Computer Engineering Department

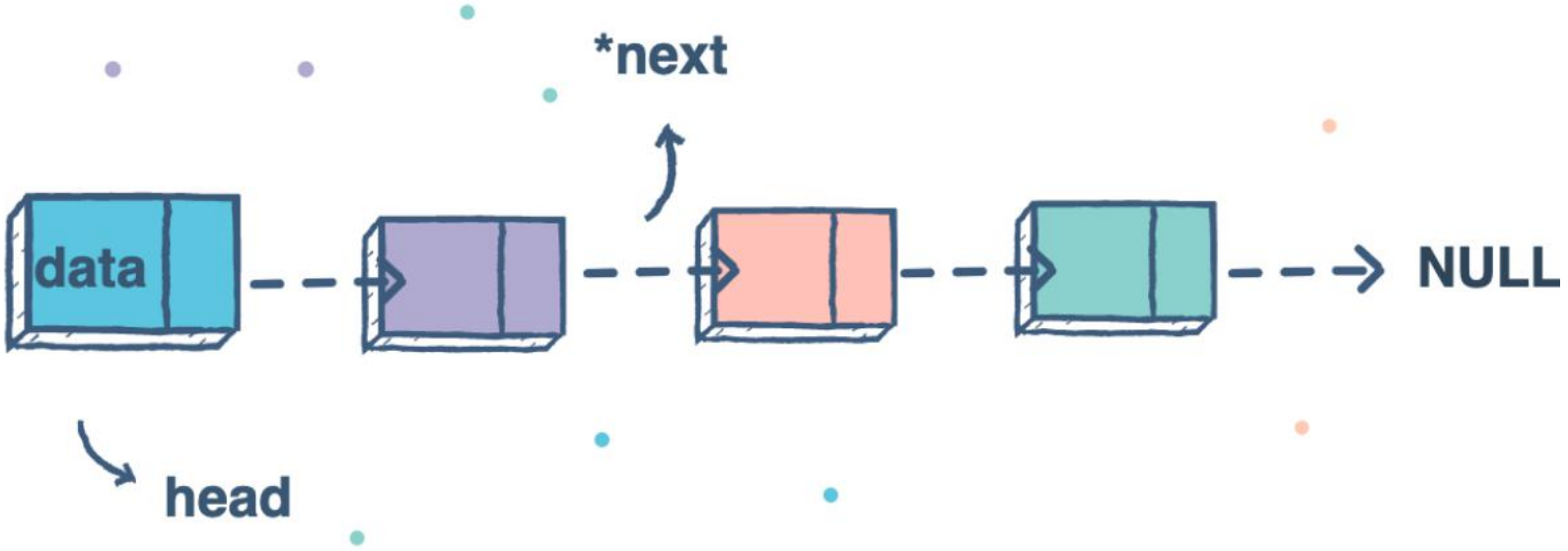
Advanced Python Programming(CE0620)

Data Strucutren in Python

By
Prof. Bhumi Shah

Linked List

- A linked list is a data structure made of a chain of node objects.
- A linked list is a sequence of data elements, which are connected together via links. Each data element contains a connection to another data element in form of a pointer.
- Linked lists, do not store data at contiguous memory locations.
- For each item in the memory location, linked list stores value of the item and the reference or pointer to the next item. One pair of the linked list item and the reference to next item constitutes a node.



Linked List

Original Python does not ship with a built-in linked list data structure

Start with a single node

we make a Node class that holds some data and a single pointer next, that will be used to point to the next Node type object in the Linked List.

```
# A single node of a singly linked list
class Node:
  # constructor
  def __init__(self, data, next=None):
    self.data = data
    self.next = next

# Creating a single node
first = Node(3)
print(first.data)
```

Join nodes to get a linked list

join multiple single nodes containing data using the next pointers, and have a single head pointer pointing to a complete instance of a Linked List.

create a LinkedList class with a single head pointer:

```
# A single node of a singly linked list
class Node:
  # constructor
  def __init__(self, data = None, next=None):
    self.data = data
    self.next = next
```

```
# A Linked List class with a single head node
class LinkedList:
  def __init__(self):
    self.head = None

# Linked List with a single node
LL = LinkedList()
LL.head = Node(3)
print(LL.head.data)
```

Add required methods to the LinkedList class

```
# A single node of a singly linked list
class Node:
  # constructor
  def __init__(self, data = None, next=None):
    self.data = data
    self.next = next


# A Linked List class with a single head node
class LinkedList:
  def __init__(self):
    self.head = None
```

```
  # insertion method for the linked list
  def insert(self, data):
    newNode = Node(data)
    if(self.head):
      current = self.head
      while(current.next):
        current = current.next
      current.next = newNode
    else:
      self.head = newNode

  # print method for the linked list
  def printLL(self):
    current = self.head
    while(current):
      print(current.data)
      current = current.next
```

```
# Singly Linked List with insertion and print methods

LL = LinkedList()
LL.insert(3)
LL.insert(4)
LL.insert(5)
LL.printLL()
```

Stacks

- A stack is a linear data structure that stores items in a Last-In First-Out (LIFO) manner.
- In stack, a new element is added at one end and an element is removed from that end only.

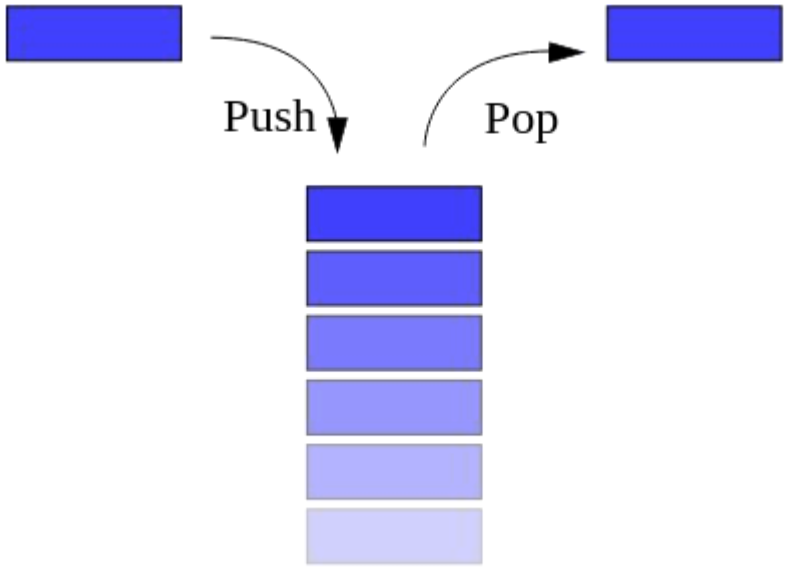
Operations on Stack

1. Push

2. Pop

3. Search

4. peep



stack.py Module

```
class Stack:
    def __init__(self):
        self.items = []

    def isempty(self):
        return self.items == []

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

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

def peep(self):
    n=len(self.items)
    return self.items[n-1]

def search(self,ele):
    if self.isempty():
        return -1
    else:
        n=self.items.index(ele)
        return len(self.items)-n
def display(self):
   return self.items
```

stackoperation.py

```
from stack import Stack
s=Stack()
choice=0
while choice<5:
    print("------------------")
    print("Stack Operations")
    print("1: Push()")
    print("2: Pop()")
    print("3: Peep()")
    print("4: Search()")
    print("5: Exit")
    print("------------------")
    choice=int(input("enter choice:"))
 if choice==1:
        el=int(input("Enter element:"))
        s.push(el)
       elif choice==2:
        el=s.pop()
        print("Popped Element:",el)
         elif choice==3:
        el=s.peep()
        print("Top Element:",el)
elif choice==4:
        el=int(input("Enter element:"))
        pos=s.search(el)
        if pos==-1 :
            print("Stack empty")
        else:
            print("Elemet is at postion:",pos)
else:
        break

    print("Stack Elements:",s.display())
```

Queue

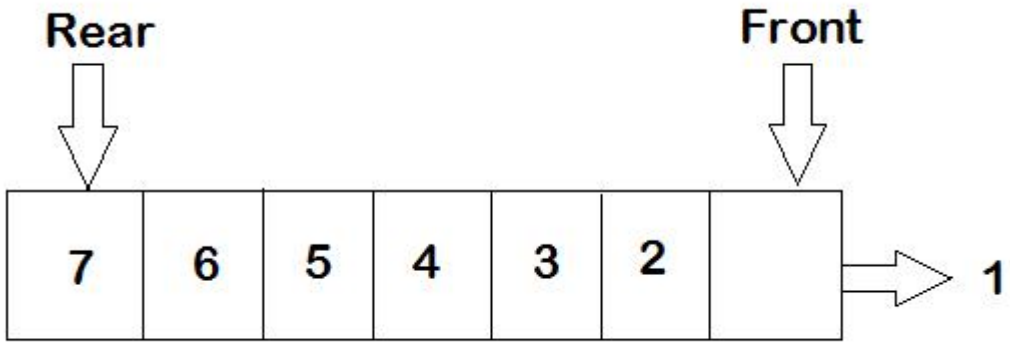
- A queue is a linear type of data structure used to store the data in a sequentially.
- The concept of queue is based on the FIFO, which means "First in First Out".
- The queue has the two ends front and rear. The next element is inserted from the rear end and removed from the front end.

Operations on Queue:

1. enqueue

2. dequeue

3. Search



queue1.py Module

```
class Queue:
    def __init__(self):
        self.qu = []

    def isempty(self):
        return self.qu == []

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

    def dequeue(self):
        if self.isempty():
            return -1
        else:
            return self.qu.pop(0)
```

```
def search(self,ele):
        if self.isempty():
            return -1
        else:
            n=self.qu.index(ele)
            return n+1
def display(self):
        return self.qu
```

queuprog.py

```
from queue1 import Queue
q=Queue()
choice=0
while choice<4:
    print("------------------")
    print("Queue Operations")
    print("1: Add Element")
    print("2: Delete Element")
    print("3: Search Element")
    print("4: Exit")
    print("------------------")

choice=int(input("enter choice:"))
if choice==1:
        el=int(input("Enter element:"))
        q.enqueue(el)


    elif choice==2:
        el=q.dequeue()
        if el==-1:
            print("Queue is empty")
        else:
            print("Removed Element:",el)
 elif choice==3:
        el=int(input("Enter element:"))
        pos=q.search(el)
        if pos==-1 :
            print("Queue empty")
        else:
            print("Elemet is at postion:",pos)
    else:
        break

    print("Queue Elements:",q.display())
```

Algorithms

- Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output.
- Algorithms are generally created independent of underlying languages
- From the data structure point of view, following are some important categories of algorithms:
  - Search − Algorithm to search an item in a data structure.
  - Sort − Algorithm to sort items in a certain order.
  - Insert − Algorithm to insert item in a data structure.
  - Update − Algorithm to update an existing item in a data structure.
  - Delete − Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

- **Unambiguous −** Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

Example:Algorithm to add two numbers

step 1 − START

step 2 − declare three integers a, b & c

step 3 − define values of a & b

step 4 − add values of a & b

step 5 − store output of step 4 to c

step 6 − print c

step 7 − STOP

Home Work

Write an algorith for following with its time coplexity,and implement a code.

1. Linear Search
2. Binary Serch
3. Merge Sort
4. Selection Sort