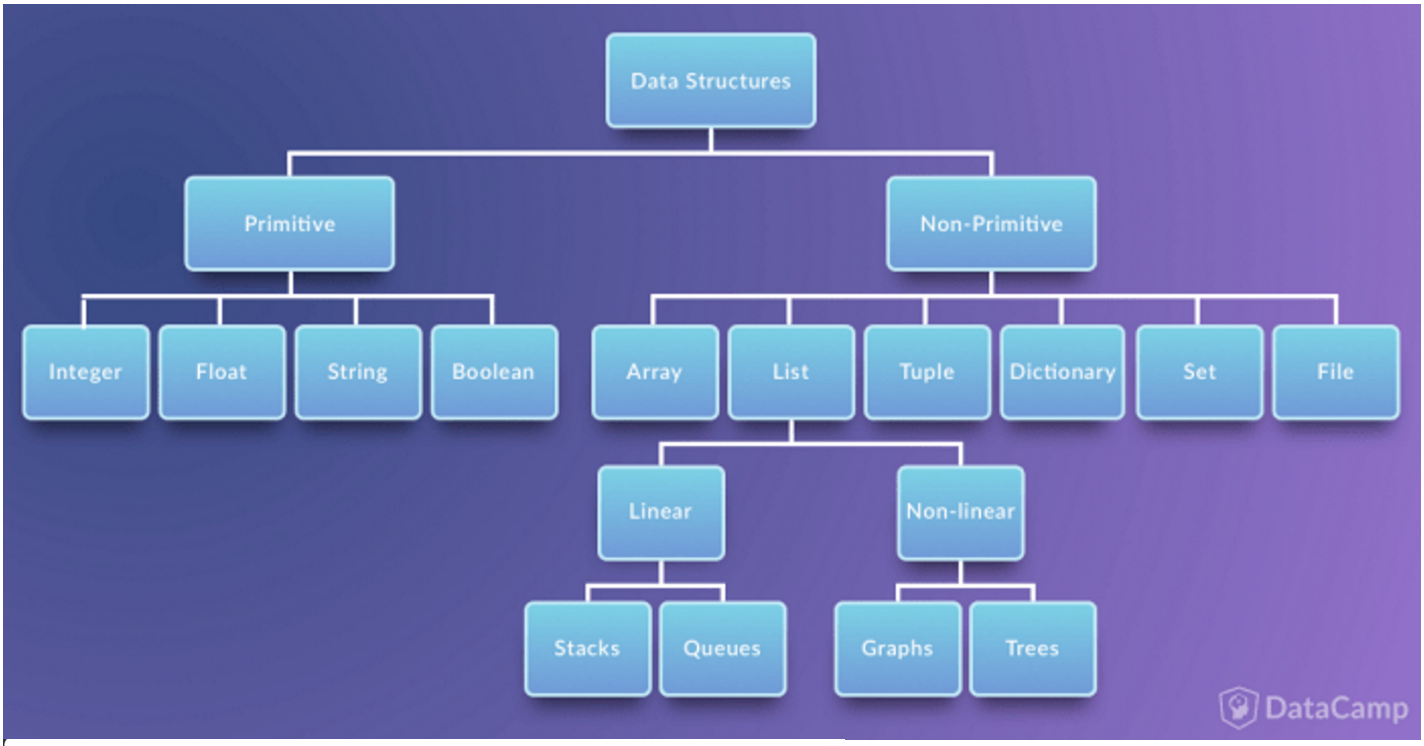
1 - Intro

jueves, 18 de abril de 2024 10:26

Data Structures



Data Structure	Python	Java	C++
Array	list	Native array ArrayList	Native array std::vector
Hash Table	dictionary	HashMap LinkedHashMap	std::map
Linked List	Not available	LinkedList	std::list

Tutorials: [Data Structures And Algorithms In Python]

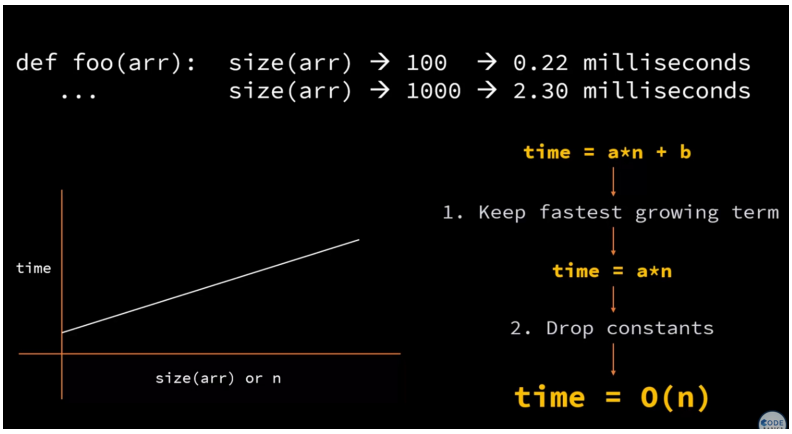
2 - Big O notation

jueves, 18 de abril de 2024 9:16

https://www.bigocheatsheet.com/

Big O notation:

Big O notation is used to perform a rough estimation of how running time or space requirements for your program grow as input size grows in worst case scenario. (tiempo de ejecución aproximado)




```
def get_squared_numbers(numbers):
    squared_numbers = []
    for n in numbers:
        square_numbers.append(n*n)
    return squared_numbers

numbers = [2,5,8,9]
get_square_numbers(numbers)
# returns [4,25,64,81]
```

O(n)

In this example, we have a function where we loop 'n' times, therefore we do 'n' operations.

The big O time complexity for this program is 0(n) 'order of n'. It grows linearly as 'n' increases.


```
def find_first_pe(prices, eps, index):
    pe = prices[index]/eps[index]
    return pe
```

O(1)

In this example we only do one operation, a constant operation. Therefore the Big O complexity is constant O(1).

In this example the execution time is constant as 'n' increases. Therefore, time complexity is O(1) 'order of 1'.

```
numbers = [3,6,2,4,3,6,8,9]

for i in range(len(numbers)):
    for j in range(i+1, len(numbers)):
        if numbers[i] == numbers[j]:
            print(numbers[i] + " is a duplicate")
            break
```

time = a* n^2 + b → O(n^2)

In this program we try to find a duplicate number in a list. We have 2 for loops where we try to find the duplicate by comparing the numbers, looping the same list twice. Time = a* n^2 + b -> O(n^2)

```
numbers = [3,6,2,4,3,6,8,9]
duplicate = None
for i in range(len(numbers)):
    for j in range(i+1, len(numbers)):
        if numbers[i] == numbers[j]:
            duplicate = numbers[i]
            break

for i in range(len(numbers)):
    if numbers[i] == duplicate:
        print(i)
```

n^2 iterations

n iterations

time = a*n^2 + b*n + c → O(n^2)

In this example we have a first block that performs n^2 iterations, and a second block that performs n iterations. In this case the time function is time = a*n^2 + b*n + c. We have to apply the principles; **First, keep only the fastest growing tearm,** which is a*n^2**, then drop the constant** 'a', so we obtain a Big O of O(n^2).

Big O is a simplistic measure of how our program time complexity is about to grow.

1. Keep fastest growing term
2. Drop constants

BigO refers to very large value of n. Hence if you have a function like,

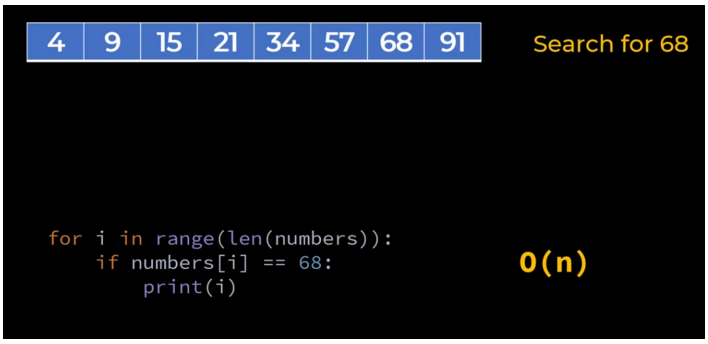
time = $5*n^2$ + 3*n + 20

When value of n is very large **b*n + c** become irrevalant

Example: n = 1000

time = $5*1000^2$ + 3*1000 + 20
time = 5000000 + 3020

Lets try to make a program more efficient by reducing big O.



We have an ordered list. Let's divide n by 2 and check if that number is > or < than 68. Based on the answer we discard half of n. We perform another iteration diving the remaining part by 2 and check the middle value again. We do that till we find 68. Last iteration will be:
Iteration k = n/2^k

Let's convert that to big O notation. Big O notation is used to measure the WORST CASE SCENARIO. The worst case scenario for this problem is reducing the list to 1 element.

1 element array/list = n/2^k iterations
N = 2^k -> Log2 (n) = log2(2^k) -> Log2 (n) = k* log2(2) -> Log2 (n) = k* 1 -> k = log2(n)
K is the number of iterations that we need. Therefore big O complexity is O(log2 n).
Log2 stands for log of base 2.

ChatGPT

Big O notation, time complexity, and space complexity are all related concepts in computer science, particularly in the analysis of algorithms.

1. **Big O Notation:** Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument approaches infinity. In the context of algorithms, it is used to describe the upper bound on the asymptotic growth rate of the running time or space usage of an algorithm in terms of the size of its input. For example, O(n^2) represents an upper bound where the algorithm's performance grows quadratically with the size of the input.
2. **Time Complexity:** Time complexity is a measure of the amount of time an algorithm takes to complete as a function of the length of its input. It is typically expressed using Big O notation to describe the worst-case, best-case, or average-case scenarios. For instance, an algorithm with a time complexity of O(n) has linear time complexity, meaning that its running time grows linearly with the size of the input.
3. **Space Complexity:** Space complexity is a measure of the amount of memory space an algorithm requires as a function of the length of its input. Similar to time complexity, it is typically expressed using Big O notation to describe the worst-case, best-case, or average-case scenarios. An algorithm with a space complexity of O(n) requires a linear amount of memory space with respect to the size of the input.

In summary, while Big O notation is a mathematical tool used to describe the limiting behavior of functions, time complexity and space complexity are specific applications of Big O notation in the analysis of algorithms, focusing on the algorithm's time and space requirements, respectively.

Big O notation is used to express time and space complexity.

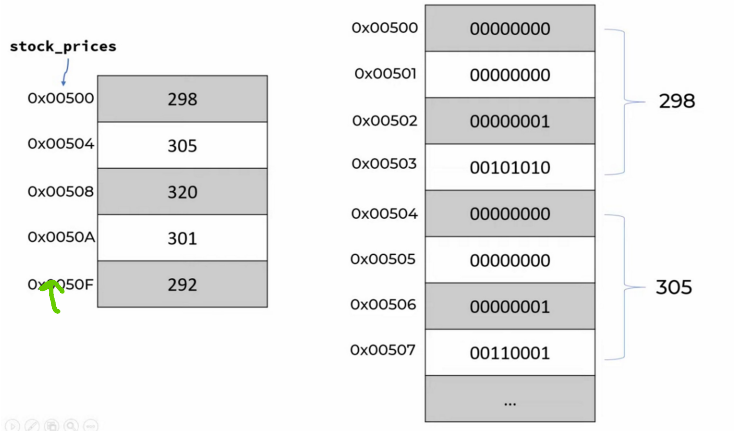
Time complexity is a measure of the amount of time an algorithm takes to complete as a function of the lenght of its input.

Space complexity is a measure of the amount of memory space an algorithm requires as a function of the lenght of its input. An algo with O(n) space complexity, requires a linear amount of memory space with respect to the size of the input.

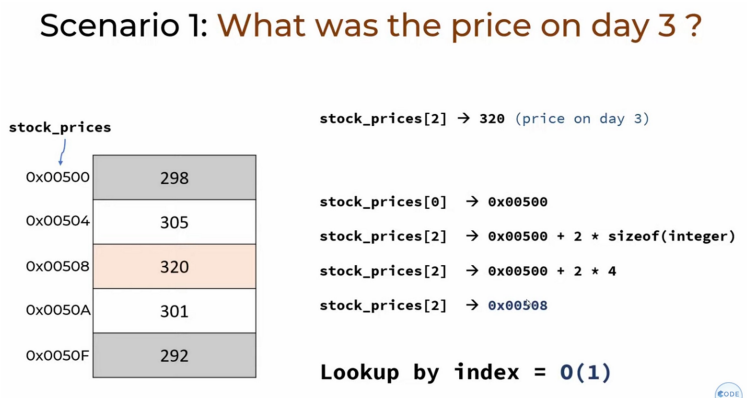
3 - Arrays/Lists

jueves, 18 de abril de 2024 11:00

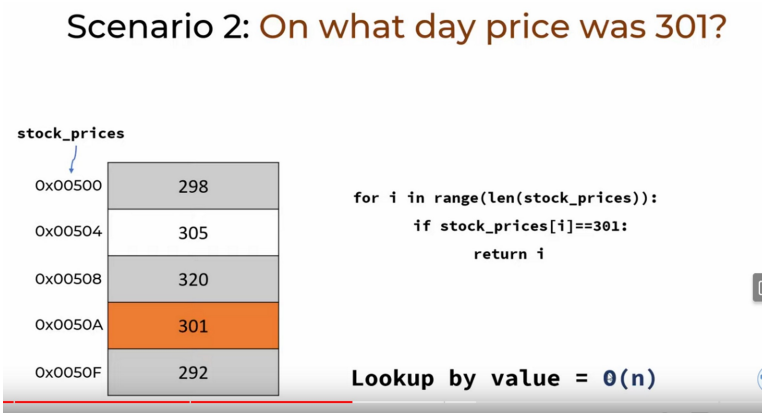
RAM storage for each item in an array. 298 in binary equals 100101010. Integers use 4 bytes to store a number. Python uses a different method for memory presentation as it uses objects and references, but this gives a simple presentation to understand the concept behind an array.



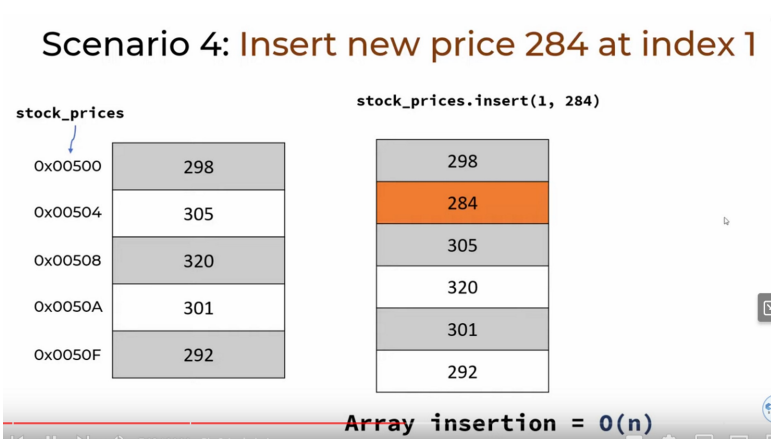
This is a memory layout.

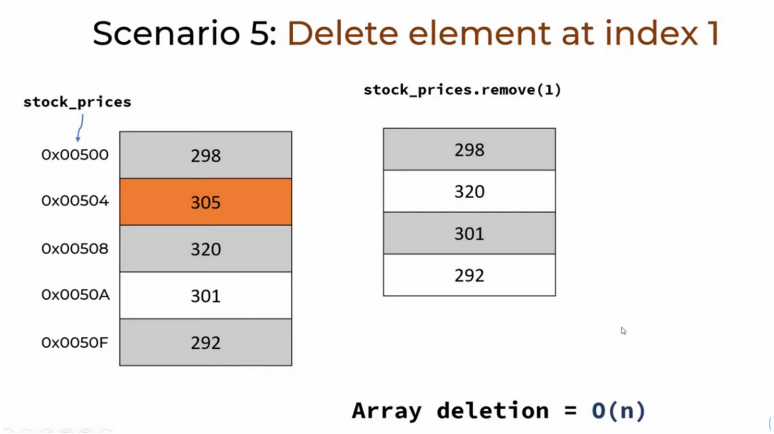


The looking up by index has a big O of O(1). A constant.



Here the insert operation (index, value), implies shifting by 1 the memory position of all elements of the array. Therefore big O is O(n).

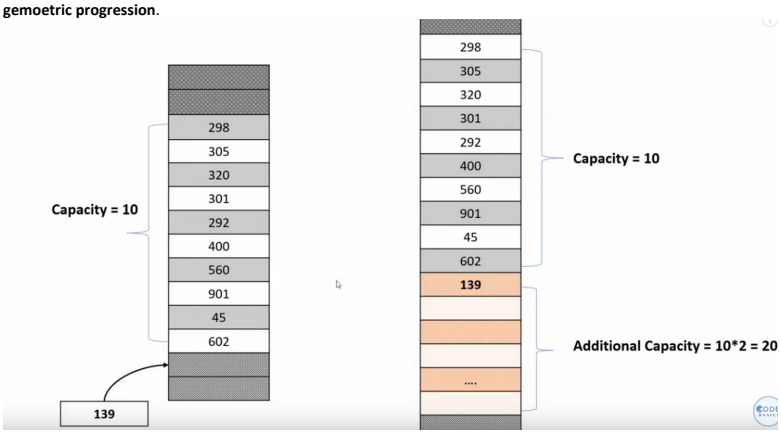




In python Lists are implemented as dynamic arrays. In other languages, like C++, we have static and dynamic arrays. Also in python we can store elements from different data types inside the array (dicts, ints, strings,..).

Static arrays/lists have a fixed size. Only the memory for given size is allocated.

Dynamic arrays/lists size can vary. For dynamic arrays, when initiallized, some space is allocated, lets say the memory for capacity = 10. When we try to add more values to it, internally it searches for a block of memory to store previous capacity plus 2 times the previous capacity (this extra space is called overhead), so now current capacity = 10 + 10*2 = 30. Therefore, we have to first copy the previously stored elements to the new memory space, and then the new value will be added. If top capacity is reached again, new memory block will be of capacity = 30 + 30*2 = 90. **This is called gemoetric progression**.



Exercise:
https://github.com/codebasics/data-structures-algorithms-python/blob/master/data_structures/2_Arrays/2_arrays_exercise.md

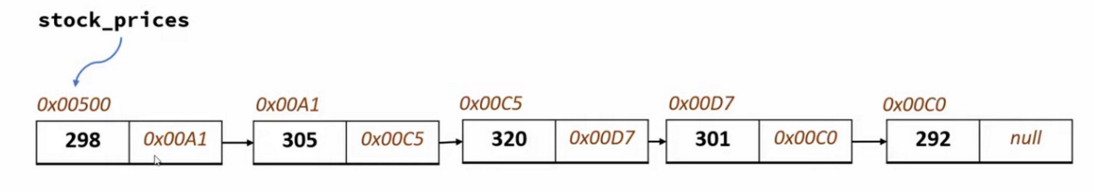
4 - Linked lists

jueves, 18 de abril de 2024 15:10

Linked lists are dynamic arrays that deal with memory operations in a more efficient way.

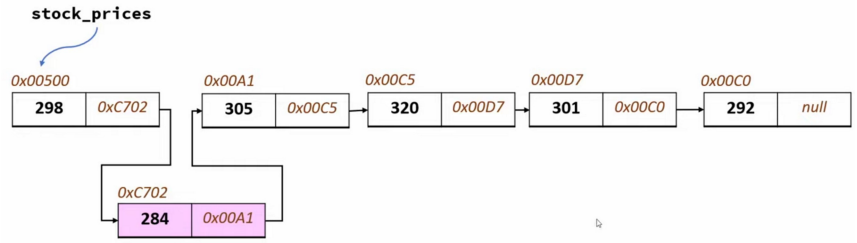
Dynamic arrays/lists size can vary. For dynamic arrays, when initiallized, some space is allocated, lets say the memory for capacity = 10. When we try to add more values to it, internally it searches for a block of memory to store previous capacity plus 2 times the previous capacity (this extra space is called overhead), so now current capacity = 10 + 10*2 = 30. Therefore, we have to first copy the previously stored elements to the new memory space, and then the new value will be added. If top capacity is reached again, new memory block will be of capacity = 30 + 30*2 = 90. **This is called gemoetric progression**. And it aims to have array elements in a CONTIUOUS MEMORY LOCATION.

Instead linked lists store elements at RANDOM MEMORY LOCATIONS, and those locations are linked by pointers.



The first element has a reference to the address of the next element. We are creating links that allow us to access the next element.

So now, when we want to insert a new element, **we only have to modify the links in our list and we dont have to copy the values from one place to another place**:



Traversal is done by following the links to the next list element.

```
Insert Element at beginning = O(1)
Delete Element at beginning = O(1)
Insert/Delete Element at the end = O(n)
```

```
Linked List Traversal = O(n)

Accessing Element By value = O(n)
```

Additionaly we do not need to pre-allocate space.

There is also the concept of **Double Linked List** which basically have a link to the next element and a link to the previous one. This makes List Traversal easier (recorrer la lista).

	Array	Linked List
Indexing	O(1)	O(n)
Insert/Delete Element At Start	O(n)	O(1)
Insert/Delete Element At End	O(1) - amortized	O(n)
Insert Element in Middle	O(n)	O(n)

The only advantadge array has is that when accessing a value by its index, we dont need to follow the links to know where element in index 5 is located in memory, therefore it is a constant operation. In linked lists we have to follow the links therefore its a O(n).

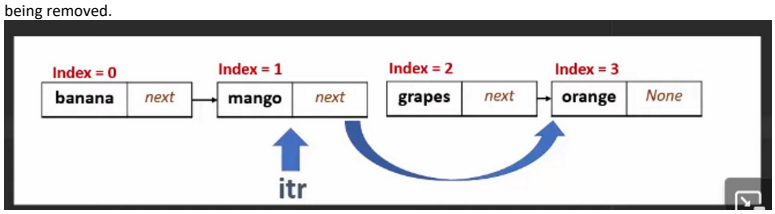
For insert and delete elements at start, for arrays it means shifting all the elements memory location, and for linked lists it only means modifying one link, therefore it is O(n) vs O(1).

Insert and delete element at end of an array, can have the huge cost of reaching the end of the allocated memory, therefore having the cost of moving and copying the data to a new memory space, so we get O(1) when that is not the case and a huge cost when it is. For linked lists we only have to follow the links towards the last element and apply the modification then, threfore the complexity is O(n).

An insert in the middle, would be the same in both cases (either shifting memory locations or following the links) but if again, we reach top capacity for an Array it would mean having the moving and copying of elements to a new memory location.

Implementation of Linked list in python:

For removing an item we only have to stop at the element before the one we want to remove, and make the link to the next element point to the one after the one being removed.




```
class Node:
      def __init__(self, data=None, next=None):
            self.data = data
            self.next = next


class LinkedList:
      def __init__(self):
            self.head = None

```

```
    def insert_at_begining(self, data):
        node = Node(data, self.head)
        self.head = node

    def print(self):
        if self.head is None:
            print("Linked list is empty")
            return

        itr = self.head
        llstr = ''
        while itr:
            llstr += str(itr.data) + '-->'
            itr = itr.next

        print(llstr)

    def insert_at_end(self, data):
        if self.head is None:
            self.head = Node(data, None)
            return

        itr = self.head
        while itr.next:
            itr = itr.next

        itr.next = Node(data, None)

    def insert_values(self, data_list):
        self.head = None
        for data in data_list:
            self.insert_at_end(data)

    def get_length(self):
        count = 0
        itr = self.head
        while itr:
            count += 1
            itr = itr.next

        return count

    def remove_at(self, index):
        # For python we do not need to cleanup the memory asociated
        # with the element removed.
        if index < 0 or index >= self.get_lenght():
            raise Exception("Invalid index")

        if index == 0:
            self.head = self.head.next

        count = 0
        itr = self.head
        while itr:
            if count == index - 1: # Stoping on previous element to modify link.
                itr.next = itr.next.next
                # Link now points to the elemen after the one being removed.
                break

            itr = itr.next
            count += 1

    def insert_at(self, index, data):
        if index < 0 or index > self.get_lenght():
            raise Exception("Invalid Index")

        if index == 0:
            self.insert_at_begining(data)
            return

        count = 0
        itr = self.head
        while itr:
            if count == index - 1: # Stoping on previous element to modify link.
                node = Node(data, itr.next) # Creating new element node.
                itr.next = node # Modifying link for the node we are stopped at.
                # Therefore, adding the new node to the linked list.
                break

            itr = itr.next
            count += 1


if __name__ == '__main__':
    ll = LinkedList()
    ll.insert_at_begining(5)
    ll.insert_at_begining(89)
    ll.insert_at_end(79)
    ll.print()
    ll.insert_values(["a", "b", "c"])
    print(f'lenght: {ll.get_lenght()}')
    ll.remove_at(2)
    ll.print()
    ll.insert_at(1, "f")
    ll.print()
```

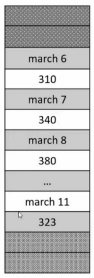
Exercises:

5 - Hash Maps

viernes, 19 de abril de 2024 14:12

Here is how arrays would handle typical data that should be stored into a hash map / dictionary (python specific implementation of a hash table).

```
In [7]: stock_prices

Out[7]: [['march 6', 310.0],
         ['march 7', 340.0],
         ['march 8', 380.0],
         ['march 9', 302.0],
         ['march 10', 297.0],
         ['march 11', 323.0]]
```



	Class	Code Sample

Python	dictionary	`prices = {`
		`  'march 6': 310,`
		`  'march 7': 430`
		`}`

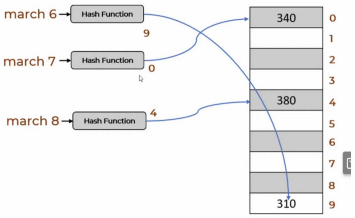
JAVA	HashMap	`HashMap<String, Integer> prices = new HashMap<String, Integer>();`
		
		`prices.put("march 6",310);`
		`prices.put("march 7",430);`

JAVA	LinkedHashMap	`LinkedHashMap<String, Integer> prices = new LinkedHashMap<String, Integer>();`
		
		`prices.put("march 6",310);`
		`prices.put("march 7",430);`

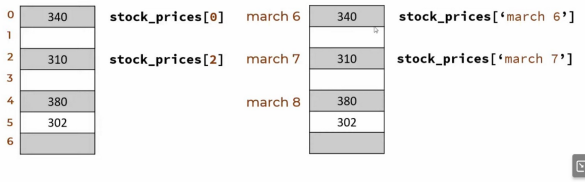
C++	std::map	`std::map<string,int> prices;`
		
		`prices['march 6']=310;`
		`prices['march 7']=430`

Instead, hash maps, convert string keys into an index that points to an array, and this is done by using a Hash function:

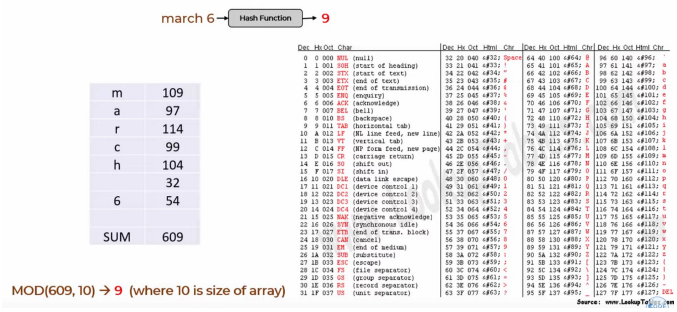
```
In [9]: stock_prices

Out[9]: {'march 6': 310.0,
         'march 7': 340.0,
         'march 8': 380.0,
         'march 9': 302.0,
         'march 10': 297.0,
         'march 11': 323.0}
```



We use string indexes instead of numerical ones like we would use in arrays:



We can use different types of hash functions to do that conversion, here is one:



The implementation of a simple hash map in python could be (Collisions are not handled):

```
# Keys are unique.
Class HashTable:
    def __init__(self):
        self.MAX =100# MAX size of list of the values array.
        self.arr = [None for i inrange(self.MAX)]# List initialization.

    # Hash function allows to get an element by key with O(1) aprox.
    # In this specific case it would be O(n) where 'n' is the number of chars in the string key assumed to be small.
    def get_hash(self,key):
        """
        Hash function to obtain the index that corresponds to the string key.
        """
        h =0
        for char in key:
            h +=ord(char)# SUM of ASCII values for each char.
        return h %self.MAX  # Mode operation to limit it to 100 positions max.

    # Overrides the set operator so we can set the value like t[key] = value.
    def __setitem__(self,key,value):
        """
        Adds key, value pair into the hashmap.
        """
        h =self.get_hash(key)# Get index using hash function.
        self.arr[h] =val # This adds the value into the list at index h.

    # Overrides the get operator so we can get a value like: t[key]
    def __getitem__(self,key):
        """
        Gets the value for given key.
        """
        h =self.get_hash(key)
        return self.arr[h]

    def __delitem__(self,key):
        """
        Deletes item given a key.
```

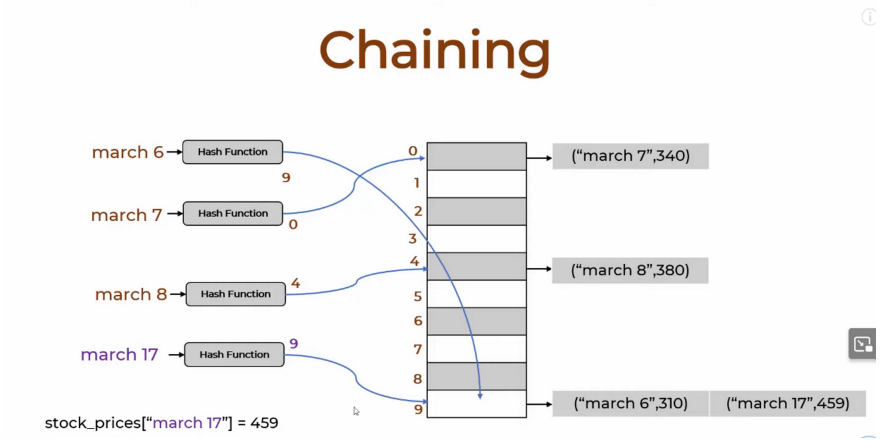
```
        """
        h =self.get_hash(key)
        self.arr[h] = None
```

Using hash maps we have the following complexities:

Look up by key is O(1) on average
Insertion/Deletion is O(1) on average

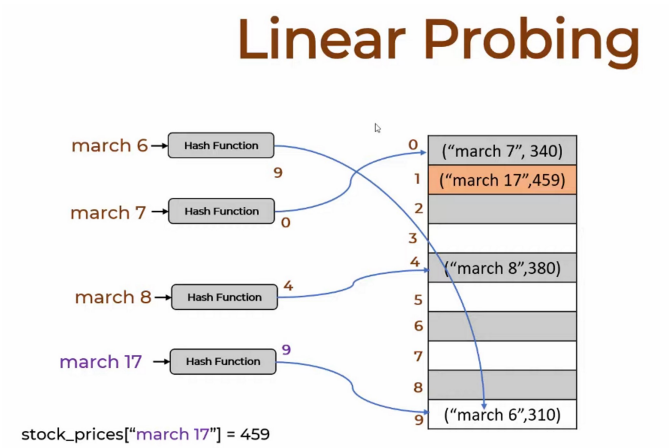
Collision Handling:

So, what happens if we have a **collision** of two different keys getting assigned the same index? Here we could implement a Chaining technique:



Chaining technique, means values we store are stored as a linked list (that contains the key and then the values for that key) instead of using a simple data type. This way different keys could coexist in the same hash. To retrieve the correct key data, we would have to iterate and check all the elements stored in that hash (or index), and check for the right key inside the values, therefore this operation would become O(n) where n is the number of lists stored in that index.

Another technique would be **Linear probing**:



Linear probing is a technique that linearly searches for an empty spot where to store the data in case of collision. If we reach the end of the array, we continue the search at the begining.

Let's add collision handling with chaining technique to previous HashMap implementation:

```
# Keys are unique.
Class HashTable:
        def__init__(self):
                self.MAX =10# MAX size of list of the values array.
                self.arr = [[]fori inrange(self.MAX)]# List of lists initialization.
                # The key value pairs will be stored as tuples inside that list of lists.
                # For example [[],[('key1', 1), ('key2', 2)], [('key3', 3)]]

        # Hash function allows to get an element by key with O(1) aprox.
        # In this specific case it would be O(n) where 'n' is the number of chars in the string key assumed to be small.
        defget_hash(self,key):
                """
        Hash function to obtain the index that corresponds to the string key.
        """
                h =0
                forchar inkey:
                        h +=ord(char)# SUM of ASCII values for each char.
                returnh %self.MAX  # Mode operation to limit it to 100 positions max.

        # Overrides the set operator so we can set the value like t[key] = value.
        def__setitem__(self,key,value):
                """
        Adds key, value pair into the hashmap.
        """
                h =self.get_hash(key)# Get index using hash function.

                # Check if item already exists in the hashmap and overrides the value if True.
                found =False
                foridx,element inenumerate(self.arr[h]):
                    iflen(element) ==2andelement[0]==key:# Checks if a tuple is stored if true checks for the key.
                        self.arr[h][idx] = (key,val)# Value override.
                        found =True
                        break
                if notfound:
                    self.arr[h].append((key,val))# This adds the value into the list at index h.

        # Overrides the get operator so we can get a value like: t[key]
        def__getitem__(self,key):
                """
```

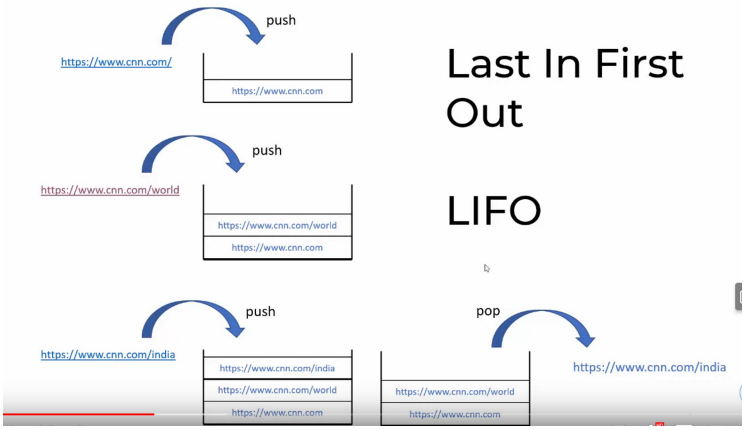
```
        Gets the value for given key.
        """
                h =self.get_hash(key)

                forelement inself.arr[h]:
                    ifelement[0] ==key:
                        returnelement[1]# Python returns None by default if no return
                        # operation is found. Correct by PEP8.

        def__delitem__(self,key):
                """
        Deletes item given a key.
        """
                h =self.get_hash(key)
                forindex,element inenumerate(self.arr[h]):
                    ifelement[0] ==key:
                        delself.arr[h][index]
```

Excersie:

6 - Stack

domingo, 21 de abril de 2024 14:50

Stack is a data structure that follows LIFO, last in first out. It can also be called LIFO queue. Think about a stack of dishes.



	Class	Code Sample

Python	list
	collections.deque
	queue.LifoQueue
		`stk = deque()`
		`stk.append(5)`
		`stk.append(9)`
		`stk.pop() # returns 89`

JAVA	Stack	`Stack<Integer> stk = new Stack<>();`
		`stk.push(5);`
		`stk.push(89);`
		`stk.pop(); // Returns 89`

JAVA	Deque	`Deque<Integer> stk = new ArrayDeque<>();`
		`stk.push(5);`
		`stk.push(89);`
		`stk.pop(); // returns 89`

C++	std::stack	`std::stack<int> stk;`
		`stk.push(5);`
		`stk.push(89);`
		`stk.pop(); // Returns 89`

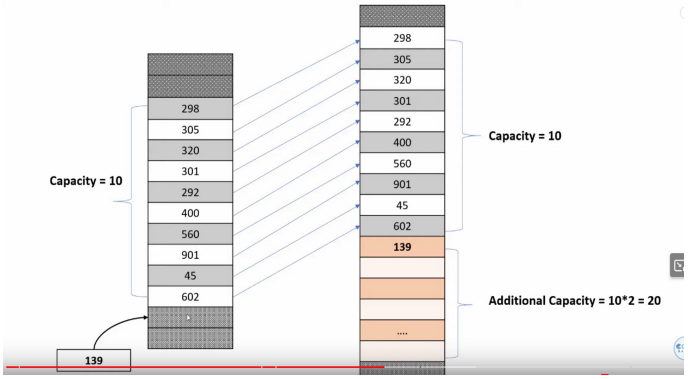
Push/Pop element: O(1)

Search element by value: O(n)

Use cases for stack

- Function calling in any programming language is managed using a stack
- Undo (Ctrl+Z) functionality in any editor uses stack to track down last set of operations

Since in python staks are dynamic arrays, memory management works as follows:



This means there is an additional cost for copying all elements into a new area when full capacity is reached.

For this reason, when we want to work with stacks in Python, using normal Lists is not the recomended approach.

The recomended approach is to use the python module, collections.deque, which are implemented using double linked list so we dont have to worry about issues faced by dynamic arrays.

Collections.deque have following methods:

```
'append',
'appendleft',
'clear',
'copy',
'count',
'extend',
'extendleft',
'index',
'insert',
'maxlen',
'pop',
'popleft',
'remove',
'reverse',
'rotate']
```

We can use this model as it is or we can implement a class Stack to use the typic methods of a stack data structure:

```
]: class Stack:
       def __init__(self):
           self.container = deque()

       def push(self,val):
           self.container.append(val)

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

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

       def is_empty(self):
           return len(self.container)==0

       def size(self):
           return len(self.container)
```

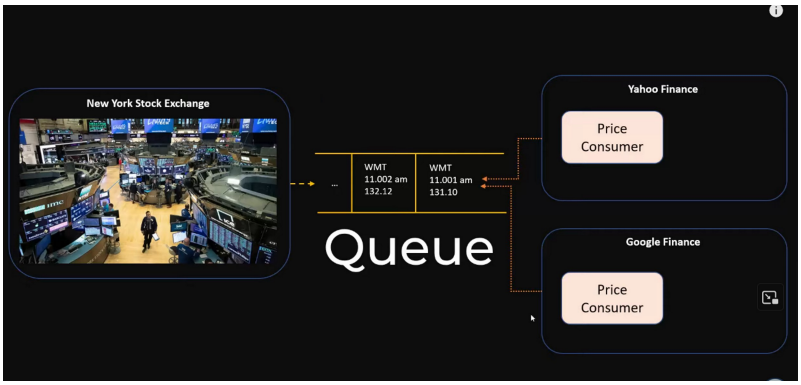
!! Push adds to the top of the stack. In the context of deque, the top of the stack is the end of the list.

Excercise:


7 - Queue

domingo, 21 de abril de 2024 19:01

Queues are buffers. Usually queues are produced by a producer and is consumed by a consumer. It is a FIFO Data Structure. First In First Out.



	Class	Code Sample

Python	list
	collections.deque
	queue.LifoQueue
		`q = deque()`
		`q.appendLeft(5)`
		`q.appendLeft(9)`
		`q.pop() # returns 5`

JAVA	LinkedList	`Queue<Integer> q = new LinkedList<>();`
		`q.add(5);`
		`q.add(89);`
		`q.remove(); // Returns 5`

C++	std::queue	`std::queue<int> q;`
		`q.push(5);`
		`q.push(89);`
		`q.pop(); // Returns 5`

Similar to stack, using list for a python implementation for a queue, is not efficient due to the memory handling of python dynamic arrays. For this reason is recomended to use collections.deque but using appendleft for insertion.

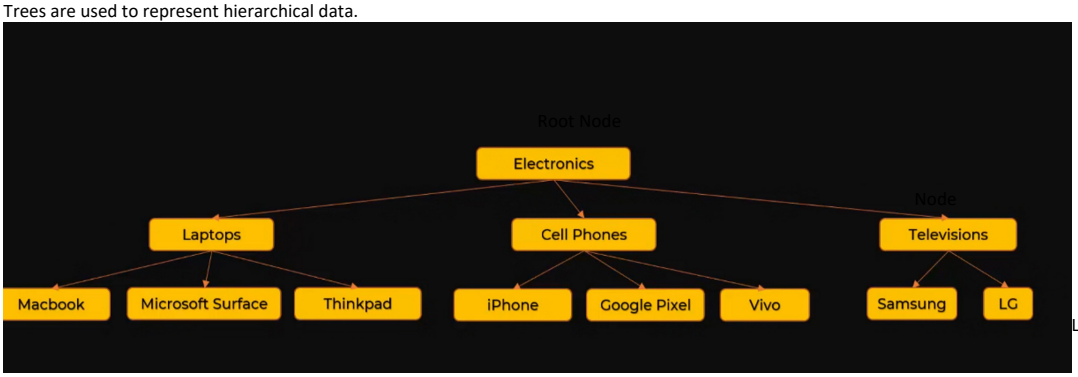
```
classQueue:
        """
    Class Queue.
    """
        def__init__(self):
                self.buffer=deque()

        defenqueue(self,val):
                """
        Adds element to buffer.
        """
                self.buffer.appendleft(val)

        defdequeue(self):
                """
        Removes element from buffer.
        """
                returnself.buffer.pop()

        defis_empty(self):
                """
        Checks if queue is empty.
        """
                returnlen(self.buffer)==0

        defsize(self):
                """
        Returns queue size.
        """
                returnlen(self.buffer)

        deffront(self):
                """
        Returns front element from the queue.
        """
                returnself.buffer[-1]
```

Excercises:

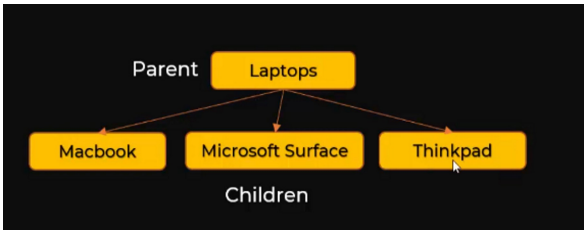
8- Trees

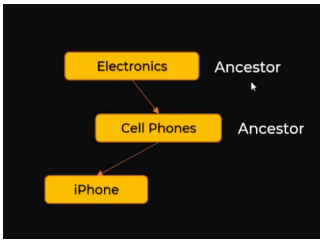
lunes, 22 de abril de 2024 12:31

Trees are used to represent hierarchical data.

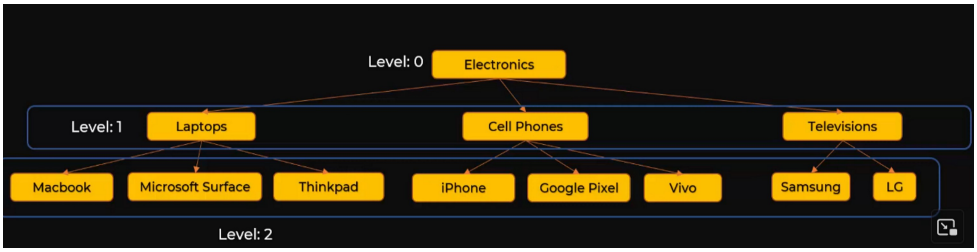


Leaf Nodes (Nodes without categories)





Cell phones and ipone are Descendants from Electronics.




```
classTreeNode:
        def__init__(self,data):
                self.data =data
                self.children = []
                self.parent =None

        # Keeping it simplem without checkig for duplicates.
        defadd_child(self,child):# Child is an instance of TreeNode class.
                """
        Adds a TreeNode as a child of self.
        """
                child.parent =self # This fills the parent property of the child TreeNode.
                # Adding a child to the self object. Self will become the parent of the child.
                self.children.append(child)

        defget_level(self):
                """
        Gets the level of the tree node.
        """
                level =0
                p =self.parent
                whilep:
                        level +=1
                        p =p.parent

                returnlevel

        defprint_tree(self,print_type,level):
                """
        Prints the tree recursively.
        """
                spaces =' '*self.get_level() *3# Adds 3 spaces for each level.
                prefix =spaces +"|__"ifself.parent else""

                ifprint_type =='both':
                        print(f'{prefix}{self.data["name"]} ({self.data["designation"]})')
                elifprint_type =='name':
                        print(f'{prefix}{self.data["name"]}')
                elifprint_type =='designation':
                        print(f'{prefix}{self.data["designation"]}')
                else:
                        raiseValueError("Print types: both, name, desingation")
                ifself.children andself.get_level() <level:# If list is not empty.
                        # We need to do a recursive print.
                        forchild inself.children:
                                child.print_tree(print_type,level)# Here is how we apply recursivety.
```

```
defbuild_tree():
        """
    Builds product tree.
    """
        ceo =TreeNode({'name':'Niupul','designation':'CEO'})

        cto =TreeNode({'name':'Chinmay','designation':'CTO'})

        inf_head =TreeNode({'name':'Vishwa',
                                        'designation':'Infrastructure Head'})
        inf_head.add_child(TreeNode({'name':'Dhaval',
                                        'designation':'Cloud Manager'}))
        inf_head.add_child(TreeNode({'name':'Abhijit',
                                    'designation':'App Manager'}))

        cto.add_child(inf_head)
        cto.add_child(TreeNode({'name':'Aamir',
                                        'designation':'Application Head'}))

        hr_head =TreeNode({'name':'Gels',
                                        'designation':'HR Head'})
        hr_head.add_child(TreeNode({'name':'Peter',
                                        'designation':'Rec Manager'}))
        hr_head.add_child(TreeNode({'name':'Waqas',
                                        'designation':'Policy Manager'}))

        ceo.add_child(cto)
        ceo.add_child(hr_head)

        returnceo

if__name__ =='__main__':
        root =build_tree()
        root.print_tree('designation',1)
```

This returns:

```
CEO
    |__CTO
    |__HR Head
```

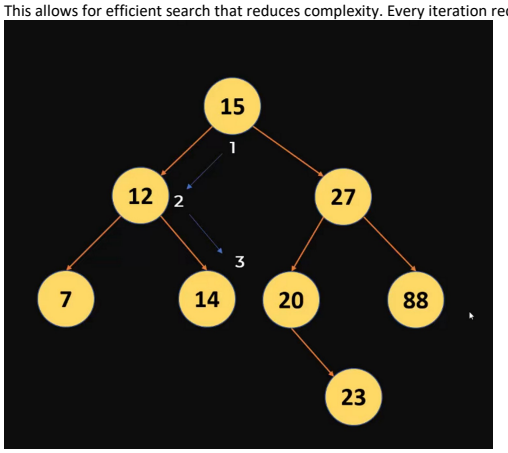
Exercises:

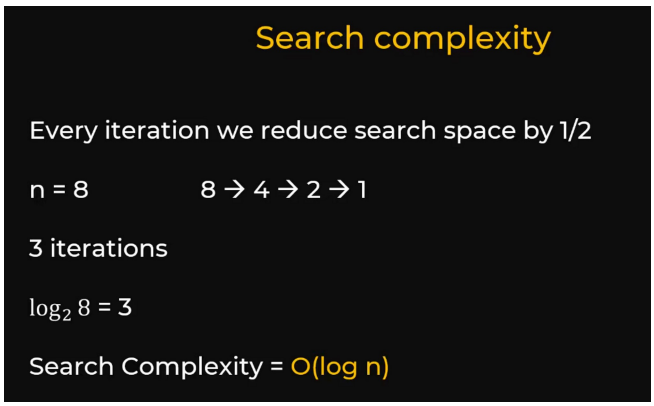
9 - Binary Trees

lunes, 22 de abril de 2024 16:16

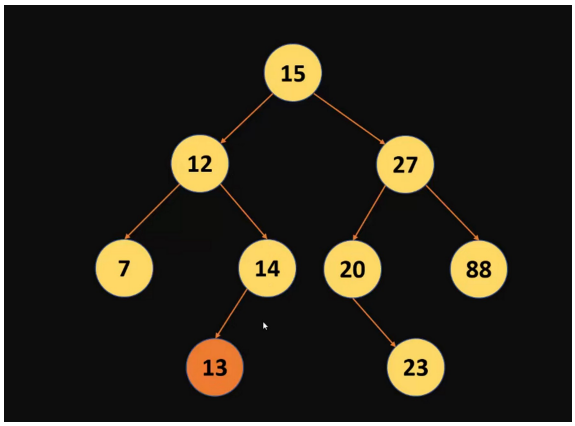
Binary Tree is a regular Tree with the constraint that each node can only have 2 child nodes top.

Binary Search Tree is a special case of Binary Tree where all elements have some kind of order. All the nodes on the left side of the parent Node must have values lower than the parent node and all nodes from the right side of the parent node must have values greater than the parent node. ELEMENTS CANNOT BE DUPLICATED. Elements are always unique. This allows for efficient search that reduces complexity. Every iteration reduces the search space by half.

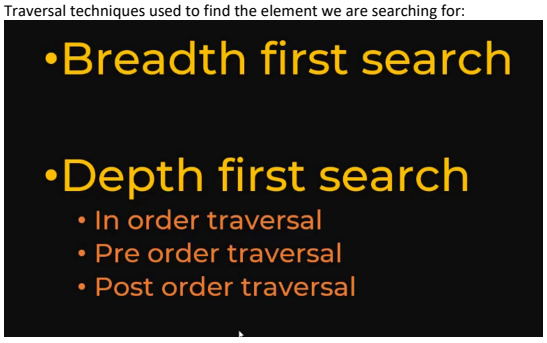




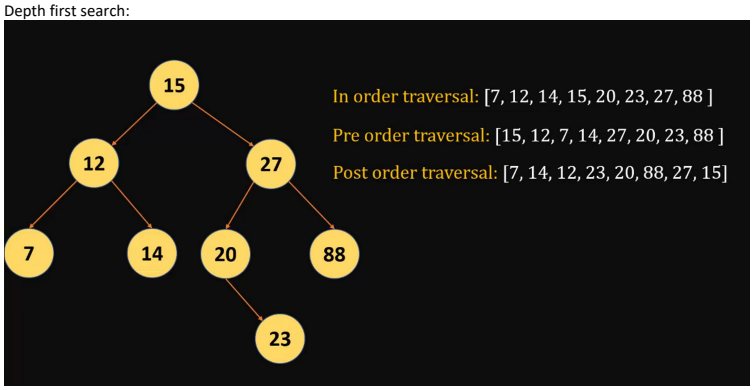
Insert has also a complexity of order of log n since we have to search for the right place to add it.



Traversal techniques used to find the element we are searching for:



Depth first search:



In order traversal: Starts from the bottom left, prioritizes visiting first the part of the tree that is more at the left side, then you go up to the intermidiate node, visits the right side of it, goes up till reaching the node on top (the base node from this example), and then you visit the tree from the right side starting from the bottom left of it, going to intermidiate node, then right. Can be used to return the elements of a list in order.

Pre order traversal: Starts from the base node, goes down visiting the left side of the tree (lower values), goes back to the intermidate node when all left tree has been searched, and continues on the right side of it, it then reaches the base node and explores the right side of it following the same strategy.

Post order traversal: It firsts visits the right side tree, then the left side tree and finally the root node. Prioritizing left.

LEFT ALWAYS GOES FIRST, TECHNIQUES DIFFER ON WHEN WE VISIT THE ROOT NODE.

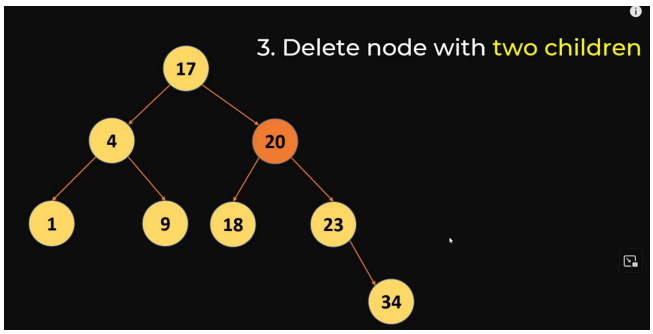
Binary trees can be used to sort the elements of a list if In order traversal is used. Duplicates are also removed, so it can also be used to implement a set().

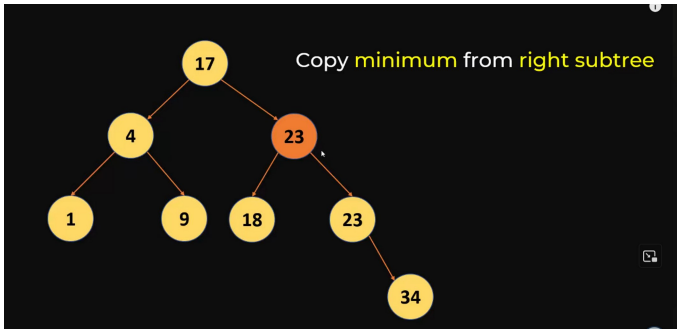
We can use strings also as values!! They are stored in alphabetical order. And this means we can do val < self.data operations.

```
classBinarySearchTreeNode:
        def__init__(self,data):
                self.data =data
                self.left =None
                self.right =None

        defadd_child(self,data):
                ifdata ==self.data:# Avoids duplicates.
                        return

                ifdata <self.data:
                        # Add data in left subtree.
                        ifself.left:# If self.left is not None call add_child recursively.
                                self.left.add_child(data)
                        else:# If free, add the new node here.
                                self.left =BinarySearchTreeNode(data)
                else:# Same for right side.
                        ifself.right:
                                self.right.add_child(data)
                        else:
                                self.right =BinarySearchTreeNode(data)

        defin_order_traversal(self):
                elements = []

                # Visits left tree adds to elements list.
                ifself.left:
                        elements +=self.left.in_order_traversal()# Recursive call.

                # Visit base node:
                elements.append(self.data)# Appends base node data.

                ifself.right:
                        elements +=self.right.in_order_traversal()

                returnelements

        defsearch(self,val):
                """
        Method searches for a value in the tree.
        """
                ifself.data ==val:
                        return True# Case value is found.

                ifval <self.data:# Value might be in left side tree.
                        ifself.left:# If there is a left tree..
                                returnself.left.search(val)# RETURNing the result of the recursive search.
                        else:# Value not in tree.
                                return False

                ifval >self.data:# Value might be in right side tree.
                        ifself.right:# If there is a right tree..
                                returnself.right.search(val)# Recursive search.
                        else:# Value not in tree.
                                return False



defbuild_tree(elements):
        root =BinarySearchTreeNode(elements[0])

        fori inrange(1,len(elements)):
                root.add_child(elements[i])

        returnroot

if__name__ =='__main__':
        numbers = [17,4,1,20,9,23,18,34]
        numbers_tree =build_tree(numbers)
        print(numbers_tree.in_order_traversal())
        print(numbers_tree.search(4))
```

This returns:

```
e_exercise.py
[1, 4, 9, 17, 18, 20, 23, 34]
True
```





Chosing the minimum value from right subtree guarantees the tree to keep following the binary tree rules, and that all the values from the right side tree will still be greater than that value.

Another approach for deletion can be:

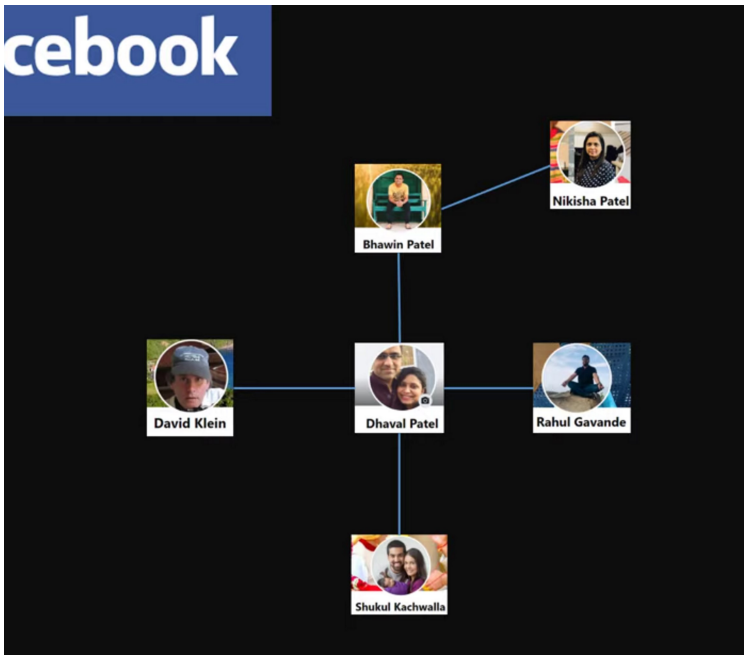
This guarantees that all elements from left subtree are less than that value.

Exercises:

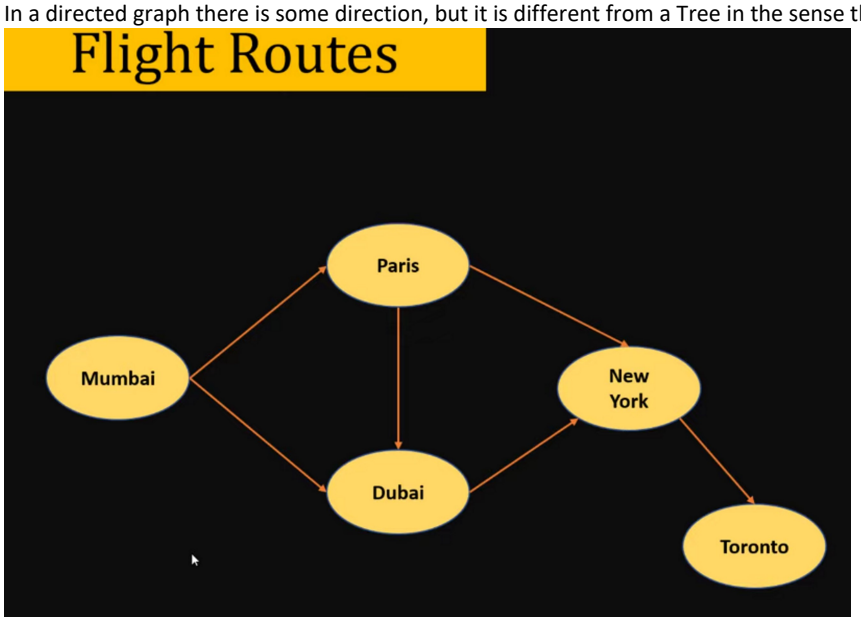
10 - Graph

lunes, 22 de abril de 2024 22:02

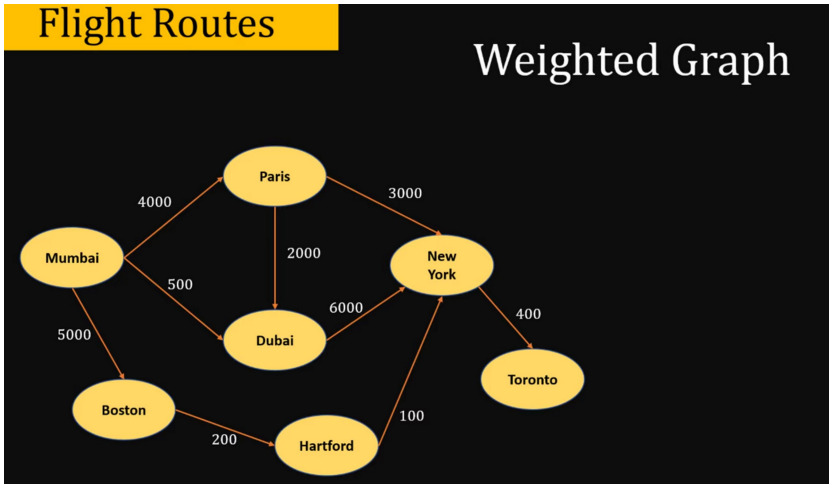
Graphs can be undirected when there is no direction between two nodes.



In a directed graph there is some direction, but it is different from a Tree in the sense that there can be multiple paths:



We can add weights to the routes than can be used to calculate the shortest path from one node to another. This is called a Weighted Graph:



Python implementation:

```
classGraph:
        def__init__(self,edges):
                self.edges =edges
                self.graph_dict = {}

                forstart,end inself.edges:
                        # Case start already in dict.
                        ifstart inself.graph_dict:
                                self.graph_dict[start].append(end)
                        # Case start not in dict.
                        else:
                                self.graph_dict[start] = [end]

                print(f"Graph dict {self.graph_dict}")

        defget_paths(self,start,end,path=None):
                """
        Returns possible paths in graph given start and end points.
        """
                ifpath is None:
                        path = [start]# This is to avoid using empty list
                                                # as default path value (not pylint complient.)
                else:
                        path =path + [start]# Path initialization.

                ifstart ==end:# Case start and end are the same.
                        return[path]# Important []: Returns/closes the path found.

                ifstart not inself.graph_dict:
                        return[]

                paths = []# List to store all possible paths.
                fornode inself.graph_dict[start]:# Search for end in all start connections.
                        ifnode not inpath:# Avoids node duplication in path.
                                # Reursive call to find a possible connexion from node to end.
                                new_paths =self.get_paths(node,end,path)
                                forp innew_paths:
                                        paths.append(p)

                returnpaths

        defget_shortest_path(self,start,end,path=None):
                """
        Gets shortest path.
        """
                ifpath is None:# This is to avoid using empty list
                                                # as default path value (not pylint complient.)
                        path = [start]
                else:
                        path =path + [start]

                ifstart ==end:
                        returnpath

                ifstart not inself.graph_dict:
```

```
                        return None

                # Case start != end.
                shortest_path =None
                fornode inself.graph_dict[start]:
                        ifnode not inpath:
                                sp =self.get_shortest_path(node,end,path)
                                ifsp:# Aims for shortest path available. sp can be None.
                                        ifshortest_path is None orlen(sp) <len(shortest_path):
                                                shortest_path =sp

                returnshortest_path


if__name__ =='__main__':
        routes = [
                ("Mumbai","Paris"),
                ("Mumbai","Dubai"),
                ("Paris","Dubai"),
                ("Paris","New York"),
                ("Dubai","New York"),
                ("New York","Toronto")
        ]
        g =Graph(routes)

        start ="Mumbai"
        end ="New York"

        print(f'Paths between{start}and{end}: ',g.get_paths(start,end))
        print(f'Paths between{start}and{end}: ',g.get_shortest_path(start,end))
```

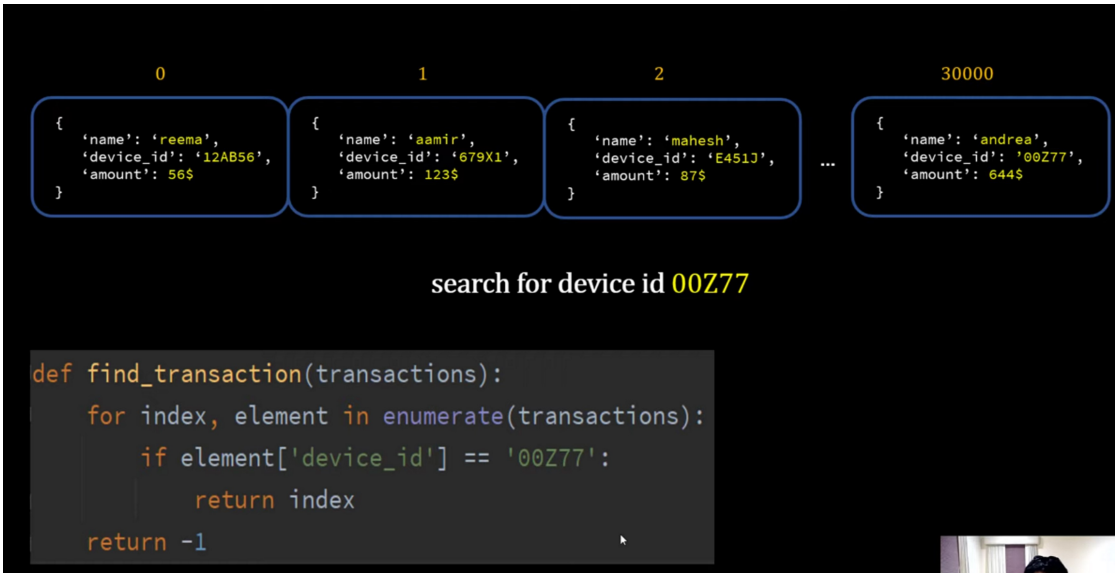
Output will be:

Graph dict {'Mumbai': ['Paris', 'Dubai'], 'Paris': ['Dubai', 'New York'], 'Dubai': ['New York'], 'New York': ['Toronto']}
Paths between Mumbai and New York: [['Mumbai', 'Paris', 'Dubai', 'New York'], ['Mumbai', 'Paris', 'New York'], ['Mumbai', 'Dubai', 'New York']]
Paths between Mumbai and New York: ['Mumbai', 'Paris', 'New York']

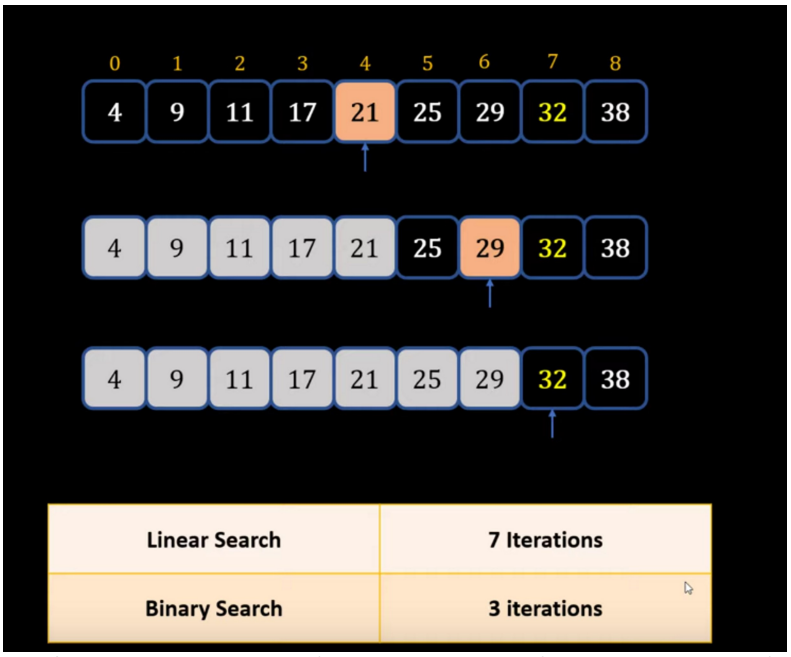
11 - Binary Search Algorithm

lunes, 22 de abril de 2024 23:20

This is a linear search. Which has a complexity of O(n).



In binary search, we have the list sorted and in each iteration for the search we discard half of it:



Linear Search	O(n)
Binary Search	Log(n)

K = n\2^k --> 1 iteration = n/2 --> iteration 2 = n/2^2 , iteration 3 = n/2^3 etc.
Therefore 1 = n/2^k -> n = 2^k - > log(n) = log (2^k) --> k = log n --> O(log n)

The types of problems where we repeat the same steps with the same exit operations, should be considered a Recursive problem and can be implemented like so.

```
"""
Implementation of binary search.
"""
importtime
fromutil importtime_it


@time_it# decorator
deflinear_search(number_list,number_to_find):
```

```
        """
    Performs linear search.
    """
        forindex,element inenumerate(number_list):
                ifelement ==number_to_find:
                        returnindex
        return-1

@time_it
defbinary_search(numbers_list,number_to_find):
        """
    Performs binary search. Returns index of found number or -1 otherwise.
    """
        left_index =0
        right_index =len(numbers_list) -1
        mid_index =0

        whileleft_index <=right_index:

                mid_index = (left_index +right_index)  //2# // ensures result will be the
intiger

                   # portion of the division 2.5 = 2.
                mid_number =numbers_list[mid_index]

                # Case found.
                ifmid_number ==number_to_find:
                        returnmid_index

                # Case mid_number is lower than number_to_find.
                ifmid_number <number_to_find:
                        left_index =mid_index +1# + 1 ensures previous mid index is not
checked again.
                # Case mid_number is greater than the number_to_find.
                else:
                        right_index =mid_index -1# - 1 ensures previous mid index is not
checked again.

        return-1# Case number not in list.


defbinary_search_recursive(numbers_list,number_to_find,left_index,right_index):
        """
    Performs binary search using a recursive algorithm.
    """
        ifright_index <left_index:
                return-1

        mid_index = (left_index +right_index)  //2# // ensures result will be the intiger

               # portion of the division 2.5 = 2.
        mid_number =numbers_list[mid_index]

        # Case found.
        ifmid_number ==number_to_find:
                returnmid_index

        # Case mid_number is lower than number_to_find.
        ifmid_number <number_to_find:
                left_index =mid_index +1# + 1 ensures previous mid index is not checked
again.
        # Case mid_number is greater than the number_to_find.
        else:
                right_index =mid_index -1# - 1 ensures previous mid index is not checked
again.

        # Recursive function. This avoids the while loop.
        returnbinary_search_recursive(numbers_list,number_to_find,left_index,right_index)


if__name__ =='__main__':
        n_list =list(range(0,1000001))
```

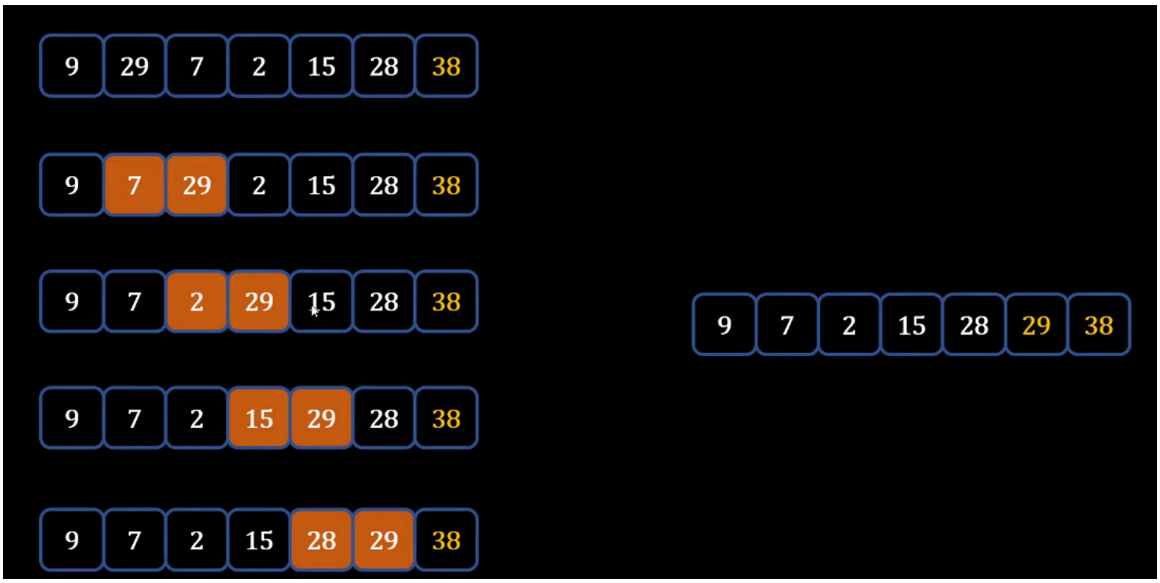
```
        print(f"Number found at index {linear_search(n_list, 1000000)} using linear
search.")
        print(f"Number found at index {binary_search(n_list, 1000000)} using binary
search.")

        start =time.time()
        recursive_search =binary_search_recursive(n_list,

    67,

    0,

    len(n_list) -1)
        end =time.time()
        print(f"Number found at index{recursive_search}using binary search recursive.")
        print(f"Binary search recursive took {round((end-start) * 1000, 2)}
miliseconds.")
```

Exercises: https://github.com/codebasics/data-structures-algorithms-python/blob/master/algorithms/1_BinarySearch/binary_search_exercise.md

12 - Bubble Sort

martes, 23 de abril de 2024 15:10

In each iteration of bubble sort we take 2 consecutive elements and compare them. If the first element is greater than the second, we change the position, we swap them.



Time Complexity	$0(n^2)$
Space Complexity	$0(1)$

Time complexity is order of n^2 because we are running 2 for loops.

Space complexity is constant because we are not using any additional space.

```
"""
Bubble sort exercises.
"""

defbubble_sort(elements):
        """
    Performs bubble sort.
    """
        size =len(elements)

        fori inrange(size -1):# We want to do this proces n-1 times (we evaluate pairs)
                                                                # as the list has to be checked n-1 times for it to be all
sorted.
                swapped =False
                forj inrange(size -1-i):# Done n-1 times - i (to avoid checking already checked
                                                                                # elements at the end). Elements at the end are
the ones
                                                                                # that will for sure be in the correct order
already.
                        ifelements[j] >elements[j+1]:
                                tmp =elements[j]
                                elements[j] =elements[j+1]
                                elements[j+1] =tmp
                                swapped =True
                if notswapped:# Breaks the outter loop if all items are already sorted.
                        break



if__name__ =='__main__':
        numbers = [5,9,2,1,67,34,88,34]

        bubble_sort(numbers)
        print(numbers)
```

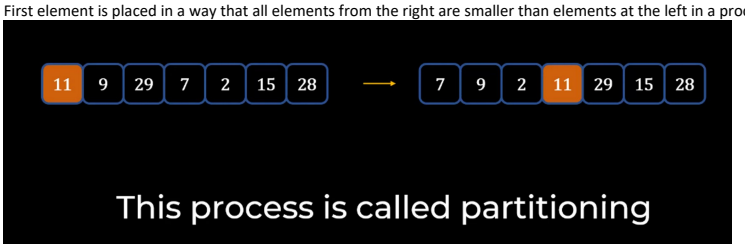
Exercise: https://github.com/codebasics/data-structures-algorithms-python/blob/master/algorithms/2_BubbleSort/bubble_sort_exercise.md

13 - Quick Sort

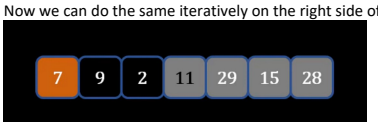
martes, 23 de abril de 2024 17:18

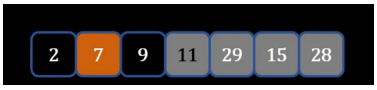
Quick sort is a divide and conquer problem:

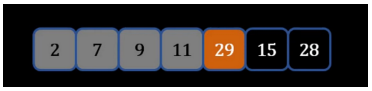
First element is placed in a way that all elements from the right are smaller than elements at the left in a process called partitioning.

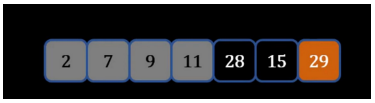


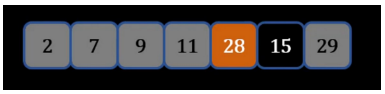
Now we can do the same iteratively on the right side of it, and at the left side of it:

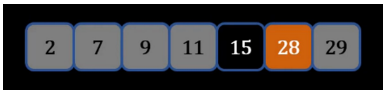




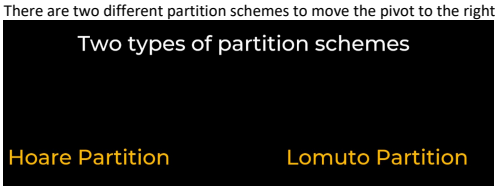




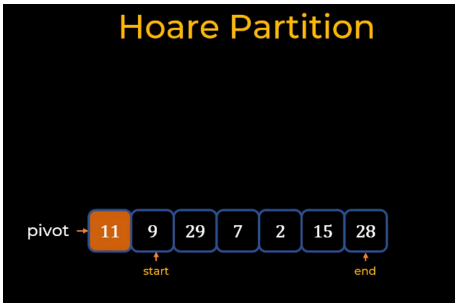




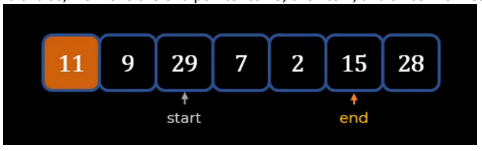
There are two different partition schemes to move the pivot to the right place.

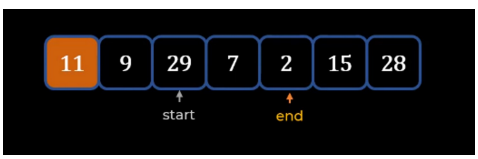


Hoare Partition:

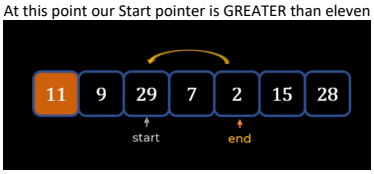


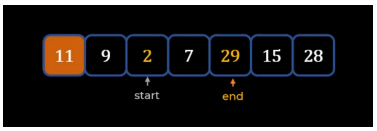
We place ourself at the start and compare that element with our pivot and check if it is GREATER than it. If 9 < 11, we ignore 9 and go to the next element. Now 29 is greater than 11, so we STOP here and check the end pointer which is 28. Now we check if 28 is LESS than 11, if it is false, we move the end pointer to 15, then to 2, and since 2 is LESS than 11, we STOP.



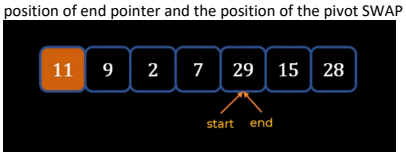


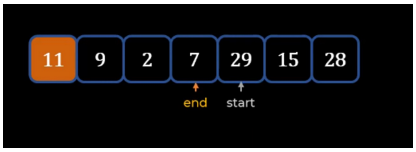
At this point our Start pointer is GREATER than eleven and our end pointer is LESS than 11. So we swap them.

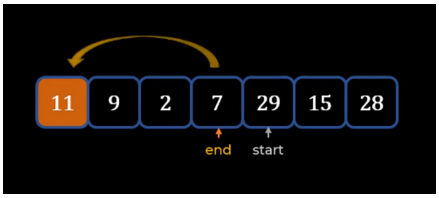


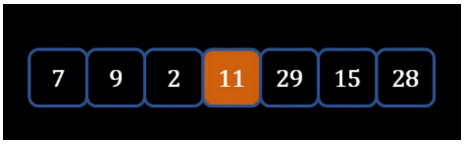


We now can move the START pointer again and keep repeating the same process. Till the condition of having all elements at the left of the pivot LOWER than it and the ones at the right GREATER than it. We STOP when end pointer CROSSES the start pointer. At this point the position of end pointer and the position of the pivot SWAP and this is how we place it in the middle.



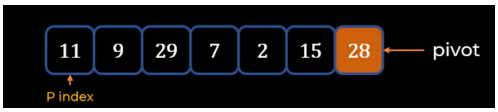




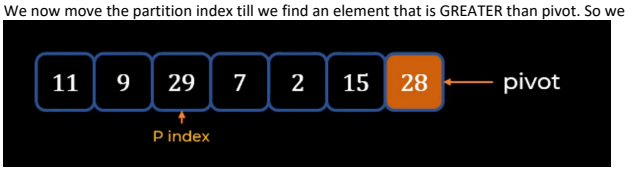


Lomuto Partition:

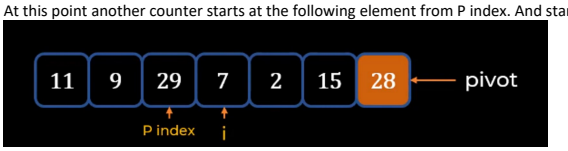
With this strategy, we decide our END element to be our PIVOT (first or middle values can also be chosen as pivot but this is the popular strategy). The start element will be the PARTITION INDEX.



We now move the partition index till we find an element that is GREATER than pivot. So we stop at 29.

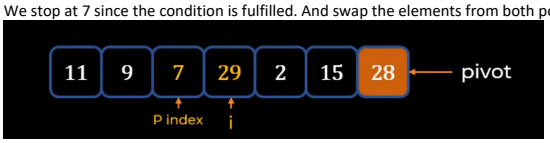


At this point another counter starts at the following element from P index. And start looking for an element that is LESS than pivot.

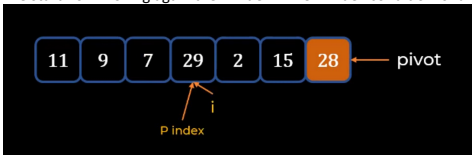


P index: value > pivot
i : value < pivot

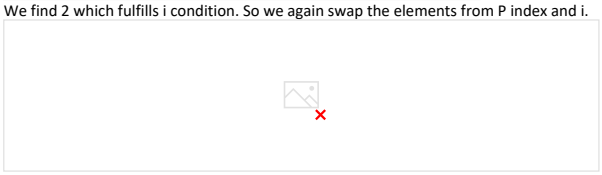
We stop at 7 since the condition is fulfilled. And swap the elements from both pointers.



We start now moving again the P index. In 29 P index condition is fulfilled. And now move i.



We find 2 which fulfills i condition. So we again swap the elements from P index and i.



We swap 15 and 29.

We reach pivot with i and switch it with p index.

Here 28 is SORTED. It is at the right position. And now all elements from its left side are LOWER than 28 and the ones at the right side are greater than 28.

Quick sort complexity:

If we partition in "equal" portions, we are reducing the search space by 2. Every time the search space is reduced by 2.

This case happens when the list is already sorted, which will create very imbalanced partitions, either we will have only a right partition or only a left partition. So this does not effectively reduce the search space. So for each element we will perform n iterations, hence a complexity of O(n^2).

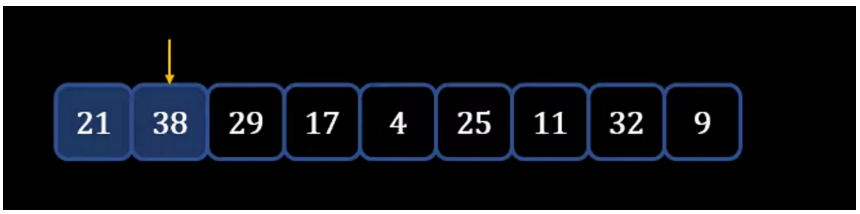
Exercise: https://github.com/codebasics/data-structures-algorithms-python/blob/master/algorithms/3_QuickSort/quick_sort_exercise.md

14 - Insertion Sort

martes, 23 de abril de 2024 21:18

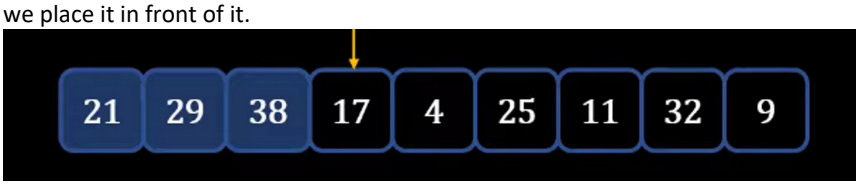
Insertion sort is NOT an efficient algorithm for long lists, but it is efficient for small ones.

In insertion sort, we start with a pointer to the second element (as len 1 array elements are always sorted), compare it to the first element and move it to the left if it is less than it or keep it in the right if it is greater than it.



We now move the pointer to the 3rd element and compare it with the sorted array (the left part of the array). We first compare it with 38 which is bigger than 29 and then we check 21 which is smaller than 29. Which means we have to insert 29 between 21 and 38.

Now we are at 17, we compare it with 38 wich is bigger, we check 29 which is also bigger, 21 next which is also bigger so we place it in front of it.



We keep doing this until the array is sorted.

Worst-case performance	$O(n^2)$ comparisons and swaps
Best-case performance	$O(n)$ comparisons, $O(1)$ swaps
Average performance	$O(n^2)$ comparisons and swaps
Worst-case space complexity	$O(n)$ total, $O(1)$ auxiliary

O(1) holds the anchor element.

Exercises:

https://github.com/codebasics/data-structures-algorithms-python/blob/master/algorithms/4_InsertionSort/insertion_sort_exercise.md

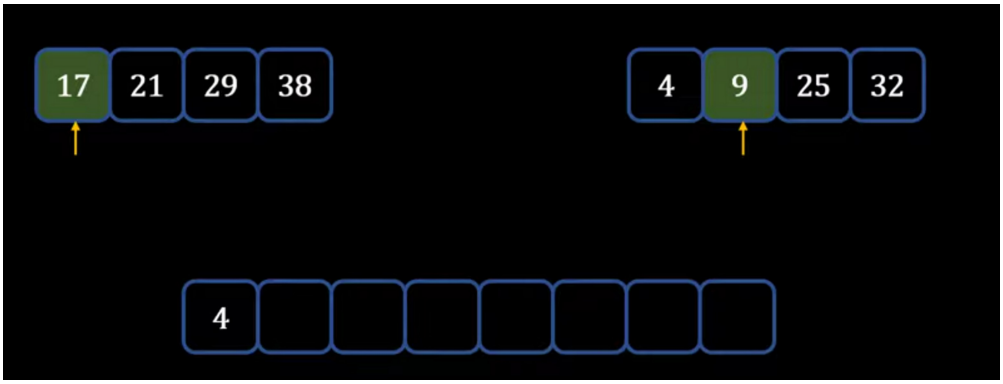
15 - Merge Sort

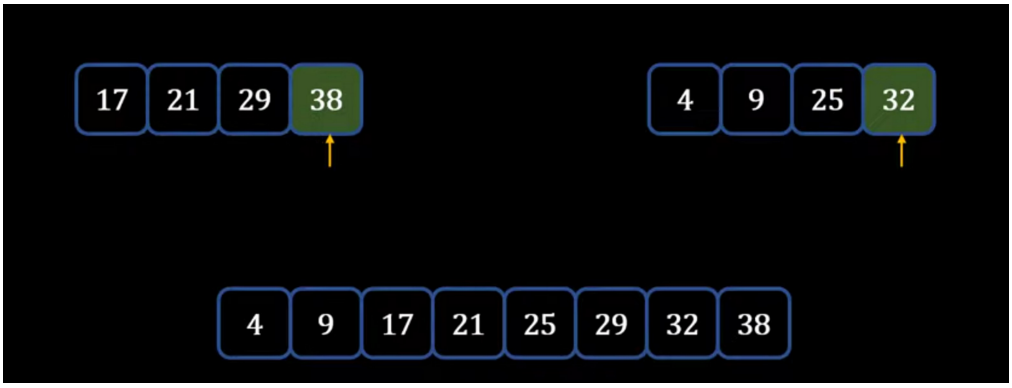
martes, 23 de abril de 2024 22:03

Merge Sort is a technique used to sort an array using a divide and conquer approach with recursion. It is more efficient that bubble sort which takes O(n^2) and quick sort (that in its worst case takes also O(n^2)).
It is an important algorithm which is used by standard python sort which applies Timsort, a combination of merge sort and insertion sort.

The concept behind is can be simply exemplified with this following problem.

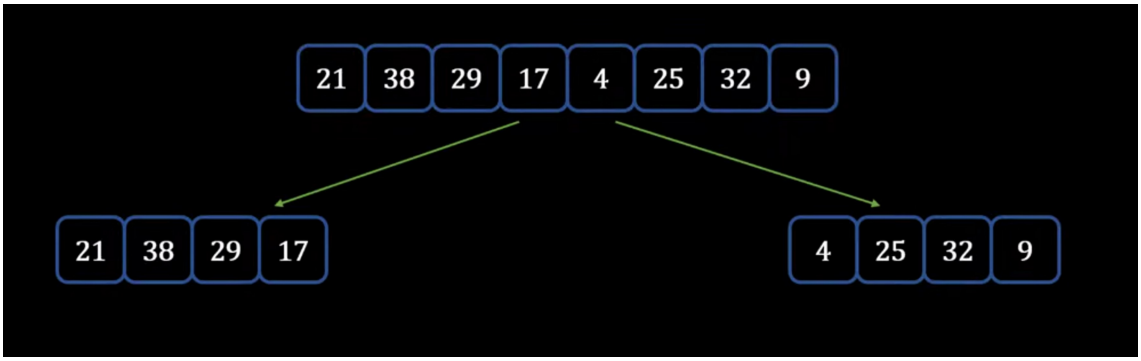
Imagine we have two sorted arrays and we want to merge them into one sorted array. We compare the first two elements and insert into the new array the smaller one, we move the pointer to the next elemenet and do the same check, this is done in a continuous way untill all elements have been placed into the new array.



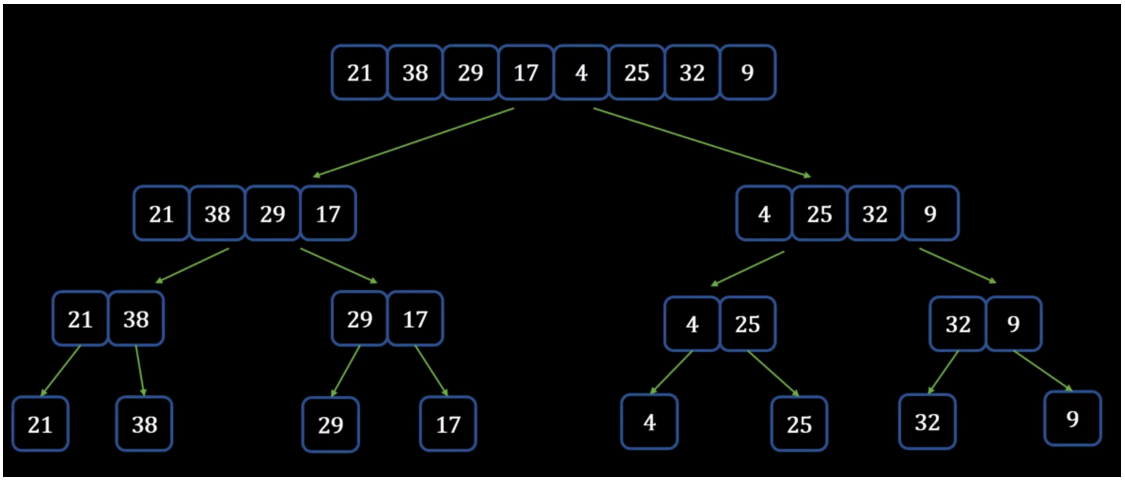


Now lets apply the same principle when we have a single unsorted array:

First we split the array in two.

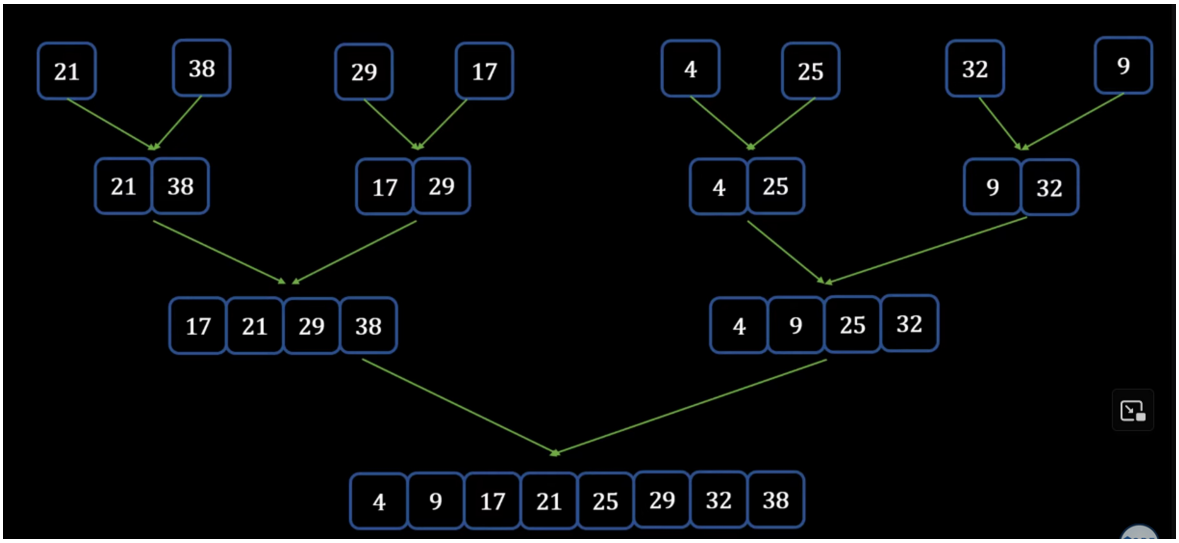


Since the two divisions are not sorted, we keep dividing them until we get sorted divisions.



Since arrays of a single element are already sorted, now we can use the technique we initially learned to merge them.

At this step we have 4 sorted arrays, so we keep going with the same aproach till we get a single sorted array.



The divide step takes constant time O(1), the conquer step where we recursively sort two subarrays of approximately n/2 elements each, takes some amount of time but we'll account for that time when we consider the subproblems. The combine step merges a total of n elements, taking O(n) time. The combination of merging and sorting is O(nlogn) total.

Implementation:

```
"""
Merge sort exercises.
"""

defmerge_two_sorted_lists(a,b,arr):
        """
    Merges two already sorted lists.
    If we avoid using an auxiliar list and just modify the arr we
    reduce space complexity.
```

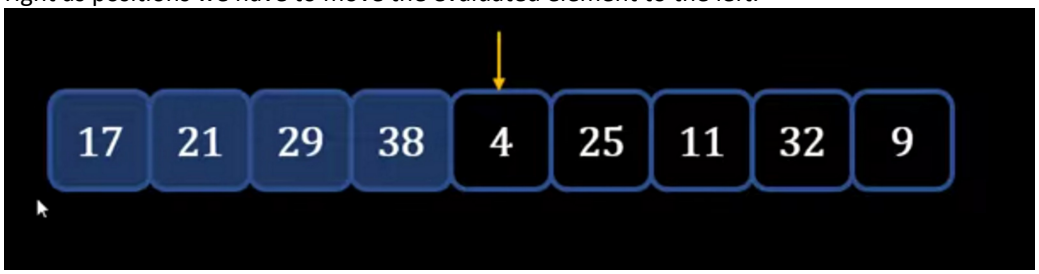
```
    """
        i =j =k =0

        whilei <len(a)andj <len(b):
                ifa[i] <b[j]:
                        arr[k] =a[i]
                        i +=1
                else:# a[i] >= b[j]:
                        arr[k] =b[j]
                        j +=1
                k +=1

        # Takes into account different len arrays:
        ifi <len(a):
                forelement ina[i:]:
                        arr[k] =element
                        k +=1
        elifj <len(b):
                forelement inb[j:]:
                        arr[k] =element
                        k +=1

defmerge_sort(arr):
        """
    Implements merge sort algorithm that sorts given array.
    """
        iflen(arr) <=1:# Stops the recursion once single element arrays have formed.

                return

        mid =len(arr) //2
        left =arr[:mid]
        right =arr[mid:]

        merge_sort(left)# Keeps dividng the array.
        merge_sort(right)# Keeps divindg the array.

        # This return is only reached once the recursion stops at single
        # element arrays, and starts going backwards.
        merge_two_sorted_lists(left,right,arr)


if__name__ =='__main__':
        f = [10,3,15,7,8,23,97,29,95]
        merge_sort(f)
        print(f)
```

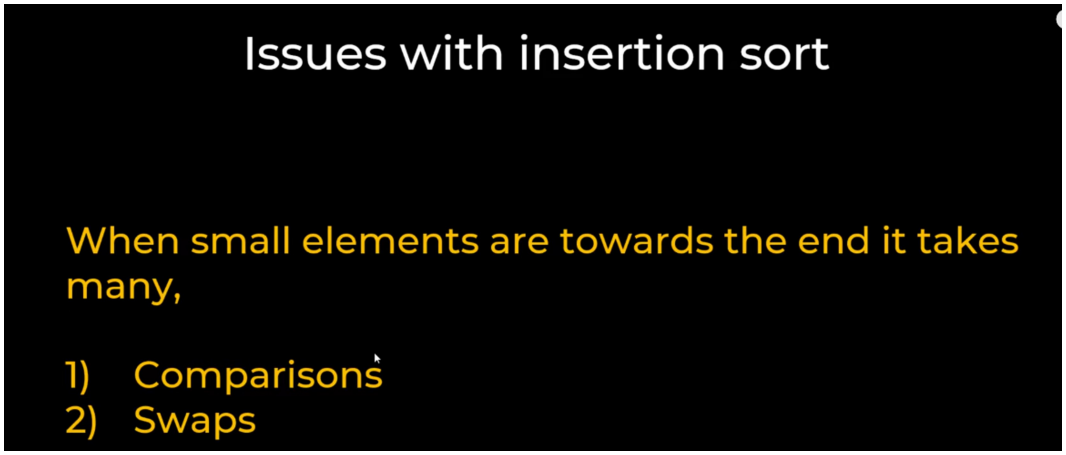
Exercises: https://github.com/codebasics/data-structures-algorithms-python/blob/master/algorithms/5_MergeSort/merge_sort_exercise.md

16 - Shell Sort

miércoles, 24 de abril de 2024 11:58

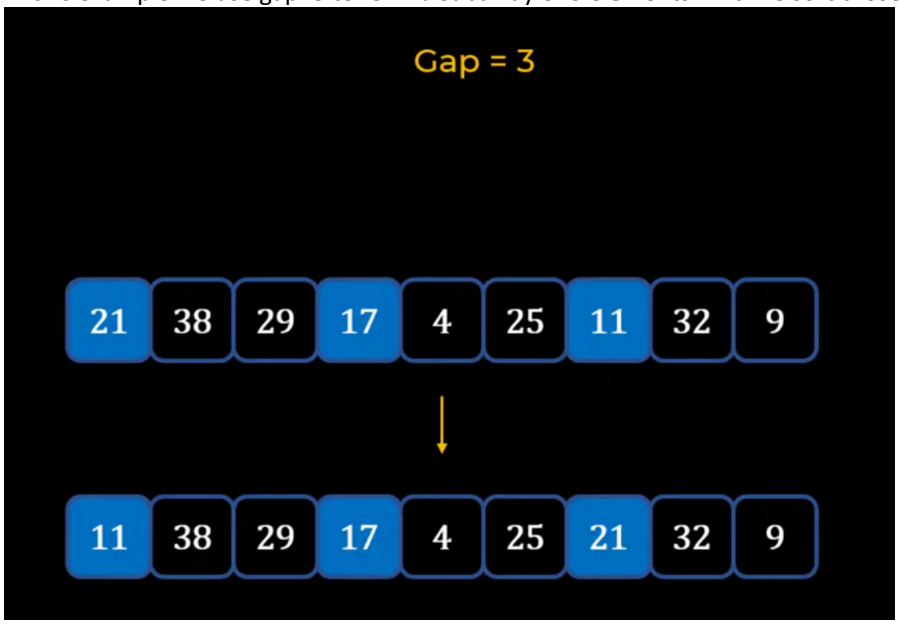
Shell sort is an optimization over insertion sort. The problem with insertion sort is that we have to do a lot of comparison. Since we compare each element on the left of the pointer to check where to place the number and perform as many swaps to the right as positions we have to move the evaluated element to the left.



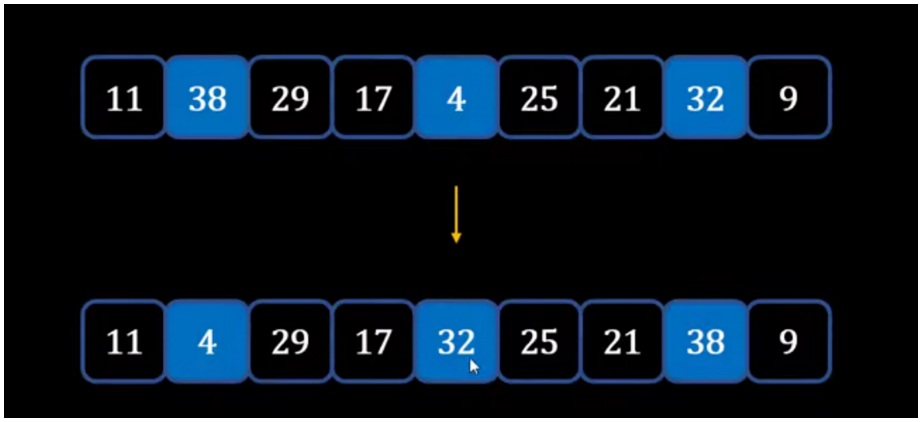


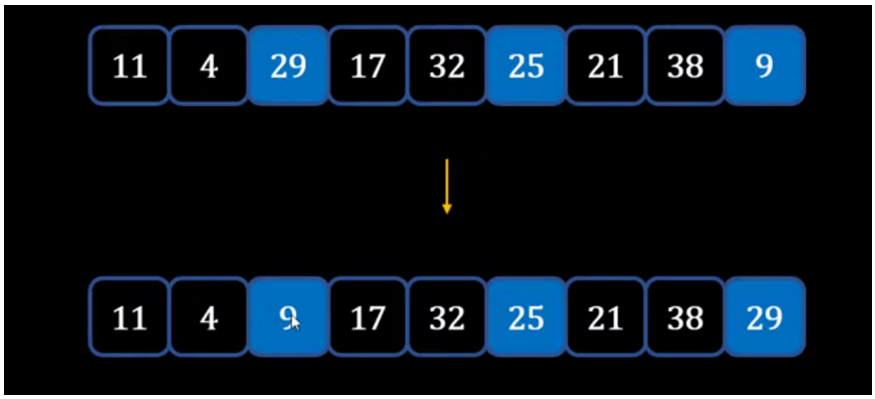
Shell sort tries to solve this problem. In shell sort we try to move the heavier element to the right side which will reduce the amount of comparisons and swaps. To do it shell sort uses the concept of a gap which are used to kinda form subarrays.

In this example we use gap=3 to form a subarray of 3 elements. And we sort those elements.

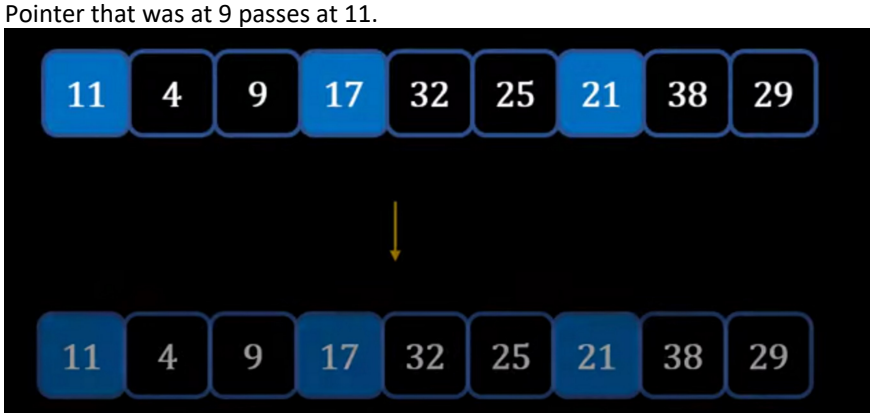


Now we move the pointers to the right to form a new subarray of three elements and repeat the process.





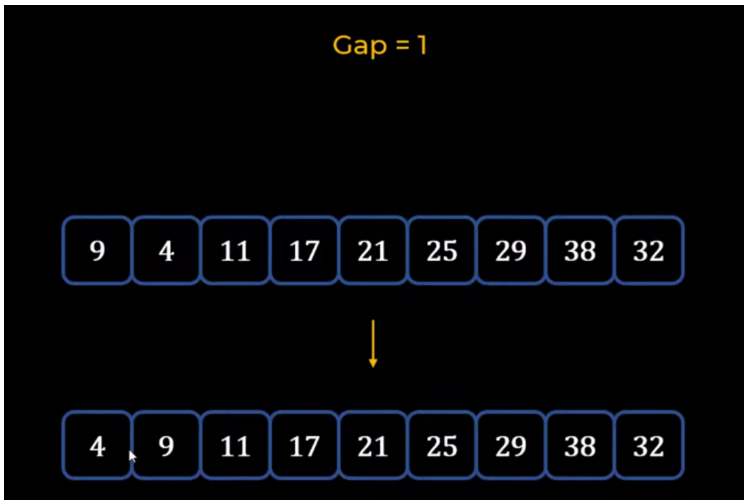
Pointer that was at 9 passes at 11.



We keep doing it till the anchor reachs the end of the array (the anchor is the first item selected depending on the gap, so for a gap=3, that would be arr[3]):

Keep in mind that resulting array is still not properly sorted, but most low values are at the left and the greater ones on the right.
Now we can apply insertion sort in a more efficient way. We have reduced the amount of swaps and comparisons.
Now we reduce the gap and repeat the process till we reach Gap=1.

Etc.



Now at gap=1 there will be only 1 swap.

Implementation:

```
"""
Shell sort exercise.
"""

defshell_sort(arr):
        """
    Shell sort implementation.
    """
        size =len(arr)
        gap =size //2

        whilegap >0:
                fori inrange(gap,size):# Loop the whole array from gap onwards.
                                                                        # Elements b4 the gap are
already checked inisde the while.
                        anchor =arr[i]
                        j =i

                        whilej >=gap andarr[j -gap] >anchor:# i.e.: arr[0] > arr[3] ?
                                arr[j] =arr[j -gap]# If true we copy in gap position

         # arr[3] the value of arr[0].
                                j -=gap
                        arr[j] =anchor # We copy the anchor initially in arr[3] to arr[0]. This
```

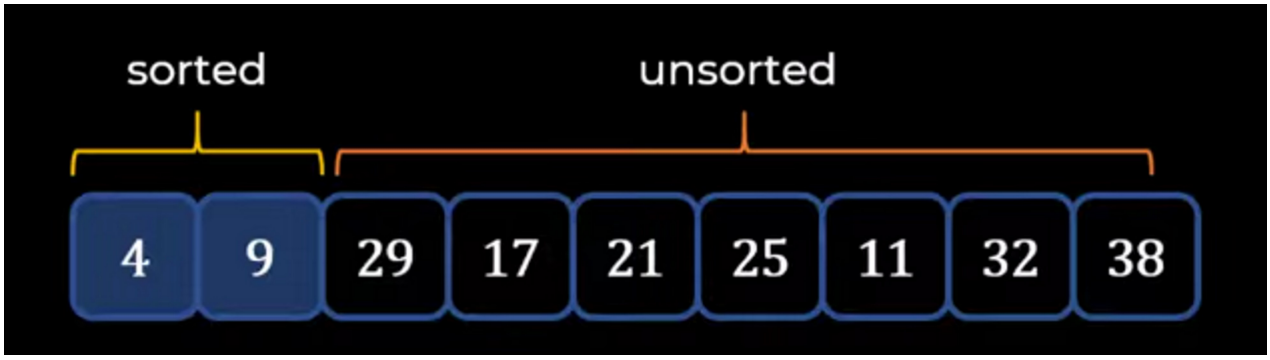
```
is the swap.
                gap =gap //2# We reduce the gap by 2.

if__name__ =='__main__':
        tests = [[21,38,29,17,4,25,11,32,9],
                            [],
                            [1,5,7,8],
                            [234,3,1,56,34,12,9,12,1399],
                            [5]
                          ]
        fortest intests:
                print(f'{test}to: ')
                shell_sort(test)
                print(test)
```

Exercise: https://github.com/codebasics/data-structures-algorithms-python/blob/master/algorithms/6_ShellSort/shell_sort_exercise.md

17 - Selection Sort

miércoles, 24 de abril de 2024 15:16

With selection Sort, we select the minimum number of the unsorted part of the array, and swap it with the element next to the sorted part for it to be included in the already sorted subarray. This is done until the whole array has been sorted.

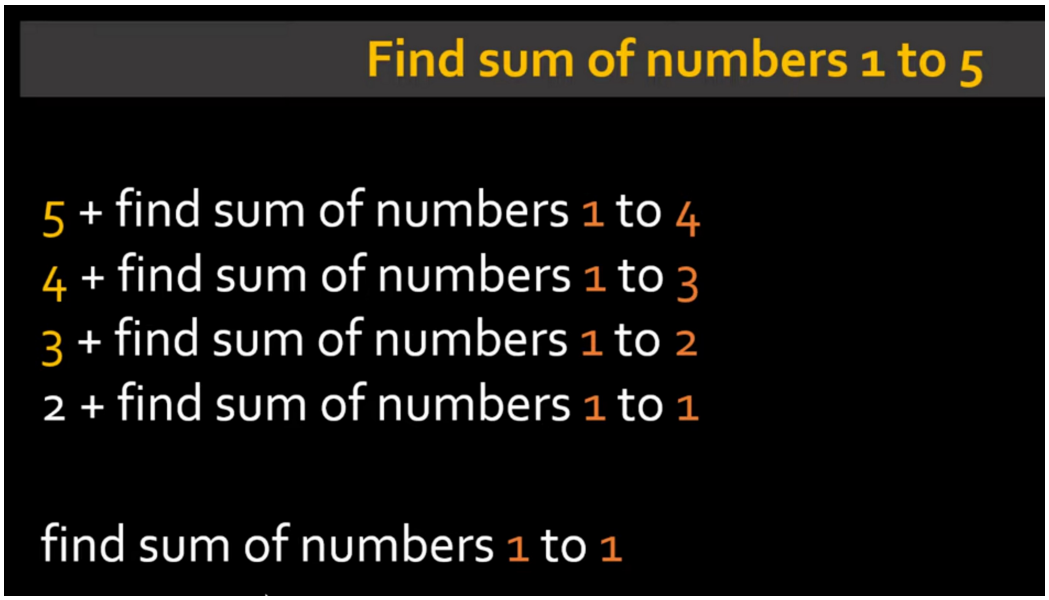


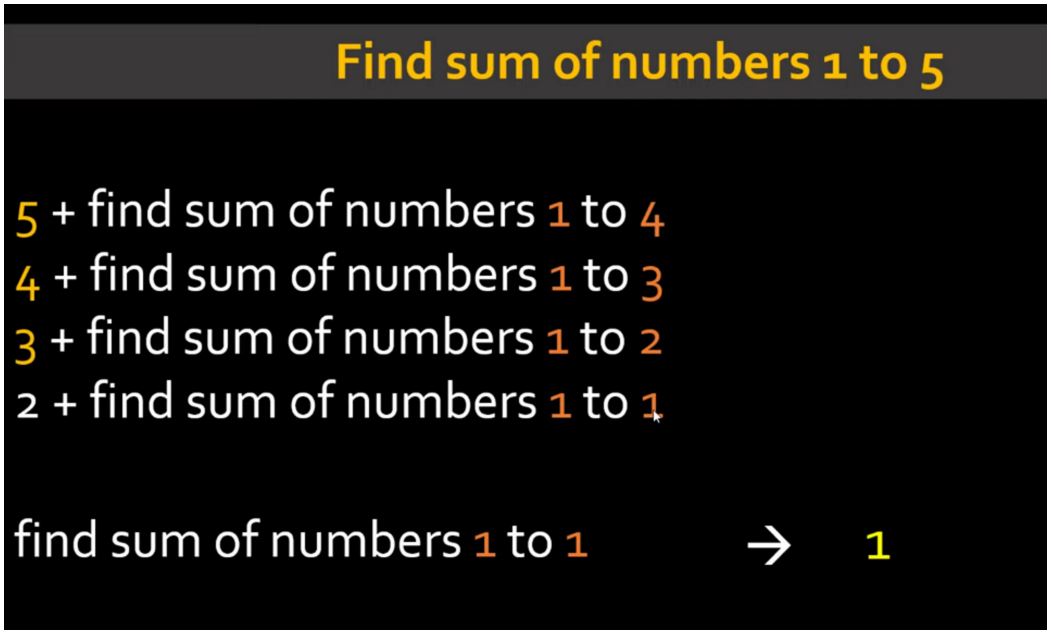
Exercises: https://github.com/codebasics/data-structures-algorithms-python/blob/master/algorithms/7_SelectionSort/selection_sort_exercise.md

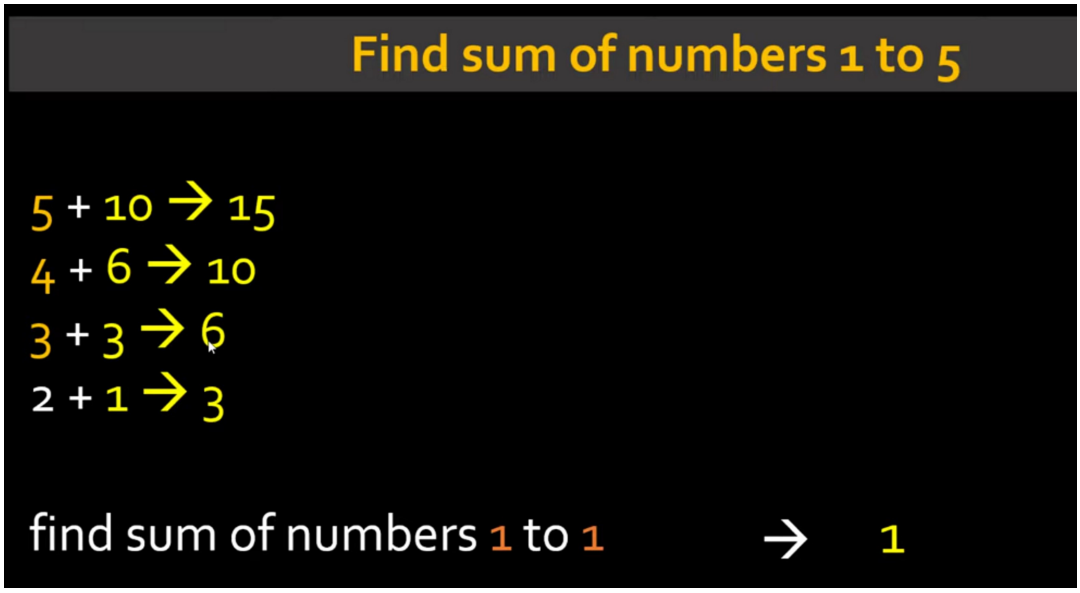
18 - Recursion

miércoles, 24 de abril de 2024 16:40

1. Divide big problem into small and simple problem.
2. Find a base condition with a simple answer.
3. Return base condition answer to solve all subproblems.








```
deffind_sum(n):
        """
    Find the sum of all the elements from 1 up to n.
    """
        ifn==1:# 2. Base condition to terminate the recursion.
                return1# 3. Return base condition with simple answer.

        returnn +find_sum(n-1)# 1. Divide big problem into small and simple
problem.

if__name__ =='__main__':
        print(find_sum(5))
```