RANCANG APLIKASI UNTUK PENGURUTAN DATA DENGAN METODE HEAP SORT

Disusun Sebagai Salah Satu Syarat Untuk Menempuh Ujian Akhir
Memperoleh Gelar Sarjana Komputer Pada Fakultas Sains dan Teknologi
Universitas Pembangunan Panca Budi
Medan

SKRIPSI

OLEH :

NAMA	**: RINALDI**
N.P.M	**: 1514370085**
PROGRAM STUDI	**: SISTEM KOMPUTER**
KONSENTRASI	**: KEAMANAN JARINGAN KOMPUTER**

FAKULTAS SAINS DAN TEKNOLOGI
UNIVERSITAS PEMBANGUNAN PANCA BUDI
MEDAN
2019

LEMBAR PENGESAHAN

RANCANG APLIKASI UNTUK PENGURUTAN DATA DENGAN METODE HEAP SORT

Disusun Oleh :

NAMA	**: RINALDI**
N.P.M	**: 1514370085**
PROGRAM STUDI	**: SISTEM KOMPUTER**

Skripsi telah disetujui oleh Dewan Pembimbing Skripsi pada tanggal : 06 – 05 - 2019

Dosen Pembimbing I	Dosen Pembimbing II
Darmeli nasution s.kom m.kom	**Nadya andika putri s.kom m.kom**

Mengetahui,

Dekan Fakultas Sains dan Teknologi	Ketua Program Studi
Sri Shindi Indira, S.T.,M.Sc	**Eko Hariyanto, S.Kom.,M.Kom**

SURAT PERNYATAAN

Saya yang bertanda tangan di bawah ini :

Nama	: Rinaldi
NPM	: 1514370085
Prodi	: Sistem Komputer
Konsentrasi	: Keamanan Jaringan Komputer
Judul Skripsi	: Rancang Aplikasi Untuk Pengurutan Data Dengan Metode Heap Sort

Denganinimenyatakanbahwa :

1. Tugas Akhir/Skripsi saya bukan hasil Plagiat
2. Saya tidak akan menuntut perbaikan nilai indeks Prestasi Kumulatif (IPK) setelah ujian Sidang Meja Hijau
3. Skripsi saya dapat dipublikasikan oleh pihak lembaga, dan saya tidak akan menuntut akibat publikasi tersebut

Demikian pernyataan ini saya perbuat dengan sebenar-benarnya, terima kasih

Medan,

Yang membuat pernyataan

Rinaldi
1514370085

PERNYATAAN ORISINALITAS

Dengan ini saya menyatakan bahwa dalam skripsi ini tidak terdapat karya yang di ajukan untuk memperoleh gelar kesarjanaan di dalam perguruan tinggi, dan sepanjang pengetahuan saya juga tidak terdapat karya atau pendapat yang pernah ditulis atau diterbitkan oleh orang lain, kecuali yang secara tertulis di acu dalam skripsi ini dan disebutkan dalam daftar pustaka.

Medan, 17 Oktober 2019

Rinaldi

1514370085

UNIVERSITAS PEMBANGUNAN PANCA BUDI
FAKULTAS SAINS & TEKNOLOGI

Jl. Jend. Gatot Subroto Km 4,5 Medan Fax. 061-8458077 PO.BOX : 1099 MEDAN

PROGRAM STUDI TEKNIK ELEKTRO	(TERAKREDITASI)
PROGRAM STUDI TEKNIK ARSITEKTUR	(TERAKREDITASI)
PROGRAM STUDI SISTEM KOMPUTER	(TERAKREDITASI)
PROGRAM STUDI TEKNIK KOMPUTER	(TERAKREDITASI)
PROGRAM STUDI AGROTEKNOLOGI	(TERAKREDITASI)
PROGRAM STUDI PETERNAKAN	(TERAKREDITASI)

PERMOHONAN JUDUL TESIS / SKRIPSI / TUGAS AKHIR*

Saya yang bertanda tangan di bawah ini :

Nama Lengkap	: RINALDI
Tempat/Tgl. Lahir	: MEDAN / 08 September 1996
Nomor Pokok Mahasiswa	: 1514370085
Program Studi	: Sistem Komputer
Konsentrasi	: Keamanan Jaringan Komputer
Jumlah Kredit yang telah dicapai	: 137 SKS, IPK 3.27
Nomor Hp	: 081396620100
Dengan ini mengajukan judul sesuai bidang ilmu sebagai berikut	:

No.	Judul
1.	RANCANG APLIKASI UNTUK PENGURUTAN DATA DENGAN METODE HEAP SORT

Catatan : Diisi Oleh Dosen Jika Ada Perubahan Judul

*Coret Yang Tidak Perlu

Medan, 05 April 2019

Pemohon,

(Rinaldi)

Rektor I,

(Ir. Bhakti Alamsyah, M.T., Ph.D.)

Tanggal : .....

Disahkan oleh :

Dekan

(Sri Shindi Indira, S.T.,M.Sc.)

Tanggal : .....

Disetujui oleh :

Dosen Pembimbing I :

(Darmeli Nasution, S.Kom.,M.Kom)

Tanggal : .....

Disetujui oleh:

Ka. Prodi Sistem Komputer

(MUHAMMAD IQBAL, S.Kom.,M.Kom.)

Tanggal : .....

Disetujui oleh:

Dosen Pembimbing II:

(NADYA ANDHIKA PUTRI, S.KOM., M.KOM)

UNIVERSITAS PEMBANGUNAN PANCA BUDI
FAKULTAS SAINS DAN TEKNOLOGI

Jl. Jend. Gatot Subroto Km 4,5 ☎ 06150200508 - Fax : 061-8455571-PO BOX 1099 Medan
Email : fastek@pancabudi.ac.id website : www.pancabudi.ac.id

PERMOHONAN MENGAJUKAN JUDUL SKRIPSI

a yang bertanda tangan di bawah ini :

m a : RINALDI
.M : 1514370085
nsentrasi : Keamanan jaringan komputer

engan ini mengajukan judul :

Rancang Aplikasi untuk Pengurutan data dengan metode Heap Sort ☑

Perancang Aplikasi objek wisata dimedan berbasis multimedia dan android ☐

Rancang Sistem Aplikasi Form Pendaftaran murid baru pada Sekolah SMK PAB ☐

Keputusan Pembimbing I : ☐ Terima ☐ Tolak

Keputusan Pembimbing II : ☐ Terima ☐ Tolak

Menyetujui,
Medan,

Dosen Pembimbing I

(Dr. Dedi Hartama, M.Kom)

Dosen Pembimbing II

(Hadiyan)

Menyetujui,
Ketua Program Studi

TANDA BEBAS PUSTAKA
No. 787 / Perp / Bp / 2019
Dinyatakan tidak ada sangkut paut dengan UPT. Perpustakaan
4 SEP 2019
Kepala UPT. Perpustakaan
Sugiarto, S.Sos., S.Pd.I

FM-BPAA-2012-041

Hal : Permohonan Meja Hijau

Medan, 24 September 2019
Kepada Yth : Bapak/Ibu Dekan
Fakultas SAINS & TEKNOLOGI
UNPAB Medan
Di -
Tempat

Telah Diperiksa oleh LPMU
dengan Plagiarisme 42 %
Medan 23 SEPTEMBER 2019
Kepala UPT
Cahyo Pramono, SE,MM

Dengan hormat, saya yang bertanda tangan di bawah ini :

Nama	: RINALDI
Tempat/Tgl. Lahir	: HELVETIA / 08 SEPTEMBER 1996
Nama Orang Tua	: sunarno
N. P. M	: 1514370085
Fakultas	: SAINS & TEKNOLOGI
Program Studi	: Sistem Komputer
No. HP	: 081396620100
Alamat	: DUSUN X JL. UTAMA GG. DARMA

Datang bermohon kepada Bapak/Ibu untuk dapat diterima mengikuti Ujian Meja Hijau dengan judul **RANCANG APLIKASI UNTUK PENGURUTAN DATA DENGAN METODE HEAP SORT**, Selanjutnya saya menyatakan :

1. Melampirkan KKM yang telah disahkan oleh Ka. Prodi dan Dekan
2. Tidak akan menuntut ujian perbaikan nilai mata kuliah untuk perbaikan indek prestasi (IP), dan mohon diterbitkan ijazahnya setelah lulus ujian meja hijau.
3. Telah tercap keterangan bebas pustaka
4. Terlampir surat keterangan bebas laboratorium
5. Terlampir pas photo untuk ijazah ukuran 4x6 = 5 lembar dan 3x4 = 5 lembar Hitam Putih
6. Terlampir foto copy STTB SLTA dilegalisir 1 (satu) lembar dan bagi mahasiswa yang lanjutan D3 ke S1 lampirkan ijazah dan transkipnya sebanyak 1 lembar.
7. Terlampir pelunasan kwintasi pembayaran uang kuliah berjalan dan wisuda sebanyak 1 lembar
8. Skripsi sudah dijilid lux 2 examplar (1 untuk perpustakaan, 1 untuk mahasiswa) dan jilid kertas jeruk 5 examplar untuk penguji (bentuk dan warna penjilidan diserahkan berdasarkan ketentuan fakultas yang berlaku) dan lembar persetujuan sudah di tandatangani dosen pembimbing, prodi dan dekan
9. Soft Copy Skripsi disimpan di CD sebanyak 2 disc (Sesuai dengan Judul Skripsinya)
10. Terlampir surat keterangan BKKOL (pada saat pengambilan ijazah)
11. Setelah menyelesaikan persyaratan point-point diatas berkas di masukan kedalam MAP
12. Bersedia melunaskan biaya-biaya uang dibebankan untuk memproses pelaksanaan ujian dimaksud, dengan perincian sbb :

1.	[102] Ujian Meja Hijau	: Rp.	100.000
2.	[170] Administrasi Wisuda	: Rp.	1,500,000
3.	[202] Bebas Pustaka	: Rp.	100,000
4.	[221] Bebas LAB	: Rp.	5,000
Total Biaya		: Rp.	~~1.605.000~~ 1.705.000

20/09 19

5 · Uk 50% (1 tahun) Rp. 3.100.000

Rp. 4.805.000

Ukuran Toga : L

30/9-19
Diketahui/Disetujui oleh :

Sri Shindi Indira, S.T.,M.Sc.
Dekan Fakultas SAINS & TEKNOLOGI

Hormat saya

RINALDI
1514370085

Catatan :

- 1.Surat permohonan ini sah dan berlaku bila ;
  - a. Telah dicap Bukti Pelunasan dari UPT Perpustakaan UNPAB Medan.
  - b. Melampirkan Bukti Pembayaran Uang Kuliah aktif semester berjalan
- 2.Dibuat Rangkap 3 (tiga), untuk - Fakultas - untuk BPAA (asli) - Mhs.ybs.

UKM CENTER
30/09/19

Telah di terima berkas persyaratan dapat di proses
Medan, 30/09/2019
Ka. BPAA
an.
TEGUH WAHYONO, SE., MM.

YAYASAN PROF. DR. H. KADIRUN YAHYA
UNIVERSITAS PEMBANGUNAN PANCA BUDI
LABORATORIUM KOMPUTER
Jl. Jend. Gatot Subroto Km 4,5 Sei Sikambing Telp. 061-8455571
Medan - 20122

KARTU BEBAS PRAKTIKUM

Yang bertanda tangan dibawah ini Ka. Laboratorium Komputer dengan ini menerangkan bahwa :

Nama	:	RINALDI
N.P.M.	:	1514370085
Tingkat/Semester	:	*Akhir*
Fakultas	:	SAINS & TEKNOLOGI
Jurusan/Prodi	:	Sistem Komputer

Benar dan telah menyelesaikan urusan administrasi di Laboratorium Komputer Universitas Pembangunan Panca Budi Medan.

Medan, 20 Juli 2019
Ka. Laboratorium

Fachrid Wadly, S. Kom

Plagiarism Detector v. 1092 - Originality Report:

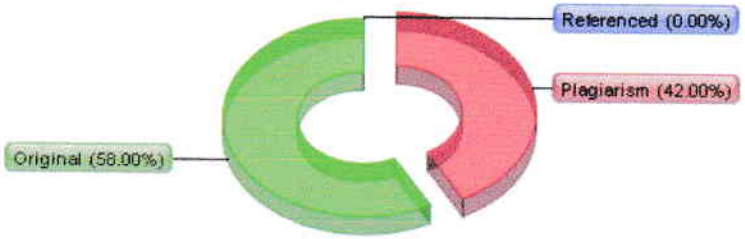
Analyzed document: 16/07/2019 15:22:24

"RINALDI_1514370085_SISTEM KOMPUTER.docx"

Licensed to: **Universitas Pembangunan Panca Budi_License4**

UNIVERSITAS PEMBANGUNAN PANCA BUDI
Berilmu, Berimtu, Berkarya

Relation chart:



Distribution graph:

Comparison Preset: Rewrite. Detected language: Indonesian

Top sources of plagiarism:

% 26	wrds: 6878	http://www.officeopenxml.com/WPsampleDoc.php
% 21	wrds: 3334	https://ejournal.unib.ac.id/index.php/pseudocode/article/download/822/716
% 17	wrds: 4383	http://www.ericwhite.com/blog/search-and-replace-text-in-an-open-xml-wordprocessingml-docu...

[Show other Sources:]

Processed resources details:

137 - Ok / 17 - Failed

[Show other Sources:]

Important notes:

Wikipedia:	Google Books:	Ghostwriting services:	Anti-cheating:
Wiki Detected!	[not detected]	[not detected]	[not detected]

Excluded Urls:

UNIVERSITAS PEMBANGUNAN PANCA BUDI
FAKULTAS SAINS & TEKNOLOGI
Jl. Jend. Gatot Subroto Km. 4,5 Telp (061) 8455571
website : www.pancabudi.ac.id email: unpab@pancabudi.ac.id
Medan - Indonesia

sitas : Universitas Pembangunan Panca Budi
es : SAINS & TEKNOLOGI
Pembimbing I : Darmeli Nasution, S.Kom., M.Kom
Pembimbing II : Nadya Anohika Putri S.Kom M.Kom
Mahasiswa : RINALDI
n/Program Studi : Sistem Komputer
Pokok Mahasiswa : 1514370085
g Pendidikan : Strata Satu (S1)
Tugas Akhir/Skripsi : Rancang Aplikasi untuk Pengurutan data dengan metode Heap Sort

NGGAL	PEMBAHASAN MATERI	PARAF	KETERANGAN
3-2019	Perbaiki penulisan BAB I		
3-2019	Acc proposal, lanjut Sempro		
06-19	Acc Bab I, Perbaiki Bab II, III, IV dan V		
07-19	Acc Bab I - V Acc Seminar Hasil.		
-19	Tambahkan flowchart dan contoh analisa dengan metode.		
8-19	Acc Bab III, IV, V		
9-19	Acc Sidang Meja Hijau		

Medan, 01 Maret 2019
Diketahui/Disetujui oleh :
Dekan,

Sri Shindi Indira, S.T.,M.Sc.

ang tidak perlu

UNIVERSITAS PEMBANGUNAN PANCA BUDI
FAKULTAS SAINS & TEKNOLOGI
Jl. Jend. Gatot Subroto Km. 4,5 Telp (061) 8455571
website : www.pancabudi.ac.id email: unpab@pancabudi.ac.id
Medan - Indonesia

as : Universitas Pembangunan Panca Budi
: SAINS & TEKNOLOGI
embimbing I : DARMELI NASUTION, S.Kom., M.Kom
embimbing II : NADYA ANDHIKA PUTRI, S.Kom., M.Kom
ahasiswa : RINALDI
Program Studi : Sistem Komputer
okok Mahasiswa : 1514370085
Pendidikan : STRATA SATU (S1)
gas Akhir/Skripsi : RANCANG APLIKASI UNTUK PENGURUTAN DATA DENGAN METODE HEAP SORT

GGAL	PEMBAHASAN MATERI	PARAF	KETERANGAN
-19	- Asistensi Bab I dan Daftar Pustaka - Perbaikan Latar Belakang		
-19	Acc Seminar Proposal		
-19	- Asistensi Bab II, Perbaikan Landasan teori, Tambahkan teori tentang Desain, Minyak Istilah ang		
-19	- Asistensi Bab III, Cek Analisa - Tambahkan Pemesan, dan Flowchart		
-19	- Asistensi Bab IV - Perbaikan Implementasi, dan Screenshot Program		
-19	- Asistensi Bab V, Penutup dan		
-19	- Acc Seminar		
-2019	- Acc Sidang		

Medan, 01 Maret 2019
Diketahui/Disetujui oleh :
Dekan,

UNIVERSITAS PEMBANGUNAN PANCA BUDI * UNPAB INDONESIA * FAKULTAS SAINS & TEKNOLOGI *

Sri Shindi Indira, S.T.,M.Sc.

ang tidak perlu

ABSTRAK

RINALDI

Rancang aplikasi untuk pengurutan data dengan metode heap sort

2019

Struktur data dari algoritma Heap Sort adalah sebuah pohon biner sempurna yang memenuhi properti heap. Node akar (root node) memiliki data terbesar atau terkecil yang terdapat pada pohon. Demikian juga pada subtree-nya, dimana node induk (parent) memiliki data yang paling besar atau paling kecil dibandingkan dengan data pada kedua anaknya (child node sebelah kiri atau sebelah kanan). Struktur data heap adalah sebuah objek array yang dapat divisualisasikan dengan sebuah complete binary tree. Hubungan antara elemen dari array dan node pada pohon merupakan hubungan korespondensi satu satu. Pohon diisi secara penuh pada semua level, kecuali kemungkinan terkecil, dimana diisi dari kiri sampai ke sebuah titik. Semua node dari heap juga memenuhi relasi bahwa nilai kunci pada setiap node minimal sama besar dengan nilai dari node anaknya. Setelah menyelesaikan perangkat lunak bantu pemahaman Heap Sort, perangkat lunak menjelaskan algoritma pengurutan Heap Sort dan gambar keadaan pohon biner secara bertahap serta menampilkan Form Teori, sehingga dapat membantu pemahaman mengenai pengurutan dengan metode Heap Sort. Algoritma pengurutan Heap Sort merupakan salah satu metode pengurutan tercepat setelah Merge Sort dan Quick Sort dengan kompleksitas O(n log n). Kompleksitas prosedur Build-Heap adalah O(n), kompleksitas prosedur Heapify adalah O(log n), sehingga kompleksitas algoritma Heap Sort adalah O(n log n).

Kata Kunci : Heap Sort, Merge Sort, Quick Sort, kompleksitas O(n log n).

DAFTAR ISI

Halaman

BAB I
PENDAHULUAN

1.1 Latar Belakang

Heap sort adalah algoritma pengurutan data berdasarkan perbandinngan, dan termasukgolongan selection sort. Walaupun lebih lambat dari padaquick sort pada kebanyakan mesi, tetapi heap sort mempunyai keungulan yaitu kompleksitas algoritma pada kasus terburuk adalah n log n. Algoritma pengurutan ini menggunakan isi suatu larik masukan dengan memandang larik masukan sebagai suatu comple Binary Tree (CBT) ini dapat dikonversi menjadi suaatu heap sort. Setelah itu complete Binery Tree (CBT) diubah menjadi suatu priority queuue

Algoritma pengurutan heap dimuali dari membangun sebuah heap dari kumpulan data yang ining diurutkan, dan kemudian menghapus data yang mempunyai nilai yang telah diurut, 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,

Struktur data *heap* adalah sebuah objek *array* yang dapat divisualisasikan dengan sebuah *complete binary tree*. Hubungan antara elemen dari *array* dan *node* pada pohon merupakan hubungan korespondensi satu satu. Pohon diisi secara penuh pada semua level,

kecuali kemungkinan terkecil, dimana diisi dari kiri sampai ke sebuah titik. Semua *node* dari *heap* juga memenuhi relasi bahwa nilai kunci pada setiap *node* minimal sama besar dengan nilai dari *node* anaknya.

Struktur data dari algoritma Heap Sort adalah sebuah pohon biner sempurna yang memenuhi properti *heap*. *Node* akar (*root node*) memiliki data terbesar atau terkecil yang terdapat pada pohon. Demikian juga pada *subtree*-nya, dimana *node* induk (*parent*) memiliki data yang paling besar atau paling kecil dibandingkan dengan data pada kedua anaknya (*child node* sebelah kiri atau sebelah kanan).

Berdasarkan uraian di atas, maka dipilih tugas akhir dengan judul "**PERANCANGAN APLIKASI UNTUK PENGURUTAN DATA DENGAN METODE HEAP SORT**". Perangkat lunak yang dirancang akan mampu untuk menjelaskan prosedur kerja dari algoritma Heap Sort.

1.2 Rumusan Masalah

Berdasarkan latar belakang pemilihan judul, maka yang menjadi permasalahan adalah menampilkan prosedur kerja dari algoritma Heap Sort.

1.3 Pembatasan Masalah

Ruang lingkup permasalahan dalam merancang perangkat lunak ini dibatasi sebagai berikut :

1. Angka yang di-*input* bertipe data bilangan *integer* positif dengan batasan maksimal 3 digit.

2. Jumlah data yang di-*input* dibatasi maksimal 50 buah.
3. Perangkat lunak memiliki fasilitas *pause* (menghentikan sementara) untuk proses kerja algoritma pengurutan Heap Sort dan *resume* untuk melanjutkan proses kerja algoritma pengurutan Heap Sort.
4. Perangkat lunak akan menggambarkan visualisasi heap sebagai pohon biner lengkap (*complete binary tree*).

1.4 Tujuan Penelitian

Tujuan penyusunan tugas akhir (skripsi) ini adalah untuk merancang suatu perangkat lunak yang mampu untuk menjelaskan dan menampilkan prosedur kerja dari algoritma Heap Sort.

1.5 Manfaat Penelitian

1. Untuk membantu pemahaman mengenai algoritma Heap Sort.
2. Perangkat lunak juga dapat digunakan sebagai fasilitas pendukung dalam proses belajar mengajar, terutama pada mata kuliah Struktur Data.

BAB II
LANDASAN TEORI

2.1 Pengertian Heap Sort

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 :

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

Jenis-jenis Heap :

1. Binary heap : Binary heap adalah heap yang dibuat dengan menggunakan pohon biner.
2. Binomial heap : Binomial heap adalah heap yang dibuat dengan menggunakan pohon binomial. Pohon binomial bila didefinisikan secara rekursif adalah:
3. 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 pohonpohon binomial dengan tinggi k-1,k- 2,…,2,1,0.
4. 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

Metode heap sort adalah metode dari pengembangan tree. Metode sorting ini mengurutkan data berdasarkan perbandingan, dan merupakan salah satu algoritma pengurutan tercepat karena mampu mengurutkan data yang sangat banyak dengan waktu yang cepat. Algoritma Heap Sort memiliki kompleksitas yang besar dalam pembuatan kodenya. Programmer sering menggunakan ini ketika berhadapan dengan array yang jumlahnya sangat besar. Algoritma ini bekerja dengan menentukan elemen terbesar (atau terkecil) dari sebuah daftar elemen, dan diletakkan pada akhir (atau awal) dari daftar tersebut. Heap sort menyelesaikan sebuah pengurutan ini

dimulai dengan membangun sebuah array heap dengan membangun tumpukan dari kumpulan data, lalu memindahkan data terbesar ke bagian belakang dari sebuah tabel hasil. Setelah itu, array heap dibangun kembali, kemudian mengambil elemen terbesar untuk diletakkan di sebelah item yang telah dipindahkan tadi. Hal ini diulang sampai array heap habis.

Jadi secara umum, algoritma ini memerlukan dua buah tabel; satu tabel untuk menyimpan heap, dan satu tabel lainnya untuk menyimpan hasil. Walaupun lebih lambat dari Merge Sort atau Quick Sort, algoritma ini cocok untuk digunakan pada data yang berukuran besar.

2.2 Aturan , Kelebihan dan Kelemahan Dalam Heap Sort

Beberapa aturan dalam Heap Sort sebagai berikut :

- Untuk mengisikan heap dimulai dari level 1 sampai ke level dibawahnya, bila dalam level yang sama semua kunci heap belum terisi maka tidak boleh mengisi dibawahnya.Heap dalam kondisi terurut apabila left child <> parent.Penambahan kunci diletakkan pada posisi terakhir dari level dan disebelah kanan child yg terakhir, kemudian diurutkan dengan cara upheap.Bila menghapus heap dengan mengambil kunci pada parent di level 1 kemudian digantikan posisi kunci terakhir, selanjutnya disort kembali metode downheap.Dalam langkah-langkahnya heap sort terbagi menjadi 2 langkah yaitu insert_heap dan build_heap.

2.3 Struktur Data

Struktur berarti susunan / jenjang, dan data berarti sesuatu simbol / huruf / lambang angka yang menyatakan sesuatu. Struktur data berarti susunan dari simbol / huruf / lambang angka untuk menyatakan sesuatu hal. Sebagai contoh, struktur program Pascal dapat didefenisikan seperti berikut,

(i) Deklarasi Nama Fungsi / Prosedur.

(ii) Deklarasi Tipe Data.

(iii) Deklarasi Konstanta (untuk variabel bernilai nilai statis).

(iv) Deklarasi Variabel.

(v) Deklarasi Label.

(vi) Badan Program (*Begin … End*.)

Gabungan dari algoritma dan struktur data akan membentuk suatu program. Adapun manfaat dari struktur data adalah sebagai berikut,

a. Mengefisiensikan program.
b. Program yang dibuat dengan menerapkan konsep – konsep yang berlaku pada struktur data akan lebih efisien dibandingkan dengan program yang dibuat dengan mengabaikan konsep struktur data.
c. Modifikasi
d. Sesuatu program harus dapat dimodifikasi apabila diperlukan, hal ini dapat dilakukan jika fasilitas yang diperlukan dibuat (disertakan) walaupun pada tahap awal belum dipakai.
e. Memilih metode yang tepat

1. Pemasukan data tidak melalui *keyboard* lagi, melainkan melalui *barcode*.
2. Membuat pemberitahuan pada kasir – kasir.

2.4 Array

Array (larik) adalah suatu tipe data terstruktur yang dapat menampung (berisikan) suatu data yang sejenis. Komponen – komponen dari *Array* antara lain,

a. Nama *Array*
b. Nilai *Array*
c. Index *Array*
d. Jenis (tipe) *Array*

Deklarasi *Array* dapat dibuat pada,

a. Bagian Tipe

   Contoh,

   Type A = Array [1..5] of Char;

b. Bagian Variabel

   Contoh,

   Var A = Array [1..5] of Char;

Array terdiri dari beberapa jenis antara lain,

a. Array 1 dimensi

   Contoh misalkan diketahui sebuah *Array* A dengan jenis data *Char*,

A =

A	B	C	D	E	F	G	H
1	2	3	4	5	6	7	8

A[1] = ‘A’

A[2] = ‘B’

A[3] = ‘C’

A[4] = ‘D’

b. *Array* Multi dimensi

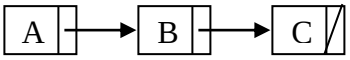
Contoh, misalkan diketahui sebuah *Array* dua dimensi B dengan jenis data *Integer*, B[1,1] = 1 B[1,2] = 2 B[2,1] = 3

Proses - proses yang mungkin dilakukan pada *Array* adalah,

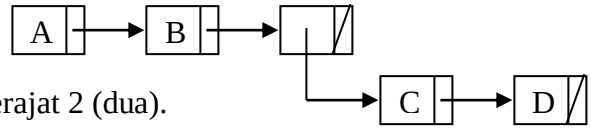
a. Proses memasukkan data
b. Proses mencari data
c. Proses menghapus data
d. Proses mencetak data
e. Proses menyisip data
f. Proses mencari posisi
g. Proses mengurutkan data
h. Dan sebagainya

2.5 Linked List

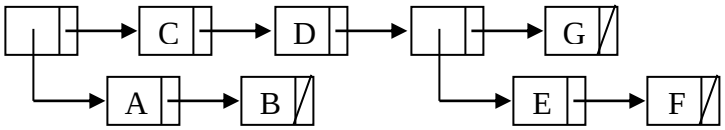
Linked List (senarai) adalah sekumpulan data yang linier satu dengan yang lain. Sekumpulan berarti identik dengan himpunan sebab himpunan adalah kumpulan dari beberapa data yang linier (setara). Bila *Linked List* ekivalen dengan himpunan maka,

1. Himpunan kosong ( ) →*Linked List* kosong 
2. Himpunan tidak kosong (A, B, C) →*Linked List* 

    a. Berderajat 1 (satu).
    b. Elemennya 3 (tiga) buah yaitu A, B dan C.
3. Bila himpunannya (A, B, (C, D)), maka

    Linked List 

    a. Berderajat 2 (dua).
        - Elemennya 4 (empat) buah yaitu A, B, C dan D.
4. Bila himpunannya ((A, B), C, D, (E, F), G), maka

    Linked List 

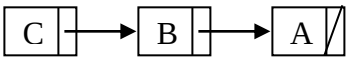
    a. Berderajat 2 (dua).
    b. Elemennya 7 (tujuh) buah yaitu A, B, C, D, E, F dan G.

Operasi – operasi yang dapat dilakukan pada *Linked List* antara lain,

a. Operasi penambahan data
b. Operasi pencarian data
c. Operasi penghapusan data

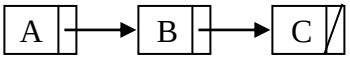
a. Depan *List*

Contoh, proses penambahan data A, B dan C secara berturut – turut pada *List*



b. Belakang *List*

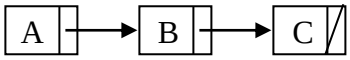
Contoh, proses penambahan data A, B dan C secara berturut – turut pada *List*



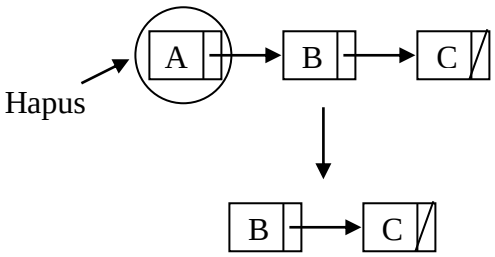
Proses penghapusan suatu simpul pada *Linked List* harus mempergunakan *pointer* bantu. *Pointer* bantu tersebut harus menempel pada simpul yang akan dihapus, baru dilakukan operasi penghapusan. Proses penghapusan dapat dilakukan pada,

a. Simpul awal

Contoh, misalkan diketahui sebuah *Linked List* dengan elemen A, B dan C.

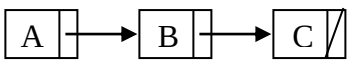


Proses penghapusan simpul awal A adalah sebagai berikut,



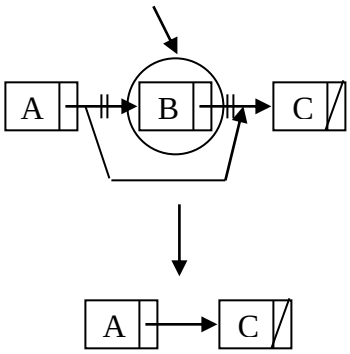
b. Simpul tengah

Contoh, misalkan diketahui sebuah *Linked List* dengan elemen A, B dan C.



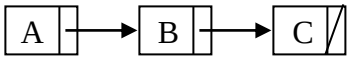
Proses penghapusan simpul tengah B adalah sebagai berikut,

Hapus

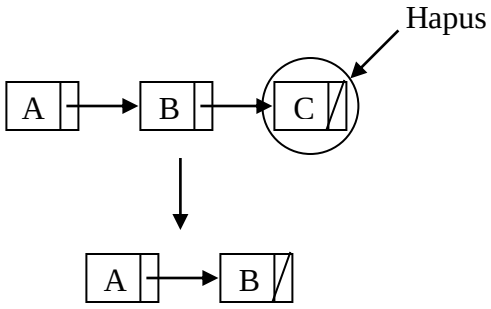


c. Simpul akhir

Contoh, misalkan diketahui sebuah *Linked List* dengan elemen A, B dan C.



Proses penghapusan simpul akhir C adalah sebagai berikut,



2.5 Double Linked List

Dalam implementasi *List* dengan *pointer* ganda (*double linked list*), *list* direpresentasikan sebagai untaian simpul yang berisi elemen (data), *pointer* mundur ke simpul sebelumnya dan *pointer* maju ke simpul berikutnya dalam untaian. Elemen pertama dari *List* L ditunjuk dengan *pointer* khusus sebagai kepala dengan nama L.Awal dan elemen terakhir sebagai ekor ditunjuk oleh L.Akhir.

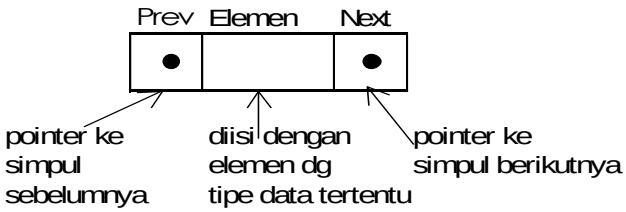
Pembentukan kepala *List* sebagai satu unit L dengan *field* Awal, Akhir dan Count memberikan fleksibilitas dalam pengiriman argumen (*actual parameter*) dan

menggunakan nama L yang tipenya adalah List, sesuai dengan definisi di bawah ini, tanpa harus mendeklarasikan Awal dan Akhir sebagai pointer dan Count sebagai byte. Hal ini merupakan salah satu konsep pengkapsulan (*encapsultion*).

Untuk implementasi tersebut dibuat deklarasi seperti terlihat berikut ini:

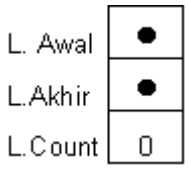
```
Type
        Pointer = ^Simpul;
        Simpul = Record
                Prev      :  Pointer;
                Elemen :  TipeData; (* tipe data menurut pilihan kita *)
                Next      : Pointer
        End;
        List = Record
                Awal : Pointer;
```

Dari deklarasi di atas, maka simpul dapat digambarkan sebagai kotak (*record*) yang terdiri dari 3 bagian yakni Prev, yaitu *pointer* ke simpul sebelumnya, Elemen (data) dan Next, yaitu *pointer* ke simpul berikutnya, seperti terlihat pada gambar 2.1 di bawah ini.



Gambar 2.1 Gambaran Simpul List dengan Pointer Ganda

Kemudian apabila definisi *List* di-*copy* ke variabel L, maka L dapat digambarkan sebagai kotak yang terdiri dari 3 komponen yakni: L.Awal; *pointer* yang menunjuk elemen pertama dalam *List*, L.Akhir; *pointer* yang langsung menunjuk ke elemen terakhir dari *List*, L.Count; digunakan untuk mengetahui dengan cepat jumlah elemen yang terdapat dalam *List* L (mula-mula diberikan dengan nilai 0 dan ditambah setiap kali penambahan data dan dikurangi 1 jika dilakukan hapus), seperti terlihat pada gambar di bawah ini. L.Count berfungsi untuk menambah fleksibilitas dalam pendefinisian operasi-operasi *List*, seperti untuk mengetahui jumlah elemen dalam *List* sehingga dengan mudah diketahui jumlah perulangan dalam pencetakan, mengetahui apakah suatu *List* L kosong atau tidak.



Gambar 2.2 Representasi List dengan Pointer Ganda

Untuk melihat lebih jelas bagaimana definisi masing-masing operasi di atas dengan implementasi Array sesuai dengan deklarasi yang sudah diberikan, yaitu: *Makenul*, *Empty*, *Start*,*Insert*, *Delete*, *Print*, dan *Retrieve* seperti terlihat berikut ini:

1. Makenul adalah mengosongkan *List* L mula-mula sekali. Ini penting dilakukan untuk menjaga jangan sampai L yang sebelumnya mungkin sudah ada masuk sebagai bagian dari *List* L yang baru atau dengan kata lain agar kita peroleh *List* L yang bersih.

```
Procedure Makenul (NyatakanL : List);
    L.Awal := Nil;          (*set L.Awal = Nil *)
    L.Akhir := Nil;         (*set L.Akhir = Nil *)
    L.Count := 0;           (*set L.Count = 0 *)
```

Prosedur Makenul di atas menghasilkan List L dalam keadaan kosong seperti terlihat pada Gambar 2.11 di atas.

2. Empty adalah sebuah fungsi yang menghasilkan Benar atau Salah. Empty bernilai True jika L kosong dan bernilai False jika L tidak kosong.

```
Function Empty (L : List) : Boolean;
    Empty := (L.Awal = Nil) atau Empty := (L.Count = 0)
    atau Empty := (L.Akhir = Nil)          (* Pilih salah satu *)
```

3. Start merupakan procedure khusus untuk memulai *List* dari yang kosong menjadi list dengan 1 (satu) elemen. Ini hanyalah untuk kenyamanan sehingga pada waktu sisip elemen baru ke dalam *List* L, L minimum sudah berisi satu elemen.

```
Procedure Start (x : TipeData; Nyatakan L : List);
    NyatakanBantu : Pointer;
    1. New (Bantu);                (*ciptakan simpul baru ditunjuk oleh Bantu *)
```

2. Bantu^.Prev := Nil; (*set pointer mundur Bantu = Nil *)

3. Bantu^.Elemen := x; (*isikan elemen Bantu = x *)

4. Bantu^.Next := Nil; (*set pointer maju Bantu = Nil *)

5. L.Awal := Bantu; (*set L.Awal = Bantu *)

6. L.Akhir := Bantu; (*set L.Akhir = Bantu *)

7. L.Count := Inc(L.Count) atau L.Count :=1(*naikkan L.Count *)

4. Insert adalah prosedur untuk menambah data (elemen) yang baru ke dalam *List* L yang sudah ada. Insert dapat dilakukan di depan, di tengah maupun di akhir dari *List* L. Agar Insert tengah dapat dilakukan ada baiknya data dalam *list* L diurutkan menaik atau menurun sehingga kita dapat menentukan posisi data baru di dalam *list* L yang sudah ada.

a. Procedure InsertFront (x : TipeData; Nyatakan L : List);

NyatakanBantu : Pointer;

1. New (Bantu); (*ciptakan simpul baru ditunjuk oleh Bantu *)

2. Bantu^.Prev := Nil; (*set mundur Bantu = Nil *)

3. Bantu^.Elemen := x; (*isikan elemen Bantu = x *)

4. Bantu^.Next := L.Awal; (*set maju Bantu = L.Awal *)

5. L.Awal^.Prev := Bantu; (*set mundur L.Awal = Bantu *)

6. L.Awal := Bantu; (*set L.Awal = Bantu *)

7. L.Count := Inc(L.Count) atau L.Count :=L.Count+1 (*naikkan L.Count*)

b. Procedure InsertRear (x : TipeData; Nyatakan L : List);

NyatakanBantu : Pointer;

1. New (Bantu); (*ciptakan simpul baru ditunjuk oleh Bantu *)
2. Bantu^.Prev := L.Akhir; (*set mundur Bantu = L.Akhir *)
3. Bantu^.Elemen := x; (*isikan elemen Bantu = x *)
4. Bantu^.Next := Nil;(*set maju Bantu = Nil *)
5. L.Akhir^.Next := Bantu; (*set maju L.Akhir = Bantu *)
6. L.Akhir := Bantu; (*set L.Akhir = Bantu *)
7. L.Count := Inc(L.Count) atau L.Count :=L.Count+1 (*naikkan L.Count *)

c. Procedure InsertMiddle (x : TipeData; Nyatakan L : List);

Nyatakan P, Bantu : Pointer;

P := L.Awal;

Ketika (P^.Next^.Elemen< x)

P:= P^.Next;

1. New (Bantu); (*ciptakan simpul baru ditunjuk oleh Bantu *)
2. Bantu^.Prev := P; (*set mundur Bantu = P *)
3. Bantu^.Elemen := x; (*isikan elemen Bantu = x *)
4. Bantu^.Next := P^.Next; (*set maju Bantu = maju P *)
5. P^.Next^.Prev := Bantu; (*set maju mundur P = Bantu *)
6. P^.Next := Bantu; (*set maju P = Bantu *)
7. L.Count := Inc(L.Count) atau L.Count :=L.Count+1 (*naikkan L.Count *)

5. Delete adalah suatu prosedur untuk menghapus elemen dari dala list L yang tidak kosong. Jika List L berisi hanya 1 elemen, menghapus elemen tersebut mengakibatkan List menjadi kosong dan jika elemen List L lebih dari 1, maka menghapus 1 elemen mengakibatkan jumlah elemen berkurang satu.

Penghapusan dapat dilakukan di depan, di tengah atau di belakang list tergantung elemen yang akan dihapus. Dengan asumsi bahwa elemen dalam list L diurutkan menaik seperti sebelumnya, maka penghapusan di depan, di tengah atau di belakang ditentukan oleh x (elemen yang ingin dihapus). Jika x sama dengan elemen yang ditunjuk oleh L.Awal maka penghapusan dilakukan di depan, jika x sama dengan elemen yang ditunjuk oleh L.akhir maka dilakukan di belakang dan apabila tidak maka penghapusan dilakukan di tengah (ingat x tersebut mungkin tidak ada di dalam list dan ini harus diidentifikasikan sebagai suatu kesalahan). Juga perhatikan jika x lebih kecil dari elemen yang ditunjuk oleh L.Awal maka dengan cepat diketahui bahwa data tersebut tidak ada dalam list, dengan demikian maka operasi Delete(x,L) tersebut dinyatakan sebagai error. Hal yang sama berlaku jika x lebih besar dari elemen yang ditunjuk oleh L.Akhir

a. Procedure DeleteFront(Nyatakan L : List);

NyatakanBantu : Pointer

JikaL.Count = 1 Maka

Makenul(L)

Jika tidak, Begin

1. Bantu :=L.Awal;

2. L.Awal := L.Awal^.Next;

3. L.Awal^.Prev := Nil;

4. Bantu^.Next := Nil;

5. Dec(L.Count);

6. Dispose(Bantu)

End;

b. ProcedureDeleteRear(Nyatakan L: List);

NyatakanBantu : Pointer;

JikaL.Count = 1 Maka

Makenul(L)

Jika tidak, Begin

1. Bantu := L.Akhir;

2. L.Akhir := L.Akhir^.Prev;

3. L.Akhir^.Next := Nil;

4. Bantu^.Prev := Nil;

5. Dec(L.Count);

6. Dispose(Bantu)

End;

c. ProcedureDeleteMiddle(x : TipeData ; Nyatakan L: List);

Nyatakan P, Bantu : Pointer;

1. P := L.Awal;

2. Ketika P^.Next^.Elemen< x

3. Jika P^.Next^.Elemen = x Maka (* X sudah ditemukan *)

Begin

a. Bantu := P^.Next;

b. P^.Next := P^.Next^.Next;

c. P^.Next^.Prev := P;

d. Bantu^.Prev := Nil;

e. Bantu^.Next := Nil;

f. Dec(L.Count);

g. Dispose(Bantu)

End;

Jika (P = L.Akhir or P^.Next^.Elemen > x) Maka

Tampilkan pesan(‘Error : ‘,x, ‘ tidak ada ‘);

Dengan demikian maka prosedur hapus (delete) dapat didefinisikan sebagai berikut:

```
ProcedureDelete(x : TipeData; Nyatakan L : List);
    Jika x <L.AwalMaka
       Tampilkan pesan(‘Error : ‘,x,’ tidak ada’)
    Jika tidak,Jika x = L.Awal^.ElemenMaka
       DeleteFront(L)
    Jika tidak,Jika x = L.Akhir^.ElemenMaka
       DeleteRear(L);
    Jika tidak,
```

DeleteMiddle(x, L)

6. Retrieve adalah suatu fungsi yang menghasilkan tampilan “data tersebut ditemukan” jika data yang dicari ada dalam *list* L dan “data tersebut tidak ditemukan” jika data yang dicari tidak ada dalam *list* L yang tidak kosong. Pencarian dimulai dari depan dan berhenti jika L.Akhir sudah dicapai. Dengan asumsi bahwa *list* terurut menaik, maka Prosedur Retrieve dapa didefinisikan sebagai berikut:

```
ProcedureRetrieve(x:TipeData; L : List);
NyatakanP : Pointer;
  P:= L.Awal;
  Ketika (P^.Elemen> X And P<> L.Akhir )
        P:= P^.Next;
  Jika P^.Elemen = x Maka
        Tampilkan pesan(x,’ ditemukan dalam list’)
  Jika tidak,
        Tampilkan pesan(x, ‘ tidak ditemukan dalam list’)
```

7. Tampil adalah prosedur yang menampilkan isi dari sebuah list L. Penampilan dapat dilakukan dari depan ke belakang (tampil maju) atau dari belakang ke depan (tampil mundur). Kedua prosedur dapat didefinisikan sebagai berikut:

a.
```
ProcedureTampilMaju(L : List);
NyatakanP : Pointer;
```

```
P := L.Awal;
Ketika P <> Nil
Begin
        Tampilkan(P^.Elemen);
        P:=P^.Next;
End;
```

b. ProcedureTampilMundur(L:List);

```
Nyatakan P: Pointer;
P := L.Akhir;
Ketika P <> Nil
Begin
        Tampilkan(P^.Elemen);
        P:=P^.Prev;
End;
```

2.7 Pohon (Tree)

Pohon (*tree*) merupakan struktur data nonlinier yang banyak digunakan dalam aplikasi sehari-hari. Contoh aplikasi pohon yang dapat kita lihat sehari-hari adalah pengelolaan *file* dalam direktori penyimpanan. Pohon merupakan struktur data yang memiliki suatu struktur hirarki pada sekumpulan elemen, dan memiliki hubungan satuke banyak (*one to may relationship*) seperti yang kita lihat dalam struktur organisasi sebuah perusahaan atau daftar isi sebuah buku.

Dalam struktur organisasi, kita dapat melihat bahwa ada level atas biasanya hanya ada satu pimpinan tertinggi. Pada level berikutnya diisi oleh beberapa orang dengan jabatan yang berbeda tetapi dalam tingkatan yang sama. Selanjutnya dapat dipecah lagi ke level berikutnya sampai struktur dapat memenuhi fungsi dan tujuan organisasi. Biasanya satu atasan memiliki beberapa bawahan yang berada dalam ruang lingkup wewenang dan tugas atasan. Begitu juga dalam daftar isi buku,
dimana satu buku terdiri dari beberapa bab dan setiap terdiri dari beberapa sub bab, satu sub bab terdiri dari beberapa sub sub bab dan seterusnya. Dengan demikian hirarki dapat kita anggap sebagai "terdiri dari" atau "bawahan" atau "diawasi" dari atas ke bawah. Salah satu keuntungan pohon dibandingkan dengan struktur data linier adalah waktu cari sebuah node maksimum (dapat) lebih kecil dari n jika jumlah data = n.

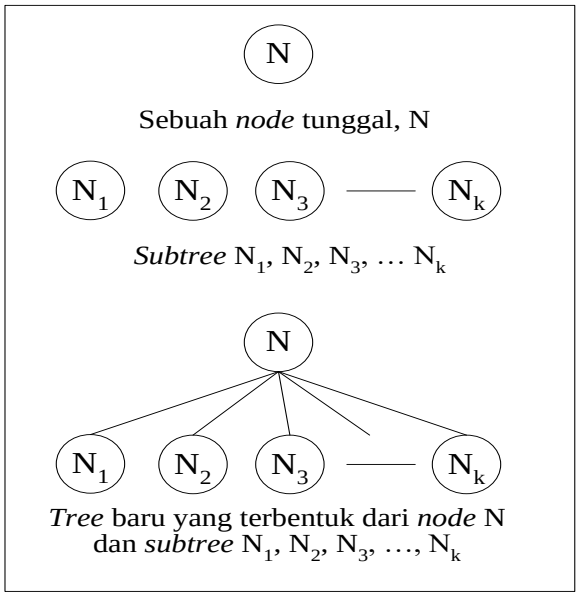
Sebuah *tree* dapat mempunyai hanya sebuah simpul tanpa sebuah sisi pun. Dengan kata lain, jika $G = (V, E)$ adalah tree, maka V tidak boleh berupa himpunan kosong, namun E boleh kosong. *Tree* juga seringkali didefinisikan sebagai graf tak-berarah dengan sifat bahwa hanya terdapat sebuah lintasan unik antara setiap pasang simpul. Selain itu, di dalam *tree* jumlah sisinya adalah jumlah simpul dikurangi satu.

Secara sederhana, sebuah *tree* bisa didefenisikan sebagai kumpulan dari elemen – elemen yang disebut dengan *node / vertex* (simpul) dimana salah satu *node* disebut dengan *root* (akar), dan sisa *node* lain terpecah menjadi himpunan yang saling tidak berhubungan satu sama lain dan disebut dengan *subtree* (pohon bagian). Jika

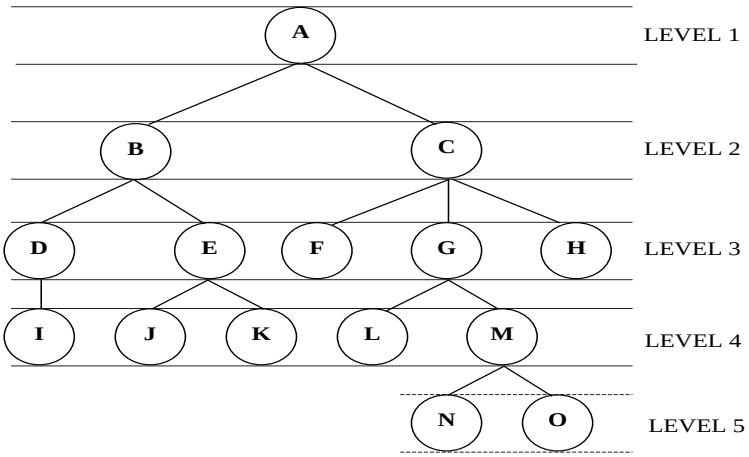
dilihat pada setiap *subtree* maka *subtree* juga mempunyai *root* dari *subtree*-nya masing – masing.

Dengan melihat istilah dasar di atas, maka sebuah *tree* secara rekursif dapat didefenisikan sebagai berikut :

1. Sebuah *node* tunggal adalah sebuah *tree*.
2. Jika terdapat sebuah *node* N dan beberapa *subtree* N_1, N_2, N_3, …, N_k maka dari *node* N dan *subtree* yang ada dapat dibentuk sebuah *tree* yang mempunyai *root* pada *node* N.



Gambar 2.3 Contoh pembentukan *Tree*



Gambar 2.4 Contoh *Tree* dengan 15 *node*

Seperti yang terlihat pada gambar 2.4, sebenarnya yang disebut dengan *node* itu adalah bagian dari *tree* yang berisikan data / informasi dan penunjuk percabangan. *Tree* pada gambar 2.4 berisi 15 *node* yang berisikan informasi berupa huruf A, B, C, D hingga huruf O lengkap dengan percabangannya masing – masing, dimana *tree* ini mempunyai *root* pada *node* A.

Hubungan antara satu *node* dengan *node* lain bisa dianalogikan seperti halnya dalam sebuah keluarga, yaitu ada anak, bapak, saudara, dan lain – lain. Dalam gambar 2.4*node* A adalah bapak dari *node* B dan C, dengan demikian *node* B dan C ini bersaudara. *Node* D dan E adalah anak dari *node* B. *Node* C adalah paman dari *node* D dan E.

Level (tingkatan) suatu *node* ditentukan dengan pertama kali menentukan *root* sebagai tingkat pertama. Jika suatu *node* dinyatakan sebagai tingkat N, maka *node* yang merupakan anaknya dikatakan berada dalam tingkat N + 1. Gambar 2.4

menunjukkan contoh *tree* lengkap dengan *level* pada setiap *node*. Di samping defenisi di atas, ada juga beberapa buku yang menyebutkan bahwa *root* dinyatakan sebagai *level* 0 dan *node* lain dinyatakan mempunyai *level* 1 tingkat lebih tinggi dari *root*.

Selain *level*, juga dikenal istilah *degree* (derajat) dari suatu *node*. *Degree* suatu *node* dinyatakan sebagai banyaknya anak atau turunan dari *node* tersebut. Sebagai contoh dalam gambar 2.4*node* A mempunyai *degree* 2, *node* B mempunyai *degree* 2, dan *node* C mempunyai *degree* 3. *Node* lain F, H, I, J, K, L, N, dan O yang semuanya mempunyai *degree* 0 disebut juga dengan *leaves* (daun). *Leaf* dan *root* tergolong *external node* (simpul luar), sedangkan selain *node* tersebut di atas disebut dengan *internalnode* (simpul dalam).

Height / depth (ketinggian / kedalaman) dari suatu *tree* adalah tingkat maksimum dari suatu *node* dalam *tree* tersebut dikurangi dengan 1. Dengan demikian *tree* pada gambar 2.4 yang mempunyai *root* pada *node* A mempunyai *height* 4.

Ancestor (leluhur) dari suatu *node* adalah semua *node* yang terletak dalam suatu jalur dengan *node* tersebut mulai dari *root* sampai *node* yang ditinjau. Sebagai contoh *ancestor* dari *node* L gambar 2.4 adalah *node* A, C, dan G.

Descendant (keturunan) dari suatu *node* adalah semua *node* yang terletak tepat di bawah *node* tersebut. Sebagai contoh *descendant* dari *node* G pada gambar 2.4 adalah *node* L, M, N, dan O.

Predecessor dari suatu *node* adalah semua *node* yang berada pada *level* di atas dari *level node* tersebut. Sebagai contoh *predecessor* dari *node* G pada gambar 2.4 adalah *node* A, B, dan C.

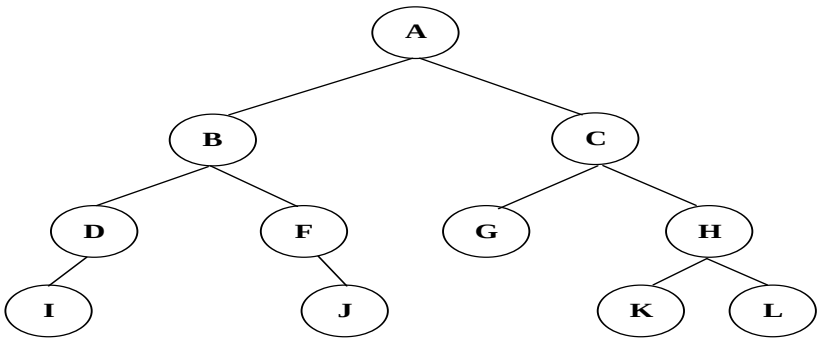
Successor dari suatu *node* adalah semua *node* yang berada pada *level* di bawah dari *level node* tersebut. Sebagai contoh *successor* dari *node* G pada gambar 2.4 adalah *node* I, J, K, L, M, N, dan O.

Sibling dari suatu *node* adalah semua *node* yang berada pada *level* yang sama yang berasal dari *node* asal (satu *level* di atas *node* yang ditinjau) yang sama dengan *node* tersebut. Sebagai contoh *sibling* dari *node* G pada gambar 2.4 adalah *node* F dan H.

2.8 Binary Tree

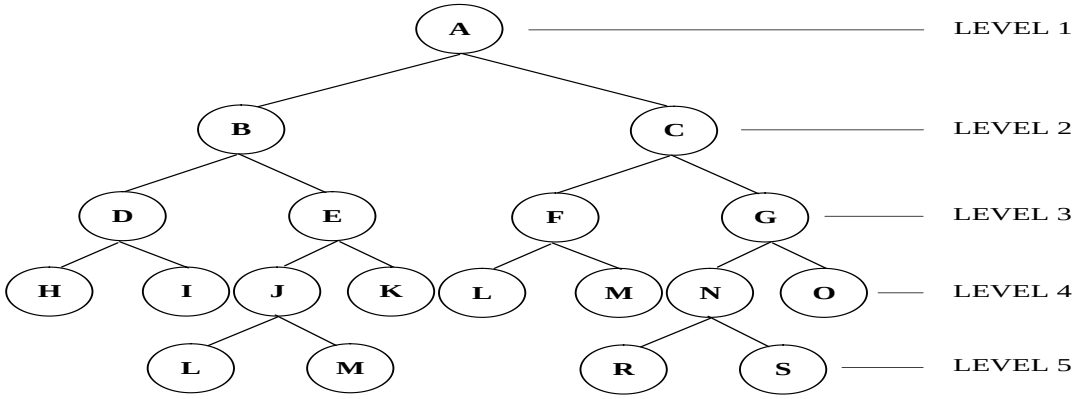
Binary Tree (pohon biner) didefenisikan sebagai suatu kumpulan *node* yang mungkin kosong atau mempunyai *root* dan paling banyak dua *subtree* (anak) yang saling terpisah yang disebut dengan *left subtree* (pohon bagian kiri/anak kiri / cabang kiri) dan *right subtree* (pohon bagian kanan / anak kanan / cabang kanan). *Subtree* bisa disebut juga dengan istilah *branch*(cabang).

Binary tree merupakan tipe yang sangat penting dari struktur data *tree*, dan banyak dijumpai dalam berbagai terapan. Lebih lanjut, dalam *binary tree* akan dibedakan antara *left subtree* dengan *right subtree*, Jadi *binary tree* merupakan bentuk *tree* yang beraturan. Karakteristik lain adalah bahwa dalam *binary tree* dimungkinkan tidak mempunyai *node*. Gambar 2.5 berikut ini menunjukkan contoh suatu *binary tree*.



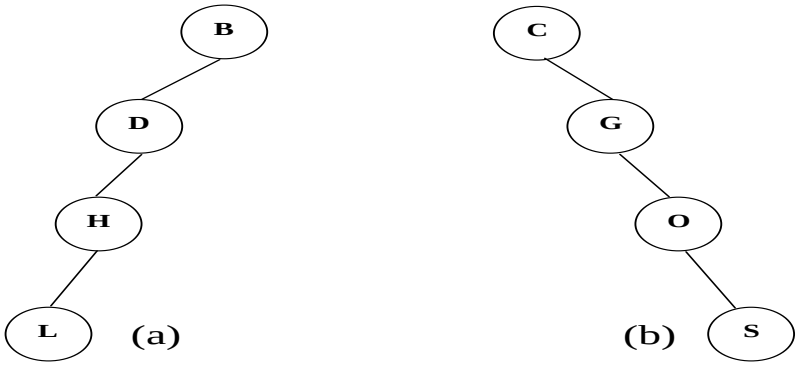
Gambar 2.5 Contoh *binary tree*

Pengertian *leaf*, *parent*, *child*, *level*, dan *degree* yang berlaku dalam *tree* juga berlaku dalam *binary tree*. Selain defenisi yang telah ada, dalam *binary tree* juga dikenal istilah *complete binary tree* (pohon biner lengkap) *level* N, yang didefenisikan sebagai sembarang *binary tree* dimana semua *leaf* – nya terdapat pada *level* N dan semua *node* yang mempunyai *level* lebih kecil dari N selalu mempunyai *left subtree*dan *right subtree*. Gambar 2.6 berikut ini merupakan contoh *complete binary tree level* 4, tetapi bukan *complete binary treelevel* 5.



Gambar 2.6 Complete *binary tree* level 4

Selain istilah *complete binary tree*, juga ada istilah *skewed binary tree* (pohon biner miring), yaitu suatu *binary tree* yang banyaknya *node* dalam *left subtree*tidak seimbang dengan banyaknya *node* dalam *right subtree*. Gambar 2.7 menunjukkan *right* dan *left skewed binary tree.*



Gambar 2.7 Contoh *skewed binary tree* (a) *Skewed left* (b) *Skewed right*

Dengan memperhatikan gambar sembarang *binary tree*, kita akan memperoleh tambahan informasi, yaitu banyaknya *node* maksimum pada *level* N adalah $2^{(N-1)}$. Sehingga banyaknya *node* maksimum sampai *level* N adalah :

$$2^{(i-1)} : \sum_{i=1}^{N} - 1$$

Dengan demikian untuk *complete binary tree level* 5, banyaknya *leaf* adalah 16 buah dan banyaknya *node* yang bukan *leaf*, termasuk *root*, adalah 15 buah.

2.9 Representasi Binary Tree

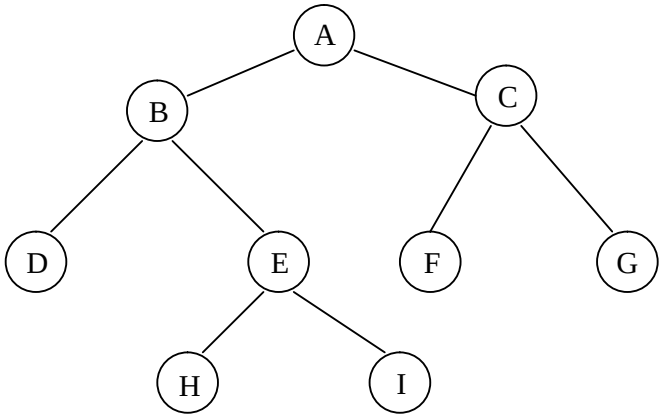
Pohon biner (*Binary Tree*) dapat disimpan / direpresentasikan dengan *array* ataupun *pointer* dimana masing-masing representasi memiliki kelebihan dan kekurangan.

a. Representasi Pohon Biner dengan Array

Pohon biner dapat direpresentasikan dengan *array* dengan ketentuan sebagai berikut :

a. Akar (r*oot)* ditempatkan pada posisi 1 dalam *array*
b. *Left subtree* data yang ada pada posisi ke-i diletakkan pada posisi 2i
c. *Right subtree* - nya diletakkan pada posisi 2i + 1

Sebagai contoh, misalkan terdapat suatu pohon biner seperti gambar 2.8.



Gambar 2.8 Pohon Biner dengan Jumlah Level 4

Sesuai dengan aturan pada gambar 2.8, maka jumlah data maksimum jika jumlah *level* = 4 adalah 15, sehingga kita menyediakan *array* dengan dimensi 15. Dengan demikian pohon di atas akan disimpan dalam *array* seperti Tabel 2.1 di bawah ini.

Tabel 2.1 Representasi *Array* untuk Pohon Biner pada Gambar 2.8

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	F	G			H	I				

Kelemahan dengan representasi *array* adalah sulitnya memperhitungkan jumlah sel *array* yang harus disediakan untuk bentuk pohon biner yang belum diketahui, karena permintaan tempat dilakukan sekaligus di awal program serta tidak dimungkinkan menambah selama eksekusi program. Kemudian jika bentuk pohon tidak seimbang (misalkan miring kiri atau kanan) maka jumlah tempat yang

terisi sangat sedikit dibandingkan dengan jumlah tempat yang sudah disediakan. Hal ini tentunya merupakan pemborosan. Di sisi lain representasi *array* memiliki keuntungan dalam waktu *akses*, karena *array* merupakan satu blok memori yang *contiguous.*

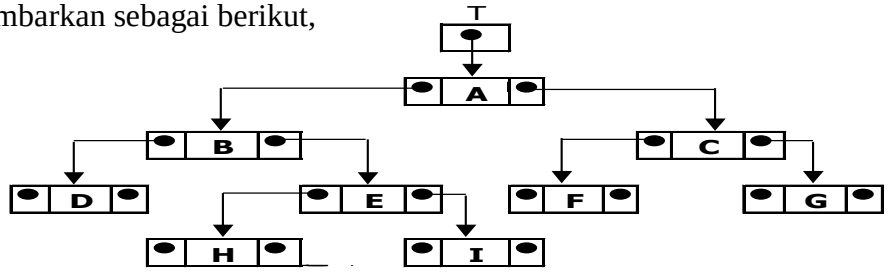
b. Representasi Pohon Biner dengan Linked List

Dengan mempertimbangkan kelemahan representasi *array*, maka pilihan dengan *linked list (pointer)* menjadi lebih sesuai dengan kondisi riel. Pohon biner

dengan representasi *pointer* hampir sama dengan *double linked list* dengan menggunakan dua buah pointer untuk setiap simpul yakni *pointer* kiri dan *pointer* kanan. Deklarasi pohon biner ini dapat dituliskan dengan illustrasi bahasa Pascal,

```
Type
       Pointer = ^Simpul;
       Simpul = Record
       Kiri: Pointer;
       Data: TipeData; (*sesuai dengan kebutuhan; misalnya char, integer*)
       Kanan : Pointer;
       End;
PohonBiner = Pointer;
```

Dengan representasi *linked list* maka pohon pada Gambar 2.8 di atas dapat digambarkan sebagai berikut,



Gambar 2.9 Representasi Linked List untuk Pohon Biner pada Gambar 2.8

Dari gambar 2.9 di atas pohon biner dengan nama T yaitu sebuah *pointer* yang menunjuk ke simpul yang berisi data A dengan *pointer* kiri menunjuk simpul yang berisi data C (sebagai cabang kanan dari A) dan seterusnya. Keuntungan representasi

linked list dibandingkan dengan representasi *array* adalah jumlah tempat yang dibutuhkan bersifat fleksibel.

2.10 Binary Tree Traversal (Penelusuran Pohon Biner)

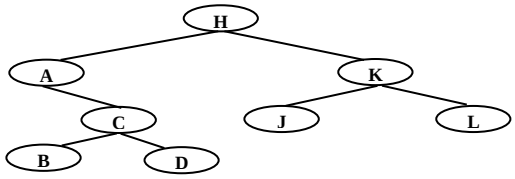
Pohon biner (*Binary Tree*) dapat ditelusuri dengan 4 cara yakni:

a. *Preorder Traversal* (Penelusuran Preorder)

Pre artinya **s**ebelum. Jadi pendaftaran *parent* dilakukan sebelum *subtree* - nya. Kemudian lakukan *preorder* terhadap *left subtree* dan *right subtree*. Sehingga kalau terhadap pohon pada Gambar 2.8 sebelumnya dilakukan penelusuran *preorder* diperoleh urutan sebagai berikut:

A, B, D, E, H, I, C, F, G

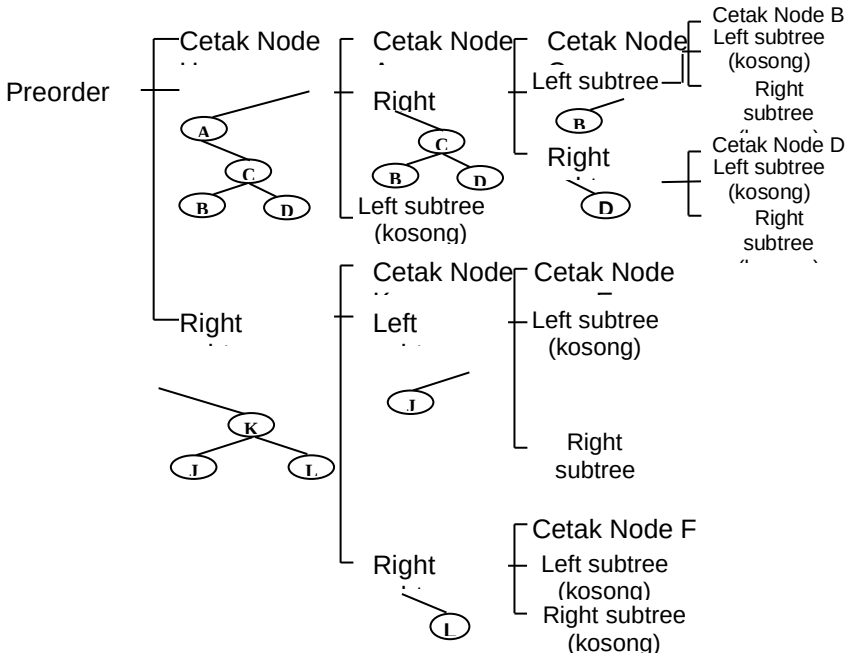
Rumus untuk *preorder traversal* adalah : Atas, Kiri (kalau ada), Kanan (kalau ada).



Gambar 2.10 Contoh binary tree

Dengan representasi *linked list* maka pohon pada Gambar 2.8 di atas dapat digambarkan sebagai berikut

,



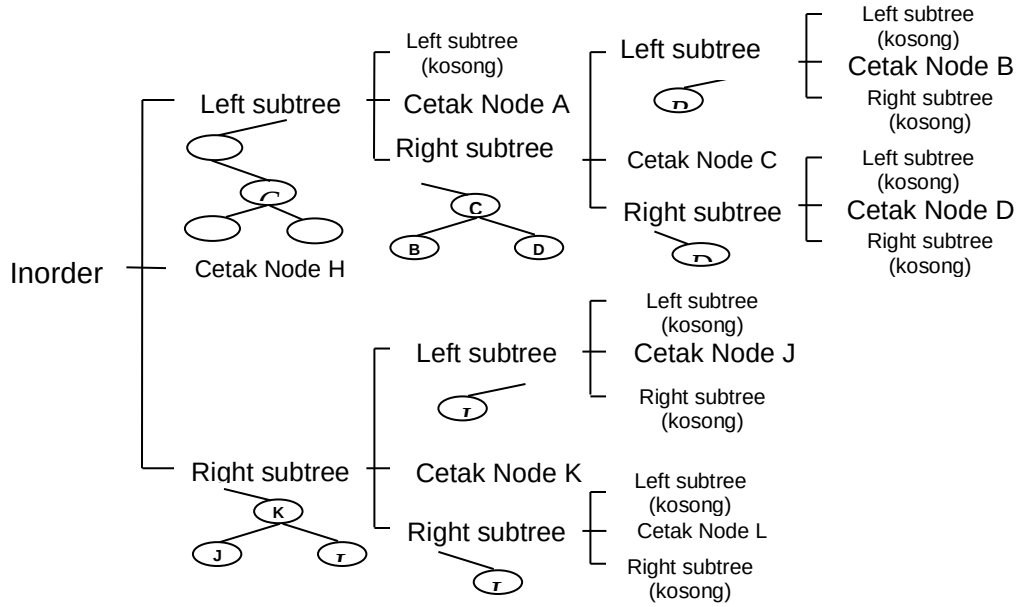
Gambar 2.11 Representasi Linked List untuk Pohon Biner pada Gambar 2.8

Dari gambar 2.9 di atas pohon biner dengan nama T yaitu sebuah *pointer* yang menunjuk ke simpul yang berisi data A dengan *pointer* kiri menunjuk simpul yang berisi data C (sebagai cabang kanan dari A) dan seterusnya. Keuntungan representasi

linked list dibandingkan dengan representasi *array* adalah jumlah tempat yang dibutuhkan bersifat fleksibel

A, B, D, E, H, I, C, F, G

Rumus untuk *preorder traversal* adalah : Atas, Kiri (kalau ada), Kanan (kalau ada).



Gambar 2.12 Proses Inorder Traversal dari binary Tree gambar 2.10

b. *Postorder Traversal* (Penelusuran Postorder)

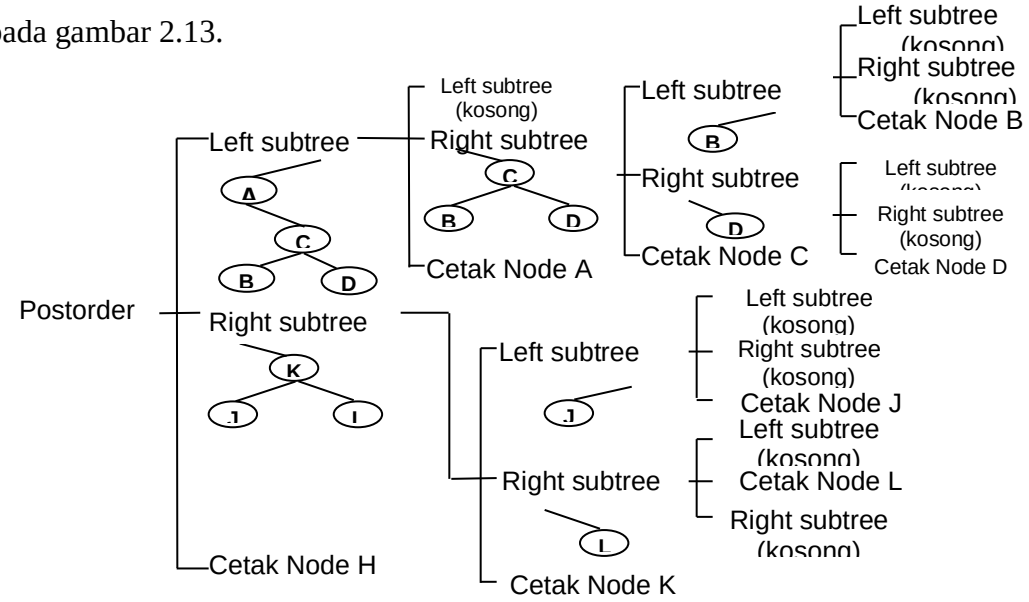
*Post*artinya **setelah**. Jadi pendaftaran *parent* - nya dilakukan setelah *left subtree* dan *right subtree*. Untuk menelusuri *left subtree* dan *right subtree* lakukan dengan cara postorder pula. Sehingga kalau terhadap pohon pada Gambar 2.8 di atas dilakukan penelusuran *postorder* diperoleh urutan sebagai berikut:

D, H, I, E, B, F, G, C, A

Rumus untuk Postorder Traversal adalah : Kiri (kalau ada), Kanan (kalau ada), Atas

.

Jika dilakukan *postorder traversal* terhadap binary tree gambar 2.10, akan menghasilkan data “B, D, C, A, J, L, K, H” dimana proses *traversal* ini bisa dilihat pada gambar 2.13.



Gambar 2.13 Proses Postorder Traversal dari binary tree gambar 2.10

c. *Levelorder Traversal* (Penelusuran Level demi Level)

Penelusuran dengan cara ini dilakukan mulai dari *root* kemudian pendaftaran *level* demi *level* dari kiri ke kanan. Sehingga kalau terhadap pohon pada Gambar 2.8 di atas dilakukan penelusuran *by level* diperoleh urutan sebagai berikut:

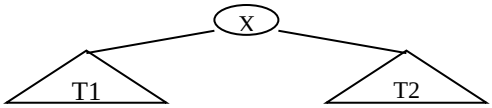
A, B, C, D, E, F, G, H, I

Jika dilakukan *levelorder traversal* terhadap *binary tree* gambar 2.10, akan menghasilkan data “H, A, K, C, J, L, B, D”.

2.11 Binary Search Tree (BST)

Binary Search Tree (BST) adalah pohon biner yang berisi elemen sejenis dan unik dengan beberapa sifat khusus. Perhatikan contoh berikut.

Misalkan diketahui sebuah pohon biner dengan akar X serta sub pohon kiri T1 dan sub pohon kanan T2 seperti Gambar 2.14 di bawah ini:



Gambar 2.14 Bentuk Pohon Biner dengan Akar X dan sub Pohon T1 dan T2

Binary tree di atas disebut pohon pencarian biner (BST) jika syarat-syarat berikut dipenuhi:

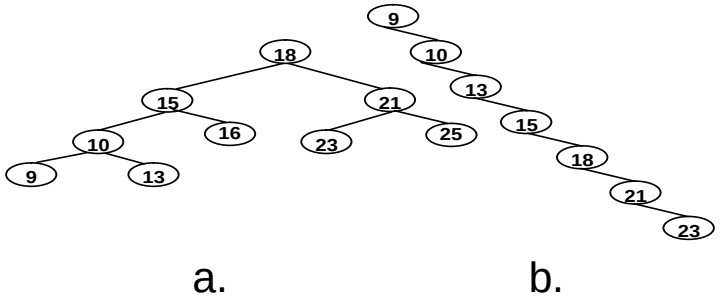
1. $\forall Y \in T1 < X < \forall Z \in T2$ dimana T1 dan T2 juga BST
2. Apabila dilakukan *Inorder Traversal* akan diperoleh urutan naik
3. *Binary tree* dengan 1 elemen dan pohon kosong adalah bentuk khusus BST

Pohon dengan sifat di atas disebut dengan pohon pencarian biner (BST) naik. BST turun dengan mudah didefinisikan dengan mengubah sifat-sifat di atas sebagai berikut:

1. $\forall Y \in T1 > X > \forall Z \in T2$ dimana T1 dan T2 juga BST
2. Apabila dilakukan Inorder Traversal akan diperoleh urutan turun
3. *Binary tree* dengan 1 elemen dan pohon kosong adalah bentuk khusus BST

BST pada prinsipnya tidak membiarkan adanya duplikasi elemen, sehingga BST adalah struktur yang merupakan set (himpunan).

Untuk jelasnya, dapat dilihat pada dua buah *binary tree* berikut ini yang memenuhi kriteria pembuatan di atas.



Gambar 2.15 Contoh binary tree (a) Skewed left (b) Skewed right

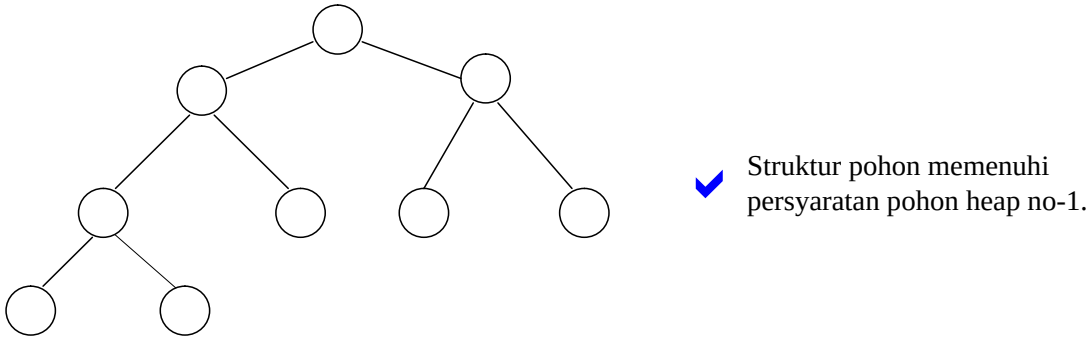
Dari kedua *binary tree* pada gambar 2.15 bisa dilihat bahwa proses pencarian data secara umum akan lebih cepat dilaksanakan pada *tree* pertama (gambar 2.15.a) dibandingkan dengan *tree* kedua (gambar 2.15.b). Sebagai contoh untuk mencari bilangan 18, pada *tree* pertama cukup dilaksanakan satu kali perbandingan, sedangkan pada *tree* kedua, perlu dilakukan lima kali perbandingan, sama halnya untuk mencari bilangan 23 dimana *tree* pertama memerlukan tiga kali perbandingan, sedangkan *tree* kedua memerlukan tujuh kali perbandingan. Dari kedua contoh tersebut dapat dilihat bahwa pencarian pada *tree* pertama lebih cepat dilaksnakan

dibanding pencarian pada *tree* kedua. Perbedaan akan semakin nyata apabila banyaknya data yang diketahui semakin banyak.

2.12 Heap Tree (Pohon Heap)

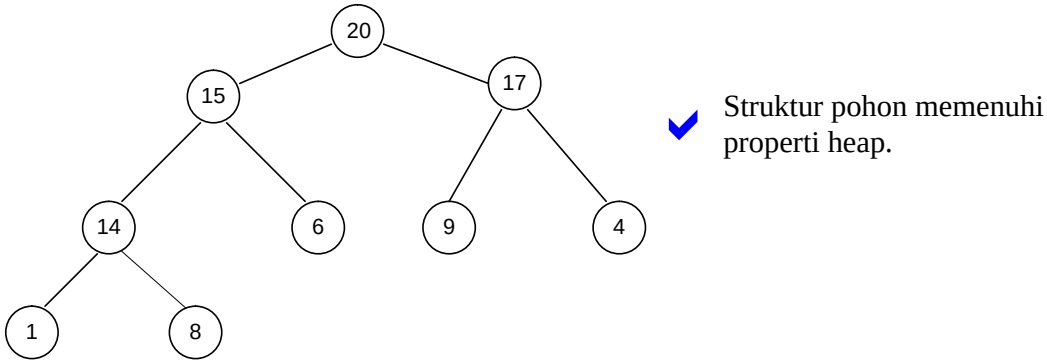
Heap Tree adalah pohon biner yang memiliki beberapa persyaratan berikut:

1. Struktur pohon harus lengkap atau sempurna, kecuali untuk *level* yang terbawah (dengan syarat: semua *node* pada level terbawah harus berada di sebelah kiri). Perhatikan contoh berikut.



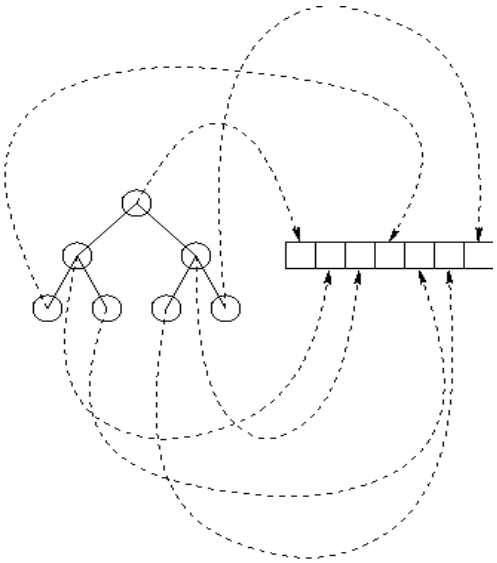
Gambar 2.16 Struktur pohon memenuhi persyaratan pohon *heap* no-1.

2. Struktur pohon memenuhi properti *heap*, yaitu:
   a. *Node* akar (*root node*) memiliki data terbesar atau terkecil yang terdapat pada pohon.
   b. *Subtree* di sebelah kiri dan sebelah kanan *node* akar memenuhi persyaratan berikut: *node* induk (*parent*) memiliki data yang paling besar atau paling kecil dibandingkan dengan data pada kedua anaknya (*child node* sebelah kiri atau sebelah kanan).



Gambar 2.19 Struktur pohon memenuhi properti *heap*.

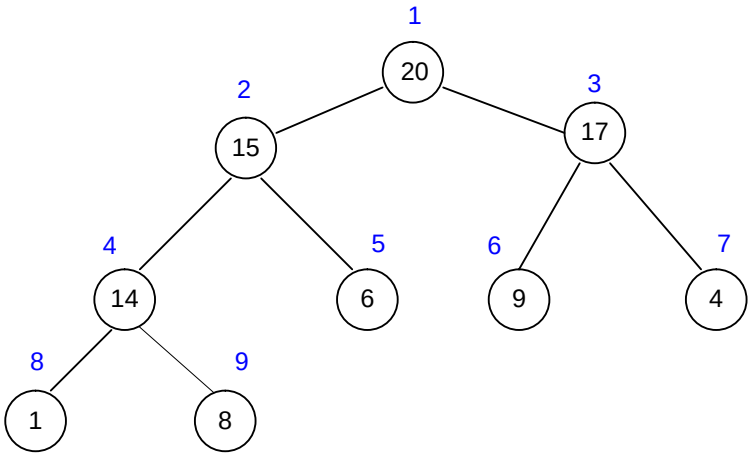
Struktur pohon *heap* dapat direpresentasikan atau disimpan dalam bentuk *array*. Setiap *node* pada pohon *heap* mewakili satu tempat di dalam *array*. Lokasi *Array* akan diisi penuh dari sebelah kiri ke sebelah kanan.



Gambar 2.21 Representasi pohon *heap* pada *array*.

Untuk mendapatkan lokasi *arraynode parent*, *node* anak sebelah kiri dan *node* anak sebelah kanan dari sebuah *node* (*i*), lakukan perhitungan sebagai berikut:

a. *Node* akar diberi *index array* 1, *node* level-2 diberi *index array* 2 dan 3, dan seterusnya.



Gambar 2.22*Index array* pada pohon *heap*.

b. Untuk mendapatkan lokasi *arraynode parent* dari sebuah *node* dengan indeks *arrayi*, lakukan perhitungan berikut:

$$\text{PARENT } (i) = \textit{return floor}(i/2)$$

c. Untuk mendapatkan lokasi *arrayleftchildnode* (*node* anak sebelah kiri) dari sebuah *node* dengan indeks *arrayi*, lakukan perhitungan berikut:

$$\text{LEFT } (i) = \textit{return } 2i$$

d. Untuk mendapatkan lokasi *arrayrightchildnode* (*node* anak sebelah kanan) dari sebuah *node* dengan indeks *arrayi*, lakukan perhitungan berikut:

RIGHT (i) = *return* ($2i$ + 1)

Prosedur dasar yang terdapat dalam *heap tree* adalah:

1. Heapify.

Prosedur ini berfungsi untuk mengatur atau memanipulasi penempatan data pada pohon *heap* agar tetap memenuhi properti *heap*, terutama setelah pemasukan data (*insertion*) atau penghapusan data (*deletion*). Properti *heap*, sebagaimana telah dijelaskan, mengharuskan data pada *node* induk lebih besar atau sama dengan data yang terdapat pada *node* anak sebelah kiri atau kanan. Apabila pada pohon *heap* ditemukan data yang melanggar aturan ini, maka prosedur Heapify akan menukarkan data pada *node* induk dengan data terbesar yang

terdapat pada *node* anak (*exchange*). Berikut adalah algoritma prosedur Heapify:

HEAPIFY (A, i)

1. $l \leftarrow$ left [i]
2. $r \leftarrow$ right [i]
3. if $l \leq$ heap-size [A] and $A[l] > A[i]$
4. then largest $\leftarrow l$
5. else largest $\leftarrow i$
6. if $r \leq$ heap-size [A] and $A[i] > A$[largest]
7. then largest $\leftarrow r$
8. if largest $\neq i$
9. then exchange $A[i] \leftrightarrow A$[largest]

10. Heapify (*A*, largest)

11.

2. Build-Heap.

Apabila diberi *input* sebuah pohon biner atau sekumpulan data *array*, maka untuk menjadikan pohon biner ini pohon *heap*, kita harus memastikan semua data pada pohon biner memenuhi properti heap. Berikut adalah algoritma Build-Heap:

BUILD-HEAP (*A*)

1. heap-size (*A*) ← length [*A*]
2. For *i*← floor(length[*A*]/2) down to 1 do
3. Heapify (*A*, *i*)

2.13 Algoritma

Kata algoritma berasal dari nama seorang ahli matematika berkebangsaan Persia yang hidup pada abad ke – 9 yang bernama Abu Abdullah Muhammad bin Musa Al-Khawarizmi. Pada awalnya, kata ‘algorism’ diartikan sebagai aturan-aturan untuk melakukan proses aritmatika menggunakan numerik Arab. Kata ‘algorism’ diubah menjadi kata ‘algorithm’ pada abad ke – 18. Sekarang, pengertian dari kata ini mencakup semua prosedur terhingga untuk menyelesaikan problema atau melakukan pekerjaan.

Penerapan pertama dari algoritma yang ditulis untuk sebuah komputer adalah ‘*Ada Byron’s notes on the analytical engine*’ yang ditulis pada tahun 1842, dimana Ada Byron dianggap oleh kebanyakan orang sebagai *programmer* pertama di dunia.

Walaupun, sejak Charles Babbage tidak menyelesaikan *analytical engine*-nya, algoritma ini tidak pernah diimplementasikan lagi.

2.14 Definisi Algoritma

Algoritma adalah suatu kumpulan terhingga (*finite set*) dari instruksi yang terdefinisi dengan baik (*well-defined instructions*) untuk menyelesaikan beberapa pekerjaan dimana diberikan *state* awal (*initial state*) dan akan dihentikan pada saat ditemukan *state* akhir (*end-state*) yang dikenal. Algoritma dapat diimplementasikan dalam pembuatan program komputer. Kesalahan dalam merancang algoritma untuk menyelesaikan suatu problema dapat menyebabkan program gagal dalam implementasinya.

Konsep dari suatu algoritma sering diilustrasikan dengan mengambil contoh sebuah resep, walaupun banyak algoritma yang jauh lebih kompleks. Algoritma sering memiliki beberapa langkah perulangan (iterasi) atau memerlukan pengambilan keputusan seperti logika (*logic*) atau perbandingan (*comparison*) sampai pekerjaan diselesaikan. Menerapkan suatu algoritma secara benar belum tentu dapat

menyelesaikan problema. Hal ini dikarenakan adanya kemungkinan algoritma tersebut rusak atau cacat, atau penerapannya tidak cocok (tidak tepat) untuk menyelesaikan problema. Sebagai contoh, sebuah algoritma hipotesis untuk membuat sebuah salad kentang akan gagal jika tidak terdapat kentang.

Suatu pekerjaan dapat diselesaikan dengan menggunakan algoritma yang berbeda dengan kumpulan instruksi (*set of instructions*) yang berbeda dengan

perbedaan waktu akses, efisiensi tempat, usaha dan sebagainya. Sebagai contoh, diberikan dua buah resep yang berbeda untuk membuat salad kentang, resep pertama mengupas kulit kentang terlebih dahulu sebelum memasak kentang tersebut, sementara resep lainnya dilakukan dengan langkah yang terbalik, dan kedua resep akan mengulangi kedua langkah tersebut dan akan dihentikan pada saat salad kentang siap untuk dimakan.

Algoritma adalah hal yang mendasar untuk komputer dalam memproses informasi, karena sebuah program komputer adalah sebuah algoritma yang memberitahukan kepada komputer langkah – langkah spesifik yang akan dijalankan (dalam urutan spesifik) untuk melakukan pekerjaan tertentu, misalnya menghitung gaji karyawan atau mencetak rapor murid. Oleh karena itu, algoritma dapat dianggap sebagai beberapa operasi sekuensial (terurut) yang dapat dijalankan oleh sebuah sistem lengkap *Turing*.

Hukum di beberapa negara seperti Amerika Serikat, mengizinkan beberapa algoritma untuk mendapatkan hak paten secara efektif, walaupun kemungkinan

tersedianya algoritma tersebut dalam sebuah perwujudan fisik (*physical embodiment*). Sebagai contoh, sebuah algoritma perkalian mungkin diwujudkan dalam unit aritmatika dalam sebuah mikroprosesor.

2.15 Kompleksitas Algoritma

Suatu algoritma memiliki kemungkinan terbaik (*best case*) dan kemungkinan terburuk (*worst case*). *Best Case* maksudnya adalah waktu eksekusi tercepat dari

algoritma, sedangkan *Worst Case* maksudnya adalah waktu eksekusi terlama dari algoritma. Waktu eksekusi dari algoritma ini biasanya disebut dengan kompleksitas algoritma. Kompleksitas algoritma biasanya dinyatakan dengan notasi O (*Big-O*).Suatu algoritma dikatakan memiliki kompleksitas $O(n^2)$ berarti bahwa waktu eksekusi terlama dari algoritma tersebut adalah sebesar kuadrat dari banyaknya elemen data yang akan diproses.

Semakin kecil kompleksitas suatu algoritma maka algoritma tersebut dikatakan lebih efisien dan efektif. Algoritma semacam inilah yang lebih disukai orang untuk digunakan dalam penyelesaian masalah.

Dalam analisis matematika khususnya analisis algoritma, untuk menentukan pertumbuhan dari fungsi, dapat digunakan *asymtotic notations* (notasi asimtot). Knuth mengutarakan beberapa notasi antara lain,

- *Big-O*
- *Big-Ω (Omega)*
- *Big-Θ (Theta)*

Notasi – notasi di atas tidak memiliki sifat matematika konvensional dan tidak berlaku ketentuan – ketentuan matematika di dalamnya. Suatu fungsi f(x) memiliki notasi O(x) dan fungsi lainnya g(x) juga memiliki notasi O(x), namun ini tidak berarti bahwa f(x) sama dengan g(x).

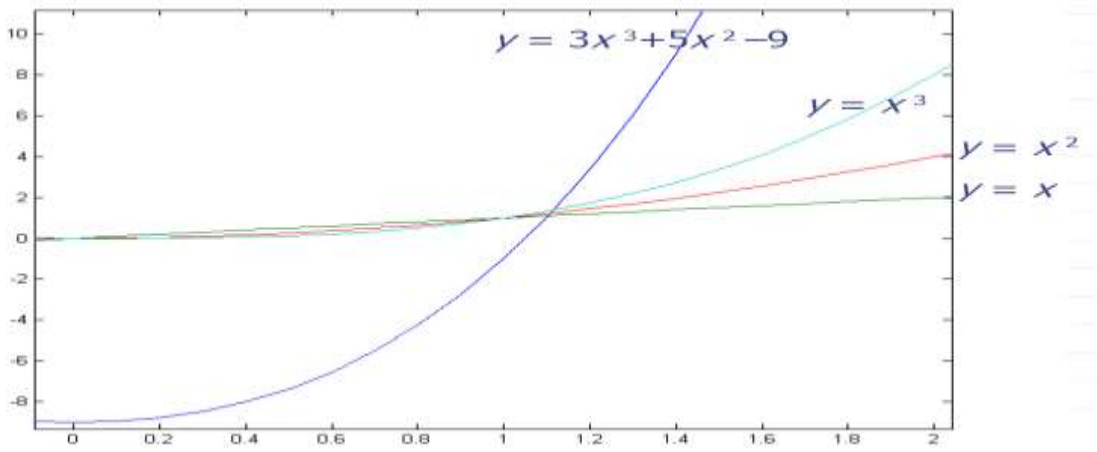
Notasi *Big-O* adalah suatu cara untuk membandingkan fungsi dan sangat berguna untuk menghitung kompleksitas dari algoritma, misalnya banyaknya waktu yang diperlukan oleh komputer untuk menjalankan program.

Contoh,

$3x^3 + 5x^2 – 9 = O(x^3)$

Persamaan di atas tidak berarti bahwa ada suatu fungsi $O(x^3)$ yang sama dengan fungsi $3x^3 + 5x^2 – 9$, namun persamaan tersebut berarti bahwa $3x^3 + 5x^2 – 9$ memiliki big-O x^3 atau dapat dikatakan bahwa $3x^3 + 5x^2 – 9$ didominasi secara asimtot oleh x^3. Notasi asimtot mencerminkan kelakuan dari fungsi untuk nilai yang besar sehingga notasi asimtot kadang kala tidak berlaku untuk nilai yang kecil.

Contoh perbandingan notasi *Big-O* dengan skala 0,2.



Gambar 2.23 Contoh perbandingan notasi *Big-O*

Sementara *Big-Ω* merupakan kebalikan dari *Big-O*. Hal ini dapat diilustrasikan seperti berikut,

$$\mathbf{f(x) = \Omega(g(x)) \Leftrightarrow g(x) = O(f(x))}$$

Sedangkan *Big-Θ* menyatakan bahwa fungsi – fungsi saling mendominasi satu sama lain, sehingga fungsi – fungsi tersebut dikatakan ekivalen secara asimtotik.

$$\mathbf{f(x) = \Theta(g(x)) \Leftrightarrow (f(x) = O(g(x)) \wedge f(x) = \Omega(g(x)))}$$

Beberapa penggunaan dari notasi asimtot dapat dilihat pada contoh di bawah ini,

- *Big-Θ* dari polinomial adalah bentuk variabel dengan orde terbesar.

  $x^4/100000 + 3x^3 + 5x^2 - 9 = \Theta(x^4)$

- Penjumlahan dari dua fungsi memiliki Big-O dari fungsi dengan orde terbesar.

  $x^4 \ln(x) + x^5 = O(x^5)$

- Konstanta di samping variabel tidak diambil.

  $17x^4 \ln(x) = O(x^4 \ln(x))$

2.16 Pengurutan (Sorting)

Pengurutan (sorting) merupakan pekerjaan yang sering kita lakukan dalam kehidupan sehari-hari. Tujuan dari pengurutan adalah menyusun sejumlah data sedemikian sehingga diperoleh suatu keteraturan apakah terurut menaik atau terurut menurun. Dengan adanya data terurut, disamping enak dipandang maka pencarian dapat dilakukan dengan cepat baik dengan algoritma biner maupun dengan rumus interpolasi. Hal ini sangat penting apalagi jika jumlah data sangat besar. Akan tetapi yang menjadi perhatian kita adalah bagaimana mendapatkan algoritma pengurutan yang efisien atau cepat. Sebab mengurutkan data dengan jumlah besar dan apalagi

jika data sering berubah (bertambah atau berkurang) sehingga pengurutan harus dilakukan berulang-ulang, maka kita sangat membutuhkan algoritma pengurutan yang sangat cepat.

Dalam pengurutan data, waktu yang diperlukan untuk melakukan proses pengurutan perlu dipertimbangkan dan juga masih ada beberapa aspek lain yang harus diperhatikan. Data yang harus kita urutkan tentunya sangat bervariasi baik dalam hal banyak data maupun jenis data yang akan diurutkan. Dalam hal ini, tidak ada satu algoritma yang terbaik untuk setiap situasi yang kita hadapi. Bahkan cukup sulit untuk menentukan algoritma mana yang paling baik untuk situasi tertentu karena ada beberapa faktor yang mempengaruhi efektivitas algoritma pengurutan.

Beberapa faktor yang berpengaruh dalam efektivitas suatu algoritma pengurutan antara lain:

3. Banyak data yang diurutkan.
4. Kapasitas pengingat apakah mampu menyimpan semua data yang kita miliki.
5. Tempat penyimpanan data, seperti piringan, pita, kartu atau media penyimpan yang lain.

Keuntungan ataupun tujuan dari pengurutan data adalah bahwa data akan lebih mudah dicari, mudah untuk dibetulkan, dihapus, disisipi atau digabungkan. Dalam keadaan terurut, kita mudah mencek apakah ada data yang hilang. Hal tersebut sangat penting dalam pemograman *database*. Bayangkan saja, jika komputer diperintah untuk mencari sebuah data dari sepuluh juta buah data yang tidak terurut,

maka komputer secepat apapun juga akan memerlukan waktu yang tidak sedikit untuk dapat menemukan data tersebut dan tentunya hal ini akan menganggu efektivitas dan efisiensi kerja. Terdapat beberapa algoritma yang dapat digunakan untuk mengurutkan data. Berikut pembahasan beberapa algoritma pengurutan.

2.17 Bubble Sort

Metode gelembung (Bubble Sort), sering juga disebut dengan metode penukaran (Exchange Sort). Metode ini merupakan metode yang mendasarkan penukaran dua buah elemen untuk mencapai keadaan urut yang diinginkan. Metode ini cukup mudah untuk dipahami dan diprogram. Tetapi dari beberapa metode pengurutan yang ada, metode ini merupakan metode yang paling tidak efisien. Metode pengurutan Bubble Sort memiliki ide pemikiran yang sangat sederhana yakni dengan membayangkan bahwa barisan elemen yang ingin diurutkan dibuat *array* secara vertikal. Misalkan kita ingin mengurutkan barisan N elemen menjadi urutan menaik atau *ascending*. Dimulai dari bawah (dasar) : bandingkan dua elemen yang berdekatan, maka elemen yang ringan (kecil) akan mengambang sampai ke atas, sementara elemen yang berat (besar) akan menyelam ke dasar. Berikut merupakan penggalan prosedur Buble Sort dalam bahasa Pascal,

```
Procedure BubbleSort(var A:Data; N:integer ; Sort:Byte);
var
   i, j : integer;
begin
     for i:=1 to N-1 do
     begin
          for j:=i+1 to N do
```

```
          begin
               case sort of
1 : if A[i]>A[j] then      {terurut menaik}
                        tukar(A[i],A[j]);
2 : if A[i]<A[j] then      {terurut menurun}
                        tukar(A[i],A[j]);
               end;
          end;
     end;
end;
```

2.18 Selection Sort

Cara kerja metode seleksi adalah sebagai berikut: Misalkan sekelompok data dengan jumlah data sebanyak n buah. Pada langkah pertama, dicari/dipilih data yang terkecil dari data pertama sampai data ke-n. Kemudian data terkecil tersebut kita tukar dengan data pertama. Dengan demikian, data pertama sekarang mempunyai nilai paling kecil dibanding data yang lain. Pada langkah kedua, data terkecil kita cari/pilih mulai dari data kedua sampai data ke-n. Data terkecil yang kita peroleh ditukar dengan data kedua. Demikian seterusnya sampai semua data dalam keadaan terurutkan. Berikut merupakan penggalan prosedur Selection Sort dalam bahasa Pascal,

```
Procedure Selection(var A:Data; N:integer ; Sort:Byte);
var
   X, J, Pos : integer;
begin
     for I:=1 TO N-1 DO
     begin
Pos:=I; X:=A[I];
          for J:=I+1 to N do
          begin
               case Sort of
1 : if X>A[J] then         {terurut menaik}
                    begin
```

```
X:=A[J];
Pos:=J;
                    end;
2 : if X<A[J] then          {terurut menurun}
                    begin
X:=A[J];
Pos:=J;
                    end;
                end;
           end;
           Tukar(A[I],A[Pos]);
      END;
END;
```

2.19 Insertion Sort

Ide dari Insertion Sort sangatlah sederhana yakni: Pada tahap ke-i (i = 2 sampai N), kita menyisipkan elemen ke-i pada posisi yang sesuai di antara A[1] .. A[i-1] yang sudah terurut di antara mereka sendiri (lokal). Setelah fase ke-I maka A[1] .. A[i] terurut secara lokal. Untuk menghindari agar penyisipan tidak sampai melewati batas A[1], maka perlu ditambah elemen artifisial A[0] yang merupakan bilangan sangat kecil jika kita ingin mengurutkan menaik dan bilangan sangat besar bila kita mengurutkan menurun. Berikut merupakan penggalan prosedur Insertion Sort dalam bahasa Pascal,

```
Procedure Insertion(var A:Data; N:integer; Sort:Byte);
var
   I, J : integer;
begin
     if sort = 1 then
A[0] := -MaxInt
     else
A[0] := MaxInt;
     for i:=2 to N do
     begin
J:=i;
```

```
          case sort of
1 : while A[J-1]>A[J] do   {terurut menaik}
                    begin
                         Tukar(A[J-1],A[J]);
J:=J-1;
                    end;
2 : while A[J-1]<A[J] do   {terurut menurun}
                    begin
                         Tukar(A[J-1],A[J]);
J:=J-1;
                    end;
          end;
     end;
end;
```

2.20 Shell Sort

Metode Shell Sort sering juga disebut dengan metode pertambahan menurun (Diminishing Increment). Metode ini dikembangkan oleh Donald L.Shell pada tahun 1959. Metode ini juga memanfaatkan penukaran sepasang elemen untuk mencapai keadaan urut. Dalam hal ini jarak dua elemen yang dibandingkan dan ditukarkan tertentu. Berikut merupakan penggalan prosedur Shell Sort dalam bahasa Pascal,

```
Procedure Shell(var A:Data; N:integer ; Sort:Byte);
Var
   Jarak, J:BYTE;
Begin
Jarak := N Div 2;
     while jarak > 0 do
     begin
          for I:=jarak+1 to N do
          begin
J:=I-jarak;
               while J>0 do
               Case Sort of
1 : if A[J] > A[J+jarak] then {terurut menaik}
                    begin
                         Tukar(A[J],A[J+jarak]);
J:=J-1;
```

```
                    end
                    else
J:=0;
2 : if A[J] < A[J+jarak] then {terurut menurun}
                    begin
                         Tukar(A[J],A[J+jarak]);
J:=J-1;
                    end
                    else
J:=0;
                end;
           end;
Jarak:=Jarak div 2;
     end;
end;
```

2.21 Quick Sort

Metode Quick Sort sering juga disebut dengan metode partition exchange sort. Metode ini diperkenalkan oleh C.A.R Hoare. Untuk mempertinggi efektifitasnya, dalam metode ini jarak kedua elemen yang akan ditukarkan nilainya cukup besar. Berikut merupakan penggalan prosedur Quick Sort dalam bahasa Pascal,

```
Procedure Quick(var A:Data ; awal,akhir,Sort:BYTE);
var
   I, J : integer;
   Procedure Atur;
   BEGIN
I:=awal+1; J:=akhir;
      Case Sort Of
1 : Begin  {terurut menaik}
             While A[awal] > A[I] do
                    Inc(I);
             While A[J] > A[awal] do
                    Dec(J);
           end;
2 : Begin  {terurut menurun}
             While A[awal]<A[I] do
                    Inc(I);
             While A[J]<A[awal] do
                    DEC(J);
```

```
          end;
      end;
      While I<J do
      begin
           Tukar(A[I],A[J]);
           Case sort of
1 : Begin {terurut menaik}
                    while A[awal] > A[I] do
                          Inc(I);
                    while A[J] > A[awal] do
                          Dec(J);
               END;
2 : Begin {terurut menurun}
                    while A[awal] < A[I] do
                          Inc(I);
                    While A[J] < A[awal] do
                          Dec(J);
               END;
           END;
      END;
      Tukar(A[awal],A[J]);
   END;
{Badan Program dari Quick Sort}
Begin
   IF awal < akhir then
   begin
      Atur;
Quick(A, awal, J-1  , sort);
Quick(A, J+1 , akhir, sort);
   end;
end;
```

2.22 Radix Sort

Metode Radix Sort didasarkan pada nilai sesungguhnya dari suatu digit bilangan yang akan diurutkan. Seperti kita ketahui, dalam bilangan desimal, nilai sesungguhnya dari setiap digit akan ditentukan oleh posisi dimana digit itu berada. Berikut merupakan penggalan prosedur Radix Sort dalam bahasa Pascal,

```
Procedure Radix(Var A:Data ; N,Sort:Byte);
Const
```

```
     angka=4; jml=max; jml1=max+1;
Type
    Tipe1 = 1..jml;
    Tipe2 = 0..jml1;
    Node = Record
info : Integer;
link : Tipe2;
          End;
Var
senarai : array[Tipe1] of Node;
depan : Array[0..10] of Tipe2;
belakang : array[0..9] of Tipe2;
P : Tipe1;
awal,Q,I,J : Tipe2;
Y,K,pangkat : Integer;
Begin
   {Membentuk linked-list}
   For I:=1 to N do
   Begin
        senarai[I].info:=A[I];
        senarai[I].link:=I+1;
   End;
   senarai[N].link:=0; awal:=1;
   for K:=1 to angka do
   Begin
      {Inisialisasi antrian}
      for I:=0 to 10 do depan[I]:=0;
      for I:=0 to 9 do belakang[I]:=0;
      {Memecah linked-list menjadi sejumlah antrian}
      While awal<>0 do
      begin
P := awal;
awal := senarai[awal].link;
Y := senarai[P].info;
pangkat:=1;
          for I:=1 to K-1 do
pangkat:=pangkat*10;
          {Dasar pemecahan}
J:=(Y Div pangkat) mod 10;
Q:=belakang[J];
          If Q=0 then
             depan[J]:=P
          ELSE
              senarai[Q].link:=P;
          belakang[J]:=P;
      end;
      {Menyusun kembali antrian dari kelompok 0 sampai 9}
```

```
J:=0;
      while (J<=9) and (depan[J]=0) do
            Inc(J);
awal:=depan[J];
      While J<=9 DO
      Begin
I:=J+1;
           While (I<=9) And (depan[I]=0) do
                 Inc(I);
           IF I<=9 then
           Begin
P:=I;
                senarai[belakang[J]].link:=depan[I];
           end;
J:=I;
      End;
      senarai[belakang[P]].link:=0;
   End;
   {Mengubah kembali linked-list menjadi vektor}
   Case Sort of
1 : for I:=1 to N do  {terurut menaik}
       begin
            A[I]:=senarai[awal].info;
awal:=senarai[awal].link;
       end;
2 : for I:=N Downto 1 do   {terurut menurun}
       begin
            A[I]:=senarai[awal].info;
awal:=senarai[awal].link;
       end;
   end;
end;
```

2.23 Heap Sort

Heap Sortmerupakan algoritma pengurutan tercepat setelah Merge Sort dan Quick Sort. Heap Sort menggunakan struktur pohon *heap* (*heap tree*) untuk melakukan pengurutan. Prosedur Heap Sort mengurutkan sekumpulan data pada sebuah *array* atau pohon *heap*. Cara kerjanya adalah, Heap Sort akan mengambil data pada *node* akar (*index array* = 1) dan menggantinya (*exchange*) dengan data pada

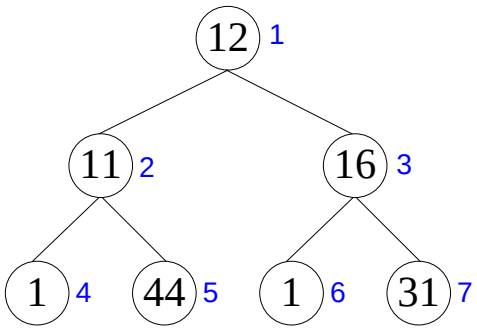
node paling akhir (*index array = index* paling maksimum dari pohon *heap*). Setelah itu, node terakhir dihapus dan Heap Sort memanggil prosedur heapify dengan tujuan agar setelah proses penggantian data, pohon masih memenuhi properti *heap*. Data-data yang dikeluarkan merupakan data yang terurut, baik menaik (*ascending*) maupun menurun (*descending*). Berikut merupakan algoritma Heap-Sort:

HEAP-SORT (A)

1. BUILD_HEAP (*A*). {prosedur untuk membangun sebuah pohon *heap* dari variabel *array* A. (Teori pohon heap dapat dilihat pada subbab 2.2.3)}
2. for *i* ← length (*A*) down to 2 do
   Get A[1] {Ambil data pada *node* pertama}
   exchange *A*[1] ↔ *A*[*i*]
   heap-size [*A*] ← heap-size [*A*] - 1
   Heapify (*A*, 1) {Prosedur ini adalah prosedur untuk mengatur penempatan data pada *array* A agar memenuhi properti heap. Lihat pada subbab 2.2.3}
3. Get A[1]

Sebagai contoh, deretan angka (A) yang akan diurut adalah: 12,11,16,1,44,1 dan 31. Properti *heap* yang digunakan adalah data pada node *parent* memiliki data terbesar. Berikut adalah proses pengurutan dengan algoritma Heap-Sort.

Data awal (sebelum diurutkan). Ukuran A = 7.

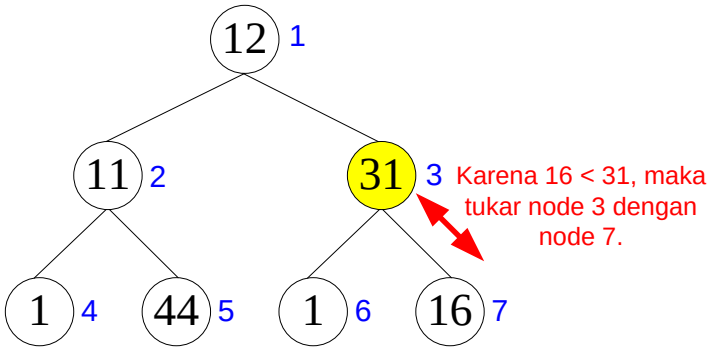


Gambar 2.24 Data awal sebelum diurutkan

2.24 BUILD_HEAP (A)

Lakukan prosedur heapifyterhadap setiap *node* induk (dari indeks 3 ke indeks 1).

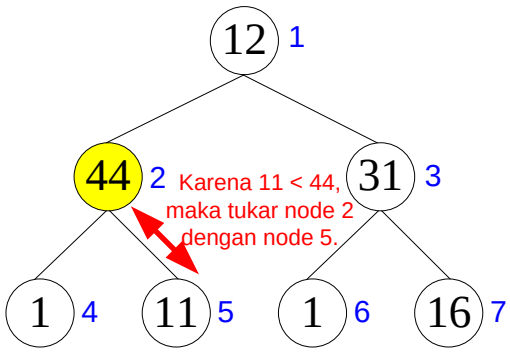
Prosedur Heapify(A, 3)



Gambar 2.25 Proses penukaran angka pada *node* 3 dan *node* 7

Proses Heapify berlanjut ke *node*7 (node yang ditukar dengan *node* 3), tetapi karena *node* 7 tidak memiliki anak, maka struktur pohon tidak berubah.

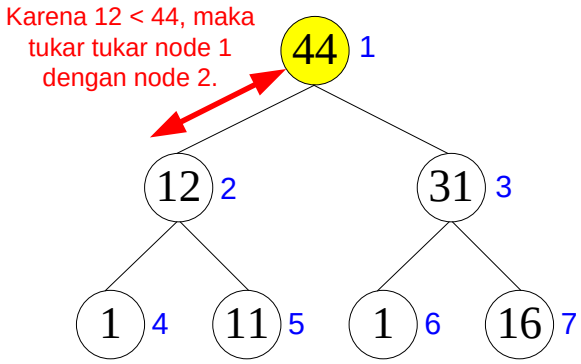
Prosedur Heapify(A, 2)



Gambar 2.26 Proses penukaran angka pada *node*2 dan *node*5

Proses Heapify berlanjut ke *node* 5 (node yang ditukar dengan *node* 2), tetapi karena *node* 5 tidak memiliki anak, maka struktur pohon tidak berubah.

Prosedur Heapify(A, 1)



Gambar 2.27 Proses penukaran angka pada *node*1 dan *node*2

Proses Heapify berlanjut pada *node* 2 (*node* yang ditukar dengan *node* 1), tetapi karena *node* 2 memiliki data terbesar dibandingkan dengan node anaknya, maka struktur pohon tidak berubah dan prosedur Heapify berakhir.

Pohon *heap* telah terbentuk dari data *array* A seperti terlihat pada gambar 2.27 atau dapat ditulis dengan: [44, 12, 31, 1, 11, 1, 16].

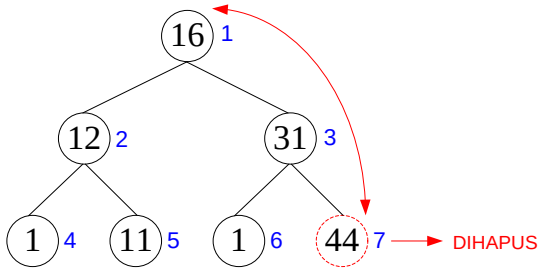
0. HEAP-SORT

Pengurutan dengan *heap-sort* mengambil data dari pohon *heap* satu per satu. Data yang diambil adalah data pada *node* 1, data pada *node* 1 kemudian ditukarkan dengan data pada *node* terakhir dan *node* terakhir dihapus.

a. Ambil angka ‘44’ pada *node* 1, sehingga deretan angka menjadi:

[12, 31, 1, 11, 1, 16], 44

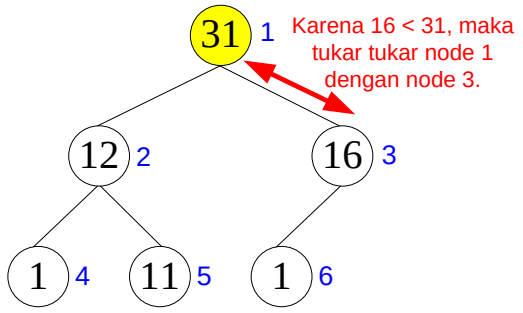
Tukar *node* 1 dengan *node*7 (*node* terakhir) dan hapus*node* terakhir.



Gambar 2.28 Proses penukaran angka pada *node* 1 dan *node* 7 dihapus

Kemudian lakukan proses Heapify(A, 1).

Prosedur Heapify(A,1)



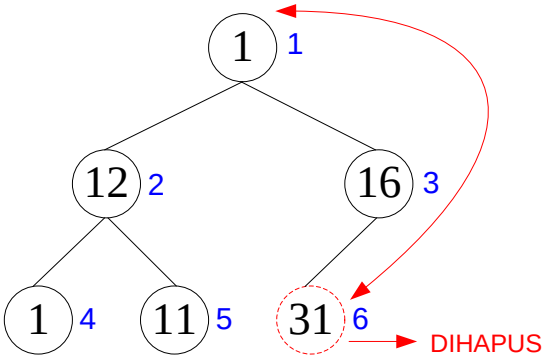
Gambar 2.29 Proses penukaran angka pada *node* 1 dengan *node* 3

Proses Heapify berlanjut pada *node* 3 (*node* yang ditukar dengan *node* 1), tetapi karena *node* 3 memiliki data terbesar dibandingkan dengan *node* anaknya, maka struktur pohon tidak berubah dan prosedur Heapify berakhir.

b. Ambil angka '31' pada *node* 1, sehingga deretan angka menjadi:

[12, 1, 11, 1, 16], 31, 44

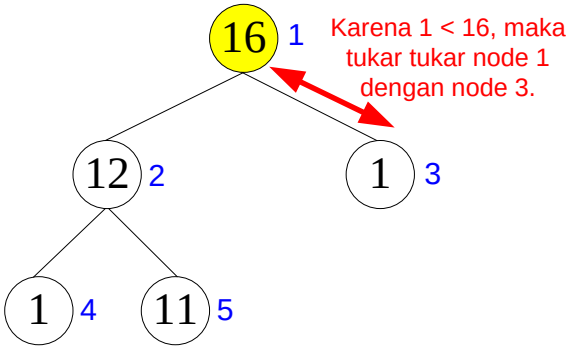
Tukar *node* 1 dengan *node* 6 (*node* terakhir) dan hapus *node* terakhir.



Gambar 2.30 Proses penukaran angka pada *node* 1 dan *node* 6 dihapus

Kemudian lakukan proses Heapify(A, 1).

Prosedur Heapify(A,1)



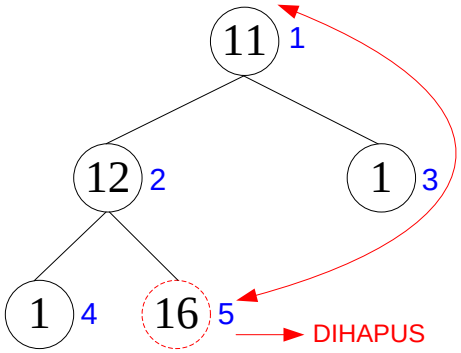
Gambar 2.31 Proses penukaran angka pada *node* 1 dengan *node* 3

Proses Heapify berlanjut pada *node* 3 (*node* yang ditukar dengan *node* 1), tetapi karena *node* 3 tidak memiliki anak, maka struktur pohon tidak berubah dan prosedur Heapify berakhir.

c. Ambil angka ‘16’ pada *node* 1, sehingga deretan angka menjadi:

[12, 1, 11, 1], 16, 31, 44

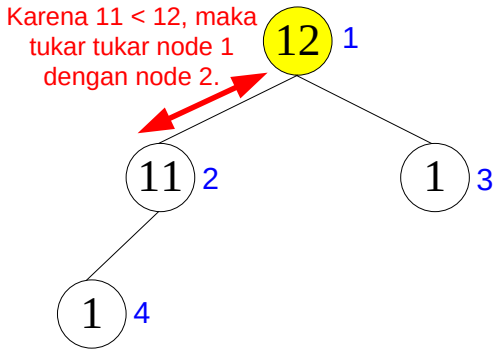
Tukar *node* 1 dengan *node* 5 (*node* terakhir) dan hapus *node* terakhir.



Gambar 2.32 Proses penukaran angka pada *node* 1 dan *node* 5 dihapus

Kemudian lakukan proses Heapify(A, 1).

Prosedur Heapify(A,1)



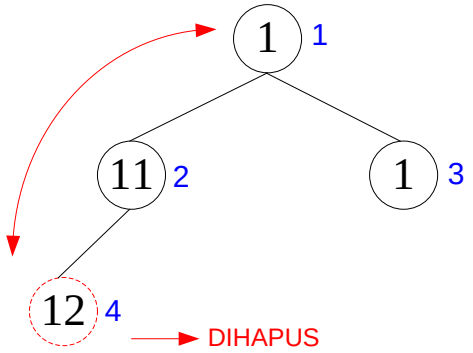
Gambar 2.33 Proses penukaran angka pada *node* 1 dengan *node* 2

Proses Heapify berlanjut pada *node*2 (*node* yang ditukar dengan *node* 1), tetapi karena *node*2 tidak memiliki anak, maka struktur pohon tidak berubah dan prosedur Heapify berakhir.

d. Ambil angka ‘12’ pada *node* 1, sehingga deretan angka menjadi:

[1, 11, 1], 12, 16, 31, 44

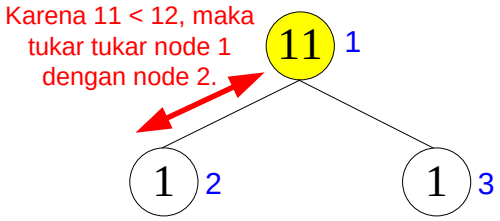
Tukar *node* 1 dengan *node* 4 (*node* terakhir) dan hapus *node* terakhir.



Gambar 2.34 Proses penukaran angka pada *node* 1 dan *node* 4 dihapus

Kemudian lakukan proses Heapify(A, 1).

Prosedur Heapify(A,1)



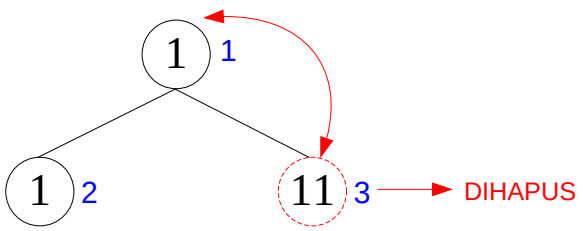
Gambar 2.35 Proses penukaran angka pada *node* 1 dengan *node* 2

Proses Heapify berlanjut pada *node*2 (*node* yang ditukar dengan *node* 1), tetapi karena *node*2 tidak memiliki anak, maka struktur pohon tidak berubah dan prosedur Heapify berakhir.

e. Ambil angka ‘11’ pada *node* 1, sehingga deretan angka menjadi:

[1, 1], 11, 12, 16, 31, 44

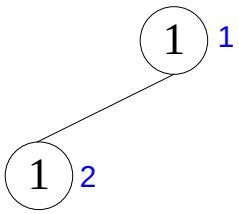
Tukar *node* 1 dengan *node* 3 (*node* terakhir) dan hapus *node* terakhir.



Gambar 2.36 Proses penukaran angka pada *node* 1 dan *node* 3 dihapus

Kemudian lakukan proses Heapify(A, 1).

Prosedur Heapify(A,1)



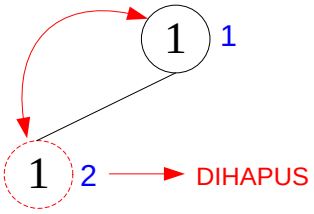
Gambar 2.37 Proses penukaran angka pada *node* 1 dengan *node* 2

Proses Heapify tidak mengubah struktur pohon karena *node parent* tidak lebih besar daripada *node* anak.

f. Ambil angka ‘1’ pada *node* 1, sehingga deretan angka menjadi:

[1], 1, 11, 12, 16, 31, 44

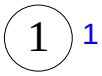
Tukar *node* 1 dengan *node* 2 (*node* terakhir) dan hapus *node* terakhir.



Gambar 2.38 Proses penukaran angka pada *node* 1 dan *node* 2 dihapus

Kemudian lakukan proses Heapify(A, 1).

Prosedur Heapify(A,1)



Gambar 2.39 Proses penukaran angka pada *node* 1 dengan *node* 2

Proses Heapify tidak mengubah struktur pohon karena *node* 1 tidak memiliki anak lagi.

g. Ambil angka '1' pada *node* 1, sehingga deretan angka menjadi:

1, 1, 11, 12, 16, 31, 44

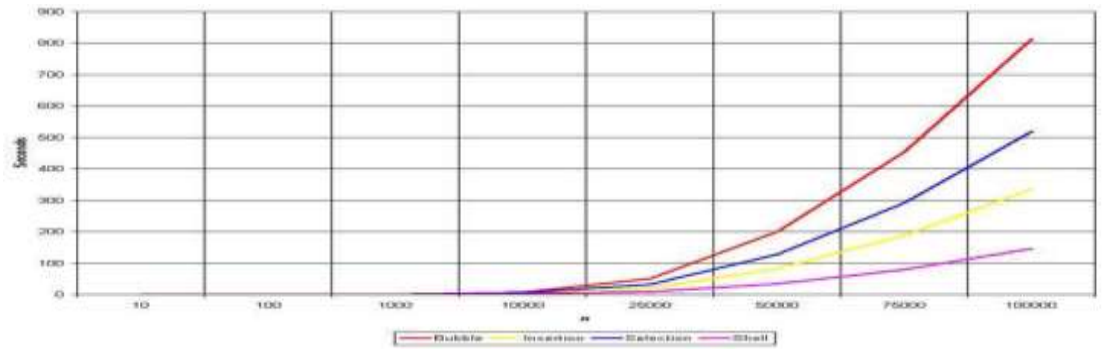
Hasil akhir adalah berupa deretan angka terurut secara menaik (*ascending*). Untuk angka terurut menurun (*descending*), tempatkan angka yang dikeluarkan dari pohon *heap* di depan deretan.

2.25 Kompleksitas Algoritma Pengurutan

Berdasarkan kompleksitas algoritma (waktu eksekusi algoritma) untuk melakukan proses pengurutan, algoritma pengurutan dapat dibagi menjadi 2 bagian, yaitu:

1. Algoritma pengurutan yang memiliki kompleksitas O(n^2), terdiri atas Bubble Sort, Insertion Sort, Selection Sort dan Shell Sort.
2. Algoritma pengurutan yang memiliki kompleksitas O(n log n), terdiri atas Heap Sort, Merge Sort dan Quick Sort. Algoritma pengurutan ini lebih cepat dibandingkan algoritma pengurutan dengan kompleksitas O(n^2).

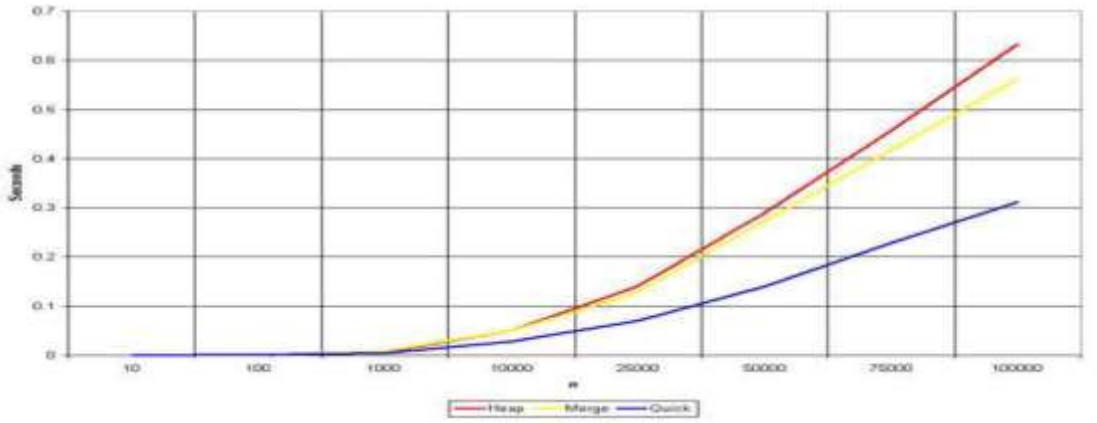
Grafik perbandingan banyak data yang diurutkan dan waktu yang dibutuhkan algoritma pengurutan dengan kompleksitas O(n^2) dapat dilihat pada gambar 2.40 berikut.



Gambar 2.40 Grafik efisiensi algoritma pengurutan dengan kompleksitas O(n^2)

Dalam kondisi terbaik (*best-case conditions*), Bubble Sort dapat mencapai kompleksitas O(n). Tetapi secara umum, kompleksitasnya adalah O(n^2). Walaupun Insertion, Selection dan Shell Sort juga memiliki kompleksitas O (n^2), namun ketiga algoritma ini jauh lebih efisien dibandingkan dengan Bubble Sort. Selection Sort memiliki performa 60% lebih baik daripada Bubble Sort, Insertion Sort memiliki performa dua kali lebih daripada Bubble Sort, sedangkan Shell Sort adalah algoritma pengurutan tercepat di kelasnya (algoritma dengan kompleksitas O(n^2)).

Grafik perbandingan banyak data yang diurutkan dan waktu yang dibutuhkan algoritma pengurutan dengan kompleksitas O(*n*log *n*) dapat dilihat pada gambar 2.41 berikut.



Gambar 2.41 Grafik efisiensi algoritma pengurutan dengan kompleksitas O(*n*log *n*)

Heap Sort adalah algoritma pengurutan terlambat di kelasnya (algoritma dengan kompleksitas O(*n* log *n*)). Akan tetapi, tidak seperti Merge Sort dan Quick Sort, Heap Sort tidak membutuhkan jumlah *array* yang besar dan rekursif yang berlebihan. Merge Sort lebih cepat dari Heap Sort, tetapi Merge Sort membutuhkan dua kali jumlah memori lebih banyak dari Heap Sort karena *array* keduanya. Quick Sort adalah algoritma pengurutan tercepat di kelas O(*n* log *n*) walaupun membutuhkan rekursif yang sangat berlebihan.

BAB III
METODE PENELITIAN

3.1 Metode penelitian

Jenis penelitian yang digunakan dalam penelitian ini adalah penelitian terapan. Penelitian terapan adalah penyelidikan yang hati-hati, sistematik dan terus menerus terhadap suatu masalah dengan tujuan untuk digunakan dengan segera untuk keperluan tertentu

Teknik pengumpulan data pada penelitian terapan ini menggunakan teknik studi pustaka (Library research). yaitu dengan mempelajari konsep-konsep dasar mengenai heapsort yang terdapat pada beberapa sumber literatur. Sumber literatur dapat berupa buku teks, paper, website, blog, laporan penelitian, karangan-karangan ilmiah, tesis dan disertasi dan sumber-sumber tertulis baik tercetak maupun elektronik yang berhubungan dengan penelitian.

3.2 Analisis

Pembuatan perangkat lunak bantu pemahaman Heap Sort ini mencakup beberapa bagian penting, yaitu:

1. Cara memasukkan *input* ke dalam perangkat lunak.
2. Cara kerja perangkat lunak.
3. Proses penggambaran pohon biner.
4. Proses pengurutan Heap Sort, mencakup 3 (tiga) buah prosedur, yaitu:

a. Prosedur Heapify.

b. Prosedur Build-Heap.

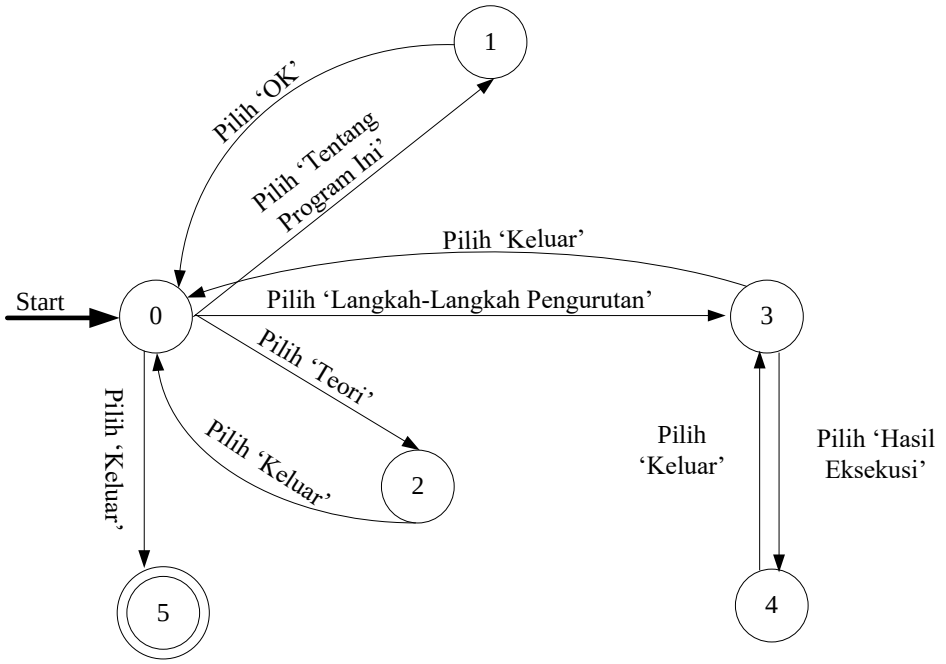
c. Prosedur HeapSort.

3.3 Cara Memasukkan Input ke dalam Perangkat Lunak

Cara memasukkan *input* ke dalam perangkat lunak adalah barisan data yang akan di-*input* harus dipisahkan dengan huruf koma di antara data yang satu dengan yang lain. Misalkan barisan data yang akan di-*input* adalah: data[1] = 55, data[2] = 25, data[3] = 20, data[4] = 15, data[5] = 60 dan seterusnya. Maka barisan data dikonversi ke dalam bentuk *string* dengan pemisah berupa huruf koma antar data, sehingga *input* data adalah: 55,25,20,15,60 .... dan seterusnya.

3.4 Cara Kerja Perangkat Lunak

Perangkat lunak dimulai dari Form Main. Pada *form* ini, *user* dapat meng-*input* barisan angka yang akan diurutkan atau menghasilkan barisan angka secara acak. Setelah itu, tekan tombol ’Langkah-Langkah Pengurutan’ akan membuka Form Pengurutan. Pada *form* ini, perangkat lunak akan menjelaskan dan menampilkan proses kerja algoritma Heap Sort dalam melakukan pengurutan. Proses pengurutan dapat dihentikan sementara (*pause*) dan dilanjutkan kembali (*resume*). Cara kerja perangkat lunak dapat dilihat pada gambar 3.1.



Gambar 3.1 Cara Kerja Perangkat Lunak

Keterangan:

0 : Form Main.

1 : Form About.

2 : Form Teori.

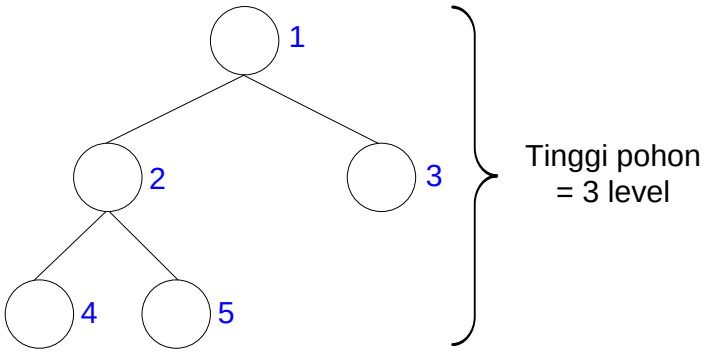
3 : Form Pengurutan.

4 : Form Hasil Eksekusi.

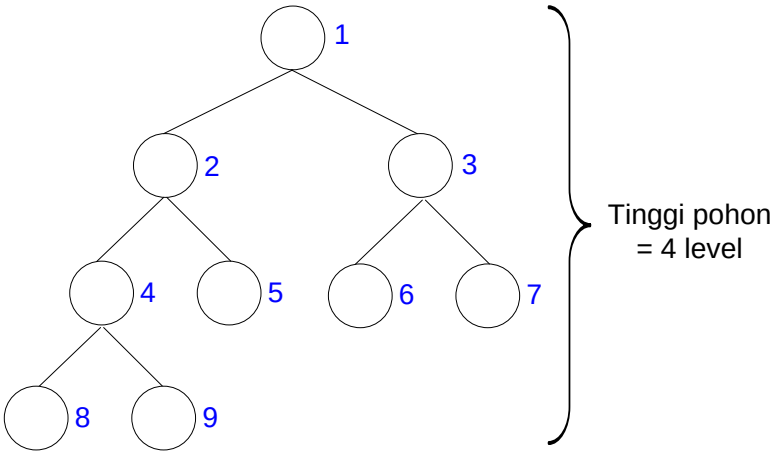
5 : *Operating Systems (Windows).*

3.5 Proses Penggambaran Pohon Biner

Proses ini menerima *input* sebuah *list array* A. *List array* A adalah sederetan angka yang akan dimasukkan ke dalam struktur pohon. Untuk menggambar pohon biner, pertama-tama tentukan terlebih dahulu tinggi (level) pohon biner yang dapat dibangun untuk menampung jumlah data *array* A. Misalkan, jumlah data *array* A = 5 buah, maka diperlukan pohon biner dengan tinggi = 3 level (dapat dilihat pada gambar 3.2). Untuk jumlah data *array* A = 9 buah, diperlukan pohon biner dengan tinggi = 4 level. (dapat dilihat pada gambar 3.3)



Gambar 3.2 Pohon dengan tinggi 3 level



Gambar 3.3 Pohon dengan tinggi 4 level

Cara menghitung tinggi pohon biner adalah ambil nilai indeks *array* tertinggi (*x*) dari A, kemudian lakukan *looping* dengan menghitung ’*floor*(*x* / 2)’. *Floor*(*x* / 2) adalah fungsi untuk menghitung indeks *node parent* dari *node x*. *Looping* dilakukan hingga nilai *x* = 1. Ini artinya *x* telah berada pada *node* akar, dan (banyak *looping* + 1) adalah tinggi pohon biner yang diperlukan untuk menampung data *array* A.

Setelah mendapatkan tinggi pohon biner, maka lakukan penggambaran pohon biner dari level paling bawah ke level 1. Pada level yang sama, setiap *node* yang berdampingan memiliki perbedaan interval posisi *x* (horizontal) sebesar 2 (nilai *y* sama), sedangkan untuk satu level yang berbeda, *node-node* memiliki perbedaan interval posisi *y* (vertikal) sebesar 1. Sebagai contoh, jumlah data = 11 buah maka proses penggambarannya adalah sebagai berikut:

1. Tentukan tinggi pohon biner.

   x = jumlah data = 11. Lakukan looping hingga floor(x / 2) = 1.

   i = 1 → x = floor(x / 2) = floor(11 / 2) = 5.

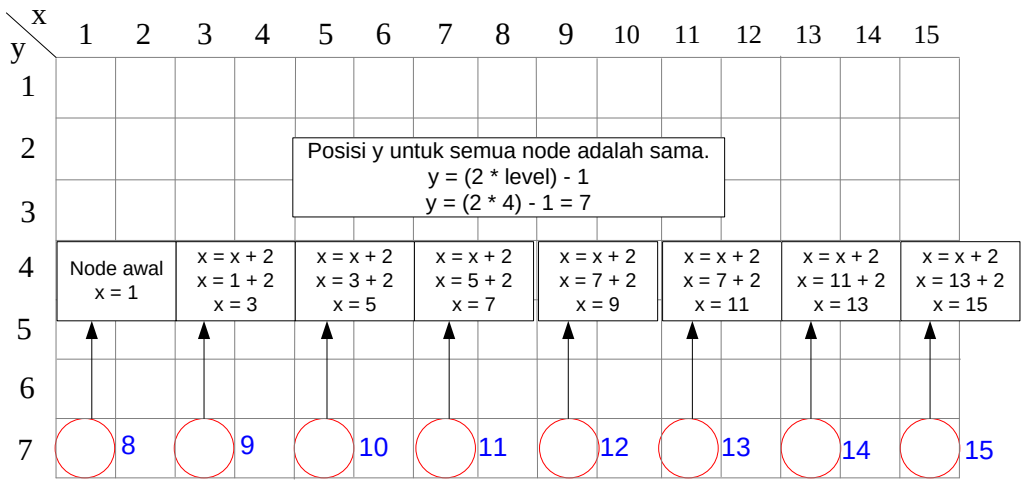
   i = 2 → x = floor(x / 2) = floor(5 / 2) = 2.

   i = 3 → x = floor(x / 2) = floor(2 / 2) = 1.

   Looping terjadi 3 kali, sehingga tinggi pohon adalah = (i + 1) = (3 + 1) = 4 level.

2. Penggambaran pohon dimulai dari level paling bawah (level 4).

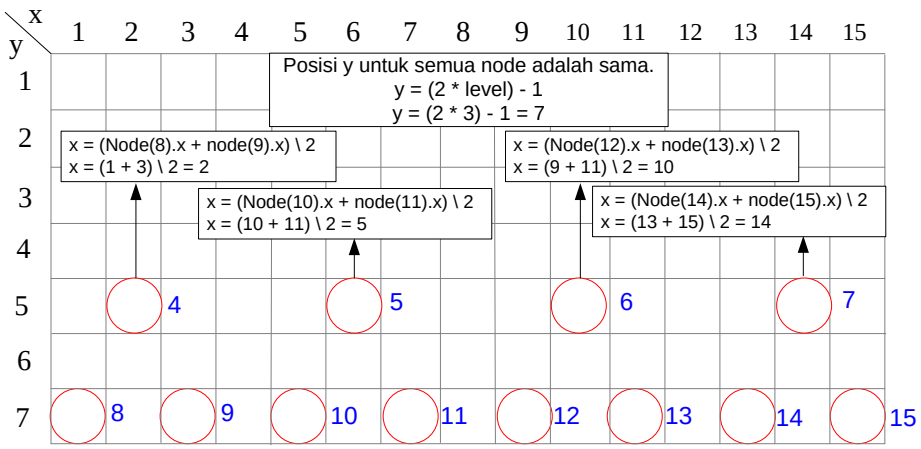
Untuk setiap node yang berdampingan, selisih posisi *x* (horizontal) adalah 2 dan nilai *y* sama, dihitung dengan rumus = (2*Level) – 1. Prosesnya dapat dilihat pada gambar 3.4 berikut.



Gambar 3.4 Penggambaran *node* level 4

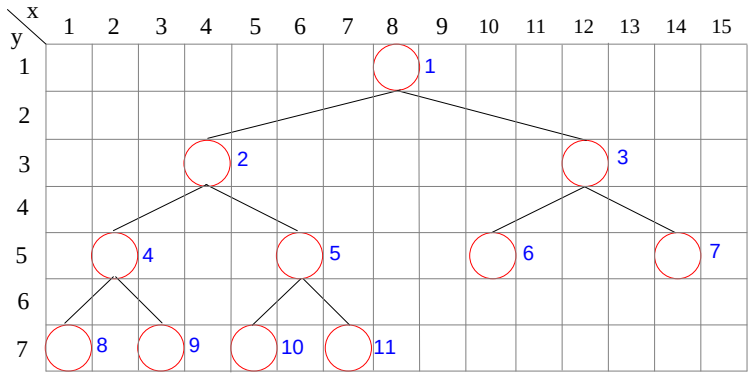
3. Penggambaran pohon dilanjutkan untuk level berikutnya (level 3, 2 dan 1).

   Untuk setiap *node* yang berdampingan, selisih posisi *x* (horizontal) adalah 2 dan nilai *y* sama, dihitung dengan rumus = (posisi *x* dari node anak1) + (posisi *x* dari node anak2) \ 2. Prosesnya dapat dilihat pada gambar 3.5 berikut.



Gambar 3.5 Penggambaran *node* level 3

Proses yang sama dilakukan juga untuk level 1 dan level 2. Setelah penggambaran node selesai, hubungkan setiap *node parent* dan *node* anak dengan garis, dan hilangkan *node* dengan indeks yang lebih besar dari jumlah data. Prosesnya dapat dilihat pada gambar 3.6 berikut.



Gambar 3.6 Gambar pohon biner untuk jumlah data = 11 buah

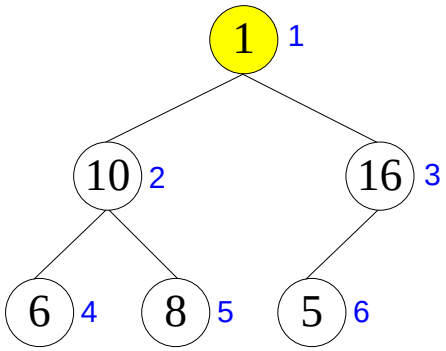
3.6 Proses Pengurutan Heap Sort

Proses pengurutan Heap Sort menggunakan struktur pohon *heap* dan prosedur dasar dari pohon tersebut. Berikut pembahasan mengenai prosedur-prosedur dasar dalam Heap Sort:

1. Prosedur Heapify (A, *i*).

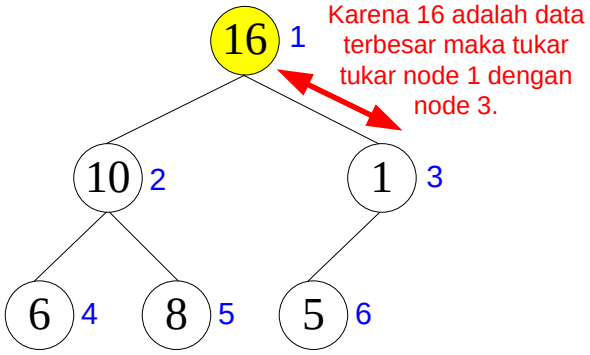
   Prosedur ini berfungsi untuk mengatur penempatan data pada pohon agar memenuhi properti *heap* (*node parent* memiliki data terbesar / terkecil / sama dengan data pada *node* kedua anaknya). Misalkan, pada properti *heap* yang mengharuskan *node parent* memiliki data terbesar, cara kerjanya adalah periksa

node ke-*i* dan kedua *node* anaknya. Apabila ada data yang lebih besar pada *node* anak, maka tukarkan data pada *node* anak tersebut dengan *node parent*. Setelah pertukaran data, lakukan juga prosedur Heapify untuk *node* anak yang ditukarkan datanya dengan *node* induk. Sebagai contoh, lakukan prosedur Heapify(A, 1) terhadap pohon pada gambar 3.7.



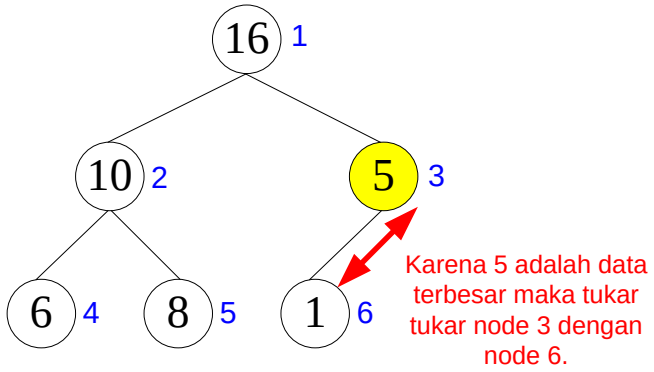
Gambar 3.7 Contoh pohon biner dengan 6 *node*

Prosedur Heapify(A, 1): Periksa *node* 1, *node* 3 memiliki data terbesar. Oleh karena itu tukarkan data pada *node* 1 dan *node* 3. Selanjutnya lakukan prosedur Heapify pada *node* 3 (*node* yang ditukarkan dengan node *parent*).



Gambar 3.8 Prosedur Heapify(A,1) pada pohon biner pada gambar 3.7

Prosedur Heapify(A, 3): Periksa *node* 3, *node* 6 memiliki data terbesar. Oleh karena itu tukarkan data pada *node* 6 dan *node* 3. Selanjutnya lakukan prosedur Heapify pada *node* 6 (*node* yang ditukarkan dengan node *parent*).



Gambar 3.9 Prosedur Heapify(A,3) pada pohon biner pada gambar 3.8

Prosedur Heapify(A, 6): *Node* 6 tidak memiliki *node* anak, sehingga struktur pohon tidak berubah.

2. Prosedur Build-Heap (A).

   Prosedur Build-Heap(A) membangun struktur pohon *heap* dari data pada *list array* A. Prosedur Build-Heap melakukan prosedur Heapify untuk semua *node* induk dimulai dari node induk paling akhir (dihitung dengan rumus floor(length[A]/2)). Secara algoritmik dapat ditulis:

   Untuk i = floor(length[A]/2) sampai 1, lakukan

   Heapify(A, *i*)

   (Prosedur Heapify adalah sama dengan pembahasan pada poin pertama di atas).

3. Prosedur Heap-Sort (A).

Prosedur Heap Sort mengurutkan sekumpulan data pada list *array* A. Cara kerjanya adalah, Heap Sort akan mengambil data pada *node* akar (*index array* = 1) dan menggantinya (*exchange*) dengan data pada *node* paling akhir (*index array* = *index* paling maksimum dari pohon *heap*). Setelah itu, node terakhir dihapus dan Heap Sort memanggil prosedur *heapify* untuk *node* 1 dengan tujuan agar setelah proses penggantian data, pohon masih memenuhi properti *heap*. Data-data yang dikeluarkan merupakan data yang terurut, baik menaik (*ascending*) maupun menurun (*descending*). Secara algoritmik dapat ditulis:

Lakukan BUILD-HEAP (A).

Untuk i = length(A) sampai 2, lakukan

Ambil nilai A[1]

Tukarkan A[1] dengan A[*i*]

Hapus *node* terakhir.

Lakukan Heapify(A, 1).

Ambil nilai A[1].

Contoh penggunaan prosedur Heap Sort dapat dilihat pada subbab 2.4.7.

3.7 Perancangan

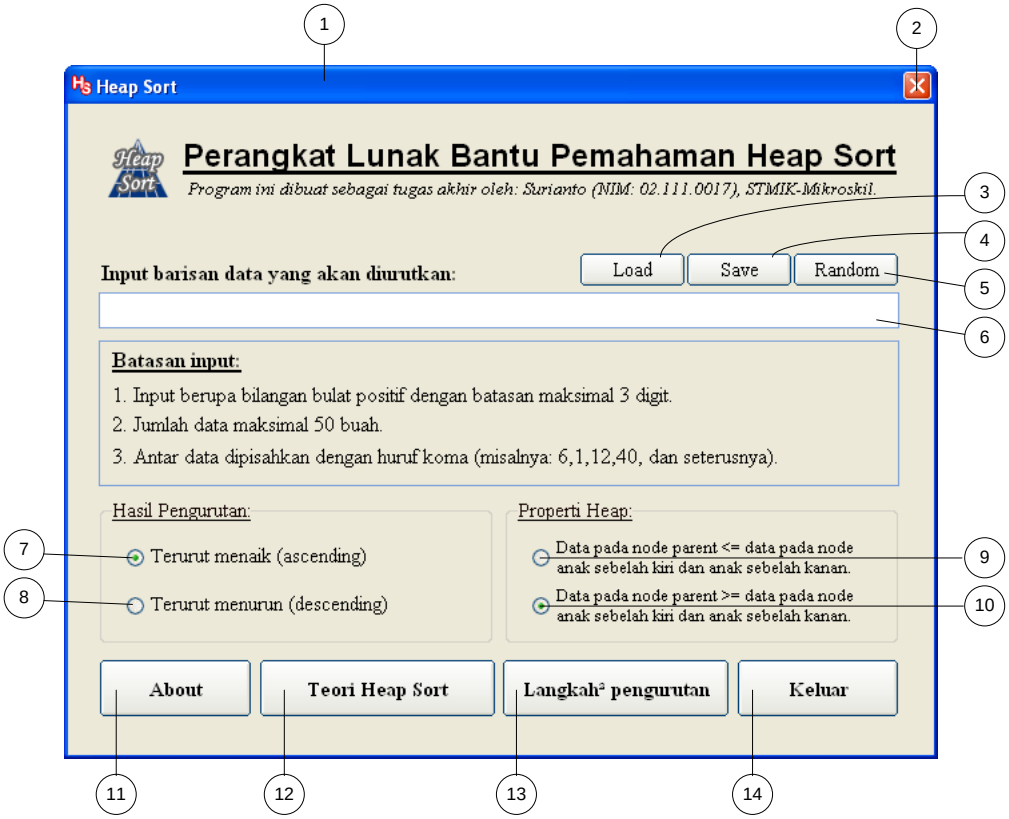
Perancangan perangkat lunak bantu pemahaman Heap Sort menggunakan bahasa pemograman *Microsoft Visual Basic* 6.0. Perangkat lunak memiliki beberapa *form*, antara lain:

1. *Form Main*.
2. *Form* Pengurutan.
3. *Form* Hasil Eksekusi.
4. *Form* Teori.
5. *Form About*.

Perangkat lunak menggunakan komponen *visual basic*, seperti *commandbutton* sebagai tombol, *textbox* sebagai tempat *input*, *optionbutton* sebagai opsi pilihan, *label* untuk menampilkan tulisan, *line* sebagai objek bentuk garis dan *MSFlexGrid*, sebagai tabel, dan komponen lainnya.

3.8 Form Main

Form ini merupakan *form* utama dan berfungsi sebagai *form* untuk memasukkan deretan angka yang ingin diurutkan, memilih hasil pengurutan (*ascending* / *descending*) dan memilih properti *heap* yang diinginkan.



Gambar 3.10 Rancangan Form Main

Keterangan:

1 : *title bar*.

2 : tombol ’Close’, berfungsi untuk menutup *form*.

3 : tombol ’Load’, berfungsi untuk membuka *file input*.

4 : tombol ’Save’, berfungsi untuk menyimpan *input* ke dalam bentuk *file*.

5 : tombol ’Random’, berfungsi untuk menghasilkan barisan data secara acak.

6 : *textbox*, berfungsi untuk memasukkan barisan angka yang dipisahkan dengan huruf koma satu dengan yang lain.

7 : *optionbutton*, berfungsi untuk memilih hasil pengurutan terurut menaik

(*ascending*).

8 : *optionbutton*, berfungsi untuk memilih hasil pengurutan terurut menurun (*descending*).

9 : *optionbutton*, berfungsi untuk memilih properti *heap*, data pada *node parent* adalah data yang terbesar.

10 : *optionbutton*, berfungsi untuk memilih properti *heap*, data pada *node parent* adalah data yang terkecil.

11 : tombol ’Tentang Program Ini’, berfungsi untuk menampilkan *form About*.

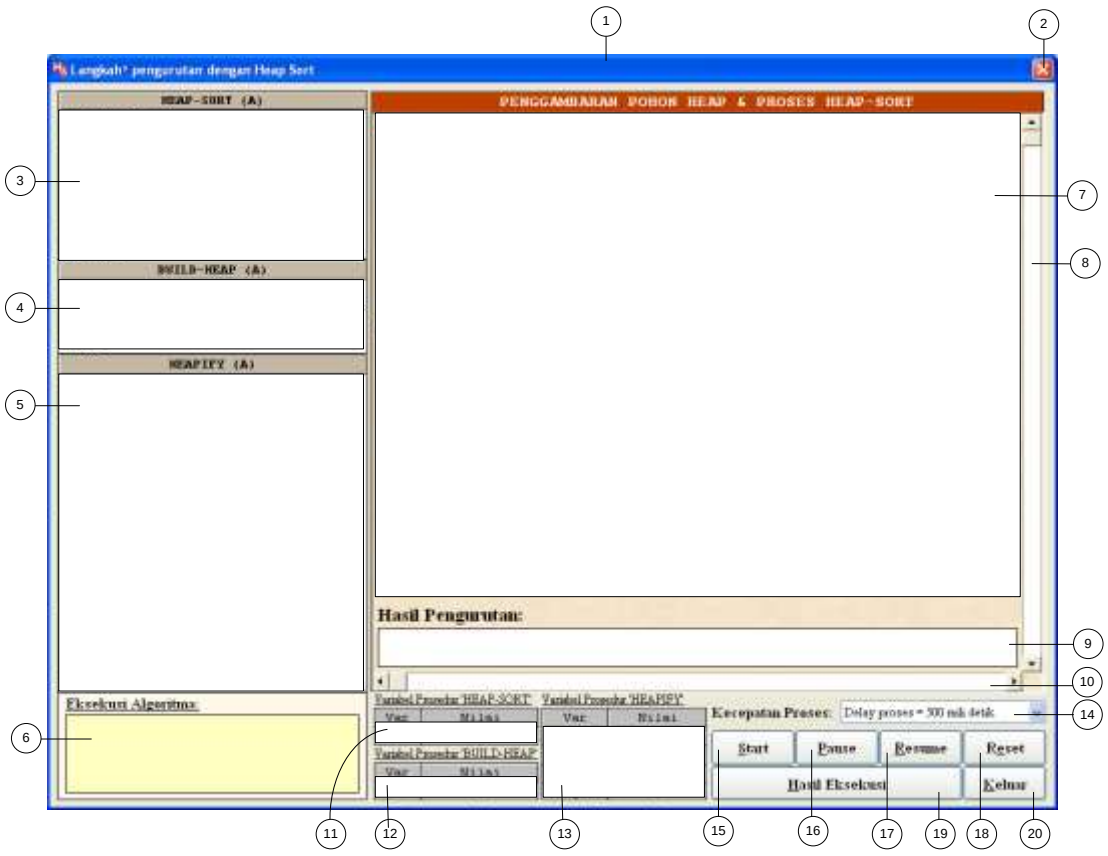
12 : tombol ’Teori’, berfungsi untuk menampilkan *form* Teori.

13 : tombol ’Langkah² Pengaturan’, berfungsi untuk menampilkan *form* Pengurutan.

14 : tombol ’Keluar’, berfungsi untuk keluar dari perangkat lunak.

3.9 Form Pengurutan

Form Pengurutan berfungsi untuk menampilkan proses pengurutan data dengan Heap Sort.



Gambar 3.11 Rancangan Form Pengurutan

Keterangan:

1 : *title bar*.

2 : tombol ’Close’, berfungsi untuk menutup *form*.

3 : label agoritma prosedur Heap-Sort.

4 : label agoritma prosedur Build-Heap.

5 : label agoritma prosedur Heapify.

6 : label, untuk menampilkan eksekusi algoritma.

7 : daerah penggambaran pohon.

8 : *vscrollbar*, untuk menggulung daerah penggambaran pohon secara vertikal.

9 : *textbox*, untuk menampilkan hasil pengurutan.

10 : *hscrollbar*, untuk menggulung daerah penggambaran pohon secara horizontal.

11 : tabel variabel prosedur Heap-Sort.

12 : tabel variabel prosedur Build-Heap.

13 : tabel variabel prosedur Heapify.

14 : *combobox*, untuk memilih kecepatan proses.

15 : tombol ’Start’, untuk memulai proses pengurutan.

16 : tombol ’Pause’, untuk menghentikan proses pengurutan untuk sementara.

17 : tombol ’Resume’, untuk melanjutkan proses pengurutan setelah dihentikan untuk sementara (*pause*).

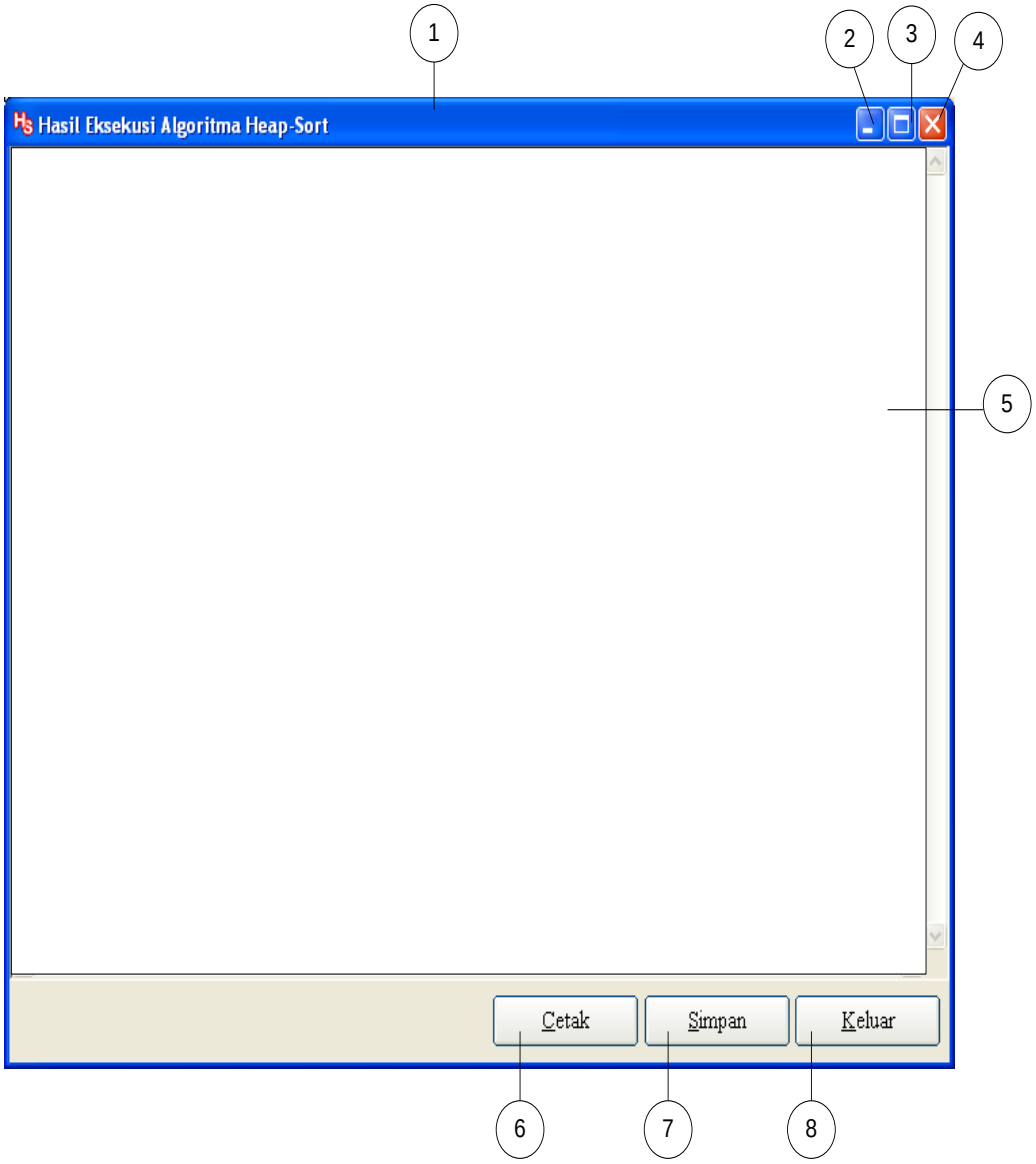
18 : tombol ’Reset’, untuk mengulangi proses pengurutan dari awal.

19 : tombol ’Hasil Eksekusi’, untuk melihat hasil eksekusi algoritma yang ditampilkan pada *form* Hasil.

20 : tombol ’Keluar’, untuk menutup *form*.

3.10 Form Hasil Eksekusi

Form Hasil Eksekusi berfungsi untuk menampilkan hasil eksekusi dari algoritma pengurutan Heap Sort.



Gambar 3.12 Rancangan tampilan Form Hasil Eksekusi

Keterangan:

1 : *title bar*.

2 : tombol ’Minimize’, berfungsi untuk me-*minimize form*.

3 : tombol ’Maximize’, berfungsi untuk mengubah ukuran *form* ke ukuran maksimal.

4 : tombol ’Close’, berfungsi untuk menutup *form*.

5 : *textbox*, untuk menampilkan hasil eksekusi algoritma.

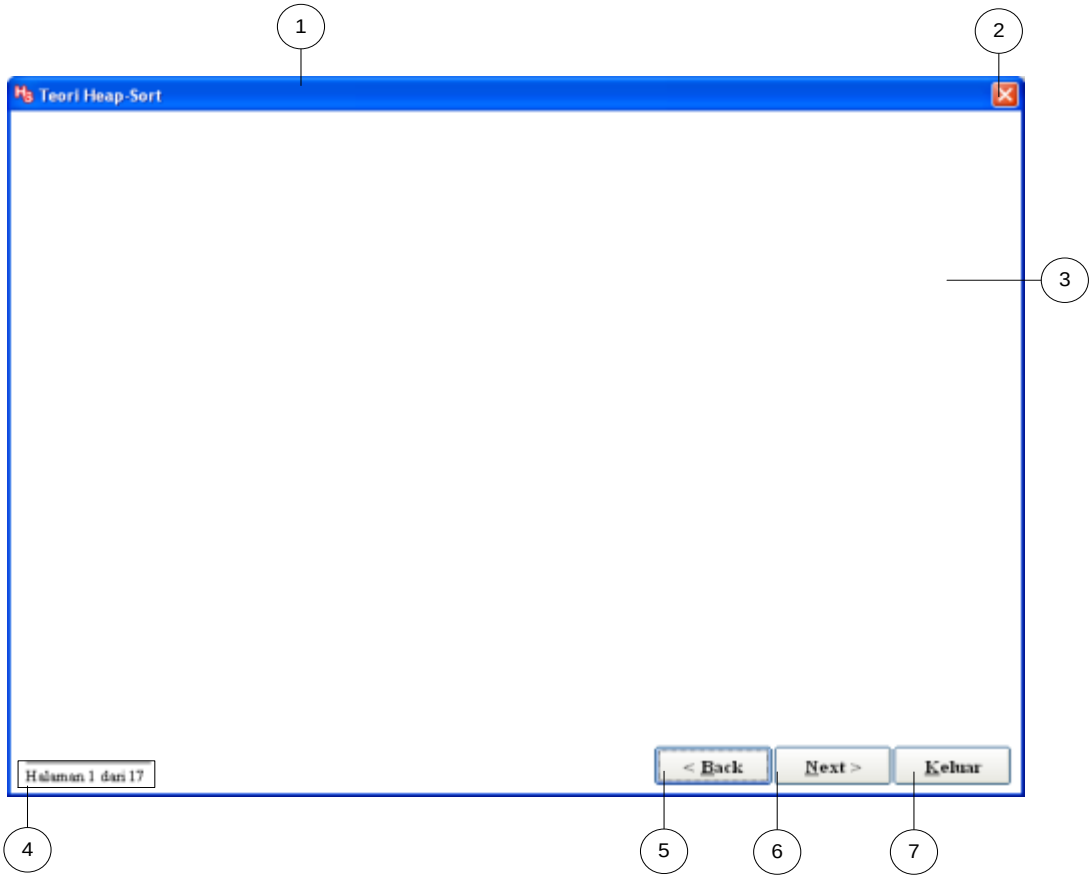
6 : tombol ’Cetak’, untuk mencetak hasil eksekusi ke *printer*.

7 : tombol ’Simpan’, untuk menyimpan hasil eksekusi ke *text file* (*.txt).

8 : tombol ’Keluar’, untuk menutup *form*.

3.11 Form Teori

Form Teori berfungsi untuk menampilkan teori-teori dasar mengenai algoritma pengurutan Heap Sort.



Gambar 3.13 Rancangan tampilan Form Teori

Keterangan:

1 : *title bar*.

2 : tombol ’Close’, berfungsi untuk menutup *form*.

3 : daerah tampilan teori.

4 : label, untuk menampilkan halaman.

5 : tombol ’Back’, untuk kembali ke halaman sebelumnya.

6 : tombol ’Next’, untuk membuka halaman berikutnya.

7 : tombol ’Keluar’, untuk menutup *form*.

BAB IV
ALGORITMA DAN IMPLEMENTASI

4.1 Algoritma

Algoritma untuk merancang perangkat lunak bantu pemahaman Heap Sort terbagi menjadi 6 (enam) bagian, yaitu:

1. Algoritma Penggambaran Pohon Biner.
2. Algoritma Heapify.
3. Algoritma Build-Heap.
4. Algoritma Heap-Sort.
5. Algoritma Fungsi-Fungsi Pendukung.

4.2 Algoritma Penggambaran Pohon Biner

Algoritma ini berfungsi untuk menggambar struktur pohon biner dari sebuah kumpulan data *array*. Algoritma penggambaran pohon biner adalah sebagai berikut:

1. Set nLevel = GetTreeLevel. Fungsi GetTreeLevel adalah fungsi yang menghitung tinggi (level) pohon biner yang harus dibangun untuk menampung jumlah data dalam *array*.
2. Redim Node((2 ^ nLevel) – 1). Bentuk variabel ‘Node’ dengan ukuran *array* = (2^nLevel – 1).
3. Load objek yang digunakan untuk menggambar pohon. Untuk X = 1 sampai ((2 ^ nLevel) - 1), lakukan algoritma berikut.

a. Load imgNode(X). Objek ‘imgNode’ adalah gambar lingkaran yang dijadikan sebagai *node*.
b. Load lblIsi(X). Objek ‘lblIsi’ adalah label yang menampilkan angka atau isi dari *node*.
c. Load lblIndex(X). Objek ‘lblIndex’ adalah label yang menampilkan indeks dari *node*.
d. Load Line1(X). Objek ‘Line1’ adalah garis yang digunakan sebagai penghubung antar *node*.

4. Set nLevelT = nLevel + 1.
5. Selama nilai nLevelT <> 1, lakukan algoritma berikut.
    a. Set nLevelT = nLevelT – 1.
    b. Atur letak node dari kiri ke kanan pada tingkatan nLevelT. Untuk X = (2^ (nLevelT – 1)) sampai ((2 ^ nLevelT) - 1), lakukan algoritma berikut.
        i. Jika nLevelT = nLevel (level terbawah), maka:
            a) Jika X = (2 ^ (nLevelT - 1)), maka set Node(X).X = 1 dan Node(X).Y = (2 * nLevelT) – 1.
            b) Jika tidak, maka set Node(X).X = Node(X - 1).X + 2 dan Node(X).Y = Node(X - 1).Y.
        ii. Jika tidak, maka:
            a) Set nLeft = LeftNode(X). Indeks array *node* anak sebelah kiri.

b) Set nRight = RightNode(X). Indeks array *node* anak sebelah kanan.

c) Set Node(X).X = (Node(nLeft).X + Node(nRight).X) \ 2.

d) Set Node(X).Y = (2 * nLevelT) – 1.

6. Untuk X = 1 sampai ((2 ^ nLevel) - 1), lakukan algoritma berikut.
   a. Set lblIndex(X) = X.
   b. Jika X <= UBound(HeapArray) maka set lblIsi(X) = HeapArray(X).
   c. Gambar node dengan memangil prosedur AturLetakNode(imgNode(X), lblIndex(X), lblIsi(X), Node(X).X, Node(X).Y).
   d. Jika X > 1, maka gambar garis dengan memanggil prosedur Call AturLetakGaris(Line1(X), imgNode(Int(X / 2)), imgNode(X))

4.3 Algoritma Heapify

Algoritma ini berfungsi untuk mengatur penempatan data pada pohon biner agar memenuhi properti *heap* (*node parent* memiliki data terbesar / terkecil / sama dengan data pada *node* kedua anaknya). Algoritma Heapify menerima input parameter A dan i (A merupakan data *array* dan *i* merupakan indeks *node* yang akan dilakukan prosedur *heapify*). Perangkat lunak mendukung 2 buah algoritma *heapify*, yaitu:

1. **Algoritma HeapifyL.** Properti *heap* yang didukung adalah properti *heap*, dimana *node parent* memiliki data terbesar atau sama dengan data pada *node* kedua anaknya. Algoritma HeapifyL(A, i) adalah sebagai berikut:
   a. Set l = LeftNode(i).

b. Set r = RightNode(i).

c. Set Largest = i.

d. Jika l <= ukuran array A, maka cek jika A(l) > A(Largest) maka set Largest = l.

e. Jika r <= ukuran array A, maka cek jika A(r) > A(Largest) maka set Largest = r.

f. Jika Largest <> i, maka:

   i. Tukarkan data pada A(i) dengan data pada A(largest).

   ii. Panggil prosedur Heapify(A, Largest).

2. **Algoritma HeapifyS.** Properti *heap* yang didukung adalah properti *heap*, dimana *node parent* memiliki data terkecil atau sama dengan data pada *node* kedua anaknya. Algoritma HeapifyS(A, i) adalah sebagai berikut:

   a. Set l = LeftNode(i).

   b. Set r = RightNode(i).

   c. Set Smallest = i.

   d. Jika l <= ukuran array A, maka cek jika A(l) < A(Smallest) maka set Smallest = l.

   e. Jika r <= ukuran array A, maka cek jika A(r) < A(Smallest) maka set Smallest = r.

   f. Jika Smallest <> i, maka:

      i. Tukarkan data pada A(i) dengan data pada A(Smallest).

      ii. Panggil prosedur Heapify(A, Smallest).

4.4 Algoritma Build-Heap

Algoritma ini berfungsi untuk membangun struktur pohon *heap* dari data pada *list array* A. Prosedur Build-Heap melakukan prosedur Heapify untuk semua *node* induk dimulai dari node *induk* paling akhir / paling bawah (dihitung dengan rumus floor(length[A]/2)). Algoritma Build-Heap(A) adalah sebagai berikut:

Untuk Pos = Int(Ukuran *array* A / 2) sampai 1 dengan pengurangan nilai 1 setiap *looping*, lakukan algoritma berikut.

1. Jika TipeHeap = ”S” (*node parent* memiliki data terkecil), maka panggil prosedur Heapify(A, Pos).
2. Jika TipeHeap = ”L” (*node parent* memiliki data terbesar), maka panggil prosedur Heapify(A, Pos).

4.5 Algoritma Heap-Sort

Algoritma ini berfungsi untuk mengurutkan sekumpulan data pada list *array* A. Cara kerjanya adalah, Heap Sort akan mengambil data pada *node* akar (*index array* = 1) dan menggantinya (*exchange*) dengan data pada *node* paling akhir (*index array* = *index* paling maksimum dari pohon *heap*). Setelah itu, node terakhir dihapus dan Heap Sort memanggil prosedur *heapify* untuk *node* ke-1 dengan tujuan supaya setelah proses penggantian data, pohon masih memenuhi properti *heap*. Data-data yang dikeluarkan merupakan data yang terurut, baik menaik (*ascending*) maupun menurun (*descending*). Algoritma Heap-Sort(A) adalah sebagai berikut:

1. Panggil prosedur Build-Heap(A).
2. Set Urut = “”.
3. Untuk j = ukuran *array* A sampai 2 dengan pengurangan nilai 1 setiap *looping*, lakukan algoritma di bawah ini.
    a. Jika (TipeHeap = “S” dan TipeUrutan = “A”) Atau (TipeHeap = “L” dan TipeUrutan = “D”) maka:
        i. Jika Urut <> “” maka set Urut = Urut & “”.
        ii. Set Urut = Urut & A(1).
    b. Jika tidak, maka set Urut = A(1) & IIf(Urut <> "", ",", "") & Urut.
    c. Tukarkan data pada A(1) dan A(j).
    d. Kurangi ukuran *array* A dengan 1.
    e. Jika TipeHeap = “S”, maka panggil prosedur HeapifyS(A, 1).
    f. Jika TipeHeap = “L”, maka panggil prosedur HeapifyL(A, 1).
4. Jika (TipeHeap = “S” dan TipeUrutan = “A”) Atau (TipeHeap = “L” dan TipeUrutan = “D”) maka:
    a. Jika Urut <> “” maka set Urut = Urut & “”.
    b. Set Urut = Urut & A(1).
5. Jika tidak, maka set Urut = A(1) & IIf(Urut <> "", ",", "") & Urut.

4.6 Algoritma Fungsi-Fungsi Pendukung

Algoritma ini digunakan sebagai pendukung bagi algoritma utama. Algoritma fungsi-fungsi pendukung adalah sebagai berikut:

1. Fungsi LeftNode(Pos), berfungsi untuk mengembalikan nilai indeks *node* anak sebelah kiri dari *node* dengan indeks = Pos. Algoritmanya adalah:

   LeftNode = 2 * Pos.

2. Fungsi RightNode(Pos), berfungsi untuk mengembalikan nilai indeks *node* anak sebelah kanan dari *node* dengan indeks = Pos. Algoritmanya adalah:

   RightNode = (2 * Pos) + 1.

3. Fungsi ParentNode(Pos), berfungsi untuk mengembalikan nilai indeks *node* induk dari *node* dengan indeks = Pos. Algoritmanya adalah:

   ParentNode = Int (Pos / 2).

4. Fungsi Exchange(A, B), berfungsi untuk menukarkan data pada variabel A dan variabel B. Algoritmanya adalah:

   a. Set T = A.
   b. Set A = B.
   c. Set B = T.

5. Fungsi GetTreeLevel, berfungsi untuk menghitung tinggi (level) pohon biner yang harus dibangun untuk menampung jumlah data dalam *array*. Algoritmanya adalah:

   a. GetTreeLevel = 1.
   b. T = ukuran *array* dari variabel HeapArray.
   c. Selama T <> 1, maka:

i. T = Int(T / 2)

ii. GetTreeLevel = GetTreeLevel + 1

4.7 Implementasi Perangkat Lunak

Spesifikasi Hardware dan Software

Spesifikasi perangkat keras yang direkomendasikan untuk menjalankan program pemahaman Heap Sort ini adalah sebagai berikut :

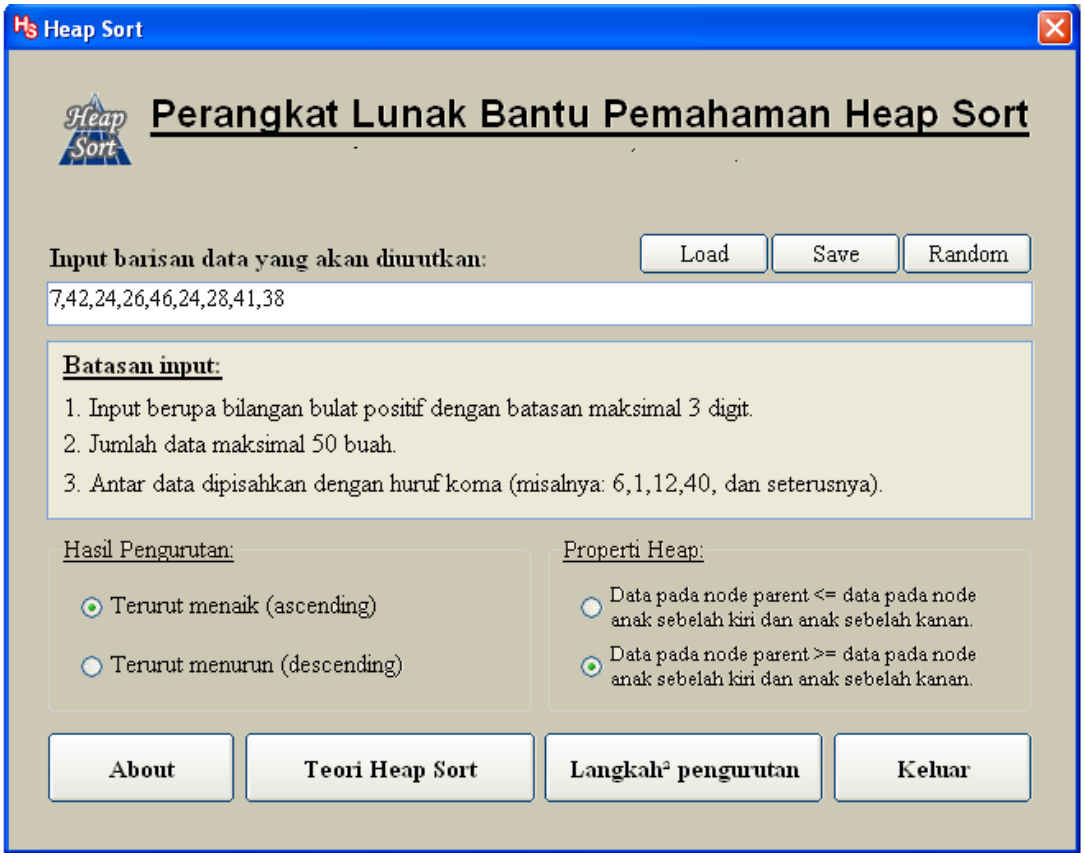
1. Prosesor *Pentium* IV 1.6 GHz.
2. *Harddisk* dengan minimal *free space* 500 MB.
3. Memori sebesar 128 MB.
4. Monitor SVGA dengan resolusi 1024 x 768.
5. *VGA Card* 64 MB.
6. *Keyboard* dan *Mouse*.

Adapun perangkat lunak (*software*) yang digunakan untuk menjalankan aplikasi ini adalah lingkungan sistem operasi *Microsoft Windows 98, 98 Second Edition* (98SE) atau *Microsoft Windows NT / 2000 / XP*.

4.8 Pengujian Program

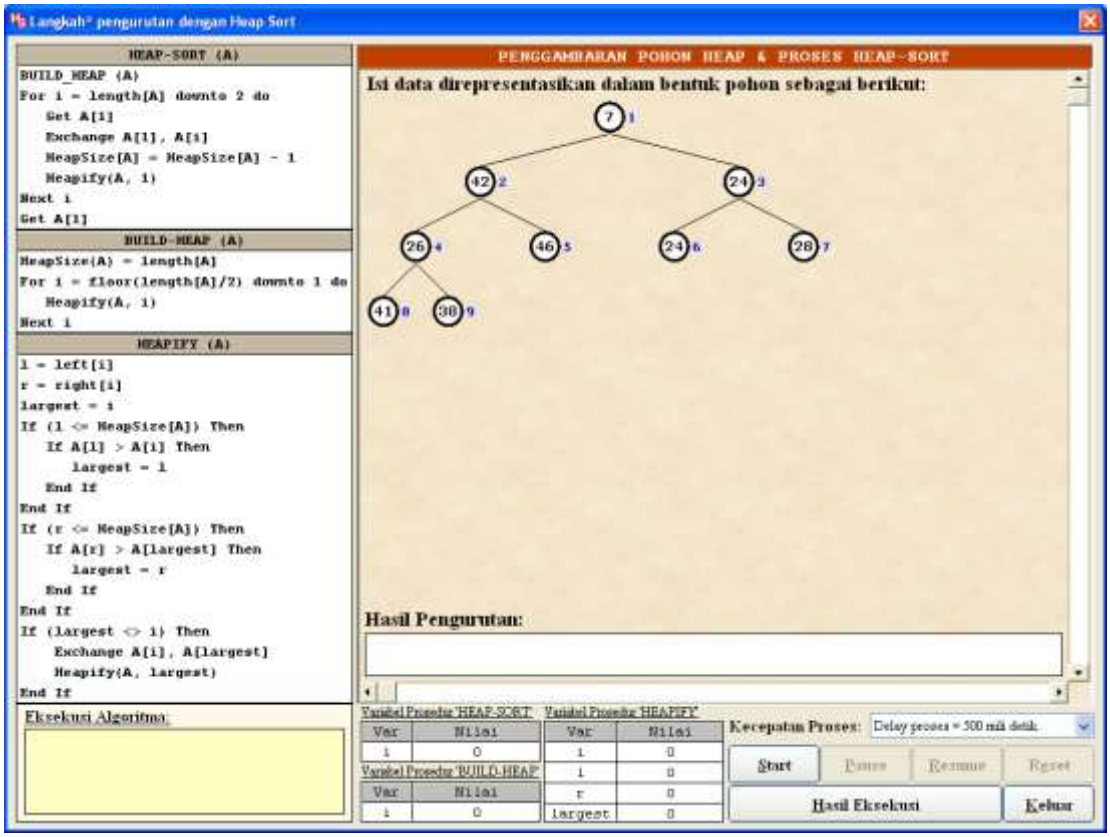
Sebagai contoh pengujian program, misalkan *input* barisan data yang akan diurutkan = 7, 42, 24, 26, 46, 24, 28, 41, 38. Hasil pengurutan yang diinginkan adalah terurut menaik (*ascending*) dan properti *heap* yang dipilih adalah properti dengan data

pada *node parent* >= data pada anak sebelah kiri atau anak sebelah kanan. Tampilan Form Main dapat dilihat pada gambar 4.1.



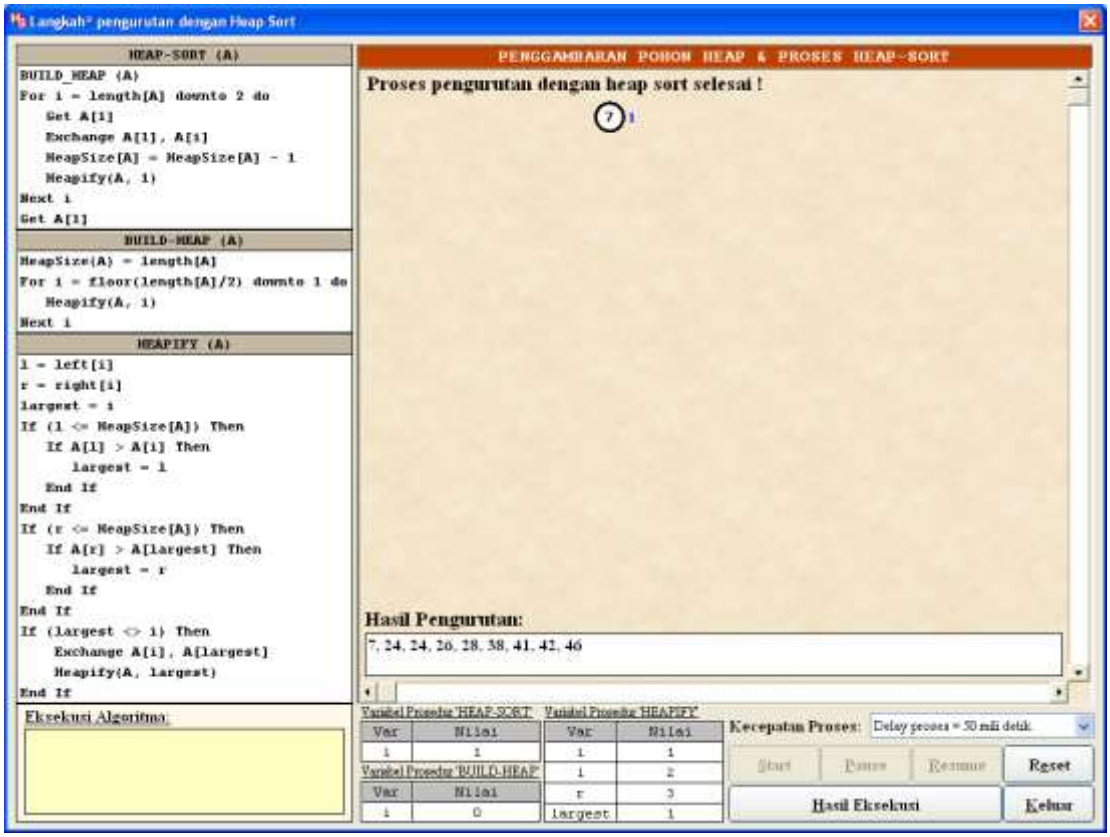
Gambar 4.1 Tampilan Form Main dengan data = 7,42,24,26,46,24,28,41,38.

Tampilan Form Pengurutan dapat dilihat pada gambar 4.2.



Gambar 4.2 Tampilan Form Pengurutan dengan data = 7,42,24,26,46,24,28,41,38.

Tampilan Form Pengurutan setelah proses pengurutan dapat dilihat pada gambar 4.3.



Gambar 4.3 Tampilan *Form* Pengurutan setelah proses pengurutan dengan data = 7,42,24,26,46,24,28,41,38.

Hasil eksekusi algoritma pengurutan *Heap-Sort* yang dihasilkan oleh perangkat lunak adalah:

```
Memulai prosedur HEAP-SORT
--------------------------
BUILD-HEAP (A)

Masuk ke prosedur BUILD-HEAP(A)
HeapSize(A) = length[A] = 9
Untuk i = 4 downto 1,
Heapify(A, 4)

Masuk ke prosedur HEAPIFY(A,4)
```

```
l = left[i] = 2 * 4 = 8
r = right[i] = (2 * 4) + 1 = 9
Largest = i = 4
l <= HeapSize[A],  8 <= 9 (True)
A[l] > A[i], 41 > 26 (True)
Largest = l = 8
r <= HeapSize[A],  9 <= 9 (True)
A[r] > A[Largest], 38 > 41 (False)
Largest <> i,  8 <> 4 (True)
Exchange A[i], A[Largest] -> Exchange A[4], A[8]
A[4] = 26, A[8] = 41  <-->  A[4] = 41, A[8] = 26
Heapify(A, 8)

Masuk ke prosedur HEAPIFY(A,8)
l = left[i] = 2 * 8 = 16
r = right[i] = (2 * 8) + 1 = 17
Largest = i = 8
l <= HeapSize[A],  16 <= 9 (False)
r <= HeapSize[A],  17 <= 9 (False)
Largest <> i,  8 <> 8 (False)
i = 3
Heapify(A, 3)

Masuk ke prosedur HEAPIFY(A,3)
l = left[i] = 2 * 3 = 6
r = right[i] = (2 * 3) + 1 = 7
Largest = i = 3
l <= HeapSize[A],  6 <= 9 (True)
A[l] > A[i], 24 > 24 (False)
r <= HeapSize[A],  7 <= 9 (True)
A[r] > A[Largest], 28 > 24 (True)
Largest = r = 7
Largest <> i,  7 <> 3 (True)
Exchange A[i], A[Largest] -> Exchange A[3], A[7]
A[3] = 24, A[7] = 28  <-->  A[3] = 28, A[7] = 24
Heapify(A, 7)

Masuk ke prosedur HEAPIFY(A,7)
l = left[i] = 2 * 7 = 14
r = right[i] = (2 * 7) + 1 = 15
Largest = i = 7
l <= HeapSize[A],  14 <= 9 (False)
r <= HeapSize[A],  15 <= 9 (False)
Largest <> i,  7 <> 7 (False)
i = 2
```

```
Heapify(A, 2)

Masuk ke prosedur HEAPIFY(A,2)
l = left[i] = 2 * 2 = 4
r = right[i] = (2 * 2) + 1 = 5
Largest = i = 2
l <= HeapSize[A],  4 <= 9 (True)
A[l] > A[i], 41 > 42 (False)
r <= HeapSize[A],  5 <= 9 (True)
A[r] > A[Largest], 46 > 42 (True)
Largest = r = 5
Largest <> i,  5 <> 2 (True)
Exchange A[i], A[Largest] -> Exchange A[2], A[5]
A[2] = 42, A[5] = 46  <-->  A[2] = 46, A[5] = 42
Heapify(A, 5)

Masuk ke prosedur HEAPIFY(A,5)
l = left[i] = 2 * 5 = 10
r = right[i] = (2 * 5) + 1 = 11
Largest = i = 5
l <= HeapSize[A],  10 <= 9 (False)
r <= HeapSize[A],  11 <= 9 (False)
Largest <> i,  5 <> 5 (False)
i = 1
Heapify(A, 1)

Masuk ke prosedur HEAPIFY(A,1)
l = left[i] = 2 * 1 = 2
r = right[i] = (2 * 1) + 1 = 3
Largest = i = 1
l <= HeapSize[A],  2 <= 9 (True)
A[l] > A[i], 46 > 7 (True)
Largest = l = 2
r <= HeapSize[A],  3 <= 9 (True)
A[r] > A[Largest], 28 > 46 (False)
Largest <> i,  2 <> 1 (True)
Exchange A[i], A[Largest] -> Exchange A[1], A[2]
A[1] = 7, A[2] = 46  <-->  A[1] = 46, A[2] = 7
Heapify(A, 2)

Masuk ke prosedur HEAPIFY(A,2)
l = left[i] = 2 * 2 = 4
r = right[i] = (2 * 2) + 1 = 5
Largest = i = 2
l <= HeapSize[A],  4 <= 9 (True)
```

```
A[l] > A[i], 41 > 7 (True)
Largest = l = 4
r <= HeapSize[A],  5 <= 9 (True)
A[r] > A[Largest], 42 > 41 (True)
Largest = r = 5
Largest <> i,  5 <> 2 (True)
Exchange A[i], A[Largest] -> Exchange A[2], A[5]
A[2] = 7, A[5] = 42  <-->  A[2] = 42, A[5] = 7
Heapify(A, 5)

Masuk ke prosedur HEAPIFY(A,5)
l = left[i] = 2 * 5 = 10
r = right[i] = (2 * 5) + 1 = 11
Largest = i = 5
l <= HeapSize[A],  10 <= 9 (False)
r <= HeapSize[A],  11 <= 9 (False)
Largest <> i,  5 <> 5 (False)
i = 0

Untuk i = 9 downto 2,

Ambil A[1] (A[1] = 46)
Exchange A[1], A[i] -> Exchange A[1], A[9]
A[1] = 46, A[9] = 38  <-->  A[1] = 38, A[9] = 46
HeapSize = 9 - 1 = 8
Heapify(A, 1)

Masuk ke prosedur HEAPIFY(A,1)
l = left[i] = 2 * 1 = 2
r = right[i] = (2 * 1) + 1 = 3
Largest = i = 1
l <= HeapSize[A],  2 <= 8 (True)
A[l] > A[i], 42 > 38 (True)
Largest = l = 2
r <= HeapSize[A],  3 <= 8 (True)
A[r] > A[Largest], 28 > 42 (False)
Largest <> i,  2 <> 1 (True)
Exchange A[i], A[Largest] -> Exchange A[1], A[2]
A[1] = 38, A[2] = 42  <-->  A[1] = 42, A[2] = 38
Heapify(A, 2)

Masuk ke prosedur HEAPIFY(A,2)
l = left[i] = 2 * 2 = 4
r = right[i] = (2 * 2) + 1 = 5
Largest = i = 2
```

```
l <= HeapSize[A],  4 <= 8 (True)
A[l] > A[i], 41 > 38 (True)
Largest = l = 4
r <= HeapSize[A],  5 <= 8 (True)
A[r] > A[Largest], 7 > 41 (False)
Largest <> i,  4 <> 2 (True)
Exchange A[i], A[Largest] -> Exchange A[2], A[4]
A[2] = 38, A[4] = 41  <-->  A[2] = 41, A[4] = 38
Heapify(A, 4)

Masuk ke prosedur HEAPIFY(A,4)
l = left[i] = 2 * 4 = 8
r = right[i] = (2 * 4) + 1 = 9
Largest = i = 4
l <= HeapSize[A],  8 <= 8 (True)
A[l] > A[i], 26 > 38 (False)
r <= HeapSize[A],  9 <= 8 (False)
Largest <> i,  4 <> 4 (False)
i = 8

Ambil A[1] (A[1] = 42)
Exchange A[1], A[i] -> Exchange A[1], A[8]
A[1] = 42, A[8] = 26  <-->  A[1] = 26, A[8] = 42
HeapSize = 8 - 1 = 7
Heapify(A, 1)

Masuk ke prosedur HEAPIFY(A,1)
l = left[i] = 2 * 1 = 2
r = right[i] = (2 * 1) + 1 = 3
Largest = i = 1
l <= HeapSize[A],  2 <= 7 (True)
A[l] > A[i], 41 > 26 (True)
Largest = l = 2
r <= HeapSize[A],  3 <= 7 (True)
A[r] > A[Largest], 28 > 41 (False)
Largest <> i,  2 <> 1 (True)
Exchange A[i], A[Largest] -> Exchange A[1], A[2]
A[1] = 26, A[2] = 41  <-->  A[1] = 41, A[2] = 26
Heapify(A, 2)

Masuk ke prosedur HEAPIFY(A,2)
l = left[i] = 2 * 2 = 4
r = right[i] = (2 * 2) + 1 = 5
Largest = i = 2
l <= HeapSize[A],  4 <= 7 (True)
```

```
A[l] > A[i], 38 > 26 (True)
Largest = l = 4
r <= HeapSize[A],  5 <= 7 (True)
A[r] > A[Largest], 7 > 38 (False)
Largest <> i,  4 <> 2 (True)
Exchange A[i], A[Largest] -> Exchange A[2], A[4]
A[2] = 26, A[4] = 38  <-->  A[2] = 38, A[4] = 26
Heapify(A, 4)

Masuk ke prosedur HEAPIFY(A,4)
l = left[i] = 2 * 4 = 8
r = right[i] = (2 * 4) + 1 = 9
Largest = i = 4
l <= HeapSize[A],  8 <= 7 (False)
r <= HeapSize[A],  9 <= 7 (False)
Largest <> i,  4 <> 4 (False)
i = 7

Ambil A[1] (A[1] = 41)
Exchange A[1], A[i] -> Exchange A[1], A[7]
A[1] = 41, A[7] = 24  <-->  A[1] = 24, A[7] = 41
HeapSize = 7 - 1 = 6
Heapify(A, 1)

Masuk ke prosedur HEAPIFY(A,1)
l = left[i] = 2 * 1 = 2
r = right[i] = (2 * 1) + 1 = 3
Largest = i = 1
l <= HeapSize[A],  2 <= 6 (True)
A[l] > A[i], 38 > 24 (True)
Largest = l = 2
r <= HeapSize[A],  3 <= 6 (True)
A[r] > A[Largest], 28 > 38 (False)
Largest <> i,  2 <> 1 (True)
Exchange A[i], A[Largest] -> Exchange A[1], A[2]
A[1] = 24, A[2] = 38  <-->  A[1] = 38, A[2] = 24
Heapify(A, 2)

Masuk ke prosedur HEAPIFY(A,2)
l = left[i] = 2 * 2 = 4
r = right[i] = (2 * 2) + 1 = 5
Largest = i = 2
l <= HeapSize[A],  4 <= 6 (True)
A[l] > A[i], 26 > 24 (True)
Largest = l = 4
```

```
r <= HeapSize[A],  5 <= 6 (True)
A[r] > A[Largest], 7 > 26 (False)
Largest <> i,  4 <> 2 (True)
Exchange A[i], A[Largest] -> Exchange A[2], A[4]
A[2] = 24, A[4] = 26  <-->  A[2] = 26, A[4] = 24
Heapify(A, 4)

Masuk ke prosedur HEAPIFY(A,4)
l = left[i] = 2 * 4 = 8
r = right[i] = (2 * 4) + 1 = 9
Largest = i = 4
l <= HeapSize[A],  8 <= 6 (False)
r <= HeapSize[A],  9 <= 6 (False)
Largest <> i,  4 <> 4 (False)
i = 6

Ambil A[1] (A[1] = 38)
Exchange A[1], A[i] -> Exchange A[1], A[6]
A[1] = 38, A[6] = 24  <-->  A[1] = 24, A[6] = 38
HeapSize = 6 - 1 = 5
Heapify(A, 1)

Masuk ke prosedur HEAPIFY(A,1)
l = left[i] = 2 * 1 = 2
r = right[i] = (2 * 1) + 1 = 3
Largest = i = 1
l <= HeapSize[A],  2 <= 5 (True)
A[l] > A[i], 26 > 24 (True)
Largest = l = 2
r <= HeapSize[A],  3 <= 5 (True)
A[r] > A[Largest], 28 > 26 (True)
Largest = r = 3
Largest <> i,  3 <> 1 (True)
Exchange A[i], A[Largest] -> Exchange A[1], A[3]
A[1] = 24, A[3] = 28  <-->  A[1] = 28, A[3] = 24
Heapify(A, 3)

Masuk ke prosedur HEAPIFY(A,3)
l = left[i] = 2 * 3 = 6
r = right[i] = (2 * 3) + 1 = 7
Largest = i = 3
l <= HeapSize[A],  6 <= 5 (False)
r <= HeapSize[A],  7 <= 5 (False)
Largest <> i,  3 <> 3 (False)
i = 5
```

```
Ambil A[1] (A[1] = 28)
Exchange A[1], A[i] -> Exchange A[1], A[5]
A[1] = 28, A[5] = 7  <-->  A[1] = 7, A[5] = 28
HeapSize = 5 - 1 = 4
Heapify(A, 1)

Masuk ke prosedur HEAPIFY(A,1)
l = left[i] = 2 * 1 = 2
r = right[i] = (2 * 1) + 1 = 3
Largest = i = 1
l <= HeapSize[A],  2 <= 4 (True)
A[l] > A[i], 26 > 7 (True)
Largest = l = 2
r <= HeapSize[A],  3 <= 4 (True)
A[r] > A[Largest], 24 > 26 (False)
Largest <> i,  2 <> 1 (True)
Exchange A[i], A[Largest] -> Exchange A[1], A[2]
A[1] = 7, A[2] = 26  <-->  A[1] = 26, A[2] = 7
Heapify(A, 2)

Masuk ke prosedur HEAPIFY(A,2)
l = left[i] = 2 * 2 = 4
r = right[i] = (2 * 2) + 1 = 5
Largest = i = 2
l <= HeapSize[A],  4 <= 4 (True)
A[l] > A[i], 24 > 7 (True)
Largest = l = 4
r <= HeapSize[A],  5 <= 4 (False)
Largest <> i,  4 <> 2 (True)
Exchange A[i], A[Largest] -> Exchange A[2], A[4]
A[2] = 7, A[4] = 24  <-->  A[2] = 24, A[4] = 7
Heapify(A, 4)

Masuk ke prosedur HEAPIFY(A,4)
l = left[i] = 2 * 4 = 8
r = right[i] = (2 * 4) + 1 = 9
Largest = i = 4
l <= HeapSize[A],  8 <= 4 (False)
r <= HeapSize[A],  9 <= 4 (False)
Largest <> i,  4 <> 4 (False)
i = 4

Ambil A[1] (A[1] = 26)
Exchange A[1], A[i] -> Exchange A[1], A[4]
```

```
A[1] = 26, A[4] = 7  <-->  A[1] = 7, A[4] = 26
HeapSize = 4 - 1 = 3
Heapify(A, 1)

Masuk ke prosedur HEAPIFY(A,1)
l = left[i] = 2 * 1 = 2
r = right[i] = (2 * 1) + 1 = 3
Largest = i = 1
l <= HeapSize[A],  2 <= 3 (True)
A[l] > A[i], 24 > 7 (True)
Largest = l = 2
r <= HeapSize[A],  3 <= 3 (True)
A[r] > A[Largest], 24 > 24 (False)
Largest <> i,  2 <> 1 (True)
Exchange A[i], A[Largest] -> Exchange A[1], A[2]
A[1] = 7, A[2] = 24  <-->  A[1] = 24, A[2] = 7
Heapify(A, 2)

Masuk ke prosedur HEAPIFY(A,2)
l = left[i] = 2 * 2 = 4
r = right[i] = (2 * 2) + 1 = 5
Largest = i = 2
l <= HeapSize[A],  4 <= 3 (False)
r <= HeapSize[A],  5 <= 3 (False)
Largest <> i,  2 <> 2 (False)
i = 3

Ambil A[1] (A[1] = 24)
Exchange A[1], A[i] -> Exchange A[1], A[3]
A[1] = 24, A[3] = 24  <-->  A[1] = 24, A[3] = 24
HeapSize = 3 - 1 = 2
Heapify(A, 1)

Masuk ke prosedur HEAPIFY(A,1)
l = left[i] = 2 * 1 = 2
r = right[i] = (2 * 1) + 1 = 3
Largest = i = 1
l <= HeapSize[A],  2 <= 2 (True)
A[l] > A[i], 7 > 24 (False)
r <= HeapSize[A],  3 <= 2 (False)
Largest <> i,  1 <> 1 (False)
i = 2

Ambil A[1] (A[1] = 24)
Exchange A[1], A[i] -> Exchange A[1], A[2]
```

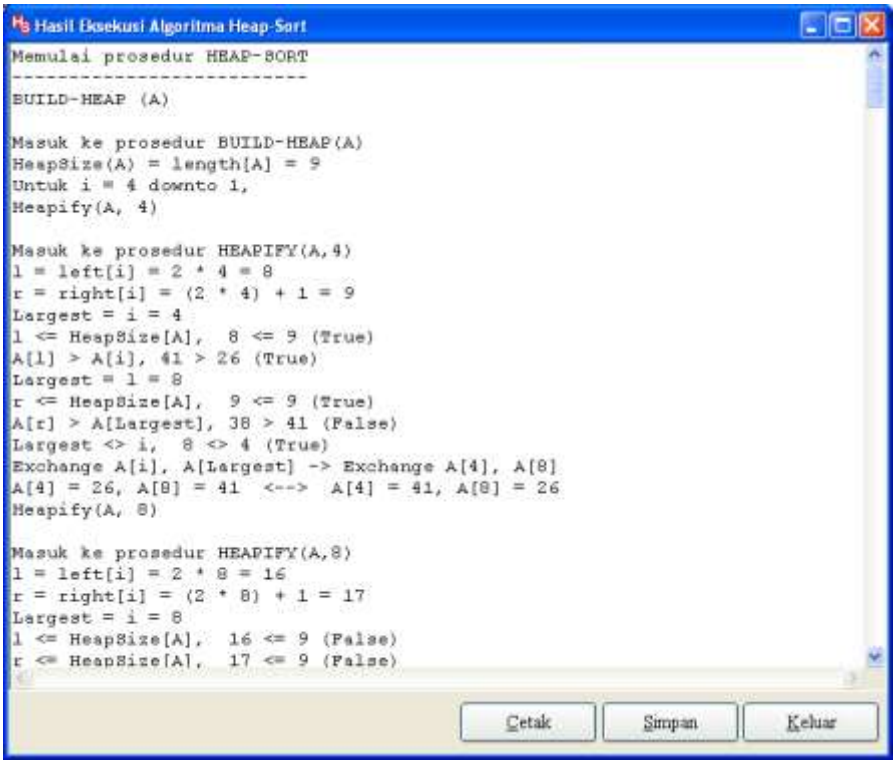
```
A[1] = 24, A[2] = 7  <-->  A[1] = 7, A[2] = 24
HeapSize = 2 - 1 = 1
Heapify(A, 1)

Masuk ke prosedur HEAPIFY(A,1)
l = left[i] = 2 * 1 = 2
r = right[i] = (2 * 1) + 1 = 3
Largest = i = 1
l <= HeapSize[A],  2 <= 1 (False)
r <= HeapSize[A],  3 <= 1 (False)
Largest <> i,  1 <> 1 (False)
i = 1

Ambil A[1] (A[1] = 7)

Hasil Pengurutan = 7, 24, 24, 26, 28, 38, 41, 42, 46
```

Tampilan hasil eksekusi pada *form* 'Hasil' dapat dilihat pada gambar 4.4 berikut.



Gambar 4.4 Tampilan *Form* Hasil Eksekusi dengan data =

7,42,24,26,46,24,28,41,38.

Tampilan teori pada perangkat lunak seperti terlihat pada gambar 4.5 berikut.

Gambar 4.5 Contoh Tampilan *Form* Teori

BAB V

KESIMPULAN DAN SARAN

5.1 Kesimpulan

Setelah menyelesaikan perangkat lunak bantu pemahaman Heap Sort, penulis menarik kesimpulan sebagai berikut:

1. Perangkat lunak menjelaskan algoritma pengurutan Heap Sort dan gambar keadaan pohon biner secara bertahap serta menampilkan Form Teori, sehingga dapat membantu pemahaman mengenai pengurutan dengan metode Heap Sort.
2. Algoritma pengurutan Heap Sort merupakan salah satu metode pengurutan tercepat setelah Merge Sort dan Quick Sort dengan kompleksitas O(n log n).
3. Kompleksitas prosedur Build-Heap adalah O(n), kompleksitas prosedur Heapify adalah O(log n), sehingga kompleksitas algoritma Heap Sort adalah O(n log n).

5.2 Saran

Penulis ingin memberikan beberapa saran yang mungkin dapat membantu dalam pengembangan perangkat lunak ini yaitu :

1. Algoritma pengurutan Heap Sort yang terdapat dalam perangkat lunak dapat dikembangkan dengan *class object oriented* sehingga dapat dipisahkan secara independen dan dapat digunakan untuk membangun perangkat lunak lain yang membutuhkan algoritma pengurutan.

2. Penggambaran proses-proses yang terjadi pada pohon biner dapat ditingkatkan dengan menambahkan kualitas animasi yang lebih baik. Animasi yang baik bisa didapatkan dengan aplikasi Macromedia Flash.

DAFTAR PUSTAKA

Agung Novian, Panduan MS. Visual Basic 6, Andi, Yogyakarta, 2013.

Andrian, Yudhi, and Purwa Hasan Putra. "Analisis Penambahan Momentum Pada Proses Prediksi Curah Hujan Kota Medan Menggunakan Metode Backpropagation Neural Network." Seminar Nasional Informatika (SNIf). Vol. 1. No. 1. 2017.

Arief Ramadhan, 36 Jam Belajar Komputer Visual Basic 6.0, PT. Elex Media Komputindo, Jakarta, 2014.

Azmi, Fadhillah, And Winda Erika. "Analisis Keamanan Data Pada Block Cipher Algoritma Kriptografi Rsa." Cess (Journal Of Computer Engineering, System And Science) 2.1: 27-29.

Bambang Hariyanto, Struktur Data : Memuat Dasar Pengembangan Orientasi Objek, Edisi Kedua, Informatika Bandung, April 2012.

Dewa Ketut Satriawan Suditresna Jaya,Ryan Ardiansyah Putra,Komang Eric Widhi Antara, Desain dan analis Algoritma heap sort, dari universitas ganesha

Erika, Winda, Heni Rachmawati, and Ibnu Surya. "Enkripsi Teks Surat Elektronik (E-Mail) Berbasis Algoritma Rivest Shamir Adleman (RSA)." Jurnal Aksara Komputer Terapan 1.2 (2012).

Hafni, Layla, And Rismawati Rismawati. "Analisis Faktor-Faktor Internal Yang Mempengaruhi Nilai Perusahaan Pada Perusahaan Manufaktur Yang Terdaftar Di Bei 2011-2015." Bilancia: Jurnal Ilmiah Akuntansi 1.3 (2017): 371-382.

Hamdi, Muhammad Nurul, Evi Nurjanah, And Latifah Safitri Handayani. "Community Development Based Onibnu Khaldun Thought, Sebuah Interpretasi Program Pemberdayaan Umkm Di Bank Zakat El-Zawa." El Muhasaba: Jurnal Akuntansi (E-Journal) 5.2 (2014): 158-180.

http://www.stanford.edu/~rashimisu/Sorting.pdf

Indra Permana, Aminuddin "Sistem Pakar Mendeteksi Hama Dan Penyakit Tanaman Kelapa Sawit Pada Pt. Moeis Kebun Sipare-Pare Kabupaten Batubara." (2013).

Jurnal Mahajana Informasi, Vol.1 No 2, 2016 e-ISSN: 2527-8290

Muttaqin, Muhammad. "Analisa Pemanfaatan Sistem Informasi E-Office Pada Universitas Pembangunan Panca Budi Medan Dengan Menggunakan Metode Utaut." Jurnal Teknik Dan Informatika 5.1 (2018): 40-43.

Muttaqin, Muhammad. "Portal Academic Portal Innovation Based On Website In The Era Of Digital 4.0 Technology Now."

Permana, A. I., and Z. Tulus. "Combination of One Time Pad Cryptography Algorithm with Generate Random Keys and Vigenere Cipher with EM2B KEY." (2020).

Permana, Aminuddin Indra. "Kombinasi Algoritma Kriptografi One Time Pad dengan Generate Random Keys dan Vigenere Cipher dengan Kunci EM2B." (2019).

Perwitasari, I. D. (2018). Teknik Marker Based Tracking Augmented Reality untuk Visualisasi Anatomi Organ Tubuh Manusia Berbasis Android. INTECOMS: Journal of Information Technology and Computer Science, 1(1), 8-18.

Puspita, Khairani, and Purwa Hasan Putra. "Penerapan Metode Simple Additive Weighting (SAW) Dalam Menentukan Pendirian Lokasi Gramedia Di Sumatera Utara." Seminar Nasional Teknologi Informasi Dan Multimedia, ISSN. 2015.

Ralph Rinaldi Munir, Leoni Lidia, Algoritma dan Pemrograman, Edisi Kedua, 2015.

Rizal, Chairul. "Pengaruh Varietas dan Pupuk Petroganik Terhadap Pertumbuhan, Produksi dan Viabilitas Benih Jagung (Zea mays L.)." ETD Unsyiah (2013).

Robert L.Kruse, Data Structures & Program Design, Second Edition, 2013.

Syahputra, Rizki, And Hafni Hafni. "Analisis Kinerja Jaringan Switching Clos Tanpa Buffer." Journal Of Science And Social Research 1.2 (2018): 109-115.

Thomas H.Cormen, Charles.E.Leiserson dan Ronald L.Riverst, Introduction to Algorithms, Mc GRAW-HILL, 1013.

Wahyuni, Sri. "Implementasi Rapidminer Dalam Menganalisa Data Mahasiswa Drop Out." Jurnal Abdi Ilmu 10.2 (2018): 1899-1902.

www.cs.auckland.ac.nz/software/AlgAnim/heapso rt.html.

www.webopedia.com/TERM/H/heap_sort.html.

www-cse.uta.edu/~holder/courses/ cse2320/lectures/applets/sort1/heapsort.html.