DSE1020-DataStructures-9-12-19-prep

September 13, 2018

1 Data Structures

There are many ways to store data in a computer's memory, or on a disk. Think of arranging books on a shelf.

- How would you arrange them if you wanted to look up a book by title?
- How would you arrange them if you wanted to look up a book by author?
- How would you arrange them if you wanted to get to a book by question you were thinking of?
- How easy would it be to find a book by title if they were arranged to make it easy to find by author?

Each data structure arranges data to make some tasks easier and others harder. Typically a data structure which makes one task the most easy will make others pretty difficult. There are data structures which are compromises that make no tasks very easy but nothing very hard.

Also, sometimes you worry about space as a resource, and sometimes you worry about time.

- If you had lots of money and lots of space how could you make it easy to look up a book by title or author as fast as possible?
- If you had minimal money and very little space but time wasn't a problem how would you arrange the books to look up a book by title or author?

1.1 Types of Data Structures

There are many types of data structures. We can also look at data structures two ways.

Abstract Data Structures: What we can do with them

Data Structure Implementations: How they are built

I will give you some references so you can study 2 in your free time. We will have some homeworks that have you implement a couple of data structures. Right now I want to talk more about 1. We will talk about some data structures that are very common and what there advantages are. Here is a list of some important ones:

1. Arrays
2. Stacks and Queues
3. Key-Value stores
4. Trees
5. Composite

1.2 Arrays

An array is a fixed length set of fixed size boxes. Each box can store something of the same type, which means the same width. We think of strings this way:

H	e	l	l	o	_	W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11

Where each character is a byte, really a number from 0-255. We can give this array a name. Lets call it barray:

```
In [1]: barray = b'Hello World!'

In [2]: barray[0]

Out[2]: 72

In [7]: str(barray[0:1],'utf-8')

Out[7]: 'H'

In [8]: str(barray[11:12],'utf-8')

Out[8]: '!'
```

1.2.1 Numpy Arrays

A good example of an array data structure is a numpy array.

```
In [9]: import numpy as np

In [ ]: help(np.ndarray)

In [13]: narray = np.ndarray(100,dtype='int')

In [14]: narray[99]

Out[14]: 0

In [15]: narray[200]


        ---------------------------------------------------------------------------

        IndexError                                Traceback (most recent call last)

        <ipython-input-15-5a896ee0efe3> in <module>()
    ----> 1 narray[200]


        IndexError: index 200 is out of bounds for axis 0 with size 100
```

```
In [16]: for ind in range(100):
             narray[ind]=ind+10

         print(narray)
```

```
[ 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  36  37  38  39  40  41  42  43  44  45
  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63
  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81
  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99
 100 101 102 103 104 105 106 107 108 109]
```

```
In [17]: narray[55]
```

```
Out[17]: 65
```

```
In [19]: narray[55]=-99
```

```
In [20]: print(narray)
```

```
[ 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  36  37  38  39  40  41  42  43  44  45
  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63
  64 -99  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81
  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99
 100 101 102 103 104 105 106 107 108 109]
```

```
In [21]: narray[10]='eyeballs'
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

<ipython-input-21-5bde058f53b1> in <module>()
----> 1 narray[10]='eyeballs'

ValueError: invalid literal for int() with base 10: 'eyeballs'
```

```
In [22]: narray[10].dtype
```

```
Out[22]: dtype('int64')
```

```
In [23]: narray[10]=1.1
```

```
In [24]: narray[10].dtype
```

```
Out[24]: dtype('int64')

In [25]: narray[10]

Out[25]: 1
```

Question 1: How can we expand the array so we can add more items at the end?
Question 2: How can we delete the third element from the middle of the array?
Question 3: How can we insert an element between the 4th and 5th element of the array?

1.3 Stacks and Queues

A key part of an array is that it is a fixed size. As we saw, we can only insert or delete elements (which changes the size) by creating a new array. Lets turn to data structures whose key features are that they change size dynamically.

1.3.1 Stacks

A key practical problem is keeping a list of changes so we can undo. Lets thing of a simple thing. We have a string and every time we change it, we want to store that change. We only care about "undos" and we don't need to index arbitrarily or "redo". Lets say we have this sequence of edits:

- 'H'
- 'He'
- 'Hel'
- 'Hell'
- 'Hellp'
- 'Hell'
- 'Hello'

So we want to store these in a way that each time we edit we can save. We call this "push". And each time we want to go back to the previous edit we can get the last version. We call this "pop". Push will expand the data structure to accommodate the new item. Pop will shrink the data structure as we remove one item.

We will use python classes and python lists to make this.

```
In [38]: class Stack:
             def __init__(self):
                 self._data = []

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

             def pop(self):
                 return self._data.pop()
```

Yes ... I know we are cheating here with python lists. Python lists are not arrays, and not stacks. They are one of those complex data structures that are compromises in order to do many things pretty well. There are also python tuples which are closer to arrays but still different. We will say a few words later but I will give you references if you like.

Note 1: We use the '_data' to indicate a private variable. People using our "Stack" class should never reference *'data' or any method/variable starting with a '*'. Unlike Java or C++ Python doesn't block dev from accessing private data (for example for debugging). It is just never make it into production code.

Note 2: If you heard about stacks before, you might know that you put data and take data off the top of the stack. You might think that should be '_data[0]', ie. the first element. Here it is the last element.

```
In [39]: buffer = Stack()

In [40]: buffer.push('H')
         buffer.push('He')
         buffer.push('Hel')
         buffer.push('Hell')
         buffer.push('Hellp')
         buffer.push('Hell')
         buffer.push('Hello')

In [41]: while True:
             print(buffer.pop())

Hello
Hell
Hellp
Hell
Hel
He
H
```

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)

<ipython-input-41-8876f26b1e19> in <module>()
  1 while True:
----> 2     print(buffer.pop())


<ipython-input-38-6212c76f2122> in pop(self)
  7
  8     def pop(self):
----> 9         return self._data.pop()
 10


IndexError: pop from empty list
```

```
In [43]: class Stack:
             def __init__(self):
                 self._data = []

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

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

             def isEmpty(self):
                 return len(self._data) == 0
```

```
In [44]: buffer = Stack()
         buffer.push('H')
         buffer.push('He')
         buffer.push('Hel')
         buffer.push('Hell')
         buffer.push('Hellp')
         buffer.push('Hell')
         buffer.push('Hello')
```

```
In [45]: while not buffer.isEmpty():
             print(buffer.pop())
```

```
Hello
Hell
Hellp
Hell
Hel
He
H
```

Question 1: How do we iterate though the stack using only push and pop non-destructively?
Question 2: How do we add a function to a stack to index into it using only push and pop non-destructively?

1.3.2 Queues

When we think about waiting in a line at the post office, the last person on the line is not the first person to get served (or shouldn't be) like a stack. Just like a stack you take things off the top list of the list. Only in a queue you add things to the end of the list. Since we already have "pop" using the last element as the "top" of the list, all we need to change is make the first element the "bottom" of the list.

Question 1: How can we implement a queue class similar to stack?
Hint: If s is a list then 's.insert(i, item)' inserts 'item' at index 'i'

```
In [54]: class Queue:
             def __init__(self):
                 self._data = []

             def enqueue(self, item):
                 self._data.insert(0,item)

             def dequeue(self):
                 return self._data.pop()

             def isEmpty(self):
                 return len(self._data) == 0

In [55]: post_office = Queue()

In [68]: post_office.enqueue('Sally')
         post_office.enqueue('Ellen')
         post_office.enqueue('Bob')
         post_office.enqueue('Pria')
         post_office.enqueue('Leroy')

In [59]: while not post_office.isEmpty():
             print(post_office.dequeue())

Sally
Ellen
Bob
Pria
Leroy
```

Advantages: * Easy to add * Easy to delete
Disadvantages: * Slower index * Slower to loop through * More space than arrays

1.3.3 Notes on python lists

Python lists go way beyond stacks or queues. They have many methods. You have probably seen some of the important ones. Just to highlight a few you will probably use often, if 's' is a list then

- s.append(item): adds to the end
- s.count(item): number of items in 's' that match 'item'
- s.index(item): returns the index of first match of item in 's'
- s.insert(ind, item): inserts the item at index 'ind'
- s.remove(ind): removes an item at index 'ind'
- s.reverse(): reverses all the items in place (and returns nothing)
- s.sort(): sorts the elements on a list

We also have overloaded functions like 's1 + s2' which concatenates list 's1' with list 's2' and 's[ind]' which gives the item at index 'ind'.

Python Lists and References To understand python lists you need to understand a bit about references which is kind of behind the scenes. Each python object has an "id"

```
In [69]: x = "foobar"

In [70]: id(x)

Out[70]: 4681129744

In [71]: y = x

In [72]: id(y)

Out[72]: 4681129744

In [73]: y = "foobar"

In [74]: id(y)

Out[74]: 4681129744

In [75]: x = "dingo"

In [76]: id(x)

Out[76]: 4681129072

In [77]: id(y)

Out[77]: 4681129744
```

Things get complicated with lists of lists.

```
In [79]: s = [[],[],[],[]]
         print(s)

[[], [], [], []]


In [81]: s[1].append(1)
         print(s)

[[], [1, 1], [], []]


In [83]: t = []
         r = [t,t,t,t]
         print(r)

[[], [], [], []]
```

```
In [85]: r[1].append(1)
         print(r)

[[1, 1], [1, 1], [1, 1], [1, 1]]
```

Question 1 What is the difference?

```
In [88]: id(s[0]), id(s[1]), id(s[2]), id(s[3])

Out[88]: (4681203912, 4487616520, 4681056328, 4681005320)

In [89]: id(r[0]), id(r[1]), id(r[2]), id(r[3])

Out[89]: (4681059656, 4681059656, 4681059656, 4681059656)
```

1.3.4 Python standard libraries

collections.deque: Python 3.5+ has a collections library that has a number of useful data structures. One is the deque object which implements a stack-queue combination which can be used for either.

queue: Python 3.5+ has a queue standard library that provides different flavors of queues. You might wonder why you need so many if you can just use lists. This has to do with concurrency. If different threads are adding and deleting things *at the same time* from the queue one has to be very careful that they all see the same queue.

1.4 Key-Value Store

There are many situations where you need a very fast lookup. For example, lets say we have a student id. A university administrator should be able to take the student id and look up, for example, the student's last name. We could do this with two lists:

```
In [93]: np.random.seed(10)
         id_list = [np.random.randint(10000,999999) for ind in range(5)]
         name_list = ['Sally', 'Ellen', 'Bob', 'Pria', 'Leroy']
         print(id_list)

[355353, 770957, 891167, 453712, 627841]


In [96]: name_list[id_list.index(770957)]

Out[96]: 'Ellen'
```

This kind of lookup is going through the whole list and looking for the first match. This is not super-fast, particularly if the list is long and the comparisons are complex. There is a much better data structure.

1.4.1 Python Dictionaries, Hash Tables

Python dictionaries use what are called hashes to implement fast lookup. The items we are going to use to look up information are called "keys" and the targets of the lookups are called "values". In our example in the last section the keys are 'id_list' and the 'values' are 'name_list'. We can build a table that allows for fast lookup as follows:

```
In [99]: name_lookup = {355353:'Sally', 770957:'Ellen', 891167:'Bob',  453712:'Pria',  627841:
         print(name_lookup)
```

```
{355353: 'Sally', 770957: 'Ellen', 891167: 'Bob', 453712: 'Pria', 627841: 'Leroy'}
```

```
In [106]: # We could do it one, by one
          name_lookup = {}
          name_lookup[355353]='Sally'
          name_lookup[770957]='Ellen'
          print(name_lookup)
```

```
{355353: 'Sally', 770957: 'Ellen'}
```

```
In [102]: # Make key value pairs
          print(list(zip(id_list,name_list)))
```

```
[(355353, 'Sally'), (770957, 'Ellen'), (891167, 'Bob'), (453712, 'Pria'), (627841, 'Leroy')]
```

```
In [108]: # One shot
          name_lookup = dict(zip(id_list,name_list))
          print(name_lookup)
```

```
{355353: 'Sally', 770957: 'Ellen', 891167: 'Bob', 453712: 'Pria', 627841: 'Leroy'}
```

```
In [109]: print(name_lookup[891167])
```

```
Bob
```

```
In [110]: name_lookup.keys()
```

```
Out[110]: dict_keys([355353, 770957, 891167, 453712, 627841])
```

```
In [111]: name_lookup.values()
```

```
Out[111]: dict_values(['Sally', 'Ellen', 'Bob', 'Pria', 'Leroy'])
```

```
In [112]: name_lookup.items()
```

```
Out[112]: dict_items([(355353, 'Sally'), (770957, 'Ellen'), (891167, 'Bob'), (453712, 'Pria'),
```

```
In [113]: del name_lookup[355353]
          print(name_lookup)
```

```
{770957: 'Ellen', 891167: 'Bob', 453712: 'Pria', 627841: 'Leroy'}
```

Hash The keys have to be 'hashable' which means that it an immutable thing that can't be modified a key must be replaced with a new key. Example a string or an int can be a key. A list (mutable) can not. A tuple (immutable) can be.

```
In [114]: hash(770957)

Out[114]: 770957

In [115]: hash('Ellen')

Out[115]: -8624026802654923174

In [116]: hash([])


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-116-4c7351eba020> in <module>()
    ----> 1 hash([])


        TypeError: unhashable type: 'list'
```

The way that a 'hash table' works under the hood is that the hash number, under-the-hood, is changed into an index into a table. Because it a function that goes from key to number to index in table, this is very fast. Things get more complex because in building the table initially, two hashes may accidentally map to the same index. In this case, a second hash is used to find another spot. Read elsewhere about implementation of hash-tables.

Advantages: * Fast lookup by key * Easy to add * Easy to delete

Disadvantages: * Generally harder to list or do a search by value * Harder to run through than queues, stacks, or arrays * More space than queues, stacks, or arrays

1.4.2 python standard library: collections

The collections python standard library has a number of extended dictionary versions. For example there is no gaurenteed ordering in ordinary dictionaries. An OrderedDict remembers the order that key-value pairs were inserted.

1.5 Trees

Often data is hierarchical. The "tree of life" is a great example. A great panda is a panda is a mammal is a vertebrate is an animal. It may be that one wants to be able to answer questions like list all mammals, or find the common category between a panda and a shark, or list all the categories of a particular animal.

Another example is directories on a file system (ignoring symbolic links). Python does not have any built in tree libraries but has many third party data structures libraries. One example is treelib code, docs.

```
In [120]: # installing
          ! pip install treelib
```

```
Collecting treelib
  Downloading https://files.pythonhosted.org/packages/cf/4f/f6dc76341c438a84672386a5db57c1a49b3
Building wheels for collected packages: treelib
  Running setup.py bdist_wheel for treelib ... done
  Stored in directory: /Users/michael/Library/Caches/pip/wheels/c4/29/a2/1bd8145c2898f2b9f2a23a
Successfully built treelib
twisted 18.7.0 requires PyHamcrest>=1.9.0, which is not installed.
Installing collected packages: treelib
Successfully installed treelib-1.5.3
You are using pip version 10.0.1, however version 18.0 is available.You should consider upgradi
```

```
In [121]: from treelib import Node, Tree
```

```
In [122]: tree = Tree()
          tree.create_node("Harry", "harry")  # root node
          tree.create_node("Jane", "jane", parent="harry")
          tree.create_node("Bill", "bill", parent="harry")
          tree.create_node("Diane", "diane", parent="jane")
          tree.create_node("Mary", "mary", parent="diane")
          tree.create_node("Mark", "mark", parent="jane")
          tree.show()
```

```
Harry
 Bill
 Jane
     Diane
        Mary
     Mark
```

```
In [125]: diane_node = tree.get_node("diane")
          print(diane_node)
```

```
Node(tag=Diane, identifier=diane, data=None)
```

```
In [126]: print(diane_node.tag)
```

```
Diane
```

```
In [128]: print(diane_node.is_leaf())
```

```
False
```

```
In [132]: # Get child
          diane_node.fpointer

Out[132]: ['mary']

In [133]: # Get parent
          diane_node.bpointer

Out[133]: 'jane'

In [140]: # Trace to root
          node = diane_node
          trace_list = []
          while not node.is_root():
              trace_list.append(node.tag)
              node=tree.get_node(node.bpointer)
          trace_list.append(node.tag)
          trace_list.reverse()
          print(' -> '.join(trace_list))


Harry -> Jane -> Diane
```

1.5.1 Other libraries and data structures

- An important implementation of a tree as a file format is the Hierarchical Data Format version 5 (HDF5) file format is an extreemly popular scientific data format based on trees. Each node can store whole numpy arrays as data, for example. A popular library for this is pytables code docs
- Something that generalizes lists and trees is a graph. A great library with lots of algorithms for general graphs (including trees) is NetworkX code docs

1.6 Composite

There are many ways to build data structures, to save them, to access them. Typically you will use multiple data structures at the same time. One common composite important data structure is a table. We can think of a table as having rows and columns.

We can think about each row as having data. The name of data is the column name (key) and the value is whatever that row has for that column. Thus we can think of a table as a list of dictionaries. A composite of lists and dictionaries.