IKI10400 • Struktur Data & Algoritma:

Linked List, Stack, and Queue

Fakultas Ilmu Komputer • Universitas Indonesia

Slide acknowledgments:
Suryana Setiawan, Ade Azurat, Denny, Ruli Manurung

LINKED LIST

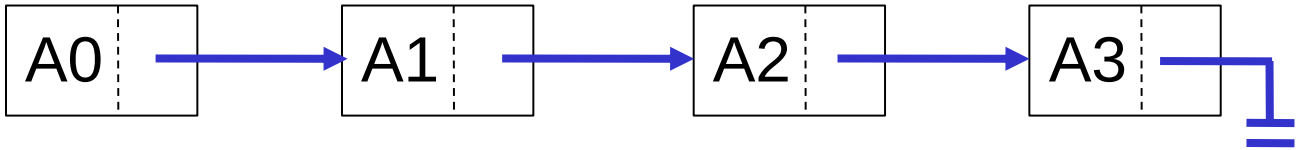
Outline

- Linked Lists vs. Array
- Linked Lists dan Iterators
- Variasi Linked Lists:
  - Doubly Linked Lists
  - Circular Linked Lists
  - Sorted Linked Lists

Tujuan

- Memahami struktur data *linked-list*
- Memahami kompleksitas dari operasi-operasi pada ADT *linked-list* antara lain *insert, delete, read*
- Dapat mengimplementasikan *linked-list*

Linked Lists



- **Menyimpan koleksi elemen secara non-contiguously.**
  - Elemen dapat terletak pada lokasi memory yang saling berjauhan. Bandingkan dengan array dimana tiap-tiap elemen akan terletak pada lokasi memory yang berurutan.
- **Mengizinkan operasi penambahan atau penghapusan elemen ditengah-tengah koleksi dengan hanya membutuhkan jumlah perpindahan elemen yang konstan.**
  - Bandingkan dengan array. Berapa banyak elemen yang harus dipindahkan bila akan menyisipi elemen ditengah-tengah array?

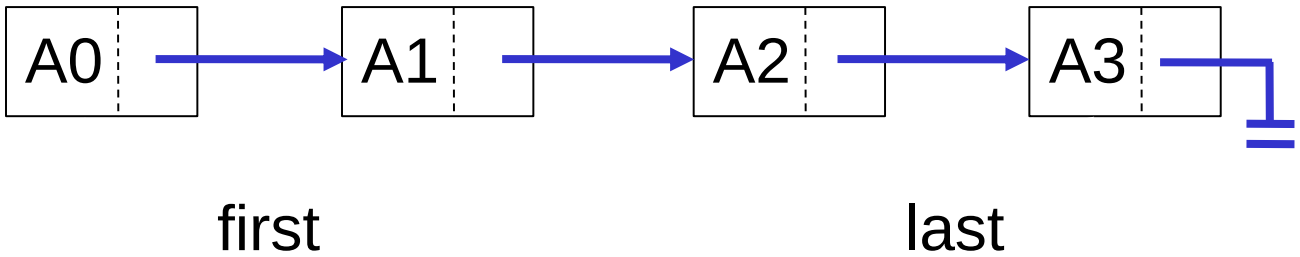
Iterate the Linked List

- Items are stored in contiguous array:

```
//step through array a, outputting each item
for (int index = 0; index < a.length; index++)
     System.out.println (a[index]);
```

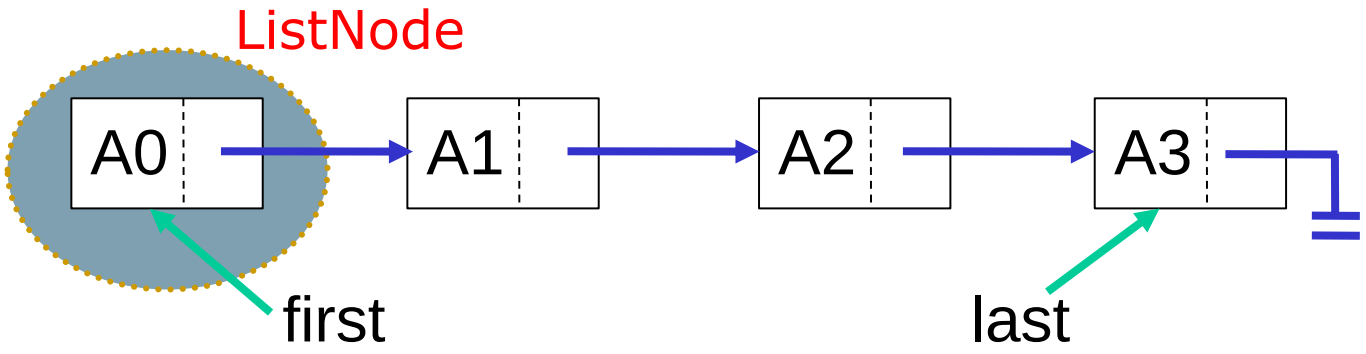
- Items are stored in a linked list non-contiguously :

```
// step through List theList, outputting each item
for (ListNode p = theList.first; p != null; p = p.next)
     System.out.println (p.data);
```



Implementasi: *Linked Lists*

- Sebuah list merupakan rantai dari object bertipe **`ListNode`** yang berisikan data dan referensi (pointer) kepada **`ListNode`** selanjutnya dalam list.
- Harus diketahui dimana letak elemen pertama!



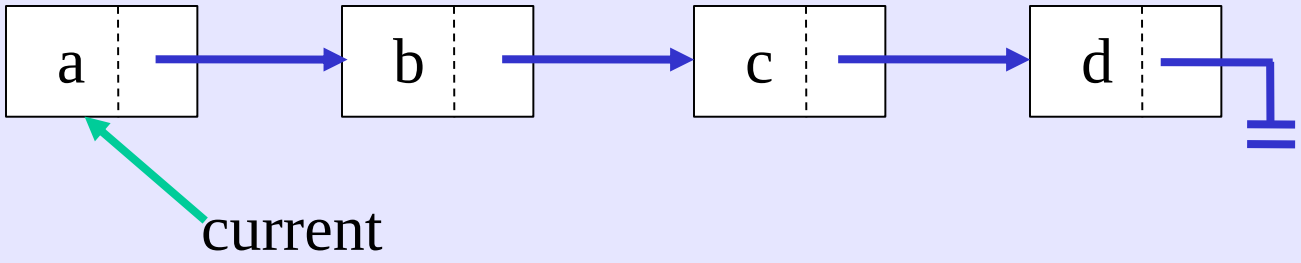
ListNode: Definisi

```
public class ListNode {
    Object element;     // data yang disimpan
    ListNode next;

    // constructors
    ListNode (Object theElement, ListNode n){
        element = theElement;
        next = n;
    }
    ListNode (Object theElement){
        this (theElement, null);
    }
    ListNode (){
        this (null, null);
    }
}
```

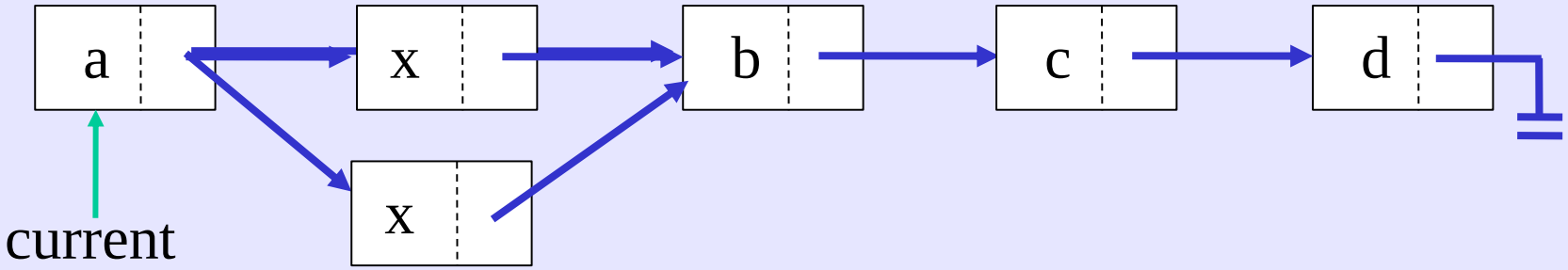
Catatan Penting!

- Yang disimpan dalam **`ListNode`** adalah reference dari object-nya, BUKAN object-nya itu sendiri atau salinan dari object-nya !!!

Linked List: Insertion



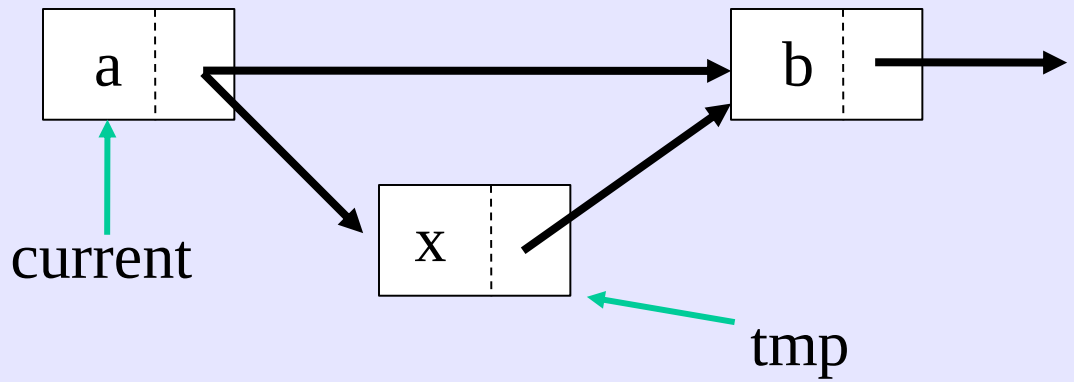
- Menyisipkan ***X*** pada lokasi setelah *current*.



Langkah-langkah menyisipkan

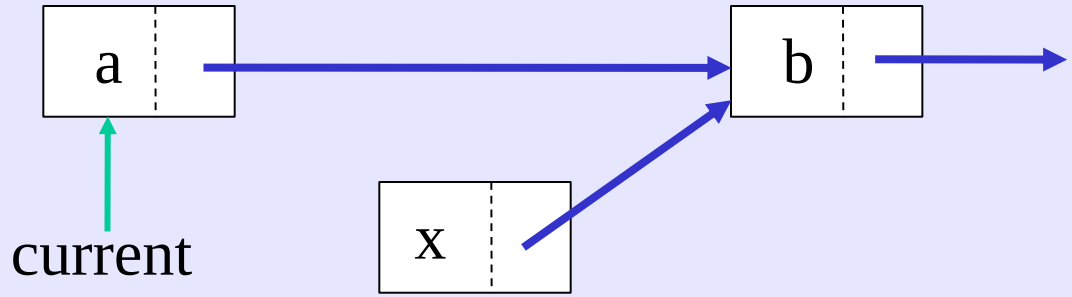
- Menyisipkan elemen baru setelah posisi *current*

```
// Membuat sebuah node
tmp = new ListNode( );
// meletakkan nilai x pada field elemen
tmp.element = x;
// node selanjutnya dari x adalah node b
tmp.next = current.next;
// node selanjutnya dari a adalah node x
current.next = tmp;
```

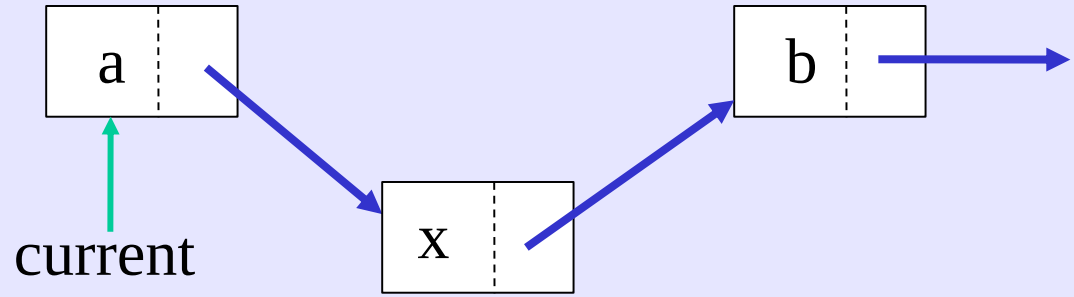


Langkah-langkah menyisipkan yang lebih efisien

```
tmp = new ListNode (x, current.next);
```



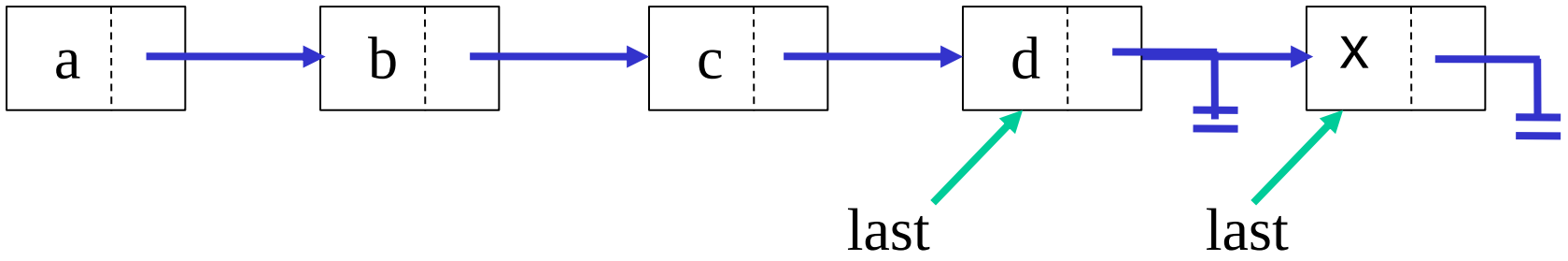

```
current.next = tmp;
```



Linked List: menambahkan elemen diakhir list

- menambahkan *X* pada akhir list

```
// last menyatakan node terakhir dalam linked list
last.next = new ListNode();
last = last.next; // adjust last
last.element = x; // place x in the node
last.next = null; // adjust next
```

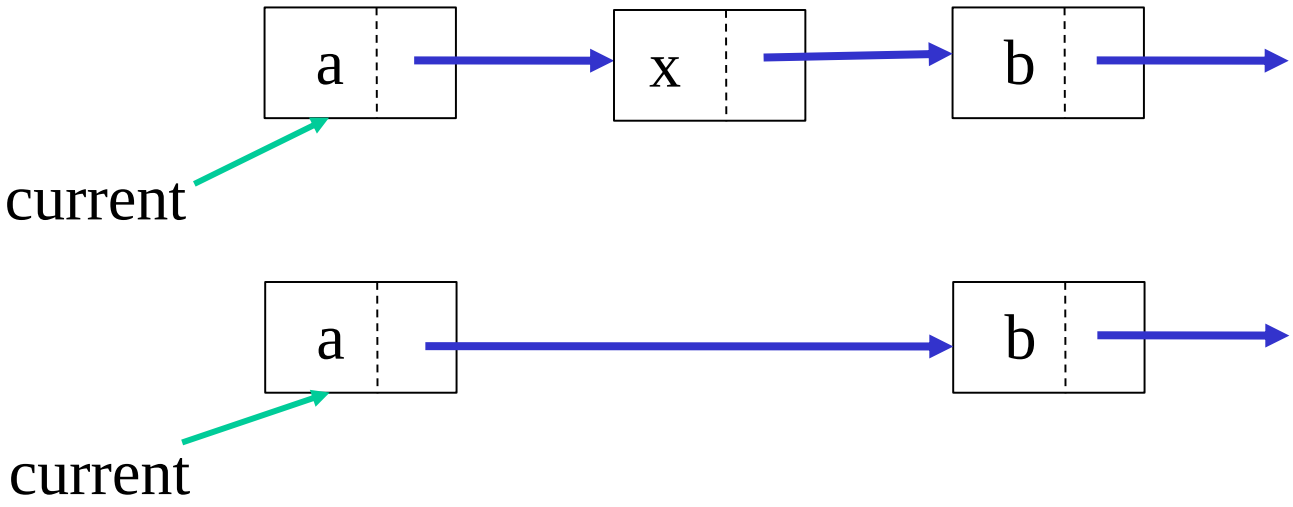


- lebih singkat:

```
last = last.next = new ListNode (x, null);
```

Linked Lists: menghapus elemen

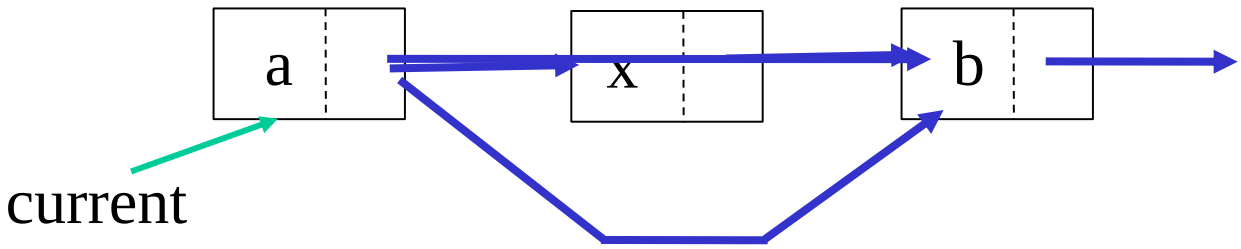
- Proses menghapus dilakukan dengan mengabaikan elemen yang hendak dihapus dengan cara melewati pointer (*reference*) dari elemen tersebut langsung pada elemen selanjutnya.
- Elemen x dihapus dengan meng-*assign field* **`next`** pada elemen a dengan alamat b.



Langkah-langkah menghapus elemen

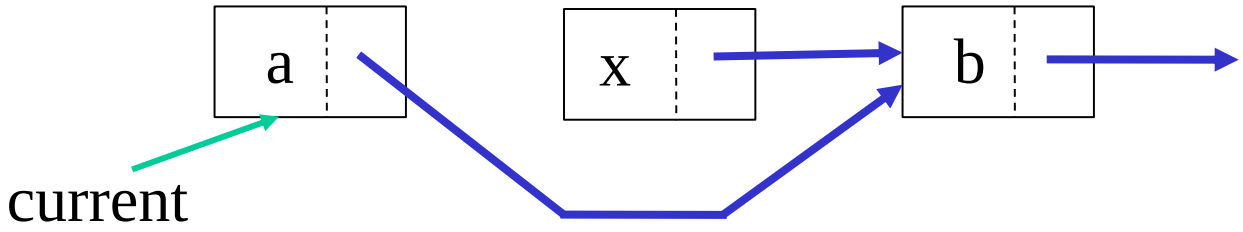
- Butuh menyimpan alamat *node* yang terletak sebelum *node* yang akan dihapus. (pada gambar *node current*, berisi elemen a)

```
current.next = current.next.next;
```



Kapan node x dihapus? Oleh siapa?

Langkah-langkah menghapus elemen



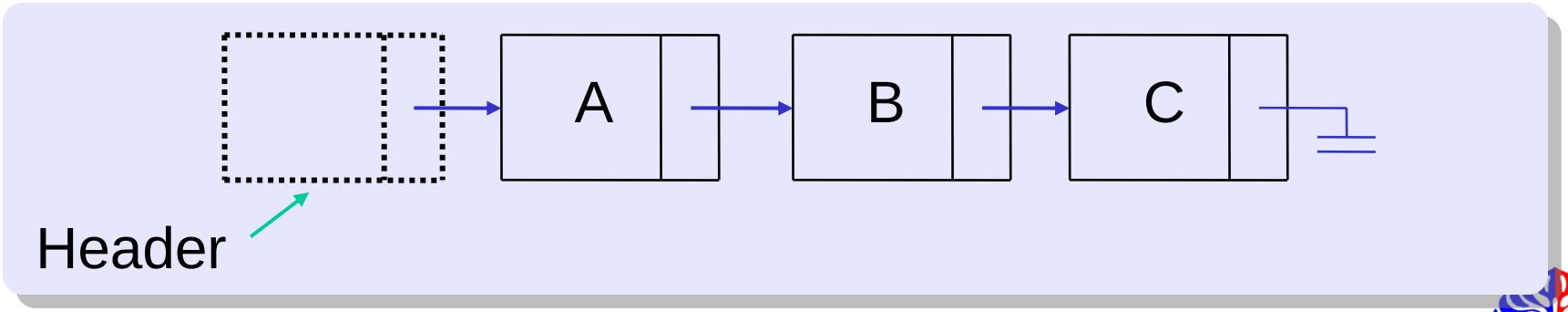
- Tidak ada elemen lain yang menyimpan alamat *node* x.
- *Node* x tidak bisa diakses lagi.
- *Java Garbage Collector* akan membersihkan alokasi memory yang tidak dipakai lagi atau tidak bisa diakses.
- Dengan kata lain, **menghapus node x.**

Pertanyaan:

- Selama ini contoh menyisipkan dan menghapus elemen dengan asumsi ada *node current* yang terletak sebelumnya.
  - Bagaimana menambahkan node pada urutan pertama pada *linked list?*
  - Bagaimana menghapus elemen pertama pada *linked list?*

Header Node

- Menghapus dan menambahkan elemen pertama menjadi kasus khusus.
- Dapat dihindari dengan menggunakan *header node*;
  - Tidak berisikan data, digunakan untuk menjamin bahwa selalu ada elemen sebelum elemen pertama yang sebenarnya pada linked list.
  - Elemen pertama diperoleh dengan: `current = header.next;`
  - Empty list jika: `header.next == null;`
- Proses pencarian dan pembacaan mengabaikan *header node*.



Linked Lists: List interface

```
public interface List
{
    /**
        Test if the list is logically empty.
        @return true if empty, false otherwise.
    */
    boolean isEmpty ();

    /**
        Make the list logically empty.
    */
    void makeEmpty ();
}
```

Linked Lists: List implementation

```
public class LinkedList implements List
{
    // friendly data, so LinkedListItr can have access
    ListNode header;

    /**
        Construct the list
     */
    public LinkedList ()
    {
        header = new ListNode (null);
    }

    /**
        Test if the list is logically empty.
        @return true if empty, false otherwise.
    */
    public boolean isEmpty( )
    {
        return header.next == null;
    }
```

Linked Lists: List implementation

```
    /**
        Make the list logically empty.
    */
    public void makeEmpty( )
    {
        header.next = null;
    }
}
```

Latihan

- Buatlah sebuah method untuk menghitung jumlah elemen dalam sebuah linked list!

```
public static int listSize (LinkedList theList)
{









}
```

Diskusi

- Apakah implementasi tersebut benar?
- Bila kita hendak menambahkan method lain pada sebuah List, haruskah kita mengakses field **`next`** dari object node dan object **`current`** dari object list?
- Dapatkah kita memiliki interface yang lebih baik, dan lebih seragam serta lebih memudahkan untuk mengembangkan method-method lain?

Iterator Class

- Untuk melakukan sebagian besar operasi-operasi pada List, kita perlu menyimpan informasi posisi saat ini. (*current position*).
- Kelas List menyediakan method yang tidak bergantung pada posisi. Method tersebut antara lain: **`isEmpty`**, dan **`makeEmpty`**.
- List iterator (**`ListItr`**) menyediakan method-method yang umum digunakan untuk melakukan operasi pada list antara lain: **`advance`**, **`retrieve`**, **`first`**.
- Internal struktur dari List di encapsulasi oleh List iterator.
- Informasi posisi *current* disimpan dalam object iterator.

Iterator Class

```
// Insert x after current position
void insert (x);
// Remove x
void remove (x);
// Remove item after current position
void removeNext( );
// Set current position to view x
boolean find( x );
// Set current position to prior to first
void zeroth ();
// Set current position to first
void first( );
// Set current to the next node
void advance ();
// True if at valid position in list
boolean isInList ();
// Return item in current position
Object retrieve()
```

- **Exceptions thrown for illegal access, insert, or remove.**

Contoh

- Sebuah method static untuk menghitung jumlah elemen dalam sebuah list.

```
public static int listSize (List theList)
{
    int size = 0;
    ListItr itr = new ListItr (theList);

    for (itr.first(); itr.isInList(); itr.advance())
    {
       size++;
    }
    return size;
}
```

Java Implementations

- Sebagian besar cukup mudah; seluruh method relatif pendek.
- **`ListItr`** menyimpan reference dari object list sebagai private data.
- Karena **`ListItr`** is berada dalam package yang sama dengan **`List`**, sehingga jika field dalam **`List`** adalah ***(package) friendly***, maka dapat di akses oleh **`ListItr`**.

LinkedListItr implementation

```
public class LinkedListItr implements ListItr

/** contains List header. */
protected LinkedList theList;
/** stores current position. */
protected ListNode current;

/**
    Construct the list.
    As a result of the construction, the current position is
    the first item, unless the list is empty, in which case
    the current position is the zeroth item.
    @param anyList a LinkedList object to which this iterator is
        permanently bound.
*/
public LinkedListItr( LinkedList anyList )
{
    theList = anyList;
    current = theList.isEmpty( ) ? theList.header :
        theList.header.next;
}
```

```
/**
  Construct the list.
  @param anyList a LinkedList object to which this iterator is
      permanently bound. This constructor is provided for
      convenience. If anyList is not a LinkedList object, a
      ClassCastException will result.
*/
public LinkedListItr( List anyList ) throws ClassCastException{
    this( ( LinkedList ) anyList );
}

/**
 * Advance the current position to the next node in the list.
 * If the current position is null, then do nothing.
 * No exceptions are thrown by this routine because in the
 * most common use (inside a for loop), this would require the
 * programmer to add an unnecessary try/catch block.
 */
public void advance( ){
    if( current != null )
        current = current.next;
}
```

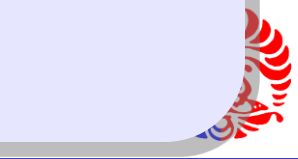
```
/**
 * Return the item stored in the current position.
 * @return the stored item or null if the current position
 * is not in the list.
 */
public Object retrieve( ){
    return isInList( ) ? current.element : null;
}

/**
 * Set the current position to the header node.
 */
public void zeroth( ){
    current = theList.header;
}

/**
 * Set the current position to the first node in the list.
 * This operation is valid for empty lists.
 */
public void first( ){
    current = theList.header.next;
}
```

```
/**
* Insert after the current position.
* current is set to the inserted node on success.
* @param x the item to insert.
* @exception ItemNotFound if the current position is null.
*/
public void insert( Object x ) throws ItemNotFound
{
    if( current == null )
        throw new ItemNotFound( "Insertion error" );

    ListNode newNode = new ListNode( x, current.next );
    current = current.next = newNode;
}
```

```
/**
 * Set the current position to the first node containing an item.
 * current is unchanged if x is not found.
 * @param x the item to search for.
 * @return true if the item is found, false otherwise.
 */
public boolean find( Object x )
{
    ListNode itr = theList.header.next;

    while( itr != null && !itr.element.equals( x ) )
        itr = itr.next;

    if( itr == null )
        return false;

    current = itr;
    return true;
}
```

```
/**
 * Remove the first occurrence of an item.
 * current is set to the first node on success;
 * remains unchanged otherwise.
 * @param x the item to remove.
 * @exception ItemNotFound if the item is not found.
 */
public void remove( Object x ) throws ItemNotFound
{
    ListNode itr = theList.header;

    while( itr.next != null && !itr.next.element.equals( x ) )
        itr = itr.next;

    if( itr.next == null )
        throw new ItemNotFound( "Remove fails" );

    itr.next = itr.next.next;   // Bypass deleted node
    current = theList.header;   // Reset current
}
```

```
/**
 * Remove the item after the current position.
 * current is unchanged.
 * @return true if successful false otherwise.
 */
public boolean removeNext( )
{
    if( current == null || current.next == null )
        return false;
    current.next = current.next.next;
    return true;
}

/**
 * Test if the current position references a valid list item.
 * @return true if the current position is not null and is
 *     not referencing the header node.
 */
public boolean isInList( )
{
    return current != null && current != theList.header;
}
```



Catatan: Exceptions

- Beberapa method dapat menthrow **`ItemNotFound`** exceptions.
- Namun, jangan menggunakan exceptions secara berlebihan karena setiap exception harus di tangkap (*caught*) atau di teruskan (*propagate*). Sehingga menuntut program harus selalu membungkusnya dengan blok **`try/catch`**
- Contoh: method **`advance`** tidak men-throw exception, walaupun sudah berada pada akhir elemen.
- Bayangkan bagaimana implementasi method **`listSize`** bila method **`advance`** men-throw exception!

Linked List *Properties*

- Analisa Kompleksitas Running Time
  - insert next, prepend - O(1)
  - delete next, delete first - O(1)
  - find - O(n)
  - retrieve current position - O(1)
- Keuntungan
  - Growable (bandingkan dengan array)
  - Mudah (quick) dalam read/insert/delete elemen pertama dan terakhir (jika kita juga menyimpan referensi ke posisi terakhir, tidak hanya posisi head/current)
- Kerugian
  - Pemanggilan operator **`new`** untuk membuat node baru. (bandingkan dengan array)
  - Ada overhead satu reference untuk tiap node

Mencetak seluruh elemen Linked List

- Cara 1: Tanpa Iterator, loop

```
public class LinkedList {

public void print ()
   {
      // step through list, outputting each item
      ListNode p = header.next;
      while (p != null) {
           System.out.println (p.data);
           p = p.next;
      }
   }
}
```

Mencetak seluruh elemen Linked List(2)

- Cara 2: Tanpa Iterator, Recursion

```
public class LinkedList {
...
private static void printRec (ListNode node)
   {
     if (node != null) {
          System.out.println (node.data);
          printRec (node.next);
     }
   }

public void print ()
   {
     printRec (header.next);
   }
}
```

Mencetak seluruh elemen Linked List(3)

- Cara 3: Recursion

```
class ListNode{
...
public void print ()  {
    System.out.println (data);
    if (next != null) {
        next.print ();
    }
  }
} // end of class ListNode

class LinkedList {
public void print ()  {
    if (header.next != null) {
        header.next.print ();
    }
  }
}
```

Mencetak seluruh elemen Linked List(4)

- Cara 4: Menggunakan iterator

```
class LinkedList
{
...
public void print ()
{
     ListItr itr = new ListItr (this);

     for (itr.first(); itr.isInList();
         itr.advance())
     {
      System.out.println (itr.retrieve ());
     }
}

}
```

Sorted Linked Lists

- Menjaga elemen selalu disimpan terurut.
- Hampir seluruh operasi sama dengan linked list kecuali **`insert`** (menambahkan data).
- Pada dasarnya sebuah sorted linked list adalah sebuah linked list. Dengan demikian *inheritance* bisa diterapkan. Kelas **`SortListItr`** dapat diturunkan dari kelas **`ListItr`**.
- Perlu diingat, elemen pada sorted linked list haruslah mengimplement **`Comparable`**.

Implementasi

- Terapkan inheritance,
- Buat method **`Insert`** yang akan mengoveride method milik kelas LinkedList.

```
public void insert( Comparable X )
```

Catatan Penting:

- **ListItr** menggunakan method **equals** pada implementasi **find** dan **remove**.
- Harus dipastikan Class yang digunakan sebagai element (mis: **MyInteger**) memiliki method **equals**
- Signature: (harus sama persis)

```
public boolean equals( Object Rhs )
```

- Signature berikut ini salah!

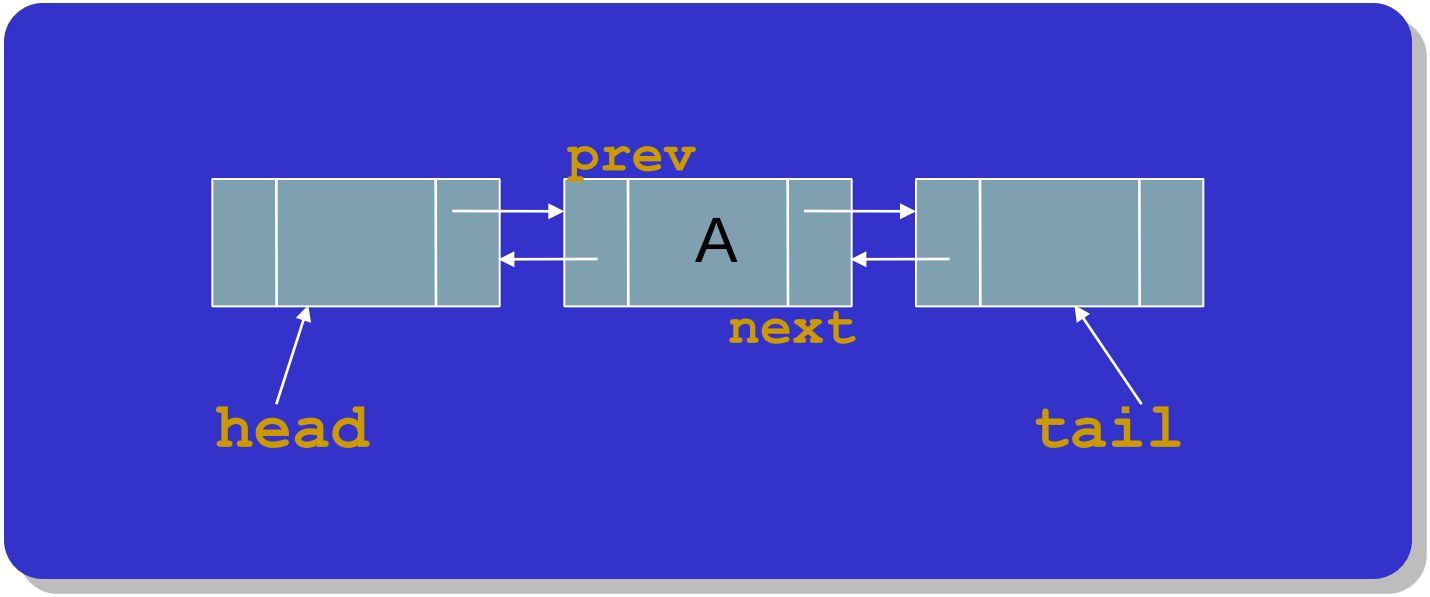
```
public boolean equals( Comparable Rhs )
```

- method **equals** dari class **Object** dapat diturunkan dan digunakan:

```
public boolean equals (Object obj){
    return (this == obj);
}
```

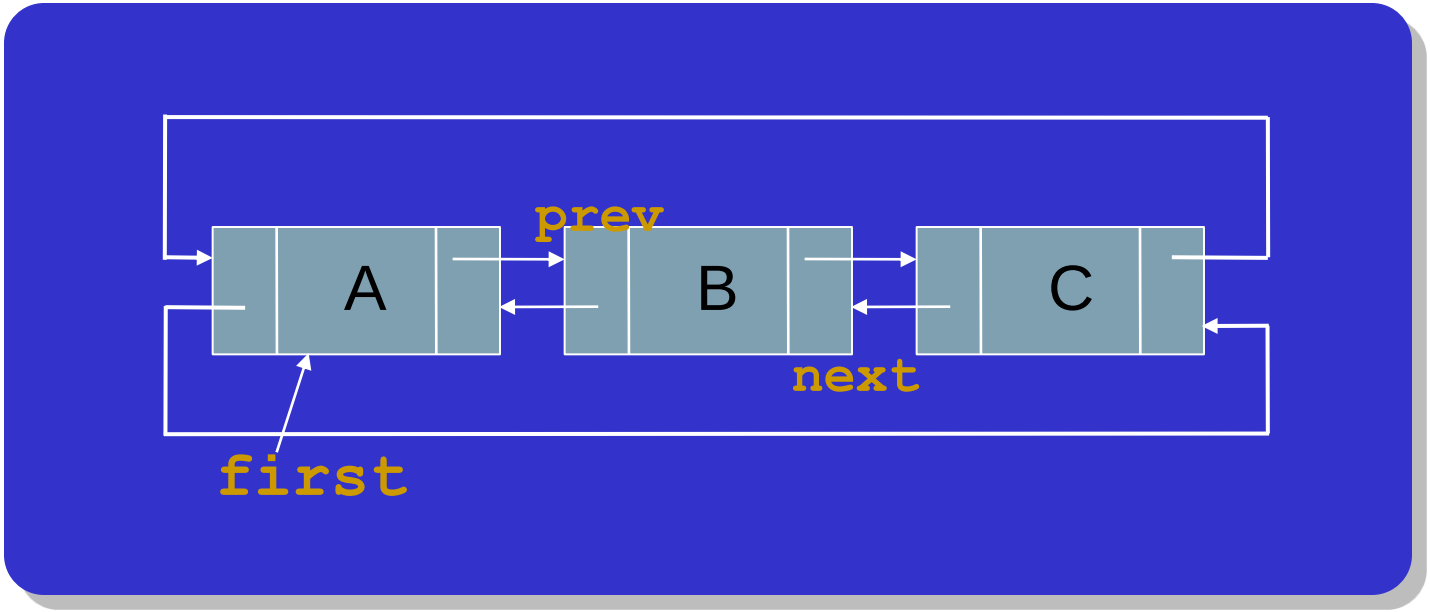
Variasi Linked Lists

- *Doubly-linked lists*: Tiap list node menyimpan referensi node **sebelum dan sesudahnya**. Berguna bila perlu melakukan pembacaan linkedlist dari dua arah.



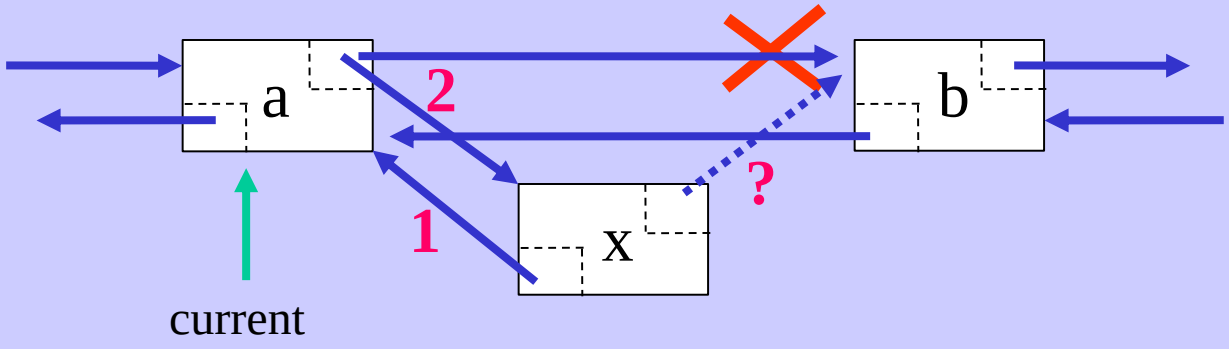
Variasi Linked Lists

- *Circular-linked lists*: **Node terakhir menyimpan referensi node pertama**. Dapat diterapkan dengan atau tanpa header node.

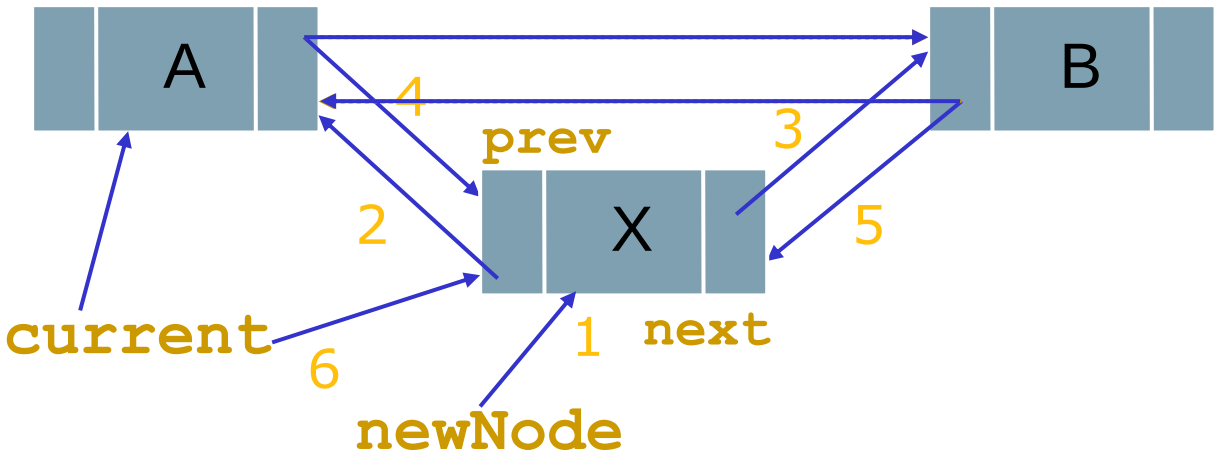


Doubly-linked lists: InsertNext

```
newNode = new DoublyLinkedListNode(x);
1 newNode.prev = current;
2 newNode.prev.next = newNode;
  …
  …
```



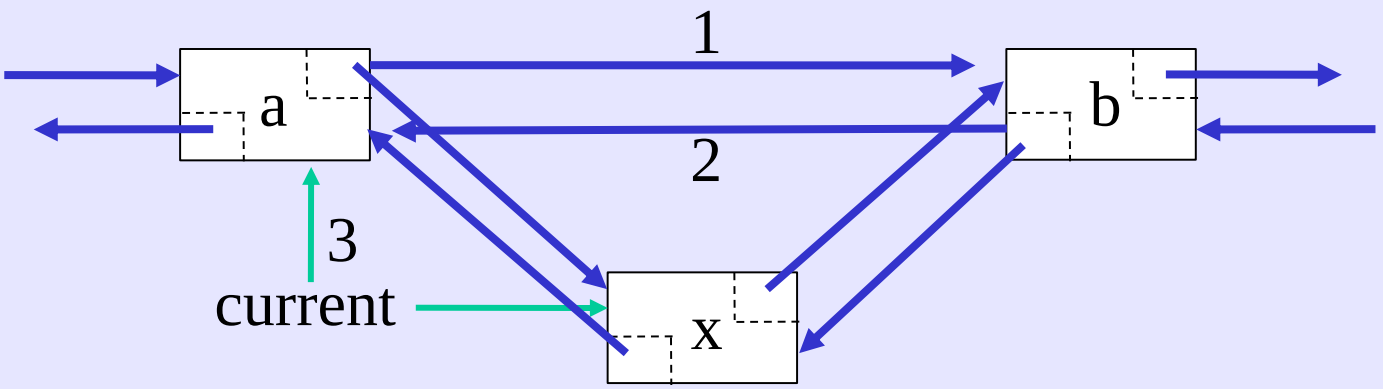
Doubly-linked lists: insertNext




```
1 newNode = new DoublyLinkedListNode(x);
2 newNode.prev = current;
3 newNode.next = current.next;
4 newNode.prev.next = newNode;
5 newNode.next.prev = newNode;
6 current = newNode;
```

Doubly-linked lists: DeleteCurrent

```
1. current.prev.next = current.next;
2. current.next.prev = current.prev;
3. current = current.prev;
```



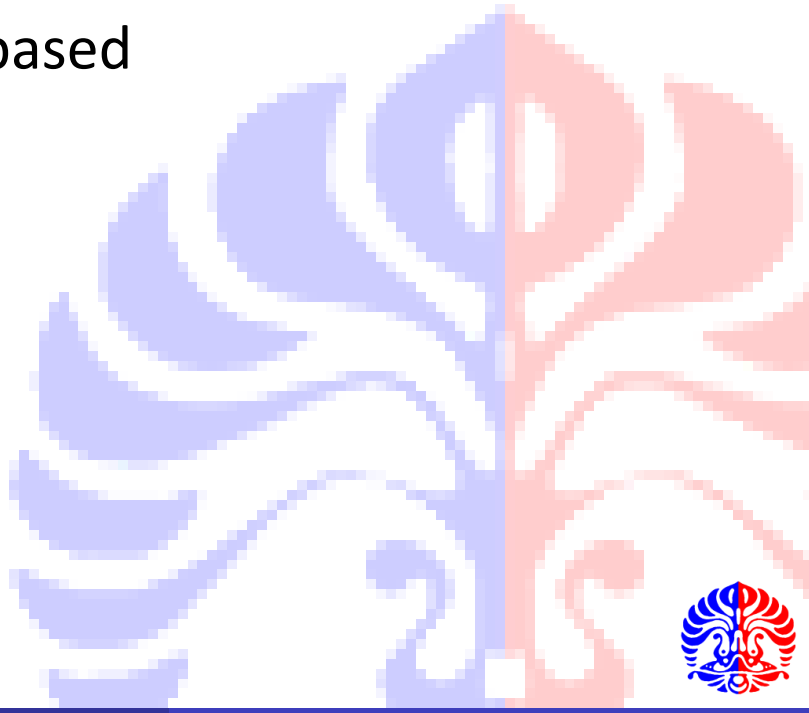
Rangkuman

- ListNode
- List, LinkedList dan variasinya
- Iterator class
- Kelebihan & kekurangan dari linked list
  - Growable
  - Overhead a pointer, new operator untuk membuat node.
  - Hanya bisa diakses secara sequential.

STACK

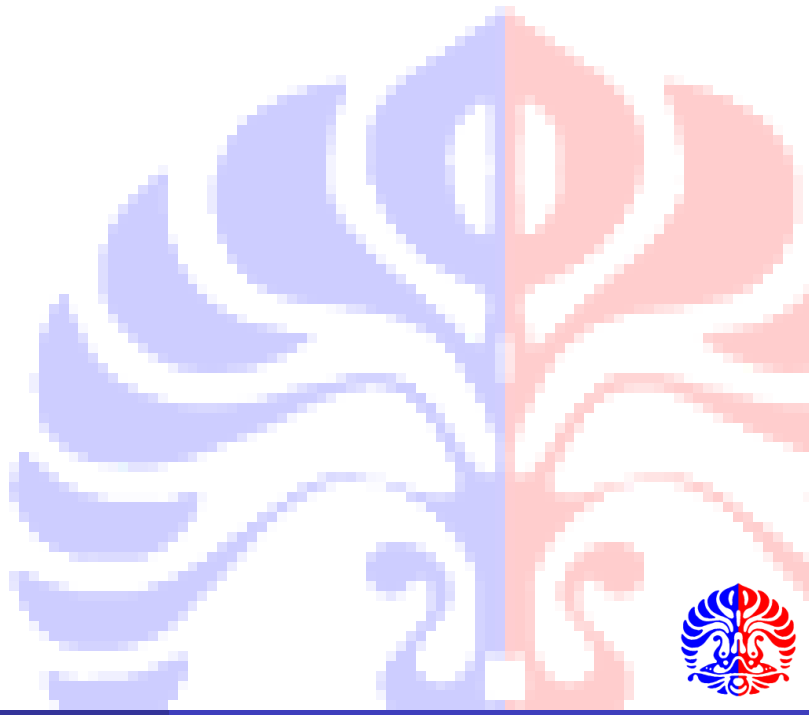
Outline

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



Tujuan

- Memahami cara kerja dan kegunaan stack
- Dapat mengimplementasi stack

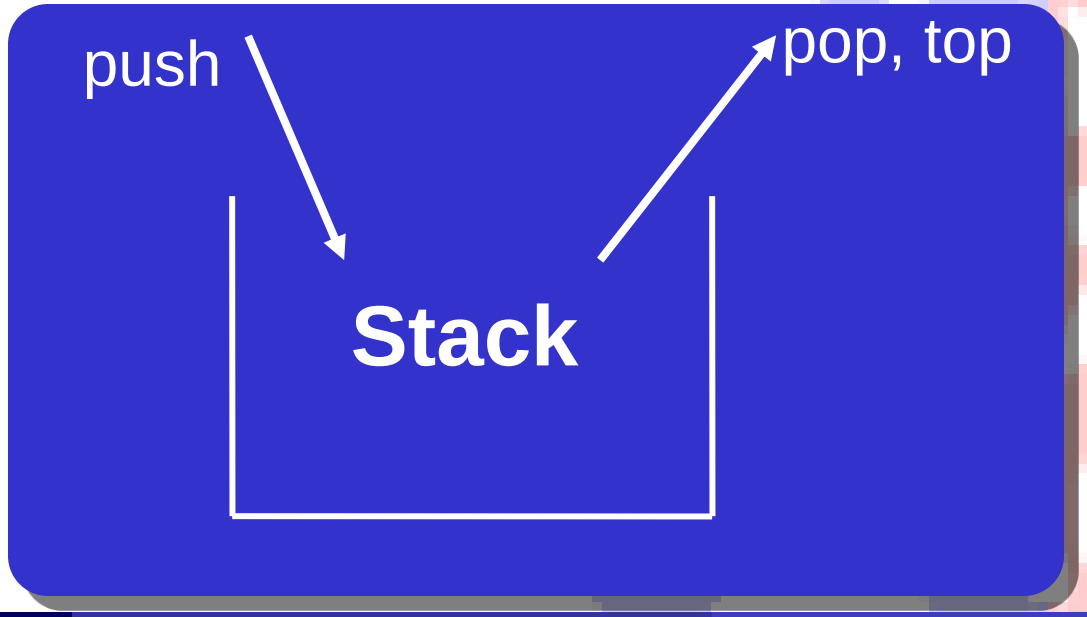


Struktur data linear

- 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).

Stack

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



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).

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!

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**.

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.

Performance

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

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.

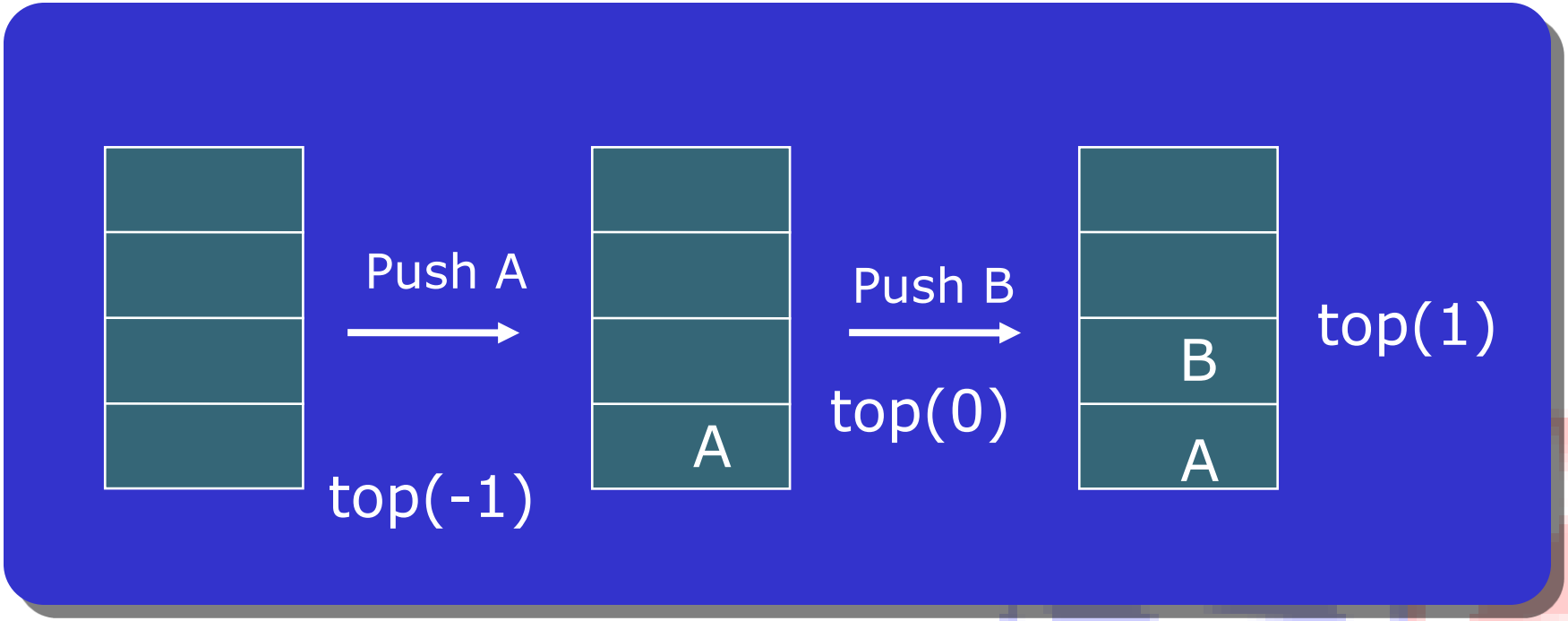
Aplikasi Lainnya

- 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* (di kuliah berikutnya: TBA)
- Pembalikan urutan (*reversing*) dapat dengan mudah dilakukan dengan bantuan stack

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`**.

Ilustrasi



Pertanyaan:

Apa yang terjadi bila nilai top = ukuran array?

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];
```

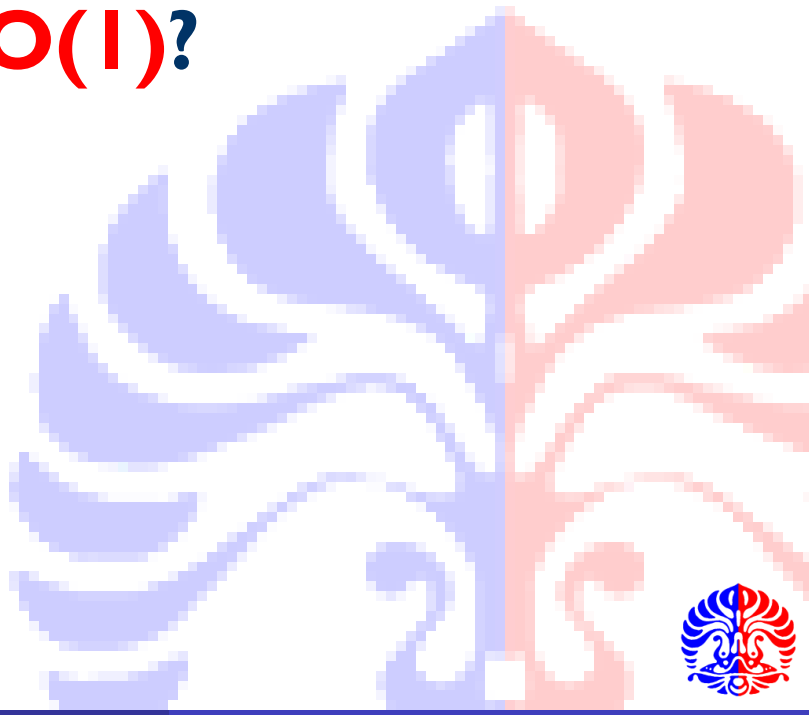
Array Doubling

- Cara lebih baik di Java:
  - menggunakan method java.util.Arrays.copyOf
  - lebih cepat karena menggunakan native implementation

```
array = Arrays.copyOf (array,
    array.length * 2);
```

Pertanyaan:

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



Running Time

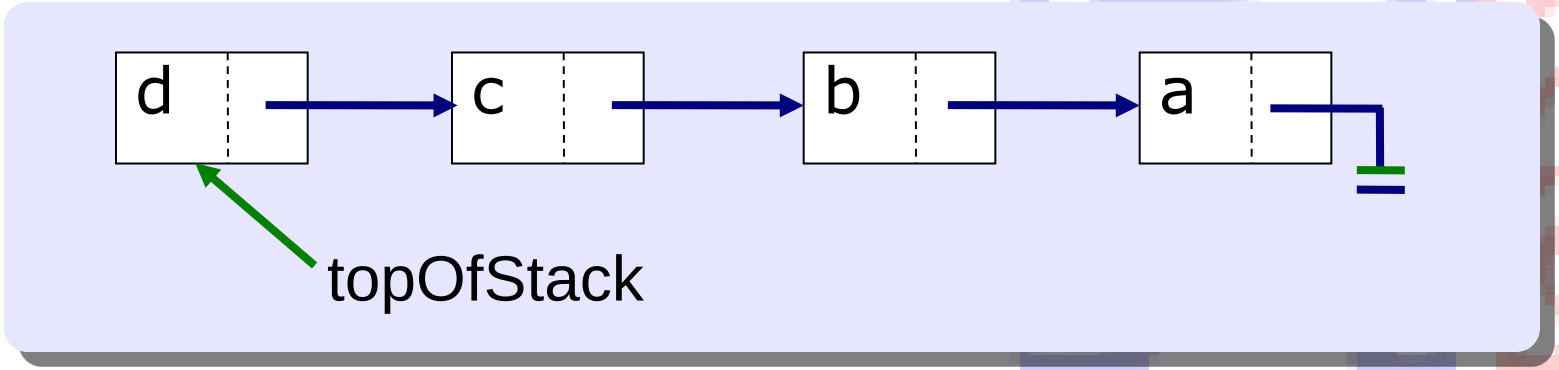
- Tanpa adanya array doubling, setiap operasi memiliki waktu konstan, dan tidak bergantung pada jumlah item di dalam stack.
- Dengan adanya array doubling, satu operasi push dapat (namun jarang) menjadi $O(N)$. Namun, pada dasarnya adalah $O(1)$ karena setiap array doubling yang memerlukan N assignments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

Stack Implementation: Array

```
public class MyArrayStack<T>
{
    private T[] array;
    private int topOfStack;
    private static final int DEFAULT_CAPACITY = 10;

    public MyArrayStack() …
    public boolean isEmpty() …
    public void makeEmpty() …
    public T top() …
    public void pop() …
    public T topAndPop() …
    public void push(T x) …

    private void doubleArray() …
}
```

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



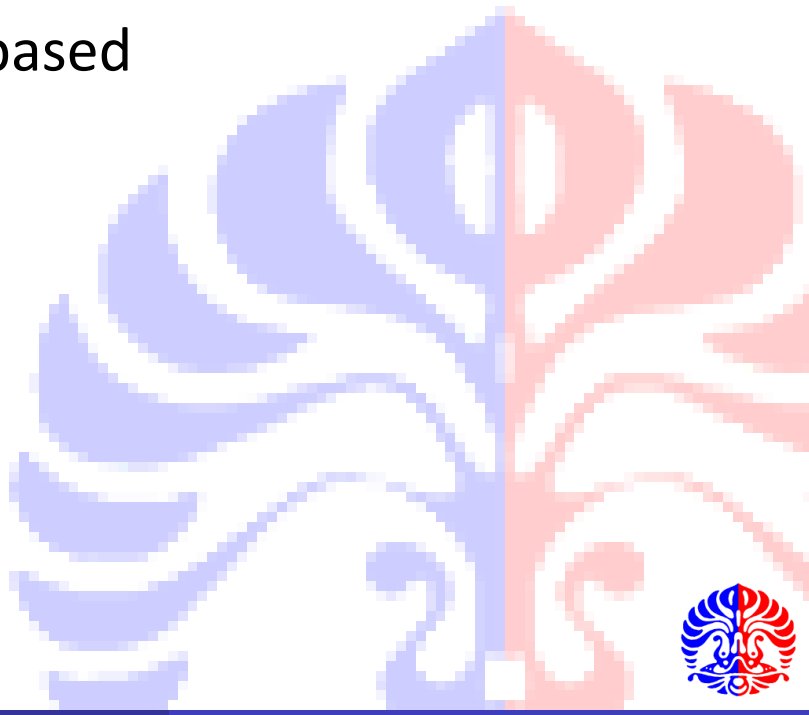
Stack Implementation: Linked List

```
public class MyLinkedListStack<T>
{
    private ListNode<T> topOfStack;

    public MyLinkedListStack() …
    public boolean isEmpty() …
    public void makeEmpty() …
    public T top() …
    public void pop() …
    public T topAndPop() …
    public void push(T x) …
}
```

QUEUE

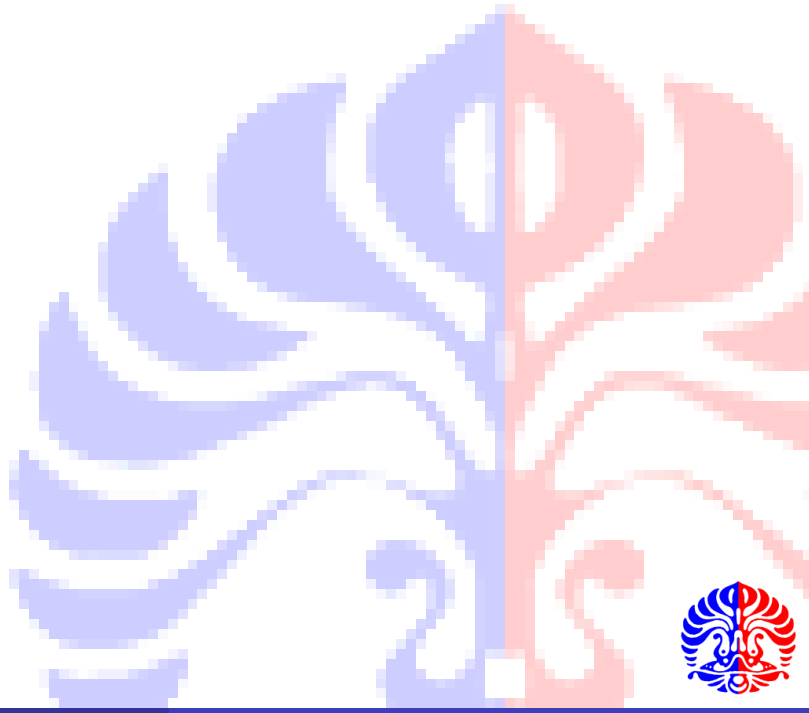
Outline

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



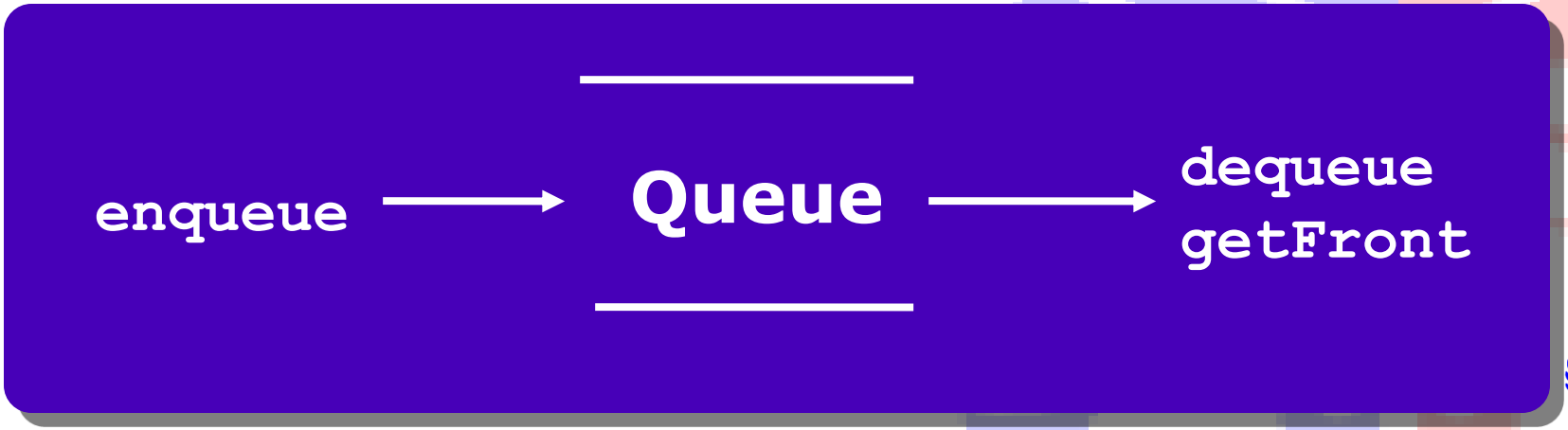
Tujuan

- Memahami cara kerja dan kegunaan queue
- Dapat mengimplementasi queue



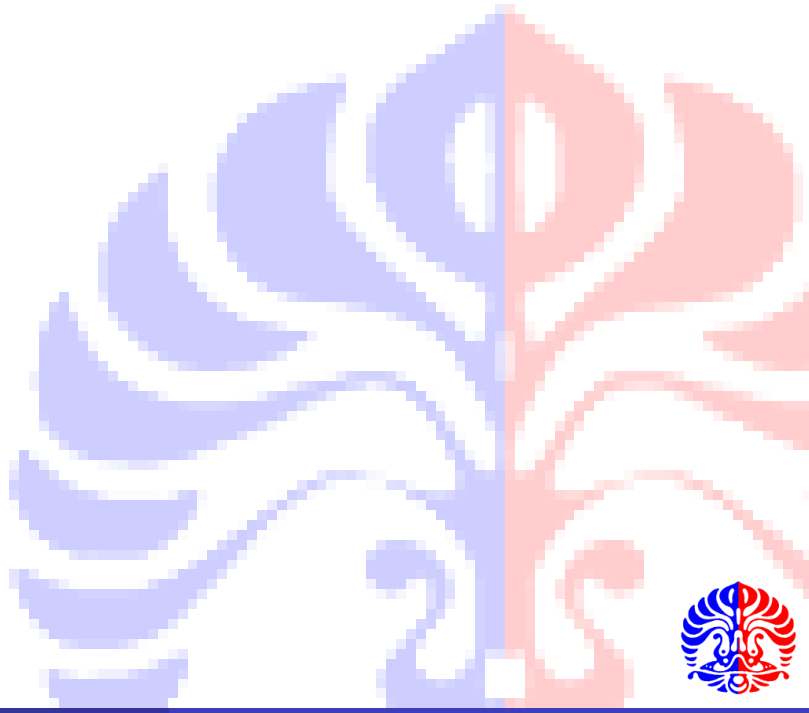
Queue

- 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

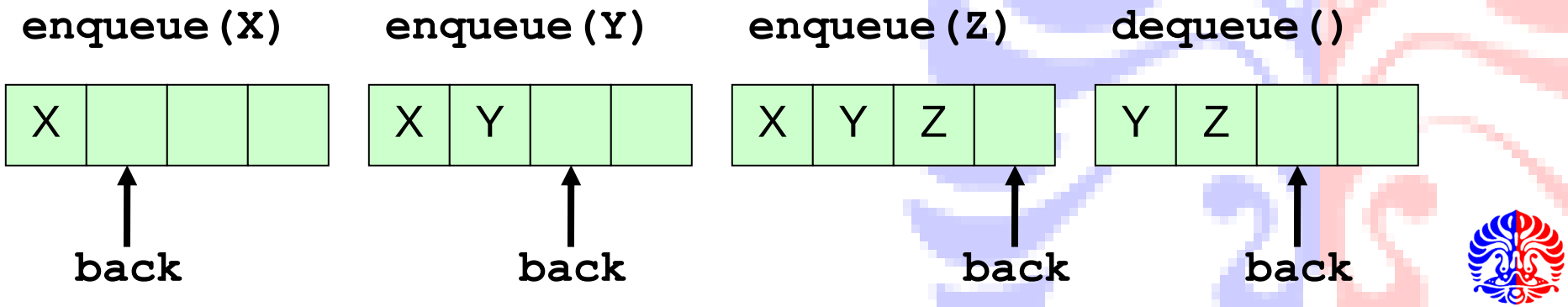


Aplikasi-aplikasi

- Queue berguna untuk menyimpan pekerjaan yang tertunda.
- Kita kelak akan melihat beberapa contoh penggunaannya dalam kuliah-kuliah selanjutnya:
  - Shortest paths problem
    - Lihat diskusi di forum rekursif mengenai *maze runner*. Apakah ADT queue dapat membantu membuat versi non-rekursif yang memberikan jarak terpendek?
  - Topological ordering: given a sequence of *events*, and pairs (*a*,*b*) indicating that event *a* MUST occur prior to *b*, provide a schedule.

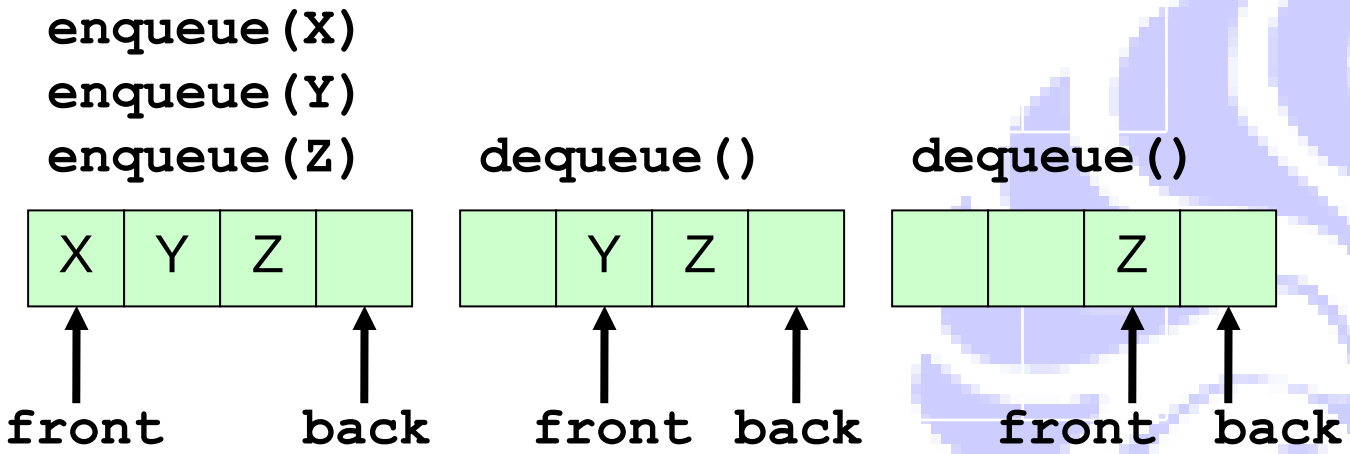
Implementasi dengan Array – cara naive

- 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).



Queue penuh?

- Apa yang terjadi bila index

  `back = Array.length-1` ?

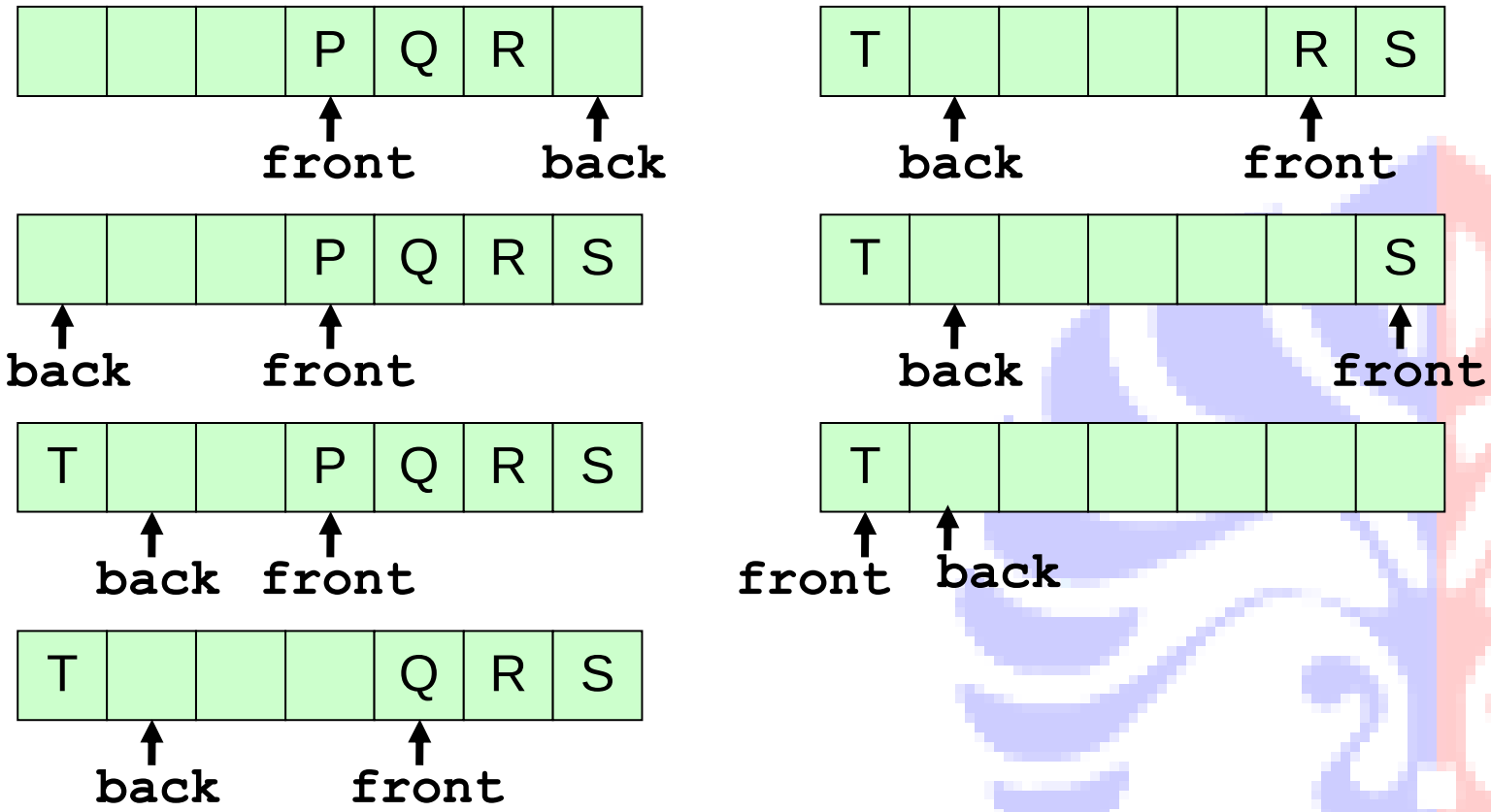
  - Queue penuh?
  - Perlukah dilakukan array doubling?
- Apa yang terjadi bila queue kemudian di enqueue, hingga index

  `first = Array.length-1` ?

  - Apakah queue penuh?
  - Perlukan dilakukan array doubling?

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.

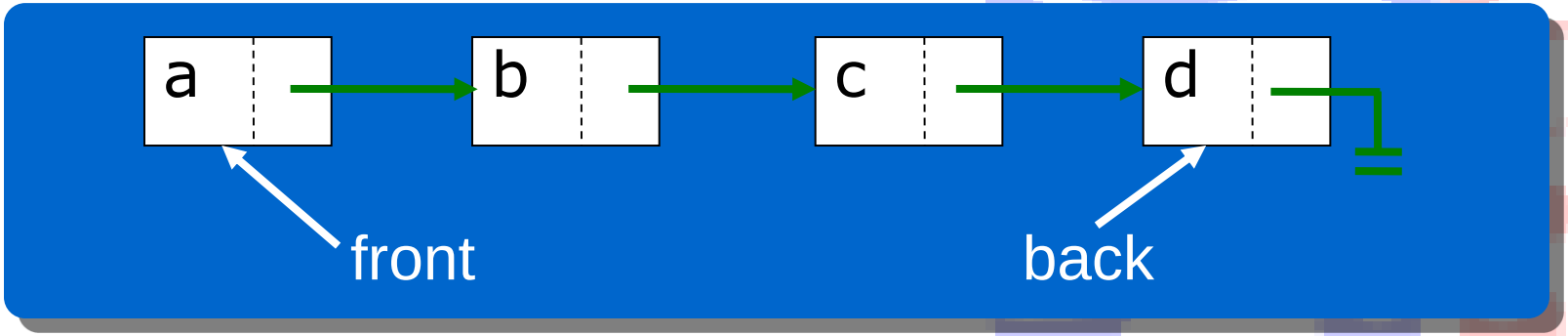


Latihan: Implementasi Array Circular

- 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`
- Kerjakan berpasangan.

Implementasi dengan Linked-List

- Simpan 2 reference: **`front`** → … → … → **`back`**
- **`enqueue(Benda x)`:**
  - Buat sebuah node baru **`N`** yang datanya **`x`**
  - if queue sebelumnya *empty*, maka **`front = back = N`**
  - else tambahkan **`N`** di akhir (dan update **`back`**)
- **`dequeue()`:**
  - Hapus elemen pertama: **`front = front.next`**



Queue Implementation: Linked List

```
public class MyLinkedListQueue<T>
{
    private ListNode<T> front;
    private ListNode<T> back;

    public MyLinkedListQueue() …
    public boolean isEmpty() …
    public void makeEmpty() …
    public T getFront() …
    public void dequeue() …
    public T getFrontAndDequeue() …
    public void enqueue(T x) …
}
```

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!