Panduan Awal Mempelajari (Binary) Heap

Sebelumnya, Anda sudah mempelajari bahwa ADT Stack dan ADT Queue dapat diimplementasikan menggunakan array dan Linked List dengan kelebihan dan kekurangannya masing-masing. Kali ini, Anda akan belajar mengenai **Binary Heap**. Ketika Anda melihat berbagai literatur tentang Binary Heap, Anda akan menemukan banyak sekali gambar Tree/Pohon. Namun, Binary Heap ini sebenarnya **BUKAN** membicarakan tentang Tree, seperti halnya Binary Search Tree, AVL Tree, maupun Red Black Tree. Binary Heap ini **LEBIH** terkait dengan **Priority Queue**. Nanti kita akan melihat bahwa Binary Heap merupakan sebuah struktur data yang efisien untuk mengimplementasikan **Priority Queue**.

Priority Queue dapat dibagi menjadi 2 jenis:

- **Minimum Priority Queue**: elemen yang pertama kali keluar (yang mempunyai prioritas tinggi) adalah elemen yang mempunyai nilai "paling kecil".
- **Maximum Priority Queue**: elemen yang pertama kali keluar (yang mempunyai prioritas tinggi) adalah elemen yang mempunyai nilai "paling besar".

Priority Queue itu sendiri pada umumnya mempunyai 3 operasi/interface yang penting (misal, untuk Minimum Priority Queue):

- **findMin()**: mengembalikan elemen yang paling kecil (prioritas paling tinggi) tanpa menghapus dari antrian
- **removeMin()**: menghapus elemen yang paling kecil yang ada pada antrian
- **add()**: menambahkan elemen baru ke dalam priority queue

Coba Anda pikirkan seandainya Anda harus mengimplementasikan Priority Queue tersebut menggunakan linked list biasa ! Misal, kita representasikan linked list menggunakan

```
[x1, x2, x3, x4, ...]
```

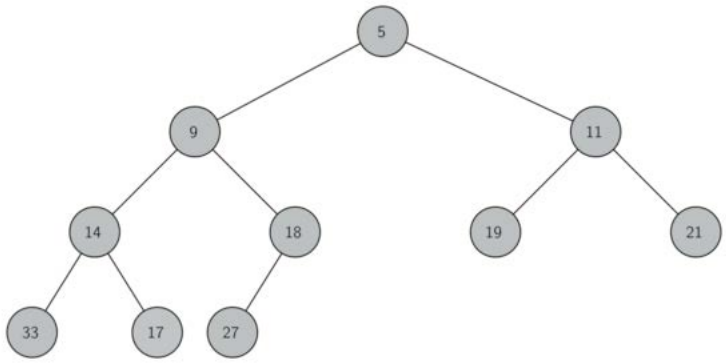
untuk menyatakan sebuah linked list dengan beberapa elemen, dengan **x1** adalah elemen pertama, **x2** adalah yang kedua, dan seterusnya. Untuk minimum priority queue, elemen pertama merupakan elemen paling kecil dan elemen yang ada di posisi terakhir adalah elemen yang paling besar. Berikut adalah contoh seandainya elemen yang berada pada minimum priority queue adalah bilangan bulat:

```
head [10, 12, 15, 17, 18, 34] tail
```

Dengan representasi tersebut, tentunya operasi **findMin()** dan **removeMin()** akan menjadi **konstan O(1)** karena cukup mengakses atau menghapus elemen pada posisi pertama. Namun, operasi **add()** akan menjadi linier **O(n)** karena perlu melakukan proses insert untuk meletakkan elemen baru pada posisi yang tepat sehingga keterurutan pada linked list tetap terjaga.

Heap adalah sebuah struktur data yang dapat menyebabkan operasi **removeMin()** dan **add()** menjadi logaritmik **O(log n)**, dan operasi **findMin()** tetap **konstan O(1)**.

Struktur data **Heap** adalah sebuah **array** yang dapat kita lihat sebagai sebuah **complete binary tree**. Perhatikan gambar berikut !



5	9	11	14	18	19	21	33	17	27
0	1	2	3	4	5	6	7	8	9

Heap itu sendiri adalah array yang ada di bagian bawah. Sedangkan yang ada di bagian atas hanyalah visualisasi dari Heap. Setiap elemen pada tree berkorespondensi dengan sebuah elemen pada array. Setiap level pada Tree terisi penuh, kecuali yang terakhir, terisi dari kiri ke kanan.

Heap ada 2 yaitu:

- **Min-Heap**: heap yang digunakan untuk mengimplementasikan minimum priority queue. Elemen yang paling kecil berada di root.
- **Max-Heap**: heap yang digunakan untuk mengimplementasikan maximum priority queue. Elemen yang paling besar berada di root.

Gambar yang ada di atas adalah contoh dari **Min-Heap**.

Heap Order Property

Heap memiliki sifat heap order property.

- Untuk Min-Heap: untuk setiap node **x** dengan parent **p**, **key yang ada pada p ≤ key yang ada pada x**.
- Untuk Max-Heap: untuk setiap node **x** dengan parent **p**, **key yang ada pada p ≥ key yang ada pada x**.

Anda dapat periksa property ini pada Min-Heap yang ada pada gambar di atas !

Operasi pada Heap

Kali ini, kita fokus ke Min-Heap. Operasi pada Max-Heap serupa dan dapat Anda pahami ketika Anda memahami operasi pada Min-Heap.

Operasi findMin (O(1)). Hanya membutuhkan waktu konstan O(1) karena kita cukup mengakses elemen yang ada pada root, yaitu elemen yang ada pada indeks 0 pada Heap Array.

Operasi add (O(log n)). Berikut adalah langkah-langkah ketika kita ingin menambahkan sebuah elemen pada Heap:

1. [**O(1)**] Tambahkan elemen pada ujung posisi kosong yang ada di kanan array. Jika kita lihat pada tree, elemen baru akan berada di level terakhir paling kanan.
2. [**O(log n)**] Penambahan elemen di posisi terakhir dapat merusak Heap Order Property. Jadi, perlu ada mekanisme untuk mengembalikan ke kondisi Heap Order dimulai dari level paling bawah. Mekanisme tersebut adalah **percolate up**.

Operasi removeMin (O(log n)). Berikut adalah langkah-langkah ketika kita ingin menghapus elemen paling kecil yang berada di root atau paling kiri pada heap array:

1. [**O(1)**] kosongkan elemen pada root atau elemen pada indeks 0
2. [**O(1)**] pindahkan elemen yang berada pada level terakhir paling kanan (paling kanan pada heap array) ke posisi root
3. [**O(log n)**] langkah 1 dan 2 dapat merusak heap order property. Mekanisme untuk mengembalikan ke kondisi heap order adalah **percolate down**.

Percolate Up & Percolate Down

Silakan rujuk ke materi kuliah SDA yang ada di SceLE ☺

Heapify

Silakan lihat slide saya yang diletakkan terpisah (dapat diakses di http://ir.cs.ui.ac.id/alfan/sda) tentang proses **Heapify** dan **Heap Sort**.