**SD – Seminar 6 – LISTE**

**10.11.2020**

n>=0, L = (e\_0, e\_1, …, e\_{n-1}), e\_i apartine unui tip de baza Elt.

Exemplu: lista de caractere: L = (‘i’ ,’n’ , ‘f’, ‘o’, ‘1’).

***I Reprezentare cu tablouri***

struct LLin {

Elt tab[0..Max-1]

int ultim

}

***II Reprezentare cu structuri inlantuite***

struct nod {

Elt inf

nod \* succ

}

struct LLin {

nod \* prim

nod \* ultim

}

**Pr 1.** Inserarea unui element intr-o lista liniara ordonata

Exemplu:

L = (2, 4, 7, 9, 19) , n = 5

insereaza(L, 8) → L = (2, 4, 7, 8, 9, 19)

*I (tablouri)*

**function** insereaza(L, e)

**begin**

**if** (L.utlim == Max – 1) **then** **throw** “memorie insuficienta”

**else** {

k ← L.ultim

**while** (k>=0 and L.tab[k] > e) **do** { L.tab[k+1] ← L.tab[k]; k ← k-1 }

L.tab[k+1] ← e; L.ultim ← L.ultim + 1

return L

}

**end**

*II (structuri inlantuite)*

**function** insereaza(L, e)

**begin**

**if** (L.prim == NULL) or (L.prim != NULL and L.prim→ elt > e) **then** {

**new**(q); q→ inf ← e; q→ succ ← L.prim; L.prim ← q

**if** (L.ultim == NULL) **then** L.ultim ← q }

**else** {

p ← L.prim

**while** (p→succ != NULL and p→ succ→ elt < e) **do** p ← p→ succ

**new**(q); q→ elt ← e; q→ succ ← p→ succ; p→ succ ← q;

**if** (p == L.ultim) **then** L.ultim ← q

}

**return** L

**end**

**Pr. 2** Sa se parcurga in ordine inversa o lista liniara simplu inlantuita, fara a se utiliza un spatiu de memorie suplimentar, proportional cu lungimea listei.

Exemplu: pentru lista L = (‘1’, ‘e’, ‘-’, ‘o’, ‘f’, ‘n’, ‘i’) se va afisa: info-e1.

1. Apel recursiv O(n)

procedure apel\_inv(nod \* p)

begin

if (p->succ != NULL) then apel\_inv(p->succ)

print p->elt

end

1. Dorim sa transformam lista (in ordine inversa)

🡪Discord

**Pr. 3** Sa se concateneze doua liste liniare simplu inlantuite de dimensiuni egale.

Exemplu: Pentru listele L1 = a → b→ c→ d si L2 = x → y→ z→ t se va returna lista

a→ x→ b→ y→ c→ z→ d→ t.

Solutie -> Discord.

**Pr. 4 (tema)**

Se considera liste liniare simplu inlantuite pentru care este posibil ca in urma unei erori la constructia listei, aceasta sa aiba o bucla, i.e., campul succesor al ultimului nod in loc de NULL sa contina adresa unui nod din lista.

Scrieti o functie care primeste la intrare o astfel de lista si returneaza **true**, in cazul in care lista contine o bucla si **false** in caz contrar.

**Pr. 5**

***Stiva***

top(S)

push(S,e)

pop(S)

esteVida(S)

stivaVida()

***Coada***

citeste(C )

insereaza(C, e)

elimina(C )

esteVida(C )

coadaVida(C )

Sa se verifice daca un sir de paranteze ale unei experesii aritmetice se inchid corect.

Exemple:

(()){[]({})} 🡪 true

){} 🡪false

(())({}] 🡪 false

[ 🡪 false

Input: n>=0 lungimea tabloului, char a[0..n-1] un tablou de paranteze

Output: true daca parantezele se inchid corect

false c.c.

function paranteze( int n, char a[0..n-1] )

begin

// TEMA

end

**Pr. 7** Fie doua stive S1 si S2. Implementati operatiile care simuleaza comportamentul unei cozi, folosind doar cele doua stive S1 si S2.

procedure coadaVida( Stiva S1, Stiva S2 )

begin

S1 <- stivaVida()

S2 <- stivaVida()

end

function esteVida( Stiva S1, Stiva S2 )

begin

return esteVida(S1) and esteVida(S2)

end

procedure insereaza( Stiva S1, Stiva S2, Elt e )

begin

push(S1, e)

end

procedure elimina( Stiva S1, Stiva S2 )

begin

if (not esteVida(S2) ) then pop(S2)

else if (esteVida(S1)) then throw “eroare – coada vida”

else {

while (not esteVida(S1)) do { push(S2, top(S1)); pop(S1) }

pop(S2)

}

end

procedure citeste( Stiva S1, Stiva S2 )

begin

// TEMA

end

**Pr. 8**

Se consideră un tablou unidimensional de dimensiune mare care conţine elemente alocate (ocupate) şi

elemente nealocate (disponibile). Ne putem imagina că acest tablou reprezintă memoria unui calculator

(element al tabloului = cuvânt de memorie). Zonele ocupate din tablou sunt gestionate cu ajutorul unei liste implementate cu structuri ı̂nlănţuite. Asupra tabloului se execută următoarele operaţii:

• **aloca(m)** – determină o secvenţă de m elemente succesive nealocate. Adresa şi lungimea acesteia

sunt adăugate ca un nou element la lista zonelor ocupate. Funcţia returnează adresa zonei determinate. Dacă nu se poate aloca o asemenea secvenţă funcţia returnează valoarea −1.

• **elibereaza(a,m)** – disponibilizează secvenţa de lungime m ı̂ncepând cu adresa a. Lista zonelor

ocupate este actualizată corespunzător.

• **defragmenteaza()** – rearanjează zonele ocupate astfel ı̂ncât să existe o singură zona ocupată. Lista zonelor ocupate va fi actualizată corespunzător.

Desigur, din motive de spatiu insuficient, tabloul de dimensiune mare nu va fi retinut ı̂n programul

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| + | \* | + | + | \* | \* | + | + | + |  |

dumneavoastră.

Starea alocarii este gestionata de o lista liniara inlantuita

Max = 10

struct nod {

int a

int l

nod \* succ

}

<a ,l >

<1,1> 🡪 <4,2>

exemple

* aloca(2)
* <1,1> 🡪 <2,2> 🡪<4,2>
* aloca(1)
* <0,1> 🡪 <1,1> 🡪 <2,2> 🡪<4,2>
* aloca(3)
* <0,1> 🡪 <1,1> 🡪 <2,2> 🡪<4,2>🡪<6,3>

<0,9>