

Laboratorio di Programmazione

Corso di Laurea in Informatica

Gr. 3 (N-Z)

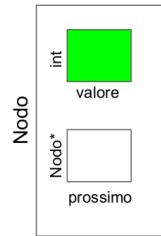
Università degli Studi di Napoli Federico II

A.A. 2022/23
A. Apicella

Prerequisito: strutture ricorsive

Una struttura è detta **ricorsiva** se contiene al suo interno un membro di tipo puntatore dello stesso tipo della struttura.

```
struct Nodo {  
    int valore;  
    struct Nodo* prossimo;  
};
```

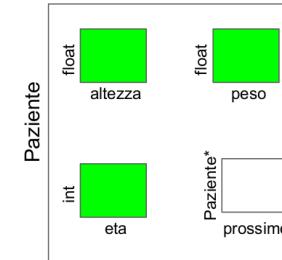


In generale, una struttura non può contenere un'istanza del suo stesso tipo, ma nulla vieta che contenga un **puntatore** ad un'istanza del suo stesso tipo

Prerequisito: strutture ricorsive

Una struttura è detta **ricorsiva** se contiene al suo interno un membro di tipo puntatore dello stesso tipo della struttura.

```
struct Paziente
{
    float altezza;
    float peso;
    int eta;
    struct Paziente* prossimo;
};
```

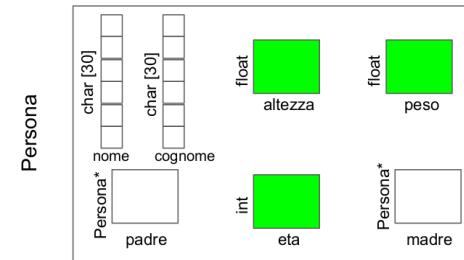


In generale, una struttura non può contenere un'istanza del suo stesso tipo, ma nulla vieta che contenga un **puntatore** ad un'istanza del suo stesso tipo

Prerequisito: strutture ricorsive

Una struttura è detta **ricorsiva** se contiene al suo interno un membro di tipo puntatore dello stesso tipo della struttura.

```
struct Persona
{
    char nome[30];
    char cognome[30];
    float altezza;
    float peso;
    int eta;
    struct Persona* padre;
    struct Persona* madre;
};
```

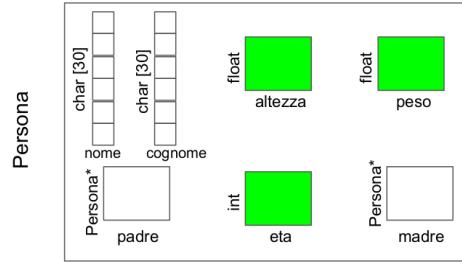


In generale, una struttura non può contenere un'istanza del suo stesso tipo, ma nulla vieta che contenga un **puntatore** ad un'istanza del suo stesso tipo

Prerequisito: strutture ricorsive

Una struttura è detta **ricorsiva** se contiene al suo interno un membro di tipo puntatore dello stesso tipo della struttura.

```
struct Persona
{
    char nome[30];
    char cognome[30];
    float altezza;
    float peso;
    int eta;
    struct Persona* padre;
    struct Persona* madre;
};
```



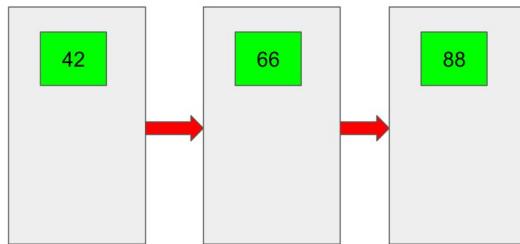
In generale, una struttura non può contenere un'istanza del suo stesso tipo, ma nulla vieta che contenga un **puntatore** ad un'istanza del suo stesso tipo

Si ricorda che quando si **definisce** una struct **non si sta allocando memoria**, ma si stanno soltanto descrivendo da quali campi dovrà essere formata la struct quando sarà materialmente allocata.

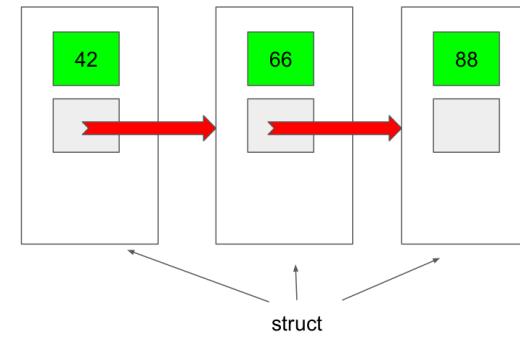
Liste concatenate

- struttura dati **dinamica**
 - gli elementi possono essere inseriti e/o eliminati in qualsiasi punto.
- In questo modo ogni Nodo può essere **allocato** se e solo quando necessario
- In base a come sono collegati i dati tra loro, possono essere creati diversi tipi di lista concatenata. Ad esempio:
 - 1. **a collegamento singolo**: ogni Nodo è collegato solo all'Nodo successivo
 - 2. **a collegamento doppio**: ogni Nodo è collegato sia con l'Nodo successivo che con l'Nodo precedente

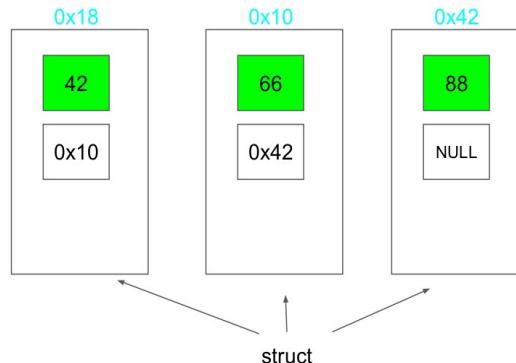
Liste a collegamento singolo



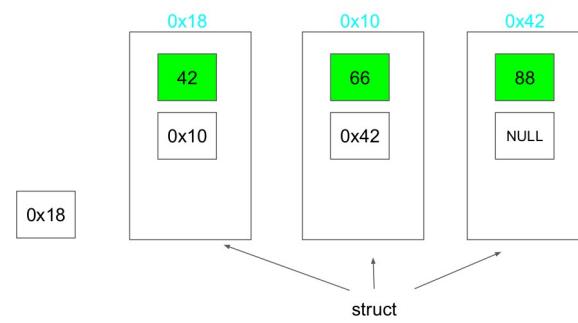
Liste a collegamento singolo



Liste a collegamento singolo



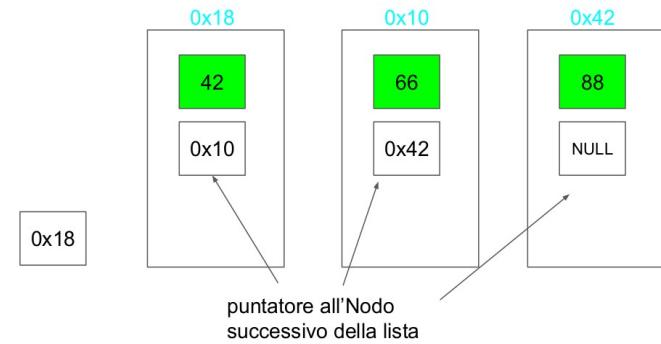
Liste a collegamento singolo



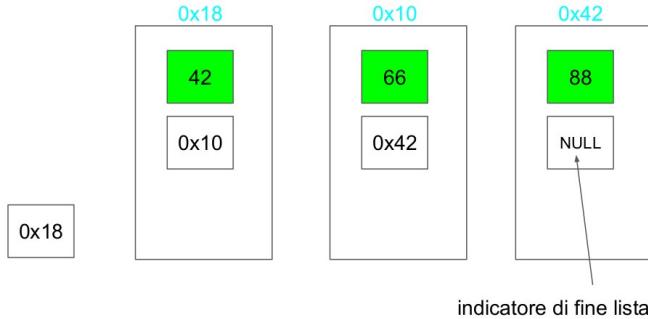
Liste a collegamento singolo



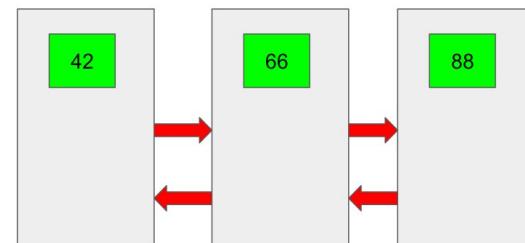
Liste a collegamento singolo



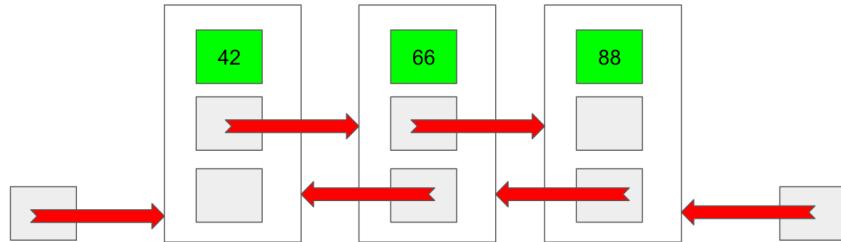
Liste a collegamento singolo



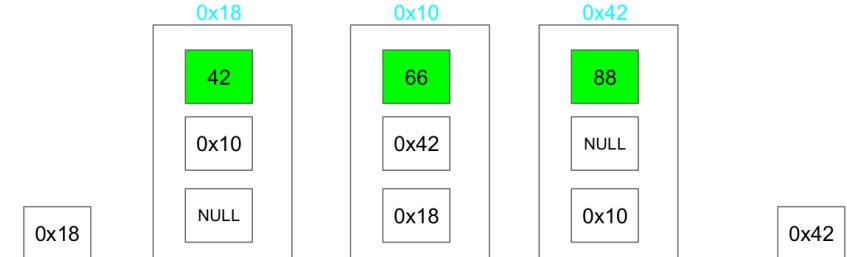
Liste a collegamento doppio



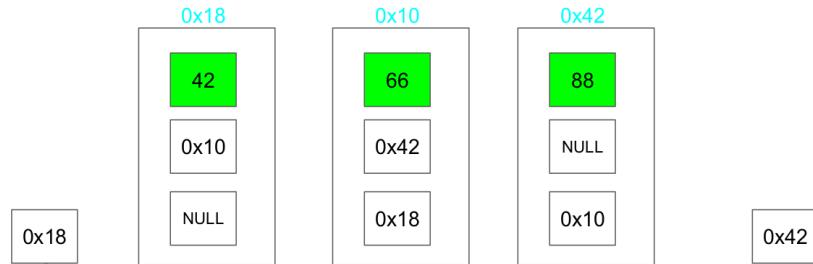
Liste a collegamento doppio



Liste a collegamento doppio

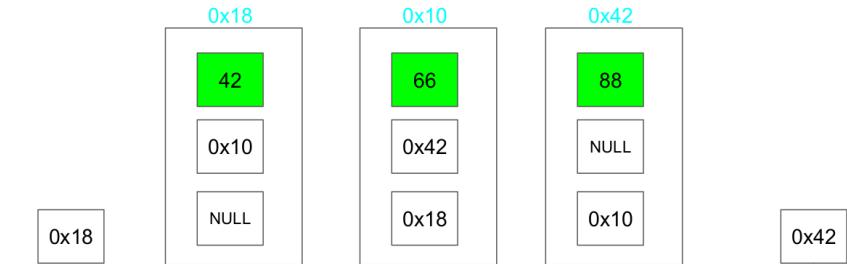


Liste a collegamento doppio



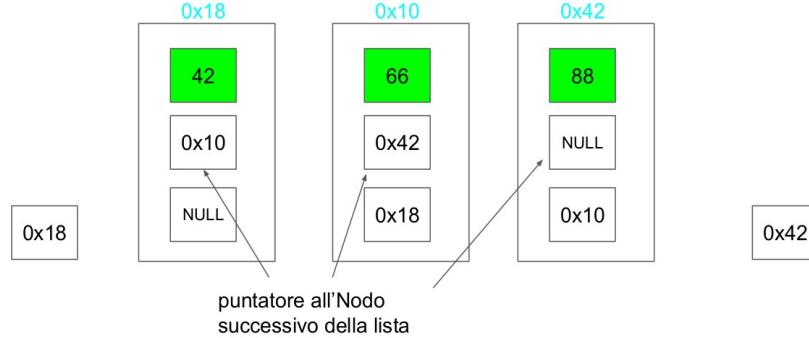
puntatore all'inizio della lista

Liste a collegamento doppio

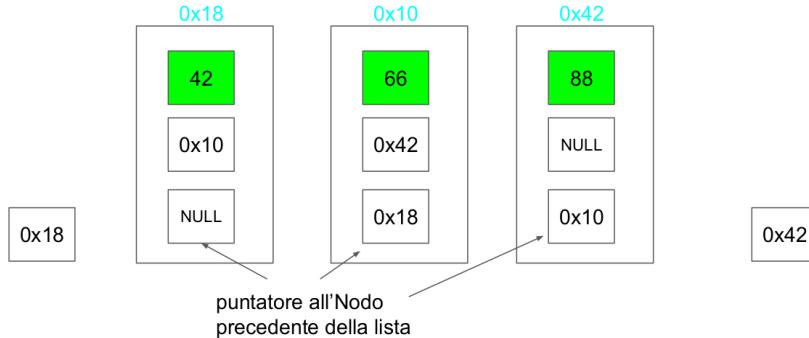


puntatore alla fine della lista

Liste a collegamento doppio



Liste a collegamento doppio



Liste concatenate

- In generale, Una lista concatenata quindi può essere vista come una sequenza di elementi (chiamati **nodi**) in cui ogni Nodo è in grado di puntare all'Nodo successivo.
- Ovviamente, è necessario un puntatore per accedere al primo Nodo della lista. Tale primo Nodo è generalmente chiamato **testa**.
- Nel caso di liste doppiamente concatenate, può essere inserito un ulteriore puntatore all'ultimo Nodo della lista.
- L'ultimo Nodo di una lista (ovviamente) non punterà ad alcun Nodo.
- Nel caso di liste doppiamente concatenate, non ci sarà alcun precedente del primo Nodo

Implementazione di liste concatenate

Esempi di cattive implementazioni

```
struct Nodo {  
    int valore;  
    struct Nodo* prossimo;  
};  
  
int main()  
{  
    struct Nodo* inizio = malloc(sizeof(struct Nodo));  
    inizio->valore = 42;  
    inizio->prossimo = malloc(sizeof(struct Nodo));  
    inizio->prossimo->valore = 66;  
    inizio->prossimo->prossimo = malloc(sizeof(struct Nodo));  
    inizio->prossimo->prossimo->valore = 88;  
    inizio->prossimo->prossimo->prossimo = NULL;  
  
    return 0;  
}
```

ogni Nodo viene allocato ed inserito
“manualmente”.

Implementazione di liste concatenate

Esempi di cattive implementazioni

```
#DEFINE N 3
struct Nodo {
    int valore;
    struct Nodo* prossimo;
};

int main()
{
    struct Nodo* inizio = malloc(sizeof(struct Nodo));

    struct Nodo* in_esame = inizio;
    for(int i = 0; i < N; i=i+1)
    {
        scanf("%d", &in_esame->valore);
        in_esame->prossimo = malloc(sizeof(struct Nodo));
        in_esame = in_esame->prossimo;
    }
    return 0;
}
```

viene allocato il primo Nodo della lista, il cui indirizzo finisce nel puntatore `inizio`

Implementazione di liste concatenate

Esempi di cattive implementazioni

```
#DEFINE N 3
struct Nodo {
    int valore;
    struct Nodo* prossimo;
};

int main()
{
    struct Nodo* inizio = malloc(sizeof(struct Nodo));
    struct Nodo* in_esame = inizio;
    for(int i = 0; i < N; i=i+1)
    {
        scanf("%d", &in_esame->valore);
        in_esame->prossimo = malloc(sizeof(struct Nodo));
        in_esame = in_esame->prossimo;
    }
    return 0;
}
```

viene allocato un nuovo puntatore `in_esame` in modo da non alterare il puntatore `inizio`

Implementazione di liste concatenate

Esempi di cattive implementazioni

```
#DEFINE N 3
struct Nodo {
    int valore;
    struct Nodo* prossimo;
};

int main()
{
    struct Nodo* inizio = malloc(sizeof(struct Nodo));

    struct Nodo* in_esame = inizio;
    for(int i = 0; i < N; i=i+1) ← in maniera iterativa, ad ogni iterazione, viene:
    {
        scanf("%d", &in_esame->valore);
        in_esame->prossimo = malloc(sizeof(struct Nodo));
        in_esame = in_esame->prossimo;
    }
    return 0;
}
```

Implementazione di liste concatenate

Esempi di cattive implementazioni

```
#DEFINE N 3
struct Nodo {
    int valore;
    struct Nodo* prossimo;
};

int main()
{
    struct Nodo* inizio = malloc(sizeof(struct Nodo));

    struct Nodo* in_esame = inizio;
    for(int i = 0; i < N; i=i+1) ← assegnato un valore al campo valore del record
    {
        scanf("%d", &in_esame->valore);
        in_esame->prossimo = malloc(sizeof(struct Nodo));
        in_esame = in_esame->prossimo;
    }
    return 0;
}
```

Implementazione di liste concatenate

Esempi di cattive implementazioni

```
#DEFINE N 3
struct Nodo {
    int valore;
    struct Nodo* prossimo;
};

int main()
{
    struct Nodo* inizio = malloc(sizeof(struct Nodo));

    struct Nodo* in_esame = inizio;
    for(int i = 0; i < N; i=i+1)
    {
        scanf("%d", &in_esame->valore);
        in_esame->prossimo = malloc(sizeof(struct Nodo));
        in_esame = in_esame->prossimo;
    }
    return 0;
}
```

allocato il successivo Nodo della lista, il cui indirizzo finisce nel puntatore successivo dell'Nodo della lista in esame

Implementazione di liste concatenate

Esempi di cattive implementazioni

```
#DEFINE N 3
struct Nodo {
    int valore;
    struct Nodo* prossimo;
};

int main()
{
    struct Nodo* inizio = malloc(sizeof(struct Nodo));

    struct Nodo* in_esame = inizio;
    for(int i = 0; i < N; i=i+1)
    {
        scanf("%d", &in_esame->valore);
        in_esame->prossimo = malloc(sizeof(struct Nodo));
        in_esame = in_esame->prossimo;
    }
    return 0;
}
```

aggiornato il puntatore in_esame in modo che punti al prossimo Nodo

Implementazione di liste concatenate

Esempi di cattive implementazioni

```
#DEFINE N 3
struct Nodo {
    int valore;
    struct Nodo* prossimo;
};

int main()
{
    struct Nodo* inizio = malloc(sizeof(struct Nodo));

    struct Nodo* in_esame = inizio;
    for(int i = 0; i < N; i=i+1)
    {
        scanf("%d", &in_esame->valore);
        in_esame->prossimo = malloc(sizeof(struct Nodo));
        in_esame = in_esame->prossimo;
    }
    return 0;
}
```

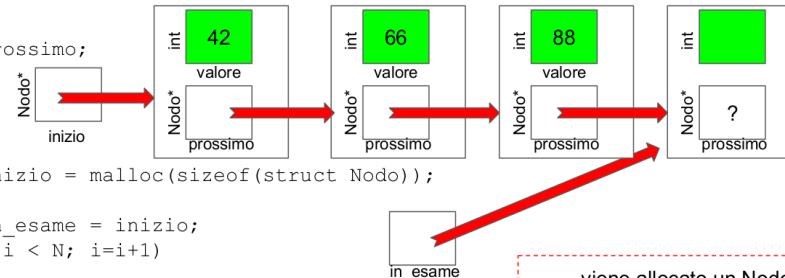
Cosa c'è di sbagliato in questa implementazione?

Implementazione di liste concatenate

esempi di cattive implementazioni

```
#DEFINE N 3
struct Nodo {
    int valore;
    struct Nodo* prossimo;
};
int main()
{
    struct Nodo* inizio = malloc(sizeof(struct Nodo));

    struct Nodo* in_esame = inizio;
    for(int i = 0; i < N; i=i+1)
    {
        scanf("%d", &in_esame->valore);
        in_esame->prossimo = malloc(sizeof(struct Nodo));
        in_esame = in_esame->prossimo;
    }
    return 0;
}
```



- viene allocato un Nodo in più
- l'ultimo Nodo non ha puntatore successore NULL

Implementazione di liste concatenate

Esempio di migliore implementazione

da evitare: ad ogni iterazione viene allocato spazio per un ipotetico Nodo successivo, senza considerare se tale Nodo sia realmente necessario o meno

possibile soluzione: generare, ad ogni iterazione, l'Nodo "attuale" su cui lavorare, e non il successivo.

- tenere conto che il primo Nodo, a differenza dei successivi deve andare nel puntatore `inizio`.

```
struct Nodo* inizio      = NULL;
struct Nodo* in_esame    = NULL;
struct Nodo* precedente = NULL;
for(int i = 0; i < N; i=i+1)
{
    in_esame = malloc(sizeof(struct Nodo)); // alloco lo spazio del nuovo Nodo da inserire;
    scanf("%d", &in_esame->valore);           // inserisco il valore;
    in_esame->prossimo = NULL;                // presuppongo che non ci siano ulteriori elementi;

    if (i == 0) // se i == 0, vuol dire che l'Nodo che si sta inserendo è il primo
    {
        inizio = in_esame;
    }
    else // se i>0, la lista non è vuota
    {
        precedente->prossimo = in_esame;
    }
    precedente = in_esame; //aggiorno il precedente con l'indirizzo dell'Nodo nuovo,
                          // così che possa essere utilizzato alla prossima iterazione
                          // come punto di inserimento di un eventuale nuovo Nodo.
}
```

Implementazione di liste concatenate

Esempio di migliore implementazione

E per visualizzare?

- è possibile identificare l'ultimo nodo, in quanto è l'unico il cui puntatore successore è `NULL`

```
in_esame = inizio;
while(in_esame != NULL)
{
    printf("%d\n", in_esame->valore);
    in_esame = in_esame->prossimo;
}
```

Implementazione di liste concatenate

Buone (e obbligatorie) norma

Prima di concludere il programma, **liberare tutto lo spazio allocato.**

```
struct Nodo* in_esame = inizio;
while(in_esame != NULL)
{
    free(in_esame);
    in_esame = in_esame->prossimo;
}
```

Implementazione di liste concatenate

Buone (e obbligatorie) norma

Prima di concludere il programma, **liberare tutto lo spazio allocato.**

```
struct Nodo* in_esame = inizio;
while(in_esame != NULL)
{
    free(in_esame);
    in_esame = in_esame->prossimo;
}
```

E' corretta questa implementazione?
Accede ad una sezione di memoria deallocated!

```
struct Nodo* in_esame = inizio;
while(in_esame != NULL)
{
    struct Nodo* prox = in_esame->prossimo;
    free(in_esame);
    in_esame = prox;
}
```

Implementazione di liste concatenate

Buone (e obbligatorie) norme

Prima di concludere il programma, **liberare tutto lo spazio allocato**.

```
struct Nodo* in_esame = inizio;
while(in_esame != NULL)
{
    free(in_esame);
    in_esame = in_esame->prossimo;
}
```

E' corretta questa implementazione?
Accede ad una sezione di memoria deallocata!

```
struct Nodo* in_esame = inizio;
while(in_esame != NULL)
{
    struct Nodo* prox = in_esame->prossimo;
    free(in_esame);
    in_esame = prox;
}
inizio = NULL; // buona norma
```

Implementazione di liste concatenate

In generale

In base a ciò che si desidera che contenga una lista concatenata (ed in che modo deve essere memorizzato), diverse funzioni possono essere implementate in lettura o scrittura.



- inserimento di un Nodo alla fine della lista (*in coda*)
 - inserimento di un Nodo all'inizio della lista (*in testa*)
 - inserimento di un Nodo in una data posizione
 - inserimento di un Nodo secondo un criterio di ordinamento
 - eliminazione dell'Nodo all'inizio
 - eliminazione dell'Nodo alla fine
 - eliminazione di un Nodo in una data posizione
 - eliminazione dell'intera lista
 - modifica dell'Nodo all'inizio
 - modifica dell'Nodo alla fine
 - modifica dell'Nodo in una data posizione
 - spostamento di un Nodo lungo la lista
- lettura dell'Nodo alla fine della lista (*in coda*)
 - lettura dell'Nodo all'inizio della lista (*in testa*)
 - lettura di un Nodo in una data posizione
 - lettura di tutti gli elementi presenti
 - ricerca di un Nodo all'interno della lista

Implementazione di liste concatenate

Inserimento di un Nodo in coda

```
void insert_at_the_end(struct nodo** start, int val)
{
    // 1. genero il nuovo nodo
    struct nodo* nuovo;
    nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;

    // 2. scorro la lista fino a raggiungere l'ultimo nodo
    struct nodo* in_esame   = *start;
    struct nodo* precedente = NULL; // tengo traccia del precedente

    while(in_esame != NULL)
    {
        precedente = in_esame;
        in_esame    = in_esame->prossimo;
    }

    // se non c'è precedente, vuol dire che si sta inserendo il primo Nodo
    if( precedente == NULL )
        *start = nuovo; //se la lista è vuota, inserisco il nuovo nodo come primo nodo
    else
        precedente->prossimo = nuovo;
}
```

Implementazione di liste concatenate

Inserimento di un Nodo in coda

```
void insert_at_the_end(struct nodo** start, int val)
{
    // 1. genero il nuovo nodo
    struct nodo* nuovo;
    nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;

    // 2. scorro la lista fino a raggiungere l'ultimo nodo
    struct nodo* in_esame   = *start;
    struct nodo* precedente = NULL; // tengo traccia del precedente

    while(in_esame != NULL)
    {
        precedente = in_esame;
        in_esame    = in_esame->prossimo;
    }

    // se non c'è precedente, vuol dire che si sta inserendo il primo Nodo
    if( precedente == NULL )
        *start = nuovo; //se la lista è vuota, inserisco il nuovo nodo come primo nodo
    else
        precedente->prossimo = nuovo;
}

int main()
{
    struct nodo* inizio = NULL;
    insert_at_the_end(&inizio, 42);
    .
    .
    .
}
```



Implementazione di liste concatenate

Inserimento di un Nodo in coda

```
void insert_at_the_end(struct nodo** start, int val)
{
    // 1. genero il nuovo nodo
    struct nodo* nuovo;
    nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore = val;
    nuovo->prossimo = NULL;

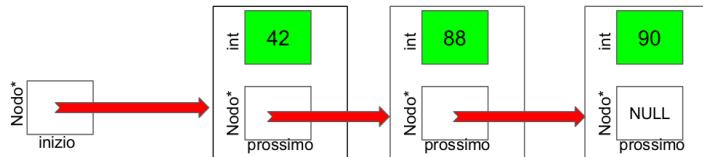
    // 2. scorro la lista fino a raggiungere l'ultimo nodo
    struct nodo* in_esame = *start;
    struct nodo* precedente = NULL; // tengo traccia del precedente

    while(in_esame != NULL)
    {
        precedente = in_esame;
        in_esame = in_esame->prossimo;
    }

    // se non c'è precedente, vuol dire che si sta inserendo il primo Nodo
    if( precedente == NULL )
        *start = nuovo; // se la lista è vuota, inserisco il nuovo nodo come primo nodo
    else
        precedente->prossimo = nuovo;
}
```

```
int main()
{
    struct nodo* inizio = NULL;
    insert_at_the_end(&inizio, 42);
    insert_at_the_end(&inizio, 88);
    insert_at_the_end(&inizio, 90);
    return 0;
}

alla fine del ciclo, precedente punterà all' ultimo Nodo e in_esame conterrà NULL
```



Implementazione di liste concatenate

Rimozione dell'Nodo in coda

```
void delete_from_the_end(struct nodo** start)
{
    if (*start == NULL)
        return;
    // 1. scorro la lista fino a raggiungere l'ultimo nodo
    struct nodo* in_esame = *start;
    struct nodo* precedente = NULL; // tengo traccia del precedente

    while(in_esame->prossimo != NULL)
    {
        precedente = in_esame;
        in_esame = in_esame->prossimo;
    }

    // 2. re-imposto la lista
    struct nodo* todel = in_esame; // salvo l'indirizzo dell'Nodo da eliminare
    if (precedente == NULL) // se non c'è precedente, il primo Nodo è anche ultimo
        *start = NULL;
    else // il penultimo nodo viene impostato come ultimo nodo
        precedente->prossimo = NULL;
    // 3. deallocazione effettiva
    free(todel);
}
```

Implementazione di liste concatenate

Rimozione dell'Nodo in coda - esempio di esecuzione

```
void delete_from_the_end(struct nodo** start)
{
    if (*start == NULL)
        return;
    // 1. scorro la lista fino a raggiungere l'ultimo nodo
    struct nodo* in_esame = *start;
    struct nodo* precedente = NULL; // tengo traccia del precedente

    while(in_esame->prossimo != NULL)
    {
        precedente = in_esame;
        in_esame = in_esame->prossimo;
    }

    // 2. re-imposto la lista
    struct nodo* todel = in_esame; // salvo l'indirizzo dell'Nodo da eliminare
    if (precedente == NULL) // se non c'è precedente, il primo Nodo è anche ultimo
        *start = NULL;
    else // il penultimo nodo viene impostato come ultimo nodo
        precedente->prossimo = NULL;
    // 3. deallocazione effettiva
    free(todel);
}
```



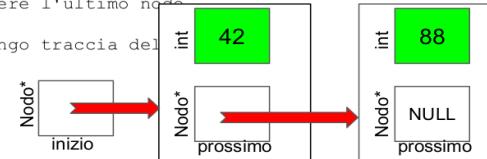
Implementazione di liste concatenate

Rimozione dell'Nodo in coda - esempio di esecuzione

```
void delete_from_the_end(struct nodo** start)
{
    if (*start == NULL)
        return;
    // 1. scorro la lista fino a raggiungere l'ultimo nodo
    struct nodo* in_esame = *start;
    struct nodo* precedente = NULL; // tengo traccia del precedente

    while(in_esame->prossimo != NULL)
    {
        precedente = in_esame;
        in_esame = in_esame->prossimo;
    }

    // 2. re-imposto la lista
    struct nodo* todel = in_esame; // salvo l'indirizzo dell'Nodo da eliminare
    if (precedente == NULL) // se non c'è precedente, il primo Nodo è anche ultimo
        *start = NULL;
    else // il penultimo nodo viene impostato come ultimo nodo
        precedente->prossimo = NULL;
    // 3. deallocazione effettiva
    free(todel);
}
```



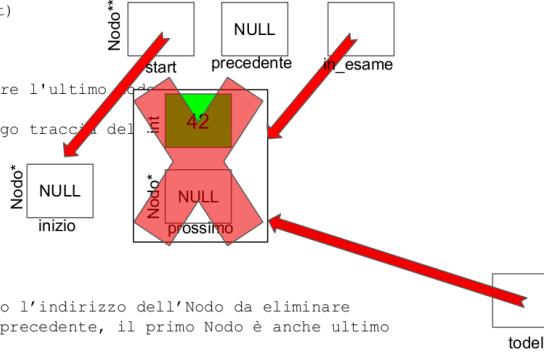
Implementazione di liste concatenate

Rimozione dell'Nodo in coda - esempio di esecuzione su lista di un solo Nodo

```
void delete_from_the_end(struct nodo** start)
{
    if (*start == NULL)
        return;
    // 1. scorro la lista fino a raggiungere l'ultimo nodo
    struct nodo* in_esame = *start;
    struct nodo* precedente = NULL; // tengo traccia del precedente

    while(in_esame->prossimo != NULL)
    {
        precedente = in_esame;
        in_esame = in_esame->prossimo;
    }

    // 2. re-imposto la lista
    struct nodo* todel = in_esame; // salvo l'indirizzo dell'Nodo da eliminare
    if (precedente == NULL) // se non c'è precedente, il primo Nodo è anche ultimo
        *start = NULL;
    else // il penultimo nodo viene impostato come ultimo nodo
        precedente->prossimo = NULL;
    // 3. deallocazione effettiva
    free(todel);
}
```



Implementazione di liste concatenate

Rimozione dell'Nodo in coda - esempio di esecuzione su lista di un solo Nodo

```
void delete_from_the_end(struct nodo** start)
{
    if (*start == NULL)
        return;
    // 1. scorro la lista fino a raggiungere l'ultimo nodo
    struct nodo* in_esame = *start;
    struct nodo* precedente = NULL; // tengo traccia del precedente

    while(in_esame->prossimo != NULL)
    {
        precedente = in_esame;
        in_esame = in_esame->prossimo;
    }

    // 2. re-imposto la lista
    struct nodo* todel = in_esame; // salvo l'indirizzo dell'Nodo da eliminare
    if (precedente == NULL) // se non c'è precedente, il primo Nodo è anche ultimo
        *start = NULL;
    else // il penultimo nodo viene impostato come ultimo nodo
        precedente->prossimo = NULL;
    // 3. deallocazione effettiva
    free(todel);
}
```



Implementazione di liste concatenate

Inserimento di un Nodo in testa

```
void insert_at_the_beginning(struct nodo** start, int val)
{
    // 1. genero il nuovo nodo
    struct nodo* nuovo;
    nuovo      = malloc(sizeof(struct Nodo));
    nuovo->valore = val;
    // 2. dato che il nuovo nodo diventerà la nuova testa,
    // il successore sarà la testa attuale
    nuovo->prossimo = *start;
    // 3. inserisco il nuovo nodo come primo nodo
    *start      = nuovo;
}
```

Implementazione di liste concatenate

Inserimento di un Nodo in testa

```
void insert_at_the_beginning(struct nodo** start, int val)
{
    // 1. genero il nuovo nodo
    struct nodo* nuovo;
    nuovo      = malloc(sizeof(struct Nodo));
    nuovo->valore = val;
    // 2. dato che il nuovo nodo diventerà la nuova testa,
    // il successore sarà la testa attuale
    nuovo->prossimo = *start;
    // 3. inserisco il nuovo nodo come primo nodo
    *start      = nuovo;
}

int main()
{
    struct nodo* inizio = NULL;
    insert_at_the_beginning(&inizio, 42);
    insert_at_the_beginning(&inizio, 88);
    insert_at_the_beginning(&inizio, 90);
    return 0;
}
```

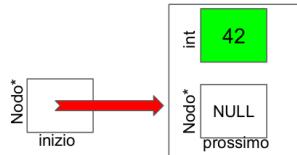


Implementazione di liste concatenate

Inserimento di un Nodo in testa

```
void insert_at_the_beginning(struct nodo** start, int val)
{
    // 1. genero il nuovo nodo
    struct nodo* nuovo;
    nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore = val;
    // 2. dato che che il nuovo nodo diventerà la nuova testa,
    // il successore sarà la testa attuale
    nuovo->prossimo = *start;
    // 3. inserisco il nuovo nodo come primo nodo
    *start = nuovo;
}
```

```
int main()
{
    struct nodo* inizio = NULL;
    insert_at_the_beginning(&inizio, 42);
    insert_at_the_beginning(&inizio, 88);
    insert_at_the_beginning(&inizio, 90);
    return 0;
}
```



Implementazione di liste concatenate

Inserimento di un Nodo in testa

```
void insert_at_the_beginning(struct nodo** start, int val)
{
    // 1. genero il nuovo nodo
    struct nodo* nuovo;
    nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore = val;
    // 2. dato che che il nuovo nodo diventerà la nuova testa,
    // il successore sarà la testa attuale
    nuovo->prossimo = *start;
    // 3. inserisco il nuovo nodo come primo nodo
    *start = nuovo;
}
```

```
int main()
{
    struct nodo* inizio = NULL;
    insert_at_the_beginning(&inizio, 42);
    insert_at_the_beginning(&inizio, 88);
    insert_at_the_beginning(&inizio, 90);
    return 0;
}
```

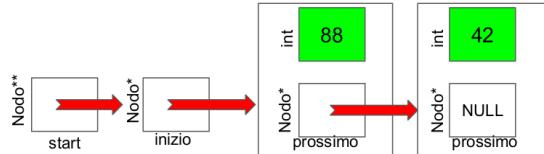


Implementazione di liste concatenate

Inserimento di un Nodo in testa

```
void insert_at_the_beginning(struct nodo** start, int val)
{
    // 1. genero il nuovo nodo
    struct nodo* nuovo;
    nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore = val;
    // 2. dato che che il nuovo nodo diventerà la nuova testa,
    // il successore sarà la testa attuale
    nuovo->prossimo = *start;
    // 3. inserisco il nuovo nodo come primo nodo
    *start = nuovo;
}
```

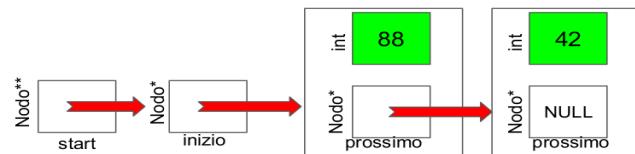
```
int main()
{
    struct nodo* inizio = NULL;
    insert_at_the_beginning(&inizio, 42);
    insert_at_the_beginning(&inizio, 88);
    ====>
    insert_at_the_beginning(&inizio, 90);
    return 0;
}
```



Implementazione di liste concatenate

Rimozione dell'Nodo in testa

```
void delete_from_the_beginning(struct nodo** start)
{
    if (*start == NULL)
        return;
    // 1. tengo traccia dell'Nodo da eliminare
    struct nodo* todel = *start;
    // 2. il nuovo primo nodo sarà l'attuale secondo nodo (se c'è)
    *start = (*start)->prossimo;
    // 3. deallocazione effettiva
    free(todel);
}
```



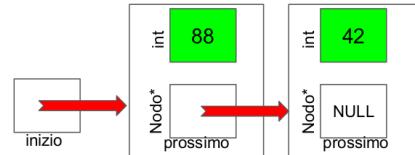
Implementazione di liste concatenate

Rimozione dell'Nodo in testa

```
void delete_from_the_beginning(struct nodo** start)
{
    if (*start == NULL)
        return;
    // 1. tengo traccia dell'Nodo da eliminare
    struct nodo* todel = *start;
    // 2. il nuovo primo nodo sarà l'attuale secondo nodo (se c'è)
    *start = (*start)->prossimo;
    // 3. deallocazione effettiva
    free(todel);
}
```

```
int main()
{
    struct nodo* inizio = NULL;
    insert_at_the_beginning(&inizio, 42);
    insert_at_the_beginning(&inizio, 88);

    ➔ delete_from_the_beginning(&inizio);
    delete_from_the_beginning(&inizio);
    return 0;
}
```



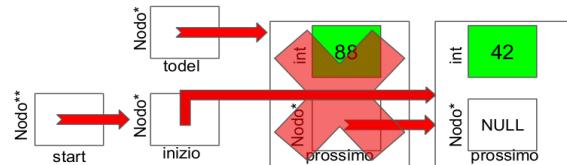
Implementazione di liste concatenate

Rimozione dell'Nodo in testa

```
void delete_from_the_beginning(struct nodo** start)
{
    if (*start == NULL)
        return;
    // 1. tengo traccia dell'Nodo da eliminare
    struct nodo* todel = *start;
    // 2. il nuovo primo nodo sarà l'attuale secondo nodo (se c'è)
    *start = (*start)->prossimo;
    // 3. deallocazione effettiva
    ➔ free(todel);
}
```

```
int main()
{
    struct nodo* inizio = NULL;
    insert_at_the_beginning(&inizio, 42);
    insert_at_the_beginning(&inizio, 88);

    delete_from_the_beginning(&inizio);
    delete_from_the_beginning(&inizio);
    return 0;
}
```

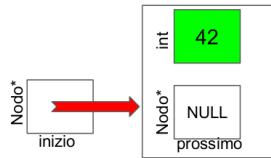


Implementazione di liste concatenate

Rimozione dell'Nodo in testa

```
void delete_from_the_beginning(struct nodo** start)
{
    if (*start == NULL)
        return;
    // 1. tengo traccia dell'Nodo da eliminare
    struct nodo* todel = *start;
    // 2. il nuovo primo nodo sarà l'attuale secondo nodo (se c'è)
    *start = (*start)->prossimo;
    // 3. deallocazione effettiva
    free(todel);
}

int main()
{
    struct nodo* inizio = NULL;
    insert_at_the_beginning(&inizio, 42);
    insert_at_the_beginning(&inizio, 88);
    delete_from_the_beginning(&inizio);
    delete_from_the_beginning(&inizio);
    return 0;
}
```



Implementazione di liste concatenate

Stampa tutti gli elementi presenti in lista

```
void print(struct nodo* start)
{
    if (start == NULL)
        return;

    struct nodo* in_esame = start;
    while(in_esame != NULL)
    {
        print("%d, ", in_esame->valore);
        in_esame = in_esame->prossimo;
    }
}
```

Implementazione di liste concatenate

Strategie alternative: inserimento in coda (V2)

```
void insert_at_the_end(struct Nodo** start, int val)
{
    // 1. genero il nuovo Nodo
    struct Nodo* nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;
    // 2. se la lista è vuota, inserisco il nuovo Nodo come primo Nodo
    if(*start == NULL)
    {
        *start = nuovo;
        return;
    }

    // 3. Se la lista non è vuota, scorri la fino a raggiungere l'ultimo Nodo

    struct Nodo* in_esame = *start;
    while(in_esame->prossimo!= NULL) // controllo il prossimo
        in_esame = in_esame->prossimo;
    // "in esame" punterà all'ultimo Nodo
    in_esame->prossimo = nuovo;
}
```

non è necessario un puntatore al precedente

Implementazione di liste concatenate

Strategie alternative: inserimento in coda (V3)

```
struct Nodo* insert_at_the_end(struct Nodo* start, int val)
{
    // 1. genero il nuovo Nodo
    struct Nodo* nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;

    // 2. se la lista è vuota, restituisco il nuovo Nodo
    if(start == NULL)
        return nuovo;

    // 3. se la lista non è vuota, scorri la fino a raggiungere l'ultimo Nodo
    struct Nodo* in_esame = start;
    while( in_esame->prossimo!= NULL)
        in_esame = in_esame->prossimo; // "in esame" punterà all'ultimo Nodo
    // della lista

    in_esame->prossimo = nuovo;

    return start;
}
```

non è necessario un puntatore a puntatore (al prezzo di un return da gestire)

Implementazione di liste concatenate

Strategie alternative: inserimento in coda (V4)

```
void insert_at_the_end(struct Nodo** start, int val)
{
    // 1. genero il nuovo Nodo
    struct Nodo* nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;

    // 2. cerco dove inserirlo

    // in questa versione, il puntatore "p_dest" punta non ad un nodo della lista,
    // ma ad un puntatore ad un nodo
    struct Nodo** p_dest = start;
    // se la lista non è vuota,
    // scorri la fino a raggiungere l'ultimo Nodo
    while( *p_dest!=NULL)
        p_dest = & ((*p_dest)->prossimo);
    // se la lista è vuota, "in_esame" punterà al puntatore che dovrà
    // contenere il primo Nodo (ossia "start_lista").
    // Altrimenti, punterà al puntatore "prossimo" dell'ultimo Nodo della lista
    *p_dest = nuovo;
}
```

utilizzare un **puntatore a puntatore**
che dica *in quale puntatore* prossimo
(e non la struttura) inserire l'indirizzo
del nuovo Nodo allocato.

Implementazione di liste concatenate

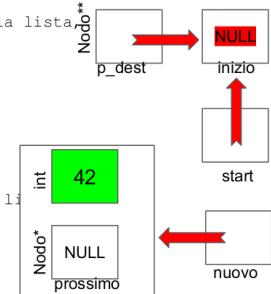
Strategie alternative: inserimento in coda (V4)

```
void insert_at_the_end(struct Nodo** start, int val)
{
    // 1. genero il nuovo Nodo
    struct Nodo* nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;

    // 2. cerco dove inserirlo

    // in questa versione, il puntatore "p_dest" punta non ad un nodo della lista,
    // ma ad un puntatore ad un nodo
    struct Nodo** p_dest = start;
    // se la lista non è vuota,
    // scorri la fino a raggiungere l'ultimo Nodo
    while( *p_dest!=NULL)
        p_dest = & ((*p_dest)->prossimo);
    // se la lista è vuota, "in_esame" punterà al puntatore che dovrà
    // contenere il primo Nodo (ossia "start_lista").
    // Altrimenti, punterà al puntatore "prossimo" dell'ultimo Nodo della lista
    *p_dest = nuovo;
}
```

```
int main()
{
    struct Nodo* inizio = NULL;
    insert_at_the_end(&inizio, 42);
}
```



Implementazione di liste concatenate

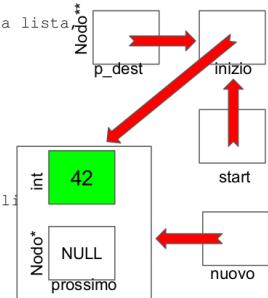
Strategie alternative: inserimento in coda (V4)

```
void insert_at_the_end(struct Nodo** start, int val)
{
    // 1. genero il nuovo Nodo
    struct Nodo* nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;

    // 2. cerco dove inserirlo

    // in questa versione, il puntatore "p_dest" punta non ad un nodo della lista
    // ma ad un puntatore ad un nodo
    struct Nodo** p_dest = start;
    // se la lista non è vuota,
    // scorri la lista fino a raggiungere l'ultimo Nodo
    while( *p_dest != NULL)
        p_dest = & ((*p_dest)->prossimo);
    // se la lista è vuota, "in_esame" punterà al puntatore che dovrà
    // contenere il primo Nodo (ossia "start_lista").
    // Altrimenti, punterà al puntatore "prossimo" dell'ultimo Nodo della lista
    => *p_dest = nuovo;
}
```

```
int main()
{
    struct Nodo* inizio = NULL;
    insert_at_the_end(&inizio, 42);
}
```



Implementazione di liste concatenate

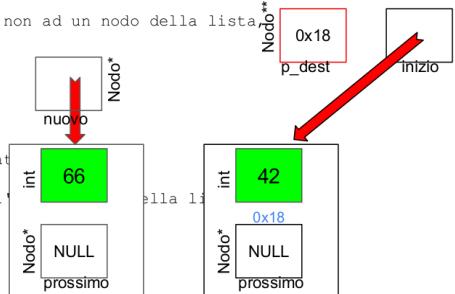
Strategie alternative: inserimento in coda (V4)

```
void insert_at_the_end(struct Nodo** start, int val)
{
    // 1. genero il nuovo Nodo
    struct Nodo* nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;

    // 2. cerco dove inserirlo

    // in questa versione, il puntatore "p_dest" punta non ad un nodo della lista
    // ma ad un puntatore ad un nodo
    struct Nodo** p_dest = start;
    // se la lista non è vuota,
    // scorri la lista fino a raggiungere l'ultimo Nodo
    while( *p_dest != NULL)
        p_dest = & ((*p_dest)->prossimo);
    // se la lista è vuota, "in_esame" punterà al puntatore che dovrà
    // contenere il primo Nodo (ossia "start_lista").
    // Altrimenti, punterà al puntatore "prossimo" dell'ultimo Nodo della lista
    => *p_dest = nuovo;
}
```

```
int main()
{
    struct Nodo* inizio = NULL;
    insert_at_the_end(&inizio, 42);
    insert_at_the_end(&inizio, 66);
}
```



Implementazione di liste concatenate

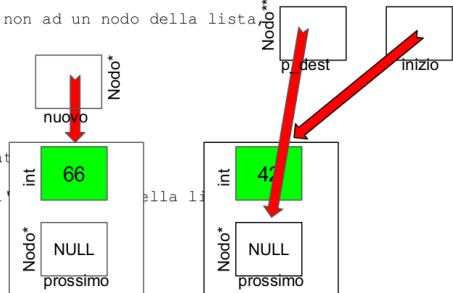
Strategie alternative: inserimento in coda (V4)

```
void insert_at_the_end(struct Nodo** start, int val)
{
    // 1. genero il nuovo Nodo
    struct Nodo* nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;

    // 2. cerco dove inserirlo

    // in questa versione, il puntatore "p_dest" punta non ad un nodo della lista
    // ma ad un puntatore ad un nodo
    struct Nodo** p_dest = start;
    // se la lista non è vuota,
    // scorrila fino a raggiungere l'ultimo Nodo
    while( *p_dest!=NULL)
        p_dest = & ((*p_dest)->prossimo);
    // se la lista è vuota, "in_esame" punterà al puntatore
    // contenere il primo Nodo (ossia "start_lista").
    // Altrimenti, punterà al puntatore "prossimo" dell'ultimo
    *p_dest = nuovo;
}
```

```
int main()
{
    struct nodo* inizio = NULL;
    insert_at_the_end(&inizio, 42);
    insert_at_the_end(&inizio, 66);
}
```



Implementazione di liste concatenate

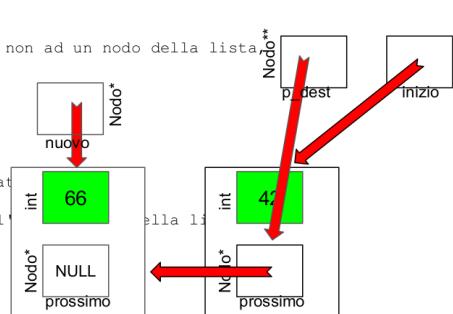
Strategie alternative: inserimento in coda (V4)

```
void insert_at_the_end(struct Nodo** start, int val)
{
    // 1. genero il nuovo Nodo
    struct Nodo* nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;

    // 2. cerco dove inserirlo

    // in questa versione, il puntatore "p_dest" punta non ad un nodo della lista
    // ma ad un puntatore ad un nodo
    struct Nodo** p_dest = start;
    // se la lista non è vuota,
    // scorrila fino a raggiungere l'ultimo Nodo
    while( *p_dest!=NULL)
        p_dest = & ((*p_dest)->prossimo);
    // se la lista è vuota, "in_esame" punterà al puntatore
    // contenere il primo Nodo (ossia "start_lista").
    // Altrimenti, punterà al puntatore "prossimo" dell'ultimo
    *p_dest = nuovo;
}
```

```
int main()
{
    struct nodo* inizio = NULL;
    insert_at_the_end(&inizio, 42);
    insert_at_the_end(&inizio, 66);
}
```



Implementazione di liste concatenate

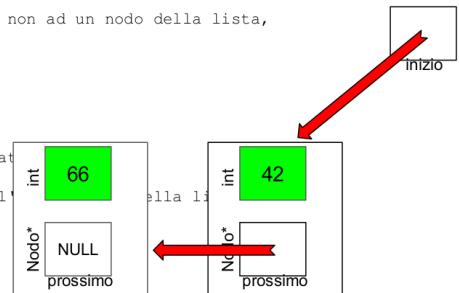
Strategie alternative: inserimento in coda (V4)

```
void insert_at_the_end(struct Nodo** start, int val)
{
    // 1. genero il nuovo Nodo
    struct Nodo* nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;

    // 2. cerco dove inserirlo

    // in questa versione, il puntatore "p_dest" punta non ad un nodo della lista,
    // ma ad un puntatore ad un nodo
    struct Nodo** p_dest = start;
    // se la lista non è vuota,
    // scorrila fino a raggiungere l'ultimo Nodo
    while( *p_dest!=NULL)
        p_dest = & ((*p_dest)->prossimo);
    // se la lista è vuota, "in_esame" punterà al puntatore
    // contenere il primo Nodo (ossia "start_lista").
    // Altrimenti, punterà al puntatore "prossimo" dell'ultimo
    *p_dest = nuovo;
}

➡
```



Implementazione di liste concatenate

Strategie alternative: lista con puntatore encapsulato

```
struct Lista
{
    struct Nodo* inizio;
};

int main()
{
    struct Lista l;
    ➡ l.inizio = NULL;
    .
    .
}
```



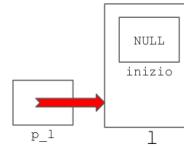
Implementazione di liste concatenate

Strategie alternative: lista con puntatore encapsulato

```
struct Lista
{
    struct Nodo* inizio;
};

void insert_at_the_end(Lista* p_l, int val)
{
    // 1. genero il nuovo Nodo
    struct Nodo* nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore = val;
    nuovo->prossimo = NULL;
    if(p_l->inizio == NULL)
    {
        p_l->inizio = nuovo;
        return;
    }
    // 2. scorro
    struct nodo* in_esame = *start;
    while(in_esame->prossimo != NULL)
    {
        in_esame = in_esame->prossimo;
    }
    // 3. inserisco
    in_esame->prossimo = nuovo;
}
```

```
int main()
{
    struct Lista l;
    l.inizio = NULL;
    ➔ insert_at_the_beginning(&l, 42);
    .
}
```



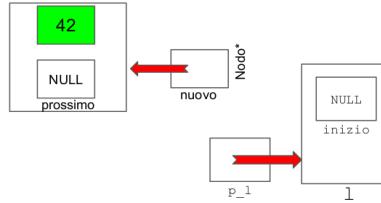
Implementazione di liste concatenate

Strategie alternative: lista con puntatore encapsulato

```
struct Lista
{
    struct Nodo* inizio;
};

void insert_at_the_end(Lista* p_l, int val)
{
    // 1. genero il nuovo Nodo
    struct Nodo* nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore = val;
    ➔ nuovo->prossimo = NULL;
    if(p_l->inizio == NULL)
    {
        p_l->inizio = nuovo;
        return;
    }
    // 2. scorro
    struct nodo* in_esame = *start;
    while(in_esame->prossimo != NULL)
    {
        in_esame = in_esame->prossimo;
    }
    // 3. inserisco
    in_esame->prossimo = nuovo;
}
```

```
int main()
{
    struct Lista l;
    l.inizio = NULL;
    insert_at_the_beginning(&l, 42);
    .
}
```



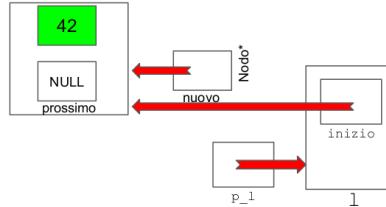
Implementazione di liste concatenate

Strategie alternative: lista con puntatore encapsulato

```
struct Lista
{
    struct Nodo* inizio;
};

void insert_at_the_end(Lista* p_l, int val)
{
    // 1. genero il nuovo Nodo
    struct Nodo* nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;
    if(p_l->inizio == NULL)
    {
        p_l->inizio = nuovo;
        return;
    }
    // 2. scorro
    struct nodo* in_esame = *start;
    while(in_esame->prossimo != NULL)
    {
        in_esame = in_esame->prossimo;
    }
    // 3. inserisco
    in_esame->prossimo = nuovo;
}
```

```
int main()
{
    struct Lista l;
    l.inizio = NULL;
    insert_at_the_beginning(&l, 42);
    .
}
```



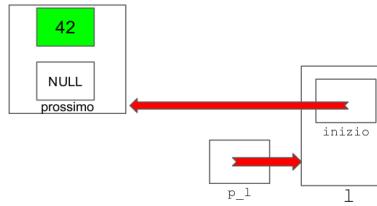
Implementazione di liste concatenate

Strategie alternative: lista con puntatore encapsulato

```
struct Lista
{
    struct Nodo* inizio;
};

void insert_at_the_end(Lista* p_l, int val)
{
    // 1. genero il nuovo Nodo
    struct Nodo* nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;
    if(p_l->inizio == NULL)
    {
        p_l->inizio = nuovo;
        return;
    }
    // 2. scorro
    struct nodo* in_esame = *start;
    while(in_esame->prossimo != NULL)
    {
        in_esame = in_esame->prossimo;
    }
    // 3. inserisco
    in_esame->prossimo = nuovo;
}
```

```
int main()
{
    struct Lista l;
    l.inizio = NULL;
    insert_at_the_beginning(&l, 42);
    .
}
```



Implementazione di liste concatenate

Strategie alternative: lista con puntatori all'inizio e alla fine

```
struct Lista
{
    struct Nodo* inizio;
    struct Nodo* fine;
};

void insert_at_the_end(Lista* p_l, int val)
{
    // 1. genero il nuovo Nodo
    struct Nodo* nuovo = malloc(sizeof(struct Nodo));
    nuovo->valore      = val;
    nuovo->prossimo     = NULL;
    if(p_l->inizio == NULL)
    {
        p_l->inizio = nuovo;
        p_l->fine   = nuovo;
        return;
    }
    p_l->fine->prossimo = nuovo;
    p_l->fine           = nuovo;
}

int main()
{
    struct Lista l;
    l.inizio = NULL;
    l.fine   = NULL;
    insert_at_the_beginning(&l, 42);
    .
    .
}
```

- immediato l'inserimento in testa o in coda
- 2 puntatori da gestire



Liste concatenate

Quando?

Appropriate quando:

- non è possibile determinare a monte il numero di elementi della struttura dati
 - la lunghezza di una LC può aumentare o diminuire in base alle necessità grazie all'assenza di un obbligo di contiguità in memoria degli elementi che la compongono
- si vogliono inserire/rimuovere elementi all'interno della struttura dati senza spostare tutti gli altri elementi
 - è possibile inserire/rimuovere elementi in/da posizioni arbitrarie della lista senza rischiare di lasciare locazioni di memoria inutilizzate o di dover spostare tutta la struttura dati

Non appropriate quando:

- i tempi di accessi contano
 - a differenza di altre strutture dati come gli array, le liste concatenate necessitano di tempo per essere navigate, in quanto per accedere ad un dato Nodo devo accedere a tutti i precedenti. Ad esempio, per leggere il contenuto della 50-esima posizione di una lista semplicemente concatenata, sarà necessario per forza scorrere sequenzialmente i primi 49 elementi, a differenza di un array in cui l'accesso in memoria alla 50-esima posizione è diretto.