Mlendea Horațiu
Grupa 214

25. TAD ListaOrdonata – implementare folosind un arbore binar de cautare.

1. Enunț

În cadrul universității s-a organizat un concurs individual de programare. Ficare student a primit un punctaj cuprins intre 0 și 100 de puncte. În funcție de punctajul obținut, dar și de numarul total al participanților, acestora li se vor acorda diferite premii și mențiuni.

Se cere crearea unei aplicații care permite:

- Adăugarea unui participant
- Afișarea listei ordonate a participanților
- Afișarea listei ordonate a premianților/mențiunilor
- Afișarea punctajului unui anumit participant
- Ștergerea și modificarea datelor unui participant

Observații:

- Fiecare participant are un numar matricol, un nume, un punctaj și numarul grupei din care face parte
- Premiile I, II și trei se acorda primilor trei participanți cu cele mai mari punctaje mai mari decât 60
- Mențiunile se acorda primilor 25% dintre participanții cu punctaje mai mari de 40

2. Decizii de implementare

Se creează un arbore binar de căutare care poate să conțină obiecte generice comparabile. În cazul nostru, fiecare nod va fi defapt un obiect de tip Participant, care va conține informațiile aferente acestuia. Deoarece majoritatea operațiilor vor avea ca și criteriu de comparare punctajul participanților, elementele (de tip comparabil) vor fi comparate în funcție de acest punctaj, iar arborele va fi construit pur și simplu prin compararea acestor elemente.
Arborele va fi abstractizat într-o lista ordonată, prin intermediul căreia se va face legătura dintre apelurile din consolă/GUI și arbore.

3. TAD OrderedList

Domeniu:

$L = \{l \mid l = [e_1, e_2, ..., e_n], e_i \in \text{TElement} \ \forall i = 1, 2, ..., n\}$

TElement

- ID: String
- Name: String
- Age: Integer
- Score: Foat

TNode

- Parent: ↑TNode
- ChildLeft: ↑TNode
- ChildRight: ↑TNode
- Element: ↑TElement

Operații:

- Create(l)
  - Pre:
    Post: $l \in L, l = \Phi$ listă vidă

```
Subalgoritm Create(l)
// Complexitatea de timp este θ(1)
    root <- NIL
    count <- 0
```

- Insert(l, e)
  - Pre: l ∈ L, e ∈ TElement
  - Post: l' = l ⊕ e

```
Subalgoritm Insert(l, e) este
// Complexitatea de timp este O(h), unde h este înălțimea arborelui
    node <- CreateNod(e)

    Dacă root = NIL atunci
        root <- node
    Altfel
        InsertNode(root, node)
    Sf_dacă

    elems <- elems + 1
Sf_Subalgoritm

Subalgoritm InsertNode(l, parent, node) este
// Complexitatea de timp este O(h), unde h este înălțimea arborelui
    Dacă node.Element < parent.Element atunci
        Dacă parent.ChildLeft = NIL atunci
            parent.ChildLeft <- node
            node.Parent <- parent
        Altfel
            InsertNode(parent.ChildLeft, node)
        Sf_Dacă
    Altfel Dacă node.Element > parent.Element atunci
        Dacă parent.ChildRight = NIL atunci
            parent.ChildRight <- node
            node.Parent <- parent
        Altfel
            InsertNode(l, parent.ChildRight, node)
        Sf_Dacă
    Sf_Dacă
Sf_Subalgoritm
```

- Remove(l, e)
  - Pre: l ∈ L, e ∈ TElement
  - Post: l' = l - e

```
Subalgoritm Remove(l, e) este
// Complexitatea de timp este O(h), unde h este înălțimea arborelui

    node <- SearchElement(l, e, l.root)
```

```
// Când nu are fii
Dacă node.ChildLeft = NIL && node.ChildRight = NIL atunci
    Dacă node.Parent != NIL atunci
        Dacă node.Parent.ChildLeft = node atunci
            [node.Parent].ChildLeft <- null;
        Altfel
            [node.Parent].ChildRight <- null;
        Sf_Dacă
    Sf_Dacă
    node <- NIL
    count <- count - 1

// Are un fiu (Stâng)
Altfel Dacă node.ChildRight = NIL atunci
    Dacă node.Parent != NIL atunci
        Dacă [node.Parent].ChildLeft = ↑node atunci
            [node.ChildLeft].Parent <- ↑node.Parent
            [node.Parent].ChildLeft <- ↑node.ChildLeft
        Altfel
            [node.ChildLeft].Parent <- ↑node.Parent
            [node.Parent].ChildLeft <- ↑node.ChildRight
        Sf_Dacă
    Sf_Dacă
    node <- NIL
    count <- count - 1

// Are un fiu (Drept)
Altfel Dacă node.ChildLeft = NIL atunci
    Dacă node.Parent != NIL atunci
        Dacă [node.Parent].ChildRight = ↑node atunci
            [node.ChildRight].Parent <- ↑node.Parent
            [node.Parent].ChildRight <- ↑node.ChildRight
        Altfel
            [node.Child.Right].Parent <- ↑node.Parent
            [node.Parent].ChildLeft <- ↑node.ChildRight
        Sf_Dacă
    Sf_Dacă
    node <- null
    count <- count - 1

// Are ambii fii
Altfel
    x <- node

    CâtTimp x.ChildLeft != NIL execută
        x <- x.ChildLeft
    Sf_CâtTimp
```

```
            node.Element <- x.Element
            nodeChild <- x.ChildLeft = NIL ? x.ChildRight : x.ChildLeft

            Dacă x.ChildLeft != NIL atunci
                Dacă [x.Parent].ChildLeft = ↑x atunci
                    [x.Parent].ChildLeft <- ↑nodeChild
                Altfel
                    [x.Parent].ChildRight <- ↑nodeChild
                Sf_Dacă
            Altfel
                Dacă [x.Parent].ChildLeft = ↑x atunci
                    [x.Parent].ChildLeft <- ↑nodeChild
                Altfel
                    [x.Parent].ChildRight <- ↑nodeChild
                Sf_Dacă
            Sf_Dacă
            count <- count - 1
        Sf_Dacă
    Sf_Subalgoritm
```

- Contains(l, e)
  - Pre: l ∈ L, e ∈ TElement
  - Post: true dacă e ∈ l, false în caz contrar
  - Complexitate de timp O(h)

```
    Funcția Contains(l, e) este
    // Complexitatea de timp este O(h), unde h este înălțimea arborelui
        Dacă SearchElement(l, e, l.root) != NIL atunci
            Contains <- TRUE
        Sf_Dacă
        Contains <- FALSE
    Sf_Funcția

    Funcția SearchElement(l, e, parent) este
    // Complexitatea de timp este O(h), unde h este înălțimea arborelui
        node <- parent

        Dacă node = NIL atunci
            SearchElement <- null
        Sf_Dacă

        Dacă el = node.Element atunci
            SearchElement <- node
        Altfel Dacă el < node.Element atunci
            SearchElement <- SearchElement(l, e, node.ChildLeft)
        Altfel
            SearchElement <- SearchElement(l, e, node.ChildRight)
        Sf_Dacă
    Sf_Funcția
```

Mlendea Horațiu
Grupa 214

- Empty(l)
  - Pre: l ∈ L
  - Post: true dacă l = Φ, false în caz contrar

  Funcția Empty(l) este
  // Complexitatea de timp este θ(1)
  Dacă l.root = NIL atunci
  Empty <- TRUE
  Sf_Dacă
  Empty <- FALSE
  Sf_Funcția

- Clear(l)
  - Pre: l ∈ L
  - Post: l = Φ

  Subalgoritm Clear(l) este
  // Complexitatea de timp este θ(1)
  l.root <- NIL
  count <- 0
  Sf_Subalgoritm

- Count(l)
  - Pre: l ∈ L
  - Post: n ∈ Integer, n este numărul de elemente din l

  Funcția Count(l) este
  // Complexitatea de timp este θ(1)
  Count <- count
  Sf_Funcția

- Iterator(l, i)
  - Pre: l ∈ L
  - Post: i ∈ I, i este un iterator pe lista l

  Funcția Iterator(l) este
  // Complexitatea de timp este θ(1)
  Iterator <- Iterator(l.root)
  Sf_Funcția

- Destroy(l)
  - Pre: l ∈ L
  - Post: l este distrusă, iar spațiul de memorie este dealocat

  Funcția Destroy(l) este
  // Complexitatea de timp este θ(1)
  l.root <- NIL
  // restul nodurilor sunt dealocate automat

Sf_Funcția

4. TAD Iterator

Domeniu:
I = {i | i este un iterator pe o listă ordonată de elemente de tip TElement}

Operații:

- Create(i, l)
  - Pre: l ∈ OrderedList
  - Post: i ∈ Iterator, s-a creat iteratorul i pe lista l, current indică către primul el

```
Subalgoritm Create(i, l) este
// Complexitatea de timp este O(h), unde h este înălțimea arborelui
    current <- FindMinimum(l.root)
Sf_Subalgoritm
```

- Element(i, e)
  - Pre: i ∈ I
  - Post: e ∈ TElement, e este elementul curent din iterație

```
Funcția Element(i)
// Complexitatea de timp este θ(1)
    Element <- current.Element;
Sf_Funcția
```

- Valid(i)
  - Pre: i ∈ I
  - Post: true dacă current referă o poziție validă din l, false în caz contrar

```
Funcția Valid(i) este
// Complexitatea de timp este θ(1)
    Dacă current != NIL atunci
        Valid <- TRUE
    Altfel
        Valid <- FALSE
    Sf_Dacă
Sf_Funcția
```

- Next(i)
  - Pre: i ∈ I
  - Post: current' referă la următorul element din listă (relativ la fostul curent)

```
Subalgoritm Next(i) este:
// Complexitatea de timp este O(h), unde h este înălțimea arborelui
    Dacă current.ChildRight != NIL atunci
        current <- FindMinimum(l, current.ChildRight)
        Return
    Sf_Dacă
```

```
        y <- current.Parent;
        x <- current;

        CâtTimp y != NIL și x = y.ChildLeftt execută
            x <- y
            y <- y.Parent
        Sf_CâtTimp

        current <- y
    Sf_Subalgoritm

    Funcția FindMinimum(root) este
    // Complexitatea de timp este O(h), unde h este înălțimea arborelui
        Dacă root = NIL atunci
            FindMinimum <- NIL
        Sf_Dacă

        Dacă root.ChildLeft != NIL atunci
            FindMinimum <- FindMinimum(root.ChildLeft)
        Sf_Dacă

        FindMinimum <- root
    Sf_Funcția
```

- Previous(i)
  - Pre: i ∈ I
  - Post: current' referă la precedentul element din listă (relativ la fostul curent)

```
    Subalgoritm Previous(i) este:
    // Complexitatea de timp este O(h), unde h este înălțimea arborelui
        Dacă current.ChildLeft != NIL atunci
            current <- FindMaximum(l, current.ChildLeft)
            Return
        Sf_Dacă

        y <- current.Parent;
        x <- current;

        CâtTimp y != NIL și x = y.ChildRight execută
            x <- y
            y <- y.Parent
        Sf_CâtTimp

        current <- y
    Sf_Subalgoritm

    Funcția FindMaximum(root) este
    // Complexitatea de timp este O(h), unde h este înălțimea arborelui
        Dacă root = NIL atunci
```

```
            FindMaximum <- NIL
        Sf_Dacă

        Dacă root.ChildRight != NIL atunci
            FindMaximum <- FindMaximum(root.ChildRight)
        Sf_Dacă

        FindMaximum <- root
    Sf_Funcția
```