Dugga/Partial Exam
Data Structures (DAT037/DAT036)

2014-12-01

- Lecturer: Ramona Enache
- For an assignment to be accepted as *solved*, it needs to basically be a correct solution. Small errors could be accepted. However, please note that the Dugga will be graded more strictly than the final exam.
- Don't leave solutions for several assignments on the same sheet.
- Write clearly; unreadable = wrong!
- Solutions which are unnecessarily complicated or poorly motivated won't be accepted.
- Use the uniform cost model for analyzing complexity
- Unless the assignment specifies otherwise, you don't need to explain the data structures and algorithms from the course, but to motivate their use.
- Since certain data structures have different implementations, which yield to different complexities for certain operations, you should specify briefly which implementation of the data structure you use if you need to compute the complexity of an algorithm which uses that data structure.

1. 1. What is the complexity of the following code ?

   ```
   for (int j = 1; j <= n; j = j*2)
      a.add(j);
   int v[] = a.toArray() ;
   insert_sort(v);
   ```

   where

   - `n` is a positive integer
   - `a` is a dynamic array
   - `v` is a normal array
   - `a.toArray` is linear in the number of elements from a
   - comparisons take O(1) time
   - `insert_sort` is insertion sort

   Compute the time complexity in the O notation and justify it.

2. Write a program that verifies if a singly-linked list of n integers is a palindrome in O(n) time complexity.

   For this purpose you can only use any of the following data structures from without having to implement it from scratch: *linked lists (singly or doubly linked), stacks, queues, heaps, graphs.* No *arrays/dynamic arrays*, except if they are used for implementing one of the previous data structures.

   Motivate the final time complexity. What is your space complexity ?

   Can you come up with a couple of tests for your solution ? Test how it works on them.

   Write Java/Haskell/pseudocode.

   - For Java/pseudocode, use the following representation for linked lists

     ```
     class Node
             {int info;
              Node next};
     ```

     The list is represented by its `head` and the last element of the list has `next = NULL`.

   - For Haskell, use Haskell lists. However, you can only use the basic list constructs (`:` and `[]`) and no other standard/predefined list function (for example `++`). You need to implement from scratch any function for lists that you wish to use in your solution.

3. Write a program that tests that if a binary tree given as input is a BST (binary search tree). The binary tree contains generic information. Your program should be a function taking a binary tree as input and returning

`true/false`. Additionally you can implement and use helper functions. Also you can use any data structure from the course without having to implement them from scratch.

What is the time and space complexity of your solution ? Justify!

Can you come up with a couple of tests for you solution ? Test how it works on them. Write Java/Haskell/pseudocode.

The binary tree representation is the one from the lectures, just adapted for arbitrary type of information.

Dugga/Partial Exam Solutions
Data Structures (DAT037/DAT036)

2014-12-08

1. The complexity of the piece of code is $O(log\, n)$. Below are the complexities of the individual elements:

- Loop: $O(log\, n)$. Loop goes from 1 to n, with increments doubling in size for each iteration, ie. $log\, n$ times.
  - add: $O(1)$. This is repeated $log\, n$ times, so the complexity of the whole loop is $O(log\, n)$.
- toArray: $O(log\, n)$. toArray is linear in the number of elements. In this case there are $log\, n$ elements in the dynamic array that is constructed in the loop.
- insert_sort: $O(log\, n)$. insert_sort is quadratic in general case, but linear **for presorted input**. In this case, it is given a sorted array of $log\, n$ elements.

We have three consecutive $log\, n$ terms, thus the complexity of the whole code is $O(log\, n)$.

Note that it is incorrect to say that insertion sort is $O(n^2)$, even if that is the general case. In this case we see clearly that the input is presorted, and we must use that information in deciding what is the complexity of that particular piece of code.

2. We are using the following representation for linked lists:

```
List {Node head;}
Node {int info; Node next;}
```

The algorithm below traverses the list twice, first time pushing each item into a stack, second time popping from the stack and comparing the two items. It returns false immediately if any two elements are not same. If it has succesfully compared the whole list to the elements in the stack, it returns true.

Algorithm 1 isPalindrome

Input: List a
Output: Boolean
  $b \leftarrow$ empty stack
  $nd \leftarrow a$.head
  while nd is not null **do**
    b.push(nd)
    $nd \leftarrow nd$.next
  end while
  $nd \leftarrow a$.head
  while nd is not null **do**
    prev $\leftarrow b$.pop()
    if nd is not *prev* **then**
      return false
    end if
    nd = nd.next
  end while
  return true

The complexity of this code is $O(n)$. We have the following elements:

- Create stack: $O(1)$.
- Get head of the list for first traversal: $O(1)$.
- First while loop: $O(n)$. Traverses the list and performs two actions which are $O(1)$: push and accessing the *next* node of the current node.
- Get head of the list for second traversal: $O(1)$.
- Second while loop: $O(n)$. Traverses the list and performs three $O(1)$ actions: pop, comparison and accessing the *next* node.

The algorithm goes through the list twice, both in $O(n)$ time, and two consecutive $O(n)$ operations is in total just $O(n)$.

3. We use the following representation for binary search trees:

```
Tree {Node root;}
Node {E info; Node left; Node right;}
```

E must be a type that has an ordering. In the pseudocode below we are using $\geq$ and $\leq$; in Java, we would compare with `compareTo` or using a comparator defined for that type. In Haskell, we would define `Ord` and `Eq` instances for the type, and we would be able to use the operators >, <, ==.

The solution below traverses the tree and keeps track of the minimum and maximum values, updating them as the algorithm descends to the subtrees. We define first a recursive helper function that takes a node and two values of type E:

Algorithm 2 isBSTUtil

Input: Node *node*, E *min*, E *max*
Output: Boolean
  if *node* is leaf **then**
    return true
  end if

  if *node*.info $\leq$ *min* **or** *node*.info $\geq$ *max* **then**
    return false
  end if

  $leftIsBST \leftarrow$ isBSTUtil(*node*.left, *min*, *node*.info)
  $rightIsBST \leftarrow$ isBSTUtil(*node*.left, *node*.info, *max*)
  return $leftIsBST$ **and** $rightIsBST$

The final algorithm takes the helper function and starts from the root of the tree, *min* and *max* initialised as the minimum and maximum value of the data type that is stored in the tree.

Algorithm 3 isBST

Input: Tree *tree*
Output: Boolean
  return isBSTUtil(*tree*.root, E.MIN_VALUE, E.MAX_VALUE)

The time complexity of this algorithm is $O(n)$. It traverses the tree once and does only operations which take constant time: accessing the fields of the nodes, comparisons and recursive function calls.

Space complexity is between $O(log\, n)$ and $O(n)$, depending how balanced the tree is. There is no additional data used; instead, the space complexity comes from the recursive step. The function call stack accumulates the function calls until the algorithm reaches the base case, where the tree is a leaf. In the case of a balanced BST, the algorithm needs $log\, n$ steps to reach to a leaf. In the worst case scenario, where the tree is a chain of length n, the algorithm needs n steps to reach a leaf.