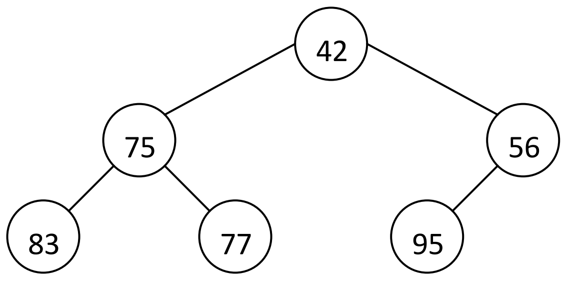
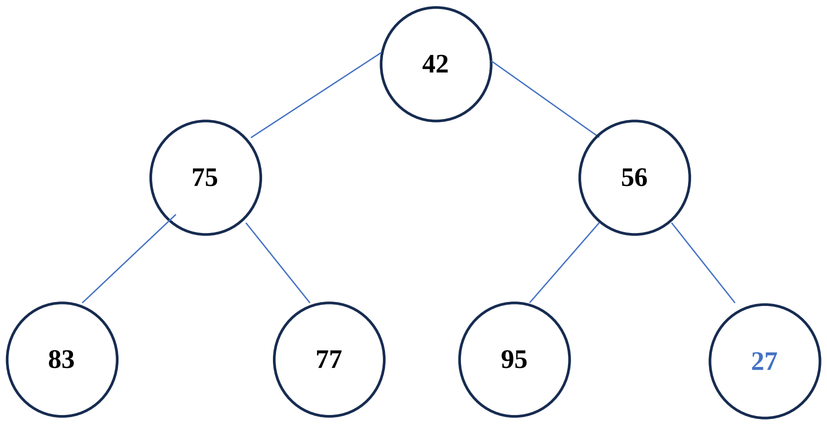
CS526 O2
Homework Assignment 4

Problem 1 (10 points). Consider the following heap, which shows integer keys in the nodes:

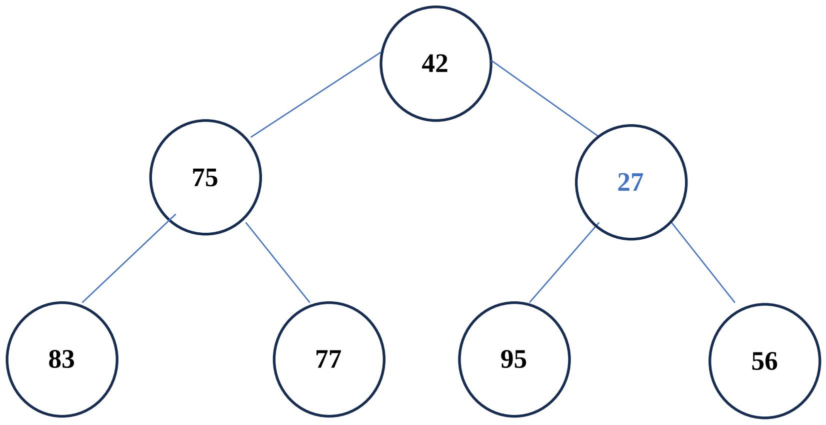


Show the resulting tree if you add an entry with key = 27 to the above tree? You need to describe, step by step, how the resulting tree is generated.

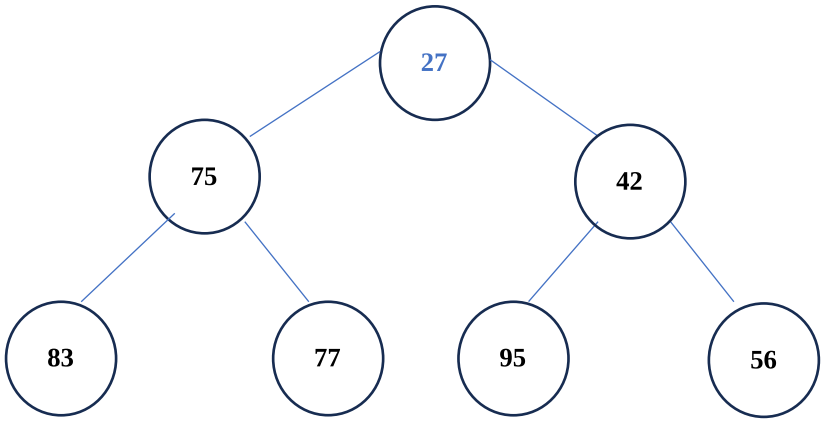
Step 1: New entry with key = 27 is added to the end of heap



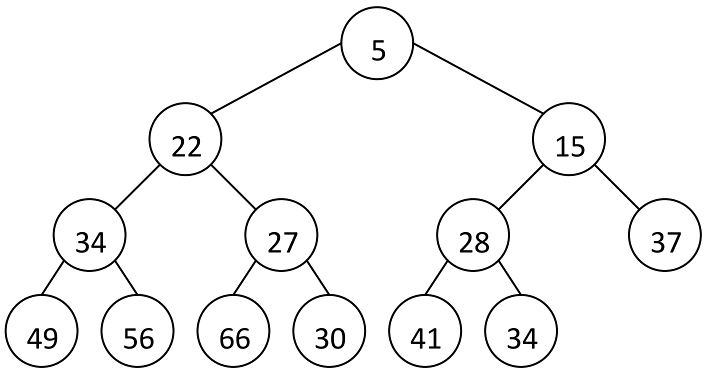
Step 2: Since 27 < 56, Entry with key = 27 is swapped with its parent with key = 56



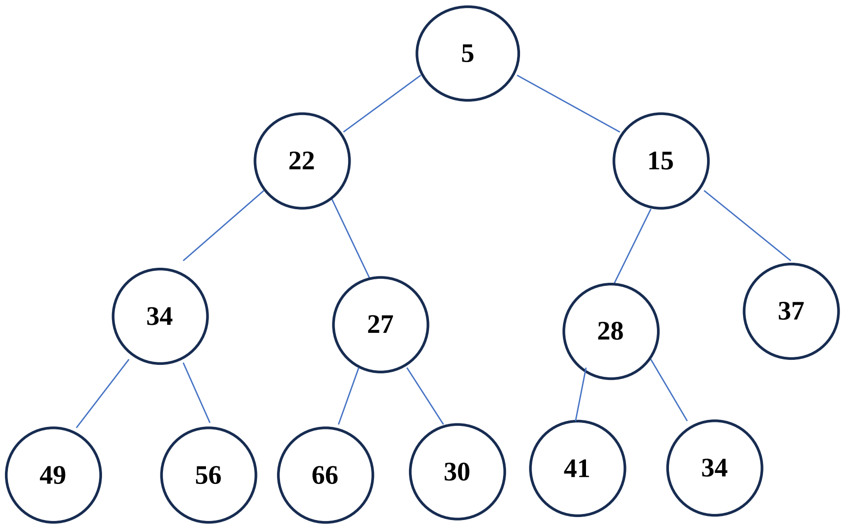
<u>Step 3:</u> Since 27 < 42, Entry with key = 27 is swapped with its parent with key = 42. We reached the root. So, stop and the heap below is the final heap.



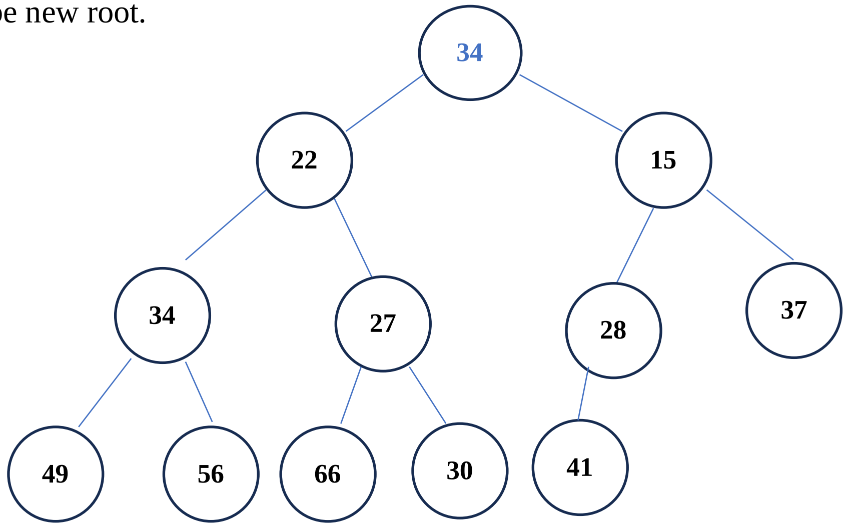
Problem 2 (10 points). Consider the following heap, which shows integer keys in the nodes:



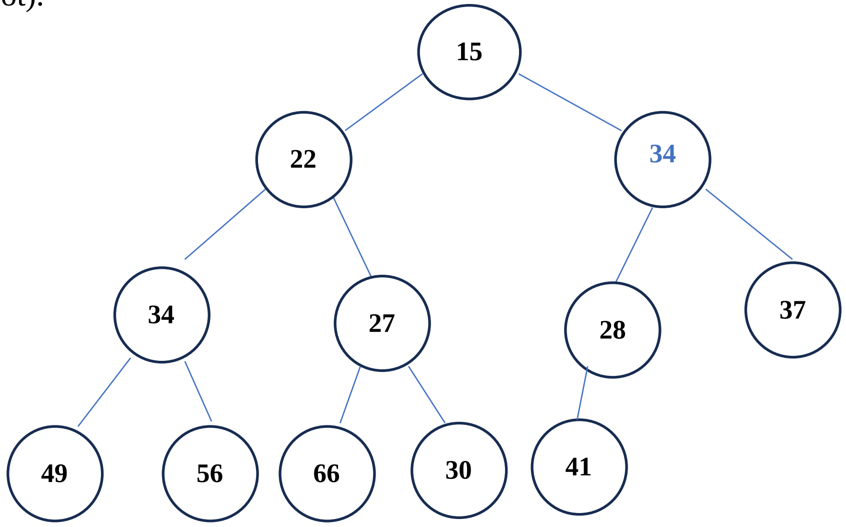
Suppose that you execute the *removeMin*( ) operation on the above tree. Show the resulting tree. You need to describe, step by step, how the resulting tree is generated.



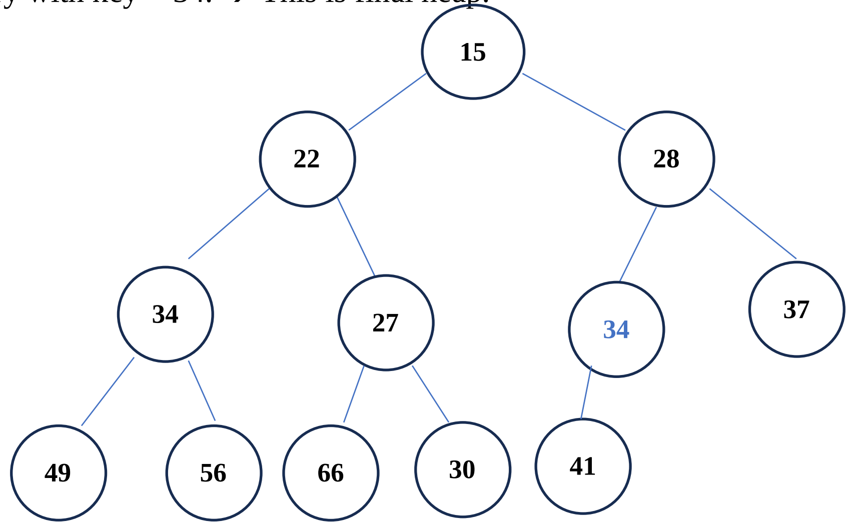
Step 1: When run removeMin() the Root entry with key = 5 is removed. The last entry with key 34 is moved up to be new root.



Step 2: Right child with key =15 is smaller key → Entry with key = 15 is swapped with Entry with key = 34 (new root).



Step 3: Left child with key =28 (of entry with key = 34) is smaller key → Entry with key = 28 is swapped with entry with key = 34. → This is final heap.



Problem 3 (10 points). This problem is about the chaining method we discussed in the class. Consider a hash table of size N = 11. Suppose that you insert the following sequence of keys to an initially empty hash table. Show, step by step, the content of the hash table.

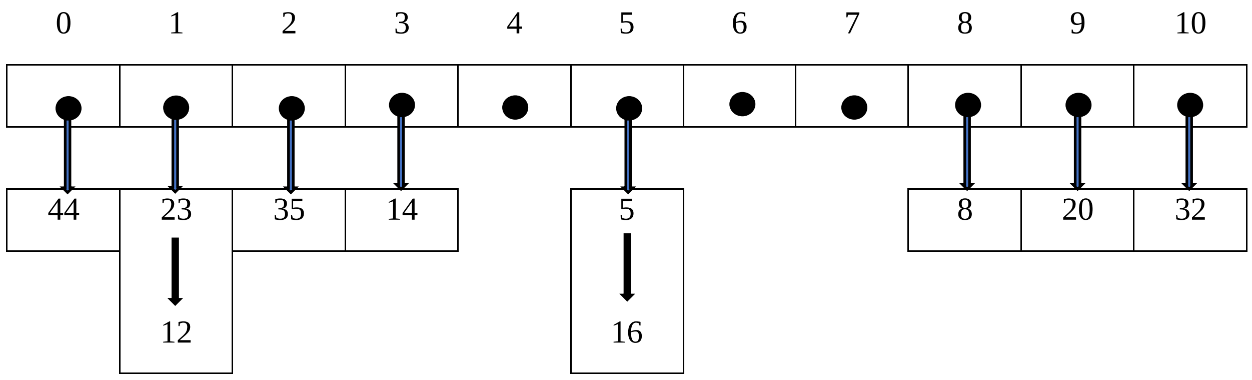
Sequence of keys to be inserted: <5, 8, 44, 23, 12, 20, 35, 32, 14, 16>

Step 1: Apply hash function to each key to determine the location in the array where each entry will be stored:

Hash function h =key mod 11

- h(5) = 5 mod 11 = 5
- h(8) = 8 mod 11 = 8
- h(44) = 44 mod 11 = 0
- h(23) = 23 mod 11 = 1
- h(12) = 12 mod 11 = 1
- h(20) = 20 mod 11 = 9
- h(35) = 35 mod 11 = 2
- h(32) = 32 mod 11 = 10
- h(14) = 14 mod 11 = 3
- h(16) = 16 mod 11 = 5

Step 2: there are keys 5, 16 are all mapped to index 5. Keys 23, 12 are all mapped with index 1



Problem 4 (10 points). This problem is about linear probing method we discussed in the class. Consider a hash table of size N = 11. Suppose that you insert the following sequence of keys to an initially empty hash table. Show, step by step, the content of the hash table.

Sequence of keys to be inserted: <5, 8, 44, 23, 12, 20, 35, 32, 14, 16>

Step 1: Apply hash function to each key to determine the location in the array where each entry will be stored:

Hash function h =key mod 1 where N =11. In the linear probing if A[j] is empty, then the entry is stored in that slot. If that slot is already occupied by another entry, then the next bucket, A[j+1] is probed to see whether it is available, we continue check like that using A[(j+i) mode N] for I = 0,1,2,.., N-1 until we find the empty slot

- h(5) = 5 mod 11 = 5 → A[h(5)] = A[5]
- h(8) = 8 mod 11 = 8 → A[h(8)] = A[8]
- h(44) = 44 mod 11 = 0 → A[h(44)] = A[0]
- h(23) = 23 mod 11 = 1 → A[h(23)] = A[1]
- h(12) = 12 mod 11 = 1 → A[h(12)] = A[1] but A[1] is occupied. So, the next slot, A[2] is probed
- h(20) = 20 mod 11 = 9 → A[h(20)] = A[9]
- h(35) = 35 mod 11 = 2 → A[h(35)] = A[2] but A[2] is occupied. So, the next slot, A[3] is probed
- h(32) = 32 mod 11 = 10 → A[h(32)] = A[10]
- h(14) = 14 mod 11 = 3 → A[h(14)] = A[3] but A[3] is occupied. So, the next slot, A[4] is probed
- h(16) = 16 mod 11 = 5 → A[h(16)] = A[5] but A[5] is occupied. So, the next slot, A[6] is probed

0	1	2	3	4	5	6	7	8	9	10
44	**23**	**12**	**35**	**14**	**5**	**16**		**8**	**20**	**32**

Problem 5 (10 points). Suppose that your hash function resolves collisions using open addressing with double hashing, which we discussed in the class. The double hashing method uses two hash functions *h* and *h*'.

Assume that the table size *N* = 13, *h*(*k*) = *k* mod 13, *h'*(*k*) = 1 + (*k mod* 11), and the current content of the hash table is:

0	1	2	3	4	5	6	7	8	9	10	11	12
		2	29		18			21	48	15		

If you insert *k* = 16 to this hash table, where will it be placed in the hash table? You must describe, step by step, how the location of the key is determined.

h'(*k*) = 1 + (*16 mod* 11) = 1 + 5 = 6
h(*k*) = *k* mod 13 = 16 mod 13 = 3
A[(*h*(*k*) + i.*h'*(*k*)) mod 13]
i=0 → A[3] is already occupied

i=1 → A[(3+1*6) mod 13] = A[9] is already occupied
i=2 → A[(3+2*6) mod 13] = A[2] is already occupied
i=3 → A[(3+3*6) mod 13] = A[8] is already occupied
i=4 → A[(3+4*6) mod 13] = A[1] is empty and can probe

0	1	2	3	4	5	6	7	8	9	10	11	12
	16	2	29		18			21	48	15		

Problem 6 (50 points). The goal of this problem is to give students an opportunity to observe differences among three data structures in Java – Java's *HashMap*, *ArrayList*, *LinkedList* – in terms of insertion time and search time.

Students are required to write a program that implements the following pseudocode:

```
create a HashMap instance myMap
create an ArrayList instance myArrayList
create a LinkedList instanc myLinkedList

Repeat the following 10 times and calculate average total insertion time and average total
search time for each data structure

        generate 100,000 distinct random integers in the range [1, 1,000,000] and store them in
        the array of integers insertKeys[ ]

        // begin with empty myMap, myArrayList, and myLinkedList each time

        // Insert keys one at a time but measure only the total time (not individual insert  //
        time)
        // Use put method for HashMap, e.g., myMap.put(insertKeys[i], i)
        // Use add method for ArrayList and LinkedList

        insert all keys in insertKeys [ ] into myMap and measure the total insert time
        insert all keys in insertKeys [ ] into  myArrayList and measure the total insert time
        insert all keys in insertKeys [ ] into myLinkedList and measure the total insert time

        generate 100,000 distinct random integers in the range [1, 2,000,000] and store them in
        the array searchKeys[ ].

        // Search keys one at a time but measure only total time (not individual search //
        time)
        // Use containsKey method for HashMap
        // Use contains method for ArrayList and Linked List

        search myMap for all keys in searchKeys[ ] and measure the total search time search
        myArrayList for all keys in searchKeys[ ] and measure the total search time  search
        myLinkedList for all keys in searchKeys[ ] and measure the total search time
```

Print your output on the screen using the following format:

```
Number of keys = 100000

HashMap average total insert time = xxxxx
ArrayList average total insert time = xxxxx
LinkedList average total insert time = xxxxx

HashMap average total search time = xxxxx
ArrayList average total search time = xxxxx
LinkedList average total search time = xxxxx
```

You can generate *n* random integers between 1 and N in the following way:

```
Random r = new Random(System.currentTimeMillis() );
for i = 0 to n – 1
    a[i] = r.nextInt(N) + 1
```

When you generate random numbers, it is a good practice to reset the seed. When you first create an instance of the Random class, you can pass a seed as an argument, as shown below:

```
Random r = new Random(System.currentTimeMillis());
```

You can pass any long integer as an argument. The above example uses the current time as a seed.

Later, when you want to generate another sequence of random numbers using the same Random instance, you can reset the seed as follows:

```
r.setSeed(System.currentTimeMillis());
```

You can also use the *Math.random*( ) method. Refer to a Java tutorial or reference manual on how to use this method.

We cannot accurately measure the execution time of a code segment. However, we can estimate it by measuring an elapsed time, as shown below:

```
long startTime, endTime, elapsedTime;
startTime = System.currentTimeMillis();
// code segment
endTime = System.currentTimeMillis();
elapsedTime = endTime - startTime;
```

We can use the *elapsedTime* as an estimate of the execution time of the code segment. Note that if the elapsed time is similar for the three data structures, you may need a more precise measure using System.nanoTime() or something similar.

Name the program *Hw4_P6.java*.

Deliverable

You need to submit the following files:

- *Hw4.p1_p5pdf*: This file must include:
- Answers to problems 1 through 5.
- Discussion/observation of Problem 6: This part must include what you observed and learned from this experiment and it must be "substantive."
- *Hw4_p6.java*
- Other files, if any.

Combine all files into a single archive file and name it *LastName_FirstName_hw4.EXT*, where *EXT* is an appropriate archive file extension, such as *zip* or *rar*.

Grading

Problem 1 through Problem 5:

- For each problem, up to 6 points will be deducted if your answer is wrong.

Problem 6:

- There is no one correct output. As far as your output is consistent with generally expected output, no point will be deducted. Otherwise, up to 20 points will be deducted.
- If your conclusion/observation/discussion is not substantive, points will be deducted up to 5 points.
- If there are no sufficient inline comments in your program, points will be deducted up to 5 points.