Liste

Le liste sono delle sequenze di elemetni collegati l'uno al proprio successivo.

Vantaggio

Una lista non è preallocata, ma viene "*popolata*" man mano che sono inseriti gli elementi, quindi **la dimensione non è limitata**.

> Ad esempio se volessi fare un “elenco” di studenti di cui non conosco il numero *posso fare un array* con un numero elevato di persone, *ma avrei uno spreco di memoria*. Per risolvere questo problema posso utilizzare la lista e man mano aggiungere elementi.

Svantaggio

Per accedere alla lista è necessario *scorrere* la lista. Pertanto la complessità **non è costante**.

Le liste si suddividono in *liste linkate semplici* e *liste doppiamente linkate*.

Le operazioni che possono essere svolte con le liste

Le operazioni effettuabili sulle liste sono molte più di quelle che vedremo.
Infatti analizzeremo solo **controllo della lista vuota**, **inserimento**, **cancellazione** e **ricerca** ed accenneremo brevemente all'*accesso alla lista*. Ma in realtà esse sono *solo alcune tra* le operazioni effettuabili.
*Le operazioni più conosciute e utilizzate sono 7, ma **non sono le uniche***:

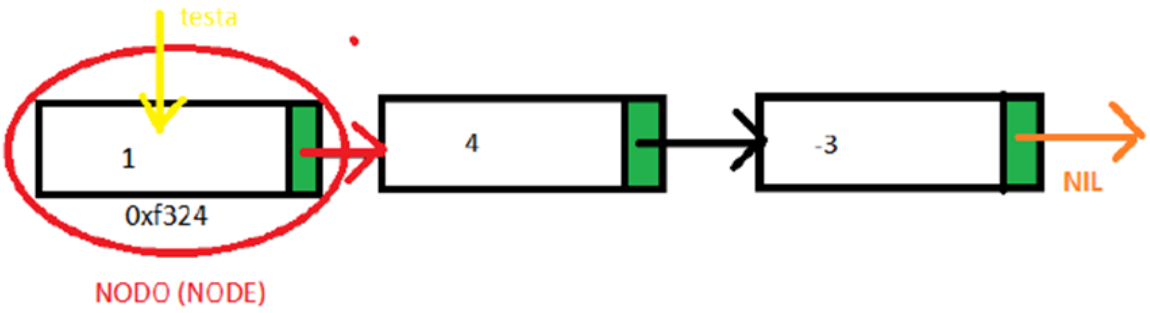
1. Inserimento (all'inizio la lista è vuota);
2. Accesso;
3. Ricerca;
4. Cancellazione;
5. Ordinamento;
6. Copia;
7. Controllo se la lista è vuota.

Liste linkate semplici

Le liste linkate semplici sono delle sequenze di elementi collegati tra loro di *seguito* con un **singolo collegamento** al nodo successivo.

Com'è fatta una lista *linkata semplice*?

Una lista è composta dai nodi.
Un nodo è composto da una variabile (*a sinistra*) e dal puntatore (*a destra*) all'elemento successivo.
Il puntatore è formalmente chiamato **link**. (*Per tale ragione la lista è formalmente chiamata **linked list***).
La lista termina con un puntatore ***NIL***.



Per definire una lista occorre:

1. il nodo testa che è un puntatore di tipo `Node<T>* head*`
2. il tipo di dato che influenza la lista e i nodi

Per definire un nodo occorre:

1. un tipo di dato che rappresenta il valore `T val`
2. un puntatore al nodo successivo `Node<T> *next`

Implementazione della lista *linkata semplice*

In primo luogo controllo se la lista è già stata definita ed includo le librerie necessarie

```C++
#ifndef LIST_H
#define LIST_H
#include <iostream>
using namespace std;
```

Successivamente implemento la classe **List** che sarà `template` e che avrà come attributo *private* il puntatore alla testa e come metodo *public* il costruttore con all'interno l'inizializzazione della testa a `nullptr`.

```C++
    template<typename T>
        class List{
        private:
          Node<T>* head;
        public:
          List(){
          head=nullptr; //inizializzo head per evitare di avere puntatori
"spuri"
          }
      };
```

È necessario inizializzare head a `nullptr` perché altrimenti sarebbe un puntatore "*spurio*". Pertanto una lista avrebbe head come puntatore a qualcosa di **non definito**.

Terminerò il procedimento scrivendo l'operatore `#endif`

Implementazione del nodo

Come abbiamo visto l'attributo della classe Lista è il puntatore di head, ma esso altro non è che un nodo. Per tale ragione occorre *definire la classe nodo prima di definire la classe lista.*

In primo luogo controllo se il nodo è già stato definito ed includo le librerie necessarie

```C++
    #ifndef LNODE_H //L sta per Linked
    #define LNODE_H
    #include <iostream>
    using namespace std;
```

Successivamente implemento la classe **Node** che sarà `template` e che avrà come attributi *private* il valore e il puntatore la nodo successivo e come metodo *public* il costruttore in cui è inizializzato next a `nullptr`.

```C++
    template <typename T>
        class Node {
        private:
          T val;
          Node<T>* next;
```

```
    public:
      Node(T val) : val(val){
      next=nullptr;
    }
};
```

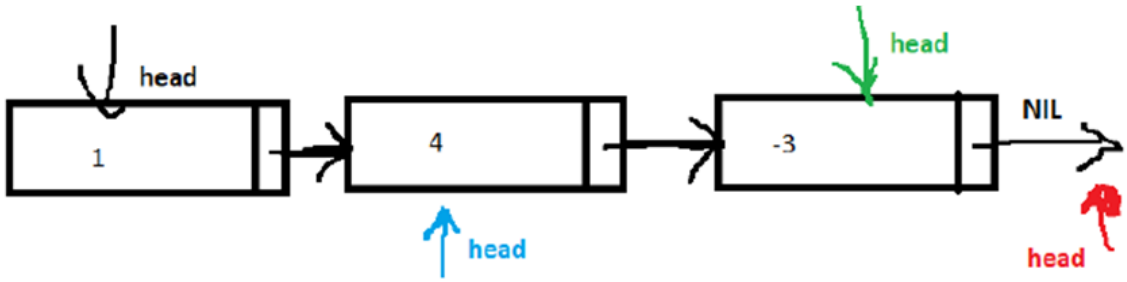
Terminerò il procedimento scrivendo l'operatore `#endif`.

Accesso alla lista *linkata semplice*

Come visto prima per accedere ad un elemento della lista è necessario scorrere la struttura. Per far questo si utilizza il **puntatore al primo elemento (*nodo*) della lista** formalmente chiamato *testa* (head). Inoltre -come già visto- ogni nodo ha il proprio successore; l'ultimo avrà come successore `nullpointer` indicante la fine della lista ed indicato con ***NIL***.

Controllo lista vuota *linkata semplice*

Quando una lista si dice vuota



Considerate le diverse frecce come indicanti l'inizio di diverse liste, avremo che l'unica lista vuota sarà quella indicata dalla freccia rossa.

Difatti mentre in ognuna delle altre liste è presente un elemento in testa, quella indicata dalla freccia rossa non ha alcun elemento in testa, *anzi* la testa corrisponde a `nullptr`.

Procedimento (e codice)

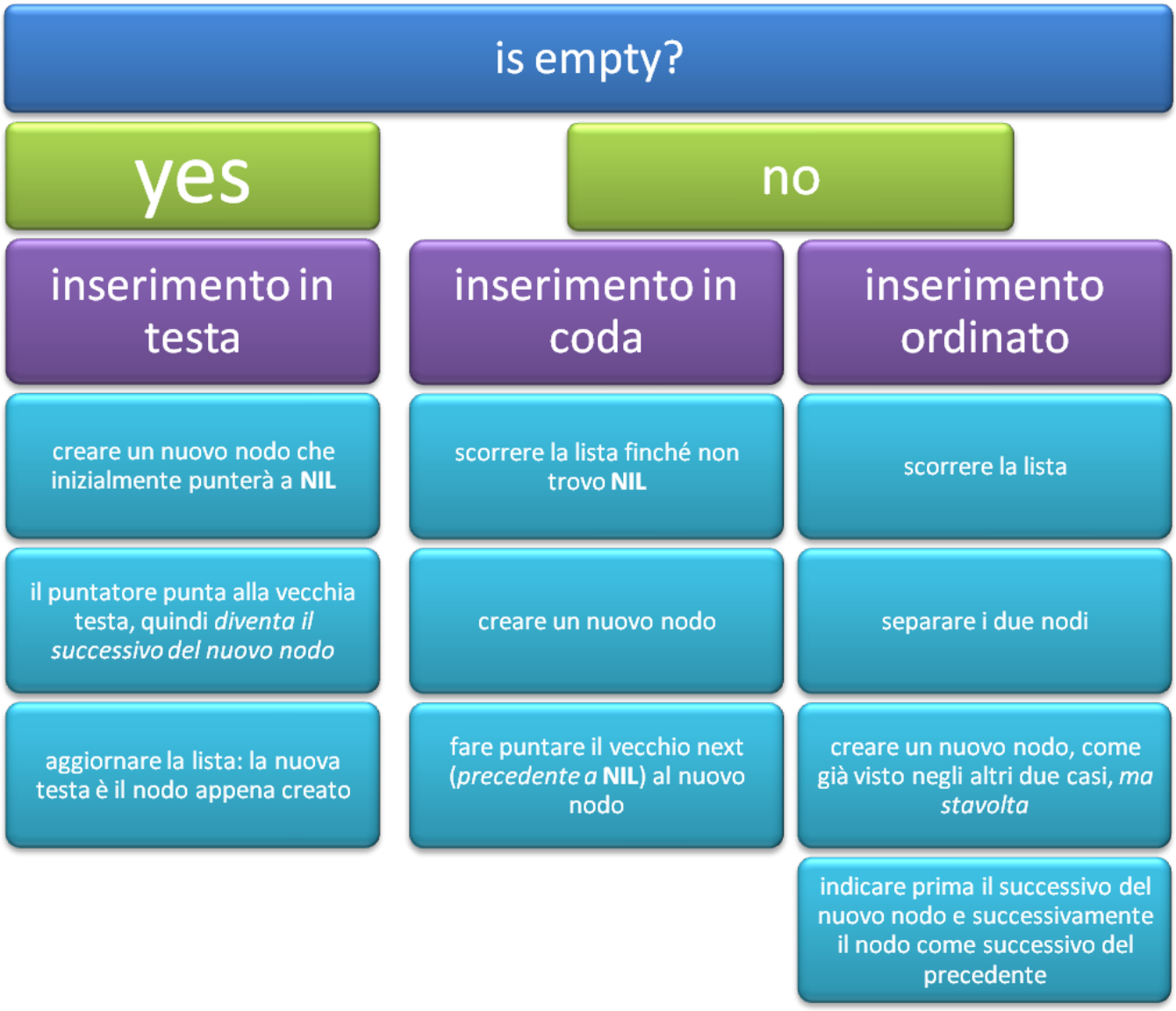
Il controllo va effettuato **sull'head**, infatti è sufficiente che essa corrisponda a `nullptr` affinché la lista sia considerata vuota.

Si scrive quindi una funzione con valore di ritorno un `bool`. Esso risulterà vero se la testa dovesse corrispondere a `nullptr`.

C++

```
bool isEmpty(){
    return head == nullptr;
}
```

Inserimento in lista *linkata semplice*



Codice inserimento in testa

In primo luogo istanziamo la funzione `insert`:

```C++
void insert(T val){
    if(this->isEmpty()){
      head = new Node<T>(val); //creo nuovo nodo e assegno il valore "val"
     }
  }
```

Successivamente istanziamo `insertHead` dichiarando la funzione di *controllo lista vuota*, richiamando la funzione *insert*.

```C++
    void insertHead(T val){
        if(this->isEmpty()){
        this->insert(val);
        return;
    }
```

Infine, creiamo un nuovo nodo, assegnamo head al nuovo nodo temp e assegnamo alla testa temp

```C++
    Node<T>* temp = new Node<T>(val);
    temp->next = head;
    this->head=temp;
```

In questo modo non posso accedere a `next` poiché è un campo *private*. Quindi utilizzerò la keyword `friend` nella classe nodo. Cambierò il nome della classe template non utilizzando acora **T**. La dichiaro di tipo `template` per indicare al nodo che la classe `List` sarà di tipo Template.

```C++
    template<typename U>
        friend class List;
```

Questa definzione è presente *nella classe* **Node** e la usiamo per indicare che in futuro sarà presente una classe template `List` che sarà ***friend*** della classe `Node`. In questo modo List può accedere ai campi **private** di node.

Si potrebbe altrimenti utilizzare un campo pubblico della classe `node`, ma non converrebbe perché violerebbe la privacy.

Metodo di stampa

Per stampare il nodo e la lista utilizzerò l'overload di `ostream&`. In particolare scriverò:

Per il nodo :

Stamperò utilizzando l'operator << come *friend* il valore del nodo e l'indirizzo del successivo. Utilizzo **friend** perché voglio accedere ai campi privati del nodo.

```C++
    friend ostream& operator << (ostream& out, const Node<T> &node){
        out << "node val " << node.val << "- next= " << node.next; //node.next
indirizzo del next
```

```
        return out;
    }
```

Per la .lista:

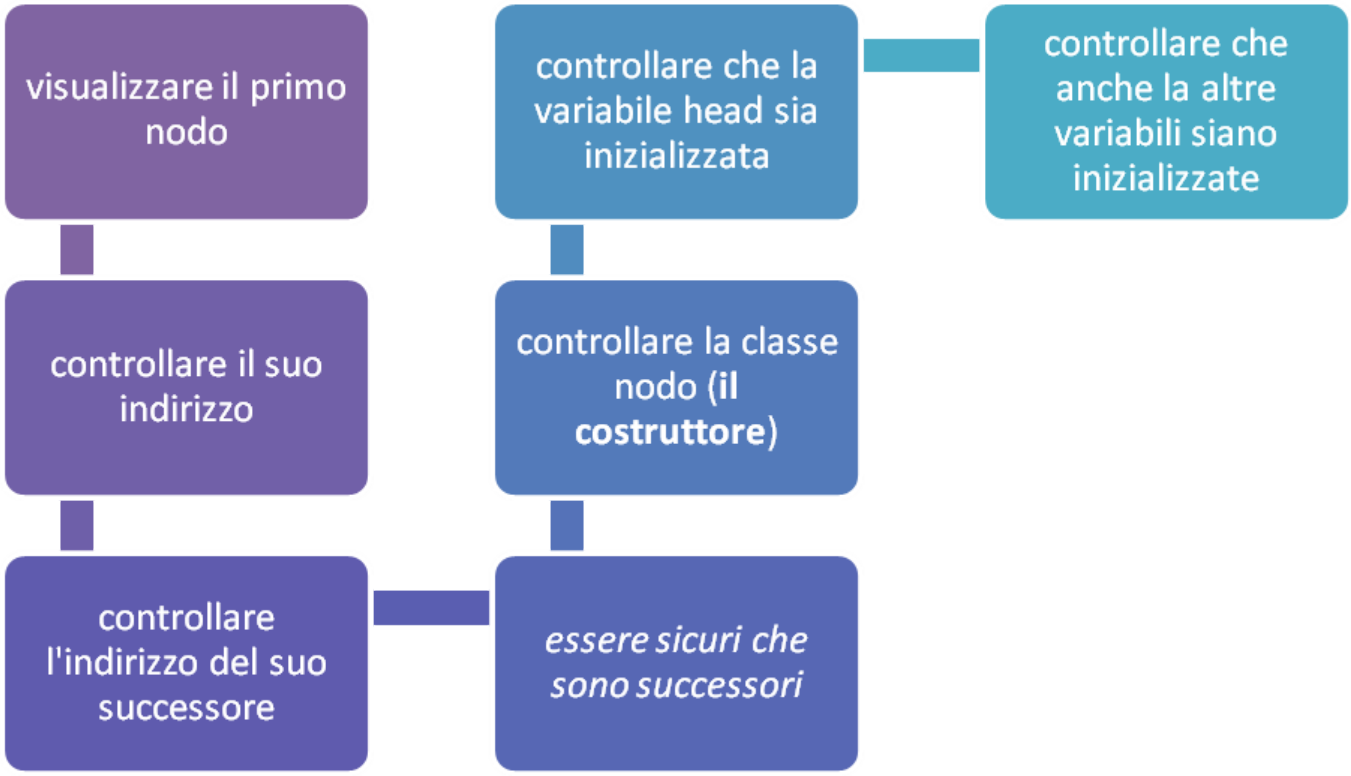
Stamperò la testa e l'indirizzo della testa. Dopodiché scorro con una lista while da `head` a `nullptr`.

```C++
    friend ostream& operator<< (ostream& out, const List<T> &list){
        out << "List head=" << list.head << endl;
        Node<T> *ptr = list.head;
        while (ptr != nullptr){
            out << "\t" << *ptr << endl;
            ptr=ptr->getNext(); //avanzamento di ptr
        return out;
    }
```

Per accedere da `iostream` a Ptr->next ***si usa getNext**** che restituisce il puntatore al nodo successivo. Difatti:

1. un metodo `get` restituisce un valore (*non importa quale*) che garantisce **accesso in lettura** (quindi un accesso più blando)
2. Se dichiaro iostream come **friend**, chiunque usa un operatore di reiderezione può intervenire sul nodo, ma io voglio solo intervenire sull'overload dell'operatore

Per risolvere eventuali problemi di inserimento, si può seguire questa procedura:



Codice inserimento in coda

controllo per prima cosa se la lista è vuota, in quel caso faccio *riferimento a* `insertHead()`.

```C++
void InsertTail(T val){
    if (this->isEmpty()){
        this->insertHead;
        return;
    }
```

Successivamente creo un puntatore a cui assegno head e che di volta in volta "*cambierò*" col nodo successivo (quindi **scorro la lista**); finché non ho il puntatore precedente a `nullptr`.

```C++
Node<T>* ptr = head;
while (ptr->get(Next) != nullptr)
    {
      ptr = ptr->get(Next);
}
```

Infine creo il nuovo nodo con valore "*val*", il next del vecchio puntatore sarà `temp`, mentre il next di temp sarà **di default** `nullptr`.

```C++
    Node<T> *temp = new Node<T>(val);
    ptr->next = temp;
    // (di default) temp -> next = nullptr;
}
```

Codice inserimento ordinato

Controllo se la lista è vuota, in quel caso richiamo la funzione `insertHead`

```C++
    void insertInOrder(T val){
        if (this->isEmpty()){
            this->insertHead(val);
            return;
         }
```

Successivamente controllo se il valore è minore o uguale del valore in testa, in quel caso richiamo la funzione `insertHead`

```C++
    if(val<=head->val){
        this->insertHead(val);
        return;
    }
```

Dopodiché inizializzo il puntatore alla testa e lo faccio scorrere finché il successivo non è `nullptr` e finché il valore è minore o uguale del successivo. Se il successivo al puntatore è minore o uguale a **val**, si fa un `break`. Altrimenti incremento il ptr e vado al prossimo nodo

```C++
    Node<T>* ptr = head;
    while (ptr->getNext() && (val <= ptr->val && val)
           {
               if (val<=ptr->next->val)
                   break;
               ptr = ptr->get(Next);
           }
```

Una volta usciti dal while controllo il "*motivo*" per cui si è usciti. Per prima cosa controlliamo se la ragione sia stata la presenza di `nullptr` a seguito del puntatore, in quel caso si richiama la funzione `insertTail`

```C++
if!(ptr->next){
    this->insertTail(val);
    return;
}
```

Altrimenti inizializziamo il nodo a val, facciamo corrispondere il successivo del nodo al successore del puntatore che stiamo considerando e facciamo corrispondere il successore del puntatore che stiamo considerando al nodo da inserire

```C++
Node<T>* toInsert = new Nodo<T>(val);
toInsert->next = ptr->next;
ptr->next = toInert;
```

> È importante che si indichi prima il successore del nodo e poi il successore del puntatore perché altrimenti parte della lista andrebbe persa (in quanto il successore del nodo da inserire è di *default* corrispondente a `nullptr`).

Cancellazione di un nodo di una lista *linkata semplice*

Codice cancellazione in testa

Per prima cosa controllo se la lista sia vuota, in tal caso stamperò semplicemente un avviso riguardo ciò

```C++
void removeHead(){
    if (this->isEmpty()){
        cout << "empty List" << endl;
        return;
    }
```

altrimenti creiamo un puntatore temporaneo inizializzato a `head`, facciamo scorrere la lista e cancelliamo `temp`.

```C++
    Node<T>* temp = head;
    head = head->next;
    delete temp;
}
```

Codice cancellazione in coda

Per prima cosa controllo se la lista sia vuota, in tal caso stamperò semplicemente un avviso riguardo ciò

```C++
    void removeTail(){
        if (this->isEmpty()){
            cout << "Empty List" << endl;
            return;
        }
```

Altrimenti inizializziamo due puntatori: uno a head e uno a `nullptr`. Istanziamo un ciclo `while` con condizione cur->next non abbia un puntatore nullo, quindi *scorriamo la lista*.

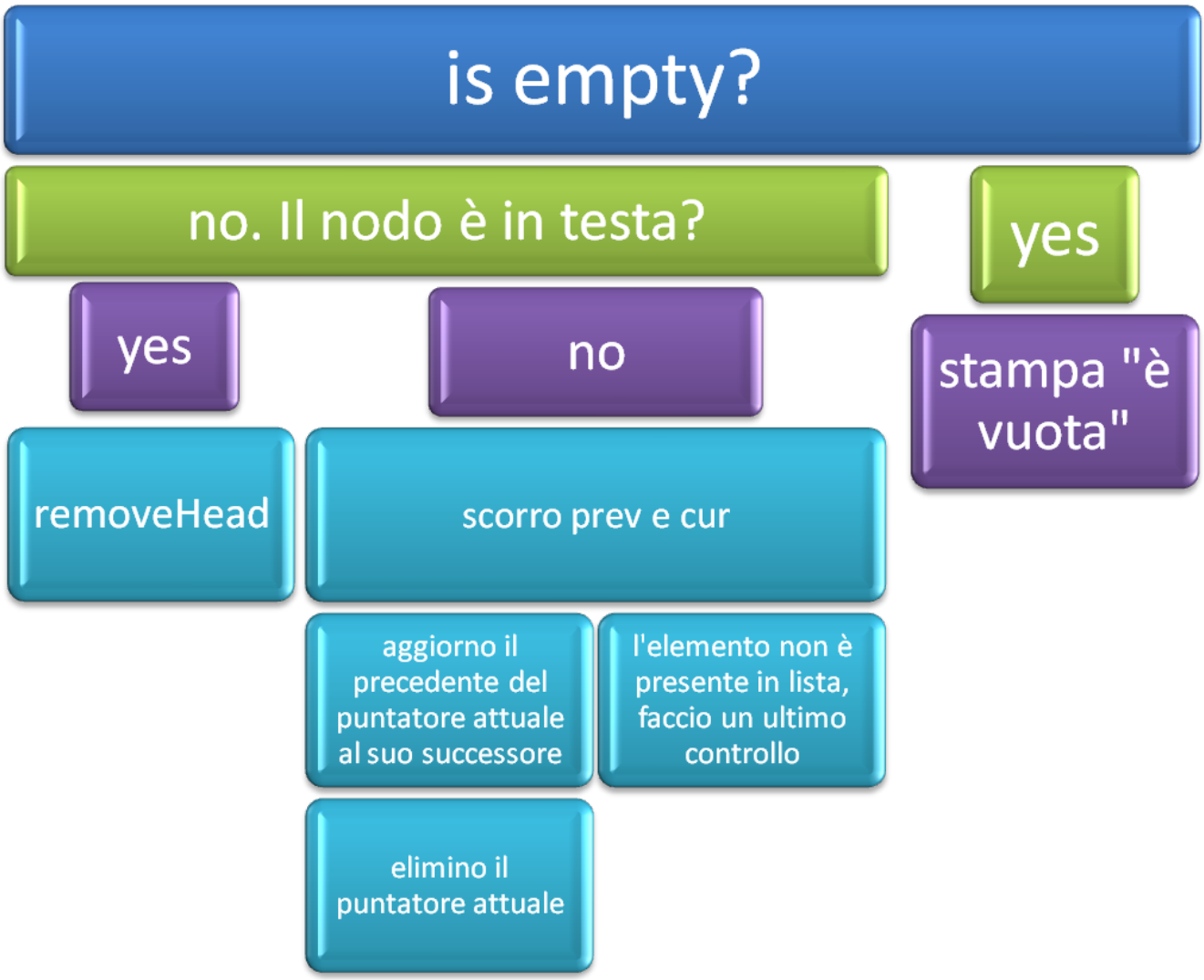
```C++
    Node<T>* cur = head;
    Node<T>* prev = nullptr;
    while (cur->next->next){
        prev = cur;
        cur = cur->next;
    }
```

Alla fine eliminiamo il nodo facendo puntare il successore di prev a `nullptr`.

```C++
    prev->next = nullptr;
```

Cancellazione di un valore specifico (*linkata semplice*)

Per far ciò mi servo delle due cancellazioni poco prima esaminate: quella in **testa** e quella in **coda**.



Codice

Per prima cosa controllo se la lista sia vuota, in tal caso stamperò semplicemente un avviso riguardo ciò

```C++
void remove(T val){
    if(this->isEmpty()){
        cout << "empty List" << endl;
        return;
    }
```

Successivamente controllo se il valore si trova in testa e richiamo la funzione `removeHead`

```C++
if(head->val == val){
    this->removeHead();
```

```
        return;
    }
```

Altrimenti inizializzo due puntatori uno a `head` ed uno a `nullptr`. Scorro la lista dando come condizione del ciclo `while` che cur non sia seguito da un nullptr e che il valore di cur non corrisponda al valore cercato

```C++
    Nodo<T>* cur = head;
    Nodo<T>* prev = nullptr;

    while ((cur->next) && (cur->val!=val)){
        prev = cur;
        cur = cur->next;
    }
```

Infine controllo il motivo per cui sono fuori dal while: se non è stato trovato il valore stamperò "valore non trovato", altrimenti pongo il predecessore del puntatore attuale come predecessore del successore del puntatore attuale ed elimino il puntatore attuale

```C++
    if(cur->val != val){
        cout << "Element with value " << val << " not found" << endl;
        return;
    }

    prev->next = cur->next;
    delete next;
}
```

Liste doppiamente linkate

Le liste doppiamente linkate sono delle sequenze di elementi collegati tra loro di *seguito* con un **doppio collegamento**: uno al nodo successivo ed uno a quello precedente.

Com'è fatta una lista *doppiamente linkata*

Una lista è composta dai nodi.
Un nodo è composto da una variabile (*al centro*) e da due puntatori (*alle estremità*) di cui quello a *destra* è quello all'elemento successivo e quello *a sinistra* è quello all'elemento precedente.
La lista termina con un puntatore ***NIL***.

Per definire una lista occorre:

1. il nodo testa che è un puntatore di tipo `Node<T>* head*`
2. il nodo coda che è un puntatore di tipo `Node<T>* tail*`
3. il tipo di dato che influenza la lista e i nodi

Per definire un nodo occorre:

1. un tipo di dato che rappresenta il valore `T val`
2. un puntatore al nodo successivo `Node<T> *next`
3. un puntatore al nodo precedente

Implementazione della lista *doppiamente linkata*

In primo luogo controllo se la lista è già stata definita ed includo le librerie necessarie

```C++
#ifndef DLLIST_H
#define DLLIST_H
#include <iostream>
using namespace std;
```

Successivamente istanzio una classe `template` che abbia come attributi *private* il puntatore alla testa e il puntatore alla coda e come metodo *public* il costruttore che inizializza `head` e `tail` a `nullptr`.

```C++
template<typename T>
    class DLList{
    private:
      Node<T>* head;
      Node<T>* tail;
    public:
      List(){
      head=nullptr; //inizializzo head per evitare di avere puntatori
"spuri"
      tail = nullptr; //inizializzo tail per lo stesso motivo
      }
  };
```

Implementazione del nodo *(DLNode)*

In primo luogo controllo se il nodo è già stato dichiarato ed includo le librerie necessarie

```C++
#ifndef DLNODE_H
#define DLNODE_H
#include <iostream>
using namespace std;
```

Successivamente istanzio una classe `template` che abbia come attributi *private* due puntatori (uno al successore, uno al predecessore) e come metodo *private* il riferimento alla classe Lista (classe ricordiamo **template**) con la keyword `friend`. Come metodo *public* invece scriviamo il **costruttore completo**.

```C++
template <typename T>
class DLNode{
    private:
    DLNode<T> *next;
    DLNode<T> *prev;
    T val;
    template <typename U>
    friend class DLList;

    public:
        DLNode(T val): val(val), next(nullptr), prev(nullptr){}
}
```

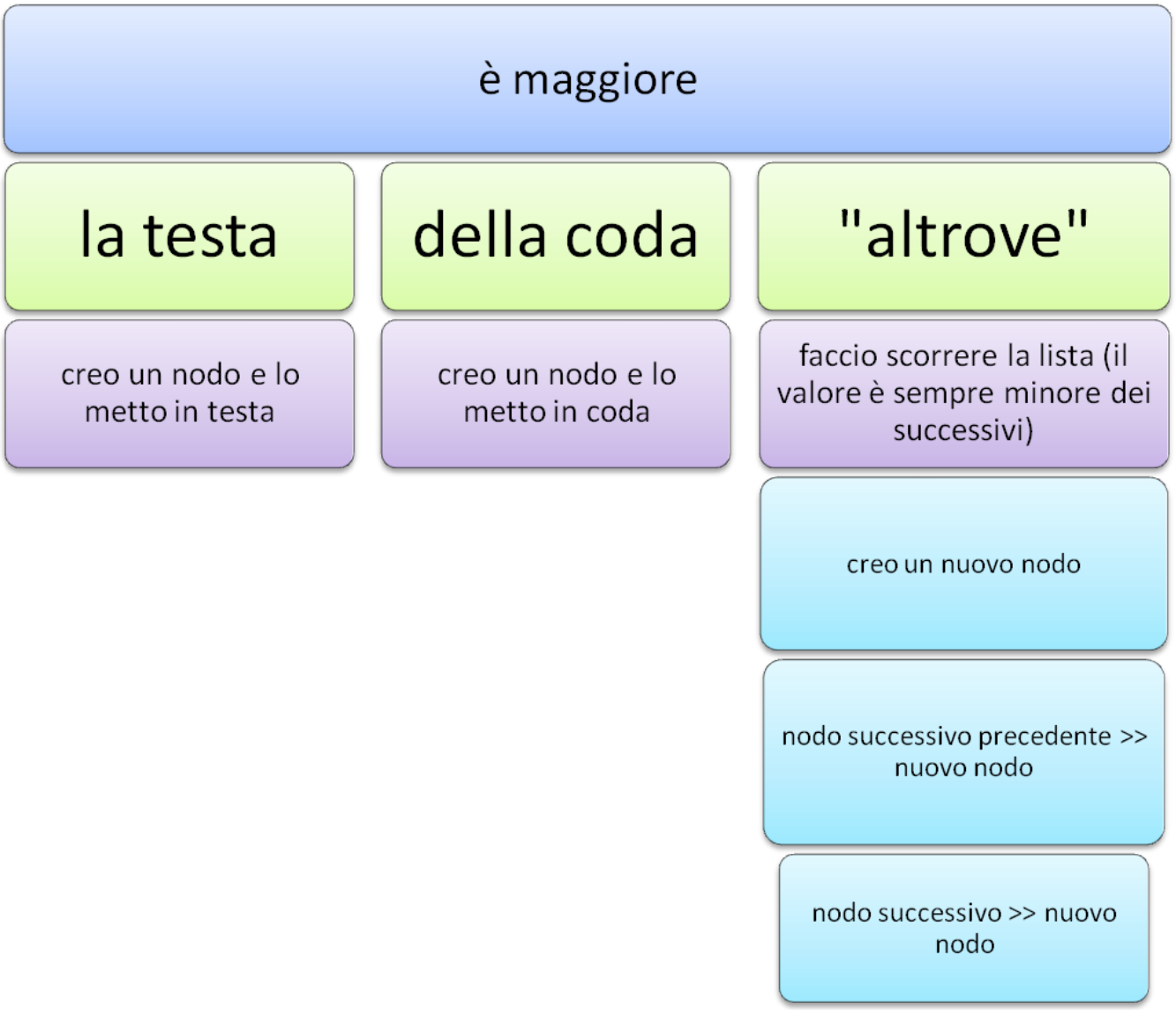
Controllo lista vuota *doppiamente linkata*

Stavolta a differenza della *lista semplicemente linkata* devo controllare **anche la coda**. Infatti anch'essa deve essere vuota affinché la lista sia vuota.

```C++
bool isEmpty(){
    return (head == tail) && (tail== nullptr)
}
```

Inserimento in lista *doppiamente linkata*


is empty?
yes
no
insertHead
creo un nuovo nodo su head
pongo la coda uguale alla testa in quanto è presente solo un nodo
insertHead
creo un nodo da inserire e pongo il suo successore uguale alla testa attuale
pongo il precedente della testa uguale al nuovo nodo
pongo la testa uguale al nuovo nodo
insertTail
creo un nuovo nodo da inserire
pongo tail come predecessore del nuovo nodo
pongo il nuovo nodo come successore di tail
pongo tail pari al nuovo nodo
insertInorder
controllo se il nodo da inserire ha il valore minore alla coda o alla testa per usare i metodi precedenti
creo un puntatore che utilizzo per scorrere tutta la lista
cerco il nodo col valore minore del valore da inserire
creo il nodo mettendo come suo nodo successivo il successore del puntatore attuale e come suo precedente il puntatore attuale
metto il successore del precedente del puntatore pari al nuovo nodo e metto il successore del puntatore pari al nuovo nodo



Codice inserimento in testa

Controllo per prima cosa se la lista è vuota, in quel caso creo un nuovo nodo su head e pongo head uguale a tail.

```C++
void insertHead(T val){
    if(this->isEmpty()){
        head = New DLNode(val);
        tail = head;
        return;
    }
```

Altrimenti creo un nuovo nodo da inserire e pongo la `head` come suo successore e pongo questo come `head->prev` (predecessore della `head`), infine pongo la head pari al nuovo nodo `toInsert`

```C++
    DLNode<T> *toInsert = new DLNode<T>(val);
    toInsert->next = head;
    head->prev = toInsert;
    head = toInsert;
}
```

Codice inserimento in coda

Per prima cosa controllo se la lista è vuota e definisco la funzione `insertHead`

```C++
    void insertTail(T val){
        if(this->isEmpty()){
            this->insertHead(val);
            return;
        }
```

Altrimenti creo un nuovo nodo `toInsert` e pongo `tail` come predecessore di `toInsert` che pongo come successore di `tail`. Infine pongo tail pari a toInsert.

```C++
    DLNode<T>* toInsert = new DLNode<T>(val);
    toInsert->prev = tail;
    tail->next = toInsert;
    tail = toInsert;
    return;
}
```

Codice inserimento ordinato

Per prima cosa controllo se la lista è vuota e definisco la funzione `insertHead`

```C++
    void insertInOrder(T val){
        if(this->isEmpty()){
            this->insertHead(val);
            return;
        }
```

Successivamente controllo se il valore è minore o uguale al valore in testa o maggiore o uguale al valore in coda e richiamo le due funzioni `insertHead` e `insertTail`

```C++
    if(head->val == val){
        this->insertHead(val);
        return;
    }

    if(tail->val==val){
        this->insertTail(val);
    }
```

Altrimenti creo un puntatore inizializzato a `head` che serve per scorrere con l'utilizzo del `while` la lista finché il valore puntato è minore del valore del valore da inserire e finché la lista non è finita (*si arriva a* `nullptr`). Al suo interno pongo il controllo sul valore dell'elemento successivo al puntatore attuale.

```C++
    DLNode<T>* ptr = head;
    while((ptr->next) && (val>=ptr->val){
        if (val<= ptr->next->val)
            break;
        ptr = ptr->next;
```

Infine creo un nuovo nodo il cui successore lo faccio corrispondere al successore del `ptr` attuale e il cui precedente lo faccio corrispondere al `ptr` stesso. Inoltre pongo il **predecessore del puntatore successivo** a `toInsert` e il successore del puntatore attuale a `toInsert`.

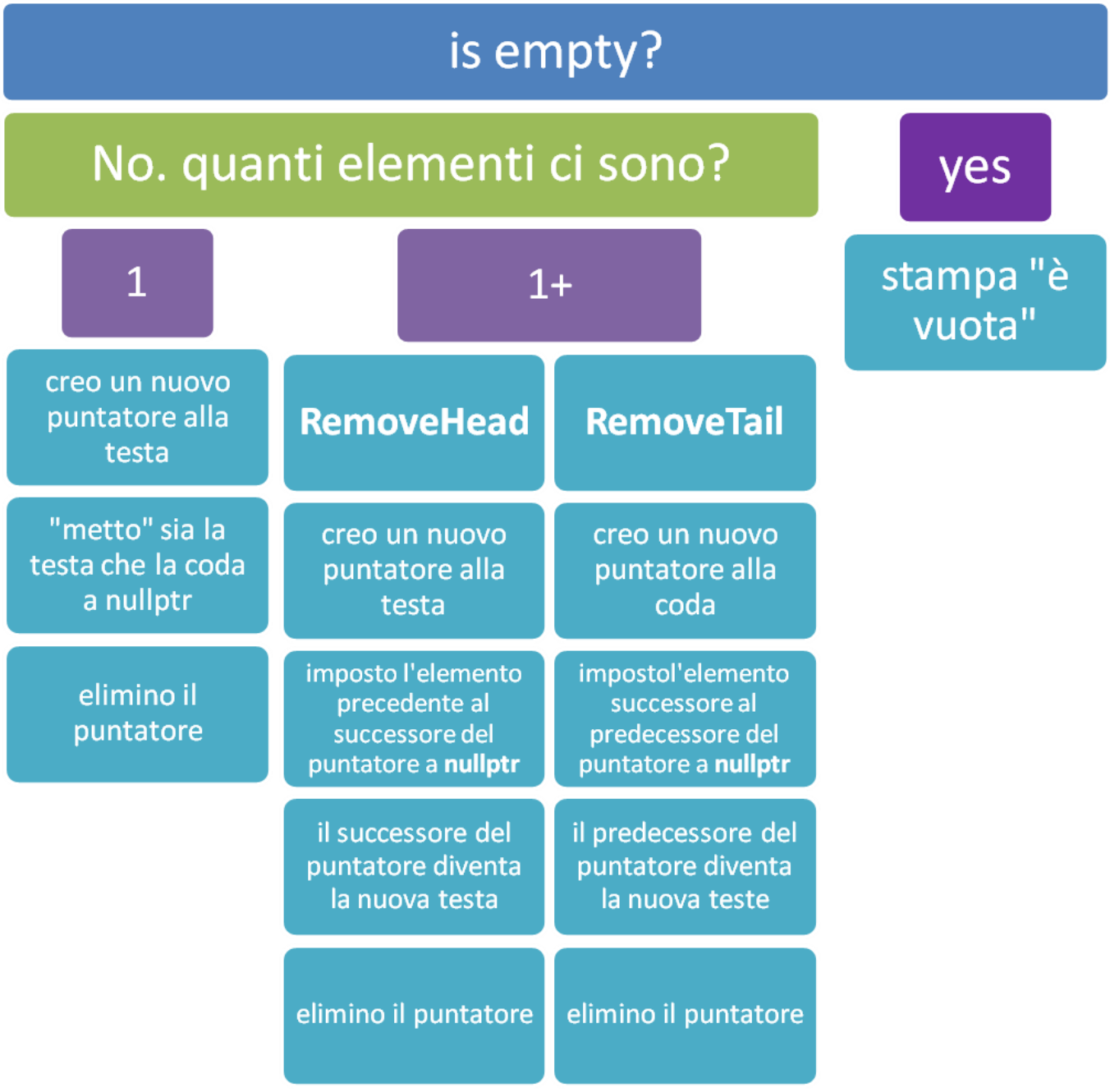
```C++
    DLNode<T>* toInsert = new DLNode<T>(val);
    toInsert->next = ptr->next;
    toInsert->prev = ptr;

    ptr->next->prev = toInsert;
    ptr->next = toInsert;
    return;
}
```

> È importante che si indichi prima il successore del nodo e poi il successore del puntatore perché altrimenti parte della lista andrebbe persa (in quanto il successore del nodo da inserire è di *default* corrispondente a `nullptr`).

> È inoltre importante che si indichi prima il predecessore del successore del puntatore e successivamente il successore del puntatore *sempre perché* altrimenti parte della lista andrebbe persa.

Cancellazione di un nodo di una lista *doppiamente linkata*



Codice eliminazione in testa

Per prima cosa controllo se la lista è vuota e stampo per comunicarlo

```C++
void removeHead(){
    if (this->isEmpty()){
```

```
            cout << "empty list" << endl;
            return;
        }
```

Successivamente controllo se la lista ha un solo nodo (ovvero se `head == tail`) e in quel caso creo un nuovo puntatotore a `head` e pongo head pari a `nullptr`, `tail` pari a `nullptr` ed elimino il puntatore

```C++
    if (head == tail){
        DLNode<T>* ptr = head;
        head = nullptr;
        tail = nullptr;
        delete ptr;
    }
```

Altrimenti se dovesse avere più di un nodo, creo un puntatore pari allla testa, pongo il predecessore del successore del puntatore a `nullptr` e pongo la testa al successore del puntatore corrente. Elimino il puntatore

```C++
    DLnode<T> *ptr = head;
    ptr->next->prev = head;
    head = ptr->next;
    delete ptr;
```

Codice eliminazione in coda

Per prima cosa controllo se la lista è vuota e stampo per comunicarlo

```C++
    void removeHead(){
        if (this->isEmpty()){
            cout << "empty list" << endl;
            return;
        }
```

Successivamente controllo se la lista ha un solo nodo (ovvero se `head == tail`) e in quel caso creo un nuovo puntatotore a `head` e pongo head pari a `nullptr`, `tail` pari a `nullptr` ed elimino il puntatore

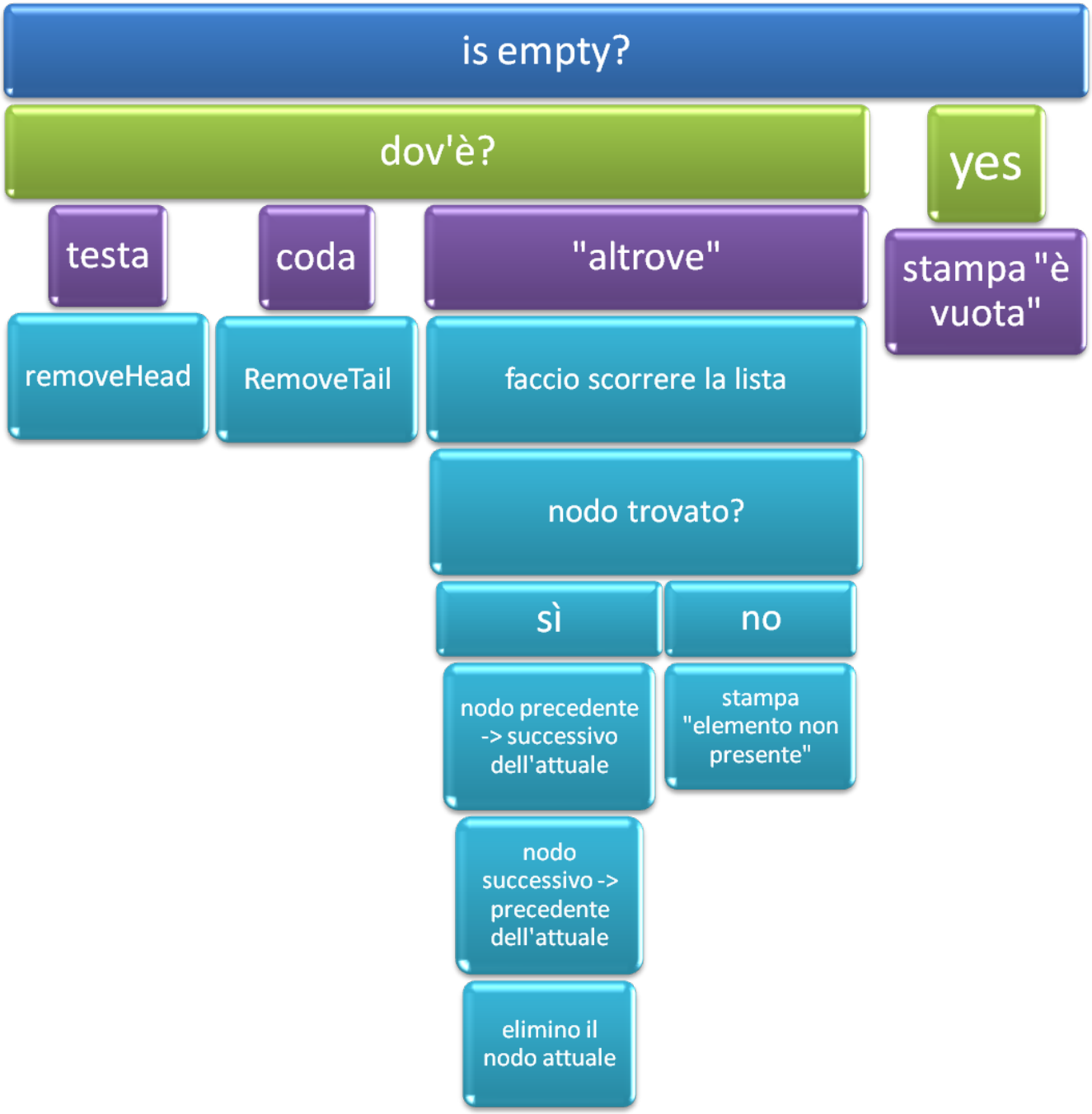
```C++
if (head == tail){
    DLNode<T>* ptr = head;
    head = nullptr;
    tail = nullptr;
    delete ptr;
}
```

Altrimenti se dovesse avere più di un nodo, creo un puntatore alla coda, pongo il successore del predecessore del puntatore a `nullptr` e pongo la coda al predecessore del puntatore corrente. Elimino il puntatore

```C++
DLNode<T>* ptr = tail;
ptr -> prev -> next = tail;
tail = ptr->prev;
delete ptr;
```

Cancellazione di un valore specifico in una lista (*doppiamente linkata*)

Per far ciò mi servo delle due cancellazioni poco prima analizzate: quella in **coda** e quella **in testa**



Codice

Per prima cosa controllo se la lista è vuota e lo comunico

```C++
void remove (T val){
    if (this->isEmpty()) {
        cout << "empty list" << endl;
        return;
    }
```

Successivamente controllo se il valore cercato si trova in testa o in coda della lista e richiamo le funzioni `removeHead` e `removeTail`

```C++
if(head->val == val){
    removeHead();
    return;
}

if(tail->val == val){
    removeTail();
    return;
}
```

se la lista dovesse avere più nodi scorro la lista. Inizializzo un puntatore quindi alla testa e lo scorro finché (con un `while`) il successore del puntatore è `nullptr` e finché il valore nel puntatore è diverso dal valore cercato.

```C++
DLNode<T>* ptr = head;
while((ptr->next) && (ptr->val != val)){
    ptr = ptr->next;
}
```

Infine controllo se si è usciti dal ciclo perché il valore non è stato trovato

```C++
if(ptr->val!=val){
    cout << "element with value " << val << " not found" << endl;
    return;
}
```

o se è stato trovato. In questo secondo caso pongo il successore dell'elemento che precede il puntatore al successore del puntatore e il predecessore del successore del puntatore al predecessore del puntatore

```C++
ptr->prev->next = ptr->next;
ptr->next->prev = ptr->prev;
```

ed elimino il puntatore

```C++
    delete ptr;
}
```