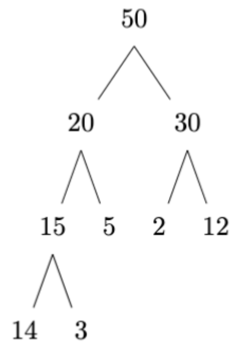
[2/3] Analysis of Algos Notes

Review of Data Structures

- **Dictionaries** (ordered set with the following operations)
  - Insert, Delete, Member, Min, Max, Predecessor, Successor
  - Implemented via (unorderd array, ordered array, linked list, balanced trees)
    - <u>Balanced binary trees</u> support all dictionary operations in $O(\log n)$
    - Arrays and lists support some operations in $O(n)$ and some in $O(1)$ depending on implementation
    - Hash tables can support insert, delete, and member in expected $O(1)$ time
- **Priority Queue**: want to design a data structure
  - Supports the following operations (insert, max, extractmax, increaseKey)
    - IncreaseKey $\rightarrow$ imagine like some ranking system, increasing that identifier
  - How to implement?
    - Balance binary trees can support all of these in $O(\log n)$ time

Heap

- **Heap**
  - <u>Definition</u>: A binary tree that is
    - (1) filled in top-down, left-to-right
    - value of parent $\geq$ value of child
  - Heap Representation
    - Number of nodes from 1
      - $\text{leftchild}(i) = 2i$
      - $\text{rightchild}(i) = 2i + 1$
      - $\text{parent}(i) = \lfloor \frac{i}{2} \rfloor$



$$[50]\ [20]\ [30]\ [15]\ [5]\ [2]\ [12]\ [14]\ [3] \tag{1}$$

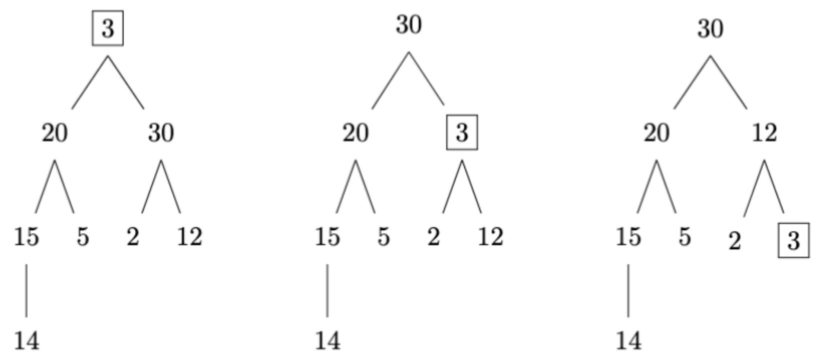
- Extract-Max implementation $\rightarrow$ runtime is $O(1) + O(\text{max-heapify})$

```
HEAP-MAXIMUM(A)
  return A[1]

HEAP-EXTRACT-MAX(A)
  if A.heap-size < 1
    error "heap underflow"              # checking if there's empty heap O(1)

  max = A[1]                            # find largest element O(1)
  A[1] = A[A.heap-size]                 # move last element to the top O(1)
  A.heap-size = A.heap-size - 1         # cut off last element O(1)
  MAX-HEAPIFY(A,1)                      # "reheap"
  return max
```

- Max Heapify: for heapsort, we will need MaxHeapify, which takes 2 heaps rooted at the children of $A[i]$ and makes a heap rooted at $A[i]$

```
MAX-HEAPIFY(A,i)

  l = LEFT(i)
  r = RIGHT(i)

  if l <= A.heapsize and A[l] > A[i]
    largest = l
  else largest = i

  if r <= A.heapsize and A[r] > A[largest]
```

```
    largest = r

  if largest =/= i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```



- (1) Swap the largest element into the "root"
- (2) Investigate the sub-root that you swapped largest with
  - You do not need to investigate the other side, that will still be a valid heap!
- (3) Redo MAX-HEAPIFY on the subroot until you hit the break condition (i.e. a leaf or a valid heap)

Other Heap Operations:

- Heap-Increase-Key: investigate the root, continue to swap up the tree until the parent is bigger than the children!

```
HEAP-INCREASE-KEY(A,i,key):

  if key < A[i]
    error "new key is smaller than current key"              # O(1)

  A[i] = key                                                 # O(1)

  while i > 1 and A[Parent(i)] < A[i]                        # O(log n)
    exchange A[i] with A[PARENT(i)]                            # O(1)
    i = PARENT(i)                                              # O(1)
```

- Max-Heap-Insert: Create a new node that is valid ($-\infty$), then call the Heap-Increase-Key to propogate up the tree!

```
MAX-HEAP-INSERT(A, key)
  A.heap-size = A.heap-size + 1                                  # O(1)
  A[A.heap-size] = -inf                                          # O(1)
  HEAP-INCREASE-KEY(A, A.heap-size, key)                         # O(1)
```

HeapSort

- Input is in B (unsorted array)
- Heap and output in A (sorted)
- Intuitively, we insert all the data into a heap, constantly extract the max and put it into some array $\rightarrow$ this case doesn't use a 3rd array and instead just decreases the heap-size so that the back from i to actual heap length is sorted!
  - Runtime is $O(n)O(\log n) + O(n)\left[2 \cdot O(1) + O(\log n)\right] \implies O(n \log n)$

```
HEAPSORT(A)
  for i = 1 to n                                                 # O(n)
    MAX-HEAP-INSERT(B[i])                                          # O(log n)

  for i = A.length downto 2                                      # O(n)
    exchange A[1] with A[i]                                        # O(1)
    A.heap-size = A.heap-size - 1                                  # O(1)
    MAX.HEAPIFY(A,1)                                               # O(log n)
```

- As we will see, $n \log n$ is actually a lower bound that we can't beat! However, we can improve loop # 1
  - We are doing n heap inserts, but we are doing them without any intervening operations so it might be possible to do the sequence faster

```
HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  for i = A.length downto 2
    exchange A[1] with A[i]
    A.heap-size = A.heap-size - 1
    MAX-HEAPIFY(A, 1)

BUILD-MAX-HEAP(A)
  A.heap-size = A,length
  for i = [A.length/2] down to 1
    MAX-HEAPIFY(A,i)
```

General Rules for Loop Invariant Proofs:

- Vibes are like induction, we use this to prove algorithm correctness
- 3 important steps
  - Initialization: It is true prior to the first ieration of the loop
  - Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration
  - Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct
- When the first two properties hold, the loop invariant is true prior to the iteration of the loop. These are very induction like.
- The third property is perhaps the most important one, since we are using the loop invariant to show correctness.

Heapsort Example

```
Build-Max-Heap(A)

  heap-size[A] = length[A]

  for i = floor(length[A]/2) downto 1
    MAX-HEAPIFY(A,i)
```

- To show why the function works correctly, use loop invariant.

Correctness claim: Build-Max-Heap produces a valid heap.

At the start of each iteration of the for loop of lines $2, 3$, each node $i+1, i+2, \ldots, n$ is the root of a max-heap.

- Initialization: Prior to the first iteraiotn of the loop, $i = \lfloor n/2 \rfloor$. Each node $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$ is a leaf node of a
- Maintenance: Invariant holds for i. To see that each iteration maintains the loop invariarnt, observe that the children of node i are numbered higher than i. By the loop invariant, therefore, they are both roots of max-heaps. This is precisely the condition required for the call $\text{MAX-HEAPIFY}(A, i)$ to make node i a max-heap root. Moreover, the MAX-HEAPIFY call preserves the proprerty that nodes $i+1, i+2, \ldots, n$ are all roots of max-heaps. Decrementing i in the for loop update restablishes the loop invariant for the next iteration.
- Termination: At termination, $i = 0$. By loop invariant, each node $1, 2, \ldots, n$ is the root of a max-heap. In particular, node 1 is.