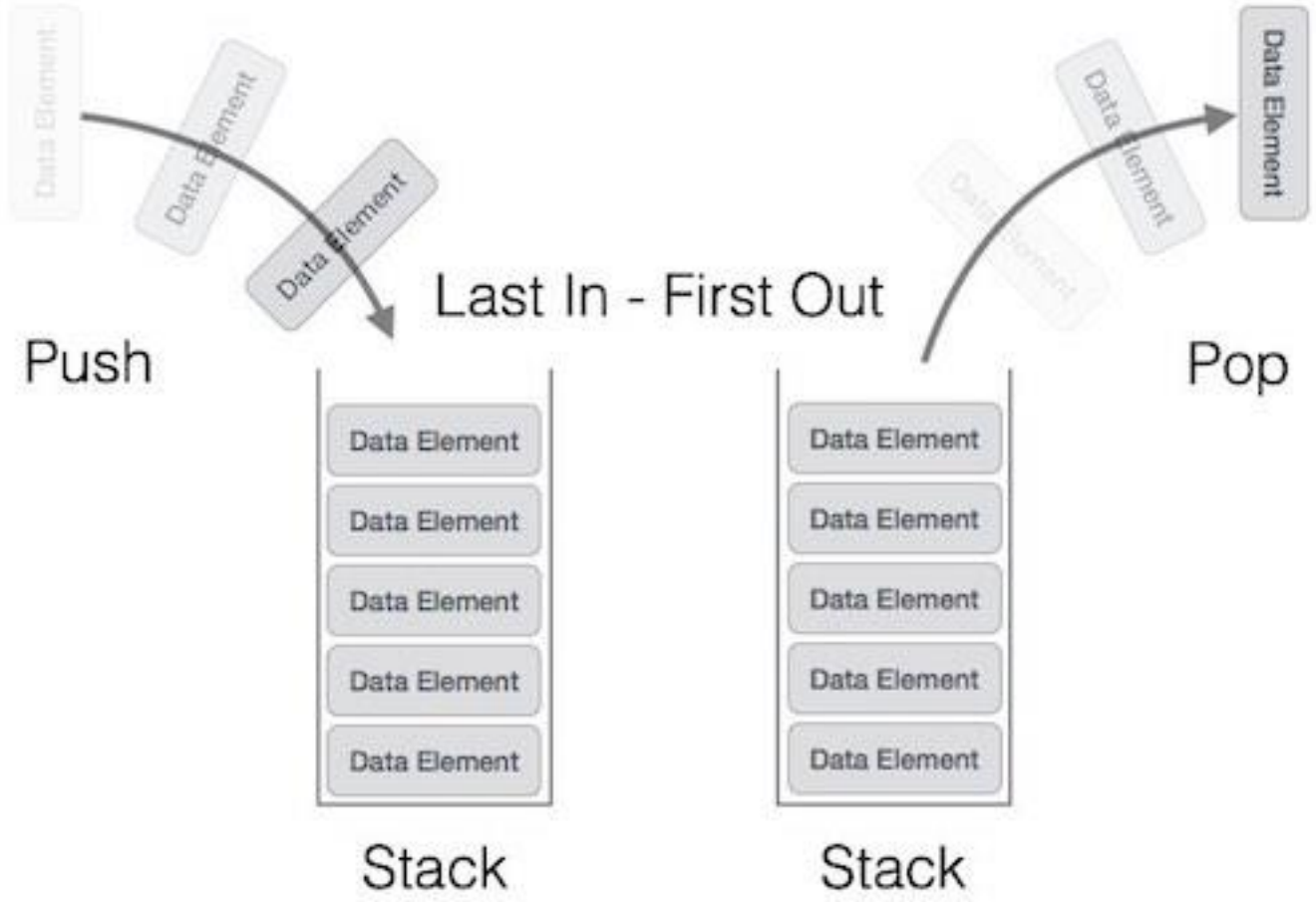
Стек

доц. д-р Нора Ангелова

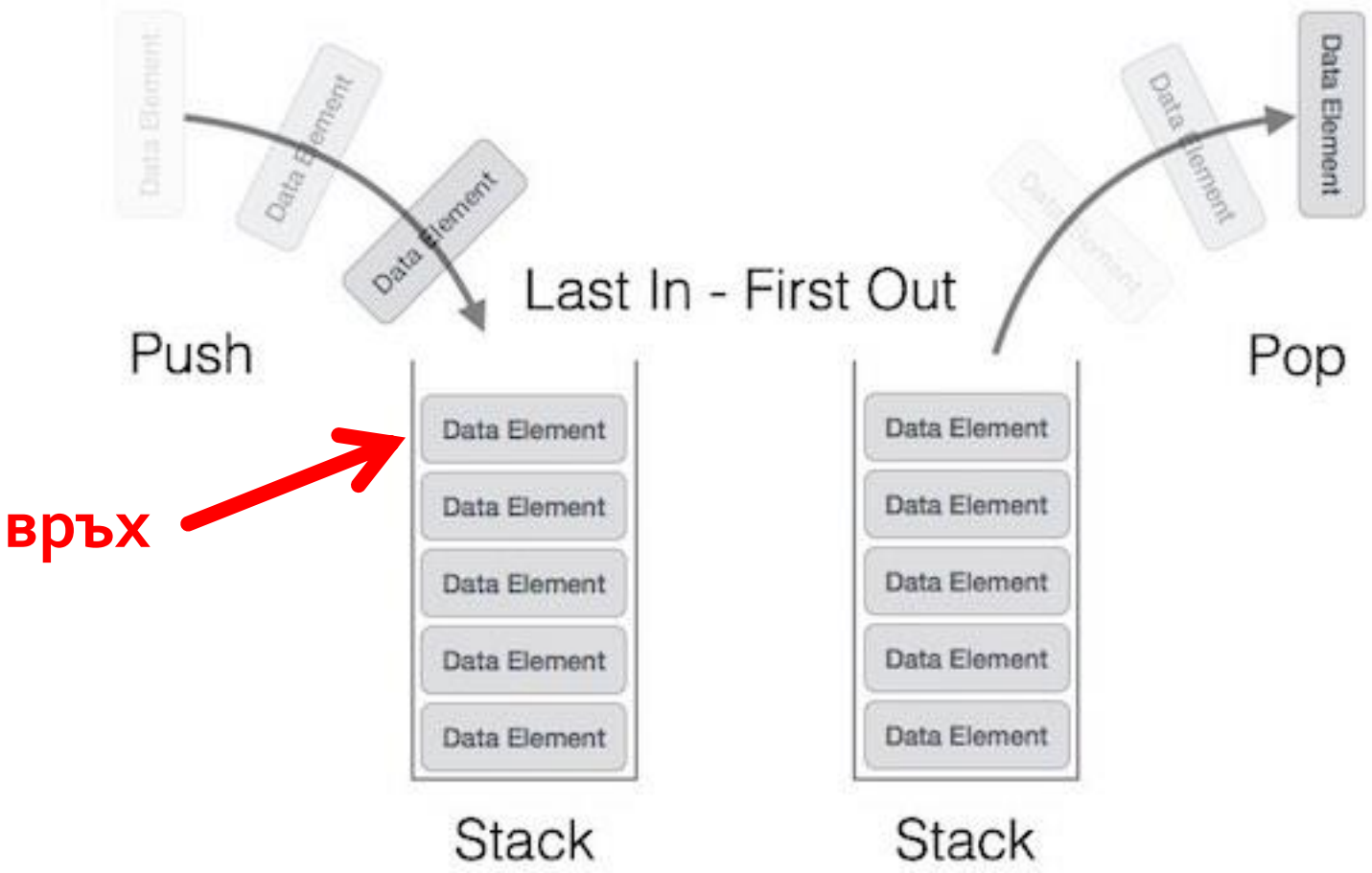
Стек

- Съставна хомогенна линейна структура от данни
- „последен влязъл - пръв излязъл“ (LIFO)



Стек

- Съставна хомогенна линейна структура от данни
- „последен влязъл - пръв излязъл“ (LIFO)



Стек

Логическо представяне

- Крайна редица от елементи от един и същ тип.
- Операции – операциите включване и изключване са допустими само за върха на стека.
- Достъп – възможен е достъп само до елемента, намиращ се на върха на стека.
  Достъпът е **пряк**.

Стек

Операции:

- empty() – проверка дали стекът е празен.
- push(x) – включване на елемент на стек.
- pop() – изключване на елемент от стек.
- top() – връщане на стойността на върха на стека.

Стек

Физическо представяне

- Последователно
- Свързано

Стек

Последователно представяне

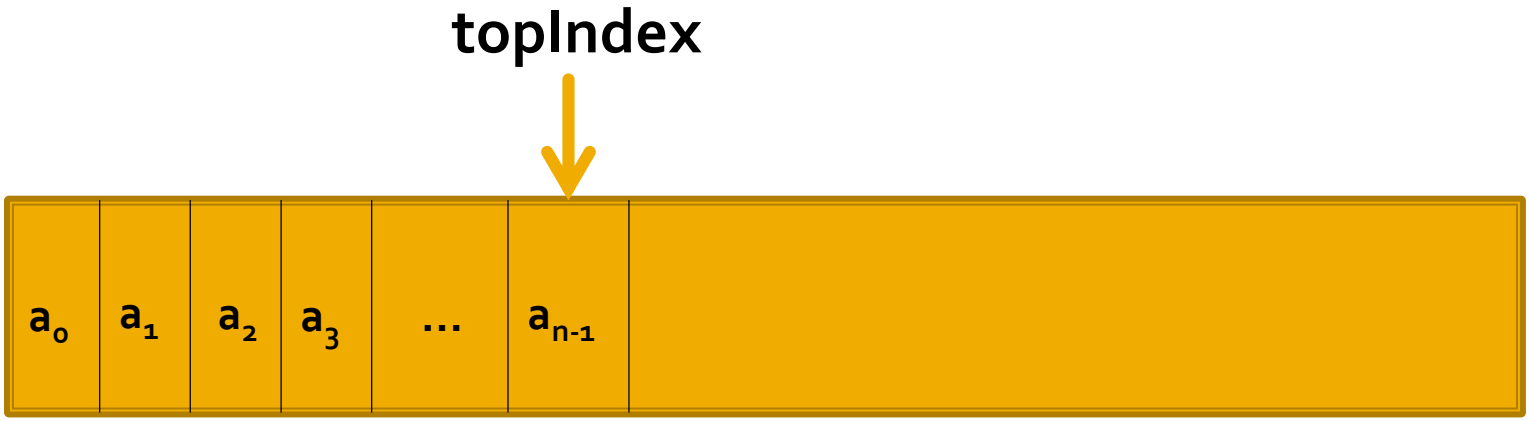
- Запазва се блок от памет, в който стекът расте и се съкращава.

Стек

- Реализации

Стек - последователно представяне

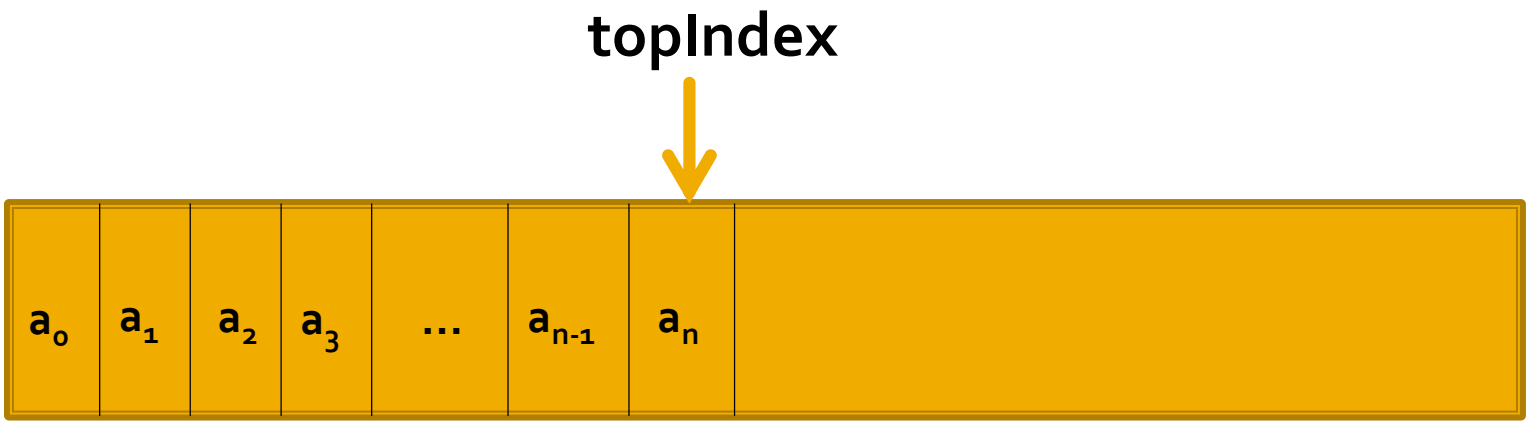
- Масив



Стек - последователно представяне

Реализация с масив

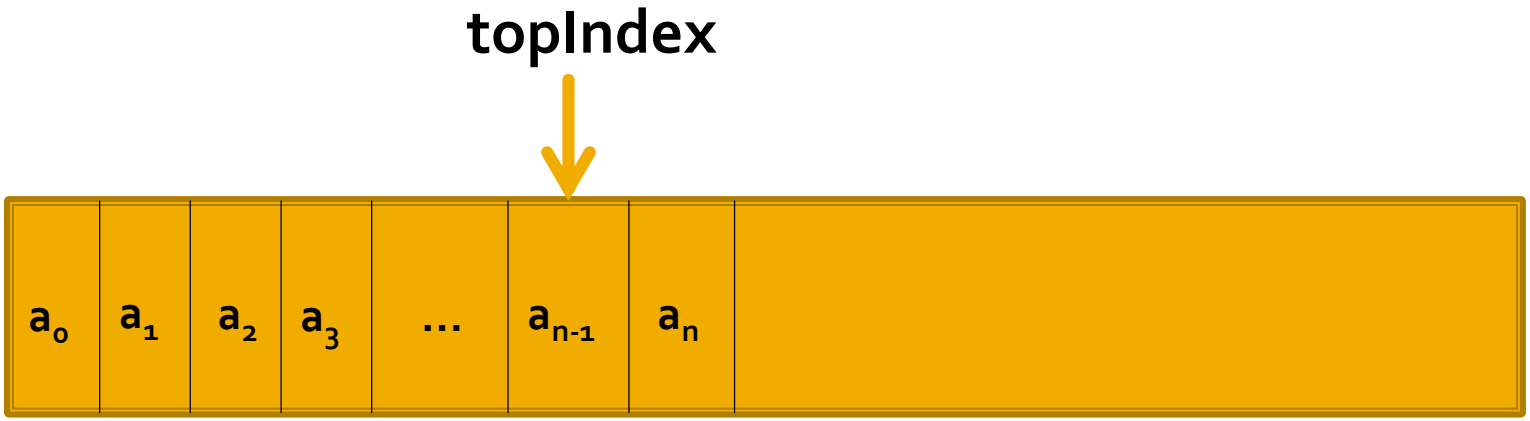
- push($\mathbf{a_n}$) – включва елемента $\mathbf{a_n}$



Стек - последователно представяне

Реализация с масив

- push($\mathbf{a_n}$) – включва елемента $\mathbf{a_n}$
- pop() – изключва елемент



Стек

```
const int MAX = 100;

template <typename T>

class Stack {
  private:
  T a[MAX];
  int topIndex;           // Индекс на върха на стека
  bool full() const;

  public:
  Stack();                // Създаване на празен стек

  bool empty() const;     // Проверка дали стек е празен
  void push(T const& x); // Включване на елемент
  T pop();                // Изключване на елемент
  T top() const;          // Достъп до върха на стека
};
```

Стек

```
template <typename T>
Stack<T>::Stack() : topIndex(-1)
{}

template <typename T>
bool Stack<T>::empty() const {
  return topIndex == -1;
}

template <typename T>
bool Stack<T>::full() const {
  return topIndex == MAX - 1;
}
```

Стек

```
template <typename T>
void Stack<T>::push(T const& x) {
  if (full()) {
    cerr << "Включване в пълен стек!\n";
  } else {
    a[++topIndex] = x;
  }
}

template <typename T>
T Stack<T>::pop() {
  if (empty()) {
    cerr << "Изкл. на елемент от празен стек!\n";
    return T();
  }
  return a[topIndex--];
}
```

Стек

```
template <typename T>
T Stack<T>::top() const {
  if (empty()) {
    cerr << "Достъп до върха на празен стек!\n";
    return T();
  }
  return a[topIndex];
}
```

Стек

- Предимства и недостатъци на решението
  - Методите top && pop не хвърлят грешка, а връщат стойност по подразбиране за съответния тип – програмата ще продължи да се изпълнява във всички случаи, но грешката ще се отчете единствено от съобщението в cerr.
  - Липсват полета за брой на елементите и капацитет в класа – паметта за всеки обект ще бъде по-малка, но се използва външна константа и трябва да се правят проверки по индекс.
  - Методът full може да бъде част от интерфейсът на класа, ако той е със статичен капацитет.
  - Предполага се, че за всеки тип T са реализирани елементите от голямата петица.

Стек с преоразмеряване

- Реализация на стек с преоразмеряване

Стек

```
template <typename T>
class RStack {
private:
  T* arr;
  int topIndex;                    // Индекс на последния елемент в стека
  int capacity;                    // Капацитет на стека

  bool full() const;               // Проверка дали стек е пълен
  void resize();                   // Разширяване на стек
  void copy(T const*);             // Копиране на паметта на стек (до capacity)
  void eraseStack();               // Изтриване на паметта
  void copyStack(RStack const&); // Копиране на стек

public:
  RStack();
  RStack(RStack const&);
  RStack& operator=(RStack const&);
  ~RStack();

  bool empty() const;
  void push(T const& x);
  T pop();
  T top() const;
};
```

Стек

```
const  INITIAL_CAPACITY = 16;

template <typename T>
RStack<T>::RStack() : topIndex(-1), capacity(INITIAL_CAPACITY) {
  arr = new T[capacity];
}

template <typename T>
bool RStack<T>::empty() const {
  return topIndex == -1;
}

template <typename T>
bool RStack<T>::full() const {
  return topIndex == capacity - 1;
}
```

Стек

```
template <typename T>
T RStack<T>::pop() {
  if (empty()) {
    cerr << "Изключване на елемент от празен стек!\n";
    return T();
  }
  return arr[topIndex--];
}

template <typename T>
T RStack<T>::top() const {
  if (empty()) {
    cerr << "Достъп до върха на празен стек!\n";
    return T();
  }
  return arr[topIndex];
}
```

Стек

```
template <typename T>
void RStack<T>::eraseStack() {
  delete[] arr;
}

template <typename T>
RStack<T>::~RStack() {
  eraseStack();
}
```

Стек

```
template <typename T>
void RStack<T>::copy(T const* stackArr) {
  for(i = 0; i < capacity; i++) {
    arr[i] = stackArr[i];
  }
}

template <typename T>
void RStack<T>::copyStack(RStack const& stack) {
  topIndex = stack.topIndex;
  capacity = stack.capacity;
  arr = new T[capacity];
  copy(stack.arr);
}
```

Стек

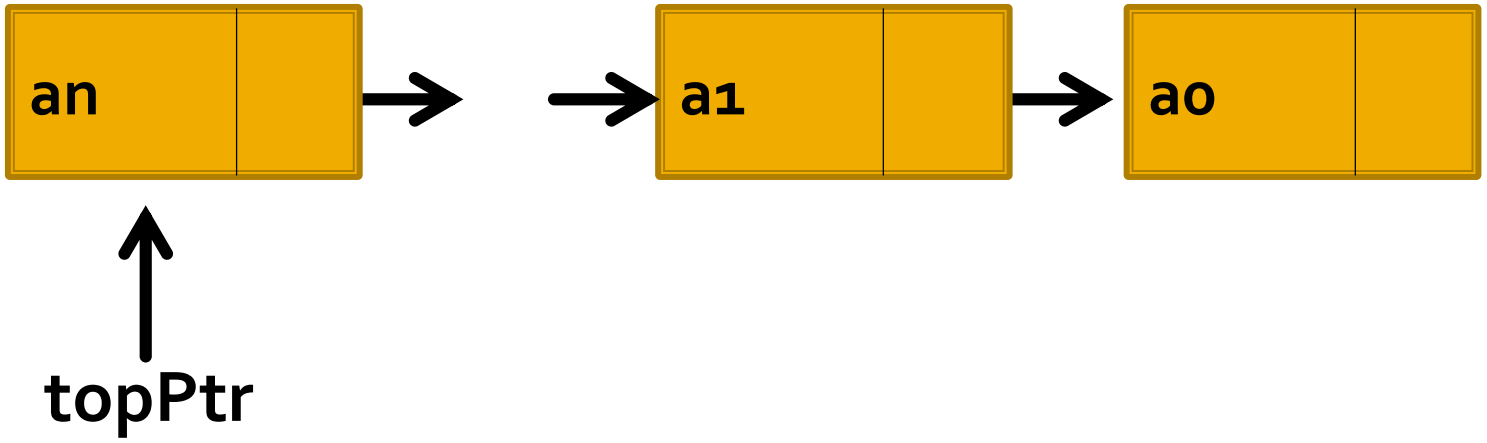
```
template <typename T>
void RStack<T>::push(T const& x) {
  if (full()) {
    resize();
  }
  arr[++topIndex] = x;
}

template <typename T>
void RStack<T>::resize() {
  T* oldStackPtr = arr;
  arr = new T[2 * capacity];
  copy(oldStackPtr);
  capacity *= 2;          // Удвояване на капацитета
  delete[] oldStackPtr; // Изтриване на старата памет
}
```

Стек

```
template <typename T>
RStack<T>::RStack(RStack<T> const& stack) {
  copyStack(stack);
}

template <typename T>
RStack<T>& RStack<T>::operator=(RStack<T> const& stack) {
  if (this != &stack) {
    eraseStack();
    copyStack(stack);
  }
  return *this;
}
```

Стек

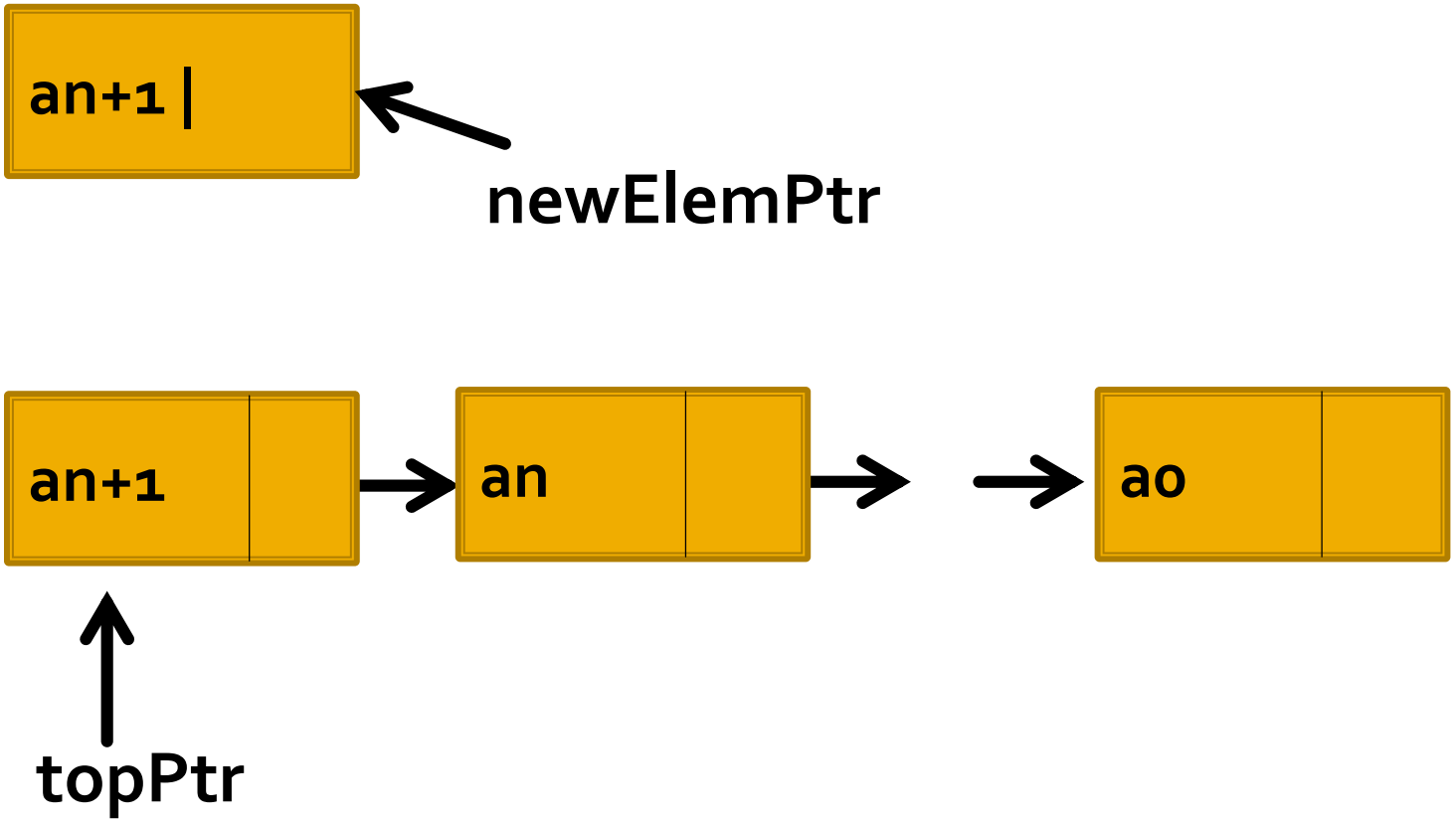
Свързано представяне



Стек

Свързано представяне

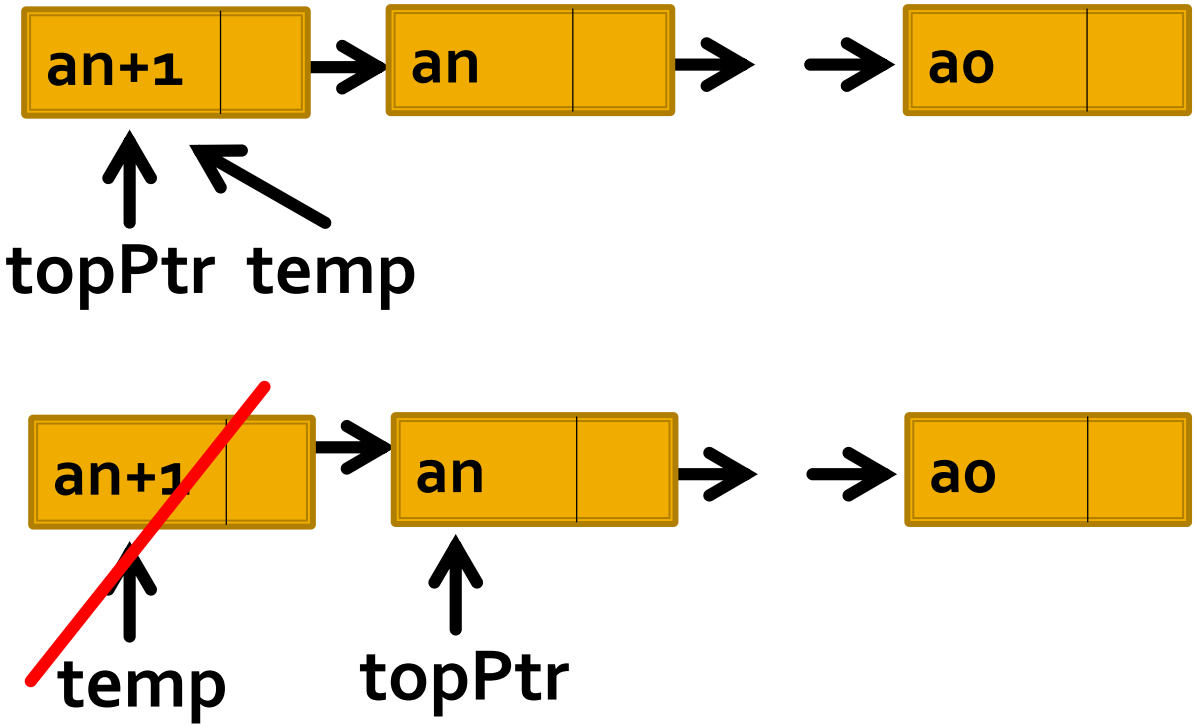
- Добавяне на елемент



Стек

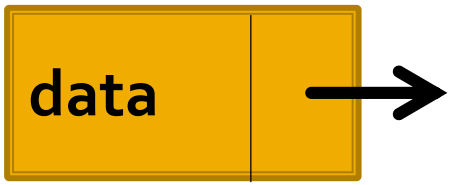
Свързано представяне

- Добавяне на елемент
- Премахване на елемент



Стек – свързано представяне

```
template <typename T>
struct StackElement {
  T data;
  StackElement<T>* link;
};
```

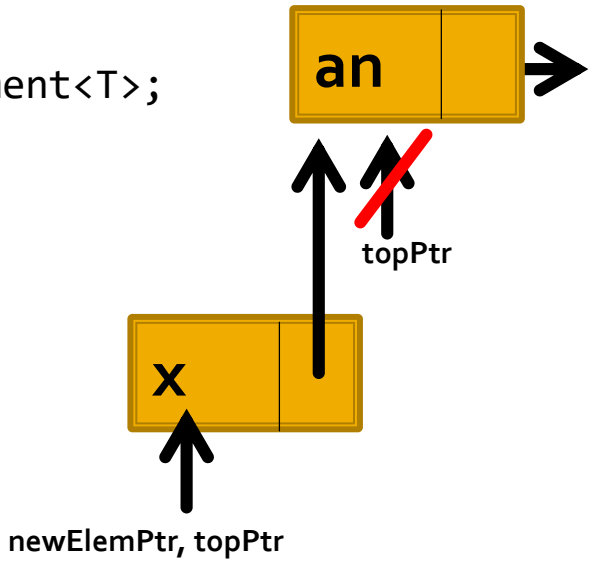


Стек

```
template <typename T>
class LStack {
private:
  StackElement<T>* topPtr;
  void copy(StackElement<T>*);
  void eraseStack();
  void copyStack(LStack const&);

public:
  LStack();                              // Създаване на празен стек
  LStack(LStack const&);                 // Конструктор за копиране
  LStack& operator=(LStack const&); // Операция за присвояване
  ~LStack();                             // Деструктор

  bool empty() const;
  void push(T const& x);
  T pop();
  T top() const;
};
```

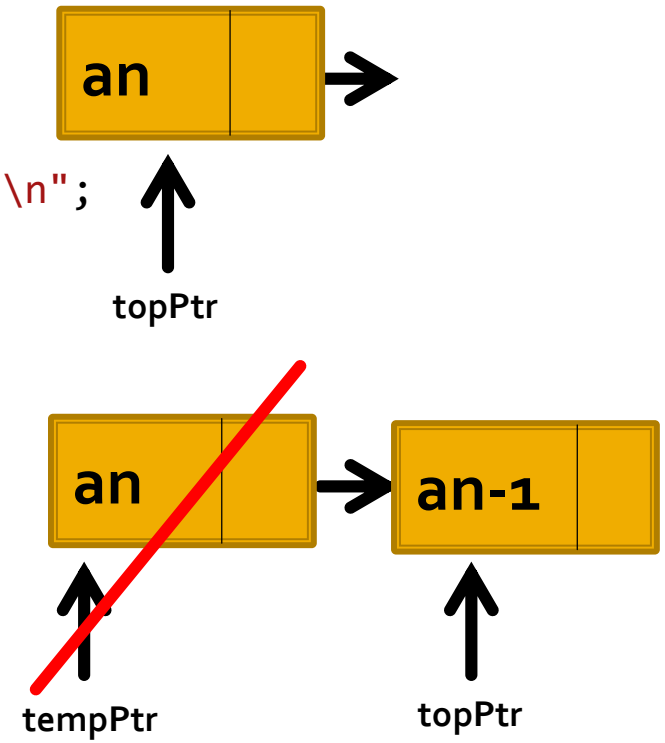
Стек

```
template <typename T>
LStack<T>::LStack() : topPtr(nullptr) {}

template <typename T>
bool LStack<T>::empty() const {
  return topPtr == nullptr;
}

template <typename T>
void LStack<T>::push(T const& x) {
  StackElement<T>* newElemPtr = new StackElement<T>;
  newElemPtr->data = x;
  newElemPtr->link = topPtr;
  topPtr = newElemPtr;
}
```



Стек

```
template <typename T>
T LStack<T>::pop() {
  if (empty()) {
    cerr << "Изкл. на елемент от празен стек!\n";
    return T();
  }

  StackElement<T>* tempPtr = topPtr;
  topPtr = topPtr->link;
  T x = tempPtr->data;

  delete tempPtr;

  return x;
}
```



Стек

```
template <typename T>
T LStack<T>::top() const {
  if (empty()) {
    cerr << "празен стек!\n";
    return T();
  }
  return topPtr->data;
}

template <typename T>
void LStack<T>::eraseStack() {
  while (!empty()) {
    pop();
  }
}

template <typename T>
LStack<T>::~LStack() {
  eraseStack();
}
```

Стек

```
template <typename T>
void LStack<T>::copy(StackElement<T>* toCopy) {
  if (toCopy == nullptr) {
    return;
  }

  copy(toCopy->link); // Копираме стека от втория елемент нататък
  push(toCopy->data); // Добавяме първия елемент отгоре
}

template <typename T>
void LStack<T>::copyStack(LStack const& stack) {
  topPtr = nullptr;
  copy(stack.topPtr);
}
```

Стек

```
// Вариант 1
template <typename T>
void LStack<T>::copyStack(const LStack<T>& stack) {
    topPtr = nullptr;

    if (stack.empty()) {
        return;
    }
    StackElement<T> * lastCopied, toCopy, copied;
    lastCopied = new StackElement<T>;
    lastCopied->data = stack.topPtr->data;

    topPtr = lastCopied;
    StackElement<T> * toCopy = stack.topPtr->link;
    while (toCopy) {
      copied = new StackElement<T>;
      copied->data = toCopy->data;
      lastCopied->link = copied;
      lastCopied = copied;
      toCopy = toCopy->link;
    }
    lastCopied->link = nullptr;
}
```

// Сложност: O(n)

Стек

```
// Вариант 2
template <typename T>
void LStack<T>::copy (StackElement<T>* toCopy) {

    if (toCopy == nullptr) {
        return;
    }

    copy(toCopy->link);
    push(toCopy->data);
}
```

```
template <typename T>
void LStack<T>::copyStack(const LStack<T>& stack) {
    topPtr = nullptr;
    copy(stack.topPtr);
}
```

// Сложност: O(n)

Стек

```
// Вариант 3
```

- Чрез използване на функции push and pop

// Сложност: O(n)

Стек

```
template <typename T>
LStack<T>::LStack(LStack const& stack) {
  copyStack(stack);
}

template <typename T>
LStack<T>& LStack<T>::operator=(LStack const& stack) {
  if (this != &stack) {
    eraseStack();
    copyStack(stack);
  }
  return *this;
}
```

STL (Standard Template Library)

Библиотека от шаблони, реализираща стандартни структури от данни и алгоритми.

- част от C++ Standard Library

Основни компоненти:

- алгоритми (<algorithm>)
- контейнери (<stack>, <queue>, <list>)
- функционални обекти (<functional>)

** Дава гаранции за сложност на алгоритми и операции над СД*

STL (Стек)

`std::stack<T>`

#include <stack>

Интерфейс:

- stack() — създаване на празен стек
- empty() — проверка за празнота на стек
- push(x) — включване на елемент на стек
- pop() — изключване на елемент от стек (void)
- top() — последен елемент на стека (reference || const_reference)
- size() — дължина на стека

- ==,!=,,<=,>= — лексикографско сравнение на два стека

Следва продължение...