COEN 12:

Data Structures

1. Data Structures: organization of data in memory or on a disk for efficiency
   * + 1. Efficiency can refer to either time or space
   1. (Common) Operations:
      1. Add/insert
      2. Delete/remove
      3. Retrieval (most commonly used)
         1. By name: searching
         2. By position or rank: indexing
   2. Efficiency
      1. Not necessarily measured in time
      2. Measured in number of steps; done in smart but lazy fashion
         1. n = number of items in a data structure
            1. Only a concern when n is a huge number, if small not significant
   3. Counting being lazy
      1. Let f(n) be a function
      2. To compute the big O complexity of f(n)
         1. Drop all terms except fastest growing term
         2. Drop coefficient of that remaining term
         3. O(\_) highest power or term growing fastest used
   4. Common operation classes
      1. O(1) Constant
      2. O(logn) logarithmic
      3. O(n) linear
      4. O(nlogn) linear logarithmic
      5. O(n^2) quadratic
      6. O(n^3) cubic
      7. O(n^k) polynomial
      8. O(2^n) exponential
      9. O(n!) factorial
      10. Independent loops not reliant on previous loops
      11. Dependent loops require previous loops changes
          1. Nested for loops stack creating O(n^2)
2. Arrays
   1. Arrays: a continuous sequence of elements in memory
      * 1. a[i] index operation and is O(1)
   2. Sequential search (linear search)
      * 1. O(n)
      1. Add/insert
         1. To add an element to the array, we will put it in slot n
            1. a[n] = x;
            2. n++;
            3. O(1)
      2. Remove/delete
         1. Requires two steps
            1. Find the element to remove O(n)
            2. Delete it physically from the array

Replace the deleted element with the last element

Shift elements to replace space created

* + - * 1. a[i] = a[n-1];
        2. n--;
  1. Improving searching an array
     1. Binary search
        1. Able to eliminate half of remaining possibilities at each step
           1. Worst case big-O runtime of O(logn)
        2. In order to apply this to an array it must be sorted in order
        3. O(logn)

Int lo, hi, mid;

Lo = 0;

Hi = n -1;

While( lo <= hi )

Mid = (lo + hi)/2;

If (x < a[mid])

Hi = mid -1;

Else if(x >a[mid])

Lo = mid +1;

Else

Return true;

Return false;

* + 1. Inserting/deleting values from an array (restriction: keep array sorted)
       1. Insertion
          1. We must insert the values in sorted order, so we cannot simply add it to the end of our array. Instead, we will have to make room for it by shifting all subsequent values to the right by one slot. In the worst case, the new value goes in the first slot
          2. Assume new value belongs at slot s
          3. O(n)

For (i = n; i >s; i--)

a[i] a[i – 1];

a[s] = x;

n++;

* + - 1. Deletion
         1. To delete a value from slots in a sorted array, we must shift all subsequent values down one slot (to the left)
         2. Assume the value to be deleted is in slot s
         3. O(n)

For (I = s +1; I < n; i++)

A[I -1] = a[i];

n--;

1. Abstract Data Type (ADT)
   * 1. An abstract data type: a data type in which the implementation is hidden from the other pairs of the code that use it (clients)
        1. Example of an ADT: FILE
           1. File is typically a structure that holds OS-dependent information
        2. NON-Example of ADT: strings in C
   1. Advantages of ADT:
      1. The main advantage of using an ADT is that the implementation can change without the clients being affected
      2. The client cannot accidentally mess up your implementation
      3. Enforce constraints by encapsulating your ADT behind a set of functions
   2. Example: ADT: set
      1. Set: is an unordered collection of elements which are distinct (no repetition)
      2. Since order is not relevant, we look up elements by name, not position
         1. Side note: implementation of set ADT in C using an array

Choice can be unsorted or sorted

Typedef struct set SET; (naming the set structure SET)

* 1. Worst Case big-O runtimes

|  |  |  |  |
| --- | --- | --- | --- |
| Set | Unsorted | Sorted | Hashtable |
| Find | O(n)  (Sequential/linear search) | O(logn) | O(1) / O(n) |
| Add | O(n) + O(1) = O(n) | O(logn) + O(n) = O(n) | O(1) / O(n) |
| Remove | O(n) + O(1) = O(n) | O(logn) + O(n) = O(n) | O(1) / O(n) |
| Min/Max | O(n) | O(1) | O(m) |

1. Hashing
   1. Hashing: Technique by which we map out each input value onto an array index, all of which are hopefully distinct
      1. The function that does like this mapping is called a hash function
      2. An array that is indexed using a hash function is called a hash table a[h(k)]
      3. Table sizes typical prime in hash
      4. Has the potential to provide inset, delete, and find in O(1)
   2. Collisions: Hash at the same position/location
      1. Want to avoid as many as possible
      2. No collisions = perfect hashing
      3. Prior knowledge of data in advance allows for the construction of a perfect hash function
   3. Collision Resolution
      1. If the table is not full, then we should find an empty space
      2. Can’t spend too much time trying to find an empty slot
      3. In open addressing
         1. Allows keys to go into slots that are not their original position
         2. Simplest open-addressing resolution strategy is probing
   4. Probing: Try alternative slots until we find one that is empty
      1. Simplest type of probing is linear probing
         1. h(k,i) = (h(k) + i) % m
            1. i is the probe number, makes it linear
            2. starts at zero for each key that we try to insert and then goes up by 1 each time until empty slot is found
            3. resets for each key
      2. We cannot simply check just the home position of a key because the key may have collided and been inserted else where when we see an empty slot, we can stop and say that our key is not present
         1. Worst case, all keys hash to the same location and hashing degenerates into sequential search
         2. Worst case big O runtime O(n)
      3. What can we expect in practice?
         1. Let’s assume that any key is equally likely to hash to any location in the table: simple uniform hashing
         2. Likely hood of a key hashing to a particular location i in a table of size m is 1/m
         3. If we assume that the table is alpha% full then the likely hood of the collision is alpha (the percentage as a fraction)
            1. If table is alpha% full, then alpha% of having a collision
            2. Alpha = n/m

N number of elements

M is size of the table

* + - 1. Load factor
         1. Expected number of tries to insert a new key into a table is 1/(1-alpha)
         2. Aplpha = number of tries/table size
    1. Linear probing leads to primary clustering
       1. Primary cluster: is the tendency of a collision resolution scheme to create long runs of filled locations near the hash locations of keys
    2. Quadratic Probing (h(k,i) = h(k) + i^2) % m
       1. The i^2 is the quadratic term
       2. One problem with quadratic probing is that unless you are careful with the size of your table, you are not guaranteed to find an empty slot should it exist
       3. Also quadratic probing leads to secondary clustering
          1. Secondary clustering: the tendency of a collision resolution scheme to create long runs of (nonconsecutive) filled locations away from the hash location of keys, such along the probe sequence
    3. Problem with both linear/quadratic probing is that the probe sequence is independent of the key. To overcome this we can use double hashing
       1. h(k,i) = (h1(k) + h2(k) + i) % m
          1. The second hash function makes the function double hashing
       2. Careful consideration of table size is still required for this function
    4. Deletion
       1. We need to leave a marker in the slot when we delete something
       2. To keep track of the state of each slot we can use two parallel arrays
       3. Data vs state
          1. Empty: never used, available for use
          2. Filled: holds valid data, not available for use
          3. Deleted: was once used, available

Can insert into deleted once empty found

Want keys close the their actual hash spot

* + 1. Hashing with deletion
       1. While probing, each slot we encounter will have one of three possible states
          1. Filled: compare our key against the key in the slot if it matches then you found it

If it doesn’t match then you continue probing

* + - * 1. Deleted: do not compare because the data is invalid

If this is the first deleted slot you’ve seen then remember it

Continue rembering

* + - * 1. Empty: stop probing

The key is not in the table

If you need to insert key then

If you previously saw a deleted slot, put it there

If you didn’t see a deleted slot, put it here

1. ADT: List: an ordered collected collection of items, which are not necessarily distinct
   * 1. Ordered does not imply sorted
     2. Ordered simply means that there is a first, second, ect.
     3. Know Difference between list/sets
        1. Order != sorted
        2. Sorted = order
     4. In contrast to a set where we usually look up by name, in a list we usually look up by position
     5. Two important types of lists
        1. Stack: a list that obeys last-in/first-out order (LIFO)
        2. Queue: a list that obeys first-in/first-out order (FIFO)
   1. Stacks: a list that obeys last-in/first-out order (LIFO)
      1. Obey LIFO order: the last item ordered is always the first item removed
         1. Examplees:
            1. History/back button in browsers
            2. Undo button in an editor
      2. Are used to remember information where we need to go back and remove the most recently remembered piece of information
      3. Operations:
         1. Add = push
         2. Remove = pop
         3. Most recent item = top
   2. Queues: a list that obeys first-in/first-out order (FIFO)
      1. A Queue obeys FIFO order: the first item added is the next one to be removed
         1. Example:
            1. Waiting in line
      2. A queue is used to remember things when we want to remove the oldest piece of information first
      3. Operations:
         1. Add = enqueue
         2. Remove = dequeue
         3. Oldest item = onqueue
   3. Implementation of a Stack using an array
      1. One can think of a stack as a list in which adding and removing both occur at the same end of the list
      2. It is easier and faster to add and remove from the rear of the array
   4. Implementation of a queue using an array
      1. One can think of a queue as a list in which adding and removing occur at opposite ends of the list