TUGAS AKHIR – KI1502

DESAIN DAN ANALISIS SPLAY TREE PADA PENYELESAIAN PERSOALAN SPOJ KLASIK 19543

M. Kemal Darmawan
NRP. 5111 100 173

Dosen Pembimbing 1
Arya Yudhi Wijaya, S.Kom., M.Kom.

Dosen Pembimbing 2
Rully Soelaiman, S.Kom., M.Kom.

JURUSAN TEKNIK INFORMATIKA
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember
Surabaya 2015

FINAL PROJECT – KI1502

DESIGN AND ANALYSIS SPLAY TRE FOR SOLVING SPOJ CLASSIC PROBLEM 19543

M. Kemal Darmawan
NRP. 5111 100 173

Supervisor 1
Arya Yudhi Wijaya, S.Kom., M.Kom.

Supervisor 2
Rully Soelaiman, S.Kom., M.Kom.

DEPARTMENT OF INFORMATICS
Faculty of Information Technology
Sepuluh Nopember Institute of Technology
Surabaya 2015

LEMBAR PENGESAHAN

DESAIN DAN ANALISIS SPLAY TREE PADA PENYELESAIAN PERSOALAN SPOJ KLASIK 19543

TUGAS AKHIR

Diajukan Untuk Memenuhi Salah Satu Syarat
Memeroleh Gelar Sarjana Komputer
pada
Bidang Studi Dasar dan Terapan Komputasi
Program Studi S-1 Jurusan Teknik Informatika
Fakultas Teknologi Informasi
Institut Teknologi Sepuluh Nopember

Oleh:
M. Kemal Darmawan
NRP. 5111100173

Disetujui oleh Pembimbing Tugas Akhir:

1. Arya Yudhi Wijaya, S.Kom., M.Kom ..... (Pembimbing 1)
   (NIP. *198409042010121002*)
2. Rully Soelaiman, S.Kom., M.Kom. ..... (Pembimbing 2)
   (NIP. *197002131994021001*)

SURABAYA
Juli, 2015

DESAIN DAN ANALISIS SPLAY TREE PADA PENYELESAIAN PERSOALAN SPOJ KLASIK 19543

Nama : **Muhammad Kemal Darmawan**
NRP : **5111100173**
Jurusan : **Teknik Informatika**
Fakultas Teknologi Informasi ITS
Dosen Pembimbing I : **Arya Yudhi Wijaya, S.Kom., M.Kom.**
Dosen Pembimbing II : **Rully Soelaiman, S.Kom., M.Kom.**

ABSTRAK

Diberikan sejumlah barisan bilangan integer non negatif awal dan diberikan sejumlah operasi untuk memodifikasi barisan bilangan tersebut atau melakukan perhitungan pada barisan bilangan tersebut. Operasi-operasi tersebut ialah penyisipan bilangan, penghapusan bilangan, perubahan bilangan dan query dengan jarak tertentu pada bilangan.

Splay Tree merupakan salah satu struktur data binary search tree yang menerapkan konsep self balancing. Tree ini menggunakan cara mendekatkan node-node yang sering diakses menuju root untuk mengefisienkan waktu pengaksesan, cara tersebut dinamakan operasi splay. Splay melakukan perpindahan suatu node menuju root pada tree.

Pada penelitian ini akan didesain dan diimplementasikan solusi untuk permasalahan query barisan dinamis dengan menggunakan Splay Tree. Penggunaan Splay Tree ditujukan untuk merepresentasikan barisan bilangan yang diberikan dan menangani operasi-operasi yang ditujukan pada barisan tersebut.

Solusi yang dikembangkan berjalan dengan kompleksitas waktu $O(M \log N)$, di mana M adalah jumlah operasi dan N ada jumlah node pada tree dan solusi mendapat umpan balik dari persoalan SPOJ 19543-GSS8, Can you answer these queries VIII

adalah Accepted dengan waktu rata-rata 18.16 detik dan memori 25 MB dari 20 kali pengumpulan.

Kata kunci: Splay Tree, Self Balancing Binary Search Tree, Query Barisan Dinamis.

DESIGN AND ANALYSIS SPLAY TREE DATA STRUCTURE FOR SOLVING SPOJ CLASSIC PROBLEM 19543

Name	**: Muhammad Kemal Darmawan**
NRP	**: 5111100173**
Department	**: Department of Informatics**
	Faculty of Information Technology ITS
Supervisor I	**: Arya Yudhi Wijaya, S.Kom., M.Kom.**
Supervisor II	**: Rully Soelaiman, S.Kom., M.Kom.**

ABSTRACT

Given a number of initial non negative integers and a number of operations to modify the array or to calculate the array. The operations consist of insertion of number, deletion of number, replacement of number and calculation of range in array.

Splay Tree is one of the binary search tree data structures which holds the concept of self balancing. This tree makes the frequently accessed node become near to the root of the tree for resulting the efficiency of the access, that operation is called Splay operation. Splay operation moves a node to the root.

In this undergraduate thesis, the problem of dynamic range query will be designed and be implemented by using Splay Tree. The purpose of use of Splay Tree is to represent the given array of numbers and handle or maintain the operations that is given for the array.

The time complexity of the solution is $O(M \log N)$, where M is the number of operation and N is the number of node in the tree and the solution got Accepted feedback from problem in SPOJ 19543-GSS8, Can you answer these queries VIII with 18.16 seconds as average time and 25 MB as memory from 20 submissions.

Key words: Splay Tree, Self Balancing Binary Search Tree, Dynamic Range Query.

KATA PENGANTAR

Puji syukur kepada Allah Yang Maha Esa atas segala karunia dan rahmat-Nya penulis dapat menyelesaikan penelitian yang berjudul:

DESAIN DAN ANALISIS SPLAY TREE PADA PENYELESAIAN PERSOALAN SPOJ KLASIK 19543

Penelitian ini dilakukan untuk memenuhi salah satu syarat memperoleh gelar Sarjana Komputer di Jurusan Teknik Informatika Fakultas Teknologi Informasi Institut Teknologi Sepuluh Nopember.

Dengan selesainya penelitian ini saya harap apa yang telah saya kerjakan dapat memberikan manfaat bagi perkembangan ilmu pengetahuan serta bagi saya sendiri selaku penulis penelitian ini.

Saya selaku penulis mengucapkan terima kasih kepada semua pihak yang telah memberikan dukungan baik secara langsung maupun tidak langsung selama saya menyelesaikan penelitian antara lain:

1. Allah SWT atas segala nikmat dan rahmat yang telah diberikan selama ini.
2. Ayah, ibu dan keluarga penulis yang tiada henti-hentinya mencurahkan kasih sayang, perhatian dan doa kepada penulis selama ini.
3. Bapak Rully Soelaiman, S.Kom., M.Kom. selaku dosen pembimbing yang telah sabar memberikan bimbingan, motivasi, nasihat dan meluangkan waktu yang sangat berharga bagi penulis untuk membantu pengerjaan penelitian ini.
4. Bapak Arya Yudhi Wijaya, S.Kom., M.Kom. selaku dosen pembimbing yang telah memberi nasihat, arahan dan bantuan sehingga penulis dapat menyelesaikan penelitian ini.

5. Seluruh teman saya dalam menempuh perkuliahan di Jurusan Teknik Informatika Fakultas Teknologi Informasi Institut Teknologi Sepuluh Nopember
6. Seluruh pihak yang tidak bisa saya sebutkan satu persatu yang telah memberikan dukungan yang telah memberikan dukungan selama saya menyelesaikan penelitian.

Saya mohon maaf apabila terdapat kekurangan dalam penelitian ini. Kritik dan saran saya harapkan untuk perbaikan dan pembelajaran di kemudian hari. Semoga penelitian ini dapat memberikan manfaat yang sebaik-baiknya.

Surabaya, Juli 2015

Penulis

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

DAFTAR KODE SUMBER

BAB I
PENDAHULUAN

Bagian ini akan dijelaskan hal-hal yang menjadi latar belakang, permasalahan yang dihadapi, batasan masalah, tujuan dan manfaat, metodologi dan sistematika penulisan yang digunakan dalam pembuatan penelitian.

1.1 Latar Belakang

Permasalahan *query* barisan dinamis adalah salah satu masalah dengan penyelesaian yang sangat bergantung akan struktur data yang digunakan. Permasalahan tersebut ialah suatu barisan yang dilakukan beberapa operasi, di mana operasi tersebut meliputi operasi perhitungan pada suatu jarak pada barisan dan operasi operasi-operasi yang membuat barisan tersebut menjadi dinamis. Struktur data di sini berperan untuk memodelkan barisan seefektif mungkin untuk dilakukan operasi-operasi terhadap barisan tersebut. Struktur data yang digunakan juga harus memiliki kompleksitas waktu yang mendukung untuk menangani operasi dengan jumlah yang banyak.

Salah satu penyelesaian *query* barisan dinamis ialah dengan struktur data *binary search tree*, namun dengan hanya mengimplementasikan *binary search tree* yang biasa tidak memungkinkan untuk menangani permasalahan secara efisien dari sisi kompleksitas waktu. Pada penelitian ini akan diimplementasikan solusi untuk permasalahan query barisan dinamis dengan struktur data dengan konsep *binary search tree* dengan lebih mengefisienkan kompleksitas waktu yaitu *self balancing binary search tree.* Struktur data tersebut tetap menjaga kedalaman pada suatu *binary search tree* untuk membuat kompleksitas waktu untuk setiap operasi lebih efisien.

1.2 Rumusan Masalah

Rumusan masalah yang diangkat dalam penelitian ini adalah sebagai berikut:

1. Merepresentasikan permasalahan *dynamic range query* menjadi permasalahan *self balancing binary search tree*.
2. Menyusun perancangan algoritma dan implementasi metode permasalahan *self balancing binary search tree* dalam menyelesaikan permasalahan *dynamic range query*.
3. Mengimplementasikan *data generator* untuk pengujian kinerja.
4. Membuat skenario pengujian untuk mengetahui kebenaran dan efisiensinya.

1.3 Batasan Masalah

Permasalahan yang dibahas pada penelitian ini memiliki beberapa batasan, yaitu sebagai berikut:.

1. Implementasi dilakukan dengan bahasa pemrograman C++.
2. Batasan masalah sesuai dengan batasan-batasan yang terdapat pada soal SPOJ 19543-GSS8, *Can you answer these queries VIII.*

1.4 Tujuan

Tujuan dari penelitian ini adalah sebagai berikut:

1. Melakukan analisis dan mereduksi permasalahan *dynamic range query* menjadi permasalahan *self balancing binary search tree*.
2. Melakukan perancangan algoritma dan implementasi metode *self balancing binary search tree.*
3. Membuat *data generator* untuk melakukan uji coba kinerja.

4. Mengevaluasi hasil dan kinerja algoritma yang telah dirancang pada permasalahan *self balancing binary search tree.*

1.5 Metodologi

Metodologi yang dilakukan pada pengerjaan penelitian ini memiliki beberapa tahapan, yaitu sebagai berikut:

1. Penyusunan proposal penelitian
   Tahap awal untuk memulai pengerjaan penelitian adalah penyusunan proposal penelitian. Pada proposal tersebut, penulis mengajukan gagasan desain dan analisis struktur data *self balancing binary search tree* pada penyelesaian persoalan SPOJ klasik 19543 GSS8.
2. Studi literatur
   Pada tahap ini dilakukan pencarian informasi dan studi literatur yang diperlukan untuk penyelesaian persoalan yang akan dikerjakan. Informasi didapatkan dari materi-materi yang berhubungan dengan algoritma yang digunakan dalam pengerjaan penelitian ini, materi-materi tersebut didapatkan dari buku acuan maupun internet.
3. Implementasi perangkat lunak
   Implementasi merupakan tahap untuk membangun algoritma yang digunakan. Pada tahap ini dilakukan implementasi dari rancangan algoritma *self balancing binary search tree* yang direpresentasikan sesuai dengan permasalahan dan diterapkan pada sebuah program. Implementasi ini dilakukan dengan menggunakan bahasa pemrograman.
4. Pengujian dan evaluasi
   Pada tahap ini dilakukan uji coba dengan menggunakan kasus uji yang dibuat dan juga dilakukan percobaan pada *online judge* SPOJ sesuai dengan permasalahan yang terkait apakah

telah sesuai dengan keluaran yang seharusnya. Pada tahap ini evaluasi sekaligus dilakukan pada *online judge* SPOJ. Perbaikan akan dilakukan hingga hasil evaluasi yang diberikan oleh *online judge* tersebut adalah diterima.

5. Penyusunan buku penelitian
   Dalam tahap akhir ini, penulis melakukan penyusunan laporan yang berisikan dokumen pembuatan dan hasil pengerjaan pada perangkat lunak yang telah dibuat.

BAB II
DASAR TEORI

Bab ini akan membahas mengenai dasar teori dan literatur yang menjadi dasar pengerjaan penelitian.

2.1 Definisi Umum

Subbab ini akan dibahas definisi-definisi umum yang selanjutnya akan sering digunakan dalam buku ini.

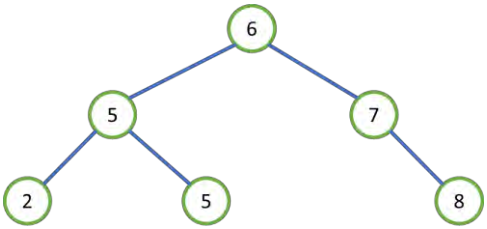
Suatu graf adalah himpunan objek yang disebut simpul (*vertex* atau *node*) yang terhubung oleh sisi (*edge*). Pohon (*tree*) merupakan salah satu jenis graf yang tidak mengandung *cycle* atau suatu lintasan yang berawal dan berakhir pada *node* yang sama di mana setiap *edge* yang dilalui tepat satu kali. Sebuah pohon biner (*binary tree*) adalah sebuah struktur data *tree* di mana setiap *node* memiliki paling banyak dua anak (*child*). Secara khusus anaknya dinamakan anak kiri (*left child*) dan anak kanan (*right child*). Beberapa definisi lain yang perlu diketahui antara lain:

1. *Parent u* ialah *node* yang berhubung langsung dengan *node u* dan membuat *node u* menjadi *child* dari *node* tersebut, baik itu *left child* ataupun *right child*.
2. *Root* ialah *node* pada *tree* yang tidak memiliki *node* lain sebagai *parent.*
3. *Depth* atau level dari suatu *node* yang diukur dari jarak *node* tersebut pada *root.*
4. *Subtree u* adalah bagian *tree* yang menempatkan *u* sebagai *root* pada sebagian *tree* tersebut.
5. *Left subtree* dari *u* ialah *subtree* yang menjadikan *left child* dari *node u* sebagai *root* dari *subtree* tersebut.
6. *Right subtree* dari *u* ialah *subtree* yang menjadikan *right child* dari *node u* sebagai *root* dari *subtree* tersebut.

7. *Size* dari *tree* atau *subtree* ialah banyaknya jumlah *node* yang dimiliki oleh *tree* atau *subtree* tersebut.

2.2 Binary Search Tree (BST)

Binary search tree ialah *binary tree* yang terorganisasi. Sebuah *tree* dapat direpresentasikan dengan data struktur yang terhubung di mana setiap *node* ialah sebuah objek. Tidak hanya *key* yang disimpan oleh setiap *node,* setiap *node* menyimpan atribut *left, right,* dan *parent* di mana menunjuk *node* lain yang sesuai dengan *left child, right child,* dan *parent* berturut-turut dari *node* tersebut.



Gambar 2.1 Ilustrasi Binary Search Tree

Key pada *binary search tree* selalu disimpan dengan cara yang memenuhi *binary-search-tree property*:

Misal x ialah sebuah *node* pada *binary search tree*. Jika y terletak pada *left subtree* dari x maka $y.key \leq x.key$. Jika y terletak pada *right subtree* dari x maka $y.key \geq x.key$ [1].

Pada Gambar 2.1 *key* dari *root* pada ialah 6 dan *node-node* yang memiliki *key* 2, 5 dan 5 terdapat pada *left subtree* dari root tersebut di mana tidak ada yang lebih besar dari 6 dan *node* yang mempunyai *key* 7 dan 8 terdapat pada *right subtree* dari root di mana tidak ada yang lebih kecil dari 6. Semua *node* pada *binary search tree* menerapkan properti yang sama. Contohnya *node* yang memiliki *key* 5 pada *left child* dari *root* memiliki *left child* yang

mempunyai *key* tidak lebih besar dari 5 dan memiliki *right child* yang *key* nya tidak kurang dari 5.

Dengan *binary-search-tree property,* struktur data ini dapat digunakan dalam hal memudahkan operasi pencarian *key* pada elemen yang ada pada *tree* tersebut dan dapat menampilkan semua *key* yang terdapat pada *binary search tree* dalam keadaan terurut dengan algoritma rekursif sederhana yang disebut *inorder traversal.*

2.3 Self Balancing Binary Search Tree

Self balancing binary search tree adalah *binary search tree* yang dapat menyesuaikan atau menyeimbangkan *tree* itu sendiri, yaitu meminimalkan kedalaman atau *depth* pada *tree* pada saat *tree* tersebut dilakukan operasi penambahan *node* atau penghapusan *node.* Pada umumnya *self balancing binary search tree* mempunyai tinggi maksimal sebesar $log(n)$ dengan *n* adalah jumlah *node*. Hal ini menyebabkan sebagian algoritma *self balancing binary search tree* mempunyai kompleksitas $log(n)$ untuk tiap operasi nya.

Terdapat banyak macam data struktur yang mengimplementasikan konsep *self balancing binary search tree*. Semua macam tersebut bertujuan sama yaitu meminimalkan kedalaman pada suatu *tree* agar mempercepat kinerja operasi-operasi yang dilakukan pada *tree* tersebut. Contoh data struktur yang populer yang mengimplementasikan *self balancing binary search tree* antara lain:

1. Red-black tree

   Red-black tree termasuk *self balancing binary search tree* yang diciptakan oleh Rudolf Bayer di tahun 1972 dan bernama *Symmetric binary B-tree,* lalu pada tahun 1978 diperbarui menjadi Red-black tree oleh Leonidas J. Guibas Robert

Sedgewick. Pada struktur data ini memiliki aturan-aturan seperti menandakan sebuah *node* dengan merah atau hitam.

2. AA tree
   AA Tree adalah salah satu jenis *self balancing binary search tree* yang ditemukan oleh Arne Andersson. *Tree* ini merupakan varian dari Red-black Tree. Dalam *tree* ini terdapat beberapa syarat seperti Red-black Tree.
3. AVL tree
   AVL Tree adalah salah satu jenis *self balancing binary search tree* yang ditemukan oleh G.M Adelson-Velskii dan E.M Landis pada tahun 1961. Pada *tree* ini terdapat *balance factor* yaitu nilai ketinggian anak pohon di kiri dikurang ketinggian anak pohon kanan. Pohon dianggap seimbang bila *balance factor* bernilai -1, 0 atau 1.
4. Scapegoat tree
   Scapegoat tree adalah salah satu jenis *self balancing binary search tree* yang diciptakan oleh Igal Galperin, Jacob Tepec dan Ronald L. Rivest. Pada *tree* ini dikatakan seimbang jika jumlah *node* di *subtree* kanan sama dengan jumlah *node* di *subtree* kiri.
5. Splay tree
   Splay tree ialah *self balancing binary search tree* yang ditemukan oleh Daniel Dominic Sleator dan Robert Endre Tarjan. *Tree* ini memegang properti di mana elemen yang baru diakses akan lebih cepat waktu pengaksesannya untuk pengaksesan selanjutnya.

Pada penelitian ini akan digunakan Splay Tree untuk penyelesaian permasalahan. Pemilihan struktur data Splay Tree untuk diterapkan pada penelitian ini dikarenakan implementasi pada struktur data tersebut pada persoalan yang berkaitan pada penelitian ini tidak diperlukan *binary search tree* yang selalu

balance melainkan hanya dibutuhkan efisiensi waktu pengaksesan suatu *node.*

2.4 Teknik Key Secara Implisit

Atribut *key* pada setiap *node* dalam *binary search tree* adalah satu hal yang sangat penting dan harus diperhatikan. Di mana *key* pada *node* yang terdapat pada *binary search tree* sangat menentukan di mana posisi *node* tersebut berada. *Key* sangat bergantung pada permasalahan yang dihadapi dan bagaimana cara hal tersebut disimpan. Umumnya *key* pada node disimpan secara eksplisit dan *key* tersebut dapat langsung diakses dengan mudah.

Untuk permasalahan yang diberikan sebuah deret atau barisan bilangan dan diharuskan tetap mengatur bilangan tersebut sesuai urutan deretan, deret bilangan tersebut dapat direpresentasikan dalam bentuk *binary search tree* dan menggunakan indeks dari tiap bilangan sebagai *key* untuk tiap *node* dan menyimpannya nilai indeks itu secara eksplisit, namun jika pada permasalahan tersebut terdapat operasi-operasi yang membuat deret tersebut menjadi dinamis seperti penyisipan atau penghapusan bilangan, penerapan pengaksesan *key* secara eksplisit membuat pengaturan untuk mempertahankan *node-node* pada *tree* tetap menyimpan indeks yang seharusnya menjadi sulit dan sangat memakan waktu karena dibutuhkan pembaruan *key* yang dimiliki oleh *node-node* yang terkena dampak dari operasi penyisipan atau penghapusan.

Terdapat teknik yang dapat digunakan untuk kasus permasalahan seperti itu dan teknik ini dapat memudahkan pengaturan untuk mempertahankan informasi yang dimiliki *node* akan letak dari bilangan yang dimiliki *node* tersebut sesuai urutan dari barisan bilangan. Teknik ini bisa disebut teknik *implicit key*. Teknik ini tetap menggunakan urutan indeks sebagai *key* nya namun diterapkan secara implisit, di mana informasi tambahan

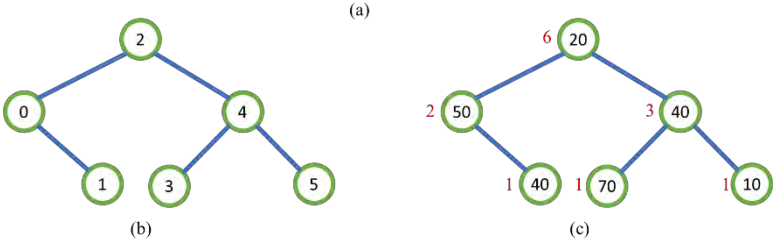
yang disimpan oleh setiap *node* ialah *size* dari *subtree* pada *node* tersebut [2].

Teknik ini tetap mempertahankan *binary-search-tree property* di mana untuk kasus ini *left subtree* dari sebuah *node u* yang mewakilkan bilangan pada indeks *x* memiliki indeks kurang dari *x* dan *right subtree* dari *node u* tersebut memiliki indeks lebih dari *x*.

Pada Gambar 2.2 (a) diberikan contoh deret bilangan yang mempunyai informasi nilai dan indeks dan contoh representasi deret tersebut dalam *binary search tree* dapat dilihat pada Gambar 2.2 (b) dengan nilai indeks pada tiap *node* nya di mana tiap *node* memiliki *left subtree* yang mempunyai indeks lebih kecil dan *right subtree* yang mempunyai nilai indeks lebih besar.

Indeks	0	1	2	3	4	5
Value	50	40	20	70	40	10

(a)



(b) (c)

Gambar 2.2 Penggunaan Implicit Key. (a) Contoh Deret Bilangan. (b) Representasi Deret pada Binary Search Tree Sesuai Indeks. (c) Representasi Deret dengan Implicit Key pada Binary Search Tree.

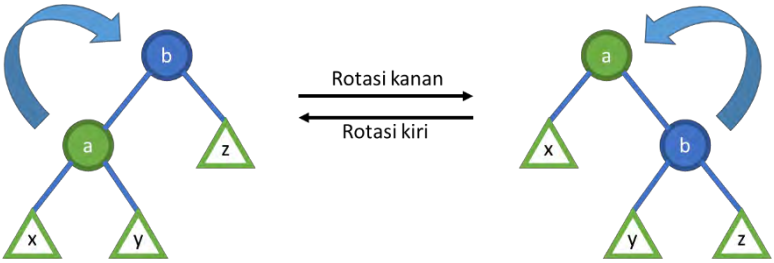
Gambar 2.2 (c) menunjukkan representasi deret tersebut dalam *binary search tree* namun tidak menyimpan informasi indeks tersebut secara eksplisit, informasi yang diperlukan hanya *size* dari setiap *subtree* pada *node*. Informasi akan indeks pada sebuah *node* bergantung pada *size left subtree* pada *node* tersebut. Dapat dilihat pada *root* dari *tree* tersebut dengan *value* = 20

memiliki *size left subtree* = 2 maka *root* tersebut memiliki indeks = *size left subtree* yaitu bernilai 2. Untuk mengetahui *right child* dari *root* dilakukan hal yang sama namun ditambahkan dengan nilai *size left subtree root* + 1, nilai 1 didapat dari *node root* itu sendiri, untuk menganggap *node* yang sudah terlewati. Ini dilakukan setiap melakukan perjalanan ke *right child* pada tiap *node*. Sebaliknya, untuk perjalanan ke *left child* tidak dilakukan demikian.

2.5 Splay Tree

Splay Tree merupakan salah satu struktur data *self balancing binary search tree*, karena itu Splay Tree mendukung semua operasi yang ada pada *binary search tree*. Pada dasarnya semua operasi pada Splay Tree dilakukan dengan operasi *splaying*. *Splaying* digunakan untuk memindahkan *node* yang baru diakses menuju *root*. Hal ini bertujuan untuk membuat waktu akses pada *node* yang
sering diakses menjadi lebih efisien, karena operasi *splaying* membuat *node-node* yang sering diakses terletak dekat dengan *root.* Untuk memenuhi kebutuhan *splaying* di mana membuat suatu *node* berpindah menuju *root* maka dilakukan rotasi terhadap *node* di mana rotasi tersebut tetap mempertahankan urutan dari sebuah *tree*.

Pada Gambar 2.3 diberikan ilustrasi rotasi kanan dan rotasi kiri dan dapat dilihat rotasi kanan dapat dilakukan pada sebuah *node* jika *node* tersebut merupakan *left child* dari *parent*-nya dan rotasi kiri dilakukan jika *node* tersebut merupakan *right child* dari *parent*-nya. *Pseudocode* fungsi rotasi ditunjukkan pada Gambar 2.4.



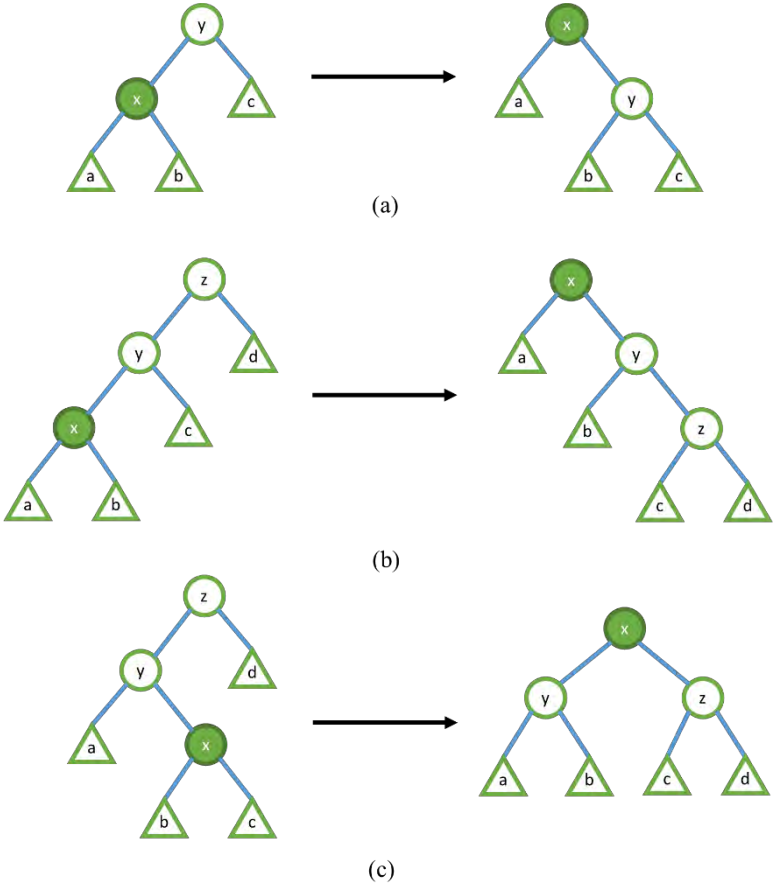
Gambar 2.3 Ilustrasi Rotasi Kanan dan Rotasi Kiri pada Node.

Gambar 2.4 menunjukkan *pseudocode* dari fungsi rotate, di mana fungsi tersebut sudah menangani fungsi rotasi kanan dan rotasi kiri.

```
/**
 * x is the element to be rotated
 */
Rotate(x)
01. y = parent of x
02. z = parent of y
03. parent of x = z
04. parent of y = x
05. if x is left child of y
06.     b = right child of x
07. else
08.     b = left child of x
09. if b is not null
10.   parent of b = y
11. if z is not null
12.   if left child of z = y
13.         left child of z = x
14.   else
15.         right child of z = x
16. if x = left child of y
17.   right child of x = y
18.   left child of y = b
19. else
20.   left child x = y
21.   right child of y = b
```

Gambar 2.4 Pseudocode Fungsi Rotate

2.5.1 Splaying Step



Gambar 2.5 Splaying Step. (a) Zig. (b) Zig-zig. (c) Zig-zag

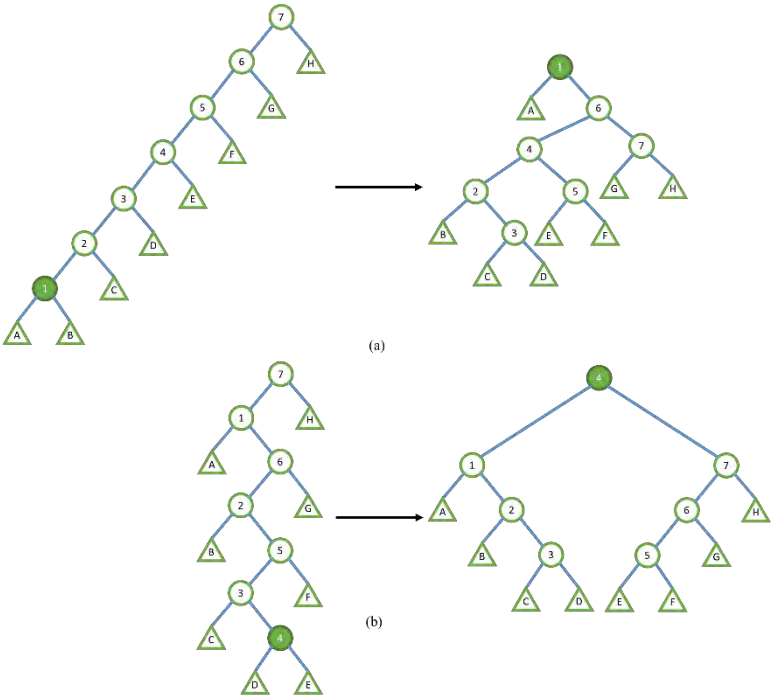
Untuk melakukan *splay* suatu *tree* pada *node x*, *splaying step* terus dilakukan hingga *x* menjadi *root* pada *tree* tersebut, untuk ilustrasi dapat dilihat pada Gambar 2.5. Terdapat tiga kasus pada *splaying step* yaitu:

1. **Zig**, jika *p(x)* atau *parent x* adalah *root* lakukan rotasi pada *x* sehingga *x* menjadi *root.* (Kasus ini merupakan proses akhir.)
2. **Zig-zig**, jika *p(x)* bukan *root* dan baik *x* maupun *p(x)* keduanya merupakan *left child* atau keduanya merupakan *right child*, lakukan rotasi pada *p(x)* terlebih dahulu lalu lakukan rotasi pada *x*.
3. **Zig-zag**, jika *p(x)* bukan *root* dan *x* merupakan *left child* dan *p(x)* merupakan *right child* atau sebaliknya, maka lakukan rotasi pada *x* lalu lakukan rotasi lagi pada *x* [2].

Splaying pada *node* x dengan kedalaman d memakan waktu $\theta(d)$, waktu tersebut merupakan waktu untuk mengakses item pada *node* x. *Splaying* tidak hanya memindahkan *node* x menuju *root* tetapi juga memotong kedalaman untuk setiap node yang terdapat pada path saat diakses.

Gambar 2.6 mengilustrasikan suatu *tree* yang melakukan operasi *splay*. Pada *tree* tersebut dapat dilihat setelah dilakukan operasi *splay*, *tree* tersebut tetap mempertahankan *binary search tree property* dan kedalaman dari *node-node* yang terpengaruh operasi *splay* mayoritas memiliki kedalaman yang lebih pendek dari sebelumnya, hal ini sangat berpengaruh untuk membuat efisien dari pengaksesan *node* atau operasi lainnya.

Pada Gambar 2.7 menunjukkan *pseudocode* dari fungsi *splay* yang di dalamnya sudah termasuk kasus zig-zig, zig-zag dan zig di mana pada baris tiga hingga baris tujuh ialah kasus untuk zig-zig dan zig-zag dan kasus tersebut ditentukan pada kondisi yang terdapat pada baris empat, untuk kasus zig ditentukan pada kondisi pada baris tiga, jika tidak memenuhi baris ke tiga maka hanya dilakukan satu rotasi dan hal tersebut ialah kasus zig. Pada fungsi tersebut memanggil juga fungsi dari rotasi yang sebelumnya sudah dibahas.



Gambar 2.6 Ilustrasi Splay pada Tree. (a) Splay pada Node 1. (b) Splay pada Node 4

```
/**
 * x is the element to be splayed
 */
Splay(x)
```

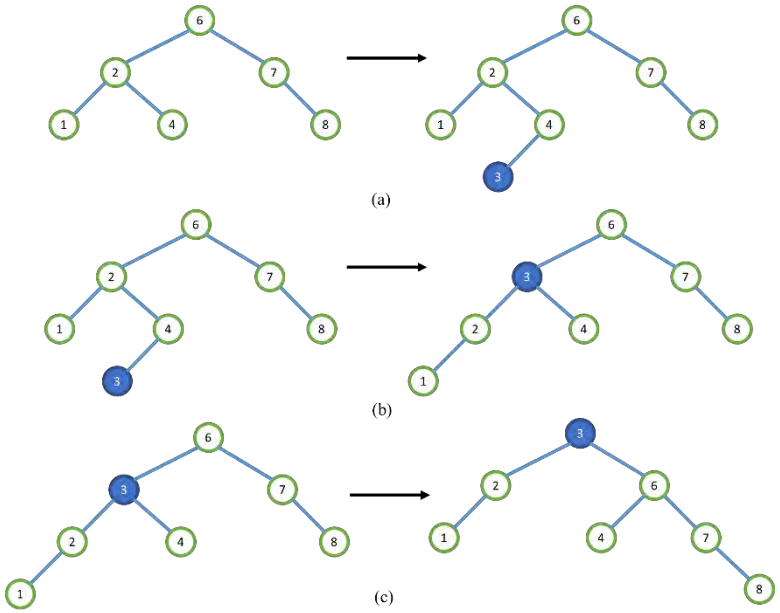
```
1. while parent of x is not null
2.      y = parent of x
3.      if parent of y is not null
4.          if left child y = x and left child of parent of y=y
5.              rotate y
6.          else
7.              rotate x
8.      rotate x
```

Gambar 2.7 Pseudocode dari Fungi Splay

2.5.2 Operasi Dasar Splay Tree

Operasi-operasi dasar pada Splay Tree umumya sama dengan operasi *binary search tree* pada umumnya, yaitu antara lain:

2.5.2.1 Operasi Penyisipan



Gambar 2.8 Ilustrasi Penyisipan Node

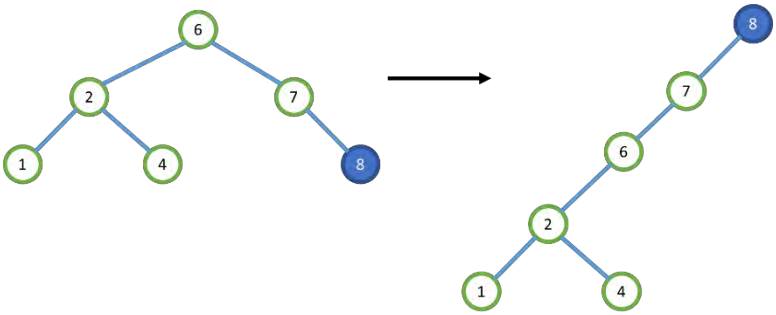
Operasi penyisipan pada Splay Tree dapat dilakukan dengan berbagai cara yang intinya membuat *node* yang baru disisipkan menjadi *root* pada *tree*. Salah satu cara yang sederhana ialah dengan penyisipan seperti penyisipan pada *binary search tree* biasa dan dilakukan operasi splay pada *node* tersebut.

Ilustrasi dari penyisipan dapat dilihat pada Gambar 2.8. Pada ilustrasi ini akan dilakukan penyisipan *node* bernilai 3 pada contoh

binary tree di mana sesuai dengan penyisipan *binary search tree*, *node* tersebut disisipkan pada anak kiri dari *node* yang bernilai 4.

Setelah dilakukan penyisipan dilakukan operasi splay pada *node* tersebut, dengan kasus yang dimiliki pada *node* yang baru disisipkan maka langkah pertama dilakukan zig-zag, ilustrasi dapat dilihat pada Gambar 2.8 (b), setelah zig-zag dilakukan, langkah selanjutnya dilakukan zig, ilustrasi dapat dilihat pada Gambar 2.8 (c), sehingga *node* bernilai 3 yang di mana baru disisipkan berada di posisi *root* pada *tree*.

2.5.2.2 Operasi Pencarian



Gambar 2.9 Ilustrasi Pencarian Node

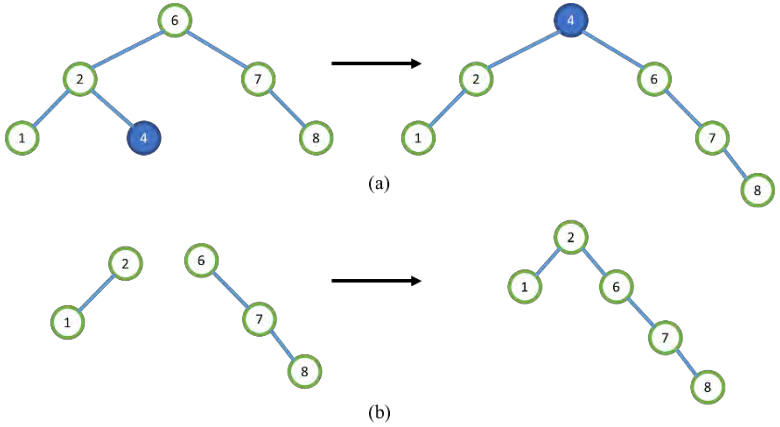
Operasi pencarian pada Splay Tree dilakukan sama seperti pencarian *binary search tree.* Setelah didapatkan *node* yang dicari maka dilakukan operasi splay pada *node* tersebut.

Ilustrasi dari operasi pencarian dapat dilihat pada Gambar 2.9. Pada ilustrasi tersebut *node* yang dicari ialah yang bernilai 8, setelah ditemukan *node* tersebut dilakukan operasi splay pada *node* tersebut, di mana pada kasus yang dimiliki *node* bernilai 8 harus dilakukan langkah zig-zig. Pada akhirnya *node* yang ditemukan menjadi *root* pada *tree.*

2.5.2.3 Operasi Penghapusan

Operasi penghapusan pada Splay Tree dilakukan dengan operasi pencarian seperti yang dijelaskan pada subbab sebelumnya sehingga membuat *node* yang akan dihapus menjadi *root* pada *tree* dan dilakukan penghapusan pada *node* tersebut seperti penghapusan pada *binary search tree*.

Ilustrasi dari operasi penghapusan dapat dilihat pada Gambar 2.10. Pada ilustrasi ini dilakukan penghapusan pada *node* bernilai 4. Langkah awal yang dilakukan ialah pencarian pada *node* tersebut dan dilakukan operasi splay pada *node* tersebut sehingga membuat *node* tersebut menjadi *root* pada *tree* dan setelah itu *node* tersebut dihapus. Setelah *node* tersebut dihapus dilakukan pencarian *node* pengganti untuk menempati *node* yang telah dihapus, pada kasus ini diambil *node* yang memiliki nilai maksimum pada anak kiri dari *node* yang telah dihapus.



Gambar 2.10 Ilustrasi Operasi Penghapusan

2.6 Sistem Penilaian Daring

Sistem penilaian daring atau *online judge* ialah sebuah sistem yang bekerja untuk melakukan pengujian suatu program. Pada

sistem penilaian daring terdapat berbagai soal yang masing-masing terdiri atas deskripsi soal, berkas data masukan dan berkas data keluaran. Deskripsi soal berisi deskripsi tugas yang harus dilakukan oleh program milik pengguna, batasan waktu eksekusi program, batasan memori program, serta batasan dan format masukan yang diterima dan format keluaran yang dihasilkan. Berkas data masukan berisi data masukan yang akan diberikan pada program, sesuai dengan batasan pada deskripsi soal. Berkas data keluaran berisi data keluaran yang akan dicek kesamaan dengan hasil keluaran program pengguna.

Sistem penilaian daring menerima program pengguna berupa berkas kode sumber yang dikumpulkan oleh pengguna melalui antarmuka sistem, kemudian melakukan kompilasi dan menjalankan program yang dihasilkan. Program hasil kompilasi dari kode pengguna dijalankan dan diuji dengan berkas data masukan soal. Kemudian sistem melakukan pengecekan hasil keluaran program dengan berkas data keluaran soal. Berkas data masukan dan data keluaran soal merupakan hasil studi pembuat soal dan telah diuji kebenarannya oleh tim ahli, sehingga kesesuaian antara deskripsi soal dengan berkas data masukan dan data keluaran dapat dipertanggungjawabkan.

Pada tahap pengujian program, pengguna tidak dapat mengetahui isi berkas data masukan dan berkas data keluaran. Pengguna hanya mendapatkan umpan balik dari sistem berupa total waktu eksekusi program dan frase yang menunjukkan kebenaran program pengguna. Program pengguna dianggap benar jika dan hanya jika hasil keluaran program pengguna sama persis dengan data keluaran soal dan memenuhi kriteria batas waktu eksekusi program. Jika program pengguna benar maka sistem akan memberikan umpan balik berupa frase *"Accepted"*. Sedangkan

jika program pengguna tidak benar, maka sistem akan memberi umpan balik berupa salah satu dari frase-frase berikut:

1. "Wrong Answer", yaitu jika hasil keluaran program pengguna tidak sama persis dengan berkas data keluaran soal.
2. "Time Limit Exceeded", yaitu jika waktu eksekusi program pengguna melebihi batasan waktu yang diberikan pada deskripsi soal.
3. "Memory Limit Exceeded", yaitu jika memori yang dibutuhkan program pengguna melebihi batasan waktu yang diberikan pada deskripsi soal.
4. "Runtime Error", yaitu jika terjadi galat ketika program pengguna sedang berjalan dan menyebabkan program berhenti sebelum selesai di eksekusi.
5. "Compile Error", yaitu ketika sistem tidak dapat melakukan kompilasi pada berkas kode sumber dari pengguna.

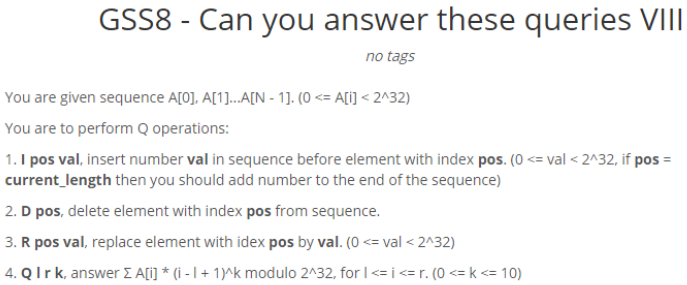
Pada awal pengembangannya, sistem penilaian daring hanya digunakan pada kontes pemrograman untuk menguji kebenaran jawaban peserta yang dikumpulkan. Namun seiring perkembangan kuantitas kontes pemrograman di dunia, sistem penilaian daring dapat digunakan di luar kontes pemrograman melalui situs penilaian daring. Pada situs penilaian daring terdapat server yang berisi sistem penilaian daring. Pengguna dapat mengumpulkan berkas kode sumber melalui web untuk diuji di server, kemudian mendapat umpan balik dari server pada situs yang sama.

2.7 Permasalahan Can you answer these queries VIII Pada SPOJ

Salah satu permasalahan yang terdapat pada SPOJ adalah *Can you answer this queries VIII* yang mempunyai nomor 19543 dan kode GSS8. Deskripsi permasalahan ditunjukkan pada Gambar 2.11.

Deskripsi singkat persoalan tersebut ialah diberikan barisan bilangan sebanyak *N* bilangan dan operasi pada bilangan tersebut sebanyak *Q* operasi. Operasi-operasi tersebut ialah:

- Memasukan bilangan pada posisi yang ditentukan
- Menghapus bilangan pada posisi yang ditentukan
- Mengubah nilai bilangan pada posisi yang ditentukan
- Menampilkan hasil *query* pada jarak yang ditentukan



GSS8 - Can you answer these queries VIII

no tags

You are given sequence A[0], A[1]...A[N - 1]. (0 <= A[i] < 2^32)

You are to perform Q operations:

1. **I pos val**, insert number **val** in sequence before element with index **pos**. (0 <= val < 2^32, if **pos** = **current_length** then you should add number to the end of the sequence)

2. **D pos**, delete element with index **pos** from sequence.

3. **R pos val**, replace element with idex **pos** by **val**. (0 <= val < 2^32)

4. **Q l r k**, answer Σ A[i] * (i - l + 1)^k modulo 2^32, for l <= i <= r. (0 <= k <= 10)

Gambar 2.11 Deskripsi permasalahan Can you answer these queries VIII

Berikut merupakan format masukan yang diminta dari permasalahan:

1. Baris pertama terdiri dari satu bilangan integer *N* yang merepresentasikan jumlah bilangan pada barisan.
2. Baris kedua terdiri dari *N* bilangan integer yang merepresentasikan barisan awal $A_{[0]} \dots A_{[N-1]}$.
3. Baris ketiga terdiri dari satu buah integer *Q* yang merepresentasikan banyaknya operasi yang akan dilakukan.
4. *Q* baris selanjutnya merupakan perintah operasi sesuai dengan format yang diberikan. Adapun format tersebut ialah:
   - **'I' pos val**, yang artinya memasukan bilangan bernilai **val** pada posisi **pos**, jika **pos** merupakan panjang

barisan saat ini maka bilangan tersebut diletakkan pada akhir barisan

- **'D' pos**, yang artinya menghapus bilangan pada posisi **pos**
- **'R' pos val**, yang artinya mengubah bilangan pada posisi **pos** dengan nilai **val**
- **'Q' l r k**, yang artinya hitung hasil *query* dari $\sum_{i=l}^{r} A_{[i]} \times (i - l + 1)^k$ modulo 2^{32}

Format keluaran dari permasalahan tersebut ialah untuk setiap operasi **'Q'**, cetak satu integer per-baris sesuai hasil yang diminta.

Batasan-batasan masalah pada persoalan dapat dilihat pada Gambar 2.11 dan Gambar 2.12.

The first line of the input contains an integer N (1 <= N <= 100000).

The following line contains N integers, representing the starting sequence A[0]..A[N-1].

The third line contains an integer Q (0 <= Q <= 100000).

Time limit:	1.5s
Source limit:	50000B
Memory limit:	1536MB

Gambar 2.12 Batasan-batasan pada Persoalan

Berikut merupakan batasan dari permasalahan tersebut:

1. $1 \leq N \leq 100.000$
2. $0 \leq A_{[i]} \leq 2^{32}$
3. $0 \leq val \leq 2^{32}$
4. $0 \leq pos \leq panjang\ barisan$
5. $0 \leq l, r \leq panjang\ barisan$
6. $0 \leq K \leq 10$

Contoh masukan dan keluaran pada permasalahan tersebut ditunjukkan pada Gambar 2.13. Untuk mengilustrasikan penggunaan Splay Tree pada permasalahan ini maka akan diberikan masukan yang hanya menggunakan operasi-operasi

dasar saja. Operasi-operasi tersebut ialah pemasukan bilangan, penghapusan bilangan dan perubahan bilangan pada barisan bilangan. Berikut ialah contoh masukan dan bentuk tree awal.

Example

```
Input:
4
1 2 3 5
7
Q 0 2 0
I 3 4
Q 2 4 1
D 0
Q 0 3 1
R 1 2
Q 0 1 0
```

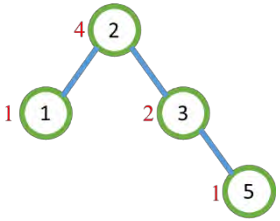
```
Output:
6
26
40
4
```

Gambar 2.13 Contoh Masukan dan Keluaran pada Permasalahan.

2.7.1 Penyelesaian Operasi Dasar

```
Input:
4
1 2 3 5
3
I 3 4
D 0
R 1 2
```

(a)



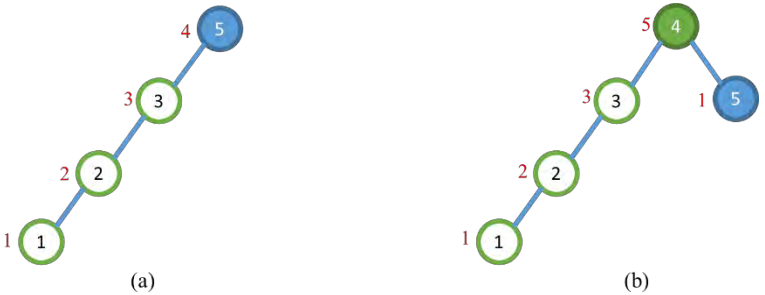
(b)

Gambar 2.14 (a) Contoh Masukan dengan Operasi Dasar. (b) Ilustrasi Tree Awal Sesuai dengan Kasus Uji pada Masukkan

Pada Gambar 2.14 (a) diberikan masukan dengan 4 integer awal pada barisan yang terdiri dari 1, 2, 3 dan 5 dan terdapat 3 operasi yaitu memasukan bilangan pada posisi 3 dengan nilai 4, hapus bilangan pada posisi 0, ubah bilangan pada posisi 1 dengan

nilai 2. Pada Gambar 2.14 (b) dapat dilihat ilustrasi *tree* awal dengan menggunakan *implicit key* di mana menyimpan nilai dan *size* dari tiap *subtree*.

Untuk melakukan penyisipan bilangan pada barisan pada suatu posisi maka dilakukan operasi *splay* pada posisi tersebut. Maka dilakukan *splay* pada posisi 3 dan Gambar 2.15 (a) menunjukkan hasil *splay* pada *tree*. Setelah dilakukan operasi *splay* maka dilakukan penambahan *node* pada posisi tersebut dan Gambar 2.15 (b) menunjukkan hasil dari penambahan *node* baru pada *tree*.

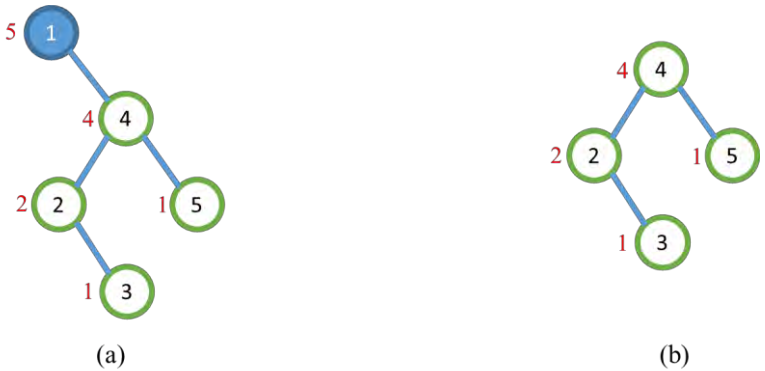


Gambar 2.15 Ilustrasi Operasi Penyisipan. (a) Tree setelah Dilakukan Operasi Splay pada Posisi 3. (b) Bentuk Tree setelah Dimasukan Nilai 4 pada Posisi .

Untuk melakukan penghapusan bilangan pada barisan maka dilakukan operasi *splay* pada *node* yang akan dihapus. Pada Gambar 2.16 (a) dapat dilihat bentuk *tree* setelah dilakukan operasi *splay* pada *node* yang merepresentasikan posisi 0. Setelah dilakukan operasi *splay* maka dilakukan operasi penghapusan *node* yang akan dihapus. Gambar 2.16 (b) menunjukkan hasil *tree* setelah dilakukan penghapusan *node*.

Untuk melakukan perubahan nilai pada barisan di suatu posisi maka dilakukan pencarian *node* yang merepresentasikan posisi tersebut dan dilakukan perubahan nilai. Gambar 2.17 (a) menunjukkan hasil dari perubahan nilai pada suatu *node*. Setelah

dilakukan perubahan nilai tersebut maka pada *node* tersebut dilakukan operasi *splay* di mana hasil dari operasi *splay* pada *tree* tersebut dapat dilihat pada Gambar 2.17 (b).



Gambar 2.16 Ilustrasi operasi penghapusan. (a) Kondisi tree setelah dilakukan operasi splay. (b) Bentuk tree setelah dilakukan operasi penghapusan

Gambar 2.17 Ilustrasi Operasi Perubahan Nilai pada Barisan. (a) Bentuk Tree setelah Dilakukan Perubahan Nilai Pada Node. (b) Kondisi Tree Setelah Operasi Splay

2.7.2 Penyelesaian Operasi Perhitungan pada suatu Jarak

Untuk operasi perhitungan pada suatu jarak dengan batas kiri l dan batas kanan r dengan nilai k yang ditentukan dapat disimbolkan dengan Q dengan Persamaan (*2.1*)

$$Q = \sum_{i=l}^{r} A_i \times (i - l + 1)^k \ mod\ 2^{32} \quad (2.1)$$

Diperlukan penyimpanan pada setiap *node* yang mewakili suatu segmen dari batas kiri l dan batas kanan r untuk setiap k bernilai 0 hingga 10 di mana S_k ialah jumlah dari suatu perhitungan dengan Persamaan (*2.2)*

$$S_k = \sum_{i=0}^{r-l} A_{l+i}\ i^k \quad (2.2)$$

Untuk menjawab perhitungan Q dibutuhkan perhitungan untuk *sub query* sebagai berikut atau disimbolkan dengan q dari suatu jarak l hingga r dan memiliki delta atau selisih dari batas kiri pada *query* dengan batas kiri yang dimiliki suatu *node* dan disimbolkan sebagai d dapat dirumuskan dengan Persamaan (*2.3)*.

$$q = \sum_{i=0}^{r-l} A_{l+i}\ (d + i)^k \quad (2.3)$$

Di mana $(d + i)^k$ dapat dihitung dengan konsep binomial koefisien menjadi persamaan pada Persamaan*(2.4)*.

$$(d + i)^k = \sum_{j=0}^{k} \binom{k}{j} d^{k-j}\ i^j \quad (2.4)$$

Dari persamaan diatas q dapat disusun menjadi persamaan pada Persamaan (*2.5)*

$$q = \sum_{i=0}^{k} \binom{k}{i} d^{k-i}\ S_i \quad (2.5)$$

Maka untuk setiap *node* yang merepresentasikan suatu segmen dengan batas kiri l dan batas kanan r akan melakukan perhitungan sesuai persamaan tersebut.

2.8 Pembuatan Data Generator Untuk Uji Coba

Pembuatan data generator dilakukan untuk membuat kasus uji yang akan digunakan untuk pengujian pada uji coba yang akan dijelaskan pada bab V. Data generator ini disesuaikan dengan format masukan yang sudah dijelaskan pada subbab 2.7.

Terdapat 3 skenario yang akan dilakukan untuk uji coba, skenario tersebut ialah pengaruh waktu terhadap jumlah *node*,

pengaruh waktu terhadap jumlah operasi dan pengaruh waktu terhadap jumlah *node* dan operasi.

Untuk skenario pertama akan dibuat data generator dengan banyak bilangan awal 100 bilangan dan terdapat banyak operasi penyisipan pada suatu posisi dari 0 hingga 900 dengan rentang 100, sehingga hasil akhir jumlah bilangan pada barisan ialah 100 hingga 1000 dengan rentang 100. Pada skenario ini operasi hanya melibatkan operasi penyisipan. Untuk *pseudocode* dari data generator untuk skenario pertama dapat dilihat pada Gambar 2.18.

Pada *pseudocode* tersebut digunakan bilangan acak untuk membuat nilai-nilai yang dibutuhkan kecuali nilai dari banyaknya bilangan awal dan banyaknya operasi. Dapat dilihat pada baris ke-10 di *pseudocode* dilakukan pengeluaran ke dalam file dengan format operasi penyisipan saja. Hasil dari program ini ialah file berjumlah 10 file dengan nama "0.txt" hingga "900.txt" sesuai banyaknya operasi.

```
Scenario1()
01. N = 100
02. for Q = 0 to 900 increment by 100
03.     filename = Q + ".txt"
04.     myfile = open file filename
05.     printonmyfile N
06.     for i = 0 to N
07.         printonmyfile random modulo 4294967297
08.     printonfile Q
09.     for i = 0 to Q
10.         printonmyfile "I " + random modulo N + " " + random
modulo 4294967297
11.     close myfile
```

Gambar 2.18 Pseudocode Data Generator Skenario Pertama

Untuk skenario kedua akan dibuat data generator dengan banyak bilangan awal yang tetap seperti sebelumnya yaitu 100 bilangan dan terdapat banyak operasi perubahan nilai pada suatu posisi dari 1.000 hingga 10.000 dengan rentang 1.000, sehingga

hasil akhir jumlah bilangan pada barisan ialah tetap 100. Pada skenario ini operasi hanya melibatkan operasi perubahan bilangan. Untuk *pseudocode* dari data generator untuk skenario pertama dapat dilihat pada Gambar 2.19.

```
Scenario2()
01. N = 100
02. for Q = 1000 to 10000 increment by 1000
03.     filename = Q + ".txt"
04.     myfile = open file filename
05.     printonfile N
06.     for i = 0 to N
07.         printonmyfile random modulo 4294967297
08.     printonfile Q
09.     for i = 0 to Q
10.         printonmyfile "R " + random modulo N + " " + random
modulo 4294967297
11.     close myfile
```

Gambar 2.19 Pseudocode Data Generator Skenario Kedua

Untuk skenario kedua mirip dengan skenario pertama namun perbedaan pada skenario ini ialah banyaknya operasi yang dimulai dari 1.000 hingga 10.000 dengan rentang 1.000 dan dapat dilihat pada baris 10 dilakukan pencetakan dengan format untuk operasi perubahan nilai bilangan pada suatu posisi. Hasil dari program ini ialah 10 file dengan nama “1000.txt” hingga “10000.txt”

Untuk skenario ketiga akan dibuat data generator dengan banyak bilangan awal yang tetap seperti sebelumnya yaitu 100 bilangan dan terdapat banyak operasi penyisipan bilangan dan perubahan nilai pada suatu posisi dari 0 hingga 1800 dengan rentang 200 di mana rasio banyaknya operasi penyisipan dan operasi perubahan ialah 1:1. Untuk *pseudocode* dari data generator untuk skenario pertama dapat dilihat pada Gambar 2.20.

Perbedaan yang terdapat pada skenario ini ialah banyaknya operasi yang dua kali lipat dari skenario sebelumnya yang dapat dilihat pada baris 8, di mana Q dikalikan dengan 2 dan terdapat 2

format operasi yang dicetak pada baris 10 dan baris 11 di mana format sesuai dengan operasi penyisipan dan operasi perubahan nilai.

```
Scenario3()
01. N = 100
02. for Q = 0 to 900 increment by 1000
03.     filename = Q + ".txt"
04.     myfile = open file filename
05.     printonfile N
06.     for i = 0 to N
07.         printonmyfile random modulo 4294967297
08.     printonfile 2*Q
09.     for i = 0 to Q
10.         printonmyfile "I " + random modulo N + " " + random
modulo 4294967297
11.         printonmyfile "R " + random modulo N + " " + random
modulo 4294967297
12.     close myfile
```

Gambar 2.20 Pseudocode Data Generator Skenario Ketiga

BAB III
DESAIN

Pada bab ini penulis menjelaskan desain algoritma dan struktur data yang digunakan dalam penelitian.

3.1 Definisi Umum Sistem

Sistem akan melakukan fungsi *preprocess* untuk kebutuhan kalkulasi pada saat operasi *query*. Setelah itu sistem akan menerima input banyaknya bilangan pada barisan awal atau banyaknya node awal dan menerima input bilangan-bilangan pada barisan tersebut, setiap bilangan yang dimasukan akan dilakukan fungi insert pada Splay Tree. Setelah itu sistem akan menerima input banyaknya jumlah operasi dan diikutkan deskripsi dari operasi tersebut. Terdapat empat jenis operasi, yaitu:

1. Penyisipan bilangan pada barisan dengan diberikan deskripsi bilangan tersebut dan posisi di mana bilangan tersebut akan disisipkan.
2. Penukaran bilangan pada barisan dengan diberikan deskripsi nilai bilangan dan posisi bilangan yang akan ditukar.
3. Penghapusan bilangan pada barisan dengan diberikan deskripsi posisi bilangan yang akan dihapus.
4. *Query* barisan dengan diberikan deskripsi batas kiri, batas kanan barisan dan nilai konstanta yang diperlukan untuk kalkulasi pada *query*.

Barisan tersebut akan direpresentasikan menjadi sebuah Splay Tree dan setiap terjadi modifikasi pada barisan akan dilakukan fungsi *insert*, *delete* atau *replace*. *Pseudocode* fungsi Main ditunjukkan pada Gambar 3.1. Pada fungsi tersebut dilakukan penerimaan masukan sesuai format masukan pada persoalan yang berkaitan.

```
Main()
01. root = null
02. preprocess()
03. input number of sequence
04. input each value of sequence
05. input number of operation
06. while number of operation > 0
07.     input command type of operation
08.     if type of operation is 'insert'
09.         input position and value
10.         Insert(position, value)
11.     else if type of operation is 'delete'
12.         input position
13.         Delete(position)
14.     else if type of operation is 'replace'
15.         input position and value
16.         Replace(position, value)
17.     else if type of operation is 'query'
18.         input lower bound, upper bound and a constant
19.         output the result of the query
20.     decrement number of operation by one
```

Gambar 3.1 Pseudocode Fungsi Main.

3.2 Desain Algoritma

Pada subbab ini akan dijelaskan fungsi-fungsi yang digunakan untuk penyelesaian masalah.

3.2.1 Desain Fungsi Preprocess

```
Preprocess()
01. for i = 0 to 10
02.     for j = 0 to min(i,10)
03.         if j = 0 or j = i
04.             coeff[i][j] = 1
05.         else
06.             coeff[i][j] = c[i-1][j-1] + c[i-1][j];
07. for i = 0 to 200000
08.     power[i][0] = 1
09. for i = 0 to 200000
10.     for j = 0 to 200000
11.         power[i][j] = power[i][j-1] * i
```

Gambar 3.2 Pseudocode Fungsi Preprocess

Pada fungsi Preprocess dilakukan perhitungan yang disimpan pada suatu variabel yang dibutuhkan untuk perhitungan *query* barisan dinamis. Pada dasarnya dilakukan perhitungan segitiga pascal dan perhitungan pangkat. *Pseudocode* dari fungsi tersebut dapat dilihat pada Gambar 3.2.

3.2.2 Desain Fungsi Rotate

Pada fungsi Rotate didesain seperti yang dijelaskan pada bab 2.5 dan dilakukan update *size* dari *node* yang mengalami perubahan. Pada fungsi ini sudah termasuk kondisi rotasi kiri dan rotasi kanan. Untuk *pseudocode* dari fungsi tersebut dapat dilihat pada .

```
/**
 * _node is the element to be rotated
 */
Rotate(_node)
```

```
01. nodeParent = parent of _node
02. nodeGrandParent = parent of nodeParent
03. if node is a left child
04.     nodeChild = right child of _node
05. else
06.     nodeChild = left child of _node
07. parent of _node = nodeGrandParent
08. parent of nodeParent = _node
09. if nodeChild is not null
10.     parent of nodeChild = nodeParent
11. if nodeGrandParent is not null
12.     if nodeParent is a left child
13.         left child of nodeGrandParent = _node
14.     else
15.         right child of nodeGrandParent = _node
16. if _node is a left child
17.     right child of _node = nodeParent
18.     left child of nodeParent = nodeChild
19. else
20.     lef child of _node = nodeParent
21.     right child of nodeParent = nodeChild
22. update nodeParent
23. update _node
```

Gambar 3.3 Pseudocode Fungsi Rotate

3.2.3 Desain Fungsi Splay

```
/**
 * _node is the element to be splayed
 */
Splay(_node)
```

```
01. while _node is not root
02.     if parent of _node is root
03.         rotate _node
04.     else if _node is a left child and parent of _node is a
left child
05.         rotate parent of _node
06.         rotate _node
07.     else if _node is a right child and parent of _node is a
right child
08.         rotate parent of _node
09.         rotate _node
10.     else
11.         rotate _node
12.         rotate _node
13. root = node
```

Gambar 3.4 Pseudocode Fungsi Splay

Dalam fungsi Splay akan diterapkan sesuai dengan yang sudah dijelaskan pada Bab 2.5.1 di mana terdapat 3 kondisi yaitu zig-zig, zig-zag dan zig.

Pada baris 1 menunjukkan kondisi *node* selama belum menjadi *root*. Pada baris 2 hingga baris 3 ialah kondisi zig di mana hanya dilakukan lokasi terakhir. Pada baris 4 hingga baris 9 ialah kondisi zig-zig, sehingga dilakukan rotasi pada parent node dan rotasi pada node tersebut dan pada baris 10 hingga 12 ialah kondisi zig-zag dan dilakukan rotasi dua kali pada node tersebut. Pada baris akhir dilakukan perubahan pointer root pada node yang baru dilakukan operasi splay.

3.2.4 Desain Fungsi FindPosition

Karena pada *tree* yang dibuat menggunakan konsep key yang implisit maka digunakan fungsi pencarian node pada suatu posisi,

fungsi tersebut ialah FindPosition, pada fungsi ini menerima input parameter sebuah *node* yang menjadi awal pencarian dan *pos* di mana nilai posisi yang akan dicari. Fungsi ini mengembalikan alamat node yang dicari *Pseudocode* dari fungsi FindPosition dapat dilihat pada .

```
/**
 * _node is the element to be checked
 * pos is the destination of position
 */
FindPosition(_node, pos)
1. sizeOfNode = size of left child of _node
2. if sizeOfNode = pos
3.     return _node
4. else if sizeOfNode > pos
5.     return FindPosition(left child of _node, pos)
6. else
7.     return FindPosition(right child of _node, pos -
sizeOfNode - 1)
```

Gambar 3.5 Pseudocode Fungsi FindPosition

3.2.5 Desain Fungsi Insert

Pada fungsi insert yaitu dilakukan penyisipan sebuah *node* pada suatu posisi yang ditentukan, langkah awal pada fungsi tersebut ialah mengecek apakah tree sudah memiliki sebuah *node* atau belum dan menentukan apakah *node* baru akan disisipkan sebelum root, atau disisipkan setelah *root*. Untuk *pseudocode* dari fungsi Insert dapat dilihat pada Gambar 3.2 di mana pada fungsi tersebut menerima posisi untuk lokasi di mana *node* tersebut akan disisipkan dan *key* sebagai nilai pada *node* tersebut.

Pada baris 1 ialah kondisi di mana *tree* sudah memiliki *node* di dalamnya dan akan disisipkan sebuah *node* baru pada posisi *pos,* untuk baris 2 hingga 5 ialah kondisi di mana *node* baru akan disisipkan sebelum posisi yang ditentukan dan pada baris 6 hingga baris 10 dilakukan penyisipan *node* pada akhir barisan atau setelah *root*. Pada baris 11 dan 12 ialah kondisi *tree* belum memiliki *node*

sama sekali. Setiap *node* yang baru disisipkan dibuat menjadi *root* pada *tree*.

Pada fungsi Insert tersebut membutuhkan 2 fungsi penyisipan yang berbeda, yaitu InsertBefore dan InsertAfter, di mana InsertBefore dibutuhkan untuk penyisipan bilangan ditengah barisan atau awal barisan dan InsertAfter dibutuhkan untuk penyisipan bilangan di akhir barisan.

Pada fungsi InsertBefore dibutuhkan parameter sebuah alamat dari *node* tujuan atau *node* yang akan disisipkan *node* baru sebelum *node* yang dijadikan parameter sehingga *node* tujuan menjadi anak kanan dari *node* baru. Fungsi ini mengembalikan nilai alamat dari *node* baru.

Pada fungsi InsertAfter dibutuhkan parameter yang sama dengan fungi InsertBefore dan mengembalikan alamat dari *node* baru namun fungsi ini menghasilkan *node* tujuan menjadi anak kiri dari *node* baru.

```
/**
 * pos is the position where node is tobe inserted
 * key is the value of the node
 */
Insert(pos, key)
```

```
01. if root is not null
02.     if pos < size of root
03.         _node = find node with position pos
04.         splay _node to the root
05.         root = insert new node with key as its value before
root
06.     else
07.         rootSize = size of root - 1
08.         _node = find node with position rootSize
09.         splay _node to the root
10.         root = insert new node with key as its value after
root
11. else
12.     root = new node with key as its value
```

Gambar 3.6 Pseudocode Fungsi Insert

```
/**
 * _node is the next element after new element inserted
 * key is the value of new element
 */
InsertBefore(_node, key)
1. newNode = new node with value key
2. right child of newNode = _node
3. left child of newNode = left child of _node
4. parent of _node = newNode
5. left child of _node = null
6. update _node
7. update newNode
8. return newNode
```

Gambar 3.7 Pseudocode Fungsi InsertBefore

```
/**
 * _node is the previous element after new element inserted
 * key is the value of new element
 */
InsertAfter(_node, key)
1. newNode = new node with key as its value
2. left child of newNode = _node
3. parent of _node = newNode
4. update newNode
5. return newNode
```

Gambar 3.8 Pseudocode Fungsi InsertAfter

3.2.6 Desain Fungsi Delete

Fungsi Delete dilakukan dengan langkah awal yaitu mencari *node* yang akan dihapus dan dilakukan operasi splay pada node tersebut sehingga node tersebut menjadi root pada tree. Setelah dilakukan operasi splay, dicari *node* pengganti untuk menempatkan posisi *node* yang akan dihapus, *node* pengganti tersebut bisa dari *node* maksimum yang berada pada anak kiri *node* atau *node* minimum yang berada pada anak kanan *node*.

Pada baris 1 dan 2 dilakukan pencarian *node* sesuai posisi dan dilakukan operasi *splay* pada node tersebut. Setelah *node* tersebut menjadi *root* pada *tree* dilakukan penentuan *node* pengganti untuk

menempati posisi *node* yang akan dihapus dengan kondisi seperti pada baris 3 dan 14 di mana pada baris 3 menentukan *node* pengganti ialah *node* maksimum yang ada pada anak kiri, dan baris 14 menentukan *node* pengganti ialah *node* minimum yang ada pada anak kanan. Pada baris 7 dan 18 ialah melakukan perpindahan pada *node* pengganti menjadi salah satu anak dari *node* yang akan dihapus dengan konsep *splay.* Setelah itu dilakukan update pada *node* untuk memperbarui informasi-informasi pada *node* pengganti.

```
/**
 * pos is the position of node to be deleted
 */
Delete(pos)
```

```
01. node = find node with position pos
02. splay node to the root
03. if left child of node is not null
04.     maxNode = left child of node
05.     while right child of maxNode is not null
06.         maxNode = right child of maxNode
07.     splay maxNode to left child of node
08.     right child of maxNode = right child of node
09.     if right child of node is not null
10.         parent of right child of node = maxNode
11.     parent of maxNode = null
12.     update maxNode
13.     root = maxNode
14. else if right child of node is not null
15.     minNode = right child of node
16.     while left child of minNode is not null
17.         minNode = left child of minNode
18.     splay minNode to right child of node
19.     left child of minNode = left child of node
20.     if left child of node is not null
21.         parent of left child of node = minNode
22.     parent of minNode = null
23.     update minNode
24.     root = minNode
```

Gambar 3.9 Pseudocode Fungsi Delete

3.2.7 Desain Fungsi Replace

Pada fungsi Replace dilakukan perubahan nilai pada sebuah *node* yang ditentukan, langkah awalnya ialah mencari *node* pada posisi yang ditentukan, setelah ditemukan nilai pada *node* tersebut diubah dengan suatu nilai yang diberikan dan dilakukan operasi *splay* pada *node* tersebut. Untuk *pseudocode* dari fungsi Replace dapat dilihat pada Gambar.

```
/**
 * pos is the position where node is tobe changed
 * key is the next value of node
 */
Replace(pos, key)
1. _node = find node at position pos
2. value of _node = key
3. splay _node to root
```

Gambar 3.10 Pseudocode Fungsi Replace

BAB IV
IMPLEMENTASI

Pada bab ini penulis menjelaskan implementasi sesuai dengan desain algoritma yang serta struktur data yang telah dilakukan.

4.1 Lingkungan Implementasi

Lingkungan implementasi yang digunakan ialah sebagai berikut:

1. Perangkat Keras:
   Processor Inter® Core(TM) i5-2410M CPU @ 2.30Ghz
   RAM 4.00 GB
   64-bit Operating System, x64 based processor
2. Perangkat Lunak:
   Operating System Windows 8.1 Pro 64 bit
   Text Editor Sublime Text 3
   Compiler g++ (tdm64-2) 4.8.1

4.2 Implementasi Fungsi Preprocess

Fungsi preprocess diterapkan sesuai dengan yang sudah dibahas pada subbab 3.2.1. Implementasi dapat dilihat pada Kode Sumber 4.1.

```
01. inline void preprocess(){
02.     for(int i=0 ; i <= 10 ; i++){
03.         for(int j=0 ; j <= min(i,10) ; j++){
04.             if(j == 0 || j == i)
05.                 c[i][j] = 1;
06.             else
07.                 c[i][j] = c[i-1][j-1] + c[i-1][j];
08.         }
09.     }
10.     for(int i=0 ; i<200000 ; i++) power[i][0] = 1;
11.     for(int i=0 ; i<200000 ; i++)
12.         for(int j=1 ; j<= 10 ; j++)
13.             power[i][j] = (power[i][j-1] *
(unsigned)i);
14. }
```

Kode Sumber 4.1 Implementasi Fungsi Preprocess

4.3 Implementasi Struct Node

Struct Node digunakan untuk merepresentasikan sebuah *node* pada *tree*. Terdapat beberapa member dan fungsi pada struct tersebut sesuai dengan kebutuhan penyelesaian masalah, sebagian fungsi sesuai dengan apa yang sudah dijelaskan pada bab 3.

```
01. typedef struct Node{
02.     ui key;
03.     ui sum[11];
04.     int size;
05.     Node *left, *right, *parent;
06.     Node(){}
07.     Node(ui key) : key(key), size(1), left(0),
right(0), parent(0){}
08.     void UpdateSize();
09.     void UpdateSum();
10.     void Update();
11.     bool isLeftChild();
12.     bool isRightChild();
13.     bool isRoot();
14.     void Rotate();
15.     void Zig();
16.     void ZigZig();
17.     void ZigZag();
18.     void Splay();
19.     void SplayUntil(Node* until);
20.     Node* FindPosition(int pos);
21.     Node* InsertBefore(ui key);
22.     Node* InsertAfter(ui key);
23.     void Insert(int pos, ui key);
24.     void Delete();
25.     void Delete(int pos);
26.     void Replace(ui key);
27. }Node;
```

Kode Sumber 4.2 Implementasi Struct Node (1)

Pada kode sumber diatas dilihat atribut-atribut dari *struct* Node dan prototype dari fungsi-fungsi yang dimiliki oleh *struct* Node. Terdapat variabel key yang berguna untuk menyimpan nilai yang merepresentasikan bilangan pada barisan, variabel sum ialah untuk penyimpanan perhitungan dari penjumlahan untuk setiap pangkat, size untuk merepresentasikan size dari *sub tree* pada *node*

tersebut. Beberapa pointer untuk mengarahkan pada *parent*, anak kiri dan anak kanan dari *node* tersebut.

```
001. inline bool Node::isLeftChild(){
002.  return (this->parent != NULL && this->parent->left
== this);
003. }
004. inline bool Node::isRightChild(){
005.  return (this->parent != NULL && this->parent->right
== this);
006. }
007. inline bool Node::isRoot(){
008.  return this->parent == NULL;
009. }
```

Kode Sumber 4.3 Implementasi Struct Node (2)

Fungsi isLeftChild menentukan apakan *node* yang memanggil fungsi tersebut sebuah anak kiri dari parent nya atau tidak, fungsi isRightChild menentukan apakah *node* tersebut termasuk sebuah anak kanan atau tidak dan fungsi isRoot menentukan apakah *node* tersebut termasuk sebuah *root* pada *tree*.

```
010. inline void Node::UpdateSize(){
011.  this->size = node_size(this->left) + 1 +
node_size(this->right);
012. }
013. inline void Node::UpdateSum(){
014.  for(int i=0 ; i<=10 ; i++){
015.        this->sum[i] = this->left?this->left-
>sum[i]:0;
016.        this->sum[i] += power[node_size(this-
>left)][i] * this->key;
017.        this->sum[i] += (this->right? query(this-
>right, node_size(this->left)+1, i) : 0);
018.  }
019. }
020. inline void Node::Update(){
021.  this->UpdateSize();
022.  this->UpdateSum();
023. }
```

Kode Sumber 4.4 Implementasi Struct Node (3)

Fungsi Update memanggil kedua fungsi UpdateSize dan UpdateSum, di mana pada UpdateSize dilakukan perubahan nilai

pada *size* dari *subtree* pada *node* tersebut dan pada UpdateSum dilakukan perubahan nilai penjumlahan untuk setiap pangkat.

```
024. inline void Node::Rotate(){
025.  if(isLeftChild()){
026.          bool parWasLeftChild = this->parent?this-
>parent->isLeftChild():false;
027.          parent->left = this->right;
028.          if(this->right)
029.              this->right->parent = this->parent;
030.          Node* newParent = this->parent->parent;
031.          this->right = this->parent;
032.          if(this->right)
033.              this->right->parent = this;
034.          this->parent = newParent;
035.          if(this->parent != NULL){
036.              if(parWasLeftChild)
037.                  this->parent->left = this;
038.              else
039.                  this->parent->right = this;
040.          }
041.          this->right->Update();
042.          this->Update();
043.      }
```

Kode Sumber 4.5 Implementasi Struct Node (4)

Fungsi Rotate diterapkan sesuai dengan desain pada bab sebelumnya yang membahas fungsi Rotate, di mana pada fungsi ini dipecah menjadi dua kondisi yaitu rotasi kanan dan rotasi kiri, pada Kode Sumber diatas ialah pada kondisi rotasi kanan, di mana kondisi tersebut mengembalikan nilai benar saat dipanggil fungsi isLeftChild maka kondisi tersebut menunjukkan syarat dari rotasi kanan. Pada fungsi ini juga memanggil fungsi Update pada *node-node* yang terpengaruh dari proses rotasi, di mana pada rotasi kanan yang terpengaruh dari rotasi tersebut ialah yang menjadi anak kanan pada *node* tersebut dan *node* itu sendiri.

Pada Kode Sumber diatas ialah fungsi Rotate saat kondisi rotasi kiri, di mana kondisi tersebut bernilai benar saat dipanggil fungsi isRightChild maka kondisi tersebut menunjukkan syarat

dari rotasi kiri. Fungsi ini juga memanggil fungsi Update diakhir proses.

```
044.     else if(isRightChild()){
045.         bool parWasLeftChild = this->parent?this-
>parent->isLeftChild():false;
046.         parent->right = this->left
047.         if(this->left)
048.             this->left->parent = parent;
049.         Node* newParent = this->parent->parent;
050.         this->left = this->parent;
051.         if(this->left)
052.             this->left->parent = this;
053.         this->parent = newParent;
054.         if(this->parent != NULL){
055.             if(parWasLeftChild)
056.                 this->parent->left = this;
057.             else
058.                 this->parent->right = this;
059.         }
060.         this->left->Update();
061.         this->Update();
062.     }
063. }
```

Kode Sumber 4.6 Implementasi Struct Node (5)

```
064. inline void Node::Zig(){
065.  this->Rotate();
066. }
067. inline void Node::ZigZig(){
068.  this->parent->Rotate();
069.  this->Rotate();
070. }
071. inline void Node::ZigZag(){
072.  this->Rotate();
073.  this->Rotate();
074. }
```

Kode Sumber 4.7 Implementasi Struct Node (6)

Pada Kode Sumber diatas terdapat fungsi Zig, ZigZig dan ZigZag di mana fungsi tersebut mengimplementasikan kondisi zig,zig-zig, dan zig-zag sesuai pada Bab 2.5.1.

Pada Fungsi Splay diterapkan sesuai desain pada Bab 3.2.3 di mana fungsi tersebut memanggil fungsi Zig, ZigZig dan ZigZag

sesuai kondisi nya. Perbedaannya dengan fungsi SplayUntil, pada fungsi SplayUntil fungsi tersebut menerima parameter pointer suatu *node* yang menunjukkan akhir dari proses *splay*.

```
075. inline void Node::Splay(){
076.  while( !this->isRoot() ){
077.        if(this->parent->isRoot())
078.              this->Zig();
079.        else if(this->isLeftChild() == this->parent-
>isLeftChild())
080.              this->ZigZig();
081.        else
082.              this->ZigZag();
083.  }
084.  root = this;
085. }
086. inline void Node::SplayUntil(Node* until)
087. {
088.     Node* grandParent = until->parent;
089.     while(this->parent != grandParent){
090.         if(this->parent == until)
091.              this->Zig();
092.         else if(this->isLeftChild() == this->parent-
>isLeftChild())
093.              this->ZigZig();
094.         else
095.              this->ZigZag();
096.     }
097. }
```

Kode Sumber 4.8 Implementasi Struct Node (7)

```
098. inline Node* Node::FindPosition(int pos){
099.  int size = node_size(this->left);
100.  if(size == pos)
101.        return this;
102.  else if(size > pos)
103.        return this->left->FindPosition(pos);
104.  else
105.        return this->right->FindPosition(pos - size -
1);
106. }
```

Kode Sumber 4.9 Implementasi Struct Node (8)

Fungsi FindPosition bertujuan untuk mencari suatu *node* dengan representasi posisi yang dicari, di mana pada fungsi ini

memiliki parameter bilangan integer yang mewakili posisi *node* yang dicari dan pada fungsi ini akan mengembalikan alamat dari *node* dengan posisi yang dituju.

```
107. inline Node* Node::InsertBefore(ui key){
108.  Node* node = new Node(key);
109.  node->right = this;
110.  node->left = this->left;
111.  if(node->left)
112.        node->left->parent = node;
113.  this->parent = node;
114.  this->left = NULL;
115.  this->Update();
116.  node->Update();
117.  return node;
118. }
119. inline Node* Node::InsertAfter(ui key){
120.  Node* node = new Node(key);
121.  node->left = this;
122.  this->parent = node;
123.  node->Update();
124.  return node;
125. }
```

Kode Sumber 4.10 Implementasi Struct Node (9)

```
126. inline void Node::Insert(int pos, ui key){
127.  if(pos < node_size(root)){
128.        this->FindPosition(pos)->Splay();
129.        root = root->InsertBefore(key);
130.  }
131.  else {
132.        this->FindPosition(node_size(root)-1)-
>Splay();
133.        root = root->InsertAfter(key);
134.  }
135. }
```

Kode Sumber 4.11 Implementasi Struct Node (10)

Fungsi InsertBefore dan InsertAfter ialah implementasi dari desain yang dijelaskan pada Bab 3.2.5 di mana fungsi tersebut akan dipanggil pada fungsi Insert sesuai dengan kondisi yang dibutuhkan. Fungsi InsertBefore digunakan untuk penyisipan sebuah *node* baru yang merepresentasikan penyisipan sebelum

suatu bilangan pada barisan. Fungsi InsertAfter digunakan untuk penyisipan sebuah *node* baru yang merepresentasikan penyisipan setelah suatu bilangan pada barisan.

Fungsi Insert ialah fungsi yang menentukan kondisi untuk pemanggilan fungsi InsertBefore atau InsertAfter sesuai desain pada Bab 3.2.5, di mana fungsi ini menerima parameter pos yang merepresentasikan posisi penyisipan *node* baru dan *key* yang merepresentasikan nilai dari *node* baru.

```
136. inline void Node::Delete(){
137.  if(this->left){
138.        Node* maxNode = this->left;
139.        while(maxNode->right)
140.              maxNode = maxNode->right;
141.        maxNode->SplayUntil(this->left);
142.        maxNode->right = this->right;
143.        if(this->right)
144.              this->right->parent = maxNode;
145.        maxNode->parent = NULL;
146.        maxNode->Update();
147.        root = maxNode;
148.  }
```

Kode Sumber 4.12 Implementasi Struct Node (11)

```
149.  else if(this->right){
150.        Node* minNode = this->right;
151.        while(minNode->left)
152.              minNode = minNode->left;
153.        minNode->SplayUntil(this->right);
154.        minNode->left = this->left;
155.        if(this->left)
156.              this->left->parent = minNode;
157.        minNode->parent = NULL;
158.        minNode->Update();
159.        root = minNode;
160.  }
161. }
```

Kode Sumber 4.13 Implementasi Struct Node (12)

Fungsi Delete menerapkan desain yang sudah dibahas pada Bab 3.2.6. Pada Kode Sumber 4.12 ialah kondisi di mana pengganti dari posisi *node* yang dihapus ialah berasal dari anak kiri pada *node*

tersebut dan dicari *node* yang paling maksimum atau terbesar sesuai representasi posisi.

Pada Kode Sumber 4.13 menjalankan kondisi di mana tidak ada anak kiri dari *node* yang akan dihapus maka dari itu dilakukan pengambilan *node* minimal atau terkecil yang terdapat pada anak kanan dari *node* yang akan dihapus.

```
162. inline void Node::Delete(int pos){
163.  this->FindPosition(pos)->Splay();
164.  root->Delete();
165. }
```

Kode Sumber 4.14 Implementasi Struct Node (13)

Fungsi Delete dengan menerima parameter pos yang mewakilkan posisi dari suatu bilangan yang akan dihapus melakukan pencarian *node* yang bersangkutan dan memanggil fungsi Delete yang sebelumnya. Implementasi dapat dilihat pada Kode Sumber 4.14.

Pada fungsi Replace dilakukan sesuai dengan desain yang dijelaskan pada Bab 3.2.7. Pada fungsi tersebut sebelumnya dilakukan pencarian pada posisi yang akan diubah nilainya dengan memanggil fungsi FindPosition, setelah itu dilakukan perubahan nilai pada *node*. Implementasi fungsi tersebut dapat dilihat pada

```
166. inline void Node::Replace(ui key){
167.  this->key = key;
168.  this->Splay();
169.  root = this;
170. }
```

Kode Sumber 4.15 Implementasi Struct Node (14)

4.4 Implementasi Fungsi Query

Pada subbab ini akan dijelaskan implementasi dari fungsi query. Fungsi ini melakukan perhitungan pada *node-node* yang terlibat untuk suatu proses *query* di mana diberikan batas kiri dan batas kanan dari suatu barisan yang akan dihitung, dan fungsi ini

melakukan perhitungan untuk dari *node* yang merepresentasikan batas kiri hingga *node* yang merepresentasikan batas kanan yang diberikan. Implementasi fungsi tersebut dapat dilihat pada Kode Sumber 4.16.

Fungsi query ini dibuat menjadi dua di mana fungsi query yang pertama dibuat untuk perhitungan pada sebuah *node* dan fungsi query yang kedua di mana menerima parameter batas kiri dan batas kanan melakukan perhitungan untuk suatu *range* dari batas-batas yang sudah diberikan, fungsi ini juga memanggil fungsi query yang pertama di mana suatu kondisi diperlukan untuk perhitungan pada sebuah *node*.

```
01. inline ui query(Node *t, ui delta, ui k){
02.     ui res = 0;
03.     for(int i=0 ; i<=k ; i++){
04.         res += c[k][i] * power[delta][k-i] * t-
>sum[i] ;
05.     }
06.     return res;
07. }
08. inline ui query(int lo, int hi, int a, int b, ui k,
Node* t){
09.     if( t == NULL || a > hi || b < lo ) return 0;
10.     if(a >= lo && b <= hi) return query(t,a-lo+1,k);
11.     int left_a = a;
12.     int left_b = a + node_size(t->left) - 1;
13.     int this_node = left_b + 1 ;
14.     int right_a = this_node + 1;
15.     int right_b = b;
16.     ui res = (this_node >= lo && this_node <= hi) ?
(power[this_node-lo+1][k] * t->key) : 0;
17.     res += query(lo,hi, left_a, left_b, k , t->left)
+ query(lo,hi, right_a, right_b, k , t->right);
18.     return res;
19. }
```

Kode Sumber 4.16 Kode Sumber Fungsi Query

BAB V
UJI COBA DAN EVALUASI

Pada bab ini penulis menjelaskan tentang uji coba dan evaluasi dari implementasi yang dilakukan.

5.1 Lingkungan Uji Coba

Lingkungan uji coba yang digunakan adalah sebagai berikut:

1. Perangkat Keras:
   Processor Inter® Core(TM) i5-2410M CPU @ 2.30Ghz
   RAM 4.00 GB
   64-bit Operating System, x64 based processor
2. Perangkat Lunak:
   Operating System Windows 8.1 Pro 64 bit
   Text Editor Sublime Text 3
   Compiler g++ (tdm64-2) 4.8.1

5.2 Skenario Uji Coba

Pada subbab ini akan dijelaskan skenario uji coba yang dilakukan. Skenario uji coba terdiri dari uji coba kebenaran dan uji coba kinerja.

5.2.1 Uji Coba Kebenaran

Uji coba kebenaran dilakukan dengan melakukan perbandingan dengan keluaran yang dihasilkan oleh program yang sudah dibuat dibandingkan dengan contoh dari masukan dan keluaran yang terdapat pada soal dan juga dengan mengirim kode sumber implementasi yang sudah dibuat ke situs SPOJ. Permasalahan yang akan diselesaikan SPOJ 19543 - GSS8, *Can you answer these queries VIII* seperti yang sudah dijelaskan pada subbab 2.7. Hasil keluaran dari masukan yang dibuat oleh program

dapat dilihat pada Gambar 5.1. Contoh masukan dan keluaran pada permasalahan dapat dilihat pada Gambar 5.2.

```
4
1 2 3 5
7
Q 0 2 0
6
I 3 4
Q 2 4 1
26
D 0
Q 0 3 1
40
R 1 2
Q 0 1 0
4

--------------------------------
Press any key to continue . . .
```

Gambar 5.1 Hasil Keluaran Program Sesuai Contoh Masukkan pada Permasalahan

Example

Input:

```
4
1 2 3 5
7
Q 0 2 0
I 3 4
Q 2 4 1
D 0
Q 0 3 1
R 1 2
Q 0 1 0
```

Output:

```
6
26
40
4
```

Gambar 5.2 Contoh Masukkan dan Keluaran pada Permasalahan

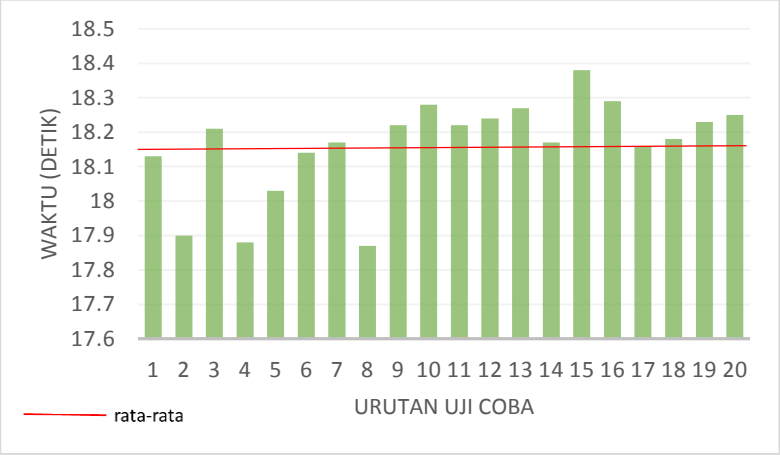
Setelah perbandingan program dengan contoh masukan dan keluaran sesuai, kode sumber implementasi dikirimkan ke situs SPOJ, sistem pada situs tersebut akan memberikan umpan balik terhadap kode sumber seperti yang dijelaskan pada subbab 2.6. Hasil uji coba pada situs SPOJ dapat dilihat pada Gambar 5.3.

14270120		2015-05-18 00:55:53	Can you answer these queries VIII	accepted edit run	18.25	25M	C++ 4.3.2

Gambar 5.3 Hasil Uji Coba pada Situs SPOJ

Dari hasil uji coba yang dilakukan, kode sumber program yang dikirimkan mendapat umpan balik *Accepted*. Waktu yang dibutuhkan program adalah 18.25 detik dan memori yang dibutuhkan program ialah 25 MB. Hal tersebut membuktikan bahwa implementasi yang telah dilakukan berhasil menyelesaikan permasalahan query barisan dinamis.

Setelah itu dilakukan pengiriman kode sumber sebanyak 20 kali untuk melihat variasi waktu dan memori yang dibutuhkan program. Hasil dari pengiriman kode sumber dapat dilihat pada Gambar 5.4.



Gambar 5.4 Grafik Hasil Uji Coba pada Situs SPOJ Sebanyak 20 Kali

Dari hasil uji coba yang telah dilakukan, seluruh kode sumber program yang dikirimkan pada situs SPOJ mendapat umpan balik *Accepted*. Waktu yang dibutuhkan program minimum 17.87 detik, maksimum 18.38 detik dan rata-rata 18.16 detik. Memori yang dibutuhkan program tetap 25 MB.

5.2.2 Uji Coba Kinerja

Untuk struktur data Splay Tree dengan terdapat M jumlah operasi dan N jumlah *node* mempunyai waktu kompleksitas $O(M \log N)$ *amortized time* untuk setiap perubahan pada *node*. Uji coba kinerja dilakukan dengan membuat Splay Tree secara acak dengan penyisipan jumlah node awal sebanyak 100 dan terdapat dua skenario yang diuji pada uji coba kinerja ini, yang pertama ialah dilakukan operasi penyisipan saja di mana mempengaruhi banyaknya operasi dan banyaknya node pada *tree* dan yang kedua dilakukan operasi penyisipan juga operasi penyisipan *node* dan juga operasi perubahan nilai pada suatu node di mana pada skenario kedua ini memiliki jumlah *node* akhir yang sama pada skenario pertama, perbedaan terdapat pada banyaknya jumlah operasi yang dilakukan pada Splay Tree.

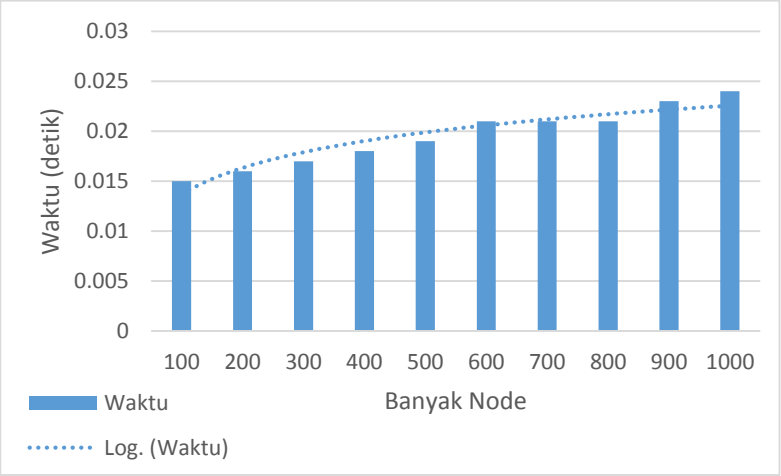
Setelah itu dilakukan komparasi terhadap kedua skenario untuk melihat pengaruh dari perbedaan yang dihasilkan dari kedua skenario tersebut. Di mana yang skenario pertama memperhatikan pengaruh banyak *node* pada waktu dan skenario kedua memperhatikan pengaruh banyaknya operasi yang berjumlah lebih banyak daripada skenario pertama.

5.2.2.1 Pengaruh Banyaknya Node Terhadap Waktu

Banyaknya penyisipan *node* awal dibuat tetap yaitu 100 operasi penyisipan awal dan dilakukan operasi penyisipan node dengan jumlah 100 hingga 1.000 dengan rentang 100. Dicatat waktu yang dibutuhkan program untuk setiap Splay Tree yang dibuat. Hasil uji coba dapat dilihat pada Gambar 5.5.

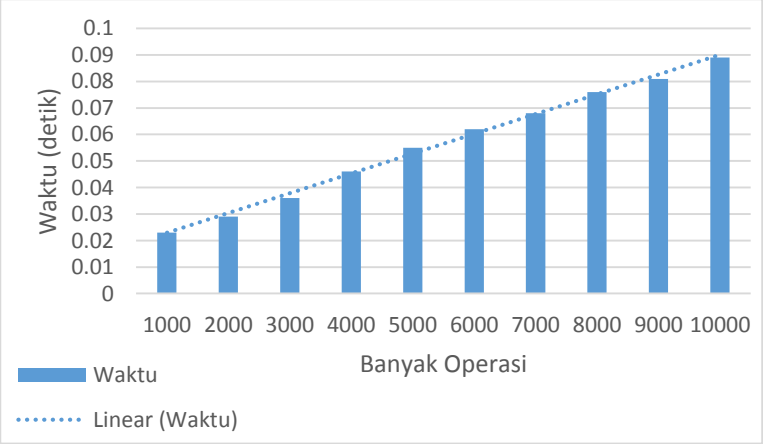
Dari hasil uji coba yang dilakukan dapat dilihat pertumbuhan waktu yang dibutuhkan program mendekati kurva logaritmik seiring dengan pertumbuhan banyak *node*. Hal tersebut membuktikan bahwa implementasi yang telah dilakukan sesuai

dengan waktu kompleksitas dari Splay Tree yang dipengaruhi banyak node secara logaritmik.



Gambar 5.5 Grafik Hasil Uji Coba Pengaruh Banyaknya Node Terhadap Waktu

5.2.2.2 Pengaruh Banyak Operasi Terhadap Waktu

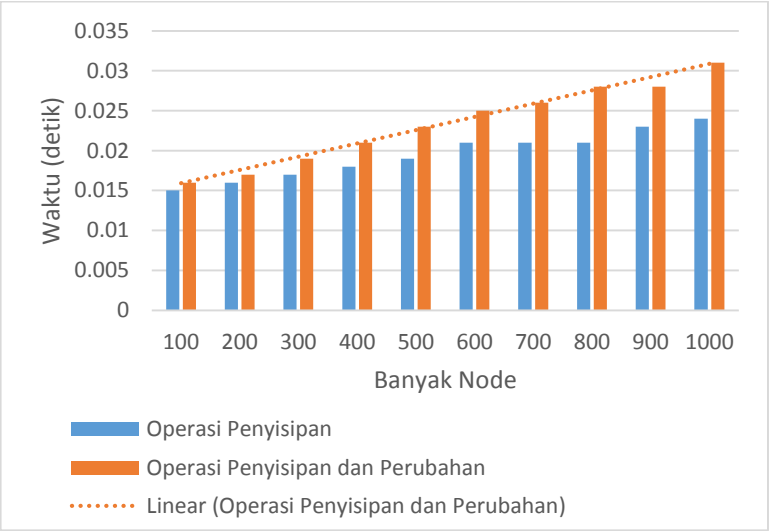


Gambar 5.6 Grafik Hasil Uji Coba Pengaruh Banyaknya Operasi Terhadap Waktu

Banyaknya penyisipan *node* awal dibuat tetap yaitu 100 operasi penyisipan awal dan dilakukan operasi perubahan node dengan jumlah 1.000 hingga 10.000 dengan rentang 1.000. *Node* yang diubah dibuat acak. Dicatat waktu yang dibutuhkan program untuk setiap Splay Tree yang dibuat. Hasil uji coba dapat dilihat pada Gambar 5.6.

Dari hasil uji coba yang dilakukan dapat dilihat pertumbuhan waktu yang dibutuhkan program mendekati kurva linier seiring dengan pertumbuhan banyak operasi. Hal tersebut membuktikan bahwa implementasi yang telah dilakukan sesuai dengan waktu kompleksitas dari Splay Tree yang dipengaruhi banyak operasi secara linier.

5.2.2.3 Pengaruh Banyak Node dan Operasi Terhadap Waktu



Gambar 5.7 Grafik Hasil Komparasi Kedua Uji Coba yang Sudah Dilakukan

Pada kali ini diuji dengan menggunakan operasi dua kali dari sebelumnya dan operasi tersebut meliputi penyisipan dan

perubahan nilai pada suatu *node*. Jumlah penyisipan awal tetap dibuat sama yaitu 100. Banyak operasi yang dilakukan ialah 0 hingga 1.800 dengan rentang 200. Dicatat waktu yang dibutuhkan program untuk setiap Splay Tree yang dibuat. Hasil uji coba dapat dilihat pada Tabel A.4.

Setelah itu dilakukan komparasi terhadap skenario yang sudah dilakukan pada subbab 5.2.2.1. Di mana pada subbab tersebut hanya dilakukan operasi penyisipan saja dan jumlah operasi tersebut berkisar setengah dari jumlah operasi uji kali ini. Hasil komparasi dapat dilihat pada Gambar 5.7.

Dari hasil uji coba yang dilakukan dapat dilihat bahwa uji coba kali ini membutuhkan waktu yang lebih besar dari sebelumnya dan dapat dilihat pula pertumbuhan waktu yang dibutuhkan program seiring perubahan operasi dan node mendekati kurva linier. Hal tersebut membuktikan bahwa implementasi yang telah dilakukan sesuai dengan waktu kompleksitas dari Splay Tree yang lebih dipengaruhi oleh banyaknya operasi secara linier.

BAB VI
KESIMPULAN DAN SARAN

Pada bab ini penulis menjelaskan kesimpulan dari hasil uji coba yang telah dilakukan dan saran mengenai hal-hal yang masih bisa untuk dikembangkan.

6.1 Kesimpulan

Dari hasil uji coba yang telah dilakukan terhadap implementasi solusi untuk permasalahan query barisan dinamis dengan menggunakan Splay Tree dapat diambil kesimpulan sebagai berikut:

1. Struktur data Splay Tree dapat merepresentasikan suatu barisan dan mendukung operasi-operasi yang dibutuhkan pada permasalahan *query* barisan dinamis seperti penyisipan bilangan, penghapusan bilangan, perubahan bilangan dan perhitungan pada suatu jarak pada barisan.
2. Implementasi struktur data *self balancing binary search tree* dengan menggunakan struktur data Splay Tree yang dilakukan dapat menyelesaikan permasalahan *query* barisan dinamis sesuai dengan persoalan SPOJ 19543 - GSS8, *Can you answer these queries VIII* dengan mendapatkan umpan balik *Accepted*, waktu rata-rata 18.16 detik dan memori yang dibutuhkan ialah 25 MB dari 20 kali pengumpulan.
3. Struktur data Splay Tree pada permasalahan ini mempunyai kompleksitas waktu $O(M \log N)$ *amortized time* untuk setiap perubahan node di mana M ialah banyaknya operasi dan N banyaknya node sesuai dengan hasil uji coba kinerja yang didapat di mana waktu yang dibutuhkan program dipengaruhi dari banyaknya operasi secara linier dan banyaknya *node* secara logaritmik.

6.2 Saran

Berikut merupakan saran pengembangan implementasi solusi penyelesaian *query* barisan dinamis dengan struktur data *self balancing binary search tree*. Implementasi yang telah dilakukan menggunakan *bottom-up* Splay Tree di mana harus menelusuri ke dalam tree tersebut untuk pencarian suatu *node* sebelum dilakukan operasi *splay*. Pengembangan implementasi dapat dilakukan dengan *top-down* Splay Tree yang tidak perlu membutuhkan pointer *parent* pada setiap node dan pencarian suatu *node* bisa dilakukan dengan konsep *split* dan *merge* bersamaan dengan operasi *splay*.

BAB VI
KESIMPULAN DAN SARAN

Pada bab ini penulis menjelaskan kesimpulan dari hasil uji coba yang telah dilakukan dan saran mengenai hal-hal yang masih bisa untuk dikembangkan.

6.1 Kesimpulan

Dari hasil uji coba yang telah dilakukan terhadap implementasi solusi untuk permasalahan query barisan dinamis dengan menggunakan Splay Tree dapat diambil kesimpulan sebagai berikut:

1. Struktur data Splay Tree dapat merepresentasikan suatu barisan dan mendukung operasi-operasi yang dibutuhkan pada permasalahan *query* barisan dinamis seperti penyisipan bilangan, penghapusan bilangan, perubahan bilangan dan perhitungan pada suatu jarak pada barisan.
2. Implementasi struktur data *self balancing binary search tree* dengan menggunakan struktur data Splay Tree yang dilakukan dapat menyelesaikan permasalahan *query* barisan dinamis sesuai dengan persoalan SPOJ 19543 - GSS8, *Can you answer these queries VIII* dengan mendapatkan umpan balik *Accepted*, waktu rata-rata 18.16 detik dan memori yang dibutuhkan ialah 25 MB dari 20 kali pengumpulan.
3. Struktur data Splay Tree pada permasalahan ini mempunyai kompleksitas waktu $O(M \log N)$ *amortized time* untuk setiap perubahan node di mana M ialah banyaknya operasi dan N banyaknya node sesuai dengan hasil uji coba kinerja yang didapat di mana waktu yang dibutuhkan program dipengaruhi dari banyaknya operasi secara linier dan banyaknya *node* secara logaritmik.

6.2 Saran

Berikut merupakan saran pengembangan implementasi solusi penyelesaian *query* barisan dinamis dengan struktur data *self balancing binary search tree*. Implementasi yang telah dilakukan menggunakan *bottom-up* Splay Tree di mana harus menelusuri ke dalam tree tersebut untuk pencarian suatu *node* sebelum dilakukan operasi *splay*. Pengembangan implementasi dapat dilakukan dengan *top-down* Splay Tree yang tidak perlu membutuhkan pointer *parent* pada setiap node dan pencarian suatu *node* bisa dilakukan dengan konsep *split* dan *merge* bersamaan dengan operasi *splay*.

LAMPIRAN A

Tabel A.1 Hasil Uji Coba pada Situs SPOJ Sebanyak 20 Kali

No	Hasil	Waktu (detik)	Memori (MB)
1	Accepted	18.13	25
2	Accepted	17.90	25
3	Accepted	18.21	25
4	Accepted	17.88	25
5	Accepted	18.03	25
6	Accepted	18.14	25
7	Accepted	18.17	25
8	Accepted	17.87	25
9	Accepted	18.22	25
10	Accepted	18.28	25
11	Accepted	18.22	25
12	Accepted	18.24	25
13	Accepted	18.27	25
14	Accepted	18.17	25
15	Accepted	18.38	25
16	Accepted	18.29	25
17	Accepted	18.16	25
18	Accepted	18.18	25
19	Accepted	18.23	25
20	Accepted	18.25	25

Tabel A.2 Hasil Uji Coba Pengaruh Banyaknya Node Terhadap Waktu

No	Banyak Node	Waktu (detik)
1	100	0.015
2	200	0.016
3	300	0.017
4	400	0.018
5	500	0.019
6	600	0.021
7	700	0.021
8	800	0.021
9	900	0.023
10	1000	0.024

Tabel A.3 Hasil Uji Coba Pengaruh Banyaknya Operasi Terhadap Waktu

No	Banyak Node	Banyak Operasi	Waktu
1	100	1000	0.023
2	100	2000	0.029
3	100	3000	0.036
4	100	4000	0.046
5	100	5000	0.055
6	100	6000	0.062
7	100	7000	0.068
8	100	8000	0.076
9	100	9000	0.081
10	100	10000	0.089

Tabel A.4 Hasil Uji Coba Pengaruh Banyak Node dan Operasi Terhadap Waktu

No	Banyak Node	Banyak Operasi	Waktu
1	100	0	0.016
2	200	200	0.017
3	300	400	0.019
4	400	600	0.021
5	500	800	0.023
6	600	1000	0.025
7	700	1200	0.026
8	800	1400	0.028
9	900	1600	0.028
10	1000	1800	0.031

DAFTAR PUSTAKA

[1] D. D. Sleator and E. R. Tarjan, "**Self-Adjusting Binary Search Trees**," *Journal of the Association for Computing Machinery,* vol. 32, pp. 652-686, 1985.

[2] H. T. Cormen, E. C. Leiserson, L. R. Rivest and C. Stein, **Introduction to Algorithms Third Edition**, Cambridge, Massachusetts: The MIT Press, 2009.

[3] "**SPOJ.com - Problem GSS8**," [Online]. Available: http://www.spoj.com/problems/GSS8/.

BIODATA PENULIS

Muhammad Kemal Darmawan, lahir di Jakarta tanggal 30 Agustus 1993, anak ketiga dari 3 bersaudara. Penulis telah menempuh pendidikan formal mulai dari jenjang TK hingga S-1 TKI An-nur Bekasi (1997-1999), SDN Cipinang Melayu 03 Pagi Jakarta Timur (1999-2005), SMPN 115 Jakarta (2005-2008), SMAN 8 Jakarta (2008-2011) dan Jurusan Teknik Informatika Fakultas Teknologi Informasi (FTIf) Institut Teknologi Sepuluh Nopember (ITS) Surabaya (2011-2015).