Data Structures - Python

List

Lists are one of the four built-in data structures in Python, together with tuples, sets and dictionaries. They are used to store an ordered collection of items, which can be of different types.

The elements of a list are separated by commas and enclosed in square brackets. They are indexed according to a definite sequence and the indexing of a list is done with 0 being the first index. Lists are mutable, and hence, they can be altered even after their creation.

Creating a list

Lists in Python can be created by just writing a sequence inside the square brackets, where the elements of the sequence are separated by commas.
A list may contain duplicate values with their distinct positions and hence, multiple distinct or duplicate values can be passed as a sequence at the time of list creation.
To know the size of the list, the function `len()` can be used.
In order to concatenate two lists, use the + operation.

INPUT

```
# Creating an empty List
List = []
print("Blank List:", List)

# Creating a List. This list has mixed data types and duplicates
List = [1, "cat", 736.8, "London", -54, 1]
print("List:", List)
print("This list has length", len(List))

# Concatenate two lists
List1 = [0,2]
print("List and List1 concatenated:", List + List1)
```

OUTPUT

```
Blank List: []
List: [1, 'cat', 736.8, 'London', -54, 1]
This list has length 6
List and List1 concatenated: [1, 'cat', -736.8, 'London', 1, 0, 2]
```

Adding elements to a list: append, insert and extend methods

Elements can be added to a list by using the built-in `append()` function. Only one element at a time can be added to the list by using `append()`.

`append()` only works for addition of elements at the end of the list. In order to add an element at a desired position, the `insert()` method can be used. Unlike `append()` which takes only one argument, the `insert()` method requires two arguments, the position and the value.

`extend()` is used to add multiple elements at the same time at the end of the list.

INPUT

```
# Addition of elements in a list
List.append("Edinburgh")
List.append(2)
print("List after addition of Three elements:", List)

# Addition of a list to a list
List2 = [102.3, "frog"]
List.append(List2)
print("List after addition of a List:", List)

# Addition of and element to the list at specific position
List.insert(4, 6)
print("List after using the inster method:", List)

# Addition of multiple elements to the end of the list
List.extend([12, "John"])
print("List after performing Extend Operation:", List)
```

OUTPUT

```
List after Addition of Three elements: [1, 'cat', -736.8,
'London', 1, 'Edinburgh', 2]
List after Addition of a List: [1, 'cat', -736.8, 'London', 1,
'Edinburgh', 2, [102.3, 'frog']]
List after using the insert method: [1, 'cat', -736.8, 'London',
6, 1, 'Edinburgh', 2, [102.3, 'frog']]
List after using the extend method: [1, 'cat', -736.8, 'London',
6, 1, 'Edinburgh', 2, [102.3, 'frog'], 12, 'John']
```

Accessing elements from a list

To access an item in a list, the index operator [ ] is used. The index must be an integer. Negative sequence indexes represent positions from the end of the list: -1 refers to the last element, -2 refers to the second-last element, etc.

INPUT

```
# Accesing the first and third element from a list
print("The first and third elements form our list are:", List[0],
"and", List[2])

# Accesing the last element of a list
print("The last element of our list is:", List[-1])

# Accesing the fourth last element of a list
print("The third last element of our list is:", List[-4])
```

OUTPUT

```
The first and third elements form our list are: 1 and -736.8
The last element of our list is: John
The third last element of our list is: 2
```

Removing elements from a list: remove and pop methods

Elements can be removed from a list by using the built-in `remove()`. The `remove()` method only removes one element at a time. If the element to be remove appears more than once in the list, the `remove()` method will only remove the first occurrence of the searched element.

The `pop()` method is used to remove and return an element from the list. By default, it removes and return the last element. To specify the position of the element to remove and return, an index of the element is passed as an argument to the `pop()` method.

INPUT

```
# Removing specific elements form the list
List.remove(2)
List.remove("London")
print("List after removing two elements:", List)
```

```
# Removing and returning the last element from the list
print("The method pop returns the element:", List.pop())
print("List after popping an element:", List)

# Removing and returning an specific element from the list
print("The method pop, specifying the index 3 returns the
element:", List.pop(3))
print("List after popping a specific element:", List)
```

OUTPUT

```
List after removing two elements: [1, 'cat', -736.8, 6, 1,
'Edinburgh', [102.3, 'frog'], 12, 'John']
The method pop returns the element: John
List after popping an element: [1, 'cat', -736.8, 6, 1,
'Edinburgh', [102.3, 'frog'], 12]
The method pop, specifying the index 3 returns the element: 6
List after popping a specific element: [1, 'cat', -736.8, 1,
'Edinburgh', [102.3, 'frog'], 12]
```

Slicing of a list

To select a specific range of elements from the list, the slice operation is used.
To select elements from the beginning to a range [: Index] is used.
To select elements from specific Index till the end [Index:] is used.
To select elements within a range, [Start index:End index] is used.
To select the entire list in reverse order, [::-1] is used.

INPUT

```
# Printing elements of a range
print("The elements in a the index range 3 to 6 are :", List[3:6])

# Printing elements from an index until the end
print("The elements from index 5 of the list till the end are:",
List[5:])

# Printing elements from the beginning until the third last
element
print("The elements from the beginning until the third last
element of the list are:", List[:-3])

# Printing elements in reverse order
print("Printing list in reverse:", List[::-1])
```

OUTPUT

```
The elements in a the index range 3 to 6 are : [1, 'Edinburgh',
[102.3, 'frog']]
The elements from index 5 of the list till the end are: [[102.3,
'frog'], 12]
The elements from the beginning until the third last element of
the list are: [1, 'cat', -736.8, 1]
Printing list in reverse: [12, [102.3, 'frog'], 'Edinburgh', 1,
-736.8, 'cat', 1]
```

Tuples

A tuple is a collection of objects which are ordered and immutable, i.e, they cannot be changed.
Creating a tuple is as simple as putting different comma-separated values. Optionally these comma-separated values can be put between parentheses.
The empty tuple is written as two parentheses containing nothing.
To write a tuple containing a single value a comma needs to be included, even though there is only one value
Like lists, tuple indices start at 0, and they can be concatenated and sliced in the same way.
Note that even though tuples cannot be updated or changed, portions of tuples can be taken in order to create new ones.

INPUT

```
#Defining and printing tuples and their length
tuple1 = "a", "b", "c", "d"
tuple2 = (1, 2, 3, 4, 5)
tuple3 = ("Python", 1992, "Java", 1987)
tuple4 = ()
tuple5 = (100,)
print("The tuples:",tuple1,",",tuple2,",",tuple3,",",tuple4,",",tu
ple5)
print("are of length:",len(tuple1),",",len(tuple2),",",len(tuple3)
,",",len(tuple4),"and",len(tuple5),"respectively.", )
```

OUPUT

```
The tuples: ('a', 'b', 'c', 'd') , (1, 2, 3, 4, 5) , ('Python',
1992, 'Java', 1987) , () , (100,)
are of length: 4 , 5 , 4 , 0 and 1 respectively.
```

INPUT

```
#Slicing and concatenating tuples
tuple6 = tuple1[0:3]
tuple7 = tuple2 + tuple3
print("The tuple:",tuple6,"is created by slicing",tuple1,"from ind
ex 0 to 3.")
print("The tuple:",tuple7,"is created by contatenating",tuple6,"an
d", tuple7)
```

OUPUT

```
The tuple: ('a', 'b', 'c') is created by slicing ('a', 'b', 'c',
'd') from index 0 to 3.
The tuple: (1, 2, 3, 4, 5, 'Python', 1992, 'Java', 1987) is
created by contatenating ('a', 'b', 'c') and (1, 2, 3, 4, 5,
'Python', 1992, 'Java', 1987)
```

INPUT

```
#Tuples are inmutable. The following will throw an error.
tuple1[0]=“e"
```

OUTPUT

```
---------------------------------------------------------------
---------
TypeError                                 Traceback (most recent
call last)
 in
      1 #Tuples are inmutable
----> 2 tuple1[0]="e"

TypeError: 'tuple' object does not support item assignment
```

Set

A set is a collection which is unordered and unindexed and they are written with curly brackets. Sets do not hold duplicate items. Items cannot be accessed by referring to an index a key. However, loops can be used or alternatively the in keyword can be use in order to check if a specified value is present in a set.
To determine how many items a set has, the `len()` function can be used.

INPUT

```
#Creating a set
set={"SVM", "Discriminant Analysis", "Naive Bayes", "KNN"}
print("This is our set:",set)

#Looping through the set and printing its items
print("The items in our set are:")
for item in set:
    print(item)

#Using the keyword in to check if an item is in our set
print("Print True if SVM is in our set:", "SVM" in set)

#Determining how many items our set has
print("Our list has", len(set), "items.")
```

OUTPUT

```
This is our set: {'Naive Bayes', 'Discriminant Analysis', 'SVM',
'KNN'}
The items in our set are:
Naive Bayes
Discriminant Analysis
SVM
KNN
Print True if SVM is in our set: True
Our list has 4 items.
```

Changing, adding and removing items in a set

Once a set is created, items cannot be changed. However, items can be added. One item is added using the `add()` method. More than one item are added using the `update()` method. The elements to be added must be added into square brackets.

To remove an item in a set, the `remove()` or the `discard()` method are used. If the item does not exist in the set the `remove()` method will throw an error whereas the `discard()` method will not.

The `pop()` method can be used to remove an item. but this method Sets are unordered, so the item removed is unknown. The return value of the `pop()` method is the removed item.

To clear a set use the `clear()` method.

INPUT

```
#Adding an item using the add method
set.add("Neural Networks")
print("Our set after adding an item:", set)

#Adding multiple items using the update method
set.update(["Decision Tree", "Logistic Regression"])
print("Our set after adding two new items:", set)

#Removing one element using the remove method
set.remove("KNN")
print("Our set after removing KNN:", set)

#Removing one element using the discard method
set.discard("Naive Bayes")
print("Our set after removing Naive Bayes:", set)

#Removing one element using the pop method
poppedItem = set.pop()
print("Our set after popping one element:", set)
print("The element popped was:", poppedItem)

#Removing every element of our set using the clear method
set.clear()
print("After using the clear method our set has now", len(set),
“elements.")
```

OUTPUT

```
Our set after adding an item: {'Naive Bayes', 'KNN', 'Neural
Networks', 'Discriminant Analysis', 'SVM'}
Our set after adding two new items: {'Naive Bayes', 'KNN', 'Neural
Networks', 'Decision Tree', 'Discriminant Analysis', 'SVM',
'Logistic Regression'}
Our set after removing KNN: {'Naive Bayes', 'Neural Networks',
'Decision Tree', 'Discriminant Analysis', 'SVM', 'Logistic
Regression'}
Our set after removing Naive Bayes: {'Neural Networks', 'Decision
Tree', 'Discriminant Analysis', 'SVM', 'Logistic Regression'}
Our set after popping one element: {'Decision Tree', 'Discriminant
Analysis', 'SVM', 'Logistic Regression'}
The element popped was: Neural Networks
After using the clear method our set has now 0 elements.
```

Other useful set methods for sets

Method	Description
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another specified set
intersection()	Returns a set that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets

Dictionary

A dictionary is a collection which is unordered, changeable and indexed. The items of a dictionary have keys and values and they are written with curly brackets.
Items can be accessed by writing their key name in square brackets or by using the `get()` method.
The `len()` function is used to determine how many items a dictionary has.

INPUT

```
#Creating and printing a dictionary
dictionary = {
    "name": "Moonface",
    "race": "Tiefling",
    "class": "barbarian",
    "level": 3
}
print("This is our dictionary:",dictionary)

#Accesing an item in two ways
print("Name:", dictionary["name"])
print(“Race:”,dictionary.get("race"))

#Determining the number of items in our dictionary
print("Our dictionary has", len(dictionary), "items.")
```

OUTPUT

```
This is our dictionary: {'name': 'Moonface', 'race': 'Tiefling',
'class': 'barbarian', 'level': 3}
Name: Moonface
Race: Tiefling
Our dictionary has 4 items.
```

Looping through a dictionary

Looping is possible through a dictionary. Keys can be returned but also values and pairs of keys and values using the `items()` method.

INPUT

```
#Printing every key of our dictionary
print("The keys of our dictionary are:")
for key in dictionary:
    print(key)

#Printing every value of our dictionary
print("The values of our dictionary are:")
for key in dictionary:
    print(dictionary[key])
```

```
#Printing every value of our dictionary using the values method
print("We obtain the same result using the values method in our lo
op:")
for values in dictionary.values():
    print(values)

#Printing every pair of keys and values of our dictionary
print("The items of our dictionary are:")
for keys, values in dictionary.items():
    print(keys,values)
```

```
OUTPUT

The keys of our dictionary are:
name
race
class
level
The values of our dictionary are:
Moonface
Tiefling
barbarian
3
We obtain the same result using the values method in our loop:
Moonface
Tiefling
barbarian
3
The items of our dictionary are:
name Moonface
race Tiefling
class barbarian
level 3
```

Changing, removing and adding values

Values of a specific item can be changed by referring to its key name.
Adding an item to the dictionary is done by using a new index key and assigning a value to it.
To remove an item, the `pop()` and `popitem()` methods and the keyword del can be used.
The `pop()` method removes the item with the specified key name.
The `popitem()` method removes the last inserted item (in older versions than Python 3.7, a random item is removed instead).
The del keyword removes the item with the specified key name. The del keyword can also delete the dictionary completely. The `clear()` method empties the dictionary.

INPUT

```
#Changing the value of an item
dictionary["level"]=4
print("Level is now ugraded to", dictionary["level"],":", dictiona
ry)

#Adding an item to the dictionary
dictionary["experience"] = 2700
print("We have added the item 'experience' to our dictiornary:", d
ictionary)

#Removing an item using the pop method
dictionary.pop("experience")
print("Our dictionary after removing the item 'experience':", dict
ionary)

#Removing the last item using the popitem method
dictionary.popitem()
print("Our dictionary after removing the last item:", dictionary)

#Removing an item using the del keyword
del dictionary["class"]
print("Our dictionary after removing the item 'class':", dictionar
y)

#Removing every item in the dictionary using the clear method
dictionary.clear()
print("Our dictionary after removing every element:", dictionary)

#Deleting our dictionary
del dictionary
print("This will throw an error as our dictionary no longer exists
:", dictionary)
```

OUTPUT

```
Level is now ugraded to 4 : {'name': 'Moonface', 'race':
'Tiefling', 'class': 'Barbarian', 'level': 4}
We have added the item 'experience' to our dictionary: {'name':
'Moonface', 'race': 'Tiefling', 'class': 'Barbarian', 'level': 4,
'experience': 2700}
Our dictionary after removing the item 'experience': {'name':
'Moonface', 'race': 'Tiefling', 'class': 'Barbarian', 'level': 4}
Our dictionary after removing the last item: {'name': 'Moonface',
'race': 'Tiefling', 'class': 'Barbarian'}
```

```
Our dictionary after removing the item 'class': {'name':
'Moonface', 'race': 'Tiefling'}
Our dictionary after removing every element: {}
---------------------------------------------------------------------
---------
NameError                                 Traceback (most recent
call last)
 in
     25 #Deleting our dictionary
     26 del dictionary
---> 27 print("This will throw an error as our dictionary no
longer exists:", dictionary)
     28
     29

NameError: name 'dictionary' is not defined
```

Nested Dictionaries

A dictionary can also contain many dictionaries, this is called nested dictionaries.
It is also possible to nest three dictionaries that already exists as dictionaries.

INPUT

```
#Creating and printing a nested dictionary
books = {
    "book1" : {
        "title" : "Brave new world",
         "year" : 1932 },
    "book2" : {
        "title" : "The curious incident of the dog in the night-time",
        "year" : 2003},
    "book3" : {
        "title" : "Momo",
        "year" : 1973}
}
print("Our nested dictionaries:", books)
```

```
#Creating a nested dictionary from existing dictionaries and print
ing it
book1 = {
    "title" : "Brave new world",
    "year" : 1932
    }
book2 = {
    "title" : "The curious incident of the dog in the night-time",
    "year" : 2003
    }
book3 = {
    "title" : "Momo",
    "year" : 1973
    }

books = {
    "book1" : book1,
    "book2" : book2,
    "book3" : book3
}
print("This prints the same output:", books)
```

```
OUTPUT

Our nested dictionaries: {'book1': {'title': 'Brave new world',
'year': 1932}, 'book2': {'title': 'The curious incident of the dog
in the night-time', 'year': 2003}, 'book3': {'title': 'Momo',
'year': 1973}}
This prints the same output: {'book1': {'title': 'Brave new
world', 'year': 1932}, 'book2': {'title': 'The curious incident of
the dog in the night-time', 'year': 2003}, 'book3': {'title':
'Momo', 'year': 1973}}
```

The dict constructor

It is also possible to use the `dict()` constructor to make a new dictionary. Note that keywords are not string literals and that equals are used instead of colons.

```
#Creating a dictionary using the dict constructor
band = dict(name="MOPA", genre="Rock", nationality="French")
print("Our new dictionary:", band)
```

```
Our new dictionary: {'name': 'MOPA', 'genre': 'Rock',
'nationality': 'French'}
```

Other dictionary methods

Method	Description
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs

Stack

A stack is a data structure that stores items in a Last-In/First-Out way. This is frequently referred to as LIFO. This is in contrast to a queue, which stores items in a First-In/First-Out (FIFO) way.

There are different way to implement a stack in Python. Some of them are: lists, collections.deque and queue.LifoQueue.

Using lists to implement a stack

Lists can be used as a stack. Instead of `push()`, `append()` can be used to add new elements to the top of the stack, while `pop()` removes the elements in the LIFO order.

INPUT

```
#Creating a stack using a list
stack = []
#Adding elements to the stack
stack.append('a')
stack.append('b')
stack.append('c')
print(stack)
```

OUTPUT

```
['a', 'b', ‘c’]
```

```
INPUT

#Removing the last element from the stack
stack.pop()

OUTPUT

‘c'

INPUT

#Current state of the stack
print(stack)

OUTPUT

['a', ‘b']

INPUT
#Removing the last element from the stack
stack.pop()

OUTPUT

‘b’

INPUT

#Current state of the stack
print(stack)

OUTPUT

[‘a’]

INPUT

#Removing the last element from the stack
stack.pop()

OUTPUT

‘a'
```

```
INPUT
#Current state of the stack
print(stack)

OUTPUT

[]

INPUT

#Trying to remove the last element of the stack
stack.pop()

OUTPUT

---------------------------------------------------------------------
---------
IndexError                                Traceback (most recent
call last)
<ipython-input-218-415460d3b717> in <module>
----> 1 stack.pop()

IndexError: pop from empty list
```

Even though lists have the advantage of being familiar an easy to use, items in a list are stored with the goal of providing fast access to random elements in the list. At a high level, this means that the items are stored next to each other in memory.
If a stack grows bigger than the block of memory that currently holds it, then Python needs to do some memory allocations. This can lead to some `.append()` calls taking much longer than other ones.

Using collections.deque to implement a stack

The collections module contains deque, which is useful for creating Python stacks. deque is pronounced “deck” and stands for “double-ended queue.”

The methods `.append()`, and `.pop()` can also used on deque.

INPUT

```
#Creating a stack using collections.deque
from collections import deque
stack = deque()
#Adding elements to the stack
stack.append('a')
stack.append('b')
stack.append('c')
print(stack)
```

OUTPUT

```
deque(['a', 'b', ‘c’])
```

INPUT

```
#Removing the last element from the stack
stack.pop()
```

OUTPUT

```
‘c'
```

INPUT

```
#Current state of the stack
print(stack)
```

OUTPUT

```
deque(['a', ‘b’])
```

INPUT

```
#Removing the last element from the stack
stack.pop()
```

OUTPUT

```
‘b'
```

INPUT

```
#Current state of the stack
print(stack)
```

```
OUTPUT

'a'

INPUT

#Current state of the stack
print(stack)

OUTPUT

deque([])

INPUT

#Removing the last element from the stack
stack.pop()

OUTPUT
---------------------------------------------------------------------
---------
IndexError                                Traceback (most recent
call last)
<ipython-input-229-9c58c1a3387f> in <module>
      1 #Removing the last element from the stack
----> 2 stack.pop()

IndexError: pop from an empty deque
```

Note that the piece of code above looks similar to the list example.

deque is built upon a doubly linked list - see linked list section. In a linked list structure, each entry is stored in its own memory block and has a reference to the next entry in the list.
A doubly linked list is just the same, except that each entry has references to both the previous and the next entry in the list. This allows to add nodes to either end of the list easily.
Adding a new entry into a linked list structure only requires setting the new entry's reference to point to the current top of the stack and then pointing the top of the stack to the new entry.

deque is a good choice for implementing a stack in Python when the code doesn't use threading.

Python stacks and threading

Python stacks can be useful in multi-threaded programs as well.
Lists should never be used for any data structure that can be accessed by multiple threads.
When using deque, a restriction to use only `append()` and `pop()` is thread safe.
The concern with using deque in a threaded environment is that there are other methods in that class, and those are not thread safe. Therefore, while it's possible to build a thread-safe stack in Python using a deque, it can cause problems it it is misused.
For a threaded program, a safe choice is to use queue.LifoQueue. While the interface for list and deque were similar, LifoQueue uses `put()` and `get()` to add and remove data from the stack.

INPUT

```
#Creating a stack using queue.LifoQueue

from queue import LifoQueue
stack = LifoQueue()
stack.put('a')
stack.put('b')
stack.put('c')
print(stack)
```

OUTPUT

```
<queue.LifoQueue object at 0x111724520>
```

INPUT

```
#Removing the last element from the stack
stack.get()
```

OUTPUT

```
'c'
```

INPUT

```
#Removing the last element from the stack
stack.get()
```

OUTPUT

```
‘b’
```

```
INPUT
#Removing the last element from the stack
stack.get()

OUTPUT

'a'

INPUT

#Trying to remove that las element from the stack using .get_nowai
t() as .get() is very slow when the stack is empty
stack.get_nowait()


OUTPUT
---------------------------------------------------------------
---------
Empty                                     Traceback (most recent
call last)
<ipython-input-240-58278300640c> in <module>
----> 1 stack.get_nowait()

/usr/local/Cellar/python@3.8/3.8.5/Frameworks/Python.framework/
Versions/3.8/lib/python3.8/queue.py in get_nowait(self)
    196         raise the Empty exception.
    197         '''
--> 198         return self.get(block=False)
    199
    200     # Override these methods to implement other queue
organizations

/usr/local/Cellar/python@3.8/3.8.5/Frameworks/Python.framework/
Versions/3.8/lib/python3.8/queue.py in get(self, block, timeout)
    165             if not block:
    166                 if not self._qsize():
--> 167                     raise Empty
    168             elif timeout is None:
    169                 while not self._qsize():

Empty:
```

Unlike deque, LifoQueue is designed to be fully thread-safe. All of its methods are safe to use in a threaded environment. It also adds optional time-outs to its operations which can frequently be a must-have feature in threaded programs.

Queue

A queue is a collection of objects that supports fast First-In/First-Out (FIFO) semantics for inserts and deletes. The insert and delete operations are sometimes called enqueue and dequeue. Unlike lists, queues typically don't allow for random access to the objects they contain.

As mentioned before, queues are similar to stacks. The difference between them lies in how items are removed. With a queue, the item removed is the least recently added (FIFO) but with a stack, the item removed is the most recently added (LIFO).

Performance-wise, a proper queue implementation is expected to take O(1) time for insert and delete operations. These are the two main operations performed on a queue, and in a correct implementation, they should be fast.

Queues have a wide range of applications in algorithms and often help solve scheduling and parallel programming problems. A well-known algorithm using a queue is breadth-first search (BFS) on a tree or graph data structure - see tree section.

Scheduling algorithms often use priority queues internally. These are specialised queues. Instead of retrieving the next element by insertion time, a priority queue retrieves the highest-priority element. The priority of individual elements is decided by the queue based on the ordering applied to their keys. A regular queue, however, won't reorder the items it carries.

Using lists to implement a queue

It's possible to use a regular list as a queue, but this is not ideal from a performance perspective. Lists are quite slow for this purpose because inserting or deleting an element at the beginning requires shifting all the other elements by one, requiring O(n) time.

INPUT

```
#Creating a queue using lists
queue = []
queue.append("a")
queue.append("b")
queue.append("c")
queue
```

OUTPUT

```
['a', 'b', 'c']
```

INPUT

```
#Removing the first element from the queue
queue.pop(0)
```

OUTPUT

‘a’

INPUT

```
#Current state of the queue
print(queue)
```

OUTPUT

['b', ‘c’]

INPUT

```
#Removing the first element from the queue
queue.pop(0)
```

OUTPUT
‘b'

INPUT

```
#Current state of the queue
print(queue)
```

OUTPUT

[‘c’]

INPUT

```
#Removing the first element from the queue
queue.pop(0)
```

OUTPUT

‘c'

```
INPUT

#Current state of the queue
print(queue)

OUTPUT

[]

INPUT

#Removing the first element from the queue
queue.pop(0)


OUTPUT
---------------------------------------------------------------------
---------
IndexError                                Traceback (most recent
call last)
<ipython-input-260-1138bcbb3520> in <module>
      1 #Removing the first element from the queue
----> 2 queue.pop(0)

IndexError: pop from empty list
```

Using collections.deque to implement a queue

The deque class implements a double-ended queue that supports adding and removing elements from either end in O(1) time. Due to the fact that deques support adding and removing elements from either end equally well, they can serve both as queues and as stacks.

```
INPUT

#Creating a queue using collections.deque

from collections import deque

queue = deque()
queue.append("a")
queue.append("b")
queue.append("c")

print(queue)
```

```
OUTPUT

deque([‘a', 'b', ‘c’])

INPUT

#Removing the first element from the queue
queue.popleft()

OUTPUT

‘a’

INPUT

#Current state of the queue
print(queue)

OUTPUT

deque([‘b', ‘c’])

INPUT

#Removing the first element from the queue
queue.popleft()

OUTPUT

‘b’

INPUT

#Current state of the queue
print(queue)

OUTPUT

deque([‘c’])

INPUT

#Removing the first element from the queue
queue.popleft()
```

```
OUTPUT

'c'

INPUT

#Current state of the queue
print(queue)

OUTPUT

deque([])

INPUT

#Removing the first element from the queue
queue.popleft()


OUTPUT
---------------------------------------------------------------------
---------
IndexError                                Traceback (most recent
call last)
<ipython-input-271-7e4b8bcc67b3> in <module>
      1 #Removing the first element from the queue
----> 2 queue.popleft()

IndexError: pop from an empty deque
```

Implementing a queue using queue.Queue

queue.Queue is a built-in module in Python to implement queues useful for parallel computing.

```
INPUT

#Creating a queue using queue.Queue
from queue import Queue
queue = Queue()
queue.put("a")
queue.put("b")
queue.put("c")
queue
```

```
OUTPUT

<queue.Queue at 0x11222bd60>

INPUT

#Removing the first element from the queue
queue.get()

OUTPUT

‘a'

INPUT

#Removing the first element from the queue
queue.get()

OUTPUT

‘b’

INPUT

#Removing the first element from the queue
queue.get()

OUTPUT

‘c’

INPUT

#Removing the first element from the queue
queue.get_nowait()

OUTPUT
---------------------------------------------------------------------
---------
Empty                                     Traceback (most recent
call last)
<ipython-input-280-bdede1d6772e> in <module>
      1 #Removing the first element from the queue
----> 2 queue.get_nowait()

/usr/local/Cellar/python@3.8/3.8.5/Frameworks/Python.framework/
Versions/3.8/lib/python3.8/queue.py in get_nowait(self)
```

```
    196         raise the Empty exception.
    197         '''
--> 198         return self.get(block=False)
    199
    200     # Override these methods to implement other queue
organizations

/usr/local/Cellar/python@3.8/3.8.5/Frameworks/Python.framework/
Versions/3.8/lib/python3.8/queue.py in get(self, block, timeout)
    165             if not block:
    166                 if not self._qsize():
--> 167                     raise Empty
    168             elif timeout is None:
    169                 while not self._qsize():

Empty:
```

Quick Review on Python Objects and Classes

Python is an object oriented programming (OOP) language. Objects have properties and methods. A class is like an object constructor, or a "blueprint" for creating objects. An object is also called an instance of a class and the process of creating this object is called instantiation.

OOP focuses on the objects rather than the logic required to manipulate them. This approach to programming is well-suited for programs that are large, complex and often updated or maintained.

The way an object-oriented program is organised also makes the method beneficial to projects that are divided into group. This is known as collaborative development.

Other benefits of OOP include code reusability, scalability and efficiency.

Object-oriented programming is based on the following principles: encapsulation, abstraction, inheritance and polymorphism.

- Encapsulation: objects are privately held inside a class. Other objects do not have access to this class and cannot make changes. However, they can call a list of public functions, or methods. This provides the program with more security and avoids unintended data corruption.

- Abstraction: Objects only reveal internal mechanisms that are relevant for the use of other objects, hiding any unnecessary implementation code. This concept helps make changes and additions over time in an easier way.

- Inheritance: Relationships and subclasses between objects can be assigned, allowing to reuse a common logic while still maintaining a unique hierarchy. This reduces development time and ensures a higher level of accuracy.

- Polymorphism: two or more classes are polymorphic when they contain methods that have different implementations while they have identical names. In such a case, objects of these polymorphic classes can be used without considering differences across the classes. It allows to have one interface to perform similar tasks in many different ways. Polymorphism makes the code easy to change, maintain and extend

Creating classes and objects

All classes have a function called `__init__()`, which is always executed when the class is being initiated. The `__init__()` function is used to assign values to object properties, or other operations that are necessary to do when the object is being created. The parameter self has to be the first parameter of any function in the class and it does not have to be named self necessarily.

```
INPUT

#Creating the Person class. The __init__() function is used to ass
ign values for name and age
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

#Creating the Object person1
person1 = Person("John", 27)

#Printing the properies name and age of person1
print(person1.name)
print(person1.age)

OUTPUT

John
27
```

Object methods

Objects can also contain methods. Methods in objects are functions that belong to the object.

INPUT

```
#Creating a method in the Person class.
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduction(self):
        print("Hello my name is " + self.name)

person1 = Person("John", 28)
person1.introduction()
```

OUTPUT

```
Hello my name is John
```

Modifying and deleting object properties

It is possible to modify properties on objects. Properties on objects can also be deleted by using the del keyword.

INPUT

```
#Setting the age of person1 to 40
person1.age = 40
print(person1.age)
#Deleting the age property from the people1 object
del person1.age
print(person1.age)
```

```
OUTPUT

40
---------------------------------------------------------------------
---------
AttributeError                            Traceback (most recent
call last)
 in
      4 #Deleting the age property from the people1 object
      5 del person1.age
----> 6 print(person1.age)

AttributeError: 'Person' object has no attribute 'age'
```

Deleting objects

Objects can be deleted by using the del keyword.

```
INPUT

#Deleting the person1 object
del person1
print(person1)

OUTPUT

---------------------------------------------------------------------
---------
NameError                                 Traceback (most recent
call last)
 in
      1 #Deleting the person1 object:
      2 del person1
----> 3 print(person1)

NameError: name 'person1' is not defined
```

The pass statement

Class definitions cannot be empty, but if you for some reason a class has no content, the pass statement has to be added in order to avoid getting an error.

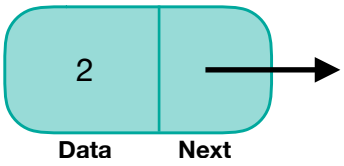
INPUT

```
#Adding the pass statement to an empty class to avoid errors
class Person:
    pass
```

Linked Lists

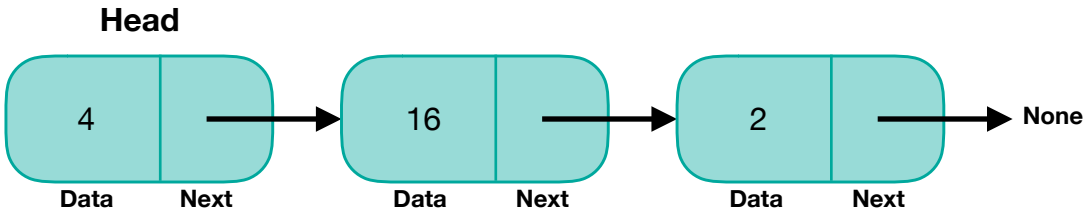
Linked lists are an ordered collection of objects and differ from lists in the way that they store elements in memory. While lists use a contiguous memory block to store references to their data, linked lists store references as part of their own elements.

Each element of a linked list is called a node, and every node has two different fields:



1. Data contains the value to be stored in the node.
2. Next contains a reference to the next node on the list.

The first node is called the head, and it's used as the starting point for any iteration through the list. The last node must have its next reference pointing to `None` to determine the end of the list.



Linked lists can be used to implement queues or stacks as well as graphs but they also useful for much more complex tasks, such as lifecycle management for an operating system application.

Collections.deque in order to implement a linked-list

deque allows to create a linked list. When initializing a deque object, any iterable can be passed as an input, such as a string or a list of objects. append() and pop() can be used to add add and remove elements on the right respectively.
Deque can also be used to to add or remove elements from the left side, or head, of the list.

INPUT

```
#Creating an empty linked list with deque
from collections import deque
LL1=deque()
print("An empty linked list:", LL1)
#Create two identical non empty linked lists with deque
LL2=deque(['a','b','c'])
LL3=deque('abc')
print("A linked list:", LL2)
print("The same linked list:", LL3)
print("Are LL2 and LL3 the same?:", LL2==LL3)
#Creating a more complex linked list with deque
LL4=deque([{'first': 'a'}, {'second': 'b'},{'third':'c'}])
print("A more complicated linked list:", LL4)
#Adding an element to a linked list on the right side
LL2.append('d')
print("LL2 after adding an element on the right:", LL2)
#Remove an element from a linked list from the right side
LL2.pop()
print("LL2 after removing an element from the right:", LL2)
#Adding an element to a linked list on the left side with deque
LL2.appendleft('z')
print("LL2 after adding an element on the left:", LL2)
#Remove an element from a linked list from the left side with dequ
e
LL2.appendleft('z')
print("LL2 after removing an element from the left:", LL2)
```

OUTPUT

```
An empty linked list: deque([])
A linked list: deque(['a', 'b', 'c'])
The same linked list: deque(['a', 'b', 'c'])
Are LL2 and LL3 the same?: True
A more complicated linked list: deque([{'first': 'a'}, {'second':
'b'}, {'third': 'c'}])
LL2 after adding an element on the right: deque(['a', 'b', 'c',
'd'])
```

```
LL2 after removing an element from the right: deque(['a', 'b',
'c'])
LL2 after adding an element on the left: deque(['z', 'a', 'b',
'c'])
LL2 after removing an element from the left: deque(['z', 'z', 'a',
'b', 'c'])
```

Implementing a listed link

It is possible to create linked lists using classes. First, a class for the linked list is created. The only information needed to be stored for a linked list is where the list starts, i.e, the head of the list. After that, a class to represent each node of the linked list needs to be created with the two main elements: data and next.

In order to have a representation of the linked list, `__repr__` can be added to both classes.

Transversing is the equivalent to iterating in a list but, in this case, for a linked list. Traversing means going through every single node, starting with the head of the linked list and ending on the node that has a next value of None.

Additional methods in order to add nodes at the start and end of the list, after and before nodes can be added as well as methods to delete the first node, the last node and a specific node.

INPUT

```
#Creating a linked list class
class LinkedList:
    def __init__(self):
        self.head = None

    def __repr__(self):
        result = ""
        ref = self.head
        while ref:
            result += str(ref.data) + " -> "
            ref = ref.next

        result = result.strip(" -> ")

        if len(result):
            return "[" + result + "]"
        else:
            return "[]"
```

```
    def traverseList(self):
        if self.head is None:
            print("List has no elements")
            return
        else:
            node = self.head
            while node is not None:
                print(node.data , " ")
                node = node.next

    def addFirst(self, node):
        node.next = self.head
        self.head = node

    def addLast(self, newdata):
        NewNode = Node(newdata)
        if self.head is None:
            self.head = NewNode
            return
        last = self.head
        while(last.next):
            last = last.next
        last.next=NewNode

    def insertAfter(self, prev_data, data):
        if prev_data is None:
            print('Can not add nodes in between of empty LinkedLis
t')
            return
        node=self.head
        while node is not None:
            if node.data==prev_data:
                break
            node=node.next
        new_node=Node(data)
        new_node.next=node.next
        node.next=new_node

    def insertBefore(self,prev_data,data):
        if prev_data is None:
            print('Can not add nodes in between of empty LinkedLis
t')
            return
        ref=self.head
        new_node=Node(data)
        while ref is not None:
            if ref.next.data==prev_data:
```

```
                break
            ref=ref.next
        new_node.next=ref.next
        ref.next=new_node

    def deleteHeadNode(self):
        if self.head is None:
            print('Can not delete nodes from empty LinkedList')
            return
        ref=self.head
        while self.head is not None:
            self.head=ref.next
            break
        ref.next=self.head.next

   def deleteLastNode(self):
        if self.head is None:
            print('Can not delete nodes from empty LinkedList')
            return
        ref=self.head
        while True:
            if ref.next.next is None:
                ref.next=None
                break
            ref=ref.next

    def deleteAfterNode(self,data):
        del_node=Node(data)
        if self.head is None:
            print('Can not delete Nodes from empty LinkedList')
            return
        ref=self.head
        while  True:
            if ref.data==del_node.data:
                break
            prev=ref
            ref=prev.next
        prev.next=ref.next

#Creating a node class
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    def __repr__(self):
        return self.data
```

```
#Creating a linked list with three nodes
LL1=LinkedList()
firstNode=Node('cat')
secondNode=Node('dog')
thirdNode=Node('bird')
LL1.head=firstNode
firstNode.next=secondNode
secondNode.next=thirdNode
print("Our three nodes linked list:")

OUTPUT

Our three nodes linked list:
[cat -> dog -> bird]

INPUT

LL1.traverseList()

OUTPUT

cat
dog
bird

INPUT

LL1.addFirst(Node("frog"))
print("LL1 after adding a new node at the start:", LL1)
LL1.addLast(Node("turtle"))
print("LL1 after adding a new node at the end:", LL1)
LL1.insertAfter("cat", "snake")
print("LL1 after adding a new node after a node:", LL1)
LL1.insertBefore("bird", "mouse")
print("LL1 after adding a new node before a node:", LL1)
LL1.deleteHeadNode()
print("LL1 after deleting the first node:", LL1)
LL1.deleteLastNode()
print("LL1 after deleting the last node:", LL1)
LL1.deleteAfterNode("dog")
print("LL1 after deleting a specific node:", LL1)
```

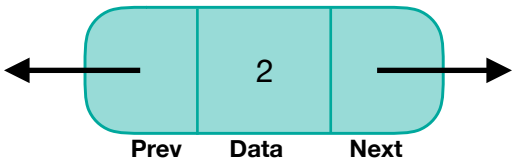
```
OUTPUT

LL1 after adding a new node at the start: [frog -> cat -> dog ->
bird]
LL1 after adding a new node at the end: [frog -> cat -> dog ->
bird -> turtle]
LL1 after adding a new node after a node: [frog -> cat -> snake ->
dog -> bird -> turtle]
LL1 after adding a new node before a node: [frog -> cat -> snake -
> dog -> mouse -> bird -> turtle]
LL1 after deleting the first node: [cat -> snake -> dog -> mouse -
> bird -> turtle]
LL1 after deleting the last node: [cat -> snake -> dog -> mouse ->
bird]
LL1 after deleting a specific node: [cat -> snake -> mouse ->
bird]
```

Doubly linked lists and circular linked lists

The type of linked lists covered so far are known as singly linked lists. But there are more types of linked lists that are used for different purposes.

Doubly linked lists

Doubly linked lists have two references instead of just one.



In order to implement a doubly linked list, the class Node needs to be modified:

```
INPUT

#Modifying the Node Class and modifying the method __repr__ for do
ubly linked lists

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

    def __repr__(self):
        return self.data
```

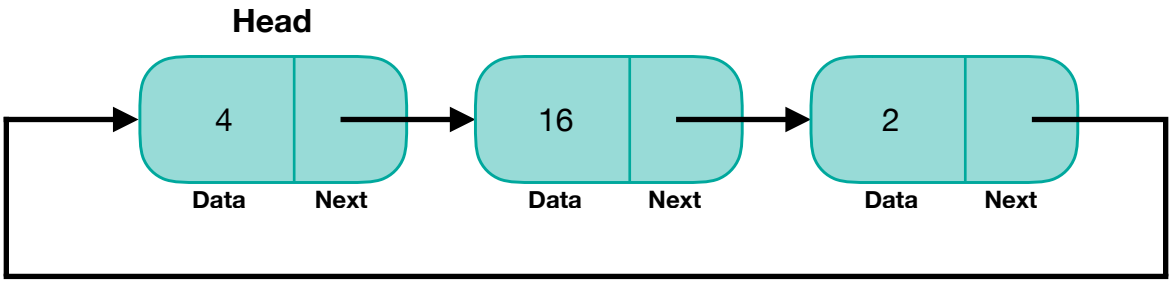
```
class DoublyLinkedList:
    def __init__(self):
        self.head = None

    def __repr__(self):
        result = ""
        ref = self.head
        while ref:
            result += str(ref.data) + " <-> "
            ref = ref.next

        result = result.strip(" <-> ")

        if len(result):
            return "[" + result + "]"
        else:
            return “[]"


#Creating a doubly linked list with three nodes
LL2=DoublyLinkedList()
firstNode=Node('sun')
secondNode=Node('moon')
thirdNode=Node('starts')
LL2.head=firstNode
firstNode.next=secondNode
secondNode.next=thirdNode
print("Our three nodes doubly linked list:")
print(LL2)

OUTPUT

Our three nodes doubly linked list:
[sun <-> moon <-> starts]
```

Note: `collections.deque` uses a linked list as part of its data structure. With doubly linked lists, `deque` can insert and delete elements from both ends of a queue with constant O(1) performance.

Circular linked lists

Circular linked lists are a type of linked list in which the last node instead of point to None, it points to the head of the list.

Circular linked lists have lots of interesting applications such as: going through each player's turn in a multiplayer game, managing the application life cycle of an operating system, implementing a Fibonacci heap....



Circular linked lists can be traversed starting at any node. In order to avoid an infinite loop, traversing needs to stop once the starting point has been reached. The starting point can be define when the list is traversed.

Often when implementing circular linked list, there exists a special link that doesn't contain meaningful data. Instead, it's a "sentinel" that provides information regarding where the beginning (and end) of the list is. This link will exist even when the list is empty, so algorithms will work on all lists, without lots of special cases needing special code.

INPUT

```
#Creating the Link and CircularLinkedList Class
class Link:
    def __init__(self, data, next):
        self.data = data
        self.next = next

class CircularLinkedList:
    def __init__(self):
        self.head = Link(None, None)
        self.head.next = self.head

    def addNode(self, data):
        self.head.next = Link(data, self.head.next)
```

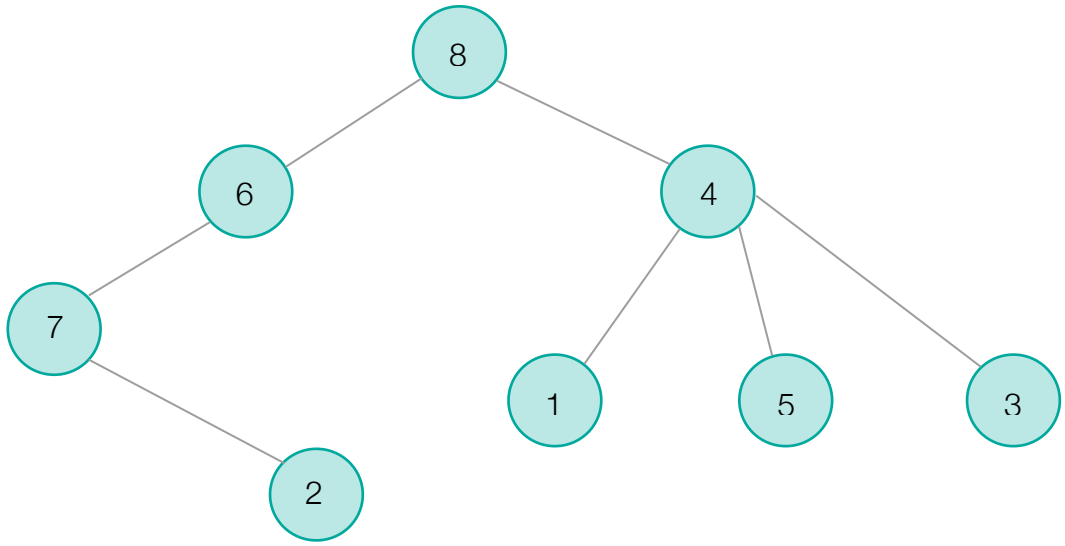
```
    def print(self):
        node=self.head
        while True:
            print(str(node.data) + "->",end="")
            if node.next is self.head:
                print(str(self.head.data),end='')
                break
            node = node.next
        print()

#Creating a circular linked list with 4 nodes
CLL1=CircularLinkedList()
CLL1.addNode("a")
CLL1.addNode("b")
CLL1.addNode("c")
CLL1.addNode("d")
CLL1.print()
```

OUTPUT

```
None->d->c->b->a->None
```

Trees

Trees are non-linear data structures that are represented by nodes connected by edges.



A **node** is an element of a tree equipped with a value (or key) and that points to its child nodes. The **root** It is the topmost node of a tree.
The last nodes of each path that do not point to child nodes are called **leaf nodes** or external nodes.
A node that has at least a child node is called a **parent node**.
In a representation of a tree, an **edge** it is the link between any two nodes.
The **height of a node** is the number of edges from the node to the deepest leaf.
The **depth of a node** is the number of edges from the root to the node.
The **height of a tree** is the height of the root node.
The **degree of a node** is the total number of branches of that node.
A collection of disjoint trees is called a **forest.**

Trees allow quicker and easier access to the data than other structures that are linear such as linked lists, queues and stacks.

Traversing trees

Traversing a tree means to iterate through all the nodes of the tree. This is necessary in order to perform operations such as to add all the values in the tree or to find the largest values.
The three most common types of traversal are: inorder, preorder, postorder.

Preorder traversal

1. Visit root node
2. Visit all the nodes in the left subtree
3. Visit all the nodes in the right subtree

Inorder traversal

1. Visit all the nodes in the left subtree
2. Visit the root node
3. Visit all the nodes in the right subtree

Postorder traversal

1. Visit all the nodes in the left subtree
2. Visit all the nodes in the right subtree
3. Visit the root node

Binary trees

A binary tree is a type of tree in which each parent node can have at most two children.
There are different types of binary trees:

- Full binary tree: every parent node has either two or no children.
- Perfect binary tree: every parent node has exactly two child nodes and all the leaf nodes are at the same level.

- Complete binary tree: every level, except possibly the last, is completely filled, and all nodes are as far left as possible
- Degenerate or pathological tree: is the tree having a single child either left or right.
- Skewed binary tree: a degenerate tree in which the tree is either dominated by the left nodes (left-skewed) or the right nodes (right-skewed).
- Balanced binary tree: the difference between the left and the right subtree for each node is either 0 or 1.

Implementing a binary tree

In order to implement a binary tree in python, one can create a node class corresponding to each node of the binary tree.

INPUT

```
#Creating a Node class for our binary tree, including the three di
fferent traversals as methods

class Node:
    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data

    def traversePreOrder(self):
        print(self.data, end=' ')
        if self.left:
            self.left.traversePreOrder()
        if self.right:
            self.right.traversePreOrder()

    def traverseInOrder(self):
        if self.left:
            self.left.traverseInOrder()
        print(self.data, end=' ')
        if self.right:
            self.right.traverseInOrder()

    def traversePostOrder(self):
        if self.left:
            self.left.traversePostOrder()
        if self.right:
            self.right.traversePostOrder()
        print(self.data, end=' ‘)
```

```
#Creating a binary tree
root = Node(8)
root.left = Node(6)
root.right = Node(4)
root.left.left = Node(7)
root.left.left.right = Node(2)
root.right.left = Node(1)
root.right.right = Node(5)

#Printing our tree using pre order traversal
print("Pre order Traversal: ", end="")
root.traversePreOrder()
#Printing our tree using in order traversal
print("\nIn order Traversal: ", end="")
root.traverseInOrder()
#Printing our tree using post order traversal
print("\nPost order Traversal: ", end="")
root.traversePostOrder()
```

OUTPUT

```
Pre order Traversal: 8 6 7 2 4 1 5
In order Traversal: 7 2 6 8 1 4 5
Post order Traversal: 2 7 6 1 5 4 8
```

Binary trees using dictionaries

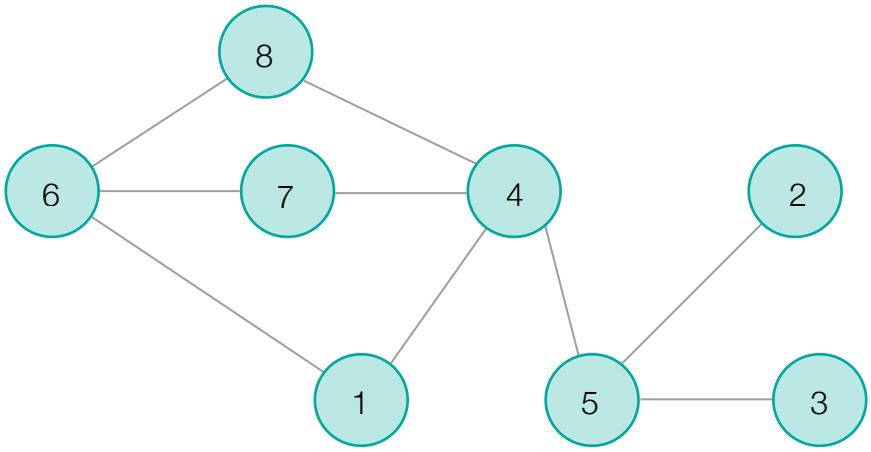
Another way to implement binary trees (and trees in general) is using dictionaries. Each key-value pair is a unique node pointing to its children and a list holds an ordered pair of left/right children. With a dict having ordered insertion, assume the first entry is the root.

INPUT

```
#Creating a binary tree using dictionaries
binaryTree = {
   "8": ["6", "4"],
   "6": ["7", None],
   "7": [None, "2"],
   "4": ["1", "5"],
}
print("Our tree is:", binaryTree)
OUTPUT

Our tree is: {'8': ['6', '4'], '6': ['7', None], '7': [None, '2'],
'4': ['1', '5']}
```

Graphs

Graphs are a type of non-linear data structures that consist of vertices connected by edges. Graphs can be directed or undirected and weighted or non-weighted. The following graph is an undirected and non-weighted graph.



Dictionaries and list make graphs implementation in Python easy, similarly to binary trees.

```
#Creating a graph using dictionaries
graph = {
    1:[4,6],
    2:[5],
    3:[5],
    4:[1,5],
    5:[3,4],
    6:[1,7,8],
    7:[4,6],
    8:[4,6]
}
```

A Graph class can be created in order to implement graph like the above. In the next piece of code, four methods inside the Graph class are created:

- `addEdges` : adds eggiest to the graph
- `showGraph:` shows all edges in the graph
- `findPath`: finds a path between two nodes
- `BFS`: Breath First Search, an algorithm for traversing trees or graphs.

The `findPath` method takes 3 arguments namely start, end and path which store the start and end nodes and the current path while the function recursively calls itself to update that path. Initially, the path is set to an empty list. Then, the start node is appended to it. If the start is the same to the end, the path is returned (which correspond to the start node).
This loop traverses all the neighbouring nodes of the start node and then recursively calls itself again until it finds a path from one of the neighbouring nodes to end node. If it finds a path, it returns such path.

Breath First Search is an important traversing algorithm and has many important applications including solving Rubik's cube, Computing shortest path (Dijkstra's algorithm) and Ford-Fulkerson's algorithm for computing maximum flows in a flow network.

A queue is used to implement this algorithm, by popping nodes from queue, adding the neighbours of that node in the queue and labelling them as visited so that the already traversed nodes are added in the queue again.

The `BFS` method creates a visited empty dictionary visited. The value of each node is set to false indicating that these nodes are not visited yet.
After this, a queue is created. The starting node is appended to it and marked as visited.
The while loop runs until the queue is empty. First, it pops the first node from the queue.
Then, it traverses all the neighbouring nodes of the popped node and if they are not already visited it marks them as visited and pushes them at the end of the queue. Finally, it prints the popped node. This allows to find the connected components in the graph.

```
#Creating a Graph class that allows to implement graphs using dict
ionaries, find a path between to nodes and find connected componen
ts

class Graph:

    graph ={}

    def addEdge(self,node,neighbour):
        if node not in self.graph:
            self.graph[node]=[neighbour]
        else:
            self.graph[node].append(neighbour)

    def showEdges(self):
        for node in self.graph:
            for neighbour in self.graph[node]:
                print("(",node,", ",neighbour,")")
```

```
    def findPath(self,start,end,path=[]):
        path = path + [start]
        if start==end:
            return path
        for node in self.graph[start]:
            if node not in path:
                newPath=self.findPath(node,end,path)
                if newPath:
                    return newPath
                return None

    def BFS(self,s):
        visited={}
        for i in self.graph:
            visited[i]=False
        queue=[]
        queue.append(s)
        visited[s]=True
        while len(queue)!=0:
            s=queue.pop(0)
            for node in self.graph[s]:
                if visited[node]!=True:
                    visited[node]=True
                    queue.append(node)
            print(s,end=" ")

#Re-creating our graph using this class
graph= Graph()
graph.addEdge('1', '4')
graph.addEdge('1', '6')
graph.addEdge('2', '5')
graph.addEdge('3', '5')
graph.addEdge('4', '1')
graph.addEdge('4', '5')
graph.addEdge('4', '7')
graph.addEdge('4', '8')
graph.addEdge('5', '2')
graph.addEdge('5', '3')
graph.addEdge('5', '4')
graph.addEdge('6', '1')
graph.addEdge('6', '7')
graph.addEdge('6', '8')
graph.addEdge('7', '4')
graph.addEdge('7', '6')
graph.addEdge('8', '4')
graph.addEdge('8', '6')
graph.showEdges()
```

OUTPUT

```
( 1 ,  4 )
( 1 ,  6 )
( 2 ,  5 )
( 3 ,  5 )
( 4 ,  1 )
( 4 ,  5 )
( 4 ,  7 )
( 4 ,  8 )
( 5 ,  2 )
( 5 ,  3 )
( 5 ,  4 )
( 6 ,  1 )
( 6 ,  7 )
( 6 ,  8 )
( 7 ,  4 )
( 7 ,  6 )
( 8 ,  4 )
( 8 ,  6 )
```

INPUT

```
#Finding a path between the nodes 8 and 6
print("A path between the nodes 8 and 6 is:", graph.findPath('8','
6'))
```

OUTPUT

```
A path between the nodes 8 and 6 is: ['8', '4', '1', ‘6']
```

INPUT

```
#Travesing the nodes using BFS starting from the node 1
graph.BFS(‘1')
```

OUTPUT

```
1 4 6 5 7 8 2 3
```