Heap in an array

Heap stored in an array

children

Heap stored in an array
Heap repair

Top removed (1)

1. Remove top

2. last → first

3. Put at the top

\[ U > B, \ U > D, \ B < D \]

⇒ swap B ↔ U
Heap repair

Top removed (2)

3 Put at the top - cont...

U > M, U > J, J < M
⇒ swap J ↔ U

U > K, U > R, K < R
⇒ swap K ↔ U
Heap repair

Top removed (3)

3. Put at the top - done.

New heap
Make a heap -- Heapify

Array

Currently created heap

Moves
Make a heap -- Heapify

Array

1. R
2. J
3. U
4. Z
5. A
6. D
7. E
8. M
9. O
10. K
11. B
12. T

Earlier heap(s)

Currently created heap

Moves

for PAL2020
Make a heap -- Heapify

Array

1. R
2. J
3. U
4. M
5. A
6. D
7. E
8. Z
9. O
10. K
11. B
12. T

Currently created heap

Earlier heap(s)

Moves

Array 1 2 3 4 5 6

Currently created heap

Earlier heap(s)

Moves

Currently created heap

Earlier heap(s)

Moves

Currently created heap

Earlier heap(s)

Moves

Currently created heap

Earlier heap(s)

Moves

Currently created heap

Earlier heap(s)

Moves

Currently created heap

Earlier heap(s)

Moves
Make a heap -- Heapify

Array

Currently created heap

Earlier heap(s)

Moves
### Make a heap -- Heapify

#### Array

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R</td>
<td>A</td>
<td>D</td>
<td>M</td>
<td>B</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>O</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Currently created heap

1. A
2. M
3. B
4. D
5. Z
6. O

#### Earlier heap(s)

1. R
2. J
3. E

#### Moves

1. A → E
2. M → B
3. Z → D
4. O → A
5. K → J
6. U → 12

---

Currently created heap

Earlier heap(s)

Make a heap -- Heapify

for PAL2020

A4B33ALG 2011/08
Make a heap -- Heapify

Array

Currently created heap

Moves

For PAL2020
def repairTop (arr, top, bottom):
    i = top    # arr[2*i] and arr[2*i+1]
    j = i*2    # are successors of arr[i]
    topVal = arr[top]
    # try to find a successor < topVal
    if j < bottom and arr[j] > arr[j+1]: j += 1
    # while successors < topVal move successors up
    while j <= bottom and topVal > arr[j]:
        arr[i] = arr[j]
        i = j; j = j*2    # move to next successor
        if j < bottom and arr[j] > arr[j+1]: j += 1
    # put topVal to its correct place
    arr[i] = topVal

def heapify (arr):
    n = len(arr)-1
    for i in range(n//2, 0, -1):    # progress backwards!
        repairTop(arr, i, n)
Priority queue implemented with binary heap -- Insert

Insert the element at the end of the queue (end of the heap).

In most cases, this violates the heap property and the heap has to be repaired.
Priority queue implemented with binary heap -- Insert

Insert A

Heap property is violated, swap the element with its parent.

Heap property is still violated, swap the element with its parent.
Inserting A

Heap property is still violated, swap the element with its parent.

Heap property is respected, the inserted element has found its place in the queue (heap).

Priority queue implemented with binary heap -- Insert
Binary heap -- Insert element more effectively

Insert A

Do not insert the element at the end of the queue. First, find its place and while searching move down other elements encountered in the search.

Finally, store the inserted element at its correct position.
Inserting represents a traversal in a binary tree from a leaf to the root in the worst case. Therefore, the Insert complexity is $O(\log_2(n))$. 

```python
# beware! array is arr[1] ... arr[n]
# bottom == ndx of last elem

def heapInsert(arr, x, bottom):
    bottom += 1  # expand the heap space
    j = bottom
    i = j/2  # parent index

    while i > 0 and arr[i] > x:
        arr[j] = arr[i]  # move elem down the heap
        j = i; i /= 2  # move indices up the heap

    arr[i] = x  # put inserted elem to its place
    return bottom
```