Lab Tutorial 6A

Binary Heap

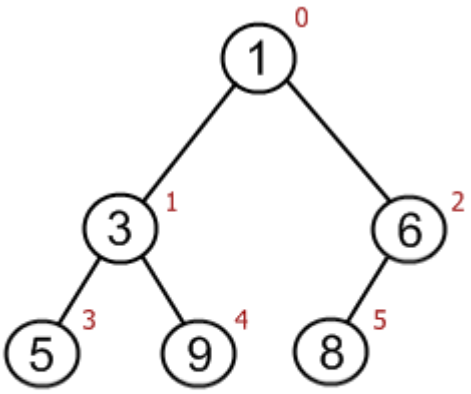
Nama berkas kode sumber	:	SDA<npm>L6A.java
Batas waktu eksekusi program	:	3 detik / kasus uji
Batas memori program	:	256 MB / kasus uji

Binary Heap adalah sebuah struktur data yang memiliki bentuk *Binary Tree*. *Binary Heap* adalah cara yang umum untuk mengimplementasikan *Priority Queue*. Secara lebih detail, *Binary Heap* adalah sebuah *Binary Tree* dengan 2 syarat tambahan, yaitu harus berupa *Complete Binary Tree* (seluruh *node* kecuali level terakhir harus memiliki 2 anak dan pada level terakhir harus diisi dari kiri ke kanan) dan setiap *node* nya memiliki nilai kurang dari sama dengan nilai anak-anaknya (jika *Min Heap*) atau lebih dari sama dengan nilai anak-anaknya (jika *Max Heap*). Meski pun secara abstrak *Binary Heap* adalah sebuah *Binary Tree*, namun pada implementasinya sering kali *Binary Heap* diimplementasikan menggunakan *array* biasa (karena sifatnya yang *complete tree*). Pada tutorial kali ini kita akan mengimplementasikan *Min Heap* menggunakan array. Pada tutorial ini anda **dilarang menggunakan interface Collections atau turunannya** dari Java kecuali ArrayList. Penggunaan Java Collections selain Array dan ArrayList pada **seluruh** bagian kode akan mengakibatkan nilai correctness otomatis **0**. Anda disarankan namun tidak diwajibkan untuk menggunakan kode **template** yang telah disediakan.

A. Tutorial

1. Index Array pada Heap

Penyimpanan nilai *Binary Heap* pada *Array* adalah terurut dari paling atas ke paling bawah, dari kiri ke kanan. Perhatikan *Heap* dan representasi *array* nya di bawah ini:



1	3	6	5	9	8
0	1	2	3	4	5

Index *Array* pada *Heap* (jika *root* dimulai dari indeks 0) selalu memenuhi sifat berikut:

Anak kiri dari indeks i: 2 * i + 1

Anak kanan dari indeks i: 2 * (i + 1)

Parent dari indeks i: *floor*((i - 1) / 2)

Sebagai contoh, anak kanan dari *node* 3 (indeks 1) adalah *node* 9 (pada indeks 2 * (1 + 1) = 4) dan anak kiri dari *node* 6 (indeks 2) adalah *node* 8 (pada indeks 2 * 2 + 1 = 5), serta *parent* dari *node* 5 (indeks 3) adalah node 3 (indeks *floor*((3 -1) / 2)) = 1)

2. **Percolate Up dan Percolate Down**

Untuk menjaga sifat-sifat dari *Binary Heap* seperti yang telah disebutkan di deskripsi awal, maka perlu dilakukan *percolate up* dan *percolate down*. *Percolate up* adalah suatu fungsi pada *Binary Heap* yang akan memeriksa nilai suatu *node*, dan bila nilai dari *node* tersebut lebih kecil dari *parent* nya (pada *Min Heap*) maka ia akan bertukar posisi dengan *parent*-nya. Hal tersebut dilakukan terus menerus hingga *parent*-nya memiliki nilai lebih kecil atau sama dengan nilai *node* tersebut, atau *node* tersebut adalah *root*. Sedangkan *percolate down* adalah sebuah fungsi pada *Binary Heap* dengan memeriksa apakah anak kiri atau anak kanan dari *node* tersebut ada yang memiliki nilai lebih kecil dari nilai *node* tersebut, dan bila ada maka tukar posisi *node* tersebut dengan anaknya yang lebih kecil. Hal tersebut dilakukan terus menerus hingga nilai *node* tersebut lebih kecil dari kedua anaknya.

3. Add

Operasi *add* pada *Binary Heap* secara abstraksi dilakukan dengan menambahkan *node* pada level paling bawah dimulai dari yang paling kiri (sifat *complete tree*), atau secara implementasi *Array*-nya adalah dengan menambahkan pada elemen terakhir, lalu dilakukan *percolate up*.

4. Peek

Implementasi operasi *peek* pada *Binary Heap* sangatlah mudah, cukup mengembalikan nilai elemen pertama pada *Array*.

5. Poll

Poll atau *remove min* pada *Min Heap* secara abstraksi dilakukan dengan membuang *root node*, memindahkan *node* paling bawah kanan ke *root* dan dan dilakukan *percolate down*. Pada implementasinya, pindahkan nilai elemen terakhir pada *Array* ke elemen pertama, lalu lakukan *percolate down*.

B. Soal

Anda akan diberikan beberapa *query* standar pada ***Min Heap*** yang akan di jelaskan di bagian input dan mengeluarkan hasilnya sesuai pada bagian output

Sekedar petunjuk tambahan, untuk mengerjakan soal ini anda **membutuhkan** struktur data bantuan selain *Binary Heap* yang telah anda buat sebelumnya, namun hanya terbatas untuk ArrayList atau Array. Anda disarankan memodifikasi kode template untuk memenuhi kebutuhan soal.

Input dan Output

Program akan menerima input untuk *command* seperti dibawah ini hingga *End of File*:

1. INSERT <VALUE>

- Masukkan node pada *heap* dengan nilai **<VALUE>**.
- **<VALUE>** adalah bilangan bulat positif pada rentang 1 sampai 100000.
- **<VALUE>** dipastikan **tidak dijamin** unik untuk setiap operasi insert.
- Kembalikan sebaris string:

  `"elemen dengan nilai <VALUE> telah ditambahkan"`

2. REMOVE

- Mengeluarkan elemen dengan nilai paling kecil pada heap.
- Jika heap sedang kosong, kembalikan sebaris string:

  `"min heap kosong"`

- Jika heap tidak kosong, kembalikan sebaris string:

  `"<VALUE> dihapus dari heap"`

  dengan **<VALUE>** adalah nilai dari elemen heap paling atas.

3. NUM_PERCOLATE_UP <VALUE>

- Menanyakan jumlah operasi percolate up yang sudah dioperasikan terhadap elemen dengan nilai **<VALUE>**.
- **<VALUE>** adalah bilangan bulat positif pada rentang nilai 1 sampai 10000.
- Jika value belum pernah ada atau sudah dikeluarkan, kembalikan sebaris string:

  `"percolate up 0"`

- selain itu, kembalikan sebaris string:

  `"percolate up <NUM>"`

dengan **<NUM>** adalah jumlah operasi percolate up yang sudah dioperasikan.

4. **NUM_PERCOLATE_DOWN <VALUE>**
   - Menanyakan jumlah operasi percolate down yang sudah dioperasikan terhadap elemen dengan nilai **<VALUE>**.
   - **<VALUE>** adalah bilangan bulat positif pada rentang nilai 1 sampai 10000.
   - Jika value belum pernah ada atau sudah dikeluarkan, kembalikan sebaris string:
     "percolate down 0"
   - selain itu, kembalikan sebaris string:
     "percolate down <NUM>"

     dengan **<NUM>** adalah jumlah operasi percolate down yang sudah dioperasikan.

5. **NUM_ELEMENT**
   - Menanyakan banyak elemen pada heap.
   - Jika heap sedang kosong, kembalikan sebaris string:
     "element 0"
   - selain itu, kembalikan sebaris string:
     "element <NUM_ELEMENT>"

     dengan **<NUM_ELEMENT>** adalah banyak elemen pada heap.

Contoh Input

```
INSERT 3
INSERT 7
INSERT 9
NUM_PERCOLATE_UP 7
REMOVE
NUM_PERCOLATE_DOWN 9
NUM_ELEMENT
```

Contoh Output

```
elemen dengan nilai 3 telah ditambahkan
elemen dengan nilai 7 telah ditambahkan
elemen dengan nilai 9 telah ditambahkan
percolate up 0
3 dihapus dari heap
percolate down 1
element 2
```

Penjelasan Contoh

tidak terjadi percolate up pada 7 karena 7 kurang dari 3

terjadi percolate down pada 9 saat 3 dihapus dari heap, mula-mula 9 yang merupakan *node* paling bawah kanan dari *root* menggantikan 3, lalu dilakukan percolate down dengan swapping terhadap 7, jumlah operasi yang dilakukan adalah 1.

Keterangan Test Case

- Test case 1: contoh input
- Test case 2 - 5: hanya berisi gabungan dari operasi **INSERT, REMOVE,** dan **NUM_ELEMENT**

  Pada test case 2 - 5 anda cukup mengikuti bagian tutorial karena hanya ada operasi *add, poll,* dan *size* yang diwakili oleh *command* di atas.
- Test case 6 - 10: gabungan dari seluruh operasi pada soal

Lab Tutorial 6B

Hash Table

Nama berkas kode sumber	:	SDA<npm>L6B.java
Batas waktu eksekusi program	:	3 detik / kasus uji
Batas memori program	:	256 MB / kasus uji

Pada tutorial ini anda **dilarang menggunakan interface Collections atau turunannya** dari Java kecuali ArrayList. Penggunaan Java Collections selain Array, ArrayList dan Collections.sort pada **seluruh** bagian kode akan mengakibatkan nilai correctness otomatis **0**.

A. Tutorial

Hashing merupakan teknik yang digunakan untuk mengidentifikasi objek dari suatu grup. Teknik ini diterapkan di kehidupan sehari-hari, contoh: NPM atau NIK yang kalian punya, atau SKU suatu produk. Dalam contoh tersebut, Setiap orang di representasikan dalam suatu NPM atau NIK yang unik. Begitu pula setiap produk di representasikan dalam SKU yang unik.

Untuk melakukan ***hashing*** ini diperlukan ***hash table***, yaitu sebuah array integer yang memetakan indeks tersebut ke dalam objek yang ingin dicari. Sebagai contoh, misal ada 3 mahasiswa, yaitu:

- Anduk dengan NPM 0
- Budi dengan NPM 1
- Cegu dengan NPM 2

Maka ***hash table*** yang terbentuk adalah:

index	value
0	Anduk
1	Budi
2	Cegu

Sehingga jika ingin mencari Anduk, kita dapat melihat di ***hash table*** dengan index NPM nya, yaitu 0.

Permasalahnnya sekarang adalah:

1. Jika ingin menyimpan nama Anduk, Budi, dan Cegu berdasarkan namanya (bukan npm), bagaimana cara memetakan String tersebut menjadi index bilangan bulat?
2. Bagaimana memastikan bahwa bila terdapat nilai pemetaan yang sama (kolisi), hash table dapat tetap berfungsi dengan baik?

Hash Function merupakan fungsi untuk memetakan objek tadi kedalam sesuatu yang dapat disimpan ke **hash table** tersebut, contohnya adalah integer. Hasil dari ***hash function*** ini adalah ***hash values***. Membuat fungsi hash yang bagus memenuhi kriteria berikut:

1. Tingkat kolisinya (Ada 2 objek yang memiliki **hash value** yang sama) kecil.
2. Gampang dan murah untuk di komputasikan.
3. Value nya lebih terdisitribusi, tidak terjadi *small clustering*.

Sebagai contoh rumus ***hash function*** untuk mahasiswa diatas adalah

```
int hashFunction(String st) {
      st = st.toLowerCase();
      int hash = 0;
      for(int i = 0 ; i < st.length(); i++) {
            hash =+ st.charAt(0) - ‘a’;
      }
      return hash % TABLE_SIZE;
}
```

TABLE_SIZE yang digunakan disarankan adalah **bilangan prima** agar nilai nya lebih terdistribusi secara rata.

Perhatikan bahwa hashFunction tersebut merubah string menjadi penjumlahan indeks alphabet huruf-hurufnya (“a” -> 0, “b” -> 1, dst). **Apakah menurut anda hash function ini sudah baik**?

Bagaimana jika hash tersebut kolisi (nilai pemetaan yang sama)? Kolisi pada **hash table** dapat diatasi dengan berbagai cara. Salah satunya adalah **hashing with chaining**,

yaitu value disimpan sebagai kumpulan elemen-elemen dengan hash yang sama (umumnya menggunakan linked list ataupun array).

Sehingga untuk NPM diatas (perhatikan huruf kapitalnya) jika TABLE_SIZE adalah 1009 (bilangan prima):

- Nilai hash dari Anduk adalah 46
- Nilai hash dari Budi adalah 32
- Nilai hash dari Cegu adalah 32

Sehingga ***hash table*** yang terbentuk adalah

index	Value
0	[]
...	[]
32	[Budi, Cegu]
..	[]
46	[Anduki]

B. Soal

Implementasikan permasalahan perpustakaan berikut dengan Hash Table yang anda buat sendiri

Input dan Output

Program akan menerima input untuk *command* seperti dibawah ini hingga *End of File* (EOF):

1. **PINJAM <NAMA> <BUKU>**
   - **<NAMA>** adalah sebuah string alfabet lowercase yang menyatakan nama peminjam. Panjang string **<NAMA>** dijamin kurang dari sama dengan 50 karakter.
   - **<BUKU>** adalah sebuah string alfabet lowercase yang menyatakan judul buku yang dipinjam. Panjang string **<BUKU>** dijamin kurang dari sama dengan 50 karakter.
   - dijamin tidak ada kasus dimana seseorang meminjam buku dengan judul yang sama lebih dari sekali.
   - Untuk operasi berikut tidak ada output yang dicetak.
2. **DAFTAR_PEMINJAM <BUKU>**
   - Cari daftar peminjam dari buku **<BUKU>**.
   - **<BUKU>** adalah sebuah string alfabet lowercase yang menyatakan judul buku yang dipinjam. Panjang string **<BUKU>** dijamin kurang dari sama dengan 50 karakter.
   - dijamin buku dengan nama **<BUKU>** tersedia dan **sedang ada yang meminjam** sehingga anda tidak perlu khawatir untuk kasus-kasus corner.
   - Keluarkan nama-nama orang yang meminjam buku tersebut terurut secara leksikografis, pisahkan dengan spasi untuk setiap 2 peminjam berbeda.
     "<NAMA_1> <NAMA_2> ... <NAMA_K>"
3. **KEMBALI <NAMA> <BUKU>**
   - Orang dengan nama **<NAMA>** mengembalikan buku dengan judul **<BUKU>**.
   - **<NAMA>** adalah sebuah string alfabet lowercase yang menyatakan nama peminjam. Panjang string **<NAMA>** dijamin kurang dari sama dengan 50 karakter.
   - **<BUKU>** adalah sebuah string alfabet lowercase yang menyatakan judul buku yang dipinjam. Panjang string **<BUKU>** dijamin kurang dari sama dengan 50 karakter.

- dijamin orang dengan nama **<NAMA>** sedang meminjam buku dengan judul **<BUKU>.**
- Untuk operasi berikut tidak ada output yang dicetak.

Contoh Input

```
PINJAM mike pintarsdatanpadukun
PINJAM dustin pintarsdatanpadukun
PINJAM will carakeluarupsidedown
DAFTAR_PEMINJAM pintarsdatanpadukun
KEMBALI dustin pintarsdatanpadukun
DAFTAR_PEMINJAM pintarsdatanpadukun
```

Contoh Output

```
dustin mike
mike
```

Perhatian

- perhatikan apabila ada kemungkinan 2 buku berbeda dengan hash yang sama pada algoritma anda, jangan sampai daftar peminjamnya tercampur!
- Untuk menghindari karakter yang tidak terduga, tolong jangan copy paste contoh input, contoh output, maupun format output.