Assignment: Binary Search Tree + Heap = Treap

October 10, 2012

Binary search tree, as shown in figure 1, is a binary tree data structure which is used to hold key-value pairs and satisfies following properties;

- The left subtree of any node contains only those nodes with keys less than the node's key.
- The right subtree of any node contains only those nodes with keys greater than the node's key.

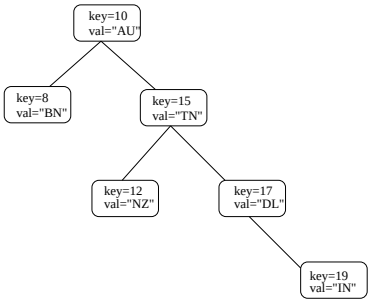

Figure 1: An example Binary Search Tree

These ordering constraints help to find out a node with a given key in O(log n) average time. Insertion and deletion operations in BST (Binary search tree) must maintain these ordering constraints.

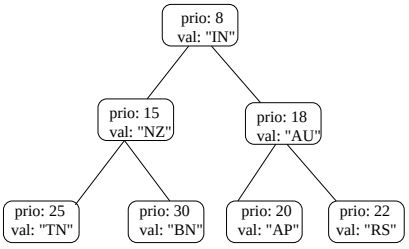

Figure 2: An example min-heap

Heap is an another binary tree data structure, as shown in figure 2, which is useful in sorting (heapsort) and implementing a priority queue. A heap can be a min-heap or a max-heap based on the priority of a node with respect to the

priority of its left and right children. In case of min-heap each node has lesser priority than its children. In case of max-heap each node has higher priority than its children. In this assignment we are going to implement a binary tree data structure which is a combination of BST and heap. This data structure is called **Treap**.

Treap Treap is a binary tree data structure where each node contains 3 attributes;

- **Priority:** This is used to maintain the heap status of treap. Each node in a **Treap** must have lower priority than its children. (in case of min-heap).
- **Key:** This is used to maintain the BST status of treap. Every node in the left sub-tree of a node must have lesser key numbers than node's key. Similarly every node in the right sub-tree of a node must have larger key numbers than node's key.
- **Value:** This represents the actual value stored in a node.

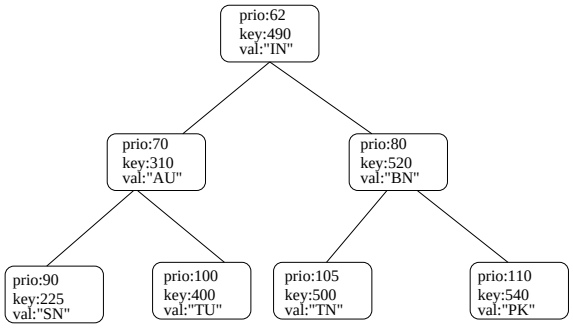

Figure 3: An example treap

An example treap is shown in figure 3 which satisfies the properties of BST, with respect to "key", and the properties of min-heap, with respect to "priority". Following operations are defined over a treap T.

- **search(key):** Given a key k find out the value associated with this key, if any, present in T. For example, calling function search(400) on the treap of figure 3 returns "TU". On the other hand, search(420) returns -1 denoting that there is no node in the treap matching this key.
- **deletion(key):** Given a key k deletes that node of T whose key matches with k. Based on the type of deleted node, internal node or leaf node, treap is adjusted (changed) so as to satisfy the Treap invariants (ordering constraints over priorities and keys).
- **insertion(key, value, priority):** Given a key k, value v and priority p this function inserts a node having these attributes at appropriate position in T. This method should make sure that the invariants of Treap are satisfied after this insertion. Note that insertion happens according to the BST property and then if any heap order violations occur, procedure similar to AVL trees is applied.

- **changing the priority (key, newpriority):** Given a key k it first searches for a node in T whose key matches with k. If such a node is found then its priority is changed to newpriority. As a result of this change the Treap invariants might get violated. Appropriate steps must be taken to establish these invariants again in T.

Problem statement There are two parts of this assignment. First, you have to implement treap data structure by writing a class `Treap` and associated methods as below.

Listing 1: An empty public java class.

```
class Treap
{

  int mKey;
  int mValue;
  int mPriority;

  Treap mLeft =  null;
  Treap mRight = null;

   //Other attributes/variables of this class as needed.

   public int Search(int key) throws KeyNotFoundInTreap {


   }

   public Treap Delete(int key) throws KeyNotFoundInTreap {


   }

   public Treap Insert(int key, int value, int priority)
                               throws KeyAlreadyExistsInTreap {

   }

   public Treap ModifyPriority(int key, int newpriority)
                               throws KeyNotFoundInTreap {

   }


   public void DumpTreap(String filename){

   }

   //Other methods of this class as needed.

}
```

You can add more methods and attributes to this class depending upon your implementation. You will also implement exception classes `KeyAlreadyExistsInTreap` and `KeyNotFoundInTreap` which are used to denote whether or not a given key

exits in a given treap. In the second part of this assignment you will use your implementation of Treap class to give following command line user interface.

```
Press any number (1-6) to denote your choice:
 1. Insert a node in the treap
 2. Delete a node in the treap
 3. Search a node in the treap
 4. Modify the priority of a node
 5. Output the treap data structure to a file
 6. Exit from this menu
```

If user enters 1 then your program will ask user for a key, a value and a priority of the new node and insert it in the treap. If user enters 2 then your program will ask user for a key and then delete the node having this key from the current treap. If user enters 3 then your program ask user for a key and returns the value associated with this key, if any, in the current treap. If user enters 4 then ask user for a key and new priority and modify the priority in the current treap. During any of these actions appropriate error messages must be shown to the user. For example searching or deleting a node that does not exists in the current treap must inform the user that no such node exists. Similarly inserting a key that is already in the current treap should inform user that a node with the same key already exists. **It is your responsibility to handle all error cases which can arise because of user's interaction with your program**. One important point to note is that **after your program completes the command given by the user it should not exit. After completing a given command it should again display the choices to the user**. User should be able to exit your program by entering option 6 from this menu.

If user enters 5 then you must ask user for an output file name and then traverse the treap data structure to write it in the given file. Following paragraph describes the format used for writing treap in a file and how to use it in your debugging activities.

Note:

- The input and all the interaction with the user would be through command line.
- You can assume an empty tree at the beginning and user would first insert an item which would become the root of the tree. i.e tree should be built from scratch.
- The attributes and methods given above should be kept according to the format mentioned. New variables or methods can be added though.

Dot format for writing graphs *Dot* is a very popular format for writing graphs in textual form. Once a graph is described using this format then there exists many graph rendering tools which take this dot file as input and draw the graph. In this assignment we will use this dot format to dump our treap data structure in a file. Following is an example dot file which contains a very small treap structure.

```
digraph G{
0 [color="Black", label="Key:1, Value:23, Prio:3"];
1 [color="Blue", label="Key:2, Value:3, Prio:5"];
2 [color="Red", label="Key:5, Value: 19, Prio:2"];
0->1;
0->2;
}
```

Every dot file starts with `digraph G`. Line 2, 3 and 4 describe nodes of this graph. Each of these lines start with an integer which uniquely identifies a given node. Rest of the text (in between square brackets) define properties of this node. In our case we want to print the key, value and the priority of that node. Therefore we put all these things as label of that node. You will notice that in lines 2, 3 and 4 while defining a node we also specify its color. We will follow the convention that if a node is a left child of its parent then it must be in blue color, if it is right child of its parent then it must be in red color and the root node must be black in color. After specifying all nodes and their properties now we need to specify their connections. Last two lines $(0- > 1)$ and $(0- > 2)$ denotes that the node denoted by integer 0 (defined in line 2) is connected to the node denoted by integer 1 (defined in line 3). Notice that this is a directed relationship and therefore 0 is the parent of 1. Similarly in line 6, $(0- > 2)$ denotes that node 0 is connected to node 2. Also note that 1 is the left child of 0 because color of node 1 is described as blue according to the above convention and similarly for node 2. For any node, the node property should be described before assigning it as child of any other node.

Let us assume that the output file used for dumping treap in dot format is "a.dot". Now we can use one of these very useful graphviewer tools; `dotty` (available for linux) or `Graphviz` (for windows you can download it from http://www.graphviz.org/) for visualizing the current state of treap. These tools graphically render the graph specified in dot format. Following command is used to view the tree dumped in this dot file.

- dotty a.dot

It is recommended that you first complete the implementation of `dumpTreap` function (to dump your treap in dot format) so that you can graphically visualize the outcome of your insert, delete, modifypriority operations and debug them.