Data Structure in Python for Data Analytics

Prof. Zhang
CISC5835, Fall 2018

It's important to know basic data structures and understand the performance impacts of choosing a certain data structure (i.e., running time of the operations to insert, delete, search for an element).

1. Basic Built-in Python Data Structures

Reference: https://docs.python.org/3/tutorial/datastructures.html

a. List

List elements are stored in contiguous memory, as a result accessing list element (i.e., a[i]) takes a constant time: taking the starting address of the list, and calcualte the offset for i-th element, and then add that to the starting address to get the address of i-th element.

a) list initialization

```
fruits = ['orange', 'apple', 'pear']
inputs=[]   ## an emapty of list

print(fruits[0]) ## orange
print(fruits[1])  ## apple
print (fruits[2]) ## pear
```

b) iterate through list elements

```
for f in fruits:  ## repeat an operation for each element f from list fruits
        print (f)
```

c) iterage through list elements using index

```
print(len(fruits))  ## 3
for i in range (len(fruits)):
        print (i)   ## 0 1 2
        print (fruits[i])
```

d) insert, remove, index

```
fruits.insert(1,'grape')  ## ['orange','grape', 'apple', 'pear']

fruits.remove('pear')   ## ['orange','grape', 'apple']
fruits.pop() ## remove and return last element
fruits.pop (1) ## remove fruits[1] and return it
```

```
fruits = ['orange', 'apple', 'pear','grape']
fruits.index('apple')  ## return 1
fruits.index('apple',2,3) ## look for apple in fruits[2…3] sublist
```

e) sort list

```
fruits.sort()   ##
```

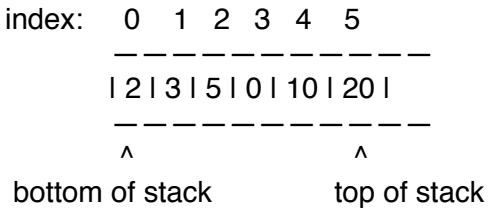
b Stack: LIFO, FILO

Data structure where elements insertion/deletion/access must follow the principle of Last In First Out, or First In Last Out. A stack data structure usually provides the following operations:

- push(x): push a new element x to the top of the stack
- pop (x): remove and return the top element in the stack
- top (x): return the top element in the stack

Efficiency of append(), pop()?

Stack can be easily implemented using list, where the end of the list is considered to be the top of stack, the beginning of the lsit is the bottom of the stack.



1) To push an element to the stack: use append() operation of list

```
s = [2,3,5, 0, 10, 20]
s.append (5) ## put 5 on the top of stack
print (stack)
```

2) To pop an element from the top of the stack: use pop() operation of list

```
s.pop() ## remove the top element (i.e., the last one being added, in the end of the list
```

3) To obtain the top element of the stack

```
s[0]
```

Effiency of these operations are all constnat-time operation.

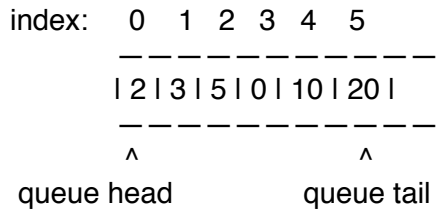
c. Queue: FIFO

Insertion/Deletion/Access of elements in a queue must follow First In First Out principle. A queue data structure supports the following operations:

- enqueue(x): append an element x to the end of the queue
- dequeue(): remove the element from the head of the queue, and return it

Queue can be implemented using list, or circular buffer/array/list.

1) implement queue using list




```
q = [2,3,5,0,10,20]

## enqueue operation:
   q.append (100)   ## enqueue 100 into q, put it in the tail of the queue

## dequeue operation
  q.pop(0) ## dequeue, i.e., remove first element in q (i.e., element at index 0)
           ## and return it
```

Efficiency of enqueue operation: constant time
running time of dequeue operation: O(n) as one needs to move all other elements forward to fill in the hole

2) Due to the linear time dequeue operation, a different implementation of queue is preferred: collections.deque

```
from collections import deque
queue = deque(["Eric", "John", "Michael"])
queue.append("Terry")          # Terry arrives: enqueue Terry
queue.append("Graham")         # Graham arrives
queue.popleft()              # The first to arrive now leaves ‘Eric’, dequeue
queue.popleft()              # The second to arrive now leaves 'John'
   ## deque([‘Michael’, 'Terry', ‘Graham'])
```

d tuple

A tuple consists of a number of values separated by commas; immutable, usually contain a heterogeneous equence of elements. (Lists are mutable).

```
# using tuple to swap two variables
        x=2
        y=3
        x,y=y,x


## using tuple to return mutliple values
def MinMax(a):
    min = a[0]
    max = a[0]

    for i in range (1,len(a)):
        if (min>a[i]):
            min = a[i]
        if (max<a[i])
            max = a[i]

    return min, max

l = [34, 4, 100, -23]
smallest, largest = MinMax (l)
```

e. sets

A set is an unordered collection with no duplicate elements. Opertaions include: testing membership, union, intersection, difference, and symmetric difference.

```
        a = set('abracadabra')   ## set() creates a set of char elements from the string of char
        b={'a','l','a','c','a'}

        alb ## union
        a & b ## a intersect b
        a-b  ## letters in a but bot in b
        a ^ b ## letters in a or b not both
```

2. heap

Reference:
https://interactivepython.org/courselib/static/pythonds/Trees/BinaryHeapImplementation.html

In a lot of applications/algorithms, we need to maintain a dynamic collection of items, and need to access/remove the item with the smallest key value. For example, a priority queue where we want to remove the item with the smallest priority level (refer to most urgent/important task).

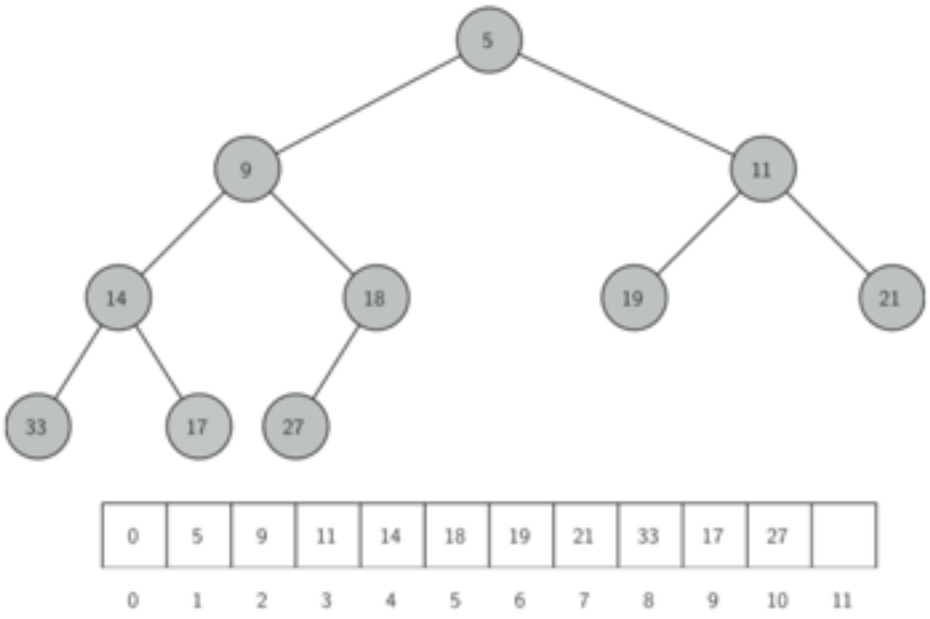
Heap is a data structure that's suitable for such application.

Complete binary tree: a structure where everyone node has at most one parent node, and at most two child node (binary tree), furthermore, all nodes except those at the bottom level have two child nodes (complete).

Storing a complete binary tree in a list: root node, nodes at first level, nodes at second level, … nodes in each level is stored from left to right order.

Q: What's the index of the parent node of a node at index i? How to find the left and right children nodes for a node at index i?

For a node at i, its parent node is at i/2
its left child node is at 2*i, and right child node is at 2*i+1



^^ here the first (zero-th) slot is not used.

Min-Heap property: For every node in the complete binary tree, the value stored in the node is smaller than those stored in the node's child or children

Demo: heapq algorithm (the library implements key operations for heap on a list)

```
import heapq
heap=[]
data=[18,19,11,14,9,5,33,14, 27, 17, 22, 19, 21]

heapify (data) ## transform list data into a heap, in linear time
print (data)

## add element into heap
newData=[200,300]
for item in newData:
        heappush(heap, item)   ## log N time operation

print(heap) ##[0, 2, 11, 3, 5, 14, 17, 14, 9, 18, 33, 20, 27, 19, 22, 19, 21]

## Another example, heap elements could be a tuple.
heapq=[]
heappush(heapq, (1,'handicapped'))
heappush(heapq, (4,'regular'))
heappush(heapq, (2, 'fast pass'))
heappush(heapq, (3, 'singler rider'))
heappush(heapq,(1,'handicapped'))
heappush(heapq,(2,'fastpass'))
print (heapq)   ##

heappop (heapq) ## remove smallest element and return (1,'handicapped'))
                ## log N time operation
```

How various heap operations are implemented and their efficiency?

1. How to take advantage of heap data structure to sort a list?

```
data=[18, 19,11, 14, 9, 5]
heapify (data)

sortedData=[]

## remove smallest element to append into sortedData
for i in range(len(data)):
    sortedData.append (heappop (data))
```

2. How to acess the smallest element in min-heap? How to remove it?

1) the smallest element is in the top of the heap, a[1]
2) swap last element with a[1]
3) repair heap property by **heap-down operation starting at root node**, where the larger element is sunk down, more specifically

```
i = 1; ## start from root node

do {
   leftChild = 2*i
   rightChild = 2*i+1

   ## find the smallest among a[i], a[leftChild], a[rightChild]
   smallest = i
   if a[leftChild] < a[smallest]:
         smallest = leftChild
   if a[rightChild] < a[smallest]:
         smallest = rightChild

   ## if i does not store the smallest among the three, swap
  if i!=smallest:
      a[smallest], a[i] = a[i], a[smallest]   ## swap by assign tuple
      i = smallest
  else
       done=true
} while (done!=true and i is not a leaf node);
```

3. Add a new item into the heap, e.g., 6? Where should 6 be stored in the heap?

1) append the element into the end of the list
2) repair heap property by **heap-up operation starting from this new leaf node**, where the smaller element is bubbled up.

The heap-up operation is similar to the heap-down operation, the difference is that we compare the element with its parent, and swap them if the element is larger than its parent. Then we repeat the process for the parent and the parent's parent, until we reach the root node, or we do not make any swap.

4. heapfiy operation: how to take a list and rearrange elements in the list so that min-heap property is stasfieid?

Please use this resource to study heap operation:

https://www.cs.usfca.edu/~galles/visualization/Heap.html

3. dictionary data structure

Dictionary (or hashtable, hashmap) are offen called “associative memory” or “associate array” that stores a set of key-value pairs, where the keys are unique. Unlike list, dictionary is indexed by keys.

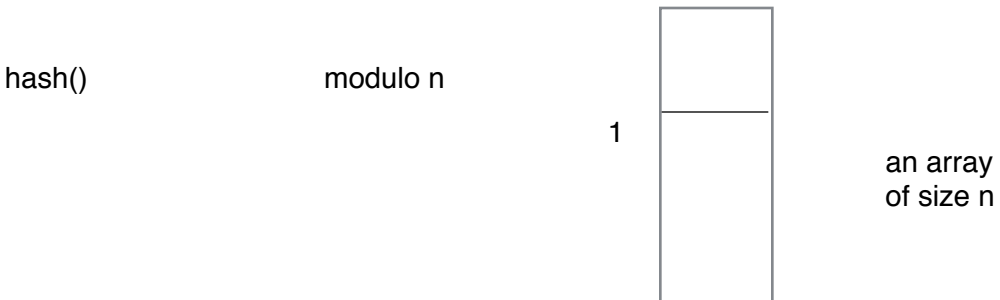
```
# phonebook is a set of key-value pairs, mapping name to phone #
phonebook={}
phonebook['alice']=123456789   ## store a new key-value pair
phonebook['jack']=111111111

phonebook['jack']=22222 # modify value for key 'jack'

print (phonebook)
print(phonebook['alice'])

del phonebook['alice']  ## remove a pair from the dict

'jone' in phonebook ## checking if a key is in the dict… false

mypb={'bob':2322, 'alice':1234,'jack':8931}
```

The keys of dictionaries can be any hashable type, i.e., there is a method to map the key to a hash value (an integer) which never changes during its lifetime.

All built-in types (except mutable object such as list, dict) are hashable:

```
hash (123)   ## 123
hash ('jack')  ##  3539536964226342402
hash ((1,2))  ## tuple is hashable: 3713081631934410656
hash ({'a','b'})  ## ERROR: unhashtable type: set
```

When accessing dictionary, e.g., phonebook['jack'] (assuming there is no collision):
is a constat time operation as illustrated below:



‘jack’ ——-> 35…2402 ——> 2
(hash value) (index into array)

‘jack’,22222

5. hash function analysis

1. Given the total number of possible key values N (taken from a large universe set U), and the size of hashtable T of m, hash function, h map U to {0,1, … m-1}

2. Pigeohold theorem: If |U|>m, then the function is many-to-one, there are at least two elements being mapped to the same value (i.e., collision).

3. Hash function design principles:

a. for any key, use all fields of the key to calculate the hash value:
e.g., do not just use the last charactor of a string …

b. use “all” information:
if you just add all characters up to calculate a string’s hash value, then ‘cat’ will collide with ‘act’.

instead: radix notation: teat a string as a base 128 numbers

‘pt’: ‘p’*128+’t’
‘tp’: ’t’*128+’p’

c. **there is no single hash function that behaves well on all possible sets of data**

Imagine that keys are from a Universe set of size **N**, and the hash table size is m.
then for any hash function, we can find a collection of **N/m** keys that are mapped to the same slot by the hash function.

(Prove by contradiction). If not, i.e., every slot has at most **N/m - 1** elements mapped to them, then in total, there are **(N/m -1)*m < N** elements being mapped. But the hash function maps every element in **N** to the slots in the hashtable. This is contracdition. Therefore assumption is wrong.

d. **Universal hashing**: introduce randomness in the hash function, done at the beginning of the hashtable creation. So that we will not be stuck with a bad hash function in mutliple runs of the program.

4. Collision resolution via chaining and performance analysis

Given collusion cannot be avoided, collision resoluion schemes have been used to decide what to do when multiple elements are hashed to the same slot in the table/array/list. We focus on the chainning method as illustrated below. The other method is so called open addressing, where a probing sequence (i.e., a sequence of hash functions) is used and when collision occurs, the next hash function is tried, and so on, until an open slot is found.

In chaining, the multiple key-value pairs hashed to the same slot are stored as a **linked list** with the address of the linked list stored in the table:

Insert(k, v): calculate hash value of k and insert (k,v) into the header of the linked list. This takes a constant time.

Search (k): calcualte hash value of k, and look for (k,*) in the linked list. This operation takes a time that is linear to the length of the linked list.

Delete (k): similar to Search() operation.

Note that a linked-list data structure is different from a list/array. In linked list, the address of second element/node is stored in the first node, and so on. So in order to access the n-th element, one has to start from the first element, go to second, and third, and so on until one finds the n-th element. The running time of such linked-list traversal is O(n).

If a hash function is chosen well, the average length of the linked lists in a hash table is **N/m**, where N is the total number of elements in the hash table, and m is the size of the hash tbale T.