# lezione 5

Liste

## Tipo di dato Lista

Una lista permette di memorizzare dati in maniera sequenziale e permette di inserire, accedere e cancellare dati in posizioni arbitrarie (non come nel caso di pile e code).

A differenza di pile e code non c'è un insieme di operazioni ben definite come nelle pile e code. Le operazioni base sono:
-insert(L,k): inserisce un elemento k nella lista L
-remove(L,k): rimuove l'elemento k dalla lista L
-contains(L,k): ritorna true se la lista L contiene l'elemento k

# Lista concatenata

Consideriamo la realizzazione del tipo di dato Lista con strutture collegate, ovvero mediante liste concatenate.

Naturalmente è possibile realizzare il tipo di dato Lista con strutture dati diverse, ad esempio con array…

NOTA: Le Liste in Java sono realizzate dalla classe LinkedList di java.util (vedi API di Java) che fornisce ben più delle operazioni base che abbiamo elencato…

# Liste concatenate & more…

Una lista concatenata può avere forme diverse:
-lista semplice (singly linked list)
-lista doppia (doubly linked list)

Può inoltre:
-avere un puntatore all'ultimo elemento
-essere circolare (circular list)
-avere un nodo sentinella (sentinel) per meglio gestire i casi limite. Si tratta di un record "dummy" cioè senza informazioni (il campo key è nil)

I dati memorizzati possono:
-essere ordinati o meno
-ammettere duplicazioni o meno

# Lista semplice (singly linked)

Una lista semplice ha nodi contenenti:
-un attributo (campo) key che memorizza l'informazione
-un attributo next che collega il nodo al successivo nodo della lista
Dato un nodo x, usiamo la notazione key[x] e next[x] per denotare i suoi attributi.
La lista semplice viene acceduta tramite un attributo head che punta alla testa della lista.

La lista è vuota quando head=nil


# Lista semplice: inserimento

Vediamo un algoritmo per l'inserimento di un nuovo elemento in testa alla lista:

**insert(L,k)**
1. sia x un nuovo nodo
2. key[x] ← k
3. next[x] ← head
4. head ← x

L'operazione viene eseguita in tempo costante.

# Lista semplice: inserimento in coda

Vediamo ora come avviene l'inserimento di un nuovo elemento in coda alla lista

**insert(L,k)**

```
1. sia x un nuovo nodo
2. key[x] ← k
3. next[x] ← nil   // il nuovo nodo deve essere l'ultimo della lista
4. if head = NIL
5.    then head ← x
6.    else index ← head
7.           while next[index] ≠ NIL
8.                do index ← next[index]
9.           next[index] ← x
```

L'operazione viene eseguita in tempo lineare rispetto al numero di elementi in lista.

# Lista semplice: ricerca

La ricerca di un elemento k nella lista L procede scandendo la lista, ovviamente partendo dalla testa

**contains(L,k)**

```
1. index ← head
2. trovato ← false
3. while index ≠ nil and not trovato
4.    do INV
5.         if key[index] = k
6.            then trovato ← true
7.            else index ← next[index]
8. return trovato
```

INV: gli elementi da head ad index non compreso sono diversi da k

Nel caso pessimo l'operazione viene eseguita in tempo lineare rispetto al numero di elementi in lista.

# Lista semplice: ricerca

INV: gli elementi da head ad index non compreso sono diversi da k
Inizializzazione: all'inizio index=head e quindi è vero che gli elementi da head a head non compreso sono diversi da k (lista vuota!)
Mantenimento: sia INV vero per index fissato. Allora:
- se key[index] = k l'algoritmo pone trovato a true e non modifica index.
-altrimenti l'algoritmo sposta index all'elemento successivo.
In entrambi i casi dopo l'esecuzione del corpo del ciclo rimane vero che gli elementi da head fino ad index non compreso sono diversi da k, cioè l'invariante viene mantenuto per il ciclo successivo.

# Lista semplice: ricerca

Terminazione: il ciclo termina per due ragioni:
-index = nil e trovato=false: in tal caso l'invariante assicura che k non è presente in tutta la lista (da head a nil)
-index≠nil e trovato=true: in tal caso l'invariante assicura che k non è presente prima di index e trovato= true che k=key[index]

In entrambi i casi l'algoritmo ritorna correttamente il valore della variabile trovato.

# Lista semplice: cancellazione

La cancellazione di un elemento k dalla lista L procede cercando la prima occorrenza dell'elemento k nella lista e cancellando il nodo corrispondente.

**remove(L,k)**

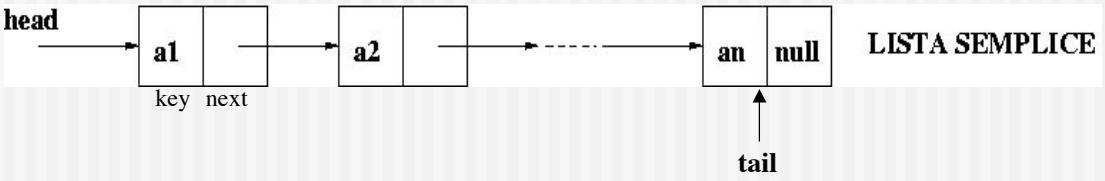
```
1.  index ← head
2.  prec ← nil
3.  trovato ← false
4.  while index ≠ nil and not trovato
5.    do INV
6.       if key[index] = k
7.       then trovato ← true
8.       else prec ← index
9.            index ← next[index]
10.  if trovato    // in tal caso index è diverso da nil
11.  then if index = head    // cancellazione in testa
12.         then head ← next[head]
13.         else next[prec] ← next[index]
```

precondizione: lista non vuota!

Nel caso pessimo l'operazione viene eseguita in tempo lineare rispetto al numero di elementi in lista.

# Lista semplice con tail

Consideriamo una lista semplice e, oltre all'attributo head, aggiungiamo un attributo tail che punta all'ultimo elemento in lista.




Come cambiano le operazioni di inserimento, ricerca e cancellazione? Migliora la loro efficienza? E quanto costa la gestione dell'attributo tail?

# Lista semplice con tail: inserimento

Vediamo un algoritmo per l'inserimento di un nuovo elemento in testa alla lista:

**insert(L,k)**

1. sia x un nuovo nodo
2. key[x] ← k
3. next[x] ← head
4. if head = NIL  // aggiornamento attributo tail
5.  then tail ← x
6. head ← x

L'operazione viene eseguita in tempo costante, come per le liste semplici senza tail.

# Lista semplice con tail: inserimento in coda

Vediamo ora come avviene l'inserimento di un nuovo elemento in coda alla lista

**insert(L,k)**

1. sia x un nuovo nodo
2. key[x] ← k
3. next[x] ← nil  // il nuovo nodo deve essere l'ultimo della lista
4. if head = NIL
5.  then head ← x
6.  else next[tail] ← x
7. tail ← x  // aggiornamento attributo tail

L'operazione viene eseguita in tempo costante. Quindi c'è un miglioramento rispetto alle liste semplici senza tail.

# Lista semplice con tail: ricerca

L'aggiunta dell'attributo tail non modifica l'algoritmo di ricerca di un elemento nella lista L.
Più in generale possiamo dire che la gestione dell'attributo tail è a carico solamente delle operazioni che modificano lo stato della lista.

# Lista semplice con tail: cancellazione

L'attributo tail deve essere aggiornato in caso di rimozione dell'ultimo elemento in lista. Se la lista diventa vuota bisogna farlo puntare a NIL.

**remove(L,k)**

```
1.  index ← head
2.  prec ← nil
3.  trovato ← false
4.  while index ≠ nil and not trovato
5.     do if key[index] = k
6.          then trovato ← true
7.          else prec ← index
8.                 index ← next[index]
9.     if trovato    // in tal caso index è diverso da nil
10.    then if index = head   // cancellazione in testa
11.           then head ← next[head]
12.                  if head = NIL     // aggiornamento attributo tail
13.                     then tail ← NIL
14.           else next[prec] ← next[index]
15.                  if index = tail    // aggiornamento attributo tail
16.                     then tail ← prec
```

precondizione: lista non vuota!

L'operazione viene eseguita in tempo lineare rispetto al numero di elementi in lista. L'aggiornamento di tail ha costo costante.

# Lista semplice con sentinella

Consideriamo una lista semplice in cui il primo nodo è una *sentinella*, ovvero un nodo con chiave NIL che serve semplicemente per uniformare il trattamento dei casi limite (lista vuota, fine lista). La lista quindi diventa:


La lista vuota non viene più rappresentata da NIL ma dal solo nodo sentinella.

Si osservi inoltre che il puntatore head non viene mai modificato dalle operazioni di inserimento e cancellazione.

# Lista semplice con sentinella

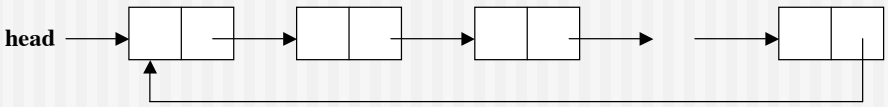
Ad esempio, per aggiungere in coda un elemento è sufficiente il ciclo:

```
index = head;
while next[index] ≠ NIL
   index = next[index]
next[index] = nuovo nodo con chiave k e next a NIL
```

Confrontate il codice con l'algoritmo di inserimento in coda per liste semplici!
Ovviamente anche il codice per le altre operazioni va modificato per tener conto della sentinella.

# Lista semplice circolare

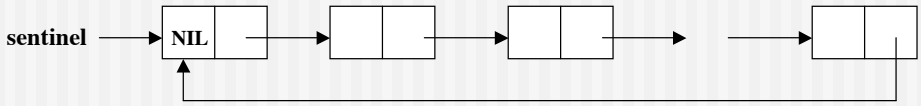
Consideriamo una lista semplice in cui l'ultimo elemento punta al primo (invece che a NIL). La lista diventa quindi circolare.



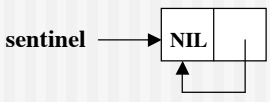

Come cambiano le operazioni di inserimento, ricerca e cancellazione? Migliora la loro efficienza?
Provate a rispondere a queste domande…

# Lista semplice circolare con sentinella

Consideriamo una lista semplice circolare, in cui il primo elemento sia un record con informazione a NIL detto sentinella. L'aggiunta della sentinella permette di uniformare il trattamento dei casi limite (lista vuota, fine lista).




L'accesso alla lista avviene attraverso il puntatore alla sentinella (sentinel). La lista è vuota quando c'è solo la sentinella.

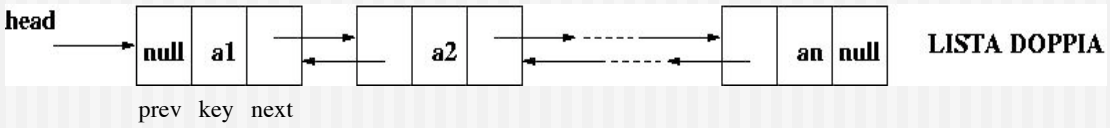



Come cambiano le operazioni di inserimento, ricerca e cancellazione? Migliora la loro efficienza?
Provate a rispondere a queste domande…

# Lista doppia (doubly linked)

Una lista doppia ha nodi contenenti:
-un attributo (campo) key che memorizza l'informazione
-un attributo next che collega il nodo al successivo in lista
-un attributo prev che collega il nodo al precedente in lista
Dato un nodo x, usiamo la notazione key[x], next[x] e prev[x] per denotare i suoi attributi.
La lista doppia viene acceduta tramite un attributo head che punta alla testa della lista.

La lista è vuota quando head=nil




# Lista doppia: inserimento

L'inserimento di un nuovo elemento avviene in testa alla lista:

**insert(L,k)**

1. sia x un nuovo nodo
2. key[x] ← k
3. next[x] ← head
4. prev[x] ← nil
5. if head ≠ nil
6. then prev[head] ← x
7. head ← x

L'operazione viene eseguita in tempo costante.

# Lista doppia: ricerca

La ricerca di un elemento k nella lista L procede scandendo la lista, ovviamente partendo dalla testa

**contains(L,k)**

```
1. index ← head;
2. trovato ← false
3. while index ≠ nil and not trovato
4.    do INV
5.         if key[index] = k
6.            then trovato ← true
7.            else index ← next[index]
8. return trovato
```

Non cambia nulla rispetto alla lista semplice!

# Lista doppia: cancellazione

La cancellazione di un elemento k dalla lista L procede cercando la prima occorrenza dell'elemento k nella lista e cancellando il nodo corrispondente.

**remove(L,k)**

```
1. index ← head;
2. trovato ← false;
3. while index ≠ nil and not trovato
4.    do INV
5.         if key[index] = k
6.         then trovato ← true
7.         else index ← next[index]
8. if trovato
9.    if prev[index] ≠ nil
10.   then next[prev[index]] ← next[index]
11.   else head ← next[index]
12.   if next[index] ≠ nil
13.   then prev[next[index]] ← prev[index]
```

precondizione: lista non vuota!

Nel caso pessimo l'operazione viene eseguita in tempo lineare rispetto al numero di elementi in lista.

# Liste doppie: varianti

Anche per le liste doppie si possono considerare tutte le varianti viste per le liste singole:
-lista doppia con attributo tail
-lista doppia circolare
-lista doppia circolare con sentinella

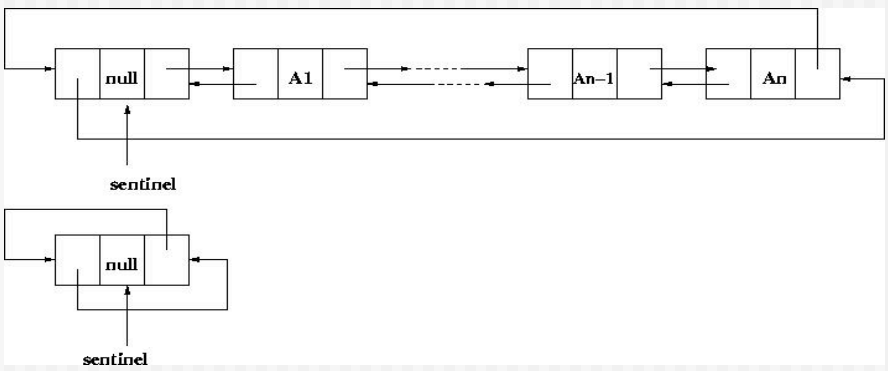
E di nuovo possiamo chiederci cosa cambia per le operazioni di base.

Vediamo come viene realizzata in Java la lista doppia circolare e con sentinella.

# Lista doppia circolare e con sentinella

Una lista doppia circolare ha nodi uguali alla lista doppia, ma organizzati circolarmente (vedi figura)
La sentinella è un nodo aggiuntivo che non contiene informazioni (key[sentinel] = nil).

La lista è vuota quando contiene solo il nodo sentinella (vedi figura)



# Interfacce

- Un'*interfaccia* è una collezione di definizioni di metodi (senza implementazione) e di costanti.
- Le interfacce servono a dichiarare metodi implementati da classi diverse... e quindi consentono di astrarre.
- Consentono di rivelare l'interfaccia di un oggetto senza rivelare la sua implementazione.
- Servono a catturare similarità tra classi senza forzare una relazione di sottoclasse.

- Dichiarando un'interfaccia, si dichiara un nuovo tipo riferimento, che può essere usato dovunque si usano i nomi dei tipi: dichiarazione di variabili, parametri di un metodo, etc.
- Per usare un'interfaccia bisogna definire una classe che la implementi, ovvero che implementi tutti i metodi in essa dichiarati

# Lista.java: un'interfaccia per le liste (1)

```
package Liste;
public interface Lista {

    // post: ritorna il numero di elementi della lista
    public int size();

    // post: ritorna true sse la lista non ha elementi
    public boolean isEmpty();

     // post: svuota la lista
    public void clear();

    // pre:  ob non nullo
    // post: aggiunge l'oggetto ob in testa alla lista
    public void insert(Object ob);

    // pre: l'oggetto passato non e' nullo
    // post: ritorna true sse nella lista c'e' un elemento
    //       uguale a value
    public boolean contains(Object value);
```

# Lista.java: un'interfaccia per le liste (2)

```
    // pre: l'oggetto passato non e' nullo
    // post: rimuove l'elemento uguale a value
    //       ritorna true se l'operazione e' riuscita, false altrimenti
    public boolean remove(Object value);

    // post: ritorna una stringa che rappresenta tutti gli elementi
    //       della lista, in sequenza
    public String toString();
}
```

# Package

Un package è un insieme di classi correlate, tutte definite in una sottodirectory comune, avente lo stesso nome del package.
Ogni file in un package deve iniziare (a parte eventuali spazi o commenti) con: package <package_name>;
È consentita una sola dichiarazione di package in un codice sorgente.

```
// classe BinaryTree del package BinTrees
package BinTrees;
public class BinaryTree{
. . .
}
```

I nomi dei package sono ordinati gerarchicamente e separati da punti, es: `BinTrees.BST`

```
home/simeoni/javapgms/
                      /BinTrees/
                               /BST
```

# Package

Le classi che implementano il tipo di dato lista e la relativa interfaccia sono logicamente correlate. Definiamo per loro il package Liste.
Quindi creiamo una directory Liste e ci mettiamo tutti i file che riguardano le liste:

```
home/simeoni/javapgms/Liste/ls
Lista.java
…
…
```

Ogni file del package inizia con il comando:

```
package Liste;
```

# Il package Liste

Vogliamo realizzare un package Liste in cui le liste sono implementate mediante liste doppie, circolari e con sentinella.

Il package Liste è così composto:
-interfaccia Lista.java
-classe RecordLD.java che rappresenta un nodo della lista doppia
-classe ListaDoppia.java che implementa l'interfaccia Lista.java attraverso una lista doppia, circolare e con sentinella

# Il comando import

Il comando import permette di includere classi o interi package nel file corrente:

```
import  <package_name>.<class_name>;
import  <package_name>.*;
```

Deve precedere tutte le dichiarazioni delle classi

```
import Liste.*;
public class usaListe {
   ...
}
```

I programmi che usano le liste (client) stanno fuori dal package e, per poterlo usare, lo importano. Devono quindi avere visibilità del package

Import permette quindi di inserire nel dominio dei nomi altri package rispetto a quello corrente (che fa sempre parte del dominio di nomi).

# Classe RecordLD.java

```
package Liste;
class RecordLD {
    Object key;             // valore memorizzato nell'elemento
    RecordLD next;          // riferimento all'elemento successivo
    RecordLD prev;          // riferimento all'elemento precedente

    // post: costruisce un nuovo elemento con valore v,
     //       elemento successivo nextel e precedente prevel
    RecordLD(Object ob, RecordLD nextel, RecordLD prevel) {
        key = ob;
        next= nextel;
        if (next != null)
           next.prev = this;
        prev = prevel;
        if (prev != null)
           prev.next = this;
    }

    // post: costruisce un nuovo elemento con valore v, e niente
    //       next e prev
    RecordLD(Object ob) { this(ob,null, null);}
}
```

La classe e tutti i suoi costituenti hanno livello di visibilità *package access*, cioè sono visibili solo all'interno del package Liste

# Classe ListaDoppia.java

```
package Liste;
public class ListaDoppia implements Lista {
    private RecordLD sentinel; // riferimento alla sentinella
    private int count;         // num. elementi nella lista

    // post: crea una lista vuota
    public ListaDoppia() {
        // la sentinella ha chiave puntatori nulli
        sentinel = new RecordLD(null,null, null);
        sentinel.next = sentinel;
        sentinel.prev = sentinel;
        count = 0;
    }

    // post: ritorna il numero di elementi della lista
    public int size() { return count; }

    // post: ritorna true sse la lista non ha elementi
    public boolean isEmpty() { return count == 0; }
```

# Classe ListaDoppia.java

```
    // post: svuota la lista
    public void clear() {
        sentinel.next = sentinel;
        sentinel.prev = sentinel;
        count = 0;
    }

    // pre:  ob non nullo
    // post: aggiunge l'oggetto ob in testa alla lista
    public void insert(Object ob) {
        RecordLD nuovo = new RecordLD(ob, sentinel.next, sentinel);
        count++;
    }
```

# Classe ListaDoppia.java

```
// pre: l'oggetto passato non e' nullo
// post: ritorna true sse nella lista c'e' un elemento
//       uguale a value
public boolean contains(Object value) {
   return contains(value, sentinel.next);
}

// pre: l'oggetto passato non e' nullo
// post: ritorna true sse nella lista da n in poi c'e' un elemento
//       uguale a value
private boolean contains(Object value, RecordLD n) {
   if (n == sentinel)
      return false;
   else
      if (n.key.equals(value))
         return true;
      else
         return contains(value, n.next);
}
```

# Classe ListaDoppia.java

```
// pre: l'oggetto passato non e' nullo
// post: rimuove l'elemento uguale a value
//       ritorna true se l'operazione e' riuscita, false altrimenti
public boolean remove(Object value) {
    RecordLD index = sentinel.next;
    while (index != sentinel && !index.key.equals(value))
       index = index.next;

    if (index != sentinel) {  // elemento trovato!
       index.prev.next = index.next;
       index.next.prev = index.prev;
       count--;
       return true;  // l'operazione e' riuscita
    }
    else // l'elemento non e' stato trovato...ritorno false
       return false;
}
```

# Classe ListaDoppia.java

```
    // post: ritorna una stringa che rappresenta tutti gli elementi
    //       della lista, in sequenza
    public String toString() {
       String s = "";
       RecordLD index = sentinel.next;
       while (index != sentinel) {
          s = s + index.key.toString() + ", ";
          index = index.next;
       }
       return s;
    }
} /* fine classe ListaDoppia */
```