University of Eastern Finland

School of Computing

Data Structures and Algorithms I

5 cp, 3621313
Spring 2017

Simo.Juvaste@uef.fi
13.3.2017 14:00

http://cs.uef.fi/pages/sjuva/dsaI.html

Contents

1 Introduction to algorithmics

⇒ Programming is much more than entering a computer program code.

- The programming process also includes program planning at several levels, as well as the final evaluation of the program.
- DSA courses teach an important part/aspect of information systems planning and programming.

1.1 From a problem to a program

⇒ The task (problem) must be defined precisely in order to know what to solve.

- For example:
  - "Write a program that calculates the sum of the figures.."
  - "Write a program that calculates and prints the sum of integers fed from the keyboard. Input will stop when you enter the number zero."
- Using the latter definition, it is much more likely, that different programmers make about the same kind of program.
- Precise definition of the task is mostly a matter of other courses.

⇒ Once the problem is defined, we can start working on the the principle of the solution, i.e., the algorithm.

Principle of solution, i.e., algorithm:

⇒ Precisely described way to solve the given problem.

- finite sequence of operations
- carried out in a finite period of time
- conducting a finite amount of work
- deterministic (usually)
- terminating

⇒ algorithm ≠ problem

Presentation of an algorithm

- Either directly using a programming language or the so-called pseudo code.
- A picture illustrates the operation of the algorithm.

⇒ An algorithm presented in pseudo code can be implemented straightforwardly using (any) programming language.

- In this course, the algorithms are presented mainly in the Java language, but using simplifications (pseudo code) to clarify the algorithms.
- We draw a picture of all algorithms and data structures...

Is algorithm ready?

- Algorithm correctness must be ensured.
  - Good planning made the correct implementation easier.
  - See also courses on programming technique and testing.
- Efficiency of the algorithm must be evaluated.
  - DSA I, DSA II, DAA
  - If the algorithm is not efficient enough, then:
    - Can it be improved?
    - Is it feasible to improve?
  - For a sequential algorithm: efficiency goes with speed
    - We can get more speed also by parallelization (but not more efficiency)
    - See course Parallel Computing

Implementation of the algorithm

- Programming courses (& DSA I).
- Gradual refinement of the algorithm pseudo code.
  - Modify the pseudo code towards a real programming language.
  - Abstract data types are changed to the concrete data structures.
- We must still remember efficiency during implementation.
  - A careless implementation can destroy efficiency.
    - A well planned and presented an algorithm will steer the implementation towards efficiency.
  - Careless use of libraries can destroy efficiency, or even correctness.
    - It is worth checking the exact operation of library methods before starting the implementation.

    - If a library has incomplete documentation (as to some extent, Java API), ensuring the effectiveness can be difficult (but by testing, see DSA II).

Testing of the implemented algorithm

- Mistakes may have happened during algorithm design or implementation.
- Full featured testing is not part of this course.
- Well designed and implemented algorithm is easy to test.
- Correctness of the program comes from correctness of algorithm, and from implementation via careful stepwise refinement.
- Confirmation of time complexity (DSA II).

Is efficiency important?

- What is meant by efficiency? (Design&implementation, or execution?)
- Performance/efficiency is not always essential (for small inputs, PCs).
- Avoiding bad inefficiency is important.
- Sometimes, the best performance (i.e., less execution time) is profitable, especially:
  - Servers
    - The more efficient the server software is, the more clients a single server can serve.
    - Electricity is saved.
  - Battery-powered devices
    - Execution of each operation takes energy, sleep mode does not.
    - More efficient algorithm (less execution time) takes less energy (battery capacity) → battery lasts longer.
  - Frequently performed operations.
    - Even a small saving is multiplied if the operation is performed often or/and by many devices.

DSA courses motivation & content

- Raising the abstraction level of algorithms (and resulting programs).
- Raising the level of abstraction of thinking of the programmers.
- Increasing modularity.
- Choosing the right data structure on all levels.
- Understanding the issues impacting efficiency.
- Predicting performance.

Why we need abstraction?

⇒ Programming is hard!

- Abstraction helps to clarify the programming work.
  - "Raise up from tinkering."
  - Inputs, operations, and outputs appear as clearer and more comprehensible entities.
  - The program will be clearer and more easily modifiable.
  - The more independent functional parts (modules) the program consists of, the easier it is to test each of them.

Every programmer imagines (sometimes) to produce clear, easily understandable programs.

- Still, programmers often find it difficult to understand programs produced by other programmers.
- Or his/her own programs, after (a) few years...

Abstraction supports solving the problem via stepwise refinement!

Example: Planning the phases of traffic lights

- Input: Possible routes of an intersection, routes that collide if used concurrently.
- Output: optimal grouping of routes, so that there will not be collisions.

The problem can be illustrated by the intersection of the Figure 1.1, where streets A and C are one-way and streets B and D two ways. There are seven possible driving routes. Among them, for example, the routes AB and DC can be used concurrently (as long as we have enough lanes), but using routes AC and DB concurrently causes a collision hazard.

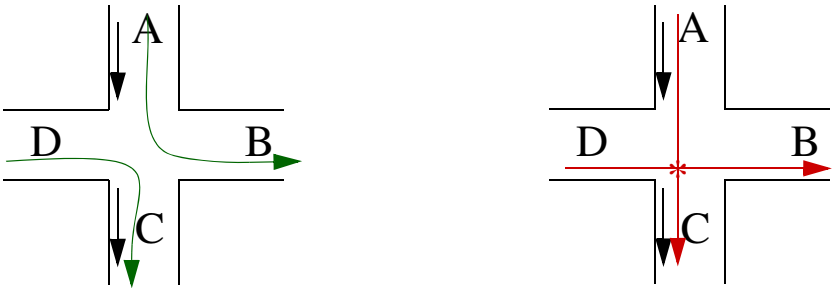


Figure 1.1: Intersection of streets.

We describe the problem as a graph, consisting of a set of nodes and a set of edges connecting the nodes. (Graphs are described briefly in Chapter 5 and in more detail on course DSA II). Lets model the intersection so that nodes represent routes and have an edge between two nodes only if those two routes can't be used concurrently. The intersection of Figure 1.1 can be modelled as a graph of Figure 1.2. Table 1.3 presents the same graph in an another form. Ones (1) of the table denote an edge, empty (or 0) denote a missing edge. Among these presentations, Figure 1.1 is best for humans and Table 1.3 is best for a computer. The graph of Figure 1.2 may not be optimal for anyone, but it contains all the needed information of the problem.

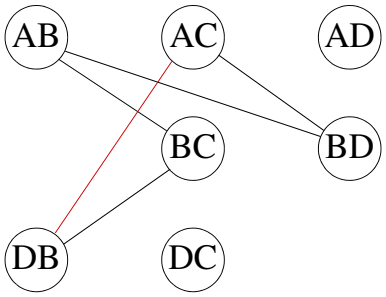


Figure 1.2: Graph of colliding routes.

We'll colour the nodes of the graph so that no edge has the same colour in nodes of both ends.

- The three-colour solution of Figure 1.4 is actually an optimal (minimal) in terms of number of colours.
- This graph cannot be coloured by using only two colours so that all neighbours would be of different colour.
  - Cycle AB, BC, DB, AC, BD, AB has odd number of edges.
- While colouring every other with one colour, and every other with second colour, there will be a problem.

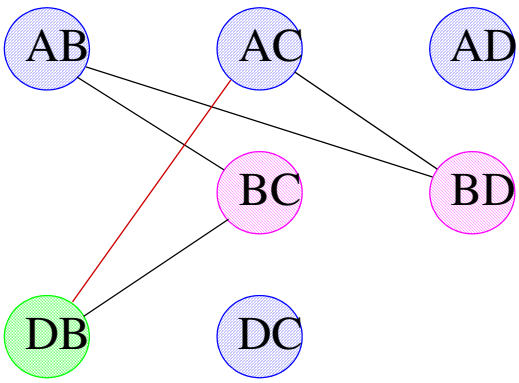


Figure 1.4: One possible grouping.

	AB	AC	AD	BC	BD	DB	DC
AB				1	1		
AC					1	1	
AD							
BC	1					1	
BD	1	1					
DB		1		1			
DC							

Figure 1.3: Matrix representation of the graph of Figure 1.2.

- Thus, we need >2 colours, and a 3-colour solution has been found. $\Rightarrow$ colouring is optimal!
- Applying colouring back to traffic lights?
  - Each color (node group) forms a light phase.
  - Same-coloured routes do not collide (because they are not neighbours).

How to find a solution to algorithmically?

- Colouring problem is difficult.
  - There are exponential amount of possible colourings.
  - There is no known fast solution to ensure optimal colouring, we need to very slowly try (almost) all possible colourings.
    - Graph colouring a so called NP-complete problem.
    - It is believed that an efficient (non-exponential time) solution can't be found!
- If we give up the optimality of colouring, we can create an algorithm that will quickly produce a reasonably good solution, but not necessarily the best solution.
  - Heuristic algorithm, randomized algorithm (exercise in Java on course DSA II).

A straightforward algorithm:

0. Colouring condition: Do not use the same colour at both ends of an edge.
1. Colour as many nodes with one colour as the colouring condition 0 allows.
2. Using the next colour, colour as many nodes as possible.
3. Repeat step 2 until all nodes have been coloured.

$\Rightarrow$ This is a so called greedy algorithm.

- Start somewhere, do whatever we can, continue.
- No backtracking, no heuristics, etc.
- Finds the optimal solution sometimes, but not always.
- Result depends on:
  - starting node
  - order of handling of nodes.

The above notation describes the algorithm pretty well, but is not nearly executable. Thus we need to refine it.

Let us write the above algorithm using a more Java-like notation. The stages 0-2 are expressed with method *greedyColor* and initialization, introduction of a new colour, and finish (stage 3) with method *greedyColorStart*.

```
public static int greedyColorStart(Graph G) {                          1
    mark all vertices non-colored;                                      2
    int numOfColors = 0;                                                3
    while (not all vertices colored)                                    4
        greedyColor(G, ++numOfColors);                                  5
    return numOfColors;                                                 6
}                                                                       7
                                                                        8
public static void greedyColor(Graph G, Color newColor) {              9
    for each uncolored vertex v of G {                                  10
        if (v not adjacent to any vertex with color newColor)           11
            v.setColor(newColor);                                       12
}                                                                       13
```

⇒ This is Java-based algorithm notation – it looks like Java, it is still precise for a human, but is not yet precise enough for a compiler.

The same algorithm, lines 10 and 11 refined:

```
public static void greedyColor(Graph G, Color newColor) {              1
    for (Vertex v : G.vertices()) {                                     2
        if (v.getColor() != noColor)                                    3
            continue;                                                   4
                                                                        5
        boolean found = false;                                          6
        for (Vertex w : v.neighbors()) {                                7
            if (w.getColor() == newColor)                               8
                found = true;                                           9
        }                                                               10
        if (! found)                                                    11
            v.setColor(newColor);                                       12
    }                                                                   13
}                                                                       14
```

⇒ This can be compiled using a Java-compiler and it works!! The readability suffers a bit, but it still is understandable.

- If necessary, we can add more steps of refining, e.g., a more structured English version between the initial version and the first Java-like version.

1.2 Abstract data types

⇒ Abstract data types (ADT) (also abstract data structure) are models to represent and process data.

Abstract data type is defined by describing the operations which are used to handle the data maintained by the ADT.

⇒ Operations (etc.) = interface of the abstract data type (”usage interface”).

Alternative illustration

- ADT is a tool to manage collections of data.
- We'll hang the data elements to be managed by the ADT.

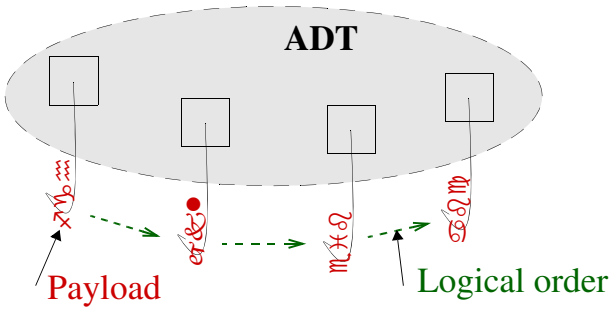


Figure 1.5: ADT as a tool to manage data.

⇒ We concentrate on the elements and relations between them!

- The selection of ADT dictates the logical order of the elements (if an order is needed).
  - Sometimes only part of the elements are "available" (as/if we won't need the rest).

Example 1.1. A set (as an abstract data type) is a collection of elements of the same (or compatible) type, e.g., personal data, strings, or vertices of a graph. Each element can be included in a set only once. Sets can be operated by, e.g., forming a union of two sets or by checking whether an element is included in a set or not.
Interface of the set can include, e.g.:

```
// returns union of this Set<E> and Set B
public Set<E> union(Set<E> B);

// returns whether element x is a member of this Set or not
public boolean member(E x);
```

Interface gives the user of the ADT all the necessary information to use it:

- types
- allowed operations (functions/methods)
- parameters of the operations
- return type of the operations
- restrictions of the operations
- time complexities of the operations

⇒ ADT is implemented in such a way that the operations of the interface function as described!

- Implementation includes the program code of the operations and the data structures needed to implement the ADT.
  - To facilitate the implementation, the implementer may add further operations for his/her own use.
    - These operations are not visible to the user (nor other issues related to the implementation).

Definition 1.2. The data type of a (programming language) variable is the set of valid values.

Example 1.3. Integer type can contain infinite amount of values: 0, 1, –1, 2, –2, 3, –3, ... In practice, the word length of the computer limits the set to be finite. For example, using 32 bits, we can represent 4 294 967 296 different values.

In Pascal, the type *set of 0..9* can contain sets { }, {0}, {1}, {2}, {3}, ..., {9}, {0,1}, {0,2}, ..., {8,9}, {0,1,2}, ..., {0,1,2,3,4,5,6,7,8,9}. Altogether there are $2^{10} = 1024$ different values.

Boolean type contains only two possible values, *false* and *true*.

Definition 1.4. A data structure is a collection of interconnected variables. Thus forming a structure of the programming language in which the bits could be drawn. Although, in the case of Java, precise drawing of bits would require more information about implementations of compiler and virtual machine.

- array
- object
- objects connected by references
- file

⇒ Abstract data type: collection abstraction ((exact) implementation unknown).

⇒ Data structure: a construct expressed using a programming language (accurate to a bit).

- In this course, we'll learn these both and the connection between them.
- In practice, the terms are mixed up in daily speech.

1.3 Complexity of algorithm execution

Algorithm requirements:

0. Algorithm must give correct result.
1. Algorithm must be easily understandable and implementable.
2. Algorithm must be efficient.

Efficiency?

- time complexity
  - how much execution time is used?
- space complexity
  - how much memory is used?
- hardware complexity
  - what are the hardware requirements (e.g., precision)?

This course concentrates on time complexity, we touch upon space complexity of bit, and mostly ignore hardware complexity (as it is not usually significant in practical algorithms).

How we assess execution time (time complexity)?

- The absolute wall clock time?
  - Important for the end user, at least occasionally.
  - Poor choice, as almost everything impacts:

    - input type
    - hardware
    - other processes (load)
    - caches
    - measurement instruments
  - Poor choice, as we can inspect only an implemented algorithm.
  - Obviously, for the final software, the wall clock time is relevant, preferably estimated in advance.
    - Well come back to this in course DSA II.

⇒ We need something more abstract than a second.

- Clock cycle (or machine language instruction) is better, but still the above factors impact.
  - Still only an implemented (compiled) algorithm can be inspected (with reasonable efforts).

⇒ We'll use unit "step".

```
a = 1;                                  1
b = 3;                                  2
```
(about) two steps

```
for (i = 0; i < 1000; i++)              1
    a = a + 1;                          2
```
(about) 1000 steps

Figure 1.6: Execution step

- A step is a single simple operation in an algorithm.
  - The above 2-step piece of code will take 2 or 4 machine language instructions, depending the implementation of the variables.
  - The execution time of the instructions can be 1-100 ns, depending the storage of the variables...
- We'll see in a moment that there is no need to worry about the accuracy of the number of steps here.

Execution time is related to the size of the input of the algorithm.

- Or to a similar quantity that dictates the execution time.
- Usually we ignore input and output.
  - Input is usually readily available (via a reference).
- Input size can vary between executions.
- Sorting in 1000 steps (or 0.1 seconds). Fast or slow?
  - 10, 1,000, 1,000,000, 1,000,000,000 elements?

⇒ We present the time complexity as a function of the input size!

```
for (i = 0; i < n; i++)                 1
    a = a + 1;                          2
```
(about) n steps

⇒ We do not fix the unit of a time complexity function. We'll call it a step (or keep it unitless).

- The real time of a single step depends on, among other things, the operation to execute and hardware used.
- Still, there can be n, $3 \times n$, $3 \times n + 4$, steps, depending how we calculate them. But such inaccuracy is mostly irrelevant.

Example 1.5. Time complexity function $T(n)=cn^2+b$, where b and c are constants, indicates that execution time is quadratic with respect to input size n. This means that if input size increases 10-fold, the execution time increases about 100-fold.

⇒ Time complexity is thus a function of input size (or another measure dictating the execution time).

- Often we use variable n to denote input size (number of elements).
  - We must, however, state it explicitly unless it is obvious in the given context.
- Occasionally we need also other variables if n is used for other purposes.
  - Especially if there are several inputs of different sizes.

Example 1.6. Time complexity of finding a key string from text string depends on both the key, and the text. Let us denote the length of the key with m and the length of the text with n. The time complexity of a simple matching algorithm is $T(n,m)=cnm$, where c is a constant.

Execution time does not only depend on size of the input, but often also on character of the input.

As we take this into account, we can divide the inspection into following three cases:

1. $T(n)$ means the worst case execution time, i.e., the longest possible execution time for input size n.
2. $T_{avg}(n)$ means the average execution time, i.e., the average execution time of all possible inputs of size n.
3. $T_{best}(n)$ means the best case execution time, i.e., the shortest possible execution time for input size n.

⇒ Usually we inspect the worst case.

- The best case is seldom interesting.
  - For example, in sorting, the best case input is readily sorted array. Time complexity of "sorting" that tells little about the goodness of the sorting algorithm.
- Analysing the average time can be quite challenging as we can not always assume all inputs to be equally probable.
  - There are exponential number of different inputs with respect to the input size.
  - If average time differs significantly from the worst case, then it is important.
- The upper limit of the worst case is usually the easiest to analyse (as would be the lower limit of the best case).
- Upper limit of the worst case is usually the most interesting in design phase.

We can compare complexity functions by comparing the speed of growth of the functions.

- Constants are not interesting. Hardware, compiler, etc. have their impacts.
- We only inspect the order of growth, or complexity classes.

Definition 1.7. Function order notations O, Ω, Θ and o:

1. $T(n) = O(f(n))$, if there exists positive constants c ja n_0 such that $T(n) \leq c \times f(n)$, for all $n \geq n_0$.
   [Read: $T(n)$ is of complexity class $f(n)$, "big-O", "ordo"; "asymptotically bounds from above"]
2. $T(n) = \Omega(g(n))$, if there exists positive constants c ja n_0 such that $T(n) \geq c \times g(n)$ for all $n \geq n_0$.
   ["omega"; "asymptotically bounds from below"]
3. $T(n) = \Theta(h(n))$, if $T(n) = O(h(n))$ and $T(n) = \Omega(h(n))$.
   ["theta"; "asymptotically bounds from above and below", "tight bound"]
4. $T(n) = o(p(n))$, if $T(n) = O(p(n))$ and $T(n) \neq \Theta(p(n))$.
   ["little-o"; "asymptotically strictly bounds from above"]

$\Rightarrow$ Provided suitable c, and letting input grow enough, the given bound applies.

Example 1.8. Let $T(n) = 5n+2$. Then $T(n) = O(n)$, which can be seen by choosing, e.g., $c = 7$ and $n_0 = 1$: $5n+2 \leq 5n+2n = 7n$, when $n \geq 1$. Besides upper bound n, we n^2, n^3 etc. are upper bounds (but not very meaningful). Because $T(n) = \Omega(n)$, which can be found true by choosing $c = 5$ and $n_0 = 1$, is n also an lower bound, thus n is tight upper (and lower) bound, i.e., $T(n) = \Theta(n)$.

Example 1.9. Upper bound proof: $T(n) = 2n^2 - 4n + 5$.
$2n^2 - 4n + 5 \leq 2n^2 + 5 \leq 2n^2 + 5 \times n^2 = 7n^2$, i.e, $2n^2 - 4n + 5 = O(n^2)$.

Example 1.10. Lower bound proof: $T(n) = 2n^2 - 4n + 5$.
$2n^2 - 4n + 5 \geq 2n^2 - 4n \geq 2n^2 - 4n \times \frac{1}{4}n$ [which is valid when $n \geq 4$] $= 1n^2$, thus $2n^2 - 4n + 5 = \Omega(n^2)$.

In lower bound proof, we can add multiplier $\frac{1}{4}n$ to the negative term, as when $n \geq 4$ (thus $n_0 = 4$), then the negative term grows, and thus decreases the total. On the other hand, e.g., multiplier n would have resulted the whole function as negative, and thus impossible.

Example 1.11. By examples 1.9 and 1.10 we can conclude that $2n^2 - 4n + 5 = \Theta(n^2)$.

Example 1.12. We'll show that function $T(n) = 5n+2$ is not $O(1)$, i.e., constant:
Antithesis: let us assume that $T(n) = O(1)$, then by definition 1.7 there would be positive constants c and n_0 such that $5n+2 \leq c$, when $n \geq n_0$. $5n+2 \leq c \Leftrightarrow n \leq (c-2)/5$. Because c is a constant, also $(c-2)/5$ is a constant. Thus inequality $5n+2 \leq c$ is not true when $n > \max\{n_0, (c-2)/5\}$, which is against initial assumption. So, we got a contradiction and the original assumption was true.

Example 1.13. We'll show that function $T(n) = 2n\log n + 3$ is not n:
Antithesis: let us assume that $T(n) = O(n)$, then by definition 1.7 there would be positive constants c and n_0 such that $2n\log n + 3 \leq cn$, when $n \geq n_0$. $2n\log n + 3 \leq cn \Rightarrow 2n\log n \leq cn \Rightarrow \log n \leq c/2 \Rightarrow n \leq 2^{c/2}$. Because c is a constant, also $2^{c/2}$ is a constant. Thus inequality $n \leq 2^{c/2}$ is not true when $n > \max\{n_0, 2^{c/2}\}$, which is against initial assumption. So, we got a contradiction and the original assumption was true.

Example 1.14. Figure 1.7 presents ten different time complexity functions. The first image is, however, misleading, as we can see on Figures 1.8 and 1.9.

Categorization of (time complexity) functions

- Exponential growth/time
  - E.g., 2^n, 3^n, $2^n/n$
  - usable only for **very** small input sizes.
- Polynomial time
  - E.g., n, n^2, n^5, n^{12345}, $n\sqrt[3]{n}$

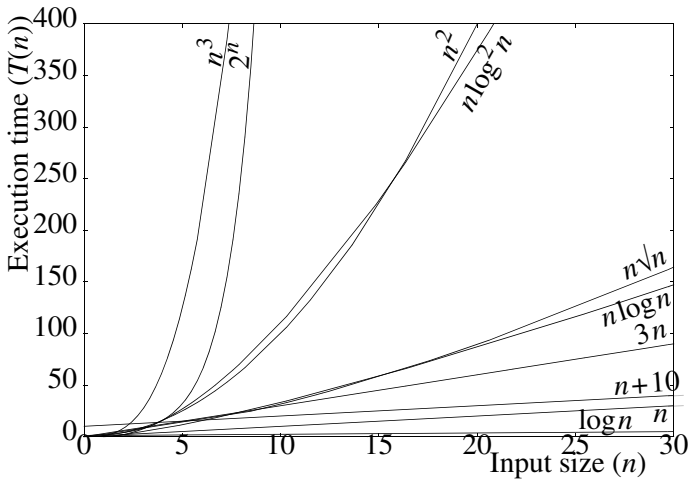


Figure 1.7: Time complexity functions.

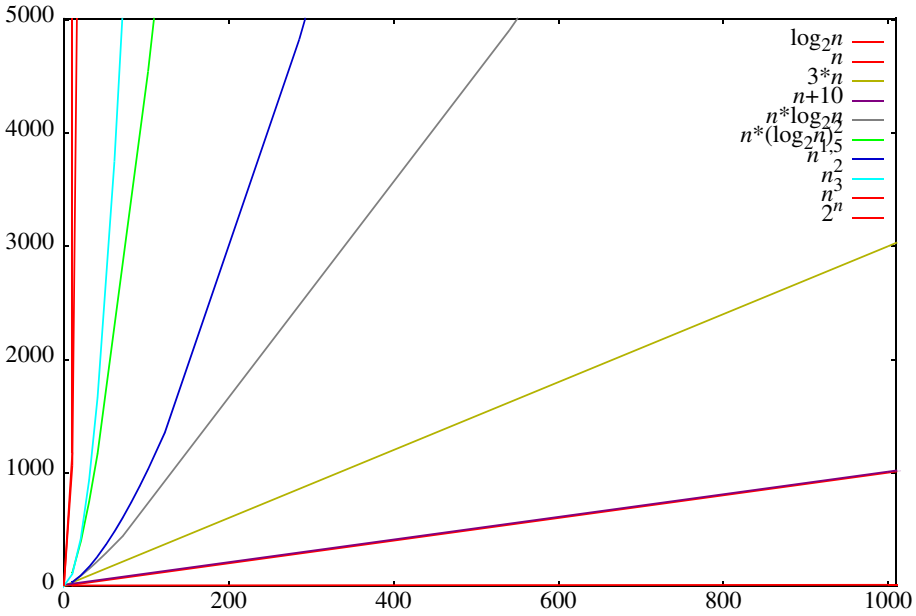


Figure 1.8: Time complexity functions, larger scale.

  - most common in practice.
  - efficient .. semi efficient
  - n linear
  - n^2 quadratic
- Logarithmic time
  - E.g.,. log$n$, loglog$n$, etc.
  - Note: log$n$ is $\log_2 n$ during this course (and usually in CS).
  - everything $o(n)$ sub-linear time complexity.
    - not in whole sequential algorithms handling the whole input.
- Unit time: $O(1)$ (time does not grow even if n grows).

⇒ Remember to observe the most significant component of the time complexity function!

Time $T(n)$, ms	Processed input size in given time:				
	1 sec	1 min	1 hour	1 day	1 year
$\log n$	10^{301}	$10^{18\,061}$	$10^{1\,083\,707}$	$10^{26\,008\,991}$	$10^{9\,493\,281\,943}$
n	1 000	60 000	3 600 000	86 400 000	31 536 000 000
$n+10$	990	59 990	3 599 990	86 399 990	31 535 999 990
$3n$	333	20 000	1 200 000	28 800 000	10 512 000 000
$n\log n$	140	4 895	204 094	3 943 234	1 052 224 334
$n\log^2 n$	36	678	18 013	266 037	48 391 041
$n\sqrt{n}$	100	1 532	23 489	195 438	9 981 697
n^2	31	244	1 897	9 295	177 583
n^3	10	39	153	442	3 159
2^n	9	15	21	26	34

Figure 1.9: Time complexity functions in tabular form.

Rules of complexity categories:

- For all constants $k>0$, $b>0$ holds $\log^k n = o(n^b)$.
  ($\log^k n$ means $(\log n)^k$)
  - I.e., all logarithmic functions grow slower than all polynomial functions.
- For all constants $a>1$, $b>0$ holds $n^b = o(a^n)$.
  - I.e., all exponential functions grow faster than all polynomial functions.

Comparing growth speeds of functions

- Usually comparing of functions is easy as long as we remember the previous classifications.
- Comparison can be done mathematically precisely and proved by inspecting the limit of the quotient of the functions when n grows infinitely:

$$\text{If } \lim_{n\to\infty}\frac{f(n)}{g(n)} = \begin{cases} 0, \text{ then } f(n)=o(g(n)) \\ c\neq 0, \text{ then } f(n)=\Theta(g(n)) \\ \infty, \text{ then } g(n)=o(f(n)) \end{cases} \tag{1.1}$$

- We can solve the limit easily using l'Hospital rule (if rules of function derivation are available/accessible):

$$\text{If } \lim_{n\to\infty} f(n)=\infty \text{ and } \lim_{n\to\infty} g(n)=\infty \tag{1.2}$$

$$\text{then } \lim_{n\to\infty}\frac{f(n)}{g(n)} = \lim_{n\to\infty}\frac{f'(n)}{g'(n)}. \tag{1.3}$$

- If both functions still grow infinitely after derivation, we repeat derivation as long as needed.

Example 1.15. $n\log n$ vs. $n^{1.5}$

$$f(n)=n\log n \tag{1.4}$$
$$g(n)=n^{1.5} \tag{1.5}$$

is $f(n)=O(n^{1.5})$ or $o(n^{1.5})$, or neither?

$$f(n)=n\log n \qquad g(n)=n^{1.5} \tag{1.6}$$
$$f'(n)=\log n+\frac{1}{\ln 2} \qquad g'(n)=\frac{3}{2}n^{\frac{1}{2}} \tag{1.7}$$
$$f''(n)=\frac{1}{n\ln 2} \qquad g''(n)=\frac{3}{4\sqrt{n}} \tag{1.8}$$

$$\lim_{n\to\infty}\frac{\frac{1}{n\ln 2}}{\frac{3}{4\sqrt{n}}}=\lim_{n\to\infty}\frac{4\sqrt{n}}{3n\ln 2}=\lim_{n\to\infty}\frac{4}{3\ln 2\sqrt{n}}\to 0 \tag{1.9}$$
$$\text{thus } f(n)=o(g(n)), \text{ i.e.: } n\log n=o(n^{1.5}) \tag{1.10}$$

Same result can be found by informally deduced, similarly to solving an inequality but with initially unknown direction of inequality:

$$n\log n \quad ?? \quad n^{3/2} \qquad | \quad :n \tag{1.11}$$
$$\log n \quad ?? \quad n^{1/2} \qquad | \quad ()^2 \tag{1.12}$$
$$\log^2 n \quad ?? \quad n \tag{1.13}$$

- Recall that $log^k n=o(n)$.
- $n^{3/2}$ grows faster.

⇒ Is is possible to create ever more efficient algorithms for each problem?

- For some problems there are known only inefficient (i.e., slow, usually exponential time complexity) solutions:
  - E.g., graph colouring, the travelling salesman problem.
  - Fast (polynomial time) solution might exist, but no one has yet found it (or, more likely, it does not exist).
  - For most difficult problems there are known faster heuristic solutions (that provide a relatively good solution faster).
- For many problems there exists a proven lower bound of time complexity:
  - e.g., general (comparison-based) sorting is $\Omega(n\log n)$ (DSA II)
  - these proofs can be difficult
  - Reaching the lower limit may be difficult. Sometimes the algorithm to reach the lower bound has not been found (thus, it is not known whether the lower limit is tight, i.e., whether it can be achieved or not).
- Further, there are problems that can not be solved algorithmically.
  - E.g., halting problem

⇒ See Theory of Computation and DAA courses ...

1.4 Calculating the time complexity in practice

⇒ Calculating the time complexity of a given algorithm can be very easy, or very difficult.

Usually it suffices to know very basic math:

- adding and multiplying
- simplifying arithmetic series
  - complexity classes (O-notation) help a lot, we won't need exact results.
- computing logarithms and exponents (in some algorithms)
- recurrences (in recursive algorithms)

Basic rules of time complexity calculation

- Let execution time of program section P_1 be $T_1(n)=O(f(n))$
- Let execution time of program section P_2 be $T_2(n)=O(g(n))$

Sum rule

- $T_1(n)+T_2(n)=O(\max\{T_1(n),T_2(n)\})$

⇒ Interpretation: slower section dictates the time complexity of two consecutive sections.

- If $T(n)=a_1n^k+a_2n^{k-1}+\cdots+a_{k-1}n^2+a_kn+a_{k+1}$, then $T(n)=O(n^k)$.
- Due to sum rule, we can leave less significant term(s) out of calculation altogether.
- Proof can be deducted from the definition O-notation 1.7 (p. 11).

Product rule

- $T_1(n)\times T_2(n) = O(f(n)\times g(n))$.
- Consequently: if $T(n)=a\times f(n)$, (where a is a positive constant), then $T(n)=O(f(n))$.

⇒ I.e., we can ignore positive constant factors.

Example 1.16.

$$\begin{aligned} T(n) &= 3n^2+5n+8, \\ &= O(\max(3n^2,5n,8)) && \text{(sum rule)} \\ &= O(3n^2) && \text{(product rule)} \\ &= O(n^2). \end{aligned}$$

Calculating the time complexity of an algorithm

⇒ Calculate from inside to out!

Example 1.17. Bubble sort algorithm can be presented as:

```
public static void bubbleSort(Comparable A[]) {                    1
    for (int i = 0; i < A.length-1 ; i++) {                        2
        for (int j = A.length-1; j > i ; j--) {                    3
            if (A[j-1].compareTo(A[j]) > 0) {                      4
                Comparable tmp = A[j-1];                           5
                A[j-1] = A[j];                                     6
                A[j] = tmp;                                        7
            }                                                      8
        }                                                          9
}   }                                                              10
```

- Input is the parameter array.
- Input size is the number of elements to sort, $A.length$, let us denote it with n.

Time complexity:

- Lines 5 - 7 each in unit time (independently of input size):
  - $O(\max\{1,1,1\}) = O(1)$
- Condition of line 4 in unit time (if elements are simple):
  - two cases: lines 5 - 7 are either executed, or not
  - in any case, lines 4 - 8 in unit time.
- Handling of iteration variable $O(1)$ (each round).
- Number of iterations of line 3 $(n-1)-(i+1)+1 = n-i-1 = O(n-i)$ rounds.
- Own time complexity of line 3 $O(1)$ (each round).
- Using product rule, lines 3 - 9 : $O((n-i)\times 1) = O(n-i)$
  - Note that we don't "know" how to simplify expression $n-i$!
- Outer iteration $n-1$ times, handling of iteration variable $O(1)$.
  - Repeated section 3-9 time complexity is $O(n-i)$ and i varies during the iteration!
- Resulting sum:

$$(n-1)+(n-2)+\cdots+(n-i)+\cdots+(n-(n-1)) \tag{1.14}$$

$$=\sum_{i=1}^{n-1}(n-i) = (n-1)\frac{(n-1)+1}{2} = \frac{1}{2}n^2 - \frac{1}{2}n = O(n^2) \tag{1.15}$$

- The whole algorithm time complexity is thus $O(n^2)$.

Space complexity

- Input: array to be sorted $O(n)$
- During sorting, we use integer variables i, j, tmp.
  - memory usage 1+1+1 words = $O(1)$
- Result: sorted array
  - no need for own space, sorting is done "in place"
- Extra space used $O(1)$.

⇒ Note: space complexity is analysed only on (extra) space used by the algorithm, because the input (and output) are needed anyway!

General instructions for calculating the execution time

- Calculate from inside out (if possible).
  - Sometimes it is necessary to examine the whole (what operations are carried out during the entire program/section execution).

- Assignments, input, and output operations are usually $O(1)$.
  - Exception: processing of large structures
  - Note the type of parameters
    - Array as a value parameter: copy $O(n)$
    - Array as variable/reference parameter: reference $O(1)$
- Valuation of an expression $O(1)$
  - Unless, naturally, if we call a subroutine!
- Array references $O(1)$
- Sequence of operations (= sequential instructions)
  - Using sum rule
- Conditional operations
  - $T(\text{valuation\_of\_condition}) + T(\text{chosen\_part})$ [usually worst case]
- Iteration
  - Using product rule
  - $\sum_i (T_{\text{iteration}_i})$
  - Ofter it suffices to calculate
    - num_of_iterations $\times T(\text{iterated\_section})$ [worst case].
    - If average case is obvious, we can use that as well.

⇒ If the worst case is vary rare, the situation must be analysed more carefully (calculate bad and good cases separately).

```
for (i = 0; i < n; i++)                                                    1
  if (i == n−1)                                                            2
    for (j = 0; j < n; j++)     // this is executed only once, O(n)       3
      a = a + 1;                                                           4
  else                                                                     5
    x = x + 1;          // this unit time operation is executed several times 6
```

- We may not ignore anything significant.

```
while (! find_coloring(k))                                                 1
  k = k + 1;                                                               2
```

- subroutine/method calls
  - $T(\text{parameter\_delivery}) + T(\text{execution\_of\_subroutine})$
- subroutines/methods
  - as a separate block
  - recursive subroutines with recurrences

Example 1.18. Factorial can be computed using following recursive algorithm:

```
public static int factorial(int n) {                                      1
    if (n <= 1)                                                            2
        return 1;                                                          3
    else                                                                   4
        return i * factorial(n−1);                                         5
}                                                                          6
```

- Lines 1-4 execute clearly in $O(1)$ time.
- On line 5, multiplication and assignment are obviously $O(1)$, but:

⇒ It takes time to evaluate the function call!

- Let $T(n)$ denote the execution time of the algorithm with input n:

$$T(n)=\begin{cases} d, \text{ when } n\leq 1 \\ c+T(n-1), \text{ when } n>1 \end{cases} \qquad (c,d \text{ are constants}) \tag{1.16}$$

- Let us take apart the above recurrence:

$$\begin{aligned} T(n)&=c+T(n-1) \\ T(n-1)&=c+T(n-2), \text{ ja} \\ T(n-2)&=c+T(n-3) \\ &\ldots, \text{ thus} \\ T(n-1)&=c+(c+T(n-3))=2c+T(n-3), \text{ thus} \\ T(n)&=c+T(n-1)=c+(2c+T(n-3))=3c+T(n-3) \end{aligned} \tag{1.17}$$

- By continuing deeper and deeper, we would notice that

$$T(n)=ic+T(n-i), \text{ when } i<n \tag{1.18}$$

- Let $i=n-1$ (go to the bottom of recursion):

$$\begin{aligned} &T(n)=c(n-1)+T(1)=c(n-1)+d \\ &\Rightarrow T(n)=O(n). \end{aligned} \tag{1.19}$$

We could prove the deduction using induction. We'll return to recursive time complexity on DSA II-course.

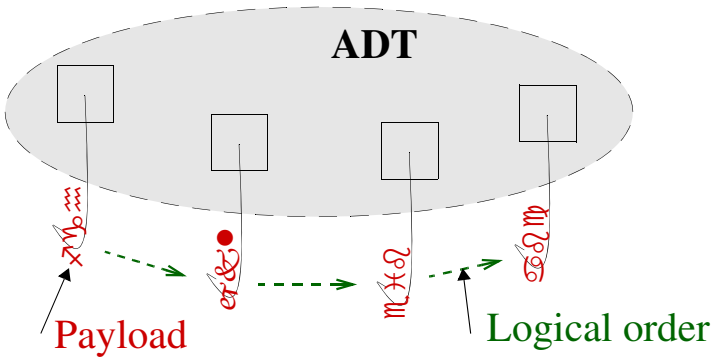
2 Abstract list structures

⇒ Let us start with simple, general purpose collection, list.

- List has several variants.
- We'll learn also general principles of Abstract Data Types.

The purpose of an abstract data type is to manage a collection of elements so that the collection can be used

- conveniently for the user (programmer)
- efficiently



Using and managing collections of elements requires possibilities to:

- add new elements to the collection
- find elements from the collection
- remove elements from the collection
- change elements of the collection
- going through (iterate over) the elements of the collection

⇒ Note, however, that we won't always need all the possibilities.

What a collection (e.g., a list) will store?

The purpose to be able to use the same ADT to manage collections of different element types.

- ADT must be work independently of the type of the elements.

⇒ We'll parametrize abstract data type with element type!

- In object-oriented languages (such as Java), the parameter is a class and a collection contains objects.
  - Objects are used by references, thus the collection contains references! compare: Figure 1.5 (p. 8).
  - Several collections can contain references to the same object.
  - When comparing object references, we must remember what to compare (in Java):
    - $X == Y$ compares equality of references X and Y (i.e., are they the same object). $X = Y$ assigns a reference.
    - $X.equals(Y)$ compares similarity of the objects.
    - Immutable elementary type objects (*Integer*, *String*, etc.), and optimizations (*Integer* 0..255) can confuse, please be careful.

Example 2.1. The interface of a list type could start with:

```
public class List<E> {
```

- Here E represents the type of the elements, the interface does not fix to type.
  - Formal type parameter, like formal parameter in subroutines.
- When we create lists, then the parameter type is fixed:

```
List<Integer> numberList = new List<Integer>();
List<List<Integer>> listList = new List<List<Integer>>();
List<Person> personList = new List<Person>();
```

If we are not interested in the type of the elements, we can create a "type-less" collection. Then the element type is *Object* (i.e., reference to an object of any class):

```
List objectList = new List();
```

In addition to Classes, using Java, we can type parametrize methods.

```
static <E> void doSomething(List<E> L) {
  E tmp = L.getFirst();
  ...
```

When writing a generic algorithm, we can restrict the possible element types if needed. E.g., require ability to compare to other (compatible) elements:

```
static void comparingAlgorithm(List<Comparable> L) { ...
static void comparingAlgorithm(List<? extends Comparable> L) { ...
```

Compiler will give us warning of this, however. We can get rid of the warnings by the correct form:

```
static <E extends Comparable<? super E>> void comparingAlgorithm(List<E> L) { ...
```

- *Comparable* is also a parametrized type, thus we need to give it a parameter that is compatible with E.

We'll return to this in the end of this course. For now, we settle for basic lists even if the compiler might give some warnings.

2.1 List

List as an abstract data type

Example 2.2. List can be represented, e.g., by naming the elements of the list sequentially. A five-element list L can be presented as:

$$L = a_1, a_2, a_3, a_4, a_5. \tag{2.1}$$

An empty list can then be presented as

$$L = . \tag{2.2}$$

The presentation of a list is chosen for each situation. An empty list can be presented also using a special symbols, such as λ or $\perp$.

The elements are mutually successive.

- For all elements, there is an unambiguous successor-predecessor -relation.
  - element a_2 precedes element a_3
  - element a_3 follows element a_2
  - The first element of a list does not have a predecessor.
  - The last element of a list does not have a follower.

⇒ When we manage elements of a list, we need to be able to somehow specify each element and its position in the list.

Position (list node)

- The distance of i^{th} element from the beginning of the list is $i-1$.
  - If we use an ordinary array of a programming language, then both distance from the beginning and position of element are unambiguous.
  - But: if/when we need to be able to insert elements to middle of list, then the distance (from beginning) of the successors of the new elements changes.

⇒ We need more abstract concept of position.

- Or accept change of position of existing elements!

Each element of a list has its own position in the list.

- We can refer to the first and last occupied position of the list (and/or the position after the last occupied position).
- The type and behaviour of the position depends on the implementation of the list.

Example 2.3. If we implement the list using a Java array, we shall obviously use the index within of an array as the position. Then, upon removing an element at position i, all the elements following position i will be moved to closer to the beginning of the array. Correspondingly, when inserting an element to position i, the element that was previously in position i, will be moved to position $i+1$ (and all the followers as well).

On the other hand, if we implement the list using a linked structure, we use usually use the address/reference of the element (node) as position. When removing an element at position i, that

position ceases to exists and other elements are just linked together, but their positions are not changed. Correspondingly, when inserting an element, we create a new position (memory address / object).

⇒ Thus: the behaviour of positions depends on the implementation (unfortunately).

java.util.LinkedList

- Interface mostly inherited from *AbstractCollection* and *AbstractList* -classes (as is, e.g., *Vector*/*ArrayList*:llakin), but time complexities of operations are different.
- Part of standard Java library, see JavaAPI documentation.
- http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html.
- Position (list node) is not available for user/programmer.
- Direct *get*(*index*), *add*(*index*, *element*), etc. operations take an index as a parameter distance from beginning of the list.
  - Time complexity $O(|L|)$ if/when they need to find the index!!
  - If not, then insert and remove are $O(1)$ time.
  - Exactly opposite to *Vector*/*ArrayList* -classes.

Example 2.4. Correct and wrong usages of linked list.

```
LinkedList<Integer> L = new LinkedList<Integer>();                          1
for (int i = 0; i < n; i++)                  // total O(n)                  2
    L.add(x);                                // insert to end O(1)          3
                                                                            4
for (int i = 0; i < n; i++)                  // total O(n)                  5
    L.add(0, x);                             // inset to beginning O(1)     6
                                                                            7
for (int i = 0; i < n; i++)                  // total O(n^2)                8
    L.add(i/2, x);                           // insert to middle (index search) O(n)   9
                                                                            10
for (int i = 0; i < n; i++)                   // total O(n^2)               11
    s = s + L.get(i);                         // index search O(n)          12
```

⇒ Iteration over list $O(n^2)$???

⇒ To efficiently iterate a list, we need to use an iterator!

- An iterator functions as a pointer/reference, as the position we are using later, but it "refers" (virtually) the "gaps" between elements (nodes) instead of elements (positions, nodes)!

Definition 2.5. *java.util.ListIterator* key operations.
(*LinkedList L*)

1. ListIterator i = L.listIterator()
   Creates and initializes a new iterator.

2. boolean i.hasNext()
   returns whether iteration i has a next element or not.

3. E i.next()
   moves iterator i to the next gap and returns the element it hopped over.

4. E i.previous()
   moves iterator to the previous gap and returns the element it hopped over. Use also (*i.hasPrevious*() to iterate backwards).

5. void i.add(x) inserts element x to the gap pointed by the iterator i. New element is returned by next *previous*() if it is called, not *next*().

6. void i.remove() removes the element previously hopped over using *next*() or *previous*().

Example 2.6. Iteration over the elements using iterator and a pre-test loop.

```
LinkedList L;                                                    1
...                                                              2
ListIterator i = L.listIterator();                               3
while (i.hasNext())                          // total O(|L|)     4
   ... = i.next();                                               5
```

- Collection can be changes during an interation only using *add*/*remove* -operations of this iteration!
- Two nested iterations cannot change a list.
- The is efficient and convenient for most simple use cases.
- *ListIterator* supports iterations backwards and makes possible to select starting point and changing an element (sa opposed to more restricted *Iterator*).

Example 2.7. Removing all occurences of element x out of list$L$ (x is not *null*).

```
ListIterator i = L.listIterator();                               1
while (i.hasNext())                          // O(N)             2
   if (x.equals(i.next()))                                       3
      i.remove();                                                4
```

foreach -loop

- If we are not going to change the list, and we go only to one direction, then the iterator can be hidden.
- We just need a reference to each element at a time.
- Java $\geq$1.5 has iteration "for each element do" -loop (*foreach*) for all collections, including lists:

```
for(E x : L)                                                     1
   x.foo();                                                      2
```

- This is implemented using an iterator.
- You may not change the collection during the iteration!
- Actually, it is enough that L implements interface *Iterable*, it doesn't have to be a collection.

Parametrizing an iterator

- *Iterator* and *ListIterator* are actually parametrized classes (*Iterator*<E>).
  - Especially *next*() and *previous*() return instances of class E (or *Object* if there we gave no parameter).
  - Thus, to make type checking possible and to avoid casts, it is a good practice to parametrize variables of type *Iterator* using a known type or by the type we got as a type parameter of a method or class.

```
static <E> void doSomething(LinkedList<E> L) {
    ListIterator<E> li = L.listIterator();
    while (li.hasNext()) {
        E x = li.next();
        ...
}   }
```

Restrictions of *java.util.LinkedList*

- Iterators are fine only for limited sequential iteration.
- Especially if we need to have several iterators, and we need to change the list.
- In complex cases, we may need to rely on indexes.
- *ListIterator* has more possibilities than *Iterator* (but does not overcome the concurrent modification limitation).
  - See, e.g., *purge*, *TraListMerge.java* vs. *JavaListMerge.java*.
- *next*() always progresses the iteration, it does not support "peeking" the "current" element without progressing.
  - If we do not progress on every step, we need to keep elements in separate variables and update them only when needed.
  - Termination of a loop must be handled differently.
  - We must ensure that also the last elements are used.
  - We must check for empty lists.

Abstract list operations (TRA-library)

- During this course, we present also a list which has positions (list nodes) available for the programmer.
- Internal implementation is the same as *java.util.LinkedList*, but the interface is different.
- `http://cs.uef.fi/pages/sjuva/dsaI.html#kirjasto`

Definition 2.8. Position-based list operations. A position is a reference to list node (*ListNode*). (Parameter types: *E x*, *ListNode p*, *TraLinkedList L*)

1. TraLinkedList<E> TraLinkedList<E>()
   Creates a new empty list. Element type E can be any class.

2. ListNode L.first()
   Returns the position of the first element of list L. If L is an empty list, returns position $L.EOL$.

3. ListNode L.last()
   Returns the position of the last element of list L. If L is an empty list, returns position $L.EOL$.

4. void L.insert(p, x)
   Inserts element x to list L in front of position p. If $p = L.EOL$, adds to the end of the list. If position p does not exists in L, the result is undefined (programming error).

5. void L.remove(p)
   Removes from list L the element in position p (and thus also the position). If position p does not exists in L, or $p = L.EOL$, the result is undefined (programming error).

6. ListNode p.next()
   Returns the position following position p in the list. If p is the last position of the list, returns position $L.EOL$. If $p = L.EOL$, or list does not have position$p$, is result undefined.

7. ListNode p.previous()
   Returns the position preceding position p in the list. If p is the first position of the list, or list does not have position$p$, is result undefined.

8. E p.getElement()
   Returns the element in position p. If $p = L.EOL$ is result undefined.

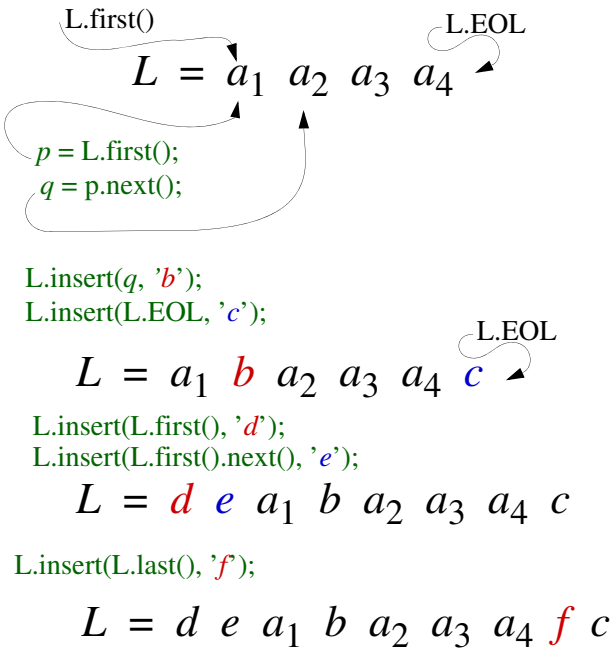


Figure 2.1: List operations

Example 2.9. Let L be an unsorted list of unknown element type. Task is to make an algorithm to remove from list L all duplicates of the same element.

```
public static void purge(TraLinkedList L) {                    1
   ListNode p = L.first();                                      2
   while (p != L.EOL) {                                         3
      ListNode q = p.next();                                    4
      while (q != L.EOL) {                                      5
         if (q.getElement().equals(p.getElement())) {           6
            ListNode r = q.next();                              7
            L.remove(q);                                        8
            q = r;                                              9
         } else                                                 10
            q = q.next();                                       11
      }                                                         12
      p = p.next();                                             13
}  }                                                           14
```

Time complexity of list operations

- *first*, *next*, *getElement* and list creation should always be $O(1)$ (unit time) to facilitate efficient iteration.

- *insert*, *remove*, *previous*, *last* and *EOL* can be made (easily) in unit time, thus if/when those are needed, they should also be unit time.

Example 2.10. To analyse algorithm of example 2.9, we need to know list time of the list operations and element comparison.

Let us assume that comparison *equals*() is $O(1)$ and that list operations used in 2.9 are also $O(1)$. Then lines 6-11 are $O(1)$ time. Both iterations go through the list element by element. Inner iteration will be shorter and shorter after each step of outer iteration as the beginning of the list can be skipped, and/or the list becomes shorter when we encounter duplicate elements. In the works case there are not duplicate elements, thus the outer iterations is executed $O(\text{list\_length})$ times and the time complexity of the whole iteration is $O(\text{list\_length}^2)$. If *equals*() is not unit time, then we can use form $O(T_{equals} \times \text{list\_length}^2)$.

Example 2.11. Comparing the equality of two lists. Lists are equal if they are of equal length and they have the same elements in the same order.

```
public static boolean compareLists(TraLinkedList L1, TraLinkedList L2) {
    ListNode p1 = L1.first();
    ListNode p2 = L2.first();
    while ((p1 != L1.EOL) && (p2 != L2.EOL)) {
        if (! p1.getElement().equals(p2.getElement()))
            return false;
        p1 = p1.next();
        p2 = p2.next();
    }

    if (p1 == L1.EOL && p2 == L2.EOL)
        return true;
    else
        return false;
}
```

Time complexity?

Applications of (linked) list

- Anywhere where the collection is used (iterated) in sequential order.
- Especially of the elements of the collection have some order.
- Especially of we do not know the size of the collection in advance.
- Especially if we do insert or remove operations to the middle of the collection.
  - (which are slow for an array!)
- E.g., "compute the sum of elements of the collection".
  - Sequential structure is ok.
- E.g.,: "find the name of a given phone number".
  - Sequential structure is slow for finding.
  - List is not good (unless such operations are very rare).

List is a general purpose data structure.

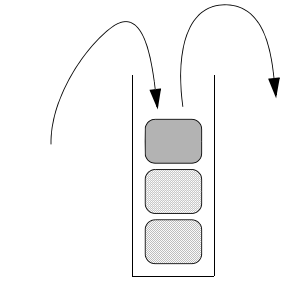
⇒ We do not always need all the functionality of list.

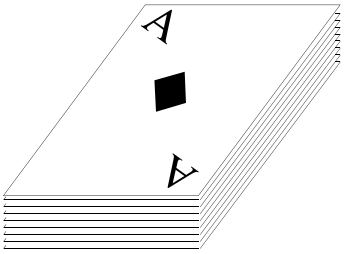
- Especially as an auxiliary data structure to temporarily store data during an algorithm.
- In such cases, it is useful to limit functionality (operations) for the given task, most often by limiting the access.
- Benefit: we'll have much more simpler operations.
  - Easier to read program.
  - Less sources of error.
  - Easier to implement.
  - More unambiguous.

Following sections present such restricted forms of list.

2.2 Stack

- Addition and removal operations are allowed only on one end of the collection.
  - Active end == top of the stack.
  - The other end == bottom of the stack.
  - Out of all elements in the stack only topmost element is available for removal.
  - Out of all elements in the stack only the topmost element is visible.
    - We might not even know the number of elements in the stack (except empty – non-empty), unless we count it separately.
- Last inserted element is the first to be removed: (LIFO, Last In First Out).





- As we restrict the access, not all list operations are meaningful.

⇒ *insert*, *getElement*, *remove* and creation of stack are enough.

- In these operations, the point of action is always obvious.
  - We won't need position concept at all, thus we won't need the position parameters for operations.
- Additionally we need an operation to check if the stack is empty or not.
- The help readability, we also change the names of the operations.

Definition 2.12. Classic naming of stack operations: (*LinkedStack S*, *E x*)

1. LinkedStack<E> LinkedStack<E>()
   creates an empty stack.

2. void S.push(x)
   adds an element x to the top of the stack. [insert]

3. E S.peek()
   return the topmost element of stack S If stack is empty, gives an exception. [getElement]

4. E S.pop()
   removes and returns the topmost element of stack S. If stack is empty, gives an exception. [remove]

5. boolean S.isEmpty()
   returns boolean value *true* if stack S has no elements, otherwise returns *false*.

⇒ All operations should be $O(1)$.

- In a low-level implementation only 1-3 clock cycles.
- in a stack machine only 0-1 clock cycles.

Example 2.13. Checking of brackets:

- Input: string of characters, including brackets ( ) [ ] { } .
- We must check if brackets are in correct order.
  - Corresponding brackets of the same type (not ”( ]”).
  - Brackets of different type are not mixed (not ”([ )]”).
  - No closing bracket before opening bracket (not ”) (”).
  - All brackets are closed before the end of the string (not ”([ ]”).
- Full code at course www-page.

```
public static boolean bracketPairs(String m) {
   LinkedStack<String> S = new LinkedStack<String>();
   for (int i = 0; i < m.length(); i++) {
      char c = m.charAt(i);
      switch (c) {
         case '(' : S.push(")"); break;
         case '[' : S.push("]"); break;
         case '{' : S.push("}"); break;
         case ')' :
         case ']' :
         case '}' :  if (! m.substring(i, i+1).equals(S.pop()))
                        return false;
      }
   }
   return S.isEmpty();
}
```

If string and stack operations are efficient, the time complexity is O(string_length) and space complexity O(bracket_depth).

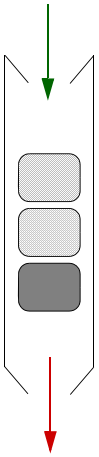
java.util.LinkedList, *java.util.Vector*, jne.

- *LinkedList* used from start or end, *Vector*/*ArrayList* used from the end (or *Stack*, an extension of *Vector*).
- *add*(), *getLast*(), *removeLast*(), *isEmpty*().

Open stack

- An open stack is like stack, but in addition to topmost element, we can ”dig” next elements using operation
  - S.top(k)
    Returns k^{th} element from top of open stack S.
    - Time complexity is $O(k)$ or $O(1)$, depending on implementation.
- Note, that still only the topmost element can be removed or changed (unless we specifically define and implement it so).

2.3 Queue



- Adding is allowed only on one end of the collection, and removal is allowed only on the other end.
  - Adding end == tail of the queue.
  - Removal end == head of the queue.
  - Only the head element can be removed.
  - Only the head element can be inspected.
    - We might not even know the number of elements in the queue (except empty – non-empty), unless we count it separately.
- Last inserted element is the last to be removed: (LILO, Last In Last Out).

- As we restrict the access, not all list operations are meaningful.
  - *insert*, *getElement*, *remove* and creation of queue are enough.
    - In these operations, the point of action is always obvious.
      - We won't need position concept at all, thus we won't need the position parameters for operations.
    - Additionally we need an operation to check if the queue is empty or not.
    - The help readability, we also change the names of the operations.

Example 2.14. Cashier queue in a shop is a typical queue: new persons always go the the end of the line. Persons are served in the order of appearance.

Definition 2.15. Queue operations of the TRA-library (traditional naming on brackets): (*LinkedQueue* Q, $E\ x$)

1. LinkedQueue<E> LinkedQueue<E>()
   creates an empty queue Q.

2. void Q.offer(x)
   adds element x to the end of queue Q. [enqueue]

3. E Q.peek()
   returns the first element of queue Q. If queue is empty, gives an exception. [front]

4. E Q.poll()
   removes and returns the first element of queue Q. If queue is empty, gives an exception. [dequeue]

5. boolean Q.isEmpty()
   returns boolean value *true* if queue Q has no elements, otherwise returns *false*.

$\Rightarrow$ All operations should be $O(1)$.

$\Rightarrow$ *java.util.LinkedList* works well as a queue.

$\Rightarrow$ *ArrayList*/*Vector* do not work as such (insert or remove is $O(n)$)!

- Queue can be implemented in an array efficiently using so called array buffer technique, see "Queue and deque in an array" page 81.
- In Java API, implementations of *java.util.Queue* (*ArrayDeque*).
  - Read the introduction carefully!

Example 2.16. The contents of a stack can be reversed using a queue and v.v.:

```
public static void reverse(LinkedStack S) {
    LinkedQueue Q = new LinkedQueue();
    while (! S.isEmpty())
        Q.offer(S.pop());
    while (! Q.isEmpty())
        S.push(Q.poll());
}

public static void reverse(LinkedQueue Q) {
    LinkedStack S = new LinkedStack();
    while (! Q.isEmpty())
        S.push(Q.poll());
    while (! S.isEmpty())
        Q.offer(S.pop());
}
```

- Time and space complexity $O(|S|)$.
- Reversing a stack using another stack does not work, we need either a queue, or two stacks.

Example: Replacing recursion

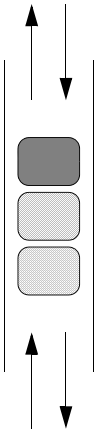
⇒ Recursive: An algorithm, that calls itself using part(s) of its input to simplify solving of the whole problem.

- Sometime recursion can be inefficient, or programming language does not support recursion.
- Sometimes the (best) order of problem solving cannot be achieved using recursion.
  - Recursion in programming languages is implemented using a stack, and thus work as LIFO.
- Very deep recursion uses a lot of memory (stack).
- One recursive call can be usually replaced with straightforward iteration, but maintaining several recursive branches of execution needs a data structure to maintain all new ”recursive calls’.

How to replace recursion?

- We maintain a stack or queue of undone subtasks.
  - In the beginning of the execution we put the initial task to stack/queue.
  - We repeat algorithm body until stack/queue is empty.
    - Instead of recursive calls, we push the (parameters defining) new subtasks to stack-/queue.
- The eventual order of handling of subtasks depend on whether we use a stack, queue, priority queue, or a set.
  - Queue: ”breadth first” iteration.
  - Stack: ”depth first” iteration.
  - Priority queue: ”best first” iteration.
  - Set: ”random” iteration.
- The same technique can be use in all cases where problem solving branches to several parts.
- We’ll return to this in tree and graph context (Chapter 3 and DSA II).

2.4 Deque



- Alternative spelling: ”dequeue”.
- Both insert and remove operation are allowed at both ends of the collection, but only at ends.
  - We need to be able to differentiate the ends:
    - top (front, first)
    - bottom (lowest, last)
  - A new element can be added to the top or the bottom of the deque.
  - Only topmost and bottom element can be removed and/or inspected.
  - We might not even know the number of elements in the deue (except empty – non-empty), unless we count it separately.
  - If deque has only one element, it is on both top and bottom.

- As we restrict the access, not all list operations are meaningful.
  - *insert*, *getElement*, *remove* and creation of deque are enough.
  - Additionally we need an operation to check if the queue is empty or not.
  - As opposed to stack and queue, now we need to be able to specify insert/removal point.
    - Full position, index, or list node system would be unnecessarily complex.
    - Top and bottom suffices.
    - We either have an extra binary parameter, or have different operations for top and bottom operations.
  - The help readability, we also change the names of the operations.

Example 2.17. Real world example of deque would be a deck of playing cards where we can take either topmost of bottom card. We can also insert a card to the top or bottom of the deck.

Definition 2.18. Deque operations (same as *java.util.Deque*, *java.util.LinkedList*): (*LinkedDeque* D, E x)

1. LinkedDeque<E> LinkedDeque<E>()
   creates an empty deque.

2. void D.addFirst(x)
   void D.addLast(x)
   inserts element x to the beginning / end of deque D.

3. E D.removeFirst()
   E D.removeLast()
   removes and returns the first / last element of deque D. If deque is empty, gives an exception.

4. E D.getFirst()
   E D.getLast()
   returns the first / last element of deque D. If deque is empty, gives an exception.

5. boolean D.isEmpty()
   return boolean value *true* if deque D is empty, otherwise returns *false*.

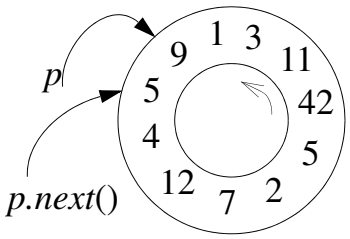
Sometimes operations are named as queue operations. Instead of distinct ... *First* and ... *Last* operations, we could have single operations and an extra parameter selecting the end of operation.

$\Rightarrow$ All operations should run in $O(1)$ time.

java.util.Deque

- http://docs.oracle.com/javase/7/docs/api/java/util/Deque.html
- Java API interface with deque operatoins.
- Both linked implementation and array implementation("ring buffer", see p. 81).
  - Note that *ArrayList*/*Vector* do not "implement" Deque as they would be very inefficient!
- Additionally special versions for concurrent programming.

2.5 Ring



- Sometimes the ends of sequential collection have no meaning, we just need to repeatedly iterate over elements of the collection.
  - Jumping from the *last* to the *first* can be delegated to the ADT.
- Inserts and possibly removes need to be convenient (and efficient).
  - Even insert position might be irrelevant if any order will do, then it suffices to have *R.next*() and *R.add*().
  - Ring can be one directional or bidirectional.
- Let
  - $R.last().next() = R.first()$ and $R.first().previous() = R.last()$,
- We get a ring, an endless list.
- Instead of *first* we can have *any* which returns some position.
- Instead of *EOL* we need *isEmpty* (as in deque).

⇒ We still need position to make iteration possible. It can be encapsulated within an iterator, though..

2.6 Array

⇒ List does not necessarily support changing of elements, neither referring an element by location (index) in unit time.

- Changing an element may require remove and insert.
- Using the elements in arbitrary order (random access) is very slow.

If we need both sequential iteration, and random access, we need (abstract) array.

- The elements of an array are in sequential order (≈list).
- The position of an element is specified using indexing (i.e., the (relative) distance from the beginning of the array).
  - Indexes are of integer type (or enumeration), we can even do simple calculations with them.

Array operations are about like list operations:

- Position is the distance from beginning of array.
- When creating an array, we should give initial capacity.
- Because insert into an array is very slow, the elementary operation is setting the element in a given index:
  - A.set(i, x)
- Sometimes we also need

  - A.force(i, x)
- which works as *set*(), but enlarges the array if needed to ensure existence of index *i*.
- To refer to boundaries of the array, we need:
  - A.first() and A.last()

Operations *set*(), *first*(), *last*(), *getElement*() and array creation take $O(1)$ time. *next*() and *previous*() only calculate indexes.

- If, however, array is initialized upon creation (as in Java), it will take linear time.

Operations *insert*(), *remove*() and *force*() take $O(\text{array\_size})$ time, or $O(\text{array\_suffix\_length})$.

- If insert or remove to/from middle is needed, array is not usually the best choice.

Open / fixed-size array

- In an open array insert and remove operations enlarge/shrink the array element by element. Force enlarges based on need.
- An array can be open either from both ends, or from one end only.
- In fixed array, *insert* and *remove* operations are usually missing.
  - If *insert* and *remove* operations are allowed in a fixed array, it is called a shelf.
    - In a shelf insert into middle will remove an element from the end. Remove from middle makes an empty position to end.

```
Object[ ]      A1;      // fully open      1
Object[1..]    A2;      // open at top     2
Object[..10]   A3;      // open at bottom  3
Object[1..10]  A4;      // fixed size      4
```

  - Note the need of operations *first* and *last*. In Java we cannot directly use an open from bottom array. The limitation can be circumvented using a mapping (*Map*, see p. 51) or using own indexing, e.g.:

```
A[abs(i)*2 + ((i < 0) ? 1 : 0)]   1
```

java.util.Vector

- Array implementation of ADT list.
- *java.util.ArrayList* is similar, but does not support concurrent access.
- Positions in use are indexed 0..*size*()–1.
- Implements list operations with array time complexities.
  - *get*(i) unit time.
  - *set*(i, x) unit time.
  - *add*(x) and *add*(i, x) are unit time to the end of list (as is *remove*).
    - Actually unit time in average.
  - *add*(i, x) to beginning or middle is linear time (as is *remove*).
- *Vector.firstIndex* == 0.
- *Vector.size*() tells the number of elements.
- *V.add*(*V.size*()+1, x) or *V.set*(*V.size*(), x) throws *ArrayIndexOutOfBoundsException*.
- *setSize*() sets the number of elements explicitly. If the size is more than previously, *null* elements will be added.
  - We use this when we need a sparse array.
  - Capacity can be added with *ensureCapacity*().

Capacity vs. number of elements.

- A Vector has initially some capacity, e.g., 10 slots.
  - For an empty vector, all these are space to expand.
  - Initial capacity can be given as parameter for constructor.
- When we add more and more elements, we may need additional capacity.
  - Vector initially allocates space for, e.g., 10 elements.
  - Enlarging space will take $O(size())$ time.
  - By default, when the space runs out, the capacity will be doubled.
    - How this impacts the time complexity?
  - Alternatively, the user can define the space increment as a constant (*capacityIncrement*).
    - This should not be used: adding n elements will take $O(n^2/c)$, time, where c is number of elements added each time (*capacityIncrement*).
- To avoid (repeated) additional space allocation, you can use operation *ensureCapacity*(), or give initial size for constructor if we know the final size, or if we can estimate it.

Arrays in programming languages

- All languages have some array type, most also multidimensional array.
- Some older languages do not support dynamic array size (Pascal, C83)
  - Array accoring to local parameter, or some other computed value.
  - Changing the size of an existing array is rarely supported.
- More general dynamic memory allocation operatations can be used to circumvent these limitations on those languages which have dynamic memory allocation on bytes.
- Besides create, assignment, and get, other operations have to be implemented manually, if those are needed.
  - We won't usually need (to implement) operations *insert*, *remove* and *force*. Use lists, trees, or hashes if those needed.
- In practice arrays are usually sufficient and especially efficient.
  - Thus, our data structure library does not contain separate array type.

2.7 Summary of list structures

- Except for array, all other types are suitable for sequential usage of elements.
  - Linked list must be used using iterators (or list nodes, or stack/queue/deque operations) – not by indexes.
  - Array structures are used with indexes – no inserts or removes elsewhere than the end.
- Usage of all these collections is easy and efficient when used correctly.

⇒ To search efficiently from a changing collection, we need better structures.

- "changing" : not array
- "searching" : not list

Example 2.19. List types (except array) are very close relatives. On top of general list, we can implement the others easily.

Following presents stack operations using list operations. From the operations, we could completely remove inherited constructor and *isEmpty*. Also *super* references to superclass are unnecessary but will probably help understanding the code.

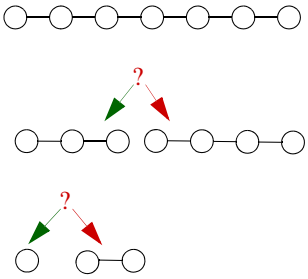
```
public class Stack<E> extends java.util.LinkedList<E> {                    1
```

```
    public Stack()                                          2
        super();                                            3
    }                                                       4
    public void push(E x) {                                 5
        super.addFirst(x);                                  6
    }                                                       7
    public E pop() {                                        8
        return super.removeFirst();                         9
    }                                                       10
    public E top() {                                        11
        return super.getFirst();                            12
    }                                                       13
    public boolean isEmpty()                                14
        return super.isEmpty();                             15
    }                                                       16
}                                                           17
```

3 Trees

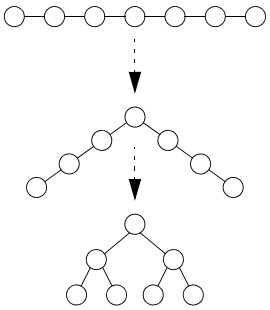
The problem of list as a collection:

- List is long, thus searching an element is slow.



⇒ Idea: split the "list to half"!

- I might not be that easy, but let us assume it initially.
- First we find out which half has the element, and search from only there → only half the elements to go through.
- Let us split even these halves into two → now only a quarter of elements to go through.
- If we repeat this log$n$ times, we only make these log$n$ decisions, no sequential iteration at all.
- This is actually binary search (which works on sorted array, not on list).
  - But: inserting to a sorted array is slow!



How this can be implemented as a collection?

⇒ Make "list" branched:

⇒ Instead of predecessor – follower system, we shall have predecessor – 2 followers.

⇒ A tree gives a hierarchical order for the elements of the collection (compared to sequential order in list).

- Tree can be used not only for searching, but also presenting hierarchical data.
- In a hierarchy, we must be able to present one predecessor, many followers.
- We shall make a distinction to two cases: two followers (binary tree) and zero or more followers (general tree)
  - Binary tree is a good search structure.
  - General tree is good to present hierarchical data.

3.1 Elementary tree concepts

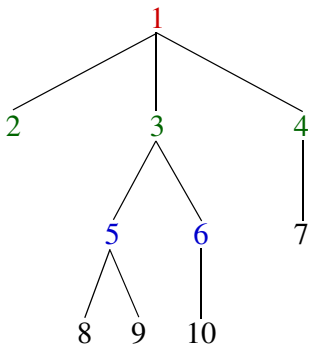
- A tree consists of (tree) nodes which are organized hierarchically.
  - Parent relation dictates the hierarchical relation.
    - ”Father” or ”mother” would actually be even better as ”parent” might not be unambiguous.
  - A non-empty tree has exactly one node, so called root, that does not have a parent.
  - All other nodes of a tree have an unambiguous parent.
  - Tree can be also defined recursively:

Definition 3.1. A single node forms a tree. Is is also the root of the tree.

If n is a node and T_1,T_2,...,T_k are tree with roots n_1,n_2,...,n_k, respectively, we can form a new tree by setting node n as the parent of nodes n_1,n_2,...,n_k. The root of the new tree is n and T_1,T_2,...,T_k are sub-trees of n. Nodes n_1,n_2,...,n_k are called children of node n.

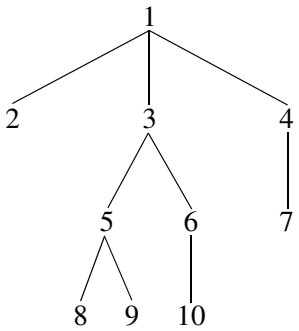
In an empty tree there are no nodes, not even a root. We can mark an empty tree with mark Λ (*null*). Sometimes an empty tree is considered the same as an empty node, which can be also marked by Λ or $\perp$ sign.

Example 3.2. Given the attached tree, root is node 1. Sub-trees of the root are 2, 3 ja 4. Sub-trees of node 3 have roots 5 and 6 and so on. Node 5 is the parent of node 9 and node 9 is a child of node 5. Node 9 has no child, thus it is no parent of any node.
As the nodes of lists, also nodes of a tree can contain element of any type. In examples we mostly use integers or letters. For clarity, we usually draw only the element. Remember, however, that there are separately a tree node and the element.



Path

- Sequence of tree nodes n_1,n_2,...,n_k is a path, if node n_i is parent of node n_{i+1} for all $i=1,...,k-1$.
  - The length of the path is $k-1$, i.e., number of edges (parent-child relations).
- If needed, we can define a path also from bottom to the top.



Leaf node

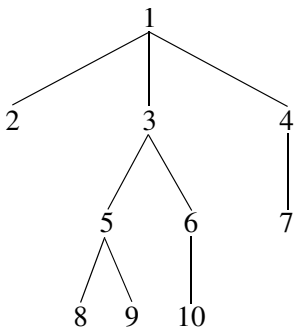
- A tree node with no children.

Branching node

- A tree node with at least one child.

⇒ Every tree node is either a leaf node or a branching node.

Height



- Height of a node is the length of the longest path from a leaf node to the node in question.
- Height of a tree is the height of the root node.

Depth

- The depth of a node is the length of the path from the root the the node in question.

Example 3.3. For attached tree, height of node 3 is two, height of node 4 is one, and height of node 2 is zero. The height of the whole tree is three. Depth of node 6 is two.

Siblings

- Children of a single tree node are called siblings.
- Siblings are usually ordered from left to right (an ordered tree).
  - In most applications, the order of the siblings is significant.
  - If we do not fix the order, we have an unordered tree.
- In a general tree, we often call the leftmost sibling as oldest, respectively the rightmost one can be called the youngest.
- If the nodes have no common parent they are not siblings.
  - If the nodes have a common grandparent, we can use term cousins.
- In a general tree, the unlimited set of children can be presented either as a list of children or by presenting the oldest child of each node and the next sibling of each node.
  - List of children may be easier to understand, but using child-sibling -system, the interface and implementation are simpler.

Ordering of nodes of trees

Figure 3.1: Two trees having the same elements in different order (but having the same structure).

- Nodes of a tree can be processed or iterated in several different orders, depending the need of the application.
- Most used orders are pre-order, in-order, post-order, and order by levels.
- We'll later on DSA II course use the same ordering also with graphs.

Definition 3.4. Pre-order, in-order, post-order, and order by levels.

a) For an empty tree, all orderings are empty.

b) For a tree with one node only, the orderings contain the single node.

c) If the root of the tree is n and the sub-trees of the root are T_1,T_2,...,T_k, (from left to right), then

i) pre-order of the nodes is:

1. n,
2. nodes of T_1 in pre-order,
3. nodes of T_2 in pre-order,
4. ...,
5. nodes of T_k in pre-order;

ii) in-order of the nodes is: (especially for binary trees where each node has two children (or empty children)):

1. nodes of T_1 in in-order,
2. n,
3. nodes of T_2 in in-order,
4. (...,)
5. (nodes of T_k in in-order;)

iii) post-order of the nodes is:

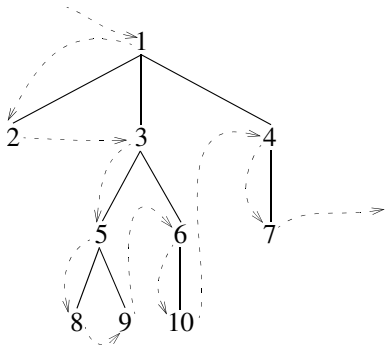
1. nodes of T_1 in post-order,
2. nodes of T_2 in post-order,
3. ...,
4. nodes of T_k in post-order,
5. n.

iv) Order by levels is:

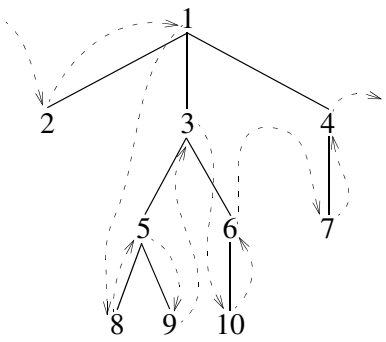
1. n,
2. nodes in depth 1 (children of n) from left to right,
3. nodes in depth 2 (grand children of n) from left to right,
4. ...,
5. deepest nodes of the tree from left to right,

Example 3.5. For the attached tree the nodes are in

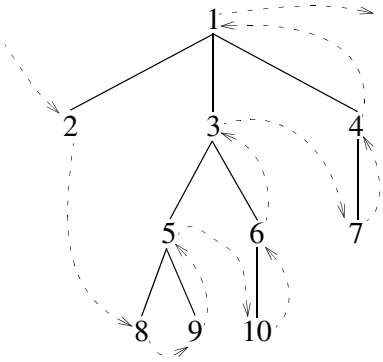
1. pre-order 1, 2, 3, 5, 8, 9, 6, 10, 4, 7;



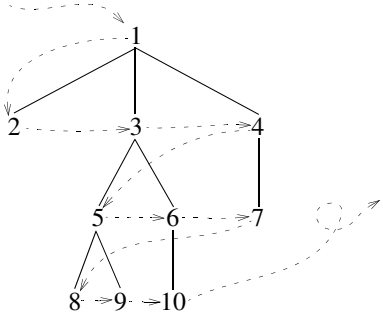
2. in-order 2, 1, 8, 5, 9, 3, 10, 6, 7, 4;



3. post-order 2, 8, 9, 5, 10, 6, 3, 7, 4, 1;



4. ordered by levels 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.



Contents of a tree

⇒ As in other collections, we want to use tree to process and store application data.

- Often the element in a tree node is called a label.
- In out data structure library, a tree is parametrized as a list.
- A tree includes tree nodes, each tree node maintains a reference to application data (subclass of *Object*), exactly as in *TraLinkedList*.
- Tree and tree nodes are type parametrized, parameter type of both (E) must match.

Example 3.6. Tree in Figure 3.2 represents arithmetic expression $(a+b)*(a+c)$. The tree has seven nodes. The figure only shows the contents of each node. Thus, we can combine the tree nodes and contents in drawings.

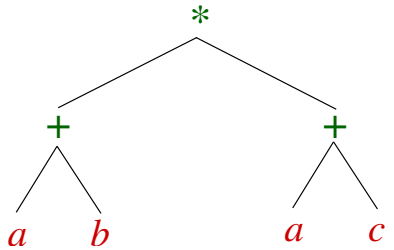


Figure 3.2: Parsing tree of an arithmetic expression.

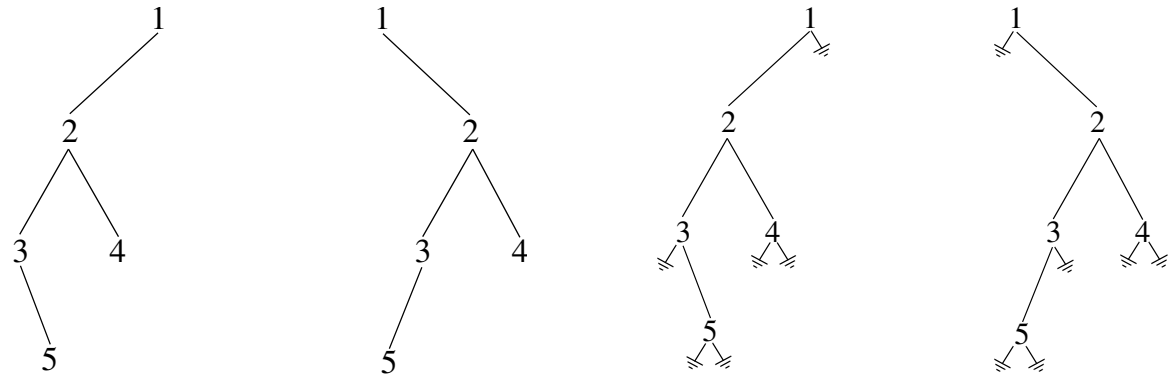
1. Each leaf node contains an operand.
2. Each branching node contains an operator so that the left sub-tree of a branching node represents the left operand and respectively the right sub-tree represents the right operand.

Example 3.7. The example expression can be read in pre-order $*+ab+ac$, in in-order $a+b*a+c$, and in post-order $ab+ac+*$. Of these forms, pre- and post-order are unambiguous. In in-order form we need brackets to show order of calculation.

3.2 Binary tree

- A node of a general tree can have arbitrary number of children.
  - This makes comprehension, usage, finding, and implementation more difficult.
  - Time complexities are often $O(\text{height} \times \text{number\_of\_siblings})$.
- Let us limit the number of children to at most/exactly two.
  - Each node has/has not a left child.
  - Each node has/has not a right child.
  - If a node has only one child, it is still either left or right (and the other "child" is "empty").
- Benefits:
  - Implementation is straightforward, direct reference to both children and (usually) to the single parent.
  - Advancing to deeper in a tree based on only single comparison directly to either child.

Example 3.8. Binary trees:



(a) Empty children not drawn.

(b) Empty children visible.

Figure 3.3: Two different 5-node binary trees.

- In Figure 3.3a the empty children of nodes are not visible.
- In Figure 3.3b the empty children are visible (⏚). This figure may clarify the idea, but is not convenient in the long run.

- In most algorithms is useful to take into account the possibility that the subroutine is called with an empty node.
  - If suffices to make the *null* check in one point only.
- In we read the nodes of trees of Figure 3.3, they are in pre-order:
  - 1, 2, 3, 4, 5
- But in in-order the nodes are
  - for left tree:
    - 3, 5, 2, 4, 1
  - whereas for the right tree:

    * 1, 5, 3, 2, 4
- Correspondingly, it is possible to find different trees for which the nodes are the same in in-order, but different in pre-order.

Binary tree operations

Definition 3.9. Usually we need following tree operations: (*BTree* T, *BTreeNode* n, m, E x)

1. BTree<E> BTree<E>()
   creates an empty binary tree.
2. BTreeNode<E> BTreeNode<E>(x)
   creates a new unconnected binary tree node with element x.
3. BTreeNode T.getRoot()
   returns root node of binary tree T. If T is an empty tree, returns empty node (*null*).
4. BTreeNode T.setRoot(n)
   Sets node n as the root of binary tree T. The possible previous nodes of the tree are lost.
5. BTreeNode n.getParent()
   returns the parent of node n. If n is root node, returns empty node (*null*).
6. BTreeNode n.getLeftChild()
   returns the left child of node n, or empty node (*null*) if node n has no left child.
7. BTreeNode n.getRightChild()
   returns the right child of node n, or empty node (*null*) if node n has no right child.
8. E n.getElement()
   returns the element maintained by node n.
9. void n.setLeftChild(m)
   sets node m as the left child of node n. The previous left child (and its descendants) of node n is lost.
10. void n.setRightChild(m)
    sets node m as the right child of node n. The previous right child (and its descendants) of node n is lost.
11. void T.killNode(n)
    destroys node n (and its descendants) from tree T.

$\Rightarrow$ Time complexity of all tree operations should be $O(1)$.

Search trees

- Why in-ordered binary tree is so good?

$\Rightarrow$ Search, insert, and remove in time: $O(\text{tree\_height})$

- Tree height = $O(n)$ (trivial, not very useful)
- Tree height = $\Omega(\log n)$

- Can we ensure that tree height = $\Theta(\log n)$?
  - With small extra tuning yes, as we'll see later (Subsection 8.5, p. 91)!
  - Search, insert, remove: $O(\log n)$, i.e., very quick even if we had huge input.

Definition 3.10. Search tree. We maintain an in-ordered binary tree, i.e., all elements of nodes of the left sub-tree of a node n precede the element in the node n and all elements of nodes of the right sub-tree of a node n follow the element in the node n.

Search: if the element in question is not found in this node, then if the element precedes the element in this node, search from left sub-tree (child), otherwise search from right sub-tree. [*BinTreeExample.java*]

```
public static boolean inorderMember(BTree T, Comparable x) {
   BTreeNode n = T.getRoot();
   while (n != null) {
      if (x.compareTo(n.getElement()) == 0)
         return true;
      else if (x.compareTo(n.getElement()) < 0)
         n = n.getLeftChild();
      else
         n = n.getRightChild();
   }
   return false;
}
```

Time complexity: on each level: 2 comparisons and following a reference ($O(1)$). Total time O(tree_height). Tree height can be $\log n..n$, where n = number of nodes on the tree.

There can be different binary trees containing the same elements!

- Gives flexibility for insert.
- We always make insert as a leaf node (replace an empty node by a new childless node).
- Because elements are stored in tree in-order and because new element has a dedicated position in the order, the point of insert is always unambiguous.
  - The structure of tree is decided based in order of insert operations.
- To prevent the tree to grow too high, we can straighten it when necessary (but still maintaining the in-order) (p. 91).

Example 3.11. Insert to an in-ordered binary tree so that the in-order is maintained: Make search in the tree until we meet an empty node where the new element would be (if it would be in the tree). Add the new element with a new tree node in place of the found empty node (it will be either left or right child of its parent). [Exercise]

Example 3.12. Successor in in-order of a given node n. If node n has a right child, the successor is the leftmost descendant of the right child. Otherwise, the successor is the closest ancestor in whose left sub-tree the node n is. [Exercise]

Example 3.13. Predecessor in in-order of a given node n. If node n has a left child, the follower is the rightmost descendant of the left child. Otherwise, the follower is the closest ancestor in whose right sub-tree the node n is. [Exercise]

Example 3.14. Remove from an in-ordered binary tree. Removal is easy if the node is a leaf node. Otherwise we need to replace the element in the node by the element of the predecessor node (or successor node) and remove the predecessor node. [*BinTreeExample.java*]

3.3 General tree

Binary tree is a good search structure, but we cannot use it to represent general hierarchy if/when we need more than two followers (children) for a node.

- In a general tree the unlimited set of children can be expressed either by having a list of children for each node, or by storing the oldest child and the next sibling of each node.
- List of children may be more clear as an idea, but using child-sibling system, the interface and implementation are simpler, thus here we use it.

Definition 3.15. Usually we need following tree operations: (*Tree* T; *TreeNode* n, m; E x)

1. Tree<E> Tree<E>()
   Creates an empty tree.

2. TreeNode<E> TreeNode<E>(x)
   Creates a new unconnected tree node with element x.

3. TreeNode T.getRoot()
   Returns the root node of tree T. If T is an empty tree, returns empty node (*null*).

4. TreeNode T.setRoot(n)
   Assigns node n as the new root of tree T. Any previous nodes of the tree are lost.

5. TreeNode n.getParent()
   Returns the parent of node n. If n is root node, returns empty node (*null*).

6. TreeNode n.getLeftChild()
   Returns the oldest (leftmost) child of node n. If n is a leaf node, returns empty node (*null*).

7. TreeNode n.getRightSibling()
   Returns the next younger (right) sibling of node n. If n is root or the youngest (rightmost) child of its parent, returns empty node (*null*).

8. E n.getElement()
   Returns the element (label, application data) of node n.

9. void n.setLeftChild(m)
   Assigns node m as the leftmost child of node n. Previous leftmost child of node n (and its siblings) are lost.

10. void n.setRightSibling(m)
    Assigns node m as the next right sibling of node n. Previous right sibling of node n is lost.

11. void T.killNode(n)
    Removes node n from tree T and all its children and right siblings.

⇒ All tree operations should run in $O(1)$ time.

Example 3.16. Recursive traversal (print elements) of tree in pre-order.

```
public static void preorderPrint(Tree T) {                                    1
    if (T.getRoot() != null)                                                   2
        preorderPrintBranch(T.getRoot());                                      3
    System.out.println();                                                      4
}                                                                              5
public static void preorderPrintBranch(TreeNode n) {                          6
    System.out.print(n.getElement() + "␣");     // actual work here            7
    TreeNode child = n.getLeftChild();                                         8
    while (child != null) {                                                    9
        preorderPrintBranch(child);                                            10
        child = child.getRightSibling();                                       11
}   }                                                                          12
```

Method *preorderPrint* is only a wrapper that calls the main recursive traversal from the root of the tree. On line 7 we can do other operations instead of printing.

Building trees

⇒ Building depends on application structure (the payload)).

- *set{Root | LeftChild | RightSibling}* and *new TreeNode*(x) are suitable for building tree node by node (which is how we usually add data).
- If we need to build the whole tree at once (e.g., we receive a set of elements, of which we create a new collection (or ad to an existing)):
  - Start from root and progress depth-first downwards..
    - Then we won't need to store loose branches in temporary collections,
    - and we can avoid repeatedly searching the point of insert from the root.
  - If we build from bottom-up, we have to store branches in a queue or deque (or recursion).
- Sometimes we can use operation *construct*(n,T_1,T_2,...,T_k)
  - which creates a tree from pieces at once.
  - Changing or reorganizing the nodes requires other operations.

About trees and nodes

Node ?=? Tree

- Could we refer to nodes and tree interchangeably?
- E.g., "Search from tree T" would work even if T would be arbitrary sub-tree (node).
- One parameter would be enough for operations.
- Especially when building a tree it would be convenient if we could add a whole previously built tree as a branch of another tree.
- This is possible if we define tree and node classes and the operations correctly. In our TRA library, this has been implemented for *set* operations. Writing full algorithms interchangeably is not possible. The limitation was losing compile time error checks.
- We'll return to this later when discussing implementation of trees.

Example 3.17. Path the tree node containing element x. Algorithm finds a path in unordered tree T from root to node which contains element x. Algorithm also prints the nodes in the path. Algorithm does exhaustive search in the tree. During the search it maintains a deque which contains the path from root the current node. See *TreeExample.searchPath*().

Example 3.18. Search element x from pre-ordered tree. Start at root, progress to leftmost child or right sibling until we find the element or we know that the element is not in the tree, or we

hit an empty node. If the element x precedes than the element in current node, then x cannot be found from tree. If x precedes element of right sibling of current node, then we proceed to left child, otherwise proceed to the right sibling. Exercise.

Example 3.19. Traversal by level (breadth first). Add root to a queue. Repeat until queue is empty. Take a node from queue, process the node, add all children to queue.

```
public static void printByLevel(Tree T) {
    LinkedQueue<TreeNode> Q = new LinkedQueue<TreeNode>();
    if (T.getRoot() != null)
        Q.offer(T.getRoot());
    while (! Q.isEmpty()) {
        TreeNode n = Q.poll();
        System.out.print(n.getElement() + "␣");     // or something useful
        n = n.getLeftChild();
        while (n != null) {
            Q.offer(n);
            n = n.getRightSibling();
        }
    }
    System.out.println();
}
```

Other trees

- Besides to constant 2, we can restrict the number of children to any other constant.
- One convenient is the size of the alphabet:
  - In each tree node there is an array of children, indexed, e.g., `a..z`.
  - The actual payload element is stored (as a reference) on depth *key_length* in the tree.
  - Search always in time O(key_length)
  - Each key has unambiguous location in the tree.
  - If there are a lot of keys (compared the length of keys), we won't waste that much space.
- B-tree (do not confuse binary tree and B-tree).
  - Ordered tree structure especially suitable for mass storage.
  - Goal is wide, very shallow tree (only 2-3 levels on mass storage).
  - Each tree node has $t..2t-1$ elements (keys) and each node has one more children than keys. t optimized based on storage properties (performance), up to tens of thousands.
  - A node is load from mass storage to main memory. Binary search within the node.
  - A copy of root node maintained in main memory.
  - Introduced in more detail in course DSA II.

4 Sets

⇒ Lists and trees are fairly concrete collections – we specify the location of each element.

A set provides more abstract interface.

⇒ Set is only a "black box" containing the elements.

One of most used collection:

- databases (a tuple is in relational database or not)
- set of nodes
- set of keywords and identifiers
- etc.

4.1 Set concepts

A set is a collection of elements of arbitrary type.

- We do not restrict the type of elements (as we haven't before).

Elements of a set are different elements.

- Elements are of same type (at least compatible)
  - One empty element (*null*) may be allowed (depending application and/or implementation).
- The same element may never be in a set twice.
- Two elements that are considered equal in the application context may not be in the set twice.
  - Equality must be unambiguous.
  - The user/programmer can (must) define (tailor) the equality function for the application context.

Often (but not always) we assume the elements of a collection to have a linear order ("ordered set", "sorted set"):

1. If a and b are elements of set S, then only one of relations

   (a) $a < b$
   (b) $a == b$
   (c) $b < a$

   is true. (i.e. the order is unambiguous)

2. If a, b and c are elements of set S so that $a < b$ and $b < c$, then $a < c$. (order is transitive).

It is easy to define ordering for most simple types.

- For complex types, e.g., sets, the ordering might not be obvious.
- E.g., order of sets of integers:
  - smallest/largest element? (draws?)
  - average | sum | median of elements?
  - number of elements?
- Ordering can be artificial if the application does not need it.

Set operations

- $\emptyset$: empty set
- $=$: equal (set)
- $\in$: member of a set
- $\subseteq$: subset

- $\subset$: proper (strict) subset
- $\cup$: union
- $\cap$: intersection
- $\setminus$: difference

- creating a set
- copy of a set
- insert element
- remove element
- traversal of elements

Definition 4.1. Operations of abstract set (in course data structure library): (*Set* A, B; *E* x)

1. Set<E> Set()
   creates a new empty set.

2. Set<E> Set(java.util.Collection<? extends E> C)
   creates a new set and copies there all the elements (references) of collection C.

3. boolean A.isEmpty()
   returns boolean value *true*, if A has no elements ($A == \emptyset$), otherwise returns *false*.

4. boolean A.equals(B)
   returns boolean value *true*, if sets A and B contain the same elements, otherwise returns *false*.

5. Set A.union(B)
   returns a new set containing elements of both set A and set B ($A \cup B$).

6. Set A.intersection(B)
   returns a new set containing those elements that are both in set A and in set B ($A \cap B$).

7. Set A.difference(B)
   returns a new set containing those elements of set A that are not in set B ($A \setminus B$).

8. boolean A.contains(x)
   returns boolean value *true*, if set A contains element x ($x \in A$), otherwise returns *false*.

9. boolean A.add(x)
   inserts element x to set A. Returns *true* if insert succeeded. If x was already in set A, then no insert is done and operation returns *false*.

10. boolean A.remove(x)
    removes element x from set A. Returns *true* if element was found and removed, *false* if element was not found.

11. Set A.clone()
    returns a full copy of set A. New set contains the same elements (references) as A, but the elements are not copied.

12. E A.first()
    returns the smallest element of set A. Result is undefined if $A == \emptyset$ or if elements of set A have no linear ordering.

13. E A.last()
    returns the largest element of set A. Result is undefined if $A == \emptyset$ or if elements of set A have no linear ordering.

We'll return to time complexities of operations later.

⇒ For most set applications, we only need some of the above operations.

Iterating elements of a set

The above operations do not support efficient traversal of elements without changing the set.

- We do not need traversal always, but quite often anyway.
- Usually we need: "for all elements in set S do the following".

In algorithm notation, the traversal is usually presented as:

```
for each x in S do                    1
    operation for element x           2
```

- In Java (version $\geq$ 1.5) this is implemented for all collections that implement interface *Iterable*

```
for (x : S)                           1
    operation for element x           2
```

- where x is a reference (variable) to element type of collection S.
- This is very convenient, but does not allow changing the collection, nor it allows influencing the iteration (order).
- Alternative iteration:
  - native iteration of Java (*while*)
  - appropriate termination condition
  - a way to find the first and next elements to process.
  - iteration variable (and its type).

Definition 4.2. Set iteration operations (*Set* S; *Iterator* i; E x)

1. Iterator<E> S.iterator()
   Initialises iteration of set S and returns an instance of iterator.

2. boolean i.hasNext()
   returns boolean value *true* if iteration i of S has not been finished, i.e. a *next* operation would return an element, otherwise returns *false*.

3. E i.next()
   returns a next element of set S that has not yet been returned by iterator i. If all elements of set S have already been returned by iterator i (i.e., *hasNext*() would have returned false), throws a runtime exception.

4. void i.remove()
   Removes from collection S the element returned by the previous *next*() operation.

- You may not change the collection during the iteration (except using this iterator's *remove* operation). You may not even add new elements.
- You may/can not change yourself the value of iteration variable.
- You cannot calculate with iteration variable (as it is a reference in Java).
- You cannot compare iteration variables (or comparing makes no sense).
- Order of traversal is not necessarily guaranteed.
  - *HashSet*: iteration order can vary between iterations.
  - *SortedSet*: iteration in natural order of elements.
  - *LinkedHashSet*: iteration in order of insert.

- Nested iterations and iterating different sets using separate iteration variables.

Example 4.3. Traversing all elements using operations of definition 4.2:

```
import java.util.Iterator;                                                    1
...                                                                           2
Iterator<E> i = S.iterator();                                                 3
while (i.hasNext()) {                                                         4
    E x = i.next();              // or a known class instead of E            5
    operation for element x;                                                  6
}                                                                             7
```

⇒ A pre-test loop is essential as also the first *next*() can fail if set is empty!

iterator, *hasNext* and *next* should work in $O(1)$ (average) time.

Example 4.4. We can generate all permutations (orderings) of array V and collect them to a list using iterations. There will be $O(|V|!)$ such permutations.

```
public static <E> LinkedList<Vector<E>> permutations(Vector<E> V) {           1
    TreeSet S = new TreeSet(V);                                               2
    LinkedList<Vector<E>> L = new LinkedList<Vector<E>>();                    3
    permute(V, 0, S, L);                                                      4
    return L;                                                                 5
}                                                                             6
                                                                              7
public static <E> void permute(Vector V, int i, TreeSet<E> S, LinkedList<Vector<E>> L) {  8
    if (S.isEmpty())                                                          9
        L.add(new Vector(V));                                                 10
    else {                                                                    11
        for (E x : S) {                                                       12
            V.set(i, x);                                                      13
            TreeSet<E> R = new TreeSet<E>(S);                                 14
            R.remove(x);                                                      15
            permute(V, i+1, R, L);                                            16
}   }   }                                                                     17
```

java.util.Set

- http://docs.oracle.com/javase/7/docs/api/java/util/Set.html
- Interface defines set operations (but unions, etc. only change sets).
- You may not change the elements so that the result of a comparison *.equals*() changes!
- Elementary operations are as in our *tra.Set*, but two set operations change the calling set instead of creating a new set.
  - Two-set operations are then *addAll*(), *containsAll*(), *removeAll*(), *retainAll*()
  - We can, however, achieve the same results, e.g.:

    ```
    Set intersect = new TreeSet(A);                                           1
    intersect.retainAll(B);                                                   2
    ```

  - Time complexity $O(n)$ (*TreeSet* since Java 1.7) or $O(n)$ (*HashSet*).
- *java.util.HashSet*<*E*>
  - Implementation of *java.util.Set* using hashing (see "Hashing" p. 86).

- Time complexities of *add*, *remove* and *contains* are usually/average $O(1)$, though when hash runs out of space, rehash takes time.
- A single *null* in a set is allowed (but not recommended).
- Iteration only on "random" order.

java.util.SortedSet (JSE6- : *NavigableSet*)

- Interface that extends interface *Set*.
  - Either elements must implement interface *Comparable*, or collection must be given a helping class implementing *Comparator* interface.
    - *Comparable* does not allow *null* elements, *Comparator* can be tailored that *null* has a unambiguous position.
    - Elements in the set may not be changes so that comparison result changes!
  - *first*(), *last*(), *headSet*(), iteration in ascending order, ...
  - Sub-set operations provide a new view to the set, do not create a new set.
- *java.util.TreeSet*<*E*>
  - Implementation of *java.util.NavigableSet* using a balanced binary tree.
  - Like our *tra.Set*, but no two set operations (except *addAll*(), *containsAll*(), *removeAll*(), *retainAll*())
  - *add*, *remove* and *contains* in time $O(\log n)$.

4.2 Dictionary

⇒ Depending the application, we can limit the set operations.

- To manage a single collection it suffices to have*insert*, *remove*, *contains*.
- Actually, *remove* is not necessarily needed either.
  - We just add elements to collection and we then search.
    - E.g., compiler symbol table.
  - Sometimes we even create the whole collection at once, and only search.
    - E.g. (CD) telephone catalogue.

We call such single set collection as dictionary.

- Parameter x of *contains* and *remove* operations is usually called a key.
- The key is not necessarily the whole element. We use the key to specify the actual application data element.
  - Then *insert* and *get* need (return) the actual element, other work with key only.
  - Or *contains* returns actual data (or *null*) instead of boolean value.
- Within JavaAPI we can use a part of element as a key by having custom *.equals*() and *.compareTo*().
  - Element is given as whole, but comparisons use only key part.
  - Or, use a mapping (*Map*), see below.

Note: some libraries use term "dictionary" for a map (function), e.g., C#.

4.3 Relation and Map (function)

⇒ If the elements of a set have a special structure, we can define set operations accordingly.

Definition 4.5. Relation.

A are B sets, $A \times B$ is their Cartesian product.

- Elements of $A \times B$ are ordered pairs (2-tuples) (a, b), $a \in A$ and $b \in B$.
- Element types of A and B can be different.

$M \subseteq A \times B$ is a relation

- notation:
  - $(a, b) \in M \Leftrightarrow a\ M\ b$,
    - i.e., a and b are in mutual relation M
  - A is input set, B is output set

⇒ Relation is implemented as a set

- Set of tuples, i.e., set of objects/references.
- There can be more than two components in a relation (k-tuples).

Definition 4.6. Function

If for all elements (a_1,b_1), (a_2,b_2) of relation M holds

$$a_1 \neq a_2 \text{ tai } b_1 = b_2 \tag{4.1}$$

such relation is a function.

- Relation is function unless there can be two tuples that have common first members..
- Then b is the output (image) of a.
- Notation $M(a)=b$.
- In other words, if the output is defined, it is unambiguous.

Example 4.7. Example functions

- square $\text{sqr}(a)=a^2$
  - function can be calculated
- relation between student-id and personal data.
  - we need to store that personal data; data is found based on student-id.
- relation $<$ (among set of integers) is not a function:
  - $1<2$ and $1<3$

Function as an abstract data type

- Because function is a special case of set, we *could* use it with operations of general set.
  - We do not usually need operations of several sets.
  - Further, as elements are pairs (tuples), it is more convenient to use them with two separate parameters, e.g., in *add* operation.
  - We shall parametrize separately key type (input) and element type (output).

Definition 4.8. Map (function) as an abstract data type (*Map* M, $K\ k$, $E\ x$)

1. Map<K, E> Map<K, E>()
   creates an empty map, keys of type K and values of type E.

2. E M.put(k, x)
   assigns key k with element x in map M. Assigning the same key again will replace the earlier value (and returns the previous value). [compare array operation *set*()]

3. boolean M.containsKey(k)
   returns boolean value *true* if key k has been assigned a value, otherwise returns *false*.

4. E M.get(k) returns the value assigned to key k if it has been assigned, otherwise returns *null*.

5. E M.remove(k)
   removes key k and corresponding value from map and returns the value, or *null* if key k was unassigned.

- In infinite (esp. continuous) functions, the discrete *set*() operation is not useful, so it is omitted and *compute* (*get*) is implemented by a function (calculation).
- Map and relation are implemented similarly as a set.
- Time complexities: $O(\log n)$ using trees, $O(1)$ using hashing successfully (average).

java.util.Map, *HashMap*, *NavigableMap*, *TreeMap*

- See http://docs.oracle.com/javase/7/docs/api/java/util/Map.html
- Elementary operations, ordered map operations, sets of keys and values, etc.
- Time complexities depend on implementation (*Tree*/*Hash*).
- Part of operations are efficient (e.g., *containsKey*(), part are very slow (e.g., *containsValue*()).
- E.g., traversal of keys:

```
for (K key : M.keySet()) {
  ...
```

- Key-value -pairs can be managed with class *Map.Entry*.
- Java8: possibility to process and modify elements using lambda expressions:
  - *Map.compute*()
  - *Map.forEach*((*k*, *v*) -> ...)

4.4 Multilist

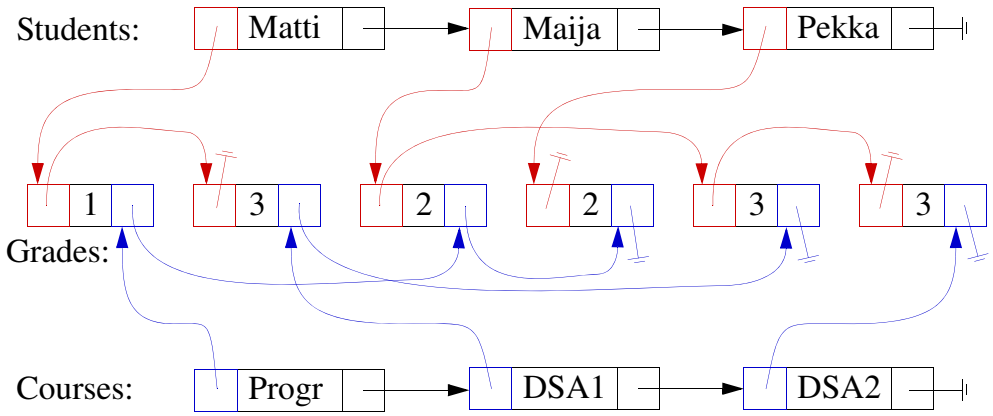


Figure 4.1: Multilist, return references omitted for clarity.

⇒ A method to implement relation.

- Returns quickly, e.g., "Which courses Matti have passed", or "who have passed course DSA1". Requires reference from Grades to Students and Courses.
- Insert $O(1)$, remove $O(1)$ (if 2-directional lists), search is fast if sets are implemented with trees instead of lists.

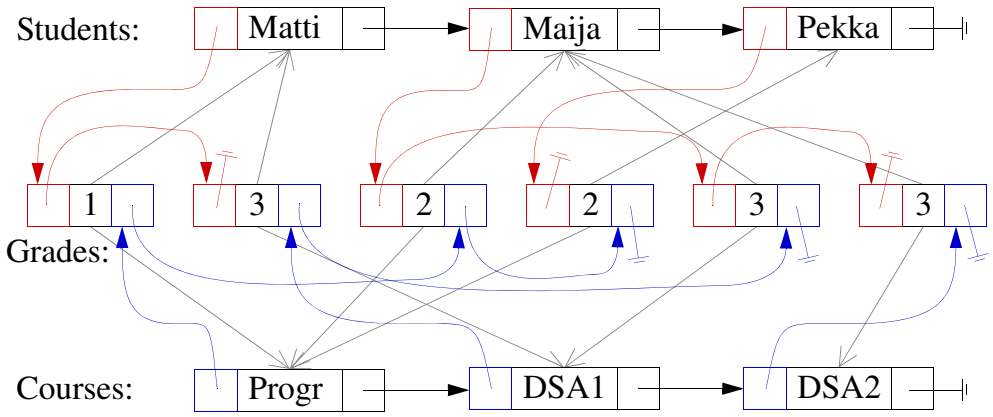


Figure 4.2: Multilist, return references visible

- If lists are 2-directional, then the relation element has 6 references and the element. Thus implementation must be careful.
- Multilists can be generalized to relations of more than two sets.
  - E.g., the teacher who gave the grade.
    - Third follower (and predecessor, and return) link in each element.
  - If we do not need fast "all grades given by a teacher", it suffices the have the return reference to teacher (leaving out the linked list of grades given by a teacher).

4.5 Priority queue

- As with list operations, we can also limit usage point and order of set operations to have an easy-to-use and efficient data structure for special needs.

Let us define a priority (importance, urgency) of elements.

- The element that is "better" ("more important") will enter processing (removed from queue) before than an element that has "worse" priority.
- Elements with equal priority will be removed in some order, before all elements that have worse priority.
- Usually the more important element is given a smaller priority value.

Example 4.9. Patients of first aid clinic form a priority queue: a patient with heart attack will overtake a patient with a broken finger — no matter how long the finger patient has been waiting.

Example 4.10. Priority queue compared to queue and stack.

- arrival order as priority ⇒ QUEUE
- reversed arrival order as priority ⇒ STACK

Determining the priority

- The determine the order of removal, we need to have an unambiguous order or elements, either by calculating priorities, or by comparing priorities.

Alternative implementations:

1. Priority is given as a floating point or integer value for each element inserted to the priority queue.

2. Priority is calculated automatically for each element inserted the the priority queue using a function *priority*(x) (which returns floating point or integer values).

3. Priority is given as priority function *compare*($x1$, $x2$) which is then used to set the order of elements.

- Of these 1 is easiest to implement but least flexible. We use this in data structure library (*AssignablePriorityQueue*).
- *java.util.PriorityQueue* uses alternative 3.
  - In practice *java.util.PriorityQueue* assumes either element type to implement interface *Comparable*, or to have a separate (*Comparator* given for the constructor of priority queue.

Changing priority?

- Occasionally we need to be able to change the priority of an element in the priority queue. Usually improving priority is enough.
- Then we need an ability to identify the elements in the queue (which is impossible using the basic operations).
  - We actually need to identify the nodes hosting the elements.
- See *AdjustablePriorityQueue*.
  - We'll return to this later with priority queue implementation (p. 89).

Priority queue operations

- As in queue, we need essentially add to queue ("somewhere") and retrieval & removal of most important element ("the remaining top priority one").
- In insert operation, we also give the priority of the element somehow.
- The element under processing (front of queue) has the lowest priority value, thus operations apply for the element that has lowest priority value.
- As opposed to set, in a priority queue we can have the same element several times, or several elements with the same priority.
  - The elements with equal priority will be removed in order of insert (or in random order).

Definition 4.11. Operation of priority queue *java.util.PriorityQueue*: (*PriorityQueue*<*E*> *P*, *E x*. *E* implements interface *Comparable*)

1. PriorityQueue<E> PriorityQueue<E>()
   Creates a new priority queue.

2. boolean P.isEmpty()
   Returns boolean value *true* if priority queue P has no elements, otherwise returns *false*.

3. void P.add(x)
   Inserts element x to priority queue P.

4. E P.poll()
   Removes and returns the first (most important) element of priority queue P.

5. E P.peek()
   Returns the first (most important) element of priority queue P.

Time complexities depend on implementation. In efficient heap implementation *add* and *poll* are $O(\log n)$ (where n is the number of elements currently in queue) and other operations are $O(1)$ (see "Implementing priority queue" p. 89).

Example 4.12. So called heapsort can be implemented using a priority queue. A simple version can be made using a priority queue interface. If elements to be sorted are in a list L, the algorithm is very straightforward: elements to priority queue and back. [*HeapSortEsim.java*]

```
public static void heapSort(Collection A) {
    Iterator i = A.iterator();
    PriorityQueue P = new PriorityQueue();
    // elements to priority queue
    while (i.hasNext()) {
        P.add(i.next());
        i.remove();          // or .clear() at once at end
    }
    // elements back to the collection
    while (! P.isEmpty())
        A.add(P.poll());
}
```

If operations of collection A, especially *remove* and *add* from/to end are efficient, the whole time complexity is $O(n \log n)$, if we have n elements to sort. We'll return to efficient (unit space) implementation of heapsort later (p. 91).

Example 4.13. k^{th} largest element of a collection. Start adding elements to a priority queue. When the priority queue has $k+1$ elements, the smallest of those can not be k^{th} largest, thus it can be removed (*poll*). Continue making *add*+*poll* operations until input collection has been iterated. Now k^{th} largest element it on the front of the priority queue (*peek*). Time complexity? [Exercise].

Algorithmic use cases of priority queue

- Optimization (generate new possibilities) best of current is selected for further processing..
- Weighted search.
- Partial sorting, selection.

4.6 Bag

⇒ Same element can occur several times in a "set".

- General set (dictionary, map) forbids several equal/same elements to be stored in the set.
- In a priority queue same element is usually allowed, as are equal priorities.

If we need multiple equal/same elements, it can be allowed!

- Then we call it a bag or multiset.

Bag as an abstract data type:

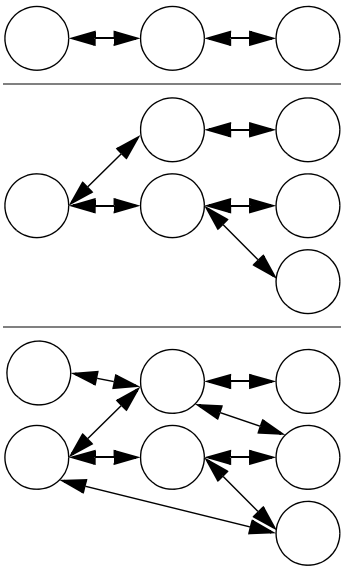
- As set (operations as needed).
- *add*(x) inserts x to the bag even if an equal element is in the bag already.
- *remove*(x) removes all occurrences of x.
- *remove*(x, i) removes at most i occurrences.

⇒ Java API does not contain Bag implementation.

- We can implement it ourself, but full implementation is fairly laborious.
- There are ready third party libraries available in the Internet, e.g., *org.apache.commons.collections.Bag*, http://commons.apache.org/.

- If elements are truly identical (no deed to store several objects), then a function (*Map*<*E*, *Integer*>) is often enough when we use the integer to store the number of elements of given key.
- Alternatively, we can use *Set* and properly built *compareTo*() method of the element type.
  - Similar elements will be store sequentially in some unambiguous order.
  - ... *All*() operations will not work, however.

5 Graphs

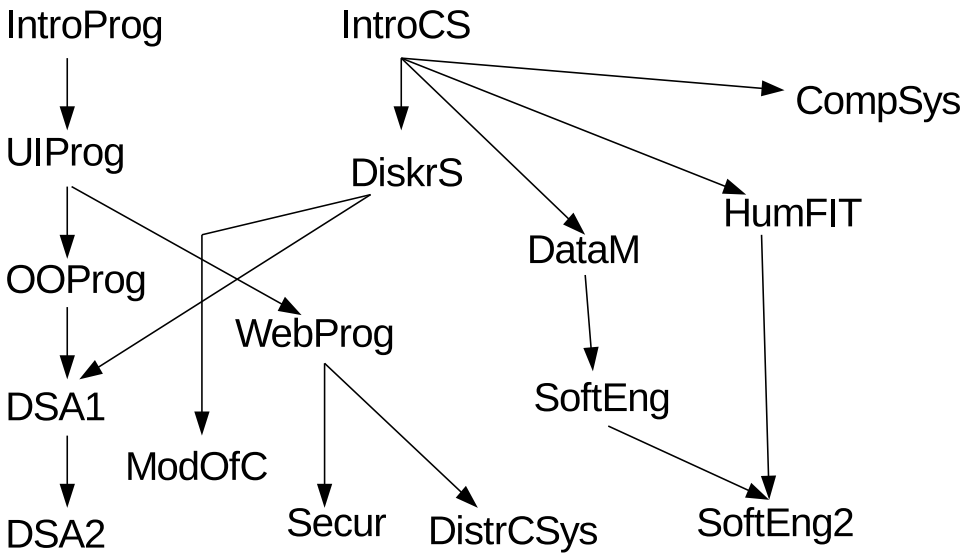


- List:
  - each element has one successor
  - each element has one predecessor
- Tree:
  - each element has one predecessor (parent)
  - each element has several successors (children)
- How about when we need
  - many predecessors and
  - many successors?

⇒ (Directed) graph.

- If predecessor/successor direction has no significance in application, then we can use undirected graph (see p. 58)).

Example 5.1. Computer Science course order for BSc (example):



Applications

- (Data/person/material/electricity/etc) traffic modelling.
  - Information network, roads, electric circuits, process diagrams.
  - Routing, traffic planning, electric current planning.

- Relational diagrams, class diagrams, entity-relation diagrams, use case diagrams, organizational diagrams.
- Hypertexts, references.
- Chemical, physical, technical, biological, sociological, economical networks and their relations and interactions.

Algorithms partly application dependent

- Here only elementary concepts very briefly.
- On DSA II course: abstract data type Graph and some elementary algorithms.

Graph concepts

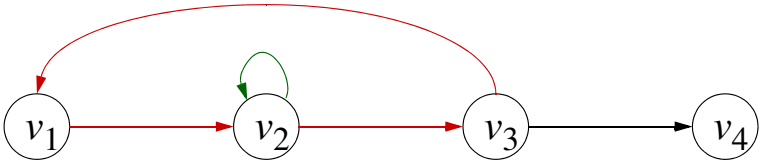
Directed graph $G=(V,E)$

- V is set of vertices (nodes).
  - basic element of a graph
- E is a set of (directed) edges ($\approx$ relation).
  - Connection between to vertices (neighbourhood)
  - Directed edge is marked as (u,v) tai $u \to v$
    - u is start vertex
    - v on end vertex
  - End vertex v is a neighbour of start vertex u.
    - but not vice versa, unless also $(v,u) \in E$.
    - $(u,v) \neq (v,u)$
- V and E are always finite $\Rightarrow$ G is finite.
  - It can be $V \neq \emptyset$ and $E = \emptyset$, but
  - if $V = \emptyset$, then also $E = \emptyset$.
- Each vertex can have arbitrary number of neighbours.
  - A vertex can be its own neighbour (edge (u,u), loop)
  - They can be (if needed) multiple parallel edges between the same vertices. Also multiple loops.
    - These can be restricted, depending the application.

Definition 5.2. Path.

- List of vertices $(v_1, v_2, \ldots, v_n)$ in a directed graph is a *path*, if (v_i, v_{i+1}) is an edge for all $i = 1,\ldots,n-1$.
  - Length of the path is then $n-1$ (number of edges).
- Path is *simple*, if $v_i \neq v_j$ when $i \neq j$ or ($i=1$ and $j=n$).
- Simple path where $v_1 = v_n$ is a *cycle*.

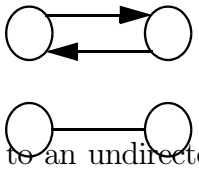
Example 5.3. A graph with four vertices:



- $V = \{v_1, v_2, v_3, v_4\}$.
- $E = \{(v_1,v_2),(v_2,v_2),(v_2,v_3),(v_3,v_1),(v_3,v_4)\}$.
- Cycles (v_1,v_2,v_3,v_1) and (v_2,v_2).
- Paths for example $(v_2,v_2,v_3,v_1,v_2,v_3,v_4)$ and (v_3,v_1).

Undirected graphs

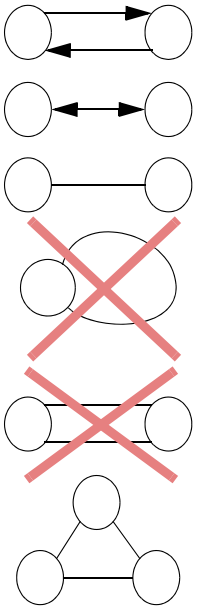
⇒ In an undirected graph both directions of an edge are always usable.



- Suites for applications where the connection is always symmetrical.
- Slightly easier to perceive and draw.
- Slightly more difficult to implement.
- Most directed graph algorithms suit as such (or with small adjustments) to an undirected graph.
  - E.g., shortest paths.
  - When following an edge, we need, however, to ensure that we do not accidentally return back along the same edge.

Undirected graph definitions

⇒ As directed graph, but:



- $(u, v) = (v, u)$
  - Edge has no direction.
  - Neighbourhood is always bi-directional.
  - Edge (and thus path) can be traversed to both directions.
- Edge from vertex back to itself (loop) is not usually allowed.
- There can (usually) be at most one edge between any two vertices.
- Because a cycle may not include the same edge twice (and remembering the two previous points), then a cycle has at least three vertices.

Application data

- Both vertices and edges can hold element data (reference), depending the needs of the application.
- Vertices and edges usually hold different types of element.
- Further, we can give edges and vertices a weight (price) based on which results are calculated.
  - E.g., (edge) distance, time, price, capacity, flow, probability, etc.
  - E.g., (vertex) price, population, volume, power, temperature, time, popularity, etc.
- If needed, we can use also marks of already processed vertices/edges. We usually call these marks as colours.
- In our data structure library, ”element” is a string, weight is floating point number, and colour is enumeration (integer).
  - Additionally, vertices have integer index which can be used to enumerate vertices.
  - If/when application needs more information, we can extend the class and add more members.

Graph as an abstract data type

Graph is made of a set of vertices and a set of edges.

- We use (applied) set operations to manage them.
  - Vertices are clearly a set.
  - Edges are actually a relation.

What operations we need?

- creating a graph
- insert, remove (vertices and edges separately)
- Existence of vertex/edge, finding and modifying element
  - Inspection and setting of colours and weights (vertices and edges)
- Usually also iterations
  - all vertices
  - all edges
  - Neighbouring vertices of a vertex.
  - Outgoing edges from a vertex.

There is no need to implement a set of all edges. Instead we store set of neighbours for each vertex.

Graph problems/algorithms

⇒ Graph theory is probably the most diverse and rich field of algorithmics — there are "endless" possibilities for applications and algorithms.

- See books on graph theory, algorithms.
- Path lengths, existence, maximum flow, separate paths
- Graph enumeration, graph isomorphism (comparison)
- Graph connectedness, strongly connectedness, (strongly) connected components, k-connected, articulation vertices, etc.
- Finding | removal | forming cycles/cliques
- Spanning trees
- Euler, Hamilton, etc. traversals (travelling salesman (with variants)
- Augmenting graph to achieve desired properties.
- Colourings
- Partitions of a graph. (forming clusters with minimum/equal weight, distance, edges, etc.
- Drawing a graph on a plane | space | sphere | torus, etc.
- Topological sorting
- k-partite graph, matching
- Covers
- For graphs of different types (vertices in d dimensions, different distance metrics, negative weights, non-connected, acyclic, etc.).

⇒ Continues on course DSA II

6 Sorting algorithms

⇒ In sorting problem, we arrange the elements of a (sequential) collection to linear order.

- Usually we leave the sorted elements to the same collection that we got as an input.

- Sometimes, we create a sorted index of an existing collection, but do not change the input.
- In internal sorting, all elements fit at once in main memory.
  - In external sorting, the elements are in mass storage. We are only able to process part of elements at a time in main memory (DSA II).

How important sorting is?

- Often large sets of data are stored "in order" (indexed) — re-sorting of all data is rarely needed.
- (Correctly done) sorting is quite fast (at least in main memory).
- In expert systems (AI) we often grade a large number of possible solutions and select a few best ones to be processed further.
  - Large amounts of data.
- Some routing protocols sort the packets based on destination — sorting must be very fast (parallel).
- Many linear algebra methods are more accurate, if we sort the input matrix appropriately.
- (Mass)memory access can be optimized by sorting the queued references.
- Sorted array/list can be used as a priority queue.
- An efficient sorting algorithm is available in (almost) all API libraries.

⇒ Sorting is not very important, but interesting and educational.

- Here we learn different algorithms for the same problem. We learn different methods (algorithm strategies). Continues on DSA II and DAA.

6.1 Memory-based sorting

- When all elements fit at main memory at once, we can access the elements in arbitrary order (Random Access Memory).
  - Sorting an array is easiest:
    - direct ($O(1)$ time) access to an index
    - insert to middle is slow ($O(n)$).
    - usually we sort by exchanging two elements (or by copying elements to empty slot)
    - or use a temporary array/list to store some or all of the elements (then space complexity is usually linear).
  - We can also sort a list efficiently:
    - only sequential processing (follower/successor) is efficient
    - insert to middle and splitting the list are efficient
    - We don't usually need extra space if the elements are in one list only in any given moment.

Sorting key

- Ascending order is decided based on sorting key.
  - Key does not have to be the whole element.
  - Either we sort the elements in order of keys, or
  - we create a new ordered index of keys which has links to the whole elements.
  - Occasionally, we need several sorting keys, e.g.,
    - primarily based on family name
    - secondarily based on given name

        * finally based on phone number
    - In some cases, the order is not total (some elements have no mutual order).
        * If two elements have no mutual order, such elements should be left to the same order they were before sorting — so called stable sort.
        * 5 3 3 6 → 3 3 5 6
        * Then we can use the same sorting algorithm in stages. First with least significant key (part), then with more significant keys (see "Radix sort" p. 66).
    - In comparison based sorting, it suffices to have operation $e1.compareTo(e2)$ of the element type which will return whether element $e1$ precedes element $e2$ or not.
        * Or, a separate comparison function/class (*Comparator*) with method $compare(e1, e2)$.

Time complexity of internal sorting

- We calculate the number of steps as usually.
- If comparison is very slow (long key), take that into account.
- Number of comparisons is usually a good measure, unless we do many other (slow) operations.
- Number of swaps, if elements are large and if we swap/move them.
    - If elements are large (and in Java always), they are handled with references, which are unit size.

⇒ Actually we sort an array of references!

6.2 Simple sorting algorithms

Bubble sort

- Swap consecutive elements repeatedly while iterating over the collection.
- Works also for list, see *SortingExample.java*.
- $O(n^2)$

Insertion sort

- Move each element to correct position in an array.
    - Or add elements to a new array.
- Making space for elements will take time, so does finding the place.
- $O(n^2)$

Selection sort

```
for (int i = 0; i < n−1; i++)                    1
  swap(A[i], min(A[i], ... , A[n])).             2
```

- Smallest of the remaining elements to the correct position.
- $O(n^2)$

Most $O(n^2)$ time sorting algorithms run in the same time in both the best and the worst case.

- By adjusting the iterations properly, we can have the best case (usually readily sorted array) in $O(n)$ time.
- If we know that the input is in almost ascending order, we can select some special sorting algorithm. (Exercise).
    - Only k last/first elements in wrong position.
    - Each element only k positions off from correct position.

  - etc.
  - Timsort: utilizes existing ascending sequences in data, makes new sequences using insertion sort. For longer sequences, use merge sort.

In general, using time $O(n^2)$ for sorting is very slow if we have more than few elements, we prefer to use $O(n\log n)$ algorithms:

Figure 6.1: Most common sorting time complexities

n	$n\log n$	n^2
10	40	100
1 000	10 000	1 000 000
1 000 000	20 000 000	1 000 000 000 000
1 000 000 000	30 000 000 000	1 000 000 000 000 000 000

- There are also sorting algorithms that run in (average) time $O(n^k)$, where k is a fraction between (1,2], e.g., Shellsort.
- Remember that $n\log n = o(n^{1+\epsilon})$
  - There is also a $O(n\log^2 n)$ version of Shellsort [Pratt].

6.3 Quicksort

- C.A.R. Hoare 1961
- In each stage, first divide elements to two parts according to chosen pivot element.
  - To the beginning of array those elements that are smaller than the pivot element.
  - To the end part those elements that are larger than the pivot element.
- Then sort recursively the beginning and end parts (and leave pivot element between).

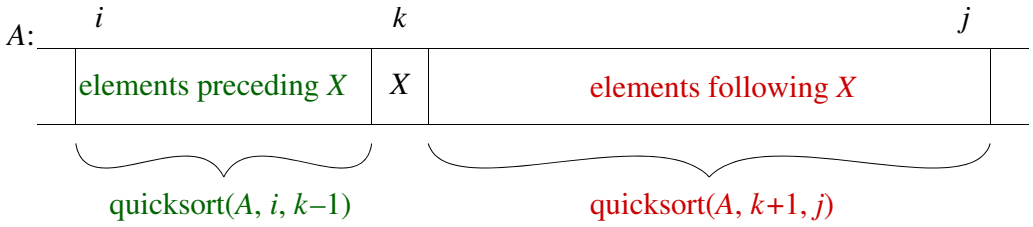


Figure 6.2: Recursion of quicksort **after** *partition* stage.

⇒ Divide-and-conquer.

Algorithm 6.1. Quicksort.

```
public static <E extends Comparable<? super E>>
                    void quicksort(E[] A, int i, int j) {
    if (i < j) {
        int k = partition(A, i, j);        // divide elements to smaller and larger than pivot
        quicksort(A, i, k−1);              // sort beginning of array
        quicksort(A, k+1, j);              // sort end part of array
    }
}
```

Whole array is sorted by calling *quicksort*(A, 0, $A.length$–1).

Choosing the pivot element

- Concerning time complexity, the optimal case would be to have subarrays of equal size ($n/2$).
  - Then the pivot element should be median of the (sub)array.
  - Unfortunately, finding median is too time consuming, see below.
- Easiest is to just pick some element as a pivot element.
  - Random element, middle element, (first element).
- We can even use some time to find a good pivot element.
  - Pivot element is chosen once on each recursive call, *partition*() takes anyway $O(n)$ time.
  - (median | average) out of k (random | equally spaced) elements.
    - We can calculate average of (sample) of elements if they are numerical.
  - Random selections improve average running time with real world inputs.
  - Always it is possible to find a rare special input where a method does not work well.

Pivot value (e.g., average) instead of pivot element

- Divide the elements of array to smaller and larger than the pivot value.
- There will not remain a pivot element, thus recursion call will be $i..k$ and $k+1..j$.

Partition algorithm

- Initially subroutine *partition* takes the pivot element to a temporary variable to have an empty slot in array for moving the elements.
  - If/when the pivot element was chosen elsewhere than start, then we'll move the first element in old place of the pivot element to have an empty slot in the beginning of (sub)array.
- We'll inspect the subarray $A[i..j]$ alternatively from both ends (indexes i and j) until the partition is finished.
  - When we find from the end of the array an element that is smaller than pivot element move it to the beginning of array to the empty slot. There will be an "empty" slot in the end.
  - When we find from the beginning of the array an element that is larger than pivot element move it to the and of array to the empty slot. There will be an "empty" slot in the beginning.
- Finally we put the pivot element to the empty slot between small and large elements.
  - The position is final for the pivot element.
  - Subarray has thus been divided to subarrays of small and large elements. New subarrays can be sorted independently and recursively.
- There are other variants of partition, but principle and result is the same anyway.
  - Instead of having a temporary variable holding the pivot element, some common variants remember where the pivot element is going during partition and always swap pivot and an element that has to be moved.

Algorithm 6.2. Partition of quicksort:

```
public static <E extends Comparable<? super E>>                                    1
              int partition(E[ ] A, int i, int j) {                                2
  E pivot = A[i];                    // only an example, never use in real program 3
  while (i < j) {                    // repeat until i and j collide               4
    // find from end element smaller than pivot                                    5
    while ((i < j) && (pivot.compareTo(A[j]) < 0))                                 6
```

```
            j--;                                                    7
        A[i] = A[j];                                                8
        // find from beginning element larger or equal than pivot   9
        while ((i < j) && (pivot.compareTo(A[i]) >= 0))             10
            i++;                                                    11
        A[j] = A[i];                                                12
    }                                                               13
    // assign pivot to correct place and return its position        14
    A[i] = pivot;                                                   15
    return i;                                                       16
}                                                                   17
```

Time complexity of quicksort

- *partition* is obviously $O(n)$, where $n = j - i + 1$.
  - On each step either j decreases or i increases.
- If pivot would always hit the middle of the array (best case), time would be

$$T_{best}(n) = \begin{cases} O(n) + 2 \times T\left(\frac{n}{2}\right) & \text{, when } n > 1 \\ O(1) & \text{, when } n \leq 1 \end{cases}$$
$$= \underbrace{n + 2\left(\frac{n}{2}\right) + 4\left(\frac{n}{4}\right) + 8\left(\frac{n}{8}\right) + \cdots + n \times 1}_{\log n \text{ terms}} \quad = O(n \log n) \tag{6.1}$$

- If pivot would always hit the first index (worst case), then time would be

$$T(n) = \begin{cases} O(n) + T(n-1) + O(1) & \text{, when } n > 1 \\ O(1) & \text{, when } n \leq 1 \end{cases}$$
$$= \underbrace{n + (n-1) + (n-2) + (n-3) + \cdots + 1}_{n \text{ terms}} \qquad = O(n^2) \tag{6.2}$$

- Average case analysis (for random input) is more difficult, we just note: $O(n \log n)$.
- Space complexity comes from recursion (number of recursion levels, or subtask stack), in best case (and average) $O(\log n)$, worst case $O(n)$.

Tuning

- When progressing in recursion, small tasks (e.g. $n \leq 9$) using bubble or insertion sort (exercise).
- Two (or more) pivot points: a bit less recursion and less movements.

Selecting pivot element in practice

- First element can be selected as pivot only on educational purposes, never for real world inputs.
- Median out of three (or 9) random (or evenly selected), or at least a random element.
- See, e.g., Java API *java.util.Arrays.sort*(*int*[]) source code ($\leq$ JavaSE6).
  - Java SE7 changed to Dual-Pivot version
  - Object sorting is made using tuned merge sort (Timsort).

Selection (median)

$\Rightarrow k^{\text{th}}$ smallest (or largest) element of an array.

- Sorting, index k: $O(n\log n)$ (or $O(kn)$)
- Priority queue: $O(n\log k)$

Using *partition* subroutine of quicksort:

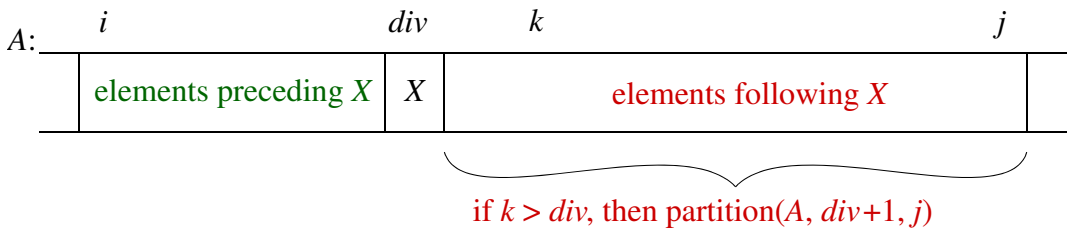


Figure 6.3: Using *partition*() in selection (*div* replaces k of *partition*).

- After executing *partition* subroutine, we know on which side of pivot the selection (k) is.
- Repeat the partition recursively only to the subarray where k is.
- The subarray to search gets shorter on each iteration. To half in best case, but at least by one.
- If division would always hit the middle, then:

$$T_{best}(n) = \begin{cases} O(n)+T\left(\frac{n}{2}\right) & \text{, when } n>1 \\ O(1) & \text{, when } n\leq 1 \end{cases}$$
$$= \underbrace{n+\frac{n}{2}+\frac{n}{4}+\frac{n}{8}+\cdots+2+1}_{\log n \text{ terms}} \qquad = O(n) \tag{6.3}$$

- If, however, the division hits always to the beginning, time complexity is $O(n^2)$ as is for quicksort.
- Pivot must be selected wisely, e.g., using median of three method, especially when input is still large.
- By selecting the medians cleverly (median of group of 5 medians) we can guarantee linear time complexity (with largish constant factor), see, e.g., Weiss p. 374).
- Notice, that this algorithm re-arranges its input, so take a copy if/when it is not acceptable.

6.4 Heapsort

- Move elements to priority queue and then back to original collection.
- See Example 4.12 (p. 55).
- $O(n\log n)$ in both best and worst case.
- With a small modification k smallest elements in time $O(n\log k)$.
- Only k last elements in wrong place: $O(n+k\log k)$.
  - If $k \leq n/\log n$, then $O(n)$. (exercise)
- See "Implementing priority queue" p. 89.
  - Constant space!

6.5 Merge sort

```
public static void mergesort(Comparable A[], int start, int end) {                1
  if (start < end) {                                                              2
    int mid = (start + end) / 2;                                                  3
    mergesort(A, start, mid);                    // sort halves recursively       4
    mergesort(A, mid+1, end);                                                     5
```

```
        merge(A, start, mid, mid+1, end);          // merge sorted halves
    } }
```

Time complexity

- *merge*(): $O(n)$, where $n = \text{end} - \text{start} + 1$.
- *mergesort*():

$$T(n) = \begin{cases} 2 \times T\left(\frac{n}{2}\right) + O(n) & \text{, when } n > 1 \\ O(1) & \text{, when } n \leq 1 \end{cases} \qquad (6.4)$$
$$= O(n \log n) \quad \text{[Best and worst case!]}$$

- Space complexity (extra space) $O(n)$ in straightforward solution, with some extra organization only $O(\log n)$.
- Works also using lists, because traverses input only in sequential order.
  - Works also using sequential access files.
  - We'll return to this during mass memory algorithms in DSA II.

6.6 Bin sort and radix sort

Bin sort:

Values of keys (or elements) are integers (e.g.) $1..m$, where m is reasonable.

- Form m bins (array $1..m$), in $O(m)$ time.
  - If whole element is (or can be represented as) an integer $1..m$, it suffices to count how many elements will be in each bin $\Rightarrow$ array of integers is enough.
  - If $1..m$ is only a key, and we need to store the rest of the element, then we'll have array of element lists.
- For each element of input collection, place it directly to correct position based on the key.
  - If implementation is efficient, each add can be done in unit time and whole binning process in $O(n)$ time.
- Finally, either combine the m bin lists into a single list ($O(m)$ time), or copy elements from the bin lists back to original collection ($O(m+n)$ time).
- Whole algorithm in $O(m+n)$ time and space.

$\Rightarrow$ Notice: not a single comparison!

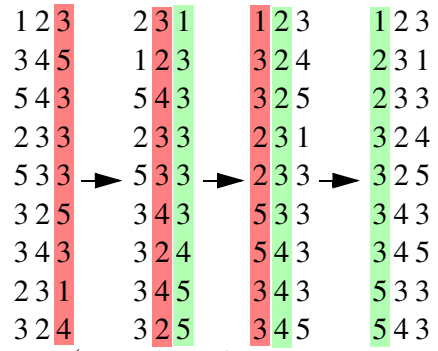
Algorithm 6.3. Bin sort.

```
public static binsort(E A[], int maxkey) {
    Vector<List<E>> bins = new Vector<List<E>>(maxkey+1);
    for (int i = 0; i <= maxkey; i++)
        bins[i] = new LinkedList<E>();
    for (int i = 0; i < A.length; i++)
        (bins.get(A[i].getKey())).add(A[i]);
    A.clear();
    for (int i = 0; i <= maxkey; i++)
        A.addAll(bins.get(i));
}
```

Radix sort

- Bin sort is inefficient if the range of values (m) is large.

  - Strings or floating point numbers as keys (too many bins).
  - Maintaining the list may take time.
- Instead, make sorting in stages according to parts of keys.
  - Split the length of keys to proper sizes. Sort first according to least significant part, then according to the next significant part, ..., finally according to the most significant part of the key.



- Key splitting should be done using modulo arithmetic, or bit operations, depending on key type.
- Sorting stages must be stable, i.e., mutually equal elements (according the current part of key) must remain in original order.

Sorting using a histogram

- We also improve from bin sort so that we do not create bins, but iterate over the input twice, first count the occurrences (histogram), and on the second iteration, assign the elements to the position given by 0-prefix sum of the occurrences array.

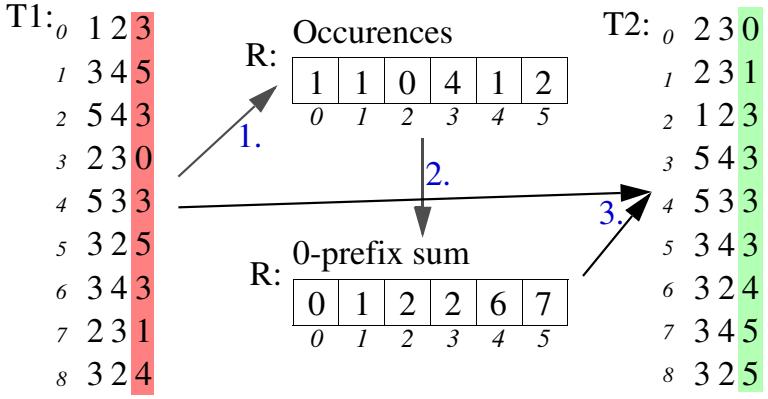


Figure 6.4: Using histogram in radix sort.

- Prefix sum points instantly to which position elements of each "bin" are placed. The elements of the same "bin" are held in original order.
  - sub-key(T1[0]) = 3 ⇒ T2[R[3]] = T[0]; R[3]++;
  - sub-key(T1[1]) = 5 ⇒ T2[R[5]] = T[1]; R[5]++;
  - sub-key(T1[2]) = 3 ⇒ T2[R[3]] = T[2]; R[3]++;
  - T2[R[T1[i]]] = T[i]; R[T1[i]]++;
  - or even T2[R[T1[i]]++] = T[i];
- Elements that are equal based on the current sub-key, are kept in correct order as they are processed in original order and the prefix sum values are incremented by one after each position is used (element assigned).
- If/when keys are not integer values, we usually use binary representation of keys r bits at a time (and often with integers also to avoid division operations).
- The number of bins is thus 2^r.
  - Typical value for r is 8-20, depending on input data and system memory architecture (R-array should fit in cache).
  - Using single temporary array, it is useful to keep $\lceil \frac{m}{r} \rceil$ as even.
  - Time complexity is $O\left(\frac{m}{r}(n+2^r)\right)$, where m is the length of the key in bits.

- Caches are used a lot (especially if r is large), second stage depends on random access write to memory.

⇒ Radix sort can be the fastest sequential sort if keys are short and input is large.

7 Implementing abstract data types

⇒ To be able to actually use an abstract data type, we have to implement it.

- Primarily, we should select an existing ready implementation, but sometimes a proper implementation is not available.
- Proper (usable, efficient enough):
  - the needed operations
    - efficient enough
  - usable element type
  - implemented for our programming language
    - in this course Java
- If no suitable implementation is available, we need to do it ourself.
- Also, understanding the implementation helps us using the data types correctly.

Even when we implement an ADT ourself, we still separate interface and implementation, to be able to reuse the implementation later on.

- Truly separate!
  - No surprises to data structure side, nor to application side.
  - Separation already in planning phase.
  - Separation also in documentation.
- Bonus from separation is modularisation of algorithm and implementation, thus both are easier to understand, test, maintain, and port.

7.1 Encapsulation and parametrisation

⇒ We'll implement a collection defined by given/needed abstract data type.

- Interface dictates implementation (not vice versa).
  - Naturally, while planning the interface and implementation, we can tune some interface details to suit for given implementation (especially to keep implementation efficient).
  - Interface documentation will be added some details only depending implementation (e.g., time complexities, limitations).
- Actual implementation we can choose freely (in ideal case).
  - In practice, implementation usually impacts also interface, e.g., position behaviour of lists.
  - In, e.g., stack and dictionary, however, implementation only impacts time complexity.

Encapsulation

- We give the user only interface, not implementation.
- User should not be able to use implementation-dependent features (except via interface).

Interface

- types

  - limitations
- operations
  - parameters, return values
  - limitations
  - time complexities
  - behaviour in error cases (exceptions)
- documentation

Implementation

- Hidden from user
- Internal types/classes
- Internal procedures/method
- Implementations of operations (procedure/function/method bodies)

⇒ Implementation exactly as interface and documentation describes.

⇒ Different programming languages support encapsulation in different ways.

Java

- We can present the *interface* and the *class* implementing it separately, e.g., *java.util.Set*, *java.util.HashSet*.
  - User can use operations defined in the interface, but creating a new instance requires either a real class, or a factory method in interface.
- In simpler cases it suffices to write the "interface" and implementation directly to the same class.
  - We don't actually write the interface separately at all.
  - Generate the class description using javadoc.
  - User won't need/get the class source code, but can get all needed information to use the collection in the generated documentation.
  - Good:
    - easy to do (as long as we adhere to javadoc comment formatting (IDE helps),
    - We can select which parts are documented. We can edit manually afterwards if needed
    - support class hierarchies (especially inherited methods)
  - Uncertain:
    - We need to remember to update the documentation.

Pascal:

DSA_BTree.p:

```
module DSA_BTREE interface;
export DSA_BTREE = (BTREE, BTREE_ASSIGN_CHILD, ...)
type BTREE = ...
procedure BTREE_ASSIGN_CHILD(T:BTREE;
      ND:BTREE_NODE; dir:BTREE_DIR; NS:BTREE_NODE);
procedure ...
end. (* interface *)
```

```
module DSA_BTREE implementation;
procedure BTREE_ASSIGN_CHILD(T:BTREE;
        ND:BTREE_NODE; dir:BTREE_DIR; NS:BTREE_NODE);
begin
        ...
end;
...
end. (* implementation *)
```

Not that clean:

- Interface and implementation in the same file.
- Compiled interface (.gpi or .TPU) is not readable.
- Redundancy (identifiers up to three times).

C/C++

DSA_BTree.h:

```
typedef struct BTREE_str * BTREE;
void BTREE_ASSIGN_CHILD (BTREE T,
        BTREE_NODE ND, BTREE_DIR dir, BTREE_NODE NS);
```

DSA_BTree.c:

```
#include "DSA_BTree.h"
void BTREE_ASSIGN_CHILD (BTREE T,
        BTREE_NODE ND, BTREE_DIR dir,
        BTREE_NODE NS)
{
    ...
}
```

- We cannot hide quite everything (otherwise type checks would not work).
- No named modules, just files in ANSI C.

Parametrisation

- When we implement a collection, we want it to work with different element types as easily as possible.
  - Preferably without creating a new implementation.

Requirements:

- Element type creation, comparisons, prints, copying, freeing, etc. must be implemented in element type.
- If element type is large (structured), then the element in ADT is a pointer/reference to the structure/object. Then we need to be careful in allocating and freeing memory.
- Types and operations should be named so that there won't be conflicts in using several different ADTs in the same program (depending the namespace limitations of used programming language.).
- Headers/interfaces and implementations separated.

Possibilities to implement parametrisation:

Textual parametrisation

- In interface and implementation of ADT, use a pseudo type (named, e.g., *element*).
- When we use the ADT, we bring the real element type to interface and implementation.
  - *type element = real /* typedef */*
  - #define *element ...*
  - #include "*int_element*.h" /* "*element.h*" */
  - *template* mechanism in those languages that support it (C++, Modula-3).
- Element type definition includes comparisons, creation, free.

⇒ In textual parametrisation, the ADT implementation will be compiled with the program (using the new element type).

- The compiled program will have an own copy of the parametrised collections/methods for each element type used.
- Usage difficulty and level of encapsulation vary from language to another.
- Good: compiler will check type compatibilities in all cases.
- Bad: inflexible in many languages, implementation must be public, compiled code will grow because of copying.

Typeless element

⇒ Element type is typeless pointer/reference (*void**, *pointer*, *Object*).

- ADT implementation can be encapsulated and compiled beforehand.
- If we want type checks, the user of the ADT has to either
  - cast the typeless pointers/references always to correct type, or
  - make (generate) wrapper macros/operations around the actual operations.
- Parameter type of comparisons, adds, etc. is always pointer, programmer has to encapsulate also simple types to pointers.
- The programmer must ensure correct usage of ADT (e.g., pop the same type from stack that was pushed there).
- Good: no need to have different implementations or operations to different elements types.
  - But, we need to give comparison etc. operations as parameter.
- Bad: compiler cannot do type checks.

Typeless element without pointers: for each element, reserve a byte array of proper size (number of bytes) within the implementation data structure.

- Operations will return a pointer to this byte array.
- Good: memory allocation and free are automatic (done with data structure allocation).
- Efficient: only one block of memory/element.
- Weakness: not that easily portable (element sizes depend on architecture).
- Requires careful usage (discipline).

Combination

⇒ Element that includes a reasonable variety of possible element types.

- variable record / union
- including typeless pointer to be used if variety does not have the needed type ready

- no extra space
- Careful use required (and/or type checks in implementation).
- For build-in types, comparisons, etc. can be readily implemented.
- Operations implemented as both typed and typeless.
- Collection knows its type.
- Element type knows its operations.
- Element (node) knows its collection or its element type.
- See Pascal and C versions of the TRA-library.
- C11 has a macro mechanism to create "generic functions" for listed types.

Object-oriented implementation (without generics system)

- Collection will be implemented as a class, operations as methods.
- Relative ADTs are implemented through inheritance (only new and changes operations need to be implemented).
  - Even different collection classes have something in common, thus they all can be subclasses of the same abstract collection.
- Any object can be used as an element.
  - Element type (class) is fixed when creating the collection (constructor parameter), or when adding the first element.
  - Element type checks at compile time (if we use casts) and/or at runtime.
- For comparisons, prints, etc. we can require the element type to be a subclass of a suitable class, or require the subclass to implement a suitable interface (cf. *Comparable*).
- Casts are a bit inconvenient if we have to use them.
- Alternatively, parametrisation can done by creating a subclass having the element type.
- Concept of ADT fits excellently for object-oriented programming, and vice versa.

Java generic collections

⇒ Apparently similar to the textual parametrisation, but implementation is different in compiler and virtual machine (JVM).

- Generic collections are parametrised with one or more classes.

```
Collection<Elementtype>
Collection<Elementtype1, Elementtype2>
```

- E.g., in mapping, we give class of key and class of value.
- Element type must be class, simple types cannot be used.
- When presenting and implementing the collection, we use formal type parameter(s), (any identifier, but we often use "*E*").

```
public class BTreeNode<E> {
    private E element;
    ...
    public E setElement(E element) {
        ...
```

- *E* is element type (class) in the program block of the class (but not outside).
- Limitation: we cannot create directly instances of *E* with *new*(), neither we can create arrays of references to *E*.
  - Circumvent: create an *Object* array and cast it to *E* array, see examples.
  - It is anyway only array of references.

  - If we need to create instances of E, we must use a factory method (*Class.newInstance*()). We seldom need to!
    * Remember that this collection is only a tool to conveniently manage existing instances of E.
- If we need to restrict the element class (usually when needing some properties), we can use declaration <*E extends NeededSuperclass*>, e.g.:

```
public class SortingList<E extends Comparable> {
```

- Then the elements of the collection can be expected to implement *compareTo* method.
- Actually, this is not sufficient in case of *Comparable*. Interface *Comparable* is parametrized to be able to ensure the "other" parameter compatibility for comparisons.
  - http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html
- Thus

```
public class SortingList<E extends Comparable<E>> {
```

- In English: E must implement *Comparable* that it parametrized with E (comparisons against class E).
- If we want to be able to use also subclasses of E in the same list, then:

```
public class SortingList<E extends Comparable<? super E>> {
```

- An alternative to using *Comparable* is to use a separate *Comparator* class.
- Thus, e.g., the sorted set of JavaAPI is declared only as *TreeSet*<*E*>.
  - Then implementation must check at runtime that either *Comparator* is given, or element class implements *Comparable*.
- When using the collection, we give actual type parameter.

```
Collection<ElementClass> MyCollection = new Collection<ElementClass>();
```

- E.g.,

```
LinkedList<String> stringList = new LinkedList<String>();
LinkedList<Set<Integer>> integerSetList =
                    new LinkedList<Set<Integer>>();
```

- If implementation of a collection has also other, helping classes, both must be given the same element class.

```
List<Integer> myList = ...;
...
ListNode<Integer> p = myList.getFirst();
Integer x = p.getElement();
```

Generic methods

- In addition to classes, we can also parametrize single methods.
- Especially useful for "helper" methods.

```
public class Exmp {
    public static <E> boolean areDuplicates(Collection<E> collection) {
        for (E x : collection) {
            ... x.method() ...
        }
    }
...
LinkedList<String> L = new LinkedList<String>( ... );
boolean ok = areDuplicates(L);
```

Java Collection framework

⇒ Includes interfaces, abstract (skeleton) implementations, and actual implementations — all generic.

- Interface defines what a given collection can do (methods to implement).
  - E.g., *Collection*, *Set*, *Map*, *List*.
- Abstract implementation contains much of functionality (as simple generic versions).
  - E.g., *AbstractCollection.addAll*(), *AbstractList*.
  - Basic functionality (e.g., insert, remove, iterations) is implemented in actual implementations.
- General purpose implementations are ready-to-use packages. There can be several implementations/extensions of the same interface/abstract implementation.
  - E.g., *TreeSet*, *HashSet*, *LinkedList*, *ArrayList*.
- `http://docs.oracle.com/javase/7/docs/technotes/guides/collections/index.html`

Iterators

- To implement interface *Iterable*, a class (often *Collection*) must have method

  Iterator<E> iterator() [1]

- which returns an instance of a class that implements interface *Iterator* to iterate over the collection.
  - *Iterator* interface requires operations *hasNext*() and *next*() (and *remove*(), which can be non-functional).
- The class that implements *Iterator* interface, can be located as an inner class of collection, where we can control its usage.
  - Method *iterator*() of collection only creates (*new*) and returns a new instance of this class.
- See Section 8.6 (p. 93) and examples.

7.2 Implementing lists

⇒ There are several ways to implement lists...

- Each of them has pros and cons.
- We cannot quite make interface independent of implementation because behaviour of position varies from implementation to another.

We'll use generic parametrized classes.

Array-based implementation of List

⇒ Store the elements to array without gaps.

- We need to allocate large enough array?
  - After running out of space, either
    - exception (not good), or
    - we must enlarge the space, which is slow, but relatively rare, i.e., average unit time, or
    - use dynamic array implementation (where space does not run out, also unit time in average).

  - Some space is waster.
- Remember indexes of first and last elements.
- Position is array index.
- Insert and remove elsewhere than end is $O(n)$ time.

⇒ Straightforward implementation.

Example 7.1. Array-based list (*ArrayList2*.java).

```
public class ArrayList2<E> {
    private E[] data;                    // references to elements
    private int n;                       // number of elements

    @SuppressWarnings({"unchecked"})
    public ArrayList2() {
        // we cannot create E[] array, we need to create Object[]
        data = (E[]) (new Object[20]);
        n = 0;
    }
    public int EOL() {
        return n;
    }
    public int first() {
        return 0;
    }
    public E getElement(int p) {
        if (p >= n || p < 0)
            throw new ListException("getElement: invalid position " + p);
        return data[p];
    }
    public void insert(int p, E x) {
        if (p > n || p < 0)
            throw new ListException("Insert: invalid position " + p);
        if (n == data.length)                // if array is full, then allocate new
            doubleCapacity();                // larger space and copy old elements
        for (int i = n; i > p; i--)
            data[i+1] = data[i];
        data[p] = x;
        n++;
    }
    public int next(int p) {
        if (p >= n || p < 0)
            throw new ListException("Invalid position " + p);
        return p+1;
    }
    // ...
}
```

Chained array-based implementation of list

⇒ In a simple array-based implementation, remove and insert are slow.

- Why to use a list if it has all the weaknesses of an array?

- Insert and remove should be done without moving elements!
- This is possible by chaining the elements. Then we can skip a removed element using links, and find the next element easily even if it was inserted elsewhere.
  - Chaining is implemented by maintaining the index of the successor element of each element in a separate array.
- Free positions are chained as another list.
  - Initially we need to link all free positions (O(storage_area_size) time).
  - When inserting, remove linking from list of free positions and add to actual list.
  - When removing, remove position from actual list and add to list of free positions.

[*ArrayLinkedList.java*, *ArrayLinkedList2.java*]

```
public class ArrayLinkedList<E> {
    private E[] data;
    private int[] next;
    private int first, last, firstFree;
    public static final int EOL = −1;
    public ArrayLinkedList(int size) {
        data = (E[]) (new Object[size]);
        next = new int[size];
        for (int i = 0; i < size−1; i++)
            next[i] = i+1;
        next[size−1] = EOL;
        first = EOL;
        last = EOL;
        firstFree = 0;
    }
    public void insert(int p, E x) {
        if (p >= data.length || p < 0)
            throw new ListException("Invalid position");
        if (firstFree == −1)
            doubleCapacity();                    // doubles also next[] array
        int i = firstFree;
        firstFree = next[firstFree];
        if (p != EOL) {                  // insert elsewhere than end
            data[i] = data[p];                   // move earlier element to a
            next[i] = next[p];                   // free position
            data[p] = x;                         // replace by the new element
            next[p] = i;                         // successor the the moved element
        } else {                         // insert to end
            data[i] = x;
            next[i] = EOL;
            if (first == EOL)            // in empty list, also the first
                first = i;
            else
                next[last] = i;              // successor of the last
            last = i;
        }
    }
    public int first() {
        return first;
    }
    public int next(int p) {
        if (p >= data.length || p < 0)
```

```
            throw new ListException("Invalid position");
        return next[p];
    }
}
```

⇒ Also in this version *remove*() and *previous*() are $O(n)$, which is not always acceptable.

- *deleteNext*() would be unit time.
- Similarity, we can move the successor of the removed element in position of removed. Then we do not have to update the *next*-reference of the predecessor.
- Removing the last element is anyway $O(n)$.
- Solution: chain the list in two directions.

```
public class ArrayDoublyLinkedList<E> {
    private E[] data;
    private int[] next;
    private int[] prev;
    private int first, last, firstFree;
    public static final int EOL = -1;
    @SuppressWarnings({"unchecked"})
    public ArrayDoublyLinkedList(int size) {
        data = (E[]) (new Object[size]);
        next = new int[size];
        prev = new int[size];
        for (int i = 0; i < size-1; i++)
            next[i] = i+1;
        next[size-1] = EOL;
        for (int i = 1; i < size; i++)
            prev[i] = i-1;
        prev[0] = EOL;
        first = EOL;        last = EOL;
        firstFree = 0;
}   }
```

Benefits of array-based implementation:

- Efficient (at least the basic version).
  - no space for links needed
  - sequential processing efficient
- No need for dynamic memory allocation (or free) with every insert/remove.

Problems

- Limited space / waster space.
- Complex when doubly linked.

Using 3 element object instead of 3 separate arrays is not convenient, not efficient in Java.

Dynamic linked list implementation

Dynamic implementation will overcome the disadvantages of array-based implementation (linear time of insert/remove, enlarging space).

- We shall allocate one list node (and element reference) at a time, free allocation when removing element (in Java, just detach references, freeing is automatic by garbage collection).

- Followers (and predecessors, if needed) are tracked by object references to list nodes.
- Functionality quite similar as in linked array-based implementation.
- Some variants as in array-based implementation.
- Partially the same problems.

One-directional list

- List is a reference to the first node.
- Each node has a reference to the successor node (only).
- Benefits: simple, *next*() and *insertAfter*() in $O(1)$ time.
  - *insert*() can be also made $O(1)$ by moving existing element to the new node and placing the new element to the existing node.
- Drawbacks:
  - *previous*(), *remove*() take $O(n)$ time.
    - *removeNext*() and *removeFirst*() can be made unit time.
    - *remove*() can take the next element to this node and remove the next one. But removing last element would still be $O(n)$.
    - Can be fixed by using two-directional links.
  - Empty list is *null*.
    - User must manually insert first element using *new*, or methods have to be static.
  - Changing the first element of the list changes the list reference — user must handle the list carefully.
    - Can be fixed by using an id-node (see below).

Example 7.2. One-directional linked list without id-node.

```
public class SingleLinkedList<E> {
    private E element;
    private SingleLinkedList<E> next;

    public SingleLinkedList(E x) {
        this.element = x;
        this.next = null;
    }
    public void insertAfter(E x) {
        SingleLinkedList<E> n = new SingleLinkedList<E>(x);
        n.next = this.next;
        this.next = n;
    }
    public E deleteNext() {
        if (next == null)
            throw new NullPointerException("Cannot delete null node");
        E x = next.getElement();
        this.next = this.next.next;
        return x;
    }
    // ...
}
```

List with id-node or sentinel node

- Sentinel node: an extra dummy node in front of all element-containing nodes. Sentinel node contains the reference to the actual first node (if it exists).
- List is a reference to the sentinel node.
- List reference changes only when creating the list, not during other operations.
- Operations need to check whether they process sentinel node or element node.
- Classes *List* and position ("*ListNode*") are actually the same.
  - Compiler cannot check the correct usage.
    - E.g., parameter order in *remove*.
- *DoubleLinkedList2.java*
- We'll return to using same class id-node with tree implementation, Section 7.4 (p. 82).
- Id-node: an instance of a separate list class (e.g., *LinkedList*) that contains references to the first (and last) list node(s) (*ListNode*), and possible other information as well.

Doubly-linked list

- In all one-directional implementations, *remove*() and *previous*() take $O(n)$ time (at least in some cases).
- If/when we need these operations, we should link the list in both directions. Then we have a reference from every node to both successor and predecessor.

Separate class for id-node

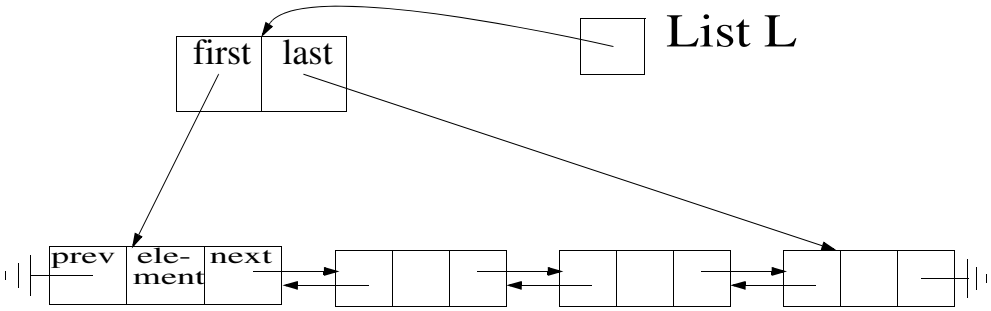


Figure 7.1: Double linked list implementation with separate id-node.

- For clarity (for both implementer and user) we declare a separate type (class) for the id-node of a list.
- Using same (*package*) or placing the node class within the list class helps references to members of the list node.
  - If we would use completely separated classes, need of use of *setX* and *getX* operations distract readability, see *DoubleLinkedList.java*.

Algorithm 7.3. Double linked dynamic list with separate id-node within a package. [*DoubleLinkedList3.java*, *DoubleLinkedList3Node.java*]

```
package List;                                                          1
public class DoubleLinkedList3<E> {                                   2
   private DoubleLinkedList3Node<E> first;                             3
   private DoubleLinkedList3Node<E> last;                              4
   public static final DoubleLinkedList3Node EOL = null;              5
   public DoubleLinkedList3() {                                        6
      first = EOL;                                                     7
```

```
        last = EOL;
    }
    public void insert(DoubleLinkedList3Node<E> p, E x) {
        DoubleLinkedList3Node<E> n = new DoubleLinkedList3Node<E>(x);
        if (p != EOL) {
            n.next = p;                    // elsewhere than end
            n.prev = p.prev;
            p.prev = n;
            if (p == this.first)
                this.first = n;                        // to beginning
            else
                n.prev.next = n;                           // to middle
        } else {
            n.next = EOL;                  // to end of list
            n.prev = this.last;
            if (this.last == EOL)
                this.first = n;                        // to empty list
            else
                n.prev.next = n;                           // to non-empty list
            this.last = n;
    }  }
    public E delete(DoubleLinkedList3Node<E> p) {
        if (p == EOL)
            throw new NullPointerException("Nonexisting node");
        E x = p.element;
        if (p.next != EOL)
            p.next.prev = p.prev;                          // other than last
        else
            this.last = p.prev;                        // remove last
        if (p.prev != EOL)
            p.prev.next = p.next;                          // other than first
        else
            this.first = p.next;                       // remove first
        p.next = EOL;                              // detach removed node
        p.prev = EOL;                              // from list
        return x;
    }
}
```

```
package List;
public class DoubleLinkedList3Node<E> {
    protected DoubleLinkedList3Node<E> prev;
    protected DoubleLinkedList3Node<E> next;
    protected E element;
    public static final DoubleLinkedList3Node EOL = null;
    protected DoubleLinkedList3Node(E x) {
        prev = EOL;
        next = EOL;
        element = x;
    }

    public DoubleLinkedList3Node<E> prev() {
        return prev;
    }
```

```
    public DoubleLinkedList3Node<E> next() {
        return next;
    }
    public E getElement() {
        return element;
    }
}
```

Good:

- Unlimited space, no wasted space (except for references).
- All operations are $O(1)$ time.

Bad

- A lot of *new* calls and corresponding frees (which are, however, fast).
- Position concept still a bit unclear (but better than in array-based implementation).

7.3 List special cases

⇒ To implement stack, queue, deque, and ring, we can apply list implementation techniques.

Often simple implementation suffices.

- Stack, Queue: linking in one direction suffices.
- Stack, Queue, Deque: no insert to middle
  ⇒ array works well if memory allocation in blocks is not a problem.

Example 7.4. Elements of a stack can be stored to an array so that the bottom element is in index 0 (or 1) and stack grows towards larger indexes. In addition to storage array we need also information (index) of location of top of the stack but there is never need to move elements as new element is always added on top of previous topmost element. However, when storage array gets full, we have to allocate larger array and move elements. Or, create a list (stack) of storage arrays.

Queue and deque in array

⇒ A queue crawls away from the beginning of array.

- We should not move remaining elements as it would take linear time.

⇒ A deque can crawl to either direction.

- Use the array as a ring:
  - $A.first().prev() == A.last()$
  - $A.last().next() == A.first()$
- Remember indexes of both head and tail.
- When head == tail, queue/deque is empty.
  - Or overfull; we need to be able to differ these cases.
- Exercise.

Stack, queue, deque, and ring dynamically

- Like list, we need two-directional links only in deque and ring (ring can be one-directional if we limit operations).
- Exercise.

"Unlimited" (dynamic) array

- Actual storage area in pieces, index of pieces.

⇒ Index array contains references to actual storage area pieces.

index[i / m][i % m] 1

- Pieces are allocated only when needed.
  - Assignment operation must check if the piece has been allocated or not.

If index grows too large, split it also to pieces, and create an upper level index.

- 10 000 elements in each piece.
- 10 000 references in each intermediate index.
- 10 000 references in main index.
- 1 000 000 000 000 elements.
- Wasted space min 10 000 words, max 100 010 000 words.
- Lowest levels(s) can be stored in mass storage, see B-tree (DSA II).

7.4 Implementing trees

⇒ Develop further from list implementations.

- Array, if we know the maximum size and shape of tree, and we do not need rotations.
- Dynamic, if we need to modify (rotate) tree, or we do not know the height.

General tree vs. binary tree:

- Both have two "younger" relatives to link.
- Either can be implemented using the the operations of the other.
- Here we discuss binary trees.

Binary tree in an array

- Static implementation: we allocate large enough array:
  - Assume tree is full (i.e., reserve space for all possible nodes).
  - We need 2^{max_height} space.
    - Relatives can be found by index calculation:
    - root in index 1 of the array
    - children of root in indexes 2 and 3
    - children of 2 in indexes 4 and 5
    - children of 3 in 6 and 7
    - parent of n in $n/2$.
    - left child of n in $2 \times n$
    - right child of n in $2 \times n+1$
  - All operations are unit time (except possibly initialization of array).
    - To check if a node exists, we need to be able to recognize empty slots (if/when node existence operation is needed).
    - Alternatively, if the tree if full, it suffices to count the number of nodes (see heap below).

  - Efficient when tree is about full (not too high).
  - No space wasted for links, etc.
  - General tree can implemented as "tilted" binary tree.
- Linked array implementation:
  - Like linked array implementation of list.
  - Separate array(s) hold locations of children.
  - Works also for general tree.

89

Dynamic tree implementation

- Same choices as with list:
  - id-node or not?
  - same or different class for id-node and tree node?
  - are tree and node compatible?

Tree and node as compatible types/classes

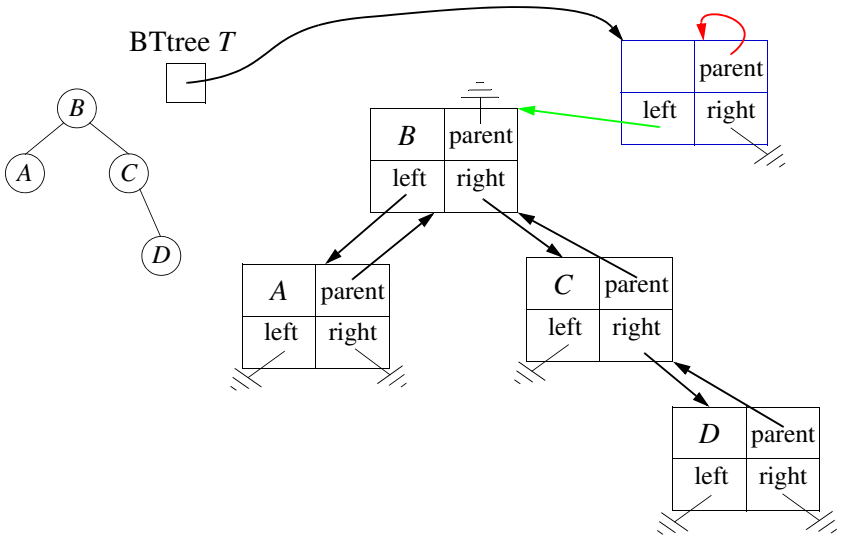
- Especially in recursive algorithms, it would be convenient if we could call any node as a tree and vice versa..
  - Starting (wrapper) method could often left out when single parameter is enough.
- Even more convenient when building and combining trees.
- Three possibilities for compatibility:

1. Implement tree without id-node, when root node represents the whole tree.
   - Good
     - simple, no special cases
   - Bad
     - no extra information about the tree (unless in every node)
     - root node can/will change $\Rightarrow$ value of collection reference changes
2. Use same class to represent the id-node as to represent tree nodes.
   - Good
     - collection reference does not change
   - Bad
     - implementation must differ between id-node and tree node
     - less clear, requires disciplined implementation and use
3. Define upper class tree component, and make tree and tree node to extend that
   - Good: easier type checking.
   - Bad: all methods need to check which type is under processing.

Example 7.5. Binary tree with same class id-node.

- Mark the id-node with $T.parent = T$, and place the reference to actual root to $T.left$. $T.right$ and $T.elem$ are unused (unless something special is needed).

Algorithm 7.6. Binary tree with same class id-node.




```
public class BTree1<E> {                                                    1
    private BTree1<E> left;                                                 2
    private BTree1<E> right;                                                3
    private BTree1<E> parent;                                               4
    private E element;                                                      5
    public BTree1() {                          // creating new tree         6
        left = null;                            // root                     7
        parent = this;                          // mark of id−node          8
        right = null;                                                       9
        element = null;                                                     10
    }                                                                       11
    public BTree1(E x) {                  // new unconnected tree node      12
        left = null;                                                        13
        parent = null;                                                      14
        right = null;                                                       15
        element = x;                                                        16
    }                                                                       17
                                                                            18
    public BTree1<E> getRoot() {                                            19
        if (parent != this)                                                 20
            throw new TreeException("Not a BTree1");                        21
        return left;                                                        22
    }                                                                       23
    public void setRoot(BTree1<E> n) {                                      24
        if (parent != this)                                                 25
            throw new TreeException("Not a BTree1");                        26
        left = n;                                                           27
        n.parent = null;                                                    28
    }                                                                       29
    public BTree1<E> getLeftChild() {                                       30
        if (parent == this)                                                 31
            throw new TreeException("Not a BTree1 node");                   32
        return left;                                                        33
    }                                                                       34
        ...                                                                 35
}                                                                           36
```

8 Implementing sets

⇒ Sets are implemented depending the needs of the application.

- Which operations are needed?
  - Set, mapping, priority queue, bag?
- Which operations are not needed?
  - E.g., operations of several sets?
- Which operations are only rarely used?
- Which operations must be (especially) efficient?
- Which properties the element type has (or what we can expect)?
- Do we know the size of value domain (universe), and/or the number of elements?
- Set operations p. 46.

⇒ Time complexities and limitations of different implementations vary quite a lot!

Implementing, or using sets?

- Most modern programming languages/platforms have some built-in or add-on set implementation.
  - E.g., *java.util.*Set*, glib, .NET dictionary, perl, Python, sh, etc.
  - Relational database is a set (relation / mapping).
  - Time complexities are usually good, but often poorly documented.
- Usually it is best to use these ready implementations.
  - Occasionally we'll encounter such special combination of requirements that does not have a ready solution.

⇒ Essential skill is to choose the correct set implementation for each application and use it efficiently (correctly).

- Correctly == not wrong (operations).
- To be able to choose the correct implementation, we need to understand their functionality.
- In sets, the choice is usually between hash table and tree.
  - In databases, index implementation must be chosen between these.
- Very similar importance of understanding as between list and array.

8.1 Simple sets

⇒ Sometimes the universe (all possible elements) is some finite (smallish) interval of integers (or otherwise enumerable):

- We can use a bit vector (boolean value vector).

  if (B[i]) *contains* **else** *not_contains* [1]

- *contains*, *insert*, and *remove* in $O(1)$ time.
- Other $O(|U|)$ (U is the universe of possible elements).
- Also non-continuous finite subset of integers can be handled, if we take it into account in operations (or storage structure).

⇒ If a bit vector cannot be used (elements cannot be enumerated, universe too large, etc.), then we have to store the actual elements within a data structure.

- Linked list

  - Sorted or non-sorted.
  - *contains*, *insert*, *remove*, (*max*): $O(n)$, where n = number of elements.
  - Two set operations $O(n)$ (if sorted) or $O(n\log n)$ (if unsorted).
- Array
  - Either store elements densely and sorted, or index directly (hashing).
  - Densely sorted: *contains* $O(\log n)$, *insert*, *remove* (and two-set operations) $O(n)$.
  - Old elements sorted, new elements unsorted: basic operations in $O(\sqrt{n})$ (average) time.
  - Using hashing *insert*, *remove* $O(1)$, two-set operations $O(n+m)$ .. $O(n\log n)$.
- (Binary) tree
  - Balanced basic operations in $O(\log n)$, two-set operations in $O(n)$.

8.2 Set special cases

Dictionary

⇒ Concentrate on searching.

- Possibly also insert, maybe remove.
- Not union, difference, etc.
- If inserts and removes are not done, then sorted array is quite good: basic operations $O(\log n)$ using binary search, very simple and space-efficient data structure.
- If we need insert and/or remove, use hashing or balance tree.

Implementing Mapping (function)

- As any other set.
  1. calculate function
  2. array
  3. hashing
  4. tree

Implementing Bag

- As other sets, just ensure that all equal elements can be found.
- Closed hashing is not good if the same element can occur (very) many times.
  - If *equals*() is true, then *hashCode*() must be same.
- Open hashing in Java8 with long lists changed to binary trees would suit well.
- Removal either one element at time, given number of elements, or all equal elements.
  - Usually different operations.

8.3 Hashing

- Storage area is an array or a file, indexed $0..m-1$.
- For each (inserted/searched) element we calculate an address in storage area (so called home address, or default address).
  - Integer $0..m-1$ calculated quickly and deterministically from the key.
  - Hash function h: $0 \leq h(key) < m$.

- Sequential (application) order does is not preserved.
  - Ordered iterations (or *min*, *max*) are not possible without extra structures, especially not in mass memory. For dual solution, see *java.util.LinkedHashSet*.
  - "Random" (hash function) order iteration in time $O(n+m)$.

⇒ In an ideal situation each element of the set would be hashed to a different address.

- In practice, this is highly improbable if/when key is, e.g., a string. Own address for each key is actually impossible if/when number of slots is (much) less than the number of possible keys.
- Thus, two or more elements will be mapped to the same home address.
  ⇒ We must prepare for collisions!

⇒ There will be collisions surprisingly often:

- 23 values randomly to 365 bins: collision with probability 50,73% (birthday paradox).
- n elements to n bins: some bin will get $\Theta(\log n)$ elements (with high probability).
- Hash function should be selected to avoid very uneven distribution of elements (very many in same (or near) home address while having a lot of empty addresses elsewhere.
  - Use whole (or a significant part of) the key.
  - Ensure that whole hash table $(0..m-1)$ is in use (with about equal probabilities.
  - Avoid formulas that cause collisions by regularities (e.g., if both m and a multiplier are powers of 2, they'll cancel each other).
- Randomness helps to avoid catastrophes caused by regularities.
- It is, however, impossible to define a hash function that could produce random values out of non-random input.

⇒ In case of collision, the element must be placed somewhere.

Closed hashing (open addressing)

- Storage area $(0..m-1)$ contains all elements/references, no further space.
- When inserting an element, if the home address h(key) is already reserved, the element will be stored to the next free address.
  - Next address: either next index (linear probing), or some other unambiguous rule, see double hashing below.
- When searching, if the element is not in home address, then the element can be in next addresses.
  - It is enough to (and we have to) search until next free address.

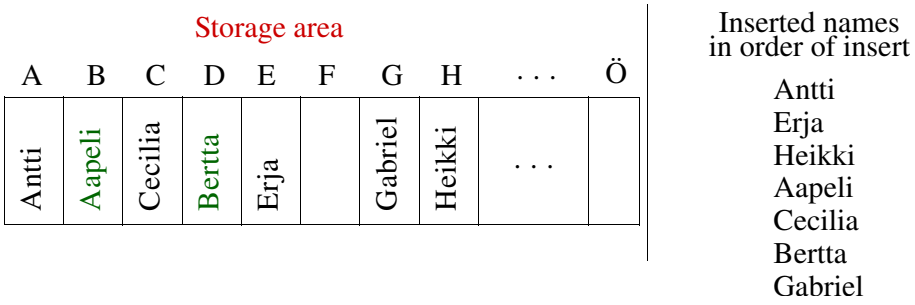


Figure 8.1: Closed hashing, first letter as hash function (poor, but illustrative).

- Removal takes some extra care, either:
  - We would have to move elements from next positions (not very handy), or preferably:
  - Instead of actually removing, just mark the element removed (replace the element with a special "removed" element).

    - Search must go past addresses that have removed mark (as an element that was added before removal may have been put further).
    - Add is made to first free address (removed, or empty).
- Good: very simple structure (especially in a file).
  - Even if we would have to browse next addresses, it is still sequential access, which is faster in memory, and much faster in mass memory.
- Bad: if storage area (almost) fills up, searches will get longer.
- When searches will get longer, we should hash everything again on larger storage area and/or using different hash function.
  - Closed hash table should not be allowed to fill even close to full.
  - Simply: rehash when, e.g., 2/3 of the storage is in use.
  - Even smarter: monitor longest consecutive reserved sequences.

Open hashing (external hashing, chained hashing)

- For each home address, there is a list to hold all the elements belonging to the address.
  - Slot in hash table contains a reference to the first node of the element list.
  - Or sometimes: hash table contains one element and reference to the first list node (which contains the second element).
    - Useful to avoid one memory reference if we can create an array of elements. In Java we can't, thus it is not useful.
- Storage area is unlimited, but searches obviously slow down when lists grow longer.

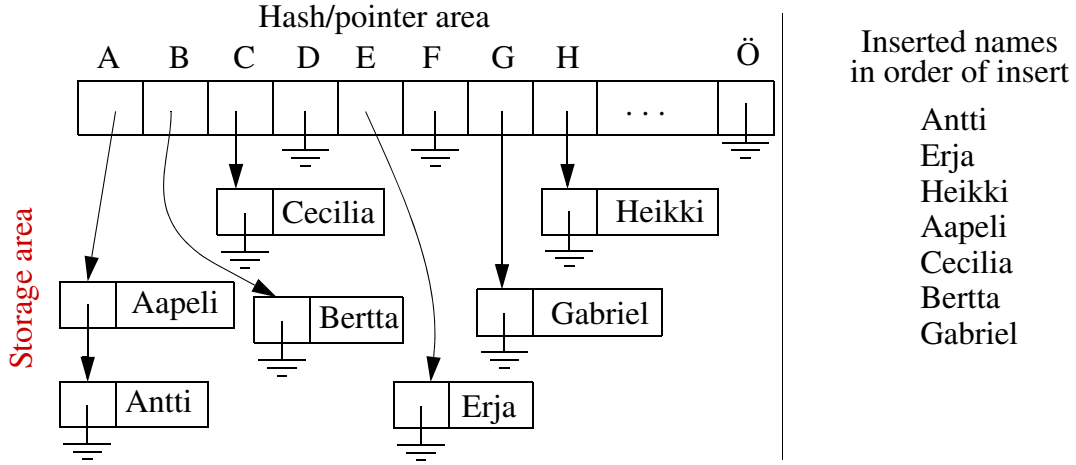


Figure 8.2: Open hashing, first letter as hash function (poor, but illustrative).

- Lists can be replaced by trees, but preferably use larger hash table to keep lists short.
  - Java8 *HashMap*: if list grows too long (>8), list will be converted to a balanced binary tree.
- Two or more memory references, depending implementation.

On designing hash functions

- Use whole key to calculate the value of hash function.
- Different parts of key with different multipliers (to avoid summing to the same pile).
- Avoid regularities.
- Use prime numbers as multipliers, or at least ensure that multipliers and m are mutually not divisible.

- Ensure that all indexes $0..m-1$ are possible (and about equally probable).
  - Even if m grows or decreases.
- See, e.g., *java.lang.String.hashCode()*, *java.util.HashMap.hash()*.

Double hashing

- Choose two (or more) completely different hash functions.
- If the first hash function gives a collision, calculate the address given by the second hash function and store the element there.
  - If also second hash function gives a collision, use linear probing (or even a third hash function).
  - We can also use initially linear probing, and only after a threshold use the secondary hash function.
- A possible pile-up according to one hash function should be spread using the second function.
- Search using the same rules as insert.
- In closed hashing straightforward (as linear probing).
- In chained hashing, we can maintain information about the length of list and resort to the second hash function after a threshold value.

Randomized hashing

- Use randomly selected hash function.
  - If it does not work well, then choose a new random hash function, and rehash whole hash table.
- With high probability, rehash is needed so seldom that it does not impact the average time complexity.
- We can also randomize double hashing.
- Suits well for asymptotic proofs...

Fibonacci hashing

- Example on multiplicative hashing and use of an irrational number.
- A method to hash integers evenly on range $0..m-1$.
- Integers 1, 2, 3, ...so that the next integer is always located to the widest remaining gap (in proportion of the golden ratio).

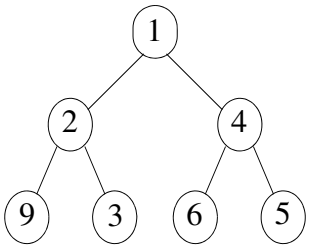
$$h(k) = \left\lfloor m \times \left(\left(\frac{\sqrt{5}-1}{2} k \right) \mod 1 \right) \right\rfloor \tag{8.1}$$

93

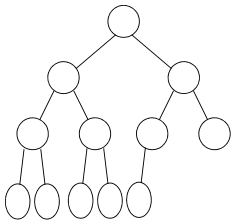
8.4 Implementing priority queue

Using a list or array: insert or remove is $O(n)$. Using a balanced binary tree, *insert*, *remove*, and *minimum* would be $O(\log n)$, but there would be unnecessary amount of rotations. Here we'll show a solution that has much simpler storage and lower constant factors.

Balanced partially pre-ordered binary tree (heap)



- Partially pre-ordered:
  - The priority of each element is better than priorities of its both children.
    - Thus, also better than priority of all descendants.
  - Order of siblings does not matter.
  - Root has the best priority.



- Balanced:
  - With the possible exception of the lowest level, every level is full.
  - The nodes of the lowest (non-full) level are located to as much left as possible.
  - Insert position is always the leftmost free position of the lowest level.
  - Remove position is always the rightmost occupied position of the lowest level.
  - Height of the tree is $\lceil \log n \rceil$.

Priority queue operations using heap

removemin

1. Store the root element as a result (best priority) ($O(1)$).
2. Move rightmost element of the lowest level in place of root (number of elements is decreased by one) ($O(1)$).
3. New root will be lowered down: swap with better priority child until swap is no more needed ($O(\log n)$).

insert

1. New element in added as new rightmost element of the lowest level ($O(1)$) (leftmost free position).
2. New node will be raised up: swap with parent if parent has worse priority than the new node. Repeat until swap is no more needed ($O(\log n)$).

replace() operation (combined *removemin* and *insert* is about the same as *removemin* (fixdown), thus saves time if used.

Simple operations, very small constant factor.

How to find insert/removal position?

- In dynamic implementation, either:
  - Chain nodes by level, or
  - Maintain information about rightmost element of the lowest level (can be maintained in $O(\log n)$ time).

Array implementation: *Element*[*max*+1] A (if index 0 is used, then $+-1$ everywhere).

- Root: $A[1]$
- Children of root: $A[2]$ and $A[3]$
- Children of 2: $A[4]$ and $A[5]$
- Children of 3: $A[6]$ and $A[7]$
- Left child of i: $A[2*i]$
- Right child of i: $A[2*i+1]$
- Parent of i: $A[i \ / \ 2]$ (integer division)
- Rightmost element of the lowest level $A[n]$, where n is the number of elements.

Array implementation is very efficient for both space, and time.

- Very small constant factors.

Example 8.1. Heapsort can be done with constant extra space. When moving elements to priority queue, the growing beginning of array is used to store the heap and the shrinking end of array to store remaining array. Array is shrunk at the same pace as the priority queue grows. When all elements are in priority queue, the whole array is storage for priority queue.

Respectively, when removing elements from priority queue, the size of queue is reduced and queue is replaced in the array by sorted elements. Elements must be removed from priority queue from largest to smallest.

By nice coincidence the elements in both array-to-heap and heap-to-array are in correct position (insert/remove position of the heap). Thus it enough to do a swap (on case of heap-to-array) and fixup/fixdown for each element. Exercise?

83

8.5 Search trees

⇒ Let us concentrate on search, insert, and remove operations.

- All in $O(\log n)$ time.
- Dynamic size of set.
- Any element type (*compareTo*() or *Comparator* suffices).
- Iteration in order of elements in $O(n)$ total time ($O(1)$ average time), (some *next*() operations are $O(\log n)$ unless we use extra linked list of nodes).
- Using iterators, the set operations in $O(n)$ time.

⇒ In-order (almost) balanced binary tree.

- Search is easy (as we saw earlier).
- Maintaining balance during insert/remove takes some extra work, but is still possible to do in $O(\log n)$ time.
- Instead of perfect balance, it suffices to maintain reasonable balance, as long as tree height is $O(\log n)$ (i.e., at most 1.5 or 2 times the optimal $\log n$.
- There are several approaches, most of them allow small imbalances, in case of larger imbalance, we balance the tree.

AVL-tree [Adelson-Vel'skij & Landis]

⇒ For each node, the height difference of left and right sub-tree may not be larger than one.

Node balance:

- In each node, we maintain information about balance, not height:
- Balances of each node are updated from bottom to up (only!) along the path up whenever we insert or remove a node:
  - If the balance of a node changes from 0 to + or –, (i.e., height of either (but not lower) sub-tree increases) changes also the balance of the parent based on whether this node is a left or right child.
  - If balance changes from + or – to 0, propagating up the change stops (height does not increase).
  - Respectively, in a remove operation, the height of either sub-tree may decrease.
- Note that we do not calculate heights, only height differences (balances).
  - Height differences are updated locally, in unit time, we never need to (re)compute the whole (sub)tree.

⇒ Balance is broken, if it is −− or ++ (−2 or +2).

- If the balance of a node breaks because of insert, we fix only the broken node.
  - We perform a rotation around the imbalanced node.
  - There a two types of rotations (and in two directions):
    - If the balance of a node is −− and the balance of left child is –, we perform rotation to right.
    - If the balance of a node is −− and the balance of left child is +, we perform double rotation to right (left-right rotation).
    - If the balance of a node is ++ and the balance of right child is +, we perform rotation to left.
    - If the balance of a node is ++ and the balance of right child is –, we perform double rotation to left (right-left rotation).
- Any rotation stops propagating up of the imbalance, just one rotation suffices.
- We do not touch the balance of other nodes (except those participating the rotation).
- Note that all rotations can be done in unit time (4-8 reference updates)!
- Removal is similar, also in $O(\log n)$ time (although, we may need rotation on several levels).

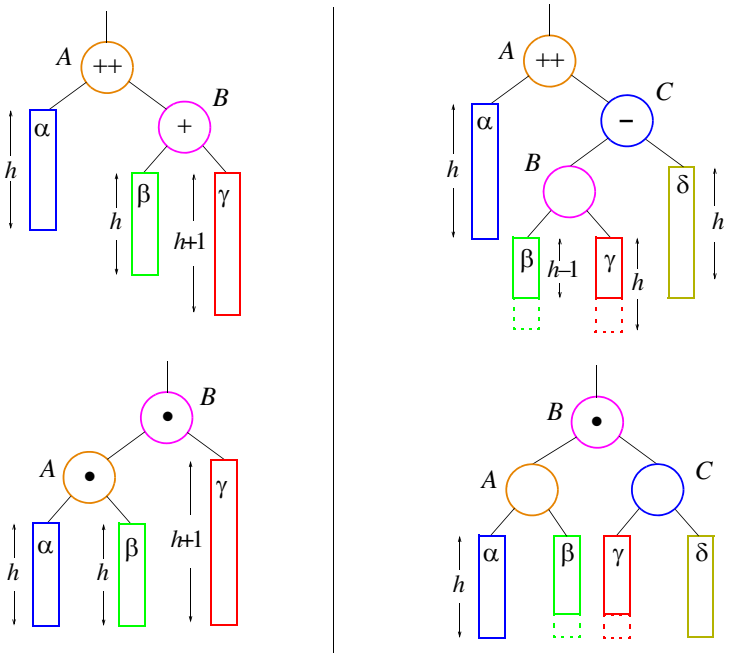


Figure 8.3: Rotations of AVL tree

Other similar data structures

- Red-Black tree (see CLR, JavaAPI).
- B-tree (especially for mass storage, see DSA II).
- 2-3(-4) tree (special case of B-tree for main memory).
- "Self-balancing" tree, Splay tree, etc.

42

8.6 Implementing iterations

- See iterators p. 22, p. 48.

iterator(), *hasNext*() and *next*() use implementation structure:

- Iteration variable refers (somehow) to the element/node under processing (or sometimes to the successor).
  - Index of array, node of list/tree, etc.
- .*iterator*() seeks the reference to the first (or any) node.
- From user perspective, *hasNext*() checks if iteration variable refers to a valid element/node (i.e., do the position actually have an element or not).
- From user perspective, *next*() "moves" iteration variable forward and returns the element that was hopped over.

Straightforward for lists and arrays, see *ArrayList2.java*, *SortedBag.java*.

In hash table, we have to take into account the empty slots, and in chained hashing iterate the lists (exercise).

- If we want unit time *next*() and maintain iteration order after rehash, link the elements, see *java.util.LinkedHashSet*.

Inside implementation

- If storage structure is not simple, it can be "difficult" to know if the current element/node is the last one or not.
- Then, sometimes it is easiest if (constructor and) *hasNext*() skip over empty slots and seek the next element.
  - *next*() only returns the current element and skips it.
    - If we want to support calling *next*() without *hasNext*(), then *next*() must check if the current position is valid, and if not, call *hasNext*() itself.
  - To implement *remove*() we need to maintain information about last returned element (node/position).
- See *SortedBag.java*.
- Alternatively, maintain reference to both previous and next element/node.
  - *next*() (and constructor) takes next as previous and seeks a new next (if it exists). *hasNext*() is trivial.

Traversing trees

- Straightforward in-order (or, pre-order, post-order, or by level).
- .*iterator*() (i.e., constructor of the implementation of *Iterator*) seeks the first node of the tree.

- *hasNext*() seeks the successor in tree, *next*() returns it, or vice versa.
  - Whole iteration in $O(n)$ time (in binary tree, follow each parent-child relation down and up, total $2n$ times).
  - In a balanced tree, two *next*() operations take about log$n$ time, other take less, average anyway $O(1)$.

Options, e.g.:

- Iteration by level, maintain queue in iterator.
  - Not in in-order.
- If in-order traversal must have all *next*() operations in unit time: Link tree nodes to wanted order (an extra next reference in each node).
  - Maintenance will be more difficult, but can still be done in logarithmic time.

Changing collection outside iteration during iteration

- Allowing requires great care and strict prerequisites.
  - Or, preparation within storage structure.
  - Collection on must be able to update the value of iteration variable whenever the collection is changed (*insert*, *remove*).
  - In practice, all iteration variables must be stored (referenced) within collection.
  - To allow several simultaneous iterations, we need a list of active iterations.

Noticing the change of collection during iteration

- Java collections use so called *FailFast* policy if collection is changed during iteration (by other than methods of this iteration).
  - If an iteration finds a potential change, it throws *ConcurrentModificationException*.
- Each collection has member *int modCount* which is increased always when a change is made (changes can be done only using the methods of the class).
  - Iterator maintains a copy of the value of *modCount*. If the value has changed, it knows that a change has been made and it throws the exception.
  - Own methods of the iterator (*add*(), *remove*()) update the value of the copy of *modCount*.
- Change is obviously found only when the iterator is used after the change.
- Operations that do not change the storage structure (e.g, *set*()) do not increment *modCount* either.

Two set operations

- *union*, *intersection*, *difference*, etc.
- Straightforward *foreach* – *add*/*remove* works well for hash table in main memory, for trees it is not optimal.
- Instead, read source sets in order to list or array ($O(n)$ time), "merge" (depending the operation) lists ($O(n)$) and create new storage optimised (balanced) ($O(n)$).
  - If data structure supports traversal in order (e.g., tree), we can do the "merge" phase already in initial step.
  - Balanced tree is easy to form from an array in linear time, but it is also possible from a list.

8.7 Implementing graphs

⇒ Formally graph is composed of sets of vertices and edges, so the implementation could use set operations.

- In practice, we seldom use two set operations with graphs. Thus simple implementation suffices.
  - Vertices are stored to a list or array (or tree).
  - Edges are store as lists for each vertex.
- Alternatively, store separately vector of vertices, and matrix of edges.
- See DSA II.

9 Literature

1. Aho A. V., Hopcroft J. E., Ullman J.D.: Data Structures and Algorithms. Addison-Wesley, 1983.

2. Cormen T. H., Leiserson C. E., Rivest R. L.: Introduction to Algorithms. MIT Press 1990.

3. Hämäläinen A.: Tietorakennekirjasto Javalla. Joensuun Yliopisto, Tietojenkäsittelytieteen laitos, 2005.

4. Knuth D. E.: The Art of Computer Programming, Volumes 1-3, (2-3ed). Addison-Wesley, 1997-1998.

5. Sun Microsystems: JavaTM 2 Platform Standard Edition 5.0 API Specification. http://java.sun.com/j2se/1.5

6. Weiss M. A.: Data Structures and Algorithm Analysis in C. Addison-Wesley, 1997.

7. Weiss M. A.: Data Structures and Algorithm Analysis in Java. Addison-Wesley, 1999.

Existing data structure/algorithm libraries

- http://www.cs.sunysb.edu/~algorith/
- http://www.mpi-sb.mpg.de/LEDA/
- C++
- 100+ classes
- Commercial
- Eiffel, Oberon, Modula-3 -libraries at SoC.
- Numerical Recipes in [Fortran [77|90] | C | Pascal (| Basic | Lisp | Modula-2 | IDL | Mathcad]
- http://www.nr.com/
- Animations: Jeliot, Polka, XTango, etc.
- DSA text/handbooks, especially Cormen, Leiserson, Rivest.
- http://docs.oracle.com/javase/7/docs/api/