Report for Assignment 7

Deep Kiran Shroti
Roll Number- 12EC35013

March 9, 2016

Interval Tree

Assignment Statement Implement suitable routines for Insertion, Search and Deletion in RB trees. Write a scheduler and process creator which works as follows.

- In every iteration of an outer loop LL, if the total number of live processes is less than some NN, the process creator creates a process with execution time between [1,1000] and priority between [1,4].

- Live processes are stored in a Red Black tree keyed by their pending execution time.

- In every iteration of an outer loop LL, the scheduler

  – checks if there is any newly created process and inserts it to the tree,

  – searches the process with least pending execution time and gives it to the CPU, a process with priority i executes for i×50 seconds once scheduled.

  – Once the process finishes its quota of execution, the scheduler inserts it back to the tree if the process has not completed its entire execution.

- For insertion of elements in the RB tree whose key value is same as the key of some existing element, put both elements at the same node in a list structure.

Main function :
Write a main function from which the value of NN can be set and the system can be simulated for execution and completion of MM number of processes. The value of M is also set in main(). The output is expected to be a text file containing a table with column headings:

- process number, creation time, priority, time stamps when a process got scheduled, time stamps when a process got preempted from CPU, time stamp when a process completed execution.

1 Red Black Tree

A red-black tree is a binary tree that satisfies the following red-black properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

Red-black tree with n internal nodes has height at most *2log(n+1)*.

The subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes. This claim can be proved by induction on the height of x. If the height of x is 0, then x must be a leaf, and the subtree rooted at x indeed contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes. For the inductive step, consider a node x that has positive height and is an internal node with two children. Each child has a black-height of either $bh(x)$ or $bh(x) - 1$, depending on whether its color is red or black, respectively. Since the height of a child of x is less than the height of x itself, we can apply the inductive hypothesis to conclude that each child has at least $2^{bh(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x contains at least $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)-1}$ internal nodes, which proves the claim.

To complete the proof, let h be the height of the tree. According to no red-red violation, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently, the black-height of the root must be at least h/2; thus,

$$n \geq 2^{h/2} - 1$$

Moving the 1 to the left-hand side and taking logarithms on both sides yields

$$lg(n+1) \geq h/2 \text{ }, or h \leq 2lg(n+1)$$

As an immediate consequence of this lemma, we can implement the dynamic-set operations SEARCH , MINIMUM , MAXIMUM , SUCCESSOR , and PREDECESSOR in *O(lg n)* time on red-black trees, since each can run in $O(h)$ time on a binary search tree of height h and any red-black tree on n nodes is a binary search tree with height *O(lg n)*

2 Implementation Details

2.1 redBlackTreeInsert()

redBlackTreeInsert() of cases which is handled by *redBlackTreeInsertFixUp* The cases are as followed:

- **Case 1:** node's uncle y is red
- **Case 2:** node's uncle y is black and z is a right child
- **Case 3:** node's uncle y is black and z is a left child

2.2 redBlackTreeDelete()

redBlackTreeDelete() of cases which is handled by *redBlackTreeDeleteFixUp* The cases are as followed:

- **Case 1:** node's sibling w is red.
- **Case 2:** node's sibling w is black, and both of w's children are black.
- **Case 3:** node's sibling w is black, w's left child is red, and w's right child is black.
- **Case 4:** node's sibling w is black, and w's right child is red.

3 Analysis

Running time of *redBlackTreeInsert()*
Since the height of a red-black tree on n nodes is of *redBlackTreeInsert()* take *O(lg n)* time. In *redBlackTreeInsertFixUp()*, the while loop repeats only if case 1 occurs, and then the pointer moves two levels up the tree. The total number of times the while loop can be executed is therefore *O(lg n)*. Thus, *redBlackTreeInsert()* takes a total of *O(lg n)* time. Moreover, it never performs more than two rotations, since the while loop terminates if case 2 or case 3 is executed.

Running time of *redBlackTreeDelete()*
Since the height of a red-black tree of n nodes is *O(lg n)*, the total cost of the procedure without the call to redBlackTreeDeleteFixUp takes *O(lg n)* time. Within *redBlackTreeDeleteFixUp()* , each of cases 1, 3, and 4 lead to termination after performing a constant number of color changes and at most three rotations.
Case 2 is the only case in which the while loop can be repeated, and then the pointer x moves up the tree at most *O(lg n)* times, performing no rotations. Thus, the procedure *redBlackTreeDeleteFixUp()* takes *O(lg n)* time and performs at most three rotations, and the overall time for *redBlackTreeDelete()* is therefore also *O(lg n)*.