HEAPSORT ALTERNATIVE SORTING IN CONTIGUOUS LIST

Dian Syahfitra[1], M. Irvan Faradian[2], Ahmad Zufri[3]

Departemen Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung

E-mail : if14021@students.if.itb.ac.id [1],
if14024@students.if.itb.ac.id [2], if14044@students.if.itb.ac.id [3]

Abstrak

Dalam membangun sebuah aplikasi, orang-orang sering kali dihadapi pada masalah pengurutan data pada aplikasi tersebut. Contoh yang paling mudah ialah pada aplikasi yang berhubungan dengan data yang dapat diurutkan, seperti Nomor Induk Mahasiswa (NIM), nilai, Nomor ID sebuah barang inventaris, dan lain sebagainya. Walaupun tersedia banyak sekali teknik pengurutan yang dapat digunakan untuk memecahkan masalah tersebut, misalnya Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, dan yang lainnya, masih dibutuhkan ketepatan untuk memilih jenis algoritma yang akan digunakan karena setiap algoritma pengurutan tersebut memiliki karakteristik yang berbeda-beda, seperti kecepatan, struktur data yang digunakan dalam teknik pengurutan, sampai ruang yang dibutuhkan untuk mengurutkan Dalam membangun sebuah aplikasi, orang-orang sering kali dihadapi pada masalah pengurutan data pada aplikasi tersebut. Contoh yang paling mudah ialah pada aplikasi yang berhubungan dengan data yang dapat diurutkan, seperti Nomor Induk Mahasiswa (NIM), nilai, Nomor ID sebuah barang inventaris, dan lain sebagainya. Walaupun tersedia banyak sekali teknik pengurutan yang dapat digunakan untuk memecahkan masalah tersebut, misalnya Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, dan yang lainnya, masih dibutuhkan ketepatan untuk memilih jenis algoritma yang akan digunakan karena setiap algoritma pengurutan tersebut memiliki karakteristik yang berbeda-beda, seperti kecepatan, struktur data yang digunakan dalam teknik pengurutan, sampai ruang yang dibutuhkan untuk mengurutkan

Kata kunci: *heapsort, pengurutan, waktu asimptotik ,contiguous lits, struktur data, ruang memori*

1. Pendahuluan

Algoritma pengurutan Heap Sort menggunakan struktur data pohon biner yang tidak harus lengkap. Pohon yang digunakan untuk mengurutkan data ini disebut heap tree.

2. Heap dan HeapTree

Algoritma pengurutan Heap Sort menggunakan struktur data pohon biner yang tidak harus lengkap. Pohon yang digunakan untuk mengurutkan data ini disebut heap tree.

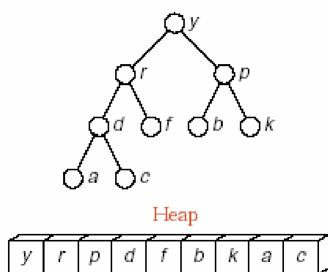


Figure 8.16. A heap as a tree and as a list

3. Heap Sorting

Heapsort adalah salah satu jenis *selection sort*

3.1 Metode

Heapsort memiliki dua fungsi utama yaitu , fungsi untuk menginisialisasi *heap* awal dari list kontigu dengan elemen acak, dan fungsi kedua yaitu fungsi untuk meng-*insert heap* baru dengan mengambil node awal heap dan mempromosikan elemen lain dibawahnya untuk menggantikan posisinya. Kali ini kami fungsi pertama akan dinamakan dengan *BuildHeap()* dan *InsertHeap()* untuk fugnsi kedua.

Untuk fungsi kedua merupakan tahap kedua di mana kita akan mengambil akar pohon *heap* yang merupakan nilai terbesar/terkecil (tergantung penyusunan) , dan nilai ini akan diletakkan di akhir list. Kita akan meyimpan indeks pengulangan dalam var *last_unsorted*

3.2 Algoritma Utama

Berikut ini algoritma utama yang mencakup dua fungsi utama, ditulis dalam bahasa c++ .

```
template <class Record>
void Sortable_list<Record> :: heap_sort( )
/* Post: The entries of the Sortable_list have been rearranged so
that their keys
are sorted into nondecreasing order.
{
Record current;    // temporary storage for moving entries
```

```
int last_unsorted; // Entries beyond last_unsorted have been sorted.
build_heap( ); // First phase: Turn the list into a heap.
        for (last_unsorted = count − 1; last_unsorted > 0;
        last_unsorted−−)
         {
            current = entry[last_unsorted]; // Extract the last entry from the list.
            entry[last_unsorted] = entry[0]; // Move top of heap to the end
            insert_heap(current, 0, last_unsorted − 1); // Restore the heap
         }
}
```

3.3 Fungsi *InsertHeap()*

Pada fungsi insert_heap ini akan melakukan penghapusan sebuah simpul pada heaptree, pemilihan sebuah simpul untuk ditempatkan di posisi yang lebih atas, dan menjaga tree tersebut tetap sebuah heaptree.

Langkah-langkanya ialah :

1. Setiap simpul yang dihapus(*low*) dicari anaknya yang memiliki kunci terbesar/terkecil(*large*)
2. Anak dengan kunci yang lebih besar dipromosikan ke tempat simpul yang di hapus.
3. Langkah 1 dan 2 akan di lakukan berulang kali sampai simpul yang dihapus tidak punya anak lagi atau simpul yang ingin dipromosikan lebih besar/kecil daripada anak dari simpul yang dihapus yang memiliki kunci terbesar/terkecil.

Berikut kode program fungsi *insert_heap()* dalam bahasa c ++ :

```
template <class Record>
void Sortable_list<Record> :: insert_heap(const Record &current, int low, int high)
/* Pre: The entries of the Sortable_list between indices
  low Ç 1 and high, inclusive,
  form a heap. The entry in position low will be discarded.
  Post: The entry current has been inserted into the
  Sortable_list and the entries
  rearranged so that the entries between indices low and
  high, inclusive,
  form a heap.
*/
{
    int large; // position of child of entry[low] with the larger key
    large = 2 * low + 1; // large is now the left child of low.
    while (large <= high) {
    if (large < high && entry[large] < entry[large + 1])
      large++; // large is now the child of low with the largest key.
    if (current >= entry[large]){
      break; // current belongs in position low.
    }
    else { // Promote entry[large] and move down the tree.
      entry[low] = entry[large];
        low = large;
        large = 2 * low + 1;
      }
}
entry[low] = current;
}
```

Berikut ilustrasi pengurutan heap y,r,p,f,k,d,c,b,a,

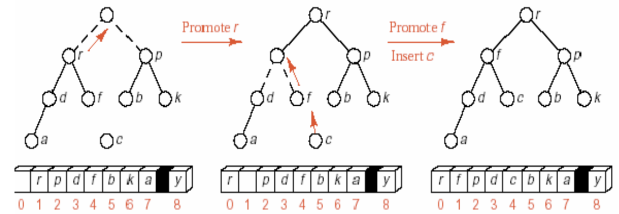
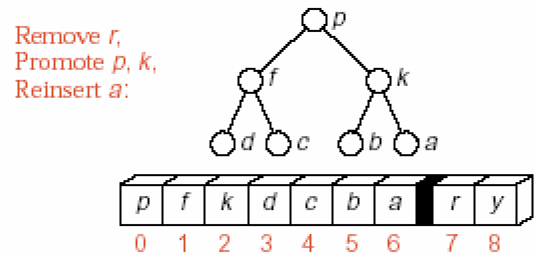
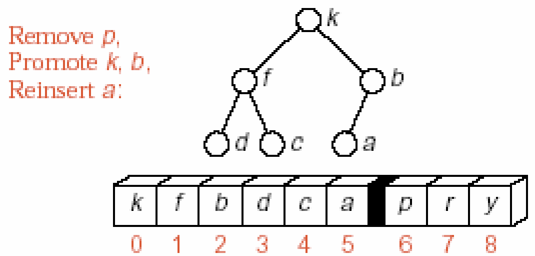
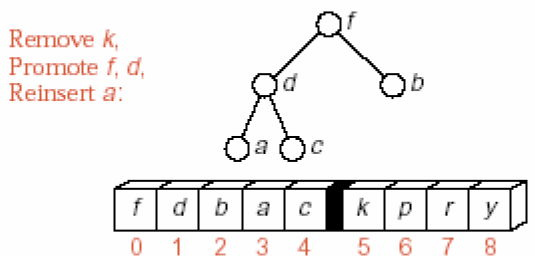
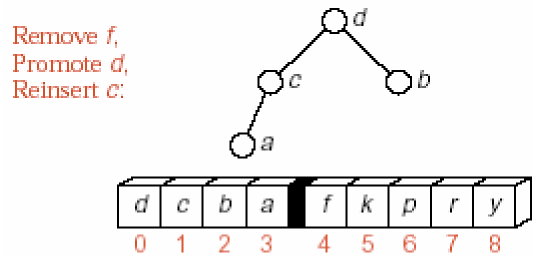


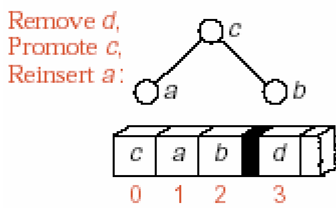
Figure 8.17. First stage of heap_sort

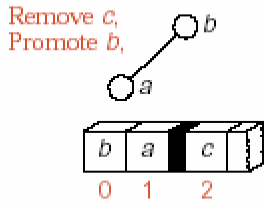


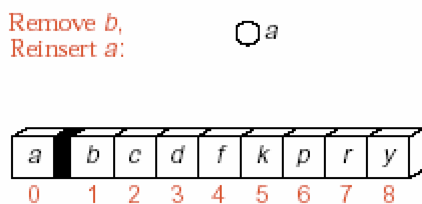












3.4 Fungsi *BuildHeap()*

Pada bagian build heap ini kiat kan menginisialisasi heap awal dari list yang acak. Untuk melakukan hal tersebut pertama-tama ingat bahwa 2 pohon dengan satu simpul secara otomatis, akan memenuhi ketentuan dari sebuah heap, dengan demikian kita tidak perlu khawatir dengan daun dari pohon; yaitu paruh kedua dari list. Jika kita memulai dari tengah list sampai ke awal, kita dapat menggunakan fungsi *insert_heap* untuk memasukkan setiap masukan ke bagian heap yang terdiri dari semua masukan yang masuk kemudian yang kemudian membentuk heap yang sempurna.

Berikut ini fungsi *build_heap* yang ditulis dalam bahasa C++.

```
template <class Record>
void Sortable_list<Record> :: build_heap( )
/* Post: The entries of the Sortable_list have been
rearranged so that it becomes
a heap.
*/
{
  int low; // All entries beyond the position  low form a
heap.
  for (low = count/2 - 1; low >= 0; low--) {
    Record current = entry[low];
    insert_heap(current, low, count - 1);
  }
}
```

4. Kompleksitas dan Analisis Heapsort

4.1 Kompleksitas

Perhitungan Kompleksitas Heapsort akan digunkan kasus terburuk terlebih dahulu. Pada kasus terburuk setiap passing melewati loop, setidaknya nilai *low*+1 akan ganda maka jumlah perlewatan tidak melebihi hg((high+1)/(low+1)). Setiap pass akan melakukan assignment dan dua perbandingan. Maka jumlah perbandingan yang terjadi pada fungsi insertheap() setidaknya 2 lg.(high +1)/(low + 1) dan jumlah assignment lg.(high + 1)/(low + 1).

Ambil m = |½n| (integer tertinggi tidak mencapai ½ n). Dalam fungsi kedua build_heap() kita akan memanggil, untuk nilai k = low berjarak dari m-1 sampai 0. Jadi jumlah operasi perbandingan adalah

$$2\sum_{k=1}^{m} \lg\left(\frac{n}{k}\right) = 2(m\lg n - \lg m!) \approx 5m \approx 2.5n,$$

Dengan pendekatan Stringling* dan lg m = lg n -1 ,
Kita peroleh :
lg*m*! _ *m*lg*m* − 1:5*m* _ *m*lg*n* − 2:5*m*
Dengan sama, di bagian sorting dan insertion . kita dapatkan

$$2\sum_{k=2}^{n} \lg k = 2\lg n! \approx 2n\lg n - 3n$$

Perbandingan .Kita dapat meyimpulkan jumlah dari operasi perbandingan adalah $2n\lg n + O.n.$.

Satu assignment dari setiap masukan dilakukan di fungsi insert_heap setaip 2 perbandingan. Maka total dari assignment adalah n lg n + O(n)

Maka pada kasus terburuk menyusun list dengan panjang n , heapsort melakukan 2n lg n + O(n) perbandingan dari setiap kunci, dan n lg n + O(n) assignment.

4.2 Analisis dan Perbandingan

Berdasarkan kompleksitasnya Heapsort cukup efisien dengan O(n log n) , heapsort akan cupup cepat untuk list yang panjang tetapi tidak untuk list yang pendek.

Heapsort sering dibandingkan dengan quicksort dan margesort. Ketiganya punya batasan yang sama O(n log n) tatapi dengan basis log yang beda , khusus quick sort memiliki kompleksitas O(n^2) untuk kasus terburuk, dari ketiganya hepsort memiliki komplesitas terendah. Pada quicksort kasus rata-rat memiliki komplesitas 1,39 n lg n + O(n) perbandin gan dan 0,69 n lg n + O(n) assignment hal ini lebih baik dari quicksort. Heapsort merupakan sorting yang tidak stabil. Suatu sorting yang stabil akan menangani permintaan relatif dari record dengan kunci yang setar ., yaitu jika suatu algoritma sorting stabil jika ada 2 record R dan S dengan kunci yang sama dan R ada sebelum S di List awal, R akan ada sebelum S juga di list tersort. Algoritma sorting yang tidak stabil dapat mengubah relatif order dari kunci yang sama/setara. Sorting yang tidak satbil dapat diimplementasikan menjadi stabil dengan cara memperlebar kunci perbandingan , sedemikian hingga perbandingan dua objek dengan dua kunci diputuskan dengan menggunakan urutan dari

masukan dalam data asli/awal sebagai tie-breaker .Margesort sendiri memiliki beberapa keungulan dari Heapsort yaitu

- margesort mempertimbangkan data cache performansi lebig dari heapsort, karena akses yang terurut tidak random (heapsort)
- mudah dimengerti
- stabil
- terparalel lebih baik
- Margesort dapat diaplikasikan pada linked-list dengan mudah , list yang besar di simpan di *slow-to-access* media. Heapsort merupakan random access sort dan sedikit locality of reference membuat sort ini lambat terhadap akses media

Berikut perbandingan beberapa algoritma sorting (Wikipedia):

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Bubble sort	O(n)	—	O(n^2)	O(1)	Yes	Exchanging	Times are for best variant
Cocktail sort	O(n)	—	O(n^2)	O(1)	Yes	Exchanging	
Comb sort	O(n log n)	—	O(n log n)	O(1)	No	Exchanging	
Gnome sort	O(n)	—	O(n^2)	O(1)	Yes	Exchanging	
Selection sort	O(n^2)	O(n^2)	O(n^2)	O(1)	No	Selection	
Insertion sort	O(n)	—	O(n^2)	O(1)	Yes	Insertion	
Shell sort	O($n$log(n))	—	O($n\log^2(n)$)	O(1)	No	Insertion	Times are for best variant
Binary tree sort	O($n$log(n))	—	O($n$log(n))	O(1)	Yes	Insertion	
Library sort	O(n)	O($n$log(n))	O(n^2)	O(n)	Yes	Insertion	
Merge sort	O($n$log(n))	—	O($n$log(n))	O(n)	Yes	Merging	
In-place merge sort	O($n$log(n))	—	O($n$log(n))	O(1)	Yes	Merging	Times are for best variant
Heapsort	O($n$log(n))	—	O($n$log(n))	O(1)	No	Selection	
Smoothsort	O(n)	—	O($n$log(n))	O(1)	No	Selection	
Quicksort	O($n$log(n))	O($n$log(n))	O(n^2)	O(log n)	No	Partitioning	Naive variants use O(n) space
Introsort	O($n$log(n))	O($n$log(n))	O($n$log(n))	O(log$n$)	No	Hybrid	
Patience sorting	O(n)	—	O($n$log(n))	O(n)	No	Insertion	Finds all the longest increasing subsequences within O(n log log(n))

5. Kesimpulan

Heapsort adalah suatu alternative sorting dengan komplesitas yang rendah . Heapsort merupakan sorting yang tergolong selection sort karena saat melakukan *insert_heap* kita akan melakukan pemilihan anak man yang akan di promosikan menggantikan *paren*t. Heapsort hanya dapat di gunakan pada list yang kontigu karena sifatnya yang *random access* .

Heapsort menggukan sorting dengan struktur data sebuah heap tree yang di representasi pada list , heaptree merupakan suatu pohon biner yang setiap parent akan memiliki nilai lebih besar/kecil (tergantung kebutuhan) dari anaknya. Begitu list dibentuk heaptreenya ,maka pasti akar adalah nilai terbesar / terkecil dari seluruh list . Akar akan dipindahkan ke belakang list ,dan anak yang terbesar kemudian akan di promosikan sebagai penggantinya. List sisa kan diproses begitu juga sampai terbentuk list yang terurut.
Heapsort me

Daftar Pustaka

1. **http://wikipedia.org/wiki/heapsort** Diakses tanggal 18 Mei 2006 pukul 23.00
2. **http://wikipedia.org/wiki/sorting** Diakses tanggal 18 Mei 2006 pukul 23.00
3. **http://wikipedia.org/wiki/heap** Diakses tanggal 18 Mei 2006 pukul 23.00
4. Kruse L. Robert and Alexander J. Ryba, *Data Stuctured and program Design in C++* Prentice-hall, New Jersey