PENGENALAN BINARY INDEXED TREE DAN APLIKASINYA

Listiarso Wastuargo-13508103

Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
hallucinogenplus@yahoo.co.id

ABSTRAK

Makalah ini membahas tentang *binary indexed tree* dan aplikasinya dalam kehidupan sehari-hari. *Binary indexed tree* adalah suatu struktur data yang digunakan untuk menyimpan frekuensi kumulatif yang diperlukan pada kompresi data *lossless* aritmatik dinamis. Hal ini dilakukan dengan membagi frekuensi kumulatif menjadi sesuatu yang sebanding dengan representasi biner dari indeks suatu tabel. Prosesnya dilakukan dengan cara melakukan iterasi pada indeks secara biner lalu melakukan *update* atau melakukan penjumlahan properti terhadapnya. Dibandingkan dengan teknik-teknik kompresi yang telah ada sebelumnya, *binary indexed tree* adalah teknik yang lebih cepat, lebih kompak dan relatif lebih mudah diimplementasikan. Kompleksitas untuk semua operasi yang diimplementasikan adalah konstan atau logaritmik. Makalah ini juga akan mengilustrasikan *binary indexed tree* sebagai salah satu media pembentuk solusi optimal pada permasalahan yang ada dalam kehidupan sehari-hari.

Kata kunci: kompresi data, *binary indexed tree*, frekuensi kumulatif, kode aritmatik.

1. PENDAHULUAN

Dewasa ini, pengembangan algoritma kompresi yang baik sangatlah vital digunakan untuk memenuhi kebutuhan kecepatan dan kekompakan suatu sistem. Suatu teknik yang

Pada pengembangan algoritma kompresi, ada sebuah masalah umum yang ditemukan, yaitu suatu cara untuk menampung tabel yang berisi frekuensi kumulatif dari sekumpulan data. Telah ditemukan teknik-teknik pengelompokan data, teknik penyimpanan dengan *heap*, dan teknik penyimpanan dengan *splay tree* yang merupakan stuktur data yang telah di optimasi untuk menghitung frekuensi kumulatif dari kumpulan data.

Pada makalah ini, akan dibahas sebuah teknik kompresi *lossless* yang menyimpan frekuensi kumulatif dari sekumpulan data yang hanya menggunakan sebuah tabel sederhana dengan kompleksitas yang relative rendah dengan tanpa pemindahan atau pereorganisasian data.

2. DASAR-DASAR TEORI

2.1 Kompresi Data

Pada *computer science* dan teori informasi, kompresi data atau *source coding* adalah proses meng-*encode* informasi menggunakan bit yang lebih kecil (atau satuan penyimpan informasi lain) daripada menggunakan representasi yang tidak ter-*encode*, dengan menggunakan teknik *encoding* yang spesifik.

Seperti semua jenis komunikasi lain, komunikasi data yang telah dikompresi hanya bekerja bila kedua belah pihak (pengirim dan penerima) mengerti skema *encoding* yang dipakai. Sebagai contoh, teks ini hanya mungkin karena teks ini ditujukan untuk diinterpretasikan sebagai karakter dalam bahasa Indonesia. Sama halnya dengan data. Data yang telah dikompresi hanya akan bisa dipahami bila metode *decoding* diketahui oleh penerima.

Kompresi sangatlah berguna karena dapat membantu mengurangi konsumsi sumber daya, seperti *hard disk* atau *bandwith*. Di lain pihak, data yang telah dikompresi harus didekompres agar bisa digunakan, dan proses ektra ini memegang peran sangat penting untuk beberapa aplikasi. Sebagai contoh, skema kompresi pada video mungkin membutuhkan *hardware* yang mahal agar video yang harus didekompres cukup cepat agar bisa dilihat sembari video didekompres (pilihan untuk mendekompresi video sebelum dilihat tidak terlalu nyaman digunakan dan membutuhkan tempat penyimpanan untuk video yang telah didekompres). Sehingga desain skema kompresi data harus melibatkan sisi baik-buruk banyak faktor, termasuk derajat kompresi, banyak distorsi yang terlibat dan komputasi sumber daya yang dibutuhkan untuk melakukan kompresi dan dekompresi data.

Jenis kompresi menurut faktor distorsi data yang terlibat terbagi menjadi dua bagian yaitu kompresi data yang *lossless* dan kompresi data yang *lossy*. Disini hanya akan dibahas kompresi data *lossless* yang akan berhubungan langsung dengan *binary indexed tree*.

2.2 Kompresi Data *Lossless*

Kompresi data *lossless* adalah kompresi data yang yang memanfaatkan pola pada data sehingga dapat merepresentasikan data lebih kompak tapi tanpa error. Kompresi *lossless* mungkin dilakukan karena kebanyakan data pada dunia-nyata memiliki statistik redundan. Sebagai contoh, huruf 'e' akan lebih sering daripada huruf 'z'.

Keuntungan penggunaan kompresi *lossless* adalah data kompresi yang didapatkan dapat didekompress tanpa error. Sedangkan kerugiannya adalah, kompresi jenis ini relatif lebih tidak kompak bila dibandingkan kompresi *lossy*.

2.3 Kode Aritmatik

Kode aritmatik adalah suatu metode untuk melakukan kompresi data *lossless*. Normalnya, sekelompok karakter seperti kata "hello world" dapat direpresentasikan menggunakan jumlah bit yang sama per karakternya, seperti kode ASCII. Seperti kode Huffman, kode aritmatik adalah bentuk enkoding entropi panjang variabel yang mengubah string yang merepresentasikan bit kecil untuk karakter yang sering muncul, dan bit besar untuk karakter yang lebih jarang muncul dengan target mencari bit total yang lebih kecil. Berbeda dengan teknik enkoding entropi yang lain yang memecah pesan masukan menjadi komponen-komponennya dan mengganti komponen-komponen tersebut menjadi kode kata, kode aritmatik meng-*encode* seluruh pesan menjadi satu bilangan, pecahan n dimana ($0.0 \leq n < 1.0$)

3. *BINARY INDEXED TREE*

3.1 Ide dasar

Setiap bilangan bulat dapat direpresentasikan dalam penjumlahan dari bilangan dua pangkat. Dengan cara yang sama, frekuensi kumulatif dapat direpresentasikan sebagai jumlah dari sekelompok subfrekuensi. Pada kasus *binary indexed tree* setiap kelompok mengandung beberapa bilangan frekuensi berurutan yang tidak saling *overlap*.

Untuk kenyamanan, akan dijelaskan beberapa variable yang akan digunakan disini dan seterusnya. **Idx** adalah indeks dari *binary indexed tree*. **R** adalah posisi angka satu yang paling tidak signifikan pada pada representasi biner angka **idx**. **Tree[idx]** adalah jumlah dari frekuensi dari indeks (**idx** – 2^r **+** 1) hingga indeks **idx**. Kita juga akan menulis **idx** bertanggung jawab atas indeks (**idx** – 2^r **+** 1) hingga indeks **idx** (tanggung jawab setiap indeks disini adalah kunci dari algoritma *binary indexed tree* dan adalah cara untuk memanipulasi pohon).

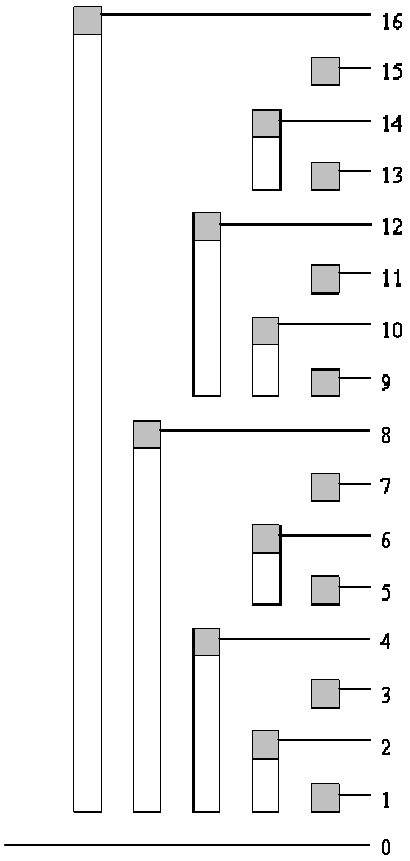
Perhatikan contoh tabel frekuensi, frekuensi kumulatif dan isi tabel yang merepresentasikan isi pohon pada *binary indexed tree* yang memiliki 16 elemen.

Tabel 3.1 Tabel frekuensi, frekuensi kumulatif dan isi pada representasi *binary indexed tree.*

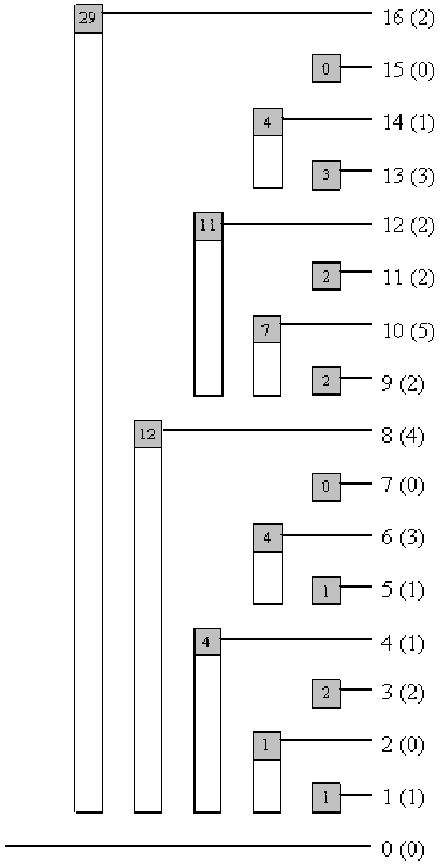
	1	2	3	4	5	6	7	8
f	1	0	2	1	1	3	0	4
c	1	1	3	4	5	8	12	14
tree	1	1	2	4	1	4	0	12
	9	10	11	12	13	14	15	16
f	2	5	2	2	3	1	0	2
c	14	19	21	23	26	27	27	29
tree	2	7	2	11	3	4	0	29

Tabel 3.2 Tabel pertanggung jawaban

	1	2	3	4	5	6	7	8
tree	1	1..2	3	1..4	5	5..6	7	1..8
	9	10	11	12	13	14	15	16
tree	9	9..10	11	9..12	13	13..14	15	1..16



Gambar 3.1 Pohon pertanggungjawaban indeks (panjang kotak menunjukkan frekuensi yang terakumulasi pada elemen teratas)



Gambar 3.2 Pohon dengan frekuensi kumulatifnya

3.2 Isolasi Digit 1 Paling Tidak Signifikan

Ada sebuah cara yang efisien untuk mendapatkan digit satu paling tidak signifikan pada representasi biner sebuah nomer. Anggap **num** adalah bilangan bulat yang akan kita cari digit 1 paling tidak signifikannya. Pada representasi biner, **num** bisa ditulis sebagai **a1e**, dimana **a** merepresentasikan bilangan biner sebelum angka 1 paling tidak signifikan dan **e** merepresentasikan angka nol setelah digit 1 tersebut.

Bilangan **–num** bila ditulis dalam representasi biner akan sama dengan

$$-num = \bar{a}0\bar{e} + 1 \qquad (1)$$

dengan **e** adalah 0 semua dan **ē** adalah 1 semua. Sehingga kita mendapatkan

$$-num = \bar{a}0\bar{e} + 1 = \bar{a}0(1..1) + 1 = \bar{a}0(0..0) = \bar{a}1e \qquad (2)$$

sehingga dengan mudah kita dapat mengisolasi bilangan 1 yang paling tidak signifikan dengan melakukan operasi bit **dan** untuk kedua bilangan.

$$\bar{a}1e \;\&\; a1e = (0..0)\ 1\ (0..0) \qquad (3)$$

3.3 Membaca frekuensi kumulatif

Kita dapat membaca frekuensi kumulatif pada indeks **idx** dengan menjumlahkan **tree[idx]**, mengurangi digit 1 paling tidak signifikan pada **idx** dengan dirinya sendiri dan mengulanginya selama **idx** lebih besar dari 0. Pseudocodenya kira-kira seperti berikut

```
sum ← 0
while (idx > 0)
      sum ← sum + tree[idx]
      idx ← idx – (idx and –idx)
end while
→ sum
```

Contoh untuk **idx** = 11

Tabel 3.3 Tabel pengambilan nilai

iterasi	idx	Posisi digit 1 paling tidak signifikan	idx dan -idx	sum
1	11=1011	0	1 (2^0)	2
2	10=1010	1	4(2^2)	9
3	8=1000	3	8(2^3)	21
4	0=0	---	---	---

Kompleksitas waktunya adalah O(log N)

3.4 Melakukan *update* frekuensi kumulatif

Seperti yang kita ketahui, *binary indexed tree* adalah struktur data kompresi yang dinamis sehingga dapat dilakukan pembacaan dan *update* pada saat struktur data telah selesai dibangun dan dengan kompleksitas yang relatif rendah.

Konsepnya adalah melakukan *update* terhadap indeks pohon yang bertanggung jawab terhadap indeks yang akan diubah nilainya. Saat membaca frekuensi kumulatif pada suatu indeks, kita mengurangi digit 1 paling tidak signifikan bilangan tersebut selama bilangan itu masih lebih besar dari 0. Pada saat mengganti frekuensi sebesar **val** pada suatu pohon, kita akan menambahkan nilai pada indeks saat ini sebesar **val**, lalu menambahkan digit 1 paling tidak signifikan pada bilangan indeks ke bilangan itu sendiri. Proses ini akan terus diulang selama indeks yang diperoleh kurang dari atau sama dengan besar pohon tersebut. Pseudocodenya akan menjadi seperti ini :

```
while (idx < TreeSize)
      tree[idx] ← tree[idx] + val
      idx ← idx + (idx and –idx)
end while
```

Contoh untuk **idx** = 11

Tabel 3.3 Tabel *update* nilai

iterasi	idx	Posisi digit 1 paling tidak signifikan	idx dan -idx
1	11=1011	0	1 (2^0)
2	12=1100	2	4(2^2)
3	16=10000	4	16(2^4)
4	32 = 10000	---	---

Kompleksitas waktunya adalah O(log N)

3.5 Membaca nilai suatu titik atau suatu daerah

Pembacaan nilai pada suatu titik atau suatu daerah dapat dilakukan dengan melakukan pengurangan terhadap selang daerah tersebut.

Sebagai contoh bila kita ingin mencari nilai dari indeks ke-13, kita dapat dengan mudah menyimpulkan bahwa :

$$\mathbf{f[13] = c[13] - c[12]} \qquad \mathbf{(4)}$$

Dimana kita dapat menghitung **c[13]** dan **c[12]** dengan mudah menggunakan *binary indexed tree*.

Sama halnya apabila kita akan membaca nilai suatu selang daerah, sebagai contoh nilai dari indeks ke-7 hingga indeks ke-11. Kita dapat dengan mudah mendapatkan bahwa

$$\mathbf{f[7..11] = c[11] - c[6]} \qquad \mathbf{(5)}$$

Dengan **c[11]** dan **c[6]** dapat dengan mudah dicari dengan menggunakan *binary indexed tree*.

3.6 *Binary indexed tree* 2 Dimensi (*quadratic binary indexed tree*)

Binary indexed tree juga dapat digunakan pada struktur data multi dimensi. Cara melakukan pembacaan dan *update* masih sama dengan cara 1 dimensi, tetapi bedanya, proses ini dilakukan dalam loop. Akan lebih mudah bila langsung saya tunjukkan pseudocodenya (anggap **idy** adalah indeks baris dan **idx** adalah indeks kolom serta **y** adalah variabel penampung kolom sementara).

```
sum ← 0
while (idx > 0)
      y ← idy
      while (y > 0)
            sum ← sum + tree[idx][y]
            y ← y - (y and -y)
      end while
      idx ← idx - (idx and -idx)
end while
→ sum
```

Dan analogi yang sama dilakukan pada proses *update*.

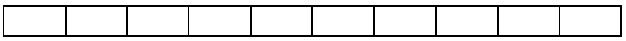
```
while (idx < TreeSize)
      y ← idy
      while (y < TreeSize)
            tree[idx][y] ← tree[idx][y]
+ val
            y ← y + (y and -y)
      end while
      idx ← idx + (idx and -idx)
end while
```

Binary indexed tree juga dapat digunakan untuk struktur data multidimensional.

4. APLIKASI *BINARY INDEXED TREE*

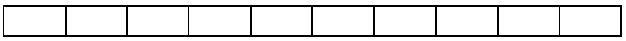
4.1 Sistem Lampu

Misalnya kita diberikan sebaris lampu yang menyala yang bisa menyala dan mati bila diberikan perintah-perintah. Perintah yang diberikan berupa perintah untuk menyalakan atau mematikan sekumpulan lampu yang berurutan sekaligus. Simak ilustrasi berikut.



Gambar 4.1 Urutan lampu yang menyala semua

Akan diberikan beberapa pertanyaan yang harus dijawab. Pertanyaannya adalah apakah lampu ke-i menyala? Mari kita lihat contoh permasalahannya.

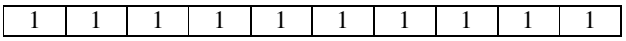


Gambar 4.2 Urutan lampu yang menyala semua mula-mula

Gambar 4.3 Urutan lampu ketika diberi perintah untuk mematikan lampu 2 hingga lampu 7

Pada permasalahan diatas, user telah melakukan *switch* terhadap lampu ke 2 dan ke 7. Lalu user bertanya, apakah lampi ke 7 menyala? Tentu permasalahan ini terlihat sangat simpel, tetapi coba bayangkan bila terdapat jutaan lampu dan operasi yang dilakukan bisa mencapai ribuan. Tentu diperlukan sebuah program yang cukup efisien untuk menghadapi permasalahan ini.

Cara yang paling pertama terpikirkan adalah dengan cara *brute force*, yaitu dengan melakukan update terhadap semua range ketika diberi perintah untuk melakukan update terhadap sekumpulan lampu.



Gambar 4.4 Urutan lampu yang menyala semua

1	0	0	0	0	0	0	1	1	1

Gambar 4.5 Urutan lampu ketika diberi perintah untuk mematikan lampu 2 hingga lampu 7

Angka satu menandakan lampu tersebut menyala dan angka 0 menandakan bahwa lampu tersebut mati.

Kita dapat melakukan sedikit pengamatan bahwa sebenarnya kita bisa menganggap lampu menyala sebagai bilangan ganjil dan lampu yang mati sebagai bilangan genap. Dari sudut pandang ini, kita bisa menganggap perintah menyala dan mematikan lampu sebagai penambahan angka 1 pada lampu pertama dan pengurangan angka 1 pada satu lampu setelah lampu terakhir.

1	-2	2	4	8	2	6	3	7	5

Gambar 4.5 Cara lain memandang nyala dan matinya lampu sebagai bilangan genap dan ganjil

Setelah melakukan pengamatan ini, kita dapat menggunakan *binary indexed tree* untuk menyimpan frekuensi kumulatif dari perintah dan menghasilkan kompleksita O(log N) pada saat *update* dan O(log N) pada saat menjawab pertanyaan.

Perbandingan kompleksitas kedua algoritma ini adalah

Tabel 4.1 perbandingan algoritma sistem lampu

Algoritma	Aksi	
	Update	Jawab
Brute Force	O(1)	O(N)
Binary Indexed Tree	O(log N)	O(log N)

4.2 Sistem Laporan Pemancar

Kita dapat menggunakan *binary indexed tree* dalam mengimplementasikan sistem laporan pemancar.

Diketahui sebuah area yang beroperasi seperti berikut. Area tersebut dibagi menjadi persegi-persegi yang dimana hanya terdapat satu pemancar dalam satu persegi. Jumlah telepon genggam yang aktif pada suatu daerah jangkauan pemancar dapat berubah karena telepon genggam tersebut dapat berpindah tempat dari daerah pemancar satu ke daerah pemancar lain (karena pengguna telepon genggam tersebut dapat pindah dari satu tempat ke tempat lain) atau karena telepon genggam itu tidak aktif. Pada waktu-waktu tertentu, tiap pemancar melaporkan jumlah telepon genggam yang aktif. Simak ilustrasi berikut.

10	7	7	1	2	3	0
2	5	5	0	0	0	0
1	6	12	4	1	21	3
2	11	2	13	5	4	6
3	4	2	5	91	8	7
4	5	6	7	8	9	0
7	6	5	4	3	1	27

Gambar 4.6 Ilustrasi sistem laporan pemancar mula-mula

10	7	7	1	2	3	0
2	5	5	0	0	5	0
1	6	12	4	1	21	3
2	11	2	13	5	4	6
3	4	2	5	91	3	7
4	5	6	7	8	9	0
7	6	5	4	3	1	27

Gambar 4.7 Ilustrasi opearsi yang mungkin terjadi pada sistem laporan pemancar

Penjelasan gambar diatas : terdapat sebuah area yang dibagi menjadi persegi 7x7 yang artinya area tersebut terbagi menjadi 49 buah pemancar yang tersusun seperti gambar tersebut. Angka yang ada didalam persegi masing-masing mencerminkan banyak telepon genggam yang aktif yang ada pada area suatu pemancar.

Pada ilustrasi selanjutnya adalah operasi-operasi yang mungkin ada pada suatu pemancar. Persegi yang berwarna jingga berarti terdapat penambahan jumlah telepon genggam yang aktif pada daerah tersebut. Sedangkan persegi yang berwarna merah berarti terdapat pengurangan jumlah telepon genggam yang aktif pada daerah tersebut. Angka yang ada didalam persegi tetap mencerminkan jumlah telepon genggam yang aktif dalam area pemancar tersebut.

Lalu bagaimana cara membuat sebuah program yang menerima laporan dari banyak pemancar dan sekaligus dapat menjawab *query* tentang jumlah telepon genggam yang aktif pada suatu rentang area yang berbentuk persegi panjang (katakanlah dari persegi (2,3) hingga persegi (5,5) seperti ilustrasi gambar dibawah)?

(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)
(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)
(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)
(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)
(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)
(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)
(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)

Gambar 4.8 Representasi daerah laporan menjadi matriks

Cara yang paling naif adalah dengan menyimpan laporan pada sebuah matriks yang ditentukan jumlahnya dan menjawab *query* dengan menghitung area yang ditanyakan dengan menghitung satu persatu persegi dari persegi (2,2) hingga persegi (5,5). Tidak ada yang salah dengan cara ini, tapi mari kita kaji kompleksitasnya.

Melakukan *update* isi persegi matriks setelah mendapatkan laporan memiliki kompleksitas O(1). Menjawab *query* dapat dilakukan dengan menjumlahkan satu persatu isi matriks dari persegi awal hingga akhir Algoritma ini memiliki kompleksitas $O(N^2)$.

Dari sistem penjumlahan matriks satu persatu pada saat menjawab *query*, kita dapat melakukan suatu pengamatan terhadap algoritma bahwa dengan melakukan penghitungan frekuensi kumulatif pada saat update, kita bisa melakukan peningkatan algoritma kita menjadi O(N) saat mendapatkan laporan dan O(N) saat menjawab *query* yaitu dengan cara mengembalikan selisih frekuensi pada suatu dua kolom tertentu pada matriks sebanyak baris yang dibutuhkan.

10	17	24	25	27	30	30
2	7	12	12	12	12	12
1	7	19	23	24	45	48
2	13	15	28	33	37	43
3	7	9	14	105	113	120
4	9	15	22	30	39	39
7	13	18	22	25	26	53

Gambar 4.9 Ilustrasi matriks frekuensi kumulatif

Tentu kita dapat melakukan simplifikasi algoritma bila kita melakukan penyimpanan frekuensi kumulatif ini dengan menggunakan *binary indexed tree* setelah mencapai pengamatan pada tahap ini dan mencapai kompleksitas *update* O(log N) dan kompleksitas menjawab *query* O(N log N).

Perhatikan bahwa ada bentuk *binary indexed tree* untuk struktur dua dimensi. Maka struktur data kita akan berubah lagi menjadi $O(\log^2 N)$ untuk *update* dan $O(\log^2 N)$ untuk menjawab *query*.

Mari kita lihat perbandingan algoritma dari hasil pengamatan pertama hingga saat ini.

Tabel 4.2 perbandingan algoritma sistem laporan pemancar

Algoritma	Aksi	
	Update	Jawab
Naif	O(1)	$O(N^2)$
Pengamatan 1	O(N)	O(N)
Pengamatan 1 + *Binary Indexed Tree*	O(log N)	O(N log N)
Quadratic Binary Indexed Tree	$O(\log^2 N)$	$O(\log^2 N)$

Kita dapat melihat bahwa *binary indexed tree* dapat membantu kita melakukan teknologi kompresi data dari kuadratik hingga ke tingkat logaritmik.

4. KESIMPULAN

Binary indexed tree adalah salah satu struktur data kompresi *lossless* terbaik saat ini dan dapat diterapkan pada banyak permasalahan yang ada di dunia nyata.

REFERENSI

[1] Fenwick, Peter M., "A New Data Structure for Cumulative Frequency Tabel", Software-Practice and Experience, 24, 3, 1994, 327-336.

[2] http://en.wikipedia.org/wiki/Data_compression 18 Desember 2010

[3] http://en.wikipedia.org/wiki/Arithmetic_coding 18 Desember 2010

[4] http://en.wikipedia.org/wiki/Binary_indexed_tree 18 Desember 2010

[5] Anonymous, "IOI 2001 Problem : Mobiles", IOI Problem Set.

[6] http://www.topcoder.com/tc?module=Static&d1=tutorials &d2=binaryIndexedTrees#prob 18 Desember 2010