COLECȚII

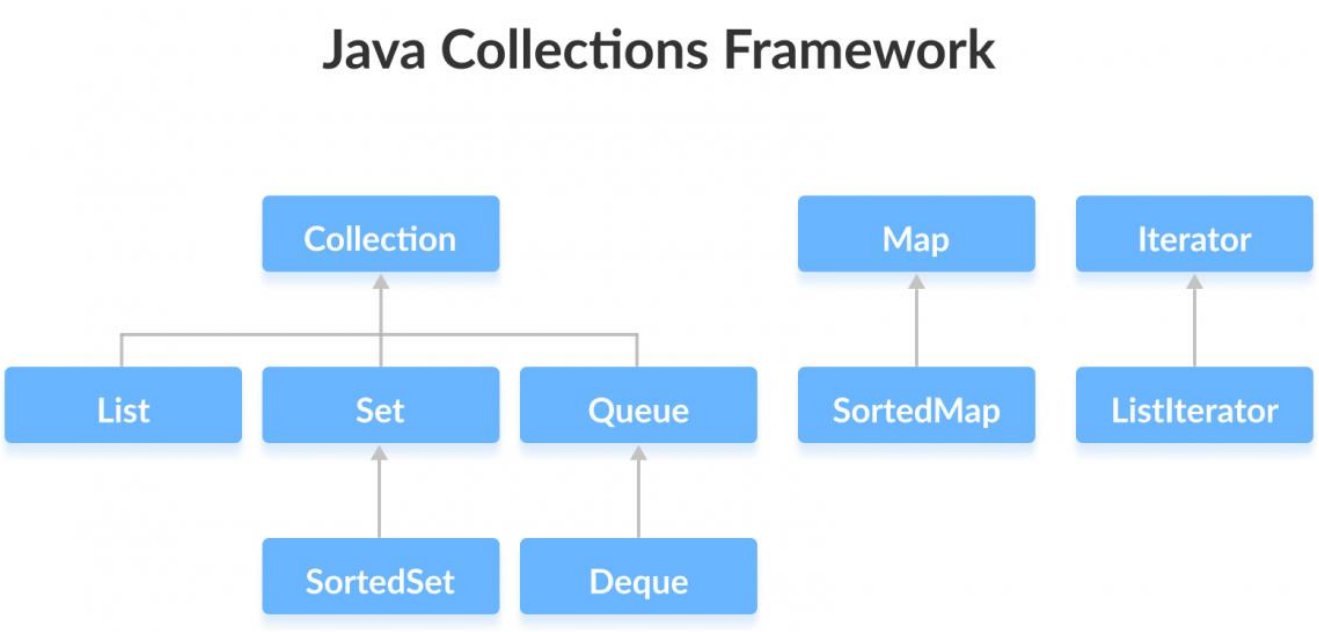
O *colecție* este un obiect container care grupează mai multe elemente într-o structura unitară. Intern, elementele dintr-o colecție se află într-o relație specifică unei structuri de date (lineară, asociativă, arborescentă etc.), astfel încât asupra lor se pot efectua operații de căutare, adăugare, modificare, ștergere, parcurgere etc.

Limbajul Java oferă un framework/API performant pentru crearea și managementul structurilor dinamice de date (tablou, liste, mulțime, tabelă de asocieri etc) – *Java Collections Framework*, astfel încât programatorul să fie degrevat de implementarea optimă a lor.

Framework-ul Collections are o arhitectură bazată pe interfețe și clase care permite reprezentarea și manipularea unitară a colecțiilor, într-un mod independent de detaliile de implementare, astfel:

- *interfețe* – definesc într-un mod abstract operațiile specifice diverselor colecții;
- *clase* – conțin implementări concrete ale colecțiilor definite în interfețe, iar începând cu Java 1.5 ele sunt generice (tipul de dată concret al elementelor din colecție se precizează prin operatorul <> (operatorul diamond): `List<Persoana> lp = new ArrayList<>();`
- *algoritmi polimorfici* – sunt metode statice, grupate în care clasa utilitară `Collections`, care implementează optim operații generice caracteristice colecțiilor de date, cum ar fi: căutare, sortare, copiere, determinarea minimului/maximului etc.

Principalele ierarhii de interfețe puse la dispoziție de framework-ul Java Collections sunt reprezentate în figura de mai jos (sursa: https://www.programiz.com/java-programming/collections):



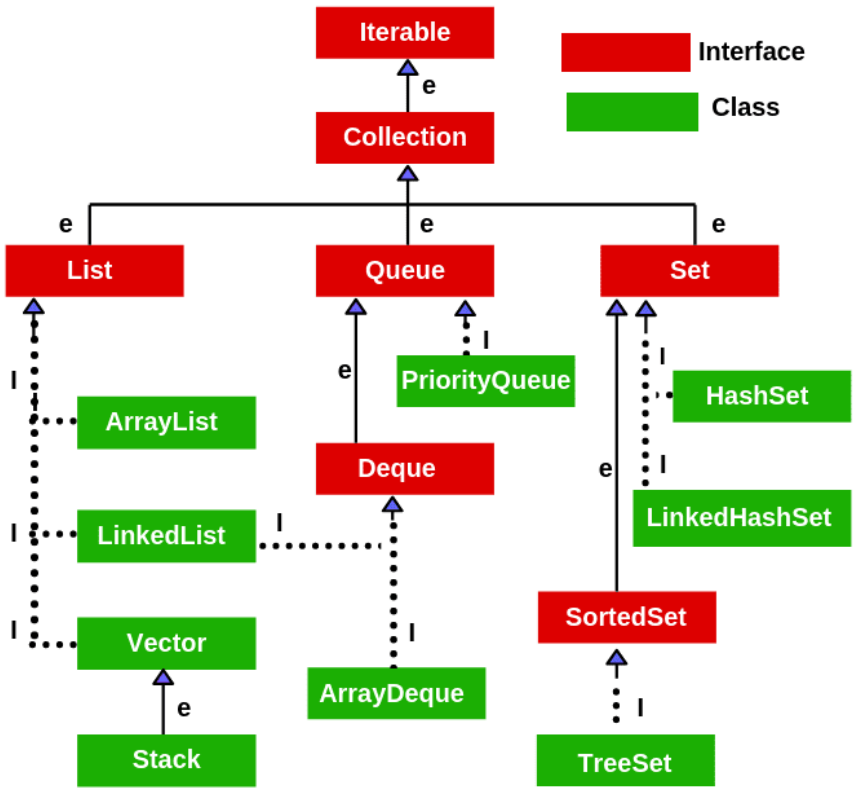
Interfața Collection

Nucleul principal al framework-ului Java Collections este reprezentat de interfața `Collection` care conține o serie de metode fundamentale de prelucrare specifice tuturor colecțiilor. O parte dintre metodele uzuale ale interfeței `Collection` sunt:

- `public int size()` – returnează numărul total de elemente din colecție;
- `public boolean add(E e)` – inserează în colecția curentă elementul `e`;
- `public boolean addAll(Collection<E> c)` – inserează în colecția curentă toate elementele din colecția `c`;
- `public boolean remove(Object e)` – șterge elementul `e` din colecția curentă;
- `public boolean contains(Object e)` – caută în colecția curentă elementul `e`;
- `public Iterator iterator()` – returnează un iterator pentru colecția curentă;
- `public Object[] toArray()` – realizează conversia colecției într-un tablou cu obiecte de tip `Object`.

Alte metode ale interfeței `Collection` sunt prezentate aici: https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html.

Ierarhia formată din interfețele care extind interfața `Collection`, precum și clasele care le implementează, este reprezentată în figura de mai jos (sursa: https://www.scientecheasy.com/2018/09/collection-hierarchy-in-java-collections-class.html/?189db0):



Parcurgerea unei colecții presupune obținerea, pe rând, a unei referințe către fiecare obiect din colecție. O modalitate generală de a parcurge o colecție, independent de tipul ei, este reprezentată de *iteratori*. Astfel, vârful celor ierarhiei `Collection` este interfața `Iterable`.

Interfața List

Interfața `List` extinde interfața `Collection` și modelează o colecție de elemente ordonate care permite inclusiv memorarea elementelor duplicate.

Interfața `List` adaugă metode suplimentare fata de interfața `Collection`, corespunzătoare operațiilor care utilizează index-ului fiecărui element, considerat ca fiind de un tip generic `E`:

- accesarea unui element: `E get(int index),E set(int index);`
- adăugarea/ștergere element: `void add(int index, E element),E remove(int index);`
- determinarea poziției unui element în cadrul colecției: `int indexOf(Object e), int lastlndexOf(Object e).`

Cele mai utilizate implementări ale interfeței `List` sunt clasele `ArrayList` și `LinkedList`.

Clasa `ArrayList`

Clasa `ArrayList` oferă o implementare a unei liste utilizând un tablou unidimensional care poate fi redimensionat dinamic:

```
List<Tip> listaTablou = new ArrayList<>();
ArrayList<Tip> listaTablou = new ArrayList<>();
```

Se poate observa cum o colecție `ArrayList` poate fi referită atât printr-o referință de tipul interfeței implementate (`List`), cât și printr-o referință de tipul colecției.

Capacitatea implicită a unei astfel de liste este egală cu 10, iar pentru a specifica explicit o altă capacitate se poate utiliza un constructor care primește ca argument un număr întreg:

```
List<Tip> listaTablou = new ArrayList<>(50);
```

Exemple:

```
List<Integer> lista1 = new ArrayList<>();
lista1.add(0, 1); // adaugă 1 pe poziția 0
lista1.add(1, 2); // adaugă 2 pe pozția 1
System.out.println(lista1); // [1, 2]

List<Integer> lista2 = new ArrayList<Integer>();
lista2.add(1); // adaugă 1 la sfârșitul listei
lista2.add(2); // adaugă 2 la sfârșitul listei
lista2.add(3); // adaugă 3 la sfârșitul listei
System.out.println(lista2); // [1, 2, 3]
```

```
// adaugă elementele din lista2 începând cu poziția 1
lista1.addAll(1, lista2);
System.out.println(lista1); // [1, 1, 2, 3, 2]

// șterge elementul de pe poziția 1
lista1.remove(1);
System.out.println(lista1); // [1, 2, 3, 2]

// afișează elementul aflat pe poziția 3
System.out.println(lista1.get(3)); // 2

// înlocuiește valoarea aflată pe poziția 1  cu valoarea 5
lista1.set(1, 5);
System.out.println(lista1); // [1, 5, 3, 2]
```

Observații:

- Accesarea unui element se realizează cu complexitatea $\mathcal{O}(1)$.
- Adăugarea unui element la sfârșitul listei prin metoda `add(T elem)` se realizează cu complexitatea $\mathcal{O}(1)$ dacă nu este depășită capacitatea listei sau cu complexitatea $\mathcal{O}(n)$ în caz contrar.
- Adăugarea unui element pe o anumită poziție prin metoda `add(E element, int index)` se realizează cu complexitatea $\mathcal{O}(n)$.
- Căutarea sau ștergerea unui element se realizează cu complexitatea $\mathcal{O}(n)$.

Clasa `LinkedList`

Clasa `LinkedList` oferă o implementare a unei liste utilizând o listă dublu înlănțuită, astfel fiecare nod al listei conține o informație de tip generic `E`, precum și două referințe: una către nodul anterior și una către nodul următor.

Constructorii clasei `LinkedList` sunt:

- `LinkedList()` – creează o listă vidă;
- `LinkedList(Collection C)` – creează o listă din elementele colecției `C`.

Pe lângă metodele implementate din interfața `List`, clasa `LinkedList` conține și câteva metode specifice:

- accesarea primului/ultimului element: `E getFirst(), E getLast();`
- adăugarea la începutul/sfârșitul listei: `void addFirst(E elem), void addLast(E elem);`
- ștergerea primului/ultimului element: `E removeFirst(),E removeLast().`

Exemple:

```
LinkedList<String> lista = new LinkedList<>();

// adăugarea unor elemente în listă
lista.add("A");
lista.add("B");
lista.addLast("C");
lista.addFirst("D");
lista.add(2, "E");
lista.add("F");
lista.add("G");
System.out.println(lista); // [D, A, E, B, C, F, G]
```

```
// ștergerea unor elemente din  listă
lista.remove("B");
lista.remove(3);
lista.removeFirst();
lista.removeLast();
System.out.println(lista); // [A, E, F]

// căutarea unui element în listă
boolean rezultat = lista.contains("E");
System.out.println(rezultat); // true

// operații de accesare a unui element
Object element = lista.get(2);
System.out.println(element); // F
lista.set(2, "Y");
System.out.println(lista); // [A, E, Y]
```

Observații:

- Accesarea unui element se realizează cu complexitatea $\mathcal{O}(n)$.
- Adăugarea unui element la sfârșitul listei, folosind metoda `add(E elem)`, se realizează cu complexitatea $\mathcal{O}(1)$.
- Adăugarea unui element pe poziția `index`, folosind metoda `add(E elem, int index)`, se realizează cu o complexitate egală cu $\mathcal{O}(n)$.
- Căutarea unui element se realizează cu o complexitate egală cu $\mathcal{O}(n)$.
- Ștergerea unui element se realizează cu o complexitate egală cu $\mathcal{O}(1)$, dacă se specifică indexul elementului.

Pentru ca o aplicație să obțină performanțe cât mai bune din punct de vedere al timpului de executare, trebuie selectată colecția corespunzători funcționalității aplicației, astfel:

- dacă operațiile de accesare sunt predominante în cadrul aplicației, atunci se preferă utilizarea colecției `ArrayList`;
- dacă operațiile de actualizare (inserare/ștergere) sunt predominante, atunci se preferă utilizarea colecției `LinkedList`.

Pe lângă cele două implementări ale interfeței `List`, mai există și alte clase care o implementează:

- clasa `Vector` – are o funcționalitate similară clasei `ArrayList` și oferă metode sincronizate specifice aplicațiilor cu mai multe fire de executare;
- clasa `Stack` – extinde clasa `Vector` și oferă o implementare a unui vector cu funcționalitățile specifice structurii de date *stiva* (LIFO) .

Interfața Set

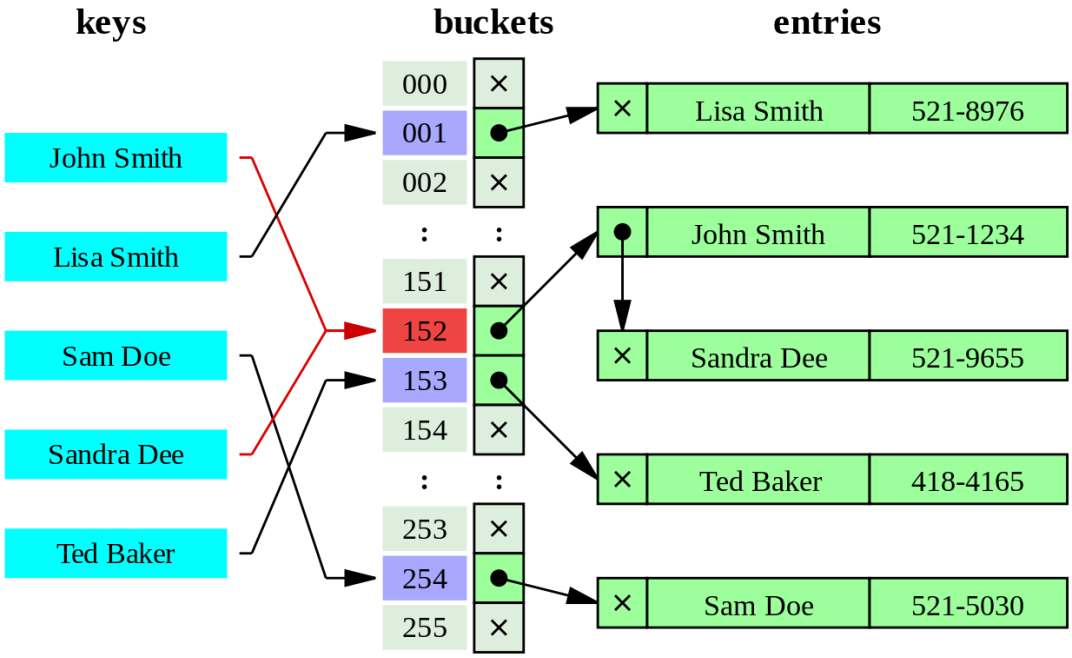
Interfața `Set` extinde interfața `Collection` și modelează o colecție de elemente care nu conțin duplicate, respectiv o colecție de tip mulțime.

Interfața `Set` nu adaugă metode suplimentare celor existente în interfața `List` și este implementată în clasele `HashSet`, `TreeSet` și `LinkedHashSet`.

Clasa `HashSet`

Într-un curs anterior, am văzut cum fiecare obiect are asociat un număr întreg, numit *hash-code*, puternic dependent față de orice modificare a datelor membre ale obiectului. Hash-code-ul unui obiect se calculează în metoda `int hashCode()`, moștenită din clasa `Object`, folosind algoritmi specifici care implică valorile datelor membre relevante pentru logica aplicației.

Clasa `HashSet` implementează o mulțime folosind o *tabelă de dispersie* (hash table). O tabelă de dispersie este un tablou unidimensional, numit *bucket array*, în care indexul unui element se calculează pe baza hash-code-ului său. Fiecare componentă a bucket array-ului va conține o listă cu obiectele care au același hash-code (*coliziuni*), așa cum se poate observa în figura de mai jos (sursa: https://en.wikipedia.org/wiki/Hash_table):



Practic, o operație de inserare a unui obiect în tabela de dispersie presupune parcurgerea următorilor pași:

- se apelează metoda `hashCode` a obiectului respectiv, iar valoarea obținută se utilizează pentru a calcula indexul bucket-ului asociat obiectului respectiv;
- dacă bucket-ul respectiv este vid, atunci se adaugă direct obiectul respectiv și operația de inserare se încheie;
- dacă bucket-ul respectiv nu este vid, se parcurge lista asociată și, folosind metoda `equals`, se verifică dacă obiectul este deja inserat în tabelă, iar în caz negativ obiectul se adaugă la sfârșitul listei.

Evident, într-un mod asemănător, se vor efectua și operațiile de căutare, actualizare sau ștergere.

Se observă foarte ușor faptul că performanțele unei tabele de dispersie sunt puternic influențate de performanțele algoritmului de calcul al hash-code-ului unui obiect, respectiv acesta trebuie să fie sensibil la orice modificare a datelor membre pentru a minimiza numărul de coliziuni (obiecte diferite din punct de vedere al conținutului, dar care au același hash-code), ideal fiind ca hash-code-ul unui obiect să fie unic. În acest caz, lista asociată oricărui bucket va fi foarte scurtă, deci operațiile de căutare/inserare/ștergere/modificare vor avea complexitatea $\mathcal{O}(1)$, altfel, în cazul existenței multor coliziuni, complexitatea poate ajunge $\mathcal{O}(n)$.

Un alt aspect foarte important îl constituie implementarea/rescrierea corectă și a metodei `equals`, moștenită tot din clasa `Object`, în concordanță cu implementarea metodei `hashCode`, respectând următoarele reguli:

- metoda `hashCode` trebuie să returneze aceeași valoare în timpul rulării unei aplicații, indiferent de câte ori este apelată, dacă starea obiectului nu s-a modificat, dar nu trebuie să furnizeze aceeași valoare în cazul unor rulări diferite;
- două obiecte egale din punct de vedere al metodei `equals` trebuie să fie egale și din punct de vedere al metodei `hashCode`, deci trebuie să aibă și hash code-uri egale;
- nu trebuie neapărat ca două obiecte diferite din punct de vedere al conținutului să aibă hash-code-uri diferite, dar, dacă acest lucru este posibil, se vor obține performanțe mai bune pentru operațiile asociate unei tabele de dispersie.

Dacă a doua regulă nu este respectată, adică două obiecte egale din punct de vedere al conținutului (metoda `equals`) au hash-code-uri diferite (metoda `hashCode`), atunci operațiile de căutare/inserare într-o tabelă de dispersie vor fi incorecte. Astfel, în cazul în care se încearcă inserarea celui de-al doilea obiect după inserarea primului, operația de căutare a celui de-al doilea obiect se va efectua după valoarea hash-code-ului său, diferită de cea a primului obiect, deci îl va căuta în alt bucket și nu îl va găsi, ceea ce va conduce la inserarea și a celui de-al doilea obiect în tabela, deși el are același conținut cu primul obiect!

De obicei, acest aspect negativ apare în momentul în care programatorul nu rescrie metodele `hashCode` și `equals` într-o clasă ale cărei instanțe vor fi utilizate în cadrul unor colecții bazate pe tabele de dispersie, deoarece, implicit, metoda `hashCode` furnizează o valoare calculată pe baza referinței obiectului respectiv, iar metoda `equals` testează egalitatea a două obiecte comparând referințele lor. Astfel, două obiecte diferite cu același conținut vor fi considerate diferite de metoda `equals` și vor avea hash-code-uri diferite!

Exemplu: Considerăm definită clasa Persoana în care nu am rescris metodele `hashCode` și `equals`:

```
HashSet<Persoana> lp = new HashSet<>();
Persoana p1 = new Persoana("Popescu Ion", 23);
Persoana p2 = new Persoana("Popescu Ion", 23);

lp.add(p1);
lp.add(p2);

System.out.println(lp.size());
```

În urma rulării secvenței de cod de mai sus, se va afișa valoarea `2`, deoarece ambele obiecte `p1` și `p2` vor fi inserate în HashSet-ul `lp`! Evident, problema se rezolvă implementând corect metodele `equals` și `hashCode` în clasa `Persoana`.

O problemă asemănătoare apare dacă se modifică valoarea unei date membre a unui obiect care este folosit pe post de cheie într-un HashMap (de exemplu, se modifică numele unei persoane), în cazul în care dacă valoarea datei membre respective este utilizată în implementările metodelor `hashCode` și `equals`. Din acest motiv, pentru chei se recomandă utilizarea unor obiecte care sunt instanțe ale unor clase imutabile!

Observații:

- Într-un HashSet se poate insera și valoare `null`, evident, o singură dată.
- O colecție de tip `HashSet` nu păstrează elementele în ordine inserării lor și nici nu pot efectua operații de sortare asupra sa.
- Implicit, *capacitatea* inițială (numărul de bucket-uri) a unei colecții de tip HashSet este 16, iar apoi aceasta este incrementată pe măsură ce se inserează elemente. Capacitatea inițială se poate stabili în momentul instanțierii sale, folosind constructorul `HashSet(int capacitate)`. În plus, pentru o astfel de colecție este definit un *factor de umplere* (load factor) care reprezintă pragul maxim permis de populare a colecției, depășirea sa conducând la dublarea capacității acesteia. Implicit, factorul de umplere este egal cu valoarea 0.75, ceea ce înseamnă că după ce se vor utiliza 75% din numărul de bucket-uri curente, numărul acestora va fi dublat. Astfel, considerând valorile implicite, prima dublare a numărului de bucket-uri va avea loc după ce se vor ocupa 0.75*16 = 12 bucket-uri, a doua dublare după ce se vor ocupa 0.75*32 = 24 de bucket-uri ș.a.m.d.

Clasa `LinkedHashSet`

Implementarea clasei `LinkedHashSet` este similară cu implementarea clasei `HashSet`, diferența constând în faptul că elementele vor fi stocate în ordinea inserării lor.

Exemplu: Pentru a găsi numerele distincte dintr-un fișier text, vom utiliza un obiect `nrdist` de tip `HashSet` în care vom insera, pe rând fiecare număr din fișier:

```
public class Test {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner in = new Scanner(new File("numere.txt"));
        HashSet<Integer> nrdist = new HashSet();

        while (in.hasNextLine()) {
            String linie = in.nextLine();
            String[] numere = linie.split("[ ,.;:?!]+");
            for (String nr : numere)
                nrdist.add(Integer.parseInt(nr));
        }

        System.out.println("Valorile distincte din fisier: ");
        for (int x : nrdist)
            System.out.print(x + " ");

        in.close();
    }
}
```

După executarea programului de mai sus, se vor afișa valorile distincte din fișierul text, într-o ordine oarecare. Dacă în locul clasei HashSet vom utiliza clasa LinkedHashSet, atunci valorile distincte vor fi afișate în ordinea inserării, adică în ordinea în care ele apar în fișierul text.

Clasa `TreeSet`

Intern, clasa `TreeSet` implementează o mulțime utilizând un arbore binar de tip Red-Black pentru a stoca elemente într-o anumită ordine, respectiv în ordinea lor naturală când se utilizează constructorul fără parametri ai clasei și clasa corespunzătoare obiectelor implementează interfața `Comparable` sau într-o ordine specificată în constructorul clasei printr-un argument de tip `Comparator`:

```
TreeSet t = new TreeSet();
TreeSet t = new TreeSet(Comparator comp);
```

Observații:

- Metodele `add`, `remove` și `contains` au o complexitate specifică structurii arborescente binare de tip Red-Black, respectiv $\mathcal{O}(\log_2 n)$.
- Colecția `TreeSet` este utilă în aplicații care necesită stocarea unui număr mare de obiecte sortate după un numit criteriu, regăsirea informației fiind rapidă.

Exemplu: Revenind la exemplu anterior, dacă dorim să valorile distincte în ordine descrescătoare, mai întâi definim comparatorul

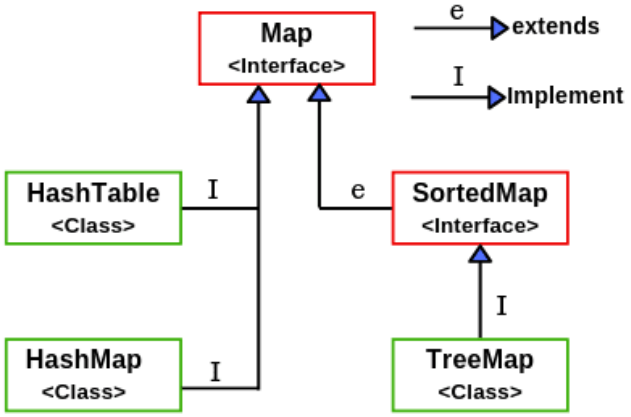
```
class cmpNumere implements Comparator<Integer> {
    @Override
    public int compare(Integer x, Integer y) {
        return y - x;
    }
}
```

și utilizăm constructorul corespunzător al clasei `TreeSet`:

```
TreeSet<Integer> nrdist = new TreeSet(new cmpNumere());
```

Interfața Map

Interfața Map, deși face parte din framework-ul Java Collections, nu extinde interfața Collection, ci este rădăcina unei ierarhii separate, așa cum se poate observa în figura de mai jos (sursa: https://www.scientecheasy.com/2018/09/collection-hierarchy-in-java-collections-class.html/?189db0)



Interfața `Map` modelează comportamentul colecțiilor ale căror elemente sunt de perechi de tipul *cheie* – *valoare* (definite în interfața `Map.Entry<T,R>`), prin care se asociază unei chei care trebuie să fie unice o singură valoare. Exemple: număr de telefon – persoană, CNP – persoană, cuvânt – frecvența sa într-un text, număr matricol – student etc.

Câteva metode uzuale din interfața `Map` sunt următoarele:

- `R put (T cheie, R valoare)` – inserează perechea `cheie-valoare` în colecție în cazul în care cheia nu există deja și returnează `null`, iar altfel înlocuiește vechea valoare asociată cheii cu noua valoare și returnează vechea valoare;
- `R get(T cheie)` – returnează valoarea asociată cheii indicate sau `null` dacă în colecție nu există cheia respectivă;
- `boolean containsKey(T cheie)` – returnează `true` dacă în colecție există cheia respectivă sau `false` în caz contrar;
- `boolean containsValue(R valoare)` – returnează `true` dacă în colecție există valoarea respectivă sau `false` în caz contrar;
- `Set<Map.Entry<K,V>> entrySet()` – returnează o mulțime care conține toate perechile cheie-valoare din colecție;
- `Set<K> keySet()` – returnează o mulțime care conține toate cheile din colecție;
- `Collection<V> values()` – returnează o colecție care conține toate valorile din colecția de tip `Map`;
- `R remove(Object cheie)` – dacă în colecție există cheia indicată, atunci elimină din colecție perechea având cheia respectivă și returnează valoarea cu care era asociată, altfel returnează `null`;
- `boolean remove(Object cheie, Object valoare)` – dacă în colecție există perechea `cheie-valoare` dată, atunci o elimină și returnează true, altfel returnează `false`;
- `void clear()` – elimină toate perechile existente în colecție.

Interfața `Map` este implementată în clasele `HashMap` și `TreeMap`, pe care le vom prezenta în continuare.

Clasa `HashMap`

Intern, implementarea clasei HashMap utilizează o tabelă de dispersie în care indexul bucket-ului în care va fi plasată o anumită valoare este dat de hash-code-ul corespunzător cheii (`cheie.hashCode()`), deci toate operațiile de căutare/inserare/ștergere se vor efectua în funcție de hash-code-ul cheii!

Complexitățile minime și medii ale metodelor `get`, `put`, `containsKey` și `remove` sunt $\mathcal{O}(1)$ în cazul implementării în metoda hashCode() a unei funcții de dispersie bune, care generează valori uniform distribuite, dar se poate ajunge la o complexitate maximă egală cu $\mathcal{O}(n)$, unde n reprezintă numărul de elemente din HashMap-ul respectiv, în cazul utilizării unei funcții de dispersie slabe, care produce multe coliziuni.

Observații:

- Într-un HashMap este permisă utilizarea valorii `null` atât pentru cheie, cât și pentru valoare.
- Într-un HashMap se poate asocia aceeași valoare mai multor chei.
- Într-un HashMap nu se menține ordinea de inserare și nici nu se poate stabili o anumită ordine a perechilor!
- Într-un HashMap se pot realiza și mapări complexe:

```
//h1 conține studenții anului I folosind perechi număr_matricol - student
HashMap<String, Student> h1 = new HashMap<>();
//h2 conține studenții anului II folosind perechi număr_matricol - student
HashMap<String, Student> h2 = new HashMap<>();
//m conține studenții din fiecare an folosind perechi an_studiu – studenti
HashMap<Integer, HashMap<String, Student>> m = new HashMap();

h1.put("11111", new Student("Ion Popescu", 141, new int[]{10, 9, 10, 7, 8}));
h1.put("22222", new Student("Anca Pop", 142, new int[]{9, 10, 10, 8}));
h2.put("12121", new Student("Ana Ionescu", 241, new int[]{8, 9, 10}));
h2.put("12345", new Student("Radu Mihai" , 242 , new int[]{9, 10, 8}));

m.put(1, h1);
m.put(2, h2);

for(Map.Entry<Integer, HashMap<String, Student>> hms : m.entrySet()) {
      System.out.println("An " + hms.getKey() + ": ");
      for(Map.Entry<String, Student> s : hms.getValue().entrySet())
            System.out.println(s);
   }
```

Exemplu: Pentru a calcula frecvența cuvintelor dintr-un fișier, vom folosi un HashMap cu perechi de forma *cuvânt – frecvență_cuvânt*. Fiecare cuvânt din fișier va fi căutat în HashMap și dacă nu există deja, va fi inserat cu frecvența 1, altfel i se va actualiza frecvența mărită cu 1 (prin reinserare):

```
public class Test {
   public static void main(String[] args) throws FileNotFoundException {
         Scanner in = new Scanner(new File("exemplu.txt"));
         HashMap<String, Integer> fcuv = new HashMap();

         while(in.hasNextLine()) {
               String linie = in.nextLine();
               String []cuvinte = linie.split("[ ,.;:?!]+");
               for(String cuvant : cuvinte)
                     if(fcuv.containsKey(cuvant))
                           fcuv.put(cuvant, fcuv.get(cuvant) + 1);
                     else
                           fcuv.put(cuvant, 1);
         }

         System.out.println("Frecventele cuvintelor din fisier: ");
         for(Map.Entry<String, Integer> aux : fcuv.entrySet())
               System.out.println(aux.getKey() + " -> " + aux.getValue());

         in.close();

   }
}
```

Clasa TreeMap

Intern, implementarea clasei TreeMap utilizează un arbore binar de tip Red-Black pentru a menține perechile *cheie-valoare* sortate fie în ordine naturală a cheilor, dacă se utilizează constructorul fără parametri, fie în ordinea indusă de un comparator transmis ca parametru al constructorului. Astfel, dacă în exemplul anterior înlocuim obiectul de tip HashMap cu un obiect de tip TreeMap și utilizăm tot constructorul fără argumente, cuvintele din fișier vor fi afișate în ordine alfabetică. Dacă dorim să afișăm cuvintele în ordinea crescătoare a lungimilor lor, iar în cazul unor cuvinte de lungimi egale în ordine alfabetică, definim comparatorul

```
class cmpCuvinte implements Comparator<String> {

    @Override
    public int compare(String s1, String s2) {
        if(s1.length() != s2.length())
            return s1.length() - s2.length();
        else
            return s1.compareTo(s2);
    }
}
```

și utilizăm constructorul corespunzător din clasa TreeMap:

```
TreeMap<String, Integer> fcuv = new TreeMap(new cmpCuvinte());
```

Observații:

- Perechile dintr-un TreeMap pot fi sortate doar folosind criterii care implică doar cheile, ci nu și valorile, deci un comparator care va fi transmis ca parametru constructorului clasei TreeMap trebuie să țină cont de această restricție! Pentru a sorta perechile folosind criterii care implică valorile, de obicei, se preferă extragerea tuturor perechilor într-o colecție care permite realizarea operației de sortare după diverse criterii într-un mod simplu (de exemplu, o listă sau un tablou unidimensional).
- Operațiile de inserare/căutare/ștergere într-un TreeMap se realizează tot pe baza hash-code-ului corespunzător cheii, dar utilizarea unui arbore Red-Black garantează o complexitate egală cu $\mathcal{O}(\log_2 n)$ pentru metodele `get`, `put`, `containsKey` și `remove`.

Interfața `Iterator`

Rolul general al unui iterator este acela de a parcurge elementele unei colecții de orice tip, mai puțin a celor care fac parte din ierarhia interfeței `Map`. Orice colecție `c` din ierarhia interfeței `Collection` conține o implementare a metodei `iterator()` care returnează un obiect de tip `Iterator<Tip>`:

```
Iterator itr = c.iterator();
```

În interfața `Iterator` sunt definite următoarele metode pentru accesarea elementelor unei colecții:

- `public Object next()` – returnează succesorul elementului curent;
- `public boolean hasNext()` – returnează `true` dacă în colecție mai există elemente nevizitate sau `false` în caz contrar.

Un iterator nu permite modificarea valorii elementului curent și nici adăugarea unor elemente noi în colecție!

Exemplu:

```
LinkedList<String> lista = new LinkedList<>();

lista.add("Ion");
lista.add("Vasile");
lista.addLast("Ana");
lista.addFirst("Radu");
lista.add(2, "Ioana");

Iterator itr = lista.iterator();
while(itr.hasNext())
   System.out.println(itr.next());
```

Orice colecție conține metode `remove` pentru ștergerea unui element având o anumită poziție și/sau o anumită valoare. Totuși, în cazul în care o colecție este parcursă fie "clasic", utilizând o instrucțiune de tip *enhanced-for*, fie cu un iterator, aceste metode nu pot fi utilizate, așa cum vom vedea în următoarele două exemple:

Exemplu 1: Ștergerea valorilor egale cu 1 dintr-o listă folosind o instrucțiune de tip *enhanced-for*

```
List<Integer> lista = new ArrayList<>();

lista.add(1);
............

for(Integer item:lista)
   if(item == 1)
         lista.remove(item);
```

Exemplu 2: Ștergerea numerelor pare dintr-o listă folosind un iterator

```
List<Integer> numere = new ArrayList<Integer>();

numere.add(101);
..............

Iterator<Integer> itr = numere.iterator();
while (itr.hasNext()) {
   Integer nr = itr.next();
   if (nr % 2 == 0)
         numere.remove(nr);
}
```

Deși apelul metodei `remove` este formal corect, în momentul executării secvențelor de cod de mai sus apare excepția `ConcurrentModificationException`, deoarece operația de ștergere se realizează în timpul iterării colecției!

De obicei, această excepție apare în aplicații multi-thread (aplicații cu mai multe fire de executare), unde nu este permis ca un fir de executare să modifice o colecție în timp ce un alt fir de executare parcurge colecția respectivă. Totuși, excepția apare și în aplicații cu un singur fir de executare, dacă se realizează parcurgea unei colecții cu un iterator de tip *fail fast iterator*, așa cum este cel utilizat în implementarea internă a instrucțiunii *enhanced for*!

O soluție sigură pentru a șterge un element dintr-o colecție presupune utilizarea metodei `void remove()` a unui iterator atașat unei colecții. Această metodă `default` este definită în interfața `Iterator` și permite ștergerea elementului curent (elementul referit de iterator):

```
Iterator<Integer> itr = numere.iterator();
while (itr.hasNext()) {
   Integer number = itr.next();
   if (number % 2 == 0)
         itr.remove();
}
```

În concluzie, ștergerea unui element dintr-o colecție se poate realiza folosind metodele `remove` definite în colecția respectivă, dacă aceasta nu este parcursă într-o manieră *fail fast iterator*, sau utilizând metoda `remove` din interfața `Iterator`, în caz contrar.