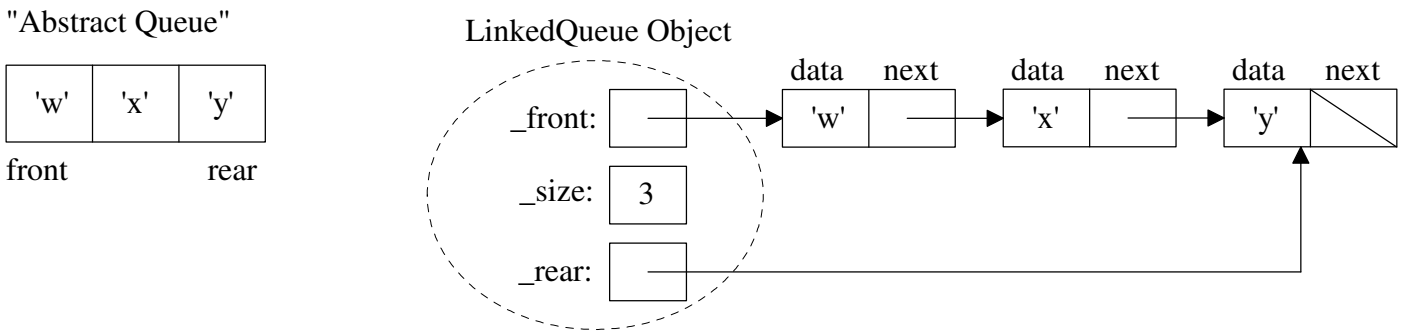
1. The Python `for` loop allows traversal of built-in data structures (strings, lists, tuple, etc) by an *iterator*. To accomplish this with ***our*** data structures we need to include an `__iter__(self)` method that gets used by the built-in `iter` function to create a special type of object called a *generator object*. The generator object executes as a separate process running concurrently with the process that created and uses the data structure being iterated. Iterators behave like stripped-down positional lists and

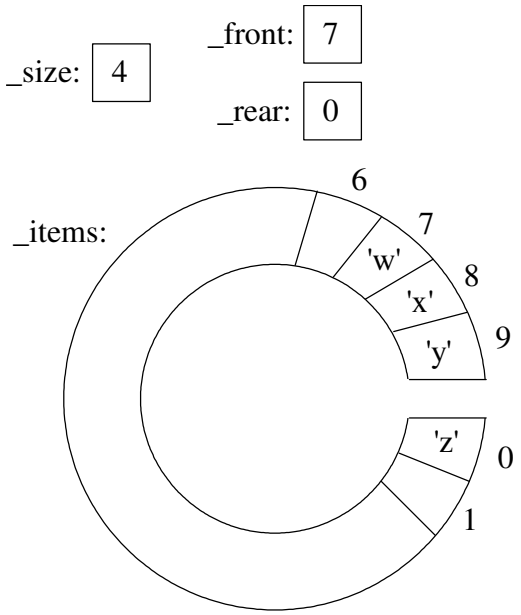
A singly-linked list implementation of the queue (`LinkedQueue` class in the text). Conceptually, a `LinkedQueue` object would look like:




```
class LinkedQueue(object):
    """ Link-based queue implementation."""


   def __iter__(self):
        """An iterator for a linked queue"""
        cursor = self._front
        while True:
            if cursor == None:
                raise StopIteration
            yield cursor.data
            cursor = cursor.next
```

The circular-array queue (`CircularArrayQueue`) implementation would look like:



a) Write an `__iter__` method for the `CircularArrayQueue`

HEAP METHOD	WHAT IT DOES
`H.add(item)`	Inserts `item` in its proper place in the heap, increasing the heap's size by one.
`H.pop()`	*Precondition*: the heap is not empty. Removes and returns the topmost item in the heap.
`H.peek()`	*Precondition*: the heap is not empty. Returns the item at the top of the heap.
`H.isEmpty()`	Returns `True` if the heap is empty, or `False` otherwise.
`H.__len__()`	Same as `len(H)`. Returns the number of items in the heap.
`H.__iter__()`	Same as `iter(H)`. Returns an iterator on the heap. This iterator visits the items from the minimum to the maximum.
`H.__str__()`	Same as `str(H)`. Returns a string representation of the heap that shows its shape.

[TABLE 18.8] The methods in the heap interface

2. So far, we have only looked at simple sorts. Recall that all simple sorts consist of nested loops:

- an outer loop that keeps track of the dividing line between the sorted and unsorted part
  - an inner loop that grows the size of the sorted part by one

Usually, the number of inner loop iterations is something like = (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 = n * (n-1)/2 which is $\Theta(n^2)$. Consider using a heap to sort a list/array.

a) If we put all of the list/array elements into a heap, what would be easily be able to determine?

b) What is the $O(\ )$ buiding the heap with all n items from the array?

c) Write an algorithm to perform the "heap" sort of an array.

d) What is the $O(\ )$ for your algorithm?

3. Another way to do better than the simple sorts is to employ divide-and-conquer to develop more "advanced" sorting algorithms that are recursive: Merge sort and Quick Sort.

In general, a problem can be solved recursively if it can be broken down into smaller problems that are identical in structure to the original problem.

a) What determines the "size" of a sorting problem?

b) How might we break the original problem down into smaller problems that are identical? Are there any additional parameters that might be needed? (recursive algorithms often need extra parameters)

c) What base case(s) (i.e., trival, non-recursive case(s)) might we encounter with recursive sorts?

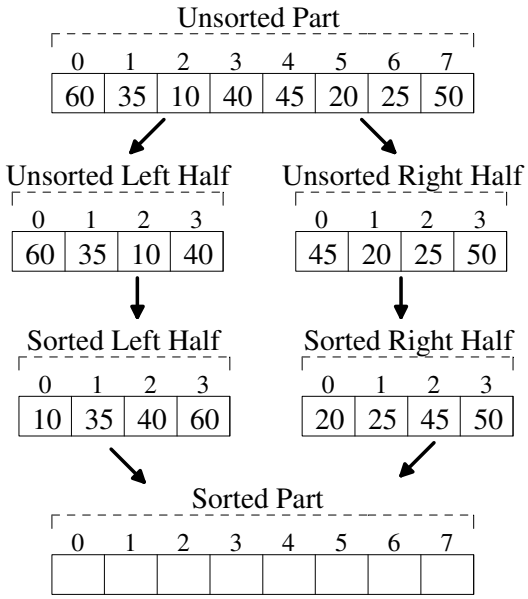
d) Consider why a recursive sort might be more efficient. Assume that I had a simple n^2 sorting algorithm with n = 100, then there is roughly 100^2 / 2 or 5,000 amount of work. Suppose I split the problem down into two smaller problems of size 50.

- If I run the n^2 algorithm on both smaller problems of size 50, then what would be the approximate amount of work?

- If I further solve the problems of size 50 by splitting each of them into two problems of size 25, then what would be the approximate amount of work?

4. The general idea merge sort is as follows. Assume "n" items to sort.

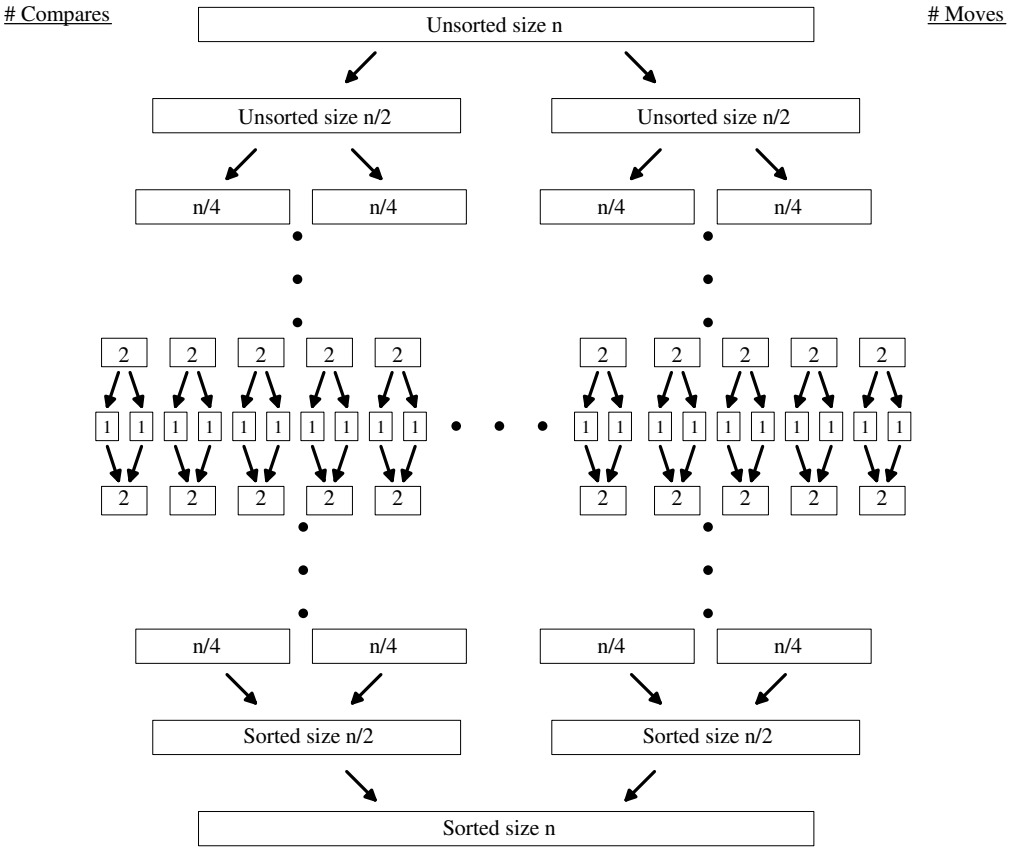
- Split the unsorted part in half to get two smaller sorting problems of about equal size = n/2
- Solve both smaller problem recursively using merge sort
- "Merge" the solution to the smaller problems together to solve the original sorting problem of size n



a) Fill in the sorted part in the above

b) Describe how you filled in the sorted part in the above example?

5. Merge sort is substantially faster than the simple sorts. Let's analyze the number of comparisons and moves of merge sort. Assume "n" items to sort.



a) On each level of the above diagram write the WORST-CASE total number of comparison for that level on the left, and the total number of moves for that level on the right.

b) What is the WORST-CASE total number of comparisons for the whole algorithm (i.e., add all levels)?

c) What is the total number of moves for the whole algorithm (i.e., add all levels)?

d) What is BEST-CASE total number of comparisons for the whole algorithm?

6. *Quick sort* is another advanced sort that often is quicker than merge sort (hence its name). The general idea is as follows. Assume "n" items to sort.

- Select a "random" item in the unsorted part as the *pivot*
- Rearrange (called *partitioning*) the unsorted items such that:

Pivot Index

All items < to Pivot	Pivot Item	All items >= to Pivot

- Quick sort the unsorted part to the left of the pivot
- Quick sort the unsorted part to the right of the pivot

a) What base case(s) would we have?

b) Because of the recursive nature of quick sort, what "extra" parameters would we need to specify the part of the list to sort?

c) Given the following `partition` function which returns the index of the pivot after this rearrangement.

```
def partition(lyst, left, right):
    # Find the pivot and exchange it with the last item
    middle = (left + right) / 2
    pivot = lyst[middle]
    lyst[middle] = lyst[right]
    lyst[right] = pivot
    # Set boundary point to first position
    boundary = left
    # Move items less than pivot to the left
    for index in xrange(left, right):
        if lyst[index] < pivot:
            temp = lyst[index]
            lyst[index] = lyst[boundary]
            lyst[boundary] = temp
            boundary += 1
    # Exchange the pivot item and the boundary item
    temp = lyst[boundary]
    lyst[boundary] = lyst[right]
    lyst[right] = temp
    return boundary
```

Complete the recursive `quicksortHelper` function.

```
def quicksort(lyst):
    quicksortHelper(lyst, 0, len(lyst) - 1)

def quicksortHelper(lyst, left, right):
```