Modul 12 Implementasi Struktur Data
Heap Tree dan *Heapsort*

12.1. Tujuan

Setelah mengikuti praktikum ini mahasiswa diharapkan dapat:

1. Mengenal konsep dan implementasi *heap tree*

2. Mengenal konsep dan implementasi *heapsort*

3. Mengenal dan dapat mengimplementasikan PriorityQueue pada Java

12.2. Alat dan Bahan

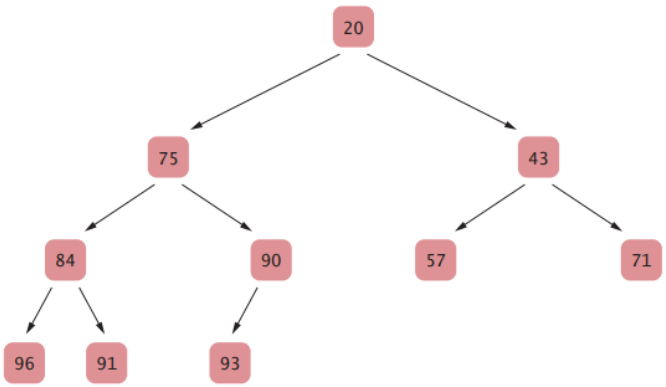
Alat & Bahan Yang digunakan adalah hardware perangkat PC beserta Kelengkapannya berjumlah 40 PC serta Software IntelliJ IDEA yang telah terinstall pada masing-masing PC

12.3. Dasar Teori

Heap adalah binary tree yang memiliki dua karakteristik khusus (pada praktikum ini kita masih membicarakan heap dengan prioritas minimum):

a. Heap tree merupakan tree yang hamper penuh; semua node – kecuali node pada level terakhir - telah terisi. Node pada level terakhir bisa jadi ada yang tidak terisi (yang tidak terisi harus condong pada sisi kanan).
b. Tree tersebut memenuhi heap property: semua node memiliki nilai paling tidak sama besar dengan descendant-nya (tidak boleh lebih besar).

Jadi, dengan min-heap ini, nilai terkecil akan tersimpan pada root. Gambar 1 memperlihatkan contoh sebuah heap tree. Implementasi heap tree ini merupakan suatu priority queue.



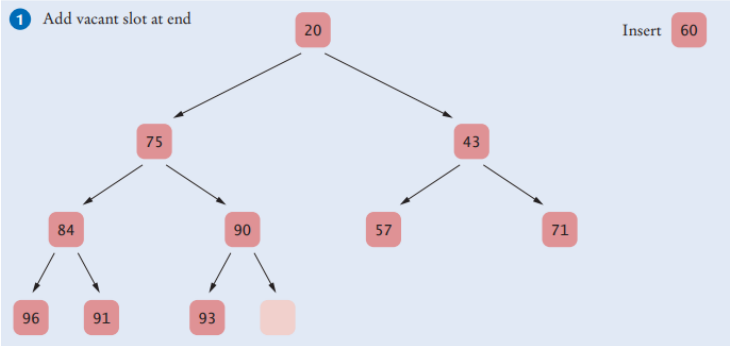
Gambar 1 Heap Tree (Deitel&Deitel, 2017)

Pada praktikum kali ini akan dibahas mengenai algoritma pembentukan *heap tree* dengan *min-heap*. Algoritma yang dibahas adalah algoritma penyisipan (*insertion*) dan penghapusan (*deletion*). Selain itu akan dibahas pula algoritma pembentukan sorting dengan *heap*; *heap sort*.

12.3.1. Heap Tree

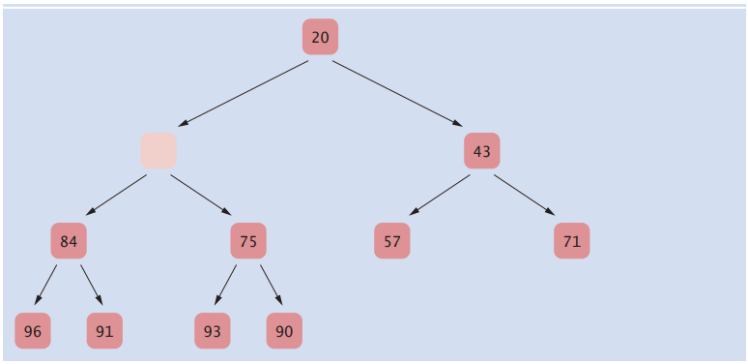
Representasi heap tree adalah dengan menggunakan array. Penyisipan pada heap tree dilakukan pada leaf terakhir, atau pada array dengan index terakhir. Berikut algoritma penyisipan pada heap tree dengan min-heap (Big Java, 2010):

a. Sebelumnya, tambahkan satu slot pada akhir tree (Gambar 2).



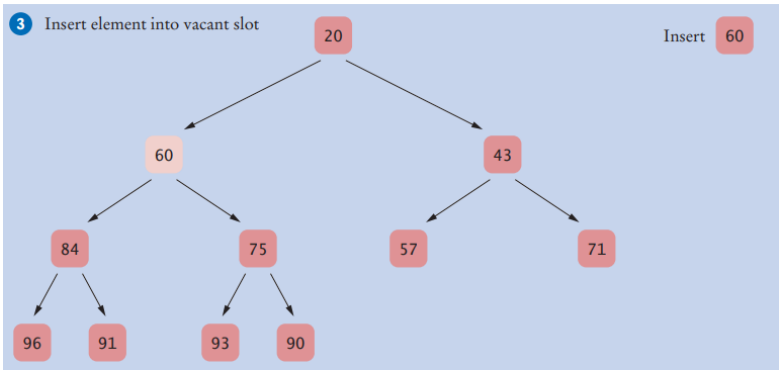
Gambar 2 Menyisipkan 60 pada heap tree

b. Lakukan heapify (penyusunan kembali); turunkan parent dari node pada slot kosong tersebut jika parent tersebut bernilai lebih besar dari elemen yang akan disisipkan. Jadi, nilai dari parent akan menempati slot yang kosong, dan isi dari slot kosong akan menempati node parent. Lakukan hal yang sama selama parent dari slot kosong masih lebih besar daripada elemen yang sisipkan (Gambar 3).



Gambar 3 Melakukan heapify

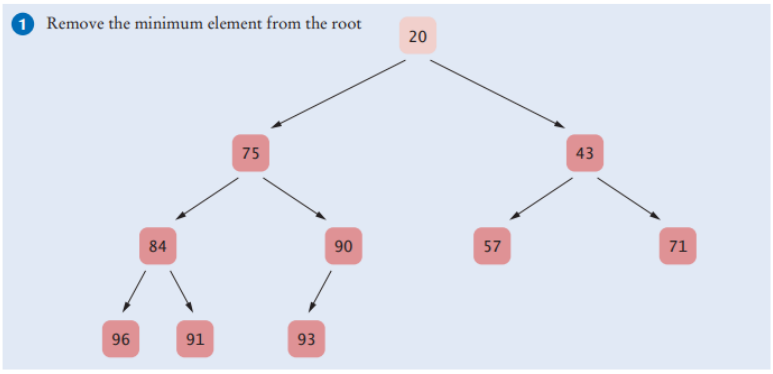
c. Ketika sudah ditemukan posisi yang sesuai, yaitu nilai parent dari slot kosong lebih kecil daripada elemen yang akan disisipkan (atau malah sudah sampai pada root), maka sisipkan elemen pada posisi tersebut (Gambar 4).



Gambar 4 Posisi yang tepat ditemukan

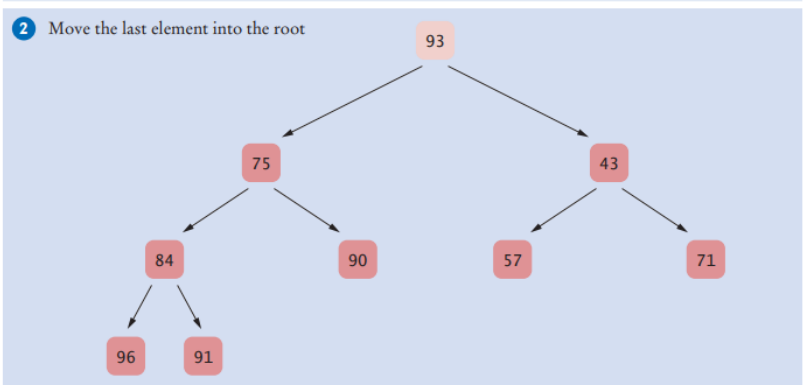
Sementara itu, operasi penghapusan dilakukan melalui root dari heap tree. Berikut algoritma penghapusan heap tree (Big Java, 2010):

a. Hapus/ ambil nilai node root (Gambar 5)



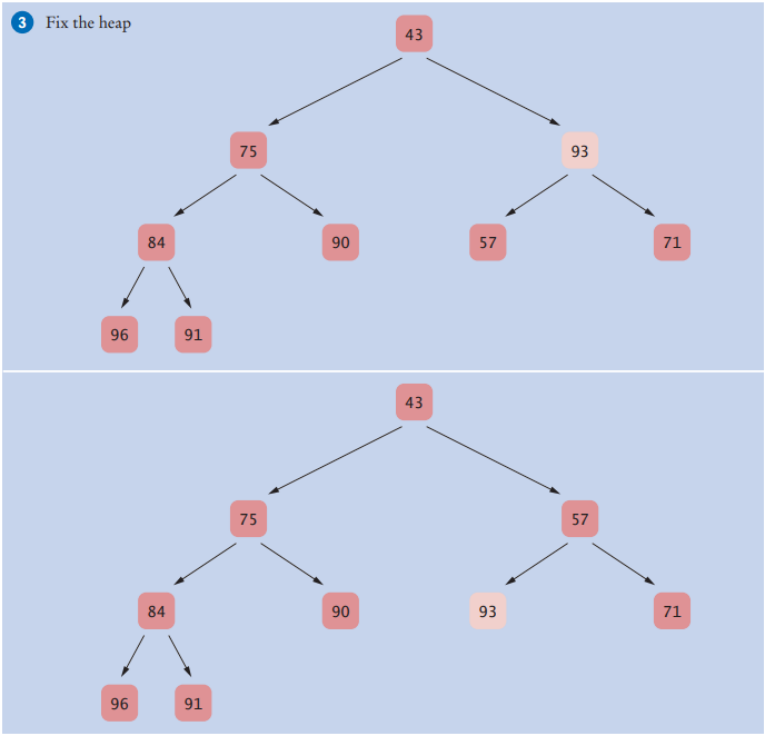
Gambar 5 Penghapusan root

b. Pindahkan nilai dari node terakhir ke root, dan hapus node terakhir. Karena operasi ini, bisa jadi nilai pada root menyalahi heap property, karena salah satu atau kedua anak dari root akan memiliki nilai lebih kecil dari root (Gambar 6).



Gambar 6 Memindahkan node terakhir

c. Lakukan re-heapify; pindahkan anak yang memiliki nilai lebih kecil dari root. Ulangi hal tersebut pada anak yang sudah dipindahkan sampai akhirnya tidak ada lagi anak yang lebih kecil nilainya dari orang tuanya. Proses ini disebut dengan "membetulkan heap (fix the heap)" (Gambar 7).



Gambar 7 Fix the heap

Berikut program min-heap berdasarkan algoritma yang telah dibahas sebelumnya (Big Java, 2010).

```
public class MinHeap {
    private ArrayList<Comparable> elements;

    /**
     * Constructs an empty heap.
     */
    public MinHeap() {
        elements = new ArrayList<Comparable>();
        elements.add(null);
    }

    /**
     * Returns the index of the left child.
     *
     * @param index the index of a node in this heap
     * @return the index of the left child of the given node
     */
    private static int getLeftChildIndex(int index) {
        return 2 * index;
    }

    /**
     * Returns the index of the right child.
     *
```

```
     * @param index the index of a node in this heap
     * @return the index of the right child of the given node
     */
    private static int getRightChildIndex(int index) {
        return 2 * index + 1;
    }

    /**
     * Returns the index of the parent.
     *
     * @param index the index of a node in this heap
     * @return the index of the parent of the given node
     */
    private static int getParentIndex(int index) {
        return index / 2;
    }

    /**
     * Adds a new element to this heap.
     *
     * @param newElement the element to add
     */
    public void add(Comparable newElement) {
        // Add a new leaf
        elements.add(null);
        int index = elements.size() - 1;

        // Demote parents that are larger than the new element
        while (index > 1 && getParent(index).compareTo(newElement) > 0) {
            elements.set(index, getParent(index));
            index = getParentIndex(index);
        }

        // Store the new element in the vacant slot
        elements.set(index, newElement);
    }

    /**
     * Gets the minimum element stored in this heap.
     *
     * @return the minimum element
     */
    public Comparable peek() {
        return elements.get(1);
    }

    /**
     * Removes the minimum element from this heap.
     *
     * @return the minimum element
     */
    public Comparable remove() {
        Comparable minimum = elements.get(1);
        // Remove last element
        int lastIndex = elements.size() - 1;
        Comparable last = elements.remove(lastIndex);

        if (lastIndex > 1) {
            elements.set(1, last);
```

```
            fixHeap();
        }

        return minimum;
    }

    /**
     * Turns the tree back into a heap, provided only the root
     * node violates the heap condition.
     */
    private void fixHeap() {
        Comparable root = elements.get(1);

        int lastIndex = elements.size() - 1;
        // Promote children of removed root while they are smaller than last

        int index = 1;
        boolean more = true;
        while (more) {
            int childIndex = getLeftChildIndex(index);
            if (childIndex <= lastIndex) {
                // Get smaller child

                // Get left child first
                Comparable child = getLeftChild(index);

                // Use right child instead if it is smaller
                if (getRightChildIndex(index) <= lastIndex
                        && getRightChild(index).compareTo(child) < 0) {
                    childIndex = getRightChildIndex(index);
                    child = getRightChild(index);
                }

                // Check if larger child is smaller than root
                if (child.compareTo(root) < 0) {
                    // Promote child
                    elements.set(index, child);
                    index = childIndex;
                } else {
                    // Root is smaller than both children
                    more = false;
                }
            } else {
                // No children
                more = false;
            }
        }

        // Store root element in vacant slot
        elements.set(index, root);
    }

    /**
     * Returns the number of elements in this heap.
     */
    public int size() {
        return elements.size() - 1;
    }
```

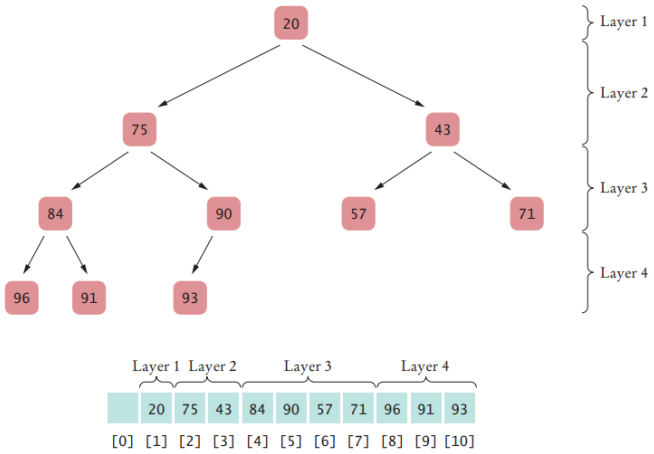
```
    /**
     * Returns the value of the left child.
     *
     * @param index the index of a node in this heap
     * @return the value of the left child of the given node
     */
    private Comparable getLeftChild(int index) {
        return elements.get(2 * index);
    }

    /**
     * Returns the value of the right child.
     *
     * @param index the index of a node in this heap
     * @return the value of the right child of the given node
     */
    private Comparable getRightChild(int index) {
        return elements.get(2 * index + 1);
    }

    /**
     * Returns the value of the parent.
     *
     * @param index the index of a node in this heap
     * @return the value of the parent of the given node
     */
    private Comparable getParent(int index) {
        return elements.get(index / 2);
    }

}
```

Pada program tersebut, heap tree dibuat pada suatu array list. Untuk mudahnya, indeks pertama (indeks 0) dari array list sengaja dikosongkan, jadi data mulai diisi dari layer 1, layer kedua, dan seterusnya (Gambar 8). Indeks dari anak dari node pada indeks i dapat dicari dengan rumus berikut:

$$indeks\ anak\ kiri = 2 * i$$

$$indeks\ anak\ kanan = 2 * i + 1$$

Sementara itu, orang tua (parent) dari suatu node dengan indeks i diperoleh dengan rumus : i/2. Jadi, kelebihan dari implementasi heap tree dibandingkan tree yang lain (BST misalnya) adalah bahwa implementasi dapat dilakukan dengan lebih sederhana karena tidak perlu membuat individual node atau menyediakan link ke node anak.



Gambar 8 Heap tree

Pada contoh program kita, data yang dimasukkan ke dalam heap tree adalah data work order, seperti terlihat pada program berikut. Properti dari heap tree diperoleh dari variabel aPriority. Karena heap tree merupakan pohon yang tersusun, maka data yang dimasukkan harus dapat dibandingkan, sehingga kelas WorkOrder harus mengimplementasikan interface Comparable.

```
public class WorkOrder implements Comparable {
    private int priority;
    private String description;

    /**
     * Constructs a work order with a given priority and description.
     *
     * @param aPriority    the priority of this work order
     * @param aDescription the description of this work order
     */
    public WorkOrder(int aPriority, String aDescription) {
        priority = aPriority;
        description = aDescription;
    }

    public String toString() {
        return "priority=" + priority + ", description=" + description;
    }

    public int compareTo(Object otherObject) {
        WorkOrder other = (WorkOrder) otherObject;
        if (priority < other.priority) return -1;
        if (priority > other.priority) return 1;
        return 0;
    }
}
```

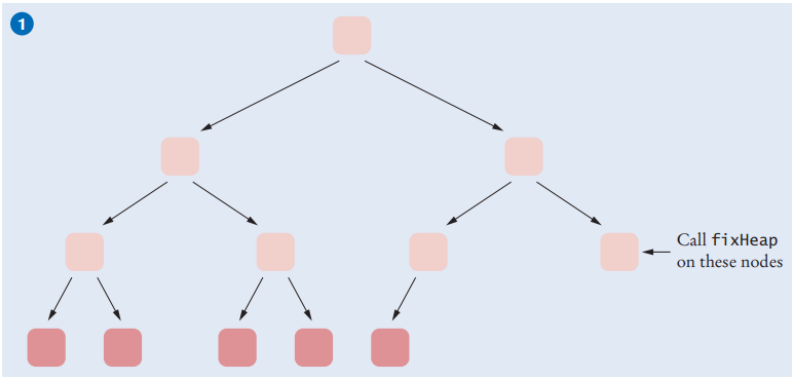
Program di bawah merupakan demo dari heap tree yang telah dibuat, dengan memasukkan work order secara acak. Hasil pogram adalah urutan pekerjaan, dimulai dari pekerjaan dengan prioritas kecil terlebih dahulu.

```
public class HeapDemo {
    public static void main(String[] args) {
        MinHeap q = new MinHeap();
        q.add(new WorkOrder(3, "Shampoo carpets"));
        q.add(new WorkOrder(7, "Empty trash"));
        q.add(new WorkOrder(8, "Water plants"));
        q.add(new WorkOrder(10, "Remove pencil sharpener shavings"));
        q.add(new WorkOrder(6, "Replace light bulb"));
        q.add(new WorkOrder(1, "Fix broken sink"));
        q.add(new WorkOrder(9, "Clean coffee maker"));
        q.add(new WorkOrder(2, "Order cleaning supplies"));

        while (q.size() > 0)
            System.out.println(q.remove());
    }
}
```

12.3.2. Heapsort

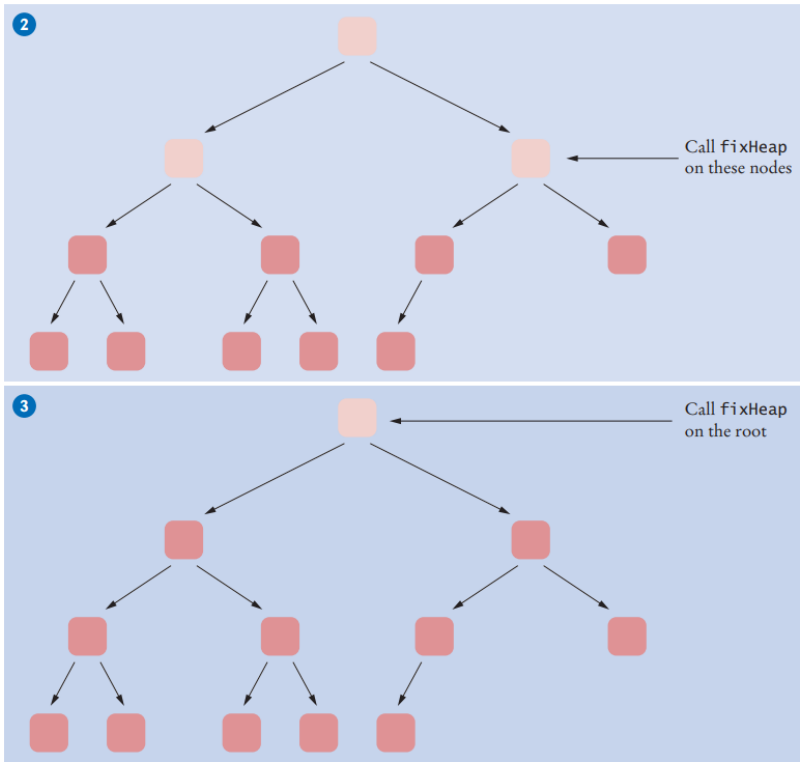
Heapsort merupakan salah satu algoritma sorting yang efisien. Pada algoritma ini, semua elemen yang akan diurutkan disimpan terlebih dahulu pada heap tree, kemudian lakukan ekstraksi (penghapusan/ pengambilan elemen) minimum.

Jadi, supaya algoritma ini lebih efisien, penyimpanan elemen pada heap tree tidak dilakukan satu per-satu, tetapi sekaligus pada satu array. Jadi seluruh elemen disimpan terlebih dahulu ke dalam array, kemudian lakukan proses “fix the heap” seperti yang telah dijelaskan pada sub bab sebelum ini. Proses ini dilakukan pada sub tree dari node terakhir pada level sebelum level paling bawah pada heap tree (Gambar 9). Jadi, algoritma yang digunakan pada praktikum kali ini sedikit berbeda dengan algoritma yang dijelaskan pada perkuliahan, dimana proses “fix the heap” di perkuliahan dilakukan mulai dari pertengahan tree (node terakhir yang memiliki anak).



Gambar 9 Pemanggilan pertama fixHeap

Proses “fix the heap” ini dilakukan bertingkat, mulai dari sub tree pada node terakhir, dimana root dari sub tree tersebut merupakan parent dari node terakhir. Kemudian, proses dilanjutkan pada level diatas sub tree tersebut, dan seterusnya sampai kepada root dari heap tree (Gambar 10). Hal ini akan menjamin tree yang terbentuk sudah memenuhi property heap.



Gambar 10 Pembentukan heap tree

Berikut program heapsort berdasarkan algoritma yang telah kita bahas.

```
/**
 *  This class applies the heapsort algorithm to sort an array.
 *
 */
public class HeapSorter {
    private int[] a;

    /**
     * Constructs a heap sorter that sorts a given array.
     *
     * @param anArray an array of integers
     */
    public HeapSorter(int[] anArray) {
        a = anArray;
    }

    /**
     * Sorts the array managed by this heap sorter.
     */
    public void sort() {
        int n = a.length - 1;
        for (int i = (n - 1) / 2; i >= 0; i--)
```

```
            fixHeap(i, n);
        while (n > 0) {
            swap(0, n);
            n--;
            fixHeap(0, n);
        }
    }

    /**
     * Ensures the heap property for a subtree, provided its
     * children already fulfill the heap property.
     *
     * @param rootIndex the index of the subtree to be fixed
     * @param lastIndex the last valid index of the tree that
     *                  contains the subtree to be fixed
     */
    private void fixHeap(int rootIndex, int lastIndex) {
        // Remove root
        int rootValue = a[rootIndex];

        // Promote children while they are larger than the root

        int index = rootIndex;
        boolean more = true;
        while (more) {
            int childIndex = getLeftChildIndex(index);
            if (childIndex <= lastIndex) {
                // Use right child instead if it is larger
                int rightChildIndex = getRightChildIndex(index);
                if (rightChildIndex <= lastIndex && a[rightChildIndex] >
a[childIndex]) {
                    childIndex = rightChildIndex;
                }

                if (a[childIndex] > rootValue) {
                    // Promote child
                    a[index] = a[childIndex];
                    index = childIndex;
                } else {
                    // Root value is larger than both children
                    more = false;
                }
            } else {
                // No children
                more = false;
            }
        }

        // Store root value in vacant slot
        a[index] = rootValue;
    }

    /**
     * Swaps two entries of the array.
     *
     * @param i the first position to swap
     * @param j the second position to swap
     */
    private void swap(int i, int j) {
```

```
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }

    /**
     * Returns the index of the left child.
     *
     * @param index the index of a node in this heap
     * @return the index of the left child of the given node
     */
    private static int getLeftChildIndex(int index) {
        return 2 * index + 1;
    }

    /**
     * Returns the index of the right child.
     *
     * @param index the index of a node in this heap
     * @return the index of the right child of the given node
     */
    private static int getRightChildIndex(int index) {
        return 2 * index + 2;
    }
}
```

Pada program tersebut, proses repetisi “fix the heap” dilakukan mulai dari node terakhir pada level sebelum level terakhir, dijalankan ke arah kiri. Kemudian dilanjutkan pada level yang lebih tinggi, sampai akhirnya sampai pada root. Karena array yang digunakan adalah array biasa, bukan array list, maka indeks pertama pada array tetap dimulai dari 0.

Setelah array berhasil dibentuk menjadi heap, maka dilakukan penghapusan pada elemen root secara berulang sampai habis semua elemen tersebut. Penghapusan elemen ini dilakukan sama seperti pada sub-bab sebelumnya, dengan memindahkan (menukar) elemen terakhir ke root dan kemudian melakukan fixHeap. Agar hasil sort sesuai dengan urutan ascending, maka pada program heapsort ini prioritas yang digunakan adalah prioritas maksimum (max-heap), bukan min-heap seperti pada algoritma sebelum ini.

12.3.3. Priority Queue

Java menyediakan satu kelas khusus yang mengimplementasikan Heap Tree dengan Heap Sort-nya. Kelas tersebut adalah kelas **PriorityQueue**. Pada kelas ini, elemen yang dimasukkan akan tersusun secara terurut, dan penghapusan elemen dilakukan pada data yang terletak paling depan. Yang perlu diperhatikan, susunan terurut ini bukanlah susunan yang benar-benar terurut secara ascending, tapi terurut berdasarkan Heap Tree. Jadi, jika dilakukan pencetakan keseluruhan elemen (dengan for, misalnya), hasil cetaknya tidak akan terurut. Hasil terurut ini baru akan muncul ketika dilakukan

penghapusan elemen, dimana elemen yang dihapus dapat dipastikan selalu merupakan elemen yang paling besar nilainya (Heap Tree prioritas maksimum).

Penggunaan Priority Queue pada kehidupan sehari-hari diantaranya adalah:

a. Task Scheduling: Priority Queue digunakan oleh sistem operasi untuk melakukan penjadwalan pada task berdasarkan level prioritas mereka. High-priority task seperti update sistem yang penting bisa jadi akan dijadwalkan terlebih dahulu daripada task dengan prioritas yang lebih rendah, seperti proses back-up.
b. Emergency Room (ER): Pada ruang emergency di rumah sakit (di IGD misalnya), pasien dirawat berdasarkan seberapa parah kondisi pasien tersebut, pasien-pasien dengan kondisi kritis harus dirawat terlebih dahulu. Priority Queue dapat digunakan untuk mengatur antrian pasien untuk memudahkan dokter dan perawat.
c. Network Routing: Pada jaringan komputer, priority queue digunakan untuk mengatur alur perjalan paket data. Paket dengan prioritas tinggi seperti data suara dan video bisa jadi akan diberikan prioritas terlebih dahulu daripada data dengan prioritas lebih rendah (seperti email dan transfer file).
d. Transport: Priority queue dapat digunakan pada sistem manajemen lalu-lintas untuk mengatur jalannya lalu-lintas kendaraan. Sebagai contoh, kendaraan seperti ambulans akan diberikan prioritas agar dapat sampai di tempat tujuan secepatnya.
e. Job Scheduling: Priority queue digunakan untuk mengatur urutan tugas yang harus dilakukan terlebih dahulu.
f. Online Marketplace: Pada marketplace, priority queue dapat digunakan untuk mengatur pengiriman barang ke konsumen.

Jadi, priority queue merupakan struktur data yang sangat berguna untuk mengatur tugas dan sumber daya berdasarkan level prioritasnya pada berbagai keadaan di dunia nyata.

Priority Queue dibuat dengan memanggil kelas PriorityQueue dari java.util. Contoh pembuatan objek dari Priority Queue adalah sebagai berikut.

```
PriorityQueue<String> myQueue = new PriorityQueue<>();
```

Untuk pembuatan objek dengan pemanggilan konstruktor seperti di atas (`PriorityQueue<>()`), maka akan membentuk suatu queue dengan kapasitas default awal 11 (ingat, queue dibentuk dengan array atau ArrayList). Selain itu, ada beberapa pembentukan objek lainnya, misal dengan menuliskan kapasitas awal yang diinginkan.

Method-method yang ada di Priority Queue diantaranya dapat dilihat pada Tabel 1.

Tabel 1 Method pada Priority Queue

METHOD	Deskripsi
add(E e)	Memasukkan elemen pada priority queue.
clear()	Menghapus semua elemen dari priority queue.
contains(Object o)	Mengembalikn nilai True jika pada queue mengandung objek tersebut.
iterator()	Membuat iterator untuk queue.
offer(E e)	Memasukkan elemen pada priority queue.
remove(Object o)	Menghapus satu elemen dari priority queue.
peek()	Mengembalikan nilai yang ada pada posisi pertama queue, tanpa menghapus elemen tersebut.
poll()	Mengembalikan sekaligus menghapus elemen pertama pada queue
toArray()	Mengubah priority queue menjadi array.

Contoh kode demo Priority Queue dapat dilihat sebagai berikut.

```
import java.util.ArrayList;

import java.util.Iterator;

import java.util.PriorityQueue;


public class DemoPriorityQueue{
    public static void main(String[] args) {
        PriorityQueue<String> myQueue = new PriorityQueue<>();

        myQueue.add("Tita");
        myQueue.add("Elena");
        myQueue.add("Sidik");
        myQueue.add("Cuke");
        myQueue.add("Maya");

        for (String nama : myQueue) {
            System.out.print(nama + " ");
        }


        ArrayList<String> tampung = new ArrayList<>();
        Iterator<String> iterator = myQueue.iterator();

        while(iterator.hasNext()){
                    tampung.add(myQueue.poll());
        }

        System.out.println();

        for (String data : tampung) {
            System.out.print(data + " ");
        }
    }
}
```

Pada program di atas, dibuat suatu Priority Queue dengan nama myQueue. Karena nilai inisial awal dikosongkan, maka queue akan memiliki kapasitas default (11). Untuk memasukkan elemen ke dalam queue, digunakan method “add”. Setelah itu, isi dari queue dicetak. Perhatikan, karena pada saat memasukkan tidak dituliskan prioritas bagi masing-masing elemen, maka penyimpanan elemen pada queue akan didasarkan pada urutan naturalnya (huruf “C” akan disimpan paling awal), tetapi berdasarkan heap tree dengan prioritas minimum, yaitu elemen pertama akan menjadi elemen paling kecil.

Setelah itu, untuk mendapatkan urutan sesuai dengan urutan natural dengan pola ascending, digunakan method poll(). Dengan ini, sisa elemen pada queue akan dijamin bahwa nilai paling awalnya adalah nilai paling kecil. Jika kita mengeluarkan elemen menggunakan method ini, maka akan diperoleh urutan sesuai dengan urutan natural elemen.

Hasil dari program dapat dilihat sebagai berikut. Hasil cetak pertama adalah isi dari queue berdasarkan heap tree, sementara hasil cetak kedua adalah hasil dari poll tree yang disimpan pada suatu ArrayList (jadi sesuai dengan urutan natural dari String).

```
PS D:\UT New Step\Ngajar\2223-2\ISD\Praktikum\codes\praktikum12>  & 'C:\Program Files\Java\jdk-11.0.12\bin\java.exe' '-cp'
'C:\Users\Cahyana\AppData\Roaming\Code\User\workspaceStorage\1a55569336264c9adc6da6440eb6dffc\redhat.java\jdt_ws\jdt.ls-jav
a-project\bin' 'DemoPriorityQueue'
Cuke Elena Sidik Tita Maya
Cuke Elena Maya Sidik Tita
```

Bagaimana jika elemen yang akan dimasukkan adalah suatu tipe data bentukan (misal, data Mahasiswa)? Maka, tentu harus dibentuk kelas POJO dari data tersebut, yang mengimplementasikan Comparable, sehingga data dapat diurutkan.

Kode program berikut merupakan contoh yang diambil dari GeeksForGeeks untuk suatu Priority Queue yang menggunakan tipe data bentukan. Silahkan dipelajari dan dipahami.

```
/ Java program to demonstrate working of
// comparator based priority queue constructor
import java.util.*;

public class Example {
    public static void main(String[] args){
        Scanner in = new Scanner(System.in);
        // Creating Priority queue constructor having
        // initial capacity=5 and a StudentComparator instance
        // as its parameters
        PriorityQueue<Student> pq = new
             PriorityQueue<Student>(5, new StudentComparator());

               // Invoking a parameterized Student constructor with
               // name and cgpa as the elements of queue
               Student student1 = new Student("Nandini", 3.2);

               // Adding a student object containing fields
```

```
                // name and cgpa to priority queue
                pq.add(student1);
                Student student2 = new Student("Anmol", 3.6);
                        pq.add(student2);
                Student student3 = new Student("Palak", 4.0);
                        pq.add(student3);

                // Printing names of students in priority order,poll()
                // method is used to access the head element of queue
                System.out.println("Students served in their priority order");

                while (!pq.isEmpty()) {
                System.out.println(pq.poll().getName());
        }
    }
}

class StudentComparator implements Comparator<Student>{

            // Overriding compare()method of Comparator
                        // for descending order of cgpa
            public int compare(Student s1, Student s2) {
                if (s1.cgpa < s2.cgpa)
                    return 1;
                else if (s1.cgpa > s2.cgpa)
                    return -1;
                                return 0;
                }
        }

class Student {
    public String name;
    public double cgpa;

    // A parameterized student constructor
    public Student(String name, double cgpa) {

        this.name = name;
        this.cgpa = cgpa;
    }

    public String getName() {
        return name;
    }
}
```

Keluaran program ketika dijalankan adalah sebagai berikut

```
Students served in their priority order
Palak
Anmol
Nandini
```

Reference

Carrano, F., M., *Data Structures and Abstraction With Java,* Prentice Hall, (2012)
Deitel, P and Deitel H., *Java How to Program: Early Objects*, 11th Ed, Pearson. (2017)
GeeksForGeeks, "PriorityQueue in Java", https://www.geeksforgeeks.org/priority-queue-class-in-java/, diakses 13 Mei 2023
palakmjn, "Implement PriorityQueue through Comparator in Java", https://www.geeksforgeeks.org/implement-priorityqueue-comparator-java/ , diakses 13 Mei 2023