Data Structures & Algorithms

Week 9

Lists

Lecturer: Dr. Msagha J Mbogholi, PhD

Flashback from Lesson 8

- A queue is an ordered list in which insertions are done at one end (*rear*) and deletions are done at other end (*front*). The first element to be inserted is the first one to be deleted. Hence, it is called First in First out (FIFO) or Last in Last out (LILO) list."
- Main operations of a queue are enqueue (to add an element) and dequeue (to remove an element).
- Queues can be implemented in three data structures namely, Lists, Linked lists and arrays. Arrays can be further implemented as standard arrays or circular arrays.
- A priority queue is a queue in which each item is assigned a priority and items with a higher priority are removed before those with a lower priority, irrespective of when they were added.
- Priority queue can be implemented using the following data structures: arrays, Linked list, Heap data structure, and binary search tree.

Content

- Overview
- Using lists
- Linked List implementations

Part 1

Overview

Part 1

Overview

1.1 Introduction

- I was recently pondering over my annual to-do list for 2023. Doesn't time really fly when you have set so many targets for a given year? I realize that I've only managed to tick just about half of the list! And I'm hoping to achieve this by December 31st 2022….a tall order indeed.
- I usually write down my list at the beginning of the year and theoretically figure out a way to try to achieve the targets; it doesn't always work (as you can see for this year) but it gives me some drive and a sense of purpose.
- For shorter targets I use the notes feature in my phone which has a list feature. Once one is complete I tick it off and it disappears from the list (which I don't like since later on I don't feel that sense of achievement to look back and see all those ticks).
- Suffice to say the use of lists is very evident in everyday life. You wake up in the morning and your brain is already ticking off the things you need to do that day, and when you miss out on some mundane task it will give you that nagging feeling that something hasn't been done yet. I'm sure you know the feeling.

1.1 Introduction

- One thing that is obvious is that the use of lists goes on even into the office, planning functions or events, and so on…the list is endless.
- Clearly there has to be a way in which you can manipulate your list, right? If its for an event you're planning you should be able to strike out what has already been done (or tick), add some items to the list, move some up or down the list (depending on some order like what needs to happen first before another event can take place), edit other items and so on.
- This introduces the idea of lists in computing; the concept is the same just as you would do it in the real world. We define what a list is, and give an overview of its functionality and operations, particularly in the Python environment.

Part 2

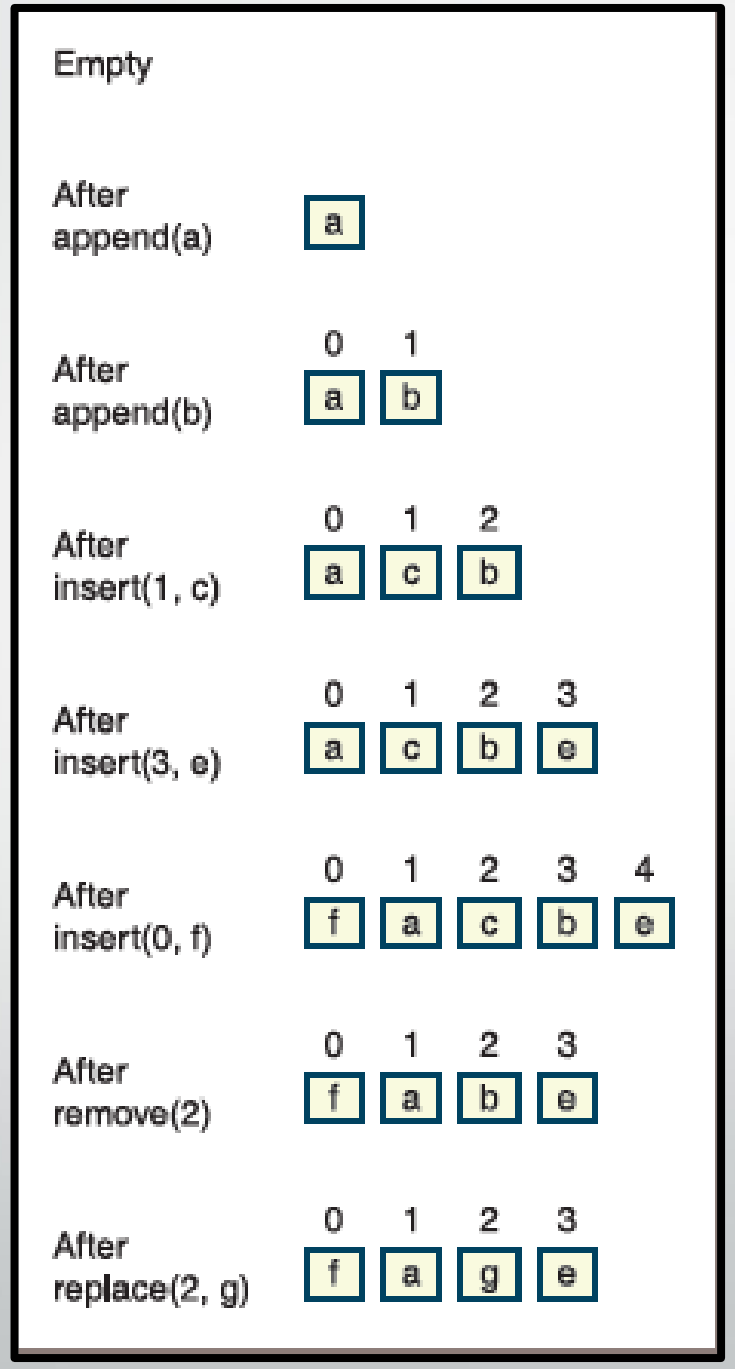
Using lists

2.1. Overview

- As we could deduce from section 1.1 for a list to be fully meaningful, it has to follow a certain order. In fact we daresay that order is critical to the list. Nonetheless, it is possible to have a list that is not sorted (take the example of when you're writing down your thoughts, you write what comes to your mind, and then sort them out later).
- Although the items in a list are always logically contiguous, they need not be physically contiguous in memory. Array implementations of lists use physical positions to represent logical order, but linked implementations do not.
- The element (item) at the beginning of the list is at it's head while the last element is at the tail. Items in a list retain position relative to each other over time, and additions and deletions affect predecessor/successor relationships only at the point of modification. (Lambert, 2019).
- List positions are counted from 0 through the length of the list minus 1. Each numeric position is also called an **index (**remember arrays?). When a list is visualized, the indices decrease as you move to the left and increase as you move to the right. Figure 1 shows how a list changes in response to a succession of operations. The operations, which represent just a small subset of many that are possible for lists, are described in Table 1.

Figure 1

Different states of a list



(From Lambert, 2019)

Table 1

List operations

Operation	What It Does
`add(item)`	Adds `item` to the tail of the list.
`insert(index, item)`	Inserts `item` at position `index`, shifting other items to the right by one position if necessary.
`replace(index, item)`	Replaces the item at position `index` with `item`.
`pop(index)`	Removes the item at position `index`, shifting other items to the left by one position if necessary.

(From Lambert, 2019)

2.2 Categories

- An examination of available literature on lists, and closer inspection of the different operations that are possible on lists yields enables these to be categorized.
- Most operations on lists are based on either the contents of the list, or on working with the index positions of items on the list. A third option is what we can simply call the positional list ADT. We thus categorize these as follows:
  - Index based operations
  - Content based operations
  - Positional list ADT

2.2.1 Index based operations

- **Index-based operations** manipulate items at designated indices within a list. In the case of array-based implementations, these operations also provide the convenience of random access. (Lambert, 2019)
- Consider a list contains *n* items. Because a list is ordered linearly (recall lists fall under linear collections), you can refer precisely to an item in a list via its relative position from the head of the list using an index that runs from 0 to *n* – 1. Thus, the head is at index 0 and the tail is at index *n* – 1. (much like arrays which were covered in lesson 4).
- Table 2 describes some of the index based operations that can be performed on a list called L. Thereafter we examine a few examples of application of these operations on some lists.

Table 2

Index based list operations

List Method	What It Does
`L.insert(i, item)`	Adds `item` at index `i`, after shifting items to the right by one position.
`L.pop(i = None)`	Removes and returns the item at index `i`.
	If `i` is absent, removes and returns the last item.
	Precondition: `0 <= i <= len(L)`.
`L[i]`	Returns the item at index `i`.
	Precondition: `0 <= i <= len(L)`.
`L[i] = item`	Replaces the item at index `i` with `item`.
	Precondition: `0 <= i <= len (L)`.

(From Lambert, 2019)

2.2.1 Index based operations

- We note that a list doesn't have to include items of the same data type only; thus it is not necessary to declare data types, only the contents will suffice. Python will tell the type of data from your declaration. Let us examine a few index based operations on lists:
- myList1 = ["Mike", 3, 22, 50.7] #list of mixed data types
- myList2 = [25, 90, 70, 120] #list of same data types
- print(item at index 0 in myList1: ", myList1[0]
- print(item at index 2 in myList2: ", myList2[2]
- The output will be:
- item at index 0 in myList1: Mike
- item at index 2 in myList2: 70

2.2.1 Index based operations

- Indexes can also be accessed using negative numbers. The effect of this is to read from the back of the list (starting from the right hand side of the list) as opposed to the traditional way of reading from the front of the list. Let us demonstrate this using our lists. Also it is worth noting that lists are always declared using square [] brackets and not (); the latter would create a special list called a tuple, which is immutable (contents can't be changed).
- myList1 = ["Mike", 3, 22, 50.7] #list of mixed data types
- myList2 = [25, 90, 70, 120] #list of same data types
- print(item at index 3 in myList1: ", myList1[-1]
- print(item at index 1 in myList2: ", myList2[-3]
- Output will be:
- item at index 3 in myList1: 50.7
- item at index 1 in myList2: 90

2.2.1 Index based operations

- Let us now remove, insert and replace some items in a list using Python operations in table 2:
- myList3 = [24, 19, 20, 18]
- myList3.pop(1) # remove the item in index 1 position
- Output:
- [24, 20, 18]
- R = myList3.pop(1)
- Print (R)
- Output:
- 19

2.2.1 Index based operations

- Let's insert and remove some items in a list:
- Cars = ["Audi", "BMW, "Toyota", "Mercedes"]
- Cars.append ("Jeep") # this adds Jeep to the end of the list
- print(Cars)
- Cars.insert(1, "VW") #inserts VW in position 1 of the list
- print(Cars)
- Cars.remove("Toyota") #removes the item Toyota
- print(Cars)
- Cars.reverse() #reverse the order of elements in the list.
- Output: ["Audi", "BMW, "Toyota", "Mercedes", "Jeep"]
- Output: ["Audi", "VW", "BMW, "Toyota", "Mercedes", "Jeep"]
- Output: ["Audi", "BMW, "Mercedes", "Jeep"]
- Output: ["Jeep", "Mercedes", "BMW", "Audi"]

2.2.1 Index based operations

- One other index based operation worth mentioning is the slice operator; it is used to split a list based on the declared index positions; essentially what we would be doing here is to create a sub-list. The syntax is as follows:
- <sublist name> = <original list name>[i:j]
- Where i is the index of the first item in the sublist, and
- j is the index of the item next to the last in the sublist.
- This will return a slice from i^{th} to $(j-1)^{th}$ items from the <original list name>.
- Further, while slicing, both operands "i" and "j" are optional. If not used, "i" is 0 and "j" is the last item in the list. Negative index can be also be used in slicing.
- Let us demonstrate this using two examples from (*Python - Access List Items*, n.d.).

2.2.1 Index based operations

- list1 = ["a", "b", "c", "d"]
- list2 = [25.50, True, -55, 1+2j]
- print ("Items from index 1 to 2 in list1: ", list1[1:3])
- print ("Items from index 0 to 1 in list2: ", list2[0:2])
- Output:
- Items from index 1 to 2 in list1: ['b', 'c']
- Items from index 0 to 1 in list2: [25.5, True]

2.2.1 Index based operations

- list1 = ["a", "b", "c", "d"]
- list2 = [25.50, True, -55, 1+2j]
- list4 = ["Rohan", "Physics", 21, 69.75]
- list3 = [1, 2, 3, 4, 5]
- print ("Items from index 1 to last in list1: ", list1[1:])
- print ("Items from index 0 to 1 in list2: ", list2[:2])
- print ("Items from index 2 to last in list3", list3[2:-1])
- print ("Items from index 0 to index last in list4", list4[:])

2.2.1 Index based operations

- Output:
- Items from index 1 to last in list1: ['b', 'c', 'd’]
- Items from index 0 to 1 in list2: [25.5, True]
- Items from index 2 to last in list3 [3, 4]
- Items from index 0 to index last in list4 ['Rohan', 'Physics', 21, 69.75]
- Really cool stuff, huh?

2.2.2 Content based operations

- **Content-based operations** are based not on an index, but on the content of a list. These operations usually expect an item as an argument and do something with it and the list.
- Some of these operations search for an item equal to a given item before taking further action. Table 3 lists three basic content-based operations for a list named L. Note that add is used instead of append, for consistency with other collections. (Lambert, 2019)
- Thereafter we examine a few examples of how to use these operations with lists.

Table 3

Content based list operations

List Method	What It Does
`L.add(item)`	Adds `item` after the list's tail.
`L.remove(item)`	Removes `item` from the list.
	Precondition: `item` is in the list.
`L.index(item)`	Returns the position of the first instance of `item` in the list.
	Precondition: `item` is in the list.

(From Lambert, 2019)

2.2.2 Content based operations

- myList3 = [1, 2, 3, 4, 5]
- print ("Original list ", myList3)
- myList3[2] = 10 #changing a value at a particular index
- print ("List after changing value at index 2: ", myList3)
- Output:
- Original list [1, 2, 3, 4, 5]
- List after changing value at index 2: [1, 2, 10, 4, 5]

2.2.2 Content based operations

- Del can also be used to delete an item from a list
- list1 = ["a", "b", "c", "d"]
- print ("Original list: ", list1)
- del list1[2]
- print ("List after deleting: ", list1)
- Output:
- Original list: ['a', 'b', 'c', 'd’]
- List after deleting: ['a', 'b', 'd']

2.2.2 Content based operations

- It is possible to delete a series of consecutive items from a list with the slicing operator introduced in section 1.3.1. Take a look at the following example from (*Python - Remove List Items*, n.d.) –
- list2 = [25.50, True, -55, 1+2j]
- print ("List before deleting: ", list2)
- del list2[0:2]
- print ("List after deleting: ", list2)
- Output:
- List before deleting: [25.5, True, -55, (1+2j)]
- List after deleting: [-55, (1+2j)]

2.2.3 Positional list ADT

- Recall the example given at the beginning of this lesson regarding those customers who would use all sorts of excuses (ludicrous or not) to cut to the front of the queue? Well this tells us that it is not really the ideal scenario to be able to use a number to determine where you are in the queue; much like the way we use indexes to determine positions in the list.
- What if I genuinely am third in line and I leave to go fill in the banking slip? By the time I get back any number of customers might have been served; worse still the person behind me might also have been served and left the banking hall! How do I convince the folks in the queue that I was genuinely in the queue before them? Clearly this is not ideal.
- (Goodrich et al., 2018) are in agreement with this and declare, "indices are not a good abstraction for describing a local position in some applications, because the index of an entry changes over time due to insertions or deletions that happen earlier in the sequence. For example, it may not be convenient to describe the location of a person waiting in line by knowing precisely how far away that person is from the front of the line"

2.2.3 Positional list ADT

- (Goodrich et al., 2018) recommend an abstraction, whereby there is some other means for describing a position. The best way to do this is to use a cursor that will determine a position without using an integer index. Lambert (2019) refers to this as a positional operator, though the approach to constructing it is the same as Goodrich et al.'s (2018).
- We describe the latter's approach as it is simpler to understand.
- Goodrich et al. (2018) begin the approach as follows:" To provide for a general abstraction of a sequence of elements with the ability to identify the location of an element, we define a ***positional list ADT*** as well as a simpler ***position*** abstract data type to describe a location within a list. A position acts as a marker or token within the broader positional list. A position *p* is unaffected by changes elsewhere in a list; the only way in which a position becomes invalid is if an explicit command is issued to delete it.
- A position instance is a simple object, supporting only the following method:

2.2.3 Positional list ADT

- p.element( ): Return the element stored at position p.
- In the context of the positional list ADT, positions serve as parameters to some methods and as return values from other methods."
- The positional list (let us call it L) will require both accessor and mutator methods. The accessor methods are described in figure 2 while the update methods are described in figure 3.

Figure 2

Accessor methods for positional list

L.first( ):	Return the position of the first element of L, or None if L is empty.
L.last( ):	Return the position of the last element of L, or None if L is empty.
L.before(p):	Return the position of L immediately before position p, or None if p is the first position.
L.after(p):	Return the position of L immediately after position p, or None if p is the last position.
L.is_empty( ):	Return True if list L does not contain any elements.
len(L):	Return the number of elements in the list.

(From Goodrich et al., 2018)

Figure 3

Update methods for positional list

L.add_first(e):	Insert a new element e at the front of L, returning the position of the new element.
L.add_last(e):	Insert a new element e at the back of L, returning the position of the new element.
L.add_before(p, e):	Insert a new element e just before position p in L, returning the position of the new element.
L.add_after(p, e):	Insert a new element e just after position p in L, returning the position of the new element.
L.replace(p, e):	Replace the element at position p with element e, returning the element formerly at position p.
L.delete(p):	Remove and return the element at position p in L, invalidating the position.

(From Goodrich et al., 2018)

2.2.3 Positional list ADT

- There is also a special accessor method for figure 2 called iter, which is an iterator. It's used as follows:
- iter(L): Return a forward iterator for the *elements* of the list.
- An iterator is an object that contains a countable number of values.
- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values. Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods __iter__() and __next__().
- Lists, tuples, dictionaries, and sets are all iterable objects. They are iterable *containers* which you can get an iterator from. (*Python Iterators*, n.d.)
- This then is the purpose of the iter(L) accessor method for our position list.
- An example of use of an iterator is also shared in the next slide.

2.2.3 Positional list ADT

- An example of an iterator courtesy (*Python Iterators*, n.d.):
- # Return an iterator from a tuple, and print each value/ can also be used for a list:
- mytuple = ("apple", "banana", "cherry")
  myit = iter(mytuple)
  print(next(myit))
  print(next(myit))
  print(next(myit))
- Output:
- apple
  banana
  cherry

2.2.3 Positional list ADT

- Goodrich et al.,(2018) explain how to use the methods described in figure 2 and figure 3 thus: "
- Note well that the first( ) and last( ) methods of the positional list ADT return the associated *positions*, not the *elements*. (This is in contrast to the corresponding first and last methods of the deque ADT.) The first element of a positional list can be determined by subsequently invoking the element method on that position, as L.first().element(). The advantage of receiving a position as a return value is that we can use that position to navigate the list.
- For example, the following code fragment prints all elements of a positional list named data:

2.2.3 Positional list ADT

- cursor = data.first( )
- while cursor is not None:
- print(cursor.element( )) # print the element stored at the position
- cursor = data.after(cursor) # advance to the next position (if any)
- This code relies on the stated convention that the None object is returned when after is called upon the last position. That return value is clearly distinguishable from any legitimate position. The positional list ADT similarly indicates that the None value is returned when the before method is invoked at the front of the list, or when first or last methods are called upon an empty list. Therefore, the above code fragment works correctly even if the data list is empty."
- Users can also use the traditional *for* loop to traverse the list. Finally we consider an example of implementation of the positional list ADT in figure 4. Position instances are identified by variables p, q, r and so on; subscript notations indicate positions and it is assumed that the list L is initially empty.

Figure 4

Implementation of positional list L

Operation	Return Value	L
L.add_last(8)	p	8_p
L.first( )	p	8_p
L.add_after(p, 5)	q	$8_p, 5_q$
L.before(q)	p	$8_p, 5_q$
L.add_before(q, 3)	r	$8_p, 3_r, 5_q$
r.element( )	3	$8_p, 3_r, 5_q$
L.after(p)	r	$8_p, 3_r, 5_q$
L.before(p)	None	$8_p, 3_r, 5_q$
L.add_first(9)	s	$9_s, 8_p, 3_r, 5_q$
L.delete(L.last( ))	5	$9_s, 8_p, 3_r$
L.replace(p, 7)	8	$9_s, 7_p, 3_r$

(From Goodrich et al., 2018)

Part 3

Linked list implementations

3.1 Singly linked lists

- In earlier lessons we described the singly linked list and its components: head, tail, nodes. The implementation of singly linked lists is applicable to two ADTs, namely:
- A stack
- A queue
- However, due to time and space constraints we discuss implementation in the stack, and implement the queue implementation in a circular linked list. It is believed the learner will use the knowledge gained in implementation in circular linked list for the queue (section 3.2 of this lesson) to apply it here.

3.3.1 Implementation with a stack

- Goodrich et al. (2018) describe this implementation: "
- In designing such an implementation, we need to decide whether to model the top of the stack at the head or at the tail of the list. There is clearly a best choice here; we can efficiently insert and delete elements in constant time only at the head. Since all stack operations affect the top, we orient the top of the stack at the head of our list.
- To represent individual nodes of the list, we develop a lightweight Node class. This class will never be directly exposed to the user of our stack class, so we will formally define it as a nonpublic, nested class of our eventual LinkedStack class … The definition of the Node class…" is shared in the next slide.
- The code for the Linkedstack class is shared in figure 5. "When implementing the top method, the goal is to return the ***element*** that is at the top of the stack. When the stack is empty, we raise an Empty exception.. When the stack is nonempty, self. head is a reference to the first ***node*** of the linked list. The top element can be identified as self. head. element." (Goodrich et al., 2018)

Finally table 4 examines the time complexity of the Linkedstack implementation.

3.3.1 Implementation with a stack

```
class Node:
"""Lightweight, nonpublic class for storing a singly linked node."""
__slots__ = _element , _next  # streamline memory usage
def __init__ (self, element, next):      # initialize node's fields
    self. _element = element # reference to user's element
    self. _next = next # reference to next node
```

Figure 5 (a)

Linkedstack class

```
1  class LinkedStack:
2    """LIFO Stack implementation using a singly linked list for storage."""
3
4    #---------------------- nested _Node class -----------------------
5    class _Node:
6      """Lightweight, nonpublic class for storing a singly linked node."""
7       __slots__ = '_element', '_next'         # streamline memory usage
8
9      def __init__(self, element, next):       # initialize node's fields
10       self._element = element                # reference to user's element
11       self._next = next                      # reference to next node
12
13   #------------------------- stack methods -------------------------
14   def __init__(self):
15     """Create an empty stack."""
16     self._head = None                        # reference to the head node
17     self._size = 0                           # number of stack elements
18
19   def __len__(self):
20     """Return the number of elements in the stack."""
21     return self._size
```

(From Goodrich et al., 2018)

Figure 5 (b)

Linkedstack class

```
23    def is_empty(self):
24      """Return True if the stack is empty."""
25      return self._size == 0
26
27    def push(self, e):
28      """Add element e to the top of the stack."""
29      self._head = self._Node(e, self._head)      # create and link a new node
30      self._size += 1
31
32    def top(self):
33      """Return (but do not remove) the element at the top of the stack.
34
35      Raise Empty exception if the stack is empty.
36      """
37      if self.is_empty():
38        raise Empty('Stack is empty')
39      return self._head._element                  # top of stack is at head of list
```

(From Goodrich et al., 2018)

Figure 5 (c)

Linkedstack class

```
40    def pop(self):
41      """Remove and return the element from the top of the stack (i.e., LIFO).
42
43      Raise Empty exception if the stack is empty.
44      """
45      if self.is_empty( ):
46        raise Empty('Stack is empty')
47      answer = self._head._element
48      self._head = self._head._next                  # bypass the former top node
49      self._size -= 1
50      return answer
```

(From Goodrich et al., 2018)

Table 4

Complexity of Linkedstack operations

Operation	Running Time
S.push(e)	$O(1)$
S.pop( )	$O(1)$
S.top( )	$O(1)$
len(S)	$O(1)$
S.is_empty( )	$O(1)$

(From Goodrich et al., 2018)

3.2 Circular linked lists

- The operations of the circular linked list was described in lesson 2.
- Goodrich et al. (2018) describe the queue implementation of a circular linked list as follows: "To implement the queue ADT using a circularly linked list, we rely on the (structure of the circular linked list), in which the queue has a head and a tail, but with the next reference of the tail linked to the head. Given such a model, there is no need for us to explicitly store references to both the head and the tail; as long as we keep a reference to the tail, we can always find the head by following the tail's next reference.
- Code in (figure 6) provide an implementation of a CircularQueue class based on this model. The only two instance variables are _tail, which is a reference to the tail node (or None when empty), and _size, which is the current number of elements in the queue. When an operation involves the front of the queue, we recognize self. _tail. _next as the head of the queue. When enqueue is called, a new node is placed just after the tail but before the current head, and then the new node becomes the tail.

3.2 Circular linked lists

- In addition to the traditional queue operations, the CircularQueue class supports a rotate method that more efficiently enacts the combination of removing the front element and reinserting it at the back of the queue. With the circular representation, we simply set self._tail = self._tail._next to make the old head become the new tail (with the node after the old head becoming the new head).”
- The code is shared in figure 6.

Figure 6 (a)

Circularqueue class

```
1  class CircularQueue:
2    """Queue implementation using circularly linked list for storage."""
3
4    class _Node:
5      """Lightweight, nonpublic class for storing a singly linked node."""
6      (omitted here; identical to that of LinkedStack._Node)
7
8    def __init__(self):
9      """Create an empty queue."""
10     self._tail = None                    # will represent tail of queue
11     self._size = 0                       # number of queue elements
12
13   def __len__(self):
14     """Return the number of elements in the queue."""
15     return self._size
16
17   def is_empty(self):
18     """Return True if the queue is empty."""
19     return self._size == 0
```

(From Goodrich et al., 2018)

Figure 6 (b)

Circularqueue class

```
20    def first(self):
21      """Return (but do not remove) the element at the front of the queue.
22
23      Raise Empty exception if the queue is empty.
24      """
25      if self.is_empty( ):
26        raise Empty('Queue is empty')
27      head = self._tail._next
28      return head._element
29
30    def dequeue(self):
31      """Remove and return the first element of the queue (i.e., FIFO).
32
33      Raise Empty exception if the queue is empty.
34      """
35      if self.is_empty( ):
36        raise Empty('Queue is empty')
37      oldhead = self._tail._next
38      if self._size == 1:                    # removing only element
39        self._tail = None                    # queue becomes empty
40      else:
41        self._tail._next = oldhead._next     # bypass the old head
42      self._size -= 1
43      return oldhead._element
```

(From Goodrich et al., 2018)

Figure 6 (c)

Circularqueue class

```
45    def enqueue(self, e):
46      """Add an element to the back of queue."""
47      newest = self._Node(e, None)          # node will be new tail node
48      if self.is_empty( ):
49        newest._next = newest               # initialize circularly
50      else:
51        newest._next = self._tail._next     # new node points to head
52        self._tail._next = newest           # old tail points to new node
53      self._tail = newest                   # new node becomes the tail
54      self._size += 1
55
56    def rotate(self):
57      """Rotate front element to the back of the queue."""
58      if self._size > 0:
59        self._tail = self._tail._next       # old head becomes new tail
```

(From Goodrich et al., 2018)

3.3 Doubly linked lists

- The working of doubly linked lists was discussed in lesson 2.
- As opposed to a single linked list where the nodes are linked in one direction, the doubly linked nodes allow traversal back and forth between nodes.
- Unlike the circular linked node, however, the double linked node does not provide a direct link between the tail and the head.
- (Sivaprakash, 2019) describes the code required, and also provides a test code to show the working of this class:

Implementation of Queue operations using Doubly LinkedList in Python

- # A complete working Python program to demonstrate all
- # Queue operations using doubly 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
- self.prev = None # Initialize prev as null

- # Queue class contains a Node object
- class Queue:

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

```
# Function to add an element data in the Queue
  def enqueue(self, data):
        if self.last is None:
            self.head =Node(data)
            self.last =self.head
        else:
            self.last.next = Node(data)
            self.last.next.prev=self.last
            self.last = self.last.next
```

```
# Function to remove first element and return the element from the queue
  def dequeue(self):

        if self.head is None:
            return None
        else:
            temp= self.head.data
            self.head = self.head.next
            self.head.prev=None
            return temp
```

```
# Function to return top element in the queue
  def first(self):
        return self.head.data


# Function to return the size of the queue
  def size(self):


        temp=self.head
        count=0
        while temp is not None:
            count=count+1
            temp=temp.next
        return count
```

```
# Function to check if the queue is empty or not
  def isEmpty(self):
        if self.head is None:
            return True
        else:
            return False


# Function to print the stack
  def printqueue(self):
        print("queue elements are:")
        temp=self.head
        while temp is not None:
            print(temp.data,end="->")
            temp=temp.next
```

- # Code execution starts here
- if __name__=='__main__':
- # Start with the empty queue
- queue = Queue()
- print("Queue operations using doubly linked list")
- # Insert 4 at the end. So queue becomes 4->None
- queue.enqueue(4)
- # Insert 5 at the end. So queue becomes 4->5None
- queue.enqueue(5)
- # Insert 6 at the end. So queue becomes 4->5->6->None
- queue.enqueue(6)
- # Insert 7 at the end. So queue becomes 4->5->6->7->None
- queue.enqueue(7)
- # Print the queue
- queue.printqueue()
- # Print the first element
- print("\nfirst element is ",queue.first())
- # Print the queue size
- print("Size of the queue is ",queue.size())
- # remove the first element
- queue.dequeue()
- # remove the first element
- queue.dequeue()
- # first two elements are removed
- # Print the queue
- print("After applying dequeue() two times")
- queue.printqueue()
- # Print True if queue is empty else False
- print("\nqueue is empty:",queue.isEmpty())

Implementation of Queue operations using Doubly LinkedList in Python

- The output of the execution is as follows:
- `Queue operations using doubly linked list`
- `queue elements are: 4->5->6->7->`
- `first element is 4`
- `Size of the queue is 4`
- `After applying dequeue() two times queue elements are: 6->7->`
  `queue is empty: False`

Summary

- The element (item) at the beginning of the list is at it's head while the last element is at the tail. Items in a list retain position relative to each other over time, and additions and deletions affect predecessor/successor relationships only at the point of modification.
- Operations on lists can be categorized as index based operations, content based operations, and positional list ADT.
- Indices are not a good abstraction for describing a local position in some applications, because the index of an entry changes over time due to insertions or deletions that happen earlier in the sequence.
- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values. Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods __iter__() and __next__().
- Various implementations of the different types of linked lists (single linked, double linked, and circular linked) can be done using Python programming language.

References

- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2018). *Data structures and algorithms in Python*. Wiley.
- Lambert, K. (2019). *Fundamentals of Python: Data Structures*. Cengage Learning.
- *Linked List Implementation of Queue - javatpoint*. (2011). Www.javatpoint.com. https://www.javatpoint.com/linked-list-implementation-of-queue
- *Python - Access List Items*. (n.d.). Www.tutorialspoint.com. Retrieved October 23, 2023, from https://www.tutorialspoint.com/python/python_access_list_items.htm
- *Python - Remove List Items*. (n.d.). Www.tutorialspoint.com. Retrieved October 23, 2023, from https://www.tutorialspoint.com/python/python_remove_list_items.htm

References

- *Python Iterators*. (n.d.). Www.w3schools.com. Retrieved October 23, 2023, from https://www.w3schools.com/python/python_iterators.asp
- Sivaprakash, A. (2019, July 31). Python | Queue using Doubly Linked List - GeeksforGeeks. *GeeksforGeeks*. https://www.geeksforgeeks.org/python-queue-using-doubly-linked-list/