Strutture dati dinamiche in C++

Puntatori e Liste

Giulio Angiani
I.I.S. "Blaise Pascal" - Reggio Emilia

Puntatori e Liste

Cos'è una "struttura dati dinamica" ?

Una struttura dati è **dinamica** se è progettata per modificare la sua dimensione (e quindi la sua capacità di memoria) a tempo di esecuzione

in **C++** esistono già molte strutture dinamiche predefinite[1] ma possono anche essere costruite dal programmatore

Le strutture dinamiche **collegate** sono tipicamente implementate tramite due strumenti:

- il **nodo** che contiene le informazioni
- il **puntatore** che permette il collegamento fra i nodi e consente di manipolare la struttura dinamica

Le operazioni tipiche sono le solite [strutture CRUD]

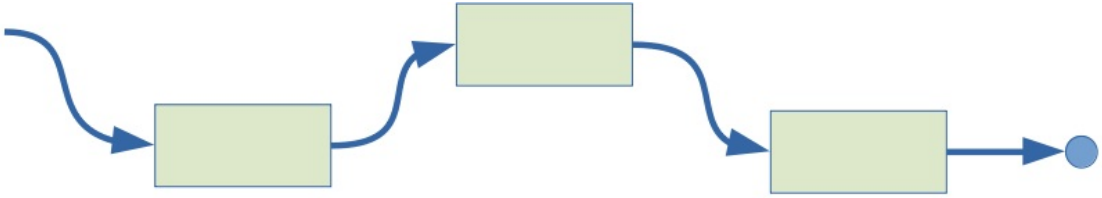
- Creazione (Create)
- Lettura (Read)
- Modifica (Update)
- Cancellazione (Delete)

1) rif: STL library

Nodi e Puntatori - Strutture collegate

Una struttura dinamica collegata è solitamente rappresentata come in figura e funziona un po' come una caccia al tesoro

In ogni nodo ci sono le informazioni per cercare il nodo successivo...

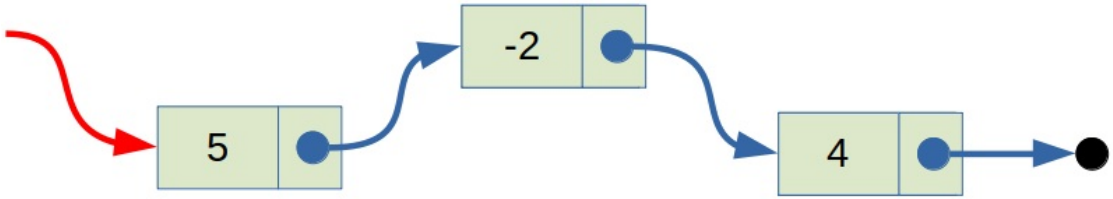


dove

- le frecce indicano un puntatore
- i rettangoli indicano un nodo

Nodi e Puntatori - Strutture collegate

Esiste sempre un puntatore **iniziale** (colorato in rosso) che permette di accedere alla struttura dati



Ogni nodo contiene due dati:

- il valore dell'informazione contenuta (nel caso indicato un numero intero)
- un puntatore al nodo successivo (per mantenere il collegamento della struttura)

NOTA:

- un puntatore non è altro che un **indirizzo di memoria** che indica dove è memorizzata una informazione
- un nodo è una **struct** che può contenere vari campi dati (nel caso indicato solo uno) e un campo puntatore dove è memorizzata la locazione di memoria del successivo nodo
- nel nodo terminale (l'ultimo della serie) il nodo puntatore ha valore **NULL** (o nullptr)

dove

Nodi e Puntatori in C++

L'implementazione in C++ dei puntatori utilizza la sintassi con *

Esempio:

C++

```cpp
int a = 3;      // a è una variabile intera
int* p = &a;    // p è un puntatore alla variabile a ottenuto tramite l'operatore di dereferenziazione &
cout << "valore di a    : " << a << endl;   // stampo il valore di a
cout << "indirizzo di a : " << p << endl;   // stampo il valore di p
```

OUTPUT

```
valore di a    : 3
indirizzo di a : 0x7ffee9f0b99c
```

Tipologie di Strutture dinamiche collegate

In funzione della topologia di collegamento e del funzionamento si possono individuare vari tipi di strutture dinamiche. Le più note sono:

- LISTE
  - semplici
  - doppie
  - concatenate
- PILE
- CODE
- ALBERI
  - binari
  - ennari
- GRAFI
  - orientati
  - non orientati
  - ciclici
  - aciclici
- TAVOLE HASH

Le Liste

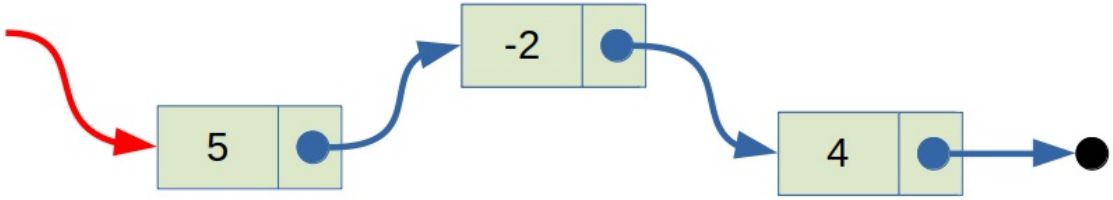
E' la struttura collegata più semplice dal punto di vista topologico.

Può essere costituita da **0 fino a N** elementi.

E' caratterizzata da un puntatore **iniziale**.

- Se la lista è **vuota** (ovvero non contiene elementi), il puntatore iniziale vale NULL (o nullptr);
- Se la lista ha almeno un elemento:
  - il puntatore iniziale ha il valore della locazione di memoria del primo **nodo** memorizzato in RAM.
  - ogni nodo contiene uno o più campi **informazione** e un puntatore al nodo successivo.
  - il nodo finale ha il puntatore al nodo successivo che vale NULL (o nullptr);

In figura una lista semplice



Gestire una lista in C++

La prima cosa da fare è definire le **strutture dati** necessario per nodo e puntatore Nel caso di esempio costruiremo una lista di interi

C++
```
struct nodo {
    int informazione;   // campo che contiene il dato
    nodo* next;         // campo che contiene il puntatore(riferimento) al nodo successivo
};
```

Per creare una lista è necessario avere un punto di partenza. Si definisce sempre un puntatore iniziale (che ha il tipo nodo*) e si inizializza a NULL (o nullptr)

C++
```
nodo* puntatoreiniziale = nullptr;
```

Con queste operazioni abbiamo realizzato una lista vuota come in figura

Aggiungere elementi ad una lista

L'aggiunta di elementi ad una lista prevede una serie di operazioni L'inserimento può essere fatto in vari modi

- in testa
- in coda
- in mezzo
- in ordine

Procediamo all'inserimento **in testa** che è il più semplice

Le operazione da compiere sono:

- definire un altro puntatore di tipo nodo* (nell'esempio lo chiameremo **nuovo**)
- allocare la memoria necessaria per contenere un nodo (tramite la funzione predefinita **new**)
- collegare il puntatore del nuovo nodo al primo nodo della lista (a NULL se è vuota)
- collegare il puntatore iniziale al nuovo nodo
- rimuovere il nuovo puntatore (perché non serve più a niente)

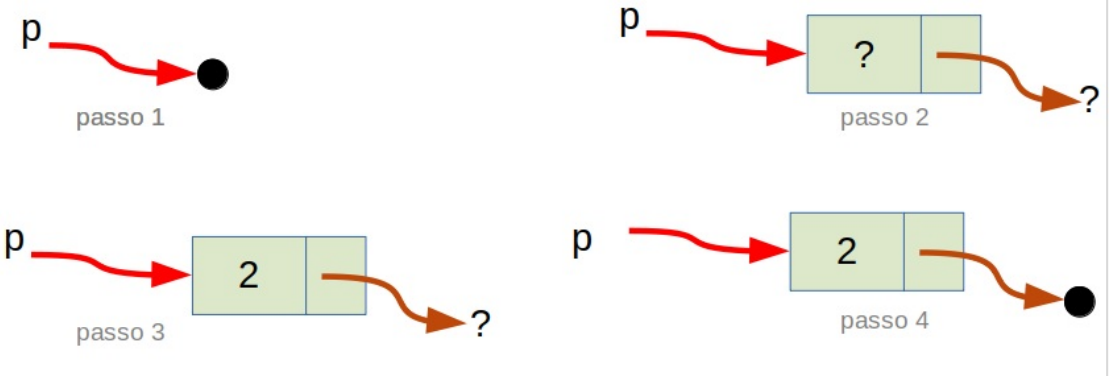
Inserire il primo elemento

Se la lista è vuota l'operazione di inserimento va fatta direttamente sul puntatore iniziale altrimenti si userà un altro puntatore temporaneo

L'inserimento del primo elemento consta di questi passi

C++

```
puntatoreiniziale = nullptr;        // passo 1 - il puntatore iniziale vale NULL => la lista è vuota
puntatoreiniziale = new nodo;       // passo 2 - allocazione in memoria dello spazio per un nuovo nodo
                                    //           questa operazione collega il puntatore al nuovo nodo
puntatoreiniziale->informazione = 2 // passo 3 - assegno il valore 2 al campo informazione del nuovo nodo
puntatoreiniziale->next = NULL      // passo 4 - collego il puntatore del nuovo nodo a NULL
```



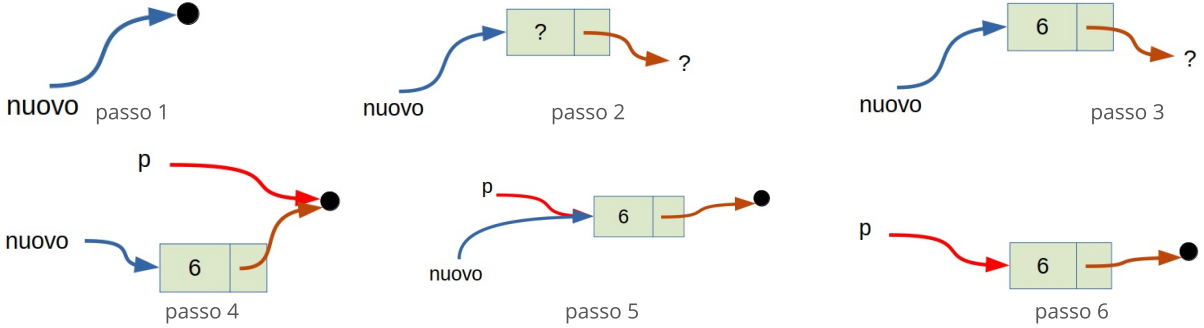
Aggiungere altri elementi ad una lista (in testa)

La definizione di un nuovo puntatore consiste nel definire un'altra variabile di tipo nodo*

C++

```
nodo* nuovo = nullptr;  // passo 1 - definizione e inizializzazione nuovo puntatore
nuovo = new nodo;       // passo 2 - allocazione in memoria dello spazio per un nuovo nodo
                        //           questa operazione collega il puntatore al nuovo nodo
nuovo->informazione = 6 // passo 3 - assegno il valore 6 al campo informazione del nuovo nodo
nuovo->next = puntatoreiniziale // passo 4 - collego il puntatore del nuovo nodo al nodo iniziale
                                //           o a NULL se è il primo nodo della lista
puntatoreiniziale = nuovo;      // passo 5 - collego il puntatore iniziale al nuovo nodo
nuovo = NULL;           // passo 6 - cancello il puntatore che ho usato
delete nuovo;           //           per inserire un nuovo nodo perché non più necessario
```

Di seguito i 6 passi indicati dalle figure



Leggere gli elementi di una lista

La lettugra è una delle operazioni base su ogni struttura dati Per leggere si parte dal puntatore iniziale e si seguono i collegamenti finché non si giunge al valore NULL

di seguito una semplice **void function** che riceve un puntatore e legge il contenuto della lista

C++

```
void leggilista(nodo* p) {
    while (p != NULL) {
        cout << p->informazione << ";";
        p = p->next;    // posso modificare p perché è una copia del parametro attuale
    }
}

[...]

leggilista(puntatoreiniziale);
```

OUTPUT

```
6;2;
```

Avendo inserito sempre in testa alla lista l'output mostra i valori in senso inverso

Inserimento in testa

Qui una procedura per inserire in testa ad una lista un elemento

C++

```
void intesta(nodo* &p, int v) {     // NOTA l'operatore & perchè l'inserimento
                                    // in testa DEVE MODIFICARE il puntatore iniziale passato
    nodo* nuovo = nullptr;
    nuovo = new nodo;
    nuovo->informazione = v;
    nuovo->next = puntatoreiniziale;
    p = nuovo;
}

[...]

intesta(puntatoreiniziale, 4);
intesta(puntatoreiniziale, 5);
intesta(puntatoreiniziale, 6);
leggilista(puntatoreiniziale);
```

OUTPUT

```
6;5;4;
```

Inserimento in coda

Più complesso l'inserimento in coda in quanto bisogna prima cercare l'ultimo elemento e poi agire di conseguenza

Dovendo agire sull'ultimo nodo della lista

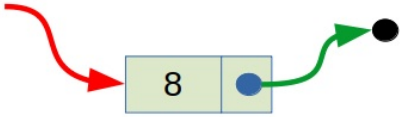
3 casi

- la lista è vuota
- la lista ha un solo nodo
- la lista ha più nodi

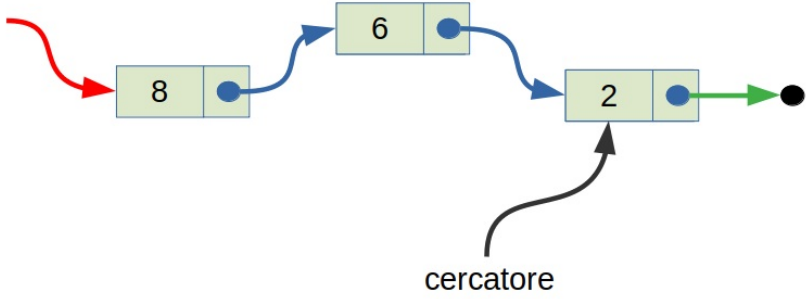
Nel caso 1 (lista vuota) il problema è lo stesso dell'inserimento del primo elemento già visto precedentemente

Inserimento in coda

Nel caso 2 (lista con un solo nodo) possiamo agire direttamente sul puntatore iniziale per raggiungere il puntatore verde dell'ultimo nodo.(fig.1)



Nel caso 3 (lista con un più nodi) scorriamo la lista con un puntatore di appoggio (cercatore) per raggiungere l'ultimo nodo e da qui agire sul puntatore verde dell'ultimo nodo (fig 2)



Inserimento in coda in C++

La funzione sottostante implementa l'inserimento in coda

C++

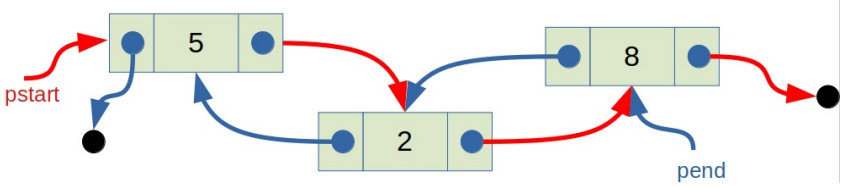
```
void incoda(nodo* &p, int v) {          // l'operatore & perchè l'inserimento
                                        // in testa DEVE MODIFICARE il puntatore iniziale passato
    if (p == NULL) {                    // caso 1 - lista vuota => uso la funzione per inserimento in testa
        intesta(p, v);
    }
    else if (p->next == NULL){          // caso 2 - un solo elemento (p->next == NULL)
        nodo* nuovo = new nodo;         // alloco un nuovo nodo
        nuovo->informazione = v;        // inserisco il valore
        nuovo->next = NULL;             // metto a NULL il puntatore del nuovo nodo
        p->next = nuovo;                // collego il puntatore iniziale al nuovo nodo
    }
    else {                              // caso 2 - più di un elemento
        nodo* cercatore = p;            // parto dal puntatore iniziale con un nuovo puntatore
        while (cercatore->next != NULL) // lo incremento finché il suo successivo non punta a NULL
           cercatore = cercatore->next; //   ovvero ho trovato l'ultimo nodo
        nodo* nuovo = new nodo;         // alloco un nuovo nodo
        nuovo->informazione = v;        // inserisco il valore
        nuovo->next = NULL;             // metto a NULL il puntatore del nuovo nodo
        cercatore->next = nuovo;        // collego il puntatore dell'ultimo nodo al nuovo nodo
    }
}
```

Liste bidirezionali

Si definisce bidirezionale una lista dove ogni nodo è collegato sia al nodo successivo che al precedente.

La struttura del nodo cambia perché si ha necessità di due puntatori per ogni elemento, uno che punti al successivo(come nel caso della lista semplice) e uno che punti al successivo.

Di solito, nelle liste bidirezionali, esiste un secondo punto di accesso alla lista che è il puntatore all'ultimo elemento (pend nell'immagine) e non solo quello che punta al primo (pstart nell'immagine).



Cambiamo quindi la struttura del nodo che diventa

C++

```
struct nodo {
    int informazione;   // campo che contiene il dato
    nodo* next;         // campo che contiene il puntatore(riferimento) al nodo successivo
    nodo* prev;         // campo che contiene il puntatore(riferimento) al nodo precedente
};
```

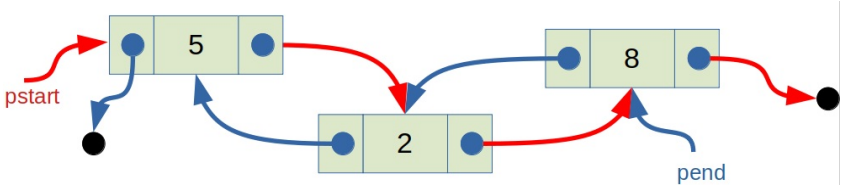
Liste bidirezionali

Vantaggi:

- è una struttura molto comoda per memorizzare informazioni in maniera ordinata.
- permette di muoversi nella struttura in maniera semplice (soprattutto se bisogna cercare un elemento precedente ad un altro)

Svantaggi:

- complessità maggiore nell'implementazione delle funzioni CRUD
- aumento di memoria



A titolo esemplificativo mostriamo la procedura di inserimento in testa ad una lista bidirezionale.

Liste bidirezionali

La funzione sottostante implementa l'inserimento in testa

C++

```
void intesta(nodo* &p, nodo* &l, int v) {
    if (p == NULL) {    // lista vuota : pstart e pend coincidono con NULL
        nodo* nuovo = new nodo;
        nuovo->informazione = v;
        nuovo->next = NULL;
        nuovo->prev = NULL;
        p = nuovo;
        l = nuovo;
    }
    else {
        // elementi in lista
        nodo* nuovo = new nodo;
        nuovo->informazione = v;
        nuovo->next = p;
        nuovo->prev = NULL;
        p->prev = nuovo;
        p = nuovo;
    }
}
```

Giulio Angiani
I.I.S. "Blaise Pascal" - Reggio Emilia