Heapsort (chapter 6)

Sorting Algorithm Terminology
- 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>

Binary Trees
- **binary tree**: A rooted tree in which each internal node has at most 2 children.
- **complete binary tree**: Each level of the binary tree is full except possibly the lowest. All nodes on lowest level are as far left as possible (i.e., tree is left filled).

Binary Heaps
- **heap**: A data structure used to efficiently find the largest or smallest element in a set.
- **binary heap**: A binary heap is a complete binary tree, i.e., it may be missing some rightmost leaves on the bottom level. The bottom level is left-filled.
Binary Heaps

Heap implementation is an array-embedded binary tree
- Encoding stores tree elements at particular indexes in an array.
- Uses "level-ordering" and 1-based indexing (our textbook).

Max-Heap

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

\[
\begin{align*}
\text{Parent}(i) & \quad \text{LeftChild}(i) \quad \text{RightChild}(i) \\
\text{return } & \quad \text{return } ((i+1)/2) \quad \text{return } (2i + 1)
\end{align*}
\]

Max-heap invariant: \( A[\text{Parent}(i)] > A[i] \)

Min-Heap

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

\[
\begin{align*}
\text{Parent}(i) & \quad \text{LeftChild}(i) \quad \text{RightChild}(i) \\
\text{return } & \quad \text{return } ((i+1)/2) \quad \text{return } (2i + 1)
\end{align*}
\]

Min-heap invariant: \( A[\text{Parent}(i)] < A[i] \)

Creating a Heap: Build-Max-Heap

- Starting from an unordered array of \( n \) elements.
- Observation: Leaves are already trivial max-heaps.
  Elements \( A[(i/2) + 1] \ldots n \) are all leaves.
- Start at parents of leaves...then go to grandparents of leaves...moving larger values up the tree and moving lower values down in the tree.

\[
\begin{align*}
\text{Build-Max-Heap}(A) \\
1. & \quad \text{for } i = \lfloor A.length/2 \rfloor \text{ downto } 1 \\
2. & \quad \text{Max-Heapify}(A, i)
\end{align*}
\]

Running Time of Build-Max-Heap
- About \( n/2 \) calls to Max-Heapify (\( O(n) \) calls)
Max-Heapify: Maintaining the Max-Heap Property

Precondition: the subtrees rooted at 2i and 2i+1 are max-heaps

Max-Heapify(A, i) /* Max-Heapify is also known as “Sink” */
1. left = 2i ; right = 2i + 1
2. largest = i /* set largest to i, parent node of left and right */
4.       largest = left /* reset largest to left child */
6.       largest = right /* reset largest to right child */
7. if  largest != i /* keep sinking i in tree */
8.        swap(A[i], A[largest])
9.        Max-Heapify(A, largest) /* continue heapifying toward leaves */

Sink: Alternate version of Max-Heapify

Sink compares (possibly smaller) parent i to (possibly larger) left and right children and swaps key of child with largest key with parent. Correctness relies on the precondition that the left and right children are the roots of max-heaps.

Assume size >> heapsize and heapsize is highest-numbered node in heap.

Max-Heapify: Maintaining the Max-Heap Property

• Precondition: 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)

• Postcondition: Max-Heapify will cause the value at A[i] to "float down" or "sink" in the heap until the subtree rooted at A[i] becomes a heap.

In a totally unordered array, execution would start at the highest numbered parent node of a leaf.

Sink: Iterative version of Max-Heapify

Precondition: the subtrees 2i and 2i+1, are max-heaps. Let n = heapsize

private void sink(int k)
{
    while (2*k <= n)
    {
        int j = 2*k;
        if ( j < n && less(j, j+1)) j++;
        if ( !less(k, j )) break;
        swap(k, j);
        k = j;
    }
}

private boolean less(int i, int j)
{
    return A[i].compareTo(A[j]) < 0;
}

private void swap(int i, int j)
{
    int t = A[i];
    A[i] = A[j];
    A[j] = t;
}

Max-Heapify: Running Time

Running Time of Max-Heapify

• every line is Θ(1) time except the recursive call
• in worst-case, last level of binary tree is half empty, so the sub-tree rooted at left child has size < (2/3)n

We get the recurrence

T(n) ≤ T(2n/3) + Θ(1)

which, by case 2 of the master theorem, has the solution

T(n) = O(\log n)

(or, Max-Heapify takes O(h) time when node A[i] has height h in the heap) The height h of the tree is the root to leaf path with the most edges. O(h) = O(\log n)

Proposition 1:

Sink-based heap construction uses fewer than 2n compares and fewer than n exchanges to construct a heap from n items.

Proof sketch:

Follows from the observation that most of the heaps processed are small. E.g., to build a heap of 127 items, we process 32 heaps of size 3, 16 heaps of size 7, 8 heaps of size 15, 4 heaps of size 31, 2 heaps of size 63, and 1 heap of size 127. Thus, there are 32X1 + 16X2 + 8X3 + 4X4 + 2X5 + 1X6 = 120 exchanges (and twice as many compares) required (at worst).
Inserting an element into a max-heap:
- increment heap size and "add" new element to the highest numbered position of array
- walk up tree from new leaf to root, swapping values. 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

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

Correctness of Build-Max-Heap

**Loop invariant:** At the start of each iteration \(i\) of the for loop, each node \(i, i+1, \ldots, n\) is the root of a max-heap.
- Initialization: \(i = (n/2)\). Each node \((n/2) + 1, (n/2) + 2, \ldots, n\) is a leaf, trivially satisfying the max-heap property.
- Inductive hypothesis: At the start of iteration \(k\) (\(1 \leq k \leq n/2\)), the subtrees of \(k\) are the roots of max-heaps.
- Inductive step (maintenance): During iteration \(k\), Max-Heapify is called on node \(k\). By the IH, the left and right subtrees of \(k\) are max-heaps. When Max-Heapify is called on node \(k\), the value in node \(k\) is "floated down" in its subtree until its value is correctly positioned in the max-heap rooted at \(k\).

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

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

Correctness of Build-Max-Heap

**Termination:** at termination, \(i = 0\). By the loop invariant, nodes \(1, 2, \ldots, n\) are the roots of max-heaps. Therefore the algorithm is correct because it produces a max-heap.

**Loop invariant:** At the start of each iteration \(i\) of the for loop, each node \(i + 1, i + 2, \ldots, n\) 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 /* decrease heap size */
5. Max-Heapify(A, 1) /* restore heap property from node \(1\) */

**Running time of HeapSort**
- \(O(n)\) time
- \(n-1\) calls to Max-Heapify

**Max-Heapify(A, 1) takes** \(O(\log n)\) time

Iterative version of Heap Sort

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

**public static void sort(Comparable[] A)\**
\{\n    int n = A.length;
    // start at highest numbered parent node
    for (int k = n/2; k >= 1; k--)\n    \{\n        sink(A, k, n);
        while (n > 1)\n        \{\n            swap(A, 1, n--);
            sink(A, 1, n);
        \} \n        \}\n\}
Iterative version of Heap Sort

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

```java
public static void sort(Comparable[] A)
{
    int n = A.length;
    // start at highest numbered parent node
    for (int k = n/2; k >= 1; k--)
        sink(A, k, n);
    while (n > 1)
    {
        swap(A, 1, n--);
        sink(A, 1, n);
    }
}
```

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

Running time of sort
- For loop iterates \( \frac{n}{2} \) times. Each call to sink takes \( O(\log n) \) time.
- \( \Rightarrow O(n \log n) \) time

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, \( O(n \log n) \) in worst case.

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-heap gives priority to keys with larger values and supports the following operations:
1. `insert(A, x)` inserts the element \( x \) into array at next highest position in \( A \).
2. `max(A)` returns value of element \( A \) with largest key.
3. `extract-max(A)` removes and returns element of \( A \) with largest key.
4. `increase-key(A, 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

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.

Heap-Increase-Key

Heap-Increase-Key(A, i, key) - If key is larger than current key at \( A[i] \), moves new value in \( A[i] \) up heap until heap property is restored.

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.

**Running Time of Comparison-Based Sorting Algorithms**

<table>
<thead>
<tr>
<th></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>
Thus, the number of leaves = \( \frac{n+1}{2} \).

So \( n \) = number of nodes = number of leaves + number of internal nodes = \( 2(\text{# of leaves}) - 1 \).

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.

**Build-Max-Heap - Tighter bound**

<table>
<thead>
<tr>
<th>Build-Max-Heap(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A.heapsize = A.length</td>
</tr>
<tr>
<td>2. for i = ( \lfloor \text{length(A)/2} \rfloor ) downto 1</td>
</tr>
<tr>
<td>3. Max-Heapify(A, i)</td>
</tr>
</tbody>
</table>

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

**Theorem 1:** The number of nodes at height \( h \) in a max-heap \( \leq \lfloor n/2^{h+1} \rfloor \)

- **Height of a node** \( v \) = longest distance from \( v \) to a leaf.
- **Depth of a node** \( v \) = distance from node \( v \) to the root.

Tight analysis relies on the properties that an \( n \)-node heap has height floor of \( \log n \) and at most the ceiling of \( n/2^{h+1} \) 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.

**Diagrammatic proof of Lemma 1**

- Internal nodes in \( T \) = \# internal nodes in \( T_1 \) + \# internal nodes in \( T_2 \) + 1
- \# leaves in \( T \) = \# leaves in \( T_1 \) + \# leaves in \( T_2 \) + 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.

**Diagrammatic proof of Lemma 1**

- Let \( T \) be 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 \( h \).
- \( L_1 = (L_2 + L_3) + 1 = (L_2 - 1) + (L_2 - 1) + 1 \) (by the IHOP)
- \( L_2 = L_3 + L_4 \) we have that \( L_2 = L_3 - 1 \).
- Note that each node at height \( h \) (e.g. 1) in \( T \) would be at height \( h-1 \) (e.g. 0) if the leaves of the tree were removed--i.e., they are at height \( h-1 \) in \( T' \).
- Let \( x \) be the number of nodes at depth \( H \), that is, the number of leaves, assuming that the tree is a complete binary tree, i.e., that \( n = 2^{H+1} - 1 \).

**Theorem 1:** The number of nodes at level \( h \) in a max-heap \( \leq \lfloor n/2^{h+1} \rfloor \)

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 \( n \).

**Basic:** Show the theorem holds for nodes with \( h = 0 \). The tree leaves (nodes at height 0) are at depth \( H \).

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

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

Note that each node at depth \( H \) has exactly 0 or 2 children.

**Theorem 1:** The number of nodes at level \( h \) in a max-heap \( \leq \lfloor n/2^{h+1} \rfloor \)

Let \( h \) be the number of internal nodes at height \( h-1 \), it holds for \( h \).

**Basic:** Show that if \( h \) is odd and \( x \) is even, so all leaves have siblings (all internal nodes have 2 children.) By Lemma 1, the number of internal nodes is the number of leaves - 1.

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

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

Note that each node at depth \( H \) has exactly 0 or 2 children.

**Theorem 1:** The number of nodes at level \( h \) in a max-heap \( \leq \lfloor n/2^{h+1} \rfloor \)

Let \( x \) be the number of nodes at height \( h \) in the proper binary tree \( T \). The proof is by induction on the height of \( T \).

Basic: \( h = 0 \), \( I = 0 \) and \( L = 1 \). \( I = L = 1 = 1 = 0 \), 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 \( h + 1 \).

We have that \( n = 2 \cdot 2^{h-1} - 1 \). Since \( L_2 = L_3 - 1 \), we have that \( L_2 = L_3 - 1 \).

Thus, the number of leaves = \( \lfloor n/2^{h+1} \rfloor \) and the theorem holds for the base case.
Since the time of Max-Heapify when called on a node of height $h$ is $O(h)$, the time of B-H-H is

$$
\sum_{h=1}^{\log n} n \cdot O(h) = O(n) \sum_{h=1}^{\log n} h
$$

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.