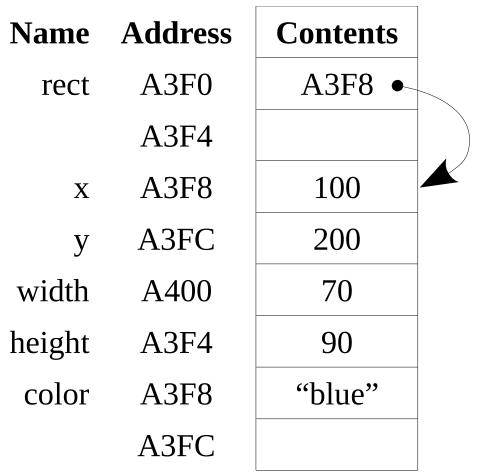
# Chapter 3 – Basic Data Structures (Linear Structures)

## *Linked List*

The problem with using the Python list for a queue is that it requires the values to be in consecutive locations in memory. So in order to insert at the front of the list, or to remove from the front of the list, we have to shift all the remaining values down one. We can do better if we don't require the values to be in consecutive locations in memory. What we can do instead is *link* the values together.

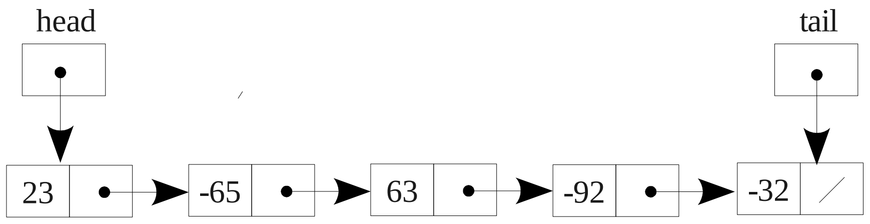
When we create a new object, that new object doesn't actually contain the data, it actually contains the address of the data, which is stored someplace else. For example, when we create an object of type myRectangle by **rect = myRectangle (x, y, width, height, color)**, rect is actually the address of the actual rectangle, as shown below.



| Name | Address | Contents |
|---|---|---|
| rect | A3F0 | A3F8 ● |
|  | A3F4 |  |
| x | A3F8 | 100 |
| y | A3FC | 200 |
| width | A400 | 70 |
| height | A3F4 | 90 |
| color | A3F8 | “blue” |
|  | A3FC |  |

Conceptually, we can think of the object rect as a pointer to the actual data. We don't really care about what the addresses are, just that it has the address of the data. Think about when you search for a book in the library. When you find a book you are interested in, the search tells you the call number. It isn't the book itself, but rather a pointer to where in the stacks you can go to find the actual book.

Given the concept of pointers, we can then put together a list of things that aren't in consecutive memory locations. For example, we could have a list of values: 23, -65, 63, -92, -32. We would like to put in a list that conceptually looks like:



This will allow these values to be any place in memory. All we need is for each value to have a reference to where we find the next value. We don't have to have the tail, but it makes it faster when we know we want to add to the end.

In order to do this, we need two data types.  One for our linked list (which has a head and tail), but another for the nodes that we put in the list that have a the data and a reference to the next one.  For the nodes in the list, we will have the following class:

```
class listNode:
    def init(self, data, next = None):
        self.data = data
        self.next = next

    def getData(self):
        return self.data

    def setData(self, data):
        self.data = data

    def getNext(self):
        return self.next

    def setNext(self, next):
        self.next = next
```

**Listing 4: Linked List Node**

The value we want to store will be held in the instance variable **data**.  The instance variable **next** will be the **listNode** that comes next.  The last node in the list will have **None** as its **next**.

(Note, I left out the two underscores from the data members.  This should still be done, but I decided to leave them out just to make the code a little more readable.)

Now to create a linked list, we need to have a reference to the first **listNode**, which we'll call the **head**, and the last **listNode**, which we'll call the **tail**.   The construct (**__init__**) starts the list off as an empty list.  We don't actually need the **tail**, but it makes adding to the end of the list fast.  If we don't have the **tail**, then we would have to *traverse* the list to find the last one.

```
class myList:
    def init(self):
        self.head = None
        self.tail = None
        self._size = 0
```

**Listing 5: Beginning of the Linked List Class**

If it doesn't seem intuitive to you that we would have to traverse the list, imagine we have a scavenger hunt.  In order to find the next clue, you have to go to where the current clue tells you to go.  It would be great if you could just go to the end, but the point of the scavenger hunt is to make you go to each one in turn.  We can't find the last node until we find the one before it.  We can't find that one until we find the one before that.  The only one we would know where to find is the first one.  So we'll keep track of the tail and be able to go right to the end if we need to. We will NOT, however, be able to short cut our way to any node in the middle.

The other short cut we will do is keep track of how many nodes are in the list with the **size** data instance. Would could just count the number of nodes each time we wanted to know how many there were, but again, that requires us to traverse the list.

Another thing would could do that would help with performance is to have links that go in the opposite direction. With only links that go forward, we can't go backwards through the list. But it is possible for us to double the number of links and have another set that go backwards through the list. This is called a *doubly linked list*, but we we not do that here.

The operations that we want to have on this list are:

| **Operation Name** | **Description** |
|---|---|
| **insert(i, x)** | Insert the value x at position i. If i is zero, insert as the first element. If i is the length of the list, then insert at the end. |
| **pop(i) or pop()** | Remove and return the value in the list at position i (starting at zero. If i is omitted, remove and return the last value. |
| **append(x)** | Append the value x to the end of the list. Same as insert(list.size(), x) |
| **index(x)** | Return the index (starting at zero) of the first occurrence of x in the list or return -1 if x is not found. |
| **remove(x)** | Removes the first occurrence of x in the list if it exists. |
| **front()** | Return the value at the front of the list but do not remove it. |
| **back()** | Return the value at the back of the list but do not remove it. |
| **size()** | Returns the number of values in the list |
| **isEmpty()** | Returns True if the list is empty, False if it is not |
| **__str__ ()** | Returns a string with all the values in the list. This allows one to print a list. |

## Insert

In order to insert a new value in the list, we need to consider a few cases: 1) inserting into an empty list, 2) inserting before the head, 3) inserting after the tail, 4) inserting into the middle.

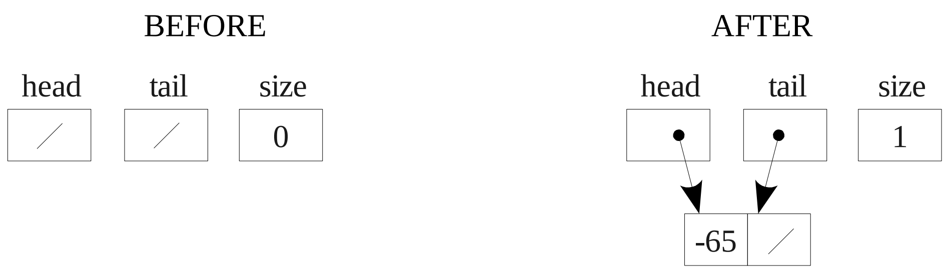
***Case (1) inserting into an empty list***

**Head** and **tail** are both **None** and **size** is zero. In that case, we need to create a new **listNode** and have both the **head** and **tail** point to it. The code to do that is:

```
        if (self.isEmpty()):
            self.head = listNode(x)
            self.tail = self.head
```

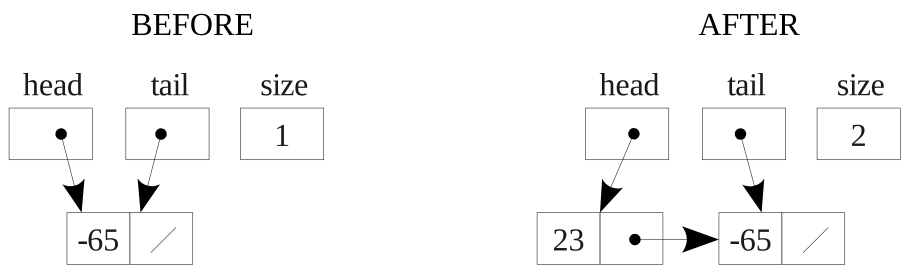
This creates a new node with the value if **x** (which is a parameter to insert) and has the **head** point to it.

We also want the **tail** to point to is so we set the **tail** to the same as the **head**.



***Case (2) inserting before the head***

In this case, we need to move the **head**. In the picture below, the tail is pointing to the same node as the **head**, but that isn't a requirement. There can be many nodes in between. The **tail** is irrelevant here. We need to make a new node, which points to the node the **head** is currently pointing to, then change what the **head** points to.



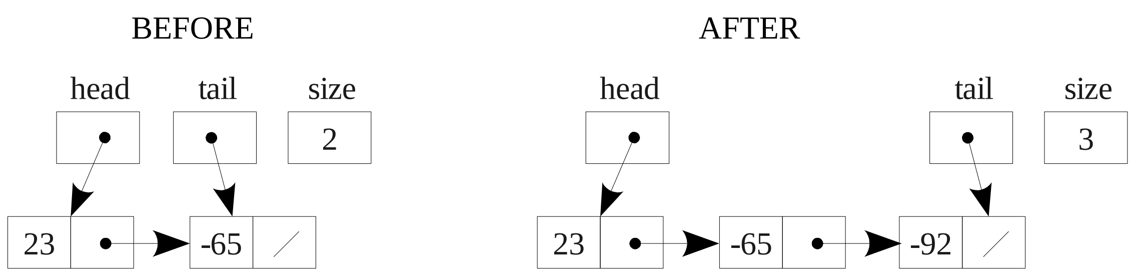
The code to do that is:

```
        if (i <= 0):
            self.head = listNode(x, self.head)
```

This creates a new **listNode** with the value of **x** and the **next** pointer points to the current **head** (which in the picture above is the node with --65). It then sets the **head** to point to the newly created node.

***Case (3) inserting after the tail***

In this case, we want to make a new node with the new value and **None** as the next. We want the **next** of the **tail** to point to the new node.



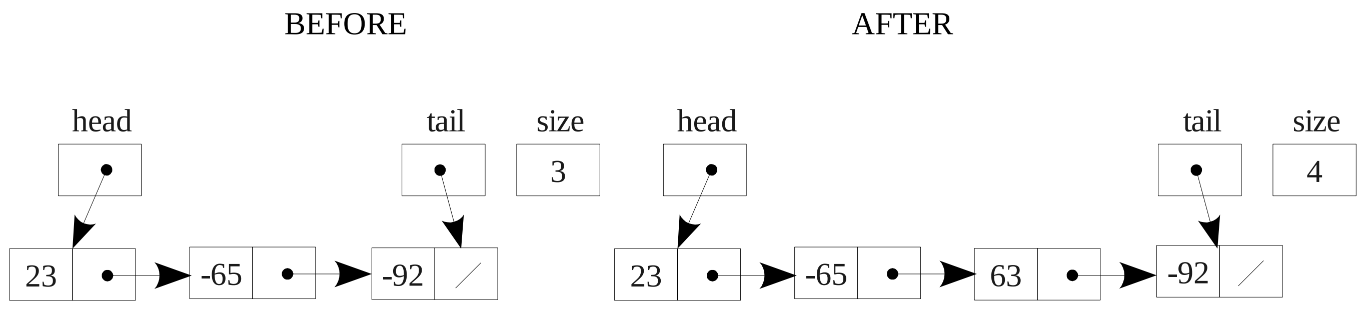
The code to do that is:

```
        elif (i >= self.size()):
            self.tail.setNext(listNode(x))
            self.tail = self.tail.getNext()
```

We create a new **listNode** (with the value of **x** and **None** as the **next**), then set the t**ail**'s **next** to point to it.  In the picture above, this would make the node with the -65 in it point to the one just created (with the -92).  Once that is done, then we want to move the **tail** to be the new node just created.

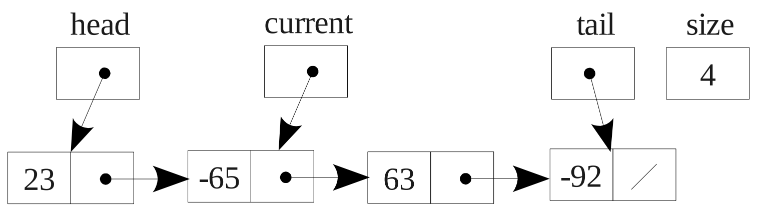
***Case (4) inserting into the middle***

In this case, we need to traverse the list until we find where to put the new value.  Notice that we need to stop traverse when we get to the node that is to be *JUST IN FRONT* of the new node, because we need that one to point to the new node.



Before we look at the code to insert, let's consider how to traverse the list.  We need to create a variable, which we will call **current**, that points to each node in the list as we traverse.  To move the **current** to the next one in the list, we can do **current = current.getNext()**.  We will do that until **current** goes off the end of the list.  We can know when it goes off the end of the list when **current** becomes equal to **None**.  So the structure of code to traverse the list looks like:

```
            current = self.head
            while (current != None):
                # Do something with current here
                current = current.getNext()
```



Now that we have the basic structure of a loop that will traverse the list, now we can traverse until we find the location of where we want to put the new value.  We need to stop one position short of where it needs to go because the previous node needs to point to the new node.

The code to do that is:

```
        else:
            # We need to traverse the list until we get to the (i-1)st one.
            current = self.head
            count = 0

            # We need to stop at the node previous to i to make
            # it point around the ith one.
            while (current != None and count < i-1):
                count += 1
                current = current.getNext()

            current.setNext(listNode(x, current.getNext()))
```

This is the basic traverse but we've added a counter so that we can stop traversing when we reach the node just before where the new node goes. Once we get there (and **current** points to it), we create a new node with the value of **x** and its **next** is what **current**'s **next** is. In the depiction above, we are inserting 63 at position 2. We would stop current when it gets to the node with the value -65. So we create a new node with the value of 63 and with a **next** point there that points to the node that -65 *USED* to point to (which is the node with the value -92). Once we have that new node, then we sent the **next** of the **current** (the node with -65) to then point to the new node.

The final thing that needs to be done in insert is to increment the size instance variable so we have an accurate count of the size of the list.

When we put all of this together, we have the following implementation of **insert(i, x)**:

---

```
    def insert(self, i, x):
        ''' Inserts x into the list at position i
        '''
        # list is empty, i doesn't matter.
        if (self.isEmpty()):
            self.head = listNode(x)
            self.tail = self.head

        # inserting before the head
        if (i <= 0):
            self.head = listNode(x, self.head)

        # inserting after the tail
        elif (i >= self.size()):
            self.tail.setNext(listNode(x))
            self.tail = self.tail.getNext()

        # inserting someplace in between
        else:
            # We need to traverse the list until we get to the (i-1)st one.
            current = self.head
            count = 0

            # We need to stop at the node previous to i to make
            # it point around the ith one.
            while (current != None and count < i-1):
                count += 1
                current = current.getNext()
            current.setNext(listNode(x, current.getNext()))

        self._size += 1
```

**Listing 6: Linked List Insert**

Before we implement the all the operations, the three that are most helpful for testing are the last three operations: size(), isEmpty(), and __str(). The implementation for these functions are given in Listing 7. By keeping track of the number of elements in the list, the size() function can simply return the _size instance variable. When we get to implementing the pop() and remove() functions, we will need to make sure that the _size is decremented.

The isEmpty() could be implemented by returning self._size == 0.

The __str__() function constructs a string that can be printed. It first creates an empty string. Then it traverses the list adding a string representation of each value to the resulting string. A finally, it returns the string. By making this __str__() function, one can simply print a list and it will show all the values.

```
def size(self):
    # By keeping track of the size as we go, we don't have to
    # count the number of nodes, which would be O(N).
    return self._size

def isEmpty(self):
    # Could return self.size == 0
    return self.head == None

def __str__(self):
    ''' Returns a string of the values in the list
    '''
    result = ""
    current = self.head
    while(current != None):
        result = result + " " + str(current.getData())
        current = current.getNext()
    return result
```

**Listing 7: Linked List isEmpty**