* 1. Создание Бинарной кучи на питоне

class BinaryHeap:

def \_\_init\_\_(self):

self.heap = []

def parent(self, i):

return (i-1)//2

def left\_child(self, i):

return 2\*i + 1

def right\_child(self, i):

return 2\*i + 2

def insert(self, key):

self.heap.append(key)

self.\_sift\_up(len(self.heap)-1)

def extract\_min(self):

if not self.heap:

return None

min\_val = self.heap[0]

self.heap[0] = self.heap[-1]

self.heap.pop()

if self.heap:

self.\_sift\_down(0)

return min\_val

def \_sift\_up(self, i):

while i > 0 and self.heap[self.parent(i)] > self.heap[i]:

self.heap[self.parent(i)], self.heap[i] = self.heap[i], self.heap[self.parent(i)]

i = self.parent(i)

def \_sift\_down(self, i):

min\_index = i

left = self.left\_child(i)

right = self.right\_child(i)

if left < len(self.heap) and self.heap[left] < self.heap[min\_index]:

min\_index = left

if right < len(self.heap) and self.heap[right] < self.heap[min\_index]:

min\_index = right

if i != min\_index:

self.heap[i], self.heap[min\_index] = self.heap[min\_index], self.heap[i]

self.\_sift\_down(min\_index)

* 1. Создание биноминального списка на Python

class BinomialNode:

def \_\_init\_\_(self, key):

self.key = key

self.degree = 0

self.parent = None

self.child = None

self.sibling = None

class BinomialHeap:

def \_\_init\_\_(self):

self.head = None

def merge(self, h2):

# Слияние двух биномиальных куч

new\_head = self.\_merge\_lists(self.head, h2.head)

if not new\_head:

return

prev = None

curr = new\_head

next\_node = curr.sibling

while next\_node:

if (curr.degree != next\_node.degree or

(next\_node.sibling and next\_node.sibling.degree == curr.degree)):

prev = curr

curr = next\_node

elif curr.key <= next\_node.key:

curr.sibling = next\_node.sibling

self.\_link\_trees(next\_node, curr)

else:

if not prev:

new\_head = next\_node

else:

prev.sibling = next\_node

self.\_link\_trees(curr, next\_node)

curr = next\_node

next\_node = curr.sibling

self.head = new\_head

def \_link\_trees(self, child, parent):

child.parent = parent

child.sibling = parent.child

parent.child = child

parent.degree += 1

def insert(self, key):

new\_heap = BinomialHeap()

new\_heap.head = BinomialNode(key)

self.merge(new\_heap)

* 1. Создание Кучи Фибоначчи на python

class FibonacciNode:

def \_\_init\_\_(self, key):

self.key = key

self.degree = 0

self.parent = None

self.child = None

self.left = self

self.right = self

self.marked = False

class FibonacciHeap:

def \_\_init\_\_(self):

self.min\_node = None

self.node\_count = 0

def insert(self, key):

node = FibonacciNode(key)

if not self.min\_node:

self.min\_node = node

else:

self.\_add\_to\_root\_list(node)

if key < self.min\_node.key:

self.min\_node = node

self.node\_count += 1

def \_add\_to\_root\_list(self, node):

node.left = self.min\_node

node.right = self.min\_node.right

self.min\_node.right.left = node

self.min\_node.right = node

def extract\_min(self):

z = self.min\_node

if z:

# Добавляем детей в корневой список

if z.child:

child = z.child

while True:

next\_child = child.right

self.\_add\_to\_root\_list(child)

child.parent = None

child = next\_child

if child == z.child:

break

# Удаляем z из корневого списка

z.left.right = z.right

z.right.left = z.left

if z == z.right:

self.min\_node = None

else:

self.min\_node = z.right

self.\_consolidate()

self.node\_count -= 1

return z.key if z else None

def \_consolidate(self):

degree\_table = {}

nodes = []

current = self.min\_node

# Собираем все корневые узлы

while True:

nodes.append(current)

current = current.right

if current == self.min\_node:

break

for node in nodes:

degree = node.degree

while degree in degree\_table:

other = degree\_table[degree]

if node.key > other.key:

node, other = other, node

self.\_link(other, node)

del degree\_table[degree]

degree += 1

degree\_table[degree] = node

# Восстанавливаем min\_node

self.min\_node = None

for node in degree\_table.values():

if not self.min\_node or node.key < self.min\_node.key:

self.min\_node = node

* 1. Создание Хеш таблицы на Python

class HashTable:

def \_\_init\_\_(self, capacity=8, load\_factor=0.75):

self.capacity = capacity

self.load\_factor = load\_factor

self.size = 0

self.buckets = [[] for \_ in range(capacity)]

def \_hash(self, key):

return hash(key) % self.capacity

def put(self, key, value):

if self.size / self.capacity >= self.load\_factor:

self.\_resize()

index = self.\_hash(key)

bucket = self.buckets[index]

for i, (k, v) in enumerate(bucket):

if k == key:

bucket[i] = (key, value)

return

bucket.append((key, value))

self.size += 1

def get(self, key):

index = self.\_hash(key)

bucket = self.buckets[index]

for k, v in bucket:

if k == key:

return v

return None

def \_resize(self):

old\_buckets = self.buckets

self.capacity \*= 2

self.buckets = [[] for \_ in range(self.capacity)]

self.size = 0

for bucket in old\_buckets:

for key, value in bucket:

self.put(key, value)