CS101 Algorithms and Data Structures
Fall 2022
Midterm Exam

Instructors: Dengji Zhao, Yuyao Zhang, Xin Liu, Hao Geng

Time: November 2nd 8:15-9:55

INSTRUCTIONS

Please read and follow the following instructions:

- You have 100 minutes to answer the questions.
- You are not allowed to bring any papers, books or electronic devices including regular calculators.
- You are not allowed to discuss or share anything with others during the exam.
- You should write the answer to every problem in the dedicated box **clearly**.
- You should write **your name and your student ID** as indicated on the top of **each page** of the exam sheet.

Name	
Student ID	
Exam Classroom Number	
Seat Number	
All the work on this exam is my own. **(Please copy this and sign)**	

THIS PAGE INTENTIONALLY LEFT BLANK

Name: ID:

1. (20 points) True or False

For each of the following statements, please judge whether it is **true(T) or false(F)**. **Write your answers in the following table**.

(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)
F	F	T	F	F	T	F	T	F	T

(a) (2') Accessing the k-th element in a linked list takes more than constant time, so we cannot apply Quick Sort or Merge Sort to an unsorted linked list.

(a) F

Solution: No random access does not matter: in the partition/divide step, we can allocate two sub-lists and append elements to the end of each; in the merge step of Merge Sort, we only access the head of each sub-list and append it to the end of the merged one.

(b) (2') If $f(n) = \omega(g(n))$, then $\log(f(n)) = \omega(\log(g(n)))$. $f(n)$ and $g(n)$ are both positive-valued.

(b) F

Solution: Consider counterexample $f(n) = n^2$ and $g(n) = n$.

(c) (2') The choice of hash function does not affect the load factor of a hash table. Assume all candidate hash functions share the same modulo as the table capacity.

(c) T

Solution: Load factor $\lambda = n/M$, where n is the number of elements in the table and M is the capacity of the table, or the modulo in hash function.

(d) (2') The reason why Merge Sort is stable is that both its best-case and worst-case time complexity are $\Theta(n \log n)$.

(d) F

Solution: We say a sorting algorithm is stable if the relative position of two elements with equal keys is preserved. It has nothing to do with time complexity.

(e) (2') In Insertion Sort, after m passes through the array, **the first m elements** are sorted in ascending order and these elements are the m smallest elements in the array.

(e) F

Solution: These m elements are a rearrangement of the first m elements in the original array, not the smallest m elements.

(f) (2') Assume we implement in-place Heap Sort with a min-heap in an array. After popping m times, **the last m elements** in the array are sorted in descending order and these elements are the m smallest elements in the array.

(f) T

(g) (2') When we Depth-First traverse a tree, a node with smaller depth will be visited before that with larger depth. Breadth-First Traversal has no such property.

(g) F

Solution: In BFS, nodes with smaller depth are visited before those with larger depth; in DFS, which node is visited first does not depend on its depth.

(h) (2') After Huffman coding, more frequently used symbols tend to have shorter binary code words and a shorter code word cannot be the prefix of a longer one.

(h) T

(i) (2') Given a lower bound a, an upper bound b and a binary search tree with n nodes, searching all the elements x in the tree such that $a \le x \le b$ takes $O(\log n)$ time.

(i) F

Solution: $O(n)$ in the worst case (a chain).

(j) (2') Building an AVL tree from an arbitrary unbalanced binary search tree with n nodes can be implemented in $O(n)$ time.

(j) T

Solution: First, obtain a sorted array from the in-order traversal of the BST, which is $O(n)$. Second, build an AVL tree from the sorted array recursively by taking the median element in the array as the root node and dividing the array into two sub-arrays, which is also $O(n)$.

2. (15 points) Single Choice

Each question has **exactly one** correct answer. **Write your answers in the following table**.

(a)	(b)	(c)	(d)	(e)
C	D	B	C	A

(a) (3') Which of the following statements about **asymptotic bounds** is TRUE?

A. $n^2 + \log n = o(n^2 + \log^2 n^2)$

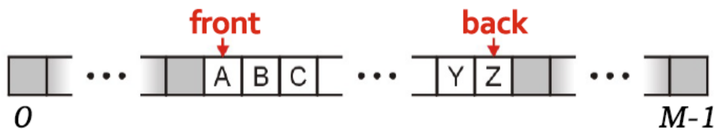
B. $\log(\log n) = \Omega(\sqrt{\log(n^{1/3})})$

C. If $T(n) = T(n/5) + T(7n/10) + 3n$, then $T(n) = O(n)$.

D. If $T(n) = 8T(n/2) + n^3$, then $T(n) = O(n^3)$.

Solution: A. $\lim_{x\to\infty} \frac{x^2+\log x}{x^2+\log^2 x^2} = \lim_{x\to\infty} \frac{x^2+\log x}{x^2+4\log^2 x} = 1$, so LHS $= \Theta$(RHS).
B. $\lim_{x\to\infty} \frac{\log(\log x)}{\sqrt{\log(x^{1/3})}} = \lim_{x\to\infty} \frac{\log(\log x)}{(\log(x)/3)^{1/2}} = \lim_{y\to\infty} \frac{\log y}{(y/3)^{1/2}} = 0$, so LHS $= o$(RHS).
C. The recursion tree will have $O(\log n)$ levels and the work in level i is $0.9^i \cdot 3n$, so the total work is $3n + 0.9 \cdot 3n + 0.9^2 \cdot 3n + \cdots + (0.9)^{\log n} \cdot 3n = O(n)$.
D. By Master Theorem, $\log_2 8 = 3$, so $T(n) = O(n^3 \log n)$.

(b) (3') Assume you implement a **queue with a two-ended array** ~~(the naïve version of circular array)~~(not a circular array) of capacity M. You have performed m push operations and n pop operations $(m, n < M)$ starting with an empty queue, as shown in the figure below. Which of the following statements is TRUE?



A. The index of the front pointer of the queue is m.

B. The index of the back pointer of the queue is m.

C. The number of elements in the queue is $m - n - 1$.

D. You can push $M - m$ more elements onto the queue at most.

Solution: A. pop n times and the front pointer moves n times (initially 0), so $\mathsf{front} = n$.
B. push m times and the back pointer moves m times (initially -1), so $\mathsf{back} = m - 1$.
C. m items are pushed and n items are popped, so $m - n$ items are remained. (or $\mathsf{back} - \mathsf{front} + 1 = (m-1) - n + 1 = m - n$).
D. $(M-1) - (\mathsf{back}+1) + 1 = (M-1) - ((m-1)+1) + 1 = M - m$

(c) (3') Given an array $\langle 1, 5, 4, 3, 2, 6 \rangle$, how many passes are needed to sort the array in ascending order with **Flagged Bubble Sort** (the implementation of Bubble Sort that stops immediately when no swaps happen in a pass)?

A. 3 B. 4 C. 5 D. 6

Solution: After the 1st pass: $\langle 1, 4, 3, 2, 5, 6 \rangle$
After the 2nd pass: $\langle 1, 3, 2, 4, 5, 6 \rangle$

After the 3rd pass: $\langle 1,2,3,4,5,6 \rangle$
No swap happens in the 4th pass, hence break.

(d) (3') Which of the following statements about **binary trees** is TRUE?

A. Let $\mathcal{F}_k$ be the number of nodes with depth k in a binary tree. Then $0 \leq \mathcal{F}_k \leq 2k$.

B. Let $\mathcal{G}_h$ be the number of nodes in a binary tree of height h. Then $h+1 \leq \mathcal{G}_h \leq 2^h + 1$.

C. Let $\mathcal{H}_n$ be the minimum height of a binary tree of n nodes. Then $\mathcal{H}_n = \lfloor \log_2 n \rfloor$.

D. If a binary tree of n nodes has the minimum height $\mathcal{H}_n$, then it is complete.

Solution: A. $0 \leq \mathcal{F}_k \leq 2^k$.
B. $h+1 \leq \mathcal{G}_h \leq 2^{h+1} - 1$. Consider the perfect binary tree.
C. In the worst case the tree is complete and the height of n-node complete tree is $\lfloor \log_2 n \rfloor$.
D. You can fill the deepest nodes from right to left.

(e) (3') Which of the following statements about **tree traversal** is TRUE?

A. The in-order traversal of an AVL tree is an ascending sequence.

B. The pre-order traversal of a binary search tree is an ascending sequence.

C. The in-order traversal of a min-heap is a descending sequence.

D. The post-order traversal of a max-heap is a descending sequence.

Solution: The in-order traversal of a BST is a sorted sequence and AVL is a special type of BST. Heap has no similar property because the relative size of left child and right child is uncertain.

3. (20 points) Multiple Choices

Each question has **one or more** correct answer(s). Select all the correct answer(s). For each question, you will get 0 points if you select one or more wrong answers, but you will get 2.5 points if you select a non-empty subset of the correct answers. **Write your answers in the following table.**

(a)	(b)	(c)	(d)
ABC	ACD	BD	ACD

(a) (5') Which of the following statements about **List ADT** is/are TRUE?

A. If we want to frequently delete the first element from the list, we prefer to implement the list using a linked list.

B. If we want to randomly access the elements in the list, we prefer to implement the list using an array.

C. Merging two sorted lists, where each list has n elements, to a sorted one takes $O(n)$ time, no matter whether the list is implemented using an array or a linked list.

D. Erasing the tail node of the linked list takes $O(1)$ time, whether the linked list is a singly linked list or a doubly linked list.

(b) (5') Given an initially empty **hash table** of capacity 10 and hash function $h(x) = x \bmod 10$. Assume we are inserting $\langle 101, 27, 91, 46, 13, 52, 42\rangle$ to the table in order. Which of the following statements is/are TRUE?

A. If we deal with collisions using chaining, there will be 5 non-empty chains.

B. If we deal with collisions using using linear probing, 2 collisions happen.

C. If we deal with collisions using quadratic probing, insertion might fail even if the table is not full.

D. It is possible for us to avoid collisions when inserting the given numbers by redesigning the hash function.

Solution:

A. There are 5 chains at bin $1, 2, 3, 6, 7$.

B. 3 collisions happen.

C. $\{k^2 | k \in \mathbb{N}\} \equiv \{0, 1, 4, 5, 6, 9\} (\mathrm{mod} 10)$, hence $\{2 + k^2 | k \in \mathbb{N}\} \equiv \{1, 2, 3, 6, 7, 8\} (\mathrm{mod} 10)$. That is, we can only step through bin $1, 2, 3, 6, 7, 8$ if we are inserting some number at index 2. However, these bins are all occupied before we insert 42.

D. It is possible to map these 7 numbers to 7 distinct bins e.g. $h(x) = (x \bmod 12) \bmod 10$.

(c) (5') In the partition procedure of **Quick Sort**, assume the pivot is chosen uniformly at random from the array $\langle A_1, \cdots, A_n\rangle$. You can assume that n is an odd number. Which of the following statements is/are TRUE?

A. The pivot is most likely to be chosen as the median of $\{A_1, \cdots, A_n\}$.

B. ~~The pivot is expected to be~~The expected value of the pivot is the mean of $\{A_1, \cdots, A_n\}$.

C. In this way, the algorithm runs in $\Theta(n \log n)$ time in the worst case.

D. After partitioning into two subarrays, ~~the length of the shorter one is expected to be~~the expected value of the length of the shorter one is about $n/4$ as $n \to \infty$.

Solution: Let Y be the index of the pivot selected and let $X = A_Y$ be the pivot. Since $Y \sim \text{Uniform}([1, n] \cap \mathbb{Z})$, the probability mass function of Y is

$$p_Y(y) = 1/n, \quad \text{for } y = 1, 2, \cdots, n,$$

and zero elsewhere. Therefore

$$\mathbf{E}(X) = \sum_{y=1}^{n} p_Y(y) A_y = \sum_{y=1}^{n} \frac{1}{n} A_y = \frac{1}{n} \sum_{y=1}^{n} A_y,$$

which is the **mean** of $\{A_1, \cdots, A_n\}$. Randomization only reduces the probability of achieving the worst case, but the worst-case time complexity is still $\Theta(n \log n)$.
Let Z be the random variable denoting the index of the pivot in the **sorted** array. The probability mass function of Z is

$$p_Z(z) = 1/n, \quad \text{for } z = 1, \cdots, n$$

and zero elsewhere. The length of the shorter subarray is

$$L = \min\{Z - 1, n - Z\}.$$

We have that

$$\begin{aligned}
\mathbf{E}(L) &= \mathbf{E}(\min\{Z, n + 1 - Z\}) - 1 \\
&= \sum_{z=1}^{n} \min\{z, n + 1 - z\} \frac{1}{n} - 1 \\
&= \frac{1}{n} \left(\frac{n+1}{2} + \sum_{z=1}^{(n-1)/2} z \right) - 1 \\
&= \frac{1}{n} \left(\frac{n+1}{2} \right)^2 - 1 = \frac{(n-1)^2}{4n}
\end{aligned}$$

Therefore

$$\lim_{n\to\infty} \frac{\mathbf{E}(L)}{n} = \frac{1}{4}.$$

(d) (5') Assume we are inserting $\langle 1, 2, 3, 4, 5, 6, 7 \rangle$ to an initially empty **AVL tree** in a random order. Which of the following insertion orders will not cause any imbalance i.e. the tree keeps AVL balanced after each insertion?

A. $4, 2, 6, 1, 3, 5, 7$

B. $3, 5, 1, 6, 4, 7, 2$

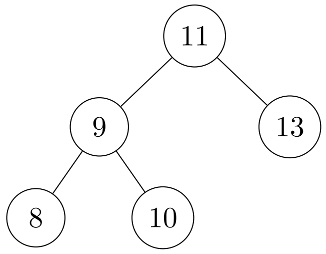
C. $4, 6, 2, 3, 5, 7, 1$

D. $5, 3, 6, 7, 4, 2, 1$

Solution: B will have imbalance after inserting 7.

4. (8 points) Play with AVL Tree

Given an AVL tree, as shown in the figure below:

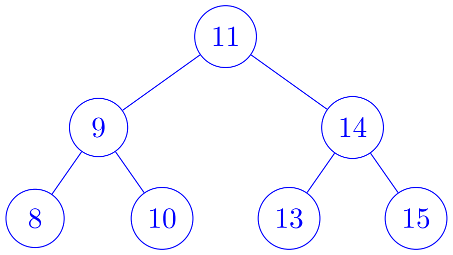


(a) (2') Assume you want to insert 15. Which nodes shall you step through to locate where to insert 15? Please write them down in order.

Solution: 11, 13

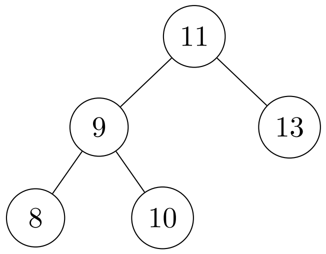
(b) (3') Based on (a), assume 15 is already inserted and we continue to insert 14. Dose this insertion cause any imbalance in the tree? If so, please draw the tree after rotation and re-balance. If not, explain why.

Solution: Yes.



(c) (3') **This sub-question is independent of (a) and (b).**

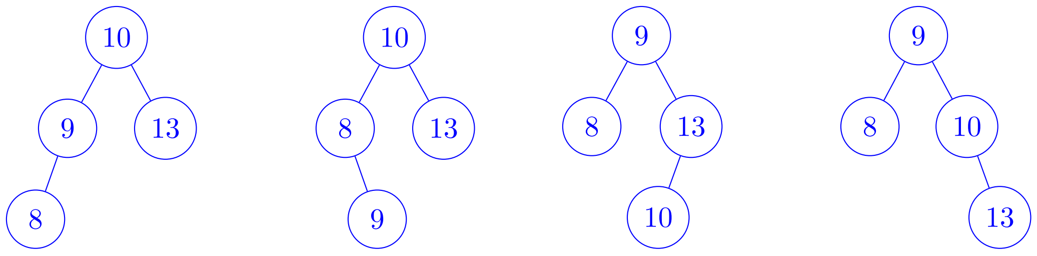
For your convenience, the figure is given again below. Is it possible that 11 is the last element inserted to the AVL tree shown in the figure? If so, please draw one possible AVL tree before inserting 11. If not, explain why.



Solution: Impossible.

Key Observation: Since there are only 4 nodes before insertion, and the height difference between two subtrees is at most 1, the height of the AVL must be 2 and the root node must be full. Hence we have 4 available places to insert one extra leaf node.

Once the structure is determined, the numbers to fill in the node is also determined. Therefore, we have 4 different possiblle candidate AVLs:



AVL A AVL B AVL C AVL D

Note that we have not inserted 11 yet, so to make 11 be the root, this insertion must cause an imbalance.

However, inserting 11 to AVL A and B would not cause any imbalance.

Similary, 11 also cannot reach the root node after inserting it to AVL C and D. Rotation would only happen among nodes 10, 11, 13 and the original root 9 is not affected.

5. (10 points) Insertion Sort on Small Arrays in Merge Sort

Although Merge Sort runs in $\Theta(n \log n)$ worst-case time and Insertion Sort runs in $\Theta\left(n^2\right)$ worst-case time, the constant factors in Insertion Sort often make it faster in practice for small size problems.

Please recall the modified version of Merge Sort, which has been covered in our lectures, where the subarrays of length $\leqslant k$ will be sorted directly using Insertion Sort, instead of continuing dividing it into two halves. Note that there will be approximately n/k subarrays to perform Insertion Sort on.

(a) (2') Show that Insertion Sort can sort the n/k subarrays of length k in $\Theta(nk)$ worst-case time.

Solution: Insertion-sort has $\Theta(k^2)$ time complexity on a subarray of length k, so the total time cost is $\Theta\left(n/k \cdot k^2\right) = \Theta(nk)$.

(b) (4') Show that it takes $\Theta(n \log(n/k))$ time to merge the n/k sorted subarrays into one.

Solution: Consider the recursion tree. Assume the recursion tree has m levels, that is, the array of length n is cut into subarrays of length k after m times of half-cutting. Hence $\frac{1}{2^m} \cdot n = k$, and we can obtain $m = \Theta(\log_2(n/k))$. Since it is $\Theta(n)$ work of merging subarrays in each level, the total work is $\Theta(n \log(n/k))$.

(c) (4') From the analysis above, we conclude that this algorithm has $\Theta\left(nk + n \log\left(n/k\right)\right)$ time compexltiy. In practice, we may observe that the algorithm takes $T(n) = ank + bn \ln(n/k)$ time, where $a, b > 0$ are some constants. What value of k (in terms of a and b) will you choose to minimize $T(n)$? Justify your answer. It is OK if your answer is not necessarily an integer.

Solution: Consider $f(x) = anx + bn \ln(n/x) \Rightarrow f'(x) = an - bn/x$. Setting $f'(x) = 0$ yields $x = b/a$. It could be verified that $f(x)$ achieves its minimal at $x = b/a$. Therefore we may pick $k = b/a$ so that $T(n)$ is minimized.

6. (12 points) Count β-Inversions

Given an array $A = \langle A_1, \cdots, A_n \rangle$, a pair of elements (A_i, A_j) form **a β-inversion** ($\beta > 0$) if

$$i < j \text{ and } A_i > \beta \cdot A_j$$

For example, a 1-inversion is the inversion introduced in our lectures when $\beta = 1$.

Note: For the following sub-questions, you should describe your algorithms in **natural language** or **pseudocode** clearly.

(a) (5') Recall that we have studied an enhanced Merge Sort to count the number of inversions in our lectures. When merging two sorted subarrays L and R, we also count the number of inversions (l, r) such that $l \in L$ and $r \in R$.

Now, please come up with a similar $\Theta(n)$ algorithm to **count the number of β-inversions** (A_i, A_j) **such that** $A_i \in L$ **and** $A_j \in R$, where $L = \langle A_1, \cdots, A_{mid} \rangle$ and $R = \langle A_{mid+1}, \cdots, A_n \rangle$ are two **sorted subarrays** of A.

Hint: You could come up with a $\Theta(n^2)$ algorithm first, and then optimize the inner loop.

Solution:

Algorithm 1 Possible Answer 1 (Outer-j)

```
function COUNT-β-INVERSIONS-L-R($L = \langle A_1, \cdots, A_{mid} \rangle, R = \langle A_{mid+1}, \cdots, A_n \rangle$ )
    $cnt \leftarrow 0$
    $i \leftarrow 1$
    for $j \in \{mid + 1, \cdots, n\}$ do
        while $i \le mid$ and $A_i \le \beta \cdot A_j$ do
            $i \leftarrow i + 1$
        end while
        $cnt \leftarrow cnt + (mid - i + 1)$        ▷ $(A_i, A_j), (A_{i+1}, A_j), \cdots, (A_{mid}, A_j)$
    end for
    return $cnt$
end function
```

Algorithm 2 Possible Algorithm 2 (Outer-i)

```
function COUNT-β-INVERSIONS-L-R($L = \langle A_1, \cdots, A_{mid} \rangle, R = \langle A_{mid+1}, \cdots, A_n \rangle$)
    $cnt \leftarrow 0$
    $j \leftarrow mid + 1$
    for $i \in \{1, \cdots, mid\}$ do
        while $j \le n$ and $A_i > \beta \cdot A_j$ do
            $j \leftarrow j + 1$
        end while
        $cnt \leftarrow cnt + (j - mid - 1)$        ▷ $(A_i, A_{mid+1}), \cdots, (A_i, A_{j-2}), (A_i, A_{j-1})$
    end for
    return $cnt$
end function
```

(b) (7') Based on (a), please come up with a **divide-and-conquer** algorithm to count the number of β-inversions in A. Analyse the time complexity of your algorithm.

Note:

- You can directly use the subroutine of merging two sorted sub-arrays into one (MERGE(L, R)) and the subroutine you implemented in (a) (COUNT-β-INVERSIONS-L-R(L, R)).
- Please describe how you **divide** a problem into sub-problems and **combine** the solutions to the subproblems clearly and remember to show the **base case**.
- Your algorithm should be better than $\Theta(n^2)$.

Solution:

Algorithm Design:

1. If A is reduced into exactly 1 element, then there is no β-inversion.
2. Else we cut the array in the middle, and recur for each half. Then we obtain two sorted halves, and the number of β-inversions in each half, say cnt_L and cnt_R.
3. We count the the number of β-inversions (l, r) such that l is in the first sorted half and r is in another, say cnt_{LR}, using the algorithm designed in (a).
4. We merge two sorted halves into one, and hence A is sorted.
5. The number of β-inversions in A is $cnt_L + cnt_R + cnt_{LR}$.

Pseudocode:

Algorithm 3 Count β-Inversions in an Array

```
function COUNT-β-INVERSIONS($A = \langle A_1, \cdots, A_n \rangle$)
    if $n = 1$ then
        return $A$, $0$
    end if
    Cut $A$ in the middle and divide it into $L_0$ and $R_0$
    $L$, $cnt_L \leftarrow$ COUNT-β-INVERSIONS($L_0$)
    $R$, $cnt_R \leftarrow$ COUNT-β-INVERSIONS($R_0$)
    $cnt_{LR} \leftarrow$ COUNT-β-INVERSIONS-L-R($L, R$)
    $A_{sorted} \leftarrow$ MERGE-SORTED($L, R$)
    return $A_{sorted}$, $cnt_L + cnt_R + cnt_{LR}$
end function
```

Time Complexity Analysis:

We divide the array into two subarrays of half size and conquer each recursively. In the merge step, counting the number of β-inversions as designed in (a) and merging two sorted subarrays both take $O(n)$. That is

$$T(n) = 2T(n/2) + O(n)$$

By the Master Theorem, the complexity of this problem is $T(n) = O(n \log n)$.

Alternative Solution:

Algorithm 4 Count β-Inversions in an Array

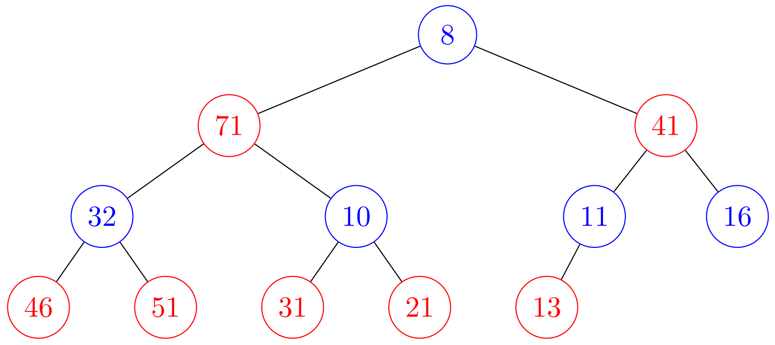
```
function COUNT-β-INVERSIONS($A = \langle A_1, \cdots, A_n \rangle$)
    if $n = 1$ then
        return $0$
    end if
    Cut $A$ in the middle and divide it into $L_0$ and $R_0$
    $cnt_L \leftarrow$ COUNT-β-INVERSIONS($L_0$)
    $cnt_R \leftarrow$ COUNT-β-INVERSIONS($R_0$)
    $L \leftarrow$ MERGE-SORT($L_0$)
    $R \leftarrow$ MERGE-SORT($R_0$)
    $cnt_{LR} \leftarrow$ COUNT-β-INVERSIONS-L-R($L, R$)
    return $cnt_L + cnt_R + cnt_{LR}$
end function
```

7. (15 points) Min-Max-Heap

A *min-max-heap* is a data structure which supports both `popMin()` and `popMax()` operations in $O(\log n)$ time. A min-max-heap is still a complete binary tree, but it has the following properties:

- For every element at even depth (on min-level), it is smaller than its parent (if existing) and greater than its grandparent (the parent node of its parent node, if existing). In short: **`x<parent(x)` and `x>grandparent(x)` for `x` on min-level.**
- For every element at odd depth (on max-level), it is greater than its parent and smaller than its grandparent(if existing). In short: **`x>parent(x)` and `x<grandparent(x)` for `x` on max-level.**

You can assume that all nodes are distinct and there are always more than one nodes in this question. Please see the following example min-max-heap, as shown in the figure below:



(a) (4') **Let's See What's New about Min-Max-Heap**

For each of the following statements about min-max-heap, please judge whether it is true or false. Tick (√) the correct answer. Assume the parent and the children of the node `x` all exist.

i. If `x` is on min-level, then its parent and children are also on min-level. True [✓ False]

ii. If `x` is on max-level, then `x` is greater than both of its children. [✓ True] False

iii. If `x` is on min-level and `x` is a strict ancestor of `y`, then `x<y`. [✓ True] False

iv. The minimum is on min-level and the maximum is on max-level. [✓ True] False

Solution:

Key Observation 1: For `x` on min-level, `x<parent(x)` and `x<child(x)`; for `x` on max-level, `x>parent(x)` and `x>child(x)`.

Key Observation 2: The minimum element `x` must be on min-level. If it is on max-level, `parent(x)<x` gives contradiction.

Key Observation 3: There is a *virtual* 4-ary min-heap in non-adjacent levels rooted at index 0 and there are 2 *virtual* 4-ary max-heaps rooted at index 1 and 2 respectively.

Key Observation 4: For `x` on min-level, `x` is the minimum of the subtree rooted at `x`; for `x` on max-level, `x` is the maximum of the subtree rooted at `x`.

(b) (3') **Where is Min and Max?**

How do we search the minimum and the maximum element in a min-max-heap with n elements? What is their time complexity?

Solution: The minimum in a min-max heap is the root node and the maximum is the greater one of the children of the root node. Both can be done in $O(1)$ time.

(c) **Percolate it Up!**
Similar to a common heap we learned in the lectures, the new element is inserted at the first available leaf and we will maintain the heap property along the path from this leaf node to the root node, which is called a **percolation-up**. For your reference, the pseudocode of **percolation-up in a common heap** is given below:

Algorithm 5 Percolation-Up in a Common Heap

```
function PERCOLATE-UP(x)                              ▷ Assume x is inserted at some leaf
    while parent(x) exists  and x<parent(x) do        ▷ Is it a min-heap or a max-heap?
        Swap x and parent(x)
    end while
end function
```

However, there are more properties to be considered in a min-max-heap. The pseudocode (with some incomplete lines) of **percolation-up in a min-max-heap** is given below:

Algorithm 6 Percolation-Up in a Min-Max-Heap

```
function PERCOLATE-UP(x)              ▷ Assume x is inserted at some leaf and parent(x) exists
    if x is on max-level then
        if ____(1)____ then            ▷ If the property between two adjacent levels is not broken
            while grandparent(x) exists  and ____(2)____ do      ▷ Similar to a max-heap
                Swap x and grandparent(x)
            end while
        else                                                     ▷ Else the property is broken
            Swap x and parent(x)                                 ▷ x is now on min-level
            while grandparent(x) exists  and ____(3)____ do      ▷ Similar to a min-heap
                Swap x and grandparent(x)
            end while
        end if
    else                          ▷ Else x is on min-level. You don't need to finish this case
        ...
    end if
end function
```

i. (2') Choose one of the following statements to fill in the blank (1):

◯ x<parent(x) ◯ x<grandparent(x) [✓ x>parent(x)] ◯ x>grandparent(x)

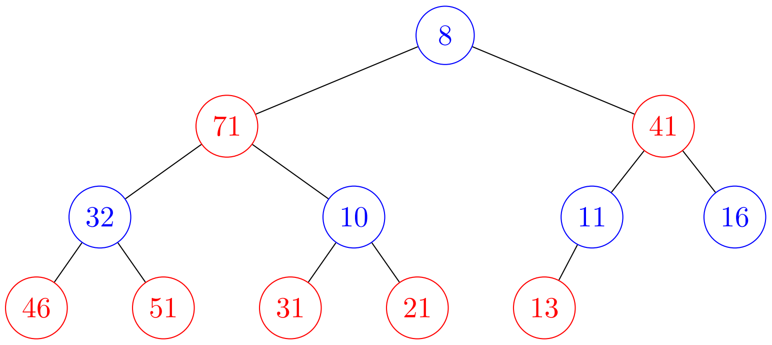
ii. (1') Fill in the blank (2): x>grandparent(x)

iii. (1') Fill in the blank (3): x<grandparent(x)

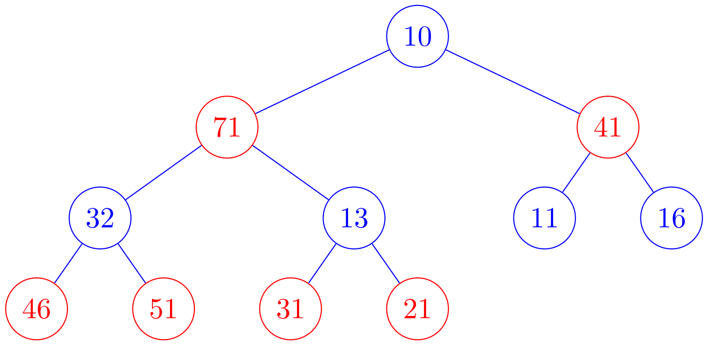
(d) (4') **Percolate it Down?**

Similarly, when popping a node off the min-max-heap, we move the last leaf node to the *hole* and percolate it down. However, the **percolation-down** procedure in a min-max-heap is a bit more complicated.

In this question, just consider you are popping the minimum off the example min-max-heap shown above (it is shown again below for your convenience). Please draw the min-max-heap after this `popMin()`.



Solution:


10
71
41
32
13
11
16
46
51
31
21