Asymptotic analysis

Big O notation

$O(g(n)) = \{ f(n) : \exists c, n_0 \text{ s.t. } (c > 0, n_0 > 0) \wedge (\forall n \geq n_0 \; 0 \leq f(n) \leq c \cdot g(n)) \}$

the set of functions s.t. past some problem size n_0, $f(n) \leq c \cdot g(n)$ and $g(n)$ is the tightest bound

$f(n) = O(g(n)) \Rightarrow f(n) \in O(g(n))$

$f(n)$ belongs to the set of $O(g(n))$.
we say that $g(n)$ is an asymptotic upper bound of $f(n)$

deriving $O(f(n))$

1. count no. of operations involved in program execution
   - 1.1 formulate sum
   - 1.2 simplify / factorize / re-express $\Rightarrow$ get some $f(n)$
2. take dominant term $\Rightarrow$ $g(n)$
3. test for tightness of bound

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

(algorithm runs; bound)

- $< \infty \Rightarrow f(n) = o(g(n))$ in general
- $= K, K \in \mathbb{R} \Rightarrow$ tight bound
  - ↳ intuition: "tight" bound will leave behind constant of proportionality c on bounding function at large problem size
- $= 0 \Rightarrow$ loose bound

case analysis

↳ best, worst, average, amortized → $\frac{\text{number of ops}}{\text{number of calls}}$

families of functions

0 →

1. log functions (base of log does not matter since scalar transform) $\log(\log n^k) = o(\log \log n) \ldots < (\log n)^k = (\log n^k) = o(\log n)$
2. sublinear power functions $n^k = o(n^k)$, $k \in (0,1)$, k is specific
3. linear functions $O(n)$
4. linearithmic functions $n \log n > \log n! = o(n \log n)$ (upper bound trick)
5. polynomial functions $n^k = o(n^k)$, $k \in (1, \infty)$, k is specific
6. Exponential functions K^{an+b}, K, a are specific (expand). a takes precedence. eg. $3^n < 2^{2n}$
7. Factorial functions $(an+b)!$, a, b are specific
8. Tetration functions n^{an+b}, a, b are specific

recursive algorithms

recursion tree method

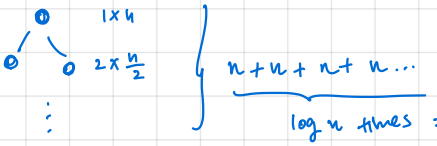
1. Define recurrence relation

eg. $T(n) = \begin{cases} T(\frac{n}{2}) + T(\frac{n}{2}) + n & \text{if } n > 1 \\ n & \text{if } n = 1 \end{cases}$

(cost at each call; cost at base case)

2. Determine cost at each level of tree

eg. split into 2 each time, each call is n



$n + n + n + n \ldots$ ($\log n$ times) $\Rightarrow n \log n$

3. sum costs & simplify

↳ AP, GP

↳ upper bound trick → to family form / → to an AP / GP

upper bound trick

↳ substitute to something that is consistently bigger to get a more uniform expression

eg. $\log n! = \log(n)(n-1)(n-2)\ldots(1)$

$= \log n + \log(n-1) + \log(n-2) \ldots$

$< \log n + \log n + \log n \ldots$

$< n \cdot \log n$

changing of variables

↳ substitute for something easier to work with

eg. $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$

↳ set $m = \log n$

$T(2^m) = 2T(2^{m/2}) + m$

↳ set $S(m) = T(2^m)$

$S(m) = 2S(m/2) + m$

$\vdots$

$T(n) = T(2^m) = S(m) = O(m \log m)$

$= O(\log n \cdot \log \log n)$

formulae

Arithmetic series

$1 + 2 + 3 \ldots n = \frac{n(n+1)}{2}$

↳ more generally, if $a_n = a_{n-1} + c$,

$a_1 + a_2 \ldots a_n = \frac{n(a_n + a_1)}{2} = \frac{n}{2}(2a + (n-1)c)$

Geometric series

$\sum_{i=0}^{n} 2^i = 1 + 2 + 4 + 2^n = 2^{n+1} - 1$

↳ more generally, if $a_n = c \cdot a_{n-1}$, $c \neq 1$,

$\sum_{i=0}^{n} a_i = a_1 + a_2 + a_3 \ldots = a_1 \frac{c^{n-1}}{c-1}$

↳ if $0 < c < 1$:

$\sum_{i=1}^{\infty} a_i = \frac{a_1}{1-c}$

sum of squares

$\sum_{i=1}^{n} i^2 = 1 + 4 + 9 \ldots n^2 = \frac{n(n+1)(2n+1)}{6}$

logarithm rules

$\log_a b = \frac{\log_c b}{\log_c a}$

$x^k = a$

$k = \log_x a$

sum of cubes

$\sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4}$

Harmonic series

$S_n = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} \ldots + \frac{1}{n} = \frac{a_n}{n!}$

$a_n = n\, a_{n-1} + (n-1)!$

$a_1 = 1$

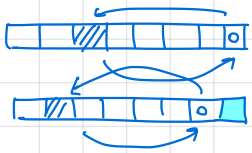
$S_n - S_{n-1} = \frac{1}{n}$

linear algorithms

① comparison-based sorting algorithms

selection sort

1. given an array of n items, iterate through to find the largest item
2. swap the largest item w/ the last item in the array
3. Repeat steps 1, 2 but each time, exclude the newly-swapped last item in the array



```
for i = arr.length - 1  to  i = 1
    max idx = 0
    for j = 1  to  i
        if  arr[j] > arr[max idx]
            max idx = j
    swap(arr, max idx, j)
```

$f(n) = \underbrace{(n-1 + n-2 + \dots 1)}_{\text{finding max @ each sub array}} + \underbrace{n-1}_{\text{swapping}}$

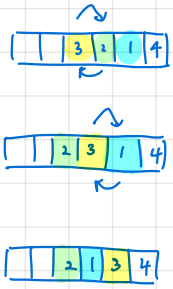
$= \frac{n(n-1)}{2} + (n-1)$

$= O(n^2)$

---

bubble sort

1. iterate through an array $n-1$ times (no. of pairs). Each time two elements are in the wrong relative order, swap them



larger items bubbled up, smaller bubbled down

```
isSorted = True
for i = 1  to  arr.length - 1
    for j = 1  to  arr.length - 1
        if  arr[j-1] > arr[j]
            swap(arr, j-1, j)
            isSorted = False
if  isSorted
    return
```

↳ boolean flag so that single pass for sorted arr.

best case

↳ array is sorted

$f(n) = n = O(n)$

worst = average case

↳ array is not sorted

$f(n) = (n-1)(n-1)$

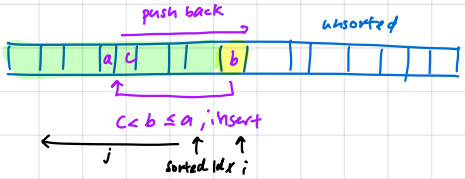
$= O(n^2)$

---

insertion sort

idea: start w/ one card in your hand, pick the next card and insert it in the right position in your hand

1. Look at the first card in the unsorted subarray and compare w/ the last element in the sorted subarray
   1.1 Iteratively compare downwards until correct position is found, insert element at that position




```
sorted idx = 0
for i = 1  to  arr.length - 1
    val = arr[i]
    if arr[i] < arr[sorted idx]
        for j = sorted idx  to  0
            if  arr[i] ≥ arr[j]
                arr[j+1] = val
                break
            // shift back
            arr[j+1] = arr[j]
    sorted idx += 1
```

best case

↳ array is sorted, never enters inner loop

$f(n) = n-1 = O(n)$

worst case

↳ array is reverse sorted, always insert at front

$f(n) = 1 + 2 + 3 \dots n-1$

$= \frac{n(n-1)}{2} = O(n^2)$

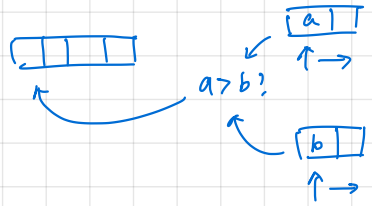
merge sort — Divide & Conquer #1

idea: a single-element array is vacuously a sorted array

Divide: split array into subarrays

Conquer: sort subarrays recursively by splitting until they are single element arrays

Combine: merge subarrays by comparing them in a "queue like" fashion



all work is done here

mergesort algorithm

inputs: start, end, arr (inclusive)

```
if start == end
    return
else
    mid = (start + end) // 2
    merge sort (start, mid, arr)
    merge sort (mid+1, end, arr)
    merge (start, end, mid, arr)
```

merge subroutine

inputs: start, end, mid, arr (inclusive)

```
length = end - start + 1
temp = new array[length]
i = 0
a_i = start
b_i = mid + 1
while a_i ≤ mid and b_i ≤ end
    a = arr[a_i]
    b = arr[b_i]
    if a ≤ b
        temp[i] = a
        a_i++
    else
        temp[i] = b
        b_i++
    i++
while a_i ≤ mid
    temp[i] = arr[a_i]
    a_i++, i++
while b_i ≤ end
    temp[i] = arr[b_i]
    b_i++, i++
for i = start to end
    arr[i] = temp[i - start]
```

queue adding

stable merging ⇒ relative order of elements in a and b preserved! on the condition that a ≤ b

adding remainder

copying result over

Analysis

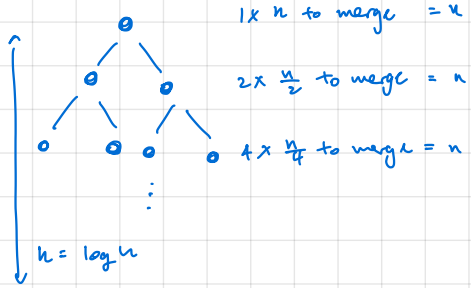
Each merge call of subarray size K:

$f(K) = K + K$ ← copying over $= O(K)$

↑ constructing array → space complexity of $O(K)$

$$T(n) = \begin{cases} T(\frac{n}{2}) + T(\frac{n}{2}) + n & n > 1 \\ 1 & n = 1 \end{cases}$$

$f(n) = T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + n \dots$



$f(n) = n \cdot \log n = O(n \log n)$

↳ for both best & worst case

Quick sort — Divide & conquer #2

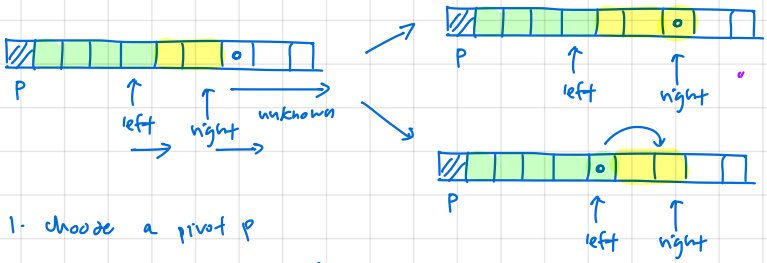
Idea: an array w/ size > 1 is sorted if for any element, left is smaller, right is larger. If left or right do not exist, then vacuously true.

Divide: choose some pivot p, compare all elements to p. If element smaller than p, put to left of p. Else put to right.

↗ all work done by a sophisticated divide strategy

Conquer: keep subdividing

Combine: nature of divide algorithm means no need for special combine algorithm



1. choose a pivot p
2. look at an unknown x
3. if $x \geq p$, swap w right+1, right ++
4. if $x < p$, swap w left+1, left ++

quickSort algorithm

input: start, end (inclusive), arr

```
if start < end
    pivotIdx = partition (start, end, arr)   // splits arr. by reference & puts pivot in the "right spot"
    quickSort (start, pivot-1, arr)
    quickSort (pivot+1, end, arr)            // recurse on split @ pivot w/o it
```

partition algorithm

input: start, end (inclusive), arr

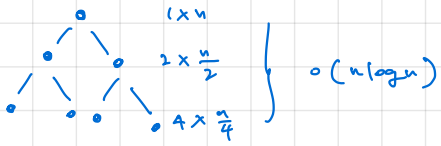
```
pivot = start
idx = start          // correct pivot position
for i = start+1 to end
    if arr[i] < p    // if should be on left
        swap (i, idx+1, arr)   // swap w left
        idx ++       // create "space" on left
swap (start, idx, arr)   // put pivot in right position
return idx
```

↳ useful: returns center of sorted array, w smallest center being kth element itself

Analysis

best case: perfect 50-50 split each time

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \underbrace{n-1}_{\text{iterating through to compare for partition}} + 1 \rightarrow \text{for final swap}$$

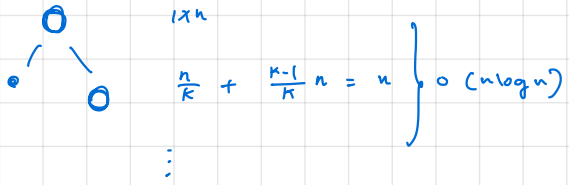


$\} \; O(n \log n)$

average case: some fractional split eg k on average

$$T(n) = T\left(\frac{n}{k}\right) + T\left(n\left(\frac{k-1}{k}\right)\right) + n - 1 + 1$$

if $0 < k < 1$, then maximal height of tree $= \log_k n$



$\} \; O(n \log n)$

worst case: array sorted in reverse order

↳ always 1 : n-1 split, so go through n-1 w/o shifting pivot

$$f(n) = n-1 + n-2 + n-3 \ldots + 1 = \frac{n(n-1)}{2} = O(n^2)$$

③ Linear time/ sorting algorithms

1) decision tree model of comparison based sorts
   - ↳ there are $n!$ permutations of any array
   - ↳ each comparison can be seen as choosing between children nodes of a binary decision tree ⇒ there are $n!$ leaves
   - ↳ to get to a leaf node, we need at least $\log n!$ decisions

   ⇒ asymptotically optimal comparison sorts are $O(n \log n)$

2) non-comparison sorting algorithms

(radix sort)

idea: in all numbers, their digit positions indicate relative size
- ↳ we can iterate through each digit position and group them by their order

$f(n) = O(d \cdot n)$

⇒ implicit comparison

eg. 1234 (A) vs 4231 (B) → if A & B have same next digit, A > B

```
input : arr, d (no. of digits (pad if necessary)), r

for j = d down to 1                          // radix base
    initialize r queues
    for i = 0 to n-1
        K = arr[i][j]
        queues[K].queue(arr[i])              // add item to queue for digit value of K. Queue is used to preserve relative order

    for q = 0 to r-1
        while queues[q].notEmpty()           // replace array w all items in group 0, then group 1, and so on
            arr[i] = queues[q].dequeue()
```

each digit gives us more information about relative ordering
⇒ preserve relative ordering at each level needed

④ selection algorithms $O(n)$

(minimum / maximum): trivial. $O(n)$

(k^{th} element): use partition algorithm, terminating when pivot idx = k

```
input : start, end, arr, k
pivotIdx = partition(start, end, arr)
if k = pivotIdx
    return arr(k)
else if k < pivotIdx
    return quickSelect(start, pivotIdx-1, arr, k)
else
    return quickSelect(pivotIdx+1, end, arr, k)
```

quickSelect( ) ⇒ median

$\frac{N}{2}+1$ (odd) or $\frac{N}{2}, \frac{N}{2}+1$ (avg.)

(simultaneous min and max)
- ↳ naive min/ max → $2n$
- ↳ by doing simultaneous comparison, can be $3\lfloor \frac{n}{2} \rfloor$, by comparing 2 elements to each other, then smaller to min, larger to max

```
for i = 0 to arr.length-2 by 2
    a = arr[i]
    b = arr[i+1]                 // in pairs
    if a > b                     // compare w each other
        toMin = b
        toMax = a
    else
        toMin = a
        toMax = b
    if min > toMin               // compare w current min & max
        min = toMin
    if max < toMax
        max = toMax
```

binary search

Iterative approach

```
binarySearch(arr, size)
        loop until beg is not equal to end
  midIndex = (beg + end)/2
  if (item == arr[midIndex] )
    return midIndex
  else if (item > arr[midIndex] )
    beg = midIndex + 1
  else
    end = midIndex - 1
```

Recursive approach

```
binarySearch(arr, item, beg, end)
  if beg<=end
    midIndex = (beg + end) / 2
    if item == arr[midIndex]
      return midIndex
    else if item < arr[midIndex]
      return binarySearch(arr, item, midIndex + 1, end)
    else
      return binarySearch(arr, item, beg, midIndex - 1)

  return -1
```

> *Say you have the sequence* `1,2,5,3,0`*. Walking backwards from the end, the first non-increasing element is* `2`*. Again walking backwards, the first element larger than* `2` *is* `3`*. You switch them,* `1,3,5,2,0`*, and then reverse the suffix,* `1,3,0,2,5`*. 13025 is the next largest number after 12530 that you can make with those digits.*

permutations

```
procedure generate(k : integer, A : array of any):
    if k = 1 then
        output(A)
    else
        for i := 0; i < k; i += 1 do
            generate(k - 1, A)
            if k is even then
                swap(A[i], A[k-1])
            else
                swap(A[0], A[k-1])
            end if

        end for
    end if
```

Claim: If array A has length n, then performing Heap's algorithm will either result in A being "rotated" to the right by 1 (i.e. each element is shifted to the right with the last element occupying the first position) or result in A being unaltered, depending if n is even or odd, respectively.

Basis: The claim above trivially holds true for $n = 1$ as Heap's algorithm will simply return A unaltered in order.

Induction: Assume the claim holds true for some $i \geq 1$. We will then need to handle two cases for $i + 1$: $i + 1$ is even or odd.

If, for A, $n = i + 1$ is even, then the subset of the first i elements will remain unaltered after performing Heap's Algorithm on the subarray, as assumed by the induction hypothesis. By performing Heap's Algorithm on the subarray and then performing the swapping operation, in the kth iteration of the for-loop, where $k \leq i + 1$, the kth element in A will be swapped into the last position of A which can be thought as a kind of "buffer". By swapping the 1st and last element, then swapping 2nd and last, all the way until the nth and last elements are swapped, the array will at last experience a rotation. To illustrate the above, look below for the case $n = 4$

```
1,2,3,4 ... Original Array
1,2,3,4 ... 1st iteration (Permute subset)
4,2,3,1 ... 1st iteration (Swap 1st element into "buffer")
4,2,3,1 ... 2nd iteration (Permute subset)
4,1,3,2 ... 2nd iteration (Swap 2nd element into "buffer")
4,1,3,2 ... 3rd iteration (Permute subset)
4,1,2,3 ... 3rd iteration (Swap 3rd element into "buffer")
4,1,2,3 ... 4th iteration (Permute subset)
4,1,2,3 ... 4th iteration (Swap 4th element into "buffer") ... The altered array is a rotated version of the original
```

If, for A, $n = i + 1$ is odd, then the subset of the first i elements will be rotated after performing Heap's Algorithm on the first i elements. Notice that, after 1 iteration of the for-loop, when performing Heap's Algorithm on A, A is rotated to the right by 1. By the induction hypothesis, it is assumed that the first i elements will rotate. After this rotation, the first element of A will be swapped into the buffer which, when combined with the previous rotation operation, will in essence perform a rotation on the array. Perform this rotation operation n times, and the array will revert to its original state. This is illustrated below for the case $n = 5$.

```
1,2,3,4,5 ... Original Array
4,1,2,3,5 ... 1st iteration (Permute subset/Rotate subset)
5,1,2,3,4 ... 1st iteration (Swap)
3,5,1,2,4 ... 2nd iteration (Permute subset/Rotate subset)
4,5,1,2,3 ... 2nd iteration (Swap)
2,4,5,1,3 ... 3rd iteration (Permute subset/Rotate subset)
3,4,5,1,2 ... 3rd iteration (Swap)
1,3,4,5,2 ... 4th iteration (Permute subset/Rotate subset)
2,3,4,5,1 ... 4th iteration (Swap)
5,2,3,4,1 ... 5th iteration (Permute subset/Rotate subset)
1,2,3,4,5 ... 5th iteration (Swap) ... The final state of the array is in the same order as the original
```

The induction proof for the claim is now complete, which will now lead to why Heap's Algorithm creates all permutations of array A. Once again we will prove by induction the correctness of Heap's Algorithm.

Basis: Heap's Algorithm trivially permutes an array A of size 1 as outputting A is the one and only permutation of A.

Induction: Assume Heap's Algorithm permutes an array of size i. Using the results from the previous proof, every element of A will be in the "buffer" once when the first i elements are permuted. Because permutations of an array can be made by altering some array A through the removal of an element x from A then tacking on x to each permutation of the altered array, it follows that Heap's Algorithm permutes an array of size $i + 1$, for the "buffer" in essence holds the removed element, being tacked onto the permutations of the subarray of size i. Because each iteration of Heap's Algorithm has a different element of A occupying the buffer when the subarray is permuted, every permutation is generated as each element of A has a chance to be tacked onto the permutations of the array A without the buffer element.

⑤ Evaluating sort algorithms

1) in-place sorts : a sort is in-place if it requires a constant amount of extra space in the heap during the sorting process.

2) stable sort : a sort is stable if the relative order of elements with the same key value is preserved

	best	worst	average	stability	in-place
selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	x	✓
bubble	$O(n)$	$O(n^2)$	$O(n^2)$	✓	✓
insertion	$O(n)$	$O(n^2)$	$O(n^2)$	✓	✓
merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	✓	x
quick	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	x	✓
radix	$O(d(n+b))$	$O(d \cdot n)$	$O(d \cdot n)$	✓	x

d: digits, b: buckets (0-10)

⑥ Java sort API

↳ Arrays → Arrays.sort

↳ List → Collections.sort

↳ generic array → comparator interface :: int compare, bool equals

1. binary search even if don't hit, will land where it would have been ⇒ can get info about would-be neighbors

List ADT

① List ADT

1) ADT : to store data dynamically

insertion	deletion	retrieval	assessment
void addAtIndex (T item, int i)	T removeAtIndex (int i)	T getItem (int i)	boolean isEmpty()
void addFront (T item)	T removeFront ()	T getFirst ()	boolean contains (T item)
void addBack (T item)	T removeBack ()	T getLast()	int size()
			int indexOf (T item)

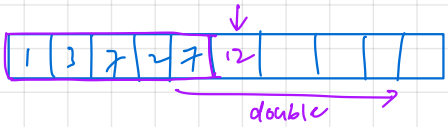
dependent on search algorithm being implemented.
O(n log n) in general by sort then binary search

2) implementations & complexity

Array list

↳ how it works:

1. uses an array under the hood, doubling or halving in size dynamically
2. elements occupy contiguous memory, caching friendly
3. uses an integer to keep track of size
4. "shifts" are done by copying



size = n

	method	how? time & space complexity?
insertion	void addAtIndex (T item, int i)	- adding to non-back, need to copy & shift remaining elements → O(n)
	void addFront (T item)	- adding to back, O(1), if need to enlarge O(n), amortized is O(1)
	void addBack (T item)	
deletion	T removeAtIndex (int i)	- removal like insertion generally needs to shift remaining items forward O(n)
	T removeFront ()	
	T removeBack ()	- removing from back is direct, O(1)
retrieval	T getItem (int i)	- access via indexing O(1).
	T getFirst ()	
	T getLast()	

1. amortized time complexity = $\frac{\text{total cost}}{\text{total no. of ops}}$
2. Each insert is O(1), doubling is O(n+1)
3. amortized time complexity = $\frac{\overbrace{n}^{\text{prior ops}} + \overbrace{(n+1)}^{\text{copying \& insert}}}{n+1} \leftarrow \text{total ops} < \frac{2n+2}{n+1} = 2$

Linked list

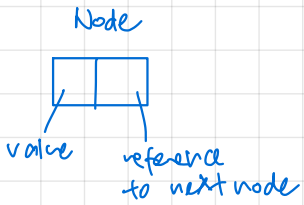
↳ how it works:

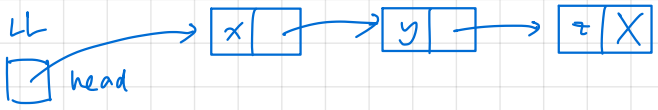
1. Each item in the list is stored in a node, which also contains a pointer to the next node
2. elements occupy non-contiguous memory → no need for resizing!
3. LL object contains reference to head node and operates on it by traversal

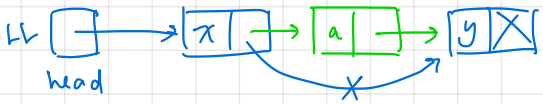
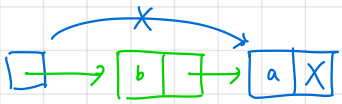
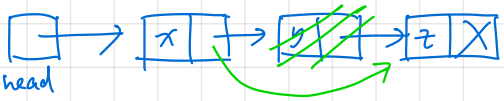
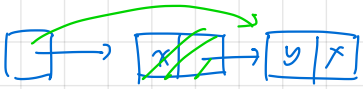
↳ edge cases: update head when anything at index 0

remember that head is a reference to the first node, need to update

special treatment for head @ index 0 needed





	method	how? time & space complexity?
insertion	void addAtIndex (T item, int i) void addFront (T item) void addBack (T item)	- create new node containing item, point preceding to itself, point itself to next, increment size. if inserting - adding anywhere else requires traversal, O(n)  -insertion at index = 0, use head as preceding → O(1) 
deletion	T removeAtIndex (int i) T removeFront () T removeBack ()	- traverse to preceding node, unlink by pointing next to node after, decrement number of nodes → O(n)  - insertion at index = 0, use head as preceding → O(1) 
retrieval	T getItem (int i) T getFirst () T getLast ()	- in general, requires traversal. O(n) - if just at front, then O(1), since direct

Tailed linked list

↳ how it works: like a regular LL, but include tail attribute that points to tail node.

↳ need to update head and tail when first, last & only items modified

↳ 3 states: empty, 1 item, n item

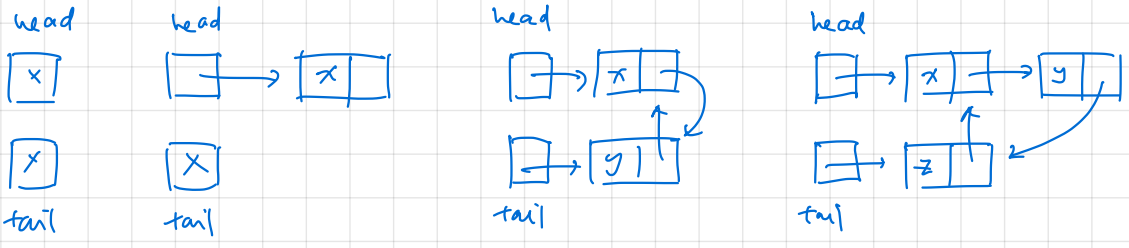
	method	how? time & space complexity?
insertion	void addAtIndex (T item, int i) void addFront (T item) void addBack (T item)	- if add at index 0, add after head, update head → O(1) ↳ if empty, update head ↳ if 1 item, move head to tail, point new node to tail & update head - if add at index = size, add after tail, update tail → O(1) - if in middle, traverse & update → O(n)
deletion	T removeAtIndex (int i) T removeFront () T removeBack ()	- if remove at index 0, update head to next ↳ if only 2 items, tail → null, head → tail → O(1) - if remove at index = size, update tail to preceding → O(1) - if middle, traverse and remove → O(n)
retrieval	T getItem (int i) T getFirst () T getLast ()	- get head, get tail → O(1) - traverse & get → O(n)

Circular linked list

↳ how it works: like tailed link list, except tail node also points to head

↳ states: empty, 1 item, 2 items, n items

↳ to note: modifications to head, tail, only item } same as TLL, just added step to point tail node to head every time head or tail changes



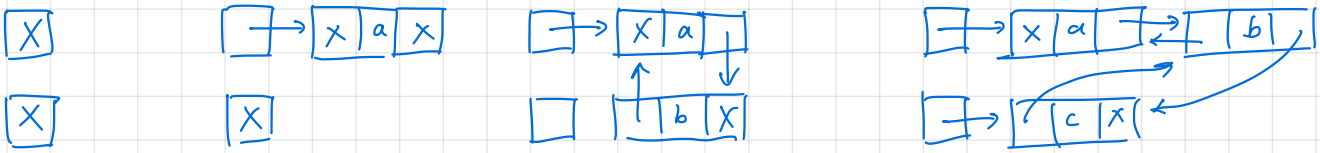
Doubly linked list

↳ how it works: every node is doubly linked & points both backward & forward

↳ generally only useful when it comes to tailed link list, since it allows for quicker reaches to middle, depending on whether index closer to head or tail

↳ states: empty, 1 item, 2 items, n items

↳ to note: modifications to head, tail, only item. Head preceding null, tail next null. always need to update after and preceding because now two-way link



② Array list vs. LinkedList

1) comparing complexities

	Method	Array_list	LL	TLL
insertion	void addAtIndex (T item, int i)	O(n)	O(n)	O(n)
	void addFront (T item)	O(n)	O(1)	O(1)
	void addBack (T item)	O(1)	O(n)	O(1)
deletion	T removeAtIndex (int i)	O(n)	O(n)	O(n)
	T removeFront ()	O(n)	O(1)	O(1)
	T removeBack ()	O(1)	O(n)	O(1)
retrieval	T getItem (int i)	O(n)	O(n)	O(n)
	T getFirst ()	O(1)	O(1)	O(1)
	T getLast ()	O(1)	O(n)	O(1)

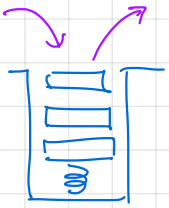
2) use cases

↳ only front back → TLL O(1)

↳ keep changing particular index. reference to node of LL O(1)

↳ a lot of accesses → array since O(1)

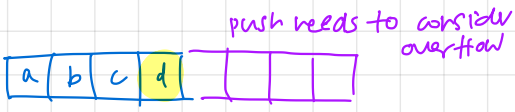
Stacks & Queues



Stack : LIFO

↳ major operations : push(), pop(), peek(), isEmpty()

Array implementation



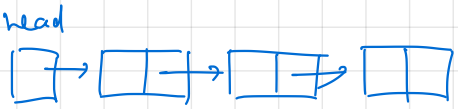
↔ use moving index to point to top element

when empty, index = -1

Size

method	complexity
push()	- O(1) normal, O(n) when need to handle overflow - O(1) amortized
pop() peek() isEmpty()	O(1)

LinkedList implementation



constant modification @ head

↳ non-contiguous memory, no need to handle overflow in push

method	complexity
push() pop() peek() isEmpty()	O(1)

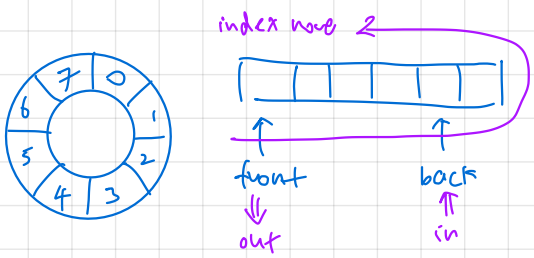
Queue

1) ADT : collection of data accessed in a FIFO manner

↳ major operations: queue(), dequeue(), peek(), isEmpty()

Array implementation

↳ circular array needed. queue() also needs to handle overflow.



↳ to update front & back upon queue & dequeue :

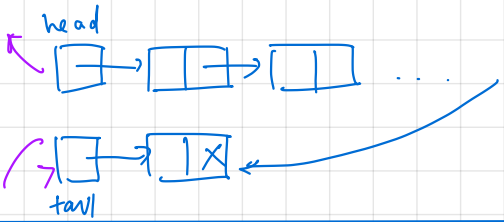
front = (front + 1) % maxSize

back = (back + 1) % maxSize

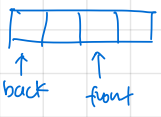
⇒ modulo uses equ. relation

TLL implementation

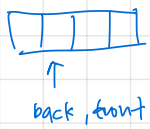
↳ trivial. AddBack, RemoveFront methods.



To resolve ambiguity of full vs. empty :



full



empty

1) use size attribute

2) leave empty space between front & back. if next back = front, full if front = back, empty

Map ADT & hashing

① Hash functions

1) good hash functions

1. fast to compute
2. scatters keys evenly throughout hashtable
3. Always finds an empty slot, deterministic

multiplication method

↳ multiply k by some constant $A \in (0,1)$ extract the fractional part & multiply by m.

↳ Similar to uniform hashing, but using some scale factor to correct for skew of key distribution.

$$h(k) = \lfloor m(k \cdot A - \lfloor kA \rfloor) \rfloor$$

$$A = \frac{\sqrt{5}-1}{2} \text{ is good (golden ratio)}$$

2) general hash functions

Uniform hash functions

↳ if k integers are uniformly distributed over 0 to X-1, we could map the keys to a hash table of size m ($m < X$)

$$k \in [0, X)$$

$$h(k) = \left\lfloor \frac{k \cdot m}{X} \right\rfloor$$

k: key; m: table size; X: max key value

↳ intuition: mapping key as its closest fraction of X to that in m

modulo operator

mapping values cyclically wrt. a "radix base"

$$h(k) = k \,\%\, m$$

m: table size

hashing strings

```
sum = 0
for each char in s
    sum += sum x 31 + char.toAscII()
```

⇒ Java String Hash function

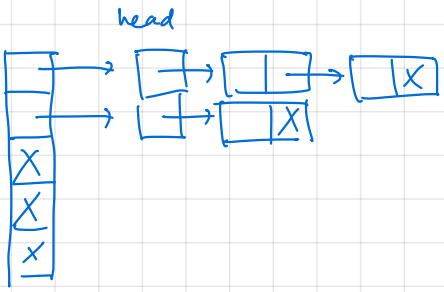
② Collision resolution

separate chaining

↳ every point in hashtable is a linked list

↳ collision, just add to head of linked list

↳ cannot benefit from caching



method	complexity
find (key) delete (key) insert (key, data)	- on average, each LL will be of length $\frac{n}{m} = \alpha$. (load factor) - insert into LL is O(1) - find & delete require traversal. $O(\alpha)$

↳ if we bound α by changing table size every time $\alpha > \alpha_0$, we are essentially getting $O(\alpha) = O(1)$

⇒ double table size to closest prime every time $\alpha > \alpha_0$

↳ amortized cost is O(1)

this method has poor space complexity however

linear probing

h(key)
(h(key) + 1) % m
(h(key) + 2) % m

quadratic probing

h(k)
(h(k) + 1) % m
(h(k) + 4) % m
(h(k) + 9) % m
:

theorem of quadratic probing

if $\alpha < 0.5$ and m is prime, then we can always find an empty slot. Else, may end up cycling & infinitely colliding

⇒ if $\alpha \geq 0.5$, double table so that $\alpha < 0.5$

modified linear probing

d is coprime to m, d > 1

h(k)
(h(k) + 1 × d) % m
(h(k) + 2 × d) % m
:

primary & secondary clustering

- primary ↓ : around same area, collision
- secondary ↓ : take same steps

$\alpha = \frac{n}{m}$ average load

Double hashing tackles both primary & secondary clustering

↳ use hash function to determine offset step
↳ DOF provided by second hash function helps to reduce both clustering types
↳ secondary function cannot map to 0 value, else cyclically stuck at first hash.

$h_1(k)$
$(h_1(k) + 1 \times h_2(k)) \% m$
$(h_1(k) + 2 \times h_2(k)) \% m$

$h_1 = k \bmod m_1$
$h_2 = m_2 - k \bmod m_2$

m_1 & m_2 co-prime

$m_2 < m_1$ must

Analysis of open addressing

methods	complexity
find(key) delete(key) insert(key, value)	↳ For unsuccessful find / delete & successful insert (all need to hit an empty slot): avg. no of probes = $\frac{1}{1-\alpha}$ ↳ For successful find & delete, need to hit filled avg. no. of probes = $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$

↳ if α is bounded by table doubling, then similarly O(1) amortized cost

Bloom Filter

N : no. of keys to be inserted

S : size of bit array

m : no. of hash functions

P : probability of false positive

$\Rightarrow$ Given N, P : $S = \left\lceil - \frac{N \cdot \ln P}{(\ln 2)^2} \right\rceil$

$m = \text{round}\left(\frac{S}{N} \cdot \ln 2\right)$

compression ratio

say, N keys, each 32 bits = 4 bytes. $B = 4$

$\Rightarrow$ Need $N \times B$ bytes. if in table

Bit array size = m, given P.

ratio = $\frac{m}{N \times B}$

in general, no. of bits of storage per key = $0.26 \ln \frac{1}{P}$ for some FP probability P

notice bits per key independent of size of key. Larger original size of key, better compression ratio

- Pros
  - Useful when number of keys too large for classical hash table to fit into RAM
  - Useful when we can accept a small amount of errors (false positives) during retrieval
  - Use M bits to represent a key regardless of the size of the key (especially useful when key is a long string)
- Cons
  - Larger overhead to insert/retrieve a key since need to compute M hash functions instead of 1 hash function
  - Not useful for problems where no error is tolerated
  - Not cache-friendly since the M bits for a key may not be close together thus cannot fit into cache

Priority queue ADT & heaps

① Priority queue ADT

↳ queue() → insert
↳ extract() → return max or min (highest/lowest priority)

---

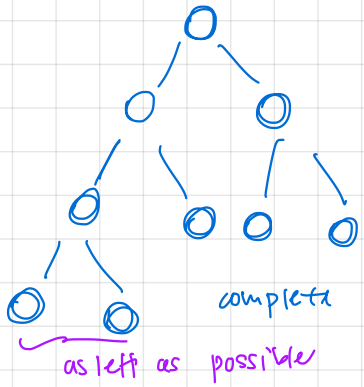
② complete binary trees

definition binary tree in which every level, except possibly the last is completely filled, and all nodes are as far left as possible

↳ if every level is completely filled, then it is a **perfect** binary tree

height of a complete binary tree

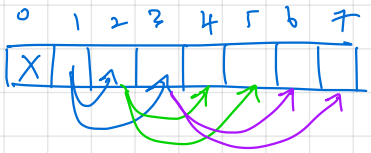
h = number of levels − 1
= max edges from root to deepest leaf
= $\lfloor \log n \rfloor = O(\log n)$



representing complete binary trees in arrays

↳ we use a 1-indexed array (for simplicity).

parent(i) = $\lfloor i/2 \rfloor$
left(i) = $2i$, no left when $2i >$ size
right(i) = $2i+1$, no right when $2i+1 >$ size



notice this is applicable for any radix base. r.x always gives exactly enough space for r children. In this case, 0 (left) and +1 (adjacent)

---

③ Binary heaps

binary heap property parent ≥ node. $A[\text{parent}(i)] \geq A[i]$

↳ equivalently, if have child (left/right), $A[\text{parent}] \geq A[\text{child}]$

↳ we use a complete binary tree to maintain this property

insert idea: insert at end to maintain "left most" (& balanced nature), then bubble up to correct position

```
insert (x)
    heapsize += 1
    A[heapsize] = x        // add at end
    bubble up (heapsize)
```

bubble up

```
bubble up (i)                          // while not root
    while i > 1 &&  A[i] > A[parent(i)]   // and heap property violated
        swap (i, parent(i), A)
        i = parent(i)
```

↳ **complexity** at most h swaps (up the tree to root) ⇒ O(log h)

---

update

1. construct an array / hashtable storing node to its value (value): key → idx in array
2. update. Do shift down & then shift up. For every swap, update idx. in hash table.

```
for i = 0 to heapsize
    key = arr[i]
    h.put (key, i)        // hash key → idx

update (oldkey, newkey)
    i = h.get (oldkey)    // update idx
    h.remove (oldkey)
    h.put (newkey, i)
    arr[i] = newkey       // update value

modified_shiftup (i)
modified_shiftdown (i)
```

} whenever swap, update h (track the modification)

complexity O(n) to preprocess
O(log n) to bubble up & down

```
modified_bubble up (i)
    while i > 1 &&  A[i] > A[parent(i)]
        swap (i, parent(i), A)
        h.put (A[parent(i)], parent(i))
        h.put (A[i] = i)
        i = parent(i)
```

complexity O(n) preprocessing, O(log n) shifting.

extract max

idea: take the root, take the last element and put at root, bubble it down to correct position

```
extractMax()
    max = A[1]  // root
    swap(A, heapsize, 1)  // swap last w first
    heapsize -= 1
    bubbleDown(1)  // bubble new root down

    return max
```

complexity

at most h swaps. so runs in O(log n) time.

bubble down

idea: among any three nodes, find the largest to be the parent

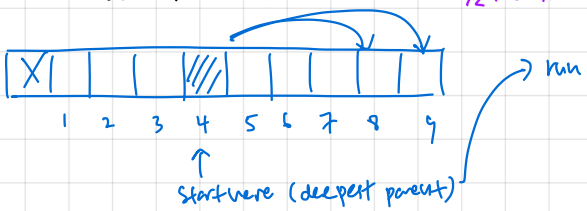
```
shiftDown(i)
    while i <= heapsize          (still valid & not at bottom)
        max = i
        if left(i) <= heapsize   (have left)
            if A[left(i)] > A[max]
                max = left(i)
        if right(i) <= heapsize  (have right)
            if A[right(i)] > A[max]
                max = right(i)
        (see if either child larger)
        if max != i
            swap(max, i)
            i = max              (keep going down)
        else
            break
```

---

build heap

idea: first, convert 0-indexed array to 1-indexed. then, for every element, shiftDown(i)

```
createHeap(arr)
    heapsize = arr.size
    for i = 1 to heapsize
        arr[i] = arr[i-1]  // 1 indexed      } copy O(n)
    for i = parent(heapsize) down to 1       (start from deepest parent)
        bubbleDown(i)                        (n/2 iterations)
```



intuition: starting from the deepest parent, (i.e. everything preceding now has a child), run bubble down, and we do so for every parent.

complexity

↳ At a given height h, there are $\lceil \frac{N}{2^{h+1}} \rceil$ nodes. bubble down runs in O(h) time. so at a given height, we run $\lceil \frac{N}{2^{h+1}} \rceil O(h)$ computations.

$$\sum_{h=1}^{\lfloor \log N \rfloor} \lceil \frac{N}{2^{h+1}} \rceil O(h) = \sum_{h=1}^{\lfloor \log N \rfloor} \lceil \frac{N}{2^{h+1}} \rceil \, c \cdot h \quad (c \text{ constants})$$

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

also: $\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$, in this case, $x = \frac{1}{2}$

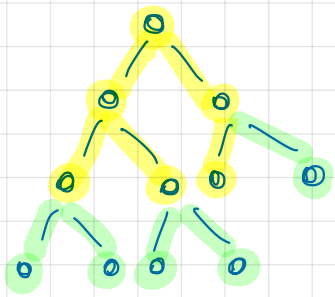
$$= O(2n)$$

$$= O(n)$$

filtering application : list all elements $\geq x$

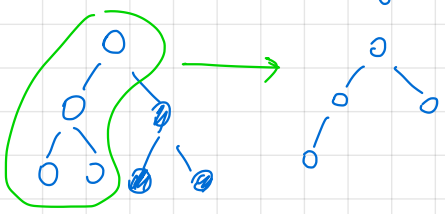
idea : heap property means that as we go down, things get smaller. so we can simply traverse the tree. so if a node $\geq x$, we print and continue going down. This will run in $O(k)$ time rather than $O(n)$.

```
findBelow ( node, x)
    if node.key ?/ x     // continue
        output node.key  // print or do something
        find below (node.left, x)
        find below (node.right, x)
    else
        // terminate
```



Delete subtree

↳ to delete subtree rooted at a given node, traverse array until root is found. Store all to be kept into a new array $\Rightarrow O(n)$. Build heap from new array. $\Rightarrow O(n)$



---

④ Heap sort

↳ create heap, extract max or min & store into new array
↳ but note that not cache friendly due to non-linear storage in heap

heapsort

complexity $O(n \log n)$

```
heapsort (arr)
    heap = build heap (arr)        → O(n)
    for i = 0 to heapsize
        arr[i] = heap.extract min()   } O( log(n!)) = O(n log n)
                                        heap gets smaller
    return arr
```

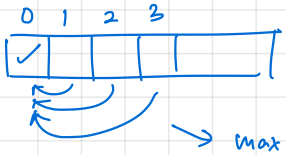
---

sorted window from unsorted array of size k from some $\underbrace{a \text{ to } b}_{\text{rank}}$

```
val_a = quickselect (arr, a)   // get values at ranks   } O(n)
val_b = quick select (arr, b)

for i in arr
    if val_a ≤ i ≤ val_b   // filter by value, O(n)
        out.append (i)

out.sort()   // in order,  O(k log m)
```

k-sorted array | max/min window

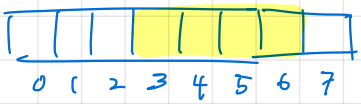
- suppose we have a k-sorted array — each value differs from its correct position by no more than k positions
- ⇒ so an element that should be placed at A[i] is the minimum in A[i, ..., i+k]
- ⇒ we can use a sliding window to look at a given k elements and extract each minimum and put it at A[i]



i=0, k=3

→ max k positions away ⇒ window of k+1 size

```
pq = min heap()

for i=0 to k
    pq.add(A[i])  // heapify first k+1 elements

for i= 0 to n-(k+1)-1   // move sliding window until end
    min = pq.extract min()  // extract min of k+1 window
    out[i] = min
    pq.add(i+k+1)  // add next

for i= n-k-1 to n-1
    min = pq.extract min()
    out[i] = min
```



---

dynamic max/min sum of top k in stack

The Great Overlord of Arithmetic has a challenge for you! He gives you a stack of n integers, and allows you to repeatedly perform the following operation:

- From any of the top k integers in the stack, you may remove one of them and add it to the Fundamental Pool of Arithmetic.

The Great Overlord of Arithmetic wants you to calculate the maximum total value of numbers that can possibly be placed in the Fundamental Pool of Arithmetic (starting from 0). Give an efficient algorithm to do this, and state the time complexity.

For example, for $k = 2$, where integers in the stack are $[2, -10, 2, -6, 5]$, the output of your algorithm should be 4. Another example for $k = 5$, where integers in the stack are $[-1, -1, -1, -1, -1, 10]$, the output of your algorithm should be 9.

Solution: Since we can only look at the top k elements in the stack, and we want to maximise the total value of elements chosen, we make use of a maximum heap of size k. We need to maintain two values, the current / running sum, and the historical highest sum achieved.

We will remove the largest element in the heap, followed by inserting the next element in the stack. This simulates the top k elements in the stack at any one point in time. When removing the largest element, we will also add that value to the current sum, and update the historical highest sum when necessary. Note that since it is possible that all the values in the heap of size k are negative, the current sum can be reduced. This is why we need to keep track of the historical highest sum. We repeat this step for $n-k$ times until the last element is added to the heap.

Note that since there are k elements remaining that could have positive values, we also need to check the remaining elements in the heap and update the sums as necessary.

lgorithm 2 Solution to Problem 5

```
: Let A be the stack of integers
: Initialise maximum heap D
: for i = 1 to k do
4:     Insert A.pop() into D
5: end for
6: max_sum = 0
7: current_sum = 0
8: for i = k + 1 to n do
9:     current_sum = current_sum + D.extractMax()
10:    Insert A.pop() into D
11:    if current_sum > max_sum then
12:        max_sum = current_sum
13:    end if
14: end for
15: while D is not empty and D.getMax() > 0 do
16:    current_sum = current_sum + D.extractMax()
17:    if current_sum > max_sum then
18:        max_sum = current_sum
19:    end if
20: end while
21: return max_sum
```

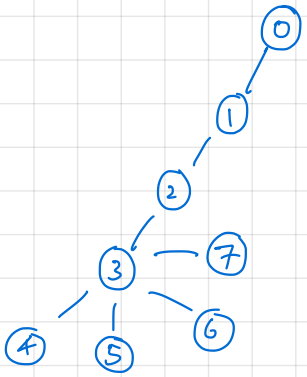
Disjoint sets

① union-find disjoint sets

1) data structure
   ↳ we use a parent array to store the parent of each item

> p[i] stores parent of i
> if p[i] == i, then i is a root

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



---

2) methods

findset idea: we can find the representative item of an item by recursively visiting parent until we find the node. And at each recursive call, we set the parent to the representative node to speed up future calls to O(1) time (path compression)

```
findset(i)
    if p[i] == i
        return i

    else
        p[i] = findset(p[i]) // path compression + recursive search
        return p[i]
```

---

isSameSet idea: checks if same set, but also does path compression

```
isSameSet(i,j)
    return findset(i) == findset(j)
```

---

unionset

↳ union by rank heuristic: rank[i] → max height of tree rooted at i
⇒ we don't maintain ranks of non-root items

```
unionset(i,j)
    x = findset(i)
    y = findset(j)

    if rank[x] > rank[y]
        p[y] = x // put y under x

    else
        p[x] = y // put x under y
        if rank[x] == rank[y]
            rank[y] += 1 // update rank of root w/o path compression
```

then when union, we put the shorter tree under the root of the taller one. then height no change. Else if same height, we arbitrarily choose one. Height grows by 1.

so tree heights only grow when unioning if same height ⇒ balanced tree

constructor

```
UFDS
    int [] p;
    int [] rank;

    ? [] A;  // group attribute

    UFDS(n)
        for i = 0 to n
            p[i] = i
            rank[i] = 0
            A[i] = ?
```

} O(n) initialization

---

② modifications to UFDS

modified unionset idea: union by rank helps with complexity of finding representative node since it creates a balanced tree. But it means relative order of unions are lost. Say, put A < B ⇒ but if A taller, A will be above B

⇒ need an extra data structure to store a representative attribute

```
unionset(i,j)
    x = findset(i)
    y = findset(j)

    if rank[x] > rank[y]
        p[y] = x   // put y under x
        // statement about A[x] (new parent)
    else
        p[x] = y   // put x under y
        // statement about A[y] (new parent)
        if rank[x] == rank[y]
            rank[y] += 1   // update rank of root w/o path compression
```

suppose some r/s between i & j
- i always under j
  A[new parent] = A[y]
- choose max(A[i], A[j])
  A[new parent] = max(,)

and so on ...
A[new parent] = f(A[x], A[y])

so tree heights only grow when unioning if same height ⇒ balanced tree

③ UFDS for contiguous sequence

↳ suppose you are given a sequence of length l. we want to know the longest consecutive sequence from some starting point a. sequence can have duplicates

intuition for each item, add it to an array keeping track of whether it has been seen before.

then we try to union it with its left & its right. when we union, we keep track of the largest in its consecutive sequence.

⇒ union only w left & right ensures consecutive

⇒ keep track of max allows us to see where sequence ends

UFDS — modifications to union

logic

```
unionset (x, y)
    x = findset (x)
    y = findset (y)
    if x == y
        return // same set, do nothing

    else if rank[x] > rank[y]
        p[y] = p[x] // union by rank
        max[x] = max ( max[x], max[y] )

    else
        p[x] = p[y] // x under y
        if rank[x] == rank[y]
            rank[y] += 1
        max[y] = max (max[x], max[y]) // maintain extreme right of sequence
```

```
for i = 0 to l-1
    idx = A[i]
    if seen[idx]   // duplicate
        continue
    else
        seen[idx] = true
        if seen[idx-1]
            unionset(idx, idx-1)
        if seen[idx+1]
            unionset(idx, idx+1)
```

⇒ print (findFurthest(a))

```
findFurthest(idx)  // searches semantic info
    idx = findset(idx)
    return max[idx] // semantic info in max array
```

---

④ UFDS w membership

↳ suppose we have <ID, set> pairs.

- merge sets, findsets.
- use hashtable to store ID → set mappings

to see if ID in valid set: isSameset(h.get(ID, set))

$O(\alpha(n))$ $O(1)$

ordered map & search Trees

① Binary search trees

size

1. subtree rooted at x has size = x.left.size + 1 + x.right.size
2. single node w/ no children has size = 1

height

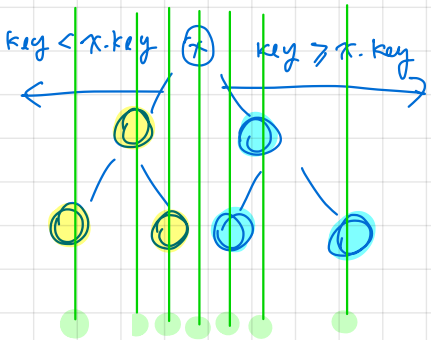
1. subtree rooted at x has $h = \max(x.left.height, x.right.height) + 1$
2. single node w/ no children has $h = 0$

BST property

remember that it is in subtree, not necessarily immediate left/right children

↳ for every vertex x, y:
- $y.key < x.key$ if y is in left subtree of x
- $y.key \geq x.key$ if y is in right subtree of y

$\Rightarrow$ implies $x.left \leq x \leq x.right$ (not other way)



h = no. of edges to leaf from root

we can visualize BST property by projecting tree down

---

② Balanced BSTs : AVL Trees

height-balanced tree property

1. a vertex is height-balanced iff $|x.left.height - x.right.height| \leq 1$
2. a BST is height-balanced iff every vertex in the tree is height balanced

minimum number of vertices in a height-balanced tree of height h

$N_h = 1 + N_{h-1} + N_{h-2}$

$N_0 = 1$

$N_1 = 2$

idea: root node + min for left + min. for right

one subtree shorter than the other to minimize no. of nodes, diff of 1 to maintain balance

min. no. of vertices in AVL w/ h

$N(0) = 1$	$N(4) = 12$	$N(8) = 88$
$N(1) = 2$	$N(5) = 20$	$N(9) = 143$
$N(2) = 4$	$N(6) = 33$	$N(10) = 232$
$N(3) = 7$	$N(7) = 54$	

③ methods

search idea: go left or right depending on key value

$O(h) = O(\log n)$ on average, $O(n)$ worst

```
search ( node , target)
    if node == null  // not found
        return null
    else if node.key == target  // hit
        return node

    else if node.key > target  // go left
        return search (node.left, target)

    else if node.key < target  // go right
        return search
```

minimum / maximum idea: go left (min) or right (max) all the way

$\Rightarrow O(h) = O(\log n)$ avg.
$= O(n)$ worst

```
minimum ( node)
    if node.left == null
        return node
    else return minimum (node.left)

maximum (node)
    if node.right == null
        return node
    else return maximum (node.right)
```

select idea: select k ranked by implicitly counting how many passed (how many more to go)

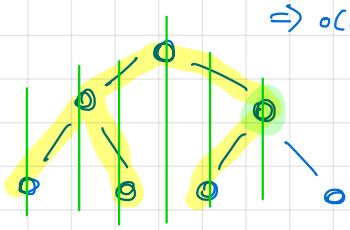
$\Rightarrow O(h) = O(\log n)$ avg.
$= O(n)$ worst

```
select ( node, rank)
    position = node.left.size + 1
    if position == rank
        return node
    else if position > rank
        return select (node.left, rank)
    else
        return select (node.right, rank - position)
```

rank → position in sorted order

idea: use subtree sizes to keep track of how many before in sorted order

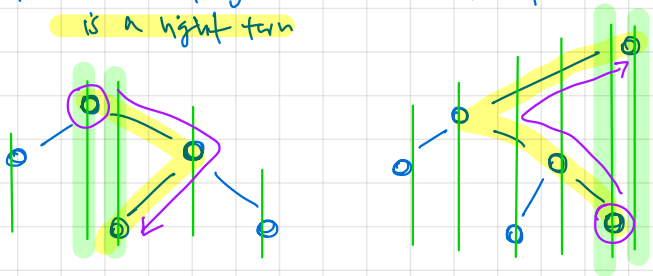
$\Rightarrow O(h) = O(\log n)$ avg.
$= O(n)$ worst



```
rank (node, target)
    if node.key == target  // hit
        return node.left.size + 1
    elseif node.key > target  // go left
        return rank (node.left, target)
    else // go right
        return node.left.size + 1
            + rank (node.right, target)
```

successor → next largest element in sorted order

idea: leftmost of right subtree or first parent that is a right turn

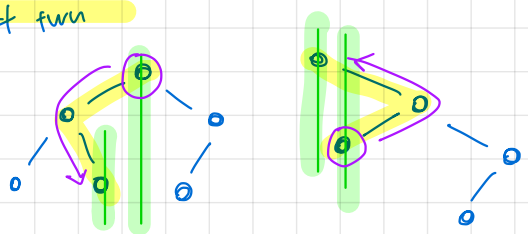


```
successor (node)
    if node.right != null
        return node.right.minimum()      } go right & down
    else
        parent = node.parent
        while parent != null             } go up until right
            if parent.left == node
                return parent
            else
                node = parent
                parent = node.parent
```

consecutive successor calls for node implementation run in $O(h)$ time

predecessor → next smallest element in sorted order

idea: rightmost of left subtree or first parent from a left turn



```
predecessor (node)
    if node.left != null
        return node.left.maximum()       } go left & down
    else
        parent = node.parent
        while parent != null             } go up until left
            if parent.right == node
                return parent
            else
                node = parent
                parent = node.parent
```

tree traversal

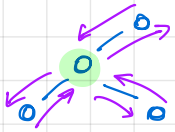
↳ in order: left → centre → right
↳ pre order: centre → left → right
↳ post order: left → right → centre

skeleton

```
morder (node)
    if node.left != null
        inorder (node.left)

    // do stuff at node

    if node.right != null
        in order (node.right)
```

complexity

⇒ each node is visited at most three times. once from parent, once from left child coming back & once from right child coming back. ⇒ $O(3n) = O(n)$

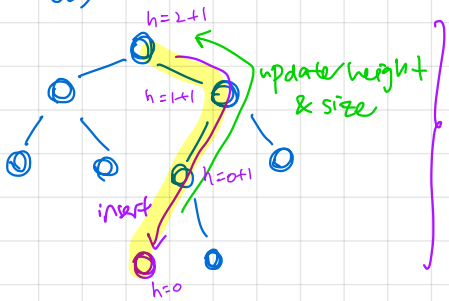


---

5) modification methods: must preserve BST properties & metadata (eg. size)

insert idea: find the correct position. if null, put there. update height & size as you go up

⇒ $O(h) = O(\log n)$
$= O(n)$ worst



observe that only vertices along insertion path will have their height & size changed by the insertion

```
insert (node, x)
    if node == null    // hit
        return x
    else if node.key > x.key // go left
        node.left = insert (node.left, x)

    else // go right
        node.right = insert (node.right, x)

    // update height & size
    node.height = max (node.left.height, node.right.height) +1
    node.size = node.left.size + node.right.size +1
    return node
```

$O(h)$

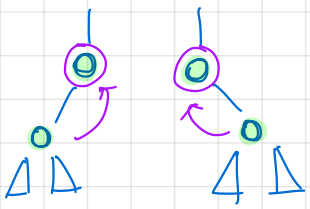
$O(1)$

will always update since stacking from bottom

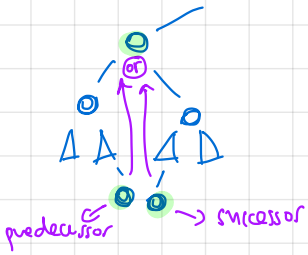
delete idea: find the node. if terminal, just delete. if single parent, replace w child. if two children, replace w predecessor or successor

⇒ $O(h) = O(\log n)$
$= O(n)$ worst

single parent



two children




```
delete (node, x)
    if node.key > x  // go left
        node.left = delete (node.left, x)
    else if node.key < x  // go right
        node.right = delete (node.right, x)
    else  // hit
        if node.left == null && node.right == null
            node = null
        else if node.left == null && node.right != null
            node = node.right
        else if node.left != null && node.right == null
            node = node.left
        else
            s = successor (node)
            n.data = s.data
            n.key = s.key
            n.right = delete (n.right, s)
                ↳ go right, since successor
    // update height & size
    return node
```

$O(h)$

$O(1)$

avoid duplicate

④ balancing with AVL trees

balance factor measures the height difference between the left & right subtrees

$$bf(x) = x.left.height - x.right.height$$

---

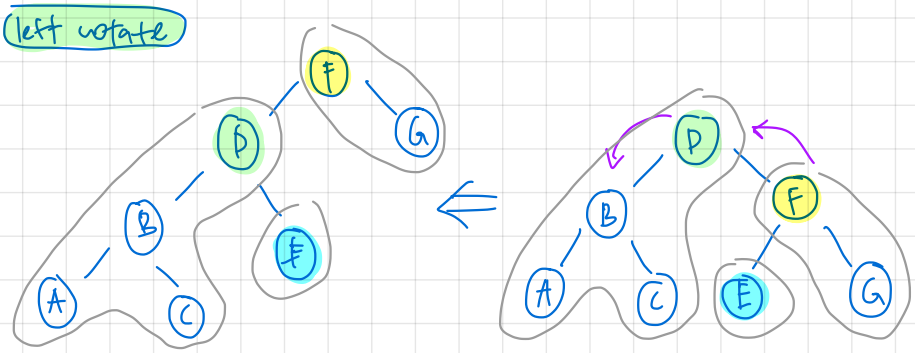
rotations

idea: there are several different ways we can represent a tree so long as the BST property holds, they are valid configurations of the BST. we can use rotations to move between different configurations.

⇒ left/right rotate:

we change the root of a subtree with its left / right child and maintain the BST property

left rotate

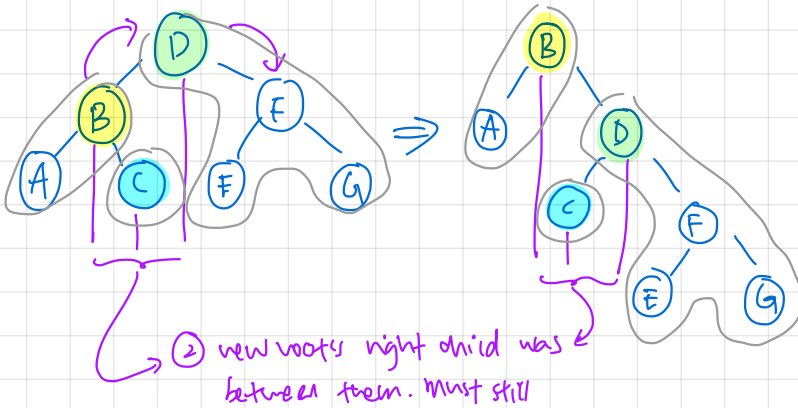



```
leftRotate(x)   // assumes x has right child
    y = x.right
    // assign parents
    y.parent = x.parent
    x.parent = y
    // switch sides
    x.right = y.left
    x.right.parent = x
    // set new root
    y.left = x
    // update x height
    // update x size
    // update y height
    // update y size
    return y
```

update lower stuff first

right rotate

① previous root becomes right child of new root.



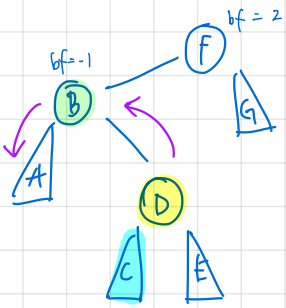
② new root's right child was between them. must still be between. Thus becomes left child of old root.

```
rightRotate(x)   // assumes has left
    y = x.left
    // assign parents
    y.parent = x.parent
    x.parent = y
    // switch child side
    x.left = y.right
    x.left.parent = x
    // set new root
    y.right = x
    // update x height
    // update x size
    // update y height
    // update y size
    return y
```

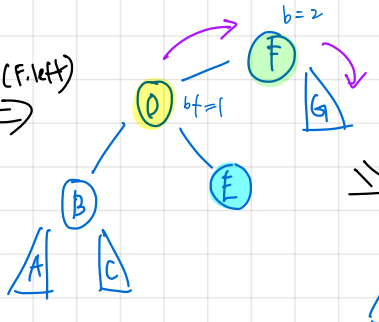
update lower stuff first

four cases of imbalance idea: during insertion & deletion, a tree will become imbalanced when $bf(x) \geq 2$. since they change in increments of 1, when $|bf(x)|$ hit 2, we can use rotations to rebalance the tree

left-right

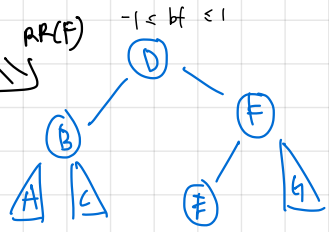


LR(F.left)

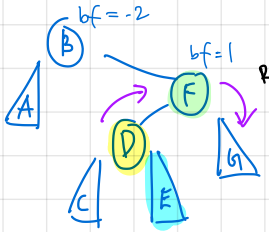


left-left

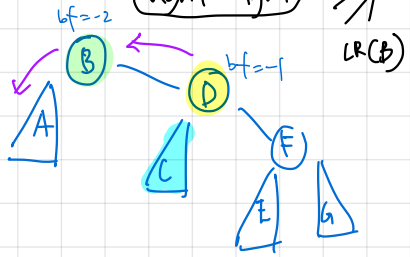
RR(F)



right-left



RR(B.right)



right-right

LR(B)

```
rebalance(x)
    if bf(x) == 2
        if 0 ≤ bf(x.left) ≤ 1 // LL
            right Rotate(x)
        else if bf(x.left) = -1 // LR
            left Rotate(x.left)
            right Rotate(x)

    else if bf(x) == -2
        if -1 ≤ bf(x.right) ≤ 0 // RR
            left Rotate(x)
        else if bf(x.right) = 1 // RL
            right Rotate(x.right)
            left Rotate(x)

    return x
```

⑤ balanced insertion & deletion

insertion idea: insert as in normal BST, but rebalance as you walk up

```
insert(node, x)
    if node == null  // hit
        return x
    else if node.key ≥ x.key // go left
        node.left = insert(node.left, x)

    else // go right
        node.right = insert(node.right, x)

    // update height & size
    node.height = max(node.left.height, node.right.height) + 1
    node.size = node.left.size + node.right.size + 1

    // rebalance
    node = rebalance(node)

    return node
```

O(h)

O(1)

will always update since starting from bottom

O(1)

deletion idea: delete as in normal BST, update heights and size, rebalance every vertex

```
delete(node, x)
  if node.key > x   // go left
    node.left = delete(node.left, x)
  else if node.key < x   // go right
    node.right = delete(node.right, x)
  else   // hit
    if node.left == null && node.right == null
      node = null
    else if node.left == null && node.right != null
      node = node.right
    else if node.left != null && node.right == null
      node = node.left
    else
      s = successor(node)
      node.data = s.data
      node.key = s.key
      node.right = delete(n.right, s)
        (-> go right, since successor
```

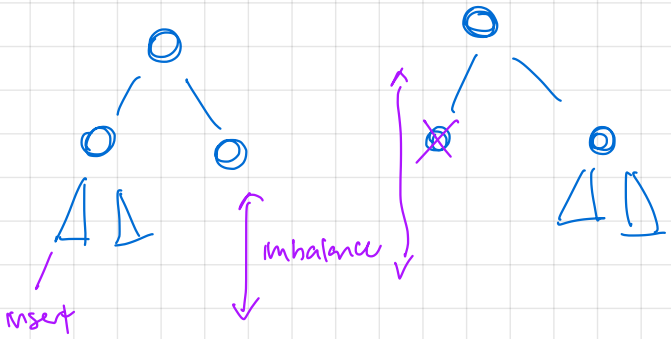
O(h) O(1) swap avoid duplicate O(h)

```
  // update height & size
  node.height = max(node.left.height, node.right.height) + 1
  node.size = node.left.size + node.right.size + 1
```

O(1) will always update since stacking from bottom

```
  // rebalance
  node = rebalance(node)   O(1)

  return node
```

Note: deletion or insertion can trigger O(h) balancing operations on the way up. It occurs when deletion on sparse, shorter subtree, or insertion on dense, taller subtree.



---

array to AVL preorder creation

sorted, initially 0, size-1

```
arrayToAVL(array, left, right)   // O(n) time
  if left ≤ right   // valid range
    mid = (left + right)/2
    v = new Vertex(array[mid]) // put mid as parent
    v.left = arrayToAVL(array, left, mid-1)
    v.right = arrayToAVL(array, mid+1, right)
    return v
  else
    return null
```

isAVLTree

```
boolean isAVL(T)
{
 if (T != null)
   if (isAVL(T.left) && isAVL(T.right))
     if (-2 < T.left - T.right < 2)
       T.height = max(T.left.height, T.right.height)+1;
       return true
     else
       return false
 else
    T.height = -1 // empty tree
    return true
}
```

This is basically post-order processing of the vertices to correctly calculate height and bf.

⑥ Splay Trees

↗ not to balance tree, but amortize access cost

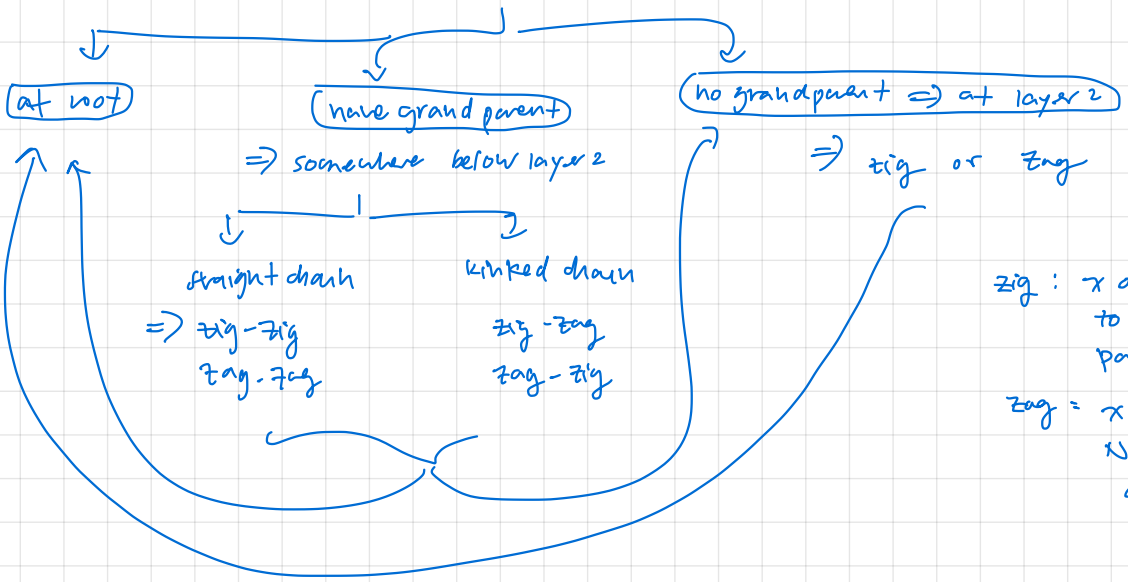
1) heuristic balancing: the most frequently accessed key will most likely be accessed again, so should be placed at the top of the tree, making future accesses $O(1)$ time.

⇒ search is modified so whenever x is found, the node is shifted to the root using rotations (splay steps)

⇒ insert & delete is their BST counter part w splaying after

---

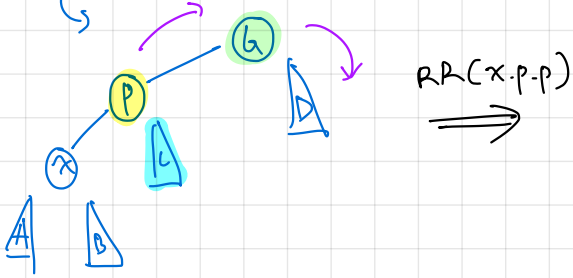
splay step idea: if we want to bubble a node to the top, we need specific steps depending on the topology of the tree. We can break any scenario down



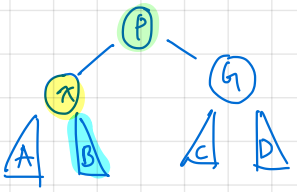
zig: x on parent left. Need to right rotate about parent

zag: x on parent right. Need to left rotate about parent

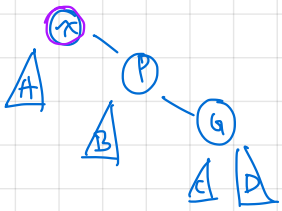
zig-zig RR(G) + R(P)



RR(x.p.p) ⇒

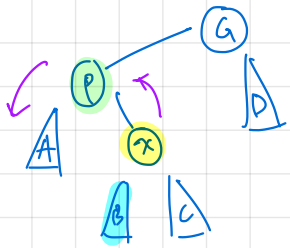


RR(x.p) ⇒

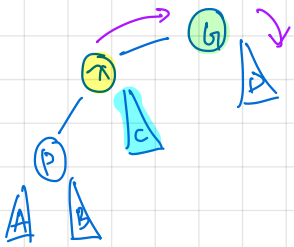


zag-zag LR(G) + LR(P)

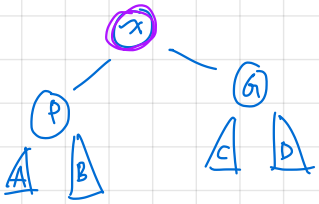
zig-zag LR(P) + RR(G)



LR(x.p) ⇒

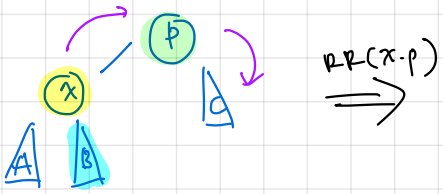


RR(x.p) ⇒

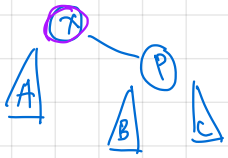


zag-zig RR(P) + LR(G)

Zig RR(P)



RR(x.p)



Zag LR(P)

splay(x) ⇒ gives O(logn) on average

```
while x.parent != null      // while x is not at root
    if x.parent.parent = null   // no grandparent
        if x = x.parent.left    // x is left child, zig
            right Rotate (x.parent)

        else if x = x.parent.right  // x is right child, zag
            left Rotate (x.parent)

    else if x == x.parent.left && x.parent == x.parent.parent.left  // zig zig
        rightRotate (x.parent.parent)
        rightRotate (x.parent)

    else if x == parent.right && x.parent == x.parent.parent.right  // zag zag
        left Rotate (x.parent.parent)
        left Rotate (x.parent)
    else if x == x.parent.right && x.parent == x.parent.parent.left  // zig zag
        leftRotate (x.parent)
        RightRotate (x.parent)
    else if x == x.parent.left && x.parent == x.parent.parent.right  // zag zig
        right Rotate (x.parent)
        left Rotate (x.parent)
```

3) operations

Search idea: search. hit, splay to top.

```
search (node, x)
    if node.key > x  // go left
        return search (node.left, x)

    else if node.key < x  // go right
        return search (node.right, x)
    else  // hit
        splay (node)
        return node
```

Insert idea: upon insert, repeated splay until at root

```
insert (node, x)
    if node.left == null && node.right == null
        if node.key >= x  // go left
            node.left = x
            x.parent = node
        else  // go right
            node.right = x
            x.parent = node
        splay (x)
        return x

    else if node.key > x  // go left
        return insert (node.left, x)

    else // go right
        return insert (node.right, x)
```

delete idea:
- if deleted is true only node, do nothing.
- if deleted has 0 or 1 child, splay x's parent
- if deleted has 2 children, splay parent of x's successor

```
delete(node, x)
  if node.key > x  // go left
    node.left = delete(node.left, x)
  else if node.key < x  // go right
    node.right = delete(node.right, x)
  else  // hit
    if node.left == null && node.right == null
      node = null
    else if node.left == null && node.right != null
      node = node.right
    else if node.left != null && node.right == null
      node = node.left
    else
      s = successor(node)
      node.data = s.data   } swap
      node.key = s.key     } avoid duplicate   } O(1)
      node.right = delete(n.right, s)   } O(h)
        (go right, since successor)

    splay(node)
    return node
```

O(h)

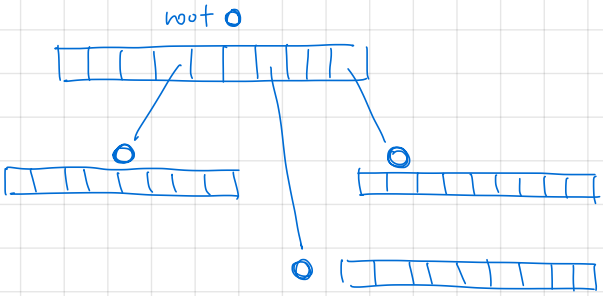
O(1)

---

⑦ Trie

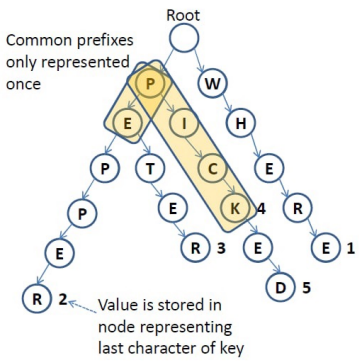
structure path from root to some node is a collection of characters
- common prefixes only stored once
- O(k) search & comparison

implementation
- use nodes, each w/ up to n children (n is number of characters in alphabet)
- value stored at node. so if terminate at node; since it took a unique path and number of steps to get there



Key/Value Pairs

WHERE 1
PEPPER 2
PETER 3
PICK 4
PICKED 5



search

Given a String key S to retrieve

1. Matching process
   i. Start from the root (level 0), check if it has a child node with character matching S[0], if it has move to child node and repeat the matching for S[1] and so on.
   ii. In general for a node at level M, the matching process check if the node has a child matching the character at index M of S
2. Terminating condition
   i. hit a node with no children node matching current character of the key ← return a miss
   ii. hit a node matching last character of key and there is a valid value for that node ← return the value (a hit)
   iii. hit a node matching last character of key and there is no valid value for that node ← return a miss

insert

- Perform a search for the key similar to retrieval operation until

1. Hit a node matching the last character of the key and there is a valid value for the node (existing key/value pair)
   ←update the value with the new value.
2. Hit a node matching the last character of the key and there is no valid value for the node
   ←store the value in this node.
3. We hit a node with no children matching the current character of the string
   ←start inserting the remaining characters as descendent nodes in a linked list like fashion and put the value in the last node.

delete

- Again perform a search for the key

1. Hit a node with no children matching current character of key ← do nothing
2. Hit a node matching the last character of the key and there is no valid value for the node ← do nothing
3. Hit a node matching the last character of the key and there is a valid value for the node ← remove the value
   i. If it has children nodes do nothing else (it is both a key and also the prefix for other existing keys)
   ii. If it has no children nodes then move back towards the root and start removing nodes. Stop when we hit a node that has at least 1 child (a prefix of some key(s))

---

(6) common patterns

common ancestors / descendents (a, b)

↳ traverse. If go different direction at a node, that node is lowest common ancestor.

↳ descendent, just flip tree and treat as if ancestor

---

greedy AVL deletion within range: suppose a ... b → want to delete in $O(k \log n)$ rather than $O(m \log n)$ ⇒ use successor

m values, k actually in BST

```
delete_range(a, b)
    node = search(a, root)   // O(log n) time
    if node == null
        insert(a)
        node = search(a, root)
    i = a
    while i ≤ b
        delete(i, root)   // O(log n)
        snode = successor(i)   // next, at most k times
        i = snode.val
```

contiguous blocks

↳ initialize a BST w all elements that stores non-hit
↳ every time hit, delete
↳ find successor / predecessor if inside — successor is modified to insert if not inside, call successor, then delete

eg. between 1 - 8
⇒ if no successor of 1 between, contigous

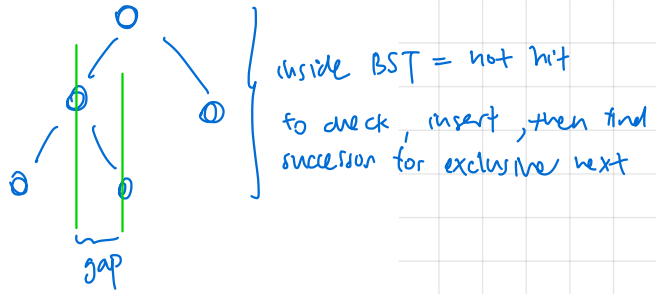
Solution: We can use a bBST to maintain the servers that are currently disabled. When a server is disabled, we insert it into the bBST, and when a server is enabled, we remove it from the bBST. To check if we are able to send a message from server i to server j, we need to check that i is not in the bBST, and either i has no successor in the bBST, or the successor of i in the bBST is greater than j.

Here, we note that the successor of x, or *successor*(x), refers to the node in the bBST containing the smallest key that is greater or equal to x. This definition of successor is similar to the `higherEntry` method in the Java `TreeMap`. A simple way to implement this is to first do a search to check for existence, then do an insertion and call the usual successor, and then doing a deletion. However, there is way to do this directly without any insertions or deletions by modifying the search operation, and is left as an exercise.

if not inside, insert

Algorithm 4 Solution to Problem 4

```
1: T ← bBST, initially empty
2: procedure ENABLE(i)
3:     T.delete(i)
4: end procedure
5: procedure DISABLE(i)
6:     T.insert(i)
7: end procedure
8: function SEND(i, j)
9:     if i is in T or T.successor(i) ≤ j then
10:        return false
11:    else
12:        return true
13:    end if
14: end function
```



Build BST from postorder

call T = RebuildTreePost(A) to rebuild the BST.

The algorithm guarantees that all the required vertices are created, since each item in A is considered (cur goes from |A|-1 to 0), thus there will be O(N) calls to recursiveBuild to create all he vertices (minus the root).
For each created vertex, recursiveBuild is called on both its left and right child. In the worst case, both left and right child are null, thus O(2N) calls are made.
In total recursiveBuild is called O(3N) = O(N) time.
Each call to recursiveBuild takes O(1) time (constant number of statements executed).
Thus total time is O(N).

```
T RebuildTreePost(A)
0. Let cur = |A|-1, cur is a global variable
1. V = create vertex for A[cur]
2. Let max = 1000,000,000, min = 0, r = A[cur]
3. cur -= 1
4. V.right = recursiveBuild(A,r+1,max) & link right child to V
5. V.left = recursiveBuild(A,min,r-1) & link left child to V
6. return V
```

```
T recursiveBuild(A,i,min,max) // [min,max] is the allowable range for A[cur]
1. if (cur < 0)  return null // processed all values in the sequence
2. r = A[cur]
3. if (min <= r <= max) // create the vertex only if A[cur] is within range
      a. V = create vertex for A[cur]
      b. cur -= 1

      c. V.right = recursiveBuild(A,r+1,max) & link right child to V
      d. V.left = recursiveBuild(A,min,r-1) & link left child to V
      e. return V
   else
      return null
```

① graphs

Vertex A node in the graph

Edge A connection between two vertices in a graph. Can be directed / undirected, weighted / unweighted.

(in/out)degree of a vertex number of (in/out) edges of a vertex

simple graphs a set of vertices where some pairs of vertices are connected by undirected edges

(complete (simple) graph) a simple graph where all vertices are connected with all other vertices. N vertices, ${}_nC_2 = \frac{n(n-1)}{2}$ edges

(sparse & dense graphs) spectrum. sparse, $|E| = O(V)$. Dense, $|E| = O(V^2)$

(simple) path sequence of vertices connected by edges. simple => no repeated vertices

(simple) directed path a simple path where all edges along the path are directed in the same direction

(simple) cycle path that starts & ends with the same vertex. simple => no repeated vertices apart from start & end

(simple) directed cycle simple cycle where all edges along the cycle are directed in the same direction

(path length / cost) in unweighted graph, cost = length = number of edges of path usually.
in weighted graph, usually sum of edge weights along the path.

tree connected graph where there is one unique path between any pair of vertices.
$|E| = |V| - 1$

② Graph Data structures

	Adjacency matrix	Adjacency list	Edge list
intuition	2D array. `A[i][j] = w`, if (i) —w→ (j)	2D array list. `A[i]` contains an array list of vertex i's neighbours and weight of edges to them.	Edge represented by triple. (from <x,y> or 1D, to <x,y> or 1D, weight) Store edges in Array List

implementation	`new A[V][V]`	`new A< A< To >` (weight, to)
		`for i in {V}`
		`A[i] = new A<Edge>`
		`A[i].add(..)`
			`new A< Edge >`

space complexity	$O(V^2)$	$O(V+E)$	$O(E)$
time complexity of enumerating neighbours for a given vertex	find row $O(1)$ find non-zero cells $O(V)$ $\Rightarrow O(V)$	find row $O(1)$ loop through list $\Rightarrow O(k)$	scan through all edges and find those w correct "from" $\Rightarrow O(E)$
counting number of edges	count non-zero cells $O(V^2)$	get size of all list $O(V) \times O(1) \Rightarrow O(V)$	size of list $\Rightarrow O(1)$
querying an edge from u ⟶ v	indexing. $O(1)$	go to `A[u]`. by convention, adj list sorted by neighbour vertex ID binary search to find $O(1) + O(\log k)$ $\Rightarrow O(\log k)$	scan through all edges $\Rightarrow O(E)$

Graph Data structures & algorithms

① Fundamental graph traversal algorithms ⇒ traversing all connected nodes exactly once

Breadth First search

flag to avoid cycling

↳ idea: using a visited[V] array to keep track of explored nodes and a queue to keep track of what node to explore next, visit from the source in an expanding-by-degree-away manner (ordered implicitly by node ID, since typically we store neighbor nodes in order of ID)

```
for all v in V
    visited[v] = false
    preceding[v] = -1
```
} initialize O(V)

```
q = {s}  // put source into queue

while !q.isempty()  // terminate when all nodes explored
    u = q.dequeue()
    { do stuff @ node }
    neighbors = A[i]  // adj list
    for all v in neighbors  // find neighbors to explore
        if visited[v] != true  // if not visited
            visited[v] = true  // mark as visited
            q.enqueue(v)  // enqueue for exploration
            preceding[v] = u  // record history
```
} O(V+E) if adjacency list since every node is enqueued once and all edges checked when finding neighbors

O(V) x O(V) if matrix = O(V²)

Depth First search

↳ idea: explore in a depth-first way; go all the way to end, backtrack, continue. We can use a stack to main order in backtracking. Recursion used as implicit stack.

```
DFS(u)
    visited[u] = true  // global visited
    { do stuff @ node }
    neighbors = A[u]  // explore neighbors
    for all v in neighbors
        if visited[v] != true  // if not visited
            preceding[v] = u  // set predecessor
            DFS(v)  // explore all the way
```
} notice logic of marking as visited/doing stuff is at node itself.

} if no unvisited neighbors, then notice it just backtracks by returning recursive call without making more calls

```
for all v in V
    visited[v] = false
    preceding[v] = -1
```
} initialisation, O(V)

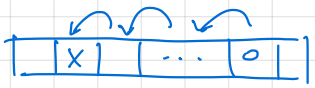
```
DFS(s)  // start from source
```
} all edges will be examined once in either direction (A)⟷(B) ⇒ O(2E) = O(E)

} O(V+E)

path reconstruction tracing back traversal sequence after termination

↳ metadata of traversal is stored in "preceding" array

↳ to see how to get back to source from some node (that it connected, would have been explored) just trace back



iterative

```
i = u
while i != s
    print(i)
    i = preceding[i]
print(s)
```

O(V)

⇒ traces in reverse order

recursive

```
backtrack(u)
    if u == -1
        print(u) // source
    else
        backtrack(preceding[u])
        print(u)
```

⇒ traces in correct order from source

---

modifying traversal algorithms

↳ graphs are good ways of representing problems where nodes have relationships to one another in a "spatial" way

↳ for problems we model as graphs, we will probably want a systematic way to start from a source node and explore / do something at neighboring nodes until some terminating condition

1. start from source node
2. conditionally explore neighboring nodes, do something
3. terminate?

BFS

```
while !q.isempty()  // terminate when all nodes explored
    u = q.dequeue()
    { do stuff @ node } — eg. if condition, return
                          else
    neighbors = A[i]  // adj list
    for all v in neighbors  // find neighbors to explore
        if visited[v] != true && condition // if not visited && should be
            visited[v] = true  // mark as visited
            q.enqueue(v)  // enqueue for exploration
            preceding[v] = u  // record history
```

DFS

```
DFS(u)
    visited[u] = true
    { do stuff @ node } — eg. if condition, return
                          else
    neighbors = A[u]  // explore neighbors in DFS way
    for all v in neighbors
        if visited[v] != true && condition // not visited && should be
            preceding[v] = u  // set predecessor
            DFS(v)  // explore all the way
```

② Reachability

↳ test whether vertex v is reachable from u

```
BFS(u) // or DFS(u)
return visited[v] == true
```

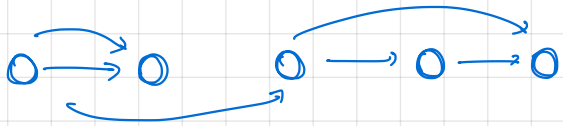
---

③ topological sort

directed acyclic graph — directed graph that has no cycle. Note that in directed graphs, cycles must have edges pointing in the same direction

topological ordering — topological ordering of a directed acyclic graph is a linear ordering in which each vertex comes before all vertices to which it has outbound edges

eg.



every DAG has one or more topological sorts

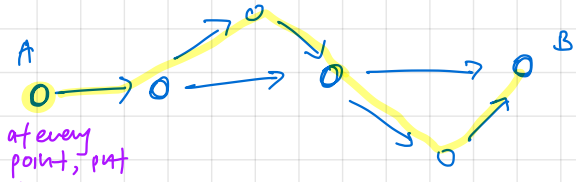
DFS-based topological sort

↳ idea: process node only after all descendent nodes have been processed. Apply recursively. reverse sequence to get topological sequence.

```
DFS(u)
    visited[u] = true
    neighbours = A[u]
    for v in neighbours      // do all neighbours
        preceding[v] = u
        DFS(v)
    out.append(u)            // process only after descendents
```

```
for v in V
    visited[v] = false
    preceding[v] = -1

out = []
for v in V
    if visited[v] == false   // include even if disconnected
        DFS(v)
out.reverse()                // reverse to get toposort sequence
```

without reverse, is "post order" traversal



at every point, put to sequence only after all children are done

① first of DFS toposort is a node that can reach all other nodes in a single-track, if it exists.

② it goes both ways. on reverse graph to find B.

(BFS-based) Kahns Algorithm

↳ intuition: count the number of incoming edges for each node. Enqueue those that have no incoming edges. Remove every one of these 0-incoming nodes. Everytime you remove, lower the in-degree of neighbouring nodes, since that incoming edge is gone. If its in-degree after removal is 0, enqueue it. By the proof, there will always be at least one node with 0-in-degree until the graph is empty.

```
for all v in V
    indeg[v] = 0
    preceding[v] = -1

for e in edges
    indeg[e.to] += 1

for all v in V
    if indeg[v] == 0
        q.enqueue(v)
```

initialisation. $O(2V) + O(E)$

```
while !q.isEmpty()
    u = q.dequeue() // remove. So remove indeg from neighbours
    out.append(u) // topologically sorted sequence
    neighbours = A[u]
    for i in neighbours
        indeg[i] -= 1      // notice can repeatedly visit
        if indeg[i] == 0
            q.enqueue(i)
            preceding[i] = u
return out
```

$O(V+E)$

(A) connected components

(connected components (undirected graph)

connected component a maximal group of vertices in an undirected graph that can visit one another via some path. 1 component $\iff$ connected graph

↳ idea: we use DFS / BFS to traverse all disconnected subgraphs

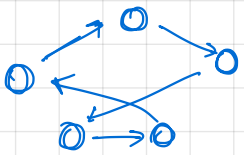
```
for v in V
    visited[v] = false

for all v in V
    if !visited[v]
        // do something eg. append node to list
        count += 1
        DFS(v) // or BFS also works
```

$O(V)$

not all will call DFS. In the end, still $O(V+E)$

strongly connected components (directed graph)

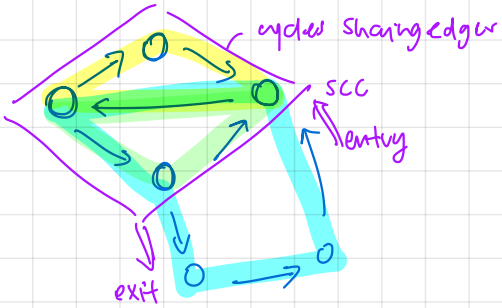
strongly connected component — a subgraph of a directed graph containing one or more vertices in which any two are connected to each other by at least one path, and is maximal
ie. can reach any vertex from any vertex



identifying SCCs — in directed graphs, SCCs are unique in the sense that within those subgraphs, all vertices must be able to reach one another despite constraint of edge direction.

⇒ look out for:

implicit (tangled) cycles sharing edges — bidirected edges — exit & entry points



Kosaraju's algorithm

↳ idea: given any SCC, reversing its edges will still result in the same SCC. If we use DFS to visit all nodes in the transposed graph, but in original topological ordering, reversed edges will stop DFS from exploring "future" SCCs.

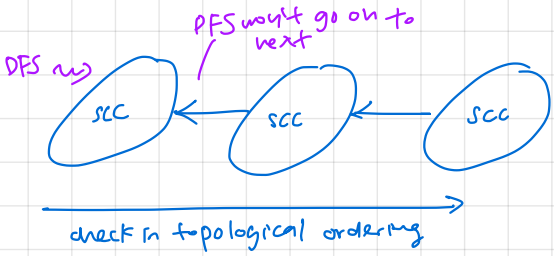
1. DFS topological sort ⇒ topologically sorted sequence of vertices
2. create a transposed graph by eg. Adj list

```
for v in V
    neighbours = A[v]
    for u in neighbours
        B[u].append(<v,weight>) // point in other direction
```

↳ if don't transpose, will just cascade down all the way (all SCCs w no in-edges)

3. explore SCCs one at a time

```
for v in V
    visited[v] = false

for v in toposort
    if !visited[v]   // start of new SCC
        cnt++=1      // do something
        DFS(v) // using transposed graph
```



⑤ bi-partite graph detection

Bipartite graph — undirected graph where we can partition (i.e. colour in alternating fashion) the vertices into two sets such that there are no edges between members of the same set

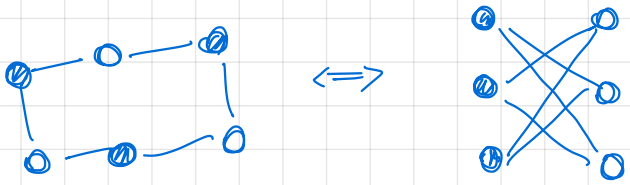
↕

1. **2-colourable**: it is possible to assign a colour to every vertex such that every vertex is coloured one of two colours (say, red or blue) such that no two adjacent vertices are coloured with the same colour

↕

2. **No odd-length cycles**: every cycle in the graph contains an even number of edges

⇒ visual intuition:



DFS-based colouring

↳ intuition: flag to indicate if graph is bipartite. check if colour of node is different from colour to be assigned ⇒ O(V+E)

```
for v in V
    colour[v] = -1    // initially uncoloured

DFS(n, c)
    if colour[n] != -1    // if coloured
        if colour[n] != c    // must be correct colour
            return false

    else
        colour[n] = c    // set colour
        neighbours = A[n]
        c = (c+1) % 2    // colour is 0 or 1
        for v in neighbours
            DFS(v, c)
```

implicit no-cycle check. If already coloured (i.e. visited before), check that correct colour. Then return (go back)

traverse & colour

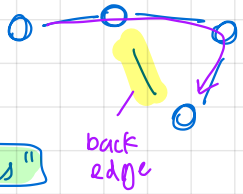
⑥ cycle detection

1) undirected graph

using $|E| = |V| - 1$

↳ if graph is undirected, acyclic graphs are trees. Trees have $|E| = |V| - 1$.
we can check by counting $|E|$.

↳ Data structures
- adj. matrix ⇒ $O(V^2)$
- adj. list ⇒ $O(V)$
- edge list ⇒ $O(E)$



find presence of "back edges"

↳ use modified DFS / BFS algorithm

```
DFS(u)
    visited[u] = true
    neighbors = A[u]
    for all v in neighbors
        if visited[v] // found back edge
            return true
        else
            preceding[v] = u
            DFS(v)
```

2) directed graph

DAG must have topological ordering

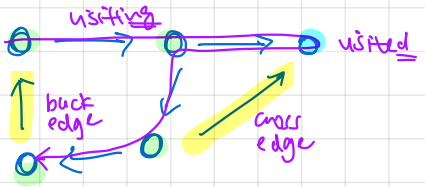
```
t = Kahns algo // or DFS toposort
for i in t
    neighbors = A[i]
    for v in neighbors
        if v < i // if behind in sequence
            return true // invalid toposort
        else
            continue
return false // valid toposort, no cycle
```

DAG has $|V|$ SCCs

↳ use Kosarajus algorithm to count number of SCCs. return n_SCC == |V|

finding back edges

↳ only a cycle if DFS hits a node on the from the same path that is not parent
↳ distinguish unvisited, visited, visiting




```
for v in V
    status[v] = -1

DFS(u, parent)
    status[u] = 1 // visiting
    neighbors = A[u]
    for v in neighbors // explore branches
        if v != parent && status[v] = 1
            return true // back edge (cycle)
        else
            DFS(v, u)

    status[u] = 2 // done visiting, back
                  tracked till here
```

⑦ Graph Colouring

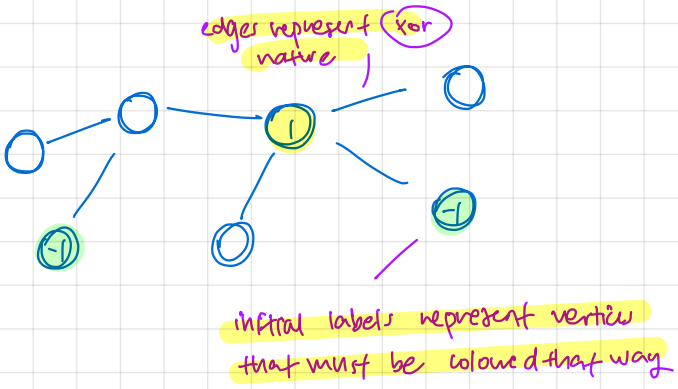
↳ we can model problems that have some sort of adjacency property (eg. A & B must be alternating, cannot be next to each other) with graph colouring

Valid configuration problems

↳ suppose we are given a problem where we know some (a, b) must both be present, a xor b must be present and neither a nor b should be present.
↳ colouring condition!

1. we model must be present as '1', must not be present as '2' vertices, and put an edge between xor vertices
   - ① initial colour state constraint
   - ② xor constraint

```
for V = 0 to |V| - 1
    colour[v] = -1    // initially uncoloured

for i in instructions
    a, b, type = instructions[i]
    if type == 'XOR'    // add edge
        adjList[a].append(b)
        adjList[b].append(a)

    if type == BOTH    // colour
        colour[a] = 1
        colour[b] = 1

    else if type == NEITHER
        colour[a] = 2
        colour[b] = 2
```



2. then we run colouring using DFS — to see if a valid state can exist

```
DFS(u, c)
    if colour[u] != -1    // if coloured
        if colour[u] != c    // must be correct colour
            return false    // cannot colour

    else
        colour[u] = c    // set colour
        neighbours = A[u]
        c = (c + 1) % 2    // colour is 0 or 1
        for v in neighbours
            DFS(v, c)
```

- implicit no-cycle check. If already coloured (i.e. visited before), check that correct colour. Then return (go back)
- traverse & colour

3. If DFS returns true, then a unique valid state is present — so long as at least one node restricted, there must be a unique solution

optimal configuration problem

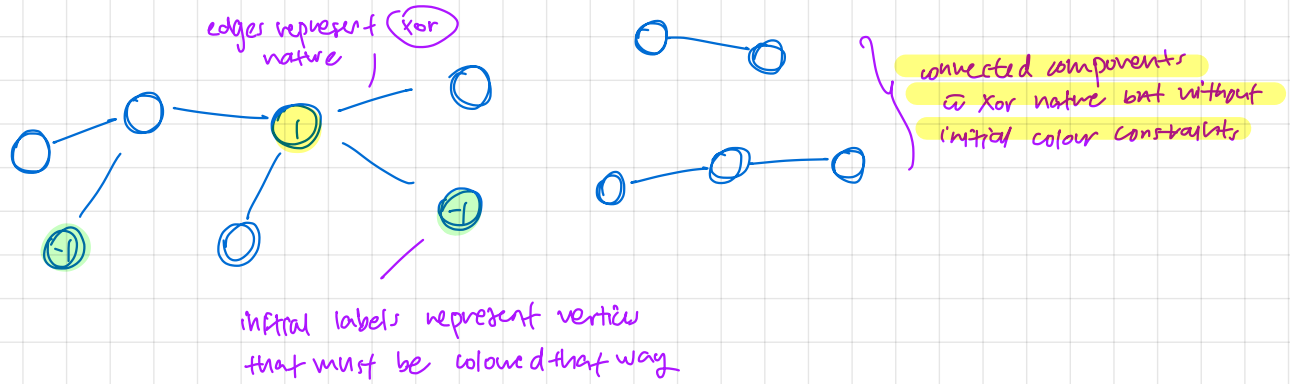
↳ suppose we are given a problem where we know some (a, b) must both be present, a xor b must be present and neither a nor b should be present.

↳ suppose we want to find the minimum number of vertices that must be of type — how?

intuition we can set it up as a valid state problem, but with additional considerations.

1. if there is at least one initially coloured vertex, then by xor nature, there is exactly one unique state possible if it exists
2. if there are no initially coloured vertices in a given connected component, then we must find the optimal

1. build graph



2. identify a node in all connected components + see if they have at least 1 colour constrained node

```
for v in V                                  } O(V)
    visited[v] = false

count = 0
for all v in V
    if !visited[v]
        starters.append(v)                  } not all will call DFS.
        count += 1                          } in the end, still O(V+E)
        colored = DFS(v)  // returns true if at least 1 colored
        starters_colored.append(colored)
```

3. for each component, run check. DFS_colouring colours & counts number of type 1 & type 2

```
wcnt = 0
for u in starters
    if starters_coloured[u]
        possible, num_t1, num_t2 = DFS_colouring(u, u.colour)
        if !possible
            return -1
        else
            count t= num_t1  // count vertices w type 1
    else // no coloured node in component
        possible, num_t1, num_t2 = DFS_colouring(u, 1) // start w arbitrary colouring
        if ! possible
            return -1
        else wcnt t= min(num_t1, num_t2)  // interchangeable colouring
```

Minimum spanning tree

① connected graphs and their MST

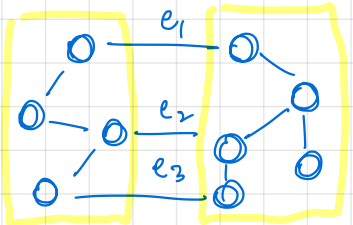
cycle property For any cycle in graph $G(V, E)$, if an edge e in c is larger than every other edge e' in c, then e cannot be in the MST of G. ie. max edge of any cycle is not in the MST

cut of a connected graph any partition of vertices of G into 2 disjoint subsets

cut set the set of edges that cross a cut

cut vertex a vertex in an undirected connected graph that upon removal disconnects the graph

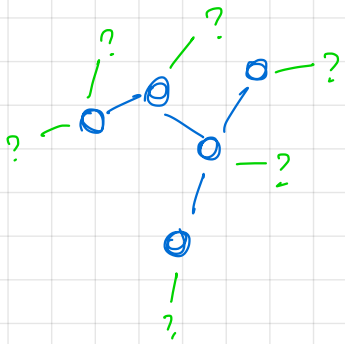
cut property of a connected graph For any cut of a connected graph, if the weight of an edge e from the cut set is strictly smaller than the weights of all other edges in the cut set, then it belongs to all MSTs of the graph.



② Algorithms to get MSTs

prim's algorithm

↳ grows an MST from a (trivial) subtree by considering the least weighted edge that can be added at the moment (a "local" view)



```
T = { source }
pq.enqueue(<source, MIN>)  // node-weight pairs

while !pq.isEmpty()
    u = pq.extractMin()  // min weight edge connected
    if !seen[u]
        T.add(u)  // if not inside, add to MST
        neighbours = A[u]
        for v in neighbours  // add all unvisited neighbours for consideration
            if !seen[v]
                pq.enqueue(<v, w(u,v)>)
                seen[v] = true
```

complexity

↳ Each edge processed twice, one to enqueue, another to dequeue. So there are $O(E)$ such operations.

↳ Each enqueue and dequeue is $O(\log K)$, where K is the size of the pq at that time. On average, it is $O(E)$ size.

$$\Rightarrow O(E) \times O(\log E) = O(E) \times O(\log V^2) = O(E \log V)$$

intuition

↳ prim's algorithm always greedily selects the smallest edge in a given cut, and thus builds an MST

prim's variant for dense graphs

↳ for dense graphs where $E = O(V^2)$, time complexity of prim's is $O(V^2 \log V)$

⇒ maintainance of PQ is more expensive than it is worth. So we replace the pq w/ a simple array and do a linear $O(V)$ scan each time we search for the minimum. We mark weight as ∞ for edges that have been included in the MST

```
source = <0, s>
T = {source}
for v = 0 to |V|-1                    // O(V)
    A[v] = <∞, v>

while T.size() != |V|                 // not all vertices found   O(V^2)
    u = findMin(A)
    A[u] = <∞, u>                     // never min again, implicit dequeue
    neighbors = adj[u]
    for v in neighbors
        if v not in T && A[v].weight > w(u,v)
            A[v] = <w(v,u), v>        // implicit enqueue
```

Kruskal's algorithm

↳ growing multiple MSTs as a forest until they merge and become a single MST

```
T = {}                                // initialize tree set
set = nFDS(|V|)                       // nFDS w/ size |V|

edges.sort()                          // O(E log V), most work done here
for v = 0 to |V|-1
    e = edges[v]                      // min, since sorted
    if set.isSameset(e.from, e.to)    // no cycle
        T.add(e)
        set.union(e.from, e.to)
        if T.size == |V|
            break                     // found
```

$O(E) \times O(\alpha(V)) \approx O(E)$

intuition use the cut property. So always include global minimum edges & their vertices.

③ MST algorithms

1) updating MSTs ⟶ non-unique MST ⟶ re-run prims / kruskals

updating MSTs given new edges iff unique MST

↳ intuition: a tree has the least number of edges to be acyclic. So adding any edges will create a cycle. Because there's only one unique path between any two vertices in a tree, we can use DFS to find the current unique path between A and B and find the heaviest edge, compare that with the new edge, and swap them if needed.

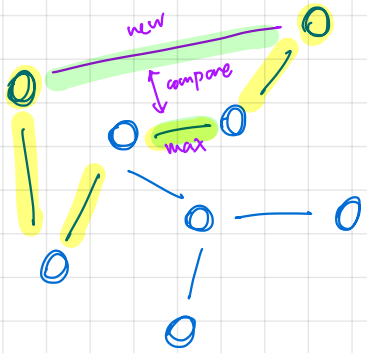
```
DFS(u, target)
    visited[u] = true
    if u == target
        return
    else
        neighbours = A[u]
        for v in neighbours
            if !visited[v]
                parent[v] = u
                DFS(v)
    return
```

1. DFS(A, B) ⇒ parent ⇒ reverse } O(V) on tree
2. path reconstruction, find max

```
v = parent[B]
max = v.weight
while v != A
    if v.weight > max.weight
        max = v
    v = parent[v]
```

} O(V)

3. if max > new.weight, replace edge.

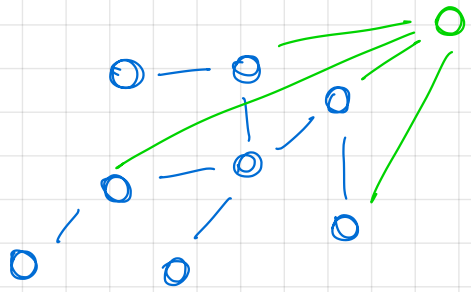


---

updating MSTs given new vertices iff unique MST

↳ intuition: when we add a new vertex and all its edges to all nodes in the graph, the new MST might be entirely different — in that it might use up to all of the new edges. so assuming the MST is unique, then the new MST must only come from the union of current MST and new vertex and edges.

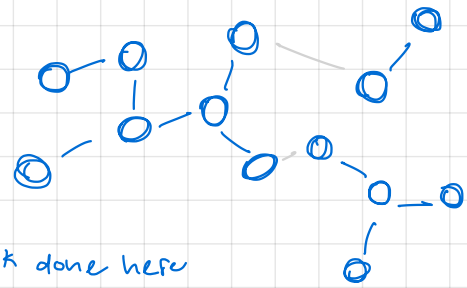
1. Add new vertices and edges to MST
2. run Prims / Kruskals ⇒ O(V log V)



growing minimum weight forest

no control over sources, but total forest weight

↳ intuition: Kruskal grows subtrees. we can add a termination condition so that we have k subtrees and a minimum weight forest.



```
T = {}  // initialize tree set
set = UFDS(|V|)  // UFDS w size |V|
total = 0

edges.sort()                                  ↳ O(ElogV), most work done here
for v = 0 to |V|-1
    e = edges[v]  // min, since sorted
    if total + e.weight > max
        return  // stop, min forest found
    if set.!isSameSet(e.from, e.to)  // no cycle      } O(E) x O(α(V))
        T.add(e)
        set.union(e.from, e.to)
        if T.size == |V|
            break  // found
```

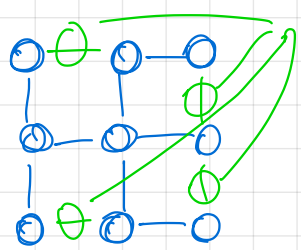
---

min/max cycle coverage

↳ problem: suppose given a graph, we want a set of edges such that there is at least one edge in every cycle in the graph, and have that be a max/min weight edge set

↳ intuition: an tree is acyclic. the introduction of any up to all remaining edges of the graph will create all cycles in the graph. So, we can view all edges not in the tree as all edges that are present in all cycles.

⇒ we can find a min/max MST to get all these edges as the complement to the MST's edge set



multi-source MST control over sources, but not limit on total forest weight

↳ intuition: prim's algorithm uses a greedy approach to choose the lowest weight edge connected to the subgraph at that moment. To grow a multi-source MST, ie. those sources must be in the forest, we can initialise the priority queue with all the sources, setting their default weight to min.

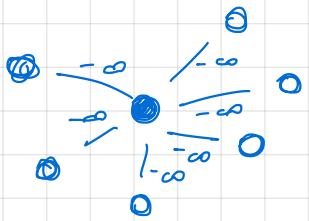
⇒ we will grow k subtrees, given k sources

1. initialise PQ w k sources

```
for k in sources
    T.add(k)
    pq.enqueue<k, MIN> // node-weight pairs
```

2. add global source node, connect to all k sources w min weight

```
T = {global}
pq.enqueue<global, MIN>
```




```
while !pq.isEmpty()
    u = pq.extractMin() // min weight edge connected
    if !seen[u] // if not inside, add
        T.add(u)
        neighbours = A[u]
        for v in neighbours // add all unvisited neighbours for consideration
            if !seen[v]
                pq.enqueue(<v, w(u,v)>)
                seen[v] = true
```

remove edges

Kruskal's for maximin from A ~~> B

↳ useful when all we want is the minimax / maximin edge of minimax / maximin path from A to B

1. Suppose source is A to B.

```
T = {} // initialize tree set
set = UFDS(|V|) // UFDS w size |V|

edges.sort() // in ascending for maximin
for v = 0 to |V|-1
    e = edges[v] // min, since sorted
    if set.isSameset(e.from, e.to) // no cycle
        T.add(e)
        set.union(e.from, e.to)
        maximin = edge.weight
    if set.isSameset(a, b) // path found
        break

- return maximin
```

↳ O(E log V), most work done here

O(E) x O(α(V))
≈ O(E)

intuition: to get maximin / minimax tree, we simply keep including edges in sorted order until A & B are same component

maximin : ascending order
minimax : descending order

---

reconnecting maximin tree

↳ suppose edge removed. then we are left with some maximin path trees

1. run connected components algorithm on T1 & T2 to label them
2. for all edges, check that not in 1, 2 but will connect 1x2 and is new maximin

Shortest paths

① Shortest path

1) definitions

(simple) path sequence of vertices connected by edges. simple ⇒ no repeated vertices.

$p = \langle v_0, v_1 \dots v_k \rangle$ and $(v_i, v_{i+1}) \in E \ \forall i$

path weight sum of all edge weights along a path, $\sum_{i=0}^{k-1} w(v_i, v_{i+1})$

shortest path weight

$$\delta(a,b) = \begin{cases} \min \text{ path weight}(a,b) & \text{if such a path exists} \\ \infty & \text{otherwise} \end{cases}$$

shortest path path from a to b w. shortest path weight

2) cycles

positive weight cycle a cycle where the path weight is positive. It can contain a negative edge, but the total weight is positive.

negative weight cycle a cycle where the path weight is negative. It can contain a positive edge, but the total weight is negative

⇒ shortest paths cannot contain cycles

↳ a shortest path cannot contain a positive weight cycle, since we can reduce the path weight by removing the cycle, or in a zero-weight cycle, just remove it.

↳ a shortest path also cannot contain a negative weight cycle, since we can always find a path with lower weight by traversing the negative weight cycle one more time

shortest path cycle property if $G = (V, E)$ contains no negative weight cycles, then the shortest path p from a given vertex a to b is a simple path

3) results

subpaths of shortest paths are shortest paths

1. suppose subpath p_{ij} in shortest path p_{ab} is not shortest path between i & j.
2. Then we can make p_{ab} shorter by replacing it w. p'_{ij}. But p_{ab} is shortest.

⇒ contradiction

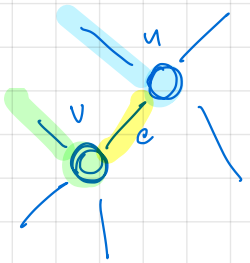
① single source shortest path

⇒ given a source node, how do we find the shortest path from it to all other nodes?

1) the "relax" operation → we can always modify how we relax to fit the problem

eg - min along all paths

$D[v] = \min(D[u], D[v])$



↳ suppose we currently have some path to v from source, length D[v], and one to u D[u]. e has not been explored yet.

↳ Now, if $D[u] + e.weight < D[v]$, then we have found a shorter path to v — by going to u, then traversing that edge.

⇒ we can recursively apply this logic by looking at every unexamined edge to find the shortest path to a given node

---

⇒ algorithms for SSSP

Bellman Ford — graph ∄ no negative cycle

```
for i=0 to |V|-1                    } initialisation, O(V)
    D[v] = ∞
    prev[v] = -1
D[s] = 0

for i = 1 to |V|-1                  } O(VxE)
    for e in edgeList
        u = e.from
        v = e.to
        weight = e.weight
        if D[v] > D[u] + weight     } relax
            D[v] = D[u] + weight
            prev[v] = u
            change = true // boolean flag
    if !change
        break
    else
        change = false
```

intuition

↳ we initialise the graph such that $D[s] = 0$, but $D[v]\ \forall\ v \in V - \{s\} = \infty$. so the relax operation done on an edge not involving s will leave $D[v] = \infty$. only those directly connected to s will relax.

↳ if we relax all edges again, now only those connected to the 1° neighbourhood of s will be relaxed.

⇒ Bellman Ford algorithm capitalizes on this idea: a given node is at max |v-1| edges away from the source. so if we relax all edges |V-1| times, we are guaranteed to relax all edges and find the SSSP to all nodes.

negative cycles

if D[v] = ∞ after |V|-1 iterations then a negative weight cycle must exist.

```
for e in edgeList
    if D[e.from] != ∞          ← visited before
        if D[e.to] > D[e.from] + e.weight   ← can still be relaxed
            //negative cycle found
```

Dijkstra's algorithm — graph w/ no negative weights, $w(u,v) \geq 0$

```
for i=0 to |V|-1
    D[i] = ∞
    p[i] = -1
D[s] = 0
```
} initialisation O(V)

```
pq = BST()
for i=0 to |V|-1
    pq.enqueue <D[i], i>
```
} initialise pq (BST), O(V log V)

```
while !pq.isempty()
    distance, u = pq.extractMin()   // get nearest to source
    neighbours = adjList[u]
    for v in neighbours
        if D[v] > D[u] + distance   // relax outgoing
            D[v] = D[u] + distance
            pq.delete(v)            // update pq after relax
            pq.insert(<D[v], v>)
            p[v] = u
```

⇒ return D, p

intuition

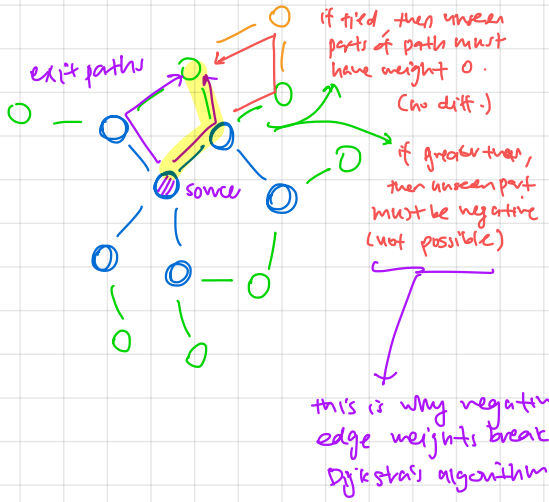
↳ the relax operation will always give you the shortest path from source to a given node, given the edges you have seen

↳ at a given point in the algorithm, we have a set of seen nodes we know the shortest distance & path to, and a "layer" around this solved set

↳ for each node in this layer, there can be multiple paths from source

↳ we pick the one with the shortest exit path to add to the solved set. We know we cannot find a shorter path to this outer node, because it is the minimum (closest) among all outer nodes.

- if there is another node of equal distance, then there could be an alternate path, but it would be an added 0 weight path. In which case, no difference.
- if no other node of equal distance, then only way to find shorter path is unseen nodes & edges is to have a negative edge.



complexity

↳ each vertex will only be extracted once, O(V log V)

↳ each edge is processed once, and can trigger an update in the PQ, which takes 2x O(log V) time; so O(E log V)

⇒ O((V+E) log V)

Modified Dijkstra's algorithm — no negative cycles

intuition

↳ original algorithm greedily searches and stops looking at a given vertex once we find its shortest "exit path". problem arises with negative edges, because a shorter path could be found later

↳ so we can modify the algorithm to, upon finding that a vertex has a new shortest distance (successful relaxation), put it up again for further relaxation

↳ we can do so lazily w a simple trick, so we don't have to update the PQ

↳ algorithm terminates when no relaxations occur → so everything is popped from PQ

```
for i=0 to |V|-1
    D[i] = ∞
    p[i] = -1
D[s] = 0
```
} initialisation O(V)

```
pq = PQ()
pq.enqueue(<0, source>)

while !pq.isEmpty()
    distance, u = pq.extractMin()
    if distance == D[u]  // latest
        neighbours = adjList[u]
        for v in neighbours
            if D[v] > D[u] + weight(u,v)  // relax
                D[v] = D[u] + weight(u,v)
                p[v] = u
                pq.enqueue(<D[v], v>)  // propagate
```

complexity

↳ when there are no negative edges there are at most as many relaxations as edges.

↳ so we can have at most E insertions and extractions from the PQ.

⇒ $O(E) \times O(\log E) = \underbrace{O(E \log E)}_{\text{modified}} \approx \underbrace{O((V+E)\log V)}_{\text{original}}$

DAG Bellman Ford

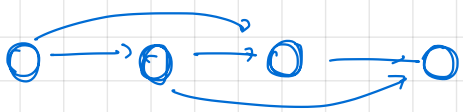
intuition

↳ every DAG has a topological sorted sequence

↳ we find D[v] wrt to first element of toposort as source

↳ for a given node in the sequence, by examining all edges in sequence, because there is no backtracking (no cycles), relating in topological order guarantees us to successfully compare all paths from source to node by time we reach it

⇒ so when reach end, done



```
toposort = k_sort(V, E)   // O(V+E)

for i=0 to |V|-1
    D[i] = ∞
    p[i] = -1
D[s] = 0

for u in toposort
    neighbours = adjList[u]
    for v in neighbours
        if D[v] > D[u] + weight(u,v)  // relax
            D[v] = D[u] + weight(u,v)
            p[v] = u
```

O(V+E)

complexity

every edge examined once at toposort. O(V+E)

2) special cases of SSSP

tree: DFS / BFS

↳ in a tree, there is exactly one path between any two nodes.
↳ so a DFS or BFS could work and handle negative edges.

```
DFS(u, distance)
    visited[u] = true
    D[u] = distance
    neighbours = A[u]  // explore neighbours in DFS way
    for all <v, weight> in neighbours
        if visited[v] != true
            preceding[v] = u  // set predecessor
            DFS(v, distance + weight)  // explore all the way

D[s] = 0
DFS(s, D[s])
```

$O(V + V-1) = O(V)$
$|E|$ in a tree

---

unweighted graph: BFS

↳ in an unweighted graph, distance is counted by number of edges from source

```
q.enqueue(s)
D[s] = 0                          // initialisation

while !q.isempty()                // terminate when all nodes explored
    u = q.dequeue()
    neighbours = A[i]             // adj list
    for all v in neighbours       // find neighbours to explore
        if visited[v] != true
            visited[v] = true     // mark as visited
            q.enqueue(v)          // enqueue for exploration
            preceding[v] = u      // record history
            D[v] = D[u] + 1       // one degree further
```

$O(V + V-1) = O(V)$

③ multi source shortest path

⇒ given source nodes of some type, how can we find the shortest path from a node to a source and which source it is closest to?

↳ we introduce an 'ID' into the mix, and instantiate our queue w source nodes already inside

unweighted graph : BFS

```
for s in sources                          // distance, ID
    pq.enqueue <0, s, s.type>

while !q.isempty()  // terminate when all nodes explored
    u, type, count = q.dequeue()
    neighbours = A[i]   // adj list
    for all v in neighbours  // find neighbours to explore
        if visited [v] != true or count+1 < D[v]  // not visited or shorter path found
            visited [v] = true  // mark as visited
            type [v] = type  // update type
            D[v] = count+1  // update distance
            preceding [v] = u  // record history
            q.enqueue (v, type, count+1)   // enqueue for exploration
```

O(V+E)

⇒ D[] : distance to nearest source
T[] : type of nearest source
P[] : parent

check nearest source @ nods of interest

---

weighted graph : Dijkstra's

```
for i=0 to |V|-1
    D[i] = ∞
    p[i] = -1
D[s] = 0

pq = PQ()
for s in sources
    pq.enqueue <0, s, s.type>          // type of closest source

while !pq.isEmpty()
    distance, u, type = pq.extract min()
    if distance == D[u]  // latest
        neighbours = adjList [u]
        for v in neighbours
            if D[v] > D[u] + weight(u,v)  // relax
                D[v] = D[u] + weight(u,v)
                p[v] = u
                type [v] = type  // update type
                pq.enqueue (<D[v], v, type>)  // propagate
```

(or) instead of tracking type, use n distance arrays. at end find nearest source.

O(E log E)

⇒ D[] : distance to nearest source
T[] : type of nearest source
P[] : parent

check nearest source @ nods of interest

④ SSSP problems

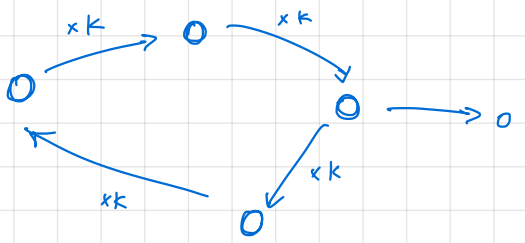
weighted cycle detection

↳ we can easily do cycle detection using DFS in both directed & undirected graphs

↳ but what if we want to detect cycles of a particular weight value?

⇒ do something to transform cycle to negative cycles, use Bellman Ford to detect

eg. multiplicative profit



1. log transform $x \times x \times \ldots \Rightarrow \log(x) + \log(x) \ldots$
2. negate $\Rightarrow -\log(x) - \log(x) \ldots$
3. if $x \times x \times \ldots > 1$ (profit), will be negative

⇒ detect

$$a \times b \times c \ldots > 1$$
$$\Leftrightarrow \log a + \log b \ldots > 0$$
$$\Leftrightarrow -\log a - \log b \ldots < 0$$

---

state-space graphs

↳ suppose a vertex represents a state. eg. x, y, θ (position, direction) ⇒ then 3 dimensional state

↳ edge: cost to transition from state to state

⇒ key insight: we use any SSSP algorithm, but each vertex is unique state, and each edge is transition cost.

↳ consider → starting state
↳ end state(s) eg. x, y fixed, θ any → multiple ending states, choose min within them

for integer weight edges bound by k, we can transform the graph into undirected graph w $O(kE)$ vertices

---

joint shortest path

↳ suppose we want to find some (C) between (A) and (B) s.t. total distance from A to C and A to B is minimum (can extend this intuitively — key point is an intersection)

↳ we run SSSP algo from A, B ... once each. ⇒ constrain to $O(1) \times O(\text{SSSP algo})$

⇒ for every $C \in V$, find min. total distance

```
A ← SSSP(a, G)
B ← SSSP(b, G)
for v in V
    d_a = A[v]
    d_b = B[v]
    total = min(d_a + d_b, total)
```

SSSP to model relative measures

↳ observe that SSSP algorithms give you some metric about a source to all other nodes.

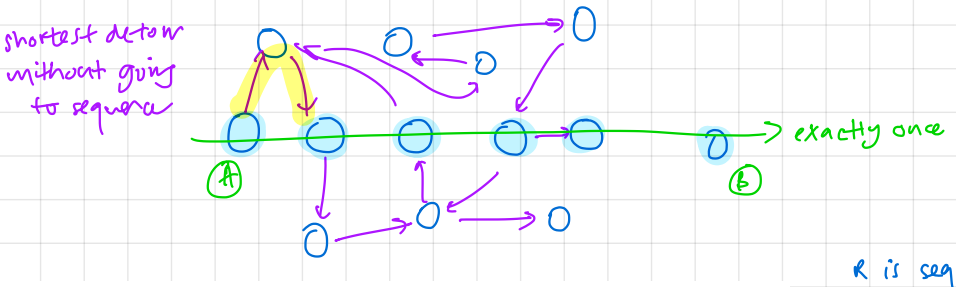
↳ if we can interpret this number as some relative (eg. temperature), then we can use source as 0 (reference value) and distance as a relative, like temperature. Just SSSP once

⇒ ΔT between a & b = $T[a] - T[b]$ wrt. $T[source]$

---

⑤ multistep shortest path

↳ suppose we have a sequence we can only traverse exactly once among all vertices and all others can repeat



Vertices are rooms and doors are undirected edges connecting adjacent rooms. Edge weight is the time taken for the door represented by the edge to open.

Let A = r_0 and B = r_{k+1}.

Since Dhell has to go through the rooms in R in their order given, shortest path to escape is SP(A,r_1)+SP(r_1,r_2)+...+SP(r_k,B).

However in the shortest path from r_0 to r_1 we cannot go through any of the rooms {r_2,r_3,.., r_{k+1}}. Similarly in the shortest path from r1 to r2 we cannot go through any of the rooms {r_0,r_3,.., r_{k+1}}, and so on.

Algorithm:

1. Hash all rooms in the sequence r_0 to r_{k+1} into a hashmap H.
2. Keep a shortest path cost C initialized to infinity.
3. Now run Dijkstra for i from 0 to K times:
   - At iteration i, remove r_i and r_{i+1} from H and run from r_i as source vertex. Whenever we relax an edge (u,v) check whether v is in H. If it is do not relax it (v is not allowed along the path). Otherwise relax it.
   - After Dijkstra's finish, add D[r_{i+1}] to C.
   - Before going to next iteration add r_i and r_{i+1} back into the hashmap.

C will contain the shortest path cost to get Dhell from A to B such that the portal will open.

Time complexity is O(K*(V+E)*logV) since we run Dijkstra's K+1 times.

R is sequence of nodes, K of them

```
for r in R
    h.put(r)

for r = 0 to |R| - 1
    cost[r] = ∞

for i = 0 to K-2          // remove to allow visitation of i & i+1
    h.delete(R[i], R[i+1])
    D = dijkstras(i, G)   // graph, source
                          // does not allow if node in R
    // min cost to next in sequence
    cost[i+1] = D[R[i+1]]

    // put i and i+1 back
    h.put(R[i], R[i+1])
```

⇒ time complexity is

O(K(V+E) log V)

⑥ SSSP w/ exact number of edges

↳ if we are given a problem to find SSSP w/ exact edge counts, of a certain kind, then unlike the case where edge count can put into PQ as a tie breaker, we must model as a state-space graph

1. for each vertex in G, we create $n+1$ states.
   $0, 1 \ldots n$ such edges traversed

```
adjList = arr[v][n+1]
```

2. connect states by identifying transition edges (special edge that increase count)

```
for edge in edgeList
    if edge.type == SPECIAL   // transition edge
        u = edge.from
        v = edge.to
        for i = 0 to n-1      // all n possible transitions
            adjList[u][i].append(<u, v, weight, i+1>)   // i → i+1
    else   // generic edge, no transition to next state / number of specific edges
        u = edge.from
        v = edge.to
        for i=0 to n
            adjList[u][i].append(<u, v, weight, i>)   // lateral transition, i→i
```

0 → 1 ... → n

O(E·n)

3. run Dijkstras, $O((V+E)\log V)$ time

---

⑦ multi path SSSP

idea: we use a predecessor "adjacency list" to keep track of tied paths.

⇒ during relaxation, if ==, append. Else if can relax, reset.

```
// Dijkstras
for v in neighbours
    if D[v] > D[u] + weight(u,v)   // relax
        D[v] = D[u] + weight(u,v)
        p[v] = new array          // reset array
        p[v].append(u)            // O(1) time
    else if D[v] == D[u] + weight(u,v)   // tie
        p[v].append(u)
```

path reconstruction

to sum up total weight of all paths

iterative

```
i = V
while i != source
    previous = preceding[i]
    for <u, weight> in previous
        total += weight
```

O(V)

⇒ traces in reverse order to sum weights

All pair shortest paths

① APSP

diameter of a graph The greatest shortest path distance between any pair of vertices in the graph

↳ how do we find the shortest path & path weight for all pairs of vertices?

---

② Floyd Warshall algorithm

```
for i=0 to |V|-1
    for j=0 to |V|-1
        if i==j
            A[i][j] = 0
        A[i][j] = ∞

for e in edges
    u = e.from
    v = e.to
    w = e.weight
    A[u][v] = w
```

initialisation of adj. matrix, $O(V^2)$

```
for k=0 to |V-1|   // consider k as an intermediate
    for i=0 to |V-1|
        for j=0 to |V-1|
            if D[i][j] > D[i][k] + D[k][j]
                D[i][j] = D[i][k] + D[k][j]
                p[i][j] = p[k][j]
```

intuition

↳ when no passes of the outerloop have occured, because of the way we initialized D, each A[i][j] contains the shortest distance from i to j without any intermediates

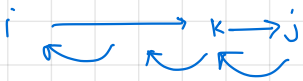
↳ The relax operation checks that given an intermediate edge, we can use the intermediate path if it is shorter

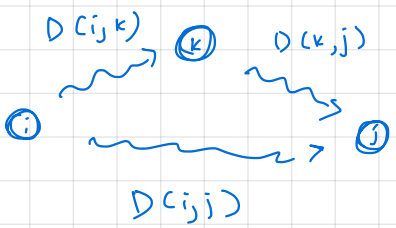
↳ so after 1 pass, we have checked if including ① as an intermediate from i to j reduces the distance. This also implicitly means we have updated the shortest D(i,k) and D(j,k) for the next iteration

↳ so after every iteration, we check if the inclusion of (k+1) will make the path between ①, ① shorter, prepping for the next iteration again

path reconstruction

```
i=i
j=j
while p[i][j] != i
    if D[i][j] != ∞   // valid path exists
        print(p[i][j])   // prints k, intermediate
        j = p[i][j]
    else
        // no path from i to j
```





complexity

↳ using SSSP algorithms, APSP complexity is $O(V) \times O(SSSP)$

⇒ Floyd Washall bounds to $O(V^2) + O(V^3)$

③ transitive closure as APSP

transitive closure given $G(V, E)$, determine if for all i, j there is a path connecting them

↳ for undirected graph, BFS or DFS is better $O(V+E)$

↳ for directed graph, VX DFS or APSP is optimal $O(V^3) \sim O(V(V+E))$

modification use 1 to indicate path present, 0 otherwise ⇒ check if $D[i][j] = 1$ for all

```
for i=0 to |V|-1
    for j=0 to |V|-1
        if i==j
            A[i][j] = 0
        A[i][j] = ∞

for e in edges
    u = e.from
    v = e.to
    A[u][v] = 1
```

initialisation of adj. matrix, $O(V^2)$

```
for k=0 to |V-1|   // consider k as an intermediate
    for i=0 to |V-1|
        for j=0 to |V-1|
            if D[i][j] == 0 && (D[i][k] == 1 && D[k][j] == 1)
                D[i][j] = 1   // valid path found
```

---

④ cycle detection

↳ in floyd marshal's algorithm, the consideration of itself as an intermediate should never change anything, since putting itself makes no difference — unless there is a cycle.

⇒ initialise $A[i][i] = \infty$ ⇒ $A[i][i]$

- $0 < x < \infty$ positive cycle
- $x < 0$ negative cycle
- $= 0$ zero cycle
- ∞ no cycle

```
for i=0 to |V|-1
    for j=0 to |V|-1
        A[i][j] = ∞
for e in edges
    u = e.from
    v = e.to
    w = e.weight
    A[u][v] = w
```

initialisation of adj. matrix, $O(V^2)$

```
for k=0 to |V-1|   // consider k as an intermediate
    for i=0 to |V-1|
        for j=0 to |V-1|
            if D[i][j] > D[i][k] + D[k][j]
                D[i][j] = D[i][k] + D[k][j]
                P[i][j] = P[k][j]
```