

3-ÁRIS FÁK KEZELÉSE

Feladat:

Van-e olyan pozitív érték egy fában¹, hogy az INORDER bejárás szerint azt megelőző k darab érték mind negatív (a feladat egy 3-áris fára vonatkozik, melyben számokat tárolunk)?

Egy általános, 3-áris fa² képét láthatjuk az 1. ábrán.

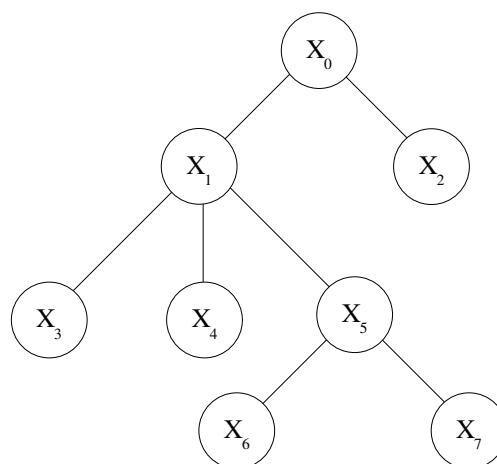
Reprezentáció:

Egy fát a gyakorlatban kétféleképpen reprezentálhatunk:

- Aritmetikai módon (két dimenziós tömb segítségével);
- Mutatók segítségével.

Az aritmetikai tömbös ábrázolása egy olyan két dimenziós tömbben történik, mely sorainak száma megegyezik az ábrázolandó fa elemeinek számával, és négy oszlopa van.

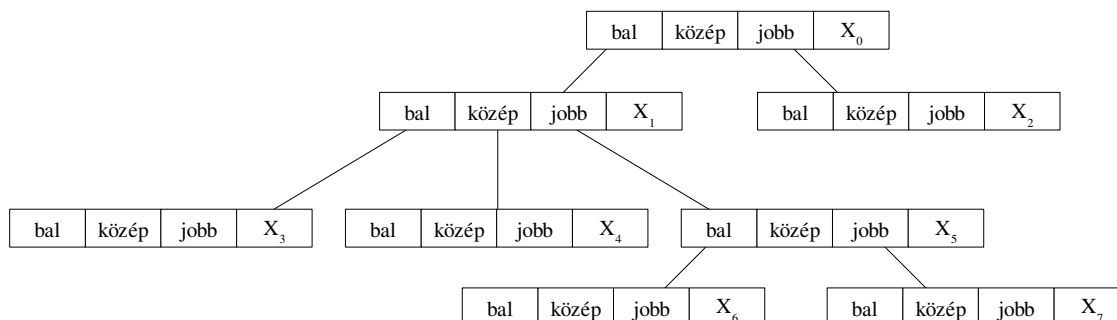
Az első három oszlopban a táblázat sorainak indexeit tároljuk a negyedik oszlopban pedig a fa megfelelő elemének értékét. Egy sor első három eleme rendre azoknak a soroknak az indexeit tárolja, melyek utolsó elemének értéke megegyezik a fa vizsgált eleme bal, középső, illetve jobb elemének értékével.



ábra 1: Egy általános 3-áris fa

Ez az ábrázolási mód nagy mennyiségű adat ábrázolása esetén lehet hasznos.

Általában a második módszert célszerű alkalmazni, így a megoldás során is ezt fogjuk használni, alkalmas adatstruktúrák definiálásának útján. Ennek vizuális szemléltetése a 2. ábrán látható.



ábra 2: Egy általános 3-áris fa reprezentációja

Az ábrát az 1. ábrán bemutatott 3-áris fa alapján lett elkészítve. A „bal”, „közép”, illetve „jobb” mezők mutatók, melyek a fa vizsgált elemének megfelelő gyerekeire mutatnak, az „ X_n ” mező pedig a vizsgált elem értékét tárolja, ahol $n \in [0..7]$, az „ X_0 ” értékmezővel rendelkező elem pedig a fa gyökere.

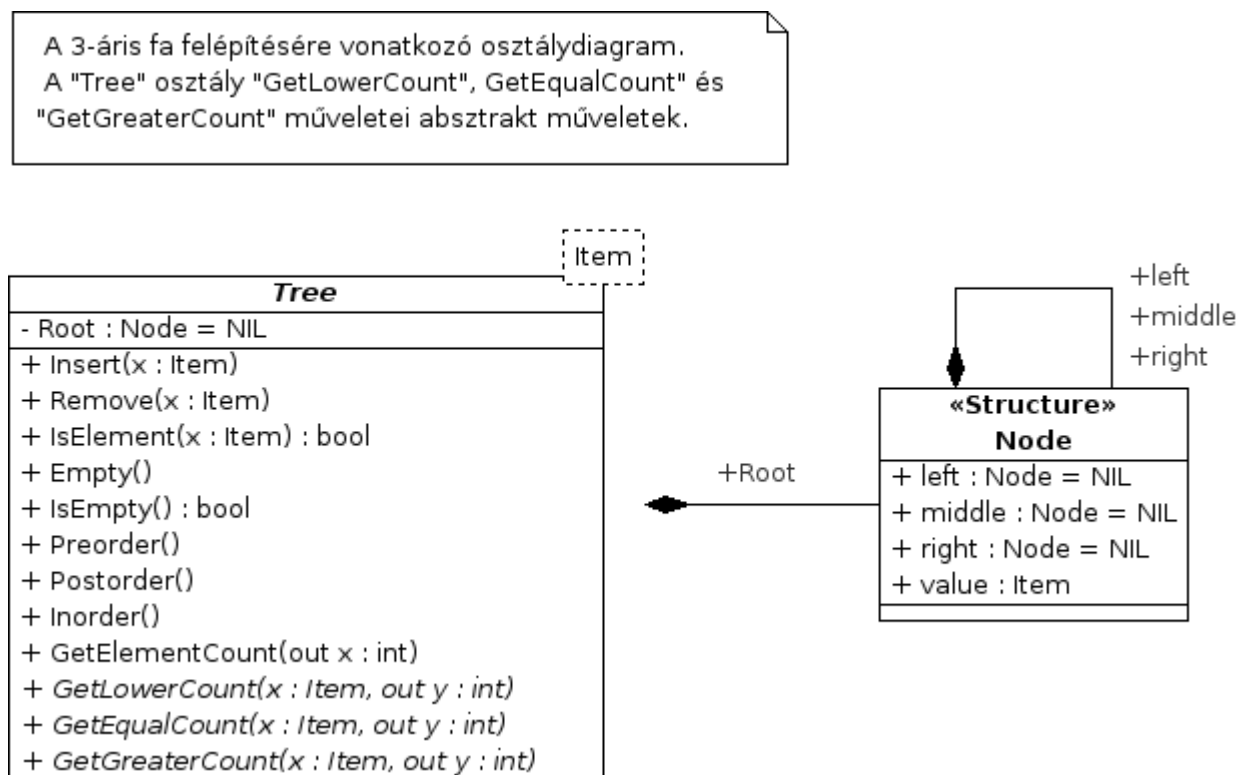
1 Olyan adatszerkezet, melynek egy vagy több eleméhez további – azonos típusú – elemek lehetnek hozzárendelve, egymással párhuzamosan

2 Olyan fa, melynek minden elemére igaz, hogy legfeljebb három gyereke lehet

Megoldás:

A legprecízebb megoldás érdekében először elkészítjük a programot felépítő egységek UML diagramját. Először elkészítjük a program egészére vonatkozó osztálydiagramot, majd részletes leírást adunk az egyes osztályok és a hozzájuk rendelt műveletek szerepéről.

A 3-áris fa adatszerkezet felépítését bemutató osztálydiagramot a 3. ábrán láthatjuk.



ábra 3: A 3-áris fa adatszerkezetének osztálydiagramja

A 3-áris fa adatszerkezetet a „Tree” osztály definiálásával készítettük el. Ennek a „Node” struktúra az al-osztálya, melynek sztereotípiáját „Structure” típusra állítottuk be. A C++ programozási nyelv lehetőséget ad „Struct” típus definiálására is, így ott ezt közvetlenül struktúraként készíthetjük majd el.

A „Tree” osztályban definiált „Root” egy „Node” típusú mutató, melynek alapértelmezett értéke „NIL”. A fa felépítésekor ez a mutató fog a fa gyökerére mutatni.

A fát úgy készítjük el, hogy több adattípusra is működőképes legyen, ennek megfelelően definiáltunk egy „Item” elnevezésű sablont. Az osztály összes művelete ezzel az adattípussal fog dolgozni a továbbiakban.

A sablon miatt volt szükség arra, hogy a „GetLowerCount”, „GetEqualCount” és a „GetGreaterCount” műveleteket absztrakt műveletekként definiáljuk. Ezeknek a műveleteknek a részletesebb tárgyalására később kerül sor.

Az ábrán jól látható, hogy az „IsElement(x : Item)” és az „IsEmpty()” függvények kivételével az osztály összes többi művelete eljárás, konkrétan „void” típusú. Ezzel azért van szükség, mert a fa bejárását az áttekinthetőség érdekében rekurzív függvényekkel kíséreljük megoldani.

Műveletek:

A műveletek leírását az 1. táblázatban láthatjuk.

Azonosító	Leírás	Típus
Insert(x : Item)	Beszúrja az „x” elemet a fa megfelelő helyére.	Eljárás
Remove(x : Item)	Eltávolítja a fából az „x” elemet.	Eljárás
IsElement(x : Item)	Lekérdezi, hogy az „x” elem megtalálható-e a fában.	Lekérdező konstans
Empty()	Kiüríti a fát.	Eljárás
IsEmpty()	Lekérdezi, hogy üres-e a fa.	Lekérdező konstans
Preorder()	Bejárja a fa elemeit preorder ³ módon.	Eljárás
Postorder()	Bejárja a fa elemeit postorder ⁴ módon.	Eljárás
Inorder()	Bejárja a fa elemeit inorder ⁵ módon.	Eljárás
GetElementCount(out x : int)	Lekérdezi a fa elemeinek számát.	Eljárás
GetLowerCount(x : Item, out y : int)	Lekérdezi az „x”-nél kisebb elemek számát.	Absztrakt eljárás
GetEqualCount(x : Item, out y : int)	Lekérdezi az „x”-el egyenlő elemek számát.	Absztrakt eljárás
GetGreaterCount(x : Item, out y : int)	Lekérdezi az „x”-nél nagyobb elemek számát.	Absztrakt eljárás

táblázat 1: Műveletek leírása

A fa szerkezete:

Definiáljuk a 3-áris fát úgy, hogy minden elemére igaz, hogy a bal gyerekének értéke kisebb, jobb gyerekének értéke nagyobb, mint a szülő értéke, középső gyerekének értéke pedig megegyezik a szülő értékével. Az 1. és a 2. ábrán láthatóknak megfelelően tehát $bal(X).érték < X.érték < jobb(X).érték \wedge közép(X).érték = X.érték$, ahol $bal(X)$, $közép(X)$, illetve $jobb(X)$ rendre a fa egy „X” elemének bal, középső, illetve jobb gyerekeire mutató pointerok.

A 3. ábrán és az 1. táblázatban feltüntetett „Insert(x : Item)” eljárás ennek megfelelően szúr be egy új elemet a fába. Üres fa esetén a beszúrt elem lesz a fa gyökere.

A „Remove(x : Item)” eljárás eltávolítja az „x” elemet a fából, amennyiben az adott elem a fa egyik levele, vagyis nincsenek gyerekei.

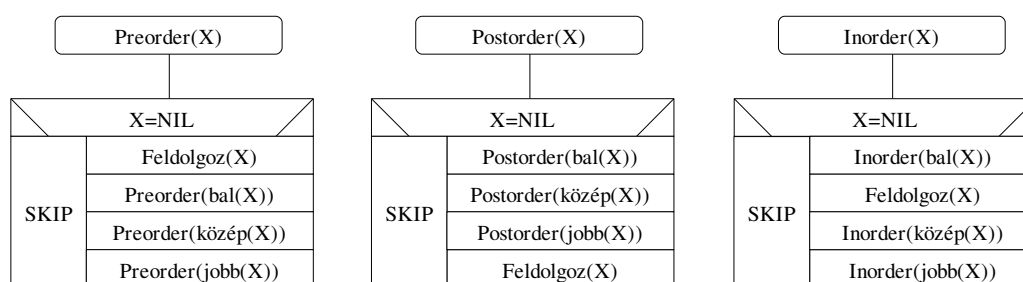
A fa bejárása:

A fa bejárása háromféleképpen történhet: *preorder*, *postorder* és *inorder* módon. Mindhárom bejárás rekurzív függvények segítségével történik, melyek szinte teljesen megegyeznek, az egyetlen különbség az elemek feldolgozási sorrendje. A fa bejárásainak algoritmusait a 4. ábrán láthatjuk.

3 Preorder bejárás esetén először a gyökeret, aztán a bal gyereket, végül pedig a jobb gyereket vizsgáljuk.

4 Postorder bejárás esetén először a bal gyereket, aztán a jobb gyereket, végül pedig a gyökeret vizsgáljuk.

5 Inorder bejárás esetén először bal gyereket, aztán a gyökeret, végül pedig a jobb gyereket vizsgáljuk.



ábra 4: Egy 3-áris fa preorder, postorder és inorder bejárásainak algoritmusai

Megvalósítás:

A konkrét megoldóprogramban a fent megadott algoritmusokhoz hasonlóan rekurzív függvényeket fogunk használni, melyek egy mutatót kapnak paraméterül. Ezen kívül szükség lesz még egy extra paraméterre is, mely egész típusú. Erre azért van szükség, hogy rekurzív függvényen belül is számlálni tudjuk a – feladatban meghatározott – negatív értékeket, továbbá egy logikai változóra is, mely alapértelmezésben hamis értéket kap, és igazra áll, amennyiben az egész típusú változó értéke eléri a „k” mennyiséget, és az aktuális mutató értéke nem „NIL”.

Mivel rekurzív eljárásokat használunk, a legcélszerűbb a feladat megoldásához az „Inorder()” eljárást felhasználni.

A fa felépítése:

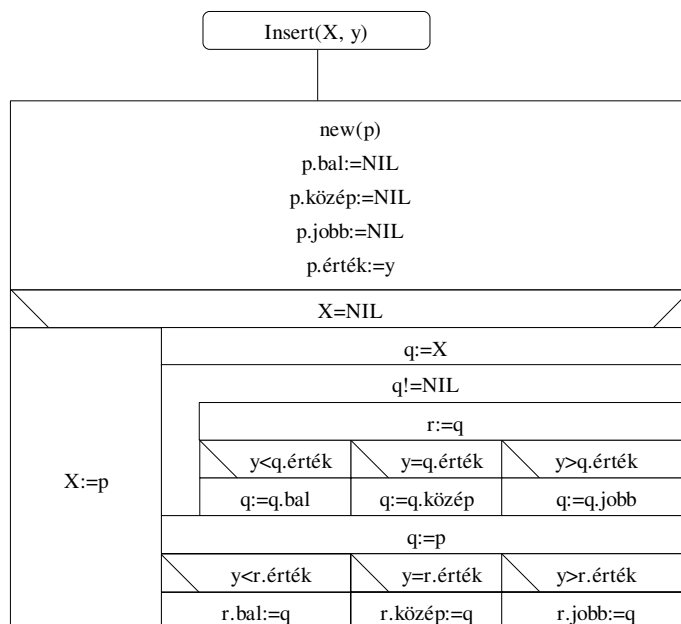
A fa felépítésére az „Insert(x : Item)” eljárás szolgál, mely a 3-áris fa definíciójának megfelelően beszúr egy elemet. Ennek algoritmusai az 5. ábrán látható.

Az eljárás megkeresi a beszúrandó elem helyét a fában, majd a leendő szülőjének megfelelő mutatóját átállítja, hogy a beszúrt elemre mutasson. Amennyiben a fa üres volt, úgy a beszúrt elem lesz a fa gyökere.

Az eljárás a fa gyökerére mutató mutatót kapja paraméterül, minden újonnan beszúrt elem esetében innen kezd a vizsgálatot.

Elem eltávolítása:

Eltávolítani csak levelet lehet, vagyis olyan elemet, melynek nincs gyereke. Az erre szolgáló eljárás megkeresi az eltávolítani kívánt elemet, ellenőrzi, hogy levél-e, majd ezt követően törli, amennyiben lehetséges.



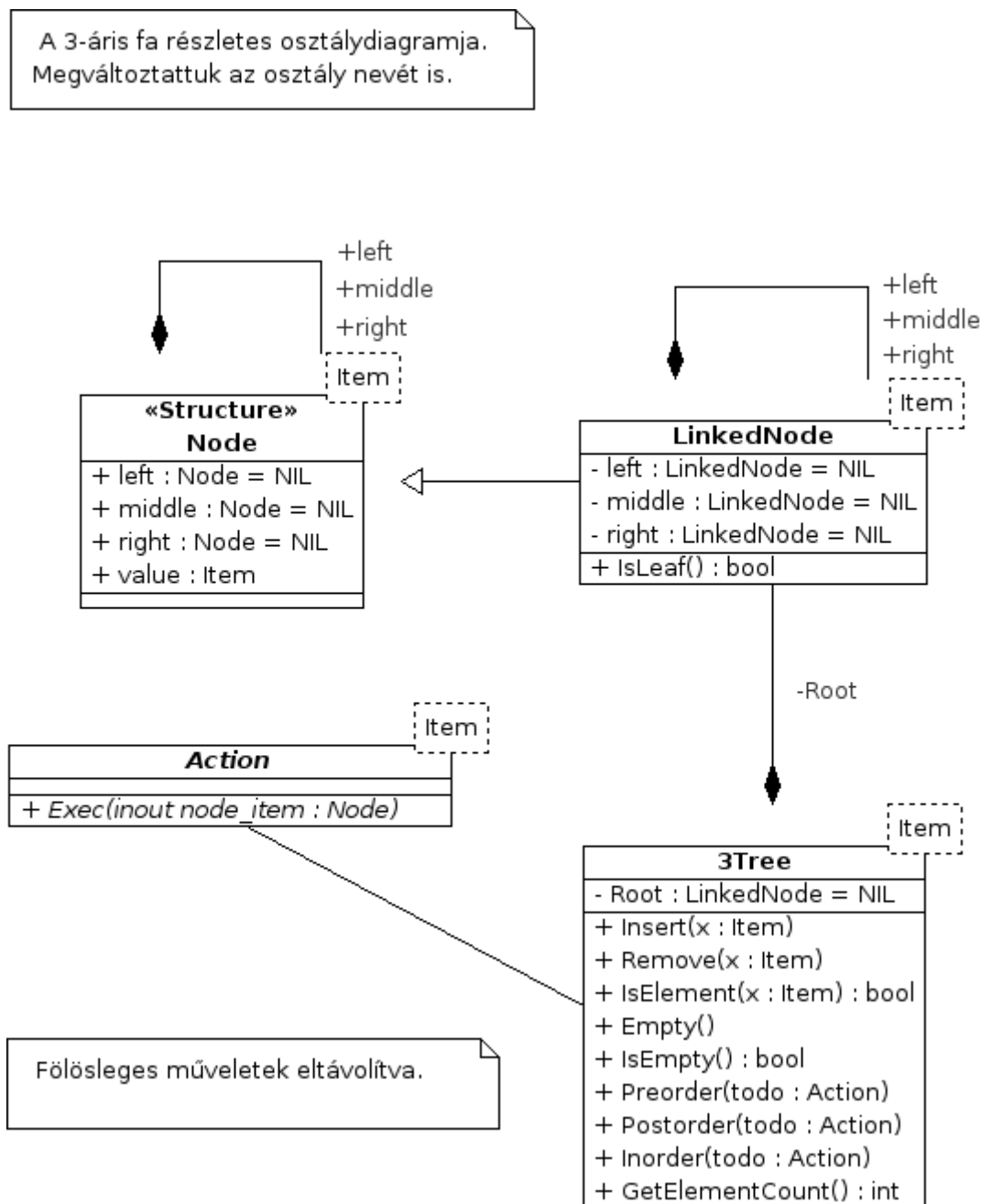
ábra 5: A beszúrás algoritmusai

A fa lebontása:

A fa lebontását a fa bejárásán keresztül tesszük meg a destruktorkban. Azért, hogy a program minél megbízhatóbban működjön, a fa lebontásánál a bejárás *postorder* módon fog történni. A kód újra felhasználhatóságának érdekében megkíséreljük a törléskor történő bejárást az alapértelmezett bejárókkal megvalósítani.

Megoldás újra felhasználhatósággal:

Azért, hogy a program részei újra felhasználhatóak legyenek, a 3-áris fa felépítésére vonatkozó osztálydiagramot módosítjuk. A fa adatszerkezethez való hozzáférésen – egyelőre – nem változtatunk, de az adatszerkezet osztályait szétbontjuk további al-osztályokra, és a köztük való kapcsolatot öröklődéssel oldjuk meg. Az új adatszerkezet diagramját a 6. ábrán láthatjuk.



ábra 6: A 3-áris fa adatszerkezetének részletes osztálydiagramja

Az ábrán látható, hogy a feladat szempontjából felesleges műveleteket eltávolítottuk az osztályból, továbbá az osztály nevét is megváltoztattuk.

Bevezettünk két új osztályt is: a „TreeNode” és az „Action” osztályt. Az előbbi összeköti a „Node” struktúrát a „3Tree” osztállyal, az utóbbi pedig meghatározza, hogy a fa egy bejárása esetén milyen műveletet hajtsunk végre a fa elemein. Mindkét osztály a kód újra felhasználhatóságát teszi lehetővé.

A feladat megoldását az „Inorder(todo : Action) bejáró teszi lehetővé, ahol a művelet a negatív elemek összeszámlálása lesz, és a feladat kérdésére választ adó logikai változó megfelelő beállítása.

Forráskód – Header-állomány (tree.h):

```
#ifndef TREE_H
#define TREE_H
#include <iostream>

//Alapelem-osztaly
template <class Item>
class Node{
public:
    Node *left;
    Node *middle;
    Node *right;
    Item value;
    Node(const Item& val){
        value=val;
    }
    //Ertek lekerdezese
    Item GetValue();
};

//Tevekenyseg osztaly (absztrakt)
template <class Item>
class Action{
public:
    virtual void Exec(Node<Item> *node_item)=0;
};

//A fa osztalya (deklaracio)
template <class Item>
class Tree;

//A fa elemeinek osztalya
template <class Item>
class TreeNode : public Node<Item>{
    friend class Tree<Item>;
public:
    TreeNode(    const Item& v,
                LinkedNode *l,
                LinkedNode *m,
                LinkedNode *r):
        Node<Item>(v), left(l), middle(m), right(r){}
    bool IsLeaf() const;
private:
    LinkedNode *left;
    LinkedNode *middle;
    LinkedNode *right;
};

//A 3-aris fa osztalya
template <class Item>
class Tree{
public:
    //konstruktor
    Tree():Root(NULL){};
    //segedfuggveny a destruktorhoz
    class DelAction : public Action<Item>{
    public:
        void Exec(Node<Item> *node){
            delete node;
        }
    };
};
```

```

};
//Destruktor
~Tree(){
    DelAction del;
    Post(Root, &del);
}
//Beszuras
void Insert(const Item& x);
//Elem eltavolitasa
void Remove(const Item& x);
//Tartalmazas ellenorzes
bool IsElement(const Item& x);
//Fa kiuritese
void Empty();
//Allapot ellenorzes
bool IsEmpty() const;
//Bejarok
void PreOrder(Action<Item>*todo){ Pre(Root,todo); }
void PostOrder(Action<Item>*todo){ Post(Root,todo); }
void InOrder(Action<Item>*todo){ In(Root,todo); }
//Elem megkeresese
LinkedList<Item> *Find(const Item& x);
//Elmszam lekerdezese
int GetElementCount();
//Kivetelkezes
enum Exception{WRELEM, AEMPTY};
private:
    //A fa gyokere
    LinkedList<Item>* Root;
    //Bejarok megvalositasai
    void Pre(LinkedList<Item> *r, Action<Item> *todo);
    void Post(LinkedList<Item> *r, Action<Item> *todo);
    void In(LinkedList<Item> *r, Action<Item> *todo);
    //Segedosztaly a szamlalashoz
    class Element : public Action<int>{
    public:
        int total;
        Element() : total(0){}
        void Exec(Node<int> *node_item){
            ++total;
        }
        //Alaphelyzet
        void reset(){ total=0; }
    };
};
#endif

```

Forráskód – Megvalósítás (tree.cpp):

```

#include <iostream>
#include "tree.h"

//Ertek lekerdezese
template <class Item>
Item Node<Item>::GetValue(){
    return value;
}

//Elem level mivoltanak lekerdezese
template <class Item>
bool LinkedList<Item>::IsLeaf() const{
    return ((left==NULL)&&(middle==NULL)&&(right==NULL));
}

//Beszuras
template <class Item>
void Tree<Item>::Insert(const Item& x){
    LinkedList<Item> *p;
    p = new LinkedList<Item>(x, NULL, NULL, NULL);
}

```

```

//Ha a fa ures
if(Root==NULL){
    Root=p;
}else{
    //Hely megkeresese
    ListNode<Item> *q=Root;
    ListNode<Item> *r;
    int NILMode;
    while(q!=NULL){
        r=q;
        if(x<q->value){
            q=q->left;
            NILMode=0;
        }else{
            if(x==q->value){
                q=q->middle;
                NILMode=1;
            }else{
                q=q->right;
                NILMode=2;
            }
        }
    }
    //Elem beszurasa
    if(NILMode==0){
        r->left=p;
    }else{
        if(NILMode==1){
            r->middle=p;
        }else{
            r->right=p;
        }
    }
}

//Elem megkeresese
template <class Item>
ListNode<Item> *Tree<Item>::Find(const Item& x){
    ListNode<Item> *q=NULL;
    ListNode<Item> *p=Root;
    while((p!=NULL)&&(q==NULL)){
        if(x<p->value){
            p=p->left;
        }else{
            if(x>p->value){
                p=p->right;
            }else{
                q=p;
            }
        }
    }
    return q;
}

//Tartalmazas eldontese
template <class Item>
bool Tree<Item>::IsElement(const Item& x){
    ListNode<Item> *p=Find(x);
    return (p!=NULL);
}

//Allapot ellenorzese
template <class Item>
bool Tree<Item>::IsEmpty() const{
    return (Root==NULL);
}

//Elemszam lekerdezese
template <class Item>
int Tree<Item>::GetElementCount(){
    Element store;
    PreOrder(&store);
    return store.total;
}

```



```

//Fa kiurítése
template <class Item>
void Tree<Item>::Empty(){
    if(Root!=NULL){
        DelAction del;
        Post(Root, &del);
        Root=NULL;
    }else{
        throw AEMPTY;
    }
}

//Elem eltávolítása
template <class Item>
void Tree<Item>::Remove(const Item& x){
    LinkedNode<Item> *p=Root;
    LinkedNode<Item> *q=NULL;
    bool found=false;
    bool removed=false;
    while((p!=NULL)&&(found==false)){
        //Keresés
        if(x<p->value){
            q=p;
            p=p->left;
        }else{
            if(x>p->value){
                q=p;
                p=p->right;
            }else{
                if(p->middle!=NULL){
                    q=p;
                    p=p->middle;
                }else{
                    //Megvan
                    found=true;
                    if(p->IsLeaf()==true){
                        //Kilancolás
                        if(q!=NULL){
                            if(x<q->value){
                                q->left=NULL;
                            }else{
                                if(x>q->value){
                                    q->right=NULL;
                                }else{
                                    q->middle=NULL;
                                }
                            }
                        }
                    }
                    //Torles
                    if(removed==false){
                        if(p==Root){
                            Root=NULL;
                        }
                        delete p;
                        removed=true;
                    }
                }
            }
        }
    }
    if(removed==false){
        throw WRELEM;
    }
}

//Bejaro megvalositasai
//Preorder
template <class Item>
void Tree<Item>::Pre(LinkedNode<Item> *r, Action<Item> *todo){
    if(r!=NULL){
        todo->Exec(r);
        Pre(r->left, todo);
    }
}

```

```

        Pre(r->middle, todo);
        Pre(r->right, todo);
    }
}
//Postorder
template <class Item>
void Tree<Item>::Post(LinkedNode<Item> *r, Action<Item> *todo){
    if(r!=NULL){
        Post(r->left, todo);
        Post(r->middle, todo);
        Post(r->right, todo);
        todo->Exec(r);
    }
}
//Inorder
template <class Item>
void Tree<Item>::In(LinkedNode<Item> *r, Action<Item> *todo){
    if(r!=NULL){
        In(r->left, todo);
        todo->Exec(r);
        In(r->middle, todo);
        In(r->right, todo);
    }
}

//Volt-e k darab negativ ertekek egy pozitiv elem elott?
class Negatives : public Action<int>{
private:
    int d,k;
public:
    bool result;
    int number;
    Negatives() : d(0), k(1), result(false){}
    void SetNumber(const int x){ k=x; }
    void Exec(Node<int> *node_item){
        if(node_item!=NULL){
            if(result==false){
                if(node_item->GetValue()<0){
                    ++d;
                }else{
                    if(d>=k){
                        number=node_item->GetValue();
                        result=true;
                    }
                    d=0;
                }
            }
        }
    }
};

//Kiiratas
template <class Item>
class Printer : public Action<Item>{
private:
    std::ostream& prt;
public:
    Printer(std::ostream &out) : prt(out){};
    void Exec(Node<Item> *node){
        prt << '(' << node->GetValue() << ' ';
    }
};

```

Forráskód – Tesztprogram (ttestt.cpp):

```
/*
 * NEGYEDIK SZAMU BEADANDO
 * Keszitette: Abraham Robert
 * EHA-kod: ABROAAI.ELTE
 * Datum: 2008/12/11.
 */

#include <iostream>
#include "tree.h"
#include "tree.cpp"

using namespace std;

//Fa tartalmanak kiiratas (Standard Output; egyszerusites)
template <class Item>
void Fa_Kiir(Tree<Item> &tree_out);

int main()
{
    //INTERFACE
    int n;
    cout << "MUVELETEK 3-ARIS FAKKAL" << endl;
    cout << "(Integer ertekekkel dolgozom)" << endl;
    do{
        cout << endl;
        cout << "A fa elemszama: "; cin >> n;
        if(n<0){
            cout << endl;
            cout << "Nem adhat meg negativ erteket!" << endl;
        }
    }while(n<0);
    //FA ELKESZITESE
    Tree<int> t;
    //FA FELTOLTESE
    int m;
    for(int i=0; i<n; ++i){
        cout << i+1 << ". elem: "; cin >> m;
        t.Insert(m);
    }
    //TARTALOM KIIRASA
    cout << endl;
    Fa_Kiir(t);
    //Van-e k db. negativ ertek utan egy pozitiv?
    int k;
    cout << endl;
    cout << "Hany negativ ertek utan talaljak egy pozitivat?" << endl;
    cout << "(Inorder bejarassal dolgozom)" << endl;
    do{
        cout << endl;
        cout << "Valasz: "; cin >> k;
        if(k<1){
            cout << endl;
            cout << "Minimum ertek: 1." << endl;
        }
    }while(k<1);
    Negatives TreeNeg;
    TreeNeg.SetNumber(k);
    t.InOrder(&TreeNeg);
    cout << endl;
    if(TreeNeg.result==true){
        cout << "Talaltam: " << TreeNeg.number << "." << endl;
    }else{
        cout << "Nem talaltam." << endl;
    }
    //Elem eltavolitasa:
    int l;
    cout << endl;
```

```

    cout << "Level torlese (ertek szerint): "; cin >> l;
    cout << endl;
    try{
        t.Remove(l);
        Fa_Kiir(t);
    }
    catch(Tree<int>::Exception hiba){
        cout << "A torles nem sikerult!" << endl;
    }
    cout << endl;
    cout << "A fa elemszama: " << t.GetElementCount() << endl;
    int ser;
    cout << endl;
    cout << "Keressuk: "; cin >> ser;
    cout << endl;
    if(t.IsElement(ser)){
        cout << "Megtalaltam." << endl;
    }else{
        cout << "Nem talalom." << endl;
    }
    cout << endl;
    cout << "Fa kiuritese..." << endl;
    cout << endl;
    try{
        t.Empty();
        cout << "Kiurites utan:" << endl;
        cout << endl;
        Fa_Kiir(t);
    }
    catch(Tree<int>::Exception hiba){
        cout << "A fa mar ures." << endl;
    }
    return 0;
}

//Fa tartalmának kiiratasa (Standard Output; egyszerusites)
template <class Item>
void Fa_Kiir(Tree<Item> &tree_out){
    Printer<int> print(cout);
    if(tree_out.IsEmpty()==true){
        cout << "A fa ures." << endl;
    }else{
        cout << "A fa tartalma:" << endl;
        cout << endl;
        cout << "Preorder bejarassal: ";
        tree_out.PreOrder(&print);
        cout << endl;
        cout << "Postorder bejarassal: ";
        tree_out.PostOrder(&print);
        cout << endl;
        cout << "Inorder bejarassal: ";
        tree_out.InOrder(&print);
        cout << endl;
    }
}

```

Tesztelés:

- Elemszám megadása (negatív, nulla és annál nagyobb pozitív egészekkel);
- Fa feltöltése;
- Kiírt tartalom áttekintése;
- A feladat által kérdezett, pozitív elem megkeresése (amennyiben van ilyen), eredmények áttekintése;
- Levél eltávolításának kipróbálása (szélsőséges és átlagos értékekkel);

- Eredmények áttekintése;
- Keresés kipróbálása;
- Utolsó műveletek eredményeinek áttekintése.