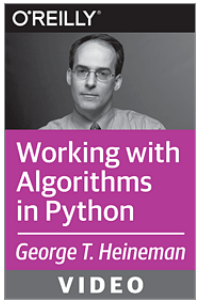

O'REILLY®

Heap Data Structure

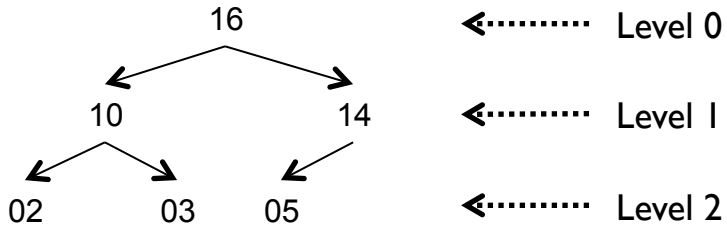


Heap
Data Structure

- Heap data structure based on Binary Tree
  - Can be used in HEAPSORT
  - Can implement priority queue
- Useful data structure to learn on its own right

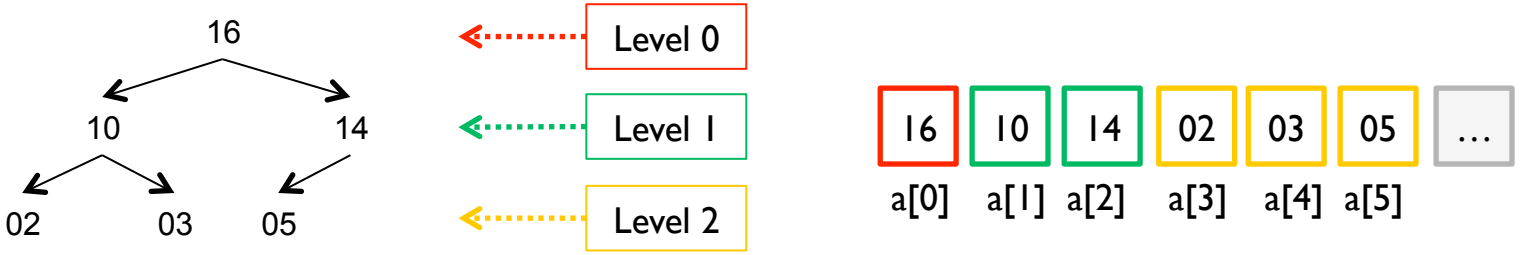
Heap
Data Structure

- A Heap is a Binary Tree that ensures two properties
  - *Shape* – Binary Tree grows by filling levels left to right
    No nodes on Level *n* until *n* – *1* completely filled
  - *Heap* – A node's value is ≥ value of either of its children



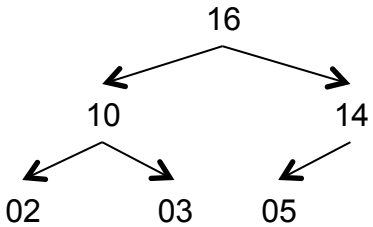
Heap
Data Structure

- Rigid structure ensures heap can be stored in array
  - Root value stored at index a[0]
  - Children for node a[*i*] are found at a[*2*i+1*] and a[*2*i+2*]



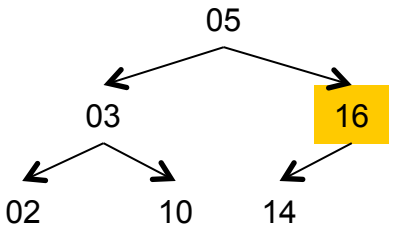
What is a Heap Good For?

- In constant time you can find the maximum value
- You can construct one “in place” in an array
  - No extra storage required
  - Makes it possible to code HEAPSORT



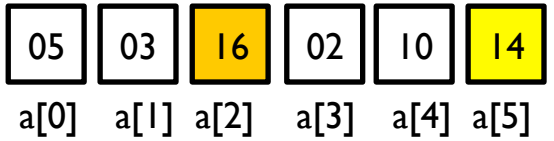
How To Build A Heap From Array

- From initial array adjust each parent node with child
  - Work in reverse order from “last parent” to root
  - Position $\frac{n}{2} - 1$ is the “last parent”
  - “Heapify” as necessary



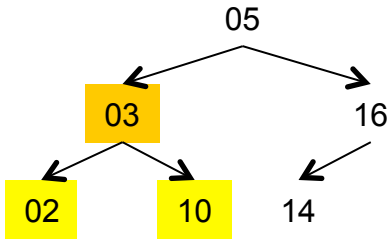
Not Yet A Heap!

initial array

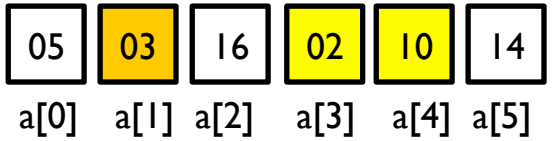


How To Build A Heap From Array

- Adjust each parent node that has a child
  - Fix nodes that violate *Heap* property

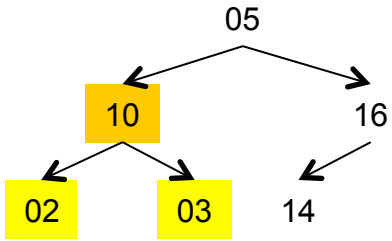


Must Adjust

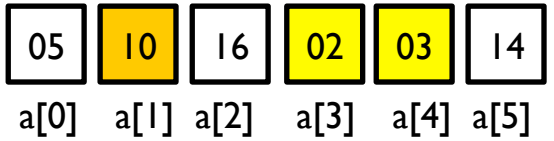


How To Build A Heap From Array

- Adjust each parent node that has a child
  - Note swapping will never violate *Shape* property

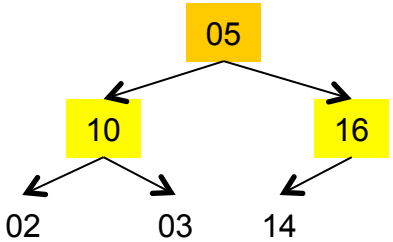


Swap Larger

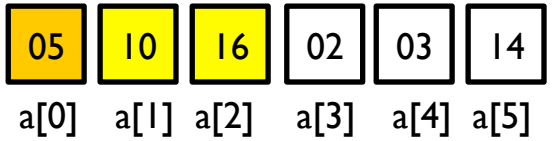


How To Build A Heap From Array

- Adjust each parent node that has a child
  - May have to propagate changes down
  - But since balanced, never more than O(log n) levels

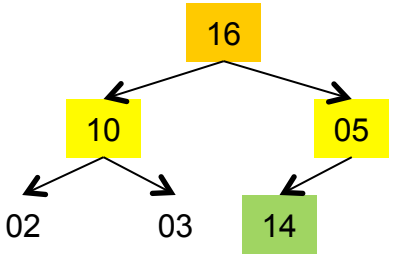


Swap Larger

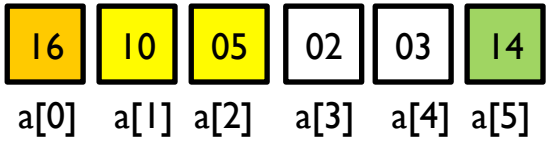


How To Build A Heap From Array

- Adjust each parent node that has a child
  - Propagate change down to third level



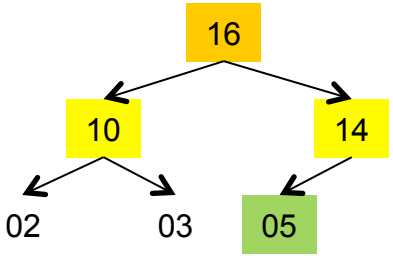
Check Down!



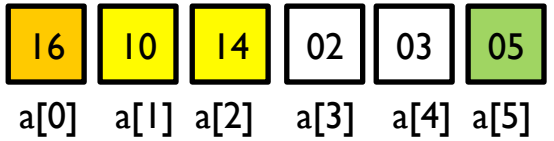
How To Build A Heap From Array

Build Heap From Array

```
def buildHeap(A):
    n = len(A)
    for i in range(n/2-1, -1, -1):
        heapify(A, i, n)
```



Heap!



Heapify Definition

Encodes process as described

Swap larger of two children and propagate change down as needed

Heapify

```
def heapify (A, idx, maxIdx):
    left = 2*idx+1
    right = 2*idx+2
    if left < maxIdx and A[left] > A[idx]:
        largest = left
    else:
        largest = idx
    if right < maxIdx and A[right] > A[largest]:
        largest = right

    if largest != idx:
        A[idx],A[largest] = A[largest],A[idx]
        heapify(A, largest, maxIdx)
```

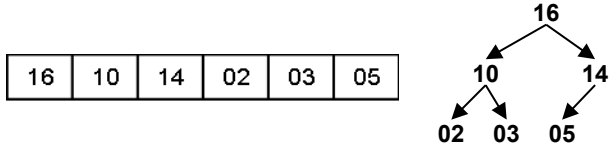
HeapSort Definition

HeapSort Array

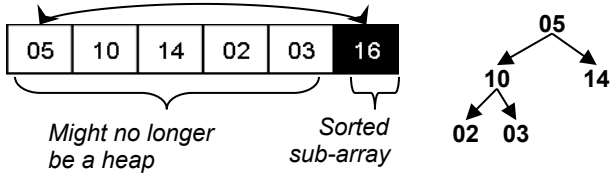
```
def heapSort(A):
    buildHeap(A)
    for i in range(len(A)-1, 0, -1):
        A[0],A[i] = A[i],A[0]
        heapify(A, 0, i)
```

For loop repeatedly invokes heapify until entire list is sorted, leads to O(n log n) behavior

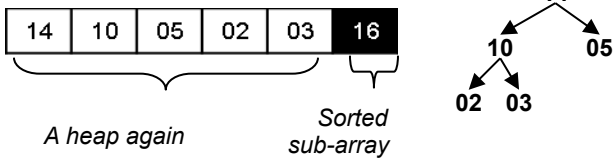
After buildHeap



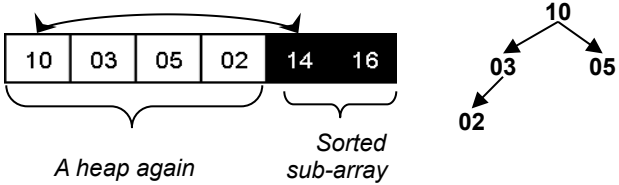
After 1st swap



After 1st heapify



After 2nd swap



Heap Problem

- Find k^{th} smallest element in collection
  - Can sort and locate it, but isn't that overkill?
  - Let's use Heap and show in Python code
- Basic strategy
  - Create Max heap from first *k* elements in collection
  - For each remaining element, if smaller than root, replace and heapify

Heap & HeapSort Summary

- Heap is a versatile structure
  - Because of properties associated with its recursive structure
- Use in other algorithms
  - Prim's minimal-spanning-tree algorithm
  - Dijkstra's Single-Source Shortest Path algorithm
  - Efficiently compute k^{th} largest item in collection