Chapter 3

Linked Lists

Objectives

Discuss the following topics:

- Drawbacks of arrays
- Singly Linked Lists
- Doubly Linked Lists
- Circular Lists
- Lists in `java.util`

Review Array

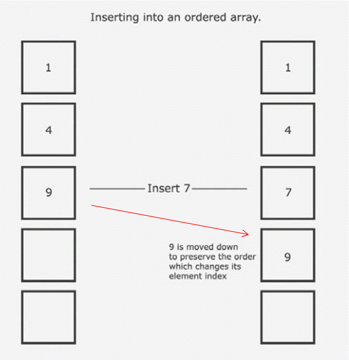
- Array is the most common data storage structure
- Arrays can be used in programs where a list of objects with the same data type is needed.
- Arrays have great use in all kinds of applications, especially games (Lines 98, Bejeweled) and simulations.
- Examples: an array of character models, an array of textures, an array of sounds.

Review Array

- Insert into an ordered array

Insert **x** to the position **pos** of the array **a** containing **n** elements.

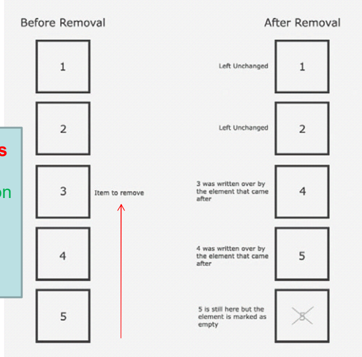
```
// Shift down all elements from the
//position pos
for (int i=n; i>pos; i--) a[i]= a[i-1];
// put x to the position pos
a[pos]=x;
```


Inserting into an ordered array.
1
4
9
Insert 7
1
4
7
9
9 is moved down to preserve the order which changes its element index

Review Array

- Removing an item from an array by copying over it

```
Remove the element at the position pos
of the array a containing n elements.
// Shift up all elements from the //position
pos +1
for (int i=pos; i<n; i++ a[i]= a[i+1];
// Desrease  the number of elements
n = n-1;
```


Before Removal
After Removal
1
2
3
4
5
Item to remove
Left Unchanged
Left Unchanged
3 was written over by the element that came after
4 was written over by the element that came after
5 is still here but the element is marked as empty
1
2
4
5
5

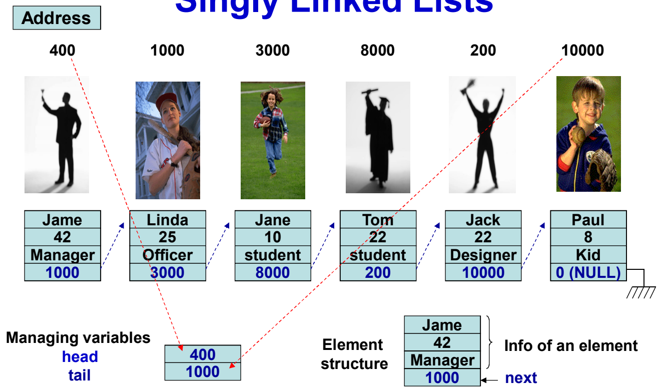
Drawbacks of Arrays

Operation	Evaluation
Store	Fixed, optimal only when the array is full but when the array is full, we can not add extra elements. A solution: Use dynamic arrays → Cost: Copy elements to new storage → O(n)
Insert	Low, O(n) for moving down elements
Remove	Low, O(n) for moving up elements
Search	O(n) – Linear search O(logn) – Binary search

- We need data structures that provide better utilities on operations, store, insert, remove.

➔ Linked lists

Singly Linked Lists

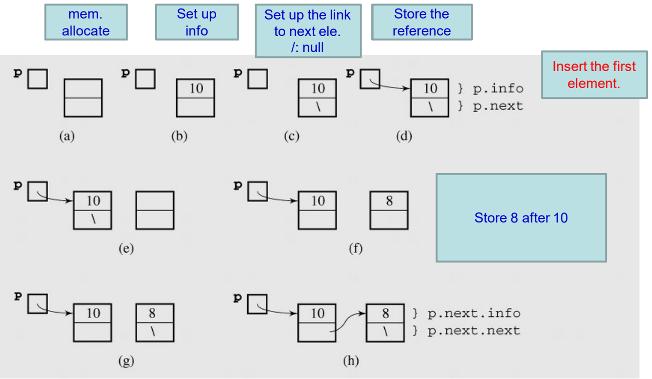


The reference **tail** helps the insert last operation with the complexity of O(1)

Singly Linked Lists

- A **linked structure** is a collection of nodes storing data and links to other nodes
- A **linked list** is a data structure composed of nodes, each node holding some information and a reference to another node in the list
- A **singly linked list** is a node that has a link only to its successor in this sequence

Singly Linked Lists (continued)



mem. allocate

Set up info

Set up the link to next ele. /: null

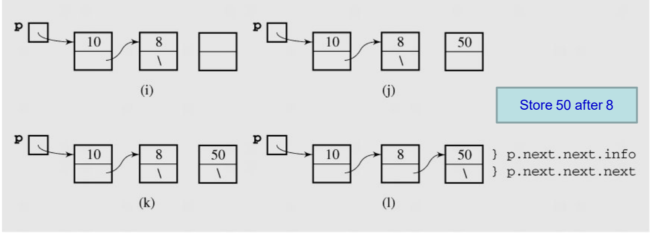
Store the reference

Insert the first element.

Store 8 after 10

Figure 3-1 A singly linked list – page 81

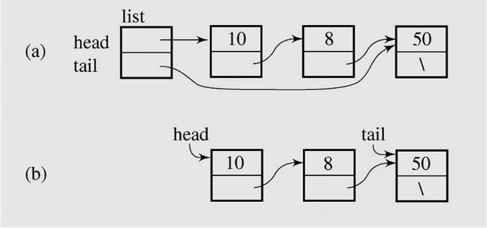
Singly Linked Lists (continued)



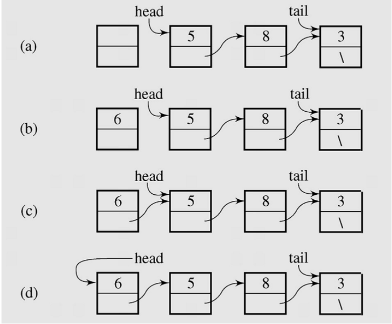
Store 50 after 8

Figure 3-1 A singly linked list (continued) – page 81

A SLL of Integers



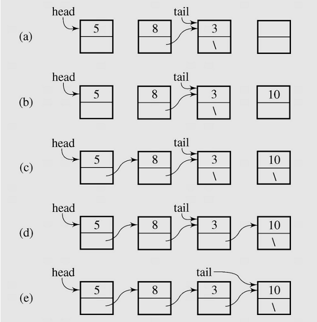
SLL: Insert new node at the beginning



```
// In case of  the list is empty
p = allocate new node
      (p.info= x; p.next= null)
head=tail = p;

// In case of  the list is not empty
p = allocate new node
      (p.info= x; p.next= null);
p.next= head;
head= p;
```

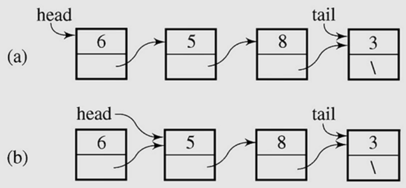
Figure 3-4 Inserting a new node at the beginning of a singly linked list

SLL: Insert new node at the end



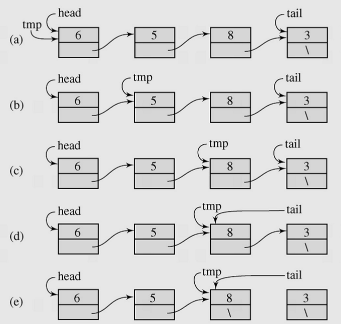
```
// In case of  the list is empty
p = allocate new node
        (p.info= x; p.next= null)
head=tail = p;

// In case of  the list is not empty
p = allocate new node
        (p.info= x; p.next= null);
tail.next= p;
tail= p;
```

SLL: Delete the beginning element



```
// In case of  the list contains
// only 1 element
if (head==tail ) head=tail=null;

// In case of  the list contains
// more than 1 element
   head= head.next;
```

SLL: Delete the last element



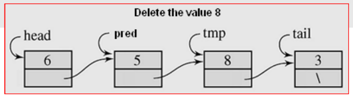
```
// In case of  the list contains only 1 element
if (head==tail ) head=tail=null;

// In case of  the list contains  more than 1 element

    // Move tmp to the element  it is right
    // previous the tail element.
    tmp = head;
    while  (temp.next!=tail)  tmp= tmp.next;
    // tail is tmp now, the old tail element is
    //removed
    tail = tmp;
    taiil.next=null'
```

Figure 3-7 Deleting a node from the end of a singly linked list

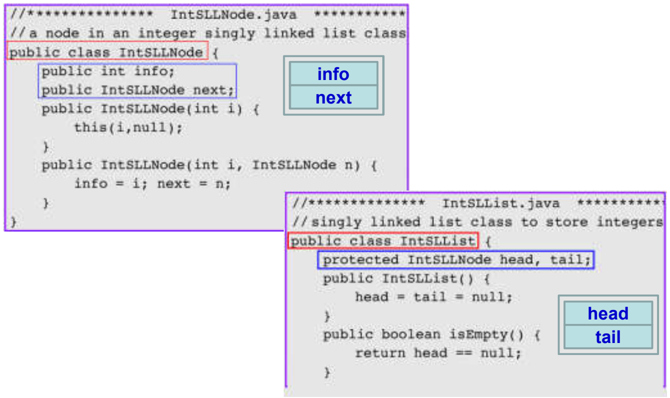
SLL: Delete a known value

```
if (!isEmpty())
    if (head == tail && el == head.info)  // if only one
         head = tail = null;              // node on the list;
    else if (el == head.info) // if more than one node on the
         head = head.next;    // list; and el is in the head node;
    else {                    // if more than one node in the list
         IntSLLNode pred, tmp;// and el is in a non-head node;
         for (pred = head, tmp = head.next;
              tmp != null && tmp.info != el;
              pred = pred.next, tmp = tmp.next);
         if (tmp != null) {   // if el was found;
             pred.next = tmp.next;
             if (tmp == tail) // if el is in the last node;
                tail = pred;
         }
    }
```

Delete the value 8



Singly Linked Lists – A demo. – page 83



```
//****************  IntSLLNode.java  ***********
//a node in an integer singly linked list class
public class IntSLLNode {
    public int info;
    public IntSLLNode next;
    public IntSLLNode(int i) {
        this(i,null);
    }
    public IntSLLNode(int i, IntSLLNode n) {
        info = i; next = n;
    }
}
```

```
//**************  IntSLList.java  **********
//singly linked list class to store integers
public class IntSLList {
    protected IntSLLNode head, tail;
    public IntSLList() {
        head = tail = null;
    }
    public boolean isEmpty() {
        return head == null;
    }
```

Singly Linked Lists (continued)

```
public void addToHead(int el) {
    head = new IntSLLNode(el,head);
    if (tail == null)
        tail = head;
}
public void addToTail(int el) {
    if (!isEmpty()) {
        tail.next = new IntSLLNode(el);
        tail = tail.next;
    }
    else head = tail = new IntSLLNode(el);
}
public int deleteFromHead() { // delete the head and return its info;
    int el = head.info;
    if (head == tail)    // if only one node on the list;
         head = tail = null;
    else head = head.next;
    return el;
```

You can refer to operations in previous slides.

Singly Linked Lists (continued)

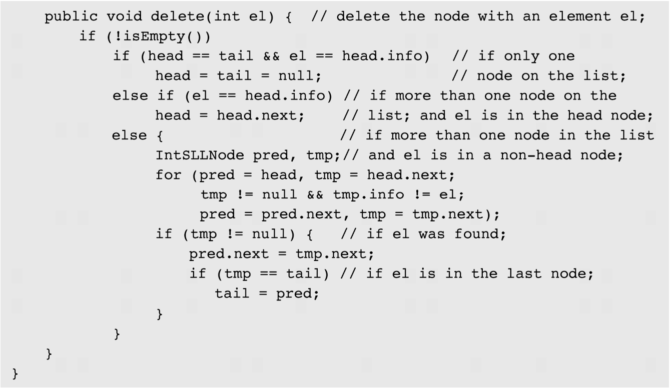
Singly Linked Lists (continued)

```
}
public int deleteFromTail() { // delete the tail and return its info;
    int el = tail.info;
    if (head == tail)       // if only one node on the list;
         head = tail = null;
    else {                  // if more than one node on the list,
         IntSLLNode tmp;    // find the predecessor of tail;
         for (tmp = head; tmp.next != tail; tmp = tmp.next);
         tail = tmp;        // the predecessor of tail becomes tail;
         tail.next = null;
    }
    return el;
}
public void printAll() {
    for (IntSLLNode tmp = head; tmp != null; tmp = tmp.next)
        System.out.print(tmp.info + " ");
}
public boolean isInList(int el) {
    IntSLLNode tmp;
    for (tmp = head; tmp != null && tmp.info != el; tmp = tmp.next);
    return tmp != null;
}
```

You can refer to operations in previous slides.

Singly Linked Lists (continued)

Singly Linked Lists (continued)



```
    public void delete(int el) {  // delete the node with an element el;
        if (!isEmpty())
            if (head == tail && el == head.info)  // if only one
                 head = tail = null;              // node on the list;
            else if (el == head.info) // if more than one node on the
                 head = head.next;    // list; and el is in the head node;
            else {                    // if more than one node in the list
                 IntSLLNode pred, tmp;// and el is in a non-head node;
                 for (pred = head, tmp = head.next;
                      tmp != null && tmp.info != el;
                      pred = pred.next, tmp = tmp.next);
                 if (tmp != null) {   // if el was found;
                     pred.next = tmp.next;
                     if (tmp == tail) // if el is in the last node;
                        tail = pred;
                 }
            }
    }
}
```

Singly Linked Lists (continued)

Overview

- Reference technique in Single List
- Traverser in Single List

LAB 1

- Implement method Delete node has value. Example: Delete(int e)

Lab 2

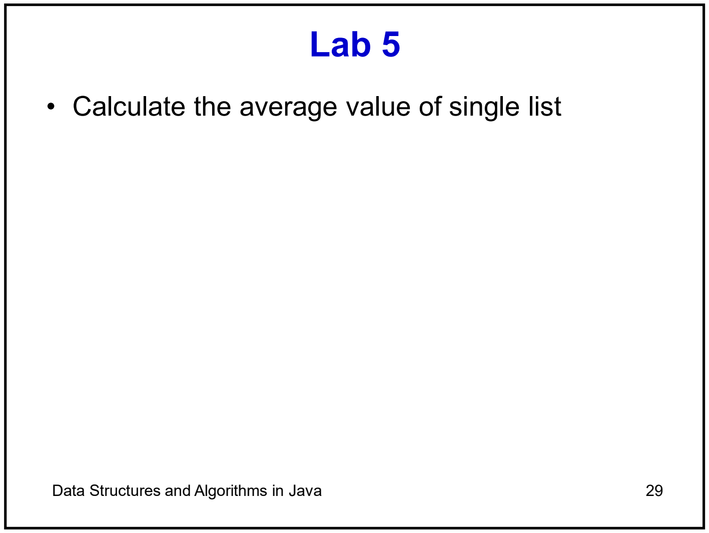
- Implement method Input(int n) to input the n random integers into single list

Lab 3

- Run all your code in main

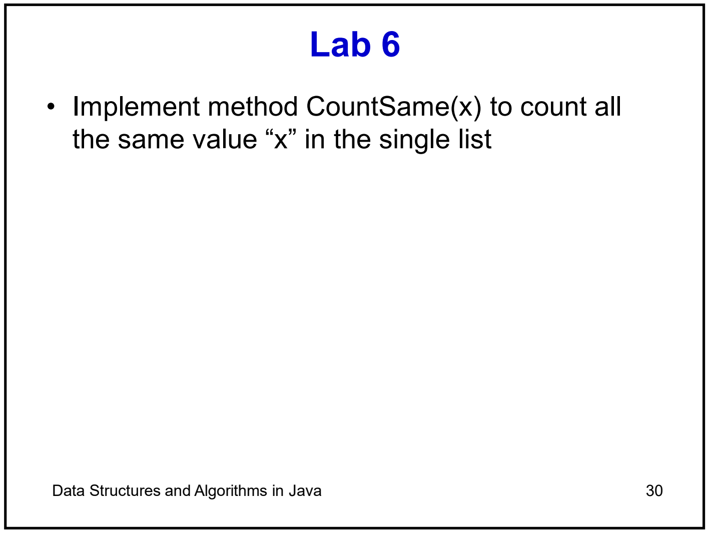
Lab 4

- Implement method CountCurrentElement to count the number of elements in Single list



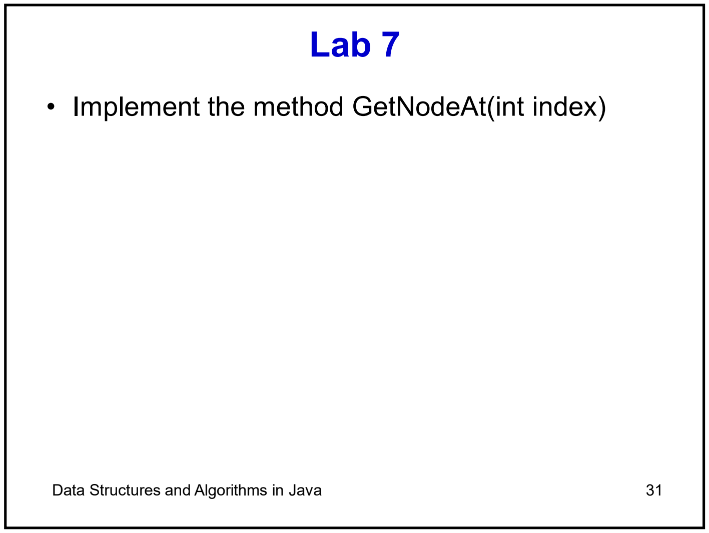
Lab 5

- Calculate the average value of single list



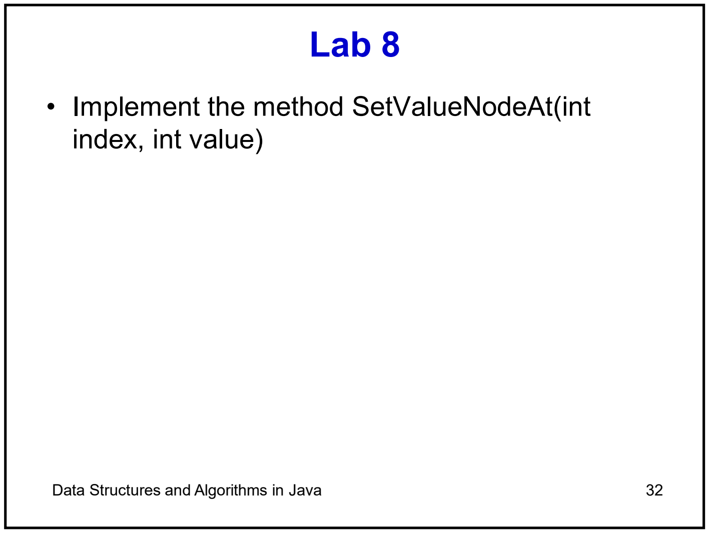
Lab 6

- Implement method CountSame(x) to count all the same value “x” in the single list



Lab 7

- Implement the method GetNodeAt(int index)



Lab 8

- Implement the method SetValueNodeAt(int index, int value)

Lab 9

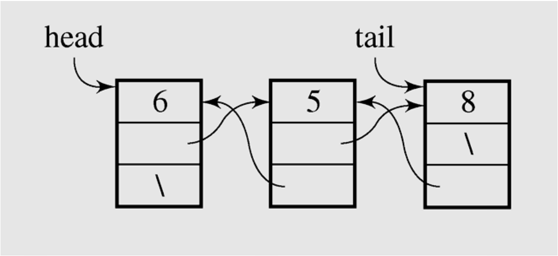
- Implement method RemoveAt(int i) to remove element at index

Lab 10

- Implement the method RemoveAll(int value) to delete all nodes which have info being value

Lab 11

- Describe the data structure following



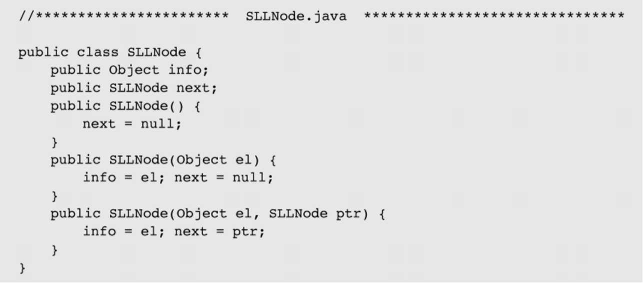
Lab 12

- Design and implement methods by yourself:
  - isEmpty
  - addToTail
  - removeFromTail

HOME_ WORK

- Lab 6,7,8,9,10,11,12

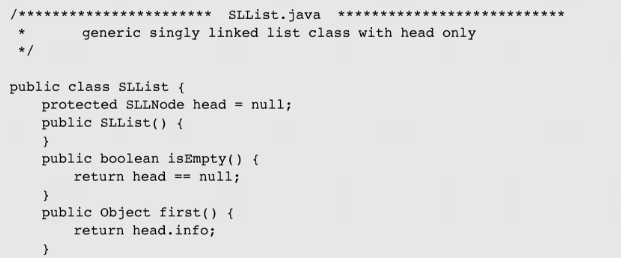
How to implement a generic SLL?



```
//**********************  SLLNode.java  ********************************

public class SLLNode {
    public Object info;
    public SLLNode next;
    public SLLNode() {
        next = null;
    }
    public SLLNode(Object el) {
        info = el; next = null;
    }
    public SLLNode(Object el, SLLNode ptr) {
        info = el; next = ptr;
    }
}
```

Figure 3-9 Implementation of a generic singly linked list

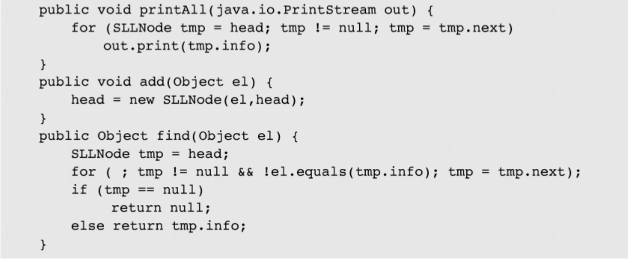
Singly Linked Lists (continued)



```
/**********************  SLList.java  ***************************
 *       generic singly linked list class with head only
 */

public class SLList {
    protected SLLNode head = null;
    public SLList() {
    }
    public boolean isEmpty() {
        return head == null;
    }
    public Object first() {
        return head.info;
    }
```

Figure 3-9 Implementation of a generic singly linked list (continued)

Singly Linked Lists (continued)



```
    public void printAll(java.io.PrintStream out) {
        for (SLLNode tmp = head; tmp != null; tmp = tmp.next)
            out.print(tmp.info);
    }
    public void add(Object el) {
        head = new SLLNode(el,head);
    }
    public Object find(Object el) {
        SLLNode tmp = head;
        for ( ; tmp != null && !el.equals(tmp.info); tmp = tmp.next);
        if (tmp == null)
             return null;
        else return tmp.info;
    }
```

Figure 3-9 Implementation of a generic singly linked list (continued)

Singly Linked Lists (continued)

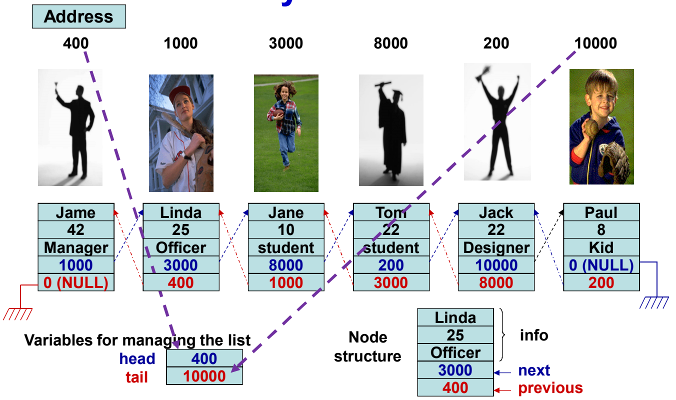
```
    public Object deleteHead() { // remove the head and return its info;
        Object el = head.info;
        head = head.next;
        return el;
    }
    public void delete(Object el) {    // find and remove el;
        if (head != null)              // if nonempty list;
            if (el.equals(head.info)) // if head needs to be removed;
                head = head.next;
            else {
                SLLNode pred = head, tmp = head.next;
                for ( ; tmp != null && !(tmp.info.equals(el));
                       pred = pred.next, tmp = tmp.next);
                if (tmp != null)      // if found
                     pred.next = tmp.next;
            }
    }
}
```

Figure 3-9 Implementation of a generic singly linked list (continued)

LAB

- Implement with list double
- Print all element

Doubly Linked Lists



Address					
400	1000	3000	8000	200	10000
Jame	Linda	Jane	Tom	Jack	Paul
42	25	10	22	22	8
Manager	Officer	student	student	Designer	Kid
1000	3000	8000	200	10000	0 (NULL)
0 (NULL)	400	1000	3000	8000	200

Variables for managing the list

head	400
tail	10000

Node structure

Linda	info
25	
Officer	
3000	next
400	previous

Two direction to access

Doubly Linked Lists

- A **doubly linked list** is when each node in a linked list has two reference fields, one to the successor and one to the predecessor

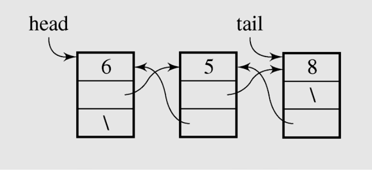
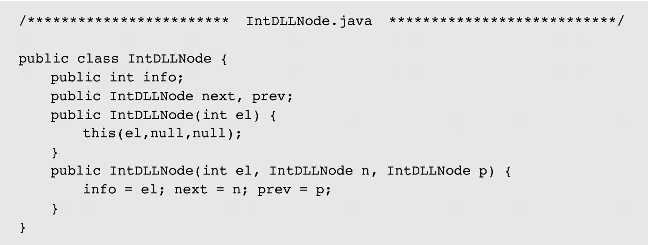


Figure 3-10 A doubly linked list

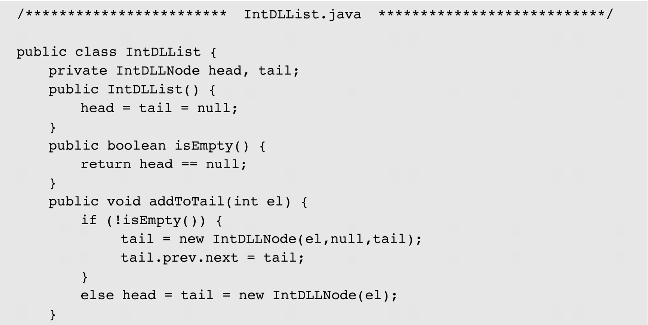
Doubly Linked Lists (continued)



```
/************************  IntDLLNode.java  ***************************/

public class IntDLLNode {
    public int info;
    public IntDLLNode next, prev;
    public IntDLLNode(int el) {
        this(el,null,null);
    }
    public IntDLLNode(int el, IntDLLNode n, IntDLLNode p) {
        info = el; next = n; prev = p;
    }
}
```

Figure 3-11 An implementation of a doubly linked list

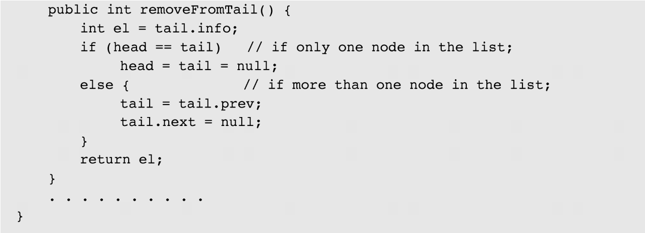
Doubly Linked Lists (continued)



```
/************************  IntDLLList.java  ***************************/

public class IntDLLList {
    private IntDLLNode head, tail;
    public IntDLLList() {
        head = tail = null;
    }
    public boolean isEmpty() {
        return head == null;
    }
    public void addToTail(int el) {
        if (!isEmpty()) {
             tail = new IntDLLNode(el,null,tail);
             tail.prev.next = tail;
        }
        else head = tail = new IntDLLNode(el);
    }
```

Figure 3-11 An implementation of a doubly linked list (continued)

Doubly Linked Lists (continued)



```
    public int removeFromTail() {
        int el = tail.info;
        if (head == tail)   // if only one node in the list;
             head = tail = null;
        else {              // if more than one node in the list;
             tail = tail.prev;
             tail.next = null;
        }
        return el;
    }
    . . . . . . . . . .
}
```

Figure 3-11 An implementation of a doubly linked list (continued)

Doubly Linked Lists (continued)

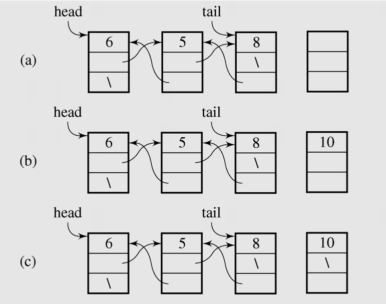


Figure 3-12 Adding a new node at the end of a doubly linked list

Doubly Linked Lists (continued)

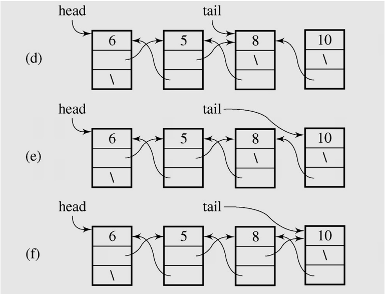


Figure 3-12 Adding a new node at the end of a doubly linked list (continued)

Doubly Linked Lists (continued)

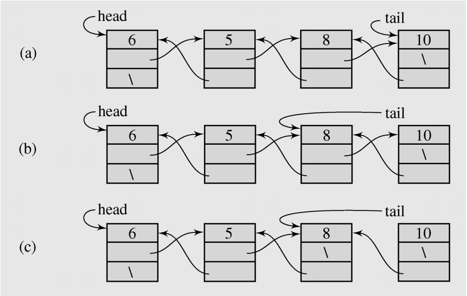
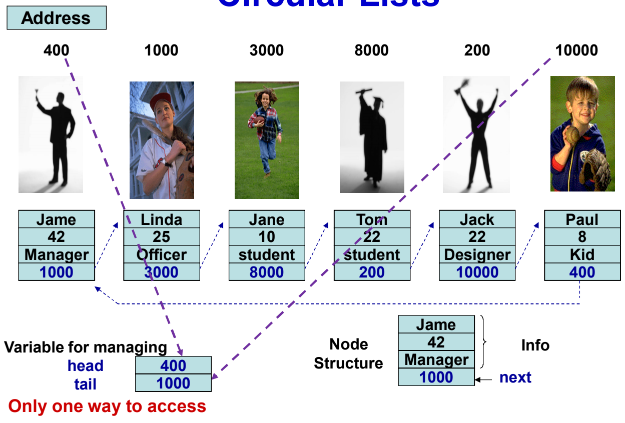


Figure 3-13 Deleting a node from the end of a doubly linked list

Circular Lists



Address

400	1000	3000	8000	200	10000
Jame	Linda	Jane	Tom	Jack	Paul
42	25	10	22	22	8
Manager	Officer	student	student	Designer	Kid
1000	3000	8000	200	10000	400

Variable for managing
head 400
tail 1000

Node Structure
Jame
42
Manager
1000
Info
next

Only one way to access

Circular Lists

- A **circular list** is when nodes form a ring: The list is finite and each node has a successor

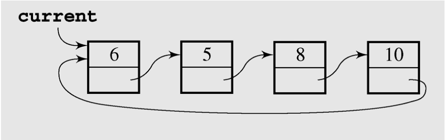


Figure 3-14 A circular singly linked list

Circular Lists (continued)

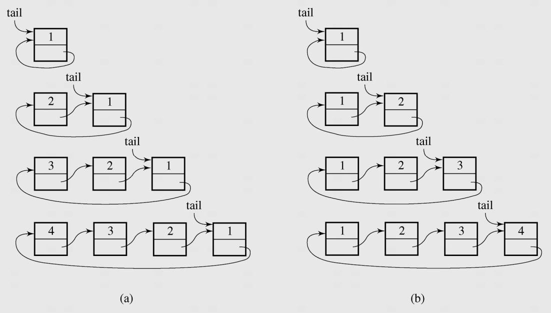


Figure 3-15 Inserting nodes at the front of a circular singly linked list (a) and at its end (b)

Circular Lists (continued)

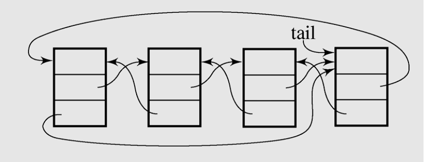
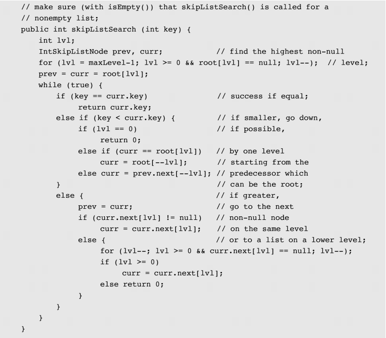


Figure 3-16 A circular doubly linked list

Skip Lists (continued)



```
// make sure (with isEmpty()) that skipListSearch() is called for a
// nonempty list;
public int skipListSearch (int key) {
    int lvl;
    IntSkipListNode prev, curr;            // find the highest non-null
    for (lvl = maxLevel-1; lvl >= 0 && root[lvl] == null; lvl--);  // level;
    prev = curr = root[lvl];
    while (true) {
        if (key == curr.key)               // success if equal;
             return curr.key;
        else if (key < curr.key) {         // if smaller, go down,
             if (lvl == 0)                 // if possible,
                  return 0;
             else if (curr == root[lvl])   // by one level
                  curr = root[--lvl];      // starting from the
             else curr = prev.next[--lvl]; // predecessor which
        }                                  // can be the root;
        else {                             // if greater,
             prev = curr;                  // go to the next
             if (curr.next[lvl] != null)   // non-null node
                  curr = curr.next[lvl];   // on the same level
             else {                        // or to a list on a lower level;
                  for (lvl--; lvl >= 0 && curr.next[lvl] == null; lvl--);
                  if (lvl >= 0)
                       curr = curr.next[lvl];
                  else return 0;
             }
        }
    }
}
```

Figure 3-18 An implementation of a skip list (continued)

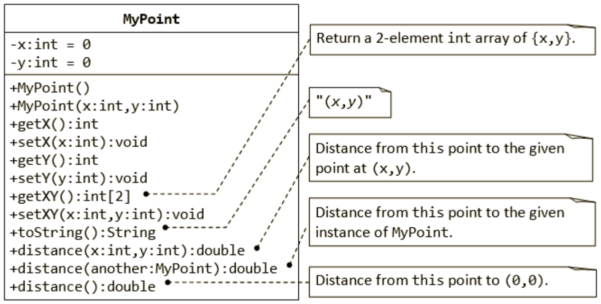
Vector & ArrayList

- API: the two classes are very similar.
- Synchronization:
  - – Vectors are synchronized -> thread safe & make them executing slow.
  - – ArrayLists are unsynchronized -> not thread safe & faster than Vectors.
- Data growth:
  - – Both hold onto their contents using an Array.
  - – Different to doubling the size of its array.

LAB (*** Project 1)

- Download code Generic DLL and study them
- Applied this library to build the application

Class MyPoint



```
MyPoint
-x:int = 0
-y:int = 0
+MyPoint()
+MyPoint(x:int,y:int)
+getX():int
+setX(x:int):void
+getY():int
+setY(y:int):void
+getXY():int[2]
+setXY(x:int,y:int):void
+toString():String
+distance(x:int,y:int):double
+distance(another:MyPoint):double
+distance():double
```

Return a 2-element int array of {x,y}.

"(x,y)"

Distance from this point to the given point at (x,y).

Distance from this point to the given instance of MyPoint.

Distance from this point to (0,0).

Class Arc

- This class has some methods
  - getHeadPoint
  - getTailPoint
  - printArc
  - getNumberPoint
  - getLengthArc
  - setClosetArc
  - checkClosetArc
  - separateCycleArc(int i)
  - …..