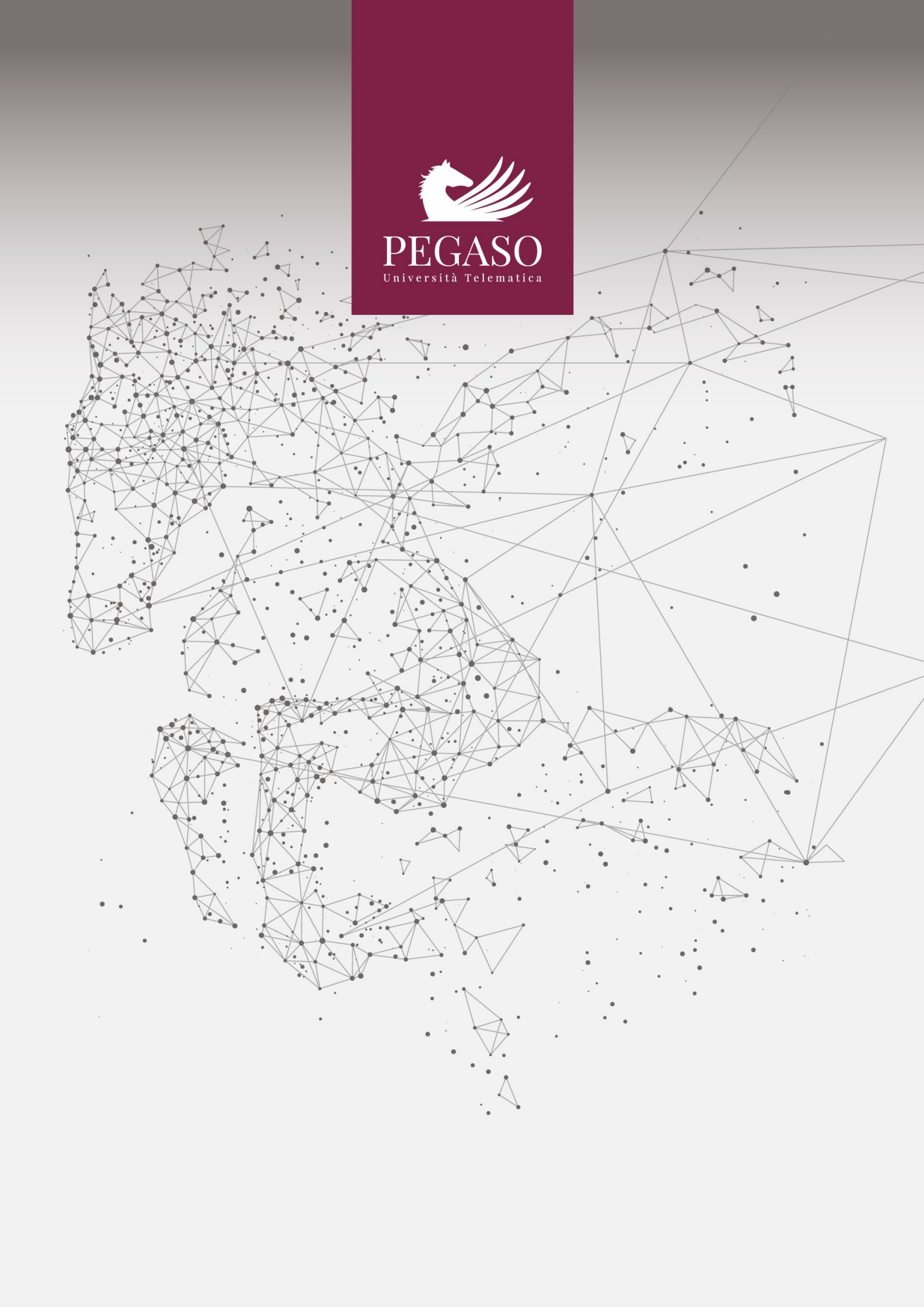
PEGASO
Università Telematica

Indice

1. Introduzione

Una struttura dati fornisce una rappresentazione organizzata e logica dei dati all'interno di un insieme.

Tra i tipi di strutture dati più comuni vi sono le **liste**: una collezione ordinata di elementi, di qualsiasi tipo, accessibili tramite un indice (le liste possono essere singolarmente collegate o a doppia).

Rispetto alla tassonomia classica, una lista risulta:

- una struttura **dinamica**: pensata per aggiungere o togliere elementi durante l'esecuzione di un algoritmo;
- una struttura **sparsa**: non si possono cioè fare ipotesi sulla posizione fisica degli elementi in memoria (o **lineare**: è accessibili solo attraverso gli estremi della sequenza);
- una struttura **ordinata**: tipicamente è una struttura in cui gli elementi sono disposti in maniera dipendente dal valore delle chiavi;

Ricordiamo il tipo di operazioni che è possibile effettuare:

- **interrogazioni** (query): restituiscono semplicemente informazioni sull'insieme;
- operazioni di **modifica**: cambiano l'insieme.

Nel dettaglio:

- **SEARCH(S, k)**: una query che, dato un insieme S e un valore chiave k, restituisce un puntatore x a un elemento di S tale che chiave[x] = k oppure NIL se un elemento così non appartiene a S.
- **MINIMUM(S)**: una query su un insieme totalmente ordinato S che restituisce un puntatore all'elemento di S con la chiave più piccola.
- **MAXIMUM(S)**: una query su un insieme totalmente ordinato S che restituisce un puntatore all'elemento di S con la chiave più grande.
- **SUCCESSOR(S, x)**: una query che, dato un elemento x la cui chiave appartiene a un insieme totalmente ordinato S, restituisce un puntatore al prossimo elemento più grande di S oppure NIL se x è l'elemento massimo.
- **PREDECESSOR(S, x)**: una query che, dato un elemento x la cui chiave appartiene a un insieme totalmente ordinato S, restituisce un puntatore al prossimo elemento più piccolo di S oppure NIL se x è l'elemento minimo
- **INSERT(S, x)**: un'operazione di modifica che inserisce nell'insieme S l'elemento puntato da x

- **DELETE(S, x)**: un'operazione di modifica che, dato un puntatore x a un elemento dell'insieme S, rimuove x da S (notate che questa operazione usa un puntatore a un elemento x, non un valore chiave).

Nel contesto delle liste, è importante la struttura "puntatore": il puntatore è un concetto importante in relazione alle strutture dati in quanto consente di creare strutture dati dinamiche.

2. Lista

Si dice lista una tripla $L = (E; t; S)$ dove E è un insieme di elementi, $t \in E$ è detto testa ed S è una relazione binaria su E, cioè $S \subseteq E \times E$ che soddisfa le seguenti proprietà:

- Per ogni $e \in E, (e,t) \notin S$
- Per ogni $e \in E$, se $e \neq t$ allora esiste uno ed un solo $e' \in E$ tale che $(e',e) \in S$.
- Per ogni $e \in E$, esiste al più un solo $e' \in E$ tale che $(e',e) \in S$.
- Per ogni $e \in E$, se $e \neq t$ allora e è raggiungibile da t, cioè esistono $e'_1, \dots, e'_k \in E$ con $k \geq 2$ tali che $e'_1 = t, (e'_i, e'_{i+1}) \in S$ per ogni $1 \leq i \leq k-1$ ed $e'_k = e$

Una lista $L = (E; t; S)$ è detta ordinata se le chiavi contenute nei suoi elementi sono disposte in modo tale da soddisfare una relazione d'ordine totale: per ogni $e_1, e_2 \in S$ allora la chiave di e_1 precede quella di e_2 nella relazione d'ordine totale.

Una lista viene rappresentata come una struttura dati dinamica lineare, in cui ogni elemento contiene solo l'indirizzo dell'elemento successivo (lista singolarmente collegata) oppure anche l'indirizzo dell'elemento precedente (lista doppiamente collegata).

L'indirizzo dell'elemento successivo contenuto nell'ultimo elemento di una lista è indefinito, così come l'indirizzo dell'elemento precedente contenuto nel primo elemento di una lista doppiamente collegata. Fa eccezione il caso dell'implementazione circolare di una lista, nella quale l'ultimo elemento è collegato al primo elemento.

Gli elementi di una lista non sono necessariamente memorizzati in modo consecutivo; quindi, l'accesso ad un qualsiasi elemento avviene scorrendo tutti gli elementi che lo precedono. Questo accesso indiretto necessita dell'indirizzo del primo elemento della lista, detto testa, il quale è indefinito se e solo se la lista è vuota.

Riassumendo:

- La lista è essere una sequenza di elementi di un determinato tipo.
- È un multi-insieme: ci possono essere ripetizioni del medesimo elemento.
- È una sequenza (collezione ordinata): siamo in grado di individuare il primo elemento della sequenza, il secondo, ecc.

Definiamo i classici problemi che insistono sulle strutture dati:

- **Problema della visita**: data una lista, attraversare tutti i suoi elementi esattamente una volta.
- **Problema della ricerca**: dati una lista e un valore, stabilire se il valore è contenuto in un elemento della lista, riportando in caso affermativo l'indirizzo di tale elemento.
- **Problema dell'inserimento**: dati una lista e un valore, inserire (se possibile) nella posizione appropriata della lista un nuovo elemento in cui memorizzare il valore.
- **Problema della rimozione**: dati una lista e un valore, rimuovere (se esiste) l'elemento appropriato della lista che contiene il valore.

Come realizzare una lista?

La realizzazione come vettore (di N componenti) non è sufficiente se durante l'esecuzione di un programma ci si trovi a dover inserire l'(N+1)-esimo elemento (cosa che per una generica lista può avvenire). È dunque necessaria una rappresentazione collegata, mediante puntatori e allocazione dinamica di memoria.

Si memorizzano gli elementi associando a ognuno di essi l'informazione (riferimento) che permette di individuare la locazione in cui è inserito l'elemento successivo (rappresentazione collegata).

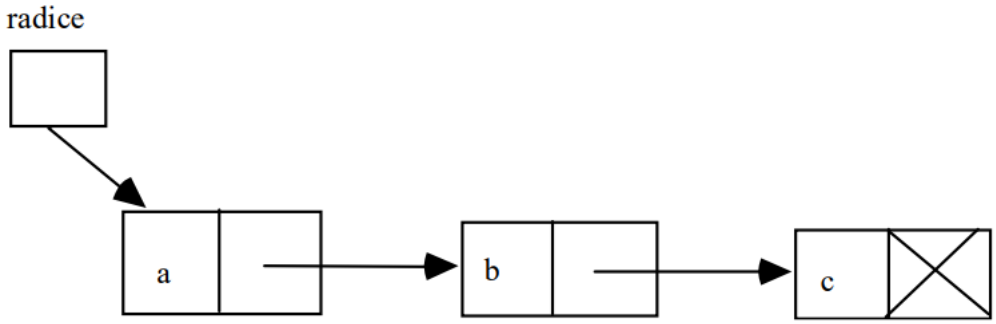

Figura 1 *- Rappresentazione collegata*

Se volessimo eliminare il primo elemento dovremmo operare nel seguente modo:

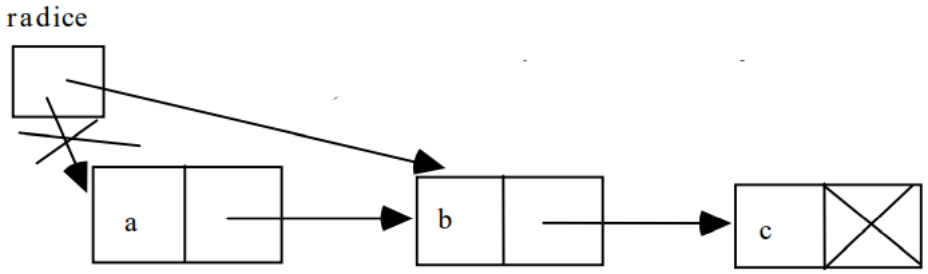

Figura 2 *- Eliminazione primo elemento*

I passi da seguire sono i seguenti:

- Identificazione del primo elemento della lista (radice / head).
- Eliminazione del collegamento tra il primo elemento e il secondo elemento della lista (solitamente tramite l'aggiornamento del puntatore del secondo elemento della lista in modo che punti al terzo elemento della lista).
- Eliminazione della memoria allocata per il primo elemento della lista (solitamente tramite l'utilizzo di delete per gestire la memoria in modo automatico).
- Aggiornamento del puntatore radice / head per puntare al secondo elemento della lista come nuovo primo elemento della lista.

Per l'inserimento in testa invece dovremmo operare nel seguente modo:

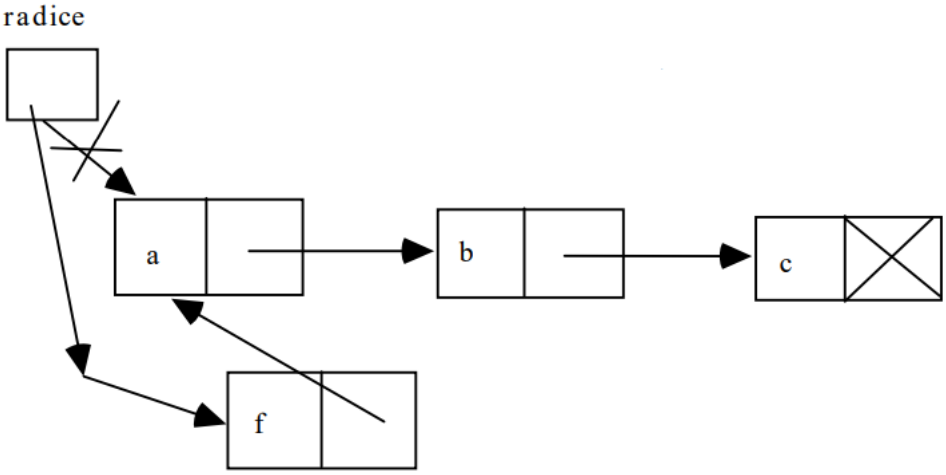

Figura 3 *- Inserimento in testa*

I passi da seguire sono i seguenti:

- Allocazione della memoria per il nuovo elemento da inserire nella lista (tramite l'utilizzo di new per gestire la memoria in modo automatico).
- Impostare i dati del nuovo elemento, solitamente tramite l'utilizzo di un costruttore o assegnando i valori delle proprietà dell'oggetto.
- Aggiornamento dei puntatori per collegare il nuovo elemento alla lista, solitamente tramite l'impostazione del puntatore del nuovo elemento che punta al primo elemento attuale della lista e l'aggiornamento del puntatore radice / head per puntare al nuovo elemento come nuovo primo elemento della lista.

Per l'eliminazione di un generico elemento dovremmo operare nel seguente modo:

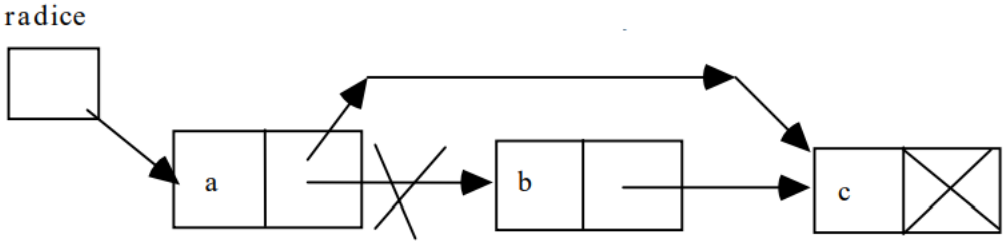

Figura 4 *- Eliminazione di un elemento*

I passi da seguire sono i seguenti:

- Individuazione dell'elemento da eliminare: questo può essere fatto tramite una ricerca sequenziale della lista, utilizzando un ciclo per scorrere gli elementi finché non si trova quello desiderato.
- Eliminazione del collegamento tra l'elemento precedente e quello successivo all'elemento da eliminare: questo può essere fatto tramite l'aggiornamento del puntatore dell'elemento precedente in modo che punti al successivo dell'elemento da eliminare.
- Eliminare la memoria allocata per l'elemento da eliminare (solitamente tramite l'utilizzo di delete per gestire la memoria in modo automatico).

È importante sottolineare che l'eliminazione di un generico elemento non in testa potrebbe richiedere un maggior tempo di esecuzione rispetto all'eliminazione del primo elemento, poiché potrebbe essere necessario scorrere tutta la lista per trovare l'elemento da eliminare.

Per l'inserimento di un elemento in un generico punto della lista dovremmo operare nel seguente modo:

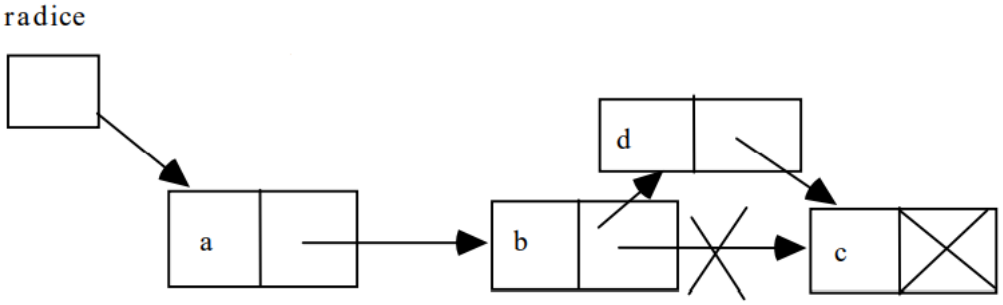

Figura 5 *- Inserimento di un elemento*

I passi da seguire sono i seguenti:

- Individuazione del punto di inserimento nella lista: questo può essere fatto tramite una ricerca sequenziale della lista, utilizzando un ciclo per scorrere gli elementi finché non si trova la posizione desiderata.
- Allocazione della memoria per il nuovo elemento da inserire nella lista (tramite l'utilizzo di new per gestire la memoria in modo automatico).
- Impostare i dati del nuovo elemento, solitamente tramite l'utilizzo di un costruttore o assegnando i valori delle proprietà dell'oggetto.
- Aggiornamento dei puntatori per collegare il nuovo elemento alla lista, solitamente tramite l'impostazione del puntatore del nuovo elemento che punta al successivo elemento dalla posizione di inserimento e l'impostazione del puntatore dell'elemento precedente alla posizione di inserimento che punta al nuovo elemento inserito.

3. Gestione della memoria dinamica

Quando si parla di gestione dinamica della memoria in C++, ci stiamo riferendo a due possibilità: **malloc()** e **new**.

L'operatore **new** utilizza l'overloading del costruttore per creare un nuovo oggetto e assegnare la memoria necessaria per esso; **malloc()** è una funzione della libreria C standard che assegna un blocco di memoria non inizializzato.

L'operatore **malloc()** è utilizzato per allocare un blocco di memoria di una determinata dimensione in bytes e restituisce un puntatore al primo byte del blocco allocato; la memoria allocata con **malloc()** deve essere liberata manualmente tramite la funzione **free()**.

L'operatore **new** in C++, al contrario di malloc, è un operatore che oltre ad allocare la memoria necessaria per un oggetto, chiama il costruttore dell'oggetto per inizializzare i suoi dati; la memoria allocata con **new** deve essere liberata utilizzando l'operatore **delete**.

L'utilizzo di **new** e **delete** (così come di **malloc()** e **free()**) è necessario perché la memoria allocata dinamicamente non viene automaticamente liberata quando un oggetto viene eliminato o quando il programma termina: se non si utilizza **delete** (o **free()**) per liberare la memoria allocata dinamicamente, si rischia di causare un memory leak, ovvero una parte del programma non rilascia la memoria che ha allocato, causando una perdita progressiva della memoria disponibile. Ciò può portare a una serie di problemi come l'esaurimento della memoria disponibile, rallentamenti del sistema, crash del programma o addirittura una situazione in cui il sistema operativo non riesce più a rispondere.

Riportiamo di seguito un esempio di codice in cui si alloca / dealloca memoria usando sia malloc() / free() che new / delete:

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main() {
    // Allocazione di memoria utilizzando malloc
    int* int_ptr_malloc = (int*) malloc(sizeof(int));
    *int_ptr_malloc = 5;
    cout<<"Valore puntato: "<<*int_ptr_malloc<<endl;
    free(int_ptr_malloc);

    // Allocazione di memoria utilizzando new
    int* int_ptr_new = new int;
    *int_ptr_new = 10;
    cout <<"Valore puntato: "<<*int_ptr_new<<endl;
    delete int_ptr_new;
```

```
        return 0;
    }
```

Quando si usa malloc() o new, viene richiesta la memoria all'Heap (gestore della memoria dinamica).

punt=(tipoelem *) malloc(sizeof(tipoelem));

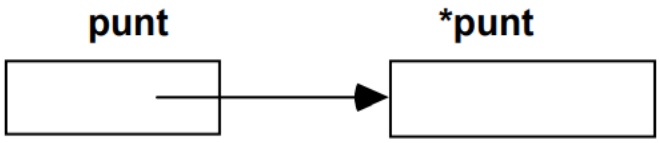

***Figura 6** - Allocazione*

Viene allocato lo spazio per la variabile *punt (la memoria allocata per *punt è prelevata dall'insieme delle celle libere dello heap)

free (punt);

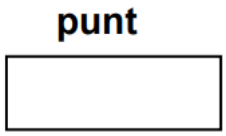

***Figura 7** - Deallocazione*

Dopo questa operazione *punt non esiste piu` (la memoria allocata per *punt è di nuovo inserita nell'insieme delle celle libere dello heap).

Bibliografia

- Alan Bertossi, Alberto Motresor: Algoritmi e strutture di dati, Città Studi Edizioni, terza edizione.
- C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione.
- Crescenzi, Gambosi, Grossi: Strutture di Dati e Algoritmi, Pearson/Addison-Wesley.
- Sedgewick: Algoritmi in C, Pearson, 2015.
- Cormen Leiserson Rivest Stein-Introduzione Agli Algoritmi E Strutture Dati-Prima Edizione.