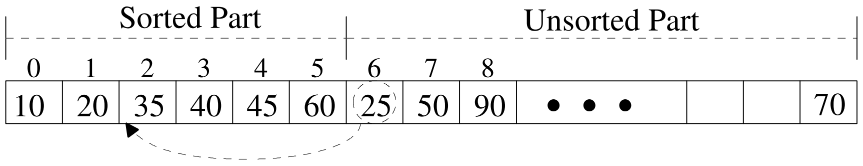
Data Structures (CS 1520) Homework #4 Due: March 23 (Sat) at 11:59 PM

Objective: Become more proficient at implementing sorting algorithms.

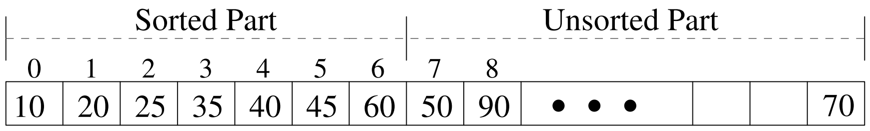
Part A: Recall the insertion sort code from lecture 16 and lab 8:

```
def insertionSort(myList):
    """Rearranges the items in myList so they are in ascending order"""

    for firstUnsortedIndex in range(1,len(myList)):
        itemToInsert = myList[firstUnsortedIndex]

        testIndex = firstUnsortedIndex - 1

        while testIndex >= 0 and myList[testIndex] > itemToInsert:
            myList[testIndex+1] = myList[testIndex]
            testIndex = testIndex - 1

        # Insert the itemToInsert at the correct spot
        myList[testIndex + 1] = itemToInsert
```

It sorts in ascending order by inserting the first item from the unsorted part into the sorted part on the left end of the list. For example, the inner-loop takes 25 and "inserts" it into the sorted part of the list "at the correct spot."



After 25 is inserted into the sorted part, the list would look like:



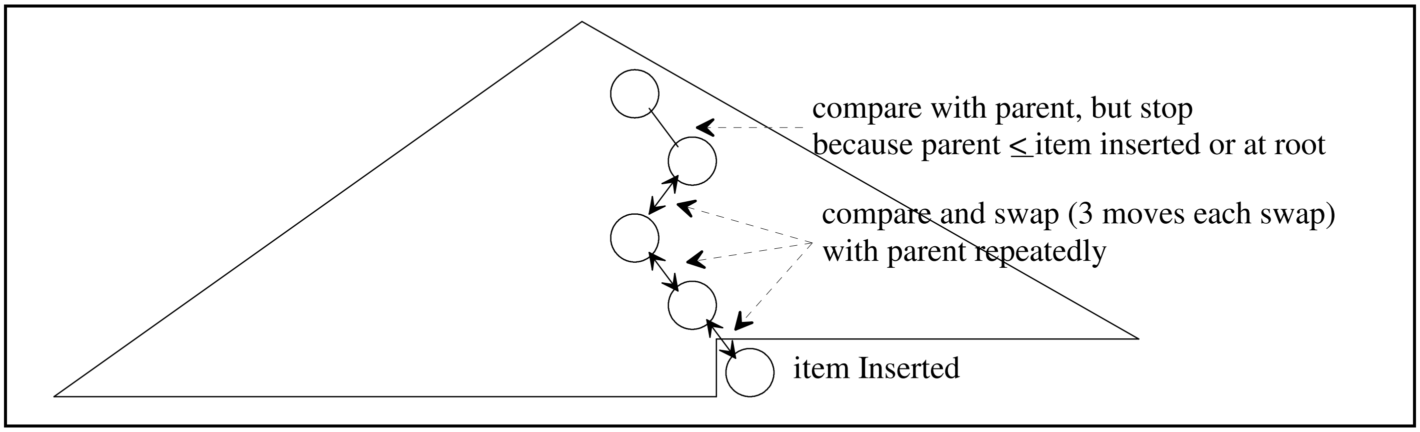
For Part A, I want you to modify and improve insertion sort (still sorting in ascending order) by:

- build the sorted part on the **right end** of the list and insert the last unsorted item into it repeatedly,
- speedup the inner-while-loop by:
  - ➢ starting the sort by finding and swapping the largest item with the right-most item
  - ➢ simplify the while-loop condition since the itemToInsert will always be ≤ the right-most item

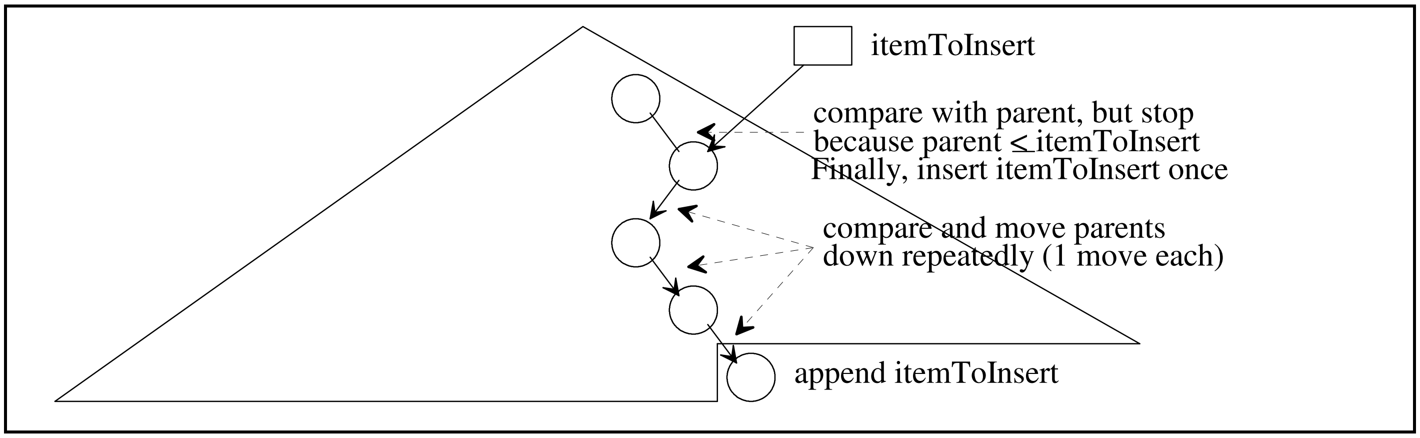
Include a timing program as in lab8.zip for your improved insertion sort that times the sorting algorithm several time with different initial orderings of 10,000 list items. The initial orderings of items are: descending order, ascending order, random order, and random order again to check for consistence. Write a brief report comparing the timings of your improved insertion sort to the original insertion sort from lab 8 -- include a table of your timings and a brief explanation of the timing results.

Part B: In lecture 17 and lab 8 we wrote a simple heap sort that used the BinHeap class. Unfortunately, my lab 8 timings for 400,000 random integers where: 10.5 seconds for heap sort, 3.37 seconds for merge sort, and 2.13 seconds for quick sort. A couple factors make this simple heap sort slower:

- this heap sort used a `BinHeap` object (e.g., `myHeap=BinHeap()`) which adds an extra layer of method calls (e.g., `myHeap.insert(item)` and `myHeap.delMin()`) which merge sort and quick sort did not have.
- this heap sort uses percUp (in `myHeap.insert(item)`) which swaps the inserted item repeatedly with its parent until the inserted item satisfies the heap-order property. Each swap in percUp requires three moves.


compare with parent, but stop
because parent ≤ item inserted or at root
compare and swap (3 moves each swap)
with parent repeatedly
item Inserted

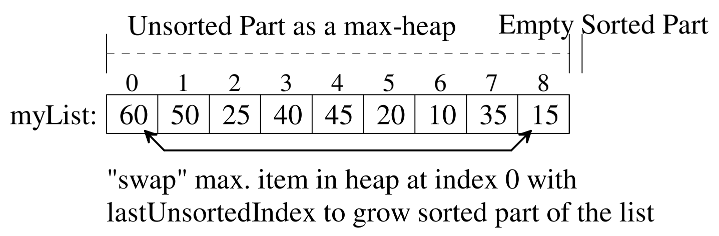
We can re-implement the percUp to reduce moves by repeatedly comparing the parents to the itemToInsert, but instead of swapping just move the parents down (1 move each). When we find the correct spot, insert the itemToInsert once into the heap.


itemToInsert
compare with parent, but stop
because parent ≤ itemToInsert
Finally, insert itemToInsert once
compare and move parents
down repeatedly (1 move each)
append itemToInsert

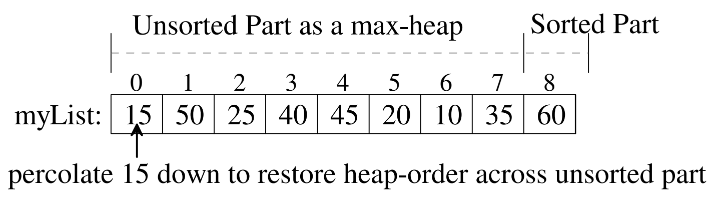
- this heap sort uses percDown (in `myHeap.delMin()`) which moves the last item to the root and swaps this item repeatedly with its smaller child until it satisfies the heap-order property. Each swap in percDown requires three moves. Similar to the above discussion for percUp, we could modify the percDown to reduce the moves by repeatedly comparing the last item (i.e, the parent) with the smallest child, but instead of swapping just move the child up (1 move each). When we find the correct spot, insert the last item once into the heap.
- this heap sort does not use the `buildHeap` method which takes as a parameter an unordered list and builds the heap from it by percDown the non-leaf items to build bigger and bigger heaps until all the items are in a single heap.

For Part B, I want you to modify and improve heap sort (still sorting in ascending order) by:

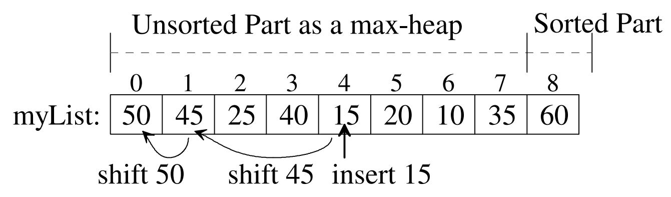
- eliminating the usage of the `BinHeap` class by using the list parameter (e.g., `myList`) to store items rearranged as a heap and "in-line" the code for heap methods right into the heap-sort code, so your heap-sort code calls no methods.
- improve the building of the initial "max-heap" containing all of the list items (called *heapifying the list*) by:
  - ➢ use index 0 to hold the root of the max-heap. This changes the relationship of parent and child indexes. A node at index i has: a left child at i*2+1, a right child at i*2+2, and a parent at (i-1)//2
  - ➢ use the concept employed in the "buildHeap" method in the BinHeap class: (1) think of the leaves as mini max-heaps of size 1 and (2) percolate down the non-leaves starting at index (len(myList)//2 - 1) through index 0 into the smaller heaps below to form bigger-and-bigger heaps.
  - ➢ while percolating down the non-leaves, don't repeatedly "swap" down a branch of the heap (e.g., 3 moves per swap), but instead "insert" the non-leaf item into the correct spot in the branch by shifting larger children up the branch (one move per shift) before inserting at the correct spot.
- use the max-heap of all list items in the unsorted part to perform a selection-sort like sort. After heapifying the list in the above step we have:



Unsorted Part as a max-heap
Empty Sorted Part
0 1 2 3 4 5 6 7 8
myList: 60 50 25 40 45 20 10 35 15
"swap" max. item in heap at index 0 with
lastUnsortedIndex to grow sorted part of the list



Unsorted Part as a max-heap
Sorted Part
0 1 2 3 4 5 6 7 8
myList: 15 50 25 40 45 20 10 35 60
percolate 15 down to restore heap-order across unsorted part



Unsorted Part as a max-heap
Sorted Part
0 1 2 3 4 5 6 7 8
myList: 50 45 25 40 15 20 10 35 60
shift 50
shift 45
insert 15

Include a timing progam as in lab8.zip for your improved heap sort that times the sorting of a randomly generated list of a user-specified size. Write a brief report comparing the timings of your improved heap sort to the original heap sort from lab 8 on 100,000 items, 200,000 items, 400,000 items, and 800,000 items -- include a table of your timings and a brief explanation of the timing results.

SUBMISSION

Submit **ALL necessary files** to run your sorts and your timing "reports" for parts A and B as a single zipped file (called hw5.zip) electronically at:

https://www.cs.uni.edu/~schafer/submit/which_course.cgi