Priority Queue

Основная теория

В чем суть структуры

Priority Queue - это абстрактная структура данных, совмещающая в себе свойства двух структур данных: Heap и Queue. От очереди нам требуется добавление в начало и удаление из конца очереди, а от кучи ее свойства поддержания наибольшего (либо наименьшего в случае мин-кучи) на верху.

Свойства кучи пригождаются тогда, когда нам нужно изменить приоритетность элемента очереди после его добавления.

Примеры использования приоритетной очереди: планировщик задач в операционной системе,

> **Важно**
>
> Если мы добавим в приоритетную очередь два элемента с одинаковыми ключами, то она не обязательно отдаст их в том же порядке, в котором они были добавлены. В таких случаях добавляются дополнительные параметры для преоретизации элементов.

Идейная реализация

Основные операции, которые добавляет приоритетная очередь, по сравнению с кучей:

- `IncreaseKey(i, newKey)` - увеличивает ключ у элемента в очереди по индексу `i`. Сначала мы изменяем значение ключа у элемента, после чего пытаемся выполнять операцию свапа с его родителем до тех пор, пока `queue.data[child].key > queue.data[parent].key` либо пока мы не дошли до корня кучи. Случай на `child == 0` **необходимо проверить**, поскольку `Parent(0) = 0` и может возникнуть бесконечный цикл.
- `ExtractMax()` - извлечение максимального элемента из очереди. Записываем в переменную максимальное значение кучи. Обмениваем значения `queue.data[0]` и `queue.data[queue.size - 1]`. Уменьшаем размер кучи `queue.size--`. Выполняем операцию `MaxHeapify` для восстановления свойств кучи.
- `Insert(value, key)` - добавление в очередь значения `value` с ключом `key`.

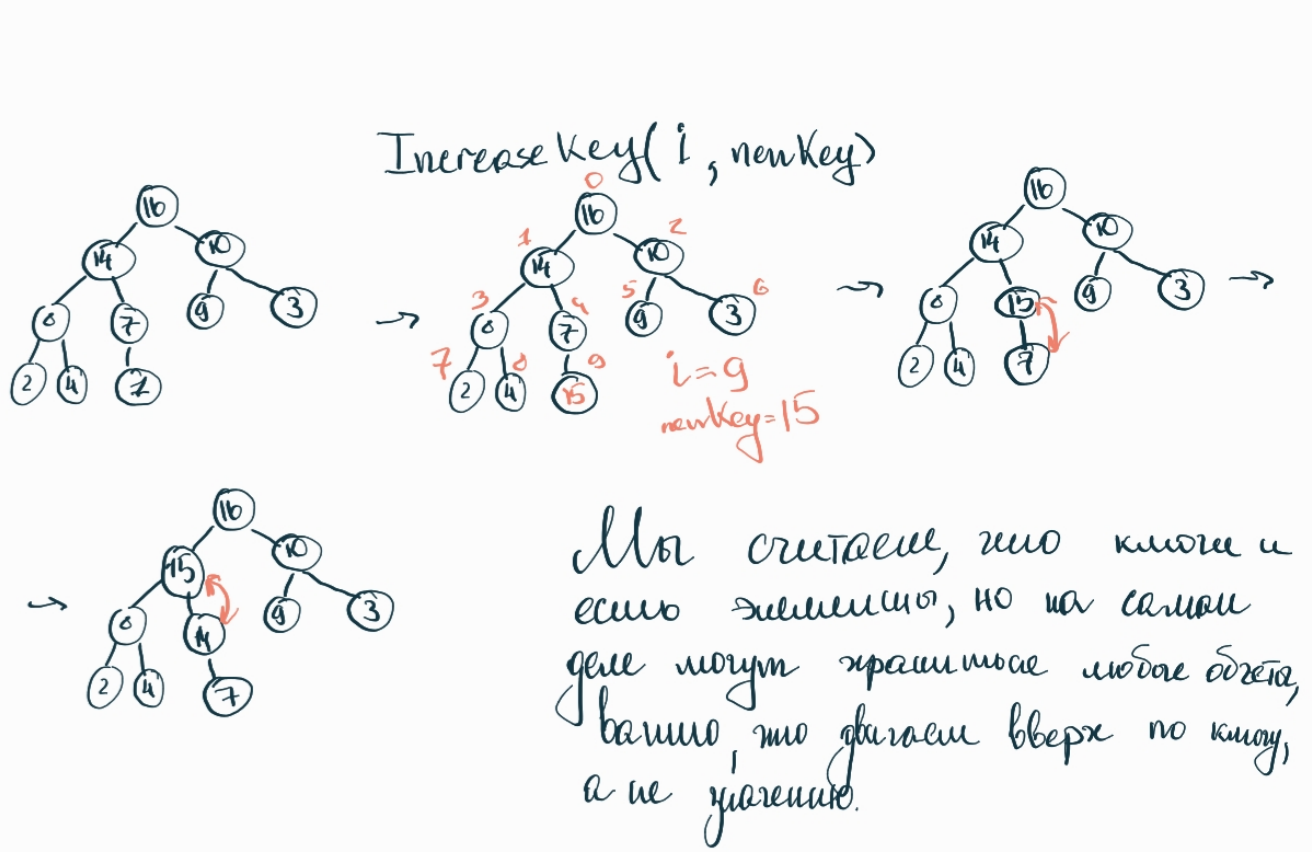
Операции над структурой и асимптотическая сложность

- `IncreaseKey(i, newKey)` - выполняется за **O(log n)**, поскольку нам необходимо в худшем случае поднять элемент выше по дереву на высоту **O(log n)**.

- `ExtractMax()` - выполняется за **O(log n)**, поскольку после извлечения нам необходимо восстановить свойства кучи.
- `Insert(value, key)` - выполняется за **O(log n)**, поскольку нам необходимо в худшем случае поднять элемент выше по дереву на высоту **O(log n)**.

Иллюстрации

Добавление нового ключа в кучу:



Примеры кода

Инициализация структур:

```
type PrQueueNode struct {
    value int
    key   int
}

type PrQueue struct {
    data []PrQueueNode
    size int
}

func NewPrQueue() *PrQueue {
    return &PrQueue{
        data: make([]PrQueueNode, 0),
        size: 0,
```

```
    }
}
```

Нахождение потомков, родителя и максимума:

```
func (q *PrQueue) Max() (PrQueueNode, error) {
    var retval PrQueueNode
    if q.size < 1 {
        return retval, ErrQueueIsEmpty
    }
    return q.data[0], nil
}

func (q *PrQueue) Parent(child int) int {
    return child / 2
}

func (q *PrQueue) Left(parent int) (int, error) {
    if parent*2+1 >= q.size {
        return 0, ErrNoLeftChild
    }

    return parent*2 + 1, nil
}

func (h *PrQueue) Right(parent int) (int, error) {
    if parent*2+2 >= h.size {
        return 0, ErrNoRightChild
    }
    return parent*2 + 2, nil
}
```

Специфичные для приоритетной очереди операции

```
func (q *PrQueue) MaxHeapify(elem int) {
    largest := elem

    left, err := q.Left(elem)
    if err == nil && q.data[left].key > q.data[elem].key {
        largest = left
    }

    right, err := q.Right(elem)
    if err == nil && q.data[right].key > q.data[largest].key {
        largest = right
    }
    // if one of child is bigger than parent - swap elements
    if largest != elem {
        q.data[elem], q.data[largest] = q.data[largest], q.data[elem]
        q.MaxHeapify(largest)
    }
```

```
}

func (q *PrQueue) ExtractMax() (PrQueueNode, error) {
    var retval PrQueueNode
    if q.size < 1 {
        return retval, errors.New("priority queue is empty")
    }
    retval, _ = q.Max()
    q.data[0] = q.data[q.size-1]
    q.size--
    q.MaxHeapify(0)
    return retval, nil
}

// function to increase key in priority queue
func (q *PrQueue) IncreaseKey(ind, newKey int) {
    q.data[ind].key = newKey

    // ind > 0, because appeared the loop, if ind = 0, (Parent(0) = 0)
    for ind > 0 {
        parent := q.Parent(ind)
        if q.data[ind].key > q.data[parent].key {
            q.data[ind], q.data[parent] = q.data[parent], q.data[ind]
            ind = parent
        } else {
            break
        }
    }
}

func (q *PrQueue) InsertKey(value, key int) {
    newNode := PrQueueNode{
        value: value,
        key:   key,
    }
    q.data = append(q.data, newNode)
    q.IncreaseKey(q.size, key)
    q.size++
}
```

Ресурсы

- What is Priority Queue | Introduction to Priority Queue - GeeksforGeeks
- Priority queue - Wikipedia