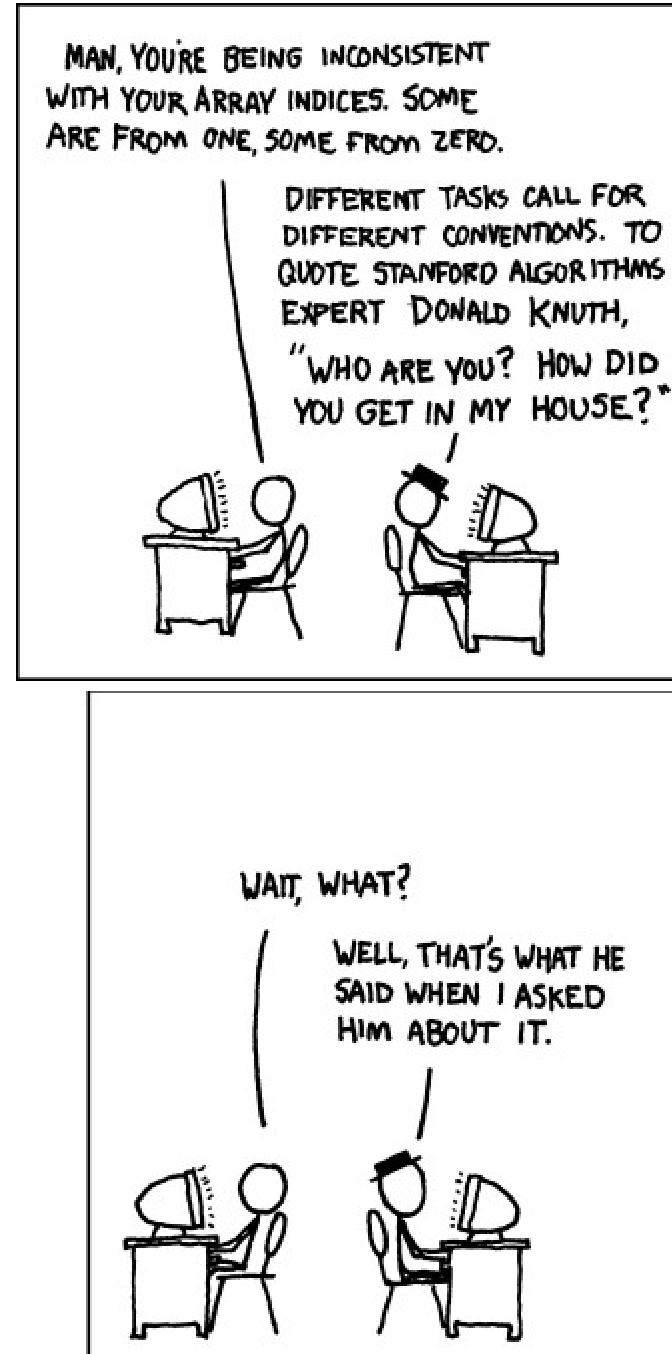
Programming Concepts: Containers

Paulo Penteado

http://www.ppenteado.net/pc/



(http://xkcd.com/163)

Containers

A single value in a variable is not enough.

Containers – variables that hold several values (the elements)

There are many ways to organize the elements: arrays are just one of them
- Each way is implementing some *data structure**.

There is no "best container":

- Each is best suited to different problems

The 3 main properties of containers:

- **Homogeneous X heterogeneous**: whether all elements are the same type.

- **Static X dynamic**: whether the number of elements is fixed.

- **Sequentiality**:
  - Sequential containers: elements stored by order, and are accessed by indices.
  - Non-sequential containers: elements stored by name or through relationships.

**A data structure* is a way of organizing data; a *structure* is just one of them.

Containers

The most common types (names vary among languages; some have several implementations for the same type)*:

- **Array / vector / matrix (1D or MD)**: C, C++, Fortran, IDL, Java, Python+Numpy, R
- **List**: C++, Python, IDL (≥8), Java, R, Perl**
- **Map / hash / hashtable / associative array / dictionary**: C++, Python, IDL (≥8), Java, R***, Perl
- **Set**: C++, Python, Java, R
- **Tree / heap**: C++, Python, Java
- **Stack**: C++, Python, Java
- **Queue**: C++, Python, Java

*Listed only when the structure is part of a language's standard library.

**A Perl array is more like a list than an array.

***Which in R are also called *named lists.*

Arrays - definition

The simplest container.

A sequential set o elements, organized **regularly**, in 1D or more (MD).

Not natively present in some recent languages (Perl, Python without Numpy).

Sometimes called **array** only when more than 1D, being called **vector** in the 1D case.

2D sometimes called tables or **matrices**

- In some languages (ex: R, Python+Numpy), ***matrix*** is different from a generic array.

Arrays - characteristics

Homogeneous (all elements must be the same **type**)

Static (cannot change the number of elements)

- “Dynamic arrays” are actually creating new arrays, and throwing away the old ones on resize (which is inefficient).

Sequential (elements stored by an order)

Organized in 1D or more (MD).

Element access through their indices (sequential integer numbers).

Usually, **the most efficient container for random and sequential access.**

Provide the means to do vectorization (do operations on the whole array, or parts of the array, with a single statement).

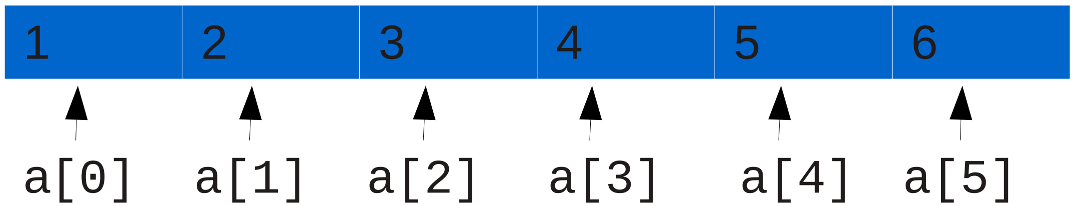
- 1D arrays are common.
- MD arrays are often awkward (2D may not be so bad): **IDL and Python+Numpy have high level MD operations**.

Internally all elements are **stored as a 1D array, even when there are more dimensions** (memory and files are 1D).

- **When over 1D, they are always regular** (each dimension has a constant number of elements).

Arrays

1D



Ex. (Python):

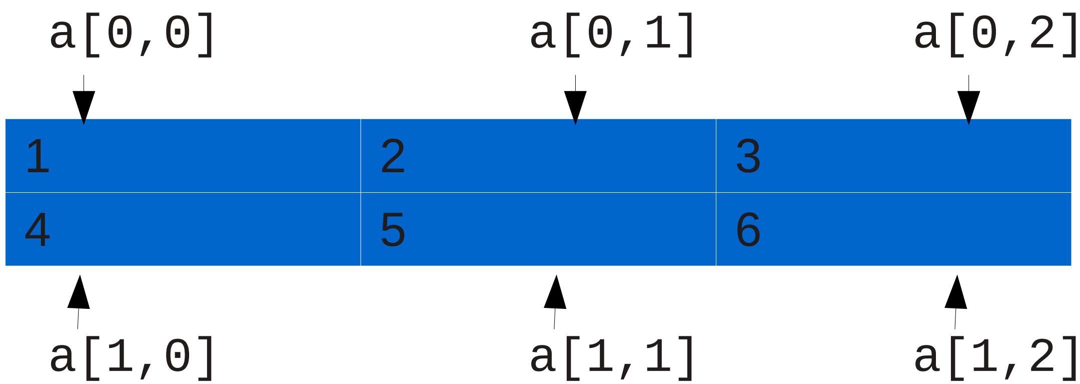
```
In [5]: import numpy as np

In [6]: a=np.arange(6)+1

In [7]: a
Out[7]: array([1, 2, 3, 4, 5, 6])

In [10]: a.size
Out[10]: 6

In [11]: a.shape
Out[11]: (6,)
```

Generates an array of integers with 6 elements, valued 1 to 6.

Most often, indexes start at 0. In some languages, the start index can be chosen.

Arrays

2D

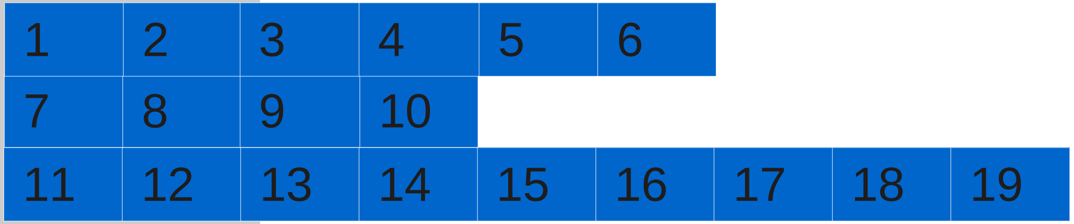


Ex. (Python):

```
In [38]: import numpy as np

In [39]: a=(np.arange(6)+1).reshape((2,3))

In [40]: a.size
Out[40]: 6

In [41]: a.shape
Out[41]: (2, 3)

In [42]: a
Out[42]:
array([[1, 2, 3],
       [4, 5, 6]])
```

Generates an array with 6 elements, in 3 columns by 2 rows, valued 1 to 6.

Must be regular: cannot be like

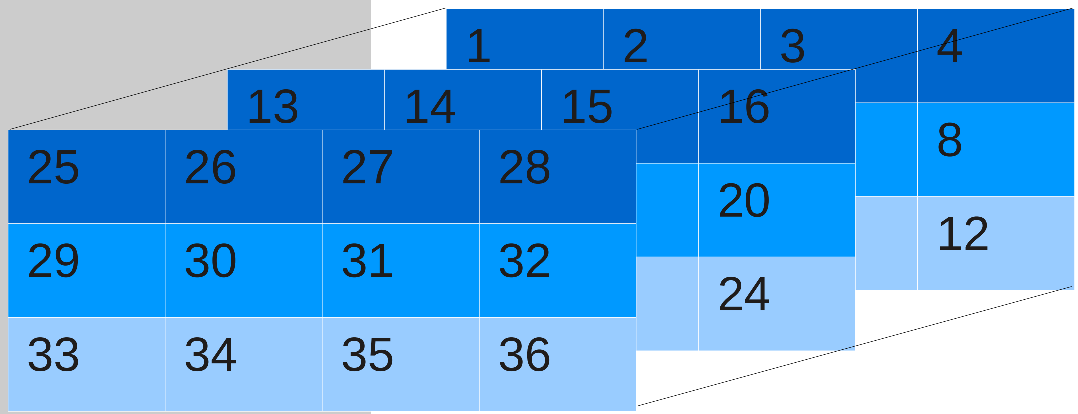


Arrays

3D is usually thought, graphically, as pile of “pages”, each page being a 2D table. Or as a brick. Ex. (Python):

```
In [67]: import numpy as np

In [68]: a=np.arange(36).reshape((3,3,4))

In [69]: a.size
Out[69]: 36

In [71]: a.shape
Out[71]: (3, 3, 4)

In [72]: a
Out[72]:
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]],

       [[24, 25, 26, 27],
        [28, 29, 30, 31],
        [32, 33, 34, 35]]])
```

Generates an array of integers, with 36 elements, over 4 columns, 3 rows, 3 “pages”, valued 0 to 35.



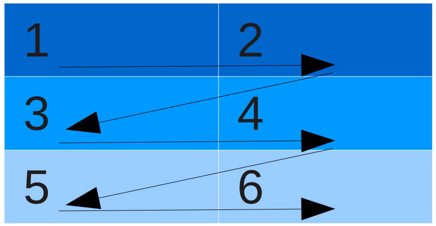
Beyond 3D, graphical representations get awkward (sets of 3D arrays for 4D, sets of 4D for 5D, etc.)

Arrays – MD storage

Internally, they are always 1D

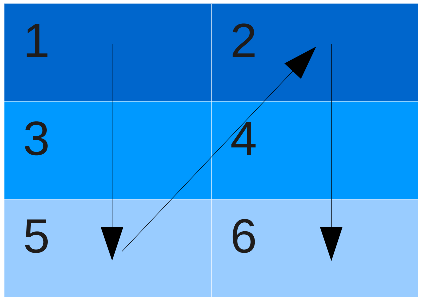
The dimensions are scanned sequentially. Ex (2D): a - 6 elements, 2 columns, 3 rows:

1)
`a[0,0] a[1,0] a[2,0] a[0,1] a[1,1] a[2,1]`
Memory position:
`0      1      2      3      4      5`



or

2)
`a[0,0] a[0,1] a[1,0] a[1,1] a[2,0] a[2,1]`
Memory position:
`0      1      2      3      4      5`



Each language has its choice of dimension order:

Column major – first dimension is contiguous (1 above): IDL, Fortran, R, **Python+Numpy**
Row major – last dimension is contiguous (2 above): C, C++, Java, **Python+Numpy**

Note that languages / people may differ in the use of the terms ***row*** and ***column***.

Graphically, usually the “horizontal” dimension (shown over a line) can be either the first of the last. Usually the horizontal dimension is the contiguous.

Arrays – basic usage

Access to individual elements, through the M indices (MD), or single index (MD or 1D). Ex. (Python):

```
In [73]: import numpy as np

In [74]: a=np.arange(4)
In [75]: b=np.arange(6).reshape((3,2))

In [76]: a
Out[76]: array([0, 1, 2, 3])

In [77]: b
Out[77]:
array([[0, 1],
       [2, 3],
       [4, 5]])
In [78]: a[2],a[-1],a[-2]
Out[78]: (2, 3, 2)

In [81]: a[a.size-2]
Out[81]: 2

In [82]: b[2,1]
Out[82]: 5

In [83]: b[2]
Out[83]: array([4, 5])

In [86]: np.where(a > 1)
Out[86]: (array([2, 3]),)
```

Return arrays of integers where each element has the value of its index.

Negative indices are counted from the end (Python+Numpy, R, IDL≥8): -1 is the last element, -2 the one before the last, etc.

Arrays – basic usage

Accessing slices: subsets, 1D or MD, contiguous or not. Ex. (Python):

```
In [113]: import numpy as np
In [114]: b=np.arange(20).reshape((5,4))
In [115]: b
Out[115]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19]])

In [116]: c=b[2:5,1:3]
In [117]: c
Out[117]:
array([[ 9, 10],
       [13, 14],
       [17, 18]])

In [118]: b[0:3,:]
Out[118]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [122]: b[0::2,1:3]
Out[122]:
array([[ 1,  2],
       [ 9, 10],
       [17, 18]])
```

Elements from columns 1 to 2, from rows 2 to 4

All columns, rows 0 to 2

Columns 1 to 2, rows 0 to last, every second row (stride 2)

Arrays – should I care whether they are row/column major?

For most light, simple use, it does not matter.

When does it matter?

1) **Vector operations**: to select contiguous elements, to use single index for MD arrays.

2) **Mixed language / data sources:**

- When calling a function from another language, accessing files / network connections between different languages.

Arrays – should I care whether they are row/column major?

3) Efficiency:

If an array has to be scanned, it is more efficient (**specially in disk**) to do it in the same order used internally.
Ex: to run through all the elements of this column major array:

a[0,0] (a[0]) : 1	a[1,0] (a[1]) : 2
a[0,1] (a[2]) : 3	a[1,1] (a[3]) : 4
a[0,2] (a[4]) : 5	a[1,2] (a[5]) : 6

In the same order used internally:

```
for j=0,2 do begin
  for i=0,1 do begin
    k=i+j*2
    print,i,j,k,a[i,j]
    do_some_stuff,a[i,j]
  endfor
endfor
```

i	j	**k**	a[i,j]
0	0	**0**	1
1	0	**1**	2
0	1	**2**	3
1	1	**3**	4
0	2	**4**	5
1	2	**5**	6

No going back and forth (shown by variable **k**).

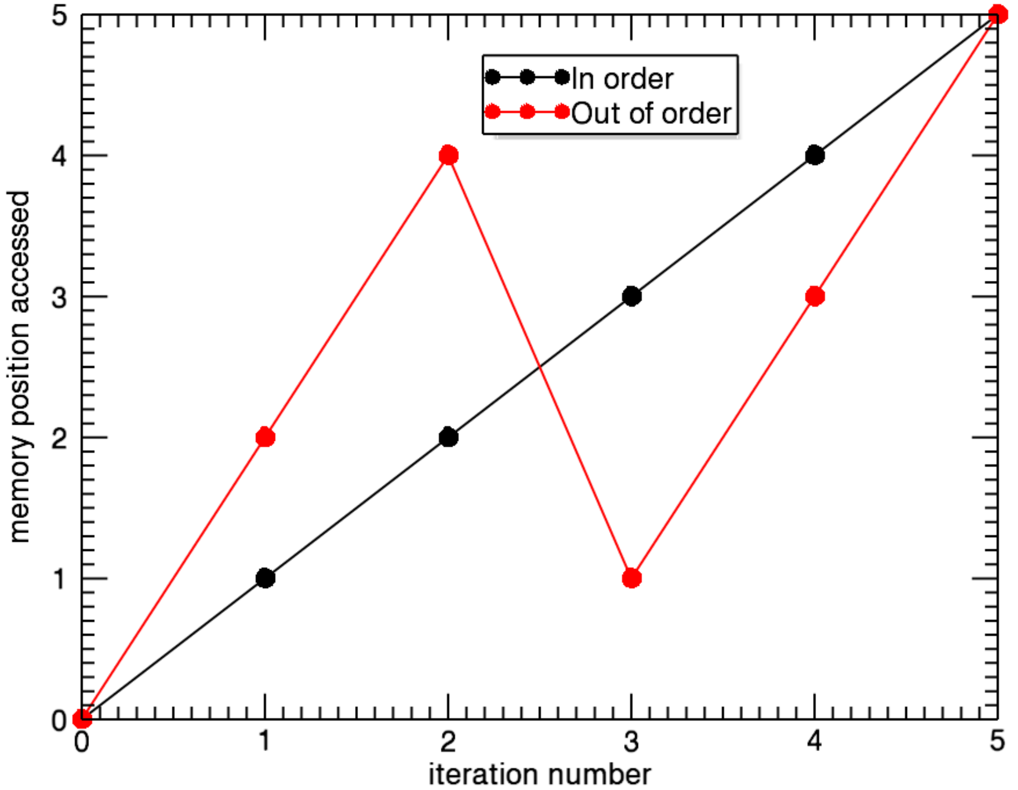
Arrays – should I care whether they are row/column major?

Reading out of order:

```
for i=0,1 do begin
  for j=0,2 do begin
    k=i+j*2
    print,i,j,k,a[i,j]
    do_some_stuff,a[i,j]
  endfor
endfor
```

i	j	k	a[i,j]
0	0	**0**	1
0	1	**2**	3
0	2	**4**	5
1	0	**1**	2
1	1	**3**	4
1	2	**5**	6

Lots of going back and forth:

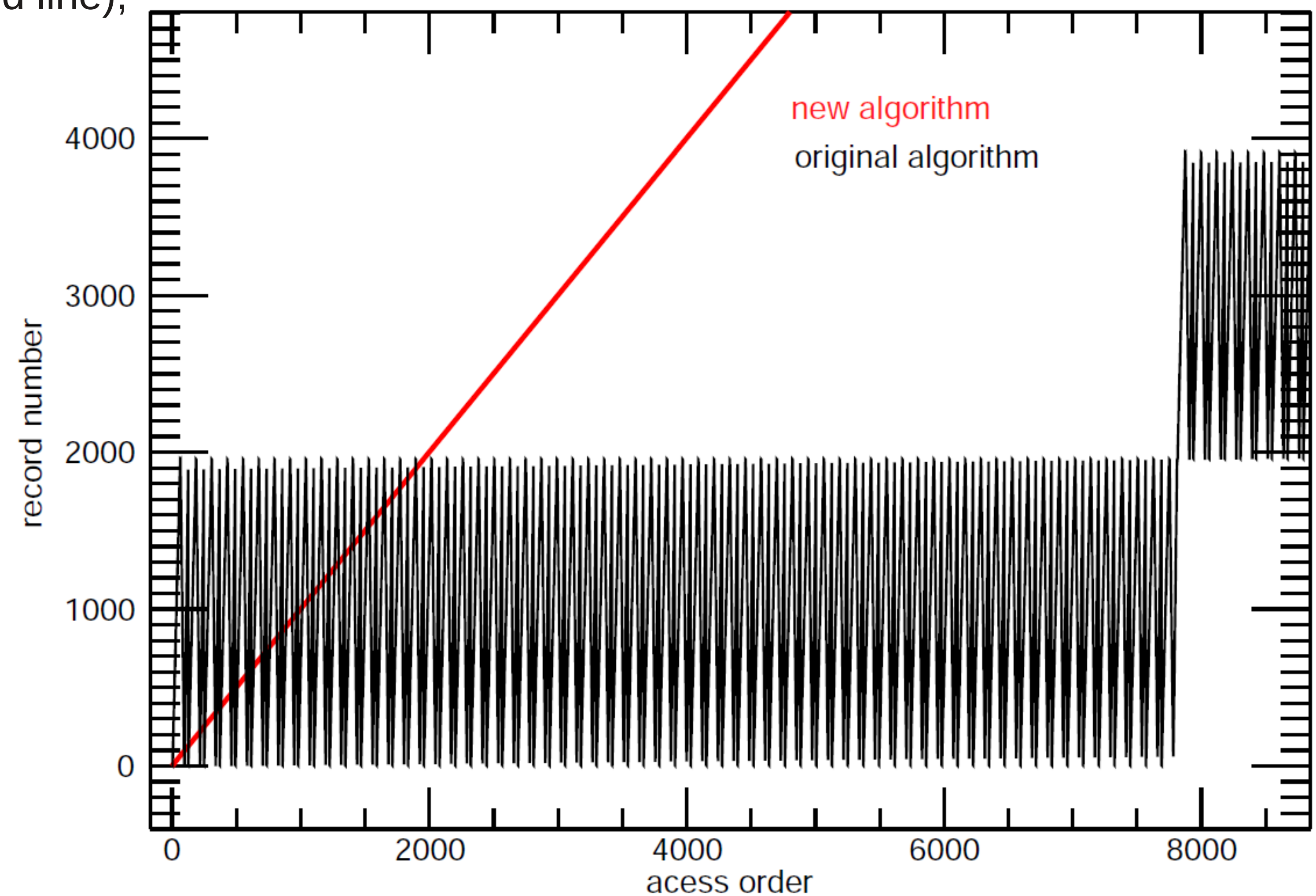


Arrays – should I care whether they are row/column major?

One real life example

The original code read through disk out of order, taking ~1h to run (black line).

When reading in order (red line), the code ran in ~3 min.



Lists - definition

Elements stored **sequentially**, accessed by their indices

- **Similar to 1D arrays.**

Unlike arrays, lists are dynamic, and, in some languages, heterogeneous (IDL, Python, R, Perl)*. Ex. (Python):

```
In [148]: l=[]
In [149]: l.append(2)
In [150]: l.append([5.9,7.0,12.0])
In [151]: l.append(['one','two'])
In [152]: type(l),len(l)
Out[152]: (list, 3)

In [154]: l
Out[154]: [2, [5.9, 7.0, 12.0], ['one', 'two']]

In [155]: l[1]
Out[155]: [5.9, 7.0, 12.0]

In [156]: l.pop(1)
Out[156]: [5.9, 7.0, 12.0]

In [157]: l
Out[157]: [2, ['one', 'two']]

In [158]: l.insert(1,[0,1,2])

In [159]: l
Out[159]: [2, [0, 1, 2], ['one', 'two']]
```

Creates an empty list

Elements added to the list

Removes element from position 1.
If position unspecified, the last
element is removed.

Add element to position 1.

Lists - characteristics

Efficient to add / remove elements, from any place in the list.

- Usually elements are added / removed to the end by default.

Most appropriate when

- The number of elements to be stored is not known in advance.
- The types / dimensions of the elements are not known in advance.
- When there will be many adds / removals of elements.

Lists – application examples

Easy storage of “non-regular” arrays.

Applications where each element in the list contains a different number of elements:

- Elements of
  - Asteroid families
  - Star / galaxy clusters
  - Planetary / stellar systems

- Neighbors of objects (from clustering / classification algorithms)
  - Observations / model results
  - Different number of observations for each object
  - Different number of sources found on each observation
  - Different number of objects used in each model

- Non regular grids
  - Model parameters (models are calculated for different values of each parameter)
  - Grids with non-regular spacing
  - Models with different numbers of objects / species

Lists – application examples

Easy storage of “non-regular” arrays. Exs. (IDL):

```
IDL> l=list()
IDL> l.add,[1.0d0,9.1d0,-5.2d0]
IDL> l.add,[2.5d0]
IDL> l.add,[-9.8d0,3d2,54d1,7.8d-3]
IDL> print,l
       1.0000000       9.1000000      -5.2000000
       2.5000000
      -9.8000000       300.00000       540.00000    0.0078000000
IDL> a=l[2]
IDL> print,a
      -9.8000000       300.00000       540.00000    0.0078000000
```

Dictionaries - characteristics

Similar to structures: store **values** by names (**keys**).

Unlike structures, **keys can be any data type** (most often used: strings, integers, reals).

Unlike indices (arrays and lists), **keys are not sequential.**

Unlike structures, dictionaries are dynamic: elements can be freely and efficiently added / removed.

- Dictionaries are to structures as lists are to 1D arrays.

May be heterogeneous – both keys and values can have different types / dimensions.

Elements may not be stored in order:

- The order the keys are listed may not be the same order in which they were put into the dictionary.

Find out whether a key is present, and retrieve the value from a key are operations that take **constant time**: It does not matter (usually) whether the dictionary has 10 or 1 million elements.

Dictionaries - characteristics

Similar to structures: store **values** by names (**keys**).

Unlike structures, **keys can be any data type** (most often used: strings, integers, reals).

Unlike indices (arrays and lists), **keys are not sequential.**

Unlike structures, dictionaries are dynamic: elements can be freely and efficiently added / removed.

- Dictionaries are to structures as lists are to 1D arrays.

May be heterogeneous – both keys and values can have different types / dimensions.

Elements may not be stored in order:

- The order the keys are listed may not be the same order in which they were put into the dictionary.

Find out whether a key is present, and retrieve the value from a key are operations that take **constant time**: It does not matter (usually) whether the dictionary has 10 or 1 million elements.

- Key/value lookup does not involve searches.
- Like a paper dictionary, a paper phone book, or the index in a paper book.

Dictionaries – basic use (ex. Python):

```
In [185]: d={}
In [186]: d['one']=[9.0,5.8]
In [187]: d[18.7]=-45
In [188]: import numpy as np;d[10]=np.zeros((2,3))
In [191]: type(d),len(d)
Out[191]: (dict, 3)

In [192]: d
Out[192]:
{10: array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]]),
 18.7: -45,
 'one': [9.0, 5.8]}

In [193]: d[10]
Out[193]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])

In [194]: d.keys()
Out[194]: [10, 18.7, 'one']
In [195]: d.values()
Out[195]:
[array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]]), -45, [9.0, 5.8]]
In [197]: del d['one']

In [198]: d.has_key('one')
Out[198]: False
```

Create an empty dictionary

Add values to it

Not the same order they were put in d!

Dictionaries - examples

Storing elements by a useful name, to avoid keep searching for the element of interest. Ex. (IDL): Storing several spectra, by the target name:

```
spectra=hash()
foreach el, files do begin
  read_spectrum,el,spectrum_data
  spectra[spectrum_data.target]=spectrum_data
endforeach
```

Which would be convenient to use:

```
IDL> help,h
H               HASH  <ID=1  NELEMENTS=3>

IDL> print,h
HR21948: { HR21948       5428.1000       5428.1390   5428.1780  5428.2170 ...
HR5438: { HR5438         5428.0000       5428.0390   5428.0780  5428.1170 ...
HD205937: { HD205937     5428.1000       5428.1390   5428.1780  5428.2170 ...

IDL> help,h['HR5438']
** Structure <90013e58>, 7 tags, length=4213008, data length=4213008, refs=6:
   TARGET          STRING    'HR5438'
   WAVELENGTH      DOUBLE    Array[1024]
   FLUX            DOUBLE    Array[1024]
   DATE            STRING    '20100324'
   FILE            STRING    'spm_0049.fits'
   DATA            DOUBLE    Array[512, 1024]
   HEADER          STRING    Array[142]
```

Dictionaries - examples

A lot of freedom in key choice:

- Strings are arbitrary, without the character limitations in structure fields (which cannot have whitespace or special symbols): **`-+*/\()[]{} ,”'`**.

- Special characters commonly appear in useful keys:
  - File names (**`some-file.fits`**)
  - Object names (**`alpha centauri, 433 Eros, 2011 MD`**)
  - Catalog identifier (**`PNG 004.9+04.9`**)
  - Object classification (**`[WC6],R*`**), etc.

- Non-strings are often useful:
  - Doubles – Julian date, wavelength, coordinates, etc.
  - Non consecutive integers, not starting at 0: Julian day, catalog number, index number, etc.

Other containers

Structures are usually implemented as types, but are also containers – **heterogeneous, static and non sequential**:

```
** Structure <9019c628>, 6 tags, length=64, data length=58, refs=2:
   ELEMENT         STRING    'argon'
   INTENSITY       DOUBLE          98.735900
   WIDTH           DOUBLE       0.0087539000
   ENERGY          DOUBLE          12.983800
   IONIZATION      INT             3
   DATABASE        STRING    'NIST Catalog 12C'
   WAVELENGTH      DOUBLE          6398.9548
```

Dictionaries are to structures (both non sequential) as lists are to arrays (both sequential): **the former is the dynamic version of the latter.**

Arrays, lists, structures and dictionaries are the 4 basic containers.

- Most others are specializations of these 4.

Container choice – lists x arrays

Lists and arrays store elements ordered by index. They share many uses.

Differences:

- Lists are dynamic, 1D and may be heterogeneous.

- Arrays are static, homogeneous, and may be more than 1D.

Usually,

- Lists are chosen when one needs:
  - “non regular arrays”
  - add/remove elements (particulalry when the number of elements to store is not known in advance).
  - elements that are not scalar, or not of the same type.

- Arrays are more convenient when one needs:
  - More than 1D
  - vector operations
  - make sure that elements are scalar and of the same type

Container choice – structures x dictionaries

Structures and dictionaries store elements by name. They share many uses.

Main difference:

- Dictionaries are dynamic
- Structures are static

Usually,

- Dictionaries are more convenient when:
  - The keys / types are not known in advance
  - The values may have to change type / dimensions
  - Adding removing fields will be necessary
  - Keys are not just simple strings
- Structures are more convenient:
  - To put them into arrays, to do vector operations
  - To enforce constant type / dimensions of values

Other containers

***Sets* –** similar to dictionaries, but only store keys, without values. Like sets in mathematics.

- Common uses: sets of elements with no repetition: one can just add elements to the set, without having to check if already present.
  - Exs: Sets of: observed objects, files used, observation dates, etc.

- Important for usefulness of set operations: **union, intersection, difference**.

- Dictionaries may be used as sets, ignoring the values

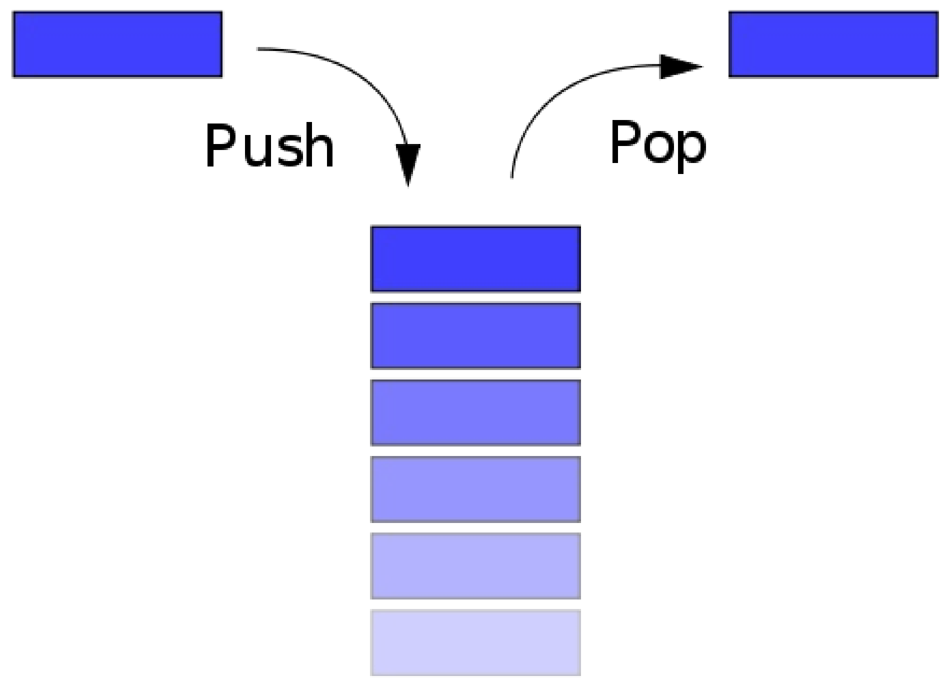
Dictionary:	Keys	HD28598	HD32765	HR592881	HR305710
	Values	sl_00198.fits	sl_00235.fits	sl_00316.fits	sl_00592.fits
Set:	Keys	HD28598	HD32765	HR592881	HR305710

Other containers

Stacks – Lists where elements are only added / removed from the end.

- Like a physical stack: one cannot remove or add a book to the bottom or middle of a stack; only to the top.

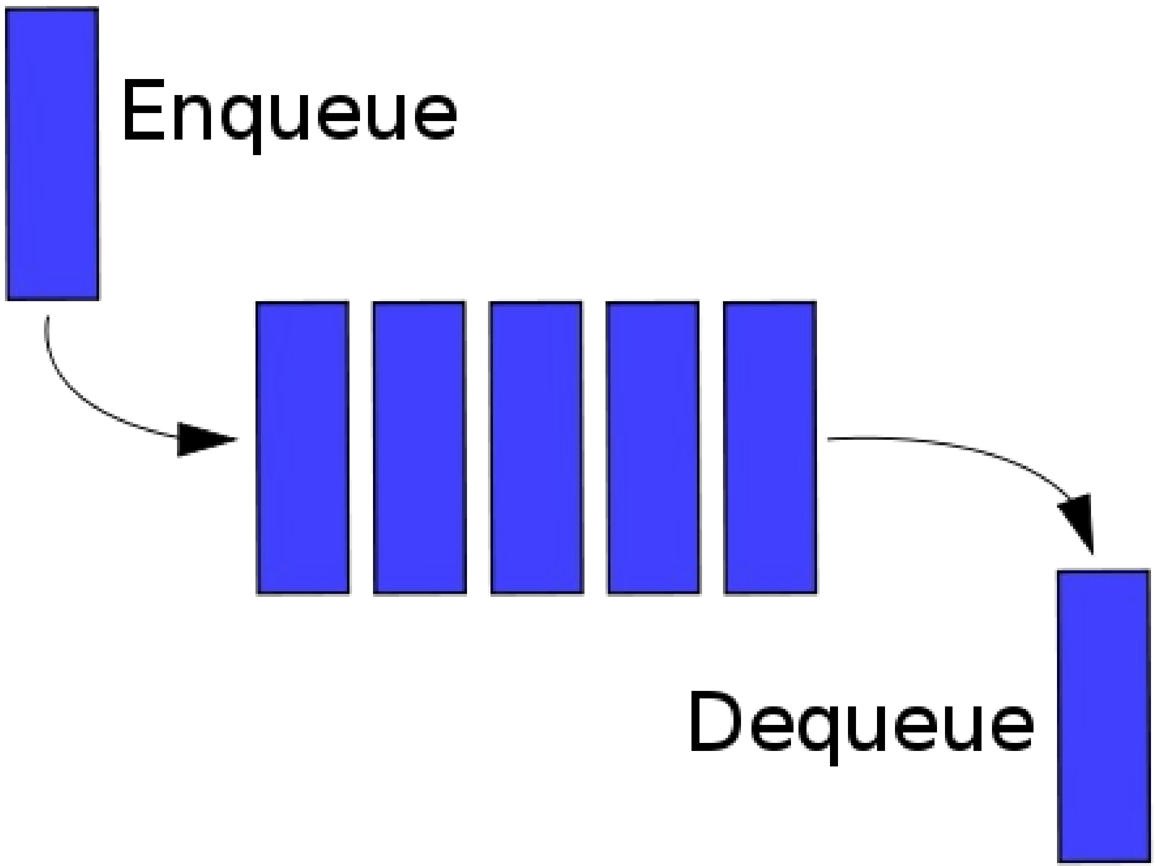
- ***LIFO – Last In, First Out***.



Other containers

Queues – **FIFO (*First In, First Out*) lists**: elements are only added to the end, and only removed from the beginning.

- Like a queue of people waiting to enter some place.



Other containers

Trees / heaps – non sequential containers where access is not by order, nor by name. A hierarchical structures is used:

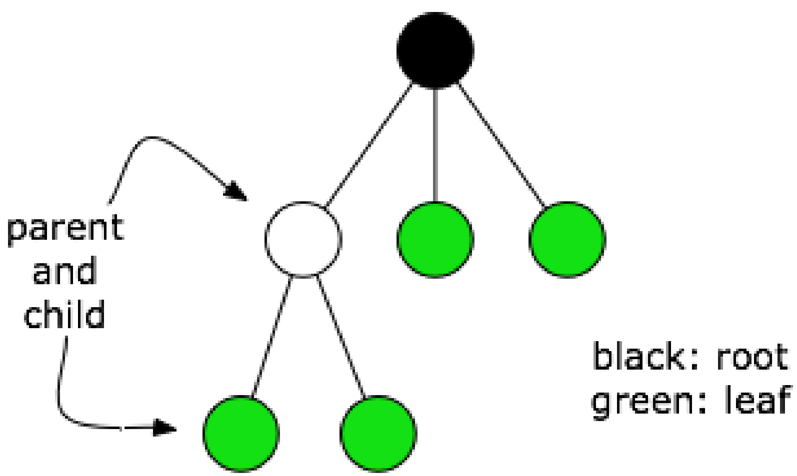
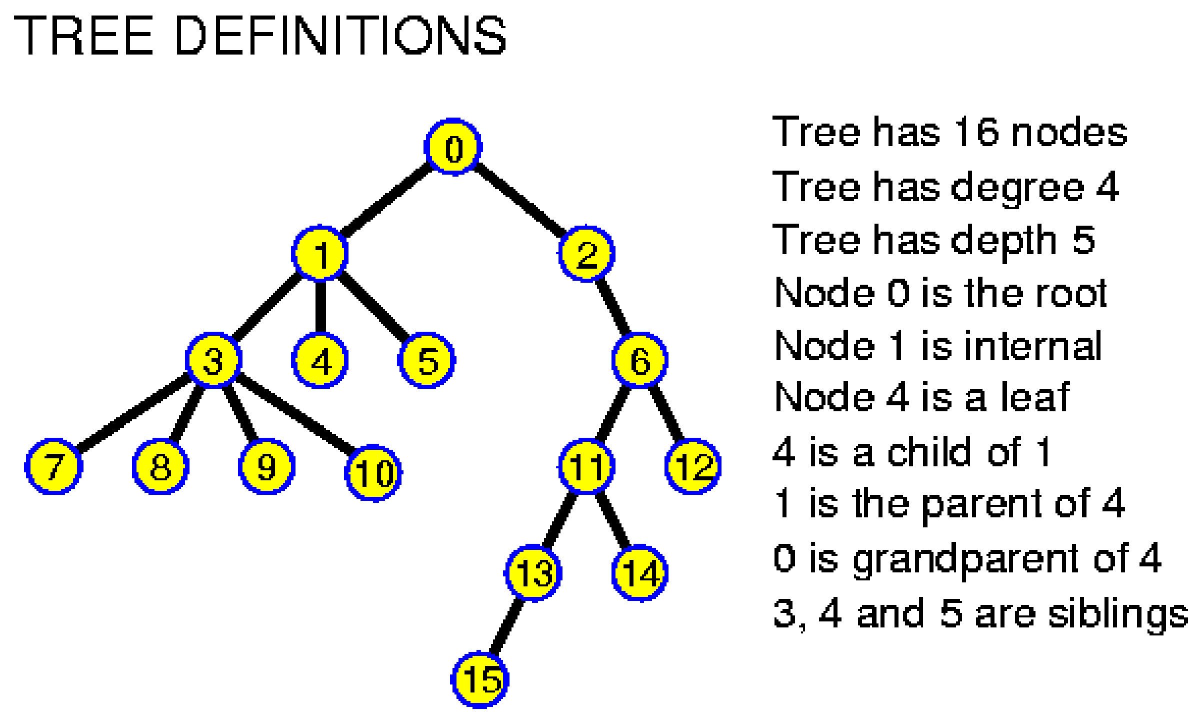


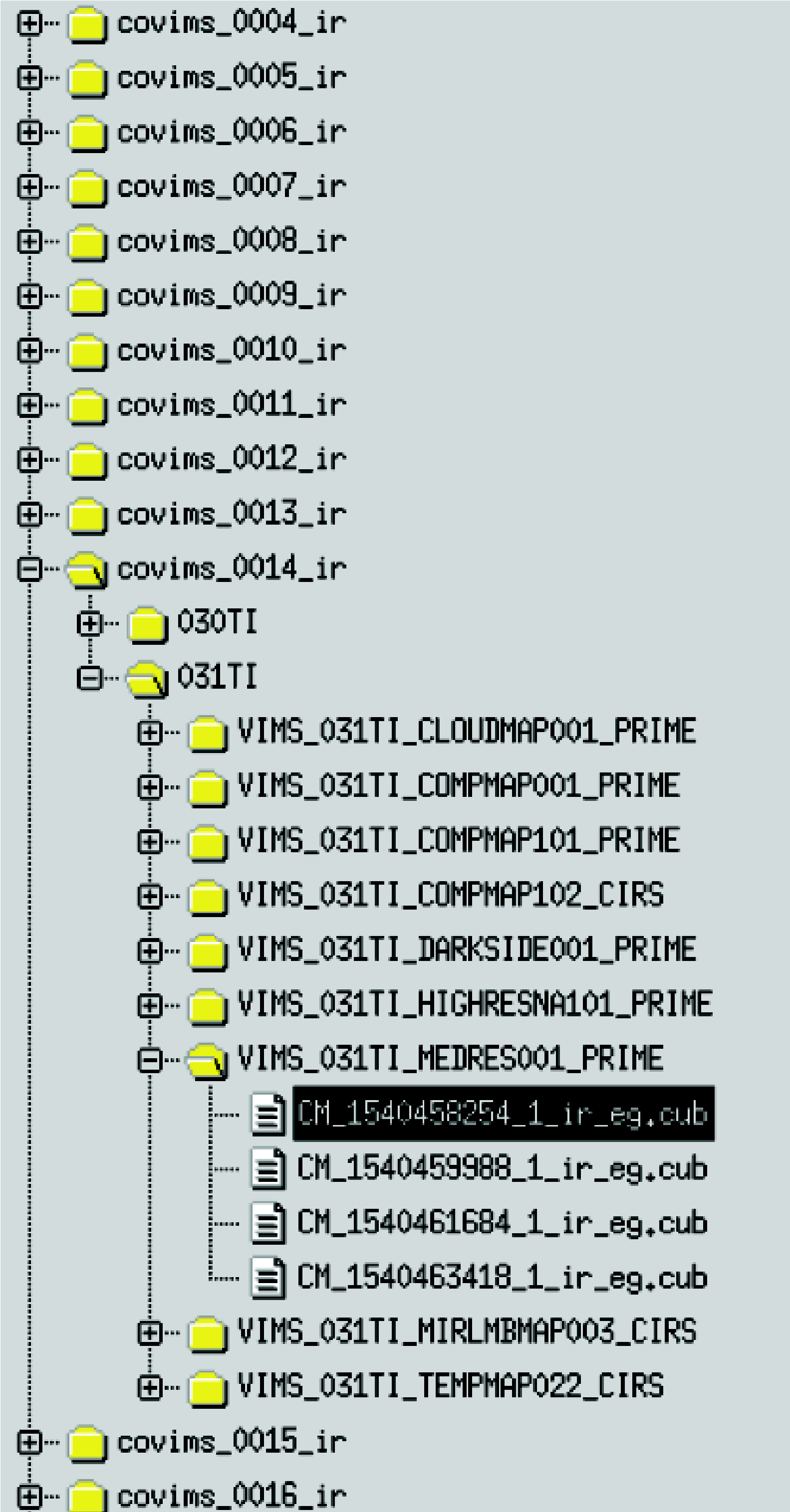
Figure: tree data structure



Other containers - trees

Exs:

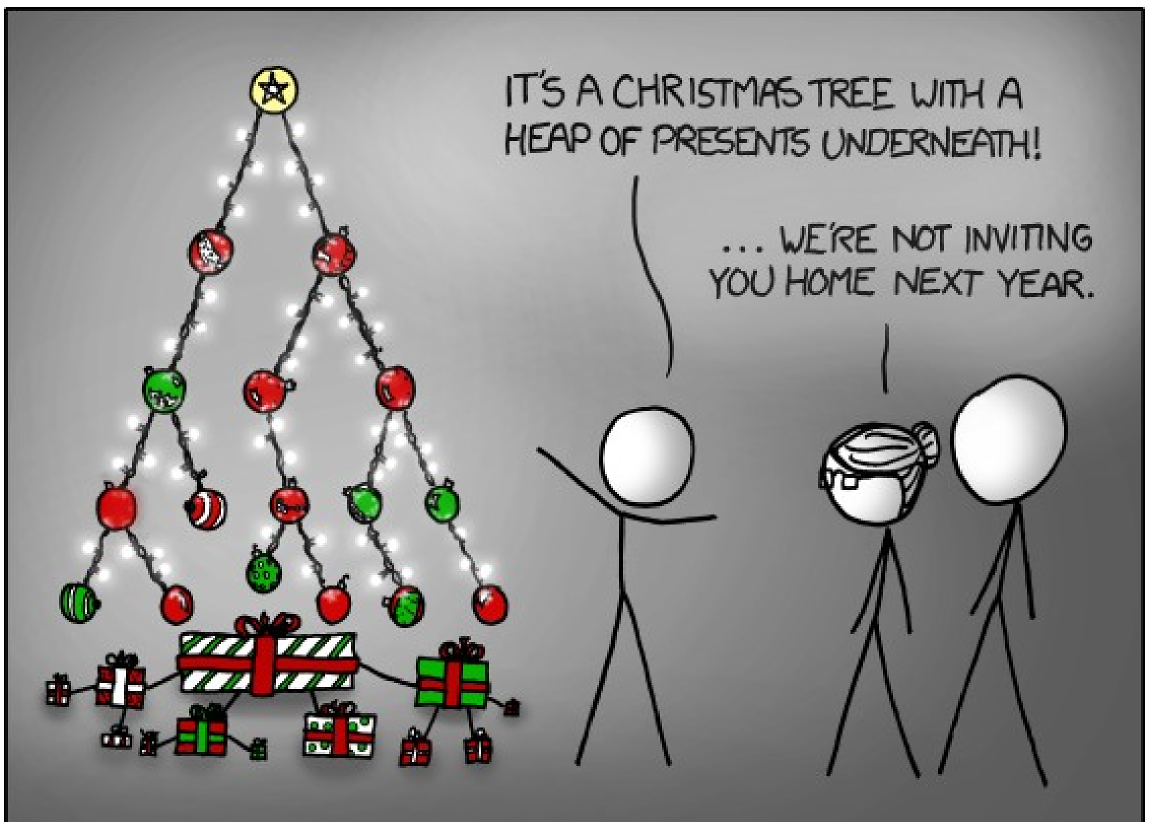
- Directory tree in a disk.
- Hierarchical classifications



Other containers - trees

Exs:

- Directory tree in a disk.
- Hierarchical classifications



(http://www.xkcd.org/835)