Министерство науки и высшего образования РФ

ФГАОУ ВПО

Национальный исследовательский технологический университет «МИСИС»

Институт компьютерных наук (ИКН)

Кафедра Инфокоммуникационных технологий (ИКТ)

Отчет по теме «Кучи. Представление кучи в компьютере. Алгоритм окучивания массива»

по дисциплине «Комбинаторика и теория графов»

Выполнила:

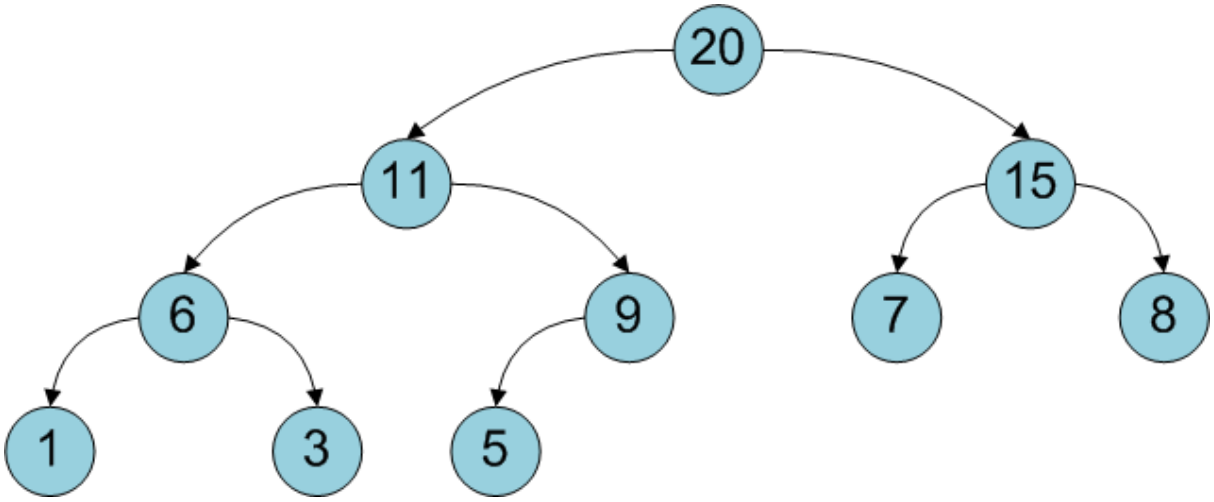
Студентка БИВТ-23-6

Мушкарина В.А.

Москва 2024

Введение

Куча (или бинарная куча) — это специализированная структура данных, которая представляет собой полное двоичное дерево, удовлетворяющее свойству кучи. Это свойство заключается в том, что для каждой вершины (узла) дерева значение этой вершины должно быть больше (в случае max-кучи) или меньше (в случае min-кучи) значений её дочерних вершин. Кучи часто используются в алгоритмах сортировки, таких как сортировка кучей (heap sort), а также в алгоритмах, требующих эффективного извлечения минимального или максимального элемента.



Основные понятия

Куча – это специализированная двоичная структура данных, которая удовлетворяет свойству кучи.

Существует два основных типа куч: макс-куча и мин-куча

- Макс-куча: Для любой вершины (i), значение в этой вершине больше или равно значениям в её дочерних вершинах.
- Мин-куча: Для любой вершины (i), значение в этой вершине меньше или равно значениям в её дочерних вершинах.

Алгоритм окучивания в массиве (Heapify)

Алгоритм окучивания (heapify) используется для преобразования массива в кучу. Он гарантирует, что элемент в заданном узле удовлетворяет свойству кучи относительно своих потомков.

Представление кучи в массиве

Кучи часто представляются в виде массивов, где:

- Корень находится в индексе 0.
- Для любой вершины с индексом (i):
  1. Левый дочерний элемент находится в индексе (2i+1)
  2. Правый дочерний элемент находится в индексе (2i+2)
  3. Родительский элемент находится в индексе (i - 1) / 2

Основные операции кучи

1. **Вставка элемента.** Новый элемент добавляется в конец массива, затем перемещается вверх по дереву, пока не будет соблюдено свойство кучи.
2. **Удаление корня.** Корневой элемент (максимум или минимум) удаляется, а последний элемент перемещается на его место. После этого дерево "окучивается" для восстановления свойства кучи.
3. **Построение кучи.** Исходный массив преобразуется в кучу за $O(n)$

Алгоритм окучивания

Идея алгоритма:

1. Проверяем, нарушает ли элемент свойства кучи (для макс-кучи — родитель должен быть больше потомков, для мин-кучи — меньше).
2. Если нарушает, меняем местами родительский узел с наибольшим (или наименьшим) из потомков.
3. Продолжаем проверять это свойство рекурсивно вниз по дереву.

Алгоритм сортировки кучей

1) Построение кучи

Чтобы построить кучу из массива:

- Определяем первый узел с потомками: индекс (n//2) – 1
- Для каждого узла, начиная с (n//2) – 1 до 0, применяем функцию окучивания

```
def build_heap(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
```

2) Функция heapify

Процесс окучивания поддерживает свойство кучи для поддерева с корнем в узле i.

Шаги:

- Сравниваем узел с его дочерними.
- Если дочерний узел нарушает свойство кучи, обмениваем его с текущим узлом.
- Повторяем процесс для изменённого поддерева.

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)
```

3) Сортировка (Heap Sort)

Построить кучу.

- Извлечь корень (максимальный элемент), обменяв его с последним элементом массива.
- Уменьшить размер массива (исключая последний элемент) и повторить окучивание

```
def heap_sort(arr):
    n = len(arr)
    build_heap(arr)
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        heapify(arr, i, 0)
```

Сравнение с другими алгоритмами

Тип сортировки	Сортировка кучей	Быстрая сортировка	Сортировка слиянием	Пузырьковая сортировка
Тип алгоритма	Основан на куче	Разбиение и обработка	Разбиение и обработка	Простые перестановки
Сложность	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$	$O(n^2)$
Стабильность	Нестабильный	Нестабильный	Стабильный	Стабильный
Работа с большими данными	Хорошо подходит	Эффективен, но требует обработки стека	Подходит, но использует много памяти	Не подходит
Простота реализации	Средне	Высокая	Высокая	Очень простая

Преимущества

- **Эффективность памяти:** Куча не требует дополнительной памяти для хранения временных массивов.
- **Гарантированная сложность $O(n \log(n))$:** Даже в худшем случае
- **Простота реализации**: Легко реализуется для массивов.

Недостатки

- **Нестабильность:** Если два элемента равны, их относительный порядок может измениться.
- **Более медленная в среднем, чем Быстрая сортировка:** Quick Sort обычно работает быстрее из-за меньшего количества операций по сравнению с Heap Sort.

- **Неоптимально для больших данных**: Алгоритмы вроде сортировки слиянием, использующие линейную память, могут быть быстрее за счёт кэш-оптимизации.

Задача

Условие: Дан массив чисел: [4, 10, 3, 5, 1].

1. Постройте из него макс-кучу.
2. Проведите одну итерацию сортировки, извлекая максимальный элемент из кучи.

Решение:

1. Построение макс-кучи:

- Начинаем с последнего родительского узла.
- Применяем heapify: сравниваем узел с потомками, меняем местами с наибольшим и повторяем для поддерева.
- Итог: массив становится макс-кучей.

2. Извлечение максимального элемента:

- Меняем корень (максимальный элемент) с последним элементом.
- Удаляем последний элемент из массива.
- Восстанавливаем свойство кучи на оставшихся элементах через heapify.

```
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left] > arr[largest]:
        largest = left

    if right < n and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)


def build_max_heap(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
```

```
def extract_max(arr):
    n = len(arr)
    if n == 0:
        return None

    arr[0], arr[n - 1] = arr[n - 1], arr[0]

    heapify(arr, n - 1, 0)


    return arr.pop()

# Исходный массив
arr = [4, 10, 3, 5, 1]

# Построение макс-кучи
print("Исходный массив:", arr)
build_max_heap(arr)
print("Макс-куча:", arr)
```

```
# Построение макс-кучи
print("Исходный массив:", arr)
build_max_heap(arr)
print("Макс-куча:", arr)

# Извлечение максимального элемента
max_element = extract_max(arr)
print("Извлечённый максимум:", max_element)
print("Куча после извлечения максимума:", arr)
```

Объяснение

1. Функция `heapify`

1. Считаем текущий узел самым большим.

2. Находим индексы левого и правого потомков.

3. Сравниваем текущий узел с потомками.

4. Если один из потомков больше текущего узла, меняем их местами.

5. Рекурсивно вызываем heapify для дочернего узла, с которым был обмен, чтобы восстановить свойства кучи.

2. Функция `build_max_heap`

1. Начинаем с последнего родительского узла (i \(n // 2 - 1\)).

2. Для каждого узла от последнего родителя до корня вызываем heapify, чтобы построить кучу.

3. Функция `extract_max`

1. Меняем местами корень с последним элементом.
2. Уменьшаем размер кучи, удаляя последний элемент.
3. Применяем heapify к новому корню, чтобы восстановить свойства кучи.

Результат:

```
Исходный массив: [4, 10, 3, 5, 1]
Макс-куча: [10, 5, 3, 4, 1]
Извлечённый максимум: 10
Куча после извлечения максимума: [5, 4, 3, 1]
```

Применение:

1. Сортировка данных
2. Реализация очередей с приоритетами
3. Поиск минимальных и максимальных элементов
4. Сжатие данных
5. Компьютерные игры

Заключение:

Алгоритмы работы с кучей, такие как Heap Sort, представляют собой эффективные методы сортировки и управления данными. Они гарантируют стабильную производительность независимо от исходных данных и требуют минимального использования дополнительной памяти. Это делает их особенно полезными в системах с ограниченными ресурсами.

Структура данных «Куча» является основой для других алгоритмов и применяется во многих областях, где требуется быстрая работа с приоритетами или эффективное управление большими массивами данных.