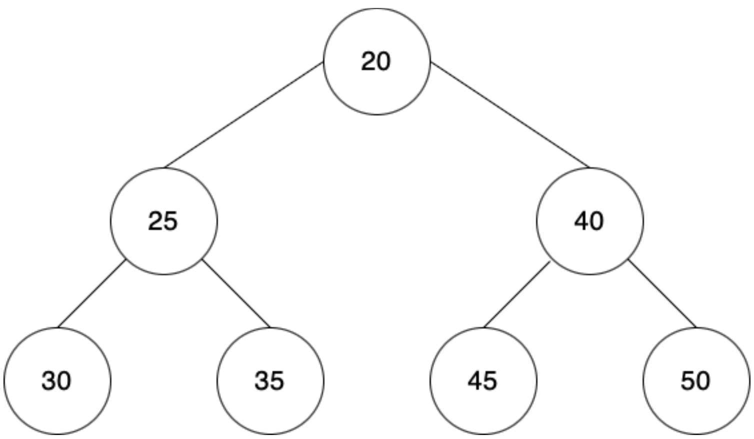
Homework – 3
Jay Kalyanbhai Savani
CWID – 20009207

EXERCISE | 5.7.11

(a) Is there a heap T storing seven distinct elements such that a preorder traversal of T yields the elements of T in sorted order? How about an inorder traversal? How about a postorder traversal?

Feedback?

Ans: Let us consider a Min–Heap Tree having 7 distinct elements, 20, 25, 30, 35, 40, 45, 50.



20-25-30-35-40-45-50 is the preorder traversal in this case.
A preorder traversal's order is Root, Left Child, Right Child.

30, 25, 35, 20, 45, 40, and 50 are the in-order traversals in this case.
For an inorder traversal, the order is Left Child, Root, Right Child.
The parent is always listed after the left child in an inorder traversal. In a tree, the parent is always at the bottom. Therefore, a heap's inorder traversal is not sorted.

Here is the Postorder Traversal: 30, 35, 25, 45, 50, 40,
In a postorder traversal, the order is Left Child, Right Child, Root.
The left child travels first in a postorder traversal, followed by the right child, and finally the parent. As a result, the Tree T components are not sorted by the postorder traversal.

EXERCISE 5.7.24

(a) Let T be a heap storing n keys. Give an efficient algorithm for reporting all the keys in T that are smaller than or equal to a given query key x (which is not necessarily in T). For example, given the heap of Figure 5.4.1 and query key $x = 7$, the algorithm should report 4, 5, 6, 7. Note that the keys do not need to be reported in sorted order. Ideally, your algorithm should run in $O(k)$ time, where k is the number of keys reported.

Feedback?

Ans: The root is the tree's minimal key. We consistently contrast the goal value with the root value. We pop out and report the key in the root if the query key is greater than the root key. When the root key is greater than the query key or when there are no more nodes in the tree, the loop finishes. The heap will self-reconstruct to ensure that the root has minimal value.

The best scenario in this situation is for each node to have just the correct children. The right child of the root should thus be a new root in the tree every time the key to the root pops after t, and this process requires O(1) time.

Given that k report numbers are assumed, the running time should be O(k).

Algorithm:

print_at max(key z, tree_node_n)
Input: key z, tree_node_n(!)
Output: The keys of all nodes of the subtree rooted at n with keys at most z.

```
class Node
{
	int data;
	Node left, right;

	public Node(int item)
	{
		data = item;
		left = right- null;
	}

	public class heap_sort_key
	{
		Node root;
		boolean isHeap (Node node, min_val, max val)
	}

	if (node=NULL)
		return true;

	else (n.get_Key()<=z)
	{
```

```
                    println(n.get_key();
                    print_at_max(z, n.getLeftChild();
                    print_at_max(z, n.getRightChild();
          }
}
```

EXERCISE 5.7.28

(a) In a ***discrete event simulation***, a physical system, such as a galaxy or solar system, is modeled as it changes over time based on simulated forces. The objects being modeled define events that are scheduled to occur in the future. Each event, e, is associated with a time, t_e, in the future. To move the simulation forward, the event, e, that has smallest time, t_e, in the future needs to be processed. To process such an event, e is removed from the set of pending events, and a set of physical forces are calculated for this event, which can optionally create a constant number of new events for the future, each with its own event time. Describe a way to support event processing in a discrete event simulation in $O(\log\ n)$ time, where n is the number of events in the system.

Feedback?

Ans: Here, we have a sequence of occurrences at times ' t_e ' in the future. Each step requires that we extract the event with the lowest ' t_e ' value, work with that event, and then add further events in a limited number that are related to our extracted event.

We may do this by identifying the smallest ' t_e ' event in each step, which will be an O(n) process.

Using priority queues will be the best course of action in this case. Our event timings are put into a minimum-priority queue. In a min-priority queue, the minimal element is placed at the front of the line. In this manner, each step just requires that we retrieve the first entry in the queue.

Complexity and Runtime of the solution:

In a priority queue:

1. Insertion = log(n)
   So, when a new element is added to the queue, it will take log(n) time to insert it.

2. Accessing the front element = O(1).
   It just needs to access the first element of the queue which is O(1).

So, it maintains a min-priority queue. In each step, access the first element of the queue. Add the new elements to the priority queue.

Minimum priority queues can be easily implemented using a minimum heap.

Pseudo Code:

```
#Function called by min-heap function to build the heap.

def min_heapify (Arr[ ] , i, N)

        left  = 2*i;
        right = 2*i+1;
        smallest;
        if left <= N and Arr[left] < Arr[ i ]
             smallest = left
        else
            smallest = i
        END If

        if right <= N and Arr[right] < Arr[smallest]
            smallest = right
        END If
        if smallest != i

            swap (Arr[ i ], Arr[ smallest ])
            min_heapify (Arr, smallest, N)
        END If

End

#function to build min-heap

def min-heap(Arr[], N)

        i = N/2
        for i: N/2 to 1 in steps of 1
            min_heapify (Arr, i);

End

#function to extract minimum time

def min_extract( A[ ] )

return A [ 0 ]

END
```

```
#functions to insert new elements

def insertion (Arr[ ], value)

   length = length + 1
   Arr[ length ] = -1
   value_increase (Arr, length, val)
END

def value_increase(Arr [ ], length, val)

        if val < Arr[ i ]
              return
           END If
           Arr[ i ] = val;
           while i > 1 and Arr[ i/2 ] < Arr[ i ]

              swap|(Arr[ i/2 ], Arr[ i ]);
              i = i/2;
           END While

END
```

The runtime of this algorithm is O(log n).

EXERCISE | 6.7.13

(a) Dr. Wayne has a new way to do open addressing, where, for a key k, if the cell $h(k)$ is occupied, then he suggests trying $(h(k)+i \cdot f(k)) \bmod N$, for $i = 1, 2, 3, \ldots$, until finding an empty cell, where $f(k)$ is a random hash function returning values from 1 to $N - 1$. Explain what can go wrong with Dr. Wayne's scheme if N is not prime.

Feedback?

Ans: In Dr. Wayne's strategy of open addressing for a key k, if h(k) is occupied then try a search (h(k) + i * f(k)) mod N cell where i=1,2,3.... and f(k) returns a random number from 1 to N-1.

For example:

Let N = 10 and f(k) produce 5 each time, then according to these values, it always shows values h(k) + 0 or h(k) + 5 cells.

20 mod 5 = 0

40 mod 5 = 0

60 mod 5= 0

80 mod 5= 0

100 mod 5= 0

Despite the fact that 5 is a prime number, all the keys are multiples of 5, hence the mod will never be more than 0. Any value that is a multiple of a number will also result in this. This kind of distribution is undesirable since collisions will still occur even when there is still room in the bucket. Therefore, to enable the probing of all cells, we should choose 'N' as a prime integer (often huge numbers).

To counteract key patterning in a hash function's distribution of collisions, prime integers are utilized. When f(k) never evaluates to zero, which is feasible by choosing prime integers, it will be a decent hash function, according to the question "f(k) is a random hash function." And for certain prime numbers q N, q - (k mod q) is a popular option for f(k).

Applications

EXERCISE | 6.7.17

(a) Suppose you are working in the information technology department for a large hospital. The people working at the front office are complaining that the software to discharge patients is taking too long to run. Indeed, on most days around noon there are long lines of people waiting to leave the hospital because of this slow software. After you ask if there are similar long lines of people waiting to be admitted to the hospital, you learn that the admissions process is quite fast in comparison. After studying the software for admissions and discharges, you notice that the set of patients currently admitted in the hospital is being maintained as a linked list, with new patients simply being added to the end of this list when they are admitted to the hospital. Describe a modification to this software that can allow both admissions and discharges to go fast. Characterize the running times of your solution for both admissions and discharges.

Feedback?

Ans: According to the question, the program uses a linked list to calculate the admission process of patients more quickly since adding a new patient to the list just requires adding them to the end of the list, which takes O(1) time. While releasing a patient takes longer since it must search through the whole list to discover that particular patient, which will take O(n) time.

We may utilize the "Hash Map" data structure, which maps keys to values, to resolve issues. The map data structure "m" is used, together with "k" for the key and "v" for the value that is mapped to the key. The following techniques may be used to solve the issue:

1. addPatient(k,v)
a. Use a hash map lookup to see whether the patient is already listed.
b. If the patient is not already present, add a patient number with the value v and the key k associated with it so that it may be added to the lookup table at the final index of the array.

2. dischargePatient(k):
a. Use a hash map lookup to see whether the patient is already listed.
b. Remove the value with a key that matches in k if the patient is present in the hash map. In this manner, the index key from the array will likewise be removed upon a patient's discharge.

3. Use the hash map lookup function searchPatient(k) to see whether the patient is on record.

Time Complexity:

The addPatient(k, v) and dischargePatient(k) operations can be performed in O(1) time, as each of the methods in the hash map lookup table takes O(1) time. In rare cases, if patients are hashed to the same key, a rehash operation must be performed. This operation takes O(n) time, but will only occur after n/2 operations and is based on the assumption of O(1).

EXERCISE | 6.7.25

(a) A popular tool for visualizing the themes in a speech is to draw a word cluster diagram, where the unique words from the speech are drawn in a group, with each word's size being in proportion to the number of times it is used in the speech. Given a speech containing n total words, describe an efficient method for counting the number of times each word is used in that speech. You may assume that you have a parser that returns the n words in a given speech as a sequence of character strings in $O(n)$ time. What is the running time of your method?

Feedback?

Ans: Cuckoo Hashing is an efficient method to count the frequency of each word used in a speech containing n total words. It uses two hash tables T0 and T1, each of size N, where N is greater than n. For each key k, the key can be stored in either T0 [h0(k)] or T1 [h1(k)].

All operations such as insertion, removal, and search can be performed in O(1) time in the worst case. If a collision occurs during insertion, the previous item is evicted and the new item is inserted. This process may cause a loop, which can be prevented by rehashing the keys. The words are stored as the keys in the hash table, and their frequency is stored as the value.

The total time to count the frequency of each word in the speech is O(n).