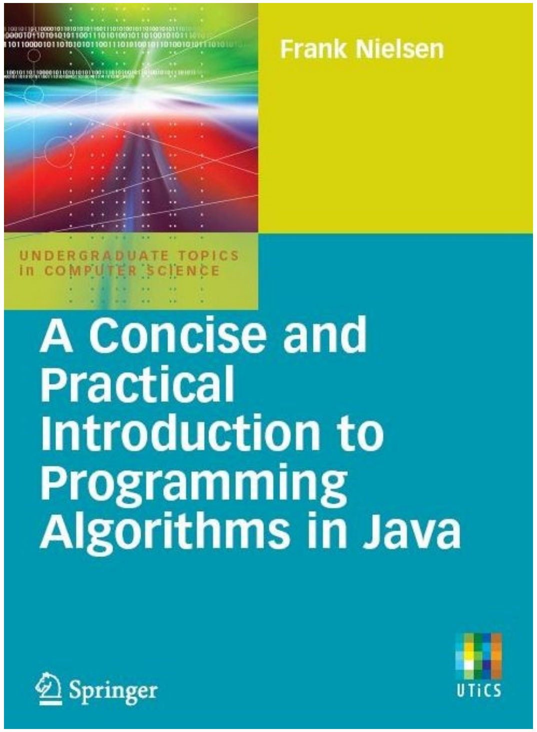
A Concise and Practical Introduction to Programming Algorithms in Java



Chapter 8: Data-structures and object methods

Frank NIELSEN

nielsen@lix.polytechnique.fr

Agenda

- FIFO data-structures
  (= First In First Out)
- Heap data-structures
- Non-static methods
  (= object methods)
- Revisiting lists (OO-style)

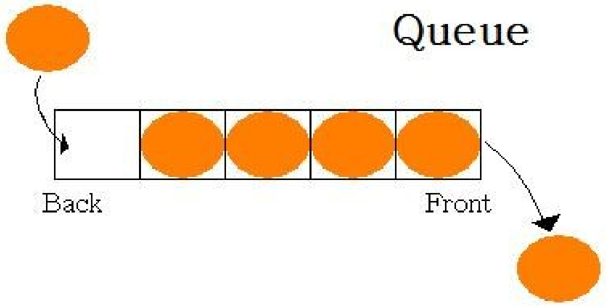
FIFOs: Mastering queues

- Objects are considered in turn, one by one

- Process each object according to their arrival time

- While objects are processed, others are **queued**

- First object should be first served!



<u>Basic examples:</u>

- Waiting in a queue at the post office.
- Printing « jobs> on a printer.

FIFO = First In First Out !

A basic solution

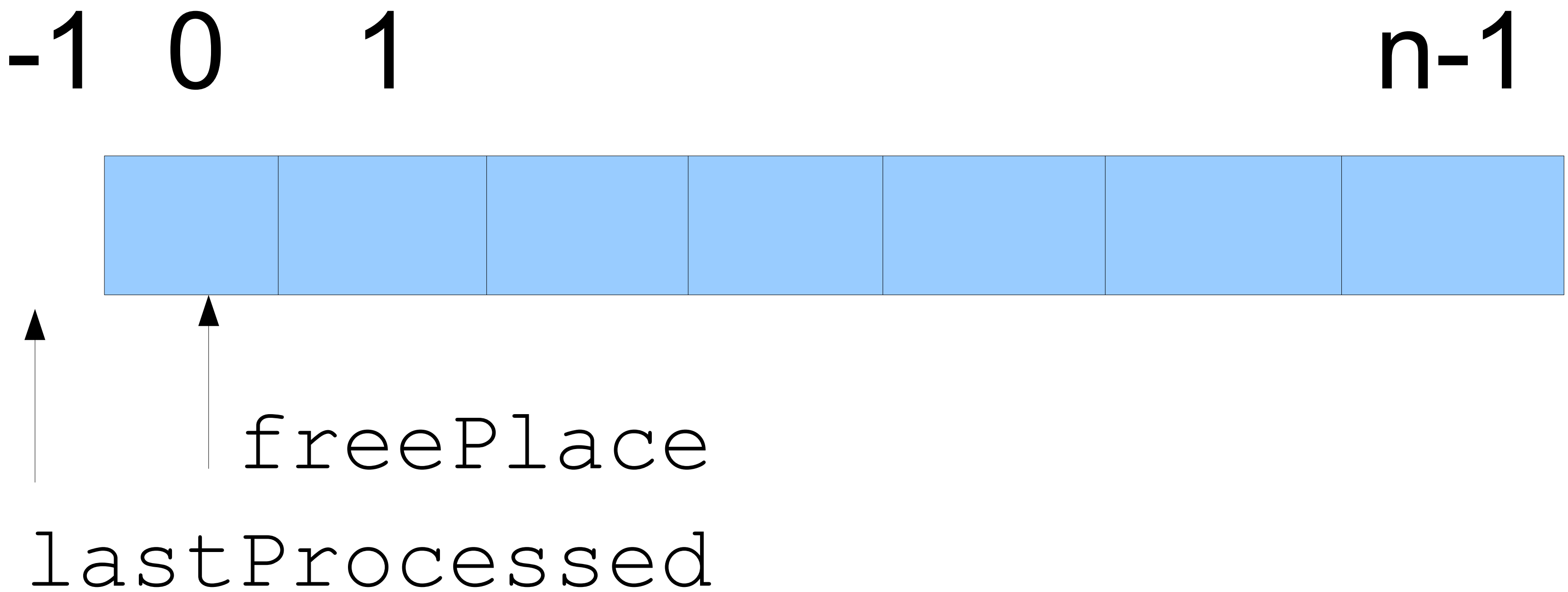
- Stack objects in an array as soon as they arrive
- To stack an incoming object, should know the **index** of the last location
- Should also know the **index** of the last processed object (so that we can process the next one)

While processing an object, others can *come in* the array (= **queued**)

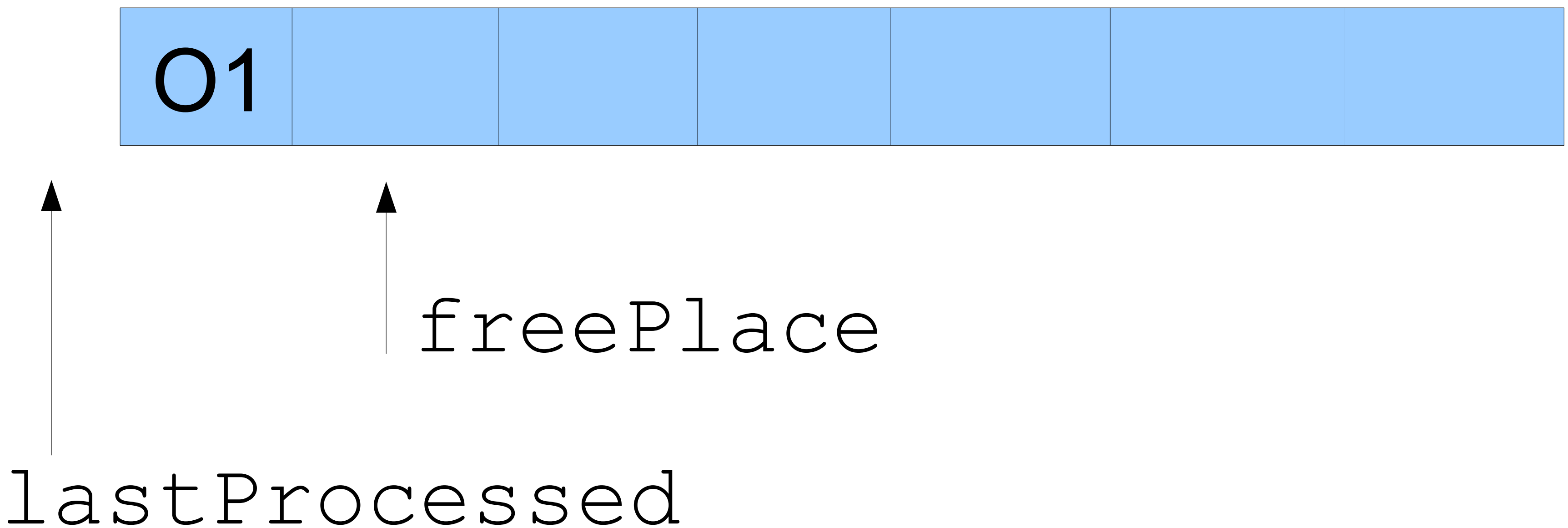
A basic solution

- An array: container `array` for storing objects
- **Two indices**: `lastProcessed` and `freePlace`
- To add an object `x`, we do `array[freePlace]=x` and we then increment: `freePlace++`
- To process an object, we increment `lastProcessed` and we process `array[lastProcessed]`

Visual depiction of a FIFO



FIFO: Queuing objects




```
array[freePlace++]=O1;
```

Queuing another object


```
array[freePlace++]=O2;
```

Processing *and* queuing


```
Process(array[lastProcessed++]);
array[freePlace++]=O3;
```

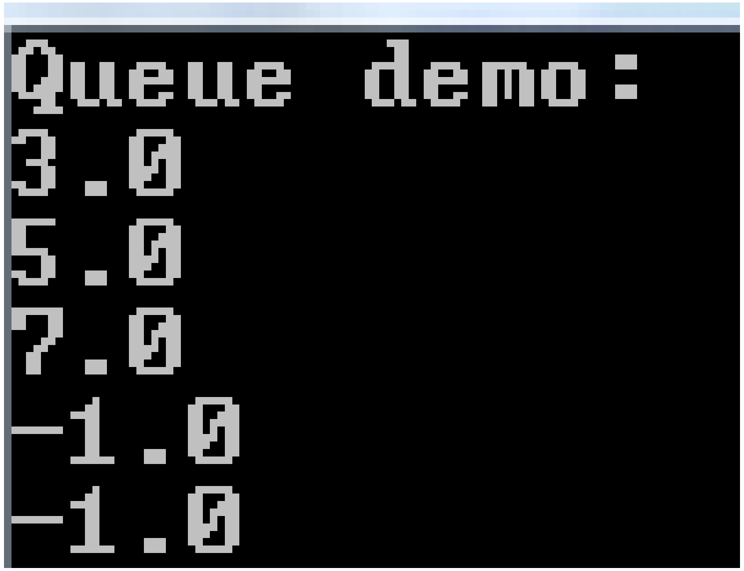
Processing and queuing can be done in parallel using threads

Programming queues

```
static int lastProcessed=-1;
static int freePlace=0;
static double[] container=new double[1000];

static void add(double Object)
{
   if (freePlace<1000)
   {container[freePlace]=Object;
   freePlace++;}
}

static double process()
{
if (freePlace-lastProcessed>1)
   {  // Do something here
      lastProcessed++;
      return container[lastProcessed];
   }
   else
      return -1.0; // special return code: no process
}
```

FIFO: First In First Out

```
class QueueDouble
{
static int lastProcessed=-1;
static int freePlace=0;
// Max objects is set to 1000
static double[] container=new double[1000];

// Stack in FIFO order
static void add(double a)
{...}

// Process in FIFO order
static double process()
{...}

public static void main(String[] arg)
{
System.out.println("Queue demo:");
add(3.0);
add(5.0);
add(7.0);
System.out.println(process());
System.out.println(process());
System.out.println(process());
System.out.println(process());
System.out.println(process());
```



Exercise: FIFO in action!

Let `A` be a **set** of integers such that:

- `1` belongs to `A`, and
- If `a` belongs to `A`, then `2*a+1` and `3*a` belongs to `A`

Question:

For a given `n`, display all integers less or equal to `n` that belong to `A`.

Programming queues

Start with a FIFO initialized with element `1`

Use a **boolean array** to store whether `a` belong to `A`
(= marks, tag elements)

For each element `a` of the FIFO do:

- Compute `2*a+1` and `3*a`
- Add them to the FIFO if they are less than `n`
...and not yet encountered (=marked)

Terminate when the FIFO is empty
Display all marked elements (=result)

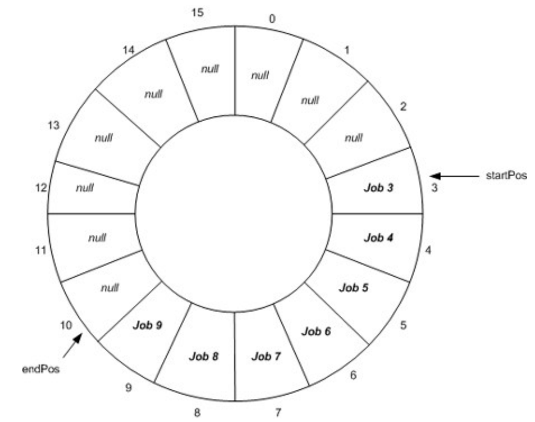
```java
final static int n=1000;   static int lastProcessed=-1;
static int freePlace=0; static int[] container=new int[n];
static boolean[] mark=new boolean[n];

static void add(int a)
{if (freePlace<n) {container[freePlace]=a;freePlace++;}}

static boolean Empty()
{ return ((freePlace-lastProcessed)==1);   }

static void process()
{int a;
      lastProcessed++; a=container[lastProcessed];
      if (a<n) mark[a]=true;
      if (2*a+1<n) add(2*a+1);
      if (3*a<n) add(3*a);
}
public static void main(String[] arg)
{int i;
for(i=0;i<n;i++) mark[i]=false;

add(1);
while(!Empty())
   process();

for(i=0;i<n;i++)
   {if (mark[i])
      System.out.print(i+" ");}
System.out.println("");
}
```

```
1 3 7 9 15 19 21 27 31 39 43 45 55 57 63 79 81 87 91 93 111 115 117 127 129 135
159 163 165 171 175 183 187 189 223 231 235 237 243 255 259 261 271 273 279 319
327 331 333 343 345 351 367 375 379 381 387 405 447 463 471 475 477 487 489 495
511 513 519 523 525 543 547 549 559 561 567 639 655 663 667 669 687 691 693 703
705 711 729 735 751 759 763 765 775 777 783 811 813 819 837 895 927 943 951 955
957 975 979 981 991 993 999
```

A few remarks on FIFOs

- Set beforehand the size of the array?
- Can wrap the array using mod `MAX_SIZE`
  (=circular ring, extend arrays, etc.)

...But how to check whether the queue is empty
... or full with circular arrrays?



Priority queues: Heaps (=tas)

- Objects are considered in turn
- Need to process them according to their **priorities**
- While processing an objects, other may arrive (= are being queued)
- Serve the object with the **highest priority first**

Examples:

- Ressource request
- Operating system tasks

Defining mathematically heaps

A heap is a sequence of integers:

$$t_1, t_2, \cdots, t_n$$

stored compactly in an **array** such that:

$$1 \leq i, j \leq n, \quad j = i/2 \;\Rightarrow\; t_j \geq t_i$$

i=7

For example:
37, 22, 31, 16, 17, 2, 23, 12, 6, 9

(heap of 10 elements)

j=(int)(7/2)=3

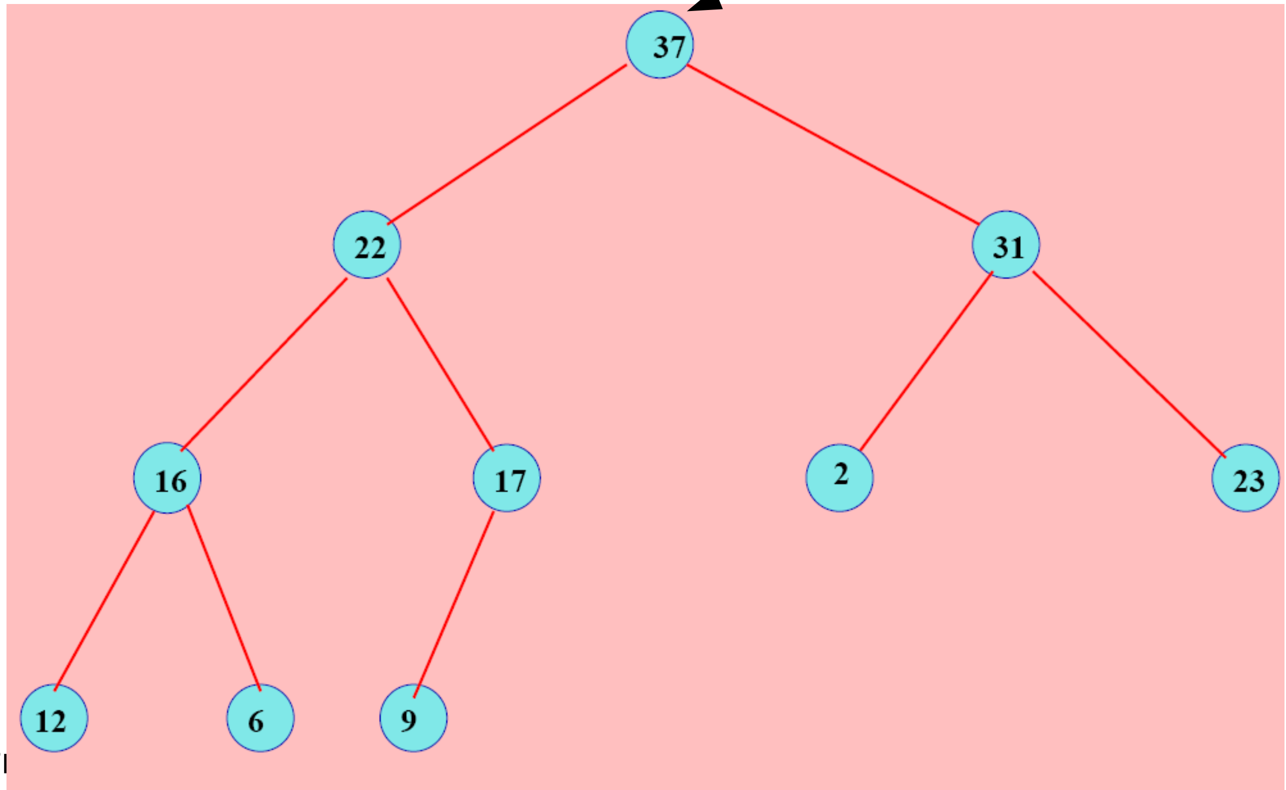
Drawing a heap

- Draw a heap as a **tree** (=special graph Vertex/Edge)
- Each node `i` contains a value `t[i]` and has

  0, 1 or 2 siblings that contain nodes of values less than its parent
- Node `i` has children `2i` and `2i+1`

`t[i]`

37, 22, 31, 16, 17, 2, 23, 12, 6, 9:
Read layer by layer,
from the root til the leaves



Storing and manipulating heaps

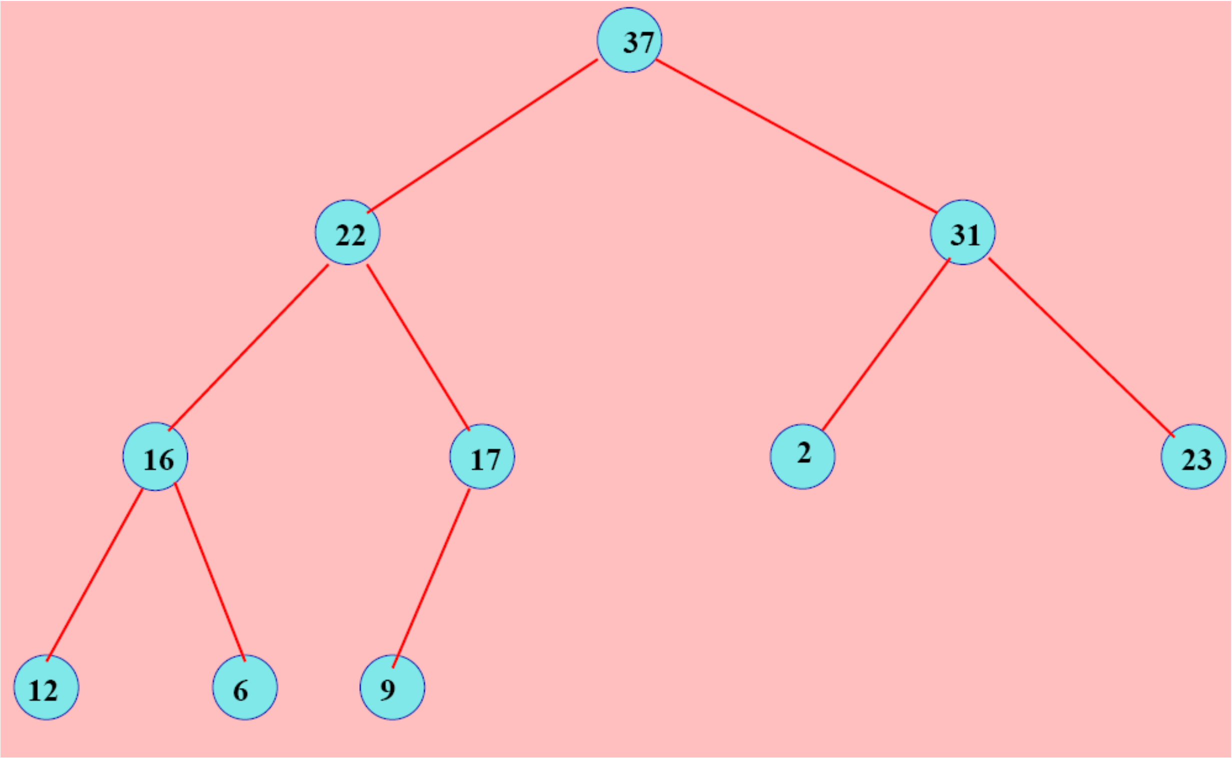
Easy to code with a linear array

```
public class Heap
{
int size;
int [] label;

static final int MAX_SIZE=10000;

Heap()
{
this.size=0;
this.label=new int[MAX_SIZE];
}


public static void main(String[] args)
{}
}
```

Fundamental property of heaps

Largest value is stored at the root of the tree, namely at the first element of the array.

```
static int maxHeap(Heap h)
{
return h.label[0];
}
```

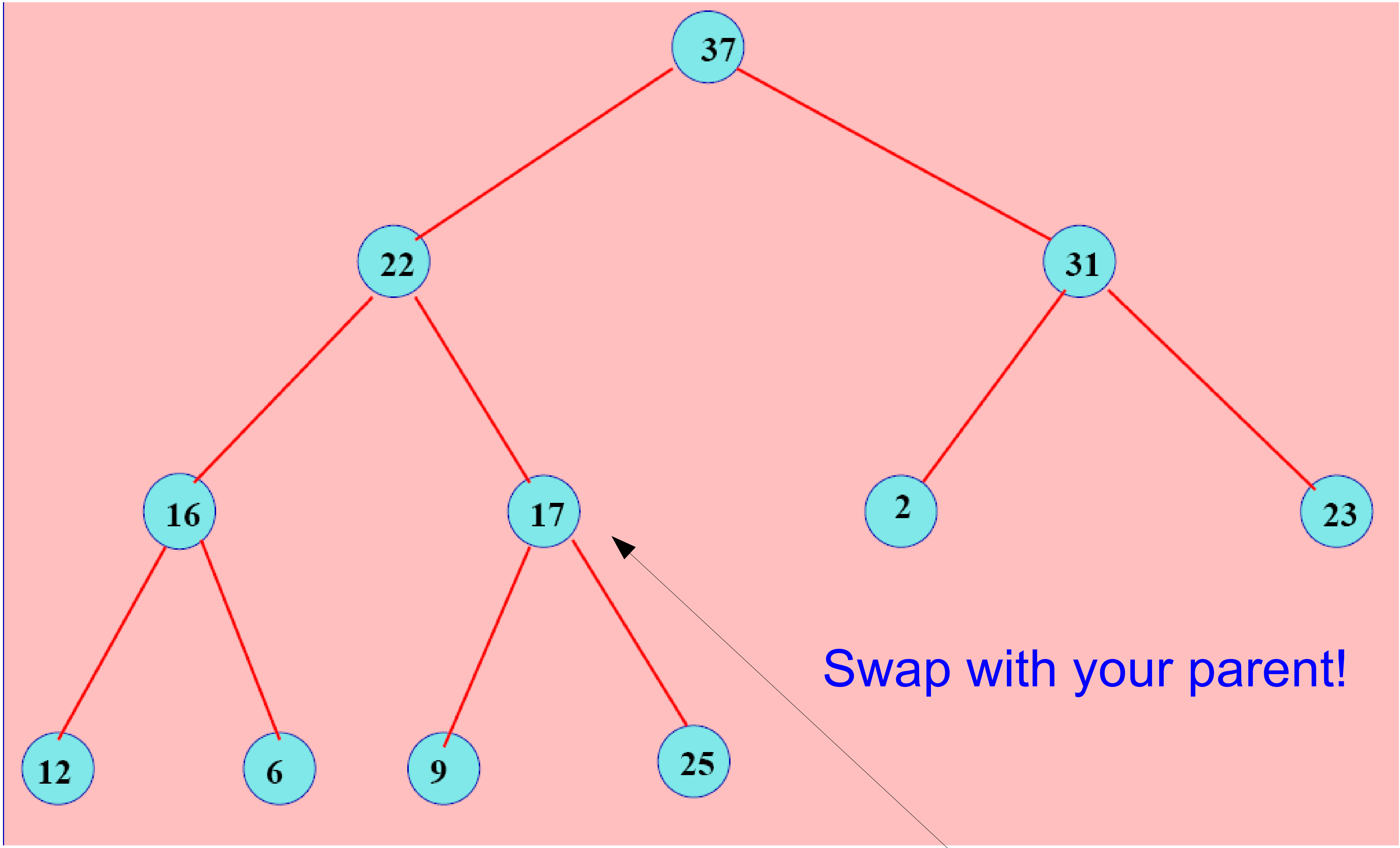


Adding an element in a heap

- Add the new element in position n (=n+1th element)...
- But the condition that the array is a heap is violated...
- So that we swap the element until...
.   ...it satisfies the **heap constraint**

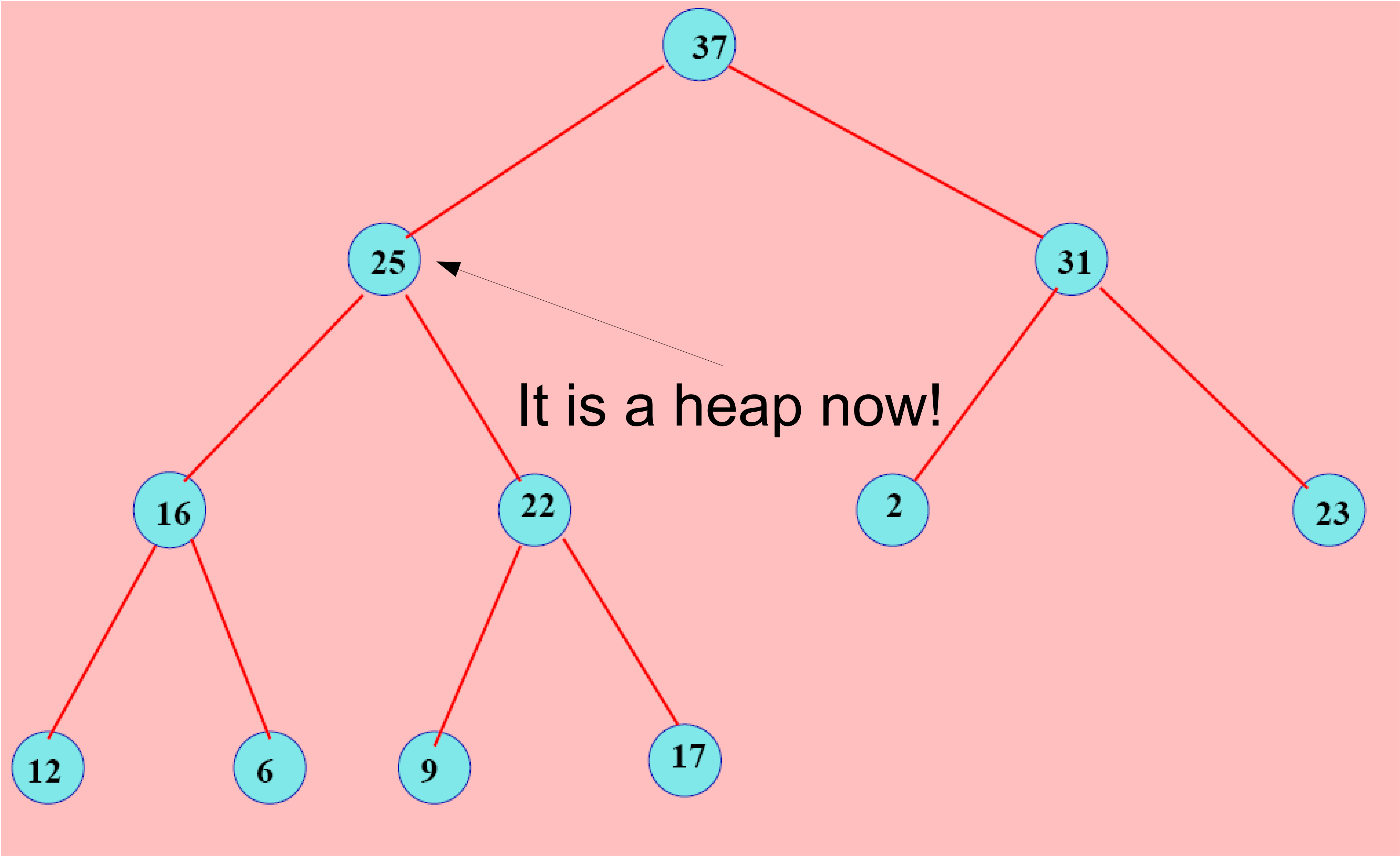
$$1 \leq i, j \leq n, \quad j = i/2 \;\Rightarrow\; t_j \geq t_i$$

Example: Add element 25



Not a heap anymore!!! 25>17

Add element 25... and swap!!!



Adding an element in the heap

```
static void addHeap(int element, Heap h)
{
h.label[h.size]=element;
h.size++;

int i=h.size;
int j=i/2;

   while (i>1 && h.label[i]>h.label[j])
   {
      int tmp=h.label[i];
      h.label[i]=h.label[j];
      h.label[j]=tmp;
      i=j; // swap
      j=i/2;
   }

}
```

Removing the largest element of a heap

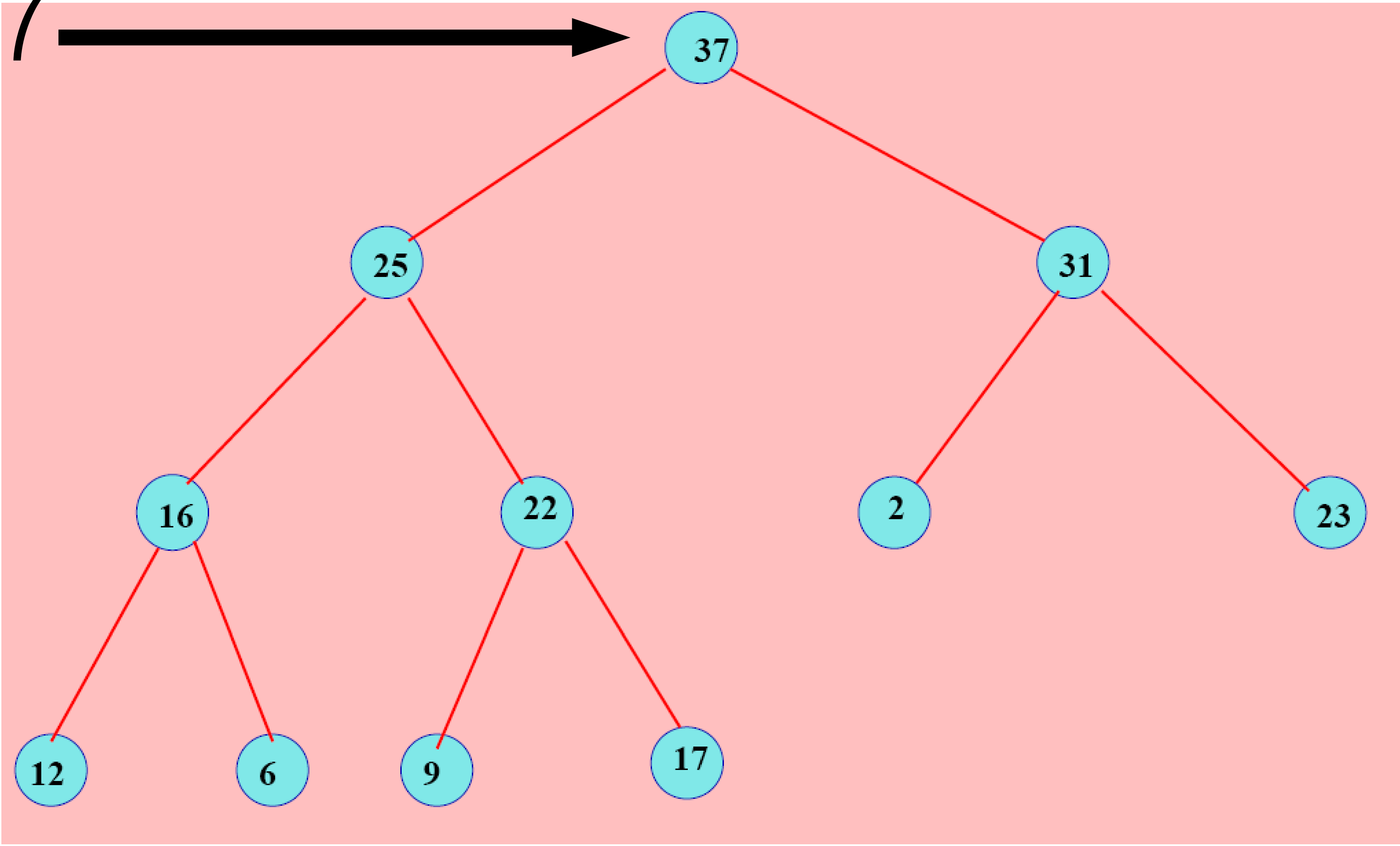
- We move the element at position (n-1) and put it at the root (position 0)

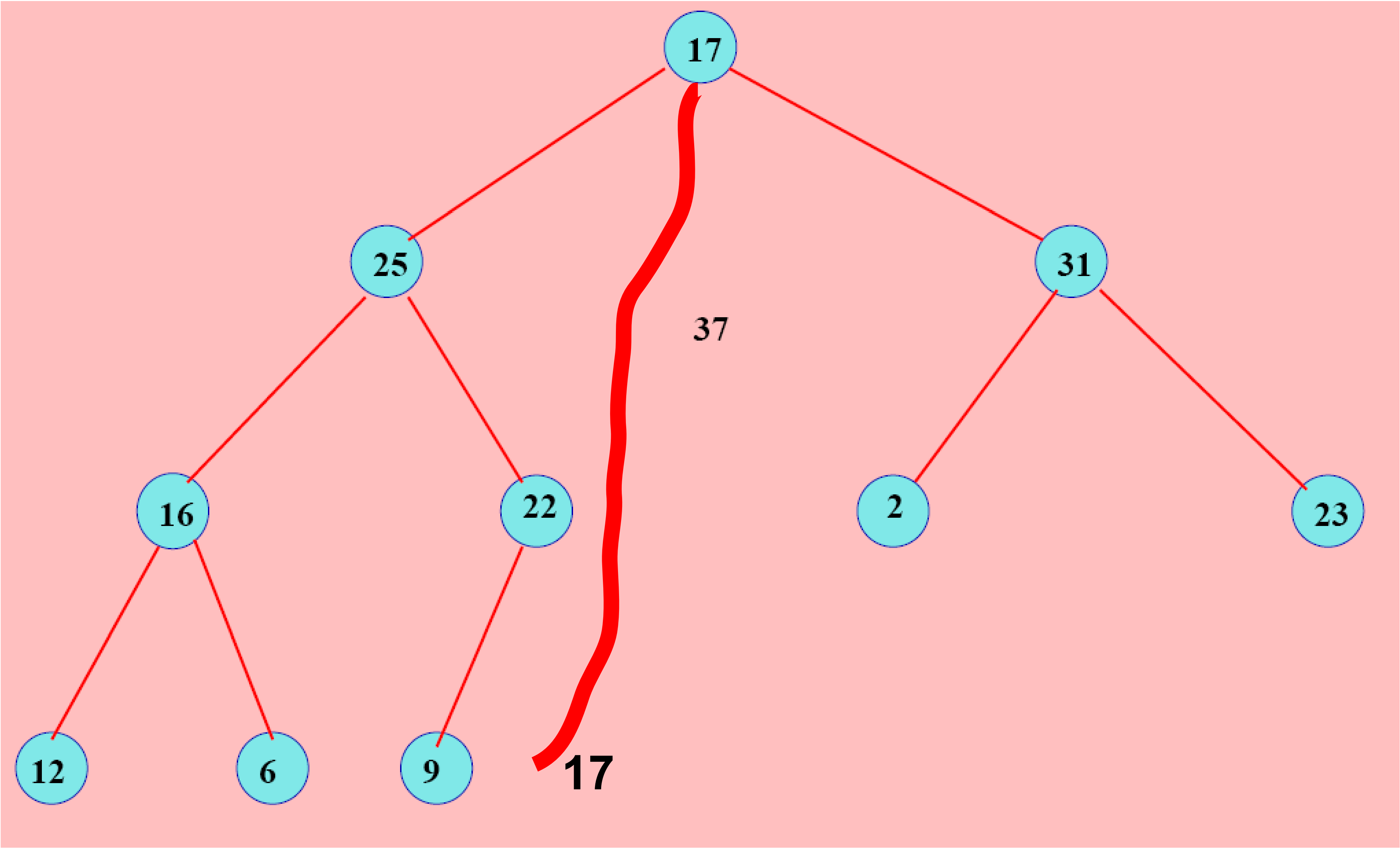
- But it is not anymore a heap...

- So we swap to the bottom until...
  ...the heap condition is satisfied

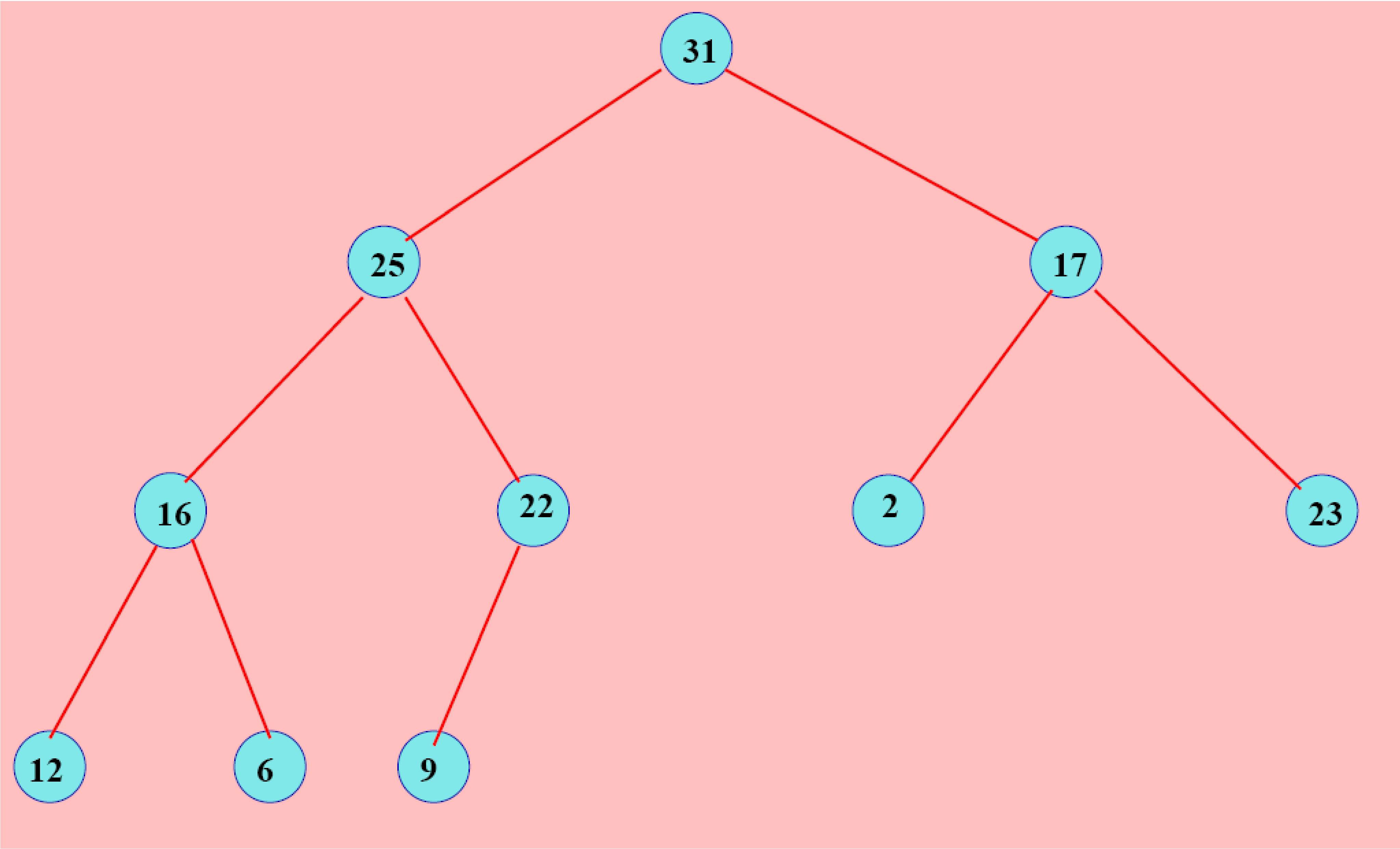
Removing the largest element of a heap

37




17
25
31
37
16
22
2
23
12
6
9
17

Then Swap parent-child...



Removing the largest element of a heap

```
static int removeHeap(int element, Heap h)
{
h.label[0]=h.label[h.size-1];
h.size--;

int i=0,j,k,tmp;

while(2*i<=h.size)
{
j=2*i;
if (j<h.size && h.label[j+1]>h.label[j])
   j++;

if (h.label[i]<h.label[j])
{tmp=h.label[i];
h.label[i]=h.label[j];
h.label[j]=tmp;
i=j;}
else break;
}
return h.label[h.size-1];
}
```

Non-static methods and objects

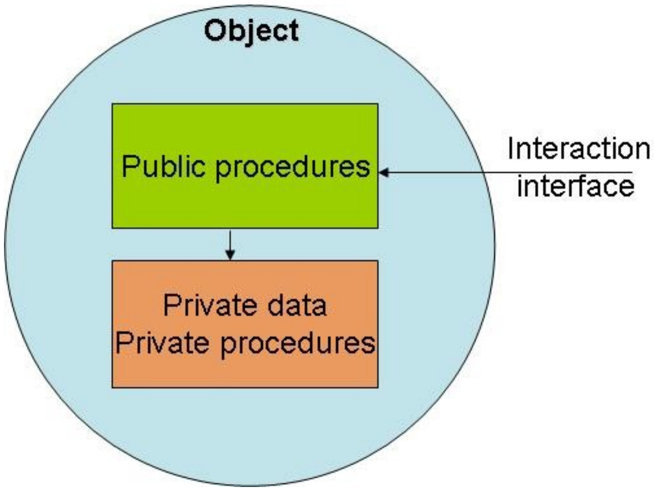
- Do not write static in front of the method
- Method is thus attached to the object for which it applies for
- For example, we prefer:
  `u.display()` rather than `display(u)`
  `u.addElement(a)` instead of `addElement(a,u)`
- To reference the object on which the method is called upon use `this`

Object-oriented programming paradigm (OO)

Object-oriented programming paradigm (OO)

- Design a software as a set of objects and methods applying on these objects

- Ask yourself first:
  - What are the objects?
  - What are the methods?



- Usually, a method often modifies the object (=fields) on which it applies for.
  (But not always, for example: `Obj.Display()`)

```
class Box
{
double width, height, depth;

Box(double w, double h, double d)
{
this.width=w; this.height=h; this.depth=d;
}

double Volume()
{return this.width*this.height*this.depth;}
}

class OOstyle
{
static double Volume(Box box)
{return box.width*box.height*box.depth;}

public static void main(String[] s)
{
Box myBox=new Box(5,2,1);

System.out.println("Volume by static method:"+Volume(myBox));
System.out.println("Volume by object method:"+myBox.Volume());

}
}
```

OO style: object methods versus static functions

Object (non-static) method

```
class Toolkit
{
static final double PI=3.14;

static double Square(double x)
{return x*x;}

static double Cubic(double x)
{return x*x*x;}
}

class StaticFuncStyle
{

public static void main(String[] s)
{
double radius=0.5;

double volSphere=(4/3.0)*Toolkit.PI*Toolkit.Cubic(radius);
double areaDisk=Toolkit.PI*Toolkit.Square(radius);

}
}
```

Static methods are useful for defining:
- Constants
- Basic functions

....in a library.

Heaps revisited in Object-Oriented style

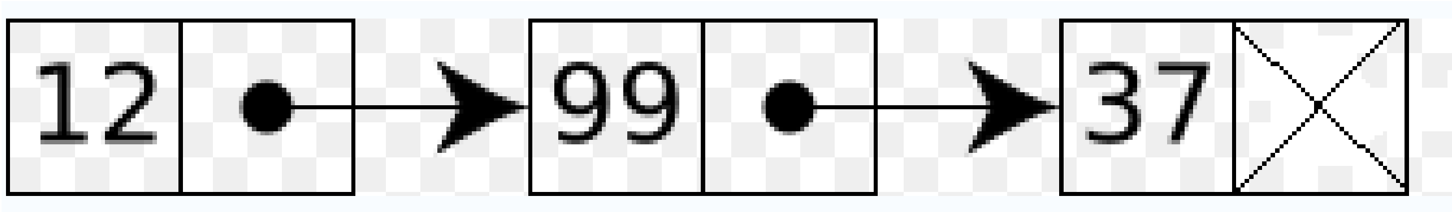
```
int maxHeap()
{
return this.label[0];
}


void add(int element)
{
...
}

void removeTop()
{
...
}
```

Observe that the keyword `static` has disappeared

List in object-oriented style

- A cell stores information (say, an integer) and point/refer to the next one.

- Pointing to another cell means storing a reference to the corresponding cell.



Pay special attention to `null` !!!

- Remember that we cannot access fields of the `null` object
- Throw the exception `nullPointerException`
- Thus we need to check whether the current object is `null` or not, before calling the method
- In the reminder, we consider that all lists (also the void list) contain a first cell that stores no information.

Revisiting the linked list (OO style)

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

List(int el, List l)
{
this.element=el;
this.next=l;
}

static List EmptyList()
{
return new List(0,null);
}

boolean isEmpty()
{
return (this.next==null);
}
```

Revisiting the linked list (OO style)

```
int length()
{
List u=this;
int res=0;
while(u!=null) {res++;u=u.next;}
return res-1;
}

boolean belongsTo(int el)
{
List u=this.next;
while(u!=null)
    {
    if (el==u.element) return true;
    u=u.next;
    }

return false;
}
```

Revisiting the linked list (OO style)

```
void add(int el)
{List u=this.next;
this.next=new List(el,u);
}


void delete(int el)
{
List v=this;
List w=this.next;

while(w!=null && w.element !=el)
    {
    v=w;
    w=w.next;
    }
if (w!=null) v.next=w.next;
}
```

Revisiting the linked list (OO style)

```
void display()
{
List u=this.next;
while(u!=null)
   {System.out.print(u.element+"->");
   u=u.next;}
System.out.println("null");
}

static List FromArray(int [] array)
{
   List u=EmptyList();
   for(int i=array.length-1; i>=0; i--)
      u.add(array[i]);
   return u;
}
```

Revisiting the linked list (OO style)

```
public static void main(String[] args)
{
int [] array={2,3,5,7,11,13,17,19,23};

List u=FromArray(array);

u.add(1);

u.display();

u.delete(5);
u.display();

System.out.println(u.belongsTo(17));
System.out.println(u.belongsTo(24));
}
```

```
1->2->3->5->7->11->13->17->19->23->null
1->2->3->7->11->13->17->19->23->null
true
false
```

Stacks (LIFO): Last In First Out

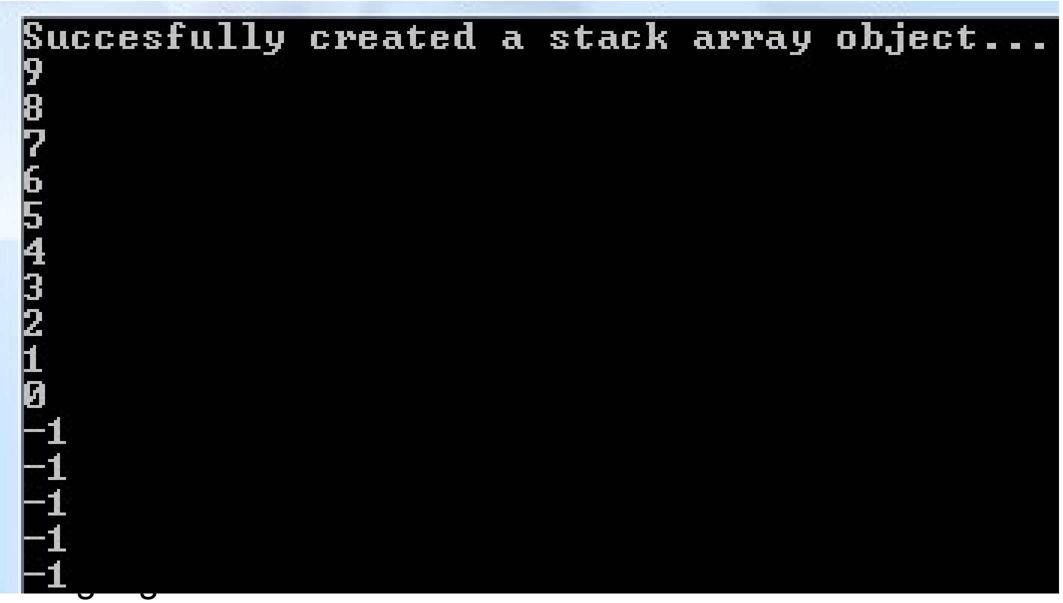
Two basic operations for that data-structure:

- `Push`: Add an element on top of the stack
- `Pull`: Remove the topmost element

Stacks (LIFO) using arrays

```
class StackArray
{
int nbmax;
int index;
int [ ] array;

// Constructors
StackArray(int n)
{
this.nbmax=n;
array=new int[nbmax]; index=-1;
System.out.println("Succesfully created a stack array object...");
}

// Methods
void Push(int element)
{
if (index<nbmax-1)
   array[++index]=element;   }

int Pull()
{
if (index>=0 ) return array[index--];
else return -1;
}
```

```
class DemoStack{

public static void main(String [] args)
{
   StackArray myStack=new StackArray(10);
   int i;

   for(i=0;i<10;i++)
       myStack.Push(i);

   for(i=0;i<15;i++)
       System.out.println(myStack.Pull());

}
```



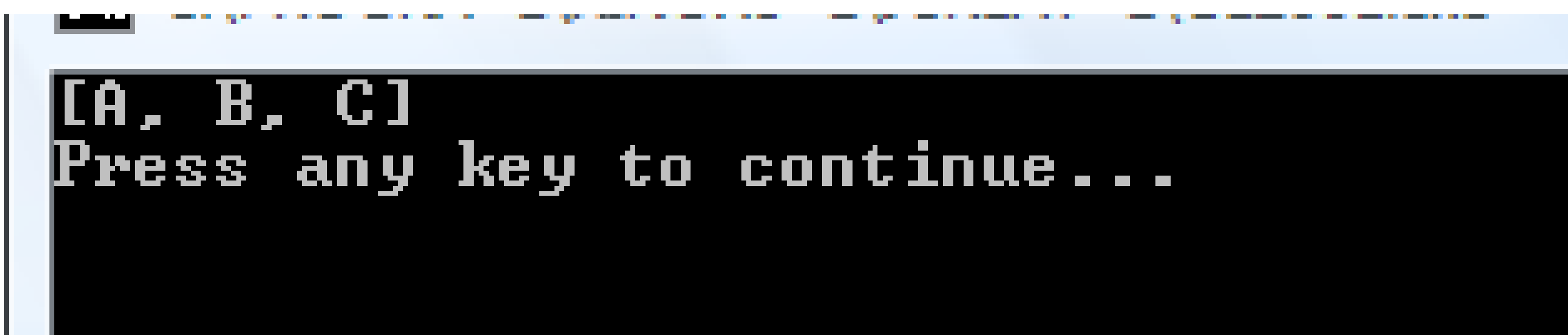
Stacks (LIFO) using linked lists

```
class List
{
int element;
List next;

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

List insertHead(int el)
{
return new List(el,this);
}

}
```

```
class Stack
{
List list;

Stack()
{
list=null;
}

void Push(int el)
{
if (list!=null)
      list=list.insertHead(el);
      else
      list=new List(el,null);
}

int Pull()
{int val;
if (list!=null)
   {val=list.element;
      list=list.next;}
      else val=-1;

return val;
}
}
```

Stacks: API

```java
// Use a Java package here
import java.util.Stack;

public class MainClass {

public static void main (String args[]) {
Stack s = new Stack();
s.push("A");
s.push("B");
s.push("C");

System.out.println(s);
}
}
```



Stacks (LIFO) using linked lists

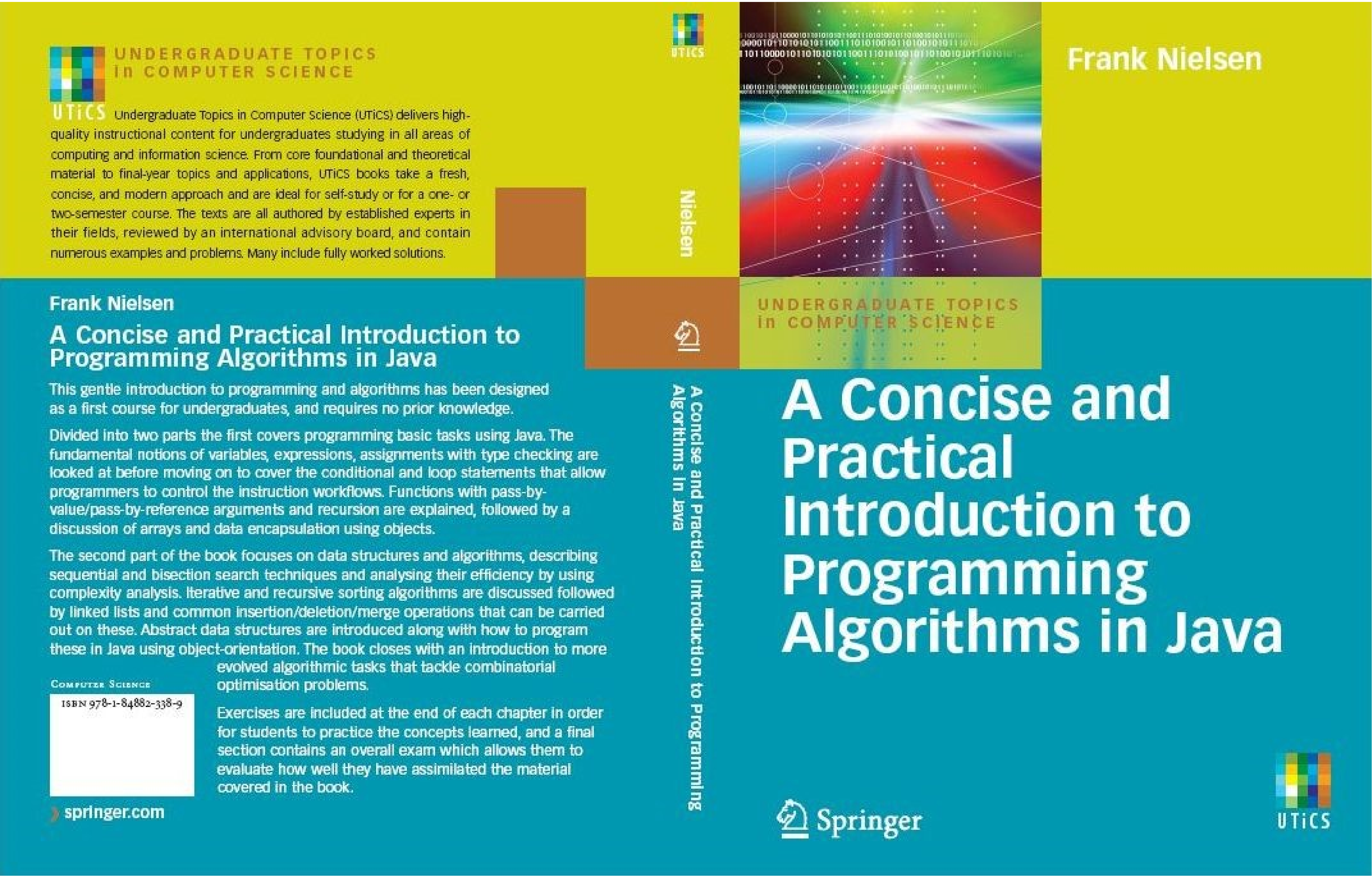
```
class DemoStackList
{

public static void main(String [] args)
{
   Stack myStack=new Stack();
   int i;

   for(i=0;i<10;i++)
       myStack.Push(i);

   for(i=0;i<15;i++)
       System.out.println(myStack.Pull());

}

}
```

Notice: Same code as StackArray demo program.

Static functions versus methods

- **Static (class) functions**:
  Access static/local variables only.
  « class methods »
  Potentially many arguments in functions

- **Object methods**:
  Access object fields (using `this`)
  Access (class) static variables too.
  Objects are instances of classes
  Data encapsulation
  (=functions with limited number of arguments)
  Constructor (= field initialization)


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