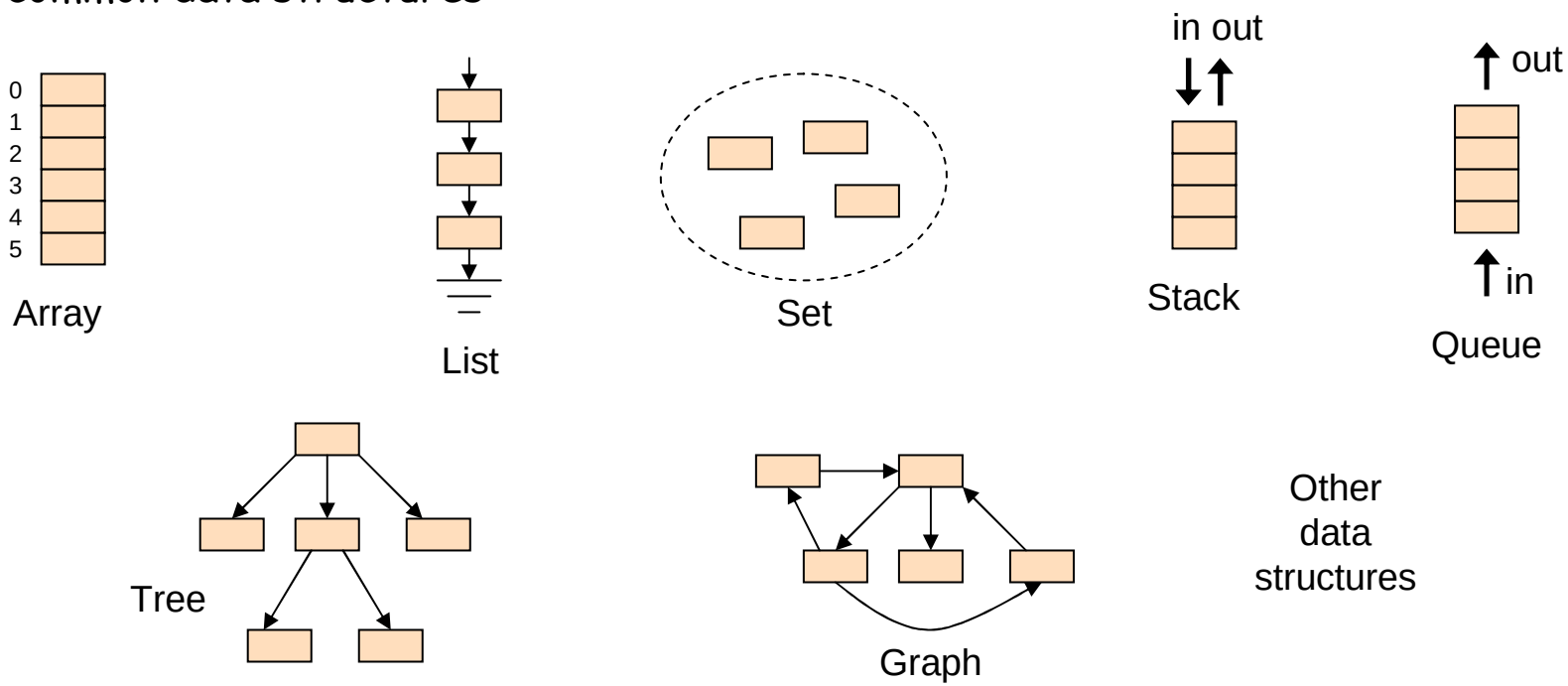
Lectures 9-1, 9-2

Collections

Data structures / collections

In computer science, a "data structure" is a collection of items characterized by an architecture and a behavior.

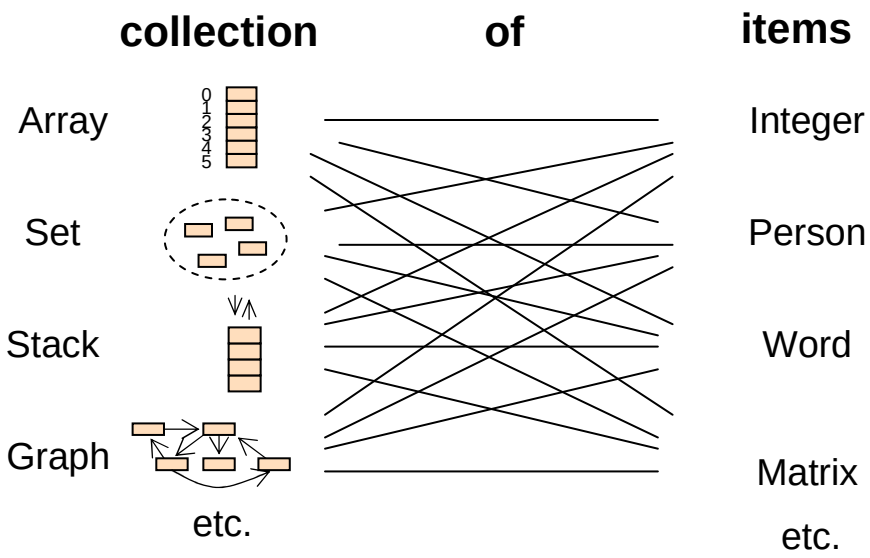
Some common data structures:



In the Java terminology, data structures are sometimes called *collections*

The Java *collection framework:* a library of classes that implement commonly used data structures.

The items of a collection can be objects of any type



The items can be objects of the same type ("homogenous collection")
or of multiple types ("heterogeneous collection")

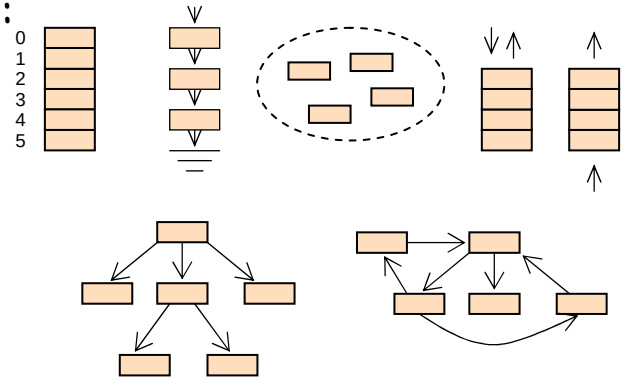
But, when we talk about collections we don't care much about the individual items;
rather, we focus on the collection as a whole

Possible source of confusion:

- The items are objects
- The collection itself is an object.

Typical operations

Collection = an abstraction that supports certain operations:



- Find an item
- Add an item
- Delete an item
- Iterate through the items
- Etc.

Different collections perform different subsets of these operations, with different degrees of efficiency and cost

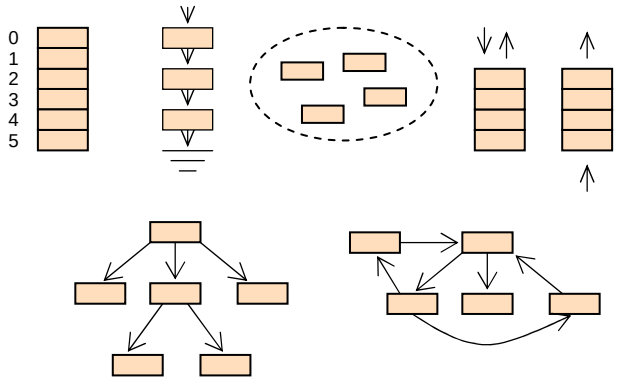
Thus, different collections are suitable for different application needs

<u>Collections come in many variants:</u>

- Arrays: 1-,2-, n-dimensional, …
- Linked lists: singly-connected, doubly connected, …
- Trees: binary, n-ary, …
- Graphs: directional, undirectional, ...

The abstraction / implementation interplay

As usual, we distinguish between
abstraction and *implementation*



<u>Abstraction:</u> the various operations that the collection supports (a functional, client-oriented view)

<u>Implementation:</u> how the collection supports these operations (a behind the scene, developer-oriented view)

<u>Collections are implemented using two main techniques:</u>

- Arrays
- Linked lists.

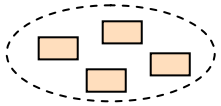
Outline

- Data structures / collections
- Set
- Stack
- Lists
- Hashing

<u>Methodology</u>

- Describe the collection (abstraction)
- Illustrate applications that use it
- Show how the collection can be realized (implementation).

Set



Rules of the game:

- The order of the items is insignificant: `{2, 3, 4} = {4, 2, 3}`
- Duplicates are not allowed

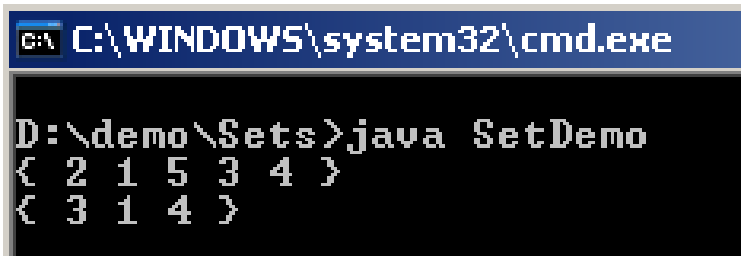
The set's operations:

- Create an empty set
- Insert an item
- Delete an item
- Check if a given item exists in the set
- Check if the set is empty
- Iterate through the set
- Perform set-oriented operations:
  - Intersection
  - Union
  - Difference
  - etc.

For simplicity, we focus on sets of integers.

Client code

```
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);
```



Set abstraction

```
public class Set {

  // Constructs a set
  public Set ()

  // Checks if this set contains x
  public boolean contains (int x)

  // Inserts x to this set
  public void insert (int x)

  // Deletes x from this set
  public void delete (int x)

  // Inserts another set into this set
  public void insertSet (Set s)

  // Returns the intersection of this
  // set and another set
  public Set intersection (Set s)

  // Returns the union of this set and another set
  public Set union (Set s)

  // Returns a textual rep. of this set
  public String toString ()

  ...
}
```

Client code

```
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 }
```

Set implementation, using an array

```
public class Set {

    private int[] elements;   // The elements of this set, in no particular order
    private int size;         // the actual number of array items used by this set

    private static final int DEFAULT_SIZE = 100;  // The default maximum size of this set


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

    public Set () {
        this(DEFAULT_SIZE);
    }

    // More Set methods follow.
}
```

Set implementation (cont.)

```
public class Set {
    private int[] elements;
    private int size;
    ...

    public boolean contains (int x) {
        for (int e : elements)
            if (e == x)
                return true;
        return false;
    }

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

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

- An inefficient implementation
- We'll improve later.

Set implementation (cont.)

```
public class Set {

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

  public void insert (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;
  }
  ...

}
```

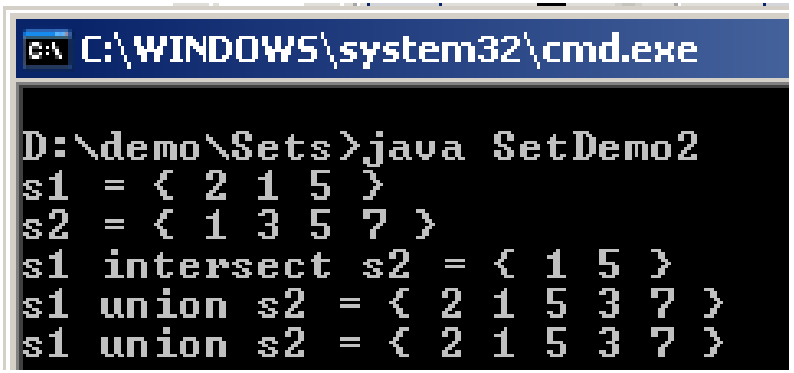
- This version of `insert` handles overflow
- It makes the set grow as much as needed to accommodate new items.

Set implementation (cont.)

```
public class Set {
   private int[] elements;
   private int size;
   ...
   public Set intersection(Set s) {
      Set intersection = new Set(size);
      for (int e : elements)
         if (s.contains(e))
            intersection.insert(e);
      return intersection;
   }
   public void insertSet (Set s) {
      for (int e : s)
         insert(e);
   }
   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

```
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));
```



Immutable set

- We now discuss an immutable set variant that supports only two operations: `contains(x)` and `toString()`
- The `contains(x)` implementation will run in $O(\log_2 N)$ time, N being the set's size.

```
public class ImmutableSet {

  // Constructor
  public ImmutableSet (int[] elements)

  // Checks if this set contains x
  public boolean contains (int x)

  // Returns a textual rep. of this set
  public String toString()
}
```

Client code

```
int[] data = {9, 7, 5, 4, 2, 1, 3};
ImmutableSet s = new ImmutableSet(data);
System.out.println(s);
```

Immutable set implementation

```
// An immutable set representation
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];
     System.arraycopy(elements, 0, this.elements, 0, elements.length);
     Arrays.sort(this.elements);
  }
  ...

}
```

Client code

```
int[] data = {9, 7, 5, 4, 2, 1, 3};
ImmutableSet s = new ImmutableSet(data);
System.out.println(s);
```

Immutable set implementation: fast find using binary search

```
public class ImmutableSet {
  ...
  // Checks if x is in this set.
  public boolean contains (int x) {
     return contains(x, 0, elements.length - 1);
  }

// Finds x in 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);
  }
  ...
}
```

C:\WINDOWS\system32\cmd.exe

```
D:\demo\Sets>java ImmutableSetDemo
{ 1 2 3 4 5 7 9 }
7 is in the set: true
11 is in the set: false
```

Client code

```
int[] data = {9, 7, 5, 4, 2, 1, 3};

ImmutableSet s = new ImmutableSet(data);
System.out.println(s);

System.out.println("7 is in the set: " +
                   s.contains(7));
System.out.println("11 is in the set: " +
                   s.contains(11));
```

Aside: recursive vs. iterative implementations

Recursive

```
public class ImmutableSet {
  ...
  // Checks if x is in this set.
  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);
  }
  ...
}
```

```
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

Outline

- Data structures / collections
- Set
- Stack
- Lists
- Hashing

Stack

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

Stack = a classical data structure with many applications in computer science.

Stack abstraction

```
// Stack of integers
public class Stack {

   // Creates an empty stack
   public Stack ()

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

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

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

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

Client code

```
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);
int 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

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

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

  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

Client code

```java
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);
int 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
```

All operations run in O(1) time.

Application example: stack arithmetic

```
create an empty stack  { }
push 8                 { 8 }
push 3                 { 8 3 }
push 2                 { 8 3 2 }
add                    { 8 5 }
mult                   { 40 }
pop (returns 40)       { }
push 10                { 10 }
push 4                 { 10 4 }
div                    { 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.

Application example: compiling arithmetic expressions

Arithmetic notations:

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

The stack abstraction provides a natural way to implement postfix arithmetic.

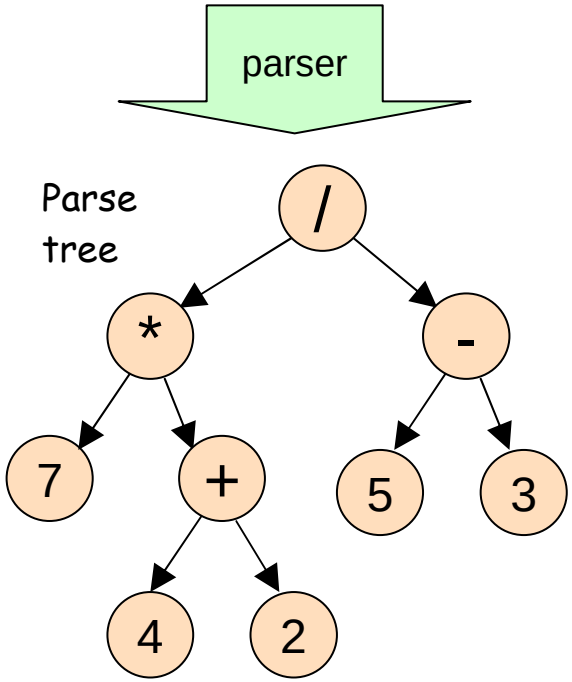
To evaluate a given arithmetic expression, we:

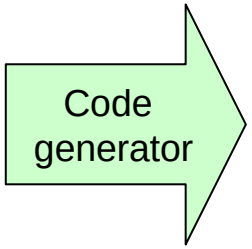
1. Rewrite the expression in postfix notation
2. Use a stack to compute its value.

Application example: compiling arithmetic expressions

High level code

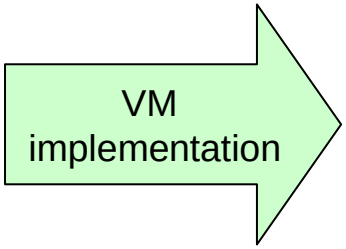
```
7 * (4 + 2) / (5 - 3)
```





Stack machine code (~Bytecode)

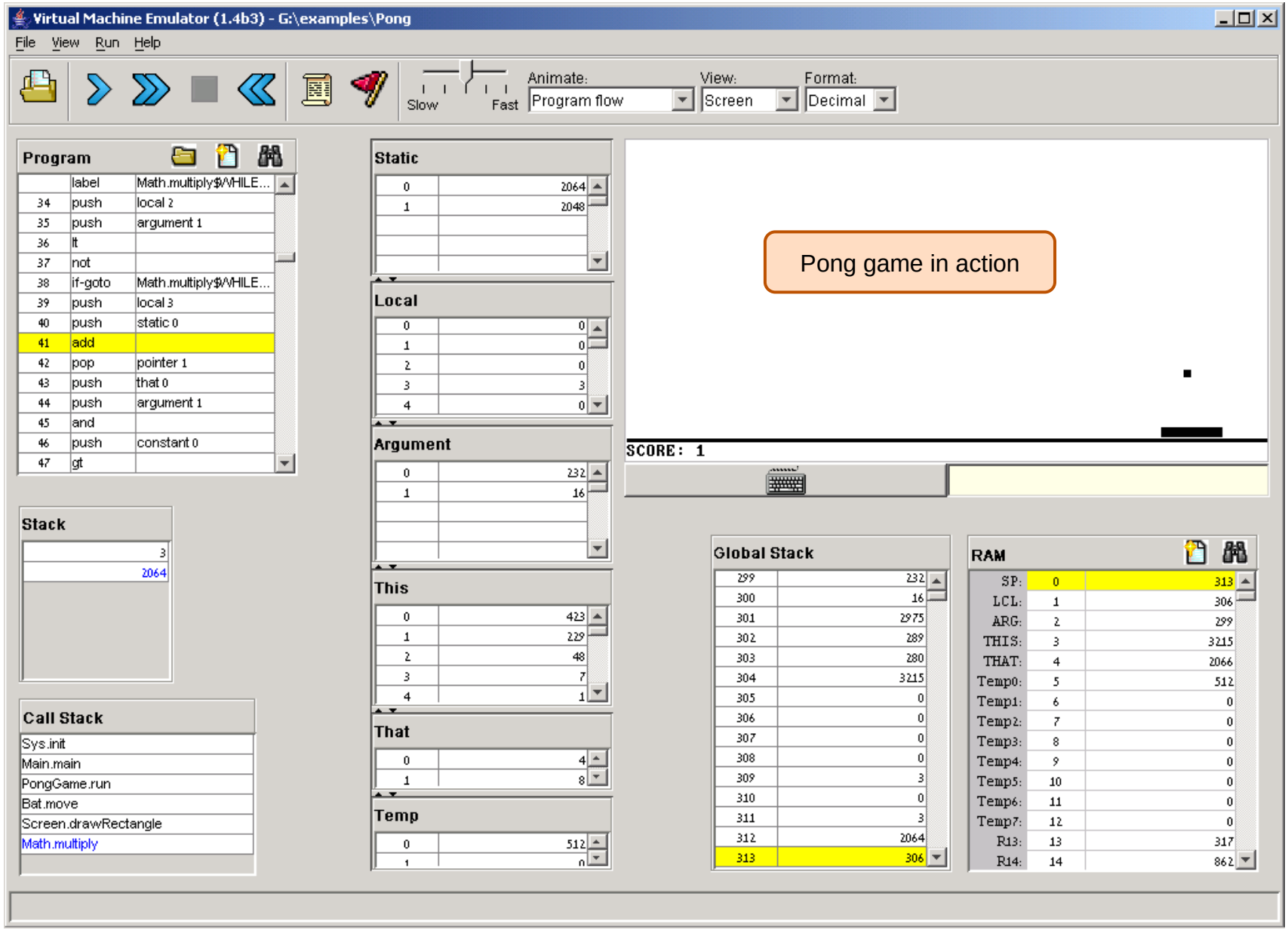
```
stack.push(7);
stack.push(4);
stack.push(2)
stack.add();
stack.mult();
stack.push(5);
stack.push(3);
stack.sub();
stack.div();
```



<u>How the Java compiler handles arithmetic expressions:</u>

1. Parser: takes a high-level expression (input) and creates a parse tree (output)
2. Code generator: takes a parse tree (input) and creates postfix stack code (output)
   Technique: depth-first traversal of the parse tree, generating code on the fly.

Example of a VM implementation



Outline

- Data structures / collections
- Set
- Stack
- Lists
- Hashing

Lists

Architecture

A list is a sequence of items, e.g. (12, 4, 5, 61, 17, 3)

Each item has a position, starting from 1. So, 5 is the 3rd item in this list

Items in the list are said to be *successors* and *predecessors* of each other

Operations:

- Find the location of an item: find(61) returns 4
- Insert an item: insert(x,3) turns the list into (12, 4, x, 5, 61, 17, 3)
- Remove an item: remove(2) turns the list into (12, x, 5, 61, 17, 3)
- PrintList
- MakeEmpty
- Other similar operations, as the situation requires; The exact nature and naming of these operations varies from one application to another.

Implementation

A list can be implemented in two main ways:

- Array
- Linked list

Array implementation

- Simple: the list is stored in a sufficiently large array
- Scanning the list can be done in time O(N) which is as good as can be expected
- Finding an item takes O(N) since the list is not sorted
- Insertions / deletions are expensive, since space must be arranged to make room or close gaps to accommodate the change. This requires copying items, which is O(N).

Linked list implementation

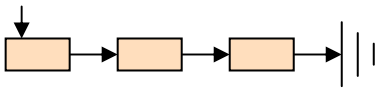
- Instead of storing the list in a contiguous array, we put the items in different places in memory and facilitate direct access from each item to its successor
- The result: a more efficient and elegant implementation.

Linked lists

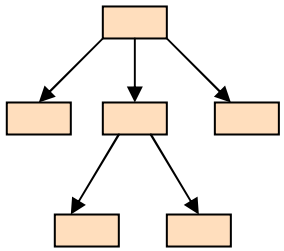
Quite often, an object has to point to other objects of the same type:

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

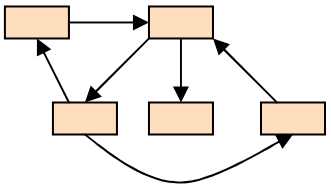
Such relationships can be managed effectively using <u>dynamic data structures</u> such as:



List

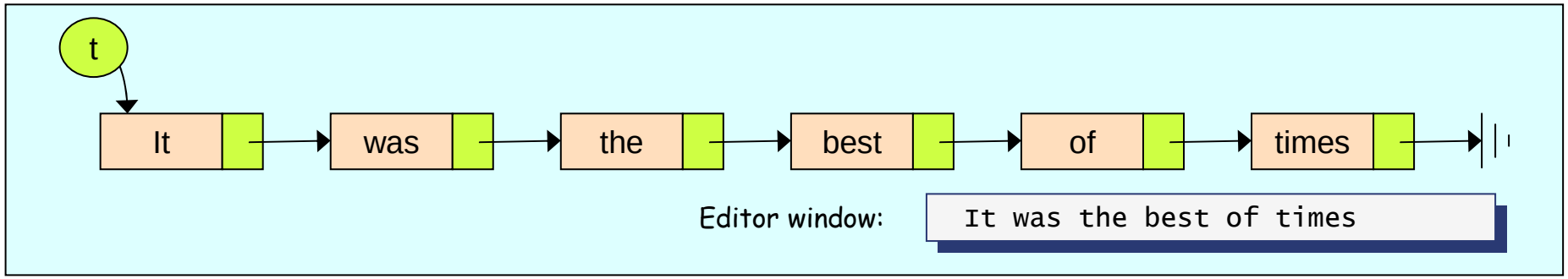


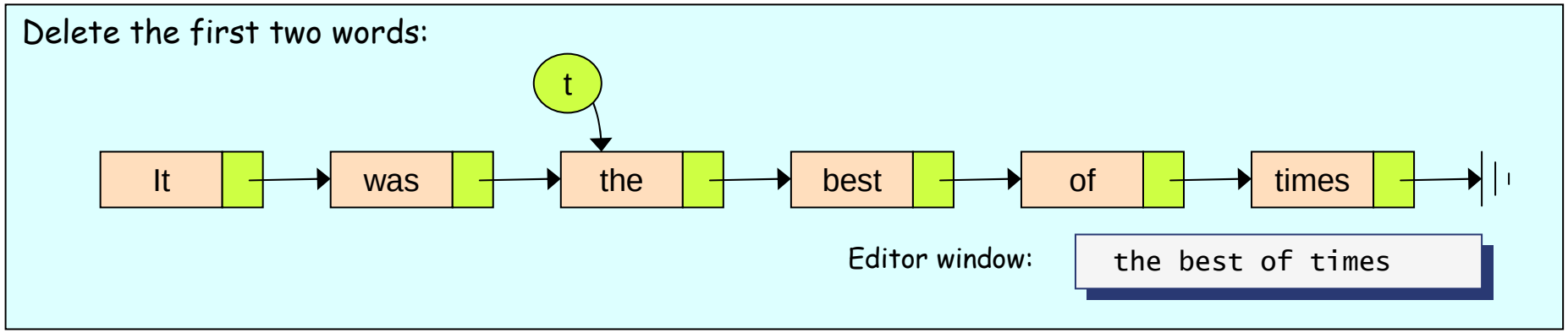
Tree

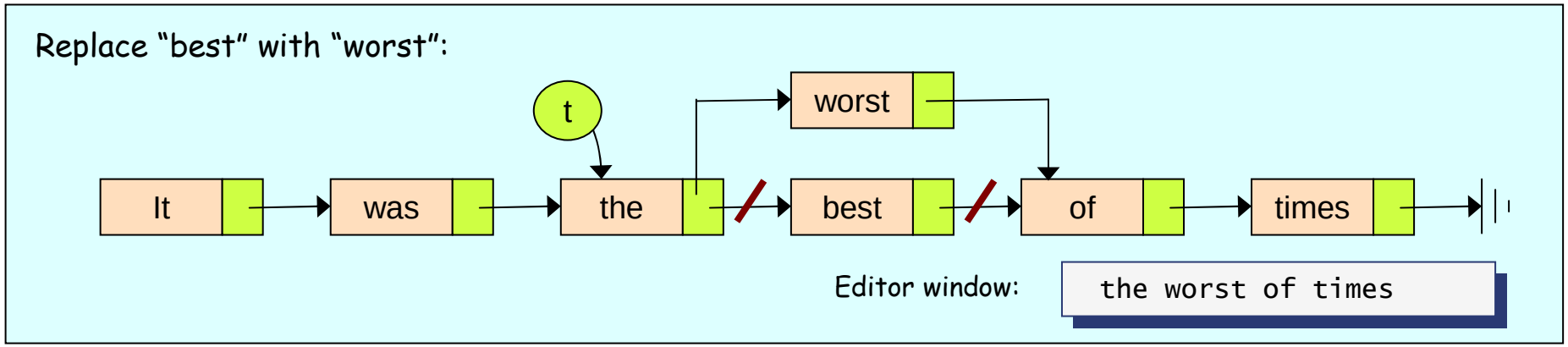


Graph

Linked list application example: word processing

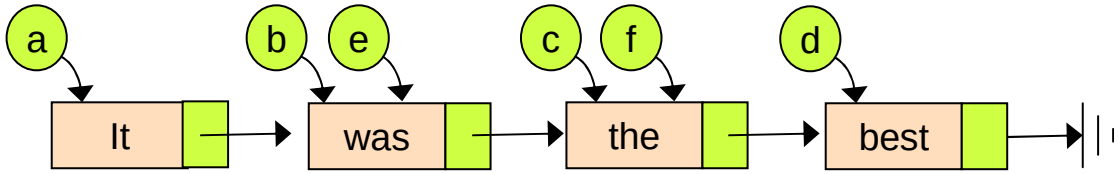






Linked list application example: word processing

Specification: A word consists of data (the word itself) and a reference to the next word.



Word abstraction / API

```
public class Word {

  // Constructs a word
  public Word (String data)

  // Assigns the next word to this word
  public void setNext (Word w)

  // Returns the value of this word
  public String getData ()

  // Returns the next word
  public Word next ()
}
```

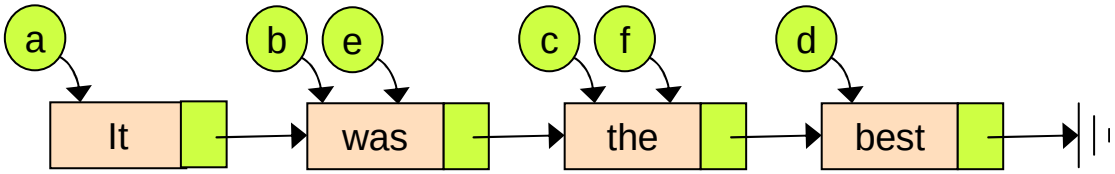
Client code

```
Word a = new Word ("It");
Word b = new Word ("was");
Word c = new Word ("the");
Word d = new Word ("best");
a.setNext(b);
b.setNext(c);
c.setNext(d);
d.setNext(null);
Word e = a.next();
Word f = a.next().next();
String txt = b.next().getData() // "the"
```

Comment: As we'll see shortly, there are more elegant and powerful ways to create and manage lists. This is just a preliminary example of the use of *references*.

Word class Implementation

Specification: A word consists of data (the word itself) and a reference to the next word



Word class implementation

```
public class Word {
    private String data;
    private Word next;

    public Word (String data) {
        this.data = data;
    }
    public void setNext (Word w) {
        next = w;
    }
    public String getData () {
        return data;
    }
    public Word next () {
        return next;
    }
}
```

Client code

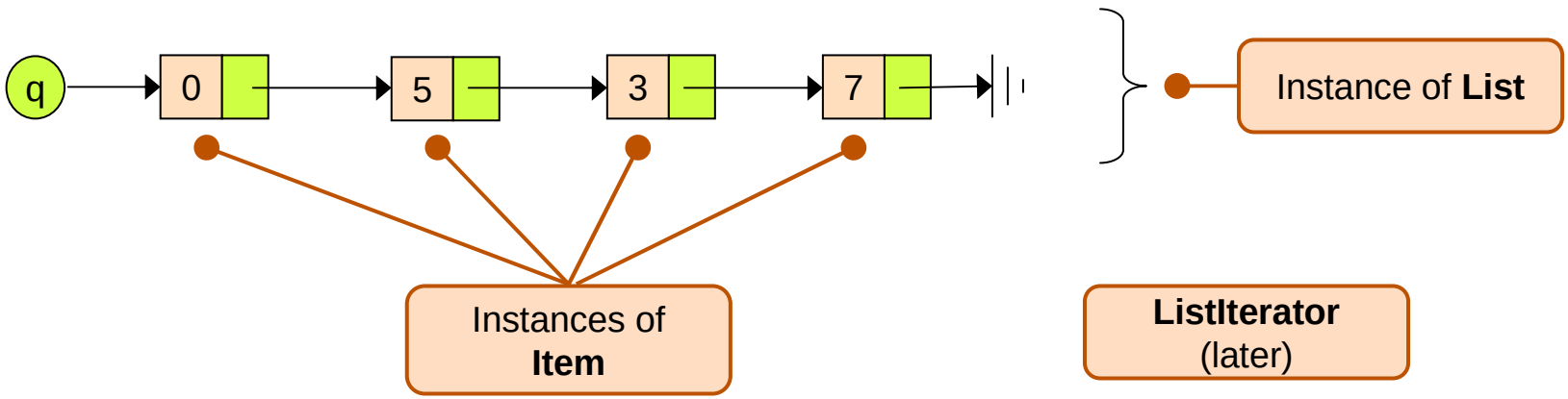
```
Word a = new Word ("It");
Word b = new Word ("was");
Word c = new Word ("the");
Word d = new Word ("best");
a.setNext(b);
b.setNext(c);
c.setNext(d);
d.setNext(null);
Word e = a.next();
Word f = a.next().next();
String txt = b.next().getData() // "the"
```

A linked list of integers

We now turn to describe a complete linked list example: list of integers.

Our linked list abstraction includes several classes:

- `Item`: represents individual items
- `List`: represents the list itself
- `ListIterator`: helps process the list



The Item class

```
package ListStructure;

// Represents a single item in the list
public class Item {
    // Friendly (package-private) fields
    int data;
    Item next;

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

    Item (int data) {
        this(data, null);
    }
}
```

No visibility modifiers

Two options for representing the item's data:

- Encapsulate the item's data by making its fields private and writing get/set methods to handle them
- Put the linked list classes in a package, and make the item's fields visible throughout the package (that's what we do in this example).

<u>Package visibility:</u>

Achieved by making the fields *package-private*

In Java, package-private is the default visibility option.

Using Item objects to form a linked list

```
package ListStructure;

// Represents a single item in the list
public class Item {
    // Friendly (package-private) fields
    int data;
    Item next;

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

    Item (int data) {
        this(data, null);
    }
}
```

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

The code above forms a list, but the resulting data structure is difficult to manage

We show it as yet another example of how references can be used

Note that a list consists of two parts:

- Head: item (aka "node" or "atom")
- Tail: a list.

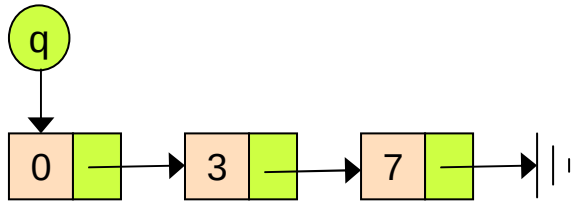
The List class

```
package ListStructure;

// Represents a list
public class List {
    private Item header;

    // Constructs a new list beginning
    // with a dummy item.
    public List () {
          header = new Item(0);
    }

    // Inserts an item at the beginning
    public void insert (Item item) {
          item.next = header.next;
          header.next = item;
    }

    public String toString () // later

    // More methods later
}
```

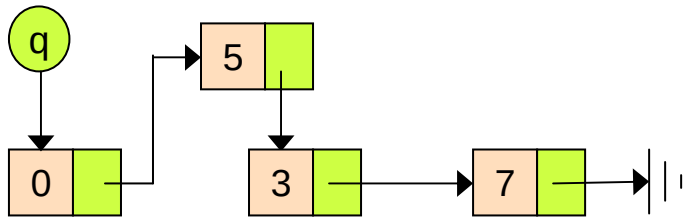
Client code

```
List q = new List();
q.insert(new Item(7));
q.insert(new Item (3));
q.insert(new Item (5));
System.out.println(q); // Prints 5 3 7
```

Before inserting 5:



After inserting 5:



The list's "handler" is its header

In this implementation, every list begins with a dummy item; This makes the implementation of the list's methods easier

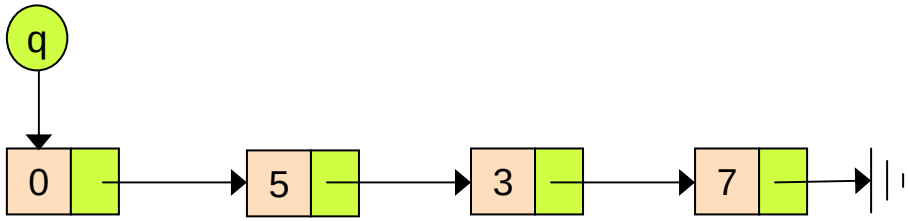
Question: Suppose we want to insert new items at the end of the list. How can we do it?

Processing the list

```
package ListStructure;

public class List {
    private Item header;

    // Returns the item containing x
    public Item find (int x) {
        Item p = header.next;
        while (!(p == null)) {
            if (p.data == x) return p;
            p = p.next;
        }
        return null;
    }

    // Returns a textual rep. of this list
    public String toString () {
        String s = "";
        Item p = header.next;
        while (!(p == null)) {
            s = s + p.data + " ";
            p = p.next;
        }
        return s;
    }
// More methods later
}
```

Client code

```
List q = new List();
q.insert(new Item(7));
q.insert(new Item(3));
q.insert(new Item(5));
System.out.println(q.find(3).data); // prints 3
System.out.println(q.find(8).data); // exception
System.out.println(q); // prints 5 3 7
```



<u>To process a list:</u>

Set a reference variable (say p) to the list's header

Process p.data

p = p.next

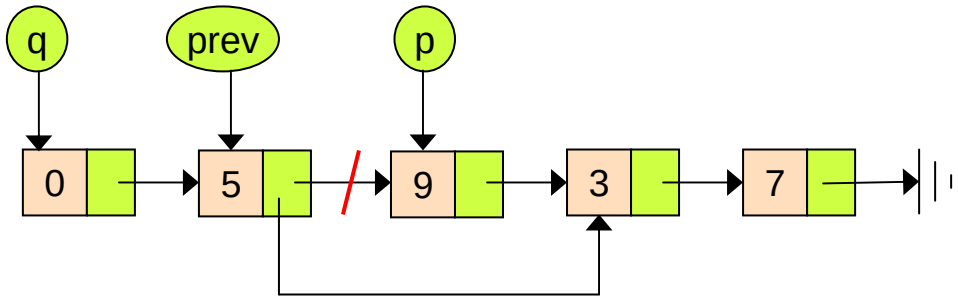
Do this until you've reached the list's end.

Removing an item

```
package ListStructure;

public class List {
   private Item header;

   // Constructor and other methods

   // Removes the item containing x
   public void remove (int x) {
      Item prev = header;
      Item p = header.next;
      while (!(p == null)) {
         if (p.data == x)
            prev.next = prev.next.next;
         prev = p;
         p = p.next;
      }
   }

    // More methods later

}
```

Client code

```
List q = new List();
q.insert(new Item(7));
q.insert(new Item(3));
q.insert(new Item(9));
q.insert(new Item(5));
System.out.println(q);  // prints 5 9 3 7
q.remove(9);
System.out.println(q);  // prints 5 3 7
```



The removed object will be recycled by the garbage collector.

Other common insertion variants

```
package ListStructure;

public class List {
   private Item header;

   // List constructor and previously shown methods

   // Inserts an item at the beginning of this list (same as before)
   public void insert (Item item)

   // Inserts an item at a specified location
   public void insertAt (Item item, int n)

   // Inserts an item at the end of this list
   public void insertAtEnd (Item item)

   // More / other methods.
}
```

Q: What about inserting …

- From an array?
- From a file?
- From another list?

Answer: KISS

(Keep it (the API) Simple, Stupid)

Other common methods of interest

```
package ListStructure;

public class List {
   private Item header;

   // List constructor and previously shown methods

   // Returns the item containing x  (same as before)
    public Item find (int x)

   // Returns the n'th item
   public Item findNth (int n)

   // Checks if the list contains x
   boolean contains (int x)

   // Checks if the list empty
   public boolean isEmpty ()

   // Empties the list
   public void makeEmpty ()

   // More / other methods.
}
```

But (remembering KISS), rather of packing all these operations into the List abstraction and API, a better design may be to let the client design them, as needed.

Set revisited: a linked list implementation

```
public class Set {
    private List elements;

    public Set() {
        elements = new List();
    }

    public boolean contains (int x) {
        return ((elements.find(x) == null) ? false : true);
    }

    public void insert (int x) {
        if (!contains(x))
            elements.insert(new Item(x));
    }

    public String toString () {
        return "{ " + elements.toString() + "}";
    }
}
```

```
Set s = new Set();
s.insert(2);
s.insert(7);
s.insert(5);
s.insert(7);
System.out.println(s);
// prints { 5 7 2 }
```

Iterators

```
package ListStructure;

public class List {
    private Item header;

    // Same as before
    public String toString1 () {
       String s = "";
       Item p = header.next;
       while (!(p == null)) {
          s = s + p.data + " ";
          p = p.next;
       }
       return s;
    }
```

An iterator is an object that provides "marching services" through a collection

The typical Iterator API includes:

- `hasNext()`: Returns true if the iteration has more items
- `next()`: Returns the next item
- `remove()`: Removes from the collection the last element returned by the iterator (optional operation).

```
    public String toString () {
          String s = "";
          for (ListIterator itr = new ListIterator(header.next); itr.hasNext();) {
                    s = s + itr.next().data + " ";
          }
          return s;
    }
}
```

Same effect as toString1 above, using an iterator

Iterator implementation

```
package ListStructure;

public class ListIterator {

    // current position in the list
    Item current;

    public ListIterator (Item item) {
        current = item;
    }

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

    public Item next () {
          Item item = current;
          current = current.next;
          return item;
    }
}
```

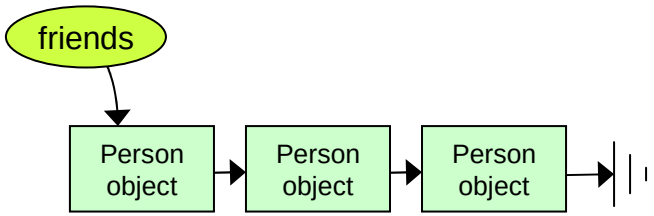
- Each collection typically offers an iterator that allows marching through the collection's items
- There are more ways to implement iterators, as we'll see later.

Generic lists

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

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

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

  // Returns a textual rep. of this person
  public String toString ()

  // Other Person methods follow.
}
```

Desired abstraction:

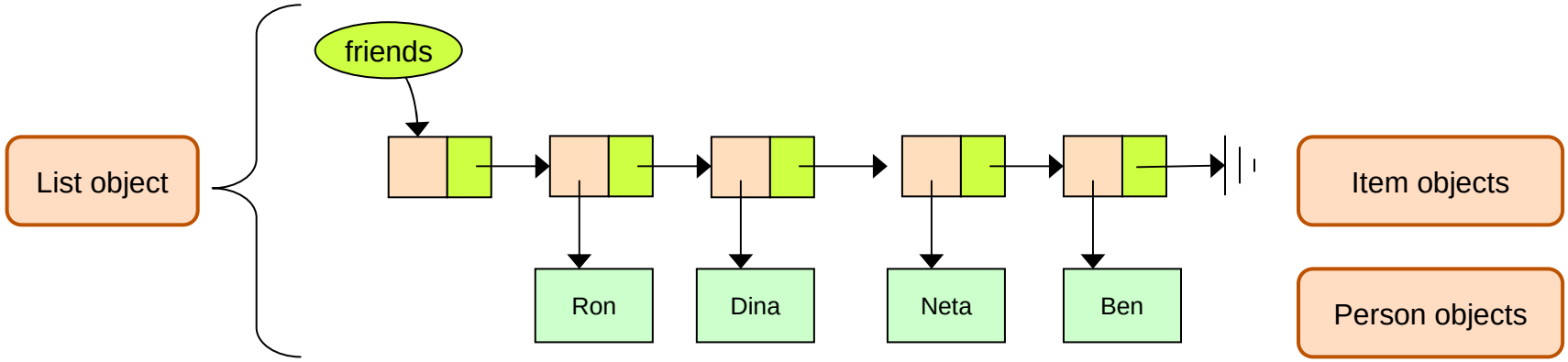


Problem: The Person object provides no mechanisms to link to other Persons

How can we evolve our List abstraction, which operates on lists of integers, into a generic list class that operates on other object types?

This extension need not be difficult, since the list is not really a list of integers, but rather a list of Item objects that include an `int` field.

Generic lists




```
public class Item {
    private Person data;
    private Item next;

    public Item (Person p, Item next) {
        this.data = p;
        this.next = next;
    }

    public Item (Person p) {
        this(p, null);
    }
}
```

- We also have to rewrite our List class, which is designed to operate on integers, not on Person objects
- Can we create a generic List class that operated on any object?
- Yes – this is called "generics".

Outline

- Data structures / collections
- Set
- Stack
- lists
- Hashing

Hashing

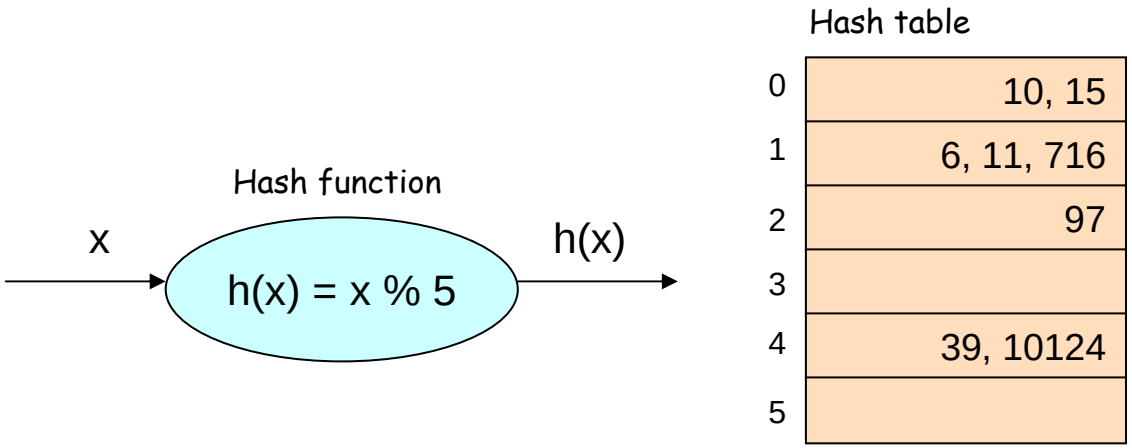
Arrays provide a reasonable solution for finding values but a a problematic solution for inserting and deleting values.

<u>Hashing:</u> a technique used to perform finding, insertions, and deletions, of values in constant average time (independent of the number of values)

Two central players:

- Hash table ADT
- Hashing function.

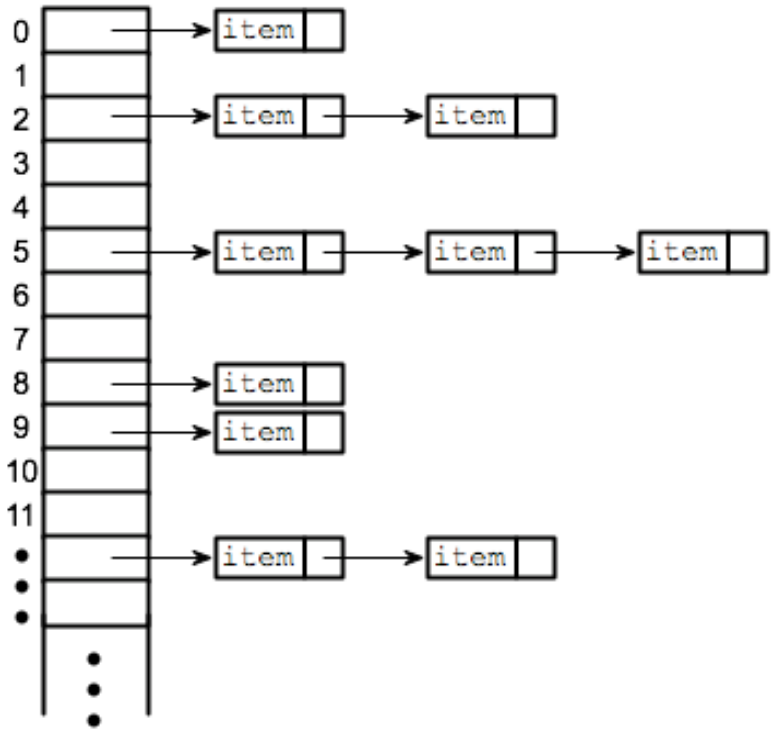
Hashing: the basic idea



- Efficient finding, insertion, deletion
- Ideally, the size of the table should be the number of keys; typically, not practical
- Ideally, the keys should distribute evenly across the table; typically, can be achieved
- A prime table size helps promote an even distribution
- Issue: how to handle collisions.

Hash table with separate chaining

Each cell in the hash table points to a linked list:



The worst possible hash function:

h(x) = 17

<u>To find / delete / insert a key x:</u>

Compute h(x) and perform the find/delete/insert operation on the respective list

<u>Efficiency:</u> The h(x) computation takes constant time;
the processing of the list takes linear time; if the lists are not long, we'll be OK.

<u>Operations like findMin, findMax, or scanning the values in order:</u> not supported.

Hash functions

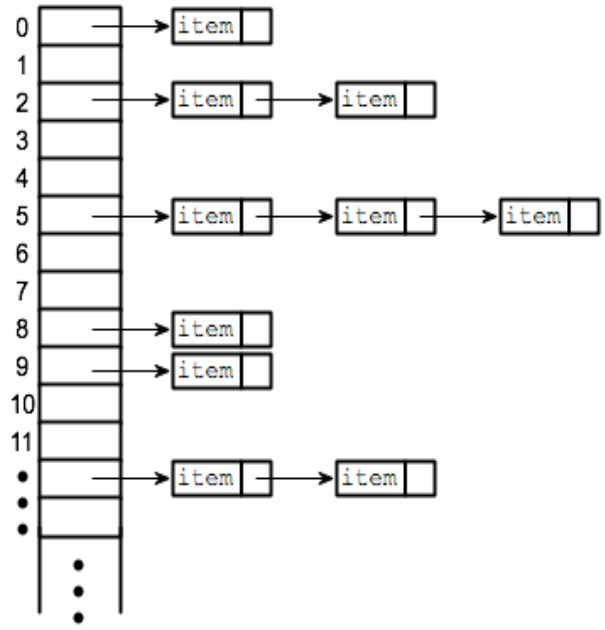
Hash function for integer keys: simple. `h(x) = x % tableSize`

Typical hash function for a `String` key

```
Public static int hash (String key, int tableSize) {
    int hasValue = 0;
    for (int i = 0; i < key.length(); i++) {
        hashValue += key.charAt(i);
    return hashValue % tableSize;
}
```

Hash table implementation

```
public class HashTable {

    private static final int DEFAULT_TABLE_SIZE = 101;

    private List[] table;

    private int tableSize;

    public hashTable ()

    public hashTable (int size)

    public void makeEmpty()

    public void insert (Object x)

    public void remove (Object x)

    public Object find (Object x)

    private static int hash (Object key, int tableSize)

    private static int nextPrime (int n)

}
```



The role of hashing in managing Java objects

- When a new object is created, Java computes a hash code for it. The hash code becomes a unique identifier of the object, and can be accessed via statements like
  `int hashCode = someObject.hashCode();`

- The `hashCode()` method is part of Java's `Object` API

- Often, class writers override the standard `hashCode()` and write their own hash code function

```
// Returns a hash code for this Fraction.
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + denominator;
    result = prime * result + numerator;
    return result;
}/
```

- IDE's (like Eclipse) provide standard `hashCode` methods

- More about this, later in the course.