Assignment 4 – COMP 252

Alexandre St-Aubin and Jonathan Campana

February 27, 2024

1. Browsing the small elements in a red-black tree. Given is an ordinary red-black tree with a pointer to the smallest element (in addition to a pointer to the root). Cells have five components: left, right, and parent pointers, color of the node, and value of the key.

   (a) We are asked to search for an element u in the tree with key key [u] = x. Show that this can be done in time $O(1+\log(k))$ if x is the k-th smallest key value stored in the tree (but we only know x, not k).

   Solution:

   Given a red black tree, we have a balanced binary tree. This means that we can find any element in the RB tree in $\log_2(n)$ time. This also means that we have at most $\lceil \log_2(n) \rceil$ levels to our tree.

   Therefore, starting at the smallest node, we work our way up the RB tree by checking if the key of the node we are at is samller than the node we are looking for (x). We keep doing this until we find a node that has a key that is bigger than the key we are looking for, or we reach a node that does not have a parent nodes (the root).

   Denote the lowest common ancestor as the node that is ancestor to the k^{th} smallest node and the smallest node (our starting point). Given that our nodes are ordered in a red black tree, the possible lowest common ancestors (we will refer to them as LCAs from now on) would be found as the most left nodes of the tree at each level. Denote the possible LCAs

   $$LCA = \{v_1, v_2, \ldots, v_n\}$$

   By the red black tree, we have $\log(n)$ possible LCAs since the tree has n nodes. We, however, will not always reach the root, which is why our complexity is dependent on the k^{th} smallest element rather than n.

   By our algorithm, we move to the parent as long as the element we are looking for is bigger than the parent of the node we are currently at. Once we reach the point where the k^{th} smallest element is smaller than the parent node, or the parent node does not exist (the root), we conclude that the element we are searching for must be in the right subtree of the node we are at.

   The k^{th} smallest element in the binary tree can be any element in the right subtree including the current node for the sake of simplicity. Suppose that our LCA is v_i. Since the node that contains k is in the right subtree, and the subtree with v_i as the LCA would have at most $2k$ elements inside.

   This is easy to see since the lower bound for the number of leaves in the v_i would be $2k$ (in big Oh) if $key[v_i] = k$, then the left subtree would have $k-1$ elements since that is the number of elements smaller than the node we are looking for, and the right subtree would have at most $k-1$ nodes as well. Suppose that k was farther inside the right subtree, k would be bigger than in the previous case, we could take without loss of generality that the v_i subtree has $2k$ nodes (even though it would have less, since the left subtree wouldn't have k nodes in it) which would give us the same $O(\log(k))$, since the number of levels in this red black v_i subtree is $\lceil \log_2 k \rceil$.

The second while loop essentially binary searches the right subtree of our LCA v_i, which has at most k many nodes in it. Since it takes $O(\log k)$ to binary search a red black tree, this while loop also has $O(\log k)$.

Thus, employing the triangle inequality with this distance metric, we have that

$$d(\text{min pt, } k^{th}) \leq d(\text{min pt, greatest common ancestor}) + d(\text{greatest common ancestor, } k^{th}) + 1$$
$$\implies d(\text{min pt, } k^{th}) \leq O(\log k + 1) + O(\log k) = O(\log k + 1)$$

The constant in our big Oh notation stems from the possibility that $k = 1$, where $\log k = 0$, but there is still a comparison, so we add the 1.

(ii) Give the algorithm for part (i).

Algorithm 1: Find_kth_smallest red-black tree.

Input: Pointers to the smallest element and root of a red black tree, and value of x.
Output: Finds node of value x in the tree.

```
while (x ≥ key[node_pointer]) do
    if key[node_pointer] = x then
        return node_pointer;
    if (key[parent[node_pointer]] ≠ NULL) then
        node_pointer ← parent[node_pointer];
    else
        break;
node_pointer ← right[left[node_pointer]] // The parent was bigger than x, thus we
    go back in the subtree and set our pointer to the right child
while (x ≠ key[node_pointer]) ∧ (key[node_pointer] ≠ NULL) do
    if x > key[node_pointer] then
        node_pointer ← right[node_pointer];
    else
        node_pointer ← left[node_pointer];
if x = key[node_pointer] then
    return node_pointer;
else
    return NULL;
```

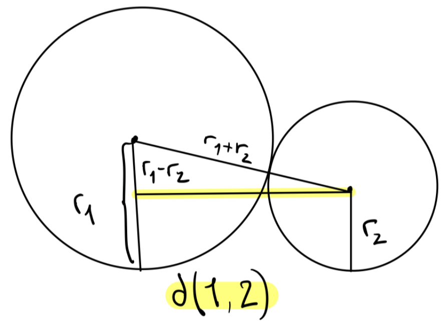

Figure 1: The distance calculated by `centre_dist()` in **Algorithm 2**.

2. GREEDY ALGORITHM. On a flat table, we have placed n disks of radii $r_1, ..., r_n$, numbered from left to right. We push them together without creating overlap, as in the figure below. Give an $O(n)$ time algorithm to compute the size of the smallest axis-aligned rectangle that can hold the disks.

   Solution:
   See the algorithm on the next page. We show that its complexity is $O(n)$. To begin, the outermost loop at line 8 iterates a total of n times. The maximum number of elements pushed onto the stack is also n (since at most one is pushed at each iteration), hence an equal number is removed, maintaining the overall $O(n)$ complexity. Finally, as we traverse through each circle once more at the end of the algorithm, the complexity remains $O(n)$.

Algorithm 2: Greedy circle packing

Input: An ordered list $\Omega := \{1,2,3,\ldots,n\}$ of n circles.
Output: The minimum width of a rectangle that can hold the disks.

```
// radius of the circle a
Function Radius(circle a):
    return radius of a;

// distance between the centres of a and b if they are pushed together.
Function centre_dist(circle a, circle b):
    return sqrt((Radius(a) + Radius(b))^2 - (Radius(a) - Radius(b))^2);

// largest subarray of Ω ending with k, decreasing in radii.
MAKENULL(possible_adjacent_stack);

// distance from the left side of the rectangle to circle at index
left_to_circle[] ← new array;

left_to_circle[0] ← 0;
for all i ∈ Ω do
    if i = 1 then
        left_to_circle[i − 1] ← Radius(i);
        PUSH(i, possible_adjacent_stack)
    else
        // initialize max_d, the maximum distance between left side of
           rectangle and circle i to Radius (i).  to account for the case
           where the circle would touch the side of the rectangle.
        max_d ← Radius(i);
        // POP each circle on the stack that's smaller than i.
        while Radius(PEEK(possible_adjacent_stack)) ≤ Radius(i) do
            max_d ← max{max_d, left_to_circle(POP(possible_adjacent_stack)) +
             centre_dist(i, possible_adjacent_stack(j))};
        // peek the first circle that's larger, but keep it on the stack
        max_d ← max{max_d, left_to_circle(PEEK(possible_adjacent_stack)) +
         centre_dist(i, possible_adjacent_stack(j))};
        // push i to stack, keeping the decreasing order
        PUSH(i, possible_adjacent_stack);
        // max_d is the distance from the left side of the rectangle to the
           center of circle i when it is pushed as much as possible without
           overlapping
        left_to_circle[i − 1] ← max_d;

// find the width
Width ← 0;
for i ← n − 1 to 0 do
    if left_to_circle[i] + Radius(i) > Width then
        Width ← left_to_circle[i] + Radius(i);

return Width;
```

3. AUGMENTED DATA STRUCTURES. Show how to maintain a dynamic set of numbers that supports the operation `min-gap`, which gives the magnitude of the difference of the two closest numbers in a set of numbers, A. For example, if $A = \{1, 5, 9, 15, 18, 22\}$, then `min-gap` (A) returns $18 - 15 = 3$, since 15 and 18 are the two closest numbers in A. Make the operations `insert, delete, search`, and `min-gap` as efficient as possible, and analyze the running times. Be concise!

4. THE NODE OF SMALLEST TENSION IN A TREE. Given is an unrooted free tree of size n. The nodes are labeled from 1 to n, and for each node, we have a linked list of its neighbors. When a node u is deleted, it breaks the tree up into a forest of disjoint trees, say $T_1, \ldots, T_k$. Let the sizes of these trees be denoted by $|T_1|, \ldots, |T_k|$. We define the tension of u as $\max_{1 \leq i \leq k} T_i$. The objective is to find a node u of smallest tension. Intuitively, it should be near the "center" of the tree. Write an algorithm that takes $O(n)$ worst-case time.

   Solution:
   See the algorithm on the next page. The complexity is easily seen to be $O(n)$, given that line 21 executes at most n times, and both loops at lines 28 and 34 iterate n times. Additionally, the post_order_traversal function is $O(n)$, as it traverses every node in the tree.

   The algorithm uses the following data structure,

```
struct NODE
{
linkedList neighbour;
int tension;
int size;
bool visited;
}
```

Algorithm 3: Smallest Tension

Input: A list $\Omega := \{1, 2, 3, ..., n\}$ of nodes, and for each node, a linked list of neighbours.
Output: The node of minimum tension.

```
Function post_order_traversal(root):
    root.visited ← True;
    if |root.neighbours| = 1 then
        root.size ← 1;
        root.tension ← n − 1;
        return;
    forall NODE child in root.neighbours do
        if child.visited = True then
            continue // means the node is a parent, so don't visit.
        post_order_traversal(child);
    // visit node
    forall NODE child in root.neighbours do
        root.size ← root.size + child.size; // one of the nodes is going to be the
            parent, but since it hasn't been visited yet, its field size will
            be 0, so this is an accurate calculation of the size.
    root.tension ← max{max_{i∈root.neighbours}{i.size}, n − root.size};
    return;
// -- Driver code --
// add fields to each node.
forall i ∈ Ω do
    temp ← i.neighbours;
    i ← new struct NODE;
    i.visited ← False;
    i.neighbours ← temp;
    i.tension ← 0;
    i.size ← 0;
// find a node that's not a leaf and initialize it as root.
forall i ∈ Ω do
    if |i.neighbours| > 1 then
        TopRoot ← i;
        exit;
// call the traversal at the top root
post_order_traversal(TopRoot);
temp ← ∞;
// find the node of minimum tension
for all i ∈ Ω do
    if i.tension < temp then
        temp ← i.tension;
return temp;
```