STRUKTUR DATA

IX. Implementasi ADT :
Stacks & Queues

Outline

- ADT Stacks
  - Operasi dasar
  - Contoh kegunaan
  - Implementasi
    - Array-based dan linked list-based
- ADT Queues
  - Operasi dasar
  - Contoh kegunaan
  - Implementasi
    - Array-based dan linked list-based

Tujuan

- Memahami cara kerja dan kegunaan stack & queue
- Dapat mengimplementasi stack dan queue

3

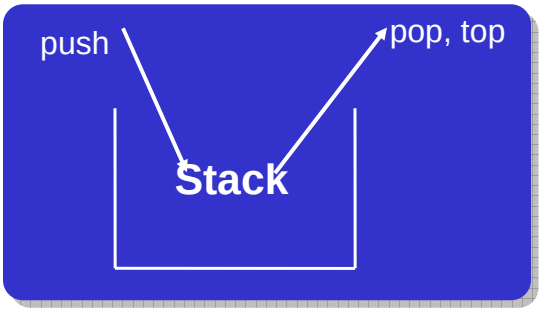
Struktur data linear

- Kumpulan elemen yang tersusun sebagai garis linear
- Stack: struktur data linear di mana penambahan/pengurangan elemen dilakukan di satu ujung saja.
- Queue: struktur data linear di mana penambahan komponen dilakukan di satu ujung, sementara pengurangan dilakukan di ujung lain (yang satu lagi).
- Kedua struktur tersebut merupakan ADT di mana *implementasi* pada tingkat lebih rendah dapat sebagai *list*, baik menggunakan struktur *sequential* (array) atau struktur berkait (linear linked-list).

4

Stack

- Semua akses dibatasi pada elemen yang paling akhir disisipkan
- Operasi-operasi dasar: **`push, pop, top`**.
- Operasi-operasi dasar memiliki waktu yang konstan



5

Contoh-contoh di dunia luar komputer

- Tumpukan kertas
- Tumpukan tagihan
- Tumpukan piring

- Waktu *O* (1) per operasi stack. (Dengan kata lain, waktu konstan per operasi, tidak bergantung berapa banyak item yang tersimpan dalam stack).

6

Aplikasi-aplikasi

- Stack dapat digunakan untuk memeriksa pasangan tanda kurung (*Balanced Symbol*), pasangan-pasangan seperti **{}**, **()**, **[]**.
- Misalnya: { [( )] } boleh, tapi { ([ )] } tidak boleh (tidak dapat dilakukan dengan penghitungan simbol secara sederhana).
- Sebuah kurung tutup harus merupakan pasangan kurung buka *yang paling terakhir ditemukan*. Jadi, stack dapat membantu!

7

Balanced Symbol Algorithm

- Buat stack baru yang kosong.
- Secara berulang baca token-token; jika token adalah:
  - Kurung buka, **push** token ke dalam stack
  - Kurung tutup, maka
    - **If** stack kosong, **then** laporkan **error**;
    - **else** **pop** stack, dan periksa apakah simbol yang di-pop merupakan pasangannya (jika tidak laporkan **error**)
- Di akhir file, jika stack tidak kosong, laporkan **error**.

8

Contoh

- Input: **{()}**
  - Push '**{**'
  - Push '**(**'; lalu stack berisikan '**{**', '**(**'
  - Pop; popped item adalah '**(**' yang adalah pasangan dari '**)**'. Stack sekarang berisikan '**{**'.
  - Pop; popped item adalah '**{**' yang adalah pasangan dari '**}**'.
  - End of file; stack kosong, jadi input benar.

9

Performance

- Running time adalah *O* (*N*), yang mana *N* adalah jumlah data (jumlah token).
- Algoritma memproses input secara sikuensial, tidak perlu backtrack (mundur).

10

Call stack

- Kasus *balanced symbol* serupa dengan *method call* dan *method return*, karena saat terjadi suatu method **`return`**, ia kembali ke method aktif yang sebelumnya.
- Hal ini ditangani dengan *call stack*.
- **Ide dasar**: ketika suatu *method call* terjadi, simpan *current state* dalam stack. Saat *return*, kembalikan state dengan melakukan *pop* stack.

11

Aplikasi Lainnya

- Mengubah fungsi rekursif menjadi non-rekursif dapat dilakukan dengan stack.
  - Lihat diskusi di forum rekursif mengenai maze runner, coba buat versi non-rekursif dengan bantuan stack.
- *Operator precedence parsing* ()
- Pembalikan urutan (*reversing*) dapat dengan mudah dilakukan dengan bantuan stack

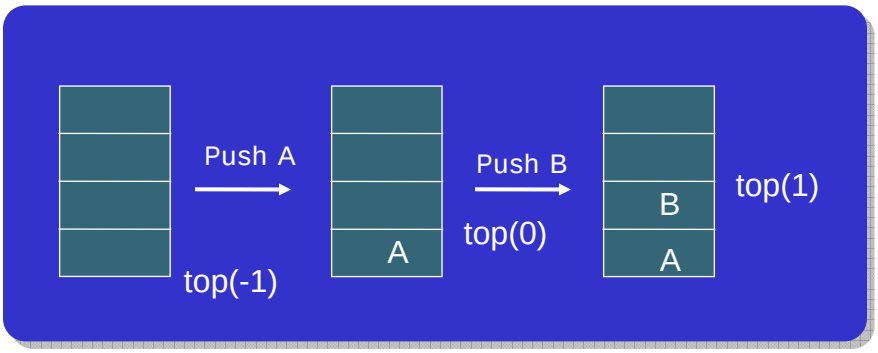
12

Implementasi Array

- Stack dapat diimplementasi dengan suatu array dan suatu integer **top** yang mencatat indeks dalam array dari top of the stack.
- Untuk stack kosong maka **top** berharga -1.
- Saat terjadi push, lakukan dengan increment counter **top**, dan tulis ke dalam posisi **top** tsb dalam array.
- Saat terjadi pop, lakukan dengan decrement counter **top**.

13

Ilustrasi



14

Pertanyaan:

Apa yang terjadi bila nilai top = ukuran array?

15

Array Doubling

- Jika stack full (karena semua posisi dalam array sudah terisi), kita dapat memperbesar array, menggunakan *array doubling*.
- Kita mengalokasi sebuah array baru dengan ukuran dua kali lipat semula, dan menyalin isi array yang lama ke yang baru:

```
Benda[] oldArray = array;
array = new Benda[oldArray.length * 2];
for (int j = 0; j < oldArray.length; j++)
    array[j] = oldArray[j];
```

16

Pertanyaan:

Dengan adanya *array doubling* apakah kompleksitas running time dari operasi push masih O(1)?

```
Benda[] oldArray = array;
array = new Benda[oldArray.length * 2];
for (int j = 0; j < oldArray.length; j++)
    array[j] = oldArray[j];
```


Running Time

- Tanpa adanya array doubling, setiap operasi memiliki waktu konstan, dan tidak bergantung pada jumlah item di dalam stack.
- Dengan adanya array doubling, satu operasi push dapat (namun jarang) menjadi $O(N)$. Namun, pada dasarnya adalah $O(1)$ karena setiap array doubling yang memerlukan N assignments didahului oleh $N/2$ kali push yang non-doubling.

Stack Implementation: Array

```
public 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;
    }
```

19

Stack Implementation: Array

```
    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;
    }
```

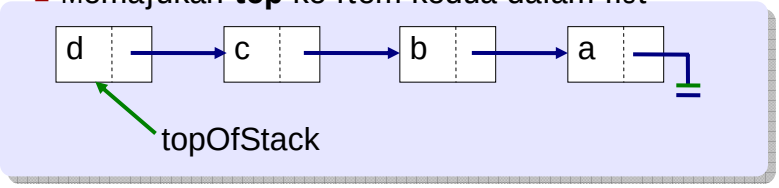
20

Stack Implementation: Array

```
public class testSeqStack {

    public static void main(String args[]) {
        SeqStack myStack = new SeqStack(4);

        myStack.push(10);
        myStack.push(30);
        myStack.push(75);
        //myStack.push(100);

       for (int j=0; j<=myStack.top; j++){
            System.out.print(myStack.memSpace[j] + " ");
       }

    System.out.println("");

   System.out.println(myStack.pop());
   System.out.println(myStack.pop());
   System.out.println(myStack.pop());

   System.out.println("");
      for (int j=0; j<=myStack.top; j++){
            System.out.print(myStack.memSpace[j] + " ");
       }

    }
}
```


Implementasi Linked-List

- Item pertama dalam list: *top of stack* (empty = **`null`**)
- **`push(Benda x)`**:
  - Create sebuah node baru
  - Sisipkan sebagai elemen pertama dalam list
- **`pop()`**:
  - Memajukan **top** ke item kedua dalam list



Stack Implementation: Linked List

```
public class LinkedStack {
    private IntStackNode top; /* head atau puncak dari stack */

    class IntStackNode {      /* class node */
        int data;
        IntStackNode next;

        IntStackNode(int n) {
            data = n;
            next = null;
        }
    }

    boolean isEmpty(){
        return top == null;
    }
```


Stack Implementation: Linked List

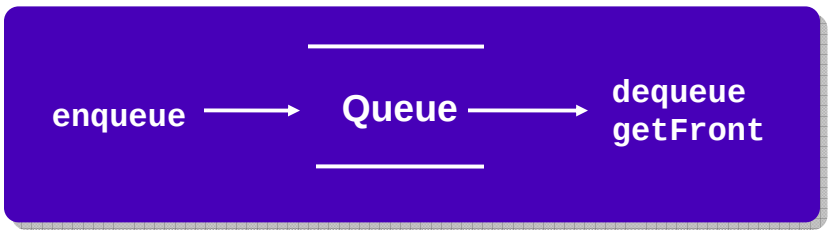
```
    void push(int n){
            /* tanpa pengecekan apakah penuh atau tidak */
        IntStackNode node = new IntStackNode(n);
        node.next = top;
        top = node;
    }

    int pop() {
        if (isEmpty()) {
            return -1;
        } else {
            int n = top.data;
            top = top.next;
            return n;
        }
    }
```

Stack Implementation: Linked List

```
    public static void main(String args[]) {
        LinkedStack myStack = new LinkedStack();

        myStack.push(5);
        myStack.push(10);

        /* mencetak elemen dari stack */
        IntStackNode currentNode = myStack.top;

        while (currentNode!=null) {
            System.out.print(currentNode.data + " ");
            currentNode = currentNode.next;
        }
        System.out.println(myStack.pop());
        System.out.println(myStack.pop());
    }
}
```


Queue

- Setiap akses dibatasi ke elemen yang paling terdahulu disisipkan
- Operasi-operasi dasar: **enqueue, dequeue, `getFront`**.
- Operasi-operasi dengan waktu konstan. Waktu operasi yang $O(1)$ karena mirip dengan stack.

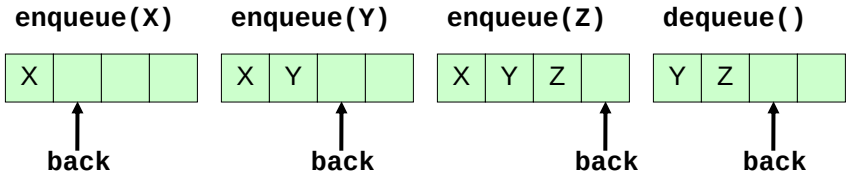


Contoh:

- Antrian printer
- Antrian tiket bioskop

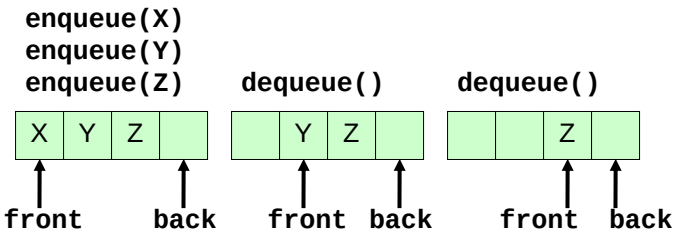
Implementasi dengan Array – cara naive

- Simpan item-item dalam suatu array dengan item terdepan pada index **nol** dan item terbelakang pada index **back**.
- Enqueue mudah & cepat: increment **back**.
- Dequeue tidak efisien: setiap elemen harus digeserkan ke depan. Akibatnya: waktu menjadi *O*(*N*).



Ide yang lebih baik:

- Menggunakan **`front`** untuk mencatat index terdepan.
- Dequeue dilakukan dengan increment **`front`**.
- Waktu Dequeue tetap O(1).



29

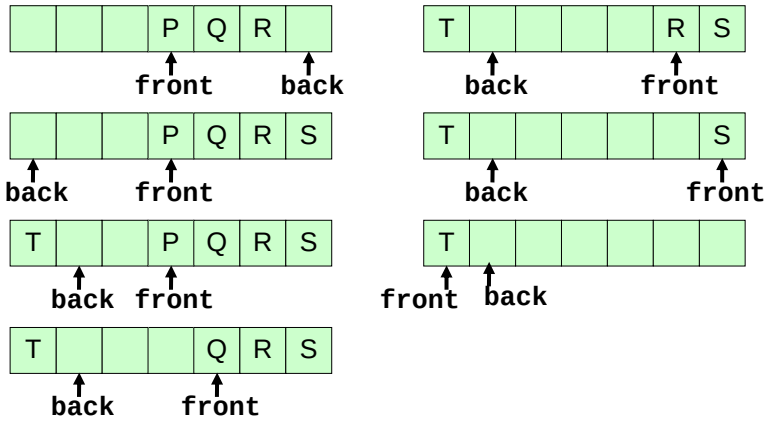
Queue penuh?

- Apa yang terjadi bila index **`back = Array.length-1`**?
  - Queue penuh?
  - Perlukah dilakukan array doubling?
- Apa yang terjadi bila queue kemudian di enqueue, hingga index **`first = Array.length-1`**?
  - Apakah queue penuh?
  - Perlukan dilakukan array doubling?

30

Implementasi Array Circular

- Solusi: gunakan *wraparound* untuk menggunakan kembali sel-sel di awal array yang sudah kosong akibat dequeue. Jadi setelah *increment*, jika index **`front`** atau **`back`** keluar dari array maka ia kembali ke 0.



Latihan: Implementasi Array Circular

- Bagaimana implementasi dari:
  - `enqueue(); // menambahkan elemen pada queue`
  - `dequeue(); // mengambil dan menghapus elemen`
  - `isEmpty(); // apakah queue kosong?`
  - `isFull(); // apakah queue sudah full?`
  - `size(); //memberikan jumlah elemen dalam queue`
  - `arrayDoubling(); // memperbesar kapasitas array`
- ???

Rangkuman

- Kedua versi, baik array maupun linked-list berjalan dengan O(1)
- Linked-list memiliki overhead akibat diperlukannya reference `next` pada setiap node
- Khusus untuk Queue, implementasi array lebih sulit dilakukan (secara *circular*)
- Memperbesar kapasitas dalam implementasi array (*arrayDoubling*) memerlukan space sekurangnya 3x jumlah item data!

33