Data structures and algorithms

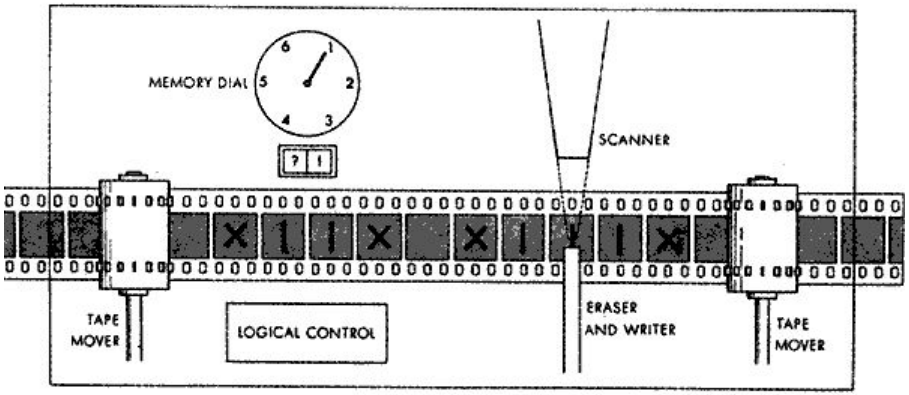
Master ATIAM - Informatique
Philippe Esling (esling@ircam.fr)
Maître de conférences – UPMC
Equipe représentations musicales (IRCAM, Paris)

Why bothering

- Data structures are **the heart of programming**
- The maxim of programming :

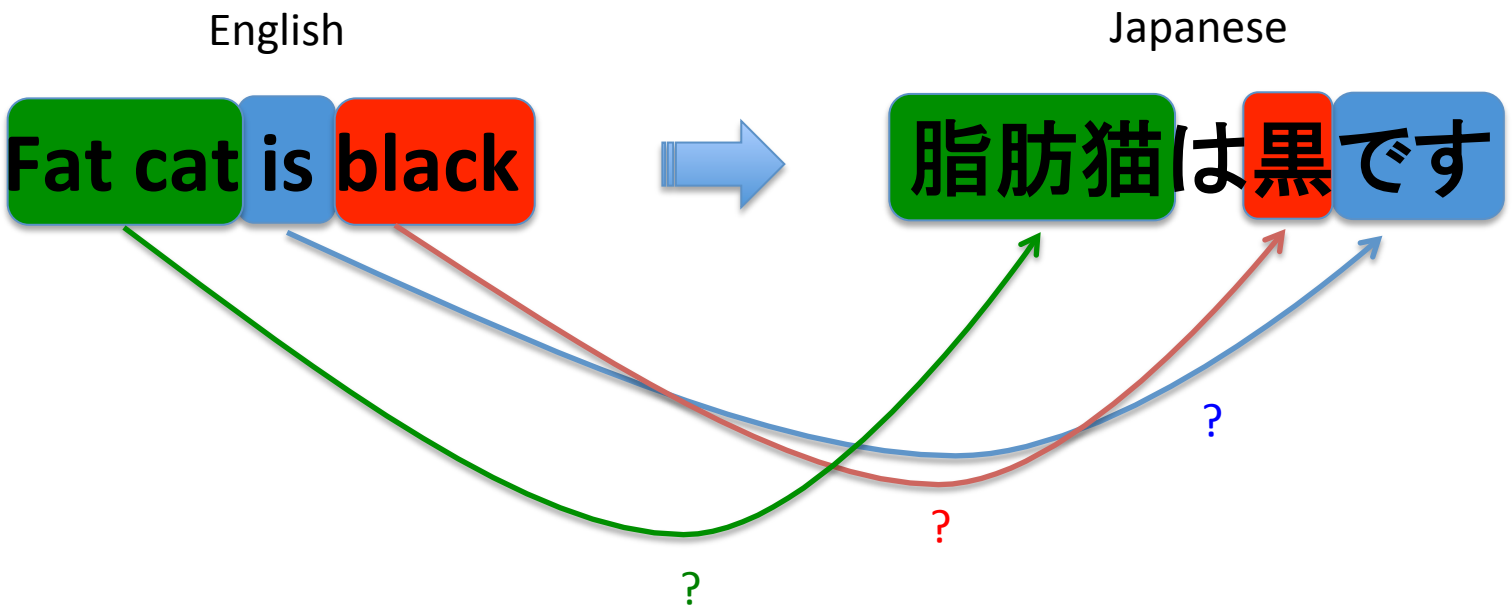
Data structures + algorithms = program

- Moving away from the Turing machine

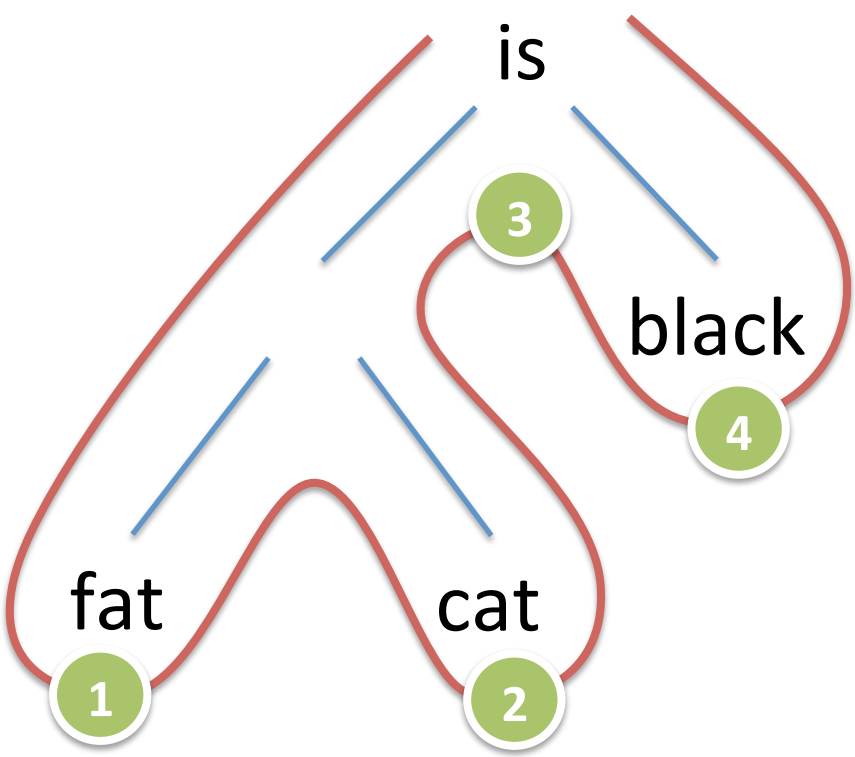


- Smart thinking of structures can **make you rich**
- Not convinced yet ?

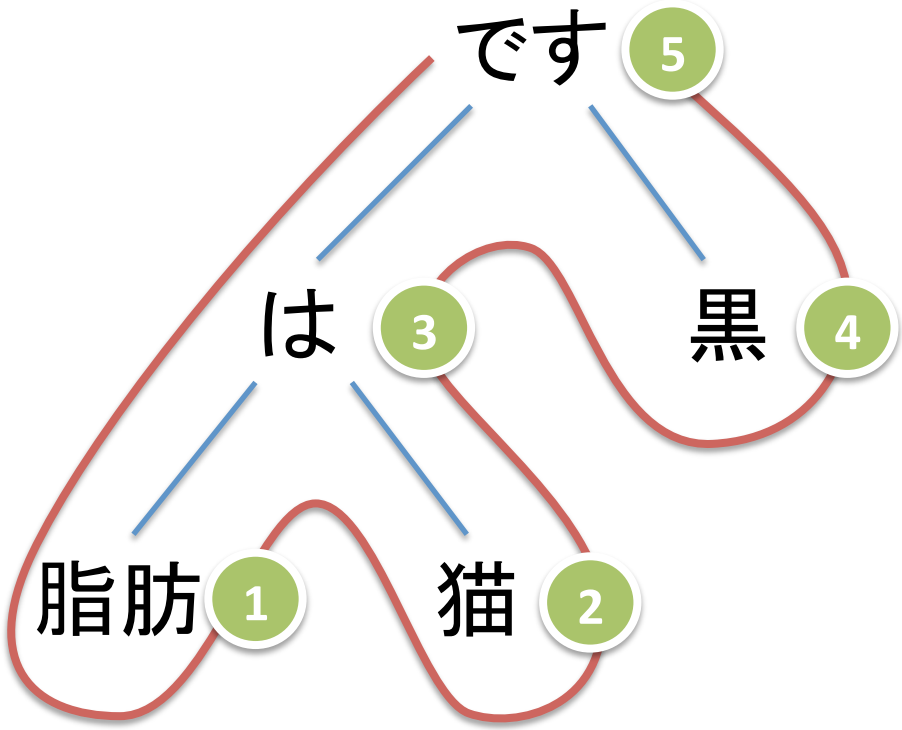
Why bothering ? Language translation



Why bothering ? Language translation



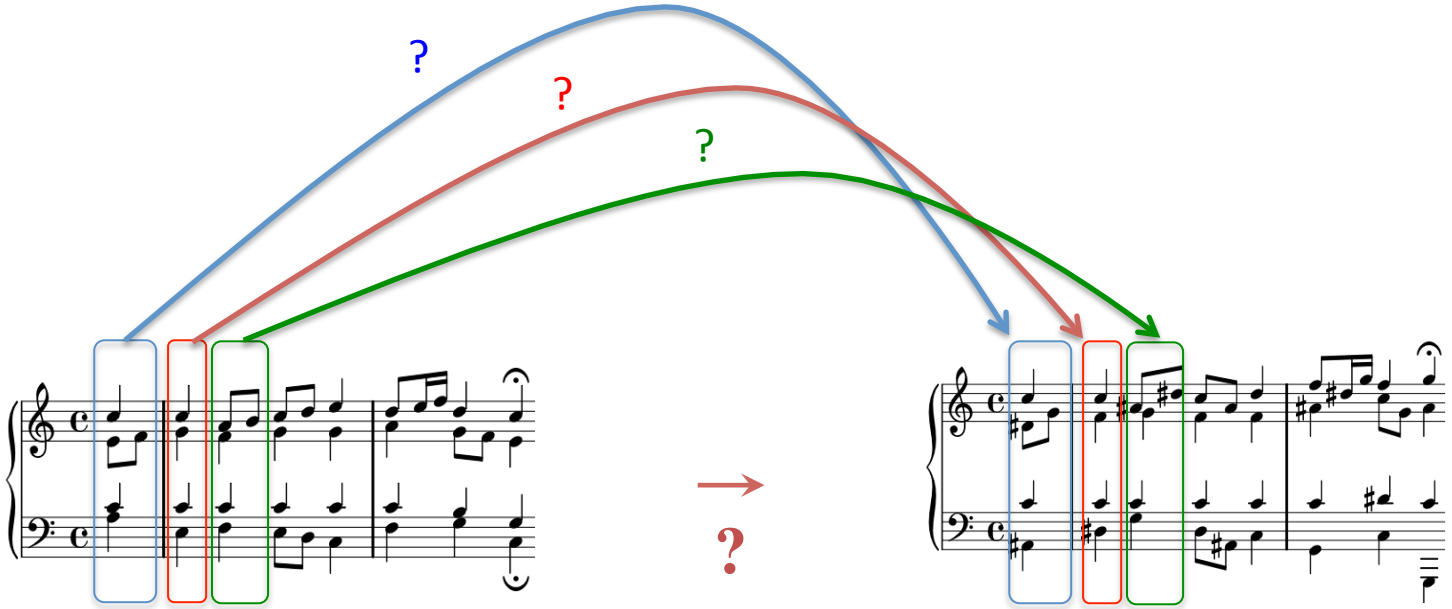
Fat cat is black



脂肪猫は黒です

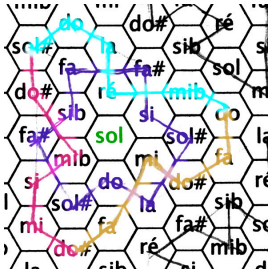
Why bothering ? Music

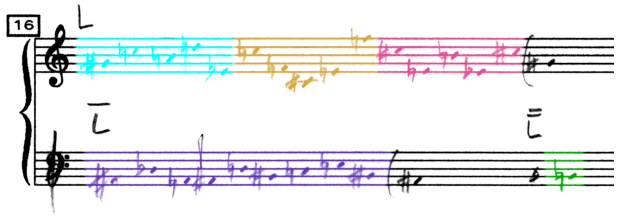
- Musical transformations of score can be tedious
- Need to segment the chords and transform them independently

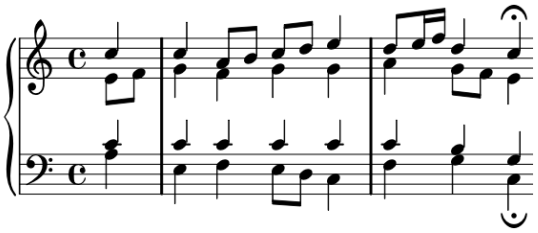


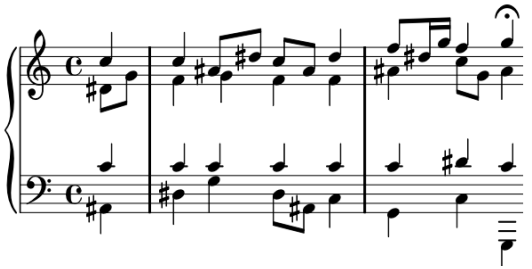
Why bothering ? Music

- Musical transformations of score can be tedious
- Need to segment the chords and transform them independently
- Work on spatial structures by Louis Bigo (IRCAM – RepMus)



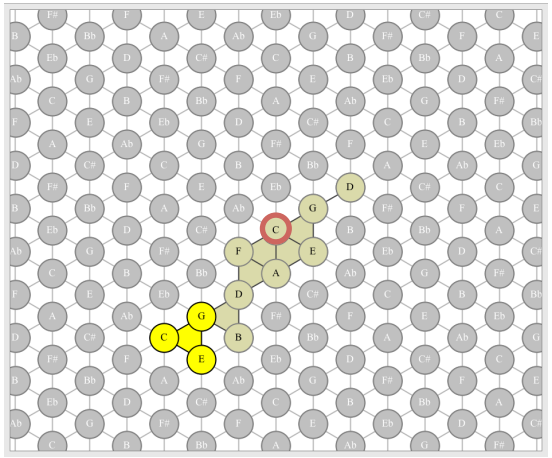




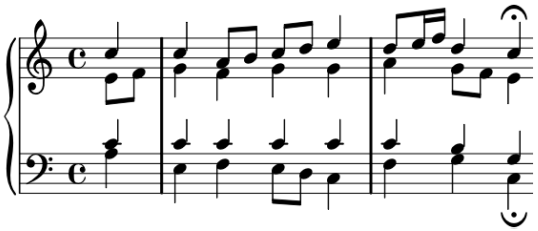


Why bothering ? Music

- Musical transformations of score can be tedious
- Need to segment the chords and transform them independently
- Work on spatial structures by Louis Bigo (IRCAM – RepMus)



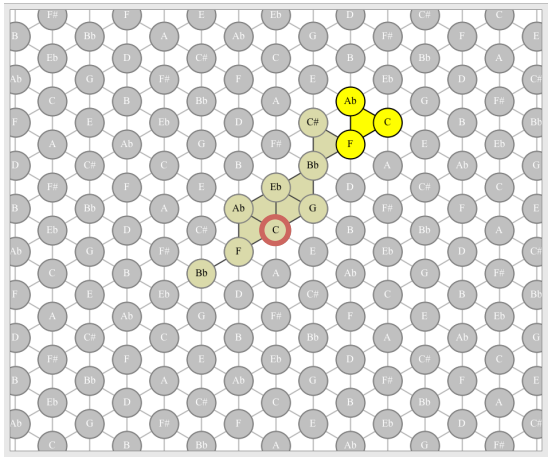
$\mathcal{K}^{u}_{TI}[3,4,5]$



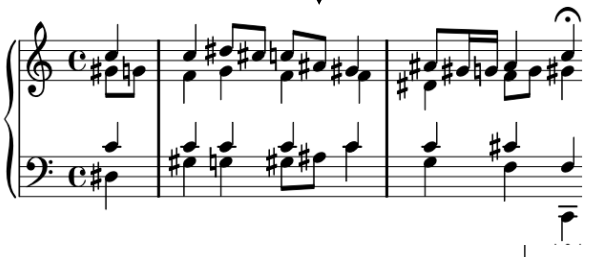
point reflection

The Beatles
Let it be

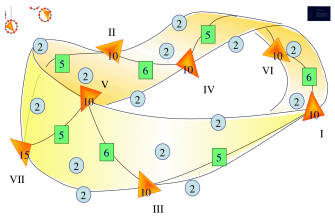
pitch inversion



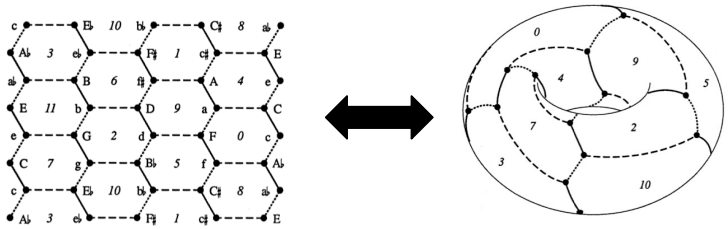
$\mathcal{K}^{u}_{TI}[3,4,5]$



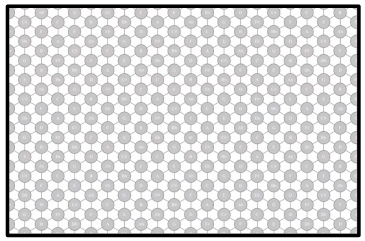
Why bothering ? Music



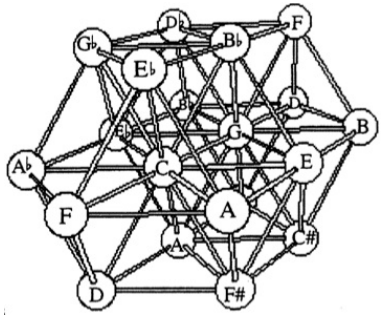
Tonality strip [Mazzola]



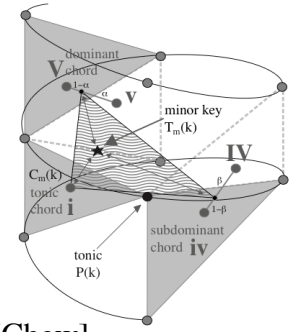
Chicken Wire Torus [Douthett & Steinbach]



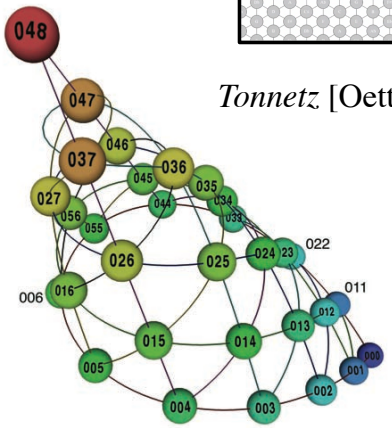
Tonnetz [Oettingen, Riemann]



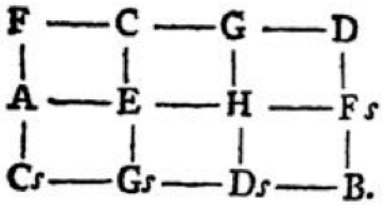
3D *Tonnetz* [Gollin]



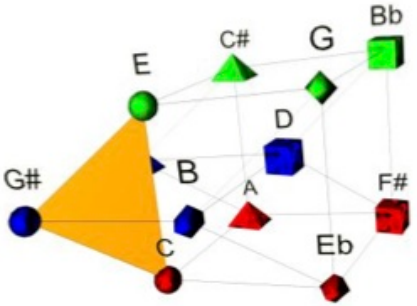
Spiral Array [Chew]



Orbifolds [Tymoczko]



Speculum Musicum [Euler]



Model Planet [Barouin]

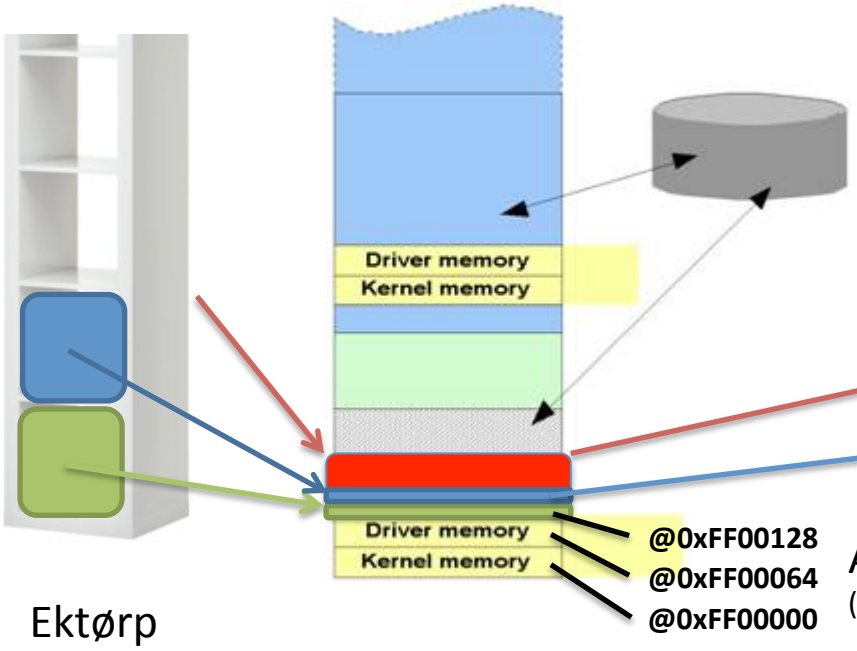
Notation and syntax

- All this lesson is in **pseudo-code** (python-like notation)
- Most language hold the same notations
- Learning syntax is a job for … Google
- Along this lesson, I will make **non-stop roundtrips** from
  - **High-level pseudo-code** and algorithms
  - **Low-level memory** representation problems
- It is very important that you **understand all levels**
- **Check if you know at least**

[variables] / [functions] / [if] / [for] / [while]

Arrays

- The simplest **structure** in informatics
- So how do they look like ?



Ektørp

Operations

Allocation

ikeaShelf = new Array(4);

Access

ikeaShelf[0];

Modification

ikeaShelf[1] = shoes;

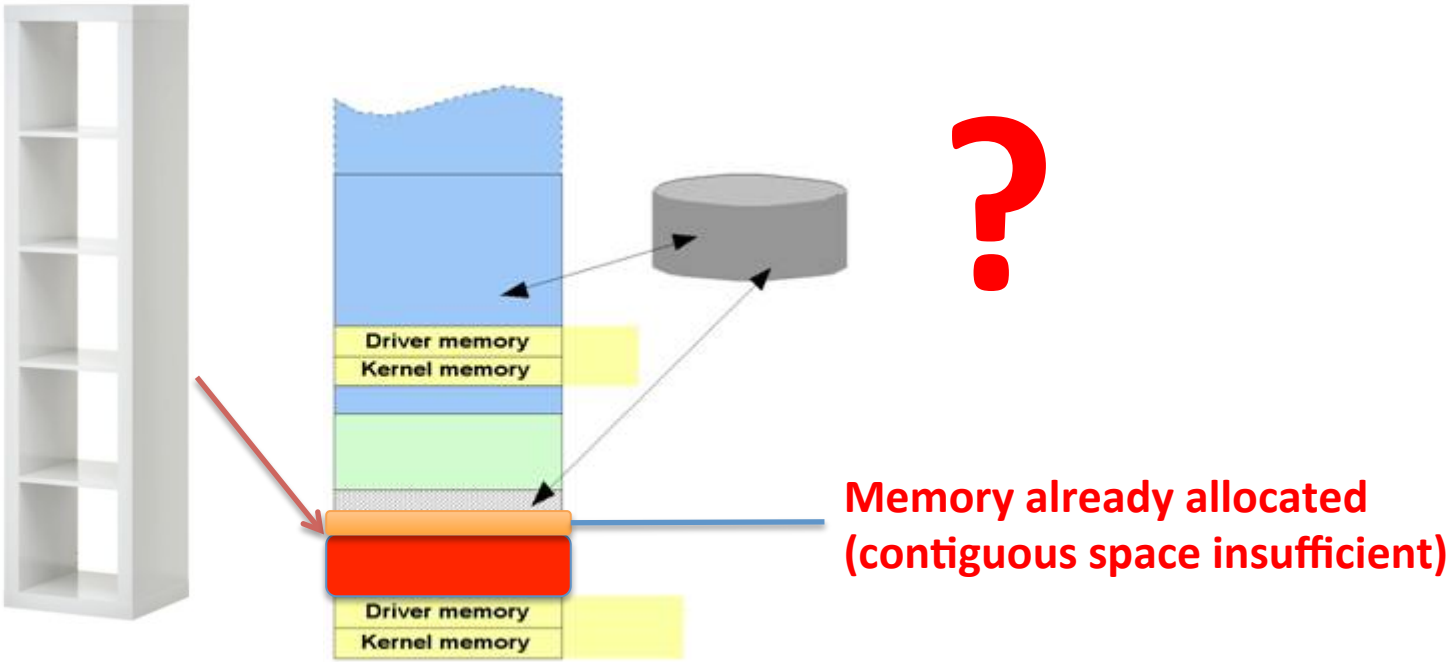
Memory block allocated for the array

Each slot has **an address**

Address are **refered by pointers**
(pointers are long int giving the corresponding memory slot)

Arrays

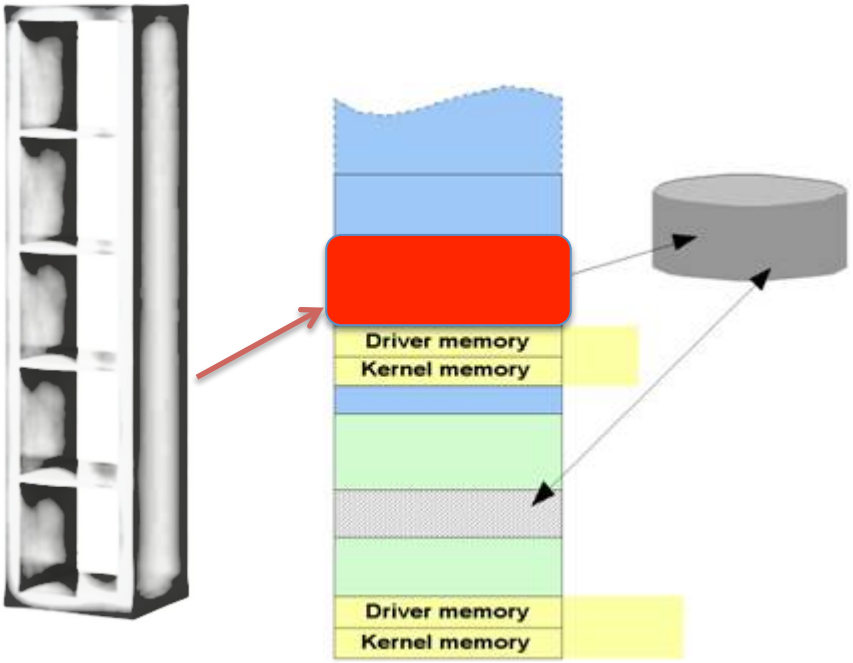
- The simplest **structure** in informatics
- So how do they look like ?



Ektørp

Arrays

- The simplest **structure** in informatics
- So how do they look like ?



Ektørp

System will automatically (without saying)

- Allocate new space (contiguous)
- Transfer each data into this space
- Remove the old space

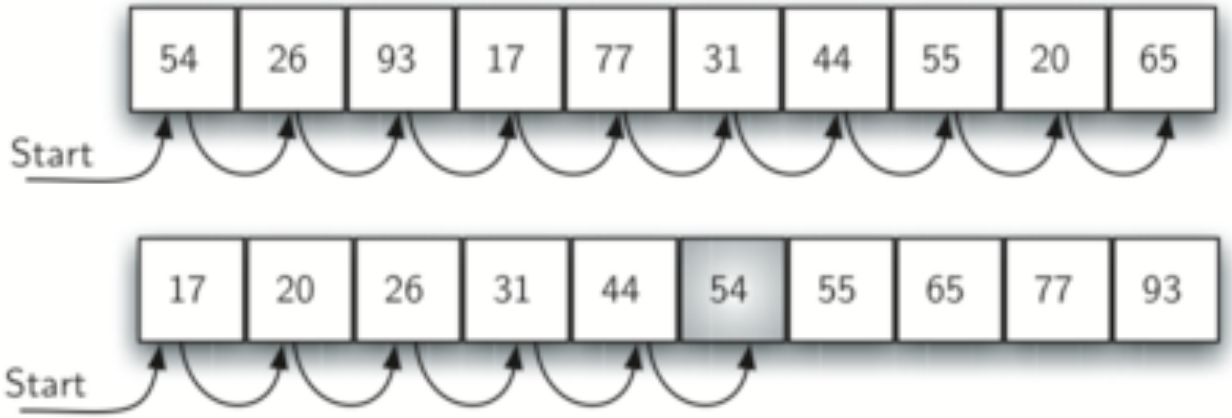
Typically a **static structure** with operations

- Create
- Access
- Modify
- Reallocate

Arrays: complexity

Access	O(1)
Modification	O(1)
Search	O(n)

Typical static structure complexities



Sequential search

Sorted case

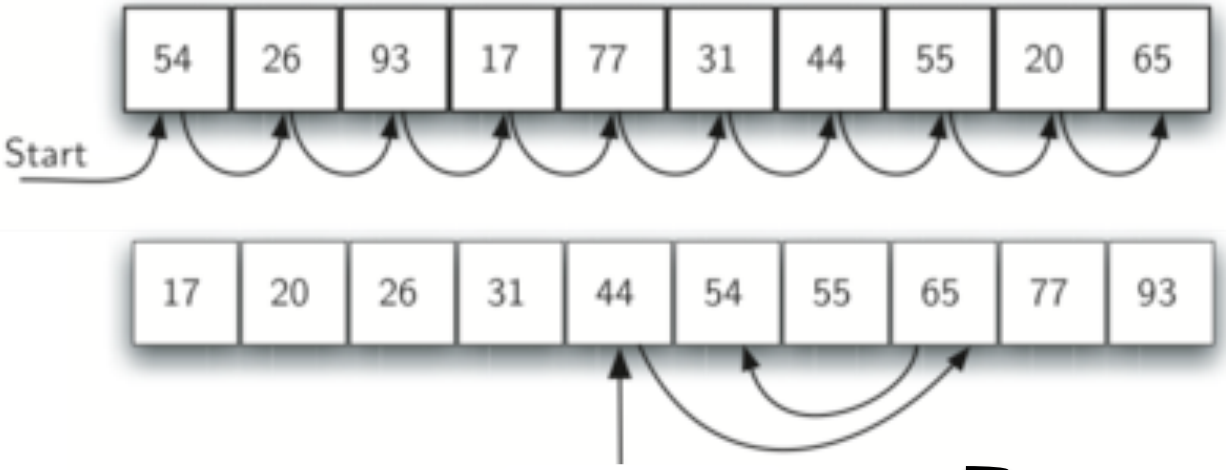
```
for (i = 0; i < intArray.length; i++)
 if (intArray[i] >= 47)
   break;
```

O(n)

Arrays: complexity

Access	O(1)
Modification	O(1)
Search	O(n)

Typical static structure complexities



Sequential search

Sorted case – Binary search
(divide and conquer) !

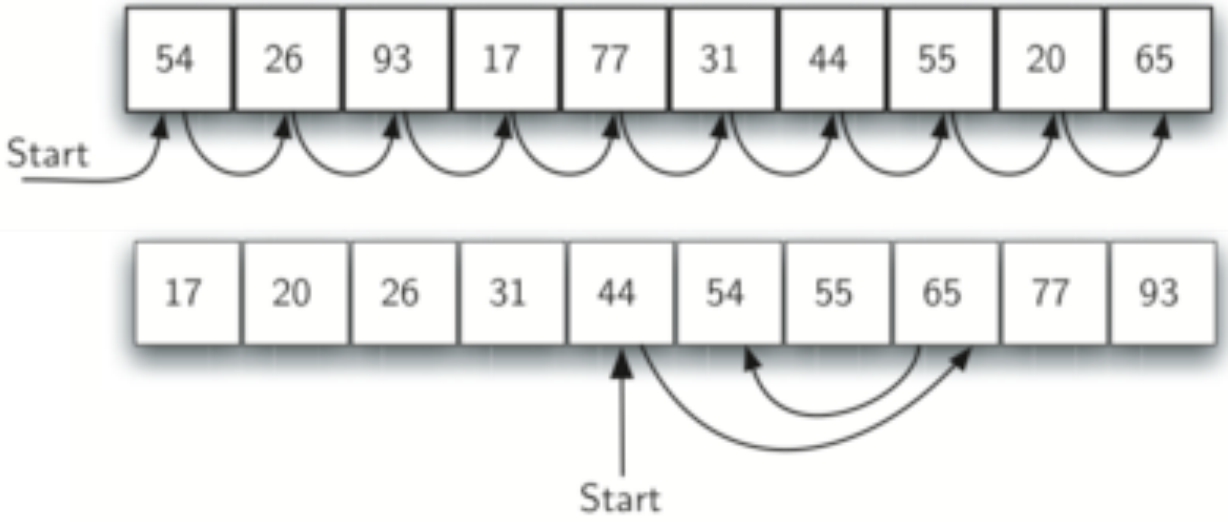
```
while (first <= last && not found)
  midp = (first + last) / 2
  if (intA[mid] == 47) break;
  if (intA[mid] < 47) first = mid;
  else last = mid;
```

O(log(n)) !

Arrays: complexity

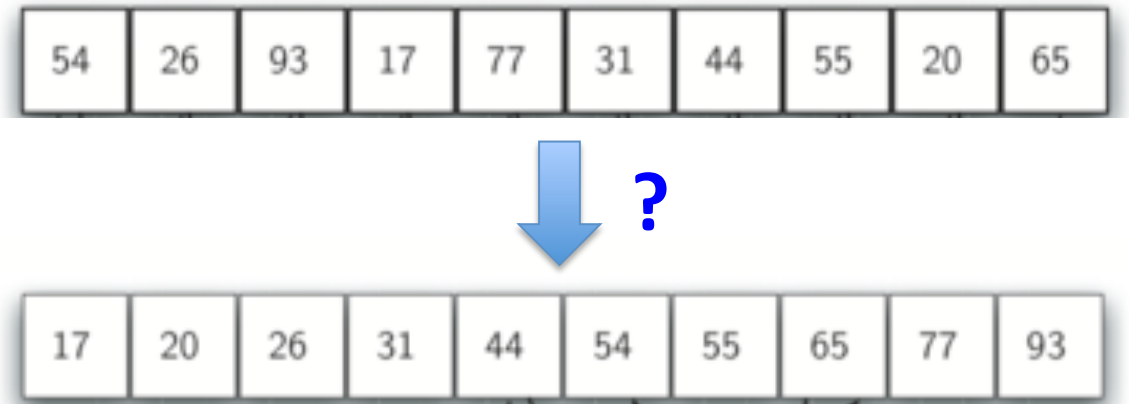
Access	O(1)
Modification	O(1)
Search	O(n)

Typical static structure complexities



One of the earliest algorithm on data structures
=> **Array sorting**

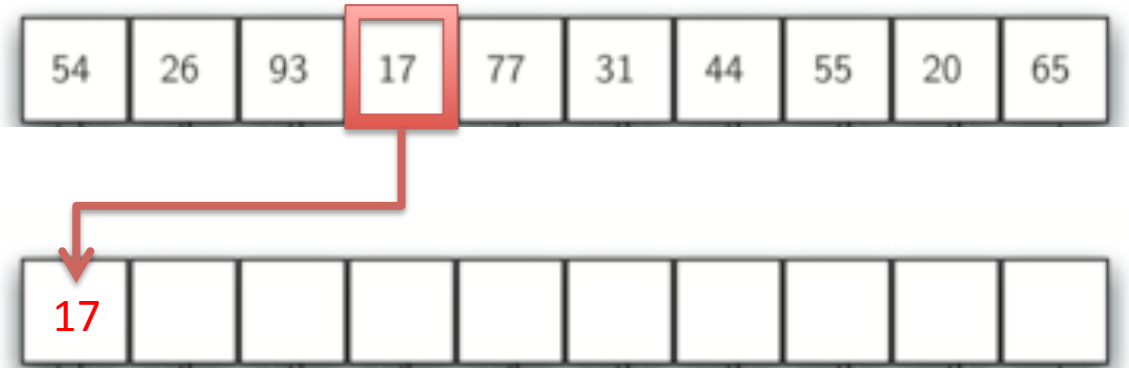
Arrays: sorting



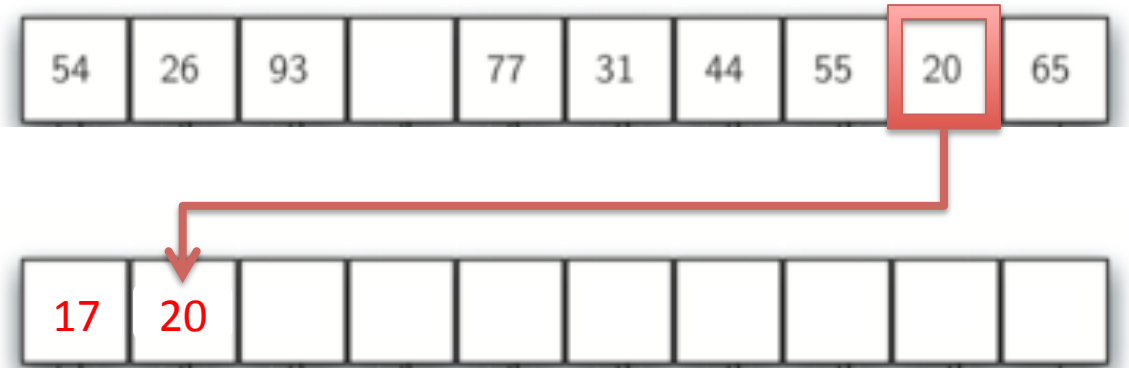
In fact lots of ways to sort an array
Going from **$O(n^3)$** down to O(n.log(n))

- Insertion sort
- Bubble sort
- Selection sort
- Merge sort
- Quick sort

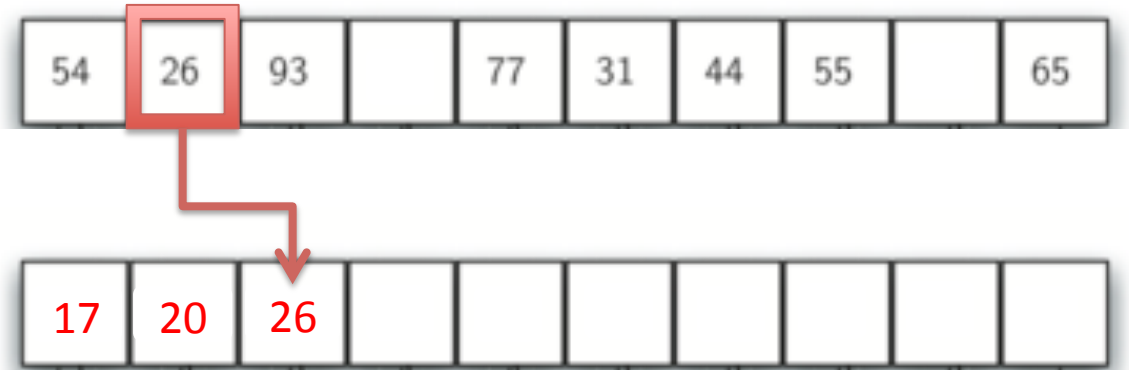
Arrays: Naïve insertion sorting



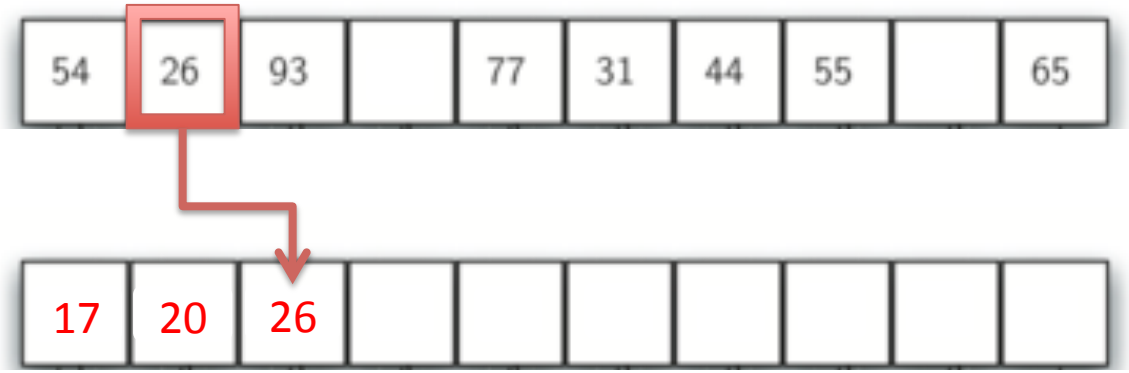
Arrays: Naïve insertion sorting



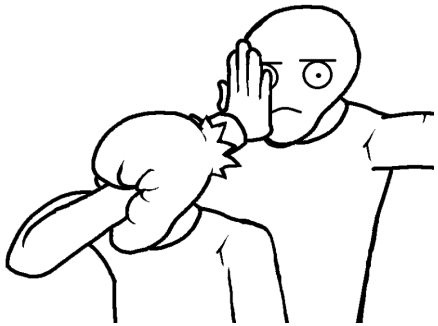
Arrays: Naïve insertion sorting



Arrays: Naïve insertion sorting




```
curPos = 0;
sorted = new Array(intArray.length);
for (i = 0; i < intArray.length; i++)
 curMin = findMinAndRemove(intArray);
 sorted[curPos++] = curMin;
```



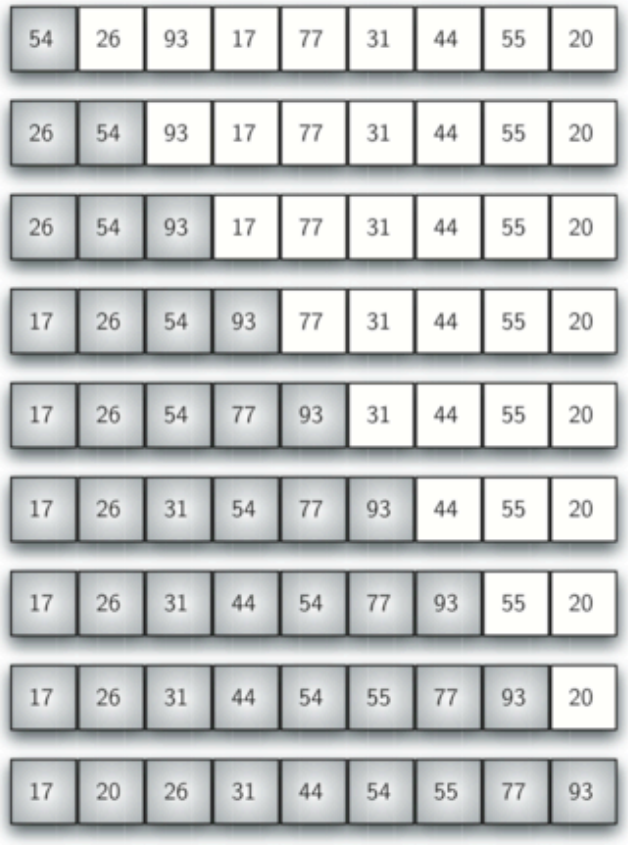
WORST SORTING ALGORITHM EVER

Best case complexity : $o(n^2)$

(+ double memory allocation)

Arrays: Insertion sort

Arrays: Insertion sort

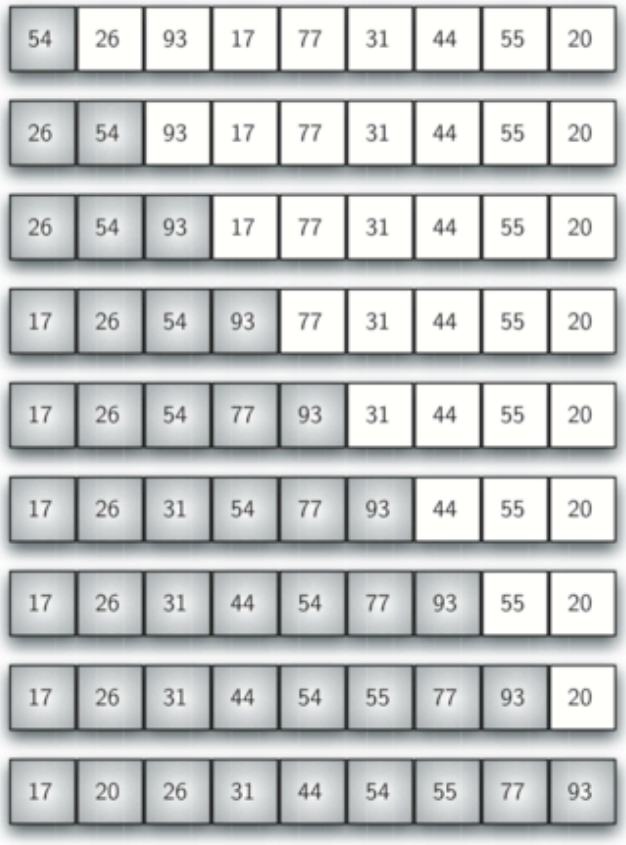



```
for (id = 1; id < array.length; id++):
      current = array[index];
      pos = id;
      while (pos > 0 && array[pos-1]>current)
          array[pos] = array[pos - 1];
          pos = pos – 1;
      array[pos] = current;
```

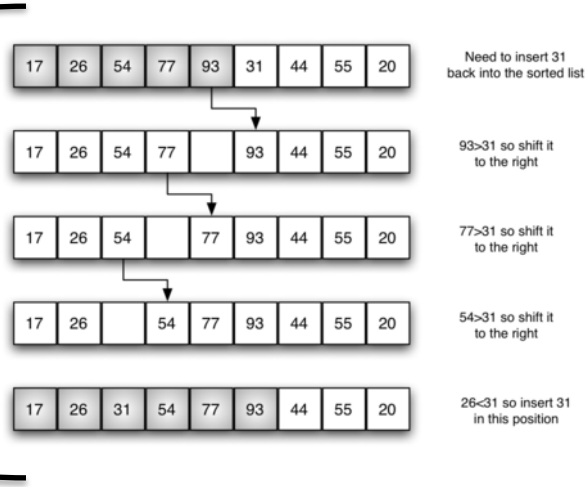
Worst case : $O(n^2)$

Best case : O(n)

Arrays: Insertion sort



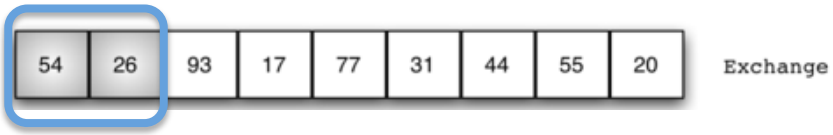

```
for (id = 1; id < array.length; id++):
    current = array[index];
    pos = id;
    while (pos > 0 && array[pos-1]>current)
```



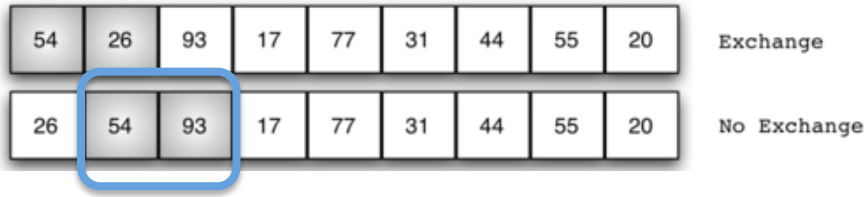

```
    array[pos] = current;
```

Arrays: Bubble sort

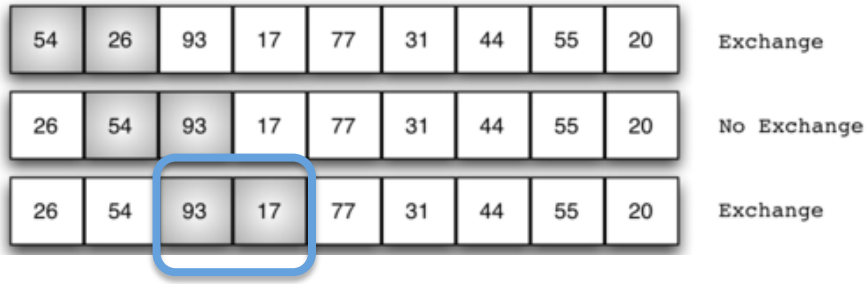
Does the bubble need swapping ?



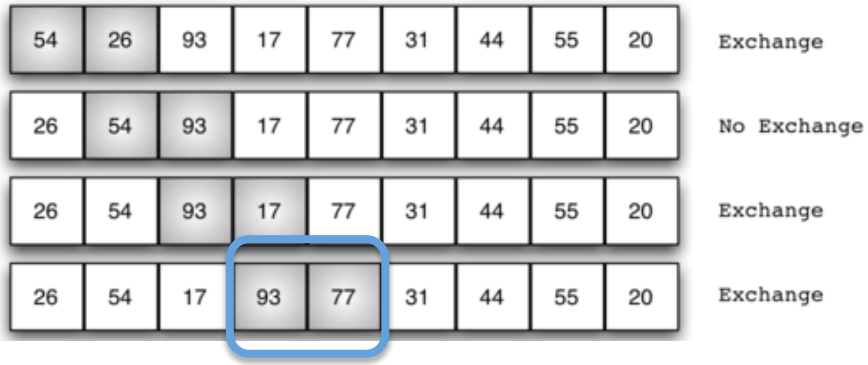
Arrays: Bubble sort



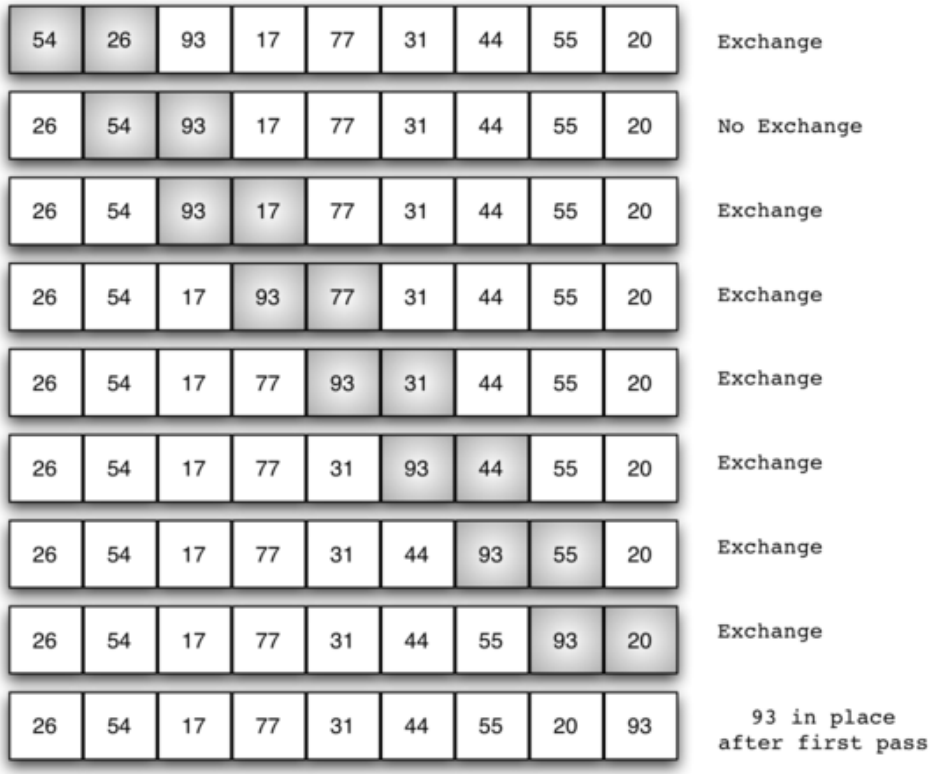
Arrays: Bubble sort



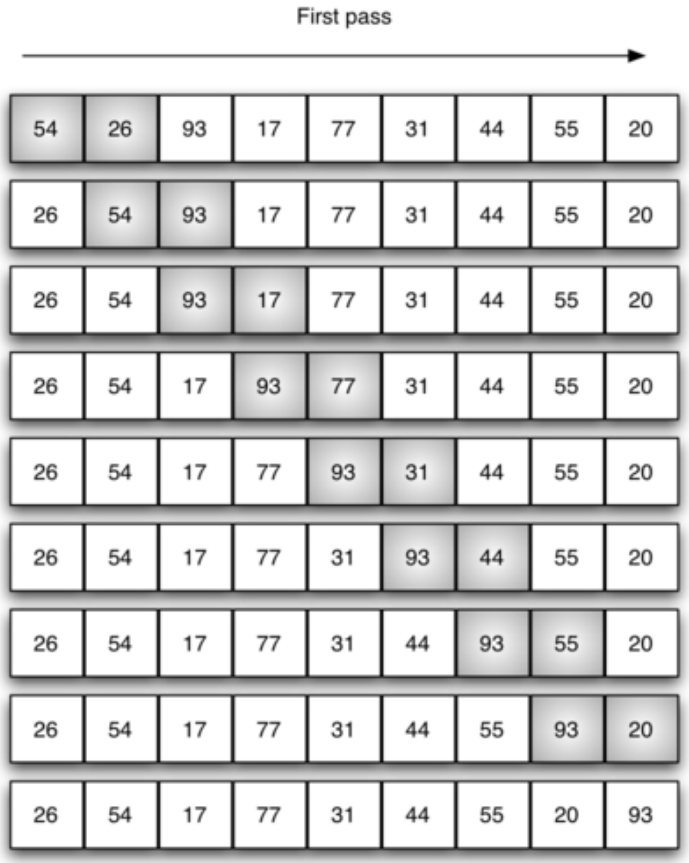
Arrays: Bubble sort



Arrays: Bubble sort



Arrays: Bubble sort




```
for (p = array.length - 1; p > 0; p--)
  swap = 0;
  for (i = 0; i < p; i++)
    if (array[i] > array[i + 1])
        temp = array[i];
        array[i] = array[i+1];
        array[i+1] = temp;
        swap = 1;
  if (swap == 0)
     break;
```

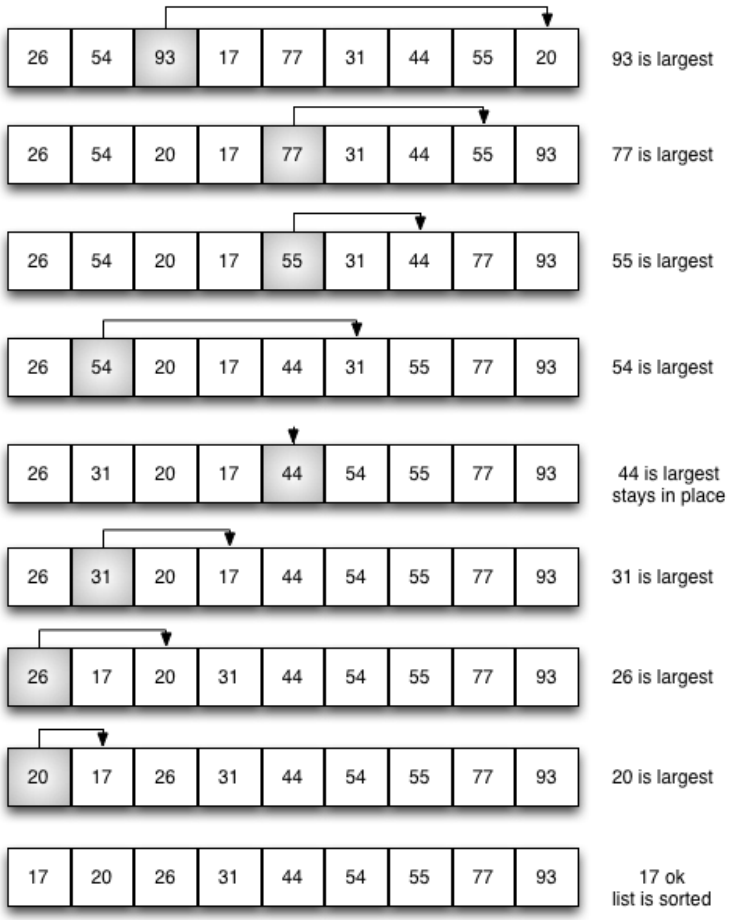
Still …

Worst case : O(n^2) (But in reality … n.(n-1)/2)

Best case : O(n)

And less memory swapping !

Arrays: Selection sort



- Combining bubble and insertion sort
- Only swap the largest with the last unchanged (depending on pass index)

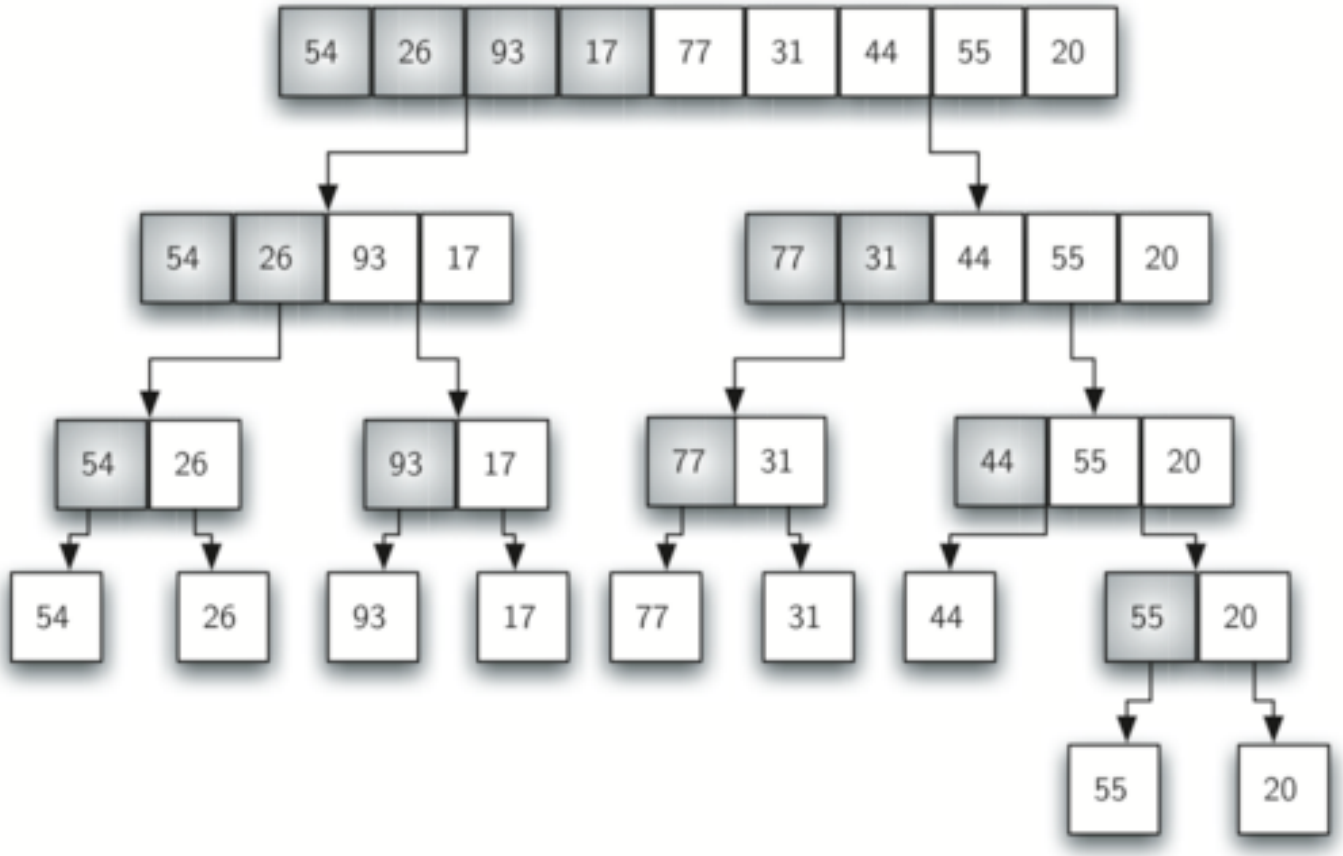
```
for (p = array.length - 1; p > 0; p--)
  max = 0;
  for (i = 0; i < p; i++)
    if (array[i] > array[max])
        max = i;
  temp = array[p];
  array[p] = array[max];
  array[max] = temp;
```

Same complexity as bubble sort …
But a **lot less memory swapping** (which is the most expensive operation)

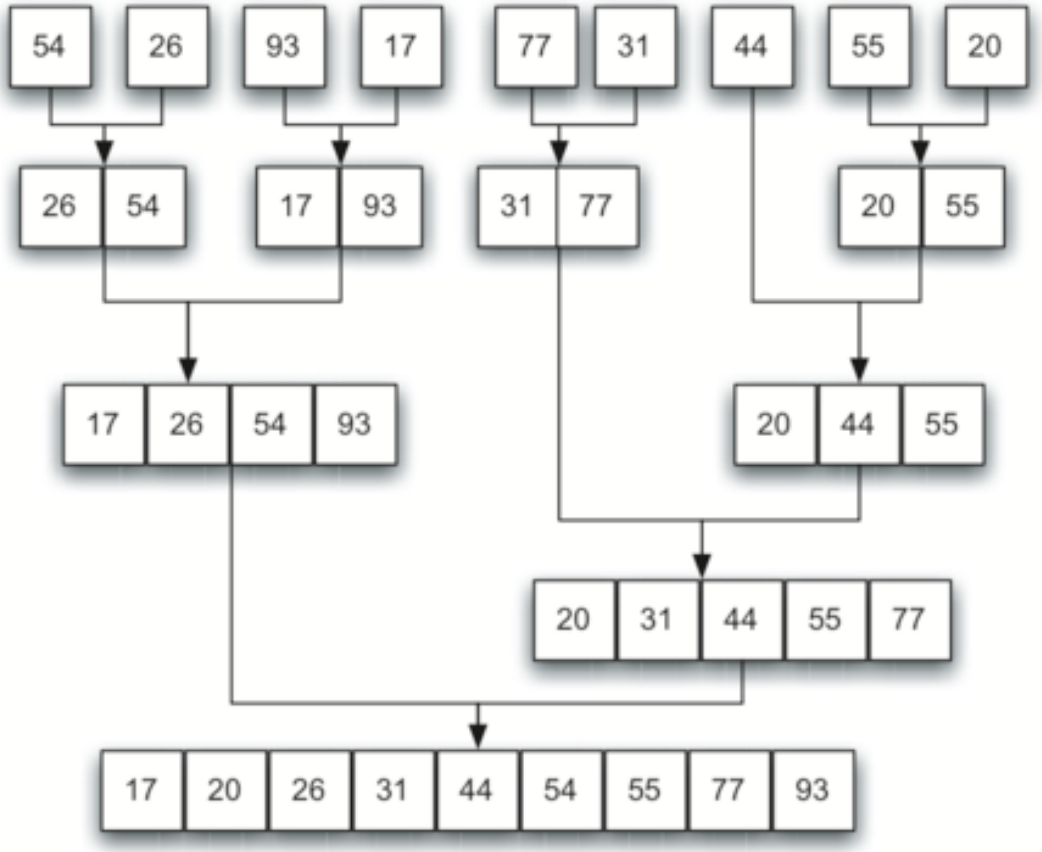
Arrays: Merge sort

DIVIDE AND CONQUER

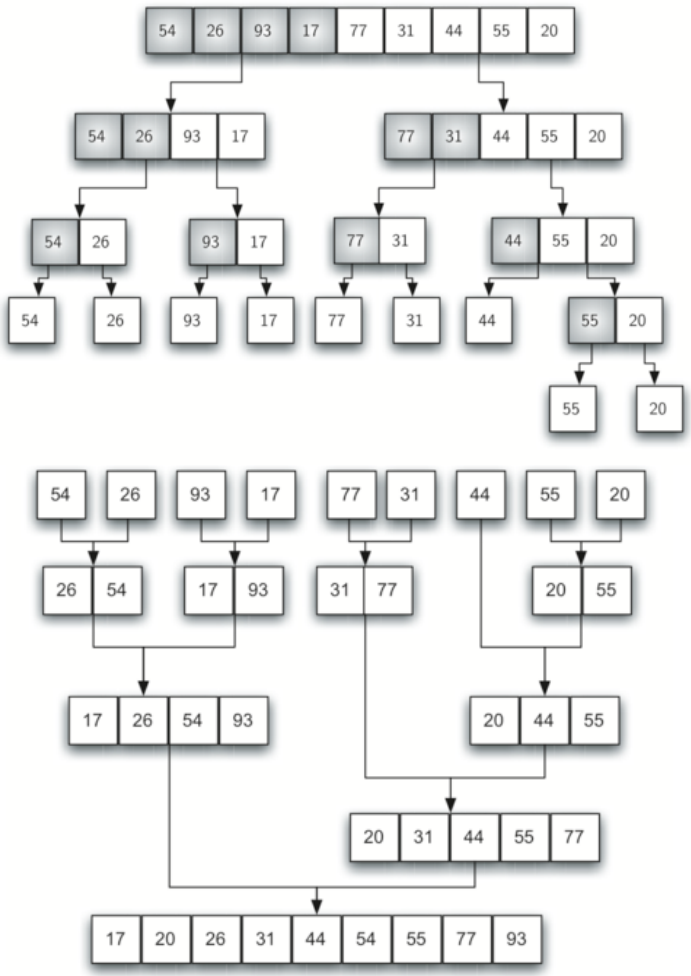
Arrays: Merge sort



Arrays: Merge sort



Arrays: Merge sort

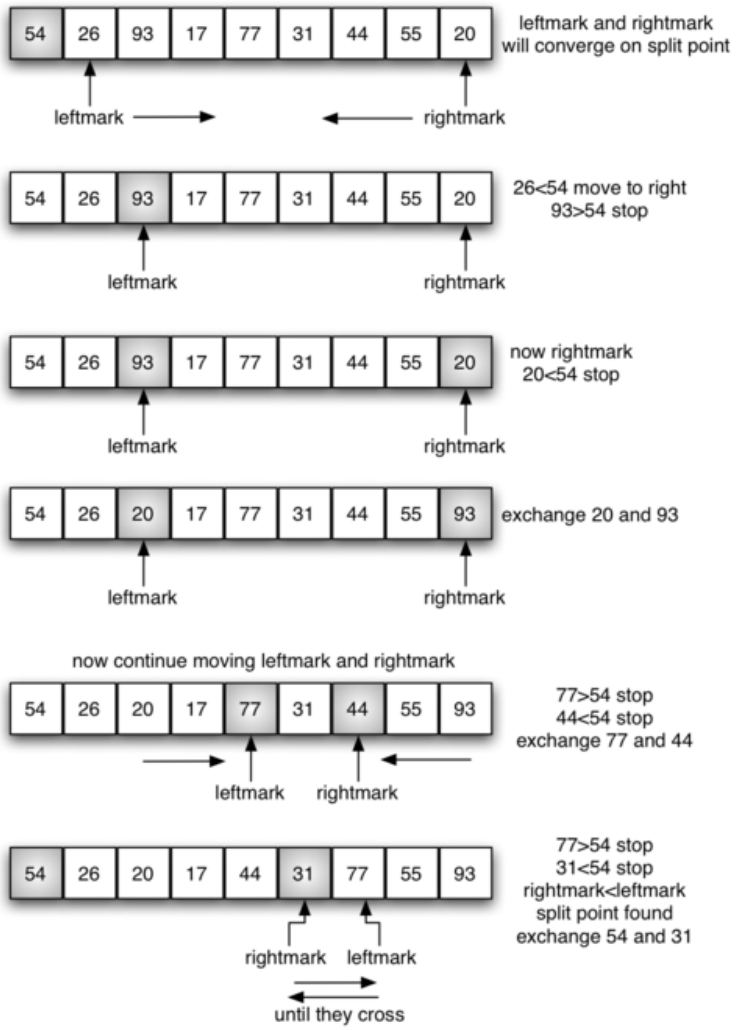



```
mergeSort(alist)
 if (alist.length > 1)
  mid = alist.length/2;
  left = alist[1:(mid-1)];
  right = alist[mid:1];
  mergeSort(left);
  mergeSort(right);

  i=0;j=0;k=0
  while (i < left.len and j < right.len)
    if (left[i] < right[j])
      alist[k] = left[i]
      i = i + 1;
    else
      alist[k] = right[j]
      j = j + 1; k = k + 1;
  while (i < left.len)
      alist[k] = left[i]
      i = i + 1; k = k + 1;
  while (j < len(righthalf)
      alist[k]=righthalf[j]
      j = j + 1; k = k + 1;
```

But requires additional storage

Complexity : O(n.log(n))

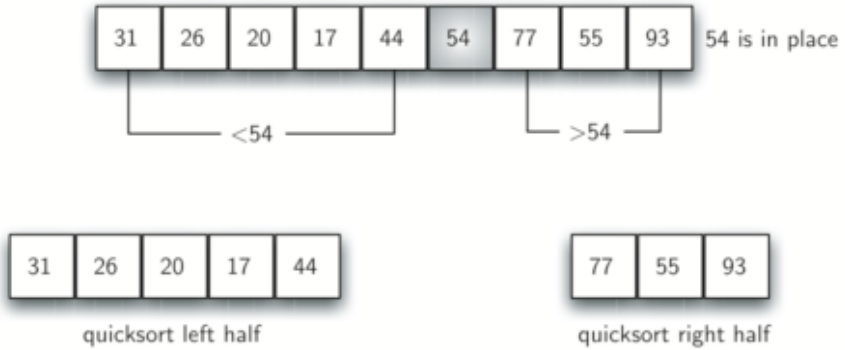
Arrays: Quick sort



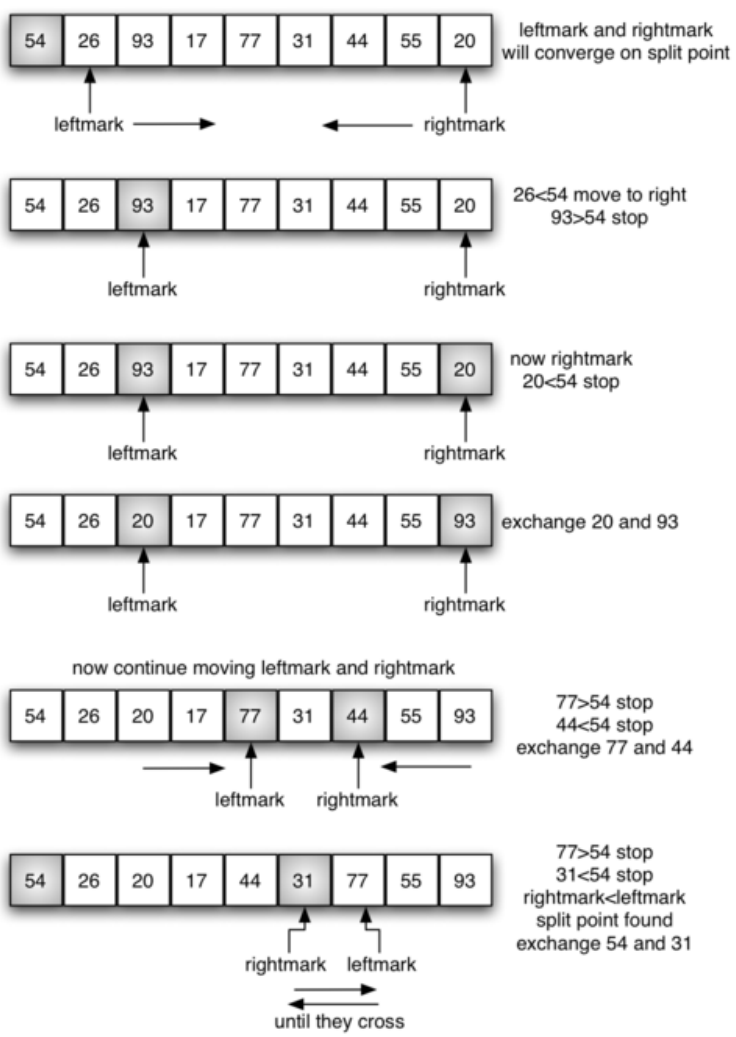
- Same divide and conquer as merge sort
- Without additionnal storage
- Based on pivot value and marking points

- O(n.log(n))
- No extra storage
- Best sorting

- So at the end of this procedure ?
- If we exchange our **pivot** with our **rightmark**
- … We just « space-partitionned » the array !

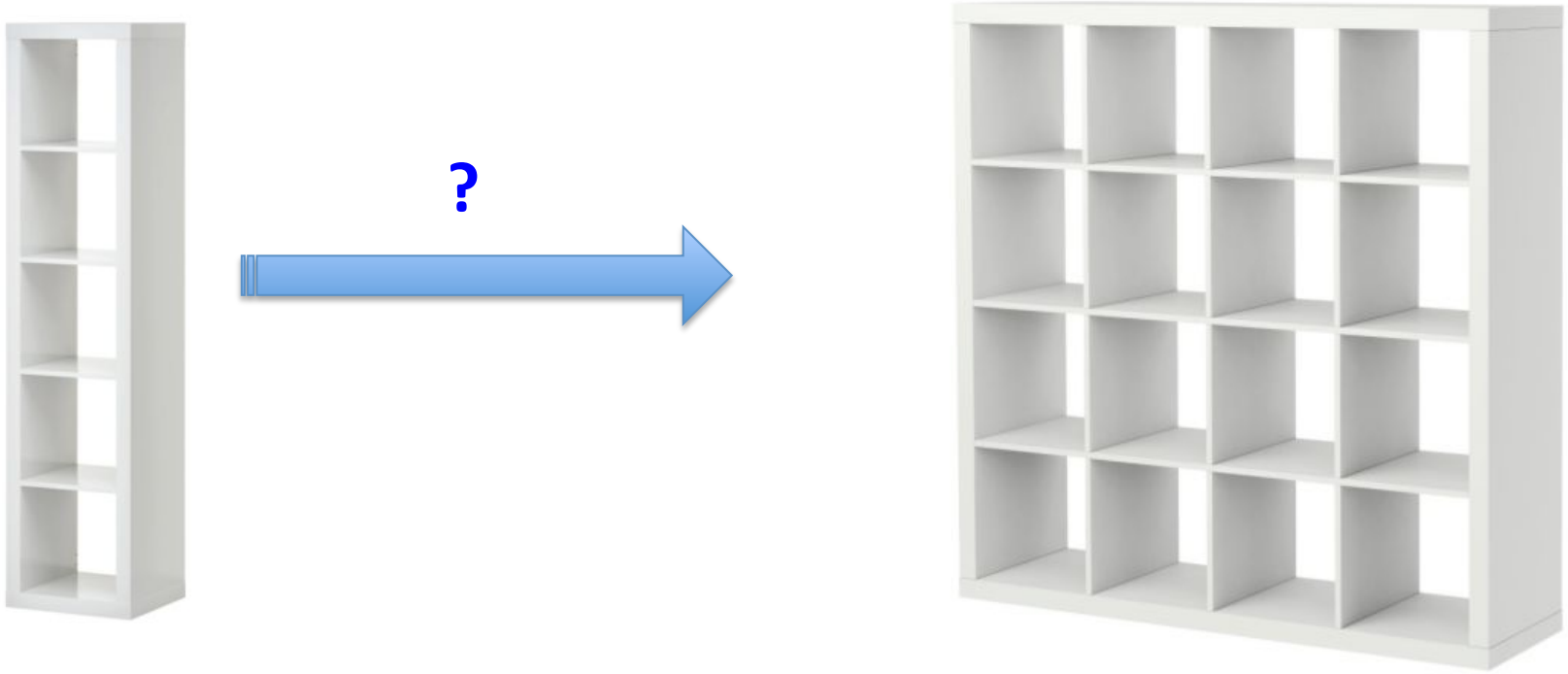


Arrays: Quick sort

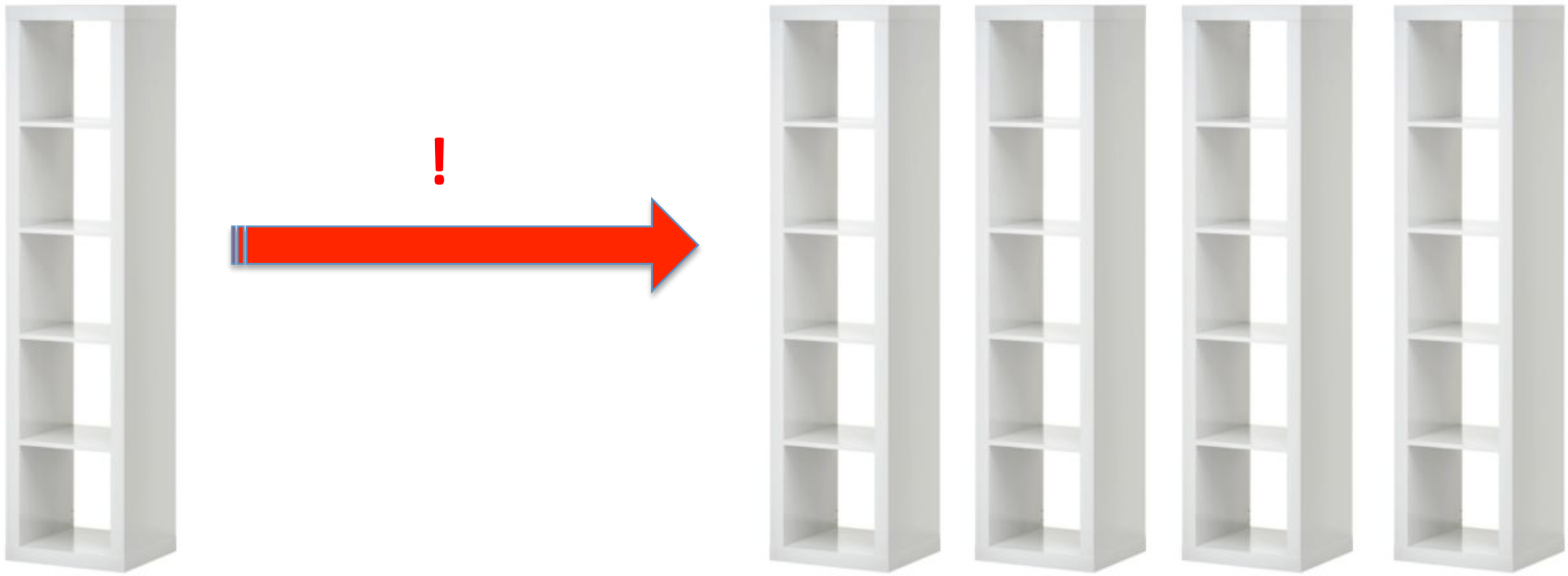



```
def quickSort(alist):
   quickSortHelper(alist,0,len(alist)-1)

def quickSortHelper(alist,first,last):
   if (first < last)
       splitpoint = partition(alist,first,last)
       quickSortHelper(alist,first,splitpoint-1)
       quickSortHelper(alist,splitpoint+1,last)

def partition(alist,first,last)
   pivotvalue = alist[first];
   leftmark = first + 1;
   rightmark = last;
   done = False;
   while (not done)
       while (leftmark <= rightmark and
                alist[leftmark] <= pivotvalue)
           leftmark = leftmark + 1;
       while (alist[rightmark] >= pivotvalue and
                rightmark >= leftmark)
           rightmark = rightmark - 1;
       if (rightmark < leftmark)
           done = True;
       else
           temp = alist[leftmark];
           alist[leftmark] = alist[rightmark];
           alist[rightmark] = temp;
   temp = alist[first];
   alist[first] = alist[rightmark];
   alist[rightmark] = temp;
   return rightmark;
```

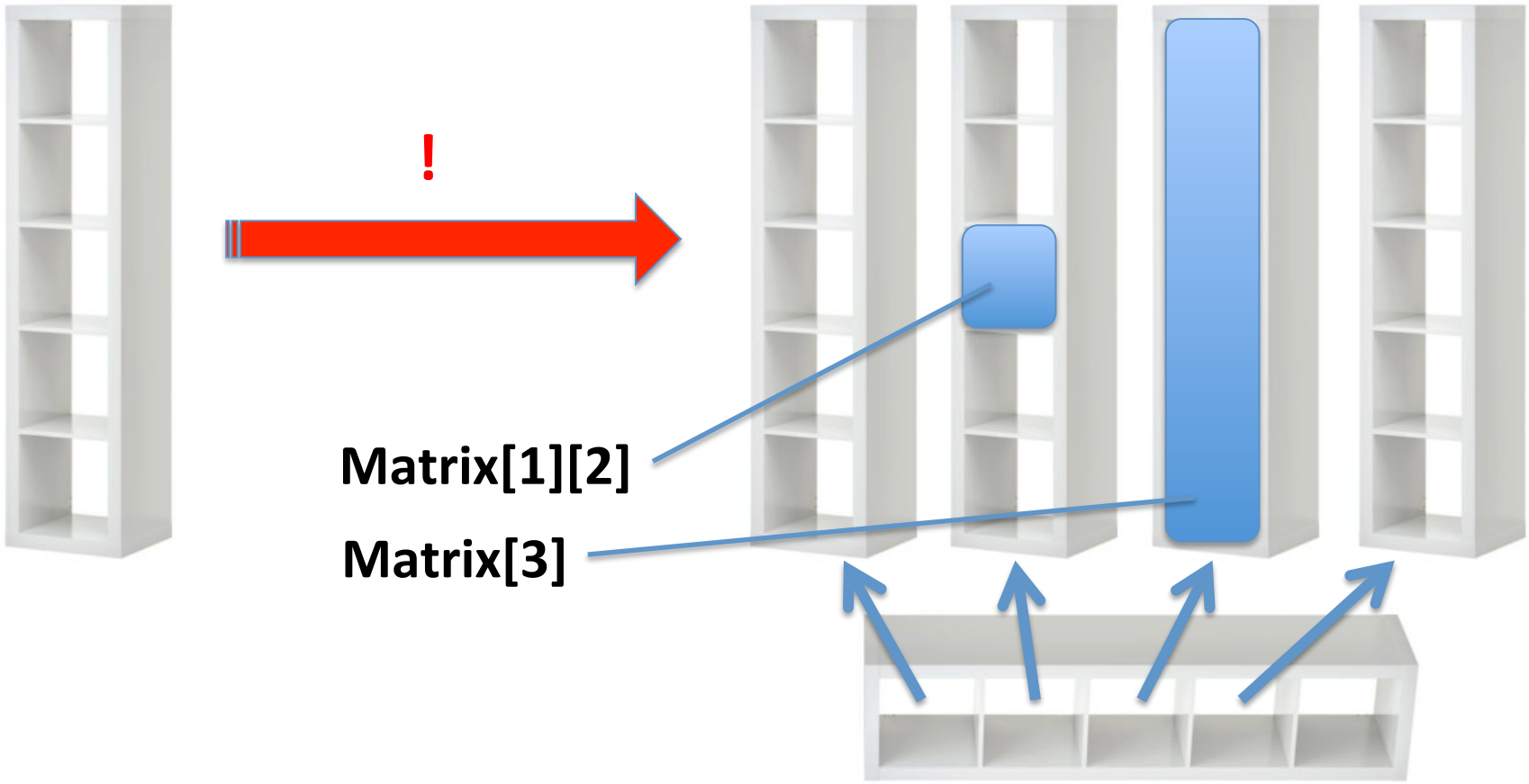
What about matrix?



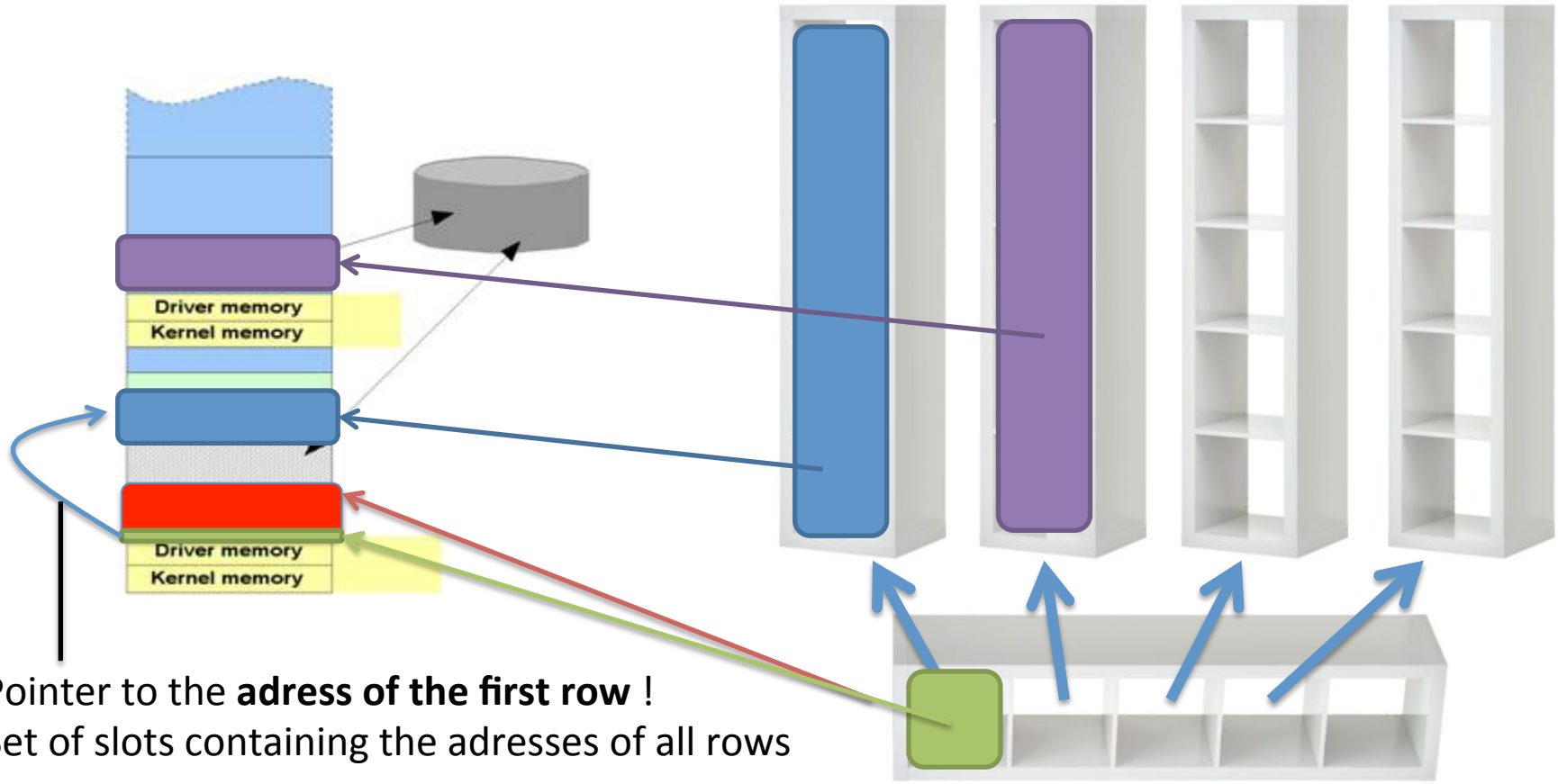
Matrix !



Matrix !



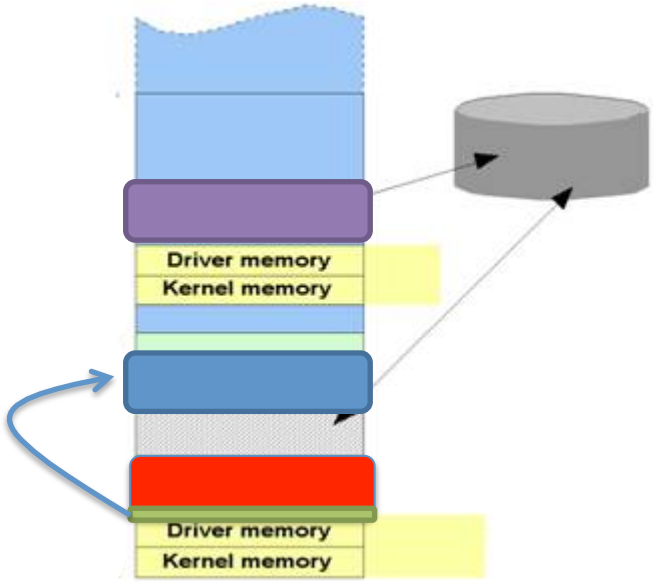
Matrix ! Memory



Pointer to the **adress of the first row** !
Set of slots containing the adresses of all rows

Two things to be a geek

- Two concepts that good'ol programmers know …
- … That youngsters don't give a s**t about …



1 – Cache locality

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    matrix[i][j] = x;
```

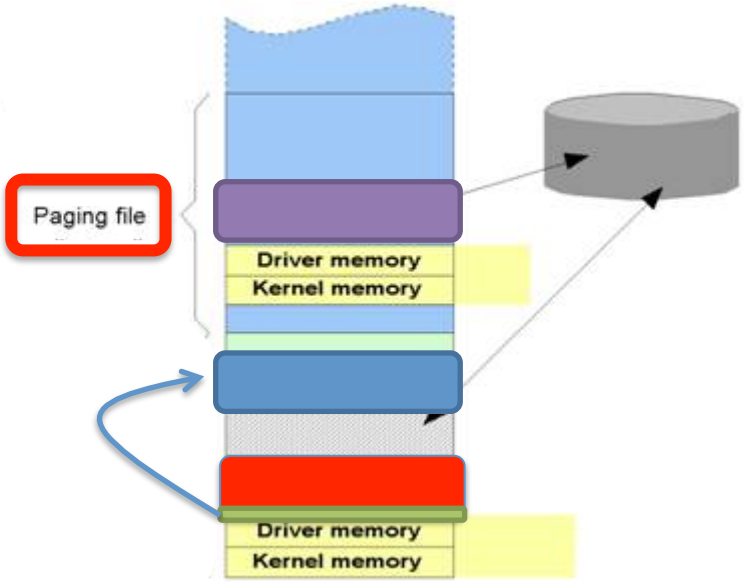
Execution time 1 sec

```
for (j = 0; j < M; j++)
  for (i = 0; i < N; i++)
    matrix[i][j] = x;
```

Execution time 10 min

Two things to be a geek

- Two concepts that good'ol programmers know …
- … That youngsters don't give a s**t about …



1 – Cache locality

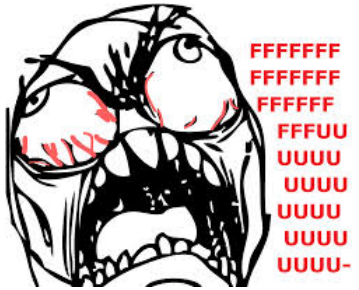
```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    matrix[i][j] = x;
```

Execution time 1 sec

```
for (j = 0; j < M; j++)
  for (i = 0; i < N; i++)
    matrix[i][j] = x;
```

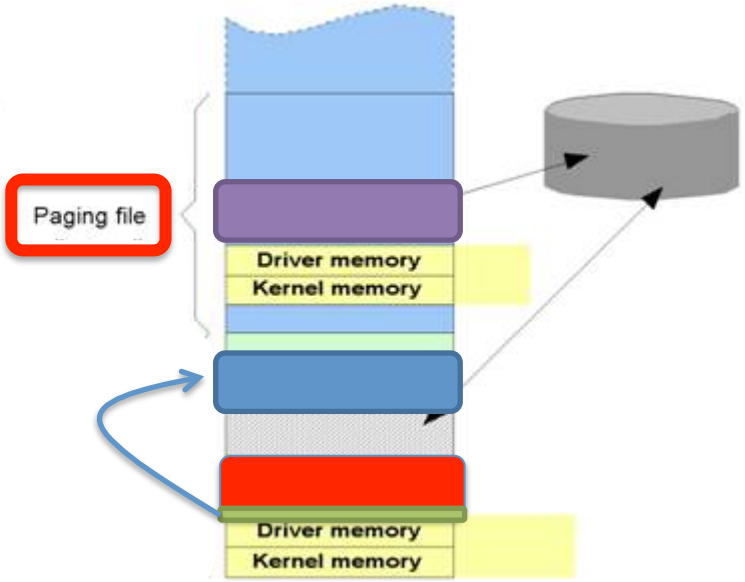
Execution time 10 min

WHY ?



Two things to be a geek

- Two concepts that good'ol programmers know …
- … That youngsters don't give a s**t about …

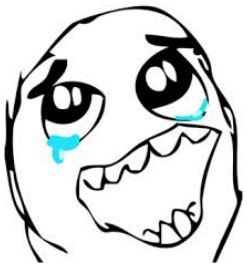


1 – Cache locality

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    matrix[i][j] = x;
```

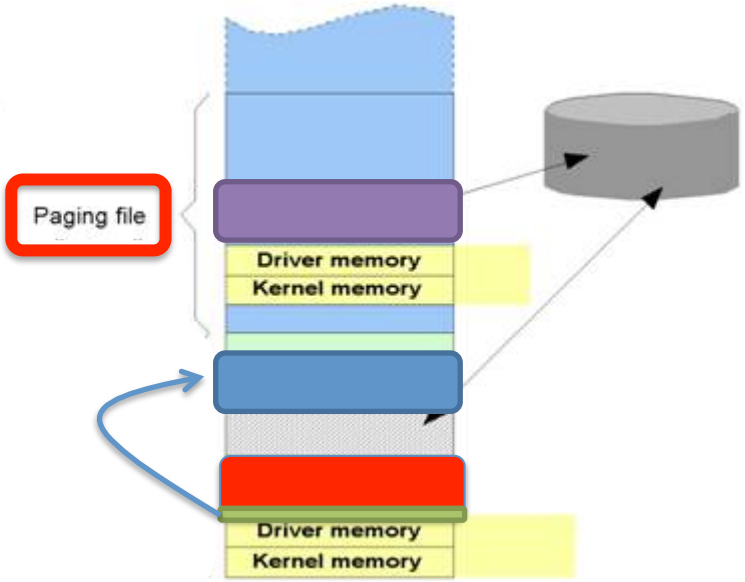
Execution time 1 sec

Always try to process data **contiguously (memory page)**



Two things to be a geek

- Two concepts that good'ol programmers know …
- … That youngsters don't give a s**t about …



2 – Disk swap

```
for (i = 0; i < N - 1; i++)
  for (j = 0; j < M; j++)
    matrix[i][j] = x;
```

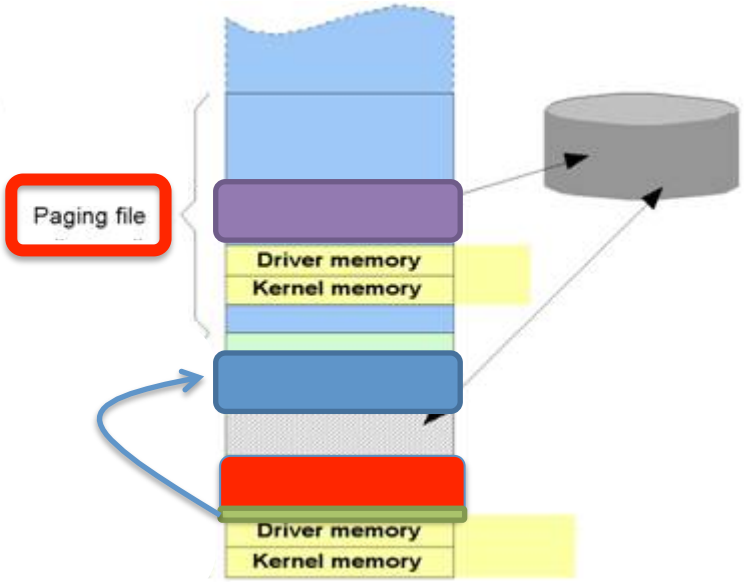
Execution time 1 sec

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    matrix[i][j] = x;
```

Execution time 2 hours

Two things to be a geek

- Two concepts that good'ol programmers know …
- … That youngsters don't give a s**t about …



2 – Disk swap

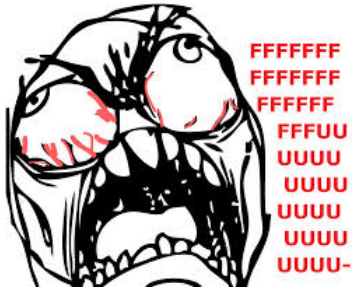
```
for (i = 0; i < N - 1; i++)
  for (j = 0; j < M; j++)
    matrix[i][j] = x;
```

Execution time 1 sec

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    matrix[i][j] = x;
```

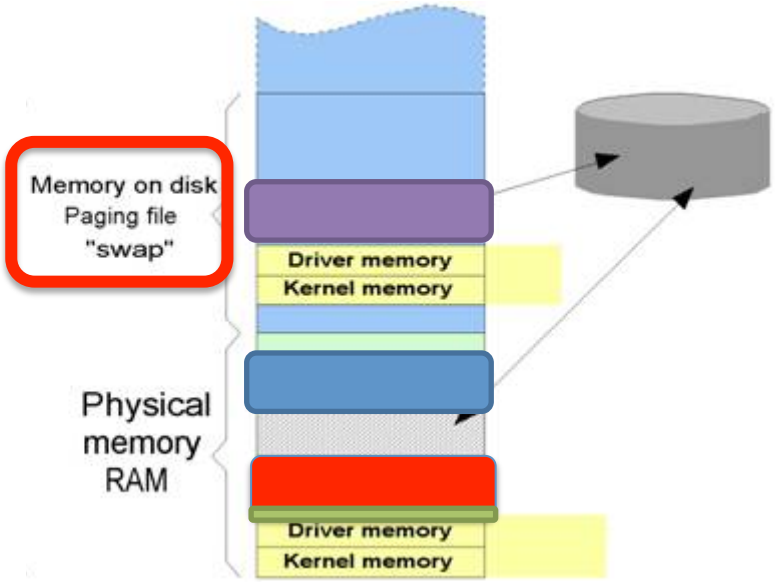
Execution time 2 hours

WHY ?



Two things to be a geek

- Two concepts that good'ol programmers know …
- … That youngsters don't give a s**t about …



2 – Disk swap

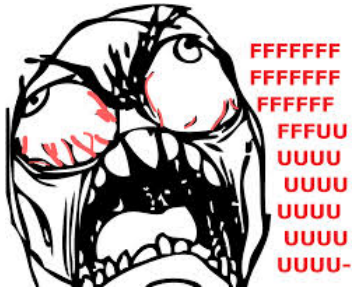
```
for (i = 0; i < N - 1; i++)
  for (j = 0; j < M; j++)
    matrix[i][j] = x;
```

Execution time 1 sec

```
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    matrix[i][j] = x;
```

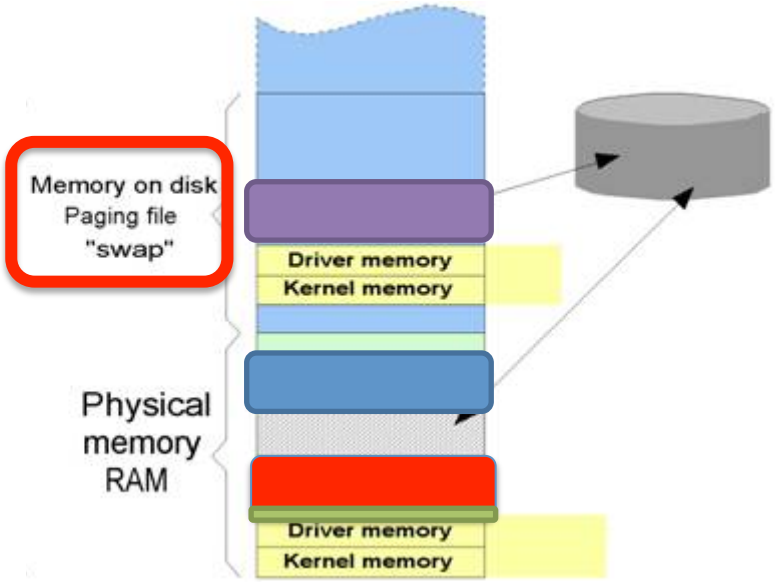
Execution time 2 hours

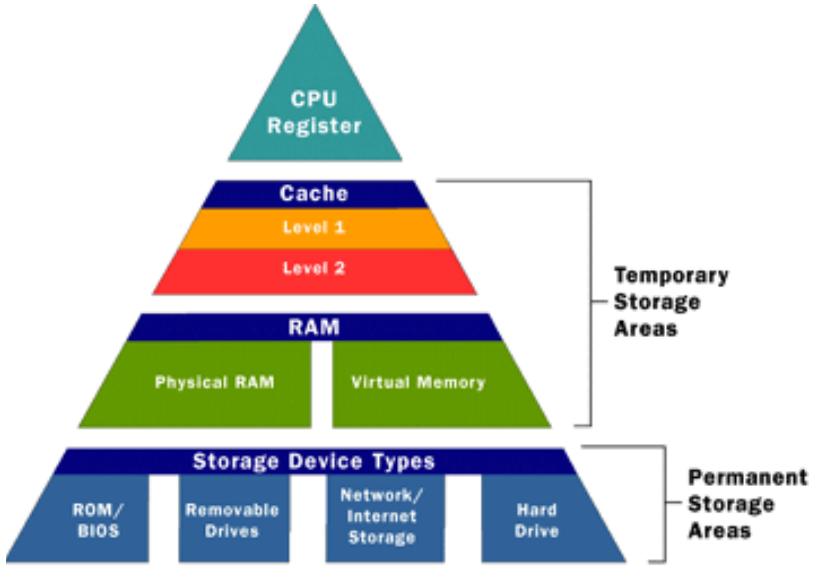
WHY ?



Two things to be a geek

- Two concepts that good'ol programmers know …
- … That youngsters don't give a s**t about …





Access times

1 ns

5 ns

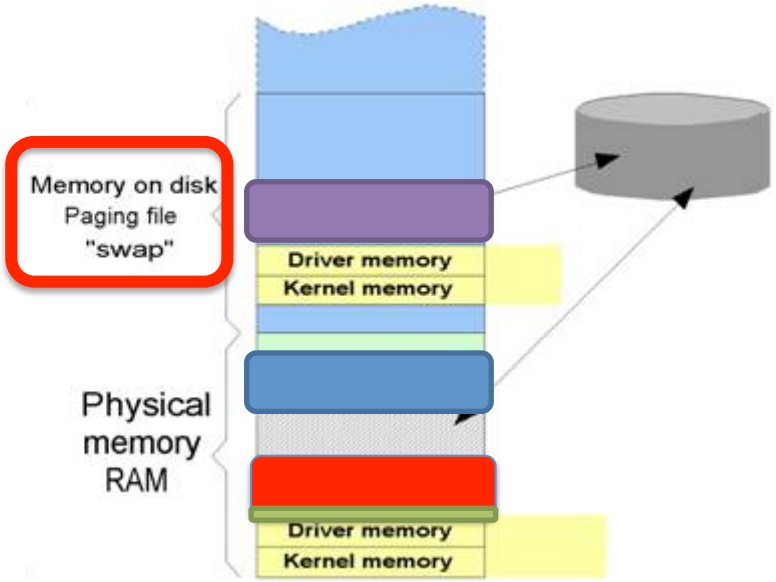
100 ns

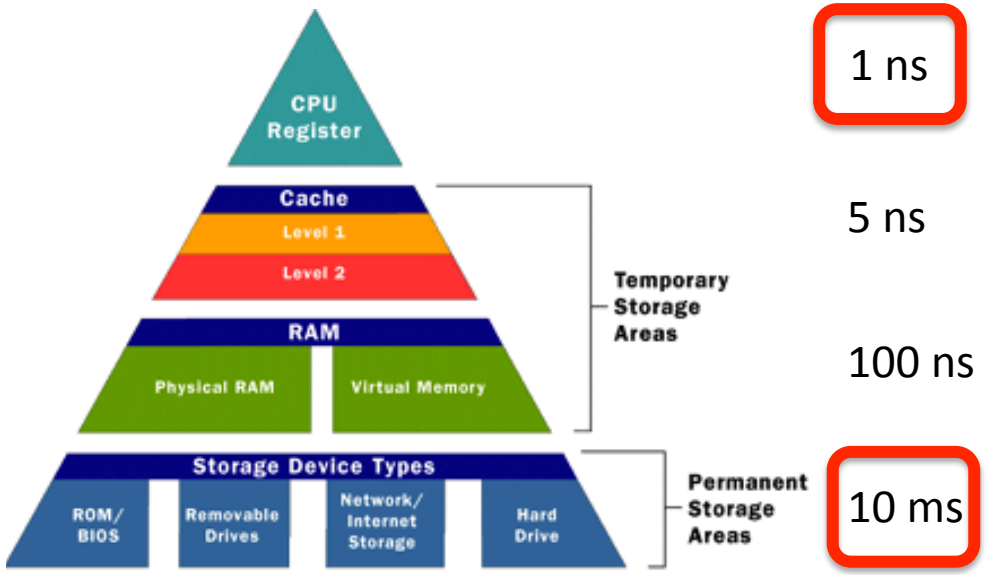
10 ms

When RAM is full, the system will **automatically start swapping**
Exchange data with the hard drive instead of crashing

Two things to be a geek

- Two concepts that good'ol programmers know …
- … That youngsters don't give a s**t about …





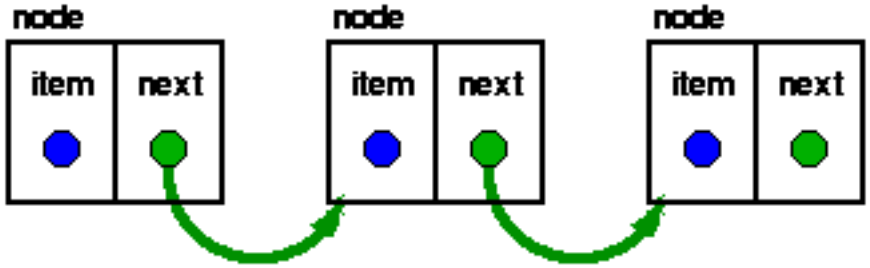
DIFFERENCE OF 10^7 TIMES SLOWER

So remember kids …

- The **biggest bottleneck** in informatics is the memory
  - Being memory handling (how and where to allocate)
  - Or where the memory is accessed
    - CPU cache vs. RAM = * 1.000
    - RAM vs Hard Drive = * 10.000
    - CPU cache vs. Hard Drive = * 10.000.000
  - File I/O is the **slowest operation** possible
- A good and fitting data structure is half the battle
- Algorithmics is not about the most « logical » solution …
- … But the most efficient (number and types of operations)
- **Static structures** will be handy most of the time
- But are insufficient **for advanced processing**
- Need to move towards **dynamic structures !**

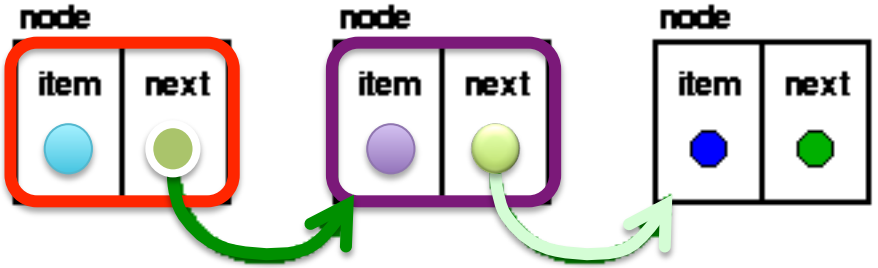
Linked lists

- Linked lists are the **basis of all dynamic structures**
- Each element (*node*) is in fact a structure itself
  - Contains an item (the actual data)
  - And a pointer (or multiple !) to link the next element(s)



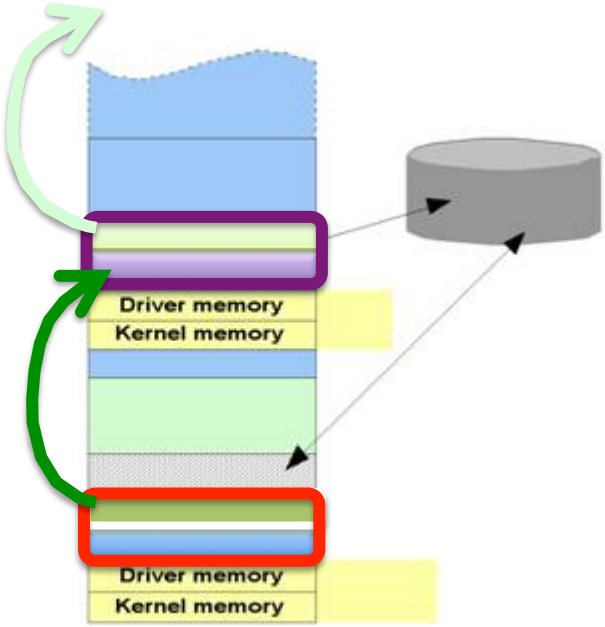
Linked lists

So what about the **memory ?**



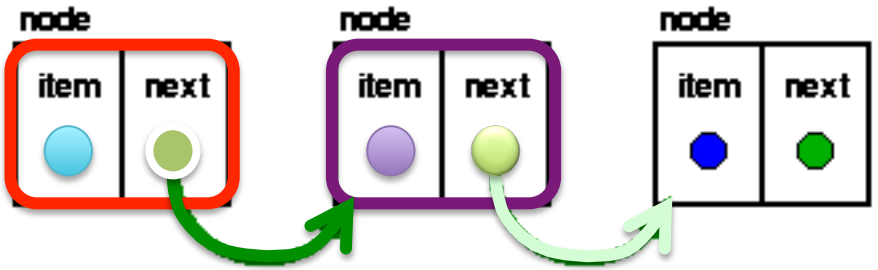
Same for the « next of the next »
Makes structure entirely dynamic

Next pointer is an **adress**
Indicates location of second element



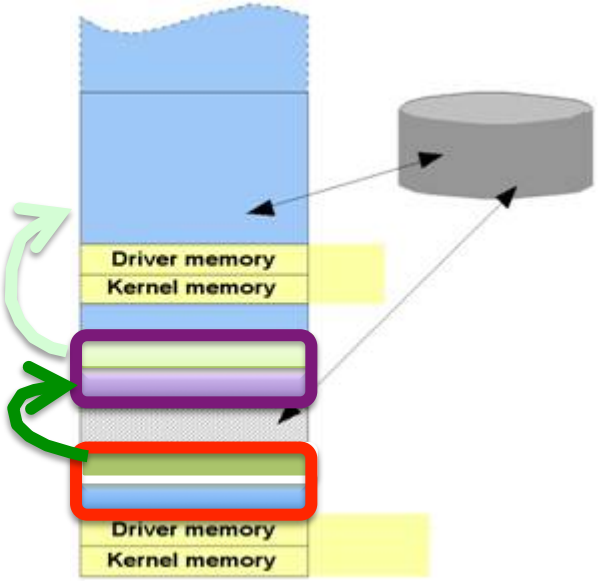
Linked lists

So what about the **memory ?**



WOW ! What about cache locality ?

Don't worry … Allocations incremental in adress space

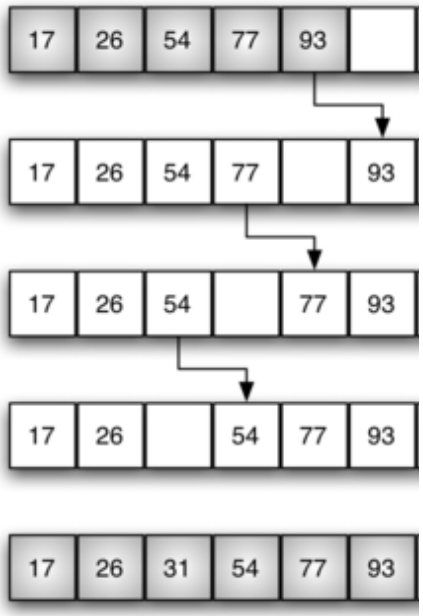


Linked lists

How to insert an element inside a **sorted structure** …

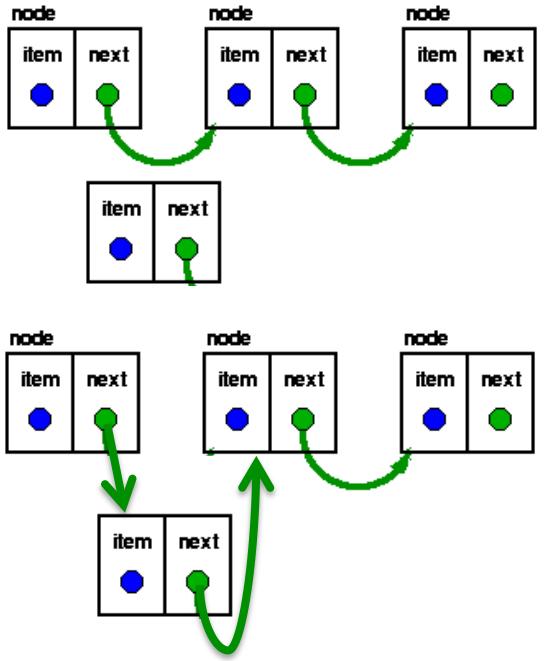
Insertion

Arrays



N operations

Linked list

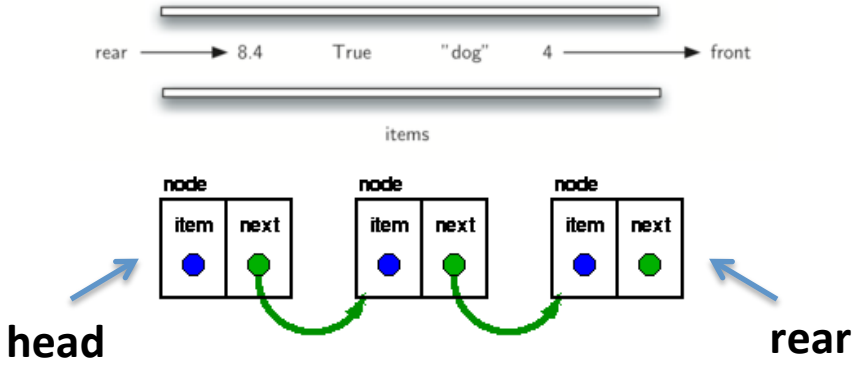


2 operations

Deletion

Queues

- Think about those damn waiting lines …
- Used in computing servers, operating systems, hash tables …
- Can model temporal processes as well (short-term memory)



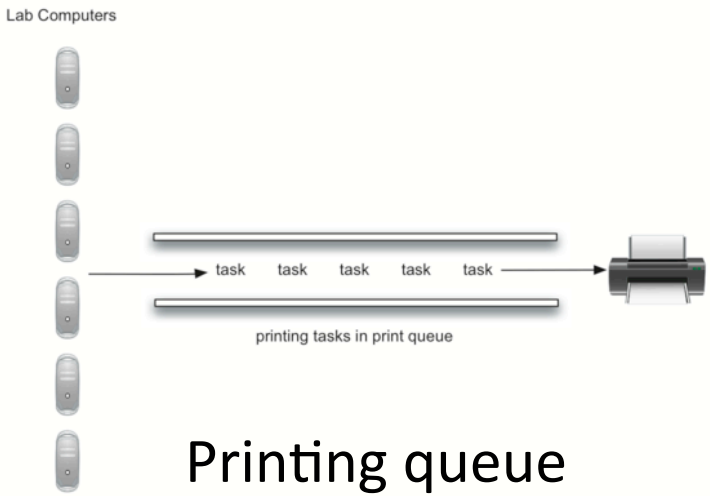
- This structure is called **FIFO (First-In First-Out)**
- We add elements at the end of the queue
- And retrieve element at the top of the queue

Queues

Operation	Contents	Return
q.**isEmpty**()	[]	True
q.**enqueue**(4)	[4]	
q.enqueue('dog')	['dog',4]	
q.enqueue(True)	[True,'dog',4]	
q.**size**()	[True,'dog',4]	3
q.**isEmpty**()	[True,'dog',4]	False
q.enqueue(8.4)	[8.4,True,'dog',4]	
q.**dequeue**()	[8.4,True,'dog']	4
q.dequeue()	[8.4,True]	'dog'
q.size()	[8.4,True]	2

Queues

Classical uses of queues



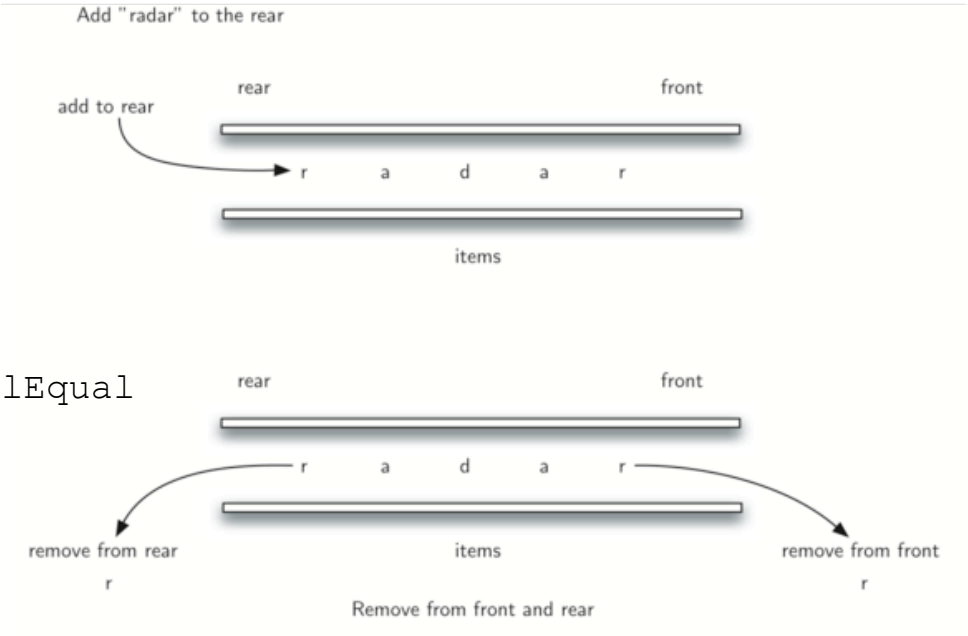
Printing queue

```
printerSpooler(numSecondes, pagePerMinute)
 labprinter = Printer(pagesPerMinute)
 printQueue = Queue()
 waitingtimes = []

 for currentSecond in range(numSeconds)
   if newPrintTask()
      task = Task(currentSecond)
      printQueue.enqueue(task)
   if (not labprinter.busy()) and (not printQueue.isEmpty())
        nexttask = printQueue.dequeue()
        waitingtimes.append(nexttask.waitTime(currentSecond))
        labprinter.startNext(nexttask)
   labprinter.tick()
   averageWait=sum(waitingtimes)/len(waitingtimes)
   print("Average Wait %6.2f secs %3d tasks remaining."%
(averageWait,printQueue.size()))
```

Queues

Classical uses of queues

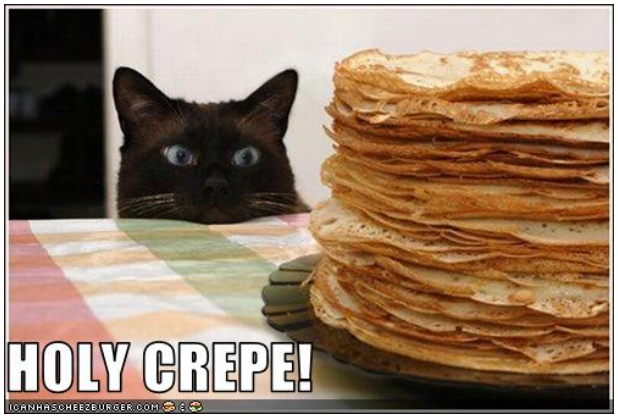
```
def palchecker(aString):
    chardeque = Deque()

    for ch in aString:
        chardeque.addRear(ch)

    stillEqual = True

    while chardeque.size() > 1 and stillEqual
        first = chardeque.removeFront()
        last = chardeque.removeRear()
        if first != last:
            stillEqual = False

    return stillEqual
```

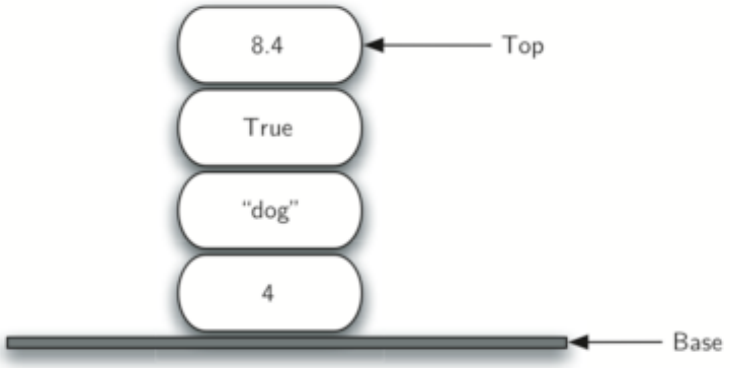


Palindrome checker

Stacks

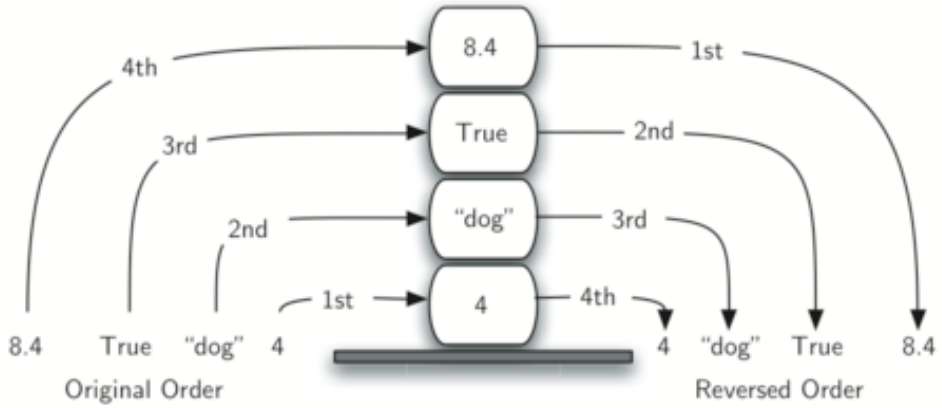
- Think about a stack of plates (or crepes)
- Used in transformational model, and most traversal algorithms
- Is also heavily used in the execution processes (heap, calls)





- This structure is called **LIFO (Last-In First-Out)**
- We add elements at the top of the stack
- … And also retrieve element at the top of the stack

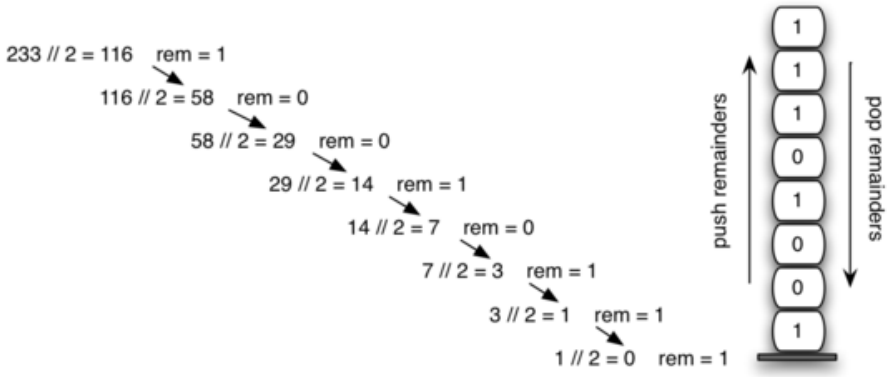
Stacks



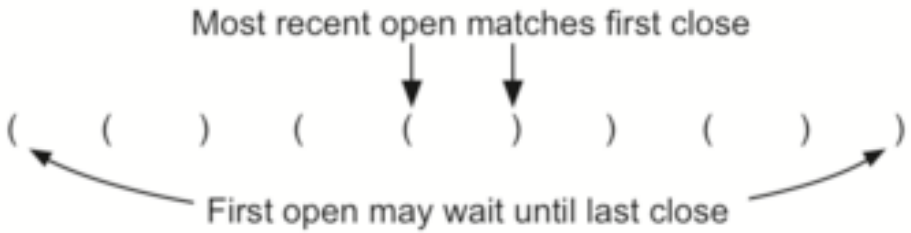
Stack Operation	**Stack Contents**	**Return Value**
s.**isEmpty**()	[]	True
s.**push**(4)	[4]	
s.push('dog')	[4,'dog']	
s.**peek**()	[4,'dog']	'dog'
s.push(True)	[4,'dog',True]	
s.**size**()	[4,'dog',True]	3
s.isEmpty()	[4,'dog',True]	False
s.push(8.4)	[4,'dog',True,8.4]	
s.**pop**()	[4,'dog',True]	8.4
s.pop()	[4,'dog']	True

Stacks

Conversion to binary

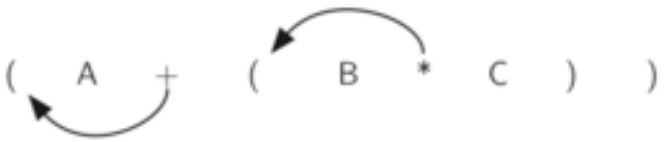


Symbol balancing



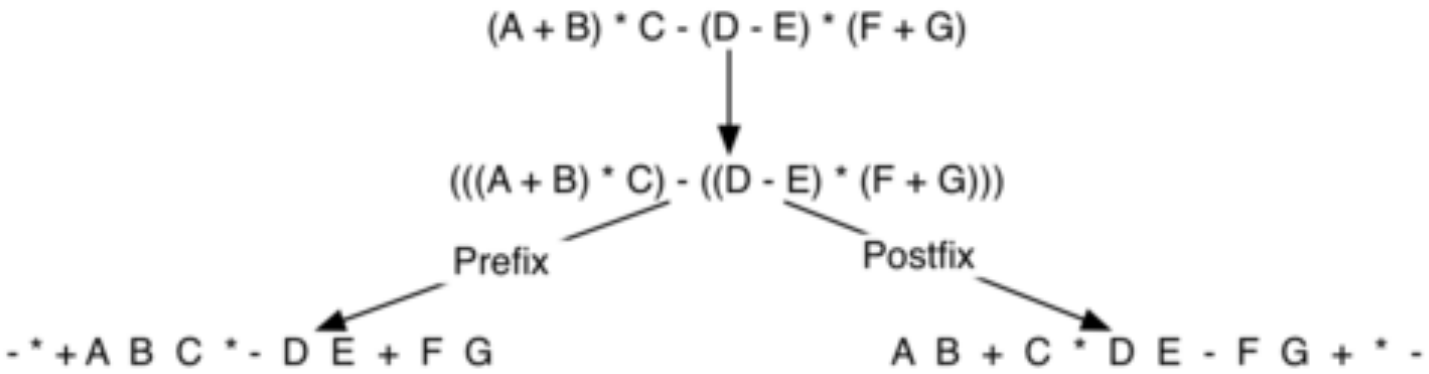
Stacks

Prefix notation



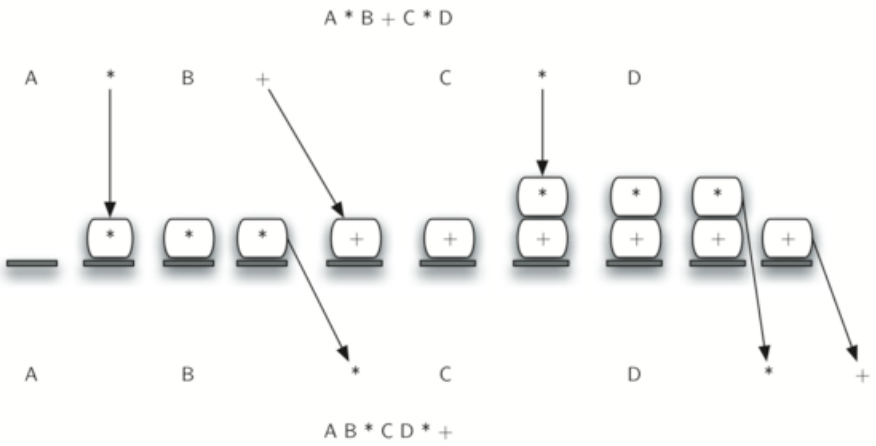
Postfix notation

Generic transformation example



- How to do this with a stack ?
- Find a musical application to it ?

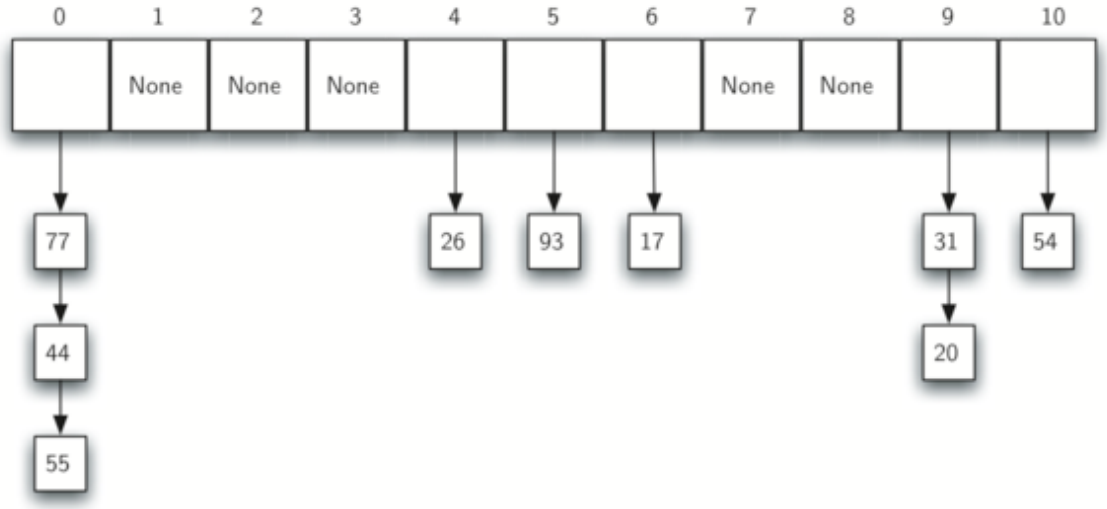
Stacks




```
for token in tokenList:
     if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ”
          postfixList.append(token)
     elif token == '(':
          opStack.push(token)
     elif token == ')':
          topToken = opStack.pop()
          while topToken != '(':
             postfixList.append(topToken)
             topToken = opStack.pop()
     else
          while (not opStack.isEmpty()) and \
               (prec[opStack.peek()] >= prec[token]):
                  postfixList.append(opStack.pop())
          opStack.push(token)
```

Hashing tables

- Think about the contacts in your phone
  - You have a complex data you want to find
  - You use a naming system to access it
- Allows to perform **search in O(1) time !**
- Is heavily used … well … everywhere ☺
- Your first mix between static and dynamic structures



Hashing functions

- All the magic of an hashing table lies in the *hashing function*
- Let's go back to our phone contacts

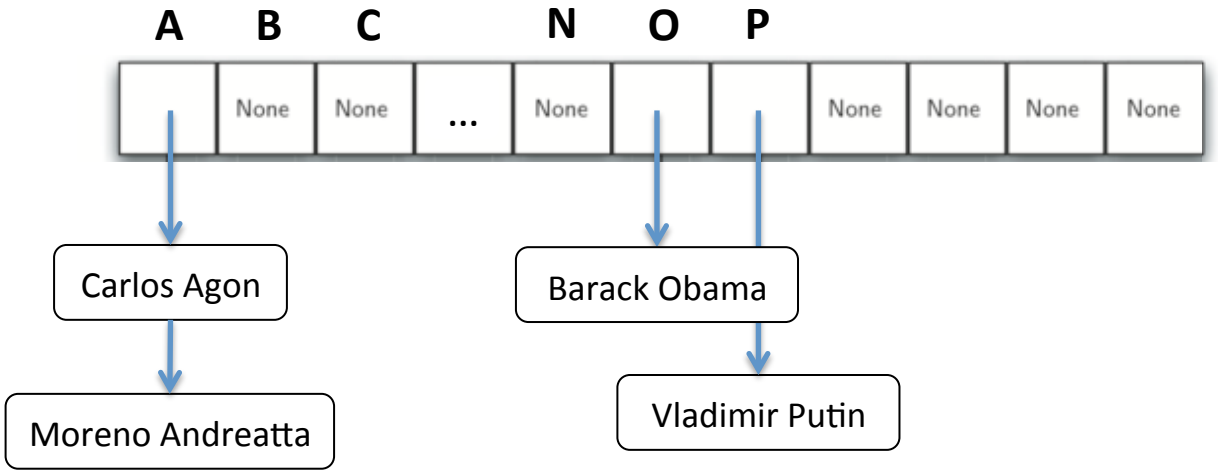
Agon Carlos => A

Obama Barack => O

Putin Vladimir => P

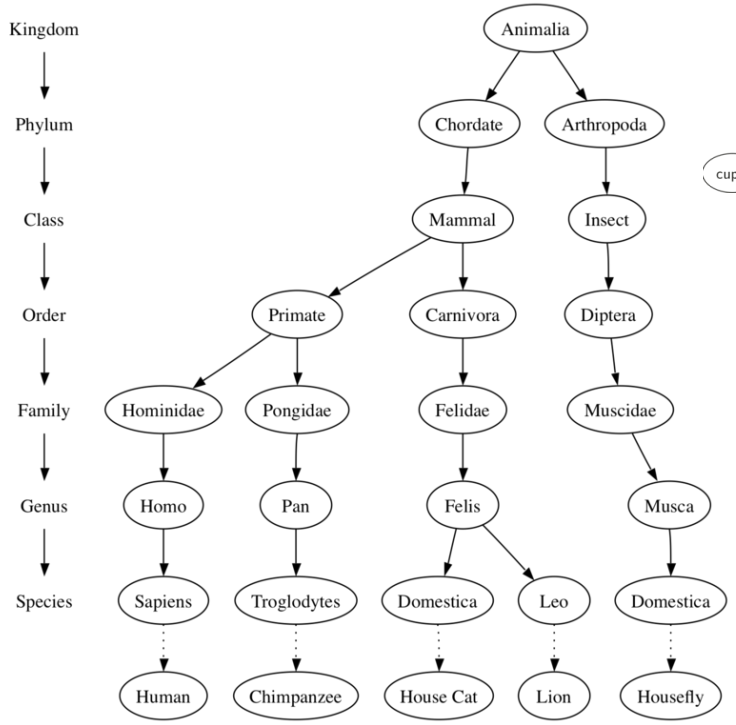
- Here our hashing function is simple :

H(e) = e.[0] – 'A'

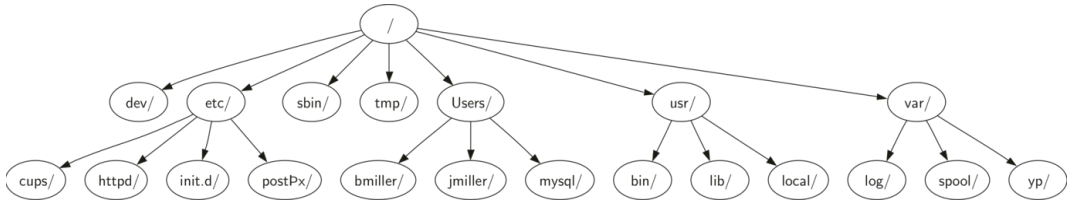


Trees

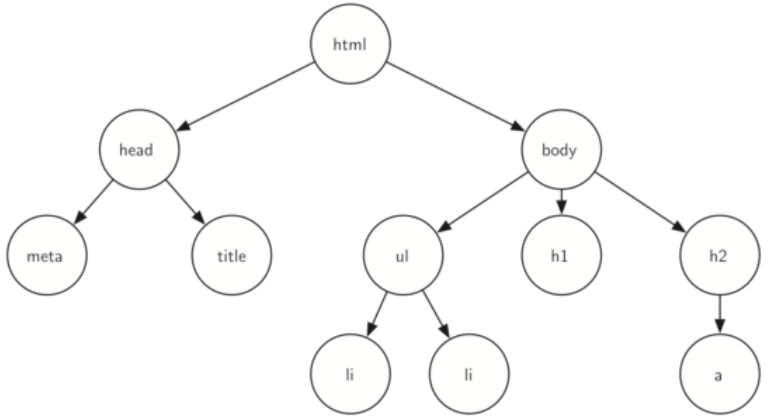
- Tree structures are found **absolutely everywhere**
- One of the **most proficient** structure to know in informatics



The tree of life



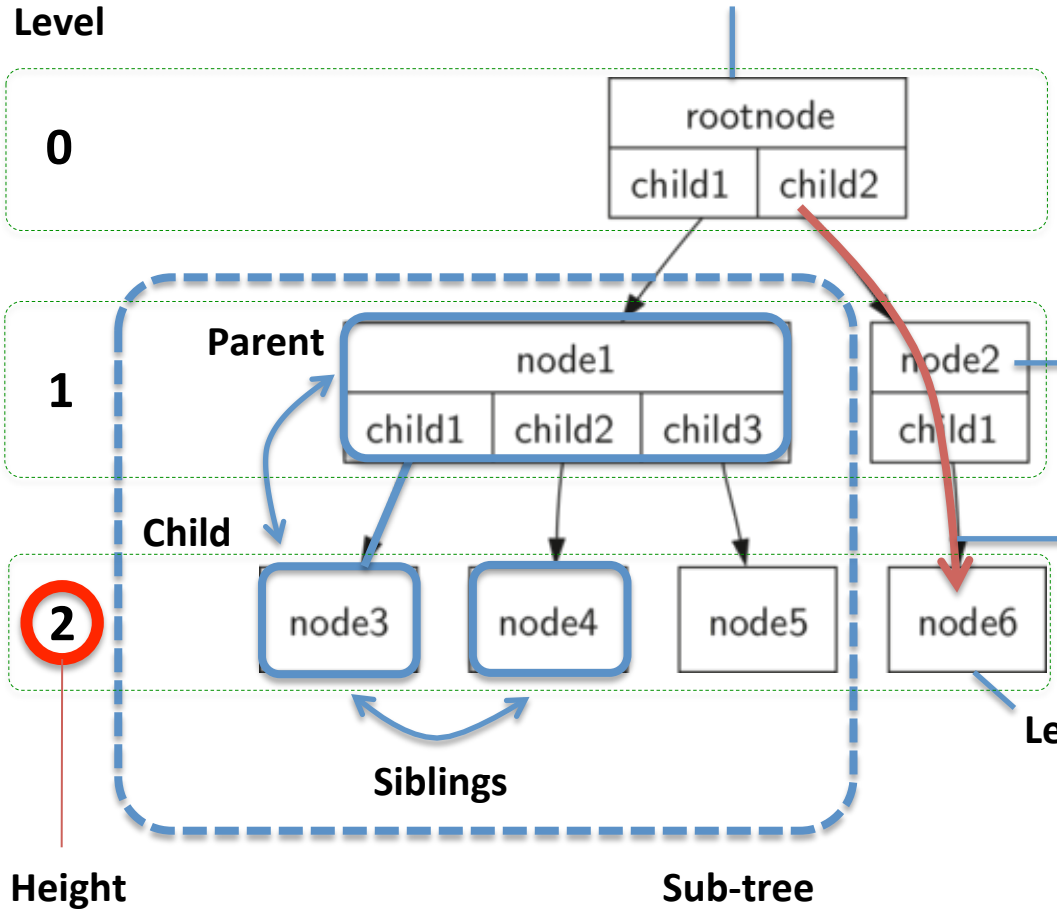
Computer drive directories



HTML document

Trees: Vocabulary

Root: « starting point » = the only node with no incoming edges



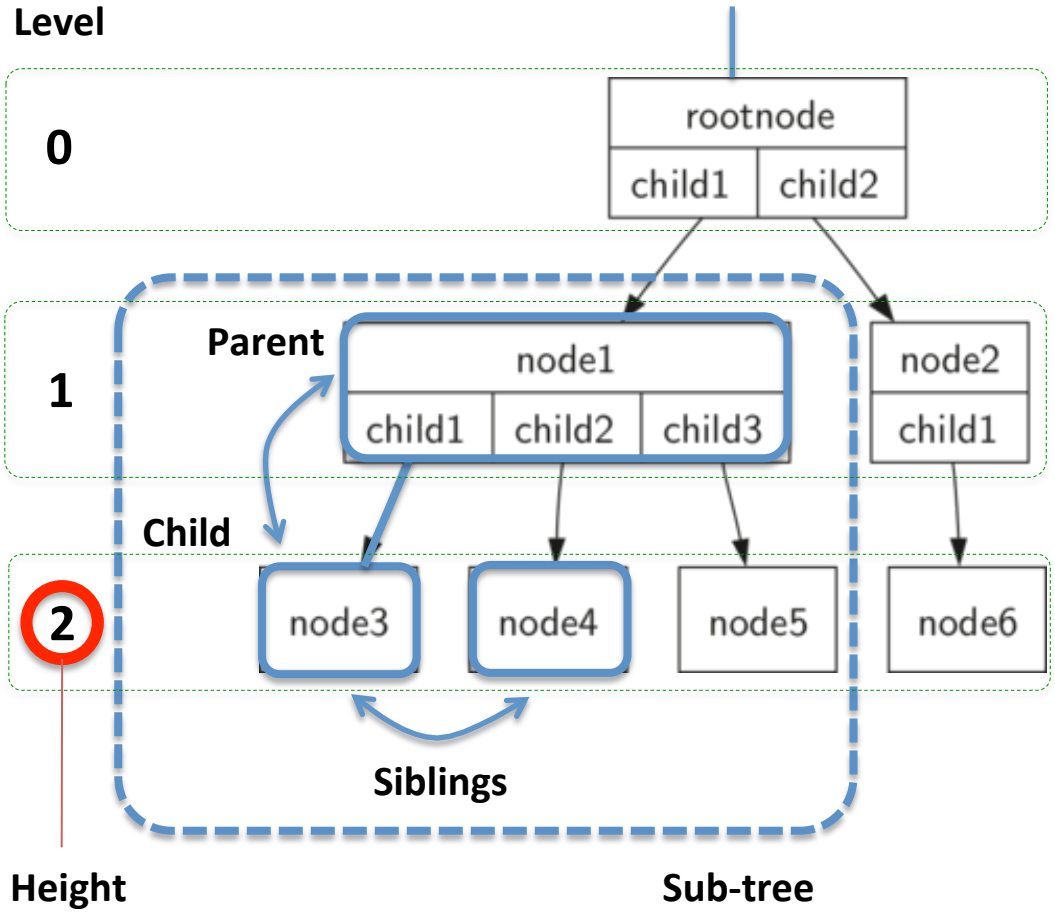
Node: fundamental element in a tree
here *internal* : has children and is not root)

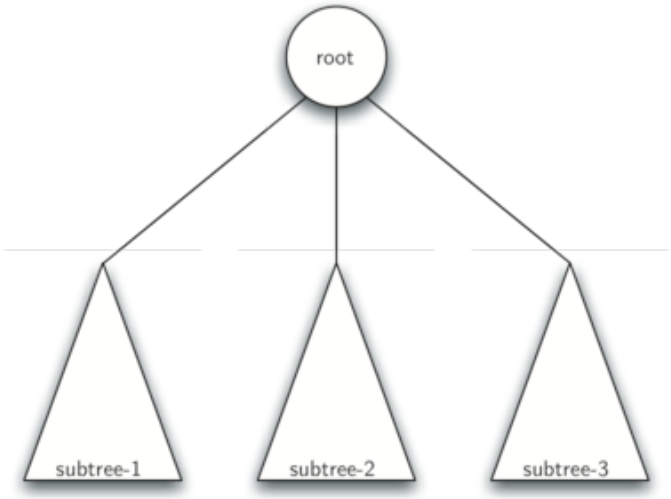
Path: an ordered list of nodes
Here : root -> node2 -> node6

Leaf: node without children

Trees: Vocabulary

Root: « starting point » = the only node with no incoming edges

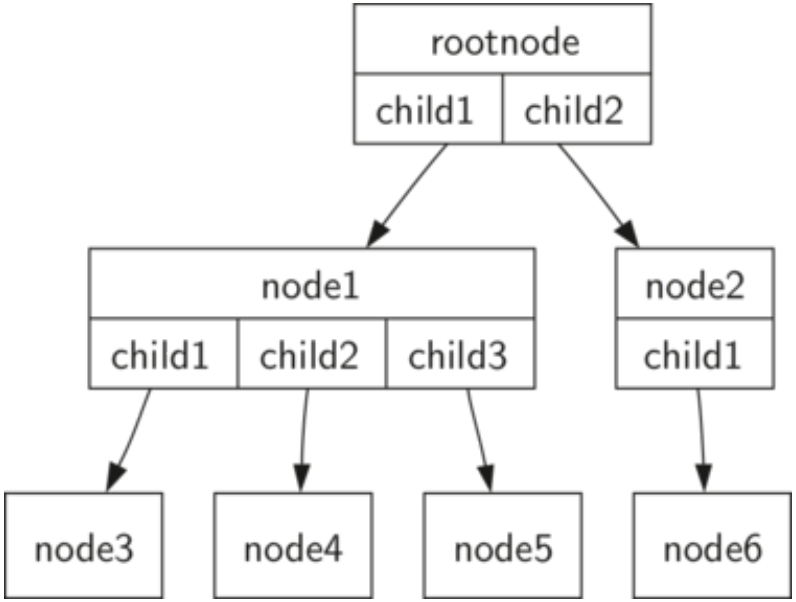




Recursive definition of a tree :
A node and its subtrees.

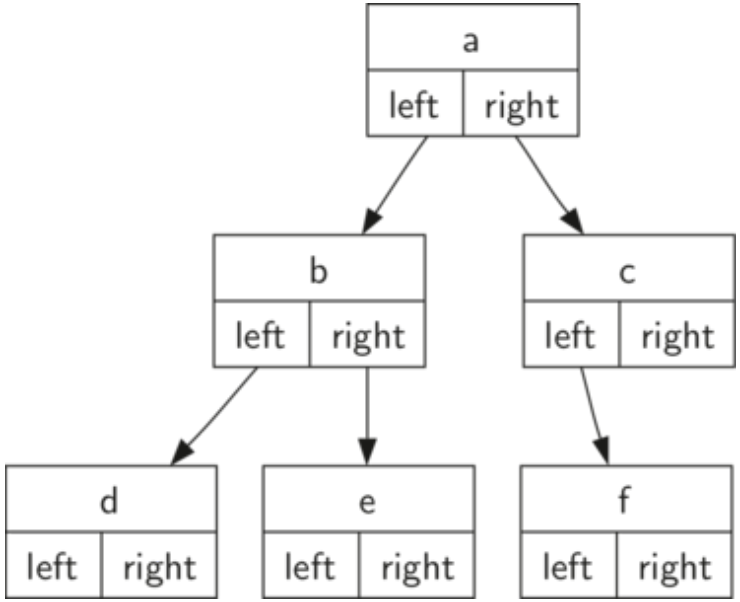
Trees: Vocabulary

- Distinction based on the *arity* of a tree
- Solely depends on the allowed number of children



N-ary tree

(any number of children for each node)

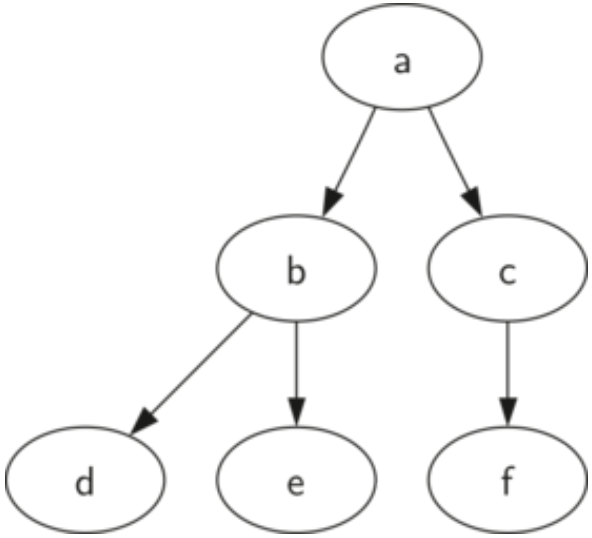


Binary tree

(Maximum 2 children for each node *left* and *right*)

Trees: Representation

- We can define trees as a « linear structure »
- Relies on a *list of lists* representation

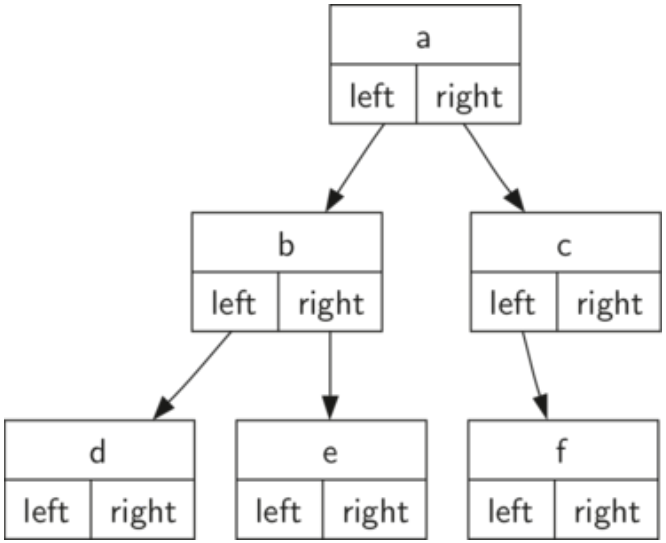


myTree = ['a', #root
['b', #left subtree
['d' [], []], ['e' [], []]],
['c', #right subtree
['f' [], []], []]]

```
def BinaryTree(r):
    return [r, [], []]

def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root

def insertRight(root,newBranch):
    t = root.pop(2)
    if len(t) > 1:
        root.insert(2,[newBranch,[],t])
    else:
        root.insert(2,[newBranch,[],[]])
    return root

def getRootVal(root):
    return root[0]
def setRootVal(root,newVal):
    root[0] = newVal
def getLeftChild(root):
    return root[1]
def getRightChild(root):
    return root[2]
```

Trees: Representation

- Best **dynamic** representation relies on pointers
- Each node is a **key** (data), and a set of **child** (pointers)

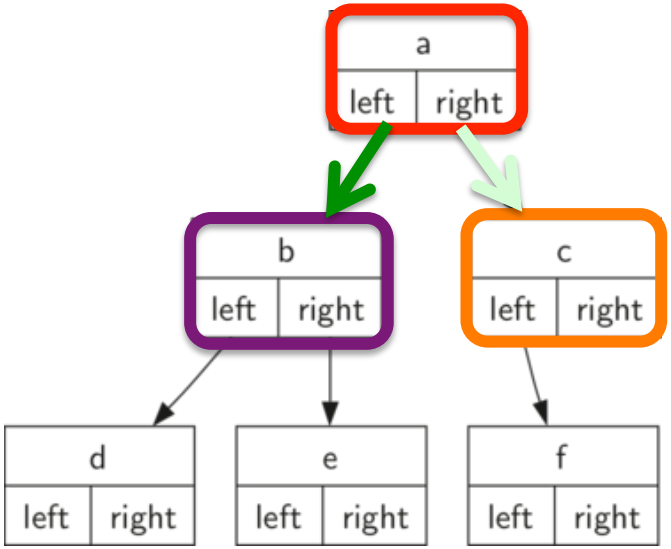


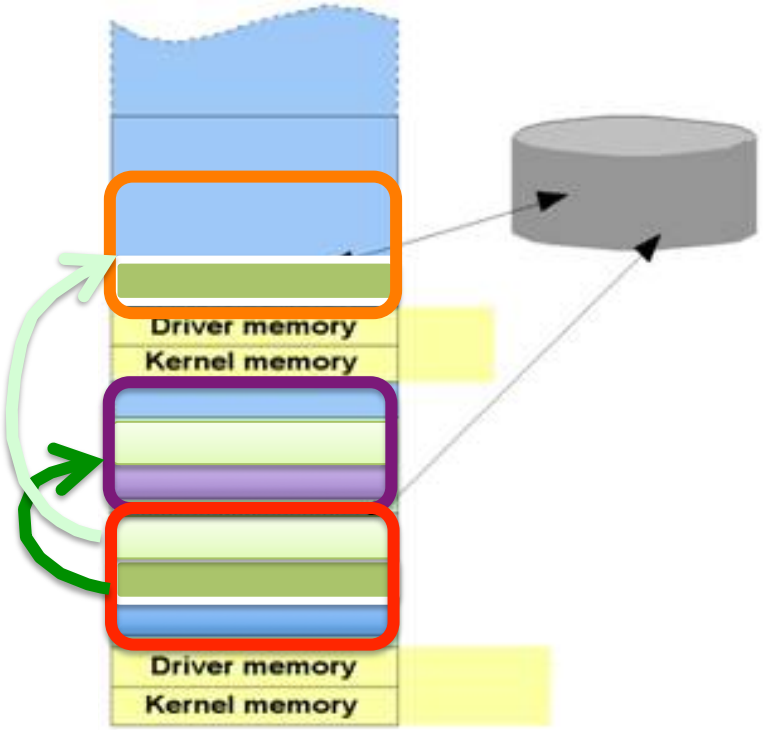

```
class BinaryTree:
   def __init__(self,rootObj):
      self.key = rootObj
      self.leftChild = None
      self.rightChild = None
```

```
def insertLeft(self,newNode):
  if self.leftChild == None:
      self.leftChild = BinaryTree(newNode)
  else:
      t = BinaryTree(newNode)
      t.leftChild = self.leftChild
      self.leftChild = t

def insertRight(self,newNode):
  if self.rightChild == None:
        self.rightChild = BinaryTree(newNode)
      else:
        t = BinaryTree(newNode)
        t.rightChild = self.rightChild
        self.rightChild = t

def getRightChild(self):
      return self.rightChild
def getLeftChild(self):
      return self.leftChild
def setRootVal(self,obj):
      self.key = obj
def getRootVal(self):
      return self.key
```

Trees: Representation

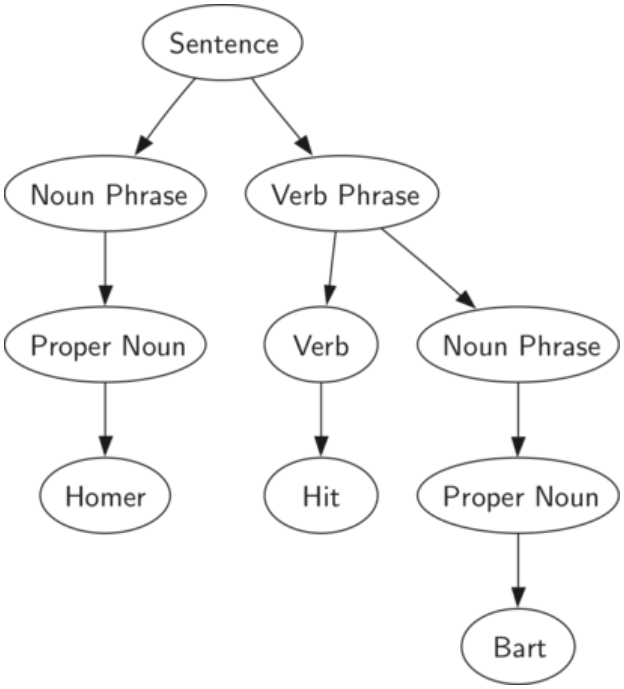
- What about the **memory ?**



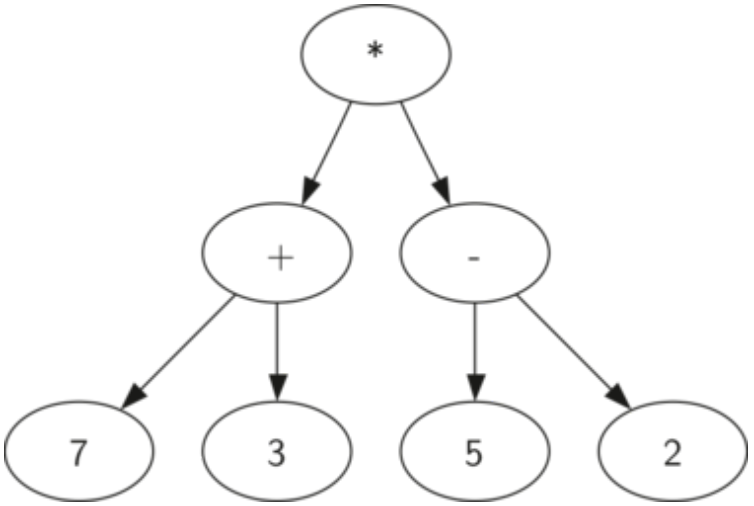


Trees: Parsing

- Trees are extremely efficient structures for parsing (cf. 1st slide)
- Can represent any constructions such as sentences or mathematical expressions



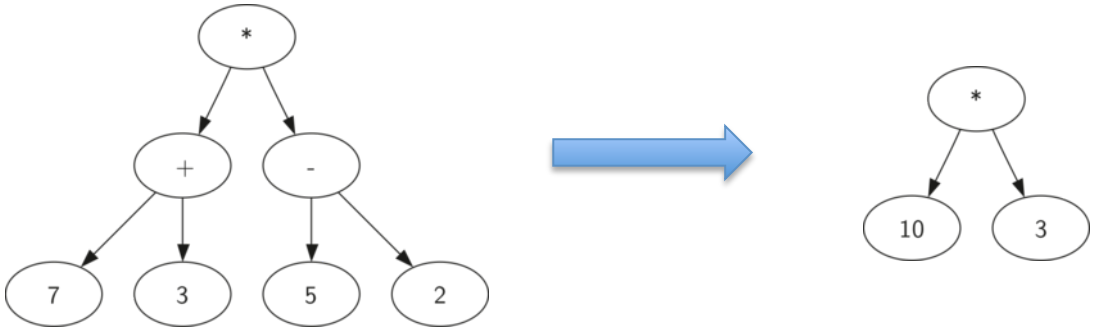
Parse tree for a simple sentence



Parse tree for ((7 + 3) * (5 – 2))

Trees: Parsing

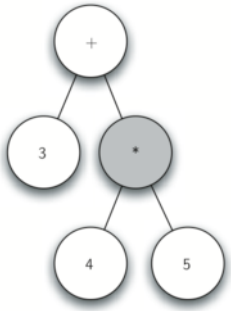
- Let's look into (and construct) the mathematical expression parsing



Original parse tree

Simplified (partly **evaluated**) tree

Let's construct a tree from the list **['(', '3', '+', '(', '4', '*', '5' ,')',')'].**

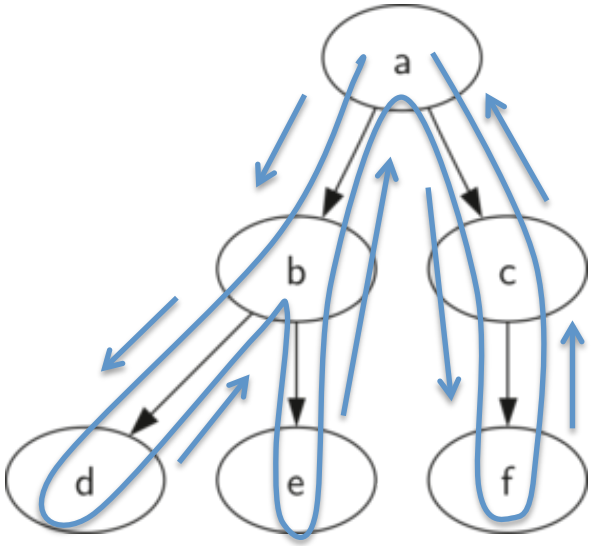


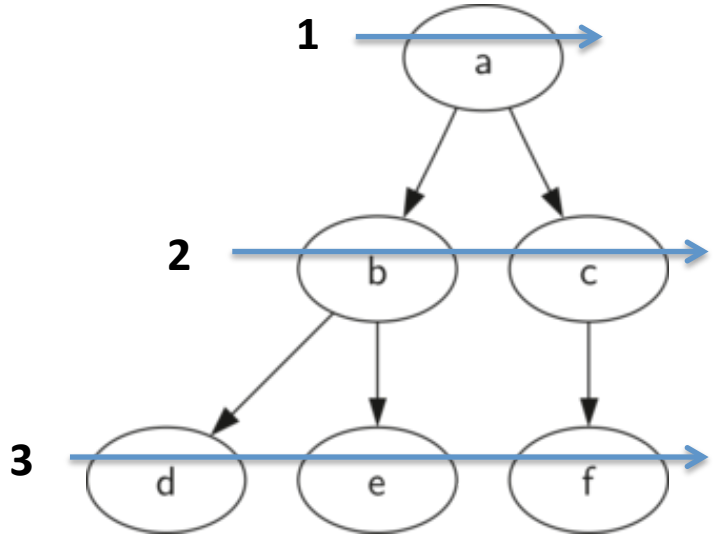
Trees: Parsing

```
def buildParseTree(fpexp):
    fplist = fpexp.split()
    pStack = Stack()
    eTree = BinaryTree('')
    pStack.push(eTree)
    currentTree = eTree
    for i in fplist:
        if i == '(':
            currentTree.insertLeft('')
            pStack.push(currentTree)
            currentTree = currentTree.getLeftChild()
        elif i not in ['+', '-', '*', '/', ')']:
            currentTree.setRootVal(int(i))
            parent = pStack.pop()
            currentTree = parent
        elif i in ['+', '-', '*', '/']:
            currentTree.setRootVal(i)
            currentTree.insertRight('')
            pStack.push(currentTree)
            currentTree = currentTree.getRightChild()
        elif i == ')':
            currentTree = pStack.pop()
        else:
            raise ValueError
    return eTree
```

Trees: Traversal

- Now the most important property is how to go through the trees
- Two families of tree traversals





Depth traversal

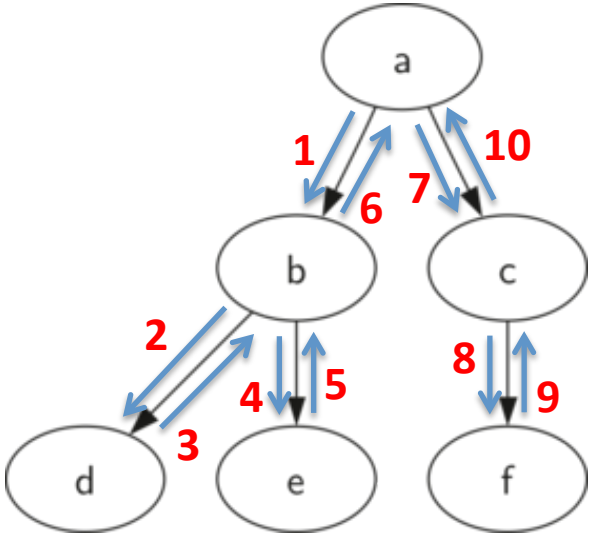
Here multiple possibilities
(cf. next slides)

Width traversal

[a, b, c, d, e, f]
[a, c, b, f, e, d]

Trees: Depth traversal

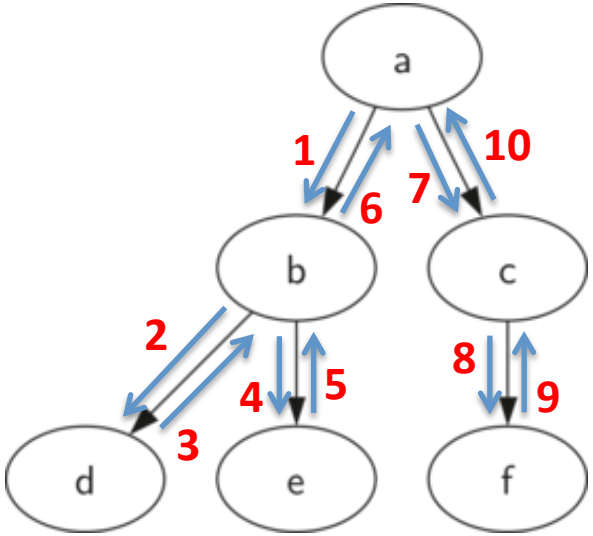
- Also two types of depth traversal



Left-hand depth traversal

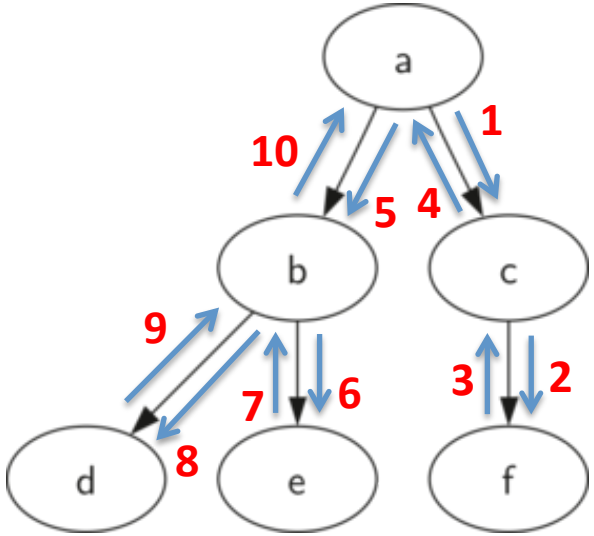
Trees: Depth traversal

- Also two types of depth traversal



Left-hand depth traversal

```
def leftDepthTraversal(tree):
  if tree.isEmpty()
    return;
  // Before going down on left
  leftDepthTraversal(tree.leftChild)
  // Coming back from left, going right
  leftDepthTraversal(tree.rightChild)
  // Coming back from right
```

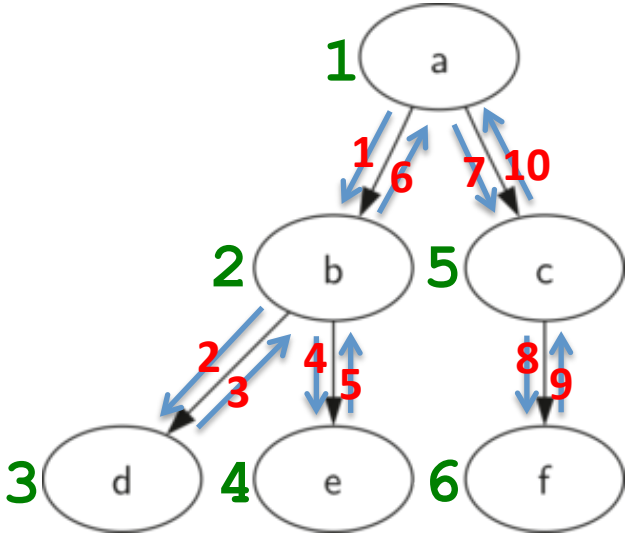


Right-hand depth traversal

```
def rightDepthTraversal(tree):
  if tree.isEmpty()
    return;
  // Before going down on right
  rightDepthTraversal(tree.rightChild)
  // Coming back from right, going left
  rightDepthTraversal(tree.leftChild)
  // Coming back from left
```

Trees: Left-hand depth traversal

- By convention the **left-hand** depth traversal is generally used



Left-hand depth traversal

```
def leftDepthTraversal(tree):
  if tree.isEmpty()
    return;
  // Before going down on left
  leftDepthTraversal(tree.leftChild)
  // Coming back from left, going right
  leftDepthTraversal(tree.rightChild)
  // Coming back from right
```

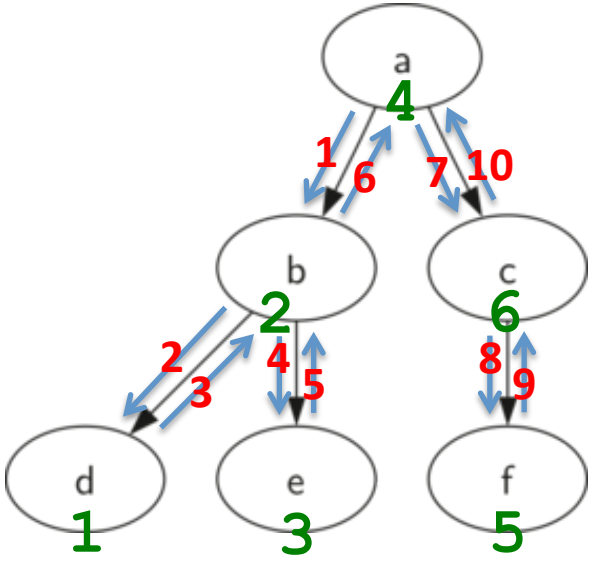
Three **processing orders** can be used :
Preorder (prefix): The processing is applied to the data before going down on any subtree

```
def leftDepthTraversal(tree):
  if tree.isEmpty()
    return;
  processData(tree.key);
  leftDepthTraversal(tree.leftChild);
  leftDepthTraversal(tree.rightChild);
```

a,b,d,e,c,f

Trees: Left-hand depth traversal

- By convention the **left-hand** depth traversal is generally used



Left-hand depth traversal

```
def leftDepthTraversal(tree):
  if tree.isEmpty()
    return;
  // Before going down on left
  leftDepthTraversal(tree.leftChild)
  // Coming back from left, going right
  leftDepthTraversal(tree.rightChild)
  // Coming back from right
```

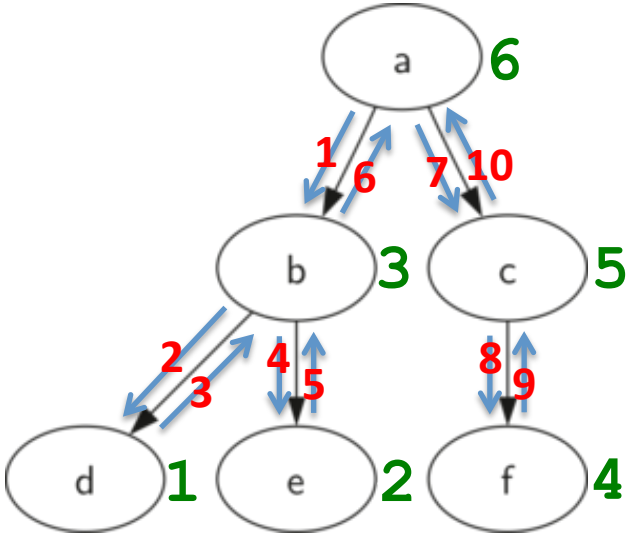
Three **processing orders** can be used :
Preorder (prefix): The processing is applied to the data before going down on any subtree
Inorder (infix): The processing is applied to the data between the two recursions

```
def leftDepthTraversal(tree):
  if tree.isEmpty()
    return;
  leftDepthTraversal(tree.leftChild);
  processData(tree.key);
  leftDepthTraversal(tree.rightChild);
```

d,b,e,a,f,c

Trees: Left-hand depth traversal

- By convention the **left-hand** depth traversal is generally used



Left-hand depth traversal

```
def leftDepthTraversal(tree):
  if tree.isEmpty()
    return;
  // Before going down on left
  leftDepthTraversal(tree.leftChild)
  // Coming back from left, going right
  leftDepthTraversal(tree.rightChild)
  // Coming back from right
```

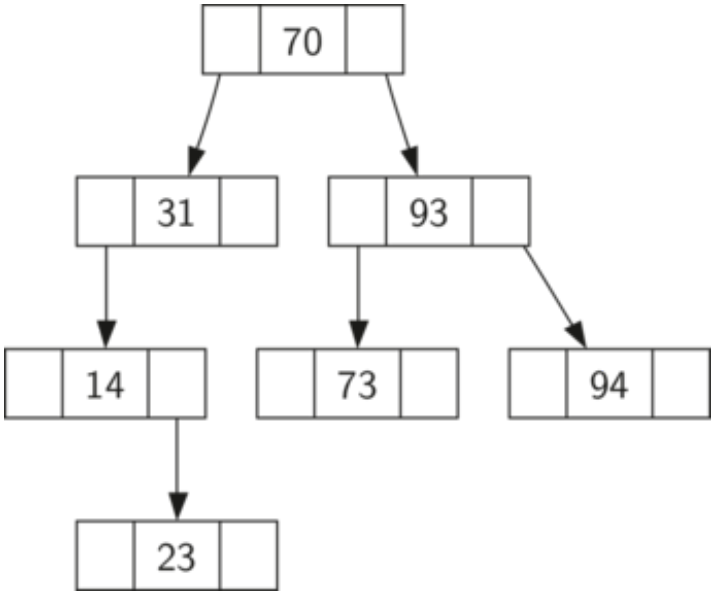
Three **processing orders** can be used :
Preorder (prefix): The processing is applied to the data before going down on any subtree
Inorder (infix): The processing is applied to the data between the two recursions
Postorder (postfix): The processing is applied to the data when coming back from all subtrees

```
def leftDepthTraversal(tree):
  if tree.isEmpty()
    return;
  leftDepthTraversal(tree.leftChild);
  leftDepthTraversal(tree.rightChild);
  processData(tree.key);
```

d,e,b,f,c,a

Binary search trees

- One of the most used indexing technique to date
- Used in B-Trees (SQL), BSP (3D Games Models), Google maps etc…



Given the key (data) **k** of a node and its subtrees
All elements in the left subtree lt are inferior

$$\forall k_l \in lt, k_l < k$$

All elements in the right subtree rt are superior

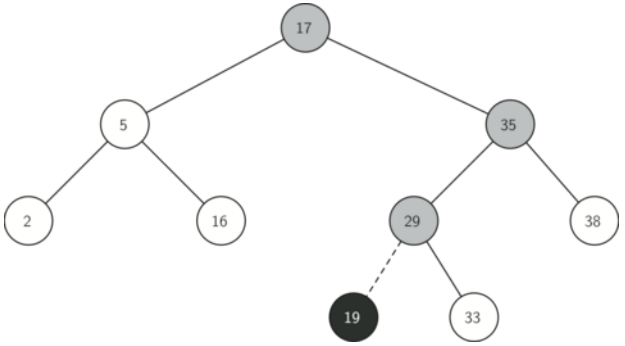
$$\forall k_r \in lt, k_r > k$$

A bit hard to construct and maintain
(insert, delete, balance)

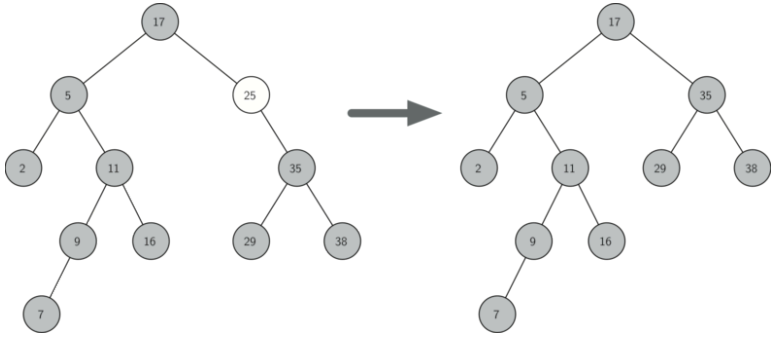
- Extremely simple to efficiently implement a wide array of queries types
- *Find, min, max, range, sort, balance, …*
- ***Quick Quizz:*** How to do these types of queries ?

Binary search trees

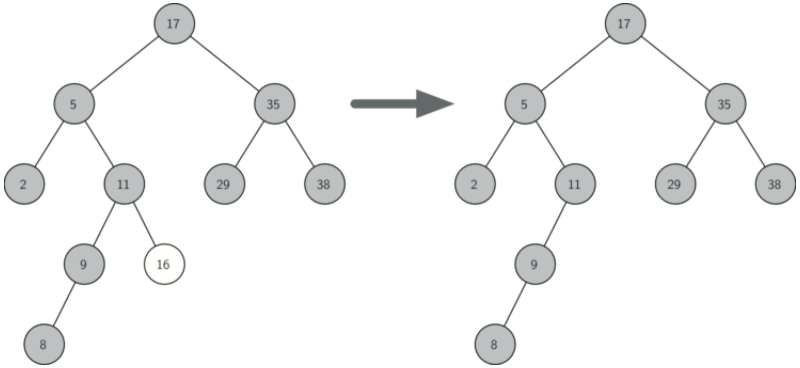
Insert a node



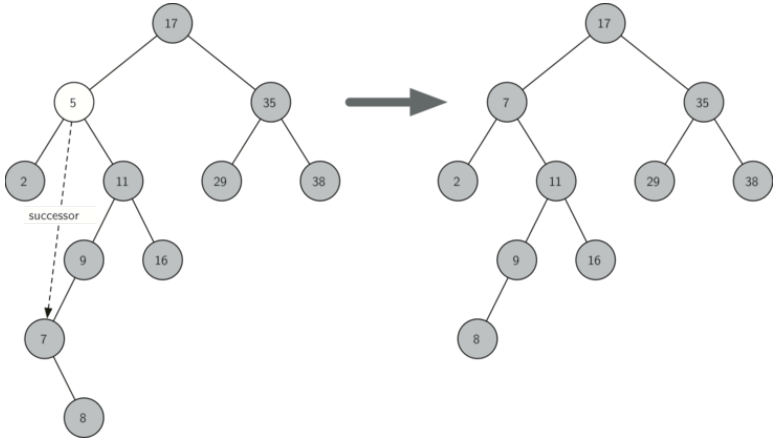
Delete with one children



Delete with no children

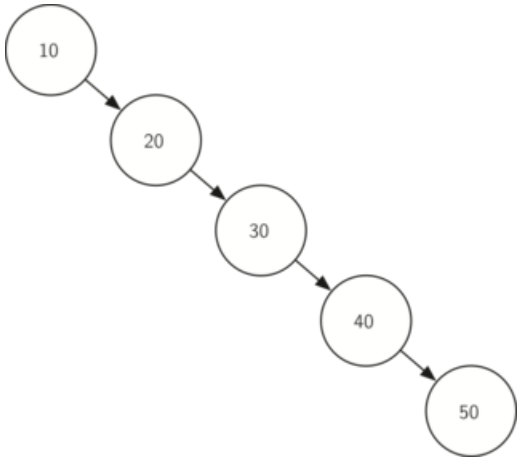


Delete with two children

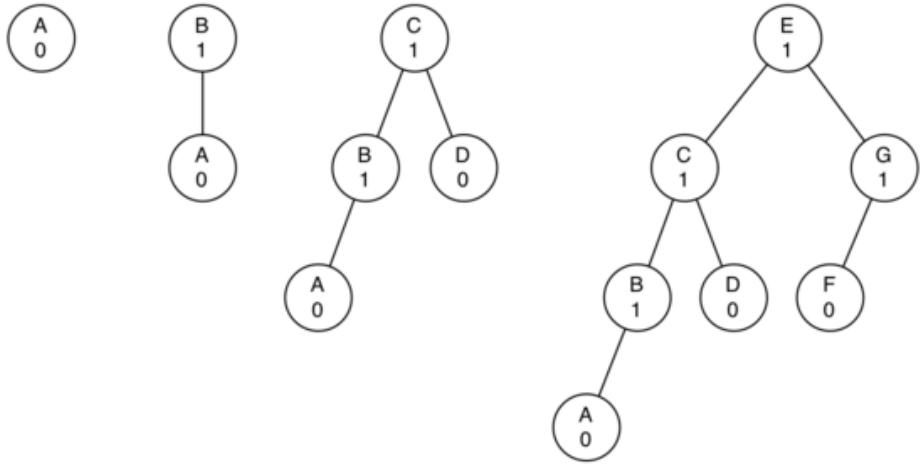


Binary search trees

- Binary search trees are normally very efficient …
- … But can have poor performances if the structure is skewed

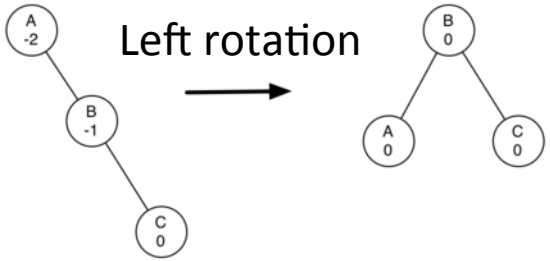


Highly skewed tree

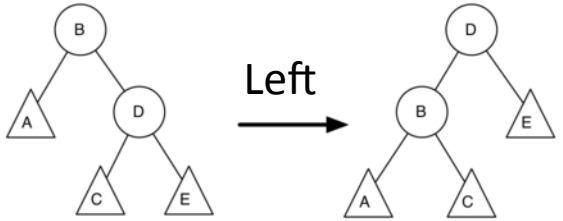


A way to solve the problem (here alphabetical)
Is to use a form of *balancing*
This is done by rotation

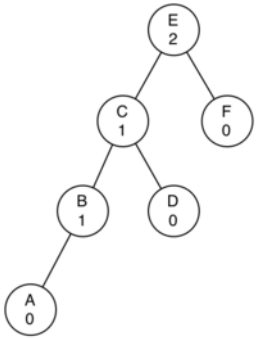
Binary search trees: rotations



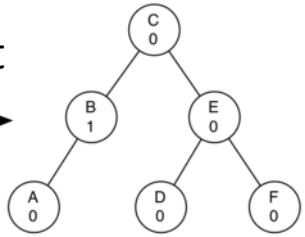
Left rotation

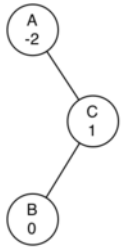


Left

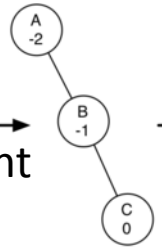


Right

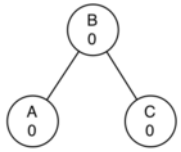




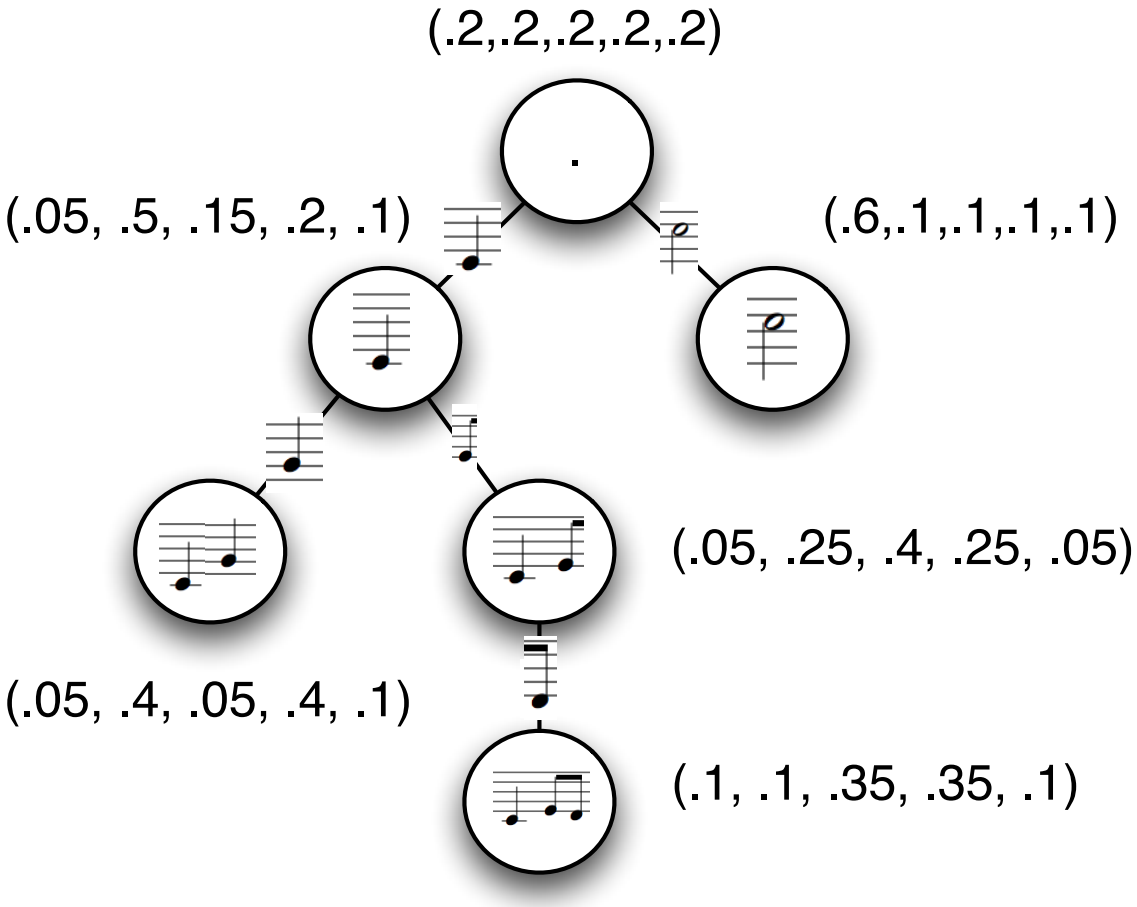
Right



Left

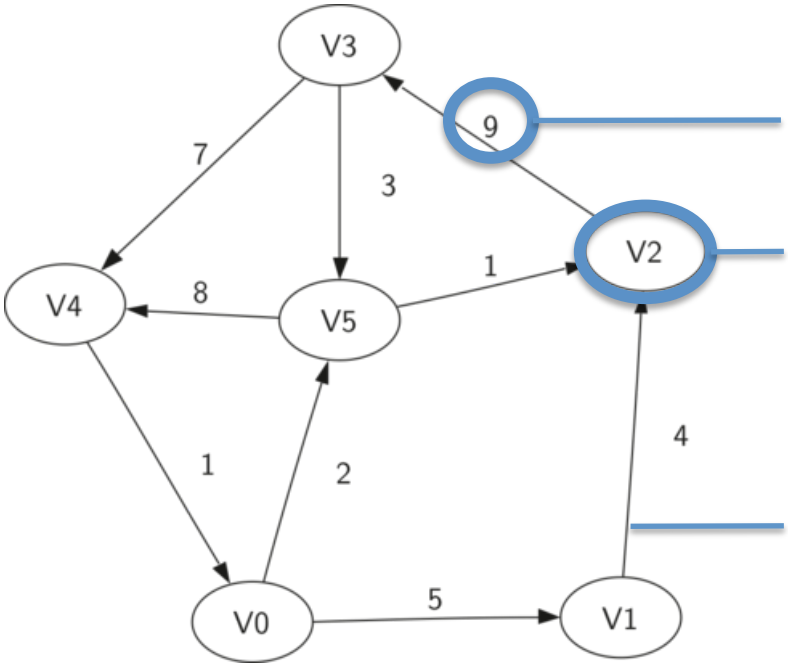


Musical example: style copying



Graphs

- Graph structures are the **generalization of the tree**
- Quick question : what makes a graph specified as a tree ?
- **The most proficient** structure to know in informatics



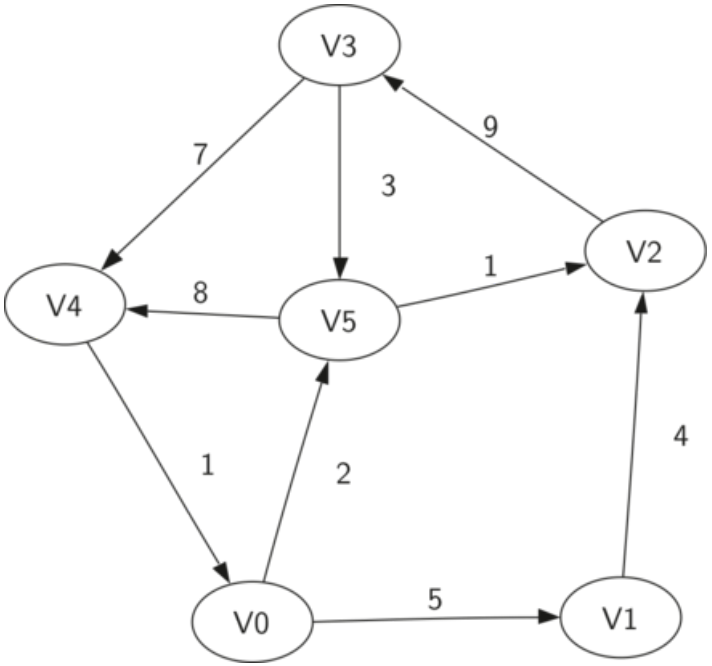
Weight (not always present)
Indicates a cost for traveling between nodes

Vertex (or *node*): fundamental element
Contains the *key* (data)

Edge (or *arc*): links node together
Can be **directed** or **non-directed** (both ways)

Graphs

- Formalization of the graphs



A graph **G** can be formalized as **G = (V,E)**

- **V** a set of vertices
- **E** a set of edges

Each edge is a tuple $e = (v, w)$ with $w, v \in V$

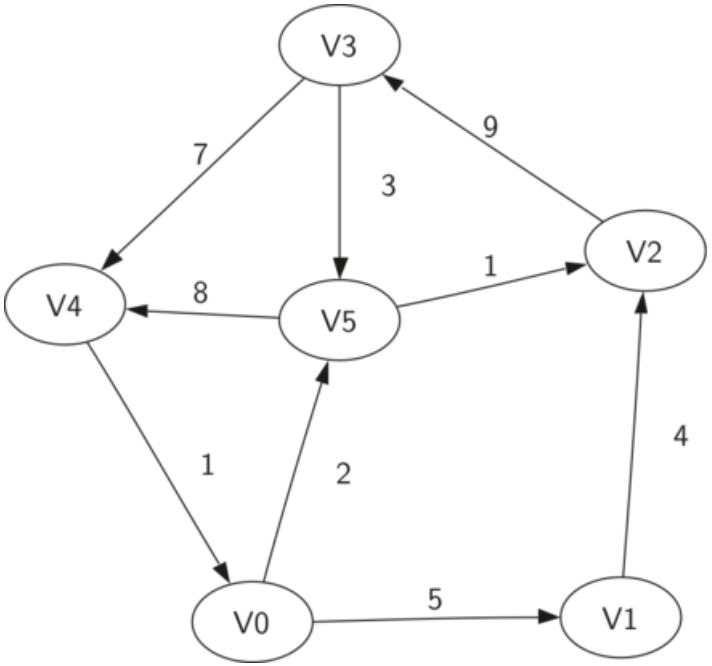
A subgraph **s = (v,e)** of **G** is a set of edges $e \subset E$ and a set of vertices $v \subset V$

A **path** in a graph is a sequence of vertices $w_1, w_2, ..., w_n$ that are connected by edges, ie. $(w_i, w_{i+1}) \in E$

A **cycle** in a graph is a path that starts and ends at the same vertex. A graph with no cycles is called an **acyclic graph**.

Graphs

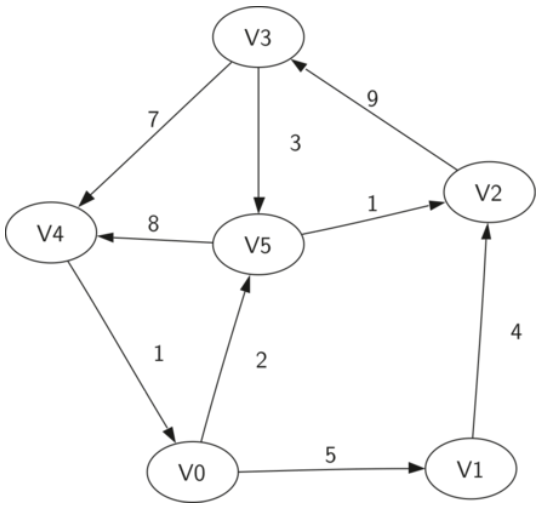
- Operations on graphs



Graph()
addVertex(vert)
addEdge(fromVert, toVert)
addEdge(fromVert, toVert, weight)
getVertex(vertKey)
getVertices()
getEdges()

Graphs: Representation

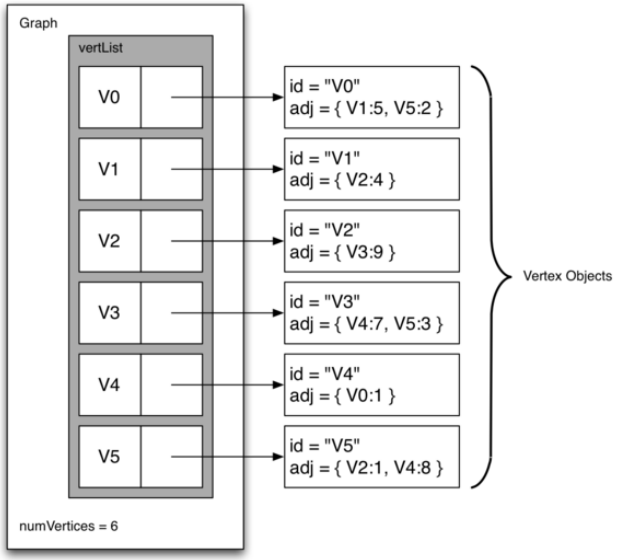
- Similarly to the trees, we can use various ways of representations
- Either using a **static** (adjacency matrix) or **dynamic** (adjacency list) structure



Corresponding graph

	V0	V1	V2	V3	V4	V5
V0		5				2
V1			4			
V2				9		
V3					7	3
V4	1					
V5			1		8	

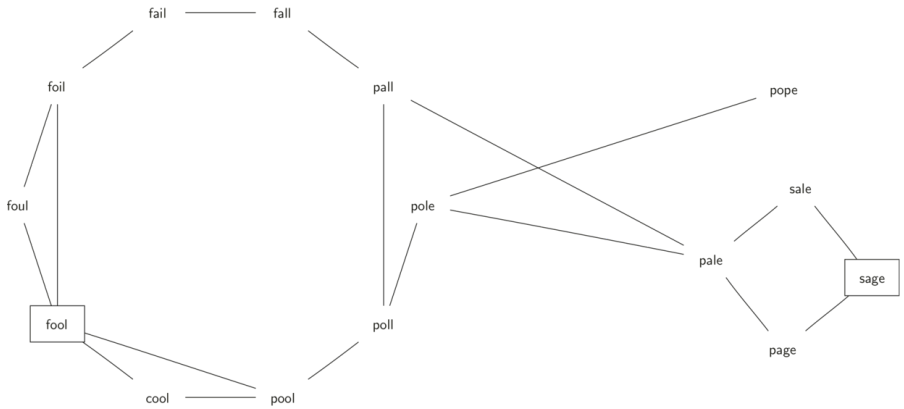
Adjacency matrix (static)

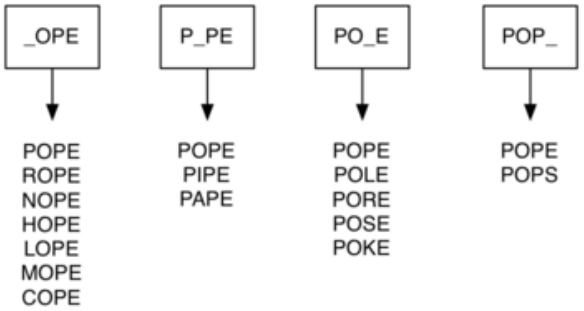


Adjacency list (dynamic)

Graphs: Word ladder problem

- We want to have an efficient search structure for a set of words
- Can we easily represent all words that are at one difference of each others
- In fact for all words, there is a highly connected graph network
- Example with simple 4 letters words (FOOL, POOL, POLL, POLE, PALE, SALE, SAGE)






```
def buildGraph(wordFile)
  d = {}
  g = Graph()
  wfile = open(wordFile, 'r')
  // Create buckets of words differing by one
  for line in wfile
    word = line[:]
    for i in range(len(word))
      bucket = word[:i] + '_' + word[i+1:]
      if bucket in d
         d[bucket].append(word)
      else
         d[bucket] = [word]
  // Add vertices and edges
  for bucket in d.keys
    for word1 in d[bucket]
      for word2 in d[bucket]
        if word1 != word2
          g.addEdge(word1, word2)
  return g
```

Graphs: Breadth-First Search

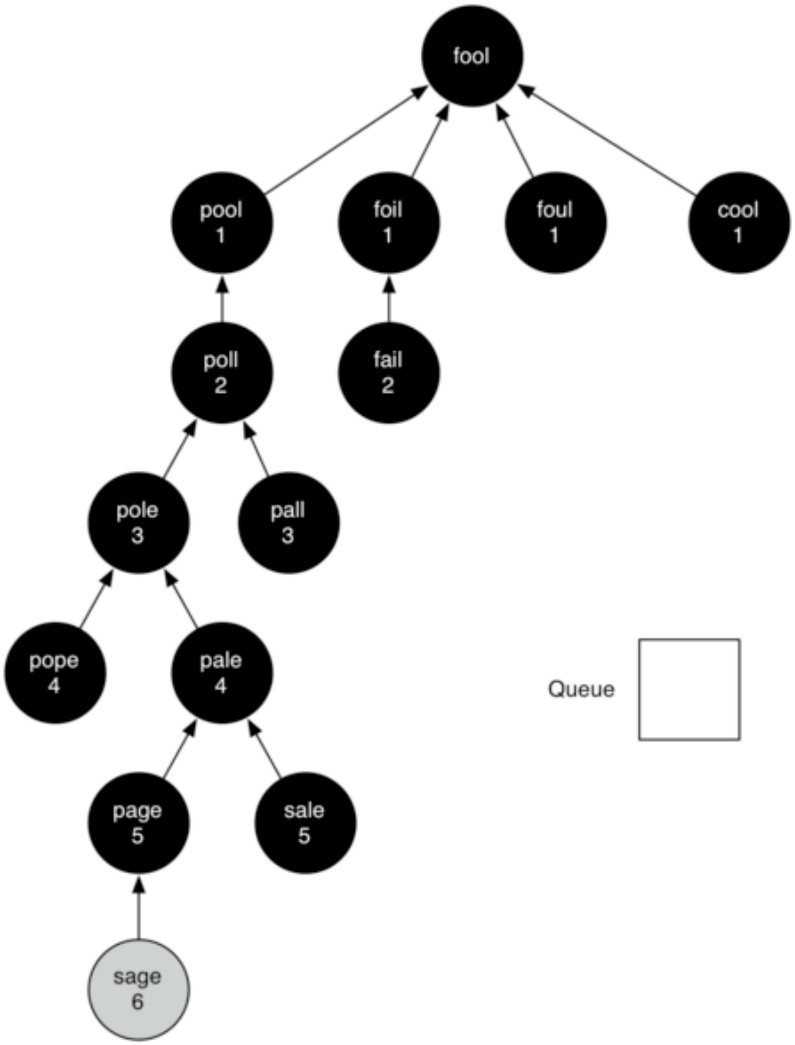
- One of the easiest algorithm for searching a graph (also prototype for further algos)
- Given a starting vertex **s**, find all vertices for which there is a path from **s**
- Finds all vertices that are at distance ***k*** from **s** before finding *any* at distance **k+1**
- Relies on « colouring » the vertices to know their status (white, gray, black)
- (This concept is heavily used in graph algorithms)

Starting from the vertex ***s*** colour it gray to know it is in processing, set its distance and predecessor and add ***s*** to a queue. Then while the queue is not empty

1. The new, unexplored vertex **nbr**, is colored gray.
2. The predecessor of **nbr** is set to the current node currentVert
3. The distance to **nbr** is set to the distance to **currentVert + 1**
4. **nbr** is added to the end of a queue. Adding nbr to the end of the queue effectively schedules this node for further exploration, but not until all the other vertices on the adjacency list of currentVert have been explored.

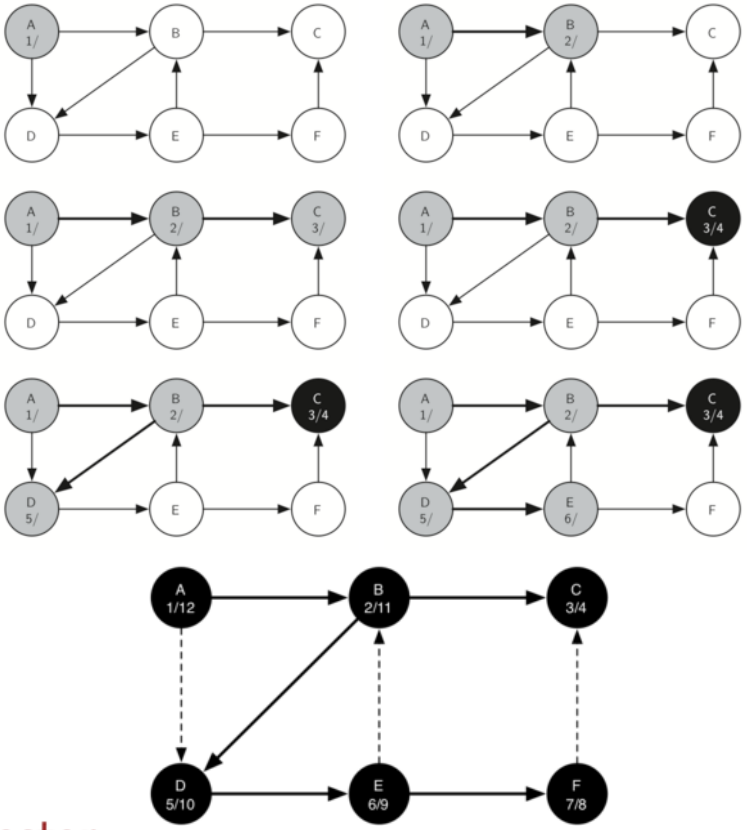
```
def bfs(g,start):
  start.setDistance(0)
  start.setPred(None)
  vertQueue = Queue()
  vertQueue.enqueue(start)
  while (vertQueue.size() > 0):
    currentVert = vertQueue.dequeue()
    for nbr in currentVert.getConnections():
      if (nbr.getColor() == 'white'):
        nbr.setColor('gray')

nbr.setDistance(currentVert.getDistance() + 1)
        nbr.setPred(currentVert)
        vertQueue.enqueue(nbr)
    currentVert.setColor('black')
```

Graphs: Breadth-First Search



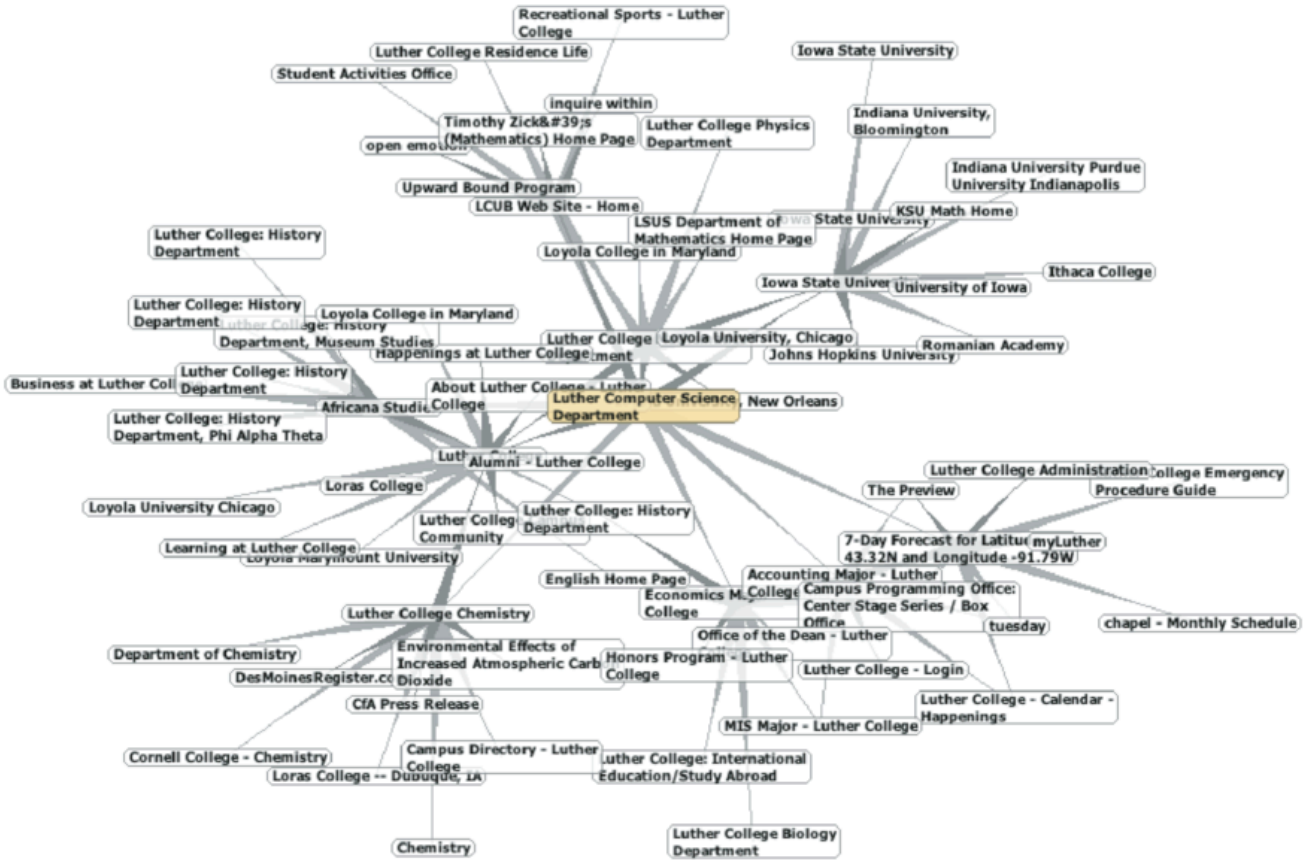
Graphs: Depth-First Search

- Goal is to search as deeply as possible to create a depth-first forest
- Allows to connect as many nodes in the graph as possible (can create multiple trees)
- Use a variable to keep track to the time of calls to the *helper* function
- Also called *finish times* that can be used to compute strongly connected

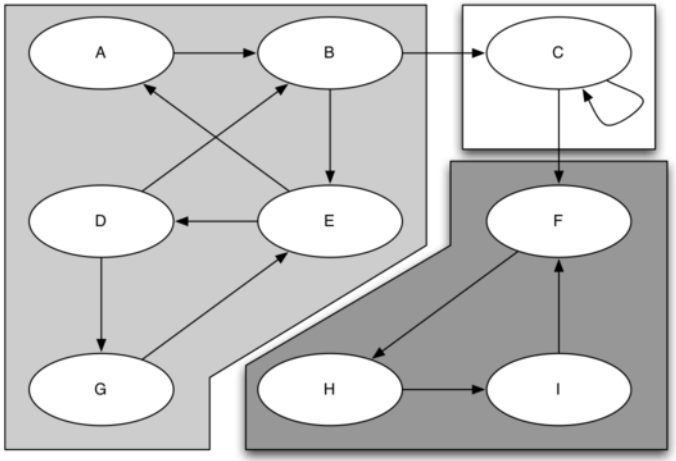


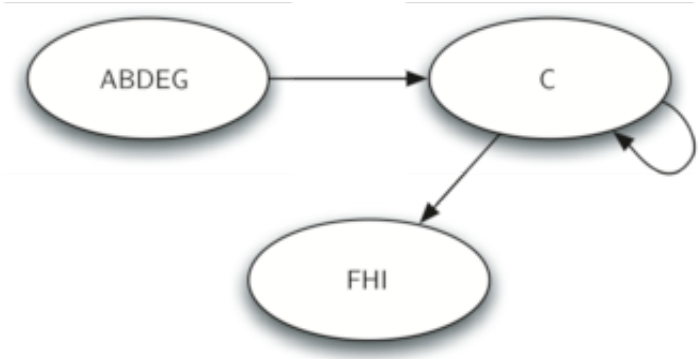

```
def dfs(self):
  for aVertex in self:
    aVertex.setColor('white')
    aVertex.setPred(-1)
  for aVertex in self:
    if aVertex.getColor() == 'white':
      self.dfsvisit(aVertex)

def dfsvisit(self,startVertex):
  startVertex.setColor('gray')
  self.time += 1
  startVertex.setDiscovery(self.time)
  for nextVertex in startVertex.getConnections()
    if nextVertex.getColor() == 'white'
      nextVertex.setPred(startVertex)
      self.dfsvisit(nextVertex)
  startVertex.setColor('black')
  self.time += 1
  startVertex.setFinish(self.time)
```

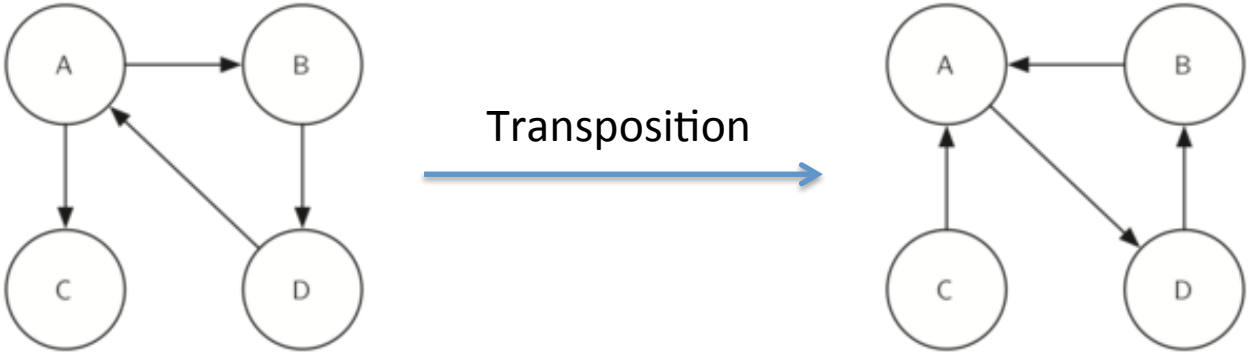
Strongly connected components



Strongly connected components

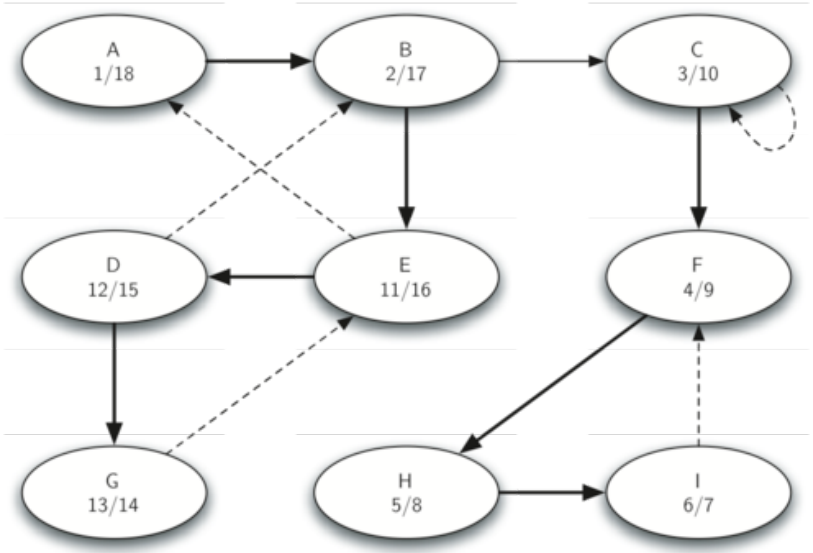


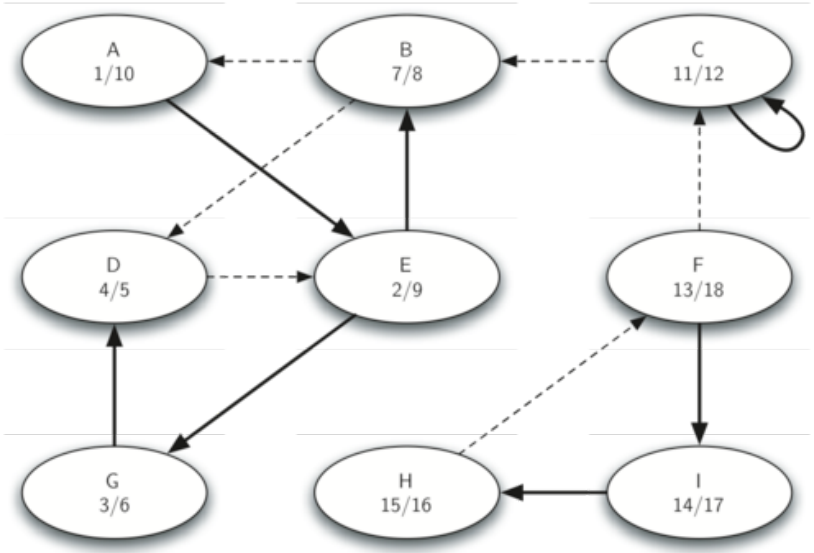




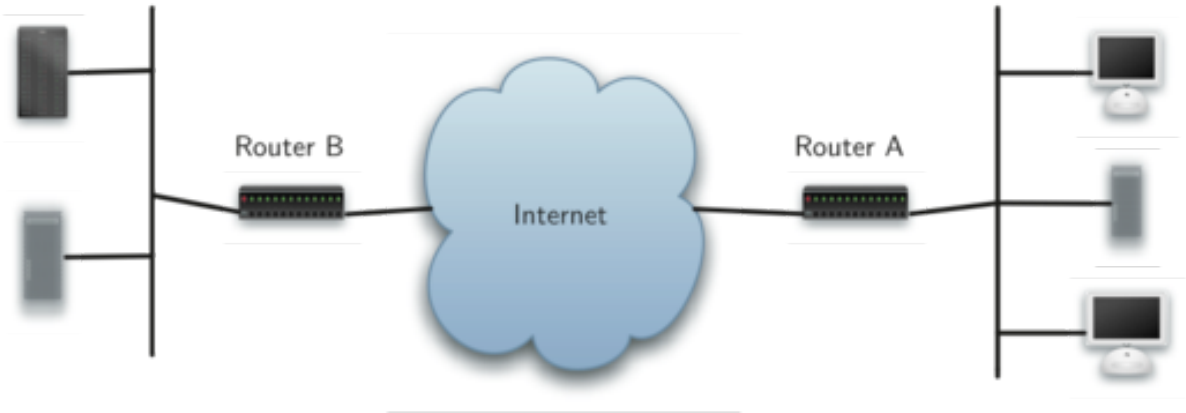
Strongly connected components

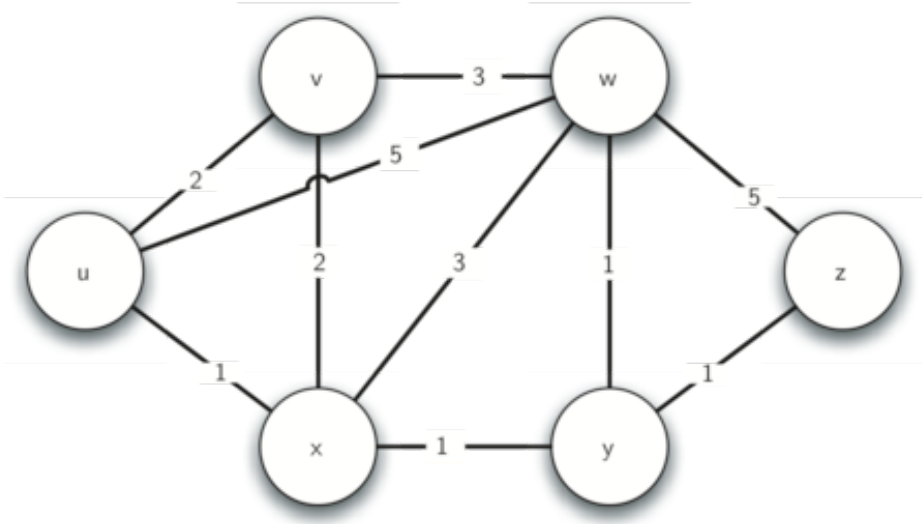
1. Call **dfs** for the graph G to compute the finish times for each vertex.
2. Compute G'.
3. Call **dfs** for the graph G' but in the main loop of DFS explore each vertex in decreasing order of finish time.
4. Each tree in the forest computed in step 3 is a strongly connected component. Output the vertex ids for each vertex in each tree in the forest to identify the component.





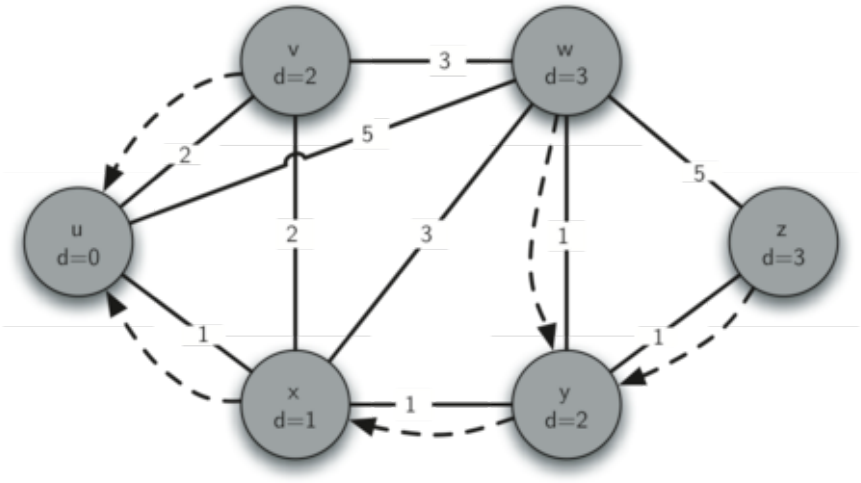
Shortest path problems





Dijkstra Algorithm

```
def dijkstra(aGraph,start)
    pq = PriorityQueue()
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v) for v in aGraph])
    while not pq.isEmpty():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
            newDist = currentVert.getDistance() \
                    + currentVert.getWeight(nextVert)
            if newDist < nextVert.getDistance():
                nextVert.setDistance( newDist )
                nextVert.setPred(currentVert)
                pq.decreaseKey(nextVert,newDist)
```

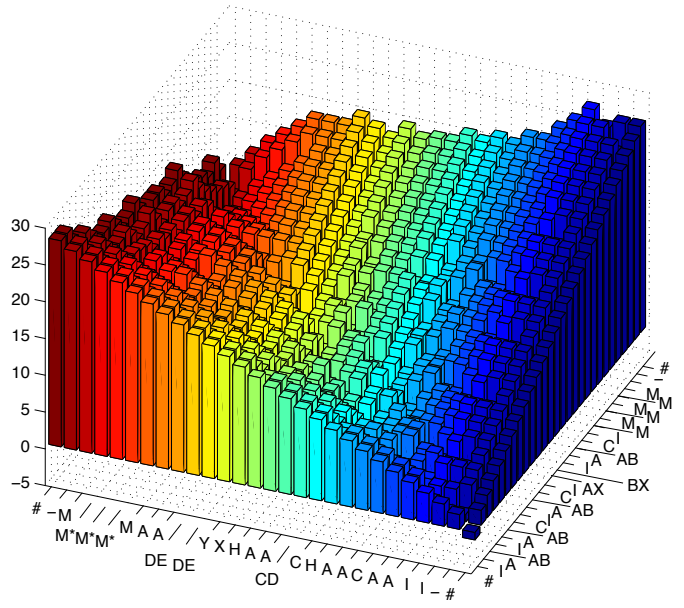


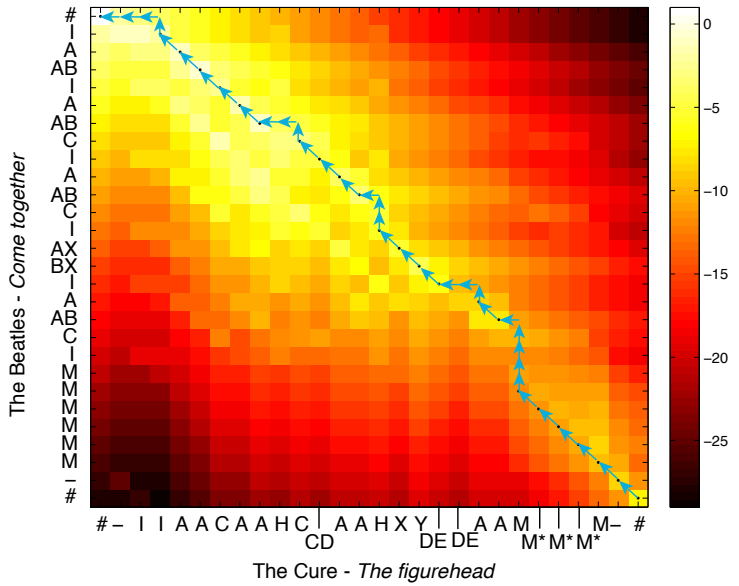
PQ = None

Sequence alignment

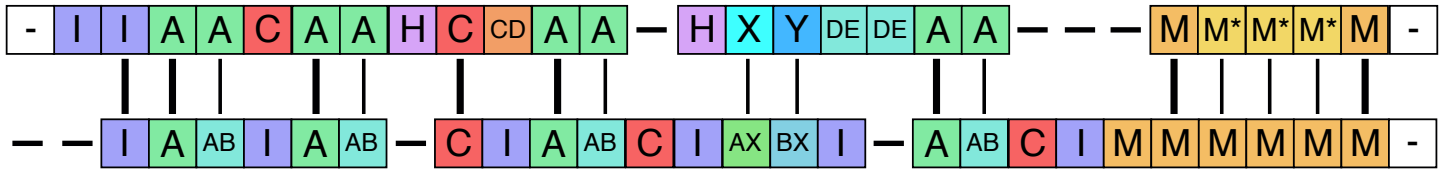
The Beatles - *Come together*

The Cure - *The figurehead*

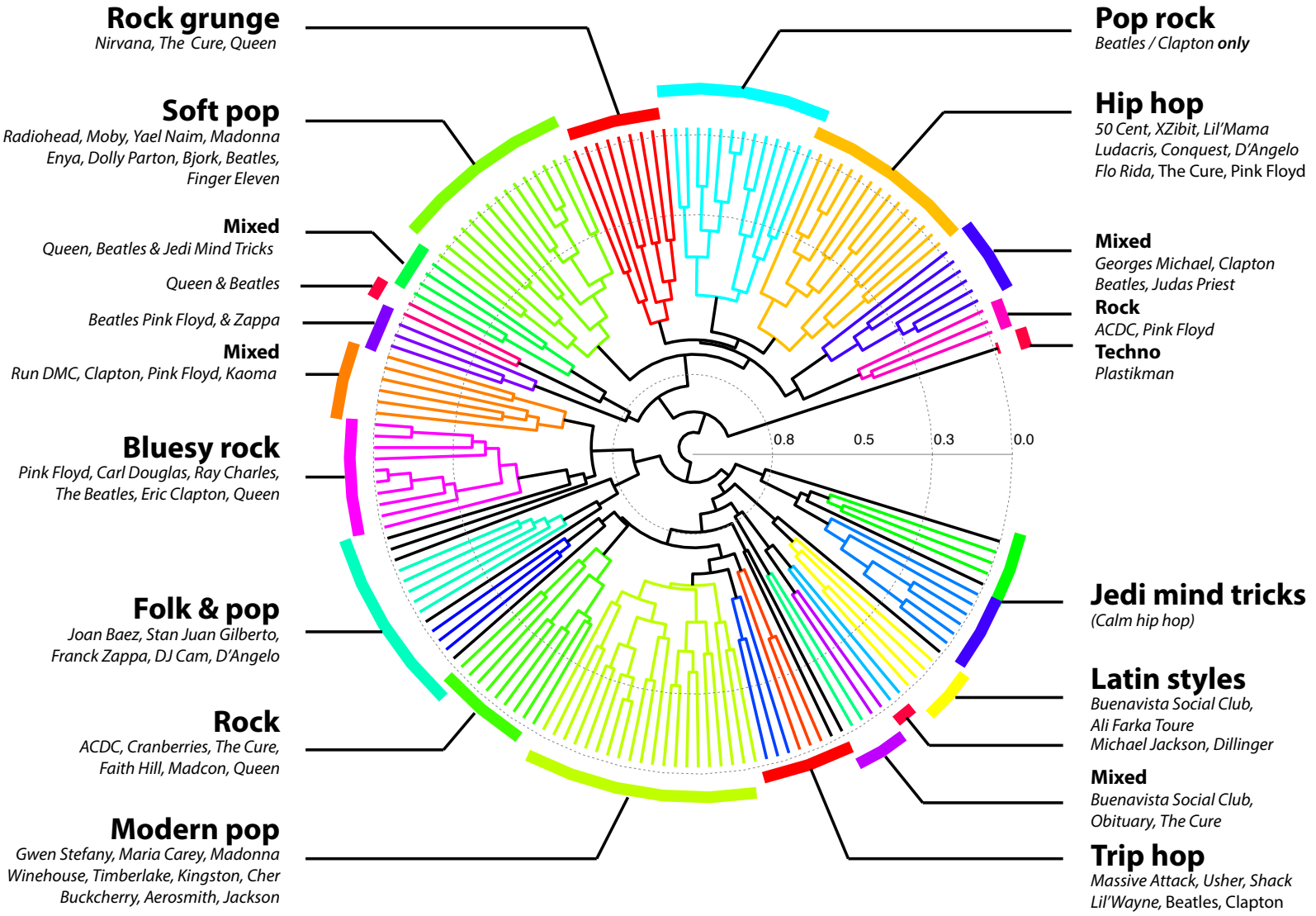




The Cure - *The figurehead*



The Beatles - *Come together*


Rock grunge
Nirvana, The Cure, Queen
Soft pop
Radiohead, Moby, Yael Naim, Madonna
Enya, Dolly Parton, Bjork, Beatles,
Finger Eleven
Mixed
Queen, Beatles & Jedi Mind Tricks
Queen & Beatles
Beatles Pink Floyd, & Zappa
Mixed
Run DMC, Clapton, Pink Floyd, Kaoma
Bluesy rock
Pink Floyd, Carl Douglas, Ray Charles,
The Beatles, Eric Clapton, Queen
Folk & pop
Joan Baez, Stan Juan Gilberto,
Franck Zappa, DJ Cam, D'Angelo
Rock
ACDC, Cranberries, The Cure,
Faith Hill, Madcon, Queen
Modern pop
Gwen Stefany, Maria Carey, Madonna
Winehouse, Timberlake, Kingston, Cher
Buckcherry, Aerosmith, Jackson
0.8
0.5
0.3
0.0
Pop rock
Beatles / Clapton only
Hip hop
50 Cent, XZibit, Lil'Mama
Ludacris, Conquest, D'Angelo
Flo Rida, The Cure, Pink Floyd
Mixed
Georges Michael, Clapton
Beatles, Judas Priest
Rock
ACDC, Pink Floyd
Techno
Plastikman
Jedi mind tricks
(Calm hip hop)
Latin styles
Buenavista Social Club,
Ali Farka Toure
Michael Jackson, Dillinger
Mixed
Buenavista Social Club,
Obituary, The Cure
Trip hop
Massive Attack, Usher, Shack
Lil'Wayne, Beatles, Clapton

Assignment !

- Dynamic programming exercise
- Is an algorithmic mix between static and dynamic structures

1. Choose the MATLAB programming language
2. Read the following research article

- Y. Jeong, M. Jeong, and O. Omitaomu, "Weighted dynamic time warping for time series classification," *Pattern Recognition*, vol. 44, pp. 2231–2240, 2011.

3. Implement the weighted DTW algorithm (15 pts)
4. Imagine and implement musical application of your choice (5 pts)
   (Don't hesitate to send me a mail with your ideas for help)

Full details and summary

http://repmus.ircam.fr/esling/atiam.html

esling@ircam.fr