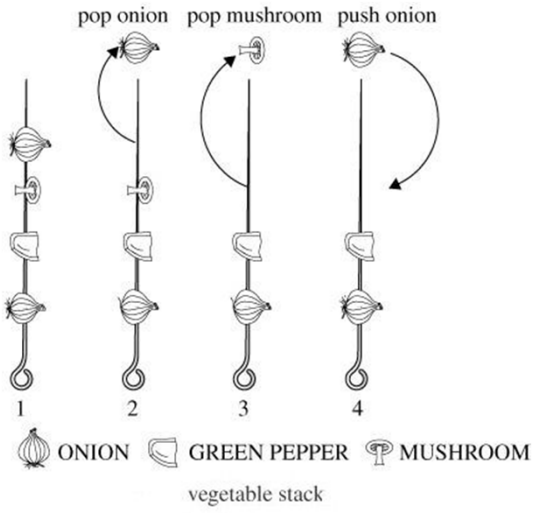
STACK (Tumpukan)

Arna F

Pengertian Stack

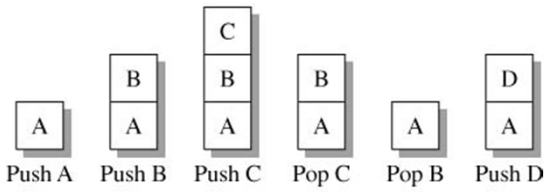
- Penyimpanan data/item dimana data/item yang diakses adalah paling akhir yang disebut top of stack.
- Item ditempatkan membentuk tumpukan
- Merupakan penyimpanan data dengan konsep LIFO (Last In First Out)

Contoh Stack



Operasi Stack

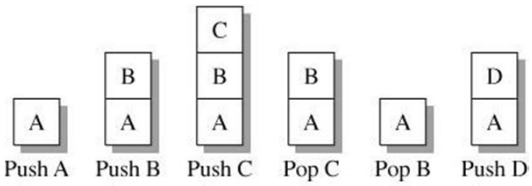
push(item) menambah item pada top of stack
pop() menghapus elemen dari top of stack
peek() mengakses nilai top of stack



Operasi Stack

- Item di hapus (pop) dari stack adalah eleman terakhir yang ditambahkan (push) ke stack atau LIFO (last-in-first-out)

```
push A
push B
push C
pop
pop
pop
push D
```



Stack Interface

- Stack interface mendefinisikan kumpulan operasi yang mengakses dan meng-update hanya satu data pada akhir list

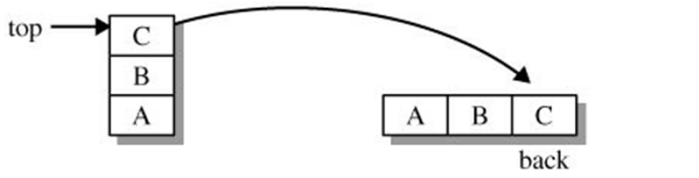
interface Stack<T>	**ds.util**
boolean	**isEmpty**() bernilai true jika stack tidak terdapat elemen dan false jika stack mempunyai sedikitnya 1 elemen
T	**peek**() bernilai elemen pada top of stack. Jika stack kosong, akan melempar (throw) exception yaitu EmptyStackException.

Stack Interface

interface Stack<T>	ds.util
T	**pop**() menghapus elemen pada top of stack dan menghasilkan nilai yang di-pop. Jika stack kosong, throw exception EmptyStackException.
void	**push**(T item) menambah item pada top of stack.
int	**size**() bernilai jumlah elemen pada stack

Class ALStack

- Interface mendefinisikan method yang terbatas. Class Stack dapat diimplementasikan menggunakan class List sebagai struktur penyimpan. Class ALStack menggunakan ArrayList dan composition.



A stack conceptually. A stack implemented as an Array List.

Class ALStack

```
public class ALStack<T> implements Stack<T>
{
   // storage structure
   private ArrayList<T> stackList = null;

   // create an empty stack by creating
   // an empty ArrayList
   public ALStack()
   {
      stackList = new ArrayList<T>();
   }

   . . .
}
```

Implementasi ALStack Method peek()

- Method has runtime efficiency O(1)

```
public T peek()
{
   // if the stack is empty, throw
   // EmptyStackException
   if (isEmpty())
      throw new EmptyStackException();

   // return the element at the back of the ArrayList
   return stackList.get(stackList.size()-1);
}
```

Implementasi ALStack Method push()

- Method has runtime efficiency O(1)

```
public void push(T item)
{
   // add item at the end of the ArrayList
   stackList.add(item);
}
```

Implementasi ALStack Method pop()

- Method has runtime efficiency O(1)

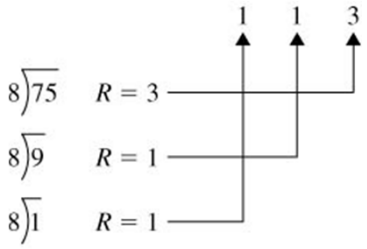
```
public T pop()
{
   // if the stack is empty, throw
   // EmptyStackException
   if (isEmpty())
      throw new EmptyStackException();

   // remove and return the last
   // element in the ArrayList
   return stackList.remove(stackList.size()-1);
}
```

Contoh Aplikasi: Multibase Numbers

- Multibase numbers menggunakan basis untuk melakukan konversi dari desimal ke basis tertentu
- Contoh n = 75 dengan basis 2, 8, 16

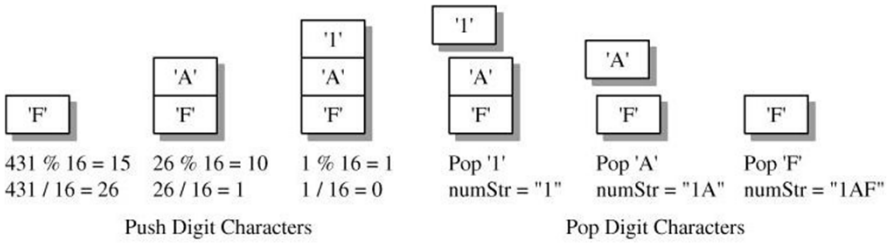
Contoh Aplikasi: Multibase Numbers

- Algoritma menggunakan pembagian berulang (n / basis) dan operator mod (n % basis) untuk membuat dan mengubah angka
- Contoh $75_{10} = 113_8$



Contoh Aplikasi: Multibase Numbers

- Contoh ilustrasi konversi integer n = 431 ke nilai basis-16 (hex) string "1AF".



baseString()

```
public static String baseString(int num, int b)
{
   // digitChar.charAt(digit) is the
   // character that represents the digit,
   // 0 <= digit <= 15
   String digitChar = "0123456789ABCDEF", numStr = "";

   // stack holds the base-b digits of num
   ALStack<Character> stk = new ALStack<Character>();

   // extract base b digits right
   // to left and push on stack
   do
   {
      // push right-most digit on the stack
      stk.push(digitChar.charAt(num % b));
      // remove right-most digit from num
      num /= b;
   // continue until all digits found
   } while (num != 0);
```

baseString() (lanjutan)

```
    while (!stk.isEmpty())   // flush the stack
    {
       // pop stack and add digit on top
       // of stack to numStr
       numStr += stk.pop().charValue();
    }
    return numStr;
}
```

Contoh Program

- Program menggunakan method baseString()
- Output dalam 4 nilai integer non negatif

Contoh Program (lanjutan)

```
import java.util.Scanner;
import ds.util.ALStack;

public class MultibaseTest
{
   public static void main(String[] args)
   {
      int num, b;    // decimal number and base
      int i;         // loop index

      // create scanner for keyboard input
      Scanner keyIn = new Scanner(System.in);

      for (i = 1; i <= 4; i++)
      {
         // prompt for number and base
         System.out.print("Enter a decimal number: ");
         num = keyIn.nextInt();
         System.out.print("Enter a base (2 to 16): ");
         b = keyIn.nextInt();
```

Contoh Program (lanjutan)

```
         System.out.println(" " + num + " base " + b +
               " is " + baseString(num, b));
      }
   }
   < listing of baseString() given in
     the program discussion >
}
```

Contoh Program (Hasil running)

```
Run:

Enter a decimal number: 27
Enter a base (2 to 16): 2
 27 base 2 is 11011
Enter a decimal number: 300
Enter a base (2 to 16): 16
 300 base 16 is 12C
Enter a decimal number: 75
Enter a base (2 to 16): 8
 75 base 8 is 113
Enter a decimal number: 10
Enter a base (2 to 16): 3
 10 base 3 is 101
```

Ekspresi Postfix

- Format Postfix (RPN) untuk ekspresi matematika, contoh

  a + b * c RPN: a b c * +

  Operator * lebih tinggi dari +.

  (a + b) * c RPN: a b + c *

  tanda kurung membentuk sub ekspresi a b +

  (a*b + c) / d + e RPN: a b * c + d/ e +

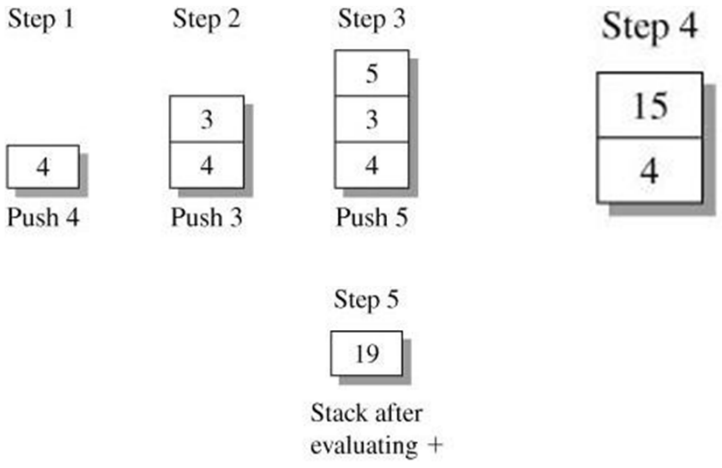
  sub ekspresi a b * c +. Pembagian lebih dahulu dari penambahan.

Evaluasi Postfix

- Untuk mengevaluasi ekspresi postfix, jalankan langkah2 berikut sampai akhir notasi.
  1. Jika operan, push ke stack.
  2. Jika operator, pop operan, masukkan operator ke stack.
  3. Sebagai kesimpulan, nilai ekspresi postfix ada di top of stack.

Evaluasi Postfix (lanjutan)

- Contoh: evaluasi "4 3 5 * +"

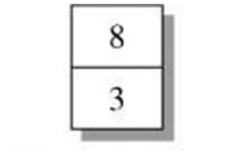


Evaluasi Postfix (lanjutan)

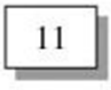
- Setiap langkah pada algoritma evaluasi postfix, status stack menuntun untuk mengidentifikasi apakan terjadi error dan penyebab error.

Evaluasi Postfix (lanjutan)

- Ekspresi 3 8 + * 9 operator * kehilangan operan ke 2. Identifikasi error tersebut jika membaca * dengan stack berisi hanya satu elemen.



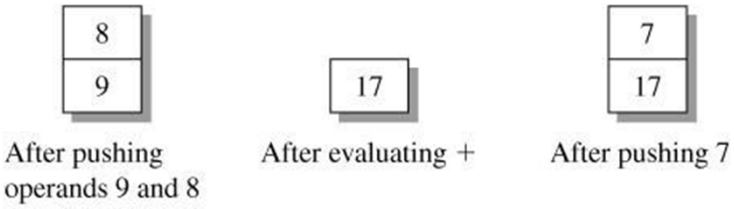
After pushing operands 3 and 8



After evaluating +

Evaluasi Postfix (lanjutan)

- Ekspresi mungkin berisi terlalu banyak operan. Identifikasi error tsb setelah memproses keseluruhan ekspresi. Konklusi dari proses, stack berisi lebih dari satu elemen.
- Contoh: 9 8 + 7



Class PostfixEval

- Class PostfixEval merupakan method untuk membaca ekspresi postfix dan menghasilkan nilai. Method kunci adalah evaluate() yang menggunakan method private compute(), getOperand(), dan isOperand().

PostfixEval
–postfixExpression: String
+PostfixEval() +compute(left: int, right: int, op: chair): int +evaluate(): int –getOperand(): int +getPostfixExp(): String +isOperator(ch: char): boolean +setPostfixExp(postfixExp: String): void

Contoh Program

```
import java.util.Scanner;

public class PostfixEvaluateTest
{
   public static void main(String[] args)
   {
      // object used to evaluate postfix expressions
      PostfixEval exp = new PostfixEval();
      // postfix expression input
      String rpnExp;
      // for reading an expression
      Scanner keyIn = new Scanner(System.in);

      System.out.print("Enter the postfix " +
            "expression: ");
      rpnExp = keyIn.nextLine();

      // assign the expression to exp
      exp.setPostfixExp(rpnExp);
```

Contoh Program (lanjutan)

```
      // call evaluate() in a try block
      // in case an error occurs
      try
      {
         System.out.println("The value of the " +
               "expression = " +
               exp.evaluate() + "\n");
      }
      // catch block outputs the error
      catch (ArithmeticException ae)
      {
         System.out.println(ae.getMessage() + "\n");
      }
   }
}
```

Contoh Program (run)

```
Run 1:
  (2 + 5)*3 - 8/3
  Enter the postfix expression: 2 5 + 3 * 8 3 / -
  The value of the expression = 19

Run 2:
  2^3 + 1
  Enter the postfix expression: 2 3 ^ 1 +
  The value of the expression = 9

Run 3:
  Enter the postfix expression: 1 9 * /
  PostfixEval: Too many operators

Run 4:
  Enter the postfix expression: 2 3 5 +
  PostfixEval: Too many operands
```

Implementasi PostfixEval

- Scanning ekspresi postfix dan menghitung hasil berada pada method evaluate() yang menggunakan stack integer, operandStack, untuk menyimpan operand.
  - Algoritma evaluate() menggunakan method private isOperand(), getOperand() dan compute().

Implementasi PostfixEval (lanjutan)

- Method boolean isOperand() dipanggil jika men-scanning karakter non-spasi untuk menentukan apakah karakter valid ('+','-','*','/','%','^').
- Jika valid, evaluate() memanggil getOperand() untuk mengambil operan pertama pada sisi kanan dan kemudian operan pada sisi kiri. Method throw ArithmeticException jika stack kosong.

Implementasi PostfixEval (lanjutan)

- Operator diproses menggunakan method compute() yang mengevaluasi operasi "left op right" dan push hasilnya ke stack.

Implementasi PostfixEval (lanjutan)

```
int compute(int left, int right, char op)
{
    int value = 0;

    // evaluate "left op right"
    switch(op)
    {
        case '+':   value = left + right;
                                    break;

        case '-':   value = left - right;
                                    break;
                . . .
        < throw ArithmeticException for
          divide by 0 or 0^0 >
    }

    return value;
}
```

Implementasi PostfixEval (lanjutan)

- Looping pada evaluate() men-scan setiap karekter dari ekspresi postfix dan berhenti jika semua karakter sudah diproses atau terdapat error. Setelah selesai lengkap, hasil akhir berada pada top of stack dan dikirim ke variabel expValue yang menjadi return value.

Implementasi PostfixEval (lanjutan)

```
int left, right, expValue;
char ch;
int i;
// process characters until the end of the string is reached
// or an error occurs
for (i=0; i < postfixExpression.length(); i++)
{
   // get the current character
   ch = postfixExpression.charAt(i);
   . . .
}
```

Implementasi PostfixEval (lanjutan)

- Scan menggunakan mothod static isDigit() dari class Character class untuk menentukan apakah karakter ch adalah angka. Method menghasilkan true jika ch >= '0' && ch <= '9'. Dalam hal ini, evaluate() push nilai Integer yang berhubungan dengan operan ke stack.

Implementasi PostfixEval (lanjutan)

```
// look for an operand, which is a single digit
// non-negative integer
if (Character.isDigit(ch))
    // value of operand goes on the stack as Integer object
    operandStack.push(ch - '0');
```

Implementasi PostfixEval (lanjutan)

- Jika karakter ch adalah operator, evaluate() menginisialisasi proses ekstraksi operan dari stack dan menggunakan compute() untuk melanjutkan kalkulasi dan menghasilkan return value. Push return value ke the stack.

Implementasi PostfixEval (lanjutan)

```
// look for an operator
else if (isOperator(ch))
{
    // pop the stack to obtain the right operand
    right = getOperand();
    // pop the stack to obtain the left operand
    left = getOperand();
    // evaluate "left op right" and push on stack
    operandStack.push(new Integer(compute(left, right, ch)));
}
```

Implementasi PostfixEval (lanjutan)

- Jika karakter ch bukan operan atau operator, evaluate() menggunakan method isWhitespace() dari class Character class untuk menentukan apakah ch adalah spasi (blank, baris baru atau tab). Jika bukan, evaluate() throw ArithmeticException; lainnya, perulangan melanjutkan karakter berikutnya.

Implementasi PostfixEval (lanjutan)

```
// any other character must be whitespace.
// whitespace includes blank, tab, and newline
else if (!Character.isWhitespace(ch))
   throw new ArithmeticException("PostfixEval: Improper char");
```

Implementasi PostfixEval (lanjutan)

- Jika ekspresi postfix berakhir tanpa error, nilai yang tepat akan diletakkan pada top of stack. Method evaluate(), pop nilai dari stack. Jika stack kosong, nilai berupa hasil final. Jika stack masih berisi element, evaluate() berakhir dengan too many operand dan throw ArithmeticException.

Implementasi PostfixEval (lanjutan)

```
// the expression value is on the top of the stack. pop it off
expValue = operandStack.pop();
// if data remains on the stack, there are too many operands
if (!operandStack.isEmpty())
    throw new ArithmeticException
        ("PostfixEval: Too many operands");
return expValue;
```

Evaluasi Ekspresi Infix

- Pada ekspresi infix, setiap operator binar berada diantara operan dan operator unary berada sebelum operan. Infix adalah format umum untuk menulis ekspresi dan format ekspresi untuk banyak bahasa pemrograman dan kalkulator. Evaluasi lebih sulit dari ekspresi postfix.

Evaluasi Ekspresi Infix (lanjutan)

- Algoritma harus mempunyai strategi untuk menangani sub ekspresi dan harus memelihara urutan dan asosiasi untuk operator. Sebagai contoh

  9 + (2 - 3) * 8

  terdapat sub ekspresi (2 - 3) yang dievaluasi lebih dahulu dan kemudian menggunakan hasilnya sebagai operato kiri untuk *. Operator * mengeksekusi sebelum operator + , karena level lebih tinggi.

Evaluasi Ekspresi Infix (lanjutan)

- Terdapat dua pendekatan untuk evaluasi
  - Mengubah ekspresi infix ke postfix dan kemudian memanggil evaluasi postfix untuk hasilnya
  - Pendekatan lain dengan menelusuri ekspresi infix dan menggunakan stack operator dan operan untuk menyimpannya. Algoritma memberi hasil langsung

Atribut Ekspresi Infix

- Ekspresi Infix terdiri dari operand, operator, dan tanda kurung yang menandakan sub ekspresi, yang dihitung terpisah
  - Urutan level yang dievaluasi adalah yang tertinggi terlebih dahulu
    - Diantara operator aritmatika, operator penambahan (+, -) mempunyai urutan terendah, dilanjutkan operator perkalian (*, /, %). Operator eksponensial (^) merupakan urutan tertinggi.

Atribut Ekspresi Infix (lanj.)

- Konsep asosiatif berpegang pada urutan eksekusi untuk operator pada level yang sama. Jika lebih dari satu operator mempunyai level yang sama, operator ter-kiri dieksekusi lebih dahulu (+, -, , /, %) dan operator ter-kanan mengeksekusi asosiatif terkanan pertama (^).

Atribut Ekspresi Infix (lanj)

```
7 * 2 * 4 / 5
Evaluasi:        ((7 * 2) * 4) /5 = (14 * 4) / 5 = 56 % 5 = 11
```

2 ^ 3 ^ 2 + 3
Evaluasi: (2 ^ (3 ^ 2)) + 3 = 2^9 + 3 = 512 + 3 = 515

Konversi Infix-ke-Postfix

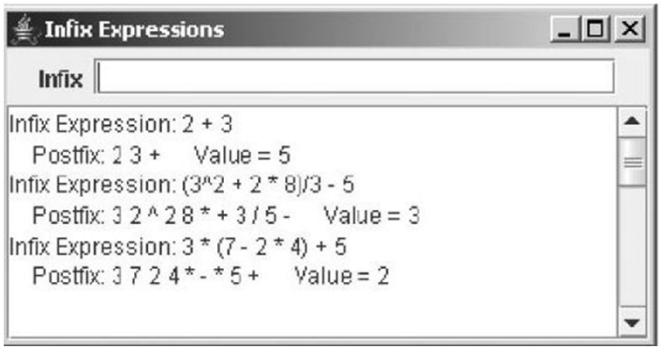
- Algoritma konversi infix-ke-postfix menggunakan ekspresi infix sebagai input string dan menghasilkan ekspresi postfix
  - Algoritma meng-copy operan ke output string dan menggunakan stack untuk menyiimpan dan memproses operator dan simbol kurung
    - Stack operator menyimpan urutan level dan asosiatif operator dan menangani sub ekspresi

Konversi Infix-ke-Postfix (lanj)

- Class InfixToPostfix mendefinisikan method untuk mengubah ekspresi infix yang terdiri dari operan berupa angka integer 0-9 dan operator +, -, *, /, %, dan ^ ke dalam bentuk postfix.

Konversi Infix-ke-Postfix (lanj)

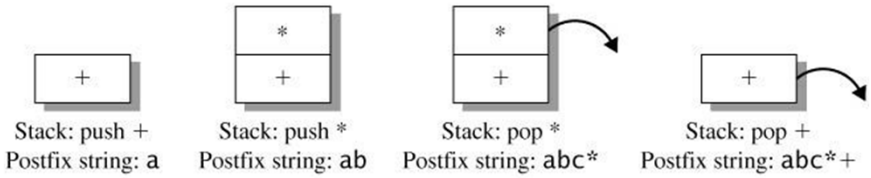
- Aplikasi InfixExpressions.java menginputkan ekspresi dan mengevaluasi hasil menggunakan class InfixToPostfix.



Konversi Infix-ke-Postfix (lanj)

- Tulis operan ke postfix. Tempatkan operator ke stack, menunggu operan sebelah kanan

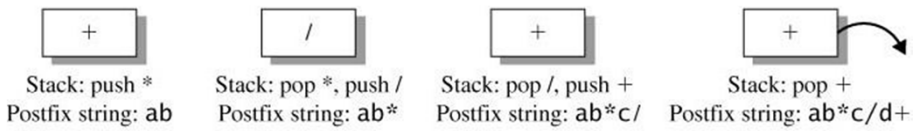
Contoh: Ekspresi infix a + b * c



Konversi Infix-ke-Postfix (lanj)

- Push operator ke stack hanya setelah menghapus semua operator yang lebih tinggi atau sama levelnya

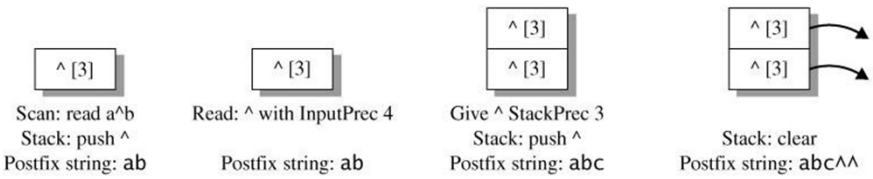
Contoh: Ekspresi infix a * b / c + d



Konversi Infix-ke-Postfix (lanj)

- Buat input dan level stack operator berbeda untuk menangani operator pada asosiasi kanan

Contoh: ekspresi infix a^b^c



Konversi Infix-ke-Postfix (lanj)

- Tanda '(' lebih besar dari stack untuk semua operator. Sehingga '(' dan operator pada sub ekspresi menempati stack sampai menemukan tanda ')'.

Konversi Infix-ke-Postfix (lanj)

Contoh: Ekspresi infix a * (b + c)