Algorithms and Data Structures

Giacomo Fiumara
gfiumara@unime.it

Academic Year 2022-2023

Section 1

Introduction

Introduction to Algorithms (1)

Informal definition (1)

An algorithm is a well defined procedure that takes a certain value (or a collection of values) as **input** and produces a certain value (or a collection of values) as **output**

It is a sequence of computational steps that transforms an input into an output

Introduction to Algorithms (2)

Informal definition (2)

An instrument to solve a well defined *computational problem*

- The specific problem defines the relation existing between input and output
- An algorithm describes a specific computational procedure to obtain this relation existing between input and output

Introduction to Algorithms (3)

Example: sorting

Input:

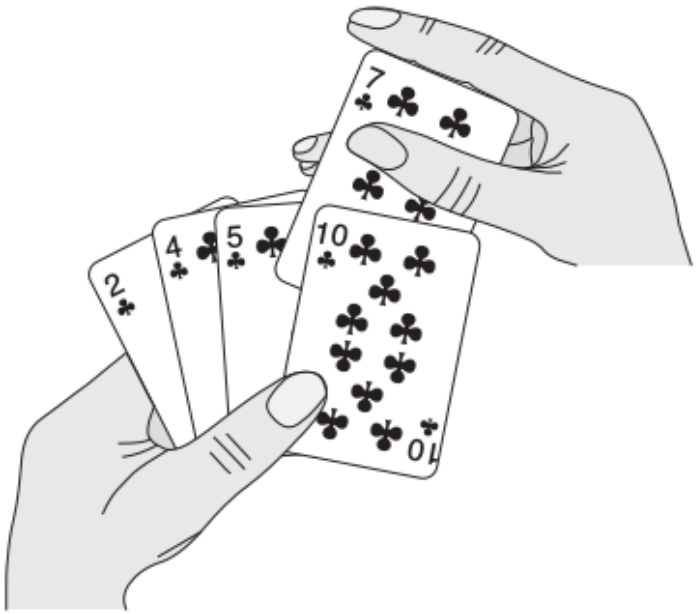
A sequence composed of n number $a_1, a_2, \ldots, a_n$

Output:

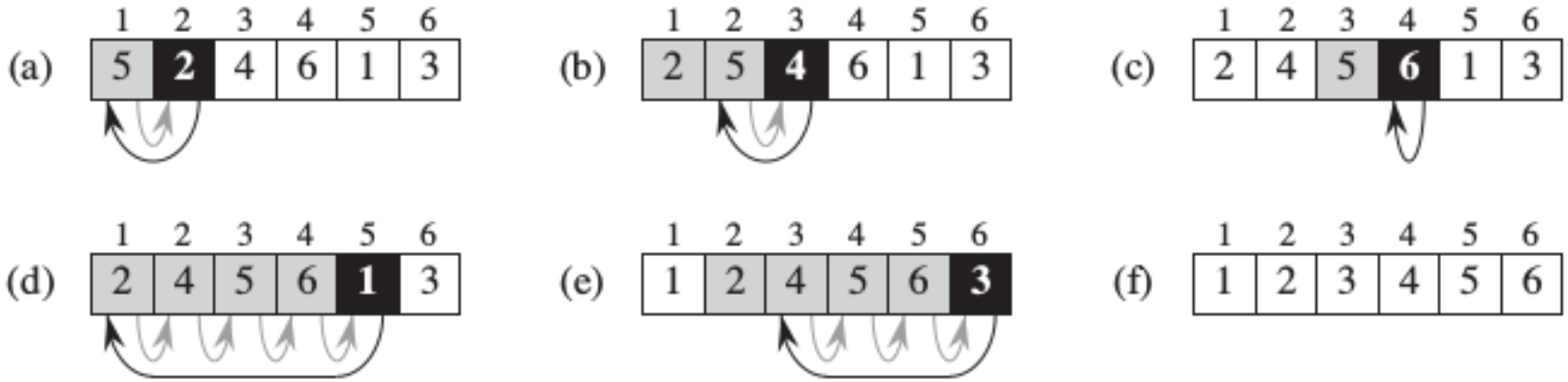
A permutation $a'_1, a'_2, \ldots, a'_n$ of the input sequence such that

$$a'_1 \leq a'_2 \leq \ldots \leq a'_n$$

Insertion sort



Insertion sort



Insertion sort

```
def InsertionSort(a):
    for j in range(1, len(a)):
        key = a[j]
        i = j
        while i > 0 and a[i-1] > key:
            a[i] = a[i-1]
            i = i - 1
            a[i] = key
            print(a)
    return a
```

ins-sort.py

Insertion sort

- At the beginning of the `for` loop, `a[1, ... j-1]` contains all sorted elements, while `a[j+1 ... n]` contains unsorted elements
- Elements `a[1, ... j-1]` which previously occupied positions `1` to `j-1`, are now sorted

- The computational cost of an algorithm depends on the input size
- Different computational costs may depend on different degrees of sorting of the input arrays

Two definitions are necessary:

Input size

- Number of elements composing the input data

- For example, the number of elements to be sorted
- In some cases, two or even more numbers may be necessary (for example, linked lists, trees, graphs)

Computational cost

- Number of elementary instructions to be executed

- Warning: this is not a time!!!
- It is erroneously named **execution time**

Insertion sort

Algorithm analysis

Instruction	Cost	No. of repetitions
`for j in range(1,len(a)):`	c_1	n
`  key = a[j]`	c_2	$n-1$
`  i = j`	c_3	$n-1$
`  while i > 0 and a[i-1] > key:`	c_4	$\sum_{j=1}^{n-1} t_j$
`    a[i] = a[i-1]`	c_5	$\sum_{j=1}^{n-1}(t_j-1)$
`    i = i - 1`	c_6	$\sum_{j=1}^{n-1}(t_j-1)$
`  a[i] = key`	c_7	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=1}^{n-1} t_j$$
$$+ c_5 \sum_{j=1}^{n-1}(t_j - 1) + c_6 \sum_{j=1}^{n-1}(t_j - 1) + c_7(n-1)$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=1}^{n-1} t_j + c_5 \sum_{j=1}^{n-1}(t_j - 1) + c_6 \sum_{j=1}^{n-1}(t_j - 1) + c_7(n-1)$$

- In general, the computational cost of an algorithm depends on the quantity and wuality of the input data
- For example, if the array is already sorted the insertion sort algorithm will require a smaller computational cost because $t_j = 1$

The equation:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=1}^{n-1} t_j + c_5 \sum_{j=1}^{n-1} (t_j - 1) + c_6 \sum_{j=1}^{n-1} (t_j - 1) + c_7(n-1)$$

can be rewritten as:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

The equation:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

implies that, in the optimal case, the computational cost is **linear** in n as it can be expressed as

$$T(n) = an + b$$

- An array sorted in reversed order represents the worst case
- In this case every element `a[j]` must be compared with every element of the sorted subarray `a[1 ... j-1]`
- In this case: $t_j = j$ per $j = 2, 3, \ldots, n$

$$\sum_{j=1}^{n-1} t_j = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$$

$$\sum_{j=1}^{n-1} (t_j - 1) = \sum_{j=1}^{n-1} (j - 1) = \frac{(n-1)^2}{2}$$

- An **Arithmetic progression** (AP) is a sequence of numbers such that the difference between the consecutive terms is constant
- Let a_1 be the first term of the AP and let d be the difference. Then:

$$a_n = a_1 + (n-1)d$$

- This relation can be written as:

$$a_k = a_j + (k-j)d$$

The sum S_n of the first n terms of an AP is given by:

$$S_n = \frac{1}{2} n(a_1 + a_n)$$

Indeed, we can write:

$$S_n = a_1 + a_2 + a_3 + \cdots + a_{n-2} + a_{n-1} + a_n$$
$$S_n = a_n + a_{n-1} + a_{n-2} + \cdots + a_3 + a_2 + a_1$$

If we sum the LHSs and the RHSs of the two above expressions:

$$\begin{aligned} 2S_n = &(a_1 + a_n) + (a_2 + a_{n-1}) + (a_3 + a_{n-2}) + \cdots \\ &+ (a_3 + a_{n-2}) + (a_2 + a_{n-1}) + (a_1 + a_n) \end{aligned}$$

Quantities in parentheses

$$2S_n = (a_1 + a_n) + (a_2 + a_{n-1}) + (a_3 + a_{n-2}) + \cdots \\ + (a_3 + a_{n-2}) + (a_2 + a_{n-1}) + (a_1 + a_n)$$

are identical. Indeed, from the definition of AP we have:

$$a_2 + a_{n-1} = a_1 + d + a_n - d = a_1 + a_n$$

$$a_3 + a_{n-2} = a_1 + 2d + a_n - 2d = a_1 + a_n$$

Therefore:

$$\begin{aligned} 2S_n &= (a_1 + a_n) + (a_1 + a_n) + (a_1 + a_n) + \cdots + (a_1 + a_n) \\ &= n(a_1 + a_n) \end{aligned}$$

from which we obtain:

$$S_n = \frac{1}{2} n(a_1 + a_n)$$

In our case, we have then:

$$\sum_{j=1}^{n-1} j = \frac{(n-1)(1+n-1)}{2} = \frac{n(n-1)}{2}$$

$$\sum_{j=1}^{n-1} (j-1) = \frac{(n-1)(1-1+n-1)}{2} = \frac{(n-1)^2}{2}$$

Equation

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=1}^{n-1} t_j + c_5 \sum_{j=1}^{n-1} (t_j - 1) + c_6 \sum_{j=1}^{n-1} (t_j - 1) + c_7(n-1)$$

becomes:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=1}^{n-1} j + c_5 \sum_{j=1}^{n-1} (j - 1) + c_6 \sum_{j=1}^{n-1} (j - 1) + c_7(n-1)$$

And finally we have:

$$T(n) = an^2 + bn + c$$

Whose coefficients a, b e c are related to the computational costs c_i.

Concluding remark

- In the best case, the computational complexity of insertion sort is linear
- In the worst case, is quadratic with n

Section 2

Divide and conquer

- Some algorithms are **recursive**, which means that they call themselves to manage subproblems of the same type but with smaller input data size
- Usually recursive algorithms adopt the **divide and conquer** approach which is composed of the following three steps:

  Divide

  Conquer

  Combine

- **Divide**

  Problems are divided in a number of subproblems, of the same kind but having a smaller input data size

- **Conquer**

  A recursive approach is adopted for subproblems. When their input data size is small enough, subproblems are eventually solved

- **Combine**

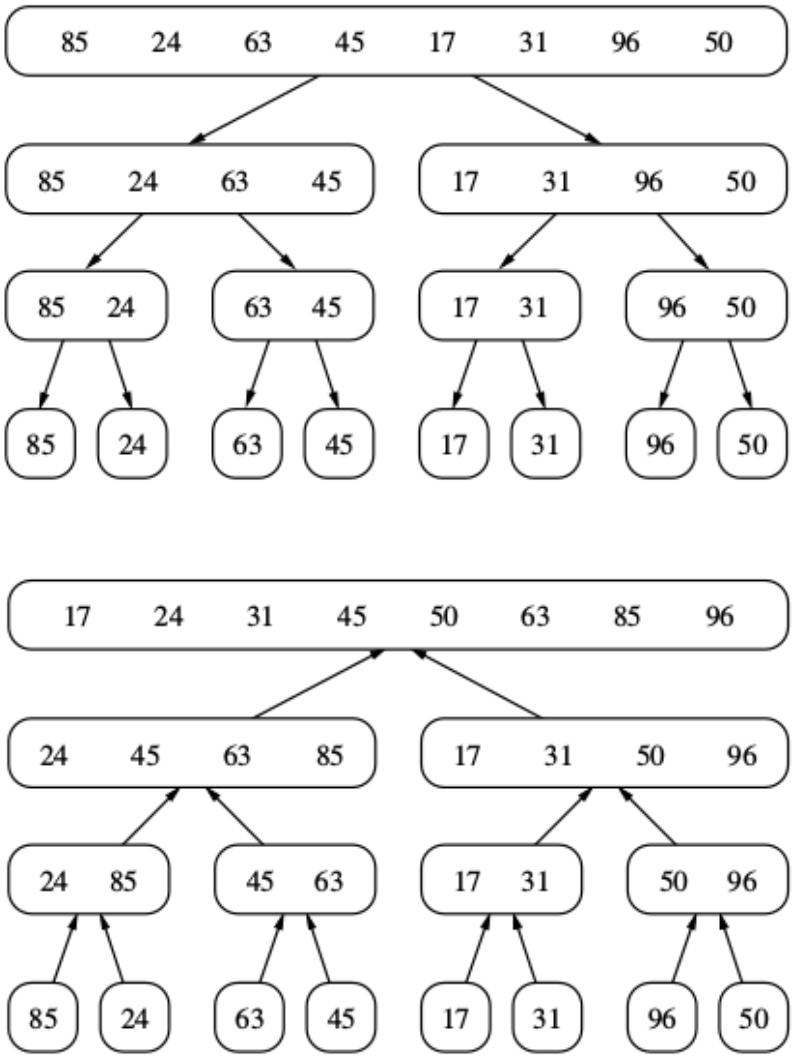
  Solutions of subproblems are eventually combined to obtain the solution of the original problem

Goal: sort a sequence `s` composed of `n` elements using the Divide and Conquer approach

- **Divide**
  If S contains at least two elements its elements must be copied in two sequences S_1 and S_2 containing $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ elements respectively
- **Conquer**
  Recursively sort sequences S_1 and S_2
- **Combine**
  Copy elements in `S` merging the two sorted sequences S_1 ed S_2 in a sorted sequence

Divide and conquer

Merge sort



Divide and conquer

Merge sort

```
def merge(s1,s2,s):
    i = j = 0
    while i+j < len(s):
        if j==len(s2) or (i<len(s1) and s1[i]<s2[j]):
            #print("IF: ",i,'\t',j,'\t')
            s[i+j] = s1[i]
            i += 1
        else:
            #print("ELSE: ",i,'\t',j,'\t')
            s[i+j] = s2[j]
            j += 1
```

Divide and conquer

Merge sort

```
def mergesort(s):
    n = len(s)
    if n < 2:
        return
    mid = n // 2
    s1 = s[0:mid]
    s2 = s[mid:n]
    print("Prima: ",s1,s2,s)
    mergesort(s1)
    mergesort(s2)
    merge(s1,s2,s)
    print(s1,s2,s)
```

Divide and conquer

Merge sort

```
Array:
[85, 24, 63, 45, 17, 31, 96, 50]

[24, 85]
[45, 63]
[24, 45, 63, 85]
[17, 31]
[50, 96]
[17, 31, 50, 96]
[17, 24, 31, 45, 50, 63, 85, 96]
```

- When an algorithm contains a recursive call, the computational cost can be computed using a **recurrence equation**
- In general, a recurrence equation expresses the computational cost of a problem having input dimension n as a function of the computational cost of the subproblems working on the parts in which the input is subdivided
- If the input size is sufficiently small ($n \leq c$), for some constant c, then the computational cost is constant
- Otherwise, the problem must be divided in a subproblems, each operating on n/b input data

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{negli altri casi} \end{cases}$$

Where

- $T(n/b)$ is the computational cost of a subproblem
- $D(n)$ is the computational cost for dividing the current problem in subproblems
- $C(n)$ is the computational cost for combining current subproblems

- $T(n/b)$

  The current problem is divided in two subproblems each working on $n/2$ input data. Therefore the **Conquer** phase costs $2T(n/2)$

- $D(n)$

  In the Division phase the median element of the subarray must be identified. This operation has a constant computational cost, $D(n) = \Theta(1)$

- $C(n)$

  `merge` function costs $\Theta(n)$

Therefore:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 2T(n/2) + \Theta(n) & \text{se } n > 1 \end{cases}$$

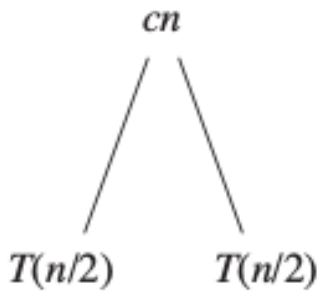
which can be rewritten as:

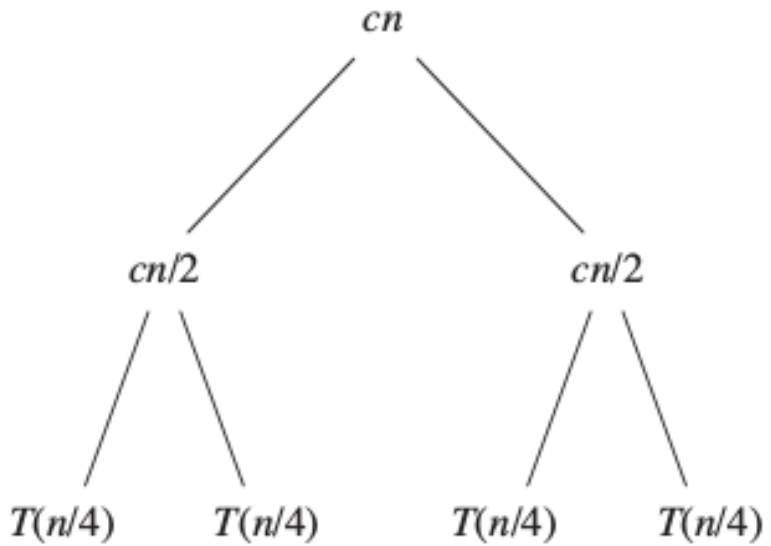
$$T(n) = \begin{cases} c & \text{se } n = 1 \\ 2T(n/2) + cn & \text{se } n > 1 \end{cases}$$

c represents the computational cost necessary to solve, divide and combine subproblems having dimension 1

Divide and conquer

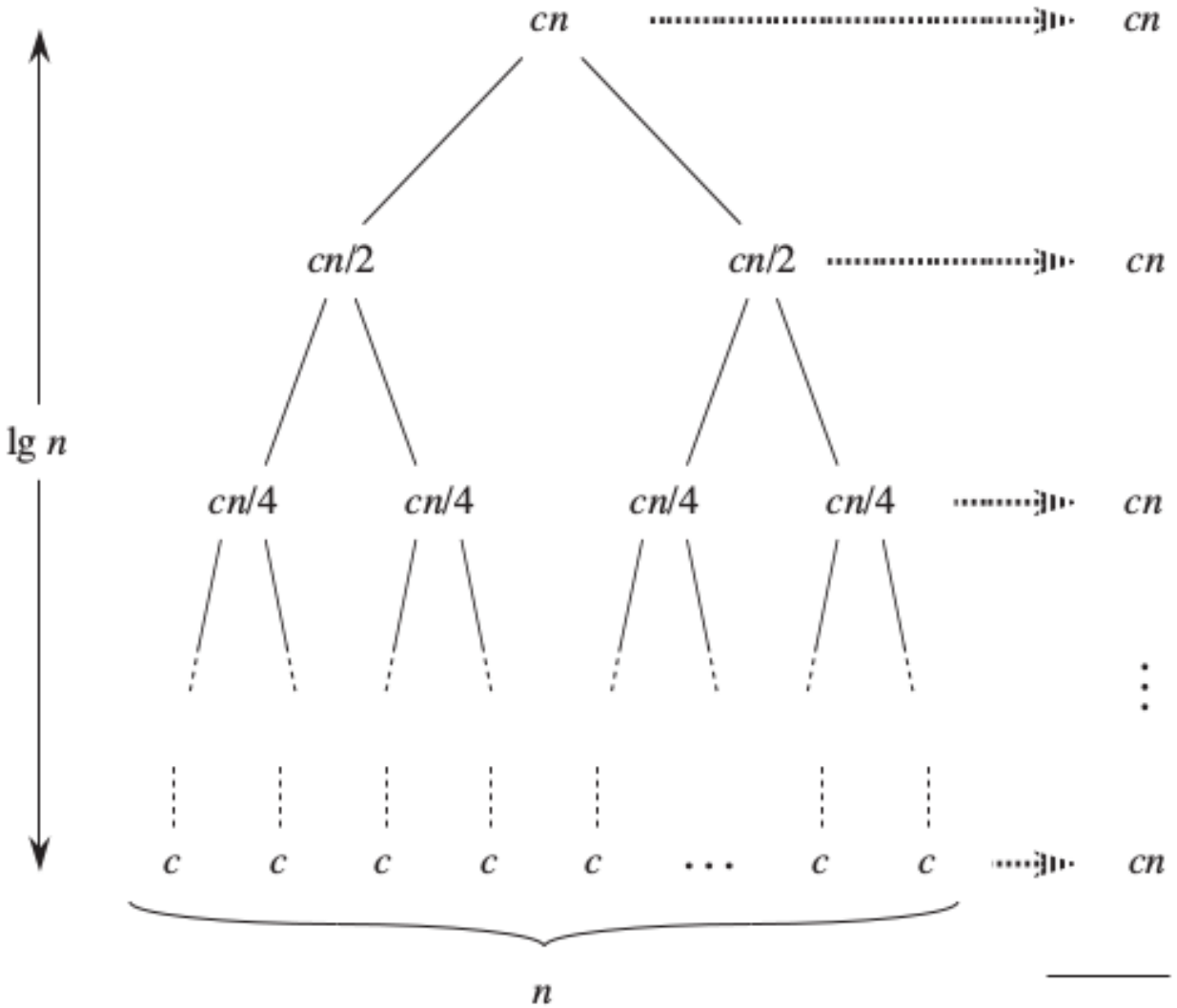
Merge sort





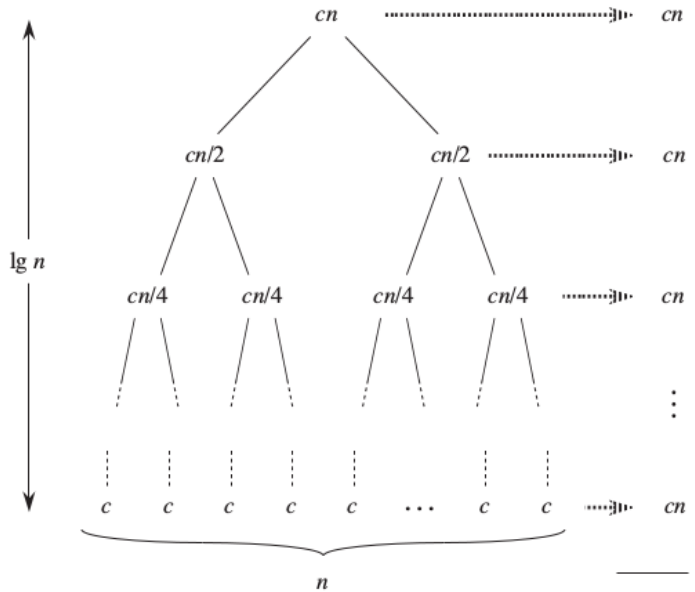
Divide and conquer

Merge sort

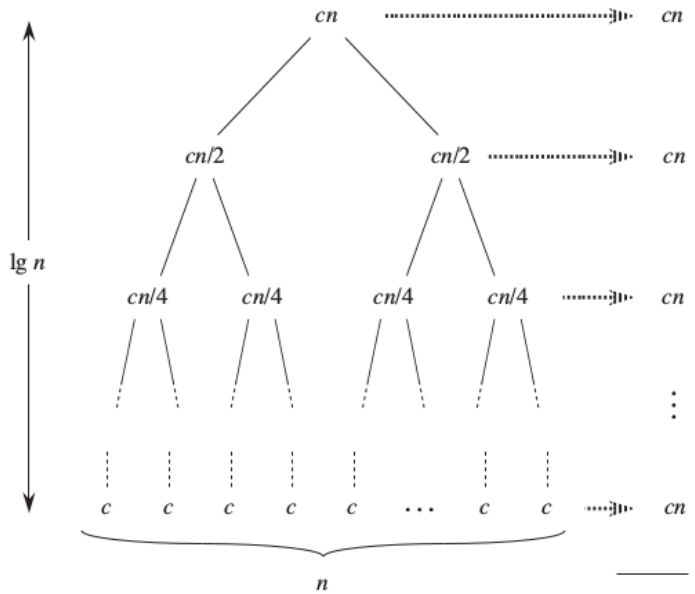


Divide and conquer

Merge sort



- How many levels of recursion an algorithm has?
- If n is power of 2, at each level the problem is divided in two subproblems $(n/2, n/4, \ldots, n/(2^i))$



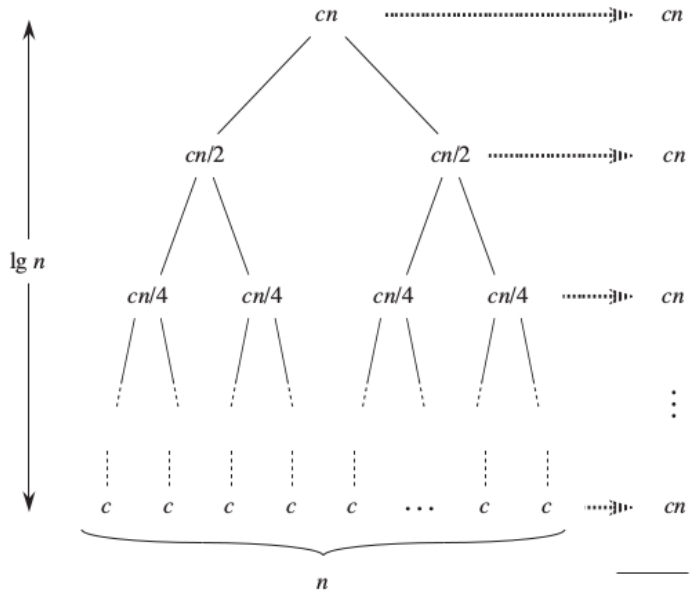
– Recursion stops when $n/(2^i) = 1$, namely when

$$n = 2^i$$

$$i = \log_2 n = \lg n$$

Divide and conquer

Merge sort



- Every level costs cn
- The problem can be divided in $\lg n$ levels
- We have therefore: $\Theta(n \lg n)$

Section 3

Growth of Functions

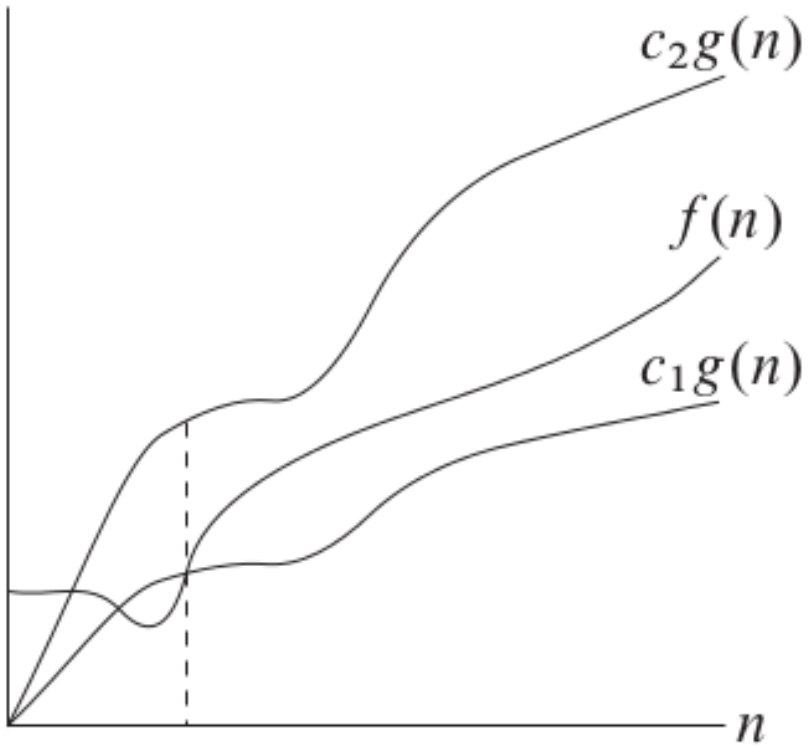
For a given function $g(n)$, we denote by $\Theta(g(n))$ the *set of functions*

$$\Theta(g(n)) = \{f(n) : \exists \text{ positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0\}$$

A function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n

Algorithm Analysis

Θ-notation



$f(n) = \Theta(g(n))$

From the definition of $\Theta(g(n))$

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0\}$$

follows that every $f(n) = \Theta(g(n))$ must be asymptotically nonnegative, that is, $f(n)$ be nonnegative whenever n is sufficiently large.

Consequently, the function $g(n)$ itself must be asymptotically nonnegative, or else the set $\Theta(g(n))$ is empty.

As an example, consider any quadratic function

$$f(n) = an^2 + bn + c$$

where a, b, c are constants and $a > 0$.

The lower-order terms of an asymptotically positive function can be ignored in determining asymptotically tight bounds because they are insignificant for large n.

Disregarding the lower-order terms and ignoring the constants yelds $f(n) = \Theta(n^2)$

More generally, for any polynomial

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

where the a_i are constants and $a_d > 0$, we have $p(n) = \Theta(n^d)$

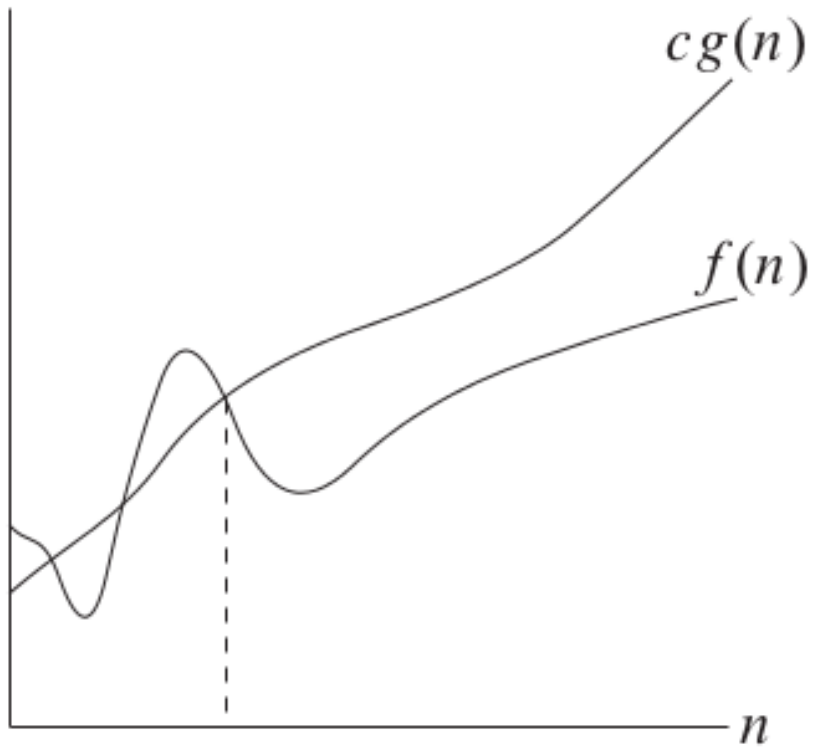
Since any constant is a degree-0 polynomial, we can express any constant function as $\Theta(n^0)$ or $\Theta(1)$

- The Θ-notation asymptotically bounds a function from above and below
- When we have only an asymptotic upper bound, we use O-notation
- For a given function $g(n)$ we denote by $O(g(n))$ the set of functions

$$O(g(n)) = \{f(n) : \exists \text{ positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0\}$$

Algorithm Analysis

O-notation

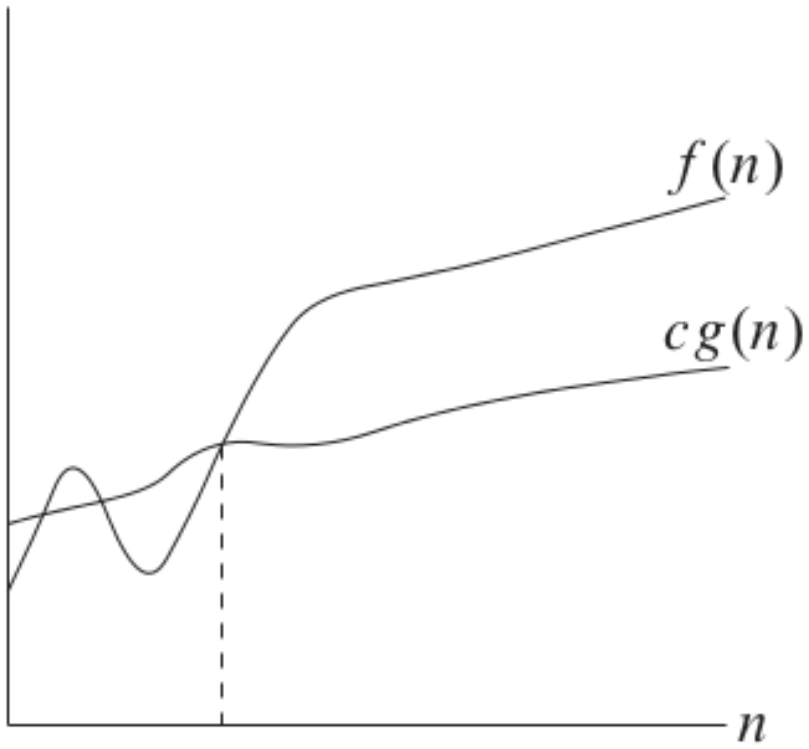


$f(n) = O(g(n))$

- We write $f(n) = O(g(n))$ to indicate that a function $f(n)$ is member of the set $O(g(n))$
- Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, since Θ-notation is a stronger notion than o-notation
- Since O-notation describes an upper bound, when we use it to bound the worst-case computational cost of an algorithm, we have a bound on the computational cost of the algorithm on every input

- Just as O-notation provides an asymptotic **upper** bound on a function, Ω-notation provides an **asymptotic lower bound**
- For a given function $g(n)$ we denote by $\Omega(g(n))$ the set of functions

$$\Omega(g(n)) = \{f(n) : \exists \text{ positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \quad \forall n \geq n_0\}$$



$f(n) = \Omega(g(n))$

From the definition of the asymptotic notations, this important theorem follows

Theorem

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Given a sequence S consisting of n numbers, we want to compute a sequence A such that $A[j]$ is the average of elements $S[0], S[1], \ldots, S[j]$ for $n = 0, 1, \ldots, n-1$, that is

$$A[j] = \frac{\sum_{i=0}^{j} S[i]}{j+1}$$

Algorithm Analysis

An example: prefix averages (first implementation)

```
def prefix_average1(S):
    n = len(S)
    A = [0] * n
    for j in range(n):
        total = 0
        for i in range(j+1):
            total += S[i]
        A[j] = total / (j + 1)
    return A
```

- The statement, `n = len(S)` is executed in constant time (the `list` class maintains an instance variable that records the current length of the list)
- The statement, `A = [0] * n`, causes the creation and initialization of a Python list having length n, and with all entries equal to zero. This uses a constant number of primitive operations per element, and thus runs in $O(n)$ time
- The body of the outer loop, controlled by j is executed n times. Therefore statements `total = 0` and `A[j] = total / (j + 1)` are executed n times each. These two statements contribute a number of primitive operations proportional to n, that is, $O(n)$ time

- The body of the inner loop, controlled by i is executed $j + 1$ times, depending on the current value of the outer loop counter j.
- Thus, statement `total += S[i]` is executed $1 + 2 + 3 + \cdots + n = n(n+1)/2$ times, which implies that the statement contributes $O(n^2)$ time
- The computational cost of this implementation is given by the sum of three terms: the first and the second terms are $O(n)$, and the third term is $O(n^2)$. Therefore the computational cost is $O(n^2)$.

Algorithm Analysis

An example: prefix averages (second implementation)

```
def prefix_average2(S):
    n = len(S)
    A = [0] * n
    for j in range(n):
        A[j] = sum(S[0:j+1]) / (j + 1)
    return A
```

- The expression `sum(S[0:j+1])` is a function call and its evaluation takes $O(j+1)$ time
- The computation of the slice `S[0:j+1]` also uses $O(j+1)$ time, as it constructs a new list instance for storage
- The computational cost of this implementation is dominated by a series of steps that take time proportional to $1+2+3+\cdots+n = n(n+1)/2$, and thus $O(n^2)$.

Algorithm Analysis

An example: prefix averages (third implementation)

```
def prefix_average3(S):
    n = len(S)
    A = [0] * n
    for j in range(n):
        total += S[j]
        A[j] = total / (j + 1)
    return A
```

- The initialization of variables n and *total* uses $O(1)$ time
- The initialization of the list `A` uses $O(n)$ time
- There is a single loop, controlled by j. This contributes a total of $O(n)$ time
- The body of the loop is executed n times, for $j = 0, 1, \ldots, n-1$. Thus, statements `total += S[j]` and `A[j] = total / (j + 1)` are executed n times each: their contribution is $O(n)$
- The computational cost of this implementation is given by the sum of the four terms: the first is $O(1)$ and the remaining are $O(n)$. Therefore, the computational cost is $O(n)$.

Section 4

Divide and Conquer

- The factorial of a positive integer n, denoted with $n!$, is defined:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{se } n \geq 1 \end{cases}$$

- The factorial function is used to express the number of ways in which n distinct items can be arranged into a sequence, that is, the number of **permutations** of n items

- There is a natural recursive definition for the factorial function. In general, $n! = n \cdot (n-1)!$
- The recursive definition can be formalized as

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n \geq 1 \end{cases}$$

- This definition is typical of many recursive definitions
- It contains one or more **base cases**, which are defined nonrecursively in terms of fixed quantities
- In this case, $n = 0$ is the base case

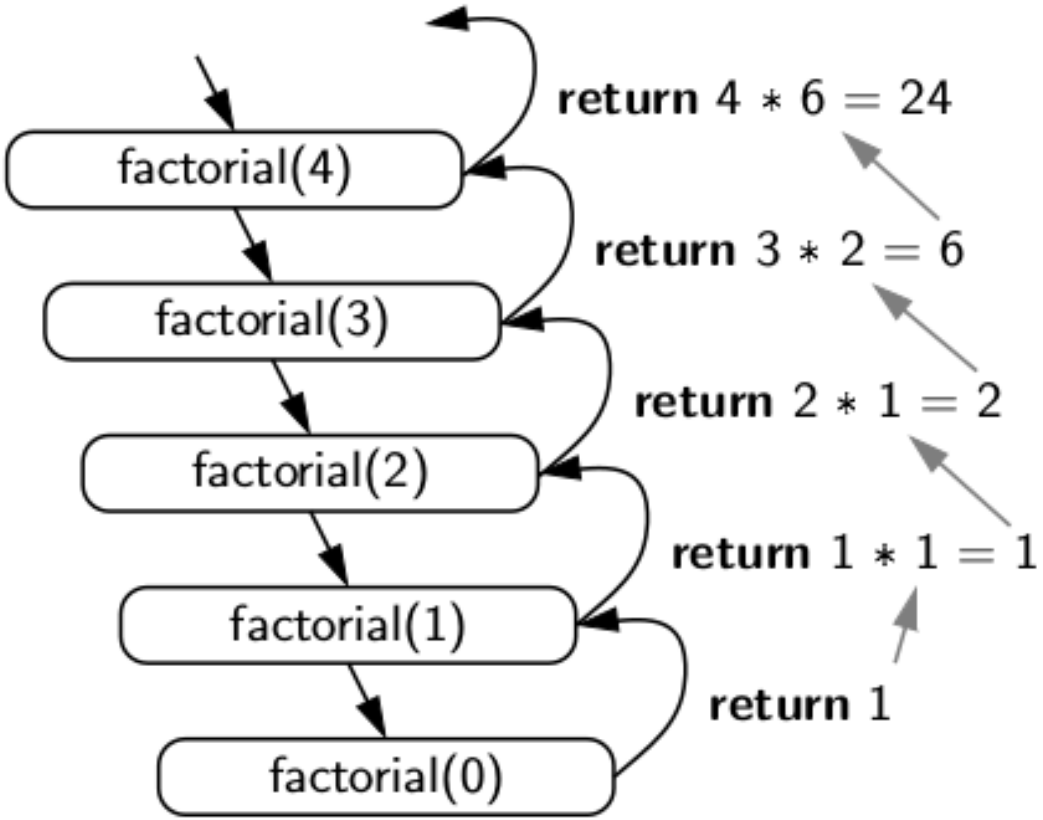
Recursion

The Factorial Function

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n*factorial(n-1)
```

Recursion

The Factorial Function



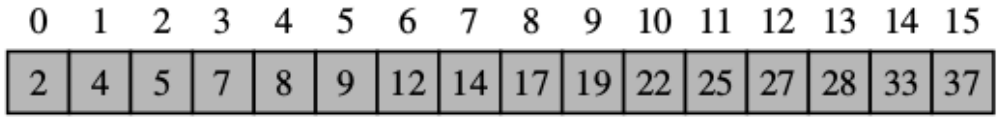
- The estimate of the computational cost of a recursive algorithm must consider the number of function calls and the operations therein contained
- In this case, $n + 1$ calls are performed
- A constant number of operation is executed at each call
- The overall computational complexity of this algorithm is therefore $O(n)$

- It is a classic recursive algorithm, that is used to efficiently locate a target value within a sorted sequence of n elements
- When the sequence is unsorted, the standard approach to search for a target value is to use a loop to examine every element (**sequential search**)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

- When the sequence is **sorted** and **indexable**, there is a much more efficient algorithm
- For any index j, all the values stored at indices $0, \ldots, j-1$ are less than or equal to the value at index j
- All the values stored at indices j $(j+1, j+2, \ldots, n-1)$ are greater than or equal to the value stored at j
- This observation allows to implement an efficient searching algorithm



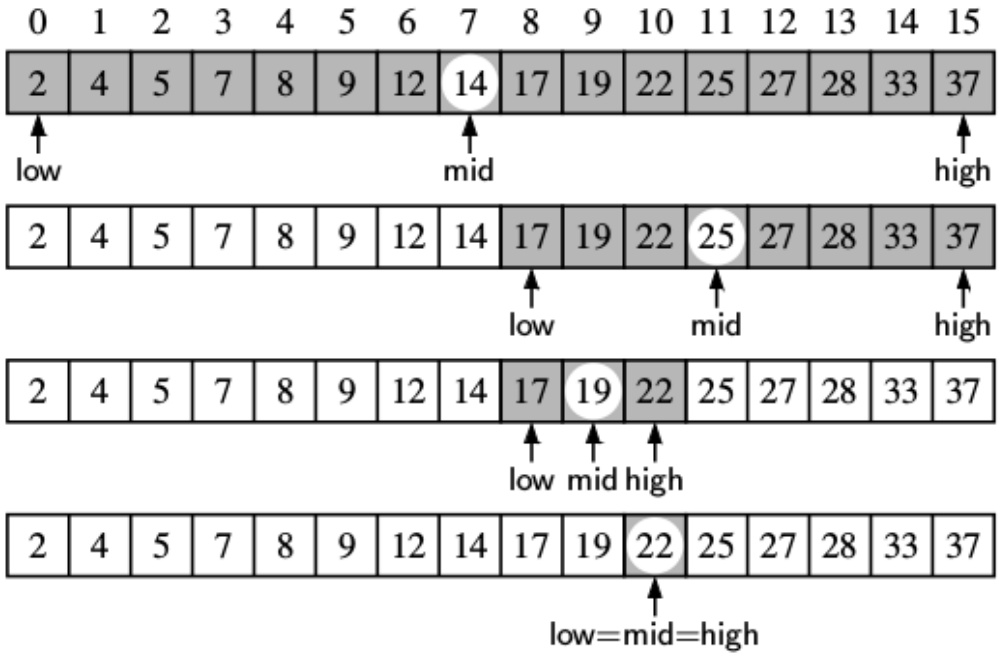
- The algorithm uses two parameters, `low` and `high`, representing the minimum and maximum indices, respectively
- Initially, `low =0` and `high =n-1`
- The target value is compared to the median value `data[mid]`, defined as:

$$mid = \lfloor (low + high)/2 \rfloor$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

Three cases:

- If *target* = *data*[*mid*] the item has been found and search terminates successfully
- If *target* < *data*[*mid*], the search must proceed recursively on the first half of the sequence, on the interval of indices `low`, `mid` -1)
- Se *target* > *data*[*mid*], the search must proceed recursively on the second half of the sequence, on the interval of indices `mid`$+1$, `high`


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low
mid
high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low
mid
high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low mid high
2 4 5 7 8 9 12 14 17 19 22 25 27 28 33 37
low=mid=high

```
def binarysearch(data, target, low, high):
    if low > high:
        return False
    else:
        mid = (low + high)//2
        print("Interval: ", low, mid, high, " Median value: "
            ,data[mid])
    if target == data[mid]:
        print("Found")
        return True
    elif target < data[mid]:
        print("Left half")
        return binarysearch(data, target, low, mid-1)
    else:
        print("Right half")
        return binarysearch(data, target, mid+1, high)
```

binarysearch.py

- At each call the number of elements of the sequence is divided in half
- Initially, the number of candidate values is n; at the second call becomes $n/2$, at the third $n/4$ and so on
- At the $j-$th call, the number of candidate values is not larger than $n/2^j$

- The maximum number of calls is the smallest number r such that:

$$\frac{n}{2^r} < 1 \qquad \text{, cioè} \qquad r = \lfloor \log n + 1 \rfloor$$

- This implies that the computational complexity of the binary search algorithm is $O(\log n)$

Section 5

Stack

Stack

- A **stack** is a collection of elements that can be inserted or removed according to the principle **last-in, first-out (LIFO)**
- Elements can be inserted in any moment, but it is possible to access or remove only the last inserted element (**top**)

Stack: ADT

– Formally, a stack is a (**Abstract Data Type (ADT)**) that supports the methods:

`push(e)`	Inserts the element `e` on top of the stack
`pop()`	Removes the element on top of the stack and returns it. If the stack is empty an error is returned

Stack: ADT

– Useful, albeit not fundamental as `push` and `pop` are:

`top()`	Returns a reference to the top element of the stack. If the stack is empty, an error is returned
`is_empty()`	Returns `True` is the stack is empty
`length(S)`	Returns the number of elements contained in the stack `S`

Stack: ADT (1/2)

```
def create():
    return []

def push(mylist, element):
    mylist.append(element)
    return mylist

def pop(mylist):
    return mylist.pop()

def top(mylist):
    return mylist[-1]
```

Stack: ADT (2/2)

```
def is_empty(mylist):
    # return False if mylist
    if mylist:
        return False
    else:
        return True

def length(mylist):
    return len(mylist)
```

Stack: ADT

– Computational costs

`push(e)`	$O(1)$
`pop()`	$O(1)$
`top()`	$O(1)$
`is_empty()`	$O(1)$
`length(S)`	$O(1)$

Section 6

Queues

Queues

Queues

- A **queue** is a collection of elements that can be inserted or removed according to the principle **first-in, first-out (FIFO)**
- Elements can be inserted in any moment, but it is possible to access or remove only the first inserted element

Queues: ADT

– Formally, a queue is a (**Abstract Data Type (ADT)**) that supports the methods:

`insert(e)`	Inserts the element `e` at the bottom of the queue
`extract()`	Removes the first element from the queue and returns it. If the queue is empty an error is returned

Queues: ADT

– Useful, albeit not fundamental as `push` and `pop` are:

`first()`	Returns a reference to the first element of the queue without removing it. If the queue is empty an error is returned
`is_empty()`	Returns `True` if the queue is empty
`length(Q)`	Returns the number of elements contained in the queue

Double-ended queues

- A **double-ended queue** (deque (pron. "deck")) is a queue that allows elements insertion or removal from both ends
- It is a generalization of LIFO and FIFO structures, in which only one of the two ends can be accessed
- Application: a people in a queue decides to quit before it is his time

Double-ended queues: ADT (1/2)

`D.add_first(e)`	An element is inserted on top of the queue
`D.add_last(e)`	An element is inserted at the bottom of the queue
`D.delete_first()`	The top element is removed and returned. If the queue is empty an error is returned
`D.delet_last()`	The last element is removed and returned. If the queue is empty an error is returned

Double-ended queues: ADT (2/2)

`D.first()`	The top element is returned (without any removal)
`D.last()`	The last element is returned (without any removal)
`D.is_empty()`	Returns `True` if the queue is empty
`length(D)`	Returns the number of elements contained in the queue `D`

Double-ended queues

- The module `collections` contains `deque`, an implementation of a double-ended queue
- Supported methods are:

`append(x)`	`appendleft(x)`	`clear()`
`copy()`	`count(x)`	`extend(iterable)`
`extendleft(iterable)`	`index(x)`	`insert(i,x)`
`pop()`	`popleft()`	`remove(value)`
`reverse()`	`rotate(n)`	

Double-ended queues: ADT (1/2)

```
def add_first(dq, element):
    dq.insert(0,element)
    return dq

def add_last(dq, element):
    dq.append(element)
    return dq

def del_first(dq):
    if len(dq):
        return dq.pop(0)
    return False

def del_last(dq):
    if len(dq):
        return dq.pop()
    return False
```

Double-ended queues: ADT (2/2)

```
def first(dq):
    if dq:
        return dq[0]
    return False

def last(dq):
    if dq:
        return dq[-1]
    return False

def is_empty(dq):
    return len(dq) == 0
```

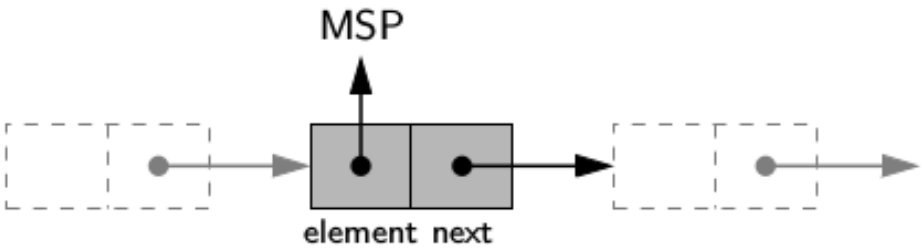
Section 7

Linked lists

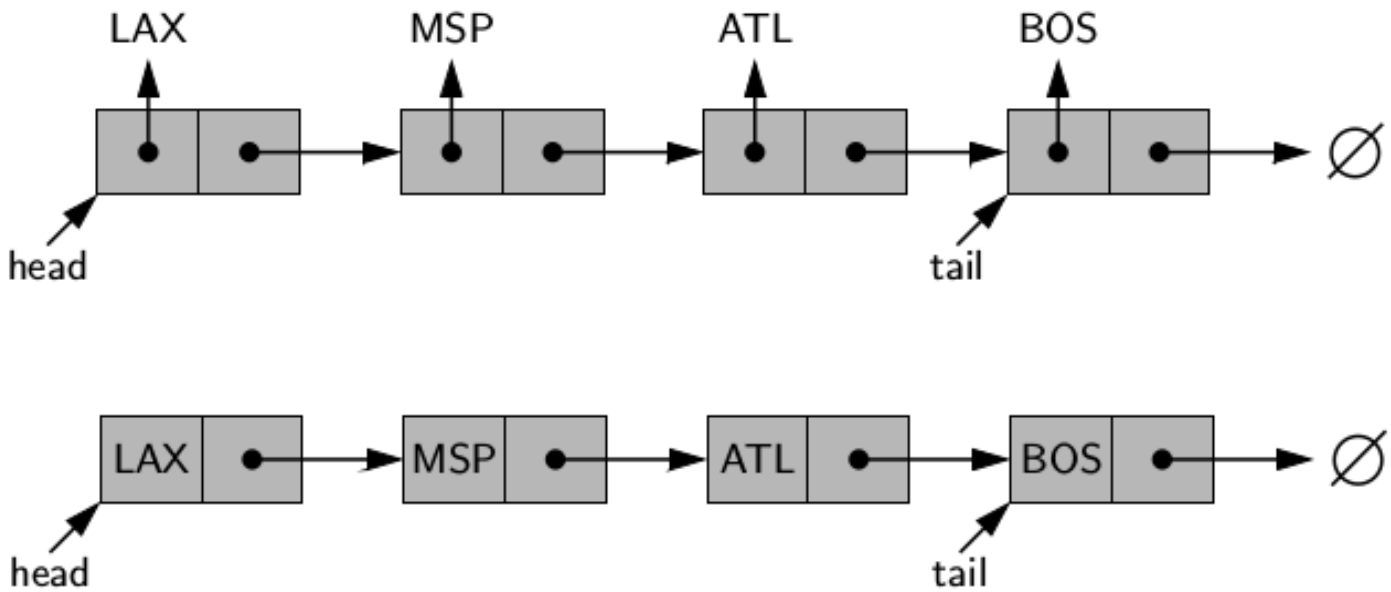
Linked Lists

- A **linked list** is a data structure that stores the elements in a certain order
- Unlike a Python list, in a linked list is focused a **node** is allocated for each element
- Each node maintains a reference to its element and one or more references to neighboring nodes
- Thanks to this references, the linear order of the sequence can be represented

Singly Linked Lists

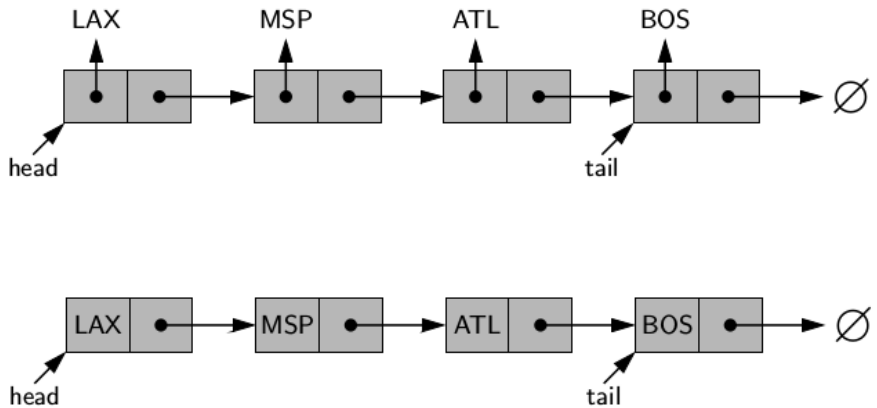


Singly Linked Lists



- The first element of a list is called **head**, the last is the **tail**
- By starting at the head and moving from one node to another by following the references, it is possible to reach the tail of the list (**traversing**)

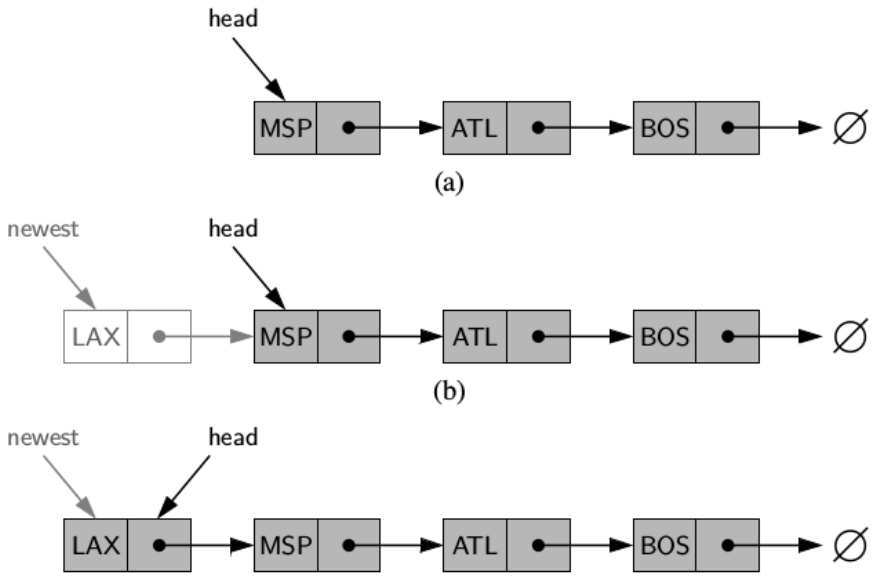
Singly Linked Lists




```
def create():
    llist = dict()
    llist['head'] = 'tail'
    llist['tail'] = 0
    return llist
```

Singly Linked Lists

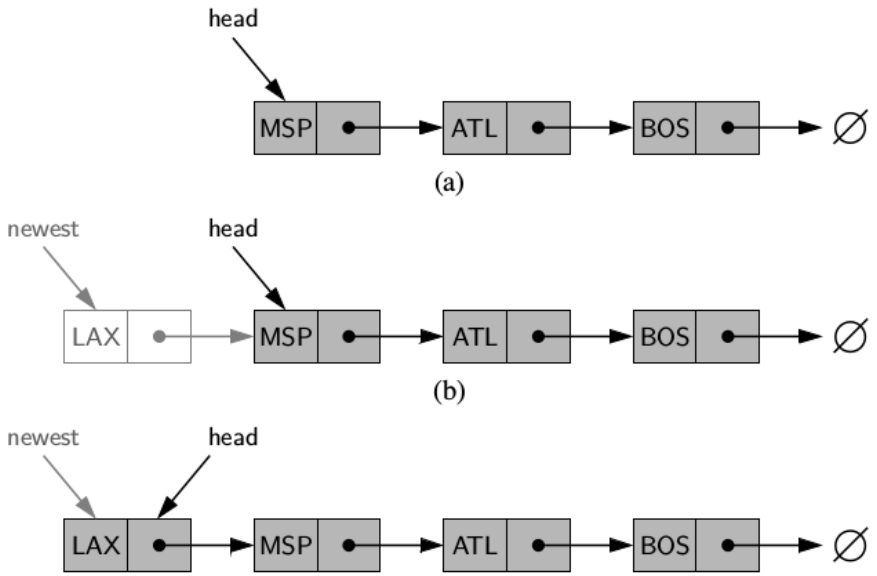
Inserting an Element at the Head



- A linked list has not a predetermined fixed size
- A new element can be inserted at the head of the list

Singly Linked Lists

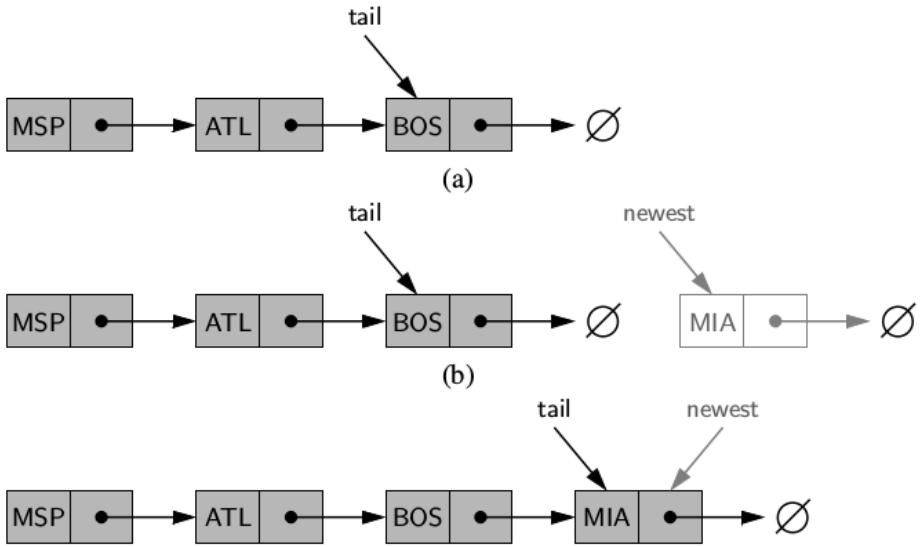
Inserting an Element at the Head




```
def add_first(llist,element):
    llist[element] = llist['head']
    llist['head'] = element
    return llist
```

Singly Linked Lists

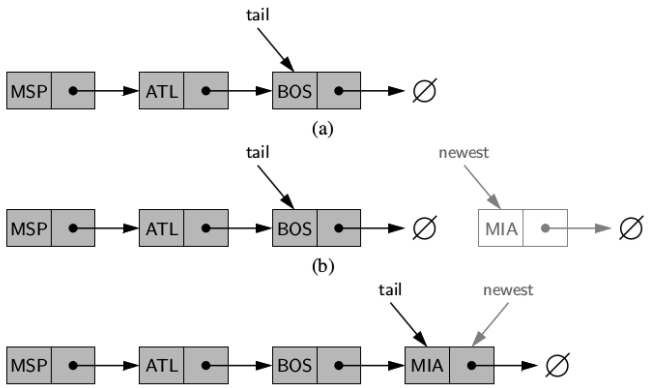
Inserting an Element at the Tail



– To insert a new element at the tail: 1) create a new element and, 2) update the tail of the list to point to this new node

Singly Linked Lists

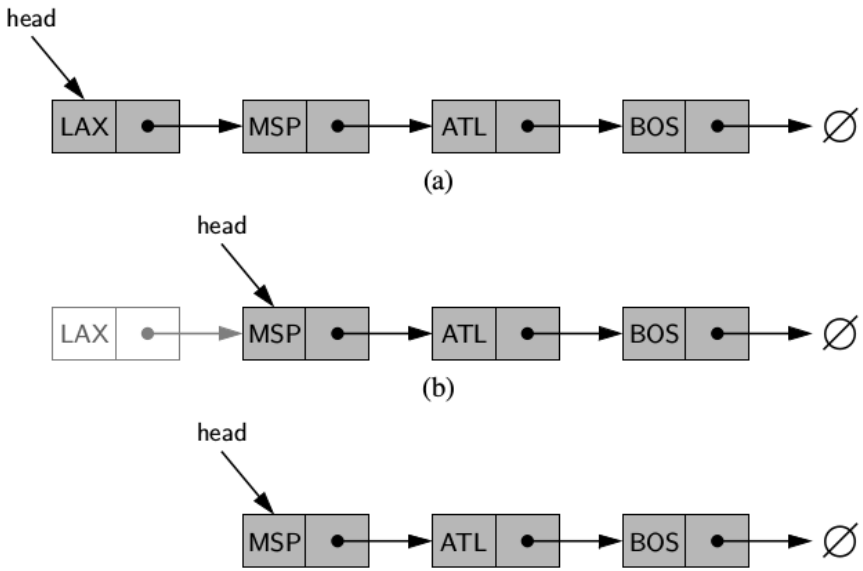
Inserting an Element at the Tail




```
def add_last(llist,element):
    curr = 'head'
    while curr != 'tail':
        next = llist[curr]
        if next == 'tail':
            llist[e] = llist[curr]
            llist[curr] = e
            break
        curr = next
    return llist
```

Singly Linked Lists

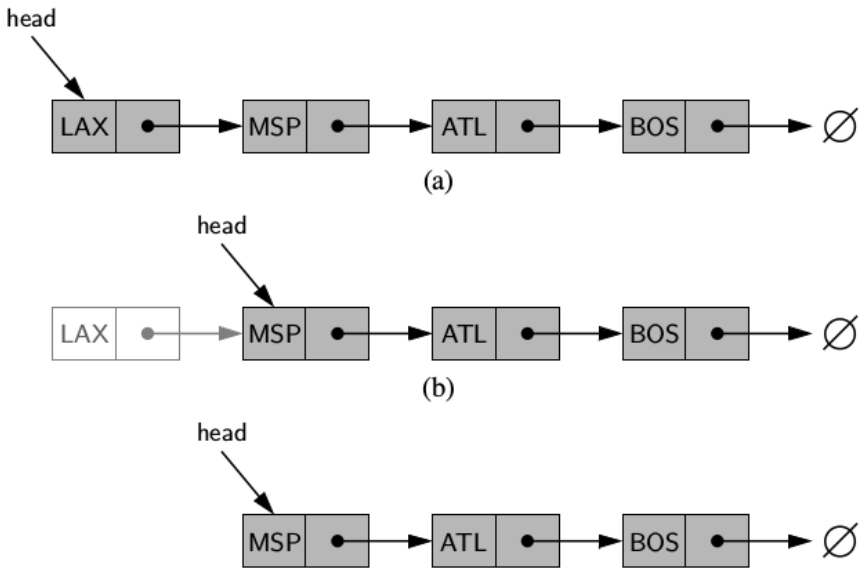
Removing an Element from a Singly Linked List



– Removing an element from the head of a list is the reverse operation of inserting a new element at the head

Singly Linked Lists

Removing an Element from a Singly Linked List




```
def del_first(llist):
    x = llist['head']
    llist['head'] = llist[x]
    del llist[x]
```

Singly Linked Lists

Removing the Last Element from a Singly Linked List

- The last node of a singly linked list cannot be easily deleted
- Indeed, it is necessary to maintain the reference to the node before the last node
- Actually, it is impossible to traverse the linked list going from the tail to the head
- Therefore, the only way consists in starting from the head and search through the list

Singly Linked Lists 1/5

```
def create():
    llist = dict()
    llist['head'] = 'tail'
    llist['tail'] = 0
    return llist

def add_first(llist,element):
    llist[element] = llist['head']
    llist['head'] = element
    return llist
```

Singly Linked Lists 2/5

```
def add_last(llist,e):
    curr = 'head'
    while curr != 'tail':
        next = llist[curr]
        if next == 'tail':
            llist[e] = llist[curr]
            llist[curr] = e
            break
        curr = next
    return llist

def del_first(llist):
    x = llist['head']
    llist['head'] = llist[x]
    del llist[x]
```

Singly Linked List 3/5

Iterative implementation of traversal

```
def itraverse(llist):
    curr = 'head'
    path = []
    path.append((0,curr))
    k = 1
    while curr != 'tail':
        next = llist[curr]
        path.append((k,next))
        curr = next
        k += 1
    return path
```

Singly Linked Lists 4/5

Recursive implementation of traversal

```
def rtraverse(llist, path, start):
    if start == 'tail':
        print('Level: ', len(path))
        path.append((len(path), start))
        return path
    else:
        print('Level: ', len(path))
        path.append((len(path), start))
        return rtraverse(llist, path, llist[start])
```

Singly Linked Lists 5/5

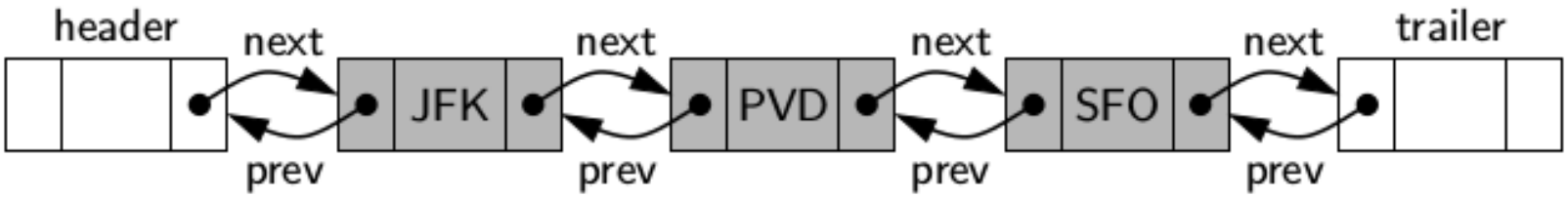
```
def printall(llist):
    curr = 'head'
    while curr != 'tail':
        next = llist[curr]
        print(curr, "\t-->\t", next)
        curr = next
```

Doubly Linked Lists

- A **doubly linked list** is defined as a list whose nodes maintain an explicit reference to the previous and next nodes
- Unlike the singly linked list, insertions and deletions at arbitrary positions can be carried out in $O(1)$
- It helps to add special nodes at both ends of the list (**header** node at the beginning, **trailer** node at the end)

Doubly Linked Lists

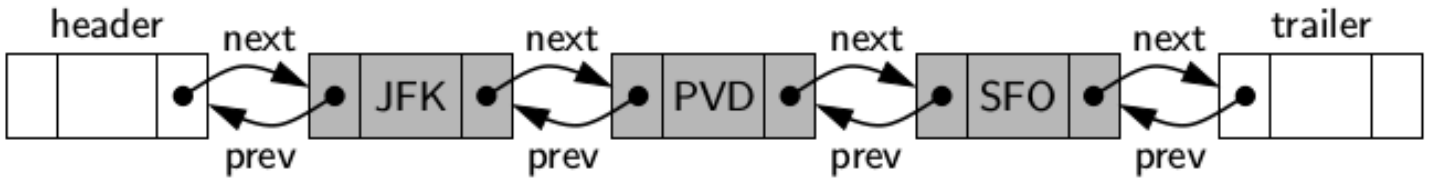
- These dummy nodes are called **sentinel nodes** and do not store elements of the list
- When using sentinel nodes, an empty list is initialized so that the `next` field of the header points to the trailer and the `prev` field of the trailer points to the header



Doubly Linked Lists

Inserting and Removing Elements

– For a nonempty list, the header's `next` will refer to a node containing the first real element of a sequence, just as the trailer's `prev` references the node containing the last real element of a sequence



Doubly Linked Lists

```
def create():
    llist = dict()
    llist['head'] = {'prev': None,'next': 'tail'}
    llist['tail'] = {'prev': 'head','next': None}
    return llist

def printall(llist):
    curr = 'head'
    while curr != 'tail':
        next = llist[curr]['next']
        print(curr, "\t<==>\t", next)
        curr = next
```

Doubly Linked Lists

```
def insert_between(llist, e, pre, suc):
    llist[pre]['next'] = e
    llist[suc]['prev'] = e
    llist[e] = {'prev': pre, 'next': suc}
    return llist

def add_first(llist, e):
    return insert_between(llist, e, 'head', llist['head']['
        next'])

def add_last(llist, e):
    return insert_between(llist, e, llist['tail']['prev'], '
        tail')
```

Doubly Linked Lists

```
def delete(llist, e):
    pre = llist[e]['prev']
    suc = llist[e]['next']
    llist[pre]['next'] = suc
    llist[suc]['prev'] = pre
    del llist[e]
    return llist
```

Section 8

Trees

Trees


La famiglia dei paperi
L'albero genealogico secondo Don Rosa
gli AMICI DEI PAPERI

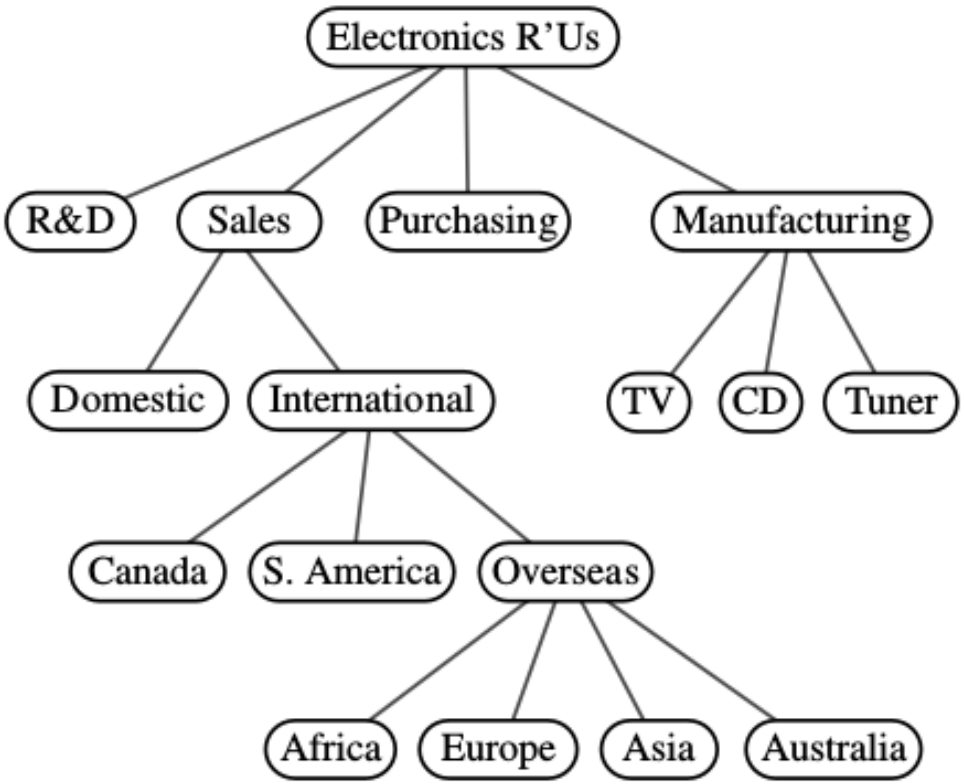
Trees

- A **tree** is a non linear data structure
- Trees allow to efficiently implement some algorithms and represent a way to manage data (file systems, graphic interfaces, databases, web sites, etc)
- A tree has a non linear structure, which allows to implement more sophisticated relationships than sequences
- Indeed, are naturally implemented parent/child (ancestor/descendant) relationships

Trees

- A tree is an abstract data type that stores elements hierarchically
- With the exception of the top element, each element in a tree has a parent element and zero or more children elements
- Formally, a tree T is a collection of nodes that store elements with a parent-child relationship such that
  - If T is nonempty there is a special node, called **root** of T, without parent node
  - Every node v of T different from the root has only one parent node w; every node having w as parent node is called **child node** of w

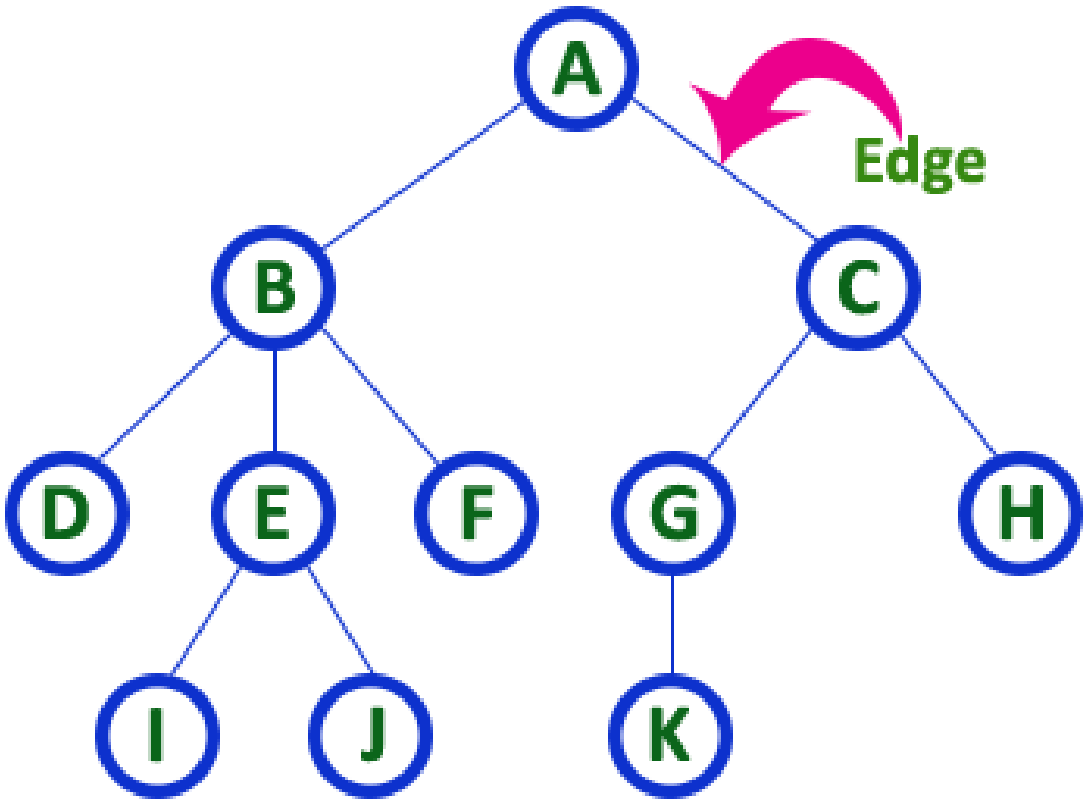
Trees


Electronics R'Us
R&D
Sales
Purchasing
Manufacturing
Domestic
International
TV
CD
Tuner
Canada
S. America
Overseas
Africa
Europe
Asia
Australia

Trees

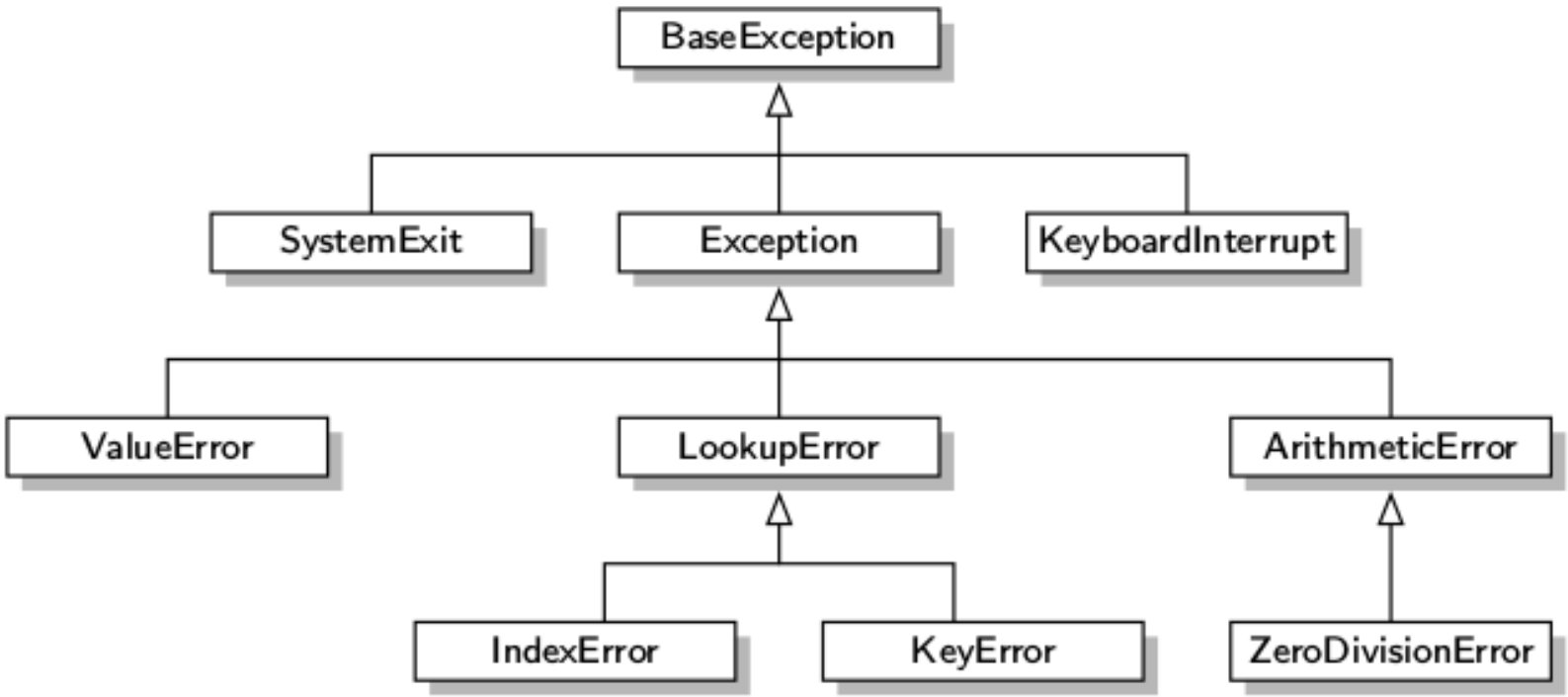
- Two nodes sharing the same parent nodes are said **sibling**
- A node with no children nodes is said **external**
- A node with one or more children nodes is said **internal**
- External nodes are also called **leaves**
- A node u is called an **ancestor** of node v if $u = v$ or u is an ancestor of the parent of v
- Conversely, a node v is a **descendant** of a node u if u is an ancestor of v
- The **subtree** of T rooted at a node v is the tree consisting of all the descendants of v in T (including v itself)

– An **edge** of a tree T is a pair of nodes (u, v) such that u is **parent** of v or viceversa



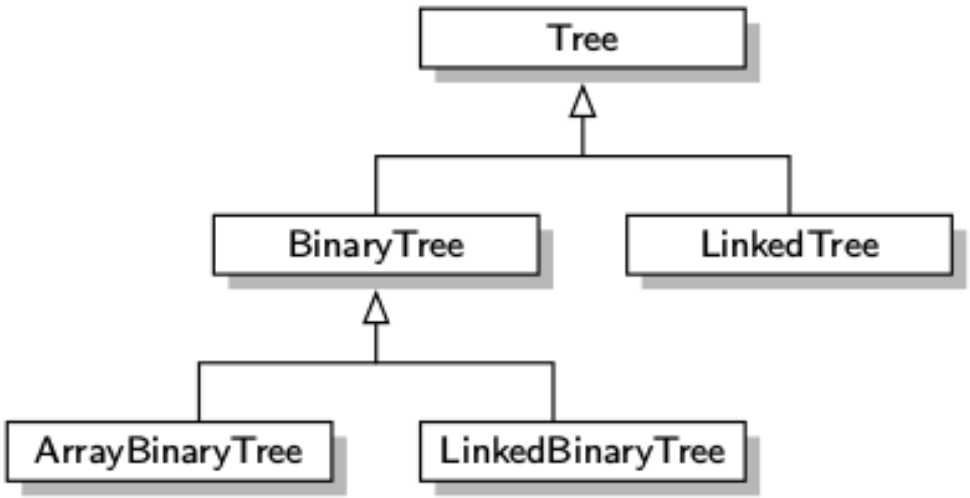
Trees

Example: hierarchy of Python exceptions

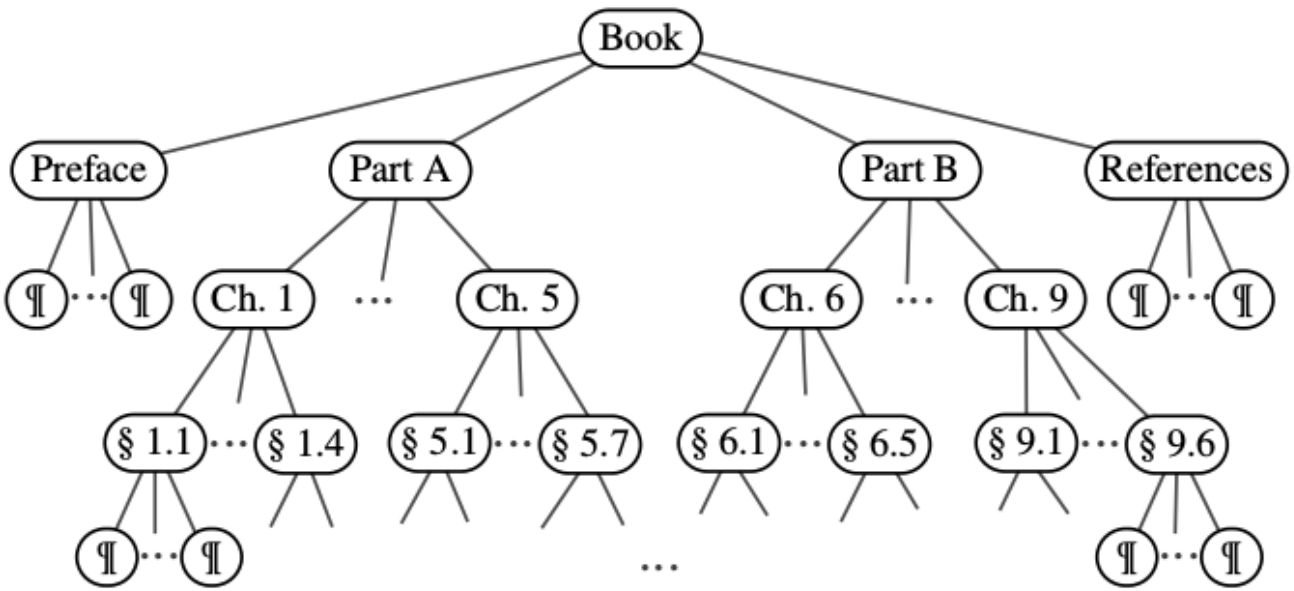


Trees

Example: hierarchy of trees



- A tree is ordered if there is a meaningful linear order among the children of each node
- This means that it is possible to identify the children of a node as being the first, second, third and so on.



Binary trees

Implementation 1/2

```
# node: {'P': None, 'L': None}, 'R': None}

def add_root(e):
    tree = dict()
    tree[e] = {'P': None, 'L: None', 'R': None}
    return tree

def add_child(tree, node, e, side):
    if node in tree and not tree[node][side]:
        tree[node][side] = e
        tree[e] = {'P': node, 'L': None, 'R': None}
```

Binary trees

Implementation 2/2

```
if __name__ == '__main__':
    mytree = add_root('root')
    add_child(mytree, 'root', 'lft1', 'L')
    add_child(mytree, 'root', 'lft2', 'L')
    add_child(mytree, 'root', 'rgt1', 'R')
    add_child(mytree, 'rgt1', 'rgt2', 'R')
    for k,v in mytree.items():
        print(k, "\t",v)
```

- The **depth** of a node is the number of ancestors of the node, excluding the node itself
- The depth of a node can be defined recursively:
  - If the node is the root of the tree, its depth is zero
  - Otherwise, the depth of the node is one plus the depth of the parent of the node

Trees

Computing Depth

```
def is_root(tree,node):
    return tree[node]['P'] == None

def depth(tree,start):
    if is_root(tree,start):
        return 1
    else:
        return 1 + depth(tree, parent(tree, start))
```

- The computational complexity of `depth(tree, start)` is $O(d_{start} + 1)$ because the algorithm performs a constant-time recursive step for each ancestor of `start`
- Thus, the algorithm runs in $O(n)$ worst-case time, where n is the total number of positions of the tree, because a position of the tree may have depth $n - 1$ if all nodes form a single branch

- The **height** of a node can be defined recursively:
  - If the node is a leaf, its height is zero
  - Otherwise, the height of a node is one more than the maximum of the heights of the children of the node
- The height of a tree is equal to the height of its root

Trees: altezza

```
def height(tree, node):
    if node is None:
        return False
    if node not in tree:
        print("The node is not contained in the tree")
        return False
    if is_leaf(tree,node):
        return 0
    else:
        return 1 + max(height(tree,c) for c in children(tree,
            node))
```

Binary Trees

A **binary tree** is an ordered tree with the following properties:

- Every node has at most two children
- Each child node is labelled as being either a **left child** or a **right child**
- A left child precedes a right child in the order of the children of a node

Binary Trees

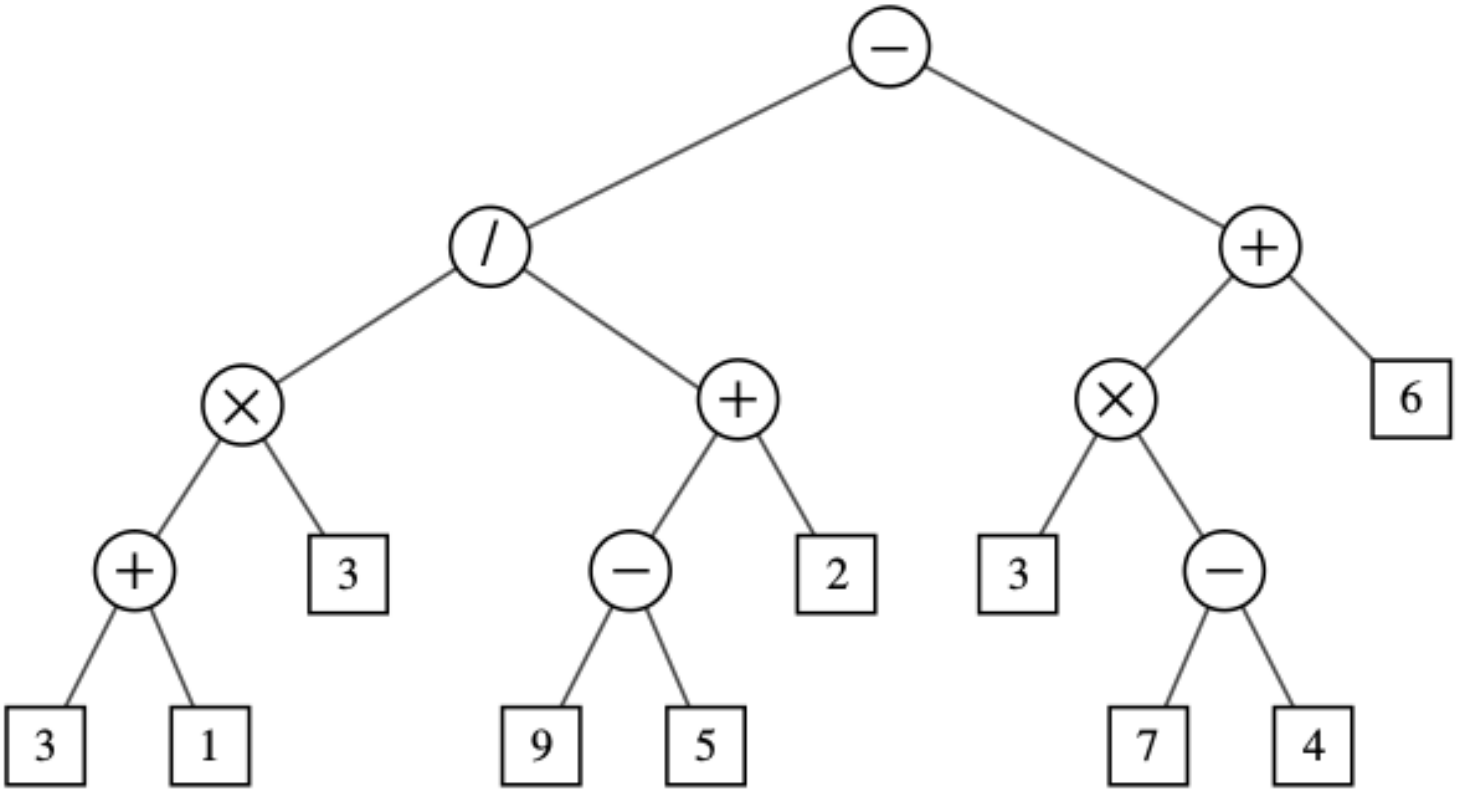
- The subtree rooted at a left (right) child of an internal node v is called a **left (right)subtree** of v
- A binary tree is **proper** if each node has either zero or two children
- In a proper binary tree, an internal node has precisely two children nodes
- A binary tree that is not proper is **improper**

Binary Trees

Decision Tree

Binary Trees

Arithmetic expression



$$\frac{(3+1)\times 3}{(9-5)+2} - ((3\times(7-4))+6)$$

Binary Trees

Recursive definition

A binary tree T is either empy or consists of:

- A node r, called the root of T, that stores an element
- A binary tree (possibly empty), called the left subtree of T
- A binary tree (possibly empty), called the right subtree of T

A binary tree is a specialization of a tree that supports three additional methods:

`left()`	Returns the left child of a given node, or `None` if the given node has not a left child
`right()`	Returns the right child of a given node, or `None` if the given node has not a right child
`sibling()`	Return the sibling of the given node, or `None` if the given node has no sibling

Binary Trees

Implementation 1/3

```
def add_root(e):
    tree = dict()
    tree[e] = {'P':None, 'L':None, 'R':None}
    return tree

def add_child(tree, node, e, side):
    if node in tree and not tree[node][side]:
        tree[node][side] = e
        tree[e] = {'P': node, 'L':None, 'R':None}
    return tree
```

Binary Trees

Implementation 2/3

```
def is_root(tree,node):
    return tree[node]['P'] == None

def depth(tree,start):
    if is_root(tree,start):
        return 1
    else:
        return 1 + depth(tree,tree[start]['P'])

def is_leaf(tree, node):
    return not (tree[node]['L'] or tree[node]['R'])
```

binarytree.py

Binary Trees

Implementation 3/3

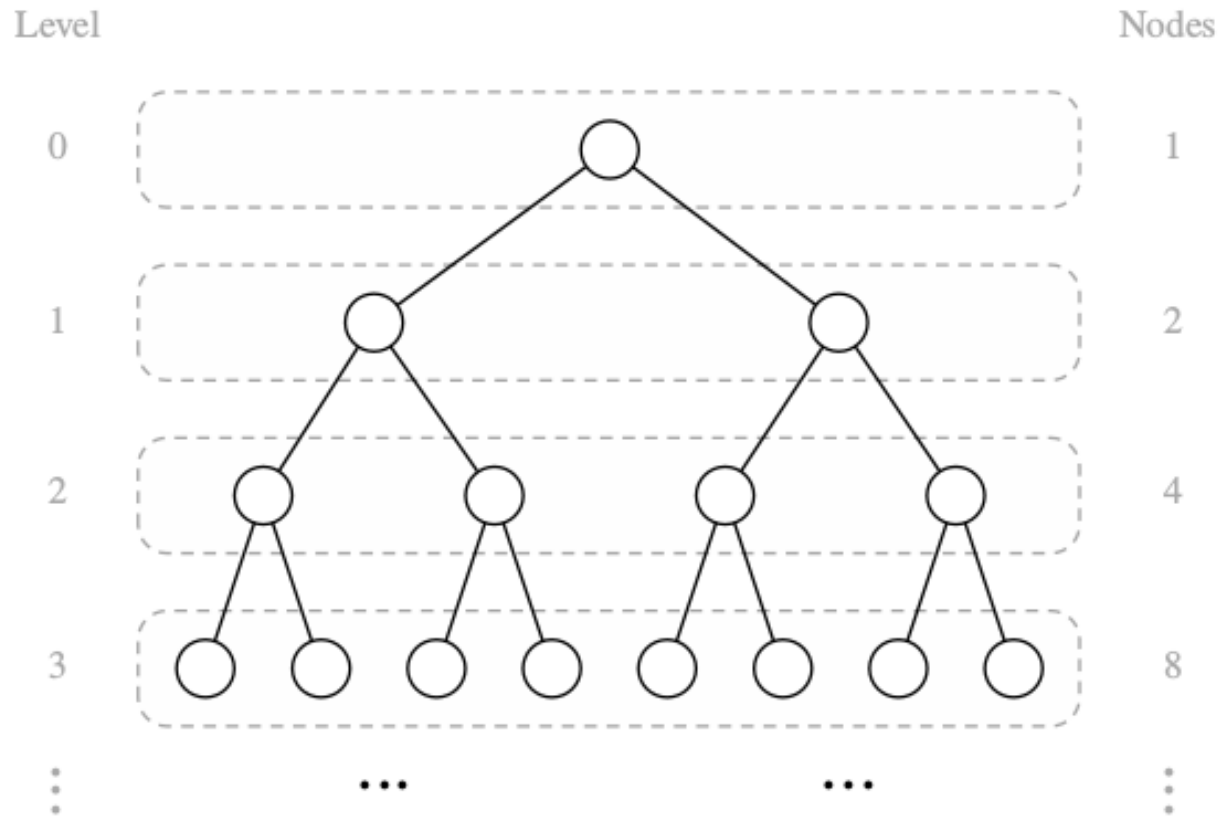
```
def sibling(tree, node):
    parent = tree[node]['P']
    if parent is None:
        return None
    else:
        if node == tree[parent]['L']:
            return tree[parent]['R']
        else:
            return tree[parent]['L']
```

binarytree.py

- Binary trees have some interesting properties dealing with relationships between their heights nand number of nodes
- The set of all nodes of a tree T at the same depth d is called **level** d of T
- In a binary tree, level 0 has at most one node (the root), level 1 has at most two nodes, level 2 has at most 4 nodes, and so on
- In general, level d has at most 2^d nodes. Thus, the maximum number of nodes on the levels of a binary tree grows exponentially

Binary Trees

Properties



Let T be a nonempty binary tree, and let n (number of nodes), n_E (number of external nodes), N_I (number of internal nodes)and h (height) of T. Then:

1. $h + 1 \leq n \leq 2^{h+1} - 1$
2. $1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h - 1$
4. $\log(n + 1) - 1 \leq h \leq n - 1$

If T is proper, then T has also the following properties:

1. $2h + 1 \leq n \leq 2^{h+1} - 1$
2. $h + 1 \leq n_E \leq 2^h$
3. $h \leq n_I \leq 2^h - 1$
4. $\log(n + 1) - 1 \leq h \leq (n - 1)/2$

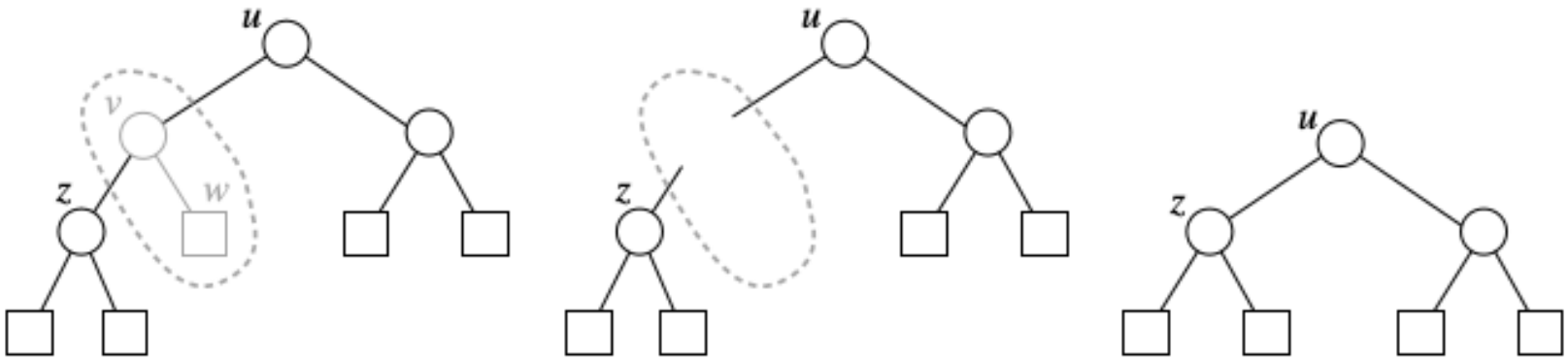
Proposition: In a nonempty proper binary tree T, with n_E external nodes and n_I internal nodes, we have $n_E = n_I + 1$

Two cases can be considered:

- If T has only one node v, then $n_I = 0$. Tus the proposition is justified.

If T has more than one node:

- It is possible to remove an external arbitrary node w with its parent v, which is an internal node
- If v has a parent u, u can be reconnected with the former sibling of w. This operation removes an internal node and an external node, and leaves the tree a proper binary tree

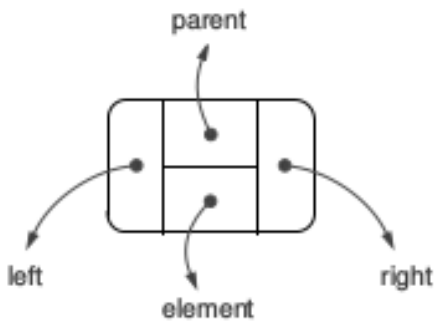


- This operation can be repeated until the final tree will consist of a single node, which will eventually be an external node
- Thus, the proposition is justified

Binary Trees

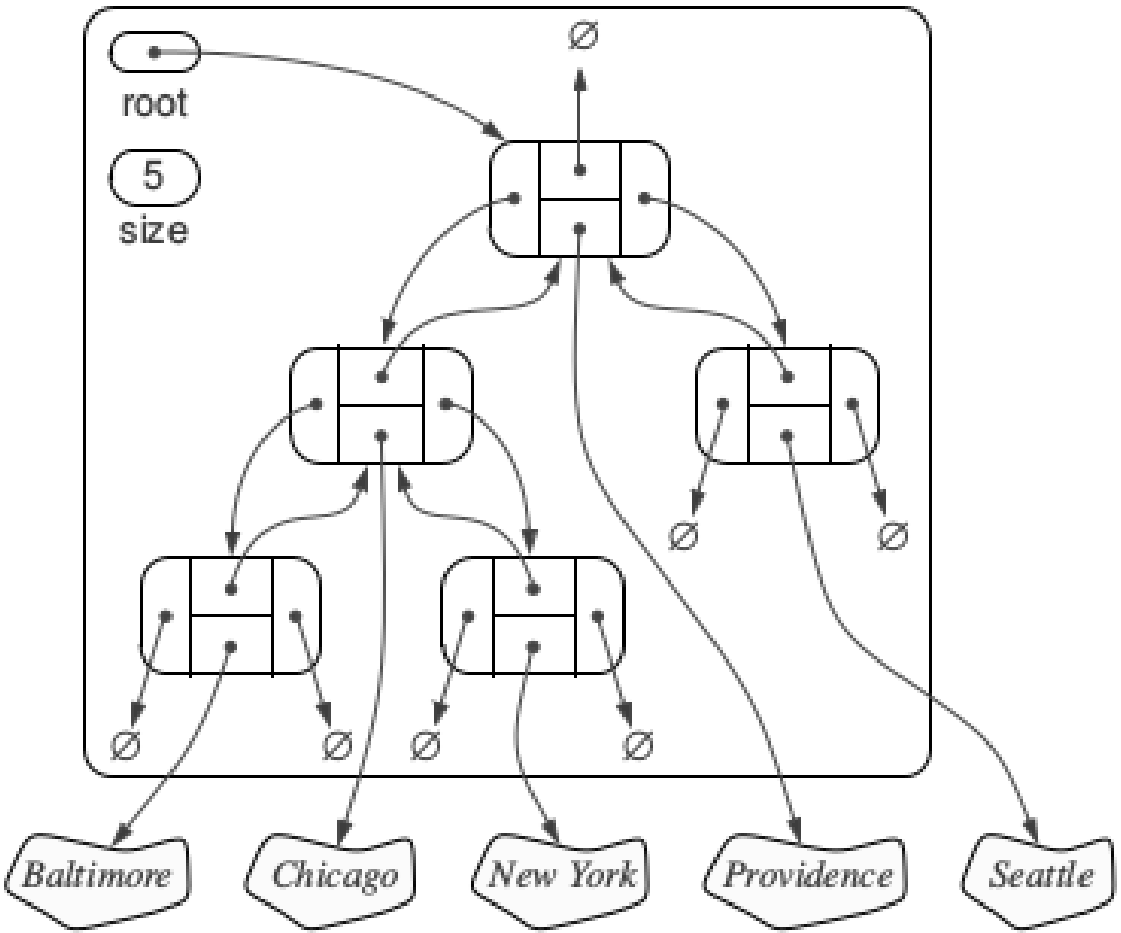
Implementation

- A binary tree T can be implemented using a linked structure
- Every node maintains references to the element stored in a certain position, the parent node and the children nodes
- If p is the root of the tree, the parent field of p is `None`

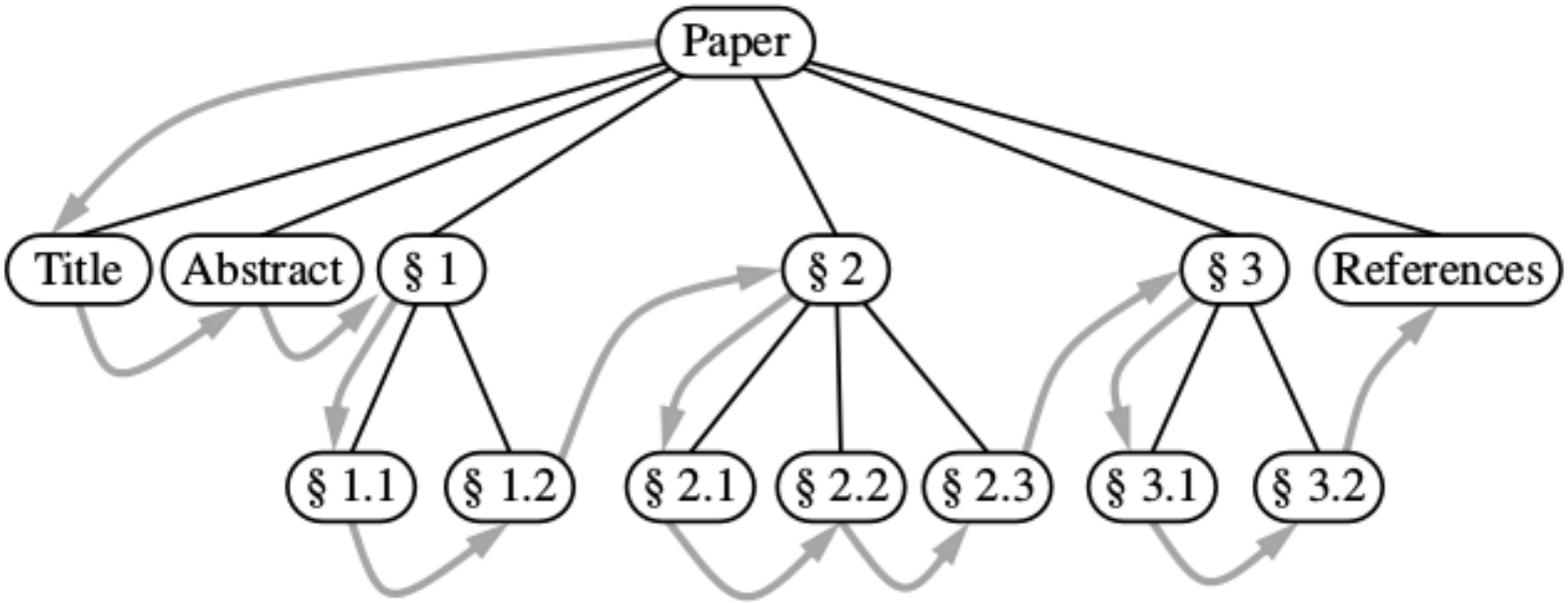


Binary Trees

Implementation



Trees: preorder traversal

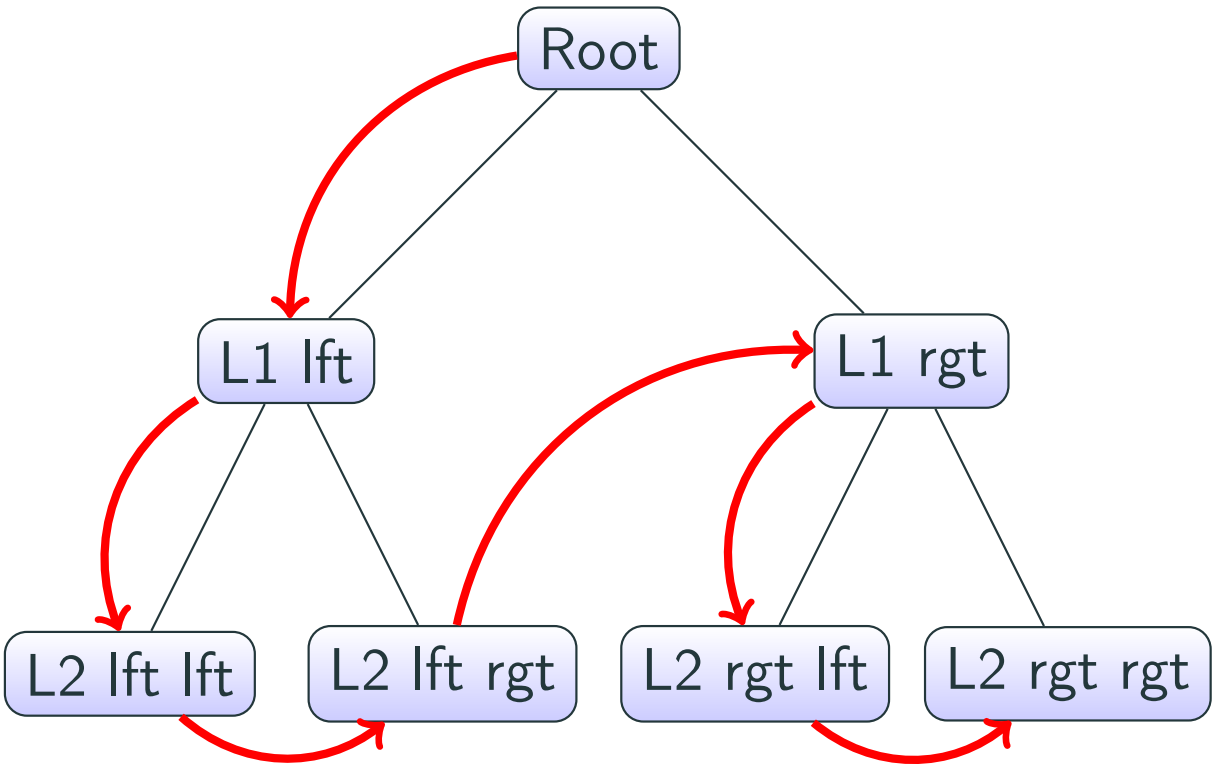


A **traversal** of a tree T is a systematic way of accessing all the positions of T

- In a **preorder traversal** of a tree T, the root of T is visited first and then the subtrees rooted at its children are traversed recursively
- If the tree is ordered, then the subtrees are visited according to the order of the children

Trees

Preorder traversal




```
root node
level 1 lft
level 2 lft lft
level 2 lft lft
level 1 rgt
level 2 rgt lft
level 2 rgt rgt
```

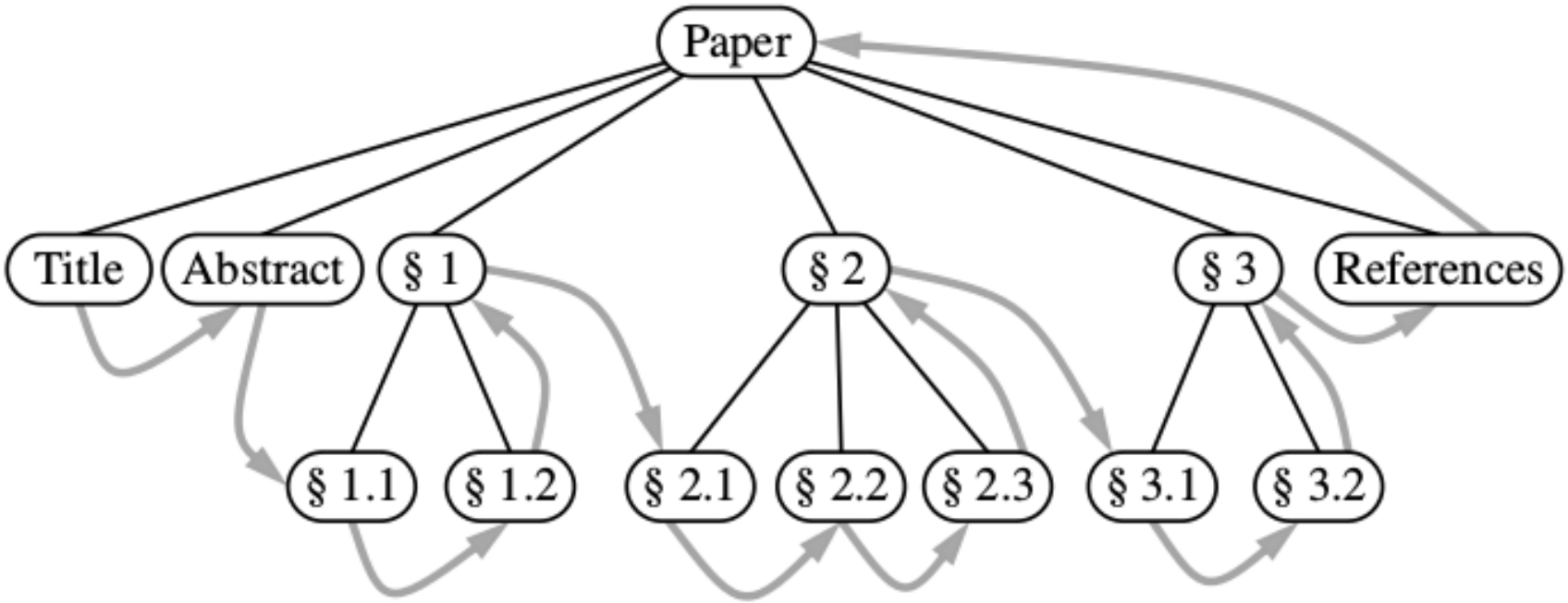
Trees

Preorder traversal

```
def preorder(tree,node,visited):
    if node != None:
        visited.append(node)
        lc = tree[node]['L']
        rc = tree[node]['R']
        if lc != None or rc != None:
            preorder(tree, lc, visited)
            preorder(tree, rc, visited)
        return visited
```

Trees

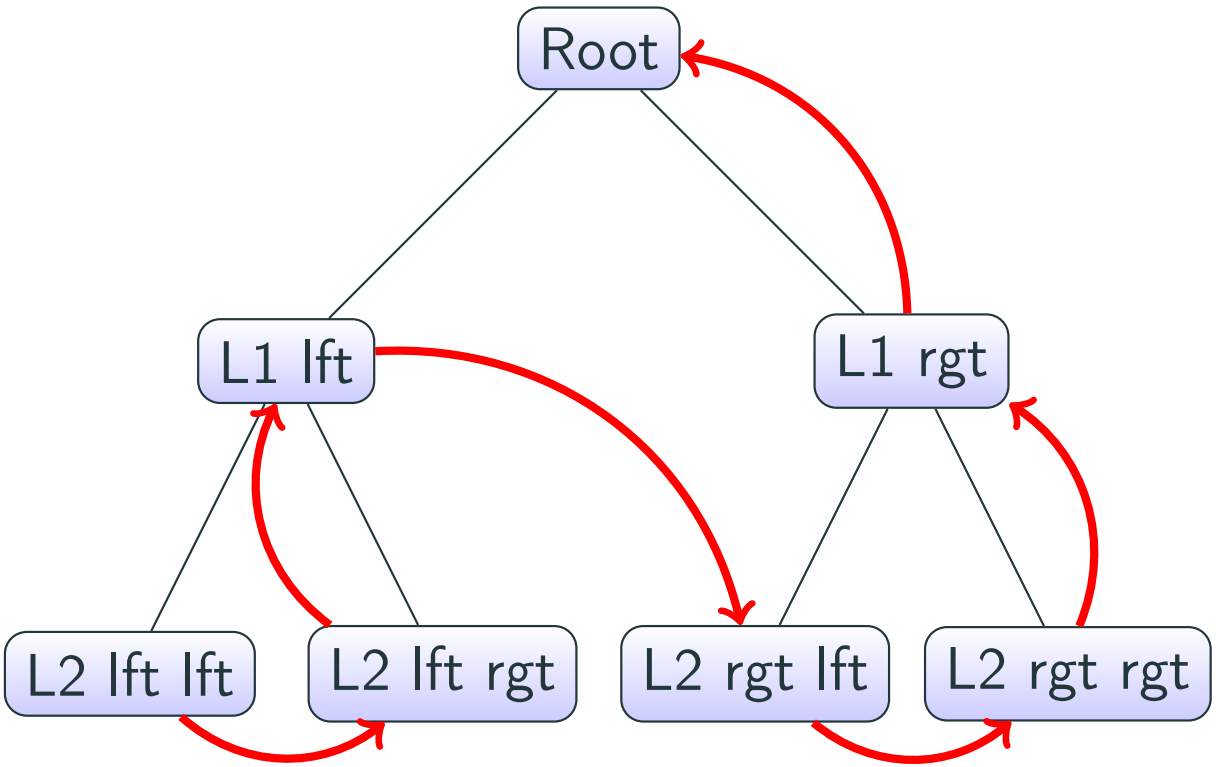
Postorder traversal



- In the **postorder traversal** the subtrees rooted at the children of the root are traversed first. Then, the root is visited.
- If the tree is ordered, then the subtrees are visited according to the order of the children

Trees

Postorder traversal




```
level 2 lft lft
level 2 lft rgt
level 1 lft
level 2 rgt lft
level 2 rgt rgt
level 1 rgt
root node
```

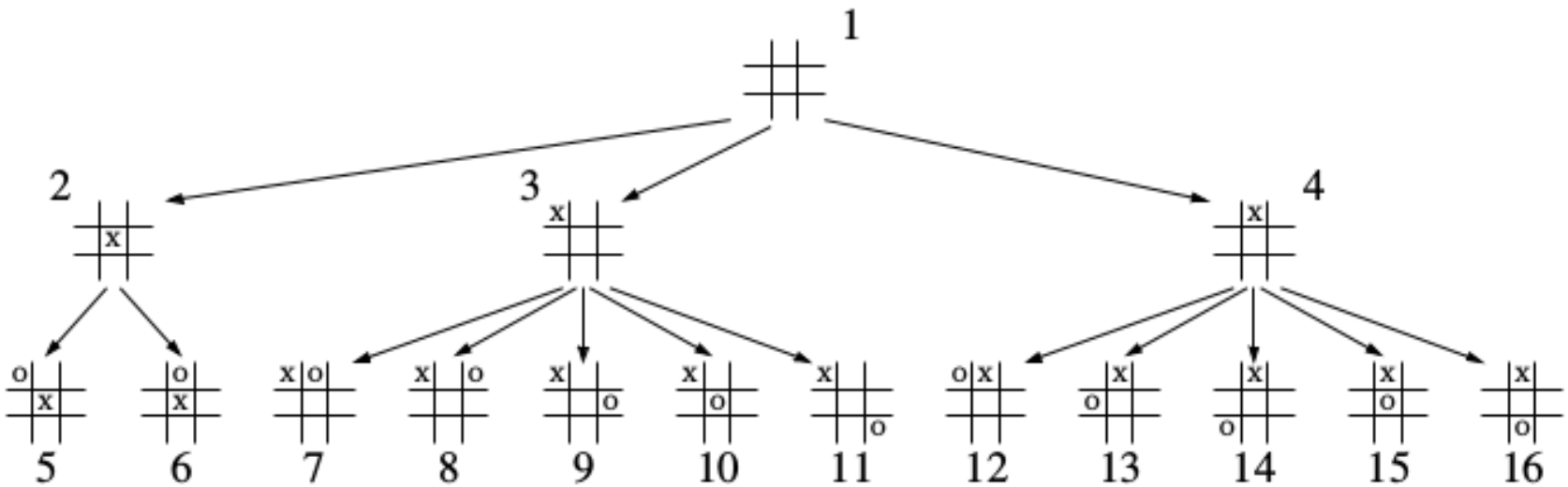
Trees

Postorder traversal

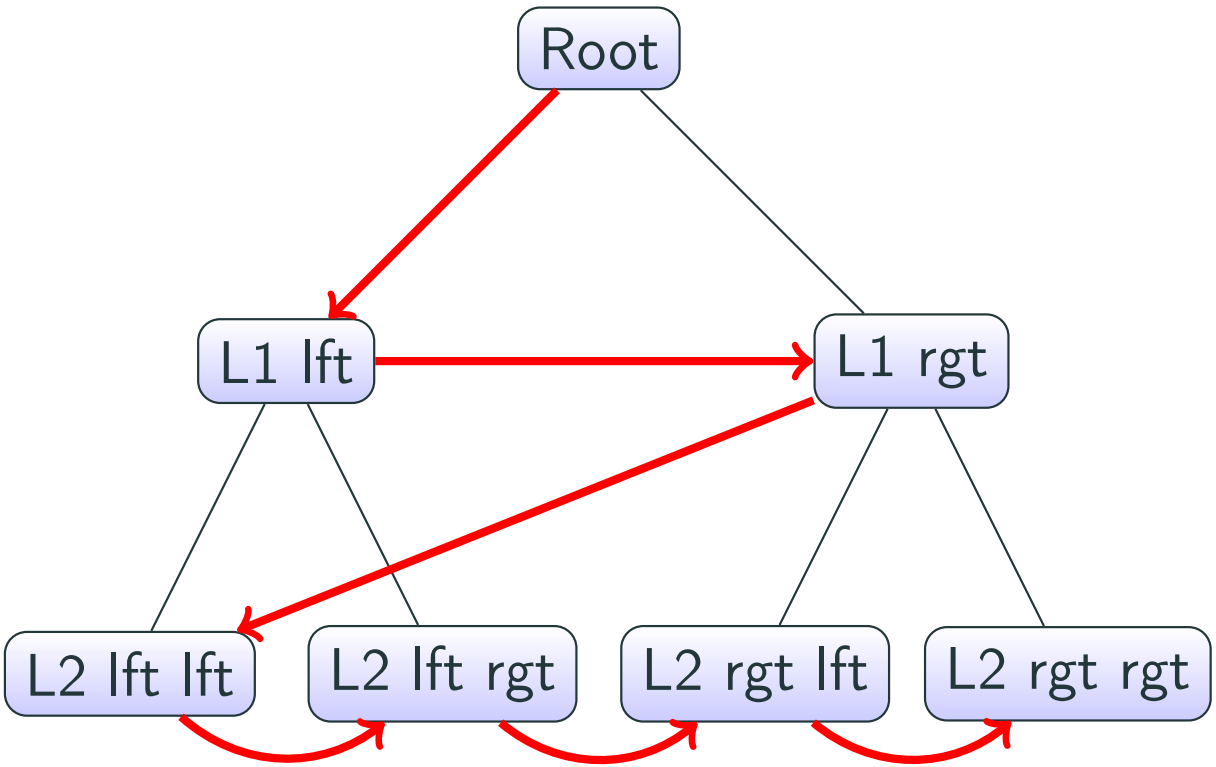
```
def postorder(tree,node,visited):
    if node != None:
        lc = tree[node]['L']
        rc = tree[node]['R']
        if lc != None or rc != None:
            postorder(tree, lc, visited)
            postorder(tree, rc, visited)
        visited.append(node)
        return visited
```

- Both preorder and postorder traversal algorithms are efficient ways to access all the positions of a tree
- At each position p, the nonrecursive part of the traversal algorithm requires time $O(c_p + 1)$ where c_p is the number of children of p, under the assumption that the visit itself takes $O(1)$ time
- The overall computing complexity for the traversal of tree T is $O(n)$, where n is the number of positions in the tree

- Another approach is to traverse a tree so that all the positions at depth d before visiting the positions at depth $d + 1$
- This algorithm is called **breadth-first traversal**



Trees binari: breadth first traversal

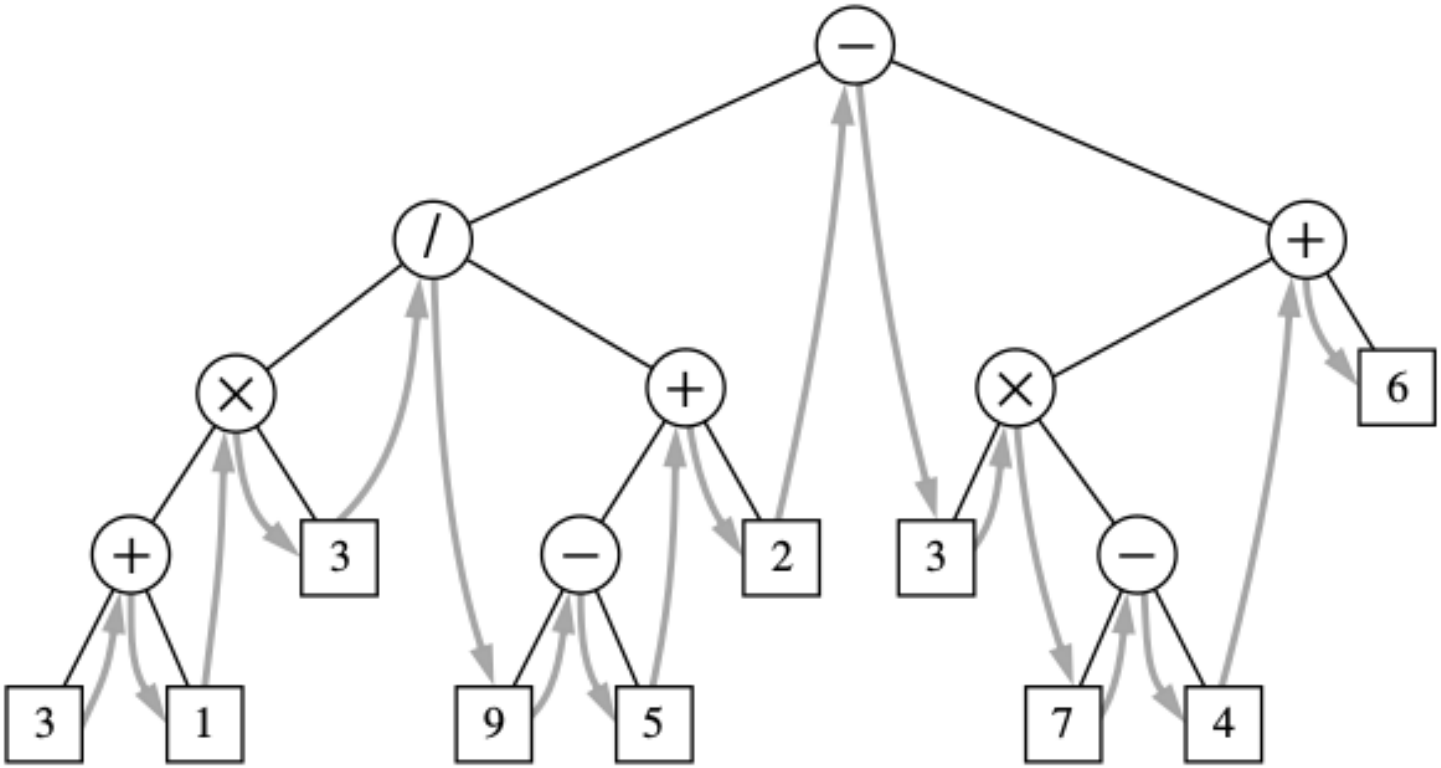



```
root node
level 1 lft
level 1 rgt
level 2 lft lft
level 2 lft rgt
level 2 rgt lft
level 2 rgt rgt
```

Trees binari: breadth first traversal

```
def bfs(tree,node):
    "Breadth First Search"
    queue,visited = [],[]
    if node == root(tree):
        queue.append(node)
    while queue:
        current = queue.pop(0)
        visited.append(current)
        if tree[current]['L']:
            queue.append(tree[current]['L'])
        if tree[current]['R']:
            queue.append(tree[current]['R'])
    return visited
```

Trees: inorder traversal



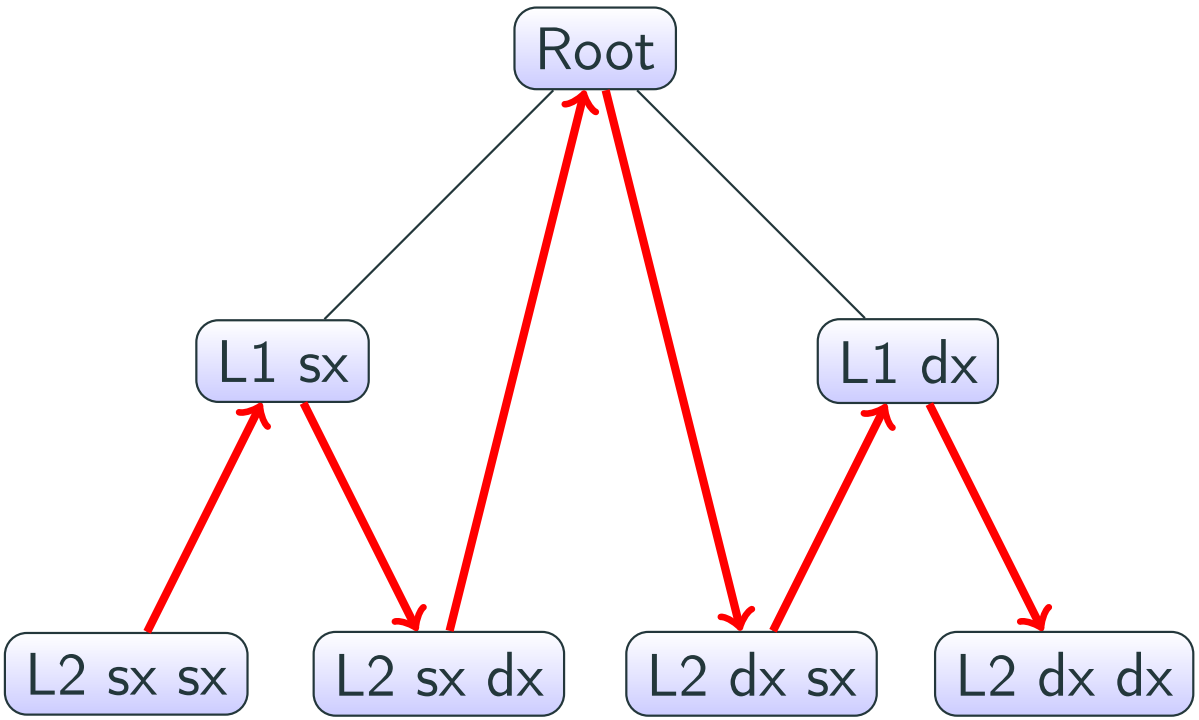
Trees: inorder traversal

- The **inorder traversal** is peculiar of binary trees
- Each position is visited after the recursive visit of its left subtree and before the recursive visit of it right subtree

Trees: inorder traversal

```
def inorder(tree,node,visited):
    if node != None:
        print(node,visited)
        lc = tree[node]['L']
        rc = tree[node]['R']
        if lc != None:
            inorder(tree, lc, visited)
        visited.append(node)
        if rc != None:
            inorder(tree, rc, visited)
        return visited
```

Trees: inorder traversal




```
level 2 sx sx
level 1 sx
level 2 sx dx
root node
level 2 dx sx
level 1 dx
level 2 dx dx
```

Section 9

Binary Search Trees

Binary Search Trees

- Search trees can be used for operations on dynamic collections (searching, priority queues
- Basic operations have a computational complexity proportional to the height of the tree
- Given a complete binary tree composed with n nodes, operations have a computational complexity $\Theta(lg\, n)$ in the worst case
- If the tree has a linear structure, the same operations have a computational complexity $\Theta(n)$

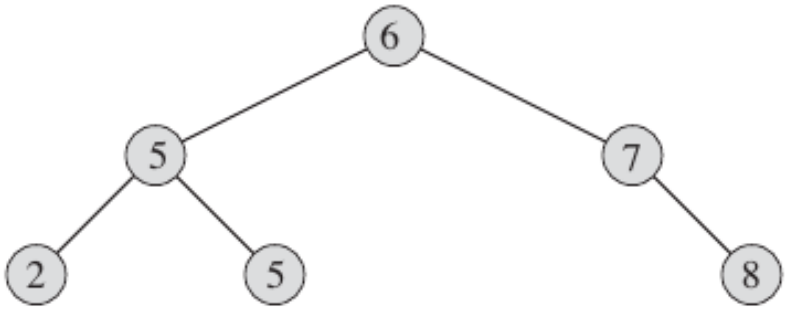
Binary Search Trees

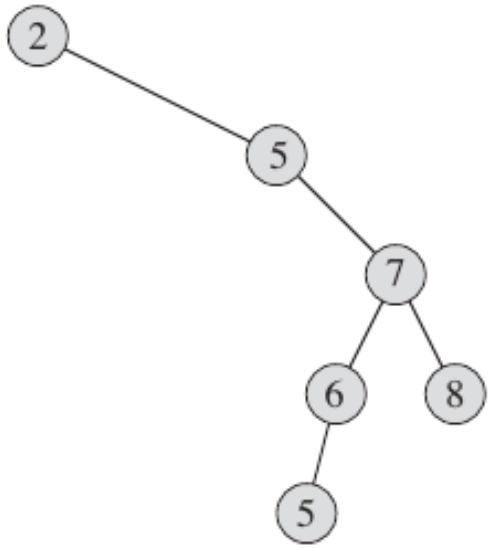
- A **Binary Search Tree (BST)** can be represented with a linked list whose nodes are objects
- Every node has the attributes **key, left, right, p** (key, left child, right child, parent node)

Properties of the BST

Let x a node in a binary search tree. If y is a node of the left subtree of x then $y.key \leq x.key$. If y is a node of the right subtree of x then $y.key \geq x.key$

Binary Search Trees





Binary Search Trees

The keys of a binary search tree can be orderly listed using a recursive algorithm consisting in a symmetric traversal of the tree

The function **inorder** has a computational complexity $\Theta(n)$

Binary Search Trees

```
def inorder(tree,node,visited):
    if node != None:
        print(node,visited)
        lc = tree[node]['L']
        rc = tree[node]['R']
        if lc != None:
            inorder(tree, lc, visited)
        visited.append(node)
        if rc != None:
            inorder(tree, rc, visited)
        return visited
```

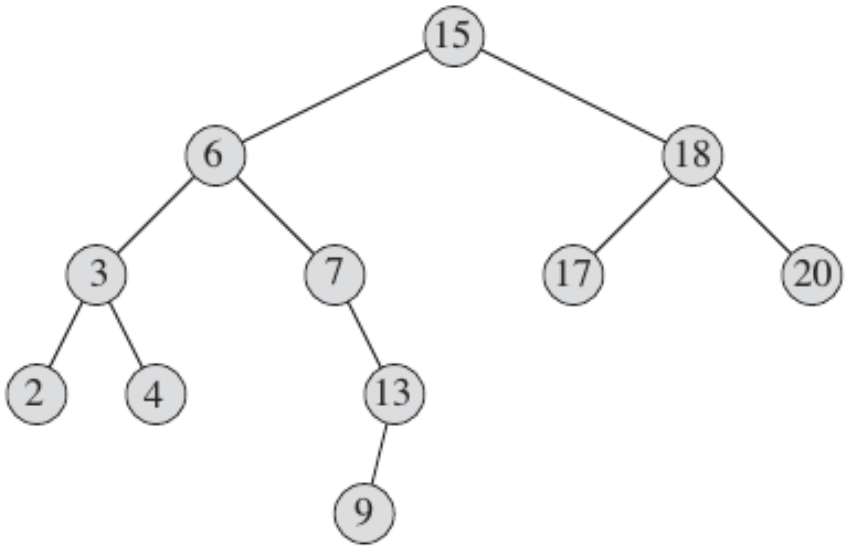
Query

```
def tree_search(tree,root,val):
    if root == None:
        print("Non existent node")
        return root
    if root == val:
        print("Found")
        return root
    if val < root:
        return tree_search(tree,tree[root]['L'],val)
    else:
        return tree_search(tree,tree[root]['R'],val)
```

- The computational complexity of `tree_search` is $O(h)$, h being the height of the tree

Binary Search Trees

Query



Binary Search Trees

Query: iterative implementation

```
def iterative_tree_search(tree,root,val):
    while root is not None and root != val:
        if val < root:
            root = tree[root]['L']
        else:
            root = tree[root]['R']
    return root
```

- The computational complexity of `tree_search` is $O(h)$, h being the height of the tree

Binary Search Trees

Searching for min/max

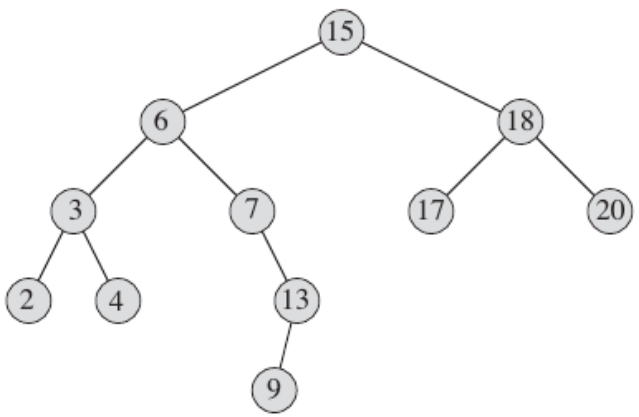
```
def tree_min(tree,root):
    while tree[root]['L'] is not None:
        root = tree[root]['L']
    return root
```

```
def tree_max(tree,root):
    while tree[root]['R'] is not None:
        root = tree[root]['R']
    return root
```

Binary Search Trees

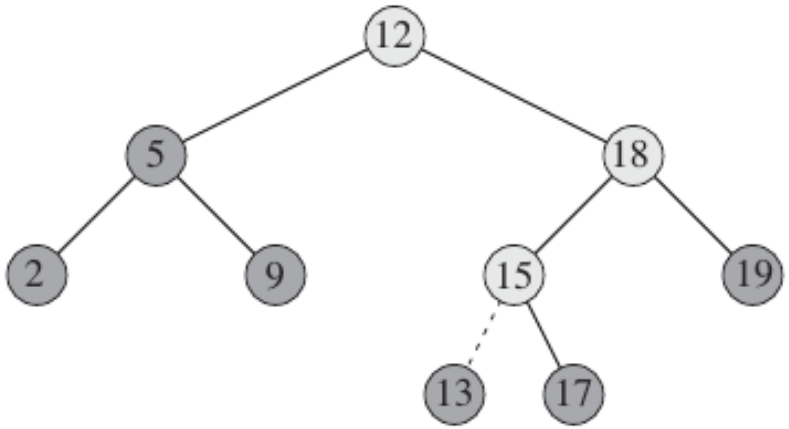
Searching for next/previous

```
def tree_successor(tree,node):
    if tree[node]['R'] is not None:
        return tree_min(tree,tree[node]['R'])
    par = tree[node]['P']
    while par is not None and node == tree[par]['R']:
        node = par
        par = tree[par]['P']
    return par
```



Binary Search Trees

Inserting an element



Binary Search Trees

Inserting an element

```
def tree_insert(tree,node):
    y = None
    x = root(tree)
    while x is not None:
        y = x
        if node < x:
            x = tree[x]['L']
        else:
            x = tree[x]['R']
    tree[node] = {'P': y, 'L':None, 'L':None}
    if y is not None and node < y:
        tree[y]['L'] = node
    else:
        tree[y]['R'] = node
```

While deleting a node z three distinct cases can occur:

- z has no children. In this case the parent node of z must be modified after deletion
- If the node z has one child only, the child node will replace the parent node
- Se il nodo z ha due figli, si trova il successore y di z (nel sottoalbero destro di z) e lo sostituisce a z

Lo spostamento di un sottoalbero viene eseguito mediante la funzione `transplant`

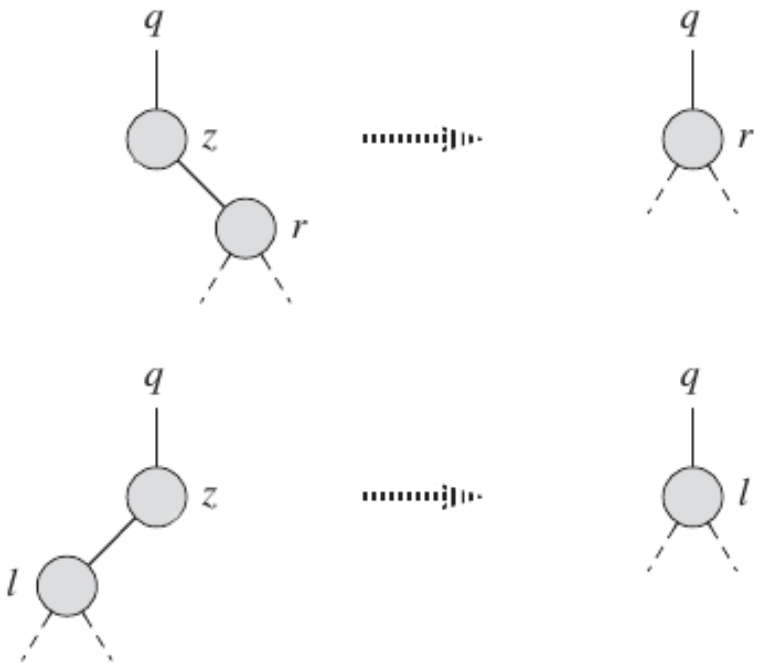
Binary Search Trees

Deleting an element

```
def transplant(tree,old,new):
    oldparent = tree[old]['P']
    if oldparent == None:
        tree[new]['P'] = None
        if tree[old]['L']:
            tree[new]['L'] = tree[old]['L']
        if tree[old]['R']:
            tree[new]['R'] = tree[old]['R']
    elif old == tree[oldparent]['L']:
        tree[oldparent]['L'] = new
    else:
        tree[oldparent]['R'] = new
    if new != None:
        tree[new]['P'] = tree[old]['P']
    del tree[old]
```

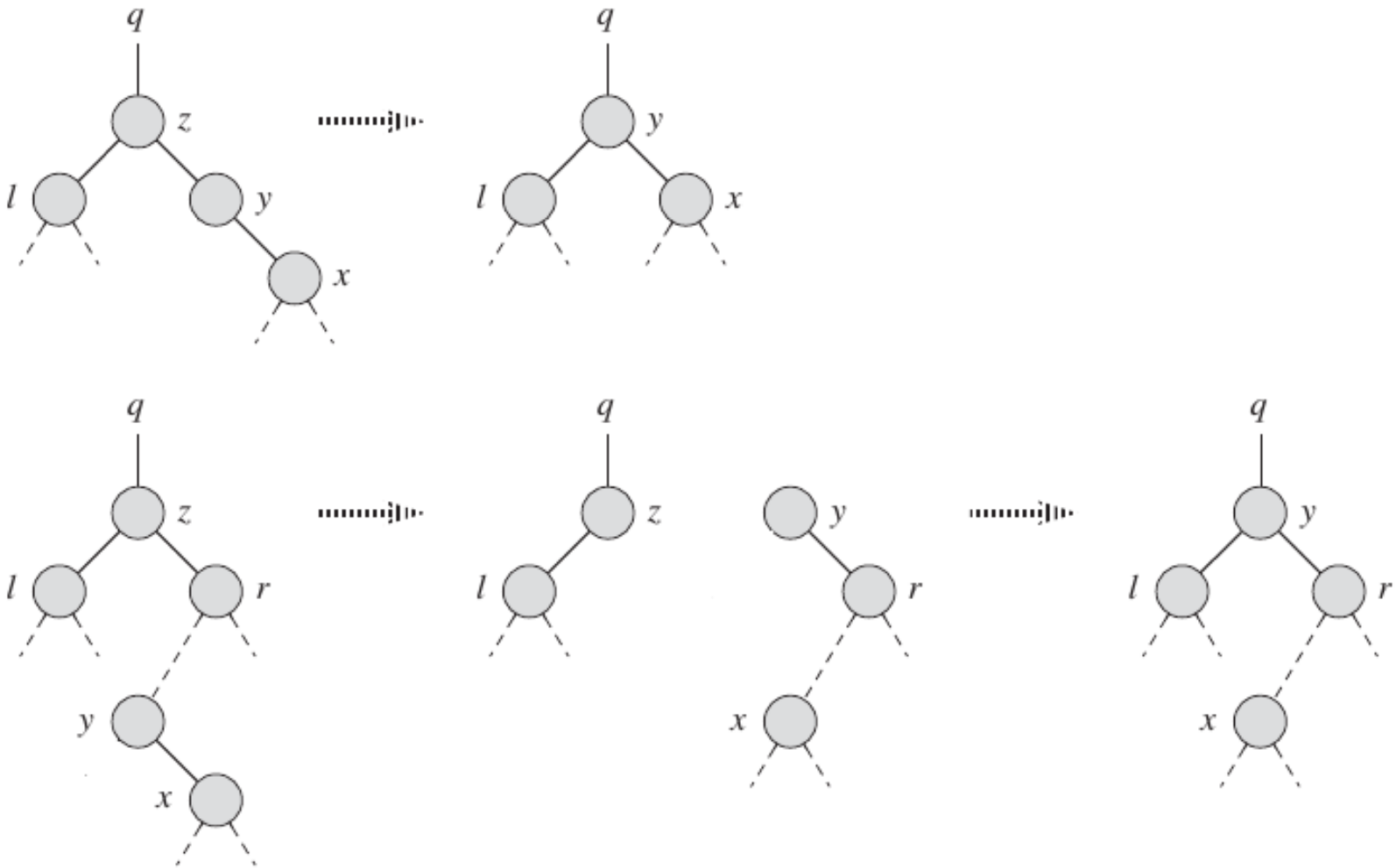
Binary Search Trees

Deleting an element



Binary Search Trees

Deleting an element



Binary Search Trees

Deleting an element

```
def tree-delete(T, z):
  if z.left is None:
    transplant(T, z, z.right)
  elif z.right is None:
    transplant(T, z, z.left)
  else:
    y = tree-minimum(z.right)
    if y.p != z:
      transplant(T, y, y.right)
      y.right = z.right
      (y.right).p = y
    transplant(T, z, y)
    y.left = z.left
    (y.left).p = y
```

Section 10

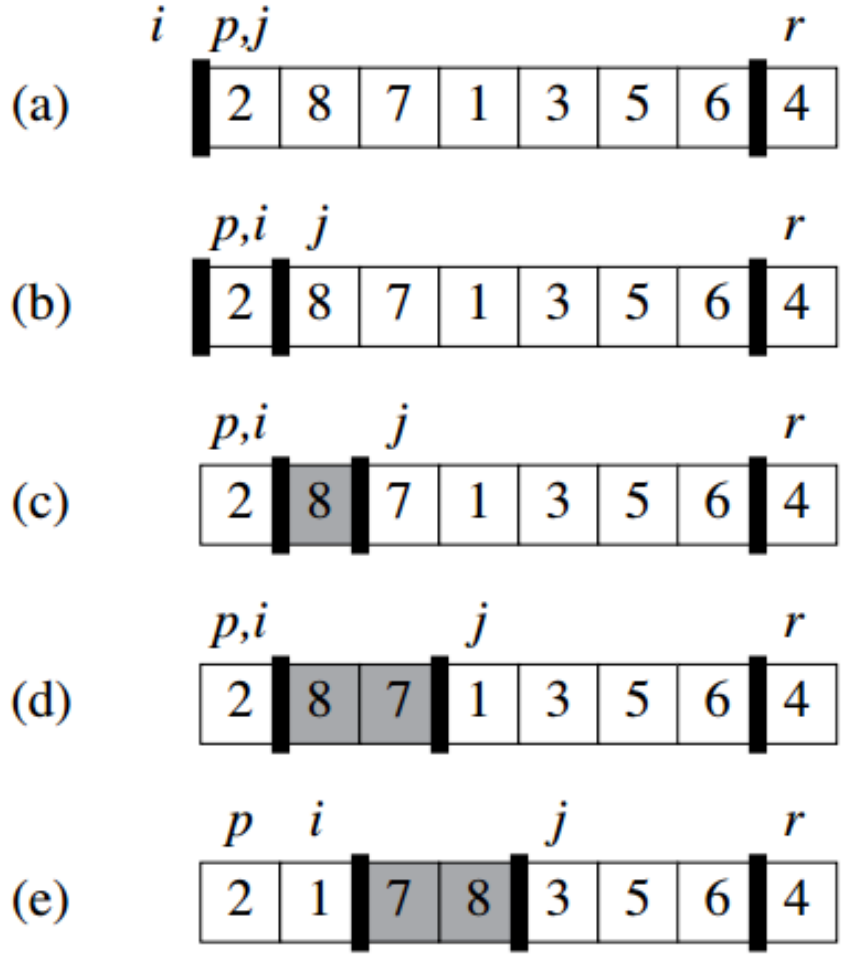
Quicksort

Quicksort

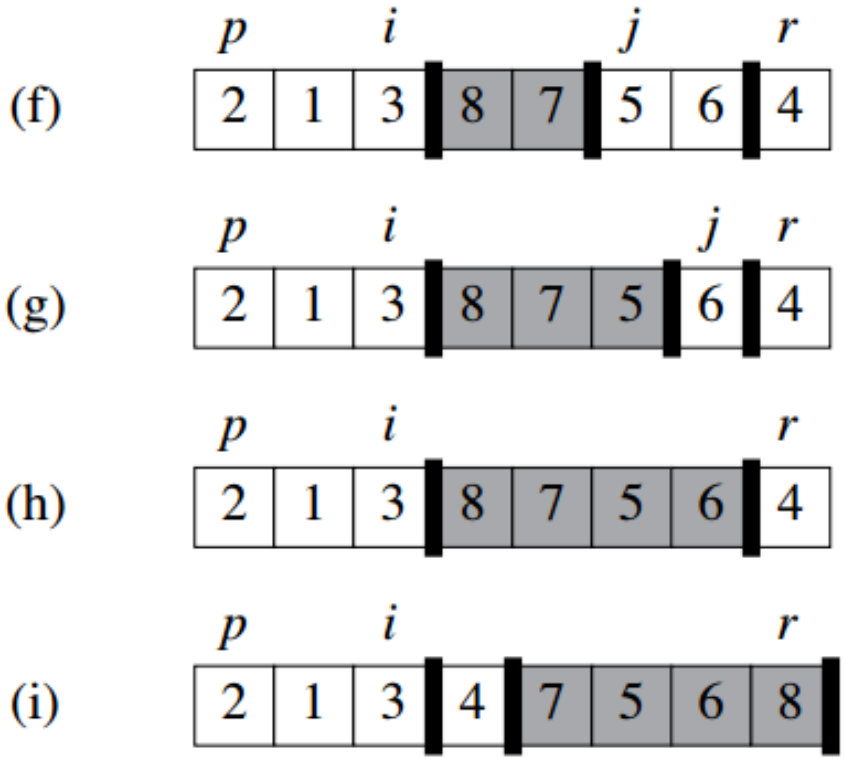
Like mergesort, quicksort relies on the *divide and conquer* paradigm

- **Divide**: the array to be sorted `data[start:end]` is divided in two subarrays `data[start:p-1]` containing elements not larger than `data[p]`, and `data[p+1:end]` whose elements are not smaller than `data[p]`
- **Conquer**: the two subarrays are recursively sorted calling `quicksort`
- **Combine**: the subarrays are already sorted and no activity is needed to combine data as the array is already sorted

Quicksort: how it works 1/2



Quicksort: how it works 2/2



Quicksort

```
def quicksort(data, start, end):
  if start < end:
    ipivot = partition(data, start, end)
    quicksort(data, start, ipivot - 1)
    quicksort(data, ipivot + 1, end)
  return data
```

quicksort.py

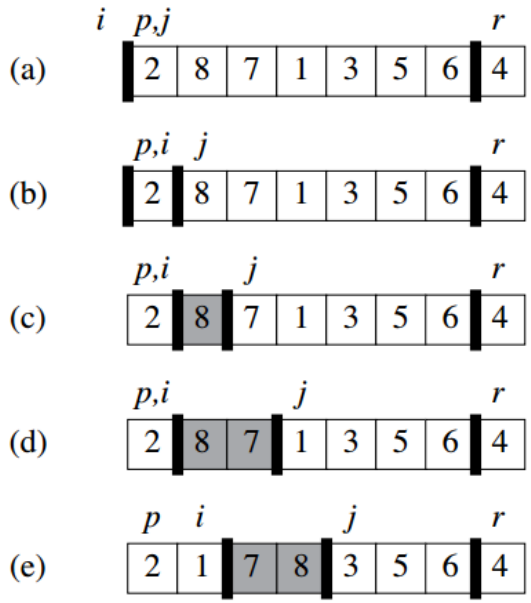
Quicksort

```
def partition(data, start, end):
  x = data[end]
  i = start - 1
  for j in range(start, end):
    if data[j] <= x:
      i = i + 1
      data[i], data[j] = data[j], data[i]
  ipivot = i + 1
  data[ipivot], data[end] = data[end], data[ipivot]
  return ipivot
```

quicksort.py

Quicksort

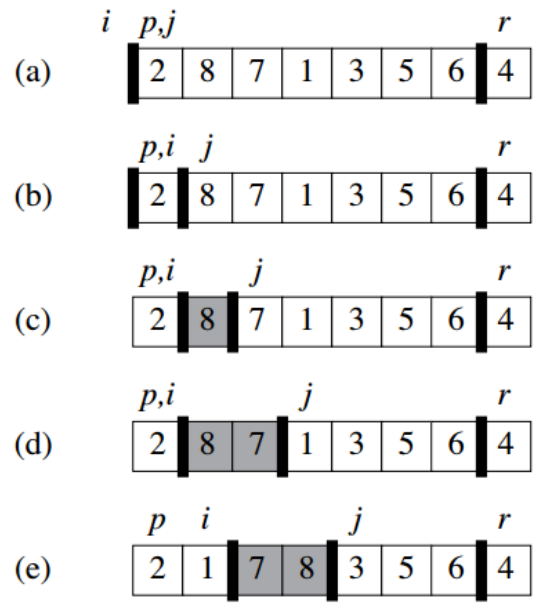
```
def partition(data, start, end):
  x = data[end]
  i = start - 1
  for j in range(start, end):
    if data[j] <= x:
      i = i + 1
      data[i], data[j] = data[j],
          data[i]
  ipivot = i + 1
  data[ipivot], data[end] = data[end
      ], data[ipivot]
  return ipivot
```



- a) Initial condition
- b) The value 2 is placed in the partition containing numbers smaller than `x`

Quicksort

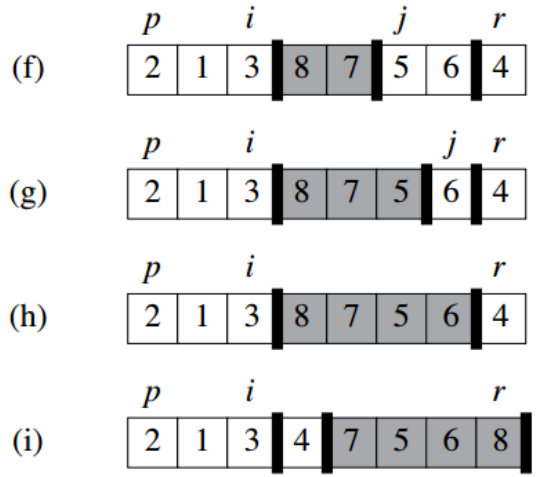
```
def partition(data, start, end):
  x = data[end]
  i = start - 1
  for j in range(start, end):
    if data[j] <= x:
      i = i + 1
      data[i], data[j] = data[j], data[i]
  ipivot = i + 1
  data[ipivot], data[end] = data[end],
      data[ipivot]
  return ipivot
```



- c-d) Values 8 and 7 remain in the partition of numbers larger than [x]
- e) 1 and 8 are swapped

Quicksort

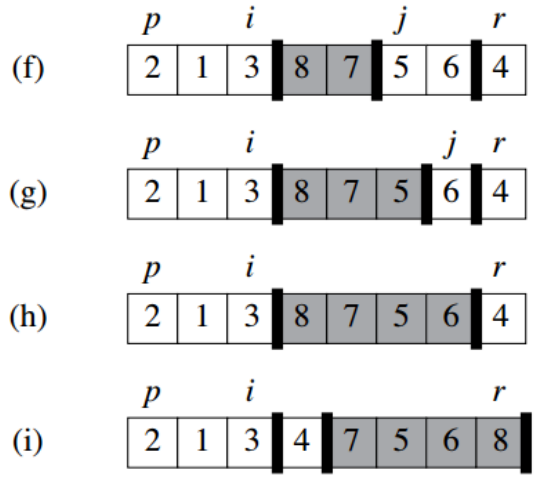
```
def partition(data, start, end):
  x = data[end]
  i = start - 1
  for j in range(start, end):
    if data[j] <= x:
      i = i + 1
      data[i], data[j] = data[j],
          data[i]
  ipivot = i + 1
  data[ipivot], data[end] = data[end
      ], data[ipivot]
  return ipivot
```



- f) 3 and 7 are swapped
- g) 5 is placed in the partition of numbers larger than `x`

Quicksort

```
def partition(data, start, end):
  x = data[end]
  i = start - 1
  for j in range(start, end):
    if data[j] <= x:
      i = i + 1
      data[i], data[j] = data[j],
          data[i]
  ipivot = i + 1
  data[ipivot], data[end] = data[end
      ], data[ipivot]
  return ipivot
```



- h) 5 is placed in the partition of numbers larger than `x`
- i) `x` is placed in t

$$
\begin{aligned}
S_n &= a_1 + a_2 + \cdots + a_{n-1} + a_n \\
q \cdot S_n &= q \cdot a_1 + q \cdot a_2 + \cdots + q \cdot a_{n-2} + q \cdot a_{n-1} + q \cdot a_n = \\
&= a_2 + a_3 + \cdots + a_n + q \cdot a_n \\
q \cdot S_n - S_n &= q \cdot a_n - a_1 \\
S_n \cdot (q-1) &= q \cdot a_n - a_1 = \\
&= q^n \cdot a_1 - a_1 = \\
&= a_1 \cdot (q^n - 1)
\end{aligned}
$$

$$
S_n = a_1 \cdot \frac{q^n - 1}{q - 1}
$$

Quicksort: computational complexity

- The computational complexity of the quicksort algorithm depends on how the partitions are balanced
- If the partition is balanced, quicksort will execute with the same asymptotic complexity of mergesort ($\Theta(nlgn)$)
- If the partition is not balanced, quicksort will execute with the same asymptotic complexity of insertion-sort ($O(n^2)$)

Quicksort: computational complexity (worst case)

- It happens when one of the two subarrays contains $n-1$ elements and the other 0 elements
- If this unbalancement occurs at every recursive call, the computational complexity of the partitioning is $\Theta(n)$
- A recursive call on an empty array has a computational complexity $T(0) = \Theta(1)$. The recurrence equation can be written as:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Quicksort: computational complexity (best case)

- When the partition is balanced, the function **`partition`** produces two subproblems whose dimension is not larger than $n/2$ (one of them $\lfloor n/2 \rfloor$, the other $\lceil n/2 \rceil - 1$)
- This is the best condition. The recurrence equation becomes:

$$T(n) \leq 2T(\frac{n}{2}) + \Theta(n)$$

Whose solution (master theorem, case 2) is:

$$T(n) = O(nlgn)$$

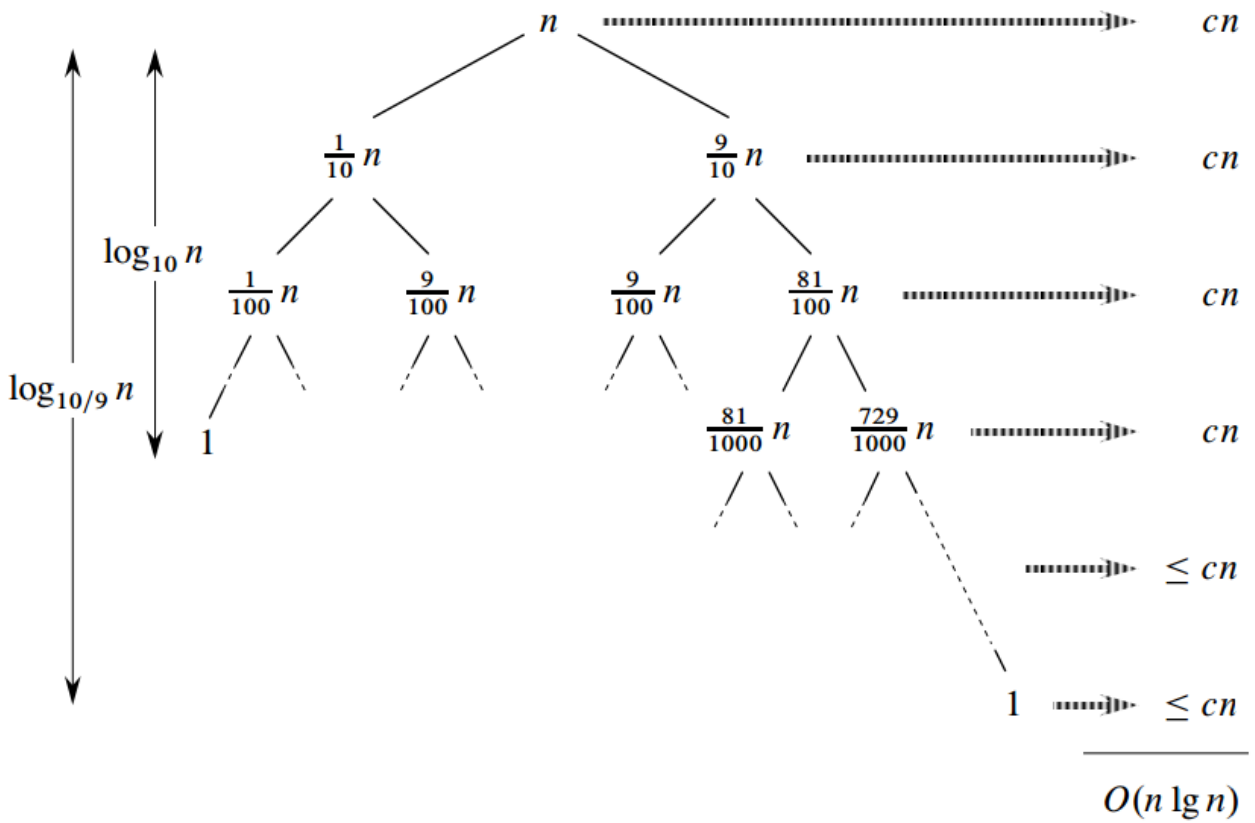
Quicksort: computational complexity (average case)

- It is similar to the best case
- This depends on the fact that the performance of the algorithm do not depend on the balance of partition. They depend on how constant is the balance at every recursive call
- Suppose, for example, that `partition` always produces a partition proportional to 9 and 1
- In this case:

$$T(n) \leq T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + cn$$

Quicksort: computational complexity (average case)

$$T(n) \leq T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + cn$$

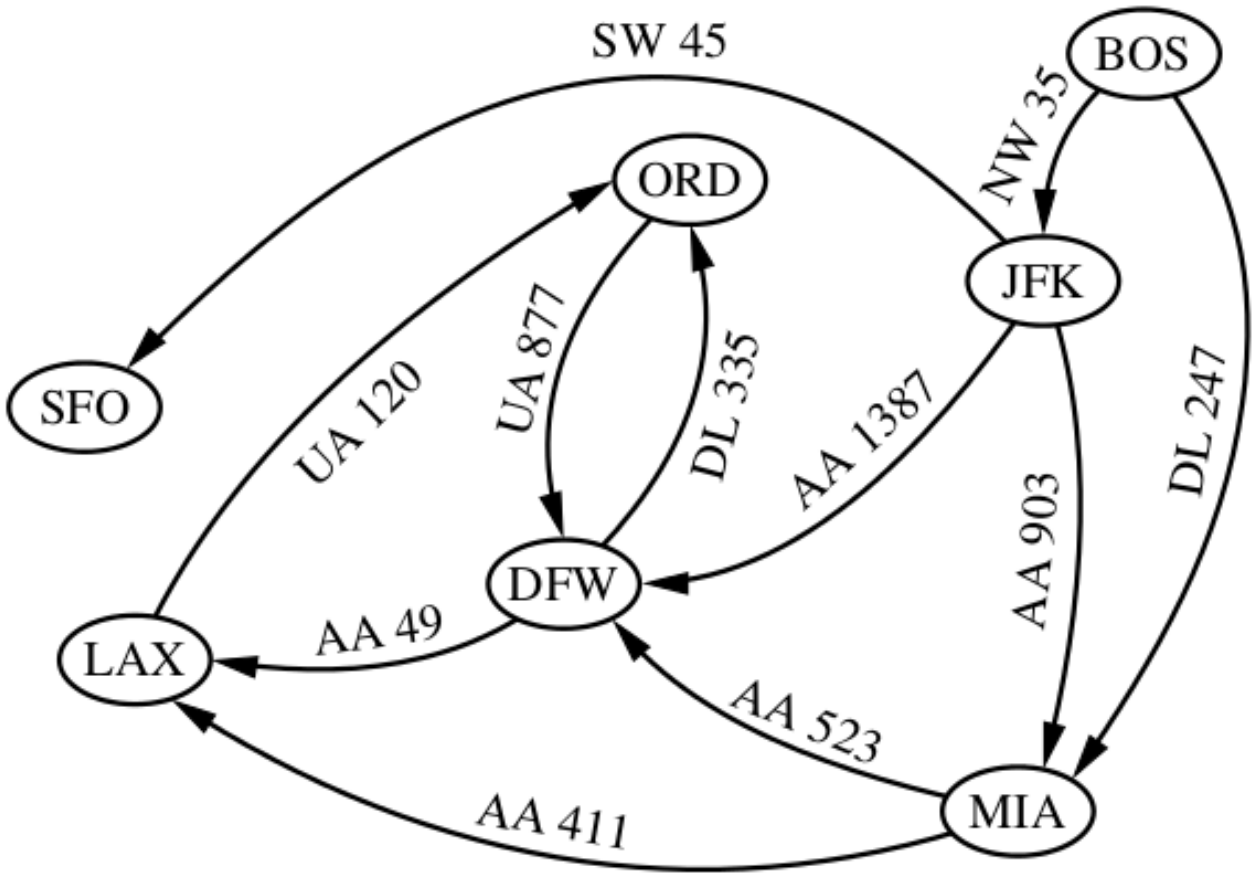


Section 11

Graphs

- A **graph** is a way of representing relationships that exist between pairs of objects
- In other words, a graph is a set of objects, called *vertices* (or nodes), together with a collection of pairwise connections between them, called *edges*
- A graph G is a set V of vertices and a collection E of edges
- Edges in a graph are either *directed* or *undirected*
  - An edge (u, v) is said directed from u to v if the pair (u, v) is ordered, with u preceding v
  - An edge (u, v) is said undirected if the pair (u, v) is not ordered

Graphs


SW 45
BOS
NW 35
ORD
JFK
SFO
UA 120
UA 877
DL 335
AA 1387
DL 247
AA 903
DFW
AA 49
LAX
AA 523
MIA
AA 411

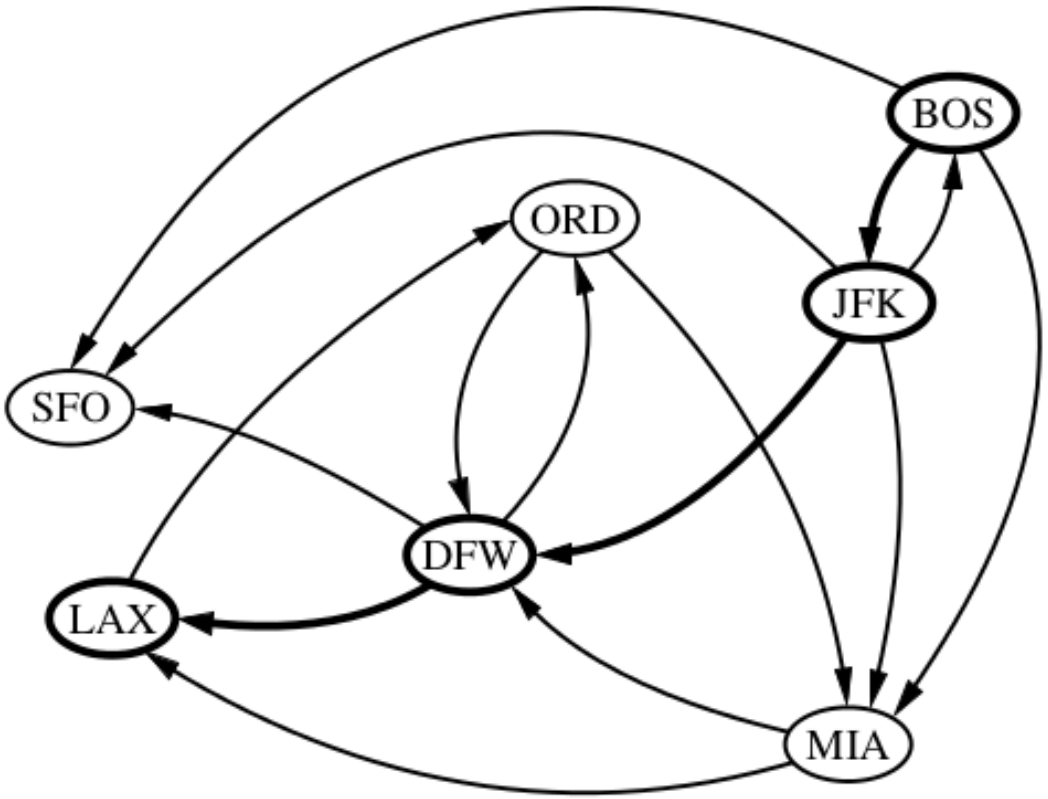
- If all the edges in a graph are undirected, the graph is an *undirected graph*
- Likewise, a *directed graph* (or *digraph*) is a graph whose edges are all directed
- A graph that has both directed and undirected edges is called a *mixed graph*
- The two vertices joined by an edge are called the *end vertices* (or *endpoints*) of the edge
- If an edge is directed, its first endpoint is called *origin* and the other is the *destination* of the edge
- Two edges are called *adjacent* if there is an edge connecting them

- An edge is said to be *incident* to a vertex if the vertex is one of the endpoints of the edge
- The *outgoing edges* of a vertex are the directed edges whose origin is in the vertex
- The *ingoing edges* of a vertex are the directed edges whose destination is in the vertex
- The *degree* of a vertex v, denoted $deg(v)$, is the number of incident edges of v
- The *in-degree* and *out-degree* of a vertex v is the number of the incoming and outgoing edges of v, and are denoted $indeg(v)$ and $outdeg(v)$ respectively

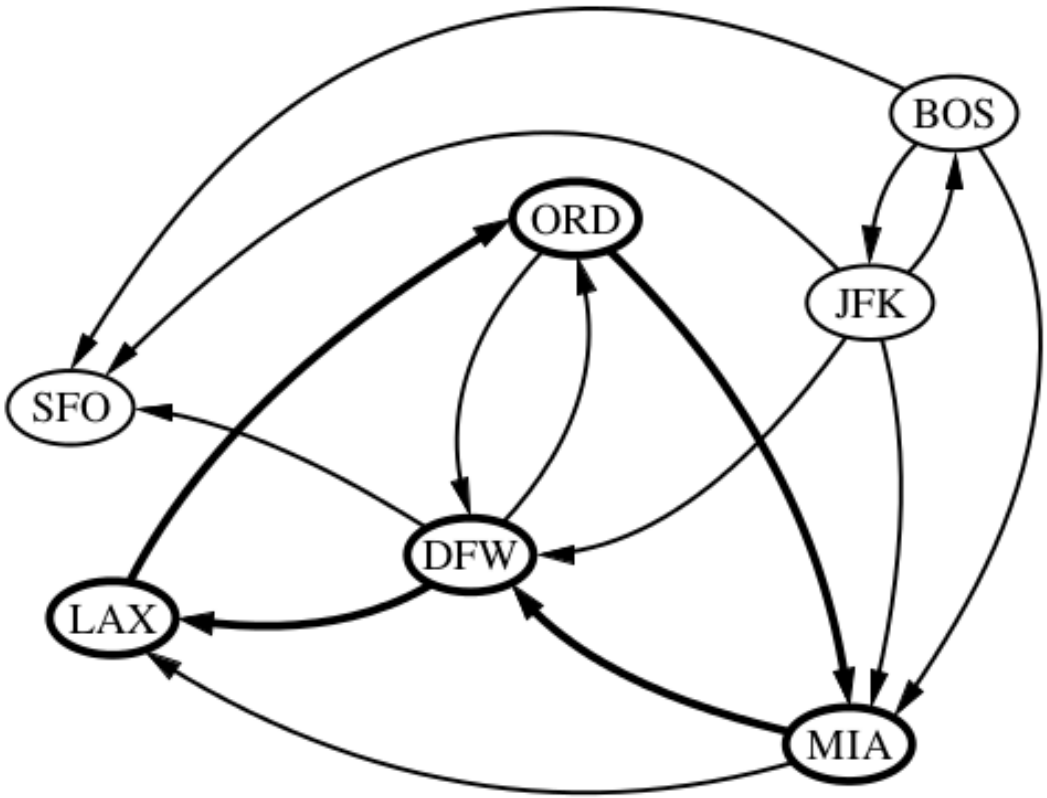
- According to the definition, the group of edges of a graph is referred to as a *collection* (not a *set*), thus allowing two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination
- Such edges are called *parallel edges* or *multiple edges*
- Another special type of edge if one that connects a vertex to itself. In other words, an edge is a *self-loop* if its two endpoints coincide
- Graphs that do not have parallel edges or self-loops are called *simple*

- A *path* is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex
- If the starting and the ending vertices coincide, the path is called *cycle*
- A cycle must include at least one vertex
- A path is called *simple* if each vertex in the path is distinct
- A *directed path* is a path such that all edges are directed are traversed along their direction
- A directed graph is *acyclic* if it has no directed cycles

- Given vertices u and v of a (directed) graph G, we say that u *reaches* v if G has a (directed) path from u to v
- In an undirected graph the notion of reachability is symmetric
- A graph is *connected* if, for any two vertices, there is a path between them
- A directed graph is *strongly connected* is for any two vertices u and v, u reaches v and v reaches u



A directed path from BOS to LAX



A directed cycle (ORD; MIA; DFW; LAX; ORD)

- A *subgraph* of a graph G is a graph H whose vertices and edges are subsets of the vertices and edges of G
- A *spanning subgraph* of G is a subgraph of G that contains all the vertices of the graph G
- If a graph G is not connected, its maximal connected subgraphs are called the *connected components* of G

Proposition 1

If G is a graph with m edges and vertex set V, then

$$\sum_{v \text{ in } V} deg(V) = 2m$$

Justification: An edge (u, v) is counted twice in the summation above; once by its endpoint u and once by its endpoint v. Thus, the total contribution of the edges to the degrees of the vertices is twice the number of edges.

Proposition 2

If G is a directed graph with m edges and vertex set V, then

$$\sum_{v \text{ in } V} indeg(V) = \sum_{v \text{ in } V} outdeg(V) = m$$

Justification: In a directed graph, an edge (u, v) contributes one unit to the out-degree of its origin u and one unit to the in-degree of its destination v. Thus, the total contribution of the edges to the out-degree (in-degree) of the vertices is equal to the number of edges.

Proposition 3

Let G be a simple graph with n vertices and m edges. If G is undirected, then

$$m \leq n(n-1)/2$$

and if G is directed, then

$$m \leq n(n-1)$$

Justification: Suppose G is undirected. Since no two edges can have the same endpoints and there are no self-loops, the maximum degree of a vertex in G is $n-1$ in this case. Thus, by Proposition 1, $2m \leq n(n-1)$.

Now suppose that G is directed. Since no two edges can have the same origin and destination, and there are no self-loops, the maximum in-degree of a vertex in G is $n-1$ in this case. Thus, by Proposition 2, $m \leq n(n-1)$.

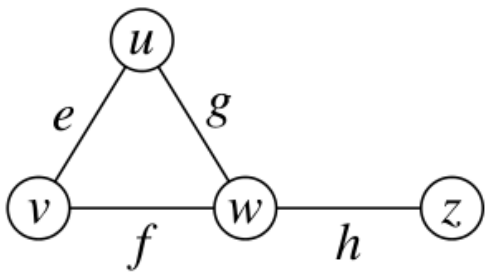
Data Structures for Graphs

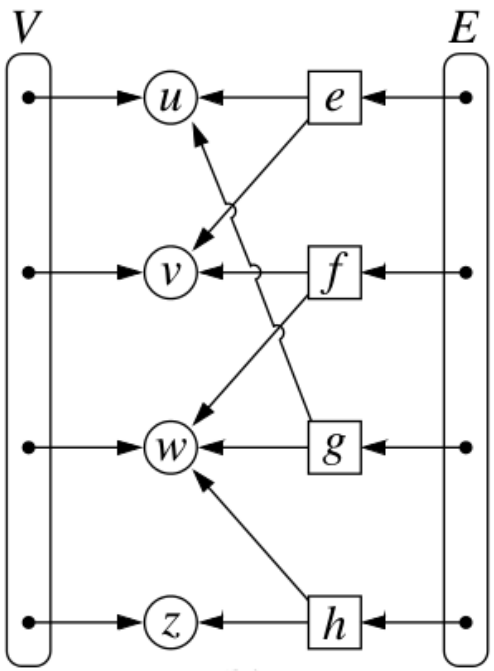
- *Edge list*
- *Adjacency list*
- *Adjacency matrix*

Data Structures for Graphs: Edge List

Edge List

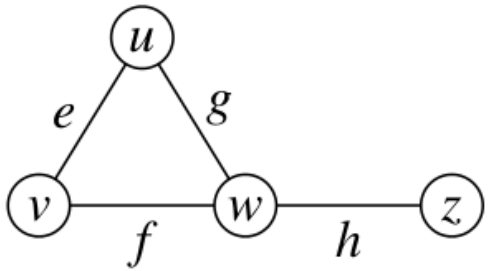
Unordered list of all edges. This is the minimum, since there is no efficient way to locate a particular edge (u, v), or the set of all edges incident to a vertex v





Data Structures for Graphs: Edge List

Edge List



In a simpler way:

```
e ==> u - v
f ==> v - w
g ==> w - u
h ==> w - z
```

Data Structures for Graphs: Edge List

Edge List

Operation	Running Time
`vertex_count()`	$O(1)$
`edge_count()`	$O(1)$
`vertices()`	$O(n)$
`edges()`	$O(m)$
`get_edge(u,v)`	$O(m)$
`degree(v)`	$O(m)$
`incident_edges(v)`	$O(m)$
`insert_vertex(v)`	$O(1)$
`remove_vertex(v)`	$O(m)$
`insert_edge(u,v,x)`	$O(1)$
`remove_edge(e)`	$O(1)$

Data Structures for Graphs: Edge List

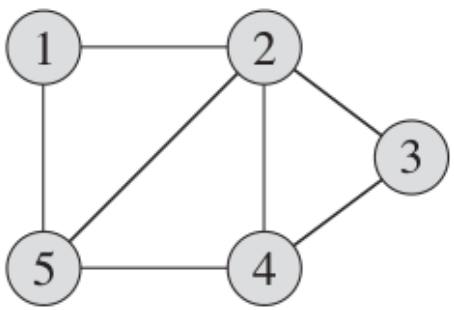
Edge List: performances

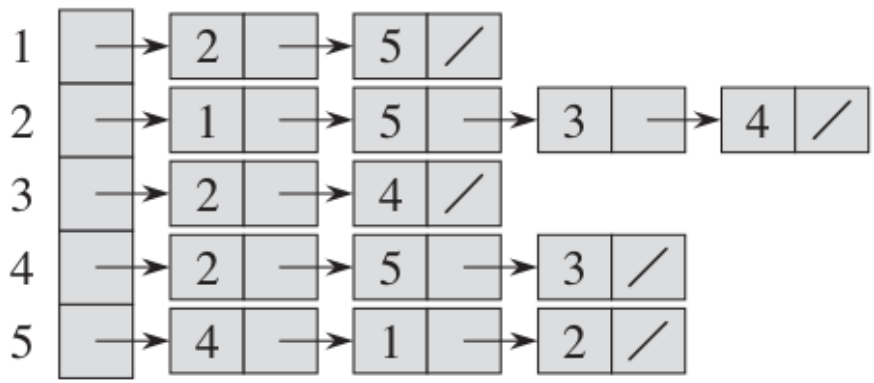
- The edge list structure performs well in terms of reporting the number of vertices or edges, or in producing an iteration of those vertices or edges
- The most significant limitations of an edge list structure are the $O(m)$ running times of methods `get_edge(u,v)`, `degree(v)` and `incident_edges(v)`. This is due to the arrangement of the graph in an unordered list, so that the only way to answer those queries is to inspect all edges
- The update of the graph (insertion of a vertex, or an edge, removal of an edge) requires $O(1)$ time
- The removal of a vertex, on the contrary, requires $O(m)$ time, because all edges incident to the vertex v must also be removed. To locate the incident edges to the vertex, all edges must be examined

Data Structures for Graphs: Adjacency List

Adjacency List

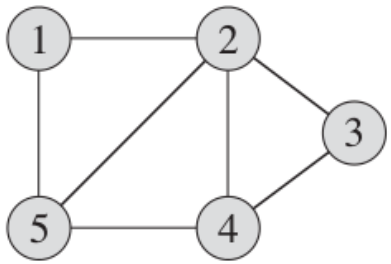
For each vertex is maintained a separate list containing those edges that are incident to the vertex. The complete set of edges can be determined by taking the union of the smaller sets

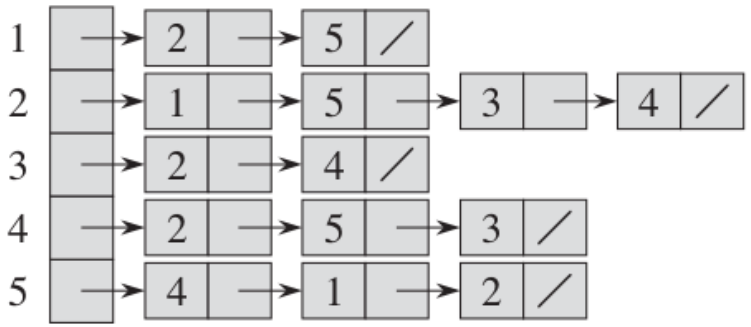




Data Structures for Graphs: Adjacency List

Adjacency List

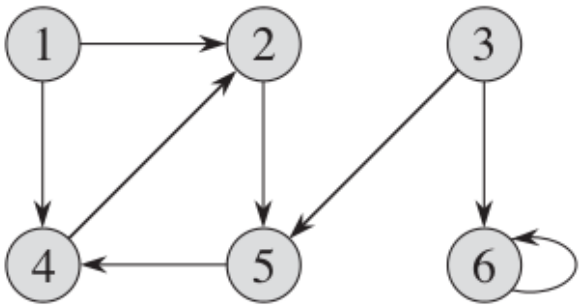


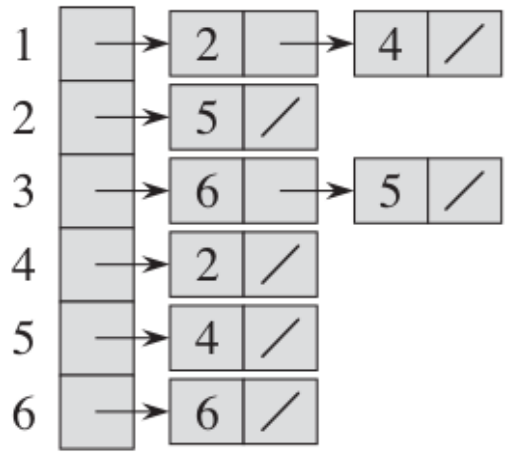



```
graph = {1: set([2, 5]),
         2: set([1, 3, 4, 5]),
         3: set([2, 4]),
         4: set([2, 3, 5]),
         5: set([1, 2, 4])}
```

Data Structures for Graphs: Adjacency List

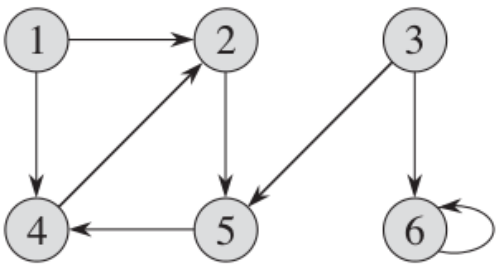
Adjacency List

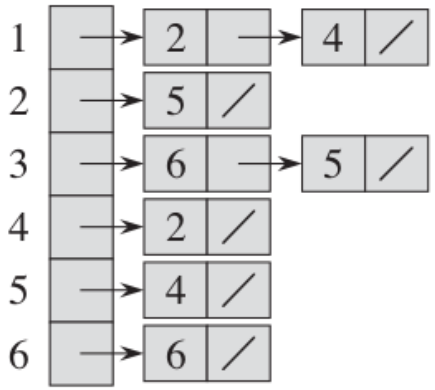




Data Structures for Graphs: Adjacency List

Adjacency List






```
graph = {1: set([2, 4]),
         2: set([5]),
         3: set([5, 6]),
         4: set([2]),
         5: set([4]),
         6: set([6])}
```

Data Structures for Graphs: Adjacency List

Adjacency List Structure

- The **adjacency list** structure groups the edges of a graph by storing them in secondary containers that are associated with each individual vertex
- For each vertex v, we maintain a collection $I(v)$, called the *incidence collection* of v, whose entries are edges incidento to the vertex v.
- In the case of a directed graph, outgoing and incoming edges can be respectively stored in two separate collections, $I_{out}(v)$ and $I_{in}(v)$.

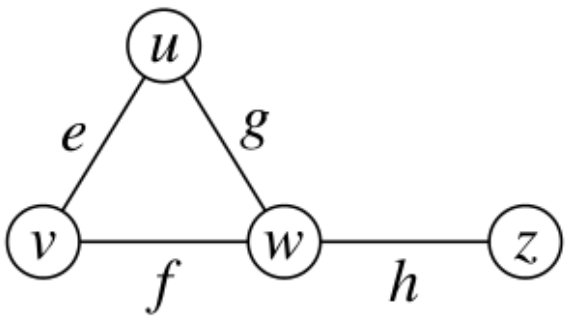
Data Structures for Graphs: Adjacency List

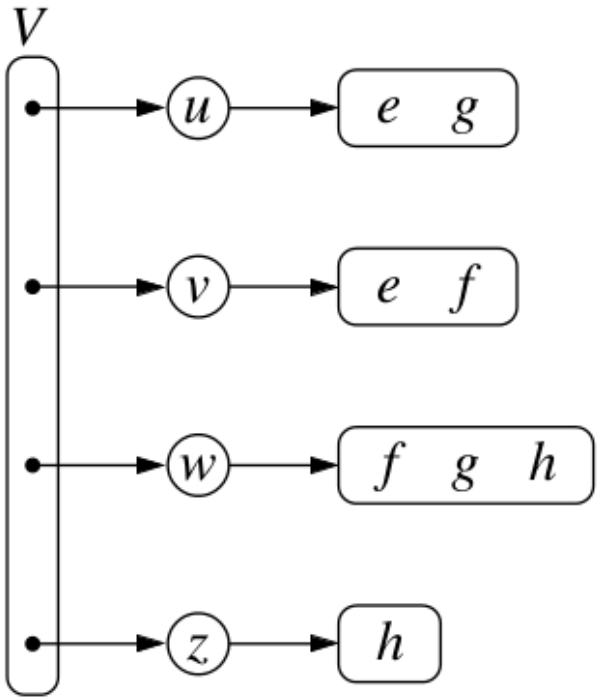
Adjacency List Structure

- The primary structure for an adjacency list must maintain the collection V of vertices in a way so that we can allocate the secondary structure $I(v)$ for a given vertex v in $O(1)$ time.
- This can be done using a positional list to represent V, with each `Vertex` instance maintaining a direct reference to its $I(v)$ incidence collection
- If vertices can be uniquely numbered from 0 to $n-1$, we could use a primary array-based structure to access the appropriate secondary lists
- The benefit of an adjacency list is that the collection $I(v)$ contains exactly those edges that should be reported by the method `incident_edges(v)`. Therefore, we can implement this method by iterating the edges of $I(v)$ in $O(deg(v))$, where $deg(v)$ is the degree of vertex v

Data Structures for Graphs: Adjacency List

Adjacency List Structure





Data Structures for Graphs: Adjacency List

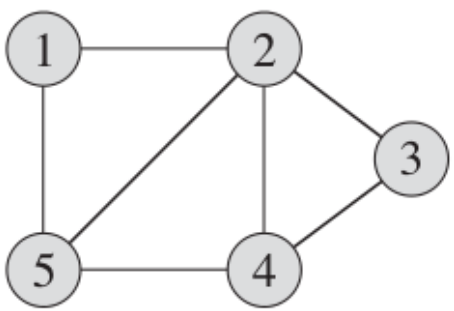
Adjacency List

Operation	Running Time
`vertex_count()`	$O(1)$
`edge_count()`	$O(1)$
`vertices()`	$O(n)$
`edges()`	$O(m)$
`get_edge(u,v)`	$O(min(d_u, d_v))$
`degree(v)`	$O(1)$
`incident_edges(v)`	$O(d_v)$
`insert_vertex(v)`	$O(1)$
`remove_vertex(v)`	$O(d_v)$
`insert_edge(u,v,x)`	$O(1)$
`remove_edge(e)`	$O(1)$

Data Structures for Graphs: Adjacency Matrix

Adjacency Matrix

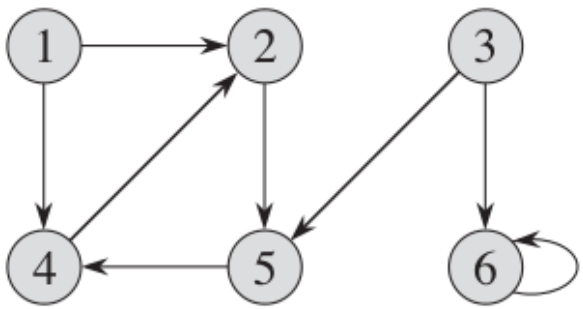
Provides worst-case $O(1)$ access to a specific edge (u, v) by maintaining an $n \times n$ matrix for a graph with n vertices. Each entry is dedicated to storing a reference to the edge (u, v) for a particular pair of vertices u and v. If no such edge exists, the entry will be `None`



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Data Structures for Graphs: Adjacency Matrix

Adjacency Matrix



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Data Structures for Graphs: Adjacency Matrix

Adjacency Matrix

Operation	Running Time
`vertex_count()`	$O(1)$
`edge_count()`	$O(1)$
`vertices()`	$O(n)$
`edges()`	$O(m)$
`get_edge(u,v)`	$O(1)$
`degree(v)`	$O(n)$
`incident_edges(v)`	$O(n)$
`insert_vertex(v)`	$O(n^2)$
`remove_vertex(v)`	$O(n^2)$
`insert_edge(u,v,x)`	$O(1)$
`remove_edge(e)`	$O(1)$

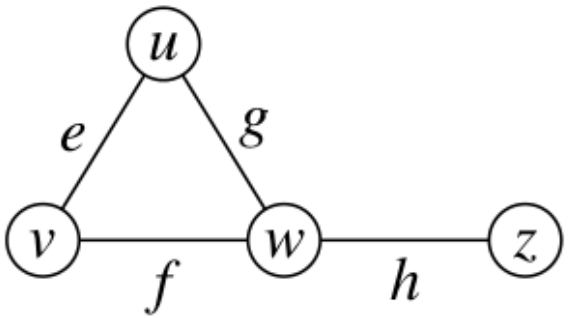
Data Structures for Graphs: Adjacency Matrix

Adjacency Matrix Structure

- The **adjacency matrix** structure for a graph G uses a matrix A, which allows us to locate an edge between a given pair of vertices in *worst-case* constant time
- In this representation, the vertices are the integers in the set $\{0, 1, \ldots n-1\}$ and the edges are pairs of such integers
- Specifically, the cell $A[i, j]$ holds a reference to the edge (u, v) if it exists, where u is the vertex with index i and v is the vertex with index j. Is there is no such edge, then $A[i, j] = None$
- Array A is symmetric is graph G is undirected, as $A[i, j] = A[j, i]$ for all pairs i and j

Data Structures for Graphs: Adjacency Matrix

Ajacency Matrix Structure



	0	1	2	3
u → 0		e	g	
v → 1	e		f	
w → 2	g	f		h
z → 3			h	

Data Structures for Graphs: Adjacency Matrix

Adjacency Matrix Structure

— Advantages

- The most significant advantage of an adjacency matrix is that any edge (u, v) can be accessed in worst-case $O(1)$ time

— Disadvantages

- To find edges incident to vertex v we must examine all n entries in the row associated with v. An adjacency list can locate those edges in optimal $O(deg(V))$ time
- Adding or removing vertices from a graph is problematic, as the matrix must be resized
- The $O(n^2)$ space usage of an adjacency matrix is usually worse than the $O(n + m)$ space required for adjacency lists, although the number of edges in a **dense** graph is proportional to n^2

Data Structures for Graphs

Operation	Edge List	Adj. List	Adj. Map	Adj. Matrix
`vertex_count()`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`edge_count()`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`vertices()`	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`edges()`	$O(m)$	$O(m)$	$O(m)$	$O(m)$
`get_edge(u,v)`	$O(m)$	$O(min(d_u, d_v))$	$O(1)$ exp.	$O(1)$
`degree(v)`	$O(m)$	$O(1)$	$O(1)$	$O(n)$
`incident_edges(v)`	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
`insert_vertex(v)`	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
`remove_vertex(v)`	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
`insert_edge(u,v,x)`	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$
`remove_edge(e)`	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$

Graph ADT: Implementation of the Adjacency List /1

```
def create_graph():
    return dict()

def add_node(graph, node):
    if not node in graph:
        graph[node] = []
    return graph

def add_edge(graph, u, v):
    if u in graph and v in graph:
        if v not in graph[u]:
            graph[u].append(v)
        if u not in graph[v]:
            graph[v].append(u)
    return graph
```

Graph ADT: Implementation of the Adjacency List /2

```
def add_nodes(graph, nodes):
    for node in nodes:
        graph = add_node(graph, node)
    return graph

def add_edges(graph, edges):
    for edge in edges:
        #u = edge[0]
        #v = edge[1]
        #graph = add_edge(graph, u, v)
        graph = add_edge(graph, *edge)
    return graph
```

Graph ADT: Implementation of the Adjacency List /3

Implementation

```
def degree(graph):
    # dizionario con il degree di ogni nodo del grafo
    deg = dict()
    for node in graph:
        deg[node] = len(graph[node])
    return deg

def is_complete(graph):
    # restituisce True se il grafo e' completo
    n = len(graph)
    for node in graph:
        if len(graph[node]) < n - 1:
            return False
    return True
```

Graph ADT: Implementation of the Adjacency List /4

Implementation

```
def get_edge(graph, edge):
    # restituisce True se esiste l'arco indicato
    u, v = edge
    if u in graph[v] and v in graph[u]:
        return True
    else:
        return False

def nodes(graph):
    # restituisce una lista dei nodi del grafo
    return list(graph.keys())
```

Graph ADT: Implementation of the Adjacency List /5

Implementation

```
def edges(graph):
    # restituisce una lista degli archi del grafo
    Edges = []
    for k, v in graph.items():
        for u in v:
            if (k,u) not in Edges and (u,k) not in Edges:
                Edges.append((k,u))
    return Edges

def vertex_count(graph):
    # restituisce il numero di vertici del grafo
    return len(graph)

def edge_count(graph):
    # restituisce il numero di archi del grafo
    return len(edges(graph))
```

Graph ADT: Implementation of the Adjacency List /6

Implementation

```
def incident_edges(graph, u):
    # restituisce la lista degli archi incidenti nel nodo u
    Edges = []
    if u in graph:
        for node in graph[u]:
            Edges.append((u, node))
    return Edges
```

Graph ADT: Implementation of the Adjacency List /7

Implementation

```
def remove_edge(graph, edge):
    # restituisce il grafo privato dell'arco
    u, v = edge
    if u not in graph or v not in graph:
        print("Almeno uno dei nodi non esiste")
    else:
        if u in graph[v]:
            graph[v].remove(u)
        if v in graph[u]:
            graph[u].remove(v)
    return graph
```

Graph ADT: Implementation of the Adjacency List /8

Implementation

```
def remove_vertex(graph, v):
    # restituisce il grafo privato del nodo
    # il nodo deve essere rimosso dalla lista di ogni
        contatto
    if v in graph:
        for node in graph[v]:
            if v in graph[node]:
                graph[node].remove(v)
        del graph[v]
    return graph
```

Graph Traversals

A **traversal** is a systematic procedure for exploring a graph by examining all of its vertices and edges. A traversal is efficient if it visits all the vertices and edges in time proportional to their number, that is, in linear time

Graph Traversals

Interesting problems that deal with reachability in an **undirected graph** G include the following:

- Computing a path from vertex u to vertex v, or reporting that no such path exists
- Given a start vertex s of G, computing for every vertex v of G a path with the minimum number of edges between s and v, or reporting that no such path exists
- Testing whether G is connected
- Computing a spanning tree of G, is G is connected
- Computing a cycle in G, or reporting that G has no cycles

Graph Traversals

Interesting problems that deal with reachability in an **directed graph** G include the following:

- Computing a directed path from vertex u to vertex v, or reporting that no such path exists
- Finding all the vertices of G that are reachable from a given vertex v
- Determine whether G is acyclic
- Determine whether G is strongly connected

Depth-First Search: Running Time of DFS

- DFS is an efficient method for traversing a graph
- DFS is called at most once on each vertex (since it gets marked as visited), and therefore every edge is examined at most twice for an undirected graph (once for each of its vertices), and at most once for a directed graph (from its origin vertex)
- If we let $n_s \leq n$ be the number of vertices reachable from a vertex s, and $m_s \leq m$ be the number of incident edges to those vertices, a DFS starting at s runs in $O(n_s + m_s)$ time, provided that ...

Depth-First Search: Running Time of DFS

...

- The graph is represented by a data structure such that creating and iterating through the `incident_edges(v)` takes $O(deg(v))$ time, and the `e.opposite(v)` method takes $O(1)$ time. The adjacency list is one such structure (not the adjacency matrix)
- There is a way to mark a vertex or edge as explored, and to test if a vertex or edge has been explored in $O(1)$ time

Given these assumptions, a number of problems can be solved

Breadth-First Search

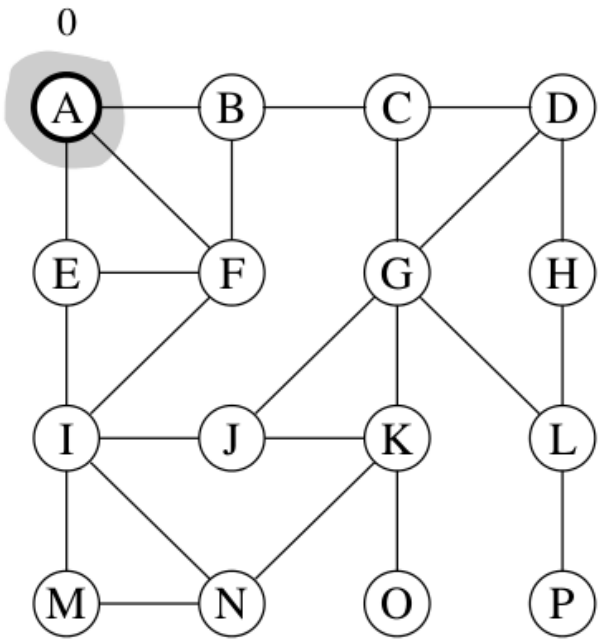
- DFS defines a traversal that could be physically traced by a single person exploring the graph
- **Breadth-First Search** (BFS) can be described as sending many explorers who collectively explore the graph (nothing to do with parallel implementations)
- A BFS proceeds in rounds and divides the vertices in levels
- In the first round, are marked as "visited" all vertices adjacent to the start vertex s (these vertices are one step away from the beginning and are placed into level 1)
- In the second round, explorations concerns vertices two steps (edges) away from the starting vertex. The new vertices are adjacent to level 1 vertices and are therefore placed in level 2
- The process continues in similar fashion terminating when no new vertices are found in a level

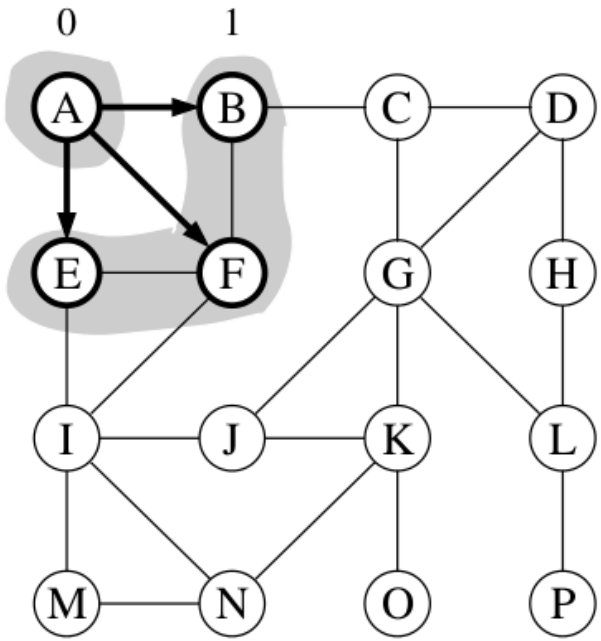
Breadth-First Search: implementation (iterative)

```
def bfs(graph, start):
    explored = []
    q = []
    q.append(start)
    while q:
        u = q.pop()
        if u not in explored:
            explored.append(u)
            for v in list(graph[u]):
                q.append(v)
                print(q)
    return explored
```

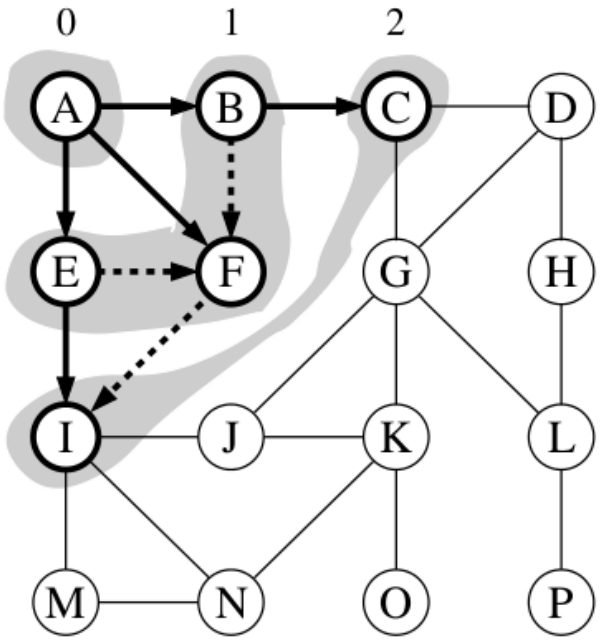
bfs.py

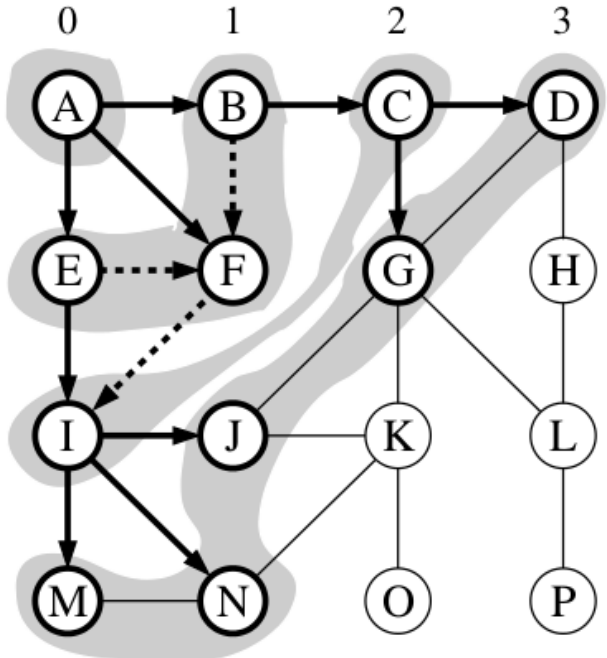
Breadth-First Search



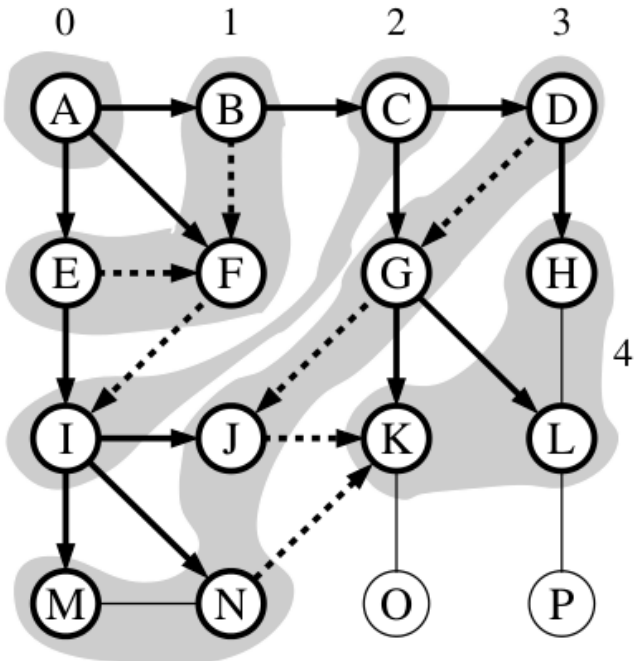


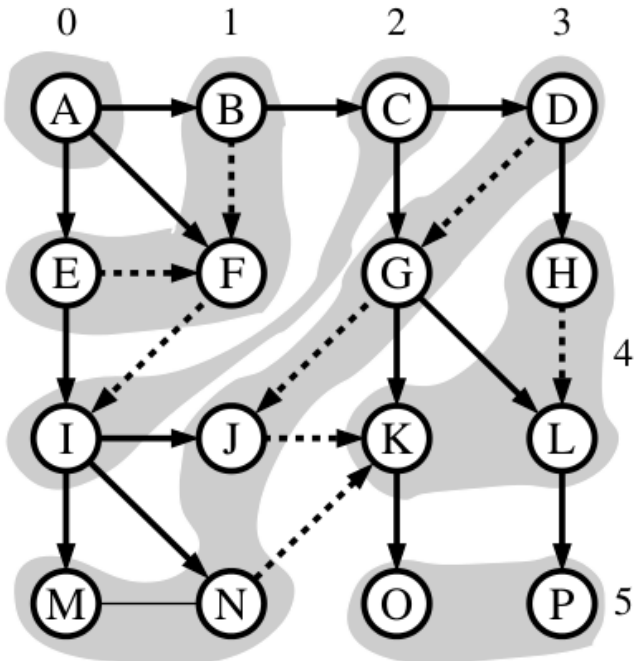
Breadth-First Search





Breadth-First Search





Type of edge	Connects ...
back edge	a vertex to one of its ancestors
forward edge	a vertex to one if descendants
cross edge	a vertex to another vertex (neither ancestor nor descendant)

For BFS:

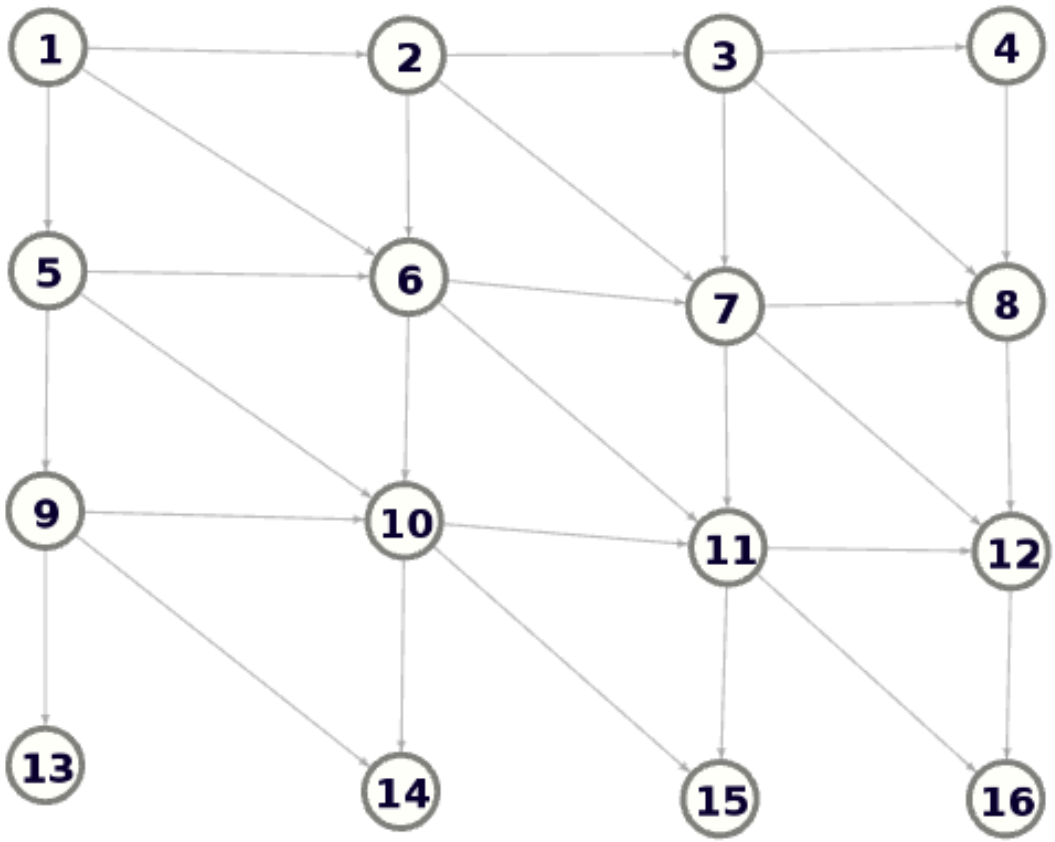
- on undirected graphs, all nontree edges are cross edges
- on directed graphs, all nontree edges are wither back edges or cross edges

Proposition

Let G be an undirected or directed graph on which a BFS traversal starting at vertex s has been performed. Then

- *The traversal visits all vertices of G that are reachable from s*
- *For each vertex v at level i, the path of the BFS tree T between s and v has i edges, and any other path of G from s to v has at least i edges*
- *If (u, v) is an edge that is not in the BFS tree, then the level number of v can be at most 1 greater than the level number of u*

Breadth-First Search



Breadth-First Search