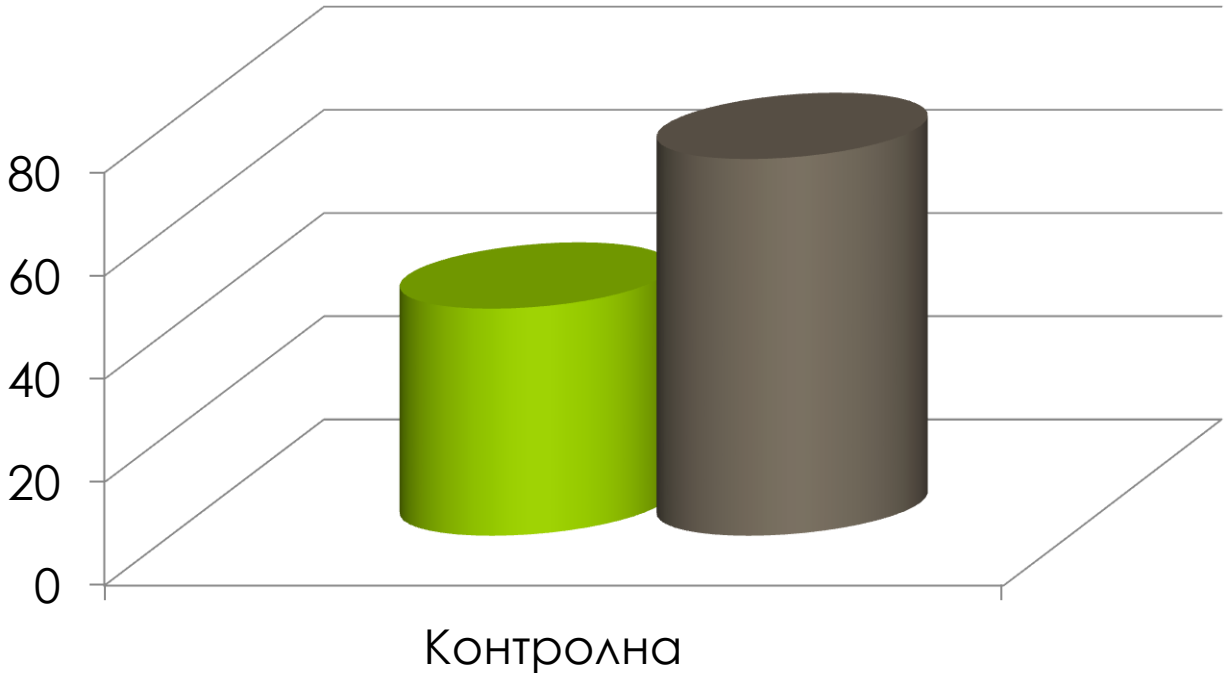
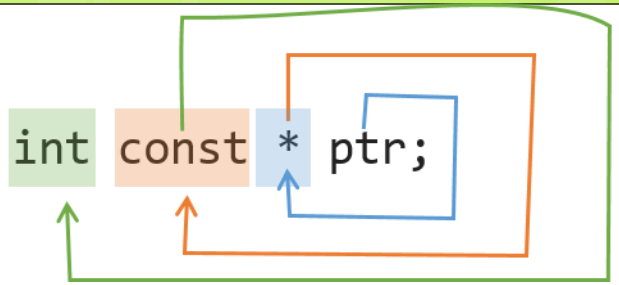
Опашка, Списък

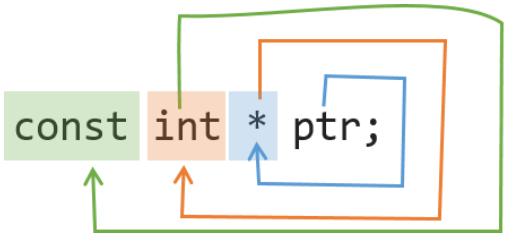
Изготвил:
гл.ас. д-р Нора Ангелова

Резултати

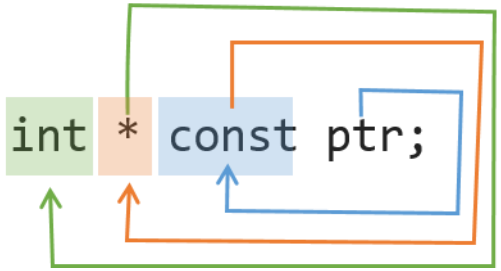




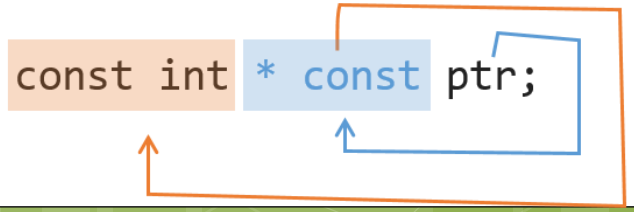
ptr is a pointer to const int



ptr is a pointer to int constant (i.e. const int)



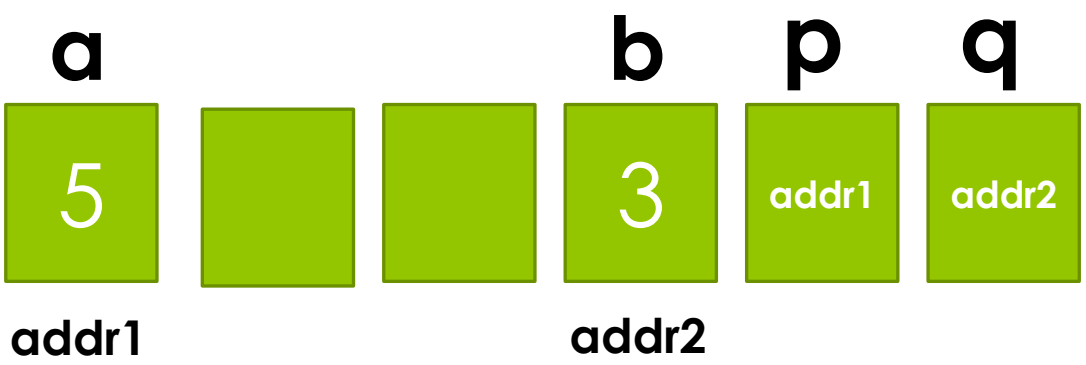
ptr is a const pointer to int



ptr is a constant pointer to const int

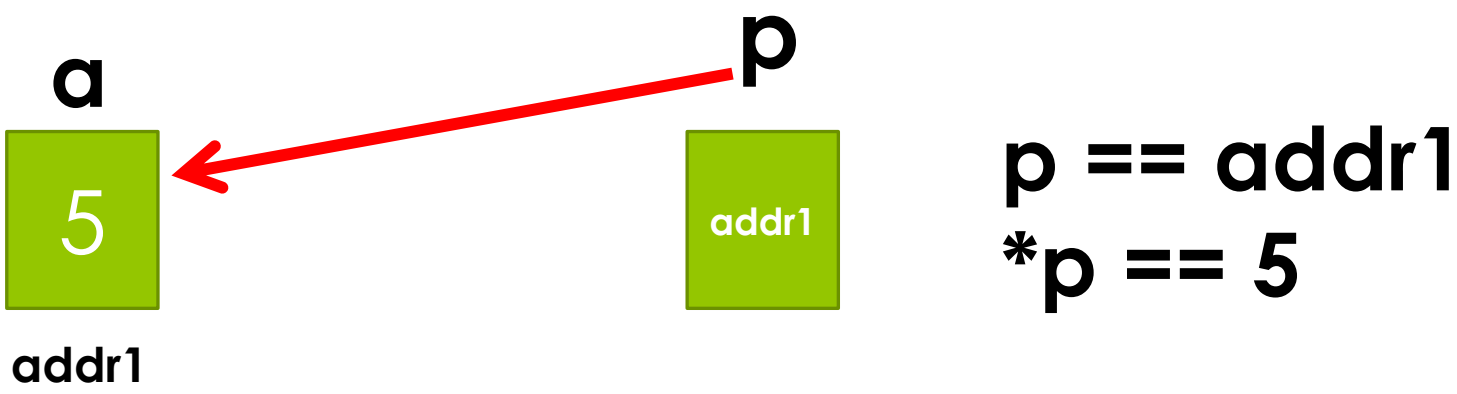
Указатели

```
int a = 5;
int b = 3
int *p = &a;
int *q = &q;
```



Указатели

```
int a = 5;
int *p = &a;
```



p == addr1

***p == 5**

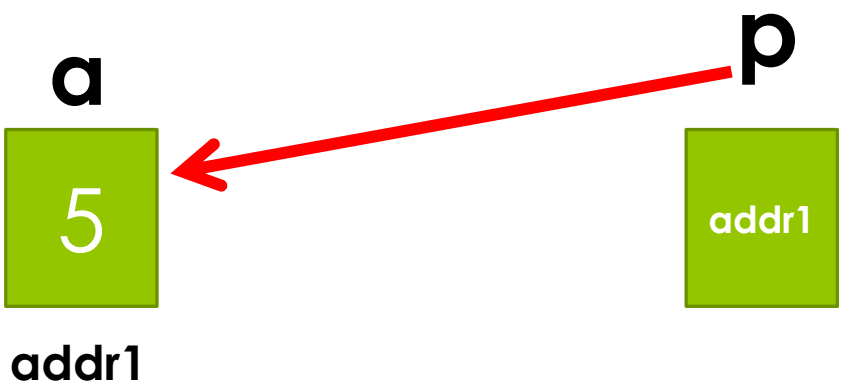
Указатели

- Изравняване на типове

```
int *p = &a;
```

a – int
p – int *
***p – int**

int* **&a**



Указатели

```
void test(int p) {
…
}
…

int a = 5;
test(a)
```

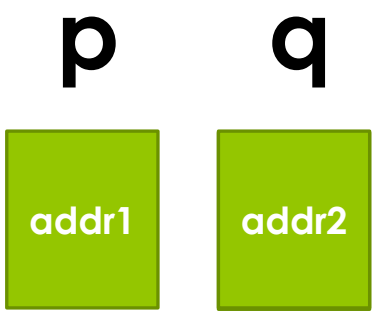
Указатели

```
void test(int* p) {
…
}
…

int a = 5;
test(&a)
```

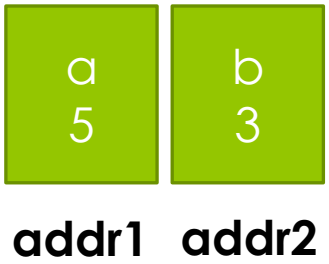
Задача

```
void swap(int* p, int* q) {
    int* temp = p;
    p = q;
    q = temp;
}

void swap2(int* p, int* q) {
    int temp = *p;
    *p = *q;
    *q = temp;
}

int main()
{
    int a = 5, b = 3;
    swap(&a, &b); // 5 3
    swap2(&a, &b); // 3 5
}
```

Разменя p и q

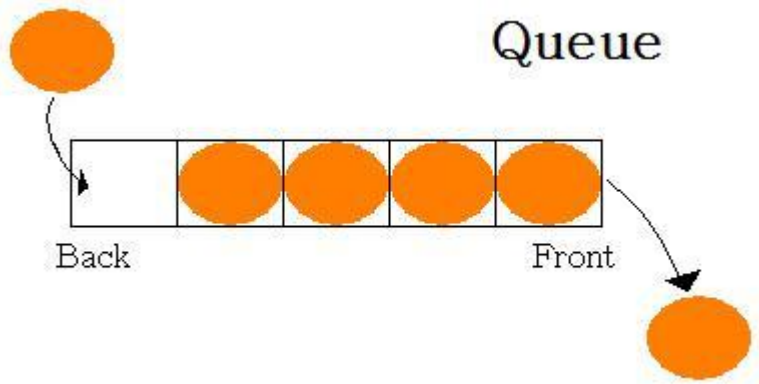


Разменя a и b



Опашка

- Хомогенна линейна структура от данни
- „първи влязъл - пръв излязъл“ (FIFO)
- Операцията включване е допустима за елементите от единия край на редицата – край на опашката
- Операцията изключване е допустима за елементите от другия край на редицата – начало на опашката
- Възможен е достъп само до елемента, намиращ се в началото на опашката



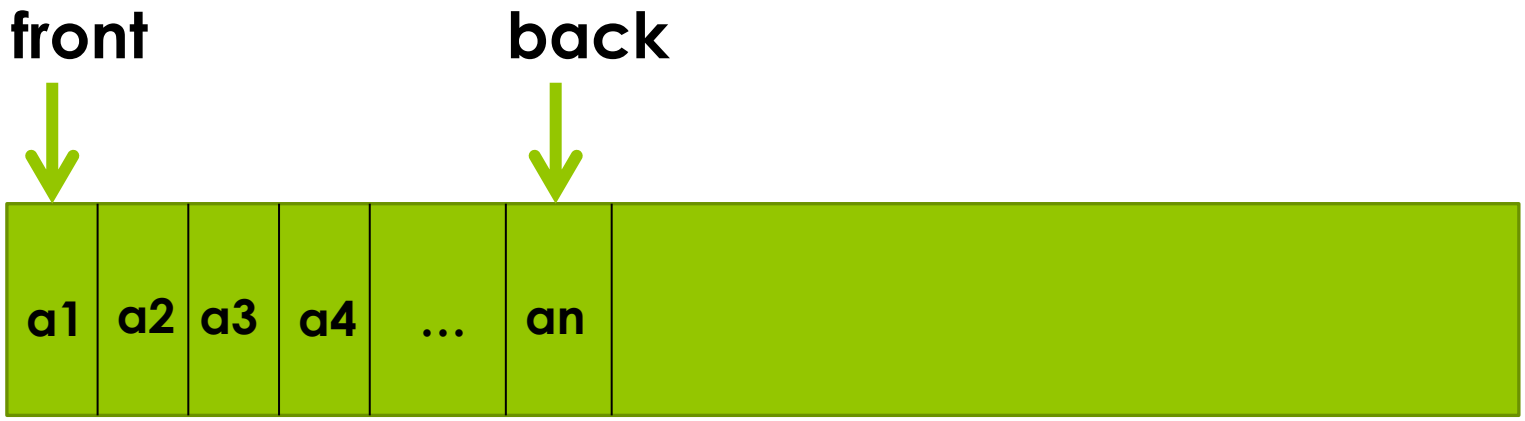
Опашка

Операции:

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

Последователно представяне

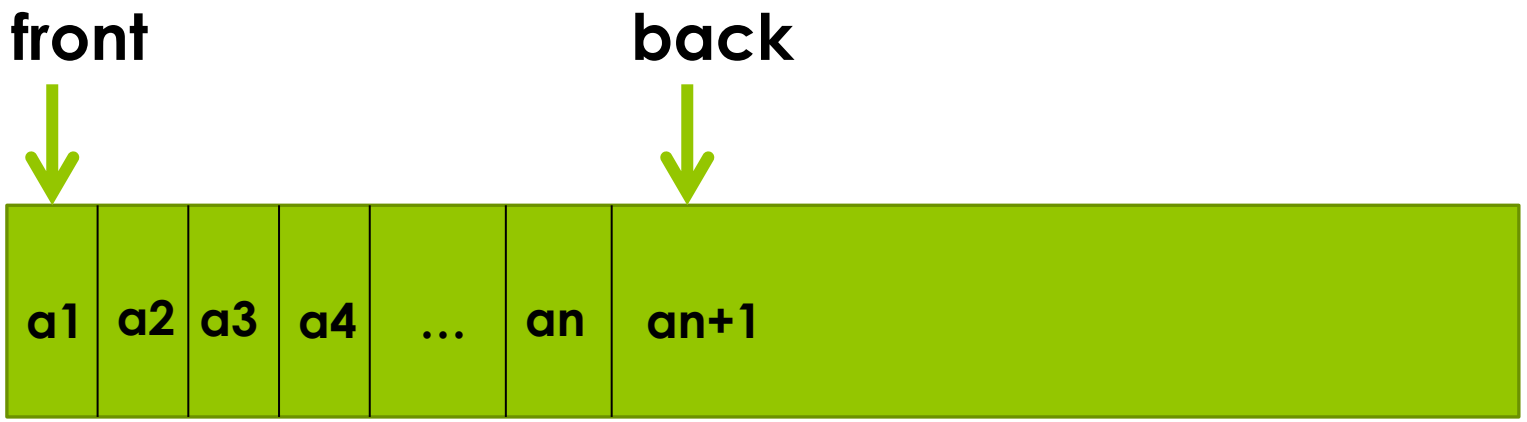
- Масив
- head/tail or front/rear or front/back



Последователно представяне

- Масив
- head/tail or front/rear or front/back

push(x) - включване на елемент

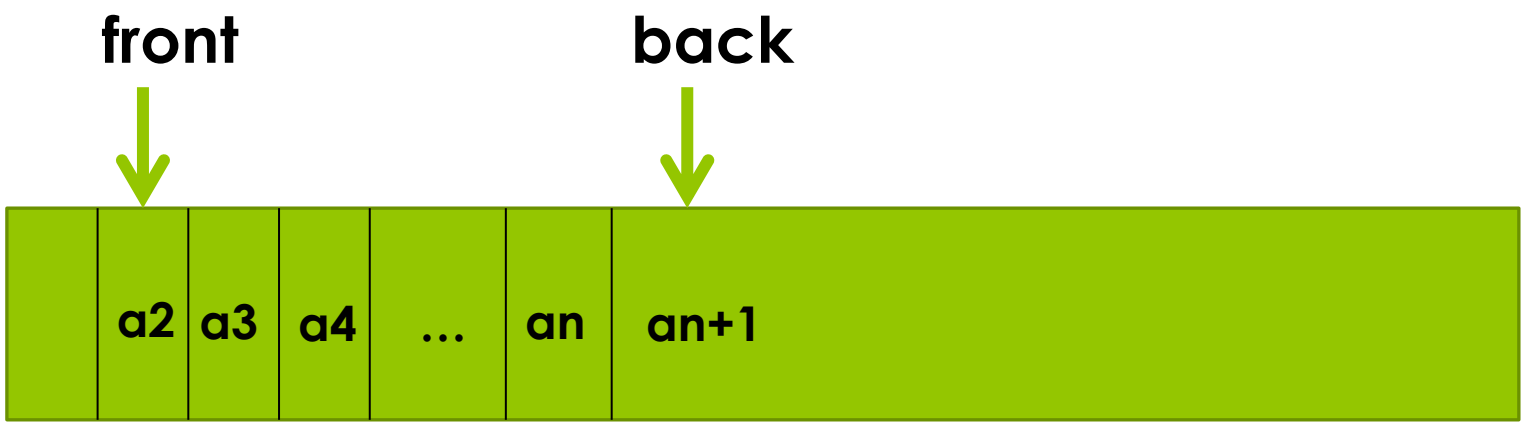


Последователно представяне

- Масив
- head/tail or front/rear or front/back

push() - включване на елемент

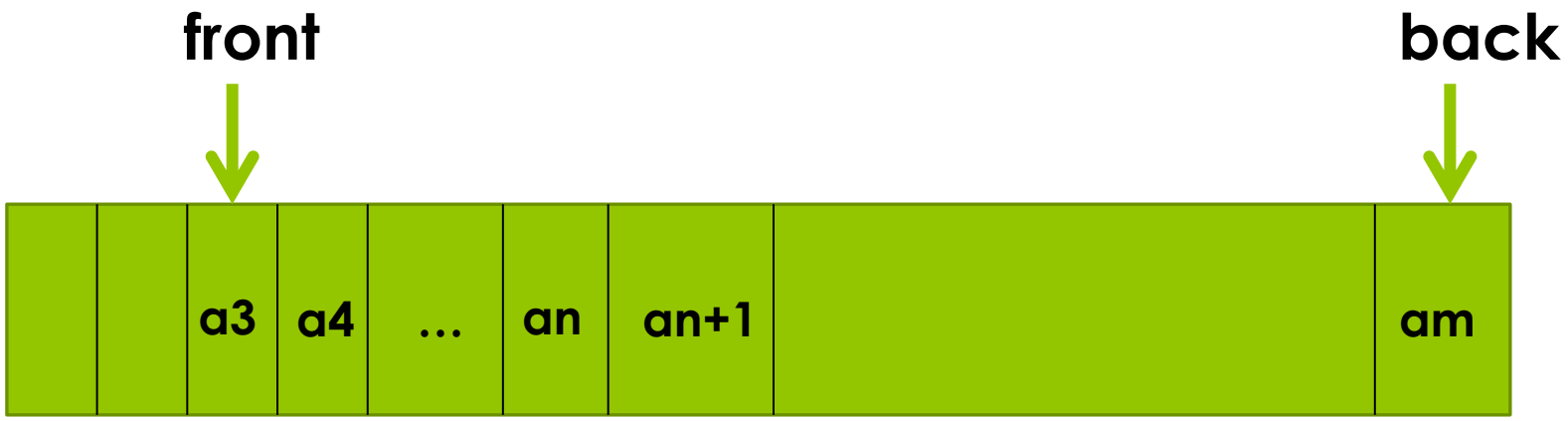
pop() - изключване на елемент



Последователно представяне

- Цикличност

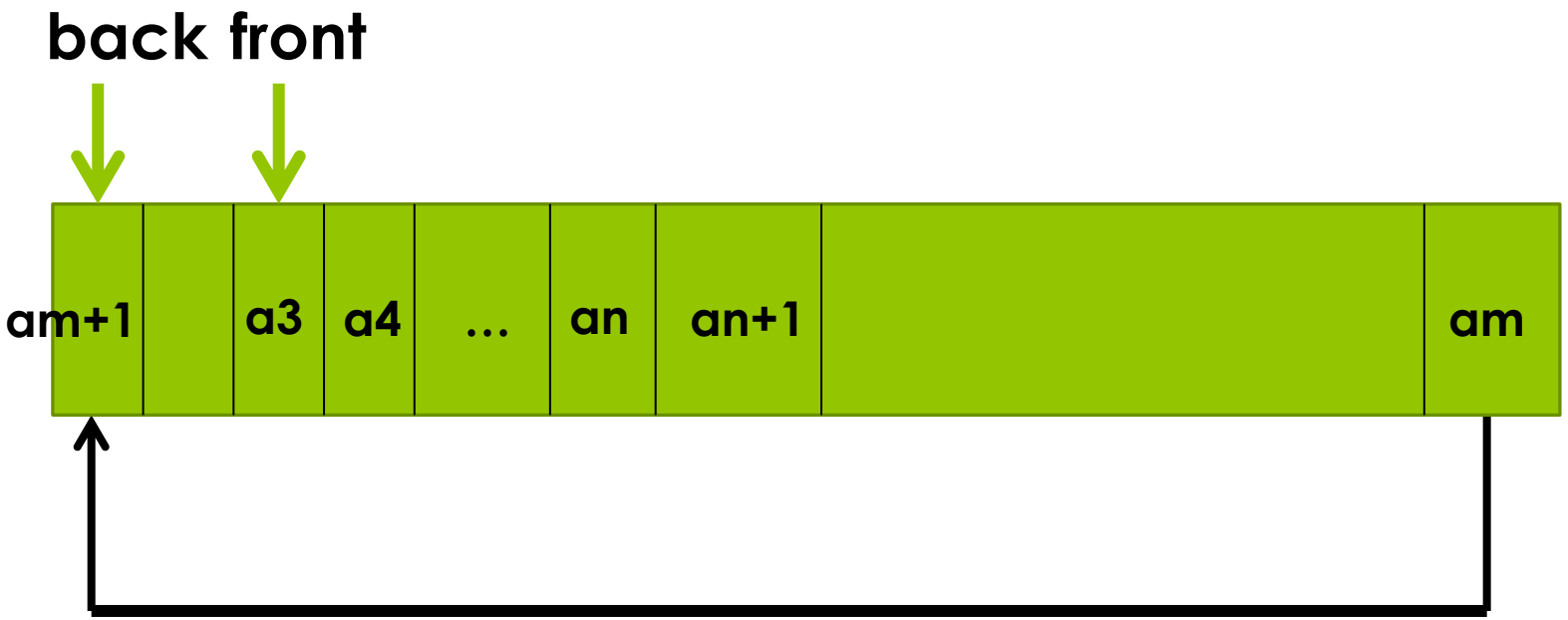
push() ?



Последователно представяне

- Цикличност

push() - включване на елемент



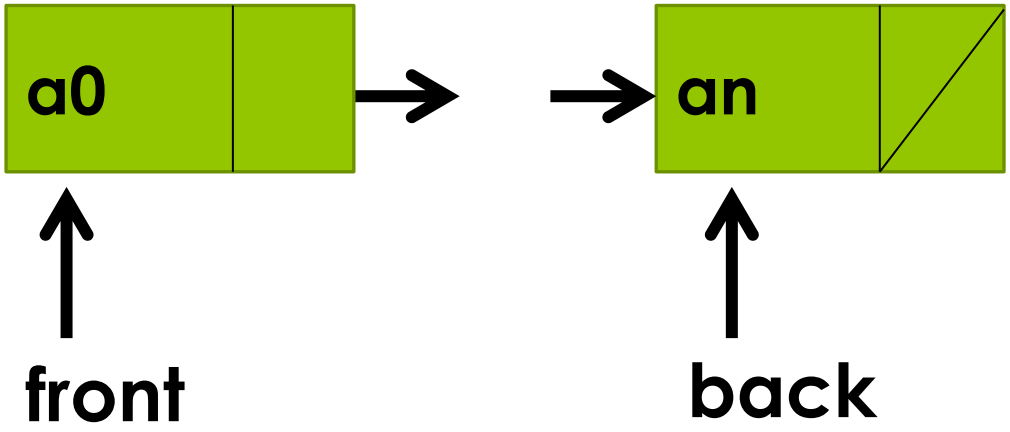
```
const int MAX_SIZE = 100;

template <typename T>
class StaticQueue {
  T a[MAX_SIZE];
  int front, back, n;

  bool full() const {
   return n == MAX_SIZE;
  }

public:
```

```
bool empty () const {
  return n == 0;
}

StaticQueue() : front(0), back(0), n(0) {}

T head() const {
  if (empty()) {
    cerr << "Опашката е празна!\n";
    return T();
  }
  return a[front];
}
// O(1)
```

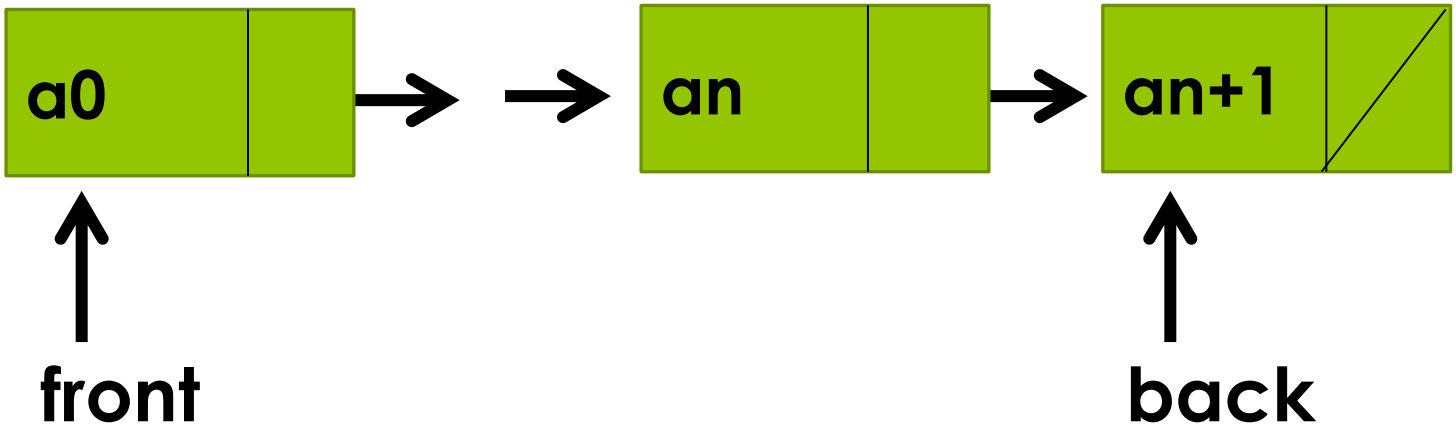
```
void push(T const& x) {
   if (full())
     cerr << "Опашката е пълна!\n";
   else {
     a[back] = x;
     n++;
     back++;
     back = back % MAX_SIZE;
   }
}
T pop() {
  if (empty()) {
    cerr << "Опашката е празна!\n";
    return T();
  }
  T x = a[front];
  n--;
  front++;
  front = front % MAX_SIZE;
  return x;
}
};
```

Опашка

- Свързано представяне

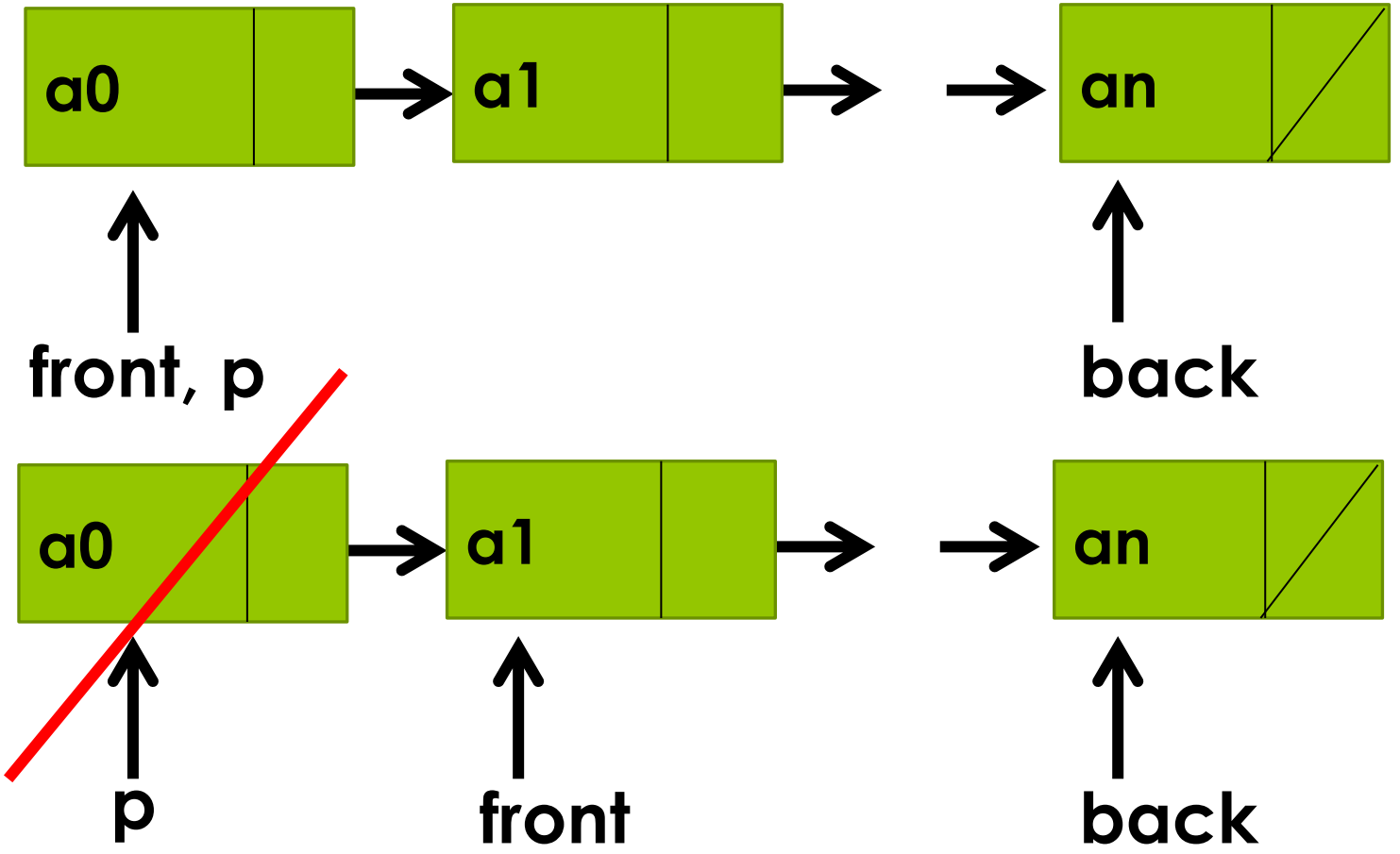


- push



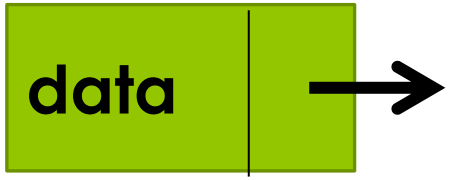
Стек

- Свързано представяне
- pop



Опашка

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



```
template <typename T>
class LQueue{
  node<T> *front, *back;

  void copy(LQueue<T> const& q) {
    back = NULL;
    if (!q.empty()) {
      node<T>* p = q.front;
      while(p) {
        push(p->data);
        p=p->next;
      }
    }
  }
  void clean() {
    while (!empty())
      pop();
  }
public:
```

```
LQueue() : front(NULL), back(NULL) {}

LQueue(LQueue const& q) : front(NULL), back(NULL) {
  copy(q);
}

LQueue& operator=(LQueue const& q) {
  if (this != &q) {
    clean();
    copy(q);
  }
  return *this;
}

~LQueue() {
  clean();
}
```

```
bool empty() const {
  return back == NULL;
}

void push(T const& x) {
  node<T>* p = new node<T>;
  p->data = x;
  p->next = NULL;
  if (!empty()) {
    back->next = p;
  } else
    front = p;
  back = p;
}
```

```
T pop() {
  if (empty()) {
    cerr << "Опит за изключване от празна опашка!\n";
    return T();
  }
  node<T>* p = front;
  T x = p->data;
  if (p == back) {
    front = NULL;
    back = NULL;
  }
  else
    front = p->next;
  delete p;
  return x;
}

T head() const {
  if (empty()) {
    cerr << "Опит за поглеждане в празна опашка!\n";
    return T();
  }
  return front->data;
}
};
```

Задача:

Да се напише функция print, която извежда елементите на опашката

```
// член функция на класа
template <typename T>
void LQueue<T>::print() {
  node<T>* p = front;
  while(p) {
    cout << p->data << “ ”;
    p=p->next;
  }
}
```

Задача:

Да се напише функция print, която извежда елементите на опашката

```
// външна функция
template <typename T>
void print(LQueue<T> q) {
  T x;
  while (!q.empty()) // ако 0 е знак за край while(x = q.pop())
    cout << q.pop() << " ";
}
```

Ще се разруши ли опашката?

Задача:

Да се напише програма, която преминава веднъж през елементите на масив от числа и извежда на екрана елементите на масива в следния ред:

- всички числа, които са по-малки от a
- всички числа в интервала [a, b]
- всички останали числа

Извеждането трябва да запазва първоначалния ред на числата.

Забележка: Без използване на допълнителни масиви

```
typedef LQueue<int> IntQueue;

void read(int* a, int n) {
  for(int i=0; i<n; i++) {
    cout << "a[" << i << "]=";
    cin >> a[i];
  }
}
int main()
{
  int n;
  do {
    cout << "n= ";
    cin >> n;
  } while(n < 1);

  int* testArray = new int[n];
  read(testArray, n);
```

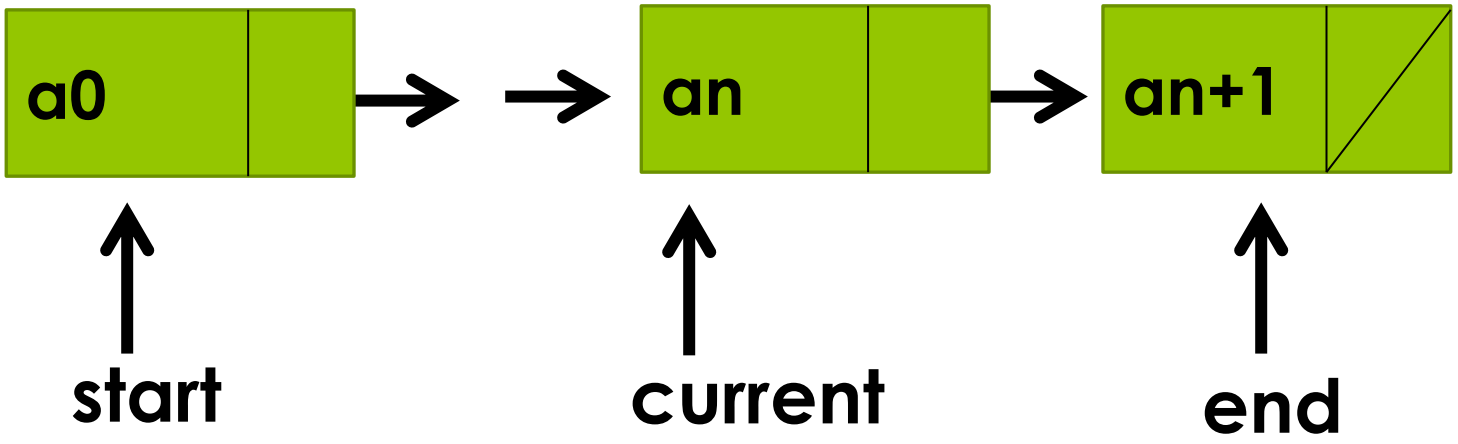
```
  int a, b;
  do {
    cout << "a < b =";
    cin >> a >> b;
  } while (a >= b);

  IntQueue q1, q2;
  for(int i=0; i<n; i++){
    if (testArray[i] < a)
      cout << testArray[i] << " ";
    else if (testArray[i] <= b)
        q1.push(testArray[i]);
    else q2.push(testArray[i]);
}
  cout << endl;
  q1.print();
  cout << endl;
  q2.print();

  delete [] testArray;
  system("pause");
  return 0;
}
```

Свързан списък

- Хомогенна линейна структура от данни
- Възможен е пряк достъп до елемента в единия край на редицата – „начало“ на списъка
- Възможен е последователен достъп до всеки от останалите елементи

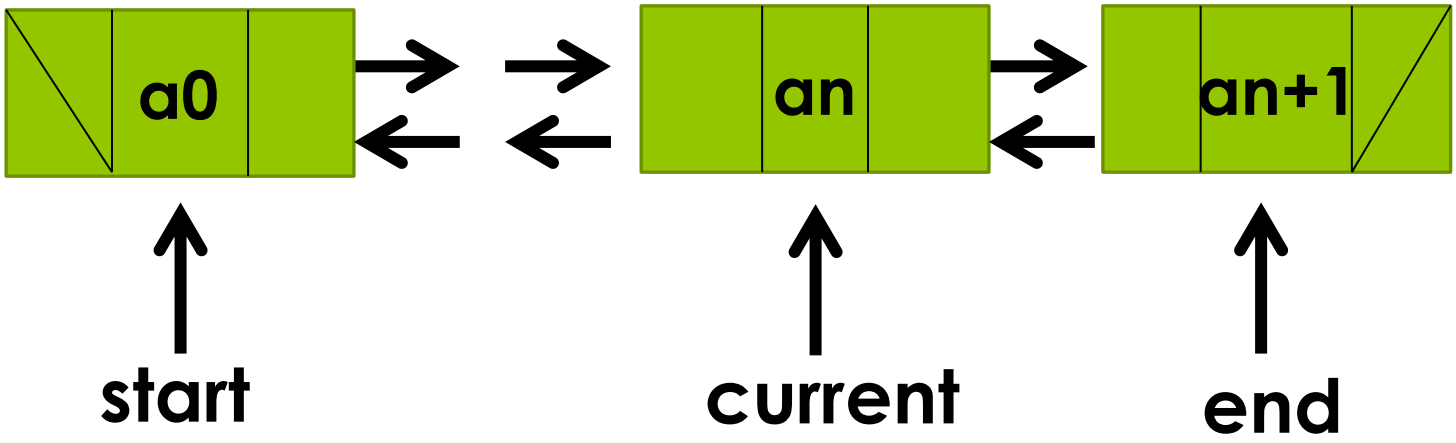
Свързан списък с една връзка

- Представяне, аналогично на свързаното представяне на стек и опашка
- Въвеждат се и указатели към края и към текущ елемент на списъка



Свързан списък с две връзки

- Въвеждат се тройни кутии, с едно информационно и две свързващи полета, съдържащи текущия елемент и адресите на предшестващия и следващия го елементи
- Въвеждат се и указатели към края и към текущ елемент на списъка



```
cout << “КРАЙ”;
```