7 Heapsort

În acest capitol vom prezenta un nou algoritm de sortare. Asemănător sortării prin interclasare, dar diferit de sortarea prin inserare, timpul de execuţie al agoritmului heapsort este $O(n \lg n)$. Asemănător sortării prin inserare şi diferit de sortarea prin interclasare, prin heapsort se ordonează elementele în spaţiul alocat vectorului: la un moment dat doar un număr constant de elemente ale vectorului sunt păstrate în afara spaţiului alocat vectorului de intrare. Astfel, algoritmul heapsort combină calităţile celor doi algoritmi de sortare prezentaţi deja.

Heapsort introduce o tehnică nouă de proiectare a algoritmilor bazată pe utilizarea unei structuri de date, numită de regulă ***heap***[1]. Structura de date heap este utilă nu doar pentru algoritmul heapsort, ea poate fi la fel de utilă şi în tratarea eficientă a unei cozi de prioritate. Cu structura de date heap ne vom mai întâlni în algoritmii din capitolele următoare.

Amintim că termenul heap a fost introdus şi utilizat iniţial în contextul algoritmului heapsort, dar acesta se foloseşte şi în legătură cu alocarea dinamică, respectiv în tratarea memoriei bazate pe "colectarea reziduurilor" (*garbage collected storage*), de exemplu în limbajele de tip Lisp. Structura de date heap ***nu*** se referă la heap-ul menţionat în alocarea dinamică, şi ori de câte ori, în această carte vom vorbi despre heap, vom înţelege structura definită în acest capitol.

7.1. Heap-uri

Structura de date ***heap (binar)*** este un vector care poate fi vizualizat sub forma unui arbore binar aproape complet (vezi secţiunea 5.5.3), conform figurii 7.1. Fiecare nod al arborelui corespunde unui element al vectorului care conţine valorile ataşate nodurilor. Arborele este plin, exceptând eventual nivelul inferior, care este plin de la stânga la dreapta doar până la un anumit loc. Un vector A care reprezintă un heap are două atribute: $lungime[A]$, reprezintă numărul elementelor din vector şi $dimensiune\text{-}heap[A]$ reprezintă numărul elementelor heap-ului memorat în vectorul A. Astfel, chiar dacă $A[1..lungime[A]]$ conţine în fiecare element al său date valide, este posibil ca elementele următoare elementului $A[dimensiune\text{-}heap[A]]$, unde $dimensiune\text{-}heap[A] \leq lungime[A]$, să nu aparţină heap-ului. Rădăcina arborelui este $A[1]$. Dat fiind un indice i, corespunzător unui nod, se pot determina uşor indicii părintelui acestuia PĂRINTE(i), al fiului STÂNGA(i) şi al fiului DREAPTA(i).

PĂRINTE(i)
  returnează $\lfloor i/2 \rfloor$

STÂNGA(i)
  returnează $2i$

DREAPTA(i)
  returnează $2i + 1$

[1] unii autori români, folosesc termenul "ansamblu".

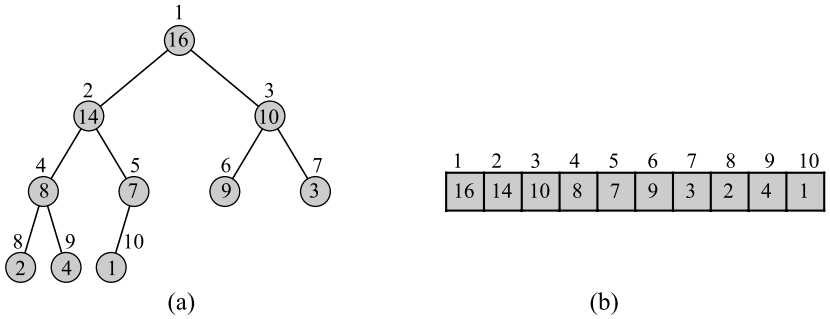

Figura 7.1 Un heap reprezentat sub forma unui arbore binar **(a)** şi sub forma unui vector **(b)**. Numerele înscrise în cercurile reprezentând nodurile arborelui sunt valorile ataşate nodurilor, iar cele scrise lângă cercuri sunt indicii elementelor corespunzătoare din vector.

În cele mai multe cazuri, procedura STÂNGA poate calcula valoarea $2i$ cu o singură instrucţiune, translatând reprezentarea binară a lui i la stânga cu o poziţie binară. Similar, procedura DREAPTA poate determina rapid valoarea $2i + 1$, translatând reprezentarea binară a lui i la stânga cu o poziţie binară, iar bitul nou intrat pe poziţia binară cea mai nesemnificativă va fi 1. În procedura PĂRINTE valoarea $\lfloor i/2 \rfloor$ se va calcula prin translatarea cu o poziţie binară la dreapta a reprezentării binare a lui i. Într-o implementare eficientă a algoritmului heapsort, aceste proceduri sunt adeseori codificate sub forma unor "macro-uri" sau a unor proceduri "in-line".

Pentru orice nod i, diferit de rădăcină, este adevărată următoarea ***proprietate de heap***:

$$A[\text{PĂRINTE}(i)] \geq A[i], \tag{7.1}$$

adică valoarea ataşată nodului este mai mică sau egală cu valoarea asociată părintelui său. Astfel cel mai mare element din heap este păstrat în rădăcină, iar valorile nodurilor oricărui subarbore al unui nod sunt mai mici sau egale cu valoarea nodului respectiv.

Definim ***înălţimea*** unui nod al arborelui ca fiind numărul muchiilor aparţinând celui mai lung drum care leagă nodul respectiv cu o frunză, iar înălţimea arborelui ca fiind înălţimea rădăcinii. Deoarece un heap având n elemente corespunde unui arbore binar complet, înălţimea acestuia este $\Theta(\lg n)$ (vezi exerciţiul 7.1-2). Vom vedea că timpul de execuţie al operaţiilor de bază, care se efectuează pe un heap, este proporţional cu înălţimea arborelui şi este $O(\lg n)$. În cele ce urmează, vom prezenta cinci proceduri şi modul lor de utilizare în algoritmul de sortare, respectiv într-o structură de tip coadă de prioritate.

- Procedura RECONSTITUIE-HEAP are timpul de execuţie $O(\lg n)$ şi este de primă importanţă în întreţinerea proprietăţii de heap (7.1).
- Procedura CONSTRUIEŞTE-HEAP are un timp de execuţie liniar şi generează un heap dintr-un vector neordonat, furnizat la intrare.
- Procedura HEAPSORT se execută în timpul $O(n \lg n)$ şi ordonează un vector în spaţiul alocat acestuia.
- Procedurile EXTRAGE-MAX şi INSEREAZĂ se execută în timpul $O(\lg n)$, iar cu ajutorul lor se poate utiliza un heap în realizarea unei cozi de prioritate.

Exerciţii

7.1-1 Care este cel mai mic, respectiv cel mai mare număr de elemente dintr-un heap având înălţimea h?

7.1-2 Arătaţi că un heap având n elemente are înălţimea $\lfloor \lg n \rfloor$.

7.1-3 Arătaţi că în orice subarbore al unui heap, rădăcina subarborelui conţine cea mai mare valoare care apare în acel subarbore.

7.1-4 Unde se poate afla cel mai mic element al unui heap, presupunând că toate elementele sunt distincte?

7.1-5 Este vectorul în care elementele se succed în ordine descrescătoare un heap?

7.1-6 Este secvenţa $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ un heap?

7.2. Reconstituirea proprietăţii de heap

Procedura RECONSTITUIE-HEAP este un subprogram important în prelucrarea heap-urilor. Datele de intrare ale acesteia sunt un vector A şi un indice i din vector. Atunci când se apelează RECONSTITUIE-HEAP, se presupune că subarborii, având ca rădăcini nodurile STÂNGA(i) respectiv DREAPTA(i), sunt heap-uri. Dar, cum elementul $A[i]$ poate fi mai mic decât descendenţii săi, acesta nu respectă proprietatea de heap (7.1). Sarcina procedurii RECONSTITUIE-HEAP este de a "scufunda" în heap valoarea $A[i]$, astfel încât subarborele care are în rădăcină valoarea elementului de indice i, să devină un heap.

```
RECONSTITUIE-HEAP(A, i)
 1: l ← STÂNGA(i)
 2: r ← DREAPTA(i)
 3: dacă l ≤ dimesiune-heap[A] şi A[l] > A[i] atunci
 4:    maxim ← l
 5: altfel
 6:    maxim ← i
 7: dacă r ≤ dimesiune-heap[A] şi A[r] > A[maxim] atunci
 8:    maxim ← r
 9: dacă maxim ≠ i atunci
10:    schimbă  A[i] ↔ A[maxim]
11:    RECONSTITUIE-HEAP(A, maxim)
```

Figura 7.2 ilustrează efectul procedurii RECONSTITUIE-HEAP. La fiecare pas se determină cel mai mare element dintre $A[i]$, $A[\text{STÂNGA}(i)]$ şi $A[\text{DREAPTA}(i)]$, iar indicele său se păstrează în variabila $maxim$. Dacă $A[i]$ este cel mai mare, atunci subarborele având ca rădăcină nodul i este un heap şi procedura se termină. În caz contrar, cel mai mare element este unul dintre cei doi descendenţi şi $A[i]$ este interschimbat cu $A[maxim]$. Astfel, nodul i şi descendenţii săi satisfac proprietatea de heap. Nodul $maxim$ are acum valoarea iniţială a lui $A[i]$, deci este posibil ca

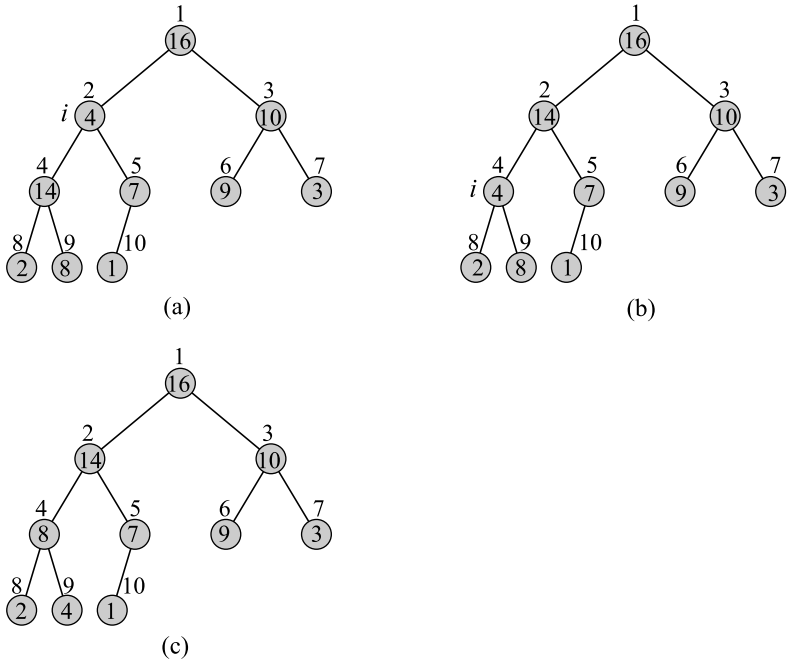

Figura 7.2 Efectul procedurii RECONSTITUIE-HEAP$(A, 2)$, unde *dimesiune-heap*$[A] = 10$. **(a)** Configuraţia iniţială a heap-ului, unde $A[2]$ (pentru nodul $i = 2$), nu respectă proprietatea de heap deoarece nu este mai mare decât descendenţii săi. Proprietatea de heap este restabilită pentru nodul 2 în **(b)** prin interschimbarea lui $A[2]$ cu $A[4]$, ceea ce anulează proprietatea de heap pentru nodul 4. Apelul recursiv al procedurii RECONSTITUIE-HEAP$(A, 4)$ poziţionează valoarea lui i pe 4. După interschimbarea lui $A[4]$ cu $A[9]$, aşa cum se vede în **(c)**, nodul 4 ajunge la locul său şi apelul recursiv RECONSTITUIE-HEAP$(A, 9)$ nu mai găseşte elemente care nu îndeplinesc proprietatea de heap.

subarborele de rădăcină *maxim* să nu îndeplinească proprietatea de heap. Rezultă că procedura RECONSTITUIE-HEAP trebuie apelată recursiv din nou pentru acest subarbore.

Timpul de execuţie al procedurii RECONSTITUIE-HEAP, corespunzător unui arbore de rădăcină i şi dimensiune n, este $\Theta(1)$, timp în care se pot analiza relaţiile dintre $A[i]$, $A[\text{STÂNGA}(i)]$ şi $A[\text{DREAPTA}(i)]$ la care trebuie adăugat timpul în care RECONSTITUIE-HEAP se execută pentru subarborele având ca rădăcină unul dintre descendenţii lui i. Dimensiunea acestor subarbori este de cel mult $2n/3$ – cazul cel mai defavorabil fiind acela în care nivelul inferior al arborelui este plin exact pe jumătate – astfel, timpul de execuţie al procedurii RECONSTITUIE-HEAP poate fi descris prin următoarea inegalitate recursivă:

$$T(n) \leq T(2n/3) + \Theta(1).$$

Soluţia acestei recurenţe se obţine pe baza celui de-al doilea caz al teoremei master (teorema 4.1): $T(n) = O(\lg n)$. Timpul de execuţie al procedurii RECONSTITUIE-HEAP pentru un nod de înălţime h poate fi exprimat alternativ ca fiind egal cu $O(h)$.

Exerciţii

7.2-1 Utilizând ca model figura 7.2, ilustraţi modul de funcţionare al procedurii RECONSTITUIE-HEAP$(A, 3)$ pentru vectorul $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0\rangle$.

7.2-2 Care este efectul apelului procedurii RECONSTITUIE-HEAP(A, i), dacă elementul $A[i]$ este mai mare decât descendenţii săi?

7.2-3 Care este efectul procedurii RECONSTITUIE-HEAP(A, i) pentru $i >$ *dimesiune-heap*$[A]/2$?

7.2-4 Procedura RECONSTITUIE-HEAP poate fi implementată eficient din punct de vedere al timpului constant de execuţie, dar se poate întâmpla ca anumite compilatoare să genereze cod ineficient pentru apelul recursiv din linia a 11-a. Elaboraţi o variantă a procedurii RECONSTITUIE-HEAP, în care recursivitatea este înlocuită cu iterativitate.

7.2-5 Arătaţi că timpul de execuţie a procedurii RECONSTITUIE-HEAP, în cazul unui heap de dimensiune n, este, în cel mai defavorabil caz, $\Omega(\lg n)$. (*Indicaţie:* În cazul unui heap format din n elemente, determinaţi valorile ataşate nodurilor astfel încât procedura RECONSTITUIE-HEAP să fie apelată recursiv pentru toate nodurile aparţinând drumurilor care pornesc de la rădăcină la frunze.)

7.3. Construirea unui heap

Procedura RECONSTITUIE-HEAP poate fi utilizată "de jos în sus" pentru transformarea vectorului $A[1..n]$ în heap, unde $n = lungime[A]$. Deoarece toate elementele subşirului $A[(\lfloor n/2\rfloor + 1)..n]$ sunt frunze, acestea pot fi considerate ca fiind heap-uri formate din câte un element. Astfel, procedura CONSTRUIEŞTE-HEAP trebuie să traverseze doar restul elementelor şi să execute procedura RECONSTITUIE-HEAP pentru fiecare nod întâlnit. Ordinea de prelucrare a nodurilor asigură ca subarborii, având ca rădăcină descendenţi ai nodului i să formeze heap-uri înainte ca RECONSTITUIE-HEAP să fie executat pentru aceste noduri.

CONSTRUIEŞTE-HEAP(A)
1: *dimesiune-heap*$[A] \leftarrow lungime[A]$
2: **pentru** $i \leftarrow \lfloor lungime[A]/2\rfloor$,1 **execută**
3: RECONSTITUIE-HEAP(A, i)

Figura 7.3 ilustrează modul de funcţionare al procedurii CONSTRUIEŞTE-HEAP.

Timpul de execuţie al procedurii CONSTRUIEŞTE-HEAP poate fi calculat simplu, determinând limita superioară a acestuia: fiecare apel al procedurii RECONSTITUIE-HEAP necesită un timp $O(\lg n)$ şi, deoarece pot fi $O(n)$ asemenea apeluri, rezultă că timpul de execuţie poate fi cel mult $O(n \lg n)$. Această estimare este corectă, dar nu este suficient de tare asimptotic.

Vom obţine o limită mai tare observând că timpul de execuţie a procedurii RECONSTITUIE-HEAP depinde de înălţimea nodului în arbore, aceasta fiind mică pentru majoritatea nodurilor. Estimarea noastră mai riguroasă se bazează pe faptul că un heap având n elemente are înălţimea $\lg n$ (vezi exerciţiul 7.1-2) şi că pentru orice înălţime h, în heap există cel mult $\lfloor n/2^{h+1}\rfloor$ noduri de înălţime h (vezi exerciţiul 7.3-3).

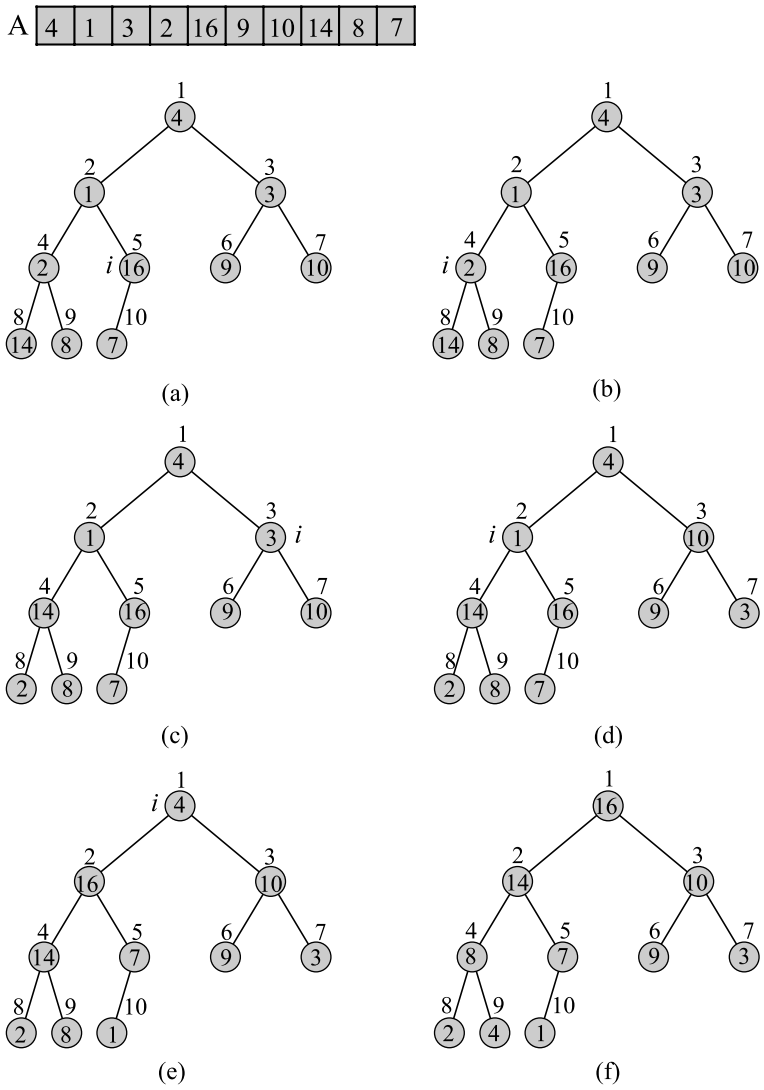

Figura 7.3 Modul de execuţie a procedurii CONSTRUIEŞTE-HEAP. În figură se vizualizează structurile de date în starea lor anterioară apelului procedurii RECONSTITUIE-HEAP (linia 3 din procedura CONSTRUIEŞTE-HEAP). **(a)** Se consideră un vector A având 10 elemente şi arborele binar corespunzător. După cum se vede în figură, variabila de control i a ciclului, în momentul apelului RECONSTITUIE-HEAP(A, i), indică nodul 5. **(b)** reprezintă rezultatul; variabila de control i a ciclului acum indică nodul 4. **(c) - (e)** vizualizează iteraţiile succesive ale ciclului **pentru** din CONSTRUIEŞTE-HEAP. Se observă că, atunci când se apelează procedura RECONSTITUIE-HEAP pentru un nod dat, subarborii acestui nod sunt deja heap-uri. **(f)** reprezintă heap-ul final al procedurii CONSTRUIEŞTE-HEAP.

Timpul de execuţie a procedurii RECONSTITUIE-HEAP pentru un nod de înălţime h fiind $O(h)$, obţinem pentru timpul de execuţie a procedurii CONSTRUIEŞTE-HEAP:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right). \tag{7.2}$$

Ultima însumare poate fi evaluată prin înlocuirea $x = 1/2$ în formula (3.6). Obţinem

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2. \tag{7.3}$$

Astfel, timpul de execuţie al procedurii CONSTRUIEŞTE-HEAP poate fi estimat ca fiind:

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n). \tag{7.4}$$

De aici rezultă că se poate construi un heap dintr-un vector într-un timp liniar.

Exerciţii

7.3-1 Utilizând ca model figura 7.3, ilustraţi modul de funcţionare al procedurii CONSTRUIEŞTE-HEAP pentru vectorul $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.

7.3-2 De ce trebuie micşorată variabila de control i a ciclului în linia 2 a procedurii CONSTRUIEŞTE-HEAP de la $\lfloor lungime[A]/2 \rfloor$ la 1, în loc să fie mărită de la 1 la $\lfloor lungime[A]/2 \rfloor$?

7.3-3 $\star$ Arătaţi că, într-un heap de înălţime h având n elemente, numărul nodurilor este cel mult $\lceil n/2^{h+1} \rceil$.

7.4. Algoritmul heapsort

Algoritmul heapsort începe cu apelul procedurii CONSTRUIEŞTE-HEAP în scopul transformării vectorului de intrare $A[1..n]$ în heap, unde $n = lungime[A]$. Deoarece cel mai mare element al vectorului este ataşat nodului rădăcină $A[1]$, acesta va ocupa locul definitiv în vectorul ordonat prin interschimbarea sa cu $A[n]$. În continuare, "excluzând" din heap cel de-al n-lea element (şi micşorând cu 1 *dimesiune-heap*$[A]$), restul de $A[1..(n-1)]$ elemente se pot transforma uşor în heap, deoarece subarborii nodului rădăcină au proprietatea de heap (7.1), cu eventuala excepţie a elementului ajuns în nodul rădăcină.

HEAPSORT(A)
1: CONSTRUIEŞTE-HEAP(A)
2: **pentru** $i \leftarrow lungime[A]$, 2 **execută**
3: schimbă $A[1] \leftrightarrow A[i]$
4: $dimesiune\text{-}heap[A] \leftarrow dimesiune\text{-}heap[A] - 1$
5: RECONSTITUIE-HEAP$(A, 1)$

Apelând procedura RECONSTITUIE-HEAP$(A, 1)$ se restabileşte proprietatea de heap pentru vectorul $A[1..(n-1)]$. Acest procedeu se repetă micşorând dimensiunea heap-ului de la $n-1$ la 2.

Figura 7.4 ilustrează, pe un exemplu, modul de funcţionare a procedurii HEAPSORT, după ce în prealabil datele au fost transformate în heap. Fiecare heap reprezintă starea iniţială la începutul pasului iterativ (linia 2 din ciclul **pentru**).

Timpul de execuţie al procedurii HEAPSORT este $O(n \lg n)$, deoarece procedura CONSTRUIEŞTE-HEAP se execută într-un timp $O(n)$, iar procedura RECONSTITUIE-HEAP, apelată de $n-1$ ori, se execută în timpul $O(\lg n)$.

Exerciţii

7.4-1 Utilizând ca model figura 7.4, ilustraţi modul de funcţionare al procedurii HEAPSORT pentru vectorul $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$.

7.4-2 Care este timpul de execuţie al algoritmului heapsort în cazul unui vector A de dimensiune n, ordonat crescător? Dar în cazul unui vector ordonat descrescător?

7.4-3 Arătaţi că timpul de execuţie al algoritmului heapsort este $\Omega(n \lg n)$.

7.5. Cozi de priorităţi

Heapsort este un algoritm excelent, dar o implementare bună a algoritmului de sortare rapidă, algoritm prezentat în capitolul 8, se poate dovedi de multe ori mai eficientă. În schimb, structura de date heap este foarte utilă. În această secţiune vom prezenta una din cele mai frecvente aplicaţii ale unui heap: utilizarea lui sub forma unei cozi de priorităţi.

Coada de priorităţi este o structură de date care conţine o mulţime S de elemente, fiecare având asociată o valoare numită ***cheie***. Asupra unei cozi de priorităţi se pot efectua următoarele operaţii.

INSEREAZĂ(S, x) inserează elementul x în mulţimea S. Această operaţie poate fi scrisă în felul următor: $S \leftarrow S \cup \{x\}$.

MAXIM(S) returnează elementul din S având cheia cea mai mare.

EXTRAGE-MAX(S) elimină şi returnează elementul din S având cheia cea mai mare.

O aplicaţie a cozilor de priorităţi constă în planificarea lucrărilor pe calculatoare partajate. Sarcinile care trebuie efectuate şi priorităţile relative se memorează într-o coadă de prioritate. Când o lucrare este terminată sau întreruptă, procedura EXTRAGE-MAX va selecta lucrarea având prioritatea cea mai mare dintre lucrările în aşteptare. Cu ajutorul procedurii INSERARE, în coadă poate fi introdusă oricând o lucrare nouă.

O coadă de priorităţi poate fi utilizată şi în simularea unor evenimente controlate. Din coadă fac parte evenimentele de simulat, fiecăruia ataşându-i-se o cheie reprezentând momentul când va avea loc evenimentul. Evenimentele trebuie simulate în ordinea desfăşurării lor în timp, deoarece simularea unui eveniment poate determina necesitatea simulării altuia în viitor. În cazul acestei aplicaţii, este evident că ordinea evenimentelor în coada de priorităţi trebuie inversată, iar

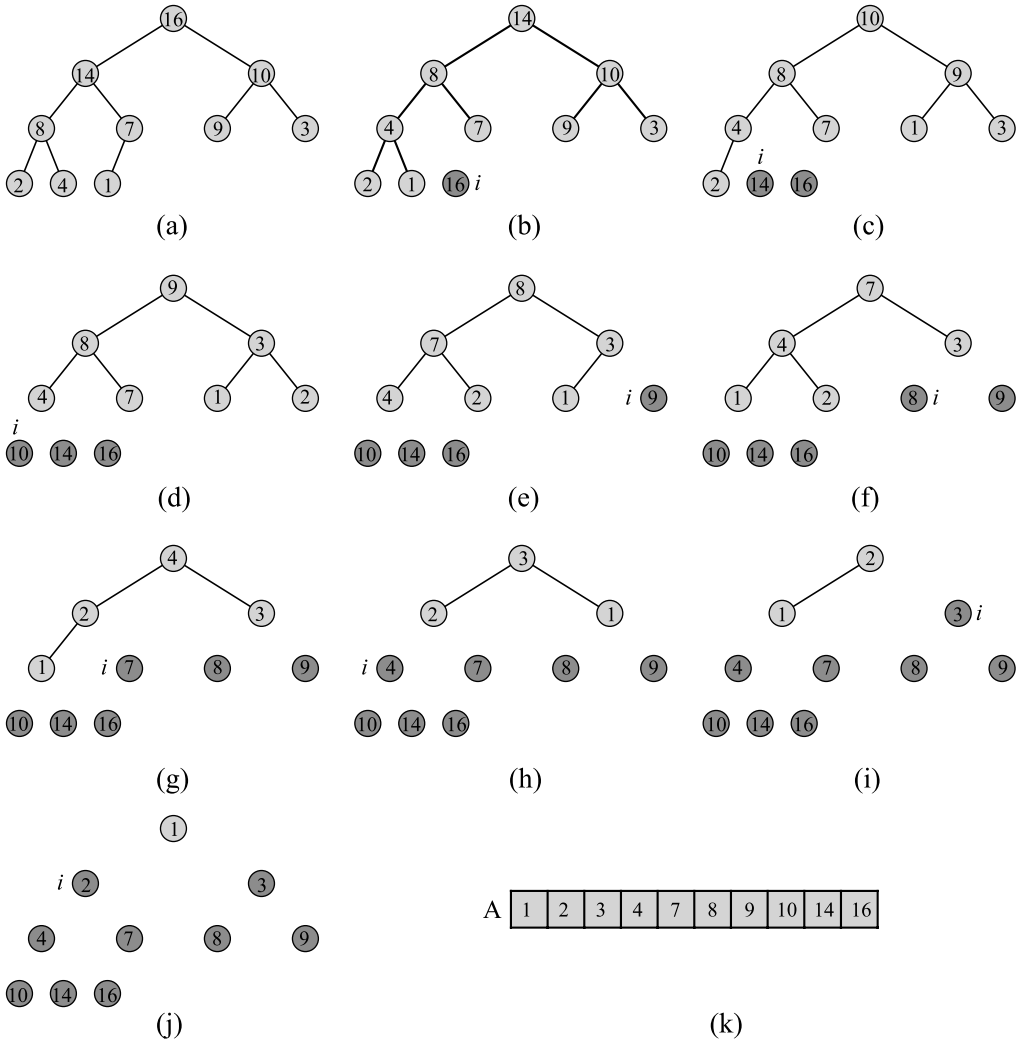

Figura 7.4 Modul de funcţionare a algoritmului HEAPSORT. **(a)** Structura de date heap, imediat după construirea sa de către procedura CONSTRUIEŞTE-HEAP. **(b)-(j)** Heap-ul, imediat după câte un apel al procedurii RECONSTITUIE-HEAP (linia 5 în algoritm). Figura reprezintă valoarea curentă a variabilei i. Din heap fac parte doar nodurile din cercurile nehaşurate. **(k)** Vectorul A ordonat, obţinut ca rezultat.

procedurile MAXIM şi EXTRAGE-MAX se vor înlocui cu MINIM şi EXTRAGE-MIN. Programul de simulare va determina următorul eveniment cu ajutorul procedurii EXTRAGE-MIN, iar dacă va trebui introdus un nou element în şir, se va apela procedura INSERARE.

Rezultă în mod firesc faptul că o coadă de priorităţi poate fi implementată utilizând un heap. Operaţia MAXIM-HEAP va determina în timpul $\Theta(1)$ cel mai mare element al heap-ului care, de fapt, este valoarea $A[1]$. Procedura EXTRAGE-MAX-DIN-HEAP este similară structurii repetitive **pentru** (liniile 3–5) din procedura HEAPSORT:

EXTRAGE-MAX-DIN-HEAP(A)
1: **dacă** *dimesiune-heap*$[A] < 1$ **atunci**
2: **eroare** "depăşire inferioară heap"
3: $max \leftarrow A[1]$
4: $A[1] \leftarrow A[$*dimesiune-heap*$[A]]$
5: *dimesiune-heap*$[A] \leftarrow$ *dimesiune-heap*$[A] - 1$
6: RECONSTITUIE-HEAP($A, 1$)
7: **returnează** *maxim*

Timpul de execuţie al procedurii EXTRAGE-MAX-DIN-HEAP este $O(\lg n)$, deoarece conţine doar câţiva paşi, care se execută în timp constant înainte să se execute procedura RECONSTITUIE-HEAP, care necesită un timp de $O(\lg n)$.

Procedura INSEREAZĂ-ÎN-HEAP inserează un nod în heap-ul A. La prima expandare a heap-ului se adaugă o frunză arborelui. Apoi, la fel ca în structura repetitivă de inserare (liniile 5–7) din SORTARE-PRIN-INSERARE (din secţiunea 1.1), se traversează un drum pornind de la această frunză către rădăcină, în scopul găsirii locului definitiv al noului element.

INSEREAZĂ-ÎN-HEAP(A, *cheie*)
1: *dimesiune-heap*$[A] \leftarrow$ *dimesiune-heap*$[A] + 1$
2: $i \leftarrow$ *dimesiune-heap*$[A]$
3: **cât timp** $i > 1$ şi $A[$PĂRINTE$(i)] <$ *cheie* **execută**
4: $A[i] \leftarrow A[$PĂRINTE$(i)]$
5: $i \leftarrow$ PĂRINTE(i)
6: $A[i] \leftarrow$ *cheie*

În figura 7.5 este ilustrat un exemplu al operaţiei INSEREAZĂ-ÎN-HEAP. Timpul de execuţie al procedurii INSEREAZĂ-ÎN-HEAP, pentru un heap având n elemente, este $O(\lg n)$, deoarece drumul parcurs de la noua frunză către rădăcină are lungimea $O(\lg n)$.

În concluzie, pe un heap se poate efectua orice operaţie specifică unei cozi de priorităţi, definită pe o mulţime având n elemente, într-un timp $O(\lg n)$.

Exerciţii

7.5-1 Utilizând ca model figura 7.5, ilustraţi operaţia INSEREAZĂ-ÎN-HEAP($A, 10$) pentru heap-ul $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$.

7.5-2 Dat fiind heap-ul $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$, ilustraţi operaţia EXTRAGE-MAX-DIN-HEAP.

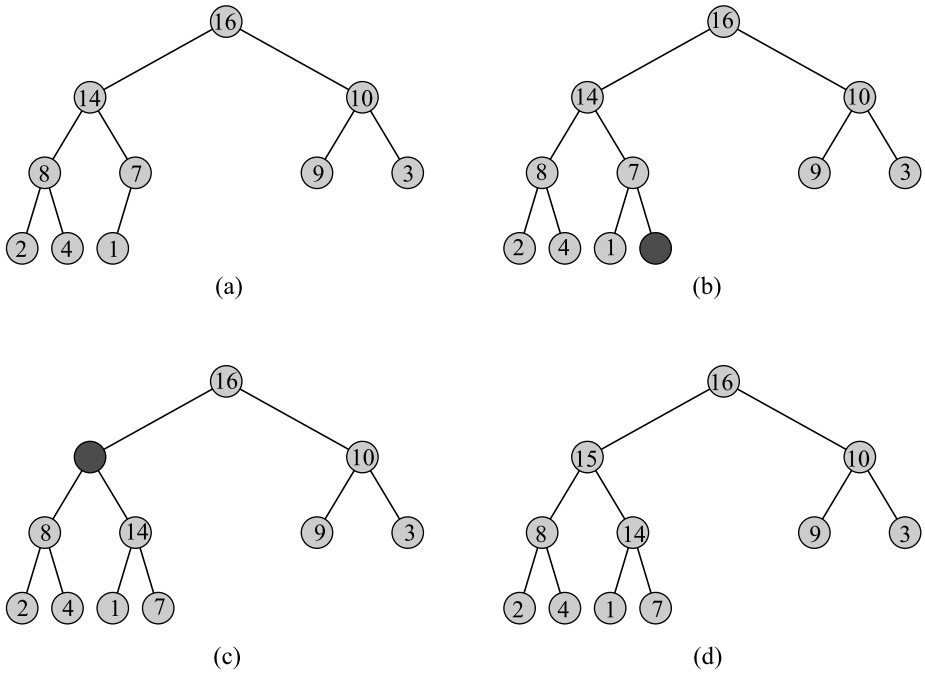

Figura 7.5 Operaţia INSEREAZĂ-ÎN-HEAP. **(a)** Heap-ul din figura 7.4(a) înainte de inserarea nodului având cheia 15. **(b)** Se adaugă arborelui o frunză nouă. **(c)** Se "scufundă" valorile de pe drumul dintre frunză şi rădăcină până la găsirea nodului corespunzător cheii 15. **(d)** Se inserează cheia 15.

7.5-3 Arătaţi cum se poate implementa o listă de tip FIFO cu ajutorul unei cozi de priorităţi. Arătaţi cum se poate implementa o stivă cu ajutorul unei cozi de priorităţi. (Listele FIFO şi stivele vor fi definite în secţiunea 11.1.)

7.5-4 Scrieţi o implementare a procedurii HEAP-CU-CHEI-CRESCĂTOARE(A, i, k) de timp $O(\lg n)$, care realizează atribuirea $A[i] \leftarrow \max(A[i], k)$, şi actualizează structura heap în mod corect.

7.5-5 Operaţia ŞTERGE-DIN-HEAP(A, i) şterge elementul ataşat nodului i din heap-ul A. Găsiţi o implementare pentru operaţia ŞTERGE-DIN-HEAP care se execută pentru un heap având n elemente într-un timp $O(\lg n)$.

7.5-6 Găsiţi un algoritm de timp $O(n \lg k)$ pentru a interclasa k liste ordonate, unde n este numărul total de elemente din listele de intrare. (*Indicaţie:* se utilizează un heap)

Probleme

7-1 Construirea unui heap prin inserare
Procedura CONSTRUIEŞTE-HEAP, din secţiunea 7.3, poate fi implementată folosind, în mod repetat, procedura INSERARE-ÎN-HEAP, în scopul inserării elementelor în heap. Fie următoarea implementare:

CONSTRUIEŞTE-HEAP$'(A)$
1: $dimesiune\text{-}heap[A] \leftarrow 1$
2: **pentru** $i \leftarrow 2$, $lungime[A]$ **execută**
3: INSEREAZĂ-ÎN-HEAP$(A, A[i])$

a. Pentru date de intrare identice, procedurile CONSTRUIEŞTE-HEAP şi CONSTRUIEŞTE-HEAP$'$ construiesc acelaşi heap? Demonstraţi sau găsiţi contraexemple.

b. Arătaţi că, în cazul cel mai defavorabil, procedura CONSTRUIEŞTE-HEAP$'$ cere un timp $\Theta(n \lg n)$ pentru a construi un heap având n elemente.

7-2 Analiza heap-urilor d-are
Un ***heap d-ar*** este asemănător unui heap binar, dar nodurile nu au doi ci d descendenţi.

a. Cum se poate reprezenta un heap d-ar sub forma unui vector?

b. Care este înălţimea (în funcţie de n şi d) a unui heap d-ar, având n elemente?

c. Descrieţi o implementare eficientă pentru operaţia EXTRAGE-MAX. Analizaţi timpul de execuţie în funcţie de n şi d.

d. Descrieţi o implementare eficientă pentru operaţia INSERARE. Analizaţi timpul de execuţie în funcţie de n şi d.

e. Descrieţi o implementare eficientă pentru operaţia HEAP-CU-CHEI-CRESCĂTOARE(A, i, k) care realizează $A[i] \leftarrow \max(A[i], k)$ şi actualizează structura heap în mod corect. Analizaţi timpul de execuţie în funcţie de d şi n.

Note bibliografice

Algoritmul heapsort a fost inventat de Williams [202], care a descris şi modul de implementare a unei cozi de priorităţi printr-un heap. Procedura CONSTRUIEŞTE-HEAP a fost propusă de Floyd [69].