Стек, Опашка

Изготвил:
гл.ас. д-р Нора Ангелова

Какво ще стане ако...

```
int main()
{
    char *temp = NULL;
    delete temp;

    system("pause");
    return 0;
}
```

OK

Какво ще стане ако...

```
int main()
{
    char *temp = NULL;
    strlen(temp);

    system("pause");
    return 0;
}
```

ERROR

```
class Student {
    char* name;
public:
    Student(char* studentName) {
        name = new char[strlen(studentName)+1];
        strcpy(name, studentName);
    }

    Student& operator=(Student const& student1) {
        cout << "operator =" << student1.name << endl;
        return *this;
    }
//...
};

int main()
{
    Student a("a");
    Student b("b");
    Student c("c");
    a = b = c;

    system("pause");
    return 0;
}
```

operator =c

operator =b

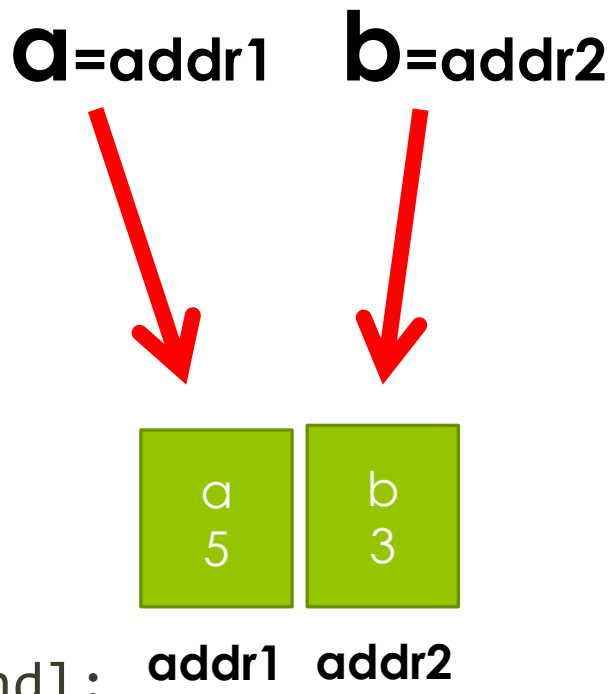
```
class Student {
    char* name;
public:
    Student(char* studentName) {
        name = new char[strlen(studentName)+1];
        strcpy(name, studentName);
    }

    Student& operator=(Student const& student1) {
        cout << "operator =" << student1.name << endl;
        return *this;
    }
//...
};

int main()
{
    Student a("a");
    Student b("b");
    Student c("c");
    (a = b) = c;

    system("pause");
    return 0;
}
```

operator =b

operator =c

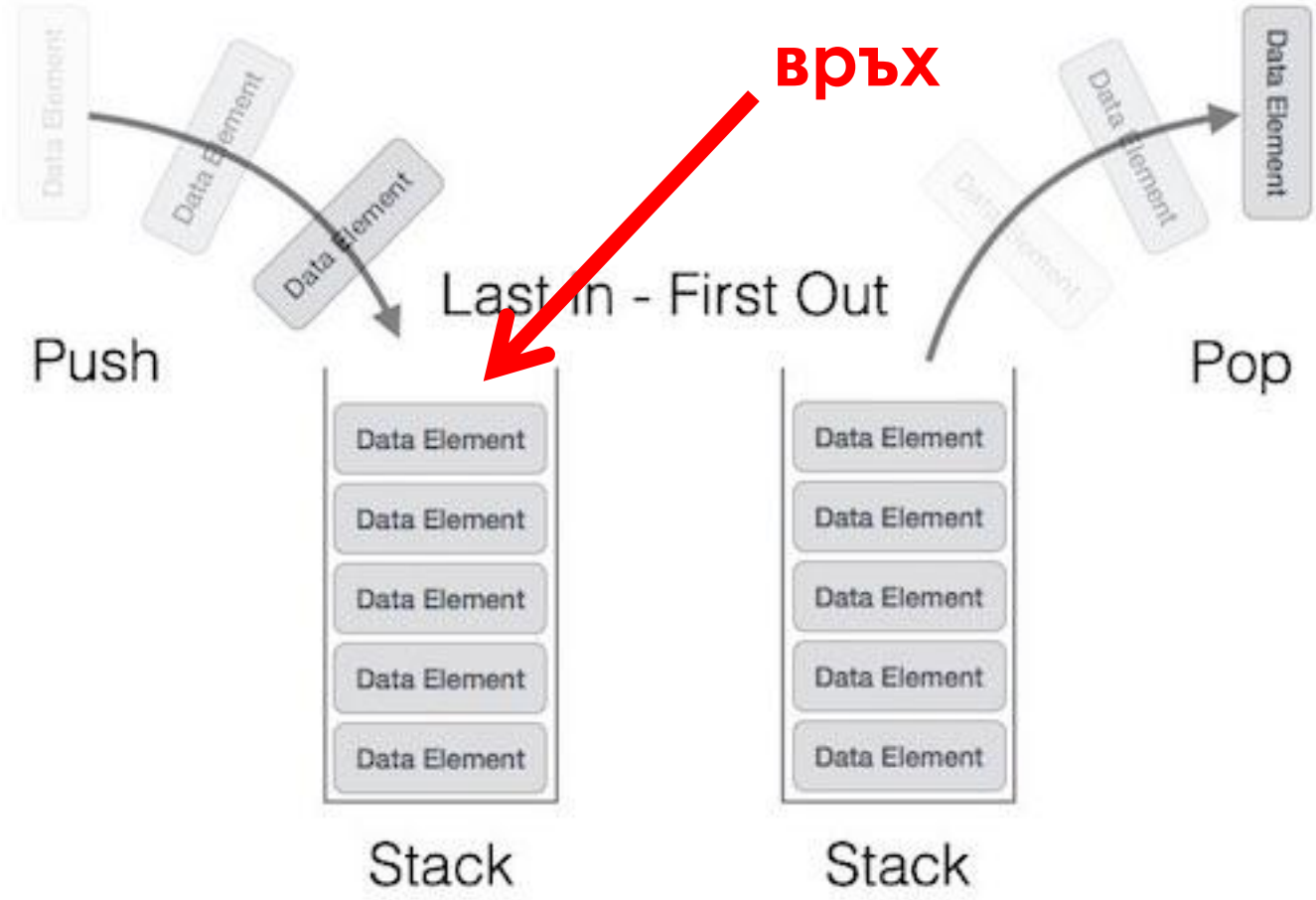
Задача

```
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int a = 5, b = 3;
    swap(&a, &b);
    cout << a << " " << b << endl;

    system("pause");
    return 0;
}
```



Стек

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



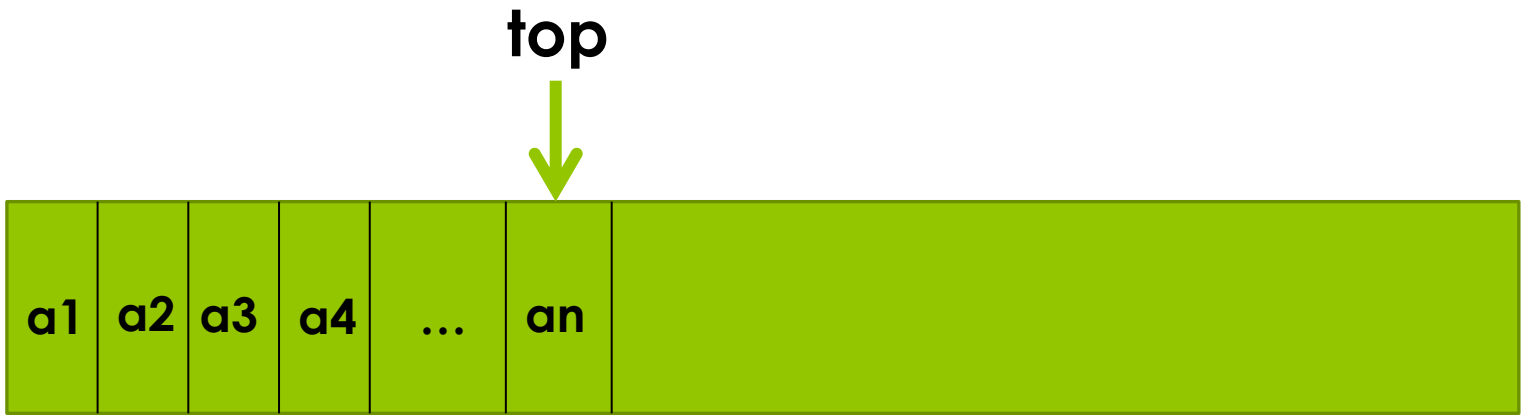
Стек

Операции:

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

Последователно представяне

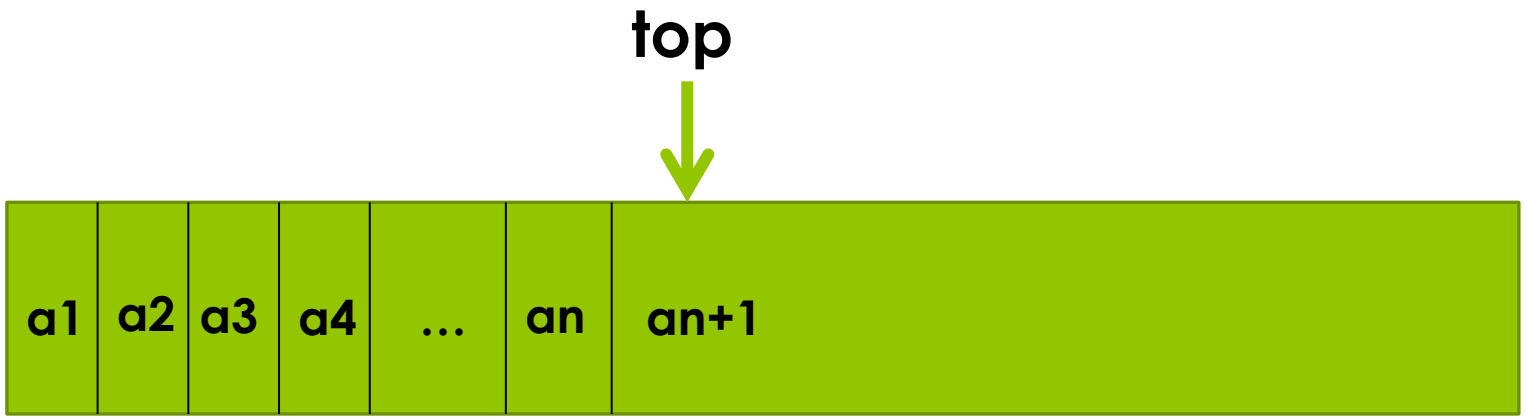
Масив



Последователно представяне

Масив

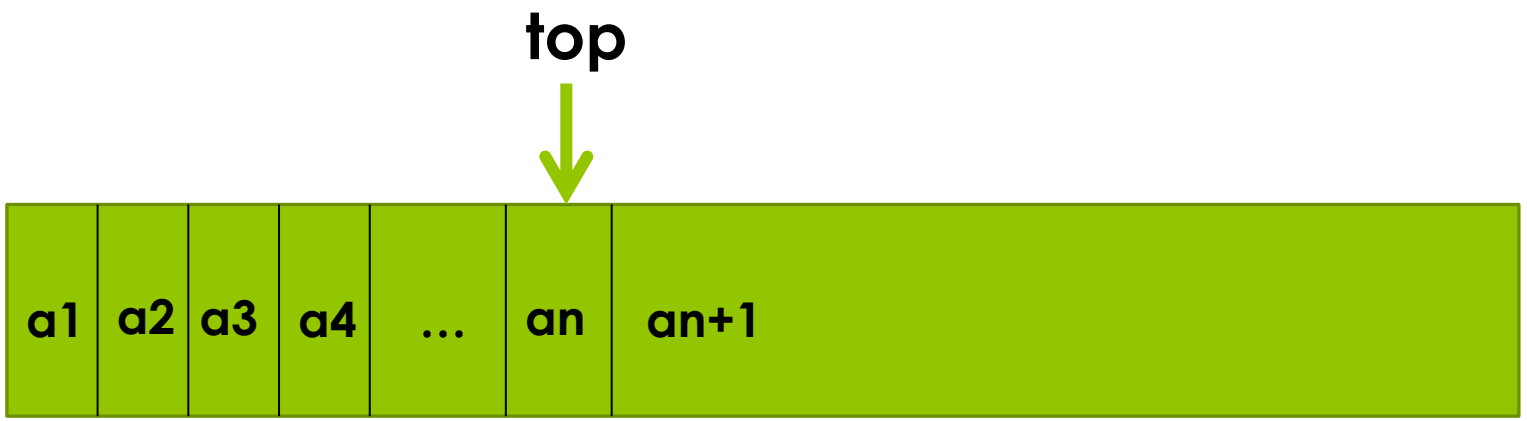
- push() - включване на елемент



Последователно представяне

Масив

- push() - включване на елемент
- pop() - изключване на елемент



Стек

```
const int MAX = 100;

class Stack {
private:
  int 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(int const& x); // включване на елемент
  int pop(); // изключване на елемент
  int top() const; // връща върха на стека
};
```

Стек

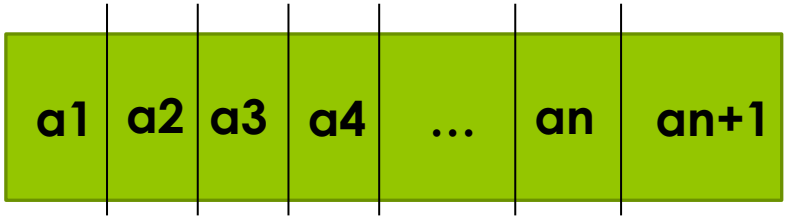
```
Stack::Stack() : topIndex(-1) {}

bool Stack::empty() const {
  return topIndex == -1;
}

bool Stack::full() const {
  return topIndex == MAX - 1;
}

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

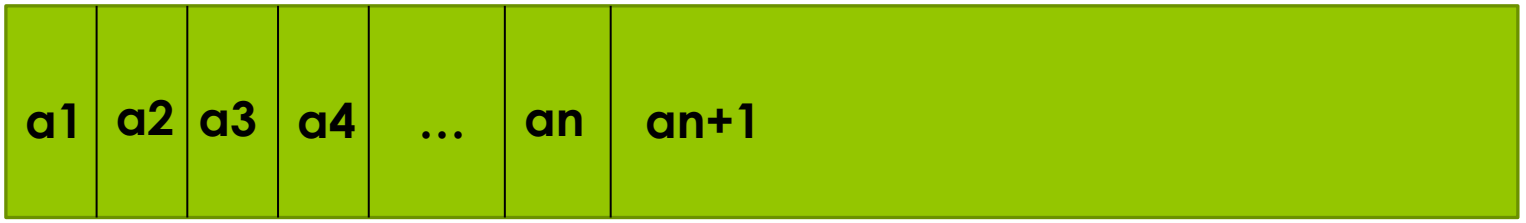
Стек

```
int Stack::pop() {
  if (empty()) {
    cerr << “Изкл. на елемент от празен стек!\n";
    return 0;
  }
  return a[topIndex--];
}

int Stack::top() const {
  if (empty()) {
    cerr << “Достъп до върха на празен стек!\n";
    return 0;
  }
  return a[topIndex];
}
```

Стек

- Динамичен масив





Стек

```
class RStack {
private:
  int* a;
  int topIndex;  // Индекс на последния елемент в стека
  int capacity; // Капацитет на стека

  bool full() const; // Проверка дали стекът е пълен
  void resize(); // Разширяване на стека
  void copy(int const*); // Копиране на паметта на стека от друго място
  void eraseStack(); // Изтриване на паметта
  void copyStack(RStack const&); // Копиране на стека

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

  bool empty() const;
  void push(int const& x);
  int pop();
  int top() const;
};
const int INITIAL = 16;
```

Стек

```
RStack::RStack() : topIndex(-1),  capacity(INITIAL) {
  a = new int[capacity];
}

bool RStack::empty() const {
  return topIndex == -1;
}

bool RStack::full() const {
  return topIndex == capacity - 1;
}

int RStack::pop() {
  if (empty()) {
    cerr << "изключване на елемент от празен стек!\n";
  return 0;
  }
  return a[topIndex--];
}
```

Стек

```
int RStack::top() const {
  if (empty()) {
    cerr << "достъп до върха на празен стек!\n";
    return 0;
  }
  return a[topIndex];
}

void RStack::eraseStack() {
  delete[] a;
}

RStack::~RStack() {
  eraseStack();
}
```

Стек

```
void RStack::push(int const& x) {
  If (full()) {
    resize();
  }
  a[++topIndex] = x;
}

void RStack::resize() {
  int* olda = a;
  a = new int[2 * capacity];
  copy(olda);
  capacity *= 2; // удвояването на  капацитета
  delete[] olda; // Изтриваме старата памет
}
```

Стек

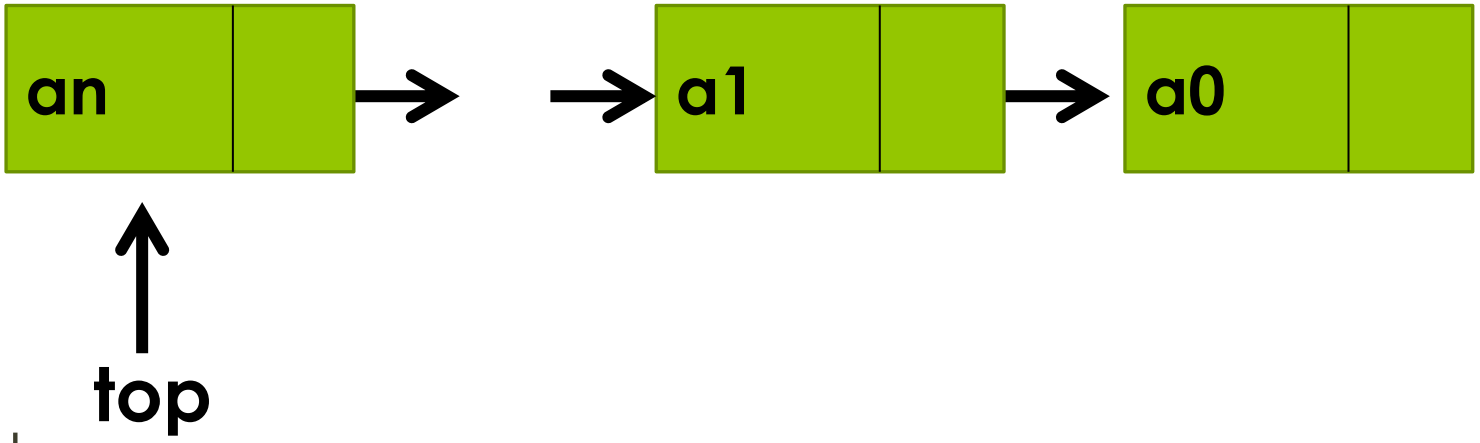
```
void RStack::copy(int const* other) {
  for(int i = 0; i < capacity; i++)
    a[i] = other[i];
}

void RStack::copyStack(RStack const& rs) {
  topIndex = rs.topIndex;
  capacity = rs.capacity;
  a = new int[capacity];
  copy(rs.a);
}

RStack::RStack(RStack const& rs) {
  copyStack(rs);
}
```

Стек

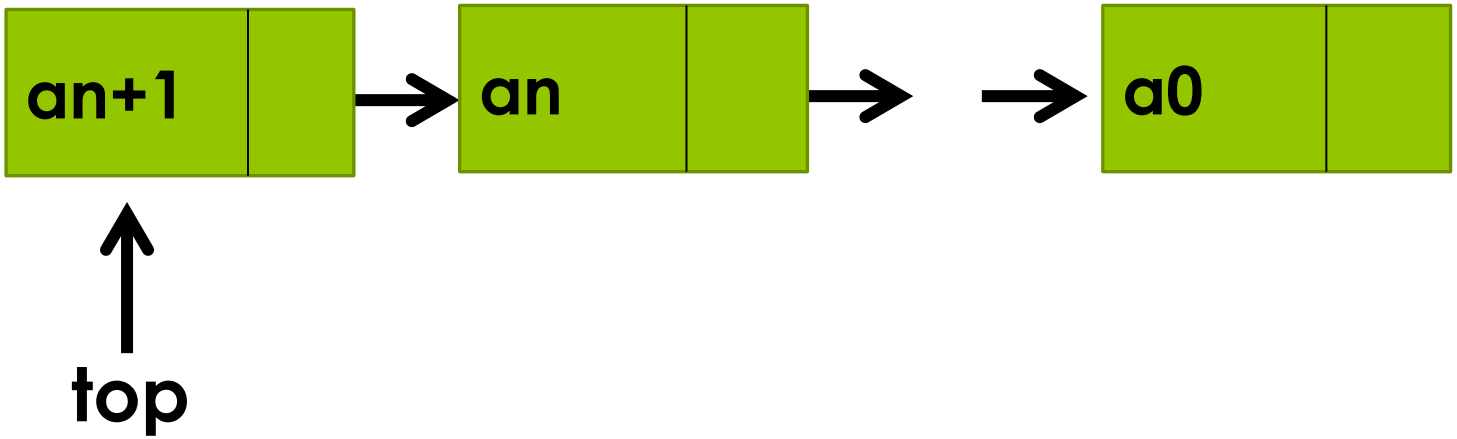
```
RStack& RStack::operator=(RStack const& rs) {
  if (this != &rs) {
    eraseStack();
    copyStack(rs);
  }
  return *this;
}
```

Стек

- Свързано представяне

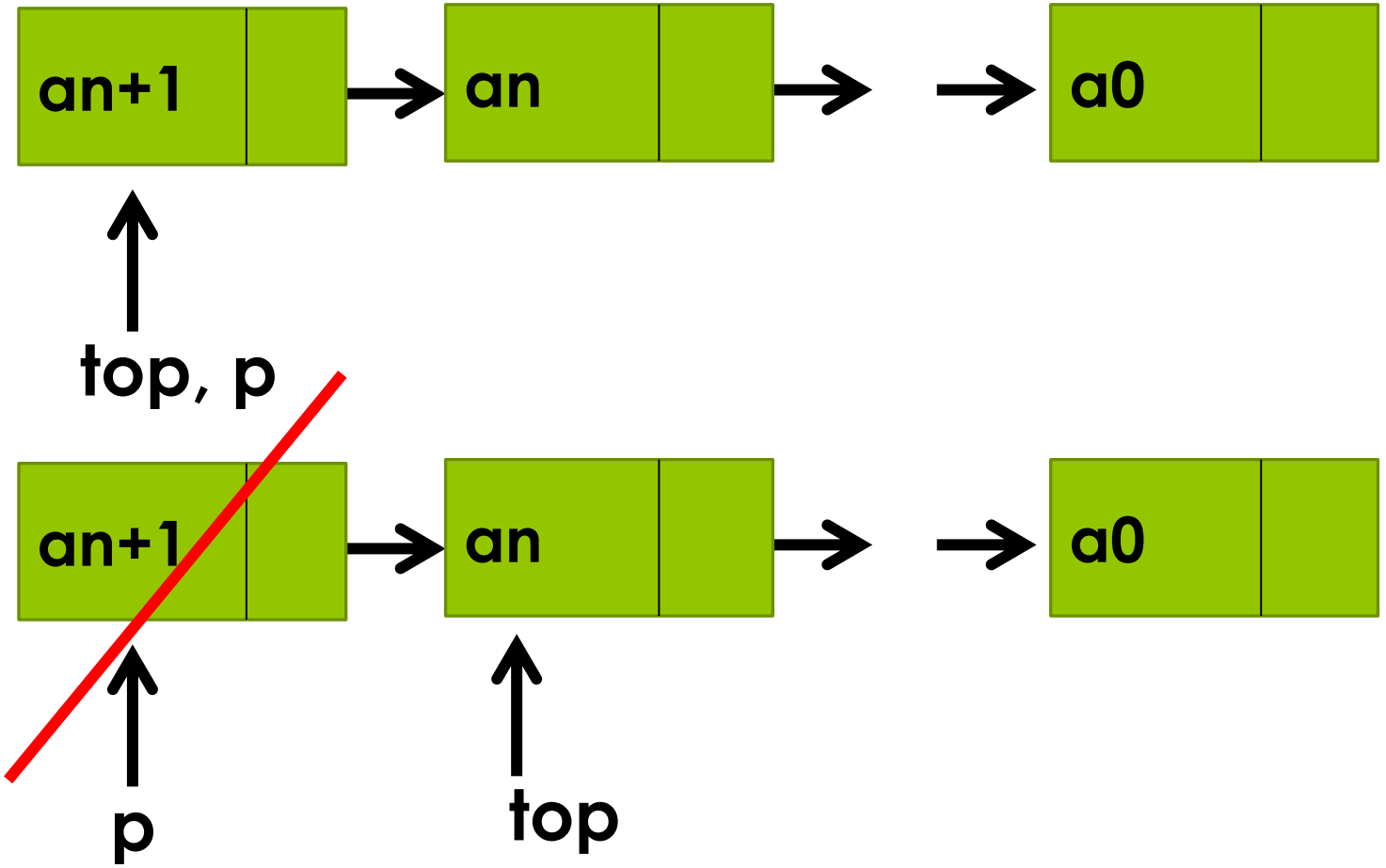


- push



Стек

- Свързано представяне
- pop



Стек

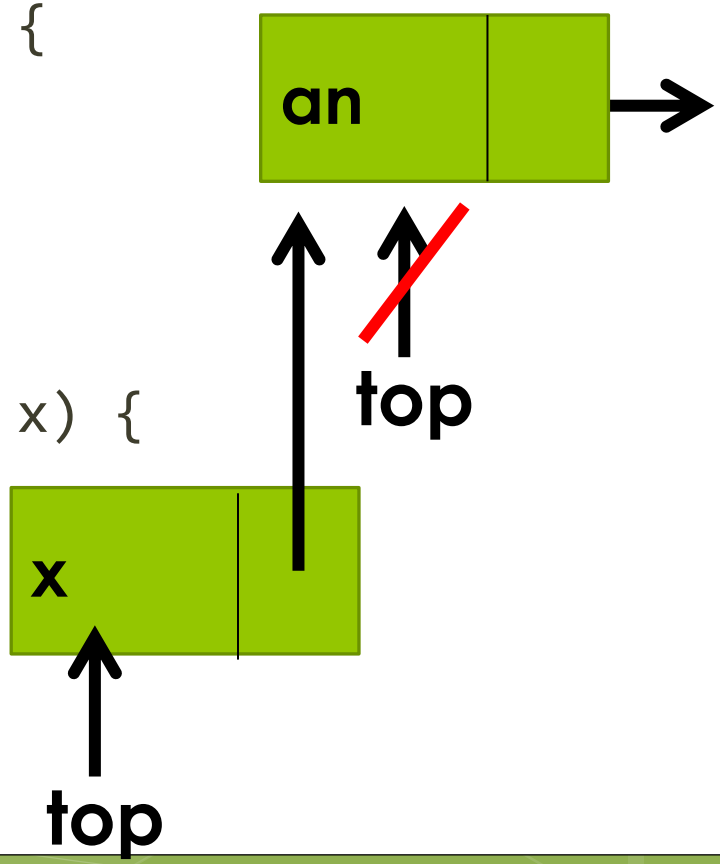
```
template <typename T>
struct node {
  T data;
  node* next;
};
```

Стек

```
template <typename T>
class LStack {
private:
  node<T>* topNode;
  void copy(node<T>* toCopy);
  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;
};
```

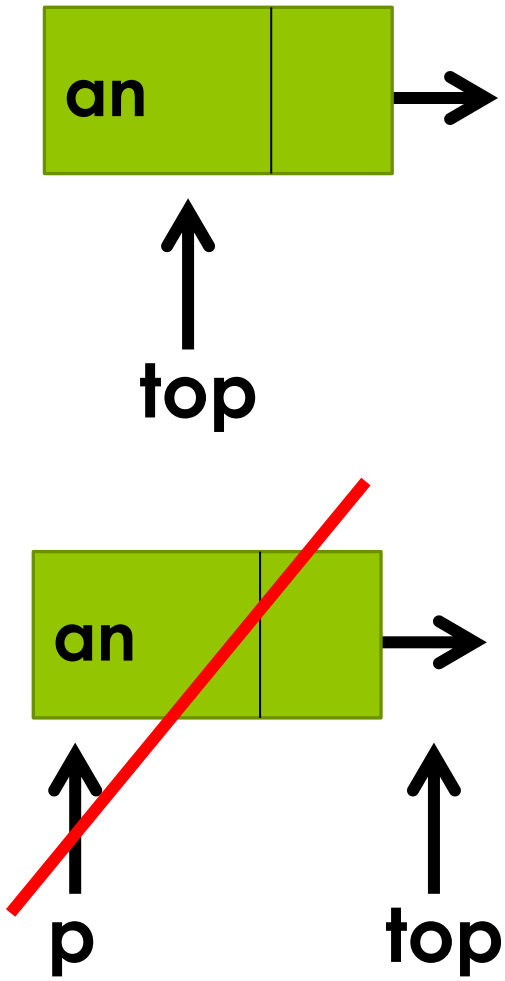
Стек

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

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

template <typename T>
void LStack<T>::push(T const& x) {
  node<T>* p = new node<T>;
  p->data = x;
  p->next = topNode;
  topNode = p;
}
```



Стек

```
template <typename T>
T LStack<T>::pop() {
  if (empty()) {
    cerr << "празен стек!\n";
    return 0;
  }
  node<T>* p = topNode;
  topNode = topNode->next;
  T x = p->data;
  delete p;
  return x;
}
```



Стек

```
template <typename T>
T LStack<T>::top() const {
  if (empty()) {
    cerr << "празен стек!\n";
    return 0;
  }
  return topNode->data;
}

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

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

Стек

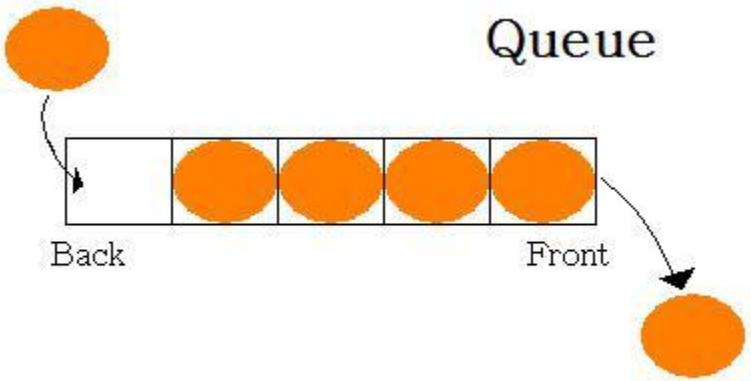
```
template <typename T>
void LStack<T>::copy(node<T>* toCopy) {
  if (toCopy == nullptr)
    return;
  copy(toCopy->next);
  push(toCopy->data); //добавяме първия елемент отгоре
}

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

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

Стек

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

Опашка

- Хомогенна линейна структура от данни
- „първи влязъл - пръв излязъл“ (FIFO)



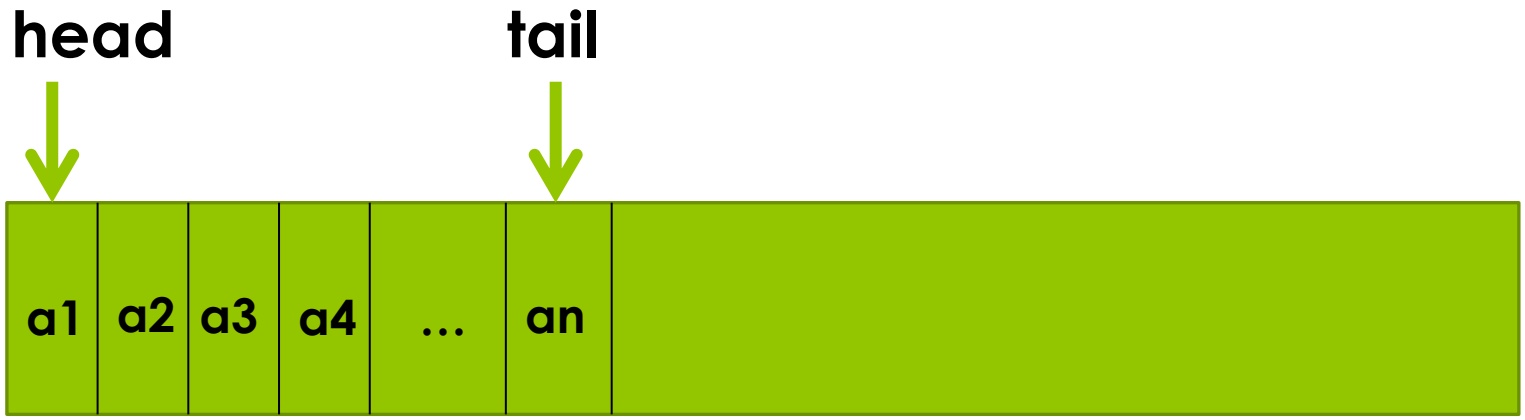
Опашка

Операции:

- empty() – проверка дали опашката е празна
- push(x) – включване на елемент в опашката
- pop() – изключване на елемент от опашката
- head() – връщане на „главата“ на опашката

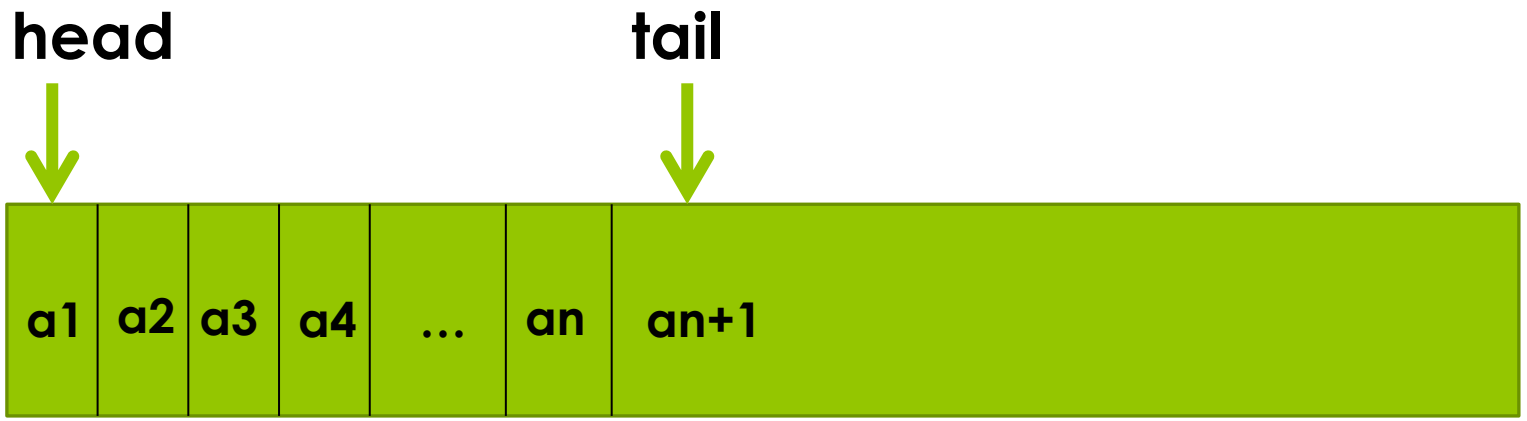
Последователно представяне

- Масив
- head/tail or front/rear



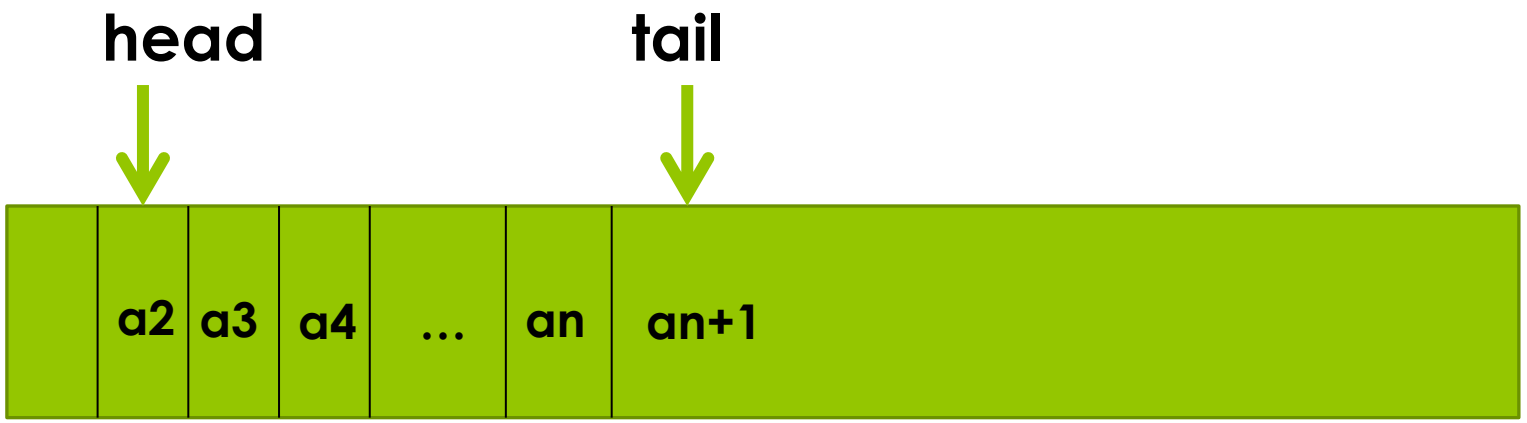
Последователно представяне

- Масив
- head/tail or front/rear
- push() - включване на елемент



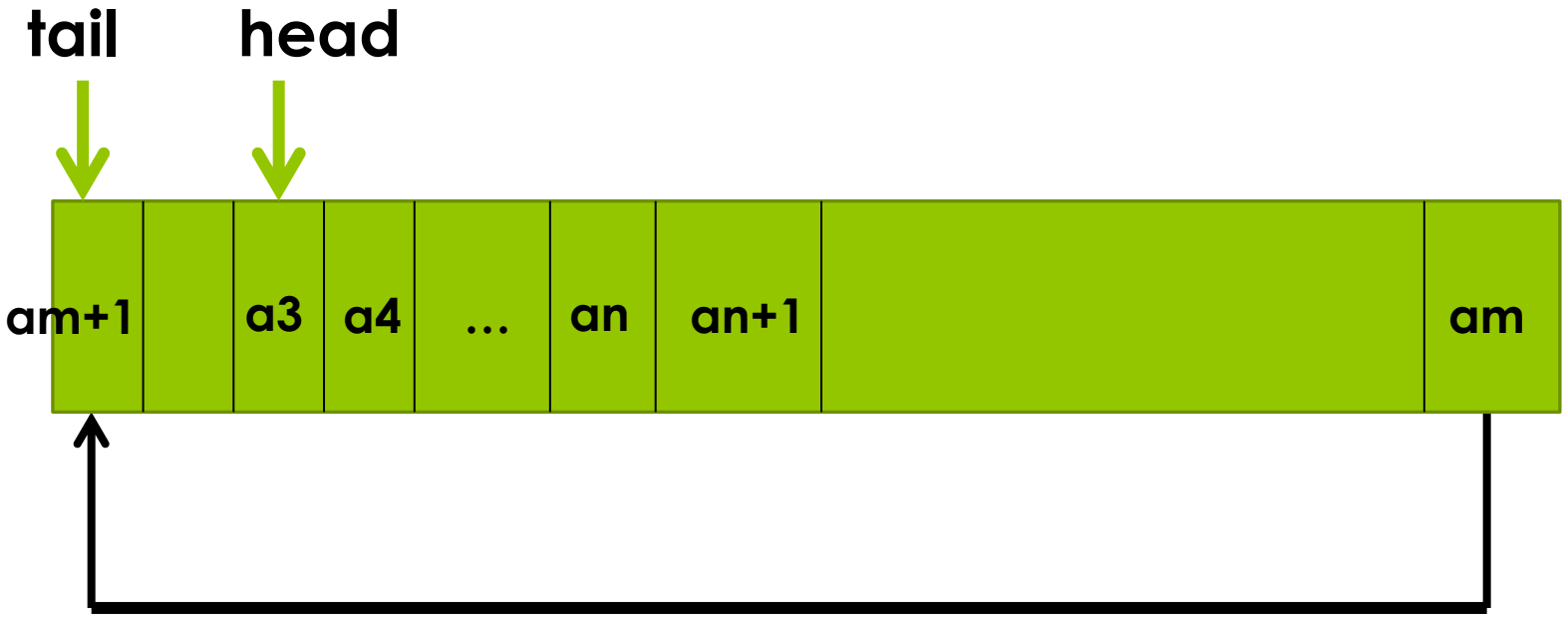
Последователно представяне

- Масив
- head/tail or front/rear
- push() – включване на елемент
- pop() – изключване на елемент



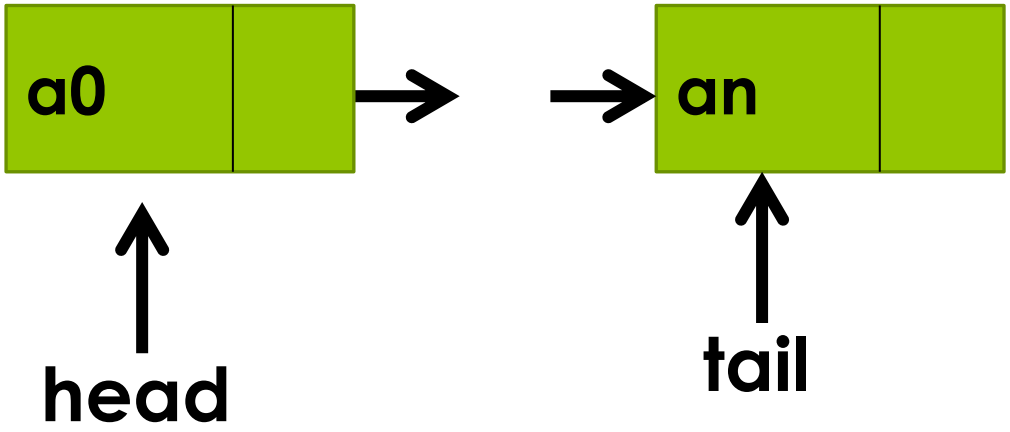
Последователно представяне

- head/tail or front/rear
- цикличност

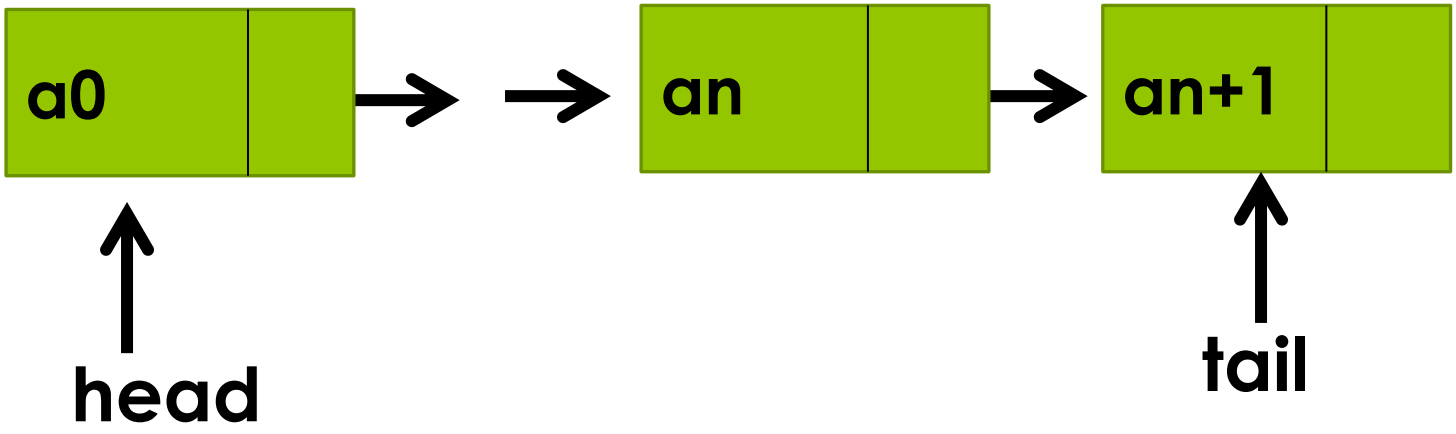


Опашка

- Свързано представяне

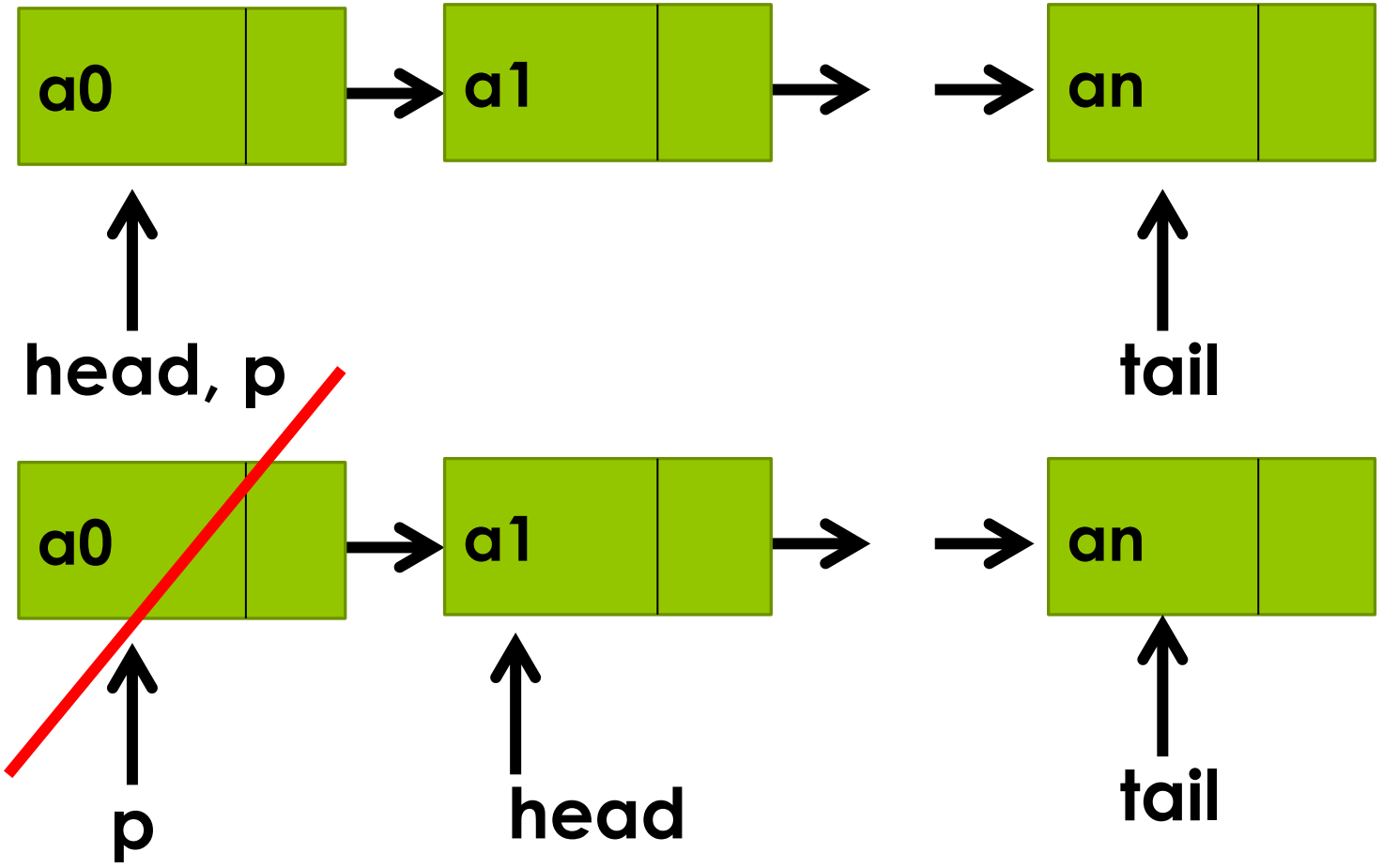


- push



Стек

- Свързано представяне
- pop



Опашка

```
template <typename T>
struct node {
  T data;
  node* next;
};
```

Опашка

```
template <typename T>
class LQueue{
  node<T> *head, *tail;

  void copy(LQueue<T> const& q) {
    for(node<T>* p = q.head;p != NULL;p = p->next)
      push(p->data);
  }

  void clean() {
    while (!empty())
      pop();
  }
public:
```

```
cout << “КРАЙ”;
```