Datastrukturer/Data structures DAT037
Version with answers

Time	Wednesday 24th April 2019, 14:00–18:00
Place	SB-multisalen
Course responsible	Nick Smallbone, tel. 0707 183062 (will visit at approximately 15:00 and 17:00)
Examiner	Nils Anders Danielsson

The exam consists of six questions, and you may answer in **English** or **Swedish**.

For each question you can get a **3**, **4** or **5**. Your grade on the exam is determined as follows:

- To get a 3 on the exam, you must get a 3 on 3 questions.
- To get a 4 on the exam, you must get a 4 on 4 questions.
- To get a 5 on the exam, you must get a 5 on 5 questions.

To get a 5 on a question, you must answer it correctly, including any parts marked **for a 4** or **for a 5**. Skipping a part marked **for a 4** or **for a 5** will prevent you from getting that grade on that question. Minor mistakes *might* be accepted; a larger mistake will lead to a reduced grade or a U on that question. Excessively complicated answers may be rejected.

Allowed aids	One A4 piece of paper of hand-written notes, which you may take with you after the exam. You may write on both sides.
Exam review	The exams will be available for review in the student office on floor 4 of the EDIT building. If you want to discuss the grading of your exam, contact Nick within 3 weeks of getting your result. In that case, do not take the exam from the student office.
Note	Begin the answer to each question on a new page. Write your anonymous code (*not* your name) on every page. Write legibly – we need to be able to read your answer!

Good luck!

1. Here is an algorithm to test if two sets `X` and `Y` are disjoint (contain no elements in common):

```
boolean isDisjoint(X, Y) {
  for every element x in X {
    if x is a member of Y then {
      return false
    }
  }
  return true
}
```

Suppose that X is implemented as an unsorted array. What is the worst-case time complexity of this algorithm if Y is implemented as:

a) an unsorted array?
Assume that X and Y both have length n, and give your answer in terms of n.

Answer: $O(n^2)$ [by using linear search].

b) a hash table (with a good-quality hash function)?
Assume that X and Y both have length n, and give your answer in terms of n.

Answer: O(n) [by using hash table lookup].

c) **For a 4 or 5:** a sorted array?
Assume that X has length m and Y has length n, and give your answer in terms of m and n.

Answer: O(m log n) [by using binary search].

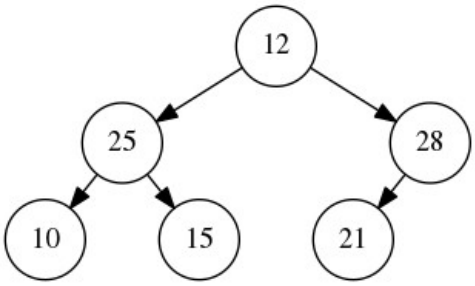
Assume that the line "`if x is a member of Y then`" is using the most efficient algorithm for the given data structure in each case.

Give your answer in big-O (or big-Θ) notation, simplifying it as far as possible; unnecessarily complicated answers may be rejected. You

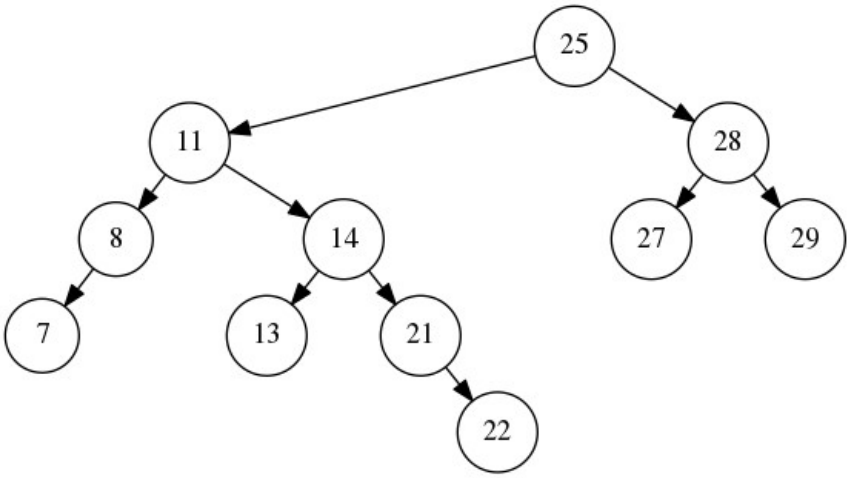
do not need to give a proof. You may assume that comparisons and hash calculations take O(1) time.

2. Have a look at the following three binary trees.

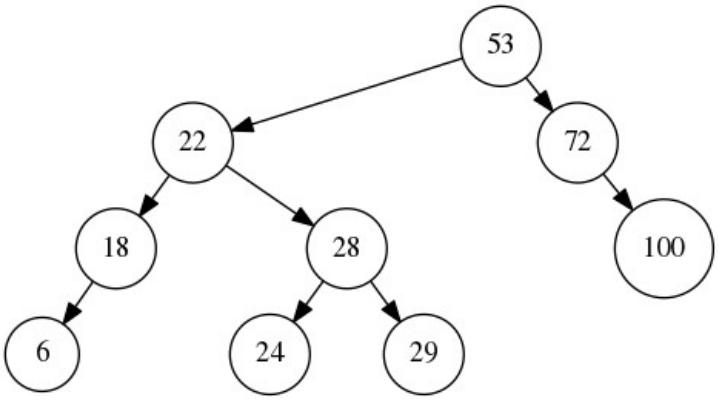
Tree A:



Tree B:



Tree C:

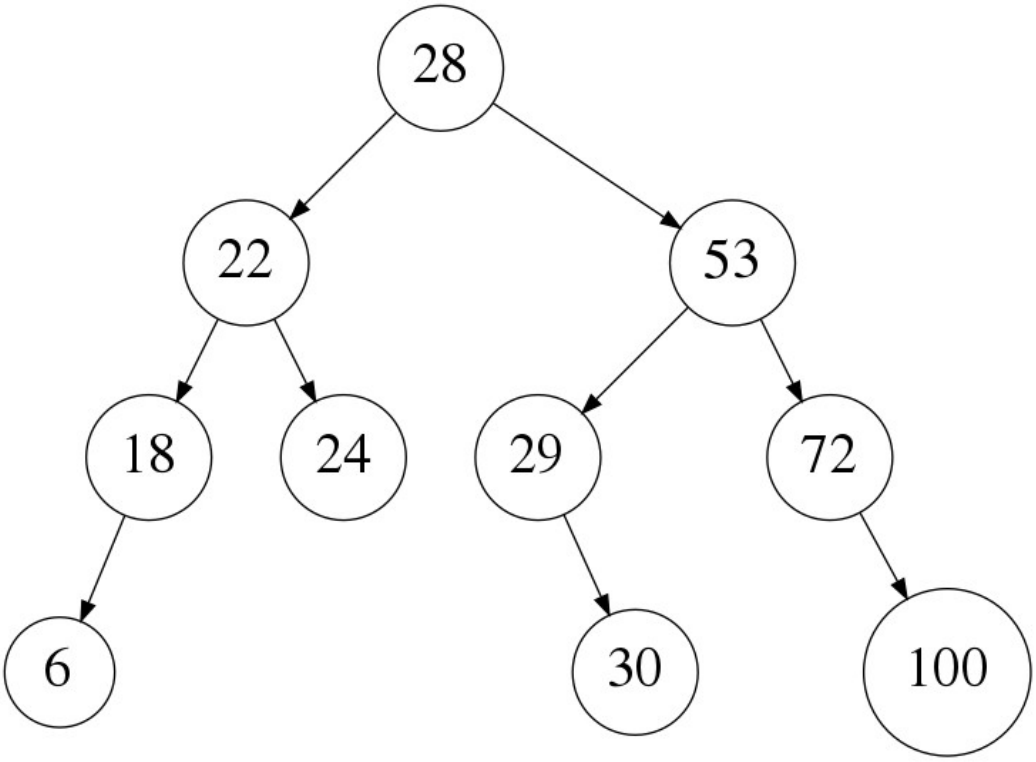


a) One of these trees is an AVL tree. Which one?

Answer: C. In A, 25 is in the wrong position relative to 12. In B, the height of the root's two children differs by 2.

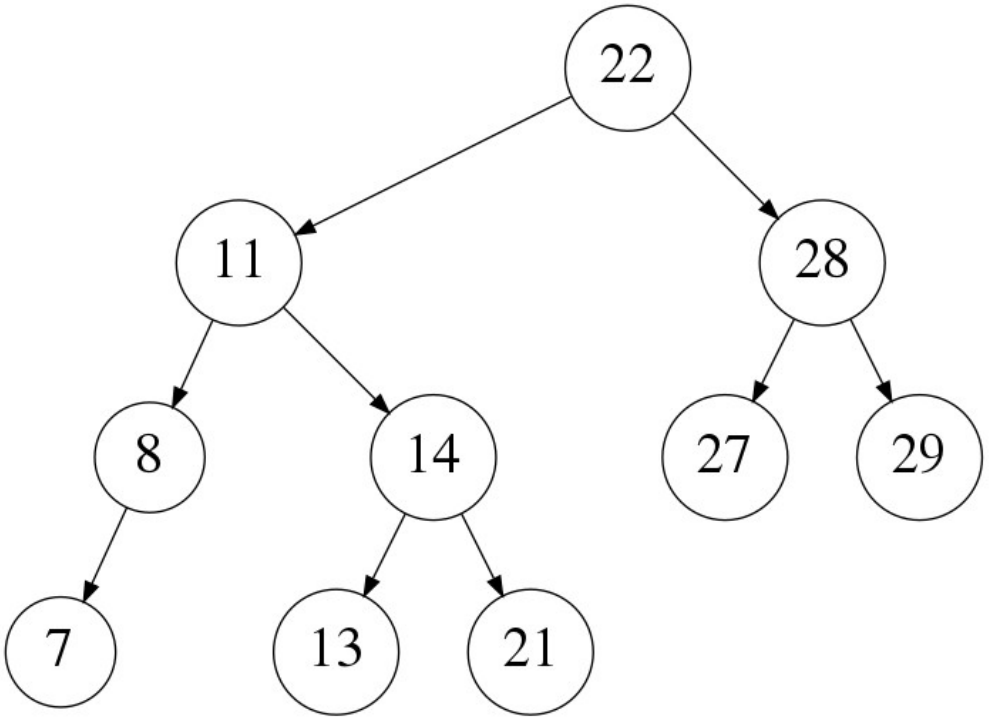
b) Insert 30 into the AVL tree using the AVL insertion algorithm. Write down the final tree.

Answer (insertion produces a left-right tree):



c) **For a 4 or 5**: delete 25 from **Tree B** using the BST deletion algorithm. Write down the final tree.

Answer (move 22 to replace 25):



3. Consider the following class for dynamic arrays in Java:

```
class DynamicArray<A> {
  private int size;
  private A[] data;

  ...
}
```

The class contains two fields, seen above: `size` is the number of elements in the dynamic array, and `data` is the contents of the dynamic array, stored in indices `data[0], ..., data[size-1]`. As usual with dynamic arrays, there may be unused space at the end of the array, in which case `size < data.length`.

For this question, only **detailed code** (not necessarily Java) will be accepted, **not pseudocode**. You may not use other methods or procedures apart from ones you have implemented yourself. You may assume that the array is not empty and that any index given as a parameter is a valid index in the array. You do not need to resize the array if it becomes small.

a) Write a method `void deleteLast()` which removes the last element from the dynamic array. It should take **O(1)** time.

Answer:

```
// optionally also: data[size-1] = null;
size--;
```

b) Write a method `void delete(int index)` which removes the element at position `index` from the dynamic array, preserving the order of elements in the array. For example, calling `delete(2)` on a dynamic array containing `{2,5,4,1,3}` should result in `{2,5,1,3}`. It should take **O(n)** time.

Only completely correct solutions will be accepted. You may want to test your code on the example above to make sure it works. Be extra careful that all loop bounds and array indexes are correct.

Answer (the idea is to move all the following elements back one

place):

```
for (int i = index; i < size-1; i++)
    data[i] = data[i+1];
size--;
```

c) **For a 4 or 5:** If we do not need to preserve the order of elements in the array, we can implement deletion more efficiently. Write a method `void deleteUnordered(int index)` which removes the element at position `index` from the dynamic array. It may alter the order of elements in the array. It should take **O(1)** time.

Answer (the idea is to move the last element of the array into the deleted element's place):

```
data[index] = data[size-1];
size--;
```

4. A *bidirectional map* is a map 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.

   It supports the usual map operations:

   - `empty()`: create a new, empty map
   - `add(k, v)`: add the mapping $k \to v$ to the map;
     any existing mapping with key k or value v is removed.
   - `lookup(k)`: if the map contains a mapping $k \to v$, return v
   - `delete(k)`: if the map contains a mapping $k \to v$, delete it

   plus the following *reverse lookup* operation:

   - `rlookup(v)`: if the map contains a mapping $k \to v$, return k

   The following example shows what the various operations do.

Operation	**Result**
`empty()`	Map is {}
`add(1,2)`	Map is {1 → 2}
`add(3,4)`	Map is {1 → 2, 3 → 4}
`lookup(1)`	Returns 2
`insert(4,2)`	Map is {3 → 4, 4 → 2}. Notice that the mapping 1 → 2 is replaced by 4 → 2.
`rlookup(2)`	Returns 4
`delete(4)`	Map is {3 → 4}

We can implement a bidirectional map using *two* maps, each implemented as an AVL tree:

- `forward` is a map from keys to values.
  In the example above, it contains 3 → 4 and 4 → 2.

- `back` is a map from values to keys.
  In the example above, it contains 4 → 3 and 2 → 4.

The invariant is that the two maps always contain the same data: `forward` contains the mapping $k \to v$, if and only if `back` contains the mapping $v \to k$.

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

- `empty()`: set `forward` and `back` to be empty AVL 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, `add` and `delete`, with O(log n) complexity. Make sure to maintain the data structure invariant; solutions that break the invariant will be rejected.

Be careful: the algorithm is a little more complicated than it seems. It's a good idea to test your solution on the previous page's example.

You should express your answer as either code or **pseudocode**, i.e. a mixture of code and textual description. Pseudocode means that you do not need to write (e.g.) valid Java code, but your answer must be detailed enough that a competent programmer could easily implement it.

You may freely use standard data structures and algorithms from the course in your answer, including insertion/lookup/deletion in a map, without explaining how they are implemented.

Answer:

For delete(k), the important thing is to delete from both forward and back:

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

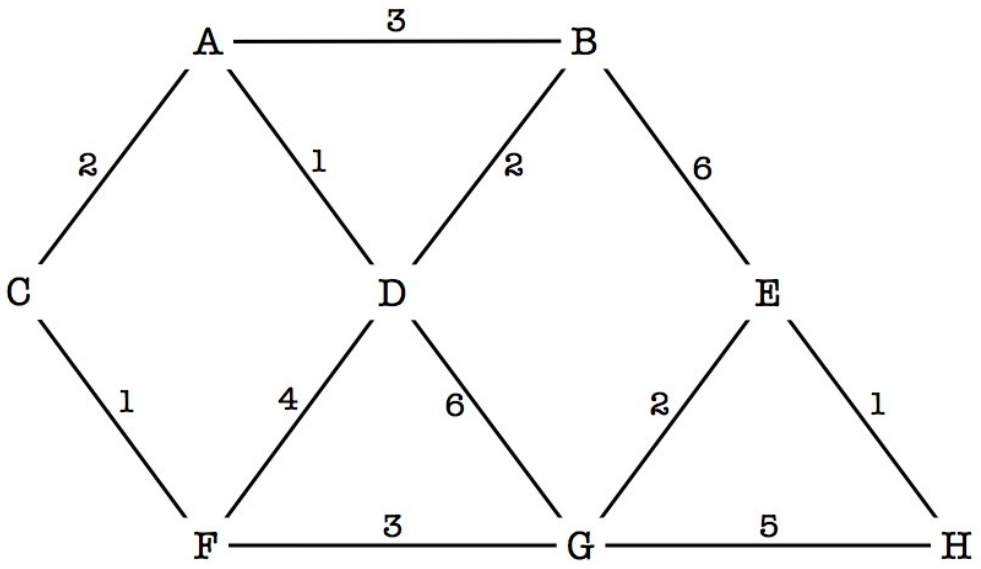
insert(k, v) is a bit trickier, because the map may already contain a mapping with key k *and* a mapping with value v, and for both of these both the forward- and back-entries have to be removed. So potentially four entries have to be removed before the insertion can happen:

```
insert(k, v) {
  if (forward.contains(k)) {
    v2 = forward.lookup(k);
    forward.delete(k);
    back.delete(v2);
  }

  if (back.contains(v)) {
    k2 = back.lookup(v);
    forward.delete(k2);
    back.delete(v);
  }

  forward.insert(k, v);
  back.insert(v, k);
}
```

5. You are given the following weighted graph:



Suppose we use Dijkstra's algorithm starting from node C. In which order does the algorithm visit the nodes, and what is the computed distance to each of them?

Answer:

C	0
F	1
A	2
D	3
G	4
B	5
E	6
H	7

(Answers using relaxation of weights as shown in the textbook were also of course accepted)

6. Your task is to design a data structure for storing a *set of integers*. The data structure should support the following operations:

   - `empty()`: create a new, empty set. If you prefer, you can model this as a constructor: `new IntSet()`.

   - `add(x)`: add the integer x to the set. **You may assume that the integer is not already present in the set.**

   - `member(x)`: return true if x is present in the set.

   - `deleteOldest()`: among all integers in the set, find the one that was added first, and delete it (you do not need to return the integer).

The operations must have the following time complexities:

- **For a 3:** O(1) `empty()`, O(log n) for `add` and `deleteOldest`, O(n) for `member`

- **For a 4 or 5:** as above except that `member` should be O(log n)

For an example, please see the next page.

You should write down what design you have chosen (e.g., what fields or variables the class would have or what data structure you are using), as well as how each operation is implemented.

You should express your answer as either code or **pseudocode**, i.e. a mixture of code and textual description. Pseudocode means that you do not need to write (e.g.) valid Java code, but your answer must be detailed enough that a competent programmer could easily implement it.

You may freely use standard data structures and algorithms from the course in your answer without explaining how they are implemented.

You may assume that comparisons take O(1) time, and that a high-quality hash function taking O(1) time is available.

Answer:

For a 3, one possible answer is to use a queue (implemented e.g. as a linked list). add(x) enqueues x. deleteOldest() dequeues an element. member(x) iterates through the elements of the queue and returns true if one of them matches.

For the higher grade, you can use both a queue and a set (stored using e.g. a hash table). The idea is that, when you add or remove elements from the queue, you also add or remove them from the set. Then member(x) just needs to look in the set. In more detail:

```
Queue<Integer> q;
Set<Integer> s;

IntSet() {
  q = new LinkedList<Integer>();
  s = new HashSet<Integer>();
}

void add(int x) {
  q.enqueue(x);
  s.add(x);
}

void deleteOldest() {
  int x = q.dequeue();
  s.delete(x);
}

boolean member(int x) {
  return s.member(x);
}
```

The following example shows how the data structure should work:

Operation	**Result**
`empty()`	Set is {}
`add(1)`	Set is {1}
`add(2)`	Set is {1, 2}
`member(1)`	Returns true
`deleteOldest()`	Set is {2} since 1 was added least recently
`add(3)`	Set is {2,3}
`add(1)`	Set is {1,2,3}
`deleteOldest()`	Set is {1,3} since 2 was added least recently