COS 226	Algorithms and Data Structures	Spring 2023

Midterm

This exam has 10 questions worth a total of 60 points. You have 80 minutes.

Instructions. This exam is preprocessed by computer. Write neatly, legibly, and darkly. Put all answers (and nothing else) inside the designated spaces. *Fill in* bubbles and checkboxes completely: ● and ■. To change an answer, erase it completely and redo.

Resources. The exam is closed book, except that you are allowed to use a one page reference sheet (8.5-by-11 paper, one side, in your own handwriting). No electronic devices are permitted.

Honor Code. This exam is governed by Princeton's Honor Code. Discussing the contents of this exam before the solutions are posted is a violation of the Honor Code.

Please complete the following information now.

Name:

NetID:

Exam room: ○ McCosh 50 ○ McCosh 62 ○ Other

Precept:

P01	P02	P02A	P03	P03A	P03B	P04	P04A	P05
○	○	○	○	○	○	○	○	○

"I pledge my honor that I will not violate the Honor Code during this examination."

Signature

1. **Initialization. (1 point)**

   In the spaces provided on the front of the exam, write your name and NetID; fill in the bubble for your exam room and the precept in which you are officially registered; write and sign the Honor Code pledge.

2. **Memory. (5 points)**

   Consider the following implementation of a BST (with `int` keys and `double` values):

```
public class BST {
    private Node root;          // root node
    private int n;              // number of key-value pairs

    private static class Node {
        private int key;        // key
        private double value;   // link to parent
        private int count;      // number of nodes in subtree
        private Node left;      // left subtree
        private Node right;     // right subtree

    }
    ...
}
```

   Use our 64-bit memory cost model to answer the following two questions:

   (a) How much memory does each `Node` object use? Count all memory allocated when a `Node` object is constructed.

   Write your answer in the box below.

	bytes

   (b) How much memory does a `BST` object use as a function of the number n of key–value pairs in the BST? Count all referenced memory.

   Use tilde notation to simplify your answer and write it in the box below.

~		bytes

3. **Data structures. (6 points)**

(a) Consider the following *parent-link* representation of a *weighted quick union* (link-by-size) data structure.

	0	1	2	3	4	5	6	7	8	9
`parent[]`	4	5	4	5	?	5	2	5	8	5

Which of the following values could be `parent[4]`?

Fill in all checkboxes that apply.

☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9

(b) Consider the following *binary heap* representation of a *maximum-oriented priority queue*, with `pq[0]` unused.

pq[]	–	65	50	35	?	30	20	15	25	40
	0	1	2	3	4	5	6	7	8	9

Which of the following values could be `pq[4]`?

Fill in all checkboxes that apply.

☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
20	25	30	35	40	45	50	55	60	65

4. **Five sorting algorithms. (5 points)**

The leftmost column contains an array of 24 integers to be sorted; the rightmost column contains the integers in sorted order; the other columns are the contents of the array at some intermediate step during one of the five sorting algorithms listed below.

Match each algorithm by writing its letter in the box under the corresponding column. Use each letter exactly once.

71	13	13	15	15	71	13
24	24	15	17	17	68	15
51	51	17	24	24	58	17
83	31	20	30	30	66	20
92	68	24	51	48	57	24
58	58	29	52	51	52	29
90	29	30	58	52	51	30
98	20	31	71	58	32	31
15	15	32	83	66	55	32
30	30	48	90	71	15	48
17	17	51	92	75	29	51
52	52	52	98	77	30	52
75	57	75	13	83	20	55
77	55	77	32	90	17	57
48	48	90	48	92	48	58
66	66	66	66	98	24	66
32	32	92	75	32	31	68
13	71	71	77	13	13	71
55	77	55	55	55	75	75
57	75	57	57	57	77	77
20	98	83	20	20	83	83
29	90	58	29	29	90	90
68	92	68	68	68	92	92
31	83	98	31	31	98	98
A						G

A. Original array

B. Selection sort

C. Insertion sort

D. Mergesort (*top-down*)

E. Quicksort (*standard, no shuffle*)

F. Heapsort

G. Sorted array

5. **Analysis of algorithms and sorting. (6 points)**

Consider an array that contains n Bs, followed by $2n$ As, followed by n Bs, where n is a power of 2. For example, here is the array when $n = 4$:

```
B B B B A A A A A A A A B B B B
```

How many *compares* does each sorting algorithm (standard algorithm, from the textbook) make as a function for n? Note that the length of the array is $4n$, not n.

For each sorting algorithm, fill in the best matching bubble.

(a) Selection sort.

○ $\sim \frac{1}{2}n^2$ ○ $\sim n^2$ ○ $\sim 2n^2$ ○ $\sim 4n^2$ ○ $\sim 8n^2$

(b) Insertion sort.

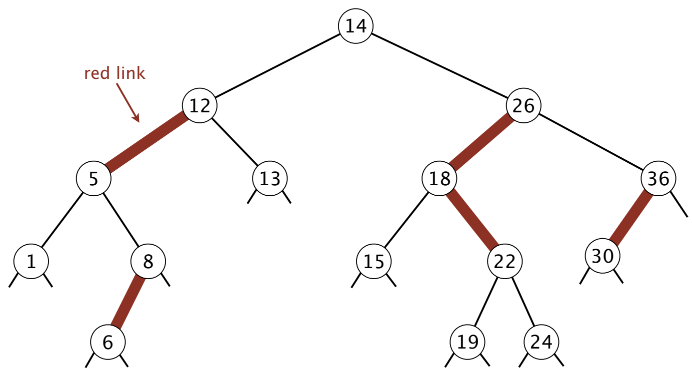
○ $\sim \frac{1}{2}n^2$ ○ $\sim n^2$ ○ $\sim 2n^2$ ○ $\sim 4n^2$ ○ $\sim 8n^2$

(c) 3-way quicksort.

○ $\sim 4n$ ○ $\sim 6n$ ○ $\sim n \log_2 n$ ○ $\sim 4n \log_e n$ ○ $\sim 4n^2$

6. **Left-leaning red–black BSTs. (6 points)**

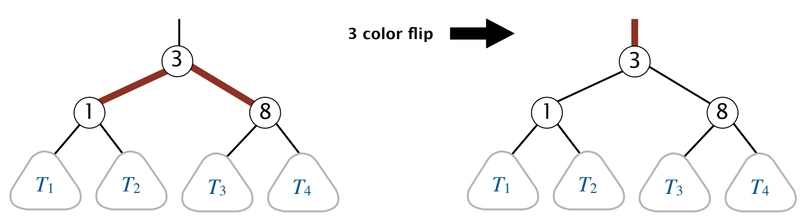
The following BST that satisfies perfect black balance, but violates the color invariants:

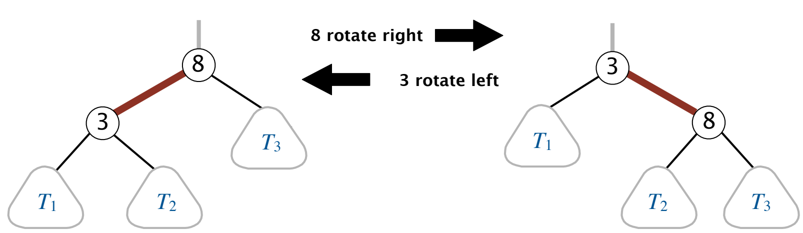


Give a sequence of 4 elementary operations that restores the color invariants.

	operation 1	*operation* 2	*operation* 3	*operation* 4
key				
color flip	○	○	○	○
rotate left	○	○	○	○
rotate right	○	○	○	○

Examples of elementary operations (for reference):





7. **Properties of algorithms and data structures. (8 points)**

Determine the *minimum* and *maximum* value of each quantity as a function of n. Assume that each algorithm is the standard version from the textbook.

For each quantity on the left, write the two letters corresponding to its minimum and maximum values. You may use each letter once, more than once, or not at all.

min	*max*	
		Number of key compares to *binary search* for a key in a *sorted array* that contains n keys.
		Number of key compares to *delete-the-maximum* in a *ternary (3-way) heap* that contains n keys.
		Number of key compares to *insert* a key–value pair into a *binary search tree* that contains n key–value pairs.
		Number of key compares to *insert* a key–value pair into a *2–3 search tree* that contains n key–value pairs.

A. $\Theta(1)$

B. $\sim \frac{1}{3}\log_3 n$

C. $\sim \frac{1}{2}\log_2 n$

D. $\sim \log_3 n$

E. $\sim \log_2 n$

F. $\sim 2\log_3 n$

G. $\sim 2\log_2 n$

H. $\sim 3\log_3 n$

I. $\sim 3\log_2 n$

K. $\Theta(n)$

.

8. **Why did we do that? (8 points)**

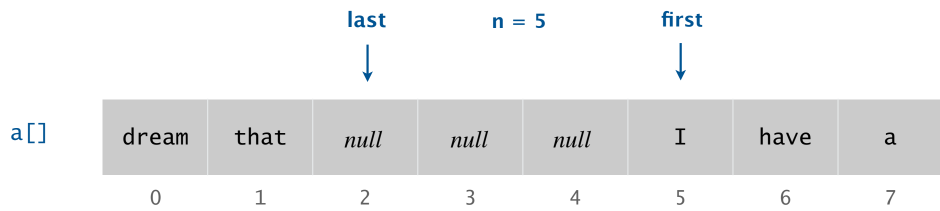
   For each design element below, identify whether it was an *important* choice (e.g., for correctness, performance, or some other useful property) or whether it was primarily an *arbitrary* choice.

   For each design element on the right, fill in the best matching bubble on the left.

Important	*Arbitrary*	
◯	◯	When implementing a *queue* with a singly linked list, implement *enqueue* by adding a new node *after the last node* in the linked list (instead of *before the first node*).
◯	◯	When finding the index of the smallest remaining element during *selection sort*, choose the *smallest index* of such an element if there are ties (instead of the *largest index*).
◯	◯	When computing the index of the middle element in *binary search*, use `(lo + hi) >>> 1` (instead of `(lo + hi) / 2`).
◯	◯	When comparing two equal keys during *mergesort*, copy the element from the the *left subarray* (instead of the one from the *right subarray*).
◯	◯	When *2-way partitioning* a subarray during quicksort, *stop* both the left and right scans on equal keys (instead of *skipping* over equal keys).
◯	◯	When quicksorting an array, recursively sort the left subarray *before* the right subarray (and not *after* the right subarray).
◯	◯	When inserting a key–value pair into a *2–3 tree* and splitting a temporary 4-node, move the *middle* key to its parent node (instead of moving the *smallest* key to its parent node).
◯	◯	When inserting a key–value pair into a *left-leaning red–black BST*, color the newly created node *red* (instead of *black*).

9. **Iteration. (5 points)**

Consider a *resizing-array* implementation of a queue, maintaining the elements in the array `a[]`; the index of the first item (least recently added) in the queue `first`; the index to one beyond the last item in the queue `last`; and the number of items in the queue `n`.



Complete the implementation of the following iterator.

```
private class MyIterator implements Iterator<Item> {
    private int i = [ 1 ] ;

    public boolean hasNext() {
        return [ 2 ] < [ 3 ] ;
    }

    public Item next() {
        Item item = a[ [ 4 ] % [ 5 ] ];
        i++;
        return item;
    }
}
```

A. `0`
B. `1`
C. `n`
D. `first`
E. `last`
F. `a.length`
G. `i`
H. `(i + first)`
I. `(i + last)`

For each numbered oval above, write the letter of the corresponding expression on the right in the space provided. You may use each letter once, more than once, or not at all.

1	2	3	4	5

10. **Data-type design. (10 points)**

Design a data type to implement a *middle queue*. A middle queue supports adding an item to either the front or back, along with removing (and returning) the item in the middle. (If there are an even number of items in the queue, remove the middle item closest to the front.)

`public class MiddleQueue<Item>`

	`MiddleQueue()`	*create an empty middle queue*
`void`	`addFront(Item item)`	*add the item to the front of the queue*
`void`	`addBack(Item item)`	*add the item to the back of the queue*
`Item`	`removeMiddle()`	*remove and return the item in the middle of the queue (midway between the front and back)*

Here are the performance requirements:

- The constructor and all instance methods must take $\Theta(1)$ time in the *worst case*.
- The amount of memory to store n items must be $\Theta(n)$, where n is the number of items in the queue.
- Partial credit for either $\Theta(\log n)$ time in the worst case or $\Theta(1)$ amortized time.

Here is an example:

```
MiddleQueue<Integer> queue = new MiddleQueue<>();   //  [ ]
queue.addBack("A");                                 //  [ A ]
queue.addBack("B");                                 //  [ A B ]
queue.addBack("C");                                 //  [ A B C ]
queue.addBack("D");                                 //  [ A B C D ]
queue.addBack("E");                                 //  [ A B C D E ]
queue.removeMiddle();                               //  [ A B D E ]      => C
queue.addFront("F");                                //  [ F A B D E ]
queue.removeMiddle();                               //  [ F A D E ]      => B
queue.removeMiddle();                               //  [ F D E ]        => A
```

Your answer will be graded for correctness, efficiency, and clarity (but not Java syntax). If your solution relies upon an algorithm or data structure from the course, do not reinvent it; simply describe how you are applying it.

(a) Using Java code, declare the instance variables (along with any supporting nested classes) that you would use to implement `MiddleQueue`. You may use any of the data types that we have considered in this course (either `algs4.jar` or `java.util` versions). You may also make modifications to these data types; if you do so, describe the modifications.

```
public class MiddleQueue<Item> {
```

(b) *Draw* the underlying data structures (such as resizing arrays, linked lists, or binary trees) for a middle queue containing the following seven items, inserted at the back (in that order): A, B, C, D, E, F, G. For linked data structures, draw all links.

(c) Give a concise English description of your algorithm for implementing `addFront()`.

(d) Give a concise English description of your algorithm for implementing `addBack()`.

(e) Give a concise English description of your algorithm for implementing `removeMiddle()`.

This page is intentionally blank. You may use this page for scratch work.