Priority Expiry Cache Problem

We'll walk through an actual phone screen question asked at Tesla in this lesson.

We all have heard of the often repeated Least Recently Used (LRU) Cache problem asked in interviews. The problem is trivial once you realize that a hash table and a doubly linked list can be used to solve it efficiently. We'll work on a variant of the LRU problem that was asked in a real-life Tesla phone screen. The intent of this problem is to exercise our minds to be able to compose various data-structures to achieve desired worst-case complexity for various operations. Without further ado, the problem is presented below:

The PriorityExpiryCache has the following methods that can be invoked:

- `get(String key)`
- `set(String key, String value, int priority, int expiry)`
- `evictItem(int currentTime)`

The rules by which the cache operates is are follows:

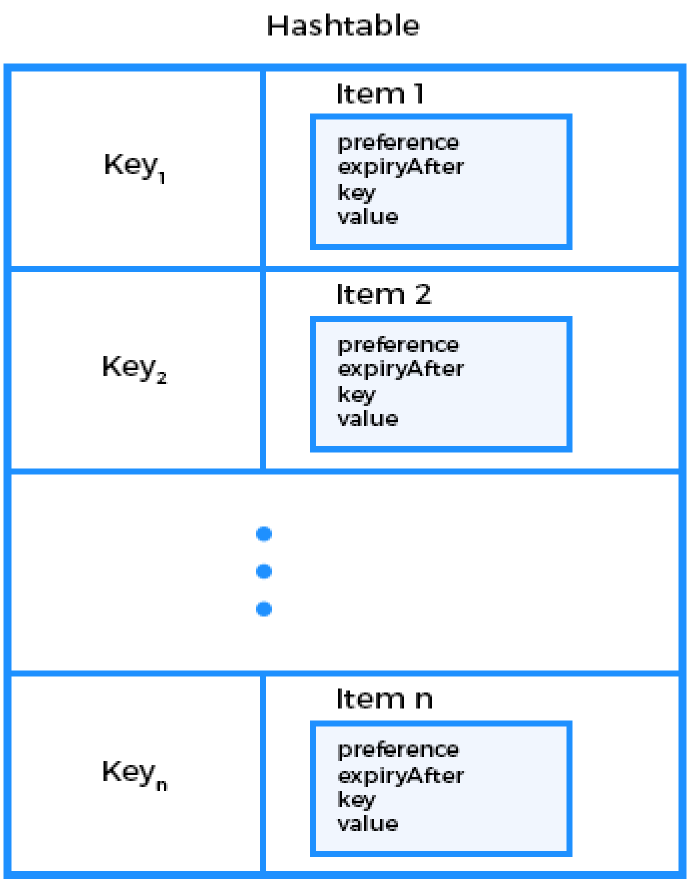
1. If an expired item is available. Remove it. If multiple items have the same expiry, removing any one suffices.
2. If condition #1 can't be satisfied, remove an item with the least priority.
3. If more than one item satisfies condition #2, remove the least recently used one.

And it goes without saying that we have to design our function implementations to be as fast as possible.

Let's start with the `get()` method. Given a key we need to return the associated value. The fastest way we can implement a retrieval operation is to use a hash table. The insertion and retrieval times would both be O(1). Besides the key and the value provided to us, we'll need to store the **expiry** time and the **preference**. So Let's create a class `Item` that holds all these three artifacts associated with the key.

```
class Item {
    int preference;
    int expireAfter;
    String key;
    String value;
}
```

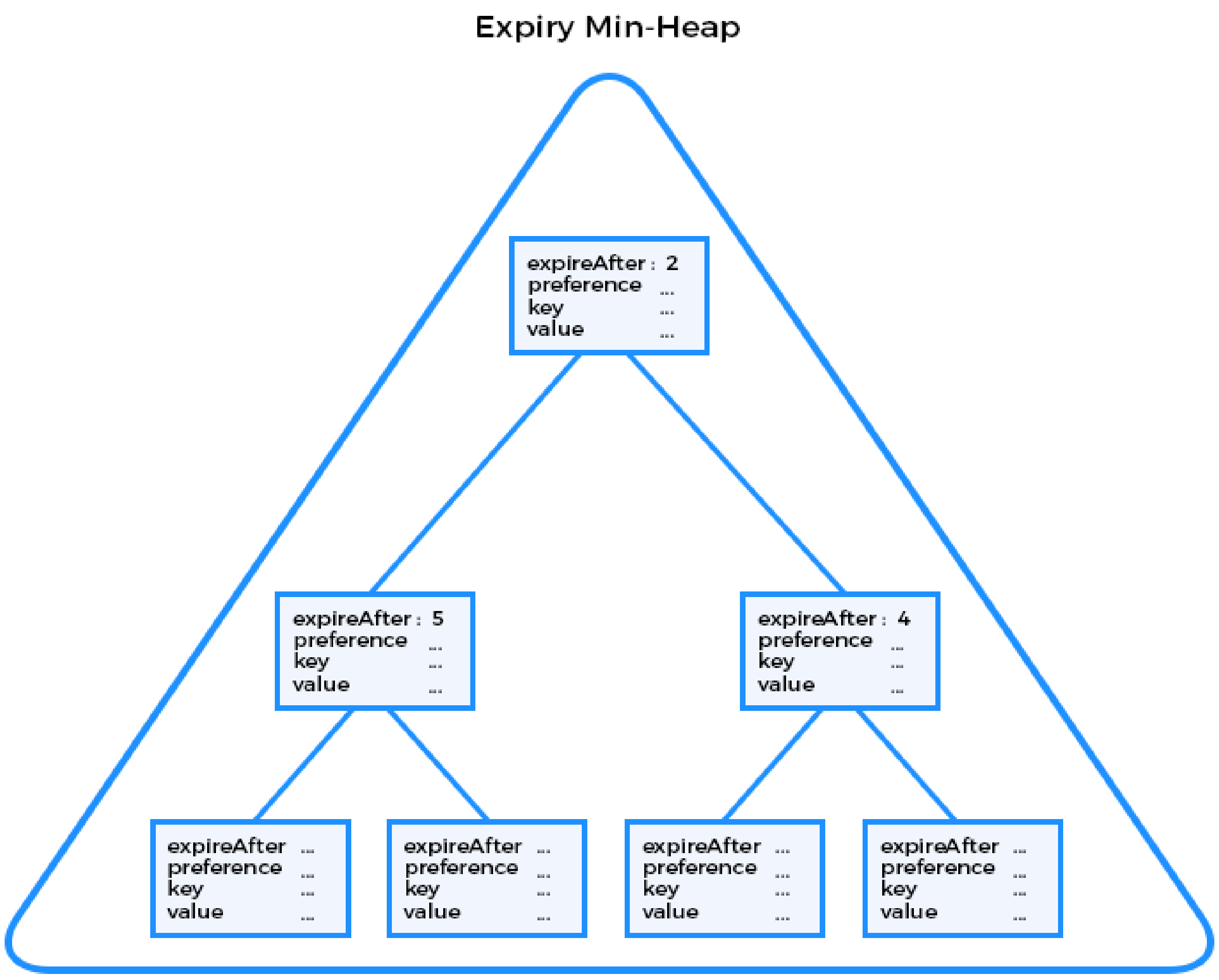
We can now create a hash table using the `Item.key` and the value as the item object itself. It should look like as follows:



We can serve `get()` requests using the above hashtable in O(1). Whenever a get is invoked, we have to capture the notion of an item being most recently used. Hold onto that thought for now as we'll solve it once we create the other needed data-structures.

Let's start rule#1 which says that an expired item should be removed first. We need to find the item with the minimum expiry time. **Whenever you hear yourself looking for the maximum or the minimum among a set of values,**

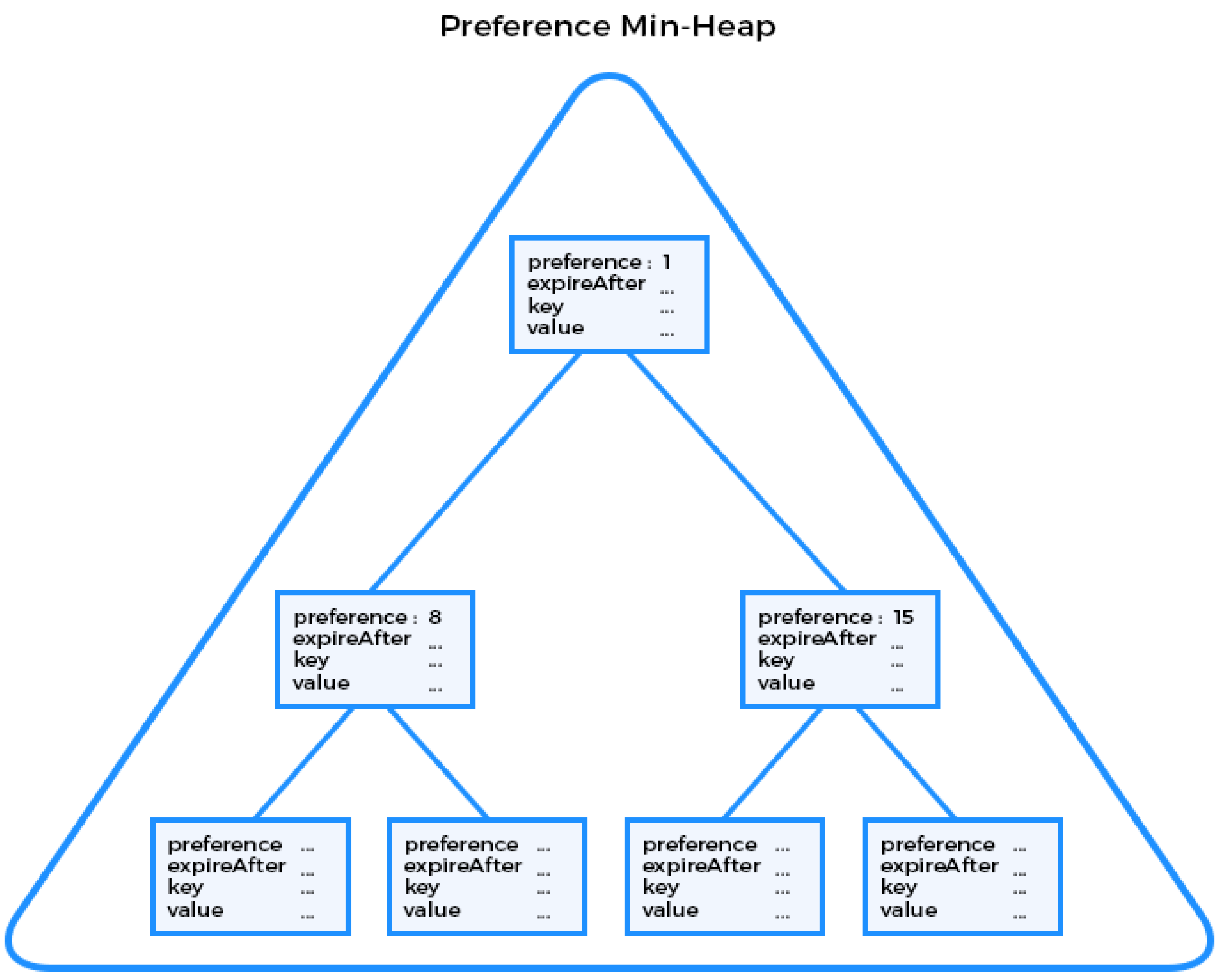
your likely choice of data-structure is a heap (called a priority queue in some language implementations) as it provides O(1) look-up (not removal) for minimum or maximum value. A max-heap can be used for looking up maximum values while a min-heap can be used for looking up minimum values. For our case, we can create a min-heap of the objects of class `Item`. The min-heap is predicated on the `expireAfter` variable, i.e. the min-heap is built based on the value of `expireAfter` variable for each object. Conceptually, the min-heap looks like as follows:



When `evictItem()` is invoked, we peek at the top of the min-heap and see if the item with the minimum expiry time is indeed expired. If so we pop the top of the min-heap and also remove it from our hashtable. Removing the minimum item from the min-heap is O(lgn) and removal from the hashtable is O(1). The total cost thus far is O(lgn) +

$$O(lgn) + O(1) = O(lgn)$$

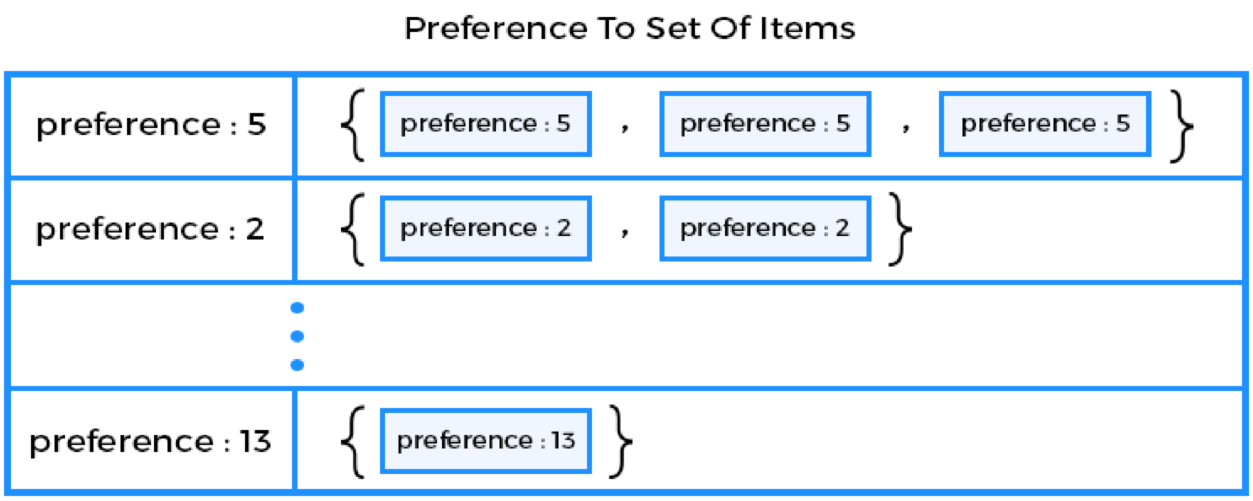
Now consider, what happens if there are no expired items. In that case we need to find an item with the least preference. Again, that requirement hints of using a min-heap. The top of the heap will hold the item with the minimum preference. So now we have two min heaps. The second min heap is predicated on the `preference` variable. The min-heap based on preference looks like:



The next part is tricky, if more than two elements have the same preference, we need to remove the one which was least recently used. This implies that unlike our expiry min-heap, **we can't simply evict the item at the top of the preference min-heap, because it may have been used recently and there might be another item in the min-heap with the same preference but was accessed least recently among all the items with the same priority.**

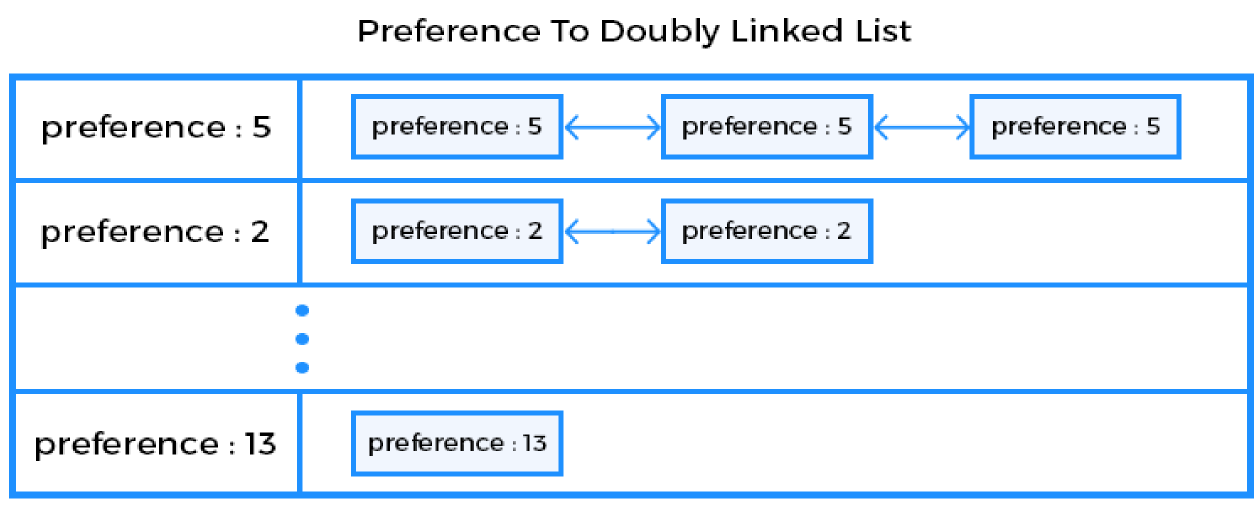
The last sentence of the previous paragraph is very important. Looking at the top of the preference min-heap we need to get to all the elements with the least preference. Once we get to those elements we can remove the one that was least recently used. Ok, now think it through, once we get the least

preference from the min-heap, how can we most efficiently get to all the elements with the same preference? Use a hashtable again! We can have a mapping of preference to the set of all elements with that preference. This should look like as follows:


Preference To Set Of Items
preference : 5
{ preference : 5 , preference : 5 , preference : 5 }
preference : 2
{ preference : 2 , preference : 2 }
preference : 13
{ preference : 13 }

However, keeping all the items with the same preference doesn't help us know which item was least recently used. We need to bring order to these items so that they are arranged in order of when they were last accessed. **Whenever you want to order entities, use a linked list. When you want a collection of entities without any order among them use a set.** So now, we have a mapping of preferences to a list. The key to the hashtable is the preference (an integer) and the value is a list of all the items with the same preference.

Now let's dig into how we can create the notion of least recently used when using a list. We can institute a convention that the head (front) of the list will always have the most recently used item. The implication is that whenever an item is accessed, we'll cut it out from its current position in the linked list and place it at the head of the list. The last item, thus, is always the least recently used item. However, we can't efficiently cut out an element from say the middle oof the list unless we know its predecessor (the one behind) and successor (the one after) nodes in O(1) time. We don't want to traverse the list whenever we want to cut out an item. This should ring a bell that we need a doubly-linked list, which provides O(1) access to both predecessor and successor nodes. Great! so now we have a hashtable of preferences to doubly-linked lists of items with the same preference, which should conceptually look like as follows:



Now we have all the pieces in the puzzle together. We ended up with:

- An expiry-time min-heap (priority queue). The constituent elements of the min-heap can be the doubly-linked list nodes that wrap an item. This will ease manipulating the doubly-linked list when required, as we'll shortly see.

- A preference min-heap (priority queue). The elements making up the min-heap should be the nodes from the doubly-linked list that wrap an item but the min-heap is predicated on the preference value instead of the expiry time.

- A hash table with a mapping of key to doubly-linked list node that wraps the item with the key.

- A hash table with a mapping of preference to doubly-linked list of all the items with the same preference.

get ( )

Now we can reason about the `get()` method's complexity. When `get()` is invoked we get the key as a parameter. Using the key we can get the doubly-linked list node that contains the corresponding `Item` object in O(1) time. Next, in O(1) time we can get the wrapped item and return it to the caller. But just before returning, we must also detach the node from its current position in the list of all items with the same preference and place it at the start of the list, to indicate it is the most recently used item. This should be trivial because we can get the preference of the item in O(1), use it to hash into the bucket

which contains the doubly-linked list of all the items with the same preference. We can remove the item from the doubly linked list in O(1) given the back and forward pointers. Once detached we can now attach it to the head of the doubly-linked list in O(1). The worst-case complexity of the `get()` method is thus:

$$O(1)$$

evictItem ( )

Before discussing `set()` we'll discuss the `evictItem()` as it will be used by the `set()` method, when the cache is full.

We can peek (not remove) the expiry min-heap first to determine if the item with the minimum expiry time is expired and can be removed or not. Peeking or finding min in a min-heap is a O(1) operation. Let's say if the item with minimum expiry is indeed expired, then we can remove the expired item in O(logn) time. We'll also remove it from the preference min-heap which is another O(lgn) operation. We'll also need to delete it from the hashtable of key to wrapped items, which will be an O(1) operation. Next, we also need to remove it from the doubly-linked list of items with the same preference. Since we were returned the doubly-linked list node wrapping the item from the expiry min-heap, we can easily remove it from the doubly-linked list in O(1). The overall complexity will be:

$$O(lgn) + O(lgn) + O(1) + O(1) = O(lgn)$$

Now, let's consider the case when there are no expired items. We'll peek the item with least preference from the preference min-heap in O(1) time. But note that this may or may not be the candidate for removal since it may have been more recently used than another item with the same preference. We'll use the preference of the wrapped item at the top of the min-heap to hash into the bucket containing the doubly-linked list of items with the same preference. Note this is an O(1) operation. Once we have the doubly-linked list preference. Note this is an O(1) operation. Once we have the doubly-linked list we can simply remove the last item from the list as it is the least recently used

with the least preference. Remember to also remove this item from the expiry min-heap, preference min-heap, and the hash table of key to wrapped item. The runtime in this scenario is same as before:

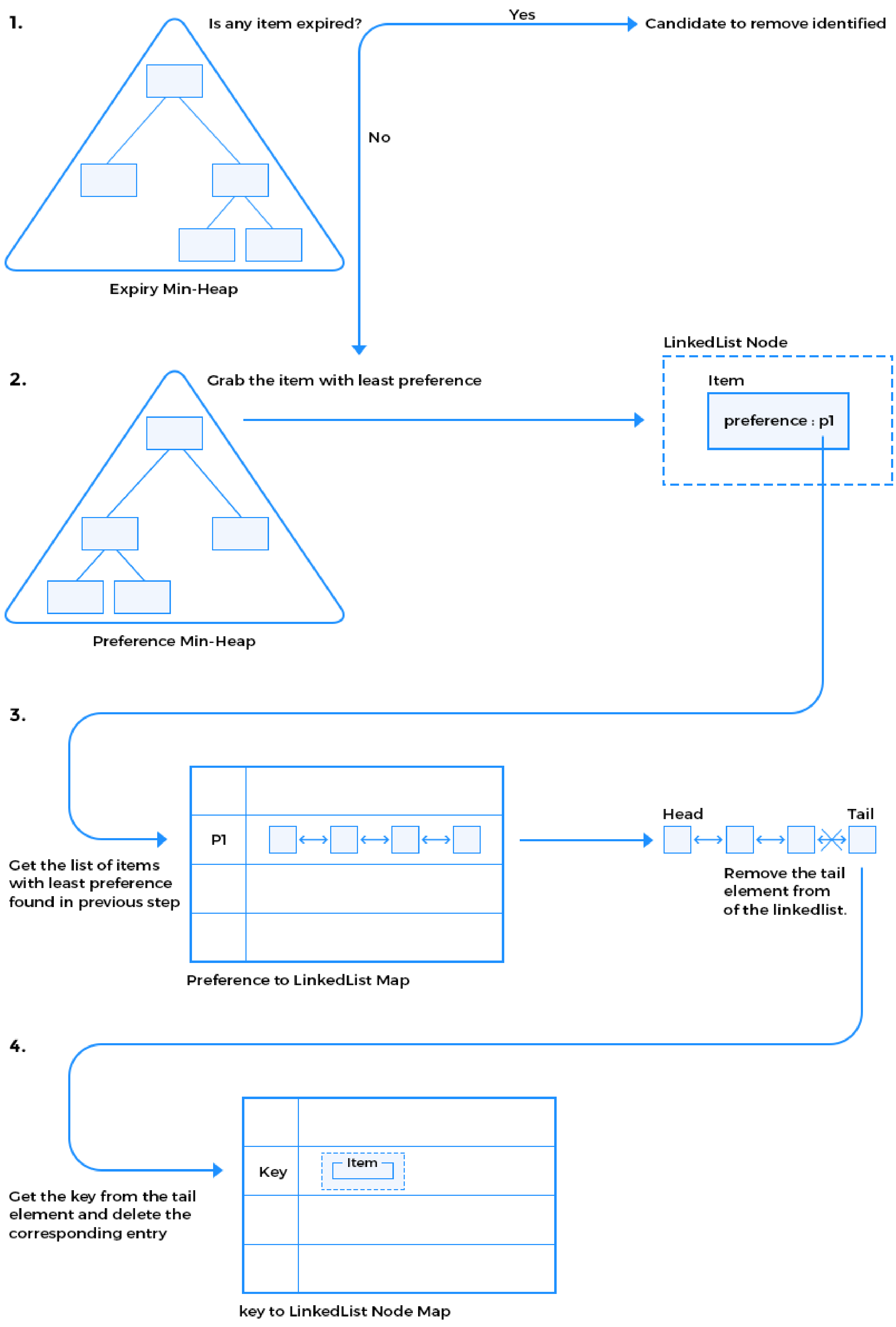
$$O(lgn) + O(lgn) + O(1) + O(1) = O(lgn)$$

The worst-case runtime complexity of the `evictItem()` method will be the worst-case complexity of the two scenarios, which is:

$$O(lgn)\ \ or\ \ O(lgn)\ \ is\ \ still\ \ O(lgn)$$

A pictorial representation of the `evictItem()` logic when no expired item is found shown below:

evictItem() with no expiredItem


1.
Is any item expired?
Yes
Candidate to remove identified
No
Expiry Min-Heap
2.
Grab the item with least preference
LinkedList Node
Item
preference : p1
Preference Min-Heap
3.
P1
Get the list of items with least preference found in previous step
Head
Tail
Remove the tail element from of the linkedlist.
Preference to LinkedList Map
4.
Key
Item
Get the key from the tail element and delete the corresponding entry
key to LinkedList Node Map

set ( )

set ( )

Consider the case when the cache has capacity to accept a new item. The `set()` method will require us to add the new element to the two min-heaps, and each operation will take O(lgN). Inserting into the key-to-wrapped-item hashtable is a O(1) operation. For the preference-to-doubly-linked-list hashtable we'll simply add at the end of the list with the same preference, which would turn out to be another O(1) operation.

The case where the cache is already full we'll first execute an evict operation which will take us *O(lgn)* as we just determined and then execute the sequence of adding the new item to all of the four data-structures. all in all a O(lgN) + O(lgN) = O(lgN) operation.

$$O(lgN) + O(lgN) = O(lgN)$$

This problem is an excellent example to learn how we can compose various data-structures to construct even more complex data-structures with efficient operations.

The explanation above is language agnostic but a solution in Java is presented below.

main.java

PriorityExpiryCache.java

ListNode.java

Item.java

DoublyLinkedList.java

```
import java.util.*;

public class PriorityExpiryCache {

    int maxSize;
    int currSize;

    PriorityQueue<ListNode<Item>> pqByExpiryTime = new PriorityQueue<>((a, b) -> a.data.expi
    PriorityQueue<ListNode<Item>> pqByPreference = new PriorityQueue<>((a, b) -> a.data.pref
    HashMap<Integer, DoublyLinkedList<Item>> preferenceToList = new HashMap<>();
    HashMap<Integer, DoublyLinkedList<Item>> preferrenceToList = new HashMap<>();
    HashMap<String, ListNode<Item>> keyToItemNode = new HashMap<>();


    public PriorityExpiryCache(int maxSize) {
        this.maxSize = maxSize;
        this.currSize = 0;
        LinkedList<Item> l = new LinkedList<>();
    }

    public Set<String> getKeys() {
        return keyToItemNode.keySet();
    }


    /**
     * 1. Remove all expired items first
     * 2. If none are expired, evict the ones with lowest preference
     * 3. If there's a tie on items with least preference, evict the ones
     * which are least recently used.
     */
    public void evictItem(int currentTime) {

        if (currSize == 0) return;

        currSize--;

        // Check expired items first
        if (pqByExpiryTime.peek().data.expireAfter < currentTime) {

            ListNode<Item> node = pqByExpiryTime.poll();
            Item item = node.data;

            DoublyLinkedList<Item> dList = preferrenceToList.get(item.preference);
            dList.removeNode(node);

            // Remove from hashmap too
            if (dList.size() == 0) {
                preferrenceToList.remove(item.preference);
            }

            // Remove from hashmap
            keyToItemNode.remove(item.key);

            // Remove from preference queue too
            pqByPreference.remove(item.preference);

            return;
        }

        // Next check if preference items are to be removed
        int preference = pqByPreference.poll().data.preference;

        DoublyLinkedList<Item> dList = preferrenceToList.get(preference);

        // Remove the end
        ListNode<Item> leastRecentlyUsedWithLeastPreference = dList.removeLast();
        keyToItemNode.remove(leastRecentlyUsedWithLeastPreference.data.key);

        // Remove from the expiry queue
        pqByExpiryTime.remove(leastRecentlyUsedWithLeastPreference);
        if (dList.size() == 0) {
            // Remove the dList too
            preferrenceToList.remove(dList);

        }
    }

    /**
     * Get the value of the key if the key exists in the cache and isn't expired.
     */
    public Item getItem(String key) {

        if (keyToItemNode.containsKey(key)) {
            ListNode<Item> node = keyToItemNode.get(key);
            Item itemToReturn = node.data;

            DoublyLinkedList<Item> dList = preferrenceToList.get(itemToReturn.preference);

            dList.removeNode(node);
            dList.addFront(itemToReturn);

            return itemToReturn;
        }

        return null;
    }

    /**
     * update or insert the value of the key with a preference value and expire time.
     * Set should never allow more items than maxItems to be in the cache. When evicting
     * we need to evict the lowest preference item(s) which are least recently used.
     */
    public void setItem(Item item, int currentTime) {

        if (currSize == maxSize) {
            evictItem(currentTime);
        }

        // Get the linkedlist for the preference queue
        DoublyLinkedList<Item> dlist = null;
        if (preferrenceToList.containsKey(item.preference)) {
            dlist = preferrenceToList.get(item.preference);
        } else {
            dlist = new DoublyLinkedList<>();
            preferrenceToList.put(item.preference, dlist);
        }

        ListNode<Item> node = dlist.addFront(item);
        keyToItemNode.put(item.key, node);

        // Update the expiry time pqueue
        pqByExpiryTime.add(node);

        // Update the preference pqueue
        pqByPreference.add(node);

        currSize++;
    }
}
```

In the above example, we print the keys left in the cache after each `evictItem()` operation. As you can see the key **C** is retained in the cache till the end. We perform a `get()` on the **C** key. that makes it the most recently used item with the same preferences. The first `evictItem()` operation removes key **B** as it is expired. The second `evictItem()` operation removes key **D** as it has the least preference out of the remaining items. The third `evictItem()` removes **A**. Note that we could also have removed **E** here as both **A** and **E** are never accessed after being added. In our implementation we always add a new element to the front of the doubly-linked list, which causes **A** to be removed. Finally, on the fourth `evictItem()` call the key **E** is removed.