15. HEAP

15.1 Structura de tip *Heap*

Heap-ul, figura 15.1, este un arbore binar care respectă proprietăţile de structură şi de ordonare. Fiecare nod al arborelui trebuie să conţină o valoare asociată numită cheie şi poate conţine alte informaţii suplimentare. Cheia trebuie să permită definirea unei relaţii de ordine totală pe mulţimea nodurilor. În funcţie de obiectivul urmărit, heap-ul poate fi organizat sub formă de *max-heap* sau *min-heap*. Cele două tipuri de heap sunt echivalente. Transformarea unui max-heap în min-heap, sau invers, se poate realiza prin simpla inversare a relaţiei de ordine.

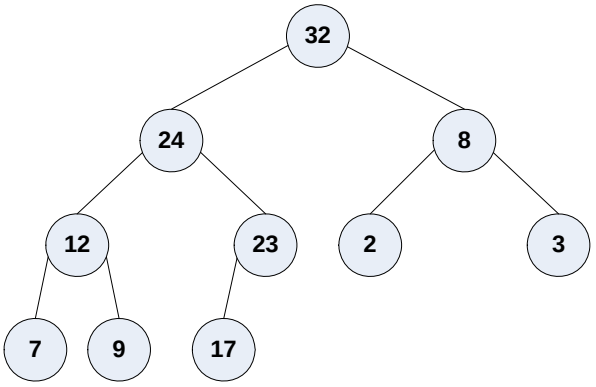

Figura 15.1 *Reprezentarea grafică a structurii heap*

Proprietatea de structură specifică faptul că elementele sunt organizate sub forma unui arbore binar complet. Prin arbore binar complet înţelegem un arbore binar în care toate nodurile, cu excepţia celor de pe ultimul nivel, au exact doi fii, iar nodurile de pe ultimul nivel sunt completate de la stânga la dreapta.

Proprietatea de ordonare impune ca valoarea asociată fiecărui nod, cu excepţia nodului rădăcină, să fie mai mică sau egală decât valoarea asociată nodului părinte. Se observă că, spre deosebire de arborii binari de căutare, nu se impune nici o regulă referitoare la poziţia sau relaţia dintre nodurile fiu.

Structura heap este preferată pentru multe tipuri de aplicaţii. Cele mai importante utilizări sunt:

- implementarea cozilor de prioritate utilizate pentru simularea pe bază de evenimente sau algoritmi de alocare a resurselor;
- implementarea selecţiei în algoritmi de tip greedy cum ar fi algoritmul lui Prim pentru determinarea arborelui de acoperire minimă sau algoritmul lui Dijkstra pentru determinarea drumului minim;
- sortarea masivelor utilizând algoritmul HeapSort.

Operaţiile principale care se execută pe o structură heap sunt cele de construire a heap-ului pornind de la un masiv unidimensional oarecare, inserarea unui element în structură şi extragerea elementului maxim sau minim pentru un min-heap.

Construirea structurii se face utilizând o procedură ajutătoare numită procedură de filtrare. Rolul acesteia este de a transforma un arbore în care doar subarborii rădăcinii sunt heap-uri ale căror înălţime diferă cu cel mult o unitate într-un heap prin coborârea valorii din rădăcină pe poziţia corectă. Structura rezultată în urma aplicării procedurii de filtrare este un heap (respectă proprietăţile de structură şi ordonare).

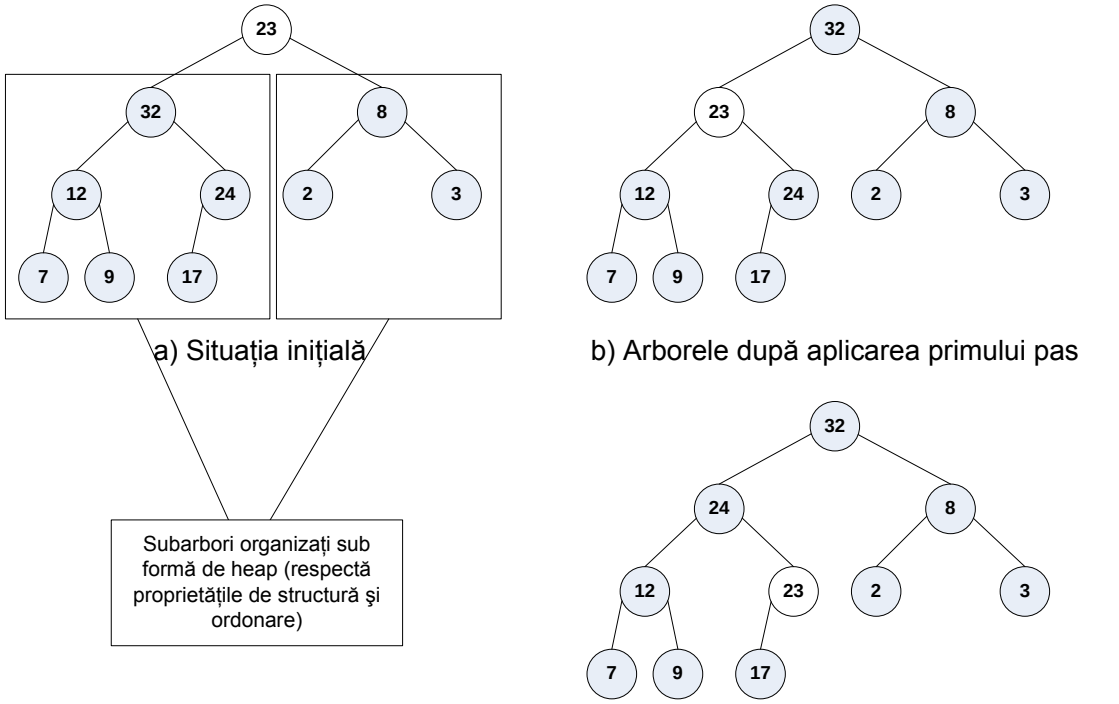

Figura 15.2 *Aplicarea algoritmului de filtrare*

Algoritmul de filtrare, figura 15.2, presupune parcurgerea următoarelor etape începând cu nodul rădăcină:

- se determină maximul dintre nodul curent, fiul stânga şi fiul dreapta (dacă există).
- dacă maximul se află în nodul curent, atunci algoritmul se opreşte.
- dacă maximul de află într-unul dintre fii, atunci se interschimbă valoarea din nodul curent cu cea din fiu şi se continuă execuţia algoritmului cu nodul fiu.

Construirea heap-ului pornind de la un arbore binar care respectă doar proprietatea de structură se face aplicând procedura de filtrare pe nodurile non frunză ale arborelui începând cu nodurile de la baza arborelui şi continuând în sus până când se ajunge la nodul rădăcină. Corectitudinea algoritmului este garantată de faptul că la fiecare pas subarborii nodului curent sunt heap-uri (deoarece sunt noduri frunză sau sunt noduri pe care a fost aplicată procedura de filtrare).

Inserarea elementelor într-un heap se poate face şi după etapa iniţială de construcţie. În acest caz adăugarea elementului nou trebuie făcută astfel încât structura rezultată să păstreze proprietatea de ordonare. Inserarea unui element, figura 15.3, în heap presupune parcurgerea următoarelor etape:

- se adaugă elementul ca nod frunză la sfârşitul arborelui pentru a păstra proprietatea de structură;
- se compară nodul curent cu nodul părinte;
- dacă nodul părinte este mai mic se interschimbă nodul curent cu nodul părinte;
- dacă nodul părinte este mai mare sau egal atunci algoritmul se opreşte.

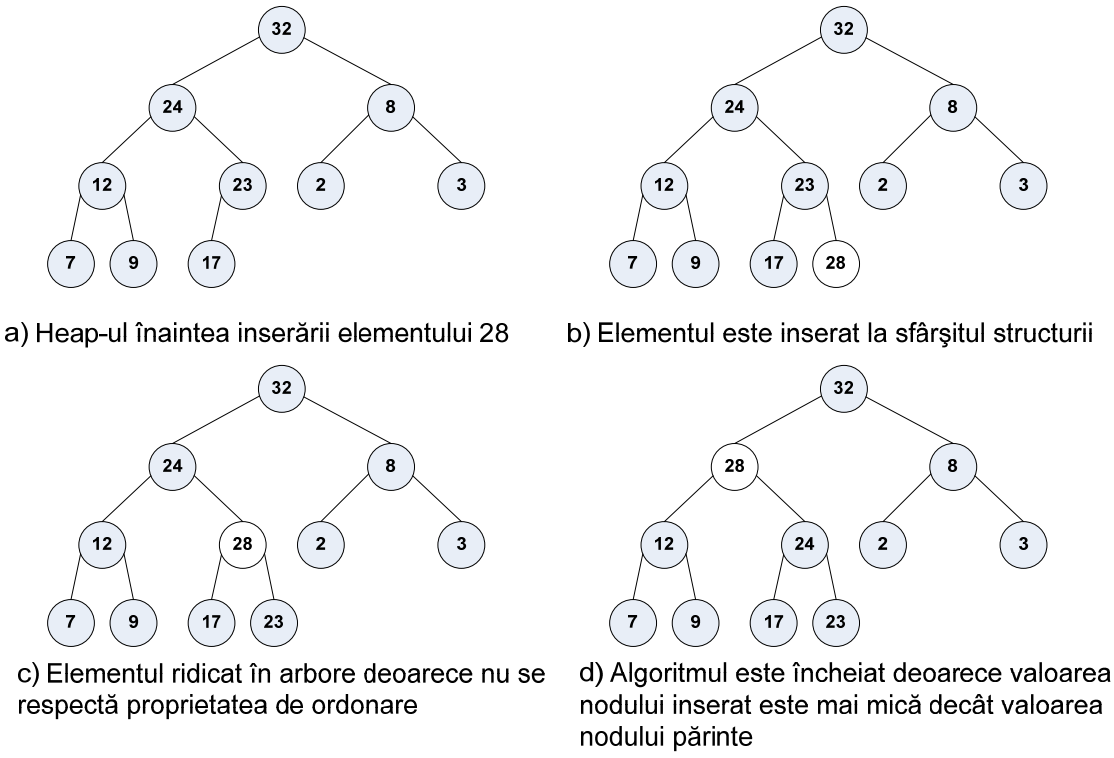

Figura 15.3 *Aplicarea algoritmului de inserare*

Procedura prezentată permite inserarea rapidă a oricărei valori în cadrul heap-ului cu păstrarea proprietăţilor de structură şi de ordonare.

Ştergerea elementelor dintr-un heap se poate efectua doar prin extragerea elementului maxim sau minim în cazul unui min-heap. Pentru păstrarea structurii de heap se utilizează procedura de filtrare prezentată anterior. Algoritmul de extragere al elementului maxim, figura 15.4, presupune parcurgerea următoarelor etape:

- se interschimbă valoarea din nodul rădăcină cu valoarea din ultimul nod al arborelui;
- se elimină ultimul nod din arbore;
- se aplica procedura de filtrare pe nodul rădăcină pentru a păstra proprietatea de ordonare;
- se returnează valoarea din nodul eliminat.

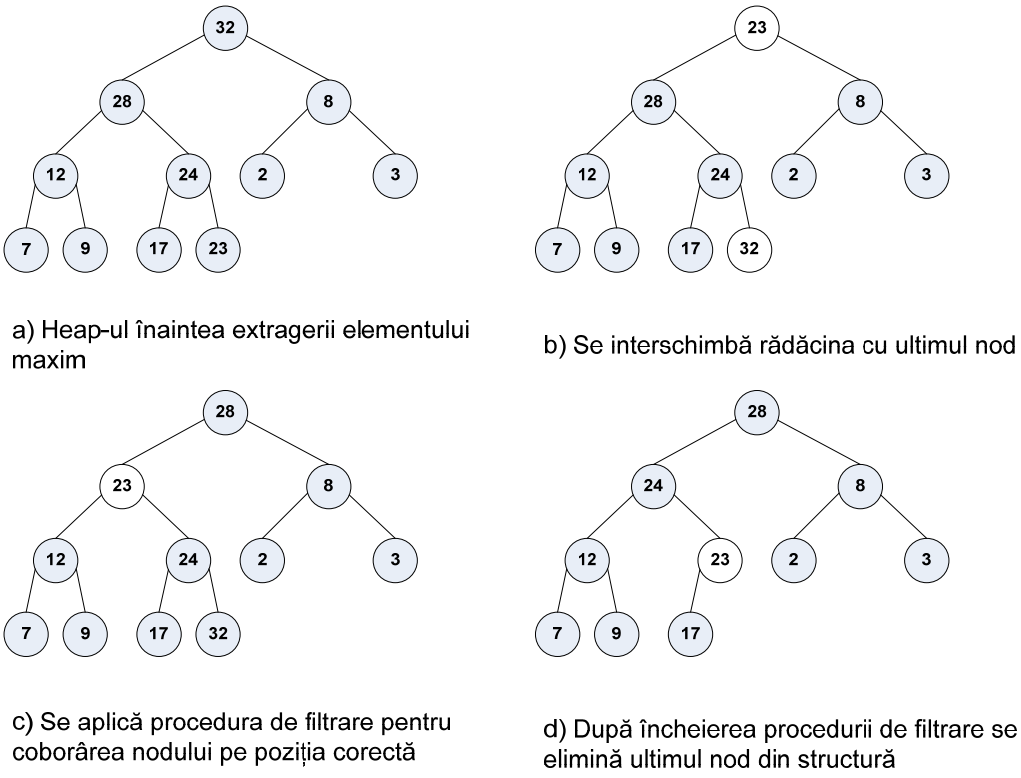

Figura 15.4 *Aplicarea algoritmului extragere element maxim*

Operaţiile prezentate în secţiunea 15.1 permit utilizarea structurii heap ca punct de plecare pentru implementarea eficientă a cozilor de prioritate şi a algoritmului de sortare HeapSort.

15.2 Implementarea structurii *Heap*

Deşi este posibilă implementarea structurii heap folosind arbori binari, datorită particularităţilor arborelui de tip heap, stocarea eficientă a acestuia poate realizează fără ajutorul pointerilor, folosind un masiv unidimensional. Elementele arborelui se stochează în masiv începând cu nodul rădăcină şi continuând cu nodurile de pe nivelurile următoare preluate de la stânga la dreapta. Reprezentarea sub formă de masiv a heap-ului din figura 15.1 este prezentată în figura 15.5.

32	24	8	12	23	2	3	7	9	17
0	1	2	3	4	5	6	7	8	9

Figura 15.5 *Reprezentarea în memorie pentru structura heap*

Navigarea între elementele arborelui se poate face în ambele direcţii folosind următoarele formule:

$$\text{Parinte}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor, \text{Stânga}(i) = 2 \cdot i + 1, \text{Dreapta}(i) = 2 \cdot i + 2 \qquad (15.1)$$

Pentru implementarea structurii heap în limbajul C++ a fost construită o clasă template denumită *Heap* care implementează algoritmii prezentaţi anterior. Clasa permite construirea de heap-uri pentru orice tip de date care implementează operatorul „<” (necesar pentru construirea relaţiei de ordine pe mulţimea nodurilor) şi un constructor implicit.

Interfaţa clasei este următoarea:

```
template <class T>
class Heap
{
public:

	// --- Constructori şi destructor --- //

	// Construire heap nou (cu dimensiunea specificată)
	Heap(int capacitateInitiala = DimensiuneMinima);

	// Construire heap pe baza unor elemente date
	Heap(T elemente[], int numarElemente);

	// Constructor de copiere
	Heap(Heap& heap);

	~Heap() { delete [] this->elemente; }

	// --- Metode coada prioritate  --- //

	// Inserează un element în structură
	void Insereaza(T element);

	// Intoarce elementul maxim si il elimina din structura
	T ExtrageMaxim();

	// Intoarce elementul maxim dar nu il elimina din structura
	T CitesteMaxim();

	// --- Acces la elemente --- //

	int GetNumarElemente() { return this->dimensiuneHeap; }

	T operator[] (int index) { return this->elemente[index]; }

private:

	// --- Datele structurii --- //

	T* elemente;		// pointer la vectorul de elemente

	// Invariant: dimensiuneHeap <= memorieAlocata
	int memorieAlocata; // dimensiunea mem alocate (ca nr de elem)
	int dimensiuneHeap; // nr de elemente existente în heap

	static const int DimensiuneMinima = 10;

	// --- Functia de reordonare a elementelor --- //

	void Filtrare(int index);

	// --- Calculare pozitie elemente --- //
```

```
	inline int Parinte(int i)
	{
		// <=> i / 2 cu indici pe 1..n
		return (i - 1) / 2;
	}
	inline int Stanga(int i)
	{
		// <=> i * 2 cu indici pe 1..n
		return 2 * i + 1;
	}
	inline int Dreapta(int i)
	{
		// <=> i * 2 + 1 cu indici pe 1..n
		return 2 * i + 2;
	}

	inline void Interschimb(int index1, int index2)
	{
		T temp = this->elemente[index1];
		this->elemente[index1] = this->elemente[index2];
		this->elemente[index2] = temp;
	}

	//--- Metode folosite pt realocarea dinamica a structurii --//

	void CresteMemoriaAlocata();
	void ScadeMemoriaAlocata();

	// Nu permitem operatii de atribuire intre obiecte de tip Heap
	Heap operator= (const Heap&);
};
```

Se observă că, în afara datelor efective stocate în masiv, este necesară stocarea a două informaţii suplimentare:

- dimensiunea memoriei alocate pentru structură;
- numărul de elemente prezente efectiv în structură.

Memoria necesară stocării masivului este alocată dinamic şi gestionată automat de către clasa Heap prin intermediul metodelor *CresteMemoriaAlocata* şi *ScadeMemoria* alocată:

```
template <class T>
void Heap<T>::CresteMemoriaAlocata()
{
	// alocăm un vector nou de dimensiune dublă şi copiem elem
	T* masivNou = new T[this->memorieAlocata * 2];
	for (int i = 0; i < this->memorieAlocata; i++)
	{
		masivNou[i] = this->elemente[i];
	}

	// înlocuim vectorul existent cu cel nou şi actualizăm dim
	delete []  this->elemente;
	this->elemente = masivNou;

	this->memorieAlocata = this->memorieAlocata * 2;
}

template <class T>
void Heap<T>::ScadeMemoriaAlocata()
```

```
{
	// alocăm un vector nou mai mic şi copiem elementele
	T* masivNou = new T[this->memorieAlocata / 2];
	for (int i = 0; i < this->memorieAlocata / 2; i++)
	{
		masivNou[i] = this->elemente[i];
	}

	// înlocuim vectorul existent cu cel nou şi actualizăm dim
	delete [] this->elemente;
	this->elemente = masivNou;

	this->memorieAlocata = this->memorieAlocata / 2;
}
```

Memoria alocată de către instanţele clasei este dealocată de către destructor. Pentru a evita cazul în care două instanţe ale clasei *Heap* conţin referinţe către aceleaşi elemente a fost interzisă operaţia de atribuire prin supraîncărcarea privată a operatorului corespunzător.

Pentru construirea structurii heap pe baza unui masiv oarecare şi pentru asigurarea proprietăţii de ordonare în cazul metodei de extragere maxim a fost implementată metoda ajutătoare *Filtrare* conform algoritmului prezentat în secţiunea 15.1:

```
template <class T>
void Heap<T>::Filtrare(int index)
{
	// 1. Determinam maximul dintre elemente[index],
	// elemente[Stanga(index)] si elemente[Dreapta(index)]
	int indexMax = index;
	int indexStanga = this->Stanga(index);
	int indexDreapta = this->Dreapta(index);

	if (indexStanga < this->dimensiuneHeap &&
		this->elemente[indexStanga] > this->elemente[indexMax])
	{
		indexMax = indexStanga;
	}

	if (indexDreapta < this->dimensiuneHeap &&
		this->elemente[indexDreapta] > this->elemente[indexMax])
	{
		indexMax = indexDreapta;
	}

	// 2. Daca varful actual nu respecta prop de ordonare atunci
	// coboram nodul in arbore si reapelam recursiv procedura
	if (index != indexMax)
	{
		this->Interschimb(index, indexMax);
		this->Filtrare(indexMax);
	}
}
```

Constructorii clasei Heap realizează iniţializarea structurii în trei cazuri distincte:

- construirea unei structuri noi cu o capacitate iniţială specificată opţional de către utilizator;

- construirea structurii noi pe baza elementelor unui masiv unidimensional primit ca parametru, folosind algoritmul de filtrare prezentat în secţiunea 15.1;
- construirea unui heap prin copierea datelor dintr-o structură heap existentă.

Implementarea constructorilor în cadrul clasei *Heap* este următoarea:

```
template <class T>
Heap<T>::Heap(int capacitateInitiala)
{
	// ne asigurăm că dimensiunea inițială a structurii este
	// cel puțin dimensiunea minimă
	capacitateInitiala = max(capacitateInitiala, DimensiuneMinima);

	// alocăm memoria pentru elemente şi inițializăm câmpurile
	this->elemente = new T[capacitateInitiala];
	this->memorieAlocata = capacitateInitiala;
	this->dimensiuneHeap = 0;
}

template <class T>
Heap<T>::Heap(T elemente[], int numarElemente)
{
	// inițializăm câmpurile
	this->memorieAlocata =
		max(numarElemente, DimensiuneMinima);
	this->dimensiuneHeap = numarElemente;

	// copiem elementele din vector în structură
	this->elemente = new T[this->memorieAlocata];
	for (int i = 0; i < numarElemente; i++)
	{
		this->elemente[i] = elemente[i];
	}

	// Rearanjăm elem a.î. să satisfacă prop de ordonare folosind
	// metoda Filtrare pt coborârea elem în arbore (elementele din
	// a doua jumatate a masivului respecta implicit proprietatea
	// de heap deoarece reprezinta subarbori cu maxim un element)
	for (int i = (numarElemente - 1) / 2; i >= 0; i--)
	{
		this->Filtrare(i);
	}
}

template <class T> Heap<T>::Heap(Heap& heap)
{
	this->memorieAlocata = heap->memorieAlocata;
	this->dimensiuneHeap = heap->dimensiuneHeap;

	this->elemente = new T[this->memorieAlocata];
	for (int i = 0; i < numarElemente; i++)
	{
		this->elemente[i] = heap->elemente[i];
	}
}
```

Medodele de inserare şi extragere nod au fost implementate conform algoritmilor prezenzaţii în secţiunea 15.1.

```
template <class T>
void Heap<T>::Insereaza(T element)
{
	// verificam faptul că avem memorie disponibilă
	if (this->dimensiuneHeap == this->memorieAlocata)
	{
		this->CresteMemoriaAlocata();
	}

	// expandam heap-ul
	this->dimensiuneHeap++;

	// adaugăm elementul nou la sfârşitul heap-ului
	int index = this->dimensiuneHeap - 1;
	this->elemente[index] = element;

	// si îl urcăm în arbore atât cât este cazul
	// pentru a păstra proprietatea de heap
	while (this->Parinte(index) >= 0 &&
		this->elemente[index] > this->elemente[Parinte(index)])
	{
		this->Interschimb(index, this->Parinte(index));
		index = this->Parinte(index);
	}
}

template <class T>
T Heap<T>::ExtrageMaxim()
{
	// ne asigurăm că avem cel puțin un element în heap
	assert(this->dimensiuneHeap > 0);

	// scădem memoria alocată dacă este cazul
	if (this->memorieAlocata > DimensiuneMinima * 2 &&
		this->dimensiuneHeap < this->memorieAlocata / 3)
	{
		this->ScadeMemoriaAlocata();
	}

	//mutăm elementul în afara heap-ului şi refacem struct de heap
	this->dimensiuneHeap--;
	Interschimb(0, this->dimensiuneHeap);

	this->Filtrare(0);

	return this->elemente[this->dimensiuneHeap];
}

template <class T>
T Heap<T>::CitesteMaxim()
{
	// ne asigurăm că avem cel puțin un element în heap
	assert(this->dimensiuneHeap > 0);

	// şi întoarcem maximul
	return this->elemente[0];
}
```

Metodele de inserare şi extragere utilizează funcţiile *CresteMemoriaAlocata* şi *ScadeMemoria* pentru extinderea / restrângerea memoriei utilizate de către heap.

15.3 Cozi de prioritate

Cozile de prioritate sunt structuri de date care suportă următoarele două operaţii de bază:

- inserarea unui element cu o prioritate asociată;
- extragerea elementului cu prioritate maximă.

Cele mai importante aplicaţii ale cozilor de prioritate sunt: simularea bazată pe evenimente, gestionarea resurselor partajate (lăţime de bandă, timp de procesare) şi căutare în spaţiul soluţiilor (de exemplu, algoritmul A* utilizează o coadă de prioritate pentru a reţine rutele neexplorate).

Structura de date de tip Heap este una dintre cele mai eficiente modalităţi de implementare a cozilor de prioritate. Prioritatea elementelor este dată de relaţia de ordine existentă între valorile asociate nodurilor. Pentru exemplificarea modului de utilizare a clasei Heap prezentată în secţiunea 15.2 vom construi un simulator discret pentru o coadă de aşteptare la un magazin.

În simularea discretă, modul de operare al unui sistem este reprezentat sub forma unei secvenţe de evenimente ordonate cronologic. În cazul de faţă evenimentele sunt sosirile clienţilor în coada de aşteptare şi servirea clienţilor. Simulatorul conţine o coadă de evenimente. Evenimentele sunt adăugate în coadă pe măsură ce timpul lor de producere poate fi determinat şi sunt extrase din coadă pentru procesare în ordine cronologică.

Un simulator discret pe bază de evenimente are următoarele componente:

- coada de evenimente – o coadă de prioritate care conţine lista evenimentelor care se vor petrece în viitor;
- starea simulatorului – conţine un contor pentru memorarea timpului curent, informaţiile referitoare la starea actuală a sistemului simulat (în cazul curent clienţii aflaţi în coadă şi starea staţiei de servire) şi indicatori;
- logica de procesare – extrage din coadă evenimentele în ordine cronologică şi le procesează; procesarea unui eveniment determină modificarea stării sistemului şi generarea de alte evenimente.

Pentru simularea propusă au fost luate în considerare următoarele ipoteze:

- există o singură staţie de servire cu un timp de servire distribuit normal, cu o medie şi dispersie cunoscută;
- există o singură coadă pentru clienţi, iar intervalul de timp dintre două sosiri este distribuit uniform într-un interval dat;
- durata simulării este stabilită de către utilizator.

Simularea se realizează prin extragerea evenimentelor din heap şi procesarea acestora pe bază de reguli. Evenimentele de tip sosire determină generarea evenimentului corespunzător sosirii următoare şi a unui eveniment de servire în cazul în care staţia este liberă la momentul curent. În cazul evenimentelor de tip servire se generează următorul eveniment de

tip servire dacă mai există clienţi în coadă. Pe măsură ce sunt procesate evenimentele sunt reţinute şi informaţiile necesare pentru calcularea indicatorilor de performanţă aferenţi sistemului simulat.

Codul sursă pentru implementarea simulatorului pe baza clasei Heap prezentată în secţiunea 15.2 este următorul:

```
#include <cmath>
#include <ctime>
#include <iostream>

#include "Heap.h"

// --- Definirea clasei Eveniment --- //

enum TipEveniment { Sosire, Servire };

class Eveniment
{
public:

	// Constructor
	Eveniment(int timp = 0, TipEveniment tip = Sosire) :
	  timp(timp), tip(tip) {}

	// Implementarea relatiei de ordine pentru min-heap
	bool operator > (Eveniment& eveniment)
	{
		return this->timp < eveniment.timp;
	}

	// Acces la elemente
	int GetTimp() { return this->timp; }
	TipEveniment GetTip() { return this->tip; }

private:
	int timp;
	TipEveniment tip;
};

// --- Generarea timpilor de sosire / servire --- //

int GenerareTimpSosire(int timpCurent,
	int minIntervalSosire, int maxIntervalSosire)
{
	return timpCurent + minIntervalSosire + rand() %
maxIntervalSosire;
}

int GenerareTimpServire(int timpCurent,
	int medieServire, int dispersieServire)
{
	// generare variabila aleatoare distribuita
	// normal folosind metoda Box - Muller
	double a = (double)rand() / RAND_MAX;
	double b = (double)rand() / RAND_MAX;

	double PI = 3.14159265358979323846;

	double c = sqrt(-2 * log(a)) * cos(2 * PI * b);
```

```
      return timpCurent + abs((int)
         (medieServire + c * dispersieServire));
}

// --- Simularea magazinului --- //

void Simulare(

      // Parametri de intrare:
      int durata, int medieServire, int dispersieServire,
      int minIntervalSosire, int maxIntervalSosire,

      // Parametri de iesire:
      double& timpMediuAsteptare,
      double& timpMediuServire,
      double& lungimeMedieCoada,
      int& numarClientiServiti)
{
      // inițializare indicatori
      int timp = 0;
      int numarMasuratoriCoada = 0;
      timpMediuAsteptare = 0;
      timpMediuServire = 0;
      lungimeMedieCoada = 0;
      numarClientiServiti = 0;

      // lista de clienti care asteapta sa fie serviti (inclusiv
      // clientul care este servit la momentul curent)
      Heap<Eveniment> clientiInAsteptare;

      // lista de evenimente programate in viitor
      Heap<Eveniment> coadaEvenimente;

      // simularea incepe cu o sosire
      coadaEvenimente.Insereaza(Eveniment(0, Sosire));

      while(timp < durata)
      {
            // extragem ev următor din coadă (în funcție de timp)
            Eveniment e = coadaEvenimente.ExtrageMaxim();
            timp = e.GetTimp();

         lungimeMedieCoada += clientiInAsteptare.GetNumarElemente();
            numarMasuratoriCoada++;

            if (e.GetTip() == Sosire)
            {
                  // A. Eveniment de tip sosire

                  // adăugăm clientul în coada de aşteptare
                  clientiInAsteptare.Insereaza(e);

                  // programăm sosirea următoare
                  Eveniment sosire = Eveniment(GenerareTimpSosire(
            timp, minIntervalSosire, maxIntervalSosire), Sosire);

                  coadaEvenimente.Insereaza(sosire);

                  // dacă este unicul client din coadă
                  if (clientiInAsteptare.GetNumarElemente() == 1)
                  {
```

```
                        // atunci programăm şi servirea
                        Eveniment servire(GenerareTimpServire(
                       timp, medieServire, dispersieServire), Servire);

                        coadaEvenimente.Insereaza(servire);

                        timpMediuServire += servire.GetTimp() - timp;
                  }
            }
            else
            {
                  // B. Eveniment de tip servire

                  // eliminăm primul client din coadă (cel servit)
                  Eveniment clientServit =
                    clientiInAsteptare.ExtrageMaxim();

                  timpMediuAsteptare += timp -
                    clientServit.GetTimp();
                  numarClientiServiti++;

                  // dacă mai avem clienți în aşteptare
                  if (clientiInAsteptare.GetNumarElemente() > 0)
                  {
                     // programăm servirea pt clientul următor
                     Eveniment servire(GenerareTimpServire(
                    timp, medieServire, dispersieServire), Servire);

                     coadaEvenimente.Insereaza(servire);

                     timpMediuServire += servire.GetTimp() - timp;
                  }
            }
      }

      // calcul indicatori
      lungimeMedieCoada = lungimeMedieCoada / numarMasuratoriCoada;
      timpMediuAsteptare = timpMediuAsteptare / numarClientiServiti;
      timpMediuServire = timpMediuServire /
      (numarClientiServiti + clientiInAsteptare.GetNumarElemente());

}


void main()
{
      // --- Citirea datelor de intrare --- //

      // Parametri de intrare:
      int durata, medieServire, dispersieServire;
      int minIntervalSosire, maxIntervalSosire;

      // Parametri de iesire:
      double timpMediuAsteptare;
      double timpMediuServire;
      double lungimeMedieCoada;
      int numarClientiServiti;

      // Citire parametri:
      cout << "Durata simularii:";
      cin >> durata;
```

```
	cout << "Media timpului de servire: ";
	cin >> medieServire;

	cout << "Dispersia timpului de servire: ";
	cin >> dispersieServire;

	cout << "Intervalul minim dintre sosiri: ";
	cin >> minIntervalSosire;

	cout << "Intervalul maxim dintre sosiri: ";
	cin >> maxIntervalSosire;


	// --- Simularea procesului --- //

	cout << endl;
	cout << "Simulare proces cu durata " << durata << "." << endl;
	cout << "Interval intre sosiri: [" << minIntervalSosire <<
		", " << maxIntervalSosire << "]" << endl;
	cout << "Timp servire ~N(" << medieServire << "," <<
		dispersieServire << ")" << endl;
	cout << endl;

	Simulare(
		// Parametri de intrare:
		durata,
		medieServire, dispersieServire,
		minIntervalSosire, maxIntervalSosire,

		// Parametri de iesire:
		timpMediuAsteptare,	timpMediuServire,
		lungimeMedieCoada,numarClientiServiti);

	// --- Afisarea rezultatelor --- //

	cout << endl << "Rezultate simulare:" << endl;
	cout << "Nr clienti serviti:" << numarClientiServiti << endl;
	cout << "Lungime medie coada:" << lungimeMedieCoada << endl;
	cout << "Timpul mediu de asteptare pentru clienti: " <<
		fixed << timpMediuAsteptare << endl;
	cout << "Timpul mediu de servire: " <<
		fixed << timpMediuServire << endl;
}
```

15.4 Sortarea datelor prin *HeapSort*

O altă aplicaţie a structurii heap este implementarea algoritmului de sortare Heapsort. Sortarea presupune extragerea elementelor din heap şi stocarea acestora, în ordine inversă, la sfârşitul masivului utilizat pentru memorarea structurii.

Codul sursă pentru implementarea algoritmului Heapsort în cadrul clasei Heap este următorul:

```
template <class T>
void Heap<T>::SorteazaHeap(T* destinatie)
{
	// salvăm numărul de elemente din heap
```

```
	int dimensiuneaInitiala = this->dimensiuneHeap;

	// sortăm vectorul în cadrul structurii
	for (int i = this->dimensiuneHeap - 1; i > 0; i--)
	{
		this->Interschimb(0, i);

		this->dimensiuneHeap--;
		this->Filtrare(0);
	}

	// copiem elementele sortate în vectorul destinație
	for (int i = 0; i < dimensiuneaInitiala; i++)
	{
		destinatie[i] = this->elemente[i];
	}
}
```

Algoritmul poate fi utilizat pentru sortarea oricărui masiv unidimensional pentru care a fost definită o relaţie de ordine.

Codul sursă pentru sortarea unui vector de numere întregi este următorul:

```
void main()
{

	// 1. Construirea vectorului de sortat si a heap-ului asociat
	const int NumarElemente = 8;
	int elemente[] = {23, 32, 2, 6, 23, 8, 3, 6};
	Heap<int> heap(elemente, NumarElemente);

	// 2. Obtinerea si afisarea vectorului sortat
	int* elementeSortate = new int[NumarElemente];
	heap.SorteazaHeap(elementeSortate);

	for (int i = 0; i < NumarElemente; i++)
	{
		cout << elementeSortate[i] << " ";
	}
	cout << endl;
}
```