Heapsort

In-place sort.
   Running time: $O(n \lg n)$

Heaps

Heap: An array $A$ representing a complete binary tree for $\text{HeapSize}(A)$ elements satisfying the heap property:

for every node $i$ except the root node.

Because the array may hold more numbers than are contained in the heap, $\text{HeapSize}(A) \leq \text{length}(A)$ (heap insertion and deletion)

- $\text{parent}(i) = \lfloor i/2 \rfloor$
• left(i) = _______ ; left child

• right(i) = _______ ; right child

Running Times

Running times depend on height of the tree — the height of a node in a tree is the number of edges in the longest simple path from the node to a leaf.

• height(node 8) = 0
• height(node 3) = _______
• height(node 1) = _______

The height of the tree is the _________________________________. Operations on heap proportional to height.

Exercise 7.1-2

Show that an n-element heap has height \text{[ } \lg n \text{ ]}
\[ 1 + \sum_{i=0}^{h-1} 2^i \leq n \leq \sum_{i=0}^{h} 2^i < \sum_{i=0}^{h} 2^i + 1 \]

\[ 1 + \frac{2^h - 1}{2 - 1} \leq n < \frac{2^{h+1} - 1}{2 - 1} + 1 \]

Simplifying a geometric series \((\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1})\)

\[ 2^h \leq n < 2^{h+1} \]

\[ h \leq \lg n < h + 1 \]

Since \(h\) is an integer, \(h = \lfloor \lg n \rfloor \).

**Maintaining the Heap Property**

Heapify(A, i)
Notice that if \( A[\text{left}(i)] > A[i] \) or if \( A[\text{right}(i)] > A[i] \) or both (take largest) then swap and recurse until recursion bottoms out at bottom of heap.

Pseudocode

```
Heapify(A, i)
1   1 = \text{left}(i)
2   r = \text{right}(i)
3   if l \leq \text{HeapSize}(A) and A[l] > A[i]
4       then largest = l
5   else largest = i
6   if r \leq \text{HeapSize}(A) and A[r] > A[largest]
7       then largest = r
8   if largest \neq i
9       then swap(A[largest], A[i])
10      Heapify(A, largest)
```

If the parent is smaller than either of its children, does it matter whether we swap the parent with the larger or the smaller of the children?
Recursive Analysis

$$n = 5, \ n_{\text{subtree}} = \lfloor 2n/3 \rfloor = 3$$

Note that this expression has the maximum value when the lowest level of the heap is exactly half full.

$$T(n) = T(\lfloor 2n/3 \rfloor) + \Theta(1)$$

Master: \( a = 1, \ b = 3/2, \ f(n) = \Theta(1) = \Theta(n^{\log_{3/2} 1}) = \Theta(n^0) = \Theta(1) \)

Case: $$T(n) = \Theta(n^{\log_{3/2} 1} \log n) = \Theta(\log n)$$

BuildHeap

BuildHeap(A)
1 HeapSize(A) = length(A)
2 for i = \lfloor \text{length(A)}/2 \rfloor \text{ downto 1}
3 Heapify(A, i)

\( A[\lfloor n/2 \rfloor + 1 \ldots n] \) are leaves and heaps.
Click on mouse to advance to next frame.

Analysis

There are $O(n)$ calls to Heapify, thus BuildHeap is $O(n \log n)$, which is an upper bound but not a tight bound ($o(n \log n)$).

The tight upper bound is $O(nh)$.

Notice that the height $h$ changes as the heap is being built.

Note: There can be at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes of height $h$ in an $n$-element heap.

Thus Heapify = __________ for nodes of height $h$.

Analysis

From this result we can analyze the run time of BuildHeap.

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n \cdot \frac{1}{2} \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h})$$

Note that $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$, for $|x| < 1$. The derivative of both sides,

$$\frac{d}{dx}(\sum_{k=0}^{\infty} x^k) = \frac{d}{dx}\left(\frac{1}{1-x}\right),$$

is equal to $\sum_{k=0}^{\infty} k x^{k-1} = \frac{1}{(1-x)^2}$.

Multiplying both sides of the equivalence by $x$ we get

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}.$$

In our case $k = h$ and $x = 1/2$. 

6
Thus \( \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2, \ x = 1/2. \)

Thus the run time is ____________________________.

---

**Heapsort**

Heapsort(A)
1    BuildHeap(A)
2    for i = length(A) downto 2
3        swap(A[1], A[i])
4    HeapSize(A) = HeapSize(A) - 1
5    Heapify(A, 1)

Click on mouse to advance to next frame.

---

**Summary**

**BuildHeap:** ________

**Heapify:** ______________________

**Heapsort:** ________
Priority Queues

A **priority queue** is a data structure for maintaining a set $S$ of elements, each with an associated key value.

**Operations:**

**Max($S$):** returns element of $S$ with largest key

**ExtractMax($S$):** removes and returns element with largest key from $S$

**Insert($S,x$):** inserts $x$ into $S$ ($S = S \cup \{x\}$)

---

**Application: Job Scheduling**

Support insertion of prioritized jobs in queue.

Support extraction of highest priority job from queue.

Using heaps:

**Max($S$) = $A[1]$, $\Theta(1)$**

**ExtractMax** $O(\log n)$, First 3 lines $\Theta(1)$, fourth line $O(\log n)$

ExtractMax($S$)

1. $\text{max} = A[1]$
3. $\text{HeapSize}(A) = \text{HeapSize}(A) - 1$
4 Heapify(A, 1)
5 return max

This corresponds roughly to one iteration of HeapSort.

Insert

While loop iterates through height of heap and is thus $O(\log n)$
Insert(A, key)
1 HeapSize(A) = HeapSize(A) + 1
2 i = HeapSize(A)
3 while i > 1 and A[Parent(i)] < key
4 A[i] = A[Parent(i)]
5 i = Parent(i)
6 A[i] = key

Quicksort

In-place, $\Theta(n^2)$ worst case.
O(n lg n) average case with small constant factors.

Description

Divide-and-Conquer

__________ Partition A[p..r] into two non-empty subquences A[p..q] and A[q+1..r] such that each element of A[p..q] is \( \leq \) each element of A[q+1..r].

__________ Sort A[p..q] and A[q+1..r] by recursive calls to Quicksort.

__________ Trivial, arrays are sorted in place.

Pseudocode

Quicksort(A,p,r)
1  if p < r
2   then q = Partition(A,p,r)
3     Quicksort(A,p,q)
4     Quicksort(A,q+1,r)

Initial Call: Quicksort(A,1,length(A))
Partitioning the Array

1. Pick ________ as the pivot
2. Move from ________ to ________ looking for an element ≤ pivot
3. Move from ________ to ________ looking for an element ≥ pivot
4. Swap the two elements
5. Repeat until pointers cross

Partition

Click on mouse to advance to next frame.

Partition(A,p,r)
1   x = A[p]
2   i = p - 1
3   j = r + 1
4   while TRUE
5       repeat
6           j = j - 1
7           until A[j] ≤ x
8       repeat
9           i = i + 1
10       until A[i] ≥ x
11       if i < j
12       then swap(A[i], A[j])
13       else return j
Partition

(pivot is the median of the sequence) leads to $\Theta(n \log n)$ like Merge Sort.

(maximally bad when array already sorted) leads to $\Theta(n^2)$ like Insertion Sort.

Worst Case

Partition always yields subarrays of size n-1 and 1.

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

$= T(n - 2) + \Theta(n - 1) + \Theta(n), \quad T(1) = \Theta(1)$

$= T(n - 3) + \Theta(n - 2) + \Theta(n - 1) + \Theta(n)$

The ith term is $T(n-i)$ and the boundary case is $i=n-1$. The summation is

$$= \sum_{k=1}^{n} \Theta(k)$$

We know that $\sum_{k=1}^{n} \Theta(f(k)) = \Theta(\sum_{k=1}^{n} f(k))$. Thus summation is then

$= \Theta(\sum_{k=1}^{n} k) = \Theta(n^2)$.

Note that Insertion Sort is $O(n)$ in this same case (already sorted).

Best Case

Partition yields subarrays of size n/2 each.
$$T(n) = 2T(n/2) + \Theta(n)$$
$$a=2, b=2, f(n) = \Theta(n) = \Theta(n^{\log_2 2}) = \Theta(n^1), \text{ Case 2}$$
$$T(n) = \Theta(n \log n)$$

**Average Case**

$$\Theta(n \log n)$$

---

**Worst Case Revisited**

Assume we do not know what the worst partition is.

$$T(n) = \max_{1 \leq q \leq n-1} (T(q) + T(n-q)) + \Theta(n).$$

By the substitution method (since we know the answer), try $$T(n) \leq cn^2$$.

$$T(n) \leq \max_{1 \leq q \leq n-1} (cq^2 + c(n-q)^2) + \Theta(n)$$

$$= c \cdot \max_{1 \leq q \leq n-1} (q^2 + (n-q)^2) + \Theta(n)$$

$$\frac{d}{dq} (q^2 + (n-q)^2) = 2q - 2(n-q) = 2q - 2n + 2q = 4q - 2n$$

$$\frac{d}{dq} (4q - 2n) = 4.$$  
For q=1: $$1^2 + (n-1)^2 = n^2 - 2n + 2.$$  
For q=n-1: $$(n-1)^2 + 1^2 = n^2 - 2n + 2.$$
\[ T(n) \leq cn^2 - 2c(n - 1) + \Theta(n) \leq cn^2. \] Picking a large enough \( c \),
\[ T(n) = \Theta(n^2) \]

Summary of Comparison Sorts

<table>
<thead>
<tr>
<th>Sort</th>
<th>Worst Case</th>
<th>Average Case</th>
<th>Best Case</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n) )</td>
<td></td>
</tr>
<tr>
<td>Merge Sort</td>
<td>( \Theta(n \log n) )</td>
<td>( \Theta(n \log n) )</td>
<td>( \Theta(n \log n) )</td>
<td>Requires Memory</td>
</tr>
<tr>
<td>Heapsort</td>
<td>( \Theta(n \log n) )</td>
<td>( \Theta(n \log n) )</td>
<td>( \Theta(n \log n) )</td>
<td>Large constants</td>
</tr>
<tr>
<td>Quicksort</td>
<td>( \Theta(n^2) )</td>
<td>( \Theta(n \log n) )</td>
<td>( \Theta(n \log n) )</td>
<td>Small constants</td>
</tr>
</tbody>
</table>

Applications