

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ «МИСИС»**

**ИНСТИТУТ КОМПЬЮТЕРНЫХ НАУК (ИKN)
КАФЕДРА АВТОМАТИЗИРОВАННЫХ СИСТЕМ УПРАВЛЕНИЯ (АСУ)**

Курс «Алгоритмы дискретной математики»

Курсовая работа на тему:

«Кучи. Представления кучи в компьютере, алгоритм окучивания массива.»

Выполнил:

Студент группы БИВТ-23-1

Адиев Марат Рустемович

Проверил:

Гласов Александр Васильевич

Оглавление

Введение.....	3
Теоретические основы.....	3
Определение и свойства кучи	3
Представление кучи в памяти компьютера	4
Основные операции с кучей.....	5
Вставка элемента (Insert)	5
Извлечение корневого элемента (ExtractRoot).....	5
Просмотр корневого элемента (Peek).....	6
Алгоритм окучивания массива	6
Принцип работы алгоритма Heapify.....	6
Построение кучи из массива	7
Алгоритм sift-down (просеивание вниз)	8
Анализ временной сложности	8
Сортировка кучей (Heap Sort).....	9
Преимущества сортировки кучей	9
Применения куч в программировании.....	10
Приоритетные очереди	10
Алгоритм Дейкстры	10
Кодирование Хаффмана.....	10
Управление памятью	10
Анализ производительности.....	10
Временная сложность операций	10
Пространственная сложность	11
Сравнение с другими структурами данных	11
Практическая реализация.....	11
Запуск примеров	12
Запуск веб-интерфейса.....	12
Заключение	14
Примечание о создании материалов	14
Источники.....	14

Введение

Куча (Heap) — это один из базовых типов структур данных. Она представляет собой особый вид двоичного дерева с определённым порядком элементов.

Куча играет важную роль в алгоритмах сортировки, управлении памятью и реализации очередей с приоритетами.

Куча — это полное двоичное дерево, в котором значение каждого родительского узла больше (в случае максимальной кучи) или меньше (в случае минимальной кучи), чем значения его дочерних узлов. Это свойство называется свойством кучи и является ключевым для всех операций с этой структурой данных.

Изучение куч представляет большой интерес, поскольку они широко применяются в различных областях программирования.

Алгоритм сортировки кучей (heapsort) обеспечивает гарантированную временную сложность $O(n \log n)$ в худшем случае, что делает его надёжным выбором для критически важных приложений.

Кучи используются в алгоритмах Дейкстры для поиска кратчайших путей, в алгоритмах сжатия данных (кодирование Хаффмана), а также в операционных системах для планирования задач.

Особого внимания заслуживает алгоритм окучивания массива (heapify), который позволяет преобразовать произвольный массив в структуру кучи за линейное время $O(n)$. Этот алгоритм является основой для эффективной реализации сортировки кучей и построения приоритетных очередей.

Теоретические основы

Определение и свойства кучи

Куча — это полное бинарное дерево, которое удовлетворяет свойству кучи. Полное бинарное дерево означает, что все уровни дерева заполнены, кроме, возможно, последнего уровня, который заполняется слева направо.

Существуют два основных типа куч:

- **Максимальная куча (max-heap)**: в каждом узле значение больше или равно значениям всех его потомков.

- **Минимальная куча (min-heap)**: в каждом узле значение меньше или равно значениям всех его потомков.

Представление кучи в памяти компьютера

$H = \log_2(n + 1)$ Одним из ключевых достоинств кучи является возможность её эффективного представления в виде массива. Благодаря тому, что бинарное дерево является полным, узлы в нём можно пронумеровать по уровням, начиная с корня, слева направо.

При представлении кучи в виде массива используются следующие соотношения для элемента с индексом i :

— **родительский элемент**: индекс $(i - 1) / 2$ (целочисленное деление);

— **левый дочерний элемент**: индекс $2i + 1$;

— **правый дочерний элемент**: индекс $2i + 2$.

Такое представление обеспечивает:

— **эффективное использование памяти**: не требуется дополнительная память для хранения указателей;

— **локальность данных**: элементы, близкие в дереве, располагаются рядом в памяти;

— **простоту навигации**: переход между родителями и детьми осуществляется с помощью простых арифметических операций.

Высоту полной бинарной кучи можно выразить формулой:

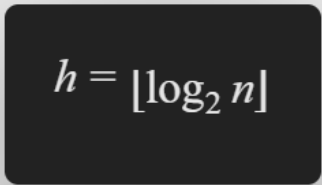

$$h = \lfloor \log_2 n \rfloor$$

Рис 1. Формула высоты полной бинарной кучи

Основные операции с кучей

Вставка элемента (Insert)

```
func (h *Heap) Insert(item int) error {
    if h.Size >= h.Capacity {
        return errors.New("переполнение кучи: достигнута максимальная емкость")
    }
    h.Items[h.Size] = item
    h.Size++
    h.SiftUp(h.Size - 1)

    return nil
}
```

Рис 2. Insert

Эта функция добавляет элемент в конец массива, после чего восстанавливает свойство кучи с помощью **sift-up**. Временная сложность операции — $O(\log n)$, что совпадает с данными таблицы сложностей ниже.

Извлечение корневого элемента (ExtractRoot)

```
func (h *Heap) ExtractRoot() (int, error) {
    if h.Size <= 0 {
        return 0, errors.New("куча пуста")
    }
    root := h.Items[0]
    h.Items[0] = h.Items[h.Size-1]
    h.Size--
    h.SiftDown(0)

    return root, nil
}
```

Рис 3. ExtractRoot

Корневой элемент меняется местами с последним, удаляется,

а затем массив "просеивается" вниз с помощью **sift-down**.

Временная сложность — **$O(\log n)$** .

Просмотр корневого элемента (Peek)

```
func (h *Heap) Peek() (int, error) {  
    if h.Size <= 0 {  
        return 0, errors.New("куча пуста")  
    }  
    return h.Items[0], nil  
}
```

Рис 4. Peek

Операция возвращает значение в корне без каких-либо преобразований, поэтому выполняется за **$O(1)$** .

Алгоритм окучивания массива

Принцип работы алгоритма Heapify

Алгоритм heapify позволяет преобразовать произвольный массив в структуру кучи. Существует два способа реализации этого алгоритма:

1. **Метод «снизу вверх»** (bottom-up heapify) — начинается с последнего элемента, который не является листом, и применяется операция sift-down к каждому элементу, двигаясь к корню. Этот метод более эффективен и имеет временную сложность $O(n)$.
2. **Метод «сверху вниз»** (top-down heapify) — последовательно вставляет каждый элемент в кучу, используя операцию sift-up. Временная сложность этого метода составляет $O(n \log n)$.

```
func Heapify(arr []int, heapType HeapType) {
    n := len(arr)
    h := &Heap{
        Items:    arr,
        Type:      heapType,
        Size:      n,
        Capacity: n,
    }

    h.BuildHeap()
}
```

Рис 5. *Heapify*

В проекте используется подход «снизу вверх»: вызов *BuildHeap* выполняет **sift-down** для всех внутренних узлов, что обеспечивает линейную сложность **O(n)**.

Построение кучи из массива

```
func NewHeapFromArray(arr []int, heapType HeapType) *Heap {
    capacity := len(arr)
    heap := &Heap{
        Items:    make([]int, capacity),
        Type:      heapType,
        Size:      capacity,
        Capacity: capacity,
    }

    copy(heap.Items, arr)

    heap.BuildHeap()

    return heap
}
```

Рис 6. *NewHeapFromArray*

Эта функция создаёт кучу напрямую из существующего массива и вызывает *BuildHeap*, что делает преобразование массива в кучу за **O(n)**.

Алгоритм sift-down (просеивание вниз)

```
func (h *Heap) SiftDown(index int) {
    largest := index
    smallest := index

    for {
        leftChildIndex := h.LeftChild(index)
        rightChildIndex := h.RightChild(index)

        if h.Type == MaxHeap {
            if leftChildIndex < h.Size && h.Items[leftChildIndex] > h.Items[largest] {
                largest = leftChildIndex
            }

            if rightChildIndex < h.Size && h.Items[rightChildIndex] > h.Items[largest] {
                largest = rightChildIndex
            }

            if largest != index {
                h.Swap(index, largest)
                index = largest
            } else {
                break
            }
        } else {
            if leftChildIndex < h.Size && h.Items[leftChildIndex] < h.Items[smallest] {
                smallest = leftChildIndex
            }

            if rightChildIndex < h.Size && h.Items[rightChildIndex] < h.Items[smallest] {
                smallest = rightChildIndex
            }

            if smallest != index {
                h.Swap(index, smallest)
                index = smallest
            } else {
                break
            }
        }
    }
}
```

Рис 7. SiftDown

SiftDown восстанавливает свойство кучи, просеивая элемент вниз по дереву. В худшем случае функция проходит высоту дерева, что даёт $O(\log n)$.

Анализ временной сложности

Временная сложность алгоритма *heapify* составляет $O(n)$. Это объясняется тем, что: листья не требуют обработки, узлы на предпоследнем уровне могут просеиваться

максимум на 1 уровень, а количество узлов увеличивается экспоненциально снизу вверх.

Сортировка кучей (Heap Sort)

```
func HeapSort(arr []int) {
    n := len(arr)
    if n <= 1 {
        return
    }

    Heapify(arr, MaxHeap)

    for i := n - 1; i > 0; i-- {
        arr[0], arr[i] = arr[i], arr[0]

        h := &Heap{
            Items:  arr,
            Type:    MaxHeap,
            Size:    i,
            Capacity: n,
        }
        h.SiftDown(0)
    }
}
```

Рис 8. HeapSort

Алгоритм стартует с того, что массив преобразуется в структуру данных, называемую «кучей». Затем он последовательно перемещает элемент с наибольшим значением в конец массива.

На каждой итерации требуется операция, сложность которой составляет $O(\log n)$. Таким образом, общая сложность алгоритма равна $O(n \log n)$.

Преимущества сортировки кучей

Гарантированная эффективность: в худшем случае - $O(n \log n)$.

Сортировка выполняется на месте, требуется всего $O(1)$ дополнительной памяти.

Нестабильный алгоритм: элементы, имеющие одинаковое значение, могут менять своё положение относительно друг друга.

Применения куч в программировании

Приоритетные очереди

Структуры данных, подобные кучам, представляют собой естественное воплощение приоритетных очередей. В таких структурах элементы обрабатываются в соответствии с их приоритетом, а не в порядке поступления.

Алгоритм Дейкстры

В процессе поиска кратчайших путей применяется структура данных, «куча», позволяет быстро определить вершину с наименьшим расстоянием.

Кодирование Хаффмана

При разработке эффективного префиксного кода, использование кучи позволяет определить узлы с наименьшей частотой для объединения.

Управление памятью

В некоторых версиях сборщиков мусора применяется механизм, который позволяет управлять приоритетами объектов при освобождении памяти.

Анализ производительности

Временная сложность операций

Операция	Временная сложность
Построение кучи (heapify	$O(n)$
Вставка элемента	$O(\log n)$
Извлечение корня	$O(\log n)$
Просмотр корня	$O(1)$
Поиск элемента	$O(n)$

Таблица 1. Временная сложность

Пространственная сложность

Массив, представляющий собой кучу, занимает $O(n)$ памяти для хранения n элементов. При этом для выполнения операций требуется $O(1)$ дополнительной памяти, что делает кучу весьма экономичной с точки зрения использования ресурсов.

Сравнение с другими структурами данных

В отличие от сбалансированных бинарных деревьев поиска, кучи имеют ряд преимуществ:

- их реализация проще;
- данные расположены более локально;
- операции вставки и удаления элементов с экстремальными значениями выполняются быстрее;
- однако поиск произвольных элементов не может быть эффективно реализован.

Практическая реализация

На языке Go была создана полноценная реализация структуры данных «куча», которая включает в себя все основные операции. В частности, были реализованы алгоритм окучивания массива, сортировка кучей, различные типы куч. Также была разработана система тестирования и бенчмарков, а для визуализации работы алгоритмов была создана система визуализации.

Код был организован в виде модулей, что обеспечивает его читаемость, возможность тестирования и повторного использования компонентов.

Запуск примеров

```
# Базовое использование кучи  
go run examples/basic/basic-usage.go  
  
# Демонстрация сортировки кучей  
go run examples/sorting/sorting-demo.go  
  
# Тесты производительности  
go run examples/performance/performance-  
test.go
```

Рис 9. PowerShell

Запуск веб-интерфейса

```
go run cmd/web/main.go
```

Рис 10. PowerShell

После запуска веб-интерфейс будет доступен по адресу <http://localhost:8080>

С помощью интерфейса можно в режиме реального времени добавлять и удалять элементы, а также следить за тем, как работают алгоритмы, и измерять, сколько времени занимает выполнение операций. (На скрине показаны все варианты)

Визуализация кучи

Курсовая работа: "Кучи. Представления кучи в компьютере, алгоритм очукивания массива"

Управление кучей

Вставить

Извлечь корень

Очистить

Изменить тип кучи

Создать кучу из массива

Куча создана из массива

Текущее состояние кучи: Минимальная куча

Представление в виде дерева

Представление в виде массива

Связи между узлами

```
      1
     / \
    3   10
   / \  / \
  5  7 20 15
 / \
8  9 12
```

Сортировка кучей

По возрастанию

Сортировать

Рис 11. Визуализация

Заключение

Куча — это структура данных, которая эффективно решает задачи, связанные с управлением приоритетами. Алгоритм `heapify` позволяет преобразовать любые данные в упорядоченную структуру за линейное время. Сортировка кучей обеспечивает производительность $O(n \log n)$.

Практическая реализация и визуализация подтверждают теоретические оценки и делают изучение этой структуры более понятным и доступным.

Примечание о создании материалов

Файлы **`algorithms.md`**, **`complexity-analysis.md`**, а также HTML-шаблон **`index.html`** были созданы с ИИ, чтобы облегчить понимание алгоритмов и наглядно представить работу кучи.

Источники

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. "Алгоритмы: построение и анализ". 3-е изд. М.: Вильямс, 2013.
2. Вирт Н. "Алгоритмы и структуры данных". М.: ДМК Пресс, 2010.
3. Исходный код проекта: [GitHub - Soln1shko/heap-coursework](https://github.com/Soln1shko/heap-coursework)