Proiect Structuri de Date si Algoritmi

*ENUNT:*

24. TAD ListaOrdonata – implementare folosind o lista dublu inlantuita cu inlanturile reprezentate pe tablou.

*PROBLEMA:*

Luca este pasionat de sport si sanatate. El vrea sa isi mareasca masa musculara prin intermediul unei alimentatii bine stabilite. Acesta are nevoie de un numar mare de calorii in zilele in care depune mult efort fizic si de un numar cat mai mic in zilele in care nu are activitate. Totusi, Luca nu poate consuma mai putin de 1.800 de calorii si mai mult de 3.000 de calorii pe zi. Prin urmare, acesta a hotarat sa faca o aplicatie care sa retina o lista de alimente si cantitatea de calorii pe care o contin acestea. In fiecare zi, Luca specifica tipul zilei (activa, sedentara) si programul selecteaza, dintr-o lista de alimente introdusa de Luca, acele alimente care au suma calorica maxima care nu depaseste 3.000 de calorii in cazul unei zile active, respectiv suma minima, de cel putin 1800 de calorii pentru celelalte zile. De asemenea, acesta poate oricand sa adauge si sa elimine alimente dupa bunul plac din lista totala de alimente.

Lista inlantuita este o structura de date dinamica, o colectie de elemente stocate in locatii numite noduri, a caror ordine este determinata de o legatura continuta in fiecare nod.

In consecinta, s-a stabilit folosirea listei ordonate pentru simplificarea extragerii in ordine crescatoare sau descrescatoare in functie de continutul caloric al fiecarui aliment, in conditiile in care alimentele se gasesc in permanenta intr-o relatie de ordine. Folosirea listei inlantuite este justificata prin faptul ca Luca poate executa un numar mare de adaugari si stergeri schimband oricand, dupa bunul plac, lista din care programul trebuie sa selecteze alimentele (dubla inlantuire va eficientiza extragerea alimentelor de la finalul listei).

*REPREZENTARE:*

Container: lista ordonata

Structura de date: lista inlantuita ( dubla, inlantuiri pe tablou)

*TAD SPECIFICARE SI INTERFATA:*

TAD- LO= lista ordonata, implementare folosind lista dublu inlantuita cu inlantuiri reprezentate pe tablou.

Domeniu: Lo ={ lo / lo este o lista cu elemente de tipul TElement, fiecare element avand o pozitie unica in lo de tipul TPozitie (Iterator) }

*LO:* cp – Intreg (capacitate)

e – TElement[]

urm – Intreg[]

prec – Intreg[]

prim – Intreg

ultim – Intreg

primLiber – Intreg

R – Relatie {TElement x TElement, R relatie de ordine}

R(t1,t2) = adevarat, t1“<= “t2;

fals, altfel;

TPozitie va fi un iterator.

*Iterator:* lo – LO (lista ordonata)

curent – Intreg

*INTERFATA LO:*

* creeaza(lo, cp, R)

{creeaza o lista ordonata vida}

pre: cp Intreg (capacitate listei), R ∈ Relatie {TElement x TElement, R relatie de ordine}

post: lo ∈ LO, lo vida

* distruge(lo)

{distruge lista lo}

pre: lo ∈ LO

post: lo a fost distrusa

* vida(lo)

pre: lo ∈ LO

post: vida = adevarat, lo este lista ordonata vida

vida = fals, altfel

* dim(lo)

pre: lo ∈ LO

post: dim=n numar Natural, n- nr de elemente ale listei lo

* adauga(lo,e)

{adauga un element e in lista lo}

pre: lo ∈ LO, e ∈ TElement

post: lo’ ∈ LO, lo’=lo+{e}, lo ramane ordonata

* sterge(lo,e)

{sterge elementul e din lista lo}

pre: lo ∈ LO, e ∈ TElement

post: : lo’ ∈ LO, lo’=lo\{e}, lo ramane ordonata

* iterator(lo,i)

pre: lo ∈ LO

post: i ∈ I, i este un iterator pe lista ordonata lo

*INTERFATA ITERATOR:*

* creeaza(i,lo)

pre: lo ∈ Lo

post: i ∈ I s-a creat iteratorul i pe lista ordonata lo

* prim(i)

pre: i ∈ I

post: Iteratorul i refera primul element din lista ordonata lo, daca acesta exista

* ultim(i)

pre: i ∈ I

post: Iteratorul i refera ultimul element din lista ordonata lo, daca acesta exista

* valid(i)

pre: i ∈ I

post: valid – true, daca iteratorul i refera un element din lista ordonata lo

valid – false, altfel

* element(i,e)

pre: i ∈ I, i este valid

post: e ∈ TElement, e este elementul curent spre care refera iteratorul i

* urmator(i)

pre: i ∈ I , i este valid

post: Iteratorul i refera urmatorul element din lista ordonata lo fata de elementul anterior pe care il referea, daca acesta exista

* anterior(i)

pre: i ∈ I , i este valid

post: Iteratorul i refera elementul anterior din lista ordonata lo fata de elementul anterior pe care il referea, daca acesta exista

*PSEUDOCOD LISTA ORDONATA:*

Subalgoritm creeaza(lo, cp, R) este:

{ pre: cp Intreg (capacitate listei), R ∈ Relatie {TElement x TElement, R relatie de ordine} }

{ post: lo ∈ LO, lo vida }

lo.prim ← -1

lo.ultim ← -1

lo.cp ← cp

pentru i ← 0, cp-2 executa

lo.urm[i] ← i+1

sf\_pentru

pentru i ← 1, cp-1 executa

lo.prec[i] ← i-1

sf\_pentru

lo.R ← R

lo.urm[cp] ← 0

lo.primLiber ← 1

Sf\_Subalgoritm

Subalgoritm distruge(lo) este:

{pre: lo ∈ LO}

{post: lo a fost distrusa}

i ← lo.ultim

cat timp i ≠ -1 executa

dealoca(i) {in implementare}

i ← lo.prec[i]

sf\_cat timp

lo.prim ← -1

lo.ultim ← -1

Sf\_Subalgoritm

Functia vida(lo) este:

{pre: lo ∈ LO}

{post: vida = adevarat, lo este lista ordonata, vida = fals, altfel}

daca lo.dim = 0 atunci

vida ← adevarat

altfel

vida ← fals

Sf\_Functie

Functia dim(lo) este:

{pre: lo ∈ LO}

{post: dim=n numar Natural, n- nr de elemente ale listei lo}

dim ← lo.dim

Sf\_Functie

Subalgoritm adauga(lo, e) este:

{pre: lo ∈ LO, e ∈ TElement}

{post: lo’ ∈ LO, lo’=lo+{e}, lo ramane ordonata}

i ← creeazaNod(e)

daca i ≠ -1 atunci:

daca lo.vida atunci:

lo.prim ← i

lo.ultim ← i

altfel:

daca lo.R(e, lo.e[lo.prim]) atunci:

lo.urm[i] ← lo.prim

lo.prec[lo.prim] ← i

lo.prim ← i

altfel

daca lo.R(lo.e[lo.ultim], e) atunci:

lo.urm[lo.ultim] ← i

lo.prec[i] ← lo.ultim

lo.ultim ← i

sf\_daca

altfel

j ← lo.prim

cat timp j ≠ -1 si lo.R(lo.e[lo.urm[j]],e) executa:

j ← lo.urm[j]

lo.prec[lo.urm[j]] ← i

lo.urm[i] ← lo.urm[j]

lo.urm[j] ← i

lo.prec[i] ← j

sf\_daca

sf\_daca

lo.dim ← lo.dim +1

sf\_daca

Sf\_Subalgoritm

Subalgoritm sterge(lo, e) este:

{pre: lo ∈ LO, e ∈ TElement}

{post: : lo’ ∈ LO, lo’=lo\{e}, lo ramane ordonata}

daca lo.e[prim] = e si lo.dim() = 1 atunci:

dealoca(lo.prim)

lo.prim ← -1

lo.ultim ← -1

lo.dim ← lo.dim-1

altfel:

daca lo.e[lo.prim] = e atunci:

aux ← lo.urm[lo.prim]

lo.prec[aux] ← -1

dealoca(lo.prim)

lo.prim ← aux

lo.dim ← lo.dim-1

altfel

daca lo.e[lo.ultim] = e atunci:

aux ← lo.prec[lo.ultim]

lo.urm[aux] ← -1

dealoca(lo.ultim)

lo.ultim ← aux

lo.dim ← lo.dim-1

altfel

j ← lo.prim

cat timp j ≠ -1 si lo.e[lo.urm[j]] ≠ e executa:

j ← lo.urm[j]

daca j ≠ -1 atunci:

aux ← lo.urm[j]

lo.prec[lo.urm[aux]] ← j

lo.urm[j] ← lo.urm[aux]

dealoca(aux)

lo.dim ← lo.dim-1

sf\_daca

sf\_daca

sf\_daca

sf\_daca

Sf\_Subalgoritm

Subalgoritm iterator(lo,i)

{pre: lo ∈ LO}

{post: i ∈ I, i este un iterator pe lista ordonata lo}

creeaza(i,lo)

Sf\_Subalgoritm

Functia aloca(lo) este:

i ← lo.primLiber

lo.primLiber ← lo.urm[lo.peimLiber]

aloca ← i

Sf\_Functie

Subalgoritm dealoca(lo,i) este:

lo.urm[i] ← lo.primLiber

lo.primLiber ← i

Sf\_Subalgoritm

Functia creeazaNod(lo,e)

i ← aloca(lo)

daca i ≠ -1 atunci:

lo.e[i] ← e

lo.urm[i] ← -1

lo.prec[i] ← -1

sf\_daca

creeazaNod ← i

Sf\_Functie

*PSEUDOCOD ITERATOR:*

Subalgoritm creeaza(i,lo) este:

{pre: lo ∈ Lo}

{post: i ∈ I s-a creat iteratorul i pe lista ordonata lo}

i.lo ← lo

i.curent ← lo.prim

Sf\_Subalgoritm

Subalgoritm prim(i) este:

{pre: i ∈ I }

{post: Iteratorul i refera primul element din lista ordonata lo, daca acesta exista}

i.curent ← i.lo.prim

Sf\_Subalgoritm

Subalgoritm ultim(i) este:

{pre: i ∈ I }

{post: Iteratorul i refera ultimul element din lista ordonata lo, daca acesta exista}

i.curent ← i.lo.ultim

Sf\_Subalgoritm

Functia valid(i) este:

{pre: i ∈ I }

{post: valid – true, daca iteratorul i refera un element din lista ordonata lo,valid – false, altfel}

daca i.curent ≠ -1 atunci:

valid ← adevarat

altfel

valid ← fals

sf\_daca

Sf\_Functie

Functia element(i) este:

{pre: i ∈ I, i este valid}

{post: e ∈ TElement, e este elementul curent spre care refera iteratorul i}

element ← i.lo.e[i.curent]

Sf\_Functie

Subalgoritm urmator(i) este:

{pre: i ∈ I , i este valid}

{post: Iteratorul i refera urmatorul element din lista ordonata lo fata de elementul anterior pe {care il referea, daca acesta exista}

i.curent ← i.lo.urm[i.curent]

Sf\_Subalgoritm

Subalgoritm anterior(i) este:

{pre: i ∈ I , i este valid}

{post: Iteratorul i refera elementul anterior din lista ordonata lo fata de elementul anterior pe {care il referea, daca acesta exista}

i.curent ← i.lo.prec[i.curent]

Sf\_Subalgoritm

*COMPLEXITATE OPERATII DIN INTERFATA:*

*Lista Ordonata:*

* creeaza(lo, cp, R) Θ(cp), cp- capacitatea listei ordonate
* distruge(lo) Θ(dim), dim- dimensiunea listei ordonate
* vida(lo) Θ(1)
* dim(lo) Θ(1)
* adauga(lo,e) Θ(dim), dim- dimensiunea listei ordonate
* sterge(lo,e) Θ(dim), dim- dimensiunea listei ordonate
* iterator(lo,i) Θ(1)
* creeazaNod(lo,e) Θ(1) amortizat

*Iterator:*

* creeaza(i,lo) Θ(1)
* prim(i) Θ(1)
* ultim(i) Θ(1)
* valid(i) Θ(1)
* element(i,e) Θ(1)
* urmator(i) Θ(1)
* anterior(i) Θ(1)

*Deductie complexitate adauga din interfata listei ordonate:*

Caz favorabil: cand adaugam pe prima sau pe ultima pozitie din lista ordonata( se cunoaste atat prima cat si ultima pozitie din lista ordonata), timp constant Θ(1).

Caz defavorabil: cand adaugam pe penultima pozitie din lista ordonata, se itereaza toate elementele listei ordonate pana se ajunge la penultima pozitie, deci avem Θ(dim-1) adica Θ(dim), unde dim este dimensiunea listei ordonate.

Caz mediu: Bucla cat timp se poate repeta o data, de doua ori, ... , de dim-1 ori (unde dim este dimensiunea listei ordonate):

T(dim) = 1/dim + 2/dim + 3/dim + ... + (dim-1)/dim

T(dim) = ( (dim-1)\*dim ) / ( 2\*dim )

T(dim) = (dim-1) / 2

T(dim) = 1/2 \*dim -1/2 => T(dim) ∈ Θ(dim)

In concluzie, complexitatea operatiei de adaugare este Θ(dim).

*DIAGRAMA DE APELURI:*

APLICATIE

prim()

urmator()

anterior()

ultim()

vida()

valid()

iterator()

element()

tiparesteLista(lo,nr)

tiparesteListaInv(lo,nr)

ziSedentara(lo)

ziActiva(lo)

sterge(nume,cal)

dim()

ui\_sterge()

ui\_adauga()

adauga(nume,cal)

distruge(lo)

creeazaLista(lo)

meniu()

*PROIECTARE APLICATIE:*

Aliment:

nume: string

calorii: real

Algoritm main este:

{algoritmul principal}

creeazaLIsta(lo)

cat timp adevarat executa:

incearca

meniu()

@citeste comanda com

daca com=1 atunci:

ui\_adauga()

sf\_daca

daca com=2 atunci:

ui\_sterge()

sf\_daca

daca com=3 atunci:

ziSedentara(lo)

sf\_daca

daca com=4 atunci:

ziActiva(lo)

sf\_daca

daca com=5 atunci:

d ← dim()

tiparesteLista(lo,d)

sf\_daca

daca com=0 atunci:

@terminare aplicatie

sf\_daca

@prinde exceptiile si afiseaza pe ecran

Sf\_Algoritm

Subalgoritm ui\_adauga() este:

{citeste si adauga un nou aliment}

@citeste nume si calorii

adauga(nume, cal)

Sf\_Subalgoritm

Subalgoritm adauga(nume, cal) este:

{adauga un nou aliment}

{Pre: nume numele alimentului, cal nr de calorii}

@creeaza alimentul a cu nume si cal

adauga(lo,a)

Sf\_Subalgoritm

Subalgoritm sterge(nume, cal) este:

{sterge un aliment}

{Pre: nume numele alimentului, cal nr de calorii}

@creeaza alimentul a cu nume si cal

sterge(lo,a)

Sf\_Subalgoritm

Functia dim() este:

dim ← dim(lo)

Sf\_Functie

Subalgoritm ui\_sterge() este:

{citeste si sterge elementul citit}

@citeste nume si calorii

dimVechi ← dim()

sterge(nume, cal)

daca dimVechi = dim() atunci:

@afiseaza Nu a fost gasit produsul de sters!

Sf\_Subalgoritm

Subalgoritm meniu() este:

@afiseaza meniul comenzilor

Sf\_Subalgoritm

Subalgoritm creeazaLista(lo) este:

{adauga cateva elemente in lista ordonata}

{Post: lo lista ordonata}

@adauga cateva elemente

@creeaza Alimentul a

adauga(a)

Sf\_Subalgoritm

Subalgoritm tiparesteLista(lo,nr)

{tipareste un nr de alimente din lista ordonata pe ecran}

{Post: lo lista ordonata, nr nr de alimente}

daca vida(lo) atunci:

@afiseaza Lista este vida!

altfel

iterator(lo,it)

prim(it)

cat timp valid(it) si nr>0 executa:

@afiseaza element(it)

nr ← nr-1

urmator(it)

sf\_cat timp

sf\_daca

Sf\_Subalgoritm

Subalgoritm tiparesteListaInv(lo,nr)

{tipareste in ordine inversa un nr de alimente din lista ordonata pe ecran}

{Post: lo lista ordonata, nr nr de alimente}

daca vida(lo) atunci:

@afiseaza Lista este vida!

altfel

iterator(lo,it)

ultim(it)

cat timp valid(it) si nr>0 executa:

@afiseaza element(it)

nr ← nr-1

anterior(it)

sf\_cat timp

sf\_daca

Sf\_Subalgoritm

Subalgoritm ziSedentara(lo) este:

{afiseaza meniul pentru o zi sedentara}

{Post: lo lista ordonata de alimente}

daca vida(lo) atunci:

@afiseaza Lista este vida!

altfel

total ← 0

nrAlimente ← 0

iterator(lo,it)

prim(it)

cat timp valid(it) si total>1800 executa:

nrAlimente ← nrAlimente+1

total ← total+ @nr de calorii ale alimentului element(it)

urmator(it)

sf\_cat timp

daca total<1800 sau nrAlimente=0 atunci:

@afiseaza Alimente insuficiente!

altfel

tiparesteLista(lo,nrAlimente)

sf\_daca

sf\_daca

Sf\_Subalgoritm

Subalgoritm ziActiva(lo) este:

{afiseaza meniul pentru o zi activa}

{Post: lo lista ordonata de alimente}

daca vida(lo) atunci:

@afiseaza Lista este vida!

altfel

total ← 0

nrAlimente ← 0

iterator(lo,it)

ultim(it)

cat timp valid(it) si total+@nr de calorii ale alimentului element(it)<=3000 executa:

nrAlimente ← nrAlimente+1

total ← total+@nr de calorii ale alimentului element(it)

anterior(it)

sf\_cat timp

daca nrAlimente=0 atunci:

@afiseaza Alimente insuficiente!

altfel

tiparesteListaInv(lo,nrAlimente)

sf\_daca

sf\_daca

Sf\_Subalgoritm

Functia R(a1,a2) este:

{functie de comparare pentru lista ordonata}

{Pre: a1 si a2 sunt obiecte de tip Aliment}

daca @a1 are mai putine calorii decat a2 atunci:

R ← adevarat

altfel

R ← fals

sf\_daca

Sf\_Functie

*COMPLEXITATE OPERATII DIN APLICATIE:*

* ui\_adauga() Θ(dim), dim- dimensiunea listei ordonate
* ui\_sterge() Θ(dim), dim- dimensiunea listei ordonate
* adauga(nume,cal) Θ(dim), dim- dimensiunea listei ordonate
* sterge(nume,cal) Θ(dim), dim- dimensiunea listei ordonate
* dim() Θ(1)
* meniu() Θ(1)
* creeazaLista(lo) Θ(1)
* tiparesteLista(lo,nr) Θ(nr), nr- numarul de elemnte care se doresc a fi afisate
* tiparesteListaInv(lo,nr) Θ(nr), nr- numarul de elemnte care se doresc a fi afisate
* ziSedentara(lo) Θ(dim), dim- dimensiunea listei ordonate
* ziActiva(lo) Θ(dim), dim- dimensiunea listei ordonate
* main() Θ(1)
* R(a1,a2) Θ(1)