

Puntatori

In linguaggi dove è possibile conoscere ed esprimere l'indirizzo della cella di memoria in cui è memorizzata una variabile è anche possibile eseguire delle modifiche sui valori degli indirizzi. In molti linguaggi c'è un forte legame tra puntatori ed array, proprio per questo è necessario incrementare e decrementare i valori dei puntatori.

Aritmetica dei puntatori

L'aritmetica dei puntatori è un insieme di operazioni aritmetiche applicabili sui valori di tipo puntatore. Tali operazioni hanno lo scopo di consentire un alto livello di flessibilità nell'accesso a collezioni di dati omogenei conservati in posizioni contigue della memoria (per esempio array). Nei linguaggi C, C++ e C# non gestito è possibile eseguire numerose operazioni sui puntatori.

Memoria stack e heap

La memoria stack è la memoria dove vengono salvate tutte le variabili locali. La struttura è la seguente: per ogni funzione, c'è una zona di memoria in cui si trovano le variabili locali della funzione. In particolare, la zona di memoria per la funzione main (ossia quella che contiene le variabili del programma principale) si trova nel punto più basso dello stack. Ogni volta che si chiama una funzione, viene creata una zona di memoria immediatamente sopra, e questa zona contiene le variabili locali della funzione chiamata (incluse le variabili che rappresentano i parametri).

La memoria dinamica consente di allocare spazi di memoria la cui dimensione si conosce solo in fase di esecuzione essa non è allocata automaticamente ma può essere allocata o rimossa solo su esplicita richiesta del programma (allocazione dinamica della memoria). L'area allocata non è identificata da un nome, ma è accessibile esclusivamente passando per un puntatore. Il suo scope, l'area di visibilità del puntatore, coincide con quello del puntatore che contiene il suo indirizzo. Il suo lifetime, il ciclo di vita della variabile, coincide con l'intera durata del programma, a meno che non venga esplicitamente deallocated; se il puntatore va out of scope, non è più possibile accedere alla variabile puntatore, l'area non è più accessibile, ma continua a occupare memoria inutilmente: si verifica l'errore di memory leak (indica un area di memoria non puntata da nessun puntatore), opposto a quello di dangling references (indica un puntatore che si riferisce ad un'area di memoria non più valida).

In molti linguaggi di programmazione i memory leak vengono curati dalla garbage collection, un motore automatico di gestione della memoria, mediante un modulo di run-time, libera le porzioni di memoria non più accessibili. I casi in cui un leak diventa serio comprendono:

- Quando il programma è lasciato in esecuzione, e consuma memoria continuamente (come lavori eseguiti in background, su server o su sistemi embedded lasciati in esecuzione per vari anni).
- Quando il programma è capace di richiedere memoria (come la memoria condivisa) non rilasciata anche quando il programma termina.
- Quando il leak viene creato all'interno del sistema operativo.
- Quando la memoria è molto limitata.

Puntatori in C e C++

Tipi di puntatori

Puntatore generico	Puntatore specializzato
<code>void * <nome puntatore></code>	<code><tipo> * <nome puntatore></code>

- | | |
|--|---|
| <ul style="list-style-type: none"> • Non permette l'aritmetica dei puntatori in quanto non si sa la dimensione della variabile. • Può essere convertito a qualsiasi altro tipo di puntatore senza un cast esplicito. • Non si può deallocare l'area puntata. • I dati puntati possono appartenere a qualsiasi tipo di dato, da un carattere ad una classe. | <ul style="list-style-type: none"> • Permette l'aritmetica dei puntatori in quanto la dimensione è nota. • Si può convertire solo a void * senza cast esplicito. • Si può deallocare l'area puntata. • I dati, possono appartenere a qualsiasi tipo di dato, ma vengono gestiti come se fossero dei dati del tipo di puntatore. |
|--|---|

Puntatori e vettori

In C e C++ un vettore è composto da un'area di memoria indirizzata da un puntatore. Il puntatore indirizza alla posizione 0 dell'array. Infatti le seguenti scritture sono equivalenti:

Tramite l'aritmetica dei puntatori

```
int main() {
    int vett[40];
    *vett = 10;
}
```

Tramite l'accesso alla posizione 0 dell'array.

```
int main() {
    int vett[40];
    vett[0] = 10;
}
```

Aritmetica dei puntatori

Ma se noi tramite l'aritmetica dei puntatori vogliamo accedere ad un altro elemento dell'array, per esempio il terzo come facciamo? Siamo obbligati ad utilizzare l'operatore di indice []?

No, è possibile usare l'operatore: +, +=, ++, -, -=, -- per spostare l'indice dell'array.

Tramite l'aritmetica dei puntatori

```
int main() {
    int vett[40];
    *(vet+1) = 10;
}
```

Tramite l'accesso alla posizione 1 dell'array.

```
int main() {
    int vett[40];
    vett[1] = 10;
}
```

Ma viene spontaneo chiedersi perché '+1' e non "+sizeof(int)" o "+4", visto che i puntatori puntano a dei byte e la dimensione di una variabile di tipo int è tipicamente 4 byte?

Semplicemente perché per i tipi puntatori, **escluso il puntatore a void**, il C e C++ definiscono le operazioni di somma e differenza tra puntatori e numeri interi. Se **k** è un intero e **ptr** è un puntatore al tipo <tipo>, **ptr + k** è un puntatore allo stesso tipo che punta alla locazione di memoria che è **k*sizeof(<tipo>)** bytes successiva rispetto a **ptr**, mentre **ptr - k** è un puntatore similmente definito ma che riferisce a **k*sizeof(<tipo>)** bytes prima rispetto a **ptr**.

sizeof(<tipo>)	sizeof(<tipo>)	sizeof(<tipo>)	sizeof(<tipo>)	sizeof(<tipo>)	sizeof(<tipo>)
ptr - 1	Ptr	ptr + 1	ptr + 2	ptr + 3	ptr + 4

Ad esempio se usiamo delle variabili di tipo intero (con relativa dimensione di 4) ed una variabile puntatore che vale 1000:

4 byte	4 byte	4 byte	4 byte	4 byte	4 byte
ptr - 1	Ptr	ptr + 1	ptr + 2	ptr + 3	ptr + 4
1000 - 4	1000	1000 + 4	1000 + 8	1000 + 12	1000 + 16

Oppure se usiamo delle variabili di tipo personalizzato (con relativa dimensione di 40) ed una variabile puntatore che vale 2000:

40 byte	40 byte	40 byte	40 byte	40 byte	40 byte
ptr - 1	Ptr	ptr + 1	ptr + 2	ptr + 3	ptr + 4
1000 - 40	1000	1000 + 40	1000 + 80	1000 + 120	1000 + 160

È anche possibile l'uso degli operatori <, <=, ==, >=, > e != per comparare gli indirizzi dei puntatori.

Questo perché i linguaggi C e C++ offrono una maggiore astrazione verso il programmatore permettendoli di ragionare come se il puntatore fosse un vettore, passiamo ad alcuni esempi:

Tramite l'aritmetica dei puntatori	Tramite l'operatore [] degli array
<pre>int main() { int vett[40]; for(int * i = vett; i < (vett+40); i++) { *i = 60; } }</pre>	<pre>int main() { int vett[40]; for(int i = 0; i < 40; i++) { *(vett + i) = 60; } }</pre>
<pre>int main() { int vett[40]; for(int * i = vett + 40 - 1; i >= vett; i--) { *i = 60; } }</pre>	<pre>int main() { int vett[40]; for(int i = 40 - 1; i >= 0; i--) { *(vett + i) = 60; } }</pre>

Allocare un'area di memoria puntata da un puntatore in C++

Prima di utilizzare un puntatore occorre essere certi che la zona di memoria puntata da p si possa utilizzare, ossia:

1. Il sistema operativo ci permette di accedere a questa zona.
2. Il contenuto di questa zona non viene modificata dal programma in modo inaspettato (più puntatori che lavorano nella stessa area di memoria in maniera non prevista dal programmatore).

In caso contrario il programma avrà comportamenti inaspettati. Fino ad adesso abbiamo visto quando si definisce una variabile di tipo puntatore, si crea una variabile il cui contenuto è un indirizzo di memoria. Se non si assegna un valore a questa variabile, il suo contenuto è indefinito ed è meglio non utilizzarlo. L'unico modo per realizzare queste due condizioni visto fino ad ora è stato quello di memorizzare in p l'indirizzo di una variabile con istruzioni del tipo di p=&a. In questo caso, infatti, si è sicuri che la zona di memoria è accessibile al programma, dato che è la zona di memoria di una sua variabile; inoltre, non può venire modificata dal programma, a meno che non si facciano delle modifiche su a.

Esiste però un altro meccanismo che soddisfa questi due requisiti, ed è quello di chiedere al sistema operativo di riservare una nuova zona di memoria. Questo meccanismo prende il nome di allocazione dinamica della memoria. Essa è una funzione che si imposta di una porzione di memoria per l'utilizzo di un programma durante la propria esecuzione e restituisce l'indirizzo alla Cella. Questa funzione garantisce che:

1. La zona di memoria è accessibile al programma.
2. Se si creano nuove variabili (per esempio con una chiamata di funzione), non saranno memorizzate in questa zone.
3. Ogni nuova chiamata alla funzione crea una zona di memoria nuova, distinta dalle precedenti.

Il primo punto garantisce che il programma può effettivamente usare queste locazioni di memoria. Il secondo e il terzo punto dicono che non ci saranno altre variabili o altri puntatori che sono associati automaticamente alla stessa zona di memoria. Questo serve a garantire che la zona di memoria non verrà cambiata inaspettatamente dalla modifica di una variabile che non avevamo messo in relazione con quella zona.

Per poter creare una nuova variabile dinamica il C++ mette a disposizione l'operatore **new** che è in grado di allocare una zona di memoria assegnandola ad una variabile di tipo puntatore. L'operatore **alloca la memoria, chiama il costruttore** se esiste, uno o più oggetti nell'area heap e ne **restituisce l'indirizzo**. In caso di errore (memoria non disponibile) restituisce NULL. Pertanto il programmatore dovrà controllare l'allocazione dello spazio di memoria è riuscito; ed è definita come:

```
tipo_variabile * new tipo_variabile [dimensione] (valore_iniziale)
```

- **tipo_variabile** è il tipo (anche astratto) dell'oggetto (o degli oggetti) da creare;
- **dimensione** è il numero degli oggetti, che vengono sistemati nella memoria heap consecutivamente (come gli elementi di un array); se questo operando è omesso, viene costruito un solo oggetto; se è presente, l'indirizzo restituito da new punta al primo oggetto;
- **valore_iniziale** è il valore con cui l'area allocata viene inizializzata (deve essere di un tipo coerente); se è omesso l'area non è inizializzata.

Vediamo alcuni esempi per capire al meglio l'utilizzo dell'operatore new. Inizializzazione:

```
int * punt = new int;
```

In questo modo si alloca un oggetto int nell'area heap e si usa il suo indirizzo per inizializzare il puntatore punt.

```
int * punt = new int ();
```

In questo modo si alloca un oggetto int nell'area heap e si usa il suo indirizzo per inizializzare il puntatore e lo si inizializza col costruttore di default.

```
int * punt = new int (10);
```

In questo modo si alloca un oggetto int nell'area heap e si usa il suo indirizzo per inizializzare il puntatore e lo si inizializza col costruttore con un parametro.

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

In questo modo si allocano 10 oggetti int nell'area heap e si usa l'indirizzo del primo oggetto per inizializzare il puntatore punt.

Assegnazione con operazione aritmetica:

```
struct nome { ..... };
nome * p_nome;
p_nome = new nome [100] + 10;
```

In tal caso definisce la struttura nome e si dichiara il puntatore p_nome a questa struttura, a cui si assegna l'indirizzo del decimo dei cento oggetti di tipo nome, allocati nell'area heap.

Allocare un'area di memoria puntata da un puntatore in C

In maniera del tutto analoga al C++ in C, per poter creare una nuova variabile dinamica il C mette a disposizione la funzione **malloc()** essa permette un'allocazione dinamica della memoria; ed è definita come:

```
void * malloc(size_t numero_di_bytes)
```

In C è consigliato l'uso dell'operatore **sizeof()** per determinare la dimensione in byte di un tipo di dato. Vediamo alcuni esempi per capire al meglio l'utilizzo della funzione malloc() e dell'operatore sizeof()

Inizializzazione:

```
int * p = (int *)malloc(sizeof(int));
```

In questo modo si alloca uno spazio per un int nell'area heap e si usa il suo indirizzo per inizializzare il puntatore punt.

```
int * p = (int *)malloc(sizeof(int)*10);
```

In questo modo si alloca uno spazio per 10 int nell'area heap e si usa il suo indirizzo per inizializzare il puntatore punt.

Deallocare un area di memoria puntata da un puntatore in C++

Per deallocare la memoria dell'area heap in C++ mette a disposizione l'operatore **delete**. Questo operatore non restituisce alcun valore e quindi deve essere usato da solo in un'istruzione.

Il prototipo dell'operatore delete si presenta in questo modo:

```
delete nome_puntatore
```

Contrariamente a quanto sembra l'operatore delete non cancella la variabile nome_puntatore, né altera il suo contenuto: l'unico effetto è quello di **richiamare il distruttore dell'oggetto e di liberare la memoria** puntata rendendola disponibile per ulteriori allocazioni (se l'operando non punta a un'area heap alcuni viene richiamato il distruttore anticipatamente senza altri effetti (memory leak o dangling references)).

Se l'operando punta a un'area in cui sono stati allocati più oggetti, delete va specificato con una coppia di parentesi quadre (senza la dimensione, che il C++ è in grado di riconoscere automaticamente). Ad esempio:

```
float * punt = new float [100];
delete [] punt;
```

L'operatore delete costituisce l'unico mezzo per deallocare memoria heap, che, altrimenti, sopravvive fino alla fine del programma, anche quando non è più raggiungibile. L'operatore delete [] non richiede la dimensione dell'array. Ad esempio:

```
int* punt = new int; // allocationa un int nell'area heap
int a;
punt = &a;
```

Deallocare un area di memoria puntata da un puntatore in C

Per deallocare la memoria dell'area heap in C mette a disposizione la funzione **free()**. Questa funzione non restituisce alcun valore e quindi deve essere usato da solo in un'istruzione.

```
void free(void *mem address)
```

Libera un blocco di memoria. Il puntatore deve contenere l'indirizzo del primo byte di memoria del blocco da liberare. La memoria allocata va sempre liberata una volta terminato il suo uso. Attenzione che la funzione non setta l'indirizzo del puntatore a NULL dopo aver liberato l'area di memoria, questo può causare comportamenti inaspettati.

Strutture Dati

Una struttura dati è un'entità usata per organizzare un insieme di dati all'interno della memoria del computer, ed eventualmente per memorizzarli in una memoria di massa. La scelta delle strutture dati da utilizzare è strettamente legata a quella degli algoritmi; per questo, spesso essi vengono considerati insieme. Infatti, la scelta della struttura dati influenza inevitabilmente sull'efficienza degli algoritmi che la manipolano.

La struttura dati è un metodo di organizzazione dei dati, quindi prescinde dai dati effettivamente contenuti.

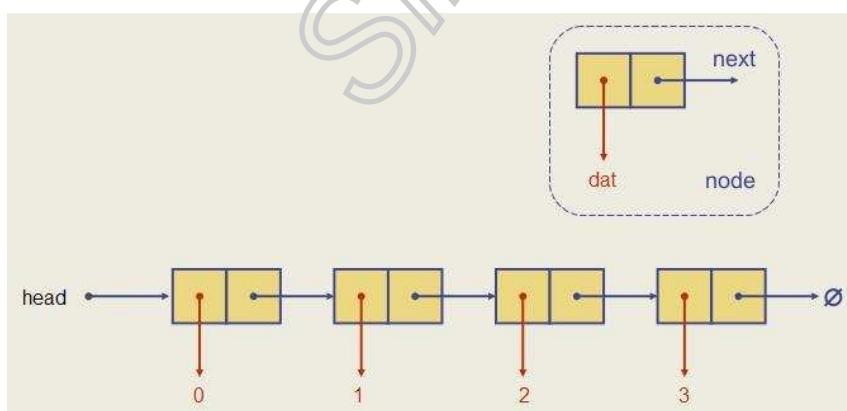
Strutture dati base

- **Array o Vettore:** è un insieme finito di elementi, *in generale* omogeneo a cui si accede tramite un indice; il numero degli elementi è rigidamente fissato a priori.
- **Struttura o Record o Classe:** è un insieme finito di elementi, *in generale* non omogeneo a cui si accede tramite il nome del campo; il numero degli elementi è rigidamente fissato a priori.
- **Tabella:** è un array di strutture, che forma una struttura tabellare.

Problemi delle Strutture Dati Statiche: Dimensione finita, impossibilità di aggiungere altri elementi, né di rimuoverli.

Strutture Dati Dinamiche

- Permettono l'aggiunta e la rimozione di elementi.
- Permettono l'utilizzo di politiche FIFO/LIFO.
- Utilizzano la memoria dinamica.
- Utilizzano un'entità chiamata «NODO»
- Obbligano un accesso sequenziale alle informazioni.
- Le operazioni di aggiunta e rimozione di elementi sono veloci, non è mai necessario aumentare la capacità.
- Gli elementi sono salvati nell'heap di memoria in maniera non contigua.
- Richiede un overhead di memoria per tenere traccia dei puntatori al nodo successivo (e precedente).
- Richiede l'utilizzo di una struttura «nodo» contenente il dato ed il puntatore al successivo.



Node

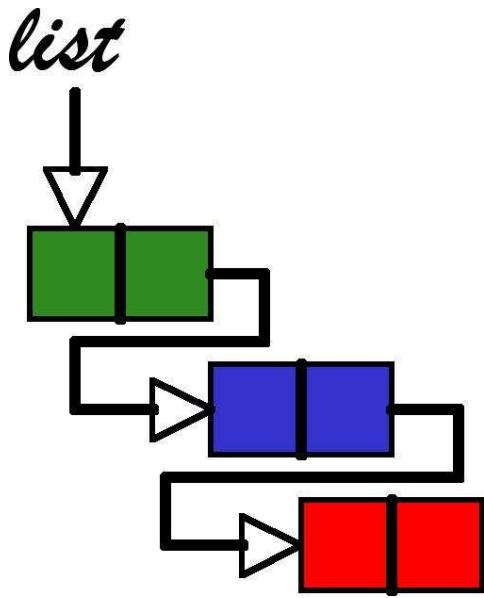
```
struct node {
    int dat; // int -> cambiare int per cambiare tipo di dato
    node *next;

    node(int dat, node *next) { // int -> cambiare int per cambiare tipo di dato
        this->dat = dat;
        this->next = next;
    }
    void clear() {
        if(next != NULL) {
            next->clear();
            delete next;
        }
    }
};
```

Simone Bortolin

Lista

È una struttura dinamica in cui è possibile accedere a tutti i dati tramite un indice, simile ad un array ma permette l'inserimento (**add**) e la rimozione (**remove**) di elementi dalla lista.



Linked List

```
class linked_list {
    node *head;
public:
    linked_list();
    ~linked_list();

    void add(int value); // int -> cambiare int per cambiare tipo di dato
    void add(int value, int index); // int -> cambiare primo int per cambiare tipo di dato
    int first(); // int -> cambiare int per cambiare tipo di dato
    int last(); // int -> cambiare int per cambiare tipo di dato
    int remove(); // int -> cambiare int per cambiare tipo di dato
    int remove(int index); // int -> cambiare primo int per cambiare tipo di dato
    bool is_empty();
    int size();
    int *to_array(); // int -> cambiare int per cambiare tipo di dato
    int &operator[](int index); // int -> cambiare primo int per cambiare tipo di dato
    int &at(int index); // int -> cambiare primo int per cambiare tipo di dato
    int search(int value); // int -> cambiare secondo int per cambiare tipo di dato
    bool contains(int value); // int -> cambiare int per cambiare tipo di dato
    void clear();

    void sort();
};

linked_list::linked_list() {
    head = NULL;
}

linked_list::~linked_list() {
    clear();
}

void linked_list::add(int value) {
    if (head == NULL) {
        head = new node(value, NULL);
    } else {
        node *aux = head;
        while (aux->next != NULL) {
            aux = aux->next;
        }
        aux->next = new node(value, NULL);
    }
}

void linked_list::add(int value, int index) {
    if (size() > index) {
        if (index == 0) {
            head = new node(value, head);
        }
        if (index == size() - 1) {
            node *aux = head;
            while (aux->next != NULL) {
                aux = aux->next;
            }
            aux->next = new node(value, NULL);
        } else {
            int i = 1;
            node *aux = head;
            while (aux != NULL) {
                if (i == index) {
                    aux->next = new node(value, aux->next);
                }
                i++;
                aux = aux->next;
            }
        }
    }
}
```

Linked List

```
    }
}

int linked_list::last() {
    if (head != NULL) {
        node *aux = head;
        while (aux->next != NULL) {
            aux = aux->next;
        }
        return aux->dat;
    }
}

int linked_list::first() {
    if (!is_empty()) {
        return head->dat;
    }
}

int linked_list::remove(int index) {
    if (size() > index) {
        if (index == 0) {
            int last = head->dat;
            node *point = head;
            head = head->next;
            delete point;
            return last;
        } else {
            int i = 1;
            node *aux = head;
            while (aux != NULL) {
                if (i == index) {
                    int el = aux->next->dat;
                    node *obsolete = aux->next;
                    aux->next = aux->next->next;
                    delete obsolete;
                    return el;
                }
                i++;
                aux = aux->next;
            }
        }
    }
}

int linked_list::remove() {
    if (!is_empty()) {
        int i = 1;
        node *aux = head;
        while (aux != NULL) {
            if (i == size() - 1) {
                int el = aux->next->dat;
                node *obsolete = aux->next;
                aux->next = aux->next->next;
                delete obsolete;
                return el;
            }
            i++;
            aux = aux->next;
        }
    }
}

bool linked_list::is_empty() {
    return head == NULL;
}

int linked_list::size() {
```

Linked List

```
int i = 0;
node *aux = head;
while (aux != NULL) {
    i++;
    aux = aux->next;
}
return i;

int *linked_list::to_array() {
    int *array = new int[size()];
    int i = 0;
    node *aux = head;
    while (aux != NULL) {
        array[i] = aux->dat;
        i++;
        aux = aux->next;
    }
    return array;
}

int &linked_list::operator[](int index) {
    int i = 0;
    node *aux = head;
    while (aux != NULL) {
        if (i == index) {
            return aux->dat;
        }
        i++;
        aux = aux->next;
    }
}

int &linked_list::at(int index) {
    int i = 0;
    node *aux = head;
    while (aux != NULL) {
        if (i == index) {
            return aux->dat;
        }
        i++;
        aux = aux->next;
    }
}

int linked_list::search(int value) {
    int i = 0;
    node *aux = head;
    while (aux != NULL) {
        if (aux->dat == value) {
            return i;
        }
        i++;
        aux = aux->next;
    }
    return -1;
}

bool linked_list::contains(int value) {
    node *aux = head;
    while (aux != NULL) {
        if (aux->dat == value) {
            return true;
        }
        aux = aux->next;
    }
    return false;
}
```

Linked List

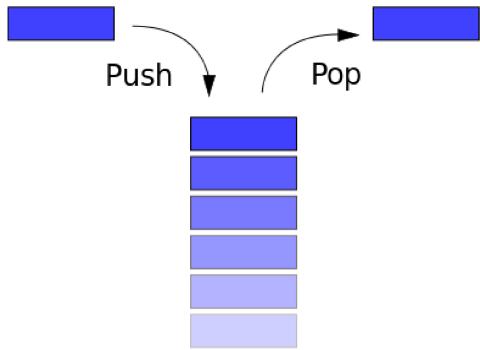
```
void linked_list::clear() {
    head->clear();
    head = NULL;
}

void linked_list::sort() {
    int count = this->size();
    node *start = NULL;
    node *curr = NULL;
    node *trail = NULL;
    node *temp = NULL;
    for (int i = 0; i < count; i++) {
        curr = trail = head;
        while (curr->next != NULL) {
            if (curr->dat > curr->next->dat) {
                temp = curr->next;
                curr->next = curr->next->next;
                temp->next = curr;

                if (curr == head) {
                    head = trail = temp;
                } else {
                    trail->next = temp;
                }
                curr = temp;
            }
            trail = curr;
            curr = curr->next;
        }
    }
}
```

Stack

È una struttura dinamica in cui le modalità d'accesso ai dati in essa contenuti seguono una modalità **LIFO** (Last In First Out), ovvero tale per cui i dati vengono estratti (**Push**) in ordine rigorosamente inverso rispetto a quello in cui sono stati inseriti (**Pop**).



Simone Bortolin

```

Linked Stack
class linked_stack {
    node *head;
public:
    linked_stack();
    ~linked_stack();
    void push(int value); // int -> cambiare int per cambiare tipo di dato
    int pop(); // int -> cambiare int per cambiare tipo di dato
    int top(); // int -> cambiare int per cambiare tipo di dato
    bool is_empty();
    int size();
    int *to_array(); // int -> cambiare int per cambiare tipo di dato
    void clear();
};

linked_stack::linked_stack() {
    head = NULL;
}

linked_stack::~linked_stack() {
    clear();
}

void linked_stack::push(int value) {
    if (is_empty()) {
        head = new node(value, NULL);
    } else {
        head = new node(value, head);
    }
}

int linked_stack::pop() {
    if (!is_empty()) {
        int last = head->dat;
        node *point = head;
        head = head->next;
        delete point;
        return last;
    }
}

int linked_stack::top() {
    if (!is_empty()) {
        return head->dat;
    }
}

bool linked_stack::is_empty() {
    return head == NULL;
}

int linked_stack::size() {
    int i = 0;
    node *aux = head;
    while (aux != NULL) {
        i++;
        aux = aux->next;
    }
    return i;
}

int *linked_stack::to_array() {
    int size = this->size();
    int *array = new int[size];
    int temp;
    int i = 0;
    node *aux = head;
    while (aux != NULL) {
        array[i] = aux->dat;
        aux = aux->next;
        i++;
    }
    return array;
}

```

Linked Stack

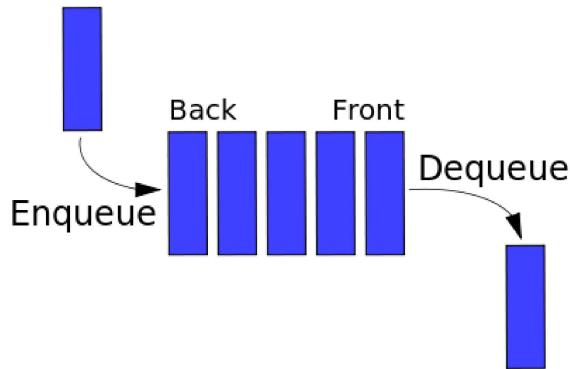
```
i++;
aux = aux->next;
}
for (int i = 0; i < size / 2; i++) {
    temp = array[i];
    array[i] = array[size - i - 1];
    array[size - i - 1] = temp;
}
return array;
}

void linked_stack::clear() {
    head->clear();
    head = NULL;
}
```

Simone Bortolin

Queue

È una struttura dinamica in cui le modalità d'accesso ai dati in essa contenuti seguono una modalità FIFO (First In First Out), ovvero tale per cui i dati vengono estratti (**Dequeue**) nello stesso ordine dell'ordine in cui vengono inseriti (**Enqueue**).



Linked Queue

```
class linked_queue {
    node *head;
public:
    linked_queue();
    ~linked_queue();
    void enqueue(int value); // int -> cambiare int per cambiare tipo di dato
    int dequeue(); // int -> cambiare int per cambiare tipo di dato
    int front(); // int -> cambiare int per cambiare tipo di dato
    int back(); // int -> cambiare int per cambiare tipo di dato
    bool is_empty();
    int size();
    int *to_array();
    void clear();
};

linked_queue::linked_queue() {
    head = NULL;
}

linked_queue::~linked_queue() {
    while (!is_empty()) {
        dequeue();
    }
}

void linked_queue::enqueue(int value) {
    if (head == NULL) head = new node(value, NULL);
    else {
        node *aux = head;
        while (aux->next != NULL) {
            aux = aux->next;
        }
        aux->next = new node(value, NULL);
    }
}

int linked_queue::dequeue() {
    if (!is_empty()) {
        int last = head->dat;
        node *point = head;
        head = head->next;
        delete point;
        return last;
    }
}

int linked_queue::back() {
    if (head != NULL) {
        node *aux = head;
        while (aux->next != NULL) {
            aux = aux->next;
        }
        return aux->dat;
    }
}

int linked_queue::front() {
    if (!is_empty()) {
        return head->dat;
    }
}

bool linked_queue::is_empty() {
    return head == NULL;
}

int linked_queue::size() {
    int i = 0;
```

Linked Queue

```
node *aux = head;
while (aux != NULL) {
    i++;
    aux = aux->next;
}
return i;
}

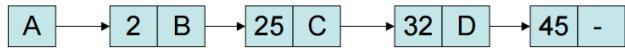
int *linked_queue::to_array() {
    int *array = new int[size()];
    int i = 0;
    node *aux = head;
    while (aux != NULL) {
        array[i] = aux->dat;
        i++;
        aux = aux->next;
    }
    return array;
}

void linked_queue::clear() {
    head->clear();
    head = NULL;
}
```

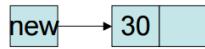
Simone Bortolin

Lista ordinata

Una lista ordinata è una lista dove gli elementi della lista concatenati in maniera ordinata.



Poichè $A \rightarrow elem = 2 < val = 30$,
 $B \rightarrow elem = 25 < val$ ma
 $C \rightarrow elem = 32 \geq val$
il punto giusto è tra C e D.



Creiamo il nodo da inserire:

Order Linked List

```
class order_linked_list {
    node * head;
public:
    order_linked_list();
    ~order_linked_list();
    void add(int value); // int -> cambiare int per cambiare tipo di dato
    int first(); // int -> cambiare int per cambiare tipo di dato
    int last(); // int -> cambiare int per cambiare tipo di dato
    int remove(int value); // int -> cambiare secondo int per cambiare tipo di dato
    bool is_empty();
    int size();
    int size(int value); // int -> cambiare secondo int per cambiare tipo di dato
    int * to_array(); // int -> cambiare int per cambiare tipo di dato
    int & operator[](int index); // int -> cambiare primo int per cambiare tipo di dato
    int & at(int index); // int -> cambiare primo int per cambiare tipo di dato
    int search(int value); // int -> cambiare secondo int per cambiare tipo di dato
    bool contains(int value); // int -> cambiare int per cambiare tipo di dato
    void clear();
};

order_linked_list::order_linked_list() {
    head = NULL;
}
order_linked_list::~order_linked_list() {
    clear();
}
void order_linked_list::add(int value) {
    node * next, *previous;

    if ((head == NULL) || (head->dat > value)) {
        head = new node(value, head);
    }
    else {
        previous = head;
        next = head->next;
        while ((next != NULL) && (next->dat < value)) {
            previous = next;
            next = next->next;
        }
        previous->next = new node(value, next);
    }
}
int order_linked_list::last() {
    if (head != NULL) {
        node * aux = head;
        while (aux->next != NULL) {
            aux = aux->next;
        }
        return aux->dat;
    }
}
int order_linked_list::first() {
    if (!is_empty()) {
        return head->dat;
    }
}
int order_linked_list::remove(int value) {
    int i = size(value);
    while (size(value) > 0) {
        node * current, *previous;
        previous = NULL;
        for (current = head; current != NULL; previous = current, current = current->next
    ) {
            if (current->dat == value) {
                if (previous == NULL) {
                    head = head->next;
                }
                else {
                    previous->next = current->next;
                }
            }
        }
    }
}
```

Order Linked List

```
        }
        delete current;
        break;
    }
}
return i;
}
bool order_linked_list::is_empty() {
    return head == NULL;
}
int order_linked_list::size() {
    int i = 0;
    node * aux = head;
    while (aux != NULL) {
        i++;
        aux = aux->next;
    }
    return i;
}
int order_linked_list::size(int value) {
    int i = 0;
    node * aux = head;
    while (aux != NULL) {
        if (value == aux->dat) i++;
        else if (i>0) break;
        aux = aux->next;
    }
    return i;
}
int * order_linked_list::to_array() {
    int * array = new int[size()];
    int i = 0;
    node * aux = head;
    while (aux != NULL) {
        array[i] = aux->dat;
        i++;
        aux = aux->next;
    }
    return array;
}
int & order_linked_list::operator[](int index) {
    int i = 0;
    node * aux = head;
    while (aux != NULL) {
        if (i == index) {
            return aux->dat;
        }
        i++;
        aux = aux->next;
    }
}
int & order_linked_list::at(int index) {
    int i = 0;
    node * aux = head;
    while (aux != NULL) {
        if (i == index) {
            return aux->dat;
        }
        i++;
        aux = aux->next;
    }
}
int order_linked_list::search(int value) {
    int i = 0;
    node * aux = head;
    while (aux != NULL) {
        if (aux->dat == value) {
```

Order Linked List

```
    return i;
}
i++;
aux = aux->next;
}
return -1;
}
bool order_linked_list::contains(int value) {
node * aux = head;
while (aux != NULL) {
if (aux->dat == value) {
return true;
}
aux = aux->next;
}
return false;
}
void order_linked_list::clear() {
head->clear();
head = NULL;
}
```

Simone Bortolin