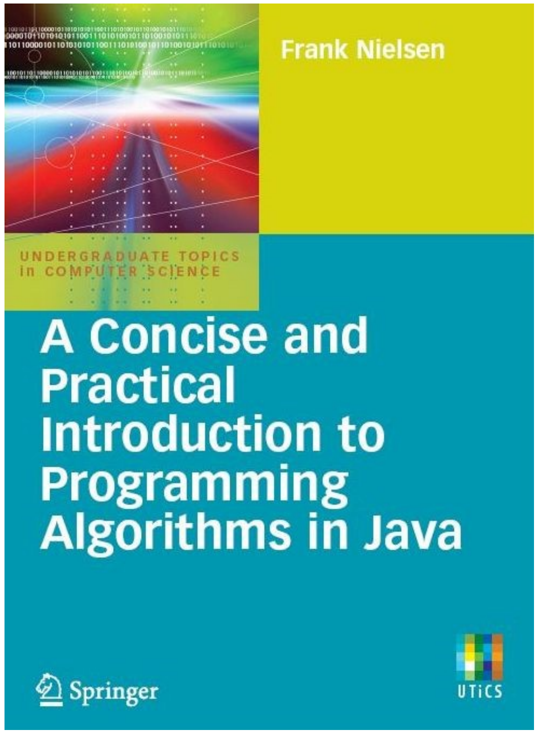
A Concise and Practical Introduction to Programming Algorithms in Java


Frank Nielsen
UNDERGRADUATE TOPICS in COMPUTER SCIENCE
A Concise and Practical Introduction to Programming Algorithms in Java
Springer
UTiCS

Chapter 7: Linked lists

Frank NIELSEN

nielsen@lix.polytechnique.fr

Agenda

- Cells and linked lists
- Basic static functions on lists
- Recursive static functions on lists
- Hashing: Resolving collisions
- Summary of search method
  (with respect to time complexity)

Summary of Lecture 7

Searching:
- Sequential search (linear time) / arbitrary arrays
- Dichotomic search (logarithmic time) / ordered arrays

Sorting:
- Selection sort (quadratic time)
- Quicksort (recursive, in-place, O(n log n) exp. time)

Hashing

Methods work on arrays...
...weak to fully dynamic datasets

Memory management in Java: AUTOMATIC

- Working memory space for functions (stack):
  PASS-BY-VALUE

- Global memory for storing arrays and objects:
  Allocate with `new`

- Do not free allocated objects, Java does it for you!
  GARBAGE COLLECTOR
  (GC for short)

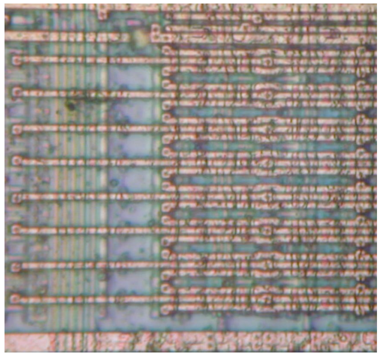
http://en.wikipedia.org/wiki/Java_(programming_language)

Memory management

DRAM: volatile memory
1 bit: 1 transistor/1 capacitor,
constantly read/rewritten

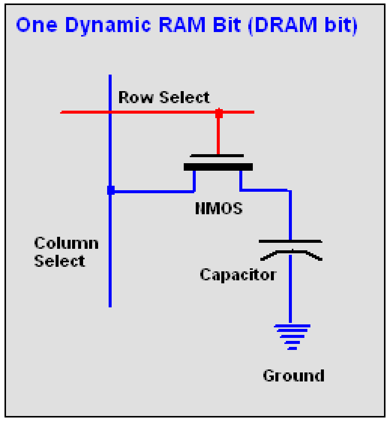
HDD: hard disk, static memory

Dynamic RAM



RAM cells

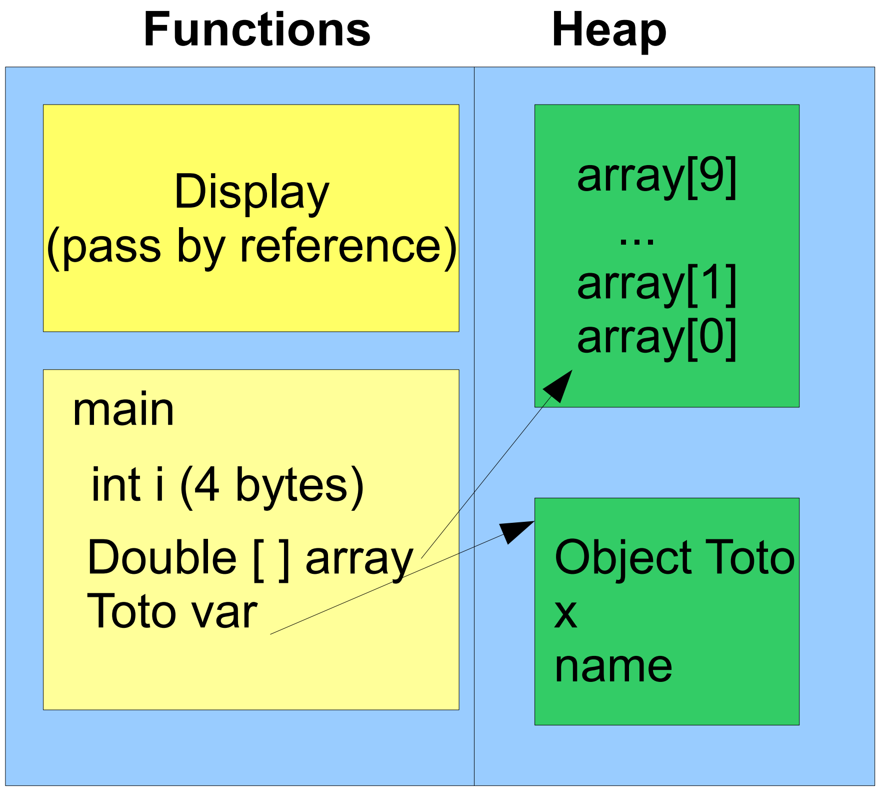
From Computer Desktop Encyclopedia
© 2005 The Computer Language Co. Inc.



Dynamic memory: Linear arrays...
Problem/Efficiency vs Fragmentation...

Visualizing memory
A representation

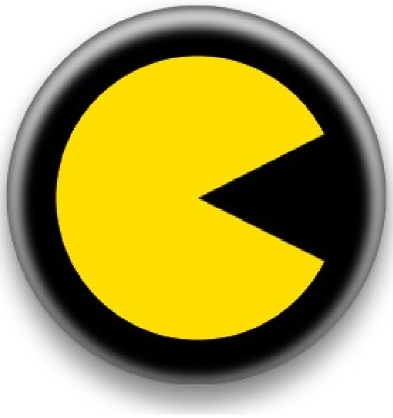
```
class Toto
{
double x;
String name;

Toto(double xx, String info)
{this.x=xx;
// For mutable object do this.name=new Object(info);
 this.name=info;}
};


class VisualizingMemory
{
public static void Display(Toto obj)
{
System.out.println(obj.x+":"+obj.name);
}

public static void main(String[] args)
{
int i;
Toto var=new Toto(5,"Favorite prime!");

double [] arrayx=new double[10];

Display(var);
}
}
```



local variables
stack execution
pass-by-value
(non-persistent)

arrays
objects
persistent

Garbage collector (GC)



No destructor:
- for objects
- for arrays

Objects no longer referred to are automatically collected

You *do not have* to explicitly free the memory
Java does it automatically on your behalf

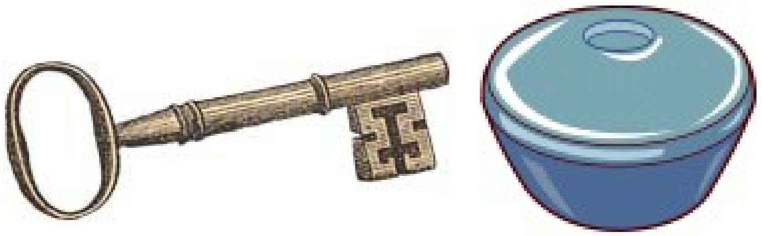
Objects no longer needed can be explicitly “forgotten”

```
obj=null;
array=null;
```

Flashback: Searching

- Objects are accessed via a corresponding **key**
- Each object stores its key and additional fields
- One seeks for information stored in an object from its key (key= a handle)
- All objects are in the main memory (no external I/O)

Today!



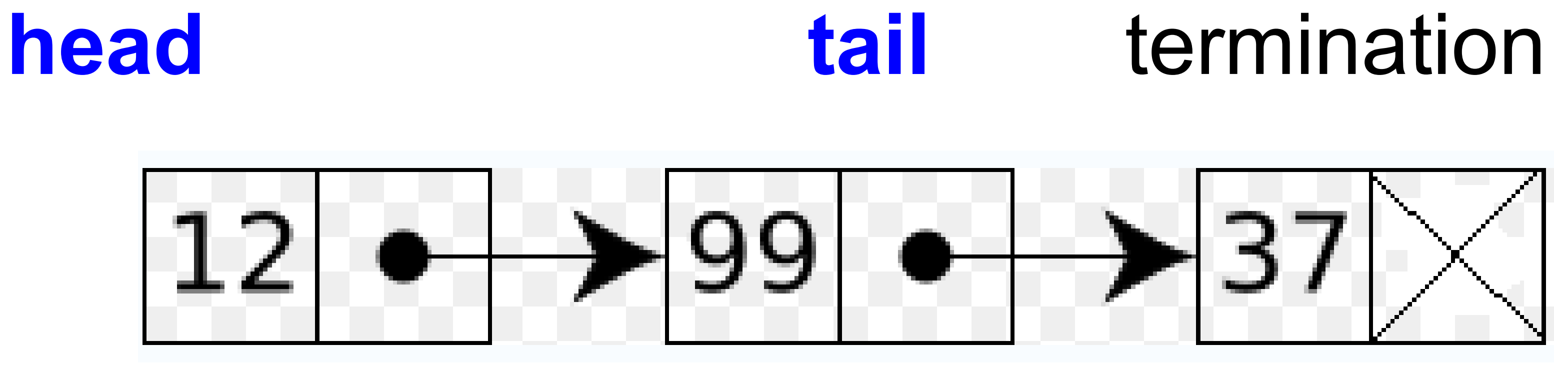
More challenging problem:

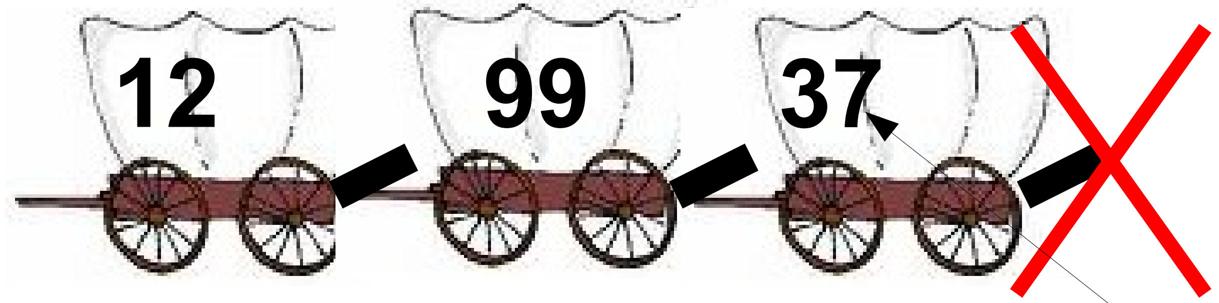
Adding/removing or changing object attributes dynamically

Linked list: cells and links

- Sequence is made of cells
- Each cell stores an object (cell=container)
- Each cell link to the following one
  (=refer to, =point to)
- The last cell links to nothing (undefined)
- To add an element, create a new cell that...
  ...points to the first one (=head)
- Garbage collector takes care of cells not pointed by others

Linked list: cells and links





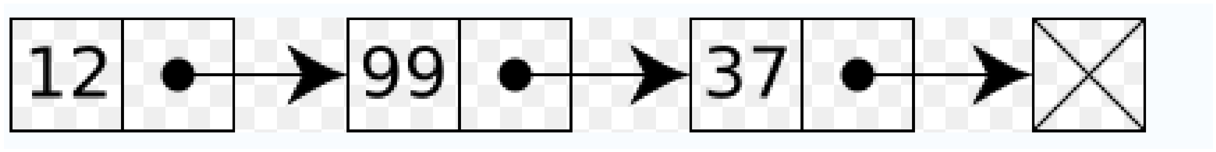
Cell = wagon
Link = magnet

Container:
Any object is fine

Lisp: A language based on lists

Lisp (1958) derives from "List Processing Language"
Still in widespread use nowdays

(list '1 '2 'foo)
(list 1 2 (list 3 4))



(12 (99 (37 nil)))
(head tail)

http://en.wikipedia.org/wiki/LISP

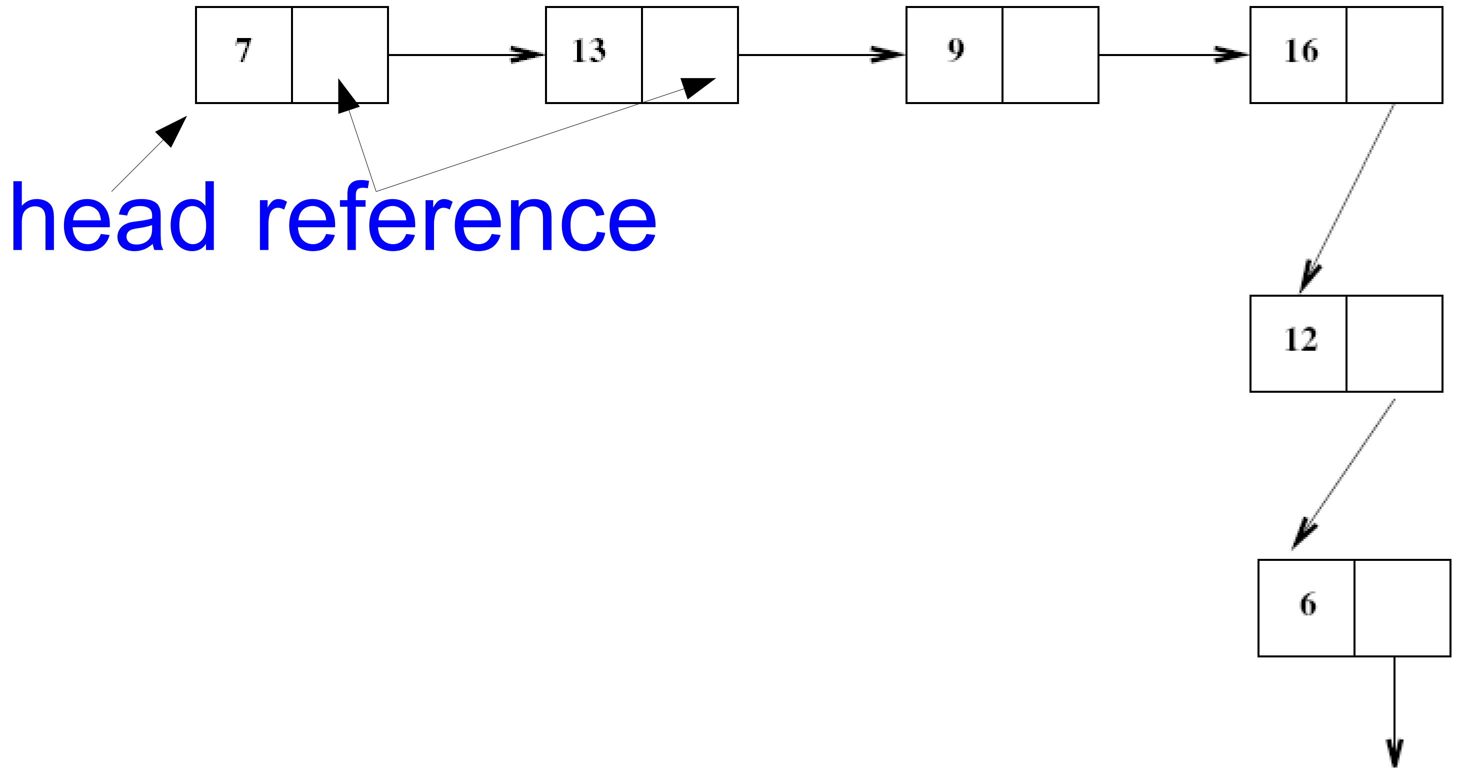
Advantages of linked lists

- Store and represent a set of objects
- But we do not know beforehand how many...
- Add/remove dynamically to the set elements

Arrays: Memory compact data-structure for static sets

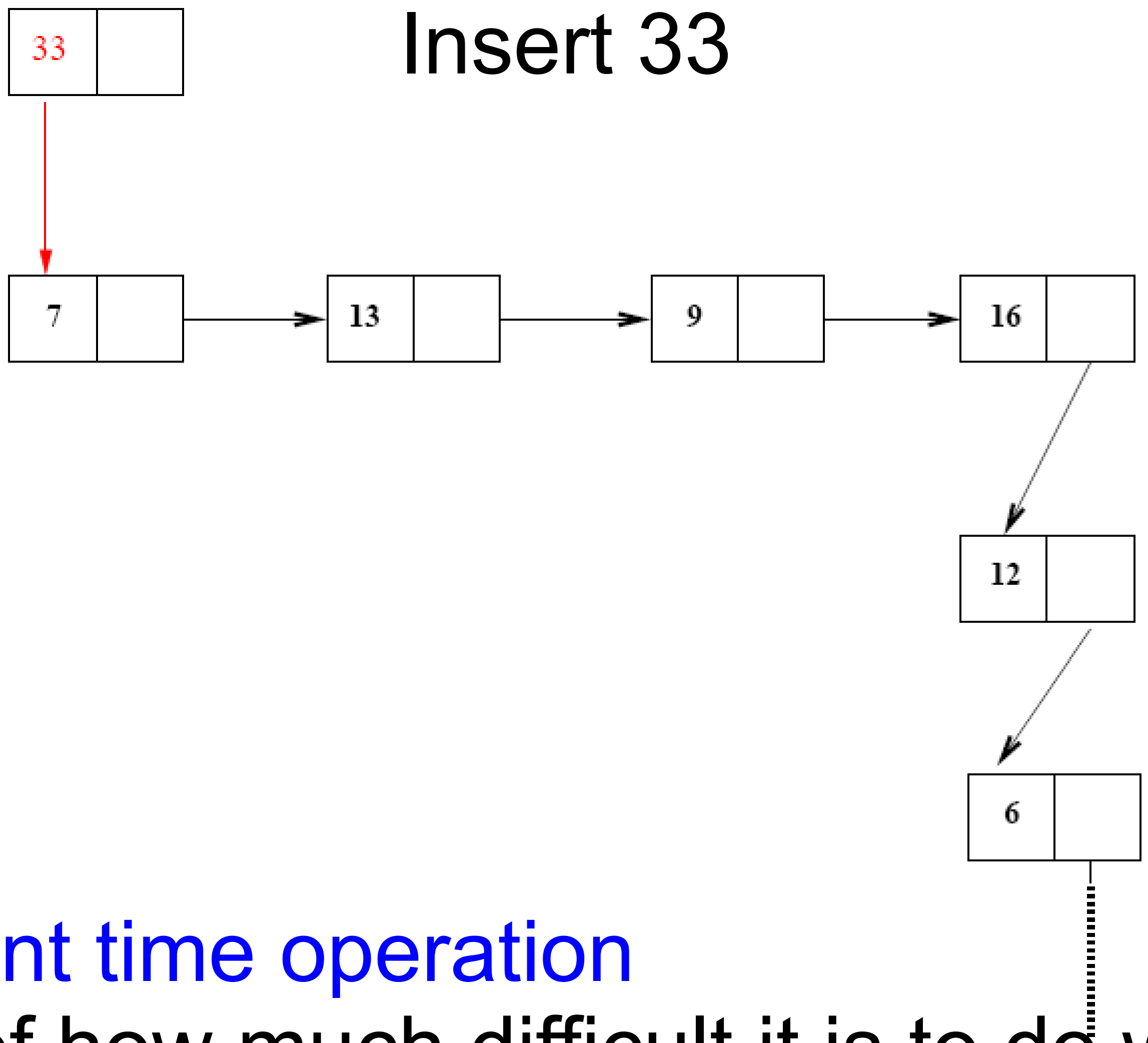
Linked lists: Efficient data-structure for dynamic sets
but use references to point to successors
(reference= 4 bytes)

Linked lists



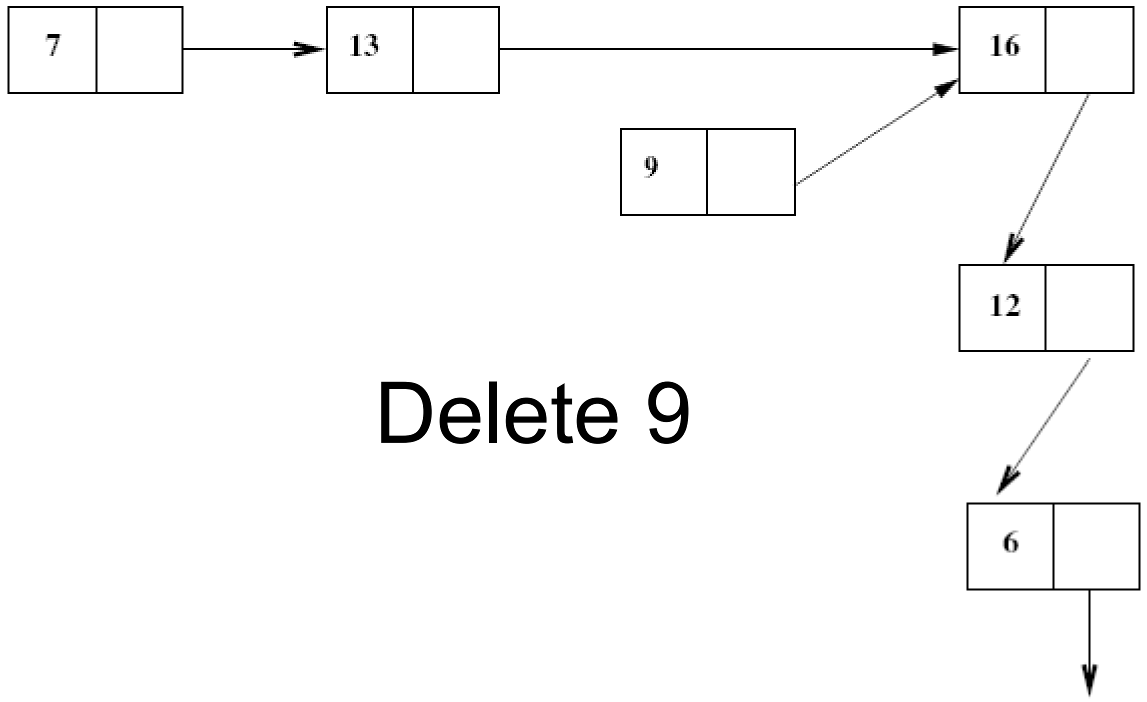
Dynamic insertion

Insert 33



Constant time operation
(think of how much difficult it is to do with arrays)

Dynamic deletion



Delete 9

Constant time operation
(think of how much difficult it is to do with arrays)

Abstract lists

Lists are *abstract data-structures* supporting the following operations (interface):

Constant:	Empty list listEmpty (`null`)
Operations:	
Constructor:	List x Object → List
Head:	List → Object (not defined for listEmpty)
Tail:	List → List (not defined for listEmpty)
isEmpty:	List → Boolean
Length:	List → Integer
belongTo:	List x Object → Boolean
...	

Linked list in Java

- `null` is the empty list (=not defined object)
- A cell is coded by an object (class with fields)
- Storing information in the cell = creating field (say, `double, int, String, Object`)
- Pointing to the next cell amounts to contain a reference to the next object

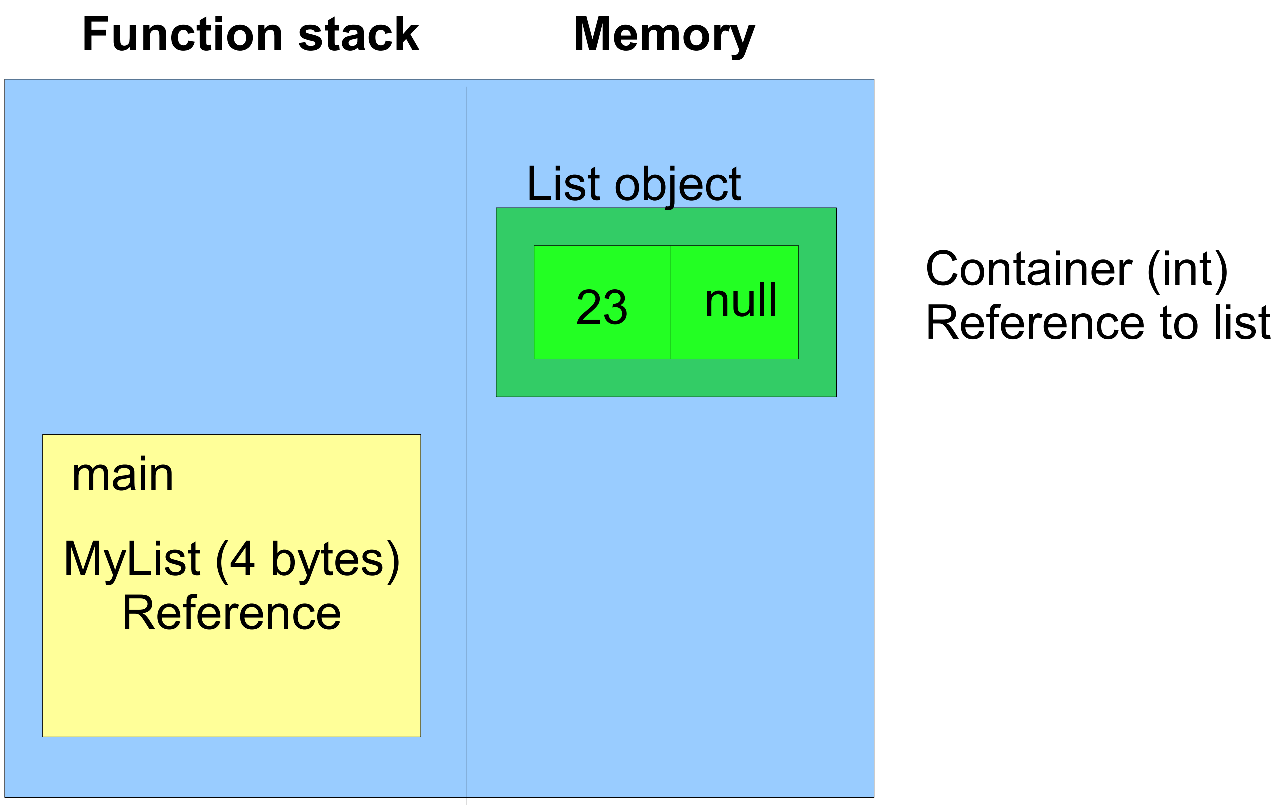
```
public class List
{
int container;
List next;

// Constructor  List(head, tail)
List(int element, List tail)
   {
   this.container=element;
   this.next=tail;
   }

static boolean isEmpty(List list)
   {// in compact form return (list==null);
   if (list==null) return true;
      else return false;
   }

static int head(List list)
   {return list.container;}

static List tail(List list)
   {return list.next;}
}
```

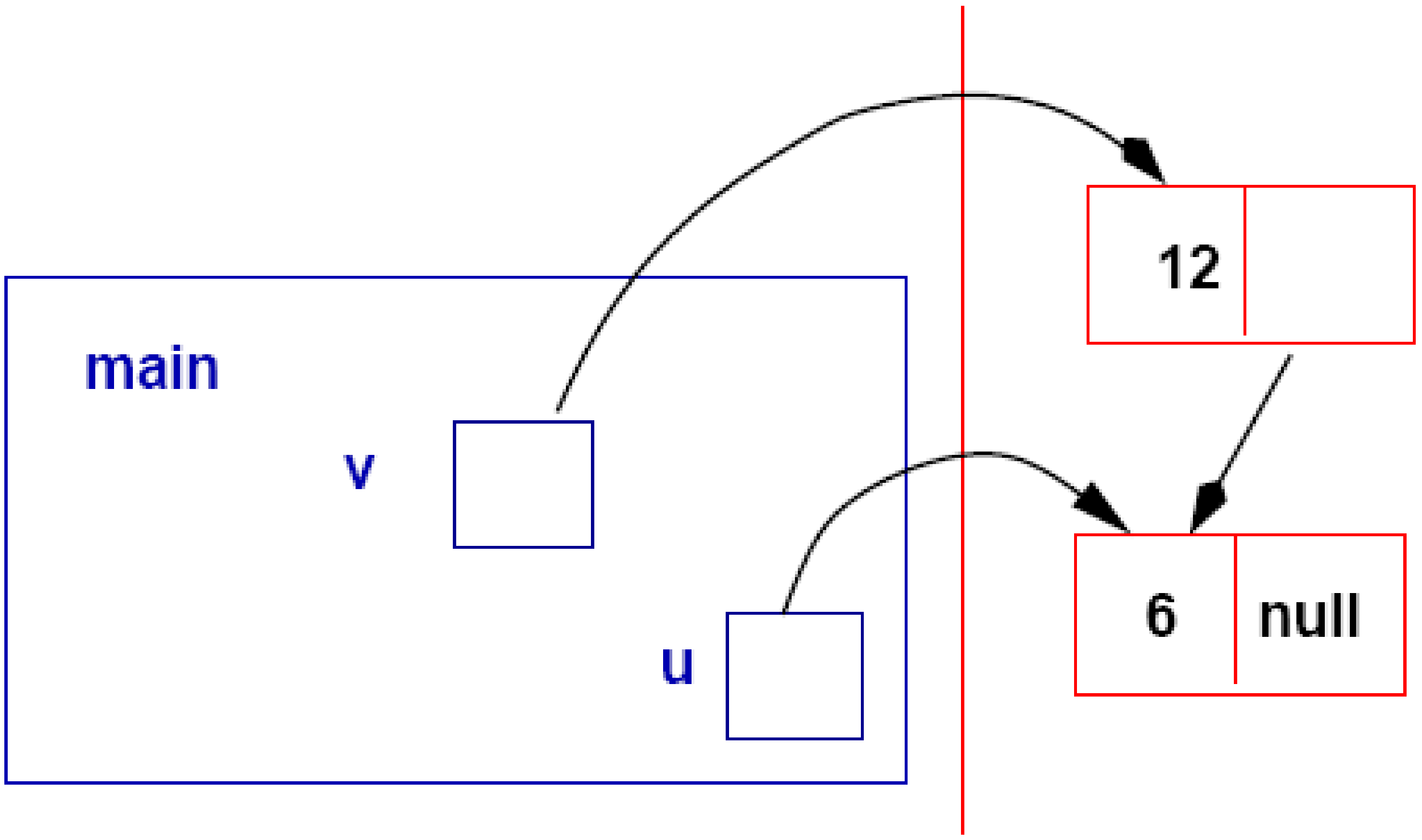
Common mistake

- Cannot access fields of the `null` object
- Exception `nullPointerException` is raised
- Perform a test `if (currentCell!=null)` to detect wether the object is void or not, before accessing its fields

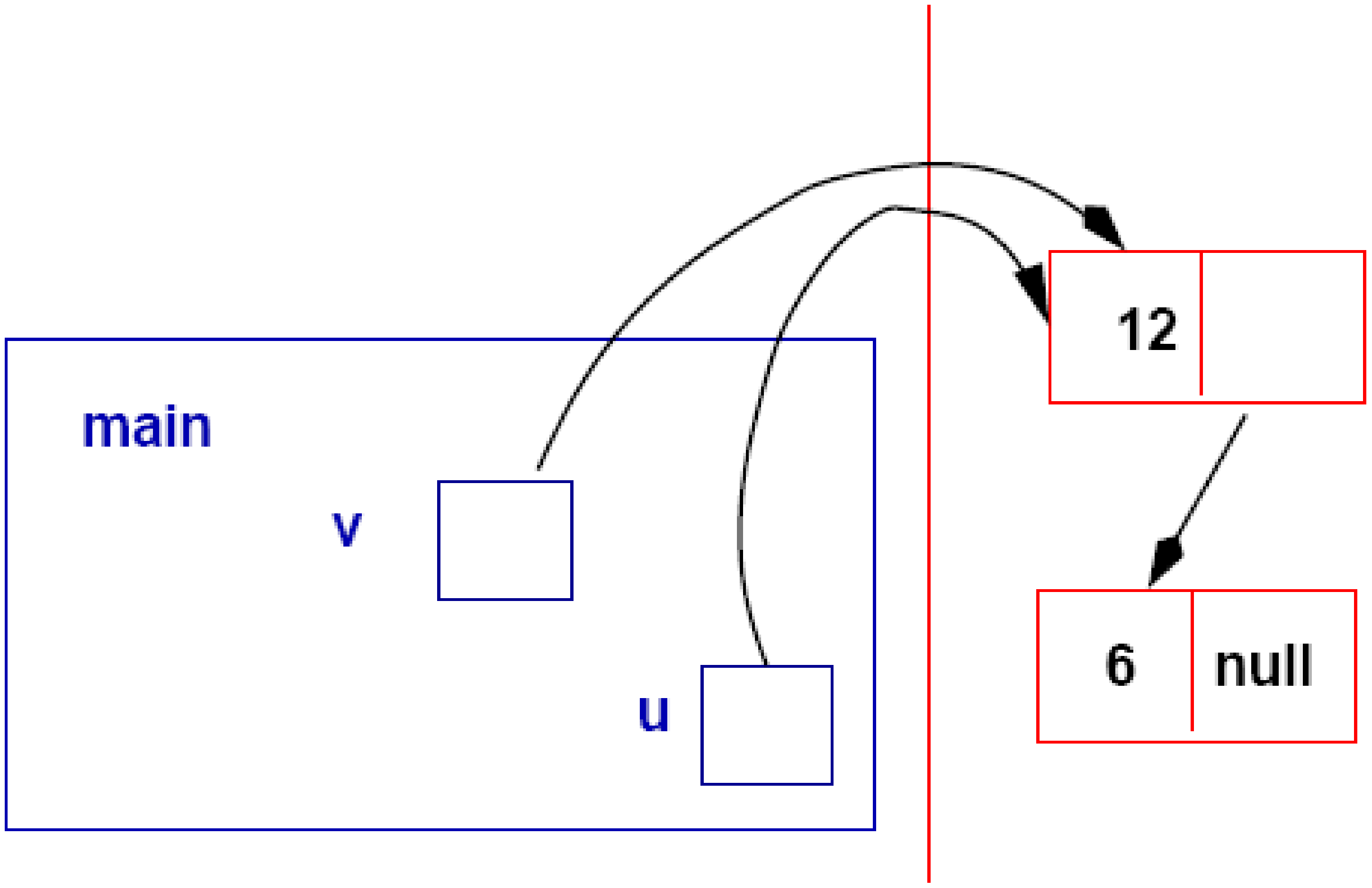
```
static int head(List list)
   {if (list!=null)
          return list.container;
             else
                return -1; }
```

```
public class List
{...}

class ListJava{

   public static void main (String[] args)
   {
      List myList=new List(23,null);
   }
}
```

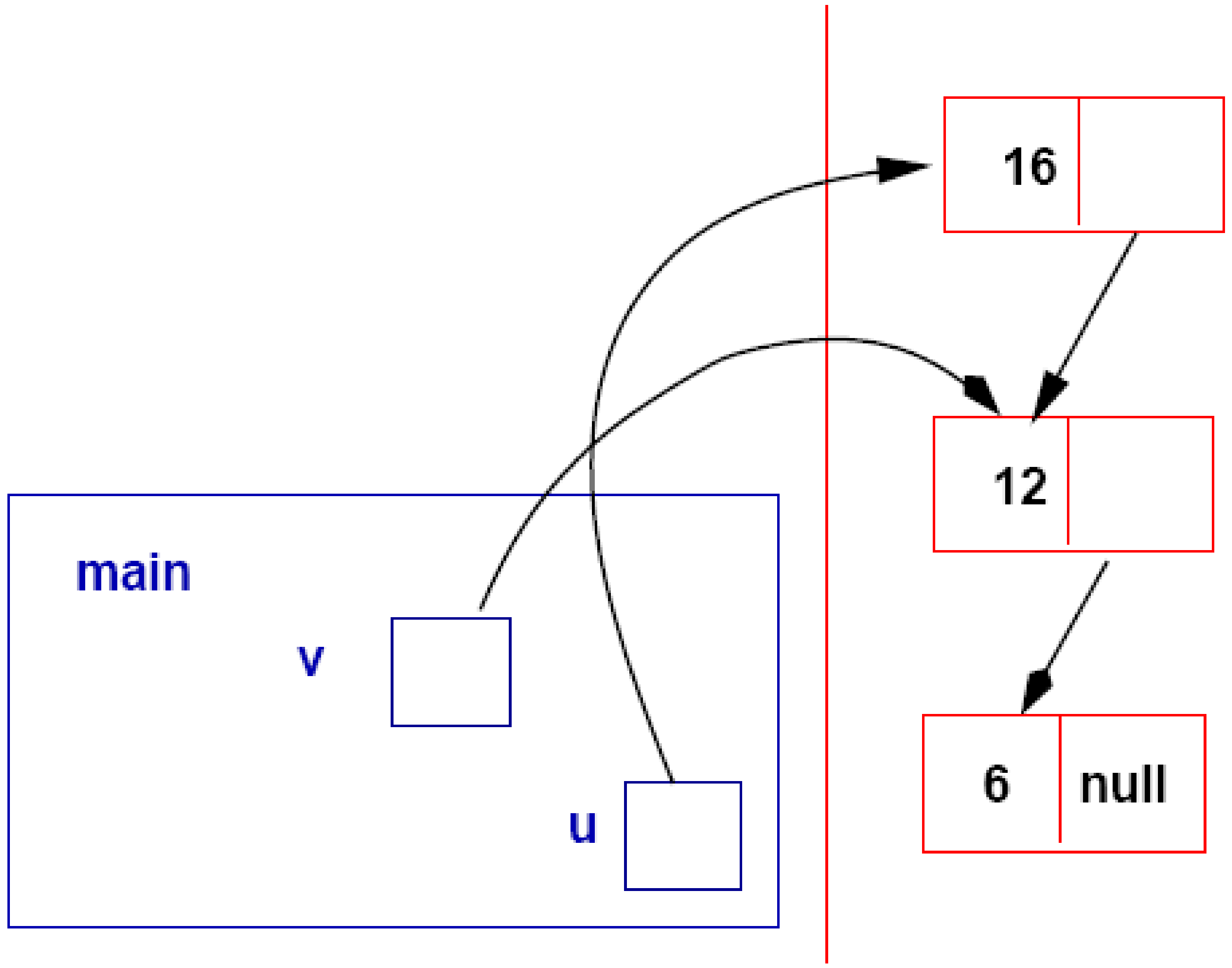




```
class ListJava{

   public static void main (String[] args)
   {
      List u=new List(6,null);
      List v=new List(12,u);
   }
}
```



```
u=v;
```



```
u=new List(16,u);
```

Browsing lists

Start from the head, and
inspect element by element (chaining with references)
until we find the empty list (termination)

```
static boolean belongTo(int element, List list)
{
while (list!=null)
   {
      if (element==list.container) return true;
      list=list.next;
   }
   return false;
}
```

Linear complexity O(n)

List: Linear search complexity O(n)

```
class ListJava{

    public static void main (String[] args)
    {
        List u=new List(6,null);
        u=new List(16,u);
        u=new List(32,u);
        u=new List(25,u);


System.out.println(List.belongTo(6,u));
System.out.println(List.belongTo(17,u));
    }
}
```

```
static boolean belongTo(in[illegible]nt, List list)
{
while (list!=null)
   {
       if (element==list.container) return true;
       list=list.next;
   }
   return false;
}
```

==

equals

Generic lists

```
class ListString
{
String name;
ListString next;

// Constructor
ListString(String name, ListString tail)
    {this.name=new String(name); this.next=tail;}

static boolean isEmpty(ListString list)
    {return (list==null);}

static String head(ListString list)
    {return list.name;   }

static ListString tail(ListString list)
    {return list.next;}

static boolean belongTo(String s, ListString list)
{
while (list!=null)
    {
        if (s.equals(list.name))
             return true;
        list=list.next;
    }
    return false;
}
}
```

Generic lists

```
class ListString
{
String name;
ListString next;
...
static boolean belongTo(String s, ListString list)
{
while (list!=null)
    {
        if (s.equals(list.name))
            return true;
        list=list.next;
    }
    return false;
}

}

class Demo{...
    ListString l=new ListString("Frank",null);
        l=new ListString("Marc",l);
        l=new ListString("Frederic",l);
        l=new ListString("Audrey",l);
        l=new ListString("Steve",l);
        l=new ListString("Sophie",l);

        System.out.println(ListString.belongTo("Marc",l));
        System.out.println(ListString.belongTo("Sarah",l));
}
```

Length of a list

OK

```
static int length(ListString list)
{
   int l=0;
   while (list!=null)
      {l++;
      list=list.next;
      }
   return l;
}
```

Note that because Java is pass-by-value
(reference for structured objects),
we keep the original value, the head of the list,
after the function execution.

```
System.out.println(ListString.length(l));
System.out.println(ListString.length(l));
```

Dynamic insertion: Add an element to a list

```
static ListString Insert(String s, ListString list)
{
return new ListString(s,list);
}
```

Call static function Insert of the class ListString

```
l=ListString.Insert("Philippe", l);
l=new ListString("Sylvie",l);
```

Pretty-printer of lists

Convenient for debugging operations on lists

```
static void Display(ListString list)
{
   while(list!=null)
   {
   System.out.print(list.name+"-->");
   list=list.next;
   }
System.out.println("null");
}
```

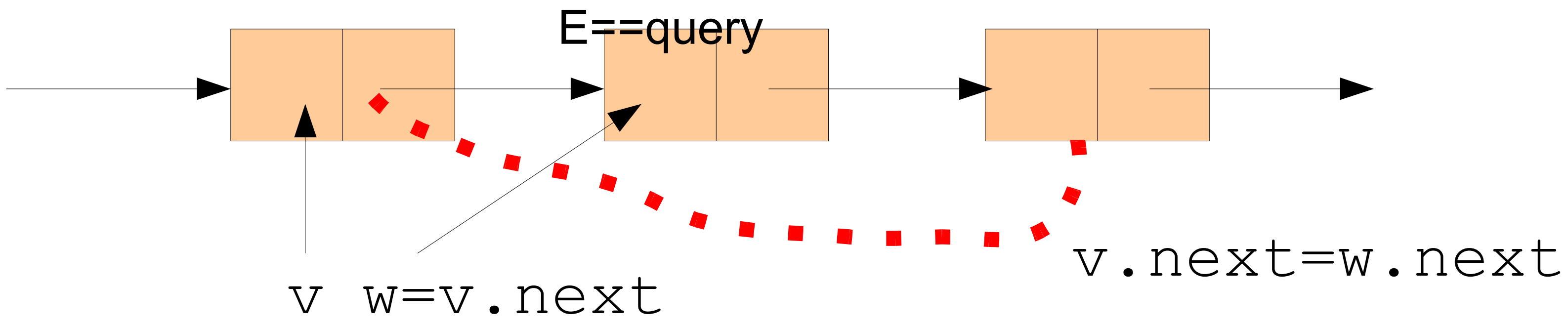
```
Philippe-->Sophie-->Steve-->Audrey-->Frederic-->Marc-->Frank-->null
```

```
ListString.Display(l);
```

Dynamic deletion: Removing an element

Removing an element from a list:

Search for the location of the element,
if found then ***adjust the list*** (kind of list surgery)



Garbage collector takes care of the freed cell

Take care of the special cases:

- List is empty
- Element is at the head

Dynamic deletion: Removing an element

```
static ListString Delete(String s, ListString list)
{
// if list is empty
if (list==null)
   return null;

// If element is at the head
if (list.name.equals(s))
   return list.next;

// Otherwise
ListString v=list;
ListString w=list.next; //tail

while( w!=null && !((w.name).equals(s)) )
   {v=w; w=v.next;}
// A bit of list surgery here
if (w!=null)
   v.next=w.next;

return list;
}
```

Complexity of removing is at least the complexity of finding if the element is inside the list or not.

Recursion & Lists

Recursive definition of lists yields effective recursive algorithms too!

```
static int lengthRec(ListString list)
{
   if (list==null)
      return 0;
   else
      return 1+lengthRec(list.next);
}
```

```
System.out.println(ListString.lengthRec(l));
```

Recursion & Lists

```
static boolean belongToRec(String s, ListString list)
{
if (list==null) return false;
   else
   {
      if (s.equals(list.name))
            return true;
            else
            return belongToRec(s,list.next);
   }
}




...
 System.out.println(ListString.belongToRec("Marc",l));
```

Note that this is a terminal recursion
(thus efficient rewriting is possible)

Recursion & Lists

Displaying recursively a linked list

```
static void DisplayRec(ListString list)
{
  if (list==null)
    System.out.println("null");
    else
      {
        System.out.print(list.name+"-->");
        DisplayRec(list.next);
      }
}

...

  ListString.DisplayRec(l);
```

Copying lists

Copy the list by traversing the list from its head, and **cloning** one-by-one all elements of cells (fully copy objects like String etc. stored in cells)

```
static ListString copy(ListString l)
{
ListString result=null;

while (l!=null)
{
   result=new ListString(l.name,result);
   l=l.next;
}
return result;


}
```

```
ListString lcopy=ListString.copy(l);
ListString.Display(lcopy);
```

Beware: Reverse the list order

Copying lists: Recursion

```
static ListString copyRec(ListString l)
{
if (l==null)
   return null;
      else
      return new ListString(l.name,copyRec(l.next));
}
```

Preserve the order

```
ListString.DisplayRec(l);
ListString lcopy=ListString.copy(l);
ListString.Display(lcopy);
ListString lcopyrec=ListString.copyRec(l);
ListString.Display(lcopyrec);
```

```
Sophie-->Audrey-->Frederic-->Marc-->null
Marc-->Frederic-->Audrey-->Sophie-->null
Sophie-->Audrey-->Frederic-->Marc-->null
```

Building linked lists from arrays

```
static ListString Build(String [] array)
{
ListString result=null;

// To ensure that head is the first array element
// decrement: from largest to smallest index
for(int i=array.length-1;i>=0;i--)
  result=new ListString(array[i],result);

return result;
}
```

```
String [] colors={"green", "red", "blue", "purple", "orange", "yellow"};
ListString lColors=ListString.Build(colors);
ListString.Display(lColors);
```

```
green-->red-->blue-->purple-->orange-->yellow-->null
```

Summary on linked lists

- Allows one to consider **fully dynamic** data structures
- Singly or doubly linked lists (`List prev,succ;`)
- Static functions: Iterative (`while`) or recursion
- List object is a **reference**
  (pass-by-reference of functions; preserve head)
- Easy to get bugs and never ending programs
  (`null` empty list never encountered)
- Do not care releasing unused cells
  (**garbage collector** releases them automatically)

Hashing: A fundamental technique

- Store object `x` in array position `h(x)  (int)`

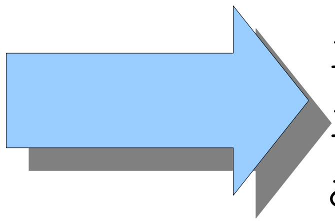
- Major problem occurs if two objects `x` and `y` are stored on the same cell: **Collision.**

Key issues in hashing:

- Finding good hashing functions that **minimize collisions**,

- Adopting a good search policy in case of collisions

```
int i;
array[i]
```



```
Object Obj=new Object();
int i;
i=h(Obj);// hashing function
array[i]
```

Hashing functions

- Given a universe X of keys and for any x in X, find an integer **h(x)** between 0 and m

- Usually *easy to transform* the object into an integer:

For example, for strings just add the ASCII codes of characters

- The problem is then to transform
  a set of n (sparse) integers

  into a compact array of size m<<N.

(<< means much less than)

Hashing functions

Key idea is to take the modulo operation

$h(k) = k \bmod m$ where m is a **prime number**.

```
static int m=23;
   // TRANSCODE strings into integers
   static int String2Integer(String s)
   {
      int result=0;

      for(int j=0;j<s.length();j++)
         result=result*31+s.charAt(j);
         // this is the method s.hashCode()

      return result;
   }

   // Note that m is a static variable
   static int HashFunction(int l)
   {return l%m;}
```

```
  public static void main (String[] args)
  {
  String [] animals={"cat","dog","parrot","horse","fish",
  "shark","pelican","tortoise", "whale", "lion",
  "flamingo", "cow", "snake", "spider", "bee", "peacock",
  "elephant", "butterfly"};

  int i;
  String [] HashTable=new String[m];

  for(i=0;i<m;i++)
     HashTable[i]=new String("-->");



  for(i=0;i<animals.length;i++)
     {int pos=HashFunction(String2Integer(animals[i]));
     HashTable[pos]+=(" "+animals[i]);
     }

  for(i=0;i<m;i++)
     System.out.println("Position "+i+"\t"+HashTable[i]);

}
```

```
Position 0     --> whale
Position 1     --> snake
Position 2     -->
Position 3     -->
Position 4     -->
Position 5     -->
Position 6     -->
Position 7     --> cow
Position 8     --> shark
Position 9     -->
Position 10     -->
Position 11     -->
Position 12     --> fish
Position 13     --> cat
Position 14     -->
Position 15     --> dog tortoise
Position 16     --> horse
Position 17     --> flamingo
Position 18     -->
Position 19     --> pelican
Position 20     --> parrot lion
Position 21     -->
Position 22     -->
```

Collisions in the hash table

Hashing: Solving collision

Open address methodology

...record in another location that is still open...

- Store object X at the **first free hash table cell** starting from position h(x)

- To seek whether X is in the hash table, compute h(x) and inspect all hash table cells **until** h(x) is found or a free cell is reached.

Complexity of search time ranges from constant O(1) to linear O(m) time

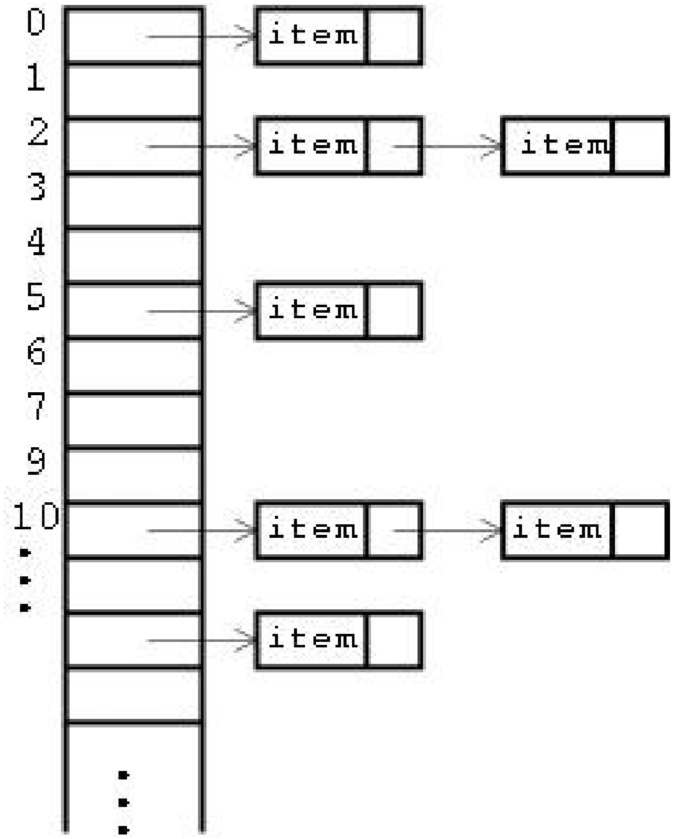
```
String [] HashTable=new String[m];
// By default HashTable[i]=null


   for(i=0;i<animals.length;i++)
   {
   int s2int=String2Integer(animals[i]);
   int pos=HashFunction(s2int);

   while (HashTable[pos]!=null)
      pos=(pos+1)%m;

   HashTable[pos]=new String(animals[i]);
   }
```

Position 0	whale
Position 1	snake
Position 2	bee
Position 3	spider
Position 4	butterfly
Position 5	null
Position 6	null
Position 7	cow
Position 8	shark
Position 9	null
Position 10	null
Position 11	null
Position 12	fish
Position 13	cat
Position 14	peacock
Position 15	dog
Position 16	horse
Position 17	tortoise
Position 18	flamingo
Position 19	pelican
Position 20	parrot
Position 21	lion
Position 22	elephant

Hashing: Solving collision

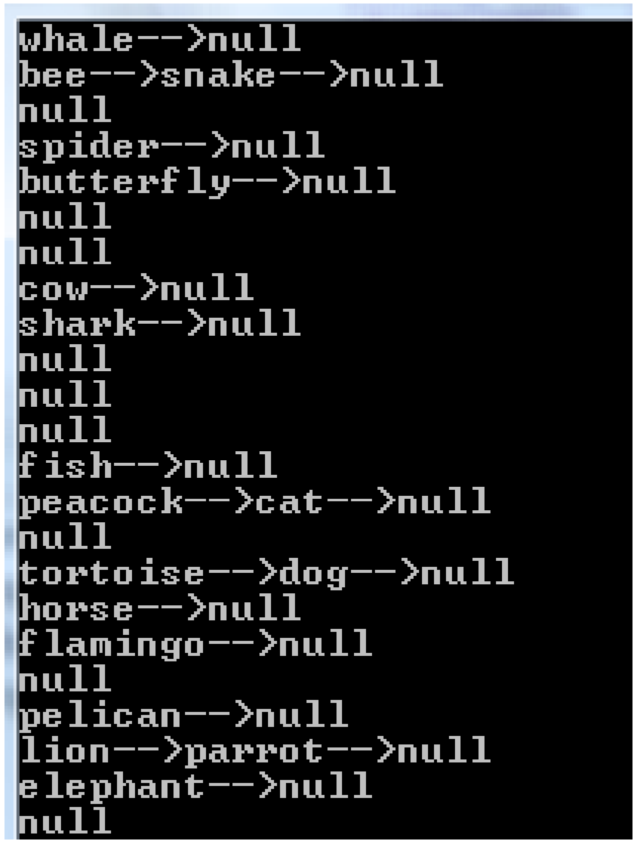
Chained Hashing

For array cells not open, create linked lists



Can add as many elements as one wishes

```
ListString [] HashTable=new ListString[m];

   for(i=0;i<m;i++)
      HashTable[i]=null;

for(i=0;i<animals.length;i++)
{
int s2int=String2Integer(animals[i]);
int pos=HashFunction(s2int);
HashTable[pos]=ListString.Insert(animals[i],HashTable[pos]);
}

   for(i=0;i<m;i++)
      ListString.Display(HashTable[i]);
```



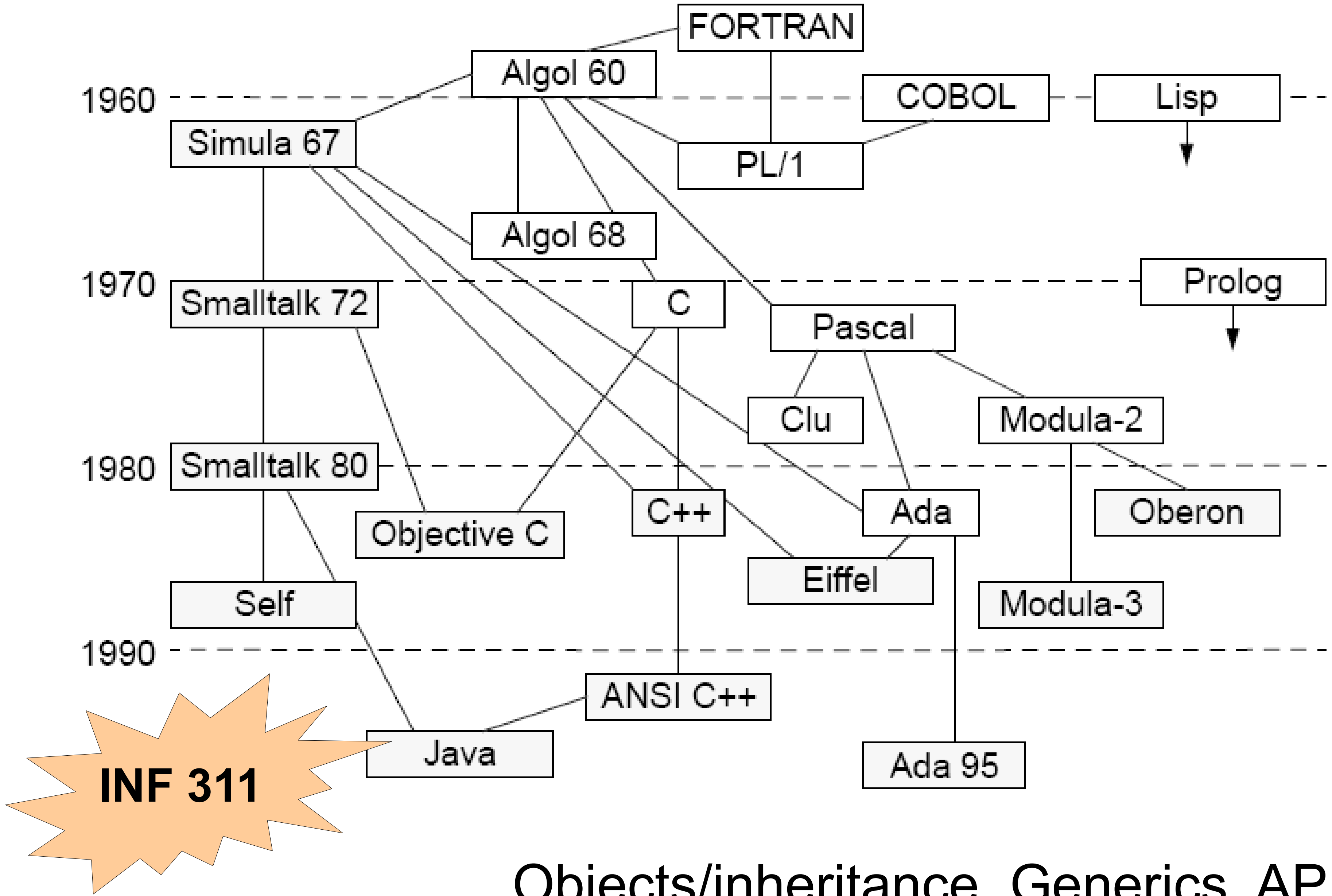
```
whale-->null
bee-->snake-->null
null
spider-->null
butterfly-->null
null
null
cow-->null
shark-->null
null
null
null
fish-->null
peacock-->cat-->null
null
tortoise-->dog-->null
horse-->null
flamingo-->null
null
pelican-->null
lion-->parrot-->null
elephant-->null
null
```

Executive summary of data-structures

Data-structure	Initializing	Search	Insert
Array	O(1)	O(n)	O(1)
Sorted array	O(n log n)	O (log n)	O(n)
Hashing	O(1)	Almost O(1)	Almost O(1)
List	O(1)	O(n)	O(1)

Arrays = Pertinent data-structure for almost static data sets
Lists = Data-structure for fully dynamic data sets

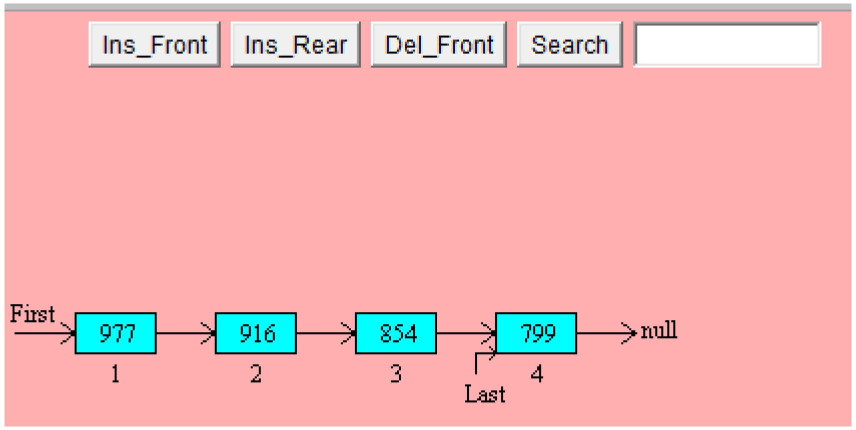
Java has many more modern features



Objects/inheritance, Generics, APIs

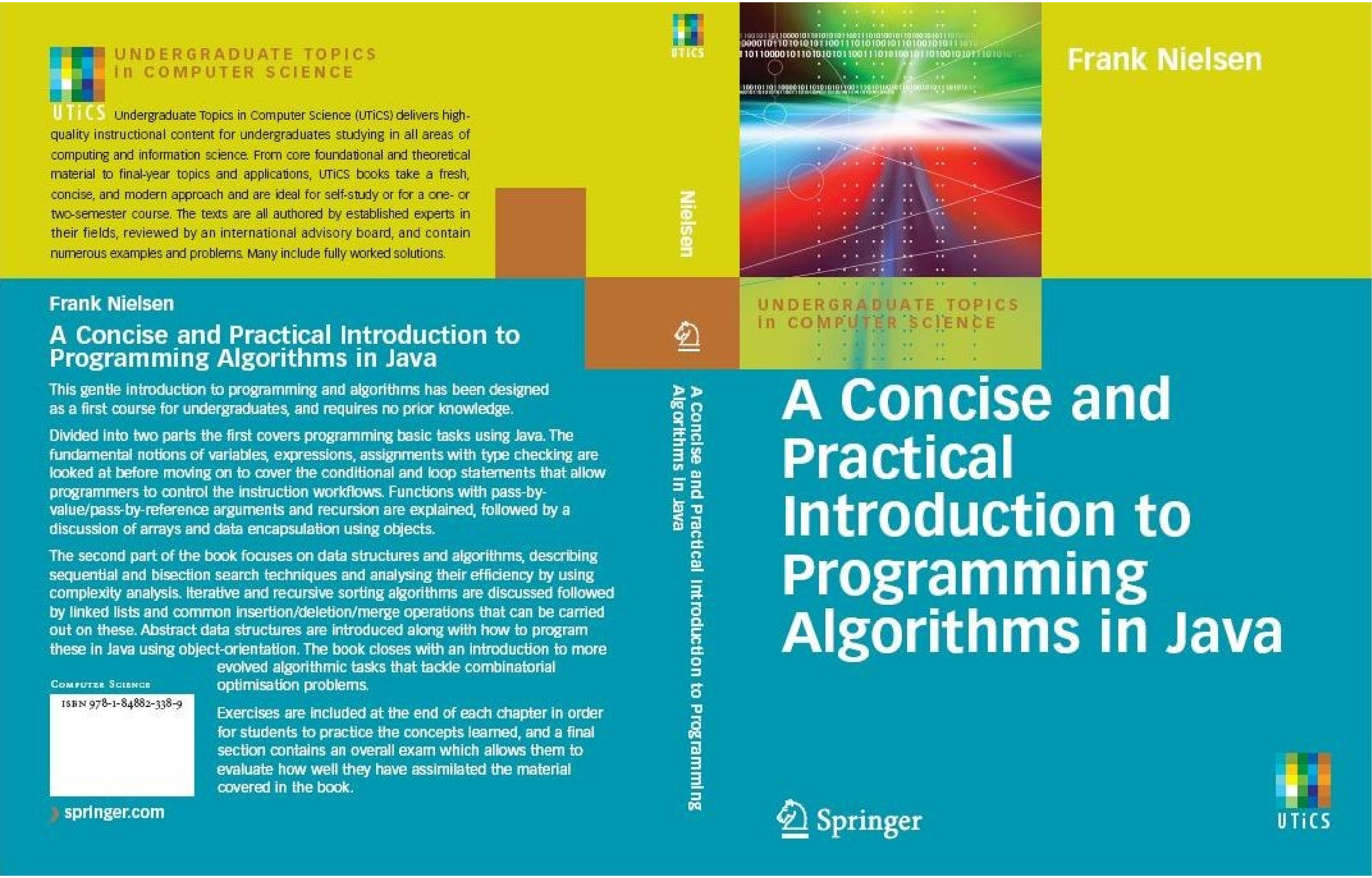
We presented the concept of **linked lists**:
A *generic abstract data-structure* with a set of plain (`while`) or recursive static functions.

In lecture 9, we will further revisit linked lists and other dynamic data-structures using the framework of objects and methods.



http://www.cosc.canterbury.ac.nz/mukundan/dsal/LinkListAppl.html

http://en.wikipedia.org/wiki/Linked_list



UNDERGRADUATE TOPICS
in COMPUTER SCIENCE

UTiCS Undergraduate Topics in Computer Science (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems. Many include fully worked solutions.

Frank Nielsen

A Concise and Practical Introduction to Programming Algorithms in Java

This gentle introduction to programming and algorithms has been designed as a first course for undergraduates, and requires no prior knowledge.

Divided into two parts the first covers programming basic tasks using Java. The fundamental notions of variables, expressions, assignments with type checking are looked at before moving on to cover the conditional and loop statements that allow programmers to control the instruction workflows. Functions with pass-by-value/pass-by-reference arguments and recursion are explained, followed by a discussion of arrays and data encapsulation using objects.

The second part of the book focuses on data structures and algorithms, describing sequential and bisection search techniques and analysing their efficiency by using complexity analysis. Iterative and recursive sorting algorithms are discussed followed by linked lists and common insertion/deletion/merge operations that can be carried out on these. Abstract data structures are introduced along with how to program these in Java using object-orientation. The book closes with an introduction to more evolved algorithmic tasks that tackle combinatorial optimisation problems.

Exercises are included at the end of each chapter in order for students to practice the concepts learned, and a final section contains an overall exam which allows them to evaluate how well they have assimilated the material covered in the book.

Computer Science
ISBN 978-1-84882-338-9

springer.com

UTiCS Nielsen

A Concise and Practical Introduction to Programming Algorithms in Java

Frank Nielsen

UNDERGRADUATE TOPICS
in COMPUTER SCIENCE

A Concise and Practical Introduction to Programming Algorithms in Java

Springer

UTiCS