Data structures and algorithm

Array Data Structure

Arrays are similar to list. Only difference is
The access. Once an array is created, its size cannot be changed.
List access in sequentially and Arrays are
Can be accessed by random.

The entire contents of an array are identified by a single name. Individual elements within the array can be accessed directly by specifying an integer subscript or index value, which indicates an offset from the start of the array

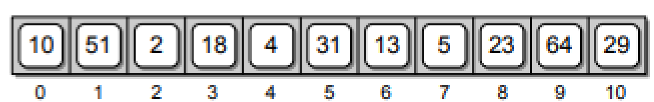


Figure 2.1: A sample 1-D array consisting of 11 elements.

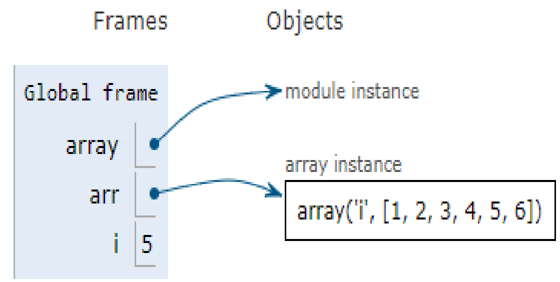
Code	C Type	Python Type	Min bytes
`'b'`	signed char	int	1
`'B'`	unsigned char	int	1
`'u'`	Py_UNICODE	Unicode	2
`'h'`	signed short	int	2
`'H'`	unsigned short	int	2
`'i'`	signed int	int	2
`'I'`	unsigned int	int	2
`'l'`	signed long	int	4
`'L'`	unsigned long	int	4
`'f'`	float	float	4
`'d'`	double	float	8

Write code in Python 3.6 (drag lower right corner to resize code editor)

```python
# importing 'array' module
import array

# initializing array
arr = array.array('i', [1, 2, 3, 4, 5])

# printing original array
print ("The new created array is : ",end="")
for i in range (0, 5):
    print (arr[i], end=" ")

# using append() to insert new value at end
arr.append(6);

# printing appended array
print ("\nThe appended array is : ", end="")
for i in range (0, 6):
    print (arr[i], end=" ")


```

Print output (drag lower right corner to resize)

```
The new created array is : 1 2 3 4 5
The appended array is : 1 2 3 4 5 6
```

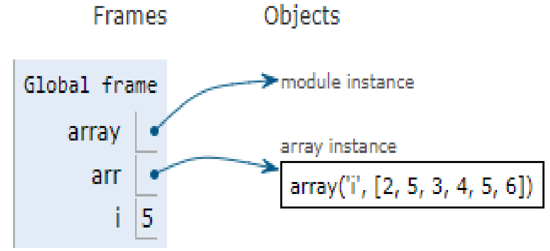


Write code in Python 3.6 ▾ (drag lower right corner to resize code editor)

```python

# importing 'array' module
import array

# initializing array
arr = array.array('i', [1, 2, 3, 4, 5])     # initialize array with in

# printing original array
print ("The new created array is : ",end="")
for i in range (0, 5):
    print (arr[i], end=" ")

# using append() to insert new value at end
arr.append(6);

# printing appended array
print ("\nThe appended array is : ", end="")
for i in range (0, 6):
    print (arr[i], end=" ")

# using insert() to insert value at specific position
# inserts 5 at 2nd position
arr.insert(2, 5)

# printing array after insertion
print ("\nThe array after insertion is : ", end="")
for i in range (0, 7):
    print (arr[i], end=" ")

arr.remove(1)

# deleting a  value from array
print ("\nThe array after deletion is : ", end="")
for i in range (0, 6):
    print (arr[i], end=" ")


```

Print output (drag lower right corner to resize)

```
The new created array is : 1 2 3 4 5
The appended array is : 1 2 3 4 5 6
The array after insertion is : 1 2 5 3 4 5 6
The array after deletion is : 2 5 3 4 5 6
```



line that has just executed

Disadvantages of Array

- **Fixed size**: The size of the array is static (specify the array size before using it, this can be overcome using Dynamic Arrays).
- **One block allocation**: To allocate the array itself at the beginning, sometimes it may not be possible to get the memory for the complete array (if the array size is big).
- **Complex position-based insertion**: To insert an element at a given position, we may need to shift the existing elements. This will create a position for us to insert the new element at the desired position. If the position at which we want to add an element is at the beginning, then the shifting operation is more expensive

Linked list

- Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location(sharing border);
- the elements are linked using pointers.
- **Why Linked List?**
  Arrays can be used to store linear data of similar types, but arrays have the following limitations.
  1) The size of the arrays is fixed:
  2) Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

Representation

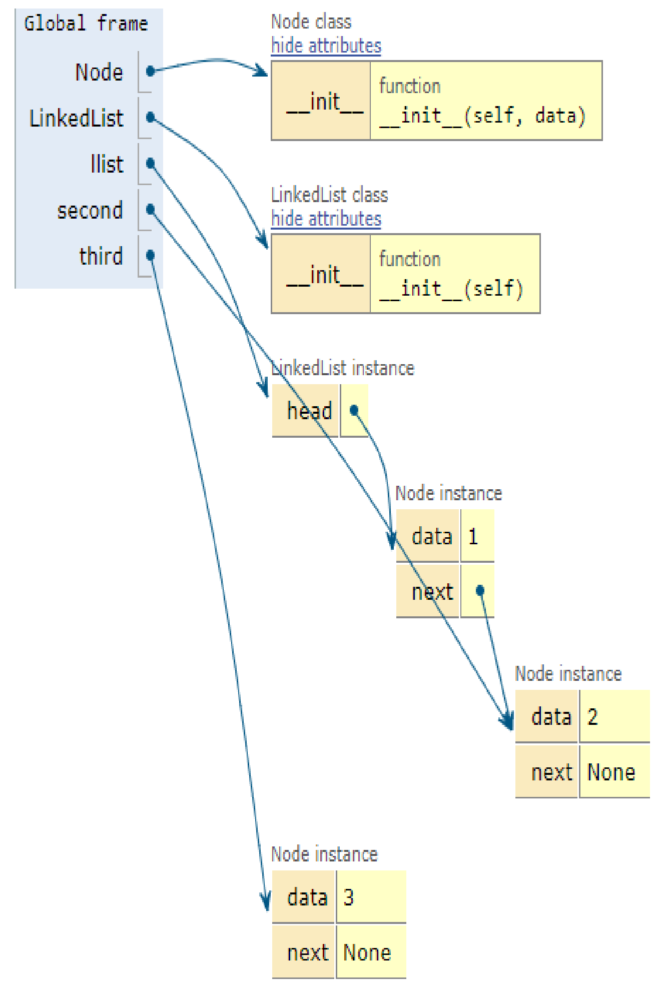
A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head is NULL.

Each node in a list consists of at least two parts:

1) data

2) Pointer (Or Reference) to the next node

```python
# A simple Python program to introduce a linked list

# Node class
class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data  # Assign data
        self.next = None  # Initialize next as null

    # Linked List class contains a Node object

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

# Code execution starts here
if __name__=='__main__':

    # Start with the empty list
    llist = LinkedList()

    llist.head = Node(1)
    second = Node(2)
    third = Node(3)

llist.head.next = second

second.next = third
```



Print the data in linked list

```python
    # This function prints contents of linked list
    # starting from head
    def printList(self):
        temp = self.head
        while (temp):
            print (temp.data,)
            temp = temp.next

# Code execution starts here
if __name__=='__main__':

    # Start with the empty list
    llist = LinkedList()

    llist.head = Node(1)
    second = Node(2)
    third = Node(3)

llist.head.next = second

second.next = third


llist.printList()
```

Examples of linked list–Adding Front

```python
class SLinkedList:
    def __init__(self):
        self.headval = None

# Print the linked list
    def listprint(self):
        printval = self.headval
        while printval is not None:
            print (printval.dataval)
            printval = printval.nextval
    def AtBegining(self,newdata):
        NewNode = Node(newdata)

# Update the new nodes next val to existing node
        NewNode.nextval = self.headval
        self.headval = NewNode

list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")
```

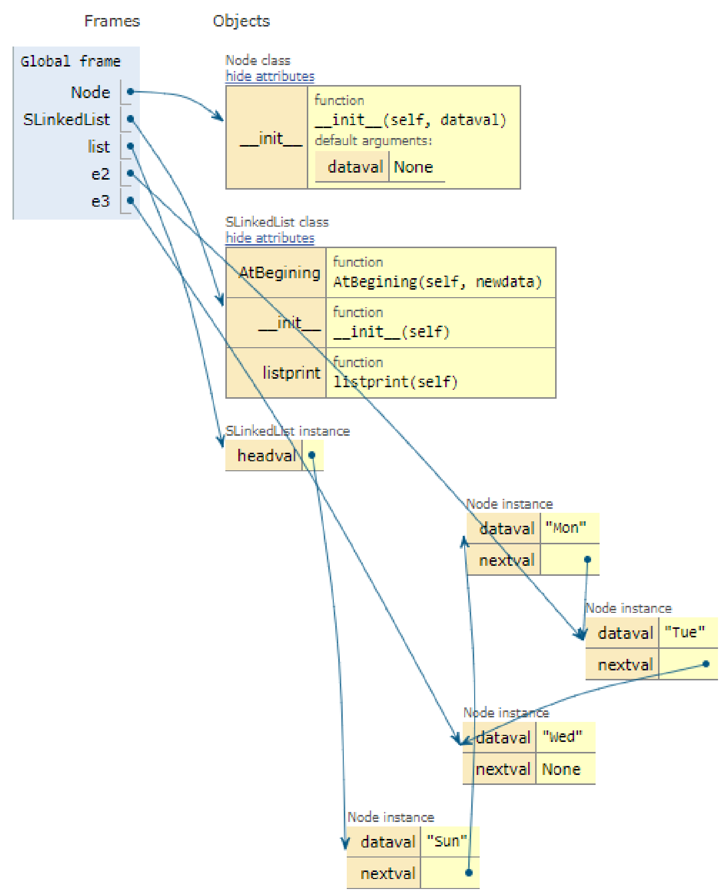
line that has just executed
next line to execute

<< First | < Back | Done running (50 steps) | Forward > | Last >>

Help improve this tool by clicking whenever you learn something:

I just cleared up a misunderstanding! | I just fixed a bug in my code!

```
Mon
Tue
Wed
```



Adding the data at End

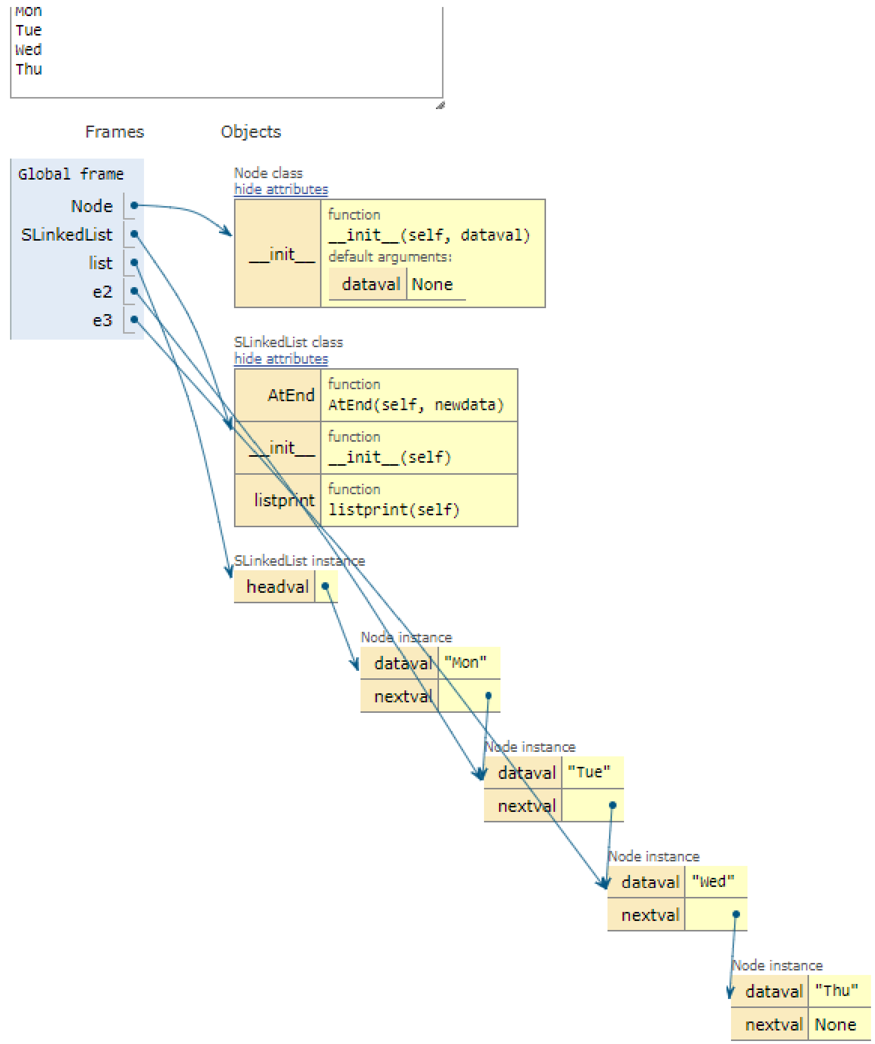
```python
        laste.nextval=NewNode

# Print the linked list
    def listprint(self):
        printval = self.headval
        while printval is not None:
            print (printval.dataval)
            printval = printval.nextval


list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")

list.headval.nextval = e2
e2.nextval = e3

list.AtEnd("Thu")

list.listprint()

```

line that has just executed
next line to execute

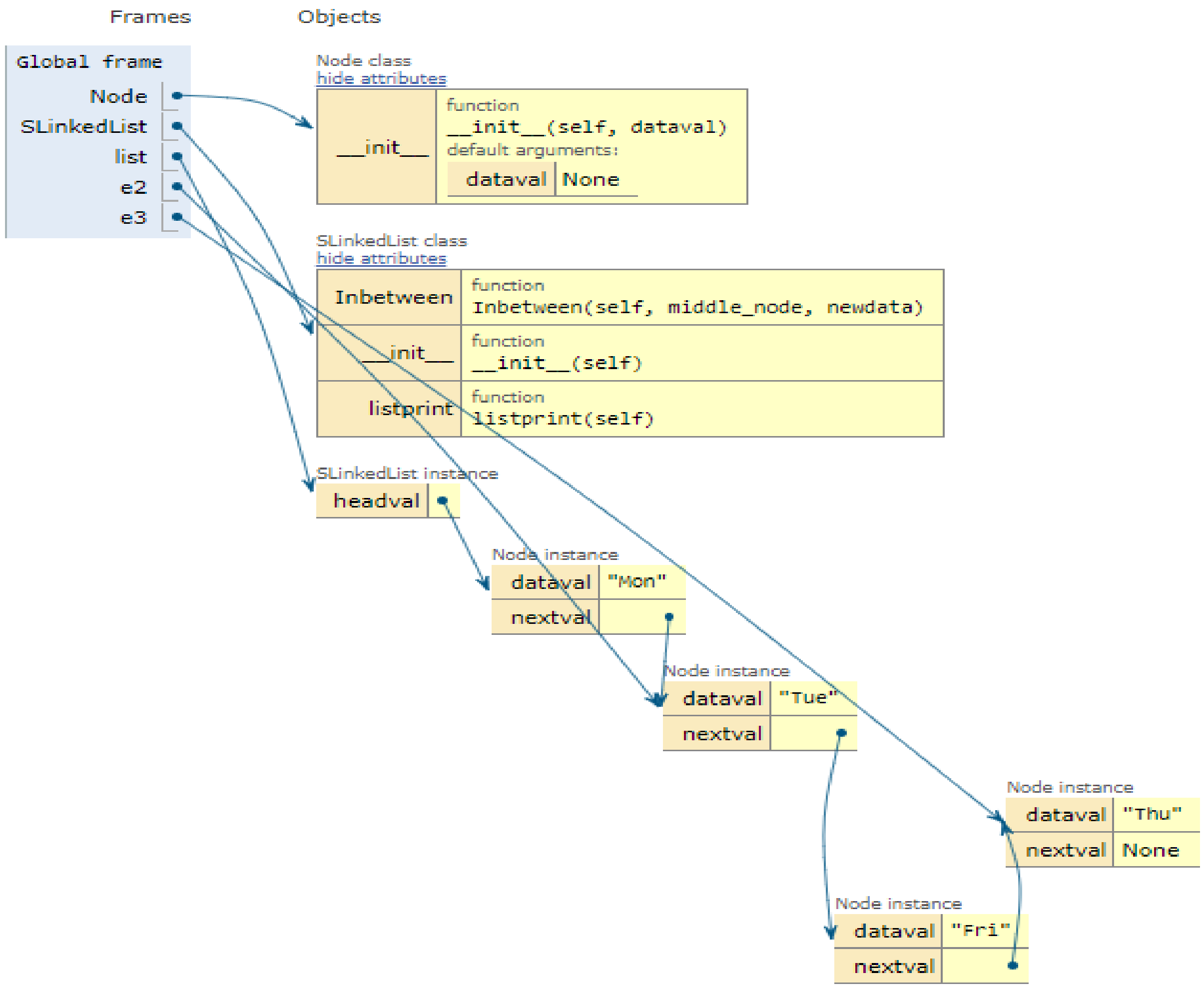
<< First | < Back | Done running (56 steps) | Forward > | Last >>

Help improve this tool by clicking whenever you learn something:

I just cleared up a misunderstanding! | I just fixed a bug in my code!



Inserting data between nodes



Stacks

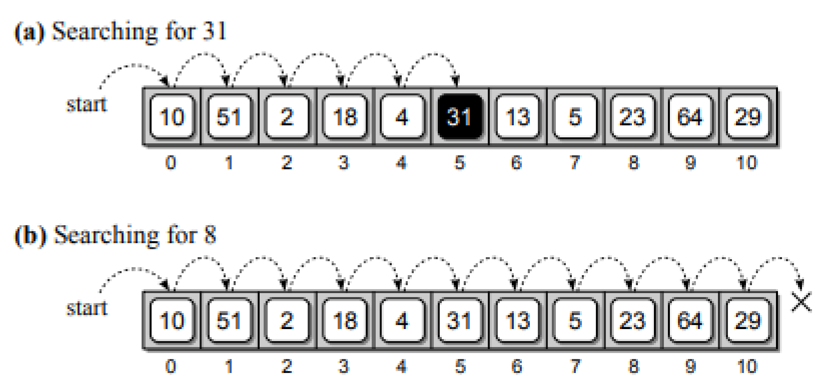
- Stack is a linear data structure which follows a particular order in which the operations are performed.
- The order may be LIFO(Last In First Out) or FILO(First In Last Out).

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns top element of stack.
- **isEmpty:** Returns true if stack is empty, else false

Searching

- Searching is the process of selecting particular information from a collection of data based on specific criteria.
- <u>The Linear Search:</u>
- The simplest solution to the sequence search problem is the sequential or linear search algorithm.

```
if key in theArray :
   print( "The key is in the array." )
else :
   print( "The key is not in the array." )
```

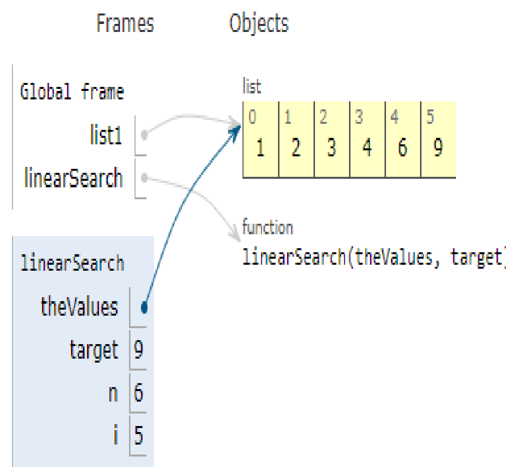


Write code in Python 3.6 (drag lower right corner to resize code editor)

```
list1=[1,2,3,4,6,9]

def linearSearch( theValues, target ) :
    n = len( theValues )
    for i in range( n ) :
        if theValues[i] == target:
            print('value found')
            return True
        elif theValues[i] > target :
            return False

    return False



print(linearSearch( list1, 9 ))


```

Print output (drag lower right corner to resize)

```
value found
```

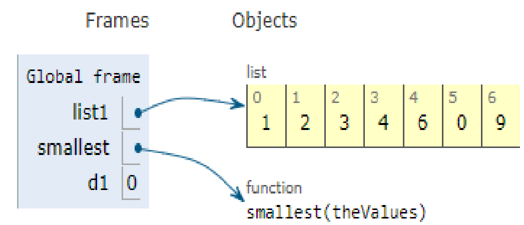


Finding the smallest value

Write code in Python 3.6

(drag lower right corner to resize code editor)

```python
list1=[1,2,3,4,6,0,9]

#find the least value in :

def smallest( theValues) :
    n = len( theValues )
    smallest=theValues[0]
    # Determine if any other item in the sequence is smaller.

    for i in range(1, n ) :
        if theValues[i] < smallest:
            smallest=theValues[i]

    return smallest


# find the smallest value in given list

d1=smallest(list1)
```



The Binary Search

- The binary search algorithm works in a similar fashion to the process described above and can be applied to a sorted sequence.
- The algorithm starts by examining the middle item of the sorted sequence, resulting in one of three possible conditions:
- the middle item is the target value, the target value is less than the middle item, or the target is larger than the middle item

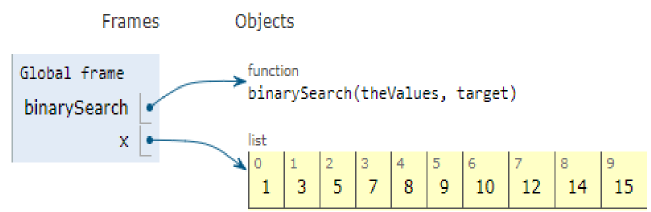
Write code in Python 3.6

(drag lower right corner to resize code editor)

```python
def binarySearch( theValues, target ) :
    low = 0
    high = len(theValues) - 1

# Repeatedly subdivide the sequence in half until the target is found
    while low <= high :
            # Find the midpoint of the sequence.
        mid = (high + low) // 2

        if theValues[mid] == target :
            return True

        elif target < theValues[mid] :
            high = mid - 1

        else :
            low = mid + 1

    return False


x=[1,3,5,7,8,9,10,12,14,15]

print(binarySearch(x, 13))
```

Print output (drag lower right corner to resize)

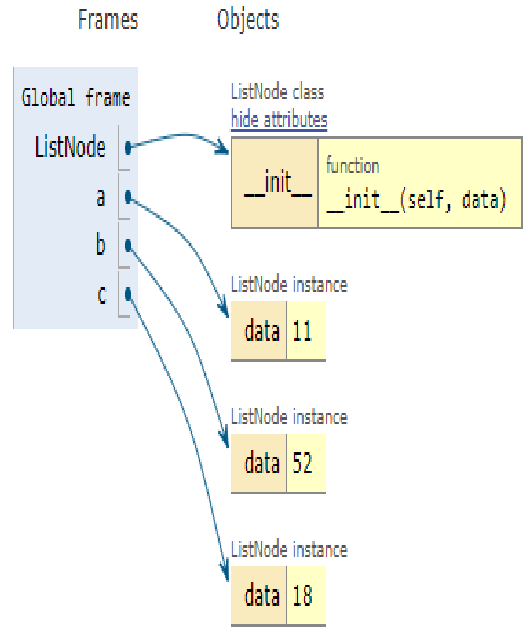
```
False
```



Linked Structures

Write code in Python 3.6 (drag lower right corner to resize code editor)

```python
class ListNode :
    def __init__( self, data ) :

        self.data = data

a = ListNode( 11 )
b = ListNode( 52 )
c = ListNode( 18 )

```



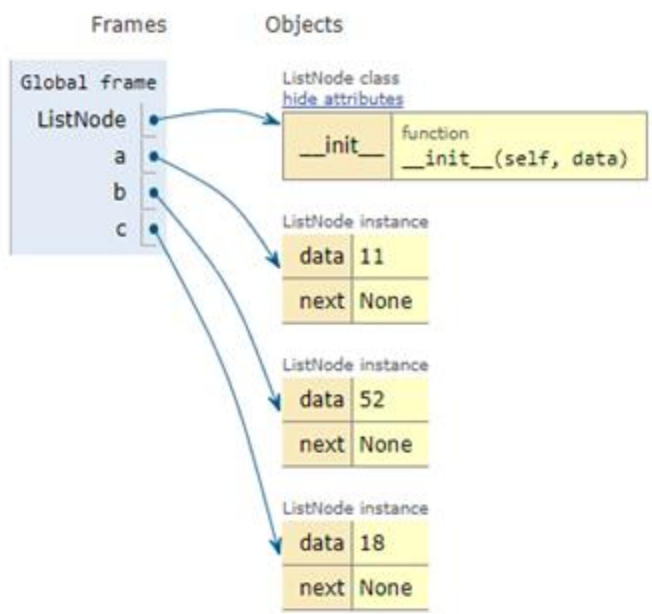
Linked list

- Now, suppose we add a second data field to the ListNode class:

Write code in Python 3.6

```
class ListNode :
    def __init__( self, data ) :

        self.data = data
        self.next = None

a = ListNode( 11 )
b = ListNode( 52 )
c = ListNode( 18 )
```

(drag lower right corner to resize code editor)

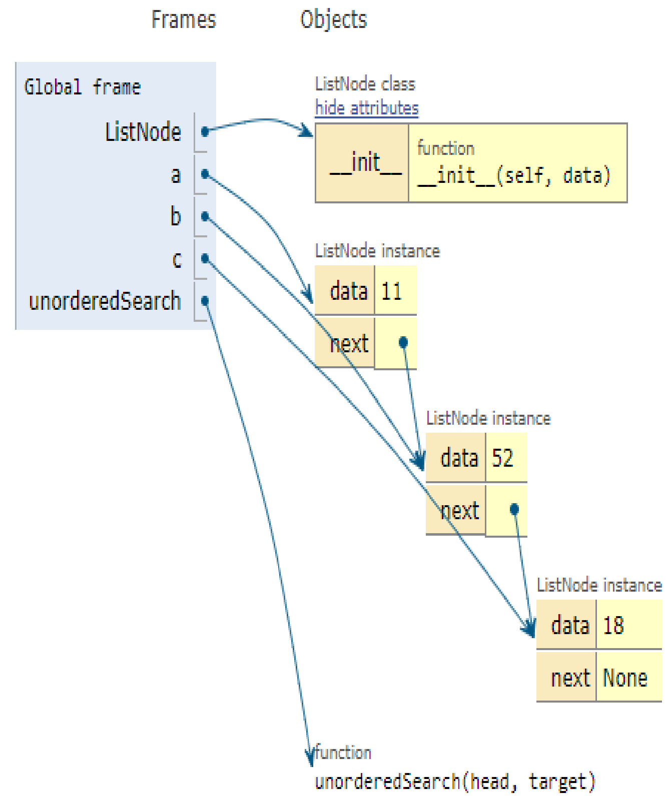


Write code in Python 3.6 (drag lower right corner to resize code editor)

```python
class ListNode :
    def __init__( self, data ) :

        self.data = data
        self.next = None


a = ListNode( 11 )
b = ListNode( 52 )
c = ListNode( 18 )


a.next=b
b.next=c

print( a.data )
print( a.next.data )
print( a.next.next.data )

def unorderedSearch( head, target ):
    curNode = head
    while curNode is not None and curNode.data != target :
        curNode= curNode.next
    return curNode is not None



print(unorderedSearch(a, 10 ))
```

Print output (drag lower right corner to resize)

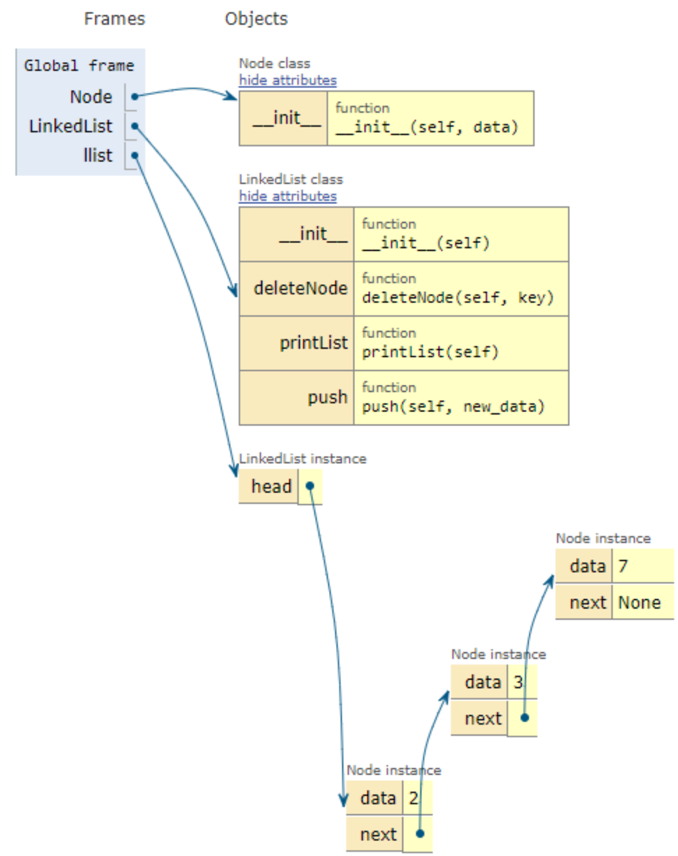
```
11
52
18
False
```



How to remove the node from linked list

```python
# Node class
class Node:

    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Given a reference to the head of a list and a key,
    # delete the first occurence of key in linked list
    def deleteNode(self, key):

        # Store head node
        temp = self.head

        # If head node itself holds the key to be deleted
        if (temp is not None):
            if (temp.data == key):
                self.head = temp.next
                temp = None
                return

        # Search for the key to be deleted, keep track of the
        # previous node as we need to change 'prev.next'
        while(temp is not None):
            if temp.data == key:
                break
            prev = temp
            temp = temp.next
```

```
Linked List after Deletion of 1:
 2
 3
 7
```



```python
        # if key was not present in linked list
        if(temp == None):
            return

        # Unlink the node from linked list
        prev.next = temp.next

        temp = None


    # Utility function to print the linked LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print (" %d" %(temp.data), )
            temp = temp.next


# Driver program
llist = LinkedList()
llist.push(7)
llist.push(1)
llist.push(3)
llist.push(2)

print ("Created Linked List: ")
llist.printList()
llist.deleteNode(1)
print ("\nLinked List after Deletion of 1:")
llist.printList()
```

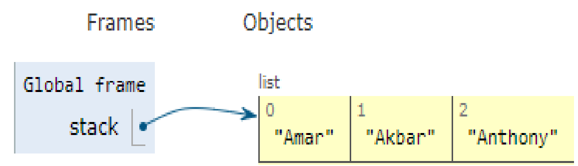
Stacks

Write code in Python 3.6 (drag lower right corner to resize code editor)

```python
# Python code to demonstrate Implementing
# stack using list
stack = ["Amar", "Akbar", "Anthony"]
stack.append("Ram")
stack.append("Iqbal")
print(stack)
print(stack.pop())
print(stack)
print(stack.pop())
print(stack)
```

Print output (drag lower right corner to resize)

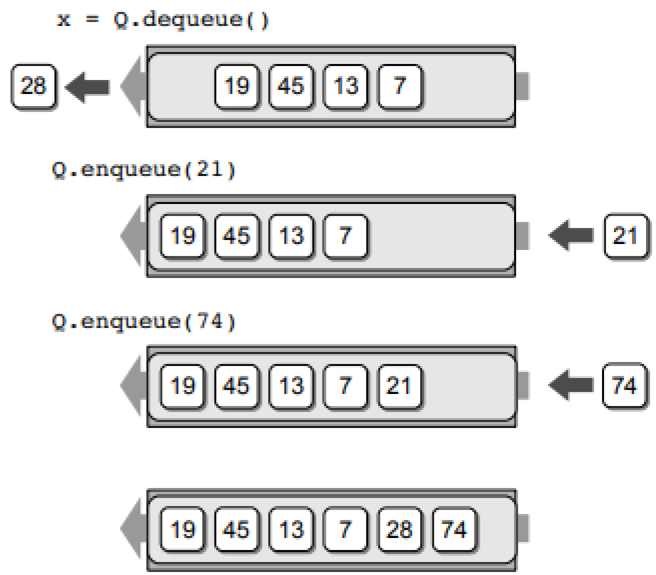
```
['Amar', 'Akbar', 'Anthony', 'Ram', 'Iqbal']
Iqbal
['Amar', 'Akbar', 'Anthony', 'Ram']
Ram
['Amar', 'Akbar', 'Anthony']
```



Queues

- A queue is a specialized list with a limited number of operations in which items can only be added to one end and removed from the other. A queue is also known as a first-in, first-out (FIFO).
- Queue(): Creates a new empty queue, which is a queue containing no items.
- isEmpty(): Returns a boolean value indicating whether the queue is empty.
- length (): Returns the number of items currently in the queue.
- enqueue(item): Adds the given item to the back of the queue.
- dequeue(): Removes and returns the front item from the queue. An item cannot be dequeued from an empty queue.


front
28
19
45
13
7
back


x = Q.dequeue()
28
19
45
13
7
Q.enqueue(21)
19
45
13
7
21
Q.enqueue(74)
19
45
13
7
21
74
19
45
13
7
28
74

Write code in Python 3.6 (drag lower right corner to resize code editor)

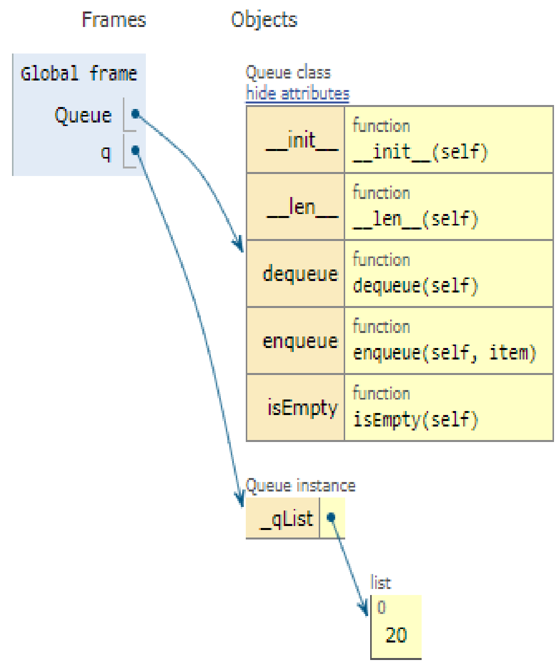
```
class Queue :
    def __init__( self ):
        self._qList = list()

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

    def __len__( self ):
        return len( self._qList )

    def enqueue( self, item ):
         self._qList.append( item )

    def dequeue( self ):
        assert not self.isEmpty()
        return self._qList.pop( 0 )

q=Queue()
print(q.isEmpty())
q.enqueue(10)
q.enqueue(20)
q.dequeue()
```

line that has just executed

next line to execute

Print output (drag lower right corner to resize)

```
True
```



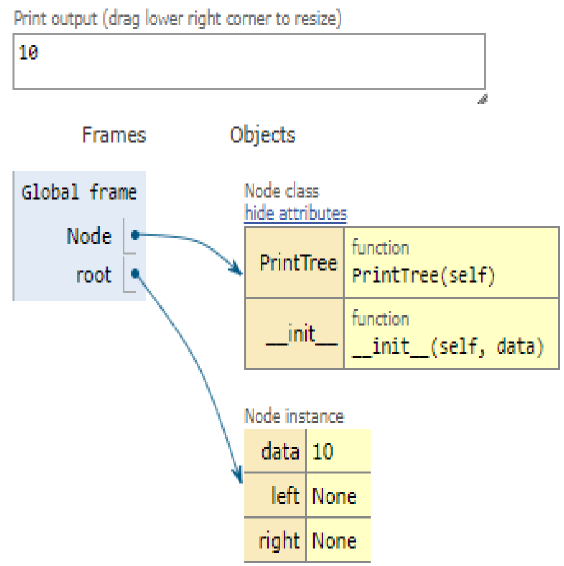
Trees

- A tree structure consists of nodes and edges that organize data in a hierarchical fashion.
- The relationships between data elements in a tree are similar to those of a family tree: "child," "parent," "ancestor," etc.
- The data elements are stored in nodes and pairs of nodes are connected by edges.
- The edges represent the relationship between the nodes that are linked with arrows or directed edges to form a hierarchical structure resembling an upside-down tree complete with branches, leaves, and even a root.

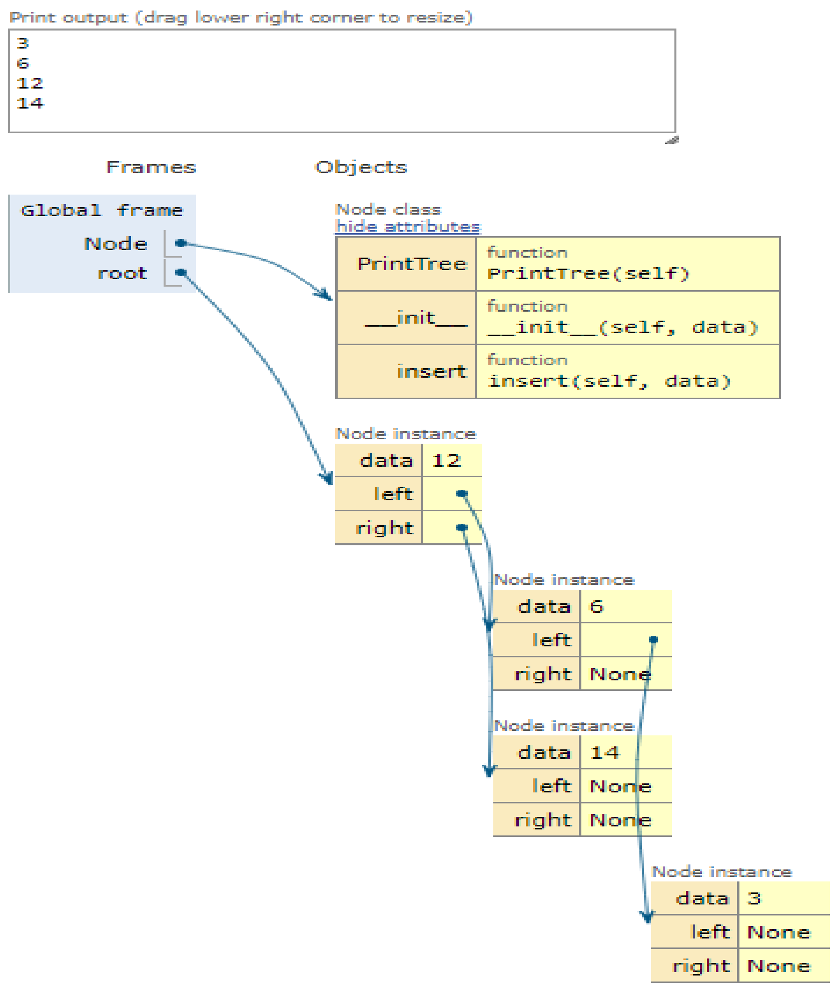
Sample example of tree

Write code in Python 3.6 (drag lower right corner to resize code editor)

```python
class Node:

    def __init__(self, data):

        self.left = None
        self.right = None
        self.data = data


    def PrintTree(self):
        print(self.data)

root = Node(10)

root.PrintTree()
```



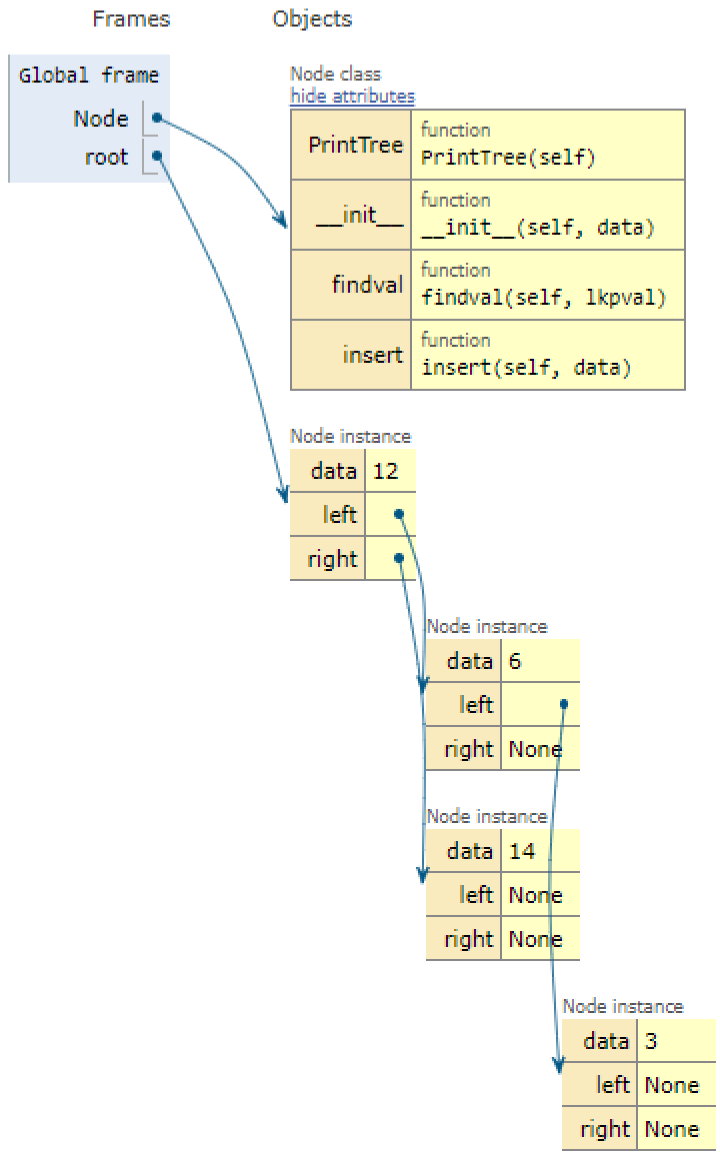
Example



Print output (drag lower right corner to resize)
3
6
12
14
Frames
Objects
Global frame
Node
root
Node class
hide attributes
PrintTree
function
PrintTree(self)
__init__
function
__init__(self, data)
insert
function
insert(self, data)
Node instance
data 12
left
right
Node instance
data 6
left
right None
Node instance
data 14
left None
right None
Node instance
data 3
left None
right None

Print output (drag lower right corner to resize)

```
7 Not Found
14 is found
None
```



srxh.txt

Python – Divide and conquer Binary search

- In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently.
- When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible.
- Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.

To be continued