

**COIS 3020H**

**Assignment 3**

**Fall 2024**

Bachelor Of Computer Science, Trent University

COIS-3020H: Data Structures and Algorithms II

Professor: Sheikh Ahmed Munieb

26th November, 2024

Dikshith Reddy Macherla - Student Id: 0789055

Dev Patel - Student Id: 0824874

**Work of Each Group Member:**

Questions 1 & 2: Dikshith Reddy

Questions 3 & 4: Dev Patel

## Assignment 3

### Question 1

Code:

```
import random

# Class to represent a node in the Treap
class TreapNode:
    def __init__(self, key, priority):
        self.key = key
        self.priority = priority
        self.left = None
        self.right = None

# Function to perform a right rotation
def rotate_right(y):
    x = y.left
    y.left = x.right
    x.right = y
    return x

# Function to perform a left rotation
def rotate_left(x):
    y = x.right
    x.right = y.left
    y.left = x
    return y

# Function to insert a key into the Treap
def treap_insert(root, key, priority):
    if root is None:
        return TreapNode(key, priority)

    # Insert in the correct subtree based on key
    if key < root.key:
        root.left = treap_insert(root.left, key, priority)
        # Rotate right if heap property is violated
        if root.left and root.left.priority > root.priority:
            root = rotate_right(root)
    elif key > root.key:
        root.right = treap_insert(root.right, key, priority)
        # Rotate left if heap property is violated
        if root.right and root.right.priority > root.priority:
            root = rotate_left(root)

    return root

# Function to print the Treap (in-order traversal)
def print_treap(root, depth=0):
    if root:
        print_treap(root.right, depth + 1)
        print("    " * depth + f"({root.key}, {root.priority})")
        print_treap(root.left, depth + 1)
```

```
# Main program to take input and build the Treap
if __name__ == "__main__":
    d = {}
    n = int(input("How many keys you want to enter: "))
    print(f>Please enter {n} keys and their priorities (key, priority): ")

    # Read key-priority pairs
    for _ in range(n):
        key, priority = map(int, input().strip("()").split(","))
        d[key] = priority

    # Build the Treap
    root = None
    for key, priority in d.items():
        root = treap_insert(root, key, priority)

    # Output the Treap structure
    print("\nTreap after insertion (in-order traversal):")
    print_treap(root)
```

## Output:

```
>>> = RESTART: C:\Users\diksh\Desktop\COIS 3020H\COIS 3020H - Assignment 3\question1
.PY
How many keys you want to enter: 6
Please enter 6 keys and their priorities (key, priority):
(12,9)
... (30,3)
... (20,1)
... (40,7)
... (50,4)
... (70,2)

Treap after insertion (in-order traversal):
          (70, 2)
        (50, 4)
      (40, 7)
    (30, 3)
  (20, 1)
(12, 9)
>>>
```

## Question 2:

Here's a detailed step-by-step insertion and deletion process for the given data:

### Input Data

- Insertions: (12,9), (30,3), (20,1), (40,7), (50,4), (70,2)
- Deletion: (40,7)

### Step-by-Step Insertion

**Step 1:** Insert (12,9)

- The tree is empty, so (12,9) becomes the root.
- Current tree:

(12,9)

**Step 2:** Insert (30,3)

- Compare keys:  $30 > 12$ , so insert (30,3) as the right child of (12,9).
- The priority of (30,3) is less than (12,9), so no rotation is needed.
- Current tree:

```

      (12,9)
       \
        (30,3)
  
```

**Step 3:** Insert (20,1)

- Compare keys:  $20 > 12$ , so move to the right of (12,9). Then,  $20 < 30$ , so insert (20,1) as the left child of (30,3).
- The priority of (20,1) is less than its parent (30,3), so no rotation is needed.
- Current tree:

```

      (12,9)
       \
        (30,3)
       /
      (20,1)
  
```

**Step 4:** Insert (40,7)

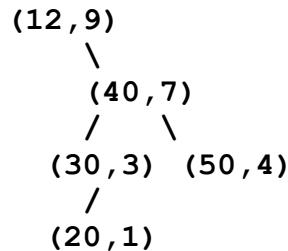
- Compare keys:  $40 > 12$ , move right. Then,  $40 > 30$ , so insert (40,7) as the right child of (30,3).
- The priority of (40,7) is greater than (30,3), so perform a left rotation around (30,3).
- After rotation:

```

      (12,9)
       \
        (40,7)
       /
      (30,3)
       /
      (20,1)
  
```

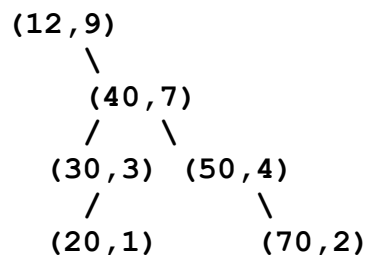
**Step 5:** Insert (50,4)

- Compare keys:  $50 > 12$ , move right. Then,  $50 > 40$ , so insert  $(50,4)$  as the right child of  $(40,7)$ .
- The priority of  $(50,4)$  is less than  $(40,7)$ , so no rotation is needed.
- Current tree:



#### Step 6: Insert $(70,2)$

- Compare keys:  $70 > 12$ , move right. Then,  $70 > 40$ , move right. Finally,  $70 > 50$ , so insert  $(70,2)$  as the right child of  $(50,4)$ .
- The priority of  $(70,2)$  is less than  $(50,4)$ , so no rotation is needed.
- Final tree after insertion:



### Step-by-Step Deletion

#### Delete $(40,7)$

1. Locate  $(40,7)$ :

- $(40,7)$  is the right child of  $(12,9)$ .

2. Reconstruct the subtree:

- Since  $(40,7)$  has two children  $(30,3)$  and  $(50,4)$ , replace  $(40,7)$  with the child node that has a higher priority. In this case,  $(50,4)$  has a higher priority than  $(30,3)$ .

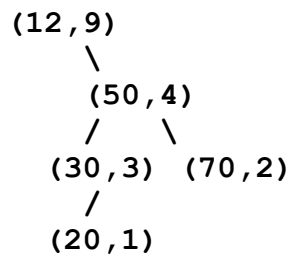
3. Promote  $(50,4)$ :

- Perform a left rotation around  $(40,7)$  to make  $(50,4)$  the new root of the subtree.

4. Reattach (30,3) and (70,2):

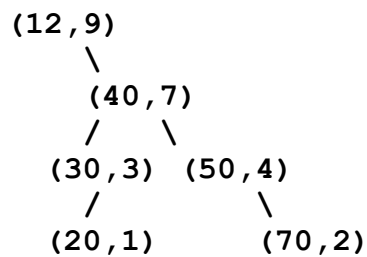
- After rotation:
  - (50,4) remains the parent.
  - Attach (30,3) as the left child of (50,4).
  - (70,2) remains the right child of (50,4).

5. Final tree after deletion:

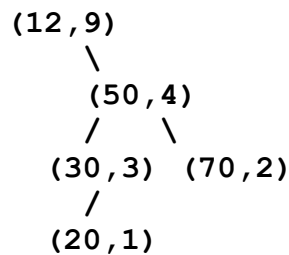


#### Summary

Final Treap after insertion:



Final Treap after deleting (40,7):



This process maintains the Treap's binary search tree and heap properties.

### Question 3:

The **union algorithm of treaps** combines two treaps,  $T_1$  and  $T_2$ , into a single treap, maintaining the **Binary Search Tree (BST)** property for keys and the **heap property** for priorities. Treaps are hybrid data structures that behave like BSTs and heaps, so the algorithm carefully ensures that both properties are preserved.

#### **Key Steps of the Algorithm:**

##### 1. Base Case:

- If one treap is empty, return the other.

##### 2. Compare Priorities:

- Compare the priorities of the roots of  $T_1$  and  $T_2$ .
- Keep the root with the higher priority (to maintain the heap property).
- If  $T_2$ 's root has a higher priority, swap  $T_1$  and  $T_2$ .

##### 3. Split the Treap with Lower Priority:

- Split  $T_2$  (the treap with lower priority root) into two treaps:
  - $L_2$ : Nodes with keys  $< T_1.\text{root.key}$ .
  - $R_2$ : Nodes with keys  $\geq T_1.\text{root.key}$ .
- This ensures the BST property is maintained.

##### 4. Recursive Merge:

- Recursively union:
  - $T_1.\text{left}$  with  $L_2$  (left subtrees).
  - $T_1.\text{right}$  with  $R_2$  (right subtrees).

##### 5. Return the Updated Root:

- The root of  $T_1$  remains the root of the resulting treap.

```

def union(t1, t2):
    """
    Computes the union of two treaps, preserving BST and heap properties.
    """
    if not t1:
        return t2

    if not t2:
        return t1

    if t1.priority < t2.priority:
        t1, t2 = t2, t1

    # Split t2 around t1's root key
    left_t2, right_t2 = split(t2, t1.key)

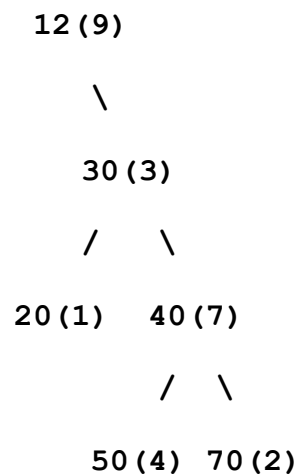
    # Recursively merge left and right subtrees
    t1.left = union(t1.left, left_t2)
    t1.right = union(t1.right, right_t2)

    return t1

```

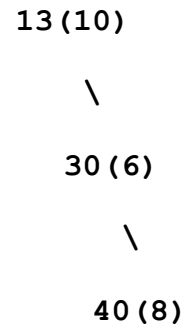
#### **Input Treaps:**

- **Treap d**





- Treap e



### Step-by-Step Union:

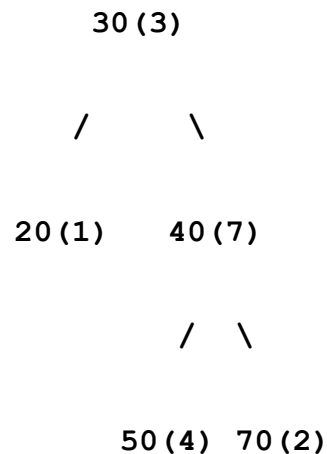
#### Step 1: Root Comparison

- Root of d: (12,9)
- Root of e: (13,10)
- Since 13(10) has higher priority than 12(9), swap d and e. e becomes the root of the resulting treap.

#### Step 2: Split d by e's root key 13

Use the **Split** function on d with key = 13:

- $L_d$  = All nodes in d with keys < 13:
  - 12(9)
- $R_d$  = All nodes in d with keys  $\geq$  13:

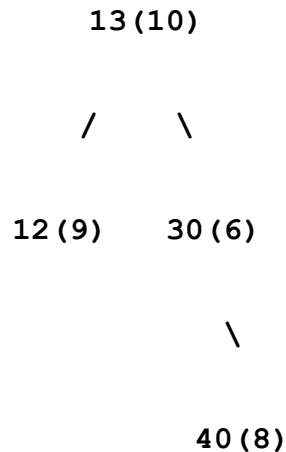


### Step 3: Merge $L_d$ into e's left subtree

Recursively call **Union** on e.left (currently None) and  $L_d=12(9)$ :

- Result: 12(9) becomes e's left child.

Intermediate Treap:



### Step 4: Merge $R_d$ into e's right subtree

Recursively call **Union** on e.right=30(6) and  $R_d$ :

1. Compare roots:

- 30(6) (from e.right) vs 30(3) (from  $R_d$ ): Keep 30(6) as it has higher priority.

2. Split  $R_d$  by key=30:

- $LR_d = 20(1)$  (nodes < 30).
- $RR_d =$  Treap with root 40(7) (nodes > 30).

3. Recursively merge  $LR_d$  (20(1)) with 30(6).left (None):

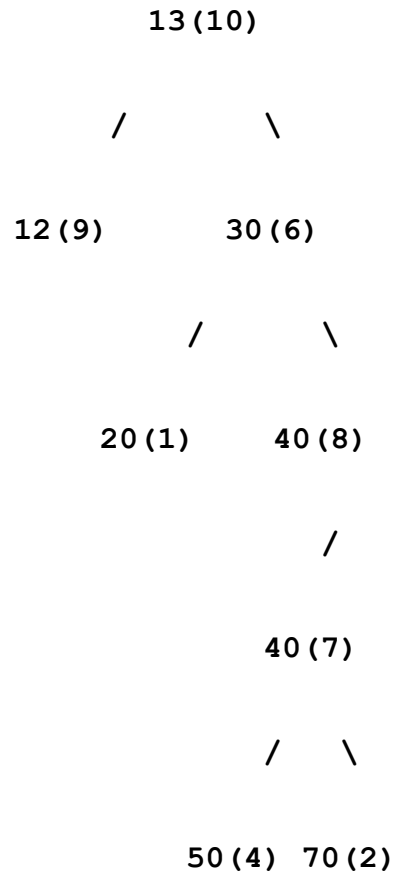
- Result: 20(1) becomes left child of 30(6).

4. Recursively merge  $RR_d$  (40(7)) with 30(6).right (40(8)):

- Compare 40(7) vs 40(8): Keep 40(8) as it has higher priority.
- Split 40(7) by key=40: Both L=None, R=None.
- 40(7) becomes left child of 40(8).

### Step 5: Combine Final Treap

Combine the intermediate e treap with the result from Step 4:



### Question 4:

#### Step 1: Inserting Words into the Trie

Given the dataset:

d = {apple, apricot, appalling, aqua, aardwolf, bear, beach, baked, baron, zebra, zoo, zest}

##### 1. Start with an empty Trie:

- A Trie is represented as a root node with children.
- Each node stores characters and their children (a mapping of characters to nodes).

- The root does not store any character. Words are marked complete with a unique indicator (e.g., `is_end_of_word`).
- 

#### **Insert "apple"**

- Root → Create a child node for 'a'.
  - 'a' → Create a child for 'p'.
  - 'p' → Create another child for 'p'.
  - 'p' → Add child for 'l'.
  - 'l' → Add child for 'e'.
  - Mark the node for 'e' as `end_of_word = True`.
- 

#### **Insert "apricot"**

- Root → 'a' already exists.
  - 'a' → 'p' already exists.
  - 'p' → Add child for 'r'.
  - 'r' → Add child for 'i'.
  - 'i' → Add child for 'c'.
  - 'c' → Add child for 'o'.
  - 'o' → Add child for 't'.
  - Mark 't' as `end_of_word = True`.
- 

#### **Insert "appalling"**

- Root → 'a' exists.
- 'a' → 'p' exists.

- 'p' → 'p' exists.
  - 'p' → Add child for 'a'.
  - 'a' → Add child for 'l'.
  - 'l' → Add child for 'l'.
  - 'l' → Add child for 'i'.
  - 'i' → Add child for 'n'.
  - 'n' → Add child for 'g'.
  - Mark 'g' as end\_of\_word = True.
- 

#### **Insert "aqua"**

- Root → 'a' exists.
  - 'a' → Add child for 'q'.
  - 'q' → Add child for 'u'.
  - 'u' → Add child for 'a'.
  - Mark 'a' as end\_of\_word = True.
- 

#### **Insert "aardwolf"**

- Root → 'a' exists.
- 'a' → Add child for 'a'.
- 'a' → Add child for 'r'.
- 'r' → Add child for 'd'.
- 'd' → Add child for 'w'.
- 'w' → Add child for 'o'.
- 'o' → Add child for 'l'.

- 'l' → Add child for 'f'.
  - Mark 'f' as end\_of\_word = True.
- 

#### **Insert "bear"**

- Root → Add child for 'b'.
  - 'b' → Add child for 'e'.
  - 'e' → Add child for 'a'.
  - 'a' → Add child for 'r'.
  - Mark 'r' as end\_of\_word = True.
- 

#### **Insert "beach"**

- Root → 'b' exists.
  - 'b' → 'e' exists.
  - 'e' → 'a' exists.
  - 'a' → Add child for 'c'.
  - 'c' → Add child for 'h'.
  - Mark 'h' as end\_of\_word = True.
- 

#### **Insert "baked"**

- Root → 'b' exists.
- 'b' → Add child for 'a'.
- 'a' → Add child for 'k'.
- 'k' → Add child for 'e'.
- 'e' → Add child for 'd'.

- Mark 'd' as end\_of\_word = True.
- 

#### **Insert "baron"**

- Root → 'b' exists.
  - 'b' → 'a' exists.
  - 'a' → Add child for 'r'.
  - 'r' → Add child for 'o'.
  - 'o' → Add child for 'n'.
  - Mark 'n' as end\_of\_word = True.
- 

#### **Insert "zebra"**

- Root → Add child for 'z'.
  - 'z' → Add child for 'e'.
  - 'e' → Add child for 'b'.
  - 'b' → Add child for 'r'.
  - 'r' → Add child for 'a'.
  - Mark 'a' as end\_of\_word = True.
- 

#### **Insert "zoo"**

- Root → 'z' exists.
- 'z' → Add child for 'o'.
- 'o' → Add another child for 'o'.
- Mark 'o' as end\_of\_word = True.

---

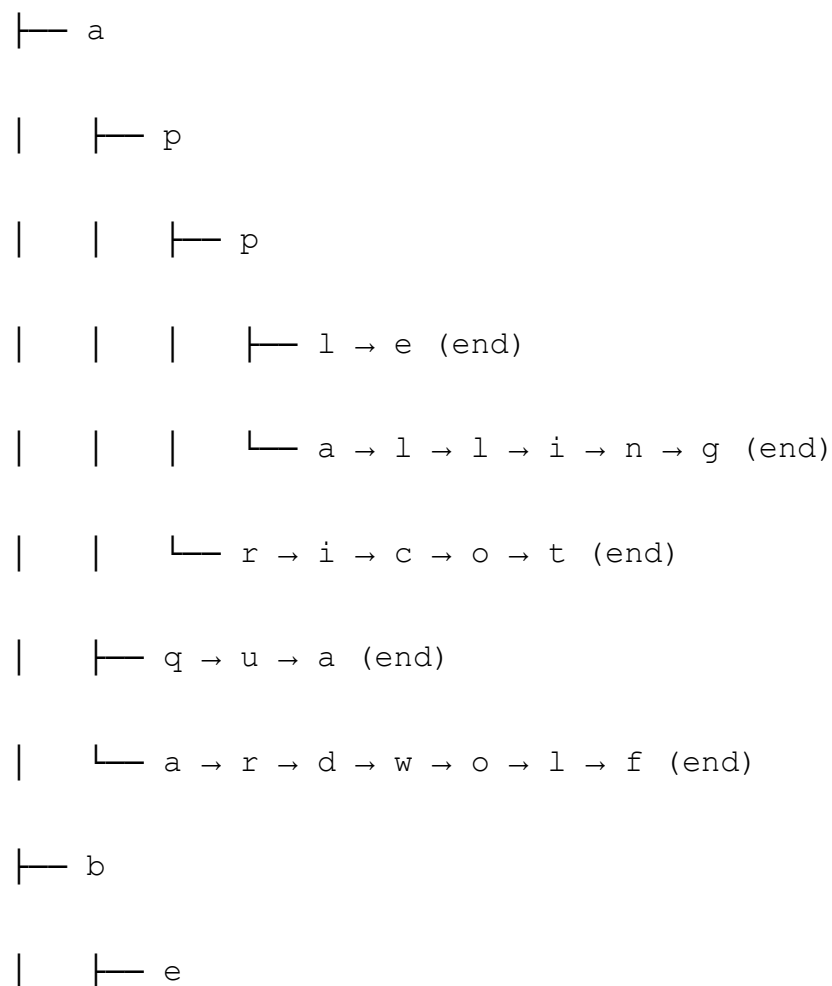
## Insert "zest"

- Root  $\rightarrow$  'z' exists.
- 'z'  $\rightarrow$  'e' exists.
- 'e'  $\rightarrow$  Add child for 's'.
- 's'  $\rightarrow$  Add child for 't'.
- Mark 't' as `end_of_word = True`.

## Trie Structure After Insertions

A visual representation of the Trie after all insertions (simplified for readability):

Root





| | └─ a → r (end)

| | └─ a → c → h (end)

| └─ a

| | └─ k → e → d (end)

| | └─ r → o → n (end)

└─ z

| └─ e → b → r → a (end)

| | └─ s → t (end)

| └─ o → o (end)

## Step 2: Deleting Words

To delete words, ensure that nodes are only removed if other words no longer share them.

### Delete "appalling"

- Traverse: a → p → p → a → l → l → i → n → g.
- Mark 'g' as end\_of\_word = False.
- Remove nodes backward if they are no longer part of another word.
- 'g', 'n', 'i', 'l', 'l', and 'a' are removed as they have no other branches.
- Trie now keeps the "apple" and "apricot" path intact.

---

### Delete "beach"

- Traverse: b → e → a → c → h.

- Mark 'h' as `end_of_word = False`.
- Remove 'h' and 'c' (no other dependencies exist).
- "bear" remains intact.

---

### Delete "zest"

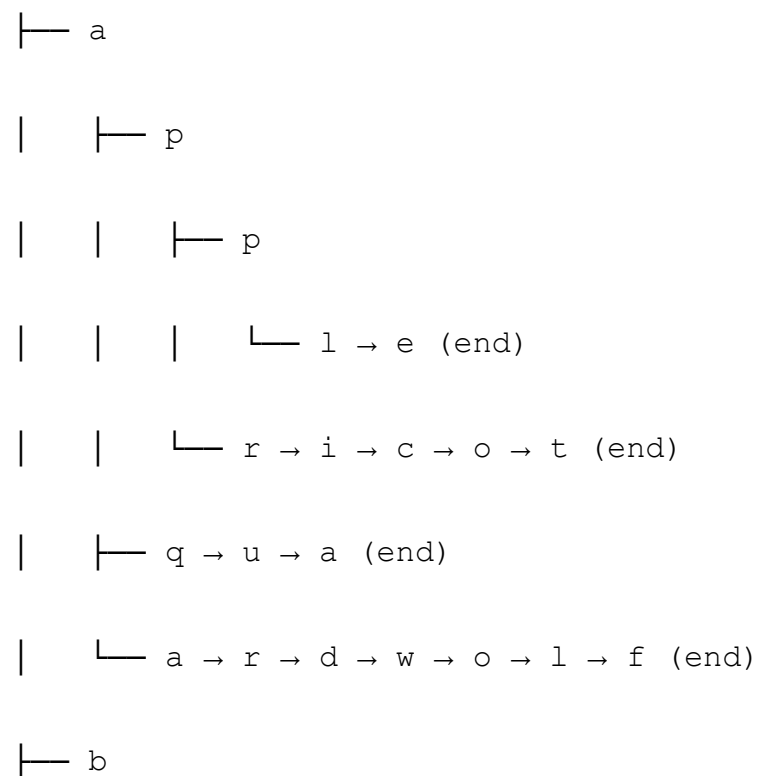
- Traverse:  $z \rightarrow e \rightarrow s \rightarrow t$ .
- Mark 't' as `end_of_word = False`.
- Remove 't' and 's' (no other dependencies exist).
- "zebra" remains intact.

---

### Final Trie Structure

After deletions:

Root



|     ⊢ e

|     |     ⊢ a → r (end)

|     ⊢ a

|     |     ⊢ k → e → d (end)

|     |     ⊢ r → o → n (end)

⊢ z

|     ⊢ e → b → r → a (end)

|     ⊢ o → o (end)