UNIVERSITY OF TRENTO - Italy

Laboratorio 22

DISI – aa 2024/25

Pierluigi Roberti

Carmelo Ferrante

Liste Semplicemente Concatenate Generali e Ordinate

Università degli Studi di Trento - Dipartimento DISI

Liste concatenate

- Quando dobbiamo scandire una collezione di oggetti in modo sequenziale e non sequenziale, un modo conveniente per rappresentarli è quello di organizzare gli oggetti in un array.
- Esempi:
  - (1, 2, -3, 5, -10) è una sequenza di interi.
  - (‘a’, ‘d’, ‘1’, ‘F’) è una sequenza di caratteri.
- Una soluzione alternativa all’uso di array per rappresentare le liste quando l’accesso non sequenziale non è un requisito, consiste nell’uso delle cosiddette *liste concatenate*.
- In una lista concatenata i vari elementi che compongono la sequenza di dati sono rappresentati in zone di memoria che possono anche essere distanti fra loro (al contrario degli array, in cui gli elementi sono consecutivi).
- In una lista concatenata, ogni elemento contiene informazioni necessarie per accedere all’elemento successivo.

Liste concatenate

- Vantaggi rispetto all'array:
  - Flessibilità di modifica.
  - Memorizzo solo quello che mi serve. Mentre con array potrei sprecare memoria (array sovradimensionato).
- Svantaggi rispetto all'array:
  - Lo svantaggio principale consiste nell'onerosità nell'accesso ai suoi elementi.
    - Unico modo per raggiungere un dato elemento consiste nello scorrere la lista dal nodo iniziale.

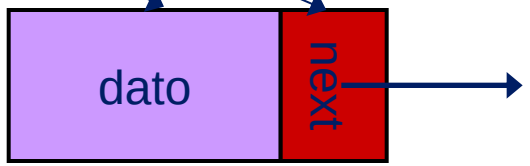
Liste semplicemente concatenate

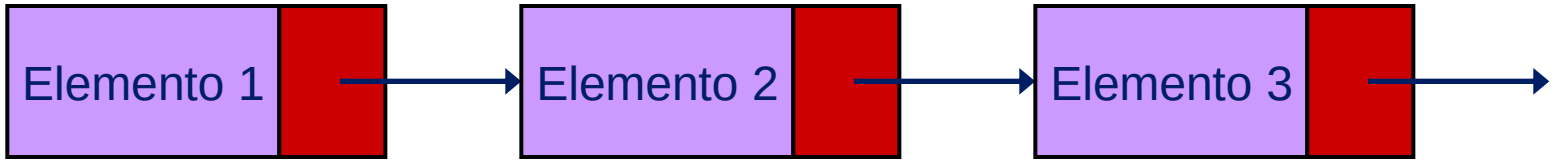
- Una *lista concatenata* è un insieme di oggetti, dove ogni oggetto è inserito in un *nodo* contenente anche un *link* ad un altro nodo.

```
// Lista concatenata di interi
typedef struct Tnodo {
    int dato;
    Tnodo* next;
}Tnodo;
```

Campo contenente le informazioni da memorizzare nel nodo

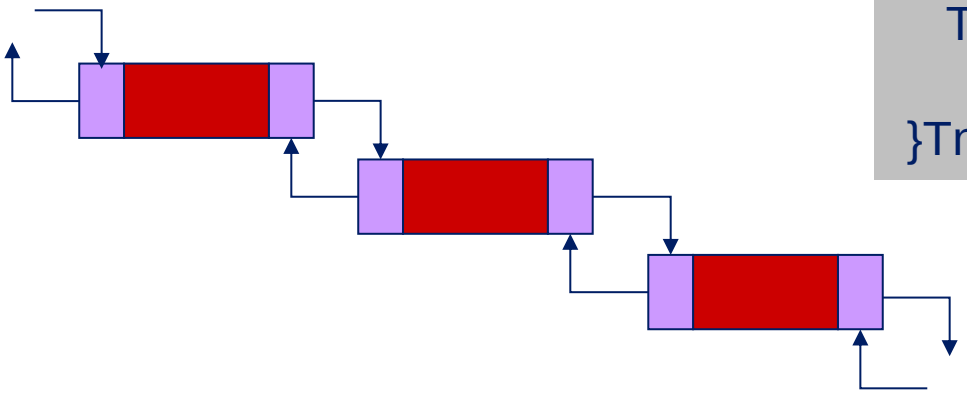
Puntatore al nodo successivo.





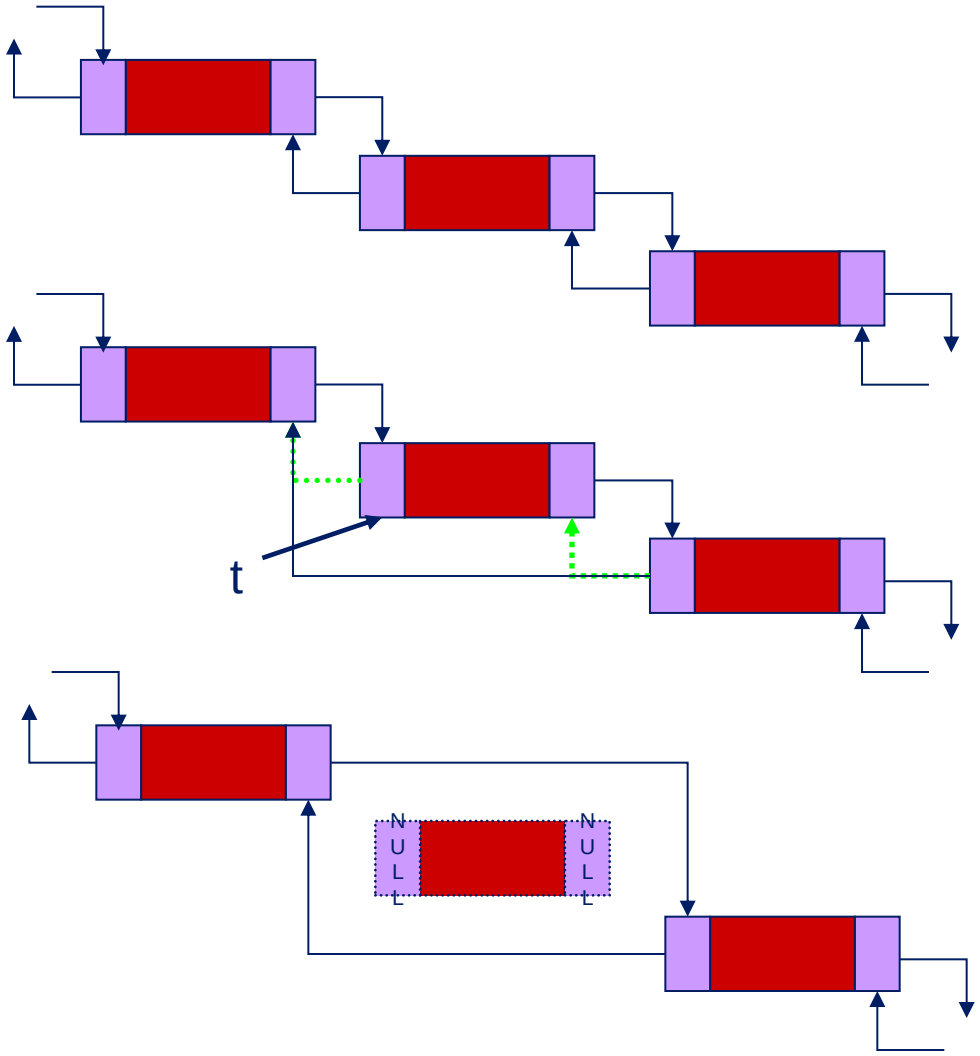
Liste doppiamente concatenate

- Derivano dalle liste definite in precedenza.
  - differiscono per la presenza di un ulteriore puntatore al nodo che lo precede.

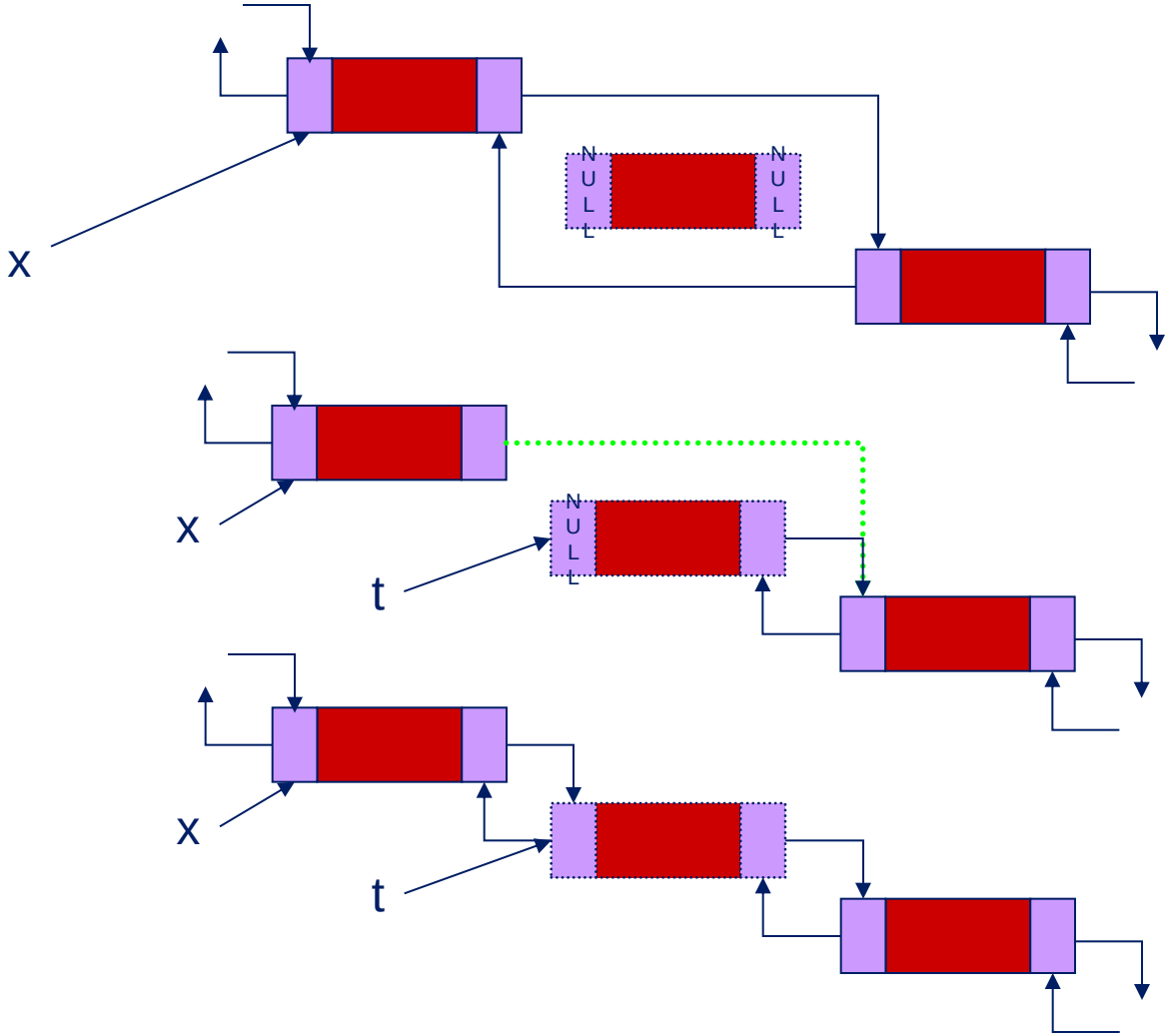


```
typedef struct Tnodo {
   int n;
   Tnodo* prev;
   Tnodo* next;
   Tnodo(int x, node * p, Tnodo* n) {
      n = x; prev = p; next = n; }
}Tnodo;
```

Cancellazione in una lista doppiamente collegata



Inserimento in una lista doppiamente collegata



Strutture di base

```
typedef struct Tdato {
    int index;
    float value;
} Tdato;
typedef Tdato Dato;
typedef struct Tnodo {
    Tdato dato;
    Tnodo *next;
    Tnodo *prev;
} Tnodo;
typedef Tnodo Nodo;
typedef Tnodo* NodoPtr;
```

Aggiungere alle 2 struct

- Costruttore 0 parametri
- Costruttore specifico
- Distruttore
- Metodo di stampa

Strutture di base - Confronto

```
// Lista semplice
typedef struct Tdato {
    int index;
    float value;
} Tdato;
typedef Tdato Dato;

typedef struct Tnodo {
    Tdato dato;
    Tnodo *next;
} Tnodo;
typedef Tnodo Nodo;
typedef Tnodo* NodoPtr;
```

```
//Lista doppiamente concatenata
typedef struct Tdato{
    int index;
    float value;
} Tdato;
typedef Tdato Dato;

typedef struct Tnodo {
    Tdato dato;
    Tnodo *next;
    Tnodo *prev;
} Tnodo;
typedef Tnodo Nodo;
typedef Tnodo* NodoPtr;
```

NodoPtr insertFirst (NodoPtr s, Tdato d)

// Lista semplice

```
NodoPtr q = new Tnodo();
q->dato = d;
q->next = s;
return q;
```

//Lista doppiamente concatenata

```
NodoPtr q = new Tnodo();
q->dato = d;
q->next = s;
q->prev = NULL;
if(s != NULL){
  s->prev = q;
}
return q;
```

NodoPtr insertFirst (NodoPtr s, Tdato d)

// Lista semplice

```
NodoPtr q = new Tnodo(d, s);
return q;
```

//Lista doppiamente concatenata

```
NodoPtr q = new Tnodo(d, NULL, s);
if(s != NULL){
  s->prev = q;
}
return q;
```

NodoPtr insertLast (NodoPtr s, Tdato d)

```
// Lista semplice
if(s==NULL){
  return insertFirst(s, d);
}
NodoPtr p = s;
while (p->next!=NULL){
  p=p->next;
}
NodoPtr t = new Tnodo(d, NULL);
p->next = t;
return s;
```

```
//Lista doppiamente concatenata
if (s == NULL) {
  return insertFirst(s, d);
}
NodoPtr p = s;
while (p->next != NULL) {
  p = p->next;
}
NodoPtr t = new Tnodo(d, p, NULL);
p->next = t;
return s;
```

void stampa (NodoPtr s)

```
// Lista semplice
void stampa(NodoPtr s){
        while (s!=NULL){
                s->stampa();
                s = s->next;
        }
        cout << endl;
}
```

```
//Lista doppiamente concatenata
  void stampa(NodoPtr s){
        while (s!=NULL){
                s->stampa();
                s = s->next;
        }
        cout << endl;
}
```

NodoPtr insertLast (NodoPtr s, Tdato d)

```
// Lista semplice
if(s==NULL){
  return insertFirst(s, d);
}
NodoPtr p = s;
while (p->next!=NULL){
  p=p->next;
}
p->next = new Tnodo(d, NULL);
return s;
```

```
//Lista doppiamente concatenata
if (s == NULL) {
  return insertFirst(s, d);
}
NodoPtr p = s;
while (p->next != NULL) {
  p = p->next;
}
p->next = new Tnodo(d, p, NULL);
return s;
```

NodoPtr removeFirst (NodoPtr s)

```
// Lista semplice
  NodoPtr n = s;
  if (s!=NULL){
    s= s->next;
    delete n;
  }
  return s;
```

```
//Lista doppiamente concatenata
  NodoPtr n = s;
  if (s!=NULL) {
    s= s->next;
    if(s!=NULL) {
      s->prev=NULL;
     }
   delete n;
  }
  return s;
```

NodoPtr removeFirst (NodoPtr s)

```
// Lista semplice
  if (s==NULL) {
    return s;
  }
  NodoPtr n = s;
  s= s->next;
  delete n;
  return s;
```

```
//Lista doppiamente concatenata
  if (s==NULL) {
    return s;
  }
  NodoPtr n = s;
  s= s->next;
  if(s!=NULL) {
    s->prev=NULL;
  }
  delete n;
  return s;
```

NodoPtr removeLast (NodoPtr s)

```
// Lista semplice
if (s==NULL){
  return NULL;
}
if (s->next==NULL){
  delete s;  return NULL;
}
NodoPtr p = s;
while (p->next->next!=NULL){
  p=p->next;
}
delete p->next;
p->next = NULL;
return s
```

```
//Lista doppiamente concatenata
if (s==NULL) {
  return s;
}
if (s->next==NULL){
  delete s;  return NULL;
}
NodoPtr p=s;
while(p->next->next!=NULL){
  p = p->next;
}
delete p->next;
p->next = NULL;
return s;
```

Tdato readFirst (NodoPtr s)

```
// Lista semplice
Tdato readFirst(NodoPtr s){
        return s->dato;
}
```

```
//Lista doppiamente concatenata
Tdato readFirst(NodoPtr s){
        return s->dato;
}
```

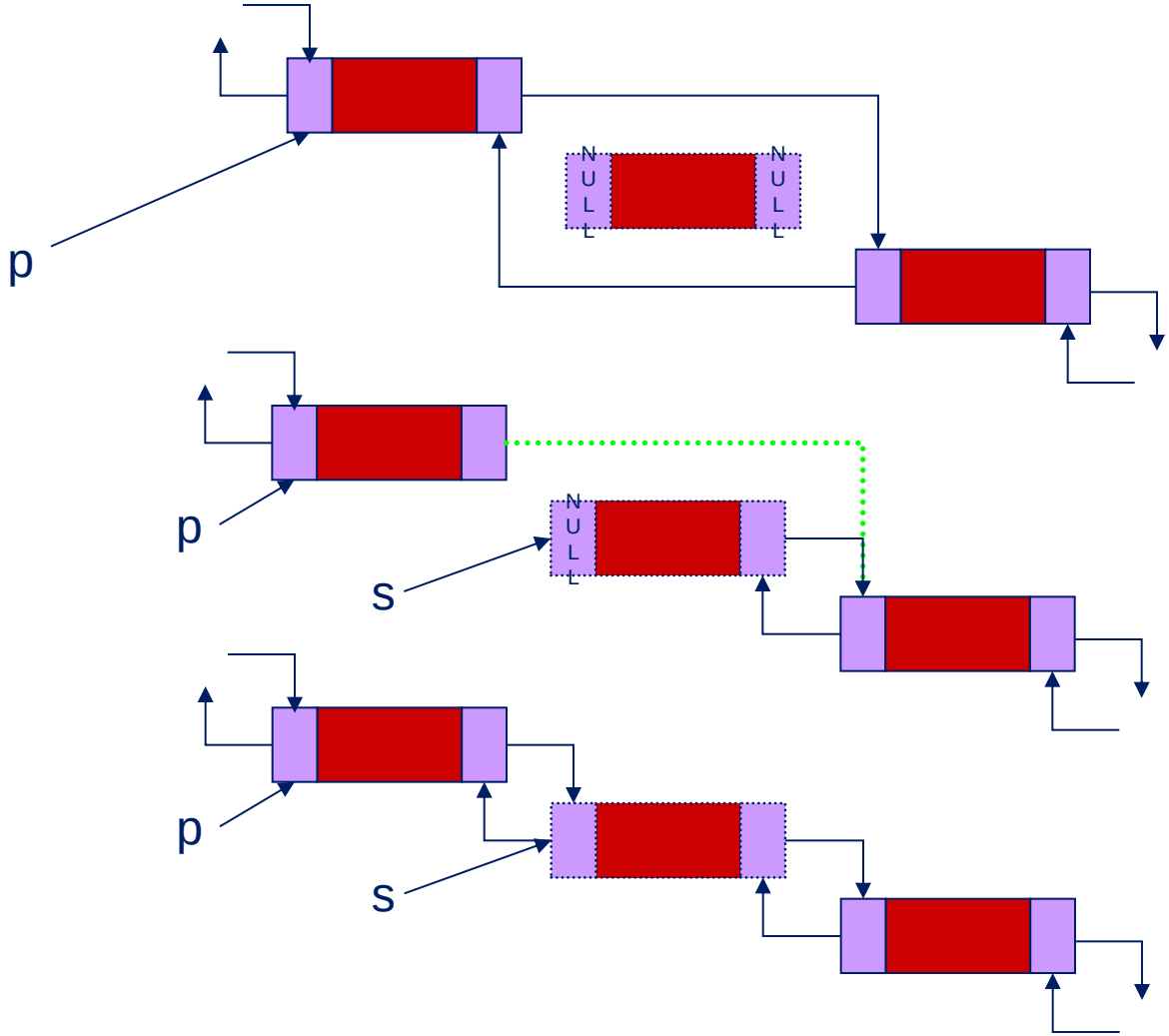
Esercizio 1

- Scrivere la funzione per l'inserimenti di dati nella lista:
  - insertOrder: NodoPtr insertOrder (NodoPtr s, Tdato d);
    (in base a campo index)
- Testare le funzioni nel main come segue:
  - Creare una lista **p** ed una lista **o** (ordinata)
  - Inserire 10 elementi (valori per index casuale tra 1 e 10 e value casuale tra -5.00 e +5.00) nella lista **p** usando **insertLast**
  - Stampare la lista **p**
  - Estrarre tutti gli elementi dalla lista **p** (**readFirst**, **removeFirst**) ed inserirli nella lista **o** (**insertOrder**)
  - Stampare la lista **o**

Modifica strutture di base

```
typedef struct Tdato {
    int index;
    float value;
    bool gt(Tdato d){
      if (index>d.index) {return true;}
      return false;
    }
} Tdato;
```

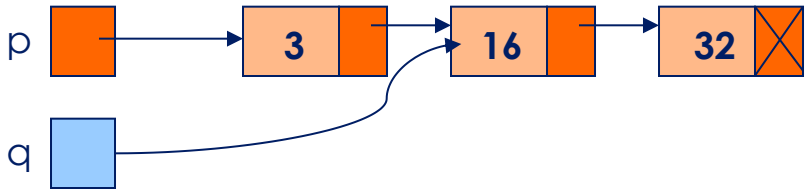
Inserimento in una lista doppiamente collegata



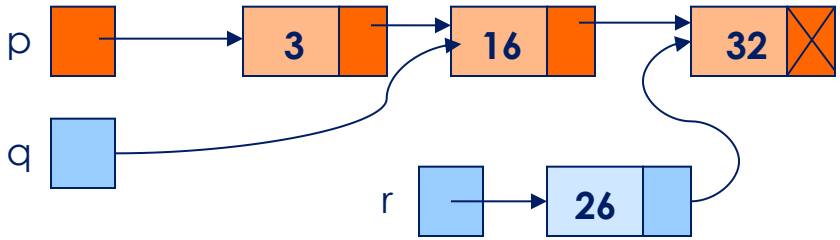
Inserimento di un elemento ordinato

lista semplicemente concatenata

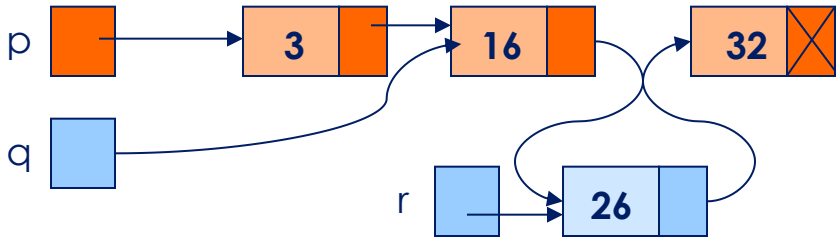
- Vogliamo inserire un nuovo elemento (contenente per esempio il numero ***26***) in una **lista ordinata** (ordine *crescente*).




```
nodo* q=p;
while(q->next->dato<=26){
    q=q->next;
}
```




```
nodo* r=new nodo;
r->dato=26;
r->next=q->next
```




```
q->next = r;
```

Attenzione: dobbiamo considerare i casi limite!

Inserimento di un elemento ordinato

lista semplicemente concatenata

- **Primo caso limite:** dobbiamo inserire l'elemento in testa
  - Perchè la lista è vuota

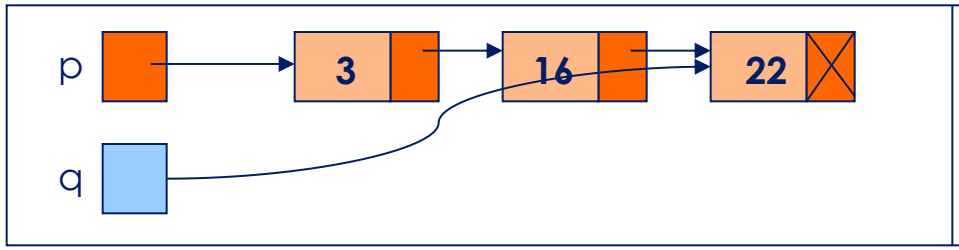
  - oppure perchè tutti gli altri elementi hanno un valore maggiore


```
//26 è il valore da inserire
if( (p==NULL) || (p->dato>= 26) ) {
   insert_first(p, 26);
}
```

Inserimento di un elemento ordinato

lista semplicemente concatenata

- **Secondo caso limite:** dobbiamo inserire l'elemento in coda
  - Perchè tutti gli altri elementi hanno un valore minore.




```
nodo* q=p;
while(q->next->info<=26)
    q=q->next;
```

OCCORRE CONTROLLARE SE È L'ULTIMO ELEMENTO!

```
nodo* q=p;
while( (q->next!=NULL) && (q->next->dato<=26) ) {
   q=q->next;
}
```

Insert Order Liste Doppie – versione 1

```
NodoPtr insertOrder(NodoPtr s, Tdato d){
  bool cond;
  NodoPtr p=s;  //indirizzo iniziale
  if ( ( s==NULL ) || ( s->dato.gt(d) ) )
    { s = insertFirst(s, d);  return s; }
  do {
    cond = true;
    if (s->next!=NULL)  { cond = ( s->next->dato.gt(CurrD) ); }
    s=s->next;
  }  while ((s!=NULL) && (!cond));
  if (s==NULL) //ultimo nodo
   { p=insertLast(p,d); return p;}
  nodoPtr q = new Tnodo(); //creo il nuovo nodo
  q->dato = d;
  q->prev = s->prev;   q->next = s;  //q->s
  s->prev->next = q;    s->prev = q; //q<-s
  return p;
}
```

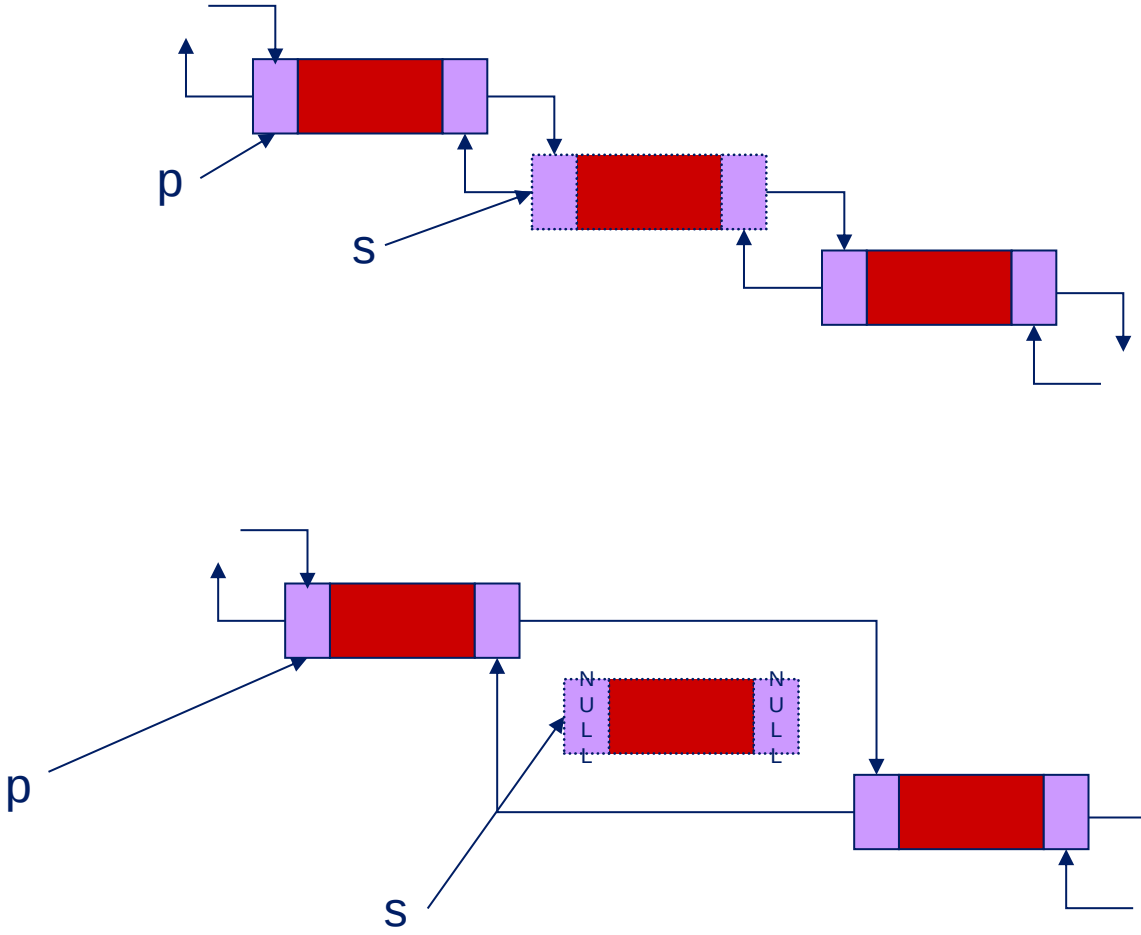
Insert Order Liste Doppie – versione 2

```
NodoPtr insertorder(NodoPtr s, Tdato d){
   NodoPtr q = new Tnodo();
   q->dato = d;
   if (s==NULL) { return q; }  //caso1: lista vuota
   if (! d.gt(s->dato) ) {          //caso2: insertfirst
      q->next = s;
      s->prev = q;
      return q;
   }
   nodoPtr p=s;
   while( p!=NULL ){
      if ( ! d.gt(p->dato) ){ break;}
      p=p->next;
   }
   if (p==NULL){ return insertlast(s,d); } //caso3: insertlast
   q->next = p;                                    //caso4: insert all'interno
   q->prev = p->prev;   p->prev->next = q;   p->prev = q;
   return s;
}
```

Esercizio 2

- Scrivere una funzione che cancelli un elemento *da* una lista doppiamente concatenata *s*
  - Remove Element: NodoPtr removeElem (NodoPtr s, Tdato d); (se presente)
  - Remove Condition: NodoPtr removeCond (NodoPtr s); (valore index pari)
- Testare le funzioni nel main come segue:
  - Creare una variabile **dd** di tipo Dato con index tra 1 e 10 e value tra 0.0 e 2.0
  - Richiamare la **removeElem** per la lista **o** passando la variabile **dd**
  - Stampare la lista **o**
  - Richiamare la **removeCond** per la lista **o**
  - Stampare la lista **o**

Rimozione in una lista doppiamente collegata



Attenzione: dobbiamo considerare i casi limite!

Remove Element Liste Doppie

```
NodoPtr removeElem(NodoPtr s, Tdato d){
  NodoPtr p = s;
  if (s==NULL)  { return s; } //lista vuota
  if (s->next==NULL && s->dato.index == d.index ) { //1 solo elemento
     delete s; return NULL;  }
  //delete 1°  elemento ma lista con più elementi
  if (s->prev==NULL && s->dato.index == d.index ) {
      s = s->next;    s->prev=NULL;
     delete p;          return s;  }
   while (s!=NULL) { //scorro la lista
      if ( s->dato.index == d.index ) {
      s->prev->next = s->next; //il precedente punta al successivo
      //escludo caso limite ultimo elemento lista
       if (s->next != NULL) {
           s->next->prev = s->prev; } //il successivo punta al precedente
        delete s;      return p;
       }
       s=s->next;
     }
    return p;}
```

Esercizio 3

1. Definire un array «**mieListe**» di tipo NodoPtr (liste doppiamente concatenate) di dimensione 3 elementi
2. Definire una lista «**cestino**» di tipo NodoPtr
3. Inizializzare le liste in modo opportuno (vuote)
4. Per 50 volte
    1. Generare un Dato **d** con valori casuali
    2. Generare un numero casuale.
    3. Se PARI: inserire **d** (**insertOrder**) in una lista a caso di array «**mieListe**»
    4. Se DISPARI: scelta una lista a caso di array «**mieListe**», se la coda ha elementi, estrarre un dato (**readFirst** + **removeFirst**) ed inserirlo (**insertLast**) in lista «**cestino**»
    5. Stampare il contenuto delle liste: **mieListe** e **cestino**
5. Cancellare tutte le liste

Usare le funzioni viste in precedenza

Esercizio 4

1. Scrivere una funzione **deleteAll** che cancella tutta la lista (usando la funzione removeFirst).
2. Scrivere una funzione **isPresent** che ritorni true se un elemento *x* occorre nella lista, false se l'elemento non occorre.
3. Scrivere una funzione **copyList** che costruisca una copia di una data lista (cioè, una nuova lista che contiene le stesse informazioni nello stesso ordine).
4. Scrivere una funzione **moveBiggest** che sposti l'elemento più grande di una lista concatenata nell'ultimo nodo della lista.
5. Scrivere una funzione **moveSmallest** che sposti l'elemento più piccolo di una lista concatenata nel primo nodo della lista.
6. Scrivere una funzione **concatList** che costruisca la lista risultante dalla concatenazione di due liste *x* e *y.*
   - Diverse soluzioni sono possibili:
     - Side effects sulla lista destinazione.
     - Nuova lista (usare copyList).