# Praktikum DAA: Priority Queue with Handle

Pada modul ini kta akan mengimplementasikan Minimum Priority Queue yang menggunakan handle. Struktur data ini nantinya dapat digunakan untuk mengimplementasikan algoritma Prim's MST dan Dijkstra's SSSP. Ingat pada modul sebelumnya kita mengimplementasikan Heap yang hanya menyimpan *key*. Kali ini kita akan menyimpan *id* dan *key*. Nantinya id dapat digunakan untuk menunjuk pada objek apapun di luar priority queue, misalnya nomor vertex pada graph.

Mula-mula kita buat kelas **`MinPrioQueue`** yang memiliki inner class bernama **`Data`** yang menampung dua atribut, yaitu **`id`** dan **`key`**. Kelas ini memiliki atribut-atribut yang mirip seperti Heap pada modul yang lalu, hanya saja ada tambahan array position. Berikut ini adalah atribut-atribut kelas **`MinPrioQueue`**:

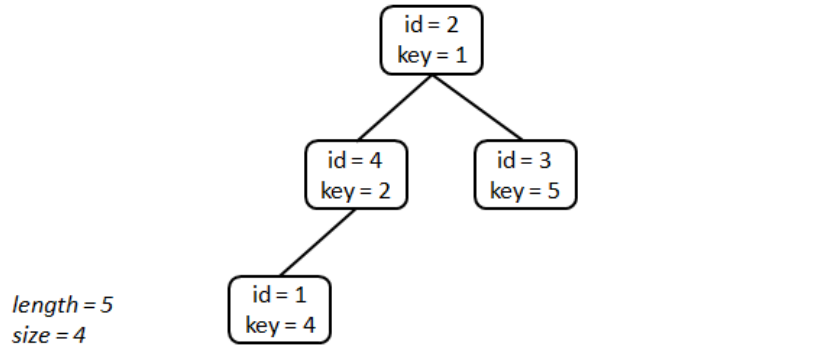
- **`Data[] heap`** → untuk menampung heap
- **`int[] position`** → untuk handle
- **`int length`** → kapasitas maksimal heap
- **`int size`** → ukuran heap saat ini

Tambahkan constructor, **`getLeft`**, **`getRight`**, dan **`getParent`**, mirip seperti pada modul yang lalu.

```
class MinPrioQueue{
    private class Data{
        int id;
        int key;

        Data(int id, int key){
            this.id = id;
            this.key = key;
        }
    }

    private Data[] heap;
    private int position[];
    private int length;
    private int size;

    public MinPrioQueue(int length){
        this.length = length;
        this.size = 0;
        this.heap = new Data[length+1];
        this.position = new int[length+1];
    }

    private int getLeft(int i){ return i<<1; }
    private int getRight(int i){ return (i<<1)|1; }
    private int getParent(int i){ return i>>1; }
}
```

## Array `position` untuk Handle

Kali ini kita memiliki array **`position`** yang berisi informasi di mana keberadaan suatu id pada array **`heap`**. Perhatikan ilustrasi berikut ini. Misalnya ada heap yang dapat menampung 5 data, namun hanya terisi 4 diantaranya. Ukuran array **`heap`** dan **`position`** adalah 6, karena index ke-0 tidak dipakai. Perhatikan bahwa array **`position`** di index ke-5 berisi 0 karena pada heap tidak terdapat elemen dengan id=5. Jadi array **`position`** sekaligus digunakan untuk menentukan apakah id tertentu ada pada heap atau tidak. Nilai *penanda* bisa apa saja selama di luar range id yang dipakai. Pada kasus ini kita akan menggunakan id antara 1 .. length, maka 0 dapat digunakan sebagai penanda.




| index | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| heap | | id = 2 key = 1 | id = 4 key = 2 | id = 3 key = 5 | id = 1 key = 4 | null |

| index | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| position | | 4 | 1 | 3 | 2 | 0 |

Misalnya kita ingin mengganti key milik *id=1*, tadinya 4 menjadi 1. Perhatikan bahwa pada saat kita mempelajari struktur data Heap, operasi DECREASE-KEY menerima masukan berupa *index* heap dan *key baru*. Tapi jika kita perhatikan, pada kasus pemakaian priority queue, yang umum diketahui adalah *id elemen*, bukan *index*-nya. Misalnya pada algoritma Prim/Dijkstra, yang kita tahu adalah nomor vertex yang ingin diganti key-nya, bukan di mana lokasi vertex itu pada array heap. Di sinilah gunanya array **`position`**. Pada kasus ini, kita dapat mengetahui *id=1* ada di index 4 dengan melihat isi array position *index* ke-1.

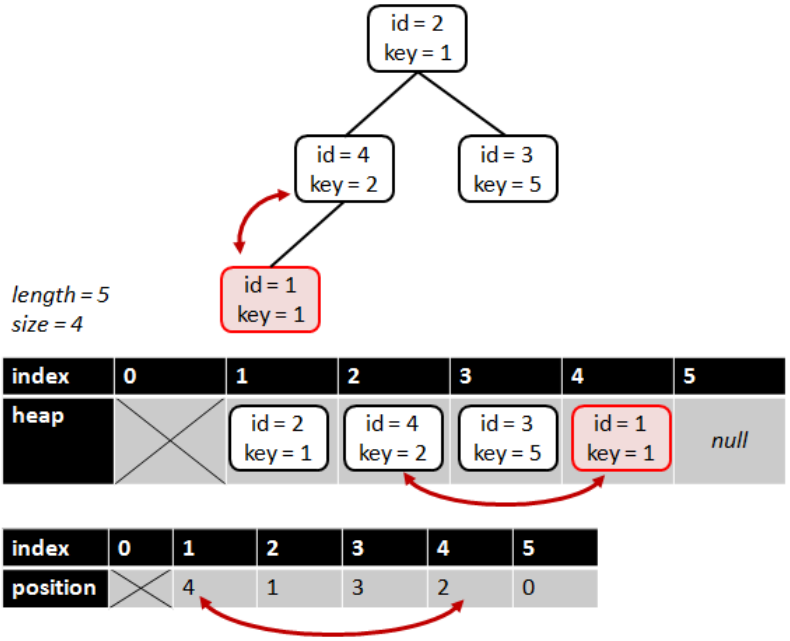
*length = 5*
*size = 4*

| index | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| heap | | id = 2 key = 1 | id = 4 key = 2 | id = 3 key = 5 | id = 1 key = 1 | null |

*nilai baru*

| index | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| position | | 4 | 1 | 3 | 2 | 0 |

Selanjutnya kita perlu melakukan swapping sehingga sifat heap tetap dipertahankan. pada kasus ini, elemen di index ke-4 (id=1) harus ditukar tempat dengan parent-nya, yaitu elemen di index ke-2 (id=4). Ketika elemen pada heap ditukar, isi array position juga harus diperbaiki. Pada kasus ini, posisi id=1 ditukar dengan posisi id=4.




Operasi swap digunakan pada operasi heapify, decrease-key, dan extract-min. Karena di sini swap menjadi sedikit lebih rumit, sebaiknya kita buat method terpisah untuk menanganinya. Tambahkan method private **`swap(idx1, idx2)`** seperti berikut ini:

```
private void swap(int idx1, int idx2){
    // tukar isi heap
    Data temp = this.heap[idx1];
    this.heap[idx1] = this.heap[idx2];
    this.heap[idx2] = temp;

    // ambil id
    int id1 = this.heap[idx1].id;
    int id2 = this.heap[idx2].id;

    // tukar isi array position
    int temp2 = this.position[id1];
    this.position[id1] = this.position[id2];
    this.position[id2] = temp2;
}
```

## Operasi `decreaseKey`

Seperti yang sudah disebutkan sebelumnya, input pada operasi **`decreaseKey`** adalah nomor *id* dan *key baru*. Hal ini juga penting untuk mengimplementasikan konsep *enkapsulasi*. Pemakai priority queue seharusnya tidak tahu menahu mengenai implementasi di dalamnya, karena itu tidak perlu tahu di mana index suatu *id* berada.

Dengan implementasi cara ini, kita juga perlu menangani jika id yang dimasukkan tidak terdapat pada priority queue. Lengkapi potongan kode **`decreaseKey`** berikut ini! Gunakan method **`swap`** yang sudah dibuat sebelumnya.

```
public void decreaseKey(int id, int newKey){
        int curr = this.position[id];   //mengambil index-nya
        if(curr == 0)   return;         //tidak ada id tersebut

        if(this.heap[curr].key > newKey){
            //lengkapi...
        }
}
```

## Method `insert`

Method insert mirip seperti pada Heap, hanya saja di sini parameternya ada dua: **`id`** dan **`key`**. Kita juga perlu menambahkan penanganan untuk *heap penuh*, *id yang di luar range*, dan *id yang sudah pernah ada sebelumnya*. Lengkapi potongan kode **`insert`** berikut ini! Gunakan method **`decreaseKey`** yang sudah dibuat sebelumnya.

```
public boolean insert(int id, int key){
    if(this.size == this.length)        //heap penuh
        return false;
    else if(id<=0 || id>this.length)    //id di luar range
        return false;
    else if(this.position[id] != 0)     //id duplikat
        return false;
    else{
        //lengkapi...
    }
}
```

## Method `extractMin`

Karena priority queue sekarang menyimpan dua data (**`id`** dan **`key`**), maka **`extractMin`** juga harus dapat mengembalikan kedua nilai tersebut. Untuk itu, kita gunakan array **`int[2]`**, di mana index ke-0 berisi **`id`** dan index ke-1 berisi **`key`**. Pada kasus ini kita tidak dapat mengembalikan objek **`Data`** secara langsung, karena **`Data`** adalah inner class. Sekali lagi, untuk menerapkan konsep enkapsulasi, pengguna priority queue tidak perlu (dan tidak dapat) mengakses inner class.

Beberapa hal yang perlu diperhatikan pada implementasi method ini:

- Jika priority queue sudah kosong, kembalikan *null*
- Tandai array posisi untuk **`id`** yang dikeluarkan (jadikan 0, menandakan **`id`** tersebut sudah tidak ada di dalam priority queue).
- Ketika elemen terakhir dipindah ke index ke-1 sebagai pengganti elemen yang dikeluarkan, jangan lupa update array posisinya

Berikut ini contoh implementasi method **`extractMin.`** Buatlah method **`private void heapify(int curr)`** yang mirip seperti pada modul sebelumnya, tapi modifikasikan untuk Minimum Heap. Gunakan method **`swap`** untuk membatu proses heapify.

```
public int[] extractMin(){
    if(this.size==0)    return null;//heap kosong

    this.swap(1, this.size);        //tukar ke paling belakang sekalian memperbaiki posisi
    Data d = this.heap[this.size];
    this.position[d.id] = 0;        //tandai id tersebut sudah dihapus dari heap
    this.size--;                    //ukuran heap berkurang 1
    this.heapify(1);

    int[] result = new int[2];
    result[0] = d.id;
    result[1] = d.key;
    return result;
}
```

## Tester dan Debugging

Untuk menguji implementasi priority queue anda, tambahkan method **`print`** yang menampilkan isi array **`heap`** dan array **`position`** seperti berikut ini:

```
public void print(){
    System.out.println("Heap");
    for(int i=1; i<=this.size; i++)
        System.out.print("(" + this.heap[i].id + "," + this.heap[i].key +") ");
    System.out.println("");

    System.out.println("Position");
    for(int i=1; i<=this.length; i++)
        System.out.print(this.position[i] + " ");
    System.out.println("");
}
```

Tambahkan kelas tester, misalnya seperti berikut ini:

```
class TesterPrioQueue{
    public static void main(String[] args){
        MinPrioQueue pq = new MinPrioQueue(5);
        pq.insert(1, 4);
        pq.insert(2, 1);
        pq.insert(3, 5);
        pq.insert(0, 4);    //gagal: di luar range
        pq.insert(6, 4);    //gagal: di luar range
        pq.insert(3, 7);    //gagal: id duplikat
        pq.insert(4, 2);
        pq.print();

        int[] res = pq.extractMin();
        System.out.println("Min: "+res[0] +"," + res[1]);
        pq.print();

        res = pq.extractMin();
        System.out.println("Min: "+res[0] +"," + res[1]);
        pq.print();
    }
}
```

**Output yang diharapkan:**

```
Heap
(2,1) (4,2) (3,5) (1,4)
Position
4 1 3 2 0
Min: 2,1
Heap
(4,2) (1,4) (3,5)
Position
2 0 3 1 0
Min: 4,2
Heap
(1,4) (3,5)
Position
1 0 2 0 0
```