CIS 606 **Homework 6** **Fall 2019**

Name: **Reza Shisheie**

Login ID: **reshishe**

1. Exercise 14.1-3 (page 344): Write a nonrecursive version of OS-SELECT .

   The recursive version of OS-Select is shown in the following:

```
OS-SELECT(x, i)
r = x.left.size + 1
if i == r
    return x
elseif i < r
    return OS-SELECT(x.left, i)
else return OS-SELECT(x.right, i - r)
```

   On the non-recursive version, the function takes the root and the $i$the position and goes into a while loop.

   - In the while loop it calculates the current position of x, which is the left branch position plus 1 (self). x is the root for the first run.
   - If the position matches the $i$the location, then this is the target. Therefore, *while* loop breaks and position value is returned.
   - If r as current position is larger than i, then target value is located in the left branch. Therefore, x is updated to $x.left$ and while loop keeps executing the left branch for ith location. If $x.left$ is *null* then such position does not exist. Therefore return -1.
   - If r as current position is smaller than i, then target value is located in the right branch. Therefore, x is updated to $x.right$ and while loop keeps executing the right branch for the remaining position which is $i-r$. If $x.right$ is *null* then such position does not exist. Therefore return -1.

```
OS-SELECT-NONRECURSIVE(x, i)
while true
    r = x.left.size + 1
    if r = i
        return x
    if r < i
        if x.left = ∅
            return -1
        x = x.left
    elseif

        if x.right = ∅
            return -1
        x = x.right
        i = i - r
```

2. Exercise 14.3-6 (page 354) Show how to maintain a dynamic set Q of numbers that supports the operation MIN-GAP , which gives the magnitude of the difference of the two closest numbers in Q. For example, if $Q = \{1, 5, 9, 15, 18, 22\}$, then MIN-GAP($Q$) returns $18-15 = 3$, since 15 and 18 are the two closest numbers in Q. Make the operations INSERT, DELETE, SEARCH, and MIN-GAP as efficient as possible, and analyze their running times.

MIN-GAP:

The solution algorithm uses an RB-tree. A tuple is assigned to every element of set Q:

$$x_i = \{key, min_{sub}, max_{sub}, gap\}$$

where, key is the element value from Q, min_{sub} is the minimum value in the sub branch, max_{sub} is the maximum value in the sub branch, and gap is the minimum gap between the following values:

- $x.key - x.left.max_{sub}$: difference between current node and max node of left branch
- $x.right.min_{sub} - x.key$: difference between current node and min node of right branch
- $x.left.gap$: minimum gap in the left branch
- $x.right.gap$: minimum gap in the right branch

A visual look of the algorithm is shown in Figure.1.

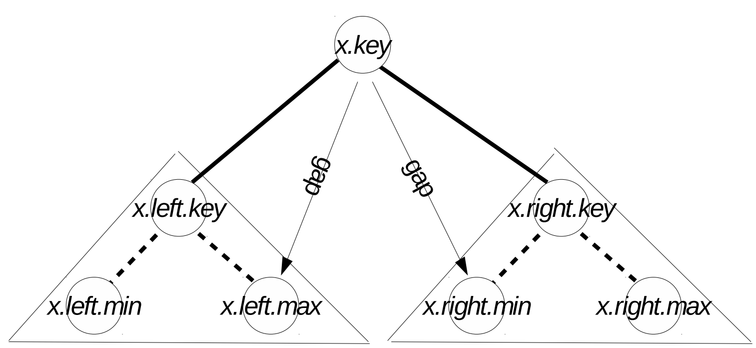

Figure 1: Algorithm branching

Before the MIN-GAP(x) is executed, each tupe is initialized to:

$$x_i = \begin{cases} key = x_{value} \\ min_{sub} = x_{value} \\ max_{sub} = x_{value} \\ gap = +\infty \end{cases}$$

It is worth noting, since the first three parameters are initialized to x_{value}:

- If the right branch does not have a subsequenct left branch ($x.right.left == \emptyset$), minimum value of the right branch ($x.right.min$) is the same as $x.right.key$
- Likewise if the left branch does not have a subsequenct right branch ($x.left.right == \emptyset$), maximum value of the left branch ($x.left.max$) is the same as $x.left.key$

The MIN-GAP(x) algorithm only takes the *root* and calucates minimum gap of each element and sub trees. Once its execution is over, *root.gap* returns the minimum gap in the whole RB-tree.

MIN-GAP(x)

// if there is no left or right child, this is the leaf→return the gap parameters of x which is $+\infty$
if $x.left = x.right = \emptyset$
    return
// if there is no right child, → return MIN-GAP($x.left$) on the left branch
elseif $x.right = \emptyset$ and $x.left \neq \emptyset$
    MIN-GAP($x.left$)
// if there is no left child, → return MIN-GAP($x.right$) on the right branch
elseif $x.left = \emptyset$ and $x.right \neq \emptyset$
    MIN-GAP($x.right$)
// if both children exist → return MIN-GAP on the both right and left branches
else
    MIN-GAP($x.left$)
    MIN-GAP($x.right$)
// Now min and max of current node has to be updated:
// if the min of the left branch¡min of currect node → update min of current node
if $x.left.min_{sub} < x.min_{sub}$
    $x.min_{sub} = x.left.min_{sub}$
// if the max of the right branch¿max of currect node → update max of current node
if $x.right.max_{sub} > x.max_{sub}$
    $x.max_{sub} = x.right.max_{sub}$
$x.gap = \min(x.key - x.left.max_{sub},$
    $x.right.min_{sub} - x.key,$
    $gap.left,$
    $gap.right)$

INSERTION:

For insertion two set of programs are recalled:

- INSERT(x, v): This function takes node v and recursively insert it into the RB-tree starting from *root*
  - If the $v.key > x.key$:
    - If there exist a $x.right$ subtree, then node v must be inserted into the right subtree $\rightarrow$ INSERT$(x.right, v)$
    - Otherwise, this is the leaf $\rightarrow$ Insert node v as new leaf to $x.right = v$ and return
  - If the $v.key \leq x.key$:
    - If there exist a $x.left$ subtree, then node v must be inserted into the left subtree $\rightarrow$ INSERT$(x.left, v)$
    - Otherwise, this is the leaf $\rightarrow$ Insert node v as new leaf to $x.left = v$ and return
- MIN-GAP-UPDATE-NODE(x): This function is a part of the MIN-GAP(x) function, which updates min_{sub}, max_{sub}, and gap of each node.

INSERT(x, v)

if $v.key > \emptyset$
    return
if $v.key > x.key$
    if $x.right \neq \emptyset$
        INSERT$(x.right, v)$
    elseif $x.right = \emptyset$
        $x.right = v$
        return
if $v.key \leq x.key$
    if $x.left \neq \emptyset$
        INSERT$(x.left, v)$
    elseif $x.left = \emptyset$
        $x.left = v$
        return
MIN-GAP-UPDATE-NODE(x)

MIN-GAP-UPDATE-NODE(x)

// if the min of the left branch¡min of currect node $\rightarrow$ update min of current node
if $x.left.min_{sub} < x.min_{sub}$
    $x.min_{sub} = x.left.min_{sub}$
// if the max of the right branch¿max of currect node $\rightarrow$ update max of current node
if $x.right.max_{sub} > x.max_{sub}$
    $x.max_{sub} = x.right.max_{sub}$
$x.gap = \min(x.key - x.left.max_{sub},$
        $x.right.min_{sub} - x.key,$
        $gap.left,$
        $gap.right)$

DELETION:

DELETION, like INSERTION, follows the same rule of RB-tree deletion. Since all data is stored locally or in children, only an update on parents - from place of change - is needed. Thus, time complexity of INSERTION and DELETION are the height of RB-tree $O(\log n)$.

SEARCH:

The goal is to find the two nodes, which have the minimum gap. The same rule as for INSERTION and DELETION applies to SEARCH as well. Staring from root of RB-tree, trace the result of MIN-GAP(x) value in the tree until both left and right braches yield larger $x.gap$ values. Calculating the minimum gap at this point, yields which two points yield the minimum gap. Therefore, time complexity of SEARCH is also $O(\log n)$