LINKEDLIST: iteratore

- Il metodo `iterator()` restituisce un oggetto iteratore sulla lista per *scandire gli elementi in sequenza a partire dall'inizio della lista*.

```
public Iterator iterator() {
    return new InnerIterator();
}
```

- La classe dell'iteratore è definita come classe interna (`InnerIterator`) alla classe `LinkedList`.

LINKEDLIST: implementazione iteratore

- Vediamo una possibile implementazione dell'iteratore.

- Si usa una variabile `cursor` che individua il nodo il cui contenuto è restituito dalla successiva chiamata a `next()`. La variabile `precursor` coincide con `cursor` tranne quando è possibile eliminare un elemento, nel qual caso precede `cursor` (cioè dopo una chiamata a `next()`).
- Inoltre è utilizzata una variabile intera `expmod`, che tiene conto delle modifiche alla lista durante l'iterazione, per verificare se sono avvenute eventuali modifiche concorrenti.

INNERITERATOR

```
public class LinkedList implements List {
   ...
  class InnerIterator implements Iterator{
      Node cursor;
      Node precursor;
      int expmod;

      public InnerIterator(){
            precursor=cursor=null;
            expmod=mod;
      }
   ...
}
```

Classe interna

INNERITERATOR

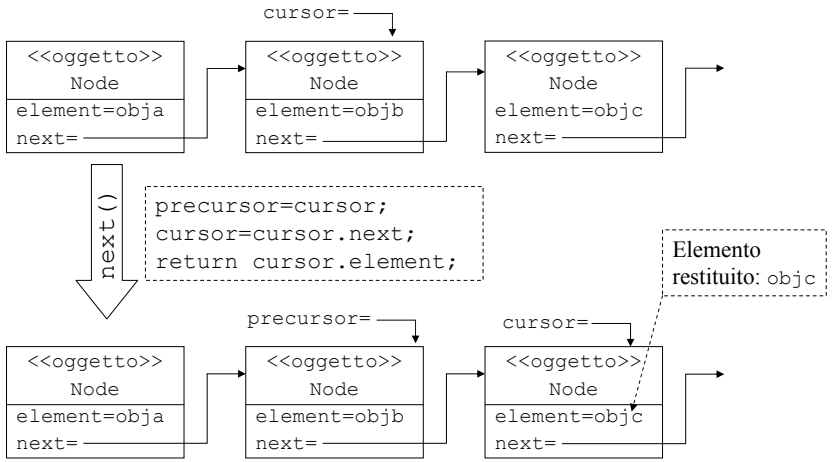
```
public boolean hasNext() {
   if (mod!=expmod)
       throw new IllegalStateException();
   return (!isEmpty()&&((cursor==null)||
                                  (cursor.next!=null)));
}

public Object next() {
   if(!hasNext())
       throw new IllegalStateException();
   precursor=cursor;
   cursor=(cursor==null)?first:cursor.next;
   return cursor.element;
}
```

Modifica concorrente

INNERITERATOR

- `cursor!=null`:



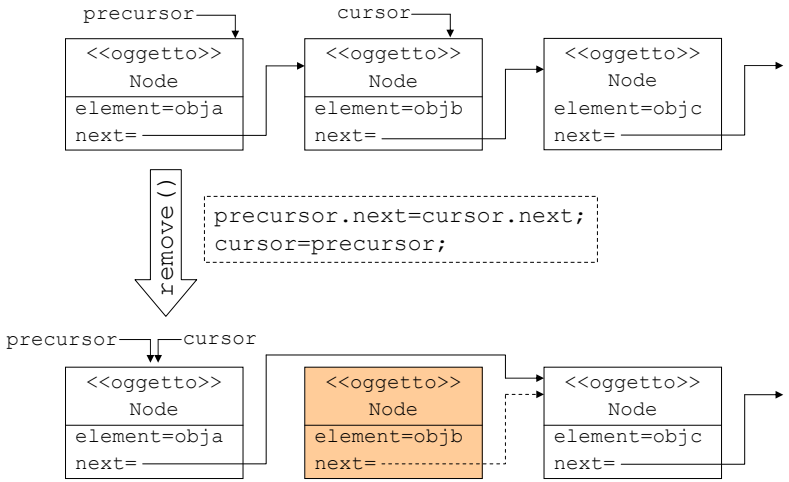
INNERITERATOR

```
public void remove() {
        if(mod++!=expmod++)
                throw new IllegalStateException();
        if(cursor==precursor)
                throw new IllegalStateException();

        if (precursor==null){   ← cursor riferisce il primo elemento
                if(first==last) ← e ultimo
                        last=null;
                first=first.next;
        }else{
                if(cursor==last)
                        last=precursor;
                precursor.next=cursor.next;
        }
        cursor=precursor;
}
```


INNERITERATOR

- `precursor!=null`:



LIST: prestazioni

- Se è necessario un *accesso immediato* a ciascun elemento, allora l'array è la scelta migliore.
- Se si accede frequentemente solo ad alcuni elementi (i primi, gli ultimi) e se la *modifica* della struttura è il cuore di un algoritmo, allora la lista concatenata è la scelta migliore.
- Un vantaggio delle liste è che usano la *quantità minima* di memoria necessaria e sono facilmente ridimensionabili. Al contrario un array può essere modificato in dimensione solo di quantità fisse (per esempio dimezzato o raddoppiato).

LIST: prestazioni

- Vediamo in dettaglio le prestazioni di un array e di una linkedlist. Si devono distinguere due casi: dati ordinati e dati non ordinati.

Elementi non in ordine.

- Con entrambe le soluzioni la ricerca è sequenziale e quindi ha un costo *O(n)*.
- L'inserimento o la rimozione di un dato ha un costo costante *O(1)*: si può inserire in fondo alla lista, dato che non deve essere mantenuto un ordinamento. Nel caso dell'array potrebbe essere necessario un ridimensionamento.

LIST: prestazioni

Elementi ordinati.

- Per un array ordinato si può sfruttare l'algoritmo di ricerca binaria con un costo *O(log n)*, mentre per una lista la ricerca è sequenziale con un costo *O(n)* (non si può accedere direttamente al centro di una lista).
- Inserimento o rimozione di un dato. Per un array è veloce trovare la posizione, *O(log n),* ma è necessario far scorrere gli elementi per mantenere l'ordinamento, costo *O(n)*. Pertanto il costo dell'operazione è lineare, *O(n)*. Inoltre potrebbe essere necessario un ridimensionamento.

LIST: prestazioni

- Inserimento o rimozione di un dato. Per una lista è più oneroso trovare la posizione, *O(n),* ma è una operazione veloce inserire o rimuovere un elemento, costo *O(1)*. Pertanto il costo dell'operazione è lineare, *O(n)*.
- Si deve tener presente che due operazioni che hanno lo stesso costo computazionale asintotico *non* implicano lo *stesso tempo di esecuzione*. La notazione asintotica è un'approssimazione del numero di operazioni base eseguite.

LIST: considerazioni

- Risulta evidente che c'è interesse ad avere delle *tabelle di dati ordinati*. Vediamo come si può affrontare il problema dell'ordinamento per le liste.
- Nel caso delle liste sono utili gli algoritmi che elaborano i dati sequenzialmente, modalità che può essere efficientemente supportata dalle liste concatenate.
- Si può pensare di inserire un metodo `sort()`, che ordini la lista lavorando direttamente sui nodi.

LIST: ordinamento

- Consideriamo l'*ordinamento per selezione* di una lista concatenata.
- Si scandisce la lista per determinare il valore minimo, si rimuove tale elemento dalla lista e lo si inserisce in fondo ad una *nuova lista*.
- Si usano dei nodi ausiliari per poter manipolare la lista. Per la rimozione è conveniente avere il riferimento al nodo che precede quello che contiene il minimo.
- Vediamo una possibile implementazione.

LIST: ordinamento

```
public void sort(){
   Node least, outfirst=null, outlast=null;
   if (isEmpty() || last==first)   return;
   Node i = new Node(new Object(),first);
   while(i.next != null){
        least=i;
        for(Node j=i.next; j.next!=null; j=j.next){
                if
   (((Comparable)j.next.element).compareTo(least.next.element)<0)
                        least=j;
        }
        if (outfirst == null)
                outfirst=outlast=least.next;
        else{
                outlast.next=least.next;
                outlast=outlast.next;
        }
        least.next=least.next.next;
   }
   first = outfirst;
   last = outlast;
   mod++;}
```


LIST: ordinamento

- Un esempio di uso del metodo `sort()`:

```
SortedLinkedList L = new SortedLinkedList();
Random rnd = new Random();
for(int i=0; i<21;i++)
      L.add(new Integer(rnd.nextInt(21)-10));
System.out.print(L);
L.sort();
System.out.print(L);
```

[0, -6, 6, -2, 2, -5, -9, 0, 8, 7, 5, -7, 8, 3, 9, -7, -10, 9, -2, -10, -9]
[-10, -10, -9, -9, -7, -7, -6, -5, -2, -2, 0, 0, 2, 3, 5, 6, 7, 8, 8, 9, 9]

LIST: ordinamento

- In molti casi non si ha bisogno di implementare esplicitamente un algoritmo di ordinamento. Si può *mantenere ordinata la lista* mentre si inseriscono degli elementi: è sufficiente modificare il metodo `add()` in modo che inserisca il nuovo elemento nella corretta posizione.
- In tal caso il costo dell'operazione di inserimento è maggiore perché implica prima una ricerca.

LIST: ordinamento

- Un'altra possibile soluzione è quella di copiare la lista in un array, ordinare l'array (con un costo *O(n log n)*) e iterare sulla lista per mettere i dati ordinati nuovamente in lista. Tale scelta evita di avere un costo $O(n^2 \log n)$ che risulta dall'ordinamento sul posto di una linkedlist.

- Questa implementazione richiede che l'iteratore fornisca un metodo `set()` per sostituire l'ultimo elemento ritornato da `next()`.

LIST: doppiamente concatenate

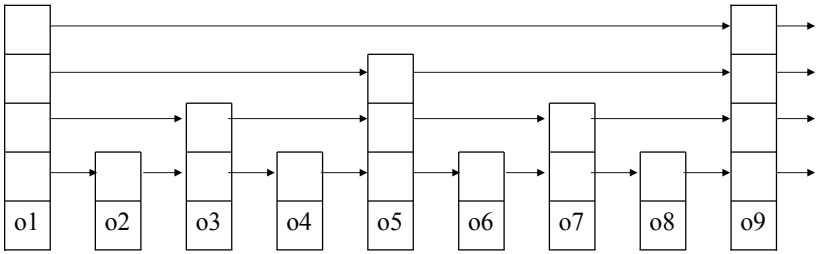
- Esistono altri tipi di lista, vediamone alcuni.

- La lista descritta è di tipo semplicemente concatenata: i nodi hanno informazione solo sul loro successore, per cui non si ha accesso al loro predecessore.
- Si può definire una *lista doppiamente concatenata* in modo che ogni nodo abbia due campi riferimento: uno al successore e uno al predecessore.
- In tal modo è possibile percorrere la lista nei due sensi.

LIST: con salti

- Per trovare un elemento in una lista è necessaria uno *scansione sequenziale*. Anche se la lista è *ordinata* è sempre necessaria una scansione sequenziale.
- Si evita questo problema utilizzando una *lista con salti*: liste che consentano di saltare alcuni nodi per evitare manipolazioni sequenziali.
- La struttura interna è più complessa: nodi con un numero di campi riferimento diverso.

LIST: con salti



- La ricerca può essere efficiente, anche *O(log n)*, tuttavia sono molto inefficienti le procedure di inserimento e cancellazione.