Лекция 12.
Верёвочные строки

Даниил Михайлович Берлизов
Старший преподаватель Кафедры вычислительных систем СибГУТИ
E-mail: sillyhat34@gmail.com

Курс «Структуры и алгоритмы обработки данных»
Осенний семестр, 2021 г.

Строковый тип данных

- В большинстве языков программирования для выполнения операций над последовательностями символов определён **строковый тип**

- **Строка** (*string*) — тип данных, значением которого является произвольная последовательность символов алфавита

“Based_and_redpilled\0”

Строковый тип данных

Основные операции, определённые над строками:

- Получение символа по индексу (номеру позиции)
- Конкатенация (сложение) строк*
- Разбиение строки на две подстроки
- Поиск подстроки в строке: проверка вхождения одной строки в другую
- Вставка и удаление заданной подстроки
- Получение длины строки

* Конкатенация может выполняться **деструктивно** (*destructive*), с перезаписью исходных строк, и **недеструктивно** (*non-destructive*)

Строковый тип данных

Как правило, строковый тип представлен массивом символов (типа данных *char*) фиксированной длины. Такой подход приводит к следующим ограничениям:

- Неэффективные конкатенация (сложение) строк и операции с подстроками
- Сложность поддержки неизменяемости строк (*immutable strings*): строки, передаваемые в сторонние функции, должны быть защищены от случайной перезаписи
- Основные операции над строками плохо масштабируются: работа с длинными строками занимает существенно больше времени, нежели с короткими
- Функции для работы со строками не способны работать с другими последовательностями символов (например, с файлами)

Rope

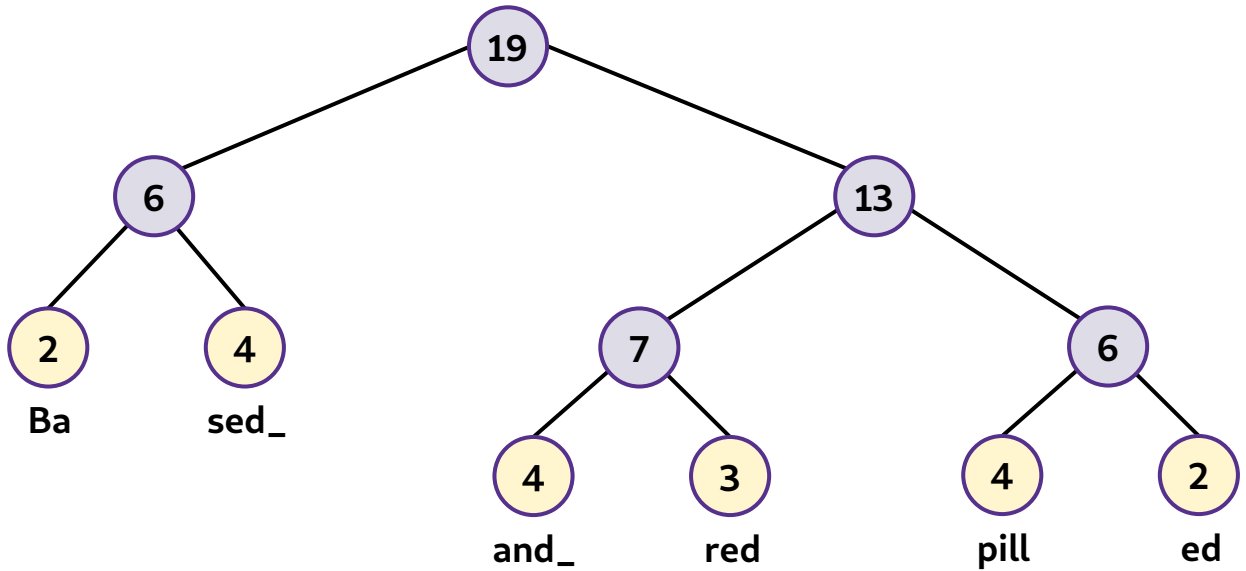
- **Верёвочная строка**[*] (*rope, cord*) — структура для хранения строкового типа данных, альтернатива массивам символов фиксированной длины
- Структура данных rope — это двоичное дерево поиска, в котором:
  - каждый узел представляет собой конкатенацию своих дочерних узлов
  - листьями узла являются обычные строки, представленные массивами символов
- Использование rope позволяет выполнять операции вставки, удаления и конкатенации над строками за время $O(\log n)$

- **Авторы:** H. J. Boehm, R. Atkinson, M. Plass (1995)

[*] Boehm H. J., Atkinson R., Plass M. **Ropes: an alternative to strings** // Software: Practice and Experience. – 1995. – Т. 25. – №. 12. – С. 1315-1330.

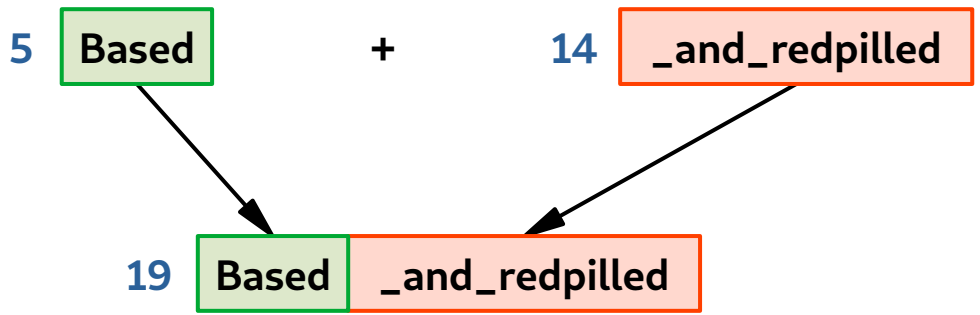
Структура узлов rope

- Каждый листовой узел rope содержит строку *S*, представленную массивом символов, и её длину
- Внутренние узлы rope представляют собой конкатенацию своих дочерних узлов и содержат вес *w* — сумму длин всех листовых узлов своего поддерева



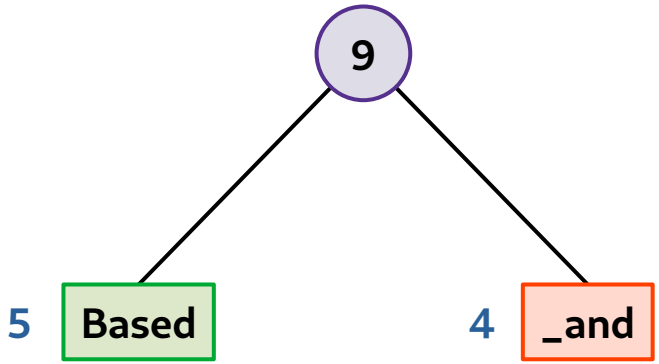
Конкатенация строк: подход на основе массивов

- **Дано:** строки S_1 и S_2, требуется соединить их и записать в строку S
- Выделяем память под строку S необходимой длины
- Поочерёдно обходим строки S_1 и S_2, записываем их элементы в S
- **Сложность операции:** $O(n)$



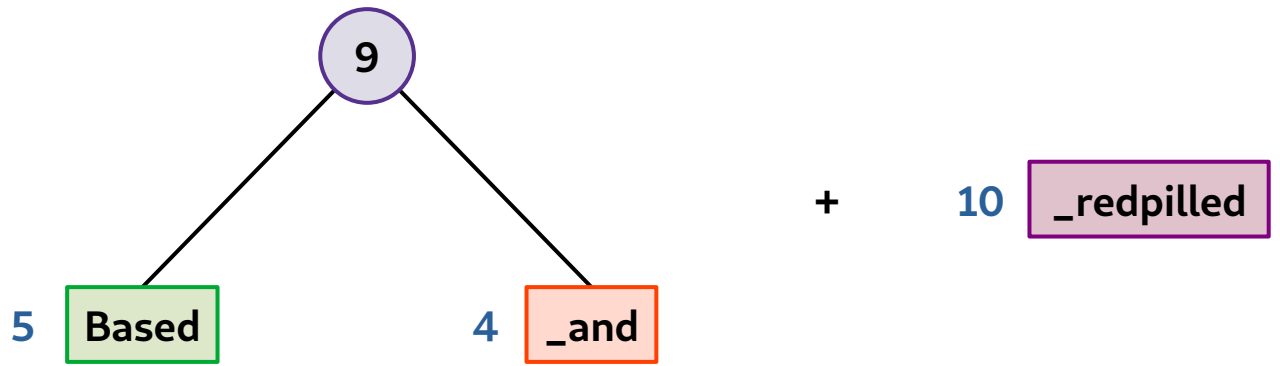
Конкатенация строк: подход на основе rope

- **Дано:** строки S_1 и S_2, требуется соединить их и записать в строку S
- Создаём «обёртку» для строк S_1 и S_2: новый корневой узел дерева
- S_1 становится левым потомком нового корня, S_2 — правым, в узел записывается сумма их длин
- **Сложность операции:** $O(1)$



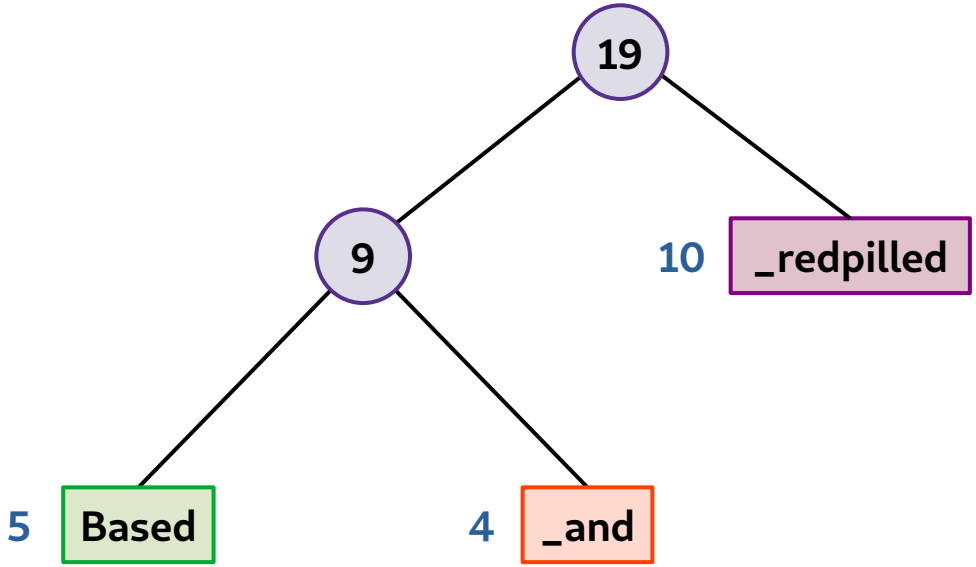
Конкатенация строк: подход на основе rope

- Как произвести конкатенацию полученной строки с другой строкой?



Конкатенация строк: подход на основе rope

- **Как произвести конкатенацию полученной строки с другой строкой?**
- Объект, полученный после соединения двух первых строк, также может рассматриваться как строка: производим аналогичные действия



Конкатенация строк

```
function RopeConcatenate(rope1, rope2, &rope)
    rope = RopeCreateNode()
    rope.left = rope1
    rope.right = rope2
    rope.weight = rope1.weight + rope2.weight
end function
```

$T_{Concatenate} = O(1)$

Получение символа по индексу

Для поиска символа в строке *S* по индексу *i* производится рекурсивный обход дерева

- Если текущий узел является внутренним (у него есть хотя бы один потомок):
  - Если вес *w* его левого поддерева больше *i*, переходим к его левому поддереву
  - Иначе производим поиск индекса *i* – *left.w* в правом поддереве (*left.w* — вес левого поддерева)
- Если текущий узел — лист, то искомый символ находится по индексу *i* в строке, которую он содержит

- Сложность этой операции составляет ***O*(log*n*)**: требуется спуск по одной из ветвей дерева до листа

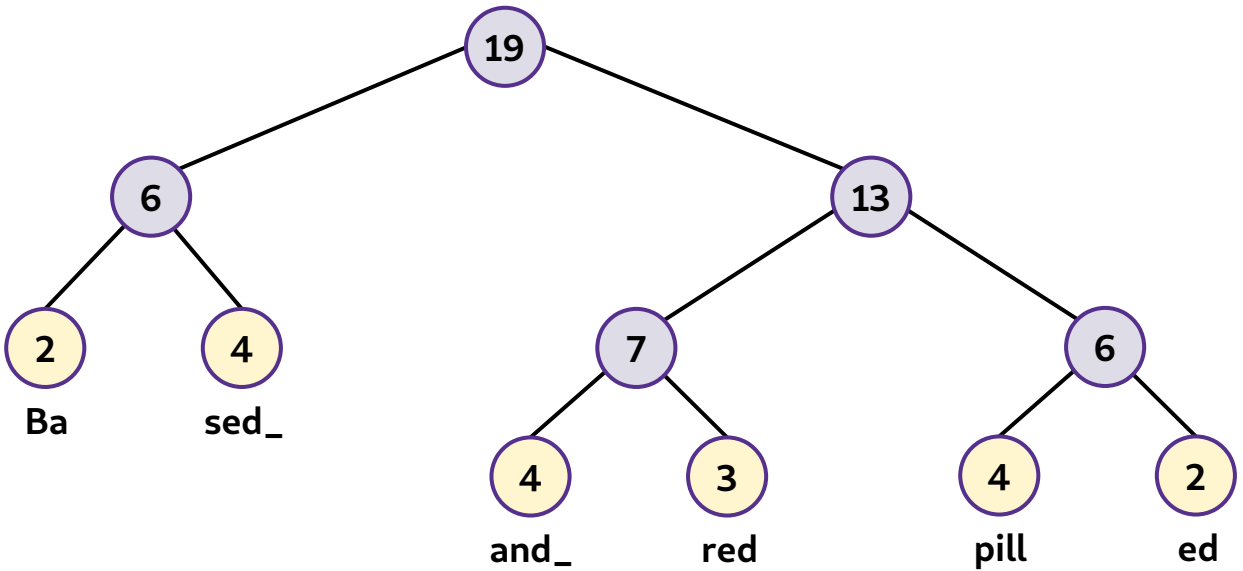
Получение символа по индексу

```
function RopeIndex(rope, index)
    if rope.left != NULL then
        if rope.left.weight > i then
            return RopeIndex(rope.left, i)
        else
            return RopeIndex(rope.right, i - rope.left.weight)
    else
        return rope.string[i]
end function
```

$T_{Index} = O(\log n)$

Получение символа по индексу

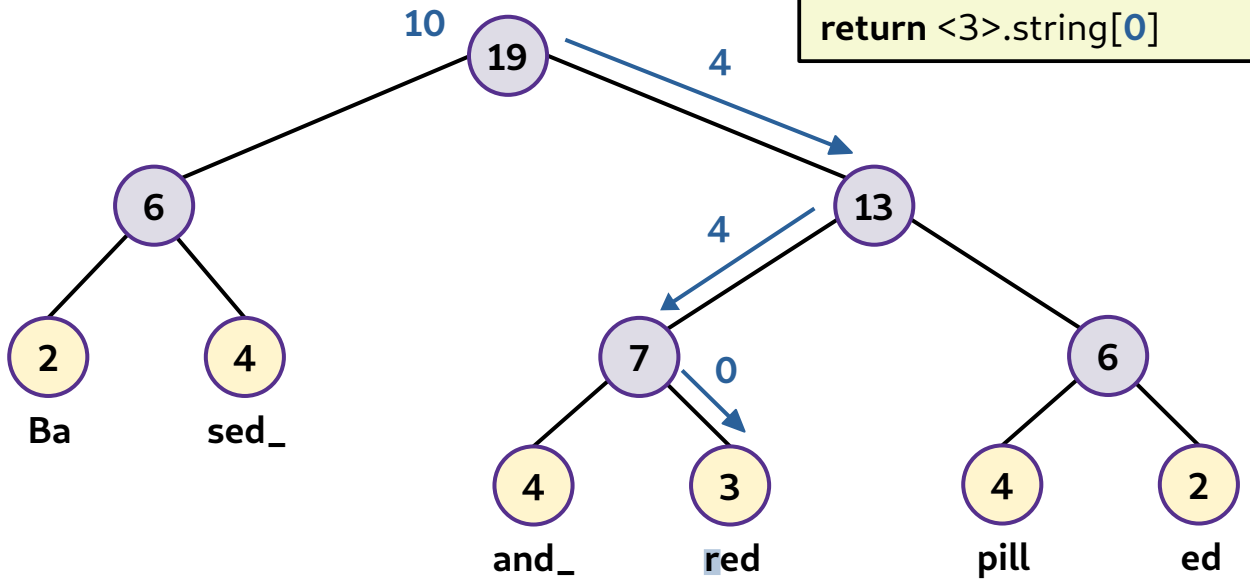
- RopeIndex(*rope*, 10) = ?



Получение символа по индексу

- **RopeIndex**(*rope*, 10) = ‘r’

```
RopeIndex(<19>.right, 10 – <19>.left.weight)
RopeIndex(<13>.left, 4)
RopeIndex(<7>.right, 4 – <7>.left.weight)
return <3>.string[0]
```



Разбиение строки

- Требуется разбить строку $S[c_0..c_m]$ по индексу i на подстроки $S_1[c_0..c_i]$ и $S_2[c_{i+1}..c_m]$. Для этого нужно:
  - Произвести спуск по дереву до листа, содержащего элемент с заданным индексом i
  - Разбить полученный листовой узел на два новых по нужному индексу
  - Разбить все внутренние узлы на пути от листа к корню, получая на каждом уровне пары узлов, принадлежащие левой и правой подстроке соответственно
- Сложность операции — $\boldsymbol{O(\log n)}$

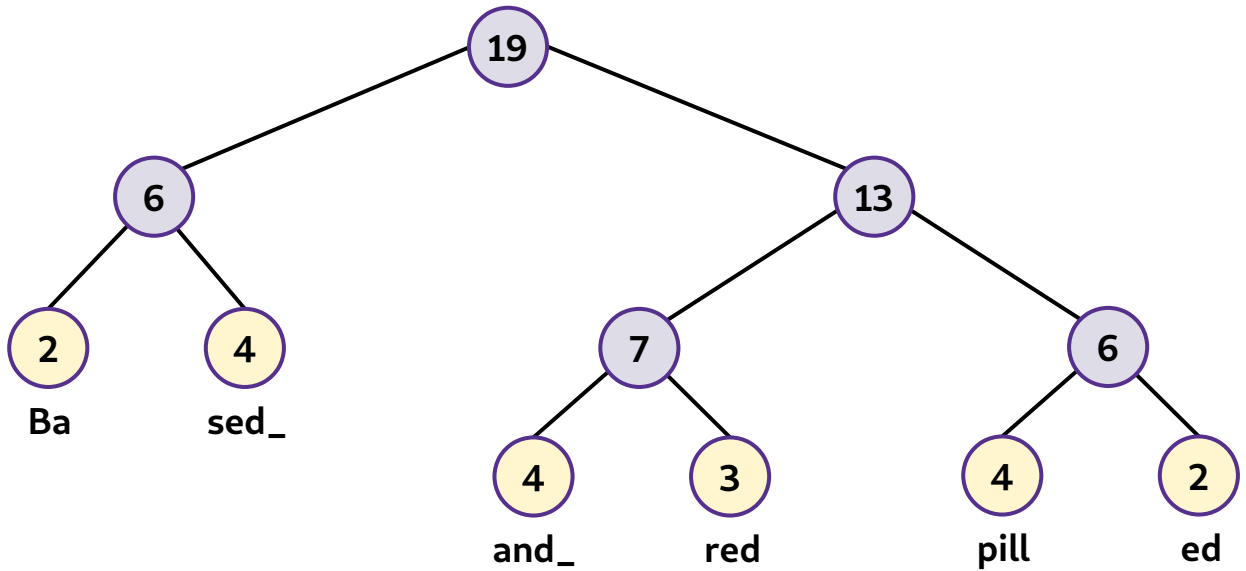
Разбиение строки

```
function RopeSplit(rope, i, &rope1, &rope2)
    if rope.left != NULL then
        if rope.left.weight > i then
            RopeSplit(rope.left, i, rope1, rope2.left)
            rope2.right = rope.right
            rope2.weight = rope2.left.weight + rope2.right.weight
        else
            RopeSplit(rope.right, i - rope.left.weight, rope1.right, rope2)
            rope1.left = rope.left
            rope1.weight = rope1.left.weight + rope1.right.weight
    else
        rope1.string = GetSubstring(rope.string, 0, i)
        rope2.string = GetSubstring(rope.string, i, rope.string.length)
        rope1.weight = i
        rope2.weight = rope.string.length - i
end function
```

$T_{Split} = O(\log n)$

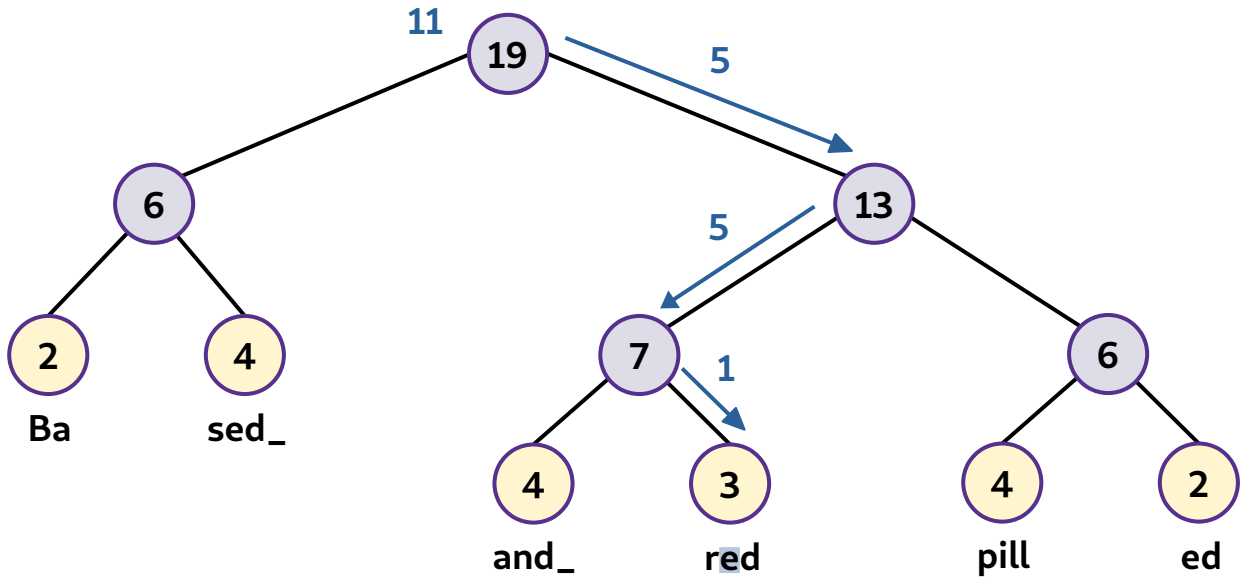
Разбиение строки

- Разбиение rope по индексу **11**



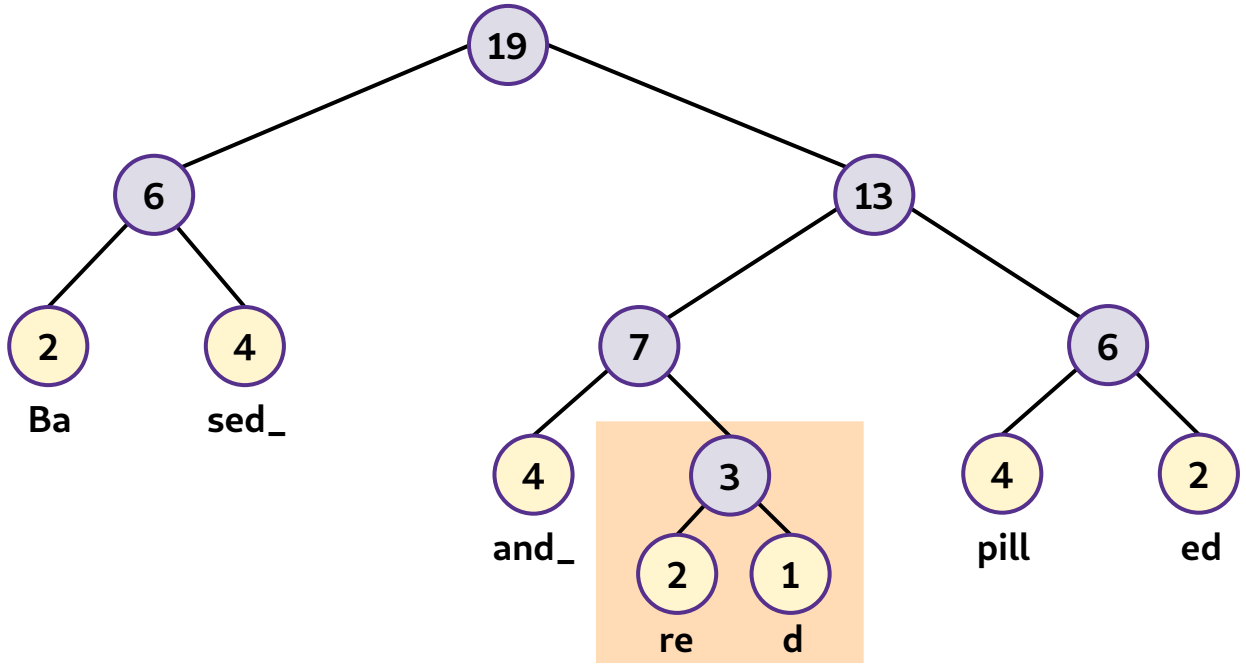
Разбиение строки

- Разбиение rope по индексу **11**
  - Произвести спуск по дереву до листа, содержащего элемент с заданным индексом *i*



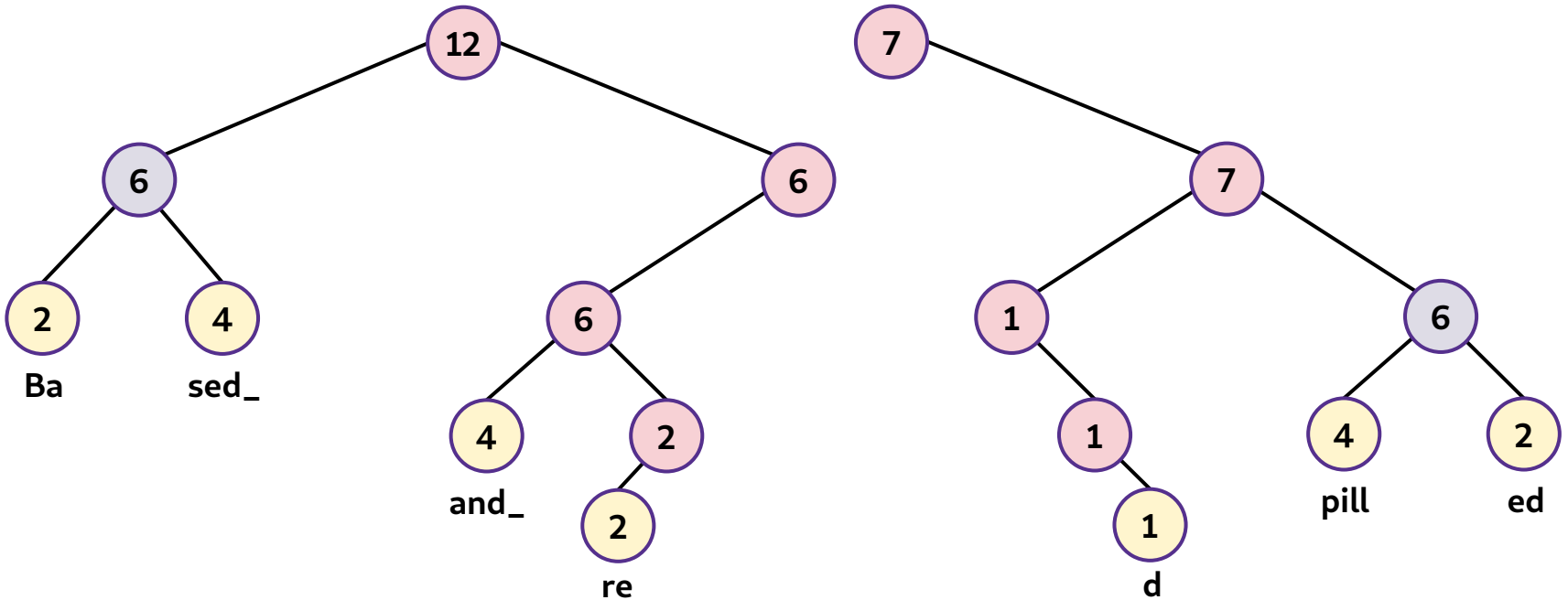
Разбиение строки

- Разбиение rope по индексу **11**
  - Разбить полученный листовой узел на два новых по нужному индексу



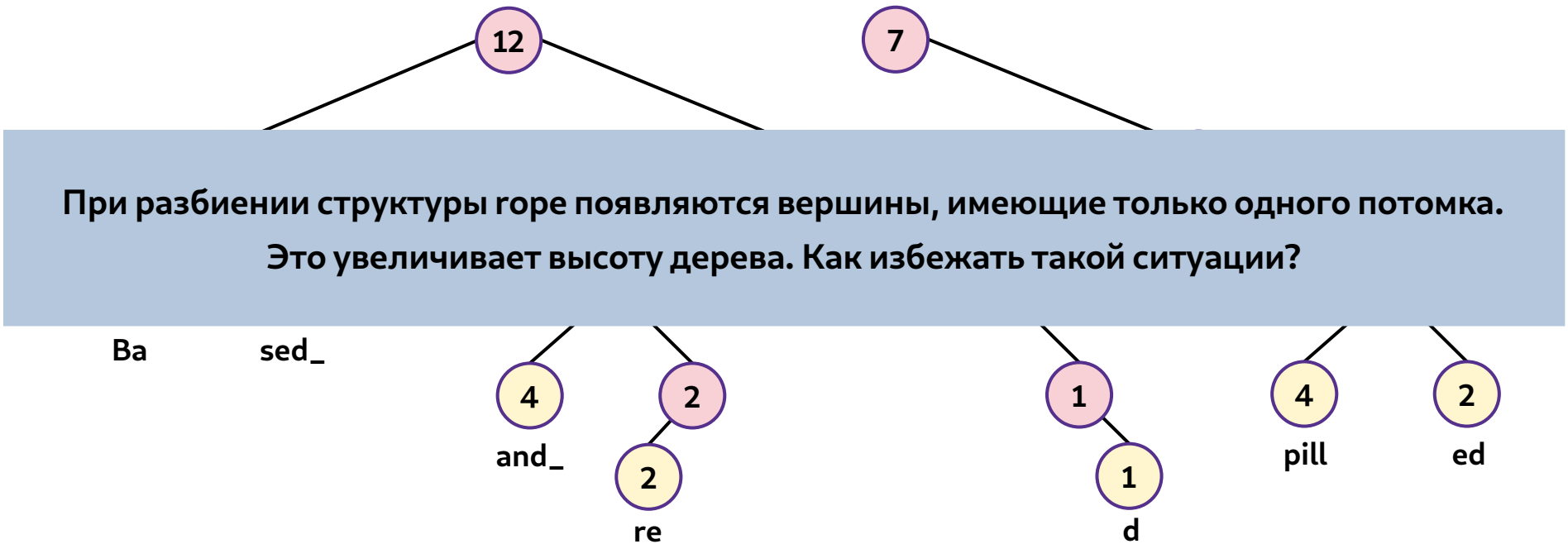
Разбиение строки

- Разбиение rope по индексу **11**
  - Возвращаясь к корню, получить на каждом уровне пары узлов для левой и правой подстрок



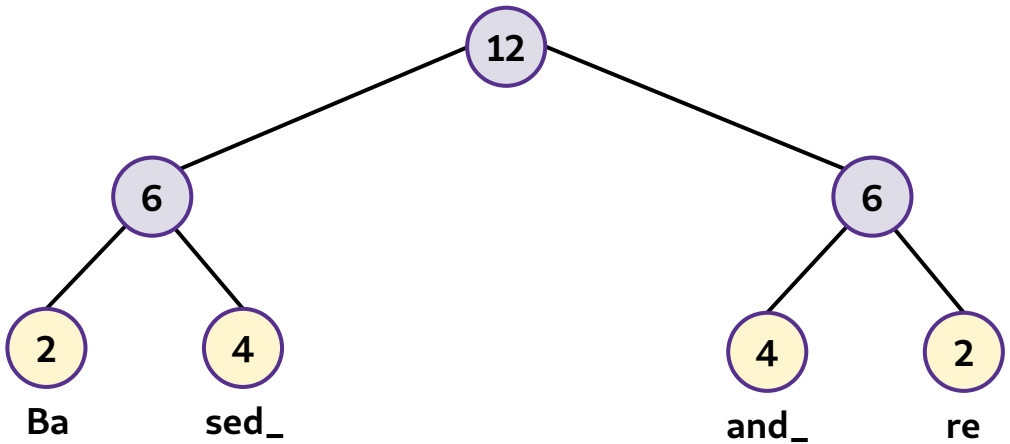
Разбиение строки

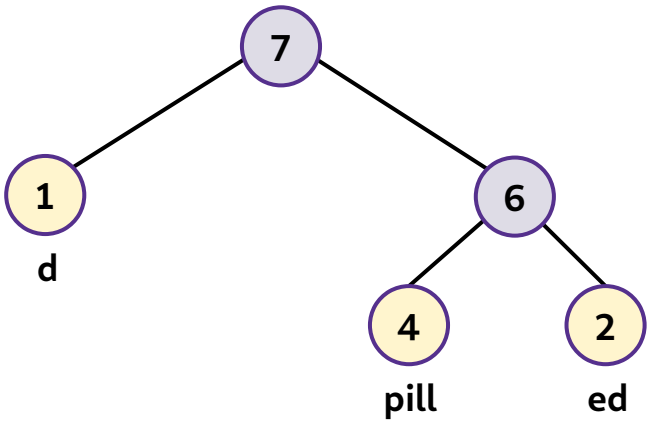
- Разбиение rope по индексу **11**
  - Возвращаясь к корню, получить на каждом уровне пары узлов для левой и правой подстрок



Разбиение строки

- Разбиение rope по индексу **11**
  - **Проходясь по полученным строкам, заменять вершины с одним потомком на этого потомка**





Вставка и удаление подстрок из строки

- Операции вставки и удаления подстрок в строку выполняются на основе операций конкатенации и разбиения
- Для вставки подстроки в строку необходимо разбить исходную строку по нужному индексу, а затем склеить полученные две подстроки и заданную в нужном порядке
- Удаление подстроки производится путём разбиения исходной строки в позициях начала и конца заданной подстроки и последующей конкатенации крайних подстрок
- Сложность операций вставки и удаления подстрок — $O(\log n)$

Вставка и удаление подстрок из строки

```
function RopeInsertSubstring(rope, string, start_index)
    RopeSplit(rope, start_index, rope1, rope3)
    rope2 = RopeCreateNode(string)
    RopeConcatenate(rope1, rope2, rope1_2)
    RopeConcatenate(rope1_2, rope3, result)
    return result
end function
```

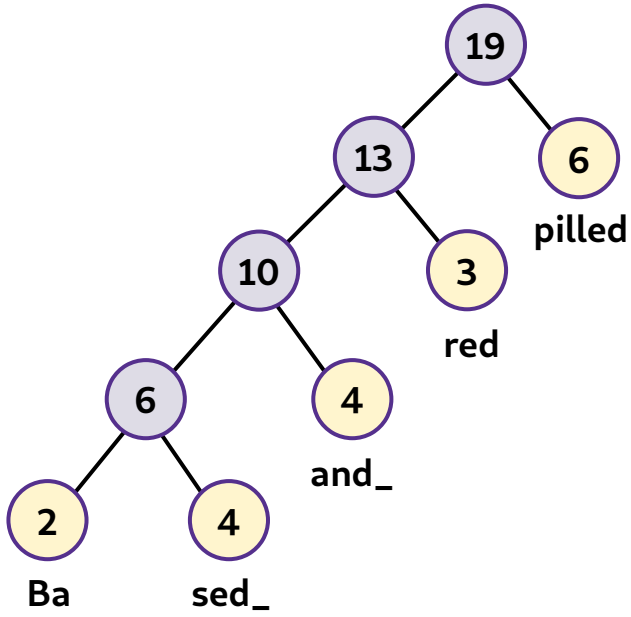
$T_{Insert} = O(\log n)$

```
function RopeDeleteSubstring(rope, start_index, end_index)
    RopeSplit(rope, start_index, rope1, rope2)
    RopeSplit(rope2, end_index - start_index, dump, rope3)
    RopeConcatenate(rope1, rope3, result)
    return result
end function
```

$T_{Delete} = O(\log n)$

Балансировка дерева rope

- Структура rope, являясь двоичным деревом, может вырождаться в связный список
- Для его балансировки возможно использование подходов, применяемых в АВЛ-дереве; также описана методика балансировки rope на основе чисел Фибоначчи* — **домашнее чтение**



* Boehm H. J., Atkinson R., Plass M. **Ropes: an alternative to strings** // Software: Practice and Experience. – 1995. – Т. 25. – №. 12. – С. 1315-1330.

Эффективность rope

Операция	Rope	String
Index(*i*)	$O(\log n)$	$O(1)$
Concatenate($rope_1$, $rope_2$)	$O(1)$	$O(n)$
Split(*i*)	$O(\log n)$	$O(1)$
InsertSubstring(*start*)	$O(\log n)$	$O(n)$
DeleteSubstring(*start*, *end*)	$O(\log n)$	$O(n)$

- Сложность по памяти — $O(n)$
- Представлены оценки сложности rope с учётом использования операции балансировки

Rope: преимущества и недостатки

- По сравнению со строками на основе массивов структура данных rope имеет следующие **преимущества**:
  - Конкатенация строк выполняется за время $O(1)$, вставка и удаление подстрок — за $O(\log n)$
  - Для хранения rope не требуются большие участки смежных адресов памяти
  - Rope является **персистентной структурой данных** — после любого её изменения возможен быстрый откат к предыдущему состоянию
- **Недостатки**:
  - Накладные расходы (*overhead*) на хранение и обработку внутренних узлов
  - Повышенная сложность реализации по сравнению с подходом на основе массивов
- Использование rope предпочтительно для больших строк, которые часто изменяются

Дальнейшее чтение

- Методики балансировки rope (подход с использованием АВЛ-дерева, балансировка на основе чисел Фибоначчи)
- Поиск подстроки в строке с использованием rope
- Реализация rope, в которой внутренние узлы хранят длину строк только своего левого поддерева
- **Буферные окна** (*gap buffer*) — динамические массивы, применяемые в текстовых редакторах для эффективной вставки или удаления элемента в заданной области
- **Cedar ropes** — реализация верёвочных строк в Cedar, расширении алголоподобного языка программирования Mesa (1984)
- **Model-T enfilade** (анфилады) — схожая с rope структура данных, реализованная в проекте Xanadu, первой концепции гипертекста (1971-1972)

ご清聴ありがとうございました!

Даниил Михайлович Берлизов
Старший преподаватель Кафедры вычислительных систем СибГУТИ
E-mail: sillyhat34@gmail.com

Курс «Структуры и алгоритмы обработки данных»
Осенний семестр, 2021 г.