Revision sheet for CSCM12 (AY 24-25)

This is meant to be a checklist of some of the main things that may be useful to retain for the exam from the lectures and the labs. This is **not** an exhaustive coverage of all topics in the module, but most should be at least touched on. If something is not marked explicitly as out of scope for the exam, you should be prepared to know that for the exam. This does not follow the order of the slides and there is a bit of overlap/repetition.

This is the second iteration of this sheet, cobbled together from the lecture/lab materials. Feedback/suggestions for impovements welcome!

1 Complexity analysis

1.1 Measuring the efficiency of programs

- Two quantities can typically be roughly estimated theoretically without having to run an algorithm through a computer and do benchmarks: asymptotic **time complexity** and asymptotic **space complexity**.
- **Time complexity** is typically proportional to the number of steps one takes by running a program by hand with pen and papers. Most operations can be assumed to take one step; some notable exceptions are allocating arrays (the time this takes is proportional to the size of the array) and calling auxiliary functions.
- **Space complexity** corresponds to the extra memory required to run a particular function on top of what was allocated for the argument[1]. This is obtained by summing the size of the memory representation of all the variables simultaneously declared in a block of code during the execution; for an `int` and other basic datatypes, this will be constant, but not for complex datatypes and arrays.
- We focused on theoretical asymptotic analyses in these lectures; they have some advantages and inconvenient:
  - They have some degree of imprecision in the sense that they only give results only **up to a multiplicative constant**. This means in particular that they are typically relevant only when your input is getting large.
  - However it also means that they have **wide applicability** and will hold if you switch platform/language/compilers etc.
  - They can be made **a priori** and do not require any testing. It means that in particular they can be used to quickly inform design decisions.
- We abstract time/space complexity as function taking an input size to a running time/memory consumption. We usually consider the **worst-case** complexity, i.e., we take the function

$$n \longmapsto \max_{I \text{ of size } n} \text{running time over } I$$

[1] So a function can have space complexity of $\mathcal{O}(1)$ while being passed an array of arbitrary size as input.

It may be of interest to also look at **average-case** complexity (where the max is replace by an expectancy[2])

- Another way of measuring perfomance in practice is via actually benchmarking (i.e. testing on a variety of inputs and checking for execution time and memory consumption). One gets precise results if things are set up right, and aspects that are ignored by an asymptotic analyses such as the behaviour of compiler optimisations and the runtime environment that influence the "constant factor" that $\mathcal{O}$s abstract away.
- Sometimes when operating over a datastructure, there might be uneven time spent across every call to an operation, like with e.g. dynamic arrays: most operations will actually run in constant time, except in the rare case when a reallocation happens. **Amortized complexity** is a notion that allows to talk about these scenarii (i.e., mathematically, it is fine to assume that adding an element to a dynamic array is $\mathcal{O}(1)$ to compute an aggregate complexity in an algorithm if we start from an empty object).
- There is a bit of an annoying discrepancy between integers in the mathematical world and on the idealized computer: an integer in java will typically be bounded by 2^{64} and we consider all arithmetic operations are thus $\mathcal{O}(1)$, while in theory, for asymptotic analyses to be valid, you may want to have unbounded integers (and the size of n is in $\Theta(\log(n))$). We sweep that issue under the rug for this module[3]. There is also a similar issue with built-in arrays.

1.2 Asymptotic analysis

- Asymptotic means "in the limit"; it means those kind of analyses will typically only work for large value of the arguments of a function, up to a multiplicative constant.
- Here we will assume we deal only with non-decreasing functions $\mathbb{N} \to \mathbb{R}$ or $\mathbb{R}_{>0} \to \mathbb{R}_{>0}$.
- To talk about complexity analysis, we use the following notations[4]

Definition 1 ($\mathcal{O}, o, \Theta$ and Ω notations)**.** *In the following, f and g are function in a single variable foing to $+\infty$:*

- $f(n) = \mathcal{O}(g(n))$ *means* $\lim_{n\to+\infty} \frac{f(n)}{g(n)} < +\infty$
- $f(n) = \Omega(g(n))$ *means* $0 < \lim_{n\to+\infty} \frac{f(n)}{g(n)}$
- $f(n) = \Theta(g(n))$ *means* $f(n) = \mathcal{O}(g(n))$ *and* $f(n) = \Omega(g(n))$
- $f(n) = o(g(n))$ *means* $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = 0$

- A maybe more intuitive reformula of $\mathcal{O}$ (similar reformulations exist for the others as well): in case $g(n) > 0$ for all n, $f(n) = \mathcal{O}(g(n))$ if there exists a constant $K > 0$ such that for every n, $f(n) \leq Kg(n)$. In particular, $f(n)$ is a $\mathcal{O}(1)$ if it is bounded.

[2] This requires setting a probability distribution over the inputs; a typical choice would be the uniform distribution.

[3] The rationale is that 1) the induced inaccuracies are not noticeable in implementations since we typically use often these fixed integers in practice and 2) if we wanted to be rigorous, the inaccuracies would lead to only log factors in most cases which are not a huge deal.

[4] For this module we take all our limits at $+\infty$, but you may find these notations elsewhere when the asymptotic analysis is done at other limit points in other kind of math/physics.

- There are a number of techniques that allow to "compute up to a $\mathcal{O}$" quickly, but if they don't apply, you should ultimately rely on spelling out constants/over-approximations[5].
- Some basic tips for computing with $\mathcal{O}$s:
  - If $f(n) \leq g(n)$ then $f(n) = \mathcal{O}(g(n))$
  - $f(n) = o(g(n))$ impies $f(n) = \mathcal{O}(g(n))$
  - for any $k > 0$, $kf(n) = \mathcal{O}(f(n))$
  - If $f(n) = \mathcal{O}(g(n))$ and $g(n) = \mathcal{O}(h(n))$ then $f(n) = \mathcal{O}(h(n))$
  - $\log(n)^k = o(n)$, $n^k = o(2^n)$ for any constant $k \in \mathbb{R}^+$
  - $n^k = o(n^{k'})$ for $k < k'$
  - $f_1(n) = \mathcal{O}(g_1(n))$ and $f_2 = \mathcal{O}(g_2(n))$ together imply $f_1(n)f_2(n) = \mathcal{O}(g_1(n)g_2(n))$
  - If $f(n) = \mathcal{O}(g(n))$, then $f(n) + g(n) = \mathcal{O}(g(n))$
- Most functions we will encounter will consists of sums/multiplications of polynomials, logarithms and exponentials. Because of this, $\mathcal{O}$ computations will typically be quite easy for this module: develop the expression and select the asymptotically bigger terms.
- To give you a rough sense of scale: the logarithm of a number is its number of digits, and an exponential is to a linear function what a linear function is to a logarithm. In practice, this means that **logarithmic terms** are going to be barely noticeable while any exponential blow-up will make a function unrunnable but for the smallest of inputs[6].
- Those notations make sense for functions in several variables $\mathbb{R}^k \to \mathbb{R}$. The adaptation consists in taking a good notion of limit:

  Definition 2 ($\mathcal{O}$ in several variables)**.** *$f(x_1, \ldots, x_k)$ is a $\mathcal{O}(g(x_1, \ldots, x_k))$ if for every $h : \mathbb{N} \to \mathbb{R}^k$ such that $\lim_{n \to +\infty} ||h(n)|| = +\infty$ we have that $f(h(n))$ is a $\mathcal{O}(g(h(n)))$.*

  In practice all of the rules above also apply when dealing with computing $\mathcal{O}$ for multivariate functions.

1.3 In practice: useful techniques for over-approximating

- For simple loops, one can count the number of iterations and bound the complexity of running through the body for all possible values of the variables, and then multiply the results
- Sometimes (and this is typically the case when dealing with `while` loops with non-trivial termination conditions), one has to figure out a smarter invariant that will decrease with each run of the body of the loop.
- Recursive algorithms will often lead to recursive equations. While you are not expected to be able to rigorously solve them, you should be able to use the following recipes for the typical cases we encounter (f is assumed to be non-decreasing):

[5] For instance, if you have a sequence $u_{n+1} = u_n + 1$, there is a fallacious "proof" that $u_n = \mathcal{O}(1)$: by induction, $u_0 = \mathcal{O}(1)$ and assuming $u_n = \mathcal{O}(1)$, then $u_{n+1} = u_n + 1 = \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1)$. This is wrong and does not hold up to scrutiny if you replace the inductive hypothesis

[6] To nuance that, bearing in mind that worst-case scenarii are not necessarily typical: for instance, SAT-solving is employed for many application, while there is no algorithm that are known to perform better than in exponential time in the worst case.

- If the equation is $T(n+1) = T(n)+f(n)$, then a solution is $\mathcal{O}(nf(n))$. In particular, if f is a polynomial, this is a $\Theta(nf(n))$.
- If the equation is $T(n+1) = aT(n) + b$ for some $a > 1$ and $b > 0$, the complexity is $\Theta(a^n)$ (typically this corresponds to having a recursive calls).
- If the equation is $T(n) = T(\frac{n}{k})+b$ for $k > 1$ and $b > 0$, the complexity is $\Theta(\log(n))$.
- If the equation has shape
$$T(n) = aT(\frac{n}{b}) + f(n)$$
one might be able to determine some bounds using the **master theorem**. You are not expected to recall the theorem statement for the exam, it will be recalled if you need it. But you should be able to apply it.

> **Theorem 1** (master theorem). *Assume that* $T(n) = aT\left(\frac{n}{b}\right) + f(n)$
>
> 1. *If* $f(n) = \mathcal{O}(n^{\log_b(a)-\varepsilon})$ *for some* $\varepsilon > 0$,
> ▷ *then* $T(n) = \Theta(n^{\log_b(a)})$
> 2. *If* $f(n) = \Theta\left(n^{\log_b(a)}\log(n)^k\right)$ *for some* $k \geq 0$,
> ▷ *then* $T(n) = \Theta\left(n^{\log_b(a)}\log(n)^{k+1}\right)$
> 3. *If* $f(n) = \Omega\left(n^{\log_b(a)+\varepsilon}\right)$ *for some* $\varepsilon > 0$,
> *and there is* $c < 1$ *such that* $af\left(\frac{n}{b}\right) \leq cf(n)$,
> ▷ *then* $T(n) = \Theta(f(n))$

- If you are computing a $\mathcal{O}$, you can over-approximate a recursive equation. For instance, if you have the recursive equation
$$T(n) = T(a(n-1)) + T(bn) + f(n-2)$$
with $a > b$ and f non-increasing, T will be a $\mathcal{O}$ of any increasing function satisfying $T'(n) = T'(an) + f(n)$, so you can try to solve the latter.
- Another technique might be to do change of parameters; for instance, an equation like $u_{n+2} = u_n + 1$ can be solved by considering $v_k = u_{2k}$ and $v'_k = u_{2k+1}$.
- While formally one should take rounding into account when dealing with divide-and-conquer approaches if we want to be rigorous since we are dealing with functions $\mathbb{N} \to \mathbb{N}$, we will consider it fine to be a bit sloppy here.

2 Techniques for coming up with algorithms

2.1 Recursion

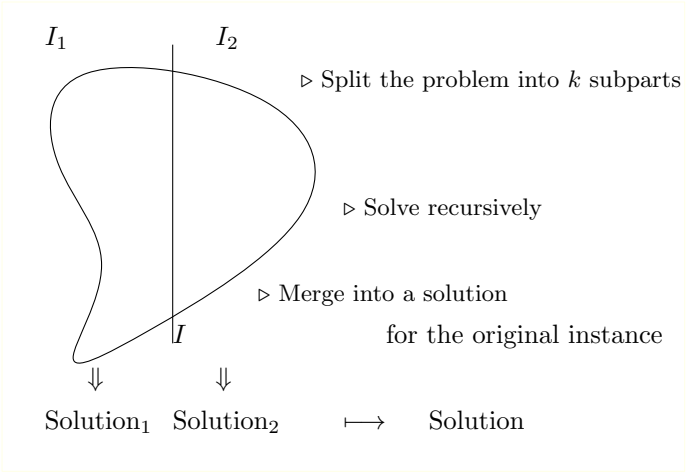
- A function implementation is called **recursive** when the the function **calls itself** during execution. It could be in the body of the function, or, in the case of mutually recursive functions, in another function.
- To terminate, recursive functions will typicaly call themselves on **strictly smaller arguments**. There are exception to this pattern, e.g. when the function involves some input-output interactions with the user/environment.

- When programming, recursion is essentially as powerful as iteration (loops). Choosing one over the other in a first implementation is mostly a matter of convenience. Recursion can be used to more easily implement some contol flows, and compiling recursive functions into iterative ones is typically a bit more challenging than the other way around (essentially because one needs to maintain a stack structure to emulate the function calls).

- Recursion can lead to some natural solutions using the following recipe: **if I can solve small instances of my problem, and for any big instance, I can solve it assuming that I can solve all strictly smaller instances, then I can solve all instances**.

2.2 Divide-and-conquer

- High level concept: divide you problems in several subparts



- Typically: greater size reduction across recursive calls, but also **multiple recursive calls**.

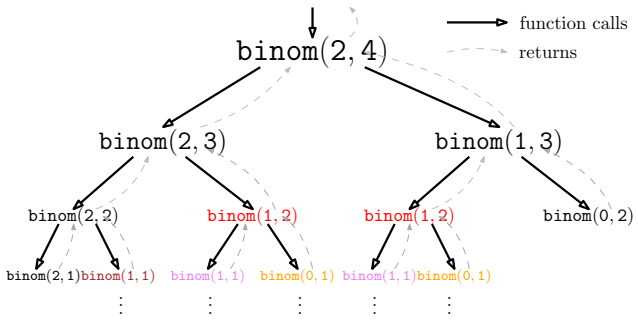
- Complexity analysis: typically using the master theorem.

- Typical examples: merge sort, fast exponentiation, dichotomy search.

2.3 Dynamic programming/memoization

- Sometimes, there is a natural recursive solution that would have **redundant recursive calls**.

```
int binom(int k, int n)
{
  if (k > n)
    return 0;
  if (k == 0 )
    return 1;
  else
    return binom(k-1,n-1) + binom(k,n-1);
}
```



Dynamic programming or **memoization**[7] is essentially the technique that consists in avoiding these kind of redundant calls.

- One easy way of doing it is modifying the recursive function by looking up in a table before doing a recursive call. On the above example it may be done as follows:

```
final int N = 100;
final int K = 20;

final int[][] cache = new Array[K][N];
//assume that main() initializes cache with -1

static int binom(int k, int n)
{
  if (cache[k][n] != -1)
    return cache[k][n];
  if (k > n)
    return cache[k][n] = 0;
  if (k == 0)
    return cache[k][n] = 1;
  else
    return cache[k][n] = binom(k-1,n-1) + binom(k,n-1);
}
```

One may use `ArrayList` and `static variables` to get cleaner implementations. If the input is a complex datatype, using a **hash table** for memoization is a good solution.

The complexity analyses are typically less straightforward to carry to compute *accurate* bounds; usually the idea is to sum the contribution of each subinputs.

- Sometimes one can also use that as inspiration to get iterative implementation like the following:

```
final int N = 100;
final int K = 20;

int binom[K][N];
binom[0][0] = 1;
for(int n = 1; n < N; n++)
```

[7] I prefer to say memoization because I find it more evocative (i.e., you memorize a bunch outputs), but dynamic programming is unfortunately more widespread; the guy who coined the term picked it to cajole some bureaucrat of the US army.

```
{
  binom[0][n] = 1;
  for(int k = 1; k <= min(n,K); k++)
    binom[k][n] = binom[k][n-1] + binom[k-1][n-1];
}
```

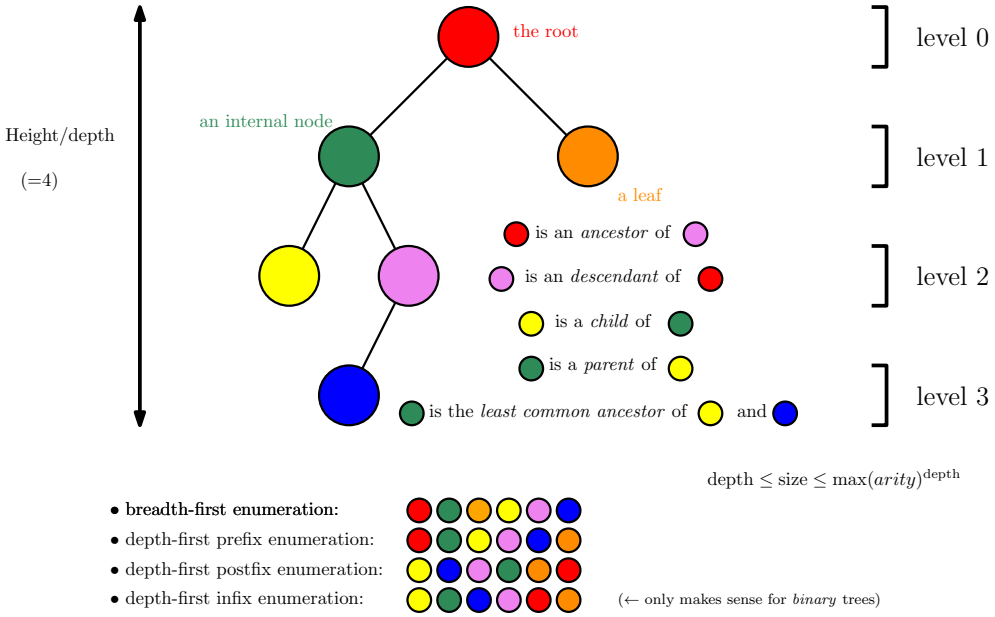
3 Trees

3.1 Basic notions/vocabulary

- One possible definition

> **Definition 3** (Finite trees, inductive definition←). *A tree with labels in L is a pair* $(\text{label}, \langle c_1, \ldots, c_n \rangle)$ *where:*
>
> – $\text{label} \in L$
>
> – $\langle c_1, \ldots, c_n \rangle$ *is a list of trees with labels in L* *(possibly an empty list)*

- Summary of the vocabulary:



- Variations: bounding the number of nodes, not caring about the order of children, different kind of labellings...

- On a mathematical level, trees can be characterized as those undirected graphs with a designated root vertex and with **exactly one path between any two vertices**.

3.2 Representation in OO languages

- Typically using a recursive class like the following:

```
class Tree<T>
{
```

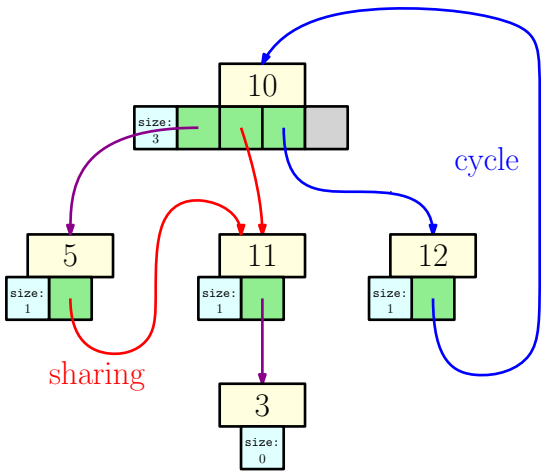
```
  public T label;
  public ArrayList<Tree<T>> children;

  public static <T> Tree<T> leaf(T x)
  {
      Tree<T> t = new Tree<T>();
      t.children = new ArrayList<Tree<T>>();
      t.label = x;
      return t;
  }
}

class BinaryTree<T>
{
  public T label;
  public Tree<T> left;
  public Tree<T> right;
}
```

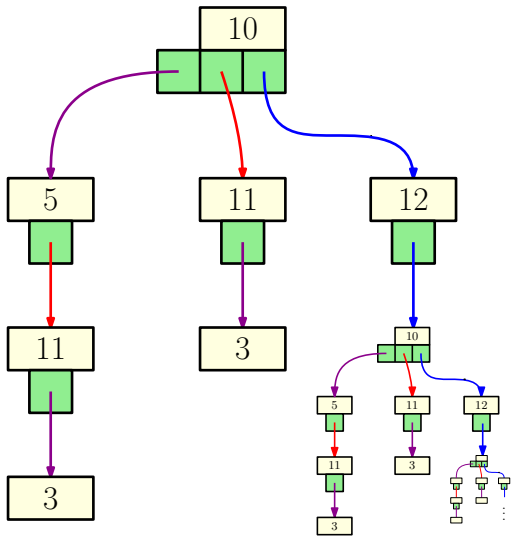
In practice, most **recursive classes** are going to have tree-like representations.

- Those classes allow to encode structures that are not tree-like: a reference may be reachable in distinct ways (**sharing**) or create a **cycle**.



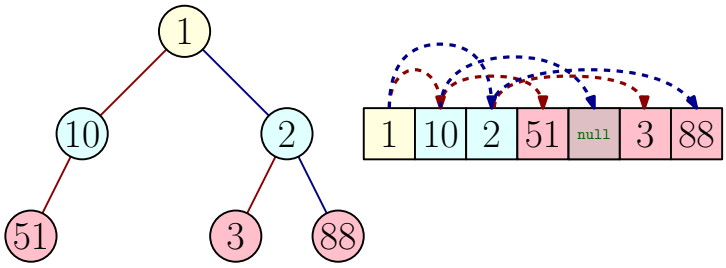
These often arise as mistakes, but sometimes this might be intended:

  - in case of cycles, this can be used to represent graphs or structures that are meant to be infinite, but this is a rather unusual use



  - for sharing, this might be more legitimate and correspond to **directed acyclic graphs** that behave like trees as long as one *does not change the values of the attributes of the object*[8].

- For trees with bounded degree (e.g. binary tree), one may use an array



  It gives fast access due to $\mathcal{O}(1)$ lookup in arrays, but *potentially* **wastes memory**. Particularly advantageous for trees with **full** levels such as **heaps**.

3.3 Tree-like datastructures

- A **heap** is a special kind of labelled tree

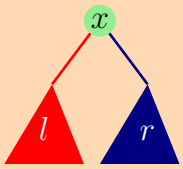
> **Definition 4** (Min-heap)**.** *A **min-heap** is a binary tree such that*
>
> - *The label of every node is smaller than its children's*
> - *All of its levels are full, except possibly the last*
> - *The last level is completely filled left-to-right up until a point*

  They support **insertion** and **deletion of the root** in logarithmic time (proportional to the depth of the heap since all levels)

- A **binary search tree** also supports insertion and deletion in time proportional to its depth, but also **lookup**.

[8] In fact, there are even techniques to *maximize* sharing in immutable tree-like datastructures called **hash-consing**.

Definition 5 (Binary search tree). *A **Binary Search Tree** (BST) is a binary tree labeled by integers such that*



$$\Rightarrow \qquad l \leq x \leq r$$

There are fancier insertion algorithms and structures such as AVL or red-black trees that allow to maintain that the trees have at most *logarithmic depth*; this means we can have datastructures for sets with basic operations in logarithmic time (implemented in the `TreeSet` class in java).

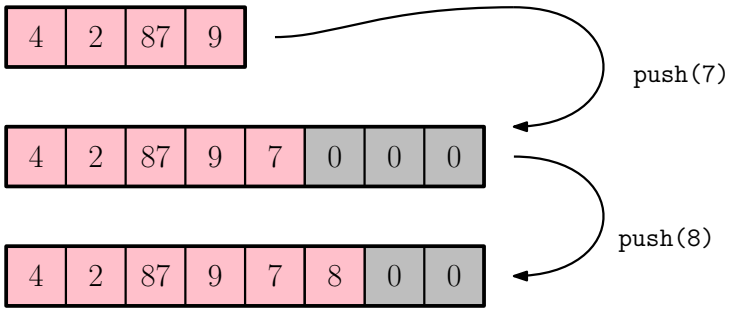
3.4 Dynamic arrays

- An overlay on top of an array with a smart memory management policy.

```
public class DynArrayInt {
private int[] internalArray;
private int size;
  ... }
```

Invariant: the size of internalArray is $= 2^{\lceil \log_2(\texttt{size}) \rceil}$

- A single addition of an element is $\mathcal{O}(n)$ in the worst case, but we have an **amotized complexity of** $\mathcal{O}(1)$.
- Otherwise the operations have the same complexity as for a primitive array.

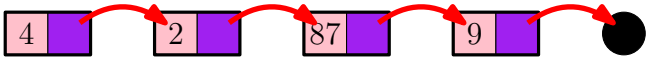


3.5 Linked lists

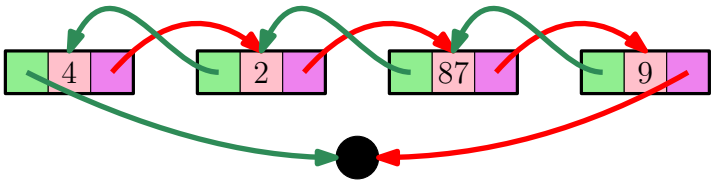
- Idea: a non-contiguous representation in memory allows easy deletion/insertion/extension.

Definition 6. ***Recursive definition*** *A linked list is either*

- *a flag denoting an empty list*
- *or a cell containing a value and a reference to a linked list*



- Useful vocabulary for non-empty values
  - **head** = value of the first cell
  - **tail** = the remainder of the list
- Via a recursively defined class as for trees.
- Possibility of adding fields to:
  - reverse in $\mathcal{O}(1)$
  - have the length in $\mathcal{O}(1)$
  - have concatenation in $\mathcal{O}(1)$
  - be able to traverse from both ends (**doubly-linked lists**)



- Some comparisons:

Op \Data	`Array`	`List`	`ArrayList`
`remove`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`contains`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`add`	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$ $\mathcal{O}(1)$ amortized
`concat`	$\mathcal{O}(n+m)$	$\mathcal{O}(1)$	$\mathcal{O}(n+m)$ $\mathcal{O}(m)$ amortized

3.6 Queues, stacks and priority queues

- A **stack** is a datastructure that stores elements and supports two operations:
  - add an element on top (**push**)
  - extract the latest inserted element (**pop**)

  It allows to process data in a **last-in-first-out** (LIFO) fashion.
- A **queue** is in a sense has a dual interface with a similar signature:
  - add an element at the end (**enqueue**)
  - extract the first inserted element (**dequeue**)

  It allows to process data in a **first-in-first-out** (FIFO) fashion.
- A **priority queue** has an additional integer argument for **enqueue** that gives a priority to the inserted label. Then the element dequeued is the first inserted element *among those with least priority.*
- Queues and stacks can be implemented with operations in $\mathcal{O}(1)$ using doubly-linked lists (amortized $\mathcal{O}(1)$ with dynamic arrays). For priority queues, heaps allow to have logarithmic time operations while lists/arrays incurr a linear penalty.
- These structures can be useful in real-time scenarii to process data (buffering), but also as useful auxiliary datastructures such as parsing or sorting.

4 Sorting algorithms

4.1 Concepts

- Sorting makes sense for several datastructures such as arrays and lists.
- One can sort arrays of integers, but also complex datatypes if a custom comparison function is provided (i.e. something extending a `Comparable` interface in Java)
- Sorting by comparisons is necessarily $\Omega(n \log(n))$.
- If one only needs to sort according to a retricted range of integers, there are linear-time sorting algorithms.
- A sorting algorithm is **in-place** if it does not require allocating any new arrays/collections of non-constant size; those algorithms typically operate with space complexity $\mathcal{O}(\log(n))$.

4.2 Classical algorithms

- **Insertion sort** is a quadratic sorting algorithm that works using an auxiliary function that inserts an element in an already ordered collection. This is fairly efficient in practice over small linked lists. Over arrays, **bubble sort** is another simpe quadratic-time sorting algorithm.
- **Merge sort** is a divide-and-conquer sorting algorithm which relies on a linear-time merging procedure. Its running-time is $\Theta(n \log(n))$. If the input and output is a linked list, this can be done in-place. This is a stable sorting algorithm.
- **Quicksort** is another divide-and-conquer sorting algorithm which first picks a distinguished pivot and recursively sorts the elements smaller and larger than the pivot. It is easy to implement it in-place if the pivot selection strategy is simple. The worst case running time is quadratic (attained for an already-sorted input), but on average this runs in $\Theta(n \log(n))$.
- **Heapsort** is a sorting algorithm that amounts to inserting all the elements of the input in a min-heap and then extracting the root repeatedly to construct the sorted output. This works in $\Theta(n \log(n))$ and is easy to implement in-place in an array.

5 Graphs

5.1 Basic notions/vocabulary

- Formally, a **graph** is simply a relation with a finite domain

  Definition 7. *A **directed graph** G is a pair (V, E) where V is a finite set and $E \subseteq V \times V$.*

  The elements of V are called **vertices** and elements of E called **edges**.
- A **path** is a sequence of vertices $v_0, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for every $i < k$. A **cycle** is a path of size > 1 with $v_0 = v_k$. A graph is called **connected** if between every two nodes there is a path.

- The **degree** of a vertex u is the number of neighbours, that is the size of

$$\{v \in V \mid (u, v) \in E \text{ or } (v, u) \in E\}$$

  We can talk of **in-degree** and **out-degree** when we talk about nodes with incoming/outcoming edges.

- There are variations of the notion of graphs:
  - **Undirected graphs**
  - **Multigraphs** (possibly multiple vertices between two vertices)
  - **Weighted graphs** where edges are labelled by integers (i.e., with a weight function $w : E \to \mathbb{N}$).

- Note that if a graph has n vertices, it may have a maximum of n^2 edges. If it is an undirected graph without self-loop, this is a maximum of $\frac{n(n-1)}{2}$.

- A class of graphs $\mathcal{C}$ is said to be **sparse** if $\max_{(V,E)\in\mathcal{C}} |E|$ is not quadratic (i.e., a $o(n^2)$).

5.2 Representations: adjacency matrices vs adjacency lists

There are two main useful ways of representing graphs, where vertices are indexed by consecutive integers (so essentially we take $V = \{0, \ldots, n-1\}$): adjacency matrices and adjacency lists.

- An **adjacency matrix** records if an edge is there or not (or if it is). This is always quadratic in the number of vertices. This is good for random access to check if two nodes are connected by an edge, but then enumerating the neighbours of a vertex is linear in the number of vertices.

- A set of **adjacency lists** consists of an array of lists such that the cell indexed by u contains the list of the vs such that $(u, v) \in E$. Typically we may keep without great loss of efficiency also another adjacency lists for predecessors, i.e. an array such that cell u contains the list of vs such that $(v, u) \in E$. Checking if an edge exists between two arbitrary vertices is no longer constant time, but linear in the maximal degree of the two nodes. On the other hand enumerating the neighbors of v can be done in time proportional to the degree of v.

- For dense graphs, adjacency matrices may be better, but generally, adjacency lists are better, especially over sparse graphs.

5.3 Algorithms: traversal, distances

- Traversal algorithms consist in traversing the graph from a starting point and performing some operations along the way.

- A **depth-first** traversal prioritizes enumerating nearby nodes first. It can be implemented using either a straightforward recursion or an explicit stack structure. If the graph is represented by an adjacency list, this is doable in $\mathcal{O}(|E|)$.

- A **breadth-first** traversal prioritizes enumerating nearby nodes first. It can be implemented using a queue (this is the only difference with a deph-first algo, replacing the stack with a queue). If the graph is represented by an adjacency list, this is doable in $\mathcal{O}(|E|)$.

- In a weighted graph, the **distance** between two nodes is the minimal weight of a path (obtained by summing the labels) between those (if there is no path, then it is ∞). Note that the notion also makes sense for non-weighted graphs, where we can say all edges have weight 1 (and weight of a path corresponds to length).
- A breadth-first traversal can allow to compute the distance from a source nodes to all the other in an unweighted graph.
- In a weighted graph, one may use a **priority queue** to compute that efficiently using **Dijkstra's** algorithm (beware, one should include the complexity of the priority queue operation)

```
Djikstra(G, source)
    Q ← an empty priority queue
    Enqueue source with priority 0 in Q
    while Q is not empty do
        Dequeue the element v with minimal priority d from Q
        if v was not visited before then
            Set the distance between source and v to be d
            for all edges v --w--> v' do
                Enqueue v' with priority d + w in Q
            end
        end
    end

    return the computed distances
```

- If one wants to compute all distances between every pair of nodes, one may use the **Floyd-Warshall algorithm** (here presented over matrices) in $\mathcal{O}(|V|^3)$.

```
FloydWarshall(M)
    D ← a copy of M
    n ← dimension of M
    for k from 0 to n − 1 do
        for i from 0 to n − 1 do
            for j from 0 to n − 1 do
                D[i][j] ← min(D[i][j], D[i][k] + D[k][j])
            end
        end
    end
    return D
```

6 Hashing

- A **hash function** converts a (potentially complex) value into an integer in a limited range. There are two desiderata:
  1. They should be computable efficiently (ideally $\mathcal{O}(1)$!)

2. They should spread their inputs **uniformly**, i.e., if we take an input of size n at random, each value in the output should have roughly the the same probability of occuring.

The output of a hash function is called the **hash** of the input. A pair of inputs (x, y) for a hashing function H such that $H(x) = H(y)$ is called a **collision** (for H) - point 2. above implies that a hash function is good when collisions rarely occur - this might depend, of course, on the specific application/probability distribution on the inputs of H.

- For cryptography and verification, we may additionally want that it is intrisically hard to compute from an output a possible input; essentially it must be hard for hackers to compute collisions.
  - One common use of hash functions is for password management: for instance, when designing a log-in for a website, to mitigate the impact of possible data leak if the server is breached, what should be stored on there should not be the user's password but **a hash of the user's password together with some string specific to the service** (typically called "salt"). Thus if a malicious actor get access to the server, the actual password that the user uses (that may use across multiple services) is not leaked. (there are schemes with stronger security guarantees, a lot of which will also use hash functions as building blocks)
  - Another use of hash functions is in checking the integrity of downloads. In many scenarii, the downloading large files can be corrupted due to e.g. a bit flip in the transfer. Detecting such errors can be done by computing the hash of the downloaded file (which is typically *much* smaller than the overall file) and check that it matches the hash given by the server. This is used implicitly for instance in most torrent clients.
- As far as datastructures are concerned, hash functions are useful to implement **hash tables**, a datastructure that can provide arrays indexed by complex data/non-consecutive integers. The goal is to have constant-time operations of regular arrays in constant-time **on average**.
- (out of scope for the exam) Hash tables are very useful to do **dynamic programming** if the input is a complex value and **hash-consing** when implementing arborescent immutable datastructures with **maximal sharing** – the latter can be interesting for enumeration problems and to speed up comparisons when the result is expected to be `false`.