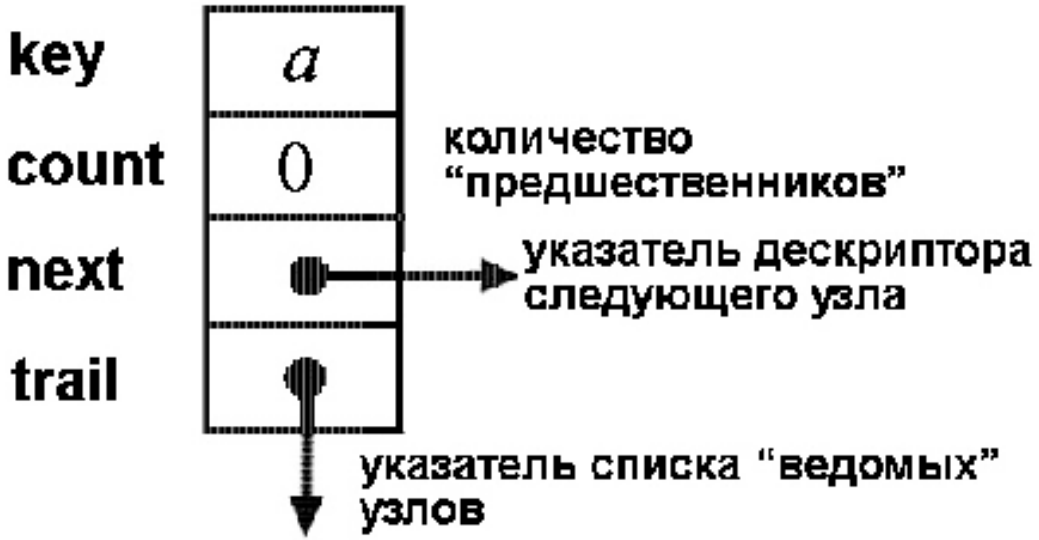
Курс «Алгоритмы и алгоритмические языки»
1 семестр 2018/2019

Лекция 19

Топологическая сортировка узлов ациклического ориентированного графа

- Структуры данных для представления узлов:
  - Каждый узел исходного графа представляется с помощью дескриптора узла, который имеет вид:

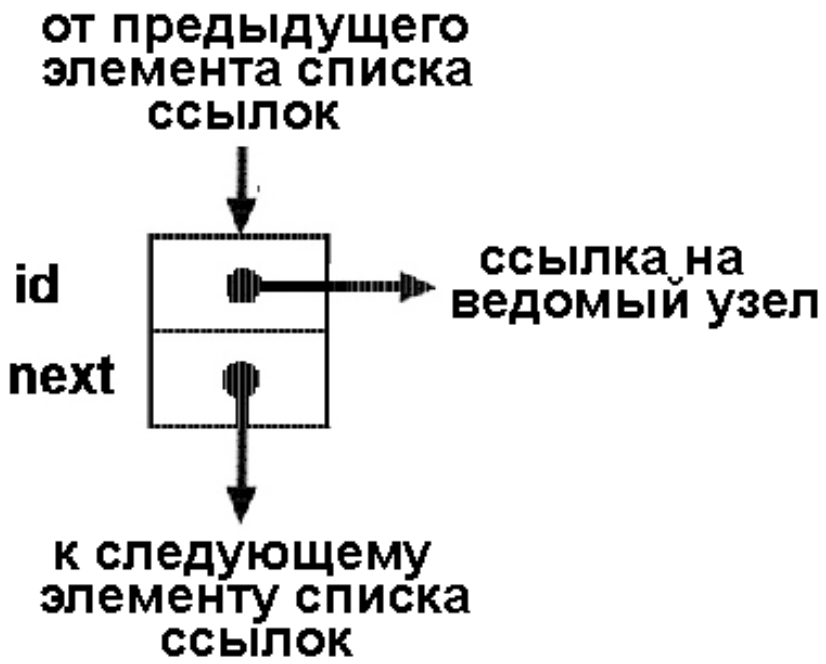


Дескриптор узла.

- *Ведомыми* для узла n будут узлы, для которых n является предшественником. Каждый узел графа (не только ведущий) может иметь один или несколько ведомых узлов.

Топологическая сортировка узлов ациклического ориентированного графа

- Структуры данных для представления узлов:
  - Дескриптор каждого узла содержит ссылки на ведомые узлы. Так как заранее неясно, сколько у узла будет ведомых узлов, эти ссылки помещаются в список. На рисунке представлен элемент списка ссылок.



Топологическая сортировка узлов ациклического ориентированного графа

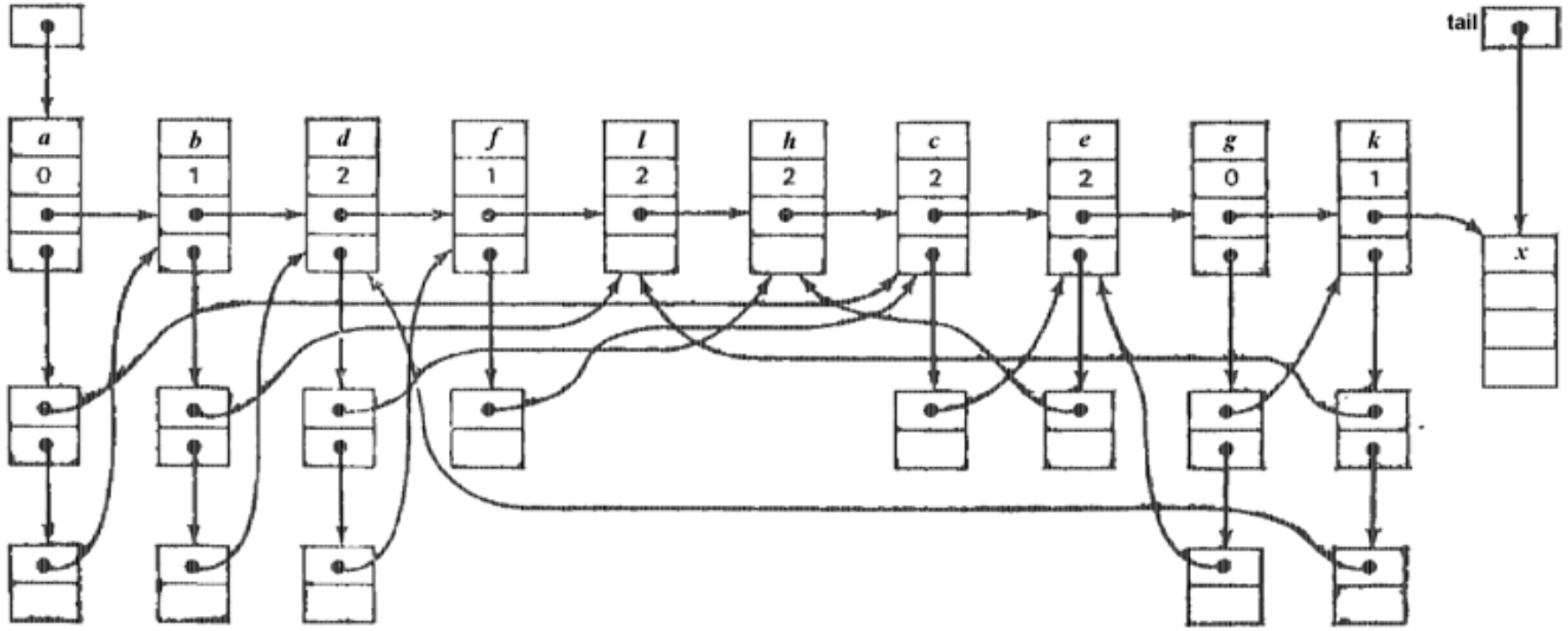
- ***1 фаза алгоритма: ввод исходного графа***

  На этой фазе вводятся пары ключей и из них формируется представление ациклического графа через дескрипторы узлов и списки ведомых узлов.

  - Исходные данные представлены в виде множества пар ключей (*), которые вводятся ***в произвольном порядке***.
  - После ввода очередной пары $x < y$ ключи x и y ищутся в списке «ведущих» и в случае отсутствия добавляются к нему.
  - В список ведомых узлов узла x добавляется ссылка на y, а счетчик предшественников y увеличивается на 1 (начальные значения всех счетчиков равны 0).

По окончании фазы ввода будет сформирована структура, показанная на следующем слайде (для множества пар ключей (*)).

Топологическая сортировка узлов ациклического ориентированного графа



Топологическая сортировка узлов ациклического ориентированного графа

- ***2 фаза алгоритма: сортировка***

  (1) В списке «ведущих» находим дескриптор узла ***z***, у которого значение поля **`count`** равно 0.

  (2) Включаем узел ***z*** в результирующую цепочку.

  (3) Если у узла ***z*** есть «ведомые» узлы (значение поля **`trail`** не **`NULL`**)

  (a) просматриваем очередной элемент списка «ведомых» узлов

  (b) корректируем поле **`count`** дескриптора соответствующего «ведомого» узла.

  (4) Переходим к шагу (1)

- Так как с каждой коррекцией поля **`count`** его значение уменьшается на 1, постепенно все узлы включаются в результирующую цепочку.

Описание алгоритма топологической сортировки на языке Си

```
#include <stdio.h>
#include <stdlib.h>
typedef struct ldr {     /*дескриптор ведущего узла*/
  char key;
  int count;
  struct ldr *next;
  struct trl *trail;
} leader;
typedef struct trl {     /*дескриптор ведомого узла*/
  struct ldr *id;
  struct trl *next;
} trailer;

leader *head, *tail;     /*два вспомогательных узла*/
int lnum;                /*счетчик ведущих узлов*/
```

Описание алгоритма топологической сортировки на языке Си

```
/* поиск по ключу w */

leader *find (char w) {
  leader *h = head;
  /* "барьер" на случай отсутствия w */
  tail->key = w;
  while (h->key != w)
    h = h->next;
  if (h == tail) {
    /* генерация нового ведущего узла */
    tail = malloc (sizeof (leader));
    /* старый tail становится новым элементом списка */
    lnum++;
    h->count = 0;
    h->trail = NULL;
    h->next = tail;
  }
  return h;
}
```

Описание алгоритма топологической сортировки на языке Си

```
void init_list() {
  /* инициализация списка «ведущих» */
  leader *p, *q;
  trailer *t;
  char x, y;

  head = (leader *) malloc (sizeof (leader));
  tail = head;
  lnum = 0;          /* начальная установка */
  while (1) {
    if (scanf ("%c %c", &x, &y) != 2)
      break;
    /* включение пары в список */
    p = find (x);
    q = find (y);
    /* коррекция списка */
    t = malloc (sizeof (trailer));
    t->id = q;
    t->next = p->trail;
    p->trail = t;
    q->count += 1;
  }
}
```

Описание алгоритма топологической сортировки на языке Си

```
/* Исходный список построен. Организация нового списка */
void sort_list() {
  leader *p, *q;
  trailer *t;
  /* В выходной список включаются все узлы старого
     с count == 0 */
  p = head;
  head = NULL; /* голова выходного списка */
  while (p != tail) {
    q = p;
    p = q->next;
    if (q->count == 0) {
      /* включение q в выходной список */
      q->next = head;
      head = q;
    }
  }
<...>
```

Описание алгоритма топологической сортировки на языке Си

```
/* Фаза сортировки и вывода результатов из нового списка */

<...>
  q = head; /* есть ведущий узел -> head != NULL */
  while (q != NULL) {
    printf ("%c\n", q->key);
    lnum--;
    t = q->trail;
    q = q->next;
    while (t != NULL) {
      p = t->id;
      p->count -= 1;
      if (p->count == 0) {
        p->next = q;  // достаточно для
        q = p;        // правильной сортировки
      }
      t = t->next;
    }
  }
  /* lnum == 0 */
}
```

Описание алгоритма топологической сортировки на языке Си

```
int main() {
  init_list ();
  sort_list ();
  return 0;
}
```

Дома. Что поменяется, если узлы идентифицируются не одним символом, а именем (строкой)? Сделайте нужные изменения в коде. Добавьте определение циклов в исходных данных.

Сортировка

- ***Постановка задачи***

  Сортировка – это упорядочение наборов однотипных данных, для которых определено отношение линейного порядка (например, <) по возрастанию или по убыванию.
  Здесь будут рассматриваться целочисленные данные и отношение порядка "<".

- В стандартную библиотеку **`stdlib`** входит функция **`qsort`**:

  ```
  void qsort (void *buf, size_t num, size_t size,
  int(*compare)(const void *, const void *));
  ```

  Функция **`qsort`** сортирует (по возрастанию) массив с указателем **`buf`**, используя алгоритм быстрой сортировки *Ч.Э.Р. Хоара*, который считается одним из лучших алгоритмов сортировки общего назначения.

  Параметр **`num`** задает количество элементов массива **`buf`**, параметр **`size`** – размер (в байтах) элемента массива **`buf`**.

  Параметр **`int(*compare)(const void *, const void *)`**задает правило сравнения элементов массива **`num.`**

Сортировка

- Функция, указатель на которую передается в **`qsort`** в качестве аргумента, соответствующего параметру **`int(*compare)(const void *, const void *)`**, должна иметь описание:

  `int` ***`имя функции`*** **`(const void *`*`arg1`**`, const void *`*`arg2`**`)`**

  и возвращать:

  - целое < 0, если ***`arg1`*** < ***`arg2`***,
  - целое = 0, если ***`arg1`*** = ***`arg2`***
  - целое > 0, если ***`arg1`*** > ***`arg2`***

Сортировка

- ***Простейший алгоритм сортировки***: сведение сортировки к задаче нахождения максимального (минимального) из *n* чисел.

  Нахождение максимума *n* чисел (*n* сравнений):

  Числа содержатся в массиве **`int a[n];`**

```
max = a[0];
for (i = 1; i < n; i++)
  if (a[i] > max)
    max = a[i];
```

- ***Алгоритм сортировки***: находим максимальное из *n* чисел, получаем последний элемент отсортированного массива (*n* сравнений);
  находим максимальное из *n* – 1 оставшихся чисел, получаем предпоследний элемент отсортированного массива (еще *n* – 1 сравнений); и так далее.
- ***Общее количество сравнений***: $1 + 2 + \ldots + n\text{-}1 + n = n(n - 1)/2$.

  Сложность алгоритма $O(n^2)$.

Сортировка

- ***Классификация алгоритмов сортировки***

  Различают *внешнюю* и *внутреннюю* сортировку.

  Рассматривается только *внутренняя сортировка*: сортируемый массив находится в основной памяти компьютера. *Внешняя сортировка* применяется к записям на внешних файлах.

  3 общих метода внутренней сортировки:

  (1) *сортировка обменами*: рассматриваются соседние элементы сортируемого массива и при необходимости меняются местами;

  (2) *сортировка выборкой*: идея описана на предыдущем слайде

  (3) *сортировка вставками*: сначала сортируются два элемента массива, потом выбирается третий элемент и вставляется в нужную позицию относительно первых двух и т.д.

Сортировка

- ***Сортировка обменами (пузырьком)***

  Общее количество сравнений (действий): $n(n - 1)/2$, так как внешний цикл выполняется $(n - 1)$ раз, а внутренний – в среднем $n/2$ раза.

```
void bubble_sort (int *a, int n) {
  int i, j, tmp;
  for (j = 1; j < n; ++j)
    for (i = n - 1; i >= j; --i) {
      if (a[i - 1] > a[i]) {
        tmp = a[i - 1];
        a[i - 1] = a[i];
        a[i] = tmp;
      }
    }
}
```

Сортировка

- ***Сортировка вставками***

  Количество сравнений зависит от степени перемешанности массива **a**. Если массив **a** уже отсортирован, количество сравнений равно *n* - 1. Если массив **a** отсортирован в обратном порядке (наихудший случай), количество сравнений имеет порядок n^2.

```
void insert_sort (int *a, int n) {
  int i, j, tmp;

  for (j = 1; j < n; ++j) {
    tmp = a[j];
    for (i = j - 1; i >= 0 && tmp < a[i]; i--)
      a[i + 1] = a[i];
    a[i + 1] = tmp;
  }
}
```

Сортировка

- ***Оценка сложности алгоритмов сортировки***
- Скорость сортировки определяется количеством сравнений и количеством обменов (обмены занимают больше времени). Эти показатели интересны для худшего и лучшего случаев, а также интересно их среднее значение.
- Кроме скорости оценивается «естественность» алгоритма сортировки:
  естественным считается алгоритм, который на уже отсортированном массиве работает минимальное время, а на не отсортированном работает тем дольше, чем больше степень его неупорядоченности.
- Важным показателем является и объем дополнительной памяти для хранения промежуточных данных алгоритма.
  Для рекурсивных алгоритмов расход памяти связан с необходимостью сохранять в автоматической памяти (стеке) локальные переменные и параметры.

Сортировка

- ***Оценка сложности алгоритмов сортировки***.
- ***Теорема***. *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq \mathrm{O}(n \cdot \log_2(n))$$

- Доказательство.

(1) *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq \log_2(n!).$$

(a) Алгоритм S можно представить в виде двоичного дерева сравнений.
Так как любая перестановка индексов рассматриваемого массива может быть ответом в алгоритме, она должна быть приписана хотя бы одному листу дерева сравнений.
Таким образом, дерево сравнений будет иметь не менее n! листьев.

Сортировка

- ***Оценка сложности алгоритмов сортировки***.
- ***Теорема***. *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq \mathrm{O}(n \cdot \log_2(n))$$

- Доказательство.

(1) *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq \log_2(n!). \qquad (*)$$

(б) Для высоты h_m двоичного дерева с m листьями имеет место оценка:

$$h_m \geq \log_2 m.$$

Любое двоичное дерево высоты h можно достроить до полного двоичного дерева высоты h, а у полного двоичного дерева высоты h 2^h листьев.

Применив полученную оценку к дереву сравнений, получим оценку (*)

Сортировка

- ***Оценка сложности алгоритмов сортировки***.
- ***Теорема***. *Для любого алгоритма S внутренней сортировки сравнением массива из n элементов количество сравнений*

$$C_S \geq O(n \cdot \log_2(n))$$

- Доказательство.

(2) К $\log_2(n!)$ применим формулу Стирлинга

$$n! = \sqrt{2\pi n} \cdot n^n e^{-n} e^{\vartheta(n)} \qquad (**)$$

$$|\vartheta(n)| \leq \frac{1}{12n}$$

Логарифмируя (**), получаем

$$\log(n!) = \frac{1}{2}\log(2\pi n) + n \cdot \log(n) - n + \vartheta(n)$$

$$\log(n!) \geq O(n \cdot \log(n))$$