Feladat

Adjuk meg, hogy egy ternáris fa INORDER *bejárás szerint sorozatba fűzött értékei között mekkora a leghosszabb csupa pozitív számot tartalmazó részsorozat.*

Ternáris fa

Típusspecifikáció

Az A alaphalmaz feletti T_3 ternáris fák halmazát rekurzívan úgy definiálhatjuk, hogy egy ternáris fa vagy üres, vagy egy A-beli értékből, és három (bal, középső, és jobboldali) ternáris fából áll.

Az elvárható műveletek egyrészt a tárolt értékek elérése és a fa bejárása, másrészt a fa felépítése és módosítása.

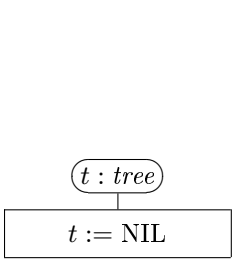
Reprezentáció

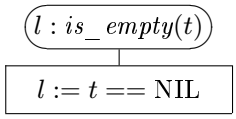
A ternáris fa egy csúcsát egy $t = (k, p, l, m, r)$ ötös reprezentálja, ahol $k \in A$ a csúcs által tárolt kulcs, $p \in T_3$ a szülő-csúcs, és $l, m, r \in T_3$ a gyermekek. A ternáris fát a gyökér-csúcs reprezentálja.

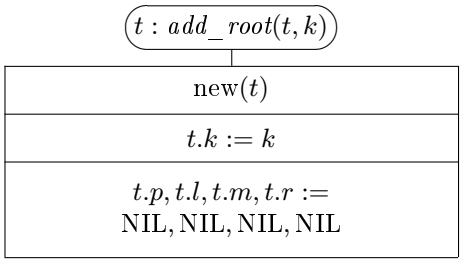
Absztrakt implementáció

Létrehozás

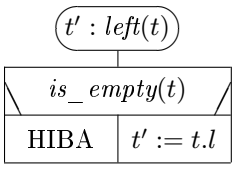
Külön *add_root* műveletre azért van szükség, hogy megengedjük és kezelni tudjuk az üres fákat.

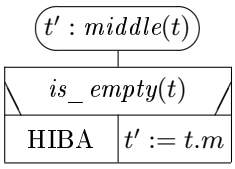


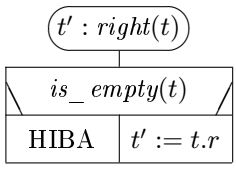


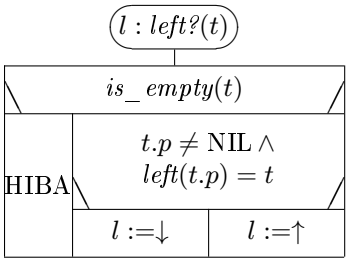


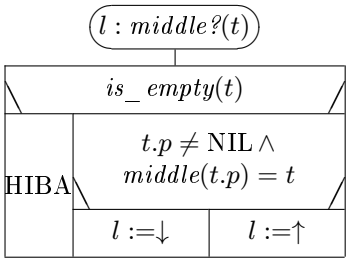
Navigáció

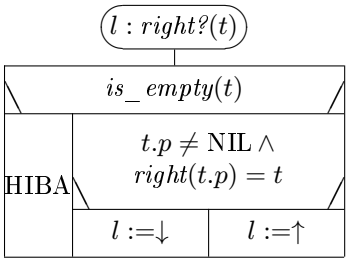




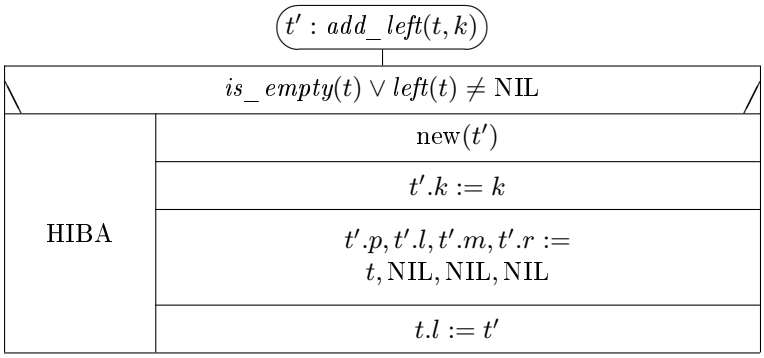








Feltöltés



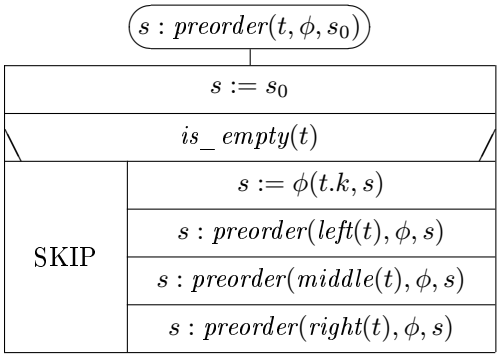
add_middle, *add_right* értelemszerűen.

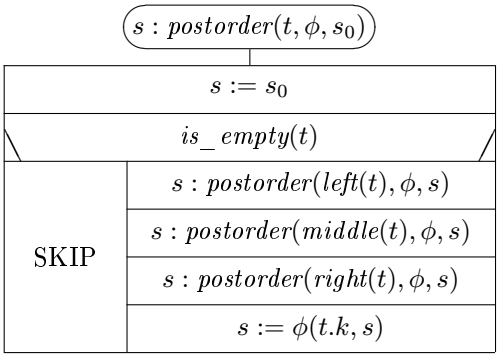
Bejárás

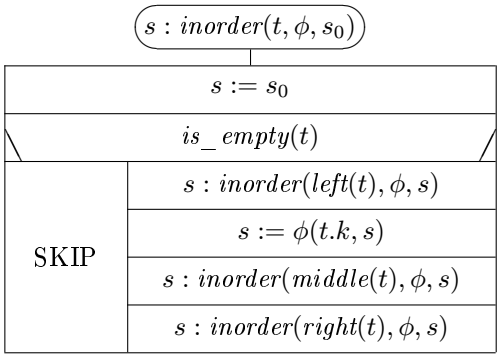
A fa bejárásait a *callback* pattern modellezésével végezzük. Ebben a modellben a ϕ callback kétváltozós, egyértekű függvényként jelenik meg, ahol a második változó, és az érték, a callback állapotát jelenti:

$$\phi : A \times S_\phi \to S_\phi$$

Így egyszerűen adódnak a bejáró eljárások rekurzív definíciói:

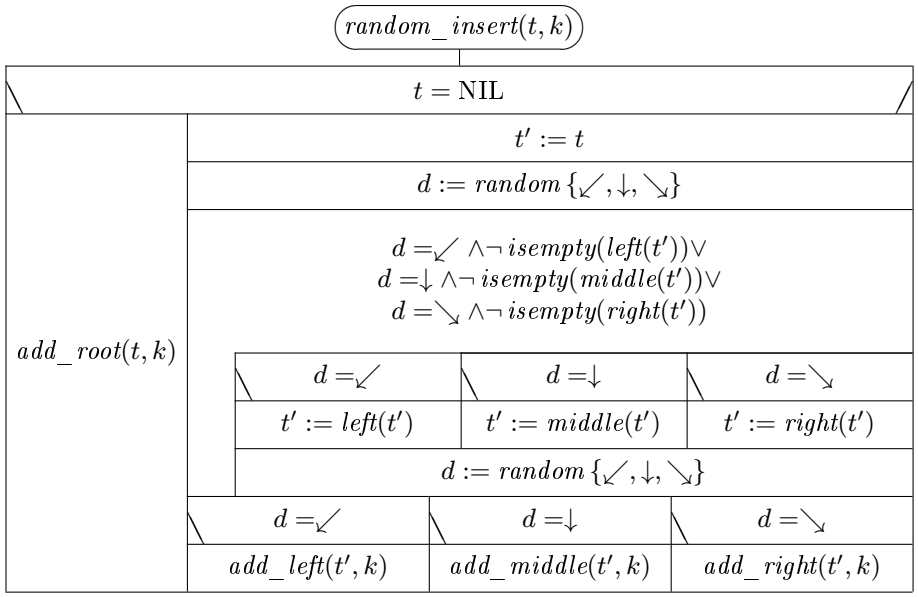


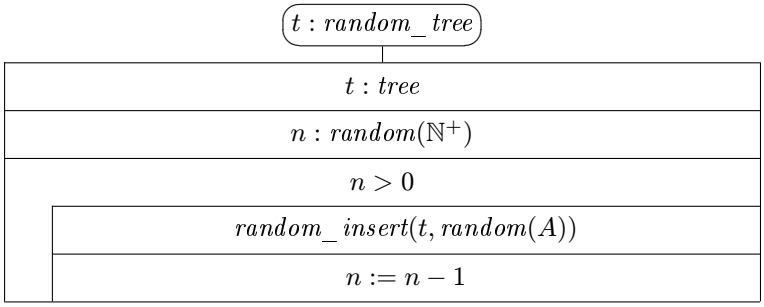




Véletlenszerű ternáris fa felépítése

A feladat megoldását véletlenszerűen generált ternáris fákon fogjuk bemutatni. Ehhez szükségünk lesz az alábbi programra, amely a megadott ternáris fán egy véletlenszerű útvonalon eljut egy levélig, és annak egy véletlenszerű helyére berakja a kapott elemet.





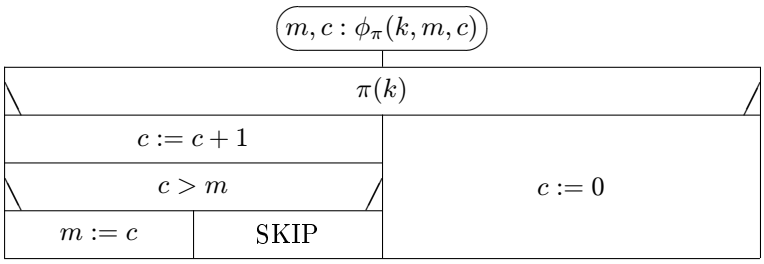
Leghosszabb részsorozat hossza

Az eddigi eredményeink alapján tehát a feladatot visszavezettük a fent definiált *inorder* eljárás alkalmas ϕ függvénnyel való alkalmazására. Ehhez tekintsük az alábbi függvénydefiníciót:

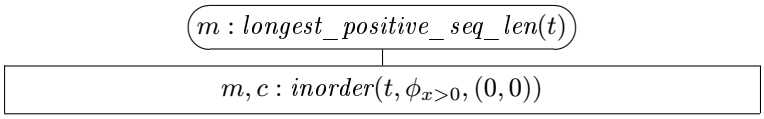
$$\phi_\pi : A \times \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$$

$$\phi_\pi(k, m, c) := \begin{cases} (m, 0) & \text{ha } \neg\pi(k) \\ (m, c+1) & \text{ha } \pi(k) \wedge c+1 \leq m \\ (c+1, c+1) & \text{ha } \pi(k) \wedge c+1 > m \end{cases}$$

Ezt könnyedén implementálhatjuk lényegében a feltételes maximumkeresés tételének „hasával”:



És így a feladat megoldása előáll a következő programmal:



C++ implementáció

Referenciaszálálásos smartpointer

Az egyetlen nehézséget az implementációban az adja, hogy a fenti modell megvalósításához a fákat referencia szerint kell átadni, ami megnehezíti a memóriakezelést. Ezért egy nagyon egyszerű, referenciaszámlálásos smartpointert fogunk használni:

```
template<typename T> class RefCountedPtr
{
    T *ptr;
    mutable unsigned int *refcount;

public:
    RefCountedPtr (T *ptr_ = 0) : ptr (ptr_), refcount (new unsigned int (1)) {}

    RefCountedPtr (const RefCountedPtr<T> &other):
        ptr (other.ptr),
        refcount (other.refcount)
        {
            ++*refcount;
        }

    ~RefCountedPtr () { dispose (); }

    T * const       operator-> ()       { return ptr; }
    const T * const operator-> () const { return ptr; }

    operator bool () const { return ptr; }
    bool operator== (const RefCountedPtr<T> &other) const { return other.ptr == ptr; }

    RefCountedPtr<T> &operator= (const RefCountedPtr<T> &other) {
        if (other.ptr == ptr)
            return *this;

        dispose ();

        ptr = other.ptr;
        refcount = other.refcount;
        *refcount += 1;

        return *this;
    }

private:
    void dispose () {
        if (!--*refcount)
        {
            delete ptr;
            delete refcount;
        }
    }
};
```

Ternáris fa

template<**typename** T> **class** TernaryTree

Alapműveletek

A fenti smartpointerrel már egyszerűen megvalósíthatóak az alapműveletek:

```
    struct TreeNode
    {
    public:
        T key;
        RefCountedPtr<TreeNode> parent;
        RefCountedPtr<TreeNode> left, middle, right;

        TreeNode (RefCountedPtr<TreeNode> parent_, const T &key_):
            key (key_),
            parent (parent_)
            {}

        ~TreeNode () {}
    };

    RefCountedPtr<TreeNode> node;

    TernaryTree (RefCountedPtr<TreeNode> node_): node (node_) {}

public:
    TernaryTree () {}
    TernaryTree (const T &root_key): node (new TreeNode (0, root_key)) {}
    TernaryTree (const TernaryTree<T> &other): node (other.node) {}

    operator bool () const { return node != 0; }

    const T& get_key () const { return node->key; }
    T& set_key (const T &key) { assert (node); return node->key = key; }
```

Navigáció

```
    TernaryTree parent () const { return node->parent; }
    TernaryTree left ()   const { return node->left; }
    TernaryTree middle () const { return node->middle; }
    TernaryTree right ()  const { return node->right; }

    bool is_root () const   { return !node->parent; }
    bool is_left () const   { return node->parent && node->parent->left == node; }
    bool is_middle () const { return node->parent && node->parent->middle == node; }
    bool is_right () const  { return node->parent && node->parent->right == node; }
```

Felépítés

```
void add_root (const T &key = T())
    {
        assert (!node);
        node = new TreeNode (0, key);
    }

TernaryTree add_left (const T &key = T ())
    {
        assert (node && !node->left);
        return node->left = new TreeNode (node, key);
    }

TernaryTree add_middle (const T &key = T ())
    {
        assert (node && !node->middle);
        return node->middle = new TreeNode (node, key);
    }

TernaryTree add_right (const T &key = T ())
    {
        assert (node && !node->right);
        return node->right = new TreeNode (node, key);
    }
```

Bejárás

A bejárást természetesen nem szigorúan a modellt követve valósítjuk meg, hanem a szokásos *callback* patternt követve:

```
class visitor
{
public:
    virtual ~visitor() {};

    virtual void visit_node (TernaryTree<T> &tree) = 0;
};

void visit_inorder (visitor &visitor) {
    if (!*this) return;

    left ().visit_inorder (visitor);
    visitor.visit_node (*this);
    middle ().visit_inorder (visitor);
    right ().visit_inorder (visitor);
}
```

Véletlen fa felépítése

```
void insert_random_node (TernaryTree<T> tree, const T &value)
{
    if(!tree)
    {
        tree.add_root (value);
    }
    else
    {
        int d = random () % 3;

        while ((d == 0 && tree.left ()) || (d == 1 && tree.middle ()) || d == 2 && tree.right
        {
            if (d == 0) tree = tree.left ();
            else if (d == 1) tree = tree.middle ();
            else tree = tree.right ();

            d = random () % 3;
        }

        if (d == 0) tree.add_left (value);
        else if (d == 1) tree.add_middle (value);
        else tree.add_right (value);
    }

}

TernaryTree<int> random_tree ()
{
    srandom (time(0));

    unsigned int nodes_num = random () % 20 + 5;

    TernaryTree<int> tree;
    tree.add_root ((random () >> 5) % 200 - 100);

    for (unsigned int i = 1; i != nodes_num; ++i)
    {
        int value = (random () >> 5) % 200 - 100;
        insert_random_node (tree, value);
    }

    return tree;
}
```

A feladatot megoldó callback

```
class LongestInorderPositiveSequenceLen: public TernaryTree<int>::const_visitor
{
    unsigned int count;
    unsigned int max_count;

public:
    LongestInorderPositiveSequenceLen ():
        count (0),
        max_count (0)
        {
        }

    void visit_node (const TernaryTree<int> &tree)
        {
            if (tree.get_key () > 0)
            {
                if (++count > max_count)
                    max_count = count;
            }
            else
                count = 0;
        }

    unsigned int result () const { return max_count; }
};

template<typename T>
int longest_inorder_positive_sequence_len (const TernaryTree<T> &tree)
{
    LongestInorderPositiveSequenceLen visitor;

    tree.visit_inorder (visitor);
    return visitor.result ();
}
```