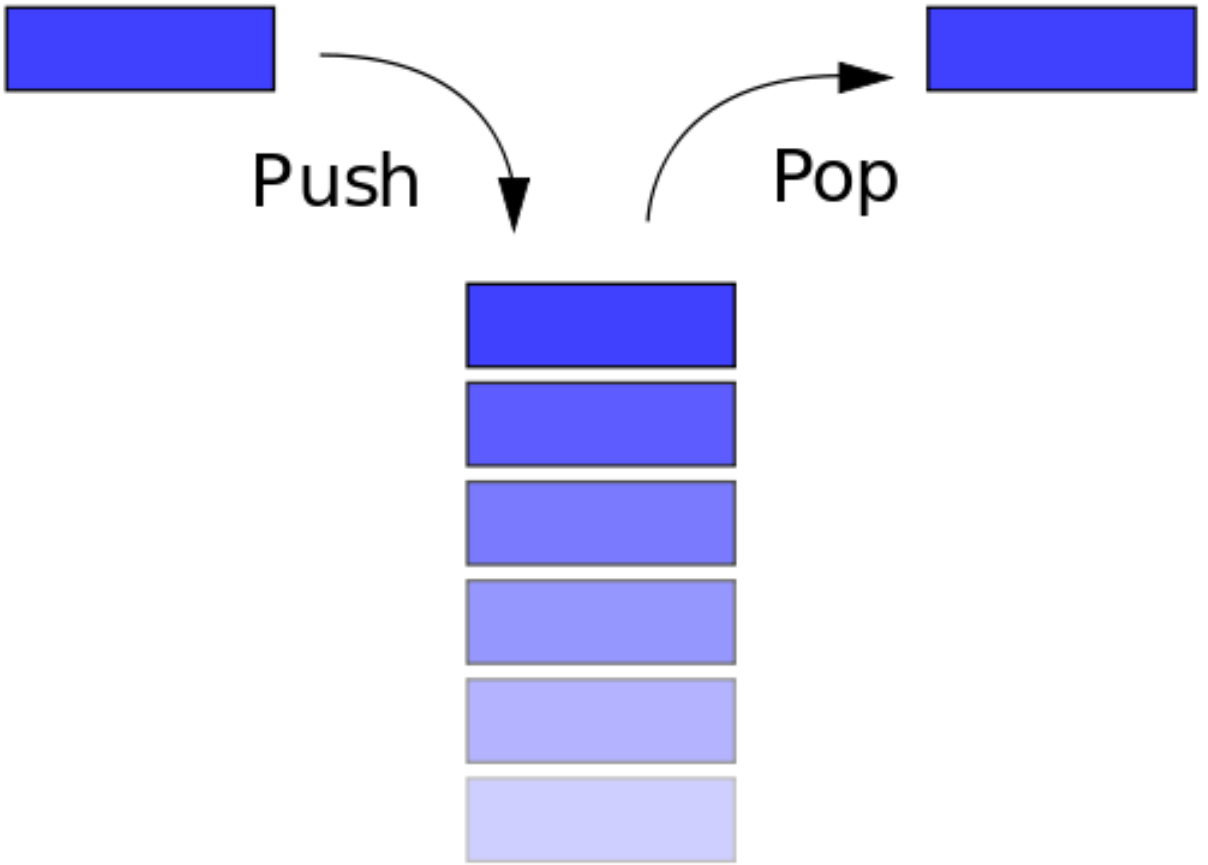
Структури от данни и програмиране

Лекция 2

Стек


Push
Pop

Линейни структури от данни

- Една съставна структура от данни е *линейна*, т.с.т.к. е или празна, или не е празна, но в този случай за нея са в сила:
  - структурата притежава точно един елемент, който е първи и точно един елемент (не непременно друг), който е последен;
  - за всеки елемент, без първия, съществува точно един елемент, който го предхожда;
  - за всеки елемент, без последния, съществува точно един елемент, който го следва.
- Примери: масив, стек, линеен списък и др.
- Примери за нелинейни структури: дърво, граф, множество и др.

Логическо описание

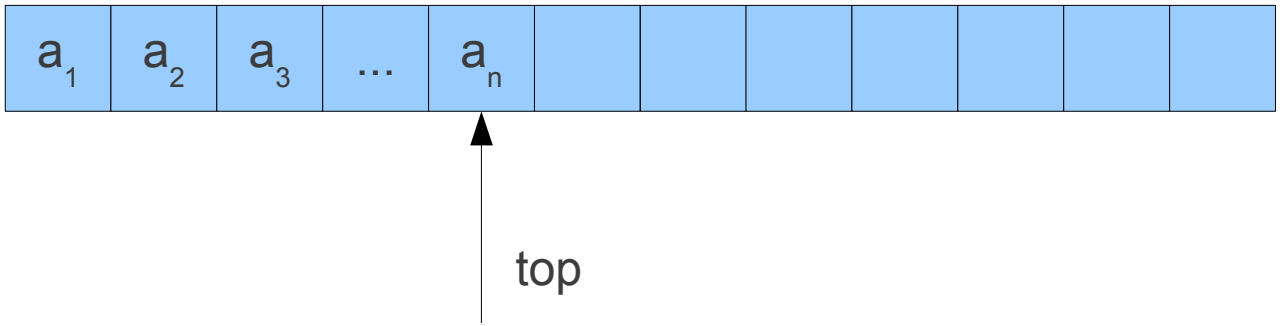
- Хомогенна линейна структура
- Последен влязъл, пръв излязъл (Last In, First Out - LIFO)
- Операции
  - създаване на празен стек (create)
  - проверка за празнота (empty)
  - включване на елемент (push)
  - изключване на елемент (pop)
  - достъп до връх на стека (top)

Логическо описание (2)

- Свойства
  - empty(create()) = true
  - empty(push(x, s)) = false
  - top(create()) = pop(create()) = ERROR
  - top(push(x, s)) = x
  - pop(push(x, s)) = s

Последователно представяне

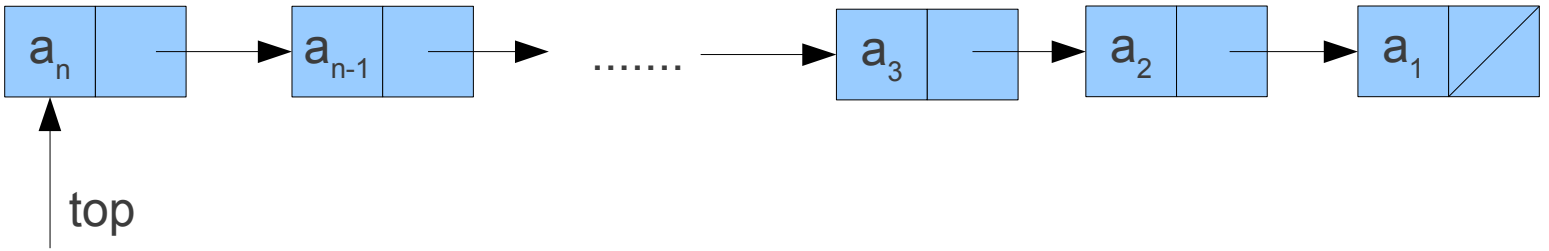
- Използва се масив



- Предимства
- Недостатъци

Свързано представяне

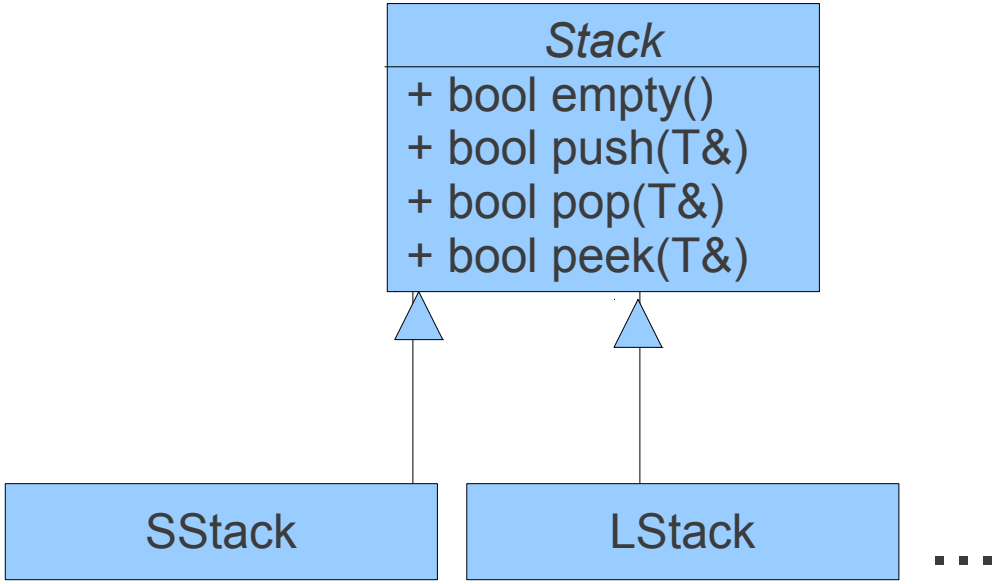
- Всеки елемент (без последният) реферира следващия



- Предимства
- Недостатъци

Реализации на различните представяния

- Ще се възползваме от възможностите на ООП



- (Допълнителен материал: фигурата се нарича UML клас диаграма)

Примерна реализация на базовия клас

- Базов абстрактен клас – дефинира интерфейса

```
template <typename T>
class Stack {
public:
    virtual bool empty() const = 0;
    virtual bool push(T const&) = 0;
    virtual bool pop(T&) = 0;
    virtual bool peek(T&) const = 0;
};
```

- Поставяме абстрактния клас в header файл stack.h

```
#ifndef STACK_H_
#define STACK_H_

#include <iostream>

template <typename T>
class Stack {
    ...
};

#endif /* STACK_H_ */
```

- За удобство добавихме защита от повторно вмъкване с #include

Примерна реализация на последователното представяне

- Файл sstack.cpp

```
#include "stack.h"

const int MAX = 100;
const int INVALID = -1;

template <typename T = int>
class SStack : public Stack<T>
{
private:
    T* s;
    int top;
    int size;
```

Канонично представяне / голяма четворка

- Защо е необходимо?
- Каква е сложността на четирите операции?

```
void copystack(SStack
        const& stack) {
    s = new T[stack.size];
    top = stack.top;
    size = stack.size;
    for(int i = 0;
        i <= stack.top; i++)
    s[i] = stack.s[i];
}
```

```
public:
SStack(int _size = MAX): top(INVALID),
        size(_size) {
    s = new T[size];
}

~Sstack() {
    delete[] s;
}

SStack(SStack const& stack) {
    copystack(stack);
}

SStack& operator=(SStack const& stack) {
    if (this != &stack) { // ако забравим?
        delete[] s;
        copystack(stack);
    }
    return *this;
}
```

Реализация на методите на базовия клас

```
bool empty() const {
    return top == INVALID;
}

bool full() const {
    return top == size - 1;
}

bool push(T const& x) {
    if (full())
        return false;
    s[++top] = x;
    return true;
}
```

- Внимаваме с граничните случаи – празен/пълен стек!

```
bool pop(T& x) {
    if (empty())
        return false;
    x = s[top--];
    return true;
}

bool peek(T& x) const {
    if (empty())
        return false;
    x = s[top];
    return true;
}
```

- Каква е сложността на операциите?

Примерна реализация на свързаното представяне

- Възел от свързаното представяне:

```
template <typename U = int>
struct elem {
    U inf;
    elem<U>* link; // следващ елемент
};
```

- Останалата част – в lstack.cpp

Приложение – пресмятане на израз

- Инфиксен запис

  (1+2)*(3-4/5)

  - Операциите имат приоритети (тегла)
- Постфиксен запис (обратен полски)

  12+345/-*
- Префиксен запис (прав полски)
  *+12-3/45
- При втория и третия запис не се използват скоби

- Нека изразите, които ще бъдат пресмятани, имат следния синтаксис:

<израз> ::= <терм> |

<израз> + <терм> |

<израз> – <терм>

<терм> ::= <множител>|

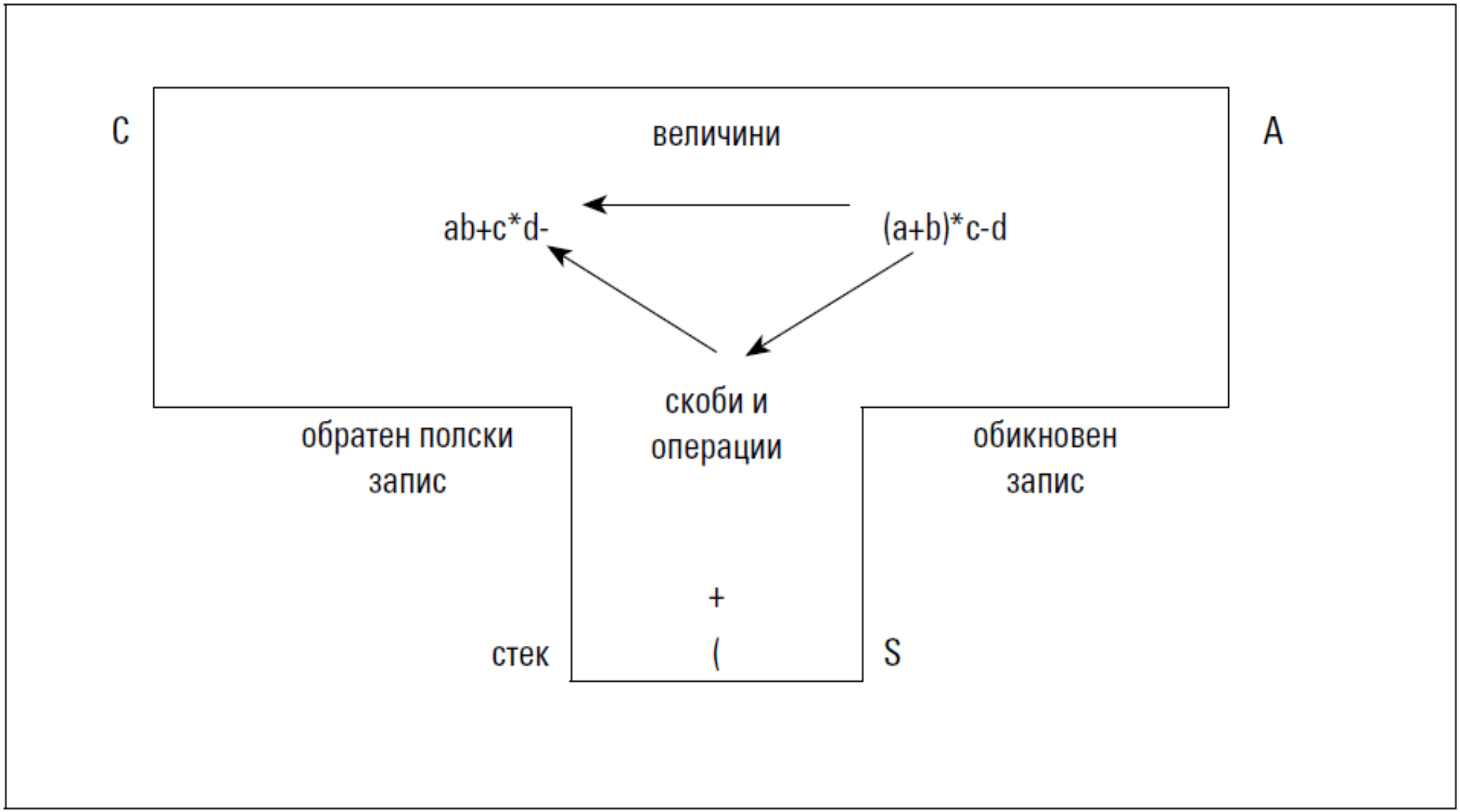
<терм> * <множител> |

<терм> / <множител>

<множител> ::= <цифра>|(<израз>)

<цифра> ::= 0|1| ... |9

Преобразуване в обратен полски запис



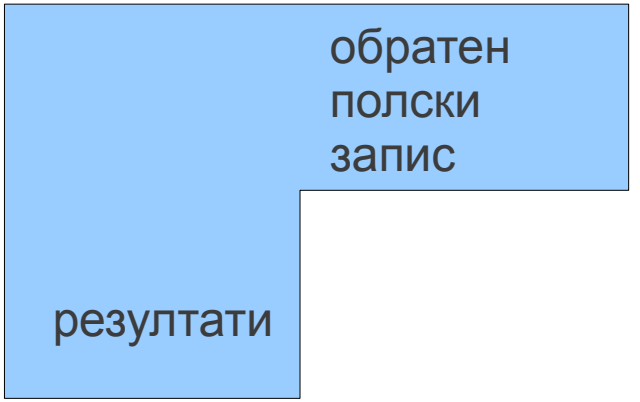
Алгоритъм

- Величините (цифрите) се преместват директно от частта A в частта C;
- Скобата ‘(’ се включва в стека S;
- Аритметичните операции +, -, * и / се включват в стека S. Ако при включване на операция в стека S под нея има операция с по-малко или с равно тегло, по-леката или с равно тегло операция се премества от S в частта C. Това се повтаря, докато се достигне до по-тежка операция, до ‘(’ или до изпразването на стека;
- Скобата ‘)’ изключва от стека S всички операции до достигане до ‘(‘. Операциите се записват в частта C в реда на изключването им, а скобата ‘(‘ се унищожава от ‘)’.
- Ако всички символи от частта A са обработени, елементите на стека S, до изпразването му или до достигане до ‘(‘, се прехвърлят в областта C.

Реализация на преобразуването до обратен полски запис

- (във външен файл)

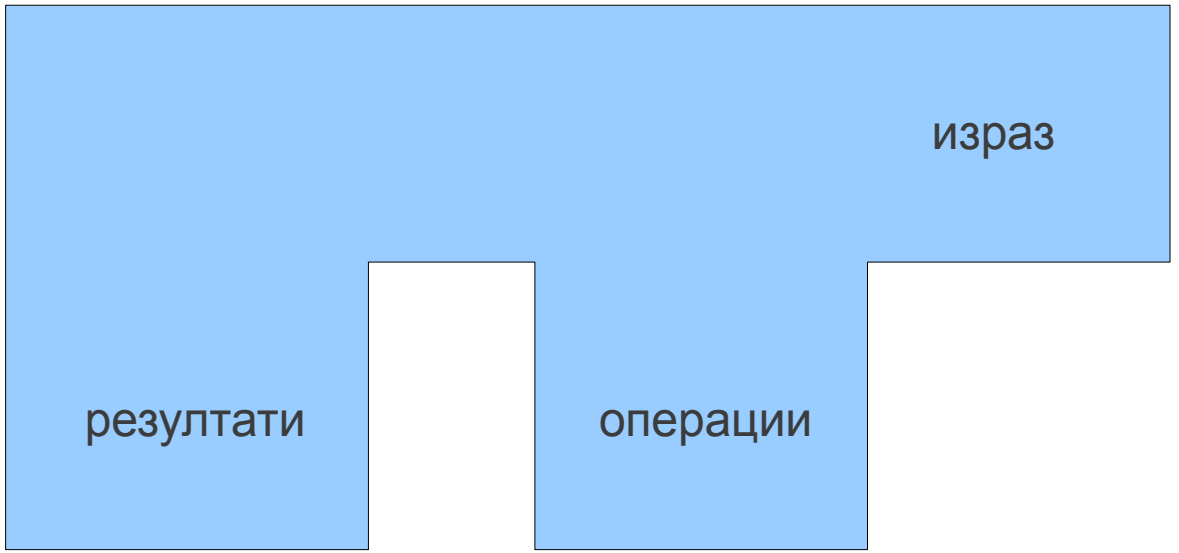
Пресмятане на израз, записан с обратен полски запис



- Констуира се празен стек от числа
- За всеки символ от ОПЗ:
  - Ако е цифра – добавя се в стека
  - Ако е операция:
    - От стека се изваждат толкова на брой стойности, колкото е *арността* на операцията (броят параметри, напр. + е бинарна операция)
    - Върху извлечените стойности се прилага операцията
    - Резултатът се записва в стека
- Накрая в стека има един елемент – стойността на израза

```
char* p = rpn;
LStack<int> s;
while (*p)
{
    if ('0' <= *p && *p <= '9')
        s.push(*p-'0');
    else {
        int a,b;
        s.pop(b);
        s.pop(a);
        switch (*p)
        {
        case '+':s.push(a+b);break;
        case '-':s.push(a-b);break;
        case '*':s.push(a*b);break;
        case '/':s.push(a/b);break;
        default:s.push(0);
        }
    }
    p++;
}
int result;
s.pop(result);
cout << "Резултатът е " << result << endl;
```

Директно пресмятане на израз



Приложение – симулиране на рекурсия

- Стекова рамка
  - при извикване на функция
  - при рекурсия
- Стек вместо стекова рамка
  - Предимство: използваме хийп паметта вместо стековата памет, по-трудно ще получим stack overflow при дълбока рекурсия
  - Недостатъци: трудно четим код

Първи пример - факториел

```
int fact(int n) {
   if (n == 0) return 1;
   return n * fact(n - 1);
}
int fact2(int n) {
    SStack<int> st; st.push(n);
    int x, f = 1;
    while (!st.empty()) {
        st.pop(x);
        if (x == 0) return f;
        f *= x;
        st.push(x - 1);
    }
    return 1;   // за да се избегне предупреждението
                // "not all control paths return a value"
}
```

Втори пример – Фибоначи

```
int fib(int n) {
    if (n == 0 || n == 1) return 1;
    return fib(n - 1) * fib(n - 2);
}
int fib2(int n) {
    ...; st.push(n);
    while (!st.empty()) {
        st.pop(x);
        if (x == 0 || x == 1) return ...;
        ...; st.push(n - 1); st.push(n - 2);
    }
    ...
}
```

Малко допълнителен материал

- Можем да напишем тестов код (Unit test), който проверява дали нашата реализация изпълнява свойствата, които разгледахме в началото
  - Ако направим промяна в кода, стартираме тестовете отново
- Можеше дори да напишем теста преди да сме написали реализацията (Test-driven development)

Още приложения

- Преобразуване на число от 10-ична в *k*-ична бройна система
- Ханойски кули
- и др. - има решения например в статията в Wikipedia

Обобщение

- Какво е стек? Кои са основните **операции**?
- Кои са основните начини за **представяне** на стек? Защо е необходимо да има различни представяния?
- За какво може да се **използва** стек?