BAB 3
Teknik Pemrograman Lanjut

3.1 Tujuan

Modul ini mengenalkan suatu teknik pemrograman yang lebih tinggi. Dalam bagian ini Anda akan mempelajari rekursif dan tipe data abstrak.

Pada akhir pembahasan, diharapkan pembaca dapat :
1. Memahami dan menggunakan rekursif
2. Mengetahui perbedaan antara stacks dan queues
2. Mengimplementasikan suatu implementasi sequensial dari stacks dan queues
3. Mengimplementasikan suatu implementasi linked dari stacks and queues
4. Menggunakan class-class *Collection* yang ada

3.2 Rekursif

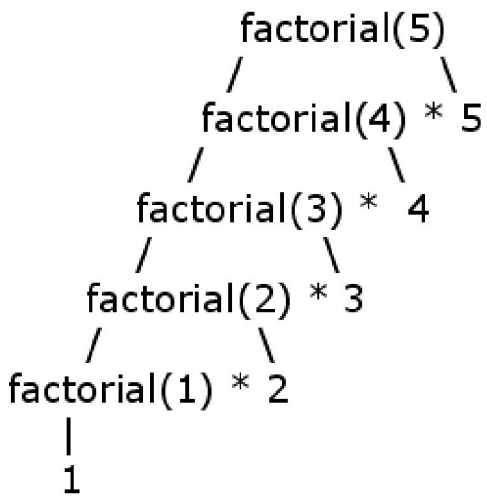
3.2.1 Apa yang dimaksud dengan Rekursif?

Rekursif adalah teknik pemecahan masalah yang powerful dan dapat digunakan ketika inti dari masalah terjadi berulang kali. Tentu saja, tipe dari masalah ini dapat dipecahkan mengunakan perkataan berulang-ulang (yaitu, menggunakan konstruksi looping seperti *for, while* dan *do-while*).
Sesungguhnya, iterasi atau perkataan berulang-ulang merupakan peralatan yang lebih efisien jika dibandingkan dengan rekursif tetapi rekursif menyediakan solusi yang lebih baik untuk suatu masalah. Pada rekursif, method dapat memanggil dirinya sendiri. Data yang berada dalam method tersebut seperti argument disimpan sementara ke dalam stack sampai method pemanggilnya diselesaikan.

3.2.2 Rekursif Vs. Iterasi

Untuk pengertian yang lebih baik dari rekursif, mari kita lihat pada bagaimana macam-macam dari teknik iterasi. Dalam teknik-teknik tersebut juga dapat kita lihat penyelesaian sebuah loop yang lebih baik menggunakan rekursif daripada iterasi.

Penyelesaian masalah dengan perulangan menggunakan iterasi secara tegas juga digunakan pada struktur kontrol pengulangan. Sementara itu, untuk rekursif, task diulangi dengan memanggil sebuah method perulangan. Maksud dari hal tersebut adalah untuk menggambarkan sebuah masalah ke dalam lingkup yang lebih kecil dari perulangan itu sendiri. Pertimbangkan penghitungan faktorial dalam penentuan bilangan bulat. Definisi rekursif dari hal tersebut dapat diuraikan sebagai berikut: factorial(n) = factorial(n-1) * n; factorial(1) = 1. Sebagai contohnya, nilai faktorial dari 2 sama dengan fatorial (1)*2, dimana hasilnya adalah 2. Faktorial dari 3 adalah 6, dimana sama dengan faktorial dari (2)*3.



Gambar 1: Contoh Factorial

Dengan iterasi, proses diakhiri ketika kondisi loop gagal atau salah. Dalam kasus dari penggunaan rekursif, proses yang berakhir dengan kondisi tertentu disebut permasalahan dasar yang telah tercukupi oleh suatu pembatasan kondisi. Permasalahan yang mendasar merupakan kejadian yang paling kecil dari sebuah masalah. Sebagai contoh, dapat dilihat pada kondisi rekursif pada faktorial, kasus yang mudah adalah ketika masukannya adalah 1. 1 dalam kasus ini merupakan dasar dari masalah.

Penggunaan dari iterasi dan rekursif dapat bersama-sama memandu loops jika hal ini tidak digunakan dengan benar.

Keuntungan iterasi dibandingkan rekursif adalah performance yang lebih baik. Hal tersebut lebih cepat untuk rekursif sejak terbentuknya sebuah parameter pada sebuah method yang menyebabkan adanya suatu CPU time. Bagaimanapun juga, rekursif mendorong pelatihan perancangan software yang lebih baik, sebab teknik ini biasanya dihasilkan dalam kode yang singkat yang lebih mudah untuk dimengerti dan juga mempromosikan reusability pada suatu solusi yang sebelumnya telah diterapkan.

Memilih antara iterasi dan rekursif merupakan permasalahan dari menjaga keseimbangan antara baiknya sebuah performance dan baiknya perancangan software.

3.2.3 Faktorial: Contoh

Listing program berikut ini menunjukkan bagaimana menghitung faktorial menggunakan teknik iterasi.

```
class FactorialIter {
   static int factorial(int n) {
      int result = 1;
      for (int i = n; i > 1; i--) {
         result *= i;
      }
```

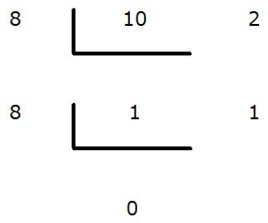
```
            return result;
         }
         public static void main(String args[]) {
            int n = Integer.parseInt(args[0]);
            System.out.println(factorial(n));
         }
      }
```

Dibawah ini merupakan listing program yang sama tetapi menggunakan rekursif.

```
      class FactorialRecur {
         static int factorial(int n) {
            if (n == 1) {            /* The base case */
               return 1;
            }
            /* Recursive definition; Self-invocation */
            return factorial(n-1)*n;
         }
         public static void main(String args[]) {
            int n = Integer.parseInt(args[0]);
            System.out.println(factorial(n));
         }
}
```

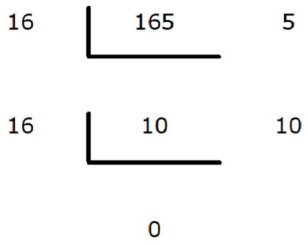
3.2.4 Print n in any Base: Contoh yang lain

Sekarang, mempertimbangkan dari masalah dalam pencetakkan suatu angka desimal yang nilai basenya telah ditetapkan oleh pengguna. Ingat bahwa solusi dalam hal ini untuk menggunakan repetitive division dan untuk menulis sisa perhitungannya. Proses akan berakhir ketika sisa hasil pembagian kurang dari base yang ditetapkan. Dapat diasumsikan jika nilai input desimal adalah 10 dan kita akan mengkonversinya menjadi base 8. Inilah solusinya dengan perhitungan menggunakan pensil dan kertas.



Dari solusi diatas, 10 adalah sama dengan 12 base 8.

Contoh berikutnya. Nilai input desimalnya adalah 165 dan akan dikonversi ke base 16.



165 adalah sama dengan A5 base 16. Catatan: A=10.

Berikut ini merupakan solusi iterative untuk masalah diatas.

```
class DecToOthers {
   public static void main(String args[]) {
      int num = Integer.parseInt(args[0]);
      int base = Integer.parseInt(args[1]);
      printBase(num, base);
   }
   static void printBase(int num, int base) {
      int rem = 1;
      String digits = "0123456789abcdef";
      String result = "";
      /* langkah interasi */
      while (num!=0) {
         rem = num%base;
         num = num/base;
         result = result.concat(digits.charAt(rem)+"");
      }
      /* mencetak reverse dari result */
      for(int i = result.length()-1; i >= 0; i--) {
         System.out.print(result.charAt(i));
      }
   }
}
```

Berikut ini merupakan rekursif untuk masalah yang sama dengan solusi sebelumnya.

```
class DecToOthersRecur {
   static void printBase(int num, int base) {
      String digits = "0123456789abcdef";
      /* Langkah Rekursif*/
      if (num >= base) {
         printBase(num/base, base);
      }
      /* Base case: num < base */
      System.out.print(digits.charAt(num%base));
   }
   public static void main(String args[]) {
      int num = Integer.parseInt(args[0]);
      int base = Integer.parseInt(args[1]);
      printBase(num, base);
   }
}
```

3.3 Tipe Data Abstract

3.3.1 Apa yang Dimaksud dengan Tipe Data Abstract?

Abstract Data Type (ADT) adalah kumpulan dari elemen-elemen data yang disajikan dengan satu set operasi yang digambarkan pada elemen-elemen data tersebut. Stacks, queues dan pohon biner adalah tiga contoh dari ADT. Dalam bab ini, Anda akan mempelajari tentang stacks dan queues.

3.3.2 Stacks

Stack adalah satu set atau urutan elemen data dimana manipulasi data dari elemen-elemen hanya diperbolehkan pada tumpukan teratas dari stack. Hal ini merupakan perintah pengumpulan data secara linier yang disebut “last in, first out” (LIFO). Stacks berguna untuk bermacam-macam aplikasi seperti pattern recognition dan pengkonversian antar notasi infix, postfix dan prefix.

Dua operasi yang dihubungkan dengan stacks adalah operasi push dan pop. Push berarti memasukkan data ke dalam stacks yang paling atas dimana pop sebagai penunjuk/pointer untuk memindahkan elemen ke atas stacks. Untuk memahami bagaimana cara kerja stacks, pikirkan bagaimana Anda dapat menambah atau memindakan sebuah data dari tumpukan data. Pikiran Anda akan memberitahu Anda untuk menambah atau memindahkan data hanya pada stack yang paling atas karena jika menggunakan cara lain, dapat menyebabkan tumpukan stack akan terjatuh.

Dibawah ini merupakan ilustrasi bagaimana tampilan dari stacks.

n-1		
…		
6		
5	Jayz	***top***
4	KC	
3	Jojo	
2	Toto	
1	Kyla	
0	DMX	***bottom***

Tabel 1.2.2: Ilustrasi Stack

Stack akan berarti penuh jika jangkauan sel teratas disimbolkan dengan n-1. Jika nilai teratas / top sama dengan -1, stack berarti kosong.

3.3.3 Queues

Queues adalah contoh lain dari ADT. Hal ini merupakan perintah pengumpulan data yang disebut “first-in, first-out”. Aplikasi ini meliputi tugas penjadwalan dalam sistem operasi, topological sorting dan graph traversal.

Enqueue dan dequeue merupakan operasi yang berhubungan dengan queues. Enqueue menunjuk pada memasukkan data pada akhir queue sedangkan dequeue berarti memindahkan elemen dari queue tersebut. Untuk mengingat bagaimana queue bekerja, ingatlah arti khusus dari queue yaitu baris. Berikut ini bagaimana cara kerja queue. Siapa yang akan mendapatkan kesempatan pertama untuk bertemu bintang idolanya dari mereka yang sedang menunggu dalam sebuah barisan? Seharusnya orang pertama yang berada pada barisan tersebut. Orang ini mendapat kesempatan pertama untuk meninggalkan barisan. Hubungkan hal tersebut dengan bagaimana queue bekerja.

Berikut ini merupakan ilustrasi dari bagaimana tampilan dari queue.

0	1	2	3	4	5	6	7	8	9	...	n-1
		Eve	Jayz	KC	Jojo	Toto	Kyla	DMX			
		front						***end***	☐	Insert	
									☐	Delete	

Tabel 1.2.3: Ilustrasi Queue

Queue akan kosong jika nilai end kurang dari front. Sementara itu, akan penuh jika end sama dengan n-1.

3.3.4 Sequential and Linked Representation

ADTs biasanya dapat diwakilkan menggunakan sequential dan linked representation. Hal ini memudahkan untuk membuat sequential representation dengan menggunakan array. Bagaimanapun juga, masalah dengan menggunakan array adalah pembatasan size, yang membuatnya tidak fleksibel. Dengan menggunakan array, sering terjadi kekurangan atau kelebihan space memori. Mempertimbangkan hal tersebut, Anda harus membuat sebuah array dan mendeklarasikannya agar mampu menyimpan 50 elemen. Jika user hanya memasukkan 5 elemen, maka 45 space pada memori akan sia-sia. Disisi lain, jika user ingin memasukkan 51 elemen, space yang telah disediakan didalam array tidak akan cukup.

Dibandingkan dengan sequential representation, linked representation lebih sedikit rumit tetapi lebih fleksibel. Linked representation menyesuaikan memori yang dibutuhkan oleh user. Penjelasan lebih lanjut pada linked representation akan didiskusikan pada bab berikutnya.

3.3.5 Sequential Representation dari Integer Stack

```
class SeqStack {
   int top = -1;     /* pada permulaan, stack kosong*/
   int memSpace[];   /* penyimpanan untuk integer */
   int limit;        /* ukuran dari memSpace */
```

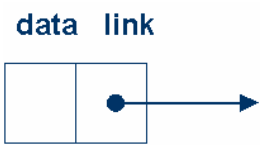
```
    SeqStack() {
       memSpace = new int[10];
       limit = 10;
    }
    SeqStack(int size) {
       memSpace = new int[size];
       limit = size;
    }
    boolean push(int value) {
       top++;
       /* memeriksa apakah stack penuh */
       if (top < limit) {
          memSpace[top] = value;
       } else {
          top--;
          return false;
       }
       return true;
    }
    int pop() {
       int temp = -1;
       /* memeriksa apakah stack kosong */
       if (top >= 0) {
          temp = memSpace[top];
          top--;
       } else {
          return -1;
       }
       return temp;
    }
    public static void main(String args[]) {
       SeqStack myStack = new SeqStack(3);
       myStack.push(1);
       myStack.push(2);
       myStack.push(3);
       myStack.push(4);
       System.out.println(myStack.pop());
       System.out.println(myStack.pop());
       System.out.println(myStack.pop());
       System.out.println(myStack.pop());
    }
}
```

3.3.6 Linked Lists

Sebelum mengimplementasikan linked representation dari stacks, pertama mari kita pelajari bagaimana membuat linked representation. Dalam hal ini, kita akan menggunakan linked list.

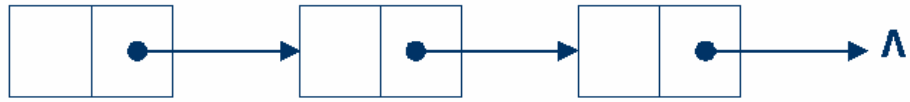
Linked list merupakan struktur dinamis yang berlawanan dengan array, dimana merupakan struktur statis. Hal ini berarti linked list dapat tumbuh dan berkurang dalam ukuran yang bergantung pada kebutuhan user. Linked list digambarkan sebagai kumpulan dari nodes, Yang masing-masing berisi data dan link atau pointer ke node berikutnya di dalam list.

Gambar dibawah ini menunjukkan tampilan dari node.



Gambar 2.6a: Sebuah node

Berikut ini merupakan contoh dari non-empty linked list dengan 3 node.



Gambar 3.6b: Non-empty linked list dengan tiga node

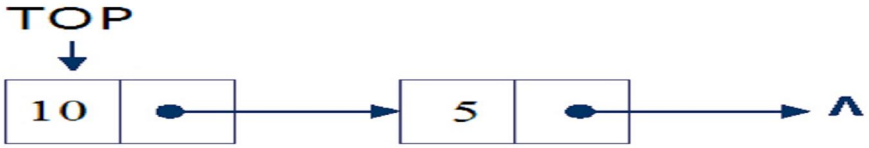
Berikut ini bagaimana class node diimplementasikan. Class ini dapat digunakan untuk membuat linked list.

```
class Node {
   int data;        /* integer data diisikan dalam node */
   Node nextNode;   /* node selanjutnya dalam list */
}

class TestNode {
   public static void main(String args[]) {
      Node emptyList = null;    /* membuat list kosong */
      /* head points untuk node pertama dalam list */
      Node head = new Node();
      /* inisialisasi node pertama dalam list */
      head.data = 5;
      head.nextNode = new Node();
      head.nextNode.data = 10;
      /* null menandai akhir dari list */
      head.nextNode.nextNode = null;
      /* mencetak elemen list */
      Node currNode = head;
      while (currNode != null) {
         System.out.println(currNode.data);
         currNode = currNode.nextNode;
      }
   }
}
```

3.3.7 Linked Representation dari Integer Stack

Sekarang Anda telah mempelajari tentang linked list. Maka Anda telah siap untuk menerapkan apa yang telah Anda pelajari untuk implementasi linked representation dari stack.

```
class DynamicIntStack{
   private IntStackNode top;  /* head atau puncak dari stack */
   class IntStackNode {       /* class node */
      int data;
      IntStackNode next;
      IntStackNode(int n) {
         data = n;
         next = null;
      }
   }


   void push(int n){
      /* no need to check for overflow */
      IntStackNode node = new IntStackNode(n);
      node.next = top;
      top = node;
   }
   int pop() {
      if (isEmpty()) {
         return -1;
         /* may throw a user-defined exception */
      } else {
         int n = top.data;
         top = top.next;
         return n;
      }
   }
   boolean isEmpty(){
      return top == null;
   }
   public static void main(String args[]) {
      DynamicIntStack myStack = new DynamicIntStack();
      myStack.push(5);
      myStack.push(10);
      /* mencetak elemen dari stack */
      IntStackNode currNode = myStack.top;
      while (currNode!=null) {
         System.out.println(currNode.data);
         currNode = currNode.next;
      }
      System.out.println(myStack.pop());
      System.out.println(myStack.pop());
   }
}
```



Gambar 1.2.7: Implementasi linked dari stack

3.3.8 Java Collections

Saat ini Anda telah diperkenalkan kepada dasar tipe data abstract. Pada intinya, Anda telah mempelajari tentang dasar dari linked lists, stacks dan queue. Berita baik bahwa tipe data abstract telah siap untuk diimplementasikan dan dimasukkan dalam Java. Class *Stack* dan *LinkedList* diperbolehkan digunakan tanpa perlu pengertian yang lengkap dari konsep ini. Bagaimanapun juga, sebagai ilmuwan komputer, sangat penting untuk mengerti konsep dari tipe data abstract. Oleh karena itu, penjelasan terperinci masih disampaikan dalam bagian yang terdahulu. Dengan peluncuran dari J2SE 5.0, queue interface telah tersedia. Untuk detail pada class dan interface ini, dapat dilihat pada dokumentasi Java API.

Kepada kita, Java telah menyajikan classes dan interfaces *Collection* yang lain, dimana semuanya dapat ditemukan di *java.util* package. Contoh dari classes *Collection* termasuk *LinkedList, ArrayList, HashSet* dan *TreeSet. Class* tersebut benar-benar implementasi dari collection interfaces yang berbeda. Induk hirarki dari collection interfaces adalah interfaces *Collection* itu sendiri. Sebuah collection hanyalah sebuah grup dari object yang diketahui sebagai elemennya sendiri. Collection memperbolehkan penggandaan/salinan dan tidak membutuhkan pemesanan elemen secara spesifik.

SDK tidak menyediakan implementasi built-in yang lain dari interface ini tetapi mengarahkan subinterfaces, interfaces *Set* dan interfaces *List* diperbolehkan. Sekarang, apa perbedaan dari kedua interface tersebut. Set merupakan collection yang tidak dipesan dan tidak ada penggandaan di dalamnya. Sementara itu, list merupakan collection yang dipesan dari elemen-elemen dimana juga diperbolehkannya penggandaan. *HashSet, LinkedHashSet* dan *TreeSet* suatu implementasi class dari interfaces *Set*. *ArrayList, LinkedList* dan *Vector* suatu implementasi class dari *List* interfaces.

<root interface> *Collection*					
<interface> *Set*			*<interface>* *List*		
<implementing classes>			*<implementing classes>*		
HashSet	LinkedHashSet	TreeSet	ArrayList	LinkedList	Vector

Tabel 1.2.8a: Java collections

Berikut ini adalah daftar dari beberapa methods *Collections* yang disediakan dalam Collection API dari Java 2 Platform SE v1.4.1. Dalam Java 2 Platform SE v.1.5.0, methods ini telah dimodifikasi untuk menampung generic types. Karena generic types masih belum selesai dibahas, sebaiknya mempertimbangkan method ini terlebih dahulu.

Disarankan bahwa Anda mengacu pada *Collection* methods yang terbaru dimana Anda lebih mudah mengerti generic types, yang akan didiskusikan pada chapter berikutnya.

Collection Methods
`public boolean add(Object o)`
Memasukkan *Object o* ke dalam collection ini. Mengembalikan nilai *true* jika *o* telah sukses ditambahkan ke dalam collection.
`public void clear()`
Menghapus semua elemen dari collection ini.
`public boolean remove(Object o)`
Menghapus single instance dari *Object o* pada collection ini, jika hal tersebut telah diinputkan. Mengembalikan nilai *true* jika *o* telah ditemukan dan dihapus dari collection.
`public boolean contains(Object o)`
Mengembalikan nilai true jika collection ini berisi *Object o*.
`public boolean isEmpty()`
Mengembalikan nilai true jika collection ini tidak berisi object atau elemen apapun.
`public int size()`
Mengembalikan jumlah dari elemen pada collection ini.
`public Iterator iterator()`
Mengembalikan sebuah iterator yang menunjukkan kita pada isi collection ini.
`public boolean equals(Object o)`
Mengembalikan nilai true jika *Object o* sama dengan yang ada pada collection ini.
`public int hashCode()`
Mengembalikan nilai hash code (yaitu ID) untuk collection ini. Objects atau collections yang sama memiliki nilai hash code atau ID yang sama.

Tabel 1.2.8b: Methods dari class Collection

Anda diharapkan mengacu pada dokumentasi API untuk mengetahui daftar lengkap dari methods dalam interface *Collection, List* dan *Set*.
Saat ini kita akan melihat beberapa classes collection. Harap mengacu pada API untuk daftar dari methods yang dimasukkan ke dalam class ini.

Pada bagian sebelumnya, Anda telah melihat bagaimana mengimplementasikan linked list dengan cara Anda sendiri. Java SDK juga telah menyediakan built-implementation dari linked list untuk kita. Class *LinkedList* berisi methods yang memperbolehkan linked list digunakan seperti stacks, queue atau ADT yang lain. Listing program berikut ini menunjukan bagaimana menggunakan class *LinkedList*.

```
import java.util.*;

class LinkedListDemo {
   public static void main(String args[]) {
      LinkedList list = new LinkedList();
      list.add(new Integer(1));
      list.add(new Integer(2));
      list.add(new Integer(3));
      list.add(new Integer(1));
      System.out.println(list + ", size = " + list.size());
      list.addFirst(new Integer(0));
      list.addLast(new Integer(4));
      System.out.println(list);
      System.out.println(list.getFirst() + ", " +
                                          list.getLast());
      System.out.println(list.get(2) + ", " + list.get(3));
      list.removeFirst();
      list.removeLast();
      System.out.println(list);
      list.remove(new Integer(1));
      System.out.println(list);
      list.remove(3);
      System.out.println(list);
      list.set(2, "one");
      System.out.println(list);
   }
}
```

ArrayList merupakan versi fleksibel dari array biasa. Yang mengimplementasikan *List* interface. Telitilah kode berikut ini.

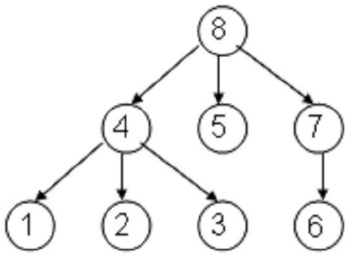
```
import java.util.*;

class ArrayListDemo {
   public static void main(String args[]) {
      ArrayList al = new ArrayList(2);
      System.out.println(al + ", size = " + al.size());
      al.add("R");
      al.add("U");
      al.add("O");
      System.out.println(al + ", size = " + al.size());
      al.remove("U");
      System.out.println(al + ", size = " + al.size());
      ListIterator li = al.listIterator();
      while (li.hasNext())
         System.out.println(li.next());
      Object a[] = al.toArray();
      for (int i=0; i<a.length; i++)
         System.out.println(a[i]);
   }
}
```

HashSet merupakan sebuah implementasi dari interface *Set* yang mempergunakan hash table. Penggunaan suatu hash table lebih mudah dan cepat untuk melihat lebih detail elemen-elemen yang ada. Tabel tersebut menggunakan suatu rumusan untuk menentukan dimana suatu objek disimpan. Teliti program ini, yang menggunakan class *HashSet*.

```
import java.util.*;

class HashSetDemo {
   public static void main(String args[]) {
      HashSet hs = new HashSet(5, 0.5f);
      System.out.println(hs.add("one"));
      System.out.println(hs.add("two"));
      System.out.println(hs.add("one"));
      System.out.println(hs.add("three"));
      System.out.println(hs.add("four"));
      System.out.println(hs.add("five"));
      System.out.println(hs);
   }
}
```

TreeSet merupakan sebuah implementasi dari interface *Set* yang menggunakan tree. Class ini memastikan bahwa yang disortir akan diurutkan secara ascending. Perhatikan, bagaimana class *TreeSet* telah digunakan dalam listing program berikut ini.

```
import java.util.*;

class TreeSetDemo {
   public static void main(String args[]) {
      TreeSet ts = new TreeSet();
      ts.add("one");
      ts.add("two");
      ts.add("three");
      ts.add("four");
      System.out.println(ts);
   }
}
```



Gambar 1.2.8: Contoh TreeSet

3.4 Latihan

3.4.1 Faktor Persekutuan Terbesar

Faktor persekutuan terbesar (FPB) dari dua angka adalah angka yang terbesar selalu dibagi oleh angka yang satunya, kemudian modulus atau sisa pembagian membagi angka kedua dan seterusnya hingga sisa pembagian dari kedua angka tersebut sama dengan nol. Menggunakan metode Euclid, buatlah dua kode untuk penghitungan dua angka. Gunakan iterasi untuk kode program yang pertama dan rekursif untuk kode program berikutnya.
Catatan pada algoritma Euclid :

1. Sebagai masukkan integers x dan y.
2. Ulangi step dibawah ini while y != 0
    a. y = x % y;
    b. x = Nilai lama y;
3. Return x.

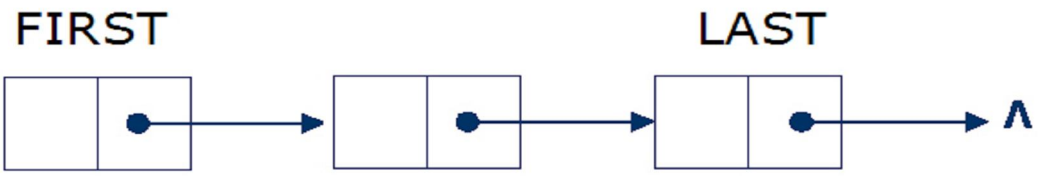
Contoh, x = 14 dan y = 6.
y = x % y = 14 % 6 = 2
x = 6
y = x % y = 6 % 2 = 0
x = 2 (FPB)

3.4.2 Sequential Representation dari Integer Queue

Dengan menggunakan array, implementasikan sebuah integer queue seperti contoh pada sequential stack.

3.4.3 Linked Representation dari Integer Queue

Dengan menggunakan ide dari linked list, implementasikan sebuah integer queue dinamis seperti integer stack dinamis yang diperkenalkan seperti contoh berikut.



3.4.4 Address Book

Dengan menggunakan Collection Java, buatlah sebuah program yang memperbolehkan user untuk insert, delete dan view address. Setiap address berisi nama, alamat dan nomor telepon dari orang yang mengisinya. Pengisian data dimasukkan dengan cara queue tetapi penghapusan dilakukan dengan cara stack.
Dalam contoh ini, kita akan menggunakan text editor untuk mengedit program Java. Juga membuka terminal window untuk meng-compile dan mengeksekusi program Java Anda.