ADVANCED DATA STRUCTURES
COP5536: Fall 2019

PROGRAMMING PROJECT REPORT

Rising City(Min Heap and RBT)

SUBMITTED BY
Bharath Shankar
UFID – 9841-4098
Email ID: bharathshankar@ufl.edu

Rising City(RBT and Min Heap)

A new city is being constructed and a software is needed to keep track of all the buildings that have been built. A building record has 3 fields: Building number, Executed time and Total time. The Min Heap is used to store the buildings based on Executed Time and the RBT stores it based on the Building Number. The building with the least Executed time is chosen to be worked on, it is worked on for either 5 days or till it has completed construction. After completion, it is removed from the data structure. The buildings are printed using the RBT.

Programming Environment

The C++ programming language has been used to code this program. It has been run on Visual Studio Code for MAC.

Function Prototypes and Structure:

1. Red Black Tree

The **RBNode** is created using struct, it has the following members/variables.

- **int bNum**: The Number assigned to each Building.
- **int eTime**: The time spent on construction of a Building so far.
- **int tTime**: The total time needed for the construction of the building
- **bool color**: Stores the color as a Boolean value, red=1 and black =0.
- **struct RBNode *lft, *rght, *parent:** Pointers to the left, right and parent nodes of the current node.
- **RBNode(int b, int t):** The constructor to build the struct RBNode with bNum and tTime as parameters.

The **RBTree** is also maintained using a struct. The members present in the struct are:

- **struct RBNode *root:** It stores the root node of the RBT.
- **RBTree():** Constructor to build the RBTree struct. It initially sets the root value to NULL.

The various functions that work on the RBT implementation include the following.

- **bool withRdChild(struct RBNode *x):** Checks if the current node has a Red Child by looking at the color of the children.

- **bool isLChild(struct RBNode *x):** Checks if the current node is the left child of the parent.

- **RBNode* gParent(struct RBNode *x):** Returns the Grand parent of the given node.

- **RBNode* uncleOf(struct RBNode *x):**Returns the uncle of the given node.
- **RBNode* sblngOf(struct RBNode *x):** Returns the sibling of the given node.
- **void goDwn(struct RBNode *newNode, struct RBNode *x):** Puts a node in the correct position after going down the tree.
- **RBNode* rtrnRoot():** Returns the root of the RBT.
- **void RRrotate(struct RBNode *x):** Rotates the node when it’s in the RR case/configuration in a specific way by modifying the neighboring nodes.
- **void LLrotate(struct RBNode *x):** Rotates the node when it’s in the LL case/configuration in a specific way by modifying the neighboring nodes.
- **void swapClr(struct RBNode *x, struct RBNode *y):** Swaps the color of two nodes.
- **void swapVlue(struct RBNode *x, struct RBNode *y):** Swaps the value of two nodes.
- **void RRcorrect(RBNode *x):** It is used to correct a Red-Red Violation by peroforming various rotations and color changes.
- **RBNode* lftMost(RBNode *x):** Finds the leftmost node in the RBT by going down till there are no more left nodes.
- **RBNode* dltSubs(RBNode *x):** Finds the substitute node after a deletion takes place.
- **void BBcorrect(RBNode *x):** Fixes a condition where there are double black nodes by using the LLrotate, RRrotate and recursive calls to BBcorrect.
- **void dltNode(RBNode *x):** Deletes a particular node x from the RBT. It then finds a substitute node and makes color changes based on various conditions.
- **RBNode* search(int n):** Finds the node for the next insertion, deletion or Print based on bNum.
- **void RBInsrtNode(int n, int m):** Inserts a particular node with given bNum and tTime, it initially sets the eTime to 0.
- **void RBDlt(int n):** Deletes a node based on bNum. It calls the dltNode(RBNode *x) after the node is found.

- **void RBPrnt(int a) :** Prints the Building node based on bNum using search(int n).

- **void RBPrnt(int a, int b):** Prints Building nodes between a range of bNum values.

- **void RBUpdate(int n):** Updates the eTime of the Buildings in the RBT concurrently with the heap.

2. Min Heap.

The Min Heap is implemented using a Struct minHp. It contains the following members:

- **bNum**
- **eTime**
- **tTime**

It maintains one main minHp-hp and another minHp-tempHp. The insertion are done into the tempHp when procd=false and pushed into the main one when pushTmp is called.

- **void Insert(int bnum, int ttime):** Inserts a node into the minHp based on bNum and tTime as inputs. The eTime is set to 0.

- **int hParent(int i):** Returns the parent of particular node in the minHp.

- **void pushTmp():** Inserts from the tempHp into the main minHp-hp.

- **void construct():** Constructs the building at the root of the minHp with each increment of a GlobalCounter(gc). It deletes the node if eTime is equal to tTime and prints the result.

- **void constrFin():** It finishes the construction on each buildings for different cases in construct() by performing heapify and pushing from the tempHp.

- **void hpify(minHp *hp, int n, int i):** It rearranges the various nodes in the tree after an Insertion or Deletion to satisfy the MinHeap condtion. This is done with various condtion checks.

- **void swap(minHp a, minHp b):** It swaps two nodes in the MinHeap.

3. Main Function- int main()

This is the main function which reads the input using ifstream gets each part of input from a line using getline. Depending on the type of Input (Insert or Print Building), the particular functions are called. The results are finally printed to a file. Constructions are done on the building with increase in gc(Global Counter)

Structure of the Program

1. The main function reads the input from the file and stores the parameters into variables. Depending on the type of input, various functions are called to either Insert or Print. The Global Counter is continually increased and construct() is called on the building untill the next Input line is read.

2. Insertion are done into both the MinHeap and the RBT. If size is exceeded then an error message is displayed. Various Swap functions are performed to ensure the correct placement of the node in the RBT and Min Heap based on condition checks.

3. When construct() is called the eTime of the Building is incremented and updated on the RBT. It is done until tTime is reached or until it's been worked on for 5 days, after which hpify() is called. hpify() then ensures the right structure of the minHp.

4. The various RBT functions are called during Inserts or Deletions. It returns different relationships between nodes or perform swaps and rotations. These are done to make sure the RBT structure conditions are satisfied.

5. When the RBPrint is called, depending on the number of parameters the overloaded function prints the correct node in the RBT to the output.

Complexity:

- For **RBPrnt(int a),** a particular node is searched for in a Red Black Tree. The search-time results from the traversal from root to leaf. Since the maximum height of a balanced tree is log(n), the running time is O(log(n)).

- For **RBPrnt(int a, int b)** range, since the printing is done between a range of values the time complexity is O(log(n)+S) where n is number of buildings in the RBT and S is the number of triplets printed.