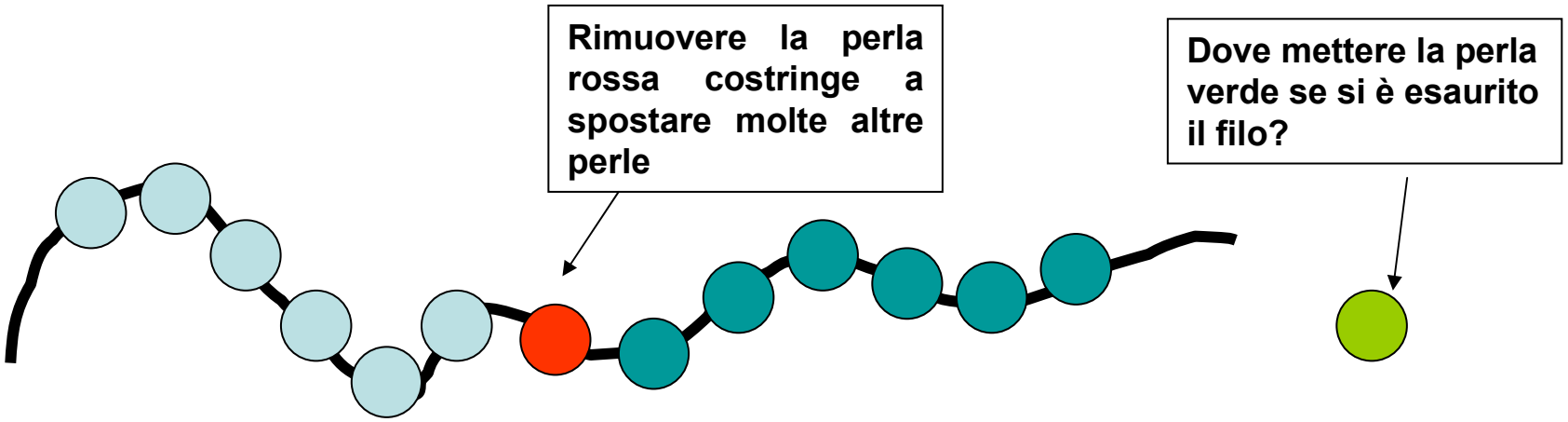
# Liste Linkate

# Le liste con array…

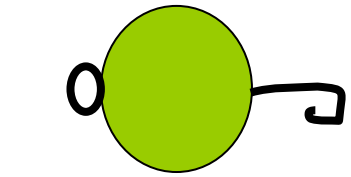
- Volendo usare una metafora, l'implementazione delle liste tramite array è come infilare le perle (gli oggetti della lista) in un filo di lunghezza predefinita.

- E' chiaro che inserire e togliere perle dal centro della collana crea problema, così come la lunghezza del filo crea problemi, anche se la soluzione "array" appare molto semplice e facilmente comprensibile.



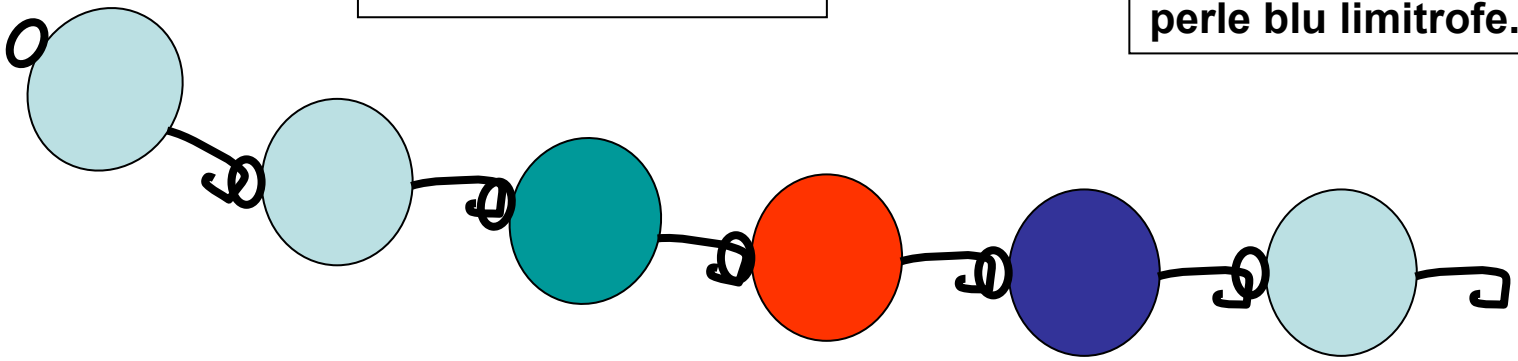
# Idea alternativa…

Ogni perla avrà un “uncino” per connettersi alla perla successiva: connettendo così le perle una dopo l’altra si realizzerà la collana anche senza avere un unico filo a sorreggerla.



**Sarà sempre possibile attaccare nuove perle in coda a all’inizio della collana**

**Per rimuovere (o inserire) la perla rossa è sufficiente staccare e riconnettere le due perle blu limitrofe.**



# Come tradurre l'idea nella programmazione?

- La *perla* è chiaramente una "**cella**" di informazione della nostra lista. Essa potrebbe essere facilmente modellata da un oggetto "**nodo**".
- L'**uncino** in termini di gestione della memoria è semplicemente ***una variabile interna all'oggetto nodo che riporta l'indirizzo RAM dove si trova conservato il successivo nodo***.

# Come tradurre l'idea nella programmazione?

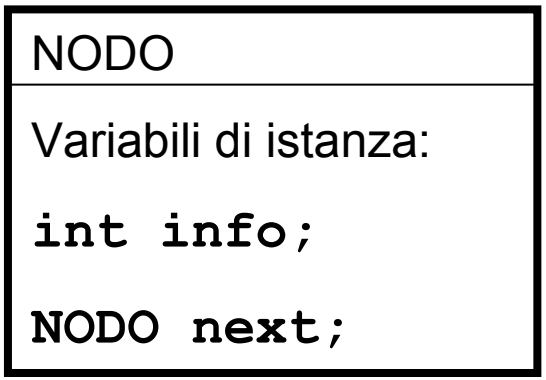
- I linguaggi che hanno l'uso esplicito dei puntatori (variabili fatte per conservare e manipolare indirizzi nella RAM) come il C realizzano questa idea in maniera assai semplice.

- In JAVA.
  - La cosa è ancora più semplice perché JAVA pur non lasciandoci intervenire sugli indirizzi in pratica li gestisce per noi in maniera molto trasparente e comoda.

- Esiste anche la possibilità di "implementare" questa idea senza usare alcun indirizzo di memoria o puntatore (sia implicito che esplicito) ma questa soluzione "vecchio stampo" la lasciamo per ultima…

- Il tipo di dati Lista realizza l'organizzazione dei dati con modalità in accesso sequenziale.
- I dati non risiedono in locazioni di memoria contigue, quindi ciascun dato deve includere oltre all'informazione vera e propria anche un riferimento ad un dato successivo.

# Schema dell'oggetto NODO




La variabile **info**, di tipo int (in questo caso), è destinata a "conservare" l'informazione vera e propria del nodo.

In particolare essa mantiene l'indirizzo RAM dove è conservato l'oggetto che si vuole mettere in lista.

La variabile **next** di tipo NODO mantiene l'indirizzo del nodo successivo nella lista.

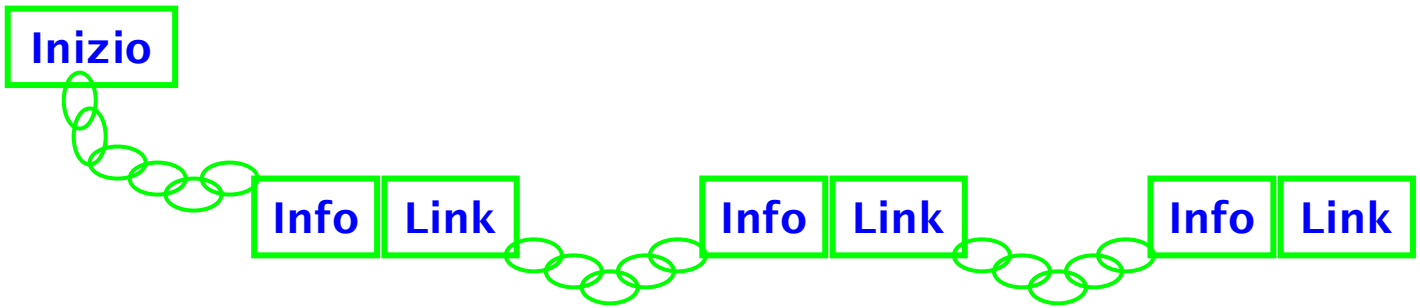
La comparsa di un riferimento ad una variabile di tipo NODO dentro la definizione dell'oggetto NODO non deve spaventare: qui registriamo solo un indirizzo RAM, quello presso il quale si trova il nodo successivo.

# Struttura di una lista linkata

Una lista linkata semplice è caratterizzata dal fatto che gli elementi vengono aggiunti dinamicamente solo quando è necessario.

Inoltre ogni elemento contiene un riferimento all'elemento successivo.

Serve anche un identificatore esterno per tener traccia della lista.



# Implementazione

In Java una lista linkata semplice può essere implementata con le seguenti classi:

```
public class Lista{
        private Nodo head;

        public Lista(){
                head = null;
        }
        …
}
```

# Implementazione

Per il generico nodo si ha:

```
public class Nodo{
	private int info;
	private Nodo next;

	public Nodo(int val){
		this(val, null);
	}
	public Nodo(int val, Nodo n){
		info = val;
		next = n;
	}
	…
```

# Implementazione

```
        public void setInfo(int val){
                info = val;
        }

        public int getInfo(){
                return info;
        }

        public void setNext(Nodo n){
                next = n;
        }

        public Nodo getNext(){
                return next;
        }
}
```

# Implementazione

Per creare un nuovo elemento scriviamo:

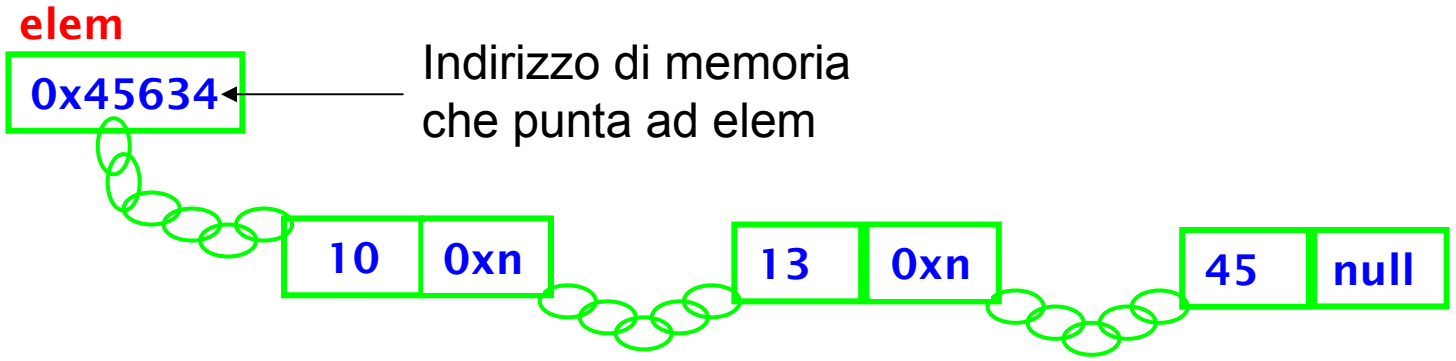
```
Nodo elem = new Nodo(10);
```

Per creare un secondo elemento:

```
elem.setNext(new Nodo(13) );
```

Per creare un terzo elemento:

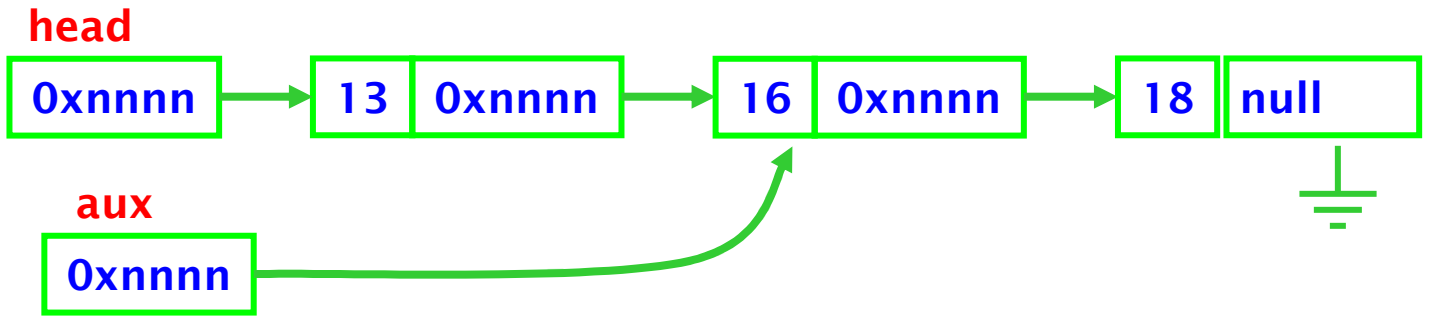
```
(elem.getNext()).setNext( new Nodo(45) );
```



# Implementazione

Per ogni nuovo nodo creato dobbiamo istanziare un nuovo oggetto della classe Nodo e dobbiamo scrivere il riferimento al nuovo nodo nel nodo precedente.

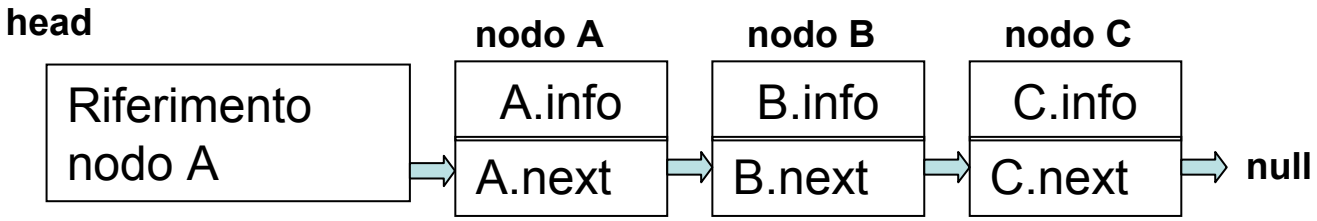
Per accedere ad un qualunque nodo dobbiamo accedere prima al nodo precedente!



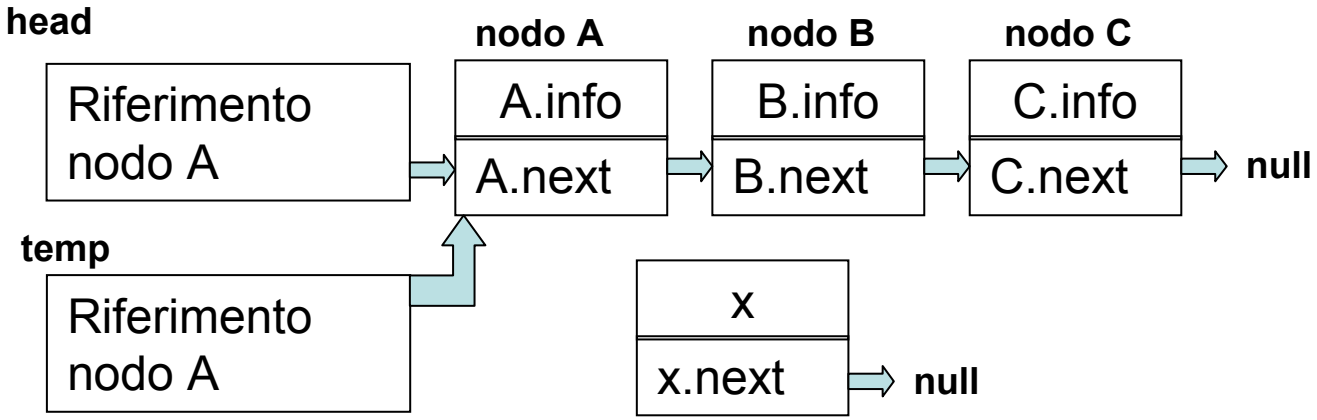

```
Nodo head = new Nodo(13);
Nodo aux = head;
aux.setNext( new Nodo(16) );
aux = aux.getNext();
aux.setNext( new Nodo(18) );
```

# Esempio: inserimento in testa

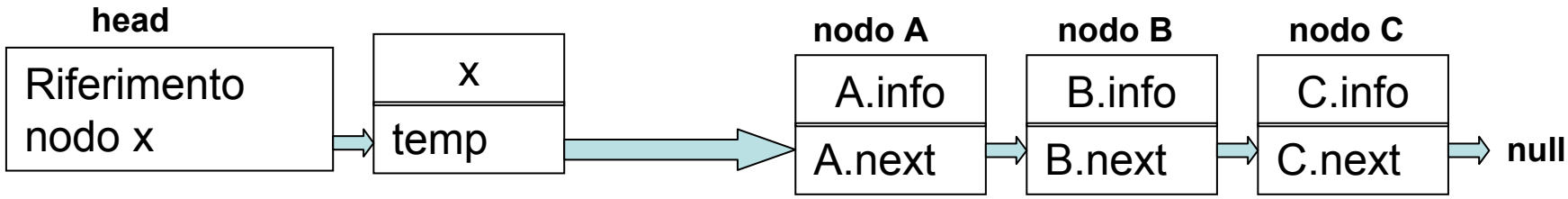
## Situazione di partenza




## Duplico il riferimento ad A e inserisco x in un nodo




## Aggiorno i puntatori in modo da ripristinare la situazione



# Inserimento di un nuovo elemento

Per inserire un nuovo elemento possiamo avere:

- inserimento in coda;
- inserimento in testa;
- inserimento ordinato.

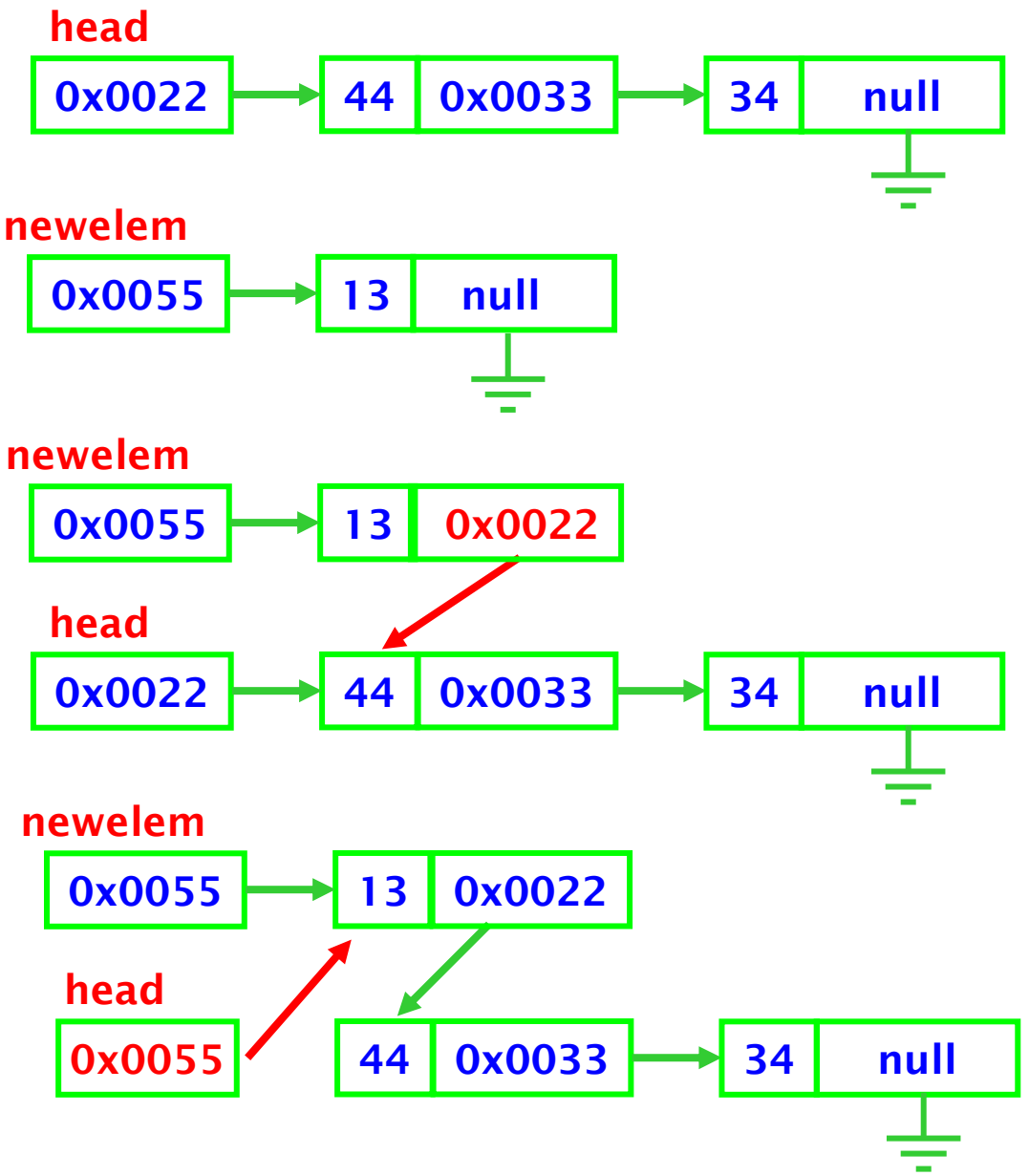
Il primo comporta lo scorrimento di tutti gli elementi già esistenti, mentre nel secondo caso dobbiamo solo modificare il valore della testa.

# Inserimento in testa

I passi da seguire per l'inserimento in testa sono:

- istanziare un nuovo elemento;
- collegare la lista già esistente al nuovo nodo;
- modificare il riferimento alla testa;

# Inserimento in testa



# Inserimento in testa

```
public class Lista{
	private Nodo head;

	public Lista(){
		head = null;
	}

	public void InsertHead(int val){
		Nodo aux = new Nodo(val,head);
		head = aux;
	}
}
```

# Inserimento in testa

La variabile aux può essere omessa.

```
public class Lista{
	private Nodo head;

	public Lista(){
		head = null;
	}

	public void InsertHead(int val){
		head = new Nodo(val, head);
	}
}
```

# Inserimento in testa

Notare che con l'inserimento in testa non è necessario verificare se il valore di head è valido o meno ( != null).

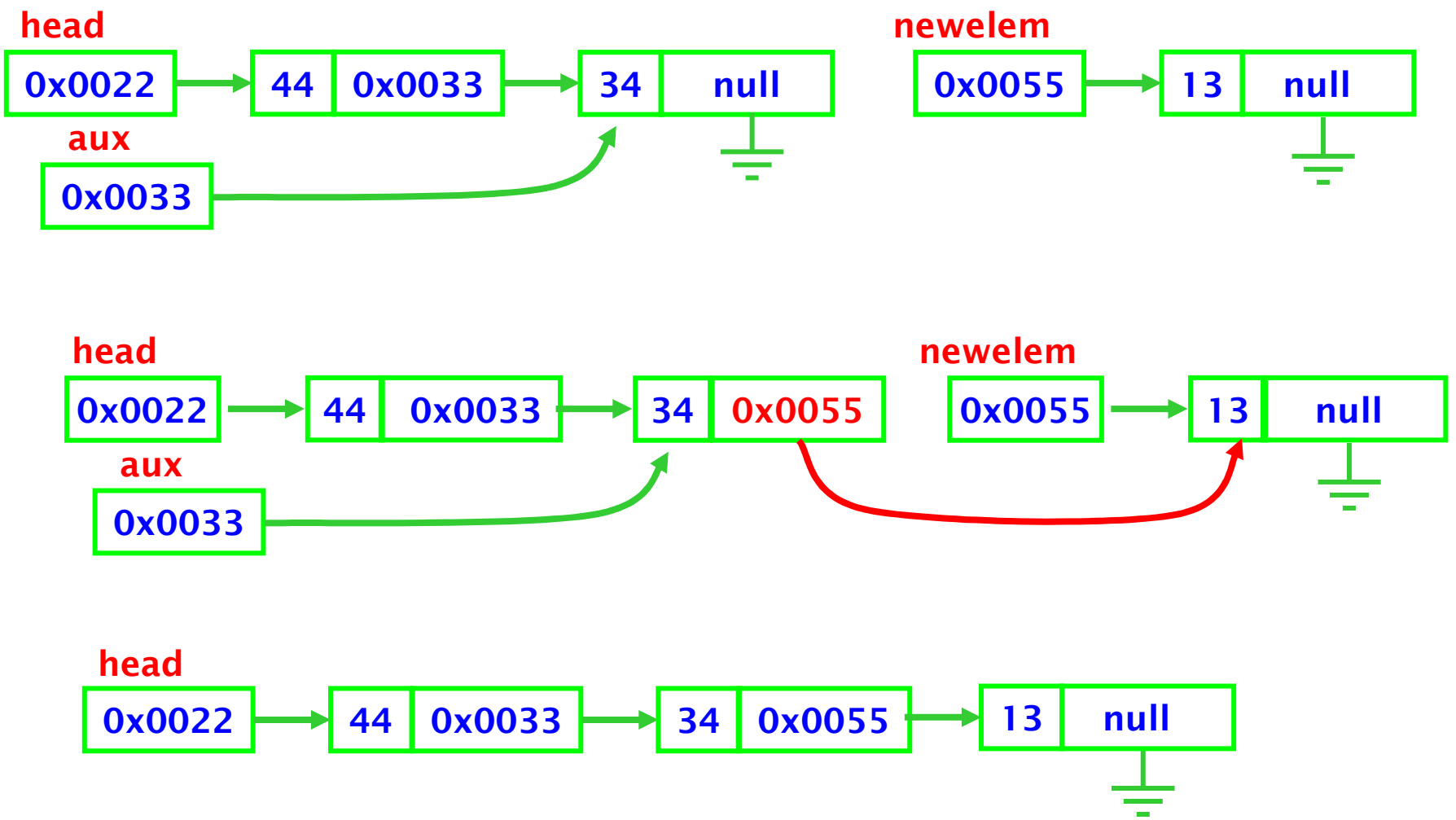
Se la lista non esiste (head==null) viene creata una nuova lista.

# Inserimento in coda

I passi da eseguire per effettuare l'inserimento in coda sono:

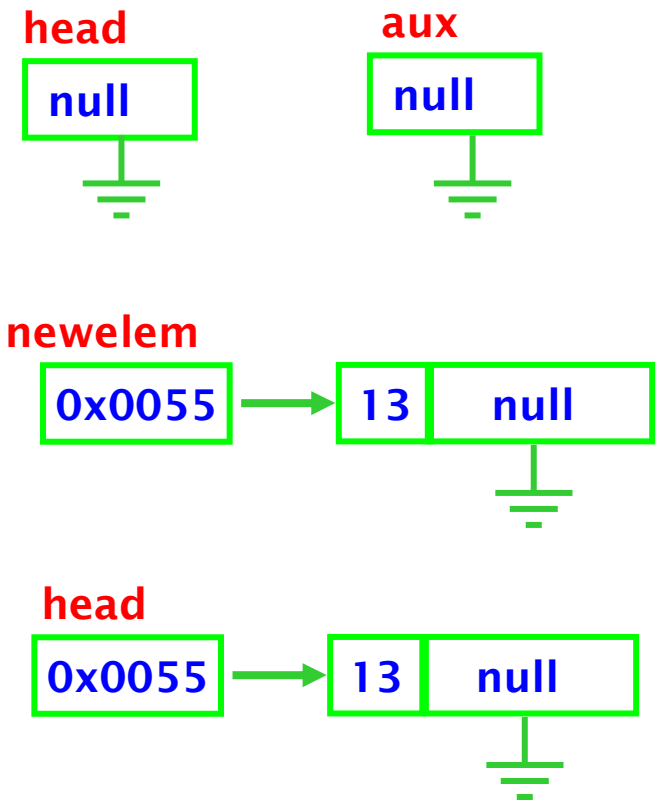
- scorrere la lista fino all'ultimo elemento;
- istanziare un nuovo elemento
- collegare l'ultimo elemento trovato con il nuovo elemento appena creato.

# Inserimento in coda



# Inserimento in coda

Se la lista è vuota bisogna però creare il primo elemento (modificare il valore di head)



# Inserimento in coda

Definiamo un metodo che ci indichi se la lista è vuota.

```
public class Lista{
	private Nodo head;

	public Lista(){
		head = null;
	}

	public boolean IsEmpty(){
		return (head == null ? true : false);
	}
	…
```

# Inserimento in coda

```
public class Lista{
  private Nodo head;

  public Lista(){
    head = null;
  }

  public void InsertTail(int val){
    if (IsEmpty() )
      head = new Nodo(val);
    else{
      Nodo aux = head;
      for(; aux.getNext()!= null; aux = aux.getNext() );
      aux.setNext( new Nodo(val) );
    }
  }
  …
```

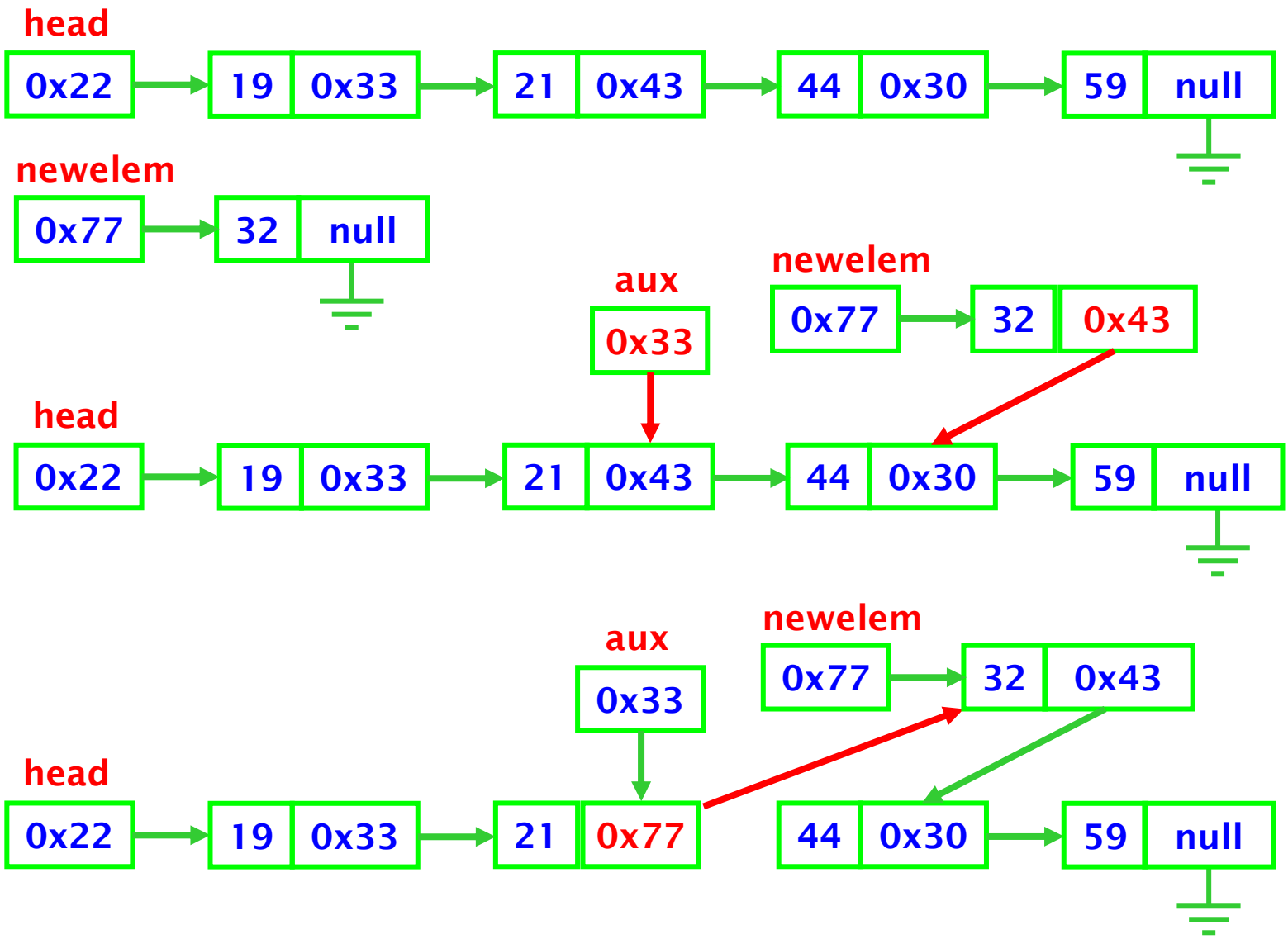
# Inserimento Ordinato

L'inserimento ordinato comporta l'aggiunta di un elemento in una qualunque posizione all'interno della lista.

Le operazioni da eseguire sono:

- scorrere la lista confrontando gli elementi già presenti con quello da inserire;
- arrestarsi appena si trova un elemento maggiore (o minore)
- collegare la parte con gli elementi minori al nuovo elemento
- collegare la parte con gli elementi maggiori.

# Inserimento Ordinato



# Inserimento Ordinato

```
public void InsertOrdered(int val){
  if (IsEmpty() )
    head = new Nodo(val);
  else
    if( head.getInfo() > val)
      head = new Nodo(val,head);
    else{
      Nodo aux = head;
      for(;( aux.getNext()!=null) &&
              (aux.getNext()).getInfo() < val );
               aux=aux.getNext() );

      aux.setNext( new Nodo(val,aux.getNext() ) );
    }
}
```

# Ricerca di un elemento

Per la ricerca, dobbiamo distinguere due casi:

- la lista è ordinata;
- la lista non è ordinata.

Nel primo caso possiamo arrestare la ricerca quando troviamo due elementi consecutivi rispettivamente maggiore e minore dell'elemento da cercare.

Nel secondo caso, se non troviamo prima l'elemento desiderato, dobbiamo arrivare fino al termine della lista.

# Ricerca di un elemento

```
public Nodo SearchOrd(int key){
  Nodo aux = head;
  for(; aux!=null&&aux.getInfo()<key; aux=aux.getNext());
  if(aux != null && aux.getInfo() == key)
    return aux;
  return null;
}


public Nodo Search(int key){
  Nodo aux = head;
  for(; aux!=null&&aux.getInfo()!=key;
      aux= aux.getNext());
  return aux;
}
```

# Cancellazione

Per la cancellazione abbiamo:

- cancellazione del primo elemento
- cancellazione dell'ultimo elemento
- cancellazione di un elemento specificato.

Nel terzo caso dobbiamo prima cercare l'elemento da cancellare.

# Eccezioni

```
public class EmptyListException  extends RuntimeException{

  public  EmptyListException(){
    super("Lista vuota!");
  }

  public  EmptyListException(String msg){
    super(msg);
  }
}
```

# Cancellazione

```
public int DeleteHead(){
  if(IsEmpty() )
    throw new EmptyListException();
  else{
    Nodo aux = head;
    head = head.getNext();
    return aux.getInfo();
  }
}
```

# Cancellazione

```
public int DeleteTail(){

  if (IsEmpty())
    throw new EmptyListException();
  else{
    Nodo aux = head;
    Nodo prev = null;
    for( ; aux.getNext() != null;
           prev = aux, aux = aux.getNext() );
    if(prev == null)
      head = null;
    else
      prev.setNext( null );
    return aux.getInfo();
  }
}
```

# Cancellazione

```
public Nodo DeleteKey(int key){
  if (IsEmpty())
    throw new EmptyListException();
  else{
    Nodo aux = head;
    Nodo prev = null;
    for(; aux != null && aux.getInfo() != key;
          prev = aux, aux = aux.getNext() );
    if (aux != null)
      if(prev == null)
        head = head.getNext() ;
      else
        prev.setNext(aux.getNext() );
    return aux;
  }
}
```

# Esercizio

Definire ed implementare una classe lista che abbia i seguenti metodi:

```
void rewind()

int getNextElem()
```

Il metodo **`getNextElem()`** scorre la lista e ad ogni invocazione restituisce i vari nodi, in sequenza.

Al termine lancia un'eccezione.

Il metodo **`rewind()`** azzera l'esecuzione dello scorrimento (fa ripartire dalla testa).

# Esercizio

- Applicazione 1: creare 1000 record di tipo persona e scriverli in un file di Object appropriato.

- Applicazione 2: caricare i record dal file e inserirli in una lista dinamica.
  Estrarre tutti gli elementi di posto pari e conservarli in un primo file.
  Estrarre tutti gli elementi di posto dispari e conservarli in un secondo file.

# Esercizio

- Creare 40 oggetti carta da gioco in modo da formarne un mazzo completo.
- Conservare il mazzo come lista.
- Rimescolare il mazzo.
- Distribuire le carte a N giocatori (N liste) con N divisore di 40.
- Il primo giocatore getta una carta a caso, i giocatori successivi a turno se hanno una carta di valore immediatamente successivo la gettano procedendo prima in senso crescente fino a giungere al re e poi in senso descrescente fino all'asso e così via.
- Se un giocatore NON ha la carta successiva da mettere PASSA.
- Vince il giocatore che termina per primo le carte.
- PRIMA DI IMPLEMENTARE LA SIMULAZIONE DEL GIOCHINO CAPIRE BENE CHE TIPO DI LISTE VI SARA' PIU' UTILE CONSIDERANDO QUALI SONO LE OPERAZIONI CHE AVRETE BISOGNO DI FARE.

# Esercizio

- Creare un oggetto (genere “record anagrafico”) a piacere;
- Scrivere un programma che “testi” le implementazioni delle liste come segue:
  - Creare una lista;
  - Quanti record si possono mettere in lista usando la implementazione linked list prima di esaurire lo spazio di memoria gestito dalla JVM?
  - Eseguire N operazioni (scelte casualmente) tra inserimento, cancellazione e locate di N record (attenzione a gestire le eccezioni, le ricerche su liste vuote etc etc)
  - Confrontare i tempi di esecuzione (N molto grande a piacere) se si usa la implementazione “vostra” array, oppure la implementazione “vostra” linked list.