# INTRODUZIONE

- Il tipo di dati astratto *insieme* (*set*) definisce operazioni, quali inserimento e rimozione, su collezioni di oggetti che presentano molteplicità uno, cioè non sono ammessi *duplicati*.
- Una possibile realizzazione del ADT *insieme* si basa su *tabelle hash con liste concatenate*.
- Le specifiche degli insiemi sono simili a quelle delle liste (si implementa anche un iteratore).



# SET: specifica

- Una possibile interfaccia è la seguente:

```
import java.util.Iterator;
public interface Set {
  boolean add(Object x);//inserisce x  se non è
                                    già presente
  boolean remove(Object x);//rimuove l'oggetto x
                                    se presente
  boolean contains(Object x);//verifica la
                          presenza dell'oggetto x
  boolean isEmpty();// verifica se l'insieme è
                                logicamente vuoto
  void clear();// svuota l'insieme
  Iterator iterator();//restituisce un iteratore
            per scandire gli elementi in sequenza
}
```



# SET: esempio

- Una porzione di codice che crea un *insieme* basato su una *tabella hash* di cinque elementi e vi inserisce 20 interi casuali di valore minore di 50.

```
Random rnd = new Random();
HashSet   S = new HashSet(5);
Integer[] v = new Integer[20];
for(int i=0; i<20;i++){
  Integer n = new Integer(rnd.nextInt(50));
          v[i]=n;
          S.add(n);
    }
```

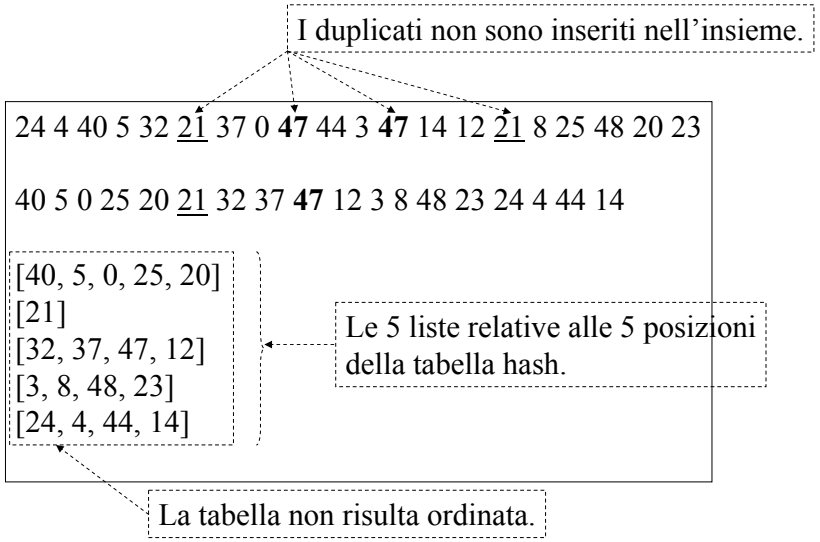


# SET: esempio

- Si stampano i numeri casuali in ordine di generazione e il contenuto dell'insieme.

```
for(int i=0; i<20;i++)
        System.out.print(v[i]+" ");
System.out.println();
for(Iterator i=S.iterator(); i.hasNext(); )
        System.out.print(i.next()+" ");
System.out.println();
System.out.print(S);
```

# SET: esempio






# SET: esempio, correttore ortografico

- Vediamo come sviluppare un semplice correttore ortografico.
- Per prima cosa si carica da file (`It.dic`) un dizionario di 277433 parole, poi si legge il file di testo (`es1.txt`) che si vuole correggere. Ogni nuova parola letta è cercata nel dizionario: se tale parola non viene trovata allora è visualizzata. Inoltre si incrementa un contatore degli errori.



# SET: esempio, correttore ortografico

```
HashSet dizionario = new HashSet(277433);
BufferedReader in = new BufferedReader(new
                      FileReader("It.dic"));
String str;
while( (str=in.readLine()) != null)
     dizionario.add(str);
in.close();
String[] strv;
int c=0;
in = new BufferedReader(new
                    FileReader("es1.txt"));
```



# SET: esempio, correttore ortografico

```
System.out.println("Parole non trovate:");
while( (str=in.readLine()) != null){
      strv=str.split(" ");
      for (int j=0;j<strv.length;j++)
            if (!dizionario.contains(strv[j])){
                  System.out.print(strv[j]+", ");
                  c++;
                  }
      }
      System.out.println("\n"+c+"\n");
      in.close();
```

# SET: esempio, correttore ortografico

- Un breve file di prova:

introdurre i principali metodi utilizzati per organizzare e rappresentare le informazioni , le strutture dati , al finne di ottenerne una elaborazione efficiente , gli algoritmis

- L'output dell'applicazione:

Parole non trovate:
finne, algoritmis,
2



# SET: esempio, correttore ortografico

- Dal punto di vista delle prestazioni si ottengono buoni risultati.
- Si è utilizzato un file con circa 58000 parole e si è usato lo stesso dizionario dell'esempio precedente.
- I tempi di esecuzione sono i seguenti:
  - Circa 1 secondo per caricare il dizionario.
  - Circa 0.5 secondi per elaborare il file di 58000 parole.



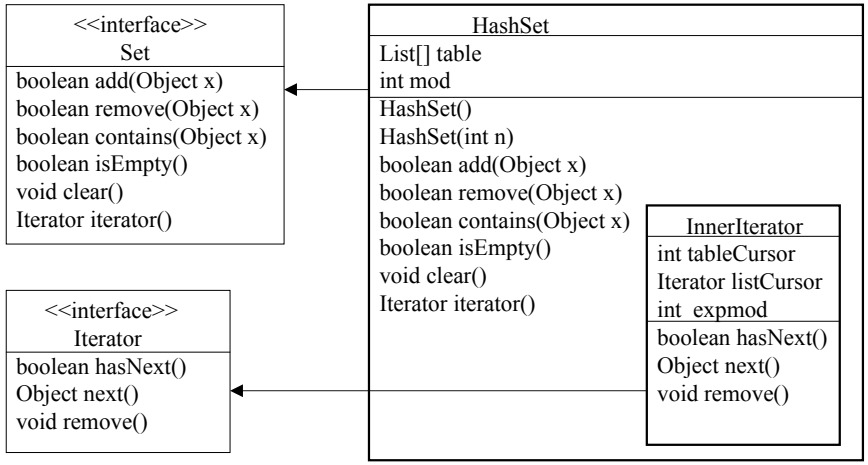
# SET: implementazione

- Una possibile realizzazione dell'insieme si basa su *tabelle hash con liste concatenate*. L'insieme è realizzato dalla classe `HashSet` che implementa l'interfaccia `Set`.
- Un oggetto di tipo `HashSet` è manipolato utilizzando i metodi resi disponibili dall'interfaccia `Set`, non si accede direttamente alla tabella hash interna. È un *tipo di dato astratto*, ADT.



# SET: implementazione

Si realizza l'interfaccia `Set` con la classe `HashSet` e l'interfaccia `Iterator` con la classe interna `InnerIterator`.



# SET: implementazione, HashSet

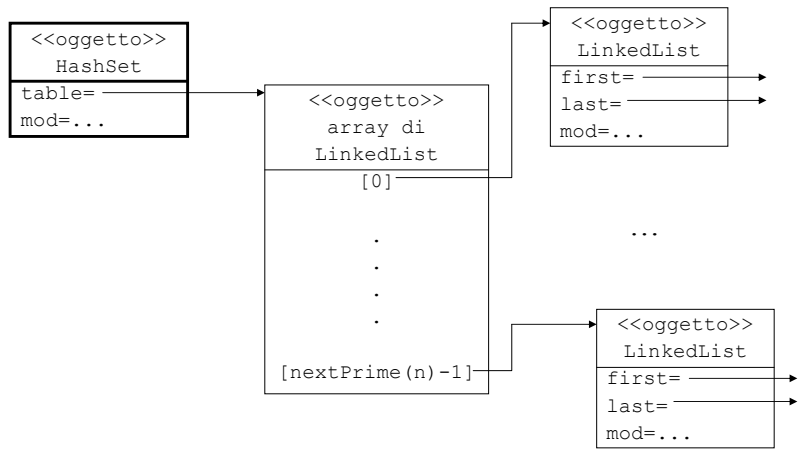
- Vediamo una possibile implementazione dell'insieme.

- Si usa un array di `LinkedList` per realizzare una *tabella hash con liste concatenate*.
- Inoltre è utilizzata una variabile intera `mod`, che tiene conto delle modifiche all'insieme, per evitare che modifiche concorrenti possano interferire con il comportamento degli iteratori.



# HASHSET

- Una possibile rappresentazione grafica:






# HASHSET

- La realizzazione dei metodi *sfrutta* le operazioni fornite dalla classe `LinkedList`.
- Vi sono dei *metodi interni* per la gestione dei numeri primi e dell'array di liste:
  - Il metodo `nextPrime()` restituisce il numero primo successivo al suo argomento; è utilizzato nel costruttore per avere una tabella hash con un numero primo di posizioni.
  - Il metodo `theListOf()` determina la lista che deve contenere un dato oggetto, basandosi sul codice hash di tale oggetto.



# HASHSET

```
public class HashSet implements Set {
   List[] table;
   int mod;
```

Numero di modifiche: chiamate a `remove()`, `add()` e `clear()`.

Array di liste.

```
   public HashSet() {
      this(97);
   }
   public HashSet(int n) {
      table = new LinkedList[nextPrime(n)];
      for (int i=0; i<table.length; i++)
           table[i]=new LinkedList();
      mod=0;
   }
```

# HASHSET: numeri primi

```
int nextPrime(int n){
    if(n>1){
         boolean[] p = primes(2*n+1);
         for (int i=n; i<p.length; i++)
           if (p[i])
               return i;
    }
    return 2;}

boolean[] primes(int n){
    boolean[] p = new boolean[n];
    for(int i=2; i<n;i++)
          p[i]=true;
    for(int i=2; i<n;i++){
          if (p[i]!=false){
                for(int j=i;(long)j*i<n;j++)
                      p[i*j]=false;}
    }
    return p;}
```



# HASHSET: gestione tabella hash di liste concatenate

```
List theListOf(Object x){
   if (x==null)
     throw new IllegalArgumentException();
   int index = x.hashCode();
   index = index%table.length;
   if(index<0)
        index+=table.length;
   return table[index];
}
```

Restituisce la lista che deve contenere l'oggetto `x` basandosi sul codice hash di tale oggetto.



# HASHSET: inserimento

```
public boolean add(Object x) {
   List xlist=theListOf(x);
   if (xlist.contains(x))
        return false;
   else{
        xlist.add(x);
        mod++;
        return true;
   }
}
```

Se la lista contiene già un oggetto `x` allora non viene inserito

L'inserimento di un oggetto è semplice perché sfrutta i metodi della `LinkedList`.



# HASHSET: rimozione e ricerca

```
public boolean remove(Object x) {
   List xlist=theListOf(x);
   if (xlist.remove(x)){
        mod++;
        return true;
   }else
        return false;
}

public boolean contains(Object x) {
   List xlist=theListOf(x);
   return xlist.contains(x);
}
```

# HASHSET: controllo e svuotamento

```
public boolean isEmpty() {
   for (int i=0; i<table.length; i++)
        if (!table[i].isEmpty())
             return false;
   return true;
}
public void clear() {
   for (int i=0; i<table.length; i++)
        table[i].clear();
   mod++;
}
```

Devono essere gestite tutte le liste della tabella.



# HASHSET: iteratore

- Il metodo `iterator()` restituisce un oggetto iteratore sulla tabella hash e di conseguenza sulle relative liste per *scandire gli elementi in sequenza a partire dall'inizio della tabella*.

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

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



# HASHSET: implementazione iteratore

- Vediamo una possibile implementazione dell'iteratore.
- Si usa una variabile intera `tableCursor` per indicare la successiva *lista* non vuota, è una posizione della *tabella hash*. La variabile `listCursor` è un *iteratore* sulla lista corrente, è un iteratore di una `LinkedList`.
- Inoltre è utilizzata una variabile intera `expmod`, che tiene conto delle modifiche all'*insieme* durante l'iterazione, per verificare se sono avvenute eventuali modifiche concorrenti.



# HASHSET: INNERITERATOR

```
public class HashSet implements Set {
  …
  class InnerIterator implements Iterator{
      int tableCursor;
      Iterator listCursor;
      int expmod;

      void nextList(){
            do{
                  ++tableCursor;
            }while((tableCursor < table.length) &&
                  (table[tableCursor].isEmpty()));
  …}
  }
  …
}
```

Classe interna

Si usa il metodo interno `nextList()` per indicare la successiva lista non vuota.

# HASHSET: INNERITERATOR

```
InnerIterator(){
  tableCursor=-1;
  nextList();
  if (tableCursor < table.length){
     listCursor=table[tableCursor].iterator();
     nextList();
  }else{
     listCursor=null;}
  expmod=mod;
}
```

Il costruttore cerca una *lista* non vuota e costruisce un *iteratore* su tale lista.



# HASHSET: INNERITERATOR

```
public boolean hasNext(){
  if (mod!=expmod)
     throw new IllegalStateException();
  if (listCursor == null)
     return false;
  if (listCursor.hasNext())
     return true;
  return (tableCursor < table.length);
}
```

Se l'iteratore *corrente* di lista ha terminato la propria lista, allora si controlla se esiste una successiva lista non vuota.



# HASHSET: INNERITERATOR

```
public Object next(){
  if(!hasNext())
     throw new IllegalStateException();
  if (listCursor.hasNext())
     return listCursor.next();
  listCursor=table[tableCursor].iterator();
  nextList();
  return listCursor.next();
}
```

Se l'iteratore *corrente* di lista ha terminato la propria lista, allora si costruisce un *nuovo* iteratore sulla successiva lista non vuota.



# HASHSET: INNERITERATOR

```
public void remove(){
  if(mod++!=expmod++)
     throw new IllegalStateException();
  if(listCursor == null)
     throw new IllegalStateException();
  listCursor.remove();
}
```

Rimuove l'oggetto restituito dall'ultima chiamata al metodo `next()` e deve essere invocato una sola volta dopo una chiamata al metodo `next()`.

# HASHSET: prestazioni

- Si indica con *lf* il fattore di carico (*load factor*): il rapporto tra il numero di oggetti memorizzati nella tabella e la dimensione di tale tabella.
- Si può dimostrare che il costo computazionale dei metodi descritti è proporzionale a *1+lf*.
- Si può pensare che il costo per trovare una posizione nella tabella hash è 1, poi si deve scorrere la relativa lista per trovare/inserire l'oggetto. La lista è lunga (se la distribuzione delle chiavi è uniforme) proprio *lf*.



# HASHSET: prestazioni

- In media ogni lista contiene un numero di oggetti pari al rapporto tra il numero di oggetti memorizzati nella tabella hash e la dimensione di tale tabella, quindi *lf*.
- Se il fattore di carico cresce e quindi le liste concatenate contengono molti oggetti, allora è necessario aumentare la dimensione della tabella e inserire tutti gli oggetti nella nuova tabella.



# HASHSET: prestazioni

- Con riferimento all'applicazione del correttore ortografico si consideri di controllare gli oggetti contenuti in una banca dati (in memoria): tale collezione è composta da 300000 elementi, e si eseguono 200000 controlli con una probabilità del 10% di non trovare l'oggetto cercato.

| Fattore di carico | Tempo esecuzione |
|---|---|
| 0.5 | 203 ms |
| 1 | 235 ms |
| 2 | 265 ms |

- La stessa applicazione sviluppata con un array ordinato e ricerca binaria impiega 610 ms.