from node import Node

class Node2Way(Node):
    def __init__(self, initdata):
        Node.__init__(self, initdata)
        self.previous = None
    def getPrevious(self):
        return self.previous
    def setPrevious(self, newprevious):
        self.previous = newprevious

3. An alternative implementation of a Deque would be a doubly-linked implementation as in:

DoublyLinkedDeque Object

3

a) Determine the big-oh, \( O() \), for each Deque operation assuming the above doubly-linked implementation. Let \( n \) be the number of items in the Deque.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Big-oh</th>
</tr>
</thead>
<tbody>
<tr>
<td>isEmpty</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>addFront</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>removeFront</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>addRear</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>removeRear</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>size</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

4. A priority queue has the same operations as a regular queue, except the items are NOT returned in the FIFO (first-in, first-out) order. Instead, each item has a priority that determines the order they are removed. A hospital emergency room operates like a priority queue -- the person with the most serious injury has highest priority even if they just arrived.

a) Suppose that we have a priority queue with integer priorities such that the smallest integer corresponds to the highest priority. For the following priority queue, which item would be dequeued next?

priority queue: 40 10 79 30 13 5

b) To implement a priority queue, we could use an unordered Python list. If we did, what would be the big-oh notation for each of the following methods: (justify your answer)

- enqueue: \( O(1) \)
- dequeue: \( O(n) \)

c) To implement a priority queue, we could use a Python list order by priorities in decending order. If we did, what would be the big-oh notation for each of the following methods: (justify your answer)

- enqueue: \( O(n) \)
- dequeue: \( O(1) \)
1. Section 6.6 discusses a very “non-intuitive”, but powerful list/array-based approach to implement a priority queue, call a binary heap. The list/array is used to store a complete binary tree (a full tree with any additional leaves as far left as possible) with the items being arranged by heap-order property, i.e., each node is ≤ either of its children. An example of a min heap viewed as a complete binary tree would be:

```
       6
      / \
     3   10
    /   / \
   / \     /
  [4] 114 8
 /   /   /   /
300 [9] 125 117
```

Python List actually used to store heap items:

```
0  6  15  10  114  20  20  50  300  125  117  13
```

a) For the above heap, the list/array indexes are indicated in []'s. For a node at index \( i \), what is the index of:
   - its left child if it exists: \( 2 \times i \)
   - its right child if it exists: \( 2 \times i + 1 \)
   - its parent if it exists: \( \lfloor i / 2 \rfloor \)

b) What would the above heap look like after inserting 13 and then 3? (show the changes on above tree)

General Idea of insert(newItem):
- append newItem to the end of the list (easy to do, but violates heap-order property)
- restore the heap-order property by repeatedly swapping the newItem with its parent until it percolates to correct spot

c) What is the big-oh notation for inserting a new item in the heap? \( O(\log n) \)

d) Complete the code for the percUp method used by insert.

```python
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0

    def percUp(self, currentIdx):
        parentIdx = currentIdx // 2
        while parentIdx >= 1 and self.heapList[parentIdx] > self.heapList[currentIdx]:
            temp = self.heapList[parentIdx]
            self.heapList[parentIdx] = self.heapList[currentIdx]
            self.heapList[currentIdx] = temp
            currentIdx = parentIdx
            parentIdx = currentIdx // 2

    def insert(self, k):
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
        self.percUp(self.currentSize)
```

Lecture 7 - Page 1
2. Now let us consider the delMin operation that removes and returns the minimum item.

- What item would delMin remove and return from the above heap?
- What is the quickest way to fill the hole left by delMin? Move last item to index 1
- What new problem does this cause? 117 needs to percolate down heap

General Idea of delMin():
- Remember the minimum value so it can be returned later (easy to find - at index 1)
- Copy the last item in the list to the root, delete it from the right end, decrement size
- Restore the heap-order property by repeatedly swapping this item with its smallest child until it percolates down to the correct spot
- Return the minimum value

d) What would the above heap look like after delMin? (show the changes on above tree)

c) Complete the code for the percDown method used by delMin.

```python
class BinHeap:
    def minChild(self, i):
        if i * 2 + 1 > self.currentSize:  # if only left child
            return i * 2
        else:
            if self.heapList[i * 2] < self.heapList[i * 2 + 1]:
                return i * 2
            else:
                return i * 2 + 1

def delMin(self):
    retval = self.heapList[1]
    self.currentSize = self.currentSize - 1
    self.heapList.pop()
    self.percDown(1)
    return retval

def __str__(self):
    return str(self.heapList[:self.currentSize + 1])

def percDown(self, currentIndex):
    while True:
        leftChildIndex = currentIndex * 2
        if leftChildIndex > self.currentSize:
            return
        else:
            minChildIndex = self.minChild(currentIndex)
        if self.heapList[currentIndex] > self.heapList[minChildIndex]:
            return
        else:
            self.heapList[currentIndex],
    self.heapList[minChildIndex] =
    temp = self.heapList[currentIndex]
    self.heapList[currentIndex] =
    self.heapList[minChildIndex]
    self.heapList[minChildIndex] = temp
```

f) What is the big-oh notation for delMin? $O(\log n)$
Once we have a working BinHeap, then implementing the PriorityQueue class using a BinHeap is a piece of cake:

```python
>>> q = PriorityQueue()
>>> print(q)
[[]
>>> q.enqueue(5)
>>> q.enqueue(1)
>>> q.enqueue(7)
>>> print(q)
[1, 5, 7]
>>> q.dequeue()
1
>>> print(q)
[5, 7]
```

3. A "list" is a generic term for a sequence of items in a linear arrangement. Unlike stacks, queues and deques access to list items is not limited to either end, but can be from any position in the list. The general terminology of a list is illustrated by:

```
"Abstract view of a list"
```

| [0] | [1] | [2] | [3] | index/index in the list
|-----|-----|-----|-----|---------------------
|     |     |     |     |
| head|     |     |     |
|     | head| head| head|
|     |     |     |     |
|     |     |     |     |
|     |     |     |     |
|     |     |     |     |
| tail| tail| tail| tail|

There are three broad categories of list operations that are possible:

- **index-based operations** - the list is manipulated by specifying an index location, e.g.,
  myList.insert(3, item)  # insert item at index 3 in myList

- **content-based operations** - the list is manipulated by specifying some content (i.e., item value), e.g.,
  myList.remove(item)  # removes the item from the list based on its value

- **cursor-based operations** - a cursor (current position) can be moved around the list, and it is used to identify list items to be manipulated, e.g.,
  myList.first()  # sets the cursor to the head item of the list
  myList.next()  # moves the cursor one position toward the tail of the list
  myList.remove()  # deletes the second item in the list because that’s where the cursor is currently located

The following table summarizes the operations from the three basic categories on a list, L:

<table>
<thead>
<tr>
<th>Index-based operations</th>
<th>Content-based operations</th>
<th>Cursor-based operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.insert(index, item)</td>
<td>L.add(item)</td>
<td>L.hasNext()</td>
</tr>
<tr>
<td>item = L[index]</td>
<td>L.remove(item)</td>
<td>L.next()</td>
</tr>
<tr>
<td>L[index] = new_value</td>
<td>L.search(item)  #return Boolean</td>
<td>L.hasPrevious()</td>
</tr>
<tr>
<td>L.pop(index)</td>
<td>i = L.index(item)</td>
<td>L.previous()</td>
</tr>
</tbody>
</table>

Built-in Python lists are unordered with a mixture of index-based and content-based operations. We know they are implemented using a contiguous block of memory (i.e., an array). The textbook talks about an unordered list ADT, and a sorted list ADT which is more content-based. Both are implemented using a singly-linked list.

a) Why would a singly-linked list be a bad choice for implementing a cursor-based list ADT?