Facultatea de Matematică și Informatică
Universitatea Babeș-Bolyai

Structuri de date și algoritmi

Documentație proiect

Enăchioiu Teofana
Grupa 213

Cuprins

1. Cuprins
2. Enunțul problemei
3. TAD – specificare și interfață
4. Reprezentarea TAD-ului
5. Implementarea operațiilor din interfața TAD-ului
6. Complexitate operații pseudocod
7. Diagrama de apeluri
8. Proiectarea aplicației
9. Complexitatea operațiilor din aplicație

Enunțul problemei

Administrare cămin privat

Să se creeze o aplicație care permite gestiunea camerelor închiriate într-un cămin privat. Fiecare cameră este unic identificată prin numărul său. Pentru o cameră se va reține: numele persoanei ce a efectuat închirierea și suma de plată pentru chirie.

Aplicația permite administratorului să efectueze următoarele operații:

- Închiriere cameră
- Efectuarea unei plăți
- Adăugare suma de plată
- Eliberare cameră
- Vizualizarea camere ce au plata chiriei neefectuate
- Vizualizarea camerelor ce sunt la zi cu plata chiriei
- Vizualizare camere închiriate

Utilitatea folosirii TAD Dicționar Ordonat

Aplicația *Administrare cămin privat* a fi implementata cu ajutorul TAD Dicționar Ordonat. Acest tip abstract de dată este potrivit pentru rezolvarea problemei deoarece, într-un cămin, camerele se memorează în ordine crescătoarea a numărul lor.

Așadar, cheia TAD-ului va fi numărul camerei, iar valoarea TAD-ului va fi clasa Cameră.

Clasa Camera
- nr_cam (intreg)
- nume_chirias (string)
- pret_chirie (intreg)

TAD – specificare și interfață

Specificarea Tipului de Date Abstract – **Dicționar Ordonat**

Domeniu:

D={d | d este un dicţionar cu elemente e = (c, v), c de tip TCheie, v de tip TComparabil, elemente aflate intr-o ordine data de o relație R}

Operaţii:

- **creează(d, R)**

pre: -

post: d∈D, d este dicţionarul vid (fără elemente), R relație de ordine

- **adaugă(d, R, c, v)**

{perechea (c,v) se adauga respectând relația de ordine R dată}

pre: d∈D, c∈TCheie, v∈TComparabil,R relație de ordine

post: d'∈D, d'=d+(c, v) (se adaugă în dicţionar elementul e=(c, v))

- **caută(d, c, v)**

pre: d∈D, c∈TCheie

post:

$$cauta = \begin{cases} adevărat\ dacă\ (c,v) \in d, caz\ în\ care\ v \in TComparabil\ e\ valoarea\ asociată\ cheii\ c \\ fals, în\ caz\ contrar, caz\ în\ care\ v = TComparabil_0 \end{cases}$$

- **şterge(d, c, v)**

pre: d∈D, c∈TCheie

post: v∈TComparabil (perechea (c, v) este ştearsă din dicţionar, dacă c∈d) sau v= TComparabil$_0$ în caz contrar

- **dim(d)**

pre: d∈D

*post: dim= dimensiunea dicţionarului d (numărul de elemente), dim∈N**

- **vid(d)**

pre: d∈D

post: $vid = \begin{cases} adevărat, în\ cazul\ în\ care\ d\ este\ dicţionarul\ vid \\ fals, în\ caz\ contrar \end{cases}$

- **chei(d, m)**

pre: d ∈D

post: m ∈M, m este mulţimea cheilor din dicţionarul d

- **valori(d, c)**

pre: d ∈D

post: c ∈Col, c este colecţia valorilor din dicţionarul d

- **perechi(d, m)**

pre: d ∈D

post: m ∈M, m este mulţimea perechilor (cheie, valoare) din dicţionarul d

- **iterator(d, i)**

{se creează un iterator pe dicţionarul d}

pre: d ∈D

post:i ∈ I, i este iterator pe dicţionarul d

- **distruge(d)**

{spaţiul de memorie alocat este eliberat}

pre: d ∈ D

post: dicţionarul d a fost distrus

Specificarea Tipului de Date Abstract – **Iterator Dicționar Ordonat**

Domeniu:

I = {i | i este un iterator pe un dicționar avand elemente de tip TElement}

Operaţii:

- **creeazaIterator(i,d)**

pre: d este un dicționar ordonat

post: i ∈I s-a creat iteratorul i pe dicționarul d

- **prim(i)**

pre: i ∈ I

post: curent referă primul element din dicționar

- **element(i,e)**

pre: i ∈I, curent e valid(referă un element din dicționar)

post: e∈TElement, e elementul curent din iterație (elementul referit de curent)

- **valid(i)**

pre: i∈I

post: $valid = \begin{cases} adevarat, daca\ curent\ refer\check{a}\ o\ pozi\c{t}ie\ valid\check{a}\ din\ dictionar \\ fals, \hat{i}n\ caz\ contrat \end{cases}$

- **următor(i)**

pre: i∈*I*

post: curent refera următorul element din container față de cel referit de curent

Reprezentarea TAD-ului

Nod:
e: TElement
urm: ↑Nod
prec: ↑Nod

Dicționar:
e: TElement
R: Relație de ordine ($R(c_1, c_2) = \begin{cases} adevărat, dacă\ c_1 < c_2 (c_1\ este\ plasat\ înaintea\ lui\ c_2) \\ fals, în\ caz\ contrar \end{cases}$)
prim: ↑Nod
ultim: ↑Nod
size: intreg

TElement:
c: TCheie
v: TComparabil

Iterator:
d: Dicționar
curent: ↑Nod

Implementarea operațiilor din interfață

Funcții suplimentare:

```
funcția creeazăNod (c,v) este
        {returnează un pointer la Nod}
        aloca(p)
        {se aloca spațiu de memorie necesar reprezentării}
        [p].e.c←c
        [p].e.v←v
        creeazăNod←p
sfârșit creeazăNod
```

Operații:

```
subalgoritm creează(d,R) este
        {se creează dicționarul nul}
        d.R←R
        d.prim←NIL
        d.ultim←NIL
        d.size←0
sfârșit creează

subalgoritm adaugă(d,R,c,v) este
        {se adaugă un nou element în dicționar respectând relația dată}
        q←d.prim
        cât timp (q!=NIL) și (c != [q].e.c) execută
                q←[q].urm
        sfârșit cât timp
        daca q!=NIL atunci
                p←creeazăNod(c,v)
                size←size+1
                daca d.prim=NIL atunci
                        {dictionarul este vid}
                        d.prim←p
                        d.ultim←p
                        @iesire din subalgoritm
```

```
            sfârșit dacă
            daca d.R(e.c, [d.prim].e.c) atunci
                  {elementul trebuie inserat pe prima poziție}
                  [p].urm←d.prim
                  [d.prim].prec←p
                  d.prim←p
                  @iesire din subalgoritm
            sfârșit dacă
            dacă ¬d.R(e.c, [d.ultim].e.c) atunci
                  {elementul trebuie inserat la sfarsit}
                  [d.ultim].urm←p
                  [p].prec←d.ultim
                  d.ultim←p
                  @iesire din subalgoritm
            sfârșit dacă
            {căutam poziția pe care trebuie adăugată cheia}
            r← d.prim
            cât timp (r!=NIL) si ¬( d.R(e.c, [r].e.c)) execută
                  r←[r].urm
            sfârșit cât timp
            [p].prec←[r].prec
            [p].urm←r
            [[r].prec].urm←p
            [r].prec←p
      sfârșit dacă
Sfârșit adaugă

functie caută(d,c,v) este
      {se caută un element în dicționar}
      p←d.prim
      cât timp (p!=NIL) și (c != [p].e.c) execută
            p←[p].urm
      sfârșit cât timp
      dacă p=NIL atunci
            @v ia valoarea TValoare0
            caută←fals
      altfel
            v←[p].e.v
```

```
            caută←adevarat
        sfârșit dacă
sfârșit caută

subalgoritm sterge(d,c,v) este
        {se sterge un element în dicționar}
        p←d.prim
        daca [p].e.c=c atunci
            {elementul se afla primul in dictionar}
            d.prim← [d.prim].urm
            daca d.prim=NIL atunci
                {dictionarul continea o singura valoare}
                d.ultim←NIL
            altfel
                [d.prim].prec←NIL
            Sfarsit daca
            dealoca(p)
            size←size-1
            @iesire din subalgormitm
        Sfarsit daca
        cât timp (p!=NIL) și (c != [p].e.c) execută
            p←[p].urm
        sfârșit cât timp
        {stergem elementul dinaintea lui p}
        dacă p=d.ultim atunci
            d.ultim← [d.ultim].prec
            [d.ultim].urm← [p].prec
        altfel
            [[p].prec].urm←[p].urm
            [[p].urm].prec←[p].prec
        sfârșit dacă
        dealoca(p)
        size←size-1
sfârșit sterge

functie dim(d) este
        {returneaza dimensiunea dictionarului}
        dim←size
```

```
sfârșit dim


functie vid(d) este
        {indica faptul ca dictionarul e vid sau nu}
        daca size=0 atunci
                dim←adevarat
        altfel
                dim←fals
        sfarsit daca
sfârșit vid

subalgoritm chei(d,m) este
        {m este multimea cheilor}
        Iterator(d,i)
        Prim(i)
        Cat timp valid(i) executa
                element(i,e)
                daca ¬cauta(m,e) atunci
                        adauga(m,e.c)
                sfarsit daca
                uramtor(i)
        sfarsit cat timp
sfârșit chei

subalgoritm valori(d,c) este
        {c este colectia valorilor}
        Iterator(d,i)
        Prim(i)
        Cat timp valid(i) executa
                element(i,e)
                adauga(c,e.v)
                uramtor(i)
        sfarsit cat timp
sfârșit valori

subalgoritm perechi(d,m) este
        {m este multimea perechilor din dictionar}
        Iterator(d,i)
```

```
        Prim(i)
        Cat timp valid(i) executa
                element(i,e)
                adauga(m,e)
                uramtor(i)
        sfarsit cat timp
sfârșit perechi

subalgoritm iterator(d,i) este
        {i este iterator pe dictionar}
        creeazaIterator(i,d)
sfârșit iterator

subalgoritm distruge(d) este
        {se dealoca spatiu de memorie rezervat elementelor din dictionar}
        Cat timp d.prim!=NIL executa
                p←d.prim
                [d.prim].prec←NIL
                d.prim←[d.prim].urm
                dealoca(p)
        Sfarsit cat timp
sfârșit distruge
```

Operatii pe iterator:

```
subalgoritm creeazaIterator(i,d) este
        {se creaza iterator pe dictionarul d}
        i.d←d
sfârșit creeazaIterator

subalgoritm prim(i) este
        {refera primul element din dictionar}
        i.curent←d.prim
sfârșit prim

functie valid(i) este
        {se verifica daca pozitia curenta este o pozitie in dictionarul d}
        daca i.curent=NIL atunci
                valid←fals
```

```
        altfel
                valid←adevarat
        sfarsit daca
sfârșit valid

subalgoritm urmator(i) este
        {se trece la urmatoarea pozitie in dictionarul d}
        i.curent←[i.curent].urm
sfârșit urmator

subalgoritm element(i,e) este
        {e este elementul de pe pozitia curenta}
        e←[i.curent].e
sfârșit element
```

Pentru implementarea dictionarului ordonat este nevoie, de asemanea, de TAD-ul Multime si TAD-ul Colectie. Operatiile necesare din aceste TAD-uri sunt adaugarea si cautarea in Multime, respectiv, adaugarea in Colectie.

TAD Colectie

Specificarea Tipului de Date Abstract – **Colectie**

Domeniu:

C={c | c este o colectie cu elemente de tip TValoare}

Operaţii:

- **creează(c)**

pre: -
post: c∈C, c este colectia vida (fără elemente)

- **adaugă(c, v)**

{elementul v se adauga in colectie}
pre: c∈C
post: c'∈C, c'=c+ v (se adaugă în colectie elementul e)

- **dim(c)**

pre: c∈C
*post: dim= dimensiunea colectiei c (numărul de elemente), dim∈N**

- **iterator(c, i)**

{se creează un iterator pe colectia c}
pre: c∈C
post:i∈ I, i este iterator pe colectia C

- **distruge(c)**

{spaţiul de memorie alocat este eliberat}
pre: c∈ C
post: colectia c a fost distrusa

Specificarea Tipului de Date Abstract – **Iterator Colectie**

Domeniu:

I = {i | i este un iterator pe o colectie avand elemente de tip TValoare}

Operaţii:

- **creeazaIterator(i,c)**

pre: c este o colectie

post: i∈I s-a creat iteratorul i pe colectia c

- **prim(i)**

pre: i∈ I

post: curent referă primul element din colectie

- **element(i,e)**

pre: i∈I, curent e valid(referă un element din colectie)

post: e∈TValoare, e elementul curent din iterație (elementul referit de curent)

- **valid(i)**

pre: i∈I

post: $valid = \begin{cases} adevarat, daca\ curent\ refer\check{a}\ o\ pozi\c{t}ie\ valid\check{a}\ din\ colectie \\ fals, \hat{i}n\ caz\ contrat \end{cases}$

- **următor(i)**

pre: i∈*I*

post: curent refera următorul element din container față de cel referit de curent

Reprezentarea Tipului de Date Abstract – **Colectie**

Nod:
e: TValoare
urm: ↑Nod
prec: ↑Nod

Colectie:
e: TValoare
prim: ↑Nod
ultim: ↑Nod
size: intreg

Iterator:
c: Colectie
curent: ↑Nod

Implementarea operatiilor Tipului de Date Abstract – **Colectie**

Funcții suplimentare:

```
funcția  creeazăNod (e) este
        {returnează un pointer la Nod}
        aloca(p)
        {se aloca spațiu de memorie necesar reprezentării}
        [p].e.←e
        creeazăNod←p
sfârșit creeazăNod
```

Operații:

```
subalgoritm creează(c) este
        {se creează colectia nula}
        c.prim←NIL
        c.ultim←NIL
        c.size←0
sfârșit creează

subalgoritm adauga(v) este
        {se adaugă un nou element în colectie}
        p←creeazăNod(v)
        size←size+1
        daca c.prim=NIL atunci
                {colectia este vida}
                c.prim←p
                c.ultim←p
        altfel
                {inserez elementul la sfarsit}
                [c.ultim].urm←p
                [p].prec←c.ultim
                c.ultim ←p
        sfârșit dacă
Sfârșit adaugă

functie dim(c) este
        {returneaza dimensiunea colectiei}
        dim←size
```

```
sfârșit dim

subalgoritm iterator(c,i) este
        {i este iterator pe colectie}
        creeazaIterator(i,c)
sfârșit iterator

subalgoritm distruge(c) este
        {se dealoca spatiu de memorie rezervat elementelor din colectie}
        Cat timp c.prim!=NIL executa
                p←c.prim
                [c.prim].prec←NIL
                c.prim←[c.prim].urm
                dealoca(p)
        Sfarsit cat timp
sfârșit distruge
```

Operatii pe iterator:

```
subalgoritm creeazaIterator(i,c) este
        {se creaza iterator pe colectia c}
        i.c←c
sfârșit creeazaIterator

subalgoritm prim(i) este
        {refera primul element din colectie}
        i.curent←c.prim
sfârșit prim

functie valid(i) este
        {se verifica daca pozitia curenta este o pozitie in colectia c}
        daca i.curent=NIL atunci
                valid←fals
        altfel
                valid←adevarat
        sfarsit daca
sfârșit valid

subalgoritm urmator(i) este
```

```
        {se trece la urmatoarea pozitie in colectia c}
        i.curent←[i.curent].urm
sfârșit urmator

subalgoritm element(i,e) este
        {e este elementul de pe pozitia curenta}
        e←[i.curent].e
sfârșit element
```

TAD Multime

Specificarea Tipului de Date Abstract – **Multime**

Domeniu:

M={m | m este o multime cu elemente de tip TValoare}

Operaţii:

- **creează(m)**

pre: -
post: m∈M, m este multimea vida (fără elemente)

- **adaugă(m, v)**

{elementul v se adauga in multime}
pre: c∈C
post: c'∈C, c'=c+ v (se adaugă în multime elementul e)

- **caută(m, v)**

pre: c∈C, c∈TValoare
post: $cauta = \begin{cases} adevărat, dacă\ v \in d \\ fals, în\ caz\ contrar \end{cases}$

- **dim(m)**

pre: m∈M
*post: dim= dimensiunea multimii m (numărul de elemente), dim∈N**

- **iterator(m, i)**

{se creează un iterator pe multimea m}
pre: m∈M
post:i∈ I, i este iterator pe multimea M

- **distruge(m)**

{spaţiul de memorie alocat este eliberat}
pre: m∈ M
post: multimea m a fost distrusa

Specificarea Tipului de Date Abstract – **Iterator Multime**

Domeniu:

I = {i | i este un iterator pe o multime avand elemente de tip TValoare}

Operaţii:

- **creeazaIterator(i,m)**

pre: m este o multime
post: i∈I s-a creat iteratorul i pe multimea m

- **prim(i)**

pre: i∈ I
post: curent referă primul element din multime

- **element(i,e)**

pre: i∈I, curent e valid(referă un element din multime)
post: e∈TValoare, e elementul curent din iterație (elementul referit de curent)

- **valid(i)**

pre: i∈I
post: $valid = \begin{cases} adevarat, daca\ curent\ referă\ o\ poziţie\ validă\ din\ multime \\ fals, în\ caz\ contrar \end{cases}$

- **următor(i)**

pre: i∈*I*
post: curent refera următorul element din container față de cel referit de curent

Reprezentarea Tipului de Date Abstract – **Multime**

Nod:
e: TValoare
urm: ↑Nod
prec: ↑Nod

Multime:
e: TValoare
prim: ↑Nod
ultim: ↑Nod
size: intreg

Iterator:
m: Multime
curent: ↑Nod

Implementarea operatiilor Tipului de Date Abstract – **Multime**

Funcții suplimentare:

```
funcția  creeazăNod (e) este
        {returnează un pointer la Nod}
        aloca(p)
        {se aloca spațiu de memorie necesar reprezentării}
        [p].e.←e
        creeazăNod←p
sfârșit creeazăNod
```

Operații:

```
subalgoritm creează(m) este
        {se creează multimea nula}
        m.prim←NIL
        m.ultim←NIL
        m.size←0
sfârșit creează

subalgoritm adauga(v) este
        {se adaugă un nou element în multime}
        p←creeazăNod(v)
        p←m.prim
        cat timp p!=NIL si v!=[p].e
                p←[p].urm
        daca p!=NIL atunci
                size←size+1
                daca m.prim=NIL atunci
                        {multimea este vida}
                        m.prim←p
                        m.ultim←p
                altfel
                        {inserez elementul la sfarsit}
                        [m.ultim].urm←p
                        [p].prec←m.ultim
                        m.ultim ←p
                sfârșit dacă
```

```
        sfârșit dacă
Sfârșit adaugă

functie cauta(v) este
        {se cauta valoarea v in multime}
        p←m.prim
        cat timp p!=NIL si v!=[p].e
                p←[p].urm
        daca p!=NIL atunci
                cauta←adevarat
        altfel
                cauta←fals
        sfarsit daca
Sfârșit cauta

functie dim(m) este
        {returneaza dimensiunea multimii}
        dim←size
sfârșit dim

subalgoritm iterator(m,i) este
        {i este iterator pe multime}
        creeazaIterator(i,m)
sfârșit iterator

subalgoritm distruge(m) este
        {se dealoca spatiu de memorie rezervat elementelor din multime}
        Cat timp m.prim!=NIL executa
                p←m.prim
                [m.prim].prec←NIL
                m.prim←[m.prim].urm
                dealoca(p)
        Sfarsit cat timp
sfârșit distruge
```

Operatii pe iterator:

```
subalgoritm creeazaIterator(i,m) este
        {se creaza iterator pe multimea m}
```

```
        i.m←m
sfârșit creeazaIterator

subalgoritm prim(i) este
        {refera primul element din multime}
        i.curent←m.prim
sfârșit prim

functie valid(i) este
        {se verifica daca pozitia curenta este o pozitie in multimea m}
        daca i.curent=NIL atunci
                valid←fals
        altfel
                valid←adevarat
sfârșit valid

subalgoritm urmator(i) este
        {se trece la urmatoarea pozitie in multimea m}
        i.curent←[i.curent].urm
sfârșit urmator

subalgoritm element(i,e) este
        {e este elementul de pe pozitia curenta}
        e←[i.curent].e
sfârșit element
```

Complexitate operații pseudocod

Complexitate operatii TAD Dictionar Ordonat

- creează(d, R) {**θ(1)**}
- adaugă(d, R, c, v) {**O(n)**}
- caută(d, c, v) {**O(n)**}
- şterge(d, c, v) {**O(n)**}
- dim(d) {**θ(1)**}
- vid(d) {**θ(1)**}
- chei(d, m) {**θ (n)**}
- valori(d, c) {**θ (n)**}
- perechi(d, m) {**θ (n)**}
- iterator(d, i) {**θ(1)**}
- distruge(d) {**θ (n)**}

Deductie complexitate pentru operatia „adauga”:

Caz favorabil: *elementul trebuie inserat inaintea primului element din dictionar sau dupa ultimul element*

$T(n)=1 \in \theta(1)$

Caz defavorabil: *elementul trebuie inserat inaintea unui element din dictionar*

$$T(n) = \sum_{i=1}^{n} 1 = n \in \theta(n)$$

Caz mediu:

$$T(n) = \sum_{i=1}^{n} \frac{i}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2} \in \theta(n)$$

Complexitate operatii TAD Colectie

- creează(c) {**θ(1)**}
- adaugă(c,v) {**θ(1)**}
- dim(c) {**θ(1)**}
- iterator(c, i) {**θ(1)**}
- distruge(c) {**θ (n)**}

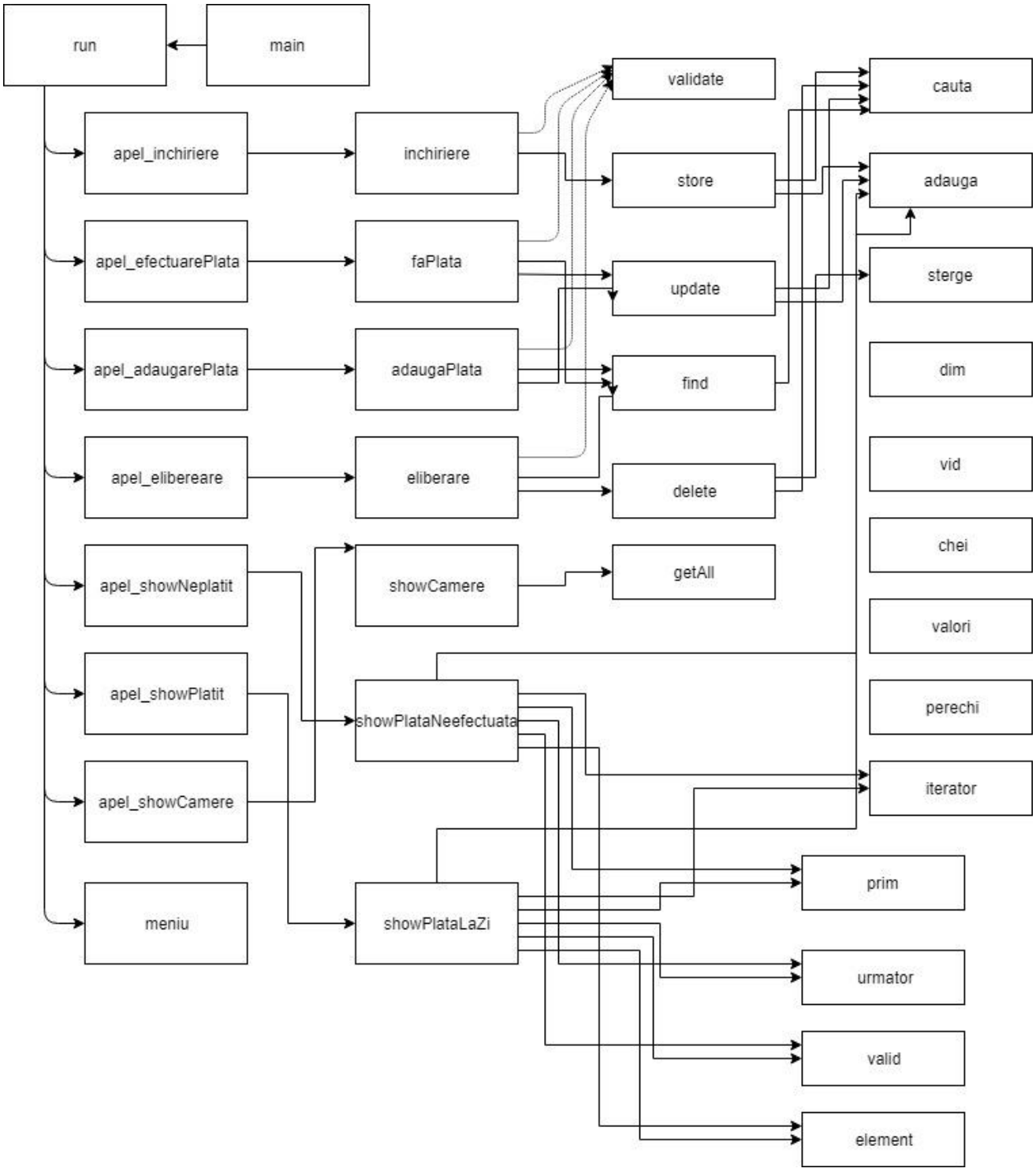
Complexitate operatii TAD Multime

- creează(c) {**θ(1)**}
- adaugă(c,v) {**O(n)**}
- cauta(c,v) {**O(n)**}
- dim(c) {**θ(1)**}
- iterator(c, i) {**θ(1)**}
- distruge(c) {**θ (n)**}

Complexitate operatii Iterator (Dictionar Ordonat, Colectie, Multime)

- creeazaIterator(i,c) {**θ(1)**}
- prim(i) {**θ(1)**}
- element(i,e) {**θ(1)**}
- valid(i) {**θ(1)**}
- următor(i) {**θ(1)**}

Diagrama de apeluri



Proiectarea aplicației

Repository

Clasa Repository
- **DictionarOrdonat** *repo* - **string** *fisierIn, fisierOut*
- store(*Camera cam*) - delete(*Camera cam*) - update(*Camera cam*) - find(*Camera cam*)

subalgoritm **store(repo, cam)** este

{elementul „camera” va fi adaugat in dictionarul repo; arunca exceptie daca elementul exista deja}

nr←cam.nr_cam

daca ¬repo.cauta(nr,cam) atunci

repo.adauga(nr,cam)

altfel

@arunca exceptie: Camera ocupata

sfarsit daca

sfârșit store

subalgoritm **delete(repo, cam)** este

{elementul „camera” va fi sters din dictionarul repo; arunca exceptie daca elementul nu exista}

nr←cam.nr_cam

daca repo.cauta(nr,cam) atunci

repo.sterge(nr,cam)

altfel

@arunca exceptie: Camera nu exista

sfarsit daca

sfârșit delete

subalgoritm **update(repo, cam)** este

{elementul „camera” va fi actualizat in dictionarul repo; arunca exceptie daca elementul nu exista}

nr←cam.nr_cam

daca repo.cauta(nr,cam) atunci

repo.sterge(nr,cam)

repo.adauga(nr,cam)

altfel

@arunca exceptie: Camera nu exista

sfarsit daca

sfârșit update

functie **find(repo, cam)** este

{se cauta elementul „camera” in dictionarul repo; arunca exceptie daca elementul nu exista}

nr←cam.nr_cam

daca repo.cauta(nr,cam) atunci

find←cam

altfel

@arunca exceptie: Camera nu exista

sfarsit daca

sfârșit find

functie **getAll(repo)** este

{returneaza tot dictionarul}

getAll←repo

sfârșit getAll

subalgoritm **loadData(repo)** este

{incarca in dictionar continutul fisierului de intrare dat}

@deschide fisier

@daca fisierul nu a putut fi deschis, arunca exceptie

@cat timp mai sunt elemente in fisier

@citeste nr,nume,pret

camera←Camera(nr,nume,pret)

getAll←repo.adauga(nr,camera)

@inchide fisier

sfârșit loadData

subalgoritm **saveData(repo)** este

{salveaza continutul dictionarului intr-un fisier de iesire dat}
@deschide fisier
@daca fisierul nu a putut fi deschis, arunca exceptie
repo.iterator(repo,i);
i.prim(i)
cat timp i.valid executa
e←i.element(i,e)
@scrie in fisier elementul
i.urmator(i)
@inchide fisier
sfârșit saveData

Service

Clasa Service
- **Repository** *repo* - **Validator** *val*
- inchiriere(*Camera cam*) - faPlata(*int nrCam*) - adaugaPlata(*int nrCam, int pret*). - eliberare(*int nr*) - showPlataNeefectuata() - showPlataLaZi() - showCamere()

subalgoritm **inchiriere(serv,cam)** este
{elementul „camera” va fi adaugat in service; arunca exceptie daca elementul nu e valid}
@Val.validate(cam) valideaza datele de intrare
serv.repo.store(cam)
sfârșit inchiriere

subalgoritm **faPlata(serv,nrCam)** este
{pretul elementul corespunzator numarului camerei va fi actualizat la 0; arunca exceptie daca nrCam nu e valid; nrCam:intreg}
@fake←Camera(nrCam,”fara”,1)
@Val.validate(fake) valideaza datele de intrare
cam← serv.repo.find(fake)
cam.pret_chirie=0

```
        Repo.update(cam)
sfârșit faPlata

subalgoritm adaugaPlata(serv,nrCam,pret) este
        {pretul elementul corespunzator numarului camerei va fi actualizat; arunca exceptie daca
nrCam nu e valid; nrCam,pret:intreg, }
        @fake←Camera(nrCam,"fara",pret)
        @val.validate(fake) valideaza datele de intrare
        cam← serv.repo.find(fake)
        cam.pret_chirie=pret
        serv.repo.update(cam)
sfârșit adaugaPlata

subalgoritm eliberare(serv,nr) este
        {camera va fi stearsa din dictionarul repo; arunca exceptie daca nr nu e valid;nr: intreg}
        @fake←Camera(nr,"fara",1)
        @Val.validate(fake) valideaza datele de intrare
        cam← serv.repo.find(fake)
        serv.repo.delete(cam)
sfârșit eliberare

subalgoritm showPlataNeefectuata(serv,dictionar,iterator) este
        { dictionar ∈ D, dictionar ordonat; dictionar contine toate elementele ce au pretul
>0;iterator∈ I, iterator pe dictionar}
        Iterator(serv.repo.getAll(),iterator)
        cat timp valid(iterator) executa
                element(iterator, pereche)
                daca pereche.v.pret>0 atunci
                        adauga(dictionar,R,pereche.c, pereche.v)
                sfarsit daca
                urmator(iterator)
        sfarsit cat timp
sfârșit showPlataNeefectuata

subalgoritm showPlataLaZi(serv,dictionar,iterator) este
        { dictionar ∈ D, dictionar ordonat; dictionar contine toate elementele ce au pretul
=0;iterator∈ I, iterator pe dictionar}
        Iterator(serv.repo.getAll(),iterator)
```

```
        cat timp valid(iterator) executa
                element(iterator, pereche)
                daca pereche.v.pret=0 atunci
                        adauga(dictionar,R,pereche.c, pereche.v)
                sfarsit daca
                urmator(iterator)
        sfarsit cat timp
sfârșit showPlataLaZi
```

functie **showCamere(serv)** este

```
        { returneaza toate camerele din dictionar}
        showCamere←serv.repo.getAll()
sfârșit showPlataNeefectuata
```

Console

Clasa Console
- **Service** serv
- apel_inchiriere() - apel_efectuarePlata() - apel_adaugarePlata() - apel_elibereare() - apel_showNeplatit() - apel_showPlatit() - apel_showCamere()

subalgoritm **apel_inchiriere(cons)** este

```
        { cons- element de tip Console, c- element de tip Camera
         c a fost adaugata in dictionar}
        @citire date de intrare pentru Camera c
        cons.serv.inchiriere(c)
sfârșit apel_inchiriere
```

subalgoritm **apel_efectuarePlata(cons)** este

```
        { cons- element de tip Console, nr-intreg
        pretul camerei cu cheia nr a fost actualizat la 0}
        @citire nr
```

```
        cons.serv.faPlata(nr)
sfârșit apel_efectuarePlata

subalgoritm apel_adaugarePlata(cons) este
        { cons- element de tip Console, nr,pret-intreg
        pretul camerei cu cheia nr a fost actualizat la „pret”}
        @citire nr,pret
        cons.serv.adaugaPlata(nr,pret)
sfârșit apel_adaugarePlata

subalgoritm apel_eliberare(cons) este
        { cons- element de tip Console, nr-intreg
        sterge camera cu cheia nr}
        @citire nr
        cons.serv.eliberare(nr)
sfârșit apel_eliberare

subalgoritm apel_showNeplatit(cons) este
        { cons- element de tip Console, dictionar- dictionar ordonat,i –iterator pe dictionar ordonat
        se afiseaza elementele dictionarului}
        cons.showPlataNeefectuata(dictionar)
        Iterator(dictionar,iterator)
        cat timp valid(iterator) executa
                element(iterator, pereche)
                @tiparire pereche
                urmator(iterator)
        sfarsit cat timp
sfârșit apel_showNeplatit

subalgoritm apel_showPlatit(cons) este
        { cons- element de tip Console, dictionar- dictionar ordonat,i –iterator pe dictionar ordonat
        se afiseaza elementele dictionarului}
        cons.showPlataLaZi(dictionar)
        Iterator(dictionar,iterator)
        cat timp valid(iterator) executa
                element(iterator, pereche)
                @tiparire pereche
                urmator(iterator)
```

```
        sfarsit cat timp

sfârșit apel_showPlatit

subalgoritm apel_showCamere(cons) este
        {cons- element de tip Console, dictionar- dictionar ordonat,i –iterator pe dictionar ordonat
        se afiseaza elementele dictionarului}
        cons.showCamere(dictionar)
        Iterator(dictionar,iterator)
        cat timp valid(iterator) executa
                element(iterator, pereche)
                @tiparire pereche
                urmator(iterator)
        sfarsit cat timp
sfârșit apel_showCamere
```

Complexitatea operațiilor din aplicație

Repository

- store(repo,cam): O(n)
- delete(repo,cam): O(n)
- update(repo,cam): O(n)
- find(repo,cam): O(n)
- getAll(repo): $\theta(1)$
- loadData(repo): $\theta(n)$
- saveData(repo): $\theta(n)$

Service

- inchiriere(cam): O(n)
- faPlata(nrCam): O(n)
- adaugaPlata(nrCam, pret): O(n)
- eliberare(nr): O(n)
- showPlataNeefectuata(): $\theta(n)$
- showPlataLaZi(): $\theta(n)$
- showCamere(): $\theta(1)$

Console

- apel_inchiriere(): O(n)
- apel_efectuarePlata(): O(n)
- apel_adaugarePlata(): O(n)
- apel_elibereare(): O(n)
- apel_showNeplatit(): $\theta(n)$
- apel_showPlatit(): $\theta(n)$
- apel_showCamere(): $\theta(n)$