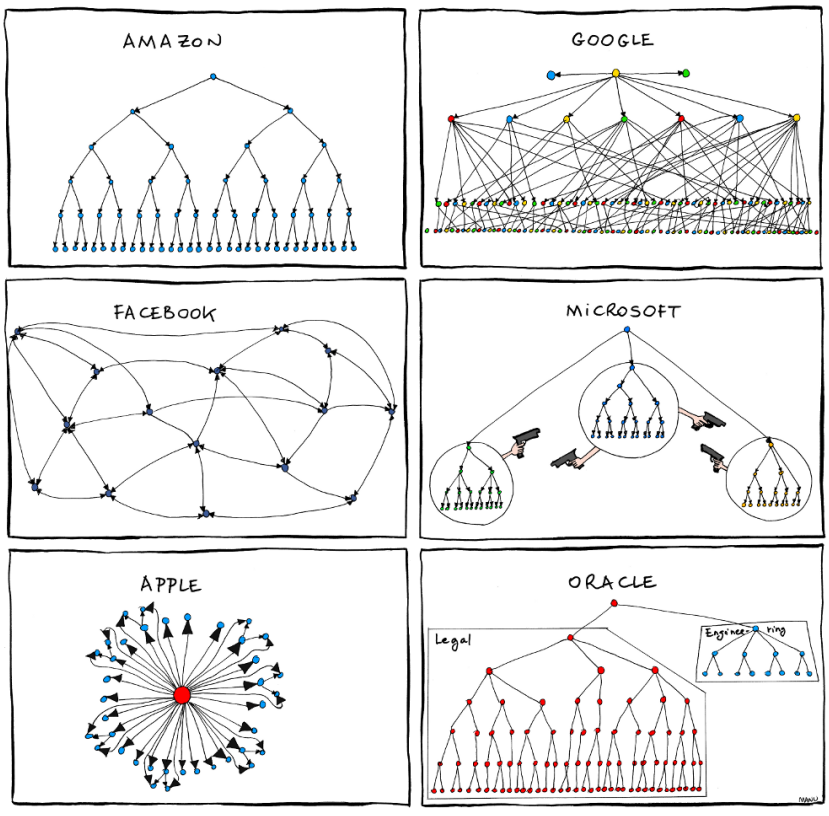
Vrije Universiteit Brussel

Excercises

Algorithms and Datastructures - Niet numerieke Algoritmen

Katka Kostkova, Lubos Omelina, Panagiotis Gonidakis,
Frederik Temmermans, Bart Jansen



Comic by Manu Cornet - www.bonkersworld.net

2022-2023

Contents

Chapter 1

Vector

A vector is one of the simplest data structures. In its simplest form it is nothing more than an array. However, in this chapter we will extend the functionality of the array with various useful methods, resulting in the vector.

Our vector will support the functionality of adding elements to it, accessing the elements, removing all elements, removing specific elements, checking whether the vector is empty, retrieving the number of elements in it, verifying whether a specific element is part of the vector, Ö and many more. So, clearly more than the simple array.

In order to implement all these methods, we will start from the array and add functionality to it.

1.1 The basic operations on a vector

1.1.1 Creating a vector

When we want to create a vector, we need to specify the size of the vector. This is necessary because the array we will build on top has a fixed size.

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

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

The vector class has a constructor that takes a single argument which is the capacity of the vector. In the constructor, the array is allocated and stored in the data variable. The variable count will store the number of elements in the vector. In the constructor, it is set to zero.

1.1.2 Accessing an element in a vector

The vector class supports the get method to retrieve an element at a given position. The implementation shown here assumes that the position is valid: it should not be negative and not be bigger or equal to the total number of elements in the list.

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

1.1.3 Adding an element to a vector

The addLast method will add a new element to the end of the vector. The implementation shown here assumes that there is an empty place available at the end of the vector and simply stores the new element at the right position, without any checks.

```
public void addLast(Object o)
{
   data[count] = o;
   count++;
}
```

1.1.4 Overwriting an element in a vector

```
public void set(int index, Object obj)
{
   data[index] = obj;
}
```

1.1.5 Getting the size of a vector, checking whether a vector is empty

```
public int size()
{
   return count;
}

public boolean isEmpty()
{
```

```
    return size() == 0;
}
```

If a method size is available which returns the total number of elements in the vector, then implementing the method isEmpty is easy: call the size method and check whether it returns 0 or not. Please note in the above code there is no if statement required to accomplish this.

1.1.6 Checking whether the vector contains a given element

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

1.2 Other operations on vectors

1.2.1 BinarySearch

When the elements in the vector are sorted, more efficient methods for finding an element in a vector could be built. In the binary search method, the vector is split in the middle into two parts and it is decided whether the element should be in the first or second halve of the vector. Then, the search continues in only one halve of the vector. Again, this subvector is split in the middle and it is decided in which of both halves of the subvector the element should be. This process is repeated over and over again on smaller subvectors, until the vector only has one element.

The above process is illustrated in figure 1.1. In that example, it is checked whether the element 15 occurs in the vector [1 5 9 14 15 16 22 37]. In the first step, the element in the middle of the vector (14 in this case), is compared to the search key (15 in this example). As the search key is bigger than the middle element, the element should be in the right halve of the vector. So, in a second step, the search continues in the right halve of the vector, which is [15 16 22 37]. The middle element of this smaller vector is 16. As the search key is smaller than the middle element, the element has to be in the left halve of the vector. Therefore, in a third step, the search continues in [15 16]. The middle element is 15, which is our key. So, we have found the element.

In the implementation given below, the elements of the vector are actually not copied into smaller subvectors. It is more efficient to use a variable start and a variable stop to mark the beginning and the end of the subvector in the bigger vector.

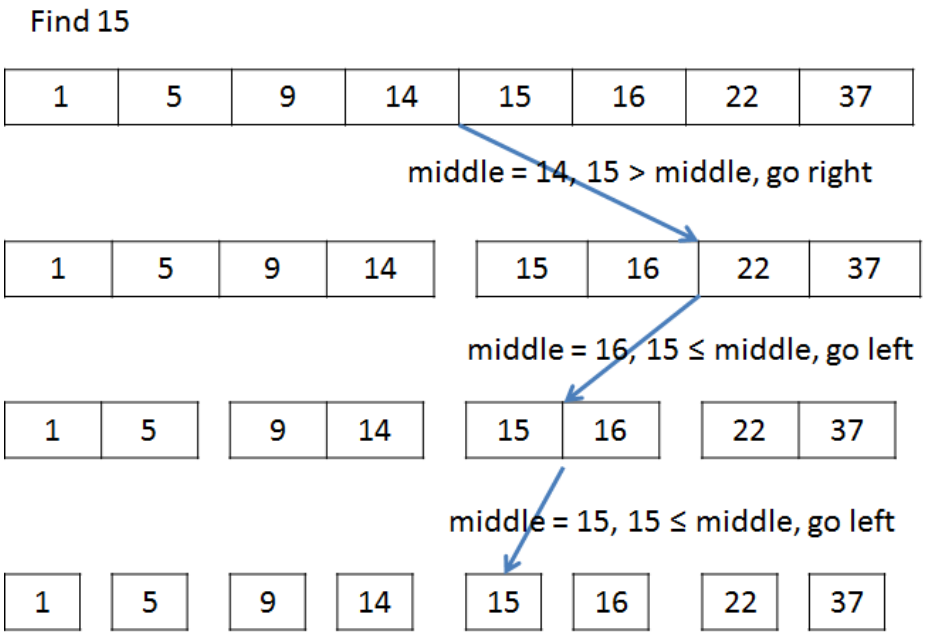


Figure 1.1: Binary Search in a vector.

```
public boolean binarySearch(Object 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;
}
```

1.3 Assignments

1. Create a vector, add the elements 1 to 100 to it, get the size of the vector, verify whether the vector contains the elements 6 and 101.

2. Add the methods getFirst and getLast. getFirst should retrieve the first element, it is at position 0. getLast should retrieve the last element. At what position is this one?

```
public Object getFirst()
{
   // add your code
}

public Object getLast()
{
   // add your code
}
```

3. Add the method toString, which converts the vector to a string representation. A vector with the elements 1, 2 and 3 should be converted to the string "[1 2 3]". The toString method is called automatically when you print to the console, e.g. "System.out.println(v);" to print a vector v.

4. Add the methods addFirst. addFirst should add the element at the beginning of the vector. In order to do so, all existing elements should be moved one place to the right. Make a drawing showing all the necessary steps. Implement this with a for loop that starts at the end of the vector and iterates to the beginning of the vector.

```
public void addFirst(Object item)
{
   // add your code
}
```

5. Add the methods removeFirst and removeLast. removeFirst should remove the first element, removeLast should remove the last element.

```
public void removeLast()
{
   // add your code
}
public void removeFirst()
{
   // add your code
}
```

6. Add the method reverse, which reverses the order of the elements of the vector. This method should not create a new vector, but really change the current vector.

```
public void reverse()
{
   // add your code
}
```

7. Implement the method "Vector repeat()", which returns a new vector that repeats the elements of the vector. Suppose you have a vector called v1 containing the elements [1 2 3]. The result of v1.repeat() should be a new vector containing the elements [1 1 2 2 3 3] while v1 remains unchanged. Implement the repeat method. Give the time complexity of your method and motivate it.

8. Implement the method "Vector interleave(Vector v2)", which interleaves two vectors. Suppose you have a vector called v1 containing the elements [1 2 3] and a second one called v2 containing the elements [4 5 6]. The result of v1.interleave(v2) should be a new vector containing the elements [1 4 2 5 3 6] while v1 and v2 remain unchanged. We do assume that both vectors have the same length! (Can you actually extend the method to deal with vectors having different length?) Give the time complexity of your method and motivate it.

9. One of the disadvantages of the vector class is that it has a fixed capacity: the vector can contain no more elements than defined by the capacity parameter, specified in the constructor. Add a method "private void extendCapacity()" to the vector class which doubles the capacity of our vector. This is done in 3 steps: (1) create a new array, called data2 which can contain "2*capacity" elements. (2) Copy the elements from data into data2. (3) set data to data2. Change the addLast method of the vector class such that it checks whether the vector is full. If the vector is not full, data can be added as it is done now. When the vector is full, you should first call "extendCapacity" before actually adding the element.

Chapter 2

Analyzing the time complexity of an algorithm

The time it takes for an algorithm to process the given input typically depends on the size of the input: for instance, it is expected that searching for an element in a vector of 1000 elements takes more time than searching in a vector of 10 elements, simply because there are more elements to compare the search key with. For instance, let us check the code to check whether a vector contains a given search key again.

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

The code consists of a single *for*-loop, which serves to iterate over the vector and access each an every element of the vector sequentially. For each element of the vector, it is checked whether it equals the key. So, if we assume that the time it takes to compare the key with an element does not depend on the element itself, the total runtime of the algorithm only depends on the number of iterations of the for loop. In order to estimate how many iterations this will be, we have to distinguish between several different cases:

- In the *best case*, the element we are searching for is the first element of the vector. In that case, there is only a single iteration, resulting in a single comparison to be done. Formally, we write this as $T(n) = 1$, meaning that the runtime T of this algorithm operating on an input of size n is 1.
- In the *worst case*, the element is not in the vector. In that case the for loop needs to go over the entire vector. So, if there are n elements in the vector, the algorithm needs to make n comparison or formally $T(n) = n$.

- In the average case, the element is found after traversing half of the vector. If the vector has n elements and it is equally probable that the key is at each of the different possible positions or not present, the average number of elements to check is $T(n) = (1 + 2 + ... + n)/n$ which equals to $T(n) = \frac{n(n+1)}{2n} = \frac{n+1}{2}$.

From the above, we learn that the time it takes to search for a given element in a vector in the average and worst case scenario is depending on the number of elements n there are in the vector. Moreover, there is in both cases a simple linear relationship. So, searching in a vector which is twice as large will also take twice as much time on average.

In computer science, the analysis of the time performance of algorithms is crucial, because for many algorithms, there is not a linear relationship between the input size and the average runtime. The relationship can be quadratic or even exponential. In such cases, the runtime becomes prohibitively long for rather small input sizes (see figures 2.1 and 2.2). So, from that perspective, it is much more important to be able to know whether this relationship is linear, quadratic, ... than to know the exact number of operations that need to be performed. So, the analysis of the performance of the runtime of algorithms is about comparing *orders of magnitude*, rather than about comparing exact numbers.

2.1 Orders of magnitude and the big-Oh notation

The big-Oh notation defined below exactly serves the purpose of formalizing the concept of *the order of magnitude*.

Definition 1 (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 2 (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 3 (big-Theta)**.** *$T(n)$ is $\Theta(F(n))$ if and only if $T(n)$ is $\mathcal{O}(F(n))$ and $T(n)$ is $\Theta(F(n))$.*

The first definition introduces the $\mathcal{O}$ symbol and defines an upper bound to the time complexity of an algorithms. For instance, if F is a linear function, the definition states that $T(N) = \mathcal{O}(n)$ if there is some c for which from n_0 on, $T(n) \leq cn$. According to this definition, the time complexity of finding an element in a vector is indeed $\mathcal{O}(n)$, because there indeed exists an n_0 and a c for which $T(n) = \frac{n+1}{2} \leq cn$. Please note that if the upper bound is $\mathcal{O}(n)$, then it is also $\mathcal{O}(n^2)$ etc. because the same inequality still holds. Obviously, in algorithm analysis we are interested in an accurate estimate of the upper bound, so we use $\mathcal{O}(n)$.

Similarly, the second definition gives a lower bound and the third definition, $T(n) = \Theta(n)$, combines the lower and upper bound to express that the growth

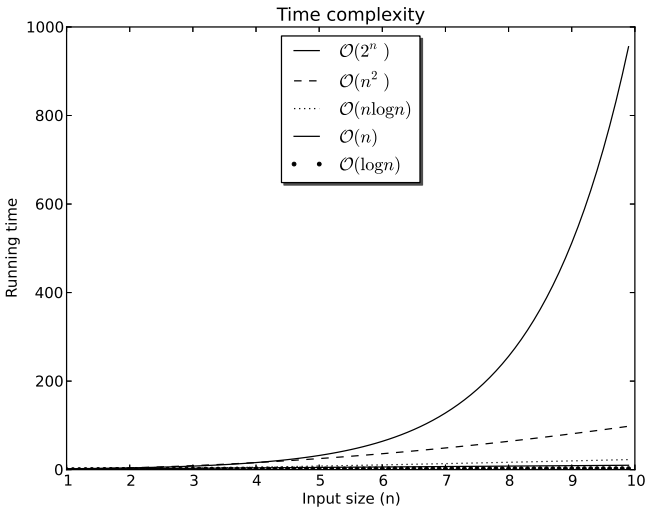


Figure 2.1: Time Complexities

rate of $T(n)$ equals $F(n)$. Although this is hence the most precise estimate of the three, it is not commonly used.

It is important to note that in reporting time complexities, some rules need to be taken into account:

- Do not put any constants in the notation, e.g. whether $T(n) = n$ or $T(n) = 2n$ or $T(n) = 0.5n$, they are all $\mathcal{O}(n)$. So, $\mathcal{O}(2n)$ or $\mathcal{O}(0.5n)$ are NOT valid, even though they give a true boundary. This is because in the analysis of algorithms, only the order of magnitude is relevant.

- Do not put any lower order terms in the notation, e.g. whether $T(n) = n^2$ or $T(n) = 2n^2+n$ or $T(n) = 0.5n^2+7$, they are all $\mathcal{O}(n^2)$. So, $\mathcal{O}(n^2+n)$ or $\mathcal{O}(0.5n^2+7)$ are NOT valid, even though they give a true boundary. This is because in the analysis of algorithms, only the order of magnitude is relevant. For sufficiently large values of n, higher order terms will always dominate lower order terms.

2.2 Some 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).
- $T(n) = 32^n + 7n^2 + 11 = \mathcal{O}(2^n)$ (exponential).
- $T(n) = 2\log(n) + 1 = \mathcal{O}(\log(n))$ (logarithmic*).

*We will use log for $\log_2$

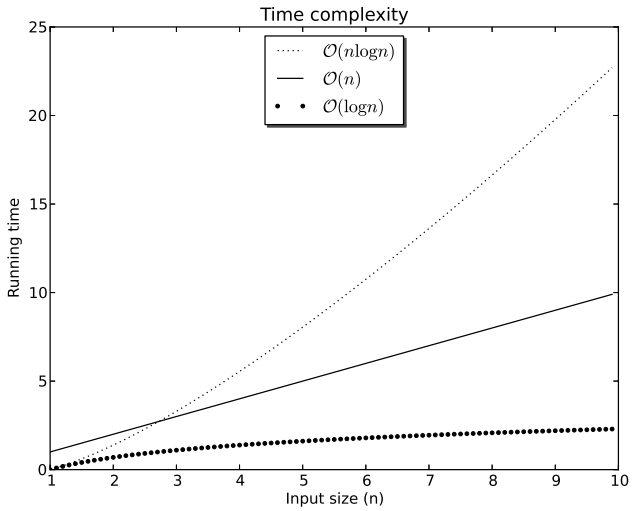


Figure 2.2: Time Complexities

Theorem 1 (The base doesn't matter)**.** *For any constant $B \geq 1$, $log_B(n) = \mathcal{O}(\log(n))$.*

Proof.

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

□

2.3 Analyzing Divide and Conquer algorithms

The basic idea of divide and conquer algorithms is to recursively subdivide a problem into smaller problems, until the problems are becoming that small that they become trivial to solve. So, divide and conquer is not an algorithm as such, but rather a paradigm for developing algorithms to tackle a variety of problems.

Assume the size of the total problem is n. The divide and conquer algorithm paradigm to solve a problem can be described as follows:

1. it 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.)

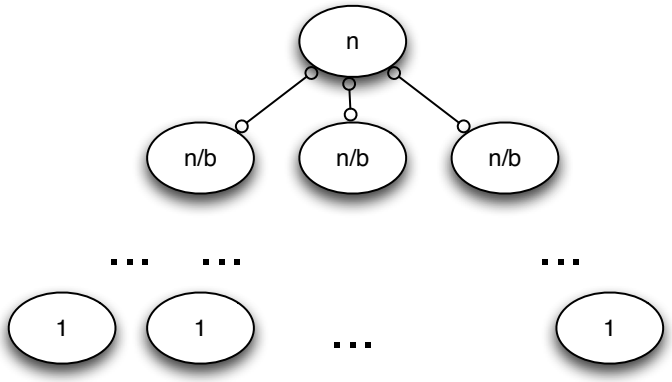


Figure 2.3: General divide and conquer algorithm outline

2. it solves the problem on each of these a pieces. (this is the recursion step)

3. it aggregates the solutions to the a smaller pieces into a single solution.

If we assume that the cost to aggregate the solutions to the a pieces of size n/b is $f(n)$, then the time complexity of this divide and conquer paradigm can be expressed by the following recurrence relation:

$$T(n) = aT(n/b) + f(n) \tag{2.1}$$

This equation expresses the common sense that the cost to solve the problem $T(n)$ is the cost to solve the simpler problem $T(n/b)$, times the number of simpler problems a, plus the cost of aggregating the different simpler problems $f(n)$.

Various mathematical methods exist for solving such recurrence relations. However, if we make two very reasonable assumptions, the recurrence relation can easily be solved:

1. We assume that we stop dividing the problem into smaller problems when the problem reaches size 1.

2. 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.

Theorem 2. *Under the above stated conditions, the solution of the recurrence equation*

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

is given by

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

Proof. (Not really a proof, rather an informal derivation)

At the start of the algorithm (we will refer to this as *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 up into smaller pieces each and so on. 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 $i = \log_b n = \log_2 n / \log_2 b$.

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:

$$T(n) = \sum_{i=0}^{\log n / \log b} a^i f(n/b^i) \tag{2.2}$$

□

This equation will allow us to compute the exact time complexity for several divide and conquer algorithms, by plugging in the appropriate values of a, b and $f(n)$.

2.3.1 The time complexity of Binary Search

Theorem 3. *The time complexity of the Binary Search algorithm on a sorted vector is* $\mathcal{O}(\log n)$.

Proof. When performing binary search on a sorted array, at each step of the algorithm, we need to decide whether to continue searching in the left or in the right sub problem. So, 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.

$$T(n) = T(n/2) + 1 \tag{2.3}$$

So, this recurrence relation is the same as equation 1, with $a = 1$, $b = 2$ and $f(n) = 1$. According to equation 2, the solution to this recurrence relation is given by

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

□

Chapter 3

Linked List

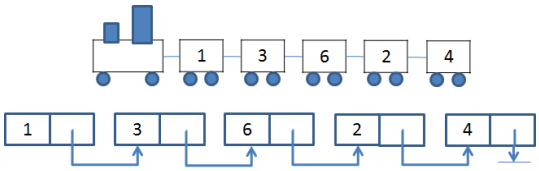


Figure 3.1: Top: a linked list represented as a train of numbers. Bottom: the same linked list as a sequence of listelements.

A linked list is a simple data structure which stores data in a linear manner, similar to the vector. However, while vectors have a fixed length, linked lists can grow and shrink dynamically. The consequence of this is that the implementation is slightly more complex. Dynamic length is obtained by storing data in a chain of elements. Each element is connected to a next element. By connecting more elements, the chain can grow. This allows for further flexibility: it is possible to remove one element from the middle of the chain (without copying data), or to add a new element in the middle of the chain.

An example illustrating this metaphor is given in figure 3. The top row shows the linked list as a train (or chain) of numbers. In order to implement this, we will slightly modify this metaphor (see bottom row of figure 3): we will build a chain of listelements, where each listelement connects to the next listelement. Note that in the implementation we will develop here, the first element in the chain is the first element of the list. There is no "special" element at the front of the chain. The last element does not connect to any next element, so this element will connect to a null pointer.

This linked list class is the basis for many other more complex datastructures, including double linked lists, circular lists, graphs, ...

3.1 Basic operations on a 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;
       }
    }

    private ListElement head;

    public LinkedList() {
       head = null;
    }

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

    public Object getFirst() {
       return head.first();
    }

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

```
    }

    public String toString() {
        String s = "(";
        ListElement d = head;
        while (d != null) {
            s += d.first().toString();
            s += " ";
            d = d.rest();
        }
        s += ")";
        return s;
    }

}
```

In order to implement the linked list, a train/chain of listelements is built. A listelement is represented by its own class and is simply a pair of two elements, in which the first element stores data and the second element contains a pointer to the next listelement. So, the listelement class only has very limited functionality. There is a constructor to make a pair of a given data element and a given next pointer and there are methods to get and set both the first and the second part of the pair.

Given the listelement class, the linkedlist class serves to store a pointer to the first listelement. This pointer will be stored in the variable "head". In the constructor of the list class, the listelement "head" is initialized to null, as the list does not contain any element yet.

3.1.1 Adding an element to the front of the list

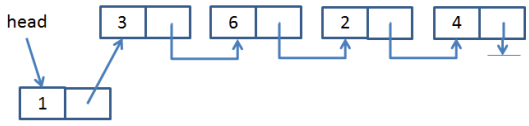


Figure 3.2: Adding data to the front of an existing linked list requires three steps: (1) making a new listelement for the new data, (2) making the rest of the listelement point to the head and (3) make the head point to the new listelement.

In order to add an element at the beginning of an already existing list, several operations need to take place and are illustrated in figure 3.1.1: (1) making a new listelement for the new data, (2) making the rest of the listelement point to the head and (3) make the head point to the new listelement.

```
public void addFirst(Object o)
{
```

```
    ListElement newElement = new ListElement(o); // step 1
    newElement.setRest(head); // step 2
    head = newElement; // step 3
}
```

These three operations can however be performed in a single line of JAVA code:

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

3.1.2 Accessing an element in the list

In order to access the n-th element in the list, we cannot do anything else than starting from the head and traveling n times to the next element in the list:

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

3.2 Assignments

1. Make a new empty linked list and add the elements 1,2,3 and 4. Print your list to verify everything is correct.
2. Many algorithms need to know the number of elements in the list. Add a size() method.
3. Add a method set(int n, Object o) which replaces the nth element in the list.
4. Add a method getLast() which returns the last element in the list.
5. Add a method addLast() which adds an element to the end of the linked list.

6. Add a method which searches for a given element in the linked list.

7. So far, your linked list class should support exactly the same set of operations as the vector class does. If this is not the case, implement the missing operators on both classes. As soon as both datastructures are supporting exactly the same operations, it is possible to run all examples using linkedlists with vectors and the other way around. Verify that all results of all methods are the same on both datastructures.

8. The "fropple" method on linked lists operates on a linked list and swaps every two consecutive elements. For example, the list [1 2 3 4 5 6] is froppled into [2 1 4 3 6 5]. This fropple operation can be performed in $\mathcal{O}(N)$, with a single iteration over the linked list. Make a detailed drawing of how the fropple operator changes the pointers in a linked list. As the order of changing pointers is crucial, please number them. Implement the fropple method on the linked list class. Your code should not create a new list; the current list should be modified! Traverse the list only once!

9. Implement the method "void append(LinkedList list2)", which concatenates two lists. Suppose you have a linkedlist called list1 containing the elements (1 2 3) and a second one called list2 containing the elements (4 5 6). The result of list1.append(list2) should be that list1 contains the elements (1 2 3 4 5 6) while list2 is remained unchanged. Draw the linked lists list1 and list2. In a second drawing, draw the result of list1.append(list2). One possibility to implement this, is to traverse list2 element by element and to add each element of list2 to the end of list1, using the addLast method of the linked list. Provide an implementation of the append method which is taking this approach.

10. **Make a table comparing the time complexity of all the operations supported on both linked lists and vectors: getFirst, getLast, addFirst, addLast, removeFirst, removeLast, get, set, find.**

Chapter 4

Double Linked List

The Vectors and the Linked Lists we've introduced so far, fail to perform addFirst, addLast, removeFirst and removeLast all in $\mathcal{O}(1)$. The optimization of a specific operation, always resulting in a penalty at some other operation. In this chapter, we will develop Double Linked Lists, because they allow for the four above mentioned operations in $\mathcal{O}(1)$.

For the Linked List, addFirst and removeFirst can be done in $\mathcal{O}(1)$. Both addLast and removeLast however, require the traversal of the entire list, resulting in $\mathcal{O}(n)$. One could think of keeping an extra pointer to the last element. That allows to implement addLast also in $\mathcal{O}(1)$. However, in order to remove the last element in $\mathcal{O}(1)$, also a pointer to the second last element is required. Obviously, keeping a pointer to both the last and the second last element does not help, because when the last element is removed, the pointer to the second last element needs to be updated (taking again $\mathcal{O}(n)$).

So, what is required is that each element has a pointer to the previous element in the list, additional to the pointer to the next element that is already present in the regular Linked List (see figure 4.1).

4.1 Implementation

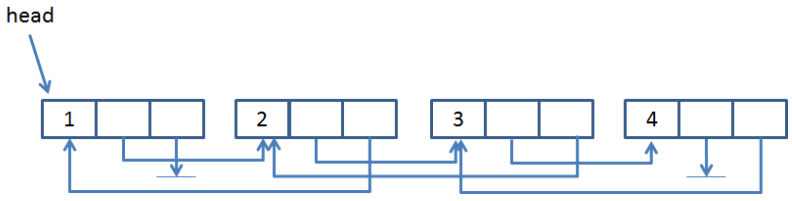


Figure 4.1: A double linked list: each element in the list has a link to the previous and to the next element.

```
public class DoubleLinkedList
```

```
{
        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 =
                        if(previousElement != null) previousElement.nextEleme
                }

                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)
                {
                        nextElement = value;
                }

                public void setPrevious(DoubleLinkedListElement value)
                {
                        previousElement = value;
                }
        }

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

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

        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 String toString()
        {
                String s = "(";
                DoubleLinkedListElement d = head;
                while(d != null)
                {
                        s += d.value().toString();
                        s += " ";
                        d = d.next();
                }
                s += ")";
                return s;

        }
}
```

4.2 Exercises

1. Make a new DoubleLinkedList, add the elements 1 to 4, print the list. This should print (1 2 3 4).

2. Add a new method called void printReverse() which prints the list in reversed order, it should print (4 3 2 1). Do not reverse the list itself, but traverse the list from tail to head in order to print the elements. As the double linked list implementation is perfectly symmetric, the print method can easily be changed to accomplish this.

3. Consider the removeFirst method given below. Make a drawing showing how this method is removing the first element from the linked list containing the elements (1 2 3). Make a new drawing for the removeFirst method operating on the double linked list containing a single element (1).

```
public void removeFirst()
{
        head = head.next();
        if(head == null) tail=null;
        else head.setPrevious(null);
        count--;
}
```

4. Implement the removeLast() method. Exploit similarity, so remove the first element from the linked list starting at the tail, rather than removing the last element of the linked list starting at the head. This can be accomplished easily by replacing the head by the tail in the removeFirst. What else must be changed?

5. Only bother about this if you can do all the rest without problems. Implement a reverse method on the double linked list. In order to have this done in an efficient manner i.e. $\mathcal{O}(n)$, you should traverse the entire double linked list only once and you should not create any new DoubleLinkedListElements or a new DoubleLinkedList. So, the entire reverse should be performed by only manipulating pointers. It might be useful to draw a linked list containing for instance (1 2 3 4) and underneath that, the reverse of this linked list, but with the same elements underneath each other.

Chapter 5

Stacks

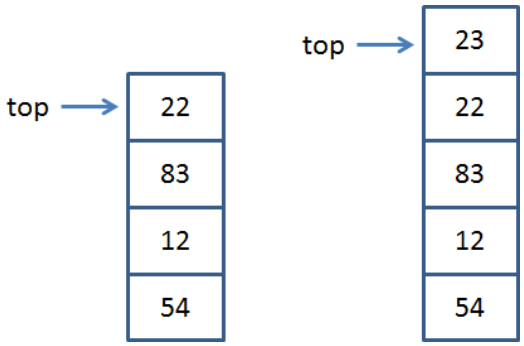


Figure 5.1: Pushing an element on the stack: the element is stored at the top of the stack.

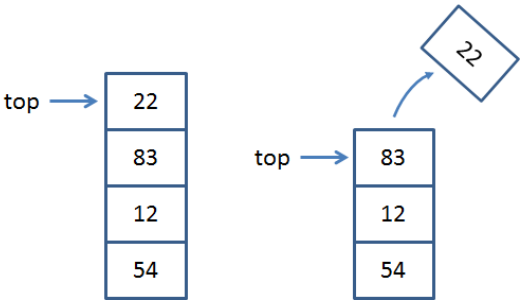


Figure 5.2: Popping an element from a stack, removes the topmost element and returns it.

A stack is a simple data structure that supports only a limited set of operations. A stack represent a pile of object and only the top of the pile can be manipulated: it is possible to add an element on top of the stack (this is called the "push" operation) and it is possible to remove the topmost element from the stack (the "pop" operation). The push is illustrated in figure 5.1. In that example, the element 23 is pushed on the stack. In figure 5.2, the topmost

element is popped from the stack and returned. A stack also supports a third method "top" which simply returns the topmost element from the stack, but does not remove it from the stack.

A stack can be implemented in a straightforward manner using the Vector class.

```
public class Stack
{
        private Vector data;

        public Stack()
        {
                // todo
        }

        public void push(Object o)
        {
                // todo
        }

        public Object pop()
        {
                // todo
        }

        public Object top()
        {
                // todo
        }

        public int size()
        {
                // todo
        }

        public boolean empty()
        {
                // todo
        }
 }
```

In the push operation, the element needs to be added at the end of the vector. In the pop operation, the element needs to be removed from the end of the vector.

5.1 Exercises

1. Complete the vector based stack implementation.

2. What is the time complexity of the push and pop operation?

3. The major disadvantage of this approach is that the stack can not grow beyond a predefined limit. Adapt the implementation of the stack such that it uses a Linked List AND remains as efficient as the vector based stack.

Chapter 6

Queue

A queue is very similar to a stack and also only supports push and pop, but with a slightly different meaning: A queue implements the functionality of queues you are familiar with: new elements are stored at the back, while elements are removed from the front of the queue. So, in a vector based implementation, the push operation adds new elements to the back of the vector, while new elements are popped from the front of the vector:

```
public class Queue
{
        private Vector data;

        public Queue()
        {
                // todo
        }

        public void push(Object o)
        {
                // todo
        }

        public Object pop()
        {
                // todo
        }

        public Object top()
        {
                // todo
        }

        public int size()
        {
                // todo
        }
```

```
        public boolean empty()
        {
                // todo
        }
}
```

6.1 Exercises

1. Complete the vector based queue implementation.
2. What is the time complexity of the push and pop operation?
3. The major disadvantage of this approach is that the queue can not grow beyond a predefined limit. Adapt the implementation of the queue such that it uses a Linked List. What is the time complexity now?
4. **Make a table comparing the time complexity of the Vector based and the Linked List based Stack and Queue implementations of the push and pop method.**

Chapter 7

Priority Queue

Similar to the Stack and Queue data structure, the priority queue is a simple structure supporting the push and pop methods. However, 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. Just as with the Stack and the Queue, the Priority Queue implementation can be based on the Vector or on the Linked List. Moreover, both for the Vector as for the Linked List, there are two different approaches:

- The first approach consist of maintaining the data structure sorted on the priority. This means that the push operation is more complex, because it now needs to insert the element at the right position in an already sorted data structure, to ensure that the structure remains sorted. As a consequence, the element with the highest priority will always be at the end or the beginning of the datastructure, such that the pop operation remains very simple. Under this scheme, the push will have a time complexity of $\mathcal{O}(n)$ while the pop will have a time complexity of $\mathcal{O}(1)$.

- 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 $\mathcal{O}(n)$.

Note that both options result in asymmetrical time complexity. One operation has a time complexity of $\mathcal{O}(1)$ while the other operation has a time complexity of $\mathcal{O}(n)$. Using simple underlying datastructures as the Vector and the Linked List, solutions that are both balanced and efficient are not possible. We will come back to this problem and develop a more efficient and balanced solution later. In this chapter, we will first implement a Linked List based implementation, which keeps the data sorted in the push method.

7.1 Sorted Linked Lists

In order to have a Priority Queue that stores the data sorted, we will first develop a sorted Linked List. This is the existing Linked List with a new operation added: "addSorted(object value)". If the Linked List is sorted, i.e. all elements are sorted from smallest to largest, then the addSorted method will add a new element at the right spot in the Linked List, to ensure that the Linked List remains sorted.

The idea of the addSorted method is to traverse the linked list from left to right until an element is found which is bigger than the new element. At that point, we now that the new data should come right before the current element of the linked list. As it is easier to add an element right after a given element, the Linked List will be traversed until the next pointer of the current element is bigger than the new element. If the linked list is empty, or if the Linked List has only one element, this approach might not be possible. Therefore, we have to identify all these cases and handle them separately:

```
// 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++;
}
```

Although the algorithmic approach of the above code is correct, it is not valid JAVA code. This is because the $<$ and $>$ operators are working on objects of the type "Object" in the above code. That is not allowed in JAVA. This is easily understood: The Linked List can contain all kinds of data, including instances of classes. So, let's say we store instances of the object "Car" in the Linked List. How can JAVA know how to compare and order cars? What is

needed is that we tell JAVA that we will only add elements to the LinkedList by means of the addSorted method for which we can guarantee that they can be compared. Technically, this is expressed by passing objects to the addSorted method that implement the Comparable interface. These type of objects should implement a CompareTo method, which implements the comparison.

So, rather then having "void addSorted(Object data)", we should put "void addSorted(Comparable data)". But that is only possible when the elements stored in the Linked List are implementing the Comparable interface. So, a cascade of changes is starting here. **In ALL code developed so far, it is required to replace ALL occurrences of "Object" by "Comparable" in order to be able to keep a consistent interface and usage of all data structures.**

Once that is done, the code below is syntactically correct JAVA code for the addSorted method.

```
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++;
}
```

7.2 A Priority Queue based on a Sorted Linked List

In order to use the Sorted Linked List as the basis of the Priority Queue, we need to make a small helper class Pair, which groups both the data and the priority of the elements to be added, as a Linked List is capable of storing a single value only per element. The pair just stores both values and has a compareTo method implemented that compares the second value. So, the second element has to be the priority.

```
import java.util.Comparator;

public class PriorityQueue
{
   private class PriorityPair implements Comparable {
      private Object element;
      private 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()
   {
      // add your code here
   }

   public Object top()
   {
      // add your code here
   }
}
```

7.3 Exercises

1. Complete the implementation of the Priority Queue class based on the Sorted Linked List.

2. We have now implemented the priority queue as a sorted linked list. We could also implement the priority queue as an unsorted linked list. Make a

diagram to show how this implementation might look like and implement it.

3. Suppose you need to store elements into a priority queue. Consider the special case where there are only two possible priorities, say A and B. If you know for sure that the priority will either be A or B, there are implementations of the priority queue with the push and pop methods in $\mathcal{O}(1)$. Provide such an implementation of a priority queue class, including a constructor, a push method and a pop method.

Chapter 8

Circular Vectors

A Circular Vector is an adjustment to the regular Vector that avoids the shifting of all elements in the addFirst and RemoveFirst methods, such that both operations in the Circular Vector have a time complexity of $\mathcal{O}(1)$ rather than $\mathcal{O}(n)$ as with the regular Vector. In order to obtain this performance gain, a more complex implementation is required.

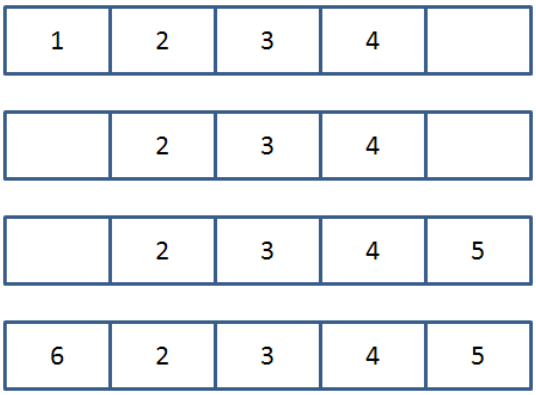


Figure 8.1: The internal state of a circular vector after the operations removeFirst(), addLast(5) and addLast(6).

The idea in the Circular Vector is that the first element of the Circular Vector (i.e. the element getFirst will return) will not always be stored at position 0 in the underlying Vector. Rather, an index "start" is stored and represents the position of the first element. An example is given in figure 8.1: the topmost row shows a Circular Vector holding the elements 1,2,3,4. In that case the value of the "start" variable is 0, as the Vector is filled from position 0 on. The second line shows the result of a removeFirst() operation: the contents of the Vector is not shifted to the left, rather the value of "start" is increased to 1. So, at that point a call to getFirst() will return 2 as the value at the "start" of the Circular Vector is now 2. The third row of the same figure shows the internal state after adding the element 5 at the end by calling addLast(5). The last row in the same figure shows the internal state after adding the element 6 at the end by calling addLast(6). Although it looked as if the Vector was completely full

after adding 5 at the end, this is not true: there is an empty place in the vector all at the beginning, before the position "start". The element is added there. So, when the end is reached it starts adding at the beginning, it is circular.

8.1 Adding an element to the front

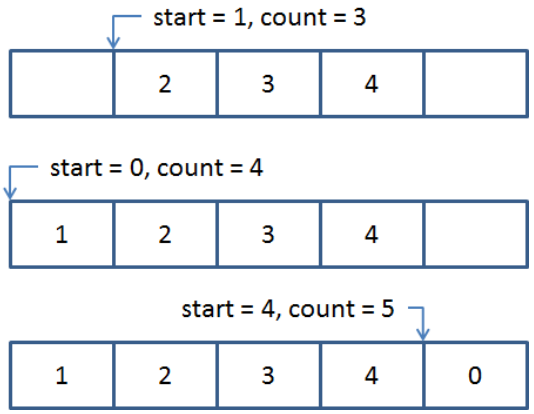


Figure 8.2: The internal state of a circular vector after the operations addFirst(1) and addFirst(0).

As already outlined above, the addFirst on the Circular Vector is different from the addFirst on the Vector, as no elements will be shifted. As shown in figure 8.2, the new element is stored right before the actual start of the contents of the vector, i.e. at position "start - 1". However, "start - 1" can become smaller than zero, as in the third row of figure 8.2. In that case, the element should be added at the last position, i.e. "capacity - 1". So, in order to cover both cases with a single line of code, we set $start = (start + capacity - 1)\%capacity$. The new element is then stored at the updated start and of course, we need to increment the count variable. This formula is not correct for the case of the empty circular vector, so that needs to be addressed in a separate case.

8.2 Adding an element to the end

The addLast is similar to the addFirst method. In this case, elements are stored at the end, which is at position "start+count". In order to avoid writing after the end of the vector, we store the element at $(start + count)\%capacity$.

8.3 Removing an element at the beginning

If we simply increment the start by one (modulo count off course) and decrement the count by one, we have shifted the start position one element and the previously first element of the Circular Vector will be overwritten.

8.4 Removing an element at the end

This fourth operation is similar to all other operations, but even simpler. The only thing we need to do is decrement count by one.

8.5 Implementation

```
public class CircularVector
{
   private Vector data;
   private int first;
   private int count;

   public CircularVector()
   {
      count = 0;
      first = 0;
      data = new Vector(5);
   }

   public int size()
   {
      return count;
   }

   public void AddFirst(Object element)
   {
      // add your code here
   }

   public void AddLast(Object element)
   {
      // add your code here
   }

   public Object GetFirst()
   {
      // add your code
   }

   public Object GetLast()
   {
      // add your code
   }

   public void RemoveFirst()
   {
      if(count > 0)
      {
         first = (first+1)%data.capacity();
         count--;
```

```
        }
    }

    public void RemoveLast()
    {
        // add your code
    }

    public String toString()
    {
     String s = "[";
        for(int i=0;i<count;i++)
        {
            int index = (first + i) % data.capacity();
            s += data.get(index).toString();
            s += " ";
        }
     s += "]";
     return s;
    }
}
```

8.6 Exercises

1. Complete the above sketched implementation of the Circular Vector.

2. Revisit the implementation of a queue using a vector. If you simply do a search/replace operation where you replace Vector by CircularVector, you would expect that your queue is still working perfectly, as we specified the same interface for the Vector and the CircularVector. Verify this.

3. Although you've just experienced that there is no operational difference between a Queue implemented as a Vector or as a CircularVector, there are strong arguments for using a circular vector. Why?

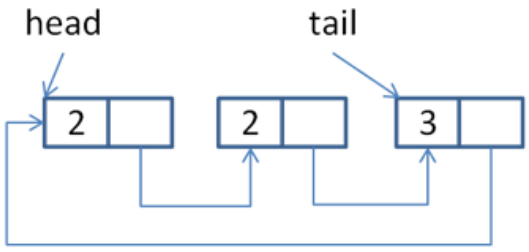


Figure 8.3: A circular linked list.

4. A circular list is a list in which the next pointer of the last element points to the first element of the list. The easiest way to implement this is to explicitly keep track of two pointers: head and tail. When adding or

removing elements, these have to be updated (see Figure 8.3). Draw in a few steps how the above shown list is modified when a new element 555 is added at the beginning of the list. Give an implementation of this addFirst(Object element) method, assuming there is a head and tail variable available in the CircularList class. Hint: this is not difficult and can be accomplished with very few code, but take care that you also can handle the addition of an element to the empty list. What is the time complexity of your addFirst method? What is the time complexity of the addLast method on circular lists? No need to implement this method.

Chapter 9

Dictionary

Key	Value
Jan	0032 2 629 18 00
Marc	0032 2 629 18 01
John	0032 5 444 56 89
Bill	0032 3 645 18 11
Simon	0026 6 345 67 60

Figure 9.1: A Dictionary is a datastructure that associates keys to values. It supports the easy lookup of the value of a given key. Think of this as a phone book: it allows to look up the phone number for a specified name.

A Dictionary is a simple datastructure that stores associations between keys and values (just as a dictionary for English, or as in a telephone book, see figure 9.1). Keys are unique identifiers for the values: so given a specific key, the dictionary serves to look up the associated value. In it's simplest form, the dictionary only support two functionalities: a method to add a new key and value and a method to lookup the value associated to a given key. A simple approach to implement the Dictionary is shown in figure 9.2. It stores an association between a given key and a given value in a Pair class (just grouping both, as in the priority queue). All pairs are stored in a vector. As a result, the add method simply consists of adding a new pair at the end the vector ($\mathcal{O}(1)$). The get method traverses the entire vector and compares the keys of all pairs in the vector to the query key, until the right key is found or the end of the vector is reached ($\mathcal{O}(n)$).

9.1 Exercises

1. Complete the implementation of the Dictionary class.

   Firstly, complete the implementation of the DictionaryPair class which is private and nested in the Dictionary class. It has two variables of type Object called key and item to store the key and the value. Your DictionaryPair class should have a constructor that can make a pair given a key

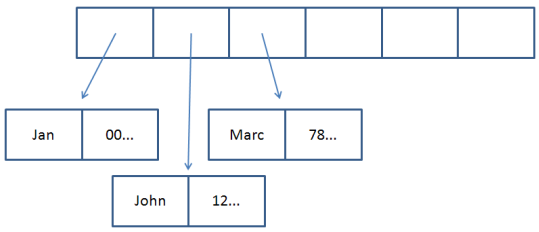


Figure 9.2: A simple approach to represent a Dictionary as a Vector of (key,value) Pairs.

and a value and which has methods for getKey(), getValue(), setKey(), setValue(). Use the Pair class from session two to store the key and the value.

Afterwards, we will continue implementing the dictionary class, which is storing all key,value pairs as instances of the "Pair" class in a Vector. Write a constructor for making an empty Dictionary. Add a method called "add(key,value)" which will create a new Pair for the given key and value and store this pair in the Vector using the addLast method of your Vector. Add a method findPosition(key) to your Dictionary which will check whether a pair with the given key is in the Dictionary. If the given key is in the Dictionary, it will return the index of the given Pair. If the key is not in the Dictionary, it will return -1. Given this findPosition method, you can implement a "find(key)" method. If the given key is in the Dictionary, this method will return the corresponding value. If the key is not in the Dictionary, return null.

```
public class Dictionary {

   private class DictionaryPair implements Comparable {
      private Object key;
      private Object value;

      public DictionaryPair(Object key, Object value) {
      }

      public Object getKey() {
      }

      public void setKey(Object key) {
      }

      public Object getValue() {
      }

      public void setValue(Object value) {
      }
```

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

        }

        private Vector data;

        public Dictionary() {
        }

        public void add(Object key, Object value) {
        }

        public int findPosition(Object key) {
        }

        public Object find(Object key) {
        }

        public void removeKey(Object key){
        }

        public int size(){
        }
    }
```

2. Modify the add method, such that it first verifies whether the given key is already in the Dictionary using the findPosition method. If the key is already in the Dictionary, then the add method should overwrite the existing value in stead of adding a new pair with the key and value.

Chapter 10

Binary Search Tree

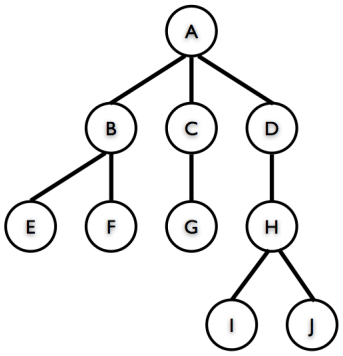


Figure 10.1: Example of a tree.

A tree is a hierarchical datastructure storing data with a parent - child relation. The data is stored in **nodes** while the relations between the nodes are expressed by **edges**. As every node has one parent (except for the top most node which is called the **root**) and as every node can have many children, this structure resembles and up-side-down tree. See figure 10.1 for a graphical representation of a tree. In the drawing an arrowed line from a node A to a node B is expressing the fact that A is the parent of B and hence that B is a child of A.

Nodes that do not have any children are called **leaf nodes**.

When starting from the root node and by going down in the tree until a destination node is reached, one is forming a **path** in the tree. For example, in Figure 10.1 the path between the root node A and the node I consists of the nodes A,D,H,I. The **depth** of a node in the tree is the length of the path between the root and the node. The depth of a tree is the maximum of the depths of the nodes of the tree.

The tree as defined here is a hierarchical structure, i.e. every node in the tree can act as the root node of a smaller tree consisting of that node and all the children. This smaller tree is called a **subtree**.

In general, a tree can have a variable number of children in each node. A

tree with at most two children per node is called a **binary tree**, a tree with at most three children per node is called a **ternary tree**.

10.1 Binary Search Tree

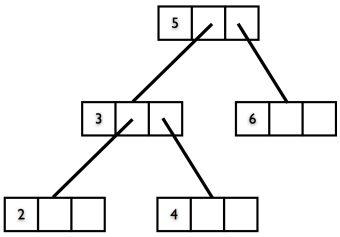


Figure 10.2: Example of a Binary Search Tree.

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. Figure 10.2 shows an example of a binary search tree. In this type of tree, there is some complexity in the operation of adding new elements to the three as we carefully need to check where exactly to put the element in the tree in order not to violate these properties. On the other hand, we will see that verifying whether a particular node is present in the three can be done efficiently.

10.1.1 Basic operations

Nodes

```
// the class for implementing a node in the tree.
// it's private and nested in the Tree class
// contains a value, a pointer to the left child and
// a pointer to the right child

private class TreeNode implements Comparable
{
    private Comparable value;
    private TreeNode leftNode;
    private TreeNode rightNode;
    public TreeNode(Comparable v)
    {
        value = v;
        leftNode = null;
        rightNode = null;
    }

    public TreeNode(Comparable v, TreeNode left,
```

```
                    TreeNode right)
    {
        value = v;
        leftNode = left;
        rightNode = right;
    }

    public TreeNode getLeftTree()
    {
        return leftNode;
    }

    public TreeNode getRightTree()
    {
        return rightNode;
    }

    public Comparable getValue()
    {
        return value;
    }

    public int compareTo(Object o)
    {
        return value.compareTo(((TreeNode)o).value);
    }
}
```

As mentioned above, a tree will be a recursive structure in which each node can be considered as a root of a smaller subtree. Therefore, it is the easiest to first make a small helper class to represent a node and only then to define a tree class which is using this TreeNode class. This TreeNode class stores the value of the node, a pointer to the root of the left subtree (this is again a TreeNode) and a pointer to the root of the right subtree (also again a TreeNode).

The tree class

The tree class itself stores a pointer to the root node and has some methods to add an element to the three, to find an element in the tree and provides some functionality to traverse the tree, i.e. to visit each element in the tree and perform some predefined action on each element.

```
import java.util.Comparator;

public class Tree
{
    public class TreeNode implements Comparable
    {
        // here goes the entire TreeNode class.
```

```
    }

    // the root of our tree
    private TreeNode root;

    public Tree()
    {
        root = null;
    }

    public void insert(Comparable element)
    {
        // here goes the code to insert an element
        // in the binary search tree
    }

    public boolean find(Comparable element)
    {
        // here goes the code the verify whether
        // an element is present in the tree
    }
}
```

Searching for an element in a binary search tree

Because the binary search tree has the property that for each node K all nodes at the left subtree of K are smaller than K and all nodes in the right subtree of K are larger than K, the search for an element in a tree can be easily defined recursively. There are four cases that can exist when searching for an element in a binary search tree:

1. The tree is empty, so the element is not present in the tree.
2. The root of the tree equals to the element, so we have found the element in the tree.
3. 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.
4. 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.

These four cases are one by one translated into the following code for searching for a given element in the 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);
}
```

Inserting an element in a binary search tree

The insert operation is a little bit more involved, as we cannot insert the element wherever we want in the tree in order not to violate the binary search tree conditions. Similar to the find method, we also define the insert method recursively and we discriminate between the following five conditions:

1. The tree is empty, so in order to add the new element to the tree, we need to make root point to it.
2. If the root equals the element we want to insert, the element is already in the tree, so there is nothing to do.
3. If the item to add in the tree is smaller than the root node, it should be inserted somewhere in the left subtree. So, we call the insert method recursively on the left subtree.
4. If the item to add in the tree is larger than the root node, it should be inserted somewhere in the right subtree. So, we call the insert method recursively on the right subtree.
5. In this way it can however happen that as a result of multiple recursive calls of the insert to insert the element in the right or left subtree, we call the insert method on an empty subtree. In that case we should insert the element as a child under the parent of this empty subtree. This requires us to know the parent of any node during the insert method. Because in general it is not possible from the tree datastructure to know the parent of a node, we added an extra argument to the insert method to keep track of the parent of the current node.

```
public void insert(Comparable element)
{
```

```
    insertAtNode(element,root,null);
}

    // we traverse the tree.
    // Current holds the pointer to the TreeNode
    // we are currently checking
    // Parent holds the pointer to the parent
    // of the current TreeNode
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)
       {
           // do we add it to the left?
           if(element.compareTo(parent.value) < 0)
           {
               parent.leftNode = newNode;
           }
           // or do we add it to the right?
           else
           {
               parent.rightNode = newNode;
           }
        }
        // the current node is empty and it has no parent,
        // we actually have an empty tree
        else root = newNode;
    }
    else if(element.compareTo(current.value) == 0)
    {
        // if the element is already in the tree,
        // what do we do?
        // at this point, we don't 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(),
                      current);
}
```

10.2 Exercises

1. Make a new tree and add the elements 1,2,3,4,5,6 in that order. Draw the tree after the inserts of all these numbers. You should be able to draw the final tree without drawing all intermediate steps BEFORE running the code.

```
public String toString()
{
    String s = "";
    Stack t = new Stack();
    t.push(root);
    while (!t.empty())
    {
        TreeNode n = (TreeNode)t.pop();
        s += n.value.toString();
        if(n.getRightTree() != null)
            t.push(n.getRightTree());
        if(n.getLeftTree() != null)
            t.push(n.getLeftTree());
        s += "\n";
    }
    return s;
}
```

2. Consider the toString() method as defined above. What is the toString() method supposed to print for this tree? Verify. Keep in mind that a toString() method needs to be implemented in the TreeNode class as well.

3. The toString() method is using a stack to store the list of nodes we will print next. Change the stack to a queue. What should be printed now? Verify.

4. The toString() method you now have, is actually an example of a more global method to traverse the tree. Consider the following code:

```
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());
```

```
        }
    }
```

This action is traversing the tree and calling the method "run" of the "TreeAction" class on each element. The TreeAction class is an abstract class with only the run method:

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

Make a subclass of the TreeAction class which you call "TreePrinter". Implement the "run" method in this class such that it will print the given node. Now use the traverse method together with your TreePrinter class to print all elements in the tree.

5. This method for traversing your tree is generating quite some overhead: for every type of traverse you want to do, you need to make a new subclass of "TreeAction" in a new file and pass this class to the traverse method. An alternative is to make an anonymous class: a class without a name. Use this pattern to implement the toString() method in your Tree.

```
public String toString()
{
    String s = " ";
    traverse(new TreeAction()
    {
        public void run(TreeNode n)
        {
            // put your code here
        }
    });
    return s;
}
```

6. The insert method and the find method which are provided, are recursive methods, i.e the insert method calls the insert method on its subtrees. Also the find method calls the find method on its subtrees. Implement a method which calculates the depth of the tree using the same strategy.

7. Use a similar strategy to find the biggest element in the tree. Firs identify on the drawings of trees you made earlier where you can find the biggest element.

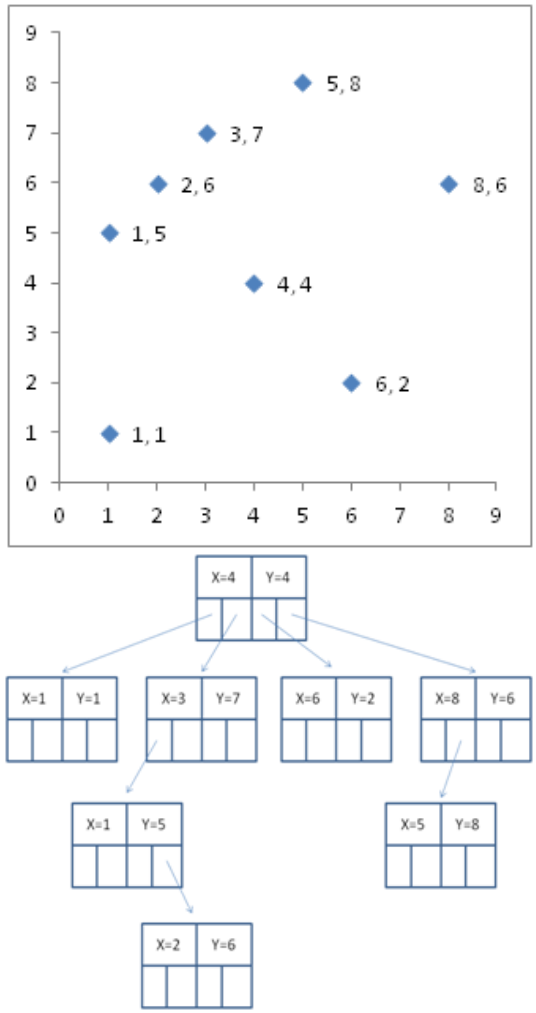


Figure 10.3:

8. John wants to store a set of points in the (x,y) plane in a data structure which allows him to find points quickly. He intends to use a binary search tree, but he cannot decide which of the coordinates he should use for building the binary search tree. Therefore, he decides to modify the binary search tree such that each node can have four children. Each of these four subtrees contains coordinates which do relate as follows to the coordinates of the point stored at the parent's node (see figure 10.3):

   - x and y coordinate are smaller or equal
   - x coordinate is smaller or equal and y coordinate bigger
   - x coordinate is bigger and y coordinate is smaller or equal
   - x and y coordinate are bigger

   Provide an implementation of the TreeNode class for this type of trees.

Implement a method which finds the point with the smallest x-coordinate in the tree.

9. In the tree class, implement a method "void swapTree()" which swaps the left and right subtree. So, each node in the left subtree should be in the right subtree and the other way around. The swap should occur in all subtrees. You can not make new TreeNode objects to accomplish this, simply change the pointers in the tree.

10. This question investigates the possibility of implementing a priority queue using a (binary search) tree.

    (a) Implement the push operation on the priority queue class such that it uses a tree to store the data in a sorted manner. (This is simply calling the appropriate method on the tree class in the right way).

    (b) Assuming that the tree is well balanced, what is the time complexity of the push operation?

    (c) Implement a pop operation on the priority class, assuming the data is stored in a binary search tree. In order to accomplish this, you need to extend the tree class with some new code as well.

    (d) Assuming that the tree is well balanced, what is the time complexity of the pop operation?

11. Consider a well sorted and balanced binary tree, such that each element in the left subtree is always smaller than each element in the right subtree and that the number of elements in the tree $N = 2^k - 1$ for some depth k.

    (a) Give a method which computes the median of the numbers in the tree. The median of a sorted sequence of elements is just the middle one.

    (b) Give a method which finds the minimum.

    (c) Give a method which calculates the average.

    (d) Give the time complexity of each of three above operators.

10.3 Exercises on the Dictionary - once more

If a binary search tree is well balanced, the time complexity of the find method and of the insert method will be $\mathcal{O}(\log(n))$ while with a linked list we had $\mathcal{O}(1)$ for the insert and $\mathcal{O}(n)$ for the find. Hence, it might be more efficient in many cases to implement your dictionary using a binary search tree in stead of using the linked list.

1. Have a look at the Dictionary class you implemented before. Use it as a start to create a new DictionaryTree class. In this class replace the linked list by a binary search tree. Just as with your original dictionary, you will store the key and the value in a pair object. This means that the insert method of the tree will have to be able to compare those pairs, to ensure the order on the tree. Obviously, we want to store the pairs ordered on their key. So, make sure the pairs you will store in the tree do implement a "compareTo" method which compares the keys. Look at the pair in the priority queue.

2. You are now ready to adapt the "add" method of your dictionary to work on the binary search tree rather than on the linked list.

3. The big disadvantage of the "find" method you implemented on your dictionary before is that it actually does not reuse the find method of the linked list. What was exactly the problem?

4. We will now do something smarter with our tree. Actually, the method we use here can easily be used on the linked list also. First step is to adapt the find method of the Tree we have. Change it such that it does not return true if it finds the given element,but rather the element itself.

5. Adapt the findKey method of the dictionary, such that it finds the given key in the binary search tree. Do not change the tree class, but simply call the find method on the binary search tree. Take care, the result will now be a pair (that's the consequence of your change in the previous assignment). Also, the argument of the findKey should be a key, while the argument of the find method in the tree should be a pair !!!

Chapter 11

Optimizing Binary Search Trees: Splay Trees and Red-Black Trees

See the course slides for the explanation, sample code and figures of splay trees and red-black trees.

11.1 Excercises

1. The provided Splay Tree implementation is a subclass of the traditional Binary Search Tree. In order to implement splay trees efficiently, every node should keep a pointer to its parent node. Adapt the TreeNode implementation so that a node can keep a pointer to its parent. Change the manipulation methods of the binary search tree to make sure that the pointers are set and updated consistently.

2. Write a splay tree based dictionary where the tree is splayed upon every element access.

3. Draw a red-black tree of depth 4 with the keys 1 to 15. Color the nodes in three different ways such that the tree is compliant with the red-black tree rules.

4. Assume a "relaxed red-black tree" where we dropped the red-black tree requirement that the root should be black. Consider such a tree where the root node is red. If we simply color the root node black and make not further changes, is the resulting tree a red-black tree? If so, why is the black root node requirement defined? If not, why not?

5. Draw the red-black tree that results of successively inserting the keys 41, 38, 31, 12, 19 and 8 into an initially empty red-black tree.

6. Consider a red-black tree formed by inserting n nodes with the RB-insert method. Argue that if $N > 1$, the tree has at least one red node.

Chapter 12

Graphs

A graph consists of a set of nodes and connections between the nodes, as in figure 12. A graph can for instance be used to represent a map: each city is a node, the roads connecting cities are the edges.

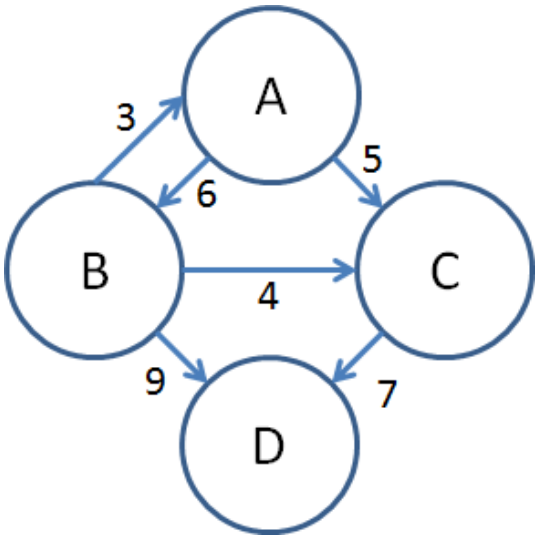


Figure 12.1: A directed acyclic weighted graph

Using the graph terminology, the nodes are called *vertices* (single is *vertex*). The connections between the vertices are called *edges*. Edges can have a *weight*, expressing for instance the distance between the cities in the graph, or some other concept. A graph with weights is a *weighted graph*, other graphs are *unweighted graphs*.

So, a 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 $(u, v) \in E$.

In the graph shown in figure 12, each edge has an arrow, i.e. the edges have a *direction*. Graphs in which all edges have a direction, are called *directed graphs*. Graphs in which edges do not have a direction are *undirected graphs*.

For instance, in most maps, the distance between A and B is the same as the distance between B and A. Rather than representing this by two edges (from A to B and from B to A), a single undirected edge is used.

Two vertices i and j are *adjacent* if they are connected by an edge (i, j) and the edge (i, j) is said to be *incident on* the vertices i and j. For example, in figure 12, the vertices A and C are adjacent, while A and D are not. 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.

A *path* is a sequence of adjacent vertices. For example, in figure 12, there are several paths from A to D (e.g. ABD, ACD and ABCD), but there is not a path from D to A. If there is a path from every vertex to every other vertex, the graph is *connected*.

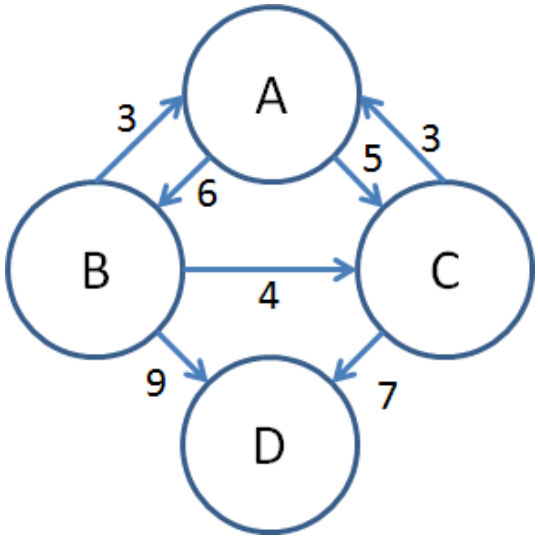


Figure 12.2: A directed cyclic weighted graph

Any path starting from a given node that terminates in the same node, is called *a cycle*. Graphs having at least one cycle are *cyclic*, while all other graphs are *acyclic*. The graph shown in figure 12 is a cyclic graph, because there is a path ABCA.

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 $E' \subseteq E$. For example, the graph in figure 12 is a subgraph of the graph in figure 12.

12.1 The adjacency matrix representation

The adjacency matrix representation is a very simple way of representing a graph. In the explanation here, we will focus on representing directed weighted

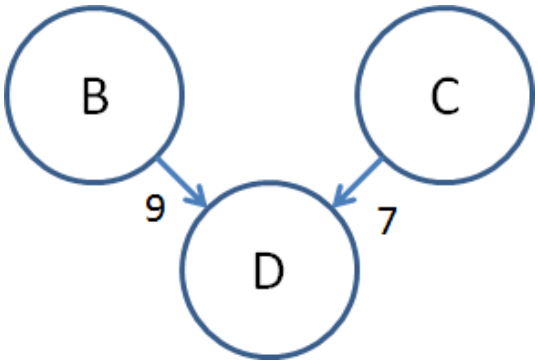


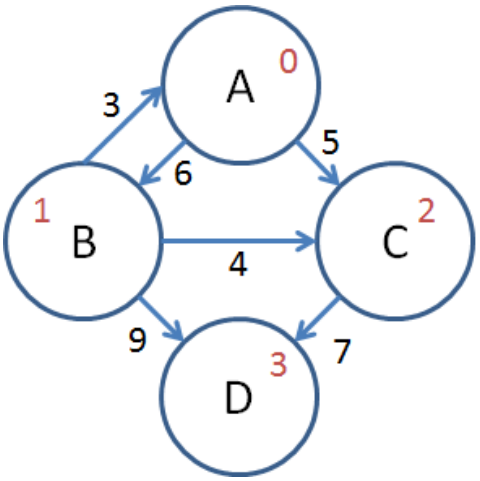
Figure 12.3: A subgraph of a directed acyclic weighted graph is again a directed acyclic weighted graph

graphs, but the representation can easily be adapted to undirected or unweighted graphs.

The idea is very simple: If a graph has N nodes, then we represent it by an N-by-N square matrix. And we decide on some numbering of the nodes. For example, in figure 12.1, we decide to label node A as 0, B as 1, ... In this way, we can store the weight of the edge between node A and node B (which is 6), by storing it in the matrix at row 0 (because node A is at position 0) and at column 1 (because node B is at position 1). The origin of the edge is the row position, the destination is the column position.

So, 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

$$a_{ij} = \begin{cases} w(i,j) & \text{if} (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$



0	6	5	0
3	0	4	9
0	0	0	7
0	0	0	0

Figure 12.4: Any graph can easily be represented by a matrix

What number should we actually put in the matrix at position a_{ij} when

there is no edge connecting nodes i and j? This depends a little bit on the application: if the weights in the graph are representing association strengths between concepts, then it is logical to assume that an association strength of zero, corresponds with the fact that there is no association. So, a weight of zero is appropriate in that case to express that there is no edge present. However, if the weights are expressing distances in a map, a weight of zero actually means that the distance between both nodes is zero. So, in order to express that there is no connection between two nodes, it is better to use a very big number in this case. In the matrix representation we develop here, we will use a weight of zero to express the fact that two nodes are not connected, as in figure 12.1.

12.1.1 The matrix class

The matrix class simple represents an N-by-N square matrix. It supports a constructor (which initializes all elements to zero), a method to set a value at a specific (row,column) in the matrix and a get method to retrieve the value at a specified (row,column).

```
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.
   }
}
```

12.1.2 The graph class

If a matrix class as defined in section 12.1.1 is available, then the a minimalistic adjacency matrix based graph class is very easy to implement:

```
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);
    }
}
```

12.2 The adjacency list representation

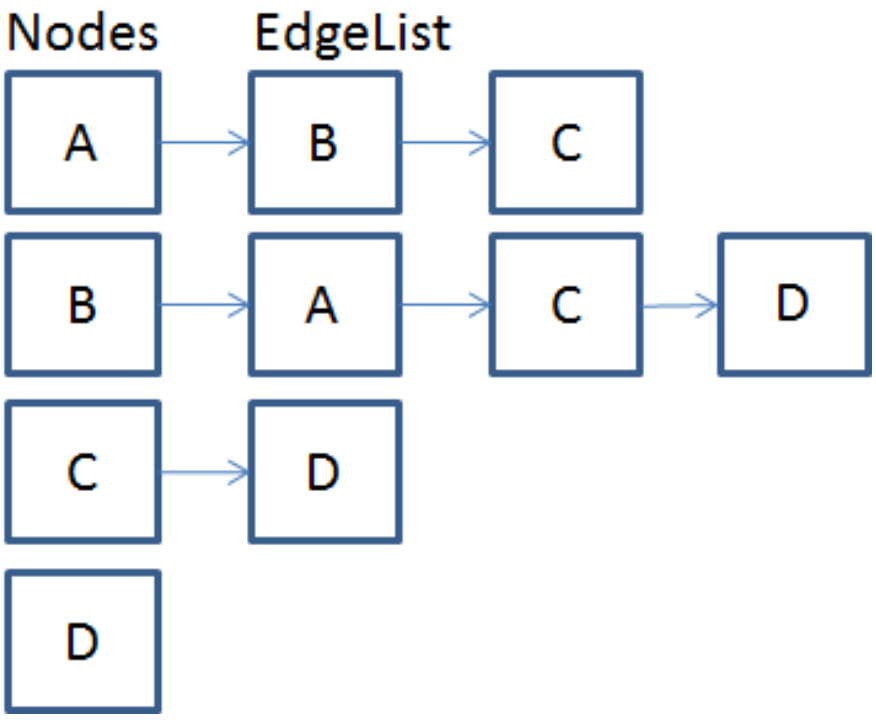


Figure 12.5: The adjacency list representation for the same graph

The adjacency list representation is based on the idea that all nodes should be stored explicitly in some datastructure, to allow for easy traversal of all the

nodes. In the initial representation we will develop here, nodes will be stored in a vector, but many other possibilities exist. In figure 12.2, the left part is showing the set of nodes, containing the nodes A,B,C and D. Now for each node, a separate structure (the adjacency list or edge list) is kept which stores all the edges starting from the current node. So, the adjacency list of node A contains the edges AB,AC,AD. In the figure 12.2 only the destinations of the nodes are shown in the adjacency lists.

12.2.1 The graph class

Representing this structure in JAVA code requires some more effort:

```
public class Graph
{
    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;
        }

    }

    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);
        }
    }

    private Vector nodes;

    public Graph()
    {
        nodes = new Vector();
    }

    public void addNode(Comparable label)
    {
        nodes.addLast(new Node(label));
    }

    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;
    }

    public void addEdge(Comparable nodeLabel1,
                        Comparable nodeLabel2)
    {
        Node n1 = findNode(nodeLabel1);
        Node n2 = findNode(nodeLabel2);
        n1.addEdge(new Edge(n2));
    }
}
```

12.3 Graph Algorithms

12.3.1 Breadth first traversal of a directed acyclic graph

12.3.2 Finding a path

A simple depth first search / breadth first search is used here to verify whether the path exists. This approach assumes that we have a directed acyclic graph; in that case the code to traverse a Tree can be adapted:

```
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;
}
```

12.4 Exercises

1. Give a proof for the so-called *handshaking lemma*: If $G = (V, E)$ is an undirected graph, then

$$\sum_{v \in V} degree(v) = 2|E|$$

2. (adjacency matrix representation) Complete the implementation of the matrix class defined in section 12.1.1. The methods get and set should run in constant time. Use datastructures developed in this course as a starting point.

3. (adjacency matrix representation) Add a toString method to the graph which outputs all the nodes and all edges.

4. (adjacency list representation) Add a toString method to the graph which outputs all the nodes and all edges.

5. (adjacency list representation) In the current implementation (section 12.2), edges do not contain any information. Extend the edge class such that you can specify a weight with every edge. Change the addEdge method to add this weight. Change the toString methods to print the weight as well.

6. (adjacency list representation) The method given in section 12.3.2 verifies whether a path between two given nodes exists. Extend this method to return the actual path, represented as a list of nodes.

7. (adjacency list representation) Also test the findPath method on a graph that contains cycles. Why does it not work?

8. (adjacency list representation) One strategy to avoid problems with cycles in the findPath method is to keep a list of visited nodes in the findPath method: whenever a node is popped from the to-do list, it is added to the visited list. Whenever a node is already visited (is in the visited list), it is not added to the to-do list anymore. Implement this strategy.

9. (adjacency list representation) The approach developed in the previous question to avoid problems with cycles is rather slow due to the search time in the linked list. A method which is often faster is the following: in the nodes of the graph, add an extra variable named "visited" (a boolean) in which we store whether this particular node was visited before yes or no. Before we can start searching for a path, we first set this boolean in all nodes to false. During the search for a path, we check whether the node was already visited before simply by checking this boolean. Change the node class such that we can check whether a given node was already visited or not. Also add a method to change the visited status of the node to a given value. Add a method to the graph class which sets the visited status of each node to false. Change the findPath method of the graph class such that it does not get stuck in cycles anymore, by checking whether each node was visited before.

10. (adjacency list representation) The findNode method (section 12.2) is rather long code as it needs a for loop and some logic to loop over all nodes and check them. Better would be that the findNode method on the graph, calls the find method on the underlying data structure, in this case the vector. Change the findNode on the graph to simply call the find method of the vector.

11. (adjacency list representation) The adjacency list representation developed so far uses vectors to store nodes and edges. Replace the type of both variables to LinkedList. Does everything still works? if not, correct and/or complete the implementation of the linked list.

12. (adjacency list representation) The adjacency list representation developed so far uses vectors to store the nodes. As a result the findNode method has time complexity $\mathcal{O}(n)$. This is an important drawback, as this method is called very often. For instance, the addEdge method does

two calls to the findNode method. Change the implementation of the Graph class, such that the time complexity of the findNode method is reduced to $\mathcal{O}(\log(n))$.

13. (general) Any binary tree can be represented by a graph. Extend the graph class with a method which traverses the entire tree and adds all connections in the tree to a graph.

14. (general) Can the graph constructed in question 13 be cyclic? Why (not)?

15. (general) Explain how the edge-list representation can be adapted to represent undirected graphs. Explain how the matrix representation can be adapted to represent undirected graphs.

16. (general) Explain the differences in memory usage for representing directed graphs in the matrix representation and the edge-list representation.

Chapter 13

Sorting

The goal of a sorting operation is to have all elements in a linked list or vector sorted with the least memory and processing efforts possible.

13.1 Insertion sort

First, we will have a look at a naive method, which can be the basis of more performant sorting algorithms. Suppose you have a linked list containing the elements (5 1 7 3 6 4). We want to have the list sorted into (1 3 4 5 6 7). One method to accomplish this is actually to build a second linked list and to make sure that we insert every element of our current list at the right position in the new linked list. Given the operations we already have on linked lists, we can achieve this very very easily:

```
public LinkedList insertionSort()
{
    LinkedList result = new LinkedList();
    ListElement d = head;
    while(d!= null)
    {
      result.addSorted(d.el1);
      d = d.rest();
    }
    return result;
}
```

The disadvantage of this sorting method is that it is slow and that it uses a lot of memory: it actually performs n times an insert on a sorted list (n times $\mathcal{O}(n) = \mathcal{O}(n^2)$). Moreover, it copies every element of the linked list into a new linked list.

Let us now apply the very same idea on a vector:

```
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);
   }
}
```

The above code shows an important further improvements compared to the rather naive version of the linked list: it can perform the sort *in place*, this means that it does not require any additional storage.

13.2 Tree sort

So far, we've shown that you can make an insertion sort on linked lists which takes $\mathcal{O}(n^2)$ time and which requires additional storage. We've also made an insertion sort on a vector, which also takes $\mathcal{O}(n^2)$ time, but which does not require additional storage. Now, we will focus on reducing the $\mathcal{O}(n^2)$ time for sorting a linked list. Imagine the following approach: (1) you traverse all elements of your linked list and insert them into a binary search tree. (2) You traverse the binary search tree in the correct order to have the elements from small to big and you add each element you traverse to a new linked list. This algorithm is called "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.el1);
        d = d.cdr();
    }
    return sortTree;
}
```

The time complexity of the tree sort algorithm is $\mathcal{O}(n \log(n))$: In the first step, the linked list is traversed, and for each element the element is added to a binary search tree (n times $\mathcal{O}(\log(n))$ if the tree is well balanced. In the second step, each element is added to the front of a linked list (n times $\mathcal{O}(1)$).

13.3 Merge sort

```
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;
}

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);
}
```

Merge sort is a well known example of a divide-and-conquer algorithm

13.3.1 Analysis of the Time Complexity of Merge sort

Remember that in merge sort, a sorted linked list is broken up into two smaller list of halve of the original size. These two sublists are recursively sorted and then these sorted sublists are aggregated back into a single list again. So, the time complexity of merge sort is

$$T(n) = 2T(n/2) + n - 1 \tag{13.1}$$

So, this recurrence relation is the same as equation 1, with $a = 2$, $b = 2$ and $f(n) = n - 1$. According to equation 2, the solution to this recurrence relation

is given by

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

13.4 A Lower bound for the time complexity of comparison sort

In this section, we will show that no sorting algorithm that uses only comparisons (i.e. *comparison sort*) can do better than $\mathcal{O}(n\log n)$. Once this theoretical lower boundary on the time complexity of comparison sort is established, we will investigate how sorting algorithms can be developed on other principles than element comparison. Such algorithms, that require additional assumptions on the data, can outperform comparison sort.

Although several different proofs exist for this fundamental lower bound on the time complexity of comparison sort, we will show that the lower bound on the time complexity of comparison sort is $\mathcal{O}(n!)$ and then show that $\log(n!) = \Omega(n\log n)$.

Lemma 1. $\log(n-1) \geq \log(n) - \frac{\log e}{n-1}$

Proof. From the Taylor series of e^x we have that $e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + ...$, So $e^x > 1 + x$. Now take $x = \frac{1}{n-1}$, then we have:

$$\begin{aligned} e^x &> 1 + x \\ e^{\frac{1}{n-1}} &> 1 + \frac{1}{n-1} \\ e^{\frac{1}{n-1}} &> \frac{n}{n-1} \\ \log(e^{\frac{1}{n-1}}) &> \log\left(\frac{n}{n-1}\right) \\ \frac{1}{n-1}\log(e) &> \log(n) - log(n-1) \\ \log(n-1 &< \log(n) - \frac{1}{n-1}\log(e) \end{aligned}$$

□

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

Lemma 2. $(1-c)\log(n) - c\log(e) \geq 0$

Proof. for c=1/2 we have:

$$1/2\log(n) - 1/2\log(e) \geq 0 \tag{13.2}$$

$$\log(n) \geq \log(e) \tag{13.3}$$

So if we take $n \geq e$, so $n \geq 3$, then for $c = 1/2$ we have that $log(n!) \geq cnlog(n)$ □

Lemma 3. $n\log(n) \geq \log(n!) \geq cn\log(n)$

Proof. A proof for $n\log(n) \geq \log(n!)$ is trivial. We will construct a proof for $\log(n!) \geq cn\log(n)$ by induction. Hence, we assume that the lemma is proven for $n-1$, so we assume $\log((n-1)!) \geq c(n-1)\log(n-1)$.

$$\begin{aligned}
\log(n!) &= \log(n)\log((n-1)!) \\
&= \log(n) + \log((n-1!)) \\
&\geq \log(n) + c(n-1)\log((n-1)!) \\
&\geq \log(n) + c(n-1)\left(\log(n) - \frac{\log(e)}{n-1}\right) && \text{[Lemma 1]} \\
&= \log(n) + c(n-1)\log(n) - c\log(e) \\
&= c(n)\log(n) + (1-c)\log(n) - c\log(e) \\
&= c(n)\log(n) && \text{[Lemma 2]}
\end{aligned}$$

□

Lemma 4. $\log(n!) = \Omega(n\log n)$

Proof. Follows immediately from Lemma 3. □

Theorem 4. *Any comparison sort algorithm must at least use $\lceil log(n) \rceil$ comparisons for some input sequence.*

Proof. □

Chapter 14

Coding guidelines

14.1 Topic 1: System.out.println and toString

When we want to display some text information on the screen, we are used to use the System.out.println construct, see below. In that example, three lines of text are printed, showing "5", "3.14" and "Hello World". This works even though "x", "y" and "z" have different types (int, double and string).

```
public class test
{
   public static void main(String[] args)
   {
      int x = 5;
      double y = 3.14;
      String z = "Hello World";

      System.out.println(x);
      System.out.println(y);
      System.out.println(z);
   }
}
```

So, what if we use System.out.println() on other types, let's say to print information on an object of the type Car? An example is given below.

```
public class Car
{
   private String color;
   private int numberOfPassengers;

   public Car(String someColor, int nrPassengers)
   {
      color = someColor;
```

```
        numberOfPassengers = nrPassengers;
    }

    public static void main(String[] args)
    {
        Car someCar = new Car("red",5);
        System.out.println(someCar);
    }
}
```

The output of the program shown in table 2 looks like "Car@3e25a5". This does not look nice. What is happening here? Internally, the "println" method is performing two different tasks:

1. Converting the argument (can be any type, e.g. int, double, string, but also Car, ...) to an object of the type String.

2. Show that String on the screen.

Java has a built in possibility to convert all simple types to String: it knows how to convert int, double, ... into a string. But how can it know how to convert Objects (reference types) into the right String? Look at our Car example shown in table 2, how can Java know what should be shown on the screen? It simply cannot.So, in order to deal with this problem, the following solution was adopted: in the Object class, a method "toString" is implemented, which converts the object into a default string, which consists of the name of the class, the "@" sign and some numbers (see table 3 for the JAVA specification of this method). As all objects (all instances of all classes you ever make) are by default subclasses of the Object class, all objects automatically inherit this default toString method.public String toString()Returns a string representation of the object. In general, the toString method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.The toString method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of: getClass().getName() + '@' + Integer.toHexString(hashCode())By implementing our own toString method in our Car class (overloading the toString method from Object), we can easily display information of the instance of our Car class, see below. With this toString method implemented, the output of this example is "Car with color red".

```
public class Car
{
    private String color;
    private int numberOfPassengers;
```

```
    public Car(String someColor, int nrPassengers)
    {
        color = someColor;
        numberOfPassengers = nrPassengers;
    }

    public String toString()
    {
        return "Car with color " + color;
    }

    public static void main(String[] args)
    {
        Car someCar = new Car("red",5);
        System.out.println(someCar);
    }
}
```

Whenever you implement a new class, add a toString method and use System.out.println to display information on the objects of that class.

14.2 Headers

Start each and every JAVA source file with a small header. In the header put some minimal useful information. When submitting code for a course, make sure you mention at least your name and the name of the file. Imagine a professor printing homework from 20 students, each submitting ten classes, without any names or filenames on the pages.

```
/*
 * Car.java
 * Bart Jansen
 * 28/04/2011
 *
 * A simple car object to represent cars with a color
 * and a number of passenger
 *
 */


public class Car
{
    private String color;
    private int numberOfPassengers;

    public Car(String someColor, int nrPassengers)
    {
        color = someColor;
```

```
        numberOfPassengers = nrPassengers;
    }

    public String toString()
    {
        return "Car with color " + color;
    }

    public static void main(String[] args)
    {
        Car someCar = new Car("red",5);
        System.out.println(someCar);
    }
}
```

14.3 Indentation

Indentation is about the layout of your code. Layout might seem to be unimportant, but actually it is. Look at the code below: which of both versions is easier to read?

```
public class Car {private String color;private
int numberOfPassengers;public Car(String
someColor, int nrPassengers){color =
someColor;numberOfPassengers = nrPassengers;}
public String toString(){return "Car with color
" + color;}public static void main(String[]
args) {Car someCar = new Car("red",5);
System.out.println(someCar);}}
```

```
public class Car
{
    private String color;
    private int numberOfPassengers;

    public Car(String someColor, int nrPassengers)
    {
        color = someColor;
        numberOfPassengers = nrPassengers;
    }

    public String toString()
    {
        return "Car with color " + color;
    }

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

```
        Car someCar = new Car("red",5);
        System.out.println(someCar);
    }
}
```

There are many different indentation styles and rules. It does not matter that much what exact you are using, as long as it is a consistent style. From the example above, you can observe the following:

1. The class definition starts at the left of the page and not somewhere in the middle "public class Car"
2. The declaration of each variable is at the same indentation as the declaration of each method. Each declaration is one tab to the right compared to the class definition
3. Inside each method, all statements are at the same indentation, one tab to the right compared to the method declaration.
4. Inside a method, for loops or other blocks can exist, resulting in more indentation levels.When submitting code for an assignment, a project, an exam, ... please use proper indentation. Your editor will help you to obtain it automatically.

14.4 Naming Conventions

When writing program code, one of the main objectives should be to write the code as easy to read for yourself and others. Besides adopting indentation conventions and writing suitable comments, clear naming of classes and variables is very useful.Consider the following piece of code:

```
public class a
{
    double apple;
    double Pie12;

    public a(double a1, double a2)
    {
        this.apple = a1;
        this.Pie12 = a2;
    }

    public double crumble12(a dog)
    {
        return Math.sqrt((apple - dog.apple) *
                         (apple - dog.apple) +
                         (Pie12 - dog.Pie12) *
                         (Pie12 - dog.Pie12));
```

```
    }
}
```

It requires some clear effort to understand the purpose of this class. Below is exactly the same code, with the same layout and exactly the same functionality, now employing clear names:

```
public class Point
{
    double x;
    double y;

    public Point(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public double distanceTo(Point p2)
    {
        return Math.sqrt((x - p2.x) *
                         (x - p2.x) +
                         (y - p2.y) *
                         (y - p2.y));
    }
}
```

In order to have easy to read code, the conventions should be used:

- The name of a class starts with a capital.
- As a consequence of that the constructor should also start with a capital, as it should have exactly the same name as the class name (imposed by JAVA, not by coding convention).
- All other things (variable names and method names) in your code start with small letters.
- So, in order to declare an object of the type point, you should put: Point corner = new Point();
- The names of the classes, methods and variables should be chosen such that their names match the concepts they represent: so Point for a Point class and not Apple to represent Points.
- Variables should not be named $x12$ or $i_3 2_3 3$, but should have clear names. Only in a limited set of cases, single letter variables are accepted: some mathematical concepts (x and y are used to represent coordinates in a two dimensional point), as an index in a for loop (for(int i=0;i¡10;i++)).

- When the name of a variable, a method or a class consists of more than one word, start all words except the first with a capital, do not use underscore. So, "*distanceToCenter*" is ok, "*distancetocenter*" and "$distance_to_center$" are not ok.

14.5 Method length

Each method inside a class should only contain the code to provide a specific functionality. Therefore, methods cannot be longer than a single printed page. In general, methods will almost always be much shorter than this. Writing shorter methods helps improving the readability and maintainability of your code!Example: if you have a nice user interface with a splash screen appearing, showing the progress of loading the application and then fading out, you should clearly be able to identify these three steps in the code. In a well designed splash screen class, you will have a method to make the screen appear, a method to update the information on the screen and one to make it fade. Putting these three functionalities in a single method is NOT a good practice.

14.6 Code duplication

Consider the program shown below, it shows three rows of ten stars. It is very simple, but repeats the same block of code three times. Better is to make a method to draw a single line of stars and call this method three times. In general, this will result is shorter code which is easier to read. Such code is also easier to maintain, as you need to change only a single method to update all three rows. This will result in less errors in your code.

```
public class Blink
{
   public static void main(String[] args)
   {
      for(int i=0;i<10;i++)
      {
         System.out.print("*");
      }
      System.out.println();
      for(int i=0;i<10;i++)
      {
         System.out.print("*");
      }
      System.out.println();
      for(int i=0;i<10;i++)
      {
         System.out.print("*");
      }
      System.out.println();
   }
}
```

```
public class Blink
{
   private static void drawLine()
   {
      for(int i=0;i<10;i++)
      {
         System.out.print("*");
      }
      System.out.println();
   }

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

14.7 Comments

Putting comments inside your code are improving the readability of your code. For instance, when implementing a complex algorithm, it might be very useful to add a reference to the paper describing the algorithm and to explain the main steps of the algorithm. These comments typically go in front of the implementation of the algorithm, so at the beginning of a class or a method. Inside the methods, important steps in the algorithm can be identified by some more comments.