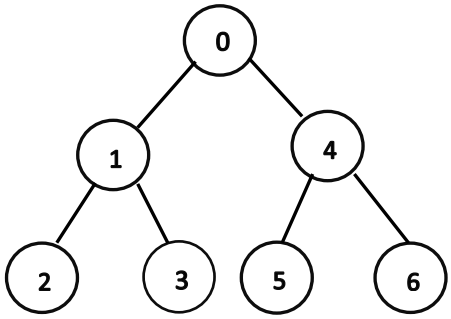
CS 600 HW 3
SHUCHI PARAGBHAI MEHTA
CWID: 20009083

- 5.7.11

  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?

  Lets consider a MIN-HEAP tree T, which has 7 distinct elements 0, 1, 2, 3, 4, 5, 6 And the T holds the elements in sorted order.

  Example:



  For the above tree, the Preorder traversal will be sequenced as Root, LeftChild, RightChild,
  i.e. 0, 1, 2, 3, 4, 5, 6
  And the Inorder traversal will be sequenced as LeftChild, Root and RightChild,
  i.e. 2, 1, 3, 0, 5, 4, 6

  As per MIN-Heap attribute, a min value element will always be placed at the root, and it can be removed or searched first. However, in Inorder Traversal LeftChild element travels first, which in MIN-Heap tree is always greater than its parent. Thus, determining the sorted order of the tree T components via Inorder Traversal is not possible.

  If we consider Postorder traversal, the sequence will be LeftChild, RightChild and Root,
  i.e. 2, 3, 1, 5, 6, 4, 0

  Now, in Postorder Traversal LeftChild traverse first and than rightChild which is always greater than its parent in MIN-Heap tree. Therefore, Postorder Traversal does not give sorted order for the tree T elements.

- 5.7.24

  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.

Algorithm keyFinder(T, x)

$root = t.root()$
$Algorithm\ keyFinderHelper(node, x)$

$if\ x < node.key()$
$return\ []$
$if\ x == node.key()$
$return\ [node.key()]$
$if\ x > node.key()$
$return\ [node.key()] + keyFinderHelper(node.left, x) +$
$keyFinderHelper(node.right, x)$

$return\ keyFinderHelper(root, x)$

This algorithm's runtime is $O(k)$ where k is the reported number of keys.

- 5.7.28

  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.

  One way to support event processing in a ***discrete event simulation***, in $O(log\ n)$ time is to use a priority queue. The priority queue can be implemented using a binary heap. Each event, e, is associated with a time, t, in the future.

  The binary heap is ordered by time, so that the event, e, with the smallest time, t, is at the top of the heap. To move the simulation forward, the event, e, at the top of the heap is removed and processed. Processing an event can optionally create a constant number of new events for the future, each with its own event time. The new events are inserted into the heap.

  For providing event processing support in a ***discrete event simulation***, an effective data structure known as a priority queue may be utilized. The event that requires the least amount of time to process can move to the front of a priority queue. This is achieved by ensuring that the queue is kept in the order of time, with the event that requires the least amount of time being placed at the top of the pile.
  The event that is currently at the top of the heap is extracted and processed to advance the simulation.

  This has the potential to generate a fixed number of brand-new events throughout the foreseeable future, each of which will have its own event time. The newly occurring events are added to the existing heap. This procedure will keep repeating itself until there are no more events stored in the heap.

- 6.7.13

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 $(h(k)+i \cdot f(k)) \mod N$, 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.

When using Dr. Wayne's open addressing approach for a key k, if $h(k)$ is already taken, try searching for $(h(k) + i * f(k)) \ mod\ N$ cell. $f(k)$ gives a random number between 1 and N-1 where i=1, 2, 3, etc.

For example:

If N = 10 and f(k) will consistently produce 5, then it will always display values for either h(k) + 0 or h(k) + 5 cells.

10 mod 5 = 0, 20 mod 5 = 0, 30 mod 5= 0, 40 mod 5= 0, 50 mod 5= 0

Even if 5 is a prime number, the mod will never be more than 0 as all the keys are multiples of 5. Same will happen for any value that is multiple of the number. This distribution is poor since collisions will still occur even when there is still room in the bucket. So, to enable the probing of all cells, we should choose 'N' as a prime number which is often a large number.

In the distribution of collisions of a hash function, prime numbers are used to neutralize the impact of patterns in the keys. As given f(k) is a random hash function, it can be good hash function when it never results in 0 because a good hash function is one that never evaluates to zero, which can be achieved by choosing prime numbers. And for some prime numbers $q < N$, a common choice for $f(k)$ is $q - (k\ mod\ q)$.

- 6.7.17

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.

As new patients are added to the end of the list, which takes O(1) time, the question claims that software quickly computes the admission procedure for the patient utilizing linked lists. However, releasing a patient from care takes longer since O(n) time is required to explore the whole list in search of the specific patient.

The "Hash Map" data structure, which maps keys to values (k, v), where k is the key and v is the value associated with that key, can be used to solve the issue. The following techniques are supported by the map data structure M:

put (k, v): Insert an item with key k and value v, at the last index of the array the patient can be inserted as keys, which can be added to the lookup table.

remove(k): An item with key equal to k will be removed from M. So, when a patient is removed, the index will also be removed from the lookup table.

So, the runtime complexity of all essential map methods will be $O(1)$, as both operations $put\ (k, v)$ and $remove(k)$ will be done in $O(1)$ runtime.

The limitation of this method is that collisions will result if the keys are not distinct integers in the range [0, N − 1]. Separate chaining, which saves all the objects that our hash function has mapped to the bucket $A[i]$ in a linked list, is used to get around this problem. The ratio of the number of entries in the hash table (n) to the table's capacity (N) is the load factor, which can be computed as n/N. If it is $O(1)$, hash table operations should complete in $O(1)$ time when collisions are addressed via separate chaining.

- 6.7.25

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?

Cuckoo Hashing is an effective method for calculating how many times each word appears in a speech with n total words.Each of the two lookup tables T_0 and T_1 in Cuckoo Hashing has a size of N, where N is greater than n. There are n elements on the map and two locations, T_0 [h_0(k)], T_1 [h_1(k)], in which an object can be placed with any key k.

In the worst case, all insertion(put), removal(remove), and search(get) operations are performed in $O(1)$ time. If a collision happens during the insertion procedure then remove the existing item from the cell and replace it with a new one. The removed item is then moved to a different table and placed there, where it may repeat the removal process with other items. However, this could lead to looping, which can be avoided by rehashing the keys in the table.

Both tables allow for the insertion of new words, which be stored as the keys and their frequencies with values in hash tables. It will take $O(n)$ time to count n words when they are used in speech.