Homework 3 Notes

<u>Building a heap</u>

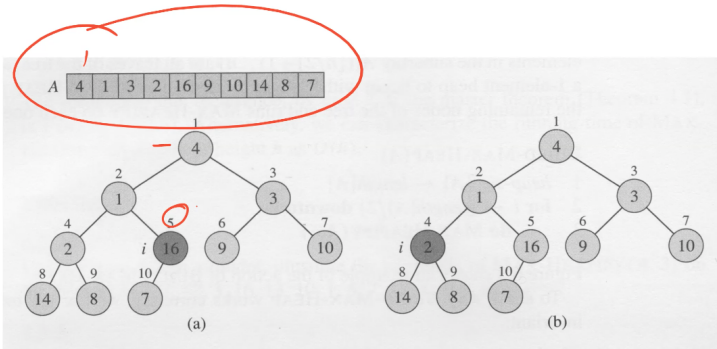
Build-Max-Heap(A)

- heap-size(A) = length(A)
- For i = floor(heap-size(A)/2) downto 1
    - Max-Heapify(A,i)

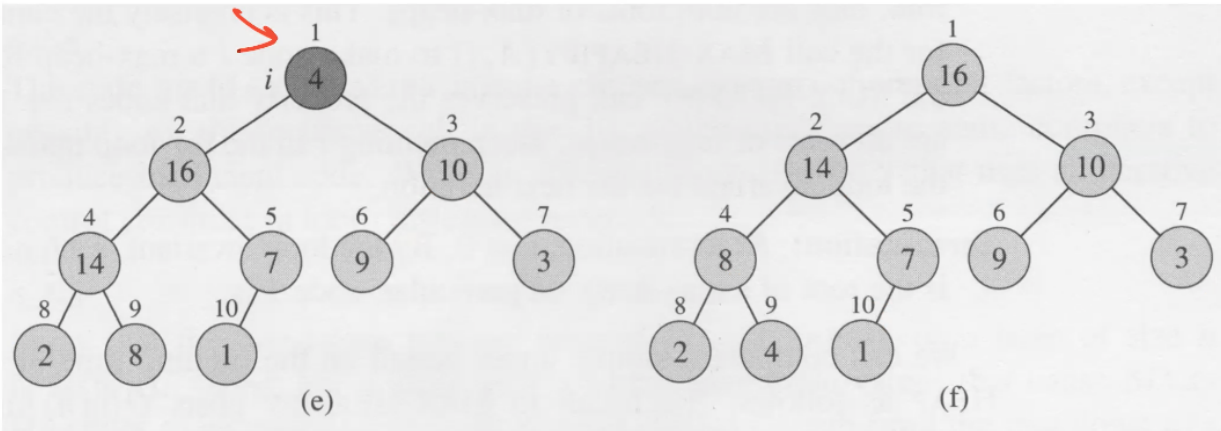
- We want to find the first <u>non-leaf node</u>, found at **<u>floor(heap-size(A)/2)</u>**
    - floor(n/2)
- We want to go bottom up the tree, because the heapify algorithm, the left and right subtree should already be heaps (you can't know that if you start at the top)

$\lfloor \frac{10}{2} \rfloor = 5$

Build-Max-Heap

- 

    - Explanation:
    - First non-leaf node = Floor(10/2) = index 5
    - There is only a left child, a 7 for index 5. Since 16 is larger, the subtree has maintained the heap properties, so nothing to do here
    - Then we decrement index 5 - 1 = 4, now i = 4
    - Looking at index 4 subtree, 14 is the largest, so values 14 and 2 switch positions
    - Decrement i = 3, value 10 switches with value 3
    - Decrement i = 2, look at indexes 2, 4, 5.



- 16 switches spots with 4, keep applying Max Heapify to make sure that 4 goes to the proper location, compare index 2 with 4,5.
- 14 switches spots with 4, 4 switches with 8
- Time complexity is **linear**, the number of items/elements are getting smaller exponentially as we go higher in the tree

The Heapsort Algorithm

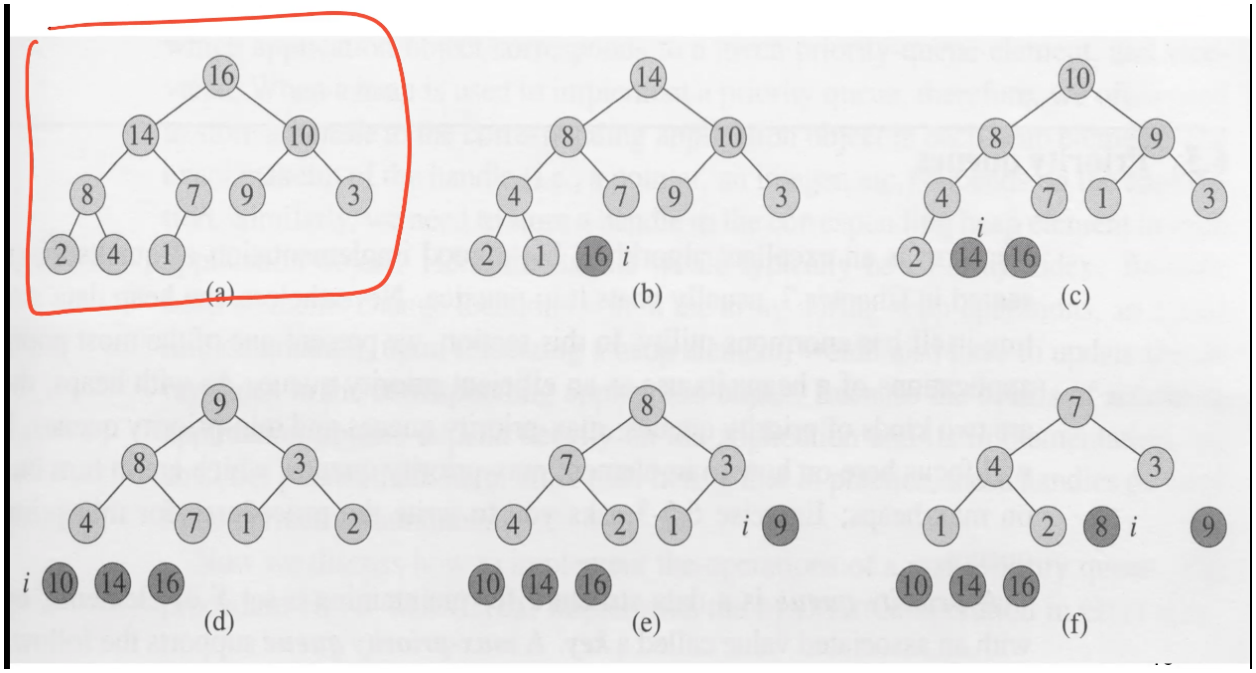
- Move the last element to the front, and remove the last element by (n-1)
- Do this recursively until the elements go in their proper location

Heapsort(A)

- Build-Max-Heap(A)
- For i = length(A) down to 2
    - Swap A[1] and A[i]
        - heap-size[A] = heap-size[A] - 1
        - Max-Heapify(A,1)

- We go down to 2 because a heap with 1 element is already sorted
- A[1] contains largest element
- A[i] or A[n] is the last element
- Now swap the two and decrement the size

- Apply max-heapify at the front, the root node



  - 16 is largest, 1 is the end. Swap. 16 goes to the end. 1 goes to root.
  - 1 is not the largest (in comparison to the subtree), between 14 and 10.
  - So 1 goes down, and 14 goes up. Compare 1 with 8 and 7, 1 and 8 switch
  - Compare 1 with 2 and 4, 4 and 1 switch (graph b)
  - The dark color bubbles are all the elements that are outside the scope of the heap
  - Now, swap 14 and 1 (last element), 1 is back at the root and 14 is out of the heap
  - Compare 1 with 8 and 10, swap 10 with 1
  - Compare 1 with 9 and 3, swap 9 with 1
    - Since 1 is a leaf node, max heapify will terminate
  - (Graph c) Swap 10 with 2 (last element), 10 is outside of heap, decrement size by 1
  - 2 is at root, compare the subtree, swap with 9, swap with 3
  - (Graph d)
- **<u>Time complexity: O(nlogn)</u>**

<u>Priority Queues</u>

- Heap data structure is used in priority queues
- These maintains S elements each with a key
- We can use this queue with a max or min priority queue, we can also:
  - Insert(S,x) O(logn)
  - Max/Min (S) O(1)
  - Extract-Max/Min(S) O(logn)
  - Increase-key (S,x,k) O(logn)

Heap-Extract Max(A)

- If heap-size[A] < 1
    - Then error “heap underflow”
- Max = A[1]
- A[1] = A[heap-size(A)]
- heap-size(A) = heap-size(A) - 1
- Max-Heapify(A,1)
- Return max

- You want to copy the element at the end to the element at the top, because the top element is being extracted
- Overwrite the value at the top
- Since I have extracting one element, I can reduce the size of the heap by 1
- The value at the root might violate the heap property
- We have to apply max-heapify again to make sure the new value at the top goes into the proper location

Heap-Maximum(A)

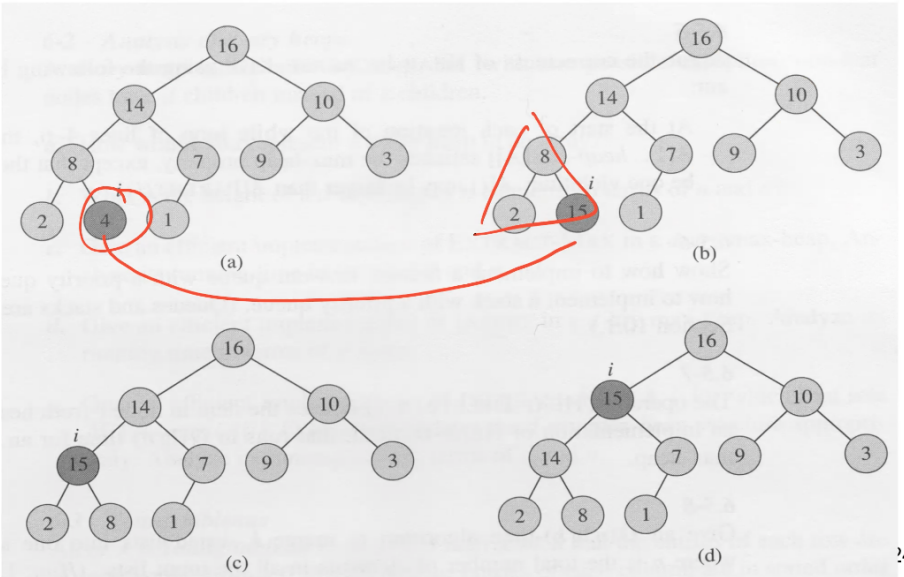
- Return A[1]

- The top is the largest number, just returning it

Max-Heap-Increase-Key(A,i,key)

- If key < A[i]
    - Then error “new key is smaller than current key”
- A[i] = key
- While i > 1 and A[parent(i)] < A[i]
    - Swap A[i] and A[parent(i)]
    - i=parent(i)

- The idea is, to increase the value of node i, to the new value
- Update the value of node i, from its current value to a new value key, the key value being larger than the current, what might happen?
- Might violate the heap property, it might become larger than the parent, we need to bubble up and move the element step-by-step to higher levels until we find the right location (where its less than parent value)
- You want the parent to be larger than the value, then the heap property is maintained
- Termination also happens when you’re all the way up at the root (i=1)
- Swap top with parent



- Explanation:
- We want to increase node i (value 4), increasing it to 15
- Compare 15 with 8 (the parent), 15 swaps with 8
- Compare 15 with 14, 15 swaps with 14
- Compare 15 with 16, i is more than 1, but parent is larger than the value at node i
    - Therefore, the while loop terminates
- **<u>Time complexity = O(logn)</u>**
- Best case, start from leaf node, worst case, go all the way to the root

<u>Max-Heap-Insert(A,key)</u>

- heap-size(A) = heap-size(A) + 1
- A[heap-size(A) = -infinity
    - Max-Heap-Increase-Key (A,Heap-size(A), key)

- We put the item at the end of the array (-infinity)
- We want to make sure it's less than the parent (invariant - should hold at every step)
- Allocate additional space for the new element at the very end
- Increase the key, so that it goes to -infinity with the key
- Maintain the heep property by assigning -infinity to that value and increase this value to key, let the previous function take over and do its work and proper location

<u>QuickSort (A,p,r)</u>

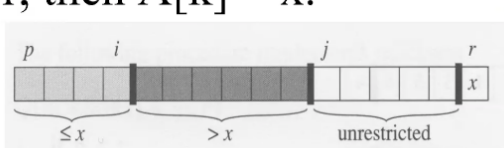
- If p < r
- Then q = PARTITION(A,p,r)
    - QUICKSORT(A,p,q-1)
    - QUICKSORT(A,q+1,r)

- If r is less than p, then the array is empty
- Inductive process if p is less than r, that means there are at least 2 elements in the array
    - Then we want to partition the array into two components around the pivot q
    - All elements from p to q-1 will contain elements smaller than the Q
    - All elements from q+1 to r will contain elements larger than the Q
- Partitions 2 segments
- Then apply quicksort recursively to each segment
- Choose the last element Ar as the pivot, so you can make one single pass on all elements except the last, then compare those elements to the pivot, if they are less than or equal to the pivot, then put into auxiliary segment.
- If larger, put it into the second segment,
- Then copy them back to the array
- What is wrong with this approach?
    - If n is large, you don't want the millions of additional data storage
    - You want to operate on the actual structure without creating a auxiliary data structure that is proportional to the data size
- We rely on loop invariants by describing what we want, we use this as a basis to design our iterative for-loop, to maintain the property
- We are looking for a partition, as we operate our input, each step in the algorithm we want to make sure that we are maintaining two segments
- First segment start from p all the way to i
    - We declare that all elements from p to i will be less than or equal to the pivot
    - All elements from i+1 to j-1 will be larger than the pivot
    - Before the jth iteration, we want to maintain these

Loop Invariant

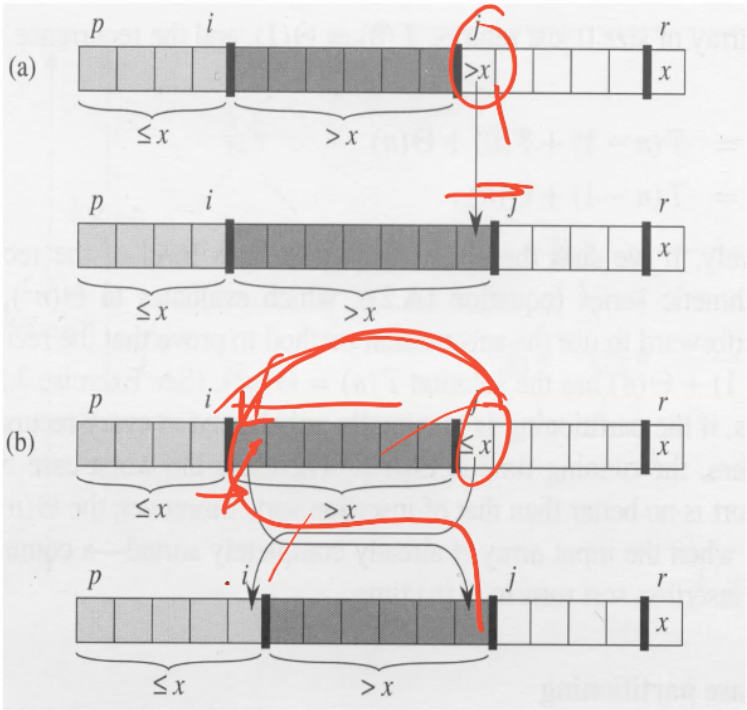
At the beginning of any iteration of the loop of lines 3-6 with a j value between p and r-1, for any array index k,

1. if $p \leq k \leq i$, then $A[k] \leq x$.
2. if $i + 1 \leq k \leq j - 1$, then $A[k] > x$.
3. if $k = r$, then $A[k] = x$.



- Any location between p and i, contains a value that is less than the pivot (x), by default we use last element
- All elements from i+1 to j-1 will contain values less than the pivot (x)
- If k is equal to r, then it is the pivot

- We need to find out that when I insert j, where would I insert it, the first or second segment, how to allocate space?



<u>Why the Loop Invariant is Maintained</u>

- Only two cases possible for what happens to A[j] in any one iteration of procedure Partition
    - Jth item (in second segment) is larger than pivot (x), then there is nothing to do, we move on
    - Jth item is less than or equal to pivot, in this case the number should go to the first segment, but there is no space
        - Doesn't say anything about the order of the elements (they are not supposed to be sorted)
        - We can increment i by 1 to make space and copy jth element
        - We overwrite the element, we can swap the jth item with the A[i+1] item

<u>Partition (A,p,r)</u>

- X = A[r] Identification of the pivot
- I = p-1 Before the first iteration, we know the first/second segments are empty
    - Highlights that the first segment is empty, invariance maintained
- For j = p to r - 1

- If A[j] <= x — Compare jth item to pivot
    - Then i = i + 1 — Allocate space
    - Swap A[i] and A[j] — Insert jth item in that space, all elements from p to i are <= to pivot
- Swap A[i+1] and A[r] — copy the pivot to i+1 location
  Guarantees that all elements from p to i are <= x
- Return i + 1 — Return index of the pivot
-

How efficient is quicksort?

- Its recursive algorithm, we need to identify the recurrences
- Depends on how you partition the array, if you do it in such a way that the input is a sorted array. Means the first segment contains remaining n-1 elements and second segment is empty
- The best case happens when you partition the array from the mid point, the one half of the array is in one segment and the second half has the rest
- How can we formulate our recurrence relations?

<u>Quicksort Recurrences</u>

- T(n) = T(size of left partition) + T(size of right) + O(Theta n)
- We know the partition is linear, single pass through the array
- T(1) = Theta(1), if p < r then go ahead, this is constant

<u>Worst Case</u>

- T(n) = T(0) + T(n-1) + Theta(n) — One of the segments is empty
    - How to solve? — Backwards subsitution
    - T(n-1) + Theta(n) — Expand right hand side
    - T(n-1) + cn
    - T(n) = cn + T(n-1)
    - T(n-1) = T(n-2) + c(n-1)
    - T(n-2) = T(n-3) + c(n-2) — Keep expanding right hand side until base case
    - T(1) — Constant
- Basically Theta(1) + cn + c(n-1) + c(n-2) + … + c
    - Basically a constant time, summation of the numbers from 1 to n
    - Factor c out
- c(n+(n-1)+...+1) — Famous arithmetic series
- = Theta(n^2)
- In the worst case, the quicksort algorithm is a polynomial **<u>Theta(n^2)</u>**

<u>Best Case</u>

- T(n) = 2T(n/2) + Theta(n) — Can we use master theorem?

- A = 2, b = 2, f(n) = Theta(n) Compare n^(log(2)(2)) = f(n)
- They are identical, which case? Case 2
- Time complexity = Theta(n^(log(b)(a)) logn)
- = Theta(nlogn)

Average Case

- Good and bad splits balance out, one part will contain a fraction of the overall
- As long as you have some fraction of the input in one size and the remaining fraction, you will end up with **nlogn** algorithm
- As long as you achieve the fraction of the partition, how can we design the algorithm to increase the likelihood that our partitions are fractional
    - Median-of-Three Pivot Picking
    - Random Sampling
- Those have a better balance of a pivot

Median-of-Three Pivot Picking(A,p,r)

- First = A[p]
- M = floor((p+r)/2)
- Middle = A[m]
- Last = A[r]
- Median-of-Three = median(first,middle,last)
- If Median-of-Three != Last then
    - If Median-of-Three = first then index = p else index = m
    - Swap A[r] and A[index]
- Return Partition(A,p,r)

- Look at the first, middle, and the last items. Find out which one is the medium
- Last item contains the medium
- Modify the algorithm to partition this procedure

Random Sampling

- Choose random number from p to r, use it as the index of the pivot
- Swap A[r] with A[i], where A[i] is the random number I just sampled from the range

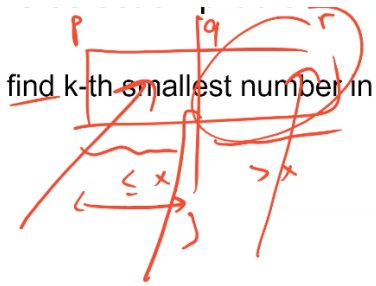
Randomized-Partition (A,p,r)

- I = Random(p,r) Random num between p and r
- Swap A[r] and A[i] Random num is now at the end
- Return Partition(A,p,r)

Homework notes

- Partition the array around the pivot

```
Quickselect(A, p, r, k)
{p & r – starting and ending indexes of array A; to find k-th smallest number in
      non-empty array A; 1≤k≤(r-p+1)}
if p=r then return A[p]
else
      q=Partition(A,p,r)
      pivotDistance=q-p+1
      if k=pivotDistance then
         return A[q]
      else if k<pivotDistance then
         return Quickselect(A,p,q—1,k)
```



1:08:44

- Partition array, find pivot
- Identify pivot distance
- If k = pivot distance, return element
- If k < pivot distance, recursively call the function on the first segment containing elements from p to q-1, returning kth element
- If k > pivot distance, looking from q+1 to r (Second segment) trying to find k-pivotDistance
- Show that this is a linear time algorithm
- Recognize that in the best case, if you are dividing the array in half, how many recursive calls do you have in your algorithm? What is A is our master theorem?
    - Invoke recursive algorithm only on one of these segments
    - Once you identify lower, ignore upper or vice versa
- T(n) = T(n/2) + Theta(n)
    - If you solve the recurrence relation, you will find its linear, even if you have a fraction, its going to be ⅔ of n, its going to be linear

<u>Counting Sort</u>

- Assume that each of the n input elements is an integer in the range 0 to k for some integer k
- Question, how many elements do we need to skip until we get to the right position in the array?
- If I know how many numbers are less than or equal to the value, we can insert that number into the proper location
- We have to count how many elements are before it

<u>COUNT_SORT(A,B,k)</u>

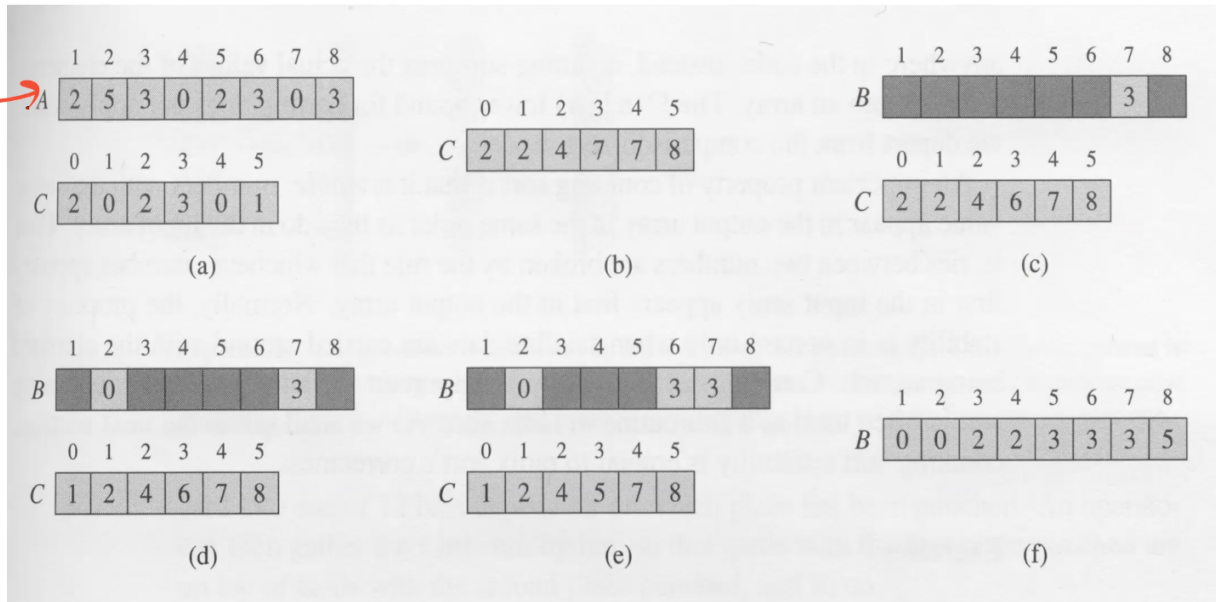
- For i = 0 to k Initializing
    - C[i] = 0 Initializing
- For j = 1 to length[A] Going through array one by one, count num of occurrences
    - C[A[j]] = C[A[j]] + 1
    - C[i] now contains the number of elements equal to i
- For i = 1 to k c[i-1] num elements less than or equal to i
    - C[i] = c[i] + c[i-1] How many elements less than or equal to
    - C[i] now contains the number of elements less than or equal to i
- For j = length[A] downto 1
    - B[c[A[j]]] = A[j]
    - c[A[j]] = c[A[j]]-1

- Explanation:
- C[i] is the count of the number of occurrences of value i in the array, initially zero
- If we know how many zeros we have in the array, we can easily say, the number of elements that are equal or less than to zero
- How many elements are less than or equal to one? The num of elements = 1 plus the num of elements = 0
- We start from 1 because the num less than or equal to zero is zero
- Need to go from the end of array A (n to 1) to the beginning
    - We look at each num in array A
    - Then find out how many elements we need to insert/skip for the proper location in the B array (sorted array)
    - Decrement count by 1 because we already copied one of these elements
    - We do “n to 1” because we switch the order if we do “1 to n”
    - Last occurrence of 3 appears first in array b, the second occurrence of 3 goes in second

8.2 Counting sort

- Stable sorting algorithm

The operation of Counting-sort on an input array A[1..8]



- Explanation:
- (Image A) There are 2 0's, 2 2's and so forth
- (B) Determines how many elements are less than or equal to 0… and so forth
    - Num of elements equal to 1 = 0; Num of elements less than 1 = 2; 0 + 2 = 4
    - Num of elements equal to 2 = 2, Num of elements less than 2 = 2; 2+2 = 4
- © The sum of the above numbers gives you the index of the sorted array, as you can see 3 = 7, then decrement the count of c(3) = 7-1 = 6 and insert the value 3 into the index
- We then go through Array A backwards, so looking at value 3, then value 0, then value 3

<u>Counting Sort Time Complexity</u>

- There are 4 loops starting from 0-k so Theta(k)
- Two loops where we go through a single array Theta(k)
- Two loops going through Theta(n)
- So together, Theta(n+k)
- As long as k is small, this is a linear time complexity

Notes:

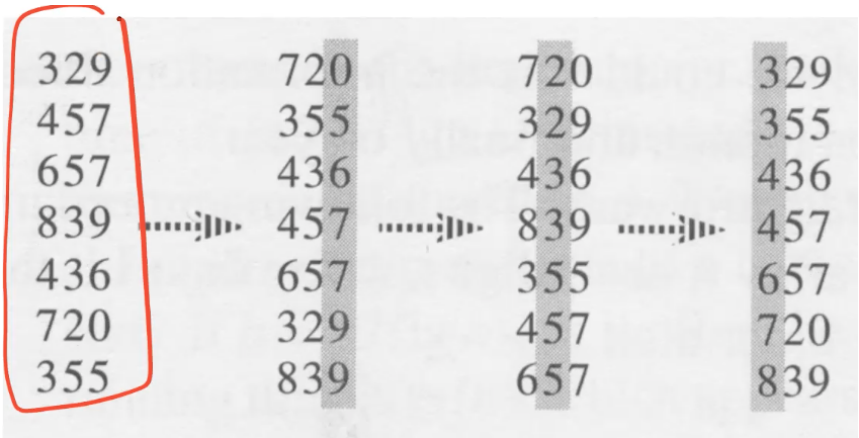
- This algorithm is sorting elements in ascending order
- What would we need to change to do it in descending order?
    - How many elements are greater than or equal?
    - You can update 6 and 7
    - I from k - 1
- What if the input range is instead, a to b?
    - For i from A to B

- If i = A, then
- Map from A to B in a count array
    - Change line 4 because the elements less than A is not 0

Radix Sort (A,d)

- For i = 1 to d
    - Do use a stable sort to sort array A on digit i

- Can represent a big number, can be used with big numbers, can sort a large number of elements in a broad range of the sort in linear time in comparison to n^2 or nlogn



- 3 digits, each digit has a value from 0-9, we have a range of values.
- The idea is to sort these numbers column from column, starting from the right (least significant) then to the left
- We are going through the array and making multiple passes, at each pass we are creating a quasi sorted temporary outcome
- When we sort the numbers based on the last digit, we sort based on the column we are looking at
- Do not undo the previous sorting
- Unless we have a stable sorting algorithm, in the last pass we have undo the previous sorting that we did, don't undo the previous digit sorting.

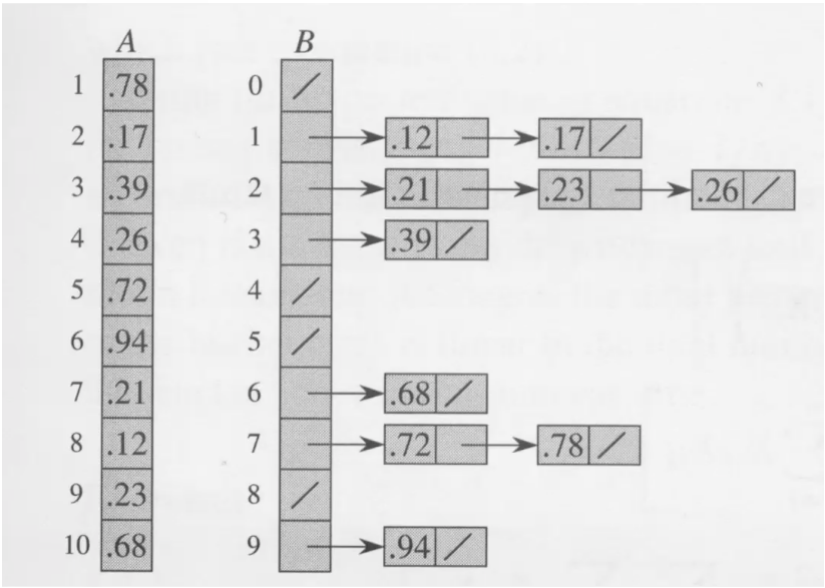
Radix Sort Time Complexity

- Assuming we are using count sort Theta(n+k)
- Theta(d(n+k)), d and k are constants
- D can be large and cover a large range of numbers

- What if we want to sort those numbers in descending order?
    - How to modify the transformation of the problem?
    - What if we negate all the numbers and sort them in ascending order, or descending?
    - Changing the representation of the input

BUCKET_SORT(A)

- N = length[A]
- For i = 1 to n Go through each element
    - Insert A[i] into list B(Floor((nA[i]))) Insert ith ele. Into bucket floor(n*A[i])
- For i = 0 to n-1
    - Sort list B[i[with insertion sort
- Concatenate B[0], B[1], … B[n-1] in order



- We have input array A, of n numbers
- We insert them into N buckets
    - We do this by slicing the range from 0 to 1, into equally large segments/fragments
    - Each segment is a bucket
    - If the number is from (0,1/n) -> bucket 0
    - (1/n, 2/n) -> bucket 1
- The range of numbers in each bucket, how do you associate the bucket ID (0 to n-1) with the range of values on the left, is there a relationship?
    - Multiply the numbers in the range by n, the range floor(0 -> 1) = 0
    - You will get the bucket item
- Look at each item, and which bucket ID it should go into
- N is the input array length
- Once you insert elements in the bucket, we know that bucket 2 will be larger than bucket 1
    - No guarantee that bucket elements are sorted
    - We need to sort each bucket with insertion sort
- If we use insertion sort, O(ni^2) time complexity
    - This is an issue, what if all elements go into one single bucket? Ni = n, and n^2 will defeat the purpose of this algorithm
- If uniformly distributed, expected number of items in each bucket = 2-1/n
- That's why the sort algorithm is a constant time, insertion sort is constant time

- What happens when the input is coming from the range [o,p]
    - How do we slice and fragment that range into n equally sized buckets?

[0,P]
[0,P/n]
[P/n, 2P/n] How to transform this?
… [n-1/n * P, n/n * P] -> n-1

- We divided range into n buckets
- How to transform a value in a particular range into its bucket item
- With what number would you multiply and divide these lower and upper bounds so that it will produce a number, the floor of which would be the index.
    - Multiple A[i] * n / p, this case it will be a number between 1 and 2, the floor of that number will be 1, which is the bucket ID
    - If you modify the requirement to deal with numbers from 0 to P, you just need to change line 3 to B(floor(n/p* A[i]))

- What if numbers are coming from the range [i,j]?
    - One approach: map the range from i to j -> to 0 to 1, then use the algorithm as is (problem transformation), apply a transformation function that will help us obtain the values from 0 to 1
    - Produces 0 if input is i, produces 1 if input is j
    - A plausible algebraic function that will produce 0 if i and 1 if j…
        - A[i] - i/ j-i

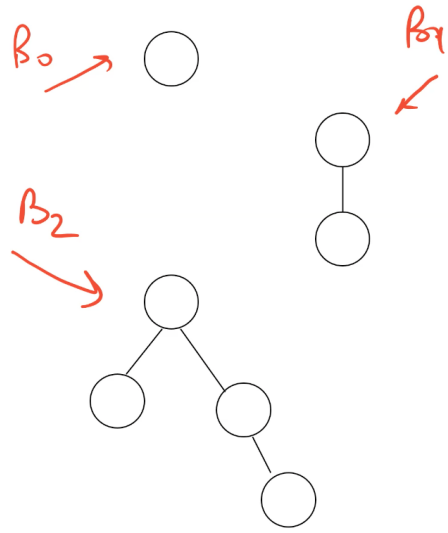
<u>Binomial Heaps AKA Binomial Queue(BQ)</u>

- Problems
    - Supports efficient insert and Extract-Min
    - But merging two heaps (with total n nodes) will require O(n) time
    - Not good enough for many applications
    - We now look at a kind of Mergeable heap which supports O(logn) merges

- Consists of a forest of Binomial Trees (BiT)
    - Each BiT is a heap

BiT

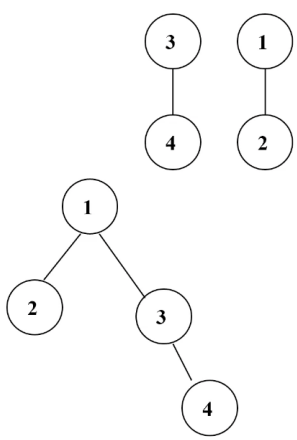
- B_k : Bit of height k, k=0,1,2,...
- B_0 : 1-node tree
- B_1 : 2-node tree
- B_2 : 4-node tree
- B_i : 2^i-node tree

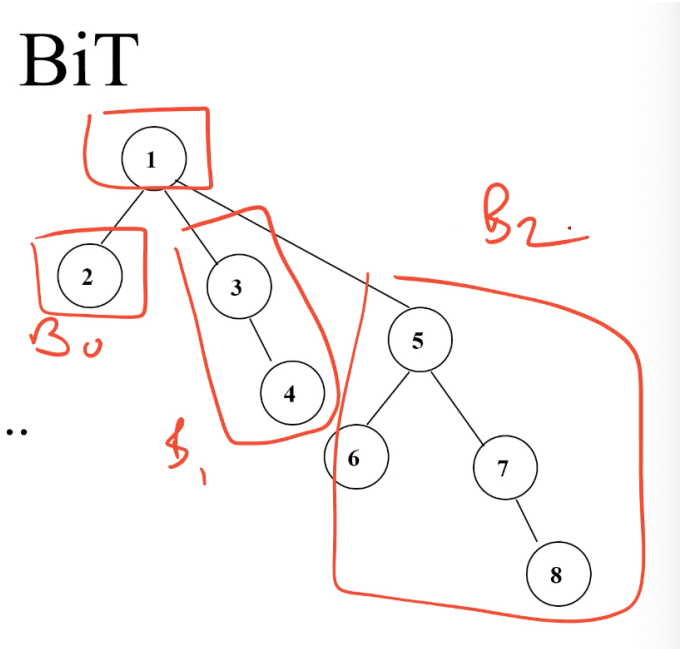


- If you have a binomial of height i, the number of nodes is 2^i node trees
- Our elements are the binomial trees, we want to focus on the binary trees instead of the individual elements
- How do we construct these increasingly larger sized binary trees?

<u>BiT</u>

- You construct a B(k) tree by attaching a B(k-1) tree to the root of another B(k-1) tree, making sure that the heap property is preserved
- Do you have a max heap or min heap?
- Let's say we want to connect B(1) with B(1), we want to connect 3 to the root of the binary tree with the smaller value (1), to obtain B(2)
- Two B(1)'s connect to obtain B(2) node tree
- If you construct these BT by following this rule, you end up with this structure





- A B(k) tree therefore consists of a root with k child subtrees: B(0), B(1) .. B(k-1) trees
- Three child subtrees, b0, b1, and b2, each one is a binomial tree on their own
- This structure, the height is 3, at 1 (depth is 0), 2 and 3 and 5 (depth 1),
- The number of nodes at depth d is represented by binomial coefficient (k choose d)

$$\binom{k}{d} = \frac{k!}{d!(k-d)!}$$

Where k is depth and d is the number of nodes

<u>Binomial Queue</u>

- A forest - a collection of heap-ordered trees
- Each tree is a Binomial (not binary) tree
- At most one Binomial Tree (BiT) of any height in a BQ
    - There should be either 0 or 1 B(1), 0 or 1 B(2) …
    - Binomial Queue of height k Is either present or absent
    - If present only one instance of that, cannot occur more than once
- BQ is a collection of trees and satisfies
    - Heap property
    - Structural property
        - How we connect b(k-1) with b(k-1) to get b(k)
- A priority queue of any size n can be uniquely represented by a BQ of size n
- Interpret binary digits as a presence or absence of binary height k

- We can represent 6 elements by a binary queue that contains no B(0)'s

How many binary trees do we need?

- To see how many and which BiTs are in the BQ, look at the binary representation of n
- How many digits are in the binary rep? Tells us how large it will be, how many binary digits do we need to represent n
  - We can prove this by induction that the number of digits that we need to represent n is floor(logn)+1
  - Ex: 7 = 111
  - log(7) = 2+1 = 3, we need 3 digits to represent 7

<u>BQ Operations</u>

- Merge is the fundamental operation
- Merge can be done in O(logn) worst case time
- Other operations in terms of Merge
  - How do we merge 2 binomial queues, one with 6 elements and the other with 7
  - Ex: 110 and 111
  - Add = 1101 = 13
  - 1101 = (1) 1 B(3) -> (1) 1 B(2) -> (0) 0 B(1) -> (1) 1 B(0)

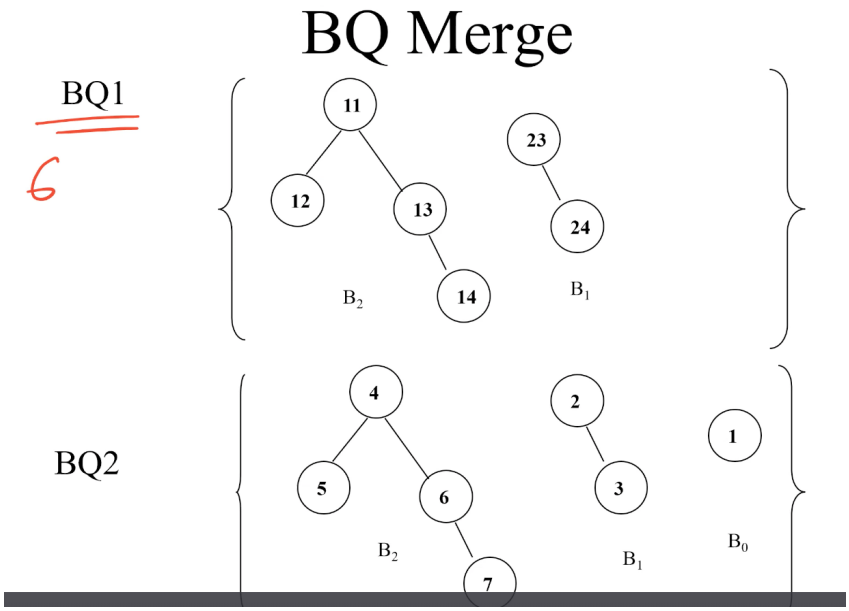
<u>BQ Merge</u>

- Adding two binary numbers as the merge operation, root of one is connected to the root of the other, and since we have logn digits, that means logn * constant time operation = logn
  - Overall time complexity = **<u>logn</u>**

- Suppose you want to merge two BQs:
  - BQ1 of size 6 = 110 $\rightarrow \{B_2, B_1\}$
  - BQ2 of size 7 = 111 $\rightarrow \{B_2, B_1, B_0\}$
  - The merged BQ will be of size 13 = 1101 $\rightarrow \{B_3, B_2, B_0\}$

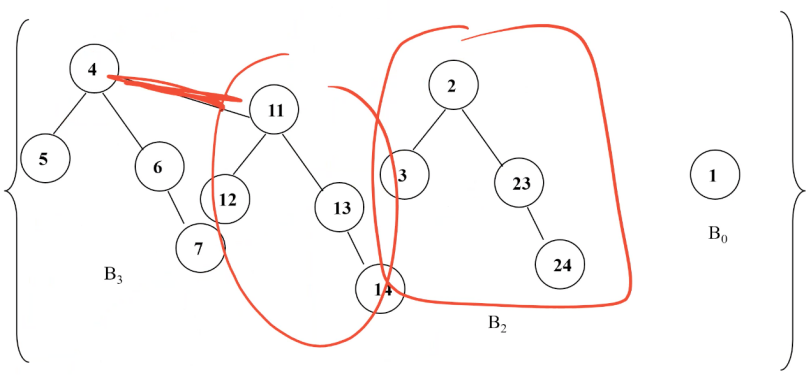
- The process is similar to binary addition with carry:

```
   1     1     1     0
   1     1     1     1
---------------------------
1  1     0     1
```



Explanation:

- BQ1 has 6 elements, BQ2 has 7
- For B(1), which one do we connect? Theres Min Heap, then we would need to connect 23 to the root of 2 which we can then connect to either B2 trees



- How to insert a new element of a BQ of size n elements?

<u>Insert(x,BQ)</u>

- Merge (BQ containing a B(0) tree, BQ)
- Also O(logn) worst case

How about extracting? You have to restructure the data so it will be binomial

- Suppose we have a min heap, the root element will contain the smallest number in each binomial tree, so we just have to scan through roots of BT (logn binomial trees to check)
- We look at the roots and choose smallest value, that we will return
- We are removing/extracting it

<u>Key Idea:</u>

{B(3), B(2), B(1), B(0)}

- Once you scan through and find the item with the smallest value
- Critical properties of BT - it has k child subtrees, each is a BT, if we remove the root, we end up with the BT from B(0) - B(k-1) where K=3 if we found one from B(3)
- We can consider the remaining Binomial Queues from the original Queue and merge them
- The remaining elements will constitute a binomial queue, just merge it from the original excluding the B(3) the node you extracted

<u>Extract-Min(BQ) - O(logn) worst case</u>

- Scan roots of all BiTs in BQ to find the minimum root
- Delete the root and eventually return the data - O(1)
- BQ1 = BQ containing all the child subtrees of the deleted root
- BQ2 = BQ containing all the other BiTs from BQ
- Merge BQ1 and BQ2 - O(logn) worst case
    - Turns out that extract-max operation is logn, efficient for all operations

<u>Data Structure for Disjoint Sets</u>

- Its implementation
- Its operation

<u>Relations</u>

- R: a relation defined on a set S of elements
- aRb is true (where a,b c- S) -> a is related to b by R
- A subset of the cartesian product AxB
- Properties:
    - R is reflexive if aRa is true for all a in S (so a=a)
    - R is symmetric if (aRb is true ⇔ bRa is true) for all a, b in S (so a=b and b=a)
    - R is transitive if (aRb is true and bRc is true -> aRc is true) for all a,b,c in S (so a=b, b=c, then a=c)
    - R is an equivalence relation (ER) if it is reflexive, symmetric, and transitive

a < a

Properties

	reflexive	symmetric	transitive	equivalence
=	Y	Y	Y	Y
<	N	N	Y	N
>=	Y	N	Y	N
married	N	Y	?	N
sibling-of	Y	Y	Y	Y
father-of	N	N	N	N

Equivalence Problem

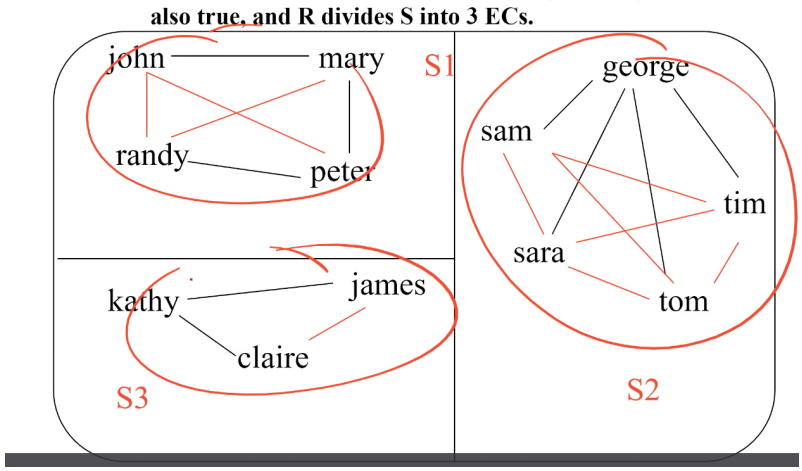
- If R is an ER defined over a set of objects S, for every pair (a,b) where a,b is in S, determine if aRb is true or false
    - Are a and b related? How do we find this out

Equivalence Class (EC)

- The EC of an element a in S is the subset of S containing all elements that are related to a by the ER R.
    - A set of elements that are actually related to a, according to the equivalence relation r
- In other words, if we define an ER R over a set S, R partitions S into a number of disjoint sets that are ECs

Example

Since R is an ER, additional relations (red lines) are also true, and R divides S into 3 ECs.



Explanation:

- Look at the people, assume the sibling relationship, the equivalence relation, will impose a partitioning of set S
- We can do transitive closure and identify these clusters since we know its an ER
  - If John and Mary are siblings, then John and Peter must be siblings
- John = {Mary, Randy, Peter} John's equivalence class, each one a disjoint set

- Now, if we maintain information about all ECs in a database of objects with a relation R defined over these objects, we can answer the question aRb? By checking if a and b belong to the same Ec.
- Are two elements in the same cluster? These clusters might be evolving and changing as we acquire new information
- Are a and b in the same disjoint set?
- Similarly, we can assert aRb in the database by putting a and b in the same EC
  - We want to perform this operation again and as efficient as possible

ECs continued …

- This means that we need to be able to efficiently implement two operations
  - Determine if aRb is true/false - how?
    - True if EC(a) = EC(b); false otherwise
    - "Find(element) returns its EC"
  - Assert aRb in the database - how?
    - If EC(a) !- EC(b) then merge EC(a) and EC(b)
    - "Union(EC1, EC2) merges the two ECs"
    - Parameters are the identifiers of the EC

<u>Implementation</u>

- Data elements in the database are numbered 1...n
- ECs are named 1...n also (dont get confused with element numbering)
- If all elements are independent and disjoint, then they will have their own EC identifiers
- Initially, we assume that no element is related to any other, i.e. each is in an EC by itself
- Two approaches to implementing such a database so that Find and Union operations can be done very efficiently

<u>Implementation Approach 1</u>

- Use an array A in which A[i] contains the "name" of the EC to which element i belongs

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

- Above is array A, lets say we are looking at element 2, the indexes are actually the values of the elements, the contents are the identifiers of the EC
  - If you look at the array, element 2 and 9 are both in the same EC with the ID=2
  - Elements 1 and 10 are in the same EC, labeled 1

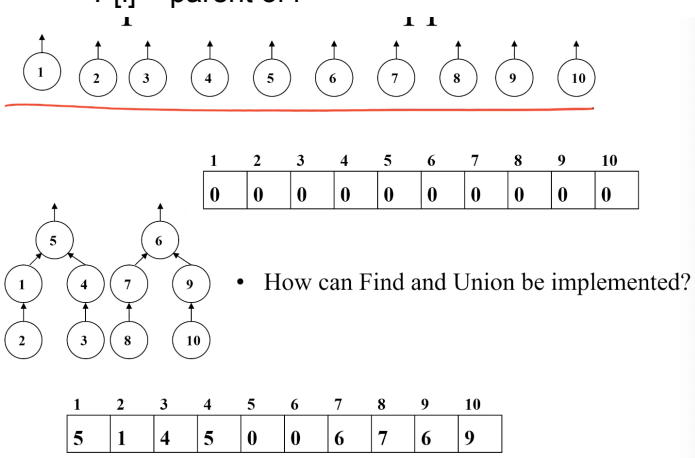
    - A[i] returns the actual ID of the EC that belongs
- How will you implement Find(i) and Union(i,j)? With what complexities?
    - Find will be “return A[i]”
    - Union is more complex, i and j are EC of two different elements
    - EC for 1 is 1, EC for 2 is 2, suppose you want to merge them
        - Value 1 and 2 are in 2 different EC and we want to assert, that they will be in the same EC
        - How to modify the data representation?
            - Go through elements, each time you see a 1, you replace it with 2, or each time you see 2 you replace it with 1. Either or should work
            - When that happens, that means everything is in EC labeled as 2, **<u>Theta(n)</u>** operation
            - We have to make a linear, single pass through the array

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

-

<u>Implementation Approach 2</u>

- Representing EC in terms of a tree, the root provides the label/ID/name of EC
- Representation generates a forest, each tree represents a particular cluster
- P[i] = parent of i



- P[i] is equal to zero initially
- Looking at the two disjoint sets, 5 and 6 are at the root
    - p(5) and p(6) will be zero, parent of 1 is 5, parent of 2 is 1 P(2) = 1, parent of 3 is 4 P(3) = 4 … so forth
    - We use value 0 to indicate it is the root of the particular disjoint set, value 5 is the identifier, parent representation of disjoint set on the right is structured accordingly above
- If this is the better representation, then what will be the time complexity for the Find and Union operations?

- If we have element = 2, how would you find the ideal EC for this element?
    - Get parent P(2) = 1, 1 is not at root, then parent of 1, go all the way up to the root, then return the value of the element (id of the EC)
    - Go all the way up to the root, iteratively, while P(i) != 0 , go up, set i = P[i] and keep going
    - While loop goes from node to root
- What is the worst case that can happen in Find()?
    - Depends on how the tree is structured, it might turn out that you may have a linear chain of these vertices, your tree can be linear of n elements, so you have to navigate from the bottom to the root, it would take at most n
    - Time complexity = O(n)
- Worst case for union?
    - When you merge two elements to be in the same EC, then all disjoint set EC will be in the same, that will be one single merged disjoint set
    - We need to find the EC for each element, we need access to root, then we connect root = 1 to root of another
    - Union operation takes two inputs, the EC id for first and for second (i,j) root values
    - We just need to connect the root of one to the root of the other, connect 5 to 6 or 6 to 5. Change the parent of either or.
    - This will be a constant operation, O(n)

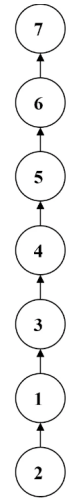
- Which one is better? How to further improve the time complexity?
- Approach 1
    - Find Theta(1)
    - Union Theta(n)
- Approach 2
    - Find O(n)
    - Union Theta(1)
- Which one to choose?
    - We need to understand theta and big O
    - First approach does not have an opportunity to improve
    - Second one is better, provides opportunity for improvement

<u>Disjoint Set Operations</u>

- Union
    - Arbitrary
        - Connecting roots together, update parent values
        - With arbitrary union, n-deep trees may be generated (linear chain of vertices resulting in
        - Suppose we have:

```
Union(Find(1),Find(2));
Union(Find(3), Find(1));
Union(Find(4), Find(3));
Union(Find(5), Find(4));
Union(Find(6), Find(5));
Union(Find(7), Find(6));
Union(Find(8), Find(7))
```

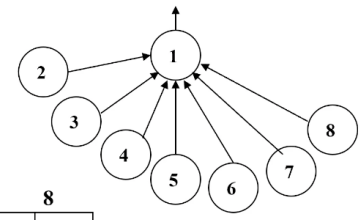
1	2	3	4	5	6	7	8
3	1	4	5	6	7	8	0



-
    - Result is on the right side, the depth is n
    - If we apply Find operating on 2, you have to search and go through the whole tree **O(n)**
    - Can we improve?
- By-Size / “Smart” Union / Union-by-Size
    - Use some smarts in deciding how to merge two trees
    - Number of disjoint sets, if you are looking at how many elements exist in the disjoint set we use -x, we can’t use a positive value because it might be interpreted as the parent node
    - Two ways: Union by size, and union by height
    - Limits the depth/height of the tree to O(logn)
    - Suppose you have two disjoint set trees, when you connect the smaller one to the larger one, the depth will increase by 1, because of the additional link that connects the smaller to the larger
    - By how much does the size of the overall tree increase from the perspective of the smaller tree?
        - Small has n elements, and the larger is larger, each time you increase the depth by 1, you increase the size of the disjoint set by double
        - Each time you at least double the size of the set, you increment the depth of the nodes in the smaller tree by 1
    - Informal Proof:
        - Any tree of depth k must have at least 2^k nodes
        - Since there are only n data items, 2^k <= n
        - So k <= logn
        - I.e. the depth of any node <= logn
        - Therefore the depth of any tree <= logn
- By-Height / Union-by-height
    - Make the shallower tree a subtree of the root of the deeper one
        - Implementation: at array cells corresponding to roots, use the negative of tree-depth instead of 0

        - Increase the height by 1 when unioning two trees of the same height
        - P[r] = -h
        - R is root of disjoint set, and h is height

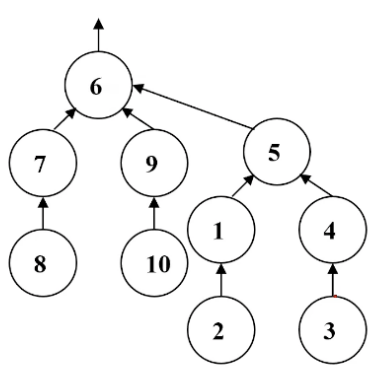
```
Union(Find(1),Find(2));
Union(Find(3), Find(1));
Union(Find(4), Find(3));
Union(Find(5), Find(4));
Union(Find(6), Find(5));
Union(Find(7), Find(6));
Union(Find(8), Find(7))
```



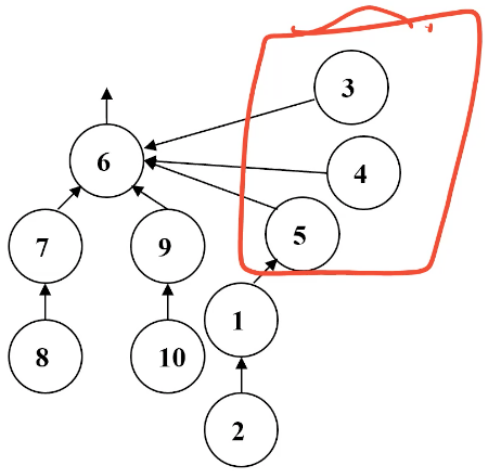
1	2	3	4	5	6	7	8
-2	1	1	1	1	1	1	1

        -
    - If you start with n nodes in the set originally, any tree resulting from Union-by-height will have a depth of at most **logn**
    - When you combine two disjoint sets, it will double in size and depth will increase by 1
    - Can we improve further?
- Find
    - without/with Path Compression
    - Path Compression
        - With smart unions we can bound the depth of any tree to logn
        - But Find operations on leaf nodes will still take longn time
        - We can improve this situation by modifying the Find operation
        - During Find(x), change the parent of every node between x and the root to be the root of the tree
        - An example of self-adjusting data structure
    - Find without Path Compression
        - Find(3)

Find(3)



1	2	3	4	5	6	7	8	9	10
5	1	4	5	6	0	6	7	6	9



- Why not direct 3 to 6, that way next time you don't have to go from 3-4-5-6, why not point 4 and 5 directly to root 6, so the next operation will take 1 step instead of a sequence of steps
- Set the parent of x to to the result
    - P[x] = find(p[x])

- Why Path Compression?
    - Gradually decreases depth of tree
    - As items are once accessed, they are more likely to be accessed again and makes it single-step accessible
    - Path compression works well with Union-by-Size, but you cannot use it as is with Union-by-height, because by height depends on the height of the tree and path compression modifies the depth of the tree, defeats the purpose of improving time complexity
- How do you find the height of a tree?
    - You can calculate it recursively
    - Recursively applying to the child subtrees and recursively return the results
    - Apply a max() operation, on all height of the child subtrees, the largest height + 1

Chapter 22.
Elementary Graph Algorithms

Conventions

- V - set of vertices or nodes
- E - set of edges
- | V| - # of vertices or nodes = n
- |E| - # of edges = m

Representation of Graphs

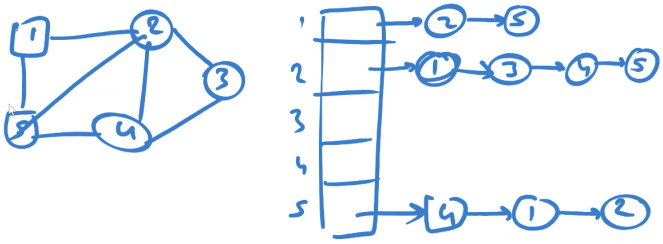
- Types: undirected, directed & weighted graphs
    - What is the max # of edges in a n-node undirected graph?

        - (n-1)n/2
    - Directed graph
        - Has clear explicit direction
    - Weighted graph
        - Assigns a weight to each edge w: E->R
        - Function that maps a set of edges and real numbers, assigning an intensity
        - In real world situations, traffic, cost of pavement (mayor) minimize the cost of payment

- Adjacency matrix representation (for dense graphs)
    - How much space do we need to represent, what is my space requirement for V
    - Theta(V^2) Requires v^2 number of cells, upper/lower bound is the same
    - How much time will it take to determine, whether or not there is an edge between U and V?
        - Theta(1) constant
    - If you have a weighted graph, you can use the actual weight on the matrix
- Adjacency-list representation (for sparse graphs)
    - When you have a small number of connections
    - Number of edges is small
- 1 if there is an edge, 0 if theres not an edge

$$
\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0
\end{array}
$$

- Row 1 has edges to 2 and 5
- If A(ij) = 1, then A(ji) = 1



- We write the neighbors of each number, 1 has neighbors 2 and 5
- Adjacent list, each element in the linked list has an attribute

Degree(u) - Degree of a vertex

- The number of edges on a vertex, the larger the degree, the more focal and central the vertex is
- In-degree + out-degree = degree(u) in and out
- Space requirement for adjacent list in terms of set of vertices and edges
    - Theta(v), at least we need V space because of the block/cell for each vertex
        - Linked-list per vertex
    - **Theta(v+e)** - e is the edge set
    - Worst case, E can be proportional to that block/cell
- How much time will it take to list/enumerate all the vertices that are adjacent to vertex U?
    - The size of the linked list (above image, right side with circles)
    - You have to make a single pass over the linked list
    - What is the length of the linked list? It's the degree
    - **Theta(degree(u))**
- How much time will it take to determine (u,v) c- E if there is an edge between vertex U and V?
    - Could take 1 step if V is at the front, worst case is degree(u), big O is worst case, if you have to look through all elements and V is at the end
    - O(degree(u))
- How to find out the in-degree and out-degree for each vertex in a graph?
- indegree(u)=n-1, incoming edge for each and every vertex
- outdegree(u) = 0
    - Answer is trivial, if theres no constraint on the time complexity
- Is there a universal sink in a graph? In linear time?
    - A vertex indegree(u) = n-1, incoming edge
    - With outdegree(u) = 0

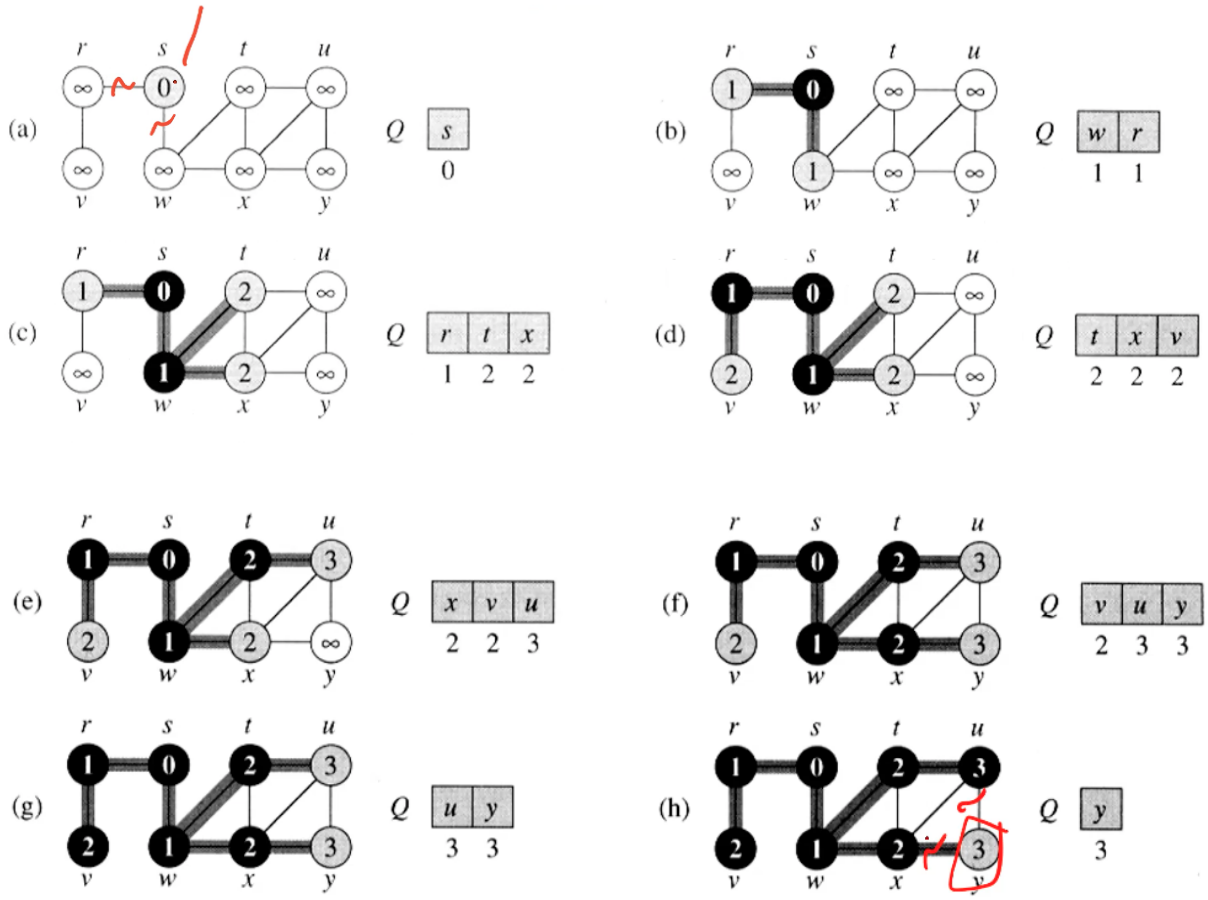
<u>Final Exam Notes:</u>

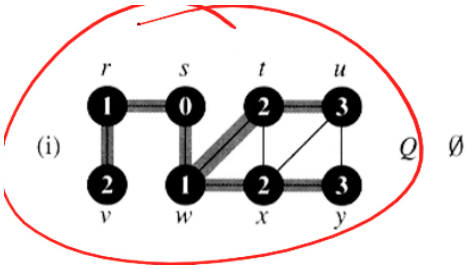
<u>Breadth-first Search</u>

- An algorithm for searching a graph

- Starting from a vertex s, visits every vertex that is reachable from s
- Called breadth-first because it visits all vertices reachable from s in 1 step (by 1 edge) first, then those reachable in 2 steps, and so on
- Produces a breadth-first tree
- Using a queue representation to mitigate the graph
- Rational behind each step Initialization:
    - BFS(G,S) where G = (V, E), for each vertex u in V[G]-{s}
    - WE are using the attributes to signal properties, tells us whether or not to visit or ignore the vertexes, we dont want to visit twice, how far is it from source?
    - First we use the color attribute color [u] = white, that vertex has not been discovered since we made it white
    - D[u] = infinity, distance of vertex is infinity initially, meaning it hasnt been discovered
    - We need to identify the shortest path from the source to the vertex, we need to modify that restriction/constraint that there are mitigations associated with edges, we assume each edge has cost of 1
    - Pie[u] = NIL; pie-> predecessor for vertex u is nil, not yet discovered a path to that vertex
- When vertexes has been discovered:
    - Color[s] = gray, every discovered is gray now, where s is source
    - D[s] = 0, source distance is 0, we are at source so its 0
    - Pie[s] = NIL, source has no predecessor
    - Q = {s} , initialize Q with source vertex, we insert new vertexes into Q
    - While q != {} While q is not empty
        - U = head[q] which vertices are reachable from this starting vertex?
        - For each vertex (v) in Adjacent[u] Looking at each vertex v adjacent to u (immediate neighbors of vertex u) If v has not yet been discovered then insert Q for further processing
        - If color[v] = white This is the first time we discovered
            - Color[v] = gray Mark that it has been discovered
            - D[v] = d[u] + 1 1 more distance away
            - Pie[v] = u Predecessor of v is now u
            - Enqueue(Q,v) Insert vertex into queue so it can be processed later
        - Dequeue(Q)
        - Color[u] = black Black means vertex is done
- Once you visit all neighbors of vertex u, theres nothing to do, all neighbors had been considered processed and added to Queue

The operation of BFS





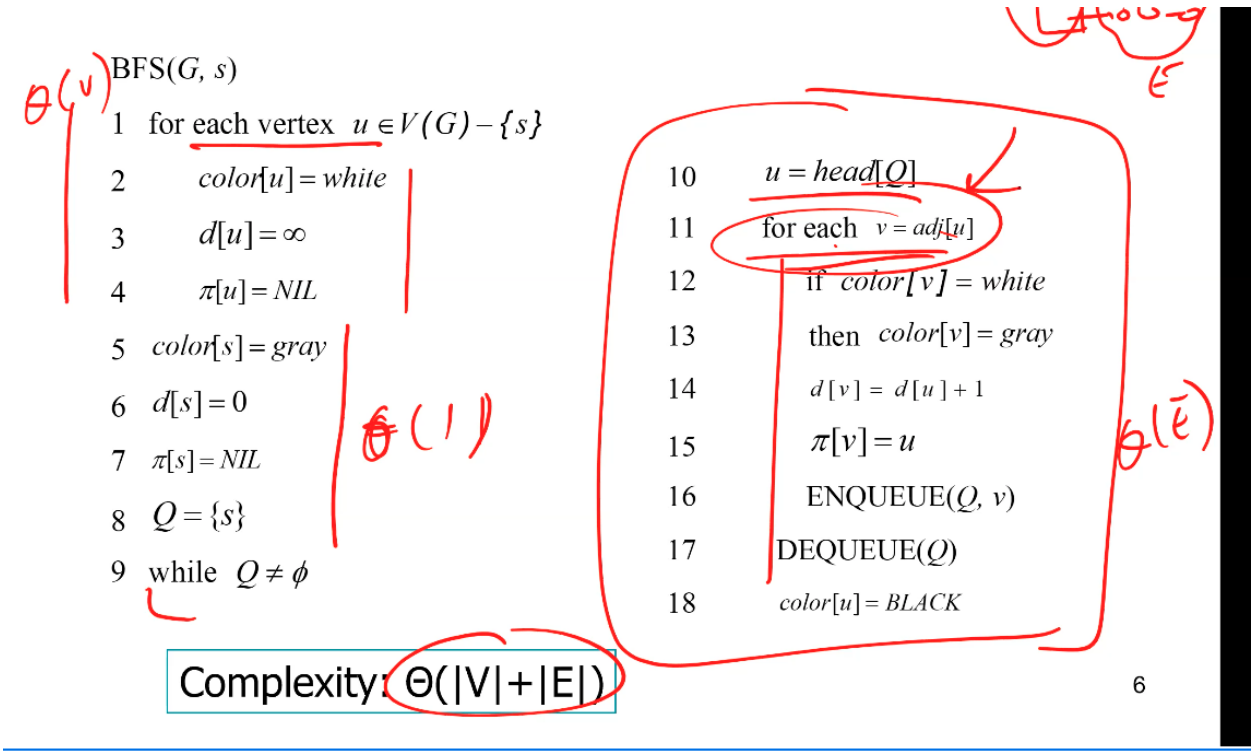
aw the breadth-

- Operation of BFS image explained
    - We start at s, marked with white, distance is infinity because they were not reachable at that point
    - Then we update colors (b) from white to gray, distance is now 1 more than source, then inserted into Q, s has been extracted, mark s black, its been processed, order of vertexes in Q will make a difference in terms of which vertex you visit next
    - Head of Q is w, looking at W now, has 3 neighbors
    - © r is the head of the Q, now we look at their nieghbors (2)
        - Only 1 has not yet been discovered which is (v)
        - So V is the next vertex, change the color of V to grey (d)
        - Extract r from Q and add V
    - At the end there is one remaining element (h)
        - Neighbors of y is black so we dont revisit
        - Extract y from Q, Q is updated and we terminate
    - Following edges for bread-first-search, shows you shortest path
- If we have graph G, show the shortest path from a given vertex source to each and every vertex (Source to vertex X)
    - Run Breadth first search algorithm to get breadth first tree
    - Start with x, following predecessor attribute until you reach root, as you go through vertices, you put them into a list, then you reverse
    - OR
    - Write recursive algorithm, shortest path from source to X is the same as the shortest path from the source to the parent of X + X

print_path(G, source, destination)

- If source = destination
    - Print source
- Else if pie[destination] = nil (If destination parent is not nil, then you have run the breadth first tree, but there was no path from source to vertex, then predecessor is nil (initialization process))

- Print “no path from” source “to” destination — Then there is no path
- Else PRINT-PATH(G,source, pie[destination]) — If path, print path from source to parent of destination recursively, append destination
    - Print destination



Time Complexity is linear of Breadth-first-search

- Initialization requires Theta(v)
- Lines 5-8 is constant Theta(1_
- While loop is visiting every vertex, goes into every Q once, except for each vertex we are also visiting its neighbors, adjacent representation
- Cumulative cost of going through all vertexes is equivalent to going through all edges
    - When you put all adj list together, they cumulatively add up to the set of edges
- Then you go through each edge lines 11+ (Theta(v))

<u>Depth-First Search</u>

- Going as deep as possible from the Source vertex (S) and visit all vertexes within that subtree before you visit the subtrees that are rooted under the next child (start from S go as deep as possible, when you’re done, backtrack and visit next vertex
- Another graph search algorithm
- Depth-first: Go as deep as possible from s, then backtrack to find unvisited nodes
- Builds a depth-first forest (several depth-first trees) in the process
- Places two time stamps on each vertex: d[v] the time at which a vertex is “discovered” and f[v] when it’s processing is finished; Each timestamp is an integer between 1 and 2n
    - Want to mark the vertex discovering time and processing time

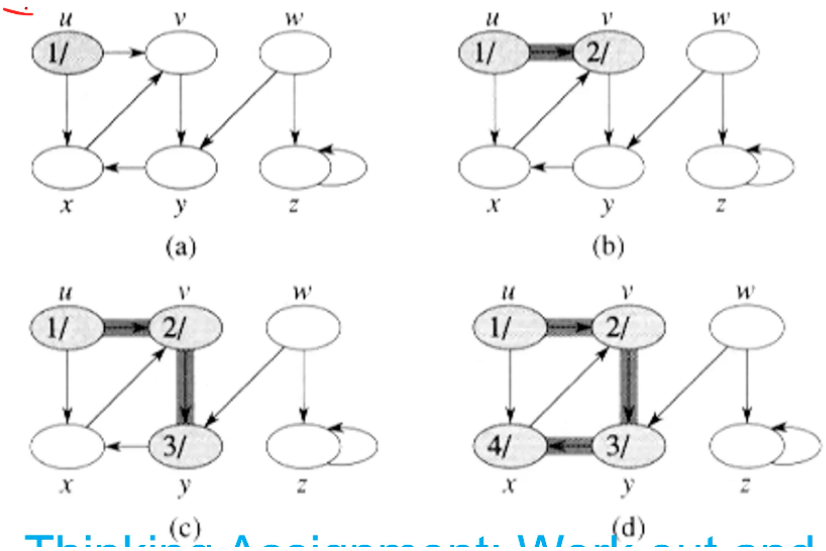
- We use Stack, LIFO

DFS(G)

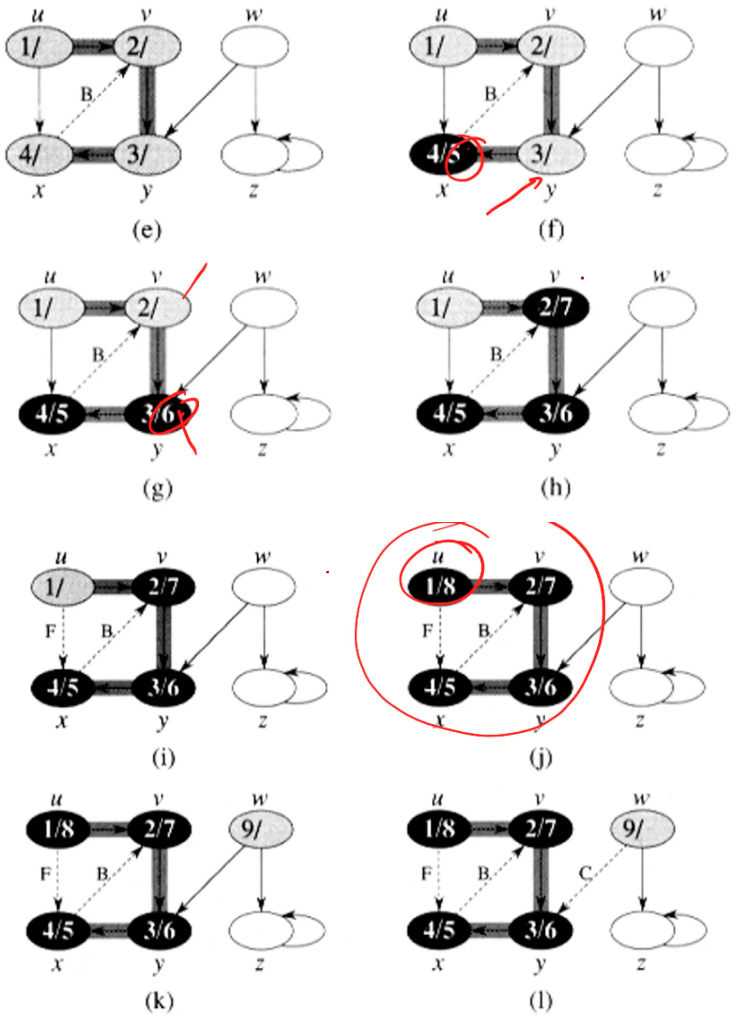
- For each vertex u in V[G]
    - Color[u] = white Not yet discovered
    - Pie[u] = nil Predecessor is nil
- Time = 0 As we discover vertex, time is increment by 1
- For each vertex u in V[G] If vertex hasnt been discovered
    - If color[u] = white We need to emulate the Stack LIFO
        - Then DFS-Visit(u) Recursive, discover all vertices reachable from u
            - If there aren't we haven't touched, we go back in for-loop

DFS-Visit(u) - Which vertices are reachable from vertex u

- Color[u] = gray
- D[u] = time+=1 Increment time and assign to Discovery time of U
- For each vertex v in Adj[u] Backtrack look at next element
    - If color[v] = white If this vertex has not been discovered (checking neighbs)
        - Pi[v] = u We found V through U (predecessor of v is u)
        - DFS-Visit(v)
- color[u] = black Able to discover
- F[u] = time = time + 1



Thinking Assignment: Work out and



- Thinking assignment walkthrough (above)
- (a) u is the first vertex, discovery time = 1, and color is changed to grey

- Call DFS-Visit on U, U has 2 neighbors, order is based on Adj[u,x], set predecessor of v is U (b), then call DFS-visit recursively
    - Discovery time = 1+1
    - Next is vertex Y, we discover Y through DFS-Visit(Y), set Discovery time to 3, change color to grey, call DFS recursively on first neighbor I discovered on adj (and so forth)
    - (d) causes a cycle, which is important
- (g) We backtrack to vertex v
    - Finish time is set to 7 for V (2/7)
    - Backtrack to vertex U, U has 1 neighbor X (which is colored black, it's processed and finished so I discovered a forward edge - another path to that particular vertex X)
- (j) Discovered all vertices from vertex U, there is no place to go
    - Then we backtrack to the for-loop and look for another vertex that has not been discovered
    - After discovering all vertices from U, there are 2 more remaining vertices W and Z that are waiting to be discovered depending on the order you visit them, you end up with a different result
    - Let's assume W is the next one, call DFS visit
- (k) Increment time by 1 from 8->9, change W to grey, and look at neighbors (2 Y and Z)
    - Y is done, we call this cross edge, can occur in two ways:
        - Cross one depth first tree to the other (like in K)
        - If you have a depth first tree and you discover a connection from one branch of that DFT to the other -> Crossing two different paths
- (n) So we don't go to Y, we go to Z, call DFS-Visit(Z), set predecessor of Z to W, increment time by 1 from 9-10, discovery time for Z (10/11)
    - Look at neighbors of Z (1 : itself) the target of the link is a grey vertex (prescence of a back edge or cycle, so we don't recursively call DFS visit on that neighbor/itself)
    - Increment time by 1 -> 10 = 11
- (o) Make z black and finish it, then backtrack to W
- (p) Look at vertex W, are there any neighbors? Increment time by 1: 12, then terminate search starting from W then go all the way back from DFS algorithm to see if there are any remaining vertexes that have not been discovered
    - At this point they all have been (the result is graph p)

<u>Depth-first Forest</u>

- Depth-first forest consists of trees that represent the edges of the original graph that the DFS algorithm traversed
- These edges are called <u>tree edges</u>

- The remaining edges in the original graph can be classified as <u>back, forward, or cross edges</u>
- DGS of an undirected graph will produce only tree and back edges
- DFS of a directed graph may produce tree, back, forward, or cross edges

<- Directed graph

-

$\Theta(V)$

```
DFS(G)
1  for each vertex u ∈ V[G]
2      color[u] = white
3      π[u] = NIL
4  time = 0
5  for each vertex u ∈ V[G]
6      if color[u] = white
7          then DFS-VISIT(u)
```

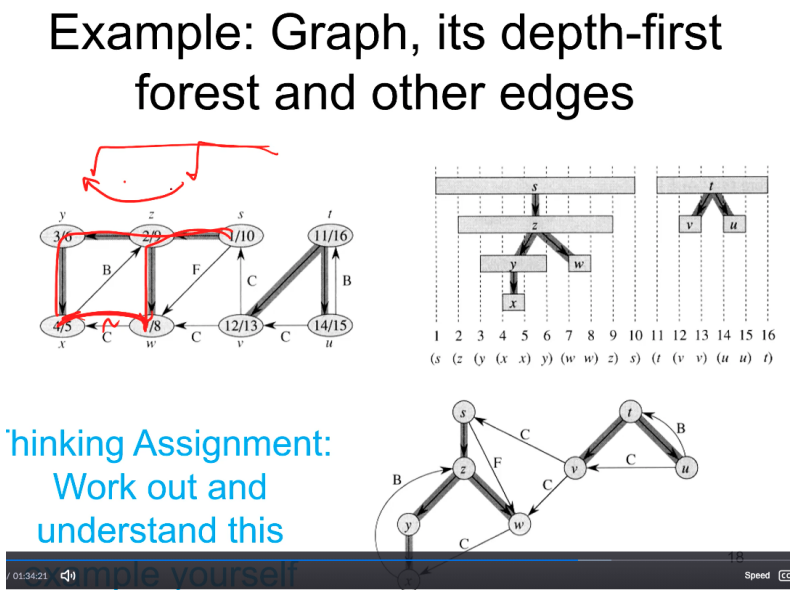
```
DFS-VISIT(u)
1  color[u] = gray
2  d[u] = time = time + 1
3  for each v ∈ adj[u]
4      if color[v] = white
5          then π[v] = u
6          DFS-VISIT(v)
7  color[u] = black
8  f[u] = time = time + 1
```

each

$\Theta(E)$

Complexity: Θ(|V|+|E|)

- Time Complexity (Linear time)
  - Lines 5-7 is iteration over all vertices in the vertex set and calling DFS-Visit
  - DFS-Visit
    - First part is constant
    - For loop-> for each vertex you visit you will touch ONCE, each time you discover a vertex you go through it's set of neighbors in Adj[] and perform set of operations including recursive call
    - It will accumulate performing a number of constant operations on a set of edges, just like breadth-first-search, which is no less than the set of edges Theta(E)

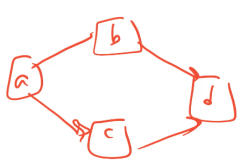


- There is a cross edge from S->Z->W and y->X->W??? (this might be wrong)

<u>Topological Sort</u>

- Of a directed acyclic graph G = (V,E) is a linear ordering of all its vertices such that if G contains an edge (u,v), then u appears before v in the ordering

a b c d
a c b d



- Linear order, a HAS to come first, then its either b or c
- What info from Depth-First-Search can help us establish this order? What can we say about A if we do Depth-First-Search (DFS)? What is the finished time of vertex A compared to the finish time of the others
    - Discovery time is 1, and you have 2 neighbors, let's choose B has neighbor
    - B has 1 neighbor d, discovery time is 3, no place to go from d, so we finished it and set finished time to 4
    - Backtrack to B, no place to go, finish time is now 5
    - Backtrack to A, is there a neighbor we can visit? Yes, C
    - Discover C, time 6, theres 1 neighbor D, but it has been explored, so we finish time = 7
    - Backtrack to A again, no neighbors, finish time = 8
    - Finish time of A is larger than the others, so run DFS, each time a vertex is finished, insert that to the front of the list of which one finished first
        - D visited first, then B, then C, then A
        - A,C,B,D

- Vertex A should appear before c in the ordering, that's what topological sort is saying, creating this linear order by running DFS and identifying discovery/finishing times, after finish time, add to list
- <u>Only if graph does not have a cycle, if there's a cycle then there's a conflict</u>
- What is one thing that separates A from others?
  - It's in-degree, there's no out-degrees
  - If there's a vertex with no out-degrees that means there are no pre-requisites, you can start immediately
- How to use this info for the basis to create linear order of vertices? Look at graph find vertex with in-degree = 0
  - Looking at A first, decrement in-degree of all neighbors and check if there is any remaining vertex with in-degree = 0. There are 2: B or C
  - We choose B, then delete all out-going links so that other tasks that are predicated on B can proceed.

<u>Topological_Sort(G)</u>

- Call DFS(G) to compute finishing time f(v) for each vertex v **<u>(Theta(V+E))</u>**
- As each vertex is finished, insert it onto the front of a link list
- Return the link list of vertices

<u>Different Topological Algorithm</u>

- In-degree of a node is the # of edges coming into it; if a node has in-degree zero, then it means that it is not dependent on any other nodes
  - Therefore, determine the in-degree of all nodes in the graph and store those
  - If there is at least one such node (what does that mean?) output that as first node in TO
  - Remove all outgoing edges from it by decrementing the in-degrees of all nodes connected by those outgoing edges
  - If that makes the in-degree of any node to be zero, output that as the first node in TO
  - Repeat 2-4 steps until all nodes have been output to the TO

----------> Thursday Jul 22, Missed first 30 mins of class

<u>GenericMST(G,w)</u>

- A = null
- While A is not a spanning tree
  - Find edge (u,v) that is safe for A
  - A = A u {(u,v)}
- Return A

- w(T') = w(T) - w(u,v) + w(x,y)

<u>Kruskal(V,E,w)</u>

- A = null (Theta(1))
- For each vertex v in V (Theta(V))
    - makeSET(v)
- Sort E into nondecreasing order by weight (Theta(ElgE))
- For each edge(u,v) taken from sorted (Theta(ElgV))
    - If find(u) != find(v)
        - A = A u {(u,v)} (Constant)
        - union(u,v)
- Return A

- Which edge is next minimum? G and I, they are not in the same disjoint set so we choose it, then two options, Ce or Cf both have value 3, we need to make a choice, we choose E-F
- Next edge that is minimum, Ce will make a cycle (so we wont choose that), D-H (two diff disjoint sets) will not cause cycle so we choose that one
- F-H will bridge gap between two disjoint sets, if we choose 6 there is a safe bet
- Next edge, we choose B-E
- Next edge, E-G, either B-C or G-H, causes cycle, we choose 10 it does not cause cycle
- Next edge, H-I causes cycle (ignore), A-C (12) (causes cycle - ignore), at this point all the vertices are in their spanning tree so Kruskal algorithm will terminate

<u>Time Complexity</u>

- E <= V^2 Largest number of edges in a graph, # of edges is no more than the square
- Lg E <= 2lgV
- lgE = Theta(lgV)
- **<u>Theta(ElgV)</u>**

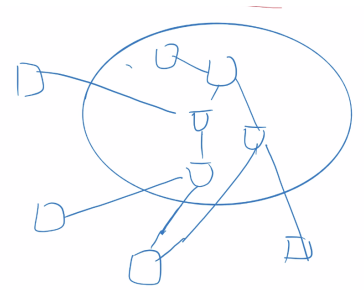
<u>Homework Help Hint</u>

- If you are introducing a new vertex to connect to a new graph (the graph has some vertices and you are inserting a new vertex into the graph)
- Do you have to rerun the algorithm from scratch or can you come up with an efficient way to account for that vertex and those edges?
- Hint: That vertex is in the minimum spanning tree, applying the cut point, we have to look at those 3 edges find out which one is min. And that min. Edge will def. Be the one that connects us to the graph, but you can't stop there, that's the tricky part
    - What about the remaining edges? What needs to be done to be able to account for those edges, can they invalidate some prior edge selection that constitutes the min. Spanning tree

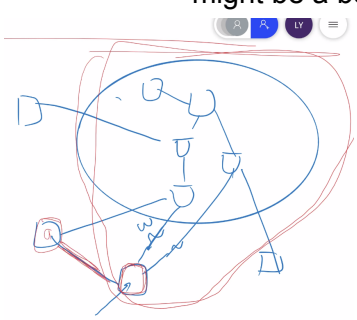
<u>Difference between Kruskal and Prim</u>

- Kruskal is the edges, growing spanning tree one edge at a time

- Prim focuses on one vertex at a time
    - Basic Idea: at any point in time during a step of the algorithm, suppose you have a min. Spanning tree and other vertices are not yet part of the tree



    - We have to associate each vertex a key value = the min. Cost of adding to the spanning tree
    - We have more than 1 edge connecting a vertex to some vertices inside the boundary, we choose the one with the minimum value, the cost is the key to the vertex, the cost of connecting is the KEY
    - Suppose you grow your spanning tree to include that vertex, what does that mean? Given this edge:
    - We need to update our key values in neighbors, might turn out that this edge might be a better alternative



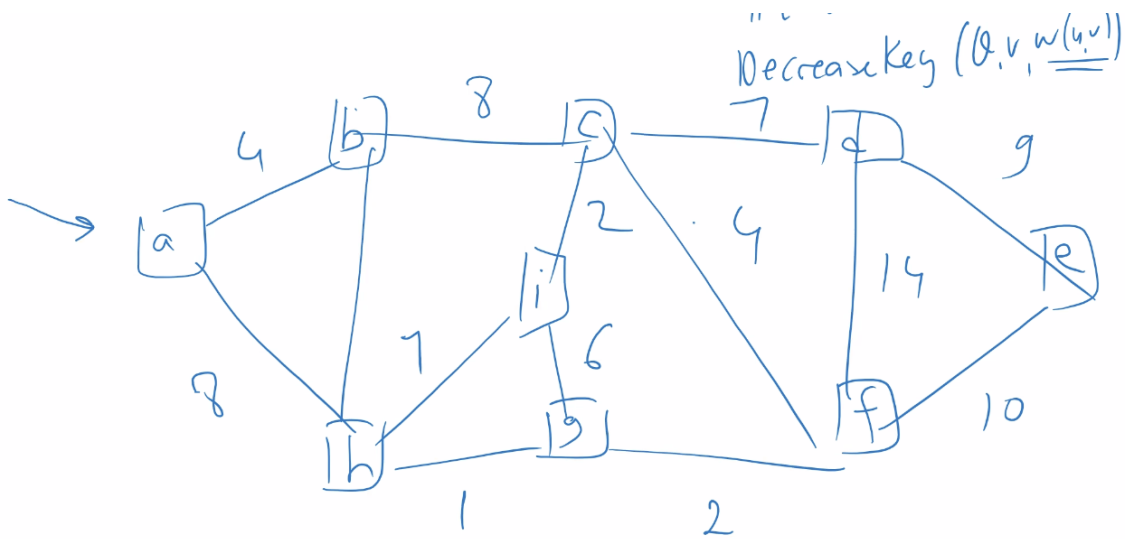
May turn out that the vertex outside is a better option

- each time we introduce a new vertex to spanning tree, we have to visit all of its neighbors and update its cost (key values) as fast as possible
- We can easily justify a priority queue, key value is the basis for making a decision, the decision can result in updates to key values in other vertices (update should be fast like logn)
- Maintain key value and update them

<u>Prim(V,E, w, r)</u> w = weight, r = root of spanning tree (chosen arbitrarily)

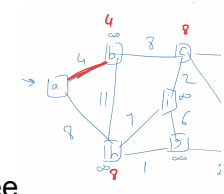
- Q = nil (empty) Contains all vertices that are not in spanning tree
- For each u in V
    - Key[u] = infinity
    - Pi[u] = nil
    - insert(Q,u) insert on Binary heap
- DecreaseKey(Q,r,0) Make root key = 0, highlight
- While Q != nil/empty

- U = ExtractMin(Q) Min. key value, extract r initially
- For each v in Adj[u]
    - If v in Q and w[u,v] < key[v]
        - Neighbor of u is not a member of spanning tree
        - Connecting neighbor is a better option
    - Pi[v] = u - It's a better option, set predecessor
    - DecreaseKey(Q,v,w(u,v))

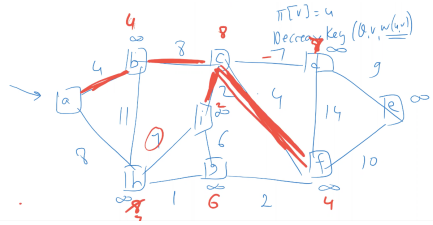


Example of Prim (above)

- Starting with A, key = 0, let's update key for neighbor (initially all infinity)
    - A has 2 neighbors, B and H
    - Cost of b = 4 to connect to spanning tree
    - Cost of h = 8, 4 and 8 are both less than infinity, update key values for both
    - Predecessor for both is A
    - Which vertices have min. Key value? 4, so B will be next vertex to be included in spanning tree



- B is now adding to spanning tree
    - Looking at it's neighbors, C, A, and H. C and H are not in spanning tree, A is not an option. We update key value for C from infinity to = 8
    - We have two vertices with value 8, we choose C
- C is now member of spanning tree
    - Has 4 neighbors, We need to update d, i, and f
    - D = 7, f = 4, and i = 2
    - Now 2 is the minimum value, i is next vertex
- I is in!

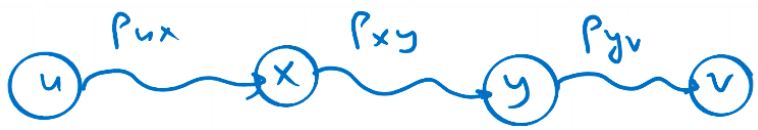
    - We just discovered a better way, decrease key value of h->i - 7
    - Update key value of g to 6
    - F has lowest value. 

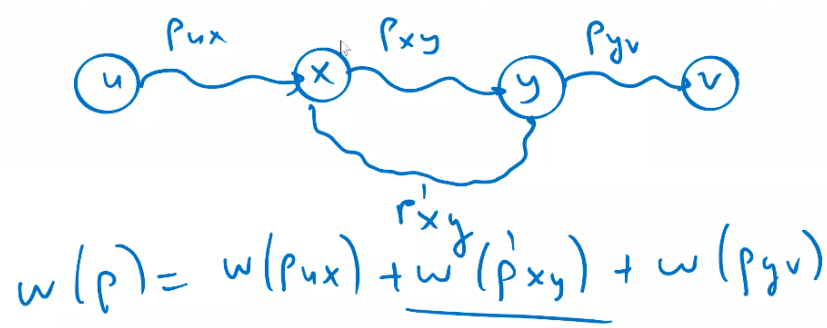
- F is in tree
    - Check neighbors, 4 neighbors, C (in tree already not considered), e = 10, d = 7 still because 14 is bigger, g = 2 (updated)
    - G is next vertex, predecessor is f, will not change anymore
- G is in tree
    - H key is updated to = 1
    - H is next vertex
- H is in tree
    - 4 neighbors, all already in tree
    - Now if condition fails in the algorithm
- What is not a part of the spanning tree, E
    - Extract-Min (9), once extracted, it's a part of the spanning tree
    - Look at neighbors, D and F are both in tree

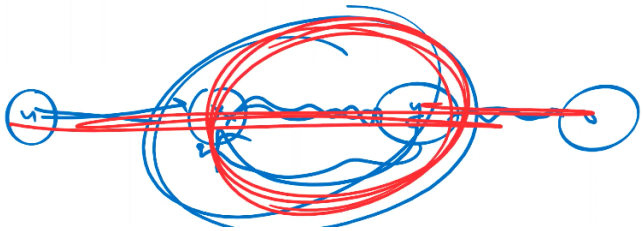
<u>Tuesday July 27</u>

G = (V,E)	- Suppose we have a graph
w : E -> R	
P = <v0,v1...vk>	- We can defined a path as a sequence, starting from initial vertex (v0) and go to (vk) - Edges between those vertices are implicit
w(p) = Sum(from i=1 to k)w(vi-1….vi)	Weight of path is defined as w(p) sum of the edge weights
delta(u,v) = { min{w(p): u -> v}, (INFINITY) }	-Shortestest path weight from u to v (delta) - multiple weights, choose the min weight if there is a path from u to v - otherwise dist is infinity, v is not reachable from u

<u>Shortest Span 3 Properties</u>

- <u>We are not supposed to have a negative weight cycle</u>
    - If a cycle in a path, we can take that path over and over and keep reducing the shortest path estimate to neg. Infinity.
- <u>Optimal substructure property</u>
    - 

    - Any subpath (Pux, Pxy, Pyv) should also be the shortest path from respective vertices (subpath is also a shortest path)
    - 

    - Suppose:
    - Proving p' is a different path, weight of path P'xy then path from x=y
        - Then w(p') < w(p)
        - By assumption we prove that w(p'xy) < w(pxy)
        - Proof by contradiction
- <u>We cannot have positive cycles</u>
    - 
    - We don't need to take this cycle, we don't need to navigate around the u, x, and y vertices, so cycle is unnecessary, you can go from U->X->Y->V
    - There is no Positive cycle, length of shortest path?
        - Length of shortest path can be no more than N-1 edges if you have N vertices in a graph
        - If cardinality of set is N, then vertex set is N, vertex set cannot be no more than N-1

Characteristics Of Algorithm

For each vertex v	
- D[v] = delta(s,v) - Initially infinity	- Shortest path estimate from source vertex, initially infinity, now delta(s,v) - Delta(s,v) is shortest path (what we are trying to compute) - Shortest path estimate
- Pi[v]	- Predecessor of V on shortest path - Pi induce a tree, which is the shortest path tree - Discover/extract shortest path sequence of vertices from source vertex S

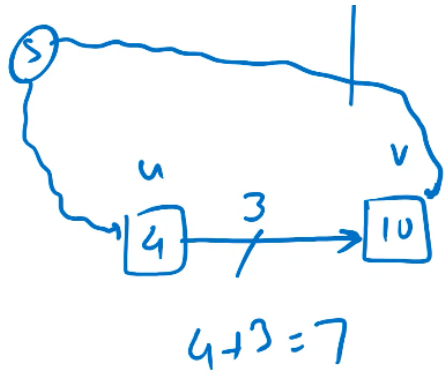
Shortest Path Algorithm

INIT_SINGLE_SOURCE(V,s)	**Initialization**
For each vertex v in V	- Go through each vertex and initialize - For each vertex in a set of vertice
- D[v] = infinity - Pi[v] = NIL	- Have not discovered distance or predecessor
D[s] = 0	- S will be at the root, distance from source to itself is obviously 0
RELAX(u,v,w)	**Relaxation** - Takes weight and two vertices
If d[v] > d[u] + w(u,v)	- If distance of vertex v is actually larger Than current shortest path estimate for U, plus weight of edge from U to V
- D[v] = d[u] + w[u,v] - Pi[V] = u	- Update shortest path estimate for vertex V by assigning smaller value and making sure we update predecessor of V - There is a better path from U to V

Relaxation (Below image) Explanation

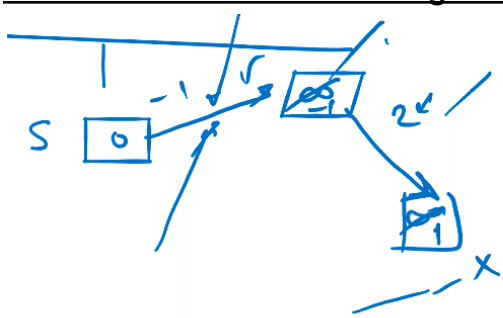
- There is some path I discovered somehow from source to vertex V, whose length is 10, and there is some path from source to vertex U is 4
- We know that there is an edge from U and V, and the weight is 3

- We want to return if reaching vertex V from U is better than current shortest path estimate for vertex V
    - This is the case because 4+3 = 7 < 10, reaching vertex V from source vertex through vertex U is a better option.



    -
    - Discovers updates, next function is basis of iterative updating mechanism

<u>Bellman-Ford Shortest-Path Algorithm</u>



- Distance from S to R is -1, and weight from R to X is 2
- Initially R and X has infinity for labels
- Worst case is N-1 over all edges
- When we go through edge R-X and apply relax function
- Infinity + 2 is still Infinity, so we do not update shortest path estimate for S
- Next edge is from S-R, 0 - 1 is -1 < Infinity
    - So we change shortest path estimate from Infinity to -1
- Next iteration
    - X is still infinity shortest path
    - -1 + 2 = 1 < infinity
    - We just discovered better option to reach X, update X(infinity) to X(1)
- Next iteration, edge from S-R
    - 0-1 = -1 !< -1
- Took 2 iterations from worst case
- Allows us to have negative edge weights (MORE EFFICIENT)

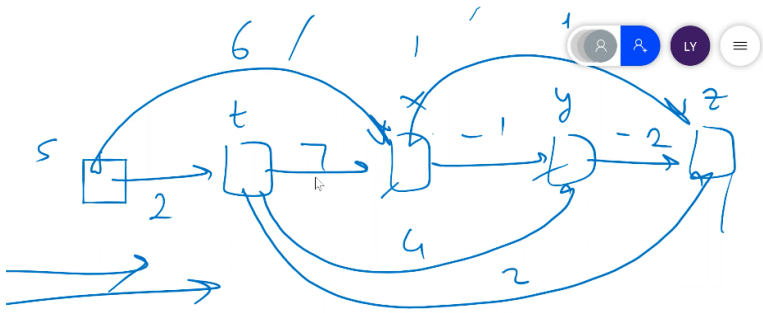
- Determines if there is a neg. Cycle, returns false indicating there is no shortest path in graph, otherwise return shortest path estimate values along with predecessor for each vertex

BellmanFord(V,E,w,s)	**Bellman-Ford**
INIT_SINGLE_SOURCE(v,s)	- Initialize first, we need to make n-1 iterations - Constant **Theta(V)**
For i = 1 to |V| - 1	- Cardinality of V is N, so |V| = N
- For each edge(u,v) in E	- Over all edges, because len. Of shortestest path is no more than n-1, each one will settle, unless there is a neg. Weight cycle - Can **PROVE** here, if none of the vertices have changed, then you can break and terminate, then network is tightly coupled - **Theta(V*E)**
- - RELAX(u,v,w)	- If we apply relax function to edges again, we can keep reducing shortest path estimate of these vertices
For each edge (u,v) in E	- Which is why we make one more iteration to determine if we can continue to reduce
- If d[v] > d[u] + w[u,v] - Return FALSE - Return TRUE	- If there is a vertex that is larger than the shortest path estimate of U + cost of weight from U to V - There is an opportunity to reduce the estimate

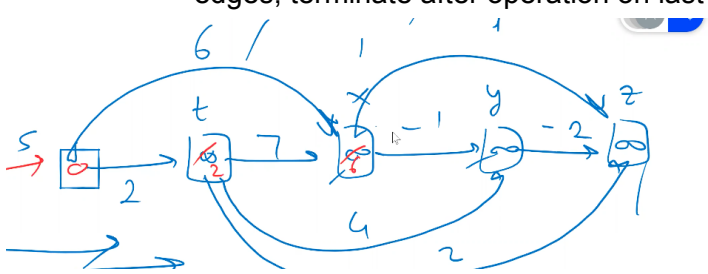
Time Complexity of Bellman-Ford

- We know that single source iterates over all vertices
- Goes through all edges Theta(V*E)
- Then one last iteration through all edges Theta(E)
- So **Theta(V*E)**
    - If E is a connected graph then you can tell it will be **Theta(V^3)** if it is tightly coupled or moderately coupled
    - Where E = V^3

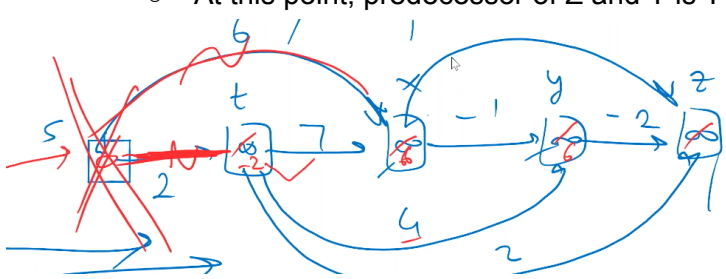
Shortest Paths in Directed Acyclic Graphs (48 minute)



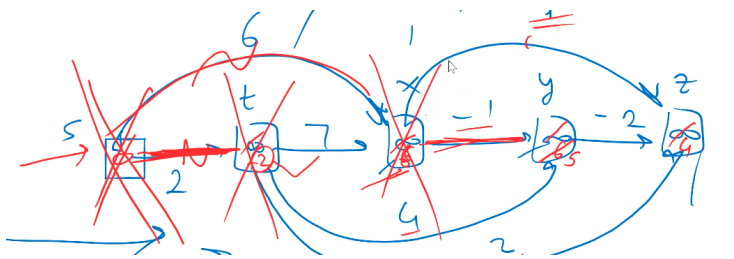
- Suppose you have run the topological sort algorithm, and the vertices are the order, each time we sort the vertices they are at the front of the list
- Each edge has a weight, all edge goes from left to right, there is no back edge, no cycle
- What does this topology tell you from Relax function?
    - Will updates on vertex propagate back? No there is no edge gong back
    - We do not visit edges again
    - Go through each vertices one by one, from left to right, apply relax function to edges, terminate after operation on last vertex



- At this point (above) any updates will not propagate back, so we ignore vertex S and move on to the next vertex T
- T has 3 neighbors, predecessor of T is S, and X is S
    - X -> Path from T to X is 7, which is less than 6, so don't update
    - Y -> 2 + 4 = 6, better than infinity, update Y from infinity to 6
    - Z -> 2 + 2 = 4, update from infinity to 4
    - At this point, predecessor of Z and Y is T



- Looking at X now, has 2 neighbors
    - Y-> 6-1 = 5, better than 6, update Y to 5, predecessor of Y is X
    - Z-> 6+1=7, not better than 4, maintain Z



- Next vertex (loop) is Y has 1 neighbor
    - Z -> 5-2 = 3, 3 < 4, update Z to 3, predecessor of Z is Y

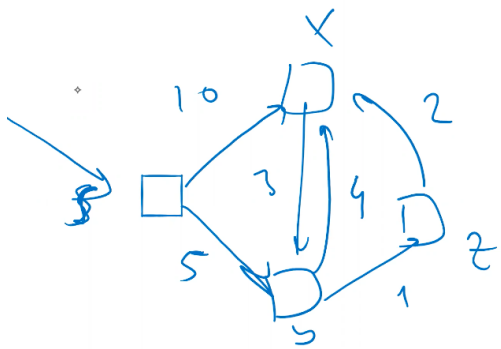
We now have a shortest path Tree, tells us how we can reach from source vertex to each vertex through optimal shortest path

<u>DAG_Shortest_Path(V,E,w,s)</u>	**<u>Directed Acyclic Graphs</u>**
Topologically sort the vertex	**Theta(V+E)**
INIT_SINGLE_SOURCE(V,s)	- Initialize graph - **Theta(V)**
For each vertex u taken in topo. Sorted order	- **Theta(E)**
- For each vertex v in Adj[u]	- Look at edges that connect vertex U to it's neighbors
- - RELAX(u,v,m)	

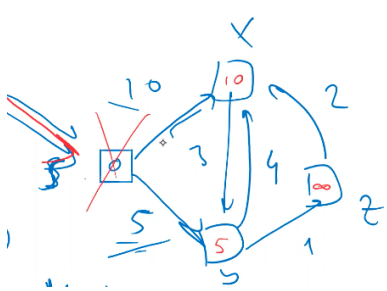
Time Complexity of DAG

- **Theta(V+E)**

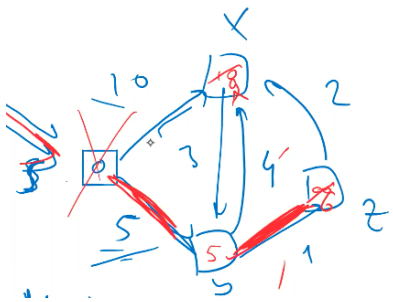
Dijkstra Algorithm Shortest Path



- All edges should have positive weights, NOT NEGATIVE
- Example above, is there any way we can reach Y by taking edge 10 and going through more edges and come up with a length value that is less than 5? 10 is greater than 5, so 10 plus any pos. Number is worse than S->Y (5).
- Key idea: look at the shortest path estimate value for each vertex in the graph, find which one is min. And ignore that vertex and neighbors and determine that we have discovered the shortest path with the min. Across all vertices and keep moving to other edges
- Applying algorithm to above image: Look at neighbors of S, two neighbors (XY) whose distance is initially infinity
    - X -> 0+10 < infinity -> X gets updated to 10
    - Y-> 0+5 < infinity -> Y updated to 5
    - At this point Source vertex is removed from Q (extracted), shortest path value is finalized



- 3 vertex remaining, 10, 5, and infinity, one with smallest value is Y, 5 < 10 and infinity so we EXTRACT and look at it's neighbors
    - Shortest path to Y is through edge 5, Predecessor of Y is S
- Now look at neighbors for Y, two neighbors (ZX)
    - Z-> 5+1 = 6 < infinity, update Z to 6, predecessor of Z is Y (highlight edge)
    - X-> 5+4 = 9 < 10, update X to 9, predecessor of X is Y



- X = 9, Z = 6, Z is min, so we EXTRACT
- Z has 1 neighbor (X)
    - X-> 6+2 = 8, 8 < 9, update X to 8, predecessor is now Z
    - Remaining vertex is Q is X, EXTRACT look at neighbors
- X (redundant to apply relax func) has 1 neighbor (Y)
    - Y-> 8+3 = 11, 11 !< 5, we already knew that, confirms our belief

Dijkstra(V,E,w,s)	**Dijkstra's Algorithm**
INIT_SINGLE_SOURCE(V,s)	- Initialize - **Theta(V)**
S = empty/nil	- Set contains all vertices that we discovered shortest path estimates - **Theta(1)**
Q = V	- Priority queue, at each point we need to get the min. - Binary heap that helps us - Applying build heap func. (linear time) - **Theta(V)**
While Q != empty	- As long as there are still vertices that we need to calc shortest path - Each vertex extracted once + Extract Min func
- U = ExtractMin(Q)	- Once extracted, look at neighbors - Extracting from priority q is height of tree with is Logn or Logv and apply it V times - **Theta(VlogV)**
- S = S u {u}	- Set S now includes one more vertex where we were able to calc shortest path - Look at it's neighbors

- For each vertex v in Adj[u]	- (Image above) looking at S-Y - Going through all vertices and visiting Adj. lists = set of edges - E * cost of applying relax func - For each edge, doing lgV because relax each edge - **Theta(E*lgV)**
- - RELAX(u,v,w)	- Appears to be constant operation - When we have chance to decrease d[v] to lower value, we are decreasing key value for vertex, have to maintain heap property - **lgV**

Time complexity:

- Largest factor is **Theta(ElgV)**

<u>Thursday, July 29th Lecture</u>

A Broad Look at Complexity

<u>Levels of algorithmic complexity</u>

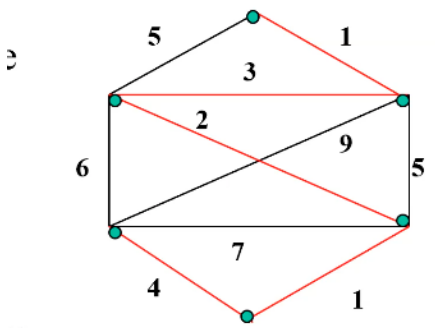
	<u>if n=8 and k=2</u>
constant time: O(1)	1
less-than-linear: O(loglogn)	approx. 1
less-than-linear: O(logn)	3
linear: O(n)	8
more-than-linear: O(nlogn)	24
polynomial: $O(n^k)$	64

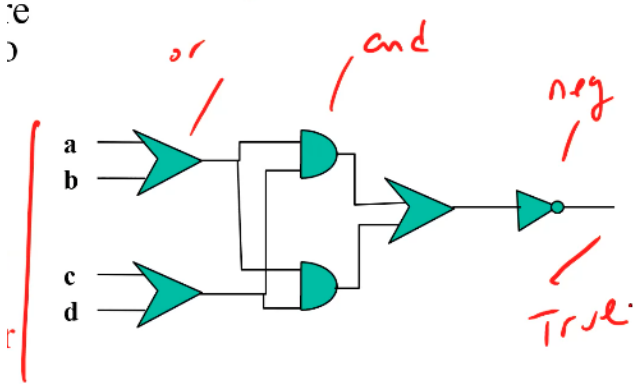
efficient

inefficient

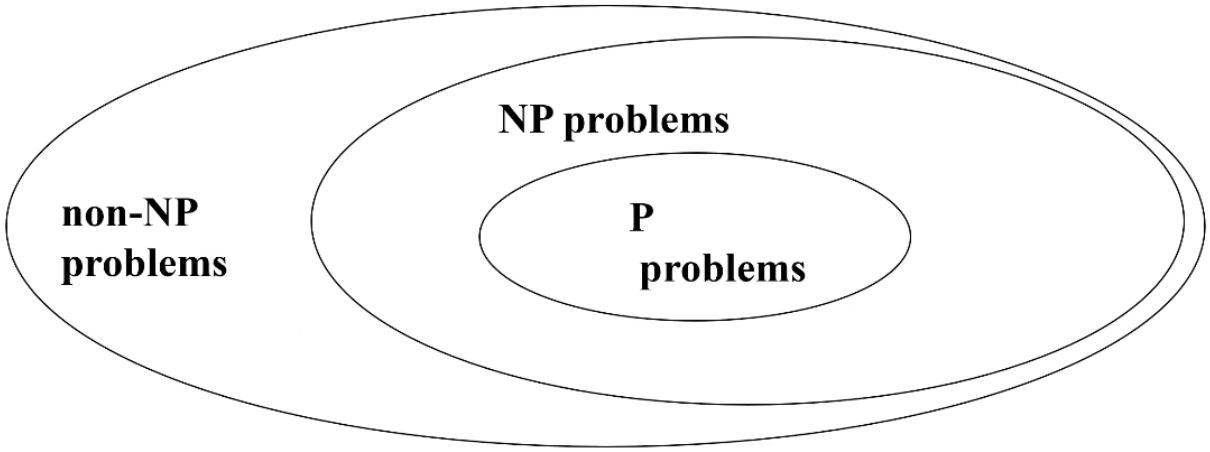
<u>Problem Classes</u>

- P(polynomial)
  - All problems that can be solved by algorithms fo polynomial complexity

    - All problems we discussed in this course are in P
- NP (Non-deterministic Polynomial)
    - This includes the problem class P
    - Also includes problems to solve which polynomial time or faster algorithms are not known (i.e., all known algorithms to solve these problems have worse than polynomial complexity)
    - But not all problems with worse than polynomial complexity are in NP
- To be in NP problems must be such that their solutions can be verified by algorithms of polynomial complexity or less
    - E.g. Boolean circuit satisfiability problem
- 



<u>Every P problem is also an NP Problem</u>



- Np - really difficult problems to solve, P is a subset of NP

<u>Is every NP problem a P problem?</u>

- I.e., if a problem is in NP but it is not in P, is it possible that it can be solved by a polynomial time or faster algorithm?
- Is p = NP? Nobody knows!
- If it was, then the two problem classes will be the same, i.e., P=NP
- An NP-Complete problem is one such that ALL problems in NP can be transformed to particular instances of it (so that you can construct a solution to the original NP problem from the solution to this instance of the NP-Complete problem and vice versa), where this translation can be done in polynomial time or less for any NP problem
    - The invention of an efficient (polynomial time or faster) algorithm to solve an NP-complete problem will provide the means to solve ALL NP problems efficiently
    - Satisfiability was the first NP-Complete problem to be discovered

<u>Undecidable Problems</u>

- The halting problem, no such halting procedure
- Proof by contradiction

The Halting Problem
Loop(P)
output = HP(P,P)
If output = yes then
    - Infinite loop
Else
    - Exit
End Loop

Final Exam

- Graph theory
    - Study algorithms, and run them on examples
    - Understand process
    - Simple graph -> minimum spanning tree version know that without knowing any algorithm
    - BFS and DFS
        - DFS -> classification of edges, backwards, forwards, and cross edges
    - Depth-first forest
    - Topological sort
- Questions from previous exams,
    - Why is stable sorting important in ready sort
    - Apply partition, what configuration do you end up with?
- Time complexity
    - Given table, cost occurred at each level of recursion
- P and NP
- Designs
    - Greedy algorithms
    - Bellman-ford
    - Different algorithm, incremental iterative greedy (be able to explain)
- Why we use a priority queue in a dkystras shortest path algorithm