obiettivi di questa seconda metà del corso

fare un passo avanti rispetto a :

- ...
- meccanismi di composizione dei dati
  - puntatori (➡ strutture dinamiche collegate)

strutture dinamiche collegate
(liste, pile, code)

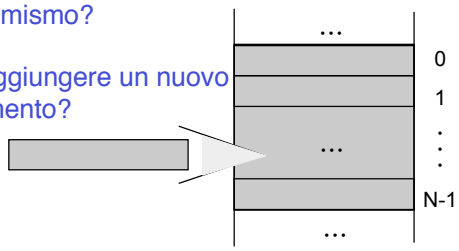
il tipo di dato *lista*

- motivazione : rappresentare sequenze “dinamiche” di elementi omogenei

  - ***sequenza*** **:** insieme di elementi **linearmente ordinato**
    - 1°, 2°, 3°, ... , n°, …
  - ***dinamica*** **:** composizione **variabile** nel tempo
    - inserzione/rimozione in qualsiasi posizione
      - p.es. per mantenere ***ordinata*** la sequenza
  - ***omogenea*** **:** elementi della sequenza dello **stesso tipo**

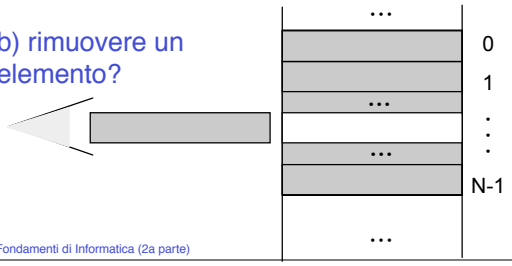
- come rappresentare una lista in C++ ?
  - quale meccanismo?

- meccanismo array?
  - elementi omogenei
  - sequenza "naturalmente ordinata"
    - indici dell'array
  - dinamismo?
  - a) aggiungere un nuovo elemento?



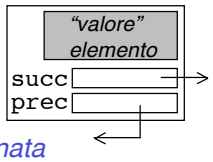
operazioni "costose"

  - non c'è spazio
  - rimedio: "sovradimensionare" l'array
  - b) rimuovere un elemento?

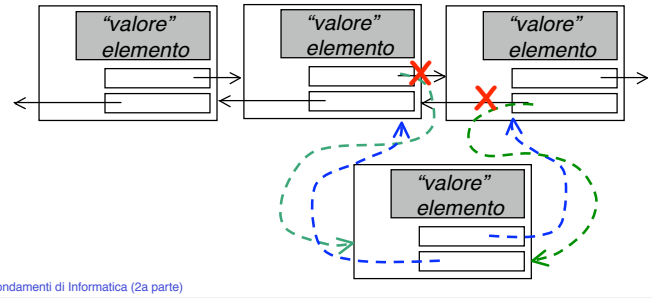


  - si crea un "buco"
  - rimedio: traslare parte dell'array

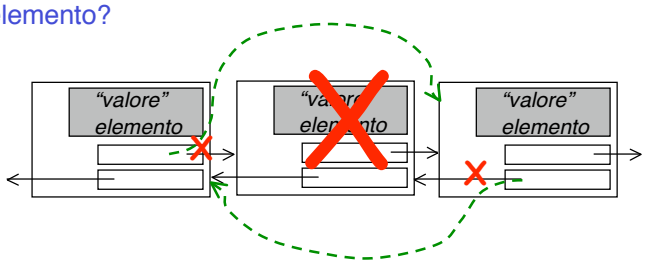
- meccanismo puntatori?



  - elementi omogenei
  - sequenza *naturalmente ordinata*
    - relazioni di *precedente/successore*
  - dinamismo?
  - a) aggiungere un nuovo elemento?



– b) rimuovere un elemento?



definizione del tipo di dato *lista* (e del tipo di dato *iteratore*)

T : insieme dei possibili "valori"
P : insieme di "iteratori"

un iteratore è un "indicatore" di un elemento in una lista

- operazioni del tipo di dato *iteratore* :

get : P → T

get(*p*) restituisce il valore dell'elemento indicato da *p*

next : P → P

next(*p*) restituisce un iteratore che indica l'elemento successivo a *p*

previous : P → P

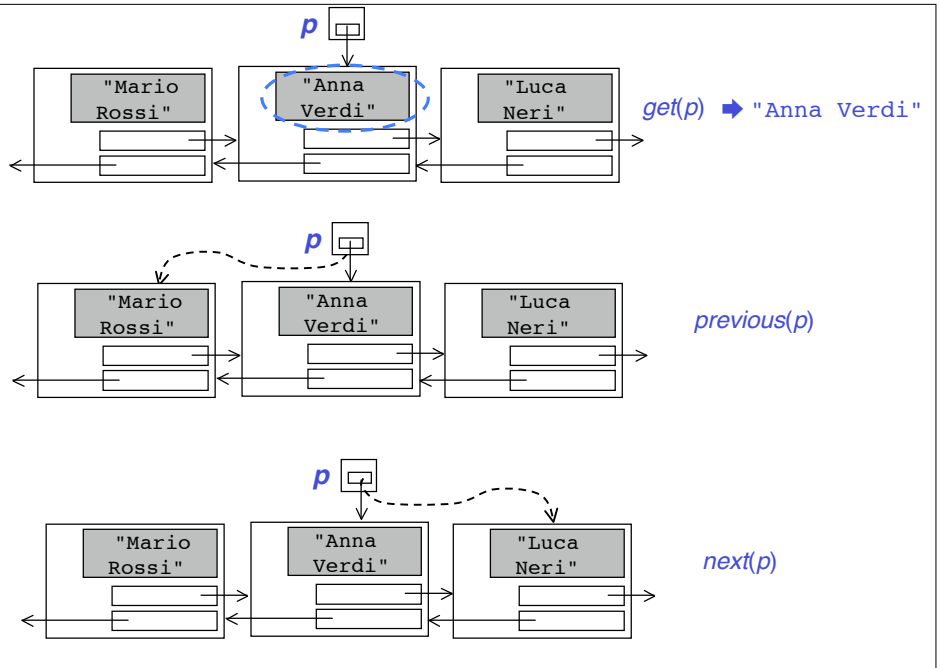
previous(*p*) restituisce un iteratore che indica l'elemento precedente a *p*

equal : P x P→ Bool

equal(*p1*, *p2*) vale vero se *p1* e *p2* indicano lo stesso elemento, falso altrimenti

is_null : P → Bool Nota: *is_null*(*p*) non definita nel libro (cap. 16)

is_null(*p*) vale vero se *p* non indica nessun elemento



- operazioni del tipo di dato *lista* :

push_back : lista(T) x T → lista(T)

push_back(*l*, *t*) inserisce *t* dopo l'ultimo elemento di *l*

insert : lista(T) x P x T → lista(T)

insert(*l*, *p*, *t*) inserisce *t* prima dell' elemento di *l* indicato da *p*

erase : lista(T) x P → lista(T)

erase(*l*, *p*) elimina da *l* l'elemento indicato da *p*

empty : lista(T) → Bool Nota: *empty*(*l*) non definita nel libro (cap. 16)

empty(*l*) vale "vero" se *l* è vuota, "falso" altrimenti

begin : lista(T) → P

begin(*l*) restituisce un indicatore al primo elemento di *l*

end : lista(T) → P

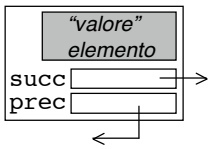
end(*l*) restituisce un indicatore all'ultimo elemento di *l*

Nota: *end*(*l*) definita diversamente nel libro (cap. 16)

realizzazione in C++ del tipo di dato *lista* (e del tipo di dato *iteratore*)

- assumiamo T = insieme di stringhe
- (➜ liste di stringhe)

- realizzazione in C++ di un elemento della lista:




```
class Node
 {public:
   Node(string s);
  private:
   string data;
   Node* prec;
   Node* succ;
   friend class List;
   friend class Iterator;
 };
```

costruttore

"valore"

puntatori

una classe qualificata come "friend" ha il diritto di accedere alla parte privata della classe che concede la qualifica

- qui `List` e `Iterator` hanno il diritto di accedere alla parte privata di `Node`

➡ meccanismo da usare con cautela !!!

- realizzazione in C++ del tipo di dato *iteratore* :

```
class Iterator
 {public:
   Iterator();
   string get() const;
   Iterator next() const;
   Iterator previous() const;
   bool equals(Iterator b) const;
   bool is_null() const;
  private:
   Node* position;
  friend class List;
 };
```

costruttore

funzioni membro

"indicatore" di un elemento della lista

Nota: nel libro (cap. 16) è definito anche un altro attributo privato `Node* last;`

- realizzazione in C++ del tipo di dato *lista* :

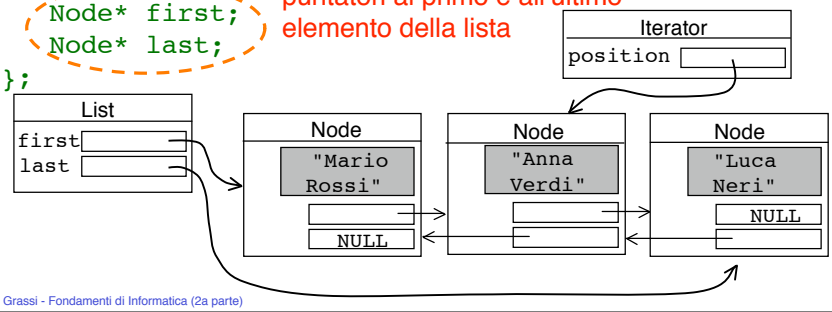
```
class List
{public:
   List();
   void push_back(string s);
   void insert(Iterator p, string s);
   void erase(Iterator p);
   bool empty() const;
   Iterator begin();
   Iterator end();
 private:
   Node* first;
   Node* last;
};
```

costruttore

funzioni membro

Nota: nel libro (cap. 16) `erase()` è di tipo `Iterator`

puntatori al primo e all'ultimo elemento della lista



esempio d'uso del tipo di dato *lista*

```
...
List elenco;
int num_nomi;
Iterator pos;
string nome, residuo;
int c;

cout<<"scrivi numero nomi da inserire in elenco \n";
cin>>num_nomi;
for (int i=1;i<=num_nomi;i++)
  {leggi nome
   inserisci nome in elenco, in ordine alfabetico;
  }

leggi nome;
inserisci nome in 4a posizione;

rimuovi il nome in 2a posizione;

leggi nome;
verifica se è presente in elenco;

stampa tutti i nomi in elenco
```

- inserzione in ordine alfabetico (1)

inserzione in ultima posizione

```
void inserisci_in_ordine(List& l, string s)
 {Iterator i;
  • se l è vuota inserisci s
    altrimenti
      {• posiziona i all'inizio della lista;
       • avanza i finchè non trovi un elemento maggiore di s;
       • se i indica un elemento maggiore di s
            inserisci s prima dell'elemento indicato da i;
        altrimenti inserisci s dopo l'elemento indicato da i;
      }
 }
```

```
void inserisci_in_ordine(List& l, string s)
 {Iterator i;
  if (l.empty()) l.push_back(s);
  else {i = l.begin();
        while (!i.equals(l.end()) && i.get()<=s)
           i = i.next();
        if (i.get()>s) l.insert(i,s);
        else l.push_back(s);
       }
 }
```


- inserzione in ordine alfabetico (2)

```
cout<<"scrivi numero nomi da inserire in elenco \n";
cin>>num_nomi;
for (int i=1;i<=num_nomi;i++)
  {leggi nome;
   inserisci nome in elenco, in ordine alfabetico;
  }
```

```
cout<<"scrivi numero nomi da inserire in elenco \n";
cin>>num_nomi;
getline(cin, residuo);
for (int i=1; i<=num_nomi; i++)
  {getline(cin, nome);
   inserisci_in_ordine(elenco, nome);
  }
```

risolve un problema di "coabitazione" tra >> e getline() (vedi libro, sez. 6.1)

- inserzione in 4a posizione

  - *leggi nome;*
  - *inserisci nome in 4a posizione;*

  - *leggi nome;*
  - *posiziona l'iteratore* pos *all'inizio della lista;*
  - *avanza* pos *fino alla 4a posizione*
  - *inserisci il nuovo nome prima dell'elemento indicato da* pos

```
cout<<"scrivi nome\n"; getline(cin, nome);
pos = elenco.begin(); c = 1;
while (c<4 && !pos.equals(elenco.end()))
   {pos = pos.next(); c++;}
if (c==4)
   elenco.insert(pos, nome);
else if (c==3)
        elenco.push_back(nome);
     else cout<<"lista con meno di 3 elementi \n";
```


- rimozione dalla 2a posizione

rimuovi il nome in 2a posizione;

  - *posizione l'iteratore* pos *all'inizio della lista;*
  - *avanza* pos *fino alla 2a posizione*
  - *rimuovi l'elemento indicato da* pos

```
pos = elenco.begin(); c = 1;
while (c<2 && !pos.equals(elenco.end()))
   {pos = pos.next(); c++;}
if (c==2)
   elenco.erase(pos);
else cout<<"lista con meno di 2 elementi \n";
```

- verifica presenza
  - *leggi nome;*
  - *verifica se è presente in elenco;*

  - *leggi nome;*
  - ***se** la lista non è vuota*
    - *posizione l'iteratore* pos *all'inizio della lista;*
    - *scorri la lista, confrontando il nome con il valore dell'elemento indicato da* pos*;*

```
cout<<"scrivi nome\n"; getline(cin, nome);
pos = elenco.begin();
if (!elenco.empty())
   {while (!pos.equals(elenco.end()) && pos.get()!=nome)
         pos = pos.next();
    if (pos.get()==nome)
       cout<<nome<<" presente nella lista \n";
    else cout<<nome<<" non presente nella lista \n";
    }
else cout<<" lista vuota \n";
```


- stampa della lista

stampa tutti i nomi in elenco

- ***se** la lista non è vuota*
  - *posizione l'iteratore* pos *all'inizio della lista;*
  - *scorri la lista, e stampa il valore dell'elemento indicato da* pos*;*

```
if (!elenco.empty())
   {pos = elenco.begin();
    while (!pos.is_null())
      {cout << pos.get() << "\n"; pos = pos.next();
      }
   }
```

```
if (!elenco.empty())
 {for (pos=elenco.begin();!pos.is_null();pos=pos.next())
     cout << pos.get() << "\n";
 }
```

versione alternativa

realizzazione in C++ del tipo di dato *nodo*

```
class Node
 {public:
   Node(string s);
  private:
   string data;
   Node* prec;
   Node* succ;
   friend class List;
   friend class Iterator;
 };
```

costruttore

"valore"

costruttore :

```
Node::Node(string s)
  {data = s;
   prec = NULL;
   succ = NULL;
  }
```


realizzazione in C++ delle funzioni membro del tipo di dato *iteratore*

```
class Iterator
 {public:
   Iterator();
   string get() const;
   Iterator next() const;
   Iterator previous() const;
   bool equals(Iterator b) const;
   bool is_null() const;
  private:
   Node* position;
   friend class List;
 };
```

costruttore

funzioni membro

"indicatore" di un elemento della lista

costruttore :

```
Iterator::Iterator()
  {position = NULL;
  }
```

get : P → T

get(*p*) restituisce il valore dell'elemento indicato da *p*

```
string Iterator::get() const
  {
   assert(position != NULL);
   return position->data;
  }
```

se `position==NULL`,non è lecito accedere ai campi dell'oggetto riferito (non esiste)

equals : P x P→ Bool

equals(*p1*, *p2*) vale vero se p1 e p2 indicano lo stesso elemento, falso altrimenti

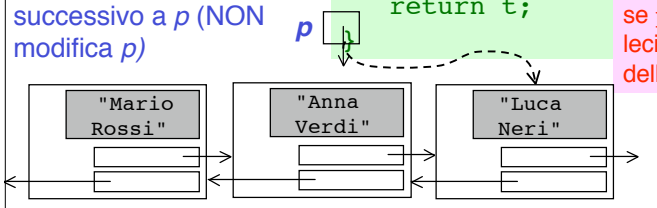
```
bool Iterator::equals(Iterator b) const
  {
   return position == b.position;
  }
```


next : P → P

next(*p*) restituisce l' "indicatore" all'elemento successivo a *p* (NON modifica *p*)

```
Iterator Iterator::next() const
{  assert(position != NULL);
   Iterator t;
   t.position=position->succ;
   return t;
}
```

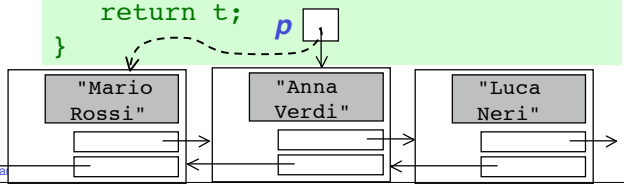
se `position==NULL`,non è lecito accedere ai campi dell'oggetto riferito (non esiste)



previous : P → P

previous(*p*) restituisce l' "indicatore" all'elemento precedente a *p* (NON modifica *p*)

```
Iterator Iterator::previous() const
{   assert(position != NULL);
   Iterator t;
   t.position=position->prec;
   return t;
}
```



is_null : P → Bool

is_null(*p*) vale vero se *p* non indica nessun elemento
falso altrimenti

```
bool Iterator::is_null() const
  {return position==NULL;
  }
```

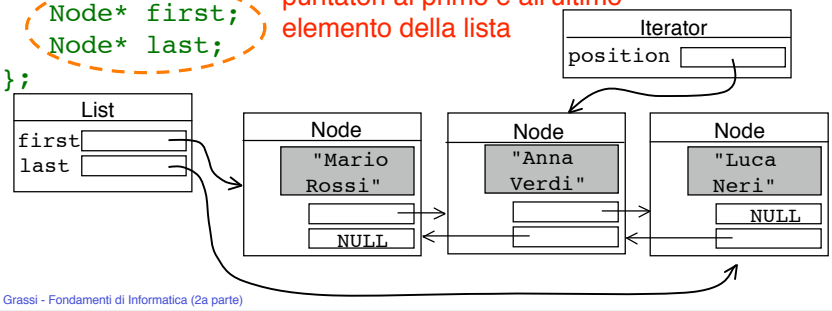

realizzazione in C++ delle funzioni membro del tipo di dato *lista*

```
class List
{public:
   List();
   void push_back(string s);
   void insert(Iterator p, string s);
   void erase(Iterator p);
   bool empty() const;
   Iterator begin();
   Iterator end();
 private:
   Node* first;
   Node* last;
};
```

costruttore

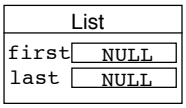
funzioni membro

puntatori al primo e all'ultimo elemento della lista



- costruttore :

```
List::List()
  {first = NULL;
   last = NULL;
  }
```



- *empty* : lista(T) → Bool

  empty(*l*) vale "vero" se *l* è vuota, "falso" altrimenti

```
bool List::empty() const
  { return first==NULL; }
```


- *begin* : lista(T) → P

  begin(*l*) restituisce un indicatore al primo elemento di *l*

```
Iterator List::begin()
  {Iterator i;
   i.position = first;
   return i;
  }
```

- *end* : lista(T) → P

  end(*l*) restituisce un indicatore all'ultimo elemento di *l*

```
Iterator List::end()
  {Iterator i;
   i.position = last;
   return i;
  }
```

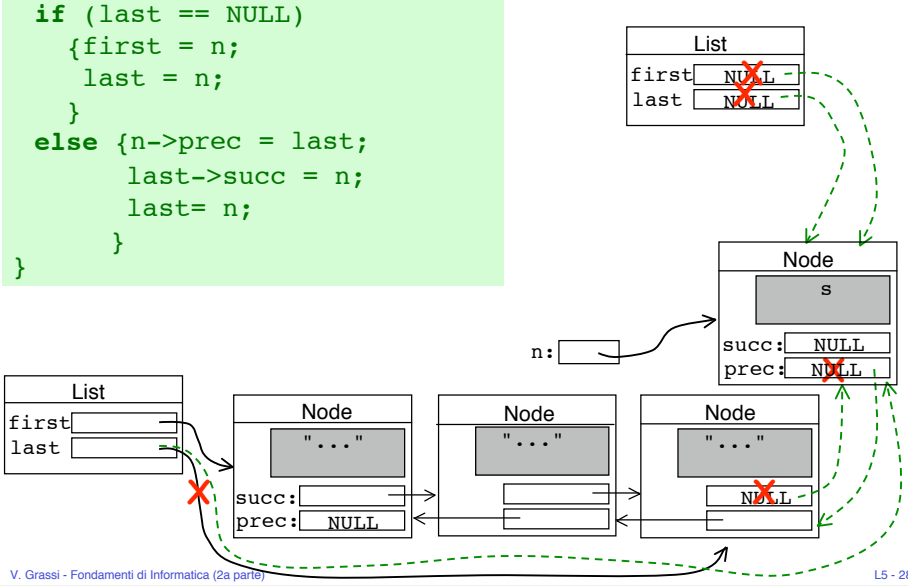
- *push_back* : lista(T) x T → lista(T)

  push_back(*l*, *t*) inserisce *t* **dopo** l'**ultimo** elemento di *l*

```
void List::push_back(string s)
{ crea un nuovo nodo (che contiene s)
  se (lista vuota)
    caso particolare
  altrimenti
    inseriscilo in ultima posizione
}
```



```
void List::push_back(string s)
{ Node* n = new Node(s);
  if (last == NULL)
    {first = n;
     last = n;
    }
  else {n->prec = last;
        last->succ = n;
        last= n;
       }
}
```



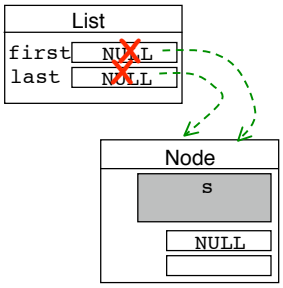
• *insert* : lista(T) x P x T → lista(T)

insert(*l*, *p*, *t*) inserisce *t* **prima** dell' elemento di *l* indicato da *p*

```
void List::insert(Iterator p, string s)
{...}
```

- **caso particolare**: lista vuota …

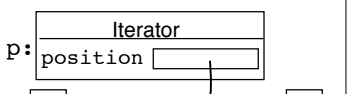
```
if (empty())
    push_back(s);
else ...
```

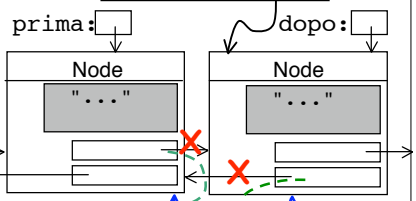


- **… altrimenti**:

1) si crea il nodo da inserire, e si creano puntatori ai nodi che stanno prima e dopo il punto di inserzione (➡ facilita le operazioni)

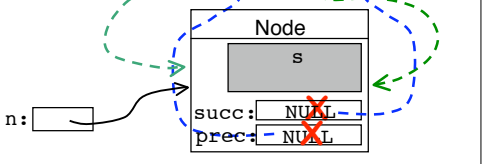
```
Node* n = new Node(s);
Node* dopo = p.position;
Node* prima = dopo->prec;
```





2) si "aggancia" il nuovo nodo alla lista

```
n->prec = prima;
n->succ = dopo;
```



3) si aggiornano i due nodi che precedono e seguono il nuovo nodo

```
dopo->prec = n;
if (prima==NULL)
  first = n;
else prima->succ = n;
```

se **p** indica il 1° elemento? occorre controllare se **p** indica l'ultimo elemento?

mettendo tutto insieme:

• *insert* : lista(T) x P x T → lista(T)

```
void List::insert(Iterator p, string s)
{if (empty())
    push_back(s);
 else {Node* n = new Node(s);
       Node* dopo = p.position;
       Node* prima = dopo->prec;
       n->prec = prima;
       n->succ = dopo;
       dopo->prec = n;
       if (prima==NULL)
          first = n;
       else prima->succ = n;
      }
}
```


• *erase* : lista(T) x P → lista(T)

erase(*l*, *p*) elimina da *l* l'elemento indicato da *p*

```
void List::erase(Iterator p)
{...}
```

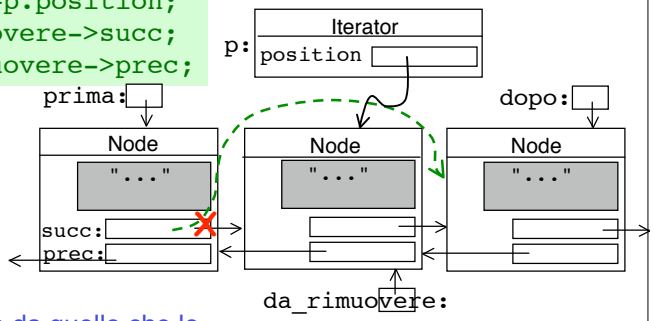
- **premessa**: *p* deve indicare un elemento …

```
assert (p.position != NULL);
```

- **... quindi ...** :

1) si creano puntatori al nodo da rimuovere, e ai due nodi che stanno prima e dopo di lui (➡ facilita le operazioni)

```
Node* da_rimuovere=p.position;
Node* dopo=da_rimuovere->succ;
Node* prima=da_rimuovere->prec;
```



2) si disconnette il nodo da quello che lo precede

oppure: `if (da_rimuovere==first)`

```
if (prima==NULL)
   first = dopo;
else prima->succ = dopo;
```

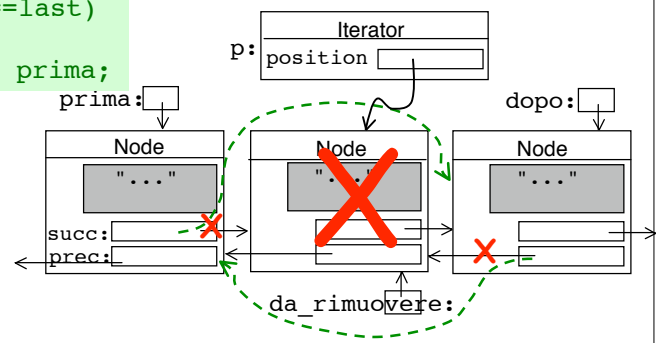
se **p** indica il 1° elemento?

- **... infine ...** :

3) si disconnette il nodo da quello che lo segue (operando analogamente al punto 2)

```
if (da_rimuovere==last)
   last = prima;
else dopo->prec = prima;
```

se **p** indica l'ultimo elemento?



4) si cancella il nodo rimosso

```
delete da_rimuovere;
```

NOTA: **p** ora indica un elemento inesistente !!!

mettendo tutto insieme:

- *erase* : lista(T) x P → lista(T)

```
void List::erase(Iterator p)
{assert (p.position != NULL);
 Node* da_rimuovere = p.position;
 Node* dopo = da_rimuovere->succ;
 Node* prima = da_rimuovere->prec;
 if (prima==NULL)
    first = dopo;
 else prima->succ = dopo;
 if (da_rimuovere==last)
    last = prima;
 else dopo->prec = prima;
 delete da_rimuovere;
}
```