1A Lent Algorithms

Supervision 3: Data structures (queues, trees)

Hayk Saribekyan

February 21, 2020

When you are asked to describe an algorithm, also analyse its time and space complexity. Mention how you would implement the solution.

In this supervision we will concentrate on data structures. We will cover queues, stacks and search trees. There are also problems from the previous chapter on algorithm design.

This example sheet is longer than I expected but all the problems are quite important to understand. Also many are just bookwork (e.g. warm up). I do not expect you to solve and write down precise solutions to everything, so think about the things you struggle with. You can go back to them during revision or when you have time later.

I would rather see a failed attempt or ideas handed in, than a beautifully written correct solution of a problem that you found easy. I do not mind if you skip a problem and write "I know how to do it." So focus on the concepts that you have most trouble with rather than writing down solutions to problems that are easy for you.

I would like to thank Leran Cai for adding some solutions and Patrick Ferris for allowing me to use some of his.

1 Refresher (Algorithm Design)

This is material from the previous supervision.

1. You are given positions of n mice and n holes on the x axis. A mouse moves from position x to $x \pm 1$ in 1 minute. Assign each hole to exactly one mouse in a way that minimises the time it takes the mice to reach their hole. Prove the correctness of your algorithm.

   Solution Suppose the positions of holes and mice are given by two arrays h_i and m_i respectively. Then the obvious solution is to sort both h and m and assign mouse i to hole i. The time it takes for them to reach their hole is

   $$\max_i |m_i - h_i|$$

   Let $m_1 < m_2$ and $h_1 < h_2$. By careful case analysis we can see that

   $$\max\{|m_1 - h_2|, |m_2 - h_1|\} \geq \max\{|m_1 - h_1|, |m_2 - h_2|\}$$

   Therefore, it is never worse to assign the hole with the least coordinate to the mouse with the least coordinate. Therefore, the sorting algorithm is a correct approach.

Note: The solution stays the same if we want to minimise the total distance the mice walk rather than the maximum that any single one walks.

2. You are given a $2 \times n$ board. In how many ways is it possible to cover the board with n dominoes, if each domino should cover exactly two adjacent cells of the board?

   Explain your solution and give an implementation in any language.

   Solution Let d_n be the number of ways we can tile the $2 \times n$ board with dominoes. Consider the last column. We can place one domino there vertically, in which case there are d_{n-1} ways to tile the rest of the board.

   Alternatively, we can place two dominoes horizontally that occupy the last two columns. Then we can tile the rest of the board in d_{n-2} ways.

   Therefore, $d_n = d_{n-1} + d_{n-2}$. Clearly, $d_1 = 1$ and $d_2 = 2$. Thus, we get a Fibonacci sequence that is slightly shifted.

3. Given a matrix of integers of size $n \times m$. Each row and column is sorted in ascending order, from left to right and from top to bottom, correspondingly. Find the location of a given number x.

   Bonus: Prove that no algorithm can solve it faster than in $\Omega(n)$ time if $n = m$.

   Solution https://www.geeksforgeeks.org/search-in-row-wise-and-column-wise-sorted-matrix/

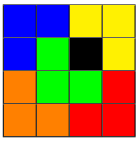
   Notice that the "optimal" solution above takes $O(n+m)$ time. Another solution is to binary search either in each row or in each column. This would result in a runtime $O(\min\{n,m\} \log \max\{n,m\})$, which can be better than the linear solution above, if say $n = 2$ and m is large.

   This shows that the O-notation as defined in lectures somewhat breaks when there are more than 1 variables involved[1].

   Bonus: Consider the square matrix a, where $a_{i,j} = -\infty$ if $i + j < n + 1$, $a_{i,j} = +\infty$ if $i + j > n + 1$. The secondary diagonal has arbitrary values. This matrix satisfies the problem conditions, however the values on the secondary diagonal have no relations among each other. Therefore $\Omega(n)$ time is necessary to find an element on this diagonal.

4. Given an $n \times n$ board, where $n = 2^k$. The cell at (i, j) is removed. Describe an algorithm that will cover the remaining cells of the board with L-shaped tetris pieces. Below is a tiling for $n = 4$. The black cell is removed.

[1] http://people.cs.ksu.edu/~rhowell/asymptotic.pdf



Hint: $n = 2^k$ *divides by two very nicely many times!* correct

Solution https://www.geeksforgeeks.org/divide-and-conquer-set-6-tiling-problem/

2 Warm-up (Data Structures)

1. (Robert Harle) Discuss the (dis)advantages of linked-list and array based implementations of queues.

   Solution Linked list provides following two advantages over arrays

   - Dynamic size.
   - Ease of insertion/deletion.

   Linked lists have following drawbacks:

   - Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
   - Extra memory space for a pointer is required with each element of the list.
   - Arrays have better cache locality that can make a pretty big difference in performance.

   For more details: https://www.geeksforgeeks.org/linked-list-vs-array/

2. (Robert Harle) Consider implementing a stack using an array. When the array is full and a push() is requested, there are two common strategies to growing the array: increase the array by a constant number of elements or double the size of the array. Analyse both strategies to find the amortized costs associated with a push() operation.

   Solution The aggregate method: Here we use the aggregate method for simplicity. There are another two methods: the accounting method and the potential method.

   Incremental strategy analysis: Suppose in the end we have n elements and each time we increase the array size by a constant c and we update the array $k = n/c$ times. The total cost is

   $$T(n) = n + c + 2c + ... + kc = O(n + k^2) = O(n^2)$$

   The amortized time of a push operation is $T(n)/n = O(n)$.

   Double strategy analysis: Suppose in the end we have n elements and we update the array $k = \log n$ times. Therefore the total cost is

   $$T(n) = n + 1 + 2 + ... + 2^{k-1} = O(n)$$

   The amortized time of a push operation is $T(n)/n = O(1)$.

3. Using the sequence $\{10, 85, 15, 70, 20, 60, 30, 50, 65, 80\}$ draw:

   (i) the BST tree

   (ii) the 2-3-4 tree

   (iii) the red-black tree

Solution The constructions of the three trees are below, but first let's discuss RB-trees again.

A note on RB-trees. When working on RB-trees, it easiest to think of them as 2-3-4 trees, and think of the colour of a node as the colour of the edge to its parent. Then, think of the red colour edges as a "glue", i.e., the red nodes are glued to their parents. The nodes that are glued together represent one node in the corresponding 2-3-4 tree. Or, conversely, to go from a 2-3-4 node u to a group of RB-nodes, one does the following:

- If u has 1 key, then it stays as is.
- If u has 2 keys, we create two nodes in the RB-tree, say x and y, one for each key. Then arbitrarily, x becomes the parent of y in the RB-tree, and we make the edge between them red. This indicates that they are glued together.
- If u has 3 keys x, y and z, we create three nodes in the RB-tree. The node representing y in the RB-tree becomes the parent of the nodes x and z, with the two edges being red.

With all of this in mind, let's consider the 5 main invariants of the RB-trees, which at first seem arbitrary but when thinking of 2-3-4 trees, they are not.

- Every node is either red or black: ok.
- The root is black: it does not have a parent to be glued to, so this makes sense.
- All leaves are black and do not have a key-value pairs: this is mainly for consistency.
- If a node is red, its children are black: this follows from the fact that a 2-3-4 node has can have only 3 values, so you are not allowed to glue more than 3 things together.
- All paths from a node to the leaves in its subtree have the same number of black nodes: this follows from the fact that 2-3-4 trees are balanced.

There is a well written article on this in Wikipedia:

https://en.wikipedia.org/wiki/Red%E2%80%93black_tree#Analogy_to_B-trees_of_order_4

Additionally, you can play with 2-3-4 trees here:

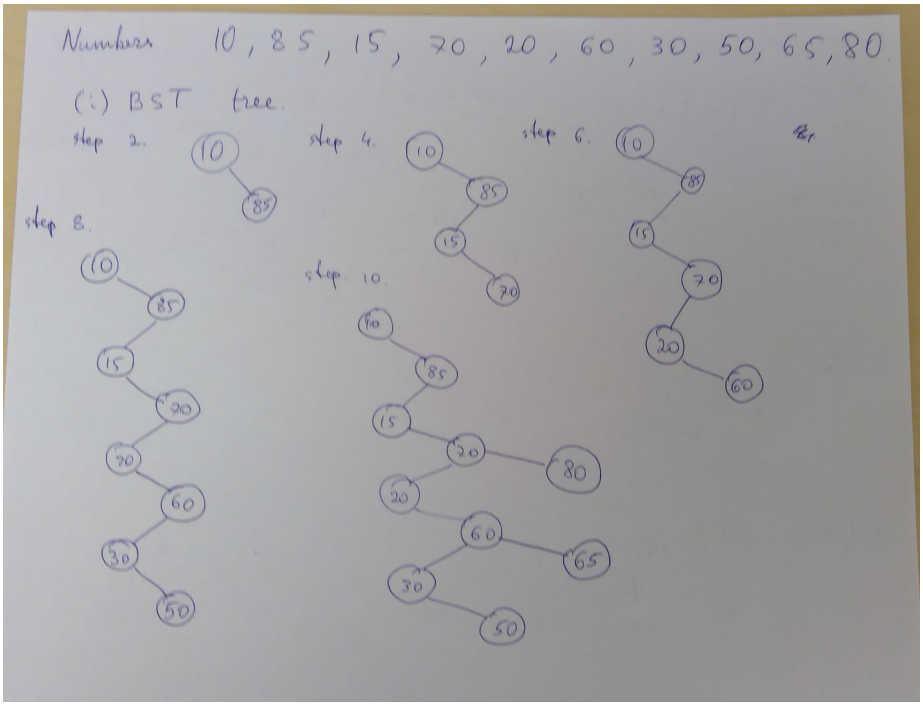
https://ysangkok.github.io/js-clrs-btree/btree.html

and see another example here:

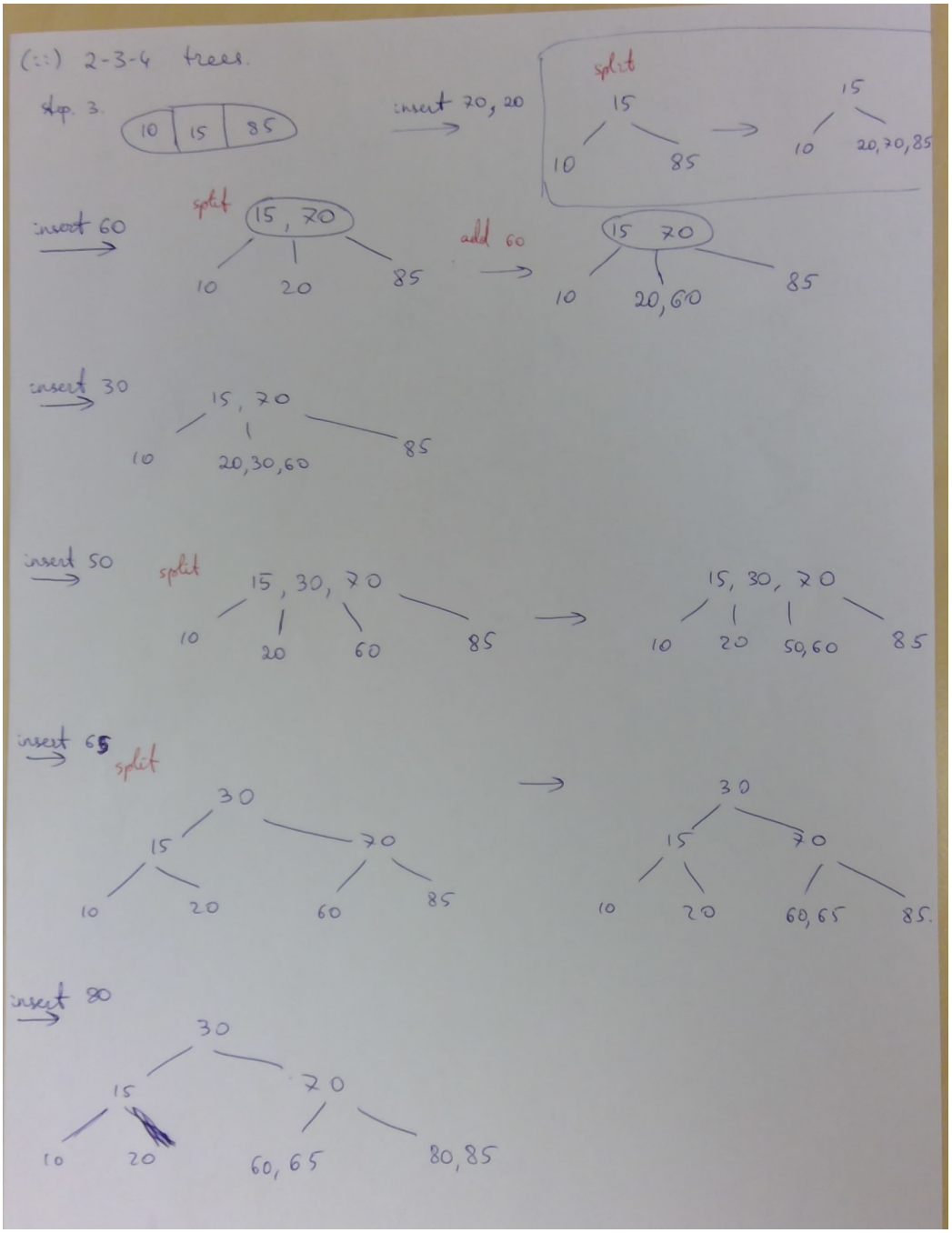
https://www.educative.io/page/5689413791121408/80001

During your supervision I may have said that "if you are inserting an element, and you encounter 4-nodes (i.e. nodes with 3 keys), you do not have to split them there, you can do it later." While this is still true, there is no point of not splitting the 4-nodes as you go down the tree, because you will have to do another pass over the path to do it anyway. Thanks to Viki for pointing this out.

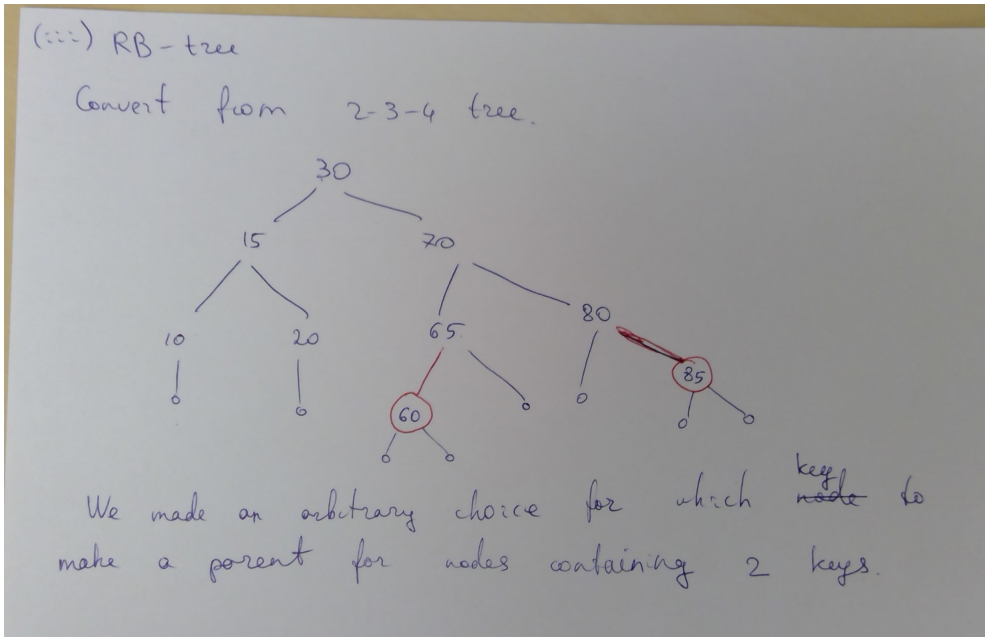
The solution is due to *Patrick Ferris*. Binary Search Tree using the first 10 numbers.



2-3-4 tree using some of the numbers.



and its corresponding RB-tree.



4. (Robert Harle) The main issue with BSTs is their tendency to become unbalanced. This can be a particular issue if the input keys have a lot of duplicates (e.g. insert $\{1, 1, 1, 1, 1, 1, 1\}$ gives a very unbalanced tree. Suggest a way in which duplicate entries in a BST could be addressed such as to produce a more balanced result.

   Solution We can allow every tree node to store count and balance the tree when having duplicate entries. However this method cannot make the entire tree balance.

3 Examples

1. (Robert Harle) Write a function (in Java/Python/Pseudocode/etc) that tests whether a string is a palindrome (the same backwards as forwards) using only the standard operations of a stack.

```
static boolean isPalindrome(String s) {
  Stack<Character> st = new Stack<Character>();

  for (int i = 0; i < s.length / 2; i++) {
      st.push(s[i]);
  }

  for (int i = (s.length + 1) / 2; i < s.length; i++) {
          if (st.pop() != s[i]) {
          return false;
```

```
            }
        }
        return true;
    }
```

2. (Robert Harle) A *table* is a set of key-value pairs (e.g. dictionary in Python, map in Java), where $set(k, v)$ means to map the key k to value v. Usually there is also a $get(k)$ function, which returns the corresponding value v.

   Write a pseudocode of the $set(k, v)$ function for the linked-list based table. Duplicate elements are not permitted i.e. $set(k, v_1)$ and $set(k, v_2)$ operations would result only in one element in the table with key k with value v_2.

   Solution: Below is a Python-pseudocode.

```
class Dictionary:
    # ... other methods ...
    def set(k, v): # assumes that the linked list is sorted
        prev = None
        curr = head
        while curr is not None:
            if curr.key < k:
                prev = curr
                curr = curr.next
            elif curr.key = k:
                curr.value = v
                return
            else:
                if prev is None: # curr is the head
                    head = new Node(k, v)
                    head->next = curr
                else:
                    prev->next = new Node(k, v)
                    prev->next->next = curr
                    return

        # if we are here, then we have to add (k, v) at the end
        if prev is None: # the set was empty
            head = new Node(k, v)
        else:
            prev->next = new Node(k, v) # add to th end
```

3. Given a string of parentheses, brackets and braces e.g. `(]{{]])(]`. Describe an algorithm that decides whether the given string is a valid. The string above is not whereas `[(){[]}]` is.

Solution Use stack. Scan the string. When an opening parentheses (of any type) is encountered push it on top of the stack. If a closing one is encountered, check that (1) the stack is not empty and (2) the top of the stack makes a correct pair with the current element. If one of the conditions fails then it's a wrong string. At the end if the stack is not empty, return false. Otherwise the string is correct.

Try with some strings and get convinced that this approach works. Essentially, whenever we match a character with the top of the stack, we eliminate a valid substring from the original string.

Below is a Python-pseudocode

```
def check_paren(s):
stack = Stack()
for ch in s:
        if ch in ['(', '[', '{']:
            stack.push(ch)
        elif stack.isEmpty():
            return false
        else:
            t = stack.pop()
            if t and ch do not make a pair:
                    return false

return stack.isEmpty()
```

4. Explain how to implement a queue using only stacks - no arrays, linked lists or any other data structures are allowed. Of course, you can use individual variables (e.g. to store sizes and indices).

   We want our queue to be efficient i.e. on average each operation that a queue supports should take $O(1)$ time. Notice that the naive implementation described below is not efficient.

   Naive implementation: To implement a queue `q`, keep a stack `s`. When `q.push(x)` is called, put the element on top of the stack `s` i.e. `s.push(x)`. On `q.front` operations, reverse `s` into another stack `s'` and return the top element of `s'`. Then, reverse the stack back into `s`.

   Solution Keep two stacks A and B to implement a queue Q. When $Q.push(x)$ is called, add it on top of A i.e. $A.push(x)$. When $Q.pop()$ is called check if B is empty. If yes, then reverse A into B. Then return $B.pop()$.

   The difference of this solution from the naive one is that once we reverse stack A into B to return the oldest element, we do not reverse it back into A. The amortised cost of each operation is $O(1)$, because each element is only moved around $O(1)$ times.

5. Implement a function `Node predecessor(Node node)` or its equivalent in Python that returns the predecessor of `node` in BST. You can assume that the class `Node` has whatever standard fields a BST would have.

   What would you change in the code to fund the successor?

Solution https://www.geeksforgeeks.org/inorder-predecessor-successor-given-key-bst/

Notice that the problem statement did not ask to find the node (unlike the above solution). So you only need to give part (3) from the solution.

6. Given a BST, write code that prints the elements in the tree in sorted order. Explain how it is different from heap-sort.

Solution By *Patrick Ferris*

```
def inorder(self, node):
    if(node is None):
        return
    self.inorder(node.leftChild)
    print(node.value)
    self.inorder(node.rightChild)
```

7. Initially you have an empty set S. You receive a stream of queries of two types:

   - `Insert(x)`: you should add x to S.
   - `Range(a, b)`: return the number of elements in S in the that are in the range $[a, b]$.

   You should process all the queries in $O(\log |S|)$ time.

Solution: For simplicity suppose we never receive any duplicate elements. Let the function $rank(x)$ return the rank of x in S i.e. the number of elements less than of equal x. Then the answer to the query $[a, b]$ is equal $rank(b) - rank(a - 1)$.

Suppose S is represented as a sorted array. Then we could computer $rank(x)$ for any x in $O(\log |S|)$ time using binary search. In that case, however, we could not support the `insert` operations in $O(\log |S|)$ time, because we would need to shift the array. This is where BSTs become handy.

We can keep the elements of S in a standard BST (forget about the balancing for now). Let each node p also store the number of elements $p.size$ in its subtree. It is easy to maintain $p.size$: when x is inserted in the tree, just increment the $p.size$ for all nodes that we traverse on our way to x's location.

Suppose now, we want to find the $rank(x)$. We implement a function `getRank(x, p)`, which returns the "rank" of x in subtree p. Below is the Python-psuedocode (which ignores some edge and base cases).

```
def getRank(x, p):
    if p.value > x:
        return getRank(x, p.left)
    if p.value == x:
        return p.left.size
    return p.left.size + 1 + getRank(x, p.right)
```

Now, we can answer the queries in $O(h)$ time, where h is the height of the tree. But h can be $\Omega(n)$. The balanced trees come to help. We can implement a RB-tree that also includes our $p.size$ field. However, since there are rotations in the tree we have to change the sizes of the subtrees that are being rotated. These are trivial to do once one draws a rotation.

Fenwick Trees or Binary Indexed Trees[2] are a beautiful and efficient data structure to solve problems like this one. If you have time and want to indulge in some beautiful algorithms, have a look.

4 Implementation (Bonus)

Implementations are critical to sharpen your understanding of data structures.I would suggest that you work on these during the break if you do not have time now. Once you have implemented a RB-tree and have fallen in all the traps yourself, you will never make those mistakes again.

In online judges, do not use standard packages provided by your language. Instead implement the data structures yourself.

1. Many of the covered data structures use a *tree-rotation* operation to keep the trees balanced. The rotations are easy to understand but surprisingly hard to implement. Splay trees[3] are possibly the simplest of such trees because they do not keep any other information per node (e.g. color). All you need to do is few rotations.

   Try implementing Splay trees (based on the description from Wikipedia) and make sure they work. Solve problems from online judges to make sure your implementations are correct.

   Petar Velickovic goes into some details of how to do this[4].

2. http://www.spoj.com/problems/RMQSQ/

[2] urlhttps://www.topcoder.com/community/data-science/data-science-tutorials/binary-indexed-trees/
[3] https://en.wikipedia.org/wiki/Splay_tree
[4] https://www.cl.cam.ac.uk/~pv273/supervisions/Algorithms/Algs-3.pdf