SOMMARIO

- Collezioni:
  - Array e liste.
  - Iteratori.
  - Classi interne.
  - Nodi: dati e riferimenti.

- Liste:
  - LinkedList: specifica e implementazione.
  - Prestazioni.

COLLEZIONI

- Una *collezione* (contenitore) è un oggetto che raggruppa più dati in una singola struttura. Vi sono due tecniche *fondamentali* per rappresentare collezioni di elementi: quella basata su *strutture indicizzate* (*array,* composti da elementi consecutivi) e quella basata su *strutture collegate* (*list*, composte da nodi: ogni nodo contiene dati e collegamenti ad altri nodi).

- Le collezioni sono usate per memorizzare, leggere, manipolare dati strutturati e per trasmettere tali dati da un metodo ad un altro.

ITERATORI

- Una tipica operazione che si esegue su collezioni è esaminare tutti gli elementi uno alla volta. Per gli array è semplice scandire tutti gli elementi attraverso un indice e un ciclo `for`.
- In generale, una generica collezione non ha i dati organizzati in una struttura così rigida e lineare.

- Di conseguenza si introduce il concetto di iteratore, *generalizzando la scansione lineare di un array*.

ITERATORI

- Un *iteratore* è un *oggetto* di una classe che implementa l'interfaccia `IEnumerator`.
- Tale interfaccia contiene:

```
bool MoveNext(); object Current{get;}
void Reset();.
```

- Un iteratore è restituito da una collezione sulla quale si è invocato il metodo:

```
IEnumerator GetEnumerator();
```

  descritto dall'interfaccia `IEnumerable`.
- Tale iteratore consente di scandire i dati della collezione usando i metodi dell'interfaccia.

ITERATORI

- Il metodo `MoveNext()` sposta l'iteratore sul successivo elemento della collezione (al momento della creazione l'iteratore precede il primo elemento), restituisce `true` se l'operazione è andata a buon fine, `false` se si è superata la fine della collezione.
- La proprietà `Current` fornisce l'elemento corrente nella collezione (*solo in lettura*).
- Il metodo `Reset()` riposiziona l'iteratore nella posizione iniziale: precedente il primo elemento della collezione.
- Un iteratore è *valido* solo se la collezione *non* è *modificata* mentre lo si usa.

ITERATORI

- Sfruttando questo strumento possiamo scrivere una porzione di codice per accedere a tutti gli elementi di una *qualsiasi collezione*.
- Vediamo un esempio con la collezione array.

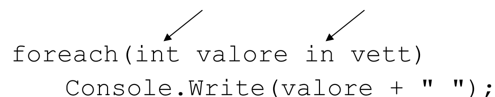
```
int[] vett = { 1, 3, 5, 2, 4, 6 };

IEnumerator iter = vett.GetEnumerator();

while (iter.MoveNext())
    Console.Write(iter.Current + " ");
```

Una possibile uscita

1 3 5 2 4 6

ITERATORI

- In C# vi è la parola chiave `foreach` che permette di accedere in modo semplificato agli iteratori delle collezioni:



```
foreach(int valore in vett)
    Console.Write(valore + " ");
```

- Da questo esempio si può intuire la potenza degli iteratori. Tuttavia la loro implementazione ci porta a considerare le *classi interne*. Perché è un oggetto che può operare sui dati privati della collezione ed è ritornato da un metodo della classe della collezione, inoltre si possono creare più oggetti iteratore sulla stessa collezione.

CLASSI INTERNE

- Una classe interna (*inner class*) è definita internamente ad un'altra classe (*outer class*).
- La classe interna può accedere a tutti i membri della classe esterna.
- Di solito è una *helper class* che non è esplicitamente utilizzata all'esterno.
- Di solito la classe interna implementa un'interfaccia che permette di invocare dall'esterno i metodi della classe interna attraverso il riferimento creato da un metodo della classe che la include.

CLASSI INTERNE

- Il fatto di creare una classe interna, invece di creare un'unica classe, ci permette di creare più oggetti della classe interna che operano sullo stesso oggetto della classe esterna.

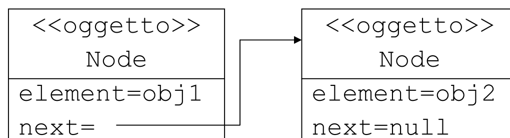
- Un esempio di classe interna è mostrato nella realizzazione di un iteratore per una collezione.

NODI

- Fino ad ora si sono usate le strutture dati array (semplici e performanti). La loro caratteristica è di avere celle memorizzate sequenzialmente che sono accessibili attraverso indici con costo *O(1)*.
- Tuttavia si può pensare di usare strutture dati in cui l'*ordinamento* dei dati è solo *a livello logico* (senza usare un sottostante ordinamento fisico).
- Questo si può realizzare associando ad ogni elemento anche un *riferimento* all'elemento che lo segue nella sequenza.

NODI

- La struttura dati che sfrutta tale idea è la *lista*.
- Per ora vediamo di realizzare questa idea di *elementi più riferimenti*: un oggetto (nodo) che contenga, oltre all'informazione sull'elemento memorizzato, anche un riferimento al nodo che contiene l'elemento che nella sequenza lo segue.
- Esempio di due nodi:



NODI

```
class Node
  {
     object Element;

     Node Next;
```

I due campi sono `private` per default.

La dichiarazione *ricorsiva* (il campo `next` è di tipo `Node`) di un nodo può sembrare pericolosa, ma è corretta. Ciò che dichiara è un riferimento ad un oggetto di tipo `Node` e non un oggetto stesso.

```
     public Node(object e)
     {
        if (e == null)
           throw new Exception("Elemento null.");
        Element = e;
        Next = null;
     }
                                   ...
```

NODI

```
…
public Node(object e, Node n)
{
    if (e == null)
        throw new Exception("Elemento null.");
    Element = e;
    Next = n;
}

public Node next
{
    get { return Next; }
    set { Next = value; }
}
public object element
{
    get { return Element; }
    set { Element = value; }
}
}
```


NODI

- Vediamo un semplice uso di tale oggetto:

```
Node n1, n2;
n1 = new Node("Primo");
n2 = new Node("Secondo");
n1.next = n2;
Console.WriteLine(n1.element);
Console.WriteLine(n1.next.element);
```

Attraverso `n1` si accede al contenuto di `n2`.

Una possibile uscita

```
Primo
Secondo
```


LISTE: introduzione

- Gli array presentano due limitazioni:
  - Modificare la dimensione dell'array richiede la creazione di un nuovo array e la copia di tutti i dati dall'array originale a quello nuovo.
  - Inserire un elemento nell'array richiede lo scorrimento di dati nell'array.
- Queste limitazioni possono essere superate da *strutture concatenate*: una collezione di nodi che contengono dati e riferimenti ad altri nodi. In particolare si fa riferimento alle *liste concatenate* (*linked list*).

LISTE: introduzione

- Uno svantaggio delle liste concatenate consiste nell'onerosità dell'accesso ai suoi elementi: l'unico modo per raggiungere un elemento della lista è quello di seguire le connessioni della lista dall'inizio in modo *sequenziale*. Mentre negli array è possibile accedere a qualsiasi elemento attraverso un indice.
- I metodi di manipolazione delle liste sono diversi da quelli visti per gli array.
- In generale, una lista rende più semplice la riorganizzazione degli elementi.

LISTE: specifica

- Una possibile (e minima) interfaccia è la seguente:

```
interface IList: IEnumerable
{
    bool Add(object x);

    bool Remove(object x);

    bool Contains(object x);

    bool IsEmpty();

    void Clear();
}
```

- IEnumerable: Per scandire gli elementi in sequenza attraverso un iteratore.
- Add: Inserisce l'oggetto `x` in fondo alla lista.
- Remove: Cancella la prima occorrenza dell'oggetto `x`.
- Contains: Verifica se la lista contiene l'oggetto `x`.
- IsEmpty: Verifica se la lista è logicamente vuota.
- Clear: Svuota la lista.

LISTE

Sia `L` un oggetto di tipo `IList`: ecco alcune operazioni senza iteratori.

Invocazione metodo	Stato della lista	risultato
`L.IsEmpty();`	[]	true
`L.Add('a');`	[a]	true
`L.Add('b');`	[a b]	true
`L.Add('c');`	[a b c]	true
`L.Contains('b');`	[a b c]	true
`L.Remove('b');`	[a c]	true
`L.Contains('b');`	[a c]	false
`L.IsEmpty();`	[a c]	false

LISTE

Alcune operazioni con iteratori: la barra | denota un cursore.

Invocazione metodo	Stato della lista	risultato
`L.Add(2);`	[2]	true
`L.Add(5);`	[2 5]	true
`L.Add(4);`	[2 5 4]	true
`IEnumerator i=L.GetEnumerator();`	[|2 5 4]	i è un iteratore
`i.MoveNext()`	[2 |5 4]	true
`i.MoveNext()`	[2 5 |4]	true
`i.Current()`	[2 5| 4]	5
`i.Reset()`	[|2 5 4]	

LISTE: esempio

- Una porzione di codice che legge un file (`nomi.txt`) e memorizza il contenuto in una lista. Viene *allocata* la *giusta* quantità di *elementi*. Stampa il contenuto della lista attraverso l'uso di un iteratore. Rimuove un elemento della lista (`Mario`) e stampa il nuovo contenuto della lista attraverso l'uso di un `foreach`.

nomi.txt

```
Luca
Mario
Elisa
Andrea
```

Una possibile uscita

```
Luca Mario Elisa Andrea
----------------------------
Luca Elisa Andrea
```

LISTE: esempio

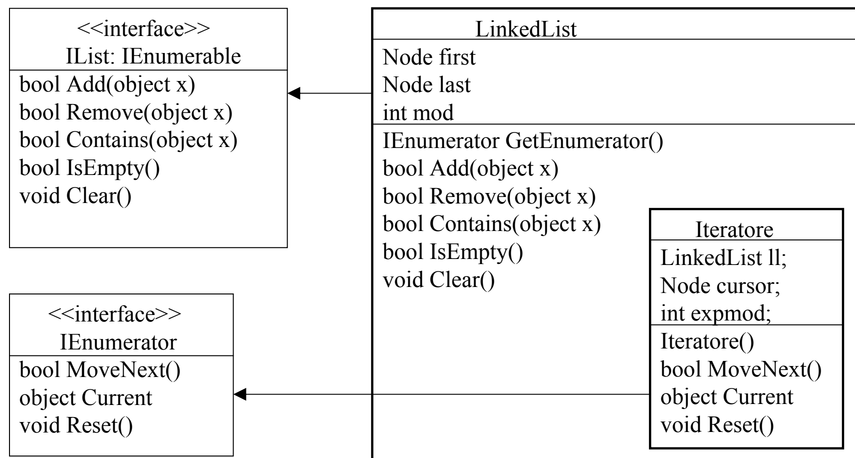
```
StreamReader fin = new StreamReader("nomi.txt");
string sInput1;
IList L = new LinkedList();

while ((sInput1 = fin.ReadLine()) != null)
    L.Add(sInput1);
fin.Close();

IEnumerator iter = L.GetEnumerator();
while (iter.MoveNext())
    Console.Write(iter.Current + " ");

Console.Write("\n---------------------\n");
L.Remove("Mario");
foreach (string s in L)
    Console.Write(s + " ");
```


LISTE: implementazione

- Una possibile realizzazione della lista consiste nel considerare una sequenza di nodi che contengono un dato e un riferimento al nodo successivo nella sequenza. Tale sequenza è realizzata nella classe `LinkedList` che implementa l'interfaccia `IList`.
- Un oggetto di tipo `LinkedList` è manipolato utilizzando i metodi resi disponibili dall'interfaccia `IList`, non si accede direttamente alla sequenza di nodi. È un *tipo di dato astratto*, ADT.

LISTE: implementazione

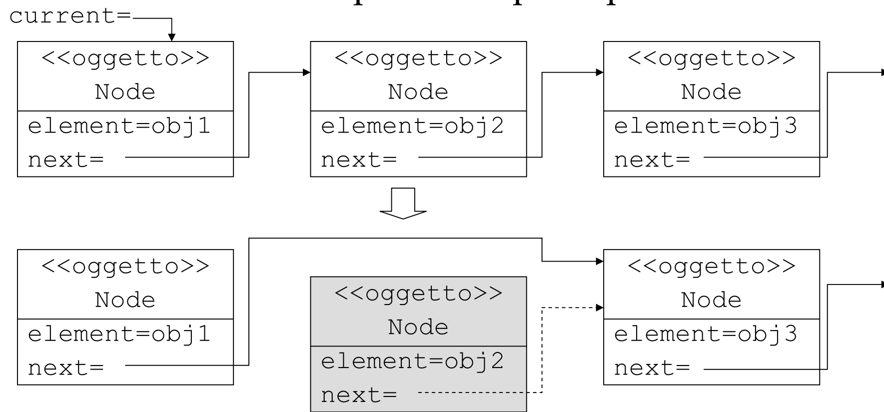
Si realizza l'interfaccia `IList` con la classe `LinkedList` e l'interfaccia `IEnumerator` con la classe interna `Iteratore`.



NODI: operazioni principali

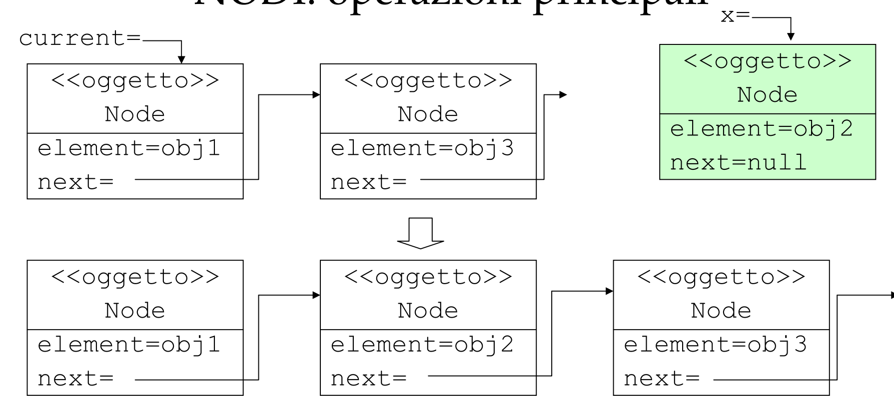
- Le due principali operazioni da realizzare sulle liste concatenate (sui *nodi* della lista nella fase di implementazione) sono: la *cancellazione* di un elemento della lista e l'*inserimento* di un elemento nella lista. Quindi diminuire o aumentare la lunghezza di una lista.
- La semplicità di tali operazioni è la ragione d'essere delle liste concatenate.

NODI: operazioni principali



Il nodo da <u>cancellare</u> viene fatto "*cadere*": `current` è un riferimento ad un oggetto `Node`, se prima della cancellazione `current.next` punta al nodo `obj2`, allora: `current.next=current.next.next;`

NODI: operazioni principali



Un nodo da <u>inserire</u>: `current` e `x` sono riferimenti ad oggetti `Node`, se prima dell'inserimento `current.next` punta al nodo `obj3`, allora:

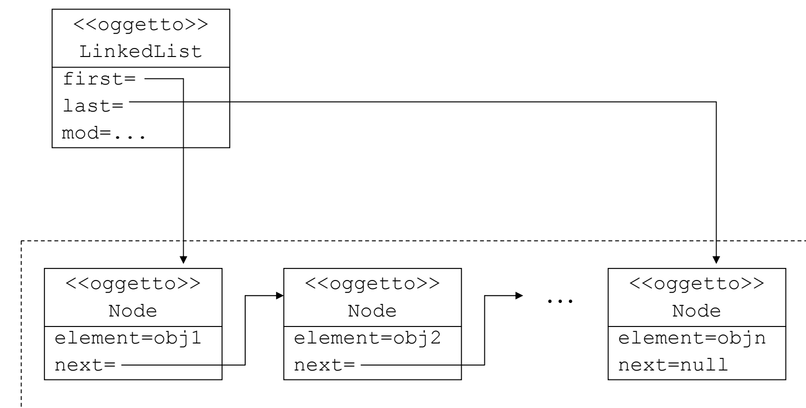
```
x.next=current.next;
current.next=x;
```


LISTE: implementazione, LinkedList

- Vediamo una possibile implementazione della lista.
- Si usa una variabile `first` che mantiene il riferimento al primo nodo della lista e una variabile `last` che mantiene un riferimento all'ultimo nodo.
- Inoltre è utilizzata una variabile intera `mod`, che tiene conto delle modifiche alla lista, per evitare che modifiche concorrenti possano interferire con il comportamento degli iteratori.

LINKEDLIST

- Una possibile rappresentazione grafica:



LINKEDLIST

```
class LinkedList : IList
{
    private Node first;
    private Node last;
    private int mod;

    public LinkedList()
    {
        first = last = null;
        mod = 0;
    }
```

Numero di modifiche: chiamate a `Remove()`, `Add()` e `Clear()`.

LINKEDLIST

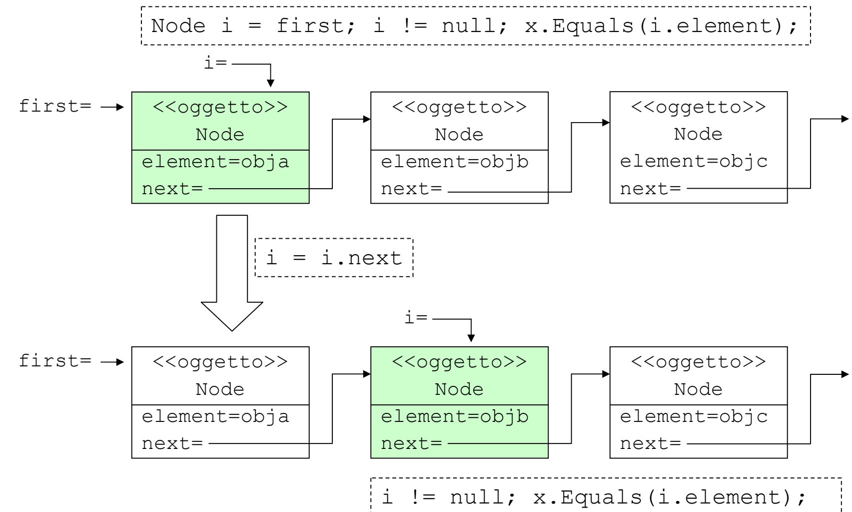
```
public bool IsEmpty()
{
    return first == null;
}

public void Clear()
{
    first = last = null;
    mod++;
}
```


LINKEDLIST: ricerca

```
public bool Contains(object x)
{
    if (x == null)
        throw new Exception("Argomento null.");
    for (Node i = first; i != null; i = i.next)
        if (x.Equals(i.element))
            return true;
    return false;
}
```

Si esaminano i nodi della lista con un ciclo `for`: si definisce una variabile `i` di tipo `Node` a cui si assegna un riferimento al primo nodo. Si esce dal ciclo quando si finiscono i nodi. L'incremento si ottiene con l'assegnamento `i=i.next`, che sposta il riferimento al nodo successivo.

LINKEDLIST: ricerca



LINKEDLIST: inserimento

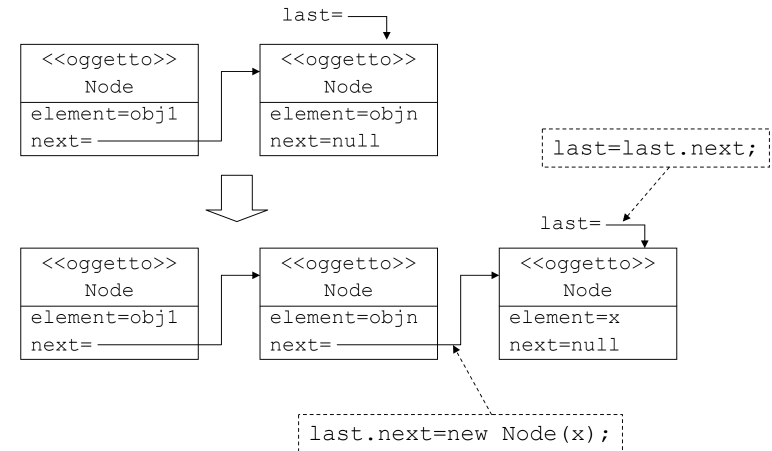
```
public bool Add(object x)
{
    if (x == null)
        throw new Exception("Argomento null.");
    if (IsEmpty())
    {
        first = last = new Node(x);
    }
    else
    {
        last.next = new Node(x);
        last = last.next;
    }
    mod++;
    return true;
}
```

Lista vuota

Lista non vuota

LINKEDLIST: inserimento

- Lista non vuota:



LINKEDLIST: prestazioni

- Aggiungere un nodo alla lista ha un costo computazionale costante *O(1)*: è indipendente dal numero di nodi nella lista. Non si devono copiare gli elementi in una nuova struttura, come nel caso di array pieni.
- Verificare se la lista contiene un dato oggetto implica l'uso di un ciclo `for`. Si consideri il caso medio: se ogni nodo della lista ha la stessa probabilità di contenere il dato, allora si ha una iterazione per verificare il primo nodo, due iterazioni per verificare il secondo nodo, e in una sequenza lunga `n` si ha in media

$$\frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = O(n)$$

LINKEDLIST: rimozione

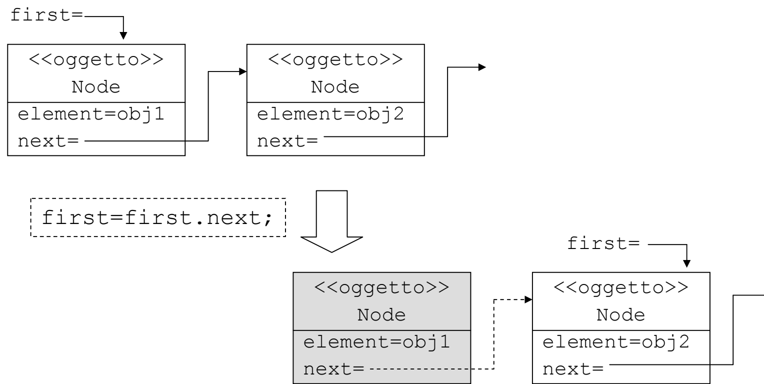
```
public bool Remove(object x)
{
    if (x == null) throw new Exception("Argomento null.");
    if (IsEmpty()) return false;
    if (x.Equals(first.element))
    {
        if (first == last)
            last = null;
        first = first.next;
        mod++;
        return true;
    }
    for (Node i = first; i.next != null; i = i.next)
    {
        if (x.Equals(i.next.element))
        {
            if (i.next == last)
                last = i;
            i.next = i.next.next;
            mod++;
            return true;
        }
    }
    return false;}
```

Primo elemento

e ultimo

LINKEDLIST: rimozione

- Primo elemento:



LINKEDLIST: prestazioni

- Considerazioni analoghe a quelle sviluppate per calcolare la complessità computazionale media del metodo per la ricerca di un dato valgono anche per il metodo di rimozione di un dato.

- Quindi il metodo `Remove()` ha in media un costo lineare, cioè $O(n)$.

LINKEDLIST: iteratore

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

```
public IEnumerator GetEnumerator()
{
    return new Iteratore(this);
}
```

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

LINKEDLIST: implementazione iteratore

- Vediamo una possibile implementazione dell'iteratore secondo l'interfaccia `IEnumerator`.

- Si usa una variabile `cursor` che individua il nodo il cui contenuto è restituito dalla proprietà `Current`. La variabile `ll` di tipo `LinkedList` fa riferimento all'oggetto che ha creato l'iteratore: attraverso tale riferimento si accede ai dati contenuti nell'oggetto lista.
- 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.

LINKEDLIST: implementazione iteratore

```
class LinkedList : IList{
   ...
  class Iteratore : IEnumerator{
     LinkedList ll;
     Node cursor;
     int expmod;

     public Iteratore(LinkedList o){
        ll = o;
        cursor = null;
        expmod = ll.mod;
     }
   ...
}
```

Classe interna

LINKEDLIST: implementazione iteratore

```
public object Current
{
    get {
        if (ll.mod != expmod)
            throw new Exception("Collezione modificata.");
        if (cursor == null)
            throw new Exception("Stato non valido.");
        return cursor.element;
    }
}

public void Reset()
{
    if (ll.mod != expmod)
        throw new Exception("Collezione modificata.");
    cursor = null;
}
```

Modifica concorrente

LINKEDLIST: implementazione iteratore

```
public bool MoveNext()
{
    if (ll.mod != expmod)
        throw new Exception("Collezione modificata.");
    if (ll.IsEmpty())
        return false;
    if (cursor == null)
    {
        cursor = ll.first;
        return true;
    }
    if (cursor != ll.last)
    {
        cursor = cursor.next;
        return true;
    }
    else
        return false;
}
```


LISTE: considerazioni

- 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).

LISTE: 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.

LISTE: 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.

LISTE: 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.

LISTE: ordinamento

- 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.

LISTE: 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.

LISTE: 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 (((IComparable)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++;}
```


LISTE: ordinamento

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

```
LinkedList L = new LinkedList();
Random rnd = new Random();
for (int i = 0; i < 15; i++)
    L.Add(rnd.Next(21) - 10);
foreach (int n in L)
    Console.Write(n + " ");
Console.Write("\n");
L.Sort();
foreach (int n in L)
    Console.Write(n + " ");
```

`Boxing` del tipo `int`.

Una possibile uscita

```
9 3 9 8 -5 4 10 1 10 8 -4 0 6 -2 9
-5 -4 -2 0 1 3 4 6 8 8 9 9 9 10 10
```


LISTE: ordinamento

- In molti casi non si ha bisogno di implementare esplicitamente un algoritmo di ordinamento. Si può *mantenere ordinata la lista* mentre si inseriscono 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.

LISTE: 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.

- In generale, potrebbe essere necessario implementare un diverso tipo di iteratore che permetta operazioni sugli elementi della lista, quali rimozioni e inserimenti.

LISTE: doppiamente concatenate

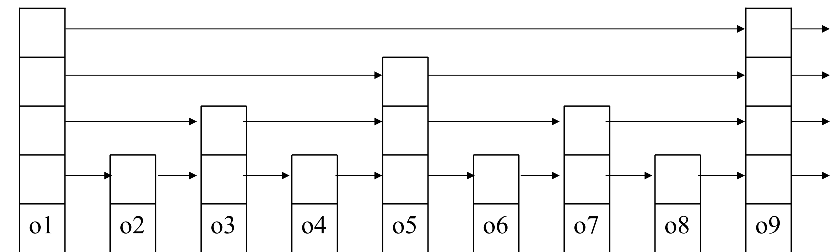
- Esistono altri tipi di lista, vediamone alcuni.

- 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.

LISTE: 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.

LISTE: con salti



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