Heaps

Josh Wilcox (jw14g24@soton.ac.uk)

February 20, 2025

Contents

1 Heaps

1.1 Properties

- A heap is a binary tree satisfying these two constraints:
  - It is a complete tree
    - Every level above the lowest level is fully occupied
    - The nodes on the lowest level are all to the left
  - Each child has a value greater than equal to its parent

1.2 Adding to Heaps

- Add the element to the next available space in the tree
  - This ensures that the tree remains complete.
  - The next available space is found by filling the tree level by level from **left to right** .
- *Percolate* the value up the tree to maintain the correct ordering
  - Compare the added element with its parent node.
  - If the added element is greater than its parent (for a max-heap) or smaller (for a min-heap), swap them.
  - Repeat this process until the correct position is found or the root is reached.

2 Using heaps for priority queues

2.1 Reminder on Priority Queues

- Each element has a priority
- The element with the highest priority (smallest number) is the **head** of the queue

2.2 Heap Priority Queue Implementation

2.2.1 removeMin

- In a priority queue heap, the minimum element is the root of the tree
- Removing this element:
  - Pop the root
  - Replace it with the last element in the heap,
  - Percolate this element down to the bottom of the heap **choosing the minimum child**

2.2.2 Example of removing the minimum element

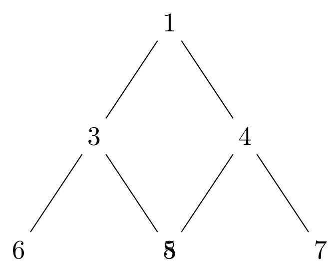


Figure 1: Initial heap with root 1

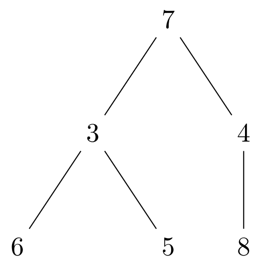


Figure 2: Replace root with last element (7)

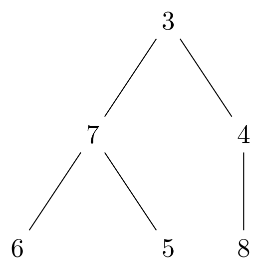


Figure 3: Percolate down: swap 7 with 3

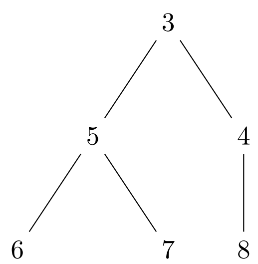
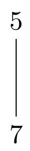
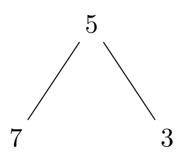
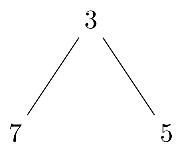
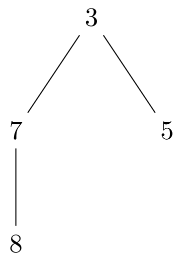
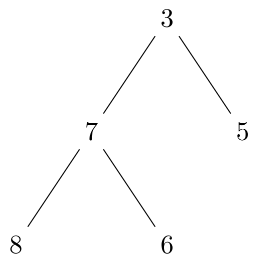
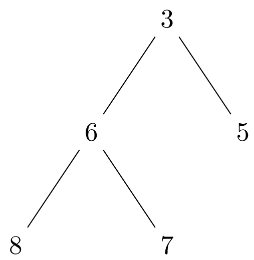
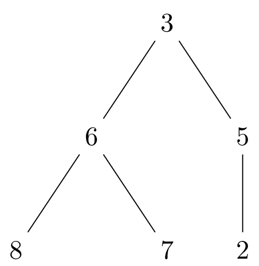
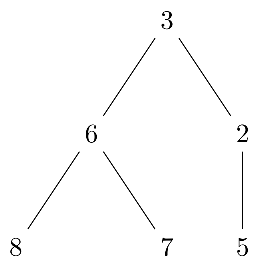


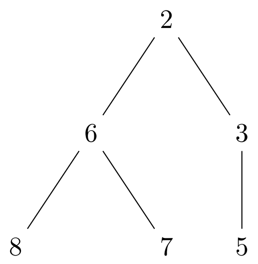
Figure 4: Percolate down: swap 7 with 5

2.3 Example of Adding Elements to a Heap

- Insert 5:
  - 5
- Insert 7:
  - 

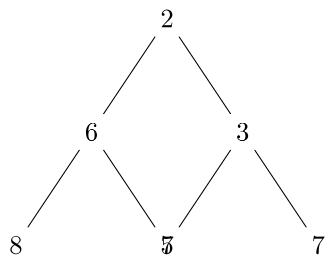
- Insert 3:
  - 

  - Swap 3 with 5: 
- Insert 8:
  - 
- Insert 6:
  - 
  - Swap 6 with 7: 
- Insert 2:
  - 
  - Swap 2 with 5: 



  - Swap 2 with 3:

- Insert 7:



  -

3 Array Implementation of Heaps

- As heaps are complete, we can just stick the elements in an array from left to right, travelling down as we go

3.1 Navigating a heap

- Root of the tree is at array location 0
- Last element is at array location `size()-1`
- Parent of a node k is at location $\lfloor (k-1)/2 \rfloor$
- The children of node k are at array location $2k+1$ and $2k+2$

```java
import java.util.ArrayList;

// HeapPQ is a priority queue implemented using a binary heap.
// It uses a min-heap structure where the smallest element is always at the root.
// The heap is stored in an ArrayList, enabling efficient array-based tree
↪  navigation.
public class HeapPQ<T extends Comparable<T>> {
    // The ArrayList that stores the heap elements.
    // A complete binary tree is represented in an array such that:
    // - The root is at index 0.
    // - For any node at index k, its left child is at 2k+1 and right child at
    ↪  2k+2.
    private ArrayList<T> heap;

    // Constructor: initializes an empty heap.
    public HeapPQ() {
        heap = new ArrayList<>();
    }

    // Returns the number of elements in the heap.
```

```
    public int size() {
        return heap.size();
    }

    // Checks if the heap is empty.
    public boolean isEmpty() {
        return heap.size() == 0;
    }

    // Retrieves the minimum element in the heap.
    // In a min-heap, the smallest element is stored at the root (index 0).
    public T getMin() {
        if (isEmpty()) {
            throw new IllegalStateException("Heap is empty");
        }
        return heap.get(0);
    }

    // Adds a new element to the heap.
    // The element is first added to the end of the array (to keep the tree complete)
    // and then moved upward (percolated up) to restore the heap order property.
    public void add(T element) {
        heap.add(element);
        // After adding, restore the min-heap structure by percolating the new element upward.
        percolateUp();
    }

    // Percolates the last element added upward to maintain the heap property.
    // Checks the element against its parent, swapping if necessary.
    private void percolateUp() {
        // Start with the index of the newly added element.
        int child = heap.size() - 1;
        // Calculate the parent's index using integer division.
        int parent = (child - 1) / 2;

        // Continue to swap while not at the root and while the current node is smaller than its parent.
        while (child > 0 && heap.get(child).compareTo(heap.get(parent)) < 0) {
            // Swap the child with its parent to move the smaller element up.
            swap(child, parent);

            // Update the child index to continue percolating up.
            child = parent;
            parent = (child - 1) / 2;  // Recalculate the new parent's index.
        }
    }
```

```java

    // Removes and returns the minimum element from the heap.
    // The procedure:
    // 1. Replace the root with the last element.
    // 2. Remove the last element.
    // 3. Percolate the new root down to restore the heap property.
    public T removeMin() {
        if (isEmpty()) {
            throw new IllegalStateException("Heap is empty");
        }
        // Store the minimum element (root) to return later.
        T min = heap.get(0);
        // Remove the last element from the heap.
        T last = heap.remove(heap.size() - 1);
        if (!heap.isEmpty()) {
            // Place the last element at the root.
            heap.set(0, last);
            // Restore the min-heap property by percolating down.
            percolateDown(0);
        }
        return min;
    }

    // Percolates the element at the given index down to its proper position.
    // At each step, it swaps the parent with the smallest of its children if
    ↪ necessary.
    private void percolateDown(int parent) {
        // Compute the indices of the left and right children.
        int leftChild = 2 * parent + 1;
        int rightChild = 2 * parent + 2;
        // Assume initially that the parent is the smallest.
        int smallest = parent;

        // Check if the left child is within bounds and smaller than the parent.
        if (leftChild < heap.size() &&
        ↪ heap.get(leftChild).compareTo(heap.get(smallest)) < 0) {
            smallest = leftChild;
        }
        // Check if the right child exists and is smaller than the smallest so far.
        if (rightChild < heap.size() &&
        ↪ heap.get(rightChild).compareTo(heap.get(smallest)) < 0) {
            smallest = rightChild;
        }
        // If a child is smaller than the parent, perform a swap and continue
        ↪ percolating down.
        if (smallest != parent) {
            swap(parent, smallest);
            // Recursively percolate the element down further.
```

```
                percolateDown(smallest);
            }
        }

        // Helper method to swap two elements in the heap given their indices.
        private void swap(int i, int j) {
            T temp = heap.get(i);
            heap.set(i, heap.get(j));
            heap.set(j, temp);
        }
    }
```

4 Time Complexity

- Percolating **one level** - $\Theta(1)$
- The height of the tree - $\Theta(log(n))$
- Percolating up and down n times - $\Theta(log(n))$
- Add / RemoveMin are - $\Theta 1$

5 Heap Sort

- Add each element to the heap
- Then pop the root node each time and add it to a new array
- This array will be sorted ascending

5.1 Time complexity

- Worst case time complexity is:
$$\mathcal{O}(n \cdot log(n))$$
  - We have to add n elements then remove n elements
  - Each add/remove is $\mathcal{O}(log(n))$ as you are percolating up/down the whole height of the tree:
    * Height of tree is $\Theta(log(n))$
    * Each percolation is $\Theta(1)$