Exam, DAT038/TDA417 (with solutions)

Data structures and algorithms

Sample exam, 2019-12-16

There will be **6 basic questions**, and **3 advanced questions**, and **two points per question**. So, the highest possible mark is 18 in total, of which 12 on the basic questions and 6 on the advanced questions. Here is what you need to do to get each grade:

- To pass the course, you must get 8 out of 12 on the basic questions.
- To get a four, you must get 9 out of 12 on the basic questions, and also get 2 out of 6 on the advanced questions.
- To get a five, you must get 10 out of 12 on the basic questions, and also get 4 out of 6 on the advanced questions.

Grade	Total points	Basic points	Advanced
3	≥ 8	≥ 8	—
4	≥ 11	≥ 9	≥ 2
5	≥ 14	≥ 10	≥ 4

Good luck!

Write your anonymous code (*not* your name) here: _____

Basic question 1 (complexity)

What are the order-of-growths for these functions? Assume that each step in the code takes a constant amount of time to run.

The following algorithm takes as input an array, and returns the input array with all the duplicate elements removed. For example, if the input array is {1, 3, 3, 2, 4, 2}, the algorithm returns {1, 3, 2, 4}.

```
S = new empty set
A = new empty dynamic array
for every element x in input array {
  if not S.contains(x) then {
    S.add(x)
    A.add(x)
  }
}
return A
```

1A) What is the order of growth of this algorithm, if the set S is implemented using a red-black tree?

Order of growth: **n log n**

(explanation, not needed to pass the question: S has size at most n, so the body of the loop takes log n time, and it runs n times)

1B) What is the order of growth of this algorithm, if the set S is implemented using a hash table with linear probing? You may assume that the hash function used is of high quality.

Order of growth: **n**

(explanation: this time the loop body takes expected constant time)

Write your anonymous code (*not* your name) here: _____

Basic question 2 (binary search)

Your job is to complete the following program for binary search. It takes an array of integers and a key to search for. It should return the index in the array where the key is found, or `-1` if the key is not found.

The binary search routine is almost finished, except for the blank spaces marked by "_____", which you should fill in. The idea of the routine is that, at each point, if the key is present in the array at all then it can be found in the interval `array[low]...array[high]`.

```
int binarySearch(int[] array, int key) {
    int low = 0;
    int high = array.length - 1;

    while (low <= high) {
        int mid = (low+high)/2;
        if (key < array[mid]) {
            hi = mid-1;
        } else if (key > array[mid]) {
            lo = mid+1;
        } else {
            return mid;
        }
    }
    return -1;
}
```

Write your anonymous code (*not* your name) here: _____

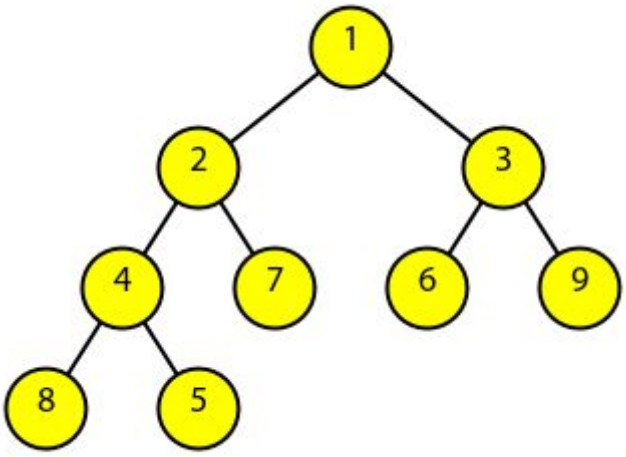
Basic question 3 (priority queues)

Insert the following elements in sequence into an empty binary min heap: 6, 8, 4, 7, 2, 3, 9, 1, 5. Draw both the tree and array representations of the heap.

3A) Array representation

1	2	3	4	5	6	7	8	9
1	2	3	4	7	6	9	8	5

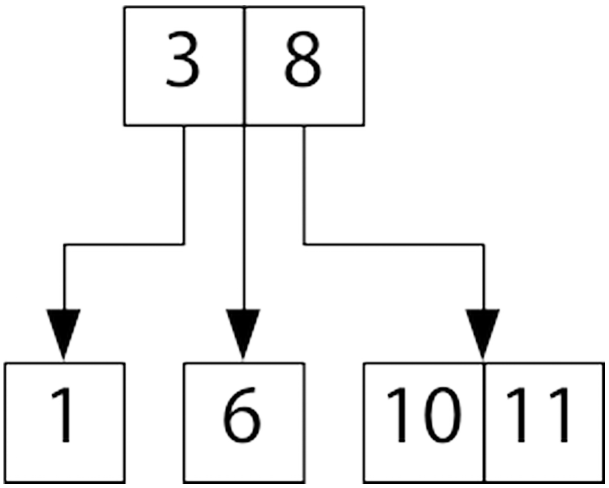
3B) Tree representation



Write your anonymous code (*not* your name) here: _____

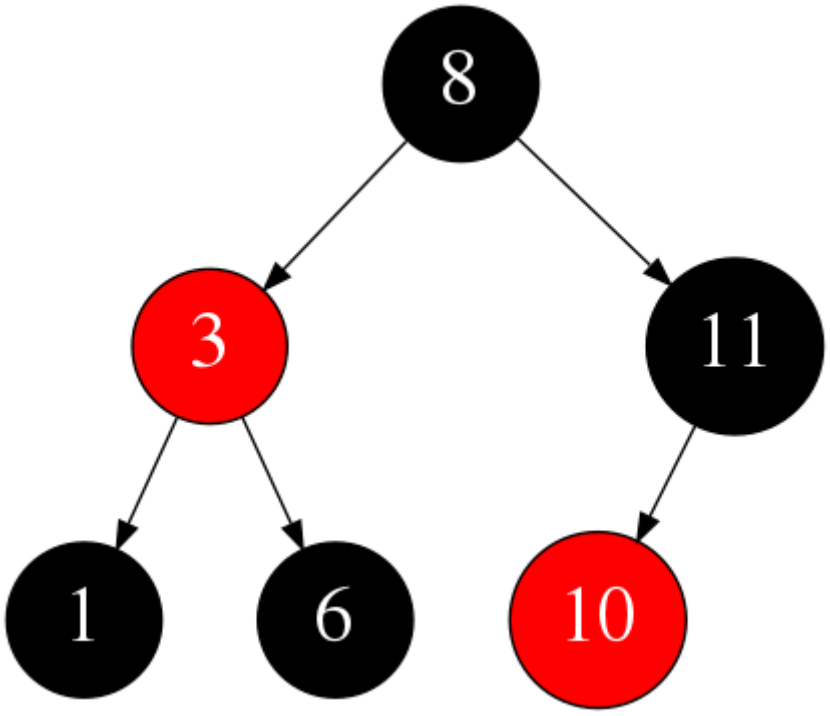
Basic question 4 (red-black BSTs)

Look at the 2-3 tree below.



4A) Draw the tree as a red-black BST. Draw black nodes as circles and red nodes as squares.

Answer:



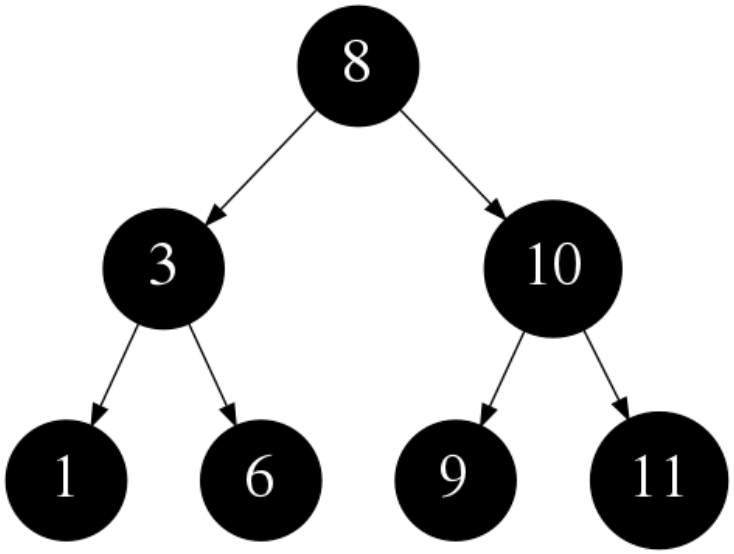
Write your anonymous code (*not* your name) here: _____

4B) Insert 9 into the tree using the red-black insertion algorithm.

Write down the final tree.

Note: if you are a student of TDA416/DAT037, in the real exam there would be an alternative question about AVL trees for you.

Answer:



(I solved this by inserting into the 2-3 tree and then converting to a red-black tree. That works too!)

Write your anonymous code (*not* your name) here: _____

Basic question 5 (hash tables)

Suppose you have the following hash table, implementing using *linear probing*. The hash function we are using is the identity function, $h(x) = x \pmod 9$.

0	1	2	3	4	5	6	7	8
9	18		12	3	14	4	21	

In which order could the elements have been added to the hash table? There are several correct orders, and you should give all of them. Assume that the hash table has never been resized, and no elements have been deleted yet.

A) `9, 14, 4, 18, 12, 3, 21`

B) `12, 3, 14, 18, 4, 9, 21`

C) `12, 14, 3, 9, 4, 18, 21`

D) `9, 12, 14, 3, 4, 21, 18`

E) `12, 9, 18, 3, 14, 21, 4`

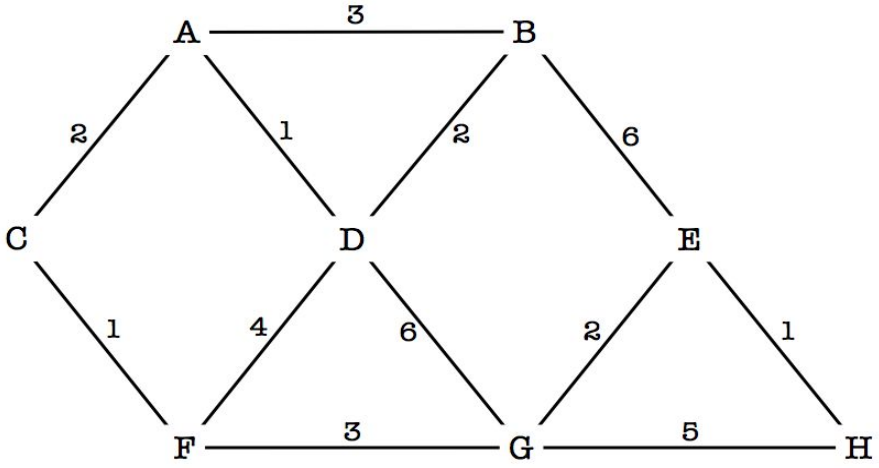
Answer: C, D.

(In A, 4 would end up at index 4. In B, 18 would end up at index 0. In E, 21 would end up at index 6.)

Write your anonymous code (*not* your name) here: _____

Basic question 6 (graphs)

You are given the following weighted graph:



6A) Do a breadth-first search of the graph, starting from node A.

Which order do you visit the nodes in? There are several possible orders that a breadth-first search could choose; choose **two** of these orders and write both of them down below.

Answer (for example):

Node	A	C	D	B	F	G	E	H
Node	A	B	D	C	E	G	F	H

6B) Perform Dijkstra's algorithm starting from node A.

In which order does the algorithm visit the nodes, and what is the computed distance to each of them? There are several possible orders to visit the nodes in - you may choose any of them.

Answer (for example):

Node	A	D	C	F	B	G	E	H
Distance	0	1	2	3	3	6	8	9

Write your anonymous code (*not* your name) here: _____

Advanced question 7

Consider the following function:

```
public void f(int[] a, int m) {
  int[] b = new int[m];
  for (int i = 0; i < a.length; i++) {
    int j = a[i];
    b[j]++;
  }
  int i = 0;
  for (int j = 0; j < m; j++) {
    for (int k = 0; k < b[j]; k++) {
      a[i] = j;
      i++;
    }
  }
}
```

7A) What is printed out when executing the following code?

```
int[] a = new int[10] {5,3,7,1,0,3,5,4,3,6};
f(a, 8);
// The following line prints the contents of the array
System.out.println(Array.toString(a));
```

Answer: 0, 1, 3, 3, 3, 4, 5, 5, 6, 7

7B) What does the function f do? Explain as well the meaning of m.

Answer: It sorts the array (using counting sort). m is the number of buckets - the input numbers must be all in the range 0 to m-1.

Write your anonymous code (*not* your name) here: _____

Advanced question 8

Answer on a separate sheet of paper.

A bidirectional map is a symbol table which supports bidirectional lookup: given a key, you can find the corresponding value, and given a value, you can find the corresponding key.

In a bidirectional map there is always a one-to-one relationship between keys and values. In other words, each key has exactly one value, and each value is found under exactly one key.

A bidirectional map supports the usual symbol table operations:

- new(): create a new, empty bidirectional map
- put(k, v): add the mapping k → v to the map;
  any existing mapping with key k *or value v* is removed.
- get(k): if the map contains a mapping k → v, return v
- delete(k): if the map contains a mapping k → v, delete it

It also provides an extra *reverse* operation:

- rlookup(v): if the map contains a mapping k → v, return k

The following example shows what the various operations do.

Operation	**Result**
new()	Map is {}
put(1,2)	Map is {1 → 2}
put(3,4)	Map is {1 → 2, 3 → 4}
lookup(1)	Returns 2
put(4,2)	Map is {3 → 4, 4 → 2}. Notice that the mapping 1 → 2 is deleted.
rlookup(2)	Returns 4
delete(4)	Map is {3 → 4}

We can implement a bidirectional map as *two* red-black trees:

- `forward` is a symbol table mapping keys to values.
  In the example above, it contains 3 → 4 and 4 → 2.
- `back` is a symbol table mapping values to keys.
  In the example above, it contains 4 → 3 and 2 → 4.

Write your anonymous code (*not* your name) here: _____

The invariant is that the two maps always contain the same data. More precisely, `forward` contains the mapping k → v, if and only if `back` contains the mapping v → k.

We can then implement new, lookup and rlookup as follows:

- `new`: set `forward` and `back` to be empty red-black trees
- `lookup(k)`: look up k in `forward` using the BST lookup algorithm
- `rlookup(v)`: look up v in `back` using the BST lookup algorithm

Your task is to implement the remaining operations, `put` and `delete`, so that they take logarithmic time.

A) Work out how to implement `delete`, and write down your solution.

```
delete(k) {
  if forward contains k {
    v = forward.get(k)
    forward.delete(k)
    back.delete(v)
  }
}
```

B) Work out how to implement `put`, and write down your solution.

This is rather tricky! The problem is that when adding k → v, there may already be a mapping with key k, *or* a mapping with value v. Both mappings have to be removed! To get this right, I would recommend testing your code by hand, on the example above, and checking that both maps have the right contents after each step.

```
put(k, v) {
  if forward contains k {
    v1 = forward.get(k)
    forward.delete(k)
    backward.delete(v1)
  }
  if back contains v {
    k1 = back.get(v)
    forward.delete(k1)
    backward.delete(v)
  }
  forward.put(k, v)
  back.put(v,k)
}
```

Write your anonymous code (*not* your name) here: _____

For both parts, you should write down your answer either as pseudocode or Java code. Your solution must take logarithmic time. **You can use the standard red-black tree operations (such as insertion and deletion) without explaining how they are implemented.**

Write your anonymous code (*not* your name) here: _____

Advanced question 9

Answer on a separate sheet of paper.

In this question, you will think about *double-ended priority queues*, a data structure that represents a collection of (e.g.) integers and supports the following operations:

- `void add(int x)`: add a new item to the priority queue
- `int deleteMin()`: remove and return the minimum item
- `int deleteMax()`: remove and return the maximum item

9A)

Here is an idea for implementing a double-ended priority queue, which unfortunately *does not work*:

- Maintain a min-heap `h1` and a max-heap `h2`, both initially empty.
- To implement `add(x)`, add `x` to both `h1` and `h2`.
- To implement `deleteMin()`, call `h1.deleteMin()`.
- To implement `deleteMax()`, call `h2.deleteMax()`.

Your job is to work out why this design does not work. Once you have done so, write down a sequence of operations (calls to `add`, `deleteMin` and `deleteMax`) which, if we run them starting from an empty priority queue, give the wrong answer. You should write down your answer in the following format (note that this example gives the right answer, however):

Operation	**Right answer**	**Actual answer**
add(3)	-	-
add(5)	-	-
deleteMin()	3	3
deleteMax()	5	5

Answer: (you can test this by hand to see what goes wrong)

Operation	**Right answer**	**Actual answer**
add(3)	-	-
deleteMin()	3	3
add(2)	-	-

Write your anonymous code (*not* your name) here: _____

deleteMax() 2 3

9B)

In a binary tree, the *level* of a node is defined as follows: the root of the tree has level 0, its children are at level 1, its grandchildren are at level 2, and so on. In other words, the level of a node is the distance from the node to the root.

A *min-max heap* is a complete binary tree satisfying the following invariant:

- If a node is at an *even* level, its value is *less than or equal* to all values in the node's subtree.
- If a node is at an *odd* level, its value is *greater than or equal* to all values in the node's subtree.

Describe how to efficiently find the minimum and maximum elements in a min-max heap. You may assume that the heap is non-empty. Write your answer either as pseudocode or as Java code.

Be careful: there are a number of special cases! Make sure that your algorithm also works for heaps that contain only one item or only two items.

Hint: try drawing a min-max heap first, to get a feel for what the invariant means.

Answer: the minimum element is at the root, just as in a min heap.

You can find the maximum element as follows:

- **If the root has two children, return the maximum of those children**
- **If the root has one child, return that child**
- **If the root has no child, return the root**

(The observation is that the root's children are at level 1, so each of them is the maximum of its respective subtree. But note the special cases when the root doesn't have two children!)