Stive.Cozi

1 Stiva - *stack*

Definiţie: Stiva este o structură 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.

Pentru stocarea elementelor unei stive pot fi folosie diferite containere:

- - un tablou unidimensional - *array* - reprezentare secvenţială
- - o listă înlănţuită - reprezentare dinamică
- - utilizând o structură de tip *dequr* - double endend- queue (standard pentru STL).

1.1 Reprezentarea secvenţială - tablou liniar

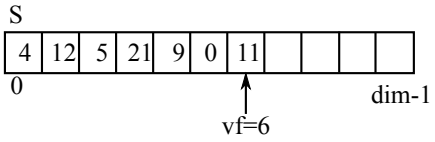

Figura 1: Stivă memorată într-un vector.

Se utilizează pentru structura stiva

- 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. Este nevoie de realocare.

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

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.

Algoritm: PUSH

Intrare: Stiva S în care adaug elementul val
daca $S.vf = S.dim - 1$ **atunci**
| realocare de memorie
sfarsit_daca
$S.vf \leftarrow S.vf + 1$
$S.data[S.vf] \leftarrow val$

Algoritm: POP

Intrare: Stiva S din care se extrage elementul din vârf
daca $S.vf \neq$ *-1* **atunci**
| $S.vf \leftarrow S.vf - 1$
sfarsit_daca

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

1.2 Reprezentare dinamică

Acest mod de reprezentare foloseşte pentru stivă o listă simplu înlănţuită. Nu este necesară o listă dublu înlănţuită deoarece nu se iterează prin container. Accesul se realizează doar în capul listei. Pentru fiecare element poate fi utilizată o structură care conţine 2 câmpuri: $info$ - pentru informaţie şi urm - pointer către elementul următor. Stiva poate fi reprezentată grafic precum în figura 2.

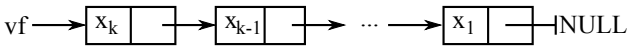

Figura 2: Stivă memorată într-o listă.

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

Algoritm: PUSH

Intrare: Stiva S în care adaug şi elementul val care se adaugă
aloca memorie pentru $nodNou$
$nodNou.info \leftarrow val$
$nodNou.urm \leftarrow S.vf$
$S.vf \leftarrow nodNou$

Algoritm: POP

Intrare: Stiva S din care se elimină elementul din vârf
daca $S.vf = nil$ **atunci**
    eroare
    return
sfarsit_daca
$vechi \leftarrow S.vf$
$S.vf \leftarrow S.vf.urm$
elibereaza mem pentru $vechi$

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

2 Coada -*queue*

Definiţie: Coada este o structură 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

Pentru stocarea elementelor unei cozi pot fi folosite diferite containere:

- un tablou unidimesional - **array/vector** - reprezentare secvenţială
- o listă înlănţuită - reprezentare dinamică
- o structură de tip *deque* - standard pentru *queue* din STL

2.1 Reprezentarea secvenţială

Se utilizează pentru structura coada

- 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 $ultim = dim - 1$ coada 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. 3.

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 coada e considerată plină - fig. 3. Acest lucru se rezolvă prin folsirea unei cozi circulare!

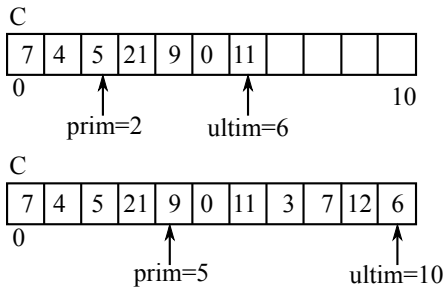

Figura 3: Din coada C s-au extras elementele 5 şi 21 şi s-au adăugat elementele 3, 7, 12, 6. În acest moment primele 4 poziţii sunt considerate neocupate, dar funcţia PUSH returnează mesajul de coadă plină.

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.4.
- 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 4 este reprezentat un exemplu de coadă circulară.

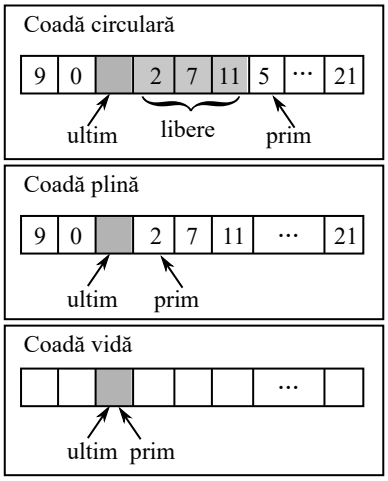

Figura 4: Coadă circulară în care la momentul curent variabila *prim* indică poziţia $n-3$ şi variabila *ultim* indică poziţia neocupată 6.

2.2 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 ca în fig. 5.

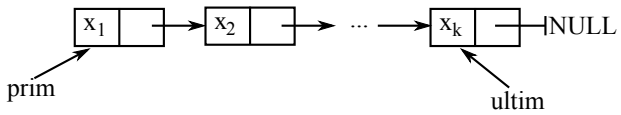

Figura 5: Coada reprezentată printr-o listă simplu înlănţuită

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!

Algoritm: PUSH

Intrare: Coada C în care adaug şi elementul *val* care se adaugă
aloca memorie pentru *nodNou*
$nodNou.info \leftarrow val$
$nodNou.urm \leftarrow nil$
daca *coada e vida* **atunci**
| $prim = ultim = nodNou$
sfarsit_daca
altfel
| $C.ultim.urm \leftarrow nodNou$
| $C.urm \leftarrow nodNou$
sfarsit_daca

Algoritm: POP

Intrare: Coada C din care se elimină primul element
daca C *e vidua* **atunci**
| eroare
| return
sfarsit_daca
$vechi \leftarrow C.prim$
$C.prim \leftarrow C.prim.urm$
elibereaza mem pentru *vechi*

3 STL - stive şi cozi

În STL stivele şi cozile sunt adaptori de containere. La baza acestora se pot afla containerele *deque* şi *list*. Pentru *stack* pot fi folosite drept containere şi $forward\_list$ şi *vector*, ceea ce nu este posibil pentru *queue*.

3.1 stack

```
template <class T, class Container = deque<T>> class stack;
```

Se observă faptul că pentru fiecare stivă trebuie definit tipul de date stocat de către aceasta. Containerul predefinit este *deque* dar poate fi schimbat de către utilizator, aşa cum s-a specificat anterior.

Clasa *stack* dispune de o serie de funţii, printre care cele mai importante sunt:

- **push** - permite adăugarea unui element nou la stivă
- **pop** - permite extragerea elementului din vârful stivei
- **top** - returnează obţinerea elementului din vârful stivei
- **empty** - verifică dacă stiva este vidă.