Algorithms and DataStructures - Niet-Numerieke Algoritmen

An introduction

Lubos Omelina, Bart Jansen

2022-2023

Preliminaries

Contact Details

Dr. Lubos Omelina
PL9.1
lomelina@etrovub.be

Panagiotis Gonidakis
PL9.1
pgonidak@etrovub.be

Katka Kostkova
PL9.1
kkostkova@etrovub.be

Prof. Bart Jansen
PL9.1
bjansen@etrovub.be

Expected knowledge

- You know very basic algebra.
- You know the basics of JAVA and can write simple programs in JAVA.
- ... Or you start working on that immediately.

Summary

The goal of this course is to truly understand a set of basic data structures and algorithms, by implementing them yourself in concrete computer programs. These algorithms and data structures are the building blocks for more complex algorithms and data structures in other courses and are required in almost all day to day programming.

Broader Semester Objectives

The goal is to learn how to write computer programs in one semester.
This is very ambitious. This involves learning to reason very analytically.
This involves learning a basic level of JAVA and Python.

Expected efforts

- You program as much as ever possible.
- This means from now on you program every day until the end of the semester !!!

Course Organization

- 24 contact hours Seminar, Exercises or Practicals
- 24 contact hours Lecture
- 36 contact hours Independent or External Form of Study

Reminder

- You program as much as ever possible.
- 36 hours effort in 6 weeks is 6 hours per week. This is close to one hour per day !!!

The same, but in other words ...

- This course is about programming skills and insight, not about knowledge.
- There is almost nothing to memorize.
- At the written exam you will make exercises as in the exercise sessions.
- "open book" means that we provide correct starting code.
- I can not teach you to program or teach you insights in data structures in 24 hours. You have to learn this by practicing a lot.

Course Topics

- Some theoretical concepts.
- Searching in arrays, building linked and double linked lists, searching in linked and double linked lists.
- Methods for sorting arrays and lists.
- Stacks, queues and priority queues.
- Methods to represent trees, for searching trees and to balance trees.
- Methods to represent graphs and to search graphs.

Organization of the exercise sessions

- In the computer room, we write computer programs.
- You can work at your own speed, not everyone has the same amount of programming experience.
- However, I expect you to have a minimal basic knowledge of JAVA.
- Every week, we start with a new topic. So, what you should complete what you did not finish.
- These assignments do not have a unique correct answer! You can verify the correctness of your program by investigating the output of your program.

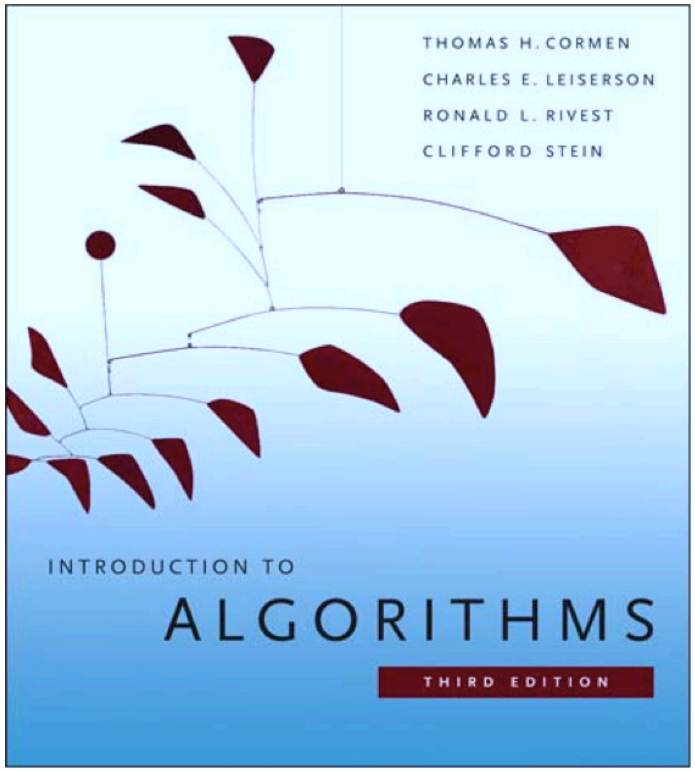
Course Material

1. Slides used in class. See Canvas.
2. The code I provide (See Canvas) and the code you write
3. The course text. See Canvas.
4. Your course notes. Without your notes, the material might be not understandable.
5. Any JAVA book. (I have Big Java by Cay Horstman)
6. If you find it helpful, get a book on algorithms and data structures in JAVA.

Reference book

A reference book far beyond the scope of this course is:
Introduction to Algorithms, Second Edition by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein
This is the type of book you can take with you the next ten years.
This is not the type of book which will help you understand basic problems with JAVA.

Reference book



Exam

The exam consists of two parts:

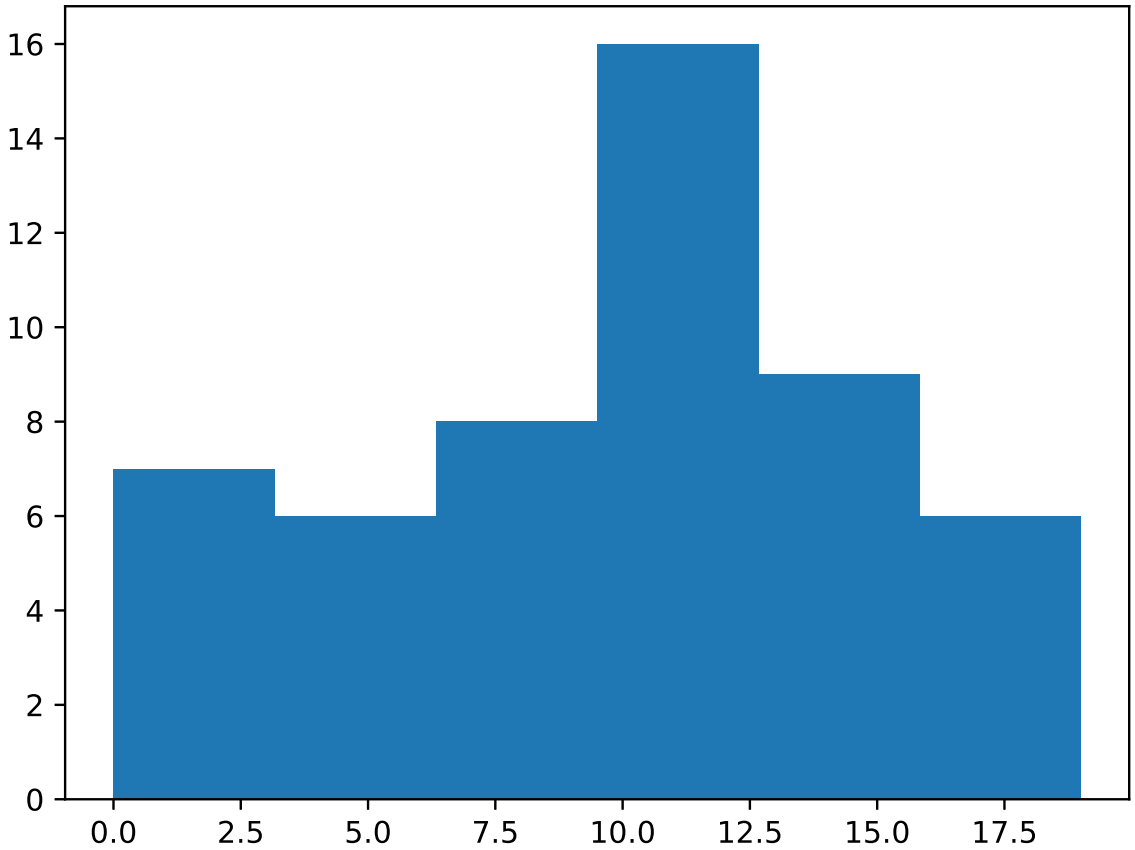
1. A written exam (= 60%). You get questions on the data structures and algorithms studied in class or on SIMILAR ones.
2. A project, with oral discussion(= 40%). You get questions on YOUR implementation of a small program that is based on the material seen in class.

Project

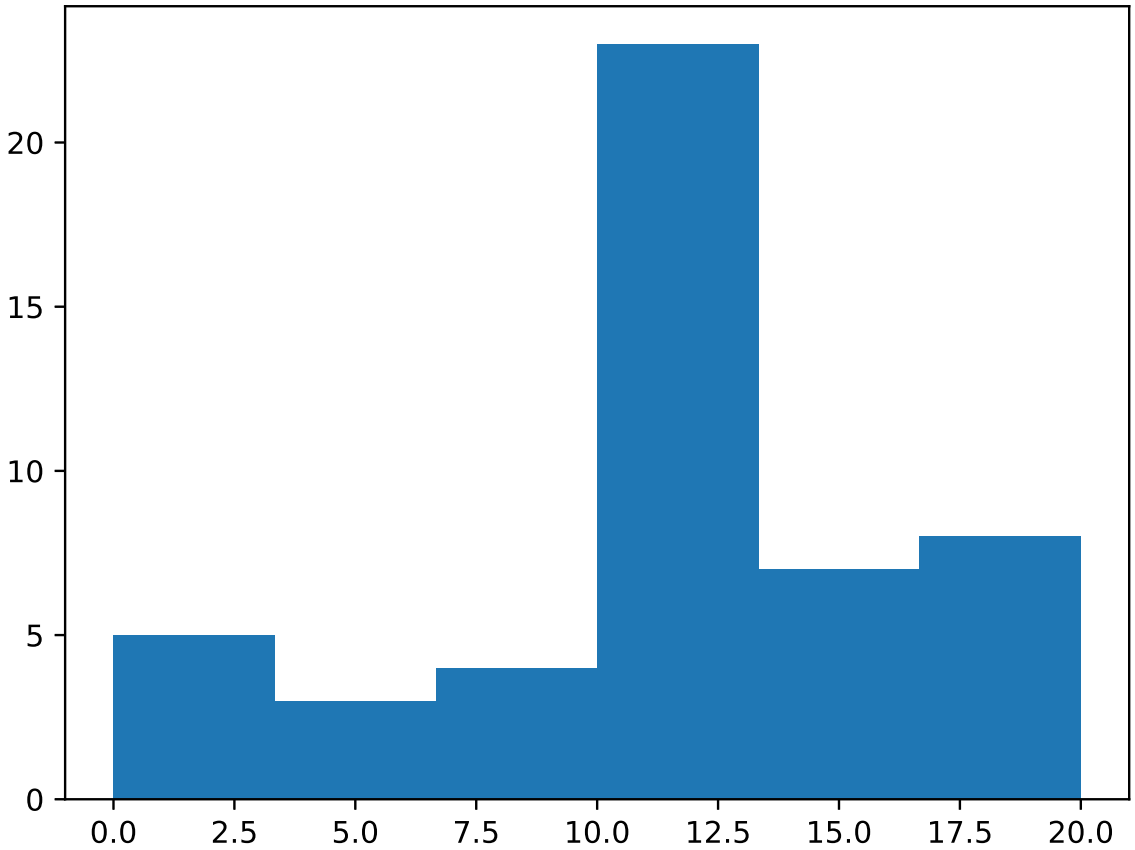
The project will force you to think about the proper selection and usage of the different algorithms and data structures. You have two options:

1. We provide you a description of the assignment. We have it split in four parts that match with the progress of the course.
2. You design your own project. More freedom, but less structure. Individual discussions to set goals. A game

Written exam: 9.7 on average

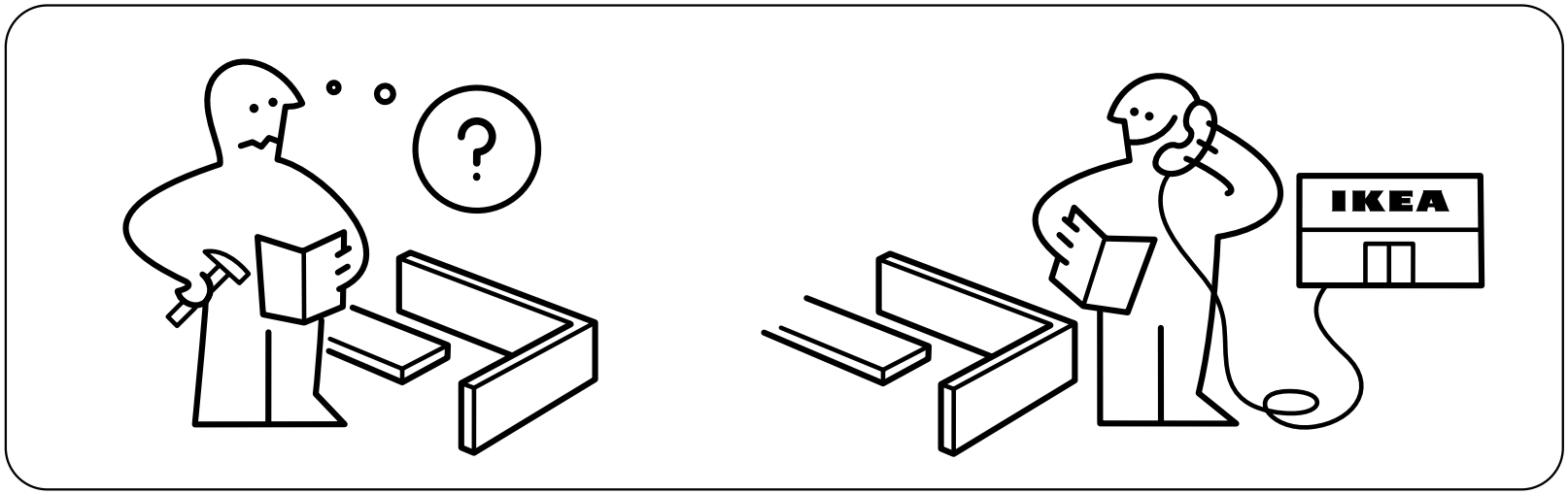


Project: 11.4 on average



Warning

- Week after week, we will build more complex code, based on previous code.
- The advantage is that you will be able to build some complex data structures and algorithms very soon.
- The danger is that you get lost soon!
- The required programming skills and the concepts get more difficult at a very fast pace.
- It is your duty and not mine to do everything you can not to get lost! Be proactive, work hard, ask for help, ...
- If you get lost in this semester (this actually happens for many students), do not wait for a miracle but do something about it.


?
IKEA


How do you program so well?
Practice.
It must be an innate gift...
A gift from God...
It's practice.
I'll never understand how some people are so talented...
A mystery...
Practice.
©Sarah Andersen

Algorithms? Data structures?

- Algorithms are recipes to compute something
  - How to test whether a number is prime or not?
  - How to search for a given word in a text
  - How to find the shortest path between 2 cities in a graph?
  - ...
- Example: The mathematical definition of prime numbers does not tell you how to test whether a given number is prime in an efficient way.
- So, this course will teach you a basic repertoire of methods to solve simple problems.

Prime numbers

Theorem 1

A number $p \in \mathbb{N}$ is prime if it can only be divided by 1 and itself.

This suggests we should check whether p can be divided by $2, 3, 4, 5, 6, ..., p-1$ This is actually stupid ... but a real efficient solution is not obvious.

Why we study algorithms and data structures

- Fundamental recipes of computer science.
- Understanding is necessary to use them as building blocks for extending them yourself.
- We look at their implementation and not only at their use.

Some JAVA preliminaries

The very first JAVA program

```java
public class MyFirstJavaProgram {

   /* This is my first java program.
    * This will print 'Hello World' as the output
    * This is an example of multi-line comments.
    */

   public static void main(String []args) {
      // This is an example of single line comment
      /* This is also an example of single line comment. */
      System.out.println("Hello World");
   }
}
```

Variables and numbers

```
public class MySecondJavaProgram {

    public static void main(String []args) {
        // for now we put all code here and don't ask why

        int x = 5;
        int y = 6;
        int z = x + y;
        double j = 1.21 + z;

        System.out.println(z);
        System.out.println(j);
    }
}
```

Conditions

```
public class MyThirdJavaProgram
{
    public static void main(String []args)
    {
        // for now we put all code here and don't ask why
        int x = 5;
        if(x > 6)
        {
          System.out.println("Too small");
        }
        else
        {
          System.out.println("Large enough");
        }
    }
}
```

Repetitions

```
public class MyFourthJavaProgram
{
    public static void main(String []args)
    {
        // for now we put all code here and don't ask why
        for(int x = 0; x < 10; x++)
        {
            System.out.println(x);
        }
    }
}
```

Repetitions (2)

```
public class MyFifthJavaProgram
{
    public static void main(String []args)
    {
        // for now we put all code here and don't ask why
        int x = 10;
        while(x>6)
        {
            System.out.println(x - 1);
        }
    }
}
```

Functions / methods

```
public class MySixthJavaProgram
{
    public static int sum(int x, int y)
    {
        return x + y;
    }

    public static void main(String []args)
    {
        int result = sum(1,2);
        System.out.println(result);
    }
}
```

Arrays

```
public class MySeventhJavaProgram
{
    public static void main(String []args)
    {
        int data[] = new int[3];
        data[0] = 3;
        data[1] = 6;
        data[2] = 9
        System.out.println(data[0]);
    }
}
```

Vectors

Vectors

- One of the simplest data structures.
- Extends the functionality of the array with various useful methods:
  - adding elements
  - accessing the elements
  - removing all elements
  - removing specific elements
  - ...

Functionality

- getFirst: returns the first element of the vector.
- getLast: returns the last element of the vector.
- addFirst: adds an element to the front.
- addLast: adds an element to the end.
- get: gets an element at a given position.
- size: returns the nr of elements in the vector.
- contains: verifies whether the vector contains a given element or not.
- removeFirst: removes the first element.
- removeLast: removes the last element.
- ...

A simple Vector implementation

```
public class Vector
{
    private int data[];
    private int count;

    public Vector(int capacity){  }

    public int size(){   }

    public int get(int index){ }

    public void addFirst(int item){  }
}
```

A simple Vector implementation

```
public class Vector
{
    private int data[];
    private int count;

    public Vector(int capacity)
    {
        data = new int[capacity];
        count = 0;
    }

    public int size()
    {
        return count;
    }

    ...
}
```

A simple Vector implementation

```
public class Vector
{
    ...

    public int get(int index)
    {
        return data[index];
    }

    public void addFirst(int item)
    {
      for(int i = count; i > 0;i--){
       data[i] = data[i-1];
      }
      data[0] = item;
      count++;
    }
}
```

Example for using the Vector

```
public class TestClass
{
    public static void main(String[] args)
    {
        Vector numbers = new Vector(10);
        numbers.addFirst(1);
        numbers.addFirst(2);
        numbers.addFirst(3);

        System.out.println(numbers.get(0));
        System.out.println(numbers.get(1));
    }
}
```

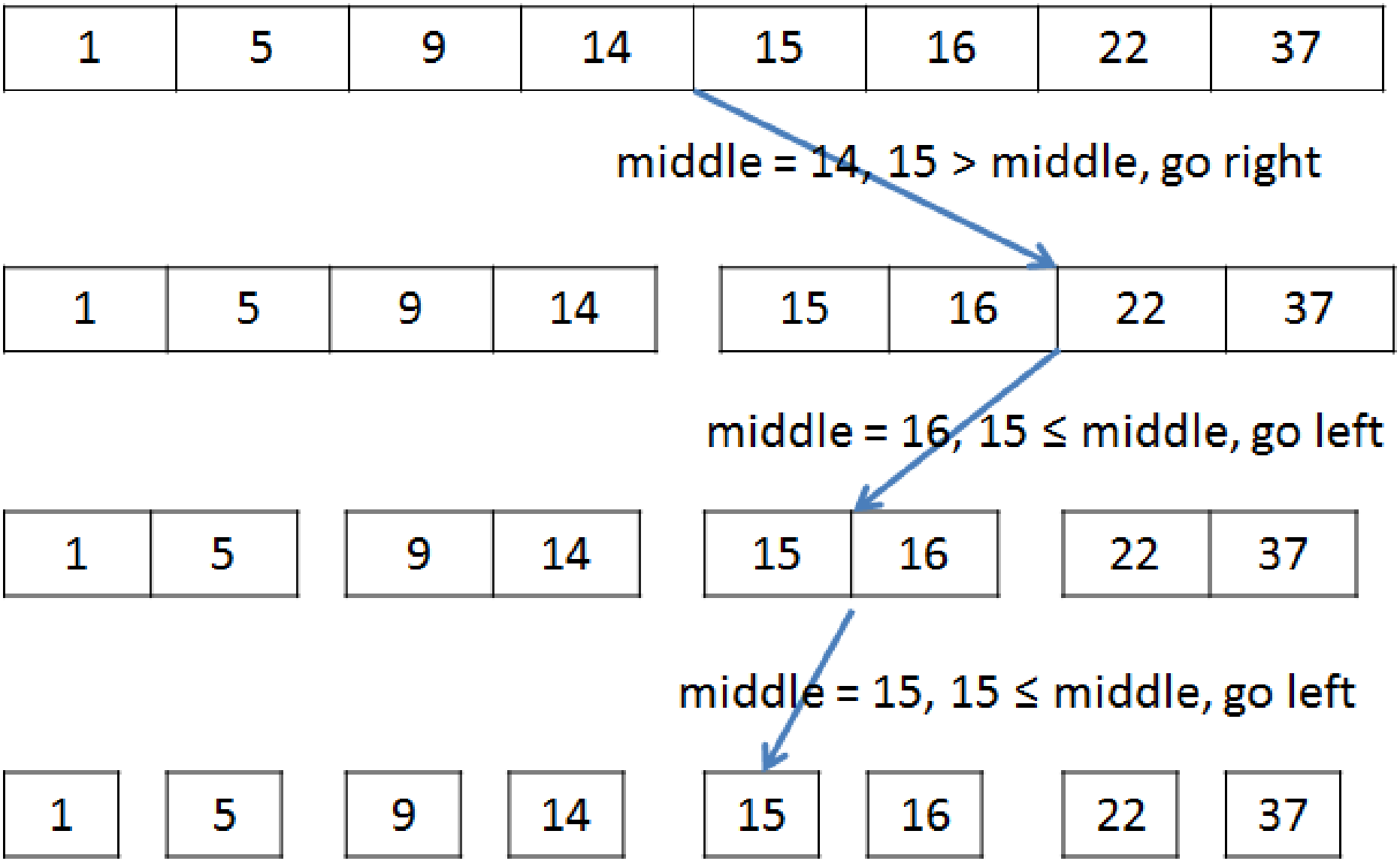
Binary Search

A divide and Conquer approach

- When the vector is sorted, searching for a given element can be done more efficiently.
- First check whether the element is supposed to be in the first or the second half of the vector.
- This way only half of it needs to be searched for.
- In the right half of the vector, check whether the element should be in the first or the second half.
- ...

Binary Search

Find 15



Binary Search Implementation

```
public boolean binarySearch(int key)
{
    int start = 0;
    int end = count-1;
    while(start <= end)
    {
        int middle = (start + end + 1) / 2;
        if(key < data[middle]) end = middle -1;
        else if(key > data[middle]) start = middle + 1;
        else return true;
    }
    return false;
}
```

Some Remarks

1. Simple, yet useful. Vector will be the base for building more complex structures.
2. Implementation is limited to a vector containing integers.
3. We need generic data structures in order to build a reusable tool set.
4. using Object, Comparable, Generics, ...

Mathematical Induction, recursion and time complexities

What is mathematical induction?

A technique to prove properties of natural numbers $P(n)$ with $n \in \mathbb{N}$.

1. Proof for $P(0)$ (base case)
2. Proof $\forall n \in \mathbb{N} : P(n) \Rightarrow P(n+1)$ (induction step)

Mathematical Induction

Theorem 2

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Proof.

1. For $n = 1$. Trivial
2. (induction step)
   $\sum_{i=1}^{n+1} i = n + 1 + \frac{n(n+1)}{2} = \frac{2n+2+n^2+n}{2} = \frac{(n+1)(n+2)}{2}$

Mathematical Induction

Theorem 3

Any positive integer greater than one can be written as a product of primes.
OR more formally
For any integer $n \geq 1$ *there exists primes* $p_1, ..., p_m$ *such that* $n = p_1 ... p_m$

Mathematical Induction

Proof.

1. n is prime. A trivial basecase
2. n is not prime. Hence, it is a product of two numbers, say n_1 and n_2, and nor n_1 nor n_2 are 1.
   n_1 and n_2 are smaller than n, hence the induction hypothesis says that n_1 and n_2 can be written as a product of primes.
   Hence, $n_1 n_2$ is also a product of primes.

Remark

Obviously, the base case does not have to be $P(0)$ ...

More formally

Definition 1 (Axiom of induction)

$$\forall P[[P(0) \wedge \forall k \in \mathbb{N}(P(k) \Rightarrow P(k+1))] \Rightarrow \forall n \in \mathbb{N}[P(n)]]$$

Why mathematical induction is mentioned in this course?

- It is the foundation for recursion: from mathematical induction to structural induction.
- It is a good technique to prove properties of algorithms.

Towards recursion

Definition 2 (Factorial)

$n! = \prod_{i=0}^{n-1} n - i$

Example: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

An iterative version of factorial

```
int factorial(int n)
{
    double result = 1;
    for(int i=0;i<n;i++)
    {
        result *= (n - i);
    }
    return result;
}
```

Another definition of factorial

Definition 3 (Factorial)

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

Factorial example

$$
\begin{aligned}
5! &= 5 \times 4! \\
&= 5 \times 4 \times 3! \\
&= 5 \times 4 \times 3 \times 2! \\
&= 5 \times 4 \times 3 \times 2 \times 1! \\
&= 5 \times 4 \times 3 \times 2 \times 1 \times 0! \\
&= 5 \times 4 \times 3 \times 2 \times 1 \times 1 \\
&= 120
\end{aligned}
$$

Recursion

Definition 4

A function is recursive if it is defined in terms of itself.

A recursive version of factorial

```
int factorial(int n)
{
    if(n == 0) return 1;
    else return n * factorial(n-1);
}
```

The greatest common divisor

Theorem 4 (gcd)

The greatest common divisor (gcd) of two positive integers is the largest integer that divides evenly into both of them. (divides evenly = the remainder of the division is zero)

For example, the greatest common divisor of 102 and 68 is 34 since both 102 and 68 are multiples of 34, but no integer larger than 34 divides evenly into 102 and 68.

The greatest common divisor

Theorem 5 (Euclid's algorithm)

If $p > q$, the gcd of p and q is the same as the gcd of q and $p \bmod q$.

The greatest common divisor

```
public int gcd(int p, int q)
{
    while (q != 0)
    {
        int temp = q;
        q = p % q;
        p = temp;
     }
     return p;
}
```

The greatest common divisor

```
public int gcd(int p, int q)
{
   if (q == 0) return p;
   else return gcd(q, p % q);
}
```

What does this function do?

```
public int mcCarthy(int n)
{
    if (n > 100)
        return n - 10;
    return mcCarthy(mcCarthy(n+11));
}
```

Towards time complexities

An introductory example

Suppose we have an array of numbers and we are checking whether the array contains a specific element ...

```
public boolean contains(int obj)
{
    for(int i=0;i<count;i++)
    {
        if(data[i] == obj) return true;
    }
    return false;
}
```

How many comparisons do we need to do before we can find the element?

Time Complexity of the contains method

The number of comparisons

- If the array has N elements, T(N) is the number of required comparisons.
- The best case: $T(N) = 1$
- The worst case: $T(N) = N$
- The average case: $T(N) = (1 + 2 + \ldots N) / N = (N+1) / 2$
- On average, searching in an array that is twice as large, will take twice as much time.
- We say the find method is *linear* in time.

Time Complexities (CL 3.1: Asymptotic Notation)

Definitions

Definition 5 (big-Oh)

$T(n)$ is $\mathcal{O}(F(n))$ if there are positive constants c and n_0 such that $T(n) \leq cF(n)$ when $n \geq n_o$.

Definition 6 (big-Omega)

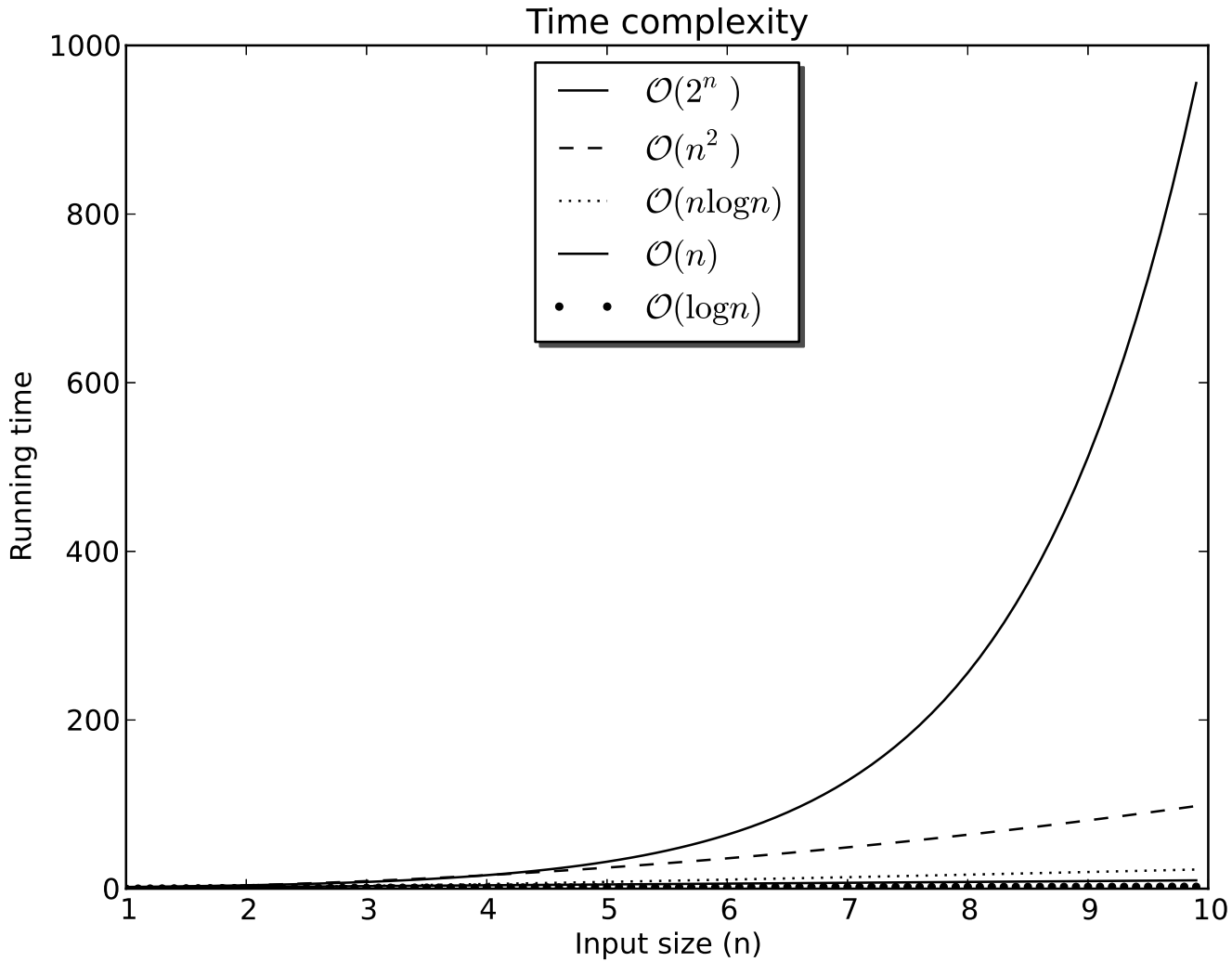
$T(n)$ is $\Omega(F(n))$ if there are positive constants c and n_0 such that $T(n) \geq cF(n)$ when $n \geq n_o$.

Definition 7 (big-Theta)

$T(n)$ is $\Theta(F(n))$ if and only if $T(n)$ is $\mathcal{O}(F(n))$ and $T(n)$ is $\Omega(F(n))$.

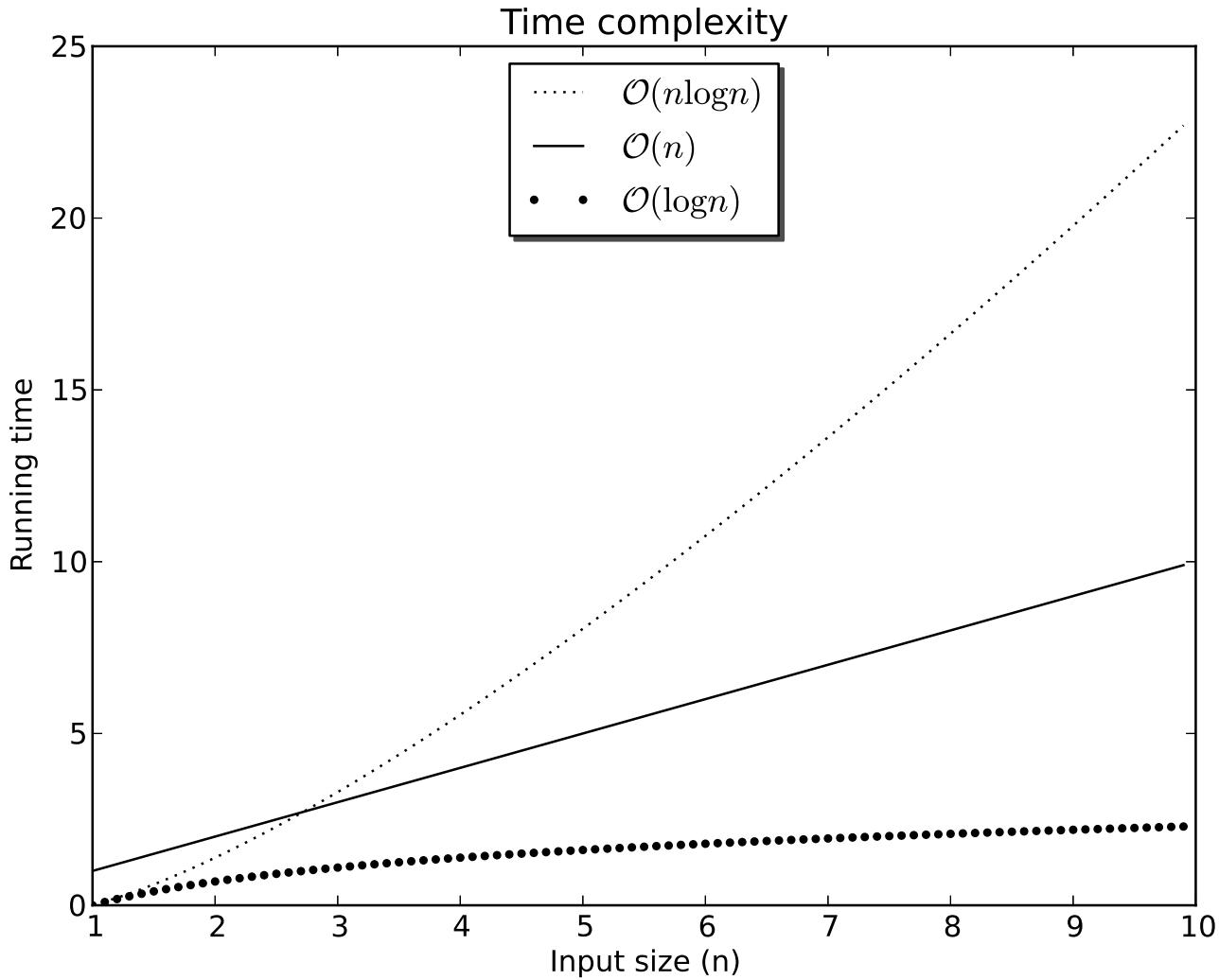
Time Complexities

Examples



Time Complexities

Examples



Time Complexities

Examples

- $T(n) = 3n^2 + 2n + 7 = \mathcal{O}(n^2)$ (quadratic).
- $T(n) = n + n + n = \mathcal{O}(n)$ (linear).
- $T(n) = 7 = \mathcal{O}(1)$ (constant).
- $T(n) = 2^n + 7n^2 + 11 = \mathcal{O}(2^n)$ (exponential).
- $T(n) = 2\log(n) + 1 = \mathcal{O}(\log(n))$ (logarithmic).

Time Complexities

Logarithmic bases

Theorem 6 (The base doesn't matter)

For any constant $B \geq 1$, $log_B(n) = \mathcal{O}(\log(n))$.

Proof.

$$\begin{aligned} T(n) &= \log_B(n) \\ &= \frac{log_2(n)}{log_2(B)} \\ &\leq cF(n) \text{ for } c = 1/log_2(B) \text{ and } F(n) = log_2(n) \\ &= \mathcal{O}(\log_2(n)) \end{aligned}$$

Time Complexities for the Vector class

- The time complexity of binary search is intuitively clear: $\mathcal{O}(\log(n))$.
- If at each step the length of the vector is divided by two, then the length of vector becomes 1 in $\log(n)$ steps.
- We should be more explicit and formal.

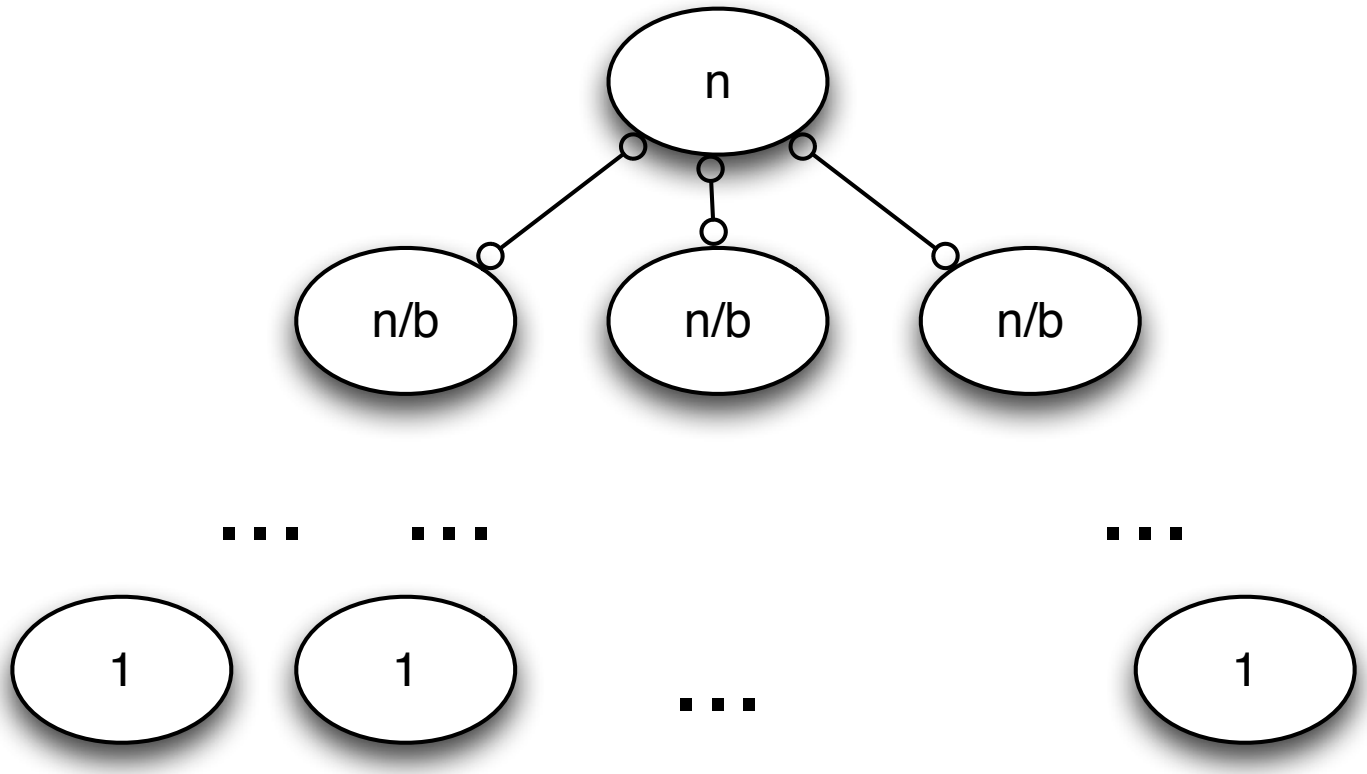
A generic approach for Divide And Conquer Algorithms

CL 4: Divide and Conquer

Assume the total size of the problem is n.

1. The algorithm splits the problem into a pieces, each of size n/b. (Please note that a is not per se equal to b, as the subproblems could for instance overlap.)
2. The algorithm solves the problem on each of these a pieces. (this is the recursion step)
3. The algorithm aggregates the solutions to the a smaller pieces into a single solution.

Divide And Conquer Algorithms



Time Complexity of Divide And Conquer Algorithms

Theorem 7

The time complexity of divide and conquer algorithms is defined by the following recurrence relation: $T(n) = a\ T(n/b) + f(n)$

where $f(n)$ is the cost of aggregating the a different simpler problems.

Time Complexity of Divide And Conquer Algorithms

- We assume that we stop dividing the problem into smaller problems when the problem reaches size 1.
- We assume that n is a power of b.

- The consequence of the second assumption is that the subproblems of size 1 are always obtained in an integer number of division steps (because the problem size is divided by a factor b in each step).
- Hence, both assumptions together result in the fact that the algorithm always stops the recursion in an integer number of steps.

A theorem for the Time Complexity of Divide And Conquer Algorithms

CL 4.5: The master theorem for solving recurrences.

Theorem 8

Under the above stated conditions, the solution of the recurrence relation

$$T(n) = aT(n/b) + f(n)$$

is given by

$$T(n) = \sum_{i=0}^{\log n/\log b} a^i f(n/b^i)$$

CL 4.6: Proof of the master theorem for solving recurrences - more formal.

Proof.

- At the start of the algorithm (*level 0*), there is a single problem of size n (or put differently a^0 pieces of size n/b^0 each).
- At level 1, the problem is divided into a pieces of size n/b each (or put differently a^1 pieces of size n/b^1 each).
- Each of these a smaller pieces will be broken down into smaller pieces.
- Consequently, at level i, there will be a^i pieces, each of size n/b^i.
- This process of dividing the problem into smaller problems continues until the problems reach size 1. But after how many steps is this exactly?
- At the lowest level i, we have that $n/b^i = 1$, so $n = b^i$, so $i = \log_b n = \log_2 n / \log_2 b$.

[Proof Continued]

- Once the problem is broken into a series of problems of size 1, these smaller problems become trivial to solve (e.g. sorting a list of a single element or finding an element in a list of a single element).
- Now as specified by step 3 of the divide and conquer paradigm, the solutions to the simpler problems need to be aggregated into a solution of the actual problem.
- At level i, the size of each of the pieces is n/b^i and so the cost of aggregating these pieces can be generally referred to as $f(n/b^i)$ for level i.
- Putting this all together, the total cost of solving a problem according to the divide and conquer paradigm, is the sum of the costs at each of the i levels.
- At level i, the cost is the product of the aggregation cost on the pieces $f(n/b^i)$ with the number of pieces at the i-th level a^i.
- So, in total we obtain: $T(n) = \sum_{i=0}^{\log n/\log b} a^i f(n/b^i)$.

Back to Binary Search

Theorem 9

The time complexity of the Binary Search algorithm on a sorted vector is $\mathcal{O}(\log n)$.

Proof.

At each level, there is one comparison to be made for deciding whether to go to the left or to the right and in each step the size of the subproblems is reduced by a factor of 2.

$$T(n) = T(n/2) + 1 \tag{1}$$

So, this is the recurrence relation from theorem 7, with $a = 1$, $b = 2$ and $f(n) = 1$.

According to theorem 8, the solution to this recurrence relation is given by

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log n} 1 \\ &= \log n + 1 \\ &= \mathcal{O}(\log n) \end{aligned}$$

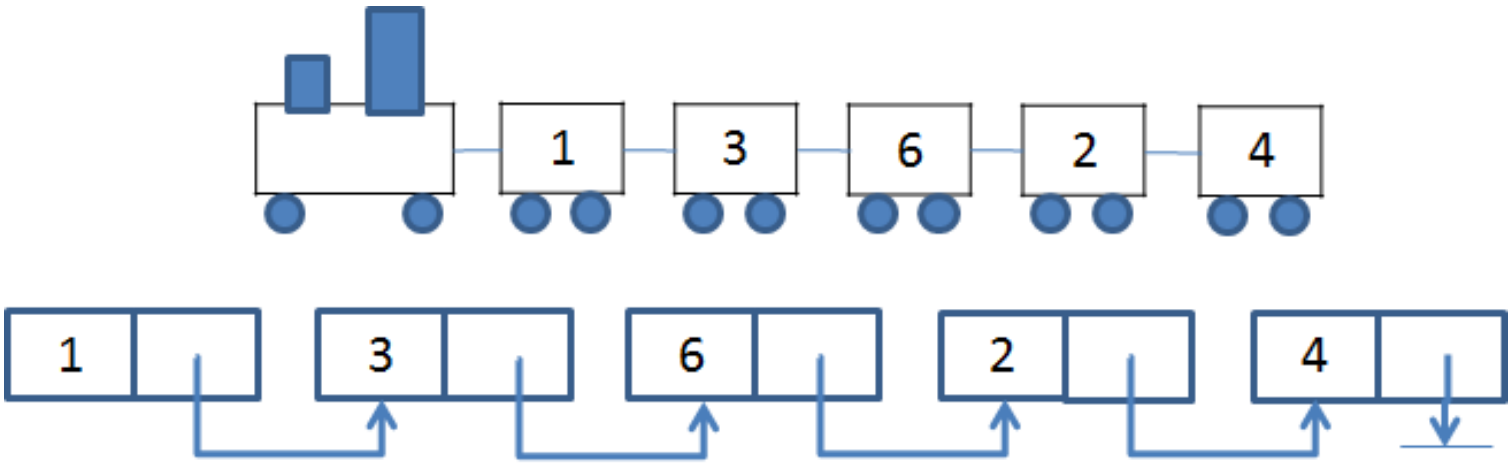
Time Complexities of the other operations

- getFirst: $\mathcal{O}(1)$
- getLast: $\mathcal{O}(1)$
- addFirst: $\mathcal{O}(n)$
- addLast: $\mathcal{O}(1)$
- get: $\mathcal{O}(1)$
- size: $\mathcal{O}(1)$
- contains: $\mathcal{O}(n)$
- removeFirst: $\mathcal{O}(n)$
- removeLast: $\mathcal{O}(1)$
- ...

Linked Lists

Linked Lists

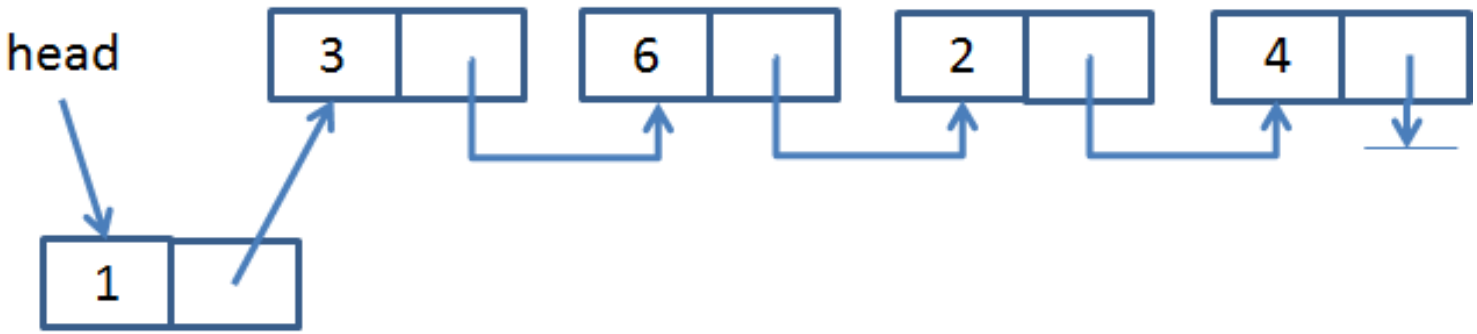
CL 10.2: Linked Lists



Linked List

```
public class LinkedList
{
    private class ListElement
     {
          private Object el1;
          private ListElement el2;
          public ListElement(Object el, ListElement nextElement)
          {
               el1 = el;
               el2 = nextElement;
          }
          public ListElement(Object el)
          {
               this(el,null);
          }
          public Object first()
          {
               return el1;
          }
          public ListElement rest()
          {
               return el2;
          }
          public void setFirst(Object value)
          {
               el1 = value;
          }
          public void setRest(ListElement value)
          {
               el2 = value;
          }
     }
```

AddFirst: Adding a new element at the beginning of the list



AddFirst: 3 crucial steps

```
public void addFirst(Object o)
{
    ListElement newElement = new ListElement(o); // step 1
    newElement.setRest(head); // step 2
    head = newElement; // step 3
}
```

AddFirst: The same 3 steps done at once

```
public void addFirst(Object o)
{
    head = new ListElement(o,head);
}
```

Linked List: retrieving the n-th element

```
public Object get(int n)
{
    ListElement d = head;
    while(n>0)
    {
        d = d.rest();
        n--;
    }
    return d.el1;
}
```

Time Complexities

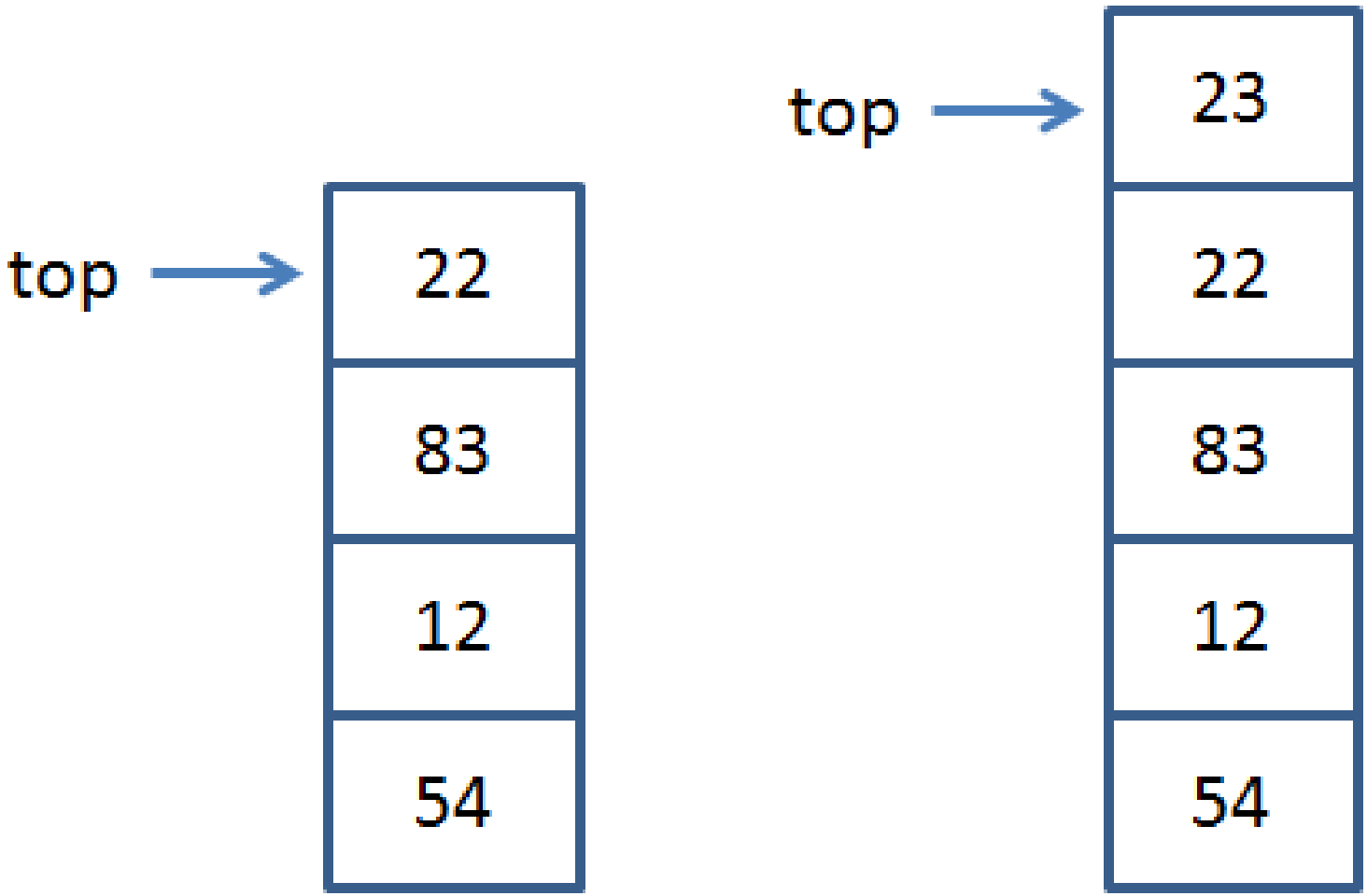
At least in our most naive version

- getFirst: $\mathcal{O}(1)$
- getLast: $\mathcal{O}(n)$
- addFirst: $\mathcal{O}(1)$
- addLast: $\mathcal{O}(n)$
- get: $\mathcal{O}(n)$
- size: $\mathcal{O}(1)$
- contains: $\mathcal{O}(n)$
- removeFirst: $\mathcal{O}(1)$
- removeLast: $\mathcal{O}(n)$
- ...

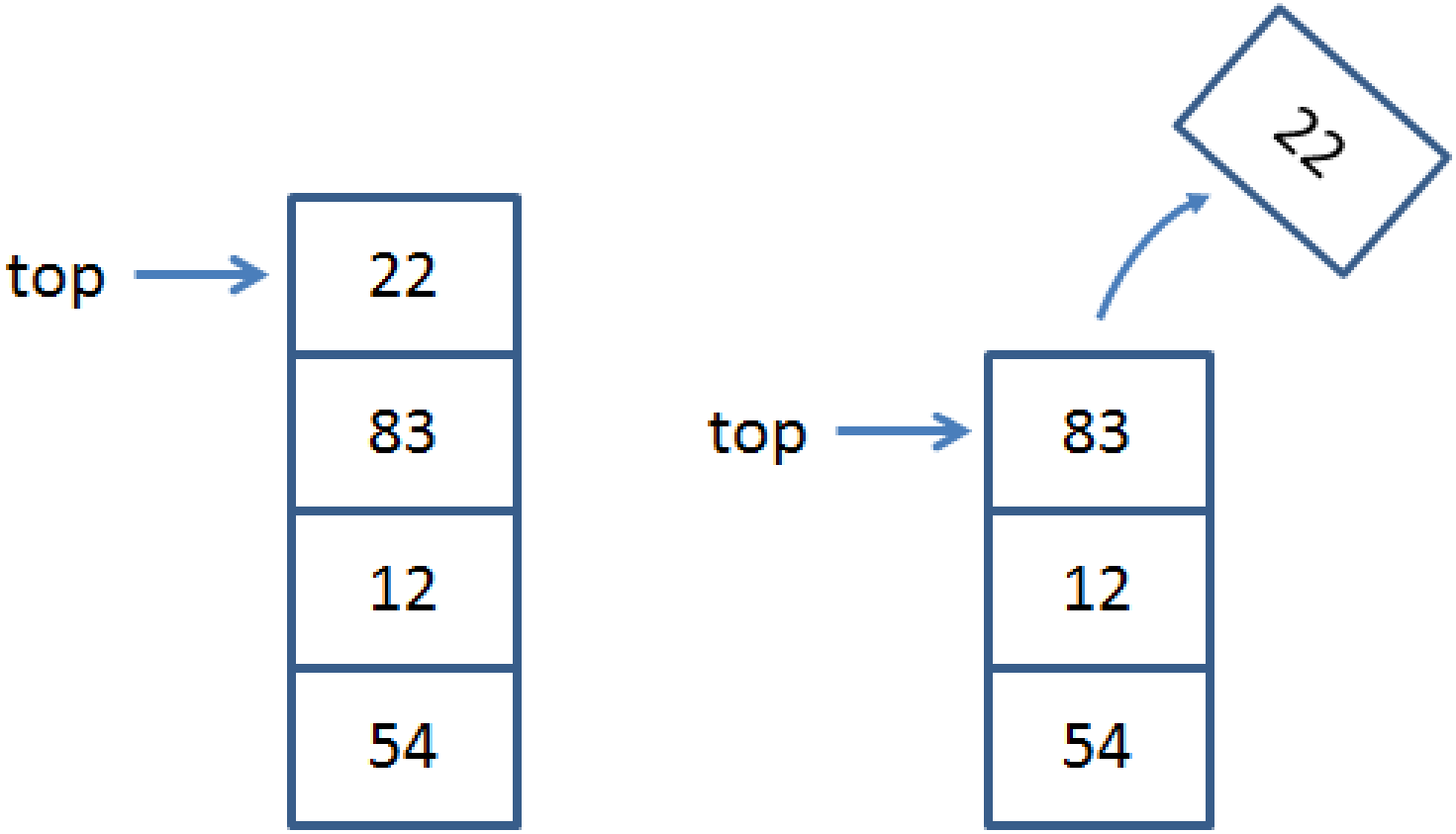
Stacks and Queues

Stacks: push

CL 10.1: Stacks and Queues



Stacks: pop



Stacks

- Push: New items are stored at the end
- Pop: new items are removed at the end

A Stack based on a Vector

```
public class Stack
{
        private Vector data;

        public Stack()
        {
                data = new Vector();
        }

        public void push(Object o){}
        public Object pop(){}
        public Object top(){}
        public int size(){}
        public boolean empty(){}
 }
```

Queue

- Push: New elements are added at the back.
- Pop: Elements are removed from the front of the queue

(Or the other way around)

A Queue based on a vector

```
public class Queue
{
        private Vector data;

        public Queue()
        {
                data = new Vector();
        }

        public void push(Object o){}
        public Object pop(){}
        public Object top(){}
        public int size(){}
        public boolean empty(){}
 }
```

Time Complexities

At least in our most naive version

- Stack (both vector and linked list based)
  - Push: $\mathcal{O}(1)$
  - Pop: $\mathcal{O}(1)$
- Queue (both vector and linked list based)
  - Push: $\mathcal{O}(1)$
  - Pop: $\mathcal{O}(n)$
- Queue (alternative)
  - Push: $\mathcal{O}(n)$
  - Pop: $\mathcal{O}(1)$

Remark

- The queue only implements two basic methods (push and pop), but their time complexities are very different.
- This is the case for both the vector and the linked list based implementation.
- See later for $\mathcal{O}(1)$ push and pop methods.

Priority Queue

Priority Queue

- Similar to the Stack and Queue data structure, the priority queue is a simple structure supporting the push and pop methods.
- The pop method does not return the topmost element in the structure, but the element with the highest priority.
- As a consequence, the push method does not only store the data, but also the priority.

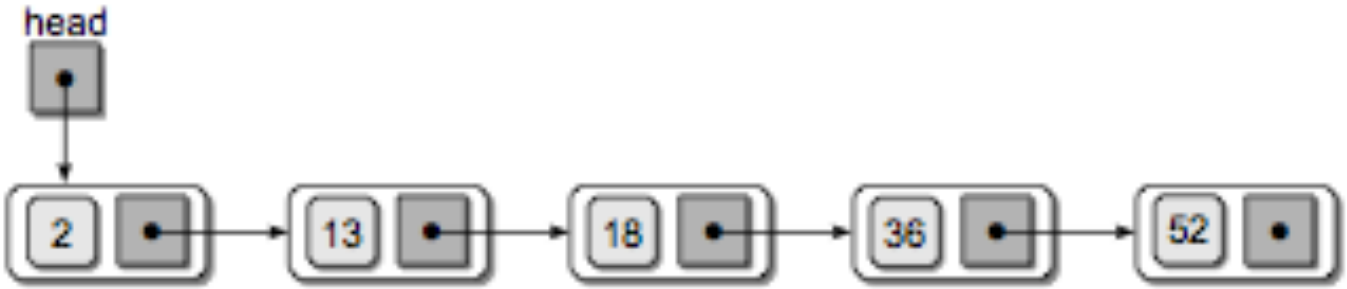
Priority Queues: Two different approaches

- Maintaining the data structure sorted on the priority.The Push method needs to insert the element at the right position in an already sorted data structure, to ensure that the structure remains sorted ($\mathcal{O}(n)$). As the element with the highest priority always is at the first / last position, the pop remains very simple ($\mathcal{O}(1)$).
- The second approach implements the opposite logic: the push operation is very simple (as in the Stack and Queue implementation it just stores the new data at the end or at the beginning), at the price of a more complex pop operation. As the data is now stored unsorted, the pop operation must check the entire data structure to identify the element with the highest priority. In this approach, the push operation has a time complexity of $\mathcal{O}(1)$, while the pop operation has a time complexity of $\mathcal{O}(n)$.

Remark

- This datastructure only implements two basic methods (push and pop), but their time complexities are very different.
- This is the case for both described approaches.
- This is the case for both the vector and the linked list based implementation.
- See later for $\mathcal{O}(\log n)$ push and pop methods.

A sorted Linked List



Adding elements in a sorted linked list

Basic Idea

```
// THIS CODE IS NOT CORRECT JAVA CODE
// BUT IT ILLUSTRATES THE LOGIC OF THE ALGORITHM

public void addSorted(Object o)
{
    // an empty list, add element in front
    if(head == null) head = new ListElement(o,null);
    else if(o < head.first())
    {
        // we have to add the element in front
        head = new ListElement(o,head);
    }
    else
    {
        // we have to find the first element which is bigger
        ListElement d = head;
        while((d.rest() != null)&&(o > d.rest().first()))
        {
            d = d.rest();
        }
        ListElement next = d.rest();
        d.setRest(new ListElement(o,next));
    }
    count++;
}
```

Adding elements in a sorted linked list

More correct

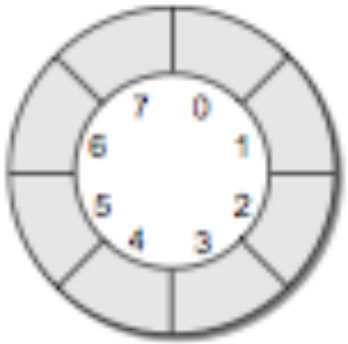
```
public void addSorted(Comparable o)
{
    // an empty list, add element in front
    if(head == null) head = new ListElement(o,null);
    else if(head.first().compareTo(o) > 0)
    {
        // we have to add the element in front
        head = new ListElement(o,head);
    }
    else
    {
        // we have to find the first element which is bigger
        ListElement d = head;
        while((d.rest() != null)&&
              (d.rest().first().compareTo(o) < 0))
        {
            d = d.rest();
        }
        ListElement next = d.rest();
        d.setRest(new ListElement(o,next));
    }
    count++;
}
```

```
import java.util.Comparator;

public class PriorityQueue
{
    private class PriorityPair implements Comparable
    {
        public Object element;
        public Object priority;

        public PriorityPair(Object element, Object priority)
        {
            this.element = element;
            this.priority = priority;
        }

        public int compareTo(Object o)
        {
            PriorityPair p2 = (PriorityPair)o;
            return ((Comparable)priority).compareTo(p2.priority);
        }
    }

    private LinkedList data;

    public PriorityQueue()
    {
        data = new LinkedList();
    }

    public void push(Object o, int priority)
    {
        // make a pair of o and priority
        // add this pair to the sorted linked list.
    }

    public Object pop()
    {
```

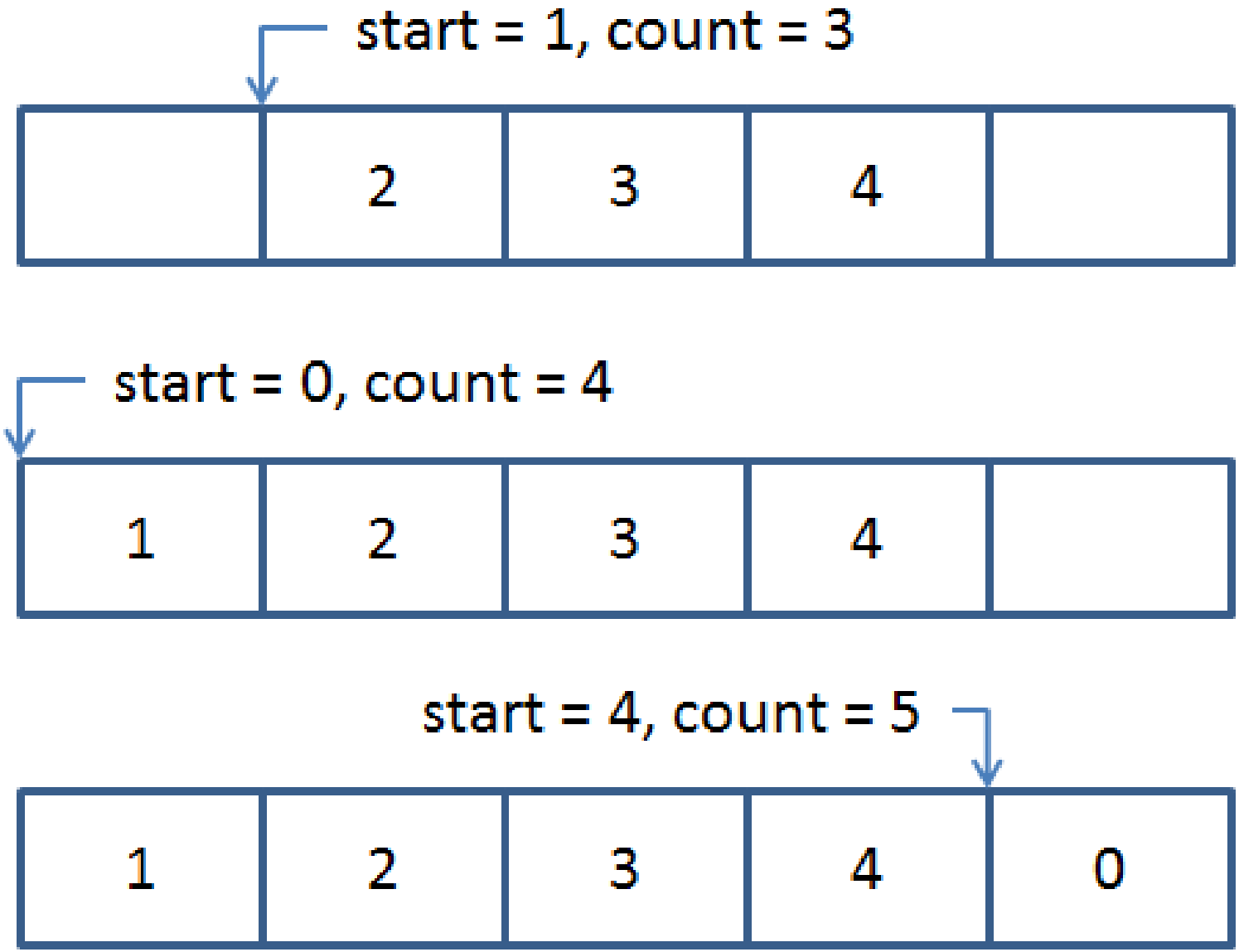
Circular Vectors

Circular Vector

Logical versus physical layout



AddFirst of 1 and 0



Operations

- addFirst
  - Add the new element at position "start - 1"
  - Unless start is 1, then we need to add at the end of the vector, i.e. at "capacity - 1".
  - start = (start + capacity - 1) % capacity
- addLast
  - Add the new element at position "start + count"
  - When start + count $\geq$ capacity, then we need to add at the beginning of the vector.
  - Store the element at (start + count) % capacity

Implementation

```
public class CircularVector
{
    private Object data[];
    private int first;
    private int count;
    private int capacity;

    public CircularVector(int capacity)
    {
        count = 0;
        first = 0;
        data = new Object[capacity];
        this.capacity = capacity;
    }

    public int size()
    {
        return count;
    }

    public void addFirst(Object element){}
    public void addLast(Object element){}
    public Object getFirst(){}
    public Object getLast(){}
    public void removeFirst(){}
    public void removeLast(){}
    public void print(){}
}
```

Implementation

```
public void removeFirst()
{
    if (count > 0)
     {
       first = (first + 1) % capacity;
         count--;
     }
}

public void print()
{
     System.out.print("[");
     for (int i=0; i<count; i++)
     {
         int index = (first + i) % capacity();
         System.out.print(data[index] + " ");
     }
     System.out.println("]");
}
```

Time Complexities - Circular Vectors

- addFirst: $\mathcal{O}(1)$
- addLast: $\mathcal{O}(1)$
- removeFirst: $\mathcal{O}(1)$
- removeLast: $\mathcal{O}(1)$

Remember ...

```
public class Queue
{
        private Vector data;

        public Queue()
        {
                data = new Vector();
        }

        public void push(Object o){}
        public Object pop(){}
        public Object top(){}
        public int size(){}
        public boolean empty(){}
 }
```

... and compare with ...

```
public class Queue
{
        private CircularVector data;

        public Queue()
        {
                data = new CircularVector();
        }

        public void push(Object o){}
        public Object pop(){}
        public Object top(){}
        public int size(){}
        public boolean empty(){}
 }
```

Double Linked Lists

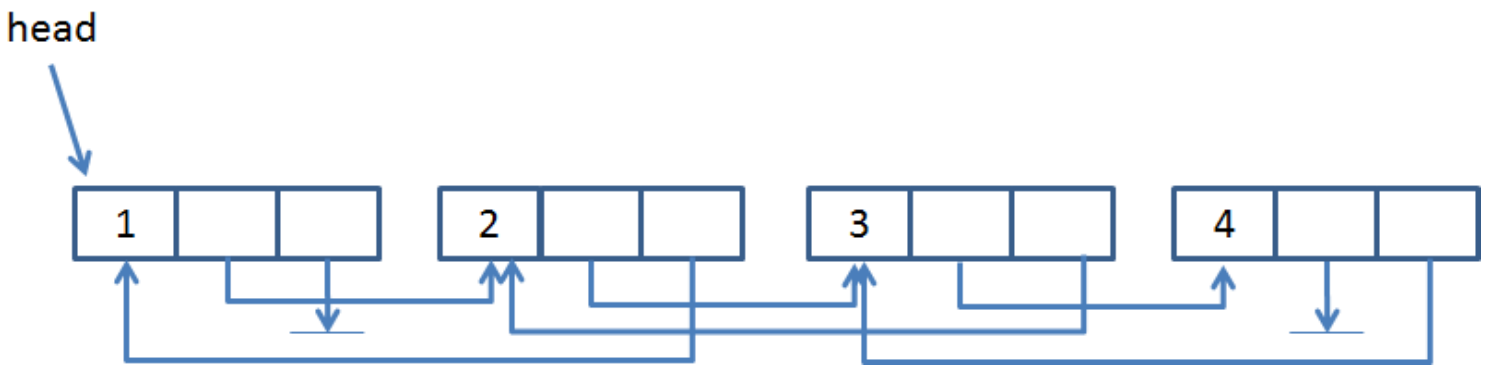
Double Linked List

- Adding and removing elements at the end of a linked list is not efficient.
- By keeping a pointer to the last element, adding an element can be done in constant time.
- For making the remove more efficient, we need a pointer to the second last element
- AND we need to be able to update it to the previous element

Double Linked List

- We have a linked list which we traverse from left to right.
- Simultaneously, we store the same data in a linked list we traverse from right to left.
- Code should make sure both lists are kept up-to-data in an efficient manner.

Double Linked List



Double Linked List

```
public class DoubleLinkedList
{
    private class DoubleLinkedListElement {...}

    private int count;
    private DoubleLinkedListElement head;
    private DoubleLinkedListElement tail;

    public DoubleLinkedList()
    {
        head = null;
        tail = null;
        count = 0;
    }

    public Object getFirst(){...}
    public Object getLast(){...}
    public int size(){...}
    public void addFirst(Object value){...}
    public void addLast(Object value){...}
}
```

Double Linked List

```
private class DoubleLinkedListElement
{
    private Object data;
     private DoubleLinkedListElement nextElement;
     private DoubleLinkedListElement previousElement;

     public DoubleLinkedListElement(Object v, DoubleLinkedListElement next,
                                    DoubleLinkedListElement previous)
     {
         data = v;
         nextElement = next;
         previousElement = previous;
         if (nextElement != null) nextElement.previousElement = this;
         if (previousElement != null) previousElement.nextElement = this;
     }
     public DoubleLinkedListElement(Object v)
     {
         this(v,null,null);
     }
     public DoubleLinkedListElement previous()
     {
         return previousElement;
     }
     public Object value()
     {
         return data;
     }
     public DoubleLinkedListElement next()
     {
         return nextElement;
     }
     public void setNext(DoubleLinkedListElement value)
     {
```

Double Linked List

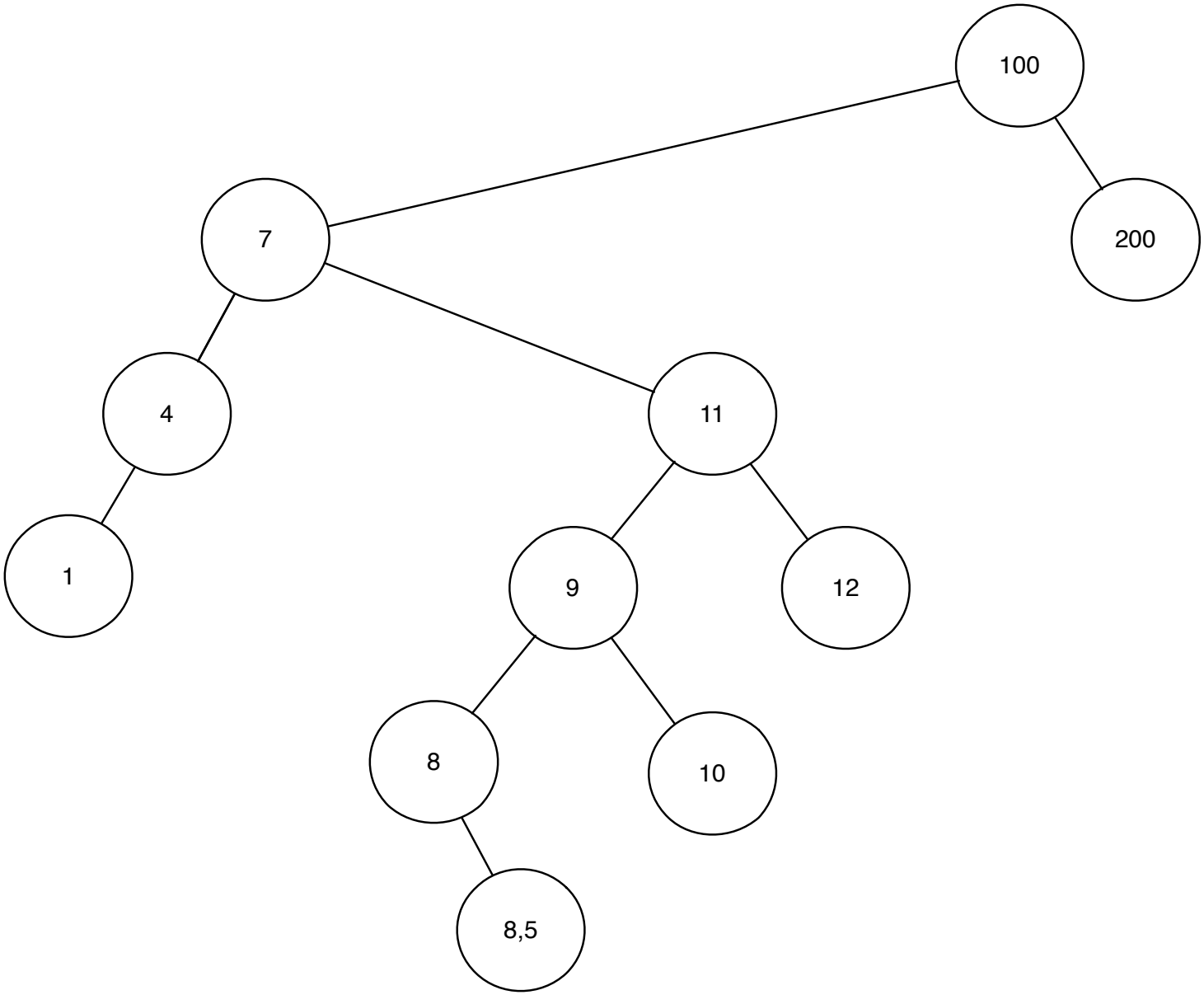
```
public Object getFirst()
{
    return head.value();
}
public Object getLast()
{
    return tail.value();
}
public int size()
{
    return count;
}
public void addFirst(Object value)
{
    head = new DoubleLinkedListElement(value, head, null);
    if (tail == null) tail = head;
    count++;
}
public void addLast(Object value)
{
    tail = new DoubleLinkedListElement(value, null, tail);
    if (head == null) head = tail;
    count++;
}
public void print()
{
    DoubleLinkedListElement d = head;
    System.out.print("(");
   while(d != null)
    {
        System.out.print(d.value() + " ");
        d = d.next();
    }
```

Binary Search Trees

Binary Search Tree

- A Binary Search Tree is a binary tree in which for each node K all the nodes in the left subtree of K are smaller than the node K and all the nodes in the right subtree of K are larger than K.

Binary Search Tree



Binary Search Tree

```
import java.util.Comparator;

public class Tree
{
   private class TreeNode implements Comparable
   {
        protected Comparable value;
        protected TreeNode leftNode;
        protected TreeNode rightNode;
        protected TreeNode parentNode;

        public TreeNode(Comparable v, TreeNode left, TreeNode right, TreeNode parent)
        {
            value = v;
            leftNode = left;
            rightNode = right;
            parentNode = parent;
        }
        public TreeNode(Comparable v)
        {
            this(v,null,null,null);
        }

      public TreeNode getLeftTree(){return leftNode;}
      public TreeNode getRightTree(){return rightNode;}
      public TreeNode getParent(){return parentNode;}
      public Comparable getValue(){return value;}

      public int compareTo(Object arg0)
      {
         TreeNode node2 = (TreeNode)arg0;
         return value.compareTo(node2.value);
      }
```

Binary Search Tree

```
public class TreeNode implements Comparable
{
    protected Comparable value;
    protected TreeNode leftNode;
    protected TreeNode rightNode;
    protected TreeNode parentNode;

    public TreeNode(Comparable v, TreeNode left, TreeNode right, TreeNode parent)
    {
     value = v;
     leftNode = left;
     rightNode = right;
     parentNode = parent;
    }
    public TreeNode(Comparable v)
    {
     this(v,null,null,null);
    }

   public TreeNode getLeftTree(){return leftNode;}
   public TreeNode getRightTree(){return rightNode;}
   public TreeNode getParent(){return parentNode;}
   public Comparable getValue(){return value;}

   public int compareTo(Object arg0)
   {
      TreeNode node2 = (TreeNode)arg0;
      return value.compareTo(node2.value);
   }
}
```

Searching for given element in the binary search tree

- The tree is empty, so the element is not present in the tree.
- The root of the tree equals to the element, so we have found the element in the tree.
- The root of the tree is bigger than the element. This means that if the element is part of the tree, it should be in the left subtree. So, we should continue the search in the left subtree.
- The root of the tree is smaller than the element.This means that if the element is part of the tree, it should be in the right subtree. So, we should continue the search in the right subtree.

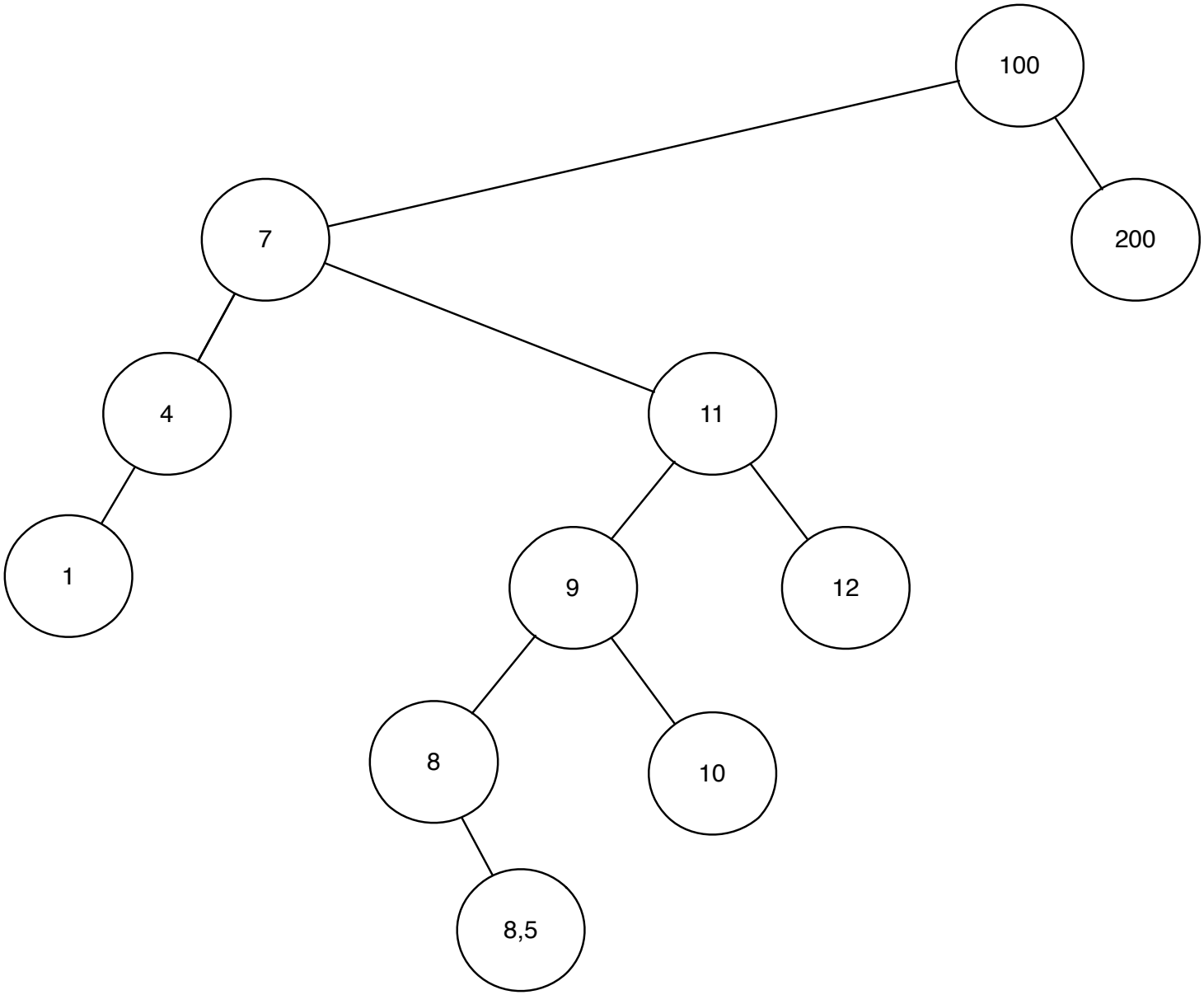
Binary Search Tree

```
private boolean findNode(Comparable element,
                         TreeNode current)
{
    if(current == null) return false;
    else if (element.compareTo(current.value) == 0)
             return true;
    else if (element.compareTo(current.value) < 0)
    {
        return findNode(element,current.getLeftTree());
    }
    else return findNode(element,current.getRightTree());
}

public boolean find(Comparable element)
{
    return findNode(element,root);
}
```

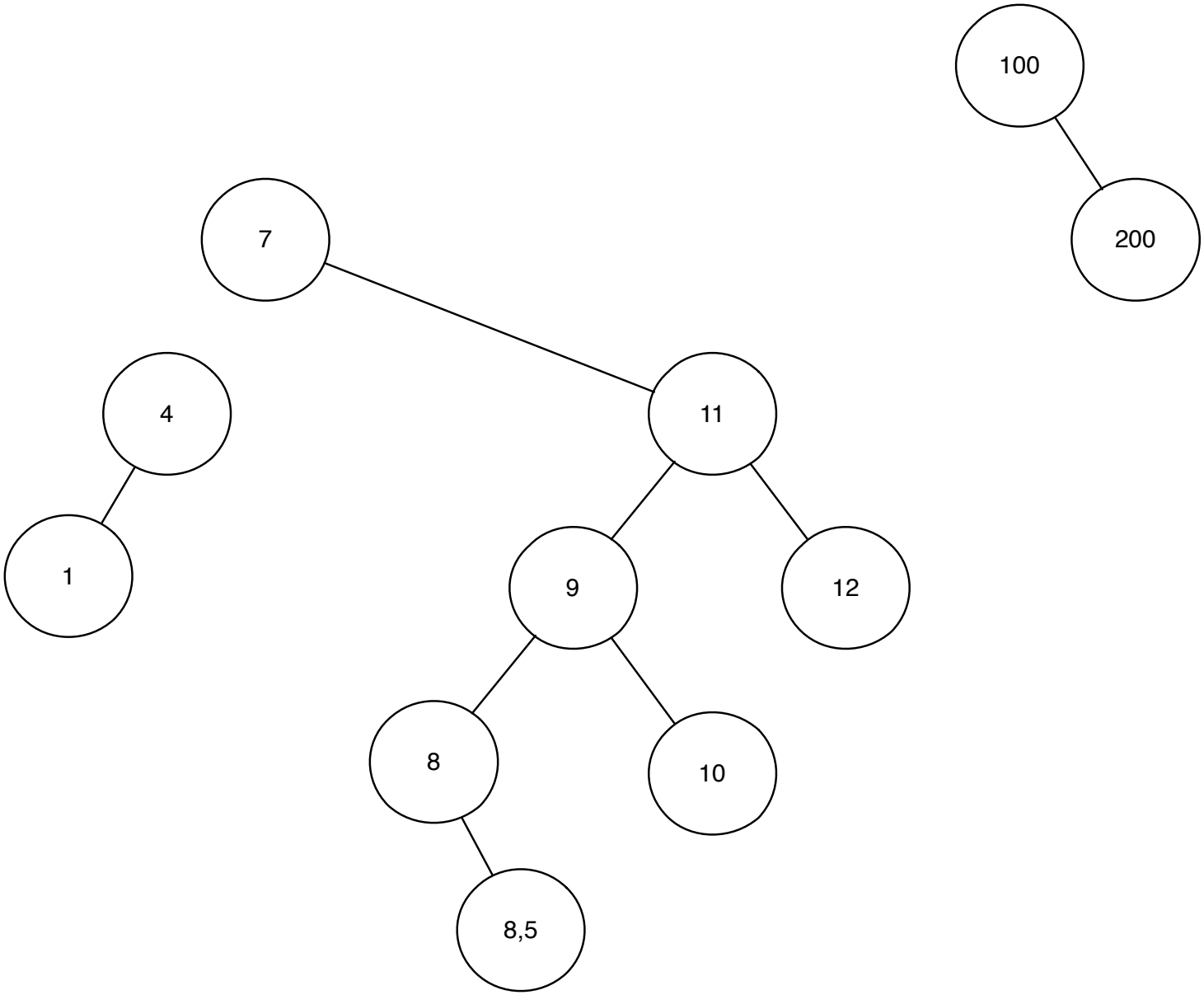
Binary Search Tree

```
public void insert(Comparable element)
{
   insertAtNode(element,root,null);
}

private void insertAtNode(Comparable element,
                          TreeNode current,
                          TreeNode parent)
{
    // if the node we check is empty
    if(current == null)
    {
       TreeNode newNode = new TreeNode(element);
       // the current node is empty, but we have a parent
       if(parent != null)
       {
           if(element.compareTo(parent.value) < 0) parent.leftNode = newNode;
           else parent.rightNode = newNode;
        }
        else root = newNode;
        newNode.parentNode = parent;
    }
    else if(element.compareTo(current.value) == 0)
    {
      // we do not care
    }
    // if the element is smaller than current, go left
    else if(element.compareTo(current.value) < 0)
    {
      insertAtNode(element,current.getLeftTree(),current);
    }
    // if the element is bigger than current, go right
    else insertAtNode(element,current.getRightTree(),
```

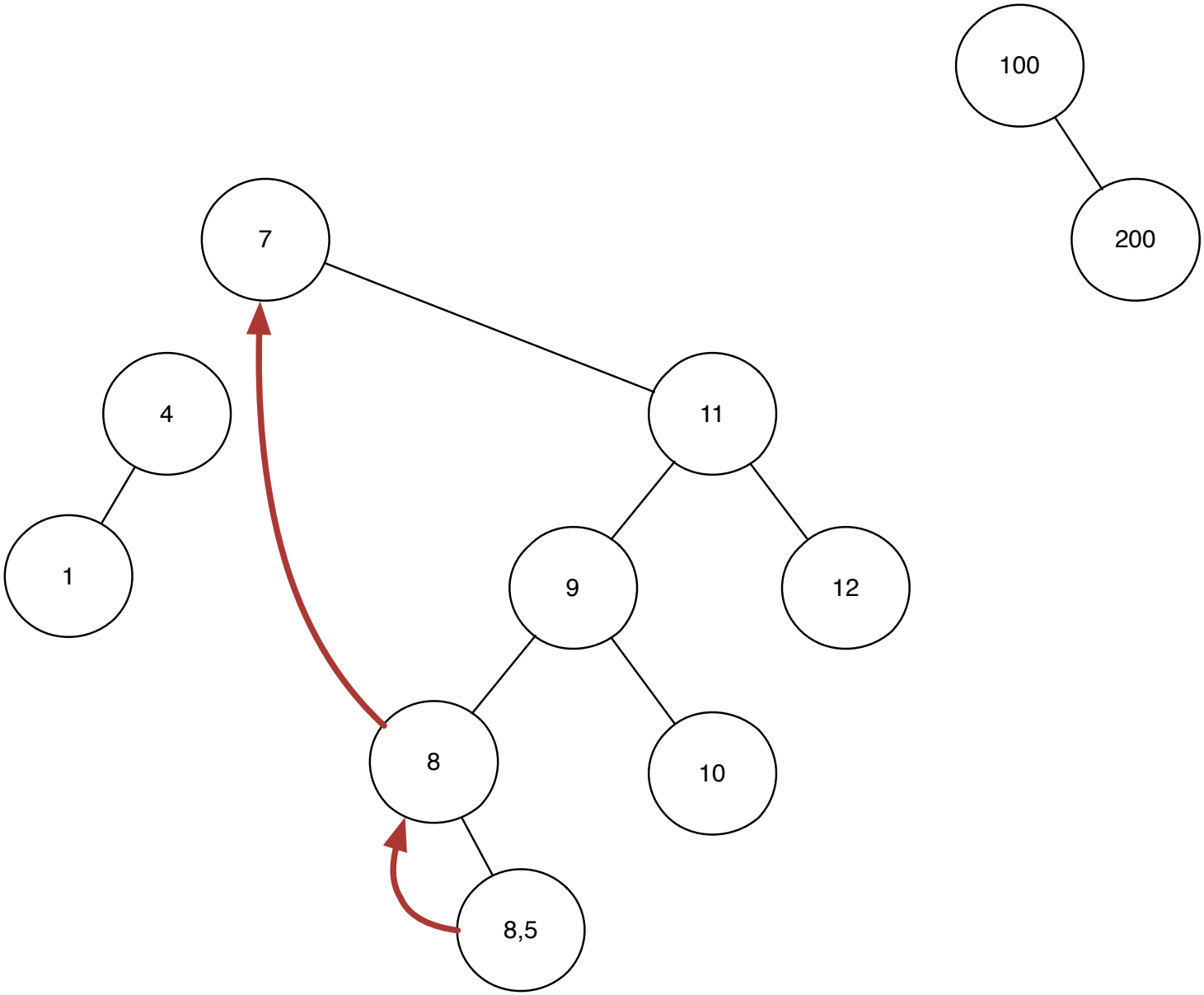
Binary Search Tree: Deleting 7



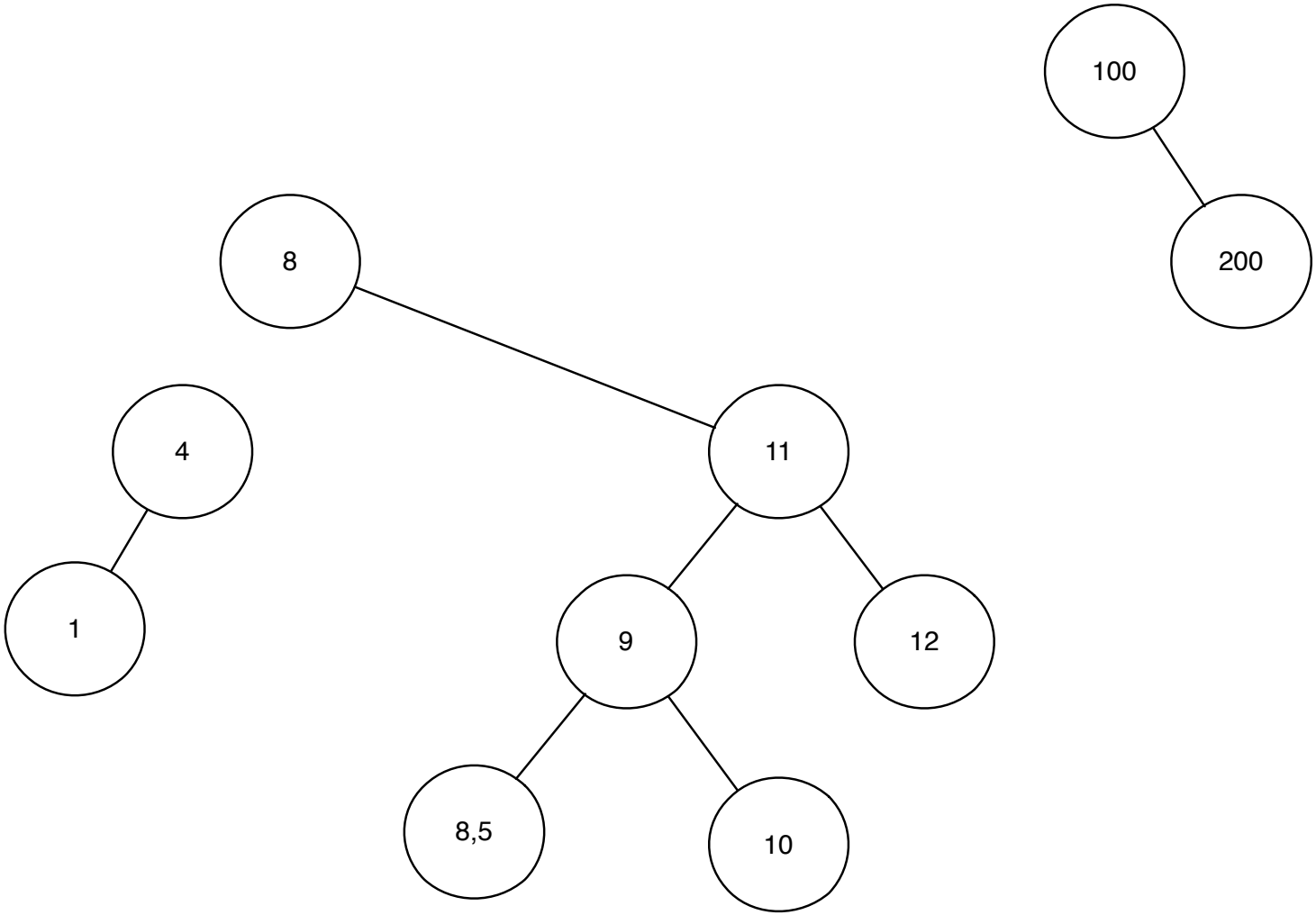
Binary Search Tree: Deleting 7



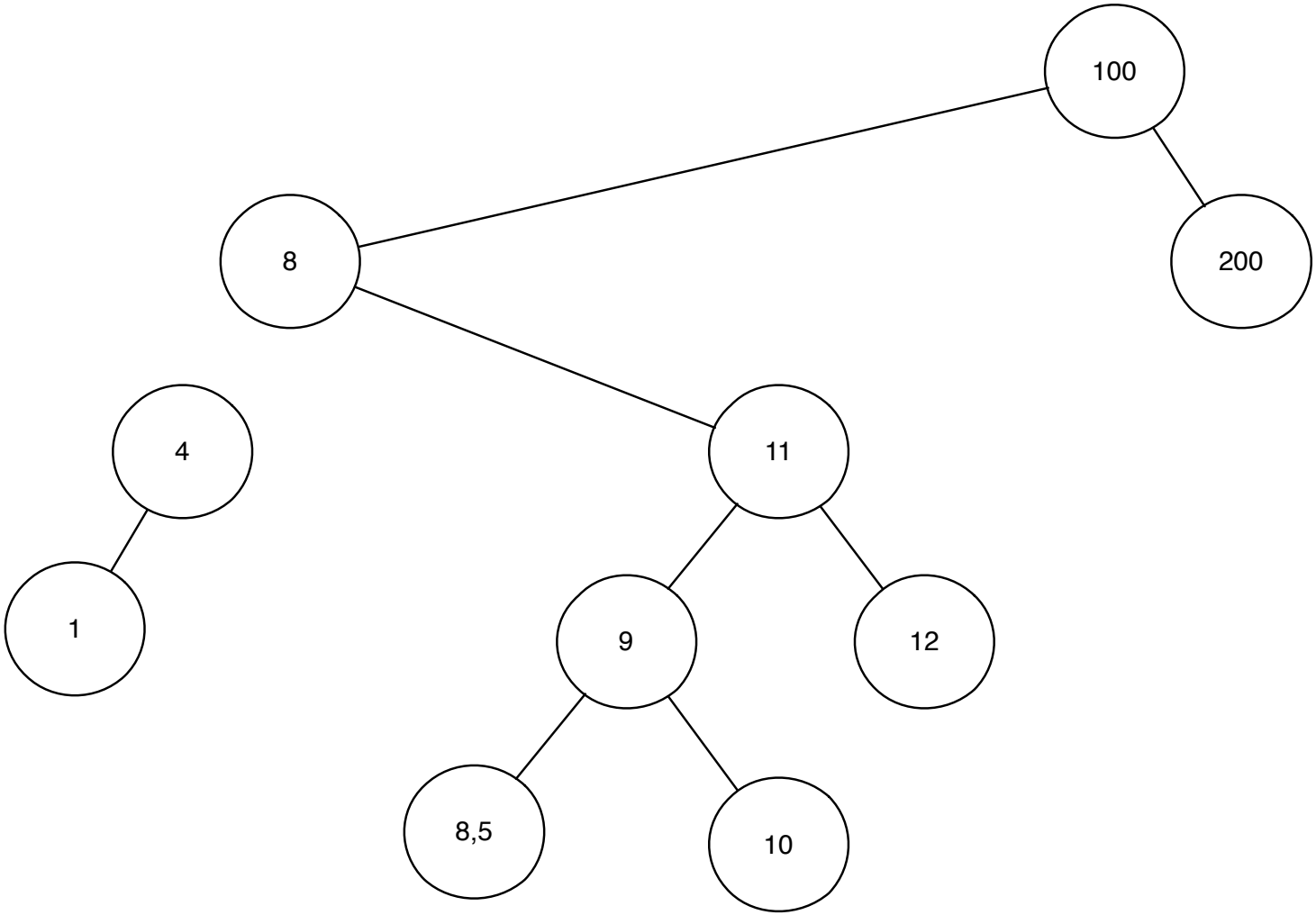
Binary Search Tree: Deleting 7



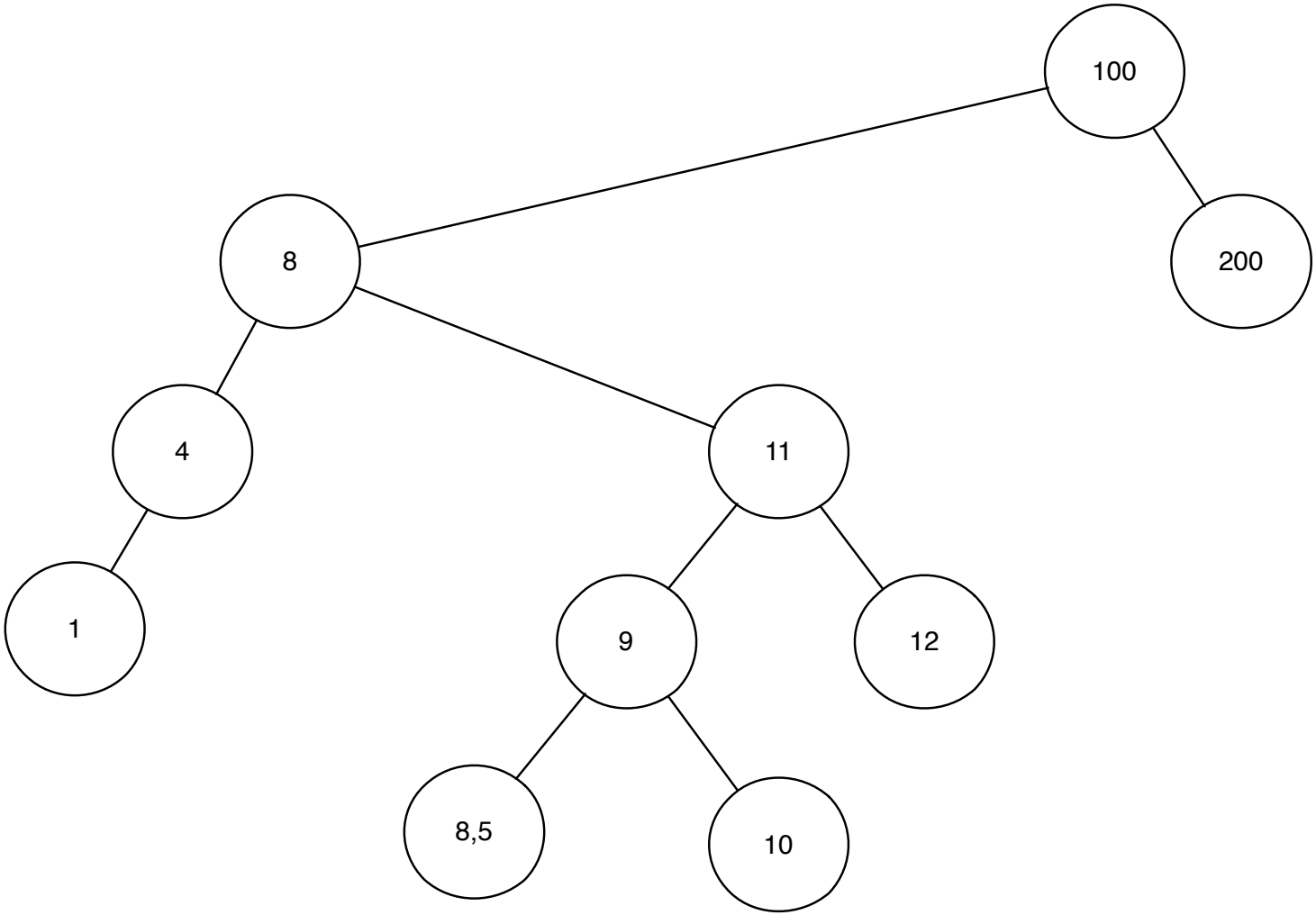
Binary Search Tree: Deleting 7



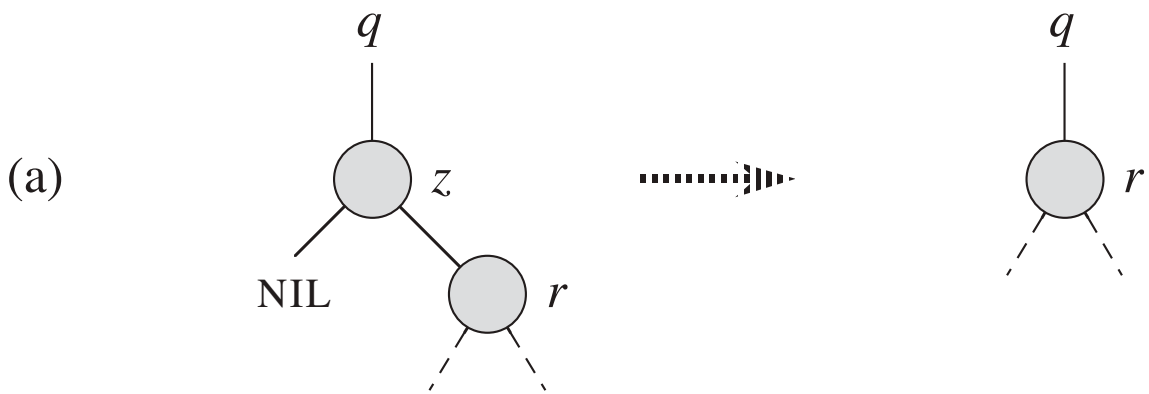
Binary Search Tree: Deleting 7



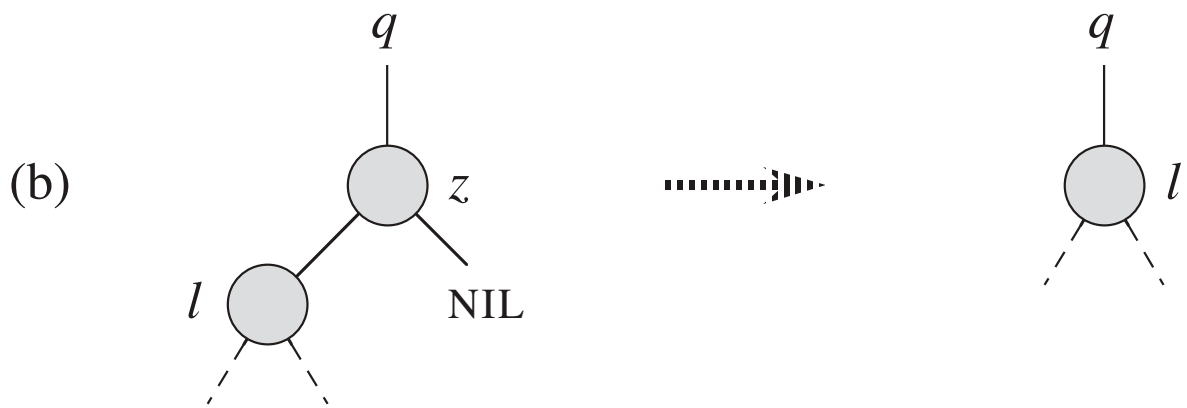
Binary Search Tree: Deleting 7



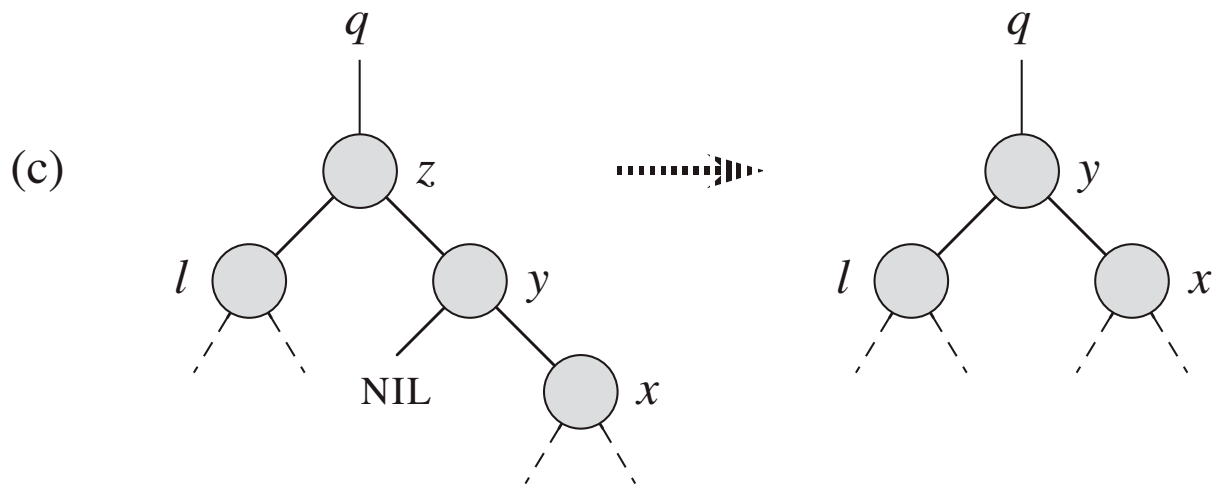
Binary Search Tree



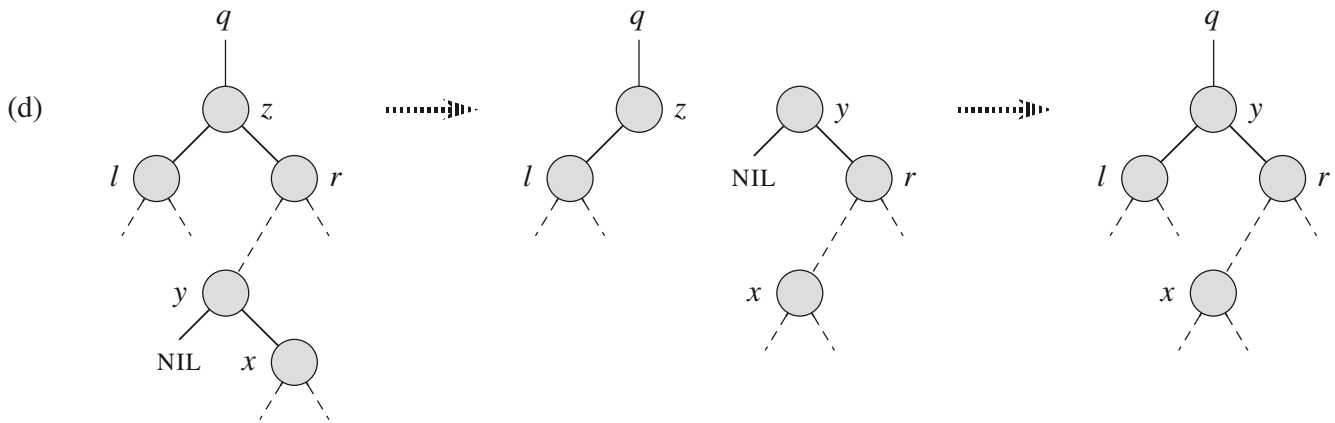
Binary Search Tree



Binary Search Tree



Binary Search Tree



FInding the smallest element in a subtree

```
public TreeNode minNode(TreeNode current)
{
    if(current == null) return null;
    else if(current.getLeftTree() == null) return current;
    else return minNode(current.getLeftTree());
}
```

Replacing a subtree with another subtree

```
private void transplant(TreeNode oldNode, TreeNode newNode)
{
    if(oldNode.parentNode == null) root = newNode;
    else if(oldNode.parentNode.getLeftTree() == oldNode)
    {
        oldNode.parentNode.leftNode = newNode;
    }
    else oldNode.parentNode.rightNode = newNode;
    if(newNode != null) newNode.parentNode = oldNode.parentNode;
}
```

Remove an element from a binary search tree

```
public void remove(Comparable element)
{
   removeNode(element, root);
}

private void removeNode(Comparable element, TreeNode current)
{
   if(current == null) return;
   else if (element.compareTo(current.value) == 0)
   {
      if(current.leftNode == null) transplant(current,current.rightNode);
      else if(current.rightNode == null) transplant(current,current.leftNode);
      else
      {
         TreeNode y  = minNode(current.rightNode);
         if(y.parentNode != current)
         {
            transplant(y,y.rightNode);
            y.rightNode = current.rightNode;
            y.rightNode.parentNode = y;
         }
         transplant(current,y);
         y.leftNode = current.leftNode;
         y.leftNode.parentNode = y;
      }
   }
   else if (element.compareTo(current.value) < 0)
   {
      removeNode(element,current.getLeftTree());
   }
   else removeNode(element,current.getRightTree());
}
```

Time Complexities

- For a tree of depth H, findNode, insertNode and removeNode are performed in $\mathcal{O}(H)$
- If we assume that insertions are purely random, than the depth of a binary search tree will be $\mathcal{O}(\log n)$ on average.
- Although this is intuitively clear, it is far from obvious to prove.

Tree Traversals, state space search

How to traverse a (binary search) tree

- A tree traverse is a method that visits each element in the tree and performs some operation on the element.
- Print all elements
- Call a method on each element
- Double each node value if it is a number
- ...

A recursive method to print the tree

```
private void printNode(TreeNode n)
{
    if (n != null)
    {
        System.out.println(n.value);
        printNode(n.leftNode);
        printNode(n.rightNode);
    }
}

public void print2()
{
    printNode(root);
}
```

The same, but using a stack to handle the recursion

```
public void recPrint()
{
    Stack t = new Stack();
    t.push(root);
    while(!t.empty())
    {
        TreeNode n = (TreeNode)t.pop();

        System.out.println(n.value);

        if(n.getRightTree() != null) t.push(n.getRightTree());
        if(n.getLeftTree() != null) t.push(n.getLeftTree());
    }
}
```

From depth first to breadth first

```
public void recPrint()
{
    Queue t = new Queue();
    t.push(root);
    while(!t.empty())
    {
        TreeNode n = (TreeNode)t.pop();

        System.out.println(n.value);

        if(n.getRightTree() != null) t.push(n.getRightTree());
        if(n.getLeftTree() != null) t.push(n.getLeftTree());
    }
}
```

What if we want to do something else than printing?

```
public void traverse(TreeAction action)
{
    Queue t = new Queue();
    t.push(root);
    while(!t.empty())
    {
        TreeNode n = (TreeNode)t.pop();
        action.run(n);
      if(n.getLeftTree() != null) t.push(n.getLeftTree());
        if(n.getRightTree() != null) t.push(n.getRightTree());
    }
}
```

The TreeAction class

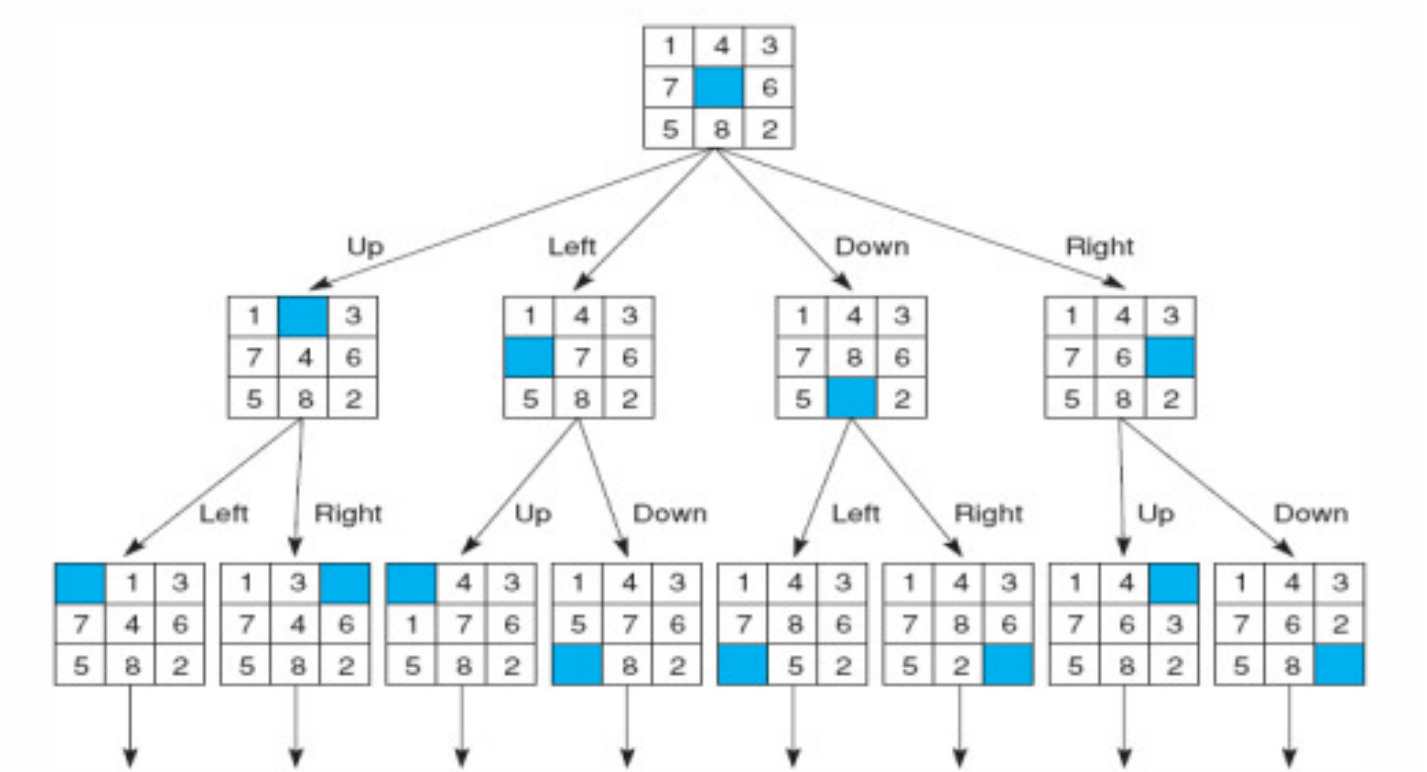
```
public abstract class TreeAction
{
    public abstract void run(TreeNode n);
}
```

Subclass from this one for each operation you want to perform.

JAVA allows to make anonymous classes to handle this

```
public void print()
{
    traverse(new TreeAction()
    {
        public void run(TreeNode n)
        {
            // put your code here
        }
    });
}
```

State space Search



State Space Search

As long as the agenda is not empty:

- The current state is the first one in the agenda.
- If the current state is the goal, we have found the goal.
- Otherwise, we generate all successor states, and add them to the agenda.

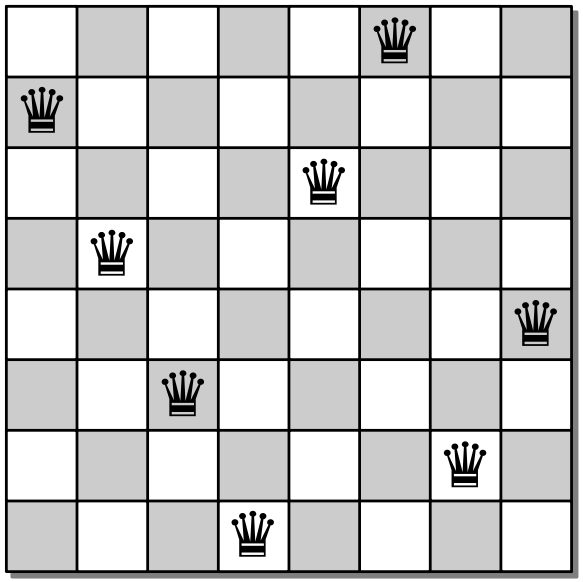
Types of search algorithms. See the Artificial Intelligence Course

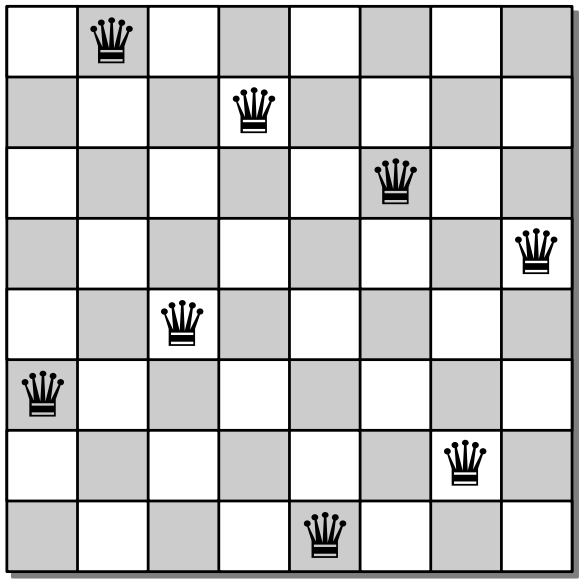
- Uninformed Search - blind search
  - Depth first search. Agenda is stack
  - Breadth first search. Agenda is queue
- Informed search - heuristic search
  - A*. Agenda is a priority queue.

State Space Search, an example

- Can we place 8 queens on a chessboard, such that no queen can take any other queen?
- Or in general, can we place n queens on an n-by-n board?
- The board is the state.
- A successor is a state with one more queen added to it.
- The goal is reached when there are n queens.

The eight queens problem





Example: The n-queens problem

```
public class NQueens {

    public NQueens(int n)
    {
        LinkedList visitedList = new LinkedList();
        ChessBoard state = new ChessBoard(n);
        Stack todoList = new Stack();
        todoList.push(state);
        while(!todoList.empty())
        {
            ChessBoard currentState = (ChessBoard)todoList.pop();
            visitedList.addFirst(currentState);
            if(currentState.goalReached())
            {
                System.out.println("GOAL REACHED");
                currentState.print();
                return;
            }
            else
            {
                LinkedList successorStates = currentState.successors();
                for(int i=0;i<successorStates.size();i++)
                {
                    ChessBoard next = (ChessBoard)successorStates.get(i);
                    if(!visitedList.find(next)) todoList.push(next);
                }
            }
        }
    }
}
```

Example: The n-queens problem

```
public class ChessBoard implements Comparable {

    private int size;
    private boolean board[][];
    private int count = 0;

    public ChessBoard(int n){}

    public ChessBoard(ChessBoard anotherBoard){}

    public boolean get(int i,int j){}

    public int getCount(){}

    public void set(int i,int j, boolean value){}

    public LinkedList successors(){}

    public boolean goalReached(){}

    public void print(){}

    private boolean canAdd(int x,int y){}

    public int compareTo(Object arg0) {}
}
```

Example: The n-queens problem

```
public ChessBoard(int n)
{
    size = n;
    board = new boolean[n][n];
    for (int i= 0;i<n;i++)
        for(int j=0;j<n;j++)
            board[i][j] = false;
}

public ChessBoard(ChessBoard anotherBoard)
{
    size = anotherBoard.size;
    board = new boolean[anotherBoard.size][anotherBoard.size];
    for (int i= 0;i<size;i++)
        for(int j=0;j<size;j++)
            board[i][j] = anotherBoard.get(i,j);
    count = anotherBoard.count;
}
```

Example: The n-queens problem

```
public boolean get(int i,int j)
{
   return board[i][j];
}

public int getCount()
{
   return count;
}

public void set(int i,int j, boolean value)
{
   if(board[i][j] && !value) count--;
   else if (!board[i][j] && value) count++;
   board[i][j] = value;
}
```

Example: The n-queens problem

```
public boolean goalReached()
{
    return count == size;
}

public void print()
{
    for (int i= 0;i<size;i++)
    {
        for(int j=0;j<size;j++)
        {
            System.out.print(board[i][j]);
            System.out.print(" ");
        }
        System.out.println();
    }
}
```

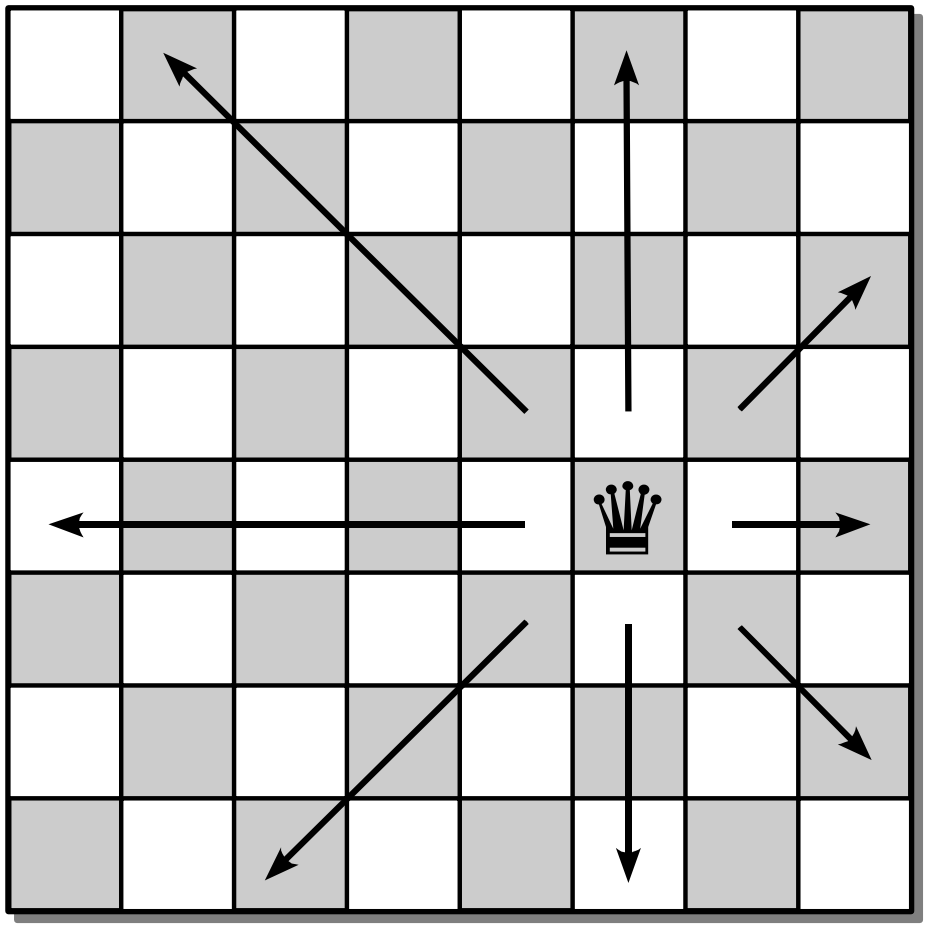
Example: The n-queens problem

```
public int compareTo(Object arg0) {
    ChessBoard another = (ChessBoard)arg0;
    for(int i=0;i<size;i++)
    {
        for(int j=0;j<size;j++)
        {
            if(board[i][j] != another.board[i][j]) return 1;
        }
    }
    return 0;
}
```

Example: The n-queens problem

```
public LinkedList successors()
{
    LinkedList result = new LinkedList();
    for (int i= 0;i<size;i++)
    {
        for(int j=0;j<size;j++)
        {
            if(canAdd(i,j))
            {
                ChessBoard child = new ChessBoard(this);
                child.set(i,j,true);
                result.addFirst(child);
            }
        }
    }
    return result;
}
```

Example: The n-queens problem



Example: The n-queens problem

```
public boolean canAdd(int x,int y)
{
    if(board[x][y]) return false;
    else
    {
        // if there is already a queen on the same row, return false
        for(int col=0;col<size;col++)
            if(board[x][col]) return false;
        // if there is already a queen on the same col, return false
        for(int row=0;row<size;row++)
            if(board[row][y]) return false;
        // if there is already a queen on any of the two diagonals, return false
        int r=x;
        int c=y;
        while((r<size)&&(c<size))
            if(board[r++][c++]) return false;
        r=x-1;c=y+1;
        while((r>=0)&&(c<size))
            if(board[r--][c++]) return false;
        r=x+1;c=y-1;
        while((r<size)&&(c>=0))
            if(board[r++][c--]) return false;
        r=x-1;c=y-1;
        while((r>=0)&&(c>=0))
            if(board[r--][c--]) return false;
        // else return true
        return true;
    }
}
```

Some Remarks

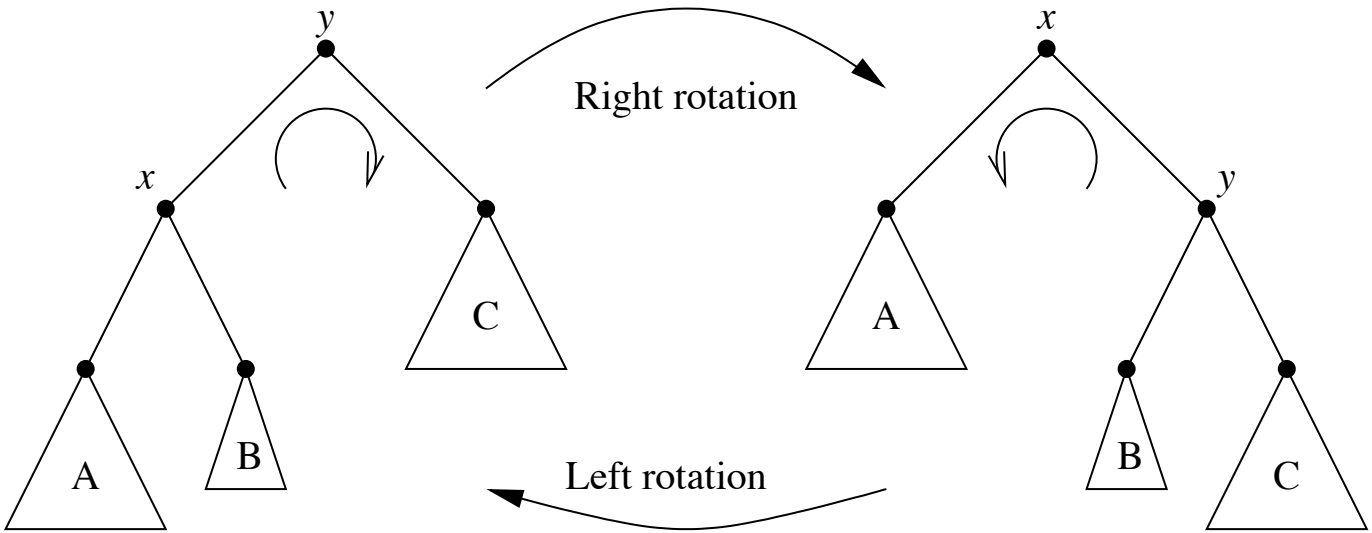
- It works!
- But we are a little bit naive in our implementation:
  - The visited list is $\mathcal{O}(n)$ for finding.
  - The board is 2D, comparing two n-by-n boards requires up to n^2 comparisons, $\mathcal{O}(n^2)$.
  - To generate a successor, the board is copied to a new board ($\mathcal{O}(n^2)$).
  - Generating the successors of a board also requires quite some checks at the horizontal, vertical and diagonal lines.

More advanced binary search trees

Two approaches for improving retrieval efficiency

- Guarantee that those elements that are accessed often are close to the root.
  - Keep the BST as is, when accessing move elements up.
  - Example: Splay Trees
- All Kinds of approaches to guarantee that the tree is more or less balanced.
  - If all paths have equal length, the length of the path H is $\log_2(n)$.
  - Example: Red-Black Trees

Towards Splay Trees: Left and Right Rotations



Rotations

```
private void rotateLeft(TreeNode x)
{
    TreeNode y = x.rightNode;
    x.rightNode = y.leftNode;
    if(y.leftNode != null) y.leftNode.parentNode = x;
    y.parentNode = x.parentNode;
    if(x.parentNode == null) root = y;
    else if(x == x.parentNode.leftNode) x.parentNode.leftNode = y;
    else x.parentNode.rightNode = y;
    y.leftNode = x;
    x.parentNode = y;
}
```

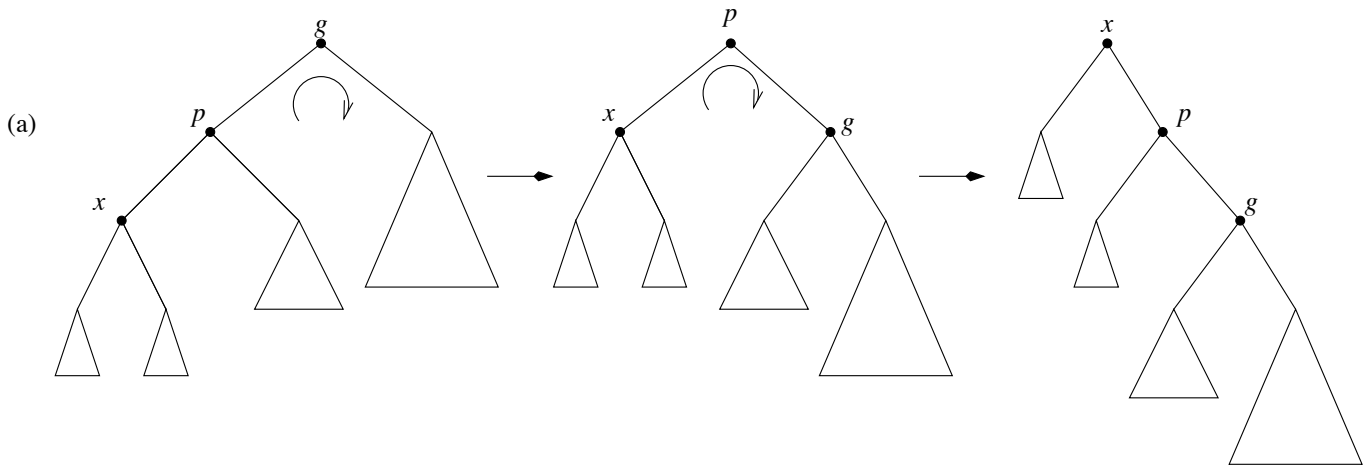
Rotations

```
private void rotateRight(TreeNode y)
{
    TreeNode x = y.leftNode;
    y.leftNode = x.rightNode;
    if(x.rightNode != null) x.rightNode.parentNode = y;
    x.parentNode = y.parentNode;
    if(y.parentNode == null) root = x;
    else if(y == y.parentNode.rightNode) y.parentNode.rightNode = x;
    else y.parentNode.leftNode = x;
    x.rightNode = y;
    y.parentNode = x;
}
```

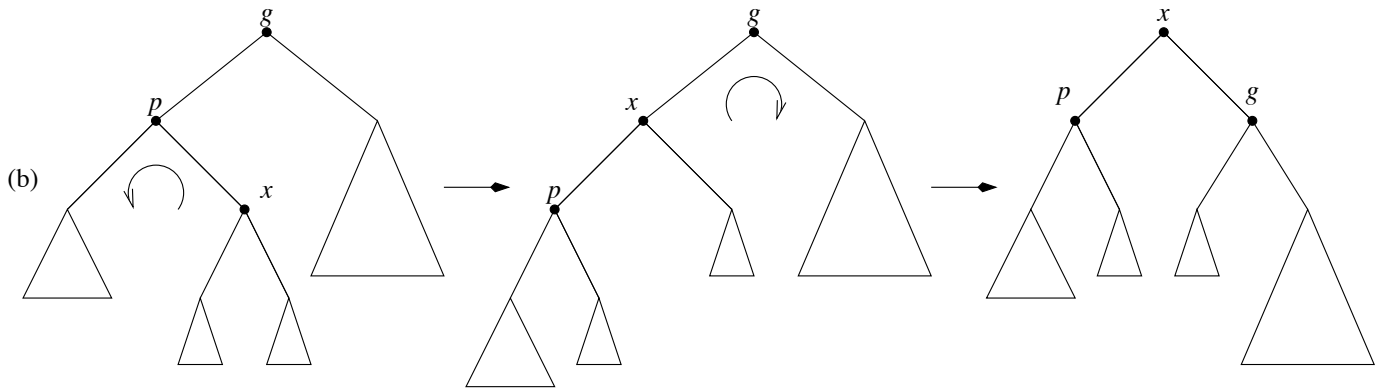
Splay Trees

- Upon access of the node x, Left and Right rotations are used to move x up to the root.
- If x is accessed in the near future, its access will be more efficient.
- Each rotation maintains the BST properties and can be done in constant time.
- Each rotation moves the node x one level up into the tree. If the tree has a depth H, then this extra cost for inserting elements is $\mathcal{O}(H) = \mathcal{O}(\log_2(n))$

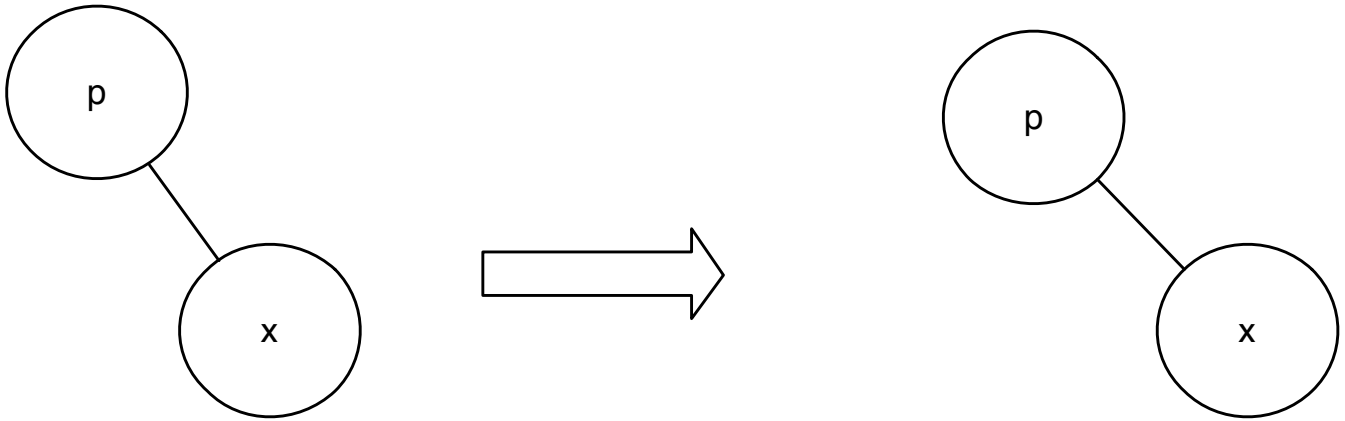
Example 1: splaying the x



Example 2: splaying the x

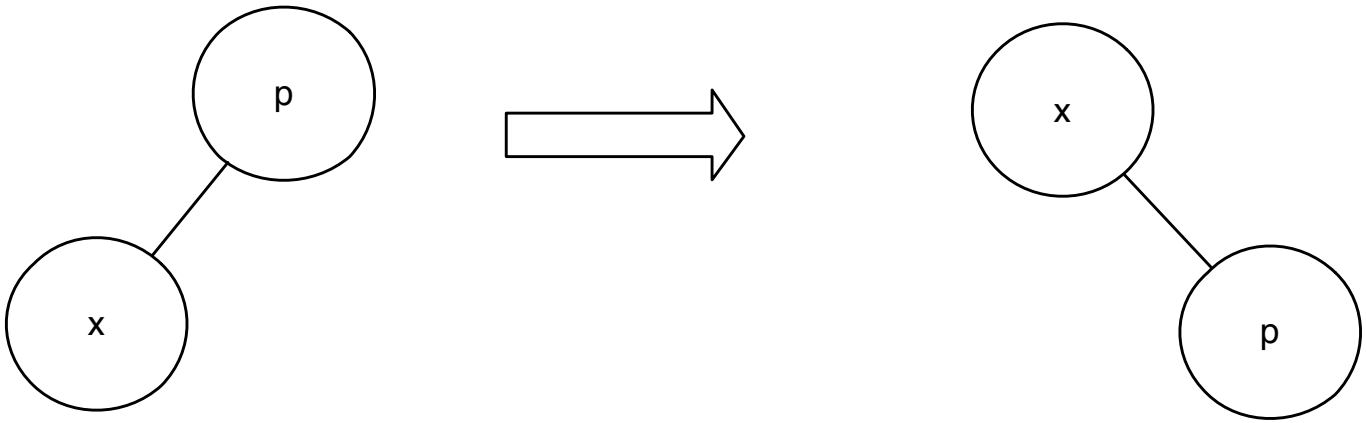


Case 0



x is right child

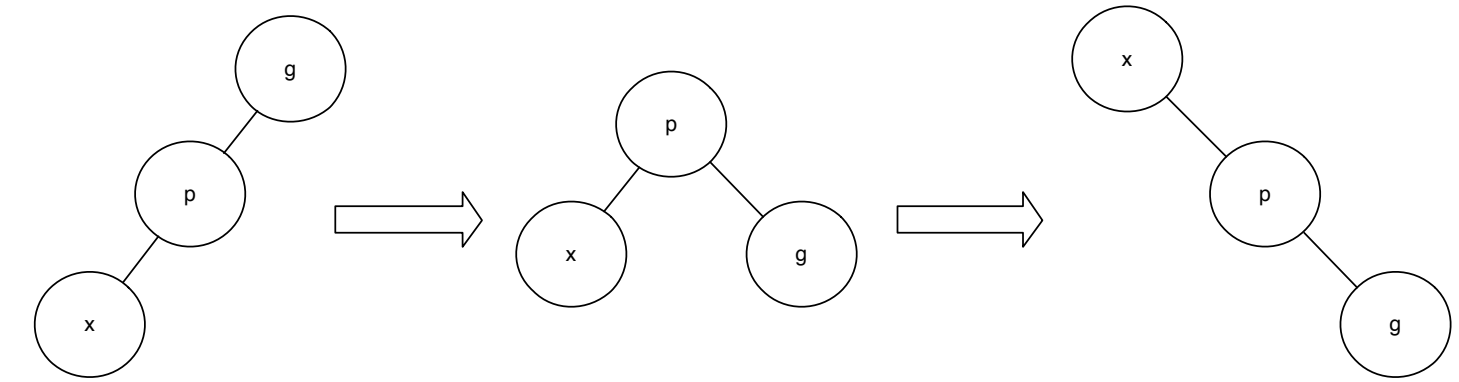
rotate g to the left

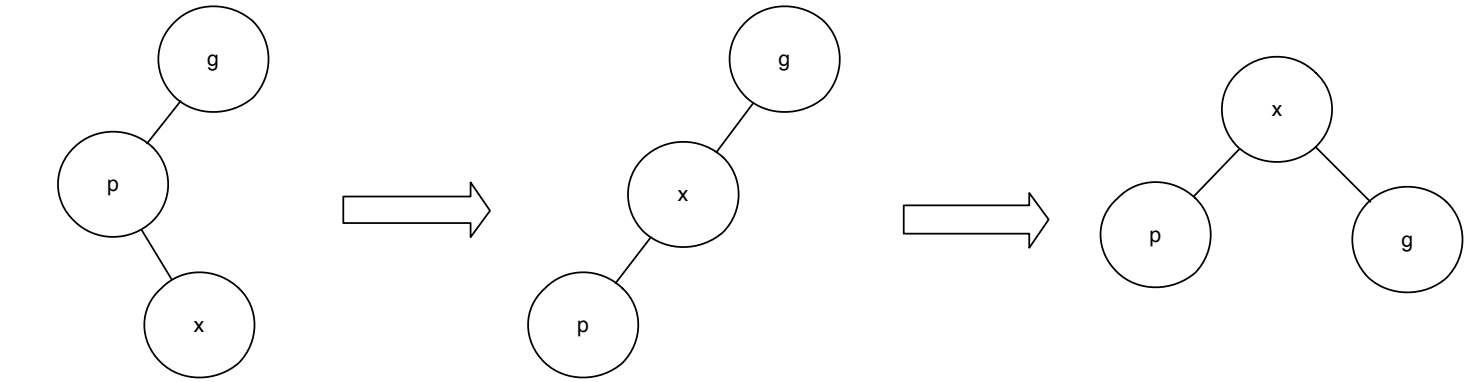


x is left child

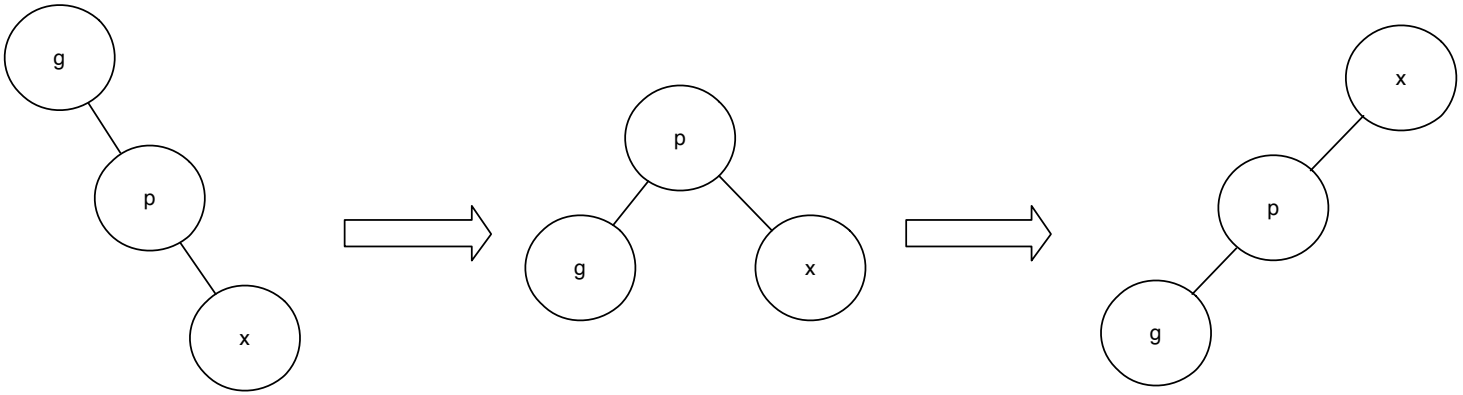
rotate p to the right

Case 1


g
p
x
p
x
g
x
p
g
p is left child
x is left child
rotate g to the right
rotate p to the right


g
p
x
g
x
p
x
p
g
p is left child
x is right child
rotate p to the left
rotate g to the right

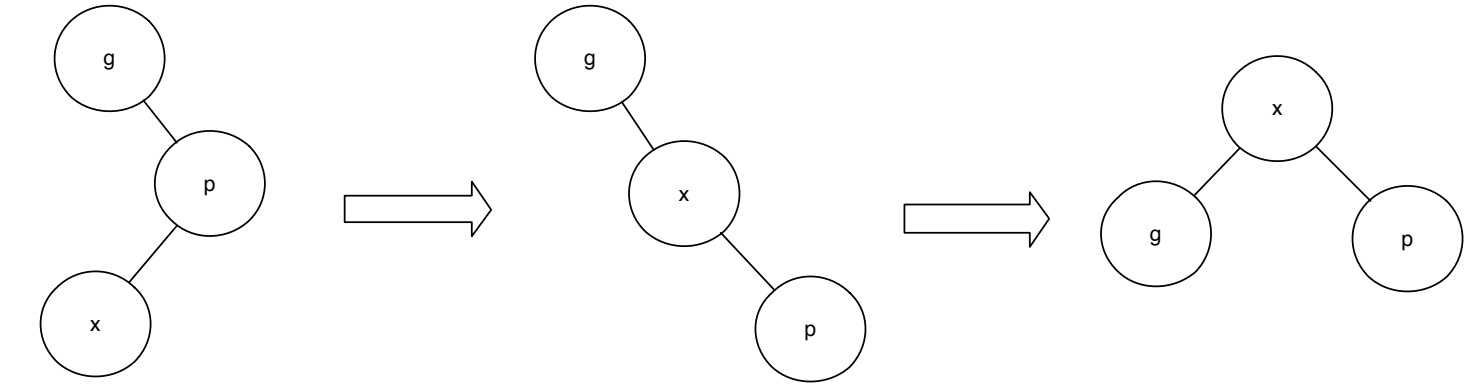
Case 2



p is right child
x is right child

rotate g to the left

rotate p to the left



p is right child
x is left child

rotate p to the right

rotate g to the left

Splay Trees

```
public class SplayTree extends BinarySearchTree
{
    public Comparable find(Comparable element)
    {
        TreeNode current = findNode(element,root);
        if(current != null) splay(current);
        return current;
    }
    private void rotateLeft(TreeNode x){}
    private void rotateRight(TreeNode y){}
    private void splay(TreeNode splayNode){}
}
```

```
private void splay(TreeNode splayNode)
{
    TreeNode parent, grandParent;
    while((parent = splayNode.parentNode) != null)
    {
        if((grandParent = parent.parentNode) == null)
        {
            if (splayNode.parentNode.leftNode == splayNode) rotateRight(parent);
            else rotateLeft(parent);
        }
        else
        {
            if(parent.parentNode.leftNode == parent)
            {
                if(splayNode.parentNode.leftNode == splayNode)
                {
                    rotateRight(grandParent); rotateRight(parent);
                }
                else
                {
                    rotateLeft(parent); rotateRight(grandParent);
                }
            }
            else
            {
                if(splayNode.parentNode.rightNode == splayNode)
                {
                    rotateLeft(grandParent); rotateLeft(parent);
                }
                else
                {
                    rotateRight(parent); rotateLeft(grandParent);
                }
            }
        }
    }
}
```

Splay Trees

- In splay tree we do not care about balancing the trees.
- We only make the access to recently accessed elements faster.
- Actually, a perfectly balanced tree can converted in a highly degenerated tree.

Red-Black Trees

Definition 8 (Red-Black Tree)

A red-black tree is a binary search tree that satisfies the following red-black properties:

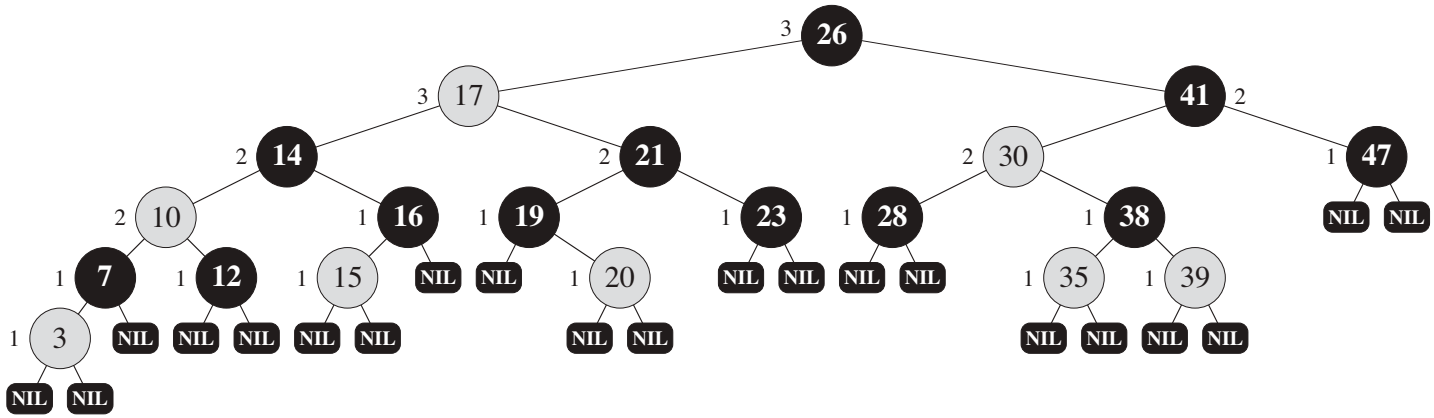
1. *Every node is either red or black.*
2. *The root is black.*
3. *Every leaf is black.*
4. *If a node is red, then both children are black.*
5. *For each node, all simple paths from the node to the descendant leafs, contain the same number of black nodes.*

Red-Black Trees: Ensuring that the tree is well balanced

Definition 9 (black-height)

The black-height of a node x $bh(x)$ in a red-black tree is the number of black nodes on any simple path from, but not including, a node x down to a leaf.

Red-Black Trees: An example tree



Lemma 1

The subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.

Proof.

The proof is based on induction on the height h of the tree x.
If $h(x) = 0$, then x is a leaf. Then $bh(x) = 0$, so $2^{bh(x)} - 1 = 0$ and the statement holds.
If however $h(x) \geq 0$, then x has two children. Each child has a black-height of either $bh(x)$ or $bh(x) - 1$, depending on whether its colour is red or black. So, black height is *at least* $bh(x) - 1$.
As the height of the subtrees is smaller than $h(x)$, we apply the induction hypothesis and find that each child has at least $2^{bh(x)-1} - 1$ internal nodes. So, the subtree rooted at x has at least
$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ nodes □

Lemma 2

A red-back tree with n internal nodes has a height of at most $2\log(n+1)$.

Proof.

Let h be the height of the tree. According to red-black property 4, at least half of the nodes on the any simple path from root to leaf, not including the root, is black. Hence, the back-height of the root must be at least $h/2$, hence $n \geq 2^{h/2} - 1$, hence $n + 1 \geq 2^{h/2}$, and consequently $\log_2(n+1) \geq h/2$ and $h \leq 2\log_2(n+1)$. □

```
public class RedBlackTree
{
    public enum TreeNodeColor {Red, Black}

    protected class ColouredTreeNode implements Comparable {}

    protected ColouredTreeNode root;

    private void rotateLeft(ColouredTreeNode x)
    {}

    private void rotateRight(ColouredTreeNode y)
    {}

    public void insert(Comparable element)
    {
        ColouredTreeNode newNode = new ColouredTreeNode(element, TreeNodeColor.Red);
        insertAtNode(newNode, root, null);
        fixUp(newNode);
    }

    protected void insertAtNode(ColouredTreeNode newNode,
    ColouredTreeNode current, ColouredTreeNode parent)
    {}


    private void fixUp(ColouredTreeNode z)
    {}
}
```

```
protected class ColouredTreeNode implements Comparable
   {
      protected TreeNodeColor color;
      protected Comparable value;
      protected ColouredTreeNode leftNode;
      protected ColouredTreeNode rightNode;
      protected ColouredTreeNode parentNode;

      public ColouredTreeNode(Comparable value, TreeNodeColor color)
      {
         this.value = value;
         this.color = color;
      }

      public String toString()
      {
         return value.toString() + " : " + color;
      }

      public int compareTo(Object arg0)
      {
         ColouredTreeNode node2 = (ColouredTreeNode)arg0;
         return value.compareTo(node2.value);
      }
   }
```

Red Black Tree: Inserting an element

- We keep the same insert method as in the original binary search tree.
- The node we add is coloured red.
- After insertion, we are guaranteed we still have a binary search tree.
- Which red-black tree properties can be violated by inserting this way?

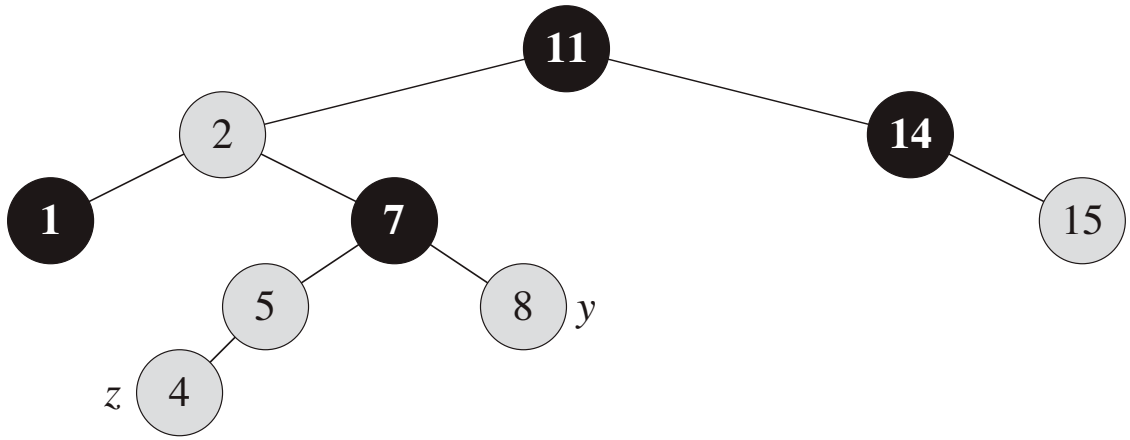
Red Black Tree: Possible violations to red-black tree properties after inserting a node z

1. Every node is either red or black. OK
2. The root is black. NOT OK (if z is the root)
3. Every leaf is black. OK
4. If a node is red, then both children are black. NOT OK (if the parent of z is also red)
5. For each node,all simple paths from the node to the descendant leafs, contain the same number of black nodes. OK

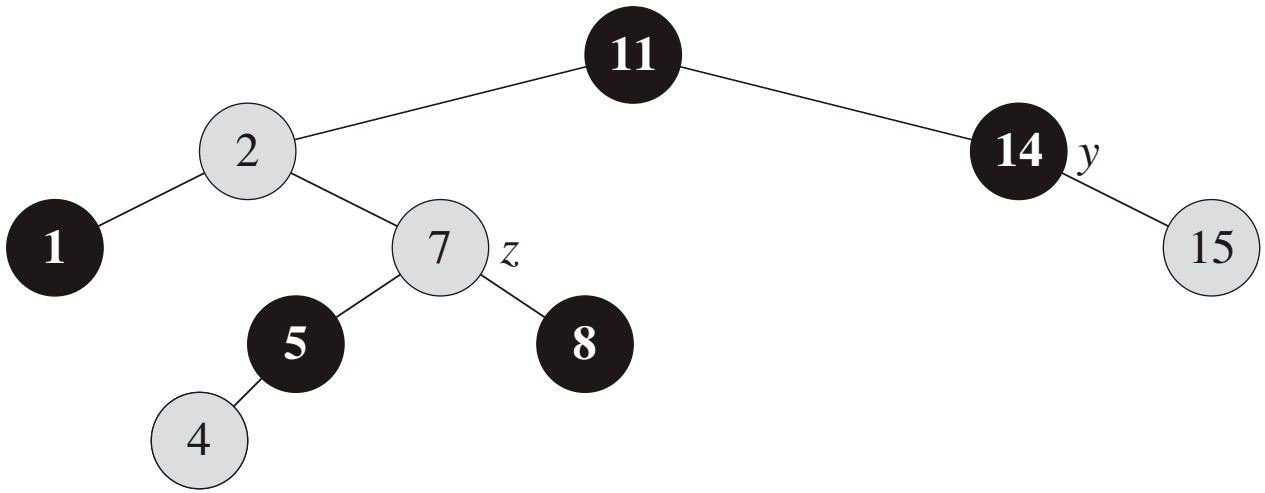
Red Black Tree: Possible violations to red-black tree properties after inserting a node z

- If the binary search tree meets all the red-black tree properties, except property 2 (the root is black), then a simple recolouring of the root to black is possible. This does never violate any of the other red-black tree properties.
- If the binary search tree violates the fourth property (if a node is red then both the children are black), then we first solve this violation.
  - This solution should not violate red-black properties 1, 3 and 5.
  - If the solution violates red-black property 2, then see above.

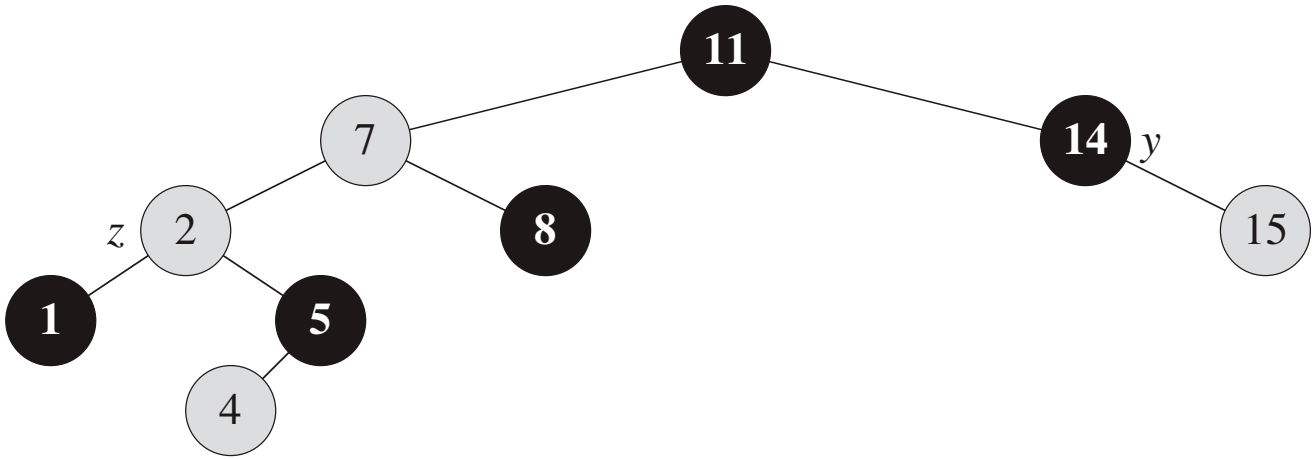
Example: Insert 4 and colour it red



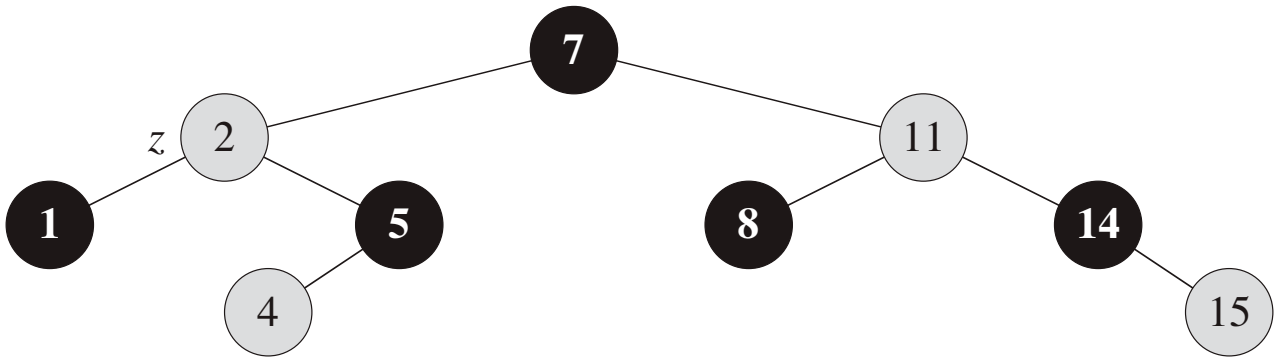
4 is red so 5 and 8 must become black, 7 becomes red.
But now 7 and 2 are red ...



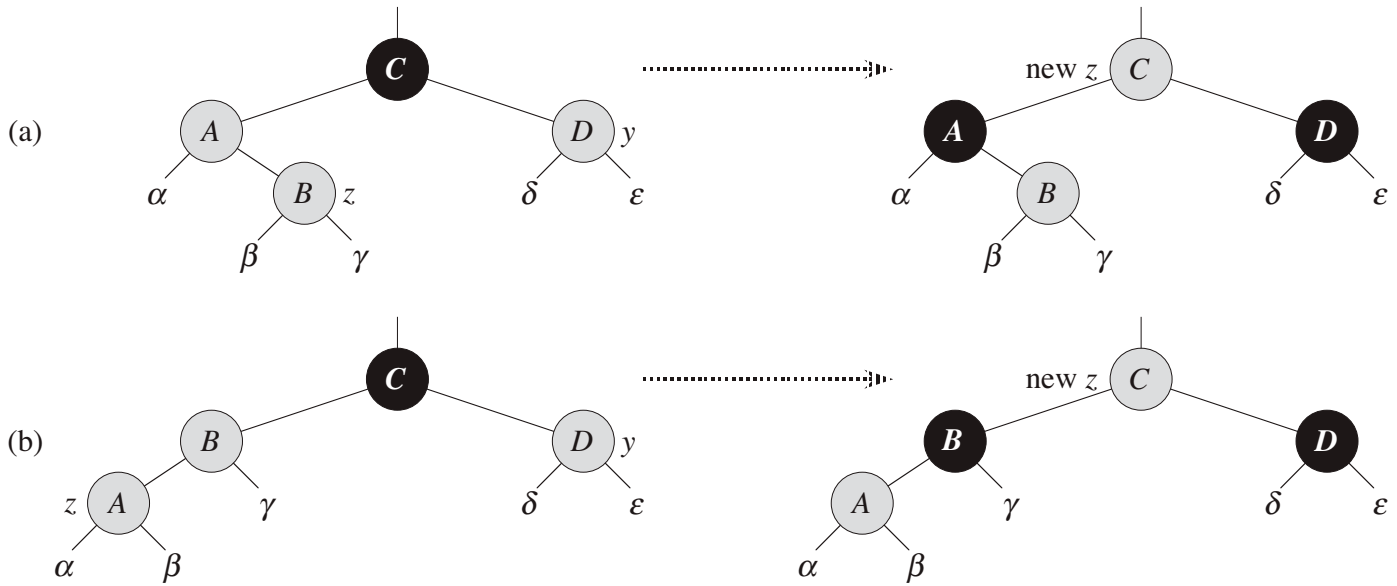
7 and 14 must become black, but 14 is already black, so we can't recolor easily. Left rotate "the problem 7" to move it higher up in the tree



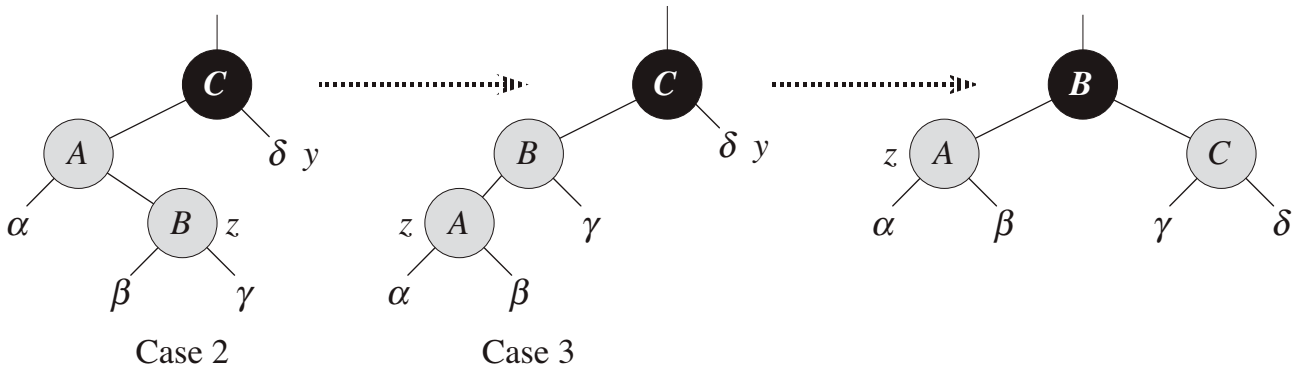
Right rotate "the problem 7" to move it even higher up in the tree. Recolor once it becomes root



Case 1: z is red, its parent is red, its uncle y is red. Recolor



Case 2: z is red, its parent is red, its uncle y is black. We can't recolor but we rotate.



case 2: z is a right child, case 3: z is a left child

```
private void fixUp(ColouredTreeNode z)
   {
      while((z != null) && (z.parentNode != null) && (z.parentNode.parentNode != null)
      && (z.parentNode.color == TreeNodeColor.Red))
      {
         if(z.parentNode == z.parentNode.parentNode.leftNode)
         {
            ColouredTreeNode y = z.parentNode.parentNode.rightNode;
            if(y.color == TreeNodeColor.Red)
            {
               z.parentNode.color = TreeNodeColor.Black;
               y.color = TreeNodeColor.Black;
               z.parentNode.parentNode.color = TreeNodeColor.Red;
               z = z.parentNode.parentNode;
            }
            else
            {
               if (z == z.parentNode.rightNode)
               {
                  z = z.parentNode;
                  rotateLeft(z);
               }
               z.parentNode.color = TreeNodeColor.Black;
               z.parentNode.parentNode.color = TreeNodeColor.Red;
               rotateRight(z.parentNode.parentNode);
            }
         }
         else
         {
            // exactly the same, with left and right swapped ...
         }
      }
      if(z == root) root.color = TreeNodeColor.Black;
   }
```

Red Black Trees in summary

- Trees are guaranteed to be balanced.
- Find in $O(\log n)$.
- Insert is a find + a fix among the path from the inserted element upto the root. The fix involves recoloring and rotations. $O(\log n)$.
- Remove is somewhat similar, just a bit more complicated ;-) $O(\log n)$.

Graphs

Terminology

- Vertex (vertices) : node
- Edge: connection between two vertices.
- Edges can have a weight: weighted graphs and unweighted graphs
- So any graph $G = (V, E)$ is fully defined by the sets V of vertices and E of edges
- For weighted graphs, there is also a *weight function* $w : E \to \mathcal{R}$, so $w(u, v)$ is the weight of the edge $(u, v) \in E$.

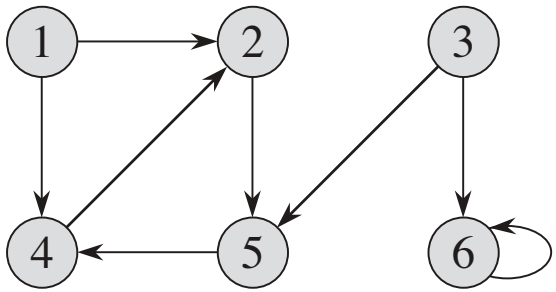
More terminology

- Edges can have a direction (marked by the arrow). Directed and undirected graphs.
- Two vertices i and j are *adjacent* if they are connected by an edge (i, j). The edge (i, j) is said to be *incident on* the vertices i and j.
- The *degree* of a vertex in an undirected graph is the number of edges incident on it.
- In a directed graph, the out-degree is the number of edges leaving it, the in-degree is the number of edges entering it. The *degree* of a vertex in a directed graph is hence the sum of its in-degree and its out-degree.

Even more terminology

- A *path* is a sequence of adjacent vertices.
- If there is a path from every vertex to every other vertex, the graph is *connected*.
- Any path starting from a given node that terminates in the same node, is called *a cycle*. Cyclic and Acyclic graphs.
- Directed graphs without cycles are often referred to as *directed acyclic graphs* or *DAG*.
- A subset of a graph is called a *subgraph*. The graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

A graph and its matrix representation



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Adjacancy matrix representation

Definition 10 (Adjacancy matrix)

if the vertices of some Graph $G = (V, E)$ *are somehow numbered* $1, 2, ..., |V|$, *then the adjaceny matrix representation is a matrix* $A = (a_{ij})$ *of size* $|V|$ *by* $|V|$ *where* $a_{ij} = \begin{cases} w(i,j) & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$

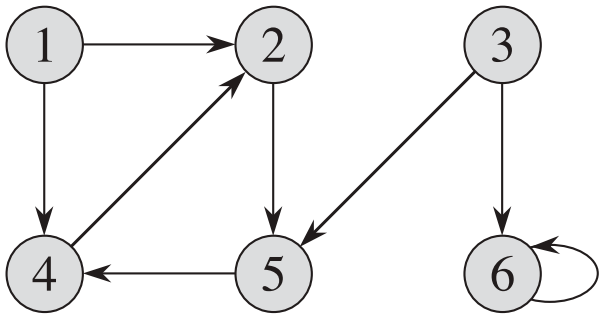
A matrix class

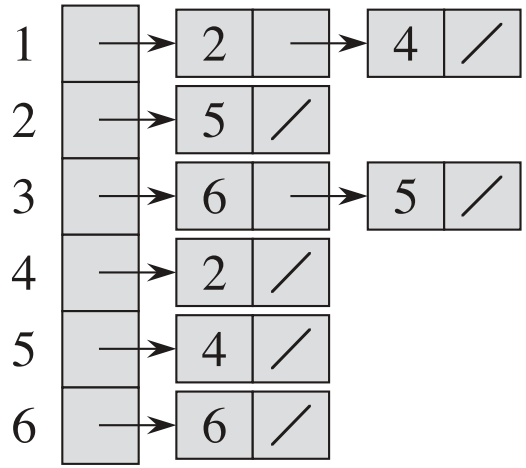
```
public class Matrix
{
    // some appropriate private members.

    public Matrix(int nrNodes)
    {
        // allocate an N-by-N matrix where N = nrNodes
        // all elements are initially 0
    }

    public void set(int row, int col, Comparable weight)
    {
        // store the weight at the given row and column.
    }

    public Comparable get(int row, int col)
    {
        // return the weight at the given row and column.
    }
}
```

Adjacancy matrix representation of a matrix

```
public class MatrixGraph
{
    private Matrix data;

    public MatrixGraph(int nrNodes)
    {
        data = new Matrix(nrNodes);
    }

    public void addEdge(int from, int to, double w)
    {
        data.set(from, to, w);
    }

    public double getEdge(int from, int to)
    {
        return (Double)data.get(from, to);
    }
}
```

Edge list representation





Edge list representation

```
public class Node implements Comparable
{
    private Comparable info;
    private Vector edges;

    public Node(Comparable label)
    {
        info = label;
        edges = new Vector();
    }

    public void addEdge(Edge e)
    {
        edges.addLast(e);
    }

    public int compareTo(Object o)
    {
        // two nodes are equal if they have the same label
        Node n = (Node)o;
        return n.info.compareTo(info);
    }

    public Comparable getLabel()
    {
        return info;
    }
}
```

Edge list representation

```
private class Edge implements Comparable
{
    private Node toNode;
    public Edge(Node to)
    {
       toNode = to;
    }

     public int compareTo(Object o)
     {
         // two edges are equal if they point
         // to the same node.
         // this assumes that the edges are
         // starting from the same node !!!
         Edge n = (Edge)o;
         return n.toNode.compareTo(toNode);
     }
}
```

Edge list representation

```
public class Graph
{
    private class Node implements Comparable
    {
        private Comparable info;
        private Vector edges;

        public Node(Comparable label)
        {
            info = label;
            edges = new Vector();
        }

        public void addEdge(Edge e)
        {
            edges.addLast(e);
        }

        public int compareTo(Object o){ ... }

        public Comparable getLabel()
        {
            return info;
        }
    }

    private class Edge implements Comparable
    {
        private Node toNode;
        public Edge(Node to)
        {
            toNode = to;
        }
```

Edge list representation

```
private Node findNode(Comparable nodeLabel)
{
    Node res = null;
    for (int i=0; i<nodes.size(); i++)
    {
      Node n = (Node)nodes.get(i);
        if(n.getLabel() == nodeLabel)
        {
            res = n;
            break;
        }
    }
    return res;
}
```

This is almost the find method on the vector ... put the right responsibility in the right datastructure.

Remember ...

```
public boolean contains(int obj)
{
    for(int i=0;i<count;i++)
    {
        if(data[i] == obj) return true;
    }
    return false;
}
```

Better would be ...

```
public Comparable contains(Comparable obj)
{
    for(int i=0;i<count;i++)
    {
        if(data[i].compareTo(obj) == 0) return obj;
    }
    return null;
}
```

Better would be ...

```
protected Node findNode(Comparable nodeLabel)
{
    return (Node) nodes.contains(new Node(nodeLabel));
}
```

FInding a path between two vertices as a depth first graph traversal

```
public boolean findPath(Comparable nodeLabel1,
                        Comparable nodeLabel2)
{
    Node startState = findNode(nodeLabel1);
    Node endState = findNode(nodeLabel2);
    Stack toDoList = new Stack();
    toDoList.push(startState);
    while(!toDoList.empty())
    {
        Node current = (Node)toDoList.pop();
        if (current == endState)
            return true;
        else
      {
            for (int i=0; i<current.edges.size(); i++)
            {
                Edge e = (Edge)current.edges.get(i);
                toDoList.push(e.toNode);
            }
        }
    }
    return false;
}
```

Some Graph algorithms and problems

Depth first search, depth first traversals, ...

- The DFS on the previous slide does not have a visited list.
- Just as in the tree search, it can easily be added, but it is time consuming to check the visited list at each step.
- Keep a boolean in each node to mark whether it was visited before or not.
- Recursive version

Depth First Graph Traversal

```
private void DFS(Node current)
{
    current.visited = true;
    for(int i=0;i<current.edges.size();i++)
    {
        Edge e = (Edge)current.edges.get(i);
        Node next = (Node)e.toNode;
        if(!next.visited) DFS(next);
    }

}

public void DFS()
{
    for(int i=0;i<nodes.size();i++)
    {
        Node current = (Node)nodes.get(i);
        current.visited = false;
    }
    for(int i=0;i<nodes.size();i++)
    {
        Node current = (Node)nodes.get(i);
        if(!current.visited) DFS(current);
    }
}
```

Time Complexity of DFS

- Set all Nodes to unvisited: $\mathcal{O}(|V|)$
- How many calls to DFS are there?
- For each node, if it is not visited yet, then visit it.
  - For each edge in the adjacency list, if not visited yet, visit the destination of the edge.
- So, the number of DFS calls is the sum of all nr of nodes in the adjacency lists of all the nodes.
- $sum_{v \in V} |adj(v)| = \mathcal{O}(|E|)$
- Time complexity of a full depth first traversal is $\mathcal{O}(|V| + |E|)$.

Topological Sorting

Definition 11 (Topological Sorting)

A topological sort of a dag G=(V,E) is a linear ordering of all its vertices such that if G contains an edge (u,v), then u appears before v in the ordering.

DFS exactly gives this ordering (see how they are printed)!

```
private void DFS(Node current)
{
    current.visited = true;
    for(int i=0;i<current.edges.size();i++)
    {
        Edge e = (Edge)current.edges.get(i);
        Node next = (Node)e.toNode;
        if(!next.visited) DFS(next);
    }
    System.out.println(current.info);
}

public void DFS()
{
    for(int i=0;i<nodes.size();i++)
    {
        Node current = (Node)nodes.get(i);
        current.visited = false;
    }
    for(int i=0;i<nodes.size();i++)
    {
        Node current = (Node)nodes.get(i);
        if(!current.visited) DFS(current);
    }
}
```

Cycle checking

- If there is a cycle, there does not exist a topological sorting.
- So, cycle checking in $\mathcal{O}(|V| + |E|)$.

Definition 12

Back edges are those edges (u,v) connecting a vertex u to an ancestor v in a depth-first tree. We consider self-loops, which may occur in directed graphs, to be back edges.

Theorem 10

A directed graph G is acyclic if and only if a depth-first search of G yields no back edges.

Proof.

Assume G has a cycle c. Define v to be the first vertex in c discovered in the DFS. Let (u,v) be an edge in c. The DFS from v will explore all vertices reachable from v, also u. So, there is a path from v to u in G. So, (u,v) is a back edge.
Assume the DFS results in a back edge (u,v). Then v is an ancestor of u in G. So, there is a path from u to v in G. This path, together with (u,v) forms a cycle.

Towards cycle checking in adjacency matrix notation

- In an acyclic graph with n vertices, there can be at most n edges.
- In an acyclic graph with n vertices, the longest path can contain at most n - 1 edges.

Towards cycle checking in adjacency matrix notation

- Let A be the adjacency representation of G.
- If $trace(A) > 0$, then there is at least one self loop. (i.e. a cycle of length 1)
- If $trace(A * A) > 0$, then there is at least one cycle of length 2.
- If $trace(A^n) > 0$, then there is at least one cycle of length n.
- Consequently, a graph with n nodes has no cycles if for all k smaller than n, $\text{trace}(A^k) = 0$.

Why is that?

- Consider C = A x B, then $c_{ij} = \sum_{k=1}^{n} a_{ik} bkj$
- So in A^2, $a_{ij} = \sum_{k=1}^{n} a_{ik} akj$
- $a_{ij} > 0$ in A^2 if there is a node k such that there is an edge (i,k) and an edge (k,j)
- (which is a path of length 2).
- So, in A^2, an element on the diagonal is larger than zero if there is an element with a path of length two to itself.

Shortest path algorithms

- We are looking for the path with the smallest total weight.
- We are assuming there is no heuristic available (otherwise A^*).
- We focus on algorithms for finding *Single-Source shortest paths*, i.e. algorithms that find the shortest path to all reachable nodes from a given node.

Shortest path algorithms

Theorem 11 (Subpaths of a shortest path are also shortest paths)

Let $p = v_0 \dots v_k$ be the shortest path from v_0 to v_k and let $p_{ij} = v_i \dots v_j$ with $0 \leq i \leq j \leq k$ be the subpath from v_i to v_j, then p_{ij} is the shortest path from v_i to v_j.

Proof.

p can be written as $v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p_{ij}} v_j \xrightarrow{p_{jk}} v_k$ and hence $w(p) = w(p_{0i}) + w(p_{ij}) + w(p_{jk})$. Assume there is a path p'_{ij} from v_i to v_j such that $w(p'_{ij}) < w(p_{ij})$, then $p' = v_0 \xrightarrow{p_{0i}} v_i \xrightarrow{p'_{ij}} v_j \xrightarrow{p_{jk}} v_k$ is a path from v_0 to v_k with a weight $w(p') = w(p_{0i}) + w(p'_{ij}) + w(p_{jk}) < w(p)$. This contradicts the assumption that p is the shortest path from v_0 to v_k.

Shortest path algorithms

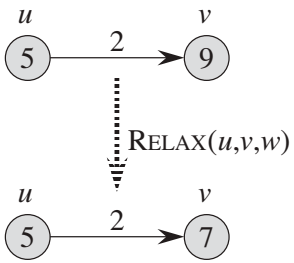
Theorem 12 (Shortest paths and cycles)

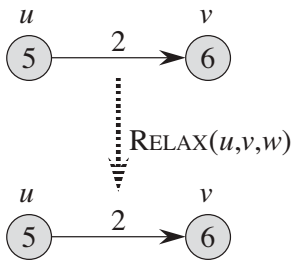
A shortest path cannot have cycles.

Relaxation Based Algorithms

- Many shortest path algorithms are based on a relaxation principle:
- To each vertex $v \in V$ a number $v.d$ is associated, representing the current estimated distance from the source vertex s to the current vertex v.
- For all nodes $v \in V$, $v.d = \infty$, except for $s.d = 0$.
- The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and, if so, updating $v.d$.

Relaxation





RELAX(u, v, w)

1 **if** $v.d > u.d + w(u, v)$
2 $\quad v.d = u.d + w(u, v)$
3 $\quad v.\pi = u$

The Bellman-Ford Algorithm

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3       for each edge (u, v) ∈ G.E
4            RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6       if v.d > u.d + w(u, v)
7            return FALSE
8  return TRUE
```

Proving the Bellman-Ford Algorithm

Lemma 3 (The path relaxtion property)

If $p = v_0 \dots v_k$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$ ($\delta(s, v_k)$is the weight of the shortest path from s to v_k).

Proving the Bellman-Ford Algorithm

Theorem 13

Let G=(V,E) be a weighted, directed graph with source s and weight function $w : E \to \mathcal{R}$, and assume that G contains no (negative-weight) cycles that are reachable from s. Then, after the $|V| - 1$ iterations of the for loop of lines 24 of BELLMAN-FORD, we have $v.d = \delta(s, v)$ for all vertices v that are reachable from s.

Proof.

Consider any vertex v that is reachable from s, and let $p = v_0 \ldots v_k$, where $v_0 = s$ and $v_k = v$, be any shortest path from s to v. Because shortest paths are simple, p has at most $|V| - 1$ edges, and so $k \leq |V| - 1$. Each of the $|V| - 1$ iterations of the for loop of lines 24 relaxes all $|E|$ edges. Among the edges relaxed in the ith iteration, for $i = 1, \ldots, k$ is (v_{i-1}, v_i). By the path-relaxation property, therefore, $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$.

The Bellman-Ford Algorithm

- This is a straight forward approach that runs in $\mathcal{O}(VE)$.
- The longest path (without cycle) has $|V| - 1$ edges.
- It repeats $|V| - 1$ times a relaxation on all edges.
- At each step i, at least the edge $(v_i - 1, v_i)$ is relaxed.
- (So, if we could traverse the nodes in "the right order", we would need far less passes.)

DAG-SHORTEST-PATHS(G, w, s)

```
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4       for each vertex v ∈ G.Adj[u]
5            RELAX(u, v, w)
```

Sorting

Insertion Sort on Linked Lists

- A very simple idea.
- Traverse the list and for each element, insert it at the right position in a new list.
- Almost no code due to addSorted on Linked List.
- The list is copied rather than sorted *in place*.
- $\mathcal{O}(n^2)$.
- Not good, but it is a start.

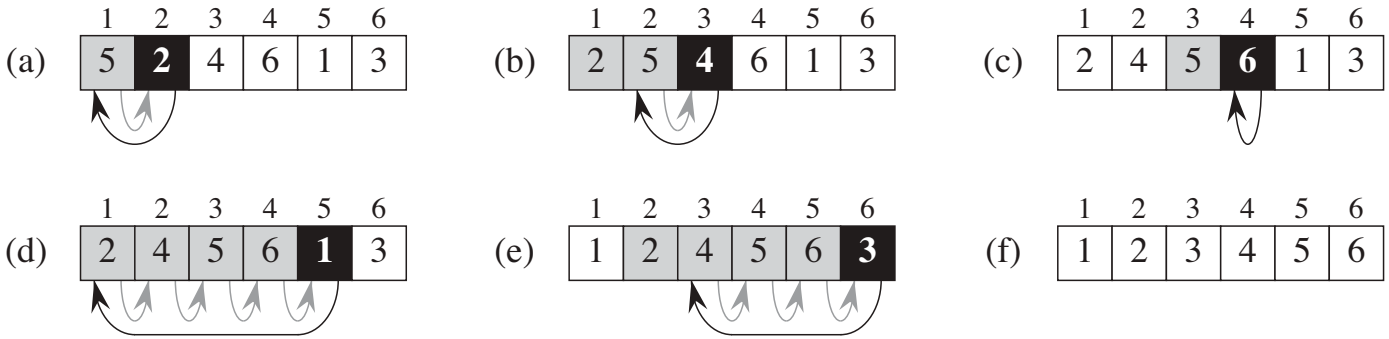
Insertion Sort: Simple, a naive version

```
public LinkedList insertionSort()
{
    LinkedList result = new LinkedList();
     ListElement d = head;
     while(d!= null)
     {
       result.addSorted(d.first());
        d = d.rest();
     }
     return result;
}
```

Insertion Sort: on vector, in place and more efficient

- We consider the vector to have two parts: a first part that is already sorted and a second part that still needs to be sorted.
- Start with the first element in the first part and the second part to be the rest.
- Take all elements from the still to be sorted part and move them one by one in the sorted part.

Insertion Sort: on vector, in place and more efficient



Insertion Sort: on vector, in place and more efficient

```
private void insertSorted(Comparable o, int last)
{
    if(isEmpty()) data[0] = o;
    else
    {
        int i = last;
        while((i>=0)&&(data[i].compareTo(o) > 0))
        {
            data[i+1] = data[i];
            i--;
        }
        data[i+1]=o;
    }
}

public void insertionSort()
{
    for(int i = 1;i<count;i++)
    {
        insertSorted(data[i],i-1);
    }
}
```

Tree Sort

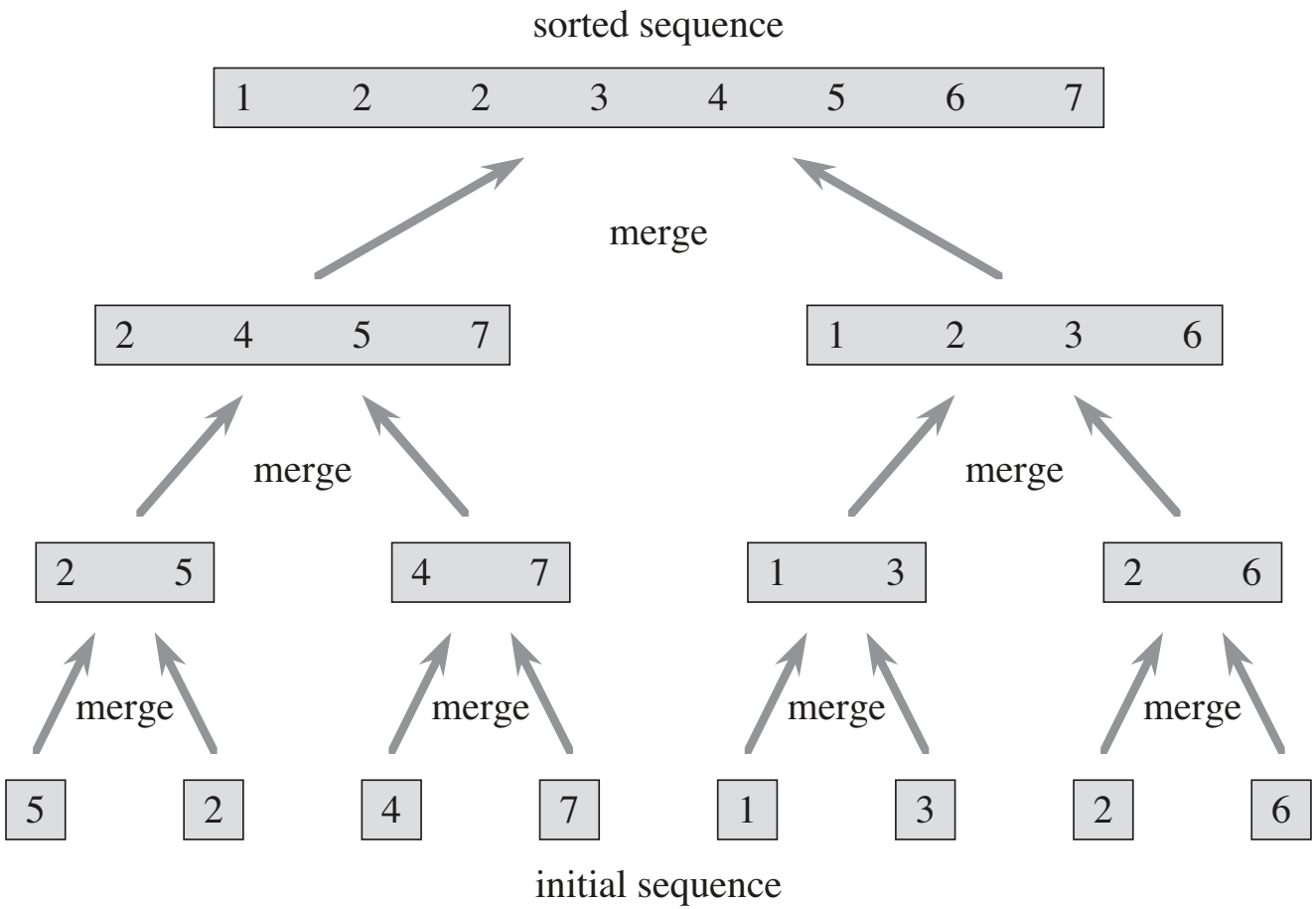
- Was explained when introducing BST
- Traverse the list, for each element, insert it in a BST.
- Traverse the BST and insert each element in a new Linked List
- (if the tree is balanced) $n\mathcal{O}(\log n) + n\mathcal{O}(1) = \mathcal{O}(n \log n)$

Tree Sort

```
public LinkedList treeSort()
{
    Tree sortTree = list2Tree();
    final LinkedList result = new LinkedList();
    sortTree.traverseInOrder(new TreeAction()
    {
        public void run(Tree.TreeNode n)
        {
            result.addFirst(n.getValue());
        }
    });
    return result;
}

public Tree list2Tree()
{
    Tree sortTree = new Tree();
    ListElement d = head;
    while(d!= null)
    {
        sortTree.insert(d.first());
        d = d.rest();
    }
    return sortTree;
}
```

Merge Sort: A divide and conquer approach

- Split the vector in two parts: the first half and the second half.
- Sort the first half and Sort the second half.
  - This is done by doing the merge sort on the first half and the second half.
  - If a vector has no elements or one element, the sorting is trivial.
- Merge both sorted halves together.


sorted sequence
1 2 2 3 4 5 6 7
merge
2 4 5 7
1 2 3 6
merge
merge
2 5
4 7
1 3
2 6
merge
merge
merge
merge
5
2
4
7
1
3
2
6
initial sequence

Merging two sorted vectors

```
static Vector merge(Vector v1, Vector v2)
{
    int lengthC = v1.size()+ v2.size();
    Vector result = new Vector();
    int indA = 0;
    int indB = 0;
    int indC = 0;
    while((indC < lengthC) && (indA < v1.size())&&
          (indB < v2.size()))
    {
        if(v1.get(indA).compareTo(v2.get(indB)) < 0)
            result.set(indC++, v1.get(indA++));
        else result.set(indC++, v2.get(indB++));
    }

    for(;indA < v1.size();indA++)
    {
        result.set(indC++, v1.get(indA));
    }
    for(;indB < v2.size();indB++)
    {
        result.set(indC++, v2.get(indB));
    }
    result.count = lengthC;

    return result;
}
```

Merge Sort

```
static Vector mergeSortAux(Vector v, int from, int to)
{
    if(to > from)
    {
        int half = (from + to) / 2;
        Vector v1 = mergeSortAux(v,from, half);
        Vector v2 = mergeSortAux(v,half+1,to);
        Vector result = merge(v1,v2);
        return result;
    }
    else
    {
        Vector result = new Vector();
        result.addLast(v.get(from));
        return result;
    }
}

static Vector mergeSort(Vector v)
{
    return mergeSortAux(v,0, v.count-1);
}
```

Time Complexity of Merge Sort

Theorem 14

The time complexity of merge sort is $\mathcal{O}(n \log n)$.

Proof.

- The number of comparisons in merge sort is given by $T(n) = 2T(n/2) + n - 1$
- Hence, $T(n) = aT(n/b) + f(n)$ with $a = 2$, $b = 2$ and $f(n) = n - 1$.
- According to theorem 8, the solution to this recurrence relation is given by $T(n) = \sum_{i=0}^{\log n / \log b} a^i f(n/b^i) = \sum_{i=0}^{\log n} 2^i (n/2^i - 1)$.

Time Complexity of Merge Sort

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log n} 2^i (n/2^i - 1) \\
&= \sum_{i=0}^{\log n} n - 2^i \\
&= n(\log n + 1) - (2n - 1)^1 \\
&= n \log n - n + 1 \\
&= \mathcal{O}(n \log n)
\end{aligned}
$$

because $\sum_{i=0}^{\log n} 2^i = 2^0 + 2^1 + ... + 2^{\log n} = 1 + ... + n/2 + n = 2n - 1$

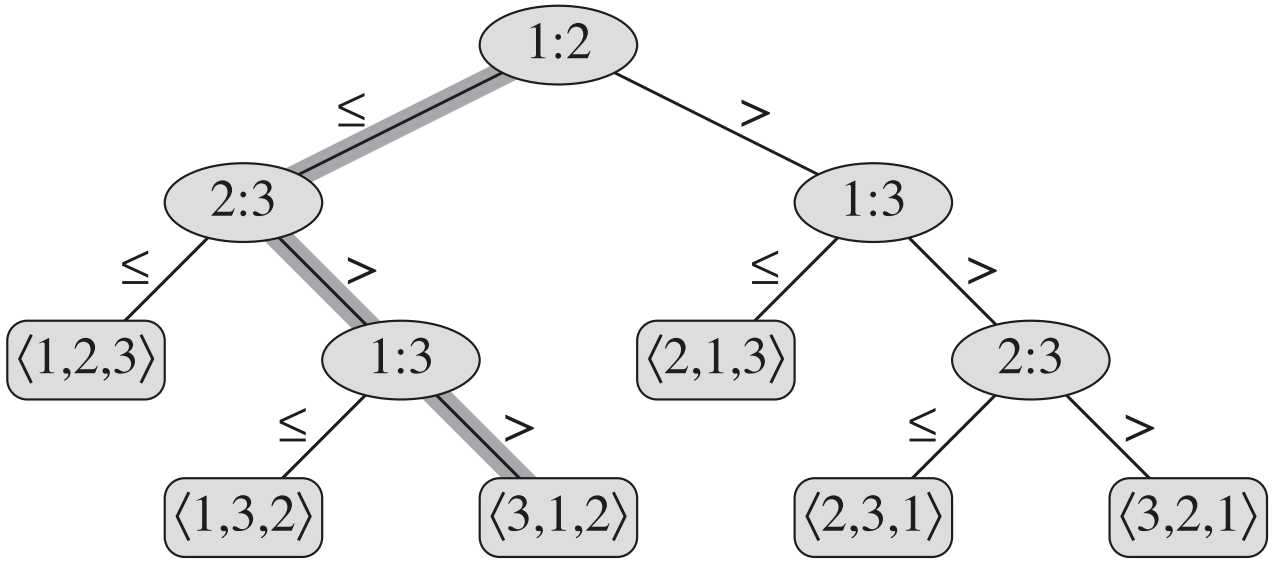
Can we do better than $\mathcal{O}(n \log n)$?

- It depends ...
- In *Comparison Sort* (i.e. determining the sorted order by comparing elements), we cannot do better.
- Why is this the lower bound on comparison sorting?
- This slide seems to imply that one can sort without comparing elements ?!?!?

Comparison Sort

- In order to sort a sequence of n elements, we will make a series of pair-wise comparisons to determine the appropriate sequence of elements.
- This sequence of comparisons can be seen as a binary tree (see next slide), in which each node is a comparison.
- The leafs of the nodes are the possible orderings of the n input elements.
- A sequence of n elements has $n!$ different permutations, so there must be at least $n!$ leafs in the tree.

Comparison Sort



A lower bound for comparison sort

Theorem 15

Any decision tree that sorts n elements has a height of $\Omega(n \log n)$.

Proof.

- Consider a tree of height h that sorts n elements. Since there are $n!$ permutations of n elements, each representing a distinct sorted order, the tree must have at least $n!$ leaves.
- Since a binary tree of height h has no more than 2^h leaves, we have that $n! \leq 2^h$ and hence:
- $h \geq \log(n!) \geq \log(n/e)^{n2} = n \log n - n \log e = n \log n - n + 1 = \Omega(n \log n)$

[2] Stirling's formula: $n! \approx (n/e)^n \sqrt{2\pi n}$

Beyond Comparison Sort

- If we now that we are sorting a vector containing only natural numbers ...
- and if we know that the largest number that can occur is K ...
- we can do better!

Counting Sort

- We keep a vector of length K that will hold the number of times each number occurs in the sequence.
- We need to traverse the entire vector and update the counts.
- From the counts, we compute the positions in the sorted vector of the new element.
- We traverse the vector once more and copy all elements to these computed positions.
- $\mathcal{O}(n + k)$!!!

Counting Sort

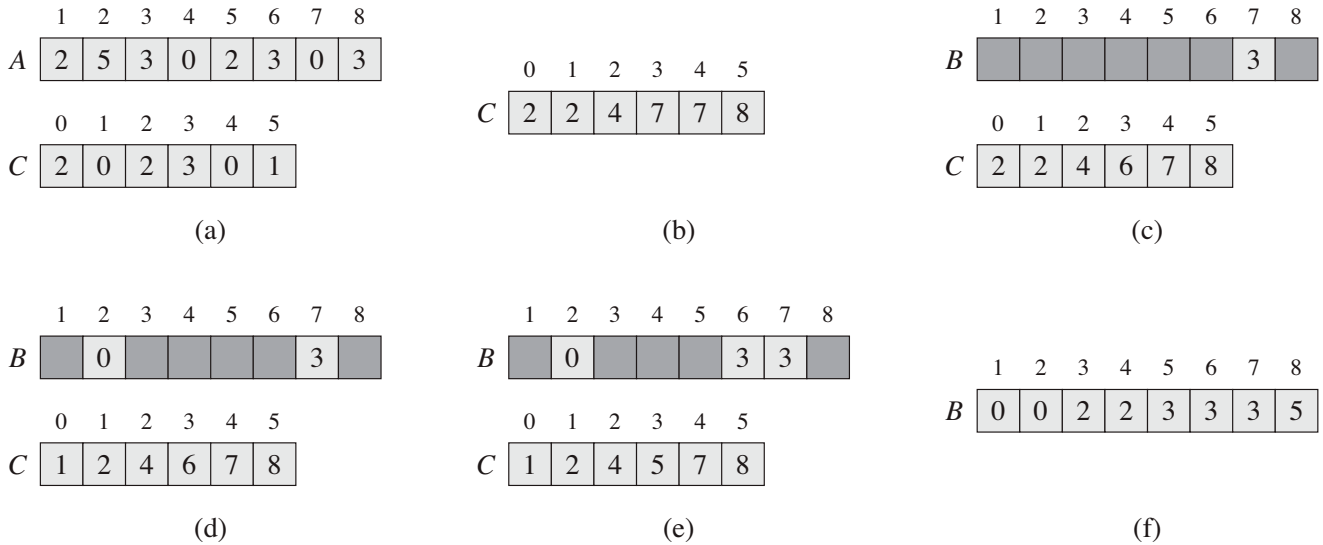


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

Counting Sort

COUNTING-SORT(A, B, k)

```
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```