Двоичная куча — это законченное двоичное дерево, в котором элементы хранятся в особом порядке: значение в родительском узле больше (или меньше) значений в его двух дочерних узлах. Первый вариант называется max-heap, а второй — min-heap. Куча может быть представлена двоичным деревом или массивом.

Почему для двоичной кучи используется представление на основе массива ?

Поскольку двоичная куча — это законченное двоичное дерево, ее можно легко представить в виде массива, а представление на основе массива является эффективным с точки зрения расхода памяти. Если родительский узел хранится в индексе I, левый дочерний элемент может быть вычислен как 2 *I + 1, а правый дочерний элемент — как 2* I + 2 (при условии, что индексирование начинается с 0).

Алгоритм пирамидальной сортировки в порядке по возрастанию:

1. Постройте max-heap из входных данных.
2. На данном этапе самый большой элемент хранится в корне кучи. Замените его на последний элемент кучи, а затем уменьшите ее размер на 1. Наконец, преобразуйте полученное дерево в max-heap с новым корнем.
3. Повторяйте вышеуказанные шаги, пока размер кучи больше 1.

Алгоритм пирамидальной сортировки на Java
Алгоритм сегментирует массив на отсортированный и неотсортированный. Неотсортированный сегмент преобразовывается в кучу (heap), что позволяет эффективно эффективно определить самый большой элемент.

```
//вернуть левого потомка `A[i]`
private static int LEFT(int i) {

return (2 * i + 1);
}

//вернуть правого потомка `A[i]`
private static int RIGHT(int i) {
   return (2 * i + 2);
}

//вспомогательная функция для замены двух индексов в массиве
private static void swap(int[] sortArr, int i, int j) {
   int swap = sortArr[i];
   sortArr[i] = sortArr[j];
   sortArr[j] = swap;
}
```

```
//рекурсивный алгоритм heapify-down. Узел с индексом `i` и 2 его прямых потомка нарушают свойство кучи
private static void heapify(int[] sortArr, int i, int size) {
    // получить левый и правый потомки узла с индексом `i`
    int left = LEFT(i);
    int right = RIGHT(i);
    int largest = i;

    //сравниваем `A[i]` с его левым и правым дочерними элементами и находим наибольшее значение
    if (left < size && sortArr[left] > sortArr[i]) largest = left;
    if (right < size && sortArr[right] > sortArr[largest]) largest = right;

    //поменяться местами с потомком, имеющим большее значение и вызовите heapify-down для дочернего элемента
    if (largest != i) {
        swap(sortArr, i, largest);
        heapify(sortArr, largest, size);
    }
}

//функция для удаления элемента с наивысшим приоритетом (присутствует в корне)
public static int pop(int[] sortArr, int size) {
    //если в куче нет элементов
    if (size <= 0) {
        return -1;
    }
    int top = sortArr[0];

    //заменяем корень кучи последним элементом массива
    sortArr[0] = sortArr[size-1];
    //вызовите heapify-down на корневом узле
    heapify(sortArr, 0, size - 1);
    return top;
}

//функция для выполнения пирамидальной сортировки массива `A` размера `n`
public static void heapSort(int[] sortArr) {
    //строим приоритетную очередь и инициализируем ее заданным массивом
    int n = sortArr.length;

    //build-heap: вызывать heapify, начиная с последнего внутреннего
    //узел до корневого узла
    int i = (n - 2) / 2;
    while (i >= 0) {
        heapify(sortArr, i--, n);
    }
```

```
    //несколько раз извлекаем из кучи, пока она не станет пустой
    while (n > 0) {
        sortArr[n - 1] = pop(sortArr, n);
        n--;
    }
}

public static void main(String args[]) {
    int[] sortArr = {12, 6, 4, 1, 15, 10};
    heapSort(sortArr);
    for(int i = 0; i < sortArr.length; i++){
        System.out.print(sortArr[i] + "\n");
    }
}
```

Сложность алгоритма: O(n log n)