# Heap Tree dan Kegunaannya dalam Heap Sort

Efendy Chalikdjen[1], Hermanto Ong[2], Satria Putra Sajuthi[3]

*Laboratorium Ilmu dan Rekayasa Komputasi*
*Departemen Teknik Informatika, Institut Teknologi Bandung*
*Jl. Ganesha 10, Bandung*

E-mail : if13068@students.if.itb.ac.id[1], if13069@students.if.itb.ac.id[2],
if13099@students.if.itb.ac.id[3]

## Abstrak

Mahasiswa Teknik Informatika dan orang-orang lain yang berkecimpung dalam bidang pemrograman (*programming*) sering sekali menghadapi masalah pengurutan dalam membangun sebuah program aplikasi. Misalnya saja dalam membangun sebuah aplikasi pengolah data mahasiswa, *programmer* akan dihadapkan pada masalah pengurutan data mahasiswa berdasarkan atribut tertentu, seperti nama, NIM, nilai, dan sebagainya. Di dalam bidang Teknik Informatika sendiri terdapat banyak sekali jenis-jenis algoritma pengurutan yang dapat digunakan untuk memecahkan masalah pengurutan tersebut, di antaranya adalah *bubble sort*, *merge sort*, *insertion sort*, *quick sort*, dan *selection sort*, serta masih banyak lagi jenis algoritma pengurutan lainnya. Oleh karena itu, teknik dan kejelian untuk memilih algoritma pengurutan yang tepat dan sesuai dengan permasalahan pengurutan yang dihadapi sangat diperlukan karena masing-masing algoritma pengurutan tersebut memiliki karakteristik yang berbeda-beda. Penulis memilih untuk mengangkat tema *Heap Sort* dalam makalah ini sebab *heap sort* merupakan salah satu algoritma pengurutan yang memiliki kompleksitas waktu asimptotik terbaik. Selain itu juga, *heap sort* menerapkan teknik yang unik di dalam memecahkan masalah pengurutan, yaitu dengan menggunakan *heap tree*. Pada makalah ini akan dibahas dan dianalisis *heap tree* dan kegunaanya dalam *heap sort*, serta contoh sederhana mengenai cara penerapan algoritma pengurutan *heap sort* dalam memecahkan suatu masalah pengurutan.

***Kata kunci**: algoritma pengurutan, mangkus, kompleksitas waktu asimptotik, heap sort, heap tree.*

## 1. Pendahuluan

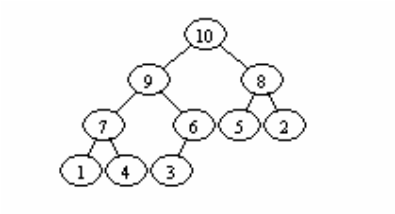
Untuk memecahkan masalah pengurutan dalam membangun suatu program aplikasi, dibutuhkan algoritma pengurutan. Di dalam bidang Teknik Informatika terdapat banyak sekali jenis-jenis algoritma pengurutan yang dapat digunakan untuk memecahkan masalah pengurutan. Oleh karena itu, teknik untuk memilih algoritma pengurutan yang tepat, sesuai dengan kebutuhan, dan mangkus sangat diperlukan karena masing-masing algoritma pengurutan memiliki karakteristik yang berbeda-beda. *Heap sort* merupakan salah satu contoh algoritma pengurutan yang memiliki kompleksitas waktu asimptotik terbaik serta menerapkan teknik yang unik/khas di dalam memecahkan masalah pengurutan, yaitu dengan menggunakan *heap tree*.

## 2. *Heap Tree* dan *Priority Queue*

### 2.1 Pengertian *Heap Tree*

Secara umum, pengertian dari *heap* adalah bagian dari memori yang terorganisasi untuk dapat melayani alokasi memori secara dinamis [2].

Suatu *heap tree* adalah *Complete Binary Tree* (CBT) di mana harga-harga *key* pada *node-node*nya sedemikian rupa sehingga haga-harga *key* pada *node-node* anaknya tidak ada yang lebih besar dari harga *key* pada *node* orang tuanya [2].




Pengertian "lebih besar" di atas tidak mutlak, untuk beberapa kasus maka dapat diganti sesuai dengan permasalahan yang dihadapi.

### 2.2 Implementasi *Priority Queue*

Berdasarkan pengertian di atas maka *heap tree* bermanfaat untuk mengimplementasikan *priority queue*. *Priority queue* merupakan struktur data yang sifatnya sangat mirip dengan antrian, yaitu penghapusan/pengurangan anggota selalu dilakukan pada anggota antrian yang terdepan dan penambahan anggota selalu dilakukan dari belakang antrian berdasarkan prioritas anggota tersebut (anggota yang memiliki prioritas lebih besar selalu berada di depan anggota yang memiliki prioritas lebih rendah).

Sebagai suatu *priority queue*, *heap tree* memerlukan beberapa metoda sebagai berikut: [2]

a. Metoda untuk menginisialisasi suatu CBT (*Complete Binary Tree*) a secara umum menjadi *heap tree*.
b. Metoda untuk mengambil data paling besar, yaitu *root* dari *heap tree*.
c. Metoda untuk menambahkan satu *key* baru ke dalam *heap tree*.
d. Metoda untuk menyusun ulang menjadi *heap tree* kembali setelah dilakukan metoda b atau c

Kriteria yang penting untuk dipenuhi adalah bahwa setiap metoda di atas beroperasi pada *tree* yang selalu berbentuk CBT (*Complete Binary Tree*) karena struktur level lebih rendahnya tetap merupakan suatu *array*[2].

## 3. Algoritma Pengurutan(*Sorting Algorithm*) *Heap Sort*

Contoh penggunaan *heap tree* dalam *priority queue* dapat kita lihat pada algoritma pengurutan *heap sort*. Algoritma pengurutan ini mengurutkan isi suatu *array* masukan dengan memandang *array* yang dimasukkan sebagai suatu *Complete Binary Tree* (CBT). Dengan metoda *a* maka *Complete Binary Tree* (CBT) ini dapat dikonversi menjadi suatu *heap tree*. Setelah *Complete Binary Tree* (CBT) berubah menjadi suatu *priority queue*, maka dengan mengambil data *root* satu demi satu dan disertai dengan metoda d, *key*-*key* dari data *root* yang kita ambil secara berturutan itu akan terurut dengan *output* hasil pengurutan akan dituliskan dalam *array* hasil dari arah kanan ke kiri.

Untuk optimasi memori, kita dapat menggunakan hanya satu array saja. Yaitu dengan cara memodifikasi metoda b untuk menukar isi root dengan elemen terakhir dalam heap tree. Jika memori tidak menjadi masalah maka dapat tetap menggunakan 2 array yaitu array masukan dan array hasil.

Algoritma utama *heap sort*:[2]

```
heapify()
for i ← 0 to n-1 do
   remove()
   reheapify()
endfor
```

### 3.1 Algoritma Metoda *Heapify*

Ide pertama yang harus kita pikirkan untuk melakukan operasi heapify adalah dari bagian mana kita harus memulai. Bila kita mencoba dari heapify dari root maka akan terjadi operasi runut-naik seperti algoritma *bubble sort* yang akan menyebabkan kompleksitas waktu yang ada akan berlipat ganda. Setelah diuji, dengan berbagai hasil maka ide yang efisien adalah membentuk *heap tree* - *heap tree* mulai dari sub*tree*-sub*tree* yang paling bawah. Jika sub*tree*-sub*tree* suatu *node* sudah membentuk *heap* maka *tree* dari *node* tersebut mudah dijadikan *heap tree* dengan mengalirkannya ke bawah.

Jadi, algoritma utama *heapify* adalah melakukan iterasi mulai dari internal *node* paling kanan-bawah (atau berindeks *array* paling besar) hingga *root*, kemudian ke arah kiri dan naik ke level di atasnya, dan seterusnya hingga mencapai *root* (sebagai *array* [0..N-1] ). Oleh karena itu, iterasi dilakukan mulai dari j = N/2 dan berkurang satu-satu hingga mencapai j = 0.

Pada internal *node* tersebut, pemeriksaan hanya dilakukan pada *node* anaknya langsung (tidak pada level-level lain di bawahnya). Di samping itu, pada saat iterasi berada di level yang lebih tinggi, sub*tree*-sub*tree*nya selalu sudah membentuk *heap*. Jadi, kemungkinan yang paling buruk adalah restrukturisasi hanya akan mengalirkan *node* tersebut ke arah bawah. Dengan demikian, algoritma metoda *heapify* ini melakukan sebanyak N/2 kali iterasi, dan pada setiap interasi paling buruk akan melakukan pertukaran sebanyak $\log_2(N)$ kali.

### 3.2 Algoritma Metoda *Remove*

Algoritma metoda *remove* hanya menukarkan elemen *array* pertama dengan elemen *array* terakhir yang terdapat di dalam *heap tree*. Secara logika, *node* yang berada paling kanan-bawah dipindahkan ke *root* untuk menggantikan *node root* yang akan diambil.

### 3.3 Algoritma Metoda *Reheapify*

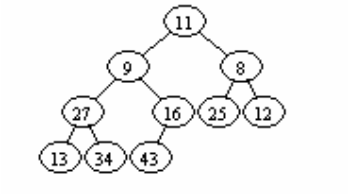
Algoritma metoda *reheapify* melakukan restrukturisasi dari atas ke bawah seperti halnya iterasi terakhir dari algoritma metoda *heapify*. Perbedaan antara metoda *heapify* dengan metoda *reheapify* terletak pada iterasi yang dilakukan oleh kedua algoritma tersebut. Algoritma metoda *reheapify* hanya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

## 4. Penerapan Algoritma Pengurutan *Heap Sort*

Salah satu contoh penerapan algoritma pengurutan (*sorting algorithm*) *heap sort* adalah sebagai berikut: Misalkan terdapat suatu *array* bilangan bulat yang terdiri dari sepuluh buah anggota dengan nilai data

11, 9, 8, 27, 16, 25, 12, 13, 34, dan 43. Kita akan mengurutkan data diatas dengan menggunakan heapsort.

Pertama-tama, *array* di atas dapat dipandang sebagai suatu Complete Binary Tree (CBT) sebagai berikut:

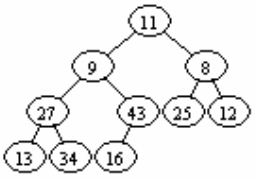


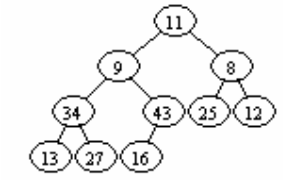

Selanjutnya algoritma metoda *heapify* dilakukan dengan iterasi dari sub*tree node* ke-4, ke-3, dan seterusnya berturut-turut hingga mencapai *root* (akar). Iterasi dilakukan mulai dari *node* ke-4 karena N/2 dalam contoh di atas adalah 5. Dan elemen kelima dari *array* memiliki nilai indeks 4 sebab indeks *array* biasanya diawali dari 0.

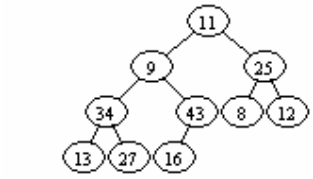
Penerapan algoritma metoda *heapify* terhadap *Complete Binary Tree* (CBT) pada contoh di atas menghasilkan operasi-operasi pertukaran sebagai berikut:

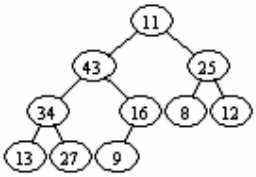
1. Sub*tree node* ke-4: pertukaran 16 dengan 43
2. Sub*tree node* ke-3: pertukaran 27 dengan 34
3. Sub*tree node* ke-2: pertukaran 8 dengan 25
4. Sub*tree node* ke-1: pertukaran 9 dengan 43, lalu pertukaran 9 dengan 16
5. Sub*tree node* ke-0: pertukaran 11 dengan 43, lalu pertukaran 11 dengan 34, serta akhirnya pertukaran 11 dengan 27

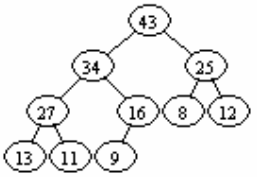
Perubahan-perubahan (pertukaran) tersebut dapat digambarkan sebagai berikut:



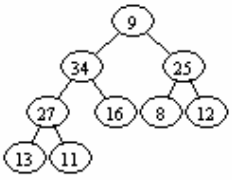



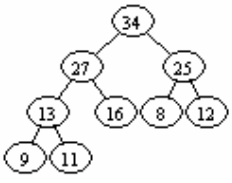

Semua perubahan di atas terjadi dalam *array* yang bersangkutan, sehingga pada akhirnya diperoleh *tree* terakhir yang merupakan *heap tree*.

Sementara itu, dalam iterasi yang melakukan/menerapkan algoritma metoda remove( ) dan algoritma metoda reheapify() akan terjadi pemrosesan berikut:

1. Setelah 43 di-*remove* dan 9 menggantikan posisi yang ditinggalkan oleh 43, maka terjadi *reheapify*: penukaran 9 dengan 34, 9 dengan 27, dan 9 dengan 13.

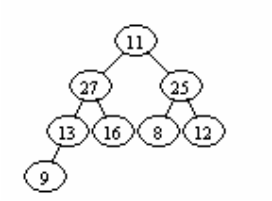



menjadi

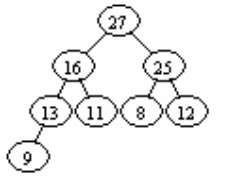



dan data yang telah terurut adalah 43.

2. Setelah 34 di-*remove* dan 11 menggantikan posisi yang ditinggalkan oleh 34, maka terjadi *reheapify*: penukaran 11 dengan 27, dan 11 dengan 16.

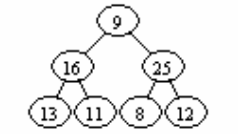



menjadi

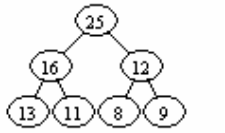



dan data yang telah terurut adalah 34, 43.

3. Setelah 27 di-*remove* dan 9 menggantikan posisi yang ditinggalkan oleh 27, maka terjadi *reheapify*: penukaran 9 dengan 25, dan 9 dengan 12.




menjadi




dan data yang telah terurut adalah 27, 34, 43.

4. Demikian seterusnya dilakukan algoritma metoda *remove* dan algoritma metoda *reheapify* hingga tidak ada lagi *node* yang tersisa. Dan pada akhirnya akan didapatkan hasil data yang telah terurut adalah 8, 9, 11, 12, 13, 16, 25, 27, 34, 43.

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

```
class BinaryTreeNode {
KeyType Key;
InfoType Info;
BinaryTreeNode Left, Right;
}
```

b. *Left* dan *Right* berharga NULL apabila tidak ada lagi cabang pada arah yang bersangkutan.
c. Struktur dari *binary tree*, termasuk hubungan-hubungan antar-*node*, 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 *node*.

## 6. Kesimpulan

Algoritma pengurutan *heap sort* dapat digunakan untuk menyelesaikan masalah-masalah pengurutan dalam membangun suatu program aplikasi dengan mangkus.

Algoritma pengurutan *heap sort* dapat dikategorikan ke dalam algoritma *divide and conquer* dengan pendekatan pengurutan sulit membagi, mudah menggabung (*hard split/easy join*) seperti halnya algoritma pengurutan *quick sort* dan *selection sort*. Hal ini disebabkan pembagian dilakukan dengan terlebih dahulu menerapkan algoritma metoda *heapify* sebagai inisialisasi awal untuk mentransformasi suatu *Complete Binary Tree* (CBT) menjadi *heap tree* dan pada setiap tahapan diterapkan algoritma metoda *reheapify* untuk menyusun ulang *heap tree*. Pembagiannya sendiri dilakukan dengan menerapkan metoda *remove* yang akan membagi data yang akan diurutkan menjadi dua bagian, masing-masing berukuran sebanyak satu dan sebanyak jumlah *node* di dalam *heap tree* dikurangi dengan satu. Sementara itu, proses penggabungannya sangat mudah sebab pada setiap tahapan, data terbesar dari tahap tersebut digabungkan dengan cara disisipkan di depan data terbesar yang diperoleh dari tahap sebelumnya.

Keunggulan algoritma pengurutan *heap sort* terletak pada kompleksitas waktu asimptotiknya yang sangat baik. Untuk melakukan algoritma metoda *heapify* dilakukan iterasi mulai dari elemen ke-N/2 sampai dengan elemen ke-1, dan pada setiap iterasi paling banyak akan dilakukan operasi pertukaran sebanyak $\log_2$ (N) kali. Oleh karena itu, kompleksitas waktu asimptotik algoritma metoda *heapify* adalah T(N) = N/2 * $\log_2$ (N) = O (N $\log_2$ (N)). Sementara itu,

algoritma metoda *remove* hanya melakukan pertukaran elemen pertama dengan elemen terakhir dalam *heap tree* sehingga kompleksitas waktu asimptotiknya adalah T(N) = O (a), a adalah konstanta. Sedangkan algoritma metoda *reheapify* hanya melakukan restrukturisasi ulang elemen *root* untuk membentuk *heap tree* setelah dilakukan algoritma metoda *remove*. Dengan demikian, kompleksitas waktu asimptotik algoritma metoda *reheapify* setara dengan T(N) = O ($\log_2$ (N)). Pada algoritma pengurutan (*sorting algorithm*) *heap sort*, algoritma metoda *heapify* dilakukan hanya satu kali sebagai inisialisasi sebelum iterasi, dan algoritma metoda *remove* dan *reheapify* dilakukan sebanyak jumlah iterasi, yaitu N. Jadi, kompleksitas waktu asimptotik algoritma pengurutan (*sorting algorithm*) *heap sort* adalah T(N) = (N/2 * $\log_2$ (N)) + (N * (a + $\log_2$ (N))) = N/2 $\log_2$ (N) + Na + N $\log_2$ (N) = 3N/2 $\log_2$ (N) + Na = bN $\log_2$ (N) + Na = O (N $\log_2$ (N)).

Karakteristik (ciri khas) dari algoritma pengurutan (*sorting algorithm*) *heap sort* terletak pada penggunaan *heap tree* sebagai sarana untuk menyelesaikan / memecahkan permasalahan pengurutan. Hal ini menjadi salah satu faktor yang membuat algoritma pengurutan (*sorting algorithm*) *heap sort* menjadi unik (khas), menarik, dan berbeda apabila dibandingkan dengan algoritma-algoritma pengurutan (*sorting algorithm*) lainnya.

## Daftar Pustaka


1. *http://www.cse.iitk.ac.in/users/dsrkg/cs210/applets/sortingII/heapSort/heap.html*. Diakses tanggal 17 Mei 2005 pukul 16.30 WIB.

2. *http://www.cs.ui.ac.id/kuliah/IKI10100/1998/handout/handout15.html*. Diakses tanggal 17Mei 2005 pukul 17.00 WIB.