Finite Geometric Series

If \( a > 1 \)

\[
\sum_{i=0}^{L} a^i
\]
Finite Geometric Series

If $a < 1$

The series multiplied by $a$

$(1 + a + a^2 + \cdots + a^L)a$

The series

$(1 + a + a^2 + \cdots + a^L)1$

The next term in the series

$a^{L+1}$

The first term

1

Solve for the series

\[
\sum_{i=0}^{L} a^i
\]
ADT: Queue

• What is it?
  • A “First In First Out” (FIFO) collection of items

• What Operations do we need?
  • Enqueue
    • Add a new item to the queue
  • Dequeue
    • Remove the “oldest” item from the queue
  • IsEmpty
    • Indicate whether or not there are items still on the queue
ADT: Priority Queue

• What is it?
  • A collection of items and their “priorities”
  • Allows quick access/removal to the “top priority” thing
    • Usually a smaller priority value means the item is “more important”

• What Operations do we need?
  • insert(item, priority)
    • Add a new item to the PQ with indicated priority
  • extract
    • Remove and return the “top priority” item from the queue
      • Usually the item with the smallest priority value
  • isEmpty
    • Indicate whether or not there are items still on the queue

• Note: the “priority” value can be any type/class so long as it’s comparable (i.e. you can use “<“ or “compareTo” with it)
Priority Queue, example

PriorityQueue PQ = new PriorityQueue();
PQ.insert(5,5)
PQ.insert(6,6)
PQ.insert(1,1)
PQ.insert(3,3)
PQ.insert(8,8)  // PQ has 5, 6, 3, and 8
Print(PQ.extract())  // 1
Print(PQ.extract())  // 3
Print(PQ.extract())  // 5
Print(PQ.extract())  // 6
Print(PQ.extract())  // 8
Priority Queue, example

PriorityQueue PQ = new PriorityQueue();
PQ.insert(5,5)
PQ.insert(6,6)
PQ.insert(1,1) // 5, 6, 1
Print(PQ.extract()) // 1
PQ.insert(3,3) // 5, 6, 3
Print(PQ.extract()) // 3
Print(PQ.extract()) // 5
PQ.insert(8,8) // 6, 8
Print(PQ.extract()) // 6
Print(PQ.extract()) // 8
Applications?

- ER
- Sorting
- Corrupt restaurant
- Restaurants with reservations
- Presales – priority registration
- Operating systems – managing tasks
- Airline boarding
- Distributing tasks
- Huffman
Thinking through implementations

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Worst case time to insert</th>
<th>Worst case time to extract</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Array</td>
<td></td>
<td>h</td>
</tr>
<tr>
<td>Unsorted Linked List</td>
<td></td>
<td>h</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For simplicity, Assume we know the maximum size of the PQ in advance (otherwise we’d do an amortized analysis, but get the same answers...
## Thinking through implementations

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Worst case time to insert</th>
<th>Worst case time to extract</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Array</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Unsorted Linked List</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

For simplicity, assume we know the maximum size of the PQ in advance (otherwise we’d do an amortized analysis, but get the same answers...)
### Thinking through implementations

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Worst case time to insert</th>
<th>Worst case time to extract</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Array</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Unsorted Linked List</td>
<td>$\Theta(1)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
</tbody>
</table>

For simplicity, assume we know the maximum size of the PQ in advance (otherwise we’d do an amortized analysis, but get the same answers...).
Trees for Heaps

- **Binary Trees:**
  - The branching factor is 2
  - Every node has \( \leq 2 \) children

- **Complete Tree:**
  - All “layers” are full, except the bottom
  - Bottom layer filled left-to-right
Heap – Priority Queue Data Structure

• Idea: We need to keep some ordering, but it doesn’t need to be entirely sorted
• $\Theta(\log n)$ worst case for extract and insert
Heap – Priority Queue Data Structure

• Idea: We need to keep some ordering, but it doesn’t need to be entirely sorted
• $\Theta(\log n)$ worst case for extract and insert
Challenge!

• What is the maximum number of total nodes in a binary tree of height $h$?
  • $2^{h+1} − 1$
  • $\Theta(2^h)$

• If I have $n$ nodes in a binary tree, what is its minimum height?
  • $\Theta(\log n)$

• Heap Idea:
  • If $n$ values are inserted into a complete tree, the height will be roughly $\log n$
  • Ensure each insert and extract requires just one “trip” from root to leaf
(Min) Heap Data Structure

- Keep items in a complete binary tree
- Maintain the “(Min) Heap Property” of the tree
  - Every node’s priority is $\leq$ its children’s priority
  - Max Heap Property: every node’s priority is $\geq$ its children
- Where is the min?
- How do I insert?
- How do I extract?
- How to do it in Java?
Heap Insert

insert(item, priority) {
    put item in the “next open” spot (keep tree complete)
    while (priority < parent’s priority) {
        swap item with parent
    }
}
Heap Insert

```
insert(item, priority)
    put item in the “next open” spot (keep tree complete)
    while (priority < parent’s priority)
        swap item with parent
```

Heap Insert

```java
insert(item, priority) {
    put item in the “next open” spot (keep tree complete)
    while (priority < parent’s priority) {
        swap item with parent
    }
}
```
Heap Insert

\[\text{insert}(\text{item}, \text{priority})\{
    \text{put item in the “next open” spot (keep tree complete)}
    \text{while (priority < parent’s priority)}\{
        \text{swap item with parent}
    \}
}\]
Heap Insert

insert(item, priority){
    put item in the “next open” spot (keep tree complete)
    while (priority < parent’s priority){
        swap item with parent
    }
}
Heap extract

```java
extract()
{
    min = root
    curr = bottom-right item
    move curr to the root
    while(curr > curr.left || curr > curr.right){
        swap curr with its smallest child
    }
    return min
}
```
Heap extract

```java
extract()
{
    min = root
    curr = bottom-right item
    move curr to the root
    while(curr > curr.left || curr > curr.right){
        swap curr with its smallest child
    }
    return min
}
```
Heap extract

extract()
{
    min = root
    curr = bottom-right item
    move curr to the root
    while(curr > curr.left || curr > curr.right){
        swap curr with its smallest child
    }
    return min
}
Heap extract

effect() {
    min = root
    curr = bottom-right item
    move curr to the root
    while (curr > curr.left || curr > curr.right) {
        swap curr with its smallest child
    }
    return min
}
Heap extract

extract()

    min = root
    curr = bottom-right item
    move curr to the root
    while(curr > curr.left || curr > curr.right){
        swap curr with its smallest child
    }

return min
Percolate Up and Down (for a Min Heap)

• Goal: restore the “Heap Property”

• Percolate Up:
  • Take a node that may be smaller than a parent, repeatedly swap with a parent until it is larger than its parent

• Percolate Down:
  • Take a node that may be larger than one of its children, repeatedly swap with smallest child until both children are larger

• Worst case running time of each:
  • $\Theta(\log n)$
Representing a Heap

• Every complete binary tree with the same number of nodes uses the same positions and edges
• Use an array to represent the heap
• Index of root: \[ \lfloor \frac{n}{2} \rfloor \]
• Parent of node \( i \): \[ \lfloor \frac{i}{2} \rfloor \]
• Left child of node \( i \):
• Right child of node \( i \):
• Location of the leaves:
insert(item){
    if(size == arr.length − 1){resize();}
    size++;
    arr[size] = item;
    percolateUp(size)
}
Percolate Up

```java
percolateUp(int i){
    int parent = i/2;    // index of parent
    Item val = arr[i];  // value at current location
    while(i > 1 && arr[i] < arr[parent]){    // until location is root or heap property holds
        arr[i] = arr[parent];    // move parent value to this location
        arr[parent] = val;    // put current value into parent’s location
        i = parent;    // make current location the parent
        parent = i/2;    // update new parent
    }
}
```
extract Psuedocode

extract()
{
    theMin = arr[1];
    arr[1] = arr[size];
    size--;
    percolateDown(1);
    return theMin;
}
Percolate Down

```java
percolateDown(int i) {
    int left = i*2; \ index of left child
    int right = i*2+1; \ index of right child
    Item val = arr[i]; \ value at location
    while(left <= size){ \ until location is leaf
        int toSwap = right;
        if(right > size || arr[left] < arr[right]){ \ if there is no right child or if left child is smaller
            toSwap = left; \ swap with left
        } \ now toSwap has the smaller of left/right, or left if right does not exist
        if (arr[toSwap] < val){ \ if the smaller child is less than the current value
            arr[i] = arr[toSwap];
            arr[toSwap] = val; \ swap parent with smaller child
            i = toSwap; \ update current node to be smaller child
            left = i*2;
            right = i*2+1;
        }
    }
    else{ return;} \ if we don’t swap, then heap property holds
}
```
Other Operations

• Increase Key
  • Given the index of an item in the PQ, make its priority value larger
    • Min Heap: Then percolate Down
    • Max Heap: Then percolate Up

• Decrease Key
  • Given the index of an item in the PQ, make its priority value smaller
    • Min Heap: Then percolate Up
    • Max Heap: Then percolate Down

• Remove
  • Given the item at the given index from the PQ
Aside: Expected Running time of Insert
Building a Heap From “Scratch”

• Suppose we had $n$ items and wanted to “heapify” them

Violate Heap Property!

Two ways for “fix” the heap:
1) Percolate Up
2) Percolate Down
Floyd’s buildHeap method

• Working towards the root, one row at a time, percolate down

```java
buildHeap()
    for(int i = size; i>0; i--)
        percolateDown(i);
```

Floyd’s buildHeap method

• Suppose we had $n$ items and wanted to “heapify” them

```
buildHeap()
{
    for(int i = size; i>0; i--)
    {
        percolateDown(i);
    }
}
```
Floyd’s buildHeap method

• Suppose we had $n$ items and wanted to “heapify” them

```
buildHeap()
{
    for (int i = size; i > 0; i--)
    {
        percolateDown(i);
    }
}
```

Violate Heap Property!
Nodes bigger than a child
Floyd’s buildHeap method

• Suppose we had $n$ items and wanted to “heapify” them

```
buildHeap()
{
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```
Floyd’s buildHeap method

• Suppose we had $n$ items and wanted to “heapify” them

```java
buildHeap()
{
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```
Floyd’s buildHeap method

• Suppose we had $n$ items and wanted to “heapify” them

```
buildHeap()
{
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```
Floyd’s buildHeap method

- Suppose we had $n$ items and wanted to “heapify” them

```
buildHeap()
{
  for(int i = size; i>0; i--){
    percolateDown(i);
  }
}
```
How long did this take?

• Worst case running time of buildHeap:
  • No node can percolate down more than the height of its subtree
    • When i is a leaf:
    • When i is second-from-last level:
    • When i is third-from-last level:

• Overall Running time:

```java
buildHeap()
{  
    for(int i = size; i>0; i--){
        percolateDown(i);
    }
}
```