
Efficient Algorithms and Data Structures I

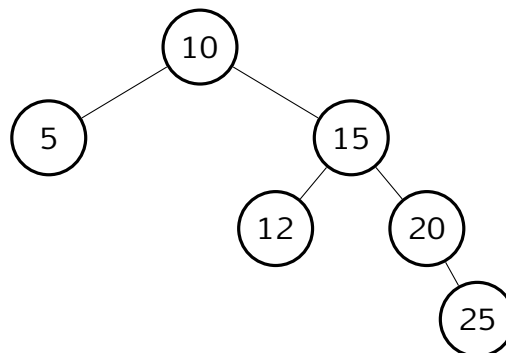
*Deadline: November 25, 10:15 am in the **Efficient Algorithms** mailbox.*

Homework 1 (4 Points)

- (a) If we insert a node in a red-black tree and then immediately delete the same node, is the resulting tree always identical to the original tree? Explain your answer.
- (b) For each power of two n , describe a red-black tree on n keys that realizes the smallest possible ratio of red internal nodes to black internal nodes.
- (c) Prove or disprove the following statement: The shortest simple path from a node x in a red-black tree to a descendant leaf has length at most half that of the longest simple path from node x to a descendant leaf.

Homework 2 (4 Points)

The teddy bear Alan is asked to decorate the Christmas tree for a major shopping center in the city center of Munich. This year, he chooses to use the glass baubles to form a nice splay tree. Initially, he obtains the following tree:



However, during the month of December, several changes must be made to the tree, as some baubles must be inspected more closely and others are added to the tree. Show what Alan's tree looks like after each change. The tree must remain a splay tree. Always carry out each operation on the result of the previous operation.

1. Search 12
2. Search 25
3. Insert 22

Homework 3 (6 Points)

We say that a family of binary tree are *balanced*, if every tree of n nodes in the family has height $\mathcal{O}(\log n)$.

For each property below, either show that the family of binary trees satisfying that property is balanced or provide a counterexample to show that it is not balanced.

Note: Remember that being balanced is an *asymptotic* property, so your counterexamples must specify an infinite set of trees in the family, not just one tree.

- Every node of the tree is either a leaf or it has two children.
- The size of each subtree can be written as $2^k - 1$, where k is an integer (k is, of course, not the same for each subtree).
- For each node in the tree, the size of the smaller child subtree of this node is at least $1/2$ times the size of the larger child subtree. For this statement, we say that a subtree can have size 0, i.e., if a node has only one child, the smaller subtree (on the side with no child) has size 0.

Homework 4 (6 Points)

Suppose that, for the purpose of the analysis, we assign some weight $w(x)$ to any node x of a given splay tree. Let $s(x) = \sum_{y \in T_x} w(y)$ be the sum of all node weights in the subtree of x and the rank $r(x) = \log(s(x))$.

Define the *improved potential function* of the splay tree T as $\Phi_w(T) = \sum_{x \in T} r(x)$.

In this exercise, we show that a clever choice of weights w helps prove interesting properties of splay trees. The simple choice $w(x) = 1$ gives the analysis seen in the lecture.

- Let the *amortized cost* of operation i with cost c_i be $\hat{c}_i = c_i + \Phi_w(T_i) - \Phi_w(T_{i-1})$. Show that any sequence of m operations on the tree T_0 has a cost of

$$\sum_i \hat{c}_i - \Phi_w(T_m) + \Phi_w(T_0) ,$$

where T_i is the tree after operation i .

- Let $W = \sum_T w(x)$. Show that the rank $r(x)$ decreases at most by $\log(W) - \log w(x)$ during any sequence of search operations.
- Suppose that item x is accessed $q(x) > 0$ times during some sequence of m search operations on tree T , i.e. $m = \sum_x q(x)$. Choose weights $w(x)$ to show that the total cost of the sequence is at most

$$\mathcal{O} \left(m + \sum_{x \in T} q(x) \log(m/q(x)) \right) .$$

Hint: Adapt the amortized analysis from the lecture using your own choice of weights. The relevant slide is Slide 186.

Tutorial Exercise 1

- (a) Describe how to implement a queue using two stacks and $O(1)$ additional memory, so that the amortized time for any ENQUEUE or DEQUEUE operation is $O(1)$. The only access you have to the stacks is through the standard subroutines PUSH and POP.
- (b) A quack is a data structure combining properties of both stacks and queues. It can be viewed as a list of elements written left to right such that 3 operations are possible:
 - (i) QPUSH: add a new item to the left end of the list
 - (ii) QPOP: remove the item on the left end of the list
 - (iii) QPULL: remove the item on the right end of the list

Implement a quack using 3 stacks and $O(1)$ additional memory, so that the amortized time for any QPUSH, QPOP, or QPULL operation is $O(1)$. Again, you are only allowed to access the stacks through the standard functions PUSH and POP.

Tutorial Exercise 2

Programmer Paul is given a magical closed-source library called `Priorizer`. A `Priorizer` supports the operations `INSERT` and `EXTRACT-MAX`. `INSERT` allows to insert an integer. `EXTRACT-MAX` extracts the largest integer from the `Priorizer` and returns `NULL`, if the `Priorizer` is empty. Both operations run in constant time.

- (a) Argue briefly how to use a `Priorizer` to obtain a data structure that supports `INSERT` and `EXTRACT-MIN` in constant time. `EXTRACT-MIN` should extract the smallest integer from the data structure.
- (b) Paul's supervisor Sunny needs a data structure that supports the operations `INSERT`, `EXTRACT-MAX` and `EXTRACT-MIN`. She requests that Paul makes all three operations run in amortized constant time.

Help Paul by designing the requested data structure. Your implementation should use two `Priorizers`. Additionally, you may use a linked list and a constant amount of additional storage (e.g., to store an integer).

Describe precisely how the operations are implemented. Then analyze their running times using a suitable potential function. You do not need to prove that your implementation works correctly.

Hint: Use one `Priorizer` for “large” elements, the other one for “small” elements. What can you do if one of the `Priorizers` runs empty?

Proudest [work]? It's hard to choose. I like the self-adjusting search tree data structure that Danny Sleator and I developed. That's a nice one.

- R. E. Tarjan