Data Structures

Topics Covered:

Amortized Analysis:

- Aggregate Method
- Banker’s Method
- Physicit’s Method

Priority Queues

Binary- max heap

Let c_i = cost of i^{th} insertion.

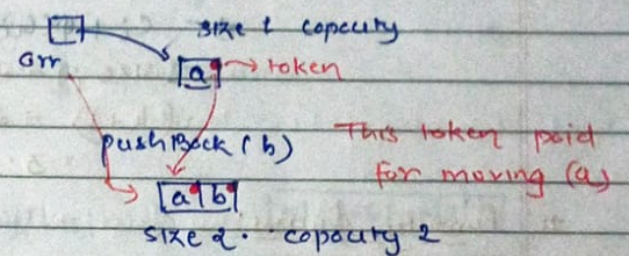
$$c_i = 1 + \begin{cases} i-1 & \text{if } i-1 \text{ is a power of 2} \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\sum_{i=1}^{n} c_i}{n} = \frac{n + \sum_{j=1}^{\lfloor \log_2 (n-1) \rfloor} 2^j}{n} \Rightarrow \frac{O(n)}{n} \Rightarrow O(1)$$

⇒ Amortized Analysis { Banker's Method }

- charge extra for each cheap operation.
- Save the extra charge as tokens in your data structure
- Use the tokens for expensive operations (conceptually)

ex



⇒ Physicist's Method:

Define a potential function, ϕ which maps states of the data structures to integers

- $\phi(h_0) = 0$
- $\phi(h_t) \geq 0$

- amortized cost for operation:

$$c_t + \phi(h_t) - \phi(h_{t-1})$$

- The cost of n-operations is $\sum_{i=1}^{n} c_i$
- The sum of the amortized cost is:

$$\sum_{i=1}^{n} (c_i + \phi(h_i) - \phi(h_{i-1})) = \phi(h_n) - \phi(h_0) + \sum_{i=0}^{n} c_i$$

- Dynamic array: n calls to PushBack.

Let $\phi(h_i)$ = 2 × size − capacity

- $\phi(h_0) = 2 \times 0 - 0 = 0$
- $\phi(h_i)$ = 2 × size − capacity > 0

since size > $\frac{\text{capacity}}{2}$

- without resize when adding element i

Amortized cost of adding element i:

$\underbrace{c_i} + \phi(h_i) - \phi(h_{i-1})$

$1 + 2 \times size - cop_i - (2 \times size_{i-1} - cop_{i-1})$

$1 + 2 \times (size_i - size_{i-1})$

$\Rightarrow 3$ {same as Banker's method}

- with resize when adding element i

let $k = size_{i-1} = cop_{i-1}$

Then

$\phi(h_{i-1}) = 2 size_{i-1} - cop_{i-1} = 2k - k = k$

$\phi(h_i) = 2 size_i - cop_i = 2(k+1) - 2k = 2$

$\Rightarrow c_i + \phi(h_i) - \phi(h_{i-1})$

$(size_i) + 2 - k$

$= (k+1) + 2 - k$

$= 3.$

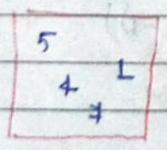
Priority Queues:

Typical Use Case: {Scheduling Jobs}

- Want to process jobs one by one in order of decreasing priority. While the current job is processed, new jobs may arrive.
- To add a job to the set of scheduled jobs, call Insert(job).
- To process a job with the highest priority, get it by calling ExtractMax().

Toy Example

Content



Queries

ExtractMax() → 7

Insert(1)

Insert(7)

Additional Operations:

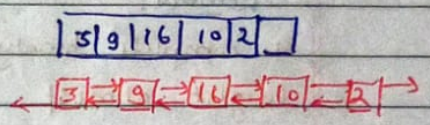
- Remove (it) removes an element pointed by an iterator it
- GetMax () returns an element with maximum priority (without changing the set of elements)
- ChangePriority (it, p) changes the priority of an element pointed by it on p

Algorithms that Use Priority Queues:

- Dijkstra's algorithm: finding a shortest path in a graph
- Prim's algorithm: constructing a minimum spanning tree of a graph
- Huffman's algorithm: constructing an optimum prefix-free encoding of a string.
- Heap sort: sorting a given sequence.

Naive Implementation:

Unsorted Array/List-

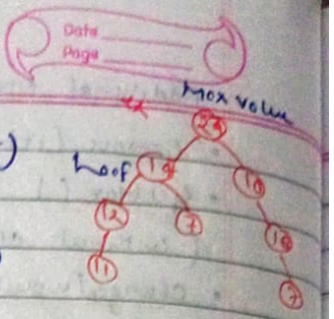


- Insert (e)
  - add e to the end
  - running time: O(1)
- ExtractMax()
  - Scan the array
  - Running time: O(n)
    - Sorted Array:
      - ExtractMax(): running time: O(1)

disadvantage: - Insert (e): running time: O(n)

But inserting in between O(1)

Binary max-heap: is a binary tree (each node has zero, one, or two children) where the value of each node is at least the value of its children.



Basic Operations: Get Max → O(1)

→ Insert: running time : O(tree Height)
shifting:

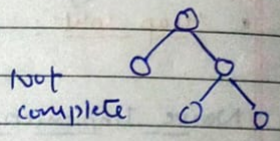
→ ExtractMax: Running time: O(tree height)

→ change Priority: running time: O(tree. height)

→ Remove: change Priority to ∞
Then call Extract-Max
running time: O(tree height)

Complete Binary Tree

A binary tree is complete if all its levels are filled except possibly the last one which is filled from left to right.



First: Advantage: Low Height

Lemma:
A complete binary tree with n-nodes has height atmost $O(\log n)$

Second Advantage: Store as Array.

$parent(i) = \lfloor \frac{i}{2} \rfloor$ $\quad leftchild(i) = 2i$

$rightchild(i) = 2i+1$

↗ what do we pay for these advantages?

→ We need to keep the tree complete

→ which binary heap operations modify the shape of the tree?

→ Only Insert and ExtractMax (Remove changes the shape by calling ExtractMax)

↗ to extract the maximum value, replace the root by the last leaf and let it shift down.

Pseudo Code:

```
SiftUp(i):
while i > 1 and H[Parent(i)] < H[i]:
    swap H[Parent(i)] and H[i]
    i ← Parent(i)

ShiftDown(i)
maxIndex ← i
l ← leftChild(i)
if l ≤ size and H[l] > H[maxIndex]:
    maxIndex ← l
r ← RightChild(i)
if r ≤ size and H[r] > H[maxIndex]:
    maxIndex ← r
if i ≠ maxIndex:
    swap H[i] and H[maxIndex]
    ShiftDown(maxIndex)
```

```
# Insert(p)
If size = maxSize:
    return ← H[size] ERROR
size ← size + 1
H[size] ← p
Shift-Up(size)
```

```
ExtractMax()
result ← H[1]
H[1] ← H[size]
size ← size - 1
SiftDown(1)
return result
```

```
# Remove(i)
H[i] ← ∞
SiftUp(i)
ExtractMax()
```

```
ChangePriority(i, p)
oldp ← H[i]
H[i] ← p
if p > oldp:
    SiftUp(i)
else:
    SiftDown(i)
```