11 Structuri de date elementare

În acest capitol vom examina reprezentarea mulţimilor dinamice ca structuri de date simple care utilizează referinţe. Deşi multe structuri de date complexe pot fi modelate folosind pointeri, le vom prezenta doar pe cele elementare: stive, cozi, liste înlănţuite şi arbori cu rădăcină. De asemenea, vom prezenta o metodă prin care obiectele şi pointerii pot fi sintetizaţi din tablouri.

11.1. Stive şi cozi

Stivele şi cozile sunt mulţimi dinamice în care elementul care este eliminat din mulţime de către operaţia ŞTERGE este prespecificat. Într-o ***stivă***, elementul şters din mulţime este elementul cel mai recent inserat: stiva implementează principiul ***ultimul sosit, primul servit*** (last-in, first-out) sau ***LIFO***. Similar, într-o ***coadă***, elementul şters este întotdeauna cel care a stat în mulţime cel mai mult (primul introdus): coada implementează principiul ***primul sosit, primul servit*** (first-in, first-out) sau ***FIFO***. Există mai multe modalităţi eficiente de a implementa stivele şi cozile cu ajutorul calculatorului. În această secţiune vom arăta cum se pot folosi tablourile simple pentru a implementa fiecare dintre aceste structuri.

Stive

Operaţia INSEREAZĂ asupra stivelor este deseori denumită PUNE-ÎN-STIVĂ, iar operaţia ŞTERGE, care nu cere nici un argument, este deseori numită SCOATE-DIN-STIVĂ. Aceste nume sunt aluzii la stivele fizice, ca de exemplu un vraf de farfurii. Ordinea în care sunt luate farfuriile din vraf este ordinea inversă în care au fost introduse în vraf, deoarece doar ultima farfurie este accesibilă.

Aşa cum se vede în figura 11.1, o stivă cu cel mult n elemente poate fi implementată printr-un tablou $S[1..n]$. Tabloul are un atribut $vârf[S]$ care este indicele celui mai recent introdus element. Stiva constă din elementele $S[1..vârf[S]]$, unde $S[1]$ este elementul de la baza stivei, iar $S[vârf[S]]$ este elementul din vârful stivei.

Când $vârf[S] = 0$, stiva nu conţine nici un element şi deci este ***vidă***. Se poate testa dacă stiva este vidă prin operaţia de interogare STIVĂ-VIDĂ.

Dacă se încearcă extragerea (se apelează operaţia SCOATE-DIN-STIVĂ) unui element dintr-o stivă vidă, spunem că stiva are ***depăşire inferioară***, care în mod normal este o eroare. Dacă $vârf[S]$ depăşeşte valoarea n, stiva are ***depăşire superioară***. (În implementarea noastră în pseudocod, nu ne vom pune problema depăşirii stivei.)

Fiecare dintre operaţiile stivei pot fi implementate prin doar câteva linii de cod.

STIVĂ-VIDĂ(S)
1: **dacă** $vârf[S] = 0$ **atunci**
2: **returnează** ADEVĂRAT
3: **altfel**
4: **returnează** FALS

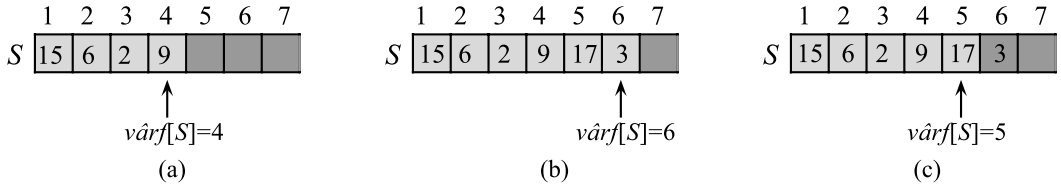

Figura 11.1 Implementarea unei stive S printr-un tablou. Elementele stivei apar doar în poziţiile haşurate folosind o culoare deschisă. **(a)** Stiva S are 4 elemente. Elementul din vârful stivei este 9. **(b)** Stiva S după apelurile PUNE-ÎN-STIVĂ (S,17) şi PUNE-ÎN-STIVĂ(S,3). **(c)** Stiva S după ce apelul SCOATE-DIN-STIVĂ (S) a întors ca rezultat elementul 3, care este cel mai recent introdus. Deşi elementul 3 apare în continuare în tabel, el nu mai este în stivă; elementul din vârful stivei este 17.

PUNE-ÎN-STIVĂ(S, x)
1: *vârf*[S] ← *vârf*[S] + 1
2: *S[vârf*[S]] ← x

SCOATE-DIN-STIVĂ(S)
1: **dacă** STIVĂ-VIDĂ(S) **atunci**
2: **eroare** "depăşire inferioară"
3: **altfel**
4: *vârf*[S] ← *vârf*[S] - 1
5: **returnează** S[*vârf*[S]+1]

Figura 11.1 ilustrează modificările produse în urma apelurilor operaţiilor PUNE-ÎN-STIVĂ şi SCOATE-DIN-STIVĂ. Fiecare dintre cele trei operaţii asupra stivelor necesită un timp $O(1)$.

Cozi

Vom numi PUNE-ÎN-COADĂ operaţia INSEREAZĂ aplicată asupra unei cozi, iar operaţia ŞTERGE o vom numi SCOATE-DIN-COADĂ; asemănător operaţiei SCOATE-DIN-STIVĂ aplicată stivei, SCOATE-DIN-COADĂ nu necesită nici un argument. Principiul FIFO, propriu cozii, impune ca aceasta să opereze asemănător cu un rând de oameni ce aşteaptă la un ghişeu. Coada are un ***cap*** şi o ***coadă***. Când un element este pus în coadă, ocupă locul de la sfârşitul cozii, ca şi un nou venit ce îşi ia locul la coada rândului. Elementul scos din coadă este întotdeauna cel din capul cozii, asemănător persoanei din capul rândului care a aşteptat cel mai mult. (Din fericire, nu trebuie să ne îngrijorăm din cauza elementelor care "se bagă în faţă".)

Figura 11.2 ilustrează o modalitate de a implementa o coadă având cel mult $n-1$ elemente, folosind un tablou $Q[1..n]$. Coada are un atribut $cap[Q]$ care conţine indicele capului ei. Atributul $coadă[Q]$ conţine indicele noii locaţii în care se va insera în coadă elementul nou venit. Elementele din coadă se află în locaţiile $cap[Q], cap[Q]+1, ..., coadă[Q]-1$, iar indicii se parcurg circular, în sensul că locaţia 1 urmează imediat după locaţia n. Când $cap[Q] = coadă[Q]$ coada este goală. Iniţial avem $cap[Q] = coadă[Q] = 1$. Când coada este vidă, o încercare de a scoate un element din coadă cauzează o depăşire inferioară în coadă. Când $cap[Q] = coadă[Q] + 1$, coada este "plină" şi o încercare de a pune în coadă cauzează o depăşire superioară în coadă.

În PUNE-ÎN-COADĂ şi SCOATE-DIN-COADĂ, verificarea erorilor de depăşire inferioară a fost omisă. (Exerciţiul 11.1-4 cere scrierea codului ce verifică aceste două condiţii de eroare.)

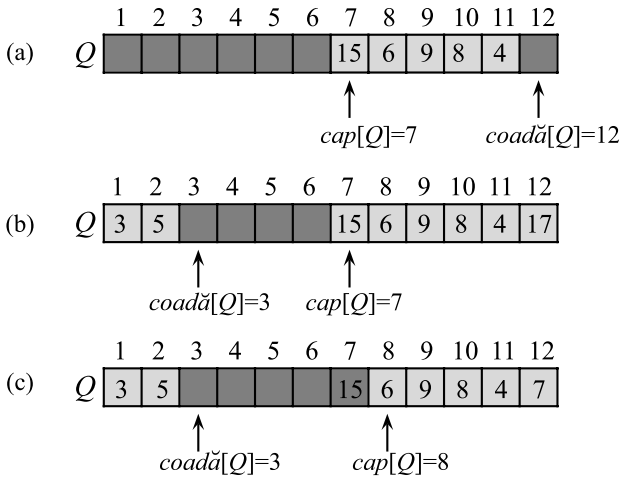

Figura 11.2 O coadă implementată utilizând un tablou $Q[1..12]$. Elementele cozii apar doar în poziţiile cu haşură deschisă. **(a)** Coada are 5 elemente, în locaţiile $Q[7..11]$. **(b)** Configuraţia cozii după apelurile PUNE-ÎN-COADĂ (Q,17), PUNE-ÎN-COADĂ(Q,3), PUNE-ÎN-COADĂ(Q,5). **(c)** Configuraţia cozii după ce apelul SCOATE-DIN-COADĂ(Q) returnează valoarea cheii 15, ce s-a aflat anterior în capul cozii. Noul cap are cheia 6.

PUNE-ÎN-COADĂ(Q, x)
1: $Q[coadă[Q]] \leftarrow x$
2: **dacă** $coadă[Q] = lung[Q]$ **atunci**
3: $\quad coadă[Q] \leftarrow 1$
4: **altfel**
5: $\quad coadă[Q] \leftarrow coadă[Q] + 1$

SCOATE-DIN-COADĂ(Q)
1: $x \leftarrow Q[cap[Q]]$
2: **dacă** $cap[Q] = lung[Q]$ **atunci**
3: $\quad cap[Q] \leftarrow 1$
4: **altfel**
5: $\quad cap[Q] \leftarrow cap[Q] + 1$
6: **returnează** x

În figura 11.2 sunt ilustrate efectele operaţiilor PUNE-ÎN-COADĂ şi SCOATE-DIN-COADĂ. Fiecare operaţie necesită un timp $O(1)$.

Exerciţii

11.1-1 Folosind ca model figura 11.1, ilustraţi efectul fiecăreia dintre operaţiile PUNE-ÎN-STIVĂ$(S, 4)$, PUNE-ÎN-STIVĂ$(S, 1)$, PUNE-ÎN-STIVĂ$(S, 3)$, SCOATE-DIN-STIVĂ(S), PUNE-ÎN-STIVĂ$(S, 8)$, SCOATE-DIN-STIVĂ(S) pe o stivă iniţial vidă, memorată într-un tablou $S[1..6]$.

11.1-2 Explicaţi cum se pot implementa două stive într-un singur tablou $A[1..n]$ astfel încât nici una dintre stive să nu aibă depăşire superioară, cu excepţia cazului în care numărul total de elemente din ambele stive (împreună) este n. Operaţiile PUNE-ÎN-STIVĂ şi SCOATE-DIN-STIVĂ trebuie să funcţioneze într-un timp $O(1)$.

11.1-3 Folosind ca model figura 11.2, ilustraţi efectul fiecăreia dintre operaţiile PUNE-ÎN-COADĂ$(Q, 4)$, PUNE-ÎN-COADĂ$(Q, 1)$, PUNE-ÎN-COADĂ$(Q, 3)$, SCOATE-DIN-COADĂ(Q), PUNE-ÎN-COADĂ$(Q, 8)$ şi SCOATE-DIN-COADĂ (Q) pe o coadă Q, iniţial vidă, a memorată într-un tablou $Q[1..6]$.

11.1-4 Rescrieţi PUNE-ÎN-COADĂ şi SCOATE-DIN-COADĂ pentru a detecta depăşirile unei cozi.

11.1-5 În timp ce stiva permite inserarea şi ştergerea de elemente doar la un singur capăt, iar coada permite inserarea la un capăt şi ştergerea la celălalt capăt, o ***coadă completă*** permite inserări şi ştergeri la ambele capete. Scrieţi patru proceduri cu timpul de execuţie $O(1)$ pentru inserare de elemente şi ştergere de elemente la ambele capete ale unei cozi complete implementată printr-un tablou.

11.1-6 Arătaţi cum se poate implementa o coadă prin două stive. Analizaţi timpul de execuţie pentru operaţiile cozii.

11.1-7 Arătaţi cum se poate implementa o stivă prin două cozi. Analizaţi timpul de execuţie pentru operaţiile stivei.

11.2. Liste înlănţuite

O ***listă înlănţuită*** este o structură de date în care obiectele sunt aranjate într-o anumită ordine. Spre deosebire de tablou, în care ordinea este determinată de indicii tabloului, ordinea într-o listă înlănţuită este determinată de un pointer conţinut în fiecare obiect. Listele înlănţuite asigură o reprezentare mai simplă şi mai flexibilă pentru mulţimi dinamice, suportând (deşi nu neapărat eficient) toate operaţiile descrise la pagina 168.

După cum se arată în figura 11.3, fiecare element al unei ***liste dublu înlănţuite*** L este un obiect cu un câmp *cheie* şi alte două câmpuri pointer: *urm* (următor) şi *prec* (precedent). Obiectul poate conţine şi alte date. Fiind dat un element x din listă, $urm[x]$ indică spre succesorul său din lista înlănţuită, iar $prec[x]$ indică spre predecesorul său. Dacă $prec[x] =$ NIL atunci elementul x nu are nici un predecesor şi deci e primul element din listă, sau ***capul*** listei. Dacă $urm[x] =$NIL atunci elementul x nu are succesor şi deci este ultimul element sau ***coada*** listei. Un atribut $cap[L]$ referă primul element al listei. Dacă $cap[L] =$NIL, lista este vidă.

O listă poate avea una din următoarele forme. Ea poate fi simplu înlănţuită sau dublu înlănţuită, poate fi sortată sau nu, şi poate fi, sau nu, circulară. Dacă o listă este ***simplu înlănţuită*** vom omite pointerul *prec* din fiecare element. Dacă o listă este ***sortată***, ordinea din listă corespunde ordinii cheilor memorate în elementele listei; elementul minim este capul listei, iar elementul maxim este coada listei. Dacă lista este ***nesortată***, elementele pot apărea în orice ordine. Într-o ***listă circulară*** referinţa *prec* a capului listei referă coada iar referinţa *urm* a cozii listei referă capul ei. Astfel, lista poate fi privită ca un inel de elemente. În restul acestei secţiuni vom presupune că listele cu care lucrăm sunt nesortate şi dublu înlănţuite.

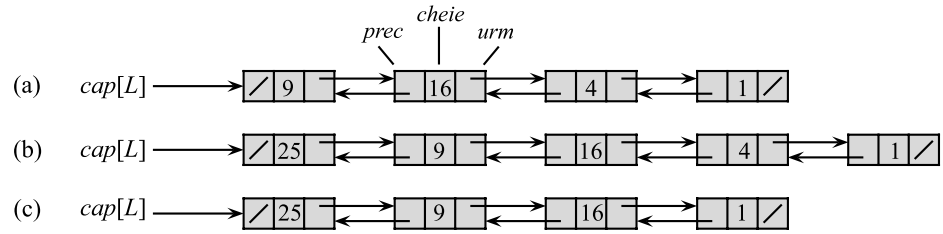

Figura 11.3 **(a)** O listă dublu înlănţuită L reprezentând mulţimea dinamică $\{1, 4, 9, 16\}$. Fiecare element din listă este un obiect având câmpuri pentru cheie şi doi pointeri (ilustraţi prin săgeţi) la elementul următor, respectiv anterior. Câmpul *urm* al cozii şi câmpul *prec* al capului sunt NIL, lucru indicat printr-un slash. Atributul $cap[L]$ referă capul listei. **(b)** După execuţia lui LISTĂ-INSEREAZĂ(L, x), unde $cheie[x] = 25$, capul listei înlănţuite este un nou obiect având cheia 25. Acest nou obiect referă vechiul cap având cheia 9. **(c)** Rezultatul apelului LISTĂ-ŞTERGE(L, x), unde x referă obiectul având cheia 4.

Căutarea într-o listă înlănţuită

Procedura LISTĂ-CAUTĂ(L, k) găseşte primul element având cheia k din lista L printr-o căutare liniară simplă, returnând pointerul la acest element. Dacă în listă nu apare nici un obiect având cheia k, atunci se returnează NIL. Pentru lista înlănţuită din figura 11.3(a), apelul LISTĂ-CAUTĂ$(L, 4)$ returnează un pointer la al treilea element, iar apelul LISTĂ-CAUTĂ$(L, 7)$ returnează NIL.

LISTĂ-CAUTĂ(L, x)
1: $x \leftarrow cap[L]$
2: **cât timp** $x \neq$ NIL şi $cheie[x] \neq k$ **execută**
3: $\quad x \leftarrow urm[x]$
4: **returnează** x

Pentru a căuta într-o listă având n obiecte, procedura LISTĂ-CAUTĂ necesită un timp $\Theta(n)$ în cazul cel mai defavorabil, deoarece va trebui să caute în toată lista.

Inserarea într-o listă înlănţuită

Fiind dat un element x al cărui câmp *cheie* a fost iniţializat, procedura LISTĂ-INSEREAZĂ îl plasează pe x în faţa listei înlănţuite, după cum se poate observa în figura 11.3(b).

LISTĂ-INSEREAZĂ(L, x)
1: $urm[x] \leftarrow cap[L]$
2: **dacă** $cap[L] \neq$ NIL **atunci**
3: $\quad prec[cap[L]] \leftarrow x$
4: $cap[L] \leftarrow x$
5: $prec[x] \leftarrow$ NIL

Timpul de execuţie pentru LISTĂ-INSEREAZĂ pe o listă cu n elemente este $O(1)$.

Ştergerea dintr-o listă înlănţuită

Procedura LISTĂ-ŞTERGE elimină un element x dintr-o listă înlănţuită L. Pentru aceasta, trebuie transmis ca argument un pointer spre x şi x va fi scos din listă, actualizându-se pointerii. Dacă dorim să ştergem un element cu o cheie dată, mai întâi trebuie să apelăm LISTĂ-CAUTĂ pentru a afla pointerul spre acel element.

```
LISTĂ-ŞTERGE(L, x)
1: dacă prec[x] ≠ NIL atunci
2:     urm[prec[x]] ← urm[x]
3: altfel
4:     cap[L] ← urm[x]
5: dacă urm[x] ≠ NIL atunci
6:     prec[urm[x]] ← prec[x]
```

Figura 11.3(c) ilustrează cum se şterge un element dintr-o listă înlănţuită. LISTĂ-ŞTERGE se execută într-un timp $O(1)$, dar dacă dorim să ştergem un element având o cheie dată, timpul necesar pentru cazul cel mai defavorabil este $\Theta(n)$, pentru că înainte trebuie să apelăm LISTĂ-CAUTĂ.

Santinele

Codul pentru LISTĂ-ŞTERGE ar fi mai simplu dacă am ignora condiţiile extreme de la capul şi coada listei.

```
LISTĂ-ŞTERGE′(L, x)
1: urm[prec[x]] ← urm[x]
2: prec[urm[x]] ← prec[x]
```

O ***santinelă*** este un obiect fictiv care ne permite să simplificăm condiţiile de la extreme. De exemplu, să presupunem că în lista L avem la dispoziţie un obiect $nil[L]$, care reprezintă NIL, dar care are toate câmpurile unui element al listei. De câte ori avem o referinţă la NIL în codul listei, o vom înlocui cu o referinţă la santinela $nil[L]$. După cum se observă în figura 11.4, santinela transformă o listă dublu înlănţuită într-o listă circulară, cu santinela $nil[L]$ plasată între cap şi coadă; câmpul $urm[nil[L]]$ indică spre capul listei, iar $prec[nil[L]]$ indică spre coadă. Similar, atât câmpul urm al cozii cât şi câmpul $prec$ al capului indică spre $nil[L]$. Deoarece $urm[nil[L]]$ indică spre cap, putem elimina atributul $cap[L]$, înlocuind referinţele către el cu referinţe către $urm[nil[L]]$. O listă vidă va conţine doar santinela, deoarece atât $urm[nil[L]]$ cât şi $prec[nil[L]]$ pot fi setate pe $nil[L]$.

Codul pentru LISTĂ-CAUTĂ rămâne la fel ca înainte, dar cu referinţele spre NIL şi $cap[L]$ schimbate după cum s-a precizat anterior.

```
LISTĂ-CAUTĂ′(L, k)
1: x ← urm[nil[L]]
2: cât timp x ≠ nil[L] şi cheie[x] ≠ k execută
3:     x ← urm[x]
4: returnează x
```

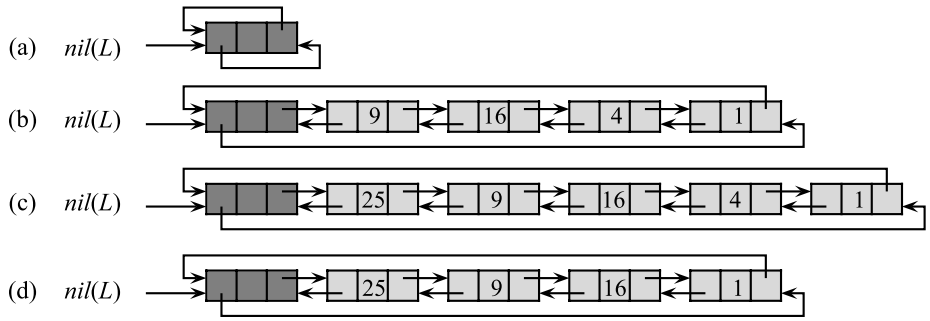

Figura 11.4 O listă înlănţuită L ce foloseşte o santinelă $nil[L]$ (haşurată cu negru) este o listă dublu înlănţuită obişnuită, transformată într-o listă circulară cu $nil[L]$ pus între cap şi coadă. Atributul $cap[L]$ nu mai este necesar, deoarece capul listei poate fi accesat prin $urm[nil[L]]$. **(a)** O listă vidă. **(b)** Lista înlănţuită din figura 11.3(a), având cheia 9 în capul listei şi cheia 1 în coada listei. **(c)** Lista după execuţia procedurii LISTĂ-INSEREAZĂ′(L, x), unde $cheie[x] = 25$. Noul obiect devine capul listei. **(d)** Lista după ştergerea obiectului având cheia 1. Noua coadă este obiectul având cheia 4.

Folosim procedura (cu două linii de cod) LISTĂ-ŞTERGE′ pentru a şterge un element din listă. Vom folosi următoarea procedură pentru inserarea unui element în listă.

LISTĂ-INSEREAZĂ′(L, x)
1: $urm[x] \leftarrow urm[nil[L]]$
2: $prec[urm[nil[L]]] \leftarrow x$
3: $urm[nil[L]] \leftarrow x$
4: $prec[x] \leftarrow nil[L]$

Figura 11.4 ilustrează efectele procedurilor LISTĂ-INSEREAZĂ′ şi LISTĂ-ŞTERGE′ pe un exemplu.

Rareori santinelele reduc marginile asimptotice de timp pentru operaţiile structurilor de date, dar pot reduce factorii constanţi. Câştigul pe care îl reprezintă folosirea santinelelor în cadrul ciclurilor este de obicei mai mult legat de claritatea codului, decât de viteză; de exemplu, codul pentru lista înlănţuită este simplificat prin utilizarea santinelelor, dar câştigăm doar un timp $O(1)$ în procedurile LISTĂ-INSEREAZĂ′ şi LISTĂ-ŞTERGE′. Oricum, în alte situaţii, folosirea santinelelor ajută la restrângerea codului dintr-un ciclu, reducându-se astfel coeficientul, de exemplu, pentru n sau n^2 din timpul de execuţie.

Santinelele trebuie folosite cu discernământ. Dacă avem multe liste mici, spaţiul suplimentar folosit de santinelele acestor liste poate reprezenta o risipă semnificativă de memorie. Astfel, în cartea de faţă santinelele se vor folosi doar acolo unde se obţine o simplificare semnificativă a codului.

Exerciţii

11.2-1 Se poate implementa operaţia INSEREAZĂ pentru mulţimi dinamice pe o listă simplu înlănţuită cu un timp de execuţie $O(1)$? Dar operaţia ŞTERGE?

11.2-2 Implementaţi o stivă folosind o listă simplu înlănţuită L. Operaţiile PUNE-ÎN-STIVĂ şi SCOATE-DIN-STIVĂ ar trebui să necesite tot un timp $O(1)$.

11.2-3 Implementaţi o coadă folosind o listă simplu înlănţuită L. Operaţiile PUNE-ÎN-COADĂ şi SCOATE-DIN-COADĂ ar trebui să necesite tot un timp $O(1)$.

11.2-4 Implementaţi operaţiile specifice dicţionarelor INSEREAZĂ, ŞTERGE şi CAUTĂ folosind liste simplu înlănţuite, respectiv circulare. Care sunt timpii de execuţie pentru aceste proceduri?

11.2-5 Operaţia pentru mulţimi dinamice REUNEŞTE are ca intrare două mulţimi disjuncte S_1 şi S_2 şi returnează mulţimea $S = S_1 \cup S_2$ ce conţine toate elementele din S_1 şi S_2. Mulţimile S_1 şi S_2 sunt de obicei distruse de către operaţie. Arătaţi cum se poate realiza REUNEŞTE într-un timp $O(1)$ folosind o structură de date de tip listă, adecvată.

11.2-6 Scrieţi o procedură care interclasează două liste simplu înlănţuite sortate într-o singură listă simplu înlănţuită sortată fără a folosi santinele. Apoi, scrieţi o procedură similară folosind o santinelă având cheia ∞ pentru a marca sfârşitul fiecărei liste. Comparaţi simplitatea codului pentru fiecare din cele două proceduri.

11.2-7 Creaţi o procedură nerecursivă de timp de execuţie $\Theta(n)$ care inversează o listă simplu înlănţuită având n elemente. Procedura va folosi un spaţiu suplimentar de memorie de mărime constantă.

11.2-8 $\star$ Explicaţi cum se pot implementa liste dublu înlănţuite folosind o singură valoare referinţă $np[x]$ pentru fiecare obiect, în locul celor două valori uzuale (*urm* şi *prec*). Se presupune că toate valorile pointerilor pot fi interpretate ca şi întregi pe k biţi şi se defineşte $np[x]$ ca fiind $np[x] = urm[x]$ XOR $prec[x]$, "sau exclusiv" pe k biţi dintre $urm[x]$ şi $prec[x]$. (Valoarea NIL e reprezentată de 0.) Acordaţi atenţie descrierii informaţiei necesare pentru accesarea capului listei. Arătaţi cum pot fi implementate operaţiile CAUTĂ, INSEREAZĂ şi ŞTERGE pe o astfel de listă. De asemenea, arătaţi cum se poate inversa o astfel de listă într-un timp $O(1)$.

11.3. Implementarea pointerilor şi obiectelor

Cum implementăm pointerii şi obiectele în limbaje cum ar fi Fortran, care nu asigură lucrul cu astfel de date? În această secţiune, vom arăta două modalităţi de implementare a structurilor de date înlănţuite fără un tip de date pointer explicit. Vom sintetiza obiectele şi pointerii din tablouri şi indici de tablouri.

O reprezentare multi-tablou a obiectelor

O colecţie de obiecte care au aceleaşi câmpuri se poate reprezenta utilizând câte un tablou pentru fiecare câmp. De exemplu, figura 11.5 ilustrează cum se poate implementa lista înlănţuită din figura 11.3(a) prin trei tablouri. Tabloul *cheie* reţine valorile curente ale cheilor din mulţimea dinamică, iar pointerii sunt memoraţi în tablourile *urm* şi *prec*. Pentru un indice de tablou x dat, $cheie[x], urm[x]$ şi $prec[x]$ reprezintă un obiect în lista înlănţuită. Într-o astfel de implementare, un pointer x este un simplu indice în tablourile *cheie, urm* şi *prec*.

În figura 11.3(a), obiectul având cheia 4 urmează după obiectul având cheia 16 în lista înlănţuită. În figura 11.5, cheia 4 apare în $cheie[2]$, iar cheia 16 apare în $cheie[5]$, deci vom avea $urm[5] = 2$ şi $prec[2] = 5$. Deşi constanta NIL apare în câmpul *urm* al cozii şi în câmpul

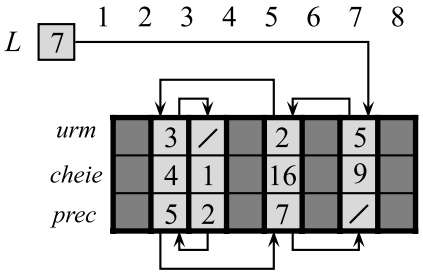

Figura 11.5 Lista înlănţuită din figura 11.3(a) reprezentată prin tablourile *cheie, urm* şi *prec*. Fiecare porţiune verticală a tablourilor reprezintă un singur obiect. Pointerii memoraţi corespund indicilor tablourilor ilustraţi în partea de sus; săgeţile arată cum trebuie interpretaţi aceşti indici. Poziţiile cu haşură deschisă ale obiectelor conţin elementele listei. Variabila L conţine indicele capului.

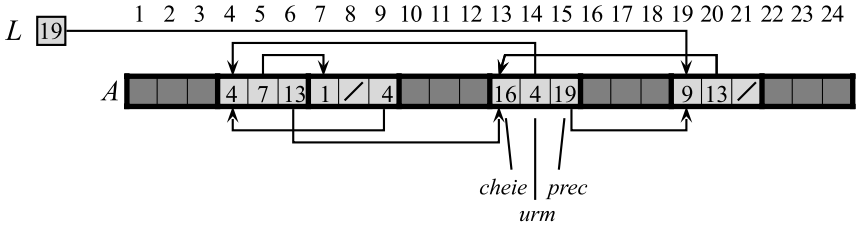

Figura 11.6 Lista înlănţuită din figura 11.3(a) şi figura 11.5 reprezentată printr-un singur tablou A. Fiecare element al listei este un obiect care ocupă un subtablou continuu de lungime 3 în cadrul tabloului. Cele trei câmpuri *cheie, urm* şi *prec* corespund deplasamentelor 0, 1 şi respectiv 2. Pointerul la un obiect este indicele primului element al obiectului. Obiectele conţinând elementele listei sunt haşurate deschis, iar săgeţile indică ordinea în listă.

prec al capului, de obicei vom folosi un întreg (ca 0 sau -1) care nu poate reprezenta un indice valid în tablouri. O variabilă *L* memorează indicele capului listei.

În pseudocodul nostru, am folosit paranteze pătrate pentru a nota atât indexarea într-un tablou cât şi selecţia unui câmp (atribut) aparţinând unui obiect. În ambele cazuri, semnificaţia lui $cheie[x], urm[x]$ şi $prec[x]$ este consistentă relativ la implementare.

O reprezentare a obiectelor printr-un tablou unic

Cuvintele din memoria unui calculator sunt în mod obişnuit adresate prin întregi între 0 şi $M-1$, unde M este un număr întreg suficient de mare. În multe limbaje de programare, un obiect ocupă o mulţime contiguă de locaţii în memoria calculatorului. Un pointer este pur şi simplu adresa primei locaţii de memorie a obiectului iar celelalte locaţii de memorie din cadrul obiectului pot fi indexate adăugând un deplasament la acest pointer.

În medii de programare care nu au tipuri de date pointer explicite putem folosi aceeaşi strategie pentru implementarea obiectelor. De exemplu, figura 11.6 ilustrează cum se poate folosi un singur tablou *A* pentru a memora lista înlănţuită din figura 11.3(a) şi figura 11.5. Un obiect ocupă un subtablou contiguu $A[j..k]$. Fiecare câmp al obiectului corespunde unui deplasament din intervalul $0..k-j$, iar un pointer la obiect este indicele *j*. În figura 11.6, deplasamentele

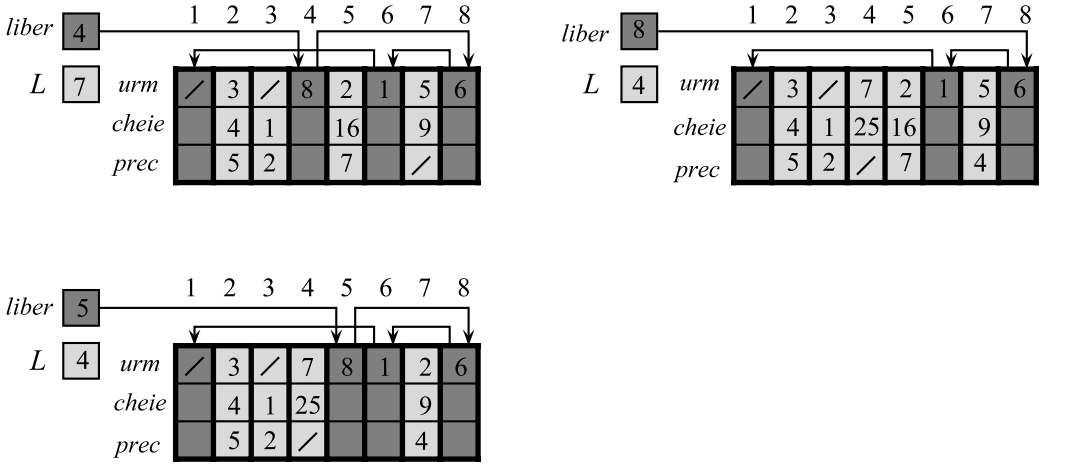

Figura 11.7 Efectul procedurilor ALOCĂ-OBIECT şi ELIBEREAZĂ-OBIECT. **(a)** Lista din figura 11.5 (partea cu haşură deschisă) şi o listă liberă (partea cu haşură închisă). Săgeţile indică structura listei libere. **(b)** Rezultatul apelului ALOCĂ-OBIECT() (care returnează cheia 4), setând $cheie[4]$ la 25 şi al apelului LISTĂ-INSEREAZĂ$(L, 4)$. Noul cap al listei libere este obiectul 8, care a fost $urm[4]$ în lista liberă. **(c)** După execuţia lui LISTĂ-ŞTERGE$(L, 5)$, apelăm ELIBEREAZĂ-OBIECT(5). Obiectul 5 devine noul cap al listei libere, cu obiectul 8 urmându-i în listă.

corespunzătoare lui *cheie*, *urm* şi *prec* sunt 0, 1, respectiv 2. Pentru a citi valoarea lui $prec[i]$ pentru un pointer i dat, adăugăm la valoarea i a pointerului deplasamentul 2, citind astfel $A[i+2]$.

Reprezentarea printr-un singur tablou este flexibilă în sensul că permite memorarea de obiecte cu lungimi diferite în acelaşi tablou. Problema gestionării unei astfel de colecţii eterogene de obiecte este mult mai dificilă decât problema gestionării unei colecţii omogene, în care toate obiectele au aceleaşi câmpuri. Deoarece majoritatea structurilor de date pe care le luăm în considerare sunt compuse din elemente omogene, pentru scopurile noastre va fi suficient să folosim reprezentarea multi-tablou a obiectelor.

Alocarea şi eliberarea obiectelor

Pentru a insera o cheie într-o mulţime dinamică reprezentată printr-o listă dublu înlănţuită, trebuie să alocăm un pointer la un obiect neutilizat la momentul respectiv în reprezentarea listei înlănţuite. Prin urmare, este util să ţinem evidenţa obiectelor care nu sunt folosite la un moment dat, în reprezentarea listei înlănţuite; astfel, ori de câte ori avem nevoie de un nou obiect, acesta va fi preluat din lista de obiecte nefolosite. În unele sisteme, un aşa numit "***garbage collector***" (colector de reziduuri) este responsabil cu determinarea obiectelor neutilizate. Oricum, sunt multe aplicaţii, suficient de simple astfel încât îşi pot asuma responsabilitatea de a returna un obiect neutilizat către gestionarul de obiecte nefolosite. Vom studia acum problema alocării şi eliberării (sau dealocării) obiectelor omogene folosind, de exemplu, o listă dublu înlănţuită reprezentată prin tablouri multiple.

Să presupunem că tablourile din reprezentarea multi-tablou au lungimea m şi că la un moment dat mulţimea dinamică conţine $n \leq m$ elemente. Atunci n obiecte vor reprezenta elementele

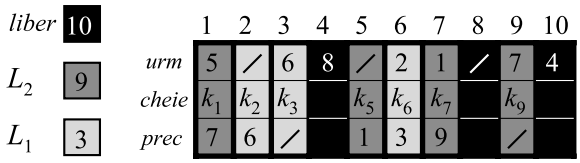

Figura 11.8 Două liste înlănţuite L_1 (porţiunea cu haşură deschisă) şi L_2 (cu haşură medie) şi o listă liberă interconectată (cu haşură închisă).

curente din mulţimea dinamică şi restul de $m - n$ obiecte sunt ***libere***; obiectele libere pot fi folosite pentru reprezentarea elementelor ce se vor insera în viitor în mulţimea dinamică.

Vom păstra obiectele libere într-o listă simplu înlănţuită, pe care o vom numi ***listă liberă***. Lista liberă foloseşte doar tabloul *urm*, care memorează pointerii *urm* din listă. Capul listei libere este memorat într-o variabilă globală *liber*. În cazul în care mulţimea dinamică reprezentată de lista înlănţuită L este nevidă, lista liberă se poate întreţese cu lista L, după cum se observă în figura 11.7. Observaţi că fiecare obiect din reprezentare este fie în lista L, fie în lista liberă, dar nu în amândouă.

Lista liberă este o stivă: următorul obiect alocat este cel mai recent eliberat. Putem folosi o implementare a operaţiilor stivei PUNE-ÎN-STIVĂ şi SCOATE-DIN-STIVĂ folosind o listă pentru a construi procedurile pentru alocarea şi, respectiv, eliberarea obiectelor. Presupunem că variabila globală *liber* folosită în următoarele proceduri referă primul element al listei libere.

ALOCĂ-OBIECT()
1: **dacă** *liber* = NIL **atunci**
2: **eroare** "depăşire de spaţiu"
3: **altfel**
4: $x \leftarrow liber$
5: $liber \leftarrow urm[x]$
6: **returnează** x

ELIBEREAZĂ-OBIECT(x)
1: $urm[x] \leftarrow liber$
2: $liber \leftarrow x$

Iniţial lista liberă conţine toate cele n obiecte nealocate. În cazul în care lista liberă este epuizată, procedura ALOCĂ-OBIECT semnalează o eroare. În mod curent se poate folosi o singură listă liberă care să servească mai multe liste înlănţuite. Figura 11.8 ilustrează două liste înlănţuite şi o listă liberă, interconectate prin tablourile *cheie*, *urm* şi *prec*.

Cele două proceduri se execută într-un timp $O(1)$, ceea ce le face eficiente. Ele pot fi modificate astfel încât să funcţioneze pentru orice colecţie omogenă de obiecte, lăsând unul din câmpurile obiectului să se comporte ca un câmp *urm* în lista liberă.

Exerciţii

11.3-1 Desenaţi o imagine a şirului $\langle 13, 4, 8, 19, 5, 11\rangle$ memorată ca o listă dublu înlănţuită folosind reprezentarea multi-tablou. Realizaţi acelaşi lucru pentru reprezentarea cu un tablou unic.

11.3-2 Scrieţi procedurile ALOCĂ-OBIECT şi ELIBEREAZĂ-OBIECT pentru o colecţie omogenă de obiecte implementată printr-o reprezentare cu un tablou unic.

11.3-3 De ce nu este necesar să setăm sau să resetăm câmpurile *prec* ale obiectelor din implementarea procedurilor ALOCĂ-OBIECT şi ELIBEREAZĂ-OBIECT?

11.3-4 De multe ori este de dorit să păstrăm toate elementele unei liste dublu înlănţuite într-o zonă compactă la memorare, folosind, de exemplu, primele m locaţii din reprezentarea multi-tablou. (Aceasta este situaţia într-un mediu de calcul cu memorie virtuală, paginată.) Explicaţi, cum se pot implementa procedurile ALOCĂ-OBIECT şi ELIBEREAZĂ-OBIECT astfel încât reprezentarea să fie compactă. Presupuneţi că nu există pointeri la elementele listei înlănţuite în afara listei înseşi. (*Indicaţie*: Folosiţi implementarea stivei printr-un tablou.)

11.3-5 Fie L o listă dublu înlănţuită de lungime m memorată în tablourile *cheie*, *prec* şi *urm* de lungime n. Să presupunem că aceste tablouri sunt gestionate de procedurile ALOCĂ-OBIECT şi ELIBEREAZĂ-OBIECT, care păstrează o listă liberă dublu înlănţuită F. Mai presupunem că din cele n locaţii, sunt exact m în lista L şi $n-m$ în lista liberă. Scrieţi o procedură COMPACTEAZĂ-LISTĂ(L, F) care, fiind date lista L şi lista liberă F, mută elementele din L astfel încât ele să ocupe poziţiile $1, 2, ..., m$ din tablou şi ajustează lista liberă F astfel încât să rămână corectă, ocupând poziţiile $m+1, m+2, ..., n$ din tablou. Timpul de execuţie pentru procedura scrisă trebuie să fie $\Theta(m)$ şi ea trebuie să utilizeze doar un spaţiu suplimentar constant. Daţi o argumentare atentă a corectitudinii procedurii scrise.

11.4. Reprezentarea arborilor cu rădăcină

Metodele pentru reprezentarea listelor date în secţiunea precedentă se extind la orice structură omogenă de date. În această secţiune, vom discuta în mod special problema reprezentării arborilor cu rădăcină prin structuri de date înlănţuite. Vom studia mai întâi arborii binari şi apoi vom prezenta o metodă pentru arbori cu rădăcină în care nodurile pot avea un număr arbitrar de descendenţi.

Vom reprezenta fiecare nod al arborilor printr-un obiect. Ca în cazul listelor înlănţuite, presupunem că fiecare nod conţine un câmp *cheie*. Restul câmpurilor care prezintă interes sunt pointeri către celelalte noduri şi pot varia în funcţie de tipul arborelui.

Arbori binari

După cum se observă în figura 11.9 câmpurile *p*, *stânga* şi *dreapta* se folosesc pentru a memora pointerii pentru părinte, descendentul stâng şi descendentul drept al fiecărui nod din arborele binar T. Dacă $p[x] =$ NIL, atunci x este rădăcina. Dacă nodul x nu are descendent stâng, atunci *stânga*$[x] =$ NIL şi similar pentru descendentul drept. Rădăcina întregului arbore T este referită de atributul *rădăcină*$[T]$. Dacă *rădăcină*$[T] =$ NIL, atunci arborele este vid.

Arbori cu rădăcină cu număr nelimitat de ramuri

Schema pentru reprezentarea arborilor binari poate fi extinsă la orice clasă de arbori în care numărul de descendenţi ai fiecărui nod nu depăşeşte o constantă k: vom înlocui câmpurile *stânga*

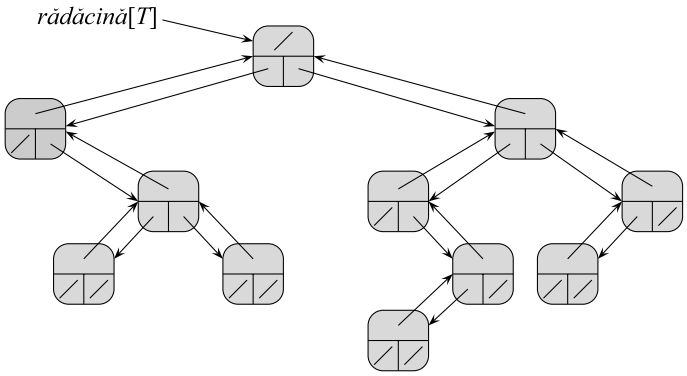

Figura 11.9 Reprezentarea unui arbore binar T. Fiecare nod x are câmpurile $p[x]$ (partea de sus), *stânga*$[x]$ (partea din stânga jos) şi *dreapta*$[x]$ (partea din dreapta jos). Câmpurile *cheie* nu sunt ilustrate.

şi *dreapta* cu $fiu_1, fiu_2, ..., fiu_k$. Această schemă nu mai funcţionează atunci când numărul de descendenţi ai unui nod este nemărginit, deoarece nu ştim câte câmpuri (tablouri în reprezentarea multi-tablou) să alocăm în avans. Mai mult, chiar dacă numărul k de descendenţi este mărginit de o constantă mare, dar majoritatea nodurilor au un număr mic de descendenţi, vom irosi multă memorie.

Din fericire, există o schemă "inteligentă" de a folosi arbori binari pentru a reprezenta arbori cu număr arbitrar de descendenţi. Ea are avantajul că utilizează un spaţiu de $O(n)$ pentru orice arbore cu rădăcină şi cu n noduri. ***Reprezentarea descendent-stâng, frate-drept*** este ilustrată în figura 11.10. Fiecare nod conţine un pointer spre părinte p, iar *rădăcină*$[T]$ referă rădăcina arborelui T. În loc să avem un pointer spre fiecare descendent, fiecare nod x are doar doi pointeri:

1. *fiu-stâng*$[x]$ referă cel mai din stânga descendent al nodului x şi
2. *frate-drept*$[x]$ referă fratele lui x, cel mai apropiat spre dreapta.

Dacă nodul x nu are descendenţi atunci *fiu-stâng*$[x] =$ NIL, iar dacă nodul x este cel mai din dreapta descendent al părintelui său atunci *frate-drept*$[x] =$ NIL.

Alte reprezentări ale arborilor

Uneori reprezentăm arborii cu rădăcină şi în alte moduri. În capitolul 7, de exemplu, am reprezentat un ansamblu, care se bazează pe un arbore binar complet, printr-un singur tablou plus un indice. Arborii care apar în capitolul 22 sunt traversaţi numai spre rădăcină, de aceea doar pointerii spre părinţi sunt prezenţi; nu există pointeri spre descendenţi. Sunt posibile şi multe alte scheme. Care este cea mai bună schemă care depinde de aplicaţie.

Exerciţii

11.4-1 Desenaţi arborele binar având rădăcina la indicele 6, care este reprezentat de următoarele câmpuri.

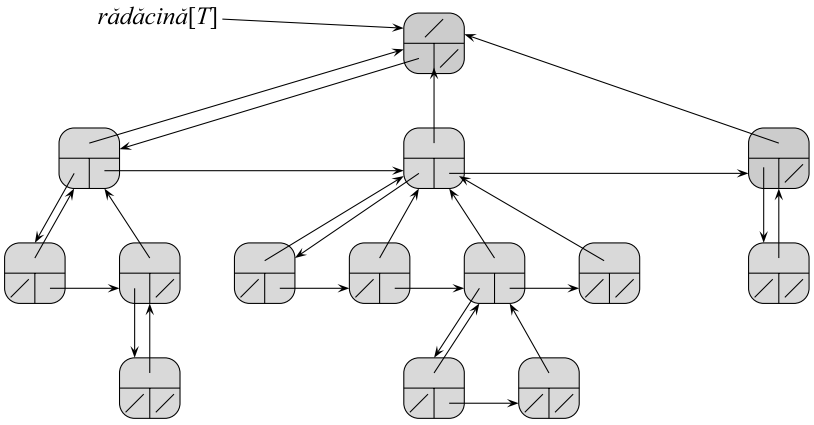

Figura 11.10 Reprezentarea descendent-stâng, frate-drept a unui arbore T. Fiecare nod x are câmpurile $p[x]$ (partea de sus), *fiu-stâng*$[x]$ (partea din stânga jos) şi *frate-drept*$[x]$ (partea din dreapta jos). Cheile nu sunt ilustrate.

indice	*cheie*	*stânga*	*dreapta*
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

11.4-2 Scrieţi o procedură recursivă cu timp de execuţie $O(n)$ care, fiind dat un arbore binar cu n noduri, tipăreşte cheia fiecărui nod din arbore.

11.4-3 Scrieţi o procedură nerecursivă cu timp de execuţie $O(n)$ care, fiind dat un arbore binar cu n noduri, tipăreşte cheia fiecărui nod din arbore. Folosiţi o stivă ca o structură auxiliară de date.

11.4-4 Scrieţi o procedură cu timp de execuţie $O(n)$ care tipăreşte toate cheile unui arbore arbitrar cu rădăcină având n noduri, dacă arborele este memorat folosind reprezentarea descendent-stâng, frate-drept.

11.4-5 $\star$ Scrieţi o procedură nerecursivă cu timp de execuţie $O(n)$ care, fiind dat un arbore binar cu n noduri, tipăreşte cheile fiecărui nod. Folosiţi numai un spaţiu suplimentar constant în afara arborelui însuşi şi nu modificaţi arborele, nici măcar temporar, în timpul procedurii.

11.4-6 $\star$ Reprezentarea descendent-stâng, frate-drept a unui arbore arbitrar cu rădăcină foloseşte trei pointeri în fiecare nod: *fiu-stâng*, *frate-drept* şi *părinte*. Din fiecare nod se pot atinge şi identifica părintele şi toţi descendenţii nodului. Arătaţi cum se poate obţine acelaşi efect, folosind doar doi pointeri şi o valoare booleană în fiecare nod.

Probleme

11-1 Comparaţii între liste

Pentru fiecare din cele patru tipuri de liste din tabelul următor, care este timpul asimptotic de execuţie în cazul cel mai defavorabil pentru fiecare dintre operaţiile pe mulţimi dinamice specificate în tabelul următor?

	nesortată, simplu înlănţuită	sortată, simplu înlănţuită	nesortată, dublu înlănţuită	sortată, dublu înlănţuită
CAUTĂ(L, k)				
INSEREAZĂ(L, x)				
ŞTERGE(L, x)				
SUCCESOR(L, x)				
PREDECESOR(L, x)				
MINIM(L)				
MAXIM(L)				

11-2 Absambluri interclasate folosind liste înlănţuite

Un ***ansamblu interclasat*** are următoarele operaţii: CREEAZĂ-ANSAMBLU (care creează un ansamblu interclasat vid), INSEREAZĂ, MINIM, EXTRAGE-MIN şi REUNEŞTE. Arătaţi cum se pot implementa ansamblurile interclasate folosind liste înlănţuite în fiecare din următoarele cazuri. Încercaţi să realizaţi fiecare operaţie cât mai eficient posibil. Analizaţi timpul de execuţie în termenii dimensiunii mulţimii (mulţimilor) dinamice cu care se operează. ***a.*** Listele sunt sortate. ***b.*** Listele nu sunt sortate. ***c.*** Listele nu sunt sortate, iar mulţimile dinamice ce se interclasează sunt disjuncte.

11-3 Căutarea într-o listă sortată compactă

Exerciţiul 11.3-4 cerea modul în care putem menţine o listă având n elemente, compactă în primele n poziţii ale unui tablou. Vom presupune că toate cheile sunt distincte şi că lista compactă este de asemenea sortată, adică, $cheie[i] < cheie[urm[i]]$, oricare ar fi $i = 1, 2, ..., n$ pentru care $urm[i] \neq$ NIL. Cu aceste presupuneri ne aşteptăm ca următorul algoritm aleator să efectueze căutarea în listă într-un timp $O(n)$.

LISTĂ-COMPACTĂ-CAUTĂ(L, k)
1: $i \leftarrow cap[L]$
2: $n \leftarrow lung[L]$
3: **cât timp** $i \neq$ NIL şi $cheie[i] \leq k$ **execută**
4: $j \leftarrow$ RANDOM$(1, n)$
5: **dacă** $cheie[i] < cheie[j]$ şi $cheie[j] < k$ **atunci**
6: $i \leftarrow j$
7: $i \leftarrow urm[i]$
8: **dacă** $cheie[i] = k$ **atunci**
9: **returnează** i
10: **returnează** NIL

Dacă ignorăm liniile 4–6 din procedură, atunci avem algoritmul uzual pentru căutarea într-o listă înlănţuită sortată, în care indicele i referă pe rând fiecare poziţie din listă. Liniile 4–6 încearcă să sară în avans la o poziţie j aleasă aleator. Un astfel de salt este benefic dacă $cheie[j]$ este mai mare decât $cheie[i]$ şi mai mică decât k; într-un astfel de caz j marchează o poziţie în listă prin care i va trebui să treacă în timpul unei căutări obişnuite în listă. Deoarece lista este compactă, ştim că orice alegere a lui j între 1 şi n referă un obiect din listă şi nu o poziţie în lista liberă.

a. De ce presupunem în LISTĂ-COMPACTĂ-CAUTĂ că toate cheile sunt distincte? Argumentaţi că salturile aleatoare nu ajută neapărat în cazul asimptotic, dacă lista conţine dubluri ale cheilor.

Performanţa procedurii LISTĂ-COMPACTĂ-CAUTĂ poate fi analizată împărţind execuţia sa în două faze. În timpul primei faze, ignorăm orice avans realizat în determinarea lui k, ce se realizează în liniile 7–9. Deci, faza 1 constă doar în avansul în listă prin salturi aleatoare. Asemănător, faza 2 ignoră avansul ce se realizează în liniile 4–6 şi astfel va opera ca o căutare liniară obişnuită.

Fie X_t variabila aleatoare care descrie distanţa în lista înlănţuită (şi anume, prin înlănţuirea pointerilor urm) de la poziţia i la cheia dorită k după t iteraţii ale fazei 1.

b. Argumentaţi că timpul de execuţie dorit pentru LISTĂ-COMPACTĂ-CAUTĂ este $O(t + E[X_t])$, pentru orice $t \geq 0$.

c. Arătaţi că $E[X_t] \leq \sum_{r=1}^{n}(1 - r/n)^t$. (*Indicaţie*: Folosiţi relaţia (6.28).)

d. Arătaţi că $\sum_{r=1}^{n-1} r^t \leq n^{t+1}/(t+1)$.

e. Demonstraţi că $E[X_t] \leq n/(t+1)$ şi explicaţi de ce această formulă este intuitivă.

f. Arătaţi că LISTĂ-COMPACTĂ-CAUTĂ se execută într-un timp de $O(\sqrt{n})$.

Note bibliografice

Aho, Hopcroft şi Ullman [5] şi Knuth [121] sunt referinţe excelente pentru structuri de date elementare. Gonnet [90] furnizează date experimentale asupra performanţelor operaţiilor pentru multe structuri de date.

Originea stivelor şi cozilor ca structuri de date în informatică este neclară, deoarece noţiunile corespunzătoare existau deja în matematică şi în prelucrarea tradiţională a documentelor înainte de introducerea calculatoarelor digitale. Knuth [121] îl citează pe A. M. Turing pentru folosirea stivelor la legarea subrutinelor în 1947.

De asemenea, structurile de date bazate pe pointeri par a fi o "invenţie din folclor". Conform lui Knuth, pointerii se pare că erau utilizaţi pe calculatoarele timpurii cu memorie pe tambur magnetic. Limbajul A-1, dezvoltat de G. M. Hopper în 1951 reprezenta formulele algebrice ca arbori binari. Knuth creditează limbajul IPL-II, dezvoltat în 1956 de A. Newell, J. C. Shaw şi H. A. Simon, cu recunoaşterea importanţei şi promovarea utilizării pointerilor. Limbajul lor IPL-III, dezvoltat în 1957, include operaţii explicite asupra stivelor.