Penerapan Pohon Dalam *Heap Sort*

Firdi Mulia
Jurusan Teknik Informatika ITB, Bandung, email: if17045@students.if.itb.ac.id

Abstract **–** *Makalah ini membahas tentang penerapan pohon heap dalam metode pengurutan data heap sort. Pohon heap seperti namanya adalah struktur data berbentuk pohon yang memenuhi sifat-sifat heap. Heap mempunyai beberapa jenis variasi yaitu binary heap, binomial heap, dan fibonacci heap yang mempunyai keunggulan dan kelemahan tersendiri sehingga jenis mana yang kita pilih tergantung dari kasus yang kita hadapi. Struktur data heap ini digunakan dalam algoritma heap sort. Representasi struktur data ini bisa dengan larik biasa atau representasi dinamik. Algoritma heap sort terdiri dari 3 proses yaitu heapify, remove, dan reheapify.*

Kata Kunci*: pohon heap, heap sort, binary heap, binomial heap, Fibonacci heap, heapify, remove, reheapify, pengurutan data.*

1. PENDAHULUAN

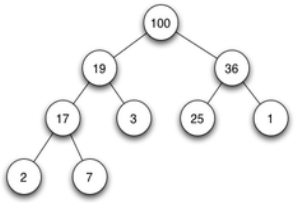
Konsep pohon merupkan salah satu konsep yang paling penting dalam teori graf. Pohon memiliki banyak variasi dan banyak diimplementasikan dalam berbagai bidang studi. Salah satu variasi pohon adalah pohon *heap* dan pohon ini dimanfaatkan dalam algoritma pengurutan data *heap sort* yang mempunyai kompleksitas asimptotiknya O(N log N). Kompleksitas ini sama dengan *quick sort*, bahkan mengungguli *quick sort* pada kasus terburuk sehingga algoritma pengurutan data ini terkenal.

2. *HEAP*

2.1 Pengertian *Heap*

Pohon *heap* adalah struktur data yang berbentuk pohon yang memenuhi sifat-sifat *heap* yaitu jika B adalah anak dari A, maka nilai yang tersimpan di simpul A lebih besar atau sama dengan nilai yang tersimpan di simpul B. Hal ini mengakibatkan elemen dengan nilai terbesar selalu berada pada posisi akar, dan *heap* ini disebut *max heap*. (Bila perbandingannya diterbalikkan yaitu elemen terkecilnya selalu berada di simpul akar, *heap* ini disebut adalah *min heap*). Karena itulah, *heap* biasa dipakai untuk mengimplementasikan *priority queue*. Operasi-operasi yang digunakan untuk *heap* adalah :

- *Delete-max* atau *delete-min*: menghapus simpul akar dari sebuah *max-* atau *min-heap*.
- *Increase-key* atau *decrease-key*: mengubah nilai yang tersimpan di suatu simpul.
- *Insert*: menambahkan sebuah nilai ke dalam *heap*.
- *Merge*: menggabungkan dua *heap* untuk membentuk sebuah *heap* baru yang berisi semua elemen pembentuk *heap* tersebut.



Gambar 1: Contoh dari *max heap*

2.2 Jenis-jenis *Heap*

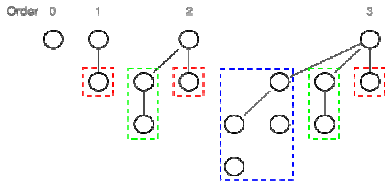
2.2.1 *Binary heap*

Binary heap adalah *heap* yang dibuat dengan menggunakan pohon biner.

2.2.2 *Binomial heap*

Binomial heap adalah *heap* yang dibuat dengan menggunakan pohon binomial. Pohon binomial bila didefinisikan secara rekursif adalah:

- Sebuah pohon binomial dengan tinggi 0 adalah simpul tunggal
- Sebuah pohon binomial dengan tinggi *k* mempunyai sebuah simpul akar yang anak-anaknya adalah akar-akar pohon-pohon binomial dengan tinggi *k*-1,*k*-2,…,2,1,0.

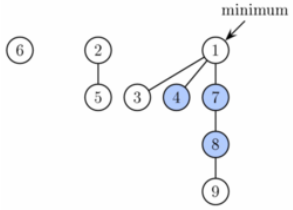


Gambar 2 : pohon-pohon binomial dengan tinggi (order) 0 sampai 3

2.2.3 *Fibonacci Heap*

Fibonacci heap adalah kumpulan pohon yang membentuk minimum *heap*. Pohon dalam struktur data ini tidak memiliki bentuk yang tertentu dan pada kasus yang ekstrim *heap* ini memiliki semua elemen dalam pohon yang berbeda atau sebuah pohon tunggal dengan tinggi n. Keunggulan dari

Fibonacci heap adalah ketika menggabungkan *heap* cukup dengan menggabungkan dua list pohon.



Gambar 2: Contoh *Fibonacci heap*

2.3 Perbandingan kompleksitas jenis-jenis *heap*

Tabel 1. Perbadingan macam-macam *heap*

Operasi	*Binary Heap*	*Binomi-al Heap*	*Fibonac-ci Heap*
Membuat *heap*	Θ(n)	Θ(1)	Θ(1)
Mencari nilai minimum	Θ(1)	Θ(log n)	Θ(1)
Menghapus nilai minimum	Θ(log n)	Θ(log n)	O(log n)
Menambah suatu elemen	Θ(log n)	O(log n)	Θ(1)
Mengubah nilai sebuah elemen dalam *heap*	Θ(log n)	Θ(log n)	Θ(1)
Menggabungkan dua *heap*	Θ(n)	O(log n)	Θ(1)

3. *HEAP SORT*

Heap Sort adalah algoritma pengurutan data berdasarkan perbandingan, dan termasuk golongan *selection sort*. Walaupun lebih lambat daripada *quick sort* pada kebanyakan mesin , tetapi *heap sort* mempunyai keunggulan yaitu kompleksitas algoritma pada kasus terburuk adalah n log n.

Algoritma pengurutan *heap sort* ini mengurutkan isi suatu larik masukan dengan memandang larik masukan sebagai suatu *Complete Binary Tree* (CBT). Setelah itu *Complete Binary Tree* (CBT) ini dapat dikonversi menjadi suatu *heap tree*. Setelah itu *Complete Binary Tree* (CBT) diubah menjadi suatu *priority queue*.

Algoritma pengurutan *heap* dimulai dari membangun sebuah *heap* dari kumpulan data yang ingin diurutkan, dan kemudian menghapus data yang mempunyai nilai tertinggi dan menempatkan dalam akhir dari larik yang telah terurut. Setelah memindahkan data dengan nilai terbesar, proses berikutnya adalah membangun ulang *heap* dan memindahkan nilai terbesar pada *heap* tersebut dan menempatkannya dalam tempat terakhir pada larik terurut yang belum diisi data lain. Proses ini berulang sampai tidak ada lagi data yang tersisa dalam *heap* dan larik yang terurut penuh.

Dalam implementasinya kita membutuhkan dua larik – satu untuk menyimpan *heap* dan satu lagi untuk menyimpan data yang sudah terurut. Tetapi untuk optimasi memori, kita dapat menggunakan hanya satu larik saja. Yaitu dengan cara menukar isi akar dengan elemen terakhir dalam *heap tree*. Jika memori tidak menjadi masalah maka dapat tetap menggunakan dua larik yaitu larik masukan dan larik hasil.

Heap Sort memasukkan data masukan ke dalam struktur data *heap*. Nilai terbesar (dalam *max-heap*) atau nilai terkecil (dalam *min-heap*) diambil satu per satu sampai habis, nilai tersebut diambil dalam urutan yang terurut.

Algoritma untuk heap sort :

```
function heapSort(a, count) is
    input:  sebuah larik tidak
            terurut a dengan
            panjang length

     (pertama letakkan a dalam max-
      heap)
     heapify(a, count)

     end := count - 1
     while end > 0 do
         remove()
        reheapify()
         end := end - 1
```

3.1 Algoritma *Heapify*

Algoritma *Heapify* adalah membangun sebuah *heap* dari bawah ke atas, secara berturut-turut berubah ke bawah untuk membangun *heap*.

Permasalahan pertama yang harus kita pertimbangkan dalam melakukan operasi *heapify* adalah dari bagian mana kita harus memulai. Bila kita mencoba operasi *heapify* dari akar maka akan terjadi operasi runut-naik seperti algoritma *bubble sort* yang akan menyebabkan kompleksitas waktu yang ada akan berlipat ganda.

Sebuah versi lain adalah membangun *heap* secara atas-bawah dan berganti-ganti ke atas untuk secara konseptual lebih sederhana untuk ditangani. Versi ini mulai dengan sebuah *heap* kosong dan secara berturut-turut memasukkan data. Versi lainnya lagi adalah dengan membentuk pohon *heap*-pohon *heap* mulai dari *subtree-subtree* yang paling bawah. Jika *subtree-subtree* suatu simpul sudah membentuk *heap* maka pohon dari simpul tersebut mudah dijadikan pohon heap dengan mengalirkannya ke bawah.

Setelah diuji, maka ide yang paling efisien adalah versi yang terakhir, yang kompleksitas algoritmanya pada kasus terburuk adalah O(n), sedangkan versi membentuk *heap tree-heap tree* dari atas ke bawah kompleksitas nya O(n log n).

Jadi, algoritma utama *heapify* adalah melakukan iterasi mulai dari internal simpul paling kanan-bawah (pada representasi larik, adalah elemen yang berada di indeks paling besar) hingga akar, kemudian kea- rah kiri dan naik ke level di atasnya, dan seterusnya hingga mencapai akar (sebagai larik [0..N-1]). Oleh karena itu, iterasi dilakukan mulai dari j= N/2 dan berkurang satu-satu hingga mencapai j=0.

Pada simpul internal tersebut, pemeriksaan hanya dilakukan pada simpul anaknya langsung (tidak pada level-level lain di bawahnya). Pada saat iterasi berada di level yang lebih tinggi, *subtree-subtree* selalu sudah membentuk *heap*. Jadi, kasus yang paling buruk adalah restrukturisasi hanya akan mengalirkan simpul tersebut kearah bawah. Dengan demikian, *heapify* versi ini melakukan sebanyak N/2 kali iterasi, dan pada kasus yang paling buruk akan melakukan iterasi sebanyak $^2\log$ (N) kali.

3.2 Algoritma *Remove*

Algoritma *remove* ini menukar akar (yang berisi nilai maksimum) dari heap dengan elemen terakhir. Secara logika, simpul yang berada paling kana-bawah dipindahkan ke akar untuk menggantikan simpul akar yang akan diambil.

3.3 Algoritma *Reheapify*

Algoritma *reheapify* ini melakukan pembuatan ulang *heap* dari atas ke bawah seperti halnya iterasi terakhir dari algoritma metoda *heapify*. Perbedaan antara metode *heapify* dengan metode *reheapify* ada pada iterasi yang dilakukan oleh kedua algoritma tersebut. Algoritma metode *reheapify* ini hanya melakukan iterasi terakhir dari algoritma *heapify*. Hal ini disebabkan baik *subtree* kiri maupun *subtree* kanannya sudah merupakan *heap*, sehingga tidak perlu dilakukan iterasi lengkap seperti algoritma *heapify*. Dan setelah *reheapify* maka simpul yang akan diiterasikan berikutnya akan berkurang satu.

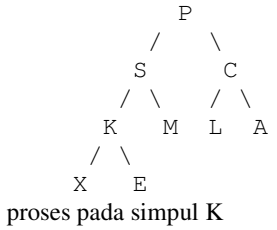
4. CONTOH IMPLEMENTASI ALGORITMA *HEAP SORT*

Salah satu contoh penerapan algoritma pengurutan *heap sort* adalah sebagai berikut : Misalkan terdapat sebuah larik karakter yang ingin diurutkan secara menurun sebagai berikut :

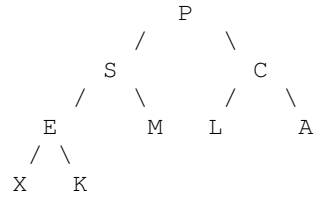
P	S	C	K	M	L	A	X	E
0	1	2	3	4	5	6	7	8

Untuk mengubah bentuk ini ke dalam *heap*, pertama yang harus dilihat adalah indeks simpul *parent* yang dapat dicari dengan (Ukuran *Heap*-2)/2. Dalam kasus ini, (9-2)/2=3. Kemudian kita terapkan algoritma heapify untuk membuat *heap*. Iterasi dilakukan pada tiap simpul dari indeks terakhir ke indeks 0.

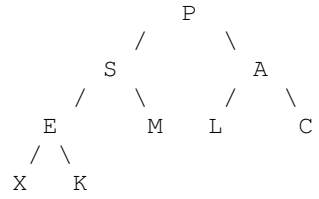
Pada contoh di atas, larik bisa disusun menjadi pohon biner seperti di bawah ini



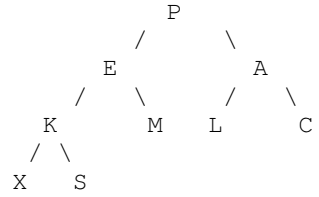
Sekarang proses pada simpul K



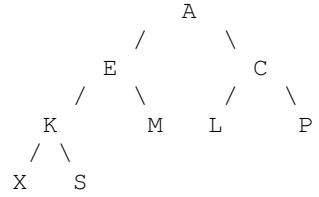
Lalu proses pada indeks 2, yaitu pada simpul C



Sekarang proses pada simpul di indeks 1, yaitu simpul S. Periksa anak dari simpul tersebut yaitu E, dan kemudian periksa juga anak dari simpul E yaitu K.
Baik E dan K naik ke atas.



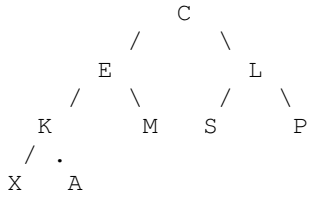
Proses simpul indeks 0, yaitu simpul akar. Periksa anaknya yaitu C. Baik A dan C naik ke atas.



Sekarang *heap* telah terbentuk. Langkah utama pertama telah selesai. Bagian lain dari algoritma *heap sort* adalah: secara berulang menghapus akar, membentuk ulang *heap*, dan menempatkan nilai dari akar yang dihapus itu ke dalam ujung larik yang telah diurutkan.

Pertama A dihapus dari *heap*, bentuk ulang *heap*

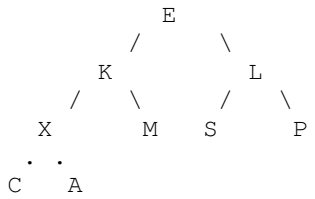
dengan memproses pada akar, dan kemudian menempatkan A pada ujung larik yang kita siapkan untuk menyimpan data yang telah terurut (karena A bukan bagian dari *heap* maka pada pohon di bawah ini A tidak terhubung ke pohon).



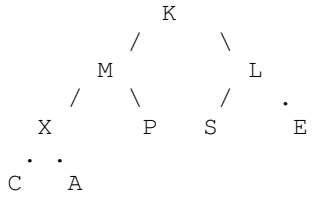
Sekarang larik tersebut menjadi

C	E	L	K	M	S	P	X	A
0	1	2	3	4	5	6	7	8

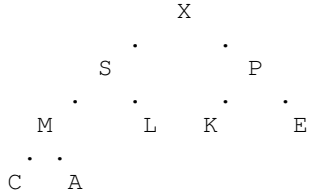
Langkah berikutnya adalah menghapus C dan menempatkannya pada ujung *heap*



Kemudian E dihapus dan ditempatkan pada ujung *heap*



Begitu seterusnya, akar dari *heap* akan dihapus dari *heap* dan ditempatkan pada ujung larik sampai



Karena hanya X yang tersisa dalam heap, dan karena pada proses di atas, huruf-huruf yang lebih kecil dari X telah dihapus dari heap maka X adalah huruf terbesar, dan tidak perlu dipindahkan karena telah berada di indeks paling awal dari larik. Sekarang lariknya telah terurut menurun :

X	S	P	M	L	K	E	C	A
0	1	2	3	4	5	6	7	8

5. REPRESENTASI ALOKASI DINAMIS ALGORITMA PENGURUTAN *HEAP SORT*

Karakteristik dari algoritma pengurutan *heap sort* adalah bahwa dalam implementasinya *heap sort* menggunakan *heap tree* agar dapat diselesaikan secara *heap sort*. Oleh karena itu, untuk mengimplementasikan algoritma pengurutan *heap sort* dalam suatu program aplikasi, dibutuhkan adanya alokasi dinamis dengan menggunakan struktur data *tree* (pohon).

Prinsip-prinsip dasar mengenai struktur data *tree* yang digunakan untuk merealisasikan *heap tree* adalah sebagai berikut:

a. *Simpul-simpul* saling berhubungan dengan menggunakan pointer. Pada struktur data *tree* ini digunakan minimal dua buah pointer pada setiap *simpul,*masing-masing untuk menunjuk ke cabang kiri dan cabang kanan dari tree tersebut. Misalnya dalam bahasa C, struktur data *tree* dideklarasikan sebagai berikut:

```
class BinaryTreeSimpul {
        KeyType      Key;
       InfoType      Info;
   BinaryTreeSimpul  Left,
Right;      // metoda-metoda
}
```

b. *Left* dan *Right* berharga NULL apabila tidak ada lagi cabang pada arah yang bersangkutan.

c. Struktur dari binary tree, termasuk hubungan-hubungan antar-*simpul,* secara eksplisit direpresentasikan oleh *Left* dan *Right*. Apabila diperlukan penelusuran naik (*backtrack)*, maka hal tersebut dapat dilakukan dengan penelusuran ulang dari *root,* penggunaan algoritma-algoritma yang bersifat rekursif, atau penggunaan stack.

d. Alternatif lain adalah dengan menambahkan adanya pointer ke parent. Namun hal ini akan mengakibatkan bertambahnya jumlah tahapan pada proses-proses penambahan/penghapusan simpul.

6. PERBANDINGAN DENGAN ALGORITMA PENGURUTAN LAIN

Heapsort hampir setara dengan *quick sort*, algoritma pengurutan data lain berdasarkan perbandingan yang sangat efisien.

Quick sort sedikit lebih cepat, karena *cache* dan faktor-faktor lain, tetapi pada kasus terburuk kompleksitasnya $O(n^2)$, yang sangat lambat untuk data yang berukuran sangat besar. Lalu karena *heap sort* memiliki $\Theta(N \log N)$ maka sistem yang memerlukan pengamanan yang ketat biasa memakai *heap sort* sebagai algoritma pengurutannya.

Heap sort juga sering dibandingkan dengan *merge sort*, yang mempunyai kompleksitas algoritma yang sama, tetapi kompleksitas ruang nya Ω(n) yang lebih besar dari *heap sort*. *Heap sort* juga lebih cepat pada mesin dengan *cache data* yang kecil atau lambat.

7. KESIMPULAN

Dengan memanfaatkan struktur data pohon, kita bisa mendapatkan algoritma pengurutan data yang mangkus yang bisa dimanfaatkan untuk membangun program aplikasi yang baik.

Algoritma pengurutan *heap sort* bisa dimasukkan ke dalam algoritma *divide and conquer* yang disebabkan pembagian dilakukan dengan terlebih dahulu menerapkan algoritma metoda *heapify* sebagai inisialisasi untuk mentransformasi suatu *tree* menjadi *heap tree*, dan pada setiap tahapan diterapkan algoritma metoda *reheapify* untuk menyusun ulang *heap tree*.

DAFTAR REFERENSI

[1] CS210: Data Structures. http://www.cse.iitk.ac.in/users/dsrkg/cs210/applets/sorting/heapSort; Tanggal akses 30 Desember 2008

[2] Heapsort – Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Heap_sort; Tanggal akses 30 Desember 2008.

[3] Heap (data structure) – Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Heap_(data structure); Tanggal akses 30 Desember 2008.

[4] Binomial heap – Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Binomial_heap; Tanggal akses 30 Desember 2008.

[5] Binary heap – Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Heap_sort; Tanggal akses 30 Desember 2008.

[6] Fibonacci heap – Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Fibonacci_heap; Tanggal akses 30 Desember 2008.

[7] Struktur Data Hirarkis: Heaptree, Binary Tree Traversal. http://ranau.cs.ui.ac.id/sda/archive/1998/handout/handout15.html; Tanggal akses 31 Desember 2008.

[8] Heaps and Heapsort http://cis.stvincent.edu/html/tutorials/swd/heaps/heaps.html; Tanggal akses 31 Desember 2008.