Problem Set #1

Josh Wolfe, Michael O'Connell and Anthony Tam

January 19, 2017

Question 1

Part a) Consider the case where the player loses the most

The worst case return value is: $-4.99 This occurs when the first value tested is equal to 263. The probability of this occuring is:

$$\frac{1}{len(A)+1}$$

We could also find the probability with a limit.

$$\lim_{n\to\infty} \frac{1}{n}$$

This limit shows that as the size of the list increases, the pribability of this occuring approaches 0%.

Part b) Consider the case where the player wins the most

The best case ruturn value cannot be determined for any input size; it depends on the length of the list. It can be determined by the following formula.

$$len(A) - 500$$

The probability of the best case for any specific sized list can be determined by:

$$\frac{len(A)}{len(A)+1}$$

For find the overall probability for any length list, we can use the following limit:

$$\lim_{n} \to \frac{n}{n+1}$$

This shows that as the list increases in size, the probability of not having 263s as value appraoches 1.

Part c) Consider the average case, what is the expected value of the winnings of a player

Let x represent the size of the list. Let E(x) represent the expected number of wins with a list of size x
The sequence of wins will then look like the following

$$E(x) = \frac{i}{i+1} + \frac{i}{i+1} \cdot \frac{i+1}{i+2} + \frac{i}{i+1} \cdot \frac{i+1}{i+2} \cdot \frac{i+2}{i+3} ...$$

This simplifies to

$$E(x) = \frac{i}{i+1} + \frac{i}{i+2} + \frac{i}{i+3} ...$$

In this case, the denominator is always the total number of possibilites for the last element of the list, so we can replace the denominator with x+1 (Number of possible outcoves for the length of the list)

$$E(x) = \frac{i}{x+1} + \frac{i}{x+1} + \frac{i}{x+1} ...$$

Turning this into a series, we get

$$E(x) = \sum_{i=263}^{x} \frac{i}{x+1}$$

Since $x + 1$ can be seen as a constant in the sequence, it can be removed from the series block

$$E(x) = \frac{1}{x+1} \cdot \sum_{i=263}^{x} i$$

Now we have a simple series which can be reduced to the following form

$$E(x) = \frac{1}{x+1} \cdot \frac{x^2 + x - 68906}{2}$$

$$E(x) = \frac{x^2 + x - 68906}{2(x+1)}$$

Part d) Suppose that you are the owner of the casino and that you want to determine a length of the input list A so that the expected winnings of a player is between 1.01 and 0.99 dollars

For this to occur, the player would need to win between 399 and 401 times. We can plug this into the formula for expected value.

$$399 = \frac{x^2 + x - 68906}{2(x+1)}$$

Computing this statement will result in x≈877
The same also must be done for 401 wins

$$401 = \frac{x^2 + x - 68906}{2(x+1)}$$

Which results in a solution of x≈880
This means the length of the list should be between 887 to 880 to have an outcome of $-1.01 and $-0.99

Question 2

Part a) Given the index i of an element in the array, what are the indices of the left child, the middle child, the right child, and the parent of the element?

The left element can be found by:

$$(3 \cdot index) + 1$$

The middle element can be found by:

$$(3 \cdot index) + 2$$

The right element can be found by:

$$(3 \cdot index) + 3$$

The parent element can be found by:

$$floor(\frac{index - 1}{3})$$

Part b) How do EXTRACT-MAX and INSERT work for a ternary max-heap?

Extract Max:
Very similar to a binary heap, the only difference being when checking for the largest child object, we need to check all 3 children instead of 2.

Insert:
There are no differences, simply keep checking the parent element and bubble up.

Part c) Write the pseudo-code of a recursive implementation of IS-TERNARY-MAX-HEAP, explain its correctness, and its asymptotic upper-bound

Pseudo-Code

```
def IS-TERNARY-MAX-HEAP(A, i = 0):
    '''
        Where right_child, middle_child, and left_child
        are the formulas from Part a)
    '''
    if left_child > len(A): # No left child, single node
        return True
    elif right_child <= len(A):
        if A[i] > A[right_child] and A[i] > A[middle_child] and A[i] > A[left_child]:
            return IS-TERNARY-MAX-HEAP(A, right_child)
                   and IS-TERNARY-MAX-HEAP(A, middle_child)
                   and IS-TERNARY-MAX-HEAP(A, left_child)
        return False
    elif middle_child <= len(A):
        if A[i] > A[middle_child] and A[i] > A[left_child]:
            return IS-TERNARY-MAX-HEAP(A, middle_child)
                   and IS-TERNARY-MAX-HEAP(A, left_child)
        return False
    elif left_child <= len(A) and A[i] > A[left_child]:
        return IS-TERNARY-MAX-HEAP(A, left_child)
     return False
```

Correctness

- If there is no child to the node, a single node is always a valid heap, so return true.
- If we have a right child, check if all 3 children are less then the parent value. If so, return the result of the smaller heaps where the child is the parent. Otherwise return false.
- If we have no right child, do the same as above for the middle and left child.
- If we have no middle child, so the same as above for only the left child.

Asymptotic Upper-Bound:

- The function will check every node in the tree to ensure it is in a valid position. $\therefore$ O(n)

Part d) Write the pseudo-code of a iterative implementation of IS-TERNARY-MAX-HEAP, explain its correctness, and its asymptotic upper-bound

Pseudo-Code

```
def IS-TERNARY-MAX-HEAP(A):
    '''
        Where right_child, middle_child, and left_child
        are the formulas from Part a)
    '''
    foreach (i in A):
        if right_child <= len(A):
            if not (i > A[right_child] and i > A[middle_child] and i > A[left_child]):
                return False
        elif middle_child <= len(A):
            if not (i > A[middle_child] and i > A[left_child]):
                return False
        elif left_child <= len(A) and i <= A[left_child]:
            return False
```

Correctness

- If we have a right child, check if all 3 children are less then the parent value. If they are not, return false.
- If we have no right child, do the same as above for the middle and left child.
- If we have no middle child, so the same as above for only the left child.

Asymptotic Upper-Bound

- The function uses a foreach to cycle every element in the list. $\therefore O(n)$

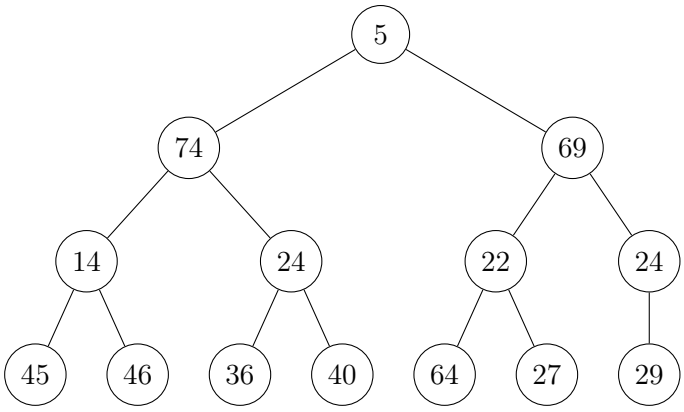
Question 3

The data structure that would satisfy the given properties is a combination of a min heap and a max heap. From here on out I will refer to this ADT as a min max heap.

What a min max heap looks like

The min max heap will have alternating min and max heaps where the minimum element is the root and the maximum is one of it's direct children.

E.g.



Here are the rules it must follow:

- For any element on a min (even numbered) level it must be smaller (or equal to) than all it's children.
- For any element on a max (odd numbered) level it must be larger than all it's children.
- Root is the smallest element.
- One of the roots direct children is the maximum.

Operations

Description

- **Insert**

  To insert an element you insert it in the most bottom right most node (inserting on a new row if the previous row is full). After inserting you check if it's greater than all nodes it inherits from that are in even rows and check if it's less than all nodes it inherits from that are in odd rows. If the checks fail swap it with it's parent and repeat.

- **Extract Minimum**

  Remove the root node, which by definition is the min, as well as the bottom right most node. Insert that node where the root used to be. Then swap it with the min of it's grand-children (or children if the grandchildren don't exist). Then repeat the same logic treating the swapped node as the root.

- **Extract Maximum**

  First you find the maximum between the two direct children of the root then apply similar logic to extracting the minimum however here you check for the max rather than the min starting with the previous max.

- **Max and Min**

  Max and min are both fairly straightforward. To find the max you simply check the two children of the root and return the max. To find the min you simply return the root.

- **Init**

  Call a bubble down method on all of the nodes that aren't in the bottom row. The method checks it's children and makes sure it satisfies the min max heap rules. If it doesn't it swaps and recurses.

Complexity

- **Insert**

  Insert has a time complexity of $log(n)$ since at most it has to bubble something up from the the lowest row. Since each row is twice the size of the previous row i.e. for the n^{th} column it would have 2^n items. This means that the height of the heap is $log(n)$ which is the maximum number of bubble ups.

- **Extract Min and Extract Max**

  Extract min has $log(n)$ complexity for similar reasons to insert. When the top element is removed it is replaced by the bottom most element. This is at most $log(n)$ swaps. For similar reasoning as insert. The max hight is $log(n)$ since it's a binary tree. Extract max works almost identically to extract min except it first needs to find which of the root children is max so it's $log(n) + 1$ operations. Which in asymptotic is the same as $log(n)$.

- **Min and Max**

  Min runs in constant time as only one operation is needed (checking the root node). Max also runs in constant time as you only need to compare the two children of the root. Neither of these operations grow with the size of the list.

- **Init**

  We can assume that half of the sub heaps are valid since they will be on the lowest level. These heaps will be valid since they have no children. So the lowest level would have $\frac{n}{2}$ which we get for free. The next level up would have $\frac{n}{4}$ nodes which have at most one swap. The level above would have $\frac{n}{8}$ nodes with at most 2 swaps each. This would continue all the way up to the root which would have at most $log(n)$ swaps.

As discussed in lecture.

$$\begin{aligned} T(n) &= 1 \cdot \frac{n}{2} + 2 \cdot \frac{n}{4} + 3\frac{n}{8} + \dots \\ &= \sum_{i=1}^{log(n)} i \cdot \frac{n}{2^{i+1}} \\ &\leq \sum_{i=1}^{y} i \cdot \frac{n}{2^{i+1}} \\ &= n \cdot \sum_{i=6} \frac{i}{2^{i+1}} \\ &\leq n \cdot 1 \\ &= O(n) \end{aligned}$$

Question 4

Sorted, Ascending

Build Version	List Length	Switch Count
V1	1000	8977
V2	1000	992
V1	5000	56809
V2	5000	4993
V1	10000	123617
V2	10000	9992
V1	50000	734465
V2	50000	49991
V1	100000	1568929
V2	100000	99990

This demonstrates the worst case runtime for both of the build heap methods, as a sorted, ascending, list requires the most swaps to achieve the max-heap property.

Sorted, Decending

Build Version	List Length	Switch Count
V1	1000	0
V2	1000	0
V1	5000	0
V2	5000	0
V1	10000	0
V2	10000	0
V1	50000	0
V2	50000	0
V1	100000	0
V2	100000	0

This demonstrates the best case runtime for both of the build heap methods, as a sorted, decending, list requires no swaps, as it alredy satisfies the max-heap property.

Unsorted, Random

Build Version	List Length	Switch Count
V1	1000	1283
V2	1000	709
V1	5000	6391
V2	5000	3769
V1	10000	13080
V2	10000	7433
V1	50000	64338
V2	50000	37039
V1	100000	127773
V2	100000	74409

This demonstrates the runtime comparrisons for a random selection of values, unsorted. The O(nlogn) vs O(n) runtime difference is clearly visable in this random sample.