Priority Queues (Heaps)

The Model
A priority queue is a queue where:
- Requests are inserted in the order of arrival
- The request with highest priority is processed first (deleted from the queue)
- The priority is indicated by a number. The lower the number - the higher the priority

Implementations
- Linked list:
  - Insert at the beginning - $O(1)$
  - Find the minimum - $O(N)$
- Binary search tree (BST):
  - Insert $O(\log N)$
  - Find minimum - $O(\log N)$
**Binary Heap**

Better than BST because it does not require links
- Insert: $O(\log N)$
- Find minimum: $O(\log N)$

Deleting the minimal element takes a constant time. However, after that the heap structure has to be adjusted, and this requires $O(\log N)$ time.

Requires that the heap size is (approximately known) in advance.

---

**Binary Heap**

**Heap-Structure Property:**
- Complete Binary Tree - Each node has two children, except for the bottom level, which is filled from left to right.
- The nodes at the last level do not have children. New nodes are inserted at the last level from left to right.

**Heap-Order Property:**
- Each node has a higher priority than its children.

---

**Binary Heap Implementation with an Array**

Root - $A(1)$
- Left Child of $A(i)$ - $A(2i)$
- Right child of $A(i)$ - $A(2i+1)$
- Parent of $A(i)$ - $A(\lceil i/2 \rceil)$

The smallest element is always at the root, the access time to the element with highest priority is constant $O(1)$. 

---

Next node to be inserted - right child of the yellow node.
Example

Consider 17:
position in the array: 5
parent 10 is at position \(\lfloor 5/2 \rfloor = 2\)
left child is at position 5*2 = 10 (this is 34)
right child - position 2*5 + 1 = 11 (empty)

Problems

Problem 1:
Reconstruct the binary heap

Problems

Problem 2: Give the array representation for

Basic Operations

• Insert a node - Percolate Up
• Delete a node - Percolate Down
• Decrease key, Increase key, Remove key
• Build the heap
Percolate Up - Insert a Node

A hole is created at the bottom of the tree, in the next available position.

Percolate Up

Insert 20

Complexity of insertion: $O(\log N)$
Percolate Down – Delete a Node

The empty hole violates the heap-structure property

Percolate Down – The Wrong Way

Last element - 20. The hole at the root

We try to insert 20 in the hole by percolating the hole down
Percolate Down

Complexity of deletion: $O(\log N)$

Other Heap Operations

1. DecreaseKey$(p, d)$
   
   Increase the priority of element $p$ in the heap with a positive value $d$ (percolate up)

2. IncreaseKey$(p, d)$
   
   Decrease the priority of element $p$ in the heap with a positive value $d$ (percolate down)
Other Heap Operations

3. Remove(p)
   a) Assigning the highest priority to p - percolate p up to the root
   b) Deleting the element in the root and filling the hole by percolating down and trying to insert the last element in the queue

4. BuildHeap
   - Input N elements
   - Place them into an empty heap through successive inserts. The worst case running time is $O(N \log N)$

Build Heap - $O(N)$

Given an array of elements to be inserted in the heap,
- treat the array as a heap with order property violated,
- and then do operations to fix the order property

Example

150 80 40 30 10 70 110 100 20 90 60 50 120 140 130

Example (cont’d)

After processing height 1
Theorem

- For a perfect binary tree of height $h$ containing $N=2^{h+1}-1$ nodes,
  - the sum $S$ of the heights of the nodes is $S=2^{h+1}-1-(h+1)=O(N)$

Proof

The tree has 1 node at height $h$, 2 nodes at height $h-1$, 4 nodes at height $h-2$, etc.

1 ($2^0$)  |  h
2 ($2^1$)  |  h-1
4 ($2^2$)  |  h-2
8 ($2^3$)  |  h-3
...........

$2^h$  |  0

$S = \sum 2^i(h-i), \ i=0 \ to \ h$
Proof (cont’d)

\[ S = h + 2(h-1) + 4(h-2) + 8(h-3) + \ldots 2^{(h-2)} \cdot 2 + 2^{(h-1)} \cdot 1 \]  
\[ (1) \]

\[ 2S = 2h + 4(h-1) + 8(h-2) + 16(h-3) + \ldots 2^{(h-1)} \cdot 2 + 2^h \cdot 1 \]  
\[ (2) \]

Subtract (1) from (2):

\[ S = (h-2h) + (2 + 4 + 8 + \ldots + 2^h) \]
\[ = -h + (2^{h+1} - 2) \]
\[ = -h - 1 + (2^{h+1} - 1) \]
\[ = (2^{h+1} - 1) - (h+1) \]

Proof (cont’d)

Note:

\[ 1 + 2 + 4 + 8 + \ldots 2^h = (2^{h+1} - 1) \]

Hence \[ S = (2^{h+1} - 1) - (h + 1) \]

Hence the complexity of building a heap with \( N \) nodes is linear \( O(N) \)

The Selection Problem

- Given a list of \( N \) elements and an integer \( k, k \leq N \), find the \( k^{th} \) largest element

Applications of Priority Queues
Solution 1

- Sort the elements in an array
- Return the element in the $k^{th}$ position

**Complexity:**
- Simple sorting algorithm $O(N^2)$

Solution 2

- Read $k$ elements in an array, sort them
- Let $S_k$ be the smallest element
- For each next element $E$ do the following:
  - If $E > S_k$
    - Remove $S_k$
    - Insert $E$ in the appropriate position in the array
- Return $S_k$

**Complexity:**
- $O(N*k)$
  - $k^2$ for the initial sorting of $k$ elements
  - $(N-k)*k$ for inserting each next element
    - $O(k^2)$ + $O((N-k)*k)$ = $O(k^2)$ + $O(Nk)$
  - Worst case: $k = N/2$, complexity: $O(N^2)$

Solution 2 (cont’d)

- Complexity: $O(N*k)$
  - $k^2$ for the initial sorting of $k$ elements
  - $(N-k)*k$ for inserting each next element
    - $O(k^2) + O((N-k)*k)$
      - $O(k^2 + Nk - k^2)$ = $O(Nk)$
  - For $k = N/2$, complexity: $O(N^2)$

Solution 3

Assume we change the heap-order property - the highest priority corresponds to the highest key value.

- Read $N$ elements in an array
- Build a heap of these $N$ elements - $O(N)$
- Perform $k$ *DeleteMax* operations - $O(k \log N)$

**Complexity:** $O(N) + O(k \log N)$

For $k = N/2$ the complexity is $O(N \log N)$
Solution 4

We return back to the usual heap-order property
- Build a heap of $k$ elements. Complexity $O(k)$
- The $k^{th}$ largest element among $k$ elements is the smallest element in that heap and it will be at the top
- Compare each next element with the top element $O(1)$
- If the new element is larger, DeleteMin the top element and insert the new element in the heap - $O(\log(k))$

Solution 4 (cont’d)

- At the end of the input the smallest element in the heap is the $k^{th}$ largest element in the list of $N$ elements

Complexity:
- $O(k) + O((N-k) \times \log(k)) = O(N\log(k))$

d-Heap

- Insert operation is $O(\log_d n)$
- DeleteMin operation is $O(d\log_d n)$
- Because the minimum of $d$-children should be found which takes $(d-1)$ comparison.
- For practical applications where the number of insertions is much greater than that of deletions, $d$-heap design may be performed to speed-up...
- But, both find operation and combining two $d$-heap are rather difficult compared to binary heap...
Leftist Heaps

- Idea: merging requires copying one array into another, which takes $O(N)$ time for equal-sized heaps...
- Can we do it better for mergeable priority queues?
  - Leftist heaps: a special form of binary heaps that attempt to be very unbalanced....
  - Establish a binary tree that is formed on the left path so that insert and merge operations can be easily done at the right path...
  - Playing on the right requires reforming the leftist property...

Null Path Length

- Null path length, $npl(X)$: Length of the shortest path from $X$ to a node without two children
- Null path length of a node with 0 or 1 child is 0
- $npl(\text{null})$ is -1

Null Path Length

- The one on the left is a leftist heap; the other is not

Leftist Heap

- Remark: A leftist heap is a tree in which the shortest path to an external node is always on the right
  - i.e., for every node in the heap, the null path length of the left child is AT LEAST as large as that of the right child
Leftist Heap

• **Theorem:** Consider a leftist tree \( T \) which contains \( n \) internal nodes. The path leading from the root of \( T \) downwards to the rightmost external node contains at most \( \lceil \log(n+1) \rceil \) nodes.

Merging Leftist Heaps

• Merge these two heaps
  - Recursively merge the heap with the larger root with the right subheap with the smaller root.

Merging Leftist Heaps: Intermediate Step

After merging \( H_2 \) with \( H_1 \)'s right subheap

After attaching previous figure as \( H_1 \)'s right child
Merging Leftist Heaps:
Final Result

After swapping children of $H_1$'s root

Leftist Heap: Complexity

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>merge</td>
<td>$O(\log n_1 + \log n_2)$</td>
</tr>
<tr>
<td>enqueue</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>dequeue</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>findMin</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>