CMPU241 Analysis of Algorithms

Optimal Comparison-Based Sorting Algorithms

Sorting Algorithms (Ch. 6 - 8)
Slightly modified definition of the sorting problem:

input: A collection of n data items \(a_1, a_2, ..., a_n\) where data item \(a_i\) has a key, \(k_i\), drawn from a linearly ordered set (e.g., ints, chars)

output: A permutation (reordering) \(a'_1, a'_2, ..., a'_n\) of the input sequence such that \(k_1 \leq k_2 \leq ... \leq k_n\)

• In practice, one usually sorts objects according to their key (the non-key data is called satellite data.)
• If the records are large, we may sort an array of pointers based on some key associated with each record.

Sorting Algorithms

• A sorting algorithm is comparison-based if the only operation we can perform on keys is to compare them.
• A sorting algorithm is in place if only a constant number of elements of the input array are ever stored outside the array.

Running Time of Comparison-Based Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>worst-case</th>
<th>average-case</th>
<th>best-case</th>
<th>in place?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>(n^2)</td>
<td>(n^2)</td>
<td>(n)</td>
<td>no</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>(n \log n)</td>
<td>(n \log n)</td>
<td>(n \log n)</td>
<td>no</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>(n \log n)</td>
<td>(n \log n)</td>
<td>(n \log n)</td>
<td>yes</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>(n^2)</td>
<td>(n \log n)</td>
<td>(n \log n)</td>
<td>yes</td>
</tr>
</tbody>
</table>

Heap-Sort (Chapter 6)
In order to understand heap-sort, you need to understand binary trees.
The algorithm doesn't use a data structure for nodes as you might be familiar with when working with binary trees.
Instead, it uses an array to abstract away from the complexity of linked binary trees. In so doing, the algorithm has a fast run time with low-cost operations: swapping the values in an array, like Insertion-Sort and Bubble-Sort do.

Binary Trees

- A rooted tree in which each internal node has at most 2 children

complete binary tree
- Each level of the binary tree is full.
Heaps

A heap is a complete binary tree or an almost-complete binary tree, with the requirement that it may be missing only the rightmost leaves on the bottom level.

We say the bottom level is left-filled.

Each node contains a key and the keys are some totally ordered, comparable type of data.

Max-Heap

In the array representation of a max-heap, the root of the tree is in A[1], given an index i of a node,

Parent(i)  return ⌊i/2⌋
LeftChild(i) return (2i)
RightChild(i) return (2i + 1)

Max-heap property: A[Parent(i)] ≥ A[i]

Min-Heap

In the array representation of a min-heap, the root of the tree is in A[1], and given the index i of a node,

Parent(i)  return ⌊i/2⌋
LeftChild(i) return (2i)
RightChild(i) return (2i + 1)

Min-heap property: A[Parent(i)] ≤ A[i]

Creating a Heap: Build-Max-Heap

• Observation: Leaves are already trivial max-heaps.
  Elements A[(n/2) + 1] ... n are leaves.
  Elements A[1...[(n/2)]] are internal nodes.

• Start at parents of leaves...then go to grandparents of leaves...moving larger values up the tree.

Build-Max-Heap(A)
  1. for i = ⌊A.length/2⌋ downto 1
  2. Max-Heapify(A, i)

Running Time of Build-Max-Heap

• About n/2 calls to Max-Heapify (O(n) calls)

Heapsort

Imaginary nodes are numbered using level ordering.

A heap is represented with an array

• root is A[1]
  - left child is in position A[2i]
  - right child is in position A[2i + 1]
  - parent is in A[⌊i/2⌋]

Variables used for the array implementation of a heap

• heapsize is number of elements in heap
• length is number of positions in array

1 2 3 4 5 6 7 8 9 10 11

Max-heaps are used for, e.g., sorting data and for priority queues.

Min-heaps are used for, e.g., priority queues in event-driven simulators.

Max-Heapify: Maintaining the Heap Property

• Precondition: when M-H is called on a node i, the subtrees rooted at the left and right children of A[i], A[2i] and A[2i + 1] are max-heaps (i.e., they obey the max-heap property)

• ...but subtree rooted at A[i] might not be a max-heap (that is, A[i] may be smaller than its left and/or right child)

• Postcondition: Max-Heapify will cause the value at A[i] to be compared and swapped with the largest child of A[i], to "float down" or "sink" in the heap until the subtree rooted at A[i] becomes a max-heap.

• In a totally unordered array, execution would start at the first parent node of a leaf because all leaves are max-heaps.
Max-Heapify: Maintaining the Max-Heap Property
Precondition: the subtrees rooted at 2i and 2i+1 are max-heaps when Max-Heapify(A, i) is called.

Max-Heapify(A, i)
1. left = 2i  /* index of left child of A[i] */
2. right = 2i + 1  /* index of right child of A[i] */
3. largest = i
5. largest = left
7. largest = right
8. if largest != i
9. swap(A[i], A[largest]) /* swap i with larger child */
10. Max-Heapify(A, largest) /* continue heapifying to the leaves */

Max-Heapify: Running Time
Running Time of Max-Heapify
• every line is $\Theta(1)$ time except the recursive call in line 10.
• in worst-case, last level of binary tree is half empty and the sub-tree rooted at left child of root has size at most $(2/3)n$.
Note that in a complete binary tree (CBT) the subtrees to left and right would be equal size.
We get the recurrence $T(n) = T(2n/3) + \Theta(1)$
which, by case 2 of the Master Theorem, has the solution $T(n) = \Theta(n)$
Max-Heapify takes $O(h)$ time when node $A[i]$ has height $h$ in the heap. The height $h$ of a tree is the longest root to leaf path in the tree. $h = O(\log n)$ in the worst case

Creating a Heap: Build-Max-Heap
• Observation: Leaves are already max-heaps. Elements $A[((n/2) + 1) \ldots n]$ are all leaves.
• Start at parents of leaves...then go up to grandparents of leaves...etc.

Build-Max-Heap(A)
1. A.heapsize = A.length
2. for $i = \lfloor A.length/2 \rfloor$ downto 1
3. Max-Heapify(A, i)

Running Time of Build-Max-Heap
• About $n/2$ calls to Max-Heapify ($O(n)$ calls)

Correctness of Build-Max-Heap
• The entire array $A$ meets the Max-Heap property.

Correctness of Build-Max-Heap
Termination: at termination, $i = 0$. By the loop invariant, nodes 1, 2, ..., $n$ are the roots of max-heaps. Therefore the algorithm is correct.

Build-Max-Heap(A)
1. A.heapsize = A.length
2. for $i = \lfloor A.length/2 \rfloor$ downto 1
3. Max-Heapify(A, i)

Reminder from last slide:
Loop invariant: At the start of each iteration $i$ of the for loop, each node $i + 1$, $i + 2$, ..., $n$ is the root of a max-heap.

Heap Sort
Input: An $n$-element array $A$ (unsorted).
Output: An $n$-element array $A$ in sorted order, smallest to largest.

HeapSort(A)
1. Build-Max-Heap(A) /* rearrange elements to form max heap */
2. for $i = A.length$ downto 2 do
3. swap(A[1] and A[i]) /* puts max in ith array position */
4. A.heapSize = A.heapSize - 1
5. Max-Heapify(A, 1) /* restore heap property */

Relies on observation that the largest element in the array is at the top of the heap.

Does this algorithm have best case and worst case running times?
Heap Sort

**Input:** An n-element array A (unsorted).

**Output:** An n-element array A in sorted order, smallest to largest.

HeapSort(A)

1. Build-Max-Heap(A) /* rearrange elements to form max heap */
2. for i = A.length downto 2 do
4. A.heapSize = A.heapSize - 1
5. Max-Heapify(A, 1) /* restore heap property */

Input:
An n-element array A (unsorted).

Output:
An n-element array A in sorted order, smallest to largest.

Build-Max-Heap(A) takes $O(n \lg n)$ time

Max-Heapify(A, 1) takes $O(\lg |A|) = O(\lg n)$ time

Running time of HeapSort
- 1 call to Build-Max-Heap() ⇒ $O(n)$ time
- n-1 calls to Max-Heapify() each takes $O(\lg n)$ time ⇒ $O(n \lg n)$ time

Heaps as Priority Queues

**Definition:** A priority queue is a data structure for maintaining a set S of elements, each with an associated key. A max-priority-queue gives priority to keys with larger values and supports the following operations:

1. `insert(S, x)` inserts the element x into set S.
2. `heap-maximum(S)` returns value of element of S with largest key.
3. `extract-max(S)` removes and returns element of S with largest value key.
4. `increase-key(S, x, k)` increases the value of element x's key to new value k (assuming k is at least as large as current key's value).

Priority Queues: Application for Heaps

An application of max-priority queues is to schedule jobs on a shared processor. Need to be able to:
- check current job's priority
- remove job from the queue
- insert new jobs into queue
- increase priority of jobs

Initialize PQ by running Build-Max-Heap on an array A. A[1] holds the maximum value after this step.

Heapsort Time and Space Usage

- An array implementation of a heap uses $O(n)$ space, one array element for each node in heap.
- Heapsort uses $O(n)$ space and is in place, meaning at most constant extra space beyond that taken by the input is needed.
- Running time is as good as merge sort, $O(n \lg n)$ in worst case.

Inserting Heap Elements

Inserting an element into a max-heap:
- increment heapsize and "add" new element to the highest numbered position of array
- go from new leaf to root, swapping values if child > parent. Insert input key at node in which a parent key larger than the input key is found

**Max-Heap-Insert(A, key)**

1. A.heapsize = A.heapsize + 1
2. i = A.heapsize
3. while i > 1 and A[parent(i)] < key
5. i = parent(i)
6. A[i] = key

Here, values are moved up to where they should be in a max-heap.

Running time of Max-Heap-Insert: $O(\lg n)$
- time to traverse leaf to root path (height = $O(\lg n)$)

Heap-Increase-Key

**Heap-Increase-Key(A, i, key)**: If key is larger than current key at A[i], moves node with increased key up heap until heap property is restored by exchanging it with its smaller parent until parent key is > A[i].

An application for a min-heap priority queue is an event-driven simulator, where the key is an integer representing the number of seconds (or other discrete time unit) from time zero (starting point for simulation).
Sorting Algorithms

- A sorting algorithm is comparison-based if the only operation we can perform on keys is to compare them.
- A sorting algorithm is in place if only a constant number of elements of the input array are ever stored outside the array.

### Running Time of Comparison-Based Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>worst-case</th>
<th>average-case</th>
<th>best-case</th>
<th>in place?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion Sort</td>
<td>$n^2$</td>
<td>$n^2$</td>
<td>$n$</td>
<td>yes</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>no</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>yes</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>yes</td>
</tr>
</tbody>
</table>

Addendum

**Build-Max-Heap - Tighter bound: O(n)**

```plaintext
Build-Max-Heap(A)
1. A.heapsize = A.length
2. for i ← ⌊length(A)/2⌋ down to 1
3. Max-Heapify(A, i)
```

Proof of tighter bound for $O(n)$ relies on following theorem:

**Theorem 1:** The number of nodes at height $h$ in a max-heap $\leq \lceil n/2^{h+1} \rceil$

Height of a node $v$ = largest number of edges from $v$ to a leaf.
Depth of a node $v$ = number of edges from node $v$ to the root.

Tight analysis relies on the properties that an $n$-node heap has height at least floor of $\log n$ and at most the ceiling of $n/2$ nodes at height $h$. The time for max-heapify to run at a node varies with the height of the node in the tree, and the heights of most nodes are small.

**Diagramatic proof of Lemma 1**

```
#Internal nodes in T = #Internal nodes in T1 + #Internal nodes in T2 + 1
= (leaves in T1 - 1) * (leaves in T2 - 1) + 1 (IHOP)
= (leaves in T1) + (leaves in T2) - 2 + 1
= #leaves in T - 1
(by observation that # of leaves in T is equal to # leaves in its subtrees.)
```

**Lemma 1:** The number of internal nodes in a proper binary tree is equal to the number of leaves in the tree - 1.

Defn: In a proper binary tree (pbt), each node has exactly 0 or 2 children.

Let $I$ be the number of internal nodes and let $L$ be the number of leaves in a proper binary tree. The proof is by induction on the height of the tree.

**Basis:** $h=0$. $I = 0$ and $L = 1$. $I = L - 1 = 1 - 1 = 0$, so the lemma holds.

**Inductive Step:** Assume lemma is true for proper binary trees of height $h$ (IHOP) and show for proper binary trees of height $h + 1$.

Consider the root of a proper binary tree $T$ of height $h+1$. It has left and right subtrees ($L$ and $R$) of height at most $h$.

$I_T = (I_L + I_R) + 1 = (L_L - 1) + (L_R - 1) + 1$ (by the IHOP) = $(L_L + L_R - 2) + 1 = L + L - 1$. Since $L_T = L + L$ we have that $I = L - 1$.

**QED**

**Theorem 1:** The number of nodes at level $h$ in a max-heap $\leq \lceil n/2^{h+1} \rceil$

Let $h$ be the height of the heap. Proof is by induction on $h$, the height of each node. The number of nodes in the heap is $n$.

**Basis:** Show the theorem holds for nodes with $h = 0$. The tree leaves (nodes at height 0) are at depths $H$ and $H-1$.

Let $x$ be the number of nodes at depth $H$, that is, the number of leaves assuming that $n$ is a complete binary tree, i.e., that $n = 2^{H+1}$.

Note that $n - x$ is odd, because a complete binary tree has an odd number of nodes (1 less than a power of 2).
Theorem 1: The number of nodes at level $h$ in a max-heap $\leq \left\lceil \frac{n}{2^{h+1}} \right\rceil$

We have that $n$ is odd and $x$ is even, so all nodes have siblings (all internal nodes have 2 children.) By Lemma 1, the number of internal nodes = the number of leaves - 1.

So $n =$ # of nodes = # of leaves + # internal nodes = $2#$ of leaves - 1. Thus, the # of leaves = $(n+1)/2 = \left\lceil \frac{n}{2} \right\rceil$ because $n$ is odd.

Thus, the # of leaves = $\left\lceil \frac{n}{2} \right\rceil$ and the theorem holds for the base case.

Inductive step: Show that if thm 1 holds for height $h-1$, it holds for $h$.

Let $n_h$ be the number of nodes at height $h$ in the $n$-node tree $T$.

Consider the tree $T'$ formed by removing the leaves of $T$. It has $n' = n - n_h$ nodes. We know from the base case that $n_0 = [n/2]$, so $n' = n - [n/2] = [n/2]$. Note that the nodes at height $h$ in $T$ would be at height $h-1$ if the leaves of the tree were removed—i.e., they are at height $h-1$ in $T'$. Letting $n_{h-1}'$ denote the number of nodes at height $h-1$ in $T'$, we have $n_h = n_{h-1}'$ and $n_h = n_{h-1}' \leq [n/2]$ (by the IHOP) $\leq \left\lceil \frac{n}{2} \right\rceil \leq [n/2] = [n/2^h]$. Since the time of Max-Heapify when called on a node of height $h$ is $O(h)$, the time of B-M-H is

$$\sum_{h=0}^{\log n} O(h) = O(n \sum_{h=0}^{\log n} h)$$

and since the last summation turns out to be a constant, the running time is $O(n)$.

Therefore, we can build a max-heap from an unordered array in linear time.