Курс «Алгоритмы и алгоритмические языки»
1 семестр 2018/2019

Лекция 18

Организация стека как библиотеки

◊ stack.h:

```
extern void push (char);
extern char pop (void);
extern int isempty (void);
```

◊ stack.c:

```
#include "stack.h"
struct stack {
  <...>
};
static struct stack stack
      = { <...> };

void push (char c ) { <...> }
char pop (void) { <...> }
int isempty (void) { <...> }
```

◊ main.c:

```
#include "stack.h"

int main (void) {
  <...push (c), pop (), ...>
}
```

```
$gcc main.c stack.c –o main
```

Организация стека как библиотеки - II

◊ stack.h:

```
struct stack;     // forward declaration
extern void push (struct stack *, char);
extern char pop (struct stack *);
extern int isempty (struct stack *);
extern struct stack* new_stack (void);
extern void free_stack (struct stack *);
```

◊ stack.c:

```
#include "stack.h"
struct stack {
  <...>
};
void push (struct stack *stack, char c ) {
  if (stack->sz == stack->sp + 1) <...>
}
<...>
```

Организация стека как библиотеки - III

◊ stack.c:

```
struct stack* new_stack (void) {
  struct stack *s = malloc (sizeof (struct stack));
  *s = (struct stack)
            { .sp = -1, .sz = 0, .stack = NULL };
  return s;
}
void free_stack (struct stack *s) {
  free (s->stack);
  free (s);
}
```

◊ main.c:

```
#include "stack.h"

int main (void) {
  struct stack *s = new_stack ();
  <...push (s, c), pop (s), ...>
  free_stack (s);          <...>
}
```

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;

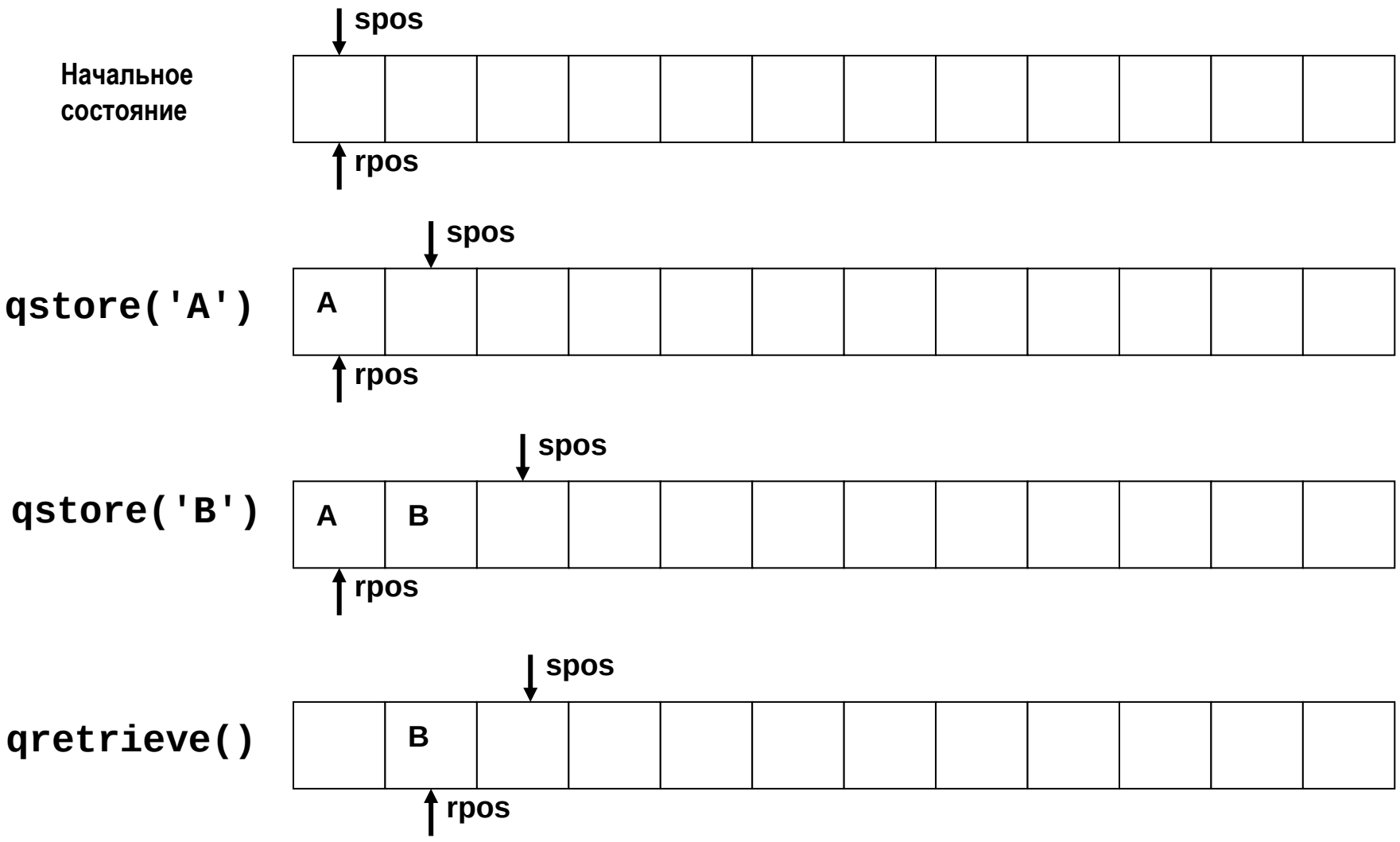
  `qretrieve()` – *удалить* элемент из начала очереди;

  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;

int qstore (int q) {
  if (spos == MAX) {
    /* Можно расширить очередь, см. реализацию стека */
    printf ("Очередь переполнена\n");
    return 0;
  }
  queue[spos++] = q;
  return 1;
}

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;
```

```
int qstore (int q) {
  if (spos + 1 == rpos
      || (spos + 1 == MAX && !rpos) {
    printf ("Очередь переполнена \n");
    // Дома. Реализуйте очередь на динамическом массиве.
    return 0;
  }
  queue[spos++] = q;
  if (spos == MAX)
    spos = 0;
  return 1;
}
```

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.

Списки

- ***Односвязный список*** – это динамическая структура данных, каждый элемент которой содержит ссылку на следующий элемент (либо **NULL**, если следующего элемента нет).
- Доступ к списку осуществляется с помощью указателя на его первый элемент.

```
struct list {
  struct data info;      /* Данные */
  struct list *next;     /* Ссылка на след. элемент */
};
```

- Выделение элемента

```
struct list *phead = NULL;
phead = (struct list *) malloc (sizeof (struct list));
```

Списки

- Добавление элемента в начало

```
struct list *phead = NULL;


struct list *add_element (struct list *phead, struct
                          data *elem) {
  struct list *new = malloc (sizeof (struct list));
  new->info = *elem;
  new->next = phead;
  return new;
}
```

Списки

- Добавление элемента в конец

```
struct list *phead = NULL;

struct list *add_element (struct list *phead, struct
                          data *elem) {
  if (! phead) {
    phead = malloc (sizeof (struct list));
    phead->info = *elem;
    phead->next = NULL;
    return phead;
  }
  struct list *ph = phead; // сохраним голову списка
  while (phead->next != NULL)
    phead = phead->next;
  phead->next = malloc (sizeof (struct list));
  phead->next->info = *elem;
  phead->next->next = NULL;
  return ph; // phead затерт, вернем сохраненный указатель
}
```

Списки

- Поиск элемента

```
struct list * phead;

int equals (struct data *, struct data *);
struct list * search (struct list *phead, struct data
                      *elem) {
  while (phead && ! equals (&phead->info, elem))
    phead = phead->next;
  return phead;
}
```

Списки

- Удаление элемента

```
struct list *remove (struct list *phead,
                     struct data *elem) {
  struct list *prev = NULL, *ph = phead;
  while (phead && ! equals (&phead->info, elem)) {
    prev = phead;
    phead = phead->next;
  }
  if (! phead)
    return ph;
  if (prev)
    prev->next = phead->next;
  else
    ph = phead->next;
  free (phead);
  return ph;
}
```

Списки

- Удаление элемента (двойной указатель)

```
void remove (struct list **pphead,
             struct data *elem) {
  struct list *prev = NULL, *phead = *pphead;
  while (phead && ! equals (&phead->info, elem)) {
    prev = phead;
    phead = phead->next;
  }
  if (! phead)
    return;
  if (prev)
    prev->next = phead->next;
  else
    *pphead = phead->next;
  free (phead);
}
```

Дома. Напишите добавление элемента с двойным указателем.

Топологическая сортировка узлов ациклического ориентированного графа

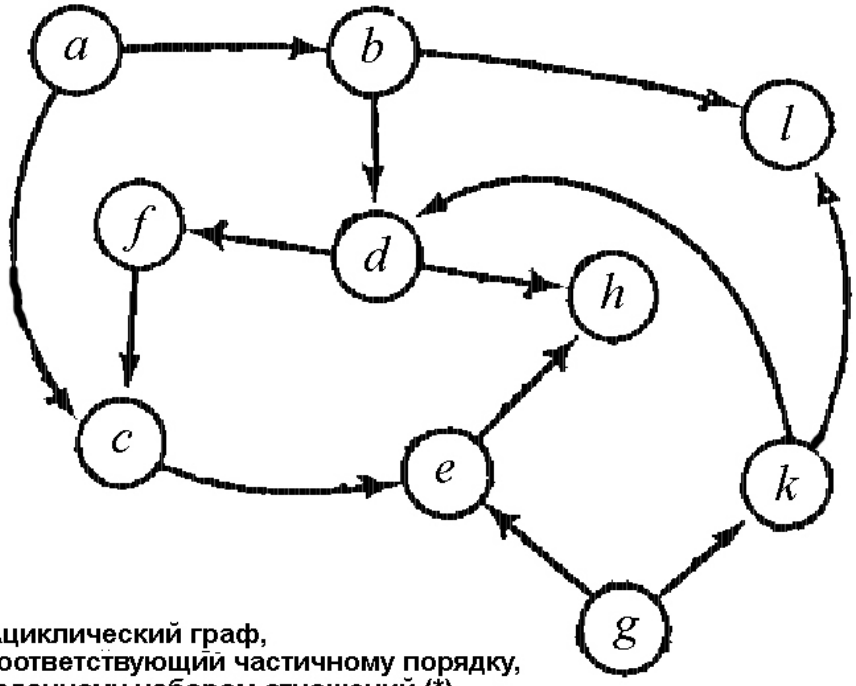
- Ациклический граф можно использовать для графического изображения *частично упорядоченного множества.*
  Цель топологической сортировки:
  преобразовать частичный порядок в линейный.
  Графически это означает, что
  все узлы графа нужно расположить на одной прямой таким образом, чтобы все дуги графа были направлены в одну сторону.

Топологическая сортировка узлов ациклического ориентированного графа

- ***Пример***. Частичный порядок (<) задается следующим набором отношений :

$$a < b,\ b < d,\ d < f,\ b < l,\ d < h,\ f < c,\ a < c, \qquad (*)$$
$$c < e,\ e < h,\ g < e,\ g < k,\ k < d,\ k < l$$

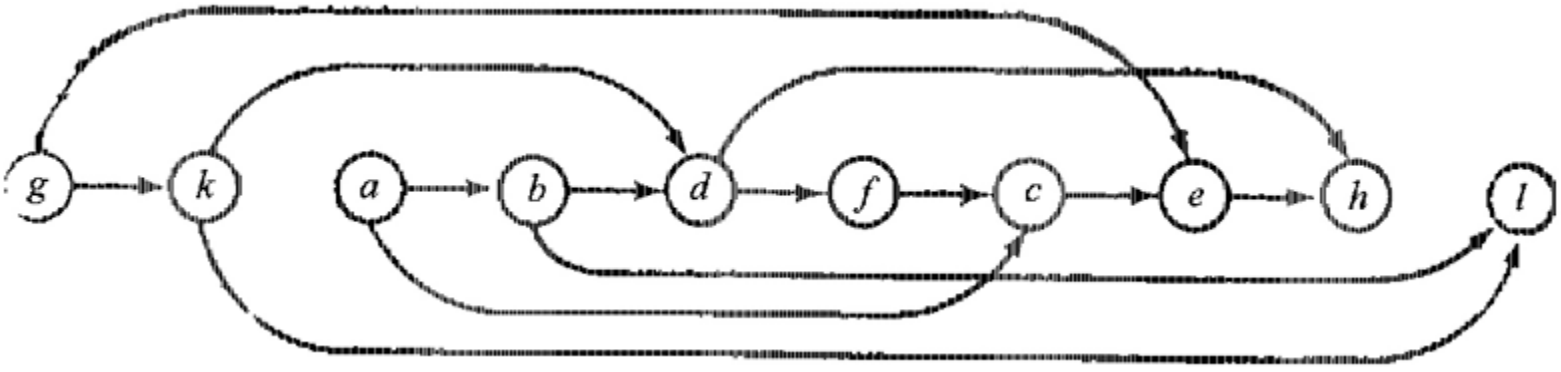
Набор отношений (*) можно представить в виде следующего ациклического графа (см. рисунок):



Ациклический граф, соответствующий частичному порядку, заданному набором отношений (*).

Топологическая сортировка узлов ациклического ориентированного графа

- Требуется привести рассматриваемый граф к линейному графу:



- На этом графе ключи расположены в следующем порядке:

g k a b d f c e h l

(поскольку топологическая сортировка неоднозначна, это один из возможных топологических порядков).

- Последовательная обработка полученного линейного списка узлов графа эквивалентна их обработке в порядке обхода графа.

Топологическая сортировка узлов ациклического ориентированного графа

- Поскольку рассматриваемый граф – ациклический, **существует** хотя бы один узел графа, у которого нет предшествующих узлов. Каждый такой узел будем называть *ведущим* узлом. **Шаг алгоритма**: Выберем один из ведущих узлов и поместим его в начало линейного списка отсортированных узлов, удалив его из исходного графа.
- Поскольку граф, у которого удалили один из ведущих узлов, останется ациклическим, в нем будет хотя бы один ведущий узел. Следовательно, можно повторить шаг алгоритма.
- Легко видеть, что каждый узел графа рано или поздно станет ведущим и попадет в формируемый линейный список, и алгоритм завершится.