COMP 775 - Advanced Design and Analysis of Algorithms Assignment 3

Jordan Sturtz

3-08-2023

Problem 6-2

A d-ary heap is like a binary heap, but (with one possible exception) non-leaf nodes have d children instead of 2 children

1. How would you represent a d-ary heap in an array?

   Answer:

   Like the binary heap, the d-ary heap would be stored in a contiguous array, such that the highest priority element comes first, starting from index 1. But the relationship betweeen parent and child is more complicated in this contiguous array. Specifically, we would like an equation that relates the index of a parent node to its children in terms of d.

   To get this equation, consider an arbitrary non-root node n in an d-ary tree with height, h. One way to think about this problem is to think about how we might count the number of steps until we reach either the beginnning or ending index of n's children. Consider that we traverse a d-ary by visiting first the right-siblings of n and then all the children of the left-siblings of n. So let l be the number of left siblings of n, and let r be the number of right siblings of n. A general formula to compute the rightmost child of n is therefore the following:

$$RIGHT(n) = n + (l+1)d + r$$

   Let us now express l and r in terms of n, d, and h.

   We note that n must equal the combined nodes in the previous layer plus l plus 1. Therefore,

$$n = l + \sum_{0}^{h-1} d^i + 1$$
$$l = n - \sum_{0}^{h-1} d^i - 1$$

   Similarly, the quantity r must equal the total sum of all the rows up to and including n minus n itself:

$$r = \sum_{0}^{h} d^i - n$$

   Substituting into our formula for the last child of n:

$$\begin{aligned}
RIGHT(n) &= n + (n - \sum_{0}^{h-1} d^i - 1 + 1)d + \sum_{0}^{h} d^i - n \\
&= n + nd - d\sum_{0}^{h-1} d^i + \sum_{0}^{h} d^i - n \\
&= nd - d\sum_{0}^{h-1} d^i + \sum_{0}^{h} d^i \\
&= nd - \sum_{1}^{h} d^i + \sum_{0}^{h} d^i \\
&= nd + \sum_{0}^{h} d^i - \sum_{1}^{h} d^i \\
&= nd + 1
\end{aligned}$$

Now that we have a general formula for RIGHT(n), we can easily compute LEFT(n) or any child in between by noting that LEFT(n) must equal RIGHT(n) - d + 1. We can also compute PARENT(n). In summary, we have these three formulae for computing indices relative to a node n:

- LEFT(n) = $nd - d + 2$
- RIGHT(n) = $nd + 1$
- PARENT(n) = $\lfloor \frac{n-1}{d} \rfloor$

2. What is the height of a d-ary heap of n elements in terms of n and d?

   Answer:

   First, we recognize that the size of the nodes of full d-ary trees follows a predictable pattern: $n = d^0 + d^1 + d^2 \cdots + d^h$, where h is the height, d is the number of children of an internal node, and h is the height of the tree. We therefore can solve for the relationship between n, d, and h in the following way:

$$\begin{aligned}
n &= d^0 + d^1 + d^2 \cdots + d^h \\
dn &= d^1 + d^2 + d^3 \cdots + d^h + d^{h+1} \\
dn - n &= d^{h+1} - 1 \\
dn - n + 1 &= d^{h+1} \\
h + 1 &= \log_d(dn - n + 1) \\
h &= \log_d(dn - n + 1) - 1
\end{aligned}$$

3. Give an efficient implementation EXTRACT-MAX in a d-ary max-heap. Analyze its running time in terms of d and n.

   Answer:

   The largest value in the max-heap is the root. So merely getting that value is constant time. However, the heap must be reconstructed when popping the root to preserve the heap property. So, to efficiently extract the MAX, we need an efficient solution to HEAPIFY.

   The algorithm to implement EXTRACT-MAX from a d-ary binary tree, A, is the following:

   The running time of EXTRACT_MAX is the same as MAX_HEAPIFY. To analyze this running time, we state a recurrence relation for the algorithm in the worst case:

Algorithm 1 EXTRACT_MAX(A)

```
1: max ← A[1]
2: A[1] ← A[A.heap_size]
3: MAX_HEAPIFY(A, 1)
4: return max
```

Algorithm 2 MAX_HEAPIFY(A, i)

```
1: n ← A.heap_size
2: d ← A.d
3: leftchild ← $d(n-1)+2$
4: rightchild ← $nd+1$
5: index_of_largest ← i
6: ptr ← leftchild
7: while ptr ≤ n and ptr ≠ rightchild do
8:    if A[ptr] > A[max] then
9:       index_of_largest ← ptr
10:   end if
11:   ptr++
12: end while
13: if index_of_largest ≠ i then
14:   temp ← A[index_of_largest]
15:   A[index_of_largest] ← A[i]
16:   A[i] ← temp
17:   MAX_HEAPIFY(A, index_of_largest)
18: end if
```

$$T(n) = d + 1 + T(n/d)$$
$$T(1) = \Theta(1)$$

The base case for T(1) is justified because if we're at the leaf node of A, then lines 1-6 are constant time operations, and the loop from 7-11 would involve no extra steps since ptr would be less than or equal to n. In other words, there are only constant time operations in the base case.

The recursive case for T(n) is justified since MAX_HEAPIFY involves constant time operations plus a loop over d elements, and then a recursive call on only n/d elements, since every recursive call to MAX_HEAPIFY will reduce the size of n by the factor d.

To solve this recurrence relation, we can unroll T(n):

$$\begin{aligned} T(n) &= d + 1 + T(n/d) \\ &= 2(d+1) + T(n/d^2) \\ &= 3(d+1) + T(n/d^3) \\ &= \dots \\ &= kd + k + T(n/d^k) \end{aligned}$$

Setting the quantity n/d^k to 1 to solve for k and substituting yields:

$$T(n) = d \log_d n + \log_d n + T(1)$$

Therefore, T(n) is $O(d \log_d n)$.

4. Give an efficient implementation of INSERT in a d-ary max heap. Analyze its running time in terms of d and n.

   Answer:

   To INSERT a new element into our d-ary heap, we will increase the size of our array and then insert our element at the end of the array. After this, the algorithm for INSERT will be similar to MAX_HEAPIFY but in reverse to force the value to "percolate up" to the correct location in the heap to preserve the heap property.

Algorithm 3 INSERT(A, x)

```
A.heap_size++
n ← A.heap_size
d ← A.d
A[n] ← x
i ← n
parent ← $\lfloor \frac{i-1}{d} \rfloor$
parentval ← A[parent]
while $i > 1$ and parentval > x do
    A[i] ← parentval
    A[parent] ← x
    parent ← $\lfloor \frac{i-1}{d} \rfloor$
    parentval ← A[parent]
end while
```

The running time can be expressed as a recurrence relation:

$$\begin{aligned} T(n) &= \Theta(1) + T(n/d) \\ T(1) &= \Theta(1) \end{aligned}$$

It it assumed the time to increase the size of A is constant, which it is not in practice. This depends on the implementation of the array data structure.

To solve this recurrence relation, we can unroll T(n):

$$\begin{aligned} T(n) &= 1 + T(n/d) \\ &= 2 + T(n/d^2) \\ &= 3 + T(n/d^2) \\ &= \dots \\ &= k + T(n/d^k) \end{aligned}$$

Setting the quantity n/d^k to 1 to solve for k and substituting yields:

$$T(n) = \log_d n + T(1)$$

Therefore, T(n) is $\Theta(\log_d n)$.

5. Give an efficient implementation of INCREASE-KEY(A, i, k), which flags an error if $k < A[i]$, but otherwise sets A[i] = k and then updates the d-ary max-heap structure appropriately. Analyze its running time in terms of d and n.

   Answer:

   If the new value, k, is not larger than A[PARENT(i)], then replacing A[i] with k is sufficient to preserve the heap property. If on the other hand the value, k, is larger than its parent, then merely replacing A[i] with k will break the heap property. But, if k is larger than its parent, then setting A[i] = k and swapping A[i] with A[PARENT(i)] will restore the heap property between i and its parent. We must do this recursively since we do not know if k is also larger than the parent of the parent of i. Moreover, since we are setting a new key k, that key could already exist in our d-ary heap. So, we should make sure when we recursively call INCREASE-KEY, that we pass k+1 to ensure we do not set the same key twice. So, the algorithm is as follows:

Algorithm 4 INCREASE_KEY(A, i, k)

```
if $k < A[i]$ then
    raise Exception
end if
parent ← $\lfloor \frac{n-1}{d} \rfloor$
parentval ← A[parent]
if k > parentval then
    A[i] ← parentval
    INCREASE_KEY(A, parent, k+1)
end if
```

The running time can be expressed as a recurrence relation:

$$\begin{aligned} T(n) &= \Theta(1) + T(n/d) \\ T(1) &= \Theta(1) \end{aligned}$$

To solve this recurrence relation, we can unroll T(n):

$$\begin{aligned} T(n) &= 1 + T(n/d) \\ &= 2 + T(n/d^2) \\ &= 3 + T(n/d^2) \\ &= \dots \\ &= k + T(n/d^k) \end{aligned}$$

Setting the quantity n/d^k to 1 to solve for k and substituting yields:

$$T(n) = \log_d n + T(1)$$

Therefore, T(n) is $\Theta(\log_d n)$.

Problem 6-3

An m x n Young tableau is an m x n matrix such that the entries of each row are in sorted order from left to right and the entries of each column are in sorted order from top to bottom. Some of the entries of a Young tableau may be ∞, which we treat as nonexistent elements. Thus, a Young tableau can be used to hold $r \leq mn$ finite numbers.

1. Draw a 4 x 4 Young tableau containing the elements { 9, 16, 3, 2, 4, 8, 5, 14, 12 }.

$$\begin{bmatrix} 2 & 3 & 4 & 5 \\ 8 & 9 & 12 & 14 \\ 16 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}$$

2. Argue that an m x n Young tableau Y is empty if $Y[1,1] = \infty$. Argue that Y is full (contains m n elements) if $Y[m,n] < \infty$.

   Answer:

   If the first element is ∞, then any element to the right of that element is also ∞ by definition. Similarly, by definition any element beneath the first element is also ∞. By recursive reasoning, every element must be ∞ since the first row and first column are all ∞. So, the array must be empty.

   If $Y[m,n] < \infty$, then any value to the left of $Y[m,n]$ must also be less than infinity and any value above $Y[m,n]$ must also be less than infinity. By recursive reasoning, therefore, all values are less than infinity, which by definition is what it means for the array to be full.

3. Give an algorithm to implement EXTRACT-MIN on a nonempty m x n Young tableau that runs in O(m + n) time. Your algorithm shoulld use a recursive subroutine that solves an m x n problem by recursively solving either an (m - 1) x n or an m x (n - 1) subproblem. (Hint: Think about MAX-HEAPIFY). Define T(p), where p = m + n Young Tableau. Give and solve a recurrence for T(p) that yields the O(m + n) time bound.

   Answer:

   To extract the min efficiently, we would like to replace the minimum with the next smallest value. This way, we preserve the properties of the Young tableau. To find the next smallest value of n, we must compare n to its neighbor to its right and beneath it. If the smaller is beneath n, then we can EXTRACT_MIN on the matrix beneath n, and if the smaller is to the right of n, then we can EXTRACT_MIN on the right submatrix. If we've successfully called EXTRACT_MIN on one of these submatrices, then that submatrix is itself ordered and its smallest element removed. Once we have this smallest element, we can then replace absolute min with that smallest element. This recursion stops only if and only if there is no element beneath or to the right of n (i.e. we've hit the corner). The recursive algorithm is below:

Algorithm 5 EXTRACT_MIN(T, m=1, n=1)

```
1: min ← T(m, n)
2: if m < T.numrows and (n == T.numcols or T(m, n+1) < T(m+1, n)) then
3:     T(m, n) ← EXTRACT_MIN(T, m+1, n)
4: else if n < T.numcols and (m == T.numrows or T(m+1, n) < T(m, n+1)) then
5:     T(m, n) ← EXTRACT_MIN(T, m, n+1)
6: end if
7: return min
```

The running time can be expressed as a recurrence relation:

$$\begin{aligned} T(p) &= T(m+n) \\ T(m+n) &= \Theta(1) + T(m+n-1) \\ T(1) &= \Theta(1) \end{aligned}$$

To solve this recurrence relation, we can unroll T(m+n):

$$\begin{aligned}T(m+n) &= 1 + T(m+n-1)\\ &= 2 + T(m+n-2)\\ &= 3 + T(m+n-3)\\ &= \dots\\ &= k + T(m+n-k)\end{aligned}$$

Setting the quantity $m + n - k$ to 1 to solve for k and substituting yields:

$$T(n) = m + n - 1 + T(1)$$

Therefore, $T(p) = T(m + n) = O(m + n)$

4. Show how to insert a new element into a nonfull m x n Young tableau in O(m + n) time.

   Answer:

   To insert into a *nonfull* Young tableau, we will insert the element at the last position, then check recursively if we need to swap that value with the larger of its neighbors. The algorithm is below:

Algorithm 6 INSERT(A, x, m=A.numrows, n=A.numcols)

```
A[m][n] ← x
if m > 1 and A[m-1][n] > x and A[m-1][n] > A[m][n-1] then
    INSERT(A, x, m-1, n)
    A[m][n] ← A[m-1][n]
end if
if n > 1 and A[m][n-1] > x and A[m][n-1] > A[m-1][n] then
    INSERT(A, x, m, n-1)
    A[m][n] ← A[m][n-1]
end if
```

The following is the recurrence relation, stated in terms of m+n:

$$\begin{aligned}T(m+n) &= \Theta(1) + T(m+n-1)\\ T(1) &= \Theta(1)\end{aligned}$$

To solve this recurrence relation, we can unroll T(m+n):

$$\begin{aligned}T(m+n) &= 1 + T(m+n-1)\\ &= 2 + T(m+n-2)\\ &= 3 + T(m+n-3)\\ &= \dots\\ &= k + T(m+n-k)\end{aligned}$$

Setting the quantity $m + n - k$ to 1 to solve for k and substituting yields:

$$T(n) = m + n - 1 + T(1)$$

Therefore, $T(m + n) = O(m + n)$

Algorithm 7 TABLEAU_SORT(n)

```
A ← new Array()
for x in n do
    INSERT(A, x)
end for
return A
```

5. Using no other sorting methods as a subroutine, show how to use an n x n Young tableau to sort n^2 numbers in $O(n^3)$ time.

   Answer:

   We can sort n^2 by iterating over all of these elements and inserting into an empty Young tableau, A. A will be a sorted array.

   If there are n^2 elements passed to TABLEAU_SORT, then since INSERT runs in O(n+m) time, and since the resultant matrix will be square matrix of size n, the runtime of TABLEAU_SORT will be $O(n^2 \times 2n) = O(n^3)$.

6. Give an O(m + n)-time algorithm to determine whether a given number is stored in a given m x n Young tableau

   Answer:

   The recursive solution considers three possibilities. First, if target value is greater than the right lower diagonal element, then the value to seek *must* be a value which is somewhere from (k, k) to (m, n). The recursion across the diagonal will eventually stop at some diagonal index, k, after which the next diagonal is too large. At that point, we have ruled out the entire submatrix from (0, 0) to (k, k) and the submatrix from (k, k) to (m, n). But this is only half of all the elements in the matrix. We must still consider the submatrix from (k, 1) to (m, k) and the submatrix from (1, k) to (k, n). Since, these submatrices are completely orthogonal to one-another, there is no more room for optimization and we return the OR of both.

Algorithm 8 CONTAINS(A, x, m=1, n=1)

```
if m > A.numrows or n > A.numcols then
    return FALSE
end if
if A[m][n] == x then
    return TRUE
end if
if m < A.numrows and n < A.numcols and x > A[m+1][n+1] then
    return CONTAINS(A, x, m+1, n+1)
end if
return CONTAINS(A, x, m+1, 1) or CONTAINS(A, x, 1, n+1)
```

In the worst-case analysis, the first diagonal we check is too large, forcing the algorithm to check the first row and first column completely. It will check these sequentially and not find the value, thus taking m+n steps. If instead the worst-case scenario is when the value is larger than all the diagonals, it would only take max(m, n) steps, which is clearly fewer.

The recurrence relation for the worst case scenario is:

$$T(m+n) = \Theta(1) + T(n+m-1)$$
$$T(1) = \Theta(1)$$

To solve this recurrence relation, we can unroll T(m+n):

$$\begin{aligned} T(m+n) &= 1 + T(m+n-1) \\ &= 2 + T(m+n-2) \\ &= 3 + T(m+n-3) \\ &= \ldots \\ &= k + T(m+n-k) \end{aligned}$$

Setting the quantity $m + n - k$ to 1 to solve for k and substituting yields:

$$T(n) = m + n - 1 + T(1)$$

Therefore, $T(m + n) = O(m + n)$

Exercise 11.2-3

Question: Professor Marley hypothesizes that he can obtain substantial performance gains by modifying the chaining scheme to keep each list in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions?
Answer: I will assume the data structures used are doubly linked lists. The hashing to determine into which bucket to place a given value is constant time. Therefore, when comparing the running time of these four operations with both approaches, we must consider only the running time of these operations on the different data structures.

In the worst case analysis, we will be inserting all the new values into the same bucket. Thus, the right way to analyze these two choices is just to analyze the running time of the operations on sorted and unsorted lists. The only operation that will be faster for the unsorted list is insertion. The rest of the operations will be linear for unsorted. All the operations can be $\Theta(n)$ for a sorted list.

	UNSORTED	**SORTED**
SUCCESSFUL SEARCH(X)	$\Theta(n)$	$\Theta(n)$
UNSUCCESSFUL SEARCH(X)	$\Theta(n)$	$\Theta(n)$
INSERT(X)	$\Theta(1)$	$\Theta(n)$
DELETE(X)	$\Theta(n)$	$\Theta(n)$

Problem 10-1

Question: For each of the four types of lists in the following table, what is the asymptotic worst-case running time for each dynamic-set operation listed?
Answer:

	unsorted, singly linked	**sorted, singly linked**	**unsorted, doubly linked**	**sorted, doubly linked**
SEARCH(L, k)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
INSERT(L, x)	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
DELETE(L, x)	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
SUCCESSOR(L, x)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
PREDECESSOR(L, x)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
MINIMUM(L)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
MAXIMUM(L)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$