Lecture 16: Heap Sort

Adding a Node to (Min) Heap

- Step 1: Add node at the end
- Step 2: Make swap if parent is bigger
- Step 3: Continue to swap if parent is bigger

Takes $O(\log n)$ steps if $n$ nodes are added

Delete a Node from (Min) Heap

- Step 1: Remove root node
- Step 2: Move last node to root
- Step 3: Make swap if child is smaller
- Step 4: Make swap if child is smaller

Takes $O(\log n)$ steps if $n$ nodes are deleted

Priority Queue Sort

- We can use a priority queue to sort a set of elements.
  1. Insert the elements one by one with a series of insert operations.
  2. Remove the elements in sorted order with a series of deleteMax operations.

Algorithm PQ-Sort($S, C$)

```
Input: sequence $S$, comparator $C$ for the elements of $S$
Output: sequence $S$ sorted in increasing order according to $C$

$P \leftarrow$ priority queue with comparator $C$

while $\neg P.isEmpty()$
  $e \leftarrow P.removeMax().getKey()$
  $S.addLast(e)$
```

Priority Queue Sort using a Heap

public void heapSort(int[] A)
{
    Heap H = new Heap();
    for (int i = 0; i < A.length; i++)
        H.insert(A[i]);
    for (int i = 0; i < A.length; i++)
        A[i] = H.deleteMax();
}

Insertion Heapifying of an Array

• Convert an unordered array into a heap

• Shift up from the root if necessary

• Continue to shift up until the last node
Insertion Heapifying of an Array

- Continue to shift up until the last node

```
24 20 3 5 13 0 50 15 17 12 53 14 33
```

```
24 20 3 5 13 0 50 15 17 12 53 14 33
```

```
24 20 3 5 13 0 50 15 17 12 53 14 33
```

```
24 20 3 5 13 0 50 15 17 12 53 14 33
```
Insertion Heapifying of an Array

• Continue to shift up until the last node

50 20 24 5 13 0 3 15 17 12 53 14 33

13

50 20 24 5 13 0 3 15 17 12 53 14 33

14

50 20 24 15 13 0 3 5 17 12 53 14 33

16

50 20 24 17 13 0 3 5 15 12 53 14 33
Insertion Heapifying of an Array

• Continue to shift up until the last node

50 20 42 17 53 0 3 5 15 12 13 14 33

Insertion Heapifying of an Array

• Continue to shift up until the last node

50 20 42 17 53 0 3 5 15 12 13 14 33
Insertion Heapifying of an Array

- Continue to shift up until the last node

Heap-Sort

- Given: an unsorted array
- Objective: sort the array
  - Make unsorted array a heap
  - Easy to find the maximum (minimum) element in heap
  - Discard the maximum (minimum) element, easy to find the maximum (minimum) element in the remaining heap
  - Repeat until all elements are sorted.
- In-place algorithm

Heap-Sort Procedure

- Step 1: Make array a heap.
Heap-Sort Procedure

• Step 1: Make array a heap.

unsorted array: 8 20 33 10 7 2 6 15 19 27
heap: 33 27 19 20 8 10 6 7

• Step 2: Swap the maximum and last number in the heap.

unsorted array: 7 27 8 19 20 2 6 15 10 33
heap: 33 27 8 19 20 2 6 15 10 7

• Step 3: Discard the last item in the heap and heapify the remaining array.

unsorted array: 7 27 8 19 20 2 6 15 10 33
heap: 27 8 19 20 6 15 10 7

• Step 2: Swap the maximum and last number in the heap.

unsorted array: 10 20 8 19 7 2 6 15 27 33
heap: 27 20 8 19 7 2 6 15 10 33

• Step 2: Swap the maximum and last number in the heap.

unsorted array: 10 20 8 19 7 2 6 15 27 33
heap: 27 20 8 19 7 2 6 15 10 33
Heap-Sort Procedure

• Step 3: Discard the last item in the heap and heapify the remaining array.

Unsorted array: 10 20 8 19 7 2 6 15 27 33

Heap: 20 19 8 15 7 2 6 10 27 33

Heap-Sort Procedure

• Repeat swap and heapify operations.

Using a heap-based priority queue,
- We can sort a sequence of \( n \) elements in \( O(n \log n) \) time.
- The resulting algorithm is called heap-sort.
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort.
A Faster Heap-Sort

- Time complexity of insertion
  - Insert $n$ keys one by one taking $O(n \log n)$ times
  - If we know all keys in advance, we can save the construction to $O(n)$ times by bottom up construction

- Bottom-up construction
  - We can construct a heap storing $n$ given keys $\log n$ with phases.
  - In phase $i$, a pair of heaps with $2^i - 1$ keys are merged into a heap with $2^{i+1} - 1$ keys.

Merging Two Heaps

- Procedure
  - Given two heaps and a key $k$, create a new heap with the root node storing $k$ and with the two heaps as subtrees.
  - We shift down to restore the heap-order property.

Analysis of Merging

- Shift-down with a proxy path
  - Goes right first and then repeatedly goes left until the leaf of the heap
  - Note: This path may differ from the actual shift-down path.
  - Each edge is traversed by at most one proxy path:

- Time complexity
  - Successive insertions: $O(n \log n)$
  - Bottom-up construction: $O(n)$

Time Complexity Comparison

- Comparison to other sorting algorithms

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Time Cost</th>
<th>Space Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick</td>
<td>$O(n \log n)$</td>
<td>1</td>
</tr>
<tr>
<td>Insertion</td>
<td>$O(n^2)$</td>
<td>1</td>
</tr>
<tr>
<td>Merge</td>
<td>$O(n \log n)$</td>
<td>$n$</td>
</tr>
<tr>
<td>HeapSort</td>
<td>$O(n \log n)$</td>
<td>1</td>
</tr>
</tbody>
</table>