

Lectures 10.1 - 10.2

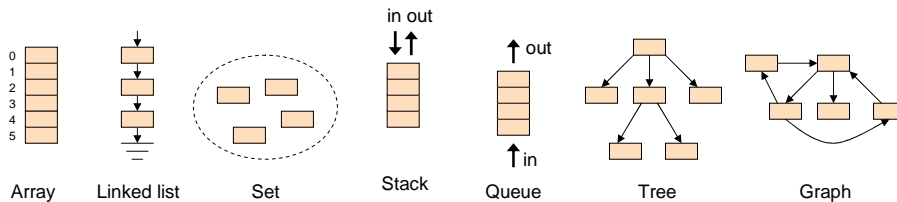
Collections

Outline

- Introduction
 - Data Structures
 - Collections
- ➔ ■ Set
 - Abstraction
 - Implementation
- Stack
- Dynamic data structures
- Linked lists
- Iterators
- Application example

Data structures

- In computer science, a "data structure" is a collection of objects that has a well-defined architecture and behavior
- Some common data structures:

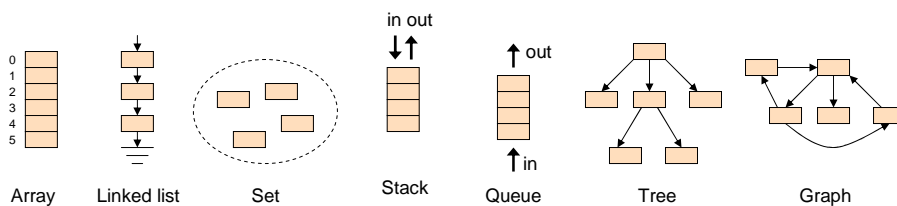


These data structures come in many variants:

- Arrays: 1-, 2-, n-dimensional, ...
- Linked lists: singly-connected, doubly connected, ...
- Sets: mutable, immutable, ...
- Tree: binary, n-ary, ...
- Graph: directional, undirectional, ...

Data structures

- As usual, we distinguish between the data structure's abstraction and implementation

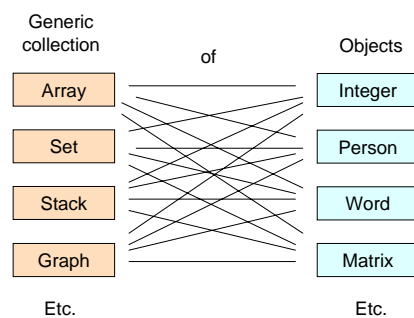


- Abstraction: each data structure is defined by a set of behaviors, or operations, that it supports
- Implementation: each data structure can be implemented using two major "tools":
 - Arrays
 - Linked lists

Collections

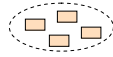
- In OO terminology, a "collection" is an object that facilitates access to, and management of, other objects
 - Add an object
 - Find an object
 - Delete an object
 - Sort the objects
 - Iterate through the objects
 - Etc.
- The objects can be of the same type ("homogenous collection") or of multiple type ("heterogeneous collection")
- The focus is not on the individual objects, but on the collection
- A collection can describe any architecture of objects. In particular, in Java, data structures are implemented using collections
- Data structure: a theoretical CS concept
- Collection: a programming concept.

Collections, data structures, ADT's ...



- A generic collection can also be viewed as an ADT, since it defines collection-specific data and operations
- The focus here is not on the individual objects, but rather on the ADT
- As we will see, the same ADT abstraction can be implemented in very different ways.

Set



- The set's data conforms to these rules:
 - The order of the items is insignificant: { 2 3 4 } = { 4 2 3 }
 - Duplicates are not allowed
- The set's behavior consists of these operations:
 - Create an empty set
 - Insert an item to the set
 - Delete an item from the set
 - Check if a given item is in the set
 - Check if the set is empty
 - Print the set
 - Perform set operations:
 - Intersection
 - Union
 - Difference
 - etc.

Client code

```
public class SetDemo {  
  
    public static void main (String args[]) {  
        Set s = new Set();  
        s.insert(2);  
        s.insert(1);  
        s.insert(5);  
        s.insert(1);  
        s.insert(3);  
        s.insert(4);  
        System.out.println(s);  
        s.delete(5);  
        s.delete(7);  
        s.delete(2);  
        System.out.println(s);  
    }  
}
```

```
cmd C:\WINDOWS\system32\cmd.exe  
D:\demo\Sets>java SetDemo  
{ 2 1 5 3 4 }  
{ 3 1 4 }
```

Set abstraction

```
public class Set {  
    // method signatures only  
  
    // Constructs a new set  
    Set()  
  
    // Checks if this set contains x  
    boolean contains (Int x)  
  
    // Inserts x to this set  
    void insert(Int x)  
  
    // Delete x from this set  
    void delete(Int x)  
  
    // Inserts another set into this set  
    void insertSet(Set s)  
  
    // Returns the intersection of this set and another set  
    Set intersection(Set s)  
  
    // Returns the union of this set and another set  
    Set union(Set s)  
    ...  
}
```

Client code

```
public class SetDemo {  
  
    public static void main (String args[]) {  
        Set s = new Set();  
        s.insert(2);  
        s.insert(1);  
        s.insert(5);  
        s.insert(1);  
        s.insert(3);  
        s.insert(4);  
        System.out.println(s);  
        s.delete(5);  
        s.delete(7);  
        s.delete(2);  
        System.out.println(s);  
    }  
}
```

We will now show an
array-based
implementation of
the Set abstraction

Set implementation, using an array

```
public class Set {
    // The elements of this set, in no particular order
    private int[] elements;
    // The number of elements in this set
    private int size;
    // The default maximum size of the set
    private static final
        int DEFAULT_LENGTH = 100;

    // Constructors
    public Set(int maxSize) {
        elements = new int[maxSize];
        size = 0;
    }

    public Set() {
        this(DEFAULT_LENGTH);
    }

    // More Set methods follow.
}
```

```
public class SetDemo {

    public static void main (String args[]) {
        Set s = new Set();
        s.insert(2);
        s.insert(1);
        // Etc.
    }
}
```

- This implementation is not efficient
- Later we will show a more efficient implementation of the same abstraction.

Outline

- Introduction
 - Data Structures
 - Collections
- ➔ ■ Set
 - Abstraction
 - Implementation
- Stack
- Dynamic data structures
- Linked lists
- Iterators
- Application example

Set implementation (cont.)

```
public class Set {
    private int[] elements;
    private int size;
    ...
    public boolean contains (int x) {
        for(int j=0 ; j<size ; j++)
            if (elements[j] == x)
                return true;
        return false;
    }

    public void insert(int x) {
        if (!contains(x))
            elements[size++] = x;
    }

    public void delete(int x) {
        for(int j=0 ; j<size ; j++)
            if (elements[j] == x) {
                elements[j] = elements[--size];
                return;
            }
    }
    ...
}
```

Client code

```
public class SetDemo {

    public static void main (String args[]) {
        Set s = new Set();
        s.insert(2);
        s.insert(1);
        s.insert(5);
        s.insert(1);
        s.insert(3);
        s.insert(4);
        System.out.println(s);
        s.delete(5);
        s.delete(7);
        s.delete(2);
        System.out.println(s);
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
```

```
D:\demo\Sets>java SetDemo
< 2 1 5 3 4 >
< 3 1 4 >
```

Insert implementation, Take 2

```
public class Set {

    private int[] elements;
    private int size;
    ...

    public void safeInsert(int x) {
        if (!contains(x)) {
            if (size == elements.length)
                resize();
            elements[size++] = x;
        }
    }

    private void resize() {
        int[] temp = new int[2 * elements.length];
        System.arraycopy(elements, 0, temp, 0, elements.length);
        elements = temp;
    }
    ...
}
```

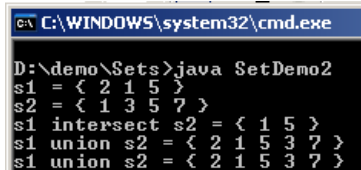
Handling
overflow

Set operations

```
public class Set {
    private int[] elements;
    private int size;
    ...
    public Set Intersection(Set s) {
        Set intersection = new Set(size);
        for(int j=0 ; j<size ; j++)
            if (s.contains(elements[j]))
                intersection.insert(elements[j]);
        return intersection;
    }
    public void InsertSet(Set s) {
        for (int j = 0 ; j < s.size ; j++)
            insert(s.elements[j]);
    }
    public Set union(Set s) {
        Set union = new Set(size + s.size);
        union.insertSet(this);
        union.insertSet(s);
        return union;
    }
    public static Set union(Set s, Set t) {
        return s.union(t);
    }
    ...
}
```

Client code

```
public class SetDemo2 {
    public static void main (String args[]) {
        Set s1 = new Set();
        s1.insert(2); s1.insert(1); s1.insert(5);
        System.out.println("s1 = " + s1);
        Set s2 = new Set();
        s2.insert(1); s2.insert(3); s2.insert(5);
        s2.insert(7);
        System.out.println("s2 = " + s2);
        System.out.println("s1 intersect s2 = " +
            s1.intersection(s2));
        System.out.println("s1 union s2 = " +
            s1.union(s2));
        System.out.println("s1 union s2 = " +
            Set.union(s1,s2));
    }
}
```



```
C:\WINDOWS\system32\cmd.exe
D:\demo\Sets>java SetDemo2
s1 = { 2 1 5 }
s2 = { 1 3 5 7 }
s1 intersect s2 = { 1 5 }
s1 union s2 = { 2 1 5 3 7 }
s1 union s2 = { 2 1 5 3 7 }
```

Immutable set

Immutable object: an object that cannot be changed once it is constructed

- We will show an immutable set abstraction and implementation
- The only operations that this abstraction supports: `contains(x)` and `toString(x)`
- We will show a `contains(x)` implementation that runs in $O(\log_2 N)$ time, N being the set's size.

```
public class ImmutableSet {
    // Method signatures only

    // Constructor
    public ImmutableSet(int[] elements)

    // Set membership
    public boolean contains(int x)

    // Textual representation
    public String toString()
}
```

Client code

```
public class ImmutableSetDemo {
    public static void main (String args[]) {
        int[] data = {9, 7, 5, 4, 2, 1, 3};
        ImmutableSet immSet = new ImmutableSet(data);
        System.out.println(immSet);
        ...
    }
}
```

Immutable set implementation: constructor

```
public class ImmutableSet {  
  
    // The elements of this immutable set,  
    // ordered from low to high  
    private int[] elements;  
  
    // Constructs an immutable set from a given array.  
    // The array does not have to be ordered.  
    // It is assumed that it contains no duplicates.  
    public ImmutableSet(int[] elements) {  
        this.elements = new int[elements.length];  
        for(int j=0 ; j < elements.length ; j++)  
            this.elements[j] = elements[j];  
        MergeSort.mergeSort(this.elements);  
    }  
    ...  
}
```

```
Client code  
public class ImmutableSetDemo {  
    public static void main (String args[]) {  
        int[] data = {9, 7, 5, 4, 2, 1, 3};  
        ImmutableSet immSet = new ImmutableSet(data);  
        System.out.println(immSet);  
        ...  
    }  
}
```

Searching a set using binary search

```
public class ImmutableSet {  
    ...  
    // Returns true if x is in this set, false otherwise.  
    public boolean contains (int x) {  
        return contains(x, 0, elements.length - 1);  
    }  
  
    // Finds x in the array implementing this set using binary search  
    private boolean contains(int x, int low, int high) {  
        if (low > high)  
            return false;  
        int med = (low + high) / 2;  
        if (x == elements[med])  
            return true;  
        else if (x < elements[med])  
            return contains(x, low, med-1);  
        else  
            return contains(x, med+1, high);  
    }  
    ...  
}
```

```
cmd C:\WINDOWS\system32\cmd.exe  
D:\deno\Sets>java ImmutableSetDemo  
< 1 2 3 4 5 7 9 >  
7 is in the set: true  
11 is in the set: false
```

```
Client code  
public class ImmutableSetDemo {  
    public static void main (String args[]) {  
        int[] data = {9, 7, 5, 4, 2, 1, 3};  
  
        ImmutableSet immSet = new ImmutableSet(data);  
        System.out.println(immSet);  
  
        System.out.println("7 is in the set: " +  
            immSet.contains(7));  
        System.out.println("11 is in the set: " +  
            immSet.contains(11));  
    }  
}
```

Aside: recursive vs. iterative implementations of binary search

```
public class ImmutableSet {
    ...
    // Returns true if x is in this set, false otherwise.
    public boolean contains (int x) {
        return contains(x, 0, elements.length - 1);
    }
    // Finds x in the array implementing this set using binary search
    private boolean contains(int x, int low, int high) {
        if (low > high)
            return false;
        int med = (low + high) / 2;
        if (x == elements[med])
            return true;
        else if (x < elements[med])
            return contains(x, low, med-1);
        else
            return contains(x, med+1, high);
    }
    ...
}
```

Recursive

```
public class ImmutableSet {
    ...
    public boolean contains (int x) {
        int low = 0;
        int high = elements.length - 1;
        while (low <= high) {
            int med = (low + high) / 2;
            if (x == elements[med])
                return true;
            if (x < elements[med])
                high = med - 1;
            else
                low = med + 1;
        }
        return false;
    }
    ...
}
```

Iterative

Introduction to Computer Science, Shimon Schocken

Outline

- Introduction
 - Data Structures
 - Collections
- Set
 - Abstraction
 - Implementation
- ➔ ■ Stack
- Dynamic data structures
- Linked lists
- Iterators
- Application example

Introduction to Computer Science, Shimon Schocken

slide 18

Stacks

A stack holds an ordered collection of items with a single entry / exit point

- ❑ Items are pushed (inserted) onto the stack top
- ❑ Items are popped (removed) from the stack top

```
Create an empty stack { }
Push 5 { 5 }
Push 7 { 5 7 }
Push 5 { 5 7 5 }
Push 8 { 5 7 5 8 }
Push 2 { 5 7 5 8 2 }
Pop (returns 2) { 5 7 5 8 }
Pop (returns 8) { 5 7 5 }
Push 4 { 5 7 5 4 }
Pop (returns 4) { 5 7 5 }
```



- A LIFO (*Last In, First Out*) setting
- A classical data structure with many applications in computer science.

Stack abstraction

```
// Stack of integers
public class Stack {
    // Method signatures only

    // Creates an empty stack
    Stack()

    // Inserts an item to the stack's top
    void push(int x)

    // Removes an item to from the stack's top
    int pop()

    // Checks is the stack is empty
    boolean isEmpty()

    // Textual stack description
    String toString()
}
```

- All operations run in $O(1)$ time.

```
public class StackDemo {
    public static void main (String args[])
    {
        int x;
        Stack stack = new Stack();
        stack.push(10);    stack.push(20);
        stack.push(30);    stack.push(40);
        stack.push(50);    stack.push(60);
        System.out.println(stack);
        x = stack.pop();
        System.out.println("Popped: " + x);
        x = stack.pop();
        System.out.println("Popped: " + x);
        System.out.println(stack);
        stack.push(stack.pop() + stack.pop());
        System.out.println(stack);
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
```

```
D:\demo\Stack>java StackDemo
10 20 30 40 50 60
Popped: 60
Popped: 50
10 20 30 40
10 20 70
```

Stack implementation

```
public class Stack {
    private int[] elements;
    private int top;
    private final static int DEFAULT_MAX_SIZE = 10;

    public Stack(int maxSize) {
        elements = new int[maxSize];
        top = 0;
    }

    public Stack() { this(DEFAULT_MAX_SIZE); }

    public void push(int x) { elements[top++] = x; }

    public int pop() { return elements[--top]; }

    public boolean isEmpty() { return top == 0; }

    public String toString() {
        String s = "";
        for(int j=0; j<top; j++)
            s = s + elements[j] + " ";
        return s;
    }
}
```

Overflow / underflow
are not handled

```
public class StackDemo {
    public static void main (String args[])
    {
        int x;
        Stack stack = new Stack();
        stack.push(10);    stack.push(20);
        stack.push(30);    stack.push(40);
        stack.push(50);    stack.push(60);
        System.out.println(stack);
        x = stack.pop();
        System.out.println("Popped: " + x);
        x = stack.pop();
        System.out.println("Popped: " + x);
        System.out.println(stack);
        stack.push(stack.pop() + stack.pop());
        System.out.println(stack);
    }
}
```

C:\WINDOWS\system32\cmd.exe

```
D:\demo\Stack>java StackDemo
10 20 30 40 50 60
Popped: 60
Popped: 50
10 20 30 40
10 20 70
```

Application example: compiling arithmetic expressions

A stack can be used as a convenient workspace for performing arithmetic operations and storing their results:

```
Create an empty stack { }
push 8 { 8 }
push 3 { 8 3 }
push 2 { 8 3 2 }
add { 8 5 }
mul t { 40 }
pop (returns 40) { }
push 10 { 10 }
push 4 { 10 4 }
di v { 2 }
neg { -2 }
Etc.
```

Stack arithmetic: the rules of the game

Each arithmetic operation pops its operands from the stack, computes a function on them, and pushes the result onto the stack.

Arithmetic notations:

- Infix: $(5 + 3) * 8$
- Prefix: $* + 5 3 8$
- Postfix: $5 3 + 8 *$

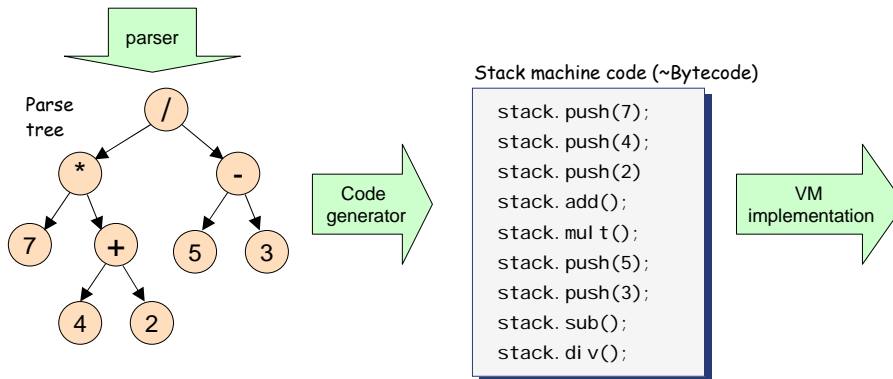
General observation, unrelated to stacks

- Stack arithmetic provides a natural way to implement postfix arithmetic
- To evaluate an arithmetic expression, we can:
 1. Rewrite the expression in postfix notation
 2. Use a stack to compute its value.

Application example: compiling arithmetic expressions

High level code

```
7 * (4 + 2) / (5 - 3)
```



The Java compiler handles expressions in two main stages:

1. Parser: creates a parse tree from the Java expression
2. Code generator: performs a depth-first traversal of the parse tree, generating bytecode on the fly

Example of a VM implementation

Virtual Machine Emulator (1.4b3) - C:\examples\Pong

File View Run Help

Slow Fast Program flow View Screen Decimal

Program

```
label Math.multiplyNHL.E
34 push local.2
35 push argument.1
36 i
37 mul
38 if-goto Math.multiplyNHL.E
39 push local.3
40 push static.0
41 add
42 pop pointer.1
43 push that.0
44 push argument.1
45 and
46 push constant.0
47 if
```

Static

0	2064
1	2048

Local

0	0
1	0
2	0
3	2
4	0

Argument

0	332
1	24

This

0	432
1	229
2	48
3	7
4	1

That

0	4
1	8

Temp

0	312
1	2

Stack

2	2064
---	------

Call Stack

- Sys.in
- Main.main
- PongGame.run
- Ball.move
- Screen.drawRectangle
- Math.multiply

Global Stack

297	332
300	24
301	2975
302	289
303	200
304	3325
305	0
306	0
307	0
308	0
309	3
310	0
311	3
312	2064
313	2064

RAM

SP	0	332
LCI	1	2064
ARG	2	299
THIS	3	3325
THAT	4	2064
Temp0	5	312
Temp1	6	0
Temp2	7	0
Temp3	8	0
Temp4	9	0
Temp5	10	0
Temp6	11	0
Temp7	12	0
P13	13	317
P14	14	842

SCORE: 1

Pong game in action

Outline

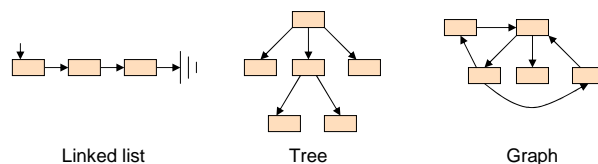
- Introduction
 - Data Structures
 - Collections
- Set
 - Abstraction
 - Implementation
- Stack
- ➔ ■ Dynamic data structures
- Linked lists
- Iterators
- Application example

Dynamic data structures

Quite often, an object has to point to other objects of the same type:

- A web page pointing to the web pages that it mentions
- A person pointing to the persons who are his friends
- A country pointing to other countries with which it trades
- An atom pointing to other atoms in the same molecule
- A word pointing to the next word in the text
- Etc.

Such relationships can be managed effectively using linked data structures such as



- Unlike arrays, which have fixed size and architecture, linked data structures are sometimes called "dynamic", meaning that their size changes dynamically to meet the application's needs.

Application example: word processing

The diagram illustrates three steps of word processing using a linked list structure and an editor window:

- Initial State:** A linked list contains the words "It", "was", "the", "best", "of", and "times". A pointer 'p' points to the first node "It". The editor window displays "It was the best of times".
- Delete the first two words:** The first two nodes ("It" and "was") are removed from the list. The pointer 'p' now points to the third node "the". The editor window displays "the best of times".
- Replace "best" with "worst":** The node containing "best" is replaced with a new node containing "worst". The editor window displays "the worst of times".

Introduction to Computer Science, Shimon Schocken slide 27

Application example: employee-boss relationship abstraction

```
public class Employee {
    // Constructs an employee
    Employee (String name)

    // Assigns a boss to this employee
    void setBoss(Employee boss)

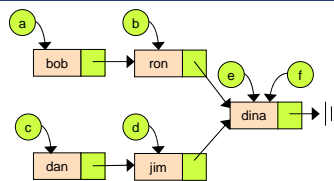
    // Returns the name of this employee
    String getName()

    // Returns the boss of this employee
    Employee getBoss()
}
```

```
public class EmployeeDemo {
    public static void main (String args[])
    {
        Employee a = new Employee ("bob");
        Employee b = new Employee ("ron");
        Employee c = new Employee ("dan");
        Employee d = new Employee ("jim");
        Employee e = new Employee ("dina");
        a.setBoss(b);
        b.setBoss(e);
        c.setBoss(d);
        c.setBoss(d);
        d.setBoss(e);
        e.setBoss(null);
        Employee f = a.getBoss().getBoss();
    }
}
```

Each employee has a boss, who is an employee, except for one employee, who has no boss.

- Pitfall: Not clear where this data structure "starts".



Application example: employee-boss relationship implementation

```
public class Employee {
    private String name;
    private Employee boss;

    public Employee (String name) {
        this.name = name;
    }

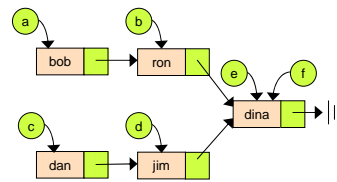
    public void setBoss(Employee boss) {
        this.boss = boss;
    }

    public String getName() {
        return name;
    }

    public Employee getBoss() {
        return boss;
    }
}
```

```
public class EmployeeDemo {
    public static void main (String args[])
    {
        Employee a = new Employee ("bob");
        Employee b = new Employee ("ron");
        Employee c = new Employee ("dan");
        Employee d = new Employee ("jim");
        Employee e = new Employee ("dina");
        a.setBoss(b);
        b.setBoss(e);
        c.setBoss(d);
        d.setBoss(e);
        e.setBoss(null);
        Employee f = a.getBoss().getBoss();
    }
}
```

Each employee has a boss, who is an employee, except for one employee, who has no boss.



Outline

- Introduction
 - Data Structures
 - Collections
- Set
 - Abstraction
 - Implementation
- Stack
- Dynamic data structures
- ➔ ■ Linked lists
- Iterators
- Application example

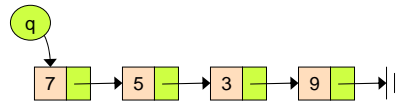
Linked list

```
public class List {
    private int data;
    private List next;

    public List(int data, List next) {
        this.data = data;
        this.next = next;
    }

    public int getData() {
        return data;
    }

    public List getNext() {
        return next;
    }
}
```



Each linked object consists of two things:

- Head: data
- Tail: a list

```
public class ListDemo {
    public static void main (String args[]) {
        List q;
        q = new List(9, null); // { 9 {} }
        q = new List(3, q);   // { 3 { 9 {} } }
        q = new List(5, q);   // { 5 { 3 { 9 {} } } }
        q = new List(7, q);   // { 7 { 5 { 3 { 9 {} } } } }
    }
}
```

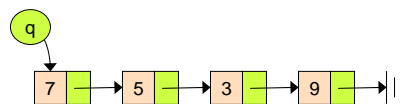
Linked list

```
public class List {
    private int data;
    private List next;

    public List(int data, List next) {
        this.data = data;
        this.next = next;
    }

    public int getData() {
        return data;
    }

    public List getNext() {
        return next;
    }
}
```



A typical linked list abstraction will include more list processing operations:

- Add an item
- Delete an item
- Search an item
- Replace an item

Every one of these operations can have several variants

But, for the purpose of the example that follows, the minimal implementation on the left is sufficient.

Set class: linked list implementation

```
public class Set {
    private List elements;
    public Set() {
        elements = null;
    }
    public boolean contains(Int x) {
        for (List c = elements ; c != null ; c = c.getNext())
            if (c.getData() == x)
                return true;
        return false;
    }
    public void insert(Int x) {
        if (!contains(x))
            elements = new List(x, elements);
    }
    ...
}
```

```
public class SetDemo3 {
    public static void main (String args[]) {
        Set set = new Set();
        set.insert(2);
        set.insert(7);
        set.insert(5);
        set.insert(7);
        System.out.println(set);
    }
}
```

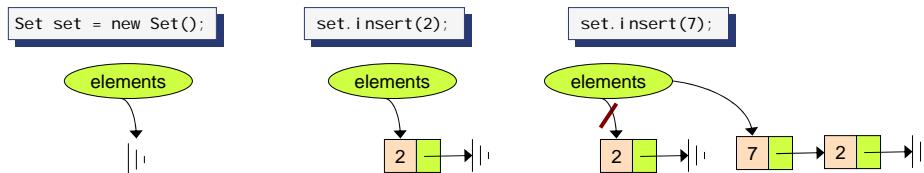
```
C:\WINDOWS\system32\cmd.exe
D:\demo\Sets>java SetDemo3
< 2 7 5 >
```

Usage simulation

```
public class Set {
    private List elements;
    public Set() { elements = null; }
    public boolean contains(Int x) {
        for (List c = elements ; c != null ; c = c.getNext())
            if (c.getData() == x)
                return true;
        return false;
    }
    public void insert(Int x) {
        if (!contains(x))
            elements = new List(x, elements);
    }
    ...
}
```

- In this particular implantation, new nodes are added at the beginning of the list
- Insignificant, since we are implementation a *set*.

Client code simulation:



Using a list iterator

```
public class Set2 {
    private List elements;

    // Constructor and Insert methods as in Set class before

    public boolean contains(int x) {
        for (ListIterator l = new ListIterator(elements); l.hasNext(); )
            if (l.nextItem() == x)
                return true;
        return false;
    }

    public String toString() {
        StringBuffer result = new StringBuffer("");
        for (ListIterator l = new ListIterator(elements); l.hasNext(); )
            result.append(" " + l.nextItem() + " ");
        result.append("\n");
        return result.toString();
    }
}
```

How does
ListIterator
work?

```
public class SetDemo {
    public static void main (String args[]) {
        Set2 set = new Set2();
        set.insert(2);
        set.insert(1);
        set.insert(5);
        set.insert(1);
        set.insert(3);
        set.insert(4);
        System.out.println(set);
    }
}
```

```
CA C:\WINDOWS\system32\cmd.exe
```

```
D:\demo>List>java SetDemo
{ 4 3 5 1 2 }
```

Introduction to Computer Science, Shimon Schocken

ListIterator

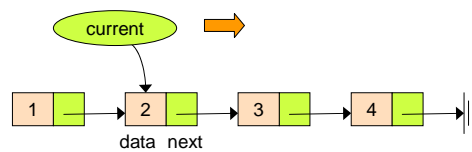
```
public class ListIterator {
    private List current;

    public ListIterator(List list) {
        current = list;
    }

    public boolean hasNext() {
        return current != null;
    }

    public int nextItem() {
        int item = current.getData();
        current = current.getNext();
        return item;
    }
}
```

- The caller wants to iterate a list. It uses the services of ListIterator
- ListIterator uses the services of List, which knows how to return the *data* and *next* values of every list item.



```
public class Set2 {
    private List elements;
    public boolean contains(int x) {
        for (ListIterator l = new ListIterator(elements); l.hasNext(); )
            if (l.nextItem() == x) return true;
        return false;
    }
}
```

Introduction to Computer Science, Shimon Schocken

slide 36

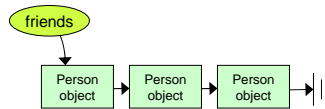
Outline

- Introduction
 - Data Structures
 - Collections
- Set
 - Abstraction
 - Implementation
- Stack
- Dynamic data structures
- Linked lists
- Iterators
- ➔ ■ Application example

Application example

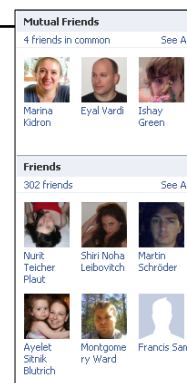
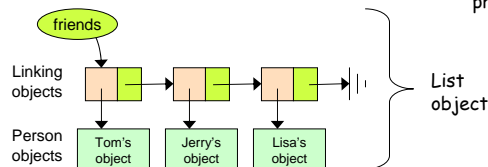
FaceBook: A collection of persons. Each person has 0 or more friends, who are persons. We wish to be able to add and delete friends easily and efficiently.

- Desired abstraction:



- Problem: The Person object does not have built-in mechanisms to link to other Persons. The linking logic seems to belong to the list abstraction, not to the atoms abstraction

- Typical implementation:



Part of the FaceBook profile of a friend

The FriendsList class

```
// A linked list of friends (Person objects)
public class FriendsList {
    private FriendNode list;

    // Constructs a list of friends
    public FriendsList() {
        list = null;
    }

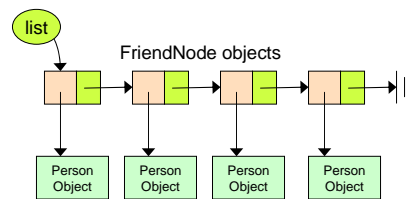
    // Adds a friend to the (end of the) list
    public void add (Person friend) {}

    // Returns this list of friends as a string.
    public String toString () {}

    // An inner class representing a friend node.
    private class FriendNode {
        public Person friend;
        public FriendNode next;
        // Constructs a friend node
        public FriendNode (Person friend) {
            this.friend = friend;
            next = null;
        }
    }
}
```

Inner class: A private inner class is accessible only from the code of its outer class

Other implementations of this particular application are possible.



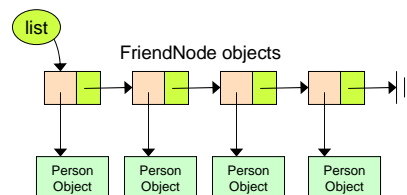
Adding friends to the friends list

```
public class FriendsList {
    private FriendNode list;

    // Adds a friend to the end of the list
    public void add (Person friend) {
        FriendNode node = new FriendNode(friend);
        FriendNode p;
        if (list == null)
            list = node;
        else {
            p = list;
            while (p.next != null)
                p = p.next;
            p.next = node;
        }
    }

    // An inner class representing a friend node.
    private class FriendNode {
        public Person friend;
        public FriendNode next;
        // Constructs a friend node
        public FriendNode (Person friend) {
            this.friend = friend;
            next = null;
        }
    }
}
```

- Q: Should we add new objects to the list's beginning, or end?
- A: Depends on the application's needs
- If we add objects to the list's end, how can we improve performance?

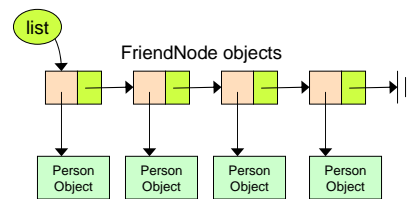


Writing the contents of the friends list

```
public class FriendsList {
    private FriendNode list;

    // Returns this list as a string.
    public String toString () {
        String result = "";
        FriendNode p = list;
        while (p != null) {
            result += p.friend + "\n";
            p = p.next;
        }
        return result;
    }

    // An inner class representing a friend node.
    private class FriendNode {
        public Person friend;
        public FriendNode next;
        // Constructs a friend node
        public FriendNode (Person friend) {
            this.friend = friend;
            next = null;
        }
    }
}
```



Usage demo

```
// Represents a linked list of Person objects
public class FriendsList {

    // Constructs a list of friends
    public FriendsList()

    // Adds a friend to the (end of the) list
    public void add (Person friend)

    // Returns this list of friends as a string.
    public String toString ()
}
```

FriendsList
API

```
// Represents a person
public class Person {

    // Constructs a new person
    public Person (String name,
                  String email) {

    // Returns this person as a string
    public String toString () {

    // Other Person methods follow.
    }
```

Person API

```
public class FriendsListDemo {
    public static void main (String args[]) {
        Person p1 = new Person("Tom", "tom@ibm.com");
        Person p2 = new Person("Jerry", "jerry@ucla.edu");
        Person p3 = new Person("Lisa", "lisa@pbs.org");
        FriendsList flist = new FriendsList();
        flist.add(p1);
        flist.add(p2);
        flist.add(p3);
        System.out.println(flist);
    }
}
```

Client code

```
C:\WINDOWS\system32\cmd.exe
D:\demo>java FriendsListDemo
Tom tom@ibm.com
Jerry jerry@ucla.edu
Lisa lisa@pbs.org
```

- Client's perspective: a friends list is a collection of Person objects
- All the linked list details are hidden from the client.

On the importance of separating abstraction and implementation

- As long as the ADT delivers its promised operations, clients don't have to care about its implementation
- The abstraction / implementation separation helps ...
 - Manage complexity
 - Hide unnecessary details
 - Enable changes to the implementation without touching the abstraction
- OO design is perfectly suited to support the abstraction / implementation separation because encapsulation is a fundamental OO principle.