Struktur Data dan Algoritma

Implementasi ADT: *Linked - List*

Suryana Setiawan, Ruli Manurung & Ade Azurat
(acknowledgments: Denny)

Fasilkom UI

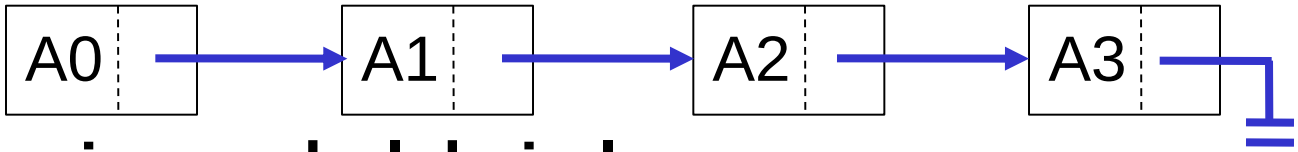
Outline

- 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?

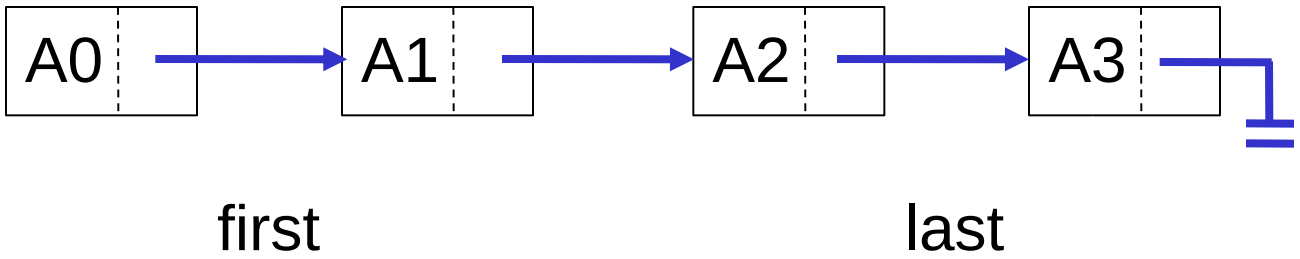
Iterate the Linked List

- 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]);
```

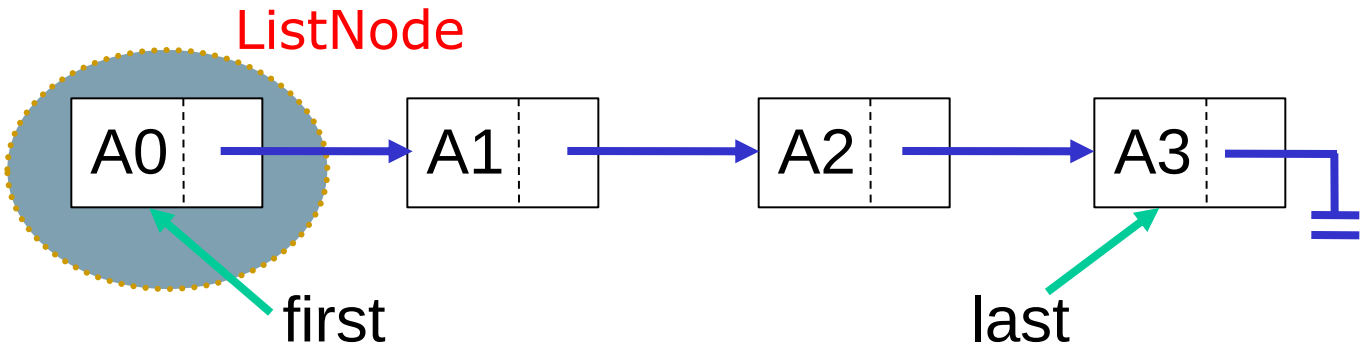
- Items are stored in a linked list non-contiguously :

```
// 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

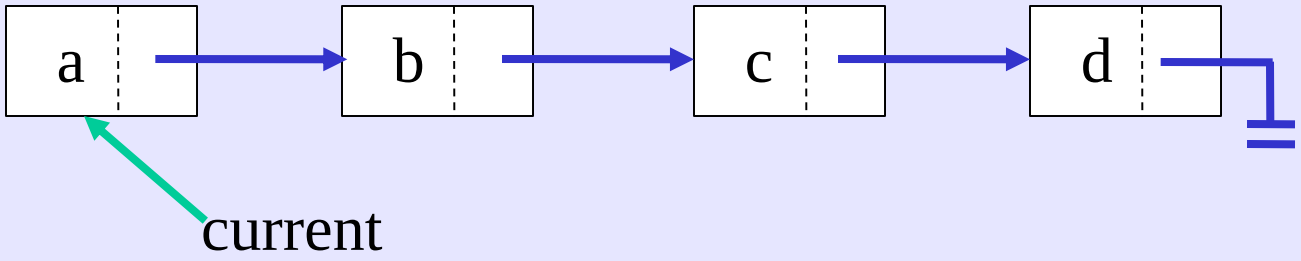
```
public class ListNode {
    Object element;     // data yang disimpan
    ListNode next;

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

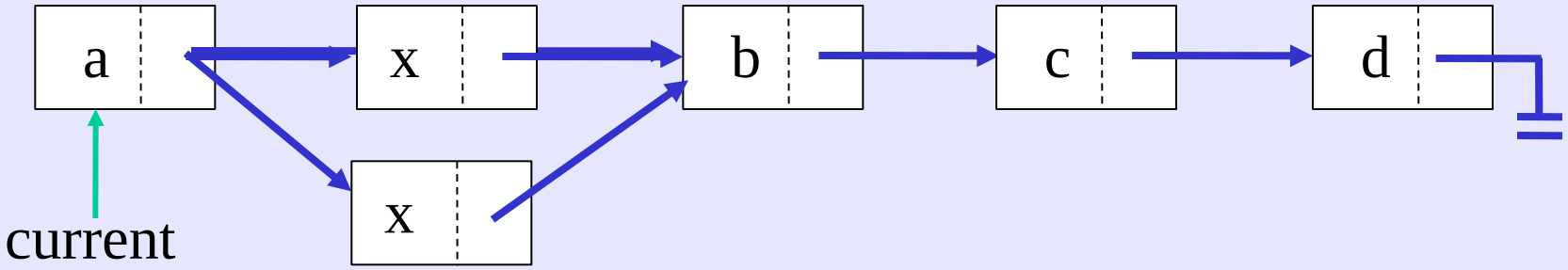
Catatan Penting!

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

Linked List: Insertion



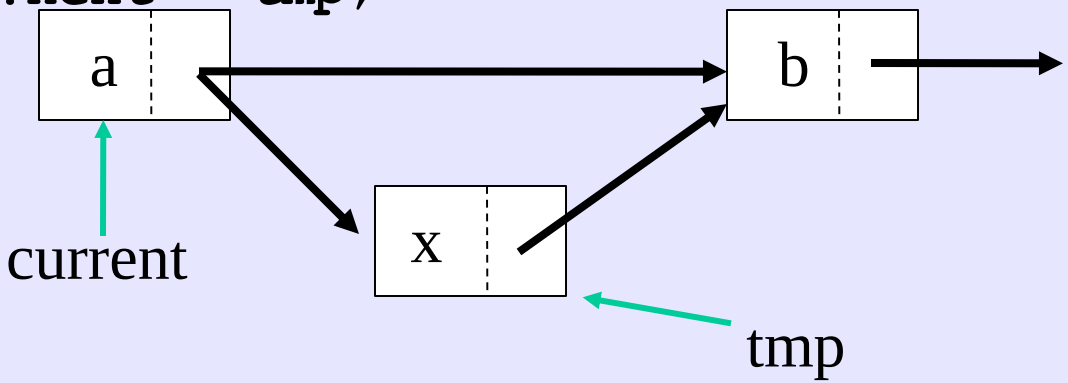
- Menyisipkan ***X*** pada lokasi setelah *current*.



Langkah-langkah menyisipkan

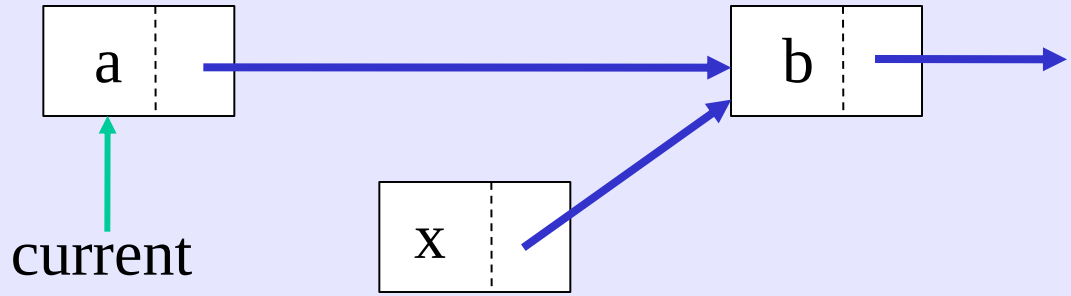
- Menyisipkan elemen baru setelah posisi *current*

```
// 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;
```

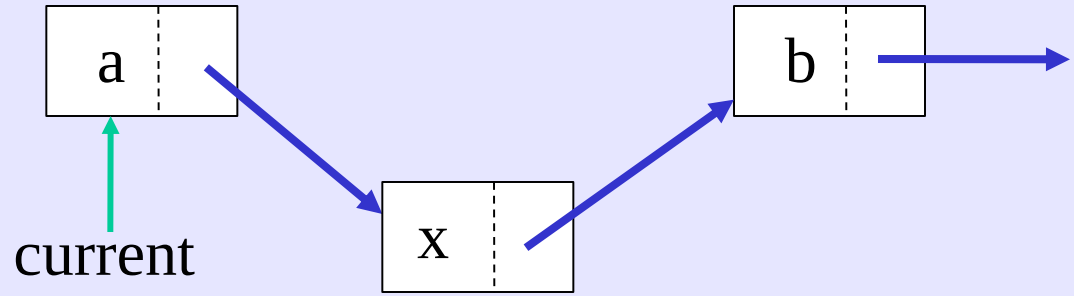


Langkah-langkah menyisipkan yang lebih efisien

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



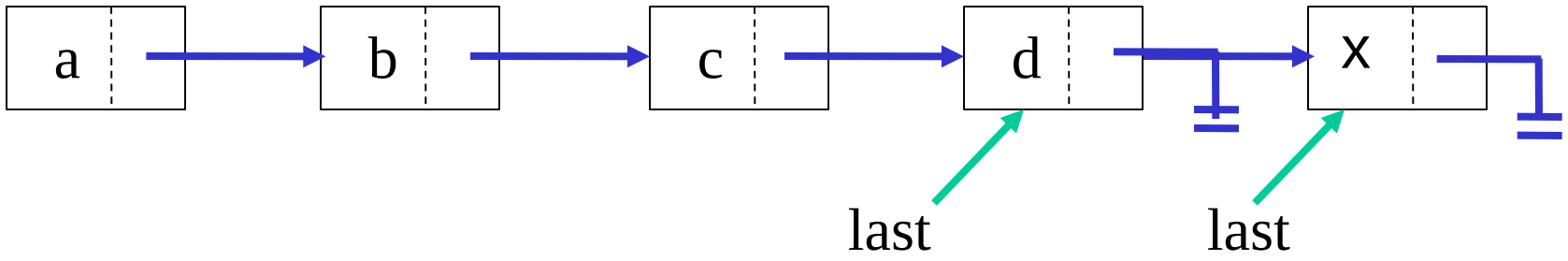

```
current.next = tmp;
```



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
```

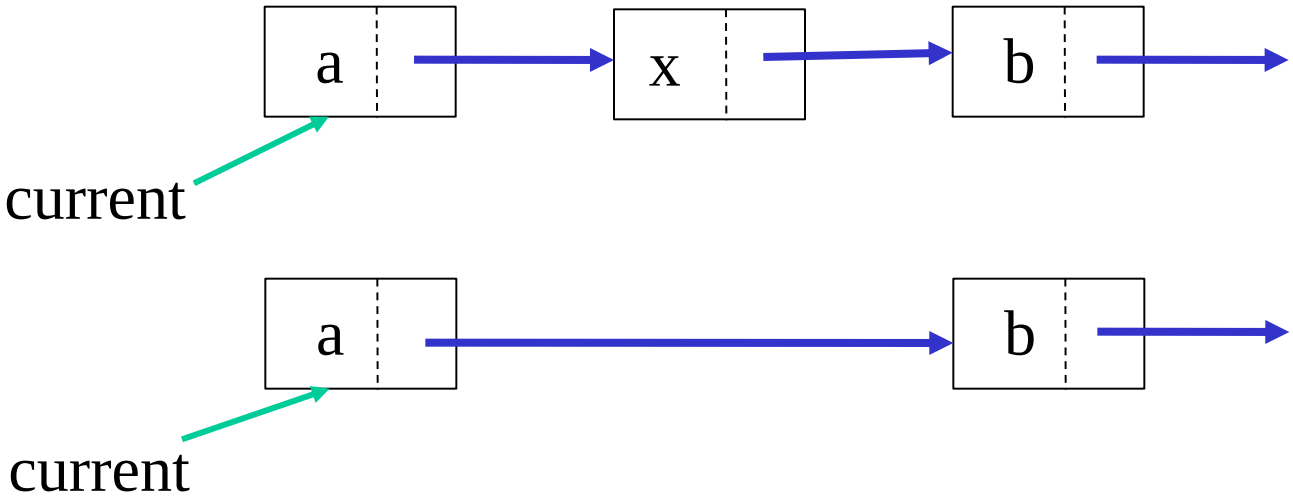


- lebih singkat:

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

Linked Lists: menghapus elemen

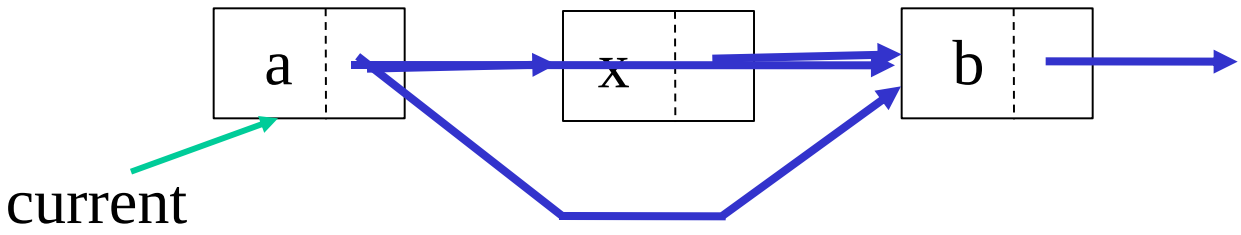
- Proses menghapus dilakukan dengan mengabaikan elemen yang hendak dihapus dengan cara melewati pointer (*reference*) dari elemen tersebut langsung pada elemen selanjutnya.
- Elemen x dihapus dengan meng-*assign field* **`next`** pada elemen a dengan alamat b.



Langkah-langkah menghapus elemen

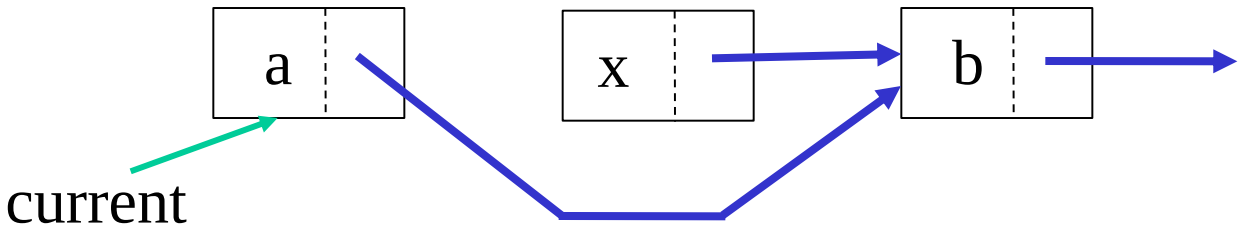
- Butuh menyimpan alamat *node* yang terletak sebelum *node* yang akan dihapus. (pada gambar *node current*, berisi elemen a)

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



Kapan node x dihapus? Oleh siapa?

Langkah-langkah menghapus elemen



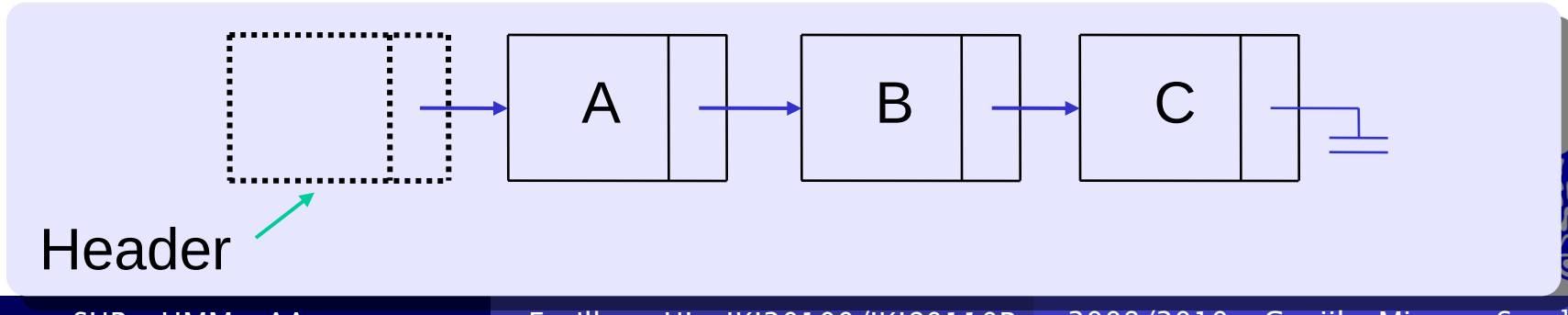
- Tidak ada elemen lain yang menyimpan alamat *node* x.
- *Node* x tidak bisa diakses lagi.
- *Java Garbage Collector* akan membersihkan alokasi memory yang tidak dipakai lagi atau tidak bisa diakses.
- Dengan kata lain, **menghapus node x.**

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 (List theList)
{
    ListItr itr = new ListItr (theList);

    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!

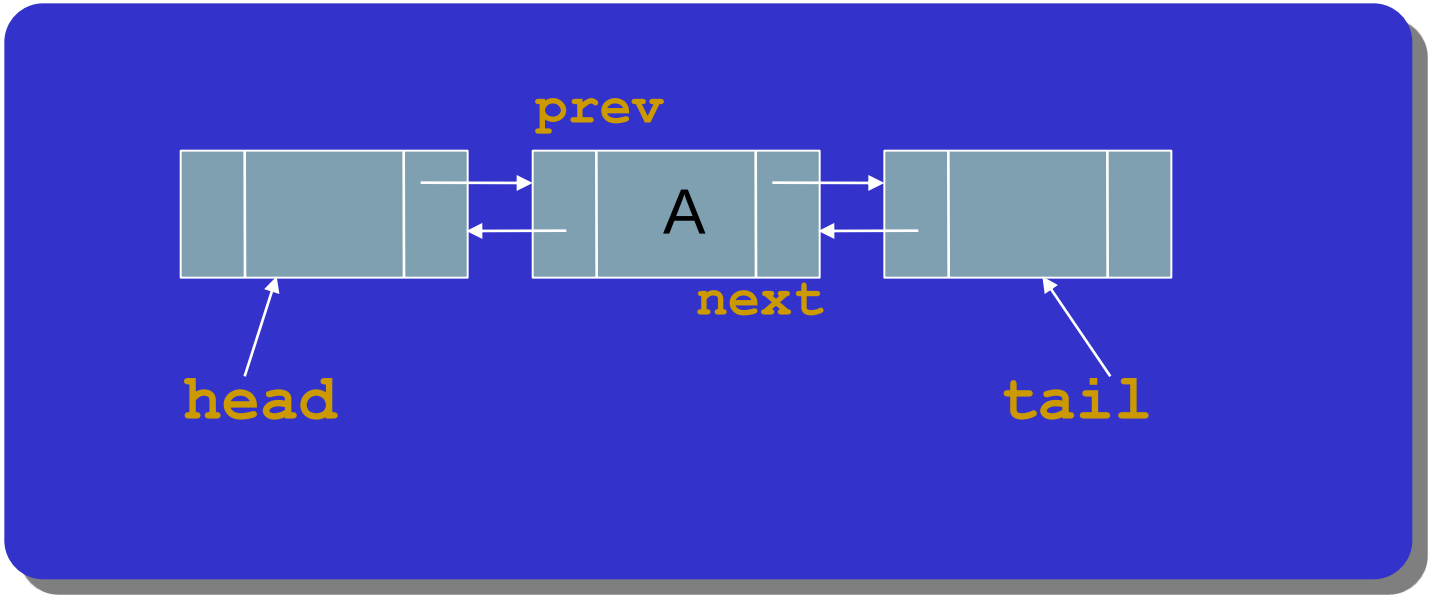
```
public boolean equals( Comparable Rhs )
```

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

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

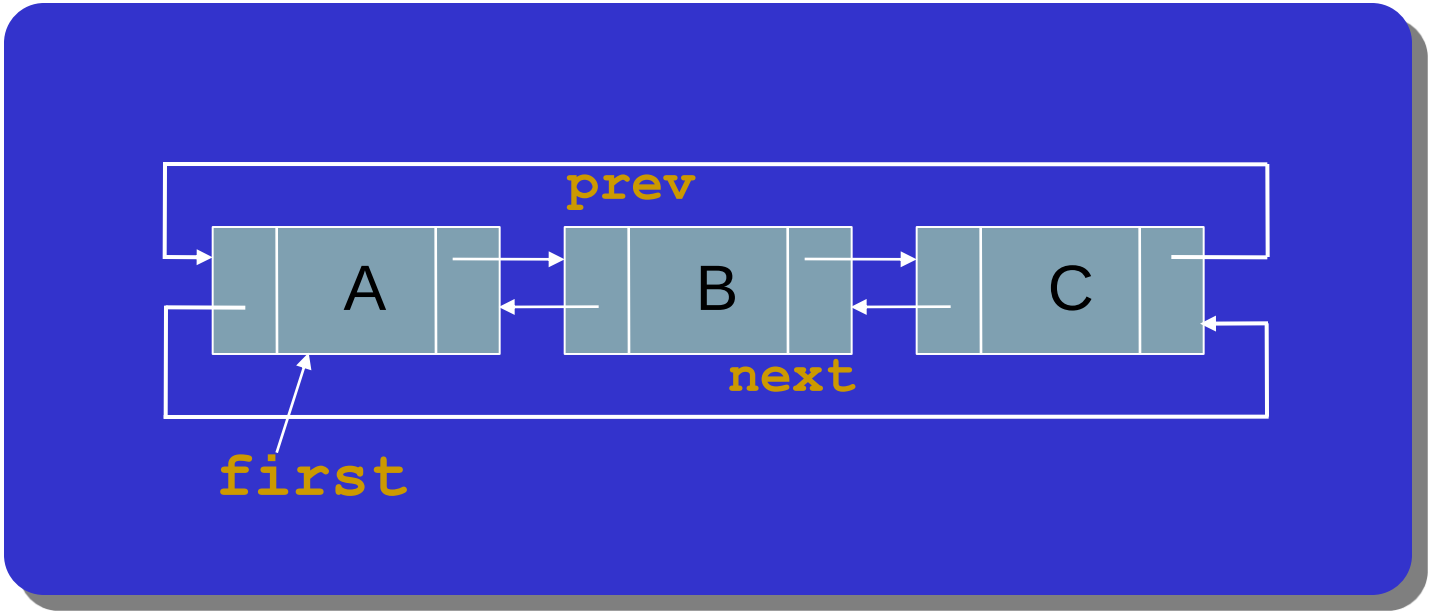
Variasi Linked Lists

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



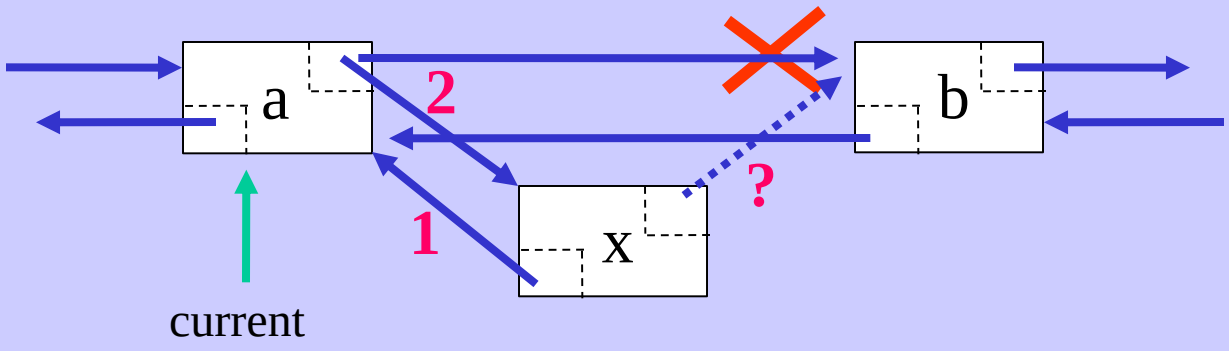
Variasi Linked Lists

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

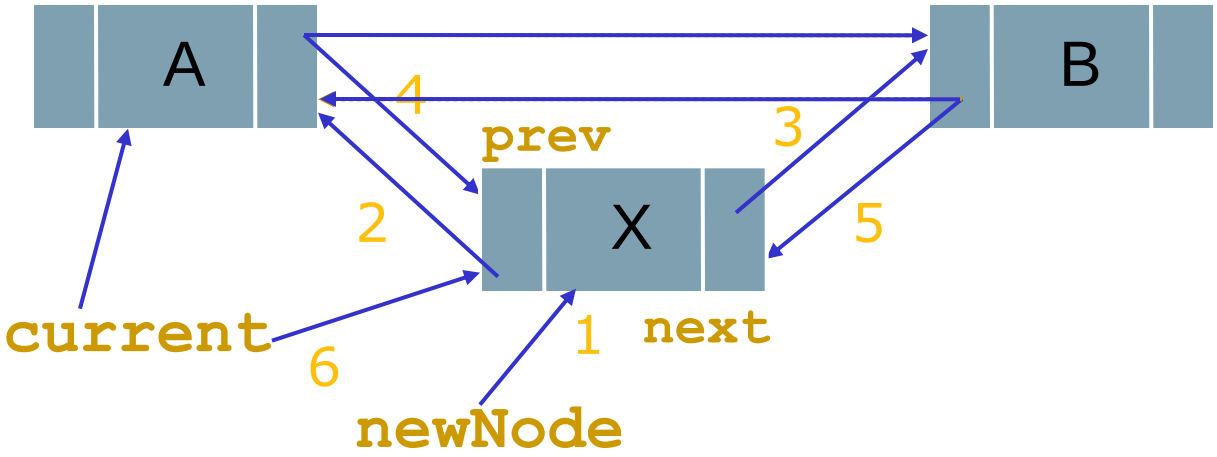


Doubly-linked lists: InsertNext

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



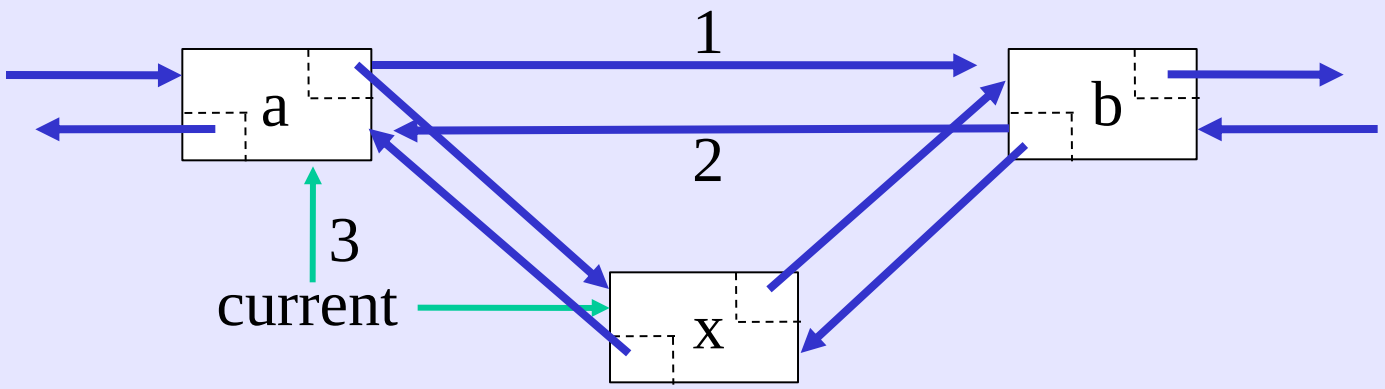
Doubly-linked lists: insertNext




```
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.

Materi Selanjutnya:

- Stacks & Queues – Bab 6, 11, 16