1 - Data Structures

Charles A. Wuethrich

Bauhaus-University Weimar
CoGVis/MMC

April 11, 2019

Definition of Algorithms

- Named after persian Mathematician
  Abu Abdullah Muhammad ibn Musa al-Khwarizmi.
- Algorithms have a long story
  - At first: Mathematical procedure using arabic numbers
  - Later: Definite procedure for solving problems
- Nowadays: very few know what they are.
  - For most: A method for computing something
- What is the correct definition?
- We first look at a real life example.

A real life example

- Perhaps most common example?
- Cooking recipes!
- To the delight of good food appreciators: I will present a typical recipe from the place I come from
- Scaluppinn al Marsala
  (Veal scallops with Marsala wine)

A real life example

- Perhaps most common example?
- Cooking recipes!
- To the delight of good food appreciators: I will present a typical recipe from the place I come from
- Scaluppinn al Marsala
  (Veal scallops with Marsala wine)

A real life example

- Perhaps most common example?
- Cooking recipes!
- To the delight of good food appreciators: I will present a typical recipe from the place I come from
- Scaluppinn al Marsala
  (Veal scallops with Marsala wine)

Scaluppinn al Marsala

Ingredients (4 servings):

4 Slices of veal scallops (100 g. each),

100 g. Butter,

250 ml. Marsala wine,

Flour,
Pepper, salt.

Preparation:
Flatten the scallops with a mallet.
Cover with flour a flat dish, and dip both sides of each scallop in the flour, so that each side is covered with flour.
In a large pan heat at high flame the butter and 100 ml. of Marsala wine until the mixture is at frying temperature.
Put the veal in the pan, and let it brown on both sides.
Add the remaining Marsala wine, lower the flame and let the Marsala wine evaporate until formed a thin coating on the meat.

Add salt and pepper, and serve.

Some thoughts on recipes

- There are two parts in a recipe:
  - The ingredients
  - The description
- Description has to be exact.
- Reader needs to be able to read and execute.

- One can see ingredients as input
- And food as output (hopefully tasty)

Some thoughts on recipes

- . . . "Reader needs to be able to read and execute": What does this mean?
  - Description has to be built out of single steps the user is able to follow.
  - The single atomic steps are the instructions of the recipe
  - An atomic instruction is an instruction one assumes the reader can follow
- What does this translate into for a computer?
  - Atomic instructions are computer instructions.
  - Computers can only follow these.
  - Which leads us to the following definition

Definition of Algorithm

Algorithm: A finite sequence of well defined instructions which, given an input, will produce in a finite time a required output.

This definition hides more constraints than one would expect at first sight.

Definition of Algorithm

Algorithm: A finite sequence of well defined instructions which, given an input, will produce in a finite time a required output.

- We need well defined instructions
- We need an input.
- We have a required output: the algorithm has to be correct, i.e. produce the required output, otherwise it does not do what we want.
- And this in a finite time, because otherwise one would wait forever.
  - In other words, the algorithm has to terminate in the required state.

Scaluppinn al Marsala: Instructions

Preparation:
Flatten the scallops with a mallet.
Cover with flour a flat dish, and dip both sides of each scallop in the flour, so that each side is covered with flour.
In a large pan heat at high flame the butter and 100 ml.
of Marsala wine until the mixture is at frying temperature. Put the veal in the pan, and let it brown on both sides.
Add the remaining Marsala wine, lower the flame and let the Marsala wine evaporate until it formed a thin coating on the meat.

Add salt and pepper, and serve.

In instruction notation:

- Flatten the scallops with a mallet.
- Cover with flour a flat dish,
- Dip both sides of each scallop in the flour.
- Put butter and 100 ml. Marsala wine in pan.
- Heat at high flame until frying temperature is reached.
- Add veal scallops and brown both sides of the slices.
- Turn fire to low
- Add rest Marsala wine.
- Let wine evaporate until it formed a thin coating.
- Season with salt and pepper.
- Serve.

Parallels between recipes and computer algorithms

- In the last slide, each step of the recipe is exactly defined.
- In a way, it is the equivalent of the execution of a single line of code in a computer language.
- One could see the cook as a processor with precise capabilities:
  - Flatten
  - Dip
  - Increase or lower flame
  - Add ingredients
  - etc . . .

- Some recipes become very complicated
- Maybe require parallel processing
- Like computer algorithms, recipes can have all degrees of complexity, and take up to several days to complete.

Back to Mathematical Algorithms

- Instructions are unit of execution of the computer.
- Input is data.
- Output is transformed data, and must be the required one.
- Output has to be obtained in a finite time.
- The instructions must work in a deterministic way, that is, the same input must always produce the same output.
    - This is called a predictable algorithm.

- In a way, an algorithm is a machine composed by computer instructions that processes input in a finite time.
- On a computer, it will have instructions expressed in a particular programming language.
- These instructions will be translated by the interpreter or compiler to instructions in the a language directly executable by the computer processor.

Applications of algorithms

- From access ruling of the computer resources.
- To secure coding and transmission of information.
- To data sorting.
- To computing cost of a journey through multiple cities.
- To computing how much water the city water company can distribute to its customers through the pipes. . .

Organization of the teaching material

- Although the applications are so diverse, the basic mechanisms used in algorithms are often the same.
- Algorithms would less differ in their mechanisms, more in the mapping of a specific problem to a known mechanism.

- This is exactly how we will organize this course.
- By mechanism category, rather than by problem solved.
- Single problem solving will be an application of the mechanisms known to a different problem.

- This view will also allow to compute more easily the efficiency of the different algorithms.

Data

- Data has to be organized so as to be efficiently processed.
- Structuring data means organizing it so as to be able to map it to a known mechanism.

- Doing this is only possible if it is well clear how data can be organized (= structured) in a clever way.
- We will also learn that structuring data does not only mean organizing the data
- It means also organizing the way how to access the data.
- In object oriented languages, this is called object <u>data</u> and <u>methods</u> to manipulate the object data.

Definition of Data Structures

Data Structure: Structured memory space and a set of operations, or actions, for accessing and manipulating such memory space..

Elementary Data Structures

- Every programming language offers to the programmer simple data types and simple constructs to organize simple data.
- Java allows the following data types:
  - `boolean`: a binary value, generally `true` or `false`,
  - `char`: a character, i.e. an 8 bit positive integer,
  - `byte`: an 8 bit integer,
  - `short`: a 16 bit integer,
  - `int`: a 32 bit integer,
  - `long`: a 64 bit integer,
  - `float`: a 32 bit floating point number,
  - `double`: a 64 bit floating point number.

Elementary Data Structures

- Basic data types allow direct access to the data once the variable is declared.
- A declaration makes the computer reserve the appropriate memory space.
- Each time the variable name is encountered, it gets replaced with the memory address containing the data.

- Suppose we declare:
  `int intvar;`
- The computer reserves a 32 bit long space.
- Memorizes that any reference to variable `intvar` has to access its memory address.
- Memorizes that integer operation rules have to be applied for that variable.

Complex Data Structures: Arrays

- Programming languages such as Java allow also more complex data types to the programmer.
- Example: arrays
- Arrays are sets of consecutive data of the same data type.

- Suppose we declare:
  `int[] intvec;`
- The computer notes that the variable `intvec` has to be treated with the rules used for arrays.
  - This means, an element of the array is accessed e.g. as `intvec[5]`
- When the instantiation `intvec = new int[100];` is made,
  - It reserves 100 consecutive 32 bit long cells to the variable.

Complex Data Structures: Arrays

- Java also allows bi- and three-dimensional arrays.
- The declaration is the following:
  `int[][] intmatr;`
- And a matrix 10x10 is initialized as:
  `intmatr = new int[10][10];`
- Recap for arrays:
  + Fast for accessing and addressing data
  - Not flexible: once declared size is defined and not modifiable.

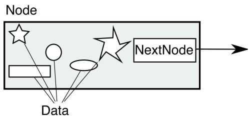
Data Structures and Classes

- Simple data-types and -structures allow limited flexibility.
- In OO programming, data is organized in classes.
- A <u>class</u> is a collection of
  - Data, either elementary or classes
  - Operations for manipulating such data, called <u>methods</u>.
- Each class has a particular method, called <u>constructor</u> which initializes the class (through `new`).
- Recall the definition of data structures:
  It looks the same!!!

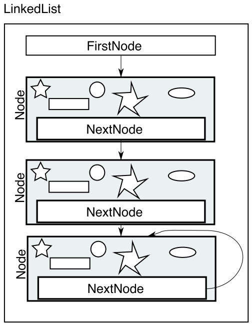
Linked Lists

- When dynamic memory allocation is available, one can implement "own" data structures allowing variable length.
- The first structure of this type is a linked list.
- A linked list is a set of individual nodes, each of which is constituted by:
  - Object data
  - A pointer, or reference to the next element of the list.
- The list itself contains only a reference to its first node
- The last node, by convention, points to itself (or to NULL).

Linked lists



(a) A node



(b) A list

Node implementations

- This is the corresponding code:

```
public class Node {
 NodeData data;
 Node nextnode;

 public Node(NodeData newdat) {
  data = newdat; nextnode = null;
 }
```

- Note how the node pointer is NULL as long as the node is not attached to a list.
- The constructor
  - Reserves memory for data and reference.
  - Initializes reference to NULL

Linked list implementation

- For the list class, first the node has to be filled with its data
- `thisdata` contains the node data
- and the node is made to point to itself (since it is the last)

```
Node newhead = new Node();
newhead.data = new NodeData(thisdata);
newhead.nextnode = newhead;
```

Linked list implementation

- Then comes the class `LinkedList` definition:

```
public class LinkedList {
 Node firstnode;

 public class NodeData {
  public NodeData(NodeData copy) {
  }
 }

 public class Node {
  public NodeData data;
  public Node nextnode;

  public Node(NodeData newdat) {
   data = newdat; nextnode = null;
  }

  public void initializehead
  (NodeData thisdata) {
   Node newhead;
   newhead = new Node(thisdata);
   newhead.nextnode = newhead;
  }
 }

 public LinkedList(Node newhead){
  // initialize LinkedList anew
  newhead.nextnode = newhead;
  firstnode=newhead;
 }
}
```

Linked Lists

- List contains only pointer to head
- If the first node does not exist, then the list points to NULL.

- Let us check if the list is empty:

```
public boolean CheckIfEmpty() {
 boolean empty = false;
 if(this.firstnode==null)
  empty = true;
 return empty;
}
```

Linked Lists

- Which methods belong to a list?
- We need to search nodes, add nodes, remove nodes.
- Add a node `tobeadded` at the head:

```
public void addNodeAtHead(Node tobeadded) {
 tobeadded.nextnode = this.firstnode;
 firstnode = tobeadded;
}
```

- We initialize the list by adding a new node and making the list point to this new head.

Linked Lists: add a Node

- Add a node tobeadded after the node previousnode:

```
public void addNodeAfter (Node previousnode,
 Node tobeadded ) {
 tobeadded.nextnode = previousnode.nextnode;
 previousnode.nextnode = tobeadded;
}
```

- Here we make tobeadded point to what previousnode was pointing to, and previousnode point to tobeadded

Linked Lists: remove a Node

- Remove a node `toberemoved`, and remember which its predecessor was:

```
public void removeNode
(Node previousnode , Node toberemoved ) {
 previousnode.nextnode =
 toberemoved.nextnode ;
 }
```

- Here we make `previousnode` point to what `toberemoved` was pointing to.

Linked Lists: search a Node

- We need to find a node in the list.
- We have to pass also who is its predecessor.

```
SearchNode(Node tobeseeked){
  NodeData data;
  Node=jumptonext;
  Node=previousnode;

  jumptonext=firstnode;
  previousnode=null;
  while((previousnode!=jumptonext)||
        (jumptonext.NodeData!=
           tobeseeked.NodeData)){
    previousnode = jumptonext;
    jumptonext=jumptonext.next;
    }
  if(previousnode==jumptonext){
    previousnode=null;
    jumptonext=null;
  }
  return(previousnode,jumptonext);
}
```

- The method returns NULL if the data was not found, `jumptonext` for the node found, and `previousnode` as its predecessor.

Linked Lists: "Fazit"

- Great structures, flexible and expandable!
  but. . .
- Search means scanning all elements
- And for backwards search?
- I have to start back from the head :-(
- And how do I do this?

Linked Lists: "Fazit"

- Great structures, flexible and expandable!
  but. . .
- Search means scanning all elements
- And for backwards search?
- I have to start back from the head :-(
- And how do I do this?

Linked Lists: "Fazit"

- Great structures, flexible and expandable!
  but. . .
- Search means scanning all elements
- And for backwards search?
- I have to start back from the head :-(
- And how do I do this?

Linked Lists: "Fazit"

- Great structures, flexible and expandable!
  but. . .
- Search means scanning all elements
- And for backwards search?
- I have to start back from the head :-(
- And how do I do this?

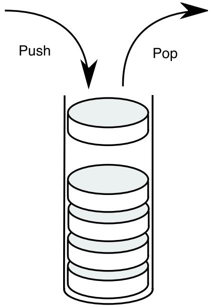
Stacks

- Sometimes, data has to be put into repository
- Like on my desk, it gets stacked
- Like in a tennis ball tube

- So one can recover it when necessary
- A tennis ball tube has a bottom
- One can insert the balls only from one side

Stacks

- Stacks are exactly like this

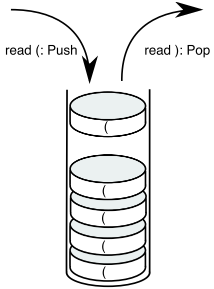


- The data is piled
- Only the top is accessible
- Two basic operations:
  - Pop
  - Push

Stacks

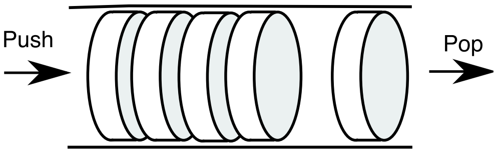
- Seems a little of an abstract data structure
- Let us think at a braces checker for an editor

- By pushing open braces in the stack when we find one
- . . . and popping them when we find a closed brace
- . . . we can count if the braces are correct!



Queues

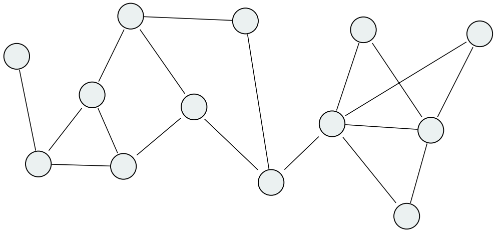
- Very similar to stack
- Only difference:
  - input on one side
  - output othe side



- Applications anyone?

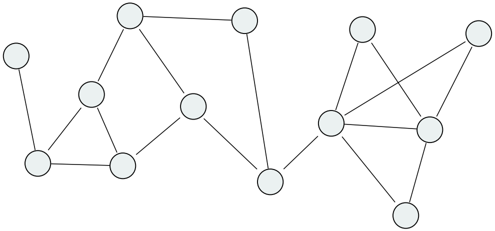
Graphs

- Who of you can help me with the definition of a Graph?



Graphs

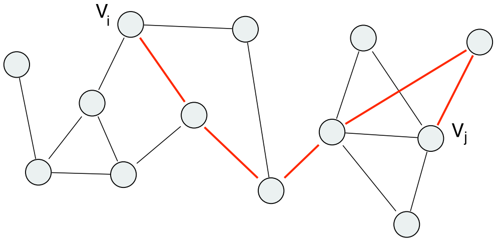
Graph: A graph $G =< V, E >$ is constituted by a finite set V of elements, called *nodes* or *vertices*, and by a set E of pairs of elements of G, called *arcs*.



- Arcs: connections between nodes
- In general: unordered pairs
- Adjacent nodes
- Adjacency relation

Graphs

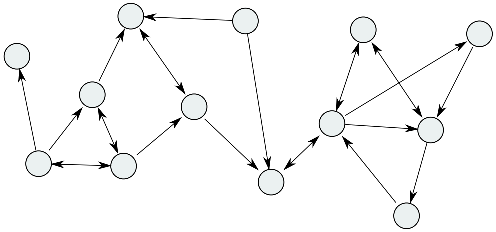
- Connected nodes V_i and V_j
- Path between V_i and V_j: a sequence of arcs ST they start at V_i, and end at V_j and are all incident



- Length of a path
- Cycle

Digraphs

- If adjacency relation not symmetric, the graph is called a Directed graph or Digraph



Implementing graphs

- There are basically two ways of implementing graphs:
  - As lists
  - Through their adjacency matrix

- List implementation quite simple: each list element is a node and contains a list of the adjacent nodes.

```
public class ListGraph {
 public class ListGraphNode {
  private List<ListGraphNode> _neighborList;
  public ListGraphNode() {
  }
 }

 private List<ListGraphNode> _nodelist;
 public ListGraph() {
 }
}
```

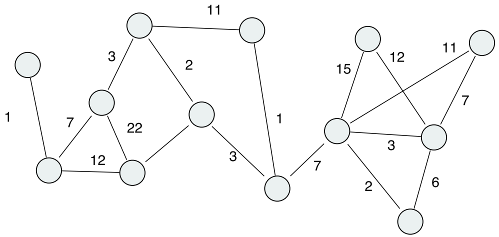
Implementing Graphs

- Second implementation: Adjacency Matrix
- Columns and rows: Nodes
- Position (i, j):
  - 1 if the arc $V_i V_j$ exists
  - 0 otherwise

- Symmetric for generic graphs
- Non-symmetric for digraphs

Weighted Graphs

- To each arc V_{ij}, a number w_{ij} called <u>weight</u> is associated.
- Here it is preferred adjacency matrix
- Position (i, j): weight of the arc



Flow Networks

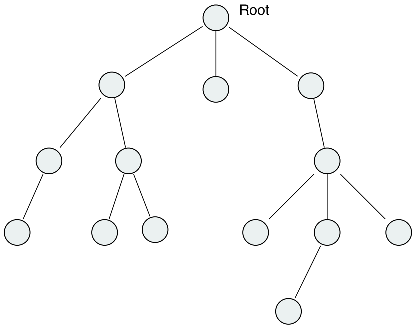
Flow Network:
A directed weighted graph having positive weights c_{ij}, called capacities.

Each arc is assigned a non-negative value f_{ij} (with $0 \leq f_{ij} \leq c_{ij}$) called flows.

- Two privileged nodes:
  - *source* has no incoming arcs
  - *drain* has no outgoing arcs
- *Flow*: A set F of all f_{ij}s such that for all nodes V_i, the sum of the incoming flows does not exceed the outgoing flow.
- Good for modeling pipelines....
- Flux: Maximum possible flow into the drain.

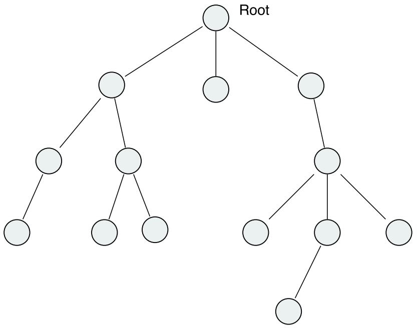
Trees

Tree: A graph which has no cycles is called a tree.



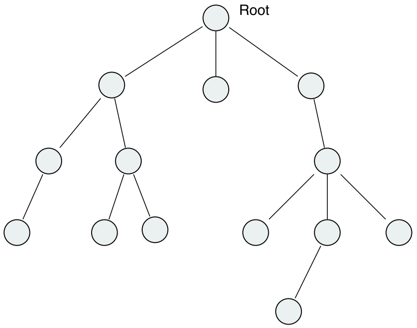
- One special node is chose, the root.
- Example: genealogy trees
- Father (parent), children, siblings.
- Depth for a node and for a whole tree.
- Level

Implementing Trees



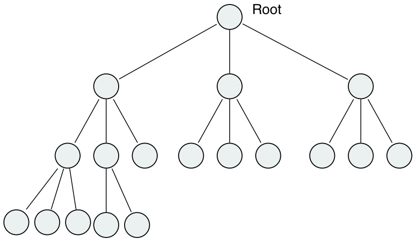
- A bit like linked lists:
- Data is stored at each node
- Reference to leftmost child.
- Reference to next sibling

Traversing Trees



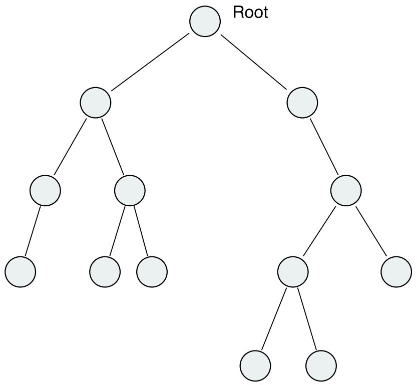
- Start at root node
- pre-order traversal: Visit first the content of the node, then its children (from left to right).
- Post-order traversal: Visit first node children (from left to right), then content of node.
- Level-order traversal: Group nodes according to depth and visit each level from left to right

n-ary Trees



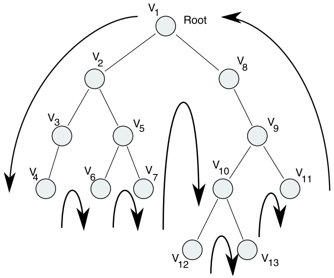
- A tree such that all of its nodes has at most n children is called an *n*-ary tree.
- Complete node
- Complete tree of depth d: all of its nodes up to the level $d-1$ are complete, and all nodes at level d are leaves,

Binary Trees



- Particularly important.
- Easy to implement: no need for siblings, just the two children

Traversing binary trees



- For binary trees, the traversal algorithms work slightly differently.
- Start at root node
    - pre-order traversal: Visit first content of node, then children subtrees (left and right).
    - Post-order traversal: Visit first node children subtrees (left and right), then node itself.
    - In-order traversal: First visit left child subtree, the node itself, then right hand child.