Institut for Matematik og Datalogi
Syddansk Universitet, Odense

08. May 2025

Exercise Sheet Week 12

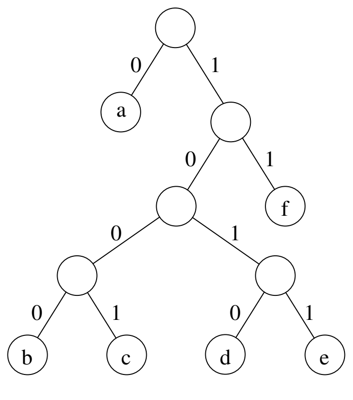
DSK814

A(I). Solve in the practice sessions

1. Consider the alphabet with the six characters a,b,c,d,e,f. The table below shows how often each character appears in a given text. Draw the tree representing the Huffmann codes for this example.

Character	a	b	c	d	e	f
Frequency	33	28	52	20	10	12

2. Use the following Huffman tree to decode the string 1101001010101011.



3. Consider a file containing the characters below with the specified frequencies.

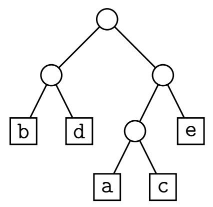
Character	a	b	c	d	e
Frequency	100	150	150	250	350

Tree H_1 is a Huffman tree for the this file.

a) Specify how many bits it takes to encoded the file as tree H_1.

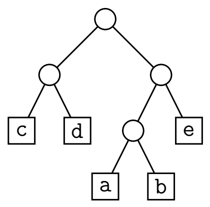
b) Decode the string 1000000110110101 using the tree H_1 and the convention that 0 corresponds to left and 1 corresponds to right.

H_1:

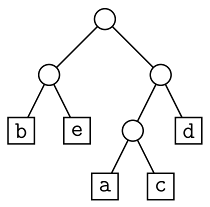


c) All trees H_2, H_3, H_4 and H_5 are optimal for the file with the table above. Specify which of the trees can be obtained by Huffman's algorithm.

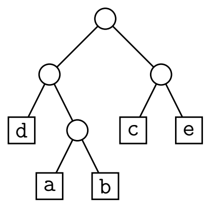
H_2:



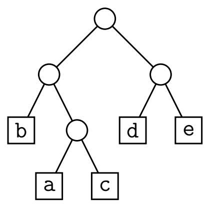
H_3:



H_4:



H_5:



□

4. Consider disjoint sets that are implemented via chained lists and use a weighted-union heuristic for UNION (i.e. the shortest list is appended to the longest list).

   a) Cormen et al., 4. Edition: Exercise 19.2-2 (page 526) [Cormen et al., 3. Edition: Exercise 21.2-2 (page 567)].

For the following series of operations, draw the state of the data structure after each operation and which element the two FIND-SET operations operations return.

```
for $i = 1$ to 16
    MAKE-SET($x_i$)
for $i = 1$ to 15 by 2
    UNION($x_i, x_{i+1}$)
for $i = 1$ to 13 by 4
    UNION($x_i, x_{i+2}$)
UNION($x_1, x_5$)
UNION($x_{11}, x_{13}$)
UNION($x_1, x_{10}$)
FIND-SET($x_2$)
FIND-SET($x_9$)
```

If the list with x_i and the list with x_j have the same length in an operation UNION(x_i, x_j), then append the list with x_j to the x_i list. Your drawings do not need to be as detailed as in Cormen et al., 4. Edition: Figure 19.2 (page 524) [Cormen et al., 3. Edition: Cormen et al., 3. Edition: Figure 21.2 (page 565)].

b) Describe an implementation where the header objects do not contain a tail pointer and do not store the list length. Note that the asymptotic running time of the operations should not be changed.

   Hint: Go through the lists synchronously. Remember to update the elements' pointers to the header.

5. Cormen et al., 4. Edition: Exercise 19.3-1 (page 531) [Cormen et al., 3. Edition: Exercise 21.3-1 (page 572)].

   Repeat exercise 4a) with disjoint sets that are implemented via trees using both union by rank and path compression. Remember to specify the rank for nodes in your drawings.

6. Cormen et al., 4. Edition: Exercise 19.3-2 (page 531) [Cormen et al., 3. Edition: Exercise 21.3-2 (page 572)].

   Create a version of FIND-SET that does not use recursion for disjoint sets that are implemented via trees.

   Hint: Run through the path twice.

A(II). Solve in the practice sessions

1. Cormen et al., 4. Edition: Exercise 20.1-1 (page 552) [Cormen et al., 3. Edition: Exercise 22.1-1 (page 592)].

   Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?

2. Cormen et al., 4. Edition: Exercise 20.1-3 (page 553) [Cormen et al., 3. Edition: Exercise 22.1-3 (page 592)].

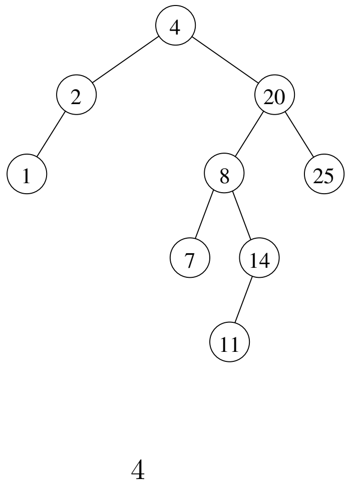
   The transpose of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. Thus, G^T is graph G with all its edges reversed. Describe efficient algorithms for computing G^T from G, for both the adjacency-list and adjacency-matrix representations of G. Analyze the running times of your algorithms.

3. Consider sorting integers between 0 and n^5 (where n is the number of keys to be sorted) in the following two special cases.

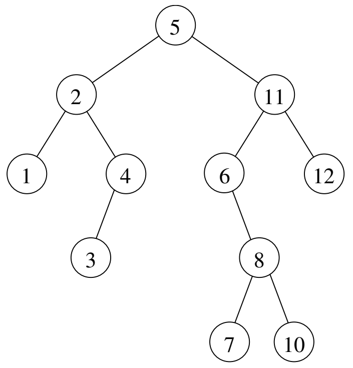
   - *Case 1:* All keys are different and sorted in reverse as in the example $\{12, 9, 8, 5, 2\}$.
   - *Case 2:* All keys are the same as in the example $\{5, 5, 5, 5, 5\}$.

   For both cases, provide the running time of InsertionSort, QuickSort and RadixSort (best possible).

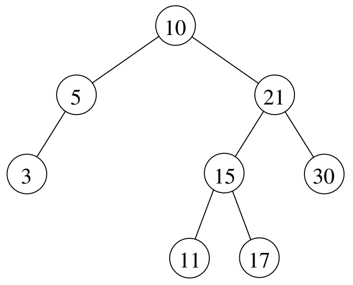
4. Is the following tree a binary search tree?



5. Consider the following binary search tree. Draw the tree after deleting the node with key 5.



6. Consider the following binary search tree. Draw the tree after inserting the node with key 16.

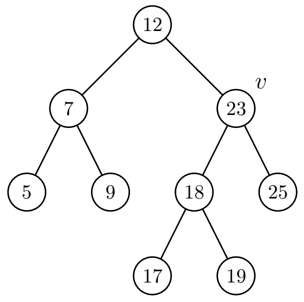


7. Perform Heap-Extract-Max on the binary heap rerepresented by the array $[10, 7, 6, 5, 4, 2, 3, 1, 2, 3, 1, 1]$. Show each step and consider drawing the tree representation rather than the array.

8. Consider extending binary search trees with information about the distances between the stored keys. In particular, we want to find the largest distance between a key and its predecessor in the tree. If a search tree stores n keys $x_1 \leq x_2 \leq ... \leq x_n$, then the searched distances are $(x_2 - x_1), (x_3 - x_2), ..., (x_n - x_{n-1})$. For example, if the keys are $3, 5, 11, 14, 23$ and 30, then the distances are $2, 6, 3, 9$ and 7. Thus, the largest distance is 9 (between 23 and its predecessor 14). In addition to $v.key$, each node v stores the following three additional pieces of information

- *v.maxGap*: the maximum distance between keys stored in *v*'s subtree
- *v.max*: the largest key stored in *v*'s subtree
- *v.min*: the smallest key stored in *v*'s subtree

A node's subtree includes the node itself. If there is only one key in *v*'s subtree, then $v.maxGap = 0$. In particular, the largest distance in the tree can be inferred from the root *r*'s information *r.maxGap* in $O(1)$ time. Consider an example of such a binary search tree below.



$v.key$	=	23
$v.maxGap$	=	4
$v.max$	=	25
$v.min$	=	17

a) Describe how the additional information of node *v* can be determined from its two children in time $O(1)$, including the keys of both *v* and its children. You do not need to describe the special case where one or both of the children are NIL.

b) Consider a red-black tree. Argue that the information in the nodes of the tree can be maintained during insertions and deletions without changing the running time of $O(\log n)$.

c) Consider a red-black tree. The largest distance in the tree between keys and their predecessors can be found by reading the root *r*'s information *r.maxGap*. Describe a search process that find a specific key that has this distance to its predecessor in time $O(\log n)$.

B(I). Solve at home before tutorial in week 13

The tasks below are all warm-up/help for the project part III.

1. A byte (a group of eight bits) is the smallest unit of data that CPUs can handle. A file is just a series of bytes, and therefore always contains a multiple of eight bits.[1]

   There are $2^8 = 256$ possibilities for the content of one byte: 00000000, 00000001, 00000010, 00000011, ..., 11111111. In Python, bytes are represented by byte objects, which can be thought of as (immutable) lists of integers with values between 0 and 255.[2]

   In Python, calling `read(1)` from file objects reads bytes from a file one at a time, if the file is opened in binary reading mode using the argument `'rb'` in `open`. Each byte is returned as a byte object of length one. If you access the first (and only) element of this byte object, you get an integer between 0 and 255. If the byte object is named `b`, the first element is accessed as `b[0]`. When you have reached the end of the file, `read(1)` returns a byte object of length zero instead of length one.

   Create a Python program that: i) reads a file one byte at a time, ii) counts how many of each of the 256 possible bytes the file contains, and iii) print a table similar to the following:

   ```
   Byte 0: 0
   Byte 1: 0
   .
   .
   Byte 97: 7
   Byte 98: 4
   .
   .
   Byte 255: 0
   ```

   Use the `read(1)` method from Python's file objects. Each read byte (an integer between 0 and 255) should be used as an index in a list

[1] A program that reads a file will usually interpret its bits as representing, for example, letters, pixels or audio oscillations (each program interprets in its own way). However, a file itself is just a series of bits grouped in groups of size eight (bytes).

[2] Slightly confusing, an entire byte object in Python is represented using ASCII characters if possible. Thus, a byte object `s` with content `[120,121,122]` is represented as `b'xyz'` (with `b` for "binary"), since the character `x` has number 120 in the ASCII table, but individual elements in byte objects *are* integers (e.g. is `s[0]` equal to `120`).

of length 256, where index i acts as a counter for byte number i. You need to read bytes one at a time and do the count on the fly.[3].

Try the program on some simple `.txt` files (without Danish letters) and use a table of the ASCII code to check the output. Also try the program on other than text files, such as `.jpg` files and `.doc` files.

2. A byte (a group of eight bits) is the smallest unit of data that CPUs can handle. Therefore, accessing individual bits in a file is not trivial. Use the two classes `BitReader` and `BitWriter` of the Python library `bitIO.py` to do that. It is not necessary to understand their internal workings, you just need to know how to use the classes.[4].

   Create a program in Python that uses the `readbit()` method to read the individual bits in an input file one by one, and prints these bits as `0` and `1` characters in the process. Use a table of ASCII code to check the output of a small `.txt` file with a few characters.

3. In addition to being able to read single bits, `BitReader` can also read four bytes (32 bits) in a row and return them as an `int` (i.e., a value between -2^{31} and $2^{31}-1$, not just between 0 and 255).

   First, create a `.txt` file with 16 characters i. Second, create a Python program that reads these 16 bytes as four `int`'s and prints each of them via four calls to `readint32bits()` method.

   With a little knowledge from the web about the two's complement representation for integers and the program from the previous task, you can check if the output matches.

B(II). Solve at home before tutorial in week 13

1. Consider sorting n integers with many duplicates. Describe an algorithm that sorts n integers in time $O(n\log(\log n))$, if there are only $O(\log n)$ distinct numbers. Assume that two numbers can be compared in constant time $O(1)$.

[3] You should *not* start by reading the entire file and save all its bytes before you start counting. The reason is, files can be large (potentially larger than your computer's RAM) and should not be stored in programs if, as here, it can be easily avoided.

[4] They work by working one byte in advance and inserting/reading single bits into it using shift operations and bitwise AND and OR

Hint: Use an appropriate data structure to collect the duplicates. Note that the difference between two numbers can easily be $\omega(\log n)$. In other words, even if there are only $O(\log n)$ different numbers, there can be a big difference in the size of the numbers. Explain how the sorted sequence can be read from the data structure and printed in time $O(n)$.

2. (∗) Cormen et al., 4. Edition: problem 15.1 (page 446) [Cormen et al., 3. Edition: problem 16.1 (page 446)].

   Find a way to pay an integer amount n with the fewest number of coins possible. The design of a country's coin set requires consideration to make it simple to give money back in a cash trade (i.e. for a natural greedy algorithm to work).

   (a) Consider the US coin set with quarters, dimes, nickels and pennies (25, 10, 5 and 1 cent). Describe a greedy algorithm that finds the fewest number of coins that add up to a given amount of n cents. Prove that the algorithm is correct.

   Hint: Show that there is always an optimal solution consisting of your first greedy choice and an optimal solution to the remainder problem. It may help to look at an optimal solution and line up its coins sorted descending by size. The argument is similar to that for the next question (which can be solved first).

   (b) Show that if for a coin set with sizes $m_1 = 1, m_2, \ldots, m_k$, m_i goes up in m_{i+1} for all i, then the greedy algorithm from question (a). (This question is a generalization of the one from the book).

   Hint: Same as for the last question.

   (c) Describe a coin set and an amount n where the greedy algorithm does not work (i.e., does not find the minimum number of coins). The smallest coin in your set must have a value one, ensuring that any amount n can be obtained.

   Hint: A coin set with three coins and an amount n below ten is enough.

   (d) Describe an algorithm that always finds the smallest number of coins to obtain an amount n. Let k be the number of different coin types (where the smallest has value one), then it should run in time $O(kn)$.

   Hint: Use dynamic programming to construct a table $R[i]$ of size $1 \times n$, where $R[i]$ contains the number of coins in an optimal *solution for amount i. This is similar to the gold chain problem—an optimal solution for amount i must contain either a coin of type 1, or one of type 2, or one of type 3, and so on.*