1) 10 Points

Compute an appropriately tight O (big-O) bound on the running time of each code fragment, in terms of *n*. Assume integer arithmetic. Circle your answer for each fragment.

a)

```
for(i = 0; i < n; i++) {
      for(j = 0; j < n; j++) {
           for(k = 0; k < i * j; k++) {
                sum++;
           }
      }
}
```

$\underbrace{i * j}_{n^2}$

$O(n^4)$

b)

```
for(i = 1; i < n; i = i * 2) {    log n
      for(j = 1; j < i; j++) {    n
           sum++;
      }
}
```

$O(n \log n)$

Actually, this is $O(n)$

It executes $2n-1$ increments

c)

```
for(i = 0; i < n; i++) { n
      myArray = new array[i];
      for(j = 0; j < i; j++) {  n
           myArray[j] = random();
      }
      mergeSort(myArray);  n log n
}
```

$O(n^2 \log n)$

d)

```
for(i = 0; i < n; i++) { n
      tree = new UnbalancedBinarySearchTree();
      for(j = 0; j < n; j++) {n
           tree.insert(j); n
      }
}
```

"list" tree results

$O(n^3)$

e)

```
tree = new AVLTree();
for(i = 0; i < n; i++) { n
      for(j = 0; j < n; j++) {  n
           tree.insert(random());
      }
}
```

inserts n^2 items at log cost for each

$O(n^2 \log n)$

2) 10 points

This problem asks you to describe how to perform queue operations in O(1) time for a **link-based** implementation of a **queue**. Declare variables used by your implementation (e.g., any pointers maintained in the queue). Provide pseudocode for the requested operations (precise English is also acceptable, but pseudocode will be more concise). Finally, show the state of your data structure after the given operations (including the values of the variables you declared in (a) and manipulated in (b) and (c)).

Assume a **`node`** class containing fields **`value`** and **`next`**.

For an **link-based** implementation of a queue with O(1) enqueue and O(1) dequeue:

a) Declare any variables used by your implementation:

```
front = front of queue
back = back of queue
```

b) Describe how to implement O(1) enqueue:

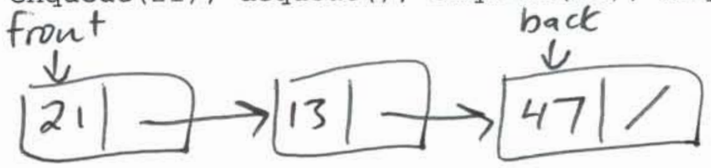
```
if front is null
    front = back = new node (value, null)
else
    back -> next = new node (value, null)
    back = back -> next
```

c) Describe how to implement O(1) dequeue:

```
if Front is null
    throw exception
else
    tmp = Front
    front = Front -> next
    if Front is null  } not required, just here for
        back = null   } cleanliness of variable state
    return tmp -> value
```

d) Draw your data structure after executing the following operations. You only need to show a total of one queue (the one which exists after all five operations complete).

`enqueue(54); enqueue(21); dequeue(); enqueue(13); enqueue(47);`



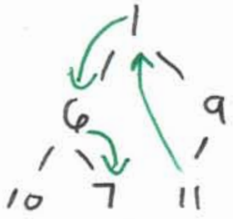
3) ~~15~~ 10 Points

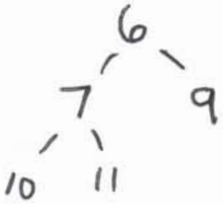
Consider this binary min-heap:

```
              1
            /   \
          6       9
        /   \   /
      10     7 11
```

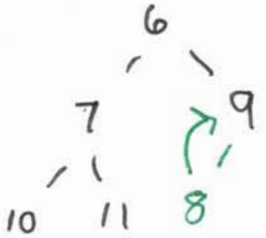
Perform the following operations in order, drawing the result after each operation and using it as the starting point for the next operation. You only need to show the result of the operation, but showing your work will allow partial credit in case of error. If the space here is insufficient, use the back of this sheet (clearly labeling your work). Circle the result of each operation so we can distinguish it from intermediate work.

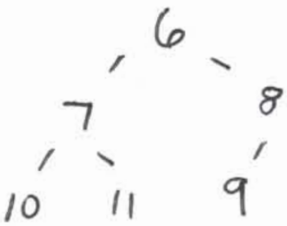
a) DeleteMin



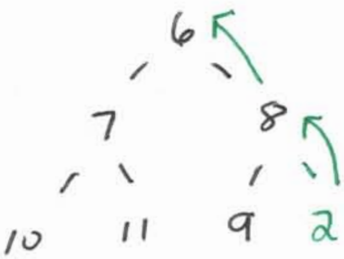


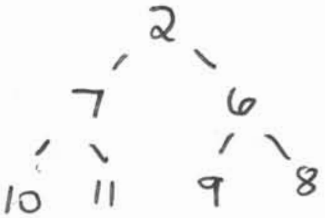
b) Insert 8





c) Insert 2





d) Draw an efficient array-based representation of your final heap from step *c*.

size

7	2	7	6	10	11	9	8
0	1	2	3	4	5	6	7

e) In your array-based representation, what is the index of:

the parent of the node at index *i*:

$i / 2$

the left child of the node at index *i*:

$2i$

the right child of the node at index *i*:

$2i + 1$

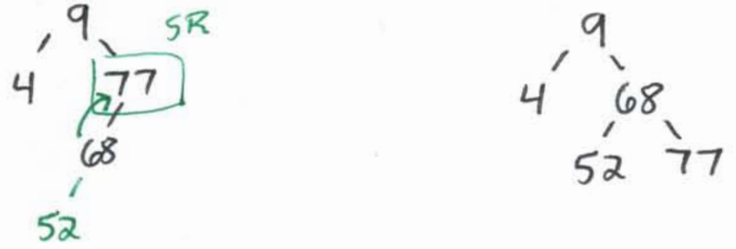
4) 10 Points

Consider this AVL tree:

```
      9
    /   \
   4     77
        /
      68
```

Perform the following operations in order, drawing the result after each operation and using it as the starting point for the next operation. You only need to show the result of the operation, but showing your work will allow partial credit in case of error. Circle the result of each operation so we can distinguish it from intermediate work.

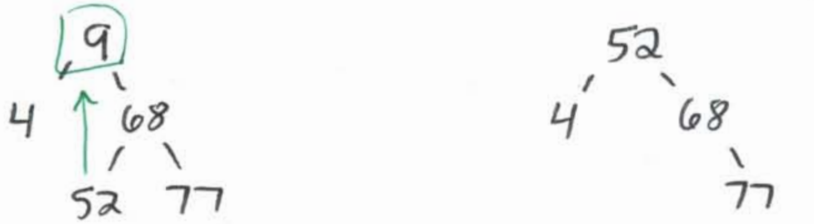
a) Insert 52



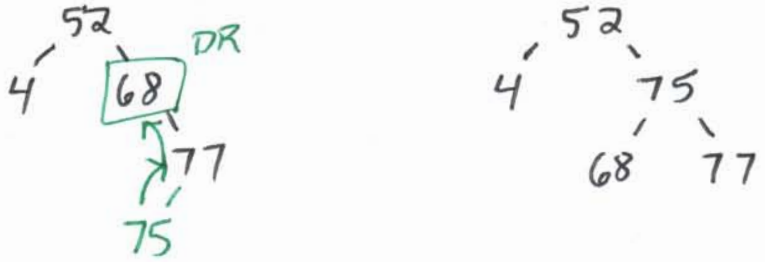
b) Delete 9 *We did not cover general AVL delete. But if you did the previous operation correctly, you can delete this without creating an imbalance.*

use successor

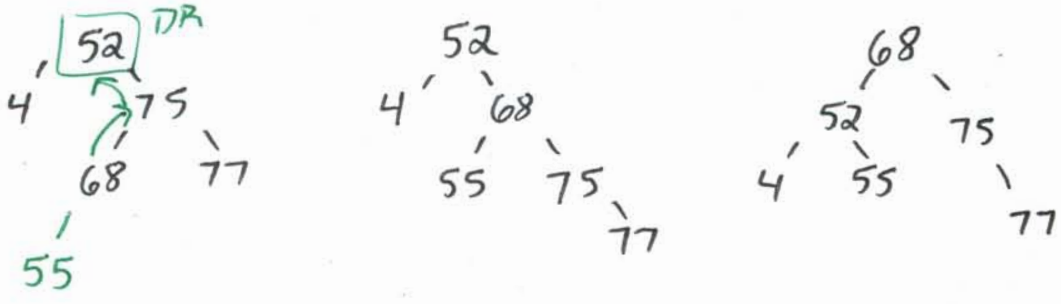
predecessor leads to an imbalance



c) Insert 75



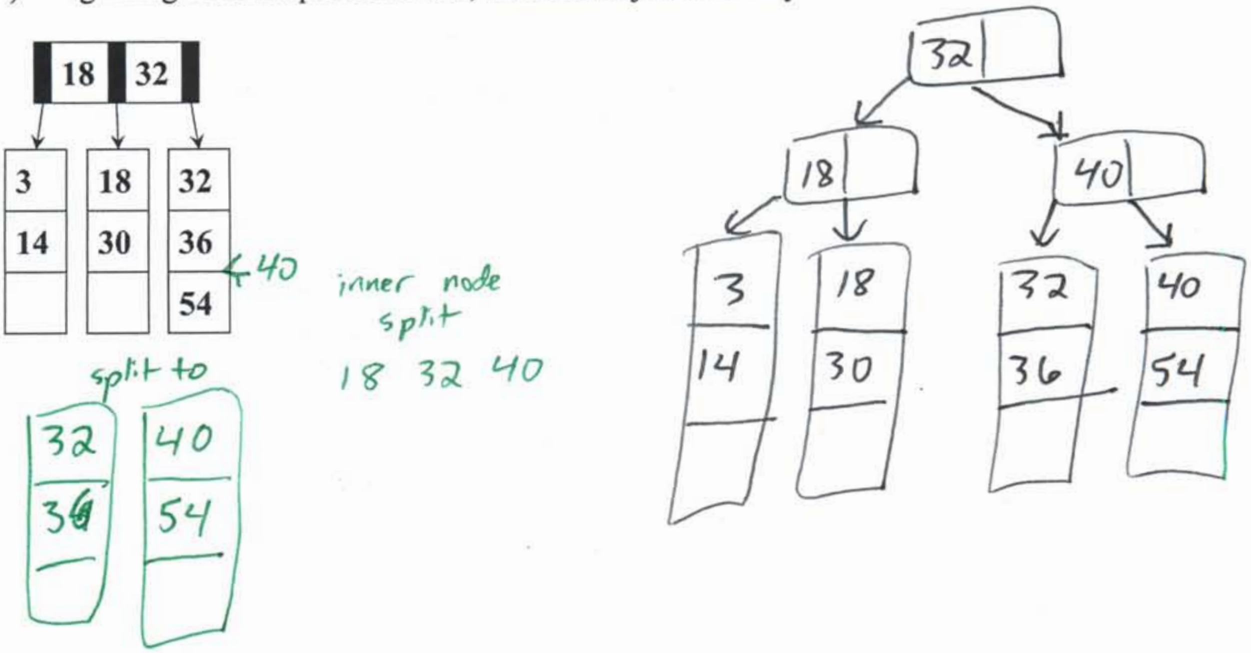
d) Insert 55



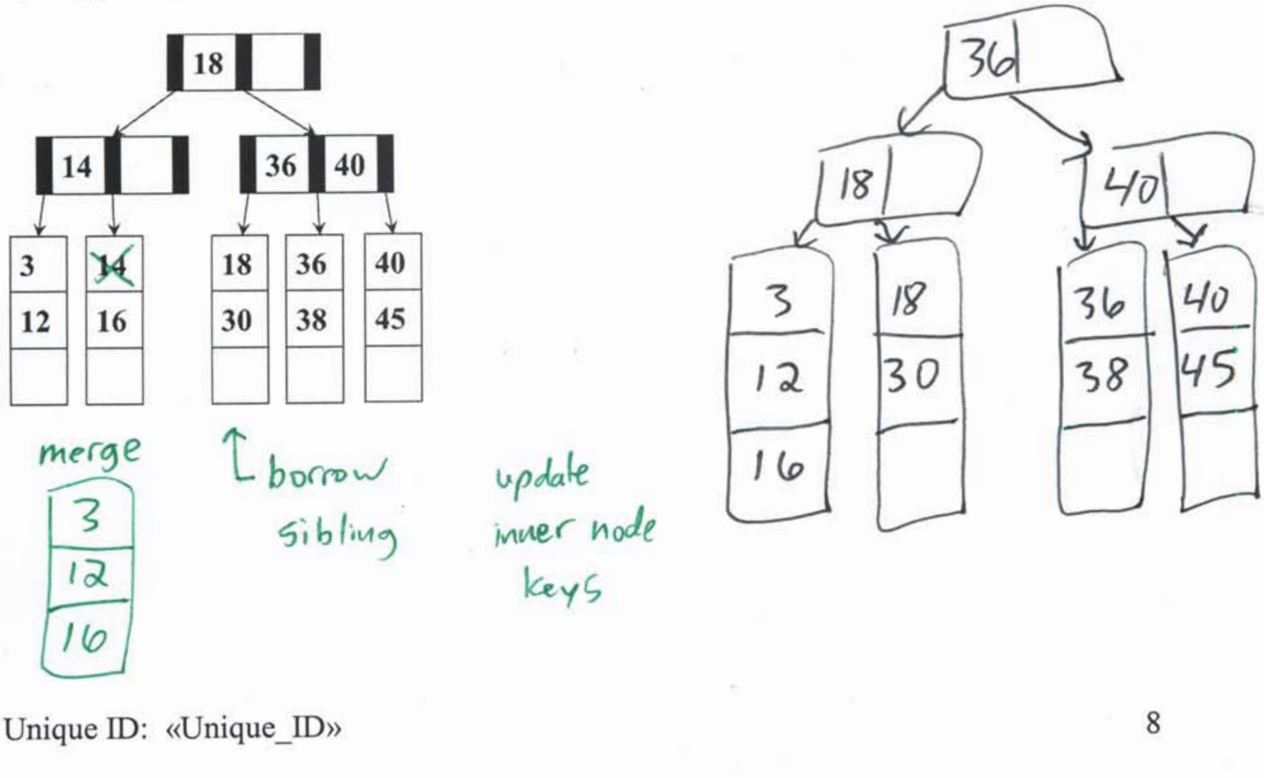
5) 10 points

Perform the following B-Tree operations. Use the algorithms presented in lecture, homework, and the text (i.e., for a B+ Tree). Do not adopt during insertion (although possible, this was not the presented algorithm). You only need to show the result of the operation, but showing your work will allow partial credit in case of error. Circle the result of each operation so we can distinguish it from intermediate work.

a) Beginning with the provided tree, insert an object with key 40.



b) Beginning with the provided tree, delete the object with key 14.



Answer the following questions regarding a B-Tree implementation. Answer based on the data structure presented in lecture, homework, and the text (i.e., based on a B+ Tree).

Your brilliant teaching assistants have implemented a B-Tree in order to better track and correct your instructor's many mistakes. They give a name to each mistake and use the name as a key to store information about when the mistake was made, who is responsible for correcting it, and information about their progress. The parameters of the tree are:

Pointer Size = 8 bytes
Key Size = 12 bytes
Data Size = 52 bytes
M = 13
L = 4

c) Assuming these parameters were appropriately chosen to fit within a disk block, what is the likely size of disk blocks on the machine where this implementation is deployed? Give **a numeric answer** and **a short justification**. This justification should be based on one or more equations using the above parameter values.

Internal Node Size $Mp + (M-1)k$

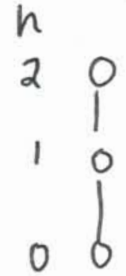
$13 \cdot 8 + 12 \cdot 12 = 104 + 144 = 248$ bytes

Leaf Node Size $L(k+d)$

$4(12+52) = 4(64) = 256$ bytes

max of the two is 256 bytes

d) Given these B-Tree parameters, what is the maximum number of items that can be stored in a tree of height 2? Give **a numeric answer**. Show your work for partial credit in case of an arithmetic error, but we do not expect a general equation.



root has M children

each has M children

each leaf has L elements

MML

$13 \cdot 13 \cdot 4$

$169 \cdot 4$

$400 + 240 + 36 = 676$

6) ~~8)~~ 10 points

Using a hash table of size 10 with **open addressing** and **quadratic probing**, perform the following sequences of actions and answer the associated questions. Objects can be represented simply by their hashcodes. Do not grow the size of the hashtable.

a) Insert a sequence of objects with hashcodes 22, 42, 47, 12, 37, 57.

0	1	2	3	4	5	6	7	8	9
	57	22	42			12	47	37	

b) What is the load factor of your table after (a)?

6/10 .6

c) Can we guarantee that any additional insert will succeed? Why or why not? Make a simple and precise argument based on your knowledge of hashtables. An argument based on brute-force hashing at every location is not appropriate.

No. Quadratic probing is guaranteed to succeed only if table size is prime and load factor is less than .5. Neither is true.

d) Delete the object with hashcode 47.

0	1	2	3	4	5	6	7	8	9
	57	22	42			12	X	37	

e) ~~d)~~ Insert a sequence of objects with hashcodes 53, 83.

0	1	2	3	4	5	6	7	8	9
	57	22	42	53		12	83	37	

7) 10 Points

a) Show the state of the array **after each execution of the outer loop** of a **selection sort**. Make the minimal number of swaps necessary in each iteration.

27	64	44	38	40	9
9	64	44	38	40	27
9	27	44	38	40	64
9	27	38	44	40	64
9	27	38	40	44	64
9	27	38	40	44	64
9	27	38	40	44	64

b) Show a quicksort **partition** of this array. Choose a pivot using a median of the first, middle, and last elements of the array. Show the array after each swap conducted in the course of the partition. Make the minimal number of necessary swaps.

The provided number of boxes does not correspond to the number of necessary swaps. A correct answer will complete the partition and leave some of these boxes blank.

After completing the partition, indicate which subarrays will be recursively quicksorted (e.g., by circling each of them and writing 'recurse' next to each of your circles).

pivot on	27	64	44	38	40	9	52	
median(27,38,52)	38	64	44	27	40	9	52	move pivot out of way
⇓ pivot on 38	38	9	44	27	40	64	52	
	38	9	27	44	40	64	52	
	27	9	38	44	40	64	52	swap pivot back
recurse								recurse