Structuri de date

În acest curs vom vorbi despre reprezentarea mulțimilor dinamice, folosind structuri de date simple. Acestea sunt stiva, coada, lista înlănțuită.

Stiva și coada

Stiva și coada sunt mulțimi dinamice în care un element va fi adăugat sau șters prin intermediul unor operații specifice. Într-o stivă, elementul șters va fi acela care a fost introdus cel mai recent în structură. Astfel **stiva** este o structură de dată de tip *last-in first out*, sau **LIFO**. Pe de altă parte, coada este structura de date în care ștergerea are loc conform regulii: *first-in, first-out* sau **FIFO**.

Operatii pe structuri dinamice

Mai jos sunt prezentate tipurile de operații care pot fi efectuate pe mulțimi dinamice (cu număr extensibil de elemente). Acestea sunt sub forma unor proceduri, pentru a abstractiza mediul de programare folosit. Operațiile se împart în două tipuri: *interogare* (vor returna informații despre elementele din mulțime) și *modificare*(vor modifica elemente din mulțime).

1. Interogare
    a. SEARCH(*S*,*k*) – O interogare în care, fiind dată structura de date *S*, și o valoare *k*, se va returna o valoare *x*, care reprezintă un pointer la un element din *S* astfel ca *key[x]=k* sau NULL dacă valoarea *k* nu aparține lui *S*
    b. MINIMUM(*S*) – O interogare a întregii mulțimi *S* ce va returna elementul din *S* cu cea mai mică valoare
    c. MAXIMUM(*S*) – O interogare a întregii mulțimi *S* ce va returna elementul din *S* cu cea mai mare valoare
    d. SUCCESOR(*S,x*) – O interogare care, pentru un element dat *x*, a cărei cheie(valoare) aparține lui *S*, și care va returna următorul element din *S,* sau NULL dacă după *x* nu mai urmează nici un alt element.
    e. PREDECESOR(*S,x*) – O interogare care, pentru un element dat *x*, a cărei cheie(valoare) aparține lui *S*, și care va returna precedentul element din *S,* sau NULL dacă înaintea lui *x* nu mai există nici un alt element.

2. Modificare
    a. INSERT(*S*, x) – o operație ce mărește mulțimea *S* cu un element indicat de *x*. Presupunem că orice câmp din elementul *x* a fost anterior inițializat.
    b. DELETE(*S, x*) – o operație ce șterge elementul indicat de *x* din mulțimea *S.* Operația folosește un pointer la elementul *x*, și nu o valoare.

Stiva

Operația INSERT de mai sus se numește în cazul stivei și PUSH, iar DELETE (operație care nu ia ca argument pe *x*) se numește POP. Aceste nume sunt consacrate și provin de la ordinea in care farfuriile sunt aranjate în stive, în restaurante de exemplu.

Așa cum apare în figura 1, se poate implementa o stivă cu *n* elemente $S[1..n]$. Șirul are un atribut $top[S]$ care reprezintă elementul cel mai recent adăugat în stivă sau altfel spus, vârful stivei.

Stiva constă din elementele $S[1..top[S]]$ unde $S[1]$ este elementul de la baza stivei și $S[top[S]]$ este elementul din vârf.

Atunci când $top[S] = 0$ se zice că stiva este *goală*. Dacă se încearcă scoaterea unui element din stivă goală, spunem că stiva va avea un *underflow* adică o operație eronată.

Atunci când $top[S]$ depășește *n* – numărul de elemente din stivă, se cheamă că stiva are un *overflow*, de asemenea o operație eronată.

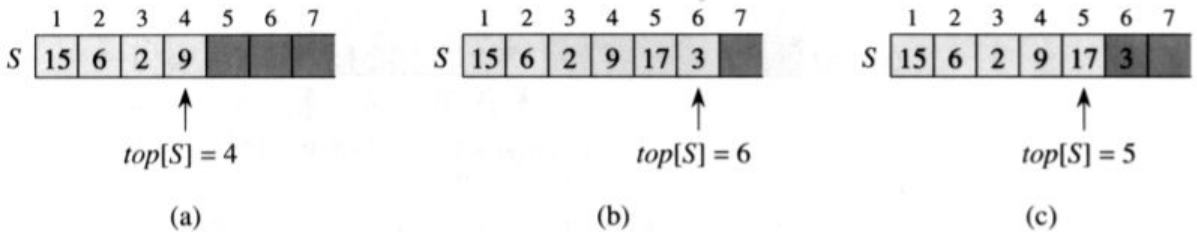


Figura 2. O implementare a unei stive S folosind un șir cu indicele pornind de la 1.

a) Stiva are 4 elemente. Vârful este 9. b) Stiva *S* după apelul PUSH(*S*,17) și PUSH(*S*,3). c) Stiva *S* după apelul POP(*S*) care a returnat valoarea 3. Deși 3 apare în imagine, el nu se mai află în stivă, vârful fiind acum elementul 17.

În continuare vom prezenta pseudocodul pentru operațiile descrise mai sus.

$PUSH(S, x)$

1. $top[S] \leftarrow top[S] + 1$
2. $S[top[S]] \leftarrow x$

$POP(S)$

1. $\boldsymbol{if}\ STACK - EMPTY(S)\ \boldsymbol{then}$
2. $\quad\boldsymbol{error}$ "$underflow$"
3. $\boldsymbol{else}\quad top[S] \leftarrow top[S] - 1$
4. $\boldsymbol{return}\ \ S[top[S] + 1]$

Procedura de verificare dacă stiva este sau nu goală este:

$STACK - EMPTY(S)$

1. $\boldsymbol{if}\ top[S] = 0\ \boldsymbol{then}$
2. $\quad\boldsymbol{return}\ TRUE$
3. $\boldsymbol{else}$
4. $\quad\boldsymbol{return}\ FALSE$

Mai jos vom exemplifica folosirea stivei ca structură de dată, pentru rezolvarea formei postfixate a unei expresii aritmetice.

Expresii aritmetice postfixate

Notația infixată a unei expresii este $3+4$ iar notația prefixată sau denumită și poloneză este $+\ 3\ 4$. Denumirea de forma poloneză provine de la matematicianul polonez Jan Lukasiewicz. De ce folosim această notație? În forma poloneză, operatorii sunt plasați după operanzi, după cum am arătat mai sus. Dacă sunt operații multiple, operatorul va urma după al doilea operand, așa încât expresia $3 - 4 + 5$ va fi scrisă $3\ 4 - 5\ +$marcând faptul că scădem 4 din 3 și apoi adunăm la suma nou obținută 5. Avantajul acestei notații este că elimină necesitatea parantezelor din notația infixată. De exemplu expresia $3 - 4 * 5$ poate fi scrisă ca $3 - (4 * 5)$, dar scrisă ca $(3 - 4) * 5$ are cu totul alt înțeles și

rezultat. În forma posfixată vom scrie expresia $3 - 4 * 5$ ca $3\ 4\ 5\ *\ -$ ce înseamnă clar, $3\ (\ 4\ 5\ *\ )\ -$ adică $3\ 20\ -$.

Cum funcționează? Pentru evaluarea unei expresii operanzii sunt plasați pe o stivă, iar în momentul efectuării operației, operanzii sunt scoși, cu ei se va efectua operația, iar rezultatul acesteia va fi plasat înapoi în stivă. Practic la final, valoarea expresiei postfixate, se află în vârful stivei.

Aplicatii practice

- Calculele au loc imediat ce operatorul este specificat. În acest fel, expresiile nu sunt evaluate ca un bloc de la stânga la dreapta, ci calculate bucată cu bucată, eficientizând timpul de calcul.
- Stiva permite stocarea unui rezultat intermediar pentru a fi folosit mai târziu, ceea ce permite calculatoarelor ce folosesc această formă, să evalueze expresii de orice complexitate, spre deosebire de calculatoarele algebrice.
- Parantezele de orice tip nu mai sunt necesare, calculele sunt deci mai simplu efectuate.
- Calculatoarele au implementate această metodă de calcul a expresiilor, expresiile infixate fiind folosite pentru ca oamenii să înțeleagă sensul algebric al acestora.

Algoritmul de evaluare a unei expresii postfixate

1. $\boldsymbol{while}$ *mai există cuvinte în expresie* $\boldsymbol{do}$
2. $\boldsymbol{citeste}$ *cuvântul următor*
3. $\boldsymbol{if}$ *cuvântul este o valoare* $\boldsymbol{then}$
4. $\boldsymbol{push}(cuvânt)$
5. $\boldsymbol{else}$ *cuvântul este un operator*
6. $\boldsymbol{pop}(\ )$*două valori de pe stivă*
7. $\boldsymbol{rezultat} =$ *operația aplicată pe cele două valori*
8. $\boldsymbol{push}(\boldsymbol{rezultat})$
9. $\boldsymbol{rezultat\ final} = \boldsymbol{pop}()$

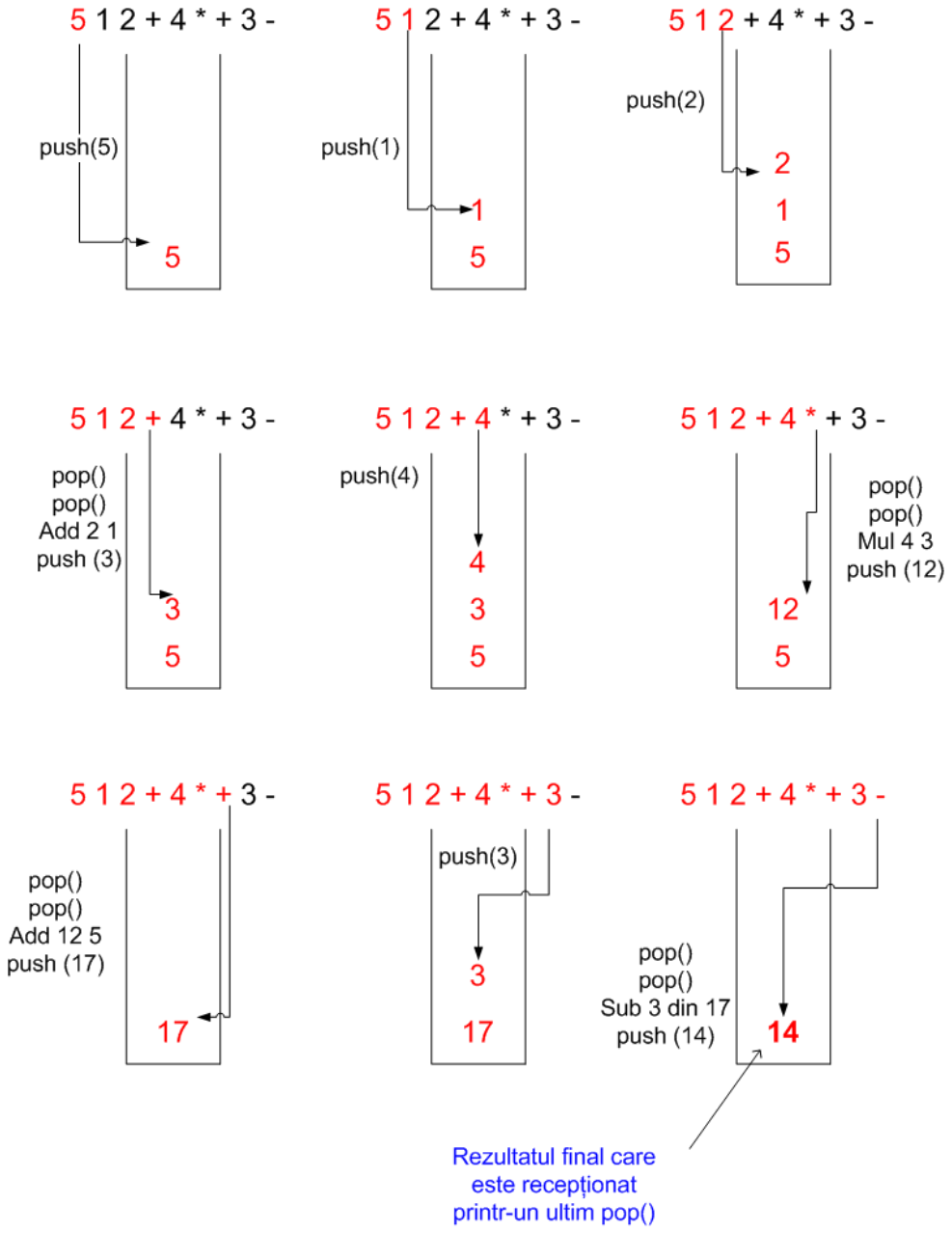
Algoritmul prezentat nu tratează eventualele erori de *underflow* ce apar de obicei atunci când expresia furnizată nu este validă. Pentru aceasta, va trebui să fie inserate operații de verificare dacă stiva este sau nu goală, iar in acest caz să aruncăm o excepție sau un mesaj.

Exemplu de folosire a algoritmului

Vom evalua expresia infixată $5 + ((1 + 2) * 4) - 3$ ce poate fi scrisă sub forma postfixată astfel:

$$5\ 1\ 2 + 4 * + 3 -$$

Mai jos este prezentată secvența de pași prin care este evaluată expresia de mai sus conform algoritmului de mai sus.



Translatarea unei expresii din forma infixată în forma postfixată

După cum am văzut anterior, forma infixată a fost mai întâi transformată în formă postfixată și după aceea s-a evaluat expresia. Trecerea de la forma infixată la cea infixată are loc conform algoritmului:

1. ***iniţializează*** *stiva*
2. ***repetă*** ***paşii***:
3. – ***daca*** *urmează o paranteză deschisă* "(", ***atunci*** *o pune in stiva*
4. – ***altfel daca*** *urmează un număr sau operand* ***atunci*** *il tipăreşte*
5. – ***altfel daca*** *urmează un operator* ***atunci***
6. – ***daca*** *în vârful stivei este un alt operator cu precedenta mai mare sau egal cu*
7. *a acestuia* ***atunci*** *scoate acel operator din stivă şi se tipăreşte*
8. – ***se pune*** *pe stivă operatorul*
9. – ***altfel*** *ceea ce urmează este o paranteză închisa* ")"
10. – ***se scot*** *simboluri din stivă şi se afişează până se întâlneşte paranteza deschisă*
11. *corespondentă, care se scoate şi ea din stivă*
12. ***până când*** *expresia se termină*
13. ***se scot*** *şi se afişează operatorii care au mai rămas în stivă* (*nu ar trebui să mai avem*
14. *paranteze deschise*)

Aplicarea algoritmului de mai sus pentru o expresie de tipul:

$$3 * 4 + (8 - 12) - 1$$

este prezentată mai jos.

3 * 4 + (8 - 12) - 1

Tiparit
3

Numerele se tipăresc

3 * 4 + (8 - 12) - 1

Tiparit
3

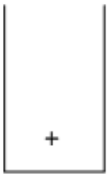
Operatorul este scris pe stivă

3 * 4 + (8 - 12) - 1

Tiparit
3 4

Operandul este tipărit

3 * 4 + (8 - 12) - 1



Tiparit
3 4 *

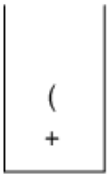
* are precedentă mai mare decât + și se tipărește

3 * 4 + (8 - 12) - 1

Tiparit
3 4 *

(se pune pe stivă

3 * 4 + (8 - 12) - 1



Tiparit
3 4 * 8

Operandul este tipărit

3 * 4 + (8 - 12) - 1

Tiparit
3 4 * 8

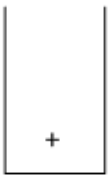
- este pus pe stivă

3 * 4 + (8 - 12) - 1

Tiparit
3 4 * 8 12

Numărul este tipărit

3 * 4 + (8 - 12) - 1



Tiparit
3 4 * 8 12 -

Se scoate tot până la paranteza (

3 * 4 + (8 - 12) - 1

Tiparit
3 4 * 8 12 - +

Tipărește + pentru că are aceeași precedență cu -

3 * 4 + (8 - 12) - 1

Tiparit
3 4 * 8 12 - +1

Operandul se tipărește

3 * 4 + (8 - 12) - 1

Golește toată stiva

Tiparit final
3 4 * 8 12 - +1 -

Coada

Coada este structura ce se conformează principiului FIFO și anume primul sosit primul ieșit. Coada are *cap*, primul element și o *coadă* adică ultimul element. Atunci când un nou element este introdus, el devine elementul *coadă* din structură. În figura 3 avem reprezentarea unei cozi sub forma unui șir.

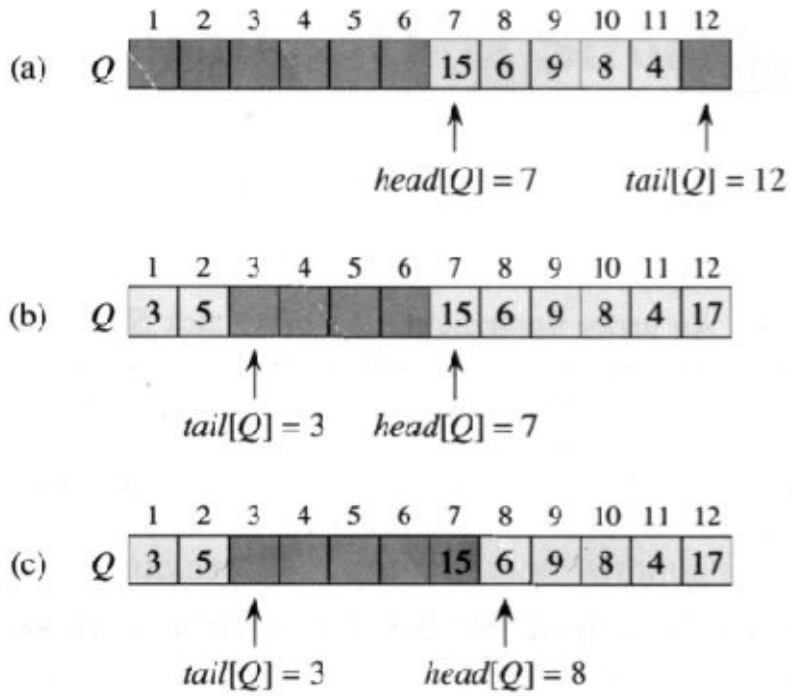


Figura 3. O coadă implementată folosind un șir de elemente $Q[1..12]$. elementele cozii sunt doar cele marcate cu gri deschis. a) Coada are 5 elemente în locațiile $[7..11]$ b) configurația cozii după apelul ENQUEUE(*Q*,17), ENQUEUE(*Q*,3) ȘI ENQUEUE(*Q*,5) c) configurația cozii după apelul DEQUEUE(Q) ce returnează valoarea 15, fostul cap al cozii. Noul cap devine 6, cel de pe poziția 8.

Asemenea stivei, avem operații specifice de adăugare și ștergere al unui element și anume ENQUEUE pentru introducerea unui nou element și DEQUEUE pentru ștergerea primului element din coadă. Figura 3 prezintă implementarea unei cozi cu cel mult $n - 1$ elemente folosind un șir $Q[1..n]$.

Coada are un atribut numit $head[Q]$ ce semnifică indexul capului, adică al primului element. Evident, elementele cozii vor fi plasate pe $head[Q]$, $head[Q]+1$, … , $tail[Q]-1$.

Atunci când $head[Q]=tail[Q]$ se numește coadă goală. Inițial orice coadă reprezentată astfel, are $head[Q]=tail[Q]=1$. Încercarea de a scoate dintr-o coadă goală duce la *underflow*. La fel ca în cazul stivei, o încercare de a introduce într-o coadă plină, duce la *overflow*. Mai jos sunt procedurile pentru introducere, respectiv scoaterea unui element în/din coadă.

$ENQUEUE(Q,x)$

1. $Q[tail[Q]] \leftarrow x$
2. $\boldsymbol{if}\ tail[Q]=length[Q]$
3. $\quad\boldsymbol{then}\ tail[Q] \leftarrow 1$
4. $\quad\boldsymbol{else}\ tail[Q] \leftarrow tail[Q]+1$

$DEQUEUE(Q)$

1. $x \leftarrow Q[head[Q]]$
2. $\boldsymbol{if}\ head[Q]=length[Q]$
3. $\quad\boldsymbol{then}\ \ head[Q] \leftarrow 1$
4. $\quad\boldsymbol{else}\ head[Q] \leftarrow head[Q]+1$
5. $\boldsymbol{return}\ x$

În implementarea actuală, nu sunt tratate erorile de *overflow* sau *underflow*, tocmai pentru că această coadă va înlocui elementele de pe primele poziții, în loc să *crească* spre dreapta, devenind astfel circulară. În prezentările acestor structuri am folosit șiruri pentru a exemplifica funcționalitatea lor, deoarece sunt mai ușor de înțeles. Cu toate acestea, în practică se folosesc structuri dinamice, precum listele.

Liste înlăntuite

O listă înlănțuită este o structură dinamică de dată formată din elemente ce respectă următoarea regulă: fiecare element conține o referință către următorul element. Practic, o listă este un set de item-uri unde fiecare este format dintr-un *nod* ce conține o *legătură* la un alt nod.

Dacă un nod conține legătură la el însuși, sau un nod conține legătură la un nod ce va direcționa prin *N* legături către nodul de start, structura se numește *ciclică*.

Ultimul nod din listă poate îndeplini doar una din următoarele condiții:

- este o legătură NULL ce nu indică nici un nod
- este o legătură *dummy* ce nu conține informație validă
- conține o legătură către primul nod, lista devenind circulară

Listele pot fi reprezentate cel mai simplu folosind tablouri. În acest caz, referințele către elementele din listă sunt indici din tablou. O alternativă este să folosim tablouri paralele: vom memora informația fiecărui nod într-o locație $VAL[i]$ a tabloului $VAL[1..n]$, iar adresa nodului într-o locație $LINK[i]$ a tabloului $LINK[1..n]$. Indicele locației primului nod este memorat în variabila $head$. În cazul listei vide $head = 0$. De asemenea următorul ultimului nod din listă este vid $LINK[ultimul\ nod\ din\ listă] = 0$. $VAL[head]$ conține informația din primul nod al listei, $LINK[head]$, conține adresa celui de-al doilea nod. . $VAL[LINK[head]]$ conține informația din al doilea nod, $LINK[LINK[head]]$ conține adresa celui de-al treilea nod, etc.

Acest mod de reprezentare este simplu, dar la o privire mai atentă, apare o problemă: cea a gestionării locațiilor libere. O soluție ar fi să reprezentăm locațiile libere tot sub forma unei liste înlănțuite. Atunci ștergerea unui nod din lista inițială implică inserarea sa în lista cu locații libere.
Un mod mai elegant de a reprezenta listele, este descris mai jos, și anume folosind referințe.

Liste simplu înlăntuite

Un item dintr-o listă simplu înlănțuită va fi alcătuit din informația fiecărui element și o referință către următorul nod din listă așa cum am reprezentat în figura 4.

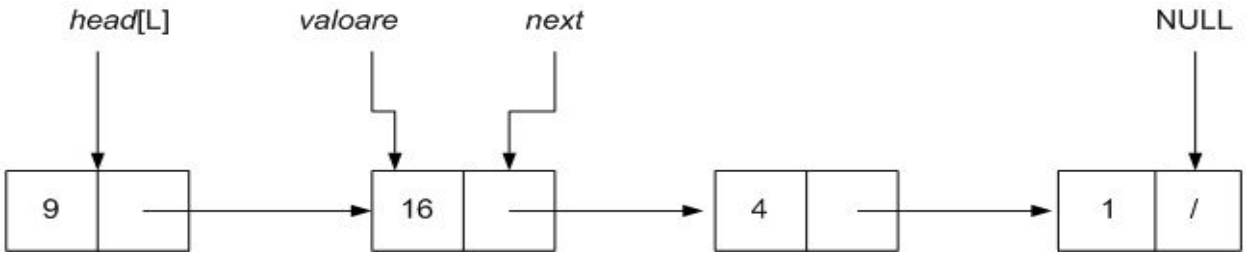


Figura 4. Reprezentarea unei liste simplu înlănțuite, cu ultimul element indicând spre NULL

Lista va fi reprezentată prin *L* iar valoarea unui element va fi notată cu *key*, și atributul $head[L]$ reprezintă primul element(*nod*) din listă. Dacă $head[L] = NULL$ atunci lista este goală. Referința către următorul element din listă este dată prin atributul *next* al fiecărui *nod*. Iată mai jos un exemplu minimal de implementare în Java, al unui nod dintr-o listă simplu înlănțuită:

```
class Node
  {
      Object key;
      Node next;
   }
```

Deoarece nu se știe tipul informației, acesta va fi *Object* și poate fi înlocuit după necesitate.

Alocarea unui nou element se va face prin:

```
Node x = new Node();
```

Odată cu această alocare, putem spune că lista a fost inițializată și *x* este capul listei. Întrebarea este cum ne vom referi la informația din următoarele noduri ale listei, cum le adăugăm? Vom utiliza *x.next* pentru a defini mai întâi următorul element din listă și apoi pentru al accesa.

De exemplu, putem crea un nou element *t* cu valoarea 16 ca în figura de mai sus astfel:

```
Node t = x.next; t.key = 16;
```

Următorul element al lui *t* va putea fi *accesat* în felul următor:

- fie prin apelul `t.next`
- fie prin apelul `x.next.next`

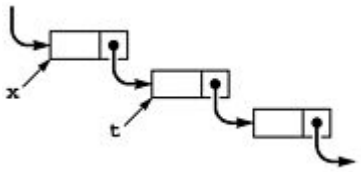


Figura 5. Accesarea următorului nod lui *t*

Ștergerea unui element, de exemplu a lui *t* se face prin următorul apel:

```
x.next = x.next.next;
```

Mai jos este o reprezentare a acestei operații, cum are ea loc:

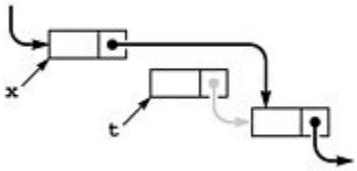


Figura 6. Ștergerea unui element din lista simplu înlănțuită

Pentru a insera un element într-o listă, de exemplu nodul *t* după *x* vom folosi următoarele instrucțiuni:

```
t.next = x.next; x.next = t;
```

Mai jos este o reprezentare grafică a acestor instrucțiuni:

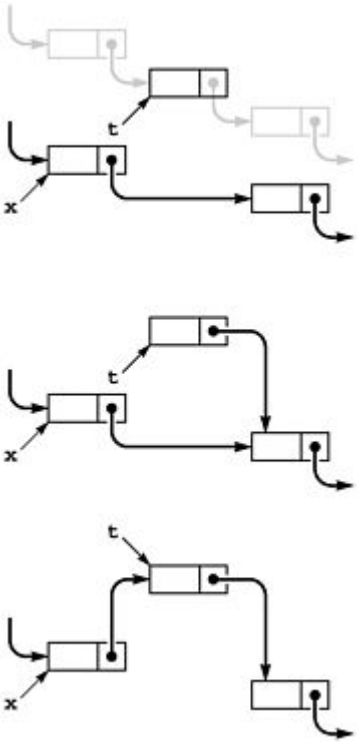


Figura 7. Inserarea unui nod *t* într-o listă simplu înlănțuită

Liste dublu-înlăntuite

În figura 8, avem o listă dublu înlănțuită, în care, un nod conține două referințe sau pointeri: *prev* și *next* pentru a indica atât predecesorul cât și nodul succesor din listă. Dat fiind un element *x* din listă $next[x]$ sau `x.next` cum am notat în Java, în exemplul anterior, este succesorul nodului *x* din listă. Asemenea $prev[x]$ este referința către predecesorul nodului *x*. Dacă $prev[x] = NULL$, elementul *x* nu

are predecesor, ceea ce înseamnă că este capul listei sau *head*. Dacă $next[x] = NULL$ atunci elementul *x* nu are succesor și este *coada* listei. Dacă $head[L]$ este *NULL* atunci lista este goală.

Căutarea într-o listă înlăntuită

Procedura descrisă mai jos, LIST_SEARCH(*L,k*), găsește elementul cu cheia *k,* din lista *L* printr-o căutare liniară, și returnează referința acelui element. Dacă nu este găsit nici un element, atunci *NULL* va fi returnat. Pentru lista simplu înlănțuită din figura 8.a), apelul LIST_SEARCH(*L,4*) returnează o referință la al treilea element din listă, iar apelul LIST_SEARCH(*L,7*) returnează NULL.

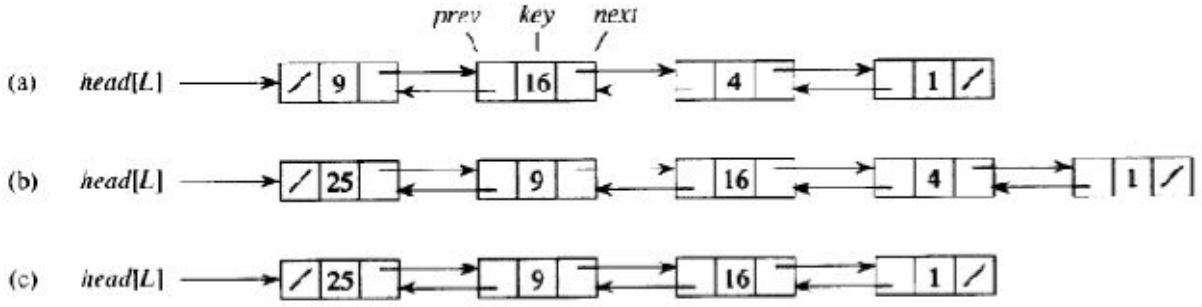


Figura 8. O listă dublu înlănțuită *L* reprezentând o mulțime dinamică $\{1,4,9,16\}$. Fiecare element din listă este un obiect ce conține o cheie și două referințe la obiectele anterior respectiv următor. *NULL* este indicat prin caracterul "/". Atributul $head[L]$ indică primul element al listei. b) Inserarea unui element cu cheia 25 înaintea primului element; acesta devine noul cap al listei. c) Ștergerea elementului cu cheia 4.

Mai jos avem algoritmul de căutare într-o listă simplu înlănțuită, după o valoare *k*

$LIST - SEARCH(L, k)$

1. $x \leftarrow head[L]$
2. $\boldsymbol{while}\ x \neq NULL\ \boldsymbol{and}\ key[x] \neq x$ do
3. $\quad x \leftarrow next[x]$
4. $\boldsymbol{return}\ x$

Timpul necesar căutării unui element într-o listă de *n* obiecte este *O(n)*.

Inserarea într-o listă dublu înlănțuită

Dat fiind un element *x* a cărei cheie a fost deja setată, procedura LIST-INSERT introduce elementul x în capul listei înaintea fostului prim element.

$LIST - INSERT(L, x)$

1. $next[x] \leftarrow head[L]$
2. $\boldsymbol{if}\ head[L] \neq NULL\ \boldsymbol{then}$
3. $\quad prev\big[head[L]\big] \leftarrow x$
4. $head[L] \leftarrow x$
5. $prev[x] \leftarrow NULL$

Timpul pentru inserarea folosind această metodă este *O(1).*

Ștergerea dintr-o listă înlăn uită

Procedura LIST-DELETE șterge un element *x* dintr-o listă *L*. *x* reprezintă referința către nodul care trebuie șters.

$LIST - DELETE(L, x)$

1. $\boldsymbol{if}\ prev[x] \neq NULL\ \boldsymbol{then}$
2. $\quad next\big[prev[x]\big] \leftarrow next[x]$
3. $\boldsymbol{else}$
4. $\quad head[L] \leftarrow next[x]$
5. $\boldsymbol{if}\ next[x] \neq NULL\ \boldsymbol{then}$
6. $\quad prev\big[next[x]\big] \leftarrow prev[x]$

Santinele

O santinelă este un obiect auxiliar, ce permite simplificarea condițiilor legate de limitele listei. De exemplu, având o listă *L* și un obiect $nil[L]$ ce reprezintă de fapt NULL, dar conține referință către elementele ce compun lista. Când vrem să specificăm o referință NULL din listă, o vom face prin intermediul unei santinele $nil[L]$. În figura 9, lista devine una circulară, având santinela plasată între cap și coadă. Câmpul $next\big[nil[L]\big]$ indică spre capul listei, și $prev\big[nil[L]\big]$ spre coada listei. Din moment

ce $next[nil[L]]$ indică spre cap, putem elimina atributul $head[L]$. O listă goală constă din santinelă, deoarece $next[nil[L]]$ și $prev[nil[L]]$ poate fi setat cu $nil[L]$.

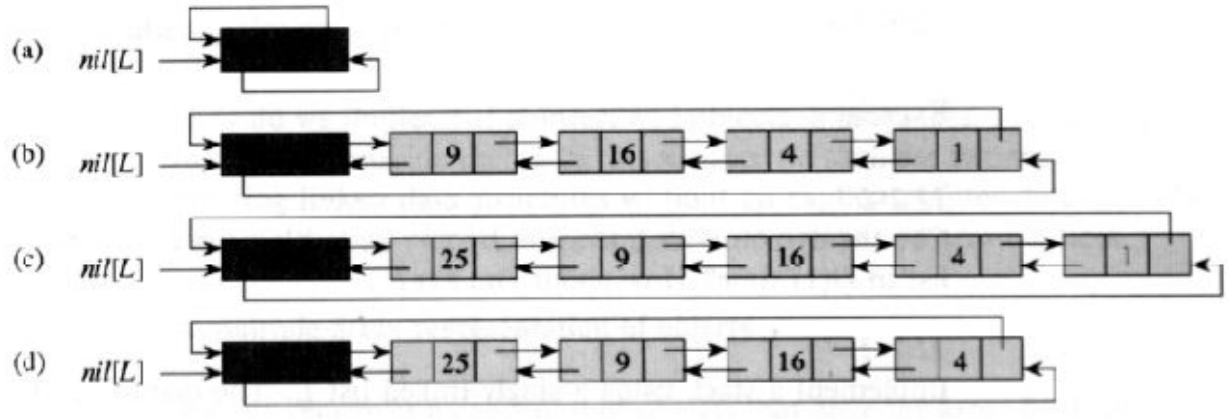


Figura 9. O listă dublu înlănțuită ce folosește o santinelă $nil[L]$.

În figura 9.a) este prezentată o listă goală, santinela având referință circulară la ea însăși.

În figura 9.b) lista conține mai multe elemente, având capul drept elementul cu cheia 9 și coada pe cel cu cheia 1. În figura 9.c) este prezentată lista după apelul LIST-INSERT'(*L,x*) unde $key[x] = 25$. În figura 9.d) se găsește lista după ștergerea obiectului cu cheia 1. Noua coadă este elementul cu cheia 4.

Mai jos avem procedurile de căutare, inserare și ștergere pentru listele cu santinelă.

$LIST - SEARCH'(L, k)$

1. $x \leftarrow next[nil[L]]$
2. $\boldsymbol{while}\ x \neq nil[L]\ \boldsymbol{and}\ key[x] \neq x\ \mathbf{do}$
3. $\quad x \leftarrow next[x]$
4. $\boldsymbol{return}\ x$

$LIST - INSERT'(L, x)$

1. $next[x] \leftarrow next[nil[L]]$
2. $prev\left[next[nil[L]]\right] \leftarrow x$
3. $next[nil[L]] \leftarrow x$
4. $prev[x] \leftarrow nil[L]$

$LIST - DELETE'(L, x)$

1. ***if*** $prev[x] \neq NULL$ ***then***
2. $\quad next\big[prev[x]\big] \leftarrow next[x]$
3. ***else***
4. $\quad head[L] \leftarrow next[x]$
5. ***if*** $next[x] \neq NULL$ ***then***
6. $\quad prev\big[next[x]\big] \leftarrow prev[x]$