ЛЕКЦИЯ 07
ДЕРЕВЬЯ

АЛГОРИТМИЗАЦИЯ И
ПРОГРАММИРОВАНИЕ

ЛЕКТОР ФУРМАВНИН С.А.

ПРЕДСТАВЛЕНИЕ БИНАРНОГО ДЕРЕВА

```
template <typename T>
struct Node {
     T val;
     Node *parent; // родитель
     Node *left;   // левый потомок
     Node *right;  // правый потомок
};
```

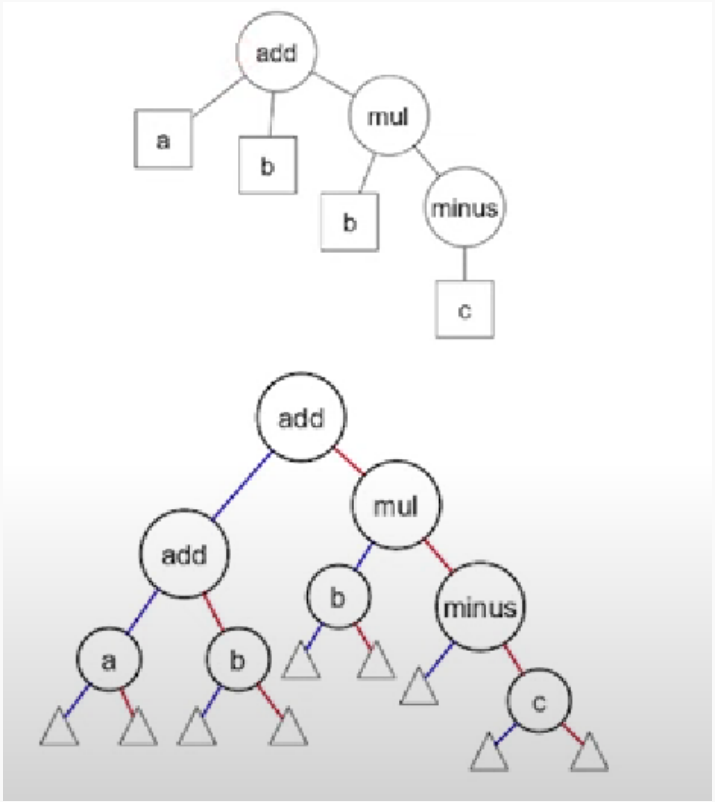
Теперь мы различаем левого и правого потомков

На практике часто требуется **обойти** дерево.

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

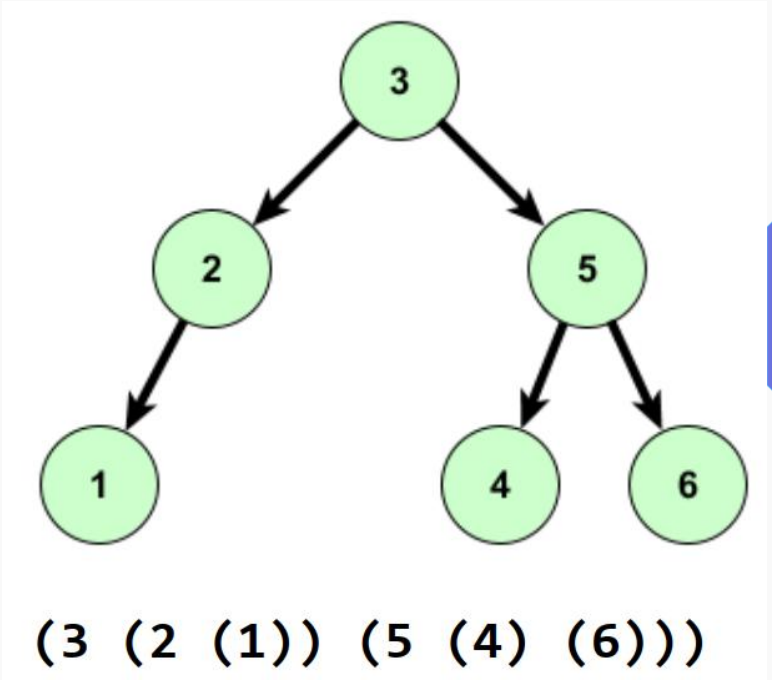


ПРЕДСТАВЛЕНИЕ БИНАРНОГО ДЕРЕВА

Классическая структура данных для бинарного дерева различает левый и правый листья

```
tree_t {
  tree_t *left;
  tree_t *right;
  int data;
};
```

Дополнительно можно хранить указателя на предка



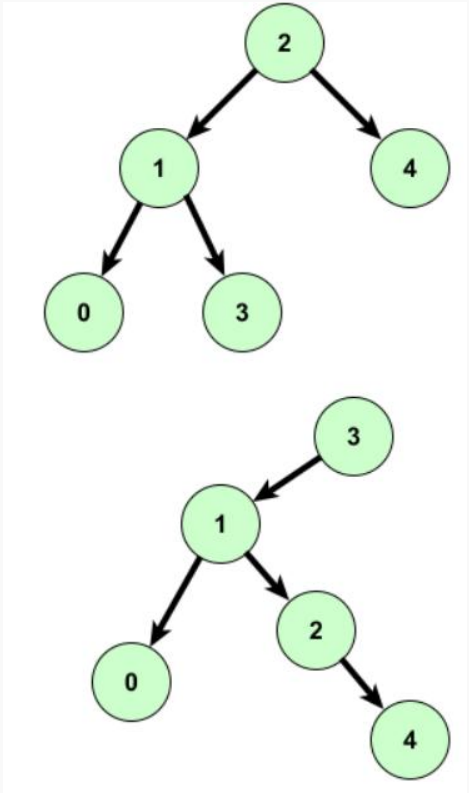
ПРЕДСТАВЛЕНИЕ В ТЕКСТОВОМ ФАЙЛЕ

Самый простой способ: хранить список ориентированных рёбер

key left right

```
5                    5
2   1   4            2 -1   4
1   0   3            1   0   2
4  -1  -1            4  -1  -1
3  -1  -1            3   1  -1
0  -1  -1            0  -1  -1
```

Можно ли упростить это представление?



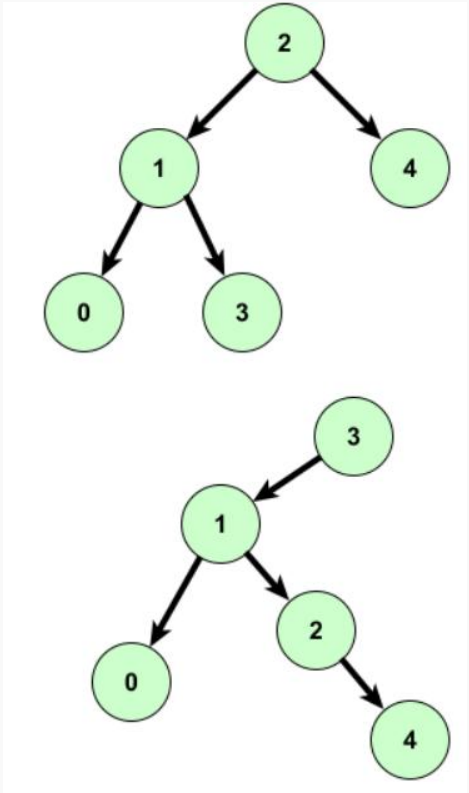
ПРЕДСТАВЛЕНИЕ В ТЕКСТОВОМ ФАЙЛЕ

Самый простой способ: хранить список ориентированных рёбер

key left right

```
5                    5
2   1   4            2  -1   4
1   0   3            1   0   2
                     3   1  -1
```

Да. Например не хранить узлы с обоими пустыми потомками



ПРЕДСТАВЛЕНИЕ БИНАРНОГО ДЕРЕВА

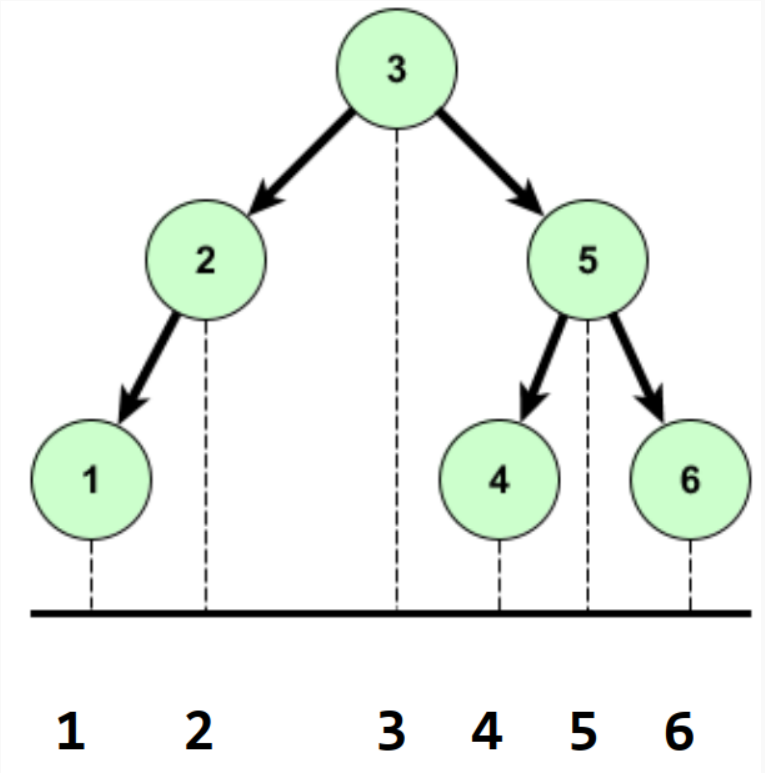
```
template <typename T>
struct Node {
     T val;
     Node *parent; // родитель
     Node *left;   // левый потомок
     Node *right;  // правый потомок
};
```

Теперь мы различаем левого и правого потомков

На практике часто требуется **обойти** дерево.

ОБХОДЫ ДЕРЕВЬЕВ

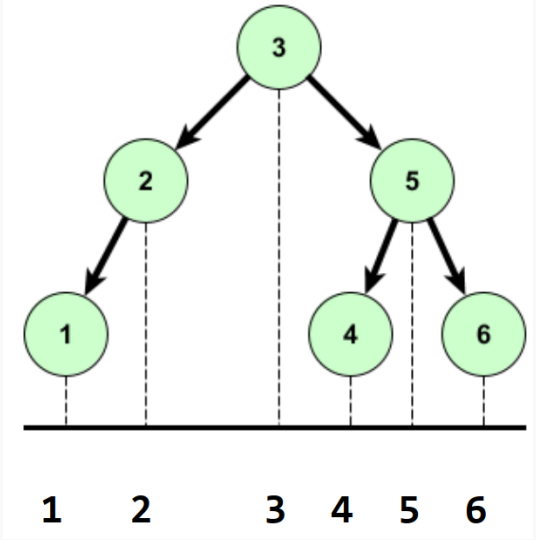
- Обходом (traverse) называется процедура, посещающая все узлы дерева
- Центрированный (inorder) обход: узлы посещаются в порядке проектирования на прямую линию
- Как написать такой обход?



ЦЕНТРИРОВАННЫЙ ОБХОД

Основная идея: для каждого узла сначала обходится левое поддерево, потом посещается сам узел, потом обходится правое поддерево

```
template <typename T>
struct Node { Node *left, *right;
              T data;
static void visit(Node *top);
static void traverse_inorder(Node *top) {
  if (top == NULL) return;
  traverse_inorder(top->left);
  visit(top);
  traverse_inorder(top->right);
}};
```

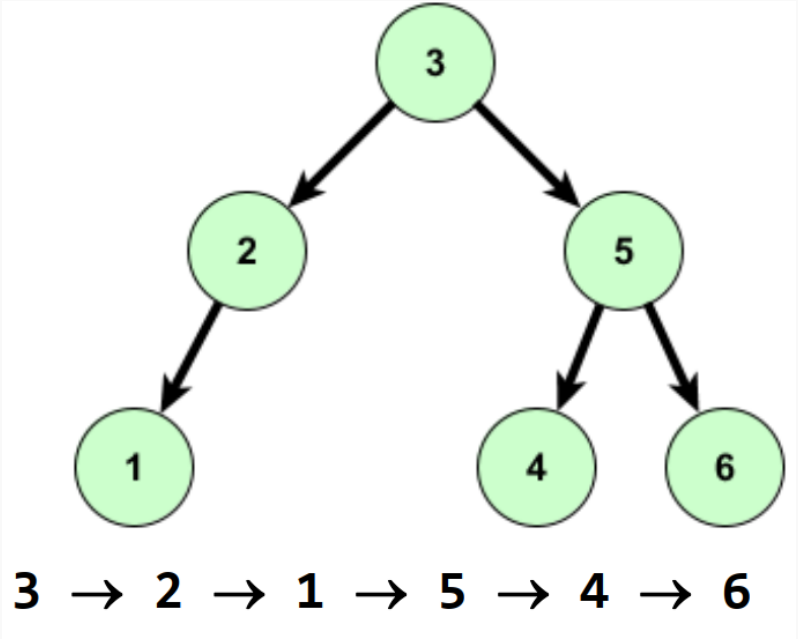


ТОПОЛОГИЧЕСКИЙ ОБХОД

- Пусть дерево это зависимости между делами
- Можно ли обойти дерево так, чтобы никакое дело не было сделано раньше его зависимостей?
- Суть алгоритма

```
traverse(top->left);
visit(top);
traverse(top->right);
```

- Что поменять чтобы получился топологический порядок?

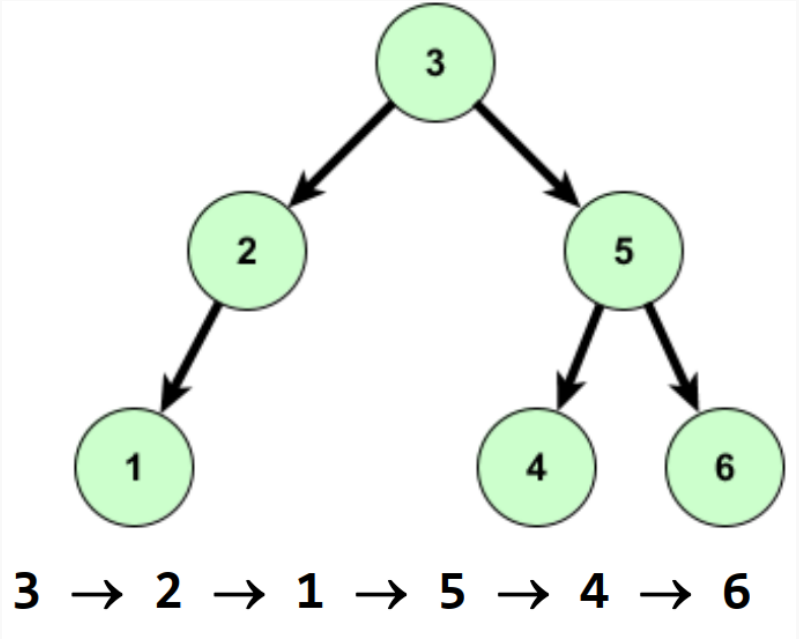


ТОПОЛОГИЧЕСКИЙ ОБХОД

- Пусть дерево это зависимости между делами
- Можно ли обойти дерево так, чтобы никакое дело не было сделано раньше его зависимостей?
- модификация алгоритма

```
visit(top);
traverse(top->left);
traverse(top->right);
```

- Это называется preorder обходом

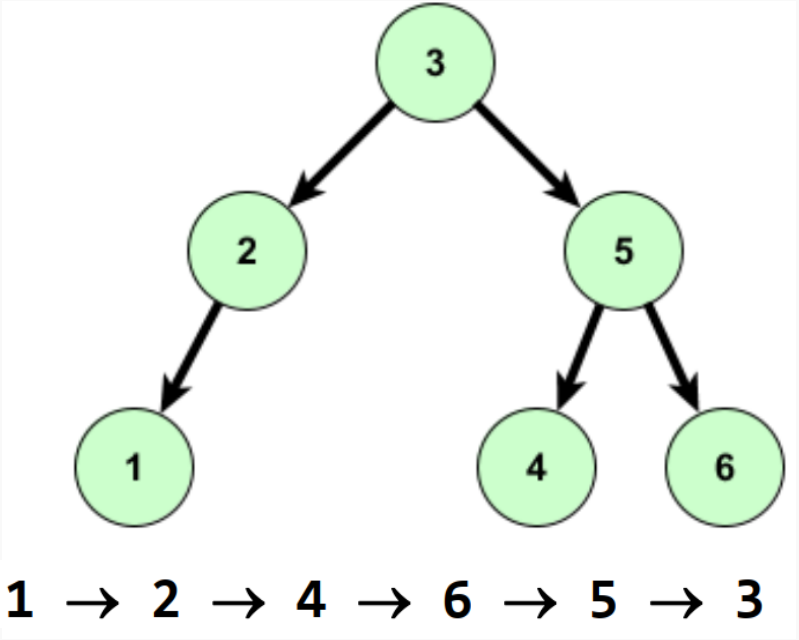


СТЕРЕТЬ ДЕРЕВО

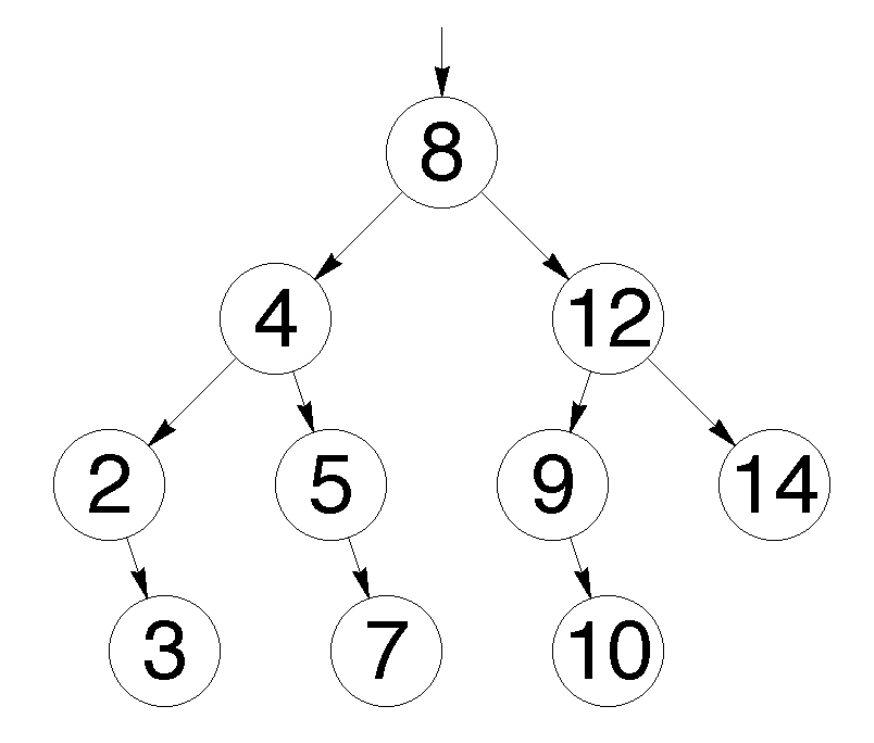
- Пусть теперь хочется стереть дерево
- Можно ли обойти дерево так, чтобы все листы были посещены раньше их предков?
- Модицикация алгоритма

```
traverse(top->left);
traverse(top->right);
visit(top); // delete
```

- Такой «стирающий» обход называется также postorder обходом



ОБХОДЫ БИНАРНОГО ДЕРЕВА

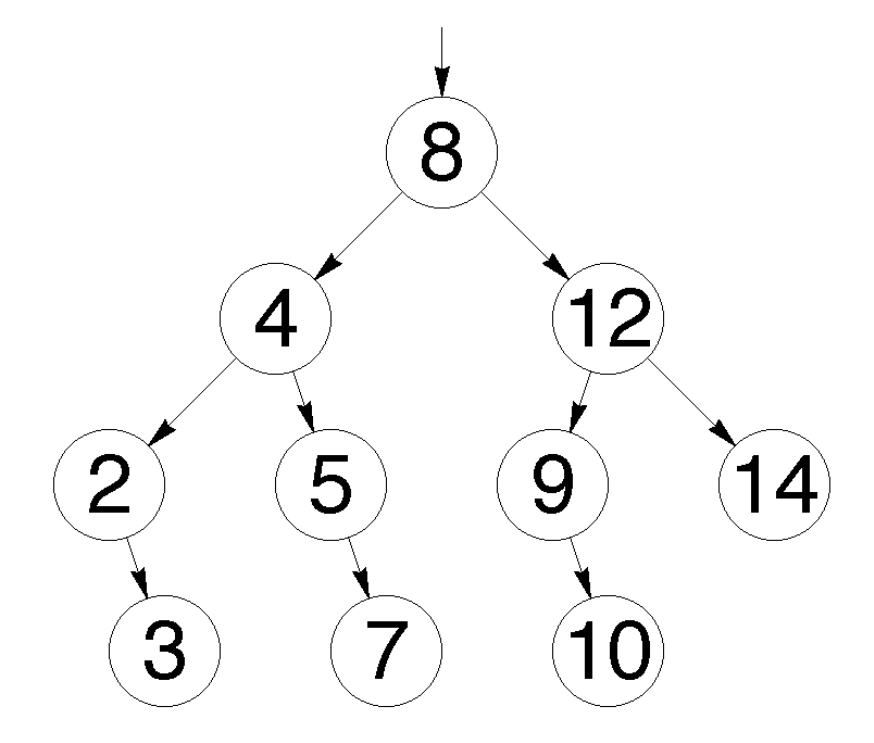


Порядок удаления:

LRV:

RLV postorder

ОБХОДЫ БИНАРНОГО ДЕРЕВА



Порядок удаления:

LRV: 3 2 7 5 4 10 9 14 12 8

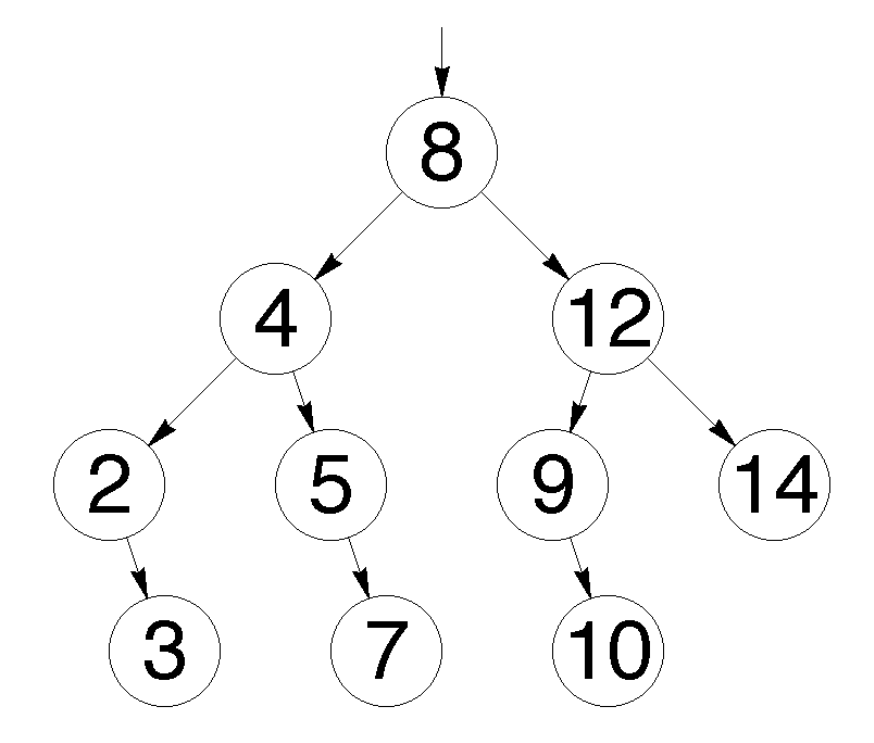
RLV postorder

Порядок проекции

LVR:

inorder

ОБХОДЫ БИНАРНОГО ДЕРЕВА



Порядок удаления:

LRV: 3 2 7 5 4 10 9 14 12 8

RLV postorder

Порядок проекции

LVR: 3 2 4 5 7 8 10 9 12 14

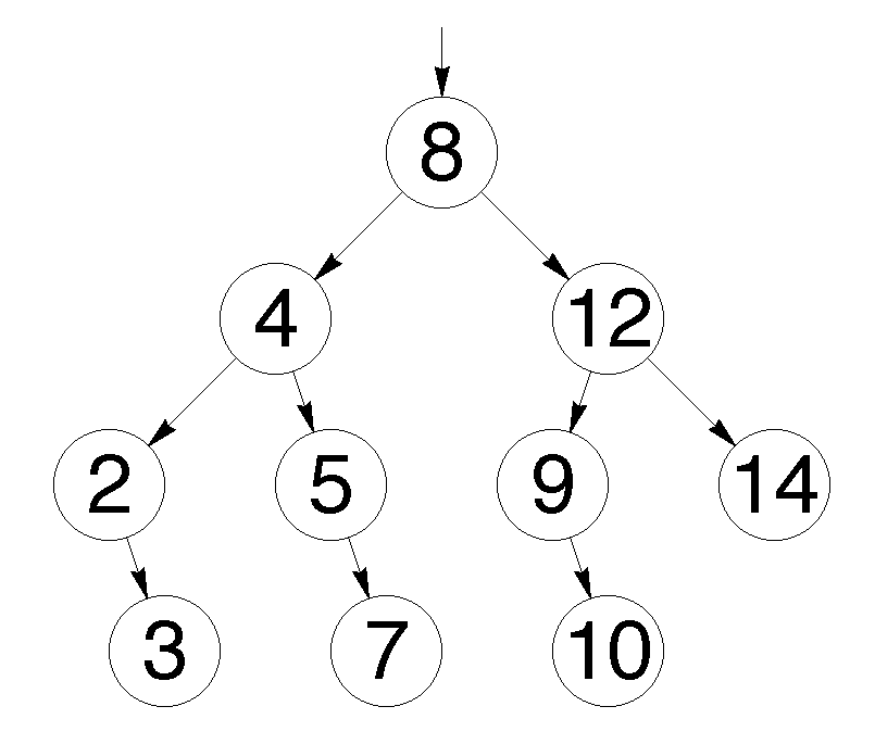
inorder

Топология

VLR:

preorder

ОБХОДЫ БИНАРНОГО ДЕРЕВА



Порядок удаления:

LRV: 3 2 7 5 4 10 9 14 12 8

RLV postorder

Порядок проекции

LVR: 3 2 4 5 7 8 10 9 12 14

inorder

Топология

VLR: 8 4 2 3 5 7 12 9 10 14

preorder

ОБХОДЫ БИНАРНОГО ДЕРЕВА

Inorder: LVR. Используется для линеаризации дерева

Preorder: VLR. Топологический порядок

Postorder: LRV. Порядок удаления дерева

Можно придумать массу других обходов, например обход в ширину, в глубину и так далее.

ЗАДАЧА

На входе имеется указатель на корень дерева.
На выходе получите preorder обход

На входе имеется указатель на корень дерева.
На выходе получите inorder обход

На входе имеется указатель на корень дерева.
На выходе получите postorder обход

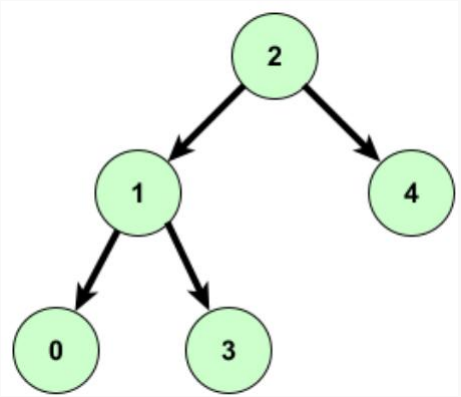
ЗАДАЧА. ДЕРЕВО ИЗ ОБХОДОВ

- На входе файл, содержащий preorder и inorder обходы дерева

```
2 1 0 3 4
0 1 3 2 4
```

- На выходе файл, содержащий дерево

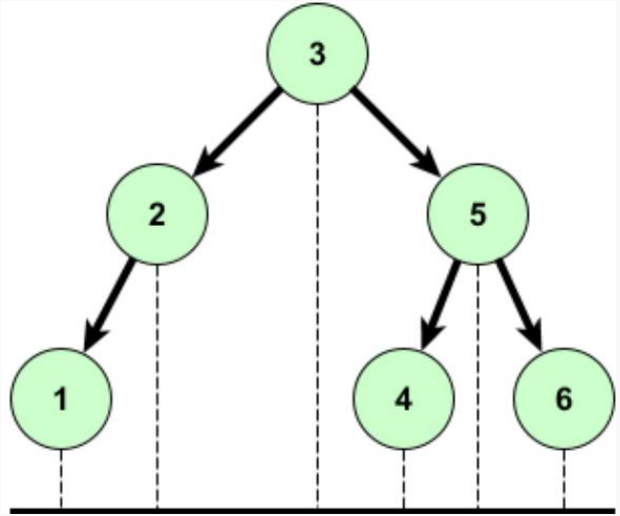
```
5
2 1 4
1 0 3
```



- Теоретическое задание: подумайте хватит ли вам любой пары из трёх обходов?

ПОИСКОВЫЕ ДЕРЕВЬЯ

- Многие деревья, например уже рассмотренное выше, обладают интересным свойством
- Для любого элемента **все элементы в левом поддереве меньше него**, а **в правом все элементы больше него**
- Такие деревья называются **поисковыми**
- Центрированный обход поискового дерева даёт его узлы в сортированном порядке



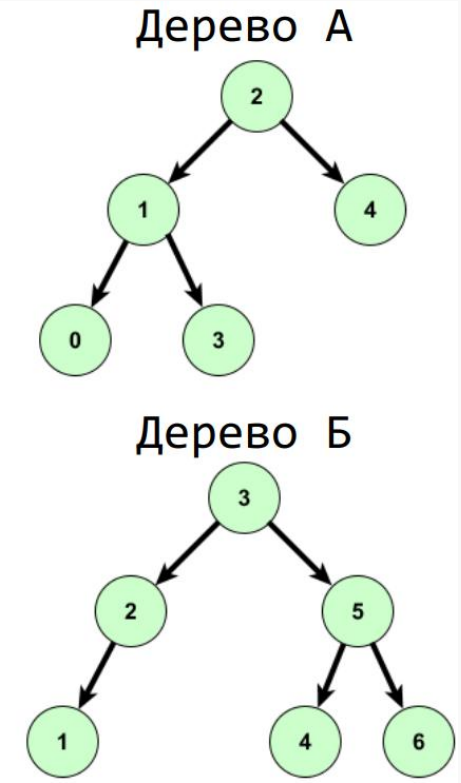
ОБСУЖДЕНИЕ

- Поисковым бинарным деревом называется дерево, для которого:

$$x \epsilon B_L \leftrightarrow x < root(B)$$

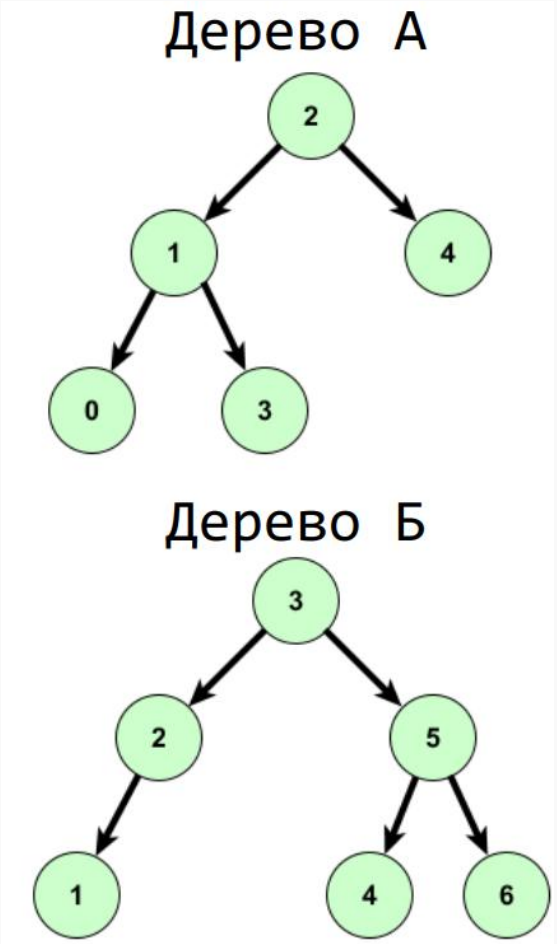
$$x \epsilon B_R \leftrightarrow x > root(B)$$

- Одно (обычно первое) из этих соотношений может быть не строгим
- Как вы думаете, почему такие деревья называют поисковыми?
- Оба ли дерева справа поисковые?



ОБСУЖДЕНИЕ

- Поисковые деревья называются поисковыми, потому что в них легко искать элемент
- Дерево А не является поисковым и чтобы найти в нём тройку надо просмотреть все узлы потратив $O(N)$ времени
- Дерево Б является поисковым и чтобы найти в нём четвёрку, мы идём от тройки направо, потом от пятёрки налево. На поиск любого элемента тратится $O(logN)$ времени, что гораздо лучше
- Кроме того в поисковых деревьях легко делать range queries



ЗАДАЧА. ПРОВЕРИТЬ ПОИСКОВОСТЬ

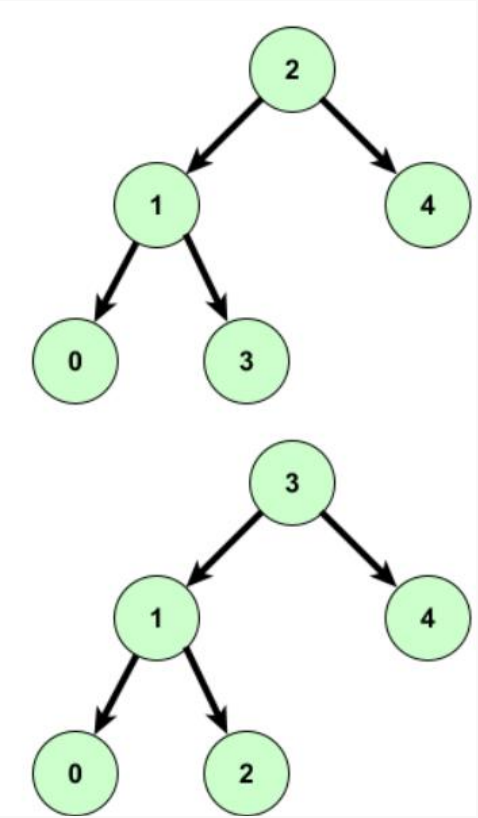
- На входе файл содержащий дерево в обычном формате

5 2 1 4 1 0 3

- На выходе true если дерево поисковое и false если нет
- В данном случае ответ false потому что узел 3 находится в левом поддереве узла 2
- Но для дерева:

5 3 1 4 1 0 2

- Ответ: true, так как это дерево поисковое



BST

Бинарное дерево поиска (англ. binary search tree, BST) — дерево, для которого выполняются следующие свойства:

- У каждой вершины не более двух детей.
- Все вершины обладают ключами, на которых определена операция сравнения (например, целые числа или строки).
- У всех вершин левого поддерева вершины v ключи не больше, чем ключ v
- У всех вершин правого поддерева вершины v ключи больше, чем ключ v
- Оба поддерева — левое и правое — являются двоичными деревьями поиска.

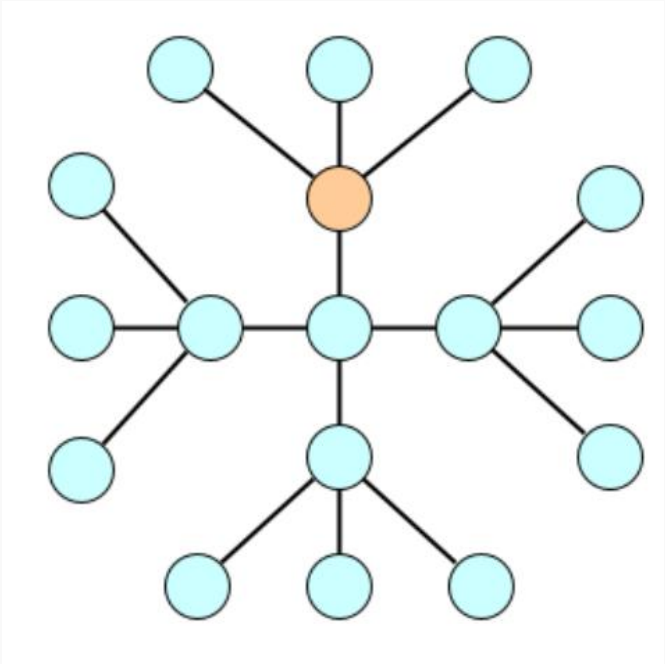
BST

Более общим понятием являются обычные (не бинарные) деревья поиска — в них количество детей может быть больше двух, и при этом в «более левых» поддеревьях ключи должны быть меньше, чем «более правых». Пока что мы сконцентрируемся только на двоичных, потому что они проще.

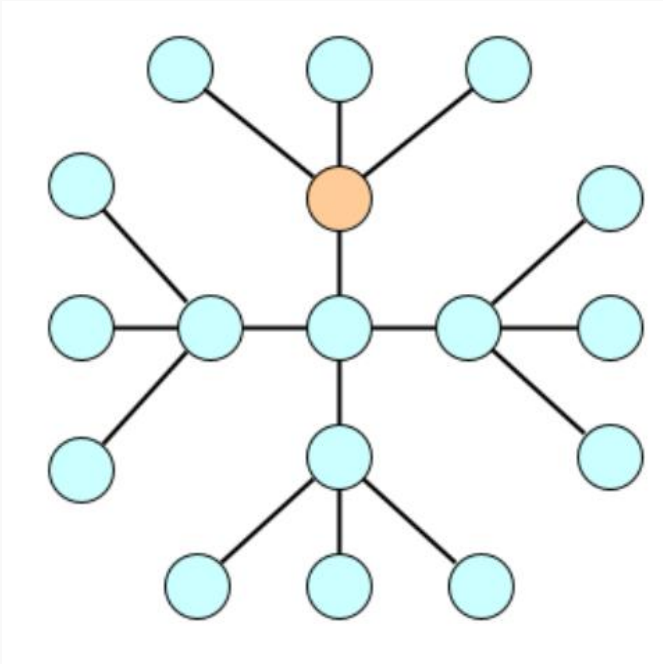
ВЕТВИ НАДО МНОЙ УХОДЯТ В ТЕМНОТУ

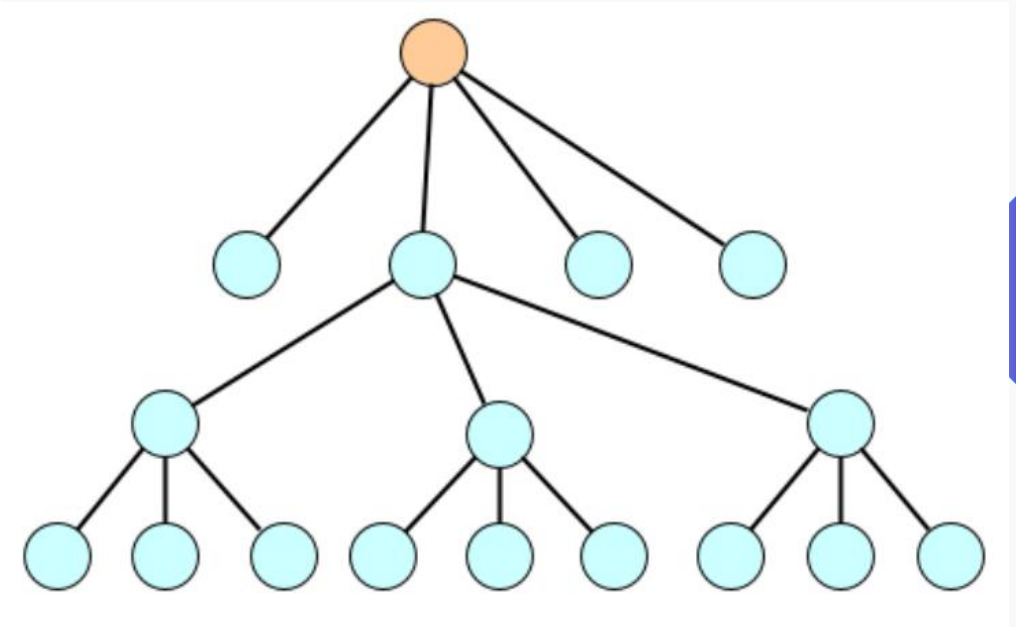
- Дополнительная и очень важная тема это **повороты и балансировка поисковых деревьев**
- Крайне интересны многомерные разновидности поисковых деревьев
- Кроме поисковых деревьев популярностью в компьютерных науках пользуются кучи (heaps) и лучи (tries)
- Суффиксные деревья играют критическую роль в биоинформатике
- Деревья и списки вместе с хеш-функциями используются в технологиях блокчейна
- Увы, все эти прекрасные темы невозможно уложить в этот курс и нам надо двигаться дальше

ЧТО ПЕРЕД ВАМИ?

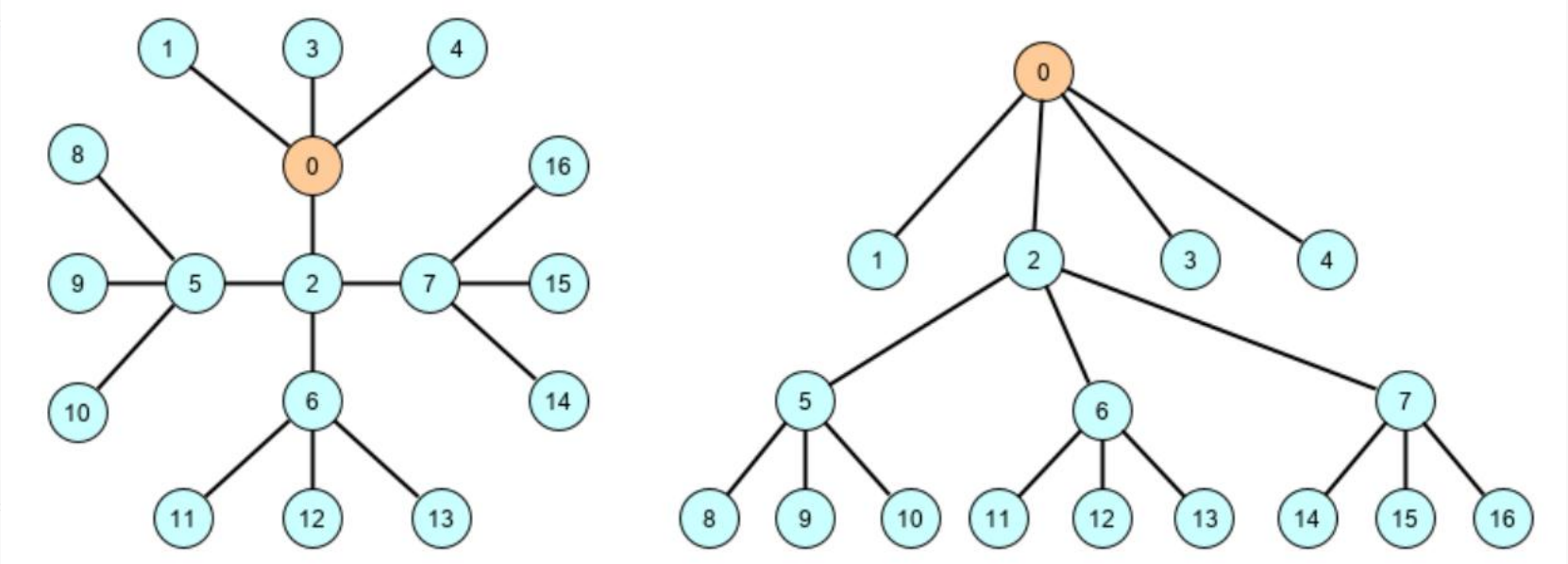


ЧТО ПЕРЕД ВАМИ?



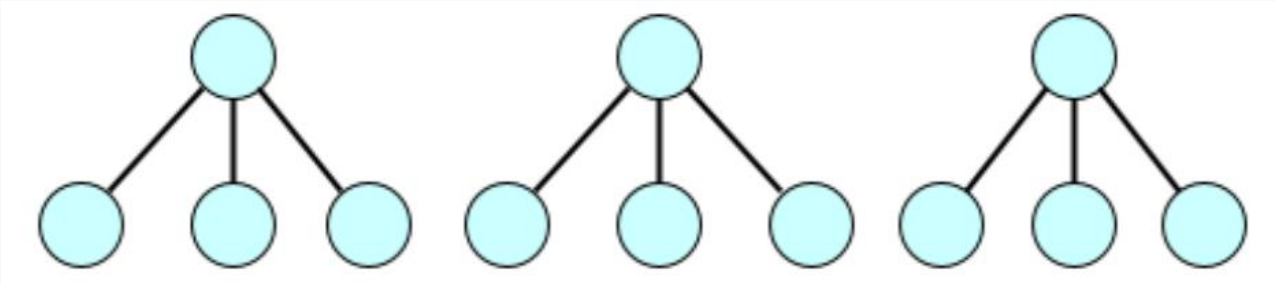


ЧТО ПЕРЕД ВАМИ?



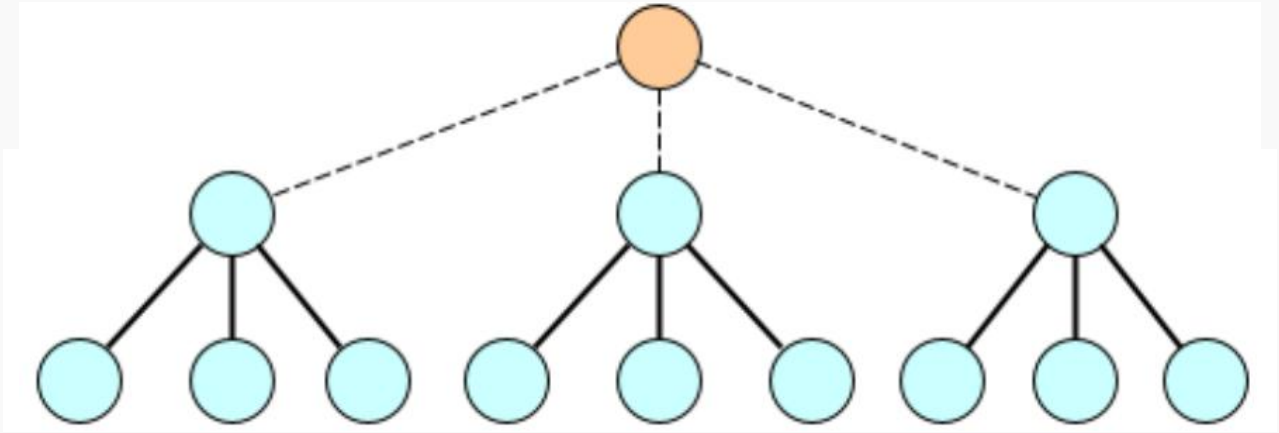
НЕМНОГО О ЛЕСАХ

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



НЕМНОГО О ЛЕСАХ

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

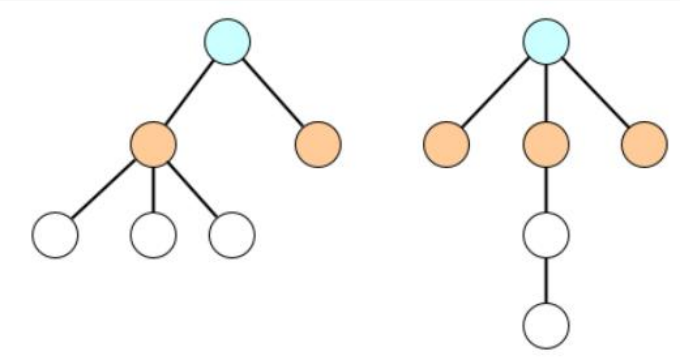


СКОБОЧНОЕ ВЫРАЖЕНИЕ ЭТО ЛЕС

- Рассмотрим правильную расстановку скобок

((()()())()())((()(()())())())

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

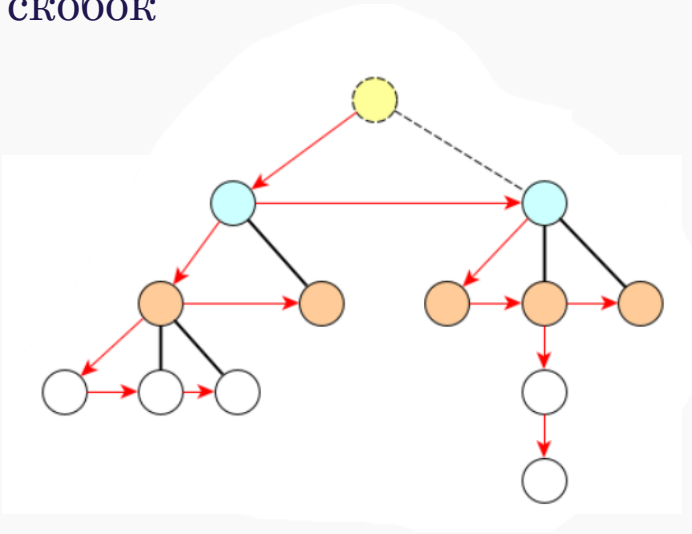


СКОБОЧНОЕ ВЫРАЖЕНИЕ ЭТО ЛЕС

- 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...

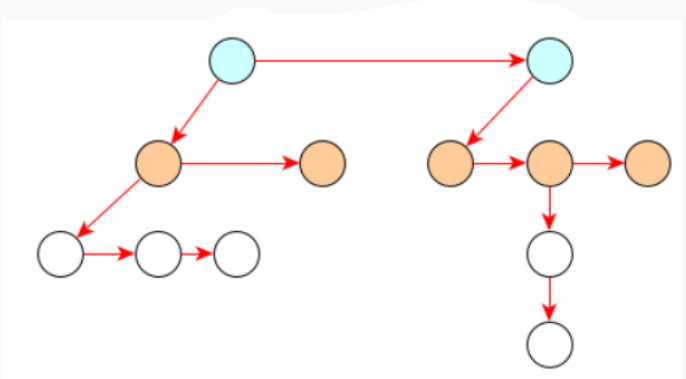


СКОБОЧНОЕ ВЫРАЖЕНИЕ ЭТО ЛЕС

- 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

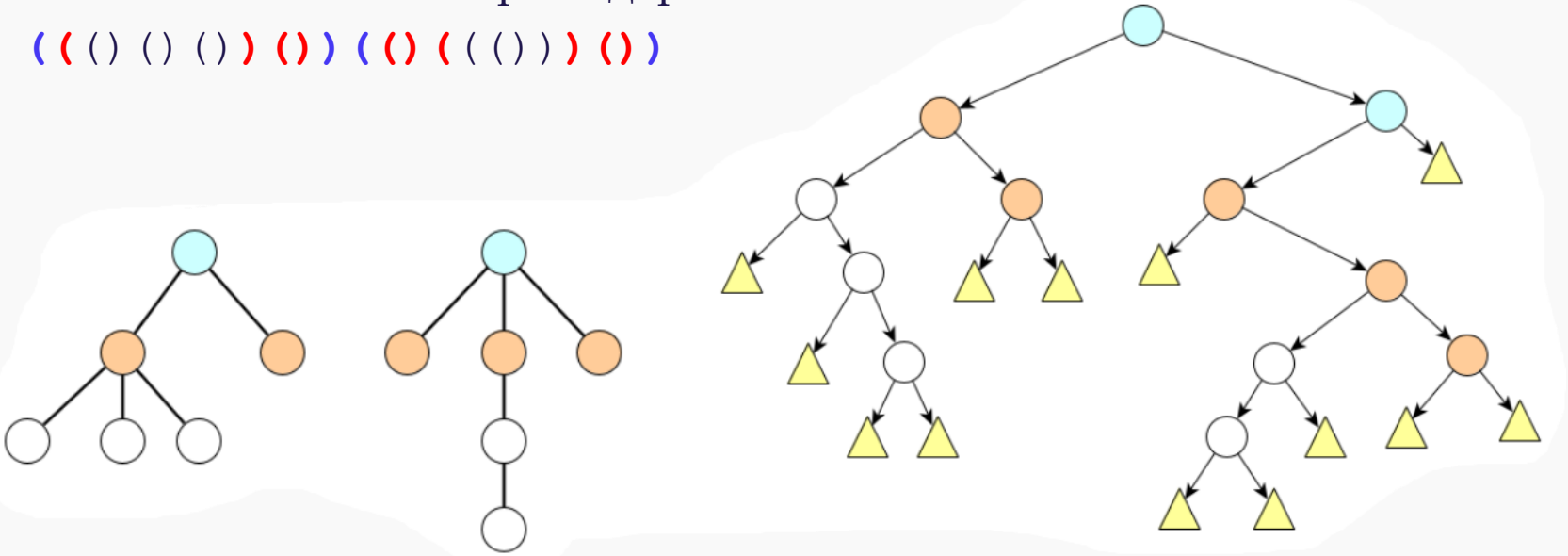


- И внезапно мы видим что у нас получилась древовидная структура с двумя связями на каждый узел
- Итак любой лес (а также любая перестановка, сочетание и разбиение) это...

ЛЕС ЭТО БИНАРНОЕ ДЕРЕВО

- Внезапно это бинарное дерево

((()()())())((()((())))())

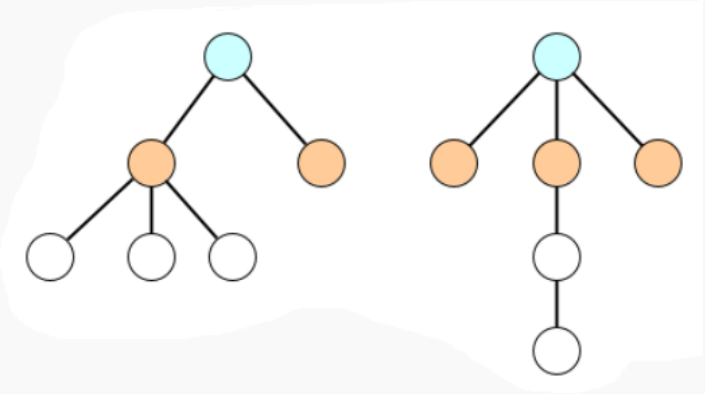


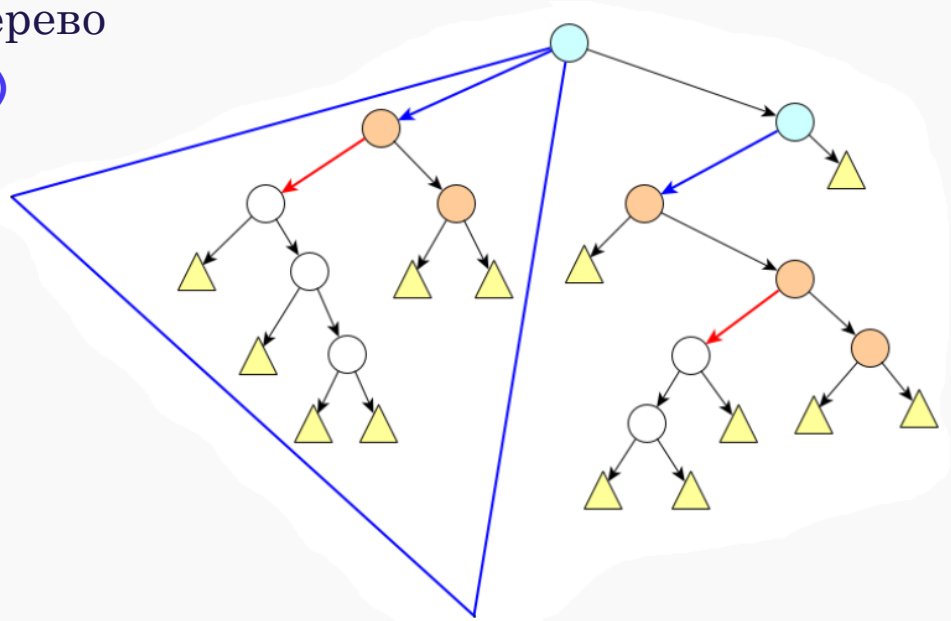
- Кстати, а вы увидели в бинарном дереве последовательность скобок?

ЛЕС ЭТО БИНАРНОЕ ДЕРЕВО

- Внезапно это бинарное дерево

((()()())())((()((()))())())



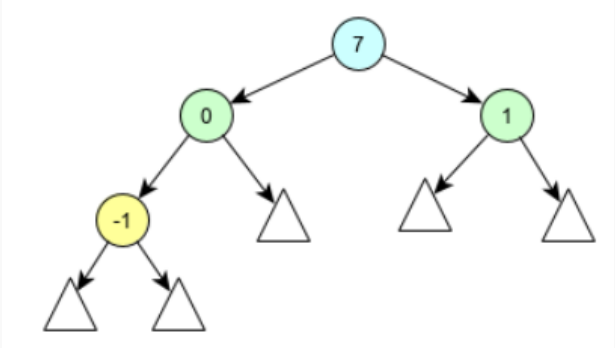


- Если есть левый потомок, это вложенные скобки
- Если есть правый потомок это скобки, стоящие рядом

УПРАЖНЕНИЕ

- Распечатайте простое дерево иерархически

```
7
..0
....-1
..1
```



- Распечатайте дерево как список рёбер

```
7 0 7 1 0 -1
```

- Распечатайте бинарное дерево как скобочное выражение:

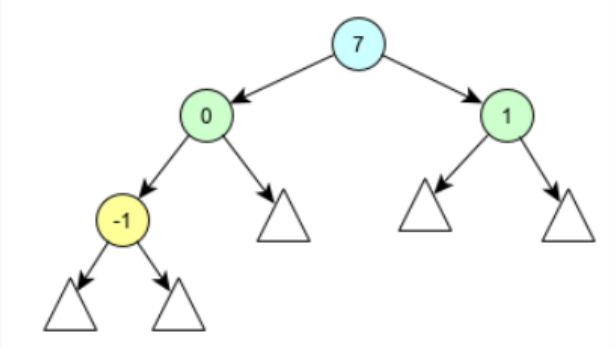
```
7(0(-1()))1()
```

- Хорошая ли идея **хранить** деревья в файле любым из этих способов?

БИНАРНЫЕ ДЕРЕВЬЯ И 0-1 ПРЕДСТАВЛЕНИЕ

- Любое бинарное дерево соответствует бинарной строке из нулей и единиц

1 1 1 0 0 0 1 0 0

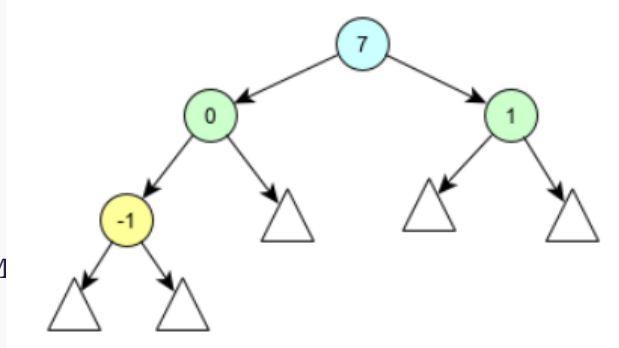


- Очевидно нулей должно быть на один больше, но это ещё не всё
- Приведите пример строки с правильным числом нулей и единиц, которой не соответствует ни одно дерево?
- Можем ли мы расширить это представление, чтобы хранить данные в узлах?
- Что если любое целое число это валидные данные?

БИНАРНЫЕ ДЕРЕВЬЯ И 0-1 ПРЕДСТАВЛЕНИЕ

- Основная идея: два раздельных потока для структуры и для ключей

```
1 1 1 0 0 0 1 0 0
7 0 -1 1
```



- Если мы считываем 1, считываем ключ
- Поскольку 63 = 31 + 32, мы можем закодировать дерево из 31 узла в 64-битном числе
- Если мы будем хранить там `int`, мы потратим 124 байта на ключи и только 8 байт на структуру дерева

УПРАЖНЕНИЕ: ДЕРЕВЬЯ И РЕКУРСИЯ

- Считайте 0-1 бинарное дерево из файла

11001110110010101100101101000000

- Корень находится на нулевом уровне
- Сколько узлов в нём находится на нечётных уровнях?

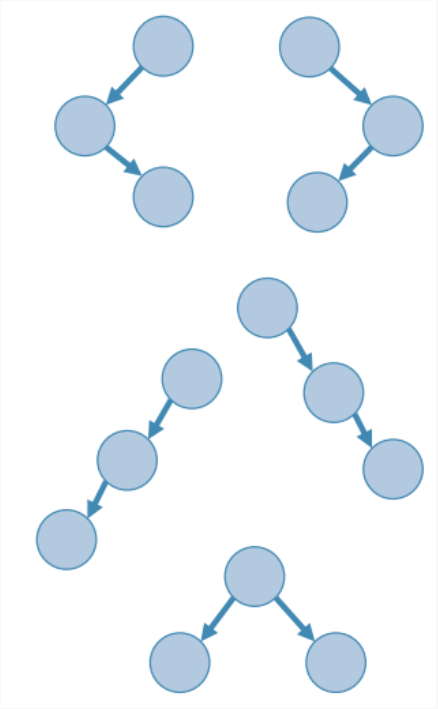
```
void count_odd(BNode_t *cur, int *oddcnt, int lvl) {
  *oddcnt += ((lvl % 2) != 0);
  if (cur->left) count_odd(cur->left, oddcnt, lvl + 1);
  if (cur->right) count_odd(cur->right, oddcnt, lvl + 1);
}
```

ДОМАШНЕЕ ЗАДАНИЕ

- Сгенерируйте N единиц и $N + 1$ нулей
- Случайно их перемешайте посредством алгоритма Fisher Yates shuffle
- Вы получили равномерно распределённое число из $2N + 1$ бит
- Утверждается что всегда можно найти единственный циклический поворот, такой, чтобы это число стало возможным деревом
- Пример: `0 0 1 0 0 1 1 1 0 -> 1 1 1 0 0 0 1 0 0`
- Найдите такой поворот и распечатайте получившееся дерево (увы такие деревья не будут в точности равномерно распределены)

ДОМАШНЕЕ ЗАДАНИЕ

- Погуглите алгоритмы прежде чем решать эти задачи, но, пожалуйста, не гуглите реализацию, потренируйтесь самостоятельно
- Посетите все различные бинарные деревья с N внутренними узлами, не посещая ни одно из них дважды. Проверьте себя:
    - для трёх узлов ваша программа должна построить пять различных бинарных деревьев, см. картинку справа
    - для 11 узлов ваша программа должна построить 58786 различных бинарных деревьев
- Сгенерируйте равновероятное случайное дерево с N внутренними узлами



РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Бьерн Страуструп, Язык программирования C++/ ред. А. Боборыкин. – 4-е изд. - Москва: Издательство БИНОМ, 2023. – 1213 с.
2. Дональд Кнут Искусство программирования, том 1. Основные алгоритмы = The Art of Computer Programming, vol.1. Fundamental Algorithms. — 3-е изд. — М .: «Вильямс» , 2006. – 720 с.
3. Дональд Кнут Искусство программирования, том 2. Получисленные алгоритмы = The Art of Computer Programming, vol.2. Seminumerical Algorithms. — 3-е изд. — М .: «Вильямс» , 2007. — 832 с.
4. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.: МЦМНО, 1999. – 960 с.
5. Скотт Мейерс, Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ / ред. Д.А. Мовчан – 3-е изд. – Москва: ДМК Пресс, 2017. – 300 с.
6. Скотт Мейерс, Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов / ред. Д.А. Мовчан – 3-е изд. – Москва: ДМК Пресс, 2016. – 298 с.