ADVANCED DATA STRUCTURES (COP 5536)

PROJECT REPORT

November 28, 2019

Raghu Vamsi Badrimahanthi

UF ID: 3791-9564

rbadrimahanthi@ufl.edu

Department of Computer & Information Science

University of Florida – Fall 2019

Working of the Project:

- ***Day*** (Global Counter) is initialised to 0 and increments by 1 every day.
- Next Input command is taken from the file, parsed and stored in an ***input_command*** vector and it is executed when ***Day*** equals the day corresponds to ***Day*** parameter in input command.
- Simultaneously, if ***current_project*** is null, then a project is obtained from the heap which is minimum element and all the elements in heap are ordered according to ***executed_time*** and in case of a tie; ***buildingNum*** is considered to break the tie.
- ***current_project*** is worked either for 5 days or until its completion, whichever is less and executed as following
  If worked for 5 days, add it back into heap and make ***current_project*** = ***null***
  Or if completed within 5 days, remove corresponding node from red-black tree and ***current_project*** is made null.
- This is repeated until input commands are finished or red black tree is empty whichever takes longer.
- If project ends and ***printBuilding*** command occurs on the same day, ***printBuilding*** is executed first and then ***current_project*** is ended

Structure of the Project:

- Classes of ***HeapNode*** and ***RBT_Node*** (Red_Black_Tree_Node) are created with following structures:

```
HeapNode(int curr_buildingNum,int curr_executed_time,int curr_total_time){
  buildingNum = curr_buildingNum;
  executed_time = curr_executed_time;
  total_time = curr_total_time;
  rbt_node = NULL;
}

RBT_Node(int buildingNum){
   val = buildingNum;
   red = true;
   left = NULL;
   right = NULL;
   parent = NULL;
   heap_node = NULL;
}
```

Where `rbt_node` and `heap_node` refer to pointers to each other in both the structures and `red` in red black tree is a ***bool*** type which denotes whether tree node is red or not.
`Left, right, parent denotes left child, right child and parent rbt_node pointers in red black tree`

- ***BuildingClass*** has been created where all the functions such as inserts , deletes or searches are performed on heap and Red Black tree

- Heap is implemented using an array of ***HeapNode*** pointers which stores the building data compared by the executed times and in case of a tie, building numbers are considered.

- Red Black tree is implemented by pointers of ***RBT_Node*** class and linking each other nodes with red black tree properties forming a tree structure.

Function Prototypes:

- int heap_curr_size = 0;
  A variable which keeps track of size of the heap.

- HeapNode* heap[2000];
  A heap implemented by array of pointers which is initialized with a size of 2000.

- RBT_Node* root = NULL;
  A red-black tree root node pointer, which points root of the Red Black Tree

- **void heap_swap_nodes**(HeapNode** parent_ptr, HeapNode** child_ptr);
  A swap function which swap contents of two pointers. This function is used in heapifying the heap after deletion or insertion of elements.

- **void heapify**(**int** child_index);
  A function which is called recursively whenever a new node is inserted into a heap. It checks with parent node and swaps if required and recursively calls the function with the parent node.

- **void min_heapify**(**int** parent_index);
  A function which is called recursively whenever a minimum is deleted from heap. Minimum is replaced by last node in the heap and checks with both left and right child and swaps with either of them if necessary and calls the function with swapped child.

- HeapNode* **heap_get_min**();
  Returns building HeapNode which is minimum in the heap, which is heap[0].

- **void heap_remove_min**();
  removes minimum HeapNode from the heap. Function replaces heap[0] with last element in the heap, I.e. heap[heap_curr_size].

- ❑ **void heap_node_insert**(HeapNode* heapNode);
  inserts a HeapNode into the heap at last position and increases the heap size by 1.

- ❑ **void balance_rotate_left**(RBT_Node* child, RBT_Node* node_parent);
  A function which rotates the child and parent in configuration of ***LL rotation***

- ❑ **void balance_rotate_right**(RBT_Node* child, RBT_Node* node_parent);
  A function which rotates the child and parent in configuration of ***RR rotation***

- ❑ **void change_colours**(RBT_Node* rbt_node,RBT_Node* node_child);
  This function changes colour of node and its corresponding child after rotation so as to remove double red configuration in red black tree.

- ❑ **void parent_null_link**(RBT_Node* rbt_node);
  links parent of deleted node to NULL so as to denote deletion of particular node.

- ❑ RBT_Node* **RBT_get_sibling**(RBT_Node* rbt_node);
  Returns the pointer to other child of parent to the current node.

- ❑ **void RBT_node_balancing**(RBT_Node* rbt_node);
  After deletion of a black node from red black tree, due to imbalance in black colour nodes, node colour adjustments and rotations are made in this function so as to balance the red black property in the tree.

- ❑ RBT_Node* **RBT_get_inorder_predecessor**(RBT_Node* rbt_node);
  Returns a pointer to inorder predecessor, so as to facilitate red black deletion of a red black node whose degree is 2.

- ❑ **void RBT_node_delete**(RBT_Node* rbt_node);
  converts degree 1 nodes and degree 2 nodes by swapping of node values with its child and inorder predecessor espectively and thus facilitating node delete from a leaf.

- ❑ **void RBT_adjust_colour**(RBT_Node* rbt_node);
  After insertion of a node into a red black tree which is generally a node, there might be a case where two red nodes are next to each other. In that case, this function does suitable rotations or colour flips for red black property of the tree to stay true.

- ❑ **void RBT_node_insert**(RBT_Node* rbt_node);
  inserts a rode node into the tree calls the function ***RBT_adjust_colours*** for red black tree property to stay true.

- ❑ RBT_Node* **RBT_node_parent_search**(RBT_Node* rbt_node,**int** target);
  Searches the parent node to which new RBT_node must be inserted.

- ❑ **void building_insert**(int curr_buildingNum,int curr_executed_time, int curr_total_time);
  Creates both red black tree node and heap node, links each other by pointers and then calls ***RBT_node_insert*** and ***heap_node_insert*** separately to ass the nodes into corresponding structures

- ❑ **void building_delete**(RBT_Node* buildingNode);
  Deletes a building from red black tree

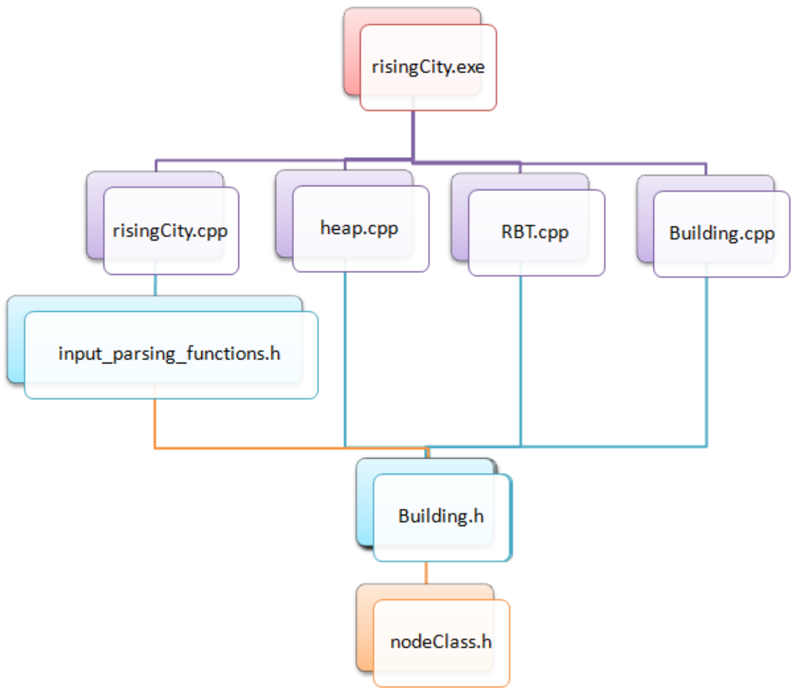
- ❑ RBT_Node* **RBT_search_val**(RBT_Node* rbt_node, int target_buildingNum);
  Uses depth first search to search for node value and returns pointer to corresponding RBT_node for printing the triplet for ***PrintBuilding*** command

- ❑ HeapNode* **get_building_triplet**(int buildingNum);
  Returns heapNode which corresponds to buildingNum and prints the triplet.

- ❑ string **print_active_buildings**(int buildingNum1,int buildingNum2);
  uses inorder traversal to return string with all the triplets that are in range of buildingNum1 and buildingNum2

- ❑ string **inorder_traversal_range**(RBT_Node* rbt_node,int buildingNum1,int buildingNum2, string result);
  recursive inorder traversal in a tree to obtain active buildings in given range.

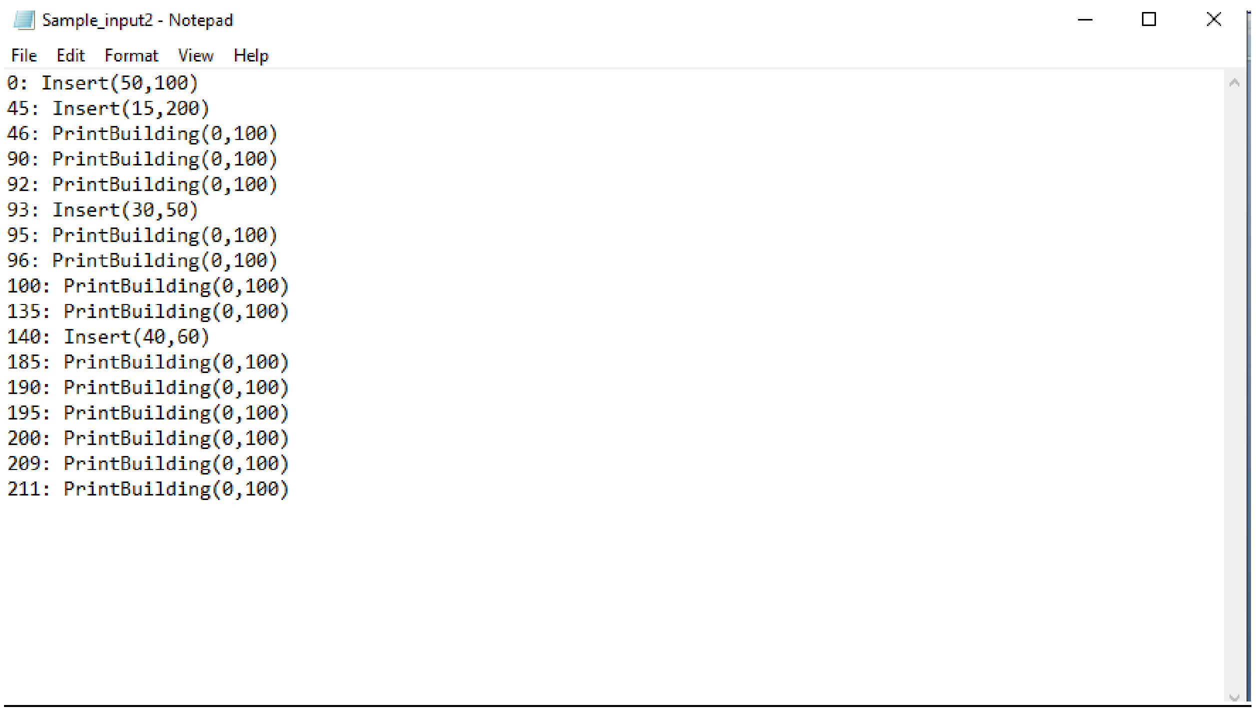
<u>**Project files and Dependencies:**</u>



Result:

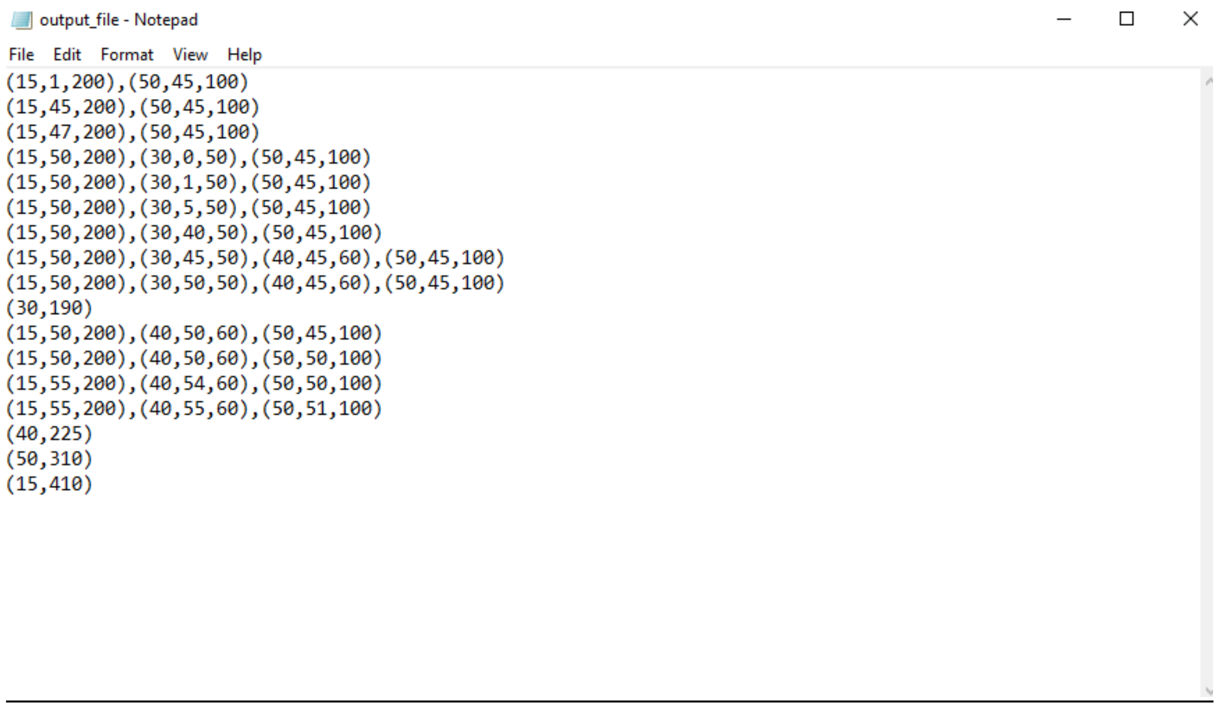
For a given sample input file, result has been taken.

Input:



```
0: Insert(50,100)
45: Insert(15,200)
46: PrintBuilding(0,100)
90: PrintBuilding(0,100)
92: PrintBuilding(0,100)
93: Insert(30,50)
95: PrintBuilding(0,100)
96: PrintBuilding(0,100)
100: PrintBuilding(0,100)
135: PrintBuilding(0,100)
140: Insert(40,60)
185: PrintBuilding(0,100)
190: PrintBuilding(0,100)
195: PrintBuilding(0,100)
200: PrintBuilding(0,100)
209: PrintBuilding(0,100)
211: PrintBuilding(0,100)
```

Output:



```
(15,1,200),(50,45,100)
(15,45,200),(50,45,100)
(15,47,200),(50,45,100)
(15,50,200),(30,0,50),(50,45,100)
(15,50,200),(30,1,50),(50,45,100)
(15,50,200),(30,5,50),(50,45,100)
(15,50,200),(30,40,50),(50,45,100)
(15,50,200),(30,45,50),(40,45,60),(50,45,100)
(15,50,200),(30,50,50),(40,45,60),(50,45,100)
(30,190)
(15,50,200),(40,50,60),(50,45,100)
(15,50,200),(40,50,60),(50,50,100)
(15,55,200),(40,54,60),(50,50,100)
(15,55,200),(40,55,60),(50,51,100)
(40,225)
(50,310)
(15,410)
```