**Бинарная куча на Python**

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\_max(self):  
 if not self.heap:  
 return None  
 max\_val = self.heap[0]  
 self.heap[0] = self.heap[-1]  
 self.heap.pop()  
 self.\_sift\_down(0)  
 return max\_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):  
 max\_index = i  
 left = self.left\_child(i)  
  
 if left < len(self.heap) and self.heap[left] > self.heap[max\_index]:  
 max\_index = left  
  
 right = self.right\_child(i)  
 if right < len(self.heap) and self.heap[right] > self.heap[max\_index]:  
 max\_index = right  
  
 if i != max\_index:  
 self.heap[i], self.heap[max\_index] = self.heap[max\_index], self.heap[i]  
 self.\_sift\_down(max\_index)

**Бинарная куча на Java**

**import java.util.ArrayList;**

**import java.util.List;**

**public class BinaryHeap {**

**private List<Integer> heap;**

**public BinaryHeap() {**

**heap = new ArrayList<>();**

**}**

**private int parent(int i) { return (i - 1) / 2; }**

**private int leftChild(int i) { return 2 \* i + 1; }**

**private int rightChild(int i) { return 2 \* i + 2; }**

**public void insert(int key) {**

**heap.add(key);**

**siftUp(heap.size() - 1);**

**}**

**public Integer extractMax() {**

**if (heap.isEmpty()) return null;**

**int result = heap.get(0);**

**heap.set(0, heap.get(heap.size() - 1));**

**heap.remove(heap.size() - 1);**

**siftDown(0);**

**return result;**

**}**

**private void siftUp(int i) {**

**while (i > 0 && heap.get(parent(i)) < heap.get(i)) {**

**swap(parent(i), i);**

**i = parent(i);**

**}**

**}**

**private void siftDown(int i) {**

**int maxIndex = i;**

**int left = leftChild(i);**

**if (left < heap.size() && heap.get(left) > heap.get(maxIndex))**

**maxIndex = left;**

**int right = rightChild(i);**

**if (right < heap.size() && heap.get(right) > heap.get(maxIndex))**

**maxIndex = right;**

**if (i != maxIndex) {**

**swap(i, maxIndex);**

**siftDown(maxIndex);**

**}**

**}**

**private void swap(int i, int j) {**

**int temp = heap.get(i);**

**heap.set(i, heap.get(j));**

**heap.set(j, temp);**

**}**

**}**

**куча Фибоначчи на Python**

class BinomialTree:  
 def \_\_init\_\_(self, key):  
 self.key = key  
 self.children = []  
 self.order = 0  
  
 def add\_child(self, child):  
 self.children.append(child)  
 self.order += 1  
  
  
class BinomialHeap:  
 def \_\_init\_\_(self):  
 self.trees = []  
  
 def merge(self, other):  
 self.trees.extend(other.trees)  
 self.trees.sort(key=lambda tree: tree.order)  
 self.\_combine\_trees()  
  
 def insert(self, key):  
 new\_heap = BinomialHeap()  
 new\_tree = BinomialTree(key)  
 new\_heap.trees.append(new\_tree)  
 self.merge(new\_heap)  
  
 def get\_min(self):  
 if not self.trees:  
 return None  
 return min(tree.key for tree in self.trees)  
  
 def extract\_min(self):  
 if not self.trees:  
 return None  
  
 min\_tree = min(self.trees, key=lambda tree: tree.key)  
 self.trees.remove(min\_tree)  
  
 new\_heap = BinomialHeap()  
 new\_heap.trees = min\_tree.children  
 self.merge(new\_heap)  
  
 return min\_tree.key  
  
 def \_combine\_trees(self):  
 if len(self.trees) <= 1:  
 return  
  
 i = 0  
 while i < len(self.trees) - 1:  
 current = self.trees[i]  
 next\_tree = self.trees[i + 1]  
  
 if current.order == next\_tree.order:  
 if current.key <= next\_tree.key:  
 current.add\_child(next\_tree)  
 self.trees.pop(i + 1)  
 else:  
 next\_tree.add\_child(current)  
 self.trees[i] = next\_tree  
 self.trees.pop(i + 1)  
 else:  
 i += 1

**куча Фибоначчи на C++**

#include <vector>

#include <algorithm>

class BinomialTree {

public:

int key;

std::vector<BinomialTree\*> children;

int order;

BinomialTree(int k) : key(k), order(0) {}

void addChild(BinomialTree\* child) {

children.push\_back(child);

order++;

}

};

class BinomialHeap {

private:

std::vector<BinomialTree\*> trees;

void combineTrees() {

if (trees.size() <= 1) return;

for (size\_t i = 0; i < trees.size() - 1; ) {

BinomialTree\* current = trees[i];

BinomialTree\* next = trees[i + 1];

if (current->order == next->order) {

if (current->key <= next->key) {

current->addChild(next);

trees.erase(trees.begin() + i + 1);

} else {

next->addChild(current);

trees[i] = next;

trees.erase(trees.begin() + i + 1);

}

} else {

i++;

}

}

}

public:

void insert(int key) {

BinomialHeap newHeap;

BinomialTree\* newTree = new BinomialTree(key);

newHeap.trees.push\_back(newTree);

merge(newHeap);

}

void merge(BinomialHeap& other) {

trees.insert(trees.end(), other.trees.begin(), other.trees.end());

std::sort(trees.begin(), trees.end(),

[](BinomialTree\* a, BinomialTree\* b) { return a->order < b->order; });

combineTrees();

}

int getMin() {

if (trees.empty()) return -1;

int minVal = trees[0]->key;

for (auto tree : trees) {

if (tree->key < minVal) {

minVal = tree->key;

}

}

return minVal;

}

};

**хеш-таблицы на Python**

class FibonacciNode:  
 def \_\_init\_\_(self, key):  
 self.key = key  
 self.parent = None  
 self.child = None  
 self.left = self  
 self.right = self  
 self.degree = 0  
 self.marked = False  
  
  
class FibonacciHeap:  
 def \_\_init\_\_(self):  
 self.min\_node = None  
 self.count = 0  
  
 def insert(self, key):  
 node = FibonacciNode(key)  
 if self.min\_node is None:  
 self.min\_node = node  
 else:  
 self.\_add\_to\_root\_list(node)  
 if key < self.min\_node.key:  
 self.min\_node = node  
 self.count += 1  
  
 def extract\_min(self):  
 z = self.min\_node  
 if z is not None:  
 # Добавляем детей в корневой список  
 if z.child is not None:  
 child = z.child  
 while True:  
 next\_child = child.right  
 self.\_add\_to\_root\_list(child)  
 child.parent = None  
 if next\_child == z.child:  
 break  
 child = next\_child  
  
 # Удаляем z из корневого списка  
 self.\_remove\_from\_root\_list(z)  
  
 if z == z.right:  
 self.min\_node = None  
 else:  
 self.min\_node = z.right  
 self.\_consolidate()  
  
 self.count -= 1  
 return z.key if z else None  
  
 def \_add\_to\_root\_list(self, node):  
 if self.min\_node is None:  
 return  
 node.left = self.min\_node.left  
 node.right = self.min\_node  
 self.min\_node.left.right = node  
 self.min\_node.left = node  
  
 def \_remove\_from\_root\_list(self, node):  
 node.left.right = node.right  
 node.right.left = node.left  
  
 def \_consolidate(self):  
 # Консолидация деревьев одинаковой степени  
 pass

**хеш-таблицы на Java**

public class HashMap<K, V> {

private static class Entry<K, V> {

K key;

V value;

Entry<K, V> next;

Entry(K key, V value) {

this.key = key;

this.value = value;

}

}

private Entry<K, V>[] table;

private int capacity;

@SuppressWarnings("unchecked")

public HashMap(int capacity) {

this.capacity = capacity;

table = new Entry[capacity];

}

private int hash(K key) {

return Math.abs(key.hashCode()) % capacity;

}

public void put(K key, V value) {

int index = hash(key);

Entry<K, V> newEntry = new Entry<>(key, value);

if (table[index] == null) {

table[index] = newEntry;

} else {

Entry<K, V> current = table[index];

while (current.next != null) {

if (current.key.equals(key)) {

current.value = value;

return;

}

current = current.next;

}

if (current.key.equals(key)) {

current.value = value;

} else {

current.next = newEntry;

}

}

}

public V get(K key) {

int index = hash(key);

Entry<K, V> current = table[index];

while (current != null) {

if (current.key.equals(key)) {

return current.value;

}

current = current.next;

}

return null;

}

}

**Производительность операций:**

| **Структура данных** | **Вставка** | **Поиск** | **Удаление** | **Особенности** |
| --- | --- | --- | --- | --- |
| Бинарная куча | O(log n) | O(1) макс | O(log n) | Простая реализация |
| Биномиальная куча | O(1) | O(log n) | O(log n) | Быстрое слияние |
| Куча Фибоначчи | O(1) | O(1) | O(log n) | Лучшая амортизация |
| Хеш-таблица | O(1) | O(1) | O(1) | Зависит от хеш-функции |

**Выводы**

1. **Бинарные кучи** наиболее просты в реализации и подходят для базовых задач с приоритетами
2. **Биномиальные кучи** эффективны при частых операциях слияния
3. **Кучи Фибоначчи** обеспечивают наилучшую амортизированную сложность для сложных алгоритмов
4. **Хеш-таблицы** предоставляют оптимальную производительность для операций поиска и вставки

Выбор конкретной реализации зависит от языка программирования, требований к производительности и специфики решаемой задачи. Python предпочтительнее для прототипирования, C++ для высокопроизводительных систем, Java для кроссплатформенных enterprise-решений.