CSE 2050: Algorithms and Data Structures II Fall 2025
Homework 06: Binary Search Trees
Total Points: 50

1 Randomly Balanced Binary Search Tree

A binary search tree (BST) is a data structure that maintains a collection of keys (along with associated values) which encodes a sorted ordering of the elements withing its inorder traversal. This allows us to perform several operations that are not possible with other data structures like hash tables. Balancing is performed using a simple probabilistic heuristic that is described later in the handout. This ensures that all root to leaf path operations run in only $O(\log n)$ time.

Our implementation of the BST will be more generic. The types of the keys and values are not important. Any comparable type that overrides the `__lt__`, `__le__`, and `__eq__` methods can be used as keys in the BST.

1.1 Getting Started

Please download the file `bst.py` from Canvas. You will be working within this file. This file currently contains an implementation of a single `Node` object in the BST. The class `Node` is fully implemented. Each `Node` class has the following attributes.

1. `key`: An attribute that defines an ordering of the elements in the BST. As mentioned, this is any comparable type that overrides the `__lt__`, `__le__`, and `__eq__` functions. Typical types used for keys are integers, strings, and tuples.
2. `value:` Each `key` can have an associated `value`. This makes the BST behave like a map data structure (that stores a collection of key, and value pairs).
3. `size:` Each `Node` in the BST is augmented with the number of elements in its subtree. This is useful to perform `select`, `join`, and in balancing the tree.
4. `left, right`: Pointers or references to the left and right children (or subtrees).

In addition to these attributes, the `Node` class contains an `update_size` function that updates the augmented `size` attribute. Call this function whenever you need an updated size of your tree. Note that this only adds the sizes of its left and right subtrees. Call this only when you know that the corresponding sizes of the left and right subtrees are correct. A `__str__` function is also provided that allows you to test by printing a string representation of an object of this class.

1.2 Randomly Build BST

Our binary search tree is implemented in the class `BST`. The following operations are to be supported.

1. `__len__`: Returns the size of the BST. This is the number of elements contained in the BST. Note that this should reflect the correct size after performing an `insert` or a `delete`.

2. `insert(key, value)`: This method inserts a `key` along with its associated `value` into the BST. No balancing is performed. See `balanced_insert` if you wish to insert while balancing the BST.

3. `find(key)`: Returns an item (key, and value pair) associated with this `key`. If multiple keys are present, it could return any one of them. If the `key` is not present, returns `(None, None)`.

4. `min()`: This returns an item (key, value pair) associated with the smallest key. If multiple keys are tied for the smallest, it could return any one of them. Returns `(None, None)` if the tree is empty.

5. `max()`: This returns an item (key, value pair) associated with the largest key. If multiple keys are tied for the largest, it could return any one of them. Returns `(None, None)` if the tree is empty.

6. `pred(key)`: This returns the item (key, value pair) with the largest key smaller than or equal to `key`. Returns `(None, None)` if there is nothing smaller than `key`.

7. `succ(key)`: This returns the item (key, value pair) with the smallest key greater than or equal to `key`. Returns `(None, None)` if there is nothing smaller than `key`.

8. `delete(key, value)`: This removes a node with the given item (key, value pair). If no node exists with this item, then nothing is removed. If there are multiple nodes with the same `key` and `value`, any one of them is removed. No balancing is performed. See `balanced_delete` if you wish to delete while balancing the BST.

9. `select(k)`: This function returns the item (key, value pair) in position `k` in an inorder traversal of the tree. Note that the inorder traversal produces a sorted ordering of the keys in the BST. If `k = 1`, this function returns the minimum item, and if `k = len(.)` or the size of the BST, it returns the maximum. Position values (also called ranks) can only be between 1 and the size of the BST. It produces an `assert` error if this is not correct.

10. `inorder()`: This method returns a generator object that can be used to iterate over and print the contents of the tree. The generator object should represent the items (key, value pairs) of the keys in the BST in an inorder traversal. So the keys are all in sorted order.

11. `split(key)`: This function should return two BST objects `L` and `R`, where `L` contains all keys (and their values) smaller than or equal to `key`, whereas `R` contains all keys (and their values) that are larger than `key`.

12. `join(L, R)`: This function takes two BST objects `L` and `R`, where all keys in `L` are strictly smaller than all keys in `R`, and joins the two trees together. The joined tree is stored in the calling object (i.e., `L`) and note that the BST `R` is modified and therefore cannot be used as a separate BST since its keys are merged with `L`. This procedure joins the two trees using a probabilistic procedure. With probability $\frac{\text{len}(L)}{\text{len}(L)+\text{len}(R)}$, the root is chosen from the left tree, and we recursively join the right subtree of `L` with `R`. Else the root is chosen from the right tree, and we recursively join the left subtree of `R` with `L`.

13. `balanced_insert(key, value)`: This method inserts a `key` along with its associated `value` into the BST. The BST is balanced using a probabilistic procedure. Every key has an equally likely chance to be at the root. So insert the new key (and its value) as the root node of the tree with probability $\frac{1}{\text{len}(.)+1}$, and recursively insert in the left or right subtree of the root otherwise.

14. `balanced_delete(key, value)`: This removes a node with the given `key` and `value`. If no node exists with this item, then nothing is removed. If there are multiple nodes with the same `key` and `value`, any one of them is removed. Balancing is performed by calling `join` to replace the node to delete with a join of its left and right subtrees.

Note that most of the above methods are public wrapper methods to other private methods. From the user's point of view, only these public methods are called so the user is not concerned about the actual implementations of these functions. Some of these functions are recursive. So these wrapper functions call other private functions, thereby offering flexibility of implementation by passing `Node` objects. Please do not change the signatures of these wrapper functions as these are the specifications of the BST. You may choose to change the signatures of other functions to cater to your implementation. Please complete implementation of all the functions. Specifically, any place that has the statement `pass` should be replaced with your implementation.

1.3 Testing

In order to thoroughly test your implementation, a test suite is provided in `bst-driver.py`. A driver program is a test program that can be used to test the different implementations of another program. This driver program imports your `bst` package. Make sure the file `bst.py` resides in the same working directory as `bst-driver.py`. The driver program calls the various functions in your BST in order, and raises exceptions if the desired output is not reached. Each error comes with an error code identifying what is wrong. You are expected to implement the functions one at a time, removing one error code at a time. When all functions are correct, your test suite completes gracefully with the appropriate message.

Note that this form of testing is called *black-box testing*. The operations are treated as black boxes. We only test by ensuring the specifications of each of these functions are satisfied by calling the BST operations many times. Thorough testing of each of the implementation methods (also called *glass-box testing*) is required to ensure all operations meet the required specifications.

2 Runway Reservation

Suppose that there is a single runway in an airport. Airlines can make reservations on the runway based on the times they need to use the runway. In order to use the runway efficiently and to accommodate for slight changes in flight schedules, we allow a grace period of k units of time between successive use of the runway. All runway reservation requests are to be processed on a first-come-first-serve basis. In this problem, we will print out all the requests that have been successful in using the runway. Note that the above constraints mean that some requests may be unsuccessful (see Figure below).

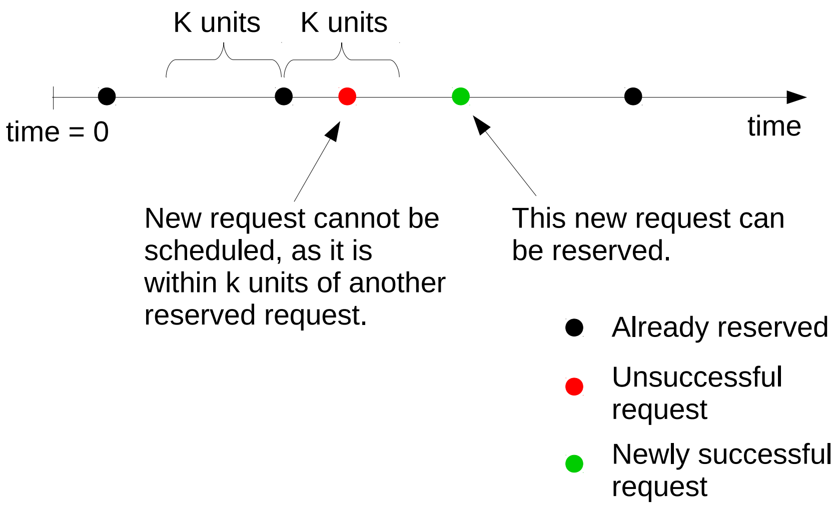
K units
K units
time = 0
time
New request cannot be scheduled, as it is within k units of another reserved request.
This new request can be reserved.
Already reserved
Unsuccessful request
Newly successful request

Figure:

2.1 Input and Output

The input file contains two numbers n and k on the first line that specifies the total number of requests and the grace time period between successive requests respectively. The next n lines provide information about a request. Each request consists of one of two commands – a 'r' or 'request reservation' command to request use of the runway for an airline or a 't' or 'advance time' command to move time forwards by some units. Each 'r' command follows with information about the airline – namely the time to use the runway, the airline flight name, number, its source and destination airports, all space separated on the same line. Each 't' request follows with some integer time units to advance the current time. See below for sample input and output.

The output should be formatted as follows. For each 't' command, we print the current time followed by all successful airlines that have used the runway after the previous setting of the current time (either the beginning of day or the previous 't' command). We assume that there is a 't' command at the end of input that advances current time to the last successful request. See below for sample input and output. Also see input files `runway-reservation.in1`, `runway-reservation.in2`, and their respective outputs `runway-reservation.out1` and `runway-reservation.out2`.

```
Sample Input:

11 5
r 20 UA 1545 MEL IAH
r 9 UA 1714 MEL IAH
r 6 AA 1141 MEL MIA
r 5 B6 725 BQN MEL
```

```
r 10 DL 461 ATL MEL
r 6 B6 79 MCO MEL
t 7
r 25 UA 1696 ORD MEL
r 7 B6 507 MEL FLL
t 10
r 24 EV 5708 IAD MEL

Sample Output:

current time = 7
current time = 17
UA 1714 MEL IAH
current time = 25
UA 1545 MEL IAH
UA 1696 ORD MEL
```

3 Cookie Inspection

Your family-owned bakery that produces the most delicious cookies in town is sometimes under inspection for compliance with federal, state, and local laws. So, the inspection firm (i.e., the government) can request some cookies be sent to them for inspection from time to time. (Hopefully these officials do not misuse their power of getting free cookies from you in the name of inspection.) Your cookies come in different sizes, all measured up to the nanometer, and you do not want to give the larger ones for inspection since these fetch you good money. You do not want to give the smaller ones as well for inspection because you fear they may not pass the stringent inspection tests. So, you always decide to give them the cookies that are close to the median in size.

3.1 Input and Output

Each line of the input describes two types of events. A line with a single integer means x that you just baked a new cookie with size x. A line that starts off with a '#' contains another integer y. These lines indicate that the inspection company wants you to send them y cookies. Each of the y cookies are chosen by selecting the median cookie that you have in your collection without replacement, one at a time. As output, for each of the lines containing a '#', please output the y cookies that were chosen for inspection in order. See the sample inputs and outputs. The input could contain at most 500,000 lines.

```
Sample Input:

1
2
3
4
# 2
3
```

```
2
# 1
# 3

Sample Output:

3 2
3
2 4 1
```

4 Postamble

4.1 Input and Output

The input for all programs will always be accepted from standard input (keyboard or `system.in`) and output will always be sent to standard output (monitor or `system.out`). If the input contains many lines, then you can put your input in a separate file, say `progname.in` and re-direct this input into your program by running

```
python3 progname.py < progname.in
```

You could also redirect the output of your program to another file, say `myoutput.txt`.

```
python3 progname.py < progname.in > myoutput.txt
```

If the intended output is in file `progname.out`, then you can use the `diff` utility to check your output by taking the difference of two files.

```
python3 progname.py < progname.in > myoutput.txt
diff myoutput.txt progname.out
```

If the output of the `diff` command returns nothing, this means that your output is the same as the intended output, and is judged correct for a test case. Note that your program is only correct if it is judged correct for all test cases. You are encouraged to design different test cases and test the correctness of your programs thoroughly.

Please do not include extanneous input or output as these would throw off our automated grading scripts. The input and the output should always match the format given in the description of each problem. Sample test cases are provided for you to test the formatting of the input and output, as well as to check the correctness of your program for simple test cases. Note that we will be testing with a large number of test cases.

4.2 Submission

Please submit all the individual files to `Canvas`. Please name your programs as given in the problem description, or refer to the names in the following table.

Problem Name	**File Name**
Randomly Balanced BST	bst.py
Runway Reservation	runway-reservation.py
Cookies Inspection	cookies-inspection.py