Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

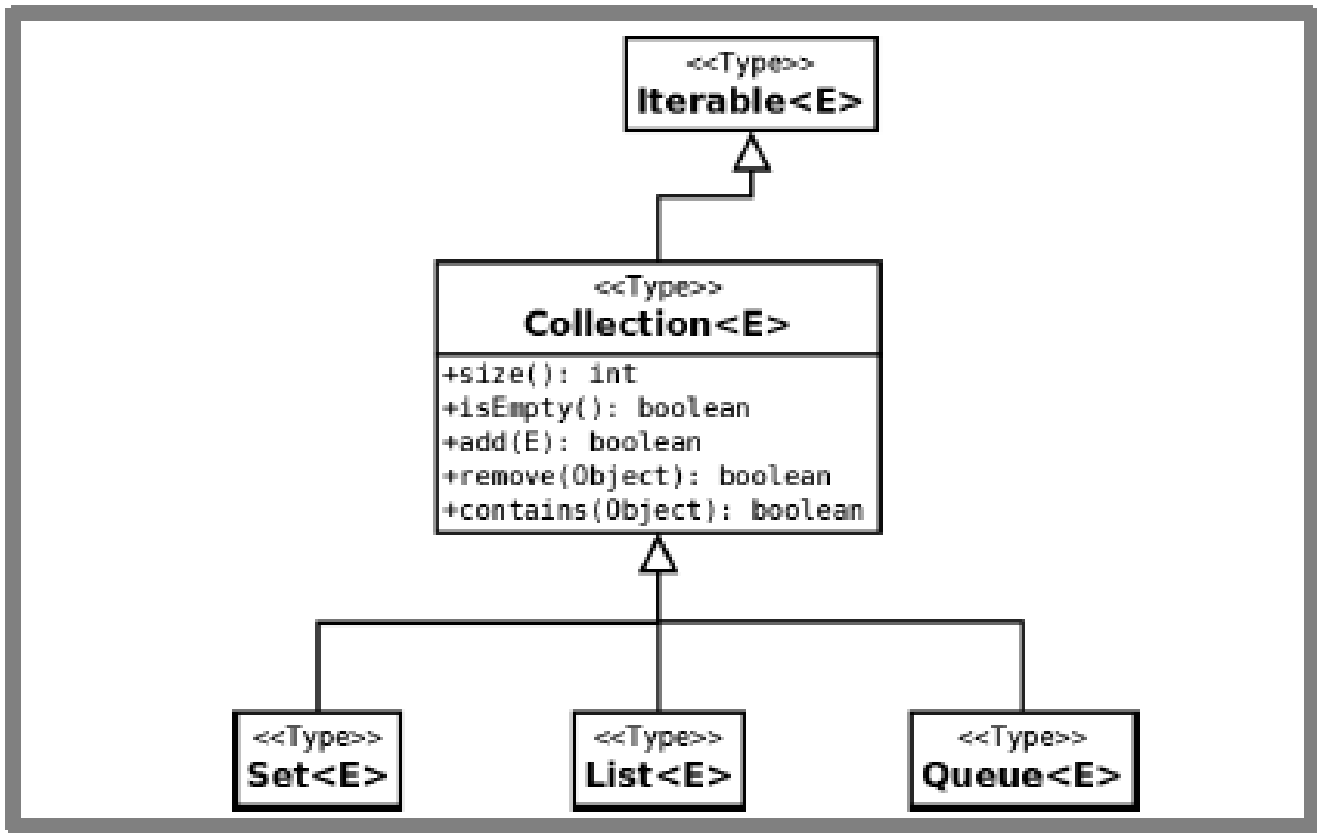
Università degli Studi dell'Aquila

Corso di Algoritmi e Strutture Dati con Laboratorio

Java Collections Framework (II parte)

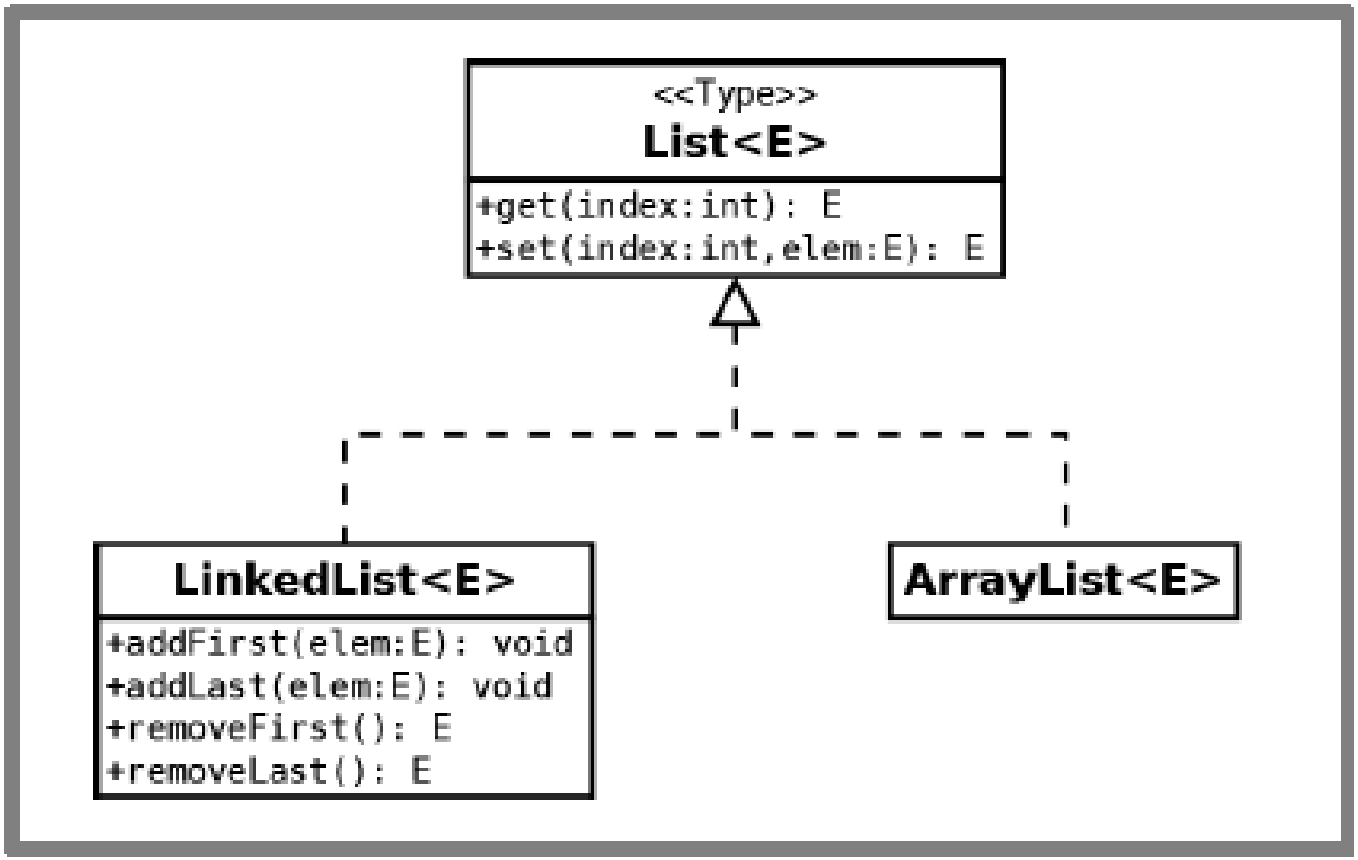
L'interfaccia Collection

- Dove eravamo rimasti…



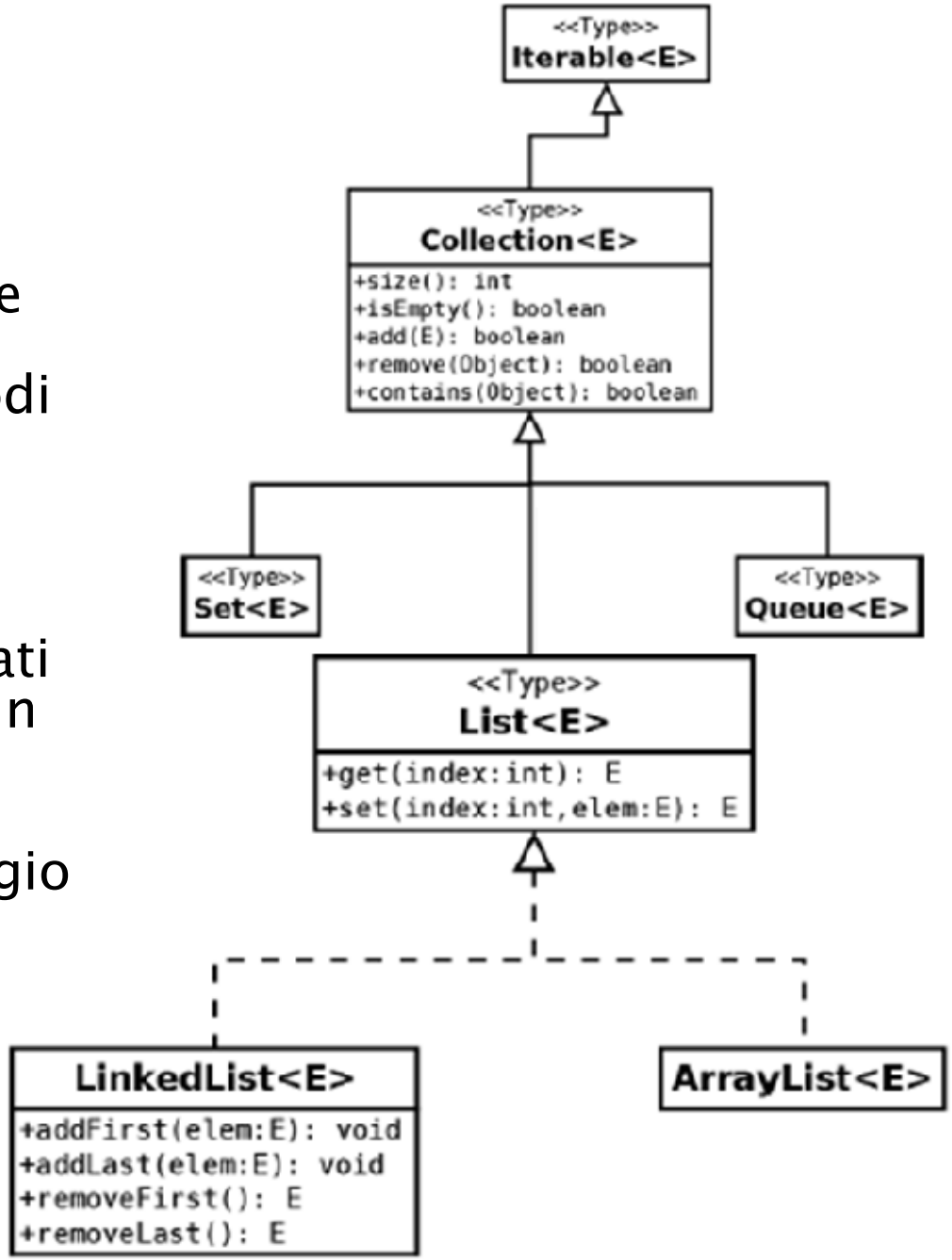
L'interfaccia List

- L'interfaccia List e le classi che la implementano:



L'interfaccia List

- L'interfaccia `List` estende l'interfaccia `Collection` aggiungendo alcuni metodi relativi all'uso di indici
- In ogni esemplare di una classe che implementa l'interfaccia `List` gli elementi sono memorizzati in sequenza, in base ad un indice
- Vista come entità indipendente dal linguaggio di programmazione, una lista è un **tipo di dato astratto**



L'interfaccia `List`: realizzazioni

- Come sappiamo per uno stesso tipo di dato sono possibili diverse realizzazioni alternative basate su strutture dati diverse
- In generale, la scelta di una particolare struttura dati consente un'implementazione delle operazioni richieste più o meno efficiente
- L'efficienza dipende anche dal modo in cui i dati sono organizzati all'interno della struttura.
- Un modo naturale per implementare una struttura dati che realizza un certo tipo di dato è scrivere una classe che ne implementa la corrispondente interfaccia

Tecniche per rappresentare collezioni di oggetti

- Tecniche fondamentali usate per rappresentare collezioni di elementi:
  - Tecnica basata su **strutture indicizzate** (array)
  - Tecnica basata su **strutture collegate** (record e puntatori)
  - La scelta di una tecnica piuttosto che di un'altra può avere un impatto cruciale sulle operazioni fondamentali (ricerca, inserimento, cancellazione, ...)

Strutture indicizzate: array

- Proprietà di base degli array:

1. (Forte) Gli indici delle celle di un array sono numeri interi consecutivi
   - Il tempo di accesso ad una qualsiasi cella è costante ed indipendente dalla dimensione dell'array
2. (Debole) Non è possibile aggiungere nuove celle ad un array
   - Il ridimensionamento è possibile solo mediante la riallocazione dell'array, ossia la creazione di un nuovo array e la copia del contenuto dal vecchio al nuovo array

Ridimensionamento di array

La tecnica del raddoppiamento-dimezzamento

- L'idea è quella di non effettuare riallocazioni ad ogni inserimento/cancellazione, ma solo ogni $\Omega(n)$ operazioni
- Se h è la dimensione dell'array e le prime $n>0$ celle dell'array contengono gli elementi della collezione, la tecnica consiste nel mantenere una dimensione h che soddisfa, per ogni $n>0$, la seguente invariante:

$$n \leq h < 4n$$

La tecnica del raddoppiamento-dimezzamento

- L'invariante $n \leq h < 4n$ sulla dimensione dell'array viene mantenuta mediante riallocazioni così effettuate:
  - Inizialmente, per $n=0$, si pone $h=1$
  - Quando $n>h$, l'array viene riallocato raddoppiandone la dimensione ($h \leftarrow 2h$)
  - Quando n scende a $h/4$ l'array viene riallocato dimezzandone la dimensione ($h \leftarrow h/2$)

La tecnica del raddoppiamento-dimezzamento

- Nota teorica: Se v è un array di dimensione $h \geq n$ contenente una collezione non ordinata di n elementi, usando la tecnica del raddoppiamento-dimezzamento ogni operazione di inserimento o cancellazione di un elemento richiede tempo ammortizzato costante
  - Previo eventuale raddoppiamento dell'array, l'inserimento si effettua in posizione n, e poi si incrementa n di 1
  - Per la cancellazione dell'elemento in posizione i, lo si sovrascrive con l'elemento in posizione n-1, decrementando n di 1 ed eventualmente dimezzando l'array

Strutture dati collegate: record e puntatori

- In Java un record può essere rappresentato in modo naturale mediante un oggetto
- I numeri associati ai record sono i loro indirizzi in memoria
- I record sono creati e distrutti individualmente ed in maniera dinamica, per cui gli indirizzi non sono necessariamente consecutivi
- Un record viene creato esplicitamente dal programma tramite l'istruzione `new`, mentre la sua distruzione avviene in modo automatico quando non è più in uso (*garbage collection*)
- Per mantenere i record di una collezione in relazione tra loro ognuno di essi deve contenere almeno un indirizzo di un altro record della collezione

Strutture dati collegate: record e puntatori

Proprietà:

- (Forte) è possibile aggiungere o eliminare record ad una struttura collegata
- (Debole) Gli indirizzi dei record di una struttura collegata non sono necessariamente consecutivi

Le classi `ArrayList<E>` e `LinkedList<E>`

- La classe `ArrayList` realizza l'interfaccia `List` mediante un array
- La classe `LinkedList` realizza l'interfaccia `List` mediante liste (doppiamente) collegate
- Esempio: la classe `RandomList` crea e manipola un oggetto List contenente numeri interi casuali (RandomList.java)

L'interfaccia `List`

RandomList.java:

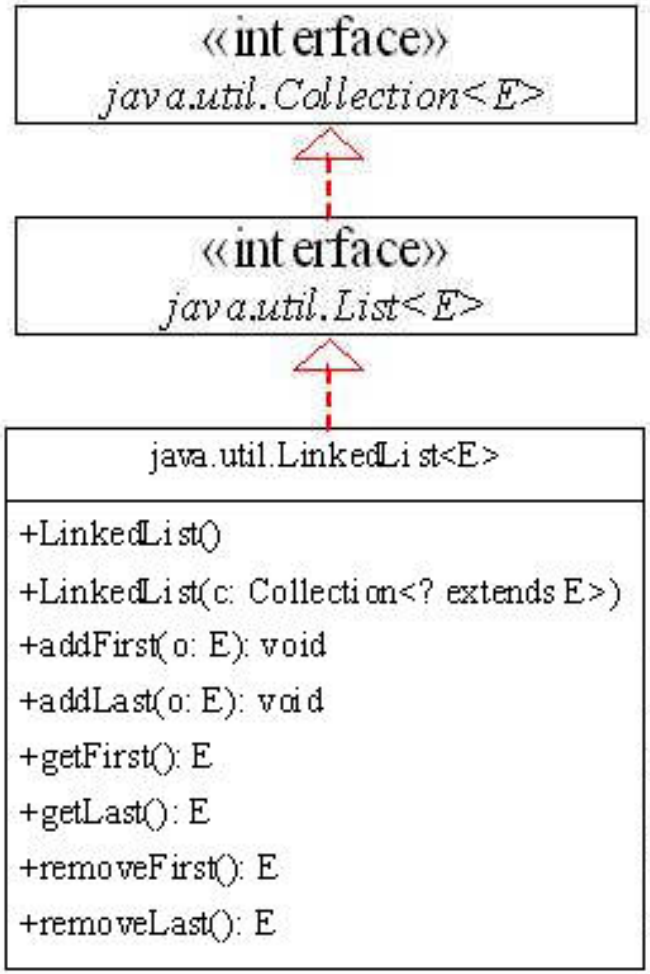
- Avremmo potuto usare un enunciato for-each avanzato per scandire `randList`? No, perché oltre a ispezionare la lista, eliminiamo alcuni elementi contenuti in essa.
- La variabile `randList` è stata dichiarata come riferimento polimorfico e inizializzata con un riferimento ad un oggetto di tipo `ArrayList`.
- Per eseguire nuovamente il programma usando un oggetto di tipo **`LinkedList`** l'unica modifica necessaria è l'invocazione del costruttore:

```
List<Integer> randList=new LinkedList<Integer>();
```

La classe `ArrayList`

- `ArrayList` è un'implementazione di List, realizzata internamente con un **array dinamico**
- La rallocazione dell'array avviene in modo trasparente per l'utente
- Il metodo `size` restituisce il numero di elementi effettivamente presenti nella lista, non la dimensione dell'array sottostante
- Il ridimensionamento avviene in modo che l'operazione di inserimento (`add`) abbia *complessità ammortizzata costante* (per ulteriori informazioni sulla complessità ammortizzata, si consulti un testo di algoritmi e strutture dati)

La classe LinkedList



Creates a default empty linked list.

Creates a linked list from an existing collection.

Adds the object to the head of this list.

Adds the object to the tail of this list.

Returns the first element from this list.

Returns the last element from this list.

Returns and removes the first element from this list.

Returns and removes the last element from this list.

La classe LinkedList

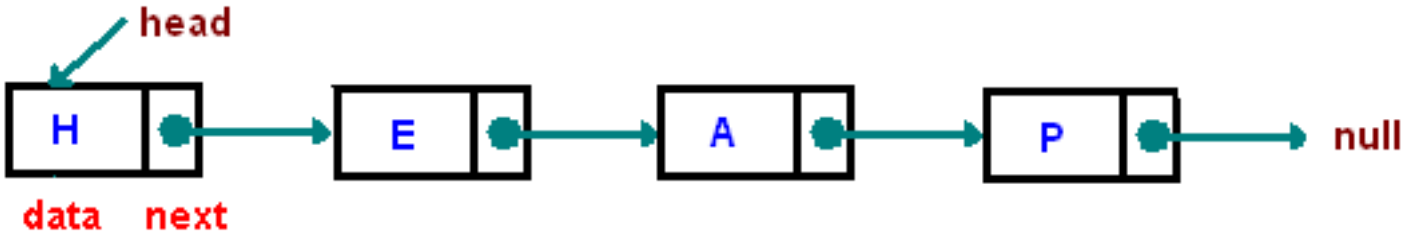
- I metodi permettono di utilizzare una LinkedList sia come **stack sia come coda**
- Per ottenere il comportamento di uno **stack (detto LIFO: last in first out), inseriremo ed estrarremo** gli elementi dalla **stessa estremità della lista**
  - ad esempio, inserendo con con addLast (o con add) ed estraendo con removeLast
- Per ottenere, invece, il comportamento di una **coda (FIFO: first in first out), inseriremo ed**
- estrarremo gli elementi da due **estremità opposte**

Le liste e l'accesso posizionale

- L'accesso posizionale (metodi `get` e `set`) si comporta in maniera molto diversa in `LinkedList` rispetto ad `ArrayList`
- In `LinkedList`, ciascuna operazione di accesso posizionale può richiedere un tempo proporzionale alla lunghezza della lista (complessità *lineare)*
- In `ArrayList`, ogni operazione di accesso posizionale richiede tempo *costante*
- Pertanto, **è fortemente sconsigliato utilizzare l'accesso posizionale su LinkedList**
- Se l'applicazione richiede l'accesso posizionale, è opportuno utilizzare un semplice array, oppure la classe ArrayList

Linked Lists

- Una linked list è una struttura dati dinamica lineare in cui ogni elemento (detto **nodo**) è un oggetto separato



- Ogni nodo è composto da due items: il dato ed un riferimento al nodo successivo
- L'ultimo nodo ha un riferimento a `null`.
- Il (riferimento al) primo nodo è detto "**`head`**" della lista. Se la lista è vuota `head` è un riferimento `null`.

Inner classes

- In Java è possibile definire una classe (B) dentro un'altra classe (A)
- La classe A è detta "outer class", mentre la B è detta "nested class"
- Ci sono due tipi di classi nested:
  - static
  - non-static (inner class)
- NB: Le classi (non-static) "Inner" sono sottoinsiemi delle classi "nested" (anche dette static inner)

Inner classes

- Una inner class B è un membro di A ed ha accesso agli altri membri (anche privati) di A.

  Viceversa la classe esterna A può accedere a tutti i membri della classe (not-static) inner.
- Una nested class (static inner) non può fare riferimento direttamente alle variabili o metodi d'istanza definiti nella sua classe esterna: essa può usarli solo attraverso un object reference.
- Una nested class può accedere solo ai membri statici di A.

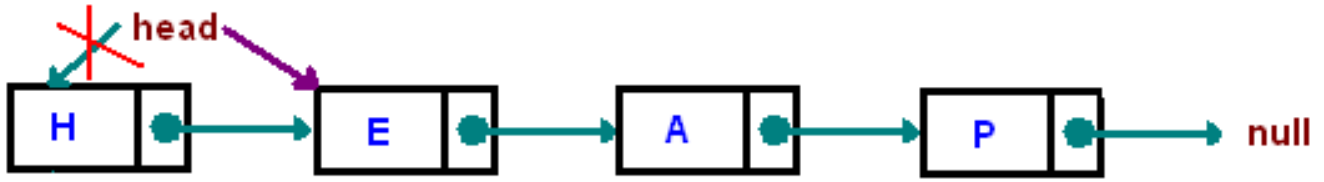
Linked Lists: la classe Node

- In seguito definiamo una classe `(my)LinkedList` con due nested classes: `static Node class` and non-static `LinkedListIterator class`

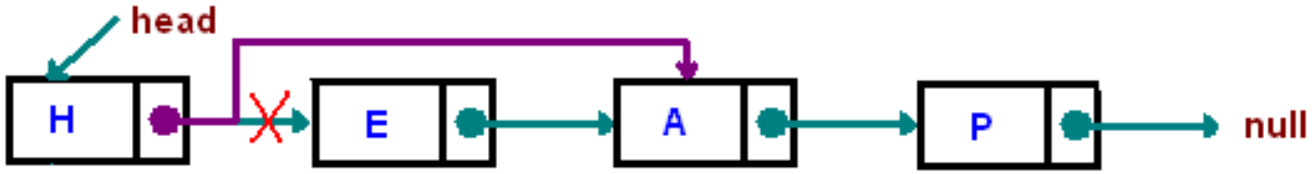
```
private static class Node<AnyType> {
	private AnyType data;
	private Node<AnyType> next;

	public Node(AnyType data, Node<AnyType> next){
		this.data = data;
		this.next = next;
	}
}
```

Esempi

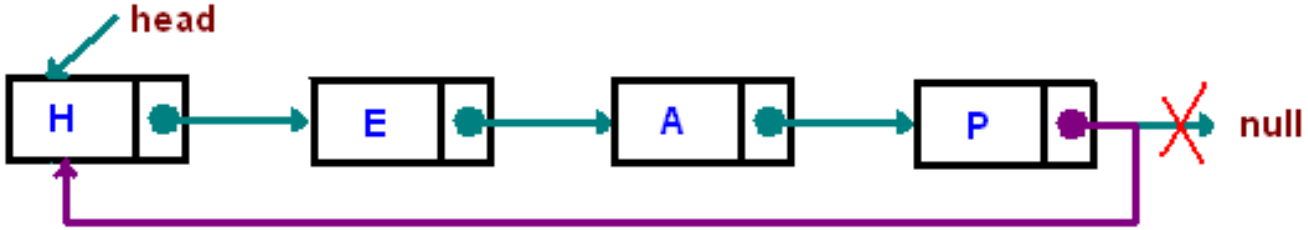
```
head = head.next;
```




```
head.next = head.next.next;
```



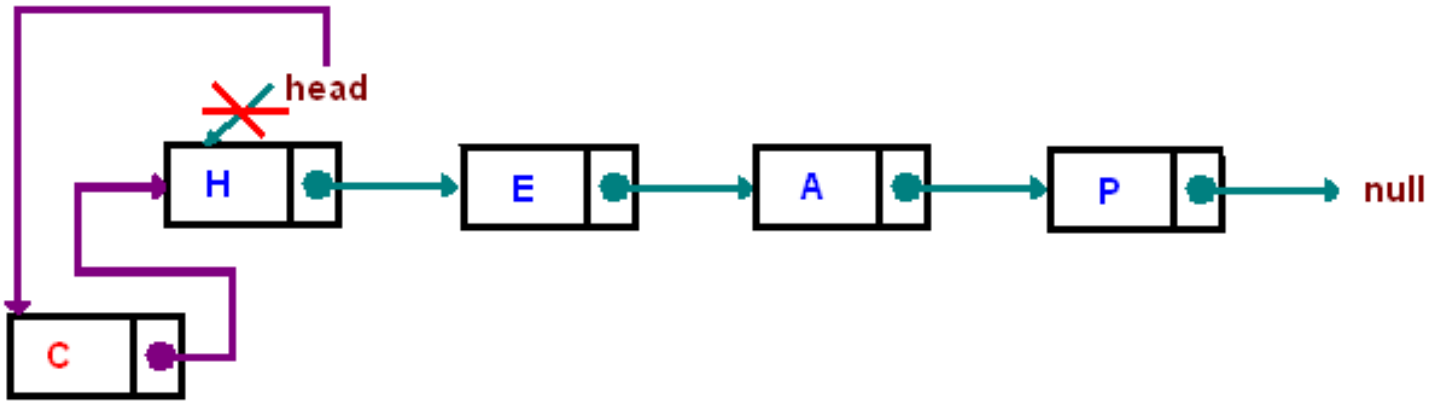

```
head.next.next.next.next = head;
```



Linked List Operations

- addFirst crea un nodo e lo aggiunge all'inizio della lista:

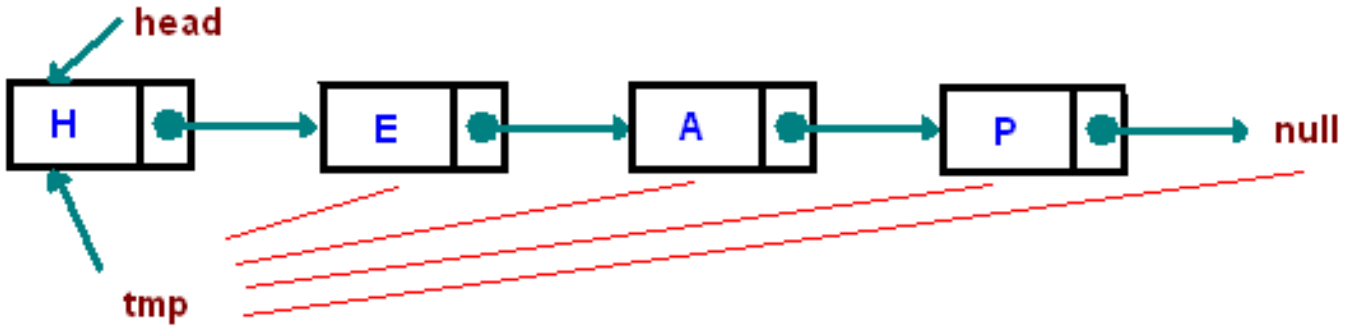
```
public void addFirst(AnyType item) {
  head = new Node<AnyType>(item, head);
}
```



Linked List Operations

- **Traversing** Iniziando da `head` (senza cambiare il riferimento) si accede ad ogni nodo fino a quando non si raggiunge `null`.

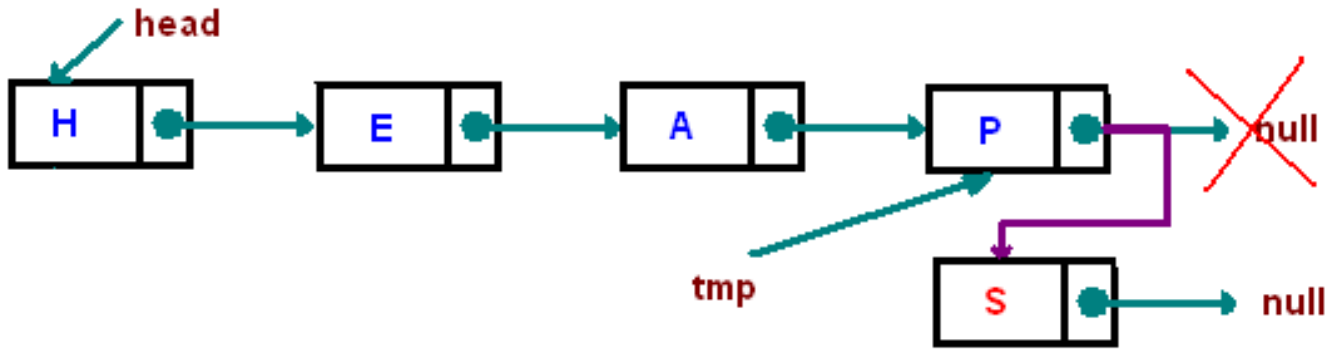
```
Node tmp = head;
while(tmp != null) tmp = tmp.next;
```



Linked List Operations

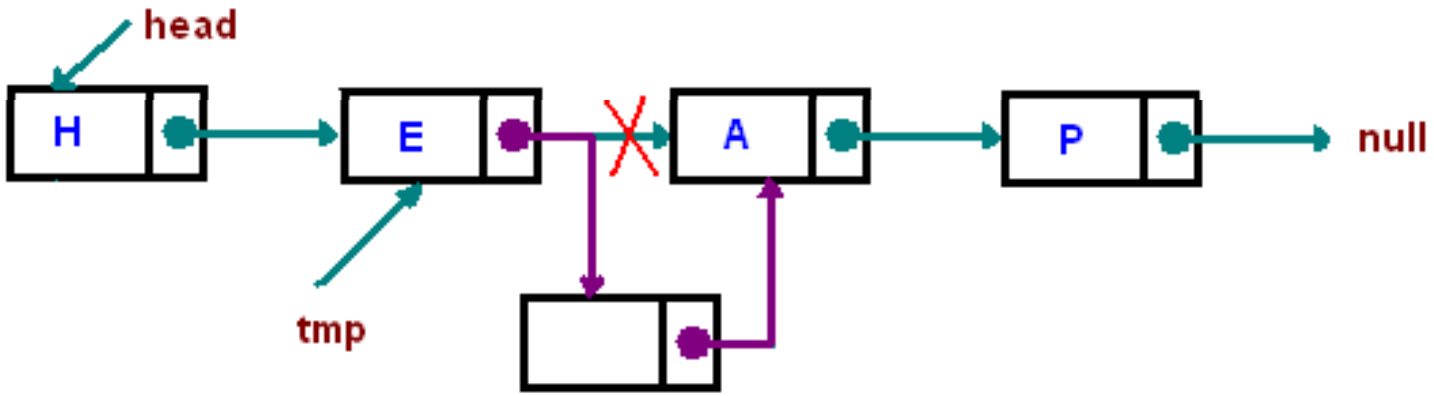
- **addLast** appende il nodo alla fine della lista.

```
public void addLast(AnyType item) {
   if(head == null) addFirst(item);
   else {
      Node<AnyType> tmp = head;
      while(tmp.next != null) tmp = tmp.next;
      tmp.next = new Node<AnyType>(item, null);
   }
}
```



Linked List Operations

- Inserting "after“ (E) : Cerca un nodo contenente “key” ed inserisce un nuovo nodo dopo di esso.

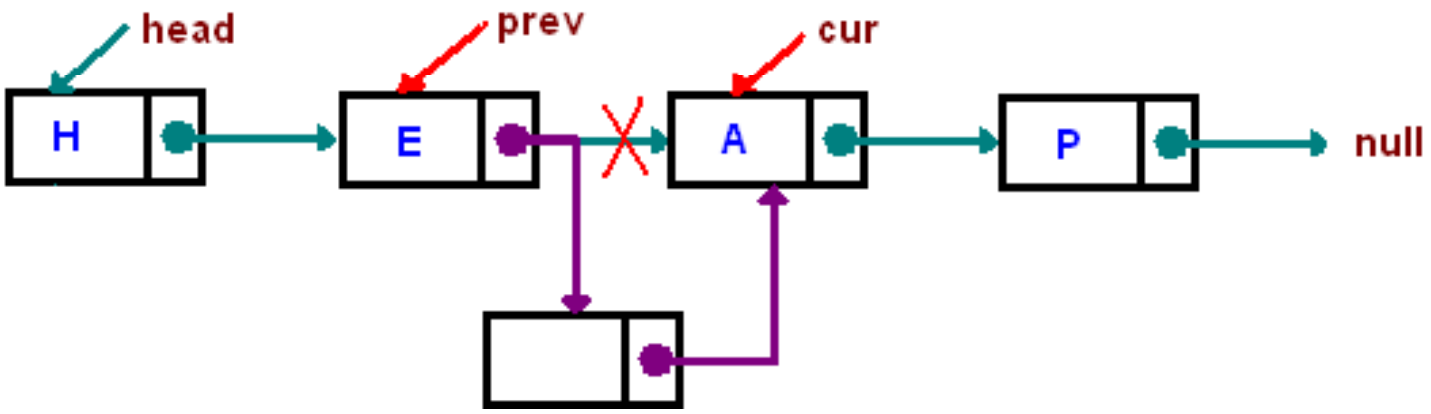


Linked List Operations

```
public void insertAfter(AnyType key, AnyType toInsert){
Node<AnyType> tmp = head;
while(tmp!=null && !tmp.data.equals(key))
      tmp = tmp.next;
if(tmp!=null) tmp.next =
      new Node<AnyType>(toInsert,tmp.next);
}
```

Linked List Operations

- Inserting "before“ (A) cerca un nodo contenente “key” e inserisce un nuovo nodo prima di esso



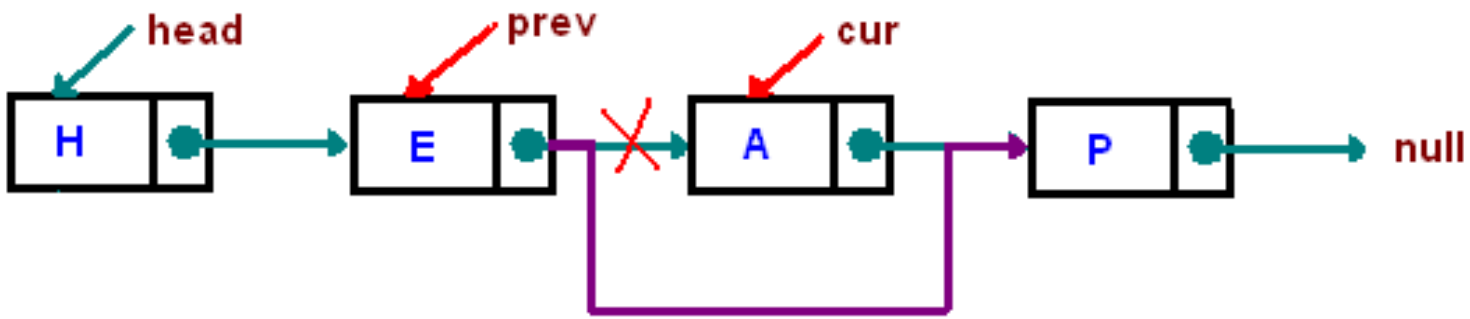
Linked List Operations

```
public void insertBefore(AnyType key,
                         AnyType toInsert) {
  if(head == null) return null;
  if(head.data.equals(key)) {
     addFirst(toInsert);
     return; }
  Node<AnyType> prev = null;
  Node<AnyType> cur = head;
  while(cur != null && !cur.data.equals(key)) {
     prev = cur; cur = cur.next;
  }
  if(cur != null) prev.next =
       new Node<AnyType>(toInsert, cur);
}
```

Linked List Operations

- Deletion (A) Cerca un nodo contenente “key” e lo cancella. Ci sono tre casi particolari da gestire:

1. La lista è vuota
2. Bisogna cancellare il primo nodo
3. Il nodo non è in lista



Linked List Operations

```
public void remove(AnyType key) {
    if(head == null)
       throw new RuntimeException("cannot delete");
    if( head.data.equals(key) ) {
       head = head.next;
       return; }
    Node<AnyType> cur = head;
    Node<AnyType> prev = null;
    while(cur != null && !cur.data.equals(key) ) {
       prev = cur; cur = cur.next; }
    if(cur == null)
       throw new RuntimeException("cannot delete");
       prev.next = cur.next;
}
```

Linked List Operations

- Iterator

```
public Iterator<AnyType> iterator() {
   return new LinkedListIterator();
}
```

- **`LinkedListIterator`** è una classe privata interna alla classe `LinkedList`