Exercise IV, Algorithms 2024-2025

These exercises are for your own benefit. Feel free to collaborate and share your answers with other students. There are many problems on this set, solve as many as you can and ask for help if you get stuck for too long. Problems marked * are more difficult but also more fun :).

Heaps and Heapsort

1 Illustrate the steps of HEAPSORT by filling in the arrays and the tree representations on the next page.

Solution: See figure on page 4.

2 (Exercise 6.1-4 in the book) Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

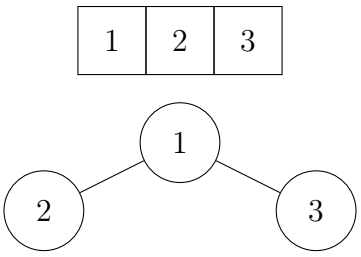
Solution: The smallest element can only reside as a leaf.

3 (half a *, Problem 6-1 in the book) We can build a heap by repeatedly calling MAX-HEAP-INSERT to insert the elements into the heap. Consider the following variation on the BUILD-MAX-HEAP procedure:

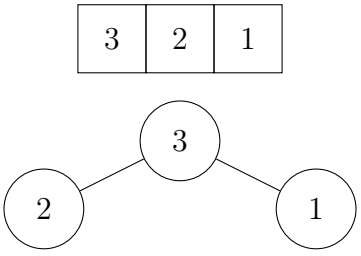
BUILD-MAX-HEAP'(A)
1. A.heap-size $= 1$
2. **for** $i = 2$ **to** A.length
3. MAX-HEAP-INSERT$(A, A[i])$

3a Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP' always create the same heap when run on the same input array? Prove that they do, or provide a counter example.

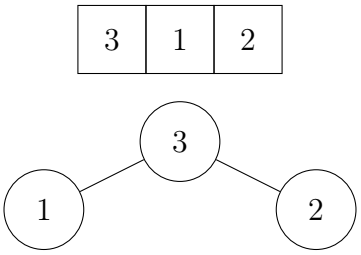
Solution: They don't. Consider the following counterexample. For input array A:



BUILD-MAX-HEAP will produce:



BUILD-MAX-HEAP' will produce:



3b Show that in the worst case, BUILD-MAX-HEAP' requires $\Theta(n \lg n)$ time to build an n-element heap (in comparison to BUILD-MAX-HEAP that we saw in class only needs time $\Theta(n)$).

Solution: Upper bound: There are $n-1$ calls to MAX-HEAP-INSERT, each taking $O(\lg n)$ time. Therefore the upper bound is $O(n \lg n)$.

Lower bound: Consider the case in which the input array is given in strictly increasing order. Each call to MAX-HEAP-INSERT then causes HEAP-INCREASE-KEY to go all the way up to the root, taking $\Theta(\lfloor \lg i \rfloor)$ for node i. The total time is:

$$\begin{aligned}
\sum_{i=2}^{n} \Theta(\lfloor \lg i \rfloor) &\geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg \lceil n/2 \rceil \rfloor) \\
&\geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg (n/2) \rfloor) \\
&= \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \lg n - 1 \rfloor) \\
&\geq \frac{n}{2} \Theta(\ln n) \\
&= \Omega(n \lg n)
\end{aligned}$$

In the worst case, therefore, BUILD-MAX-HEAP' requires $\Theta(n \lg n)$ time to build an n-element heap.

4 (*, Exercise 6.5-9 in the book) Give an $O(n \lg k)$-time algorithm to merge k sorted arrays into one sorted array, where n is the total number of elements in all the input arrays.

Hint: Use a min-heap of size k for k-way merging.

Solution: The idea is to use a min-heap of k elements to find the smallest remaining element in all k arrays in $O(\lg k)$ time in each iteration.

The input to the algorithm is represented with an array A of n elements, and an associated array L of $k+1$ elements which holds indices for the k sorted arrays. Sorted array i is located in the array A in elements $A[L[i]], \ldots, A[L[i+1]-1]$.

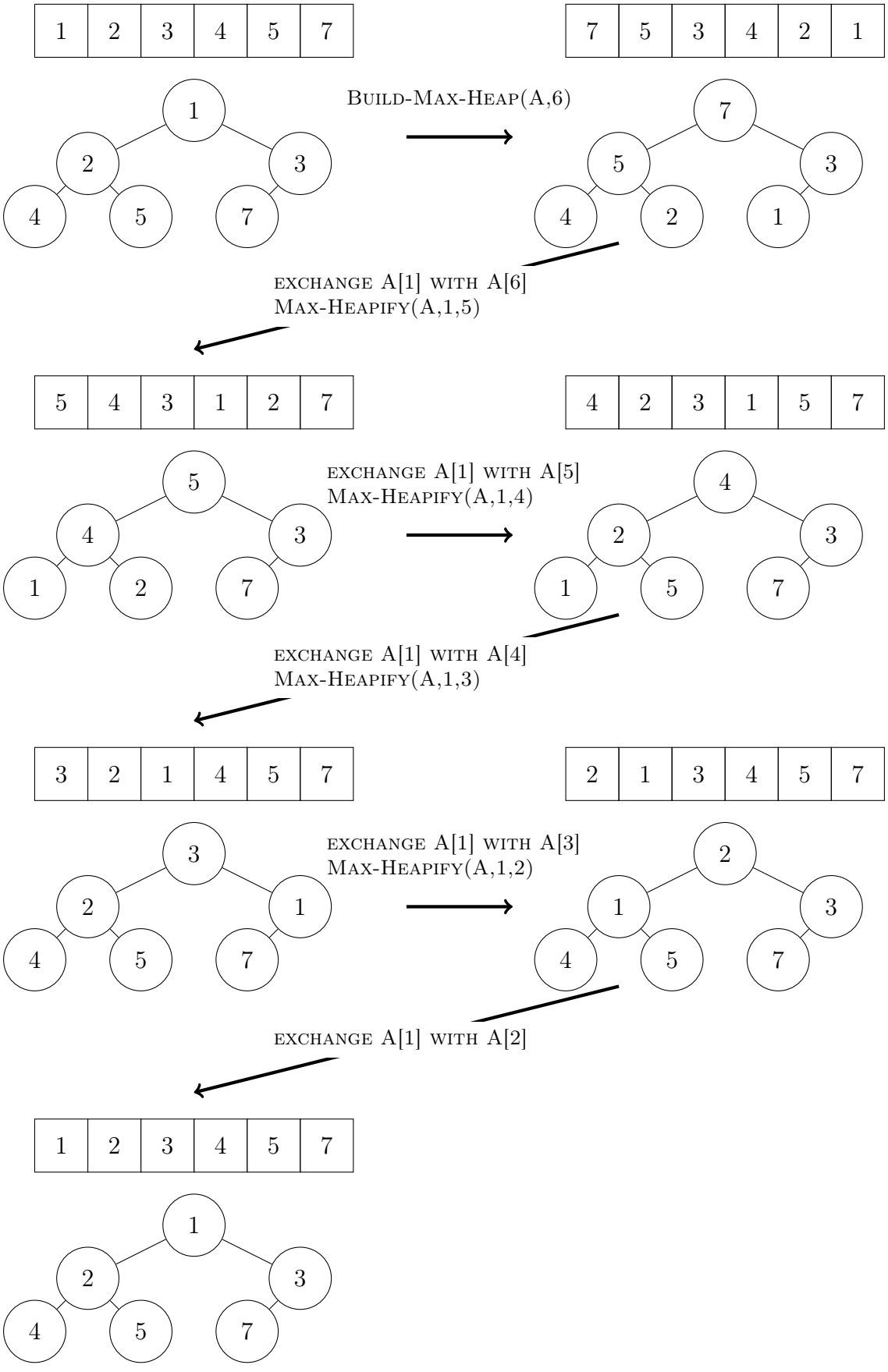
We also use one array B of size k for the min-heap and one array C of size n for the output. The min-heap needs to keep track of two additional values besides the element key. These are the array number and the index within array A of the element. The algorithm starts by initializing array B with the first (smallest) element of each array, and building a min-heap over it. Then in each iteration of the loop:

- The first (smallest) element of the min-heap is copied to the output array.
- It is replaced by the next element of the same array, or ∞ if no more elements are left in the array.
- MIN-HEAPIFY is run to rebuild the min-heap.

After n iterations, the output array C will contain the sorted array. Here is the algorithm:

```
SORT(A, L, n, k)
1. for i = 1 to k
2.     B[i].element = A[L[i]]
3.     B[i].array = i
4.     B[i].index = L[i]
5. BUILD-MIN-HEAP(B, k)
6. for i = 1 to n
7.     C[i] = B[1].element
8.     if B[1].index + 1 == L[B[1].array + 1] then
9.         B[1].element = ∞
10.    else
11.        B[1].index = B[1].index + 1
12.        B[1].element = A[B[1].index]
13.    MIN-HEAPIFY(B, 1, k)
```

The complexity of the initialization phase is $\Theta(k)$ to copy the elements to the min-heap, and $O(k)$ to build the min-heap. The main loop runs n times, each time calling MIN-HEAPIFY with $O(\lg k)$ time. Therefore, the algorithm runs in $O(n \lg k)$ time.


1 2 3 4 5 7
7 5 3 4 2 1
BUILD-MAX-HEAP(A,6)
EXCHANGE A[1] WITH A[6]
MAX-HEAPIFY(A,1,5)
5 4 3 1 2 7
4 2 3 1 5 7
EXCHANGE A[1] WITH A[5]
MAX-HEAPIFY(A,1,4)
EXCHANGE A[1] WITH A[4]
MAX-HEAPIFY(A,1,3)
3 2 1 4 5 7
2 1 3 4 5 7
EXCHANGE A[1] WITH A[3]
MAX-HEAPIFY(A,1,2)
EXCHANGE A[1] WITH A[2]
1 2 3 4 5 7

5 *(*, previous exam question)* Consider the following problem:

> **INPUT:** A positive integer k and an array $A[1 \dots n]$ consisting of $n \geq k$ integers that satisfy the max-heap property, i.e., A is a max-heap.
>
> **OUTPUT:** An array $B[1 \dots k]$ consisting of the k largest integers of A sorted in non-decreasing order.

Design and **analyze** an efficient algorithm for the above problem. Ideally your algorithm should run in time $O(k \log k)$ but the worse running time of $O(\min\{k \log n, k^2\})$ is also acceptable.

Solution: Let us describe an algorithm that takes $\Theta(k \log k)$ time. First some intuition. Note that the largest element of the heap is located in $A[1]$. Therefore, we know that $B[k] = A[1]$. Now where can the second largest element be? By the max-heap property it can either be $A[2]$ or $A[3]$. Suppose that it is $A[2]$ then we put $B[k-1] = A[2]$. Now the 3rd largest element can either be $A[3]$ or one of the children of $A[2]$, i.e., $A[4]$ or $A[5]$ and so on. To find the largest of these elements we need to pick the maximum. To be able to pick the maximum quickly in each iteration we will keep track of a second max-heap called H. Our algorithm now works as follows:

> LargestK(A, B)
>
> 1. Create an empty heap H
> 2. $B[k] = A[1]$
> 3. Insert $A[2]$ and $A[3]$ to H
> 4. **for** $i = k-1, k-2 \dots, 1$
> 5. tmp = Extract-Max(H)
> 6. $B[i] = tmp$
> 7. Insert tmp's children in A to H, i.e.,
>
> if tmp is $A[i]$ then insert $A[2i]$ and $A[2i+1]$ to H.

Note that to implement Step 7 efficiently we need to be able to find the index i quickly so that tmp corresponds to i. This can easily be done by for example, storing the index along with the key in a tuple $(A[i], i)$ in the heap H (where $A[i]$ is the key).

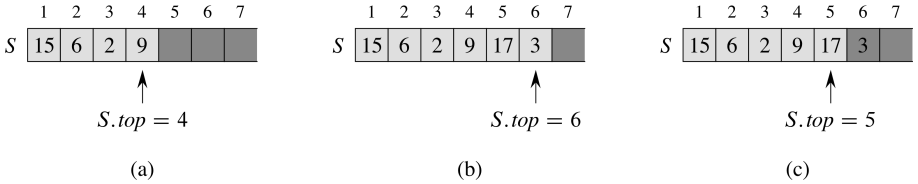
That the above works follows from the max-heap property. Indeed, the next largest element must be a child of already selected elements. To see this, suppose toward contradiction that the next largest element is not a child. But then its parent is larger and was not yet selected which is a contradiction (since then the parent would be the next largest element).

Let us now turn to the running time of our algorithm. To create an empty heap takes constant time. Similarly, Steps 2 and 3 take constant time. Step 5 takes at most $O(\log k)$ time as H will contain at most $2k$ elements. Step 6 takes constant time. Step 7 takes $O(\log k)$ time again because H contains at most $2k$ elements. Since Steps $5-7$ are executed k times the total running time will be $O(k \log k)$.

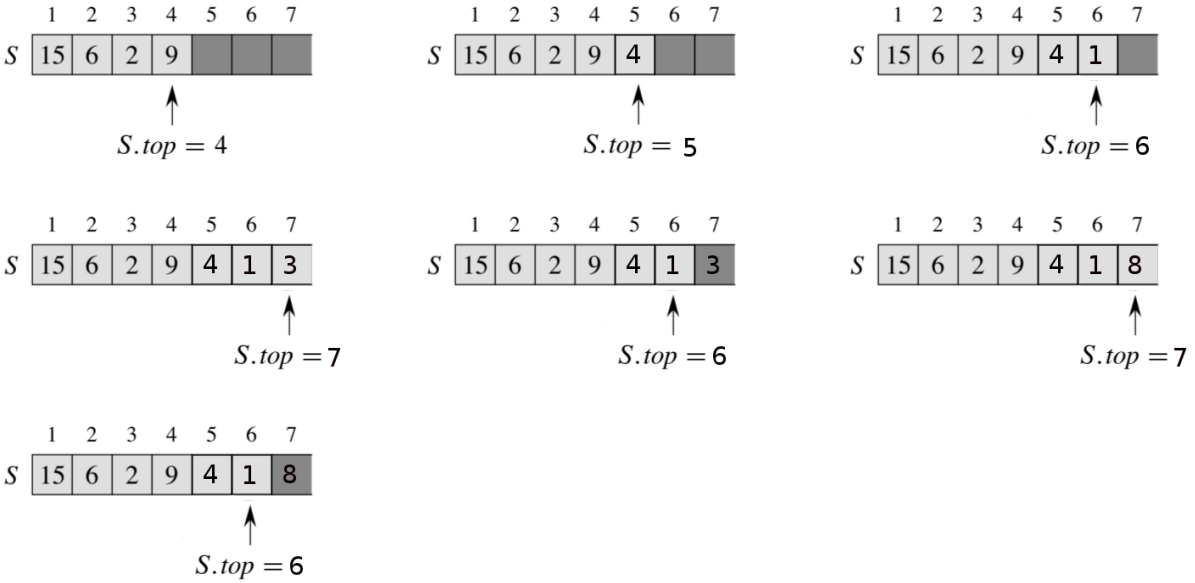
As a final comment, let me shortly express the idea of how to get an algorithm that runs in time $O(\min\{k^2, k \log n\})$ (which is worse than the previous running time). Observe that any element that we are interested in is of distance at most k from the root (this follows because all ancestors of an element are larger than that). Therefore we can restrict our attention to the heap $A[1 \dots \min\{2^k, n\}]$. Running k steps of heap sort on this restricted heap will take time $O(\min\{k^2, k \log n\})$.

Queues, Stacks, Lists

6 (Exercise 10.1-1 in the book) Using Figure 10.1 in the book (see below) as a model, illustrate the result of each operation in the sequence PUSH(S,4), PUSH(S,1), PUSH(S,3), POP(S), PUSH(S,8), and POP(S) starting from the stack depicted in (a).

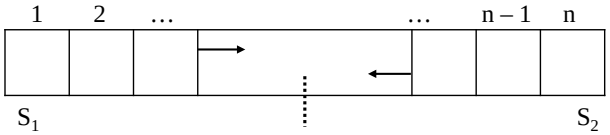


Solution:



7 (Exercise 10.1-2 in the book) Explain how to implement two stacks in one array $A[1 \ldots n]$ in such a way that neither stack overflows unless the total number of elements in both stacks together is n. The PUSH and POP operations should run in $O(1)$ time.

Solution: We will implement the two stacks $S1, S2$ in one array $A[1..n]$ by storing the i-th entry of stack $S1$ at the i-th entry of A and the j-th entry of stack $S2$ at the $(n-j+1)$-st entry of A:



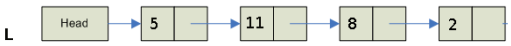
We use two pointers, $A.topS1$ and $A.topS2$, to point to the head of the stacks $S1$ and $S2$. We initialize $A.topS1$ and $A.topS2$ with 0 and $n+1$. If we try to pop $S1$ (or $S2$) but $S1$ (or $S2$) is empty (if $A.topS1 = 0$ or $A.topS2 = n+1$), then we say that stack $S1$ (or stack $S2$) **underflows**. If we try to push an element on $S1$ or $S2$ and the ends of the two stacks "meet", i.e., either $A.topS1 + 1 = A.topS2$ or $A.topS1 = A.topS2 - 1$, we say that the stack **overflows**. Note that there can be an overflow only if all n slots of the array A are occupied.

```
STACK-EMPTY-S1(A)
1   if A.topS1 = 0 then
2      return true;
3   else
4      return false;

STACK-EMPTY-S2(A)
1   if A.topS2 = n + 1 then
2      return true;
3   else
4      return false;

STACK-FULL-S1(A)
1   if A.topS1 + 1 = A.topS2 then
2      return true;
3   else
4      return false;

STACK-FULL-S2(A)
1   if A.topS2 − 1 = A.topS1 then
2      return true;
3   else
4      return false;

PUSH-S1(A,x)
1   if STACK-FULL-S1(A) then
2      // handle stack overflow
3   else
4      A.topS1 ← A.topS1 + 1;
5      A[A.topS1] ← x;

PUSH-S2(A,x)
1   if STACK-FULL-S2(A) then
2      // handle stack overflow
3   else
4      A.topS2 ← A.topS2 − 1;
5      A[A.topS2] ← x;

POP-S1(S)
1   if STACK-EMPTY-S1(A) then
2      // handle stack underflow
3   else
4      A.topS1 ← A.topS1 − 1;
5      return A[A.topS1 + 1];

POP-S2(S)
1   if STACK-EMPTY-S2(A) then
2      // handle stack underflow
3   else
4      A.topS2 ← A.topS2 + 1;
5      return A[A.topS2 − 1];
```

8 (Exercises 10.2-2 and 10.2-3 in the book)

8a Show how to implement a stack by a singly linked list (the operations PUSH and POP should still take $O(1)$ time).

Solution:

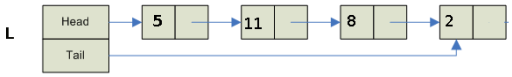


PUSH(L,x)
1 $x.next = L.head$
2 $L.head = x$

POP(L)
1 **if** $L.head \neq NIL$ **then**
2 $element = L.head$
3 $L.head = L.head.next$
4 **else**
5 $element = NIL$
6 **return** *element*

8b Show how to implement a queue by a singly linked list (the operations ENQUEUE and DEQUEUE should still take $O(1)$ time).

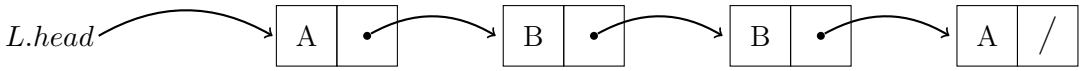
Solution:



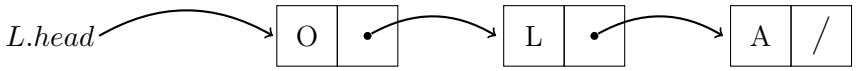
ENQUEUE(L,x)
1 **if** $L.tail \neq NIL$ **then**
2 $L.tail.next = x$
3 **else**
4 $L.head = x$
5 $L.tail = x$

DEQUEUE(L)
1 **if** $L.head \neq NIL$ **then**
2 $element = L.head$
3 $L.head = L.head.next$
4 **else**
5 $element = NIL$
6 **return** *element*

9 *(15pts)* **Palindrome.** A word is a palindrome if its reverse is equal to itself. For example, ABBA is a palindrome whereas OLA is not. One way of representing a word in a computer is to have a single-linked list where we have a list-element for each letter. For example, ABBA is represented by the single-linked list



and OLA is represented by the single-linked list



Design and **analyze** an algorithm that, given a pointer to the head of a single-linked list which represents a word, outputs YES if the word is a palindrome and NO otherwise.

For full score your algorithm should run in **linear time** and **should not use any other data structures than single-linked lists**, i.e., no arrays, stacks, queues, etc..

Solution: Let L be the linked list representing our word; the idea of the algorithm is to create a new linked list Q which contains the word of L reversed, and then compare the two words. We create the linked list Q by "walking through" the list L and for each element in L, inserting this element in the *head* of the list Q. At the end of this procedure, the list Q will be equal to the reverse of L.

The following pseudocode describes this procedure. Note that the function INSERT(Q,y) inserts the element y at the head of the linked list Q.

```
PALINDROME(L)
1. x ← L.head
2. Q ← new linked list
3. while x ≠NIL
4.       y ← new list element
5.       y.key← x.key
6.       INSERT(Q, y)
7.       x ← x.next
8. end while
9. x ← L.head
10. y ← Q.head
11. while x ≠NIL
12.       if x.key≠ y.key
13.             return NO
14.       x ← x.next
15.       y ← y.next
16. end while
17. return YES
```

Both while loops will be executed n times, where n is the length of the word. Since each loop consists of a constant number of operations, each of constant cost (recall that inserting into the head of a list takes constant time), the total running time will be $\Theta(n)$.

10 (*, Exercise 10.2-7 in the book) Give a $\Theta(n)$-time nonrecursive procedure that reverses a singly linked list of n elements. The procedure should use no more than constant storage beyond that needed for the list itself.

Solution:

Denote the singly linked list by L and suppose that the list contains n elements. The following non-recursive procedure needs constant additional storage (for $prev\_elem$, $curr\_elem$, and $next\_elem$) and takes time $\Theta(n)$ (as it performs a constant number of steps for each element of the singly-linked list):

```
REVERSE-LIST(L)
1   prev_elem ← NIL;
2   curr_elem ← L.head;
3   if L.head ≠ NIL then
4     next_elem ← L.head.next;
5     while curr_elem ≠ NIL do
6       curr_elem.next ← prev_elem
7       prev_elem ← curr_elem
8       curr_elem ← next_elem
9       if next_elem ≠ NIL then
10          next_elem ← next_elem.next
11    L.head ← prev_elem
```

Binary Search Trees

11 Give an $O(\log n)$-time algorithm that takes as input a *sorted* array $A[1 \dots n]$ of n numbers and a key k, and outputs "YES" if A contains the number k and "NO" otherwise.

Solution: As the numbers in A are sorted, we use the binary search algorithm. The binary search algorithm takes as input array A, key k, indices p and q and it will return "YES" if $A[p \dots q]$ contains the key k and "NO" otherwise. As $A[p \dots q]$ is sorted, we can compare k with the midpoint $mid = \lfloor \frac{p+q}{2} \rfloor$ and

1. if $A[mid] = k$ return "YES";
2. if $A[mid] > k$ then search for k in array $A[p \dots (mid-1)]$ by recursively calling BINARY-SEARCH$(A, k, p, mid-1)$;
3. if $A[mid] < k$ then search for k in array $A[(mid+1) \dots q]$ by recursively calling BINARY-SEARCH$(A, k, mid+1, q)$.

The pseudo-code of the procedure is as follows:

BINARY-SEARCH(A, k, p, q)

```
1. if q < p
2.    return "NO"        //array is empty so it doesn't contain k
3. else
4.    mid ← ⌊(p+q)/2⌋
5.    if A[mid] = k
6.       return "YES"
7.    elseif A[mid] > k
8.       return BINARY-SEARCH(A, k, p, mid − 1)
9.    else    // A[mid] < k
10.      return BINARY-SEARCH(A, k, mid + 1, q)
```

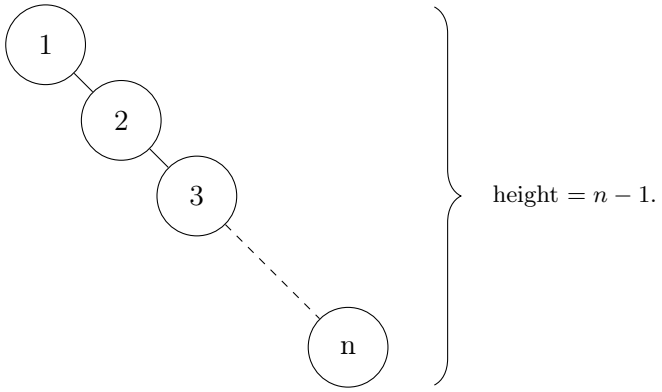
Note that we solve the original problem by calling BINARY-SEARCH$(A, k, 1, n)$. To analyze the running time we write down a recurrence of the running time of BINARY-SEARCH. If we let $N = q - p + 1$ be the size of array $A[p \dots q]$, then the time $T(N)$ it takes to execute BINARY-SEARCH(A, k, p, q) is given by the following recurrence

$$T(N) = \begin{cases} \Theta(1) & \text{if } N = 0, \\ T(N/2) + \Theta(1) & \text{otherwise.} \end{cases}$$

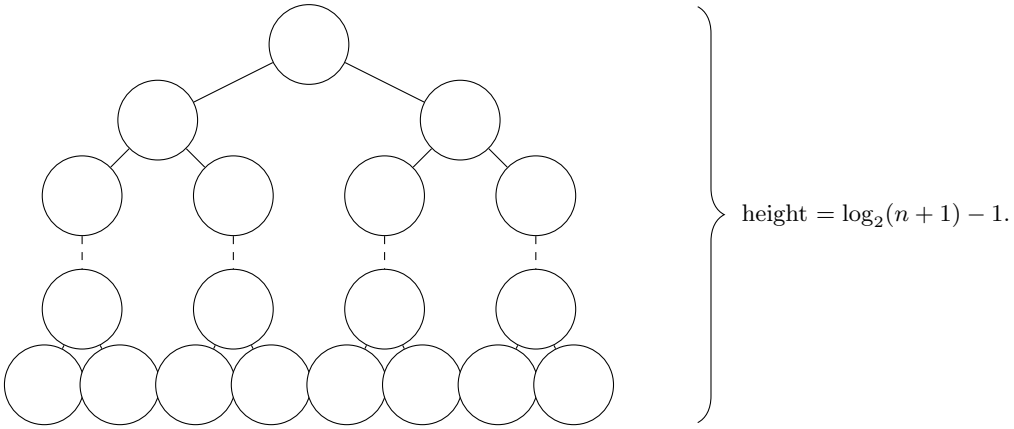
By the master method we have that $T(N) = \Theta(\log N)$ and hence BINARY-SEARCH$(A, k, 1, n)$ runs in time $\Theta(\log(n - 1 + 1)) = \Theta(\log n)$ as required.

12 What is the maximum and minimum height of a binary search tree of n elements? Which tree is better? Motivate your answers.

Solution: The maximum height is when the tree consists of only one branch:



The minimum height is when the binary tree is complete, i.e., we cannot add any node without increasing the height of the tree:

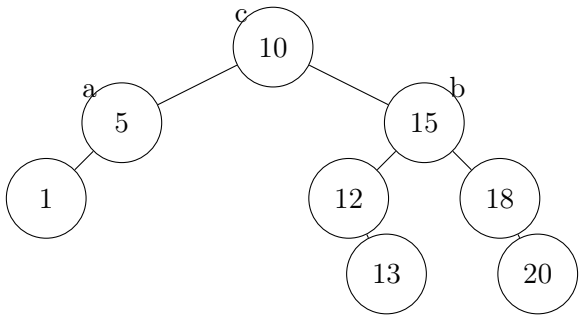


However, this works only if $n = 2^k - 1$ for some $k \in \mathbb{N}$. Otherwise, the tree will have some missing nodes, hence we have to round up the fractional solution. We will therefore get: height $= \lceil \log_2(n + 1) \rceil - 1$.

The second tree is better for a binary search tree, since we have to walk along the branches of our tree and in the worst case, we will make $\Theta(n)$ steps in the first tree, whereas we make only $O(\log n)$ in the second tree.

13 Consider the following binary search tree:

$T =$

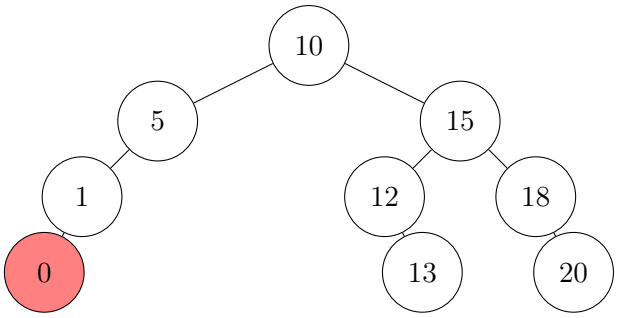


Draw the resulting trees obtained after executing each of the following operations (each operation is executed starting from the tree T above, i.e., they are *not* executed in a sequence):

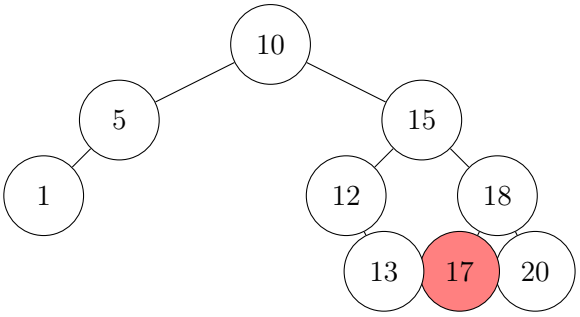
A: TREE-INSERT(T, z) where $z.key = 0$
B: TREE-INSERT(T, z) where $z.key = 17$
C: TREE-INSERT(T, z) where $z.key = 14$
D: TREE-DELETE(T, a)
E: TREE-DELETE(T, b)
F: TREE-DELETE(T, c)

Solution:

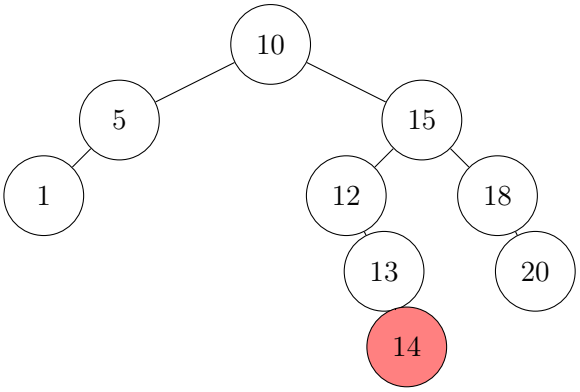
A:



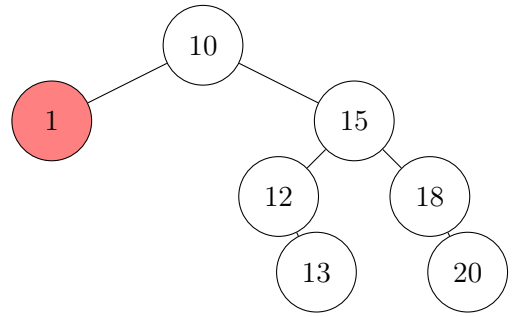
B:



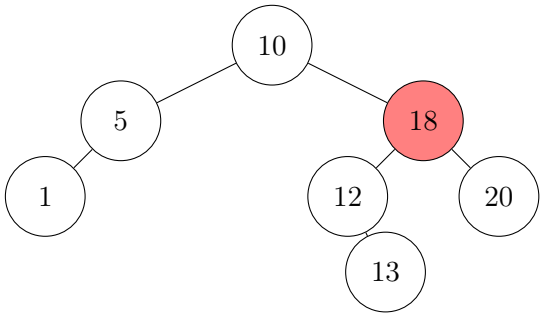
C:



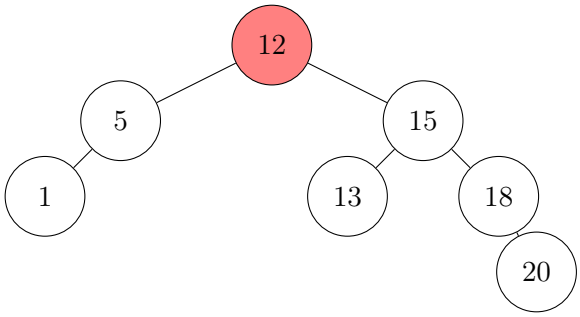
D:



E:



F:



14 (Exercise 12.1-3)

Give a nonrecursive algorithm that performs an inorder tree walk.

Hint: An easy solution uses a stack as an auxiliary data structure. A more complicated, but elegant, solution uses no stack but assumes that we can test two pointers for equality.

Solution: ***Variant 1: Using a stack***

The following iterative procedure uses a stack to perform an inorder traversal of a binary search tree, which does not use a visited flag. The auxiliary procedure PRINT($node$) displays the content of $node$.

```
INORDER-TREE-WALK(T)
1    S ← empty stack;
2    curr_node ← T.root;
3    while STACK-EMPTY(S)= false or curr_node ≠ NIL do
4      if curr_node ≠ NIL then
5        PUSH(S, curr_node);
6        curr_node ← curr_node.left;
7      else
8        curr_node = POP(S);
9        PRINT(curr_node);
10       curr_node = curr_node.right;
```

Variant 2: Using pointers (this is a little bit tricky)

The following iterative procedure, also known as Morris' algorithm[1], traverses a binary search tree without using a stack. It first creates links to the inorder successor of each node and prints the data using these links. It then reverts these changes.

```
INORDER-TREE-WALK(T)
1   curr_node ← T.root;
2   while curr_node ≠ NIL do
3     if curr_node.left = NIL then
4       PRINT(curr_node);
5       curr_node ← curr_node.right;
6     else
7       pre_node ← curr_node.left;
8       while pre_node.right ≠ NIL and pre_node.right ≠ curr_node do
9         pre_node ← pre_node.right;
10      if pre_node.right = NIL then
11        pre_node.right ← curr_node;
12        curr_node ← curr_node.left;
13      else
14        pre_node.right ← NIL;
15        PRINT(curr_node);
16        curr_node ← curr_node.right;
```

15 (Exercise 12.3-3) We can sort a given set of n numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worst-case and best-case running times for this sorting algorithm?

Solution: The algorithm is the following:

```
TREE-SORT(A)
1   Let T be an empty binary search tree
2   for i = 1 to n
3     TREE-INSERT(T, A[I])
4   INORDER-TREE-WALK(T.ROOT)
```

TREE-INSERT algorithm requires $\Theta(h)$ time, where h is a height of the tree. INORDER-TREE-WALK requires $\Theta(n)$ time, where n is a size of a tree.

Where can the difference between the best-case and the worst-case running time appear? INORDER-TREE-WALK running time depends only on the size of the tree and its contribution to the overall time is the same in all cases. This difference is introduced by the sequence of inserts, more precisely, by the changes of the height of the tree.

Consider the example, where we have a sorted array as an input. In this case each element will be assigned as a right child of the previous one. At all steps the tree has the form of a chain of elements, where all of the nodes have no left children and the height of this tree is equal to the number of nodes minus 1. Thus, the i^{th} insert requires $\Theta(i)$ time, and summing up all of them gives us $\Theta(n^2)$. The overall running time of the algorithm is $\Theta(n^2) + \Theta(n) = \Theta(n^2)$. This case is the worst one, as the height of the tree can not be greater than the number of nodes in it.

[1] See also *http://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion-and-without-stack/*

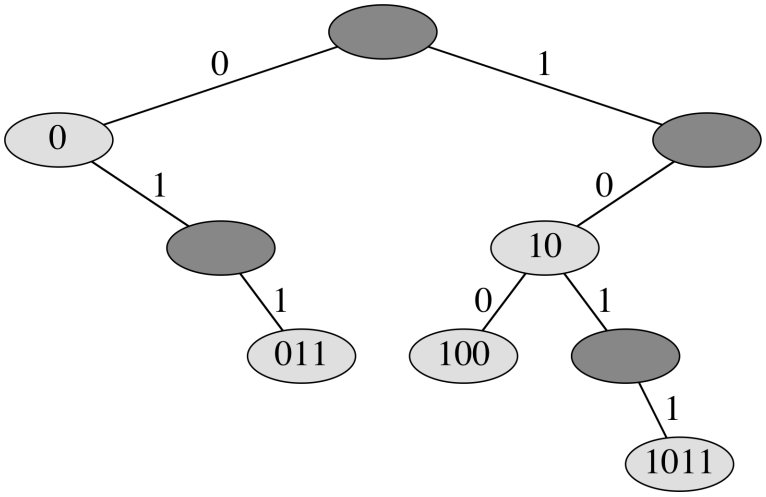


Figure 1. A radix tree storing the bit strings $1011, 10, 011, 100$, and 0. We can determine each node's key by traversing the simple path from the root to that node. There is no need, therefore, to store the keys in the nodes; the keys appear here for illustrative purposes only. Nodes are heavily shaded if the keys corresponding to them are not in the tree; such nodes are present only to establish a path to other nodes.

For the best case we should consider the tree with the smallest height. For the binary tree, this is logarithmic in the size of the tree. Thus summing up all of the inserts we receive $\Theta(n \log n)$, which is the overall best-case running time.

16 (*, Problem 12-2) **Radix trees**

Given two strings $a = a_0 a_1 ... a_p$ and $b = b_0 b_1 ... b_q$, where each a_i and each b_j is in some ordered set of characters, we say that string a is ***lexicographically less than*** string b if either

1. there exists an integer j, where $0 \leq j \leq \min(p, q)$, such that $a_i = b_i$ for all $i = 0, 1, ..., j-1$ and $a_j < b_j$, or
2. $p < q$ and $a_i = b_i$ for all $i = 0, 1, ..., p$.

For example, if a and b are bit strings, then $10100 < 10110$ by rule 1 (letting $j = 3$) and $10100 < 101000$ by rule 2. This ordering is similar to that used in English-language dictionaries.

The ***radix tree*** data structure shown in Figure 1 stores the bit strings $1011, 10, 011, 100$, and 0. When searching for a key $a = a_0 a_1 ... a_p$, we go left at a node of depth i if $a_i = 0$ and right if $a_i = 1$. Let S be a set of distinct bit strings whose lengths sum to n. Show how to use a radix tree to sort S lexicographically in $\Theta(n)$ time. For the example in Figure 1, the output of the sort should be the sequence $0, 011, 10, 100, 1011$.

Solution: Comparison of two bit strings is very simple with radix tree. The definition of the lexicographical order can be rewritten in the following way. String a is ***lexicographically less than*** string b if either

1. There exists a node n such that the path from the root to a goes through n and then to the left child of n i.e. a is in the left subtree of n, and the path from the root to b goes through n and then to the right child of n i.e. b is in the right subtree of n.
2. The node a is on the path from root to b.

In the first case strings are similar up to n and the next bit of a is 0, whereas the next bit of b is 1. In the second case a is simply a substring of b.

To sort the set of bit strings lexicographically, we need to use the preorder tree walk algorithm, which is similar to INORDER-TREE-WALK, but outputs the root before both subtrees. One additional modification to the tree walk is to print only nodes which represent the string from the set S. Here is an algorithm.

```
PREORDER-TREE-WALK(x)
1  if (x ≠ nil)
2     if (x.key represents the string from S) print x.key
3     PREORDER-TREE-WALK(x.left)
4     PREORDER-TREE-WALK(x.right)
```

The remaining thing is to build a radix tree. This can be made by inserting each string into the tree. For a string $a = a_0a_1...a_p$ we start from the root and go to the left if the current bit is 0 or to the right if the current bit is 1, creating all intermediate nodes if necessary.

Inserting of a string a requires $\Theta(a.length)$ running time. Thus, all inserts are made in time $\Theta(n)$. The number of nodes in the tree is less or equal to n, consequently the tree walk is also bounded by $\Theta(n)$.

17 *(old exam question)* **Quadtrees.** A quadtree is a search tree data structure in which each internal node has exactly four children. Quadtrees are usually used to partition the two-dimensional plane by recursively subdividing it into four quadrants or regions. In this exercise, we shall use quadtrees to represent cities according to their geographical position.

Figure 2 shows an example of 4 cities in western Switzerland. It also shows the 4 quadrants induced by the city Fribourg: the first, named NW (for *north-west*), contains the city Neuchâtel; the two following (NE for *north-east* and SE for *south-east*) are empty; the last (SW for *south-west*) contains the two remaining cities, Yverdon and Lausanne.

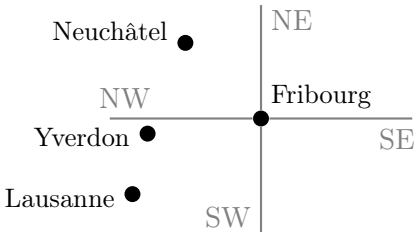


Figure 2. Geographical location of 4 cities in Switzerland

Each node v in the quadtree will store the name ($v.name$) and the coordinates ($v.x$ and $v.y$) of the cities. In addition, the node v will contain a pointer $v.p$ to its parent (or NIL if it's the root) and pointers $v.NW, v.NE, v.SE, v.SW$ to its four children. Similarly as with binary search trees, the key properties that make quadtrees useful are the following:

- if u is in the quadtree rooted by $v.NW$ then $u.x < v.x$ and $u.y \geq v.y$;
- if u is in the quadtree rooted by $v.NE$ then $u.x \geq v.x$ and $u.y > v.y$;
- if u is in the quadtree rooted by $v.SE$ then $u.x > v.x$ and $u.y \leq v.y$;
- if u is in the quadtree rooted by $v.SW$ then $u.x \leq v.x$ and $u.y < v.y$.

Figure 3 shows a possible quadtree representation of the cities in Figure 2. The root is the node corresponding to Fribourg that has x and y coordinates equal to 578461 and 183802, respectively.

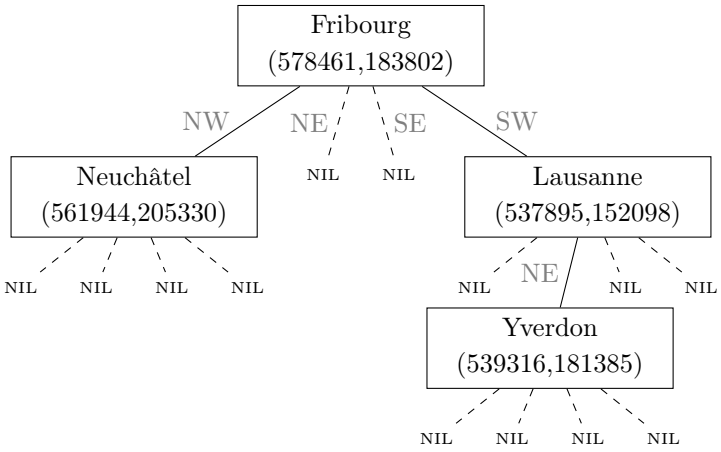


Figure 3. A possible quadtree representation of the cities in Figure 2

Design (give the pseudocode) and **analyze** the running times of the following procedures for the quadtree data structure:

> SEARCH(Q.root, x, y) — prints the name of the city located at (x, y) if such a city exists in the quadtree rooted at $Q.root$.
>
> PRINTSOUTHMOST(Q.root) — prints the name of the southmost city (the city with the smallest y coordinate) in the quadtree rooted at $Q.root$.

Your runtime analyses should be tight for the typical case when the height of the quadtree is logarithmic in the number of cities that it contains. In addition, for full score, the running times of your implementations should not be unnecessarily large.

Solution: Let us begin with the SEARCH operation; this operation will be very similar to searching in a binary tree, with the difference that now we have to operate along two dimensions. Specifically, we will design a recursive procedure, which given an input node and two coordinates x, y will compare these coordinates with the input node's coordinates, in order to decide in which direction the node we are searching for is; after this, we will apply SEARCH in the input node's child which is in the correct direction, and when we finally reach our target node, we will print its name.

```
SEARCH(Q.root, x, y)
1. u ← Q.root
2. if u =NIL
3.       return does not exist
4. if u.x = x and u.y = y
5.       return u
6. if u.x < x and u.y ≥ y
7.       SEARCH(u.SE,x, y)
8. if u.x ≥ x and u.y > y
9.       SEARCH(u.SW,x, y)
10. if u.x > x and u.y ≤ y
11.       SEARCH(u.NW, x, y)
12. if u.x ≤ x and u.y < y
13.       SEARCH(u.NE,x, y)
```

Each execution of SEARCH executes a constant number of operations branches out to at most one other execution of SEARCH, and the maximum recursion depth is h, where h is the height of the tree; hence, the total running time is $O(h)$.

Let us continue with the PRINTSOUTHMOST operation: here, we will again design a recursive procedure RETURNSOUTHMOST, which returns the y-coordinate and the name of the subtree rooted at the input node. It works as follows: first, it applies RETURNSOUTHMOST to both the southwest and the southeast subtrees of the input node, and it stores the y-coordinate of the most southern node in the southwest(southeast) subtree in variable $w(e)$. Then, it compares w and e in order to determine in which of the two subtrees the most southern node lies, and then returns its y coordinate and name(*wname* for the southwest subtree and *ename* fot the southeast tree).

```
RETURNSOUTHMOST(Q.root)
1. u ← Q.root
2. w, e ← ∞
3. if u.SW≠NIL
4.       (w, wname) ←RETURNSOUTHMOST(u.SW)
5. if u.SE≠NIL
6.       (e, ename) ←RETURNSOUTHMOST(u.SE)
7. if w ≤ e, u.y
8.       return (w, wname)
9. if e ≤ w, u.y
10.       return (e, ename)
11. if u.y ≤ e, w
12.       return (u.y, u.name)
```

Finally, in order to actually print the name of the most southern node, all we have to do is apply RETURNSOUTHMOST to the tree rooted at Q.root, and print out its name.

```
PRINTSOUTHMOST(Q.root)
1. (s, sname) ←RETURNSOUTHMOST(Q.root)
2. PRINT sname
```

Let us analyze its running time: this recursive procedure will visit each node at most once. Therefore, its running time is $O(n)$, where n is the number of nodes in the tree.

This is tight as we do two recursive calls in each step (RETURNSOUTHMOST($u.SW$) and RETURNSOUTHMOST($u.SE$)) and all remaining cities may be located in these subtrees. That is we have in the worst case a running time proportional to $T(n) = 2T(n/2) + \Theta(1)$ which is $\Omega(n)$ even if the height of the tree is logarithmic.