MODULUL 1 - STRUCTURI DE DATE LINIARE

O structură de date se numeşte liniară, dacă elementele sale alcătuiesc o secvenţă. O astfel de structură are un prim element şi un ultim element, iar trecerea de la un element la următorul se face în mod secvenţial şi depinde de modul de parcurgere respectiv de disciplina de intrare/ieşire a structurii. Pe baza modului de manipulare a elementelror structurii, respectiv pe baza disciplinei de intrare/ieşire distingem mai multe tipuri de structuri liniare, de exemplu stive, cozi, liste simplu sau dublu înlănţuite.

1 Stiva - *stack*

Definiţie: Stiva este o structură liniară de tip **LIFO** - *Last In First Out*, adică ultimul element introdus va fi primul care se extrage pentru prelucrare. Accesul la elementele stivei se realizează doar prin vârful stivei, unde se află ultimul element introdus.

Reprezentarea în memorie:

- - utilizând un tablou liniar - *array*
- - utilizând o listă înlănţuită

1. Reprezentarea secvenţială - tablou liniar

Se utilizează pentru stiva S

- un vector *data* de dimensiune dim = nr. maxim de elemente ce pot fi introduse.
- o variabilă vf ce reprezentă vârful stivei = poziţia în vector pe care se află ultimul element aparţinând stivei

Observaţii:

- Dacă $vf = -1$ atunci stiva este vidă şi nu pot extrage elemente
- Dacă $vf = dim - 1$ stiva este plină şi nu pot adăuga elemente noi.

Un exemplu de stivă reprezentată secvenţial este prezentat în figura 1. Stiva conţine 7 elemente, iar variabila $vf = 6$.

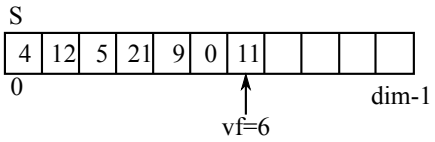


Figure 1: Stivă reprezentată printr-un vector..

Considerând structura STIVA cu atributele *data* - vector de chei, *dim* - dimensiunea maximă a stivei şi vf -poziţia vârfului, putem utiliza următoarele funcţii de adăugare şi eliminare a unui element din stivă S.

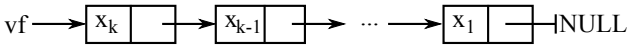
```
PUSH(S, x)
    daca S.vf = S.dim - 1 atunci
         scrie ("Stiva este plina!")
    altfel
         S.vf = S.vf + 1
         S.data[S.vf] = x
    sfarsit daca
RETURN
```

```
POP(S)
   daca S.vf = -1 atunci
        scrie("Stiva este vida")
   altfel
        S.vf = S.vf - 1
   sfarsit daca
RETURN
```

Complexitate: ambele operaţii au complexitatea $O(1)$.

2. Reprezentare dinamică

Pentru fiecare element poate fi utilizată o structură care conţine 2 câmpuri: INFO - pentru informaţie şi NEXT - pointer către elementul următor. Stiva poate fi reprezentată grafic astfel:



Funcţiile de adăugare şi eliminare a elementului din vârful stiveri S sunt:

```
PUSH(S, x)
    aloca memorie pentru y
    y.info = x
    y.next = S.vf
    S.vf = y
```

```
RETURN

 POP(S)
    daca S.vf = NULL atunci
         scrie ("Stiva este vida")
    altfel
         y = S.vf
         S.vf = S.vf.next
         dealoca memoria alocata lui y
    sfarsit daca
RETURN
```

Complexitate: ambele operaţii au complexitatea $O(1)$.

2 Coada -*queue*

Definiţie: Coada este o structură liniară de tip **FIFO** - *First In First Out*, adică primul element introdus va fi primul care se extrage pentru prelucrare. Coada modelează procese care presupun formarea de cozi, de exemplu deservirea clienţilor la un ghişeu. De asemenea se utilizează cozi pentru operaţii precum parcurgerea în lăţime a unui graf sau a unui arbore.

Comparare cu stiva. Spre deosebire de stivă, unde se utilizează o variabilă pentru accesarea vârfului stivei şi atât adăugarea cât şi extragerea elementelor se realizează pe baza acesteia, în cazul unei cozi este necesară memorarea a două elemente: primul - aici se face extragerea şi ultimul - aici se face adăugarea.

Reprezentare

O coadă, ca şi o stivă, poate fi construită în două moduri:

- secvenţial - cu ajutorul unui tablou liniar - **array**
- dinamic - cu ajutorul unei liste înlănţuite

1. Reprezentarea secvenţială

Se utilizează pentru coada C

- un vector *data* de dimensiune dim = nr. maxim de elemente ce pot fi introduse.
- două variabile *prim* şi *ultim* care reprezentă poziţia în vector pe care se află primul, respectiv ultimul element aparţinând cozii. Dacă $ultim < prim$ atunci coada este vidă şi nu pot fi extrase elemente, dacă $ultim = dim - 1$ stiva este plină şi nu pot fi adăugate elemente noi.

Un element nou se adaugă mereu după ultimul element şi se creşte variabila *ultim*, iar un element se extrage din coadă prin creşterea variabilei *prim*. De fapt, elementul respectiv nu se şterge efectiv din memorie, dar nu mai este considerat ca făcând parte din coadă. Acest lucru este ilustrat în fig. 2.

La inserţie trebuie verificat dacă este plină coada, iar la ştergere, dacă este goală.

Observaţie: la fiecare extragere şi adăugare, coada migrează înspre dreapta. Astfel se poate ajunge la situaţia în care în vectorul de date corespunzător cozii sunt multe poziţii neocupate, dar totuşi la apelarea functiei PUSH se primeşte mesaj de coadă plină - fig. 2. Acest lucru se evită utilizând cozi circulare!

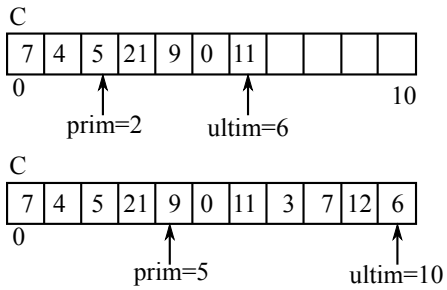


Figure 2: Din coada C s-au extras elementele 5 şi 21 şi s-au adăugat elementele 3, 7, 12, 6. În acest moment primele 4 poziţii sunt considerate neocupate, dar funcţia PUSH returnează mesajul de coadă plină.

Coada circulară

În cazul unei cozi circulare, atunci când variabila $ultim = dim - 1$, la următoarea inserţie *ultim* devine 0, deci se reia circular parcurgerea cozii de la început. Similar se procedează cu variabila *prim* la extragere. Astfel se va primi mesaj de coadă plină doar atunci când sunt ocupate toate poziţiile alocate în vector pentru elementele cozii.

Condiţiile de coadă vidă/plină

În cazul cozilor circulare condiţiile de coada plină şi coadă vidă descrise mai sus nu mai sunt valabile:

- Inegalitatea $prim > ultim$ nu înseamnă decât faptul că s-a reluat parcurgerea cozii de la început - vezi fig.3.
- Evident condiţia $ultim = dim - 1$ nu mai generează mesajul de coadă plină

Soluţionarea problemei detecţiei cozii vide sau pline: Această problemă se rezolvă în modul următor. O poziţie din vectorul *data* indicată de elementul *ultim* nu va fi ocupată niciodată. Ea este utilizată doar pentru marcarea sfârşitului cozii. De fapt poziţia indicată de *ultim* reprezintă prima poziţile liberă în vector, după ultimul element din coadă. Condiţiile de coadă vidă / coadă plină sunt în acest caz:

- Dacă $prim = ultim$ atunci coada este vidă
- Dacă $ultim + 1 = prim$ atunci coada este plină. Atentie: cand $ultim = dim - 1$ coada este plină dacă $prim = 0$.

În figura 3 este reprezentat un exemplu de coadă circulară.

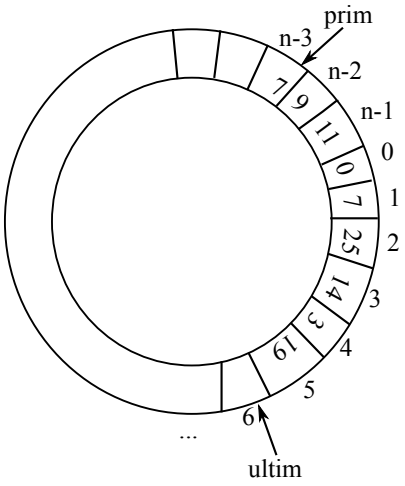
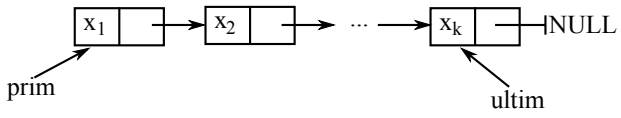


Figure 3: Coadă circulară în care la momentul curent variabila *prim* indică poziţia $n - 3$ şi variabila *ultim* indică poziţia neocupată 6.

2. Reprezentare dinamică

Pentru fiecare element poate fi utilizată o structură care conţine 2 câmpuri: INFO - pentru informaţie şi NEXT - pointer către elementul următor. Coada poate fi reprezentată grafic astfel:



Alocarea memoriei pentru coadă se face dinamic, nu static, aşa ca la vector, adică la introducerea un element nou, trebuie mai întâi alocată memorie, iar la extragerea unui element, trebuie eliberată zona de memorie care era alocată acelui element.

Sunt necesare două variabile *prim* şi *ultim*, în care se reţin adresele primului respectiv ultimului element din coadă. Elemente noi se adaugă mereu după ultimul element iar la eliminare se consideră primul element din coadă!

Algoritmi inserţie / eliminare

```
 PUSH(C,x)
      aloca memorie pentru y
      y.info = x
      y.next = NULL
      daca C.prim = NULL atunci //coada vida
            C.prim = C.ultim = y
      altfel
            C.ultim.next = y
            C.ultim = y
      sfarsit daca
RETURN
```

```
 POP(C)
     daca C.prim = NULL atunci
          scrie ("Coada este vida")
     altfel
          y = C.prim
          C.prim = C.prim.next
          sterge y
     sfarsit daca
RETURN
```

3 Liste înlănţuite

3.1 Liste simplu înlănţuite

O listă simplu înlănţuită este o structură de date liniară în care fiecare element conţine un câmp de legătură către elementul următor din listă, câmp pe care îl vom nota prin *next*.

De fapt stivele şi cozile alocate dinamic sunt cazuri particulare de liste simplu înlănţuite cu o anumită disciplină de intrare/ieşire.

Comparaţie cu stive/cozi: Spre deosebire de stive şi cozi, listele în general permit inserarea respectiv ştergerea în orice poziţie a listei. De asemenea suportă operaţia de căutare a unei chei.

Acces: Lista se accesează prin capul listei, reprezentând primul element. Astfel este necesară o variabilă pentru păstrarea adresei capului listei. Notăm în continuare capul listei prin *head*.

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

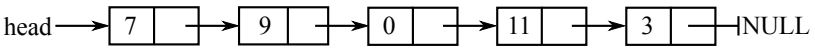


Figure 4: Listă simplu înlănţuită.

Algoritmii pentru implementarea operaţiilor cu liste înlănţuite sunt prezentaţi în continuare.

1. Căutarea unui element

```
CAUTA_LISTA(L,k)
      x = L.head
      cat timp x ≠ NULL si x.info ≠ k
                x = x.next
      sfarsit cat timp
RETURN x
```

Complexitate: În cel mai defavorabil caz, atunci când k nu se găseşte în listă, complexitatea este $O(n)$, unde n = lungimea listei.

2. Inserarea elementului x în listă se realizează la începutul listei

```
INSERTIE_LISTA(L, x)
        x.next = L.head
        L.head = x
RETURN
```

Complexitate: $O(1)$

3. Ştergerea unui element din listă - se presupune ca elementul există în listă şi a fost identificat prin funcţia CAUTA_LISTA.

```
STERGERE_LISTA(L, x)
        daca x = L.head atunci
             L.head = L.head.next
        altfel
             y = L.head
             cat timp y.next ≠ x
                       y = y.next
             sfarsit cat timp
             y.next = x.next
        sfarsit daca
RETURN
```

Complexitate: $O(n)$ - presupune căutarea predecesorului lui x. Acest lucru se evită în cazul listelor dublu înlănţuite.

3.2 Liste dublu înlănţuite

Listele dublu înlănţuite sunt structuri liniare de date în care fiecare element posedă atât o legătură către elementul predecesor *prev* cât şi una către elementul următor *next*. Accesul în listă se realizează prin capul listei către care pointează o variabilă *head*.

Un exemplu de listă dublu înlănţuită este prezentat în figura 5. În cele ce urmează sunt prezentaţi algoritmii pentru operaţiile de căutare, inserţie şi ştergere a unui element din listă.

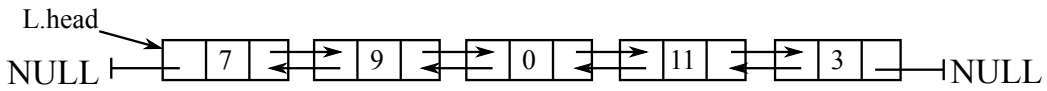


Figure 5: Listă dublu înlănţuită.

1. **Căutarea unui element**: la fel ca pentru liste simplu înlănţuite

```
CAUTA_LISTA_D(L, k)
      x = L.head
      cat timp x ≠ NULL si x.info ≠ k
               x = x.next
      sfarsit cat timp
RETURN x
```

2. **Inserarea** elementului x in listă se realizează la începutul listei

```
INSERTIE_LISTA_D(L, x)
         x.next = L.head
         daca L.head ≠ NULL atunci
```

```
            L.head.prev = x
        sfarsit daca
        x.prev = NULL
        L.head = x
RETURN
```

Complexitate: $O(1)$

3. Ştergerea unui element din listă - se presupune ca elementul există în listă şi a fost identificat prin funcţia de căutare.

```
STERGERE_LISTA_D(L, x)
        daca x.prev ≠ NULL atunci
            x.prev.next = x.next
        altfel
            L.head = x.next
        sfarsit daca
        daca x.next ≠ NULL atunci
            x.next.prev = x.prev
RETURN
```

Complexitate:

- Stergerea efectiva: O(1)
- Dacă elementul x trebuie întâi căutat: O(n)