Curs 6

Liste inlantuite

Lista simplu înlănțuita este cea mai simpla structura de data înlănțuita. O astfel de lista este o secvența de obiecte alocate dinamic, fiecare obiect având referința către succesorul sau in lista. Exista o multitudine de moduri de a implementa aceasta structura de date. In figura 6.1 avem câteva reprezentări mai cunoscute a listelor simplu înlănțuie.

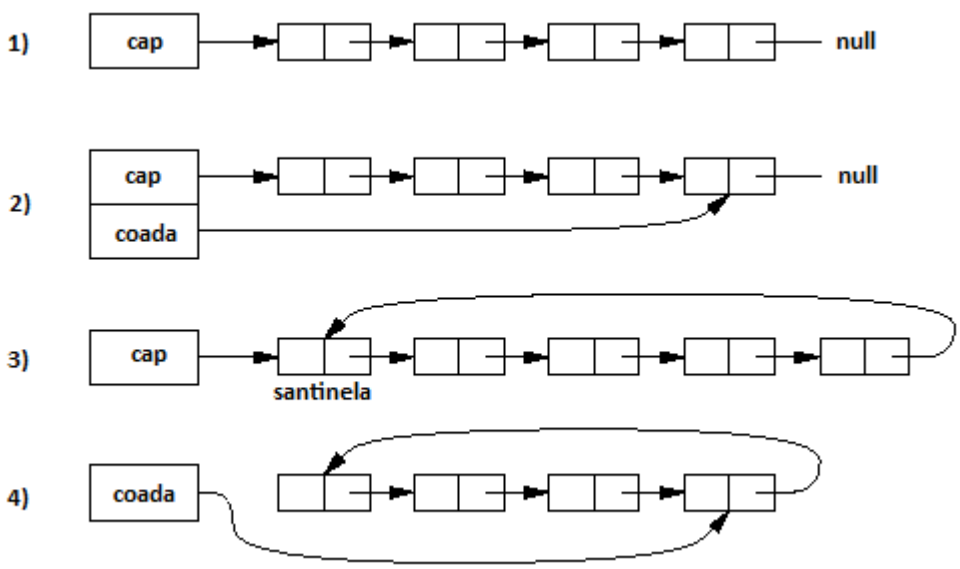

Figura 6.1 Tipuri de liste simplu inlantuite

Cea mai simpla lista dublu înlănțuia este in figura 6.1 1) in care fiecare element din lista indica spre succesorul lui, iar ultimul element către *null*. Variabila *cap* are rolul de a păstra primul element din lista. Acest tip de lista este ineficienta atunci când se dorește adăugarea elementelor la ambele capete ale listei. Pentru a adăuga un element la capătul listei, este necesar sa localizam ultimul element. Aceasta presupune parcurgerea întregii liste pentru o simpla operație de adăugare.

In figura 61. 2) avem reprezentarea unei liste având *cap* si *coada*, tocmai pentru a eficientiza adăugarea elementelor in lista. Singura problema ar fi de spațiu pentru variabila *coada*.

Lista simplu înlănțuită din figura 6.1 3) prezintă doua ajustări. Exista aici un element nou, santinela care nu este folosit pentru a conține date, dar este întotdeauna prezent. Avantajul principal in folosirea santinelei este ca simplifica anumite operații. De exemplu, din cauza acestui element, nu va trebui niciodată sa modificam variabila *cap*. De asemenea ultimul element va indica spre aceasta santinela, in loc sa indice către *null*; vorbim astfel despre o lista circulara. Avantajul este ca, inserarea la capul listei, inserarea la coada listei sunt operații identice.

Se poate realiza o lista simplu înlănțuită care nu are nevoie de o *santinela*. Figura 6.1 4) prezintă o variație in care, doar o variabila este folosita pentru a tine lista, de aceasta data *coada*, si anume ultimul element din lista.

Mai jos avem o reprezentarea acestor liste atunci când nu conțin nici un element.

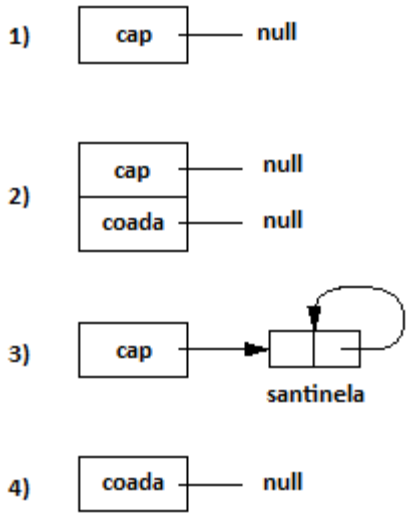

Figura 6.2 Listele goale

Reprezentarea listelor

Cum este reprezentat un element dintr-o lista simplu înlănțuita?

In cel mai simplu caz el consta intr-o clasa cu doi membrii: data si următorul element din lista. Mai jos avem exemplu de o astfel de lista:

```
class LinkedList
{
	int data;
	LinkedList next;
	public LinkedList()
	{
		next = null;
		data =0;
	}
}
```

Se poate observa ca aceasta clasa conține un membru *next* tot de același tip ca si clasa. Aceasta pentru ca, elementul din lista va indica spre următorul element din lista care este tot de același tip de data. Pe lângă acest membru, clasa mai conține un element de tip int *data*, care retine informații despre nodul curent din lista. Bineînțeles ca pe lângă acest membru, se pot declara o serie de alți membrii care pot retine diverse informație, in funcție de necesitați.

```
class LinkedListDemo
{
	public static void main(String args[])
	{
		LinkedList list = new LinkedList();
		list.data =23;
		list.next = new LinkedList();
		list.next.data = 10;
		list.next.next = new LinkedList();
		list.next.next.data = 49;
		//acum parcurgem lista de la capat
		//si anume din list
		LinkedList iterator = list;
		do
		{
			System.out.println(iterator.data);
		}
		while((iterator = iterator.next)!=null);

		iterator = list;
		do
		{
			System.out.println(iterator.data);
		}
		while((iterator = iterator.next)!=null);
	}
}
```

In clasa LinkedDemo avem un exemplu de folosire a listelor simplu înlănțuite. Se declara prima data un obiect de tip LinkedList cu un constructor fără parametrii. Apoi se instanțiază membrul *next*. In acest moment se numește ca am alipit primului element din lista un nou element si anume: `list.next`. Acest nou element este instanțiat la rândul lui tot cu un obiect de tip LinkedList: `list.next = new LinkedList();`. Din acest moment avem posibilitatea de a modifica datele acestui nou obiect cat si de a alipi un nou element acestuia, s.a.m.d. Vom vedea in continuare si cum se face adăugarea intr-un mod elegant.

Parcurgerea este un proces interesant. Desigur ca putem accesa elementele următoare prin sintaxe de genul `list.next.next.data`. Însa practica, nu ne permite acest mod de acces: daca avem o lista cu sute sau mii de elemente? De aceea o parcurgere presupune existenta unui iterator, sau element care sa se „transforme” in succesorul sau. Practic se trece de la element la element prin

instrucțiunea `iterator = iterator.next.` Aceasta se poate face intr-un *while*, pana când următorul element devine *null*, adică am ajuns la capătul listei.

In exemplul de mai sus, pentru a parcurge lista de doua ori, a trebuit sa mă folosesc de un element auxiliar. Daca as fi parcurs lista direct, folosind obiectul *list*, a doua oara nu as mai putea „reveni" de la începutul listei. De aceea este indicat sa ținem un element pe post de „capul" listei.

Mai jos avem un exemplu de lista simplu înlănțuită in care fiecare element tine o referință către capul listei. Acest fapt este avantajos mai ale in cazul in care ștergem un element, sau accesam un element in interiorul listei.

```
class LinkedList
{
	int data;
	LinkedList next;
	LinkedList head;
	public LinkedList()
	{
		head = this;
		next = null;
		data =0;
	}

	public LinkedList(LinkedList head)
	{
		this.head = head;
		next = null;
		data =0;
	}
}
```

De aceasta data avem doi constructori dintre care unul ce are ca parametru primul element din lista.Mai jos se poate vedea cum are loc aceasta instanțiere : `list.next.next = new LinkedList(list);`. Primul element din lista, adică *head*, este dat ca parametru al constructorului si anume *list*. In cazul in care folosim constructorul fără parametru, înseamnă ca instanțiem tocmai primul element din lista, de aceea avem instrucțiunea `head = this;` in acest constructor.

```
class LinkedListDemo
{
	public static void main(String args[])
	{
		LinkedList list = new LinkedList();
		list.head = list;
		list.data =23;
```

```
            list.next = new LinkedList(list);
            list.next.data = 10;
            list.next.next = new LinkedList(list);
            list.next.next.data = 49;
            //acum parcurgem lista de la capat
            //si anume din list
            while (list.next !=null)
            {
                  System.out.println(list.data);
                        list = list.next;
                }
                System.out.println(list.data);
            list = list.head;
            while (list.next !=null)
            {
                  System.out.println(list.data);
                        list = list.next;
                }
                System.out.println(list.data);
      }
}
```

In legătura cu utilizarea listelor de acest fel, putem remarca faptul ca lista se poate folosi si după o parcurgere `list = list.head;`, însa ce e mai interesant este ca se poate relua aceasta parcurgere din orice element al listei, din moment ce fiecare tine o referință către capul listei.

Sunt multe erori care pot apare, încă de la începutul definiției unei liste, astfel ca e bine sa încercam sa prevedem situațiile nefericite ce pot sa apară. De exemplu daca se încearcă accesarea unui element null, ca de exemplu primul element al listei, sa aruncam o excepție. Pentru aceasta membrii listei trebuie sa fie private sau protected (daca lista va fi moștenita), iar excepțiile vor apare in *get-eri* si *set-eri*.

Iată o parte a clasei LinkedList cu modificările de mai sus:

```
class LinkedList
{
      int data;
      protected LinkedList next;
      protected LinkedList head;
        public LinkedList GetNextElement()
        {
```

```
            //Aici nu se impune sa aruncam o exceptie
            //metoda apelanta va decide in functie de
            //faptul ca next este null sau nu
            return next;
        }
        public int GetNextElementData()
        {
            if (next==null)
                throw new NullPointerException("Elementul urmator este
null!");
            return next.data;
        }
```

Iar utilizarea acestor metode:

```
LinkedList list = new LinkedList();
int i = list.GetNextElementData();
```

Evident si apelul metodelor trebuie sa se afle intr-un bloc de try catch pentru a prinde excepțiile aruncate din aceste metode.

In exemplele următoare nu sunt implementate aceste „prevederi" legate de excepții, însă un bun exercițiu este sa modificați exemplele adăugând si tratările posibilelor situații ce pot sa apară. Excepțiile care pot sa apară nu sunt tratate, tocmai pentru a simplifica exemplele.

Operatiuni cu liste simplu înlăntuite

Pentru a înțelege mai bine toate operațiunile descrise in continuare vom reface structura elementelor de compun o lista si anume cele de clasa *Nod*:

```
public class Nod
{
    private Nod next;
    private int data;
    public Nod(int data, Nod next)
    {
        this.data = data;
        this.next = next;
    }
    public int getData ()
    {
        return data;
    }
    public Nod getNext ()
    {
        return next;
    }
```

```
public void setData (int newData)
{
    data = newData;
}
public void setLink (Nod newNext)
{
    next = newNext;
}
public void AddNodeAfter(int element)
{
    next = new Nod(element,next);
}
public void RemoveNodeAfter()
{
    next = next.next;
}
//iterare prin lista de la prima pozitie pana la
//pozitia idicata
public static Nod PositionInList (Nod start, int index) throws Exception
{
    int i;
    Nod iterator;
    if (!(index>0)) throw new Exception("Index incorect!");
    iterator = start;
    for(i=1;(i<index && iterator!=null);i++)
    {
        iterator = iterator.next;
    }
    return iterator;
}
public static Nod SearchInList(Nod start, int target)
{
    Nod iterator = start;
    for(;iterator!=null;iterator=iterator.next)
    {
        if (iterator.data == target)
            return iterator;
    }
    return null;
}
public String ListToString()
{
    Nod iterator = this;
    StringBuffer str=new StringBuffer();
    for(;iterator!=null;iterator=iterator.next)
    {
        str.append(iterator.data);
        str.append(" ");
```

```
        }
        return str.toString();
    }
     public String toString()
     {
         return data +"";
     }
}
```

Pe lângă cei doi membrii si anume *next* si *data*, se pot observa o serie de funcții pentru a înlesni adăugarea, căutarea, ștergerea elementelor dintr-o lista. Clasa *Nod* reprezintă un element din lista, însă prin intermediul acelui nod din lista putem de exemplu adăuga element imediat dupa capul listei:

```
public void AddNodeAfter(int element)
{
    next = new Nod(element,next);
}
```

Figura de mai jos explica ce se întâmpla când adăugam un nou element il lista:

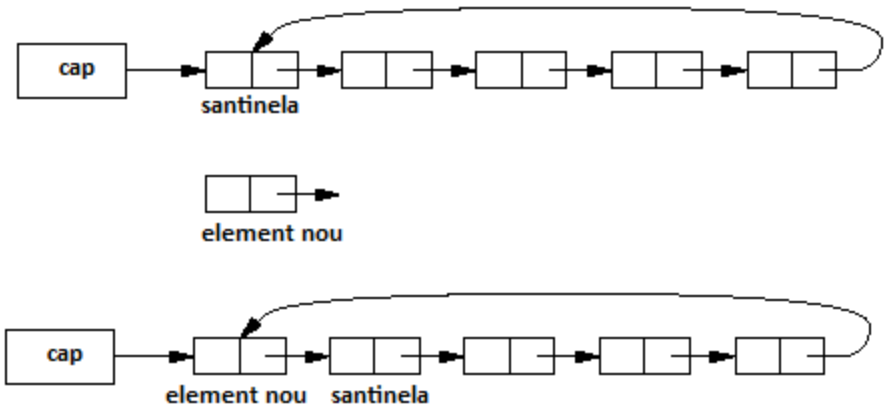

Figura 6.3 Adăugarea unui element in lista

Se poate remarca faptul ca noul element este introdus intre cap si vechea santinela, urmând ca noul element sa fie succesorul capului, ceea ce îl face pe acesta noua santinela.

Ștergerea va fi tot a acestei santinele, care înseamnă primul element dupa capul listei. Cand ștergem santinela, de fapt, sărim peste aceasta in felul următor: `next = next.next;` ceea ce înseamnă ca de acum capul listei va indica spre elementul ce urma santinelei. Acest lucru este reprezentat in figura 6.4.

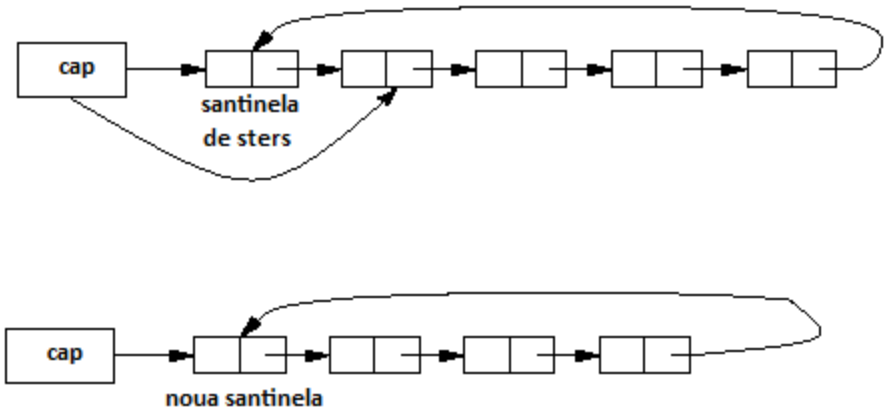

Figura 6.4 Ștergerea unui element din lista

Pe lângă metodele de adăugare, ștergere exista si metode de parcurgere in lista , `ListToString`, de returnarea a unui element de la o anumita poziție `PositionInList`, de căutare a unui element după valoare in lista: `SearchInList`. Toate aceste metode presupun existenta unui obiect numit iterator care primește de obicei primul element din lista `Nod iterator = start;`. Totuși lăsam posibilitatea de a specifica orice element din lista pentru ca afișarea, sau parcurgerea sa aibă loc de oriunde din cadrul listei, si nu neapărat de la primul element.

Parcurgerea presupune iterarea prin elementele listei adică trecerea de la un element la succesorul sau, iar aceasta se realizează cu *iterator=iterator.next*, instrucțiune pe care o găsim in bucla

```
iterator = iterator.next;
```

O serie de alte funcții sau de adăugări sau de ștergeri pot fi implementate, in aceasta clasa pentru ca operațiile sa fie cat mai ușor realizate. Iată un exemplu de inserare după un element din lista.

```
public void InsertAfterAnElement(int dataVal, Nod start, int newVal)
    {
        Nod tmp = new Nod(newVal,null);
        Nod prev = start;
        Nod iterator = start;
        do
        {
            if (iterator.data == dataVal)
            {
                prev = iterator.next;
                break;
            }
            iterator=iterator.next;
        }
        while(iterator!=null);
        tmp.next = prev;
        iterator.next = tmp;
    }
```

Aceasta metoda realizează inserarea după un element specificat ca prim parametru. Valoarea noului parametru este al treilea parametru. `tmp` reprezintă elementul nou, de aceea la creare nu specificam *next*-ul lui. Apoi parcurgem lista pana la elementul după care inseram, reținem următorul acelui element in `prev` si la sfârșit `tmp` va indica spre `prev` iar `tmp` devine succesorul elementului de dinaintea lui `prev`. Pentru o mai buna înțelegere a situației avem imaginea de mai jos figura 6.5.

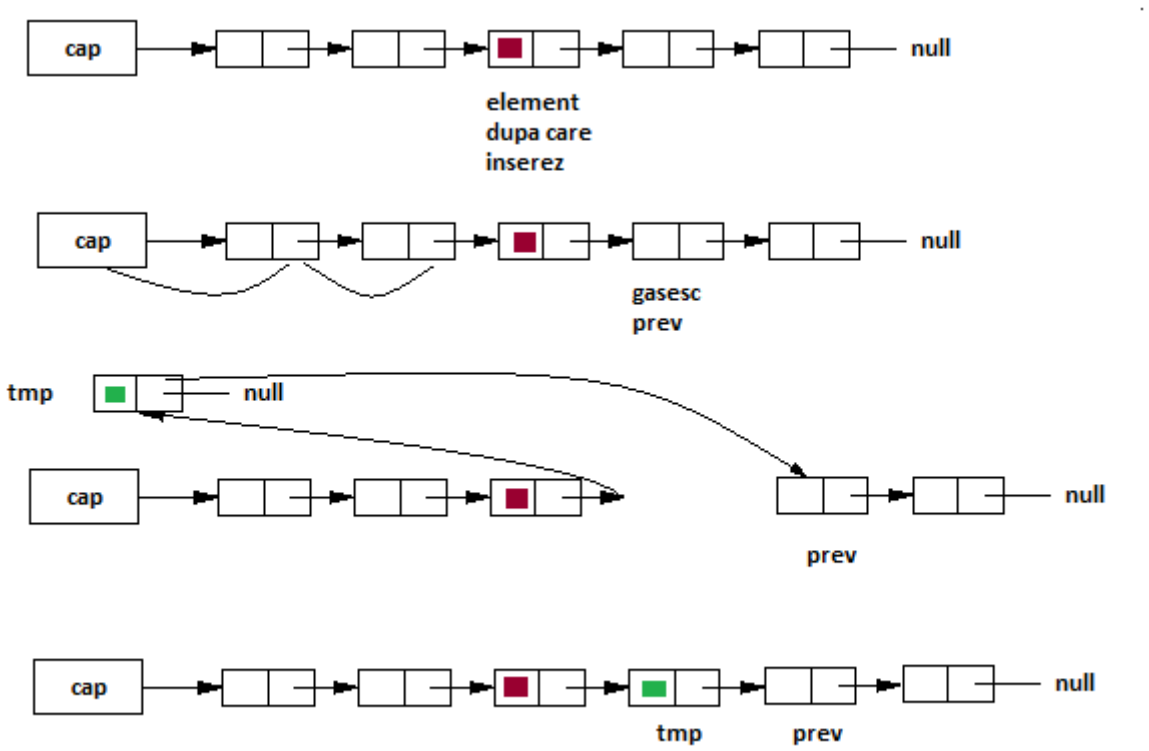

Figura 6.5 Inserarea dupa un element

Mai jos avem funcția de ștergere dupa un element:

```
public void DeleteAfterAnElement(int dataVal, Nod start)
    {
        Nod prev = start;
        Nod iterator = start;
        do
        {
            if (iterator.data == dataVal)
            {
                prev = iterator.next;
                break;
            }
            iterator=iterator.next;
        }
        while(iterator!=null);
        iterator.next = prev.next;
    }
```

Ca si in cazul inserării, se parcurge lista de la primul element pana la cel căutat, si se sare peste succesorul acestuia, marcând astfel ștergerea din lista: iterator.next = prev.next;
Pentru o mai buna înțelegere a acestei operații avem figura 6.6 mai jos:

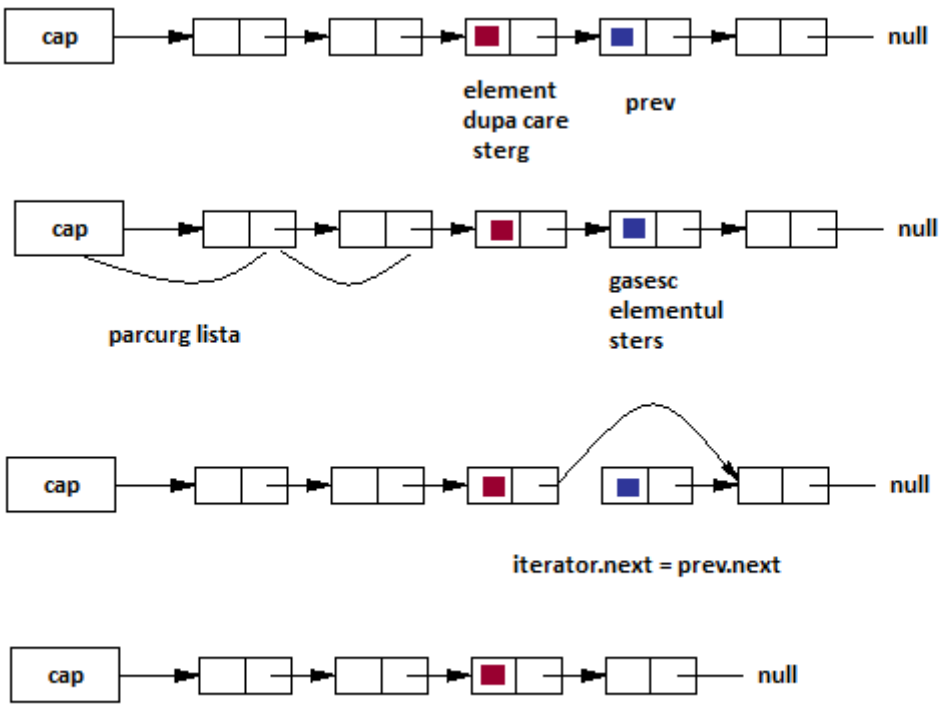

Figura 6.6 Ștergerea după un element din lista

In continuare vom vedea modul in care aceste structuri de date sunt implementate in Java, si mai ales Framework-ul pe care acestea sunt construite. Mai jos avem un exemplu de folosire a clasei de mai sus, cu toate metodele discutate:

```
public static void main(String[] args) throws Exception
    {
        Nod head = new Nod(3,null);
        head.AddNodeAfter(15);
        head.AddNodeAfter(12);
        head.AddNodeAfter(20);
         System.out.println(head.ListToString());
        head.InsertAfterAnElement(3,head,36);
         System.out.println(head.ListToString());
        head.DeleteAfterAnElement(36,head);
         System.out.println(head.ListToString());
        head.AddNodeAfter(40);
        head.AddNodeAfter(60);
        System.out.println(head.ListToString());
        System.out.println(Nod.PositionInList(head, 4));
```

```
        System.out.println(Nod.SearchInList(head, 40));
        head.RemoveNodeAfter();
        System.out.println(head.ListToString());

    }
```

Liste dublu inlantuite

In cazul in care dorim sa parcurgem ușor lista si in celalalt sens (si nu doar de la predecesor la succesor), avem lista dublu înlănțuită.

Aceasta păstrează conceptele listei simplu înlănțuite, cu specificarea ca fiecare element al listei mai conține o referință către predecesorul, sau anteriorul sau, așa cum apare in figura 6.7

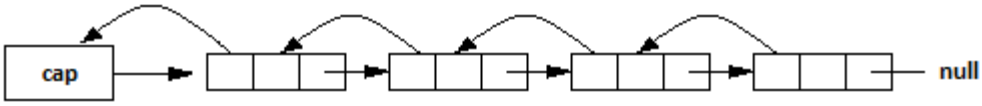

Figura 6.7 Reprezentarea unei liste dublu înlănțuite

Mai jos este o clasa ce reprezintă un element al acestei liste:

```
public class Nod
{
    private Nod next;
    private Nod prev;
    private int data;
    public Nod(int data, Nod next,Nod prev)
    {
        this.data = data;
        this.next = next;
        this.prev = prev;
    }
…
}
```

In continuare se păstrează aceleași metode, concepte ca in cazul listei simplu înlănțuite, dar ținând cont si de a doua legătura. Astfel in cazul adăugării, ștergerii, parcurgerii trebuie sa ținem cont de legătura *prev*.

FrameWork-ul Java pentru Colectii

Framework, in general înseamnă o serie de clase, librarii care joaca rol de „schelet” intr-o aplicație, permițând extinderea şi dezvoltarea ei pe baza acestor elemente. In Java , Framework-ul este asemănător cu Standard Template Library (STL) din C++. Exista aproximativ douazecisicinci de clase, interfețe, care constituie acest nucleu.

Bazele Framework-ului

Acest Framework de colecții costa din trei părți: interfețe, implementări si algoritmi. Implementările sunt acele clase concrete pe care Framework-ul le are, iar algoritmii sunt acțiuni, metode predefinite care pot exista in clase.

Interfetele

Exista mai multe interfețe in acest Framework si anume: Collection, List, Set, Map, SortedSet, si SortedMap.
Ierarhia acestora este prezentata in figura de mai jos:

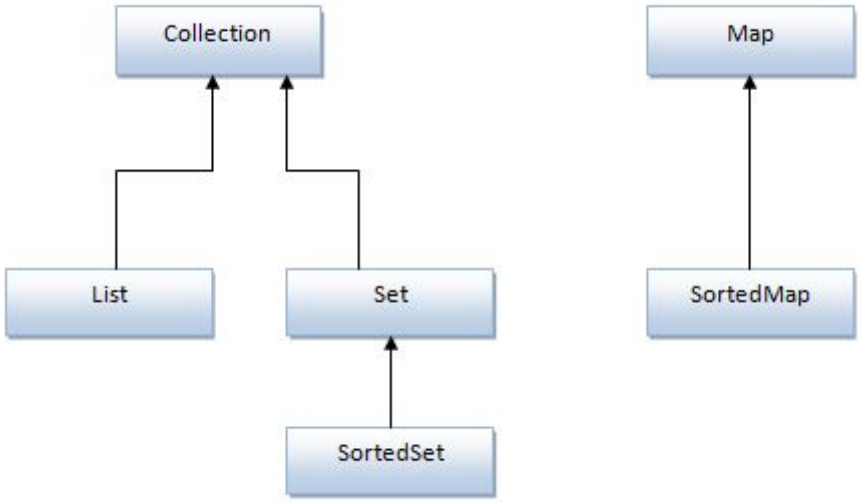

Figure 6.8 Interfetele din Collection

Se poate observa care este ierarhia de mosteniri in aceasta figură. *Collection* este un grup de date generic, in care fiecare membru este un *element*. Un tip *Collection* poate contine duplicate si poate sa nu aiba elementele sortate. Interfata *List* este o colectie specializata, in sensul ca, defineste o anumita ordinde pentru elementele sale. Pot exista duplicate, dar important este ca exista o ordine. Set-ul este acea colectie ce simuleaza o multime matematica. Daca avem nevoie de o multime sortata, avem la dispozitie SortedSet.
Cealalta ierarhie de clase este Maps si anume multimi de perechi *cheie-valoare*. Deoarece map-urile au un element compus din doua parti, au o alta implementare decat Collection. In caz ca lucram cu o colectie de chei sortate putem folosi *SortedMap*.

Clasele Framework-ului

Iata cum se poate folosi un obiect din acest framework:

```
List list = new …();
```

Mai jos este un tabel cu aceste clase concrete care moștenesc interfețele prezentate mai sus:

	Clase concrete				
Interfețe	**HashTable**	**Array mutabil**	**Arbore balansat**	**Lista înlănțuita**	**Altele**
Set	HashSet		TreeSet		
SortedSet			TreeSet		
List		ArrayList		LinkedList	Vector, Stack
Map	HashMap		TreeMap		Hashtable, Properties
SortedMap			TreeMap		

Toate clasele implementează ori interfața Collection, ori interfața Map. Pentru a înțelege acest tabel el va trebui citit de la stânga la dreapta si anume: pe al doilea rând: care sunt clasele ce implementează interfața *Set*? Pe fiecare coloana vom găsi acea clasa ce implementează interfața, daca este cazul.
Vom analiza pe rând, toate aceste clase si interfețe. Daca dorim sa creăm clase ce implementează aceste interfețe este bine sa ținem cont de câteva aspecte:

- Toate implementările sunt desincronizate. Se poate adăuga acces sincron, dar nu este necesar.
- Toate clasele oferă iteratori *fail-fast*. Daca o colecție este modificata in timp ce iteram prin elementele acesteia, va fi aruncata o excepție de tip ConcurrentModificationException.
- Toate implementările lucrează cu elemente null.
- Clasele se bazează pe un concept de *metode opționale* in interfete. Daca o clasa nu suporta o anumita operațiune, va arunca o excepție numita UnsupportedOperation Exception.

Algoritmii Framework-ului

Exista o serie de algoritmi predefinit care se găsesc in *Collections* si in *Arrays,* ce ajuta in lucrul cu aceste colecții si pe care ii vom analiza in curând.

Interfata Collection

Aceasta interfața consta din o serie de metode necesare in lucru cu o colecție si anume:

Metoda	Descrierea
add()	Adaugă un element intr-o colecție
addAll()	Adaugă o colecție in alta colecție
clear()	Șterge toate elementele dintr-o colecție
contains()	Verifica daca un element se afla intr-o colecție
containsAll()	Verifica daca o colecție se afla intr-o alta colecție
equals()	Verifica egalitatea a doua colecții
hashCode()	Returnează un *id* specific unei colecții
isEmpty()	Verifica daca o colecție este goala
iterator()	Returnează un obiect dintr-o colecție ce permite vizitarea elementelor acesteia
remove()	Șterge un element dintr-o colecție
removeAll()	Șterge elementele unei colecții din colecția curenta
retainAll()	Șterge elementele dintr-o colecție care nu sunt in alta colecție
size()	Returnează numărul de elemente dintr-o colecție
toArray()	Returnează elementele dintr-o colecție ca sir.

Adaugarea elmentelor

Putem adăuga doar un element folosind metoda *add*. In mod normal, daca nu se arunca nici o excepție, acest element va fi in colecție după return-ul din funcție. In acest caz valoare returnata este *true*.

```
public boolean add(Object element)
```

Totuși daca ceva neprevăzut se întâmplă si elementul nu poate fi adăugat in colecție se va returna valoarea *false*.

Putem de asemenea adăuga o colecție prin metoda *addAll*.

```
public boolean addAll(Collection c)
```

Fiecare element din colecția *c* va fi adăugat in colecția ce apelează metoda. Daca in urma apelului, colecția se modifica, metoda returnează *true*, altfel este returnat *false*. Daca apar duplicate, iar acest lucru nu este suportat de acel tip de colecție, va fi aruncata excepția de tip *UnsuporedOperationException*.

Stergerea elementelor

Putem şterge toate elementele dintr-o colecție folosind metoda *clear*:

```
public void clear()
```

In cazul in care colecția este read-only vom primi o excepție *UnsuporedOperationException.*

Se pot șterge anumite elemente dintr-o colecție:

```
public boolean remove(Object element)
```

Pentru a determina daca un element este in colecție sau nu, trebuie sa ne bazam pe metoda *equals*. Atunci cand se determina care este elementul din colecție ce va fi șters, metoda *equals* va fi invocata. Daca ștergerea nu este permisa, vom primi excepția de mai sus.

Se pot șterge si elementele unei colecții folosind funcția:

```
public boolean removeAll(Collection c)
```

Metoda șterge toate instanțele din colecția sursa, care se afla si in colecția *c*. Daca avem elementele:

[1,5,8,1,2,4,2,5,7,6]

si colecția *c* este:

[1,5]

colecția rămasă în urma ștergerii este:

[8,2,4,2,7,6]

Retinerea colectiei

Aceasta operațiune este inversa lui *removeAll* si anume presupune ca doar elementele din colecția *c* sunt păstrate in colecția originală:

```
public boolean retainAll(Collection c)
```

Practic metoda funcționează ca o intersecție de mulțimi. Daca aplicam aceasta operație pentru colecția:

[1,5,8,1,2,4,2,5,7,6]

si *c* este:

[1,5,23,29]

colecția ce rezulta este:

[1,5]

Alte Operatiuni

Returnarea elementelor

Exista o singura metoda pentru a returna elemente din colecție. Prin folosirea unui iterator, si anume prin apelul metodei iterator() putem vizita toate elementele unei colecții:

```
public Iterator iterator()
```

Vom reveni mai târziu asupra acestui subiect.

Cautarea elementelor

Înainte de a căuta un element este indicat sa verificam existenta lui in colecție:

```
public boolean contains (Object element)
```

Daca obiectul *element* este găsit, atunci funcția va returna *true*, altfel va returna *false*. Ca si in cazul *remove()*, se folosește metoda *equals* pentru comparare.

Se poate de asemenea verifica daca o colecție conține alta colecție prin funcția:

```
public boolean containsAll(Collection c)
```

Aceasta metoda va căuta subsetul *c* in colecția curenta. Daca doar un element din *c* nu este găsit in colecția curenta se returnează *false*.

Funcția pentru aflarea mărimii unei colecții este:

```
public int size()
```

Aceasta funcție returnează numărul de elemente din colecție.

Pentru a verifica daca o colecție este goala sau nu avem funcția:

```
public boolean isEmpty()
```

Metoda returnează *true* daca nu avem nici un element in colecție.

Clonarea colectiilor

Interfața Collection nu implementează nici *Cloneable* si nici *Serializable*. Pentru a copia o colecție, este indicat sa o transmitem la instanțierea obiectului ca parametru al constructorului.

Un alt mod de a copia o colecție, se realizează prin intermediul metodei *toArray()*

```
public Object[] toArray()
public Object[] toArray(Object[] a)
```

Prima metoda va returna un sir de obiecte si anume elementele din colecție. Deoarece metoda returnează un *Object[]*, de fiecare data când avem nevoie sa luam un element din sir va trebui sa il transformam către tipul de baza folosind operatorul de cast.

A doua versiune a metodei este folositoare când vrem sa determinam mărimea șirului de returnat pe baza șirului pasat ca parametru. Daca mărimea colecției este mai mica decât *a.length* atunci elementele sunt puse in sir si returnate. Daca șirul e prea mic, se creează un nou sir cu mărimea egala cu cea a colecției, si acel sir este returnat. Daca șirul dat ca parametru este prea mare atunci metoda înlocuiește cu null, elementele de după ultimul element copiat: `a[collection.size()]=null`.

Iată un mod de folosire a acestei funcții:

```
// creem un sir cu elemente din colectie:
Collection c = ...;
String array[] = new String[0];
array = (String[])c.toArray(array);

//Marimea sirului o vom stabili noi
Collection c = ...;
String array[] = new String[c.size()];
array = (String[])c.toArray(array);
```

Egalitatea este verificata prin funcția *equals*:

```
public boolean equals(Object o)
```

Aceasta metoda se poate suprascrie si vom vedea in curând si cum.

Interfata Iterator

Iteratorul este acel obiect ce permite parcurgerea colecțiilor. Pentru aceasta el trebuie sa implementeze interfața *Iterator*.

Interfața *Iterator* conține trei metode:

hasNext() – ce verifica daca mai sunt elemente de iterat

next() – returnează următorul element din lista

remove() – șterge un element din iterator

Cum se folosește un iterator?

Asemănător unui Enumeration, se va parcurge intr-o buclă ca mai jos:

```
Collection c = ...
Iterator i = c.iterator();
while (i.hasNext())
{
      process(i.next());
}
```

Se verifica daca mai avem element de iterat prin `hasNext()` si apoi se preia acest element cu `next()`, pentru a fi utilizat mai departe.

Metoda *remove()* este o noutate si nu are echivalent in *Enumeration*. Atunci când este apelata, va șterge din colecția sursa, in cazul in care aceasta operație este suportata. Atenție, daca se iterează in colecție cu metoda *next()* vom primi o excepție de tipul ConcurrentModificationException.

Iterator pentru filtrare

Pe lângă posibilitatea de a parcurge elementele din colecție, putem aplica un *predicat*, si anume interfața având o metoda ce filtrează elementele din colecție.

```
interface Predicate
{
      boolean predicate(Object element);
}
```

Atunci când apelam metoda *next()* a iteratorului, va fi returnat acel următor element din colecție care îndeplinește condiția data de metoda `predicate()`.

Mai jos avem un exemplu de folosire al acestui mecanism:

```
interface IPredicate
{
    public boolean predicate(Object o);
}
class Predicate implements IPredicate
{
    public boolean predicate(Object o)
    {
        return o.toString().startsWith("A");
    }
}
```

Avem aici interfața *Ipredicate* si clasa ce o implementează si anume *Predicate*.

In implementarea din clasa *Predicate*, metoda `predicate` va returna *true*, doar daca String-ul, si anume elementul curent, începe cu litera „A”. Pentru a vedea implementarea clasei particularizate *Iterator* si anume *IteratorWithPredicate*, avem exemplul de mai jos:

```
import java.util.*;
public class IteratorWithPredicate implements Iterator
{
    //elementul cu ajutorul caruia parcurgem
    //colectia care va folosi IteratorWithPredicate
    Iterator iter;

    //elementul care va filtra
    Predicate pred;

    //urmatorul obiect din colectie
    Object next;

    //variabila care ne avertizeaza ca am
    //parcurs intreaga colectie
```

```
    //adica daca mai am element de returnat
    boolean doneNext = false;
    public IteratorWithPredicate(Iterator iter, Predicate pred)
    {
        this.iter = iter;
        this.pred = pred;
    }
    public void remove()
    {
        //inca nu oferim cod valid pentru aceasta metoda
        throw new UnsupportedOperationException();
    }
    //implementam metoda hasNext
    public boolean hasNext()
    {
        doneNext = true;
        boolean hasNext;
        while (hasNext = iter.hasNext())
        {
            next = iter.next();
            if (pred.predicate(next))
            {
                break;
            }
        }
        return hasNext;
    }
    //si metoda next ale interfetei Iterator
    public Object next()
    {
        if (!doneNext)
        {
            boolean has = hasNext();
            if (!has)
            {
              throw new NoSuchElementException();
            }
        }
        doneNext = false;
        return next;
    }
}
```

Clasa `IteratorWithPredicate` implementează interfața definită in Java `Iterator`, deci va trebui sa suprascrie metodele `remove()`, `next()` si `hasNext()`. In cazul in care nu dorim sa implementam o metoda, cu un corp particularizat, însa suntem nevoiți sa o facem din condiții de polimorfism, vom arunca o eroare in acea funcție. Pentru a exemplifica cele spuse avem metoda `remove()`. In metoda `hasNext()` se preia următorul element din lista prin intermediul variabilei *iter*,

care este un iterator Java, si se aplica funcția `predicate` cu parametru obiectul obținut prin *iter*. Daca funcția returnează *true*, nu vom trece mai departe, rămânem pe elementul găsit si se returnează *false* in cazul in care mai avem elemente sau *true* daca am ajuns la finalul colecției. Daca însa `predicate` returnează *false*, mergem mai departe in colecție, pana când funcția va returna *true*, sau am ajuns la finalul colecției.

In continuare avem un mic exemplu de folosire a acestor clase, si cu precădere a iteratorului personalizat.

```
class PredTest
 {
    static Predicate pred = new Predicate();
    public static void main (String args[])
    {
        String[] str ={"Anca","Razvan","Maria","Daniela","Paul",
"Adrian","Dorin"};
        List list = Arrays.asList(str);
        Iterator i1 = list.iterator();
        Iterator i = new IteratorWithPredicate(i1, pred);
        while (i.hasNext())
        {
            System.out.println(i.next());
        }
    }
}
```

In acest mic exemplu, se transforma șirul *str*, in colecție, si anume *list*:

```
List list = Arrays.asList(str);
```

Apoi se construiește un iterator personalizat pe baza iteratorului lui *list*:

```
Iterator i = new IteratorWithPredicate(i1, pred);
```

Folosind acest iterator se parcurge lista returnându-se elementele pe baza filtrului din *Predicate*, si anume acele elemente ce încep cu „A”.

Exceptiile ce pot apare in colectii

Exceptii de modificare concurentiala

Exceptia *ConcurrentModificationException* apare datorita proprietății de *fail-fast* a iteratorilor. Daca o colecție este modificata in timp ce iteram prin elementele acesteia, apare acest conflict. Pentru a exemplifica avem clasa de mai jos:

```
import java.util.*;
public class CollectionException
```

```
{
    public static void main (String args[])
    {
        String[] str = {"1","a","et"};
        List list = new ArrayList(Arrays.asList(str));
        Iterator i = list.iterator();
        while (i.hasNext())
        {
            System.out.println(i.next());
            list.add("Element");
        }
    }
}
```

Problema este ca dorim sa adăugăm un element in colecția *list* in momentul parcurgerii acesteia: `list.add("Element");`, si atunci va apare excepția:

```
1
Exception in thread "main" java.util.ConcurrentModificationException
        at
java.util.AbstractList$Itr.checkForComodification(AbstractList.java:372)
        at java.util.AbstractList$Itr.next(AbstractList.java:343)
        at curs6.CollectionException.main(CollectionException.java:23)
```

Metodele de modificare a colecției sunt:

```
add()
addAll()
clear()
remove()
removeAll()
retainAll()
```

Exceptii pentru operatii neimplementate

Excepția *UnsupportedOperationException* este aruncata atunci când se apelează o metoda a unei colecții, însa aceasta metoda nu este implementata corespunzător. Cu alte cuvinte acea colecție nu are capabilitatea de a efectua operațiunea ceruta. Pentru a exemplifica, sa luam metoda *Arrays.asList(),* ce returnează o colecție de lungime fixa. Nu putem adăuga un nou element in acest tip de colecție:

```
List list = Arrays.asList(args);
list.add("Add"); // Arunca UnsupportedOperationException
```

Liste: List

List este in Java o interfață ce implementează Collection. Exista doua clase ce implementează aceasta interfața: ArrayList si LinkedList. Interfata *List* oferă posibilitatea de a lucra ordonat, deci permite păstrarea secvențiala a elementelor dintr-o colecție.

Metodele List

Pe lângă metodele implementate pentru ca *List* este o *Collection* mai avem:

Metoda	Descriere
indexOf()	Cauta un element in lista
lastIndexOf()	Cauta un element in lista începând de la sfârșitul acesteia
listIterator()	returnează iteratorul personalizat al listei
set()	modifica un element specificat din lista
get()	returnează un element din lista
remove()	șterge un anumit element din lista
subList()	returnează o parte din lista

Fiecare din clasele concrete vor implementa aceste metode. Pe lângă, vor avea si metode specifice in funcție de clasa.

ArrayList

Aceasta clasa este echivalentul clasei Vector, dar sub forma unei colecții. Un *ArrayList* este o colecție de elemente indexate intr-o anumită ordine, dar nu neapărat sortate.

Aceasta indexare permite accesul rapid la date, dar o insertie si stergere mai lenta.

Iată câteva dintre funcțiile oferite in plus de ArrayList:

Metoda	Descriere
ArrayList()	Constructorul pentru o lista goala
ensureCapacity()	Creează un buffer intern cu o capacitate dubla fata de cea anterioara
removeRange()	Șterge un anumit interval de elemente din lista
trimToSize()	limitează capacitatea buffer-ului intern la mărimea specificata

Crearea unui ArrayList

Se pot folosi doi constructori pentru acest lucru si anume:

```
public ArrayList()
public ArrayList (int initialCapacity)
```

Primul este pentru a instantia un obiect cu o lista goala. Al doilea constructor instanțiază o colecție de elemente null, cu dimensiunea specificata in parametru.

De exemplu:

```
String elements[] = {"Shopping List","Wine List","Food List"};
List list = new ArrayList(Arrays.asList(elements));
```

Adaugarea unor elemente

Se va face utilizând funcțiile de *add*:

```
public boolean add(Object element)
public boolean add(int index, Object element)
```

Prima funcție adăuga un obiect in lista la sfârșitul listei. A doua funcție, ce supraîncărca funcția *add,* permite adăugarea la indexul specificat si elementele de după, sunt împinse mai la dreapta cu o unitate. Indexarea pornește de la zero.

Pentru a exemplifica adăugarea, avem porțiunea de mai jos:

```
List list = new ArrayList();
list.add("3");
list.add("abs");
list.add("58");
// Adaug in interiorul listei
list.add(1, "un nou element");
```

De asemenea, ArrayList fiind o colecție, putem adăuga prin metodele de *addAll,* colecții:

```
public boolean addAll(Collection c)
public boolean addAll(int index, Collection c)
```

fiecare element din colecția, data ca argument, va fi pusa in lista, prin apelul metodei *add()*, in cazul primei metode *addAll().* In cazul in care folosim si un parametru *index*, inserarea in lista va avea loc de la indexul specificat. Daca lista, cu ajutorul căruia apelam aceste metode se modifica, atunci metoda va returna *true*. Daca adăugarea nu este suportata, va apare o excepție de tipul *UnsupportedOperationException.*

Returnarea unui element

Se realizează prin metoda *get*:

```
public Object get(int index)
```

şi are ca efect returnarea elementului de la poziția specificata.

Stergerea unui element

Se pot sterge toate elementele dintr-o lista:

```
public void clear()
```

De asemenea se pot șterge elemente specifice din lista:

```
public boolean remove(Object element)
public Object remove(int index)
```

A doua metoda șterge un element de la o anumita poziție, daca poziția este valida. Prima metoda șterge un element din lista comparând elementele listei cu parametrul. Pentru a verifica egalitatea se folosește metoda *equals()*. In acest caz se șterge primul element din lista egal cu parametrul.

Se poate șterge si o colecție prin metoda

```
public boolean removeAll(Collection c)
```

Aceasta metoda va șterge toate instanțele obiectelor din lista găsita in colecția *c*. De exemplu, daca lista originala era:

```
{”element”,”index”,”element”,”sir”,”clasa”}
```

Si colectia data este:

```
{”lista”,”multime”,”element”}
```

atunci lista rezultanta este:

```
{”index”,”sir”,”clasa”}
```

Retinerea unei colectii

Funcția *retainAll* este prezentata si mai sus, când am vorbit despre colecții:

```
public boolean retainAll(Collection c)
```

Prin aceasta funcție se rețin in colecție doar acele elemente comune listei si colecției *c*.

Stergerea unui interval

Funcția removeRange() este o metoda implementata doar de ArrayList:

```
protected void removeRange(int fromIndex, int toIndex)
```

Evident, nu poate fi folosita decat in clase ce extind ArrayList si are ca efect ștergerea elementelor cuprinse intre `fromIndex` si `toIndex`.

Operatii cu Liste

Returnarea elementelor din lista

Aceasta se va realiza prin intermediul iteratorului sau a metodei listIterator(), specifica doar listelor:

```
public Iterator iterator()
public ListIterator listIterator()
public ListIterator listIterator(int index)
```

Al treilea obiect, are ca parametru un index, pentru ca sa putem începe căutarea elementelor de la o anumita poziție. De exemplu:

```
List list = Arrays.asList(new String[] {"323", "re", "12","eo"});
Iterator iter = list.iterator();
while (iter.hasNext())
{
      System.out.println(iter.next());
}
```

Gasirea elementelor

Înainte de a returna un element in lista putem apela metoda *contains,* pentru a verifica daca un element se alfa in lista sau nu:

```
public boolean contains(Object element)
```

Putem apoi căuta un element si returna indexul pe care acesta se găsește:

```
public int indexOf(Object element)
public int lastIndexOf(Object element)
```

Prima metoda iterează începând cu primul element si in momentul in care a găsit elementul egal cu parametrul returnează poziția acestuia in lista. Compararea se face pe baza metodei equals.

A doua metoda are același comportament ca si prima, cu mențiunea ca, căutarea începe cu ultimul element.

De asemenea se poate folosi si funcția *containsAll* in condițiile mai sus menționate.

Modificarea unui element

Aceasta operațiune se realizează cu metoda *set()*:

```
public Object set(int index, Object element)
```

Se poate astfel modifica valoarea unui element de la o poziție specificata prin *index*.

Mărimea unei liste

Folosind metoda *size()* se poate afla cate elemente sunt intr-o lista. Pe deasupra, prin metoda *ensureCapacity()*, se poate redimensiona buffer-ul intern care conține lista.

Pentru a micșora numărul de elemente din lista avem funcția *trimToSize*.

Pentru a concluziona cele prezentate mai sus avem un mic exemplu de folosirea acestor tipuri de liste:

```
public class ArrayLists
{
    public static void main(String args[])
    {
        ArrayList list = new ArrayList();
        list.add("prim");
        list.add("2");
        list.add("al treilea");
        list.add(1,"intre 1 si 2");
        if (list.contains("al treilea"))
            list.remove("al treilea");
        System.out.println(list);
        System.out.println(list.indexOf("2"));
    }
}
```

Egalitate in liste

De mai multe ori am afirmat ca egalitatea se verifica prin metoda *equals*. Aceasta poate fi suprascrisa, in cazul in care lucram cu o lista particularizata după cum vom vedea in continuare.

Pe lângă metoda *equals*, atunci când se compara doua elemente, se compara si id-urile lor adică ceea ce returnează funcția *hashCode()*. Am precizat in cursul anterior ca aceasta funcție returnează un int ce reprezintă codul unic al obiectului, si aceasta funcție respecta mai multe conditii.

Mai jos avem declarația acesteia

```
public int hascode()
```

Pentru a înțelege cum se folosesc aceste funcții avem exemplul de mai jos:

```
import java.util.ArrayList;
class Point
{
      public int x;
      public int y;

      public Point (int x, int y)
      {
            this.x = x;
            this.y = y;
      }
    public boolean equals(Object o)
    {
        if (!(o instanceof Point))
            return false;
        Point pt = (Point)o;
        return ((pt.x == this.x) &&
                (pt.y==this.y));
    }
    public int hashCode()
    {
        return 17*this.x +23*this.y+43;
    }
    public String toString()
    {
        return "x = " +x+ " y = " +y + " id = "+ hashCode()+ "\n";
    }
}
public class Mylist
{
    public static void main(String args[])
    {
        ArrayList list = new ArrayList();
        list.add(new Point(1,2));
        list.add(new Point(3,4));
        list.add(new Point(2,1));
        list.add(new String("un punct"));
        Point pt = new Point(-1,-1);
        System.out.println(list.contains(pt));
        System.out.println(list.contains("un punct"));
        System.out.println(list.contains(new Point(3,4)));
        System.out.println(list);

    }
}
```

Întotdeauna funcția *hashcode()* va trebui sa returneze un număr diferit daca obiectele sunt diferite. Spre exemplu :

```
list.add(new Point(1,2));
list.add(new Point(2,1));
```

Avem doua obiecte diferite. Daca in funcția *hashCode()* s-ar fi returnat simplu suma celor doua coordonate, obiectele ar fi fost egale. De aceea se încearcă o formula care sa evite aceasta situație, si anume prin înmulțire cu numere prime si apoi însumare. In urma rulării acestui program, acesta este rezultatul:

```
false
true
true
[x = 1 y = 2 id = 106
, x = 3 y = 4 id = 186
, x = 2 y = 1 id = 100
, un punct]
```

Evident obiectul de tip String „un punct”, nu face parte din clasa Point, deci nu va respecta acele metode equals si *hashCode()*. Pentru a forța ca intr-o lista sa fie acceptate doar elemente de un anumit tip de data, se folosesc liste generice despre care vom vorbi imediat.

In continuare vom studia cealaltă clasa ce implementează ArrayList si anume LinkedList

LinkedList

Aceasta este implementarea listei înlănțuite descrise la începutul acestui curs si anume lista dublu înlănțuita. Vom discuta in continuare doar de metodele specifice LinkedList, celelalte metode fiind implementate din interfața Collection.

Crearea unei LinkedList

Constructorii pentru crearea unei liste simplu înlănțuite sunt:

```
public LinkedList()
public LinkedList(Collection col)
```

Adaugarea in LinkedList

Pentru a adăuga un element intr-o lista exista:

```
public boolean add(Object element)
public boolean add(int index, Object element)
```

Într-adevăr, se pot adăuga elemente la un anumit index, pentru ca si intr-o lista inlantuita elementele sunt ordonate. De asemenea o lista simplu înlănțuita poate funcționa ca o stiva sau ca o coada, ca atare implementează metodele:

```
public boolean addFirst(Object element)
public boolean addLast(Object element)
```

Cele doua metode adăugare permit plasarea elementului, fie la sfârșitul listei fie la capul ei. Pentru a obține elementele de la capete avem:

```
public Object getFirst()
public Object getLast()
```

Stergerea elementelor

Se pot sterge elementele de la capete folosind metodele:

```
public Object removeFirst()
public Object removeLast()
```

In cazul in care operațiunea de ștergere nu este suportata, va apare excepția UnsupportedOperationException.

Iteratorul LinkedList

Iteratorul se numește ListIterator si extinde Iterator. Deoarece listele dublu înlănțuite conțin elemente cu referința către *next* si *prev*, iteratorul va permite deplasarea in ambele direcții.

Iata mai jos metodele implementate de acest iterator:

Metoda	**Descriere**
hasNext	verifica pe direcția înainte, daca mai sunt elemente
hasPrevious	verifica pe direcția înapoi, daca mai sunt elemente
next	returnează următorul elemente
nextIndex	returnează indexul din colecție al următorului element
previousIndex	returnează indexul din colecție al elementului anterior
remove	șterge un element din iterator
set	modifica elementul curent

Pentru a înțelege mai bine lucrul cu aceste liste avem următorul exemplu:

```
public class LinkedLists {

    public static void main(String args[])
    {
        LinkedList list = new LinkedList();
        list.add(new Integer(2));
        list.addFirst(new Double(5.6));
        list.addFirst(new Double(5));
        list.addFirst(new Float(3.4));
        list.addLast(new Short((short)10));
        ListIterator it =  list.listIterator();
        //parcurgere de la cap la coada
        while (it.hasNext())
            System.out.print(" " + it.next());
        //sterg din iterator ultimul element
        //care a fost returnat
        System.out.println();
        it.remove();
        //parcurgere de la coada la cap
        while (it.hasPrevious())
        {
            System.out.print(" "+ it.previous());
            System.out.print(" " +it.previousIndex());
        }
        System.out.println();
        System.out.println(list);
    }
}
```

Rezultatul este următorul:

```
3.4 5.0 5.6 2 10
 2 2 5.6 1 5.0 0 3.4 -1
[3.4, 5.0, 5.6, 2]
```

Multimi: Set

Interfața *Set* reprezintă un grup de elemente fără duplicate. Nu exista o condiție anume ce impune acest lucru: adică sa nu fie duplicate elementele unui Set, ci implementările din clasele *Set,* sunt cele care impun aceasta condiție.

Interfața *Set* deriva din *Collections*, deci va implementa aceleași metode ca si aceasta interfața, cu specificarea ca elementele trebuie sa fie unice. De asemenea un element, aflat deja in mulțime, nu poate fi modificat. Aceasta interfața va fi implementata de doua clase: TreeSet si HashSet.

HashSet

Înainte de a prezenta aceasta clasa, vom spune ca HashSet este implementată ca un HashMap, sau un hashtable. Vom insista asupra acestor colecții in cursul următor. In continuare, vom studia operațiunile principale cu acest tip de data.

Crearea unui HashSet

```
public HashSet()
public HashSet(int initialCapacity)
public HashSet(int initialCapacity, int loadFactor)
```

Primii doi constructori sunt asemănători celor studiați. In cazul celui de-al treilea constructor, se poate specifica un factor de mărire a capacității, in cazul in care dorim acest lucru. De exemplu, daca nu dorim sa mărim cu 100% colecția, atunci când mărim capacitatea, putem specifica 75%, 50% etc.

Iată mai jos un exemplu de instanțiere a unui HashSet:

```
String elements[] = {"Englez", "German", "Roman","Italian"};
Set set = new HashSet(Arrays.asList(elements));
```

Adaugarea elementelor

Se pot adăuga elemente unul cate unul prin metoda:

```
public boolean add(Object element)
```

Metoda are ca efect adăugarea elementului, daca acesta nu este in *Set*, in caz contrar nu se adăuga elementul si metoda returnează *false*. Comparația intre doua elemente se realizează cu ajutorul metodei *equals*. Se poate adăuga si o colecție de elemente cu:

```
public boolean addAll(Collection c)
```

Același regim se va aplica si aici, adică doar elemente unice din colecție, care nu sunt in mulțime, vor fi adăugate.

Stergerea elementelor

Pentru a șterge un anumit element din mulțime exista metoda:

```
public boolean remove(Object element)
```

La fel, se pot șterge mai multe elemente, prin:

```
public boolean removeAll(Collection c)
```

Efectul acestei funcții este de a elimina din mulțime doar o singura data, elementele găsite in colecția *c*. de exemplu daca avem mulțimea:

```
{Englez", "German", "Roman","Italian"}
```

si ștergem din Set colecția:

```
{Englez", "German", "Englez","Englez"}
```

Va rămâne in mulțimea originala:

```
{"Roman","Italian"}
```

Pentru a retine anumite elemente exista funcția

```
public boolean retainAll(Collection c)
```

ce are efectul invers funcției `removeAll`.

Celelalte operații din mulțime sunt perfect asemănătoare cu cele din ArrayList, cu mențiunea ca se va respecta condiția ca elementele sa fie unice.

Insistam asupra metodei hashCode si equals:

```
public boolean equals(Object o)
public int hashCode()
```

Aceste doua metode sunt cele prin care se verifica daca un element este deja in mulțime sau nu. Pentru o buna funcționare a verificării daca un obiect este in mulțime sau nu, va trebui sa suprascriem corect ambele metode.

Folosind exact clasa Point, din exemplul de mai sus, iată cum se pot folosi mulțimile:

```
public class Multimi
{
    public static void main(String args[])
    {
        // Create the set
        Set set = new HashSet();

        // Adaug in multime
        set.add(new Point(1,2));
        set.add(new Point(2,1));
        set.add("c");
        // Sterg un element din multime
        set.remove("c");
        //Marimea unei multimi
        int size = set.size();
        System.out.println(size);
        // Adaug un element ce exista deja
         set.add(new Point(1,2));
         //fara a avea insa efect
        size = set.size();
        System.out.println(size);
        //Verificam daca un element este deja
        //in multime
        boolean b = set.contains(new Point(2,1));  // true
        System.out.println(b);
        b = set.contains("c");          // false
        System.out.println(b);
        // Parcurgem multimea
        Iterator it = set.iterator();
        while (it.hasNext())
        {
            //si afisam elementele
            Object element = it.next();
            System.out.println(element);
        }
    }
}
```

Observam ca , se pot introduce in mulțime, ca si in ArrayList obiecte de diverse tipuri. Uneori se dorește lucrul cu anumit tip de obiecte. In acel caz vom folosi liste generice.

Liste Generice

Se declara folosind „<” si „>” pentru a specifica tipul de data ce va fi admis in aceste colecții.

In rest comportamentul este exact același. Reluam un exemplu de la ArrayList folosind liste generice.

```
List<Point> list = new ArrayList();
  list.add(new Point(1,2));
  list.add(new Point(3,4));
  list.add(new Point(2,1));
  list.add(new String("un punct"));
  Point pt = new Point(-1,-1);
…
```

Eroarea de sintaxa, va apare atunci când încercăm sa adăugăm un element de tip *String*, altul decât cel admis in listă. Listele generice admit doar referințe, si nu admit tipuri primitive.