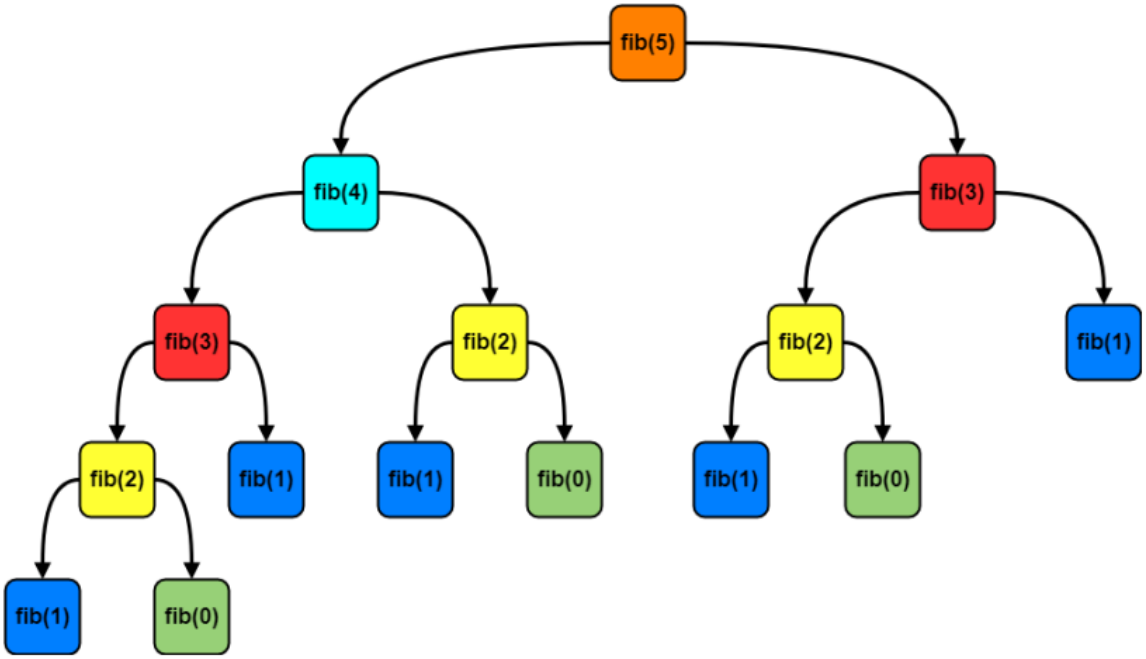
Complessità?
O(max(m,n)) *con m ed n lunghezza delle liste L1 e L2*

Altro pattern:

```
LINK build(LINK l1, LINK l2) {
     LINK p, head, tail;
     head=NULL; tail=NULL;
     while ((l1 != NULL) && (l2 != NULL)) {
          p=newnode();
          OPERAZIONE SU p->d DIPENDENTE DA l1->d E l2->d;
          p->next = NULL;
          if (head == NULL){ head=p; tail=p;}
          else { tail->next=p; tail=p;}
          l1 = l1->next; l2 = l2->next;
     }
     while (l1 != NULL) {
          p=newnode();
          OPERAZIONE SU p->d DIPENDENTE DA l1->d;
          p->next = NULL;
          if (head == NULL){head=p; tail=p;}
          else {tail->next=p; tail=p;}
          l1 = l1->next;
     }
     while (l2 != NULL) {
          p=newnode();
          OPERAZIONE SU p->d DIPENDENTE DA l2->d;
          p->next = NULL;
          if (head == NULL){head=p; tail=p;}
          else {tail->next=p; tail=p;}
          l2 = l2->next;
     }
     return (head);
```

Introduzione alla ricorsione

Una funzione è ricorsiva se la funzione è chiamata dentro se stessa. Serve per programmare in maniera efficente (non sempre) richiede più spazio per i record di attivazione.



Ci sono due tipi di ricorsione: **diretta** e **indiretta**.

- **Diretta**: A richiama A
- **Indiretta**: A richiama B e B richiama A

Si possono ovviamente applicare queste funzioni alle liste in questo modo:

```
void printlis_rt(LINK lis) {
    if (lis != NULL) {
        printf(">>>> %d\n", lis->d);
        printlis_rt(lis->next);
    }
}
```

```
… visita_incondizionata(LINK lis) {
    if (lis != NULL) {
        OPERAZIONE SUL SINGOLO ELEMENTO
        visita_incondizionata(lis->next);
    }
}
```

Diciamo di coda quando c'è solo il return senza altre operazioni prima del **return**.
Possiamo anche creare una lista con la ricorsione seguendo il seguente algoritmo:

```
LINK buildlis_rnf() {
    int x;
    LINK p;
    printf("nuovo numero da inserire in lista:\n");
    scanf("%d", &x);
    if (x<=0) return NULL;
    else {
        p=newnode();
        p->d = x;
        p->next = buildlis_rnf();
        return p;
    }
}
```

Ma anche duplicare una lista

```
LINK duplis(LINK lis) {
    LINK p;
    if (lis == NULL) return NULL;
    else {
        p=newnode();
        p->d = lis->d;
        p->next = duplis(lis->next);
```

```
        return p;
    }
}
```

Oppure cancellare tutti i nodi

```
void diposelis_r(LINK *lis) {
    LINK p;
    if (*lis != NULL) {
        p=*lis;
         *lis=(*lis)->next;
        diposelis_r(lis);
        free(p);
    }
}
```