Data Structures

An introduction

Francesco Andreussi

Bauhaus-Universität Weimar

26 April 2019

Bauhaus-Universität Weimar
Fakultät Medien

Basic Information

- Class every second week, 15:15-16:45 in this room, every change will be communicate to you as soon as possible via mail
- Every time will be presented an assignment regarding the arguments covered both by Prof. Wüthrich and me to be fulfilled before the following class. **You MUST work alone!**
- The submissions are to be done sending me an email before the deadline with your personal data and either a zip archive or a link to a Github repository, as long as I can download your work, and name it using the format **SurnameName_MatricNum_AssignmentNum**.

Delays in the submissions and cheats (e.g. code copied from the Internet or other groups) WILL NOT be accepted!

If you find code online **DO NOT COPY-PASTE IT**, otherwise I have to consider it cheating and you will be rewarded with a 5.0.

For any question you can write me at *francesco.andreussi@uni-weimar.de*.
The material will be constantly uploaded to *this page*.

Algorithms

If learning a programming language can be seen as learning a (rather simple) foreign language, learning how to write and use algorithms can be seen as learning how to be a poet.

Jokes apart and more formally, an **algorithm** is a **finite sequence of instructions** that

- can be followed **unambiguously**,
- takes **finite space** and **time** to be executed,
- requires possibly some **input**,
- returns the **required output**,
- and has sometimes **side effects** on the state of the executor.

Data Structures (Abstract Data Types)

Data Structures are essentially what the name suggests: ways of organising (or **structuring**) data which provide also a **set of operations** in order to easily and efficiently **access** them, and **add** and **delete** items from the current set.

Data Structures and Algorithms are treated together because they are tightly connected: access and manipulation methods of each data structure are algorithms and almost every algorithm exploits a data structure to store wisely the data it has to use.

Now, we will have an overview of some of the most common data structures.

Arrays (and Matrices)

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Fig. 1: A 4×3 bidimensional array

- **Static low-level** data structures
- Allowing only **coherent** data types
- Extremely **efficient** but **less convenient**

Linked Lists

The Structure

Very similar to the array concept, yet far more flexible.

- **Dynamic** data structures both in dimension and data types to be stored
- Many different possibilities for implementing a linked list (same basic common structure but several other things can be done)
- It is at least a series of **nodes** where every node stores some data and a pointer to the next node while the list itself holds a pointer to the head of the list and perhaps some other values

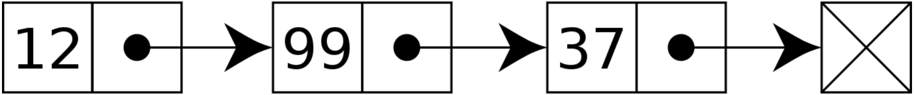

Fig. 2: A basic Single-linked list

Linked Lists

Basic Operation

The two basic operation offered by a linked list are the insertion and the deletion of a node which concepts are the following.

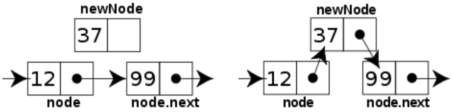

Fig. 3: Adding a node

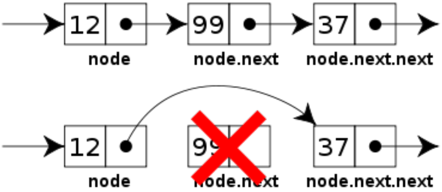

Fig. 4: Deleting a node

Linked Lists

Stacks

- Single-linked lists that allow operation **only** on the **first element** of the lists
- Inserting, removing and accessing operations are commonly known as **push**, **pop** and **peek** respectively
- These operations follow the **LIFO** (last-in, first-out) accounting method

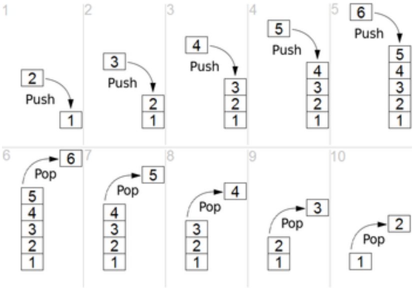

Fig. 5: Sequence of pushes and pops in a stack

Linked Lists

Queues

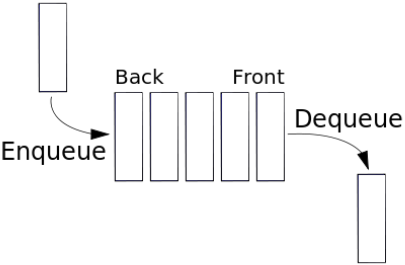

Fig. 6: Push and pop in a queue

- Single-linked lists as stacks, yet using **FIFO** (first-in, first-out) policy
- That means that the **last added** (enqueued) element becomes the **head** of the list while the **tail** is the element to be removed at the next pop (dequeue)

Trees

- Extension of the multi-linked list concept: every node has a **parent** and a series (possibly empty) of **children**
- This makes a tree a **non-linear recursive** ADT; trees **cannot contain a cycle** of connections between its nodes
- A node without parent is called **root**, one without children is called **leaves** and all the others are **internal nodes**
- Various kind of trees are possible, but in order to be a tree there should be a **unique** path root-to-leaf for every leaf

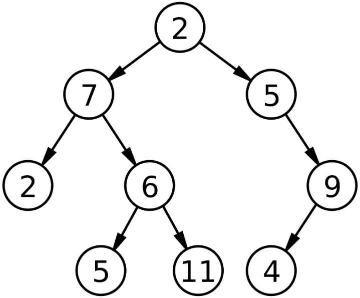

Fig. 7: A Binary Tree

Trees

Some More Terminology

- **Degree**: for a given node, its number of children. A leaf has, then, degree zero
- **Edge**: the connection between one node and another
- **Path**: a sequence of nodes and edges connecting a node with one of his descendants
- **Level**: for a given node, $1 +$ the number of edges between the node and the root
- **Depth**: for a given node, the number of edges between the node and the root
- **Height of node**: the height of a node is the number of edges on the longest path between that node and a descendant leaf
- **Height of tree**: the height of a tree is the height of its root node.
- **Forest**: a forest is a set of disjoint trees.

Trees

Traversing Order & Other Common Operations

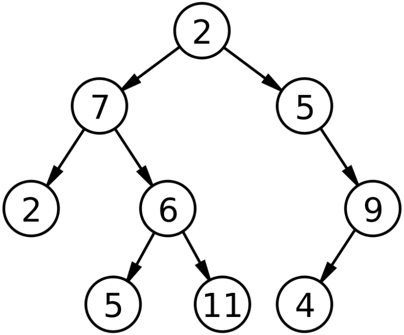

Fig. 8: A Binary Tree

- **Pre-Order**: 2-7-2-6-5-11-5-9-4
- **In-Order**: 2-7-5-6-11-2-5-4-9
- **Post-Order**: 2-5-11-6-7-4-9-5-2
- **Level-Order**: 2-7-5-2-6-5-11-9-4

Common operations are: **enumeration of elements**, **searching**, **adding elements**, **removing elements**, **pruning** (removing subtrees) and **grafting** (adding subtrees).

Graphs (1)

- Extension of the tree concept adding the possibility of **cycles**
- Formally it is the ordered pair of two sets (V, E) where V is the set of **vertices** and E is the one of **edges**
- Practically a node stores all the necessary data and holds references to all the connected nodes
- In this case $G = (\{A, B, C, D\}, \{(A, B), (B, A), (B, D), (B, C), (C, B), (C, D), (D, B), (D, C)\})$

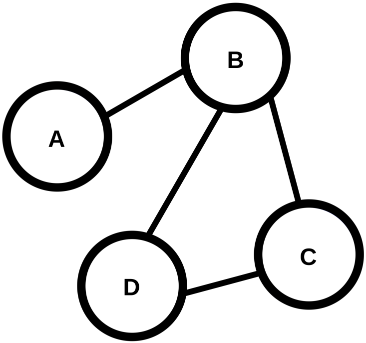

Fig. 9: A Generic Undirected Graph

Graphs (2)

Another way of representing a graph is using an **adjacency matrix**:

	A	B	C	D
A	0	1	0	0
B	1	0	1	1
C	0	1	0	1
D	0	1	1	0

Table 1: The Adjacency Matrix for the former graph

Graphs

Other Types of Graphs

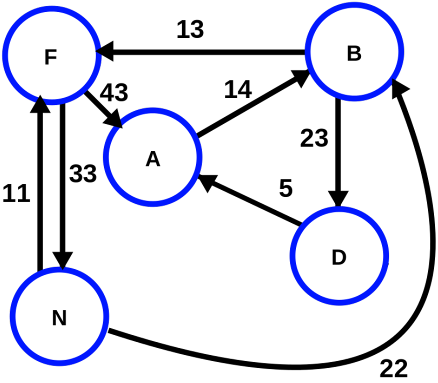

Fig. 10: A Weighted Directed Graph

- **Directed**: edges here are not bidirectional (the adjacency matrix is not symmetric)
- **Weighted**: edges are associated with a positive number (weight) representing (usually) the cost of crossing the specific edge

In graphs it should be at least possible to **access**, **add** and **remove vertices**, **get** and **set** the vertices values, **access the adjacent nodes** of a vertex and **test for adjacency**.

P.S.: take a look also to *Introduction to Algorithms, 3rd Edition*

Thanks for the Attention!