Penerapan Struktur Data Pohon dalam Implementasi Algoritma *Heapsort* dan Tinjauan Kompleksitas Waktunya

Paskahlis Anjas Prabowo 13515108[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13515108@std.stei.itb.ac.id

***Abstract*— Dunia komputasi tidak bisa dihindarkan dari pengolahan data. Salah satu hal penting dalam mengolah data adalah prosedur pengurutan. Begitu banyak algoritma pengurutan yang dapat diterapkan dalam mengolah data, salah satunya adalah *Heapsort*. Algoritma ini menerapkan struktur data pohon dalam implementasinya. Kompleksitas waktu yang sama untuk semua kasus menjadikan *Heapsort* algoritma yang cocok untuk mengurutkan data dengan jumlah yang cukup besar. Pengurutan menggunakan algoritma *Heapsort* mengabaikan kestabilan. Kestabilan dalam hal ini adalah sifat untuk memprioritaskan urutan data awal jika ada nilai yang sama tapi tidak sejenis. Untuk itu penerapan *Heapsort* harus dihindari untuk kasus tersebut.**

***Keywords*—struktur data, algoritma heapsort, kompleksitas waktu.**

I. PENDAHULUAN

Dunia komputasi erat kaitannya dengan pengolahan data. Dalam pengolahan data terdapat beberapa prosedur umum sangat diperlukan. Salah satu prosedur yang sangat diperlukan dalam hal mengolah data adalah pengurutan. Pengurutan merupakan proses untuk mengurutkan. Mengurutkan bermakna menjadikan urut. Adapun makna urut adalah teratur [1]. Dengan demikian, dalam konteks ini pengurutan merupakan proses penyusunan ulang suatu data sedemikian sehingga susunan data yang baru memiliki keteraturan.

Kebutuhan akan prosedur pengurutan dalam berbagai urusan komputasi mengharuskan *programmer* untuk terus melakukan optimasi terhadap algoritma-algoritma pengurutan yang sudah ada. Banyak algoritma pengurutan yang dikenal selama ini, di antaranya adalah *Insertion Sort, Shell Sort, Bubble Sort, Selection Sort, Quicksort, Sample Sort, Mergesort, Bucketsort, Heapsort,* dan lain sebagainya [2]. Struktur data yang digunakan dalam implementasi algoritma-algoritma tersebut cukup beragam. Salah satu struktur data yang dapat diterapkan adalah struktur data pohon. Struktur data yang diterapkan dalam implementasi algoritma *Heapsort* adalah pohon biner atau dalam hal ini dikenal sebagai struktur data *heap*.

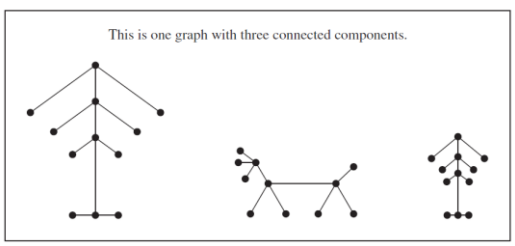
Heapsort merupakan salah satu algoritma pengurutan yang sering diterapkan dalam kasus yang mengharuskan komputasi data dengan jumlah yang cukup besar. Algoritma ini diterapkan dengan beberapa alasan di antarannya adalah kompleksitas waktu baik untuk kasus terburuk, rata-rata, maupun terbaik besarnya sama, yaitu *O(n log n)*. Hal ini menjamin bahwa, pertumbuhan waktu terhadap jumlah elemen yang diolah selalu mengikuti pola *O(n log n)* yang dalam hal ini lebih baik daripada kommpleksitas waktu pada kasus terburuk dari algoritma lain yang sebesar $O(n^2)$ [3].

Berdasarkan data dan ulasan tersebut, perlu adanya penguraian lebih lanjut mengenai algorima *Heapsort* baik secara konsep maupun implementasinya secara langsung sebagai kode program. Penguraian konsep dan implementasi *Heapsort* sekaligus dapat membuka kemungkinan ditemukannya algoritma pengurutan baru yang lebih mangkus daripada algoritma-algoritma yang sudah ada. Penguraian ini ditinjau dari struktur data maupun algoritmanya sebagai implementasi dari konsep dalam matematika diskrit.

II. TEORI DASAR

A. Pohon

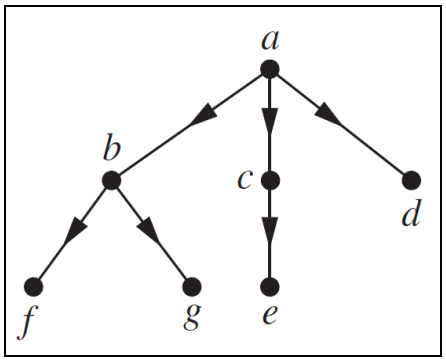
Pohon pada dasarnya adalah sebuah graf. Pohon merupakan graf tak-berarah yang terhubung dan tidak mengandung sirkuit. Graf tak-berarah dinyatakan sebagai pohon jika dan hanya jika terdapat lintasan yang unik antar setiap simpul yang terdapat pada graft tersebut [4]. Adapun kumpulan pohon yang saling lepas dinamakan hutan.



Gambar 2.1. Hutan

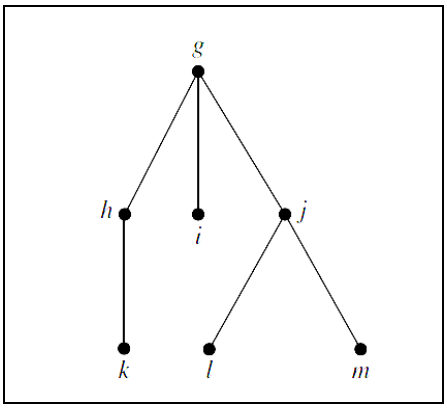
B. Pohon Berakar

Pohon berakar adalah pohon yang satu buah simpulnya diperlakukan sebagai akar dan setiap sisinya berarah menjauhi akar menuju simpul lain.



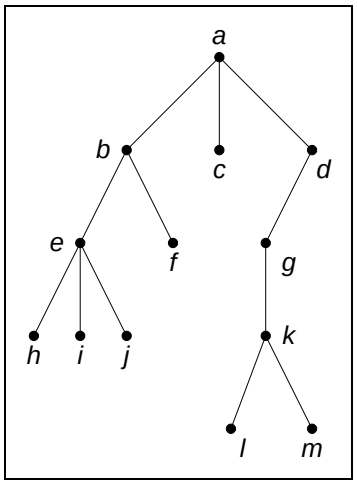
Gambar 2.2.a. Pohon Berakar

Dalam beberapa urusan, telah disepakati bahwa penandaan arah di setiap sisi pada pohon berakar bisa dihilangkan.



Gambar 2.2.b. Pohon Berakar

Terdapat beberapa terminologi khusus pada pohon berakar. Adapun istilah-istilah pada pohon berakar adalah sebagai berikut [5].



Gambar 2.3. Ilustrasi Terminologi

1. Anak dan Orang tua

Ilustrasi pada Gambar 2.3 menunjukkan bahwa *a* adalah orang tua dari *b*, *c*, dan *d*. Dengan demikian sebuah simpul yang saling bersisian pasti memiliki hubungan anak-orang tua.

2. Lintasan

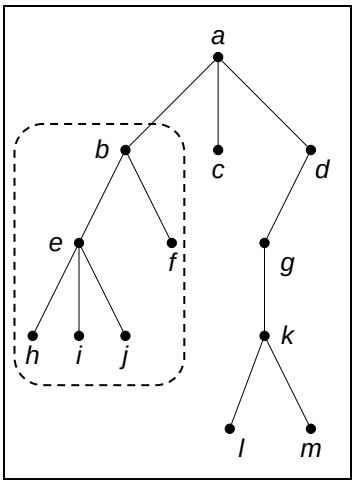
Berdasarkan ilustrasi pada Gambar 2.3, terdapat lintasan dari *a* ke *h*. Adapun lintasan tersebut adalah *a*, *b*, *e*, *h*. Dengan demikian definisi lintasan pada pohon sama dengan definisi lintasan yang ada pada graf.

3. Saudara Kandung

Terminologi saudara kandung disematkan kepada dua atau lebih simpul yang memiliki orang tua sama. Pada Gambar 2.3, dapat disimpulkan bahwa *h*, adalah saudara kandung *i*, begitupula dengan *j* karena simpul-simpul tersebut memiliki satu orang tua yang sama, yaitu *e*. Akan tetapi, *e* bukan saudara kandung dari *g* karena *e* dan *g* memiliki orang tua yang berbeda.

4. Upapohon

Pohon dapat dipandang sebagai struktur yang rekursif, dengan kata lain pohon terdiri dari akar dan pohon juga. Oleh karena itu terdapat istilah upapohon atau *subtree*.



Gambar 2.4. Ilustrasi Upapohon

5. Derajat

Derajat merupakan salah satu properti dari simpul pada pohon. Derajat sebuah simpul menyatakan jumlah upapohon pada simpul tersebut. Derajat sebuah simpul juga dapat menyatakan jumlah anak dari simpul tersebut. Pada Gambar 2.3, derajat *a* dan *e* adalah 3, derajat *b* dan k adalah 2, sedangkan derajat *d* adalah 1. Jadi, derajat dalam hal ini adalah derajat-keluar.

6. Daun

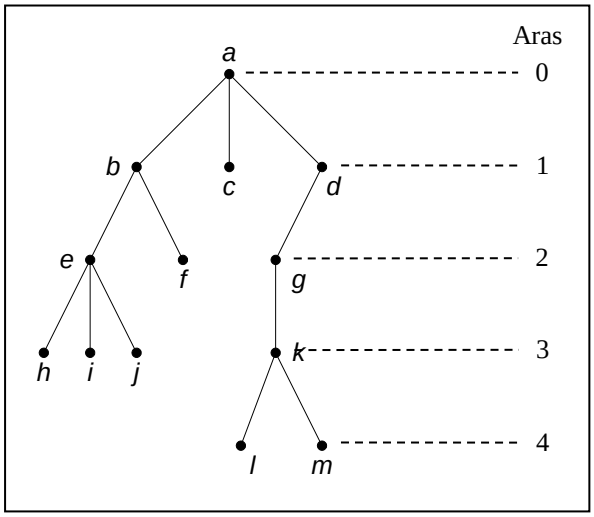
Simpul pada pohon yang tidak memiliki anak dinamakan daun. Dengan kata lain, daun merupakan simpul pada pohon yang berderajat nol. Daun-daun yang terdapat pada pohon pada Gambar 2.3 adalah *h*, *i*, *j*, *f*, *c*, *l*, dan *m*.

7. Simpul Dalam

Simpul yang mempunyai anak disebut simpul dalam. Adapun pohon pada Gambar 2.3 memiliki *b*, *e*, *d*, *g*, dan *k* sebagai simpul dalam.

8. Aras atau Tingkat

Aras merupakan salah satu properti simpul pada pohon. Aras menyatakan *level* suatu simpul pada pohon yang dihitung berdasarkan kedudukan relatuf simpul tersebut ditinjau dari akar utama.



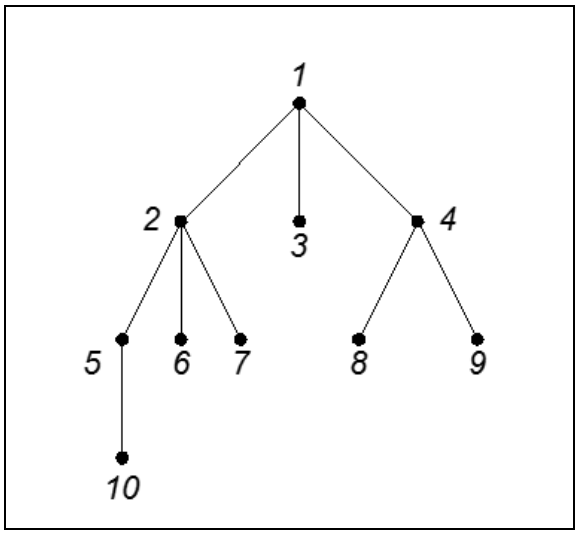
Gambar 2.5. Ilustrasi Aras

9. Tinggi atau Kedalaman

Tinggi atau kedalaman pada pohon menyatakan aras maksimum yang terdapat pada pohon tersebut. Pada Gambar 2.5, pohon tersebut memiliki tinggi atau kedalaman 4.

C. Pohon Terurut

Pohon berakar yang memperhatikan urutan anak-anaknya merupakan pohon terurut [5].



Gambar 2.6. Ilustrasi Pohon Terurut

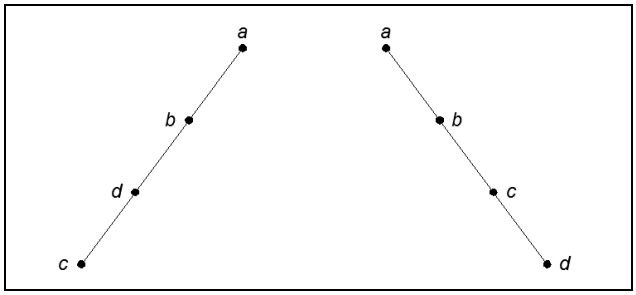
D. Pohon N-ary Algoritma

Pohon *n-ary* merupakan pohon berakar yang setiap simpul cabangnya memiliki masimal *n* buah anak [5]. Pohon *n-ary* dikatakan teratur atau penuh jika setiap simpul cabangnya memiliki tepat *n* anak. Pohon pada Gambar 2.3 merupakan pohon 3-*ary* karena simpul pada pohon tersebut memiliki maksimal 3 buah anak. Simpul yang memiliki 3 buah anak adalah simpul *a* dan *e*.

E. Pohon Biner

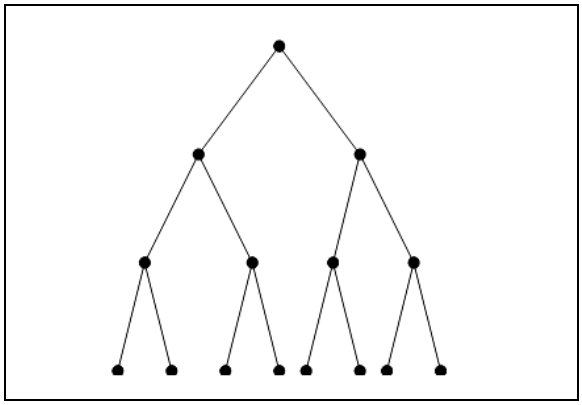
Pohon biner merupakan pohon *n-ary* dengan $n = 2$ [5]. Setiap simpul pada pohon biner memiliki maksimal 2 buah anak. Dua buah anak dari sebuah simpul pada pohon biner dipandang berbeda. Anak-anak dari simpul tersebut dibedakan menjadi anak kiri dan anak kanan. Pembedaan antara anak kiri dan anak kanan menyebabkan pohon biner juga tergolong pada pohon terurut.

Pohon biner memiliki beberapa terminologi khusus. Condong-kiri dan condong-kanan merupakan istilah untuk menyatakan sifat suatu pohon biner yang setiap simpulnya hanya memiliki anak kiri (condong-kiri) maupun anak kanan (condong-kanan).



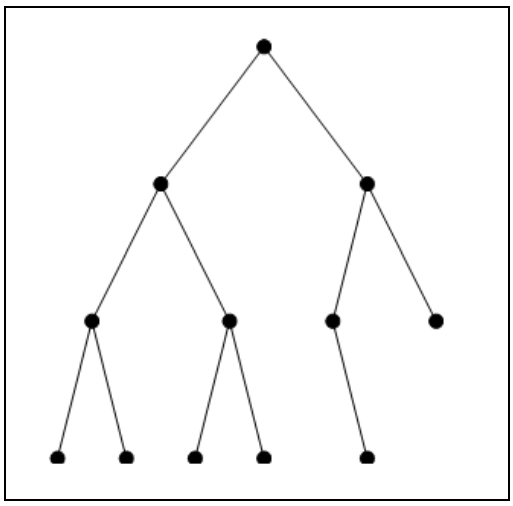
Gambar 2.7. Ilustrasi Pohon Condong-kanan dan Condong-kiri

Pohon biner penuh merupakan istilah untuk menyatakan bahwa semua simpul dalam suatu pohon biner memiliki 2 anak dengan tinggi yang sama.



Gambar 2.8. Ilustrasi Pohon Biner Penuh

Pohon biner seimbang merupakan istilah untuk menyatakan bahwa perbedaan tinggi antara upapohon kanan dan kiri maksimal satu.



Gambar 2.8. Ilustrasi Pohon Biner Seimbang

F. Kompleksitas Algoritma

Kompleksitas algoritma dinilai dari beberapa tinjauan. Tinjauan kompleksitas yang mungkin dilakukan terhadap suatu algoritma adalah kompleksitas ruang dan kompleksitas waktu [5]. Kommpleksitas waktu *T(n)* merupakan pengurukran terhadap jumlah tahapan komputasi yang dibutuhkan untuk menjalankan algoritma sebagai fungsi dari ukuran masukan *n*. Kompleksitas ruang *S(n)* dapat diukur melalui memori yang digunakan oleh struktur data yang terdapat di dalam algoritma sebagai fungsi dari ukuran masukan *n*.

G. Kompleksitas Waktu

Kompleksitas waktu dihitung berdasarkan berapa kali suatu operasi dilaksanakan dalam sebuah algoritma sebagai fungsi dengan ukuran masukan *n* [5]. Operasi yang diperhitungkan dalam kompleksitas waktu suatu algoritma hanya operasi-operasi khas yang mendasari algoritma tersebut. Adapun operasi-operasi tersebut adalah operasi aritmatika sederhana dan operasi perbadingan.

Kompleksitas waktu digolongkan menjadi 3 macam, yaitu $T_{max}(n)$ atau kompleksitas waktu untuk kasus terburuk, $T_{avg}(n)$ atau kompleksitas waktu untuk kasus rata-rata, dan $T_{min}(n)$ atau kompleksitas waktu untuk kasus terbaik. Nilai masing-masing besaran tersebut sangat beragam untuk tiap-tiap algoritma. Sebagai contoh, pada algoritma *sequential search*, $T_{min}(n) = 1$, $T_{max}(n) = n$, dan $T_{avg}(n) = 0.5(n+1)$.

G. Kompleksitas Waktu Asimptotik

Notasi untuk kompleksitas waktu asimptotik adalah *"O-*besar" (*Big-O*) [5]. Dalam hal ini *"T(n) = O(f(n))"* bermakna bahwa orde maksimum *T(n)* adalah *f(n)*, atau dengan ekspresi lain dapat dinyatakan sebagai $T(n) \leq C(f(n))$ untuk $n \geq n_o$. Dalam hal ini *f(n)* adalah batas lebih atas dari *T(n)* untuk *n* yang besar.

Urutan spektrum kompleksitas waktu algoritma dari kecil ke besar adalah sebagai berikut.

Tabel 2.1. Urutan Spektrum Kompleksitas Waktu Algoritma dari Kecil ke Besar

No.	Kelompok Algoritma	Nama
1.	*O(1)*	konstan
2.	*O(log n)*	logaritmik
3.	*O(n)*	lanjar
4.	*O(n log n)*	*n* log *n*
5.	$O(n^2)$	kuadratik
6.	$O(n^3)$	kubik
7.	$O(2^n)$	eksponensial
8.	*O(n!)*	faktorial

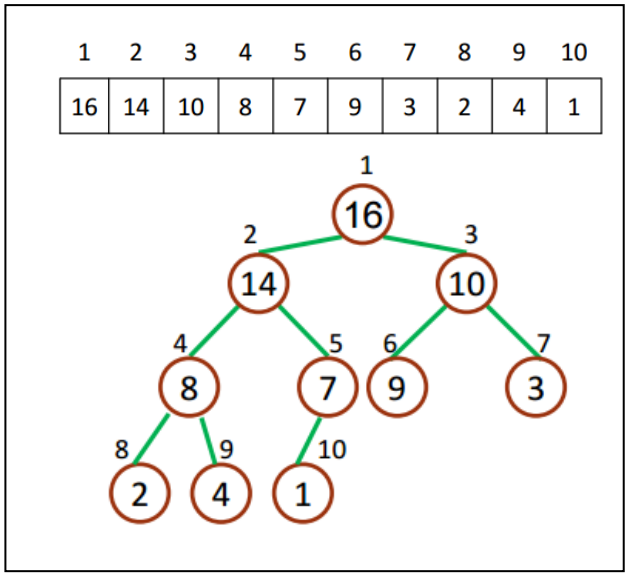
III. Penerapan Struktur Data Pohon pada Implementasi Algoritma Heapsort

A. Struktur Data Heap

Struktur data (biner) *heap* merupakan objek larik (*array*) yang dapat ditinjau sebagai pohon biner yang mendekati penuh [6]. Sehingga dalam konteks larik, struktur data *heap* merupakan susunan lain dari elemen-elemen array yang ditinjau sebagai pohon biner. Hal ini menyebabkan setiap simpul dari pohon tersebut berkorespondensi dengan tiap-tiap elemen array.

Struktur data *heap* dapat diklasifikasi lebih lanjut menjadi *max heap* dan *min heap*. *Max heap* adalah istilah untuk menyatakan bahwa struktur data *heap* yang tersusun memenuhi kondisi yaitu setiap nilai yang terkandung orang tua selalu lebih besar dibanding anak-anaknya. Berlaku sebaliknya untuk *min heap*, nilai yang terkandung pada orang tua selalu lebih kecil dibanding anak-anaknya. Penggolongan ini kemudian akan dimanfaatkan sebagai ketentuan untuk mengurutkan data secara menaik (*ascending*) maupun menurun (*descending*).

Berikut adalah ilustrasi korespondensi struktur data larik dan *heap* [7].



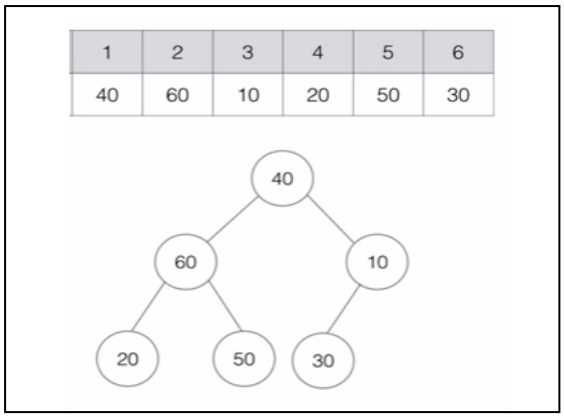
Gambar 3.1. Korespondensi Elemen pada Struktur Data Larik dan *Heap*

B. Alur Pengurutan

Terdapat dua golongan prosedur pengurutan menurut hasil akhirnya, yaitu menaik dan menurun. Hal ini mempengaruhi struktur data *heap* apa yang harus diterapkan untuk melakukan implementasi algoritma *Heapsort.* Terlepas dari itu, hal awal yang dilakukan adalah sama, yaitu membangun sebuah *heap* dari masukan sebuah larik.

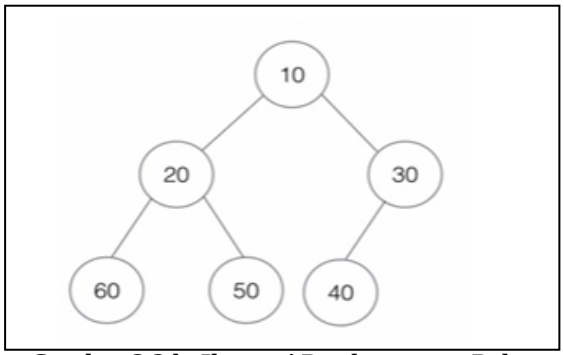
Berikut adalah alur pengurutan menggurnakan algoritma *Heapsort* pada kasus pengurutan menaik [8].

1. Pembangunan sebuah *heap* dari larik dapat dilakukan dengan memasukan elemen-elemen pada larik sesuai dengan urutannya ke dalam sebuah pohon biner (*heap*).



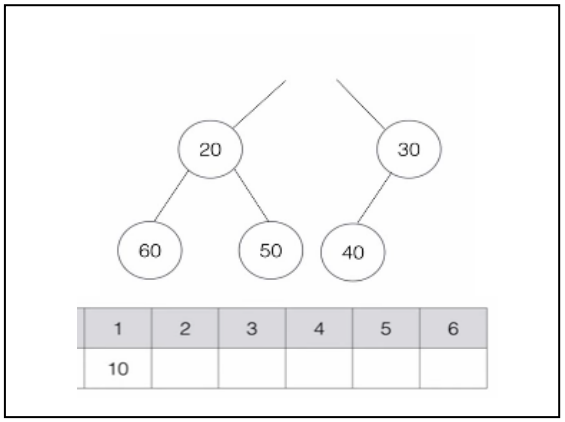
Gambar 3.2.a. Ilustrasi Pembangunan Pohon *Heap*

2. Setelah dilakukan pembangunan *heap*, hal yang harus dilakukan selanjutnya adalah melakukan pembangunan struktur data *min heap*.



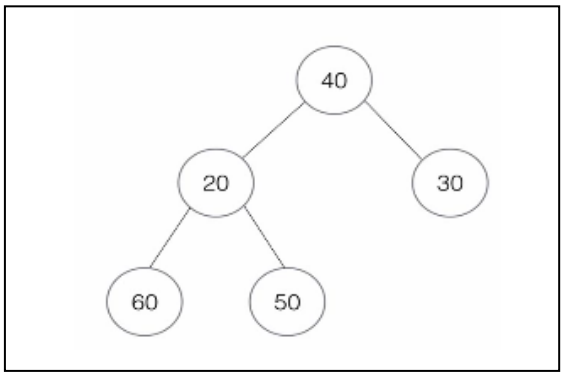
Gambar 3.2.b. Ilustrasi Pembangunan Pohon *Min Heap*

3. Setelah itu, dilakukan penghapusan akar pohon, dan memindahkannya ke indeks terkecil yang berlum terisi dari larik yang baru.



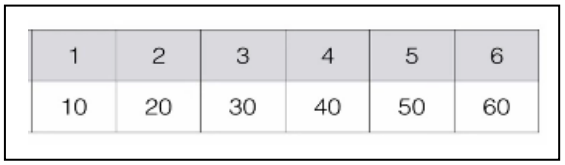
Gambar 3.2.c. Ilustrasi Penambahan Elemen Akar ke Larik Baru

4. Kemudian, dilakukan pemindahan daun paling kanan *heap* ke akarnya.



Gambar 3.2.d. Ilustrasi Pemindahan Elemen

5. Setelah itu, dilakuka pengulangan langkah 2 sampai 4 selama masih ada elemen pada pohon *heap*.
6. Setelah didapati bahwa semua elemen pada pohon sudah dipindahkan ke larik yang baru, maka susunan larik baru yang sudah terurut menaik akan terlihat seperti Gambar 3.2.e.



Gambar 3.2.e. Larik dengan Elemen yang Terurut Menaik

Alur tersebut juga dapat dijalankan untuk kasus pengurutan menurun hanya dengan mengganti langkah 2 menjadi pembangunan struktur data *max heap*.

B. Kode dalam Bahasa C

```
#include <stdio.h>

void display(int arr[], int size);
void heapSort(int arr[], int size);
void minHeap(int arr[], int size);

int main() {
```

```
	//unsorted elements
	//so, index 0 is set to -1
	int arr[] = {-1, 40,60,10,20,50,30};

	//size of the array
	int n = sizeof(arr)/sizeof(arr[0]);

	//output unsorted elements
	display(arr, n);

	//sort the elements
	heapSort(arr, n);

	//display sorted elements
	display(arr, n);

	return 0;
}

void display(int arr[], int size) {
	int i;
	for(i = 0; i < size; i++) {
		printf("%d ", arr[i]);
	}
	printf("\n");
}

void heapSort(int arr[], int size) {
	//variables
	int i, j, tmp, sorted[size];
	sorted[0] = -1;	//index 0 is not
used so set to -1

	//sort
	for(i = size , j = 1; i >= 1; i--,
j++) {
		minHeap(arr, i);
		sorted[j] = arr[1];
		arr[1] = arr[i - 1];
	//put last element to root node
	}

	//set the result
	for(i = 0; i < size; i++) {
		arr[i] = sorted[i];
	}
}

void minHeap(int arr[], int size) {
	int i, left, right, tmp, val;

	for(i = size / 2; i >= 1; i--) {

		//taking 1 as the start index
		//if parent node = i
		//so left child = 2i
		//and right child = 2i+1
		tmp = i;
		left = (2 * i);
		right = (2 * i) + 1;

		if(left < size && arr[left] <
arr[tmp]) {
			tmp = left;
		}

		if(right < size && arr[right]
< arr[tmp]) {
			tmp = right;
		}

		if(tmp != i) {
			val = arr[i];
			arr[i] = arr[tmp];
			arr[tmp] = val;
			minHeap(arr, size);
		}
	}
}
```

IV. Analisis Kompleksitas Waktu Algoritma Heapsort

Berikut merupakan tinjauan kompleksitas waktu pada tiap-tiap operasi yang dijalankan pada algoritma *Heapsort* [9].

1. Kompleksitas untuk memasukan satuu elemen ke *heap* kosong adalah *O(1)*. Kompleksitas untuk memasukan satu elemen ke *heap* dengan *n* simpul adalah *O(log n)*. Dengan demikian kompleksitas untuk memasukan *n* elemen ke dalam *heap* dengan jumlah simpul sebanyak *n* adalah *O(n log n)*.
2. Dalam kontesks pohon *max heap*, kompleksitas untuk menghapus elemen terbesar adalah *O(1)*. Kompleksitas untuk menghapus elemen terkecil pada pohon *max heap* adalah *O(log n)*. Dengan demikian, kompleksitas untuk menghapus *n* elemen dalam pohon *heap* adalah *O(n log n)*.

Hal ini menyebabkan kompleksitas untuk membangun sebuah pohon *heap* adalah *O(n log n)*. Dalam *Heapsort* diperlukan operasi penyusunan elemen (dalam hal ini memasukan elemen ke dalam *heap*) dan penghapusan elemen terbesar sebanyak jumlah data *n*. Masing-masing kompleksitas waktu asimpotik untuk prosedur tersebut adalah *O(n log n)*. Hal ini mengimplikasi bahwa kompleksitas waktu asimptotik untuk *Heapsort* adalah *O(n log n)*.

V. Kesimpulan

Algoritma *Heapsort* cocok untuk diterapkan pada pengolahan data dengan jumlah yang cukup besar karena kompleksitas waktunya yang sama untuk semua kasus yaitu *O(n log n*). Implementasi algoritma *Heapsort* harus dihindari untuk pengurutan yang memperhatikan kestabilan hasil pengurutannya. Kestabilan dalam hal ini adalah sifat memprioritaskan urutan pada keadaan awal sebelum algoritma pengurutan dijalankan.

VII. Ucapan Terima Kasih

Syukur pertama kali saya ucapkan pada Tuhan Yang Maha Esa karena atas limpahan rahmatnya saya dapat menyelesaikan makalah ini. Tidak lupa, tanpa menguranngi rasa hormat saya, terima kasih saya sampaikan pada dosen matakuliah IF2120 Matematika Diskrit, Dr. Rinaldi Munir dan Dra. Harlili S., M. Sc. atas bimbingannya. Terakhir, saya sampaikan terima kasih pada keluarga dan rekan-rekan saya yang telah mendukung terselesaikannya makalah ini.

REFERENSI

[1] Kemdikbud. *Kamus Besar Bahasa Indonesia*. Edisi V. Jakarta: Balaipustaka, 2016.

[2] Hosam M. Mahmoud, *Sorting: A Distribution Theory*. New York: John Wiley & Sons, 2000.

[3] John Morris, "*Data Structures and Algorithms: Comparing Quick and Heap Sorts"*. https://www.cs.auckland.ac.nz/software/-AlgAnim/qsort3.html. Diakses pada 9 Desember 2016 pukul 02.51 WIB.

[4] Kenneth H. Rosen, *Discrete Mathematics and Its Applications, Seventh Edition.* New York: McGraw-Hill, 2012, ch. 11.

[5] Rinaldi Munir, *Matematika Diskrit, Edisi ke-3.* Informatika Bandung: Bandung, 2010.

[6] Thomas H. Cormen et al., *Introduction to Algorithm, Third Edition*. Cambridge: MIT Press, 2012, ch. 6.

[7] Ching-Chi Lin, "*Algorithms Chapter 6 Heapsort"*. http://ind.ntou.edu.tw/~litsnow/al98/pdf/Algorithm-Ch6-Heaps-ort.pdf. Diakses pada 9 Desember 2016 pukul 14.42

[8] Dyclassroom, "*Heap Sort: Sorting Algorithm"*. https://www.dyclas-sroom.com/sorting-algorithm/heap-sort. Diakses pada 9 Desember 2016 pukul 15.25.

[9] Akshay Upadhyay, *"Heap Sort Complexity"*. http://www.cprogra-mto.com/heap-sort-complexity/. Diakses pada 9 Desember 2016 pukul 15.48.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 9 Desember 2016

Paskahlis Anjas Prabowo 13515108