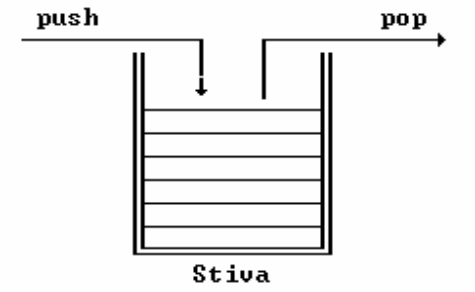
Laborator de Structuri de Date si Algoritmi – Lucrarea nr. 5

Stive si cozi

1. Stive

1.1. Introducere

O stiva este o structura de date de tip *container* (depoziteaza obiecte de un anumit tip) organizata dupa principiul LIFO (Last In First Out). Operatiile de acces la stiva (***push*** - adauga un element in stiva si ***pop*** - scoate un element din stiva) sint create astfel incit *pop* scoate din stiva elementul introdus cel mai recent.



O stiva este un caz particular de lista, si anume este o lista pentru care operatiile de acces (inserare, stergere, accesare element) se efectueaza la un singur capat al listei.

Daca STACK este tipul stiva si ATOM tipul obiectelor continute in stiva atunci operatiile care definesc tipul structura de stiva pentru tipul STACK sint:

■ CREATE() -> STACK
Operatia CREATE nu primeste parametri si creeaza o stiva care pentru inceput este vida (nu contine nici un obiect).

■ PUSH(STACK, ATOM) -> STACK
Operatia PUSH primeste ca parametri o stiva si un obiect si produce stiva modificata prin adaugarea obiectului in stiva.

■ POP(STACK) -> STACK, ATOM
Operatia POP primeste ca parametri o stiva pe care o modifica scotind un obiect. Deasemenea produce ca rezultat obiectul scos din stiva.

■ TOP(STACK) -> ATOM
Operatia TOP intoarce ca rezultat obiectul din virful stivei pe care o primeste ca parametru.

■ ISEMPTY(STACK) -> boolean
Operatia ISEMPTY este folosita pentru a testa daca stiva este vida.

1.2. Implementari ale conceptului de stiva in C/C++

[A] Stiva ordonata

O stiva poate fi organizata pe un spatiu de memorare de tip *tablou* (vector). Astfel, o stiva va fi formata dintr-un *vector* de elemente de tip Atom si un *indicator* (index) al virfului stivei.

```
struct Stack{
    int sp;                    // "stack pointer"
    Atom vect[DIMSTACK];
    };
```

Operatiile care definesc structura de stiva vor fi implementate prin urmatoarele functii:

```
void initStack(Stack& S);
void push(Stack& S, Atom a);
Atom pop(Stack& S);
Atom top(Stack S);
int isEmpty(Stack S);
```

Observatie:

Un obiect Stack este un obiect de dimensiuni mari, fapt pentru care nu este eficienta pasarea obiectelor de acest tip prin valoare. Este indicata folosirea parametrilor referinta pentru pasarea stivei chiar daca procedura nu are intentia de a o modifica (cazul functiilor top si isEmpty).

Laborator de Structuri de Date si Algoritmi – Lucrarea nr. 5

TEMA 1

Scrieti si testati functiile ce implementeaza operatiile ce definesc structura de stiva ordonata (*initStack*, *push*, *pop*, *top* si *isEmpty*).

[B] Stiva inlantuita

O stiva poate fi implementata ca o lista inlantuita pentru care operatiile de acces se fac numai asupra primului element din lista. Deci operatia PUSH va insemna inserare in prima pozitie din lista (in fata) iar POP va insemna stergerea primului element din lista. Pentru a manevra o stiva vom avea nevoie de un pointer la primul element din inlantuire, deci vom echivala tipul Stack cu tipul "pointer la element de lista", iar functiile care implementeaza operatiile de acces vor avea aceleasi prototipuri cu cele date mai sus.

```
struct Element {
    Atom data;
    Element* link;          //legatura
    };

typedef Element* Stack;
```

TEMA 2

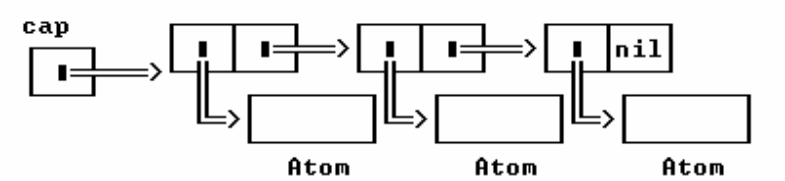
Scrieti si testati functiile ce implementeaza operatiile ce definesc structura de stiva inlantuita (*initStack*, *push*, *pop*, *top* si *isEmpty*).

[C] Stiva generica

Operatiile de acces la o stiva sint aceleasi indiferent de tipul obiectelor continute in stiva. Ar fi normal sa folosim codul scris pentru a implementa procedurile *push* si *pop* pentru operatiile de acces la orice tip particular de stiva (stiva de intregi, stiva de ferestre, etc.). Acest lucru nu este posibil in implementarile date mai sus, unde procedurile care implementeaza operatiile de acces la stiva sint particularizate pentru tipul Atom.

Iata o solutie care ar permite definirea operatiilor de acces la stiva independent de tipul informatiei care se depune in stiva. Vom folosi o *stiva inlantuita de pointeri fara tip*. Astfel fiecare element de lista va memora adresa unui bloc de memorie in care este memorata informatia utila.

Laborator de Structuri de Date si Algoritmi – Lucrarea nr. 5



Vom obtine aceasta implementare vom defini tipul Atom drept la tipul "pointer" in definitia stivei inlantuite de mai sus.

O astfel de definitie va avea avantajul ca o aceeasi stiva sa pastreze elemente de tipuri diferite, tipul pointer insemnind tocmai "pointer la orice". Ramine in sarcina programatorului sa defineasca un mecanism prin care sa recunoasca tipul elementului scos din stiva.

TEMA 3

1. ***Forma poloneza*** (notatia postfixata) a unei expresii aritmetice se caracterizeza prin aceea ca operatorii apar in ordinea in care se executa operatile la evaluarea expresiei. De aceea evaluare unei expresii in forma poloneza se face parcurgand intr-un singur sens expresia si executand operatiile tinand seama de paritatea lor.

Definirea recursiva a expresiilor in forma poloneza (f.p.) este urmatoarea:

- Pentru orice operand (constanta sau variabila) *E*, *E* este forma poloneza a operandului *E*
- Daca *E* este o expresie de forma *E1 op E2* , f.p. a expresiei este *E1′ E2′ op* unde *E1′* si *E2′* sunt respectiv f.p. ale expresiilor *E1* si *E2*
- Daca *E* este o expresie de forma (*E1*). f.p. a expresiei *E1* este de asemenea f.p. a expresiei *E*

Exemple:
(9-5)+2 are f.p. 95-2+
9-(5+2) are f.p. 952+-

Obs. Expresiile in forma poloneza nu contin paranteze.

Un algoritm simplu de evaluare a expresiilor in forma poloneza este urmatorul:

```
// Expresia se afla intr-un tablou s cu elemente simboluri de
// tip operand sau operator binar

i=0
while (nu s-a terminat sirul s) do
        if (s[i] este operand) then
                depune s[i] in stiva
```

```
        else    if (s[i] este operator) then
                        extrage din stiva doua simboluri t1 si t2;
                        executa operatia s[i] asupra lui t1 si t2;
                        depune rezultatul in stiva
                else semnalizeaza eroare
                endif
        endif
        i=i+1
endwhile
```

Obs. Algoritmul presupune ca expresiile contin doar operatori binari.

Scrieti un program C (C++) de care sa rezolve problema evaluarii unei expresii date in forma poloneza.

2. O alta forma fara paranteze a unei expresii aritmetice este ***forma prefixata***.

Exemplu: (9-5)+2 are forma prefixata +-952

a. Dati o definitie recursiva pentru forma prefixata a unei expresii aritmetice.

b. Scrieti un program C (C++) de trecere a unei expresii din forma postfixata in forma prefixata.

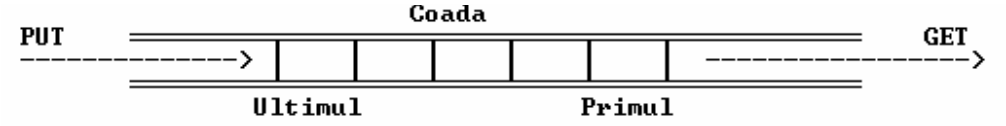
c. Scrieti un program C (C++) de care sa rezolve problema evaluarii unei expresii date in forma prefixata.

Laborator de Structuri de Date si Algoritmi – Lucrarea nr. 5

2. Cozi

2.1. Introducere

O ***coada*** este o lista in care operatiile de acces sint restrictionate la inserarea la un capat si stergerea de la celalat capat.



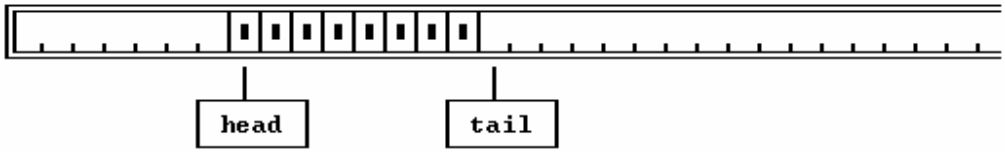
Pricipalele operatii de acces sint:

PUT(Coada, Atom) --> Coada
Adauga un element la coada.
GET(Coada) --> Coada, Atom
Scoate un Atom din coada. Returneaza atomul scos.

2.2. Implementari ale conceptului de coada

[A] Coada ordonata liniara

O coada poate fi organizata pe un spatiu de memorare de tip tablou (vector).



Sint necesari doi indicatori:
head - indica: primul element care urmeaza sa fie scos.
tail - indica: locul unde va fi pus urmatorul element adaugat la coada.

Conditia "coada vida" este echivalenta cu: *head = tail*
Indicatorii vor fi initializati astfel incit sa indice ambii primul element din vector.

Operatia **PUT** inseamna:
- *V[tail]* primeste Atomul adaugat
- incrementeaza *tail*

Operatia **GET** inseamna:
- intoarce *V[head]*
- incrementeaza *head*

Se observa ca adaugari si stergeri repetate in coada deplaseaza continutul cozii la dreapta, fata de inceputul vectorului. Pentru a evita acest lucru ar trebui ca operatia GET sa deplaseze la stinga continutul cozii cu o pozitie. Primul element care urmeaza sa fie scos va fi intotdeauna in prima pozitie, indicatorul *head* pierzindu-si utilitatea. Dezavantajul acestei solutii consta in faptul ca operatia GET necesita o parcurgere a continutului cozii.

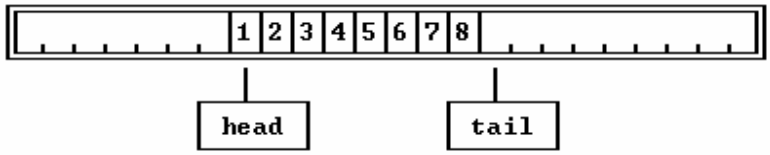
[B] Coada ordonata circulara

Pentru a obtine o coada ordonata circulara vom porni de la o coada liniara ordonata simpla (cu doi indicatori) si vom face in asa fel incit la incrementarea indicatorilor head si tail, cind acestia ating ultima pozitie din vector sa se continue cu prima pozitie din vector. Functia urmatoare poate realizeaza aceasta cerinta:

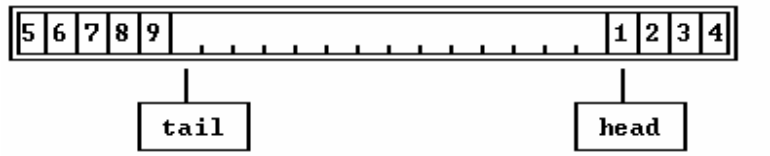
```
int nextPoz(int index)
{
        if (index<DIMVECTOR-1) return index+1;
        else return 0;
}
```

unde DIMVECTOR este dimensiunea vectorului in care se memoreaza elementele cozii.

Continutul cozii va arata asa:



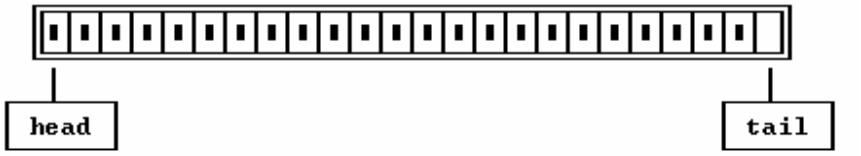
sau asa:



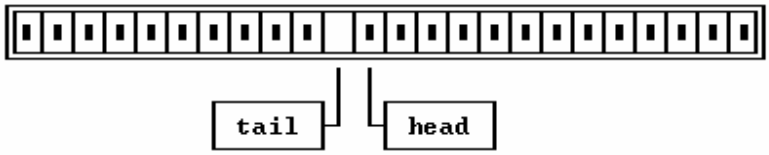
■ Conditia "coada vida" ramine echivalenta cu: *head = tail*

■ Coada va fi plina daca: *head=1* si *tail=DIMVECTOR*

Laborator de Structuri de Date si Algoritmi – Lucrarea nr. 5



sau daca: *tail+1 = head*



Ambele situatii sint continute in conditia:

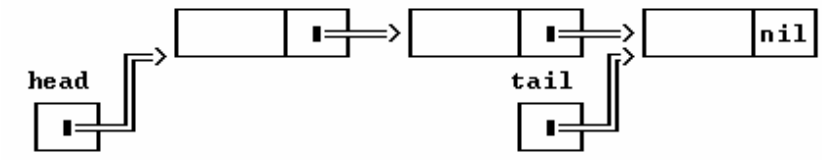
nextPoz(tail) = head // conditia "coada plina"

TEMA 4

Costruiti fisierul QUEUE1.CPP care sa contina implementarea unei cozi circulare, respectind specificatiile din fisierul QUEUE1.H. Testati implementarea realizata.

[C] Coada inlantuita liniara

O coada poate fi implementata printr-o lista liniara simplu inlantuita la care operatiile de acces sint restrictionate corespunzator.



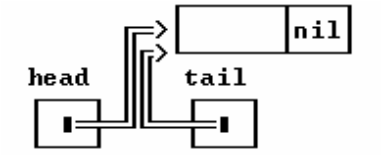
Este nevoie de doi indicatori (pointeri):
head - indica primul element din coada (primul care va fi scos);
tail - indica ultimul element din coada (ultimul introdus).

O coada vida va avea ***head=tail=nil***

In mod obisnuit adaugarea unui element in coada modifica numai *tail* iar stergerea unui element numai *head*. Intr-un mod special trebuie sa fie tratate cazurile:

- *adaugare intr-o coada vida:*

Initial: head=tail=nil
Final: Coada cu un element:



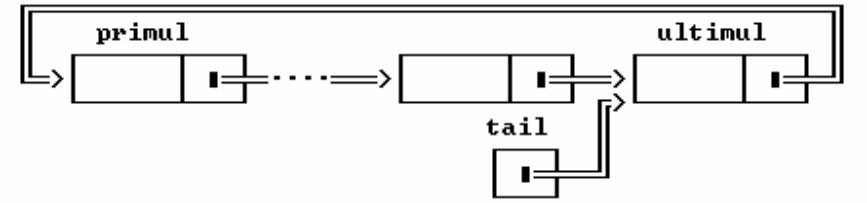
- *stergere dintr-o coada cu un element:*

Initial: Coada cu un element
Final: head=tail=nil

In aceste cazuri se modifica atit *head* cit si *tail*.

[D] Coada inlantuita circulara

Daca reprezentam coada printr-o structura inlantuita circulara va fi nevoie de un singur pointer prin intermediul caruia se pot face ambele operatii de adaugare si stergere din coada:



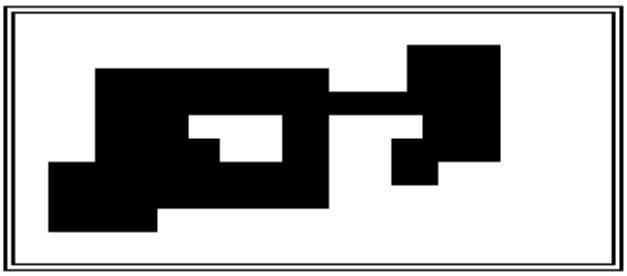
TEMA 5

Construiti fisierele QUEUE2.H si QUEUE2.CPP continind implementarea unei cozi inlantuite (in una din cele doua variante). Testati implementarea realizata.

Problema

Definim "domeniu" ca fiind o zona din ecranul grafic care contine pixeli invecinati direct sau indirect, pe verticala sau orizontala, si care au acelasi culoare. De exemplu:

Laborator de Structuri de Date si Algoritmi – Lucrarea nr. 5



Mai sus sint definite trei domenii:
-cel colorat cu negru;
-cel colorat cu alb, aflat in exterior;
-cel colorat cu alb, in interiorul zonei ngre (insula).
Sa se scrie un program care schimba culoarea unui domeniu, pornind de la un pixel oarecare din acel domeniu.

Schita metodei de rezolvare propuse

Descrierea metodei:
Se foloseste o coada in care se plaseaza coordonate ale unor pixeli.

```
struct Pozitie {
        int x,y;
        };

typedef Pozitie Atom;       // elementele cozii sint de tip Pozitie
```

Se coloreaza si se pune in coada mai intii pozitia pixelului initial. Vom prelucra pe rind elementele scoase din coada. Pentru fiecare element scos din coada se pun in coada si se coloreaza toti vecinii sai care apartin domeniului si nu au fost atinsi (nu au fost colorati). Astfel vor trece prin coada toate pozitiile incluse in domeniu.

Pseudocod:

```
coloreaza_domeniu(pozitie_initiala)
{
 memoreaza culoarea domeniului (culoarea pozitiei initiale)
 put(C, pozitie_initiala);
 coloreaza pozitia_initiala;
 WHILE not isEmpty(C) DO
```

```
p := get(C);    // scoate in p urmatoarea pozitie din coada
FOR pi:=toate pozitiile invecinate lui p DO
   IF pi nu iese in afara ecranului si
        pi are culoarea domeniului THEN
        put(C, pi);
        coloreaza pi;
}
```

Observatie:

Sa incercam sa punem in evidenta, pentru cazul nostru, "vocatia" cozii: adaptarea intre un producator si consumator nesincronizati. In coada folosita in aceasta problema consumatorul este ritmic (la fiecare iteratie se scoate din coada si se prelucreaza o pozitie). Producatorul, datorat plasarii in coada a pozitiilor invecinate, este aritmic (tratarea unei pozitii pune in coada mai multi - cel mult 4 – sau nici un vecin).

TEMA 6

■ Studiati implementarea in limbaj C a acestui algoritm din fisierul COLORARE.CPP .
■ Creati un project care sa includa fisierul COLORARE.CPP si una din implementarile conceptului de coada relizate in acest laborator (QUEUE1.CPP sau QUEUE2.CPP), in care, tipul Atom sa fie echivalat cu Pozitie, si testati algoritmul.

TEMA 7

Coada cu prioritati