Data Structures
CSCI C343, Spring 2025

Midterm [A]

Name: ____________________

This exam has 11 questions, for a total of 100 points.

1. 6 points What is the output of running the `main` method of class `C`?

```
class Person {
    String name;
    public Person(String n) { name = n; }
    void change_name(String n) { name = n; }
}
public class C {
    static void f(Person a, int b) {
        a.change_name("Baker");
        b += 10;
    }
    public static void main(String[] args) {
        Person x = new Person("Smith");
        int y = 5;
        f(x, y);
        System.out.println(x.name);
        System.out.println(y);
    }
}
```

Solution:

```
Baker
5
```

Name: ______________________

2. 12 points What is the big-O time complexity of the following `mergesort` function? Explain your answer. The functions `length`, `take`, and `drop` are all $O(n)$ in the length of their input list. The function `take(L,k)` returns a list of length k. The function `drop(L,k)` returns a list of length (`length`(L) − `k`).

```
class Node {
    int data;  Node next;
    Node(int d, Node n) { data = d; next = n; }
}
static Node merge(Node A, Node B) {
    if (A == null) {
        return B;
    } else if (B == null) {
        return A;
    } else if (A.data <= B.data) {
        return new Node(A.data, merge(A.next, B));
    } else {
        return new Node(B.data, merge(A, B.next));
    }
}
static Node mergesort(Node N) {
    if (N == null || N.next == null) {
        return N;
    } else {
        int n = length(N);
        Node a = mergesort(take(N, n / 2));
        Node b = mergesort(drop(N, n / 2));
        return merge(a, b);
    }
}
```

Solution: The time complexity of `merge` is $O(n)$ because the non-recursive parts of `merge` are $O(1)$, there is at most one recursive call, and the input size decreases by one in the recursive call. **(6 points)**

The time complexity of `mergesort` is $O(n \log n)$ because it is $O(n)$ time per level of the recursion tree and there are $\log n$ levels (the size of the input is cut in half with each recursive call). **(6 points)**

3. 12 points What is the big-O time complexity of the following `flood` method in terms of the total number of tiles, represented by n? Provide an argument for your answer that analyzes every statement in the method and how their individual time complexities combine into the total time complexity.

```
public static void flood(WaterColor color,
                         LinkedList<Coord> flooded_list,
                         Tile[][] tiles,
                         Integer board_size) {
    boolean[][] flooded = new boolean[board_size+1][board_size+1];
    ArrayList<Coord> flooded_array = new ArrayList<>(flooded_list);
    for (Coord c : flooded_list)
        flooded[c.getY()][c.getX()] = true;
    for (int i = 0; i != flooded_array.size(); ++i) {
        Coord c = flooded_array.get(i);
        for (Coord n : c.neighbors(board_size)) {
            if (!flooded[n.getY()][n.getX()]
                    && tiles[n.getY()][n.getX()].getColor() == color) {
                flooded_array.add(n);
                flooded_list.add(n);
                flooded[n.getY()][n.getX()] = true;
            }
        }
    }
}
```

Solution:

1. The allocation of `flooded` and `flooded_array` is $O(n)$ **(1 point)**.
2. The first `for` loop is $O(n)$ **(1 point)**.
3. The second `for` loop iterates $O(n)$ times. **(1 point)**
   (a) The `get(i)` method call is $O(1)$ **(3 points)**.
   (b) The inner `for` loop iterates at most 4 times and the operations inside it are all $O(1)$, so this loop is $O(1)$ **(2 points)**.
   (c) Thus, the body of the second `for` loop is $O(1)$ **(1 point)**.

   So the second `for` loop's time complexity is $O(n)$ **(2 points)**.

Thus, the time complexity of `flood` is $O(n)$ **(1 point)**.

4. $\boxed{\text{8 points}}$ Show that $3n + 10 \in O(n)$ using the definition of big-O.

Solution: Choose $c = 4$ and $k = 10$ because $3n + 10 \leq 4n$ for $n \geq 10$, as shown in the table below. (There are other valid choices for c and k.)

n	3n + 10	4n
1	13	4
2	16	8
10	40	40
11	43	44
12	46	48

Name: ____________________

5. 12 points Write the code for the following function that returns the position of the first `true` in the half-open range `[begin,end)` within array `A`. If there are no `true` values in the half-open range, return `end`. The elements of the array are already sorted. The function must have time complexity $O(\log_2 n)$ where n is the length of the half-open interval.

```
static int find_first_true_sorted(boolean[] A, int begin, int end) {
```

Solution:

```
static int find_first_true_sorted(boolean[] A, int begin, int end) {
    if (begin == end) { // 2 points
        return end;     // 2 points
    } else {
        int mid = begin + ((end - begin) / 2);             // 2 points
        if (A[mid]) {                                      // 2 points
            return find_first_true_sorted(A, begin, mid);   // 2 points
        } else {
            return find_first_true_sorted(A, mid + 1, end); // 2 points
        }
    }
}
```

6. 10 points Fill in the blanks to complete the following proofs.

```
theorem T1: all P:bool, Q:bool. if P and Q then P
proof
  arbitrary P:bool, Q:bool
  ___(a)___
  ___(b)___
end

theorem T2: all T:type, x:T, y:T.
  @[]<T> ++ node(y, []) = node(y, [])
proof
  arbitrary T:type, x:T, y:T
  ___(c)___
end

theorem T3: all P:bool, Q:bool. if (if P then Q) and P then Q
proof
  arbitrary P:bool, Q:bool
  assume prem: (if P then Q) and P
  have p: P by prem
  have pq: if P then Q by prem
  conclude Q by ___(d)___
end

theorem T4: all P:bool, Q:bool. if (P and Q) then (P or Q)
proof
  arbitrary P:bool, Q:bool
  assume prem: P and Q
  ___(e)___
end
```

Solution: 2 points each

```
(a) assume prem    (OK to choose a different label.)
(b) prem           (OK to use recall instead of the label.)
(c) definition operator++   (OK to use evaluate)
(d) apply pq to p
(e) prem           (or conjunct 0 of prem, or conjunct 1 of prem)
```

7. [10 points] Fill in the blanks to complete the following proof that the length of the list returned by `take(n, xs)` is `n`, if `n` is less or equal to the length of the list `xs`.

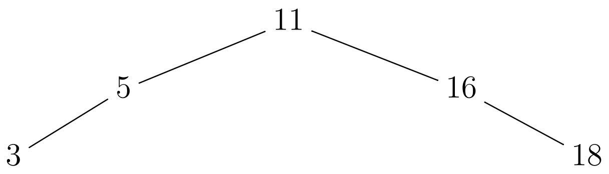
```
function length<E>(List<E>) -> Nat {
  length(empty) = 0
  length(node(n, next)) = 1 + length(next)
}
function take<T>(Nat, List<T>) -> List<T> {
  take(0, xs) = empty
  take(suc(n), xs) =
    switch xs {
      case empty { empty }
      case node(x, xs’) { node(x, take(n, xs’)) }
    }
}

theorem length_take: all T:type. all n:Nat, xs:List<T>.
  if n <= length(xs) then length(take(n, xs)) = n
proof
  arbitrary T:type
  ___(a)___
  case 0 {
    arbitrary xs:List<T>
    ___(b)___
    conclude length(take(0, xs)) = 0 by evaluate
  }
  case suc(n’) assume
    IH: all xs:List<T>. if n’ <= length(xs) then length(take(n’, xs)) = n’
  {
    ___(c)___
    switch xs {
      case [] {
        assume prem: suc(n’) <= length(@[]<T>)
        conclude ___(d)___ by evaluate in prem
      }
      case node(x, xs’) {
        assume prem: suc(n’) <= length(node(x, xs’))
        have n_xs: n’ <= length(xs’) by evaluate in prem
        have len_take: length(take(n’, xs’)) = n’ by ___(e)___
        equations
              length(take(suc(n’), node(x, xs’)))
            = 1 + length(take(n’, xs’))          by evaluate
        ... = 1 + n’                             by rewrite len_take
        ... = suc(n’)                            by evaluate
      }
    }
  }
end
```

Solution: 2 points each

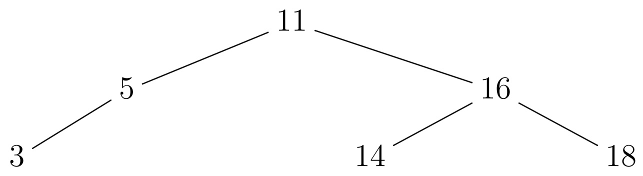
(a) `induction Nat`
(b) `assume prem`

```
    (c) arbitrary xs:List<T>
    (d) false
    (e) apply IH to n_xs
```

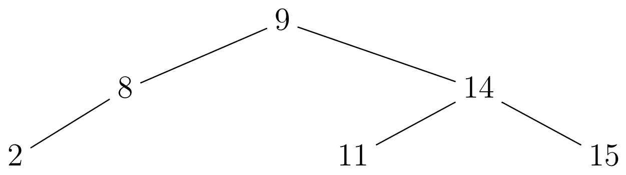
8. 4 points Given the following binary search tree, insert key 14 into the tree, maintaining the binary search tree property, and draw the resulting tree.



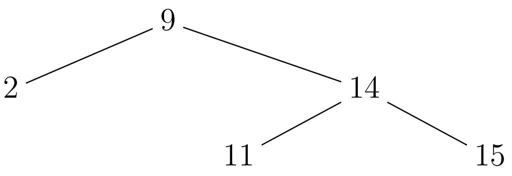
Solution:



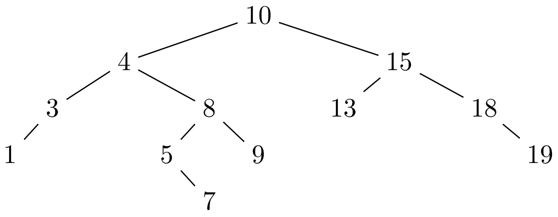
9. 4 points Given the following binary search tree, remove key 8 from the tree, maintaining the binary search tree property, and draw the resulting tree.



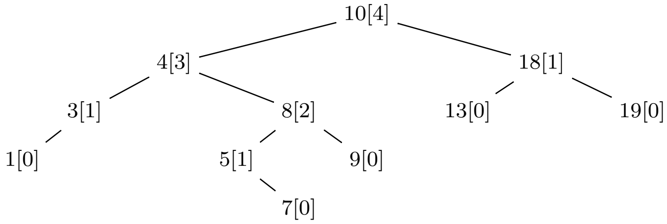
Solution:



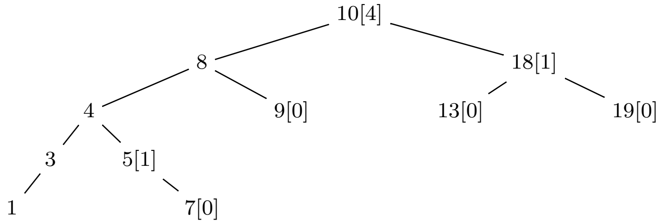
10. 12 points Given the following AVL binary search tree, remove key 15, maintaining the binary search tree and AVL properties using the right and left rotation operations. Identify which nodes do not satisfy the AVL property and explain each rotation that you make to the tree. Draw the tree after each rotation.



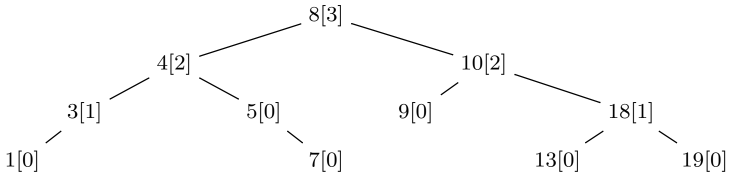
Solution: One alternative is to replace 15 with 18.



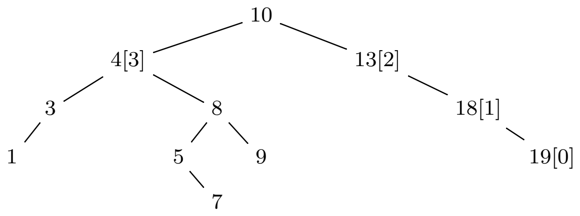
Node 10 is not AVL. It's tallest grandchild is 8, a zig-zag. Rotate left on 4.



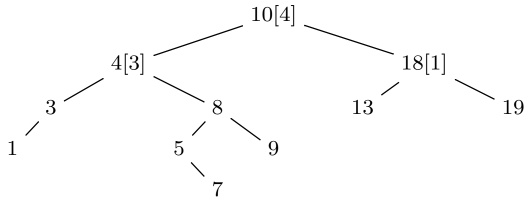
Rotate right on 10.



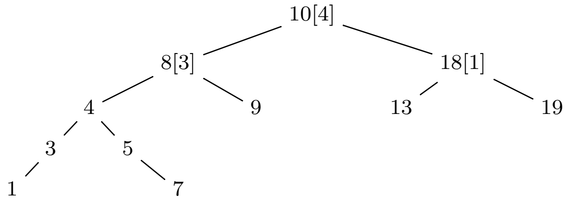
Another alternative is to replace 15 with 13.



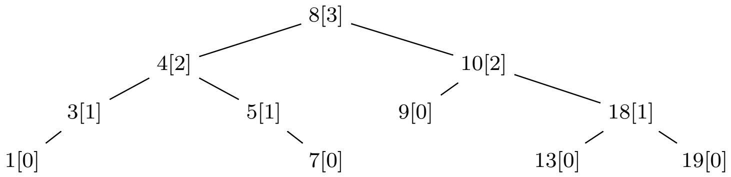
Node 13 is not AVL. Rotate left on 13.



Node 10 is not AVL. There's a zig-zag to its tallest grandchild, node 8. Rotate left on 4.



Rotate right on 10



11. 10 points Fill in the blanks to complete the following implementation of the Merge Sort algorithm.

```
public interface Iterator<T> {
    T get();
    void set(T e);
    void advance();
    void advance(int n);
    boolean equals(Iterator<T> other);
    Iterator<T> clone();
}
static <E extends Comparable<? super E>>
Iterator<E> merge(Iterator<E> begin1, Iterator<E> end1,
                  Iterator<E> begin2, Iterator<E> end2,
                  Iterator<E> result) {
    Iterator<E> i = begin1.clone(), j = begin2.clone(), out = result.clone();
    while (!i.equals(end1) && !j.equals(end2)) {
        if (i.get().compareTo(j.get()) <= 0) {
            ___(a)___;
            out.advance();
            i.advance();
        } else {
            ___(b)___;
            out.advance();
            j.advance();
        }
    }
    Iterator<E> out2 = copy(i, end1, out);
    return copy(j, end2, out2);
}
static <E extends Comparable<? super E>>
void sort(Iterator<E> begin, Iterator<E> end) {
    int n = distance(begin, end);
    if (n < 2) {
        return;
    } else {
        Iterator<E> mid = begin.clone();
        ___(c)___;
        sort(begin, mid);
        sort(mid, end);
        ArrayList<E> tmp = make_array(n);
        ArrayListIterator<E> tmp_begin = new ArrayListIterator<>(tmp, 0),
                tmp_end = new ArrayListIterator<>(tmp, tmp.size());
        merge(begin, mid, ___(d)___, end, ___(e)___);
        copy(tmp_begin, tmp_end, begin);
    }
}
```

Solution: 2 points each

(a) `out.set(i.get())`
(b) `out.set(j.get())`
(c) `mid.advance(n / 2)`

```
(d) mid
(e) tmp_begin
```