Liste înlănţuite

O listă înlănţuită este o structură de date în care fiecare element este la rândul său o structură cu mai multe câmpuri, dintre care unul conţine informaţia, iar celelalte reprezintă legături de tip pointer către elementele vecine din listă.

1 Liste simplu înlănţuite

În cazul listelor simplu înlănţuite, fiecare element este o structură dispunând de un câmp pentru informaţie şi un câmp de legătură către următorul element din listă, ca în figura 1.

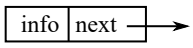

Figura 1: Structura unui nod: un câmp pentru informaţie şi un câmp de legătură către următorul element.

Spre deosebire de un vector / array, elementele unei liste înlănţuite nu sunt neapărat memorate în zone de memorie adiacente şi deci nu este necesară alocarea unui bloc de memorie compact pentru elemente sale. În schimb, de fiecare dată când se adaugă un nou element la listă, este necesară alocarea de memorie pentru acest element, iar când se şterge un element din listă, se elibereaă memoria respectivă.

Accesul la elementele listei se realizează prin capul listei, reprezentând primul element. Astfel este necesară o variabilă pentru păstrarea adresei capului listei. Un exemplu de listă simplu înlănţuită este prezentat schematic în figura 2.

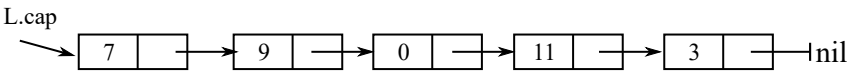

Figura 2: Listă simplu înlănţuită.

Operaţiile principale pe o listă înlănţuită sunt: adăugarea / ştergerea unui element la începutul listei, ştergerea unui element cu o anumită valoare şi parcurgerea listei. Pentru căutarea unui element, trebuie parcursă lista pornind de la primul element. Algoritmii pentru implementarea unora dintre operaţii sunt prezentaţi în continuare. Vom considera

pentru aceasta o structură LISTA_S, care are ca membru un pointer la primul element din listă, pe care îl vom denumi **cap**. Fiecare element al listei este o structură de tip NOD, cu un câmp **cheie** pentru informaţie şi un câmp **urm**, care indică următorul element din listă. Pentru ultimul element câmpul **urm** este *nil*.

Algoritm: Adauga_la_inceput(valoare)

Intrare: Lista simplu înlănţuită L în care adaug elementul *valoare*
aloca memorie pentru *nod_nou*
$nod\_nou.cheie \leftarrow valoare$
$nod\_nou.urm \leftarrow L.cap$
$L.cap \leftarrow nod\_nou$

Acest algoritm permite adăugarea unui element la începutul listei şi este ilustrat în figura 3

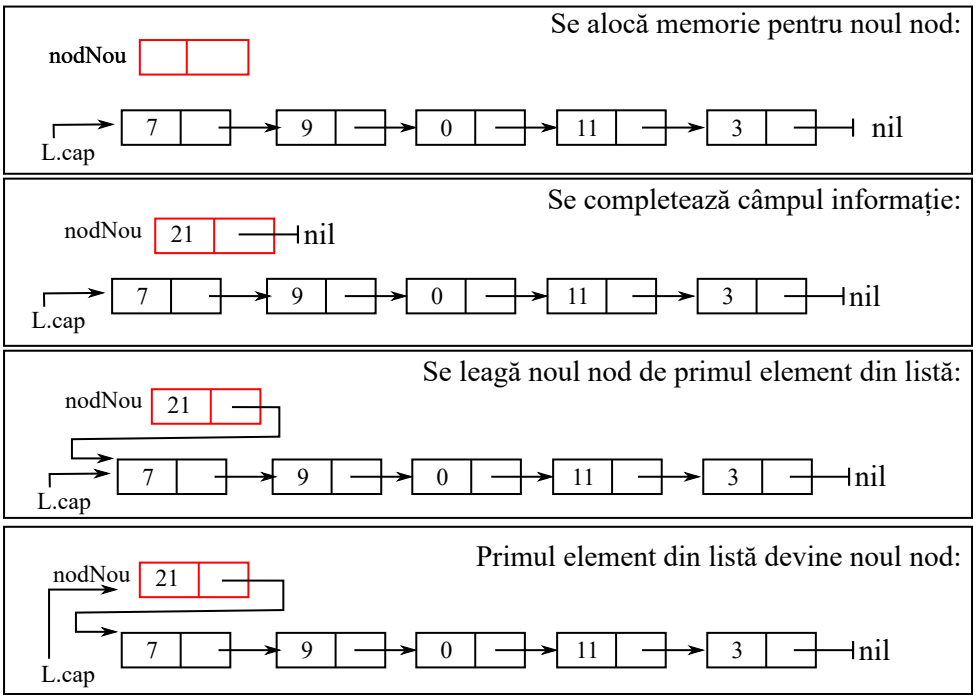

Figura 3: Adăugarea valorii 21 în lista simplu înlănţuită din imagine.

Complexitate: Algoritmul de adăugare a unui element la începutul listei are complexitate constantă de timp şi de memorie - $O(1)$

Pentru a şterge o anumită valoare dintr-o listă înlănţuită, aceasta trebuie mai întâi găsită în listă. Apoi trebuie realizată legătura dintre nodul care precede valoarea căutată şi nodul care urmează.

Algoritm: STERGE(valoare)

Intrare: Lista simplu înlănţuită L din care se şterge elementul *valoare*
daca $L.cap = nil$ **atunci**
  return
sfarsit_daca
daca $L.cap.cheie = valoare$ **atunci**
  $sterg \leftarrow L.cap$
  $L.cap \leftarrow L.cap.urm$
  elibereaza memoria pentru *sterg* return
sfarsit_daca
$curent \leftarrow L.cap$
cat_timp $curent.urm \neq nil$ *si* $curent.urm.cheie \neq valoare$ **executa**
  $curent \leftarrow curent.urm$
sfarsit_cat_timp
$sterg = curent.urm$
$curent.urm \leftarrow curent.urm.urm$
elibereaza memoria pentru *sterg*

În algorituml de mai sus variabila *curent* reprezită nodul cu care se parcurge lista şi, în final, nodul dinaintea nodului care va fi şters. Ştergerea din listă presupune de fapt legarea elementului *curent*, de nodul care îi urmează lui *curent.urm*.

Complexitate: Ştergerea unui element cu o anumită valoare din listă presupune parcurgerea acesteia, pâna se întâlneşte valoarea respectivă. În cel mai defavorabil caz, atunci când elementl nu se află în listă sau se află pe ultima poziţie, trebuie parcursă întreaga listă. În medie complexitatea este liniară, $\Theta(n)$, unde n = lungimea listei.

Algoritm: CAUTA(valoare)

Intrare: Lista simplu înlănţuită L în care caut *valoare*
$curent \leftarrow L.cap$
cat_timp $curent \neq nil$ *si* $curent.cheie \neq valoare$ **executa**
  $curent \leftarrow curent.urm$
sfarsit_cat_timp
return *curent*

Algoritmul *CAUTA* returnează un pointer către zona de memorie în care este stocat elementul cu valoarea *valoare*. În cazul în care acest element nu există, algoritmul returnează *nil*.

Complexitate: Căutarea unui element cu o anumită valoare din listă presupune parcurgerea acesteia, pâna se întâlneşte valoarea respectivă. În cel mai defavorabil caz, atunci când elementl nu se află în listă sau se află pe ultima poziţie, trebuie parcursă întreaga listă. În medie complexitatea este liniară, $\Theta(n)$, unde n = lungimea listei.

Desigur, există o serie de alte operaţii, care pot fi efectuate pe o listă simplu înlănţuită, majoritatea presupunând iterarea prin elementele acesteia. Spre exemplu, se pot insera sau şterge elemente pe o anumită poziţie, se pot efectua sortări, etc. Implementarea

acestor operaţii este propusă în temele de laborator.

2 Liste dublu înlănţuite

Elementele unei liste dublu înlănţuite posedă, spre deosebire de elementele unei liste simplu înlănţuite, alături de o legătură spre următorul element, şi o legătură către elementul predecedent. Accesul în listă se poate realizează prin capul listei, care indică primul element, dar şi prin ultimul element. Vom considera fiecare element din lista dublu înlănţuită ca fiind o structură de tip NOD, care dispune de câmpurile *cheie* pentru informaţie, *prec* pentru legătura la elementul precedent şi *urm* pentru legătura la elementul următor din listă.

Algoritmii prezentaţi în (Cormen) tratează liste, în care accesul se realizează doar la capul listei, prin variabila *head*. Adăugarea în listă se face în acest caz doar la începutul listei.

În acest curs însă vom considera implementarea prezentă în STL, în care accesul poate avea loc atât de la primul elemement, cât şi de la ultimul, adăugarea/ ştergerea unui element la începutul listei se va face prin funcţii de tip **Adauga_prim** (**push_front**) / **Sterge_prim** (**pop_front**), iar adăugarea /stergerea unui element la sfârşitul listei se va realiza prin **Adauga_ultim** (**push_back**) / **Sterge_ultim** (**pop_back**) în complexitate constantă. Structura LISTA va dispune de două câmpuri: *prim* şi *ultim*, care reprezintă pointeri către primul, respectiv ultimul element din listă.

Un exemplu de listă dublu înlănţuită este prezentat în figura 4.

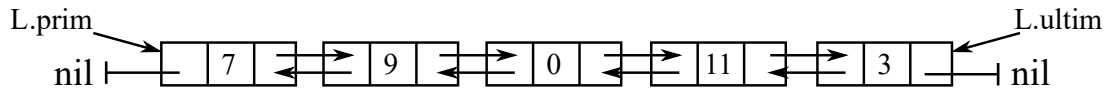

Figura 4: Listă dublu înlănţuită.

Printre cele mai importante operaţii asupra unei liste dublu înlănţuite sunt: adăugarea / eliminarea unui element de la începutul / sfârşitul listei sau de pe orice poziţie din listă, ştergerea unei valori din listă, iterarea prin listă în ambele direcţii. Căutarea unui element presupune parcurgerea listei, până la găsirea acestuia sau până la sfârşitul listei, în caz de eşec.

În continuare sunt prezentaţi algoritmii pentru adăugarea unui element la începutul listei, pentru ştergerea unui element de pe o anumită poziţie din listă, indicată printr-un pointer şi pentru ştergerea unui element cu o anumită cheie din listă. De asemenea va fi prezentat un algoritm de căutare a unui element în listă.

Algoritm: Adauga_prim(valoare)

Intrare: Lista dublu înlănţuită L în care adaug elementul *valoare*
aloca memorie pentru *nod_nou*
$nod\_nou.cheie \leftarrow valoare$
$nod\_nou.urm \leftarrow L.prim$
$nod\_nou.prec \leftarrow nil$
daca $L.prim \neq nil$ **atunci**
| $L.prim.prec \leftarrow nod\_nou$
sfarsit_daca
altfel
| //Atat primul cat si ultimul element sunt reprezentate de
| // *nod_nou*, care e unicul element
| $L.ultim \leftarrow nod\_nou$
sfarsit_daca
$L.prim \leftarrow nod\_nou$

Acest algoritm permite adăugarea unui element la începutul listei şi este ilustrat în figura 5.

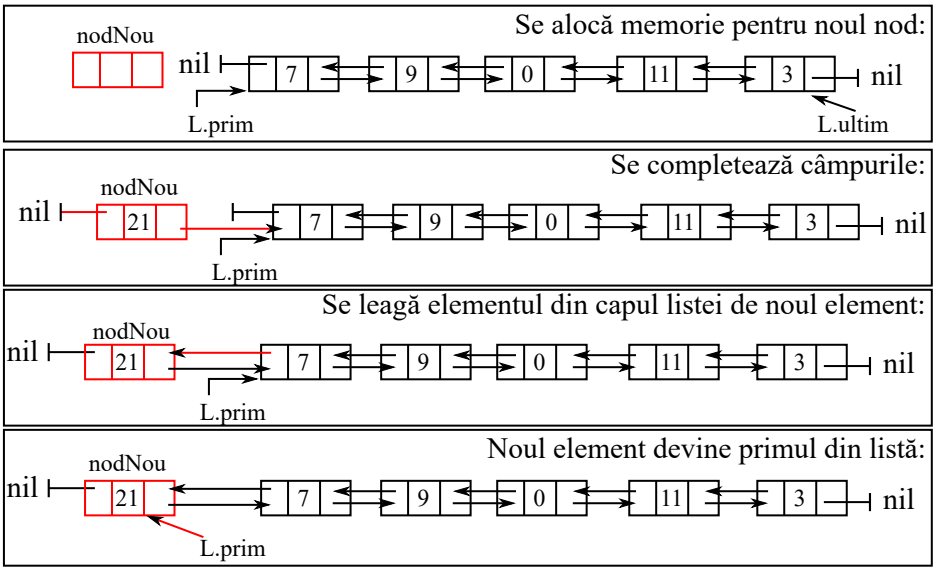

Figura 5: Adăugarea valorii 21 într-o listă dublu înlănţuită.

Complexitate: Algoritmul de adăugare a unui element la începutul listei are complexitate constantă de timp şi de memorie - $Theta(1)$

Funcţia *STERGE* (**erase**) presentată mai jos, realizează ştergerea unui element dintr-o listă înlănţuită reprezentat printr-un pointer la nodul respectiv. Acest lucru se poate face, dacă în prealabil a fost identificat elementul pe baza unei căutări, respectiv iterări prin listă.

Algoritm: STERGE(nod)

Intrare: Lista dublu înlănţuită L din care se şterge elementul indicat de pointerul *nod*

daca $nod.prec \neq nil$ **atunci**
| $nod.prec.urm \leftarrow nod.urm$
sfarsit_daca
altfel
| $L.prim \leftarrow nod.urm$
sfarsit_daca
daca $nod.urm \neq nil$ **atunci**
| $nod.urm.prec \leftarrow nod.prec$
sfarsit_daca
altfel
| $L.ultim \rightarrow nod.prec$
sfarsit_daca
dealoca memoria pentru *nod*

Complexitate: Operaţia de ştergere efectivă are complexitate constantă. Desigur, identificarea elementului, care trebuie şters, presupune iterarea prin listă şi deci complexitate liniară.

Algoritm: ELIMINA(valoare)

Intrare: Lista dublu înlănţuită L din care se şterge elementul *valoare*
$curent \leftarrow L.prim$
cat_timp $curent \neq nil$ *si* $curent.cheie \neq valoare$ **executa**
| $curent \leftarrow curent.urm$
sfarsit_cat_timp
daca $curent = nil$ **atunci**
| return
sfarsit_daca
STERGE(curent)

Funcţia *ELIMINA* (**remove**) are ca parametru o valoare ce se doreşte eliminată din lista dublu înlănţuită. Acest lucru presupune întâi căutarea valorii în listă, iar apoi apelarea funcţiei *STERGE*, definită mai sus. Algoritmul prezentat elimină prima apariţie a valorii date de parametru.

Complexitate:

- Stergerea efectivă: $\Theta(1)$
- Căutarea valorii: $\Theta(n)$, n - numărul de elemente din listă.

Putem defini şi o funcţie de căutare a unui element în listă. Algoritmul corespunzător are în mod evident complexitate medie liniară.

Observaţie: În cazul clasei **list** din STL funcţia **remove(value)** elimină TOATE elementele ca au chiea egală cu *value*. Algoritmul de mai sus poate fi adaptat în aşa fel, încât să realizeze acest lucru.

Algoritm: CAUTA(valoare)

Intrare: Lista dublu înlănţuită L în care caut *valoare*
curent $\leftarrow$ *L.prim*
cat_timp *curent* $\neq$ *nil si curent.cheie* $\neq$ *valoare* **executa**
  curent $\leftarrow$ *curent.urm*
sfarsit_cat_timp
return *curent*

Celelalte operaţii pentru liste dublu înlănţuite sunt propuse pentru implementare la secţiunea de exerciţii.

2.1 Utilizarea listelor înlănţuite

Dacă ar fi să comparăm listele înlănţuite cu structuri de tip vector, se poate observa uşor că cel mai important dezavantaj al listelor este acela, de a nu putea accesa elementele prin poziţie. Accesul rapid prin poziţie este specific vectorilor şi este posibil, datorită memorării elementelor acestuia în locaţii de memorie succesive. În schimb, inserearea sau ştergerea unui element la începutul structurii, sau de pe o poziţie arbitrară este mult mai eficientă în cazul listelor înlănţuite.

Astfel, dacă aplicaţia presupune o mulţime de elemente, care se modifică relativ rar sau preponderent prin adăugarea de elemente la sfârşit, dar în care accesul prin poziţie este frecvent, evident o structură de tip vector este preferabilă. În schimb, acolo unde modificările prin adăugare / ştergere sunt frecvente, iar accesul prin poziţie este de mică importanţă, se preferă liste înlănţuite.

Avantajul unei liste simplu înlănţuite faţă de o listă dublu înlănţuită este acela că necesită semnificativ mai puţină memorie pentru stocarea legăturilor către elemente vecine. În schimb, spre deosebire de listele dublu înlănţuite, o listă simplu înlănţuită permite iterarea doar într-o singură direcţie, de la capul listei spre sfârşit.

Atunci când este necesară deplasarea prin listă doar într-un singur sens, se preferă liste simplu înlănţuite. Un astfel de exemplu este a acela al implementării tabelelor de repartizare - *hash tables*, care vor fi discutate în capitolul următor.

Atunci când este necesară navigarea în ambele sensuri, se vor prefera liste dublu înlănţuite. Un exemplu este acela al navigării inainte şi inapoi între pagini web sau implementarea bucket-urilor în algoritmul *Bucket-Sort*.