Курс «Алгоритмы и алгоритмические языки»

Лекция 12

Очередь

- Очередь (*queue*) – это линейный список информации, работа с которой происходит по принципу *FIFO*.

  Для списка можно использовать статический массив: количество элементов массива (*MAX*) = наибольшей допустимой длине очереди.

- Работа с очередью осуществляется с помощью **двух функций**:

  `qstore()` – *поместить* элемент в конец очереди;

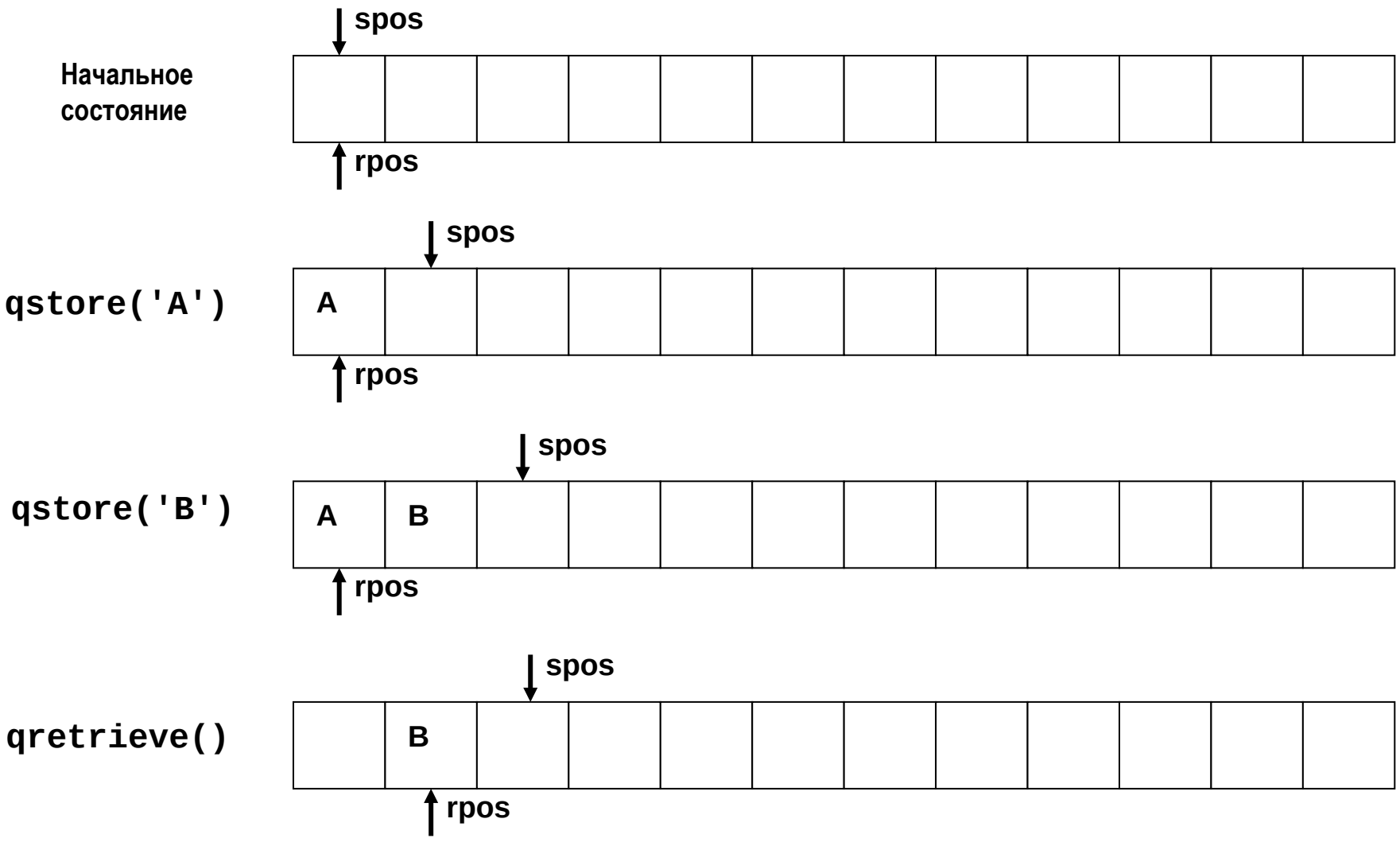
  `qretrieve()` – *удалить* элемент из начала очереди;

  и **двух глобальных переменных**:

  `spos` (индекс первого свободного элемента очереди: его значение <= **`MAX`**)

  `rpos` (индекс очередного элемента, подлежащего удалению: «кто первый?»)

Очередь


spos
Начальное состояние
rpos
spos
qstore('A')
A
rpos
spos
qstore('B')
A
B
rpos
spos
qretrieve()
B
rpos

Очередь

- Тексты функций **qstore()** и **qretrieve()**

```
#define MAX    67
int queue[MAX];
int spos = 0, rpos = 0;

void qstore (int q) {
  if (spos == MAX) {
    printf ("Очередь переполнена \n");
    return;
  }
  queue[spos++] = q;
}

int qretrieve (void) {
  if (rpos == spos) {
    printf ("Очередь пуста \n");
    return -1;
  }
  return queue[rpos++];
}
```

Улучшение – «зацикленная» очередь

```
#define MAX    67
int queue[MAX];
int spos = 0, rpos = 0;
```

```
void qstore (int q) {
  if (spos + 1 == rpos
      || (spos + 1 == MAX && !rpos) {
    printf ("Очередь переполнена \n");
    return;
  }
  queue[spos++] = q;
  if (spos == MAX)
    spos = 0;
}
```

Улучшение – «зацикленная» очередь

```
int qretrieve (void) {
  if (rpos == spos) {
    printf ("Очередь пуста \n");
    return -1;
  }
  if (rpos == MAX - 1) {
    rpos = 0;
    return queue[MAX – 1];
  }
  return queue[rpos++];
}
```

- Зацикленная очередь переполняется, когда **spos** находится непосредственно перед **rpos**, так как в этом случае запись приведет к **rpos == spos**, т.е. к пустой очереди.

Указатели на функцию

- Каждая функция располагается в памяти по определенному адресу. Адресом функции является ее точка входа (при вызове функции управление передается именно на эту точку).
- Присвоив значение адреса функции переменной типа указатель, получим указатель на функцию.
- Указатель функции можно использовать вместо ее имени при вызове этой функции. Указатель «лучше» имени тем, что его можно передавать другим функциям в качестве их аргумента.
- Имя функции **`f()`** без скобок и аргументов (**`f`**) по определению является указателем на функцию **`f()`** (аналогия с массивом).

Указатели на функцию

- ***Пример***. Сравнение двух строк символов, введенных пользователем (функция **check()**).

```
#include <stdio.h>
#include <string.h>
typedef int (*pfunc_t) (const char*, const char*);

static void check (char *a, char *b, pfunc_t pf) {
  printf ("Проверка на совпадение: ");
  if (! pf (a, b))
    printf ("равны\n");
  else
    printf ("не равны\n");
}

int main (void) {
  char s1[80], s2[80];

  printf ("Введите две строки \n");
  fgets (s1, 80, stdin); s1[strlen (s1) - 1] = 0;
  fgets (s2, 80, stdin); s2[strlen (s2) - 1] = 0;
  check (s1, s2, strcmp);
  return 0;
}
```

- Объявление **`int (*p)(const char *, const char *);`** сообщает компилятору, что **`p`** – указатель на функцию, имеющую два параметра типа **`const char *`** и возвращающую значение типа **`int`**.
- Скобки вокруг **`*p`** нужны, так как операция **`*`** имеет более низкий приоритет, чем **`()`**: Если написать **`int *p(...)`**, получится, что объявлен не указатель функции, а функция **`p`**, которая возвращает указатель на целое.
- **`(*cmp)(a, b)`** эквивалентно **`cmp(a, b)`.**
- У функции **`check`** три параметра: два указателя на тип **`char`** и указатель на функцию **`pf`**. Указатель **`pf`** и функция **`strcmp`** имеют одинаковый формат, что позволяет использовать имя функции в качестве аргумента, соответствующего параметру **`pf`**.
- В данном случае использование указателя функции позволяет не менять программу сравнения, и тем самым получается более общий алгоритм.

```
int compvalues (const char *a, const char *b) {
  return atoi(a) != atoi (b);
}
```