For:

B-tree T of height h containing $n \geq 1$ keys and a minimum degree $t \geq 2$

then $h \leq \log_t \left(\frac{n+1}{2}\right)$

In which case $n \geq 1 + (t-1) \sum_{i=1}^{h} 2t^{i-1}$

$$\boxed{n \geq 2t^h - 1}$$

B-tree operations (continued)

- Deletion...

Process:

- search recursively starting from root. Traverse down tree to the leaf level
- before progressing to the (next level)/(lower level) in the tree, ensure the node contains >= t keys

3 stages/scenarioes during deletion:

→ Case 1: key k found in node

→ Case 2: key k found in internal node.

→ Case 3: key k suspected to be found in a lower level node.

- Case 1:

conditions: • if the key k is in node x, and x is a leaf, delete k from x.

- Case 2: if the key k is in node x, & x is not a leaf, delete k from x

~~conditions~~:

process: a) → if the child y that preceedes k in node x has atleast t key then find the predecessor k' of k in the sub-tree rooted at y.
Recursively delete k' and replace k with k' in x.

b) "symmetrically for successor node z" — WHAT ??

c) If both y & z have on (t-1) keys, then merge k with the contents of z into y, so that x loses both k and the pointers to z, and y now has (2t-1) keys. Free z and recursively delete k from y.

d) Descending down the tree, if k isn't found in the current node x, the find the sub-tree $c_i[x]$ that has to contain k.

e) If $c_i[x]$ has only (t-1) keys, ensure we descend to a node of size at least (t) keys.

(i) f) def: Distribution: if $c_i[x]$ has only t-1 keys but a sibling has at least t keys, give $c_i[x]$ an extra key by moving a key from x to $c_i[x]$, moving a key from $c_i[x]$'s immediate left & right siblings up into x and moving the appropriate child from the sibling into $c_i[x]$. This process is called distribution.

f) ii) Merging:

If $c_i[x]$ and both of $c_i[x]$'s immediate siblings have $t-1$ keys, merge c_i with one siblings, i.e. move a key from x down into the new merged node to be the new median key for the merged node.

This process is called merging.

Run-time $O(t*h) = O(t * \log_t n)$

July 9 2019

HEAPS

July 26th def: A binary heap data structure A

→ array

→ "nearly a complete binary tree."

- all levels except the lowest level are completely filled

→ The key in root is greater or equal to all its children and the left and right subtrees are again binary heaps.

Binary heaps have 2 attributes: (eg bin. heap data structure A)

→ length[A]

→ heap-size[A]

Max Heap:

def: a heap that has the following property:

$$A[\text{parent}(i)] \geq A[i]$$

where (A) is bin heap data structure, and (i) is the "key" that exists in A and parent (i) is a function as defined below.

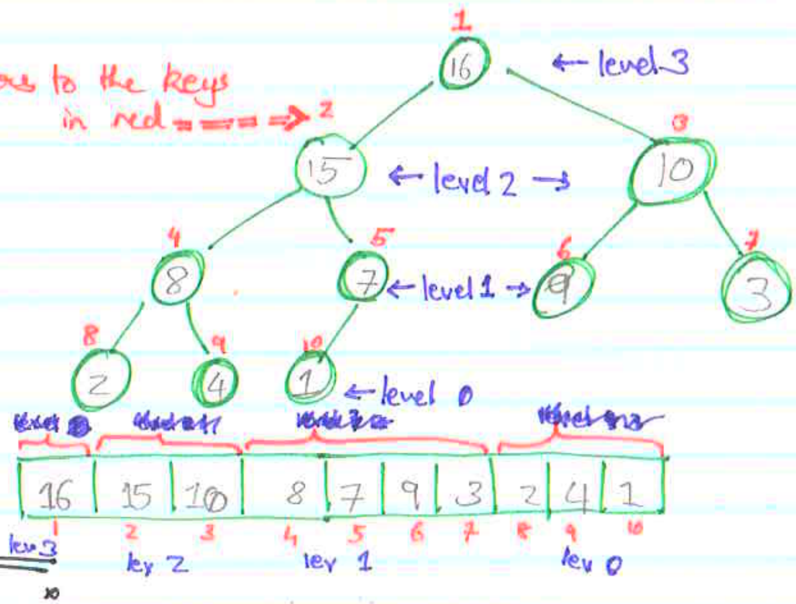
- Parent (i)
  return $\lfloor i/2 \rfloor$

  i → refers to the keys in red

- Left (i)
  return $2i$

- Right (i)
  return $2i + 1$

A →



Heapify

i. i is the index to array A (similar to above)

ii. Binary trees rooted at left(i) and right (i) are heaps

iii. But A[i] might be smaller than its children violating the heap property

Heapify makes array A a heap once more by moving A[i] down the heap until the heap property is satisfied once more.

Pseudocode for heapify:

- n = total number of elements

Heapify (A, i) {

step1 → ensure (??) Left & Right subtrees are heaps
step 2 → make subtree rooted at "i" a heap.

- l = Left(i) // ($l = 2i$
- r = Right(i) $r = 2i + 1$)

- if (l ≤ n) and (A[l] > A[i]) {
  largest = l
  } else {
  largest = i
  }

- if (r ≤ n) and (A[r] > A[largest]) {
  largest = r
  } ~~else {~~

- if (largest != i) {
  swap A[i] and A[largest]
  Heapify [A, largest) //call heapify recursively with largest now.
  }

}

Heapify run time:

→ On a subtree of size n, rooted at node i is:

- determine relationship b/w elements $\Theta(1)$
- + time to run Heapify on a subtree rooted at one of the children of i where $(2n/3)$ is the worst case size of the subtree.

$$T(n) \leq T(2n/3) + \Theta(1) \Rightarrow T(n) = O(\log n)$$

Building a heap:

1) Convert array $A[1..n]$ $(n = \text{length}[A])$ into a heap.

2) Notice that elements in the subarray $A[(\lfloor n/2 \rfloor + 1) \ldots n]$ are already 1-element heaps to begin with. (i.e they are leaves)

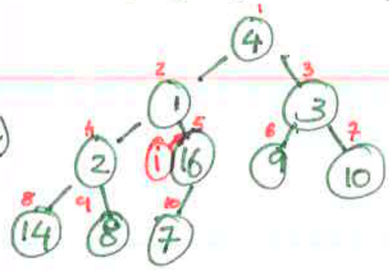
eg A →

4	1	3	2	16	9	10	14	8	7
1	2	3	4	5	6	7	8	9	10

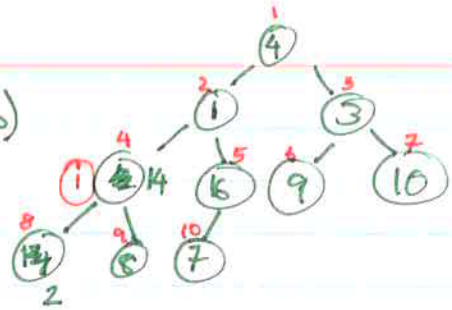
```
Build-heap (A) {

    for (int i = floor(n/2); i >= 1; i--) {
        Heapify (A, i)
    }
}
```

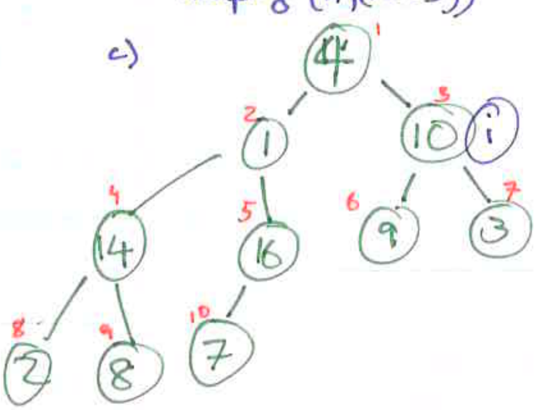
a) Heapify(A, (i==5))



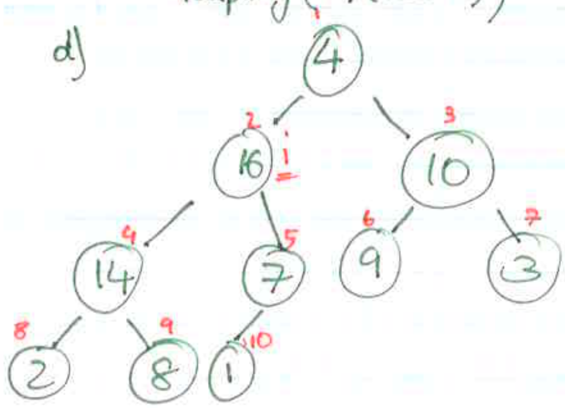
b) Heapify (A (i==4))



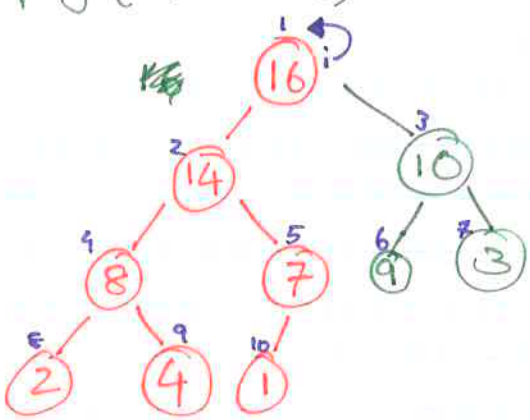
c) Heapify (A, (i==3))



d) Heapify (A, (i==2))



e) Heapify (A, (i==1))



- End of heapify fn
- End of Build-heap(A)

RUNTIME:

n calls on Heapify

$\Rightarrow T(n) = O(n \log n)$

Imp :

1) height of node = longest path from node to leaf

2) height of tree = height of root.

3) time to heapify = O(height (k) of subtree rooted at i)

↳ assume $n = 2^k - 1$

(where $k = \lfloor \log n \rfloor$ for a complete binary tree's height)

$$T(n) = O\left(\frac{n+1}{2} + \frac{n+1}{4}\cdot 2 + \frac{n+1}{8}\cdot 3 + \dots\ 1\cdot k\right)$$

$$= O\left((n+1)\sum_{i=1}^{\lfloor \log n \rfloor} \frac{i}{2^i}\right)$$

$$\left(\text{but} \sum_{i=1}^{\lfloor \log i \rfloor} \frac{i}{2^i} = \frac{1/2}{(1-1/2)^2} = 2\right)$$

$$\Rightarrow T(n) = O(n)$$

(run time to build a heap)