

# СТРУКТУРЫ ДАННЫХ

---

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

К. Владимиров, Syntacore, 2023  
mail-to: [konstantin.vladimirov@gmail.com](mailto:konstantin.vladimirov@gmail.com)

# СЕМИНАР 4.1

---

Односвязные списки.

# 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));
```

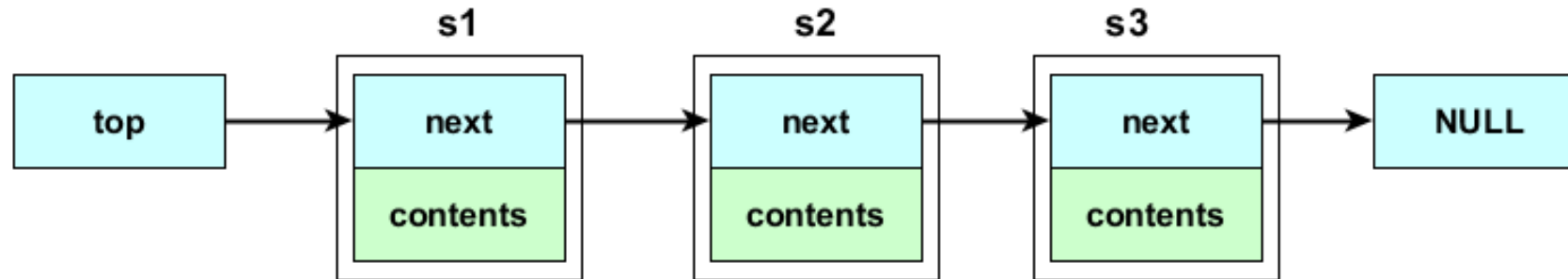
```
top->next = calloc(1, sizeof(struct node_t));
```

- 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; };
```

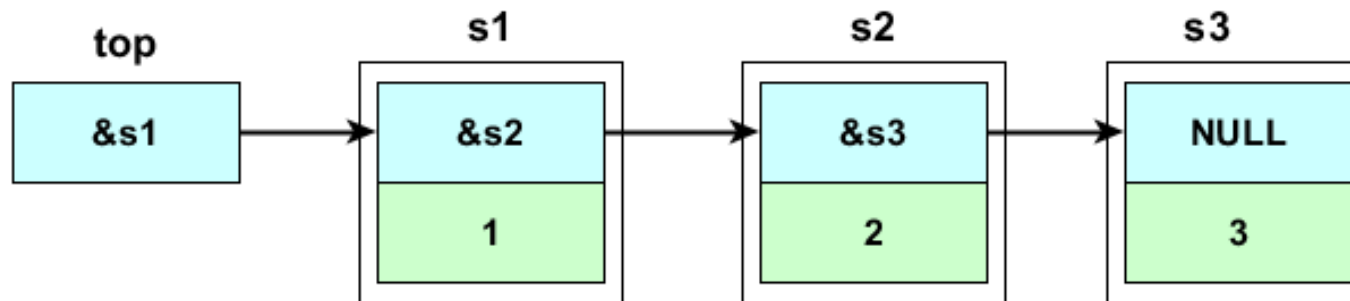


- Картинку можно несколько конкретизировать, учтя значения полей, выставленные на предыдущем слайде

# 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);
```

# Bucket sort: мозговой штурм

- Давайте подумаем как применить списки к сортировке.
- Основной проблемой counting sort была необходимость выделять в плохих случаях куда больше бакетов чем элементов в массиве.

|     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|
| 126 | 348 | 343 | 432 | 316 | 171 | 556 | 670 |
|-----|-----|-----|-----|-----|-----|-----|-----|

0   1   2   3   4   5   6   ...   126   ...

|   |   |   |   |   |   |   |     |   |     |
|---|---|---|---|---|---|---|-----|---|-----|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | ... | 1 | ... |
|---|---|---|---|---|---|---|-----|---|-----|

- Если у нас есть динамические структуры данных, можно ли это митигировать?

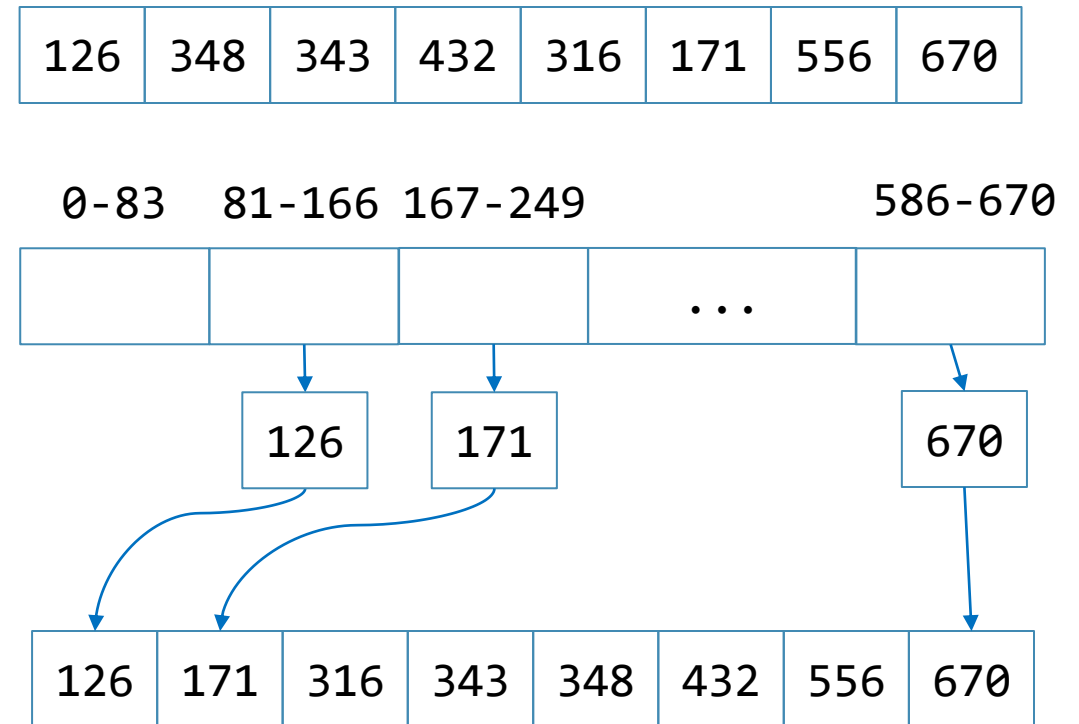
# Bucket sort: основная идея

- Выделим **столько же** бакетов, сколько элементов в массиве.
- При вставке в каждый бакет элемент ставится на своё место

335-418



- В конце мы просто проходим все бакеты и комбинируем их.



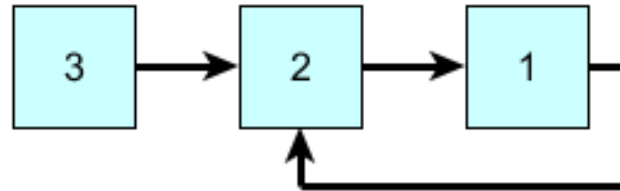
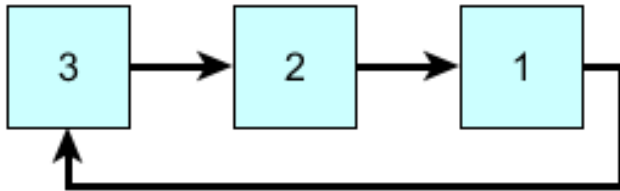
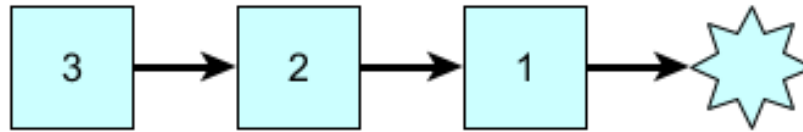


# Problem RBS

- Сформируем бакеты разделив максимальное число  $M$  на  $N$  частей. В бакете с номером  $k$  хранятся числа от  $(k - 1) \left\lfloor \frac{M}{N} \right\rfloor$  до  $k \left\lfloor \frac{M}{N} \right\rfloor$ . В последнем бакете хранятся все до конца.
- Ваша задача распечатать через пробелы все бакеты. Разделите их нулевыми символами (ноль в конце каждого бакета).
- Вход: 8 126 348 343 432 316 171 556 670
- Выход: 0 126 0 171 0 316 0 343 348 0 432 0 556 0 670 0
- Вход: 10 187 329 731 517 71 468 429 237 621 860
- Выход: 71 0 0 187 237 0 329 0 429 0 468 0 517 0 621 0 731 0 860 0

# Обсуждение

- Кажется циклический список мало отличается от обычного...



# Задача: есть ли петля?

- Вам на вход приходит связный список

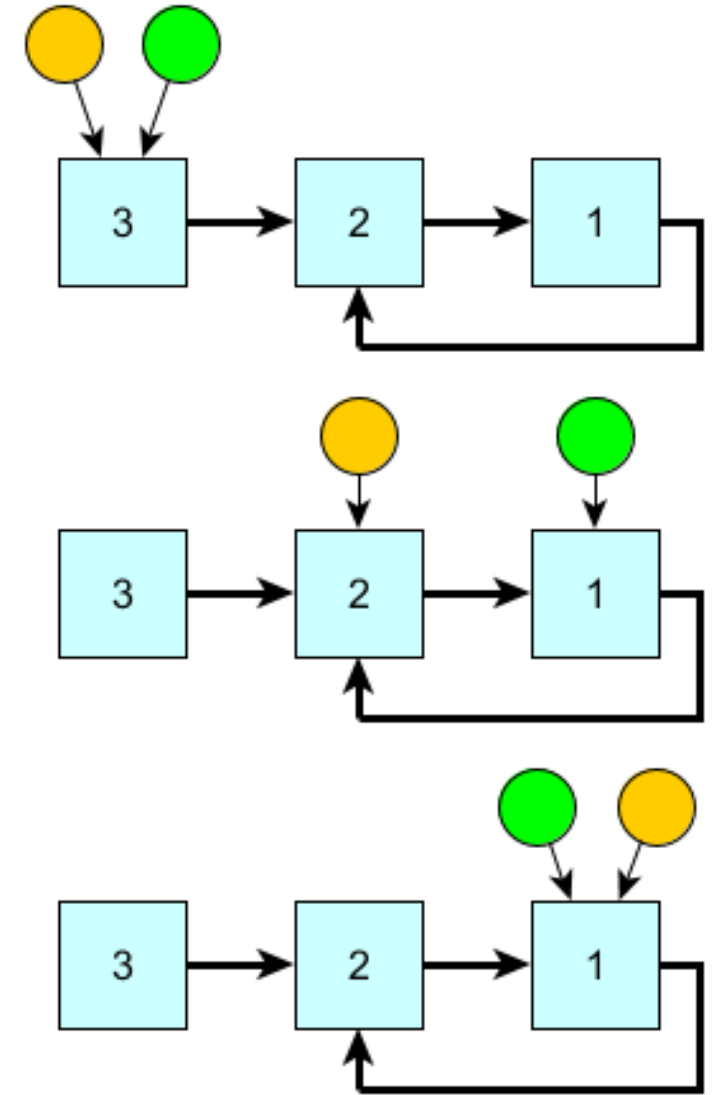
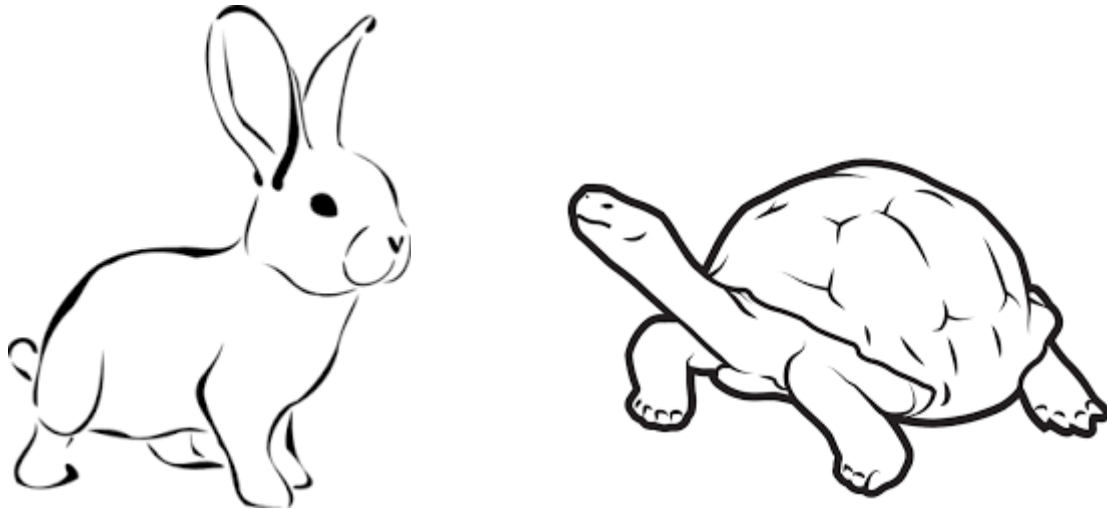
```
struct list_t {  
    struct list_t *next;  
    int data;  
};
```

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

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

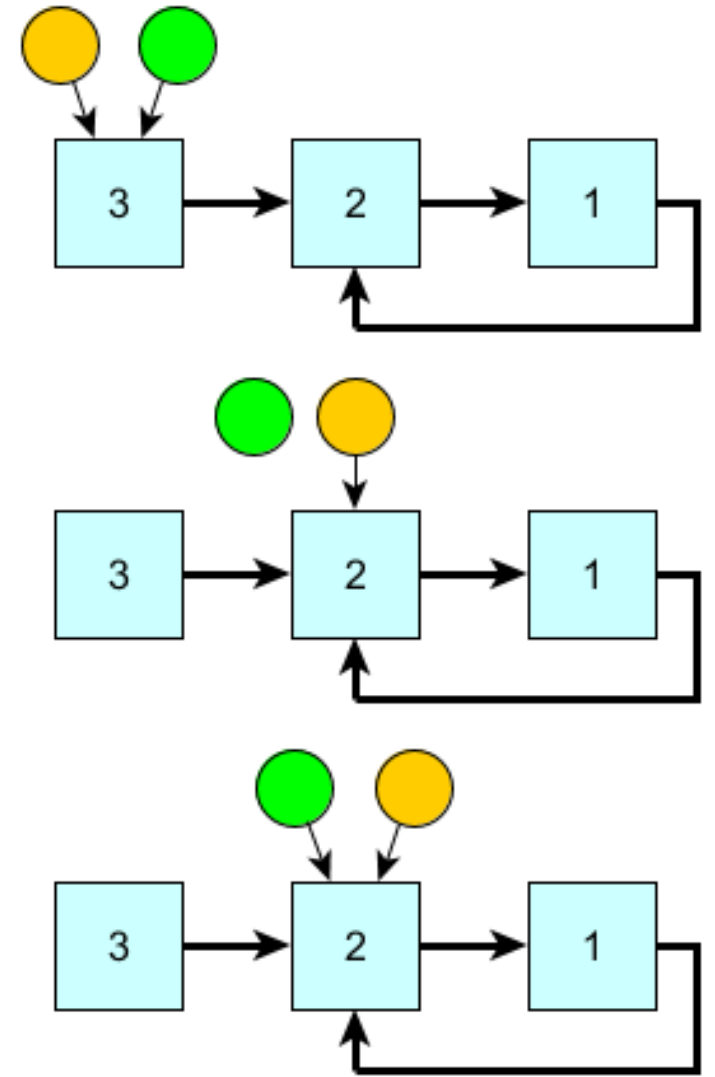
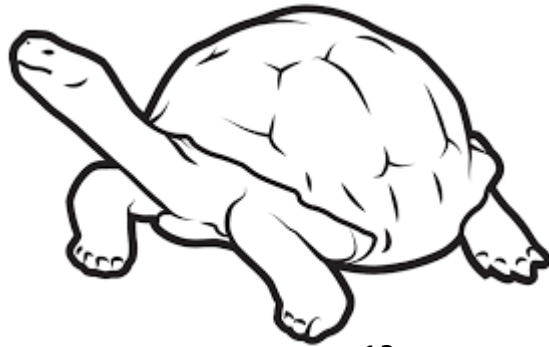
# Алгоритм Флойда

- 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  $f$
- 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);
```

- 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);
```

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

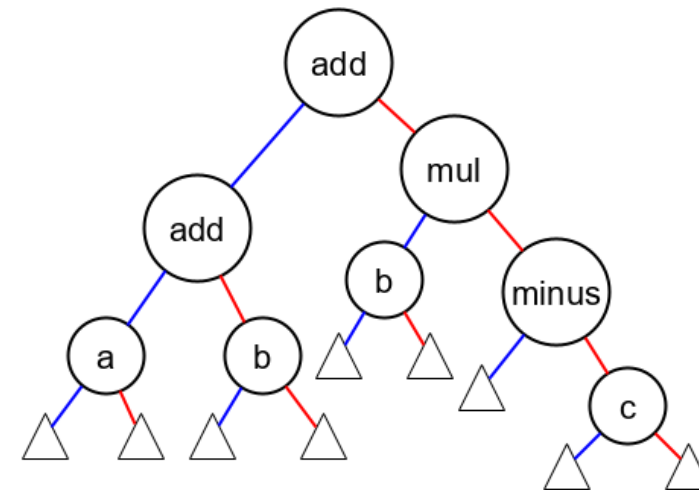
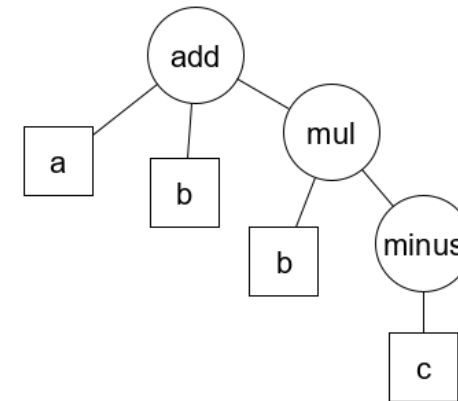
# СЕМИНАР 4.2

---

Деревья и их обходы

# 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  $B_R$ .



# Как представить бинарное дерево на C

```
struct node {  
    struct node *parent; // родитель  
    struct node *left;   // выделенный левый потомок  
    struct node *right;  // выделенный правый потомок  
    void *data;          // данные в узле  
};
```

- Теперь мы различаем левого и правого потомка.
- Чаще всего нас просят так или иначе **обойти** дерево.

# Классификация обходов

- Inorder: LNR
  - Используется для линеаризации дерева.
- Preorder: NLR
  - Топологический порядок.
- Postorder: LRN
  - Порядок удаления дерева.
- Можно придумать много других обходов, например обход в ширину, в глубину и так далее.

# Problem IPO

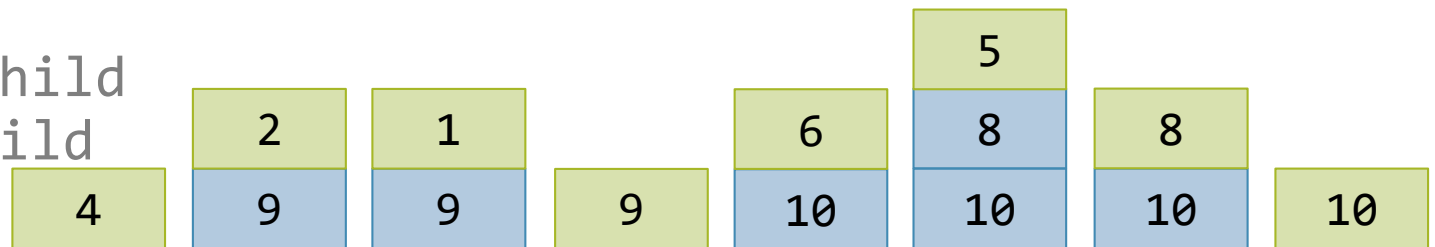
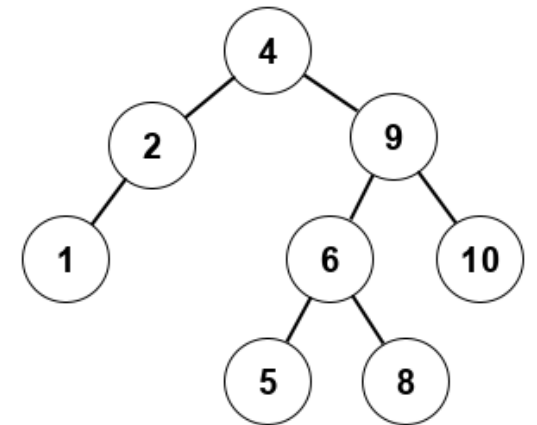
- На входе указатель на вершину дерева.
- На выходе её preorder обход.



# Обходы итерацией и стек

- Скорее всего прошлую проблему вы решали рекурсивно.
- Как могло бы выглядеть решение без рекурсии?

```
void iterative_preorder(node* root) {  
    stacknode* s = NULL;  
    stack_push(&s, root);  
  
    while (!empty(s)) {  
        // pop item  
        // print it  
        // push its right child  
        // push its left child  
    }  
}
```



# Организация стека

- Внезапно стек это просто односвязный список.

```
struct stacknode {  
    struct stacknode *next;    // следующий элемент  
    struct node *data;         // узел в стеке  
};
```

- Довольно легко написать push и pop.

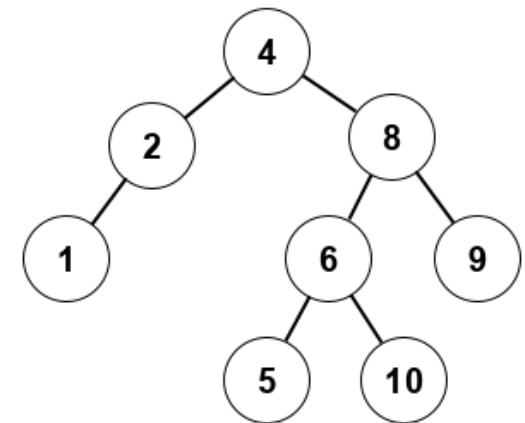
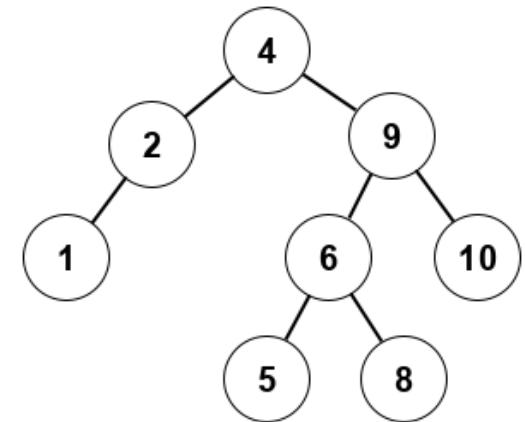
```
void push(struct stacknode **stack, struct node *data) {  
    struct stacknode *tmp = calloc(1, sizeof(struct stacknode));  
    tmp->data = data; tmp->next = *stack;  
    *stack = tmp;  
}
```

# Problem SPO

- На входе указатель на вершину дерева.
- На выходе её preorder обход, полученный итеративно.

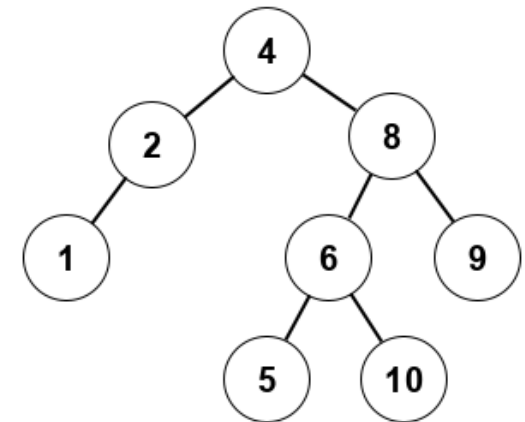
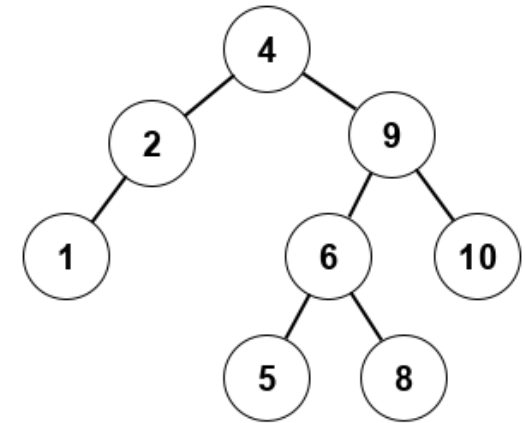
# Поисковые бинарные деревья

- 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?



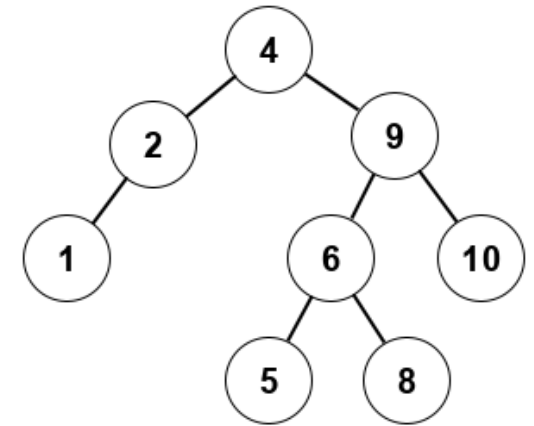
# Problem IS – проверить поисковость

- На входе указатель на вершину дерева.
- На выходе 1 если дерево поисковое и 0 если нет
- В первом случае поисковое, во втором нет, см. картинку справа.



# 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 preorder порядок

4 2 1 9 6 5 8 10

- Обратите внимание: естественный порядок задаёт класс эквивалентности.

# Problem TT\* – дерево из обходов

- На входе файл, содержащий данные а также preorder и inorder обходы дерева

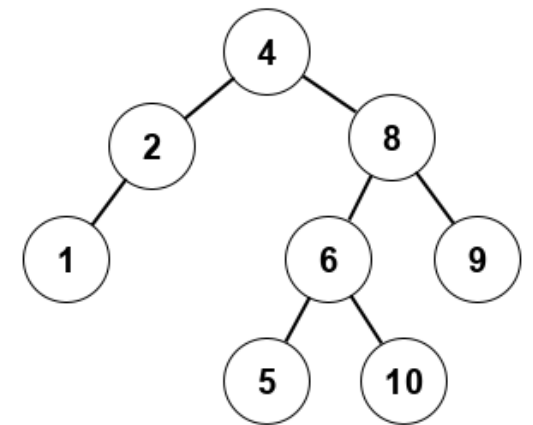
8

```
1 2 4 5 6 10 8 9  /* inorder */  
4 2 1 8 6 5 10 9  /* preorder */
```

- На выходе postorder порядок.

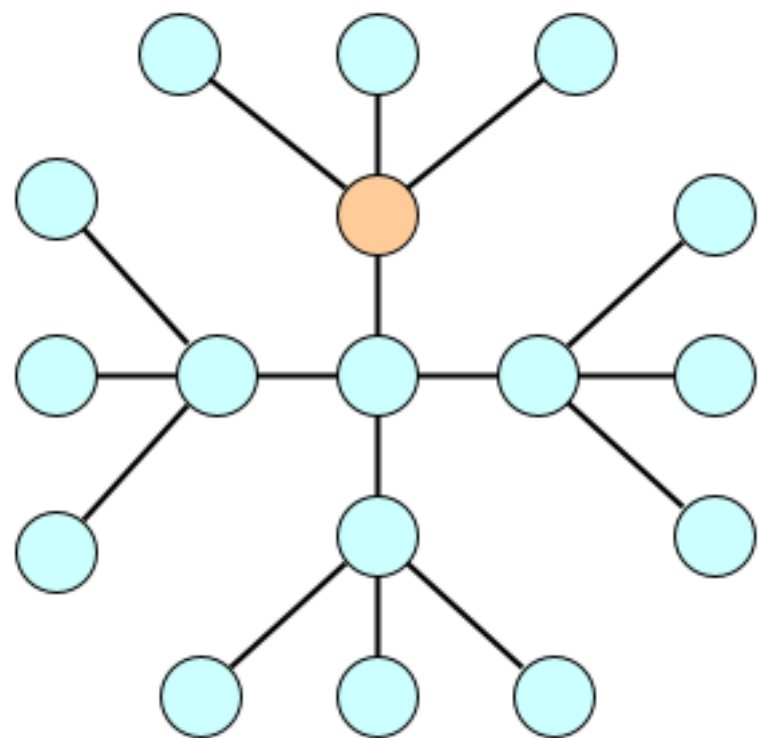
```
1 2 5 10 6 9 8 4  /* postorder */
```

- Теоретическое задание: подумайте хватит ли вам любой пары из трёх обходов, чтобы восстановить третий?

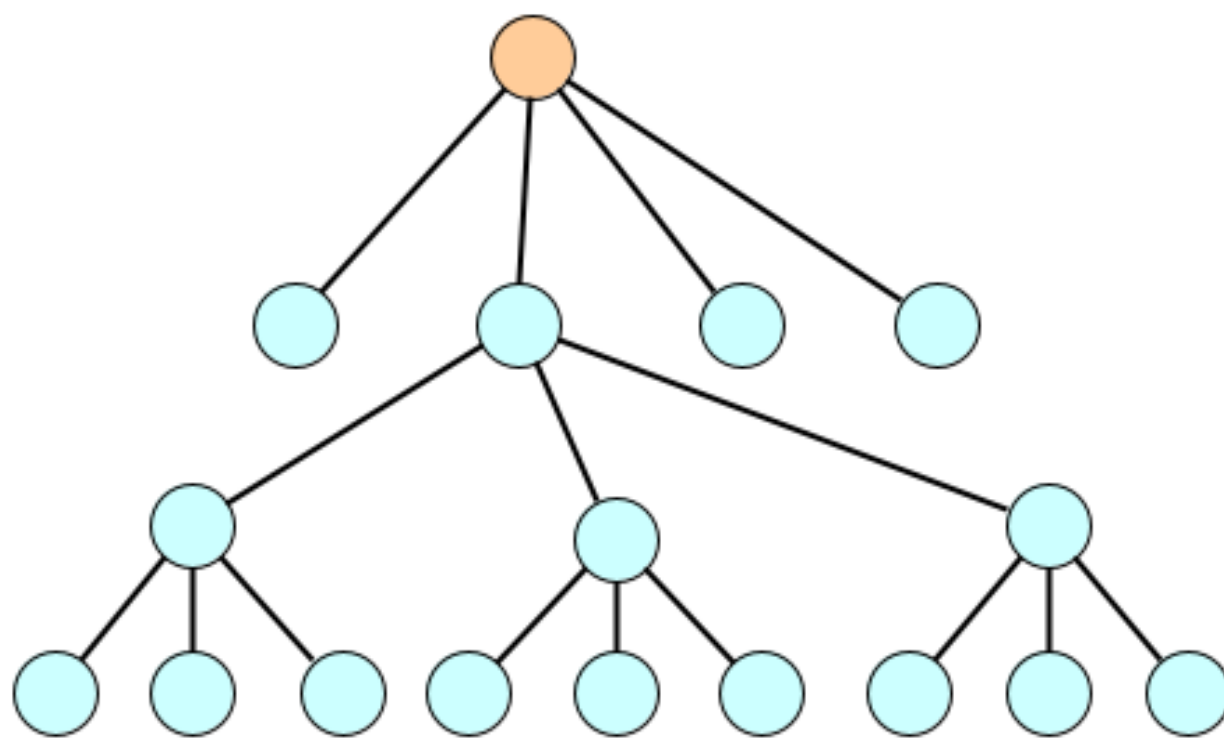
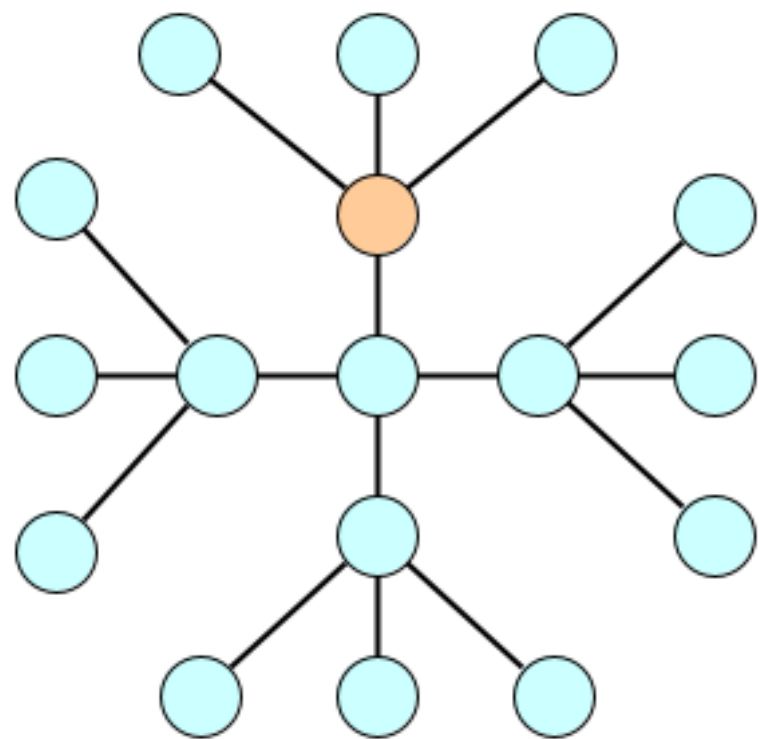




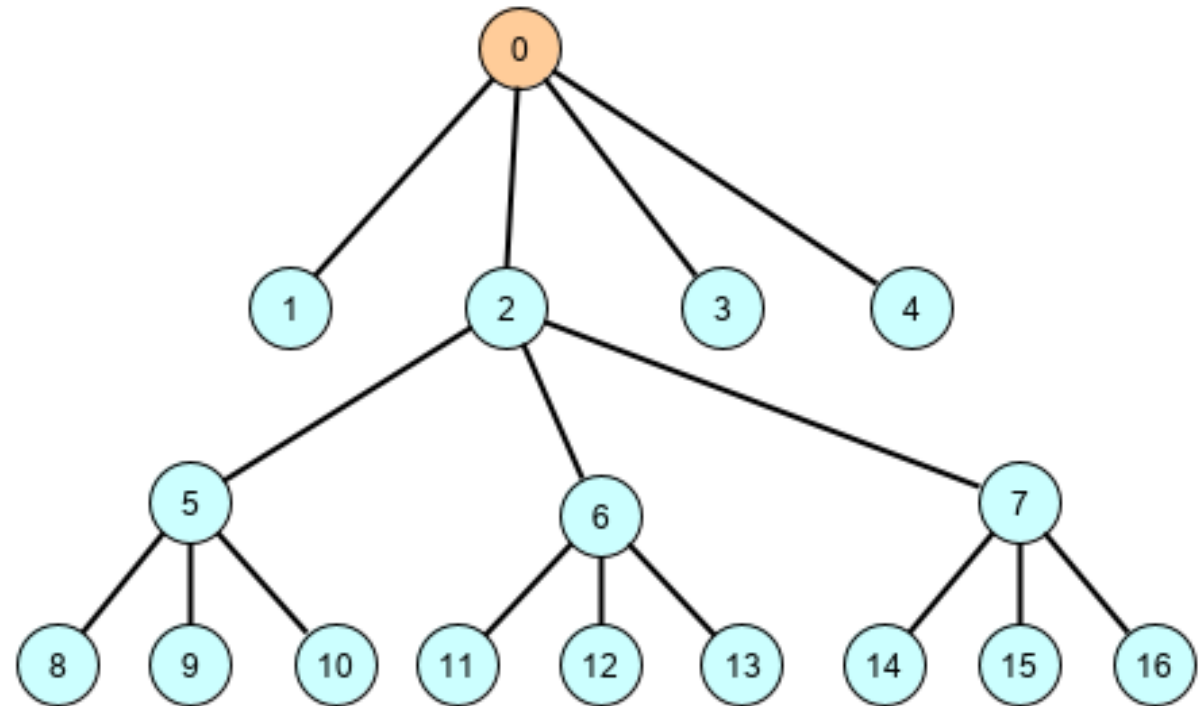
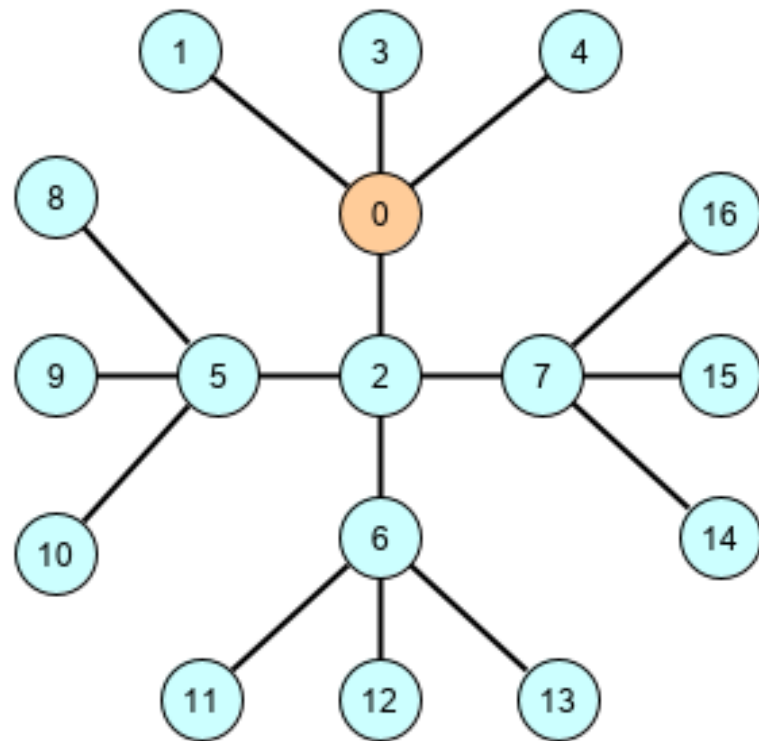
# Что перед вами?



# Что перед вами?



# Что перед вами?



# Как представить дерево на С

```
struct childlist_t;

struct node_t {
    struct node_t *parent;
    struct child_list_t *nodes; // список связанных вершин
    void *data;                // данные в узле
};

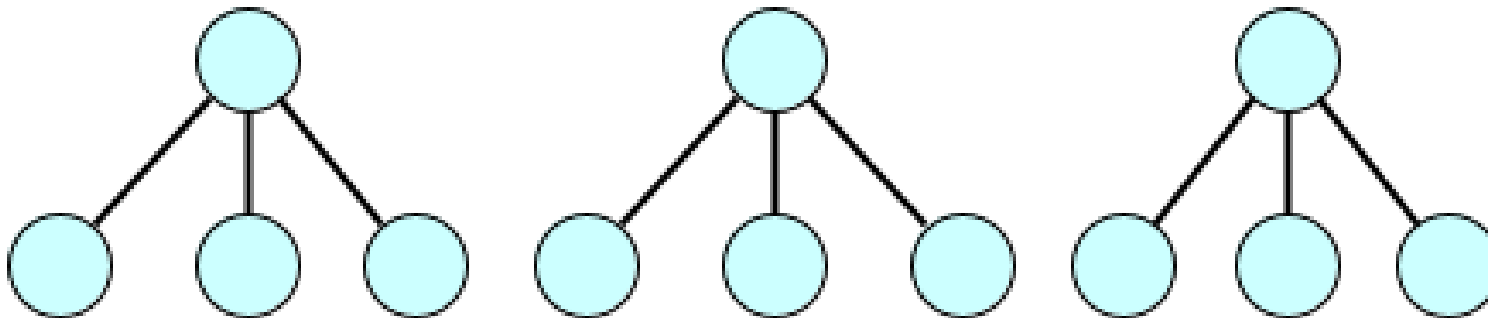
struct childlist_t {
    struct node_t *child;
    struct childlist_t *next;
};
```

- Это наиболее общее и при этом не слишком удобное представление.

# Немного о лесах

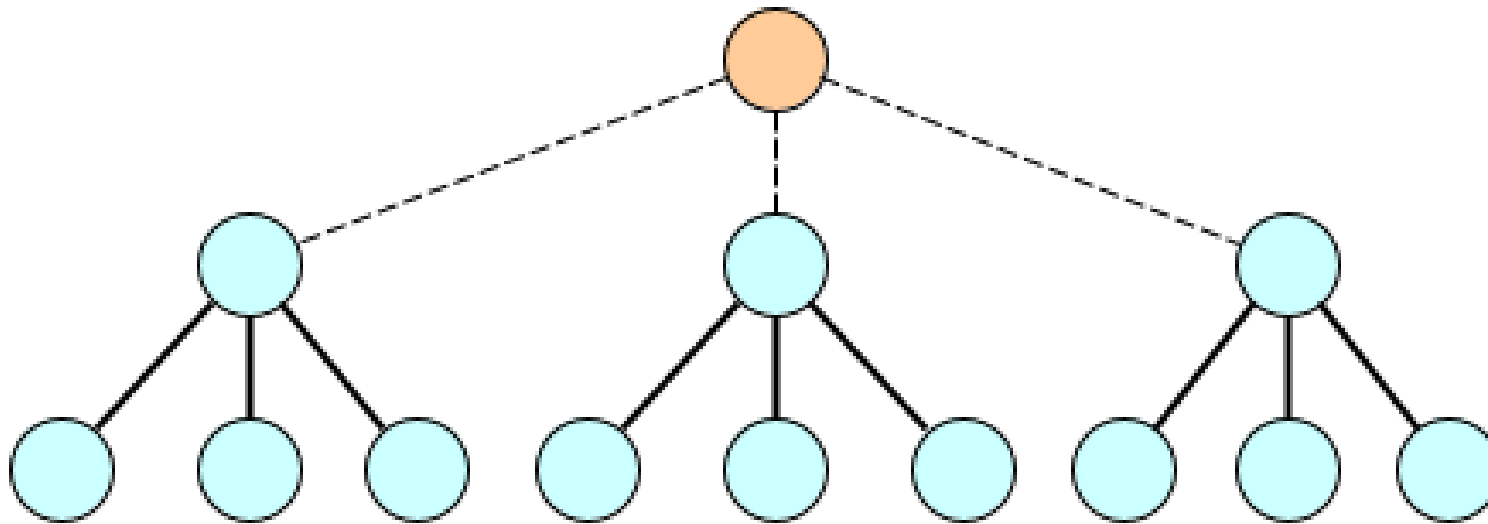
- Лес это одно или несколько деревьев. Как представить лес в программе на С?

```
struct forest_t {  
    struct child_list_t* topnodes_; // список деревьев  
};
```



# Немного о лесах

- Лес это одно или несколько деревьев. Как представить лес в программе на С?
- Оказывается лес это дерево без вершины

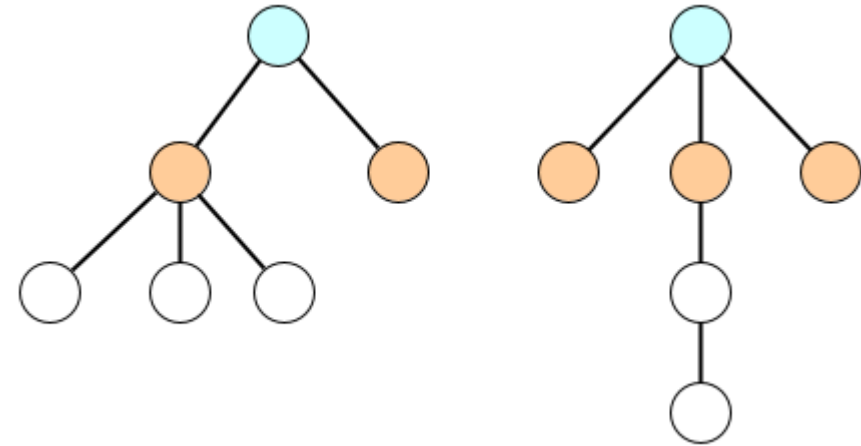


# Скобочное выражение это лес

- Рассмотрим правильную расстановку скобок

$((()())())((()((()))))()$

- Довольно очевидно, что это лес
- Но такое чувство что это не самое удобное представление для работы

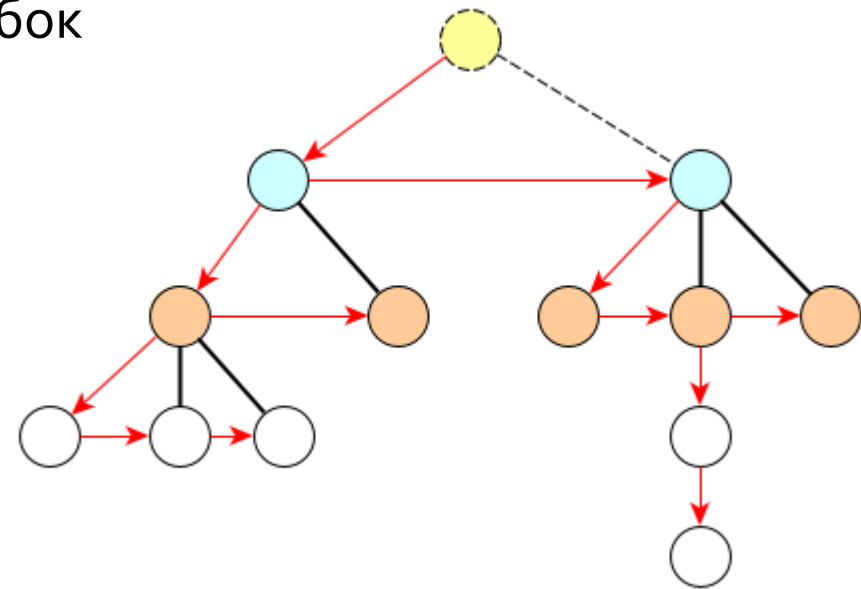


# Скобочное выражение это лес

- Рассмотрим правильную расстановку скобок

$((()())())((()((()))))()$

- Довольно очевидно, что это лес
- Но такое чувство что это не самое удобное представление для работы
- Проведём стрелки к первому брату и к первому потомку
- И внезапно мы видим...



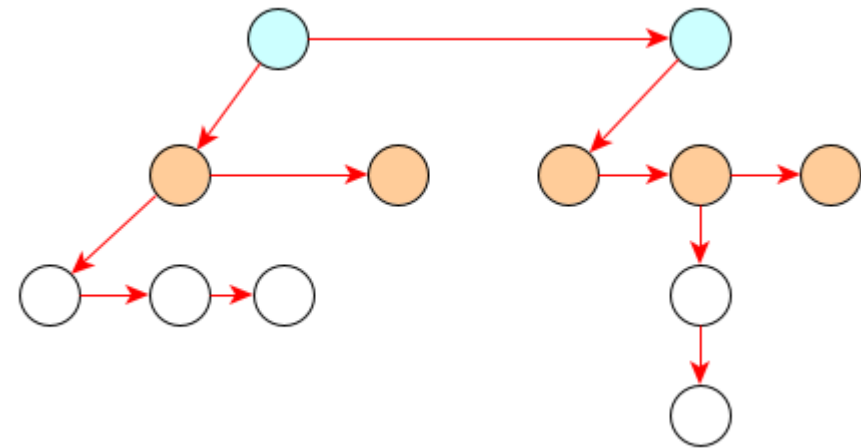


# Скобочное выражение это лес

- Рассмотрим правильную расстановку скобок

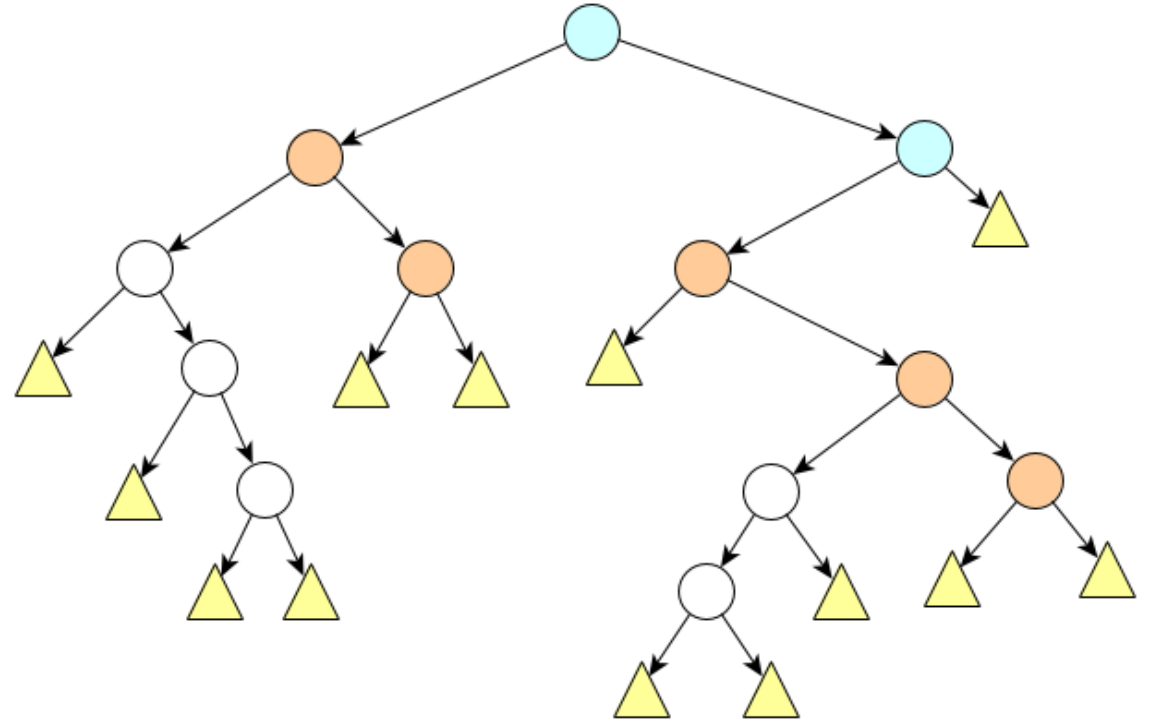
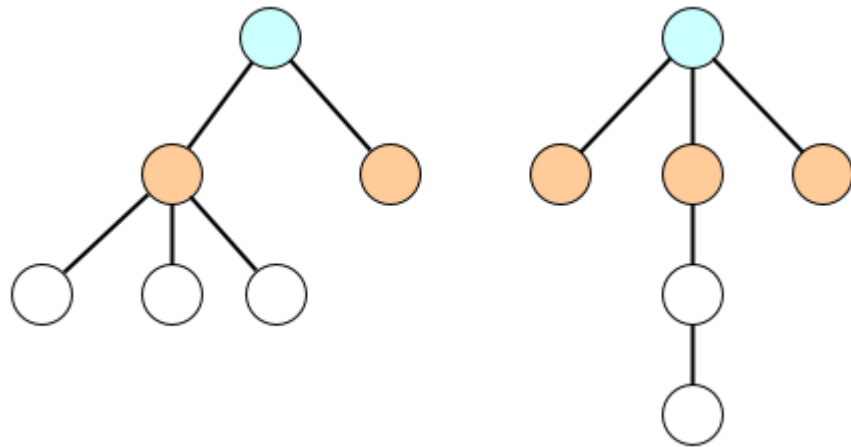
$((()())())((()((()))))()$

- Довольно очевидно, что это лес
- Но такое чувство что это не самое удобное представление для работы
- Проведём стрелки к первому брату и к первому потомку
- И внезапно мы видим что у нас получилась древовидная структура с двумя связями на каждый узел. И так любой лес это...



# Лес это бинарное дерево

- Внезапно это бинарное дерево

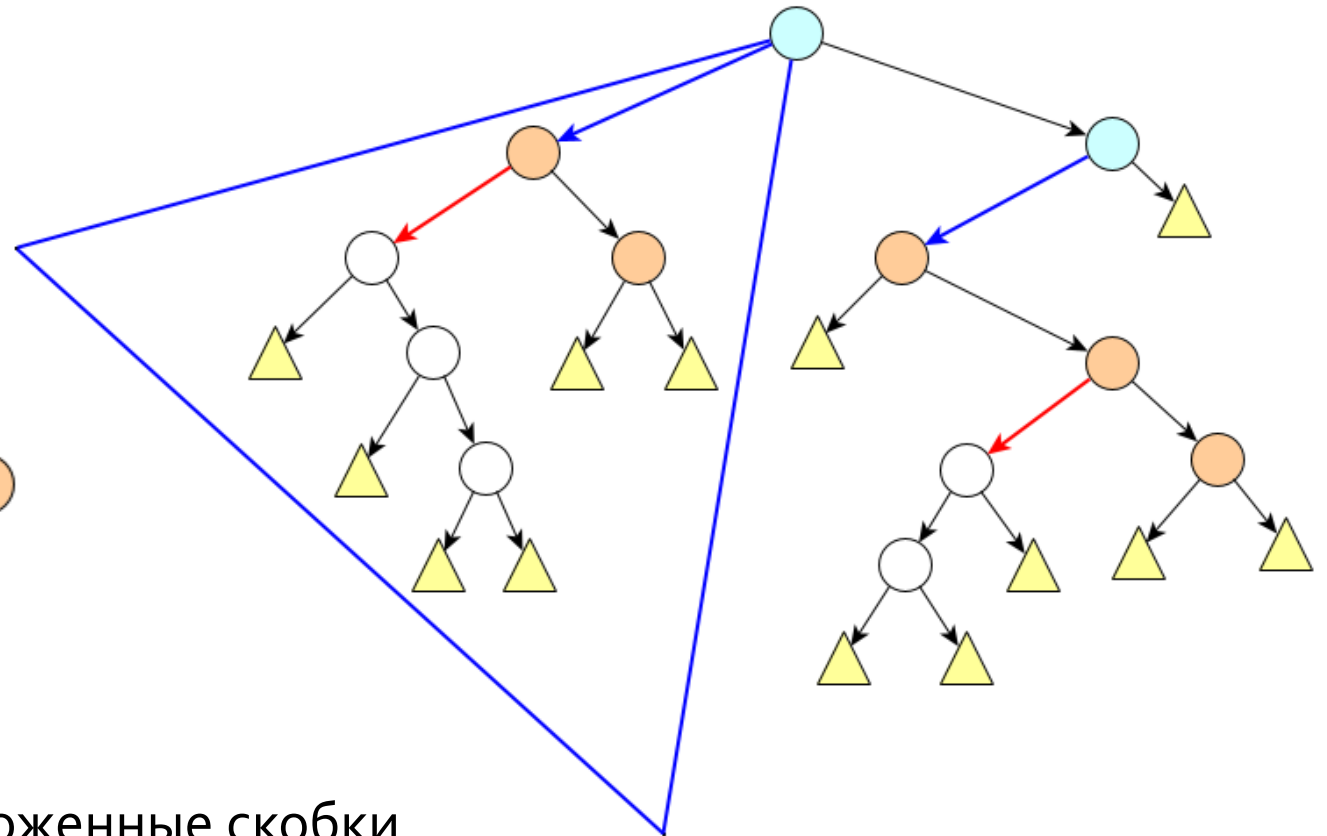
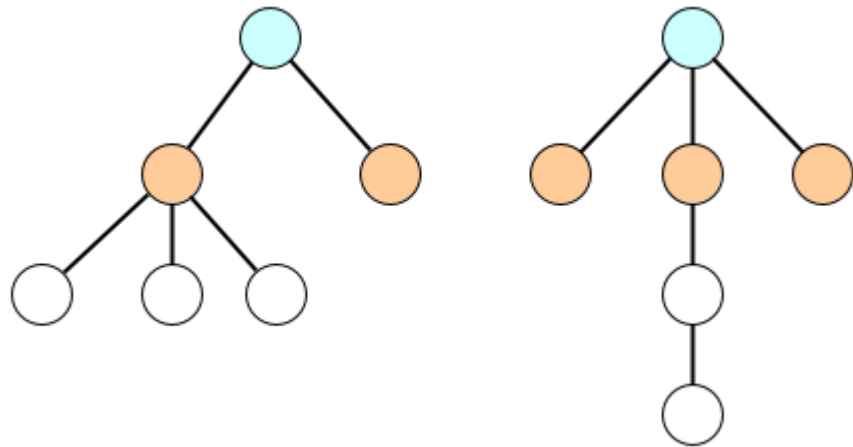


- Кстати, а вы увидели в бинарном дереве последовательность скобок?

# 2-дерево это расстановка скобок

- Внезапно это бинарное дерево

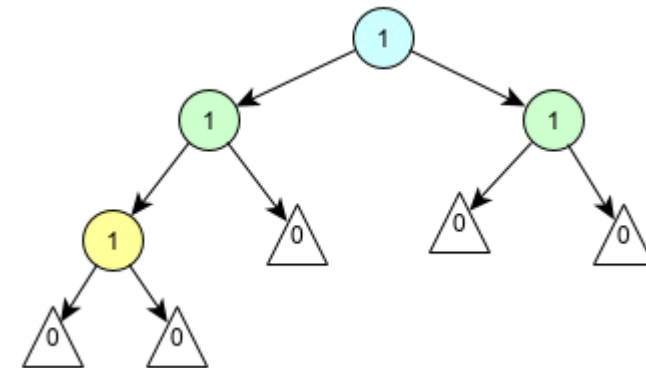
$((()())())((()((()()))))()$



- Если есть левый потомок, это вложенные скобки
- Если есть правый потомок это скобки, стоящие рядом

# Бинарные деревья и 0-1 представление

- Любое бинарное дерево соответствует строке из нулей и единиц.
- 1 1 1 0 0 0 1 0 0
- Очевидно нулей должно быть на один больше, поэтому последний ноль можно сэкономить.
- Приведите пример строки с правильным числом нулей и единиц, которой не соответствует ни одно дерево?



# Представление в текстовом файле

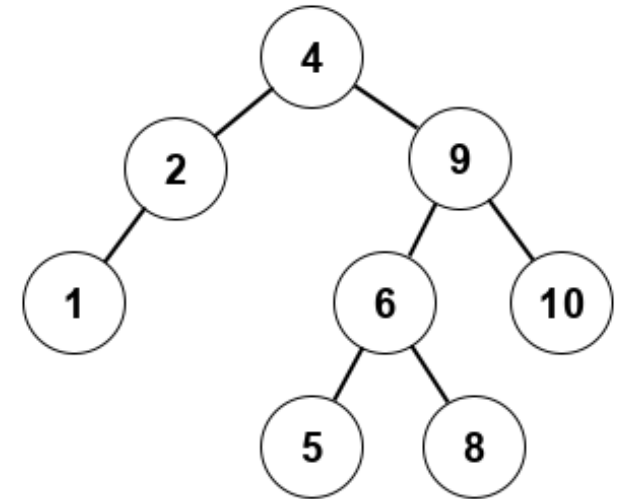
- Естественное представление: хранить топологию отдельно и значения отдельно.

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

- Обратите внимание на картинке справа не нарисованы нулевые листья, но мы всегда помним, что они есть.
- Также при таком представлении мы естественным образом храним preorder обход.



# СЕМИНАР 4.3

---

Хеш-таблицы, структуры данных и многомодульные программы.

# 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

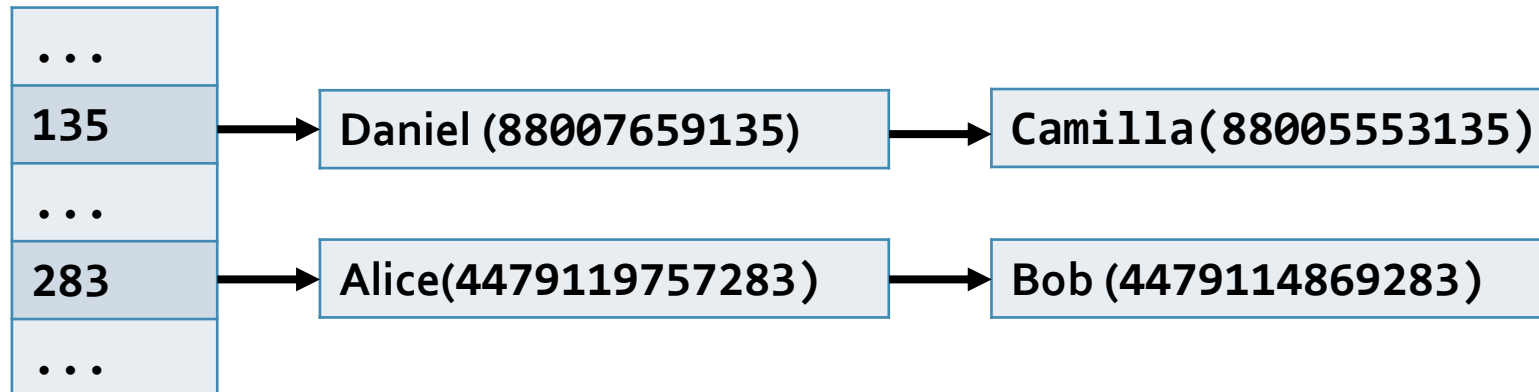
# Обсуждение

- 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

- 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:

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

- Для строк это семейство:

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

# 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.

# 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](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

# Поиск подстроки в строке

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

- Наивный алгоритм: перебирать каждую подстроку с каждой.

```
for (Off = 0; Off < HaystackSize - NeedleSize; ++Off) {  
    int Found = 1;  
    for (Pos = 0; Pos < NeedleSize; ++Pos)  
        if (HayStack[Off + Pos] != Needle[Pos]) {  
            Found = 0; break  
        }  
    if (Found == 1) return Off;  
}
```

# Поиск 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

- 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**

# Алгоритм RQ – поиск диапазона

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

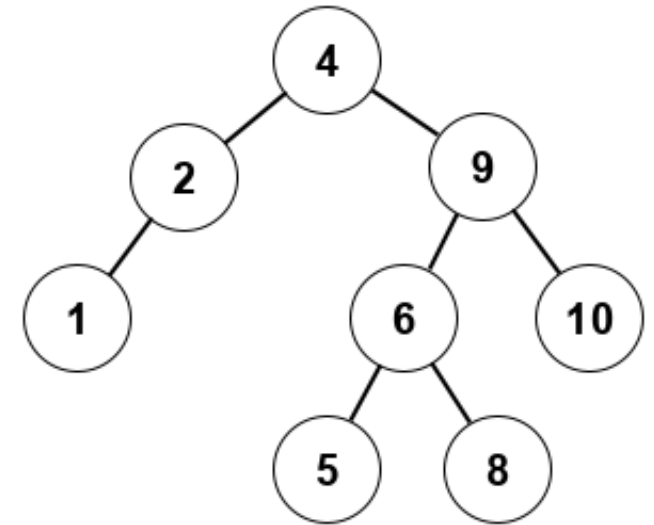
```
typedef void (*visit_t)(const struct node_t *);  
void visit_range(struct node_t *top, int l, int r, visit_t v) {  
    if (NULL == top) return;  
    if (l <= node_data(top) && r >= node_data(top))  
        v(top);  
    if (l <= node_data(top))  
        visit_range(node_left(top), l, r, v); 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

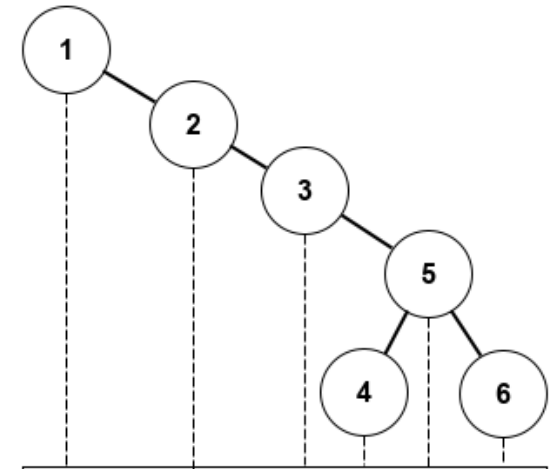
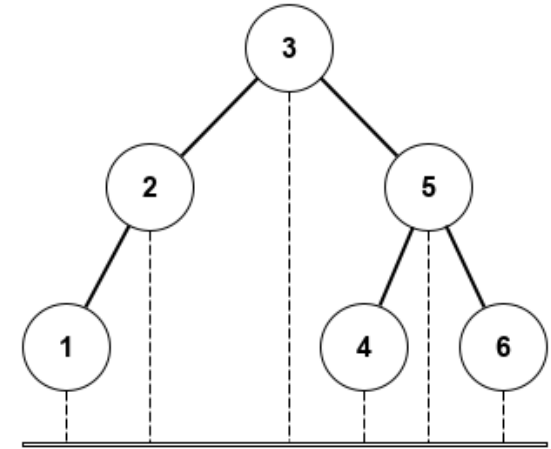


# Problem RBQ: запросы городов

- 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

# Разбалансированность

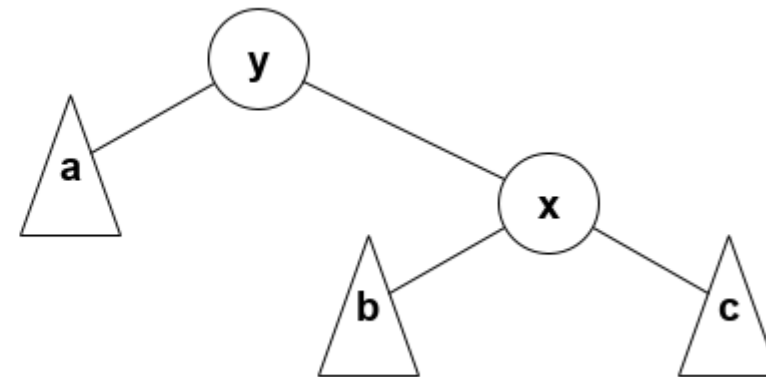
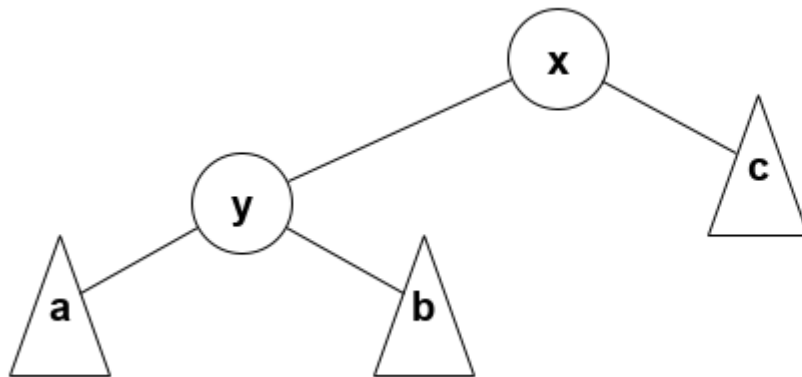
- Существенной проблемой для алгоритма RQ является то, что он логарифмический только в лучшем случае
- Мы можем легко представить плохой случай: при неудачном порядке добавления тех же узлов дерево может не быть сбалансированным
- В этом случае поиск в дереве (даже в поисковом) не лучше, чем поиск в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





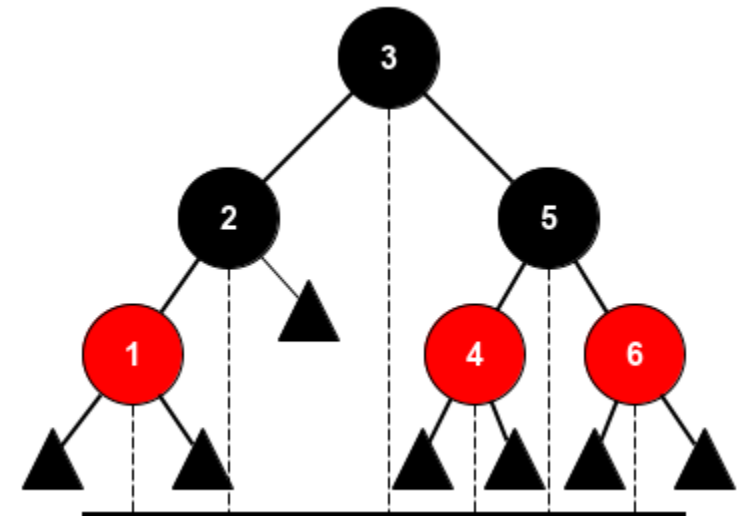
# Поддержка свойств поисковости

- 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

- Структуры данных слишком легко запутать.
- Можно запутать список, хеш-таблицу и даже поисковое дерево.
- Что если в структуре данных появятся лишние связи?
- Например попробуйте прогнать процедуру переворота на зацикленном списке или поиск в дереве на сломанном дереве и посмотрите результат.
- Интересный вопрос: что же делать?

# Обсуждение

- Плохой вариант: при каждом вызове любой функции проверять целостность структуры данных.
- Хороший вариант несколько менее очевиден. И он называется **инкапсуляция**.
- Но прежде давайте немного скажем о многомодульных программах.

# Многомодульные программы

- Предположим, что вы написали очень интересную программу.

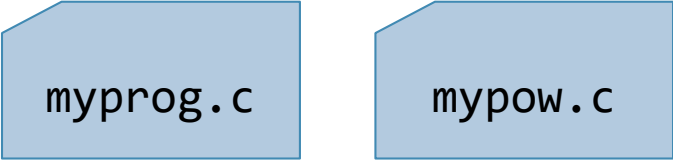
```
// ipow возводит n в степень x
unsigned long long ipow(unsigned n, unsigned x) {
    // тут очень умная реализация
}

int main () {
    // тут основная программа, использующая ipow
}
```

- Она работает, но вы обнаружили, что функция `ipow` может быть вам нужна и в других программах, т.е. может быть **переиспользована**.

# Выносим функцию в модуль

- Вы можете сделать отдельный модуль с функцией `ipow`.
- В модуле `myprog` вам нужно только определение.



`myprog.c`

`mypow.c`

```
> gcc myprog.c mypow.c -o myprog
```

```
// myprog.c
unsigned long long ipow(unsigned n, unsigned x);

int main() {
    // тут основная программа, использующая ipow
}
```

- Это работает. Все ли видят проблемы с таким подходом?

# Заголовочные файлы

- Выход это составить заголовочный файл с определением и включить его и в файл с реализацией и в файл с использованием.

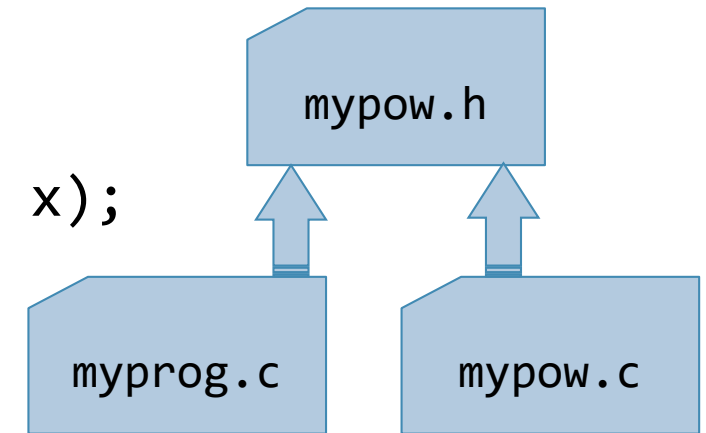
```
// --- файл mypow.h ---
```

```
unsigned long long ipow(unsigned n, unsigned x);
```

- Внутри myprog.c мы воспользуемся текстовым включением.

```
#include <stdio.h>  
#include "mypow.h"
```

- Вид скобок определяет путь поиска файлов: треугольные скобки – файл ищется по системным путям.



```
> gcc myprog.c mypow.c -o myprog
```

# Стражи включения

- Один заголовочный файл может быть включён в тысячи файлов в одном проекте.
- Чтобы избежать лишних включений, можно использовать прагму.

```
// --- файл trow.h ---
```

```
#pragma once
```

```
unsigned ipow(unsigned n, unsigned x);
```

- Позднее мы поговорим о стражах включения подробнее.



# 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?

# 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

# Структура дерева: 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

# 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)

# Обсуждение

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

# 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 Sedgewick Algorithms, 4th edition, 2011
- [WUNI] [en.wikipedia.org/wiki/Universal\\_hashing](https://en.wikipedia.org/wiki/Universal_hashing)

