CSC2710 Analysis of Algorithms — Fall 2022
Unit 8 - Transform and Conquer - Representation Change

Book: §6.1 – 6.4; §31.6

8.1 Representation Change - Heap and Heapsort

- Today we are going to start by looking at a data structure that efficiently implements a *priority queue*.
  - Priority queue: A queue were elements are in the queue in a partially sorted order. In particular all elements with the same priority are grouped together in FIFO ordering.
  - Priority queues support three main operations
    1. Finding the element with highest priority.
    2. Deleting the element with the highest priority.
    3. Adding a new element to the priority queue.
- Priority queues are important ADTs that arise in all sorts of areas.
  - Scheduling processes in an OS
  - Managing phone calls on a cell phone tower.
- One of the most common methods for implementing the priority queue ADT is to use a *heap*.

  Definition 1 (Heap)**.** *A heap is a binary tree with one key assigned to each node subject to the following conditions:*
  - ***Shape Property****: The binary tree is almost complete. In other words, all levels are full except for possibly the last level where there may be missing right-most leaves.*
  - ***Heap Property****: The key in a node is greater than or equal to it's two children.*
    - *A leaf automatically satisfies the heap property by definition.*
- There are many important properties for heaps the following are extremely important.
  1. There exists exactly one almost complete binary tree with n nodes. Its height is equal to $\lfloor \lg n \rfloor$.
  2. The root of the heap always contains the largest key.
  3. A node of a heap together with its descendants form a heap.
     - In other words, a heap is a recursive structure.
  4. A heap can be implement as an array by recording its elements in the top-down, left-to-right fashion. Generally the root is placed in $H[1]$ instead of $H[0]$ for ease of index calculation. In this representation:
     - The parent node keys will be in the first $\left\lfloor \frac{n}{2} \right\rfloor$ positions of the array.
     - The leaf node keys will be in the last $\left\lceil \frac{n}{2} \right\rceil$ positions of the array.
     - The children of a key in parent position i ($1 \leq i \leq \left\lfloor \frac{n}{2} \right\rfloor$) will be in position $2i$ and $2i+1$
     - The parent of a key in position j will be at position $\left\lfloor \frac{j}{2} \right\rfloor$.
- There are two ways to construct a heap given a sequence of keys.
  1. Bottom-up

2. Top-down

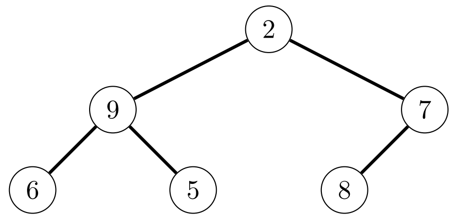
- A bottom up construction works by initializing a complete binary tree with n nodes by placing keys in the order given and then using a special heapify operation.
  - Heapify: starting with the last parent node, check whether the heap property holds for the key in this node.
    - If the heap property does not hold exchange the nodes key k with the larger key of its two children.
    - Then check the nodes parent for the heap property
    - This process is continued until the heap property is satisfied.
- The pseudocode for the bottom up heap construction is as follows:

```
# Input: An array H of integers
# Output: A heap H
def HeapBottomUp( H ):
    for i in range( math.floor(len(H)/2), -1, -1 ):
        Heapify( H, i )
```

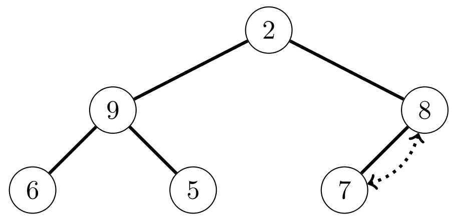
- The Heapify algorithm does most of the heavy lifting and its pseudo-code is:

```
# Input: Array H is an array of orderable objects and i >= 1
# Output: The subtree rooted at H[i] is a heap
def Heapify( H, i ):
    k = i                                                    # Line 1
    v = H[k]      # Keep the original element                  Line 2
    heap = False # Is it currently a heap?                     Line 3
    # While we haven't created a heap yet
    while( heap == False and 2*k < len(H) ):                 # Line 4
        # Jump to the child of the current node
        j = 2 * k                                            # Line 5
        if( j < len(H) ):                                    # Line 6
            # Check the other child
            if( H[j] < H[j+1]):                              # Line 7
                j = j + 1                                    # Line 8
        # Is the tree rooted at v a heap?
        if( v >= H[j] ):                                     # Line 9
            heap = True                                      # Line 10
        # Otherwise, put larger child into parent position
        else:                                                # Line 11
            H[k] = H[j]                                      # Line 12
            k = j                                            # Line 13
    # Place the original key at the correct location
    H[k] = v                                                 # Line 14
```

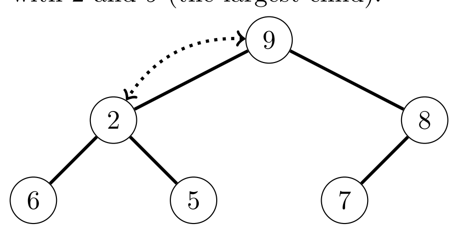
- Let's look at an example of building the heap on the sequence $2, 9, 7, 6, 5, 8$.
    1. We start with the binary tree[1]



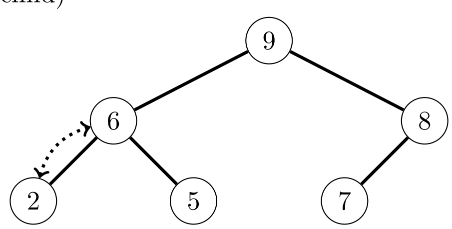
    2. Call HEAPIFY on 7 which results in 7 and 8 swapping places.



    3. We call HEAPIFY on 9 wich results in no change as the parent node, 9 is larger than both children.
    4. We call HEAPIFY on 2.
        (a) This first results in a swap with 2 and 9 (the largest child).



        (b) Swap 2 and 6 (the largest child)



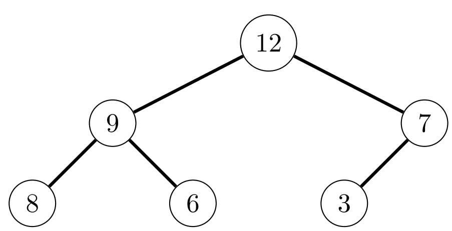
- What is the time efficiency of HEAPBOTTOMUP? One can safely assume that $n = 2^k - 1$ so the heaps tree is full.
    - Let h be the height of tree.
    - Prop. 1 of heaps requires $h = \lfloor \lg n \rfloor$
        * Therefore, $k - 1 = \lfloor \lg (n + 1) \rfloor - 1$
    - Notice that HEAPIFY will only move each key on level i to level h in the worst case.
        * Each move requires two comparisons.
    - This implies on a key on level i requires $2(h - i)$ comparisons in the worst case.

[1] This is a *binary tree* but *not* a BST.

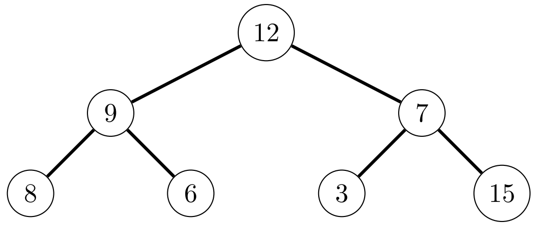
– The total number of key comparisons, and thus the running time of the algorithm is:

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{h-1} \sum_{\text{keys at level } i} 2(h-i) \\
&= \sum_{i=0}^{h-1} \left(2(h-i)\,2^i\right) \\
&= 2\sum_{i=0}^{h-1} \left(2^i (h-i)\right) \\
&= 2\left(\sum_{i=0}^{h-1} 2^i h - \sum_{i=0}^{h-1} i2^i\right) \\
&= 2\left(h\sum_{i=0}^{h-1} 2^i - \sum_{i=0}^{h-1} i2^i\right) \\
&= 2\left(h\left(2^h - 1\right) - \sum_{i=0}^{h-1} i2^i\right) \\
&= h2^{h+1} - h - 2\sum_{i=0}^{h-1} i2^i \\
&= h2^{h+1} - h - 2\left((h-2)\,2^h + 2\right) && \text{By the fact series } \sum_{i=0}^{n} i2^i = (n-1)\,2^{n+1} + 2. \\
&= 2n\lg n - \lg n - 2\left((\lg n - 2)\,n + 2\right) && \text{Substitute the fact that } h \approx \lg n. \\
&= 2n\lg n - \lg n - 2n\lg n + 4n - 4 \\
&= 4n - \lg n - 4 \\
&\in O(n)
\end{aligned}
$$

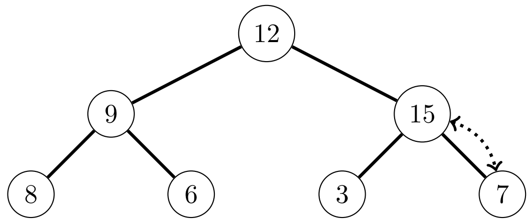
- The top-down construction of a heap is a bit more inefficient then as we are inserting keys into the heap, we don't have all the keys a priori. We build the heap as follows.
  – Insert the element with key k in the left-most empty leaf of the heap.
  – Move the new key up into its correct location. This is done by comparing the key k with its parent's key.
    * If the parents key is greater than or equal to k we have a valid heap.
    * If k is greater than the parent key, swap the keys and and repeat the process using k's new parent.
- Let's look at an example of inserting a new key into a heap. Consider inserting the key 15 in the heap below:



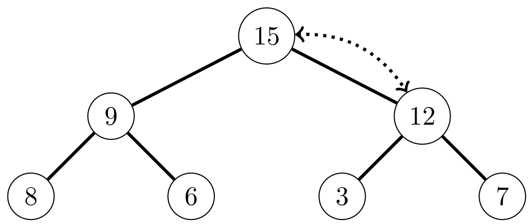
1. Insert 15 into the left-most free leaf.



2. Compare 15 to the parent 7. Since, $15 > 7$ we swap to get:

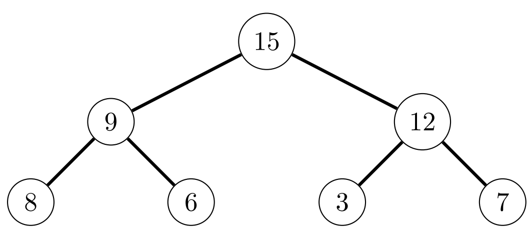


3. Compare 15 to its parent 12. Since, $15 > 12$ we swap to get:

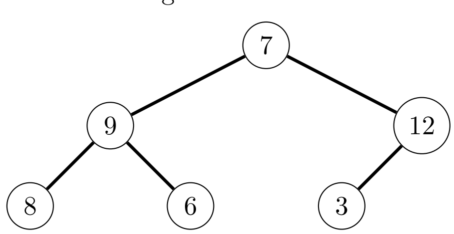


We have reached the root so we are done.

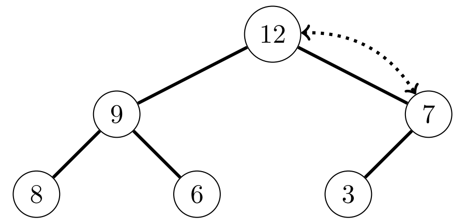
- **Practice**: What is the time complexity of inserting a new key into the heap?
- We also care about removing the root node with key r from the heap.
  1. Exchange r with the right-most leaf's key k. This results in $H[1] \leftarrow k$.
  2. Decrease the heap's size by one.
  3. Call $\textsc{Heapify}(H, 1)$.
- Let's look at an example of removing the root from the following heap:



1. Remove the root and replace it with the right-most leaf and we obtain



2. We call HEAPIFY on the root 7.
   (a) We compare 7 with its largest child 12 and realize we must swap 12 and 7



   (b) Since we swapped the key we must compare 7 with its new largest child which is 3. Since $7 > 3$ we are done.

- **Practice**: What is the time complexity of deleting the root from the heap?

8.1.1 Heapsort

- Using a heap we can get construct an optimal sorting algorithm
  - **Practice**: What is the running time of an optimal sorting algorithm?
- The basic HEAPSORT algorithm works in two phases as follows:
  1. **Heap Construction**: Construct a heap for a given array.
  2. **Delete Maximums**: Delete the maximum node (root) of the heap $n - 1$ times.
- Why do you think this works?
  - Notice that every deletion of a maximum causes the maximum element to be put at $H[s]$ where s is the size of the heap.
  - When maximum is done it decrements s.
  - We end up with the element in location $H[i] \leq H[j]$ where $i > s$ and for all $j > i$.
- To see an example consider succesively removing the root in our delete maximum example.
- If we use an array based heap, what is the running time of HEAPSORT?
  - The *heap construction* phase takes $\Theta(n)$ time.
  - The *Delete maximums* phase takes $\Theta(n \lg n)$ why?
    * delete a node from the heap with requires in the worst case h comparisons.
    * $h = \lfloor \lg n \rfloor$ based on heap property 1.
    * So, we have $T(n) = \sum_{i=1}^{n-1} \lg i$.

- * What time efficiency is $T(n)$?

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n-1} \lg i \\
&\leq \int_1^n \ln x dx && \text{approximate upper bound by a definite integral.} \\
&= \int_1^n u du && \text{Integration by parts where, } u = \ln x, \\
& && du = \frac{dx}{x},\ v = x, \text{ and } dv = dx^2 \\
&= x \ln x \Big|_1^n - \int_{i=1}^n \frac{x}{x} dx \\
&= (n \ln n - 1 \ln 1) - x \Big|_1^n \\
&= n \ln n + n + 1 \\
&= \frac{1}{\lg e} n \lg n + n + 1 \\
&\in O(n \lg n)
\end{aligned}
$$

- We can actually make the bound tight if we both upper bound and lower bound the sum.
  * Hint: For a monotonically increasing function $f(x)$,

$$\int_{\ell-1}^{u} f(x)\, dx \leq \sum_{i=\ell}^{u} f(i) \leq \int_{\ell}^{u+1} f(x)\, dx$$

- It is also safe to memorize or lookup the fact that $\sum_{i=1}^{n} \lg i \approx n \lg n$. You do not have to know integration by parts as Calc II (MTH1218) is not a prereq. It is shown for those of you who have taken calc II and wonder why Computer Scientists might use calc II.

[2] Integration by parts says that $\int u(x)\, v(x)\, dx = u(x)\, v(x) - \int u'(x)\, v(x)\, dx$.

8.2 Representation Change - Horner's Rule and Binary Exponentiation

- An interesting application of representation change is the problem of computing the value of a polynomial $p(x)$ of degree n.
  - A polynomial $p(x)$ of degree n is defined as:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0.$$

- **Practice**: What is the naïve/brute force method for computing the value of a polynomial?
- **Practice**: What is the running time of the naïve method in terms of n?
  - Specifically how many multiplications do you have to perform in terms of n.
- You can reduce the number of multiplications by apply *Horner's Rule*
  - Horner's Rule essentially says you should factor out as many x's in the polynomial as possible
  - Example: $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$ can be re-written as:

$$p(x) = x(x(x(2x-1)+3)+1) - 5.$$

- Observation: Horner's rule looks gross! Thankfully if we want to use it with pen and pencil we don't have to actually do all the factoring.
- We can use a two-row table to compute the value of $p(x)$ using Horner's rule.
  - Row one contains the coefficients in the polynomial list from highest exponent to lowest (i.e., a_n to a_0).
  - Row two contains a_n followed by x's value times the previous entry in the second row plus the coefficient above the current cell in the first row.
  - The last entry in row two is the value of the polynomial at a given value of x.
  - Example: evaluate $p(x) = 2x^4 - x^3 + 3x^2 + x - 5$ at $x = 3$.

2	-1	3	1	-5
2	$3 \cdot 2 + (-1) = 5$	$3 \cdot 5 + 3 = 18$	$3 \cdot 18 + 1 = 55$	$3 \cdot 55 + (-5) = 160$

  Therefore, $p(3) = 160$.

- We are not going to build a table in the computer instead we will use the following simple algorithm where we store the n coefficients of the polynomial in an array.

```
# Input: P is an array storing the coefficients of the polynomial
# Output: The value of the polynomial evaluated at x
def HornerEvaluate( P, x ):
    p = P[ len(P) - 1 ]                          # Line 1
    for i in range( len(P)-1, -1, -1 ):          # Line 2
        p = x * p + P[ i ]                       # line 3
    return p                                     # Line 4
```

- What is the running time, $T(n)$ of the algorithm?
  - Counting the operations:

Line	**Cost**	**Count**
1	c_1	1
2	c_2	$n+1$
3	c_3	$n+1$
4	c_4	1

- Therefore the algorithm has running time $\Theta(n)$. In fact, it only performs $\Theta(n)$ multiplications and additions.

- Horner's rule reaches its worst case for polynomials of the form $p(x) = x^n$.
  - Do you see why?
- We can do better in this case, if we use the binary exponentiation method.
  - In fact, we use a change of representation that expresses the exponent in terms of its binary expansion.
  - **Practice**: Can you think of a decrease-by-a-constant factor and conquer algorithm for computing the binary expansion?
- What happens when we apply Horner's rule to the converted exponent?
  - We get an inclusion-exclusion summation.
  - Example: The binary expansion of 13 is 1101 giving us the polynomial $p(2) = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$.
  - x^n becomes $x^{p(2)} = x^{b_I 2^I + \cdots + b_i 2^i + \cdots + b_0}$ where I is the number of bits in n and b_i is the value of the i-th bit in the binary expansion[3].
- Using Horner's rule straight we would get a snippet of code of the form

```
x**p = x                             # Line 1
for i in range( i-1, -1, -1 ):       # Line 2
    x**p = x**(2 * p + B[i])         # Line 3
```

- It turns out we just have to initialize an accumulator to x and scan the binary expansion of n
  - If the digit is a zero, just square the accumulator.
  - If the digit is a one, square the accumulator and multiply by x.
- Formally the algorithm is:

```
# Input: A number x and the array B of size lg(n)
#   which holds the binary expansion of the exponent n
#   Its highest digit is 1
# Output: The value x^n
def LeftRightBinaryExpansion( B, x ):
    prod = x                                            # Line 1
    # Start at the floor of the lg of n, which is
    #   the same as len(B)-2 (when the highest digit
    #   of B is guaranteed to be a 1)
    for i in range( len(B)-2, -1, -1 ):                 # Line 2
        prod = prod * prod                              # Line 3
        if( B[i] == 1 ):                                # Line 4
            prod = prod * x                             # Line 5
    return prod                                         # Line 6
```

- How many multiplications does this algorithm make?
  - In the worst case every binary digit is a 1, which will force us to have 2 muliplications per digit.
  - We know there are only $\lfloor \lg n \rfloor$ digits.
  - We have that there are $2 \lfloor \lg n \rfloor$ multiplications in the worst case.
  - This implies that our running time is $\Theta(\lg n)$.

[3] The number of bits in a number n is given by $\lfloor \lg n \rfloor + 1$.

Challenge Problem

1. Heap and Heapsort

   - What is the time complexity of inserting a new element into a heap?
   - Devise an algorithm to `search` for an element in a Heap.
     - What is the time complexity of your algorithm?
     - What algorithm class (of the ones covered so far) does it belong to, if any?

2. Representation Change:

   - Design a representation change data structure and algorithm to reduce the running time of a *nearest neighbor query* for a set of 2D points P.