I111E Algorithms and Data Structures

2019, Term 2-1
Ryuhei Uehara and Giovanni Viglietta (Room I67, {uehara,johnny}@jaist.ac.jp)

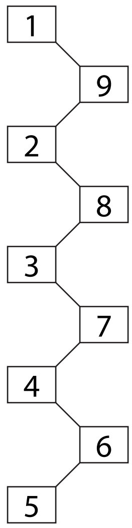
Problem 1A (10pts):

(a) Starting from an empty Binary Search Tree, we perform the following insertions in order: `insert(1)`, `insert(9)`, `insert(2)`, `insert(8)`, `insert(3)`, `insert(7)`, `insert(4)`, `insert(6)`, `insert(5)`. Draw the resulting Binary Search Tree.

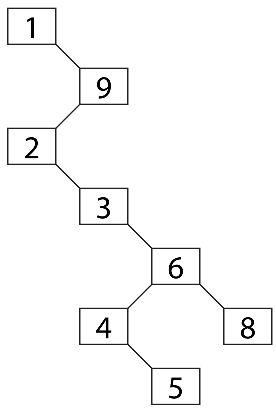
(b) On the same Binary Search Tree of part (a), we perform the following operations in order: `delete(8)`, `insert(8)`, `delete(7)`. Draw the resulting Binary Search Tree.

Solution:

(a)



(b)



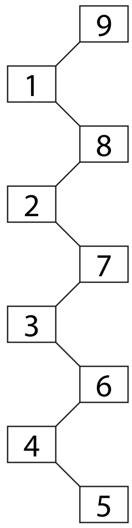
Problem 1B (10pts):

(a) Starting from an empty Binary Search Tree, we perform the following insertions in order: `insert(9)`, `insert(1)`, `insert(8)`, `insert(2)`, `insert(7)`, `insert(3)`, `insert(6)`, `insert(4)`, `insert(5)`. Draw the resulting Binary Search Tree.

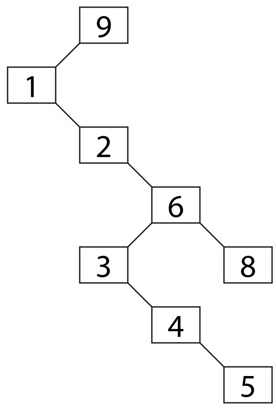
(b) On the same Binary Search Tree of part (a), we perform the following operations in order: `delete(8)`, `insert(8)`, `delete(7)`. Draw the resulting Binary Search Tree.

Solution:

(a)



(b)



Problem 2A (10pts):

(a) Starting from an empty heap, we insert the following numbers in order: 20, 3, 28, 22, 12, 8, 31, 15. Draw the resulting heap in its array representation.
(b) Starting from the heap of part (a), we remove the minimum element from the root twice. Draw the resulting heap in its array representation.

Solution:

(a)

3	12	8	15	20	28	31	22

(b)

12	15	22	31	20	28

Problem 2B (10pts):

(a) Starting from an empty heap, we insert the following numbers in order:
19, 2, 27, 23, 11, 9, 32, 16. Draw the resulting heap in its array representation.
(b) Starting from the heap of part (a), we remove the minimum element from the root twice. Draw the resulting heap in its array representation.

Solution:

(a)

2	11	9	16	19	27	32	23

(b)

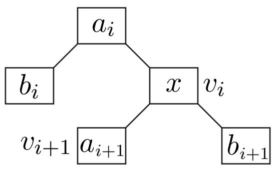
11	16	23	32	19	27

Problem 3 (10pts): We learned how to deal with a heap after removing the minimum item from its root. Prove that the process does not break the consistency of the heap.

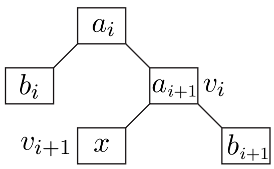
Solution: The removal algorithm takes the last element x of the heap (which is found in a leaf) and places it at the root. Then, x "slides down" the heap until it is larger than both its children nodes. As x slides down, it is exchanged with some values, say a_1, a_2, ..., a_k, originally located in nodes v_1, v_2, ..., v_k, respectively. We only need to prove that the heap's fundamental property (i.e., that the value in any node is smaller that the values in each of its children) remains true for these k nodes, because all other nodes remain unchanged.

Let us prove by induction that the heap's property is preserved on all the v_i's. Assume that, at "time i", i.e., when x reaches node v_i, the property is true from node v_1 to node v_{i-1}. If x is smaller than the values stored in both v_i's children nodes, then the algorithm terminates without modifying the heap. In this case, the heap's property is still true for nodes v_1, ..., v_{i-1} (because nothing has changed), and it is also true for v_i, by our assumption.

Otherwise, x is greater than at least one value stored in the children of v_i. To complete the proof by induction, we need to show that, at "time $i+1$", i.e., when x reaches node v_{i+1}, the property is true from node v_1 to node v_i. Let a_{i+1} and b_{i+1} be the values stored in the children of v_i, with $a_{i+1} < b_{i+1}$. By assumption, we have $a_{i+1} < x$, and so the heap's property is satisfied for v_i at time $i+1$. We still have to prove that the property remains satisfied from node v_1 to node v_{i-1} at time $i+1$. This is also true from v_1 to v_{i-2} by inductive hypothesis (because the values in these nodes and their children did not change from time i to time $i+1$), but we still need to check it for node v_{i-1}. This node's value remained unchanged (it is still a_i), but one of its children changed value from x to a_{i+1}. To conclude the proof, we need to show that $a_i < a_{i+1}$: this is true because, in the original heap (i.e., at time 0), a_i was in v_i, and a_{i+1} was in its child v_{i+1}, implying that $a_i < a_{i+1}$ by the consistency of the original heap.



time i



time $i+1$

Problem 4 (20pts): Given the array $a[0] \sim a[n-1]$ consisting of n real numbers, we want to compute the function $f(x) = a[0]+a[1]x+a[2]x^2+\cdots+a[n-1]x^{n-1}$. Consider the two following algorithms:

1. Following the definition, compute
$a[0] + a[1] \times x + a[2] \times x \times x + a[3] \times x \times x \times x + \cdots + a[n-1] \times x \times \cdots \times x$ step by step.
2. Compute the function after the following modification:
$a[0] + x \times (a[1] + x \times (a[2] + x \times (a[3] + x \times (a[4] + \cdots + x \times (a[n-2] + x \times a[n-1])))))$

Evaluate the number of summation and multiplication operations respectively as functions of n, and discuss which of the two algorithms is better.

Solution: For (1), the summation operations are $n-1$, and the multiplication operations are $\sum_{i=0}^{n-1} i = n(n-1)/2 = O(n^2)$.

For (2), the summation operations are $n-1$, and the multiplication operations are $n-1 = O(n)$. Algorithm (2) is better, because it performs the same number of summations and asymptotically fewer multiplications than (1).

Problem 5 (20pts): Two arrays of size n are given: A is sorted in increasing order, and B is sorted in decreasing order. Give an efficient algorithm that determines if there is an index i such that $A[i] = B[i]$, and determine its running time.

Solution: If we compare $A[i]$ and $B[i]$, we can determine if the required index lies to the left or to the right of i: if $A[i] < B[i]$, then the left portion of the two arrays can be excluded, and vice versa. This suggests that the binary search algorithm can be adapted to this problem. Here is a possible C implementation:

```
int problem_5(int A[], int B[], int n) {
    int left = 0;
    int right = n - 1;
    do {
        int mid = (left + right) / 2;
        if (A[mid] == B[mid]) return mid;
        if (A[mid] > B[mid]) right = mid - 1;
        else left = mid + 1;
    } while (left <= right);
    return -1;
}
```

The running time is the same as binary search: $O(\log n)$.

Problem 6A (20pts): Let $(a_n)_{n\geq 1}$ be the following sequence of real numbers:

$$a_n = \begin{cases} 1 & \text{if } n = 1 \\ 0 & \text{if } n = 2 \\ \left(\frac{n}{n-2} + a_{n-1}\right) \cdot a_{n-2} & \text{if } n > 2 \end{cases}$$

Describe an optimal algorithm that takes n as input and outputs a_n. What are the running time and space complexity of your algorithm?

Solution: We can prove by induction that

$$a_n = \begin{cases} 0 & \text{if } n \text{ is even} \\ n & \text{if } n \text{ is odd} \end{cases}$$

This is true for $n = 1$ and $n = 2$. Suppose now that $n > 2$, and let the inductive hypothesis hold for all a_i's up to a_{n-1}. If n is even, then $n-1$ is odd and $n-2$ is even, and so $a_{n-1} = n-1$ and $a_{n-2} = 0$. Hence, $a_n = \left(\frac{n}{n-2} + n - 1\right) \cdot 0 = 0$, as desired. On the other hand, if n is odd, then $a_{n-1} = 0$ and $a_{n-2} = n-2$, and therefore $a_n = \left(\frac{n}{n-2} + 0\right) \cdot (n-2) = n$. This concludes the proof.

So, to compute a_n, we only have to test n mod 2. Here is a C implementation of the algorithm:

```
int problem_6A(int n) {
   if (n % 2 == 0) return 0;
   else return n;
}
```

The running time and space complexity are constant, which is obviously optimal.

Problem 6B (20pts): Let $(a_n)_{n\geq 1}$ be the following sequence of real numbers:

$$a_n = \begin{cases} 0 & \text{if } n = 1 \\ 2 & \text{if } n = 2 \\ \left(\frac{n}{n-2} + a_{n-1}\right) \cdot a_{n-2} & \text{if } n > 2 \end{cases}$$

Describe an optimal algorithm that takes n as input and outputs a_n. What are the running time and space complexity of your algorithm?

Solution: We can prove by induction that

$$a_n = \begin{cases} n & \text{if } n \text{ is even} \\ 0 & \text{if } n \text{ is odd} \end{cases}$$

This is true for $n = 1$ and $n = 2$. Suppose now that $n > 2$, and let the inductive hypothesis hold for all a_i's up to a_{n-1}. If n is even, then $n - 1$ is odd and $n - 2$ is even, and so $a_{n-1} = 0$ and $a_{n-2} = n - 2$. Hence, $a_n = \left(\frac{n}{n-2} + 0\right) \cdot (n - 2) = n$, as desired. On the other hand, if n is odd, then $a_{n-1} = n - 1$ and $a_{n-2} = 0$, and therefore $a_n = \left(\frac{n}{n-2} + n - 1\right) \cdot 0 = 0$. This concludes the proof.

So, to compute a_n, we only have to test n mod 2. Here is a C implementation of the algorithm:

```c
int problem_6B(int n) {
   if (n % 2 == 0) return n;
   else return 0;
}
```

The running time and space complexity are constant, which is obviously optimal.