Zijian Gao
26319560
2/11/2016
CS311 HW1

6-2)

a. Modify the binary heap array rule to a d-ary heap rule as follows:

```
Parent(i)
  return [i/d]
Child(i, j) // j'th child(0 based) of the i'th element
  return d * i + j
Root of tree is at A[1]
```

This is correct because it is simply a generalization of the binary heap rule, except instead of only a left and right child, there can be d children, so for example the 4 children of the root A[1] for a 4-ary heap would be stored at A[4..7], A[2..3] are parents for the children A[8…11] and A[12…15], and this can be proven to be correct because Parent(Child(i, j)) for any i, j from 1 to d, and d : [d*i+j]/d = i + j/d, and since j is less than d, the division operation would truncate so Parent(Child(i, j)) = i.

b. the height of a d-ary heap of n elements is $ceil(\log_d((d-1)n+1)-1)$, this is because at each height h, there are d^h possible elements with lower bound at $\sum_{h=0}^{h-1} d^h = 1 + d + d^2 + \ldots + d^{h-1} = \frac{d^h-1}{d-1}$ and maximum bound $\sum_{h=0}^{h} d^h = 1 + d + d^2 + \ldots + d^h = \frac{d^{h+1}-1}{d-1}$ possible elements, solved using a summation calculator. n must fall between this range.

Solving the inequality:

$$\frac{d^h-1}{d-1} < n \le \frac{d^{h+1}-1}{d-1}$$

$$d^h < (d-1)n+1 \le d^{h+1}$$

$$h < \log_d((d-1)n+1) \le h+1$$

$$\log_d((d-1)n+1)-1 \le h$$

Since this is less than or equal to h, take the ceiling of the equation to yield the height.

c.

```
d-ary-extract-max(A[])
if (A.heap-size) < 1
  error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
D-Ary-Max-Heapify(A, 1, d)
return max
```

```
D-Ary-Max-Heapify(A[], i, d):
largest = i
for child from 0 to d - 1 do
   if (Child(i, child) ≤ A.heap-size AND A[Child(i, child)] > A[largest])
      largest = Child(i, child)
if (largest != i)
   temp = A[i]
   A[i] = A[largest]
   A[largest] = temp
   D-Ary-Max-Heapify(A, largest, d)
```

The runtime of d-ary-extract-max is clearly constant time plus the time it takes to max-heapify the d-ary max-heap, as it simply reads the value from the root and swaps the minimum value to the root to bubble-down the heap.

Runtime analysis of D-Ary-Max-Heapify shows that for each node i, the method runs d times to find the largest child, then the swap operation is done in constant time if the largest element is found at a child node. This method recurses at most the depth of the tree, at which point it would stop finding larger child nodes and return. Since the depth of the tree is $ceil(\log_d((d-1)n+1)-1) = \theta(\log_d(n))$ in terms of runtime since d is a constant multiple in the logarithm, it can be disregarded in the analysis. The runtime of max-heapify is thus $\theta(d\log_d(n))$, and the runtime of d-ary-extract-max is the same since it only adds constant time operations.