Principles of Computer Science II

Abstract Data Types

Ioannis Chatzigiannakis

Sapienza University of Rome

Lecture 11

Queues

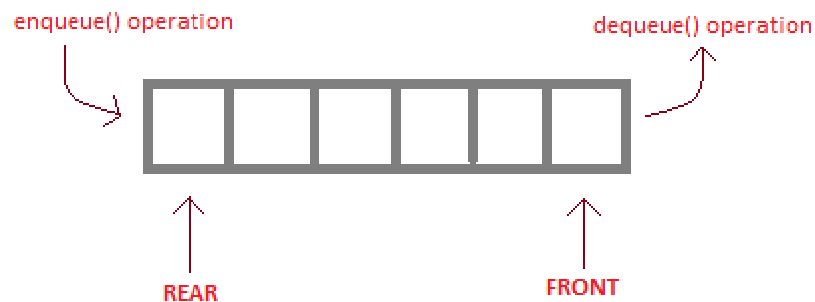
Queue is also an abstract data type or a linear data structure, in which

- the first element is inserted from one end called REAR (also called tail),
- and the deletion of existing element takes place from the other end called as FRONT(also called head).

This makes queue as FIFO(First in First Out) data structure, which means that element inserted first will also be removed first.

- The process to add an element into queue is called Enqueue.
- the process of removal of an element from queue is called Dequeue.

Queues



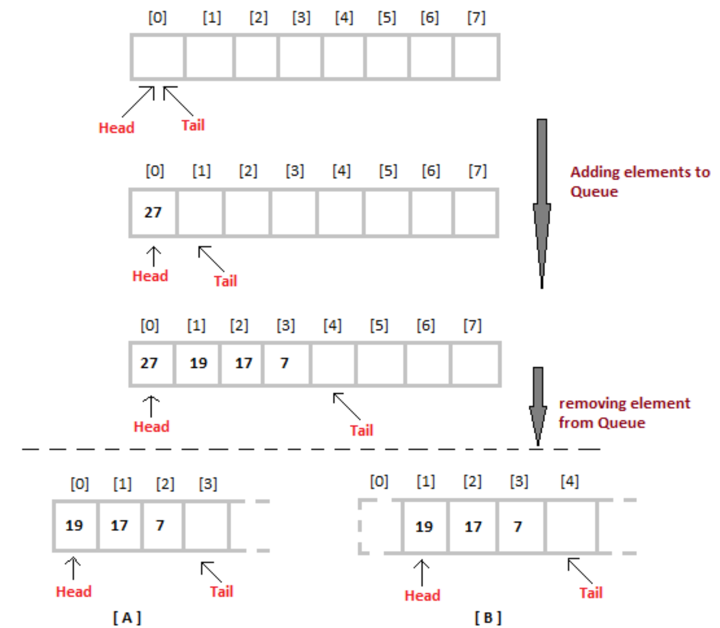
Basic features of Queues

1. Like Stack, Queue is also an ordered list of elements of similar data types.
2. Queue is a FIFO (First in First Out) structure.
3. Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
4. peek() function is oftenly used to return the value of first element without dequeuing it.

Implementation of Queues

- Queue can be implemented using an Array, Stack or Linked List.
- The easiest way of implementing a queue is by using an Array.
- Initially the head(FRONT) and the tail(REAR) of the queue points at the first index of the array (starting the index of array from 0).
- As we add elements to the queue, the tail keeps on moving ahead, always pointing to the position where the next element will be inserted, while the head remains at the first index.

Queues: An Example



Queues Code: Initialization

```
1 class Error(Exception):
2     """Base class for exceptions in this module."""
3     pass
4
5 class QueueError(Error):
6     def __init__(self, expression, message):
7         self.expression = expression
8         self.message = message
9
10 class Queue(object):
11     def __init__(self):
12         self.content = []
13
14     def size(self):
15         return len(self.content)
16
17     def isEmpty(self):
18         return not bool(self.content)
```

Queues: Algorithm for ENQUEUE operation

1. Check if the queue is full or not.
2. If the queue is full, then raise overflow error and exit the program.
3. If the queue is not full, then increment the tail and add the element.

```
1     def enqueue(self, value):
2         return self.content.append(value)
```

Queues: Algorithm for DEQUEUE operation

1. Check if the queue is empty or not.
2. If the queue is empty, then raise underflow error and exit the program.
3. If the queue is not empty, then return the element at the head and increment the head.

```
1   def dequeue(self):
2       if self.content:
3           return self.content.pop()
4       else:
5           raise QueueError("Queue is Empty")
```

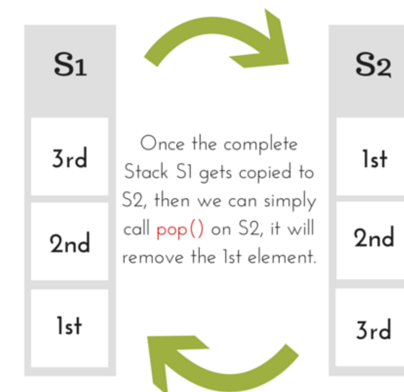
Queues: Help function

```
1   def __repr__(self):
2       if self.content:
3           return '{}'.format(self.content)
4       else:
5           return "Queue empty!"
```

Queues: Testing

```
1 if __name__ == '__main__':
2     queue = Queue()
3     print("Is the queue empty? ", queue.isEmpty())
4     print("Adding 0 to 10 in the queue...")
5     for i in range(10):
6         queue.enqueue(i)
7     print("Queue size: ", queue.size())
8     print("Queue peek : ", queue.peek())
9     print("Dequeue...", queue.dequeue())
10    print("Queue peek: ", queue.peek())
11    print("Is the queue empty? ", queue.isEmpty())
12
13    print("Printing the queue...")
14    print(queue)
```

Queues implemented with Stacks



Introduction to Linked Lists

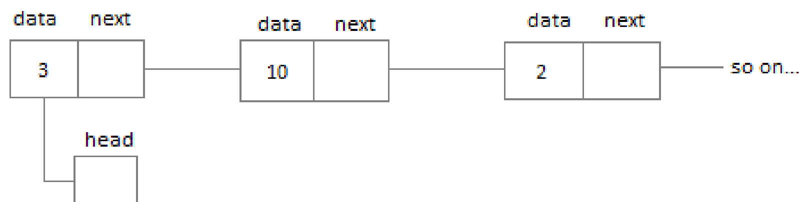
Linked List is a linear data structure and it is very common data structure which consists of group of nodes in a sequence which is divided in two parts. Each node consists of its own data and the address of the next node and forms a chain. Linked Lists are used to create trees and graphs.

Basic features of Linked Lists

1. They are a dynamic in nature which allocates the memory when required.
2. Insertion and deletion operations can be easily implemented.
3. Stacks and queues can be easily executed.
4. Linked List reduces the access time.
5. Pointers require extra memory for storage.
6. No element can be accessed randomly; it has to access each node sequentially.
7. Reverse Traversing is difficult in linked list.
8. Linked lists let you insert elements at the beginning and end of the list.

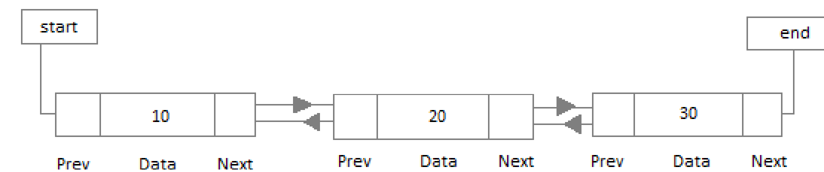
Linear Linked Lists

Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in sequence of nodes. The operations we can perform on singly linked lists are insertion, deletion and traversal.



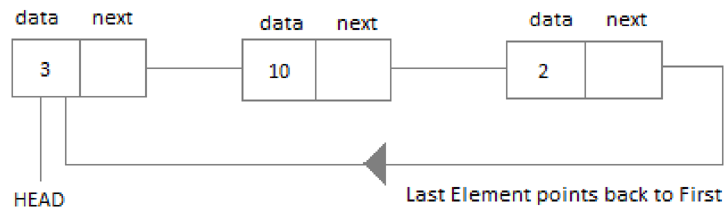
Double Linked Lists

In a doubly linked list, each node contains two links the first link points to the previous node and the next link points to the next node in the sequence.



Circular Linked Lists

In the circular linked list the last node of the list contains the address of the first node and forms a circular chain.



Linked List Nodes as Pointers

```
1 class Node(object):
2     def __init__(self, value=None, pointer=None):
3         self.value = value
4         self.pointer = pointer
5
6     def getData(self):
7         return self.value
8
9     def getNext(self):
10        return self.pointer
11
12    def setData(self, newdata):
13        self.value = newdata
14
15    def setNext(self, newpointer):
16        self.pointer = newpointer
```

Nodes: Testing

```
1 if __name__ == '__main__':
2     L = Node("a", Node("b", Node("c", Node("d"))))
3     assert(L.pointer.pointer.value=='c')
4
5     print(L.getData())
6     print(L.getNext().getData())
7     L.setData('aa')
8     L.setNext(Node('e'))
9     print(L.getData())
10    print(L.getNext().getData())
```

Linked List Code: Initialization

```
1 from node import Node
2
3 class LinkedList(object):
4
5     def __init__(self):
6         self.head = None
7         self.length = 0
8         self.tail = None # this is different from ll lifo
```

Linked List: Insertion at the Beginning

1. The first Node is the Head for any Linked List.
2. When a new Linked List is instantiated, it just has the Head, which is Null.
3. Else, the Head holds the pointer to the first Node of the List.
4. When we want to add any Node at the front, we must make the head point to it.
5. And the Next pointer of the newly added Node, must point to the previous Head, whether it be NULL(in case of new List) or the pointer to the first Node of the List.
6. The previous Head Node is now the second Node of Linked List, because the new Node is added at the front.

Linked List: Insertion at the Beginning

```
1    def addFirst(self, value):
2        self.length = 1
3        node = Node(value)
4        self.head = node
5        self.tail = node
```

Linked List: Insertion at the End

1. If the Linked List is empty then we simply, add the new Node as the Head of the Linked List.
2. If the Linked List is not empty then we find the last node, and make it' next to the new Node, hence making the new node the last Node.

Linked List: Insertion at the Beginning

```
1    def add(self, value):
2        self.length += 1
3        node = Node(value)
4        if self.tail:
5            self.tail.pointer = node
6        self.tail = node
7
8    def addNode(self, value):
9        if not self.head:
10           self.addFirst(value)
11       else:
12           self.add(value)
```

Linked List: Printing the elements

```
1   def printList(self):
2       node = self.head
3       while node:
4           print(node.value)
5           node = node.pointer
```

Linked List: Finding an elements

```
1   def find(self, index):
2       prev = None
3       node = self.head
4       i = 0
5       while node and i < index:
6           prev = node
7           node = node.pointer
8           i += 1
9       return node, prev, i
```

Linked List: Deleting a Node

1. We first search the Node with data which we want to delete.
2. If the Node to be deleted is the first node, then simply set the Next pointer of the Head to point to the next element from the Node to be deleted.
3. If the Node is in the middle somewhere, then find the Node before it, and make the Node before it point to the Node next to it.

Linked List: Deleting a Node

```
1   def deleteNode(self, index):
2       if not self.head or not self.head.pointer:
3           self.deleteFirst()
4       else:
5           node, prev, i = self.find(index)
6           if i == index and node:
7               self.length -= 1
8               if i == 0 or not prev :
9                   self.head = node.pointer
10              else:
11                  prev.pointer = node.pointer
12              if not self.tail == node:
13                  self.tail = prev
14          else:
15              print('Node with index {} not found'.format
                    (index))
```

Linked Lists: Testing

```
1 if __name__ == '__main__':
2     ll = LinkedList()
3     for i in range(1, 5):
4         ll.addNode(i)
5     print('The list is:')
6     ll.printList()
7     print('The list after deleting node with index 2:')
8     ll.deleteNode(2)
9     ll.printList()
10    print('The list after adding node with value 15')
11    ll.add(15)
12    ll.printList()
13    print("The list after deleting everything...")
14    for i in range(ll.length-1, -1, -1):
15        ll.deleteNode(i)
16    ll.printList()
```

2nd Assignment

1. https://www.hackerrank.com/
   - Complete 25 Algorithms challenges under the following subdomains:
     Warmup (10), Sorting (any 10), Strings (any 5).
2. Do a benchmark analysis for each one: Quicksort, MergeSort
   - Generate a random list of integers (where n = {10, 100, 1000, 10000, . . . }).
   - Use timeit to measure execution time.
   - Produce a report in GitHub explaining the performance of the two algorithms.
3. Email `ichatz@diag.uniroma1.it`
   Subject: [PCS2] Homework 2
   A link to a github repository with your python solutions, for all challenges.
4. **Deadline: 7/December/2018, 23:59**