Pohon Biner Sebagai Struktur Data Heap dan Aplikasinya

Muhammad Adinata/13509022
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
mail.dieend@gmail.com

***Abstract*—Heap (tumpukan) merupakan sebuah pohon biner dengan sifat khusus. Makalah ini membahas bagaimana struktur data heap bekerja, kelebihan-kelebihan struktur data heap, dan beberapa aplikasi struktur data heap. Heap memiliki operasi getMax, insert, fixheap, delete, dan buildHeap. Untuk optimasi, heap dapat diimplementasikan dengan tree menggunakan array. Heap dapat digunakan sebagai optimasi beberapa algoritma, seperti pengurutan, pohon merentang minimum prim, dan algoritma pencarian jarak terpendek antara dua simpul djikstra.**

***Kata Kunci*—Heap, prioritas, optimasi.**

I. PENDAHULUAN

Heap merupakan sebuah pohon biner dengan sifat khusus. Definisi dari heap meliputi deskripsi struktur dan kondisi data pada simpul, yang disebut *partial order tree property* (sifat pohon setengah rapi) atau sifat heap. Heap memberikan implementasi yang efisien dari antrian berprioritas. Pada heap, elemen dengan prioritas "tertinggi" berada di posisi akar dari pohon biner. Bergantung pada pengertian prioritasnya, tertinggi bisa bermaksud elemen dengan prioritas tertinggi (heap membesar) atau dengan prioritas terendah (heap mengecil)[1].

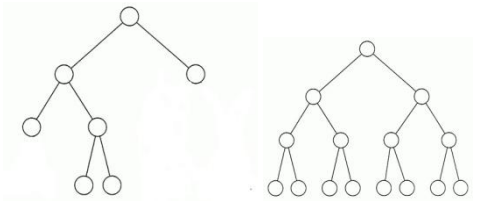
Secara formal, pohon biner *T* dengan tinggi *h* merupakan sebuah struktur heap jika dan hanya jika T memenuhi kondisi berikut:

1. *T* merupakan pohon biner penuh (*complete tree)* setidaknya hingga aras *h* – 1.
2. Semua daun berada pada aras *h* atau *h* – 1.
3. Semua lintasan ke daun dengan aras *h* selalu berada di kiri lintasan ke daun dengan aras *h* – 1.

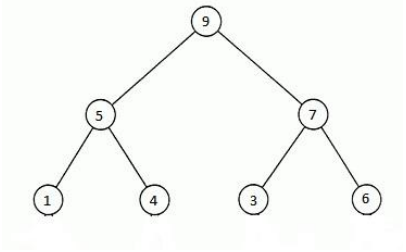
Simpul dalam paling kanan yang berada pada aras *h* – 1 bisa memiliki anak kiri dan tidak memiliki anak kanan tapi tidak sebaliknya. Simpul dalam lainnya tepat memiliki dua anak.

Definisi diatas belum lengkap jika belum menyertakan kondisi data pada simpul (sifat pohon setengah rapi). Sebuah pohon *T* disebut pohon setengah rapi (membesar) jika dan hanya jika kunci dari simpul manapun lebih besar atau sama dengan kunci dari setiap anaknya (jikaada).

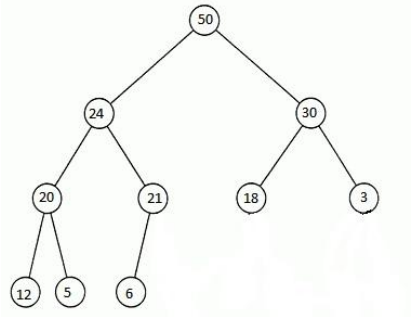
Sebuah pohon biner penuh merupakan struktur heap tanpa definisi kondisi data. Ketika simpul baru ditambahkan ke sebuah heap, simpul tersebut harus di tambahkan dari kiri ke kanan dari aras tertinggi. Dan jika sebuah simpul di lepaskan, simpul tersebut harus merupakan simpul terkanan pada aras tetinggi jika struktur hasil masih harus merupakan heap. Jika diperhatikan dengan seksama, dapat dilihat bahwa sebuah heap merupakan pohon dengan anak kiri dan anak kanan (jika ada) berupa heap dengan tinggi yang sama atau anak kiri lebih tinggi satu dari anak kanan. Dan akar dari tiap upaheap lebih kecil dari akar heap.



(a) Pohon biner (b) Pohon penuh



(c) Heap 1



(d) Heap 2

Gambar 1. Pohon biner, pohon biner penuh, dan heap[6]

Tujuan dari penulisan makalah ini adalah untuk mengenalkan struktur data heap yang banyak digunakan untuk optimasi program. Diharapkan makalah ini dapat mempermudah pembelajaran lebih lanjut tentang struktur data heap dan dapat digunakan untuk program sederhana.

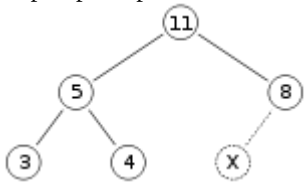
II. METODE

A. GetMax

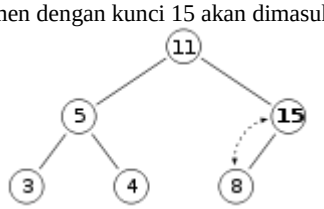
Struktur data heap memiliki sifat elemen dengan kunci terbesar akan berada pada akar. Dengan demikian untuk mendapatkan elemen terbesar cukup dengan melihat akar dari heap.

B. InsertHeap

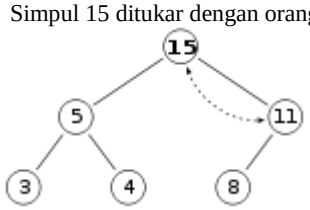
Menambahkan elemen ke dalam heap tentunya harus mempertahankan bentuk dan sifat dari heap. Dengan meletakkan simpul di posisi terkanan pada aras tertinggi maka bentuk heap dapat dipertahankan. Namun sifat dari heap belum tentu bertahan. Untuk mempertahankan sifat dari heap dapat dilakukan dengan membandingkan kunci simpul yang baru dengan orang tuanya. Jika ternyata kunci orang tua lebih besar, tukar elemen dari simpul yang baru dengan orang tuanya, sehingga elemen yang baru berada pada simpul orang tua yang baru. Hal ini terus dilakukan hingga elemen yang baru berada pada simpul yang tepat atau elemen yang baru telah berada di akar. Akar tidak memiliki orang tua, sehingga elemen yang baru merupakan elemen dengan kunci terbesar sehingga sifat heap dapat dipertahankan.



(a) Elemen dengan kunci 15 akan dimasukkan ke heap



(b) Simpul 15 ditukar dengan orang tuanya

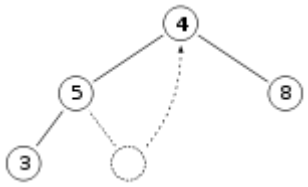


(c) Simpul 15 ditukar dengan orang tuanya, karena merupakan akar, berhenti.

Gambar 2. Ilustrasi InsertHeap

C. DeleteRoot

Untuk menghapus akar dari root, kita akan menyalin elemen dari simpul di posisi terkanan pada aras tertinggi ke akar dan kemudian menghapus simpul tersebut. Dengan demikian bentuk heap dapat dipertahankan. Namun sifat heap akan hilang sehingga kita perlu memperbaiki sifat heap dengan fixHeap. Perhatikan bahwa anak kanan dari akar merupakan sebuah upaheap, demikian juga anak kirinya, dengan selisih tinggi kedua upaheap itu tidak lebih besar dari 1.

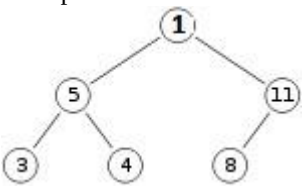


Gambar 3. Ilustrasi DeleteRoot

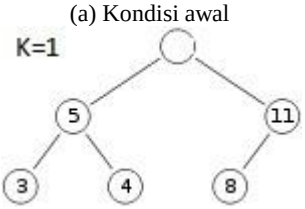
D. FixHeap

FixHeap memperbaiki pohon setengah rapi pada heap dimana sifat heap itu telah ada di simpul manapun kecuali mungkin di akar. Lebih spesifik, ketika fixHeap dilakukan, kita memiliki sebuah struktur heap dengan akar yang "kosong" dan anak kiri maupun anak kanan merupakan heap dan selisih tinggi kedua upaheap itu tidak lebih besar dari 1.

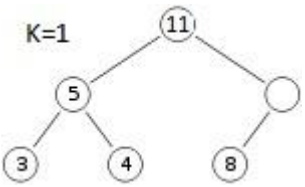
FixHeap bekerja dengan "mengosongkan" akar terlebih dahulu dan menyalin nilainya kedalam K. Lalu mulai dari akar dan simpul yang "kosong" (akar "kosong" saat pertama kali dilakukan) ditukar dengan anak kiri atau anak kanannya. Hal ini terus dilakukan hingga simpul berada pada posisi yang tepat. Posisi yang tepat berarti kunci K harus lebih besar atau sama dengan anak-anaknya. Jadi pada setiap langkah K dibandingkan dengan anak dari simpul "kosong" saat ini. JikaK lebih besar (atau sama dengan), K bisa disalin ke simpul kosong tersebut. Jika tidak, anak yang lebih besar disalin ke simpul yang "kosong" sehingga simpul dengan anak yang lebih besar tersebut menjadi anak yang "kosong". Proses ini diulangi hingga simpul "kosong" berada posisi yang tepat atau merupakan daun.



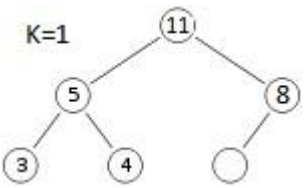
(a) Kondisi awal



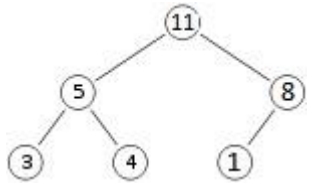
(b) akar "kosong", K=1



(c) Anak terbesar dari simpul yang kosong, 11, lebih besar daripada K, sehingga simpul tersebut naik dan simpul yang kosong turun



(d) Anak terbesar dari simpul yang kosong, 8, lebih besar daripada K, sehingga simpul tersebut naik dan simpul yang kosong turun



(e) Karena simpul kosong merupakan daun, K=1 disalin.

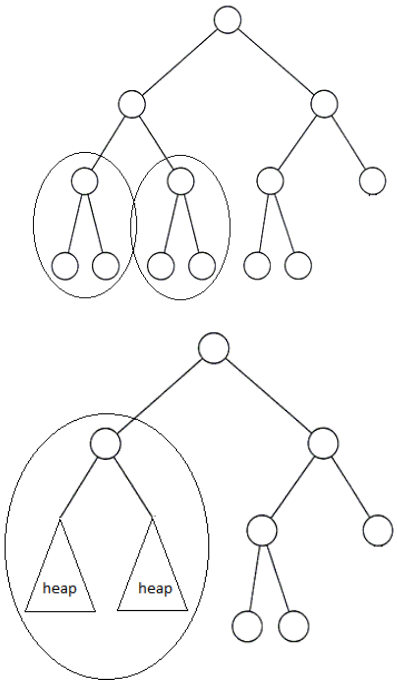
Gambar 4. Ilustrasi FixHeap

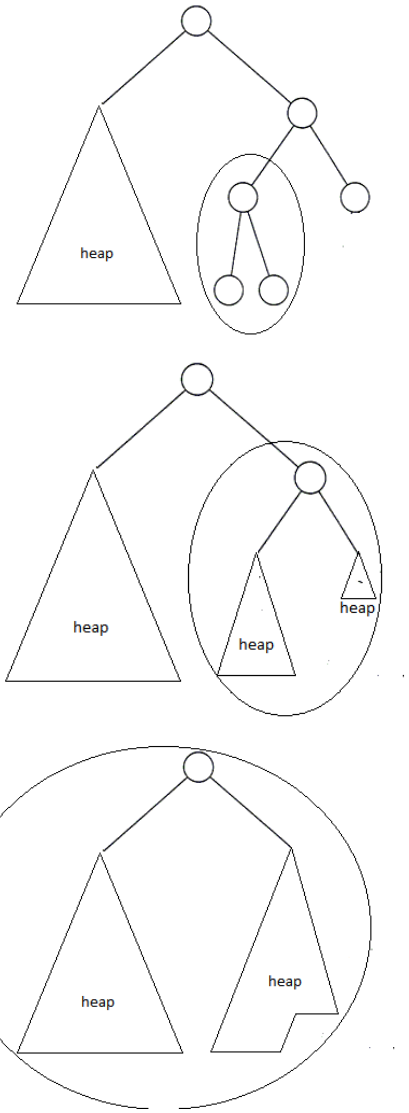
E. Membangun Heap: Brute-Force

Sebuah heap dapat dibangun menggunakan insertHeap berkali-kali. Jika terdapat n elemen, maka dilakukan n kali insertHeap.

F. Membangun Heap: Divide-and-Conquer

Misalkan kita memulai dengan meletakkan seluruh elemen pada heap dengan urutan sembarang, dimana sifat pohon setengah rapi tidak perlu dipenuhi di upaheap manapun. Prosedur fixHeap menimbulkan ide pendekatan *Divide-and-Conquer* untuk memenuhi sifat pohon setengah terurut pada heap tersebut. Dua upaheap dapat diubah menjadi heap secara rekursif dan fixHeap dapat digunakan untuk memindahkan elemen pada akar ke tempatnya yang benar, lalu mengkombinasikan kedua upaheap dan akar menjadi satu heap secara keseluruhan. Basis dari rekursinya adalah tree dengan 1 elemen (daun) karena sudah merupakan heap dengan sifat pohon setengah terurut.





Gambar 5. Membangun heap dengan pendekatan *Divide-and-Conquer:* Daun merupakan heap. FixHeap dilakukan untuk setiap upapohon yang dilingkari

III. HASIL DAN ANALISIS

A. Optimasi dengan array

Heap dapat diimplementasikan dengan tree menggunakan array. Hal ini dapat dilakukan dengan baik karena heap merupakan pohon biner penuh. Akar akan berada di indeks 1, anak kiri berada di indeks 2, dan anak kanan berada di indeks 3. Simpul dengan indeks n akan memiliki anak di simpul 2n dan 2n+1.

Heap pada gambar 1 (c) disimpan dalam array berikut:

Indeks	1	2	3	4	5	6	7	8	9
kunci	9	5	7	1	4	3	6		

Dengan demikian orang tua dari simpul dengan indeks n adalah n div 2. Optimasi ini dapat mempercepat waktu yang digunakan untuk mengakses simpul dari heap dan mempercepat akses ke daun terkanan dengan aras tertinggi karena pasti sama dengan besar heap.

B. Implementasi dalam Bahasa Algoritmik

1. GetMax

```
Function GetMax(H: Heap)
  -> H[1]
```

2. InsertHeap

```
Procedure Insert (i/o H:Heap,in N:infotype)
I.S: Heap sesuai definisi
F.S: Elemen heap bertambah 1, yaitu N
Kamus
  i : idx
Algoritma
  i <- H.size+1;
  H[i] <- N
  While (i/2>0 and N.key>H[i/2].key)
    Tukar(H[i],H[i/2])
    i<-i/2
  H.size <- H.size+1
```

3. DeleteHeap

```
Procedure DeleteMax(i/o H: Heap)
I.S: Heap sesuai definisi tidak kosong
F.S: Akar heap dihapus, Heap sesuai
definisi
Algoritma
  H[1] <- H[H.size]
  H.size <- H.size-1
  FixHeap(H,1,H[1])
```

4. FixHeap

```
Procedure FixHeap (i/o H:Heap,in akar:idx,
K:infotype)
I.S: anak kiri dan kanan merupakan heap
sesuai definisi
F.S: H menjadi heap sesuai definisi
Kamus Lokal
  L, R, largerSubHeap : idx
Algoritma
  L <- 2*akar, R <- 2*akar+1
  if L>H.size then
    H[akar] <- K {akar adalah daun}
  else
    if (L=H.size) then {tidak ada anak R}
      largerSubHeap = L
    else if (H[L].key>H[R].key) then
      largerSubHeap = L
    else
      largerSubHeap = R
    if (K.key ≥ H[largerSubHeap].key) then
      H[akar] <- H[largerSubHeap]
      FixHeap(H,largerSubHeap,K)
```

5. BuildHeap1

```
Procedure BuiltHeap1 (out H:Heap)
I.S: Sembarang
F.S: Heap terisi input dari user
Kamus Lokal
  K : infotype
  sz : integer
Algoritma
  Input(sz)
  While (sz>0) do
    Input(K)
    Insert(H,K)
    sz<-sz-1
```

6. BuildHeap2

```
Procedure BuiltHeap2 (i/o H:Heap, in
akar:idx)
I.S: Heap sembarang (tidak harus memenuhi
sifat heap)
F.S: Heap terisi input dari user
Kamus Lokal
  K : infotype
  sz : integer
Algoritma
  if (akar*2 < H.size) then
   BuiltHeap2(H,akar*2);
   BuiltHeap2(H,akar*2+1);
   K <- H[akar]
   fixHeap(H,K)
```

C. Analisis Kompleksitas

1. GetMax

GetMax langsung mengambil dari akar heap. Operasi ini konstan dengan kompleksitas T(n) = O(1)

2. InsertHeap

Kasus terburuk dari InsertHeap adalah saat elemen yang dimasukkan merupakan elemen dengan kunci terbesar, sehingga elemen akan terus ditukar dengan orang tuanya hingga mencapai akar dari heap. Paling banyak adalah sebesar tinggi dari heap. Karena heap merupakan pohon biner, maka untuk pohon dengan elemen n, banyak penukaran maksimum adalah sebanyak $^2\log(n)$, kompleksitasnya adalah T(n) = O(log n)

3. FixHeap

Banyaknya perbandingan yang dilakukan fixheap pada sebuah heap dengan n simpul paling banyak adalah 2 log n. 2 karena untuk setiap simpul dilakukan 2 kali perbandingan, yaitu untuk membandingkan kunci anak yang lebih besar dan untuk membandingkan anak terbesar tersebut dengan kunci dari K. Sedangkan log n karena untuk kasus terburuk dari heap dengan n simpul adalah sebesar tinggi dari pohon, atau sama dengan log n. Oleh karena itu, kompleksitasnya adalah T(n) = O(log n)

4. DeleteHeap

DeleteHeap melakukan sekali operasi penukaran dan operasi FixHeap. Oleh karena itu kompleksitasnya adalah T(n) = O(log n)

5. Membangun Heap: Brute-Force

Secara brute force, insert dengan kompleksitas O(log n) dilakukan sebanyak n kali, sehingga kompleksitas untuk membangun heap secara bruteforce adalah O(n log n)

6. Membangun Heap: Divide-and-Conquer

Membangun heap secara brute force bukanlah cara terbaik. Jika seluruh upapohon dengan tinggi h (diukur dari bawah) merupakan heap secara definisi, pohon pada h+1 bisa disesuaikan dengan definisi dengan menukar akarnya kebawah. Proses ini melakukan O(h) operasi penukaran untuk tiap simpul. Dan sebagian besar dari penukaran ini memiliki tinggi yang kecil. Banyaknya simpul pada aras h $\leq \frac{n}{2^{h+1}}$ sehingga kompleksitas melakukan fixHeap untuk seluruh node adalah

$$\sum_{h=0}^{\log n} \frac{n}{2^{h+1}} O(h) = O\left(n \sum_{h=0}^{\log n} \frac{h}{2^h}\right)$$
$$\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right)$$
$$= O(n).$$

Jika dirangkum akan menjadi tabel berikut:

Tabel I. Kompleksitas Operasi pada Heap

Operasi	Kompleksitas
GetMax	O(1)
Insert	O(log n)
Delete	O(log n)
FixHeap	O(log n)
BuildHeap1	O(n log n)
BuildHeap2	O(n)

IV. Pembahasan

Dari analisis dapat dilihat bahwa heap merupakan sebuah struktur data yang optimal secara waktu. Kelebihan ini dapat diaplikasikan ke algoritma lain, seperti contoh berikut.

A. Aplikasi heap untuk pengurutan

Dikenal dengan istilah heapsort. Dengan membangun sebuah heap, dan kemudian mendelete seluruh elemennya sambil mengambil elemen dengan kunci terbesar, maka akan didapatkan dengan hasil yang terurut, sesuai dengan pengertian prioritas "tertinggi". Kompleksitas dari heapsort ini adalah O(n log n), sebanding dengan quicksort dan mergesort. Kelebihan dari heapsort dibanding quicksort adalah heapsort memiliki kasus terbaik yang optimal. Sedangkan dibandingkan dengan mergesort, heapsort dapat dilakukan dalam array elemen itu sendiri tanpa menggunakan array temporer.

Perbandingan antara beberapa jenis pengurutan dapat dilihat pada tabel berikut.

Tabel II. Perbandingan Algoritma Pengurutan

Algoritma	*Kasus terburuk*	*Kasus Rata-rata*	*Penggunaan memori*
Insertion Sort	$n^2/2$	$O(n^2)$	Di tempat
Quicksort	$n^2/2$	O(n log n)	Memori tambahan sebanding dengan log n
Mergesort	n log n	O(n log n)	Memori tambahan sebanding dengan n untuk *merging*
Heapsort	2n log n	O(n log n)	Di tempat

B. Aplikasi heap untuk antrian berprioritas

Antrian berprioritas (*priority queue*) akan bekerja secara optimal jika menggunakan struktur data heap. Karena dibandingkan menggunakan struktur data yang lain seperti list atau array, akan memiliki kasus terburuk mendekati n untuk operasi insert. Sedangkan dengan heap kasus terburuknya adalah log n.

C. Aplikasi heap untuk algoritma pohon merentang minimum prim

Heap dapat digunakan untuk optimasi algoritma pohon merentang minimum ketika mencari simpul terdekat dari pohon merentang minimum sementara yang telah kita punya. Algoritma prim pertama kali bekerja dengan mengambil salah satu simpul sembarang dari graf. Dengan memasukkan seluruh simpul yang bertetanggaan dengan simpul tersebut ke dalam heap dengan sifat heap sisi dari simpul awal ke simpul baru terkecil memiliki prioritas tertinggi. Sehingga pencarian hanya membutuhkan log n. Untuk setiap simpul baru yang ditambahkan kedalam pohon merentang minimum sementara, heap diperbaharui dengan simpul yang bertetanggaan dengan simpul baru tersebut.

Kompleksitas pohon merentang minimum dapat dioptimasi dari $O(V^2)$ menjadi $O(E \log(V))$[8].

V. Kesimpulan

Heap memiliki kemiripan dengan antrian berprioritas. Simpul dengan prioritas tertinggi akan selalu berada di akar, dan ketika dihapus, penggantinya akan menjadi yang memiliki prioritas tertinggi. Heap berguna ketika membuat sebuah algoritma yang memerlukan sesuatu yang diproses dalam urutan penuh, tapi tidak ingin melakukan sort secara menyeluruh ataupun peduli dengan simpul-simpul lain selain simpul dengan prioritas tertinggi. Dengan menggunakan heap, banyak algoritma yang dapat dioptimasi, contohnya adalah pohon merentang minimum prim dan pencarian jarak terpendek antara dua simpul pada graf, algoritma Dijkstra.

Dengan pemilihan struktur data yang tepat, dapat memangkas kompleksitas sebuah algoritma.

VI. Ucapan Terima Kasih

Penulis ingin mengucapkan rasa syukur kepada Allah SWT yang telah memberikan rahmat dan rezekinya sehingga makalah ini dapat penulis selesaikan dengan segala kendalanya. Penulis juga mengucapkan terima kasih kepada Bapak Rinaldi Munir dan Ibu Harlili yang telah mengajar mata kuliah Struktur Diskrit selama satu semester dan menyampaikan berbagai ilmu yang pasti akan bermanfaat di perkuliahan kami dan hidup kami.

REFERENSI

[1] S. Baase, A. Gelder, *Computer Algorithms Introduction to Design and Analysis.* United States of America: Addison Wesley Longman, 2000.

[2] Jong Jek Siang, *Matematika Diskrit dan Aplikasinya pada Ilmu Komputer*. Yogyakarta: Penerbit ANDI, 2002.

[3] S. Halim, *Competitive Programming Increasing the Lower Bound of Programming Contests.* Singapore: National University of Singapore (NUS), 2010.

[4] "Heap (data structure)", http://en.wikipedia.org/wiki/Heap_(data_structure), diakses pada 15 Desember 2010.

[5] "Binary heap", http://en.wikipedia.org/wiki/Binary_heap, diakses pada 15 Desember 2010.

[6] "heap", http://douc54.cs.edinboro.edu/~bennett/class/csci385/fall2008/notes/four/heap.html, diakses pada 15 Desember 2010

[7] "Prim's Algorithm", http://en.wikipedia.org/wiki/Prim's_algorithm, diakses pada 16 Desember 2010.

PERNYATAAN

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

Bandung, 16 Desember 2010

ttd

Muhammad Adinata (13509022)