Data Structures and Algorithms Exam Slides

SLIDES THAT ARE ALLOWED TO BE USED FOR FINAL EXAMINATIONS

Course Web Sites

Both sites are identical and synchronized
Use the second if the first is down

http://brd4.braude.ac.il/~samyz/DSAL

http://tinyurl.com/samyz/dsal/index.html

Introduction to:

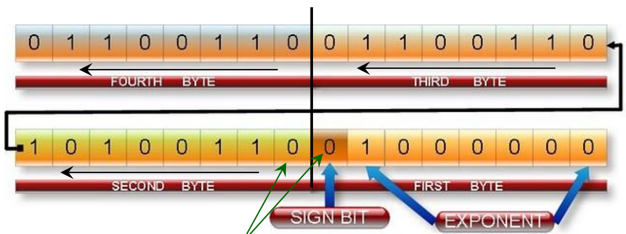
DATA STRUCTURES AND ALGORITHMS

Data Structures

- Systematic methods for organizing information in a computer
- A **data type** consists of the values it represents and the operations defined upon it
- In the **C** programming language, a data type is usually represented by the **struct** concept.
- But the **struct** represents only the data type values and does not describe what kind of operations can be applied on the data type
- In object oriented languages, the **class** concept extends the **struct** concept by also adding methods that can be applied on a data type

Data Type Binary Representation

- Data types may be viewed in several ways:
  - As **abstract entities**
  - As **concrete implementations**
- For example, there are many ways to represent a floating number like x=5.2 – here is one common way to do it (32 bit arch):



exp = e +127
See comment
Below slide

5.2 = 5 + 0.2 = 101 + 0.0011001100110011001100110011…
= 101.0011001100110011001100110011…
= + 1.0100110011001100110011001100110011… * 2^2

Data Type Binary Representation

- Note that some data types may not have a fully accurate representation!
- For example, the float number x=5.2 is not really equal to its binary representation above! Moreover, it will have a different value in a 64 bit architecture!
- This is however will not concern us in this course as we're more concerned with the **abstract view** of data types!
- Binary representations of data types is the business of other courses and not ours!
- We do however need to be aware of the basic ideas of representations in order to be able to do realistic analysis of algorithms, estimate input and output sizes, estimate space and run time figures

Abstract Data Type (ADT)

- An **abstract data type** (**ADT**) is a programmer-defined data type that specifies a set of data values and a collection of **well-defined operations** that can be performed on those values
- Only the formal definition of the data type is important and **NOT** how it is implemented in binary form or in hardware
- This is sometimes called:
  "Separation of Interface and Implementation"
- **Information Hiding** – how the data is represented and how the operations are implemented is completely <u>irrelevant</u> when we define a new **A**bstract **D**ata **T**ype (**ADT**) !

Example: String ADT

```
String Data Type:
    An string of characters like
    s = "Hello World"
    s = "Guido Van Rossum, 1993"

Operations:
    upper(s)           All characters to upper case
    lower(s)           All characters to lower case
    find(s,w)          Find a word w in s (return index)
    replace(s,w1,w2)   Replace sub word w1 with w2
```

EXAMPLE CODE:

```
s = "Hello World"
upper(s) = "HELLO WORLD"
lower(s) = "hello world"
find(s, "Wo") = 6
replace(s, "lo", " NEW") = "Hel NEW World"
```

ADT As Interface Design

- Note that the term ***"string of characters"*** does not imply anything about its implementation (how English characters are represented?)
- It can be implemented as a C **array of characters** terminated by a **NULL**
- It can be implemented like a Java or C++ String object
- We may even decide to encode and compress the string if it size is too large
- We can decide to break each string to chunks of 4K in different memory locations and keep a central table for accessing these chunks, etc …

ADT As Interface Design

- Similarly, nothing on how the **find()** and **replace()** algorithms should be implemented is mentioned!
- All we care is about how we **<u>Interface</u>** with the string data type? (How to do? instead of how it is done?)
- All implementation issues are **<u>irrelevant</u>** to the ADT specification!

Algorithms

- After defining an ADT we will proceed to the second part of our course: ***ALGORITHMS***
- Named after the mathematician Muḥammad ibn Mūsā al-Khwārizmī (Bagdad 780-850) which invented the concept and the first mathematical algorithms (including an algorithm for solving quadratic equations)
- **ALGORITHM:**
  - An effective method expressed as a finite list of well-defined instructions for calculating a function (Wikipedia)
  - Simply put, a ***data structure*** is a systematic way of organizing and accessing data, and an ***algorithm*** is a step-by-step procedure for performing some ***task*** in a finite amount of time (Goodrich/Tamassia/Goldwasser book)

أبو عبد الله محمد بن موسى الخوارزمي

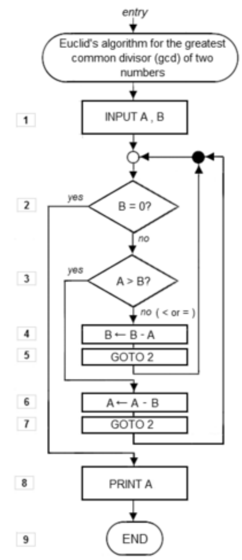
Example: Euclid's GCD Algorithm

- **GCD = Greatest Common Divisor**
- Perhaps one of the most famous algorithms in history
- Formulated by Euclid around 300 BC (without knowing the algorithm concept)
- **Problem:** given two integers A and B, find the largest integer G which divides both A and B
- Here is the most naïve way to solve the problem:

```
def gcd1(a, b):
    if a == 0: return b
    if b == 0: return a
    m = min(a,b)
    greatest = 1
    d = 1
    while d <= m:
        if a%d == 0 and b%d == 0:
            greatest = d
        d += 1
    return greatest
```

Flow Charts

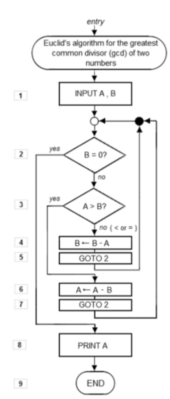
- Modern algorithms are often written as "Flow Charts" as the figure on the right side which describes Euclid's algorithm
- There are many graphical computer programs for drawing beautiful Flow Charts which you can use for designing your algorithms
- Here is a Flow Chart for a popular version of Euclid's Algorithm:



Euclid's GCD Algorithm in Python

- The other method for expressing Algorithm is by a semi-formal language called Pseudo-Code
- Since Python is simple and very readable as pseudo-code and at the same time it is also a fully running formal language, there are more and more courses and books that use it for a data structures and algorithms courses

```
def gcd2(a, b):
    if b == 0:
        return a
    else:
        if a>b:
            a = a-b
        else:
            b = b-a
        return gcd2(a,b)
```



Euclid's GCD Algorithm: Correctness Proof

- Theorem: Assume that a>b>0, are two integers.
  For any integer **d**: **d** divides **a** and **b** ⇔ d divides a-b and b
- Proof is easy!
- Definition: div(a,b) = {d | d divides a and b}
- Consequence: div(a,b) = div(a-b, b)
- Consequence: gcd(a,b) = gcd(a-b, b)

Euclid's GCD Algorithm in Python Recursive Algorithm

However the gcd2 is recursive, and thus can fail if a and b are very large:

```
def gcd2(a, b):
    if b == 0:
        return a
    else:
        if a>b:
            a = a-b
        else:
            b = b-a
        return gcd2(a,b)
```

Problem with recursion:

```
RuntimeError: maximum recursion depth exceeded in cmp
```

Euclid's GCD Algorithm in Python Non-recursive Algorithm

```
def gcd3(a, b):
    "Find the greatest common divisor for two integers: a,b"
    if a == 0:
        return b
    elif b == 0:
        return a
    while a != b:
        if a > b:
            a = a - b
        else:
            b = b - a
    return a
```


Python's GCD Algorithm

- Python contains an official **GCD** algorithm as part of the **fractions** module:

```
def gcd(a,b):
    while a:
        a, b = b%a, a
    return b
```

- This follows immediately from: gcd(a,b) = gcd(a, b-a)
- For any integer k, gcd(a,b) = gcd(a, b – ka) = gcd(b-ka, a)
- If k = b/a, then b-ka = b%a, and we get: gcd(a,b) = gcd(b%a, a)
- Why the algorithm must stop? (could be an infinite loop?)
  Prove that the numbers are decreasing until a==0

Run Time Analysis

```
import time

def gcd_time_test(f, a, b):
    print "Running %s(%d,%d)" % (f.func_name, a,b)
    start = time.time()
    try:
        print "gcd =", f(a,b)
    except Exception as e:
        print e
    end = time.time()
    print "runtime = %.3f seconds" % (end-start,)
```


Which Algorithm is the fastest?

This is just a simple performance test.
A more rigorous test should sample a larger variety of numbers and each calculation should be repeated several times (average time)

```
def test1():
    a = 2**13 * 3**4 * 5**3
    b = 2**7 * 3**5 * 5**2
    gcd_time_test(gcd1, a, b)
    gcd_time_test(gcd2, a, b)
    gcd_time_test(gcd3, a, b)
    gcd_time_test(gcd4, a, b)
```

Example 2: Primality

- Data Type: unsigned integers: **0, 1, 2, 3, 4, 5, …**
- Definition: a **prime number** is an integer p>1 which has exactly two divisors: 1, and p.
- Problem: Given a positive integer **n**, find if **n** is a prime number?
- Here is a Naïve simple algorithm that solves this problem:

```
def is_prime(n):
    if n <=1: return False
    i=2
    while i<n:
        if n%i==0:
            return False
        i += 1
    return True
```


Container/Collection Terminology

- In Object Oriented Design, a **Container** is any object that contains other objects in itself
- Other words: a collection is a group of values with no implied organization or relationship between the individual values (Rance Necaise book)
- Some languages restrict the elements to a specific data type such as integers or floating-point values
  - Python collections do not have such restriction

Collection Types

- The programming languages and literature are full with many such object with many different names
  - **List**
  - **Array**
  - **Sequence**
  - **Vector**
  - **Set**
  - **Stack**
  - **Queue**
  - **Heap**
  - **Map**
  - **Hash Table**
  - **Dictionary**
  - **Tree**
  - **Graph**
  - **Multimap**
  - **Multiset**
  - **Priority Queue**
  - **String**

Leaf Objects

- In contrast to Container object, a Leaf Object is an object that does not contain any reference to other objects ("has no child objects")
- In Python these are sometimes called "primitive types"
  - Integer
  - Float
  - Complex number
  - Boolean
- Leaf Objects are the building blocks from which all other objects are built

Primitive Types

- Integer: -5, 19, 0, 1000 (C long)
- Float: -5.0, 19.25, 0.0, 1000.0 (C double)
- Complex numbers: a+bj
- Boolean: True, False
- Long integers (unlimited precision)
- Immutable string: “xyz”, “Hello, World”

Arithmetic Operations

Operation	Result
x + y	sum of *x* and *y*
x - y	difference of *x* and *y*
x * y	product of *x* and *y*
x / y	quotient of *x* and *y* (Integer division if x,y integers
x % y	remainder of x / y
-x	*x* negated
+x	*x* unchanged
abs(x)	absolute value or magnitude of *x*
int(x)	*x* converted to integer
long(x)	*x* converted to long integer (this is very long …)
float(x)	*x* converted to floating point
complex(re,im)	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero
c.conjugate()	conjugate of the complex number *c*. (Identity on real numbers)
divmod(x, y)	the pair (x / y, x % y)
pow(x, y)	*x* to the power *y*
x ** y	*x* to the power *y*

Comparisons

Operation	Meaning
<	strictly less than
<=	less than or equal
>	strictly greater than
=>	greater than or equal
==	equal
!=	not equal
is	object identity
is not	negated object identity

Bitwise Operations

Operation	Result
x | y	bitwise *or* of *x* and *y*
x ^ y	bitwise *exclusive or* of *x* and *y*
x & y	bitwise *and* of *x* and *y*
x << n	*x* shifted left by *n* bits
x >> n	*x* shifted right by *n* bits
~x	the bits of *x* inverted

The Complex Numbers Class

- The **cmath** module defines Complex numbers arithmetic
- Python contains a built-in type (class) for complex numbers
- A complex number object has two fields and one method:

imag	imaginary part
real	real part
conjugate()	The conjugate number

```
import cmath
z = cmath.sqrt(-9)
⇒ 3j
z = cmath.sqrt(5-12j)
⇒ (3-2j)
z.imag
⇒ -2.0
z.real
⇒ 3.0
z.conjugate()
⇒ (3+2j)
```


Abstract Data Types Operations

Constructors	Methods for creating new objects
Accessors	Methods for accessing internal data fields without modifying the data!
Mutators	Methods for modifying object data fields
Iterators	Methods for processing data elements sequentially

List Abstarct Data Type
Procedural Design – part 1

- `L = list_create1(e0, e1, e2,... ,en-1)`
  - Create a new list `L` from n elements: `e0, e1, …, en`
- `L = list_create2(other)`
  - Create a new list `L` from other list or another container structure
- `get_item(L,i)` - Get element `i` of list `L`
- `set_item(L,i,e)` - Set element `i` of list `L` to `e`
- `contains(L,e)`
  - Check if element `e` belongs to list `L.` Returns: Boolean `True` or `False`
- `append(L,e)`
  - Add a new element `e` to `L`
  - What if `e` already belongs to `L`? (answer: duplications are allowed!)
- `remove(L,e)`
  - Remove an element `e` from `L`
  - What if `e` is not in `L`? (two possibilities: 1. do nothing, 2. raise an error)

The List ADT
Procedural Design – part 2

- `insert(L, index, e)`
  - Insert a new element `e` at index `index`
  - Side effect: list grows by one element
- `size(L)`
  - Return the size of `L`
- `extend(L,L2)`
  - Extend list `L` by list `L2`
- `reverse(L)`
- `slice(L,i,j)`
  - Return a sub-list consisting of all elements of `L` from index `i` to index `j-1`
- `index(L,e)`
  - Find the index of element `e` in `L`

Test Driven Development

- In this highly recommended methodology you write your tests before the implementation of your ADT !!!
- After implementation, your tests should run and **PASS** after each modification you make to your implementation (“nightly test regression”)
- The following tests are your “insurance policy” that your implementation is correct. The more tests you write, the better you’re insured

```
# Testing our List ADT
L1 = list_create1(2, 3, 5, 7, 11)
L2 = list_create2(L1) # copy constructor
assert L2 == L1       # Assertion
append(L1, 37)
remove(L1, 2)
remove(L1, 3)
L3 = list_create1(5, 7, 11, 37)
assert L1 == L3       # Assertion
```


ADT Implementation

- After defining an abstract data type, we need to implement it in a specific programming language
- First we must define a concrete data structure in the particular language for representing our abstract data
- Python basic data structures are usually implemented in the C programming language
- More complex data structures are usually implemented over the Python languages itself, and later transformed to C code if performance is critical

Python List ADT Implementation

```
typedef struct {
    int ob_refcnt ;
    struct _typeobject *ob_type ;
    int ob_size ;
    PyObject **ob_item ;
    int allocated ;
} PyListObject ;
```

- Lists in Python are implemented as a **C <u>array</u> of PyObject pointers**
- `**ob_item` is an **<u>array</u>** of pointers to **PyObject pointers**
- A Python list is therefore an array of references to any **Python objects**!
- A `PyListObject` can grow and shrink (so there could be many calls to `malloc` and `free` on the way … but Python users shouldn't care)

C Implementation of append

```
static int app1(PyListObject *self, PyObject *v) {
    Py_ssize_t n = PyList_GET_SIZE(self) ;

    assert (v != NULL) ;
    if (n == PY_SSIZE_T_MAX) {
        PyErr_SetString(PyExc_OverflowError,
            "cannot add more objects to list") ;
        return -1 ;
    }

    if (list_resize(self, n+1) == -1)   /* increase list size by +1 */
        return -1 ;

    Py_INCREF(v) ;                      /* incr reference count of v */
    PyList_SET_ITEM(self, n, v) ;       /* add pointer v at the end */
    return 0 ;
}
```

C Implementation of insert

```
static int ins1(PyListObject *self, Py_ssize_t where, PyObject *v) {
    Py_ssize_t i, n = Py_SIZE(self) ;
    PyObject **items ;
    if (v == NULL) {
        PyErr_BadInternalCall() ; return -1 ;
    }
    if (n == PY_SSIZE_T_MAX) {
        PyErr_SetString(PyExc_OverflowError,  "cannot add more objects to list") ;
        return -1 ;
    }
    if (list_resize(self, n+1) == -1)
        return -1 ;
    if (where < 0) {
        where += n ;
        if (where < 0)
            where = 0 ;
    }
    if (where > n)
        where = n ;
    items = self->ob_item ;
    for (i = n ; --i >= where ; )          /* Move all items [i:n] to [i+1:n+1] ! */
        items[i+1] = items[i] ;
    Py_INCREF(v) ;
    items[where] = v ;                     /* insert the new value v at index where */
    return 0 ;
}
```

No time in class
Home reading!

C Implementation of list reverse

```
/* Reverse a slice of a list in place, from lo to hi (exclusive) */
static void reverse_slice(PyObject **lo, PyObject **hi) {
    assert(lo && hi) ;    /* make sure lo and hi are not NULL */
    PyObject* tmp
    --hi ;                /* hi itself is excluded */
    while (lo < hi) {
        tmp = *lo ;
        *lo = *hi ;
        *hi = t ;
        ++lo ;
        --hi ;
    }
}
```

List Reverse Implementation (2)
procedural design, Python, Recursive

```
def _reverse_recursive(S, begin, end):
    """ Reverse elements in slice S[begin:end+1] """
    if end>begin:
        # swap first and last elements
        S[begin], S[end] = S[end], S[begin]
        # Recursion:
        _reverse_recursive(S, begin+1, end-1)

def reverse_recursive(S):
    _reverse_recursive(S, 0, len(S)-1)
```


Reverse Implementation (3)
procedural design, Python, Iterative

```
def reverse_iterative(S):
    """ Reverse elements in sequence S."""
    a, b = 0, len(S)-1
    while a < b:
        S[a], S[b] = S[b], S[a]
        a, b = a+1, b-1
```

```
Example:
S = [0, 1, 2, 3]
a, b = 0, 3  ==>  [3, 1, 2, 0]
a, b = 1, 2  ==>  [3, 2, 1, 0]
a, b = 2, 1  ==>  done
```

Testing your implementation

- Remember: tests must be written before you even think about an implementation!
- Make sure your tests cover the major features
- After writing an implementation you must run your tests: if they fail, then your implementation is bad
- After changing an implementation you must run all the tests again
- You may decide to throw away the whole implementation and write a new one, without any change to your ADT specification ("same Interface different implementation") – your tests should pass again with the new implementation!

Interface and Implementation
Totally Separated Things !!!

- There should be a **total separation** between an **ADT** specification (sometimes called "Interface specification") and its possibly many implementations
- For example, the Python Language has a full implementation over Java (called **Jython**), and at the same time Microsoft has a full implementation of Python over C# which is called **IronPython**
- The Python implementation over **C** is called **CPython**
- The same **Python tests** must all pass in all three implementations: **CPython**, **Jython**, and **IronPython** !
- The Python language itself is a pure interface! Unlike low level languages such as C it does not have any business with hardware registers, contiguous memory cells, etc. No relation to hardware at all!

Problems with Procedural Design

- No clear separation between major and minor data types
- For example, when we see `append(a,b)` it's not always clear which is the list and who is the element?
- Composite expressions like:
  `insert(append(extend(L,L2),a3),7,b4)`
  can be very hard to read and understand
- Generic method names like `append()`, `insert()`, `remove()`, `size()`, etc., cannot be reused for a different data structure (like `FILE` or `Vector`), since they are global and already taken by the List data type … this is a serious trouble.
- Code reuse is difficult

The List ADT
Object Oriented Design – part 1

- `L = list_create1(e0, e1, e2,..., en-1)` `[constructor]`
  - Create a new list `L` from n elements: `e0, e1,... , en-1`
- `L = list_create2(other)` `[constructor]`
  - Create a new list `L` from other list or a container structure
- `L.item(i)` - Get element `i` of list `L` `[accessor]`
- `L.contains(e)` `[accessor]`
  - Check if element **e** belongs to list `L`
  - Returns: boolean `True` or `False`
- `L.append(e)` `[mutator]`
  - Add a new element `e` to `L`
  - What if **e** already belongs to `L`? (answer: duplications are allowed!)
- `L.remove(e)` `[mutator]`
  - Remove an element `e` from `L`
  - What if `e` is not in `L`? (two possibilities: 1. do nothing, 2. raise an error)

The List ADT
Object Oriented Design – part 2

- `L.replace(index, e)` [mutator]
  - Replace element at index **index** with **e**
- `L.insert(index, e)` [mutator]
  - Insert a new element **e** at index **index**
  - Side effect: list grows by one element
- `L.size()` [accessor]
  - Return the size of **L**
- `L.extend(L2)` [mutator]
  - Extend list **L** by list **L2**
- `L.reverse()` [mutator]
- `L.slice(i,j)` [accessor]
  - Return a sub-list consisting of all elements of **L** from index **i** to index **j-1**
- `L.index(e)` [accessor]
  - Find the index of element **e** in **L**

Test Driven Development

- We need to update all our procedural oriented test to be object oriented

```
# Testing our List ADT
L1 = list_create1(2,3,5,7,11)
L2 = list_create2(L1)      # “copy constructor”
assert L2 == L1            # Assertion
assert L2.item(0) == 2
L1.append(37)
L1.remove(2)
L1.remove(3)
L3 = list_create1(5,7,11,37)
assert L1 == L3            # Assertion
assert L3.index(37) == 3   # Assertion
L3.reverse()
L4 = list_create1(37,11,7,5)
assert L3 == L4            # Assertion
```

Procedural notation

- The functional notation
  `foo(x)`, `bar(x,y)`, `baz(x,y,z)`
  was invented by the Mathematician Leonard Euler at 1748
- There is no specific sacred or holly reason for this notation! Euler could at the same time use ‘<x>f’ or ‘f-x-’ or many other possible notations
- We already have exceptions to this rule when we write x+y instead of add(x,y), or x**n instead of power(x,n).
- Python writes: `L = [a, b, c]` instead of `list_create(a,b,c)`

Python List Constructors

- The most basic constructor for lists is:
  `L = [x0, x1, x2, ..., xn]`
  - It corresponds to: `list_create1(x0, x1, x2, …, xn)`
- The other constructor is list(container_object)
- Lists can be created from a variety of other container objects such as: set, array, dictionaries, and other list

Naming Issues

- Specification name and Implementation name do not have to be the same!
- For example, in Python, the call

```
L = list_create1(e0, e1, e2,..., en-1)
```

has been changed to:

```
L = [e0, e1, e2, …, en-1]
```

and the call

```
L.contains(e)
```

Has been changed to:

```
e in L
```

- The only essential thing is that the name conveys the meaning of the operation, and the operation is precisely defined

Python List Syntactic Sugar

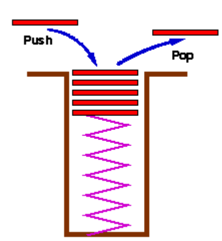
Operation	Python Syntactic Sugar
`L=list_create1(a,…,b)`	`L = [a, ...,b]`
`L=list_create2(other)`	`L = list(other)`
`L.contains(e)`	`e in L`
`L.item(i)`	`L[i]`
`L.size()`	`len(L)`
`L.slice(i,j)`	`L[i:j]`
`L.equals(other)`	`L == other`
`L.remove_by_index(i)`	`del L[i]`
`L1.add(L2)`	`L1+L2`
`L.mul(n)`	`L*n or n*L`

A Word About Destructors

- Some object oriented languages (like C++) contain an additional method type: **destructor**
- A **destructor** is a method for destroying (or terminating) an object
- A destructor usually frees the memory that was used by the object and may also perform additional cleanup and finalization tasks
- In such languages, failure to delete objects at the right time can lead to serious memory problems, and even to program crash
- Modern object oriented languages such as **Java**, **C#**, and **Python**, contain a mechanism (called “garbage collection”) which automatically deletes objects as soon as they’re not needed anymore
- We will therefore not bother about this concept anymore in this course
- In extreme cases if needed you can use the Python del operator to delete objects: `del L`

Stack Abstract Data Type
Description

- Sequence type (container) in which elements are pushed and popped out from the top end
- AKA LIFO – Last In First Out



Stack Abstract Data Type
Interface

- `s = Stack()` **Constructor**
  - Create a new empty stack
- `s.push(item)` **Mutator**
  - Add an item to the top of the stack
- `s.pop()` **Mutator**
  - Pop an item to the top of the stack
- `s.peek()` **Accessor**
  - Return the item to the top of the stack (don't pop it!)
  - Return `None` if stack is empty (this is not a good idea, why?)
- `s.size()` **Accessor**
  - Return the number of items in the stack
- `s.is_empty()` **Accessor**
  - Return `True` if stack is empty, `False` if stack is non-empty

Stack Test 1

```
s = Stack()
s.push(1)
s.push(1)
s.push(2)
assert s.pop() == 2
assert s.pop() == 1
assert s.pop() == 1
assert s.is_empty()
```

Stack Test 2

```
s = Stack()
expression = "a+(b*(c+d)+x*(y-a)+z)-n“

# Check if left/right parens are
# legally balanced
for char in expression:
    if char == '(':
        s.push('L')
    if char == ')':
        if s.peek() == 'L':
            s.pop()
        else:
            s.push('R')

assert s.is_empty()
```


Stack Test 2: Stack Frames

```
s = Stack()
expression = "a+(b*(c+d)+x*(y-a)+z)-n“

Frame 0: empty stack
Frame 1: L
Frame 2: L, L
Frame 3: L
Frame 4: L, L
Frame 5: L
Frame 6: empty stack
```

Stack Implementation

```
class Stack :
  def __init__(self) :
    self.items = []

  def push(self, item) :
    self.items.append(item)

  def pop(self) :
    return self.items.pop()

  def peek(self):
    return self.items[-1]

  def is_empty(self) :
    return (self.items == [])
```

http://www.greenteapress.com/thinkpython/thinkCSpy/html/chap18.html

Part 2

DICTIONARY, SET, TABLE

Set Abstract Data Type

- A set data structure is a container of objects with the following properties
- Elements are unique. A set cannot contain two instances of the same element (like a list or an array)
- Elements do not have an order. All we know about an element **e** is whether it belongs or does not belong to a set
- Set data structure originate in the mathematical theory of Set Theory, but have useful applications in computer science

The Set Abstract Data Type
Object Oriented Design – Part 1

- `s = set_create1()`
  - Create a new empty set s
- `s = set_create2(container)`
  - Create a new set s from other set or any another container object
- `s.add(e)`
  - Add element e to set s
- `s.remove(e)`
  - Remove an element `e` from the set `s`
  - If e is not in s, raise an error
- `s.contains(e)`
  - Check if element `e` belongs to the set `s`
  - Returns: boolean `True` or `False`
  - Efficiency requirement: should be very fast! O(1)

The Set Abstract Data Type
Object Oriented Design – Part 2

- `s.union(container)`
  - Set union of s elements with elements in container
  - Container can be any Python container (including a dictionary!)
  - Does not modify s! Just return the result!
- `s.intersect(container)`
  - Intersection of s with any other Python container
  - Does not modify s! Just return the result!
- `s.subtract(container)`
  - Remove from s all elements in container
- `s.discard(e)`
  - Remove an element from a set if it is a member
  - If the element is not a member, do nothing
- `s.clear()`
  - Remove all elements of s (make s an empty set)

The Set Abstract Data Type
Object Oriented Design – Part 3

- `s.copy()`
  - Create a copy of s
  - Same as: `s2 = set(s)`
- `s.issubset(container)`
  - Check if s is a subset of container. Return: `True` or `False`.
  - Container can be any Python container (even a dictionary!)
- `s.isdisjoint(container)`
  - Check if `s` is disjoint to container (no common elements)
- `s.issuperset(container)`
  - Check if s includes container elements. Return: `True` or `False`.
- `s.pop()`
  - Remove an arbitrary element from `s`
  - Raise an error if `s` is empty

The Set Abstract Data Type
Object Oriented Design – Part 4

- `s.equal(s2)`
  - check if two sets are equal (same as: s == s2)
- `s.update(container1, container2, ..., containern-1)`
  - Add elements from other containers
- `s.iterator()`
  - Create an iterator object for iterating over the set elements
- `s.size()`
  - Get the size of s (number of elements)

Set Test

```
s1 = set_create1()
s1.add(17)
s1.add(18)
s1.add(18)                # adding 18 twice!
assert s1.contains(17)
assert s1.size() == 2
A = list_create1(2, 4, 6, 8, 2, 6)    # list container
B = list_create1(4, 8, 2, 6)          # list container
s2 = set_create2(A)
s3 = set_create2(B)
assert s2.equals(s3)
s3.add(100)
assert s2.issubset(s3)
s3.remove(100)
assert s2.equals(s3)
```

This is just a small example of how ADT regression test should look like.
A real test should cover all the ADT operations from all possible angles.
After every implementation change, the test should pass.

Set Implementation as List

- Python set is already implemented as a C hash table
- But it could also be implemented by the standard Python List data structure
- The implementation is available at this link:
  Link to Set implementation as list
- You also need to download
  Link to three set tests

The Dictionary (Map) ADT
Object Oriented Design – Part 1

- The dictionary data structure store key/value pairs
- Its critical advantage is the speed for getting a value from a key! We'll later explain what O(1) is and why this is the fastest time
- `d = dict_create1()`
  - Create a new empty dictionary
- `d = dict_create2(key1: value1, key2: value2, ...)`
  - Create a new dictionary from a list of key/value pairs
- `d = dict_create3(map_object)`
  - Create a new dictionary from other map_object
- `d = dict_create4(iterable)`
  - Create a new dictionary from an iterator which returns key/value pairs

The Dictionary (Map) ADT
Object Oriented Design – Part 2

- `d.contains(key)`
  - Check if dictionary d contains a key
- `d.add(key, value)`
  - Adds a new key/value pair to the dictionary if the key is not already there
  - If the key already there, then the old value is replaced with the new value
- `d.remove(key)`
  - Remove key (and its associated value) from the dictionary
- `d.get(key)`
  - Get the value associated with key
- `d.iterator()`
  - Creates and returns an iterator that can be used to iterate over the keys
- `d.copy()`
  - Copy a dictionary

The Dictionary (Map) ADT
Object Oriented Design – Part 3

- `d.clear()`
  - Remove all keys and values
- `d.items()`
  - Return a list of all key/value pairs stored in the dictionary
- `d.pop(key)`
  - Return the value associated with key, and remove key (and its associated value) from the dictionary
- `d.popitem()`
  - Remove an arbitrary key/value pair from the dictionary and return it
  - Raise an error if dictionary empty
- `d.update(map_object)`
  - Extend dictionary with additional key/value pairs from map_object

Python Dictionary

- Python provides a very efficient and easy to use dictionary class
- There are two ways to create and initialize a Python dictionary
- Python dictionary has all the standard dictionary methods and more

```
# Create a new empty dictionary
d = dict()

# Create and initialize a dictionary
d = dict(name='Avi Cohen', age=32, id=5802231, address='Hayarden 43, Gedera')

# Alternative constructors:
# Create a new empty dictionary
d = {}

# Create and initialize a dictionary
d = {name: 'Avi Cohen', age: 32, id: 5802231, address: 'Hayarden 43, Gedera'}
```


Python Dictionary Methods

```
print "Avi's age is:", d['age']
print "Avi's address is:", d['address']
print "Avi has moved to a new town:"
d['address'] = 'Hayarkon 25, Haifa'
del d[key]    # deletes the mapping with that key from d
len(d)        # return the number of keys
x in d        # return True if x is a key of d
x not in d    # return False if x is not a key of d
d.keys()      # returns a list of all the keys in the dictionary
d.values()    # returns a list of all the values in the dictionary
```

MultiSet Abstract Data Type

- A multiset is a set in which elements may occur several times
- Example: words in a text file. It’s not enough to know the set of words, we’re also interested in how many times each word occurs?
- As with set, multiset elements are not ordered. All we know about an element e is the number of times it appears
- In some implementations, the number of occurrences can be 0 and even negative !

The MultiSet Abstract Data Type
Object Oriented Design – Part 1

- `m = multiset_create1()`
  - Create a new empty set s
- `m = multiset_create2(container)`
  - Create a new set s from other set or any another container object
- `m.add(e, n=1)`
  - Add element e with n occurrences
- `m.remove(e)`
  - Remove an element `e` from the multiset `m`
  - Be silent If e is not in s (usual behavior)
- `m.contains(e)`
  - Check if element `e` belongs to the multiset `m`
  - Returns: boolean `True` or `False`
  - Efficiency requirement: should be very fast! O(1)

The MultiSet Abstract Data Type
Object Oriented Design – Part 2

- `m.subtract(container)`
  - Remove from s all elements in container
- `s.discard(e)`
  - Remove an element from a set if it is a member
  - If the element is not a member, do nothing
- `s.clear()`
  - Remove all elements of s (make s an empty set)

The Table Abstract Data Type

- The Table data type is the most important data type in the field of databases (“relational databases”), spread sheet software (like Microsoft Excel), and also in mathematics (for representing a **matrix** or a **two-dimensional array** of numerical data). In VLSI used for **Gate Arrays** and **FPGA**
- Data in a table is organized into **rows** and **columns**.
- Data element is accessed by two indices:
  - row index
  - column index
- This pair of indices (i,j) is called a cell

	A	B	C	D	E
1	1	15	11	5	12
2	4	4	29	2	2
3	6	54	4	7	88
4	21	1	8	40	11
5	14	5	19	2	4
6	2	11	10	6	17
7	7	24	8	7	60

The Table Abstract Data Type

- `t = Table(nrows, ncols)`
  - Create a new table with number of rows = `nrows`, number of columns = `ncols`
- `t.numRows()`
  - Return the number of rows in the table `t`
- `t.numCols()`
  - Return the number of columns in the table `t`
- `t.clear(value)`
  - Clear and set all elements to `value`
- `t.setitem(i, j, value)`
  - Sets (or modifies) the content of the cell `(i,j)`
  - Both indices must be within valid bounds: `0<=i<nrows, 0<=j<ncols`
- `t.getitem(i, j)`
  - Get the content of the cell `(i,j)`
  - Both indices must be within valid bounds: `0<=i<nrows, 0<=j<ncols`

"Syntactic Sugar"

- Most programming languages provide the more traditional syntax (based on a very long mathematical history)

```
v = t[i,j]   ⟺  v = t.getitem(i,j)
t[i,j] = v   ⟺  t.setitem(i,j,v)
```

sometimes round parens are used instead of brackets

```
v = t(i,j)   ⟺  v = t.getitem(i,j)
t(i,j) = v   ⟺  t.setitem(i,j,v)
```

- Providing shorter more intuitive syntax is sometimes called "Syntactic Sugar"

Table Indexing

	Column 0	Column 1	Column 2	Column 3
Row 0	T[0,0]	T[0,1]	T[0,2]	T[0,3]
Row 1	T[1,0]	T[1,1]	T[1,2]	T[1,3]
Row 2	T[2,0]	T[2,1]	T[2,2]	T[2,3]

3x4 table

"Syntactic Sugar" in C

- The C programming language supports multi-dimensional arrays (same type) but is using a different kind of syntactic sugar:

```
v = a[i][j]   ⟺   v = a.getitem(i,j)
a[i][j] = v   ⟺   a.setitem(i,j,v)
```

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

Table Implementations - C

- The Table ADT can be implemented in several ways
- In C, a two dimensional array is implemented as an “array of arrays”

```
typedef struct {
    double value;
} cell ;

# Static allocation
cell table[30][40];

# Dynamic allocation
cell **table = (cell **)malloc(30 * sizeof(cell*)) ;
for (col = 0; col < 40; ++col)
    table[col] = (cell *)malloc(40 * sizeof(cell)) ;
```


Table Implementations - C

- As you can see, the C **two-dimensional array** requires a single type for all cells
- Table code must be duplicated for every new cell type
- The worst part is that it does not include Table methods
- Methods must be defined separately for every new cell type

```
# Implementing the 'clear' method:

void clear(cell **table, int numrows, int numcols, cell value)
{
    int row, col ;

    for(row = 0; row < numrows; row++)
        for(col = 0; col < numcols; col++)
            table[row][col] = value ;
}
```

Table Implementation 2 – C
Procedural Design

- Idea: cell (row,col) can be encoded by a single integer:
  `row * numcols + col`
- We therefore can represent a `numrows*numcols` table by a single **1-dimensional** array:

```
typedef struct {
    double value;
} cell ;

# Dynamic allocation of a 3x4 table
cell* table = (cell *)malloc(3*4 * sizeof(cell));
```

- In spite of the extra multiplication/addition needed for indexing, this approach has big advantage from a CPU cache point of view!

Table Python Implementation 1 (List)
Procedural Design

- Table can be implemented as a list of lists
- Cell values can be of any mixed types
- The numRows() and numCols() methods are easily defined as: len(table) and len(table[0])

```
table = [ [0,1,2,3] , [4,5,6,7] , [8,9,10,11] ]

# setitem method:
table[2][0] = 1978

def clear(table, value):
    numrows = len(table)
    numcols = len(table[0])
    for row in range(numrows):
        for col in range(numcols):
            table[row][col] = value
```

Table Python Implementation 1 (List)
Procedural Design

- This is essentially the same as the C “array of arrays” idea

```
table = [ [0,1,2,3] , [4,5,6,7] , [8,9,10,11] ]

# setitem method:
table[2][0] = 1978

def clear(table, value):
    numrows = len(table)
    numcols = len(table[0])
    for row in range(numrows):
        for col in range(numcols):
            table[row][col] = value
```


Table Python Implementation 2 (Dict)
Procedural Design

- Table can also be implemented as a dictionary whose keys are cell indices (row,col)
- We can use the dictionary to store additional information like the number of rows and columns:

```
def new_table(nrows, ncols, value=0):
    table = dict()          # table is a dictionary !
    for row in range(nrows):
        for col in range(ncols):
            table[row,col] = value
    table['nrows'] = nrows  # save num rows in dict !
    table['ncols'] = ncols
    return table
```

Table Python Implementation 2 (Dict)
Procedural Design

```
# setitem method:
# table[2][0] = 1978

def setitem(table, row, col, value):
    table[row,col] = value

def getitem(table, row, col):
    return table[row,col]

def numRows(table):
    return table['nrows']

def numCols(table):
    return table['ncols']
```


Table Python Implementation 2 (Dict)
Procedural Design

```
def clear(table, value=0):
    nrows = numRows(table)
    ncols = numCols(table)
    for row in range(nrows):
        for col in range(ncols):
            table[row,col] = value

def printTable(table):
    nrows = numRows(table)
    ncols = numCols(table)
    for row in range(nrows):
        for col in range(ncols):
            print "table[%d,%d] = %s" % (row, col, table[row,col])
```

Table Python Implementation 2 (Dict)
Procedural Design

- Download the table2.py file and run the test below

```
def test1():
    table = new_table(3,4)
    clear(table,17)
    table[0,0] = 40
    table[2,3] = 50
    printTable(table)
```


Table Python Implementation 3 (List)
Object Oriented Design

- To fully match the Table ADT we need to do it in an OOD way
- We will show two different ways:
  - List of lists representation
  - Dictionary representation
- There are of course many other ways to implement a Table ADT, some are more efficient, but the point of this discussion is to make a clear distinction between **Interface** and **Implementation**!

Class: Table
numRows()
numCols()
setitem(row,col,value)
getitem(row,col)
clear(value)

Table Python Implementation 3 (List)
Object Oriented Design

```
class Table:
    def __init__(self, nrows, ncols, value=0):
        self.nrows = nrows
        self.ncols = ncols
        self.list = list()
        for r in range(self.nrows):
            row = ncols * [value]
            self.list.append(row)

    def setitem(self, row, col, value):
        self.list[row][col] = value

    def getitem(self, row, col):
        return self.list[row][col]

    def numRows(self):
        return self.nrows

    def numCols(self):
        return self.ncols
```


Table Python Implementation 3 (List)
Object Oriented Design

```
class Table:

    # . . . continued

    def clear(self, value=0):
        for row in range(self.nrows):
            for col in range(self.ncols):
                self.list[row][col] = value

    def __str__(self):    # print method !
        tbl = ""
        for row in range(self.nrows):
            for col in range(self.ncols):
                tbl += "table[%d][%d] = %s, " % (row, col, self.list[row][col])
            tbl += "\n"
        return tbl
```

Table Python Implementation 3 (List)
Object Oriented Design

- Here is a small test for testing our class
- Of course, a real life test should be more extensive !
- Download the source code and run the test

```
def test1():
    table = Table(4,5)
    table.clear(17)
    table.setitem(0,0,40)
    table.setitem(3,2,80)
    print table
    print "Number of rows =", table.numRows()
    print "Number of columns =", table.numRows()
```


Table Python Implementation 4 (Dict)
Object Oriented Design

```
class Table:
    def __init__(self, nrows, ncols, value=0):
        self.nrows = nrows
        self.ncols = ncols
        self.dict = dict()
        for row in range(self.nrows):
            for col in range(self.ncols):
                self.dict[row,col] = value

    def setitem(self, row, col, value):
        self.dict[row,col] = value

    def getitem(self, row, col):
        return self.dict[row,col]

    def numRows(self):
        return self.nrows

    def numCols(self):
        return self.ncols
```

Table Python Implementation 3 (List)
Object Oriented Design

```
class Table:

    # . . . continued

    def clear(self, value=0):
        for row in range(self.nrows):
            for col in range(self.ncols):
                self.dict[row,col] = value

    def __str__(self):                  # print method !
        tbl = ""
        for row in range(self.nrows):
            for col in range(self.ncols):
                tbl += "table[%d,%d] = %s, " % (row, col, self.dict[row,col])
            tbl += "\n"
        return tbl

    def __setitem__(self, key, value):   # overload the [] operator
        self.dict[key] = value

    def __getitem__(self, key):          # overload the [] operator
        return self.dict[key]
```


Table Python Implementation 3 (List)
Object Oriented Design

- Same test1() from implementation 3 should give identical result!
- We also add a test2() for testing the brackets overloading

```
def test1():
    table = Table(4,5)
    table.clear(17)
    table.setitem(0,0,40)
    table.setitem(3,2,80)
    print table
    print "Number of rows =", table.numRows()
    print "Number of columns =", table.numRows()

def test2():
    table = Table(4,5)
    table.clear(17)
    table[0,0] = 40
    table[3,2] = 80
    print table[3,2]
    print table
```

Part 3

SEARCHING AND SORTING

Searching

- Searching is the process of finding particular information from a collection of data based on specific criteria
- Search operations can be performed on every collection data structure (string, array, list, stack, dictionary, set, …)
- Search operation accepts two inputs:
  - Collection (or sequence) object
  - Search key
- Search key can have several forms
  - An item that we want to find in a list
  - Part of an item to search
  - Multiple parts for searching matching items (Google search)

Search Modes

- There are four different types of search operations
- **In or out**: Checking if the collection contains or does not contain the item
  Example: **item in L**
- **First match**: Finding the first occurrence of the key and reporting its location in the collection
  Example: **List.index(item)**
- **All matches**: Finding all the items in the collection that match the key
  Example: **fnmatch.filter(Names, “Dan*”)**
- **Partial matches**: Find the first n items that match the key

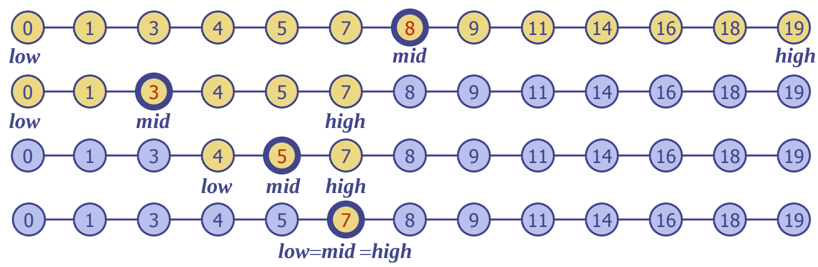
Linear Search (return first match)

```
def linear_search(List, item):
    n = len(List)
    for i in range(n):
        if item == List[i]:
            return i
    return -1
```

- Linear search is already implemented by the list index method except that when the item is not in the list you get an error
- The run time order of the linear search algorithm is O(n)
- Question: suppose that our sequence is sorted, could this help to speed the search process?

Binary Search

```
L = [0, 1, 3, 4, 5, 7, 8, 9, 11, 14, 16, 18, 19]
L is a sorted list in increasing order!
binary_search(L, 7)
low = 0, high = len(L) = 12
mid = (low+high) / 2 = 6
```



Binary Search Algorithm (Recursive)

```
def binary_search_rec(List, item, low=0, high=None):
    if high is None:
        high = len(List)

    if low >= high:    # empty list
        return -1

    mid = (low + high) / 2
    mid_value = List[mid]
    if item < mid_value:
        return binary_search_rec(List, item, low, mid)
    elif item > mid_value:
        return binary_search_rec(List, item, mid+1, high)
    else:
        return mid
```

Binary Search Algorithm

```
def binary_search(List, item, low=0, high=None):
    if high is None:
        high = len(List)

    while low < high:
        mid = (low + high) / 2
        mid_value = List[mid]
        if mid_value < item:
            low = mid+1
        elif mid_value > item:
            high = mid
        else:
            return mid
    return -1
```


SORTING

- Although binary search run time is fast O(log n), it depends on <u>sorting the sequence </u>!!!
- **Questions:**
  - What is the cost of sorting a sequence container?
  - What sorting algorithms do we have?
  - And which are the best sorting algorithms?
- In the next slides we will explore several (out of many) sorting algorithms and check their run time and quality

Why Sorting?

- Sorting is among the most important, and well studied computational problems
- Data sets are often stored in sorted order, for example, to allow for efficient searches with the binary search algorithm
- Many advanced algorithms rely on sorting as a subroutine

Bubble Sort

- [YouTube Bubble Sort Dance]
- The simplest and most intuitive sorting algorithm

```
# L is a list of integers that we want to sort

def bubble_sort(L):
    N = len(L)
    while True:
        sorted = True
        for i in range(0,N-1):
            if L[i+1] < L[i]:
                sorted = False
                L[i], L[i+1] = L[i+1], L[i]
        if sorted:
            return
```

Bubble Sort – version 2

- Here is a different version of Bubble Sort:

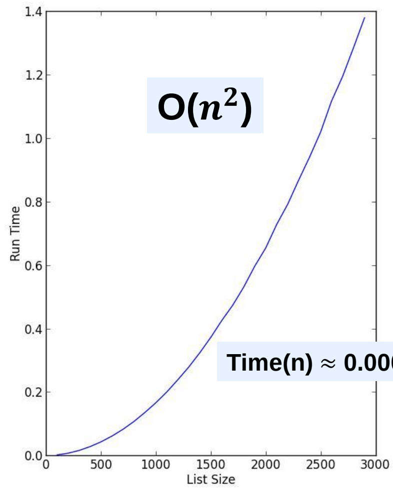
```
# L is a list of integers

def bubble_sort2(L):
    N = len(L)
    for i in range(0,N-1):
        for j in range(i+1, N):
            if L[j] < L[i]:
                L[i], L[j] = L[j], L[i]
```


Bubble Sort Test

```
def bubble_sort_test():
    for i in range(24):
        L = range(0,10)
        random.shuffle(L)
        print "L = ", L
        bubble_sort(L)
        print "Bubble sort:", L
        assert L == range(0,10)
        raw_input("Press any key to continue:")
```

Bubble Sort Run Time Data

Run time results obtained by running
Python 2.7.5 on a core-i7 ASUS laptop



List Size	Run Time (seconds)
100	0.0017
200	0.007
300	0.0157
400	0.0278
500	0.0429
600	0.0611
700	0.0824
800	0.1071
900	0.1355
1000	0.1663
1100	0.2003
1200	0.2387
1300	0.2789
1400	0.3238
1500	0.3723
1600	0.4252
1700	0.4737
1800	0.5308
1900	0.5964
2000	0.6538
2100	0.7279
2200	0.7914
2300	0.8676
2400	0.9406
2500	1.0191
2600	1.1171
2700	1.1941
2800	1.2853
2900	1.3791

Bubble Sort – Run Time Analysis

- Another name for **O(n^2) is** “Quadratic Time Complexity” which is considered industry-bad unless the input size is expected to be small in almost all practical cases
- The above 30 experiments allows us to predict what will happen if our list size grows
- Lists of size 10M are not very rare. For example, chip floor-plan models may contain more than 1 billion transistors - 6 months run time for a 10M size list is of course unacceptable

List Size	Run Time (seconds)
10000	16.6 seconds
100000	1660 seconds
1000000	166000 seconds
10M	16600000 seconds ~ 6 months

Time(n) ≈ 0.000000166*n^2

Bubble Sort – Average Time Tests

- Python code for the Bubble sort algorithm and the tests code can be downloaded from:
  http://brd4.braude.ac.il/~samyz/cgi-bin/view_file.py?file=DSAL/CODE/bubble_sort.py
- Here is a typical routine for calculating average run time by generating many random shuffles of a list

```
import random

def bubble_sort_average_time(list_size, num_tests):
    times = list()
    L = range(0, list_size)

    for i in range(num_tests):
        random.shuffle(L)
        t0 = time.time()
        bubble_sort(L)
        t1 = time.time()
        t = t1-t0
        times.append(t)

    return sum(times)/num_tests
```


Bubble Sort – Average Time

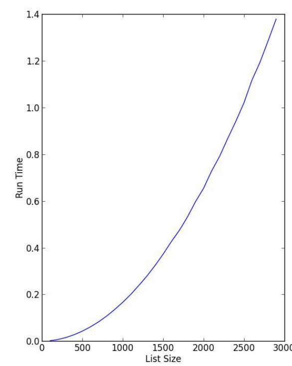
- Code for computing average time is also in:
  http://brd4.braude.ac.il/~samyz/cgi-bin/view_file.py?file=DSAL/CODE/bubble_sort.py
- We expect the student to copy paste and apply it to other algorithms!

```
# Create num_tests lists of size list_size and compute
# average time for doing bubble_sort on these lists

def bubble_sort_average_time(list_size, num_tests):
    times = list()
    L = range(0, list_size)

    for i in range(num_tests):
        random.shuffle(L)
        t0 = time.time()
        bubble_sort(L)
        t1 = time.time()
        t = t1-t0
        times.append(t)

    return sum(times)/num_tests
```

Bubble Sort – Average Time Graph

- Code for drawing average time graphs is also in:
  http://brd4.braude.ac.il/~samyz/cgi-bin/view_file.py?file=DSAL/CODE/bubble_sort.py
- We expect the student to apply it to other algorithms!

```
def bubble_sort_runtime_graph():
    import matplotlib.pyplot as pyplot
    Size = [100*i for i in range(1,30)]
    Time = list()
    for N in Size:
        print "N=", N
        t = bubble_sort_average_time(N,16)
        t = round(t,4)
        Time.append(t)

    pyplot.plot(Size,Time)
    pyplot.xlabel('List Size')
    pyplot.ylabel('Run Time')
    pyplot.show()
    header = ('List Size', 'Run Time (seconds)')
```



The Halting Problem

- Could there be a special list on which Bubble sort runs forever ?
- The general halting problem: given an algorithm and an input, can we determine whether the algorithm will eventually halt or will run forever?
- Being able to <u>prove</u> that a given algorithm will halt for <u>all</u> its possible <u>inputs</u> is a critical !
- Proving that an algorithm must halt for all its inputs is usually very hard, and in many cases <u>impossible</u>.
- It may involve very complicated mathematical proofs and/or very long and expensive computations (e.g., QA, verification of an VLSI unit)

Why Bubble Sort Always Halt?

- We'll prove that for the second version
- Idea: prove an **invariant** is true for all iterations
  - It holds initially
  - If it holds at stage i, then it holds for stage i+1
  - Eventually must hold for all the list
- For bubble sort 2, the invariant is:
  at iteration i, the sub-list L[0:i] is sorted and any element in L[i:n] is greater or equal to any element in L[0:i]
- Since i is increasing, it eventually reaches n, and the algorithm halts

Why Bubble Sort Always Halt?

- For bubble sort 1, the invariant starts from the end (watch the Hungarian dance again …)
- The largest element must always "float" to the top, after which it will never move again!
- Therefore the problem is reduced to L[0,n-1]
- This proves that by at most n iterations of the loop, the list must be sorted. The inner loop also has n iterations, so by a total of n**2 steps the sorting is done
- Example: how many swaps are needed to sort the list
  L = [n, n-1, n-2, n-3, …, 2, 1, 0] ?
- This example demonstrates why bubble sort is O(n**2)

Selection Sort

- Yet one more intuitive method for sorting a list
- For simplicity, let L be a list of integers whose size is n=len(L)
- The idea in selection sort is:
  - Find the minimal element of L[0], L[1], …, L[n-1] and then make it the first (L[0])
  - Find the minimal element of L[1], L[2], …,L[n-1] and make it the second element (L[1])
  - Find the minimal element of L[2], L[3], …,L[n-1] and make it the third element (L[2])
  - Repeat this process until the list is fully sorted

Selection Sort: the idea

```
L = [7, 2, 8, 4, 6, 5, 1, 3]
    [1, 2, 8, 4, 6, 5, 7, 3]
    [1, 2, 8, 4, 6, 5, 7, 3]
    [1, 2, 3, 4, 6, 5, 7, 8]
    [1, 2, 3, 4, 6, 5, 7, 8]
    [1, 2, 3, 4, 5, 6, 7, 8]
    [1, 2, 3, 4, 6, 5, 7, 8]     Sorted!
```

Selection Sort: simpler version

```
1. Start with i=0
2. For every j from i+1 until n-1, if L[j] is
   smaller than L[i], swap L[i] and L[j]
3. Increment i (i = i+1)
4. Repeat step 2 until i=n-1
```

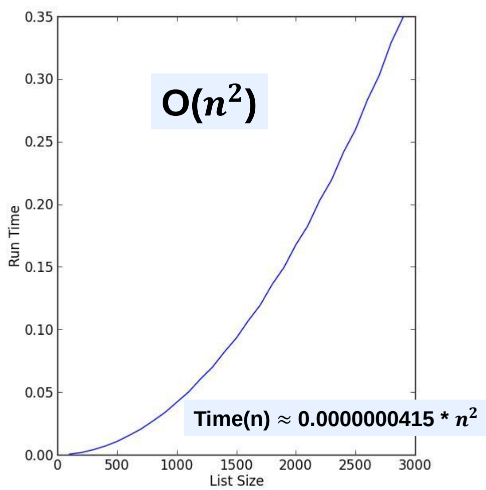
- This is a slightly different version than the heuristic one (two slides back)
- In this version we also compute the minimal value as part of the algorithm (instead of relying on an external method)

Selection Sort: Algorithm

```python
def selection_sort(L):
    n = len(L)
    for i in range(n):
        min_index = i
        for j in range(i + 1, n):
            if L[j] < L[min_index]:
                min_index = j
        L[i], L[min_index] = L[min_index], L[i]
```

Selection Sort Run Time

Run time results obtained by running
Python 2.7.5 on a core-i7 ASUS laptop



List Size	Run Time (seconds)
100	0.0005
200	0.0017
300	0.004
400	0.0069
500	0.0106
600	0.0154
700	0.0205
800	0.0269
900	0.0336
1000	0.0419
1100	0.0501
1200	0.0605
1300	0.0699
1400	0.082
1500	0.0931
1600	0.1069
1700	0.1193
1800	0.1358
1900	0.1495
2000	0.1676
2100	0.1827
2200	0.203
2300	0.2194
2400	0.2415
2500	0.2594
2600	0.2831
2700	0.3029
2800	0.329
2900	0.3491

Selection Sort – Run Time Analysis

- Although Selection sort is 4x faster that Bubble sort, it's time complexity is still **O(n^2)** ("Quadratic Time Complexity") which is means it is essentially as bad as Bubble sort ☹
- This is obvious from the following table, which shows that for sorting a 40M random list may take about 2 years

List Size	Run Time (seconds)
10000	4.15 seconds
100000	415 seconds
1000000	41510 seconds
40M	66,416,171 seconds ~ 2 years

Time(n) ≈ 0.0000000415 * n^2

Average Time Tests

- Python code for the Selection sort algorithm and the tests code can be downloaded from:
  http://brd4.braude.ac.il/~samyz/cgi-bin/view_file.py?file=DSAL/LAB/selection_sort.py
- Here we introduce a more general function for computing average time which can be used by any other sorting algorithm!

```
import random

def sort_average_time(sorter, list_size, num_tests):
    times = list()
    L = range(0, list_size)

    for i in range(num_tests):
        random.shuffle(L)
        t0 = time.time()
        sorter(L)
        t1 = time.time()
        t = t1-t0
        times.append(t)

    return sum(times)/num_tests
```


Sort Average Time

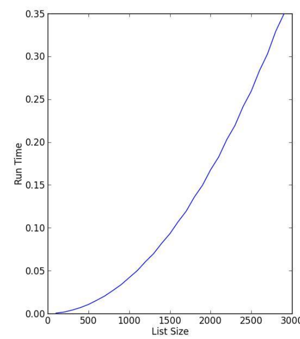
- Code for computing average time is also in:
  http://brd4.braude.ac.il/~samyz/cgi-bin/view_file.py?file=DSAL/LAB/sort_bench.py
- The following code can be used for any sort algorithm !

```
# sorter is any function that sorts a list
# Create num_tests lists of size list_size and compute
# average time for doing bubble_sort on these lists

def sort_average_time(sorter, list_size, num_tests):
    times = list()
    L = range(0, list_size)

    for i in range(num_tests):
        random.shuffle(L)
        t0 = time.time()
        sorter(L)
        t1 = time.time()
        t = t1-t0
        times.append(t)

    return sum(times)/num_tests
```

Sort – Average Time Graph

- Code for drawing average time graphs is also in: http://brd4.braude.ac.il/~samyz/cgi-bin/view_file.py?file=DSAL/CODE/sort_bench.py
- The following code can be used for any sort algorithm !

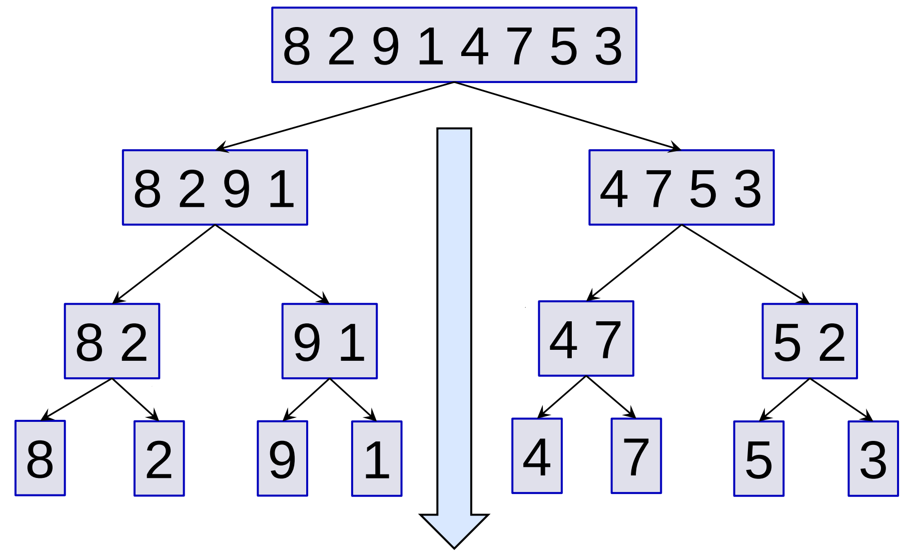
```
def sort_runtime_graph(sorter, n=30, ntests=16):
    import matplotlib.pyplot as pyplot
    import sys
    Sizes = [100*i for i in range(1,n)]
    Times = list()
    for N in Sizes:
        print "N=", N
        t = sort_average_time(sorter, N, ntests)
        t = round(t,4)
        Times.append(t)

    pyplot.plot(Sizes, Times)
    pyplot.xlabel('List Size')
    pyplot.ylabel('Run Time')
    pyplot.show()
```



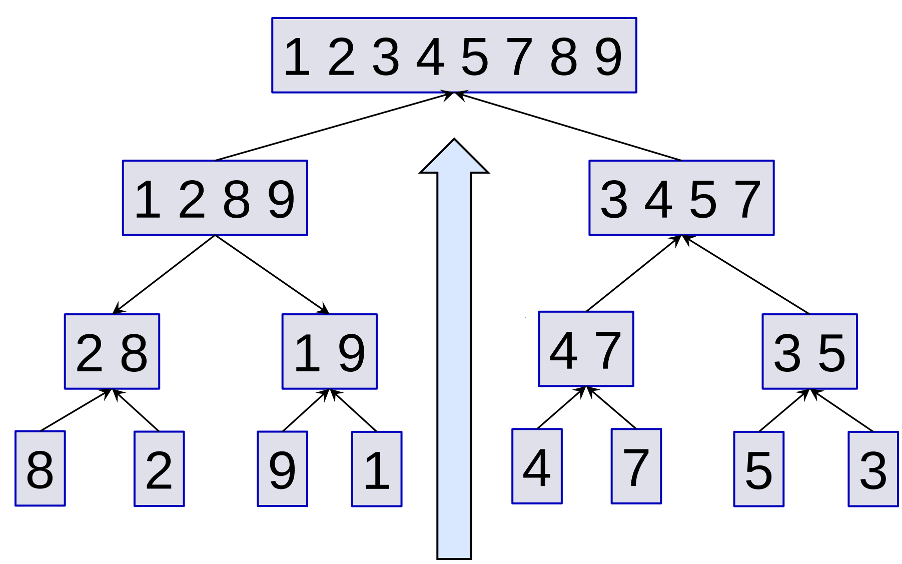
MERGE SORT / Divide and Conquer

- **Divide**
  - If the sequence is too small (1 or two elements) then sorting is easy
  - If the sequence is big, divide it to two parts and solve each part separately
- **Conquer**
  Recursively solve the subproblems associated with the subsets
- **Combine**
  Take the solutions to the sub problems and merge them into a solution to the original problem

Example: Divide



Example: Merge



The merge_sort algorithm

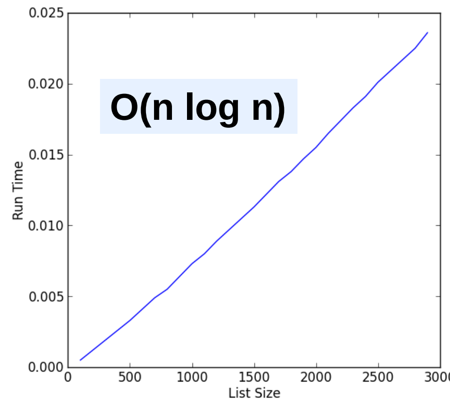
```
def merge_sort(L):
    n = len(L)
    if n <= 1:
        return
    mid = n / 2
    A = L[0:mid]
    B = L[mid:]
    merge_sort(A)
    merge_sort(B)
    M = merge(A,B)
    for i in range(n):
        L[i] = M[i]
```


The merge algorithm

```
def merge(A, B):
    "merge sorted lists A and B. Return a sorted result"
    result = []
    i = 0
    j = 0

    while True:
        if i >= len(A):          # If A is done,
            result.extend(B[j:])  # Add remaining items from B
            return result         # And we're totally done

        if j >= len(B):          # Same again, but swap roles
            result.extend(A[i:])
            return result

        # Both lists still have items, copy smaller item to result.
        if A[i] <= B[j]:
            result.append(A[i])
            i += 1
        else:
            result.append(B[j])
            j += 1
```

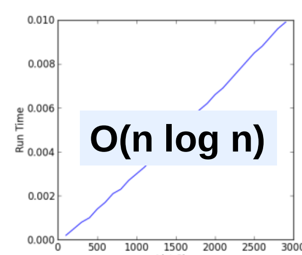
Merge Sort Run Time Benchmark

Merg Sort	Algorithm
List Size	Run Time (seconds)
600	0.0041
700	0.0049
800	0.0055
900	0.0064
1000	0.0073
1100	0.008
1200	0.0089
1300	0.0097
1400	0.0105
1500	0.0113
1600	0.0122
1700	0.0131
1800	0.0138
1900	0.0147
2000	0.0155
2100	0.0165
2200	0.0174
2300	0.0183
2400	0.0191
2500	0.0201
2600	0.0209
2700	0.0217
2800	0.0225
2900	0.0236



Time(n) ≈ 0.000001021 * n * log n

Merge Sort Run Time



Time(n) ≈ 0.0000004282 * n * log n

List Size	Run Time (seconds)
10000	0.0940 seconds
100000	1.1754 seconds
1000000	14.1056 seconds
10M	164.5657 seconds (bubble was 6 months !!!)
1000M	21158 seconds - less than 6 hours vs. 5200 years with bubble sort

QUICK SORT

- Invented by Tony Hoare 1960 (Moscow Univ.)
- **Divide**
  - The first item is selected as the pivot, p. The pivot value is used to partition the list to two sub-lists A and B, such that
    - A consists of all elements less than p
    - B consists of all elements bigger or equal to p
- **Conquer**
  Recursively solve the sub-problems by applying `quick_sort` to A and B
- **Combine**
  Combine the solutions of `quick_sort(A)` and `quick_sort(B)` by a simple concatenation (A then B)

The partition algorithm

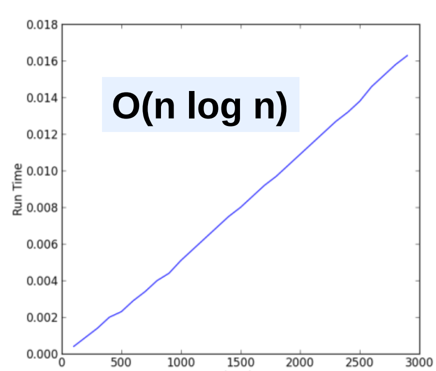
```
def partition(L, pivot):
    A = []
    B = []
    for element in L:
        if element < pivot:
            A.append(element)
        else:
            B.append(element)
    return A, B
```

The qsort algorithm

```
def qsort(L):
    n = len(L)
    if n <= 1:
        return
    pivot = max(L[0], L[-1])
    A, B = partition(L, pivot)
    qsort(A)
    qsort(B)
    A.extend(B)
    for i in range(n):
        L[i] = A[i]
```


Run Time Benchmark

List Size	Run Time (seconds)
500	0.0023
600	0.0029
700	0.0034
800	0.004
900	0.0044
1000	0.0051
1100	0.0057
1200	0.0063
1300	0.0069
1400	0.0075
1500	0.008
1600	0.0086
1700	0.0092
1800	0.0097
1900	0.0103
2000	0.0109
2100	0.0115
2200	0.0121
2300	0.0127
2400	0.0132
2500	0.0138
2600	0.0146
2700	0.0152
2800	0.0158
2900	0.0163



Time(n) ≈ 0.0000007050 * n * log n

In Place Sorting

- The quick sort algorithm from last slide, although very fast as compared to the previous algorithms, suffers from one major problem:
- The partition routine I using additional memory (except of L) to generates the two sub-lists (which are returned to the caller)
- The amount of extra space used for an algorithm as a function of its input size is called is space complexity
- Exercise: what is the space complexity of this version of qsort?
- A more efficient approach is to perform the partition “in place” – that is perform partition on the list itself

Tony Hoare Partition Algorithm (1960)

```
def partition(L, start, end):
    pivot = L[start]
    i = start+1
    j = end
    while True:
        while i <= j and L[i] <= pivot:
            i += 1
        while i <= j and pivot <= L[j]:
            j -= 1
        if j < i:
            break
        else:
            L[i], L[j] = L[j], L[i]

    # pivot should move to the middle
    L[start], L[j] = L[j], pivot
    return j
```

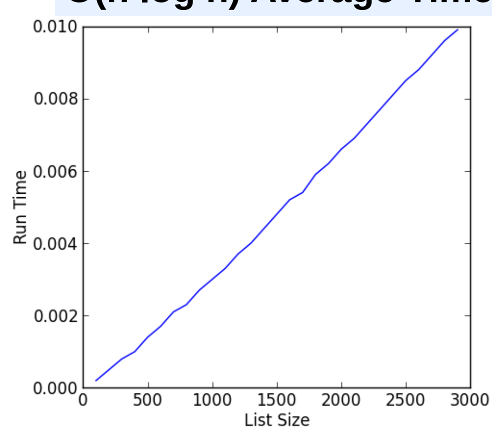
Tony Hoare qsort Algorithm

```
def qsort(L, start=0, end=None):
    if end is None: end = len(L) - 1
    if start < end:
        pivot = partition(L, start, end)
        qsort(L, start, pivot-1)
        qsort(L, pivot+1, end)
```


Quick Sort 2 (Tony Hoare)

List Size	Run Time (seconds)
500	0.0013
600	0.0017
700	0.002
800	0.0023
900	0.0027
1000	0.0029
1100	0.0033
1200	0.0036
1300	0.0041
1400	0.0043
1500	0.0048
1600	0.0052
1700	0.0055
1800	0.0058
1900	0.0063
2000	0.0066
2100	0.007
2200	0.0073
2300	0.0077
2400	0.008
2500	0.0085
2600	0.0089
2700	0.0092
2800	0.0096
2900	0.0099

O(n log n) Average Time

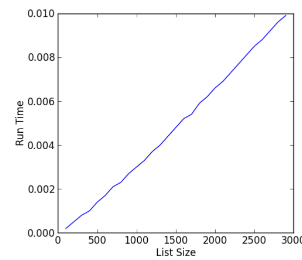


Time(n) ≈ 0.0000004283 * n * log n

O(n**2) worst case !!

Quick Sort 2 (Tony Hoare)

O(n log n) Average Time
O(n2) worst case!**



Time(n) ≈ 0.0000004283 * n * log n

List Size	Run Time (seconds)
10000	**0.0394 seconds**
100000	**0.4930 seconds**
1000000	**5.9171 seconds**
10M	**69.0176 seconds (bubble was 6 months !!!)**
1000M	**8875.7747 seconds, less than 3 hours vs. 5200 years with bubble sort**

So Why Bubble Sort is Important?

- Bubble is a very important example of an algorithm which is very intuitive, very easy to understand, and very easy to prove its correctness, yet this is the worst algorithm with respect to run time complexity
- It proves that an easy and elegant algorithm is not necessarily good!
- It is also a great example to Tim Peters Zen principles:

If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.

RADIX SORT

- Intuitively method based on alphabetizing a large list of names (like in a dictionary)
- The list of names is first sorted according to the first letter: the names are arranged in 26 buckets
- Similarly we can sort numbers according to the most significant digit
- But Radix sort goes by sorting on the least significant digit first. Then on the second pass, the entire numbers are sorted again on the second least-significant digit and so on

Radix Sort

It works great for decimal numbers with equal decimal length

INPUT	1st pass	2nd pass	3rd pass
329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Radix Sort

But if our numbers do not have equal length?
In such case we fill "empty digits" as zeros

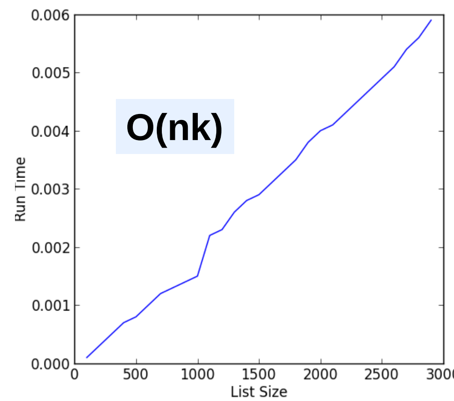
INPUT	VIEW	1st pass	2nd pass	3rd pass	4th pass	5th pass
29	00029	06720	06720	00029	00029	00029
1457	01457	00355	00029	00057	00057	00057
57	00057	00436	00436	00355	00355	00355
31839	31839	01457	31839	00436	00436	00436
436	00436	00057	00355	01457	01457	01457
6720	06720	00029	01457	06720	31839	06720
355	00355	31839	00057	31839	06720	31839

Radix Sort Algorithm (2002)

```
def radix_sort(L):
    RADIX = 10
    deci = 1

    while True:
        buckets = [list() for i in range(RADIX)]
        done = True

        for n in L:
            q = n / deci      # q = quotient
            r = q % RADIX     # r = remainder = last digit
            buckets[r].append(n)
            if q > 0:
                done = False  # i has more digits

        i = 0   # Copy buckets to L (so L is rearranged)
        for r in range(RADIX):
            for n in buckets[r]:
                L[i] = n
                i += 1

        if done: break
        deci *= RADIX      # move to next digit
```

Radix Sort Run Time Benchmark

List Size	Run Time (seconds)
500	0.0008
600	0.001
700	0.0012
800	0.0013
900	0.0014
1000	0.0015
1100	0.0022
1200	0.0023
1300	0.0026
1400	0.0028
1500	0.0029
1600	0.0031
1700	0.0033
1800	0.0035
1900	0.0038
2000	0.004
2100	0.0041
2200	0.0043
2300	0.0045
2400	0.0047
2500	0.0049
2600	0.0051
2700	0.0054
2800	0.0056



Time(n) ≈ 0.0000019 * n
k = average num digits

Radix Sort Run Time

Time(n) ≈ 0.0000019 * n
k = average num digits

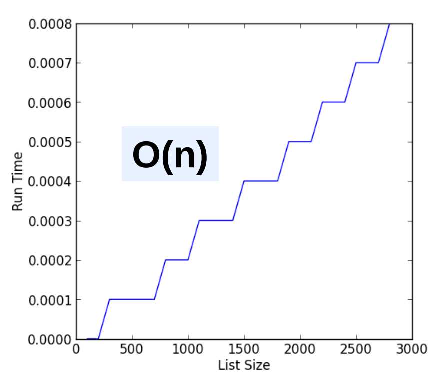
List Size	Run Time (seconds)
10000	0.019 seconds
100000	0.19 seconds
1000000	1.9 seconds
10M	19 seconds (bubble was 6 months !!!)
1000M	1900 seconds – half hour vs. 5200 years with bubble sort

TIM SORT

- Python’s built-in sort algorithm was invented by Tim Peters around 2002
- It is considered to be one of the best sort algorithms in use
- We will not cover it in this preliminary course, but if you’re interested, here are a few interesting links:
  http://en.wikipedia.org/wiki/Timsort
  http://www.youtube.com/watch?v=NVIjHj-lrT4
- Link to a simple test of Tim sort

Tim Sort Run Time Benchmark

List Size	Run Time (seconds)
500	0.0001
600	0.0001
700	0.0001
800	0.0002
900	0.0002
1000	0.0002
1100	0.0003
1200	0.0003
1300	0.0003
1400	0.0003
1500	0.0004
1600	0.0004
1700	0.0004
1800	0.0004
1900	0.0005
2000	0.0005
2100	0.0005
2200	0.0006
2300	0.0006
2400	0.0006
2500	0.0007
2600	0.0007
2700	0.0007
2800	0.0008



Time(n) ≈ 0.0000002857 * n

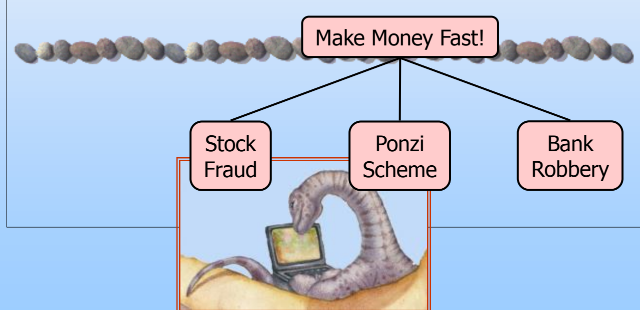
Tim Sort Run Time (average)

Time(n) ≈ 0.0000002857 * n
Worst case is still O(n * log n)

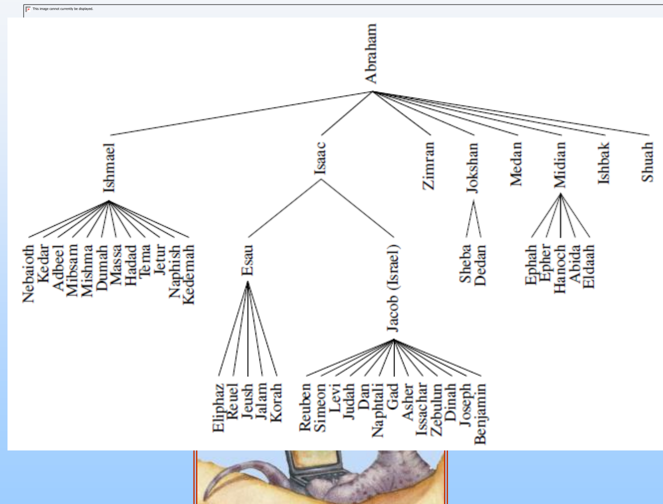
List Size	Run Time (seconds)
10000	0.00286 seconds
100000	0.0286 seconds
1000000	0.286 seconds
10M	2.86 seconds (bubble was 6 months !!!)
1000M	286 seconds – 5 minutes vs. 5200 years with bubble sort

Part 4: Trees

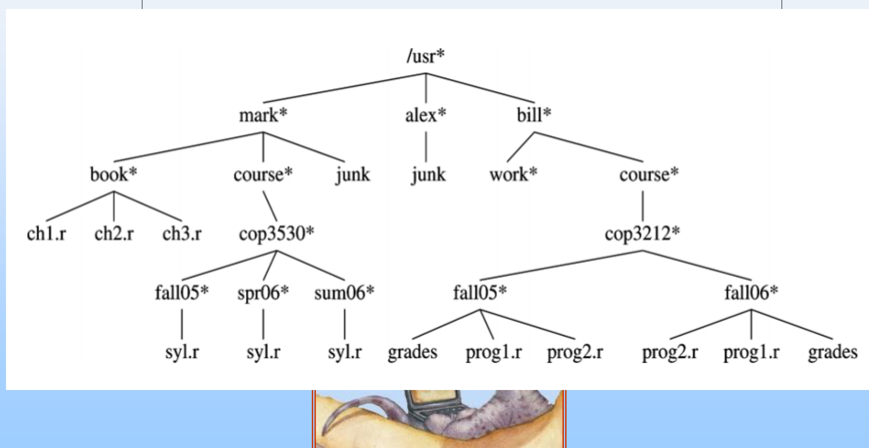
Trees



Example: Family Tree

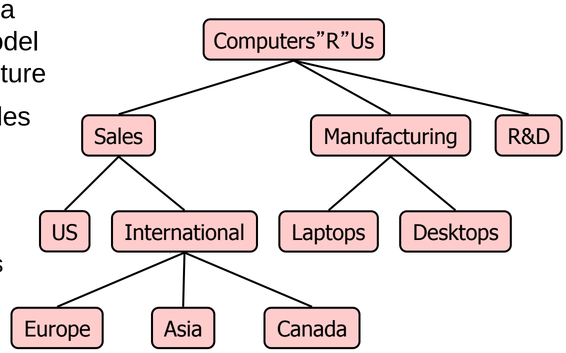


Example: Unix File System

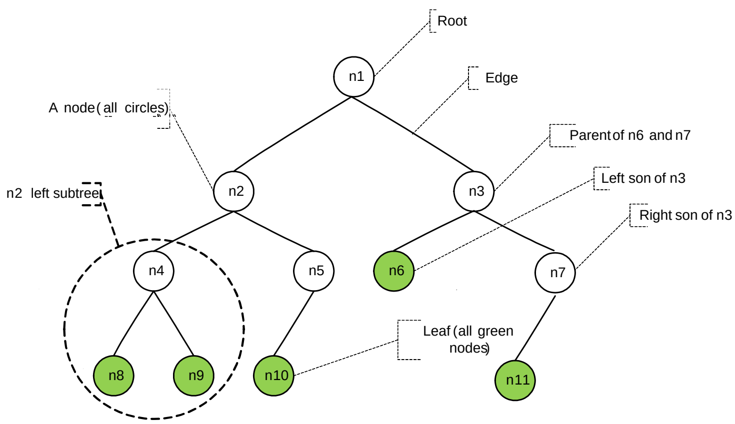


What is a Tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



What is a Tree (Daniel Geva)

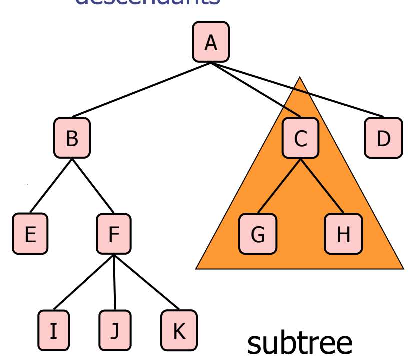


Node height – number of edges on the longest path to a leaf
Tree height – height of the root
Balanced Tree – All non-leaf have two sons

Tree Terminology

- **Root**
  node without parent (A)
- **Internal node**
  node with at least one child (A, B, C, F)
- **Leaf (External node)**
  node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node:
  parent, grandparent, grand-grandparent, etc.
- **Depth of a node:**
  number of ancestors
- **Height of a node:**
  1 + Max height of children
  (leaf height = 0)
- **Height of a tree**
  maximum depth of any node (3)
- **Descendant of a node**
  child, grandchild, grand-grandchild, etc.

- Subtree: tree consisting of a node and its descendants



Tree ADT

- We use positions to abstract nodes, left key is return type:
- Generic methods:
  - Integer len()
  - Boolean is_empty()
  - Iterator positions()
  - Iterator iter()
- Accessor methods:
  - position root()
  - position parent(p)
  - Iterator children(p)
  - Integer num_children(p)

- Query methods:
  - Boolean is_leaf(p)
  - Boolean is_root(p)
- Update method:
  - element replace (p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

Note: A tree **position** is like a list **index**

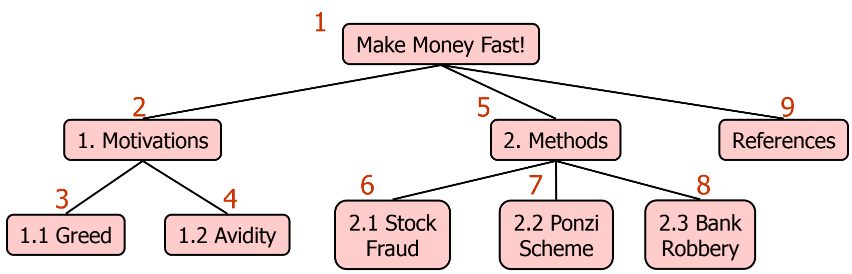
Abstract Tree Class in Python

```
class Tree:
  """Abstract base class representing a tree structure."""

  #------------------------------- nested Position class -------------------------------
  class Position:
    """An abstraction representing the location of a single element."""

    def element(self):
      """Return the element stored at this Position."""
      raise NotImplementedError('must be implemented by subclass')

    def __eq__(self, other):
      """Return True if other Position represents the same location."""
      raise NotImplementedError('must be implemented by subclass')

    def __ne__(self, other):
      """Return True if other does not represent the same location."""
      return not (self == other)            # opposite of __eq__

  # ---------- abstract methods that concrete subclass must support ----------
  def root(self):
    """Return Position representing the tree's root (or None if empty)."""
    raise NotImplementedError('must be implemented by subclass')

  def parent(self, p):
    """Return Position representing p's parent (or None if p is root)."""
    raise NotImplementedError('must be implemented by subclass')

  def num_children(self, p):
    """Return the number of children that Position p has."""
    raise NotImplementedError('must be implemented by subclass')

  def children(self, p):
    """Generate an iteration of Positions representing p's children."""
    raise NotImplementedError('must be implemented by subclass')

  def __len__(self):
    """Return the total number of elements in the tree."""
    raise NotImplementedError('must be implemented by subclass')

  # ---------- concrete methods implemented in this class ----------
  def is_root(self, p):
    """Return True if Position p represents the root of the tree."""
    return self.root( ) == p

  def is_leaf(self, p):
    """Return True if Position p does not have any children."""
    return self.num_children(p) == 0

  def is_empty(self):
    """Return True if the tree is empty."""
    return len(self) == 0
```

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

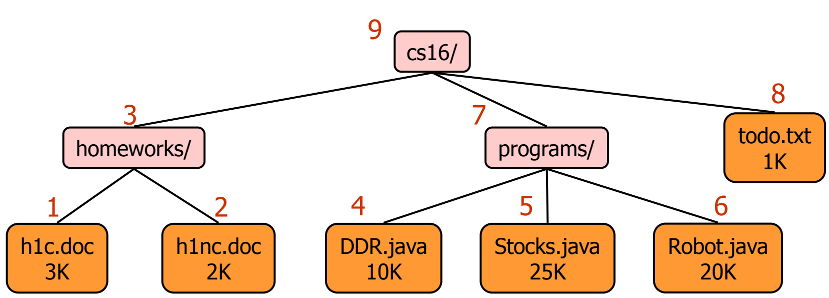
Algorithm *preOrder*(*v*)
    visit(*v*)
    for each child *w* of *v*
        preOrder (*w*)



Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

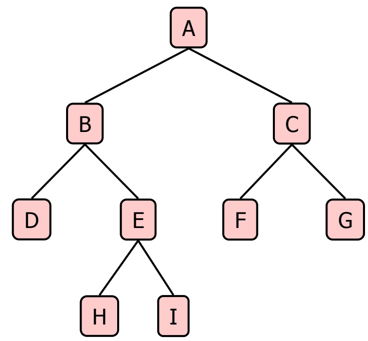
Algorithm *postOrder*(*v*)
    for each child *w* of *v*
        postOrder (*w*)
    visit(*v*)



Binary Trees

- A binary tree is a tree with the following properties:
  - Each internal node has **at most** two children (exactly two for **proper binary trees**)
  - The children of a node are an **ordered pair**
- We call the children of an internal node left child and right child
- Proper Binary Tree: every node is a leaf or must have exactly two children

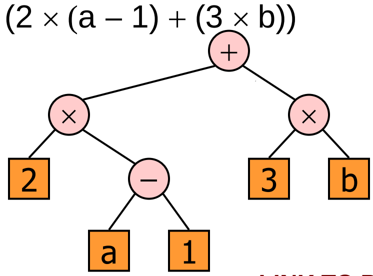
- Applications:
  - arithmetic expressions
  - decision processes
  - searching



LINK TO PYTHON CODE

Arithmetic Expression Tree

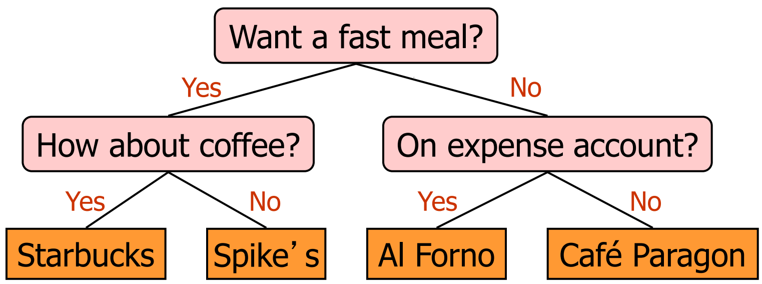
- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



LINK TO PYTHON CODE

Decision Tree

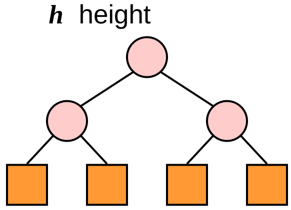
- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision

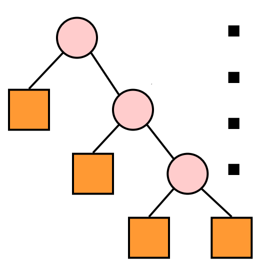


Properties of Proper Binary Trees

- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height

- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \le i$
  - $h \le (n - 1)/2$
  - $e \le 2^h$
  - $h \ge \log_2 e$
  - $h \ge \log_2 (n + 1) - 1$





BinaryTree ADT

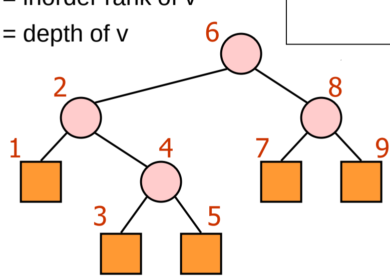
- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - position left(p)
  - position right(p)
  - position sibling(p)
- Update methods may be defined by data structures implementing the BinaryTree ADT

LINK TO PYTHON CODE

Inorder Traversal

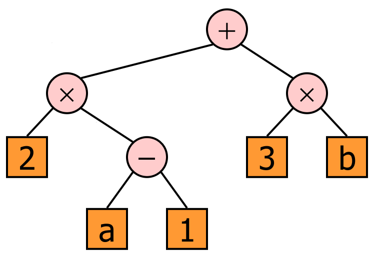
- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

Algorithm *inOrder(v)*
    if *v* **has a left child**
        inOrder (left (v))
    visit(v)
    if *v* **has a right child**
        inOrder (right (v))



Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print “(“ before traversing left subtree
  - print “)“ after traversing right subtree



Algorithm *printExpression(v)*
    if *v* **has a left child**
        print(“(’ ’)
        inOrder (*left*(*v*))
    print(*v.element* ())
    if *v* **has a right child**
        inOrder (*right*(*v*))
        print (“)’ ’)

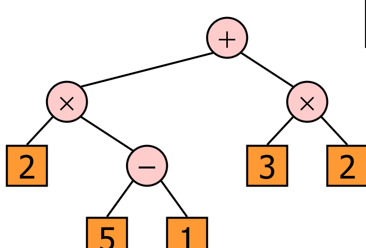
$((2 \times (a - 1)) + (3 \times b))$

LINK TO PYTHON CODE

Evaluate Arithmetic Expressions

- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

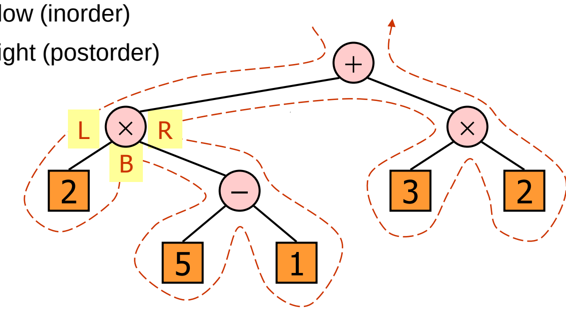
Algorithm *evalExpr*(*v*)
    if *is_leaf* (*v*)
        return *v.element* ()
    else
        $x \leftarrow$ *evalExpr*(*left* (*v*))
        $y \leftarrow$ *evalExpr*(*right* (*v*))
        $\Diamond \leftarrow$ operator stored at *v*
        return $x \Diamond y$



LINK TO PYTHON CODE

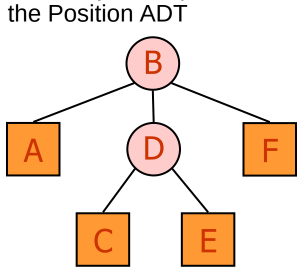
Euler Tour Traversal

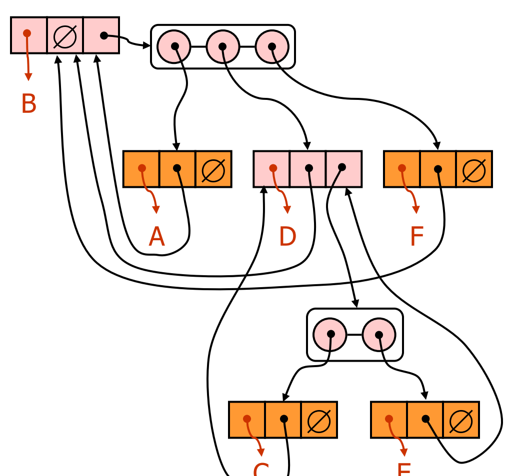
- Generic traversal of a binary tree
- Includes a special cases the preorder, postorder and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (preorder)
  - from below (inorder)
  - on the right (postorder)



Linked Structure for Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes
- Node objects implement the Position ADT



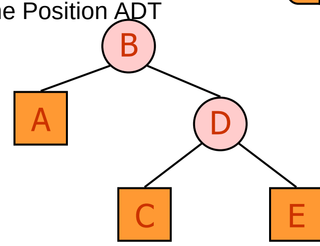


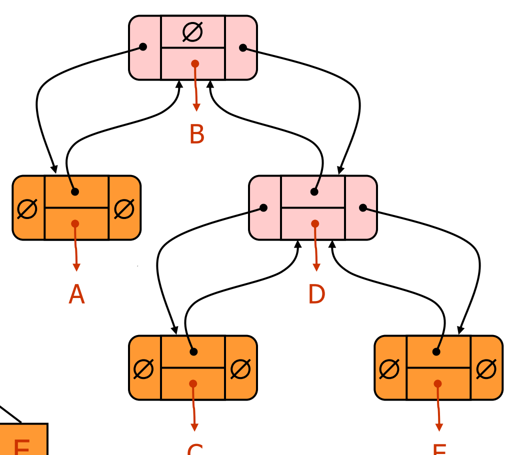
The Node Class

```
class Node:
    "Class for storing a binary tree node"

    def __init__(self, element, parent=None, left=None, right=None):
        self.element = element
        self.parent = parent
        self.left = left
        self.right = right
```


Linked Structure for Binary Trees

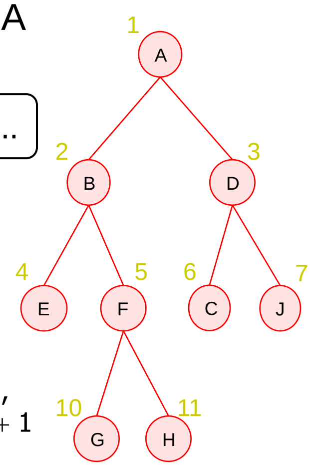
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT





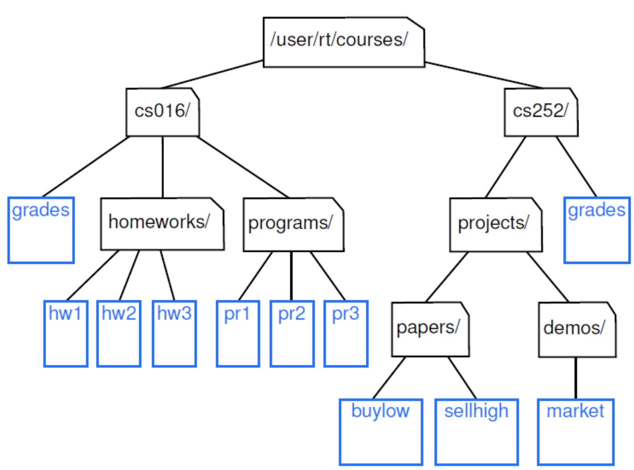
Array-Based Representation of Binary Trees

- Nodes are stored in an array A



- Node v is stored at A[rank(v)]
  - rank(root) = 1
  - if node is the left child of parent(node), rank(node) = 2 · rank(parent(node))
  - if node is the right child of parent(node), rank(node) = 2 · rank(parent(node)) + 1

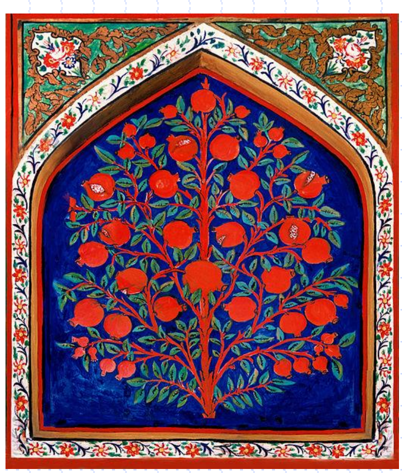
Example: Directory Disk Space



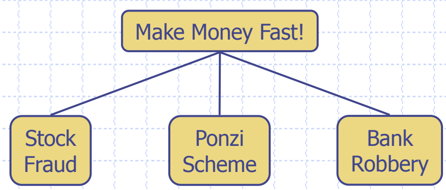
Example: Directory Disk Space

```
import os

def disk_space(dir):
    size = 0
    for file in os.listdir(dir) :
        path = dir + "/" + file
        if os.path.isfile(path):
            size += os.path.getsize(path)
        else:
            size += disk_space(path)
    return size
```

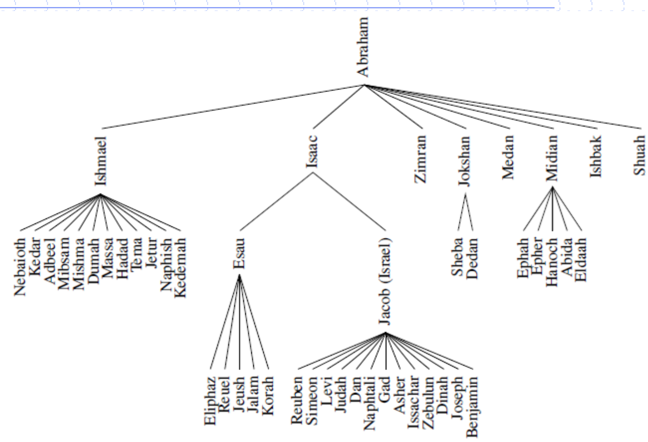

Part 4: Trees



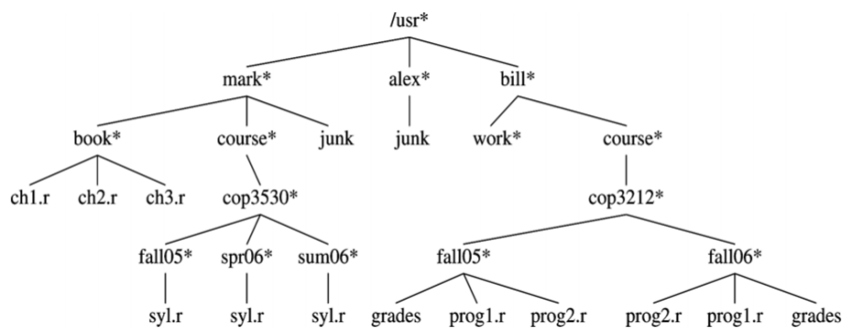
Trees



Example: Family Tree



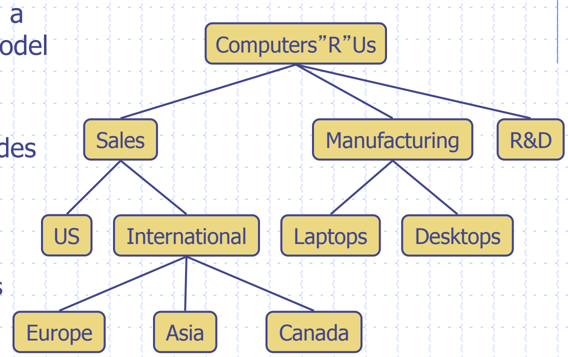
Example: Unix File System



 Trees 179

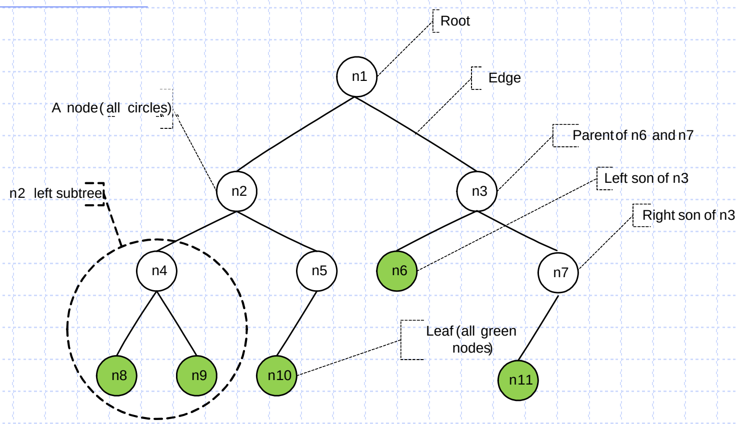
What is a Tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



 Trees 180

What is a Tree (Daniel Geva)

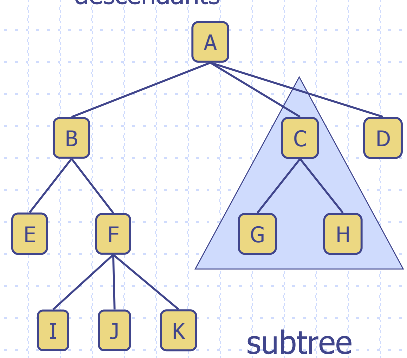


Node height – number of edges on the longest path to a leaf
Tree height – height of the root
Balanced Tree – All non-leaf have two sons

Tree Terminology

- **Root**
  node without parent (A)
- **Internal node**
  node with at least one child (A, B, C, F)
- **Leaf (External node)**
  node without children (E, I, J, K, G, H, D)
- **Ancestors** of a node:
  parent, grandparent, grand-grandparent, etc.
- **Depth of a node:**
  number of ancestors
- **Height of a node:**
  1 + Max height of children
  (leaf height = 0)
- **Height of a tree**
  maximum depth of any node (3)
- **Descendant of a node**
  child, grandchild, grand-grandchild, etc.

- Subtree: tree consisting of a node and its descendants



Tree ADT

- We use positions to abstract nodes, left key is return type:
- Generic methods:
  - Integer len()
  - Boolean is_empty()
  - Iterator positions()
  - Iterator iter()
- Accessor methods:
  - position root()
  - position parent(p)
  - Iterator children(p)
  - Integer num_children(p)
- Query methods:
  - Boolean is_leaf(p)
  - Boolean is_root(p)
- Update method:
  - element replace (p, o)
- Additional update methods may be defined by data structures implementing the Tree ADT

Note: A tree **position** is like a list **index**

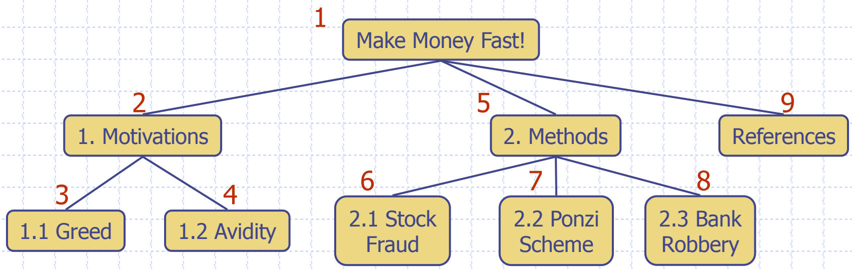
Abstract Tree Class in Python

```
class Tree:
  """Abstract base class representing a tree structure."""

  #------------------------------- nested Position class -------------------------------
  class Position:
    """An abstraction representing the location of a single element."""

    def element(self):
      """Return the element stored at this Position."""
      raise NotImplementedError('must be implemented by subclass')

    def __eq__(self, other):
      """Return True if other Position represents the same location."""
      raise NotImplementedError('must be implemented by subclass')

    def __ne__(self, other):
      """Return True if other does not represent the same location."""
      return not (self == other)            # opposite of __eq__

  # ---------- abstract methods that concrete subclass must support ----------
  def root(self):
    """Return Position representing the tree's root (or None if empty)."""
    raise NotImplementedError('must be implemented by subclass')

  def parent(self, p):
    """Return Position representing p's parent (or None if p is root)."""
    raise NotImplementedError('must be implemented by subclass')

  def num_children(self, p):
    """Return the number of children that Position p has."""
    raise NotImplementedError('must be implemented by subclass')

  def children(self, p):
    """Generate an iteration of Positions representing p's children."""
    raise NotImplementedError('must be implemented by subclass')

  def __len__(self):
    """Return the total number of elements in the tree."""
    raise NotImplementedError('must be implemented by subclass')

  # ---------- concrete methods implemented in this class ----------
  def is_root(self, p):
    """Return True if Position p represents the root of the tree."""
    return self.root( ) == p

  def is_leaf(self, p):
    """Return True if Position p does not have any children."""
    return self.num_children(p) == 0

  def is_empty(self):
    """Return True if the tree is empty."""
    return len(self) == 0
```

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

Algorithm *preOrder*(*v*)
    visit(*v*)
    for each child *w* of *v*
        preOrder (*w*)

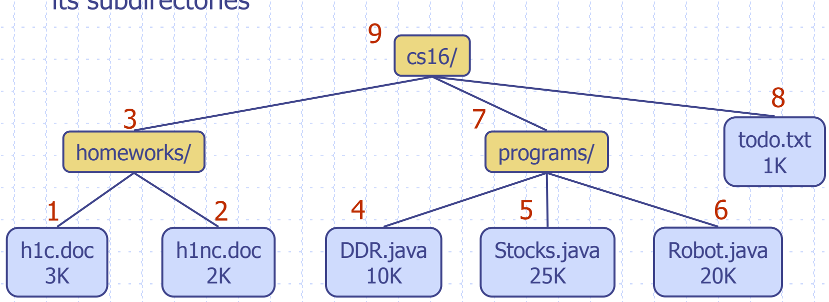


 Trees 185

Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

Algorithm *postOrder*(*v*)
    for each child *w* of *v*
        postOrder (*w*)
    visit(*v*)

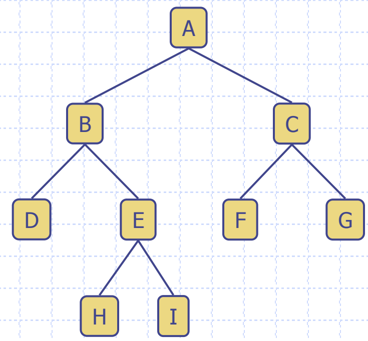


 Trees 186

Binary Trees

- A binary tree is a tree with the following properties:
  - Each internal node has **at most** two children (exactly two for **proper binary trees**)
  - The children of a node are an **ordered pair**
- We call the children of an internal node left child and right child
- Proper Binary Tree: every node is a leaf or must have exactly two children

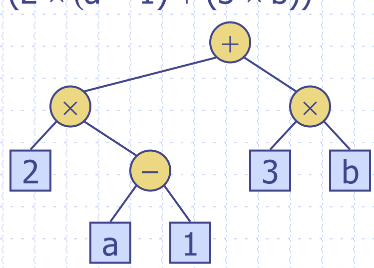
- Applications:
  - arithmetic expressions
  - decision processes
  - searching



LINK TO PYTHON CODE

Arithmetic Expression Tree

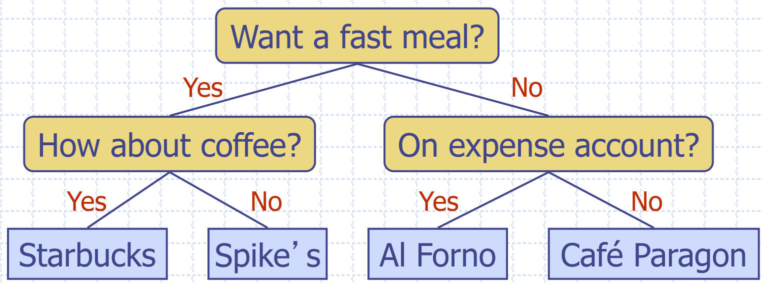
- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



LINK TO PYTHON CODE

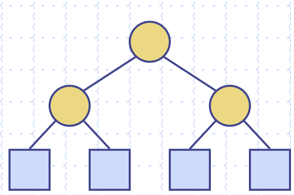
Decision Tree

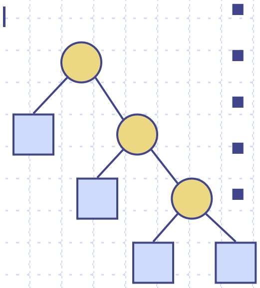
- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



Properties of Proper Binary Trees

- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height





- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$

BinaryTree ADT

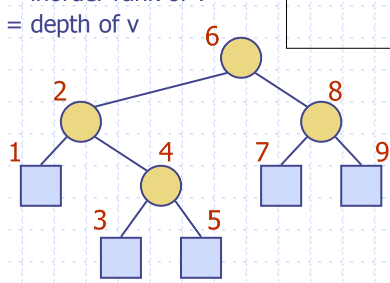
- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - position left(p)
  - position right(p)
  - position sibling(p)
- Update methods may be defined by data structures implementing the BinaryTree ADT

LINK TO PYTHON CODE

Inorder Traversal

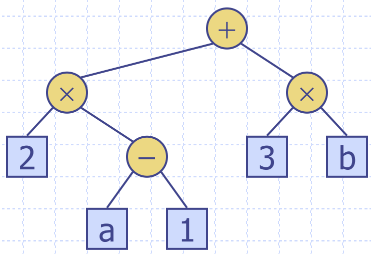
- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

Algorithm *inOrder(v)*
  if *v* **has a left child**
    inOrder (*left* (*v*))
  visit(*v*)
  if *v* **has a right child**
    inOrder (*right* (*v*))



Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print "(" before traversing left subtree
  - print ")" after traversing right subtree



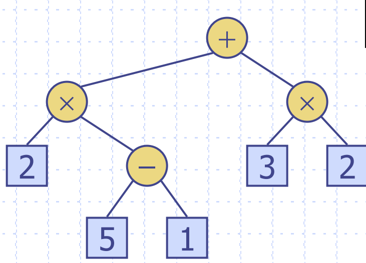
Algorithm *printExpression(v)*
  if *v* **has a left child**
    print("(' ')
    inOrder (*left*(*v*))
  print(*v.element* ())
  if *v* **has a right child**
    inOrder (*right*(*v*))
    print (")' ')

$((2 \times (a - 1)) + (3 \times b))$

LINK TO PYTHON CODE

Evaluate Arithmetic Expressions

- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

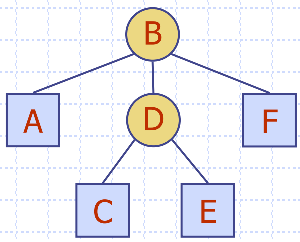


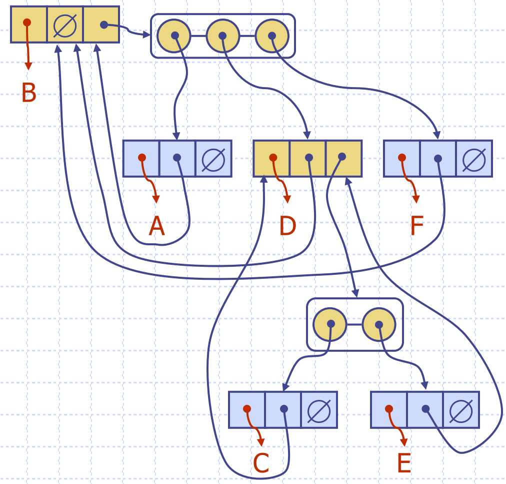
Algorithm *evalExpr*(*v*)
  if *is_leaf* (*v*)
    return *v.element* ()
  else
    $x \leftarrow$ *evalExpr*(*left* (*v*))
    $y \leftarrow$ *evalExpr*(*right* (*v*))
    $\Diamond \leftarrow$ operator stored at *v*
    return $x \Diamond y$

LINK TO PYTHON CODE

Linked Structure for Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes
- Node objects implement the Position ADT



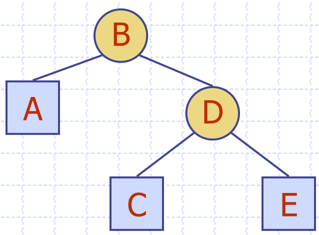


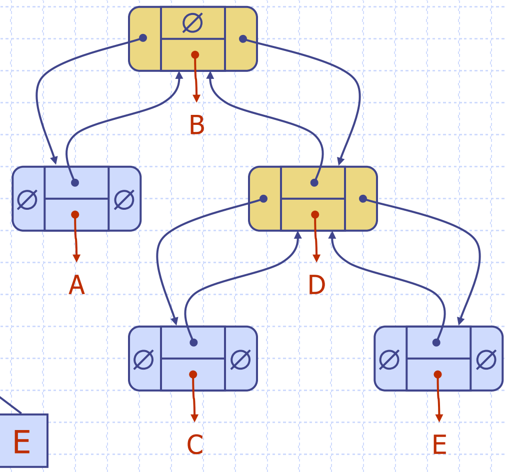
The Node Class

```
class Node:
    "Class for storing a binary tree node"

    def __init__(self, element, parent=None, left=None, right=None):
        self.element = element
        self.parent = parent
        self.left = left
        self.right = right
```

Linked Structure for Binary Trees

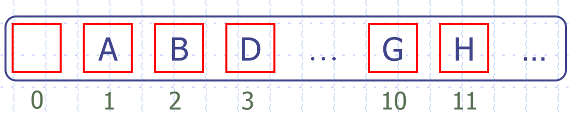
- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node
- Node objects implement the Position ADT

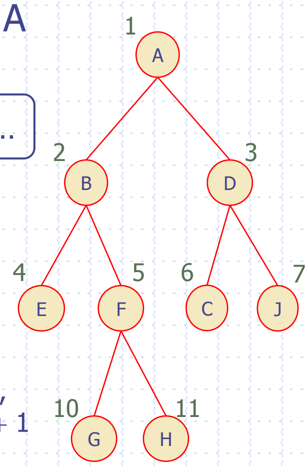




Array-Based Representation of Binary Trees

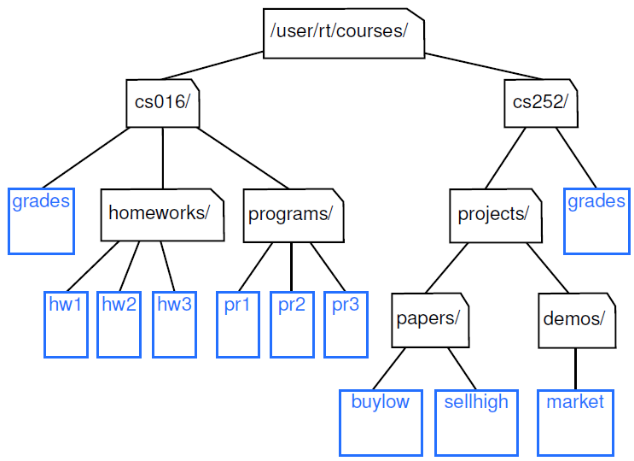
- Nodes are stored in an array A





- Node v is stored at A[rank(v)]
  - rank(root) = 1
  - if node is the left child of parent(node), rank(node) = 2 · rank(parent(node))
  - if node is the right child of parent(node), rank(node) = 2 · rank(parent(node)) + 1

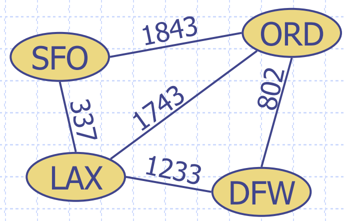
Example: Directory Disk Space



Example: Directory Disk Space

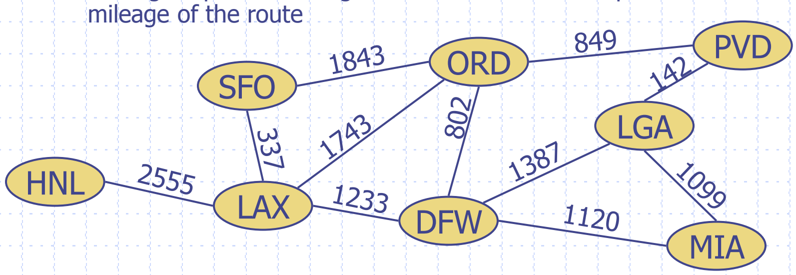
```
import os

def disk_space(dir):
    size = 0
    for file in os.listdir(dir) :
        path = dir + "/" + file
        if os.path.isfile(path):
            size += os.path.getsize(path)
        else:
            size += disk_space(path)
    return size
```

Graphs



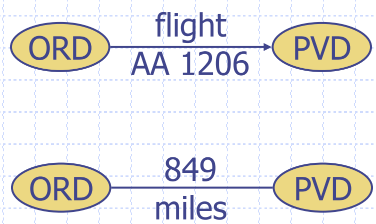
Graphs

- A graph is a pair (V, E), where
  - V is a set of nodes, called vertices
  - E is a collection of pairs of vertices, called edges
  - Vertices and edges are positions and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route



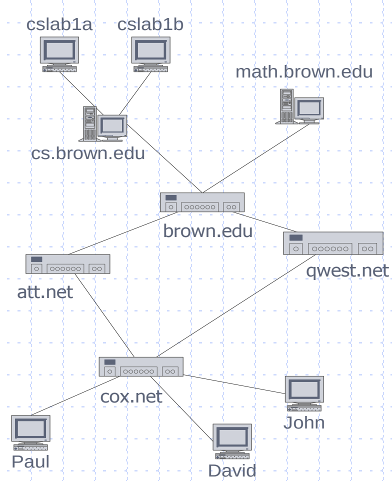
Edge Types

- Directed edge
  - ordered pair of vertices (u,v)
  - first vertex u is the origin
  - second vertex v is the destination
  - e.g., a flight
- Undirected edge
  - unordered pair of vertices (u,v)
  - e.g., a flight route
- Directed graph
  - all the edges are directed
  - e.g., route network
- Undirected graph
  - all the edges are undirected
  - e.g., flight network



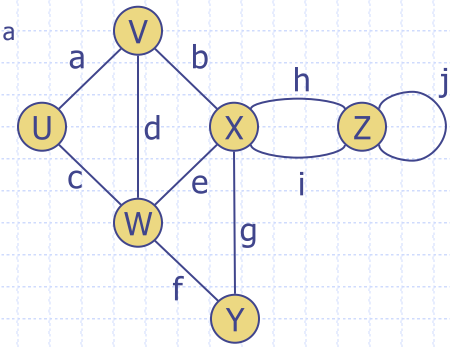
Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit
- Transportation networks
  - Highway network
  - Flight network
- Computer networks
  - Local area network
  - Internet
  - Web
- Databases
  - Entity-relationship diagram



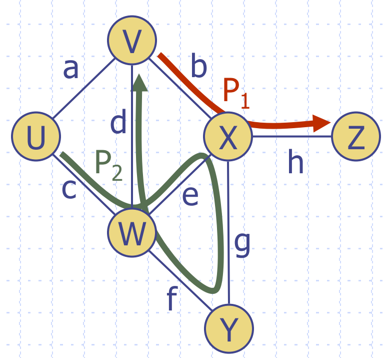
Terminology

- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on V
- Adjacent vertices
  - U and V are adjacent
- Degree of a vertex
  - X has degree 5
- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop



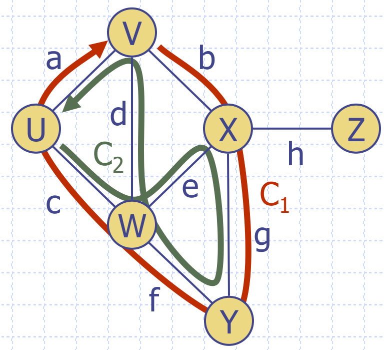
Terminology (cont.)

- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
- Examples
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple



Terminology (cont.)

- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- Simple cycle
  - cycle such that all its vertices and edges are distinct
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,↵) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,↵) is a cycle that is not simple



Properties

Property 1

$\Sigma_v \deg(v) = 2m$

Proof: each edge is counted twice

Property 2

In an undirected graph with no self-loops and no multiple edges

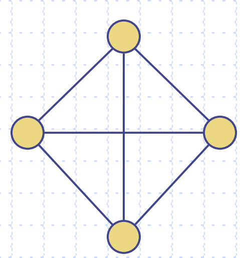
$m \leq n\,(n-1)/2$

Proof: each vertex has degree at most $(n-1)$

What is the bound for a directed graph?

Notation

n	number of vertices
m	number of edges
$\deg(v)$	degree of vertex v



Example

- $n = 4$
- $m = 6$
- $\deg(v) = 3$

Vertices and Edges

- A **graph** is a collection of **vertices** and **edges**.
- We model the abstraction as a combination of three data types: Vertex, Edge, and Graph.
- A **Vertex** is a lightweight object that stores an arbitrary element provided by the user (e.g., an airport code)
  - We assume it supports a method, element(), to retrieve the stored element.
- An **Edge** stores an associated object (e.g., a flight number, travel distance, cost), retrieved with the element( ) method.
- In addition, we assume that an Edge supports the following methods:

endpoints( ): Return a tuple (u,v) such that vertex u is the origin of the edge and vertex v is the destination; for an undirected graph, the orientation is arbitrary.

opposite(v): Assuming vertex v is one endpoint of the edge (either origin or destination), return the other endpoint.

Graph ADT

vertex_count( ): Return the number of vertices of the graph.

vertices( ): Return an iteration of all the vertices of the graph.

edge_count( ): Return the number of edges of the graph.

edges( ): Return an iteration of all the edges of the graph.

get_edge(u,v): Return the edge from vertex u to vertex v, if one exists; otherwise return None. For an undirected graph, there is no difference between get_edge(u,v) and get_edge(v,u).

degree(v, out=True): For an undirected graph, return the number of edges incident to vertex v. For a directed graph, return the number of outgoing (resp. incoming) edges incident to vertex v, as designated by the optional parameter.

incident_edges(v, out=True): Return an iteration of all edges incident to vertex v. In the case of a directed graph, report outgoing edges by default; report incoming edges if the optional parameter is set to False.

insert_vertex(x=None): Create and return a new Vertex storing element x.

insert_edge(u, v, x=None): Create and return a new Edge from vertex u to vertex v, storing element x (None by default).

remove_vertex(v): Remove vertex v and all its incident edges from the graph.

remove_edge(e): Remove edge e from the graph.

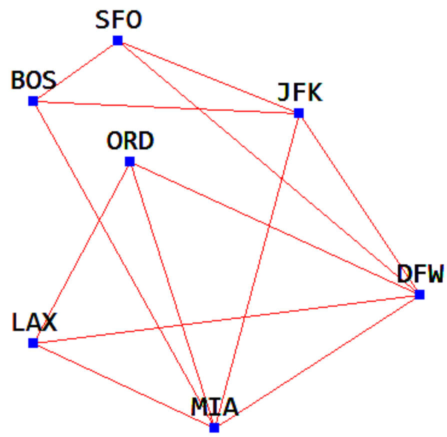
Graph ADT: Basic Usage

```
def basic_graph_example_1():
    g = Graph()
    v1 = g.insert_vertex(1)
    v2 = g.insert_vertex(2)
    v3 = g.insert_vertex(3)
    v4 = g.insert_vertex(4)
    v5 = g.insert_vertex(5)

    e1 = g.insert_edge(v1,v4)
    e2 = g.insert_edge(v3,v1)
    e3 = g.insert_edge(v5,v3)
    e4 = g.insert_edge(v2,v5)

    print "Vertices:"
    for v in g.vertices():
        print v.element()

    print "Edges:"
    for e in g.edges():
        a,b = e.endpoints()
        print a.element(), b.element()
```


Graph ADT: Airport Map Example

```
loc = {
        'BOS':  (80,90),        # BASCO Airport
        'SFO':  (150,40),       # San Francisco International Airport
        'JFK':  (300,100),      # John F. Kennedy Airport, NY
        'MIA':  (230,360),      # Miami Airport, Florida
        'DFW':  (400,250),      # Dallas/Fort Worth International Airport
        'ORD':  (160,140),      # Chicago O'Hare International Airport
        'LAX':  (80,290),       # Los Angeles International Airport
      }

E = (   # Airport connections
        ('BOS','SFO'), ('BOS','JFK'), ('BOS','MIA'), ('JFK','BOS'),
        ('JFK','DFW'), ('JFK','MIA'), ('JFK','SFO'), ('ORD','DFW'),
        ('ORD','MIA'), ('LAX','ORD'), ('DFW','SFO'), ('DFW','ORD'),
        ('DFW','LAX'), ('MIA','DFW'), ('MIA','LAX'),
    )
```

Graph ADT: Graphical View



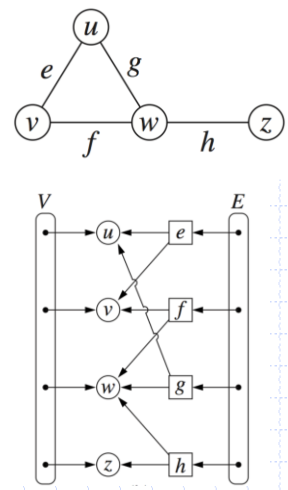
Graph ADT: Code

```
def draw_airport_map():
    g = Graph(True)  # directed graph !
    vert = dict()    # dictionary from label to vertex object
    for a in loc:
        vert[a] = g.insert_vertex(a)

    for a,b in E:
        g.insert_edge(vert[a], vert[b])

    for v in g.vertices():
        airport = v.element()
        p = Point(*loc[airport])
        p.draw()
        p.text(airport)

    for e in g.edges():
        a, b = e.endpoints()
        x1, y1 = loc[a.element()]
        x2, y2 = loc[b.element()]
        l = Line.from_coords(x1, y1, x2, y2)
        l.draw(fill="red", width=1, arrow="last", arrowshape=[10,14,4])
```

```
loc = {
        'BOS':  (80,90),       # BASCO Airport
        'SFO':  (150,40),      # San Francisco International Airport
        'JFK':  (300,100),     # John F. Kennedy Airport, NY
        'MIA':  (230,360),     # Miami Airport, Florida
        'DFW':  (400,250),     # Dallas/Fort Worth International Airport
        'ORD':  (160,140),     # Chicago O'Hare International Airport
        'LAX':  (80,290),      # Los Angeles International Airport
      }

E = (   # Airport connections
        ('BOS','SFO'), ('BOS','JFK'), ('BOS','MIA'), ('JFK','BOS'),
        ('JFK','DFW'), ('JFK','MIA'), ('JFK','SFO'), ('ORD','DFW'),
        ('ORD','MIA'), ('LAX','ORD'), ('DFW','SFO'), ('DFW','ORD'),
        ('DFW','LAX'), ('MIA','DFW'), ('MIA','LAX'),
    )
```

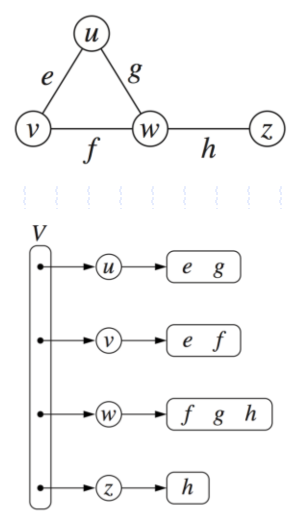
Edge List Structure

- Vertex object
  - element
  - reference to position in vertex sequence
- Edge object
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence
- Vertex sequence
  - sequence of vertex objects
- Edge sequence
  - sequence of edge objects



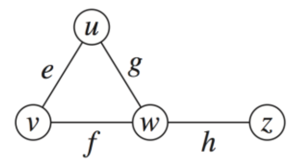
Adjacency List Structure

- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices



Adjacency Matrix Structure

- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non nonadjacent vertices
- The "old fashioned" version just has 0 for no edge and 1 for edge



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

Performance

▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	deg(v)	n
areAdjacent (v, w)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	deg(v)	n^2
removeEdge(e)	1	1	1

Python Graph Implementation

- We use a variant of the ***adjacency map*** representation.
- For each vertex *v*, we use a Python dictionary to represent the secondary incidence map *I*(*v*).
- The list *V* is replaced by a top-level dictionary *D* that maps each vertex *v* to its incidence map *I*(*v*).
  - Note that we can iterate through all vertices by generating the set of keys for dictionary *D*.
- A vertex does not need to explicitly maintain a reference to its position in *D*, because it can be determined in *O*(1) expected time.
- Running time bounds for the adjacency-list graph ADT operations, given above, become ***expected*** bounds.

Vertex Class

```
#------------------------- nested Vertex class -------------------------
class Vertex:
  """Lightweight vertex structure for a graph."""
  __slots__ = '_element'

  def __init__(self, x):
    """Do not call constructor directly. Use Graph's insert_vertex(x)."""
    self._element = x

  def element(self):
    """Return element associated with this vertex."""
    return self._element

  def __hash__(self):          # will allow vertex to be a map/set key
    return hash(id(self))
```


Edge Class

```python
#------------------------- nested Edge class -------------------------
class Edge:
  """Lightweight edge structure for a graph."""
  __slots__ = '_origin', '_destination', '_element'

  def __init__(self, u, v, x):
    """Do not call constructor directly. Use Graph's insert_edge(u,v,x)."""
    self._origin = u
    self._destination = v
    self._element = x

  def endpoints(self):
    """Return (u,v) tuple for vertices u and v."""
    return (self._origin, self._destination)

  def opposite(self, v):
    """Return the vertex that is opposite v on this edge."""
    return self._destination if v is self._origin else self._origin

  def element(self):
    """Return element associated with this edge."""
    return self._element

  def __hash__(self):          # will allow edge to be a map/set key
    return hash( (self._origin, self._destination) )
```

 Graphs 221

Graph, Part 1

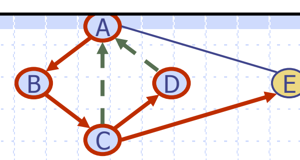
```python
class Graph:
  """Representation of a simple graph using an adjacency map."""

  def __init__(self, directed=False):
    """Create an empty graph (undirected, by default).

    Graph is directed if optional paramter is set to True.
    """
    self._outgoing = { }
    # only create second map for directed graph; use alias for undirected
    self._incoming = { } if directed else self._outgoing

  def is_directed(self):
    """Return True if this is a directed graph; False if undirected.

    Property is based on the original declaration of the graph, not its contents.
    """
    return self._incoming is not self._outgoing # directed if maps are distinct

  def vertex_count(self):
    """Return the number of vertices in the graph."""
    return len(self._outgoing)

  def vertices(self):
    """Return an iteration of all vertices of the graph."""
    return self._outgoing.keys( )

  def edge_count(self):
    """Return the number of edges in the graph."""
    total = sum(len(self._outgoing[v]) for v in self._outgoing)
    # for undirected graphs, make sure not to double-count edges
    return total if self.is_directed( ) else total // 2

  def edges(self):
    """Return a set of all edges of the graph."""
    result = set( )        # avoid double-reporting edges of undirected graph
    for secondary_map in self._outgoing.values( ):
      result.update(secondary_map.values( ))     # add edges to resulting set
    return result
```

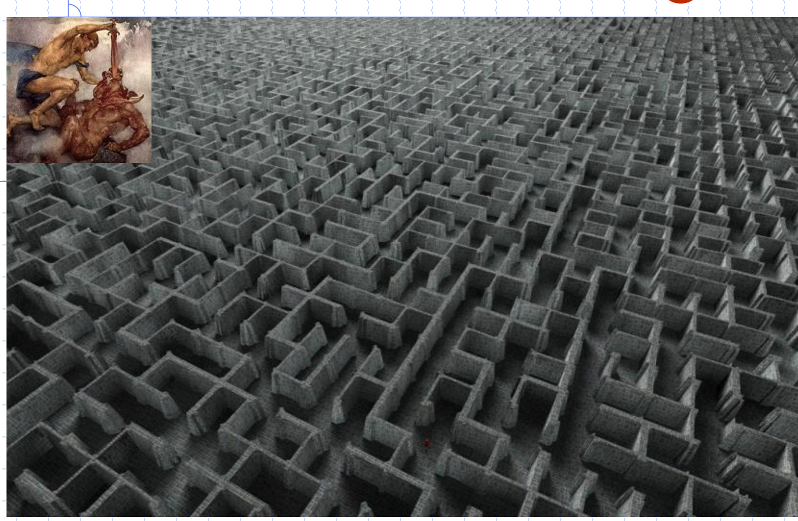
 Graphs 222

Graph, end

```
def get_edge(self, u, v):
  """Return the edge from u to v, or None if not adjacent."""
  return self._outgoing[u].get(v)          # returns None if v not adjacent

def degree(self, v, outgoing=True):
  """Return number of (outgoing) edges incident to vertex v in the graph.

  If graph is directed, optional parameter used to count incoming edges.
  """
  adj = self._outgoing if outgoing else self._incoming
  return len(adj[v])

def incident_edges(self, v, outgoing=True):
  """Return all (outgoing) edges incident to vertex v in the graph.

  If graph is directed, optional parameter used to request incoming edges.
  """
  adj = self._outgoing if outgoing else self._incoming
  for edge in adj[v].values( ):
    yield edge

def insert_vertex(self, x=None):
  """Insert and return a new Vertex with element x."""
  v = self.Vertex(x)
  self._outgoing[v] = { }
  if self.is_directed( ):
    self._incoming[v] = { }          # need distinct map for incoming edges
  return v

def insert_edge(self, u, v, x=None):
  """Insert and return a new Edge from u to v with auxiliary element x."""
  e = self.Edge(u, v, x)
  self._outgoing[u][v] = e
  self._incoming[v][u] = e
```

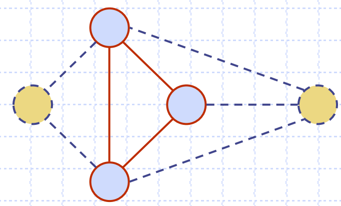

Depth-First Search

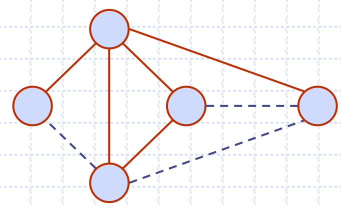




Subgraphs

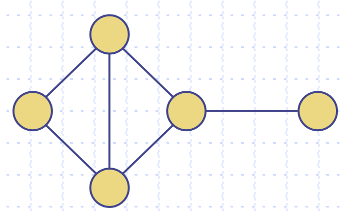
- A subgraph S of a graph G is a graph such that
  - The vertices of S are a subset of the vertices of G
  - The edges of S are a subset of the edges of G
- A spanning subgraph of G is a subgraph that contains all the vertices of G

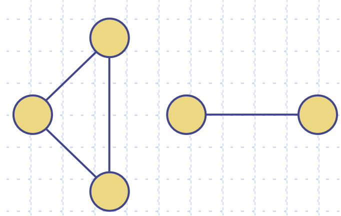

Subgraph


Spanning subgraph

Connectivity

- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G

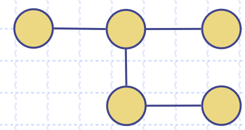

Connected graph

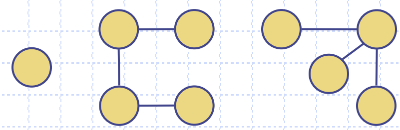

Non connected graph with two connected components

Trees and Forests

- A (free) tree is an undirected graph T such that
  - T is connected
  - T has no cycles

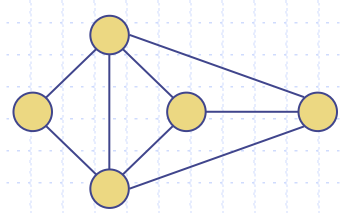
  This definition of tree is different from the one of a rooted tree
- A forest is an undirected graph without cycles
- The connected components of a forest are trees

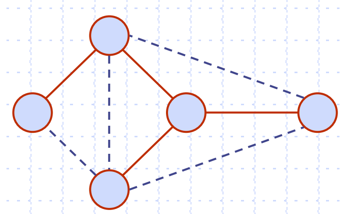




Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest





Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- DFS on a graph with n vertices and m edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Find a cycle in the graph
- Depth-first search is to graphs what Euler tour is to binary trees

DFS Algorithm

- The algorithm uses a mechanism for setting and getting "labels" of vertices and edges

```
Algorithm DFS(G)
    Input graph G
    Output labeling of the edges of G
        as discovery edges and
        back edges
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      DFS(G, v)
```

```
Algorithm DFS(G, v)
  Input graph G and a start vertex v of G
  Output labeling of the edges of G
    in the connected component of v
    as discovery edges and back edges
  setLabel(v, VISITED)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        DFS(G, w)
      else
        setLabel(e, BACK)
```

Python Implementation

```
def DFS(g, u, discovered):
  """Perform DFS of the undiscovered portion of Graph g starting at Vertex u.

  discovered is a dictionary mapping each vertex to the edge that was used to
  discover it during the DFS. (u should be "discovered" prior to the call.)
  Newly discovered vertices will be added to the dictionary as a result.
  """
  for e in g.incident_edges(u):          # for every outgoing edge from u
    v = e.opposite(u)
    if v not in discovered:              # v is an unvisited vertex
      discovered[v] = e                  # e is the tree edge that discovered v
      DFS(g, v, discovered)              # recursively explore from v
```


Example

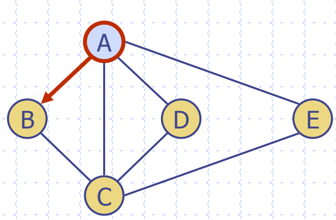
A unexplored vertex

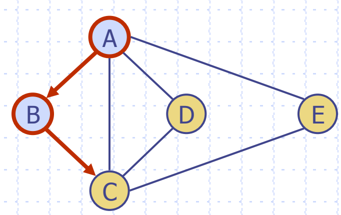
A visited vertex

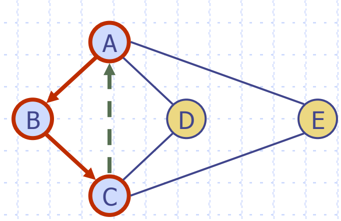
unexplored edge

discovery edge

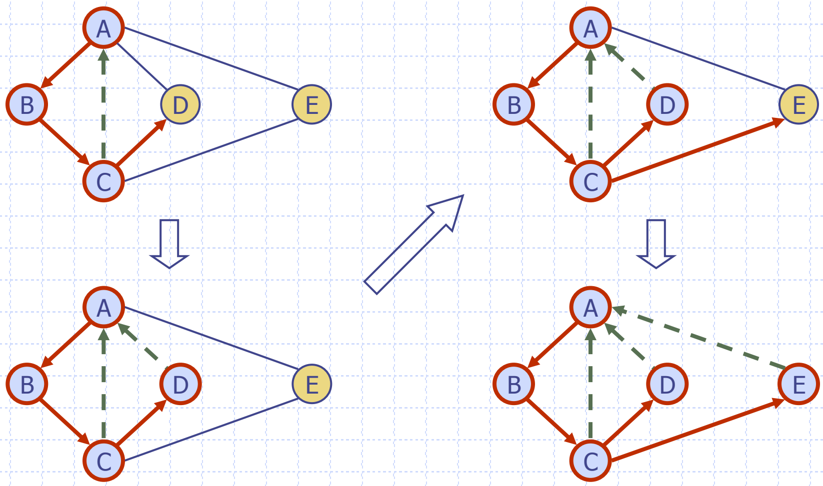
back edge



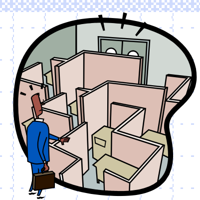




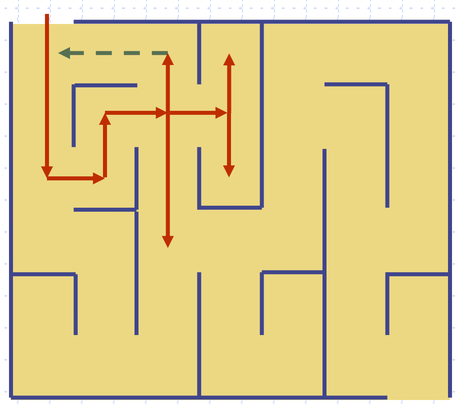
Example (cont.)



DFS and Maze Traversal



- The DFS algorithm is similar to a classic strategy for exploring a maze
  - We mark each intersection, corner and dead end (vertex) visited
  - We mark each corridor (edge) traversed
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack)



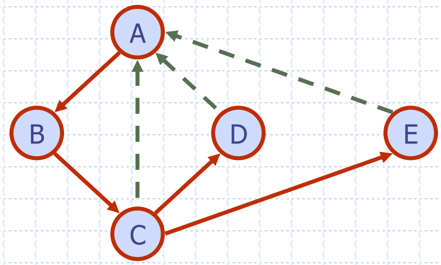
Properties of DFS

Property 1

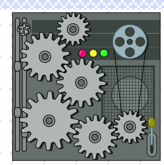
$DFS(G, v)$ visits all the vertices and edges in the connected component of v

Property 2

The discovery edges labeled by $DFS(G, v)$ form a spanning tree of the connected component of v

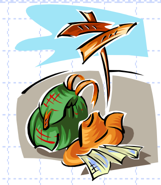


Analysis of DFS



- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$

Path Finding



- We can specialize the DFS algorithm to find a path between two given vertices *u* and *z* using the template method pattern
- We call *DFS*(*G*, *u*) with *u* as the start vertex
- We use a stack *S* to keep track of the path between the start vertex and the current vertex
- As soon as destination vertex *z* is encountered, we return the path as the contents of the stack

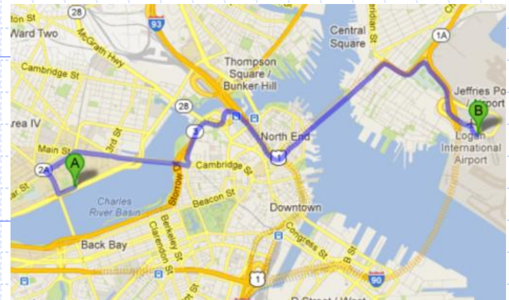
```
Algorithm pathDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  if v = z
    return S.elements()
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        S.push(e)
        pathDFS(G, w, z)
        S.pop(e)
      else
        setLabel(e, BACK)
  S.pop(v)
```

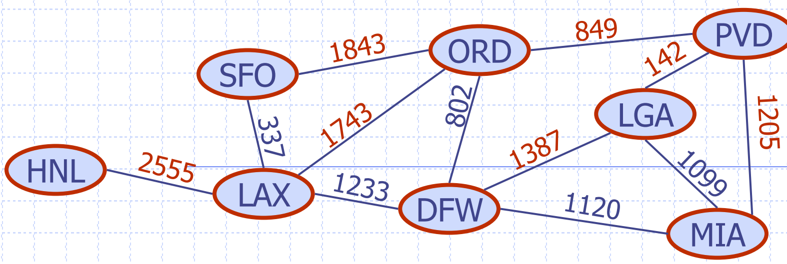

Cycle Finding

- We can specialize the DFS algorithm to find a simple cycle using the template method pattern
- We use a stack *S* to keep track of the path between the start vertex and the current vertex
- As soon as a back edge (*v*, *w*) is encountered, we return the cycle as the portion of the stack from the top to vertex *w*

```
Algorithm cycleDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      S.push(e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        pathDFS(G, w, z)
        S.pop(e)
      else
        T ← new empty stack
        repeat
          o ← S.pop()
          T.push(o)
        until o = w
        return T.elements()
  S.pop(v)
```

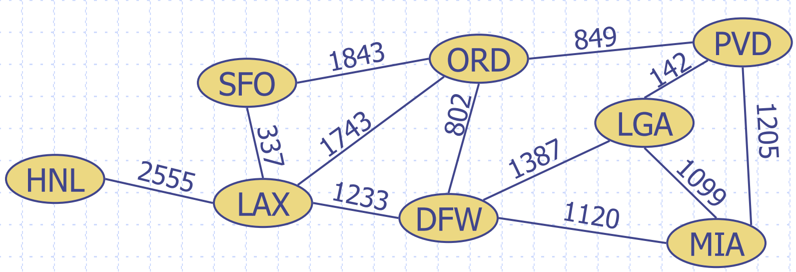

Shortest Paths





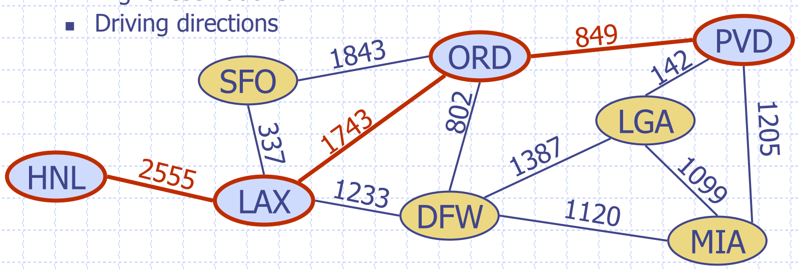
Weighted Graphs

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Edge weights may represent, distances, costs, etc.
- Example:
  - In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports
  - What is the shortest path from HNL to PVD ?



Shortest Paths

- Given a weighted graph and two vertices u and v, we want to find a path of minimum total weight between u and v.
  - Length of a path is the sum of the weights of its edges.
- Example:
  - Shortest path between Providence and Honolulu
- Applications
  - Internet packet routing
  - Flight reservations
  - Driving directions

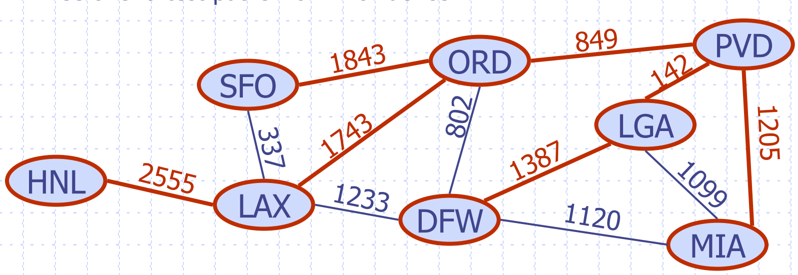


Shortest Path Properties

Property 1:
A subpath of a shortest path is itself a shortest path

Property 2:
There is a tree of shortest paths from a start vertex to all the other vertices

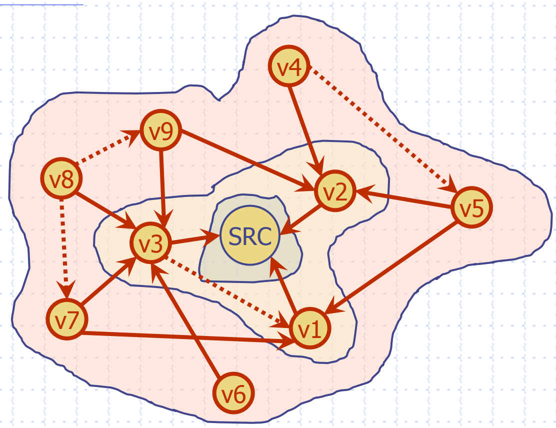
Example:
Tree of shortest paths from Providence



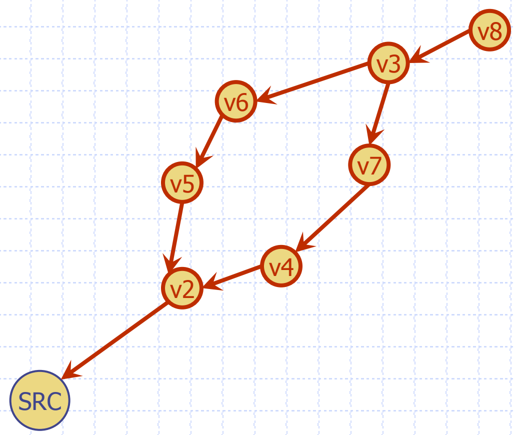
Dijkstra's Algorithm

- The distance of a vertex v from a vertex s is the length of a shortest path between s and v
- Dijkstra's algorithm computes the distances of all the vertices from a given start vertex s
- Assumptions:
  - the graph is connected
  - the edges are directed
  - the edge weights are nonnegative
- We grow a "cloud" of vertices, beginning with s and eventually covering all the vertices
- We store with each vertex v a label $d(v)$ representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices
- At each step
  - We add to the cloud the vertex u outside the cloud with the smallest distance label, $d(u)$
  - We update the labels of the vertices adjacent to u

Cloud Progresssion



Correctness Proof

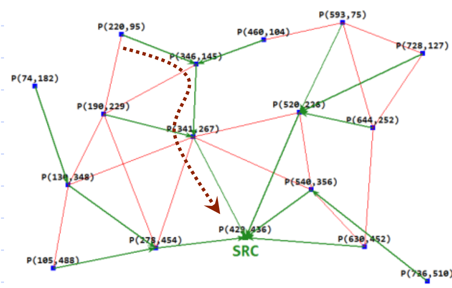


Dijkstra's Algorithm

```
def dijkstra(g, src):
    cloud = {src: 0}       # cloud of visited vertices/edges and their distance from src
    gps = {}               # gps dictionary maps a vertex to edge toward source src
    distance = {}          # distance dictionary: distance[u] = min distance from u to src
    vertices = set(g.vertices())
    vertices.remove(src)   # src is the single element currently in cloud
    distance[src] = 0      # distance from src to itself is 0
    for u in vertices:     # distance of any other vertex to source is infinity
        distance[u] = float('Infinity')

    while True:
        # Construct the next ring
        ring = []
        for v in cloud:
            for edge in g.incident_edges(v, False):   # incoming edges to v
                u = edge.opposite(v)
                du = distance[v] + edge.element()
                if du < distance[u]:
                    distance[u] = du
                    gps[u] = edge
                if u not in cloud:
                    ring.append(u)
   if not ring:
            break

        for u in ring:
            cloud[u] = distance[u]

    return cloud, gps
```

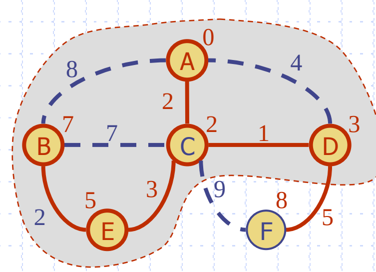
Shortest Path

```
# Given a graph g, a cloud tree as above
# we can easily compute a path from source to destination

def shortest_path(g, tree, source, destination):
    path = []
    v = destination
    while True:
        if not v in tree:
            break
        e = tree[v]
        path.append((v,e))
        v = e.opposite(v)

    return path
```



Why Dijkstra's Algorithm Works

- Dijkstra's algorithm is based on the greedy method. It adds vertices by increasing distance.
  - Suppose it didn't find all shortest distances. Let F be the first wrong vertex the algorithm processed.
  - When the previous node, D, on the true shortest path was considered, its distance was correct
  - But the edge (D,F) was relaxed at that time!
  - Thus, so long as d(F)≥d(D), F's distance cannot be wrong. That is, there is no wrong vertex



 Shortest Paths