Алгоритмы и структуры данных

Лекция 6
Специальные деревья.

Сергей Леонидович Бабичев

План лекции

1. Бинарная куча и абстракция *приоритетная очередь*.
2. Дерево отрезков.

Абстракция *приоритетная очередь*

Приоритетная очередь

- Приоритетная очередь (`priority queue`) — очередь, элементы которой сравнимы и имеют приоритет.
- После вставки элемента очередь остаётся в упорядоченном по приоритету состоянии.
- Первым извлекается наиболее приоритетный элемент (максимум или минимум).

Интерфейс абстракции:

- `insert` — добавление элемента из очереди;
- `extractMin(Max)` — извлекает самый приоритетный элемент;
- `fetchMin(Max)` — получает самый приоритетный элемент.

Приоритетная очередь

Пример приоритетной очереди:

Значение (`value`)	Приоритет (`priority`)
Москва	12000000
Казань	1500000
Урюпинск	10000
Малиновка	200

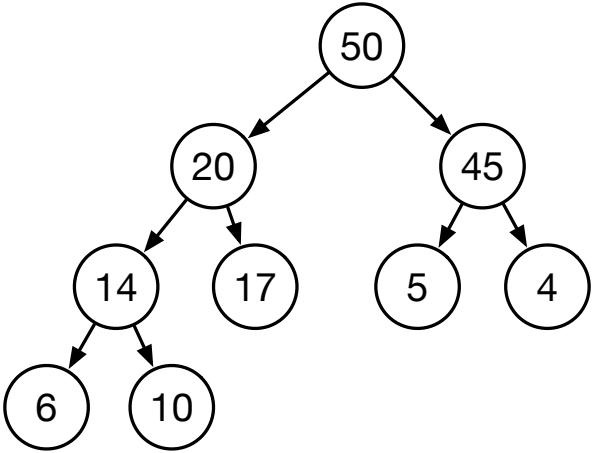
Приоритетная очередь:представление

- *Бинарная куча* — бинарное дерево, удовлетворяющее условиям:
  - Для любой вершины её приоритет не меньше (больше) приоритета потомков.
  - Дерево является правильным подмножеством полного бинарного.

Другое название — пирамида (`heap`).

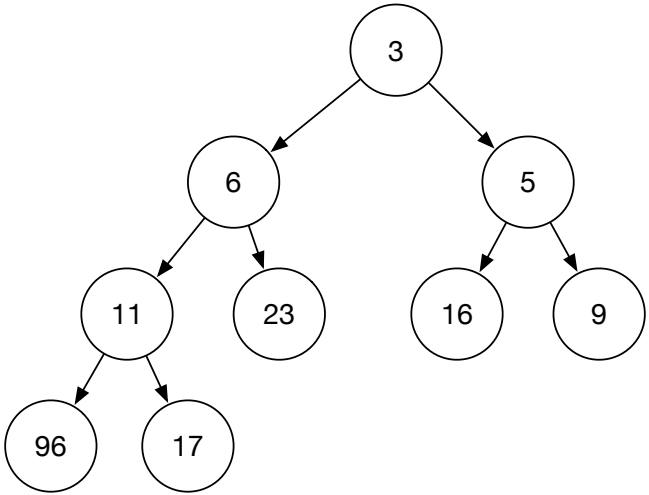
Приоритетная очередь

- Невозрастающая пирамида

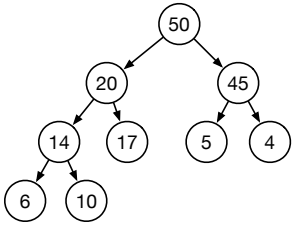


Приоритетная очередь

- Неубывающая пирамида



Приоритетная очередь:реализация



Хранение в виде массива с индексами от 1 до N:

50	20	45	14	17	5	4	6	10

- Индекс корня дерева всегда равен 1 — максимальный (минимальный) элемент
- Индекс родителя узла i равен $\lfloor \frac{i}{2} \rfloor$
- Индекс левого потомка узла i равен $2i$
- Индекс правого потомка узле i равен $2i + 1$

Приоритетная очередь: реализация на базе массива.

```
struct bhnode { // node
  string data;
  int priority;
};

struct binary_heap {
  bhnode *body;
  int     bodysize;
  int     numnodes;
  binary_heap(int maxsize);
 ...
};
```

Бинарная куча

- Создание бинарной кучи

```
binary_heap::binary_heap(int maxsize) {
  body = new bhnode[maxsize+1];
  bodysize = maxsize;
  numnodes = 0;
}

~binary_heap::binary_heap() {
  delete body;
}

void binary_heap::swap(int a, int b) {
  std::swap(body[a],body[b]);
}
```

$$T_{create} = O(N)$$

Бинарная куча: поиск минимума(максимума)

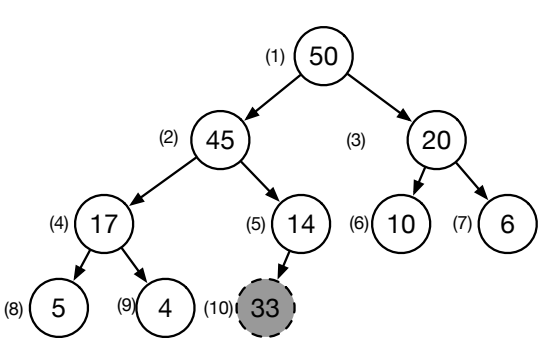
- Поиск в `min-heap`:

```
std::optional<bhnode> binary_heap::fetchMin() {
   if (numnodes <= 0) return std::nullopt;
   return {body[1]};
}
```

$$T_{fetchMin} = O(1)$$

Бинарная куча: вставка элемента

- Этап 1. Вставка в конец кучи.

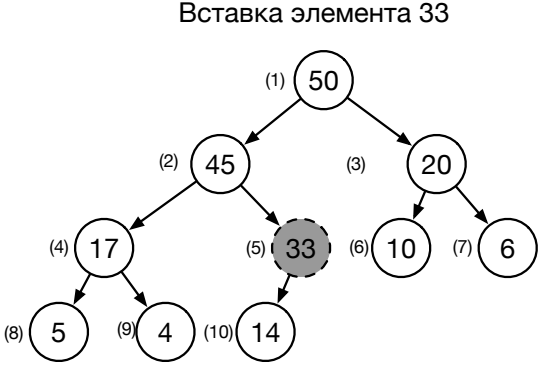


Отлично! *Структура* кучи не испортилась!

50	45	20	17	14	10	6	5	4	33

Бинарная куча: вставка элемента

- Этап 2. Корректировка значений.

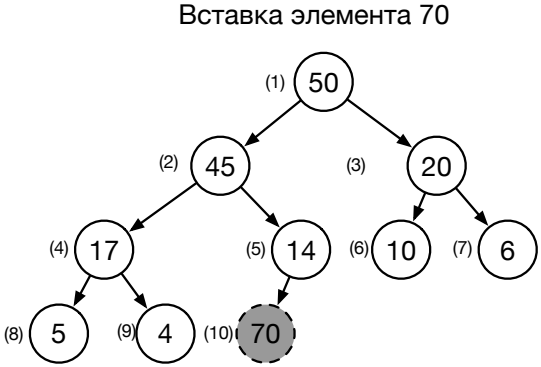


Куча удовлетворяет всем условиям.

50	45	20	17	33	10	6	5	4	14

Бинарная куча: вставка элемента

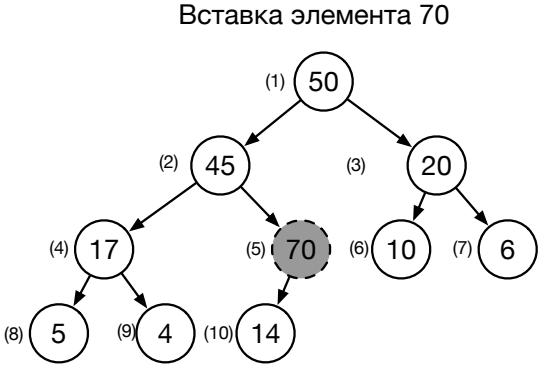
- Попытаемся вставить максимальный элемент.



50	45	20	17	14	10	6	5	4	70

Бинарная куча: вставка элемента

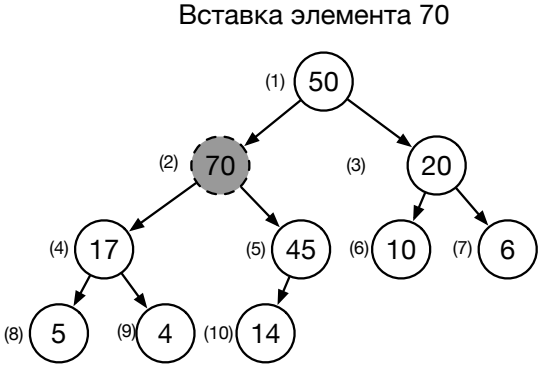
- Максимальный элемент ползёт вверх по дереву.



50	45	20	17	70	10	6	5	4	14

Бинарная куча: вставка элемента

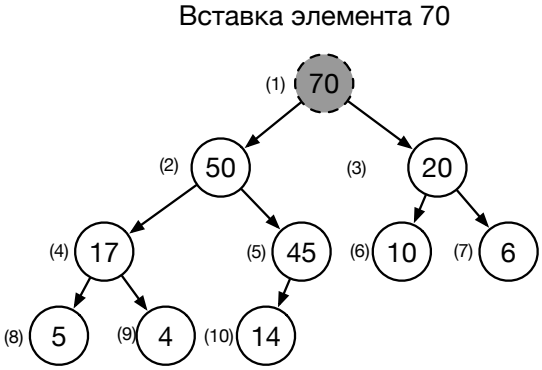
- Максимальный элемент ползёт вверх по дереву.



50	70	20	17	45	10	6	5	4	14

Бинарная куча: вставка элемента

- Максимальный элемент ползёт вверх по дереву.



70	50	20	17	45	10	6	5	4	14

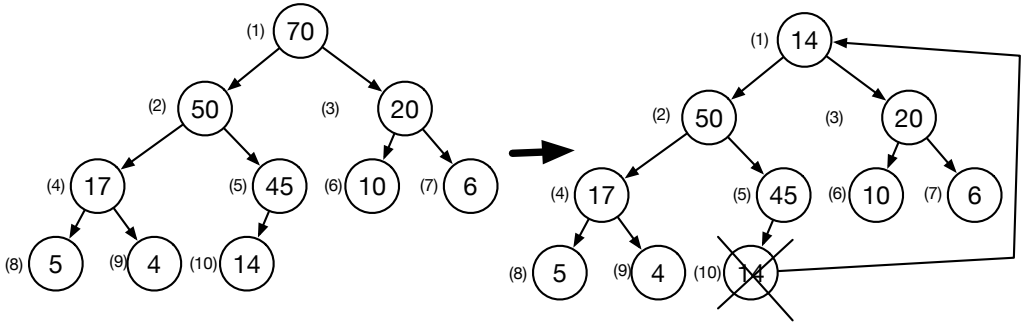
Бинарная куча: вставка элемента

- Реализация.

```
int binary_heap::insert(bhnode node) {
  if (numnodes > bodysize) {
    return -1; // or expand
  }
  body[++numnodes] = node;
  for (size_t i = numnodes;
   i > 1 && body[i].priority > body[i/2].priority;
   i /= 2) {
    swap(i, i/2);
  }
  return 0;
}
```

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

Бинарная куча: удаление максимального (минимального)



Свойства кучи нарушены. Требуется восстановление.

Бинарная куча: восстановление свойств

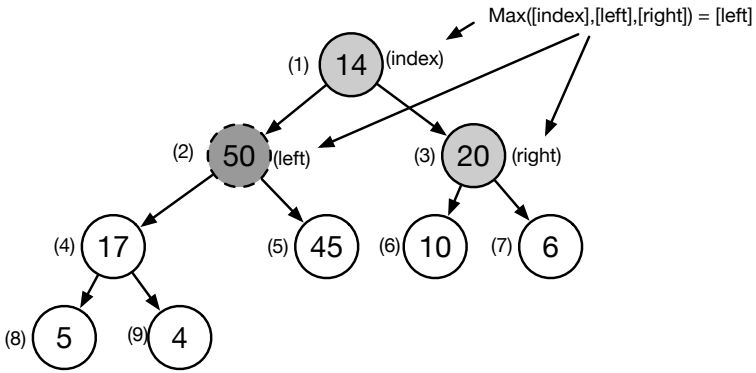
- Для восстановления свойств применяем функцию $heapify$.

```
void binary_heap::heapify(size_t index) {
  for (;;) {
    auto left = index + index, right = left + 1;
    // Who is greater, [index], [left], [right]?
    auto largest = index;
    if (left <= numnodes && body[left].priority > body[index].priority)
      largest = left;
    if (right <= numnodes && body[right].priority > body[largest].priority)
      largest = right;
    if (largest == index) break;
      swap(index, largest);
    index = largest;
  }
}
```

$$T_{heapify} = O(\log N)$$

Бинарная куча: восстановление свойств

- Восстановление индекса (1)

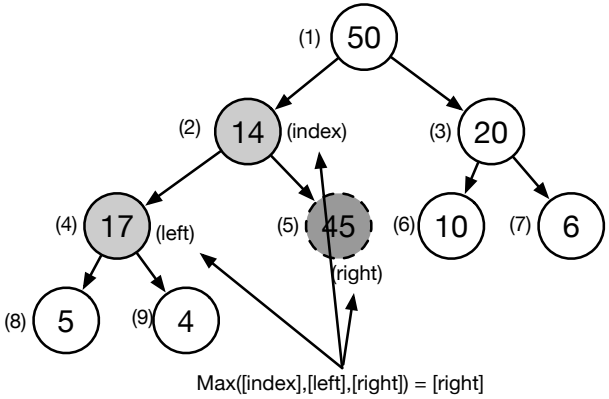


14	50	20	17	45	10	6	5	4

Новый индекс для восстановления (2)

Бинарная куча: восстановление свойств

- Восстановление индекса (2)

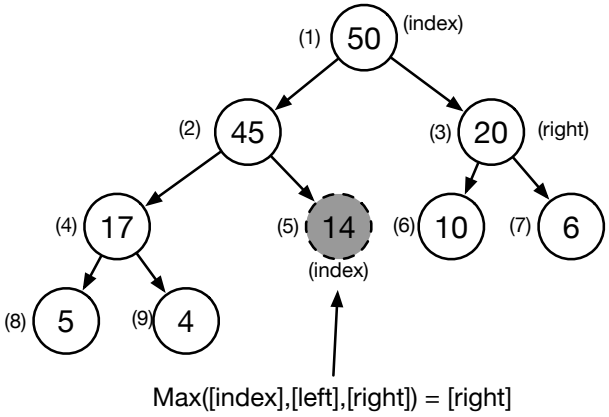


50	14	20	17	45	10	6	5	4

Новый индекс для восстановления (5)

Бинарная куча: восстановление свойств

- Восстановление индекса (5)



50	45	20	17	14	10	6	5	4

Восстановление закончено.

Бинарная куча: сложность операций

- **insert** — $O(\log N)$
- **extractMin(Max)** — $O(\log N)$
- **fetchMin(Max)** — $O(1)$

HeapSort

HeapSort

- На основе бинарной кучи можно реализовать алгоритм сортировки со сложностью $O(N \log N)$ в *худшем* случае.
- Как?
- Является ли отсортированным массив, являющийся представлением бинарной кучи?

HeapSort

- На основе бинарной кучи можно реализовать алгоритм сортировки со сложностью $O(N \log N)$ в *худшем* случае.
- Как?
- Является ли отсортированным массив, являющийся представлением бинарной кучи?
- Нет.
- Кто виноват? Что делать?

HeapSort

- Нужно скомбинировать методы бинарной кучи.
  1. Создать бинарную кучу.
  2. Вставить в неё элементы массива
  3. Извлекать из неё максимальный (минимальный) элемент с удалением.

HeapSort

- Примерный код сортировки HeapSort

```
struct bhnode { // node
  int priority;
};

void heapsort(int v[], size_t vsize) {
  binary_heap *h = new binary_heap(vsize);
  for (size_t i = 0; i < vsize; i++) {
    bhnode b; b.priority = v[i];
    h->insert(b);
  }
  for (size_t i = 0; i < vsize; i++) {
    v[i] = h->extractMin()->priority;
  }
  delete h;
}
```

HeapSort

- Сложность алгоритма:
  1. Создание бинарной кучи — $T_1 = O(1)$.
  2. Вставка N элементов — $T_2 = O(N \log N)$.
  3. Извлечение удалением N элементов — $T_3 = O(N \log N)$.

$$T_{heapsort} = O(1) + O(N \log N) + O(N \log N) = O(N \log N)$$

HeapSort

- Реализация не особенно хороша: требуется $O(N)$ добавочной памяти на бинарную кучу.
- Небольшая хитрость — и добавочной памяти можно избежать.

HeapSort

Модифицируем функцию heapify. size — размер кучи.

```
void heapify(int v[], size_t i, size_t size)
{
  auto curr = v[i];
  auto index = i;
  for (;;) {
    auto left = index + index + 1, right = left + 1;
    if ( left < size && v[left] > curr)
      index = left;
    if ( right < size && v[right] > a[index])
      index = right;
    if (index == i ) break;
    v[i] = v[index];
    v[index] = curr;
    i = index;
  }
}
```

HeapSort

- Создаём бинарную кучу размером n на месте исходного массива, переставляя его элементы.
- Затем на шаге i мы обмениваем самый приоритетный элемент кучи из позиции 0 с элементом под номером $n - i - 1$.
- Размер кучи при этом уменьшается на единицу, а самый приоритетный элемент занимает теперь положенное ему по рангу место.

```
void sort_heap(int v[], size_t n) {
  for(size_t i = n/2-1; i >= 0; i--) {
    heapify(a, i, n);
  }
  while( n > 1 ) {
    n--;
    swap(a[0],a[n]);
    heapify(a, 0, n);
  }
}
```

HeapSort vs QuickSort

- `HeapSort` гарантирует сложность $O(N \log N)$ даже в худшем случае.
- `QuickSort` такой сложности не гарантирует.
- Почему не забыть о `QuickSort` в пользу `HeapSort`?

HeapSort vs QuickSort

- `HeapSort` гарантирует сложность $O(N \log N)$ даже в худшем случае.
- `QuickSort` такой сложности не гарантирует.
- Почему не забыть о `QuickSort` в пользу `HeapSort`?

- Причина 1: в быстрой сортировке используется меньшее количество операций обмена с памятью.
- Причина 2: N обращений к последовательным ячейкам памяти исполняются до 10-15 раз быстрее, чем столько же обращений к случайным ячейкам памяти из-за организации кэш-памяти.

Дерево отрезков

Дерево отрезков

Пусть нам надо решить задачи:

- Многократное нахождение максимального значения на отрезках массива.
- Многократное нахождение суммы на отрезке массива.

Мы умеем совершать эти действия за время $O(N)$, где $N = R - L + 1$. При определённой подготовке их можно совершать за $O(\log N)$.

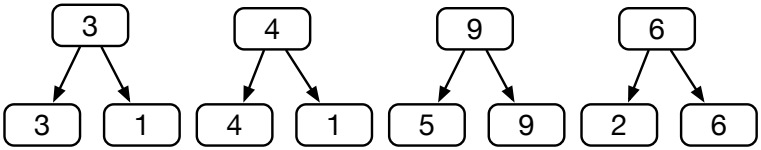
Дерево отрезков

Попробуем воспользоваться бинарными деревьями.
Для примера возьмём массив {3, 1, 4, 1, 5, 9, 2, 6}.

Вот как выглядит этот массив:

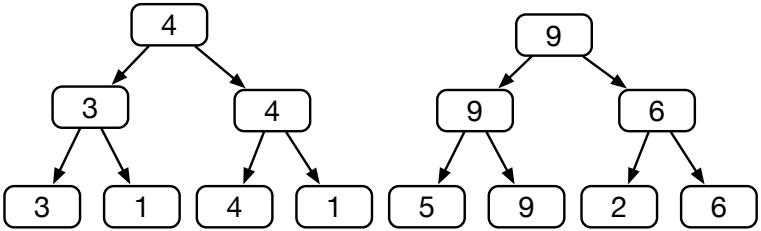
Дерево отрезков

Попарно соединим соседние вершины, поместив в узел-родитель значение функции `max(left,right)`.



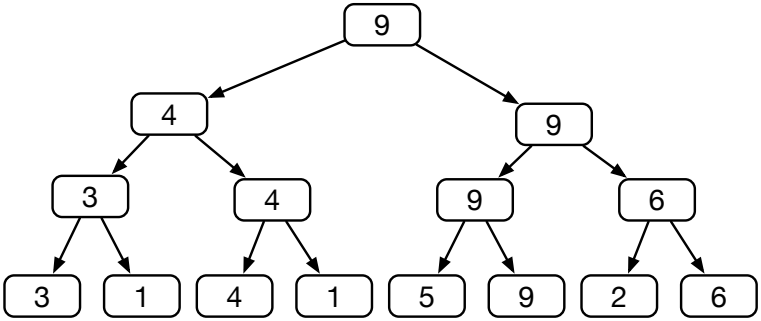
Дерево отрезков

Проделаем эту же операцию от получившихся узлов:



Дерево отрезков

Наконец:



Родитель каждого узла называется *доминирующим узлом*.

Дерево отрезков: представление

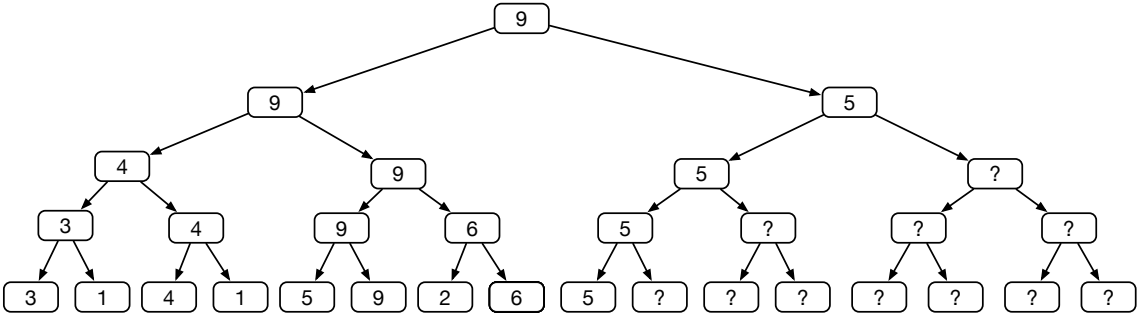
Возможный вариант представления — обычное бинарное дерево с указателями.

- На каждый узел требуется два указателя вниз.
- Для удобной работы требуется индикатор «левый/правый узел» и один указатель на родителя.
- Минимум 4 элемента на узел.

Но ведь это же полное бинарное дерево? Тогда почему не использовать бинарную кучу?

Дерево отрезков: бинарная куча

Бинарная куча требует полного бинарного дерева. Количество элементов должно быть степенью двойки.



Что должно находиться в узлах, отмеченными знаками вопроса?

Дерево отрезков

Все значения в узлах вычисляются с помощью функции

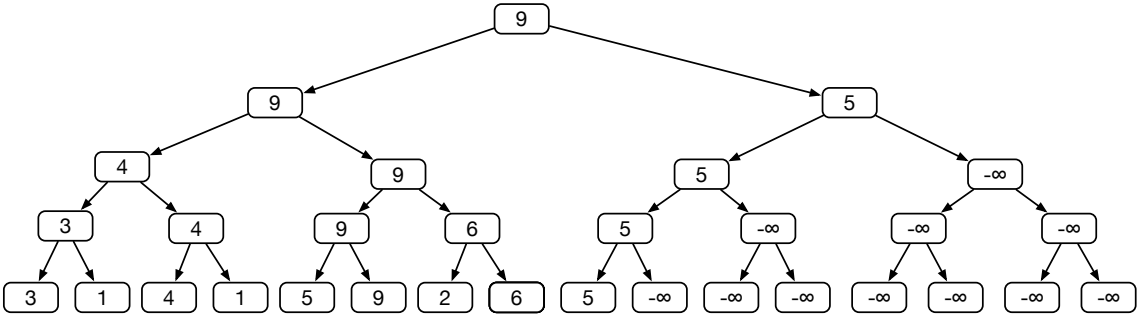
$$P = \max(L, R).$$

Чтобы не плодить сущности, то же самое должно происходить с элементом `'?'`.

То есть элемент `'?'` есть $-\infty$.

Для функции `max` число $-\infty$ есть *нейтральный элемент*.

Дерево отрезков



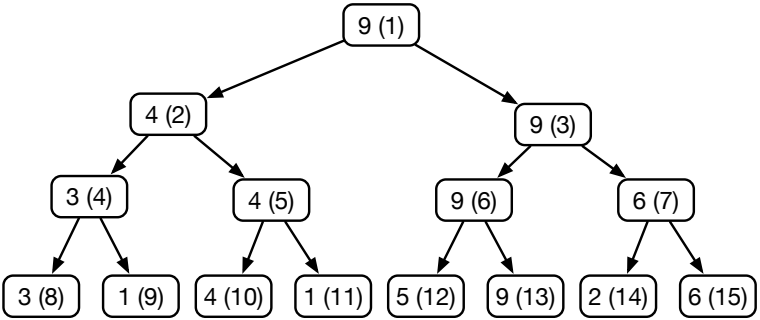
Дерево отрезков

Идея дерева отрезков распространяется на все такие функции, в которых:

$$\begin{aligned} A \circ B &= B \circ A \\ A \circ (B \circ C) &= (A \circ B) \circ C \\ \exists E : A \circ E &= A \end{aligned}$$

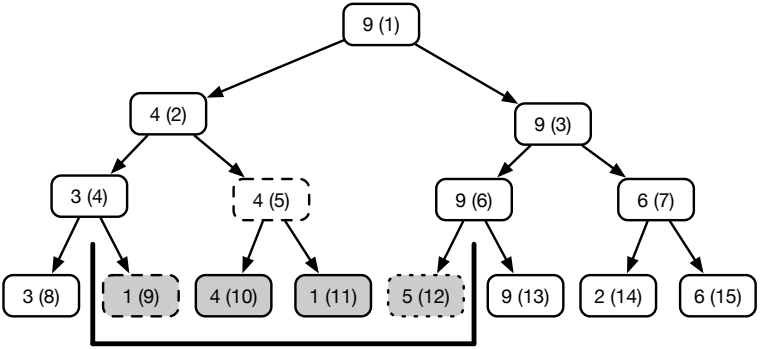
Операция	Нейтральный элемент
`max`	$-\infty$
`min`	$+\infty$
`+`	0
`*`	1

Дерево отрезков: алгоритмы



- ***Create(size):*** создаётся бинарная куча, инициализированная нейтральными элементами. $C = \min(2^k) : C \geqslant size$.
- ***Insert/Replace(i, val):*** `body[i+C]=val; propagate(i);`

Дерево отрезков: функция на отрезке



- ***Func(left,right):***
  - `Res = E`
  - `if (left % 2 == 1) Res = Op(Res,body[left++])`
  - `if (right % 2 == 0) Res = Op(Res,body[right--])`
  - `if (right > left) Res = Op(Res,Func(left/2, right/2))`

Дерево отрезков

Сложность операций:

- Требуемая память: $\min = O(2N) \ldots \max = O(4N)$.
- Операция ***Insert/Replace***: $O(\log N)$.
- Операция ***Func*** на любом подотрезке: $O(\log N)$.

Спасибо за внимание.

Следующая лекция — списки, деревья.