# Final Homework

Matt Forbes

December 7, 2010

Problem 1

The cost of going from exit j to k is $C_j + C_{j+1} + C_{j+2} + \cdots + C_{k-1}$. I propose the data structure H such that $H_i = C_1 + C_2 + \cdots + C_{i-1}$. To calculate the cost exit j to k using H, it would simply be $H_k - H_j$. This expression expands to $(C_1 + C_2 + \cdots + C_{j-1}) - (C_1 + C_2 + \cdots + C_{k-1})$, which simplifies to $C_j + C_{j+1} + \cdots + C_{k-1}$. Showing that $H_k - H_j$ is equivalent to the cost we calculated for exit j to k. Given that H is already calculated, this computation is a simple subtraction, $O(1)$.

Generating this data structure is very easy and would take $O(n)$ time and holds n elements. Each element H_i is equal to $C_i + H_{i-1}$ which lends itself easily to an accumlating loop from 1 to n.

```
H[1] = C[1]
for i = 2...n:
  H[i] = C[i] + H[i-1]
```

Problem 2
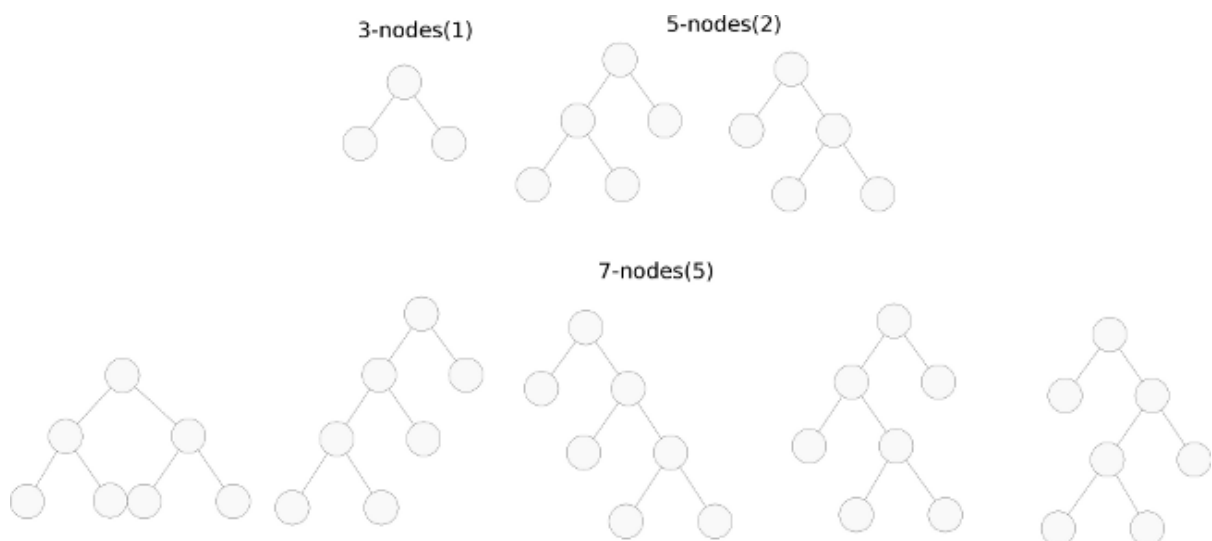
a) $B_3 = 1, B_5 = 2, B_7 = 5$.

b) You can't construct a full binary tree with an even number of nodes. Every node always has zero or two child nodes, meaning everytime the tree grows, it must grow by a multiple of two nodes. So starting with the root, and growing n times, the total number of nodes will always be of the form $1 + 2n$, which is odd.

c)

$$
B_n = \begin{cases} 1 & n \leq 3 \\ 2 & n = 5 \\ 5 & n = 7 \\ 2(B_{n-1} + B_{n-2}) & n > 7 \end{cases}
$$

Problem 3

```
structure weirdqueue {
    pushstack (stack pointer)
    popstack (stack pointer)
}

def enqueue(Q, elt):
    Q.pushstack.push(elt)

def dequeue(Q):
    if Q.pushstack and Q.popstack are empty:
        error underflow
    if Q.popstack is empty:
        while Q.pushstack is not empty:
            Q.popstack.push( Q.pushstack.pop() )
        swap Q.popstack pointer with Q.pushstack
    return Q.popstack.pop()
```

a) Under the assumption that the 'popstack' is empty, we would have to pop each element off the 'pushstack' and then push that on to the 'popstack.' By doing this we are reversing the order, guaranteeing that we get the first queued item, but means we are also doing work proportional to the size of the structure, $O(n)$.

b) In practice, we could not possibly have to do this mass popping and pushing to reorient the structure every dequeue. This means that we will have a much faster amortized analysis of the running time. If we follow the lifetime of one element in the structure, there are only about 4 operations associated with it. We initially push it on to the 'pushstack' and then at some later time we will transfer it to the 'popstack' and finally pop it one more time when it is removed. So there can never be more than 4 operations per element lifecycle. Using amortized analysis, we can see that n insertions

could never be worse than about $4n$ stack operations. Therefore, the average cost per enqueue/dequeue operation is $\frac{4n}{n} = 4$, which is $O(1)$.