PriorityQueue

Неубывающая очередь с приоритетами

Толстых Тимур, 11-401

Введение

Определение: Приоритетная очередь — это абстрактная структура данных наподобие стека или очереди, где у каждого элемента есть приоритет. Элемент с более высоким приоритетом находится перед элементом с более низким приоритетом.

Сложность операций

	Average	Worst
Enqueue	O(log(n))	O(log(n))
Dequeue	O(log(n))	O(log(n))
Peek	O(1)	O(1)
DecreaseKey	O(log(n))	O(log(n))

*Такая асимптотика достигается за счет реализации очереди на двоичной куче

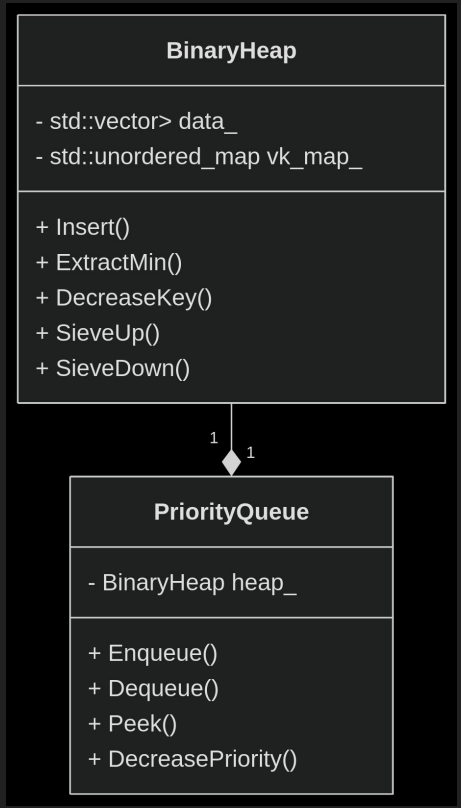
Реализация

Реализация очереди с приоритетами использует бинарную кучу, содержащую в качестве данные std::pair<Key, Value>, где Key - это приоритет, а Value - шаблонный параметр.

Чем меньше Key, тем больше приоритет элемента Value в очереди.

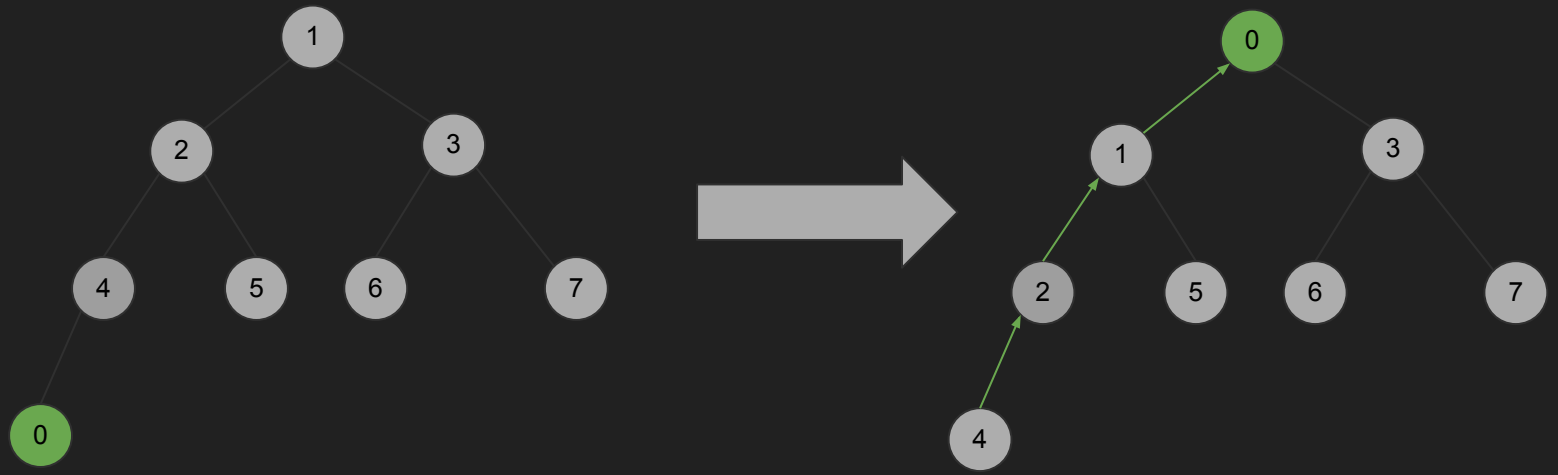
В корне дерева лежит пара с минимальным значением Key (самый приоритетный элемент).

Элементы Value уникальны в очереди



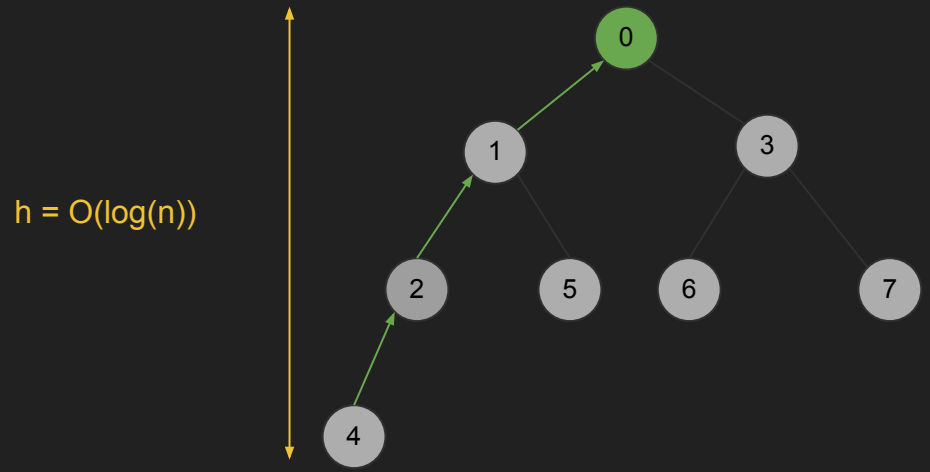
Enqueue

В реализации используется **min-heap,** дочерние ноды узла должны быть больше него. Сама операция сводится к добавлению элемента в конец массива и его последующего просеивания, пока он не встанет на свое место.



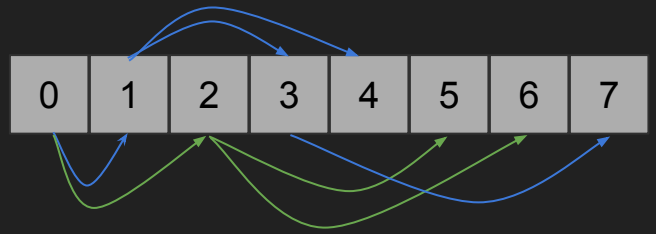
Enqueue

Так как при построении **min-heap** дерево сбалансировано, то такая операция стоит O(log(n)), где n - количество элементов в дереве



Enqueue

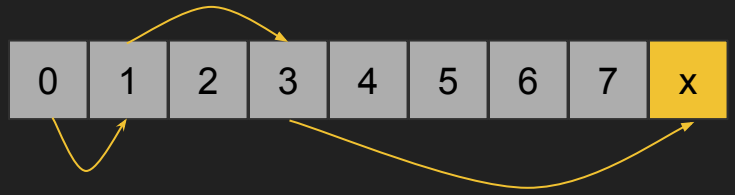
Так как min-heap строится на основе массива, то важно понять как устроены индексы дочерних узлов а также предка:



Переход из ноды в **левого** потомка: i -> 2 * i + 1

Переход из ноды в **правого** потомка: i -> 2 * i + 2

Переход от ноды к **предку:** i -> (i - 1) / 2



При добавлении новой ноды в конец массива, она автоматически становится листом в этом дереве, останется лишь просеять ее вверх при необходимости

Enqueue

Реализация SieveUp: после вставки элемента в конец массива, начнем его просеивать вверх. Процедура останавливается, либо если узел стал корнем дерева, либо его предок меньше, что значит, что узел нашел свое место.

Строка 3 - проверка основного условия

Строка 4 - меняем данные местами в массиве

Строка 5 - обновляем index старым значением parent, в parent кладем индекс следующего предка

```
1 template <typename Value> void BinaryHeap<Value>::SieveUp(size_t index) {
2   size_t parent_index = (index - 1) / 2;
3   while (index > 0 && data_[index].first < data_[parent_index].first) {
4     SwapData(index, parent_index);
5     index = std::exchange(parent_index, (parent_index - 1) / 2);
6   }
7 }
```

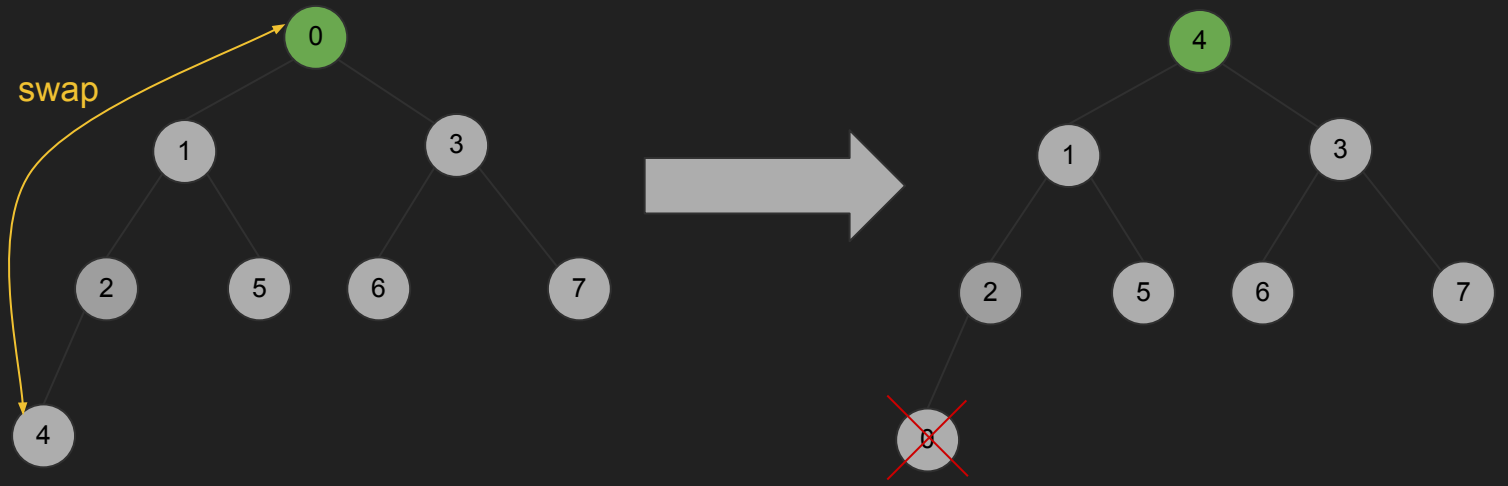
Peek

Так как в min-heap минимальный элемент множества всегда в корне дерева, то вернем его значение за O(1), обратившись к первому элементу массива.

```
template <typename Value>
auto BinaryHeap<Value>::PeekMin() const -> const value_type & {
  if (IsEmpty())
    throw std::out_of_range("Heap is empty");
  return data_.front();
}
```

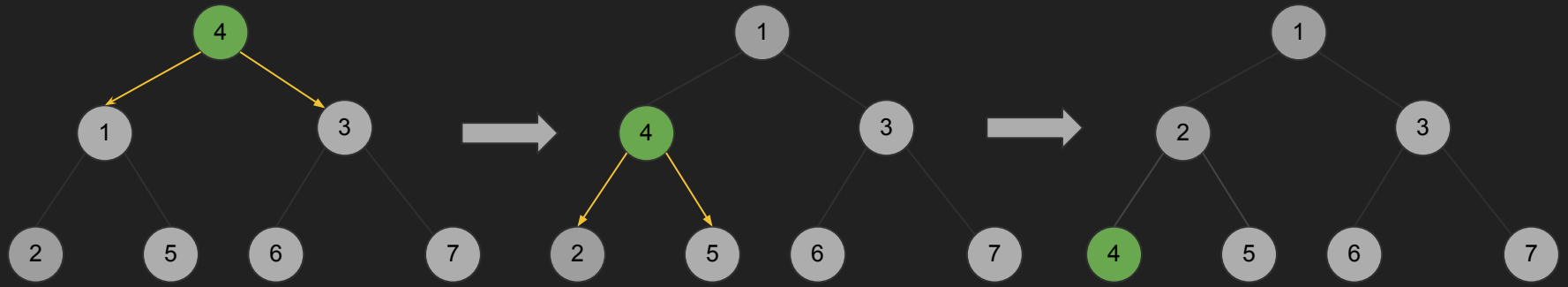
Dequeue

Задача Dequeue - удалить минимальный элемент в дереве (корень). Для этого поменяем местами **значение корня** дерева с **последним элементом** в массиве. Удаляем последний элемент, а значение корня **просеиваем вниз**, пока оно не найдет место в дереве.



Dequeue

При просеивании выбираем наименьшего потомка и меняемся местами с ним, пока не найдем свое место:



Dequeue

Реализация sieveDown: идем вниз по дереву пока есть ноды, выбираем наименьшего потомка, если наша нода больше этого потомка, то делаем swap, иначе прекращаем процедуру, потому что нода нашла свое место.

Так как спуск по дереву происходит один раз, то сложность Dequeue как и у Enqueue - O(log(n))

```
template <typename Value> void BinaryHeap<Value>::SieveDown(size_t index) {
  while (2 * index + 1 < data_.size()) {
    size_t left = 2 * index + 1;
    size_t right = 2 * index + 2;
    size_t curr_index = left;
    if (right < data_.size() && data_[right].first < data_[left].first) {
      curr_index = right;
    }
    if (data_[index].first <= data_[curr_index].first)
      break;
    SwapData(index, curr_index);
    index = curr_index;
  }
}
```

DecreaseKey

Находит Value в массиве и меняет его приоритет на новый.

Чтобы операция работала за O(log(n)) используем unordered_map с отображением Value -> index (значение и его индекс в массиве). Получаем позицию Value за O(1), далее ставим новый приоритет и просеиваем вверх.

```cpp
template <typename Value>
void BinaryHeap<Value>::DecreaseKey(const Value &value, key_type new_key) {
  auto it = vk_map_.find(value);
  if (it == vk_map_.end()) {
    throw std::invalid_argument("Value not found");
  }

  size_t index = it->second;
  if (data_[index].first <= new_key) {
    return;
  }

  data_[index].first = new_key;
  SieveUp(index);
}
```

DecreaseKey

Чтобы поддерживать актуальные индексы в хэш-таблице, при операциях просеивания меняем местами не только данные но и индексы.

```
template <typename Value>
void BinaryHeap<Value>::SwapData(size_t i1, size_t i2) {
  if (i1 == i2)
    return;
  assert(i1 < data_.size() && i2 < data_.size());
  std::swap(data_[i1], data_[i2]);
  vk_map_[data_[i1].second] = i1;
  vk_map_[data_[i2].second] = i2;
}
```

Плюсы

+ Все критические операции за O(log(n))
+ Константное время для Peek
+ Хорошая кэшируемость из-за локальности данных

Минусы

- Нет эффективного поиска по значению (только по приоритету, либо хранить хэш-таблицу Value -> index)
- При массовых обновлениях проигрывает другим структурам
- Невозможно удалить произвольный элемент за O(log(n))

Применение

- Алгоритмы на графах (Дейкстра, Прим, A*)
- Системы реального времени с приоритетами
- Задачи с частым обновлением приоритетов