

СТРУКТУРЫ ДАННЫХ

Списки. Понятие абстрактной структуры данных. Хеш-таблицы и
поисковые деревья

К. Владимиров, Intel, 2020
mail-to: konstantin.vladimirov@gmail.com

Идея односвязного списка

- Идея односвязного списка довольно проста: каждый узел содержит указатель на следующий

```
struct node_t {  
    struct node_t *next;  
    int contents;  
};
```

- Узлы такого рода динамических структур данных обычно выделяются в куче

```
struct node_t *top = calloc(1, sizeof(struct node_t));
```

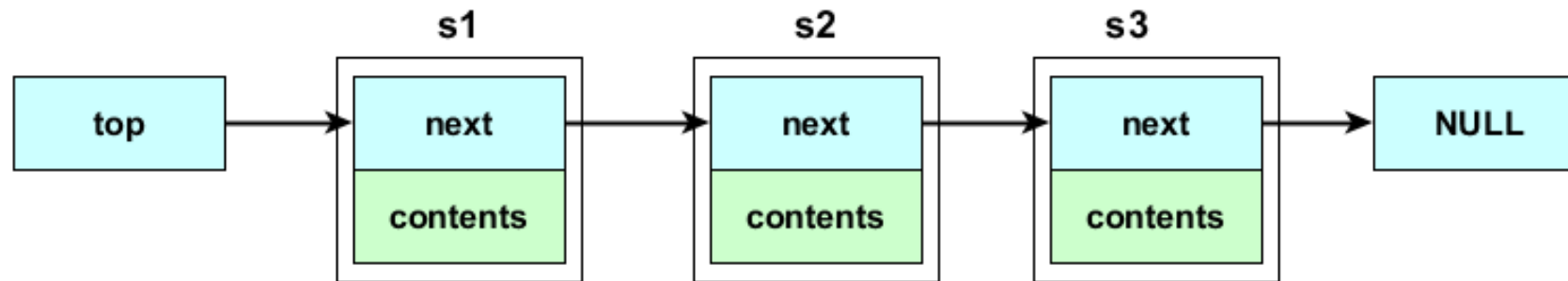
```
top->next = calloc(1, sizeof(struct node_t));
```

- Это бывает удобно изобразить картинкой

Идея односвязного списка

- Идея односвязного списка довольно проста: каждый узел содержит указатель на следующий

```
struct node_t {struct node_t *next; int contents; };
```

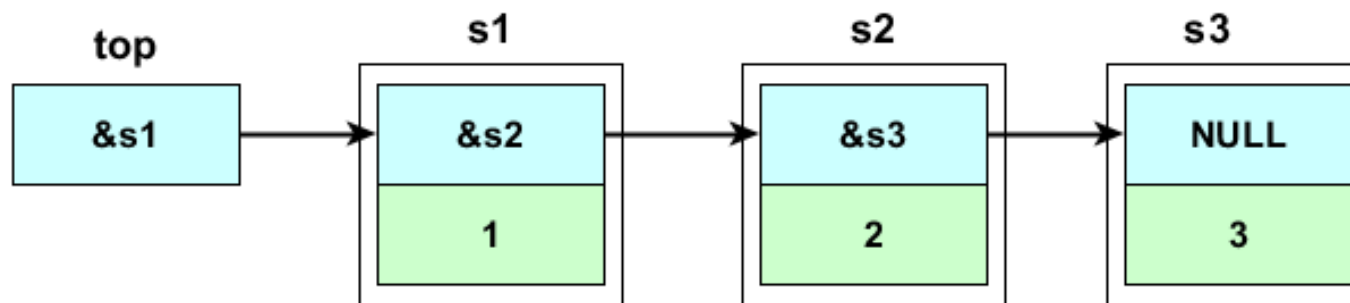


- Картинку можно несколько конкретизировать, учтя значения полей, выставленные на предыдущем слайде

Идея односвязного списка

- Идея односвязного списка довольно проста: каждый узел содержит указатель на следующий

```
struct node_t {struct node_t *next; int contents; };
```



- Динамические структуры данных крайне полезны
- По традиции мы начнём их изучение со списков (хотя они как раз сами по себе не слишком полезны, зато очень просты)

Problem AL: работа со списками

- Ваша задача: написать две функции
- Построить односвязный список из файлового ввода, такой, что все чётные числа идут в начале, а все нечётные в конце

```
struct node_t *read_list(FILE *inp);
```

- Файл представляет собой просто целые числа разделённые через пробел

```
1 65 78 2 34
```

- Удалить односвязный список, получив указатель на первый элемент

```
void delete_list(struct node_t *top);
```

Снова динамическое программирование

- Любая часть оптимальной траектории является оптимальной траекторией
- Пусть дана сумма 11. Как оптимально разменять её номиналами 4, 3 и 1?

1	2	3	4	5	6	7	8	9	10	11
1	2	1	1	2	2	2	2	3	3	?



- Здесь
$$N(k) = \min \begin{cases} N(k - m_1) + 1 \\ N(k - m_2) + 1 \\ N(k - m_3) + 1 \end{cases}$$

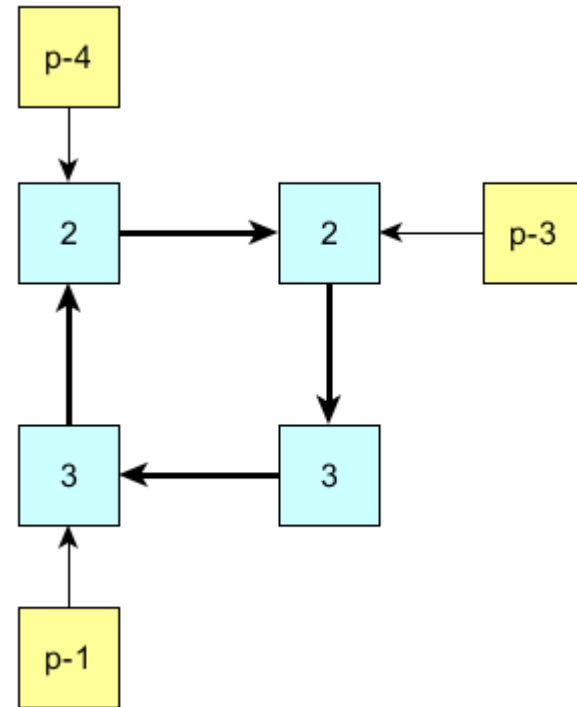
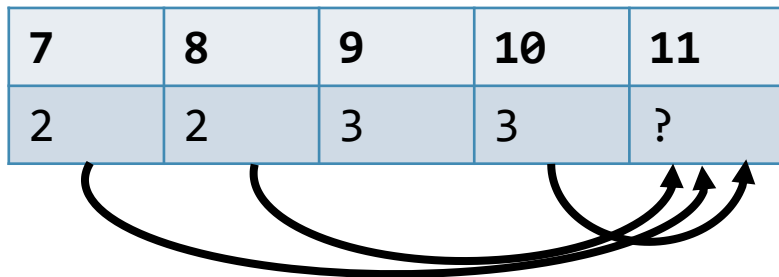
- В общем случае нужно разменять сумму M номиналами m_1, m_2, \dots, m_n
- Последовательное заполнение таблицы требует $O(M * n)$ времени

Обсуждение

- Мы хотели бы разменять сумму M номиналами t_1, t_2, \dots, t_n
- Сколько для этого нам потребуется памяти?

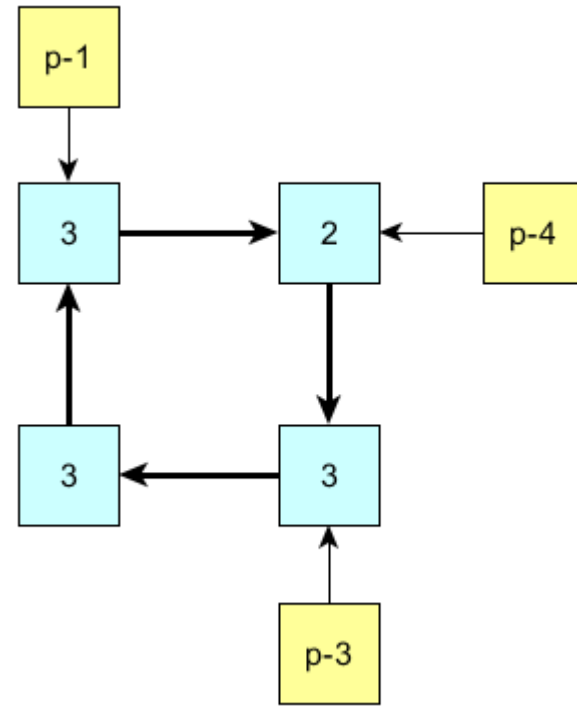
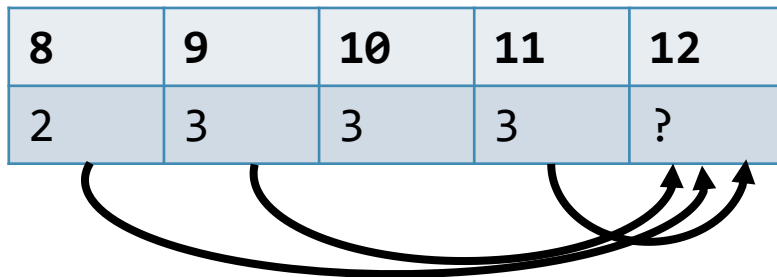
Обсуждение

- Мы хотели бы разменять сумму M номиналами m_1, m_2, \dots, m_n
- Сколько для этого нам потребуется памяти?
- Неправильный ответ: $O(M)$
- Правильный ответ: $O(m_n)$
- Основная идея: циклический список



Обсуждение

- Мы хотели бы разменять сумму M номиналами m_1, m_2, \dots, m_n
- Сколько для этого нам потребуется памяти?
- Неправильный ответ: $O(M)$
- Правильный ответ: $O(m_n)$
- Основная идея: циклический список



Интерфейс циклического списка

- Основная структура ничем не отличается от обычного односвязного списка

```
struct list_t {  
    struct list_t *next;  
    int data;  
};
```

- Основные операции (файл cl-impl)

```
// добавить элемент после указанного  
struct list_t *cl_push(struct list_t *pre, int d);
```

```
// удалить элемент после указанного  
struct list_t *cl_pop(struct list_t *pre);
```

Problem CL – циклический размен

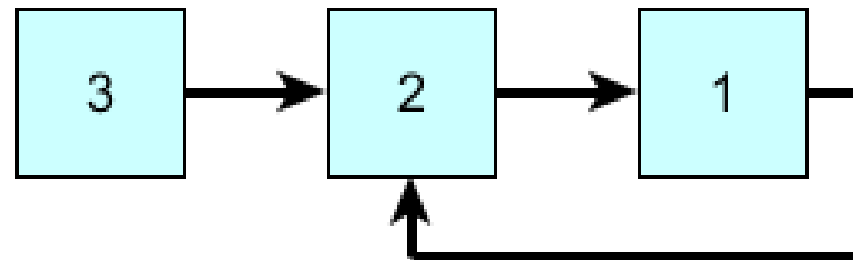
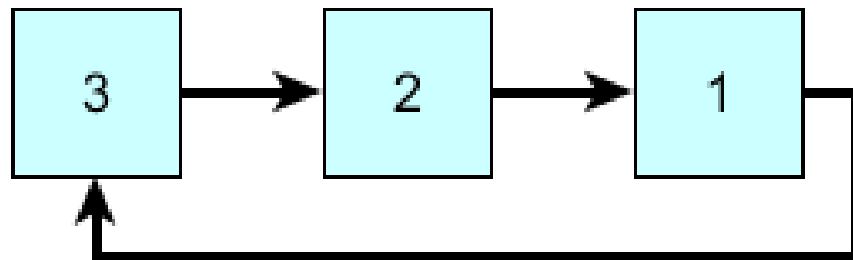
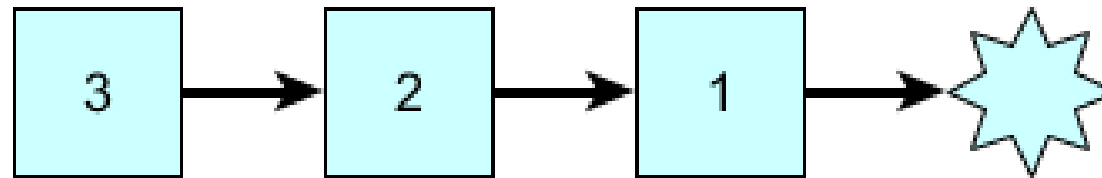
- Входной файл содержит сумму для размена, количество номиналов и все номиналы размена
- Ваша задача считав его со стандартного ввода вывести на стандартный вывод правильное количество использованных монет
- Пример. Входные данные:
11 3 1 3 4
- Выход:
3
- Сумма для размена может быть очень большой, используйте циклический буфер

Обсуждение

- Тот факт, что циклический список мало чем отличается от обычного должен, в принципе, вас насторожить

Обсуждение

- Тот факт, что циклический список мало чем отличается от обычного должен, в принципе, вас насторожить



Задача: есть ли петля?

- Вам на вход приходит связный список

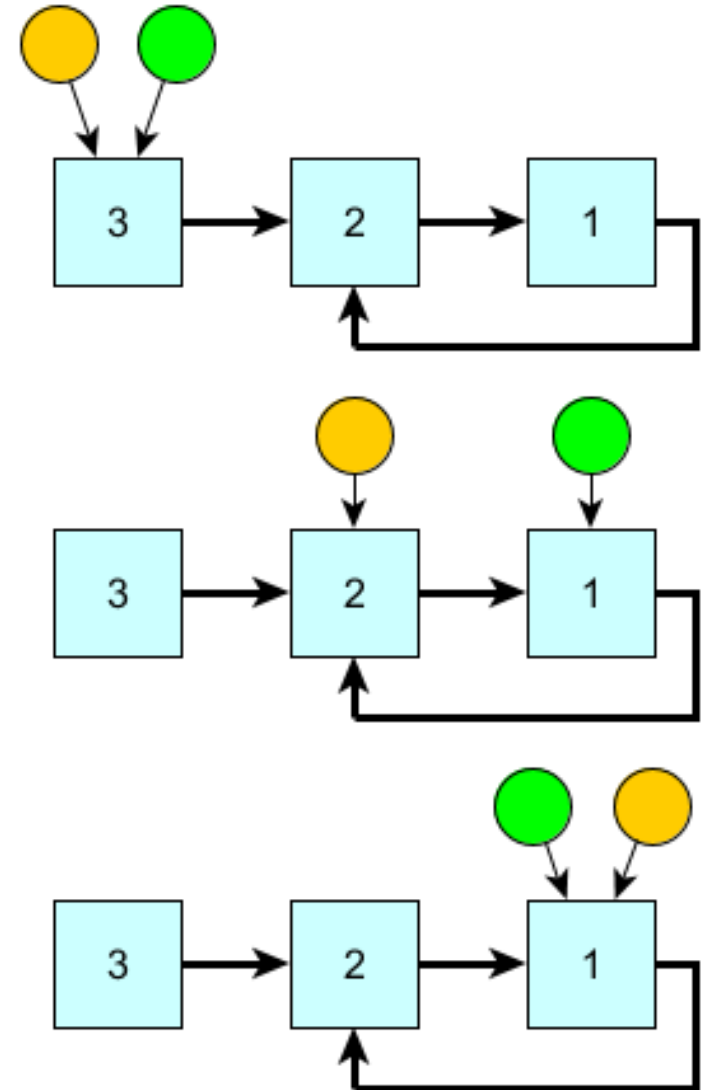
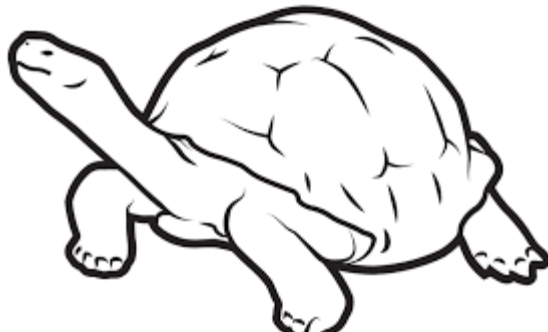
```
struct list_t {  
    struct list_t *next;  
    int data;  
};
```

```
int somefunc(struct list_t *top);
```

- Возможно в нём есть петля. Возможно он кончается на нулевой указатель. Как это определить?

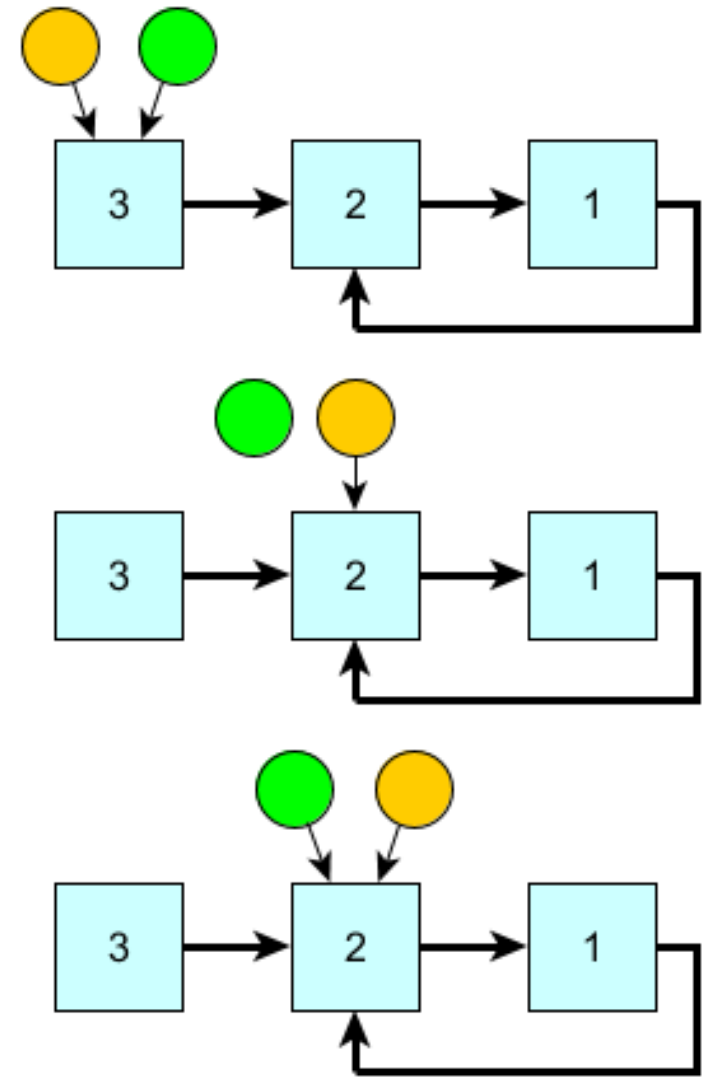
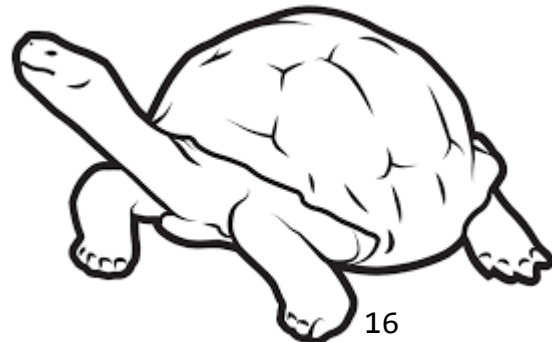
Алгоритм Флойда

- Начинают два указателя: заяц и черепаха
- Заяц за каждый ход продвигается вперёд на два элемента, а черепаха на один
- Если они встретились, значит петля есть



Алгоритм Брента

- Начинают два указателя: заяц и черепаха
- Заяц за каждый ход продвигается вперёд на единицу и несёт с собой телепорт
- Как только он прошёл очередную степень двойки, он телепортирует туда черепаху
- Если они встретились, значит петля есть



Problem HL – петля в связном списке

- Вам на вход приходит какой-то список, заданный всё той же структурой

```
struct list_t {  
    struct list_t *next;  
    int data;  
};
```

- Вы должны написать функцию, которая определяет есть ли в нём петля

```
int list_is_a_loop(struct list_t *top) {  
    // TODO: ваш код здесь  
}
```

- Вернуть 0 (нет) или 1 (есть). Используйте любой из двух описанных методов

Problem HL2 – длина петли

- Вам на вход приходит какой-то список, заданный всё той же структурой

```
struct list_t {  
    struct list_t *next;  
    int data;  
};
```

- Вы должны написать функцию, которая определяет **длину петли** если она есть и возвращает 0, если её нет

```
int loop_len(struct list_t *top);
```

- Попробуйте адаптировать один из ранее приведённых алгоритмов для этого
- Разумное решение не будет использовать много дополнительной памяти

Problem RG: период генератора

- Вам дан генератор $x_{j+1} = f(x_j)$, при этом вы не знаете функцию f
- Генерируемые числа выглядят случайными. Вам нужно написать функцию, которая ищет длину цикла в генераторе, начиная с $x_0 = 0$

```
typedef int (*generator_t)(int);  
  
unsigned cycle_len(generator_t gen) {  
    // TODO: ваш код здесь  
}
```

- Например при $f(x) = (x + 2) \% 5$, $x_0 = 0$ генерируется последовательность 0, 2, 4, 1, 3, 0, 2, ... и длина цикла равна 5
- Заметьте, вовсе не факт, что x_0 встретится ещё раз (генератор может быть смещённым). Но для любого $j > i$, $x_j = x_i$ означает цикл

Обсуждение

- Представьте, что у вас стоит задача **развернуть** односвязный список, не содержащий петли
- Как вы представляете себе её решение?

Алгоритм RFL – рекурсивный разворот

- Основная идея в том, что $\text{reverse}(x:xs) = \text{reverse}(xs):x$

```
struct list_t * reverse(struct list_t *top) {
    struct list_t *xs;
    if (NULL == top) return NULL;
    if (NULL == top->next) return top;
    xs = reverse(top->next);
    top->next->next = top; // единственное тонкое место
    top->next = NULL;
    return xs;
}
```

- Алгоритм сравнительно прост и красив. Увы достаточно длинный список переполнит стек

Problem LR – развернуть итеративно

- Это ещё одна обычная проблема вида рекурсия-в-итерацию
- Ваша задача написать функцию разворота односвязного списка, которая будет работать за $O(1)$ по памяти

```
struct list_t *reverse(struct list_t *top);
```

- Обратите внимание, что алгоритм RFL тратит $O(N)$ памяти на стек вызовов

Обсуждение

- Что если в вашем входном односвязном списке всё же появится петля?
- Попробуйте прогнать вашу процедуру переворота на зацикленном списке и посмотрите результат
- Интересный вопрос: что же делать?

Обсуждение

- Что если в вашем входном односвязном списке всё же появится петля?
- Попробуйте прогнать вашу процедуру переворота на зацикленном списке и посмотрите результат
- Интересный вопрос: что же делать?
- Плохой вариант: при каждом вызове переворота искать петлю. Это практически удваивает время на переворот
- Хороший вариант несколько менее очевиден. И он называется **инкапсуляция**

Список как структура данных

- Представьте, что список вынесен в модуль как в `encap-sll.h / encap-sll.c`
- Внешний интерфейс выглядит примерно так

```
struct list_t; // объявление списка, детали спрятаны
```

```
struct list_t *list_create(int d);  
struct list_t *list_push(struct list_t * pre, int d);  
struct list_t *list_pop(struct list_t * pre);
```

- Такой список в принципе не может случайно зациклиться, так как вся работа с ним осуществляется только функциями его **открытого интерфейса**
- И каждая из этих функций **поддерживает инварианты своего типа данных**

Структуры данных

- Список не единственная структура данных. Самые известные это:
 - Динамические массивы
 - Линейные и циклические списки
 - Хеш-таблицы
 - Поисковые деревья
- Обсуждение: чем **отличаются** друг от друга структуры данных?

Структуры данных

- Список не единственная структура данных. Самые известные это:
 - Динамические массивы
 - Линейные и циклические списки
 - Хеш-таблицы
 - Поисковые деревья
- Обсуждение: чем **отличаются** друг от друга структуры данных?
- Неправильный ответ: реализацией. Реализация у них спрятана и считается не слишком важной. Операции с каждым типом происходят только через его интерфейс
- Правильный ответ: алгоритмической сложностью функций открытого интерфейса

Пример: телефонная книга

- У вас есть список друзей и список их номеров в международном формате (до 15 цифр)

Alice	44-7911-975-72-83
Bob	44-7911-486-92-83
Camilla	8-800-555-31-35
Daniel	8-800-765-91-35

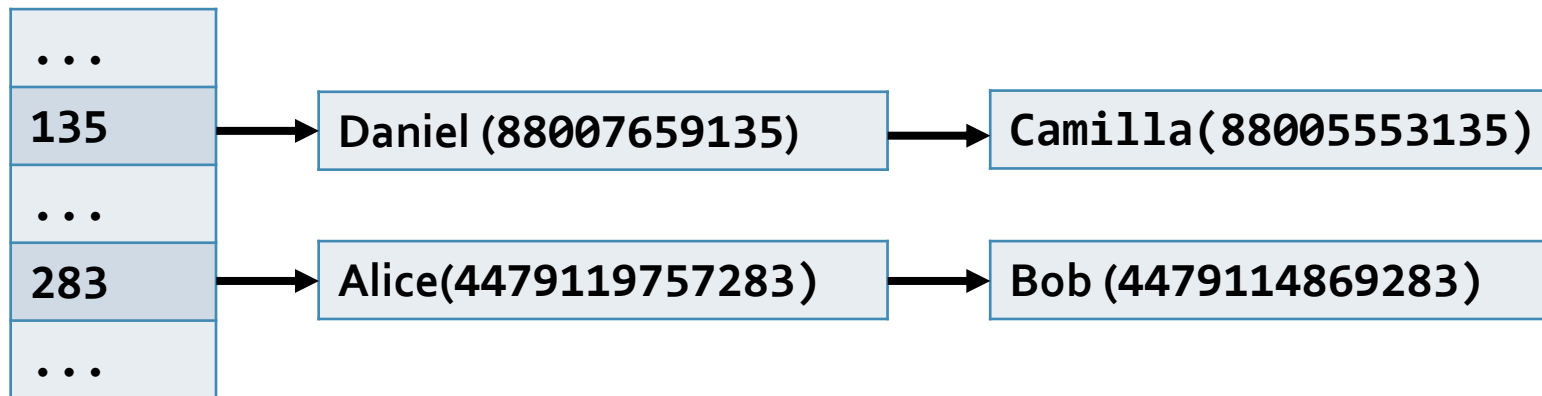
- Вам нужно выбрать структуру данных, которая позволяла бы:
 - Быстро добавить человека и его номер
 - Найти имя человека по номеру телефона
- Просто хранить массив строк, индексированный номерами телефонов затратно

Обсуждение

- Можно выбрать сортированный по номеру массив
 - Преимущества: быстрый поиск $O(\lg N)$
 - Недостатки: медленная вставка $O(N)$
- Можно выбрать список
 - Преимущества: быстрая вставка $O(1)$
 - Недостатки: медленный поиск $O(N)$
- Конечно лучше всего было бы сделать прямую адресацию по номерам телефонов, тогда и поиск и вставка были бы $O(1)$
- Можем ли мы **отобразить** все номера телефонов на разумный диапазон, скажем $0 \div 999$?

Хеширование

- Самый простой способ это взять три последние цифры номера
- Запишем это как функцию $h(n) = n \% t$ (где t это **мощность** отображения, в данном случае $t = 1000$). Мощность определяет количество **бакетов**
- Тогда имеем массив из 1000 списков



- Уже сейчас видно, что если повезёт и **коллизий** не будет, то поиск $O(1)$ и вставка $O(1)$. Увы сейчас **хеш-функция** $h(n)$ довольно плоха

Универсальные семейства функций

- Случайно выбранная функция из такого семейства гарантированно будет в среднем не хуже, чем любая другая для этого класса
- Везде далее m это мощность хеша, p это простое число, большее m
- Для целых чисел это семейство

$$h_{a,b}(n) = ((ax + b) \% p) \% m, \text{ при этом } a \neq 0$$

Для строк это семейство

$$h_r(c_1 \dots c_l) = h_{int} \left(\left(\sum_{i=1}^l c_i r^{l-i} \right) \% p \right) \text{ где } h_{int} \text{ это произвольно выбранная } h_{a,b}$$

- Иногда параметры действительно выбирают случайно и меняют с каждым запуском программы

Реализация хеш-функций

- Если что-то считается миллионы раз за выполнение программы, его реализации лучше быть как можно более оптимальной
- Пусть w это количество бит в машинном слове (обычно 16 или 32) и мощность $m = 2^M, M < w$
- Тогда неплохая хеш-функция для целых реализуется как

```
unsigned hashint(unsigned a, unsigned b, unsigned x) {  
    return (a*x + b) >> (w - M);  
}
```

- Заметьте, здесь $(a*x + b)$ уже вычисляется по модулю 2^{32} без явного деления
- Аналогично выкручиваются со строками (см. [WUNI])

Problem XC: подсчёт коллизий

- Ваша задача: имея произвольную функцию хеширования строки и произвольную последовательность строк, подсчитать количество коллизий

```
typedef int (*get_hash_t)(const char *s);  
  
int ncollisions(char **strs, get_hash_t f) {  
    // TODO: your code here  
}
```

- Эта функция впоследствии может быть использована для оценки качества хеш-функций над строками

Хеш-таблица как структура данных

- Как структура данных, хеш-таблица тоже инкапсулируется в отдельном модуле
- Интерфейс может выглядеть как:

```
struct hashmap_t; // объявление таблицы, детали спрятаны
```

```
struct hashmap_t *hashmap_create(unsigned m);  
int hashmap_add(struct hashmap_t *h, unsigned key,  
               const char *value);
```

```
const char *hashmap_find(struct hashmap_t *h, unsigned key);  
void hashmap_destroy(struct hashmap_t *h);
```

- Разумеется реальный интерфейс может быть гораздо богаче и интереснее

Домашняя работа HWH – словарь

- На стандартном вводе название текстового файла, лежащего по рабочему пути программы и список слов
 - На стандартном выводе список частот встретившихся слов
 - Например пусть text.txt содержит стишок отсюда:
https://en.wikipedia.org/wiki/This_Is_the_House_That_Jack_Built
- ```
> echo "text.txt farmer horse Jack" > test.stdin
> ./a.out < test.stdin
1 2 12
```
- Поскольку слово farmer встретилось 1 раз и так далее
  - Постарайтесь использовать хеш-таблицы для упрощения подсчёта

# Снова о поиске подстроки в строке

- Можем ли мы использовать хеширование для эффективного поиска подстроки?

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | B | A | B | A | C | A | A | B | A | C | A | B | C | A | C | B | C | A | A | A |
| A | B | A | C | A | B |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

- Вычислим хеш  $h_{target} = h_{string}(ABACAB)$
- Здесь функция  $h_{string}$  это упрощённая универсальная функция для строк  
$$h_r(c_1 \dots c_l) = \left( \sum_{i=1}^l c_i r^{l-i} \right) \% p$$
- Первые шесть символов строки в которой мы ищем дают  $h_{string}(ABABAC)$
- Можем ли мы легко перейти от него к  $h_{string}(BABACA)$ ?

# Циклические свойства хеш-функции

- Ещё раз посмотрим на формулу  $h_r(c_1 \dots c_l) = (\sum_{i=1}^l c_i r^{l-i}) \% p$
- $h_r(c_2 \dots c_{l+1}) = ((h_r(c_1 \dots c_l) - c_1 r^{l-1}) * r + c_{l+1}) \% p$
- Интуитивно: мы убираем **первый** символ, сдвигаем строку умножением и добавляем **последний**
- Можно предварительно подсчитать  $n = r^{l-1} \% p$

• Сигнатура для функции обновления

```
unsigned update_hash(unsigned hash, unsigned n, char cf, char cl);
```

- Напишите эту функцию

# Обсуждение

- Циклические свойства хеш-функций наталкивают на идею «циклического хеша», который обновляется с каждым новым продвижением подстроки

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | B | A | B | A | C | A | A | B | A | C | A | B | C | A | C | B | C | A | A | A |
|   | A | B | A | C | A | B |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

- Вместо  $O(Nm)$  имеем  $O(N)$  потому что обновление хеша происходит за константное время
- Это называется алгоритмом Рабина-Карпа

# Алгоритм РК – поиск подстроки

- Проверяет наличие подстроки `needle` в строке `haystack`

```
// assume strlen(needle) much lesser then strlen(haystack)
int rabin_karp(const char *needle, const char *haystack) {
 unsigned n, target, cur, count = 0, left = 0, right = strlen(needle);

 target = get_hash(needle, needle + right);
 cur = get_hash(haystack, haystack + right);
 n = pow_mod(R, right - 1, Q); // алгоритм POWM

 while(target != cur && haystack[right] != 0) {
 cur = update_hash(cur, n, haystack[left], haystack[right]);
 left += 1; right += 1;
 }

 return (target == cur) ? left : 0;
}
```

# Problem КС: коллизии Рабина-Карпа

- В алгоритме РК никак не обработан случай коллизии хеш-функции, когда хеши совпали, а строка найдена неверно

- Ваша задача доработать с учётом коллизий функцию

```
int rabin_karp(const char *needle, const char *haystack);
```

- Выберите в качестве хеш-функции нечто не слишком совершенное, например

$$h(c_1 \dots c_l) = \left( \sum_{i=1}^l c_i 10^{l-i} \right) \% 31$$

- Проверьте как работает поиск с коллизиями



# Обсуждение: гистограммный поиск

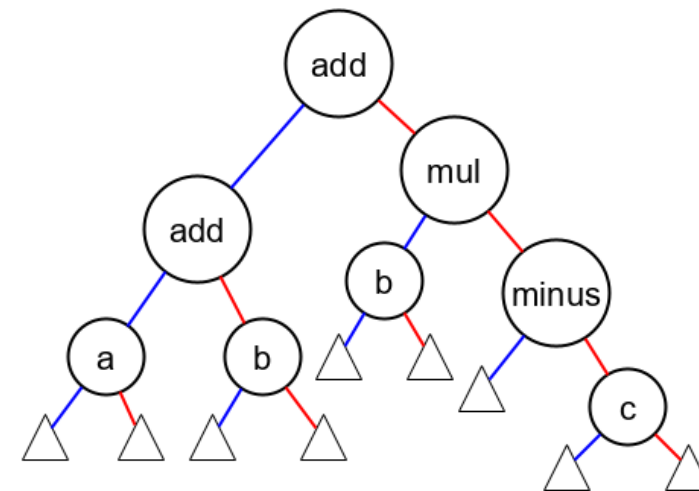
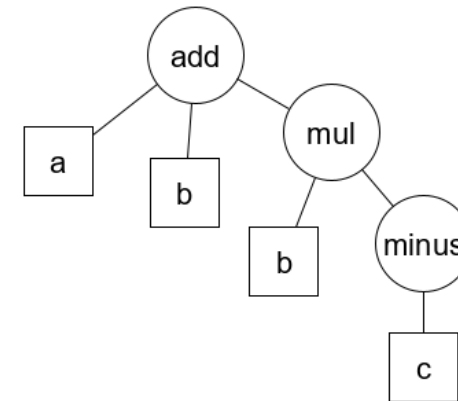
- Что сложнее найти в строке: конкретную подстроку или любую перестановку символов из конкретной подстроки?
- Как бы вы решали задачу нахождения перестановки символов?
- Например: ищем все перестановки AABCD
- Находим: AB**BAD**CA**DD**C
- Можно ли решить эту задачу очень просто, используя хеш-таблицы?

# Обсуждение

- Главный недостаток хеш-таблиц (и шире хеш-функций как идеи) это стирание информации о естественном порядке объектов
- Например пусть нужно индексировать города расстоянием от Москвы
  - Сложить их в хеш-таблицу по этому расстоянию легко
  - Также легко получить город на расстоянии 60 километров
  - Но увы, получить все города от 50 до 100 километров невозможно
- Говорят, что хеш-отображения не позволяют делать [range-based queries](#)
- И это естественным образом приводит нас к [поисковым деревьям](#)
- Но сначала давайте поговорим о просто деревьях

# Деревья и бинарные деревья

- **Дерево**  $T$  это множество узлов, один из которых является корнем  $root(T)$ , а остальные можно разделить на непересекающиеся множества  $T_i$  (поддеревья  $T$ ), каждое из которых также является деревом
- **Бинарное дерево**  $B$  это множество узлов, которое либо пусто, либо содержит корень  $root(B)$  и элементы двух непересекающихся бинарных деревьев: левого  $B_L$  и правого  $B_R$
- Далее мы будем изучать бинарные деревья



# Представление в текстовом файле

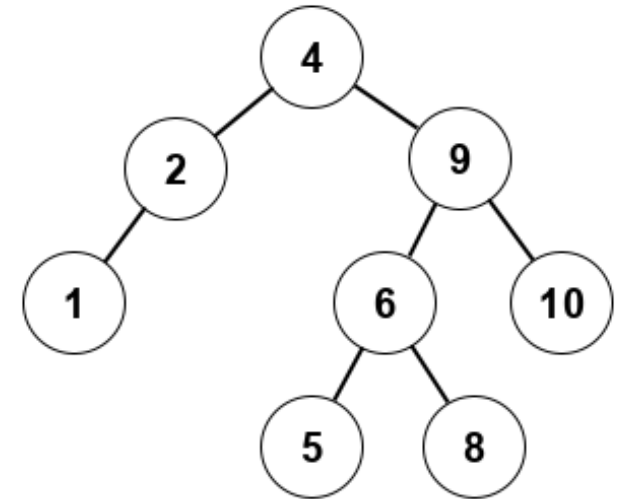
- Естественное представление: хранить топологию отдельно и значения отдельно

8

1 1 1 0 0 0 1 1 1 0 0 1 0 0 1 0

4 2 1 9 6 5 8 10

- Далее будем называть это "обычным представлением"
- Обратите внимание на картинке справа не нарисованы нулевые листья, но мы всегда помним, что они есть



# Структура данных: первая попытка

- Самый низкоуровневый способ работать с бинарными деревьями это работать со структурой отдельного узла

```
struct node_t;
```

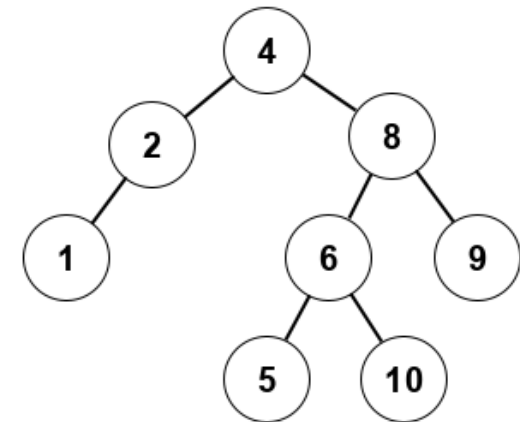
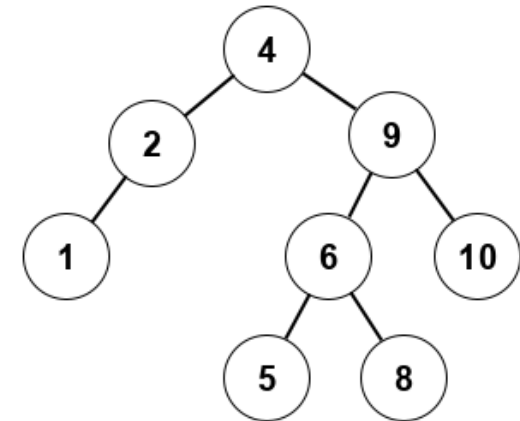
```
struct node_t *node_alloc(int, struct node_t *, struct node_t *);
void node_free(struct node_t *);
```

```
struct node_t * node_left(struct node_t *);
struct node_t * node_right(struct node_t *);
int node_data(struct node_t *);
```

- В таком виде инкапсуляции в общем немного. Такую структуру можно запутать даже не зная представления node\_t

# Поисковые бинарные деревья

- Поисковым бинарным деревом называется дерево для которого:
  - $x \in B_L \leftrightarrow x < \text{root}(B)$
  - $x \in B_R \leftrightarrow x > \text{root}(B)$
- Одно (обычно первое) из этих соотношений может быть не строгим
- Как вы думаете почему такие деревья называют поисковыми?
- Оба ли дерева справа поисковые?



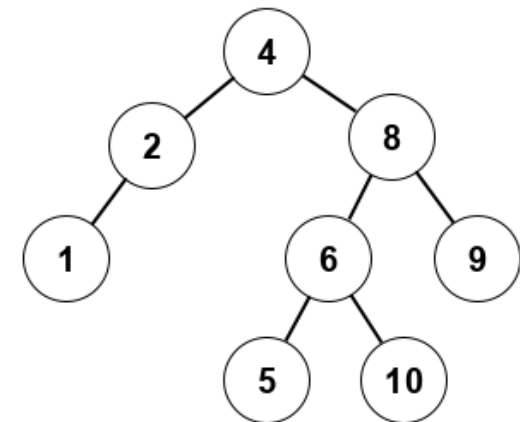
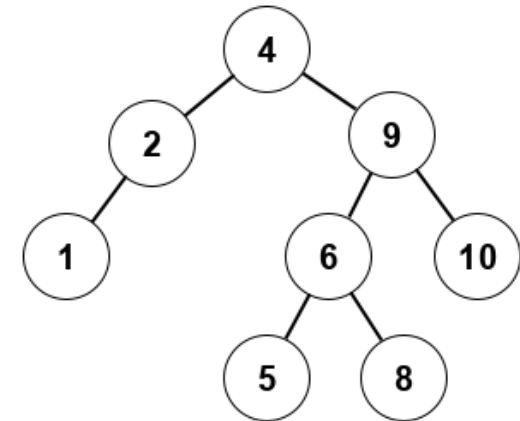
# Problem IS – проверить поисковость

- На входе файл содержащий дерево в обычном представлении
- На выходе 1 если дерево поисковое и 0 если нет

8 1 1 1 0 0 0 1 1 1 0 0 1 0 0 1 0 4 2 1 9 6  
5 8 10

8 1 1 1 0 0 0 1 1 1 0 0 1 0 0 1 0 4 2 1 8 6  
5 10 9

- Ответ: в первом случае поисковое, во втором нет, см. картинку справа



# Алгоритм RQ – поиск диапазона

- Важнейшим преимуществом поисковых деревьев над хеш-таблицами является возможность запроса диапазона

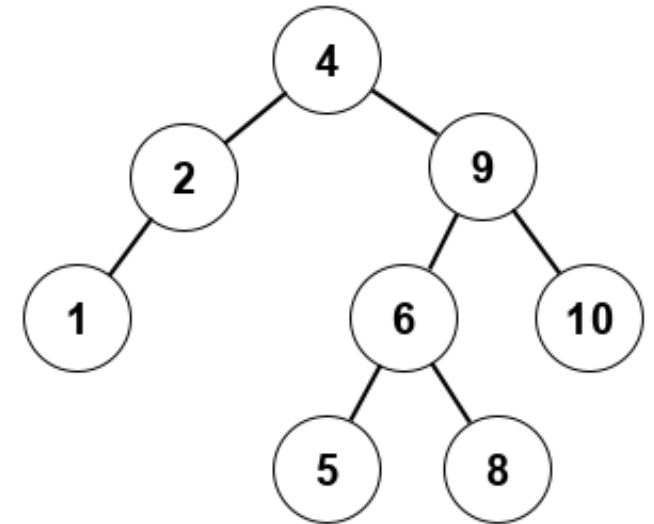
```
typedef void (*visit_t)(const struct node_t *);
void visit_range(struct node_t *top, int l, int r, visit_t v) {
 if (NULL == top) return;
 if (l <= node_data(top) && r >= node_data(top))
 v(top);
 if (l <= node_data(top))
 visit_range(node_left(top), l, r, v);
 if (r >= node_data(top))
 visit_range(node_right(top), l, r, v);
}
```



# Обход дерева алгоритмом RQ

- Можно сказать, что алгоритм RQ обходит дерево
- Допустим на входе дерево как слева с запросом [2, 8]
- В каком порядке узлы будут выведены?

```
if (/* some check */)
 visit(root);
if (/* some check */)
 visit_range(/* to left */, l, r, v);
if (/* some check */)
 visit_range(/* to right */, l, r, v);
```



- Можно ли изменить алгоритм RQ для сортированного вывода: 2 4 5 6 8?

# Удаление и топологический обход

- Попробуем реализовать метод

```
void node_free(struct node_t *);
```

- Для этого тоже нужно как-то обойти дерево, но как?
- Дополнительно: представим, что наше дерево это дерево зависимостей при этом поддеревья не могут быть выполнены раньше чем корень

```
typedef void (*visit_t)(const struct node_t *);
```

```
void node_topological(struct node_t *, visit_t);
```

- Можем ли мы вывести дерево в топологическом порядке 4 2 1 9 6 10 5 8?

# Классификация обходов

- Inorder: LNR
  - Используется для линейаризации дерева и для поисковых деревьев даёт сортированный массив
- Preorder: NLR
  - Топологический порядок если потомки зависят от корня
- Postorder: LRN
  - Порядок удаления дерева
- Можно придумать много других обходов, например обход в ширину, в глубину и так далее

# Problem TT – дерево из обходов

- На входе файл, содержащий данные а также preorder и inorder обходы дерева в терминах номера узла в массиве с данными

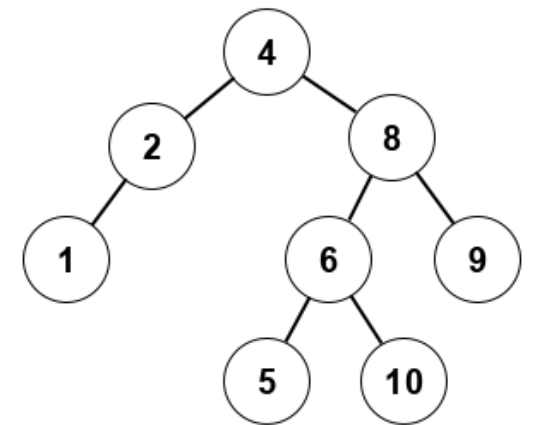
8

```
1 2 4 5 6 8 9 10 /* node #0 have data '1' */
```

```
0 1 2 3 4 7 5 6 /* inorder */
```

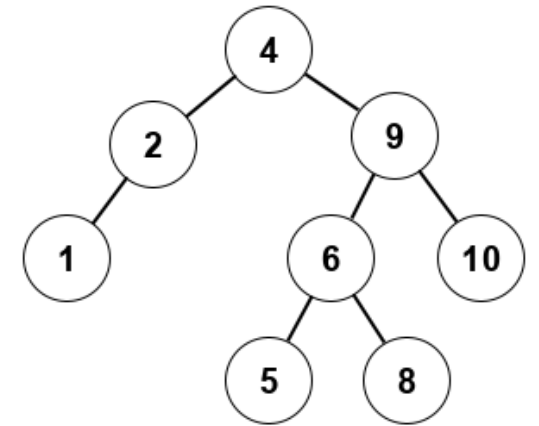
```
2 1 0 5 4 6 3 7 /* preorder */
```

- На выходе дерево в обычном представлении
- Теоретическое задание: подумайте хватит ли вам любой пары из трёх обходов?



# Естественный порядок

- Если бинарное дерево является поисковым, то его топология диктуется добавлением в него новых элементов
- Например 4 2 1 9 6 10 5 8 и т.д.
- Мало того, поисковые деревья обладают своего рода устойчивостью к небольшим нарушениям порядка
- Скажем 4 2 9 10 6 8 5 1 даст то же дерево справа
- В целом мы можем переставлять любые поддеревья
- Тем не менее, 4 1 2 9 10 6 8 5 или 1 2 4 9 10 6 8 5 это уже несколько другие деревья (нарисуйте их)



# Problem NO – natural order

- На входе количество узлов и потом последовательность чисел для вставки

8

4 2 9 10 6 8 5 1

- На выходе -- бинарное дерево в обычном представлении

8

1 1 1 0 0 0 1 1 1 0 0 1 0 0 1 0

4 2 1 9 6 5 8 10

- Обратите внимание: обычное представление задаёт класс эквивалентности

# Структура данных для дерева

- Теперь мы можем работать с поисковым бинарным деревом не в терминах его узлов, а в терминах его данных

```
struct tree_t; // balanced search tree
```

```
void tree_add_value(struct tree_t *top, int value);
```

```
void tree_has_value(struct tree_t *top, int value);
```

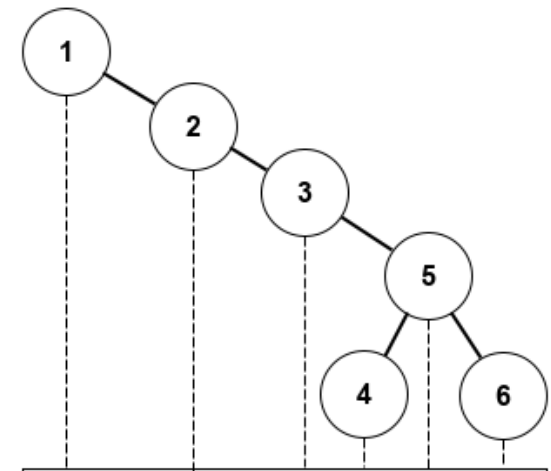
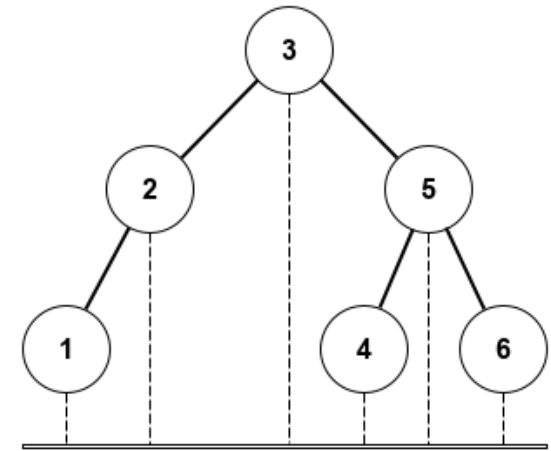
```
void tree_visit_range(struct tree_t *top, int l, int r, visit_t v);
```

```
void tree_free(struct tree_t *top);
```

- И только в этом случае упорядоченное отображение (дерево) становится такой же настоящей структурой данных, какой ранее стало неупорядоченное (хеш)

# Разбалансированность

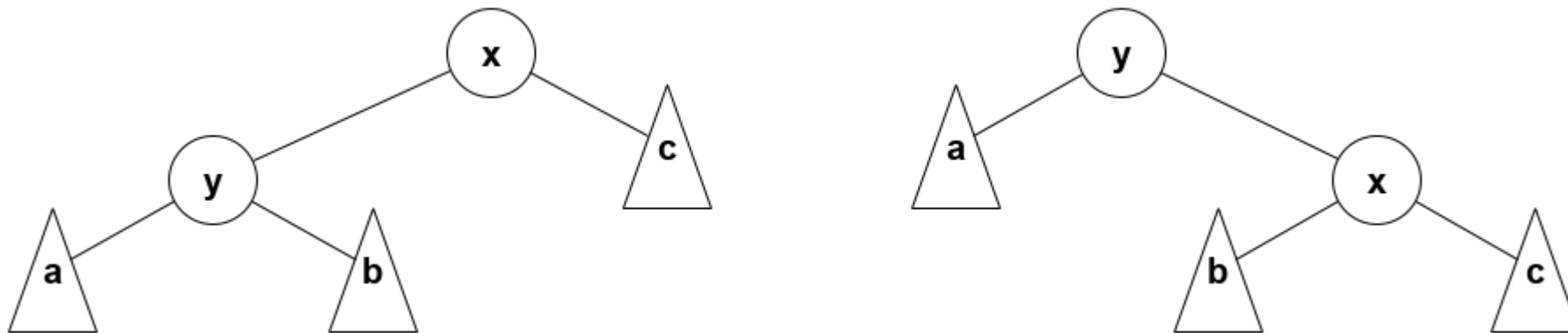
- Существенной проблемой для алгоритма RQ является то, что он логарифмический только в лучшем случае
- Мы можем легко представить плохой случай: при неудачном порядке добавления тех же узлов дерево может не быть сбалансированным
- В этом случае поиск в дереве (даже в поисковом) не лучше, чем поиск в списке
- Поэтому поисковые деревья принято при построении балансировать





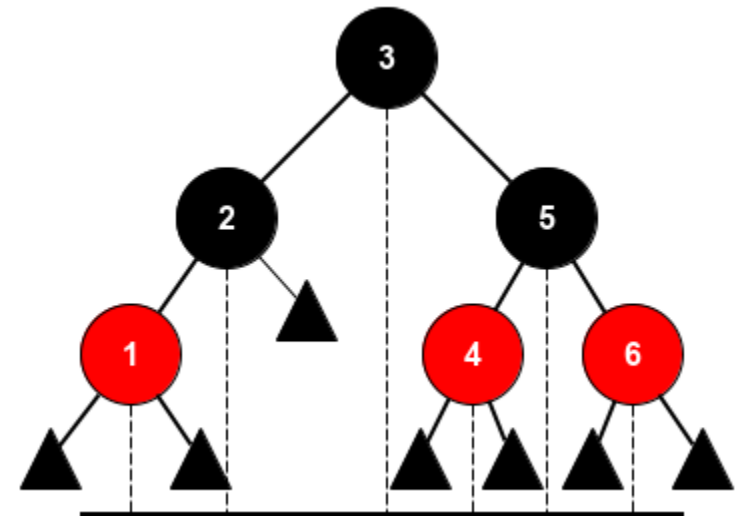
# Поддержка свойств поисковости

- Базовая трансформация сохраняющая свойства поисковости но меняющая относительную высоту поддеревьев это левый и правый поворот



# Красно-черные деревья

- Надлежащим количеством поворотов можно сделать любое дерево полезным, но это нетривиальная задача
- Гораздо проще при каждой вставке поддерживать поворотами какой-нибудь инвариант, который гарантирует нам полезность дерева
- **Красно-черный** инвариант:
  - Корень черный
  - Все нулевые потомки черные
  - У каждого красного узла все потомки чёрные
  - На любом пути от данного узла до каждого из нижних листьев одинаковое количество чёрных узлов



# Обсуждение

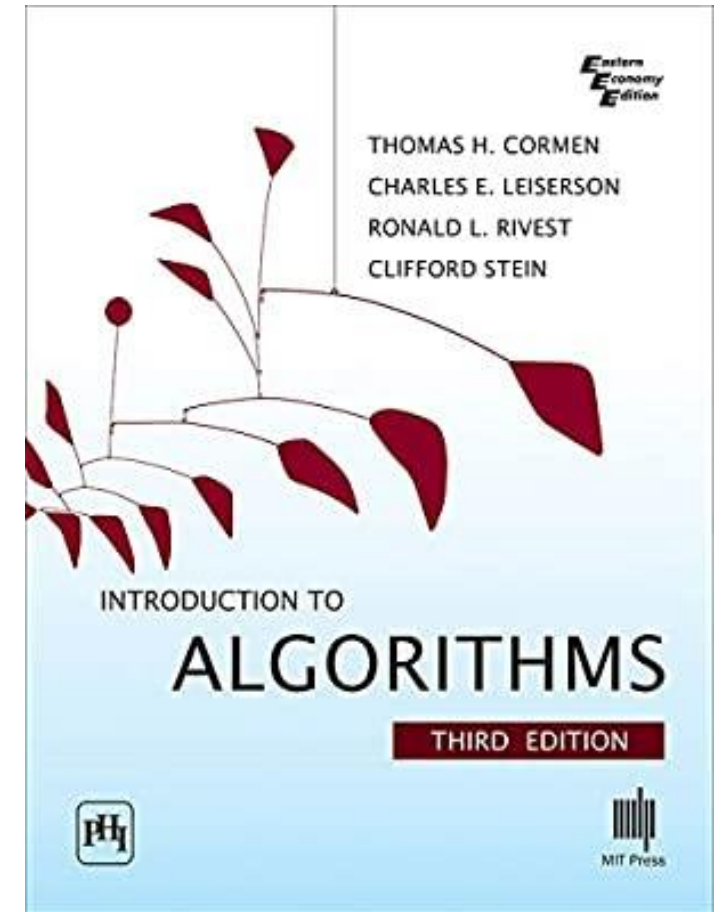
- Если мы добавим поддержку красно-черного инварианта при вставках и удалениях, изменит ли это нашу структуру данных?

# Problem RBO: запросы городов

- На выходе количество записей а потом на каждой строчке название города без пробелов и далее расстояние до центра страны
- Далее на входе количество запросов и сами запросы (два числа от и до)
- На выходе: количество городов на расстоянии от левой до правой границы запроса включительно

# Литература

- [C11] ISO/IEC – "Information technology – Programming languages – C", 2011
- [K&R] Brian W. Kernighan, Dennis Ritchie – The C programming language, 1988
- [Cormen] Thomas H. Cormen – Introduction to Algorithms, 2009
- [TAOCP] Donald E. Knuth – The Art of Computer Programming, 2011
- [SALG] Robert Sedgwick Algorithms, 4th edition, 2011
- [WUNI] [en.wikipedia.org/wiki/Universal\\_hashing](http://en.wikipedia.org/wiki/Universal_hashing)



# СЕКРЕТНЫЙ УРОВЕНЬ

---

Перестановки

# Интересное наблюдение

- Inorder обход поискового дерева всегда отсортирован
- При этом один и тот же сортированный массив мы можем "поднять" в разные деревья и у всех у них будут разные preorder обходы
- Это значит, что дерево задаёт перестановку
  - $1\ 2\ 3\ 4\ 5\ 6 \rightarrow 3\ 2\ 1\ 5\ 4\ 6$  т.е.  $(1\ 3)(4\ 5)$
  - $1\ 2\ 3\ 4\ 5\ 6 \rightarrow 1\ 2\ 3\ 5\ 4\ 6$  т.е.  $(4\ 5)$
- Найдите перестановку не представимую деревом.
- Что общего у тех, которые представимы?

