Algorithms and Data Structures I. Exercises

Ed. by **Tibor Ásványi**
Department of Computer Science
Eötvös Loránd University
Budapest, 2014.

asvanyi@inf.elte.hu

1.1 Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64n \lg n$ steps. For which values of n does insertion sort beat merge sort?

1.2 What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

2.1-1 Using the lecture as a model, illustrate the operation of INSERTION-SORT on the array
`A = < 31; 41; 59; 26; 41; 58 > .`

2.1-2 Rewrite the INSERTION-SORT procedure to sort into nonincreasing instead of nondecreasing order.

2.1-3 Consider the searching problem:
Input: An array of n numbers $A =< a_1; a_2; \ldots ; a_n >$ and a value v.
Output: An index i such that $v = A[i]$ or zero if v does not appear in A.
Write pseudocode ($T(n) \in O(n)$) for linear search, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

2.1-4 Consider the problem of adding two n-bit binary integers, stored in two n-element arrays A and B. The sum of the two integers should be stored in binary form in an $(n+1)$-element array C. Write pseudocode ($T(n) \in O(n)$) for adding the two integers.

2.2-1 Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ-notation.

2.2-2 Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. Then find the second smallest element of A, and exchange it with $A[2]$. Continue in this manner for the first $n-1$ elements of A. Write pseudocode for this algorithm, which is known as (minimum) selection sort. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements, rather than for all n elements? Give the best-case and worst-case running times of selection sort in Θ-notation.

2.2-3 Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence need to be checked in the best case? How about in the worst case? What are the best-case and worst-case running times ($mT(n)$ and $MT(n)$) of linear search in Θ-notation? Justify your answers.

2.2-4 How can we modify almost any algorithm to have a good best-case running time?

2.3-1 Using the lecture as a model, illustrate the operation of merge sort on the array
`A = <3; 41; 52; 26; 38; 57; 9; 49> .`

2.3-5 Referring back to the searching problem (see Exercise 2.1-3), observe that if the sequence A is sorted, we can check the midpoint of the sequence against v and eliminate half of the sequence from further consideration. The binary search algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode (rather iterative than recursive) for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

2.3-6

```
InsertionSort(A[1..n])
1  for j := 2 to n
   // Insert A[j] into the sorted sequence A[1..j-1].
2      if A[j] < A[j-1]
3          key := A[j]
4          A[j] := A[j-1]
           // Insert  key  into the
           // sorted sequence  A[1..j-2].
5          i := j-2
6          while i > 0 and A[i] > key
7              A[i+1] := A[i]
8              i := i-1
9          A[i+1] := key
```

Observe that the while loop of lines 6–8 of the InsertionSort procedure above uses a linear search to scan backward through the sorted subarray $A[1..j-2]$. Can we use a binary search (see Exercise 2.3-5) instead to improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

3.1-1 Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Using the basic definition of Θ-notation, prove that $max(f(n), g(n)) \in \Theta(f(n) + g(n))$.

3.1-2 Show that for any real constant a and natural number b, $(n+a)^b \in \Theta(n^b)$

3.1-3 Explain why the statement, "The running time of algorithm A is at least $O(n^2)$" is meaningless.

3.1-4 Is $2^{n+1} \in O(2^n)$? Is $2^{2n} \in O(2^n)$?

3.1-5 Using the basic definitions of Θ, Ω, and O-notation, prove that for any two functions $f(n)$ and $g(n)$, we have $f(n) \in \Theta(g(n))$ if and only if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.

3.1-6 Prove that the running time of an algorithm is $\Theta(g(n))$ if and only if its worst-case running time is $O(g(n))$ and its best-case running time is $\Omega(g(n))$.

3.1-7 Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.

6.1-1 What are the minimum and maximum numbers of elements in a heap of height h?

6.1-2 Show that an n-element heap has height $\lfloor \lg n \rfloor$.

6.1-3 Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.

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

6.1-5 Is an array that is in sorted order a min-heap?

6.1-6 Is the array with values
`< 23; 17; 14; 6; 13; 10; 1; 5; 7; 12 >` a max-heap?

6.1-7 Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1$, $\lfloor n/2 \rfloor + 2, \ldots, n$.

6.2-1 Let us suppose that array $A[1..n]$ is a nearly complete binary tree and the operation $sink(A, i, n)$ transforms the subtree with root reference i into max-heap provided that its left and right subtrees are already max-heaps. (See the lecture.) Illustrate the operation of $sink(A, 3, 14)$ on the array
`A = < 27;17;3;16;13;10;1;5;7;12;4;8;9;0 >` .

6.2-2 Starting with the procedure $sink(A[], i, n)$ above, modify the pseudocode so that it performs the corresponding manipulation on a minheap. How does the running time of the modified procedure compare to that of the original one?

6.2-3 What is the effect of the original operation $sink(A, i, n)$ when the element $A[i]$ is larger than its children?

6.2-4 What is the effect of calling $sink(A, i, n)$ for $i > n/2$?

6.3-1 Using the lecture as a model, illustrate the operation of building a max-heap on the array
$A =< 5; 3; 17; 10; 84; 19; 6; 22; 9 >$.

6.3-2 Consider the following procedure building a max-heap on the array $A[1..n]$ (where $floor(x) = \lfloor x \rfloor$ by definition).

```
BuildMaxHeap(A[1..n])
1  for i := floor(n/2) downto 1
2      sink(A,i,n)
```

Why do we want the loop index i in line 1 of $BuildMaxHeap(A[1..n])$ to decrease from $\lfloor n/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor n/2 \rfloor$?

6.3-3 Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

6.4-1 Using the lecture as a model, illustrate the operation of $HeapSort(A[1..n])$ on the array
`A = < 5; 13; 2; 25; 7; 17; 20; 8; 4 > .`

6.4-2

```
HeapSort(A[1..n])
1  BuildMaxHeap(A[1..n])
2  i := n
3  while i > 1
4      swap(A[1],A[i])
5      i := i-1
6      sink(A,1,i)
```

Argue the correctness of $HeapSort(A[1..n])$ using the following loop invariant:

At the start of each iteration of the loop of lines 3–6, the subarray $A[1..i]$ is a max-heap containing the i smallest elements of $A[1..n]$, and the subarray $A[i+1..n]$ contains the $n-i$ largest elements of $A[1..n]$, sorted.

6.5-1 Illustrate the operation removing the maximum of a priority queue on the max-heap
$A =< 15; 13; 9; 5; 12; 8; 7; 4; 0; 6; 2; 1 >$.

6.5-2 Illustrate the operation of the insertion of 10 into a priority queue on the max-heap
`A = < 15; 13; 9; 5; 12; 8; 7; 4; 0; 6; 2; 1 > .`
We suppose that the physical array containing the heap is long enough.

6.5-3 Write pseudocode for the procedures HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY, and MIN-HEAP-INSERT that implement a min-priority queue with a min-heap.

6.5-8 The operation $HeapDelete(A, m, i)$ deletes the item in node i from heap $A[1..m]$.

Give an implementation of $HeapDelete(A, m, i)$ that runs in $O(\lg m)$ time for an m-element max-heap.

7.1-1 Using the lecture as a model, illustrate the operation of $partition(A[p..r])$ of $QuickSort(A[p..r])$ on the following array. (We suppose that during the *partition* we want to separate the elements using the leftmost one.)
`A = < 11; 19; 9; 5; 12; 8; 7; 4; 21; 2; 6; 13 >` .

7.1-2 What value of q does $partition(A[p..r])$ return when all elements in the array $A[p..r]$ have the same value? Modify $partition(A[p..r])$ so that $q = \lfloor (p + r)/2 \rfloor$ when all elements in the array $A[p..r]$ have the same value.

7.1-3 Give a brief argument that the running time of $partition(A[p..r])$ on a subarray of size $n = r - p + 1$ is $\Theta(n)$.

7.1-4 How would you modify $QuickSort(A[p..r])$ to sort into nonincreasing order?

7.2-2 What is the running time of $QuickSort(A[1..n])$ when all elements of array $A[1..n]$ have the same value?

7.2-3 Show that the running time of $QuickSort(A[1..n])$ is $\Theta(n^2)$ when the array $A[1..n]$ contains distinct elements and is sorted in decreasing order.

7.2-4 Banks often record transactions on an account in order of the times of the transactions, but many people like to receive their bank statements with checks listed inorder by check number. People usually write checks in order by check number, and merchants usually cash them with reasonable dispatch. The problem of converting time-of-transaction ordering to check-number ordering is therefore the problem of sorting almost-sorted input. Argue that the procedure INSERTION-SORT would tend to beat the procedure QUICKSORT on this problem.

10.1-1 We suppose that `n=6` is a global integer constant and type `T=int`.

```
class Stack

  - A[0..n-1] : T
  - size : 0..n

  + Stack()
        size := 0

  + isEmpty():bool
        return(size=0)

  + isFull():bool
        return(size=n)

  + Push(x:T)
        if size < n
            A[size] := x
            size := size+1
        else
            error ‘‘stack_overflow’’

  + Pop():T
        if size > 0
            size := size-1
            return A[size]
        else
            error ‘‘stack_underflow’’
```

Using the definition of `Stack` above illustrate the result of each operation in the sequence

```
    S.Push(4); S.Push(1); S.Push(3);
    a:=S.Pop(); S.Push(8); b:=S.Pop()
```

on an initially empty `S : Stack`.

10.1-2 Explain how to implement two stacks in one array $A[1..n]$ 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.

10.1-3 We suppose that `n=6` is a global integer constant and type `T=int`.

```
class Queue{

  - A[0..n-1]:T
  - first : 0..n-1
  - length : 0..n

  + Queue()
        first := 0
        length := 0

  + isEmpty():bool
        return(length=0)

  + isFull():bool
        return(length=n)

  + enQueue(x:T)
        if length < n
            A[(first+length) mod n] := x
            length := length+1
        else
            error ‘‘queue_overflow’’

  + deQueue():T
        x : T
        if length > 0
            x := A[first]
            first := (first+1) mod n
            length := length-1
            return x
        else
            error ‘‘queue_underflow’’
```

Using the definition of `Queue` above, illustrate the result of each operation in the sequence

```
    Q.enQueue(4); Q.enQueue(1); Q.enQueue(3);
    a:=Q.deQueue(); Q.enQueue(8); b:=Q.deQueue();
```

on an initially empty `Q : Queue`.

10.2 Notations:
A formal parameter like +p means that p is a value parameter.
(Reference parameter is the default.)
SL = singly linked list (one-way, acyclic, no sentinel element)
HL = headed list (one-way, acyclic, sentinel head)
(The sentinel head is the zeroth element of the list.)
SYL = symmetrical list (two-way (i.e. doubly linked), circural, sentinel head)
We suppose that the list elements of a SYL have type list2node:

```
class list2node

  + a : data // if a,b:data then a<b or a>b or a=b
             //           and a:=b is well defined.
  - prev, next : list2node*

  + Exceptions : enum { PrecedeItselfErr,
      FollowItselfErr, PrecedeNullErr, FollowNullErr}

  + list2node()
  // *this becomes a cyclic list of a single element

  + pre():list2node*
  // return the address of the previous element

  + suc():list2node*
  // return the address of the successor element

  + out()
  // *this is detached, it becomes a cyclic list itself

  + precede(+p:list2node*)
  // if this=p or p=0 then an exception is raised
  // else *this is detached and inserted left to *p

  + follow(+p:list2node*)
  // if this=p or p=0 then an exception is raised
  // else *this is detached and inserted right to *p

  + virtual ~list2node()
  // *this is detached and deleted
```

10.2-1 Can you implement the dynamic-set operation INSERT on a singly linked list (SL or HL) in $O(1)$ time? How about DELETE? What if we want to avoid double and multiple elements? Write pseudocode for these operations. Give the best-case and worst-case running times of your programs in Θ-notation.

10.2-2 Implement a stack using a singly linked list (SL) L. The operations PUSH and POP should still take $O(1)$ time.

10.2-3 Implement a queue by a singly linked acyclic list L. The operations ENQUEUE and DEQUEUE should still take $O(1)$ time. (You may use some sentinel node if you want.)

10.2-4 Implement a queue by a singly linked circural list L (using sentinel head). The operations ENQUEUE and DEQUEUE should still take $O(1)$ time. (You may change the sentinel in some operation.)

10.2-5 Implement the dictionary operations INSERT, DELETE, and SEARCH using singly linked, circular lists (with sentinel head). What are the running times of your procedures? Try the same using SYLs.

10.2-6 The dynamic-set operation UNION takes two disjoint sets S_1 and S_2 as input, and it returns a set $S = S_1 U S_2$ consisting of all the elements of S_1 and S_2. The sets S_1 and S_2 are usually destroyed by the operation. Show how to support UNION in $O(1)$ time using a suitable list data structure.

10.2-7 Give a $\Theta(n)$-time nonrecursive procedure that reverses a singly linked list (SL or HL) of n elements. The procedure should use no more than constant storage beyond that needed for the list itself.

10.2-x.1 Rewrite the INSERTION-SORT procedure on HL L to sort into nonincreasing instead of nondecreasing order.

10.2-x.2 Rewrite the INSERTION-SORT procedure on SYL L to sort into nonincreasing instead of nondecreasing order.

10.2-x.3 Consider sorting HL L by first creating an empty SL $L2$. Then repeat the following loop while L is a nonempty HL: find and remove the greatest element of L and insert it at the front of $L2$. At last connect $L2$ after the sentinel head of L. Write pseudocode for this algorithm, which is known as (maximum) selection sort. What loop invariant does this algorithm maintain? Give the best-case and worst-case running times of (maximum) selection sort in Θ-notation.

10.2-x.4 Consider sorting SYL L by first dividing the list into an unsorted part (containing initially each item) and an initially empty sorted end. (This division can be done by a pointer referring to the beginning of the sorted part.) Then repeat the following loop while the unsorted part of L contains at least two nodes: find the greatest element of the unsorted part of L and put it to the front of the sorted part. Write pseudocode for this algorithm, which is known as (maximum) selection sort. What loop invariant does this algorithm maintain? Give the best-case and worst-case running times of (maximum) selection sort in Θ-notation.

10.2-x.5 Consider sorting SYL L by first dividing the list into an (initially empty) sorted beginning and an unsorted end (containing initially each item). (This division can be done by a pointer referring to the end of the sorted part). Then repeat the following loop while the unsorted part of L contains at least two nodes: find the smallest element of the unsorted part of L and put it to the end of the sorted part. Write pseudocode for this algorithm, which is known as (minimum) selection sort. What loop invariant does this algorithm maintain? Give the best-case and worst-case running times of (minimum) selection sort in Θ-notation.

10.4 By default, the type of the nodes of a binary tree is `Node`:

```
class Node{ +k:key; +left,right:Node* }
```

If there is a parent pointer p, this type is `NodeP`:

```
class NodeP{ +k:key; +left,right,p:NodeP* }
```

10.4-1 Draw the binary tree rooted at index 6 that is represented by the following attributes:

index	key	left	right
1	12	7	3
2	15	8	0
3	4	10	0
4	10	5	9
5	2	0	0
6	18	1	4
7	7	0	0
8	14	6	2
9	21	0	0
10	5	0	0

10.4-2 Write an $O(n)$-time recursive procedure that, given an n-node binary tree, prints out the key of each node in the tree.

10.4-3 Write an $O(n)$-time nonrecursive procedure that, given an n-node binary tree, prints out the key of each node in the tree. Use a stack as an auxiliary data structure.

10.4-4 There is a clever scheme to represent trees where each node may have arbitrary number of children. It uses only the same amount of space for any n-node rooted tree as for an n-node binary tree. This is the left-child, right-sibling representation:
`class Node{ +k:key; +left_child,right_sibling:Node* }`
Instead of having a pointer to each of its children, each node `(*q)` has only two pointers:
1. `q->left_child` points to the leftmost child of node `(*q)`, and
2. `q->right_sibling` points to the sibling of `(*q)` immediately to its right.
If node `(*q)` has no children, then `q->left_child=0`, and if node `(*q)` is the rightmost child of its parent, `q->right_sibling=0`. (In some programs the nodes also contain a parent pointer p, but it is supefluous in this exercise, and in many other cases.)

Write an $O(n)$-time procedure that prints all the keys of an arbitrary rooted tree with n nodes, where the tree is stored using the left-child, right-sibling representation. (Try to print the key of any node before the keys of its children.)

10.4-5* Write an $O(n)$-time nonrecursive procedure that, given an n-node binary tree (with a parent pointer in each node), prints out the key of each node. Use no more than constant extra space outside of the tree itself and do not modify the tree, even temporarily, during the procedure.

10.4-6* Let us suppose that the left-child, right-sibling representation of a rooted tree uses three pointers in each node: left-child, right-sibling, and parent. From any node, its parent can be reached and identified in constant time and all its children can be reached and identified in time linear in the number of children. Show how to use only two pointers and one boolean value in each node so that the parent of a node or all of its children can be reached and identified in time linear in the number of children.

10.4-x The textual representation of a nonempty binary tree is the following:
(leftSubtree Root rightSubtree)
The empty tree is represented by the empty string.
The lexemes of a textual representation are
- the opening bracket,
- the closing bracket,
- and the roots of the subtrees.

For example, a tree is represented by the string
“(((1)2)3((4(5))6(7)))”
if and only if it can be drawn like this:

```
         3
        / \
       2   6
      /   / \
     1   4   7
          \
           5
```

10.4-x.1 Give recursive pseudocode which prints the textual form of tree `t:Node*`. $T(t) \in \Theta(n(t))$.

10.4-x.2* Given the textual form of a nonempty binary tree in sequential file F. Write recursive pseudocode building the linked representation of the tree in $\Theta(n)$ time where n is the number of lexemes of the textual form. We suppose that we have a read statement that can read a lexeme x in time $\Theta(1)$. And we can check whether $x =' ('$ or $x =')'$ also in constant time.

12 A *binary search tree* is a binary tree satisfying the following property: Let `(*x)` be a node in a binary search tree. If `(*y)` is a node in the left subtree of `(*x)`, then `y->key < x->key`. If `(*y)` is a node in the right subtree of `(*x)`, then `y->key > x->key`.

A *binary sort tree* is a binary tree satisfying the following property: Let `(*x)` be a node in a binary search tree. If `(*y)` is a node in the left subtree of `(*x)`, then `y->key =< x->key`. If `(*y)` is a node in the right subtree of `(*x)`, then `y->key >= x->key`.

Note: We adopt the default introduced in 10.4 for the nodes of the binary trees.

12.1-1 For the set of `{1; 4; 5; 10; 16; 17; 21}` of keys, draw binary search trees of heights 2, 3, 4, 5, and 6.

12.1-2 What is the difference between the binary-search-tree property and the max-heap property (see page 152 in [1])? Can the max-heap (or min-heap) property be used to print out the keys of an n-node tree in sorted order in $O(n)$ time? Show how, or explain why not.

Note: The sorting algorithms like insertion sort, merge sort, quicksort, maximum selection sort, minimum selection sort, and heapsort share an interesting property: the sorted order they determine is based only on comparisons between the input elements. We call such sorting algorithms *comparison* *sorts*. In Section 8.1 of [1] it is proved that any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case to sort n elements. (Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.)

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.)

12.1-4 Give recursive algorithms that perform preorder and postorder tree walks in $O(n)$ time on a tree of n nodes.

12.1-5 Argue that since sorting n elements takes $\Omega(n \lg n)$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of n elements takes $\Omega(n \lg n)$ time in the worst case.

12.2-1 Suppose that we have numbers between 1 and 1000 in a binary search tree, and we want to search for the number 363. Which of the following sequences could not be the sequence of nodes examined?
a. 2, 252, 401, 398, 330, 344, 397, 363.
b. 924, 220, 911, 244, 898, 258, 362, 363.
c. 925, 202, 911, 240, 912, 245, 363.
d. 2, 399, 387, 219, 266, 382, 381, 278, 363.
e. 935, 278, 347, 621, 299, 392, 358, 363.

12.2-2 Write recursive versions of functions `TREE_MINIMUM(+t:Node*)` and `TREE_MAXIMUM(+t:Node*)` returning a pointer to the node containing the minimal/maximal key in the nonempty tree `t`. $T(t) \in O(h(t))$

12.2-3a Write the function `TREE_SUCCESSOR(+p:NodeP*)` where `p` refers to some node of a binary search tree. (There is a parent pointer in each node of the tree.) The function must return a pointer to the node containing the smallest key which is greater than the key of `(*p)` in time $O(h)$ where h is the height of the tree. Actually the pointer returned also identifies the successor of node `(*p)` in the inorder traversal of the tree.

12.2-3b Write the function `TREE_PREDECESSOR(+p:NodeP*)` where `p` refers to some node of a binary search tree. (There is a parent pointer in each node of the tree.) The function must return a pointer to the node containing the greatest key which is smaller than the key of `(*p)` where h is the height of the tree. Actually the pointer returned also identifies the predecessor of node `(*p)` in the inorder traversal of the tree.

12.2-4 Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Bunyan claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a < b < c$. Give a smallest possible counterexample to the professor's claim.

12.2-5 Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

12.2-6 Consider a binary search tree T. (Remember that its keys are distinct.) Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. (Recall that every node is its own ancestor.)

12.2-7 An alternative method of performing an inorder tree walk of an n-node binary search tree (with root pointer `t`) finds the minimum element in the tree by performing `p:=TREE_MINIMUM(t)` and then making $n-1$ calls to `p:=TREE_SUCCESSOR(p)`. Prove that this algorithm runs in $\Theta(n)$ time.

12.2-8 Prove that no matter what node we start at in a height-h binary search tree, k successive calls to `p:=TREE_SUCCESSOR(p)` take $O(k+h)$ time.

12.2-9 Let T be a binary search tree. (Remember that its keys are distinct.) Let `(*x)` be a leaf node, and let`(*y)` be its parent. Show that $y- > key$ is either the smallest key in T larger than `x->key` or the largest key in T smaller than `x->key`.

12.3-1a At the lecture we have seen the recursive version of inserting a value into a search tree. We supposed that the nodes of the tree do not have parent pointers. Modify this procedure, in order to handle the parent pointers, too.

12.3-1b Make the same modification on our recursive procedure deleting the node "containing" a given key.

12.3-2 Write a recursive procedure inserting a value into a binary sort tree t. Prove that $T(t) \in O(h(t))$ and $MT(t) \in \Theta(h(t))$.

12.3-3 We can sort a given set of n numbers by first building a binary sort tree containing these numbers (using the tree insertion in 12.3-2 repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. Prove that for this sorting method $mT(n) \in O(n \lg n)$ and $MT(n) \in \Omega(n^2)$.

12.3-4* Is the operation of deletion "commutative" in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why it is or give a counterexample.

12.3-5 Suppose that the type of the nodes of a binary search tree is `NodeS`:
`class NodeS{ +k:key; +left,right,s:NodeS* }`
where the `s` pointers are undefined. Give pseudocode to ensure that the attribute `s` in each node refers to the successor of the node according to the inorder traversal.

12.3-6 When we want to delete a key from a binary search tree, and the node z containing it has two chidren, we delete its successor y (the leftmost node in the right subtree of z) and copy the data in y into z. In this procedure we could choose node y as z's predecessor (the rightmost node in the leftt subtree of z) rather than its successor. What other changes to this deleting procedure would be necessary if we did so? Some have argued that a fair strategy, giving equal priority to predecessor and successor, yields better empirical performance. How might this deleting procedure be changed to implement such a fair strategy?

12.3-x Let us suppose that the inorder traversal of a binary tree prints the keys of the tree in a strictly increasing manner. Prove that the tree is a binary search tree.

13.x-1 Given an initially empty AVL tree t. Using the lecture as a model, illustrate the insertion of numbers 1 2 7 3 5 6 8 9 4 into t in the given order.

Starting with the resulting AVL tree t, illustrate the deletion of numbers 1 3 4 8 9 2 from t in the given order.

Redraw the tree after each insertion/deletion. If balancing is required, show which subtree must be balanced, and after each balancing also redraw the tree. On each drawing show the balances of the nodes in the manner you have seen at the lectures.

13.x-2 Draw an AVL tree with 12 nodes which has the maximum height of all 12-node balanced trees. Give an example sequence of the keys, in which order the keys of the nodes inserted into an initially empty AVL tree, we receive the tree drawn.

18.x Textual representation of B+-trees:
Here we omit the satellite data of the B+-trees. We illustrate only their structure and keys. The textual form of a leaf of a B+-tree: $[k_1, \ldots, k_b]$ where each k_j is a key. The textual form of a nonleaf subtree of a B+-tree: $(T_1 s_1 T_2 \ldots s_{d-1} T_d)$ where each T_i is a subtree, and each s_j is a split-key.

18.x-1 Let us suppose that we have the following B+-tree of degree 4.
```
( [1,4,9] 16 [1,25] )
```
Using the lecture as a model, illustrate the insertion of numbers 20, 13, 15, 10, 11, 12 into it, in the given order.

18.x-2 Let us suppose tha we have the following B+-tree of degree 4.
```
( ( [1,4] 9 [9,10] 11 [11,12] ) 13 ( [13,15] 16 [16,20,25] ) )
```
Using the lecture as a model, illustrate the deletion of numbers 13, 15, 1 from it, in the given order.

Keys to Selected Exercises

1.1 $n \leq 43$

1.2 $n \geq 15$

2.2-1 $n^3/1000 - 100n^2 - 100n + 3 \in \Theta(n^3)$

2.2-3 In linear search (see Exercise 2.1-3) just the first element of the input sequence need to be checked in the best case: this may be the the item we look for. Each element must be checked in the worst case: we might not find the item we look for, or find it as the last element of the sequence. Thus $mT(n) \in \Theta(1)$ and $MT(n) \in \Theta(n)$.

3.1-1 Let n_0 be such a natural number that for each $n \geq n_0$, $f(n) \geq 0$ and $g(n) \geq 0$. Let $n \geq n_0$. Then

$$max(f(n), g(n)) = \frac{f(n) + g(n)}{2} + \frac{|f(n) - g(n)|}{2}$$

and $f(n) + g(n) \geq |f(n) - g(n)|$. Thus

$$\frac{f(n) + g(n)}{2} \leq max(f(n), g(n)) \leq f(n) + g(n)$$

.

10.4-5* Write procedure `preorder_next(p)`.
Call `print_key` and `preorder_next(p)` repeatedly. Prove that n iterations need $\Theta(n)$ time if n is the size of the tree.

12.1-2 In a max-heap, a node's key is $\geq$ both of its children's keys. In a binary search tree, a node's key is $>$ its left subtree's keys, but $<$ its right subtree's keys. The max-heap property, unlike the binary-search-tree property, doesn't help print the nodes in sorted order because in a heap, the largest element smaller than a node could be in either subtree. Note that if the max/min-heap property could be used to print the keys in sorted order in $O(n)$ time, we would have an $O(n)$-time algorithm for sorting, because building the heap takes only $O(n)$ time. But we know that a comparison sort must take $\Omega(n \lg n)$ time.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to Algorithms* (Third Edition), The MIT Press, 2009.
`http://aszt.inf.elte.hu/~asvanyi/ds/`
`Introduction_to_algorithms_3rd_edition.pdf`
`Introduction_to_Algorithms_Cormen.pdf`

[2] Niklaus Emil Wirth, *Algorithms and Data Structures*
(Oberon version: August 2004.)
`http://aszt.inf.elte.hu/~asvanyi/ds/AD.pdf`

[3] Dr. Carl Burch, *B+-trees*
`http://aszt.inf.elte.hu/~asvanyi/ds/B+trees.zip`