Memory Address:

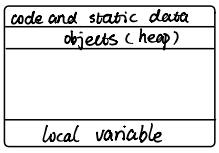
1. Structure of memory:

(1) Basic unit: bit 8 bits = 1 byte

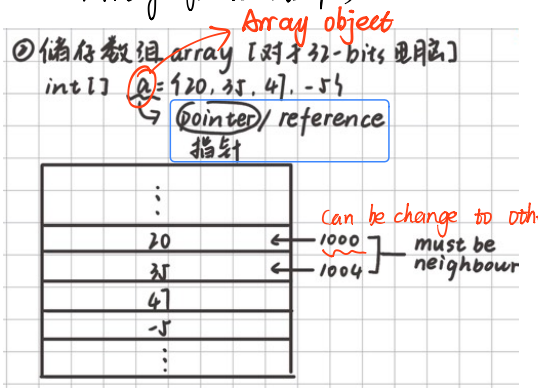
smallest addressing unit: byte 1 address ⟷ 1 byte

=> 32-bits laptop ⇒ $2^{32}-1$ addresses (4 billion)

2. Memory Allocation:



(1) Store Array (for 32-bits pc)



others also OK.

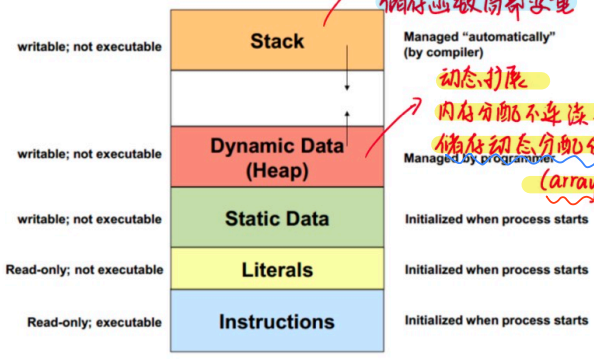
※ ① a points the the first element with address 1000

② invoke element a[i]: a + i x size of int

initial

interpretation: int[] a = new int[20]
totally 80 bytes
(20 x 4) size of data

(2) Stack 栈 Heap 堆 [两个重要的内存区域]



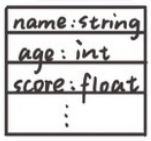
Once method is created ⇒ store local variable to stack frame (from stack)

Once method returns, its stack frame is erased.

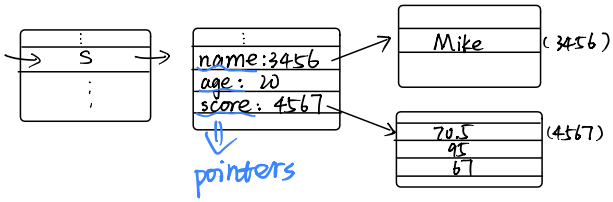
⇒ the value stored in heap will be erased, too

once new object is created ⇒ Java allocates space from heap.

eg: ①class student → string




```
Student s = new Student();
s.name = new String("Mike");
s.age = 20;
s.score = {70.5, 95, 6]};
```



If modify 'Mike' ⇒ ① a new address will be created to store new element and ② 'Mike' 3456 变成 garbage.

②

- The local variable a is allocated in the current stack frame and is assigned the value (address), which identifies the object.

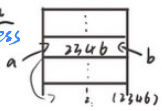
[Step 1] int[] a = {1, 2, 4, 4, 5}

[Step 2] b[2] = 5

What is a[2]? (5)

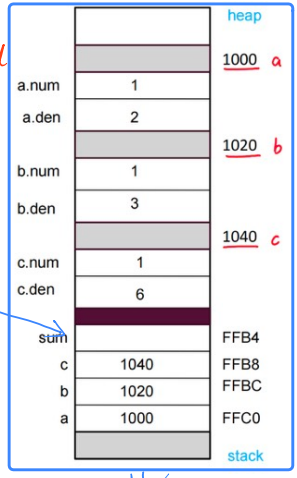
int[] b = a

copy address of a to b

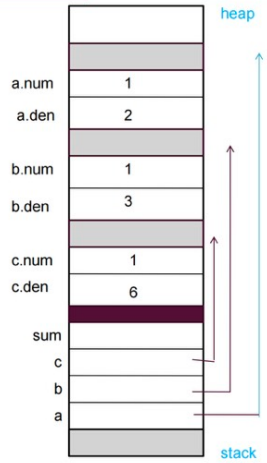


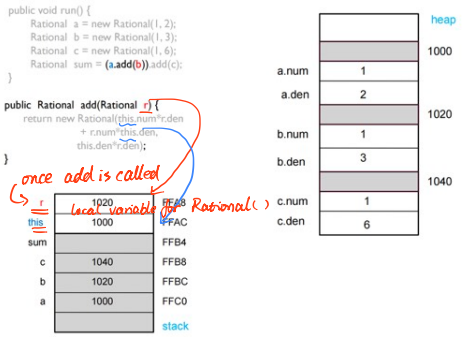
① address model:

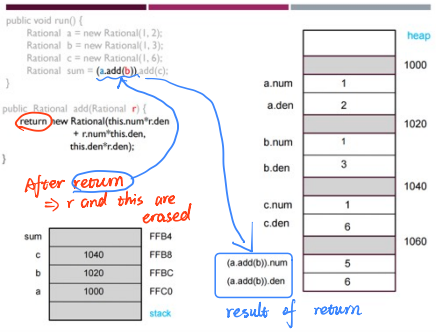
```
public void run(){
    Rational a = new Rational(1, 2);
    Rational b = new Rational(1, 3);
    Rational c = new Rational(1, 6);
    Rational sum = (a.add(b)).add(c);
}
```

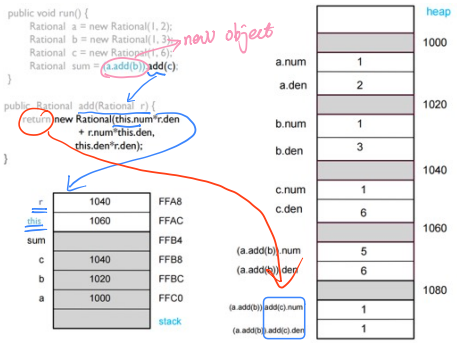


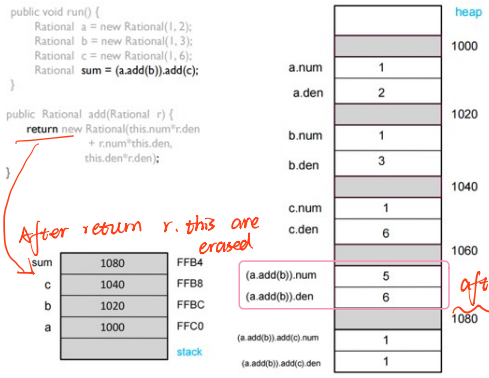
pointer model: ⇒











after add(c) ⇒ a.add(b) become garbage.

✅ 3. Performance Skepticism:

If you're inside a loop, blindly using `String` is dangerous:

```java
String result = "";
for (int i = 0; i < 1000; i++) {
    result += i; // Creates 1000 new String objects (inefficient)
}
```

immutable

Better approach:

```java
StringBuffer result = new StringBuffer();
for (int i = 0; i < 1000; i++) {
    result.append(i); // Efficient, no unnecessary objects
}
```

mutable

modify original data. (value)

Complexity Analysis: ☆ depend on input size and data structure.

An algorithm is a finite set of precise instructions for performing a computation or for solving a problem.

1. Basic operation: O(1)

- Arithmetic operations
  - a+b*c+d/e, a mod 7
- Assigning a value to a variable
  - x ← 3 ⇒ locating
- Indexing into an array / accessing value of objects/structs
  - arr[i], arr.length, student.name
- Calling a method
  - x ← factorial(n)
- Returning from a method
  - return y
- Comparison
  - x==y, x>y, x<y, x>=y, x<=y, x!=y

A reasonable assumption: each basic operation takes constant time

2. Analyzing running time:

(1) For loop:

① `for (i=0; i<n; i++):` from 0 to n ⇒ O(n+1)

- `i=0` : 1 (assigning O(1))
- `i<n` : compare `i<n` : n+1 (arithmetic O(n)) — 一直加到 i=n, stop.
- `i++` : increment for n times : n — 在比较 n=n 后, stop. 不会 increment i=n-1 to n

⇒ 2n+2.

eg: sum(A,n)

```
1  tempsum = 0                  → 1
2  for i = 0 to n-1             → 2n+2
3     tempsum += A[i]           → 2n   : increment for i and assign tempsum
4  return tempsum               → 1
```

② Nested loop: 数清每次传入的 input size

maxSubArraySum(A, n)

```
1  maxSum = A[0]                → 2 : ①Find A[0] ②Assign maxSum
2  for i = 0 to n-1             → 2n+2
3     subSumFromI = 0           → n
4     for j = i to n-1          : (2n+2)+(2(n-1)+2)+···+(2·1+2) ☆
5        subSumFromI += A[j]    → 2(n+(n-1)+···+1)
6        if subSumFromI > maxSum → (n+(n-1)+···+1)   = 左右两边 max的次数
7           maxSum = subSumFromI → (n+(n-1)+···+1)
8  return maxSum                → 1
```

j depend on i ⇒ plus

☆ 若 j independent with i:

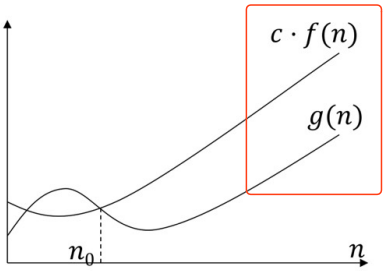
```
for (int i = 1; i < n; i *= 2) {  // Outer loop runs in log(n) steps
    for (int j = 0; j < n; j++) {  // Inner loop runs n times
        // Constant time operation
    }
}
```

⇒ O(logn)·O(n) = O(nlogn)

3. Big-Oh:

(1) Def: $g(n) = O(f(n))$ if and only if there exist positive constants c and n_0 such that $g(n) \le c \cdot f(n)$ for all $n \ge n_0$ ⇒ Find c, n_0

- $g(n)$ grows asymptotically no faster than $f(n)$
- $f(n)$ is usually much simpler than $g(n)$



① Care about tail
② c·f(n) is upper bounded.

Warning: upper bound is infinite.

eg: $2n = O(n^2)$

Fina a constant c, n_0 such that ☆ 例用化简

$2n \le cn^2$ for all $n \ge n_0 \Leftrightarrow 2 \le cn$ for all $n \ge n_0$.

Let $c = 2, n_0 = 1$, the above inequality holds. $g(n) = O(n^2)$

$g(n) = 2n = O(n \cdot \log n) = O(n^2) = O(n^3) = O(2^n)$

let $f(n) = 2n \cdot \ln n$ $c=2$

$2n \le 2n \cdot \ln n \Rightarrow 2n = O(n \cdot \ln n)$ ☆

① $g(n) = a \cdot n + b$, $a \ge 0$, $b \ge 0$:

⇒ let $f(n) = (a+b)n$, $c = a+b$.

When $n_0 = 1$: $g(n) = an + b \le an + bn$ for $n \ge 1$

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

② same type different base → 换底: ☆ $\log_a n = \frac{\log_b n}{\log_b a}$

$g(n) = \log_a(n) = O(\log_b n)$, $a>1$, $b>1$

$f(n) = \frac{1}{\log_b a} \cdot \log_b n$, $c = \frac{1}{\log_b a}$

$n_0 = 1 \Rightarrow g(n) = O(\log_b n)$

(2) Polynomial Rule: Given $a > b \geq 0, n^a$ grows faster than n^b

- $\lim_{n\to\infty} \frac{n^b}{n^a} \to 0$

If $g(n)$ is a non-negative polynomial of degree **d**, i.e., $g(n) = a_d \cdot n^d + a_{d-1} \cdot n^{d-1} + \cdots + a_1 \cdot n + a_0$, then $g(n)$ is $O(n^d)$.

$g(n) \leq n^d(|a_d| + \cdots + |a_0|)$ $(x \leq |x|)$ with $n \geq 1$

(3) ▸ **Product property** sum of two power.

- $g_1(n) = O(f_1(n)), g_2(n) = O(f_2(n))$
  - Then, $g_1(n) \cdot g_2(n) = O(f_1(n) \cdot f_2(n))$
- $g(n) = (n^3 + n^2 + 1)(n^7 + n^{12})$
- What is the time complexity of $g(n)$?
  - Let $g_1(n) = (n^3 + n^2 + 1)$, $g_2(n) = (n^7 + n^{12})$
    - $g_1(n) = O(n^3), g_2(n) = O(n^{12})$
    - $g(n) = g_1(n) \cdot g_2(n) = O(n^3 \cdot n^{12}) = O(n^{15})$

(4) ▸ **Sum property** The bigger of the two big

- $g_1(n) = O(f_1(n)), g_2(n) = O(f_2(n))$
  - Then, $g_1(n) + g_2(n) = O(\max(f_1(n), f_2(n)))$

(5) log:

① Smaller than power.

▸ Log functions grow slower than power functions

- $\lim_{n\to\infty} \frac{(\log n)^a}{n^b} \to 0$, for $a, b > 0$

▸ If $g(n) = (\log n)^a + n^b$ where $a, b > 0$, compare base ⇒ $\log n$ and n (same type; log and power.)
- Then $g(n) = O(n^b)$.

▸ If $g(n) = (\log n)^a + b \cdot (\log n)^x$, where $b > 0, a > x \geq 0$ (same base ⇒ compare power.)
- Then $g(n) = O((\log n)^a)$. power ☆

exp
power
log
⇒ Compare type ⇒ Compare base ⇒ compare power.

② log only faster than constant.

(6) Exp: faster than power

▸ Exponential functions grow faster than power functions

- $\lim_{n\to\infty} \frac{n^k}{a^n} \to 0$ if $a > 1$ for any k

▸ If $g(n) = a^n + n^b$, where $a > 1$ (Compare type)
- Then $g(n) = O(a^n)$

▸ If $g(n) = a^n + b^n$, where $a > b > 1$ (same type ⇒ compare base.)
- Then $g(n) = O(a^n)$

eg: $g(n) = 2^n + 3^n$

- $O(3^n)$, why? (for $c = 2, n_0 = 1$ $g(n) = 2^n + 3^n \leq 3^n + 3^n$ for $n_0 \geq 1$)

$g(n) = 2^n + 3^n \leq c \cdot 3^n$

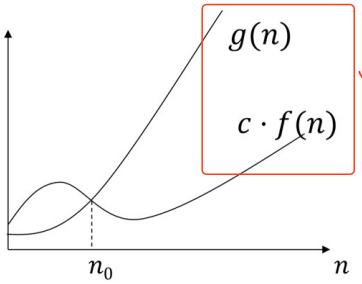
$= 2 \cdot 3^n \Rightarrow c = 2$ $f(n) = 3^n$

$(c-1)3^n \geq 2^n \Rightarrow c \geq 2$.

4. Big Omega:

(1) Def: ▸ **Big-Omega definition:**

- $g(n) = \Omega(f(n))$ if and only if there exist positive constants c and n_0 such that $g(n) \geq c \cdot f(n)$ for all $n \geq n_0$ ⇒ Find c and n_0.
  - $g(n)$ grows asymptotically no slower than $f(n)$
  - $f(n)$ is usually much simpler than $g(n)$



lower bound
Can be infinite.

☆ limit form of big O:

One way to understand Big O using limits is to look at the limit of the ratio between $g(n)$ and $f(n)$ as $n \to \infty$:

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

⇒ $g(n) = O(f(n))$

- If L is a **finite constant** (i.e., not infinity), then $g(n)$ is asymptotically bounded by $f(n)$ up to constant factors.
- If L is **zero**, this means that $g(n)$ grows slower than $f(n)$, so $g(n) = O(f(n))$.
- If L is **infinity**, this means that $g(n)$ grows faster than $f(n)$, and so $g(n)$ is not $O(f(n))$.

① For linear search: $g(n) = 4n + 4$, prove $g(n) = \Omega(n)$

Proof: We can find $c = 4$, $n_0 = 1$ such that
$g(n) = 4n + 4 \geq 4n$ for $n \geq 1$. Cancel 低 power 的项.
Therefore $g(n) = \Omega(n)$.

② $g(n) = 2n$, prove that $g(n) \neq \Omega(n^2)$

Proof: If $g(n) = \Omega(n^2)$, then we need to find a positive constant c and n_0 such that
$g(n) = 2n \geq cn^2$ for all $n \geq n_0$.
That is to say we need to guarantee that:
$2 \geq c \cdot n$ for $n \geq n_0$.
However, c is a constant, and the above inequality does not always hold, contradiction. Therefore, $g(n) \neq \Omega(n^2)$.

using contradiction.

(2) If $g(n) = \Omega(f(n))$, then $f(n) = O(g(n))$
If $g(n) = O(f(n))$, then $f(n) = \Omega(g(n))$

Proof: We prove the first part. The second part can be proved in a similar way.
According to the definition, we know that there exist constants positive constants c, n_0, such that
That is to say: $g(n) \geq c \cdot f(n)$ for $n \geq n_0$. 保证 $c > 0$
$f(n) \leq 1/c \cdot g(n)$ for $n \geq n_0$.
We find a constant $c' = 1/c, n_0' = n_0$ such that $f(n) \leq c' \cdot g(n)$ for $n \geq n_0'$.
Therefore, $f(n) = O(g(n))$ according to the definition.

(3) Rules:

- Rule (i): The polynomial rule (the biggest eats all)
  - $g(n) = n^3 + n^2 + n$
  - n^3 has the biggest power, it eats all other terms, so $g(n) = \Omega(n^3)$
- Rule (ii): Product property (the big multiplies the big)
  - $g(n) = (n + 1) \cdot (n + n^5)$
  - The first big $\Omega(n)$; the second big $\Omega(n^5)$, so $g(n) = \Omega(n \cdot n^5) = \Omega(n^6)$
- Rule (iii): Sum property (the bigger of the two big)
  - $g(n) = (n + 1) + (n + n^5)$
  - The first big $\Omega(n)$; the second big $\Omega(n^5)$, so $g(n) = \Omega(\max(n, n^5)) = \Omega(n^5)$
- Rule (iv): log only beats constant
  - $g(n) = n^{0.000001} + (\log n)^{1000}$
    - $\Omega(n^{0.000001})$
  - $g(n) = 0.00000001 \cdot \log n + 100000$
    - $\Omega(\log n)$
- Rule (v): exponential beats powers
  - $g(n) = 1.00001^n + n^{999999}$
    - $\Omega(1.00001^n)$
  - $g(n) = 3^n + 4^n$
    - $\Omega(4^n)$

5. Big Theta:

(1) Def:

- We use Big-Theta (Θ) to make the growth-rate tight
  - If $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$
  - Then, $g(n) = \Theta(f(n))$
    - $g(n) = O(f(n))$: $g(n)$ grows asymptotically no faster than $f(n)$
    - $g(n) = \Omega(f(n))$: $g(n)$ grows asymptotically no slower than $f(n)$
    - If $g(n) = \Theta(f(n))$, it means the growth rate of $g(n)$ is asymptotically the same as $f(n)$

eg: ① $g(n) = 3n+4$
$g(n) \geq 3n = \Omega(n)$
$g(n) \leq 7n = O(n)$
$\Rightarrow g(n) = \Theta(n)$

② $g(n) = (3n^2 + \sqrt{n}) \cdot (n \log n + n)$
用 product rule 与 sum rule.
Ω: $\Omega(n^2) \cdot \Omega(n \log n) = \Omega(n^3 \log n)$
O: $O(n^2) \cdot O(n \log n) = O(n^3 \log n)$
$\Rightarrow g(n) = \Theta(n^3 \log n)$.

6. Analysis for complexity:

(1) If two algorithms have the same Big-Theta function, they can be considered as equally good
  - $g_1(n) = 10000n = \Theta(n)$ and $g_2(n) = 9n = \Theta(n)$

(2) An algorithm with the slower growth rate $\Theta(f(n))$ than others is asymptotically faster than other algorithms (solving the same problem)
  - An algorithm with $\Theta(\log n)$ time complexity is asymptotically better than the one with $\Theta(n)$ since the former grows slower than the latter

Divide and Conquer (complexity for recursion)

1. Binary Search:

```
BinarySearch(arr, searchnum, left, right)
1  if left >= right                                          O(1)
2      if arr[left] == searchnum                             O(1)
3          return left                                       O(1)
4      else                                                  O(1)
5          return -1                                         O(1)
6  middle = (left + right)/2                                 O(1)
7  if arr[middle] == searchnum                               O(1)
8      return middle                                         O(1)
9  elseif arr[middle] < searchnum                            O(1)
10     return BinarySearch(arr, searchnum, middle+1, right)  O(?)
11 else
12     return BinarySearch(arr, searchnum, left, middle -1)  O(?)
```

⇒ 二选一来 recurse → $g(\frac{n}{2})$

$g(n) = g(\frac{n}{2}) + a$

① $n = 2^x$ 正好可以对半分

$g(n) = g(\frac{n}{2}) + a$

$= g(\frac{n}{2^2}) + a + a$

$= \cdots$ ($\frac{n}{2^x} = 1 \Rightarrow x = \log n$)

$= g(1) + ax$

$= a \cdot \log_2 n + b$

② $n \neq 2^x$ using a upper bound extend to symmetric

n ⇒ let $n' = 2^x$

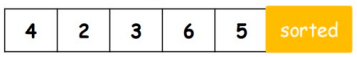
$\Rightarrow n \le n' = 2x$

$\Rightarrow n' \le 2n$ ☆

$g(n) \le g(2^x) = a \cdot x + b \le g(2n) = a \cdot \log_2(2n) + b = a \cdot \log_2 n + a + b$ ☆

这里 $n' = 2^x$ 为用上 even case 的公式.

2. Selection Sort:



从后向前遍历.

$g(n) = g(n-1) + O(n)$

$\le g(n-1) + cn \le g(n-2) + cn + c(n-1) \le \cdots \le g(1) + c \cdot [n + (n-1) + \cdots + 2]$

$\le c \cdot [n + (n-1) + \cdots + 1] + b$ ⓑ $g(1) = \frac{c}{2}(n+1)n + b = O(n^2)$

scaling

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

☆ $\sum_{k=1}^{n} k^2 = \frac{k(k+1)(2k+1)}{6}$

3.

```
maxInArray2(arr, left, right)
1  if left == right                                  O(1)
2    return arr[left]                                O(1)
3  else                                              O(1)
4     mid = (left+right)/2                           O(1)
5     maxLeft = maxInArray2(arr, left,middle)        ? g(n/2)
6     maxRight = maxInArray2(arr,middle+1,right)     ? g(n/2)
7     return max(maxLeft, maxRight)                  O(1)
```

⇒ 同时 recurse → $2g(\frac{n}{2})$

$g(n) = 2g(\frac{n}{2}) + O(1)$ 把 $O(1)$ 换回 function ☆

① $n = 2^x$: $g(n) \le 2g(\frac{n}{2}) + c \le 2(2g(\frac{n}{2^2}) + c) + c \le \cdots \le 2^x \cdot g(1) + c + 2c + \cdots + 2^{x-1} \cdot c$

$= 4g(\frac{n}{2^2}) + 2c + c$

$x = \log_2 n$, $g(1) = b$

$\Rightarrow g(n) \le bn + cn - c$

☆ geometric series:

$c(1 + 2 + 4 + \cdots + 2^{x-1})$

$= c \cdot \frac{2^x - 1}{2 - 1} = c(2^x - 1)$

$= cn - c$

② $n \neq 2^x$, let $n' = 2^x$, $n' \le 2n$,

$g(n) \le g(n') = g(2^x) = bn + cn - c \le g(2n) = 2bn + 2cn - c$

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

4. Master's Theorem:

- Let $T(n)$ be the running cost depending on the input size n, and we have its recurrence:
  - $T(1) = O(1)$
  - $T(n) \le a \cdot T(n/b) + O(n^d)$
  - a, b, d are constants such that $a \ge 1, b > 1, d \ge 0$. Then,

- a is the number of recursive calls,
- b is the factor by which the problem size shrinks in each call,
- $O(n^d)$ is the cost of dividing and merging the problem.

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

Note that n/b may not be an integer, but it will not affect the asymptotic behavior of recurrence

a and b^d <=> $\log_b a$ and d

1. **Case 1:** $a = b^d \rightarrow O(n^d \log n)$
   - The work done at each level is balanced across all levels.
   - The recurrence tree has $\log_b n$ levels, and each level contributes equal work $O(n^d)$. $= O(\log n)$
2. **Case 2:** $a < b^d \rightarrow O(n^d)$
   - The work done outside recursion (the $O(n^d)$ term) dominates.
   - The recursive calls shrink quickly, leading to fewer levels in the recursion tree.

Case 3: $a > b^d \rightarrow O(n^{\log_b a})$

- The recursion dominates because each recursive call contributes significant work.
- The number of recursive calls grows exponentially.

eg:
- $g(1) = c_0, g(n) \le g(n/2) + c$ $O(1)$
  - We have that $a = 1, b = 2, d = 0$
  - Since $\log_b a = d$, we know that: $g(n) = O(n^0 \cdot \log n) = O(\log n)$
- $g(1) = c_0, g(n) \le g(n/2) + c_1 \cdot n$ $O(n)$
  - We have that $a = 1, b = 2, d = 1$
  - Since $\log_b a < d$, we know that: $g(n) = O(n^d) = O(n)$
- $g(1) = c_0, g(n) \le 2 \cdot g(n/2) + c_1 \cdot n^{0.5}$ $O(n^{0.5})$
  - We have that $a = 2, b = 2, d = 0.5$
  - Since $\log_b a > d$, we have that: $g(n) = O(n^{\log_b a}) = O(n)$
- $g(1) = c_0, g(n) \le 2 \cdot g(n/4) + c_1 \cdot \sqrt{n}$ $O(n^{0.5})$
  - We have $a = 2, b = 4, d = 0.5$
  - Since $\log_b a = d$, we have that: $g(n) = O(n^d \cdot \log n) = O(\sqrt{n} \cdot \log n)$

eg: ① 奇偶版: ② $\sum_{k=1}^{n} k^2 = \frac{k(k+1)(2k+1)}{6}$

$T(n) = T(n-2) + n^2$
$= T(n-4) + (n-2)^2 + n^2$
$= \cdots$
$= T(1) + 3^2 + \cdots + (n-2)^2 + n^2$ (if n is odd)
$= T(2) + 4^2 + \cdots + (n-2)^2 + n^2$ (if n is even)

=> If even:

let $n = 2k$:

$T(n) = (2\times1)^2 + (2\times2)^2 + \cdots + (2k)^2$
$= 2^2(1^2 + 2^2 + \cdots + k^2) = 4 \times \frac{k(k+1)(2k+1)}{6}$
$= \frac{n(n+2)(n+1)}{6} = O(n^3)$

=> If odd:

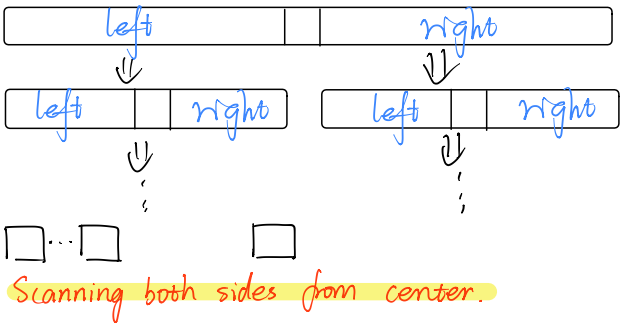
$T(n) = (1^2 + 2^2 + 3^2 + \cdots + n^2) - (2^2 + 4^2 + \cdots + (n-1)^2)$

$\sum_{k=1}^{n} k^2$

5. Maximum Subarray:

(1) Brute Force: iterate all the possible outcome

(2) Divide-and-Conquer



Scanning both sides from center.

```
MaxCrossSubarray(A, i, k, j)
  left_sum = -∞                          O(k-i+1)
  sum=0                                  ┐
  for p = k downto i                     │
    sum = sum + A[p]                     │
    if sum > left_sum                    │
      left_sum = sum                     │
      max_left = p                       │
                                         ├ = O(j-i+1)
  right_sum = -∞                         │
  sum=0                                  │
  for q = k+1 to j                       │
    sum = sum + A[q]                     │
    if sum > right_sum                   │
      right_sum = sum                    │
      max_right = q                      ┘
                                         O(j-k)
  return (max_left, max_right, left_sum + right_sum)
- MaxSubarray(A, i, j)
  if i == j // base case
    return (i, j, A[i])
  else // recursive case
    k = floor((i + j) / 2)
Divide (l_low, l_high, l_sum) = MaxSubarray(A, i, k)          T(k-i+1)   <- 进入 recursion 的 element 数量.
       (r_low, r_high, r_sum) = MaxSubarray(A, k+1, j)        T(j-k)   Conquer
       (c_low, c_high, c_sum) = MaxCrossSubarray(A, i, k, j)

  if l_sum >= r_sum and l_sum >= c_sum // case 1
    return (l_low, l_high, l_sum)
  else if r_sum >= l_sum and r_sum >= c_sum // case 2   Combine
    return (r_low, r_high, r_sum)
  else // case 3
    return (c_low, c_high, c_sum)
```

Maxsubarray 被放进 Stack

1st: Implement Divide

Maxsubarray(), 把 input 的 array 全拆成单个的

(Implement base case)

3rd: (l_low, l_high, l_sum) (r_low, r_high, r_sum) (c_low, c_high, c_sum) 根据 Stack 的 order 从上向下 return combine

Array:

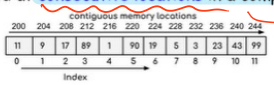
- An array stores the same type of objects together, and we access the objects by their indices

linear.

(1) ☆ drawback: { fixed space
insert/delete object is hard ⇒ ① if insert within two objects, change 左右两边的index.
② 可能出现把原objects挤出

(2) 特点:

An array is a linear data structure that hosts a collection of data elements stored at consecutive locations in a computer's memory



→ easy to calculate address.

- The individual values are called elements; all the elements are of the same type
- The number of elements is called the length of the array, which is fixed when the array is created
- Each element is identified by its position in the array, which is called index (In C/C++/Java, the index numbers begin with 0)

But Arraylist is mutable

(3) Creat array:

- An array is characterized by
  - Element type
  - ☆ Length: type[ ] identifier = new type[length];

```
int[] numbers = new int[5];
```

Traverse Array:

```
for (int i = 0; i < array.length; i++)

for (int value:array)
```

(4) Initialization:

- A convenient way of initializing an array:

```
int[ ] digits = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
String[ ] US_CITIES_OVER_ONE_MILLION = {
   "New York",
   "Los Angeles",
   "Chicago",
   "Huston",
   "Phoenix",
}
```

⇒ directly gives inner value.

2. Operations:

- createArray(n)
  - Initialized an array of size n
- retrieve(arr, i)
  - arr[i]
  - Return the item stored in the i-th position of the array

→ index problem:

- store(arr, i, itemToStore)
  - Store itemToStore to the i-th position of the array
  - arr[i] = itemToStore

- Starting index numbering at 0 can be confusing, so we use two standard ways:
  - Use Java's index number internally, and then add one when presenting to the user

```
for (int i = 0; i < arr.length; i++) {
    System.out.println("Element " + (i + 1) + ": " + arr[i]);
}
```

  - Use index values beginning at 1, and ignore the first (0) element in each array

```
int[] arr = new int[6]; // Size is 6, but only use arr[1] to arr[5]
☆
arr[1] = 10;
arr[2] = 20;
arr[3] = 30;
arr[4] = 40;
arr[5] = 50;

// Now, ignore arr[0]
for (int i = 1; i <= 5; i++) {
    System.out.println("Element " + i + ": " + arr[i]);
}
```

3. Algorithm using array:

(1)

Algorithm 1: *Linear Search*
Input: An array a of integers with length n, an integer searchnum
Output: the index i such that the value stored in the i-th position equals searchnum, or -1 if no such i exists

```
1 int i;
2 for (i=0; i<n; i++){
3     if( retrieve(a,i) == searchnum )
4         return i;
5 }
6 return -1;
```

(2) dot product:

Input: Vector a_1, a_2, a_3 of length n
Output: Dimension-product of a_1 and a_2 which is stored in a_3

```
1 int i,i_dimension_product;
2 for (i=0; i<n; i++){
3     i_dimension_product = retrieve(a1,i)*retrieve(a2,i);
4     store(a3, i, i_dimension_product);
5 }
6 return a3;
```

4. Pass by value or reference:

(1) By value:

- A copy of the variable is passed to the function.
- Changes made inside the function do not affect the original variable.

```
public class Main {
    public static void modifyValue(int num) {
        num = num + 5;  // Modifying the local copy of num
        System.out.println("Inside modifyValue: " + num);
    }

    public static void main(String[] args) {
        int x = 10;
        System.out.println("Before function call: " + x);
        modifyValue(x);
        System.out.println("After function call: " + x);  // Original variable is unchanged
    }
}
```

create a new num inside function.

一般: { int, float, char

{ Array of objects (reference): ☆ String[]
Array of primitive value: { int[], boolean[], float[], char[]

(2) Reference:

- The function operates directly on the original variable (via its memory address).
- Changes made inside the function will affect the original variable.

① arrays' append; (modification)

array
list
class instance.

② class :

```
public class ReferenceExample {
    public static void main(String[] args) {
        Person p1 = new Person("Alice");
        modifyName(p1);
        System.out.println(p1.name); // 输出 "Bob"，因为对象的内容被修改
        p1 = new Person("Charlie"); // 修改引用指向新的对象
        modifyReference(p1);
        System.out.println(p1.name); // 输出 "Charlie"，引用没有被修改
    }
    public static void modifyName(Person person) {
        person.name = "Bob"; // 修改对象内容
    }
    public static void modifyReference(Person person) {
        person = new Person("David"); // 重新赋值不会影响原始引用
    }
}
```

5. 2D array: E.g., a 3-by-2 matrix can be represented by a two-dimensional array:

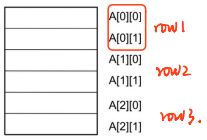
```
int[][] A = new int[3][2];
int A[3][2] = {{1, 4}, {2, 5}, {3, 6}};
```

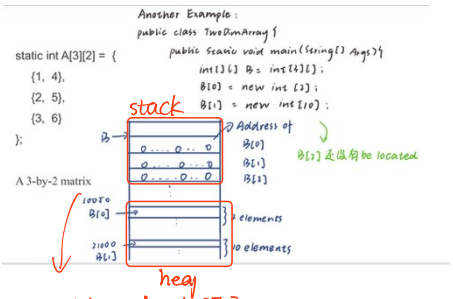
row column

{...} store row data

storage: 3x2x4 (4 bytes) — m x n size of data

Memory allocation (row orientation)





Address for A[i][j]:

① A[i]: m x i x b ⇒ 用于存第i ith row

A[i][j]: j x b. ⇒ locate j th entry in i th row.

Also object

A[0][0]: 1000 ⇒ initial address.

6. Store element when out of maximum of index.

1. Insert into array when over size:

eg: A[] with size n.

create: int[] newArray = new int[n+1]

copy: from 0 to n: newArray[i] = A[i].

append: Array[n+1] = x.

```
public static void main(String[] args)
{
    //original array
    String[] colorsArray = {"Red", "Green", "Blue" };
    System.out.println("Original Array: " + Arrays.toString(colorsArray));

    //length of original array
    int orig_length = colorsArray.length;
    //new element to be added to string array
    String newElement = "Orange";
    //define new array with length more than the original array
    String[] newArray = new String[ orig_length + 1 ];
    //add all elements of original array to new array
    for (int i = 0; i < colorsArray.length; i++)
    {
        newArray[i] = colorsArray [i];
    }
    //add new element to the end of new array
    newArray[newArray.length - 1] = newElement;
    //make new array as original array and print it
    colorsArray = newArray;
    System.out.println("Array after adding new item: " + Arrays.toString(colorsArray));
}
```

copy 原本的

2. Delete element

Iterate to find x with index i

Shift A[i+1 ... n-1] one unit to left.

Reduce size.

1. Operations of polynomial.

(1) Single variable: $F = \sum_{i=0}^{N} A_i x^i$ ⇒

```
class Poly:
    int coef[MaxDegree+1]
    int highPower.
```

constant : power 0.

$f_1 = \sum_{i=1}^{N} A_i x^i$

$f_2 = \sum_{i=1}^{M} B_i x^i$

i. polynomialArray { coef; maxPower.

polynomialArray.length = max{maxPower$_1$, maxPower$_2$} + 1.

initialization

ii. ZeroPoly with length = polynomialArray.length

① AdditionArray:

Function zeroPoly(Poly, MaxDegree)

for j from 0 to MaxDegree − 1 then:

Poly.coef[j] = 0.

Poly.highPower = 0.

Function Addition(Poly1, Poly2, PolySum)

zeroPoly(PolySum)

highPower = max{Poly1.highPower, Poly2.highPower}

for i from 0 to highPower do:

PolySum.coef[i] = Poly1.coef[i] + Poly2.coef[i]

end for

② Multiplication:

```
for (int i = 0; i <= poly1.highPower; i++)
    for (int j = 0; j <= poly2.highPower; j++)
      polyProd.coeffArray[i + j] += poly1.coeffArray[i] *
                                    poly2.coeffArray[j];
```

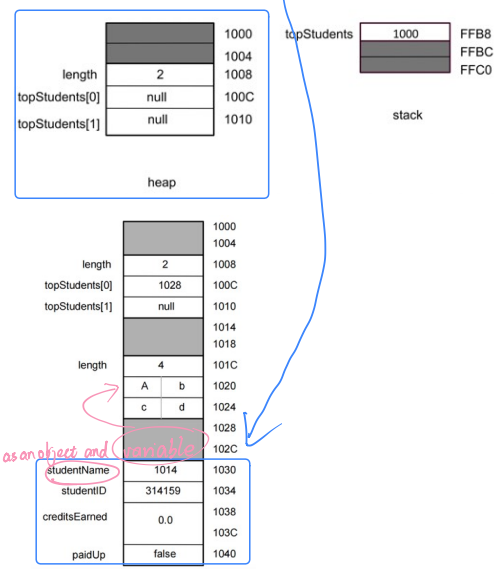
index → power

guarantee that result array index → power and coef is final coef.

☆ => should spend a lot space for 0.

two objects { top…[0] top…[1]

```
Student[ ] topStudents = new Student[2];
topStudents[0] = new Student("Abcd", 314159);
```



☆ 字母频率: ☆

```
public class LetterFrequency {
    public static void main(String[] args) {
        String text = "Hello World! This is an example text.";

        int[] letterCounts = new int[26];
        for (int i = 0; i < text.length(); i++) {
            char ch = text.charAt(i);
            // Check if the character is a letter (A-Z or a-z)
            if (Character.isLetter(ch)) {
                // Convert character to uppercase and calculate its index
                int index = Character.toUpperCase(ch) - 'A';

                letterCounts[index]++;
            }
        }
    }
}
```

Linked List:

☆ but arrays is ⇒ Array queue and stack also but linked list implemented are not.

Singly linked list:

☆ Not contiguous ⇒ Need pointer to link.

head → N1 → N2 → null.

N1 is set as head defaultly

1. Node: { element, pointer next }

```
public class Node{
    public int element;
    public Node next;

    public Node(int x){
        element = x;
        next = null;
    }
}
```

→ More cost than array: { Node, pointer. }

Why insertion is faster in linked list?

- Linked List's each element maintains two pointers (addresses) which points to the both neighbor elements in the list

Does linked list have index?

- It's important to mention that, unlike an array, linked lists do not have built-in indexes.
- In order to find a specific point in the linked list, you need to start at the beginning and traverse through each node, one by one, until you find what you're looking for.

不储存在Node里面. index 作 local var ⇒ delete, insert 后 index 会变

(1) Advantage:
- Dynamic structure
- Memory efficient: size is dynamic
- Insert and delete easier
  - 不用 move 左右的 element ☆ (不care index)

Disadvantage:
- Memory usage
- Traversal
- Reverse traversing.

2. Operations:

① Traversal: using a pointer to iterate from head to null.
pointer is also a node

☆ Need known head to keep tracking
☆ No need adjacent in memory

```
public class Traversal {
    public static void traverseList(Node head){
        Node current = head;

        while (current != null){
            current = current.next;
        }
    }
}
```

☆ Check if empty ☆

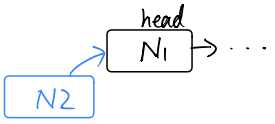
② Searching:

```
Function Search(head, x):
    cur ← head
    While cur != null do:
        if cur.element = x do:
            return true
        end if
        cur = cur.next.
    end while
```

```
public class Searching{
    public static boolean search(Node head, int x){
        Node current = head;
        while (current != null){
            if (current.element == x){
                return true;
            }
            current = current.next;
        }
        return false;
    }
}
```

② Insert:

i. At beginning:

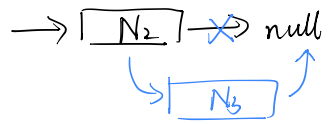


① Node N2 = new Node(x);
② N2.next = N1
③ head = N2

Change pointer → Change head

```
public class InsertAtBeginning {
    public static void Insert(Node head, int x){
        Node newNode = new Node(x);
        newNode.next = head;
        newNode = head;
    }
}
```

ii. At end:



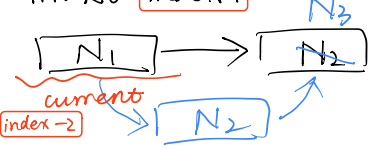
Traversal to end → insert.

```
Node cur = head
a. If (cur is null):
       head = newNode
b. while (cur.next != null):
       cur ← cur.next.

newNode.next ← null.
cur.next ← new Node.
```

iii. At index:

☆ 注: start at 0 ☆

Find index−2

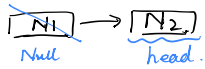


☆ Check if out of bound.

```
newNode ← Node(x)
cur ← head
count ← 0
While cur != null and count < ind-1 do:
        cur ← cur.next
        count += 1
end while
if cur = null then:
    out of bound
end if
newNode.next ← cur.next
cur.next ← newNode
```

⑤ Delet:

i. Beginning:




```
Node tmp = head.
head = head.next.
tmp = null.
```

```
public static Node deleteBegin(Node head) {
    if (head == null) {
        return null;
    }
    head = head.next;
    return head;
}
```

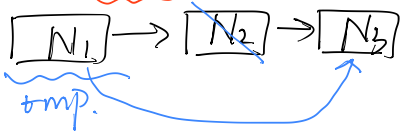
traverse to the last two

ii. End:


```
Node tmp = head.
temp → Nm-1
temp.next = null
```

```
public static Node deleteEnd(Node head){
    Node tmp = head;
    while (tmp.next != null){
        tmp = tmp.next;
    }
    tmp.next = null;
    return tmp;
}
```

iii. At index:



=

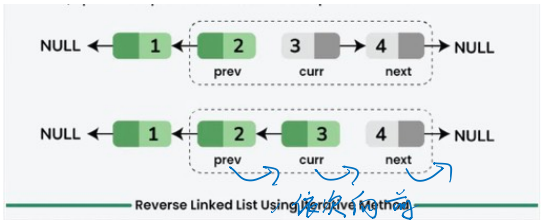
```
Node tmp = head.
temp → Nindex-1
temp.next = temp.next.next.
```

```
public static Node deleteIndex(Node head, int index) {
    Node current = head;
    int count = 0;
    while (count < index - 1 && current != null) {
        current = current.next;
        count++;
    }
    current.next = current.next.next;
    return head;
}
```

改用 pre 与 cur

pre.next = cur.next

⑤ Reverse:




```
public static Node reverse(Node head){
    Node pre = null;
    Node cur = head;
    Node next = head.next;
    while (cur != null){
        cur.next = pre;
        pre = cur;
        cur = next;
        if (next != null){
            next = next.next;
        }
    }
    return cur;
}
```

head → 1 → 2 → 3 → 4 → 5 → ⋯

pre cur next @next

① pre ② cur

Node { pre = null / cur = head / next = head.next.

一边反转一边 reverse

iterate next to last node:

① Reuse pointer

② update node.

cur.next = pre. reverse

```
pre = cur
cur = next
next = next.next
```

反转.

⑥ Middle:

Function Middle(head):

Node slow, fast

while fast != null and fast.next != null and fast.next.next != null do ☆

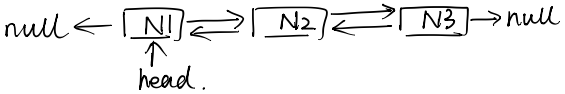
slow ← slow.next

fast ← fast.next.next

end while.

return slow.

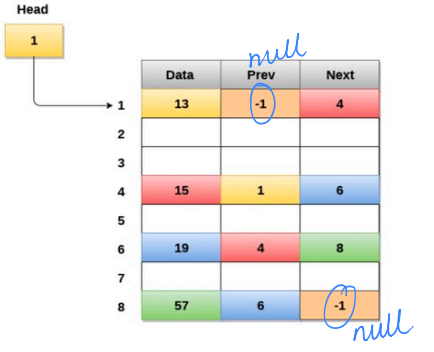
Doubly linked list:



① Node:

```
public class Node {
    public int data;
    public Node next;
    public Node prev;

    public Node(int data) {
        this.data = data;
        this.next = null;
        this.prev = null;
    }
}
```

表用 table 标指针的变化.



Memory Representation of a Doubly linked list

② Traversal:

Node cur. = head.

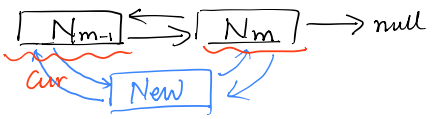
Iterate cur:

cur = cur.next

⑥ Insert:

i. At beginning:

ii. At index.



Traverse to N_{m-1} as cur

new. pre = cur

new. next = cur. next

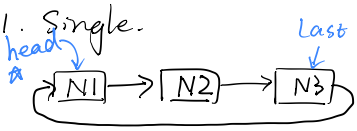
先把两头的接上.

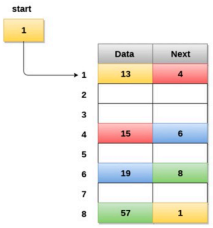
cur. next. pre = new

cur. next = pre.

Circularly linked list. ⇒ no starting and ending Node ⇒ traverse in any direction.

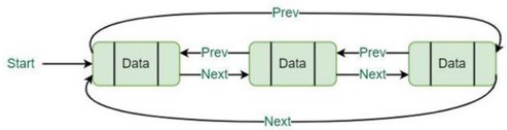
1. Single.





Memory Representation of a circular linked list

2. Double:



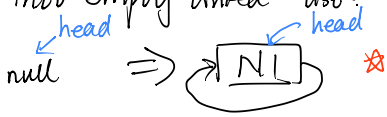
① Node:

int data

Node next.

② Insert:

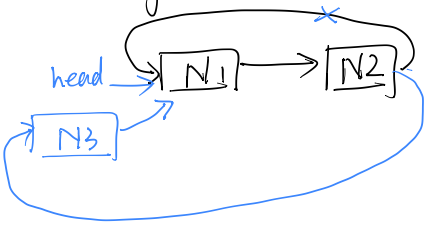
i. into empty linked list:



new. next = new.

head = new

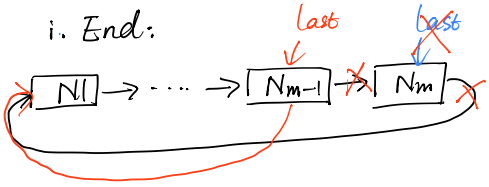
ii. Begin.



Traverse to the end: while cur. next. next ! = head

③ Delete:

i. End:



Applications:

① Addition Linked List: 不用0占位. 节省空间, 省略零的 operation.

f_A: | C_{A1} | P_{A1} | → | C_{A2} | P_{A2} | → ··· → | C_{Am} | P_{Am} | → null

f_B: | C_{B1} | P_{B1} | → | C_{B2} | P_{B2} | → ··· → | C_{Bn} | P_{Bn} | → null

may not same length.

Compare according to order. using merge method:

class PolyLL:

int power

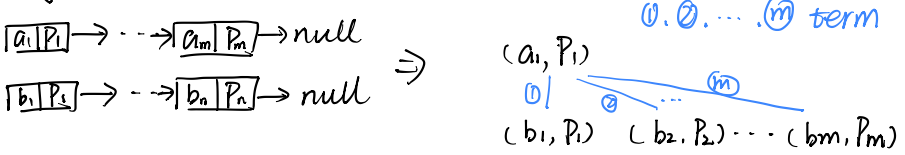
int coef

PolyLL next

```
function AddLL(F1, F2, Results):
While (F1 != null or F2 != null):
    If (F1 is null):
        Insert F2 to Results term by term
    Elif (F2 is null):
        Insert F1 to Results term by term
    Else(F1 and F2 != null):
        If F1.power > F2.power:
            add F1.coef and power.
            move F1 to next.
        If F2.power > F1.power:
            add F2 coef.
            move F2 to next.
        Elif F1.power = F2.power:
            F1.coef + F2.coef .
            move F1 and F2 to next term.
    Return Results.
```

③ Multiply LL:

Logic 1: Term by term product, then insert to Results.



① ② ... (m) term by term insert to Results.

```
multiplyLL(Polynomial F1, Polynomial F2):
Output: product of F1 and F2
Results ← null
for term P1 in F1 do:
    for term P2 in F2 do:
        coef ← P1.coef × P2.coef.
        Power ← P1.Power + P2.Power.
        Results ← addTerms(Results, Coef, Power)
    End for
End for
Return Results
```

```
function addTerms(Polynomial Results, int Power, int coef):
① newTerm ← Polynomial(Power, coef)
  If Results is null then:
      return newTerm.
  End if
```

info class of newTerm

⇒ Results null, then return newTerm as 1st node

②

```
temp ← result
prev ← null
while temp is not null and temp.degree > degree do
    prev ← temp
    temp ← temp.next
end while
if temp is not null and temp.degree == degree then
    temp.coeff ← temp.coeff + coeff
else
    if prev is null then
        newTerm.next ← result
        return newTerm
    else
        prev.next ← newTerm
        newTerm.next ← temp
    end if
end if
return result
```

⇒ Not null: Traverse and compare: using prev and cur to track

- new Power is the largest → begin ⇒ prev = null → newTerm.next = Results
- within the Results's Powers → insert ⇒ prev.next ← newTerm. newTerm.next ← temp.
- same Power : temp.coef += coef.

logic 2: $[a_1|P_1] \to \cdots \to [a_m|P_m] \to null$

$[b_1|P_1] \to \cdots \to [b_n|P_n] \to null$

$\Rightarrow$

RLL_1 RLL_2 RLL_m $\Rightarrow$ merge m LL

(a_1, P_1) $(a_2, P_2) \cdots$ (a_m, P_m)

(b_1, P_1) $(b_2, P_2) \cdots (b_m, P_m)$

Insert term by term to a null LL.

```
Function multiplyPoly(Poly1, Poly2):
Output: Result.
Result ← null.
for term P1 in Poly1 do:
    Partial ← null
    for term P2 in Poly2 do:
        Coef ← P1.coef × P2.coef.
        Power ← P1.Power × P2.Power.
        New = Polynomial(Power, Coef)
        Append New to Partial
    End for.
Result = Merge(Result, Partial)
End for
Return Result
```

Function Merge(Result, Partialproduct):
using Function addTerms in ⑤

2. Matrix Representation:

```
Node: int row
      int colum
      int value
      Node right
      Node down
```

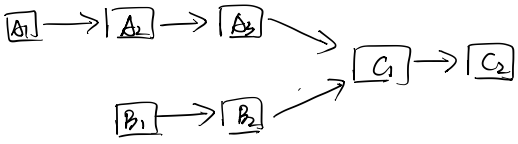
Initialize matrixLL: → assuming row, rank continuous

head = Node(-1, -1, 0)

Node Arrays for row and rank ⇒ size of matrix.

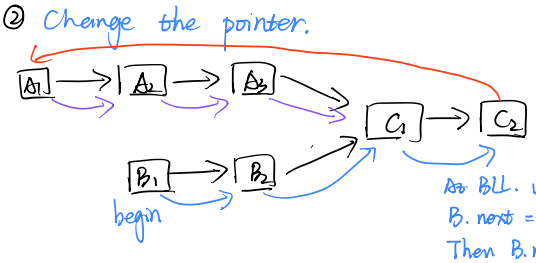
iterate col and row set value as 0.

3. linkedLL Find intersection:

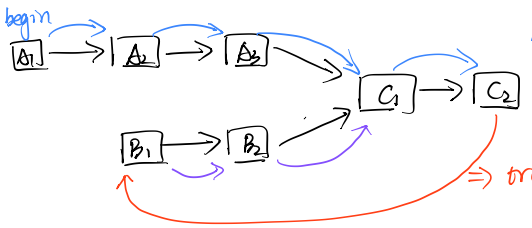


① Interate each Node. ⇒ $O(M) \times O(N) = O(MN)$

② Change the pointer.



As BLL. when at C_2
B.next = null
Then B.next = A.head.



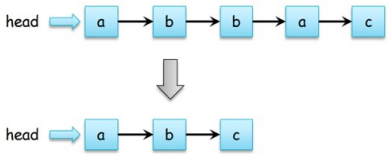
As ALL. when at C_2
A.next = null
Then A.next = B.head.

```
A = L1.head
B = L2.head
while TRUE
        if A == B
                return A
        if A.next == NULL
                A = L2.head
        else A = A.next
        if B.next == NULL
                B = L1.head
        else B = B.next
```

⇒ traverse 添加 pointer 改成 doubly circular linked list ⇒ check if exists cycling

两条 traverse 反复直到 A == B，且 traverse 相同总数的 Nodes

4. Linked List find duplicate items: (traverse then delete)



① 逐个比对: 用两个 Node

```
Node pre, B
if B.data = A.data then:
    pre.next = B.next.
else:
    pre = B
    B = B.next
end if
```

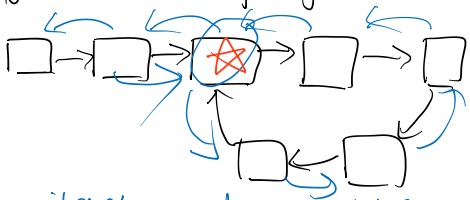
② 用 boolen array capture if repeated.

```
while A != null
    dataA = A.data
    index = dataA - 'a'
    if b[index] == false
        b[index] = true
        pre = A      pre = L.head
    else      ⇒ share address of L
        pre.next = A.next
```

用 pre 的 pointer 对 L 删改.

5. Detect Cycle:

① 用 reverse LL: if cycle next 会指向 pre

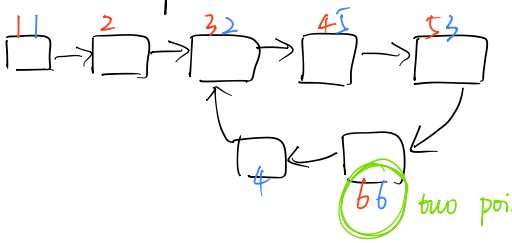


iterate ⇒ change pointer

If cur = cur.next.next.

Retur True.

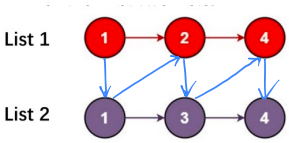
② fast-slow pointer.



two pointer meet. ⇒ cycle ✓

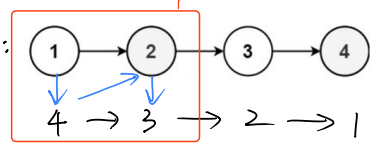
6. Merge two linked list

Compare → change pointer.



using merge sort logic

7. Reorder list:



reverse: 把数据读入 stack

If data(Node) = data(Node.next) then: break

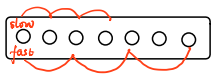
but Reverse latter n//2 is enough

Linked list merge sort implementation:

① GetMid: using pointer fast and slow.

fast: 2 steps as 1 unit

slow: 1 step as 1 unit.




```
public ListNode getMiddle(ListNode head) {
    if (head == null) return null;
    ListNode slow = head;
    ListNode fast = head;
    while (fast != null && fast.next != null) {
        slow = slow.next; // slow moves one step
        fast = fast.next.next; // fast moves two steps
    }
    return slow;
}
```

② Merge: Input Node head to traversal, then GetMid to spilt.

```
public ListNode mergeSort(ListNode head) {
    if (head == null || head.next == null) {
        return head;  // Base case: list is empty or has only one element
    }

    // Step 1: Split the list into two halves
    ListNode mid = getMiddle(head);
    ListNode left = head;
    ListNode right = mid.next;
    mid.next = null; // Disconnect the left half from the right half

    // Step 2: Recursively sort each half
    left = mergeSort(left);
    right = mergeSort(right);

    // Step 3: Merge the two sorted halves
    return merge(left, right);
}
```

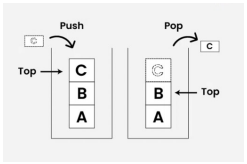
③ Sort: { Traverse
compare → insert.

Stack:

- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO (Last In, First Out) or FILO (First In, Last Out).

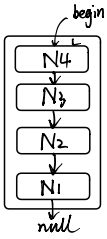
后进先出　　先进后出

1. Operations:



{ Fixed stack: ☆ check, update int top. capacity
  Dynamic size:

2. Implementation by Linked List:



(1) class Node:
    Node next
    Object data

(2) ① push: Insert at beginning

```
push(x):
    newNode ← Node(x)
    newNode.next ← head.next
    head.next ← newNode
```

② Pop: delete at beginning and return

```
pop():
    if isEmpty() then:        ← ☆Check if empty
        return null
    else:
        tmp ← head.next
        head.next ← tmp.next
        return tmp.data
    end if
```

③ Top: return head data

```
public Object top() {
    if (!isEmpty())
        return head.next.element;
    else {
        return null;
    }
}
```

3. Implementation by Array with fixed size

(1)

```
class Stack {
    final static int MIN_STACK_SIZE = 5;
    int topOfStack = -1;
    Object[ ] array;
}
```

Constructor:

```
public Stack (int maxElements) {
    int capacity = maxElements;

    if (maxElements < MIN_STACK_SIZE)
        capacity = MIN_STACK_SIZE;

    array = new Object[capacity];
}
```

☆ num of data 不☆ exceed size limitation ☆

(2) ① isFull():

```
return topOfStack = array.length - 1
```

② push(x):

```
if (isFull())
    return false;
else
    array[++topOfStack] = x;
return true;
```

③ top():

```
if (!isEmpty())
    return array[topOfStack];
else
    return null;
```

④ pop():

```
if (!isEmpty())
    return array[topOfStack--];
else
    return null;
```

⇓

Advantage { Linked list: space flexiblility
  Array: faster { memory allocation (directly using index) ⇒ accessibility
    Continous memory can be loaded into cache

3. Application:

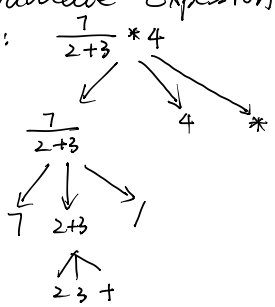
(1) Symbol check: open in close pop

⇒
- If the symbol is an opening symbol, push it onto the stack
- If it is a closing symbol
  - If the stack is empty, return false
  - Otherwise, pop from the stack; if the symbol popped does not match the closing symbol, return false

Step 3: at the end, if the stack is not empty, return false (unbalanced), else return true (balanced)

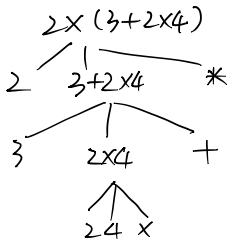
(2) Evaluate Expression : postfix.

eg: $\frac{7}{2+3} * 4$



⇒ 7 2 3 + / 4 *

once access operator ⇒ pop twice.

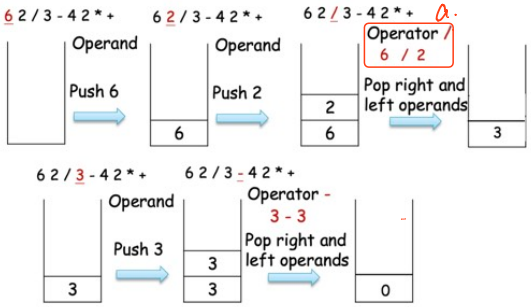


2 3 2 4 x + *

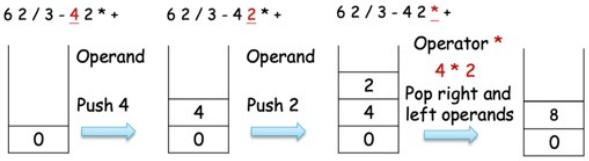
反过来把 postfix 放进 stack evaluate expression:

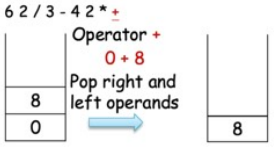
using recursive stack idea:

▸ Evaluate the postfix expression: 6 2 / 3 - 4 2 * +



▸ Evaluate the postfix expression: 6 2 / 3 - 4 2 * +





Return 8 as the answer

a. stop push when meet operator.

b. determine which numbers are applied to that operator.

(3) Using two stack to sort :

stack1 : store input data

stack2 : sorted stack.

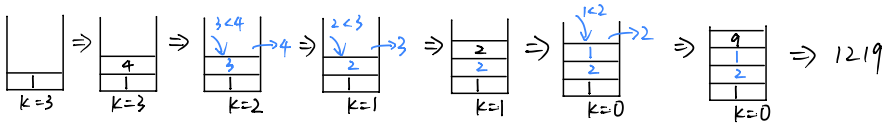
tmp : using to compare ⟶ determine { 放入 stack2 / 交换，返回 stack2.top to stack1

```
While stack1 != null do:
    tmp ← stack.pop()
    while stack2 and stack2.top() > tmp do:
        stack1.push(stack2.pop())
    end while
    stack1.push(tmp)
end while
```

(4) Given string num representing a non-negative integer num, and an integer k, return the smallest possible integer after removing k digits from num

Example 1:

```
Input: num = "1432219", k = 3
Output: "1219"
Explanation: Remove the three digits 4, 3, and 2 to form the new number 1219 which is the
smallest.
```



(5) Given a list of prices of a stock for **N** days. The task is to find the stock span for each day.

Stock span can be defined as the number of consecutive days before the current day where the price of the stock was equal to or less than the current price.

Examples:
Input: A[] = [100,60,70,65,80,85]
Output: [1,1,2,1,4,5]

①

```
int[] span = new int[n];                                   Copy  Edit
Stack<Integer> stack = new Stack<>();

span[0] = 1;  // Span of first day is always 1
stack.push(0);  // Push the index of the first day

for (int i = 1; i < n; i++) {
    // Pop while the current price is higher than stack's top price
    while (!stack.isEmpty() && prices[i] >= prices[stack.peek()]) {
        stack.pop();
    }

    if (stack.isEmpty()) {
        span[i] = i + 1;
    } else {
        span[i] = i - stack.peek();
    }

    stack.push(i);  // Push current day index
}

return span;
```

day index

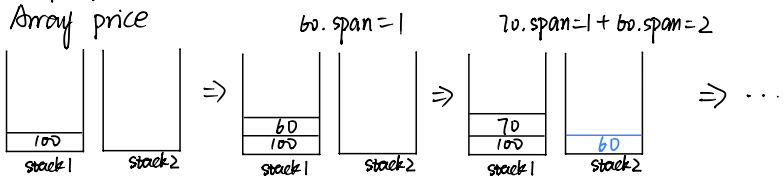
i = peek + 1

p[i] > p.peek ⇒ { 大于 p.peek 所大于的天数. / 大于 p.peek 的天数依然在 stack 里

② Using two stack:

Map span

Array price



a. using stack1 update new price

↓

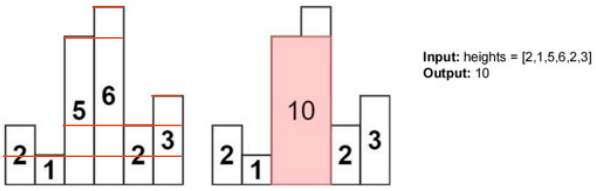
once read data, span + 1

b. Compare:

```
if stack.isEmpty() then:
    span[newNode] += 1
else:
    if newNode < peek then:
        stack1.push(newNode)
    else:
        while newNode > stack.peek() do:
            a = span[stack.peek()]
            span[newNode] += a
            stack2.push(stack1.pop())
        end while
    end if
end if
stack1.push(newNode)
```

(6): Exercise 3: Largest Rectangular Area in a Histogram using Stack (LeetCode P84)

Given an array of integers heights representing the histogram's bar height where the width of each bar is 1, return the area of the largest rectangle in the histogram.



Input: heights = [2,1,5,6,2,3]
Output: 10

⇒ stack 与 array 结合处理

(partly) consecutive data question

Similar to (5)①, using stack store index.

⇒ if larger: push

else: calculate area, then pop.

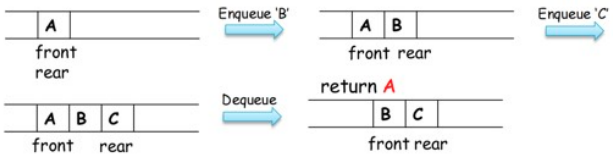
```
MaxArea = 0
for i from 0 to heights.length do:
    while peek > lengths[i] do:
        top = stack.pop()
        width = i if not stack else i - top
        Area = width x lengths[top]
        MaxArea = max{Area, MaxArea}
    end while
    push(i)
end for
```

处理剩余的: ⇒ 读到 top 指.

```
while stack do:
    top = stack.pop()
    width = i if not stack else i - top
    Area = width x lengths[top]
    MaxArea = max{Area, MaxArea}
end while.
```

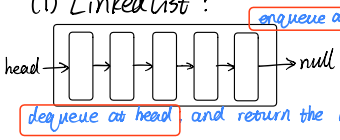
Queue:

1. Linear queue:



2. Operation:

(1) Linked list:



☆ Check if empty

① Create Node:

```
Node(x):
    data = x
    next = null
```

```
Queue():
    front = null
    rear = null
```

② enqueue(x):

```
tmp ← Node(x)
if front = null then:
    front, rear ← tmp
else:
    rear.next ← tmp
    rear ← tmp
```

insert and update rear.

③ dequeue:

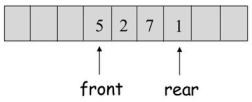
```
tmp ← front
front ← front.next
if front = null:
    rear ← null
return tmp.data
```

delete then return data.

④ peek():

```
if isEmpty:
    return null
else:
    return front.data
```

(2) Array:



①

```
public class QueueArray {
    int capacity;
    int front, rear;
    ElementType[] array;
}
```

从 default Node 变成 index

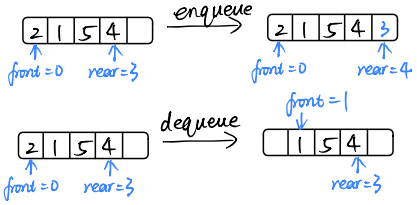
update ☆: ++ or --

② Operations: check if isFull

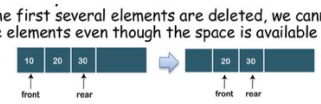
- To enqueue an element
  1. Increase rear, then set
  2. array[rear] = X ⇒ array[++rear] = X
- To dequeue an element
  1. Return the value of array[front], and then increase front

```
a = array[front]
front++
return a
```

eg:

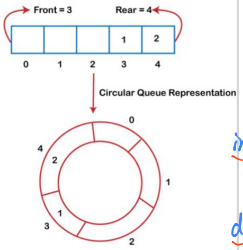


☆ Problem: run out of space.



→ using circular queue to solve

2. Circular Queue: 用于循环进出的 Array (持续有进有出)



front 与 rear 的 index 可以不是 0~n

⇒ enqueue 的时候从 1st available position insert.

insert index: rear = (rear + 1) % capacity

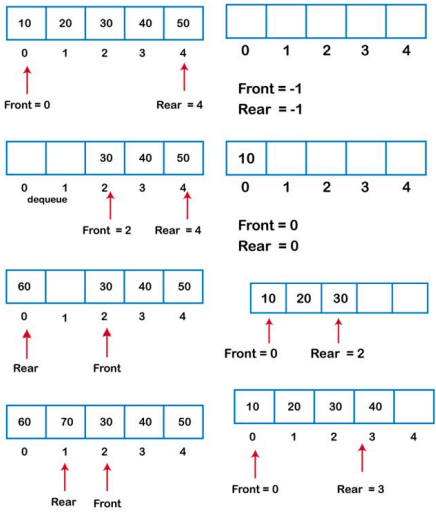
rear 向后

dequeue index: front = (front + 1) % capacity

front 向后.

front +1: 基于从 0 开始

% capacity: 从 1 开始



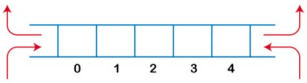
3. Priority Queue: only support for comparable element ☆

enqueue, dequeue = f(data)

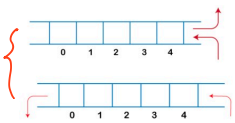
4. Double ended queue:

Not follow FIFO but still linear.

⇒ Can handle the room problem of array queue



⇒ Can be both used as stack or queue.



(1) Operation:

① Enqueue: 可以在 front 或 rear enqueue

Also, 我们可以限制单边的进出 ☆

☆ a. 对 front:

☆ front = (front − 1 + capacity) % capacity

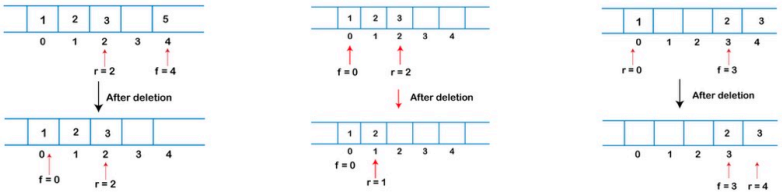
b. 对 rear:

rear = (rear + 1) % capacity

① Check if isFull:

if (rear + 1) % size == front.

② Dequeue:



if rear − 1 = null

n − 1 一定有有 data.

② dequeue: size −−

enqueue: size ++

a. 对 rear:

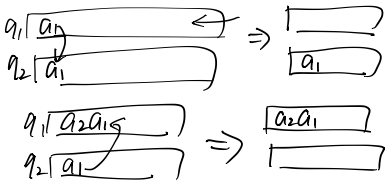
☆ rear = (rear − 1 + capacity) % capacity

b. 对 front:

front = (front − 1) % capacity

☆ adjust order.

5. using two queues implement stack

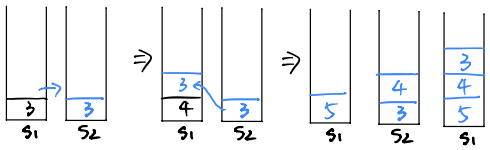


把 q_2 reorder 的结果再 enqueue 进 q_1

最后 q_2 为空 ✓

```
while !q1.isEmpty do:
    q2.enqueue(q1.dequeue)
end while
q1.enqueue(data)
while !q2.isEmpty do:
    q1.enqueue(q2.dequeue)
```

6. using two stack implement queues:




```
while !S1.isEmpty do:
    S2.push(S1.pop)
S1.push(data)
While !S2.isEmpty do:
    S1.push(S2.pop)
```

最后保证 S_2 空的.

S_2 的 data 是 stack LIFO order

eg:

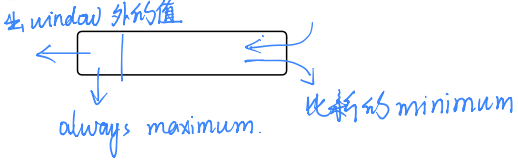
Exercise 2: Sliding Window Maximum(LeetCode P239)

Given an array of integers, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.

Example 1:

Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]
Explanation:

Window position	Max
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7



用 double ended queue construct monotonic queue

可以双向 monotonic

☆ restricted DEQueue

Sorting:

1. Insertion Sort:

(1) Def:

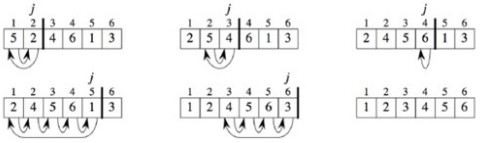
- A simple algorithm for a small number of elements
- Similar to sort a hand of cards
  - Start with an empty left hand
  - Pick up one card and insert it into the correct position
  - To find the correct position, compare it with each of the cards in the hand, from right to left
  - The cards in the left hand are sorted

① { tmp 用于 left iterate
  { cur: to be inserted
  record { current ind
  { min ind

② once cur < tmp. tmp 向前1个 ind.

左边是 sorted 的. 一旦 key > arr[j] ⇒ stop. insert ☆

eg:



(2) Code:

① Non recursion:

```
for i ← 1 to n-1:                    # traverse right part
    tmp ← arr[i]
    int j
    for j ← i-1 to 0 and arr[j] > tmp:
        arr[j+1] ← arr[j]
    arr[j+1] ← tmp
```

② Recursion:

```
if length = 1:
    return
insertSort(arr, length-1)
tmp ← arr[n-1]
j ← n-1
while j >= 0 and arr[j] > tmp:
    arr[j+1] = arr[j]
    j--
arr[j+1] = tmp.
```

☆ The left sorted array maintains sorted.

(3) loop invariant:

- Proof: loop invariant: in each iteration, A[1,...,j-1] is sorted
  - Initialization: true before the begin of loop
    Only one element A[1]
  - Maintenance: true before an iteration and after it
    A[j] is in the correct position j' ⇔ A[j'-1]<= A[j']<= A[j'+1]
  - Termination: when the loop stops, use the loop invariant to show the algorithm is correct
    j = n+1 when loop stops, A[1,... j-1] is sorted

(4) Time complexity:

```
INSERTION-SORT(A)
for j ← 2 to n                          C1, n-1
    do key ← A[j]                       C2, n-1
    ▷ Insert A[j] into the sorted sequence A[1..j-1].   0, n-1
    i ← j-1                             C4, n-1
    while i > 0 and A[i] > key          C5, Σ_{j=2}^{n} t_j      每次要移动几次
        do A[i+1] ← A[i]                C6, Σ_{j=2}^{n} (t_j-1)
           i ← i-1                      C7, Σ_{j=2}^{n} (t_j-1)
    A[i+1] ← key                        C8, n-1
```

key 只交换 2次. 但我的 key 向后 move 多次

eg: A[i-2] | A[j-1] | A[j] | key

在一个 loop 内. share same complexity

Only need to sort n-1 items.

$$\Rightarrow T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} t_j + c_6\sum_{j=2}^{n}(t_j-1) + c_7\sum_{j=2}^{n}(t_j-1) + c_8(n-1).$$

T(n) depends on n and t_j

- Best case: the array is sorted
  => $t_j = 1$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

- Worst case: the array is in reverse order
  => $t_j = j$

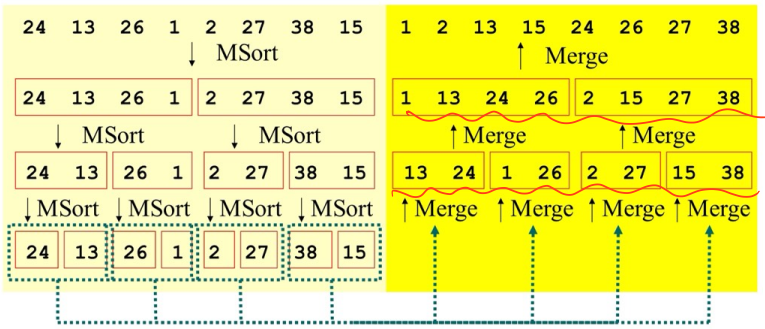
$\sum_{j=2}^{n} j = \left(\sum_{j=1}^{n} j\right) - 1$, it equals $\frac{n(n+1)}{2} - 1$

$\Rightarrow$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8). \end{aligned}$$

$\sum_{j=2}^{n}(t_j - 2)$ ⇒ (2到n)每个-1 ⇒ 1到n-1: $\frac{n(n-1)}{2}$.

2. Merge Sort:



⇒ { divide into $\log n$ levels
conquer spend $O(n)$ for each level.

↳ 不斷在 partly sorted 的 array 來 sort.

(1) Code:

①
```java
public static void mergesort1(int[] arr, int[] tmp, int left, int right){
    if (left == right){
        return;
    }

    int mid = (left + right) / 2;
    mergesort1(arr, tmp, left, mid);
    mergesort1(arr, tmp, mid + 1, right);
    merge1(arr, tmp, left, right, mid);
}
public static void merge1(int[] arr, int[] tmp, int left, int right, int mid){
    int i = left;
    int j = mid + 1;
    int k = left;

    while (i <= mid && j <= right){
        if (arr[i] < arr[j]){
            tmp[k++] = arr[i++];
        } else {
            tmp[k++] = arr[j++];
        }
    }
    while (i <= mid){
        tmp[k++] = arr[i++];
    }
    while (j <= right){
        tmp[k++] = arr[j++];
    }
    for (int m = left; m <= right; m ++) {
        arr[m] = tmp[m];
    }
}
```

②
```java
public static int[] mergesort2(int[] arr, int left, int right){
    if (left == right){
        return new int[] { arr[left] };
    }

    int mid = (left + right) / 2;
    int[] leftArr = mergesort2(arr, left, mid);
    int[] rightArr = mergesort2(arr, mid + 1, right);
    return merge2(leftArr, rightArr);
}
public static int[] merge2(int[]leftArr, int[] rightArr){
    int[] merged = new int[leftArr.length + rightArr.length];
    int i = 0; // for leftArr
    int j = 0; // for rightArr
    int k = 0; // for merged
    while (i < leftArr.length && j < rightArr.length){
        if (leftArr[i] < rightArr[j]){
            merged[k++] = leftArr[i++];
        } else {
            merged[k++] = rightArr[j++];
        }
    }
    while (i < leftArr.length){
        merged[k++] = leftArr[i++];
    }
    while (j < rightArr.length){
        merged[k++] = rightArr[j++];
    }
    return merged;
}
```

Analysis for merge:

☆ merge 在循环里是逐层做的.
但 mergeSort 在每层里一次性 implement 结果.

a.
```java
while (i <= mid && j <= right){ O(n/2) + O(n/2)
    if (arr[i] < arr[j]){ O(n/2)
        tmp[k++] = arr[i++]; O(n/2)
    } else {
        tmp[k++] = arr[j++]; O(n/2).
    }
}
```
⇒ $O(n)$

b.
```java
while (i <= mid){ O(n/2)
    tmp[k++] = arr[i++]; O(n/2)
}
while (j <= right){ O(n/2)
    tmp[k++] = arr[j++]; O(n/2)
}
```
⇒ $O(n)$

⇒ sum: $O(n)$

(2) Complexity: $T(n) = \begin{cases} c & \text{if } n = 1, \\ 2T(n/2) + cn & \text{if } n > 1. \end{cases}$

$a = 2 = b^d \Rightarrow O(n \log n)$

$$T(1) = C$$
$$T(N) = 2T(N/2) + CN$$
$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + C = \dots = \frac{T(1)}{1} + C\log N$$
$$T(N) = CN\log N + CN = O(N\log N)$$

3. Randomized algorithm:

(1) flip coin:

Algorithm: flipCoin()
```
1 r ← RANDOM(0,1)
2 while r != 1
3     r = RANDOM(0,1)
```

→ geometric distribution. $f(x) = (\frac{1}{2})^{x_i - 1} \cdot (\frac{1}{2}) = (\frac{1}{2})^{x_i}$

How to determine the running cost:

⇒ Let R.V. X: num of program's operation.

⇒ If get 1 once: { random → $O(1)$; compare → $O(1)$ } ⇒ $O(2)$. ⇒ $P(X=2) = f(1) = \frac{1}{2}$.

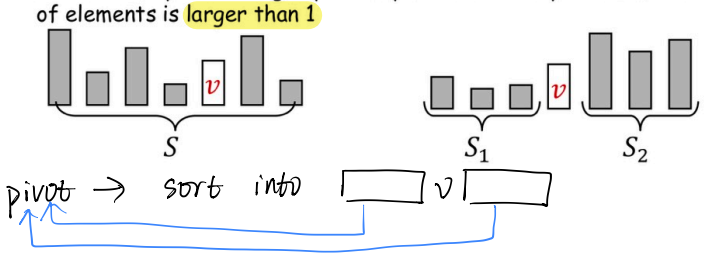
$\Rightarrow P(2y_i) = f(y_i) = (\frac{1}{2})^{y_i}$

$E(X) = \sum_{i=1}^{\infty} 2y_i \cdot (\frac{1}{2})^{y_i} = 2 \times \frac{1}{1-\frac{1}{2}} = 4 = O(1)$

(2) Quick Sort: $O(n \log n)$

① Ideal:

- Randomly pick an element, denoted as the pivot, and partition the remaining elements to three parts
  - The pivot
  - The elements in the left part: smaller than the pivot
  - The elements in the right part: larger than the pivot
  - For the left part and right part: repeat the above process if the number of elements is larger than 1



② Algorithm:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[-1]

    left = [x for x in arr[:-1] if x <= pivot]
    right = [x for x in arr[:-1] if x > pivot]

    return quick_sort(left) + [pivot] + quick_sort(right)
```

right 从右 left

⇒ Record: partition ind. traverse ind.

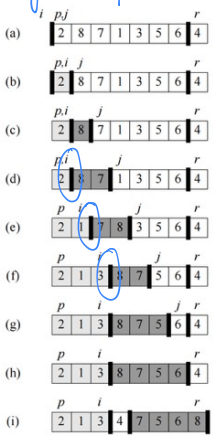
Once a smaller found: swap traverse and partition, update partition ind

找一个 pivot. ⇒ partition { left: < pivot, right: > pivot.

⇓ sort the arr

output updated pivot position.

eg: 单次 partition:



⇒ updated pivot position.

Code:

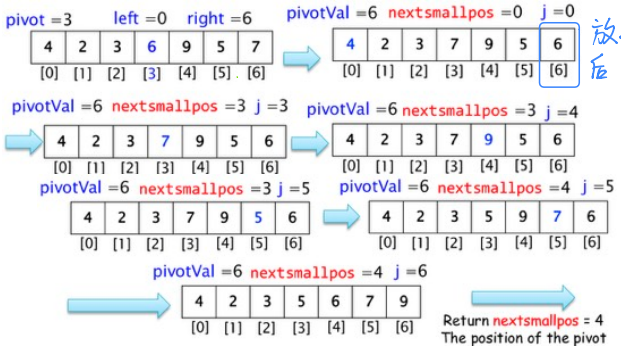
```
public static void quicksort(int[] arr, int left, int right){
    if (left >= right){
        return;
    }

    int pivot = (left + right) / 2;                          O(1)
    int pivotPos = partition(arr, left, right, pivot);       O(n)

    quicksort(arr, left, pivotPos - 1);                      T(n/2)
    quicksort(arr, pivotPos + 1, right);                     T(n/2)
}

public static int partition(int[] arr ,int left, int right, int pivot){
    int pivotVal = arr[pivot];
    int newPivotPos = left;                                  O(1)
    arr[pivot] = arr[right];
    arr[right] = pivotVal;
    for (int i = left; i <= right - 1; i ++){                O(n)
        if (arr[i] < pivotVal){
            int tmp = arr[newPivotPos];
            arr[newPivotPos] = arr[i];                       O(1)
            arr[i] = tmp;
            newPivotPos++;
        }
    }
    arr[right] = arr[newPivotPos];
    arr[newPivotPos] = pivotVal;
    return newPivotPos;
}
```

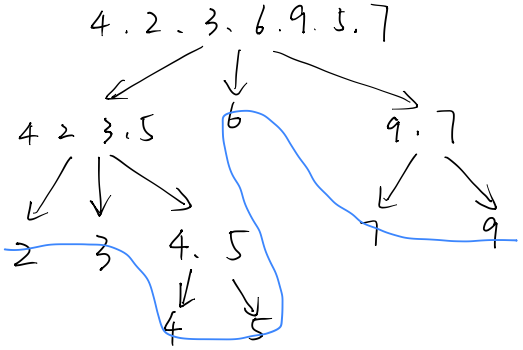
choose pivot → partition → divide.

$\Rightarrow T(n) = 2T(\frac{n}{2}) + O(n)$

$\Rightarrow O(n \log n)$

eg:



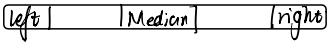


③ Comparison:

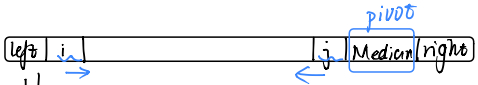
- When MergeSort executes the merge operation
  - Requires an additional array to do the merge operation
  - Needs to do additional data copy: copy to additional array and then copy back to the input array
- When QuickSort executes the partition operation
  - Operates on the same array
  - No additional space required

④ Median of 3 implementation:
to choose a good pivot.

a. choose: leftmost, median and right
Sort, choose the middle one as pivot



b.

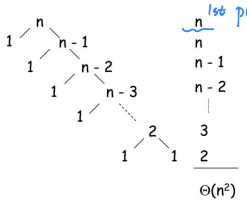


⇓

- i++ if arr[i] < pivot
- j-- if arr[j] > pivot
- swap i, j if arr[i] > arr[j]
- stop when i > j
- swap arr[i] and arr[pivot]

```java
public static void median3(int[] arr, int left, int right){
    if (left >= right){
        return;
    }
    int mid = (left + right) / 2;
    int[] tmp = {arr[left], arr[mid], arr[right]};
    InsertionSort.insertSort(tmp);
    arr[left] = tmp[0];
    arr[mid] = tmp[1];
    arr[right] = tmp[2];

    int newPivotPos = partition2(arr, left, right, mid);
    median3(arr, left, newPivotPos - 1);
    median3(arr, newPivotPos + 1, right);
}
public static int partition2(int[] arr, int left, int right, int pivot){
    int pivotVal = arr[pivot];

    arr[pivot] = arr[right - 1];
    arr[right - 1] = pivotVal;

    int i = left + 1;
    int j = right - 2;

    while (i <= j) {
        if (arr[i] < pivotVal){
            i ++;
        } else if (arr[j] > pivotVal){
            j --;
        } else {
            int tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
            i ++;
            j --;
        }
    }
    int temp = arr[i];
    arr[i] = arr[right - 1];
    arr[right - 1] = temp;
    return i;
}
```

⑤ Worst case partitioning:
⇒ unbalanced left and right:
One side none or 1 element:

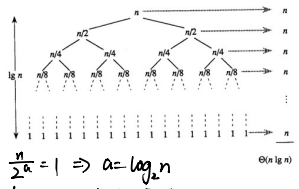


1st pivot ⇒ n个 element compare n次

n
n
n - 1
n - 2
3
2

$\Rightarrow T(n) = T(1) + T(n-1) + n$
$= 2T(1) + T(n-2) + n + (n-1)$
$= 3T(1) + T(n-3) + n + (n-1) + (n-2)$
$= (n-1)T(1) + T(1) + n + (n-1) + \cdots + 2$
$= nT(1) + \frac{n(n+1)}{2} - 1 = O(n^2)$

⑥ Best case:

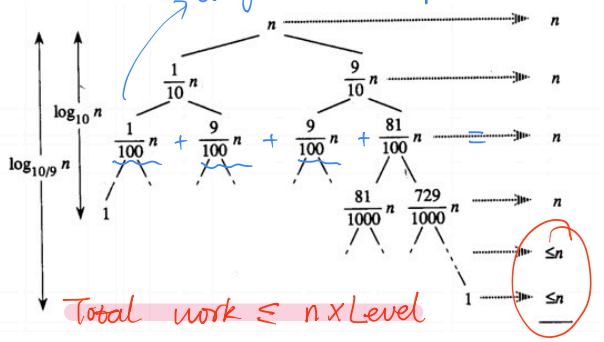


$T(n) = 2T(\frac{n}{2}) + O(n)$
$\Rightarrow T(n) = O(n \log_2 n)$

$\frac{n}{2^a} = 1 \Rightarrow a = \log_2 n$
每层有 n 个数据关计

every element compare + move = O(1) ⇒ O(n)

eg:



$T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + n$

$\frac{n}{10^a} = 1$, $a = \log_{10} n$ ⇓ lower bound

$\frac{n}{(\frac{10}{9})^b} = 1$, $b = \log_{\frac{10}{9}} n$ ⇓ upper bound

$T(n) \le n(\log_{\frac{10}{9}} n + 1)$ begin from level 0. ☆

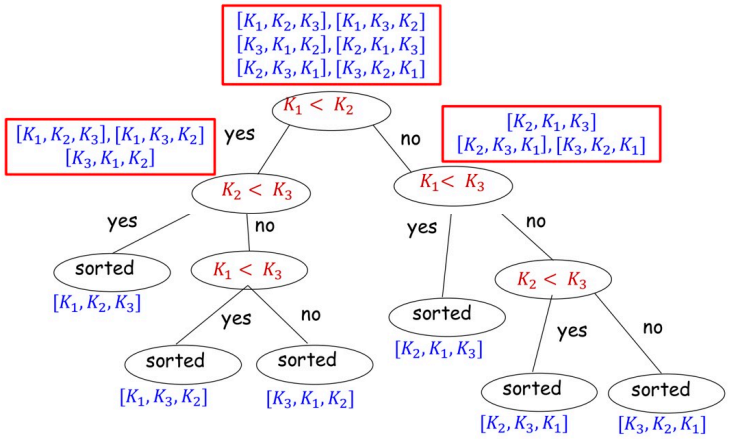
⑦ Stability:

** A sorting algorithm is said to be stable, if two objects with equal keys appear in the same order in sorted output, as they appear in the input array
Selection Sort: 5, 2, 3, 5*, 1 => 1, 2, 3, 5*, 5

(3).

① :Decision tree:

Sorting an array ⇒ find the best sol. in n! permutations.



② ▸ **Theorem**: Any decision tree that sorts n distinct keys has a height of at least $\log_2 n! + 1$.

⇓

n! permutations

⇒ n! permutations

let height = k → $\frac{n}{2^k} \le 1$, $k \ge \log_2 n + 1$

$$\Rightarrow \log_2 n! = \sum_{i=1}^{n} \log i \ge \sum_{i=\frac{n}{2}}^{n} \log i \ge \frac{n}{2} \cdot \log \frac{n}{2} = \Omega(n \cdot \log n)$$

initial item

4. Shell Sort:

using a gap to group → each group implement insertion sort → reduce the gap → recurse

每次減少reduce eg: 5→3→1

```
public static void shellSort(int[] arr){
    int n = arr.length;
    int gap = n / 2;

    while (gap > 0){
        for (int i = gap; i < n; i += gap){
            int key = arr[i];
            int j;
            for (j = i - gap; j >=0 && key < arr[j]; j -= gap){
                arr[j + gap] = arr[j];
            }

            arr[j + gap] = key;
        }
        gap /= 2;
    }
}
```

```
public static void shellSortRec(int[] arr, int gap){
    if (gap <= 0){
        return;
    }

    for (int i = gap; i < arr.length; i += gap){   O(n/gap)
        int key = arr[i];
        int j;
        for (j = i - gap; j >= 0 && arr[j] > key; j -= gap){   O(n/gap)
            arr[j + gap] = arr[j];
        }
        arr[j + gap] = key;
    }

    shellSortRec(arr, gap / 2);
}
```

$\frac{n}{2} \to \frac{n}{4} \to \cdots \to 1$

i. best case ⇒ never go into inner loop
$O(\frac{n}{gap}) = O(n)$
$\Rightarrow O(n) \times \log n = O(n \log n)$

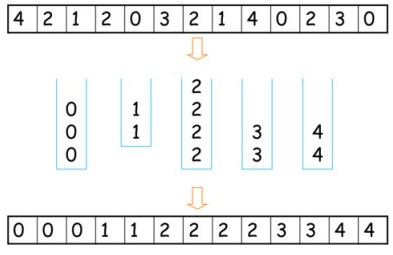
ii. Worst case ⇒ go into inner loop
$O(n) \times O(n) = O(n^2)$

(2) **Theorem**: the worst-case running time of Shellsort, using some increment, is $\Theta(N^2)$

5. Counting Sort: ▸ Steps

- Start with m empty buckets numbered 0 to m-1
- Scan the list and place element s[i] in bucket s[i]
- Output the buckets in order

⇒ No need to do comparison.



2) Implement:

```
public static int[] countingSortArray(int[] arr){
    int max = Arrays.stream(arr).max().getAsInt();
    int min = Arrays.stream(arr).min().getAsInt();

    int[] bucket = new int[max - min + 1];
    for (int i = 0; i < arr.length; i++){
        bucket[arr[i] - min] ++;
    }

    int[] sortedArr = new int[arr.length];
    int index = 0;
    for (int i = 0; i < bucket.length; i ++){
        while (bucket[i] > 0){
            sortedArr[index] = i + min;
            index++;
            bucket[i]--;
        }
    }
    return sortedArr;
}
```

O(n) O(n) ⇒ O(3n + m) = O(m+n)

Using HashMap:

```
public void countingSortMap(int[] arr) {
    Map<Integer, Integer> countMap = new HashMap<>();

    for (int num : arr) {
        countMap.put(num, countMap.getOrDefault(num, defaultValue:0) + 1);
    }
    int index = 0;
    for (int num : countMap.keySet()) {
        int count = countMap.get(num);
        for (int i = 0; i < count; i++) {
            arr[index] = num;
            index++;
        }
    }
}
```

6. Bucket Sort:

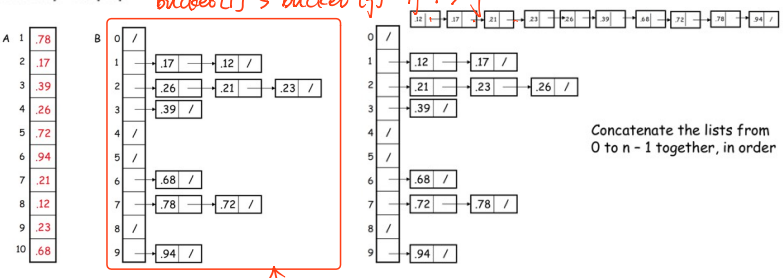
(1) Assumption:

- The input is generated by a random process that distributes elements uniformly over [0, 1)

- Key steps:
  - Divide [0, 1) into n equal-sized buckets
  - Distribute the n input values into the buckets ⇒ follow 任意或 specific distribution rule.
  - Sort each bucket (e.g., using QuickSort)
  - Go through buckets in order, listing elements in each one
  - Extra array: B[0 . . n - 1] of linked lists, each of which is initially empty

bucket[i] > bucket[j] if i > j

eg.



(2) Code:

```
BUCKET-SORT(A, n)
for i ← 1 to n
    do insert A[i] into list B[⌊nA[i]⌋]
for i ← 0 to n - 1
    do sort list B[i] with QuickSort
concatenate lists B[0], B[1], . . . , B[n -1]
together in order
return the concatenated lists
```

Bucket allocation: O(n)

QuickSort: O(n)

similar to merge: O(n)

O(n)

hashing ⇒ garantee elements in larger index bucket bigger than smaller index bucket

⇒ n elements with n buckets Because uniformly distributed.

1 bucket with 1 element ⇒ Compare with its self ⇒ O(1)

```
public static void bucketSort(float[] arr) {
    if (arr == null || arr.length <= 1) {
        return;
    }
    float min = arr[0];
    float max = arr[0];
    for (float num : arr) {
        if (num < min) min = num;
        if (num > max) max = num;
    }
    int bucketCount = arr.length;
    List<Float>[] buckets = new List[bucketCount];
    for (int i = 0; i < bucketCount; i++) {
        buckets[i] = new ArrayList<>();
    }
    for (float num : arr) {
        int bucketIndex = (int) ((num - min) / (max - min) * (bucketCount - 1));
        buckets[bucketIndex].add(num);
    }
    for (List<Float> bucket : buckets) {
        Collections.sort(bucket);
    }
    int index = 0;
    for (List<Float> bucket : buckets) {
        for (float num : bucket) {
            arr[index++] = num;
        }
    }
}
```

分配 buckets

7. Selection Sort: 反复找最小的往左 insert.

{ left part: sorted
  right part: candidate

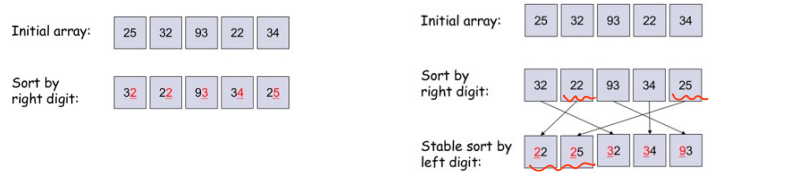
① Record { current segment (a)
           min index (b) find in the right part.

② swap (a) and (b)

```
for i ← 0 to n-1:
    min = i
    tmp ← arr[i]
    for j ← i+1 to n:
        if arr[j] < arr[min]:
            min = j
    arr[i] = arr[min]
    arr[min] = tmp.
```

8. Radix Sort: Apply BucketSort on each digit (from Least Significant Digit to Most Significant Digit)

eg:
- Suppose we sort some 2-digit integers
- Phase 1: Sort by the right digit (the least significant digit)
- Phase 2: Sort by the left digit (the second least significant digit)



※ When input has same length

从低位向高位进行 sort.

因为 1st digit is sorted
⇒ Same with 2 as 2nd digit
在 2 的 inner order 一样但在整体 arr 的排序变了.

(2) Code:

```
void radixsort(int A[], int n, int d)
{
   int i;
   for (i=0; i<d; i++)
      bucketsort(A, n, i);
}

// To extract d-th digit of x
int digit(int x, int d)
{        put the extracteed digit into bucketSort.
   int i;
   for (i=0; i<d; i++)
      x /= 10; // integer division
   return x%10;
}
```

9. Topological Sort:

(1) Topological ordering: ⇒ 用于有依赖关系的. 便被依赖的先被处理

① An ordering of all vertices in a directed acyclic graph, such that if there is a path from v_i to v_j, then v_j appears after v_i in the ordering
- If there is no path between v_i and v_j, then any order between them is fine

> ☆Warning:
> - Topological ordering is not possible if there is a cycle in the graph
> 有向无环图.
> - A DAG has at least one topological ordering

If cycle happen
⇒ no vertex with indegree = 1

(2) Sorting algorithm

1. Compute Indegree of All Vertices

Indegree of a vertex is the number of incoming edges to it.

- **Method**:
  - Initialize an array `indegree` of size `|V|` (number of vertices) with zeros.
  - For each edge `(u → v)` in the adjacency list:
    - Increment `indegree[v]` by 1 (since `v` has an incoming edge from `u`).

⇒ construct adjacent array

2. Find a Vertex with Indegree Zero

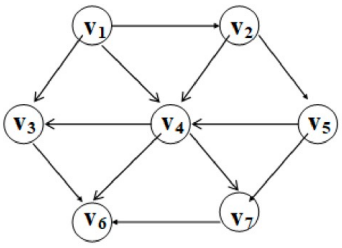
- Use a queue to track vertices with `indegree = 0`.
- Initially, add all vertices where `indegree[v] == 0` to the queue.

3. Process the Vertex and Remove Its Edges

- **Step 1**: Dequeue a vertex (e.g., `0`) and print it (part of the result).
- **Step 2**: For each neighbor `v` of `0` (i.e., `1` and `3`):
  - Decrement `indegree[v]` by 1 (because edge `0 → v` is "removed").
  - If `indegree[v]` becomes zero, add `v` to the queue.

logic of BFS

eg:



V_1: [V_2, V_3, V_4]
V_2: [V_4, V_5]
V_3: [V_6]
V_4: [V_3, V_6, V_7]
V_5: [V_4, V_7]
V_6:
V_7: [V_6]

⇒ indegree: $V_1=0$, $V_2=1$, $V_3=2$, $V_4=3$, $V_5=1$, $V_6=3$, $V_7=2$

steps:

① Q=[1], R=[]
deg: $V_1=0$, $V_2=1$, $V_3=2$, $V_4=3$, $V_5=1$, $V_6=3$, $V_7=2$

② Q=[2], R=[1]
deg: $V_1=0$, $V_2=0$, $V_3=1$, $V_4=2$, $V_5=1$, $V_6=3$, $V_7=2$

③ Q=[5], R=[1,2]
deg: $V_1=0$, $V_2=0$, $V_3=1$, $V_4=1$, $V_5=0$, $V_6=3$, $V_7=2$

④ Q=[4], R=[1,2,5]
deg: $V_1=0$, $V_2=0$, $V_3=1$, $V_4=0$, $V_5=0$, $V_6=3$, $V_7=1$

④ Q=[3,7], R=[1,2,5,4]
deg: $V_1=0$, $V_2=0$, $V_3=0$, $V_4=0$, $V_5=0$, $V_6=2$, $V_7=0$

⑤ Q=[7], R=[1,2,5,4,3]
deg: $V_1=0$, $V_2=0$, $V_3=0$, $V_4=0$, $V_5=0$, $V_6=2$, $V_7=0$

⑥ Q=[], R=[1,2,5,4,3,7]
deg: $V_1=0$, $V_2=0$, $V_3=0$, $V_4=0$, $V_5=0$, $V_6=1$, $V_7=0$

Q.size ≠ V

有cycle ☆

(3) Code:

```
void topsort () {
    for (int counter = 0; counter < numVertex; counter++) {
        Vertex v = FindNewVertexOfInDegreeZero (); //check all vertices
        if (v == null) {
            Error("Cycle Found"); return;
        }
        v.topNum = counter;
        for each Vertex w adjacent to v
            w.indegree--;
    }
}
```

每次少check 1个 ⇒ $\frac{(V+1)V}{2} = O(V^2)$

No cycle, 不会有 V-1个与 j th 相 adjacent ⇒ bounde by cycle
每次少一个 ⇒ $O(V^2)$

Running time is $O(|V|^2)$

```java
public static List<Integer> topologicalSort(int V, List<List<Integer>> Graph){
    int[] Indegree = new int[V];
    List<Integer> sortedArr = new ArrayList<>();
    for (int i = 0; i < V; i ++){
        for (int v: Graph.get(i)){
            Indegree[v]++;
        }
    }
    Queue<Integer> queue = new LinkedList<>();
    for (int i = 0; i < V; i++){
        if (Indegree[i] == 0){
            queue.add(i);
        }
    }
    while(!queue.isEmpty()){
        int u = queue.poll();
        sortedArr.add(u);
        for (int neighbor : Graph.get(u)){
            Indegree[neighbor]--;
            if (Indegree[neighbor] == 0){
                queue.add(neighbor);
            }
        }
    }
    return sortedArr;
```

10. Heap Sort:

(1) Algorithm:

① Using input arr construct complet BT

② (Max heap):

☆ Heapify: from bottom to root:

Compare parent and two children.

→ parents always the larger.

(2) Code:

```java
public static void buidHeap(int[] arr){
    int n = arr.length;
    for (int i = n / 2; i >= 0; i--){
        heapify(arr, i);
    }

}
public static void heapify(int[] arr, int parent){
    int parent1 = parent;
    int left = parent * 2 + 1;
    int right = parent * 2 + 2;

    if (left < arr.length && arr[left] > arr[parent]){
        parent = left;
    }
    if (right < arr.length && arr[right] > arr[parent]){
        parent = right;
    }

    if (parent != parent1){
        swap(arr, parent, parent1);
        heapify(arr, parent);
    }
}
```

11. Buble Sort:

two part: { left, right => sorted.

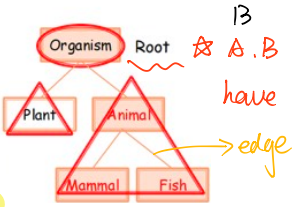
```java
public static void bubbleSort(int[] arr){
    int n = arr.length;
    int i = n;
    while (i > 0){
        for (int j = 0; j < i - 1; j++){
            if (arr[j] > arr[j + 1]){
                int tmp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = tmp;
            }
        }
        i --;
    }
}
```

Sorting algorithm	Stability	Time cost			Extra space cost
		Best	Average	Worst	
Bubble sort	√	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Insertion sort	√	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Selection sort	×	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
MergeSort	√	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
HeapSort	×	O(nlogn)	O(nlogn)	O(nlogn)	O(1)
QuickSort	×	O(nlogn)	O(nlogn)	$O(n^2)$	O(1)
RadixSort	√	O(nk)	O(nk)	O(nk)	O(n)

Tree:

1. Def:

(1) A tree is a finite set of one or more nodes such that
- Each node stores an element
- There is a special node called the root
- The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, \ldots, T_n$ where each of these sets is a tree
- We call $T_1, \ldots, T_n$ the subtrees of the root

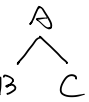


edge

△: subtree

n Nodes has n-1 edge

every Node is a root for some subtree.



☆ A, B are root for their own subtree, but a node can only have one root.

(2) ① Parent
- Node A is the parent of node B if B is the root of the left or right sub-tree of A

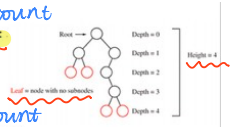
② Leaf
- A node is called a leaf if it has no children

③ Left (right) child
- Node B is the left (right) child of node A if A is the parent of B

④ Sibling
- Node B and node C are siblings if they have the same parent

⑤ A path from node n_i to n_k
- A sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1} for $1 \leq i < k-1$
- The length of this path is the number of edges on the path, namely k-1
- Notice that in a tree, there is exactly only one path from the root to each node

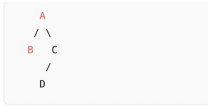
⑥ Depth of a node → 从上向下 count
- The depth of a node n_i is the length of unique path from the root to n_i
- The root is at depth 0



Height of a node → 从下向上 count
- The height of a node n_i is the length of the longest path from n_i to a leaf
- All leaves are at height 0

Height tree = Height root
Depth tree = Depth deepest leaf.
max Height = max Depth

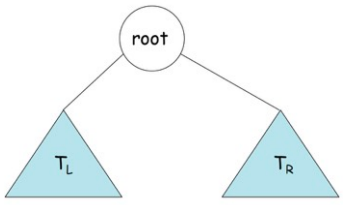
eg:



- Height of A = 2 (A → C → D).
- Height of B = 0 (since B is a leaf).
- Height of C = 1 (C → D).

Binary Tree:

1. Def: A binary tree is a tree, in which
- No node can have more than two children (subtrees): T_L and T_R, both of which could possibly be empty



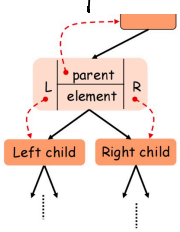
(1) ① Full binary tree
- A binary tree where all the nodes have either two or no children

② Complete binary tree
- A binary tree where all the levels are completely filled except possibly the lowest one, which is filled from the left

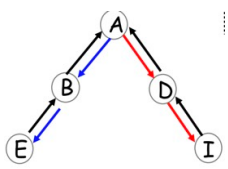
数据从左向右存.

2. Implementation (using pointer):

(1) Node:
- node.parent: store the address of its parent,
- node.leftchild: store the address of its left child,
- node.rightchild: store the address of its right child
- node.element: store the values



parent
leftchild
rightchild
(Omitted links points to NULL)



(2) Make Tree:

Algorithm: MakeBT(bintree1, element, bintree2)

```
1  rootNode <- allocate new memory
2  rootNode.element = element
3  rootNode.parent = NULL
4  rootNode.leftchild = bintree1
5  rootNode.rightchild = bintree2
6  if bintree1 != NULL
7    bintree1.parent = rootNode
8  if bintree 2 != NULL
9    bintree2.parent = rootNode
10 return  rootNode
```

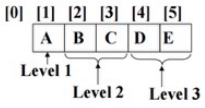
deal Root

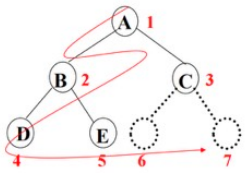
deal children

Final Root contains all the subtree information.

3. Implementation (using Array)

- An array representation
  - Given a complete binary tree with n nodes, for any i-th node, $1 \le i \le n$,
    - parent(i) is $\lfloor i/2 \rfloor$
    - leftChild(i) is at $2i$ if $2i \le n$; otherwise, i has no left child
    - rightChild(i) is at $2i+1$ if $2i+1 \le n$; otherwise, i has no right child

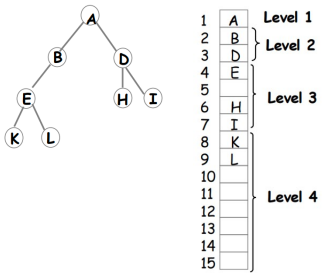




※ For array, start at 0 ⇒ {left: 2i+1, right: 2i+2

※ Disadvantage: If not full tree ⇒ null will occupy one space

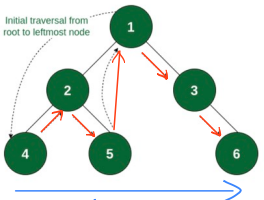
eg:



4. Traverse:

(1) Inorder:

Left → Root → Right



⇒ Out: 4 2 5 1 3 6

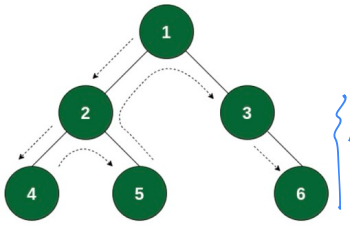
```
static void printInOrder(Node node){
    if (node == null) return;

    printInOrder(node.leftChild);
    System.out.print(node.element + " ");
    printInOrder(node.rightChild);
}
```

From left to right in order.

Stretch to an array in this direction

① begin at left most, stop at right most, root separate two subtree

(2) Preorder:



⇒ out: 1 2 4 5 3 6

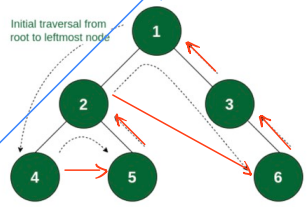
从上向下
从左向右

```
static void printPreorder(Node node){
    if (node == null) return;

    System.out.print(node.element);
    printPreorder(node.leftChild);
    printPreorder(node.rightChild);
}
```

begin at Root. stop at right most. ※ divide subtree into left and right half

Stretch to an array in this direction

(3) Postorder:



⇒ out: 4 5 2 6 3 1

从左向右的从下向上.

```
static void printPostorder(Node node){
    if (node == null) return;

    printPostorder(node.leftChild);
    printInorder(node.rightChild);
    System.out.print(node.element);
}
```

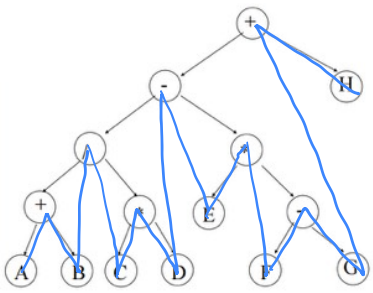
begin at leftmost. stop at Root.

(A+B)/(C*D)-E*(F-G)+H

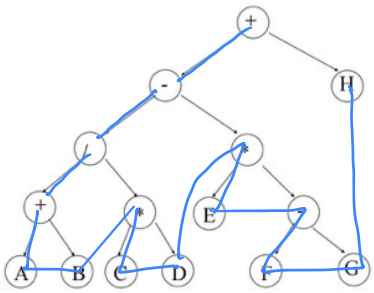
Preorder:
+-/+AB*CD*E-FGH

Inorder :
A+B/C*D-E*F-G+H

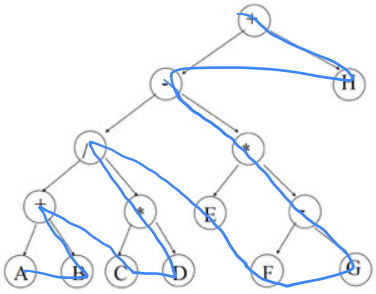
Postorder:
AB+CD*/EFG-*-H+



in order.



pre



post

(5) Reconstruction:

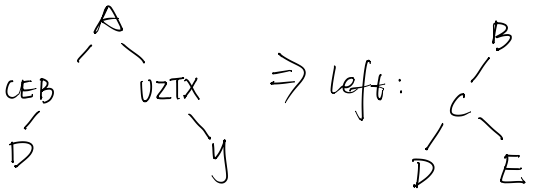
Combine Inorder and Preorder:

inorder 定首尾 / postorder 定 root } combine 分左右

inorder : DCEBAUZTXY → check 左右.

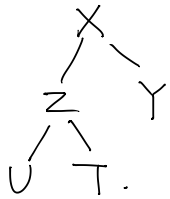
preorder : ABCDEXZUTY → 用 Pre check 深度 ; postorder → check 深度

Root A分为 subnode

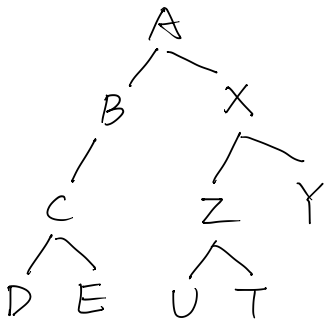


⇒ left :

⇒ Right :

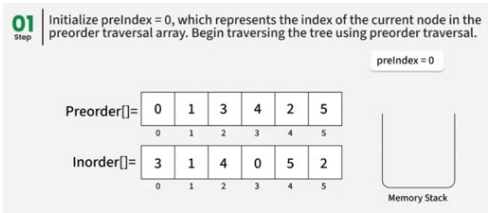


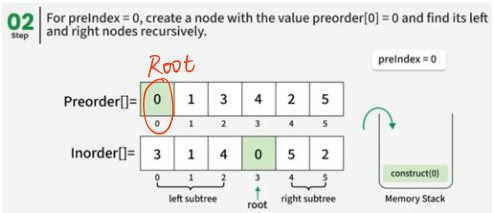
⇒

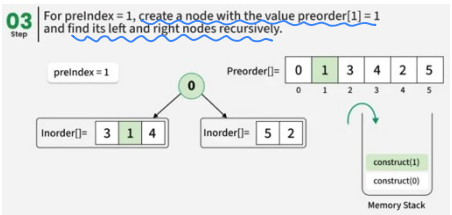


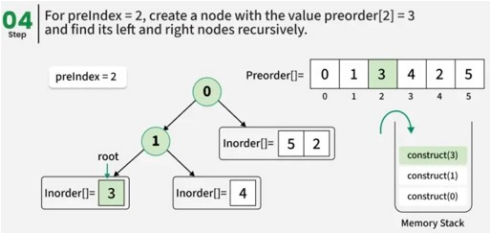
Only 左右 combine 深度 ⇒ unique tree

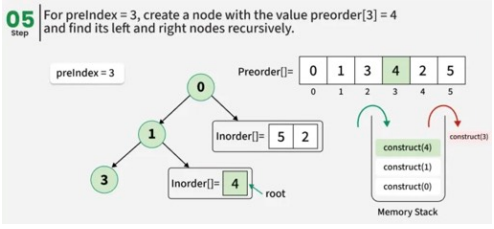
*Take the **first** element of the **pre-order array** and create **root** node. Find the index of this node in the **in-order** array. Create the **left subtree** using the elements present on left side of root node in in-order array. Similarly create the **right subtree** using the elements present on the right side of the root node in in-order array.*

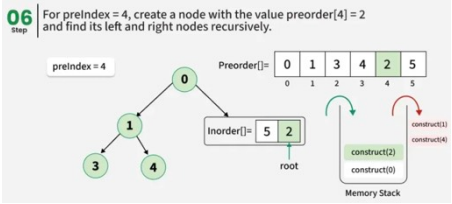


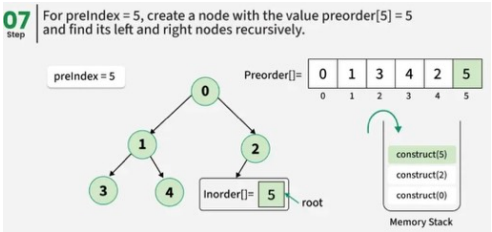






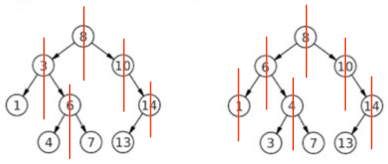






Binary Search Tree:

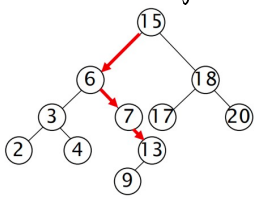
1. Def:
   - BST is a binary tree such that for each node T,
     - the key values in its left subtree are smaller than the key value of T
     - the key values in its right subtree are larger than the key value of T



※① Root 的左边 subtree 一定不会大于 root 的 parent.
② left sub < right sub

2. Operations:

(1) Search for a key:



```
Findkey(node, k)
    If node = null:
        return null
    If node = k:
        return node
    If k < node:
        return Findkey(node.left, K)
    else:
        return Findkey(node.right, K)
```

(2)
```
findMin(x)
1. while left[x] ≠ NIL
2.     do x ← left[x]
3. return x
```

```
findMax(x)
1. while right[x] ≠ NIL
2.     do x ← right[x]
3. return x
```

(3) Successor(x) : node x 的下一个 node.

Def: successor (x) = y, such that key [y] is the smallest key > key [x]

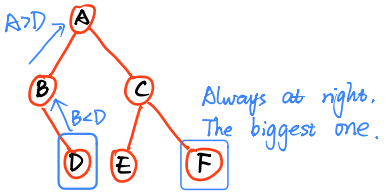
Find 2nd 大的: successor 的 successor

⇒ ① If right[x] ≠ null: return findMin(right[x])

leftmost of right subtree.

② right[x] = null: go up to find the node that x is in its left subtree

```
y ← parent[x]
while y != null and x = right[y]:
    do x ← y
       y ← parent[y]
return y
```



(4) Predecessor:

Def: predecessor (x) = y, such that key [y] is the biggest key < key [x]

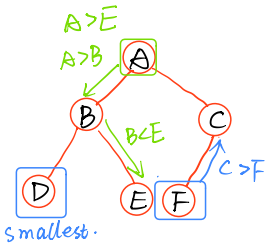
① If left[x] ≠ null: return: findMax(left[x])

right most. of left subtree

② Else:

```
y ← p[x]
while y != null:
    if x = right[y]
        return y
    x ← y
    y ← p[y]
```

go up to find the node that x is in the right subtree.



(5) Insertion: Always insert behind a leaf. ⇒ iterate until right[x] or left[x] = null

```
x ← root:
While x != null do:
    y ← x
    if key[z] < key[x]:
        x ← left[x]
    else:
        x ← right[x]
```

Find corresponding leaf

```
p(z) ← y
If y is null:
    root ← z
if key[z] < key[y]:
    left[y] ← z
else:
    right[y] ← z
```

tree is empty

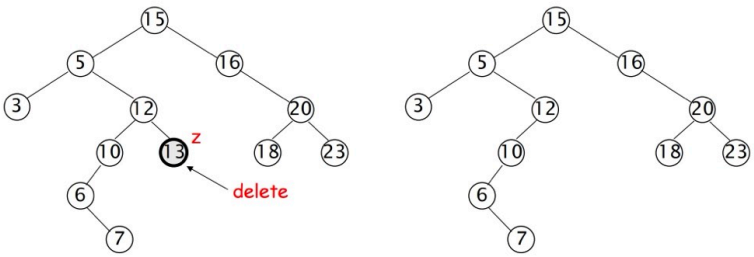
deal left and right.

Recursion implementation:

```java
private Node insertRec(Node root, int value) {
    // If the tree is empty, create a new node
    if (root == null) {
        root = new Node(value);
        return root;
    }

    // Otherwise, recur down the tree
    if (value < root.value) {
        root.left = insertRec(root.left, value);
    } else if (value > root.value) {
        root.right = insertRec(root.right, value);
    }

    // Return the (unchanged) node pointer
    return root;
}
```

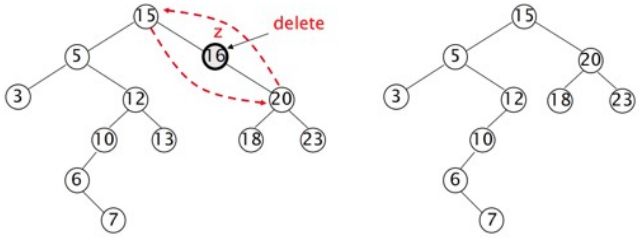
(6) Deletion:

① **Case 1:** z has no children
- Delete z by making the parent of z point to NIL




```
y ← p(z)
If z = left(y):
    left(y) ← null
else:
    right(y) ← null
```

② **Case 2:** z has one child
- Delete z by making the parent of z point to z's child, instead of to z, and link the parent with the new child




```
if left[z] = NIL and right[z] ≠ NIL
    y = right[z]
    if p[z] = NIL
        root[T] = y
    else
        p[y] = p[z]
        if z = left[p[z]]
            left[p[z]] = y
        else
            right[p[z]] = y
if left[z] ≠ NIL and right[z] = NIL
    y = left[z]
    if p[z] = NIL
        root[T] = y
    else
        p[y] = p[z]
        if z = left[p[z]]
            left[p[z]] = y
        else
            right[p[z]] = y
```

⇒ right child

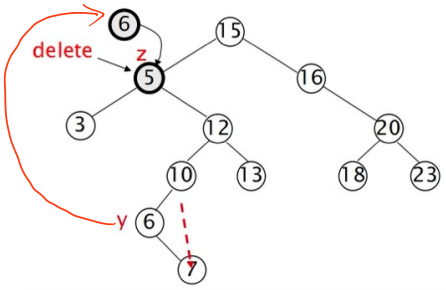
⇒ left child

~~③~~ Case 3: z has 2 children

⇒ Find successor(z) or predecessor(z)

leftmost of right subtree.

rightmost of left subtree




```
y ← successor(z)
If p[y] = z:
    right[z] = right[y]
    if right[y] ≠ null:
        p[right[y]] ← z
else:
    If right[y] != null:
        p[right[y]] ← p[y]
        left[p[y]] ← right[y]
    else:
        left[p[y]] = null
key[z] ← key[y]
```

Assuming x = left[y] != null

⇒ { x < y
  { x is in right subtree ⇒ x > z ⇒ x is successor

contradict.

(7) Operation is commutative:

- In a binary search tree, are the insert and delete operations commutative?
  - delete(a) then delete(b) ⇔ delete(b) then delete(a)?
  - insert(a) then insert(b) ⇔ insert(b) then insert(a)?

eg: Count num of leaves

Recursion ⇒ Recursively go down.

```
int Mytree::count_leaf(TreeNode* p)
{
    if (p == NULL)
        return 0;
    else if ((p->left == NULL) && (p->right == NULL))
        return 1;
    else
        return count_leaf(p->left) + count_leaf(p->right);
}
```

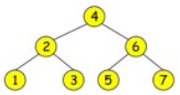
eg: Lowest Common ancestor:

Approach: 左右分拆的思想.

1) Start will the root
2) If root>n1 and root>n2 then lowest common ancestor will be in left subtree
3) If root<n1 and root<n2 then lowest common ancestor will be in right subtree
4) If Step 2 and Step 3 is false then we are at the root which is LCA, return it

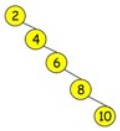
AVL Tree ⇒ balanced BST

- What is the best-case tree?
  - A complete binary tree



- What is the worst-case tree?
  - All nodes form a chain
  - E.g., inserting 2, 4, 6, 8, 10 into an empty BST

degenerate to linked list.



- Want a complete tree after every operation
  - All the levels are completely filled except possibly the lowest one, which is filled from the left
  - A complete binary tree has height of $O(\log n)$

the number of levels grows logarithmically with respect to the number of elements n.

All levels are filled

- In a **balanced binary tree**, each level doubles the number of nodes.
- So, if the total number of nodes is **n**, the height **h** satisfies:

$$2^h \approx n$$

1. Why unbalanced ⇒ orders of keys determine structure of tree

AVL:
- For every node in the tree, the height of the left subtree differs from the height of the right subtree by at most 1
- Balance factor of a node: height(left subtree) - height(right subtree)

The only condition for balance, 不用 complete tree 从 from left to right

⇒ i. { BF < 0: right subtree is too heavy
       BF > 0: left subtree is too heavy

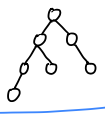
ii. BF ∈ {-1, 0, 1} is valid

Once = 2 or -2 ⇒ need balance

- If at any time they differ by more than one, rebalancing is done to restore this property

2. Theorem 1: Given a balanced binary search tree T of n nodes, the height, or equivalently the depth, of T is $O(\log n)$.

Proof: Let $f(h)$ be the min num of nodes at height h

$f(0)=1$: ○ $f(1)=2$: $f(2)=4$:  $f(3)=7$: 

⇒ $f(h)=f(h-1)+f(h-2)+1$

① When h is even:

$$f(h) > f(h-1)+f(h-2) > 2f(h-2) > 4f(h-4) > 8f(h-6) > \cdots > 2^{\frac{h-2}{2}} f(2) = 2^{\frac{h}{2}+1}$$

$2^{\frac{x}{2}} f(h-x)$

② When h is odd:

$$f(h) > f(h-1)+f(h-2) > 2f(h-2) > 4f(h-4) > \cdots > 2^{\frac{h-1}{2}} f(1) = 2^{\frac{h+1}{2}}$$

smaller

$\Rightarrow n > 2^{\frac{h+1}{2}} \Rightarrow n < 2\log_{\sqrt{2}} n - 1 = O(\log n)$

3. Insertion:

(1) Two principles:
- Imbalance will only occur on the path from the inserted node to the root (only these nodes have had their subtrees altered - local problem)
- Rebalancing should occur at the deepest unbalanced node (local solution too)

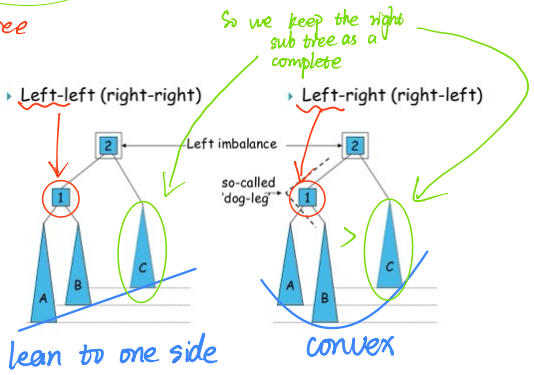
from def of BF ⇒ { BF < 0: right subtree is too heavy; BF > 0: left subtree is too heavy }

(2) General idea of rebalancing
- After the insertion, go back up to the root node by node, updating heights
- If a new balance factor is 2 or -2, rebalance tree by rotation around the node

⇒ 怎么做？

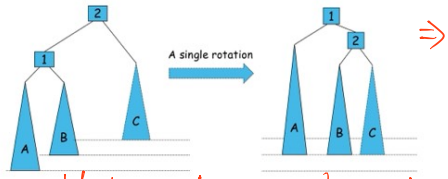
从 2 以下的为被 balanced 的 subtree

(3) Cases:
- Outside cases (require single rotation):
  - Left-left: insertion into left subtree of left child
  - Right-right: Insertion into right subtree of right child
  - (These two cases are symmetry)
- Inside cases (require double rotation):
  - Left-right: insertion into left child's right subtree
  - Right-left: insertion into right child's left subtree
  - (These two cases are symmetry)



⇒ Rotation:

① For Lean-to-one-side: ☆ Analyze the height of subtrees
- Subtrees B and C have the same height
- Subtree A is one level higher
- Therefore, make 1 the new root, 2 its right child and B and C the subtrees of 2

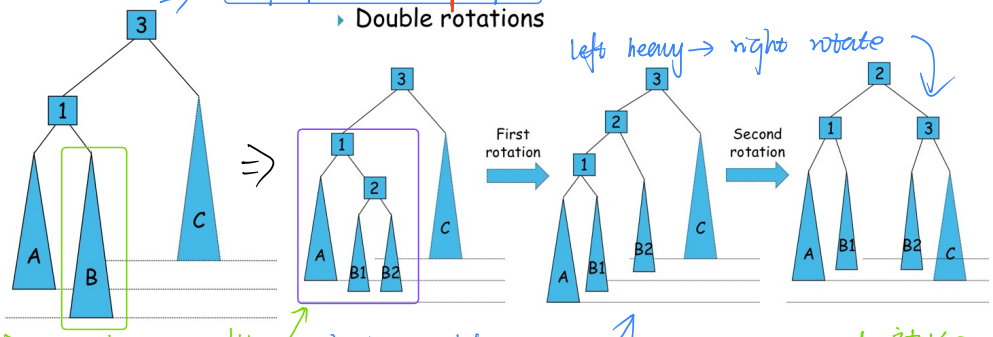


⇒ exchange root.

left is too heavy ⇒ right rotation

② Cur: ⇒ | A | 1 | B | 3 | C | → 隔板与中间偏大远了

- Double rotations



left heavy → right rotate

B is too heavy → split

right heavy → left rotate

method 1:

Split B → B_1 and B_2 ⇒

{ Method 1: Rotate: 从 subtree 内 拍 { left-right: 一半最大; right-left: 一半最小 } → one side imbalance method. (Transform to one side imbalance)

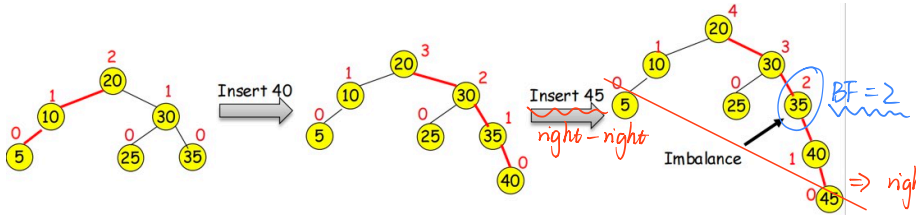
new node 被插入的 subtree

Method 2: 把拆开后的 树用 inorder traversal expand:

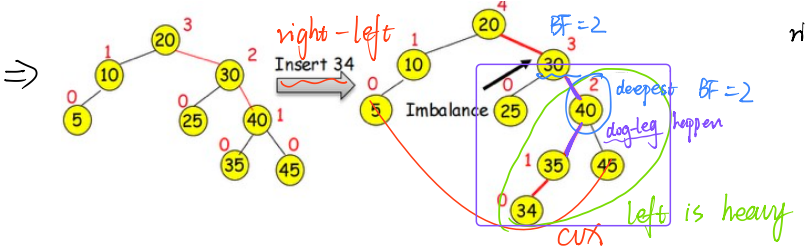
| A | 1 | B_1 | 2 | B_2 | 3 | C |

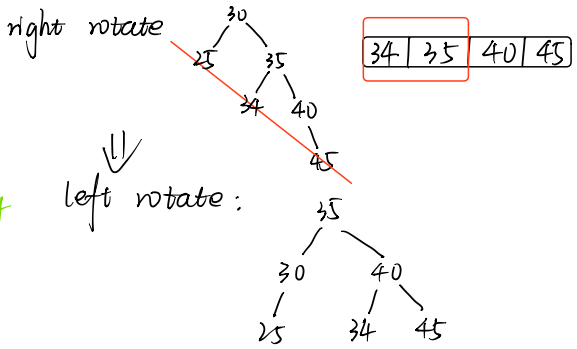
middle as new root.

eg:



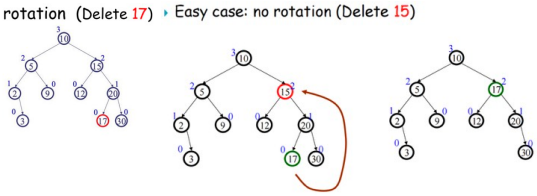
⇒ right one side imbalance, left rotate



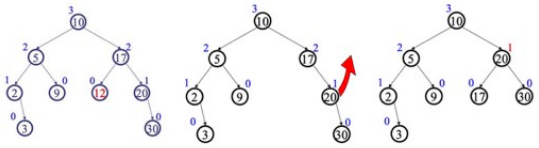


4. Delete: Imbalance may propagate upward.

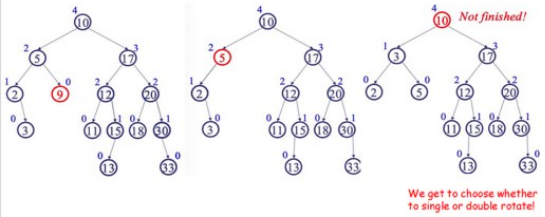
① Easy case: no rotation (Delete 17) Easy case: no rotation (Delete 15)



② Case 1: single rotation (Delete 12)



③ Case 2: double rotation (Delete 9)

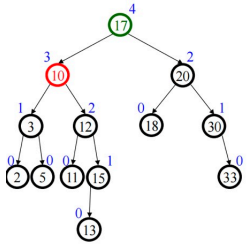


i. Single Rotation:

17 45 两边都 imbalance

⇒ 假设 attention 放在 20 subtree ⇒ right right ⇒

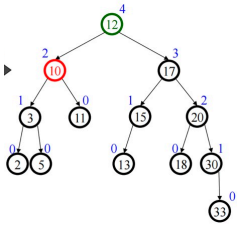
one side



ii. Double Rotation:

⇒ 假设 attention 放在 12 subtree ⇒ right left ⇒

conver.



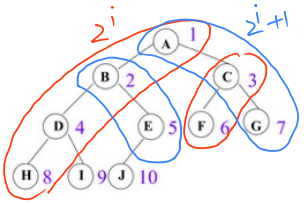
Heap

1. Binary heap.

(1) Structure:

- A heap is a complete binary tree
  - A binary tree that is completely filled, except at the bottom level, which is filled from left to right
- A complete binary tree of height h has between 2^h and $2^{h+1}-1$ nodes
  - $\Rightarrow 2^{h+1}-1$: $1+2+2^2+\cdots+2^h=\frac{2^{h+1}-1}{2-1}=2^{h+1}-1$ → 有 $h+1$ 项
  - 2^h: only $h-1$ levels are completely filled with 1 node at level h
    $\sum_{i=0}^{h-1} 2^i+1=\frac{2^h-1}{2-1}+1=2^h$
- The height of a complete binary tree = $\lfloor \log N \rfloor$
  - round down, e.g., $\lfloor 2.7 \rfloor = 2$

(2) Implement using array:

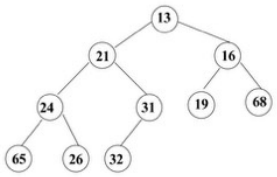


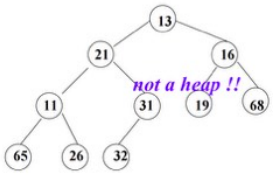
- The root is at position 1 (reserve position 0 for the implementation purpose)
- For an element at position i,
  - its left child is at position $2i$
  - its right child at $2i+1$
  - its parent is at floor $\lfloor i/2 \rfloor$

→ index follow order: from left to right $\begin{cases} \lfloor \frac{2i}{2} \rfloor = i \\ \lfloor \frac{2i+1}{2} \rfloor = i \end{cases}$

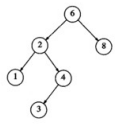
(3) Order of heap:

- The value at any node should be smaller than (or equal to) all of its descendants (guarantee that the node with the minimum value is at the root)

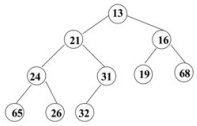




☒ Difference between BST:



A binary search tree



A binary heap

{ 小于 root ⇒ left child
  大于 root ⇒ right child

root < child 即可. child 无序.

2. Priority queue:

(1) Priority queue property:

- For two elements in the queue, x and y, if x has a ***lower priority*** value than y, x will be deleted before y

在 priority 的前提下 FIFO

(2) Implement:

- Singly linked list (Suggestion 1)
  - Insert at the front in $O(1)$
  - Delete minimum in $O(N)$
- Sorted array (Suggestion 2)
  - Insert in $O(N)$
  - Delete minimum in $O(N)$
- Binary heap (Suggestion 3)
  - Insert in $O(\log N)$
  - Delete minimum in $O(\log N)$
  - Two properties: structure property & heap order property

3. Implementation of Priority Queue using binary heap:

```
class ElementType {
    int priority;
    String data;

    public ElementType(int priority, String data) {
        this.priority = priority;
        this.data = data;
    }

    public boolean isHigherPriorityThan(ElementType e) {
        return priority < e.priority;
    }
}
```

⇒ 可以被改成 other 的 priority

Compare current obj. with given element

constructor of heap:

① Array:

```
public class BinaryHeap {
        private int currentSize;
        private ElementType arr[ ];

        public BinaryHeap (int capacity) {
                currentSize = 0;
                arr = new ElementType[capacity + 1];
        }
}
```

② Linked list:

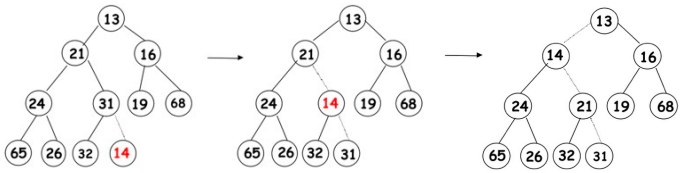
```
class Node {
    ElementType data;
    Node parent;
    Node left;
    Node right;

    public Node(ElementType data) {
        this.data = data;
        this.parent = null;
        this.left = null;
        this.right = null;
    }
}
```

```
public class LinkedListBinaryHeap {
    private Node root;
    private int size;

    public LinkedListBinaryHeap() {
        root = null;
        size = 0;
    }
```

↓

直接构建 binary tree.

(1) Insertion:



从下向上传递 ⇒ Worst case: Complexity = O(log n)

① Array:

13	21	26	24	31	19	68	65	26	32	14
1	2	3	4	5	6	7	8	9	10	11

$\lfloor \frac{11}{2} \rfloor = 5$

arr[5] > 14 ⇒ swap

13	21	26	24	14	19	68	65	26	32	31
1	2	3	4	5	6	7	8	9	10	11

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

arr[2] > 14 ⇒ swap

13	14	26	24	21	19	68	65	26	32	31
1	2	3	4	5	6	7	8	9	10	11

Function Insert(x):

```
if isFull( ) then:
    false
end if.
hole ← ++currentSize
while hole>1 and x.isHigherPriority(arr[hole/2]) do :
        swap arr[hole] and arr[hole/2]
        hole /= 2
end while
arr[hole] = x
```

② Linked list:

先用 traversal 找到 last level left most position.

```
if isEmpty then:
        root ← newNode
end if.
```

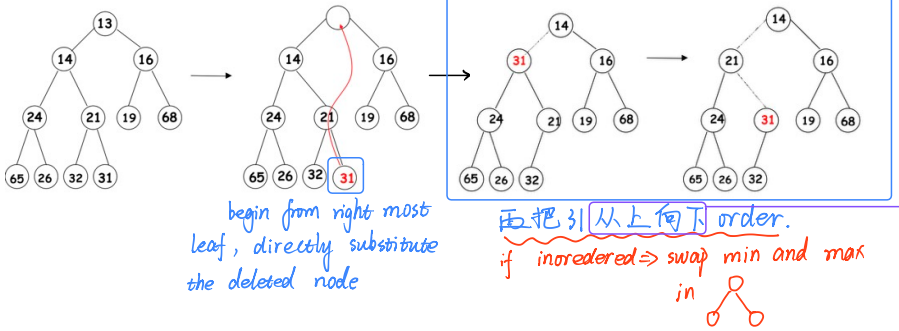
```
Queue<Node> queue = new LinkedList<>();
queue.add(root);
while (!queue.isEmpty()) {
    Node current = queue.poll();
    if (current.left == null) {
        current.left = newNode;
        newNode.parent = current;
        break;
    } else if (current.right == null) {
        current.right = newNode;
        newNode.parent = current;
        break;
    }
    queue.add(current.left);
    queue.add(current.right);
```

再 percolate up and swap:

```
while parent != null and newNode.isHigherPriority(parent) do
        swap newNode and parent
end while
size++
```

→ Compare child. 再 compare parent.

(2) Delete:



14<16
31>14 ⇒ swap 31 and 14

21<24
31>21 ⇒ swap 31 and 21.

begin from right most leaf, directly substitute the deleted node

再把 31 从上向下 order.

if inordered ⇒ swap min and max in

parentIndex · 2 = left child
right child = left child + 1

Warning:

1. Some node may have only one child
2. worst complexity: O(log n)

For a **heap with** `n` **elements**, the height of the tree is:

$$h = \log_2(n)$$

Since each step in **percolate up (insert)** or **percolate down (deleteMin)** moves **only one level up or down**, the maximum number of steps required is at most **the height of the tree,** `O(log n)`.

① Array:

1. **Replacing the root with the last element.**
2. **Removing the last element.**
3. **Percolating Down** to restore heap order.

→ delete 1st node.

```
DeleteMin():
    min = arr[1]
    last = arr[size--]
    hole = 1
    arr[1] = arr[size--]
    while hole·2 ≤ size do
        child ← hole·2
        if child != size and arr[child+1].isHigherPriority(arr[child]) then
            child ++
        if last.isHigherPriority(arr[child]) then:
            break
        arr[hole] = arr[child]   ⇒ if inordered, move child up.
        hole ← child
    end while
    arr[hole] = last
```

② Linked list:

```
private Node findLastNode() {
    Queue<Node> queue = new LinkedList<>();
    queue.add(root);
    Node lastNode = null;

    while (!queue.isEmpty()) {
        lastNode = queue.poll();
        if (lastNode.left != null) queue.add(lastNode.left);
        if (lastNode.right != null) queue.add(lastNode.right);
    }

    return lastNode;
}
```

```
private void percolateDown(Node node) {
    while (node.left != null) {
        Node smallest = node.left;
        if (node.right != null && node.right.data.isHigherPriorityThan(node.left.data
            smallest = node.right;
        }

        if (node.data.isHigherPriorityThan(smallest.data)) {
            break;
        }

        swap(node, smallest);
        node = smallest;
    }
}
```

```
public ElementType deleteMin() {
    if (root == null) {
        System.out.println("Heap is empty");
        return null;
    }

    ElementType minElement = root.data;  // Store

    if (size == 1) {
        root = null;  // If only one element, just
    } else {
        // Find last inserted node
        Node lastNode = findLastNode();

        // Replace root with last node's data
        root.data = lastNode.data;

        // Remove last node from its parent
        if (lastNode.parent.left == lastNode) {
            lastNode.parent.left = null;
        } else {
            lastNode.parent.right = null;
        }

        // Restore heap property by percolating do
        percolateDown(root);
    }

    size--;  // Reduce heap size
    return minElement;
}
```

4. Construct heap:

(1) Insert Node one-by-one => O(nlogn)

(2) Faster:

① 从上向下 Append inorderedly → O(1)

② percolate down:

start at last non leaf node.

Heapify each node to ensure that the subtree rooted at that node obeys the heap property.

⇓

individual steps:

Step 1: Build the Heap

```
for i = floor(n/2) - 1 downto 0 do
    heapify(arr, n, i)
```

Construct heap 从下向上.

- Start at the last non-leaf node.
- Heapify each node to ensure that the subtree rooted at that node obeys the heap property.

Step 2: Heap Sort

```
for i = n - 1 downto 1 do
    swap(arr[0], arr[i])
    heapify(arr, i, 0)
```

deduce heap into sorted array.

- Swap the first element (largest) with the element at the end of the heap.
- Reduce the heap size by one.
- Heapify the root to restore the heap property.

Step 3: Heapify Function

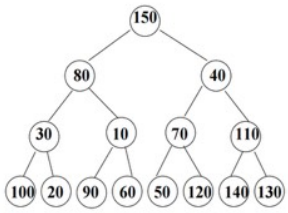
```
function heapify(arr, heapSize, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < heapSize and arr[left] > arr[largest]:
        largest = left

    if right < heapSize and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        swap(arr[i], arr[largest])
        heapify(arr, heapSize, largest)
```

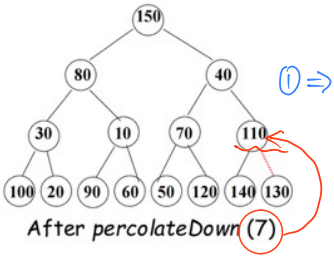
largest 与 i 换 entry

实际为交换后 i 的 index => 从上向下 update.

- Compare the node at index `i` with its left and right children.
- If one of the children is larger than the node, swap the node with the largest child.
- Recursively heapify the affected subtree.

eg:





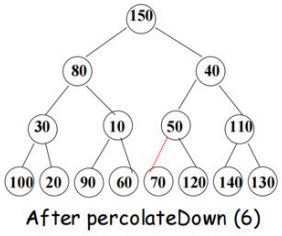
① => Completely filled.

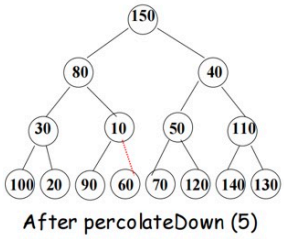
$N = 2^{3+1} - 1 = 15$ $i = 15//2 = 7$

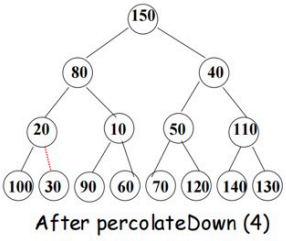
② 每 iterate 到一个 node.

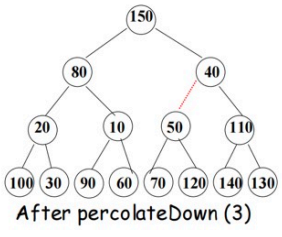
a. Compare its children

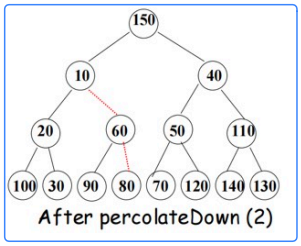
b. Compare node with smaller child.

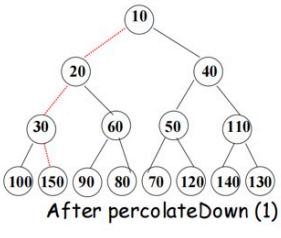






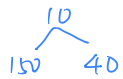




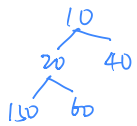


东要 150 下面有 child 且 inordered => continue to iterate.

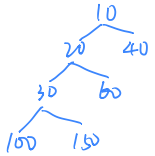
10 < 40, 150 > 10 => swap 150 and 10 =>



20 < 60, 150 > 20 => swap 150 and 20 =>



30 < 100, 150 > 30 => swap 150 and 30 =>



3. Heap sort:

```
public class HeapSort {

    // Main function to perform heap sort
    public static void heapSort(int[] arr) {
        int n = arr.length;

        // Build max heap (heapify bottom-up)
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arr, n, i);
        }

        // One by one extract elements from heap
        for (int i = n - 1; i >= 0; i--) {
            // Move current root (largest) to the end
            swap(arr, 0, i);
            // Call heapify on the reduced heap
            heapify(arr, i, 0);
        }
    }

    // Heapify a subtree rooted with node i, heapSize is the size of the heap
    private static void heapify(int[] arr, int heapSize, int i) {
        int largest = i;        // Initialize largest as root
        int left = 2 * i + 1;     // left child index
        int right = 2 * i + 2;    // right child index

        // If left child is larger than root
        if (left < heapSize && arr[left] > arr[largest]) {
            largest = left;
        }

        // If right child is larger than largest so far
        if (right < heapSize && arr[right] > arr[largest]) {
            largest = right;
        }

        // If largest is not root, swap and continue heapifying
        if (largest != i) {
            swap(arr, i, largest);
            heapify(arr, heapSize, largest);
        }
    }

    // Helper method to swap two elements in an array
    private static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    // Main method to test heap sort
    public static void main(String[] args) {
        int[] arr = {12, 11, 13, 5, 6, 7};
        System.out.println("Original array: " + Arrays.toString(arr));
        heapSort(arr);
        System.out.println("Sorted array: " + Arrays.toString(arr));
    }
}
```

Hashing:

1. Searching:

- Find items with keys matching a given search key
  - Given an array A, containing n keys, and a search key x, find the index i such as x = A[i]

Key	other data

(1) Directly addressing:

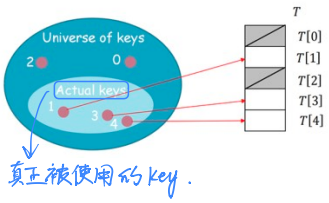
① Assumptions:
- Key values are distinct
- Each key is drawn from a universe U = {0, 1, . . . , m - 1}

② Idea:
- Store the items in an array, indexed by keys

- Direct-address table representation:
  - An array T[0 . . . m - 1]
  - Each corresponds to a key in U
  - T[k] stores a pointer to x (or x itself) with key k
  - T[k] may be empty

⇒ key is index
item is value.

- If the keys of the records are integers from $[U] = \{0,1,2,\cdots,U-1\}$
  - We maintain an array T of size U
  - To insert a record r: $T[r.key] = r$
  - To delete a record r: $T[r.key] = NULL$
  - To search the record with key k: return $T[k]$



真正被使用的 key.

② Limitation:

- The universe of the keys are usually very large
  - All the integers in the range $[0,2^{31}-1]$, i.e., $U = 2^{31}$
  - But the number of records may be far less than U

⇒ waste space.

(2)

- Solving search using:
  - Direct addressing
  - Ordered/unordered arrays
  - Ordered/unordered linked lists
  - Binary search tree

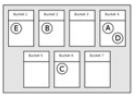
	Insert	Search
direct addressing (keys are the indexes)	O(1)	O(1) ⇒ key as index → O(1)
ordered array (keys are not indexes)	O(N)	O(logN) using binary search
ordered linked list	O(N)	O(N)
unordered array (keys are not indexes)	O(N)	O(N)
unordered linked list	O(1)	O(N)
binary search tree	O(logN)	O(logN)

insert at head

2. Hashing:

(1)

- In hashing, the element is stored in slot $h(k)$, i.e., $T[h(k)]$, where h is a hash function
  - Assume keys are integers in the range of $[0, U-1]$ ⇒ key set
  - Denote by $[x]$ the set of integers from 0 to $x-1$ ⇒ hash value set
  - A hash function h is a function from $[U]$ to $[m]$
    - For any integer k, $h(k)$ returns an integer in $[m]$
    - The value $h(k)$ is called the hash value of k
    - $U > m$



① hashing function

In hashing, a **hash function** maps **keys** to **indices** in a hash table. This mapping ensures efficient data retrieval and storage.

The **mapping function** can be defined as:

$$h(k) = index$$

where:

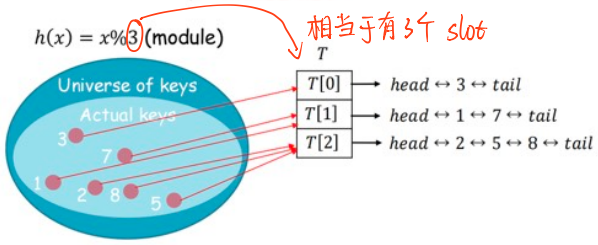
- k is the key,
- $h(k)$ is the hash function,
- $index$ is the computed position in the table.

3. Operations:

- createTable(sizem): create a hash table of size sizem
- search(hashtable, key): return the value if the hashtable contains the key, otherwise return NULL
- insert(hashtable, key, value): insert the key value pair to the hashtable
- delete(hashtable, key): remove the key and values stored on the hashtable
- isFull(hashtable): return true if no element can be inserted to hashtable, otherwise return false

(1) Using chaining to deal with collision:

Chaining: we place all elements that hash to the same slot into the same linked list



① Create hash table:

Algorithm: createTable(sizem)

```
1 hashtable←allocate an array of size sizem
2 for i from 0 to sizem-1
3     hashtable[i] <- allocate an empty linkedlist
4 return hashtable
```

② Search:

Algorithm: search(hashtable, key)

```
hashid = h(key)
node=hashtable[hashid].head.next
while node != hashtable[hashid].tail
    if node.data.key == key
        return node.data.value
    node = node.next
return NULL
```

③ Insert:

- L_i: the linked list containing elements hashes to i
- Find the linked list $L_{h(key)}$
- Insert r to the end of $L_{h(key)}$

Algorithm: insert(hashtable, key, record)

```
1 hashid = h(key)
2 keyValuePair← (key, record)
3 Insert_linklist(hashtable[hashid], keyvaluepair)
```

Or simply:

```
Insert_linklist(hashtable[hashid], key, record)
```

⇒ Find hash valued Linked list, then implement LL insertion.

④ Deletion:

- Retrieve linked list $L_{h(key)}$
- Search from $L_{h(key)}$ and get the node containing key key, and delete this node

Algorithm: delete(hashtable, key)

```
1 hashid = h(key)
2 node=hashtable[hashid].head.next
3 while node != hashtable[hashid].tail
4     if node.data.key == key
5         break
6     node = node.next
7 if node != hashtable[hashid].tail
8   delete_linkedlist(hashtable[hashid], node)
```

⇒ garantee that a matching node is found in LL

eg: Given a hash function $h(k) = k\%7$, show the hash table after inserting 1, 3, 9, 20, 30, 51, 25, 23, 36

- T[0]:
- T[1]: 1, 36
- T[2]: 9, 30, 51, 23
- T[3]: 3
- T[4]: 25
- T[5]:
- T[6]: 20

⑤ Complexity analysis:

- Load factor α: the average number of elements stored in a chain
  - If the hash table has size m and stores n elements in it, then $\alpha = \frac{n}{m}$
  - Assumption: uniform hashing of $h(k)$ ⇒ on average, one bucket contains α keys
    - Elements are equally likely to hash into any of the m slots
    - L_i: the linked list containing the elements hashes to i

Theorem 1: In a hash table with collision resolved by chaining, the search/insertion/deletion takes $O(1+\alpha)$ time in expectation if $h(k)$ is uniform hashing.

① 定位 slot: By choosing m s.t $\frac{n}{m} = \alpha = O(1)$, query time is $O(1)$
② iterate LL: $O(\alpha)$
time cost to compute hash function is $O(1)$

(2) Using open addressing.

- All elements are stored in the hash table
  - Unlike chaining, no elements are stored outside the table
  - The hash table may "fill up" such no insertion can be made
    - Load factor α is always smaller than 1 → reserve slots for collision

⇒ Using probing:

- For insertion, we examine, or probe, the hash table until an empty slot is found to put the key
  - How to probe the hash table?
    - Linear probing, double Hashing
    - Quadratic probing (not discussed in the lecture)

Linear probing:

①
- For insertion, we probe $h(k), h(k)+1, h(k)+2, \cdots, h(k)+(m-1)$ one by one until we find an empty slot, and insert the record to this slot
  - Formally we probe $h(k,i) = (h(k)+i)\%m$ from $i=0$ to $i=m-1$ until we find an empty slot to insert ⇒ % 是为防止 index 超出 hash 的 size:

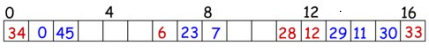
如果 $h(k)=7$，那么：
- $(7+0) \mod 10 = 7$
- $(7+1) \mod 10 = 8$
- $(7+2) \mod 10 = 9$
- $(7+3) \mod 10 = 0$ 这里回到表头，形成环状结构
- $(7+4) \mod 10 = 1$

→ circular probing

- When searching for a record with a certain key,
  - Compute $h(k)$
  - Examine the hash table buckets in order $T[h(k,i)]$ for $0 \le i \le m-1$ until one of the following happens:
    - $T[h(k,i)]$ has the record with key equal to k
    - $T[h(k,i)]$ is empty, then no record contains key k in the hash table

eg:

- If we have a hash table with size $m=17$ and the hash function $h(k) = k \% 17$
  - Assume that we use linear probing to address collisions
    - Given the following hash table, show the hash table after inserting 21 and 13
    - Given the following hash table, show the records examined when searching for 14



14 % 17 → 11
15 % 17 → 30
16 % 17 → 33
17 % 17 → 34
1 % 17 → 0
2 % 17 → 45
3 % 17 → empty ⇒ 14 not in hash.

② Deletion is not supported.

eg: $h(k) = k \% 7$:

Current Hash Table

```
Index:  0  1  2  3  4  5  6
Data:   _  _  _  10 24 31 _
```

Now, we **delete key 10** at index **3**.

What Happens?

If we simply remove **10** and mark index **3** as empty:

```
Index:  0  1  2  3  4  5  6
Data:   _  _  _  _  24 31 _
```

Now, if we **search for 24**:

- We start at $h(24) = 3$.
- **Index 3 is empty!**
  - Since we assume a key should be in its correct slot, **we falsely conclude that 24 is not in the table**, even though it's at index **4**.

③ Defficiency of linear probing : ◦ Long sequence of occupied slots, which degrades the query efficiency

(2) Double hashing :

① ◦ We have an additional hash function $h' > 0$ → 加大跨越格子 step.
◦ Insertion: we probe $h(k,i) = (h(k) + i \cdot h'(k)) \% m$ one by one for i from 0 to $m-1$ until an empty slot is found
◦ Search: we search $h(k,i)$ for i from 0 to $m-1$ until one of the following happens:
- $T[h(k,i)]$ has the record with key equal to k
- $T[h(k,i)]$ is empty, then no record contains key k in the hash table

eg: ▸ If we have a hash table with size $m = 17$ and the hash function $h(k) = k \% 17$ and $h'(k) = 1 + k\%5$
▸ Insert the following records: 6, 12, 34, 29, 28 using double hashing to resolve collisions
◦ $h(6,0) = 6\%17 = 6$ → empty, insert here
◦ $h(12,0) = 12\%17 = 12$ → empty, insert here
◦ $h(34,0) = 34\%17 = 0$ → empty, insert here
◦ $h(29,0) = 29\%17 = 12$ → not empty,
try $h(29,1) = (12 + 1 + 29\%5)\%17 = 0$, not empty,
try $h(29,2) = (12 + 2 \cdot (1 + 29\%5))\%17 = 5$, empty, insert here
◦ $h(28,0) = 28\%17 = 11$ → empty, insert here

0				4				8				12				16
34					29	6						28	12			

eg: ▸ Consider a hash table with size $m = 17$ and the hash function $h(k) = k \% 17$ and $h'(k) = 1 + k\%5$
◦ Assume that we use double hashing,
- Given the following hash table, show the hash table after inserting 11 and 27
- After inserting 11 and 27, show the records examined when searching for 23

11%17 = 11
(11+1+11%5) = 13
27%17 = 10

0				4				8				12				16
34					29	6					27	28	12	11		

23%17 = 6,
(6+1+23%5) %17 = 10
(6+2(1+23%5)) %17 = 14, empty

② ▸ Load factor α: If the hash table T has size m and we store n elements in the hash table
◦ $\alpha = \frac{n}{m}$ If $\alpha \to 1 \Rightarrow$ table close to be full $\Rightarrow$ collision gradually ↑

▸ Assumption: uniform hashing of $h(k,i)$
◦ The probing sequence $\langle h(k,0), h(k,1), h(k,2), \cdots, h(k,m-1)\rangle$ used to insert or search for each key k is equally likely to be any permutation of $\langle 0,1,2,\cdots,m-1\rangle$

Theorem 2: In a hash table with collision resolved by open addressing, the search takes $O(\frac{1}{1-\alpha})$ time in expectation if $h(k,i)$ is uniform hashing.

$\Rightarrow$

- As long as $\alpha < 1$ (i.e., the table isn't completely full), you can still expect a **finite** average search time.
- However, as $\alpha \to 1$, $\left(\frac{1}{1-\alpha}\right)$ grows very large, reflecting the fact that if the table is nearly full, finding a slot (or confirming a key isn't there) can take many probes.

Comparison of open addressing and chaining:

① Pros of chaining
◦ Less sensitive to hash functions and load factors (α can be larger than 1), while open addressing requires to avoid long probes, and its load factor $\alpha < 1$
◦ Support deletion, while open addressing is difficult to support deletion

② Pros of open addressing
◦ Usually much faster than chaining

4. Hashing function: good one should satisfy uniform hashing.

① ◦ Division is effective in practice without any theoretical guarantee on $O(1)$ query time

◦ Each key is equally likely to hash to any of the m slots, independent of where other keys will hash to

② ◦ Universal hashing provides theoretical guarantees on $O(1)$ query time

(1) Division: mapping key k to a slot in hash table size m.

- $h(k) = k \bmod m = k \,\%\, m$
  - If $m = 10^p$, then $h(k)$ only uses the lowest-order p digits of the key value k
    - We cannot use all digits to generate hash keys, and we should not choose such an m
    - In a similar way, we should not choose $m = 2^p$
  - Option of m: choose a prime number not close to the power of 2 or 10
    - Example: $U = 2000$, we choose $m = 701$

⇒

Loss of Higher-Order Information:

- **Example:** Consider two keys, 123456 and 223456.
- If you compute $h(k) = k \mod 1000$, both keys yield:

$$123456 \mod 1000 = 456 \quad \text{and} \quad 223456 \mod 1000 = 456.$$

- Even though the keys are different in the higher-order digits (123 vs. 223), this information is completely ignored. Both keys hash to 456.

(2) Universal hashing:

- Let $\mathcal{H}$ be a family of hash functions from $[U]$ to $[m]$
- $\mathcal{H}$ is called universal if the following condition holds:

> Let k_1, k_2 be two distinct integers from $[U]$. By picking a function $h \in \mathcal{H}$ uniformly at random, we guarantee that
> $$\Pr[h(k_1) = h(k_2)] \leq \frac{1}{m}$$

  - Then, we choose one from $\mathcal{H}$ uniformly at random and use it as the hash function h for all operations

⇒ Using h, we have query time of chaining: $O(1+\alpha)$

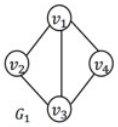
① Design $\mathcal{H}$:

- Pick a prime number p such that $p > U$
- For every $a \in \{1,2,\cdots,p-1\}$, and every $b \in \{0,1,2,\cdots,p-1\}$
  - $h_{a,b}(k) = ((a \cdot k + b) \bmod p) \bmod m$
- This defines $p \cdot (p-1)$ functions, which constitutes $\mathcal{H}$

Graph

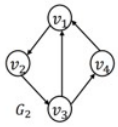
1. A graph G is defined as a pair of (V, E) where:
   - V is the set of objects, each of which is called a node (or a vertex)
   - E is the set of edges, where each edge connects two nodes u and v
   - $n = |V|$, $m = |E|$

(1) A graph is an undirected graph, if there is no order in an edge



- We use (u, v) to represent an edge
  - (u, v) and (v, u) are the same edge
- In many social networks, e.g., Facebook

- A graph is a directed graph, if there is an order in an edge



- We use $\langle u, v \rangle$ to represent an edge
  - $\langle u, v \rangle$ and $\langle v, u \rangle$ represent two different edges
- In some social networks, e.g., Twitter

(2) Neighbor: v is called a neighbor of u, if there is an edge between v and u

- In directed graphs, v is called the out-neighbor of u is there is an edge $\langle u, v \rangle$; v is called the in-neighbor of u is there is an edge $\langle v, u \rangle$

$u \to v$ $\quad$ $v \to u$

(3) Degree: the degree $d(v)$ of a node v is the number of neighbors of this node

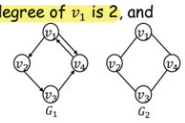
- In directed graphs, the out-degree $d_{out}(v)$ of a node v is the number of out-neighbors of this node; the in-degree $d_{in}(v)$ of a node v is the number of in-neighbors of this node

$\Rightarrow$ ① If directed, degree = din + dout
② If undirected, degree = num of edges.

- A graph is connected if there is a path from every vertex to every other vertex
  - A tree is a connected, acyclic "undirected" graph

eg:

- In G_1, the degree of v_1 is 3, the out-degree of v_1 is 2, and the in-degree of v_1 is 1
- In G_2, the degree of v_3 is 2
- The number of edges in G_1 is 5
  - The degrees of v_1, v_2, v_3 and v_4 are 3, 2, 2, and 3, respectively
  - In G_1, $\sum_{v \in V} d(v) = 3 + 2 + 2 + 3 = 10 = 2 \cdot 5 = 2 \cdot m$
  - In G_1, $\sum_{v \in V} d_{out}(v) = 2 + 1 + 1 + 1 = 5 = m$, $\sum_{v \in V} d_{in}(v) = 1 + 1 + 1 + 2 = 5 = m$

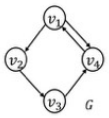


(4) A path from node u to node v in a graph G is a sequence of nodes, $(u, v_1, \ldots, v_k, v)$ such that there exist a sequence of edges

- $((u, v_1), (v_1, v_2), \ldots, (v_k, v))$ if G is an undirected graph
- $((u, v_1), (v_1, v_2), \ldots, (v_k, v))$ if G is a directed graph

- A simple path is a path in which all nodes except the first and last are distinct
- A cycle is a simple path in which the first and the last nodes are the same

- Example:
  - (v_1, v_3) is not a path
  - $(v_1, v_2, v_3, v_4, v_1, v_4)$ is a path but not a simple path
  - (v_1, v_2, v_3, v_4) is a simple path but not a cycle
  - $(v_1, v_2, v_3, v_4, v_1)$ is a simple path and a cycle

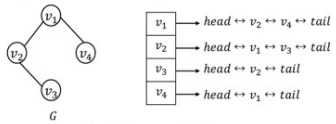


2. Representation of graph:

(1) Adjacency list: a hash table

① undirected:

- Adjacency list for undirected graph
  - Each node $v \in V$ is associated with a linked list that stores all neighbors of v; we map an ID for each node

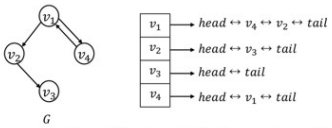


  - Space cost: $O(n + m)$, where n is the number of nodes and m is the number of edges

using linked list store neighbors.

② Directed:

- Adjacency list for directed graph
  - Each node $v \in V$ is associated with a linked list that stores all out-neighbors of v

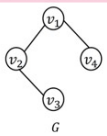


  - Space cost: $O(n + m)$, where n is the number of nodes and m is the number of edges

(2) Adjacency matrix:

① Undirected:

- Adjacency matrix for undirected graph
  - A $n \times n$ two dimensional matrix A where $A[u][v] = 1$ if $(u, v) \in E$, or 0 otherwise



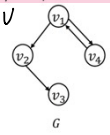
	v_1	v_2	v_3	v_4
v_1	0	1	0	1
v_2	1	0	1	0
v_3	0	1	0	0
v_4	1	0	0	0

  - A is symmetric
  - Space cost: $O(n^2)$ where n is the number of nodes

② Directed:

- Adjacency matrix for directed graph
  - A $n \times n$ two dimensional matrix A where $A[u][v] = 1$ if $(u, v) \in E$, or 0 otherwise

$u \to v$



	v_1	v_2	v_3	v_4
v_1	0	1	0	1
v_2	0	0	1	0
v_3	0	0	0	0
v_4	1	0	0	0

  - A may not be symmetric
  - Space cost: $O(n^2)$ where n is the number of nodes

(3) Time Cost:

- Adjacency list:
  - Space: $O(n+m)$, save space if the graph is sparse, i.e., $m \ll n^2$
  - Check the existence of an edge (u,v): $O(k)$ time where k is the number of neighbors of u ⇒ 耗时check neighbor existence
  - Retrieve the neighbors of a node: $O(k)$ time
  - Add a node: $O(n)$
  - Delete a node: $O(n+m)$ ⇒ ① Delete ② traverse and remove any edges pointed to the deleted node.
  - Add an edge: $O(1)$ ⇒ append to adjacency list ⇒ constant time
  - Delete an edge: $O(k)$ ⇒ delete v from u's list ⇒ k = length of list.
- Adjacency matrix:
  - Space consumption: $O(n^2)$
  - Check the existence of an edge (u,v): $O(1)$ time
  - Retrieve the neighbors of a node: $O(n)$ time
  - Add/delete a node: $O(n^2)$, (create a new matrix)
  - Add/delete an edge: $O(1)$

(4) Code:

`g.addEdge(v1, v2);`	adds an edge between two vertexes
`g.addVertex(name);`	adds a vertex to the graph
`g.clear();`	removes all vertexes/edges from the graph
`g.getEdgeSet()` `g.getEdgeSet(v)`	returns all edges, or all edges that start at **v**, as a Set of pointers
`g.getNeighbors(v)`	returns a set of all vertices that **v** has an edge to
`g.getVertex(name)`	returns pointer to vertex with the given name
`g.getVertexSet()`	returns a set of all vertexes
`g.isNeighbor(v1, v2)`	returns `true` if there is an edge from vertex **v1** to **v2**
`g.isEmpty()`	returns `true` if queue contains no vertexes/edges
`g.removeEdge(v1, v2);`	removes an edge from the graph
`g.removeVertex(name);`	removes a vertex from the graph
`g.size()`	returns the number of vertexes in the graph
`g.toString()`	returns a string such as `"{a, b, c, a -> b}"`

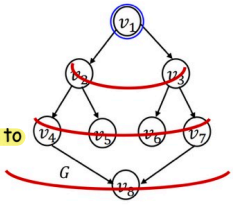
20

3. BFS:

(1) Intuition of BFS

- Given a source node s, always visit nodes that are closer to the source s first before visiting the others

- The result is not unique, if we do not define an order among out-going edges from a node
  - Possible results
    - $v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$
    - $v_1, v_3, v_2, v_7, v_6, v_5, v_4, v_8$
  - If we impose an order by going from smaller id to larger id, then the result will be unique



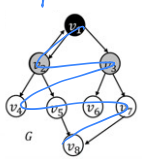
(2) steps:

- At the beginning, color all nodes to be white
- Create a queue Q, enqueue the source s to Q, and color the source to be gray (meaning s is in the queue)
- Repeat the following until queue Q is empty
  - Dequeue from Q, let the node be v
  - For every out-neighbor u of v that is still white
    - Enqueue u into Q, and color u to gray (to indicate u is in queue)
  - Color v to be black (meaning v has finished)

每次处理一个 Queue 的 peek 的 neighbors.

eg1: Example:

- Assume the source is v_1
  - ① $Q = (v_1)$
  - ⇓ After dequeuing v_1
  - $Q = (v_2, v_3)$



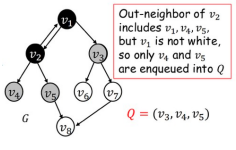
⇒ level order.

② Deal with v_2's neighbor:

i. Dequeue v_2

ii. Check unprocessed out neighbors: v_1 has been processed.

$Q = (v_3, v_4, v_5)$



③ i. Deq: v_3

ii. $Q = (v_4, v_5, v_6, v_7)$

④ i. Deq: v_4

$Q = (v_5, v_6, v_7)$

⑤ $Q = (v_6, v_7, v_8)$

⑥ $Q = (v_7, v_8)$

⑦ $Q = (v_8)$

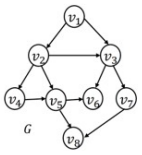
⑧ $Q = ()$

⇒ Traversal:

$v_1 \to v_2 \to v_3 \to v_4 \to v_5 \to v_6 \to v_7 \to v_8$

eg2: Given the following graph G, show the process of the BFS if the source is v_2
- Assume that smaller vertex ids should be visited before larger vertex ids

v_1 cannot be found
⇒ this graph is not strongly connected



① Q=(v_2)
② Q=(v_3, v_4, v_5), v_2
③ Q=(v_4, v_5, v_6, v_7), $v_2 \to v_3$
④ Q=(v_5, v_6, v_7), $v_2 \to v_3 \to v_4$
⑤ Q=(v_6, v_7, v_8) $v_2 \to v_3 \to v_4 \to v_5$
⑥ Q=(v_7, v_8) $v_2 \to v_3 \to v_4 \to v_5 \to v_6$
⑦ Q=(v_8) $v_2 \to v_3 \to v_4 \to v_5 \to v_6 \to v_7$
⑧ Q=( ) $v_2 \to v_3 \to v_4 \to v_5 \to v_6 \to v_7 \to v_8$

(3) Code:

```
BFS(Graph, start):
    n ← Graph.size
    color ← [0] x Graph.size
    Q ← Queue
    Q.enqueue(start)
    color[start] ← 1
    while Q:
        V = Q.dequeue
        for u in sorted(Graph[v]):
            if color[u] = 0:
                Q.enqueue(u)
                color[u] = 1
        color[v] = 2.
```

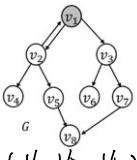
(4) Complexity:
- When a node v is dequeued,
  - We examine all of its neighbors (check their color), enqueue them and color them to gray if they are white
  - After that, we color v as black
  - This incurs $c(1 + d_{out}(v))$ costs for node v where c is a constant (if we use adjacency list to represent the graph)
- Each node is dequeued at most once
  - Why? we enqueue a node at most once
  - If it is in the queue, its color is gray and we will not further enqueue it
- Therefore, the total running time with adjacency list representation is:
  - $\sum_{v \in V} c(1 + d_{out}(v)) = c(n + m) = O(n + m)$

4. DFS:

(1) Step:
- Initialization:
  - At the beginning, color all nodes to be white
  - Create a stack S, push the source s to S, and color the source to be gray (meaning s is in the stack)
- Repeat the following until S is empty
  - Get the top node, denoted as v, on stack S, do not pop v
  - If v still has white out-neighbors
    - Let u be such a white out-neighbor of v ⇒ 一次只进一个 neighbor
    - Push u to S, and color u to gray
  - Otherwise (v has no white out-neighbors)
    - Pop v and color it as black (meaning that v has finished)

eg: Example:
- Assume that v_1 is the source

① S = [v_1]
② S=(v_1, v_2)
③ S=(v_1, v_2, v_4) black
④ S=(v_1, v_2)
⑤ S=(v_1, v_2, v_5)
⑥ S=(v_1, v_2, v_5, v_8)

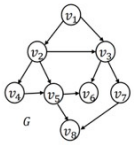


⑦ S=(v_1, v_2, v_5) → (v_1, v_2) → (v_1)
⑧ S=(v_1, v_3)
⑨ S=(v_1, v_3, v_6) → (v_1, v_3)
⑩ S=(v_1, v_3, v_7) → (v_1, v_3) → (v_1) → ( )

⇒ $v_1 \to v_2 \to v_4 \to v_5 \to v_8 \to v_3 \to v_6 \to v_7$

eg2: Given the following graph G, show the process of the DFS if the source is v_2
- Assume that smaller vertex ids should be visited before larger vertex ids

① S=(v_2)
② S=(v_2, v_3) → (v_2, v_3, v_6) → (v_2, v_3, v_6, v_7) → (v_2, v_4, v_7, v_8)
③ S=(v_2, v_4) → (v_2, v_4, v_5)



⇒ $v_2 \to v_3 \to v_6 \to v_7 \to v_8 \to v_4 \to v_5$.

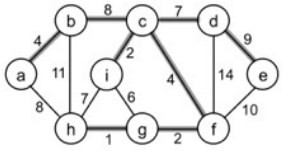
(2) Code:

```
Algorithm 1: DFS(V, E, s)
color[]← initialize an array of size n with all zero values
// Use 0 : white, 1 : gray, and 2: black
S ← an empty stack
S.push(s) and print(s)
color[s] ← 1
while !S.isEmpty()
   v ← S.top()
   if v still has white-neighbor u
         S.push(u) and print(u)
         color[u]=1
   else
       color[v] ← 2
       S.pop()
Free color array if necessary
```

Minimum Spanning Tree

1. Spanning tree
   - A tree (i.e., connected, acyclic graph) which contains all the vertices of the graph

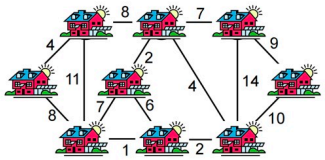
2. (1) Minimum spanning tree (MST)
   - Spanning tree with the minimum sum of weights
   - If a graph is not connected, then there is an MST for each connected component of the graph



(2) Properties: MST { not unique; has no cycle

eg:
- Given an undirected weighted graph, find a set of edges such that: (1) everyone stays connected and (2) the total cost is minimum

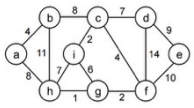
Find $T \subseteq E$ such that:
1. T connects all vertices
2. $w(T) = \Sigma_{(u,v)\in T} w(u, v)$ is minimized



Sol:
- Grow a set A of edges (initially empty)
- Incrementally add edges to A such that they would belong to an MST

⇒

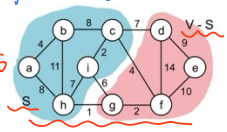
1. $A \leftarrow \varnothing$
2. **while** A is not a spanning tree
3. **do** find an edge (u, v) that is safe for A
4. $A \leftarrow A \cup \{(u, v)\}$
5. **return** A



⇒

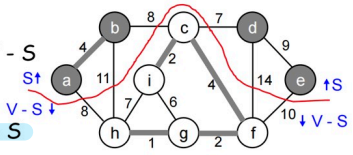
How to find a safe edge:
- Let's look at edge (h, g)
  - Is it safe for A initially?

分成2部分



- Yes. Why?
  - Let S ⊂ V be any set of vertices that includes h but not g (so that g is in V - S)
  - In any MST, there has to be one edge (at least) that connects S with V - S
  - Why not choose the edge with minimum weight (h, g)?

(3) Def:
- A **cut** (S, V - S) is a partition of vertices into two disjoint sets S and V - S
- An edge **crosses** the cut (S, V - S) if one endpoint is in S and the other in V - S



- A cut **respects** a set A of edges ⇔ no edge in A crosses the cut

⇒ A的所有 edges 都在 S 或 V-S 中.

- ✅ You can have edges (x_1, x_2) in A where both $x_1, x_2 \in S$
- ✅ You can have edges (y_1, y_2) in A where both $y_1, y_2 \in V - S$
- ❌ You **cannot** have an edge $(u, v) \in A$ with $u \in S$ and $v \in V - S$

- An edge is a **light edge** crossing a cut ⇔ its weight is minimum over all edges crossing the cut
  - Note that for a given cut, there can be > 1 light edges crossing it

(4) Let A be a subset of some MST, (S, V - S) be a **cut** that respects A, and (u, v) be a **light edge** crossing (S, V-S). Then (u, v) is safe for A.

⇒

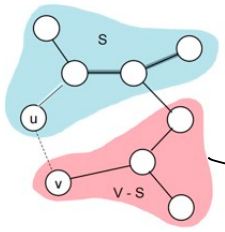
1. **MST 和 A**：你已经构建了最小生成树的一部分，边集合 A 是这部分边。A 中的边可以看作是你已经完成的工作。
2. **割 (S, V - S)**：想象你把图的节点分成了两个部分 S 和 $V - S$，比如你把图中的节点分成两组，组 S 和组 T。割就是这条“分界线”，把两个部分分开。
3. **尊重 A 的割**：割 $(S, V - S)$ 的尊重意味着，集合 A 中的边都完全位于组 S 或组 T 中，没有一条边跨越这个分界线。
4. **轻边 (u, v)**：轻边是连接这两个组的最便宜的边。你可以想象，割将两个部分分开，而轻边是连接两个部分的最短路径。在所有跨越割的边中，**(u, v)** 是权重最小的那一条。
5. **安全边**：这里的“安全边”意味着，如果你把轻边 (u, v) 加入到集合 A 中，它不会破坏 A 已经是最小生成树的事实。换句话说，加入 (u, v) 后，你依然能够保证图中的最小生成树特性：①没有形成回路，②且代价最小。

proof:

Proof:

- Let T be an MST that includes A
  - edges in A are shaded
- Case1: If T includes (u,v), then it would be safe for A
- Case2: Suppose T does not include the edge (u, v)
  - **Idea**: construct another MST T' that includes A + {(u, v)}

Assuming (u,v) is lightest edge



⇒ Cut (S, V-S) respect ⇒ cut 未改变A的连接情况.

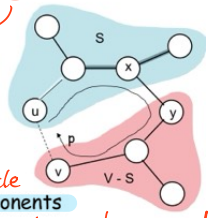
→ Assuming this is a MST

⇒ Construct T'

If T don't contain ⇒ 但一定在(u,v)中间有其他path，T中原本的edge不是min的.
就换一个 min edge

- T contains a unique path p between u and v
- Path p must cross the cut (S, V - S) at least once: let (x, y) be that edge
- Let's remove (x, y) ⇒ breaks T into two components
- Adding (u, v) reconnects the components
  T' = T - {(x, y)} + {(u, v)}

① avoid cycle ② update least edge

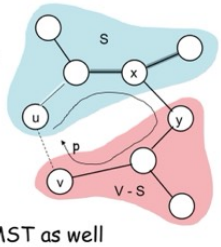


⇒ Prove T' is MST

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

Have to show that T' is an MST:

- (u, v) is a light edge
  ⇒ w(u, v) ≤ w(x, y)
- w(T') = w(T) - w(x, y) + w(u, v)
  ≤ w(T)
- Since T is an MST,
  w(T) ≤ w(T') ⇒ T' must be an MST as well



3. Finding MST: Prim's algorithm:

(1)

- The edges in set A always form a single tree
- Starts from an arbitrary "root": V_A = {a}
- At each step:
  - Find a light edge crossing (V_A, V - V_A)
  - Add this edge to A
  - Repeat until the tree spans all vertices

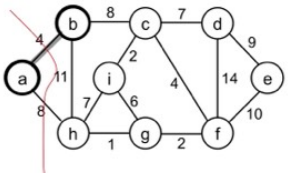
⇒

using Q to implement.

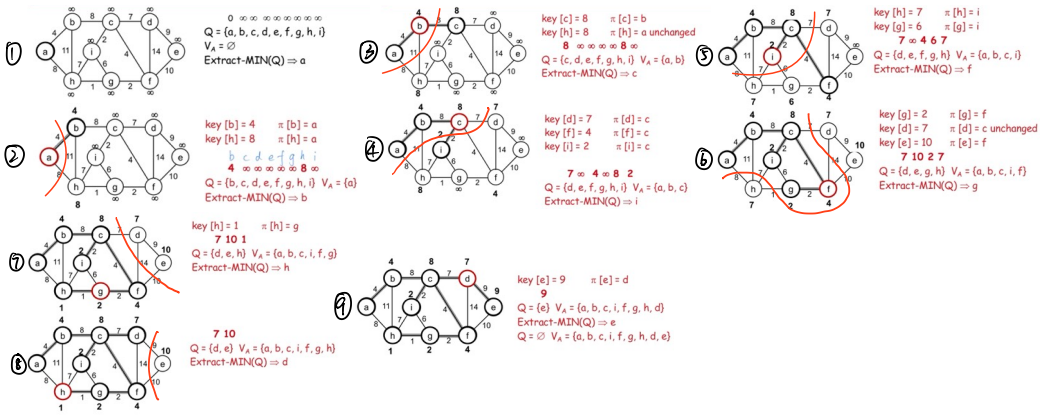
- Use a priority queue Q to include vertices not in the tree, i.e., (V - V_A)
  - V_A = {a}, Q = {b, c, d, e, f, g, h, i}
- Associate a key to each vertex v in Q:
  - key[v] = minimum weight of any edge (u, v) connecting v to V_A
  - key[v] = ∞, if v is not adjacent to any vertices in V_A

key[a] = min(w_1, w_2)

- After adding a new vertex to V_A, update the weights of all vertices adjacent to it
  - E.g., after adding a, k[b]=4 and k[h]=8



Sol.:



(2) Code:

```
Prim(G, V, root):
  Q ← ∅;
  for each vertex u in G.V:
      key[u] = ∞      → 存 weight
      π[u] = null;    → 存 parent
      insertHeap(u)
  key[root] = 0
  deleteHeap(u)
```

initialization

① Complexity analysis:

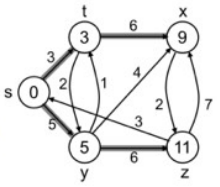
```java
class Node {
    int vertex;
    int key;
    Node(int v, int k) {
        vertex = v;
        key = k;
    }
}

// Prim's algorithm with Min-Heap
void primMST(Graph graph, int start) {
    int V = graph.numVertices;
    int[] key = new int[V];           // key values used to pick min weight edge
    int[] parent = new int[V];        // store MST
    boolean[] inMST = new boolean[V];  // true if vertex is included in MST

    // Initialize all keys as infinite
    for (int i = 0; i < V; i++) {
        key[i] = Integer.MAX_VALUE;
        parent[i] = -1;
    }

    // Use PriorityQueue as Min-Heap
    PriorityQueue<Node> pq = new PriorityQueue<>(Comparator.comparingInt(n -> n.key));

    // Start from the start vertex
    key[start] = 0;
    pq.add(new Node(start, key[start]));

     while (!pq.isEmpty()) {
        Node u = pq.poll();  // Extract min
        int uVertex = u.vertex;
        inMST[uVertex] = true;

        // Traverse adjacent vertices
        for (Edge edge : graph.adj[uVertex]) {
            int v = edge.dest;
            int weight = edge.weight;

            // If v is not in MST and weight is less than current key[v]
            if (!inMST[v] && weight < key[v]) {
                key[v] = weight;
                parent[v] = uVertex;
                pq.add(new Node(v, key[v]));  // simulate decrease-key
            }
        }
    }

    // Print MST
    for (int i = 0; i < V; i++) {
        if (parent[i] != -1)
            System.out.println(parent[i] + " - " + i + " : " + key[i]);
    }
}
```

4. Finding MST: kruskal's algorithm.

Weighted Graph Shortest Path

1. Set up:

- **Input:**
  - Directed graph G = (V, E)
  - Weight function w : E → **R**



- **Weight of path** $p = \langle v_0, v_1, \ldots, v_k \rangle$

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- **Shortest-path weight** from u to v:

$$\delta(u, v) = \min \begin{cases} w(p) : u \overset{p}{\rightsquigarrow} v & \text{if there exists a path from u to v} \\ \infty & \text{otherwise} \end{cases}$$

- **Note:** there might be multiple shortest paths from u to v

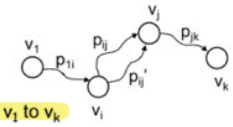
2. Types:

- **Single-source shortest paths**
  - G = (V, E) ⇒ find a shortest path from a given source vertex s to each vertex v ∈ V
- **Single-destination shortest paths**
  - Find a shortest path to a given destination vertex t from each vertex v
  - Reversing the direction of each edge ⇒ single-source
- **Single-pair shortest path**
  - Find a shortest path from u to v for given vertices u and v
- **All-pairs shortest-paths**
  - Find a shortest path from u to v for every pair of vertices u and v

3. Optimal Substructure theorem:

Given:

- A weighted, directed graph G = (V, E)
- A weight function w: E → **R**,
- A shortest path $p = \langle v_1, v_2, \ldots, v_k \rangle$ from v_1 to v_k
- A subpath of p: $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$, with $1 \le i \le j \le k$



Then: p_{ij} is a shortest path from v_i to v_j

Proof: $p = v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$ → 中间的 path default as shortest.

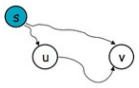
$w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$

Assume ∃ p_{ij}' from v_i to v_j with $w(p_{ij}') < w(p_{ij})$ Suppose there's shorter path.

⇒ $w(p') = w(p_{1i}) + w(p_{ij}') + w(p_{jk}) < w(p)$ contradiction!

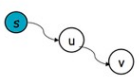
(1) Triangle inequality:

- For all (u, v) ∈ E, we have:
  $\delta(s, v) \le \delta(s, u) + \delta(u, v)$



Proof?

- If u is on the shortest path to v we have the equality sign



(2) Shortest path cannot contains cycle

4. For unweighted graph. ⇒ using BFS

(1)

- **A simple algorithm**
  1. Mark the starting vertex, s
  2. Find and mark all unmarked vertices adjacent to s
  3. Find and mark all unmarked vertices adjacent to the marked vertices
  4. Repeat Step 3 until all vertices are marked

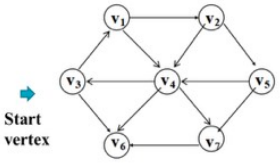
- For each vertex, keep track of
  - whether the adjacent vertex has been marked
  - its distance from s(d_v)
  - previous vertex of the path from s(p_v)

⇒ 用 BFS track { vertex 与 s 的 distance / path.

(2) Code:

```
unweightShortest(Vertex q):
    Queue q;
    For v: {dv = -∞}
    ds = 0
    q.enqueue(s)
    while(!q.isEmpty()):
        v = q.dequeue()
        for w adjacent to v:
            if dw = -∞:   因为w可能与多个 vertex adjacent
            dw = dv + 1
            Pw = v
            q.enqueue(w)
```

eg:



v	Known	d_v	p_v
v_1	F	∞	0
v_2	F	∞	0
v_3	F	0	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

v_3 Dequeued

v	Known	d_v	p_v
v_1	F	1	v_3
v_2	F	∞	0
v_3	T	0	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	1	v_3
v_7	F	∞	0
Q	v_1, v_6		

v_1 Dequeued

v	Known	d_v	p_v
v_1	T	1	v_3
v_2	F	2	v_1
v_3	T	0	0
v_4	F	2	v_1
v_5	F	∞	0
v_6	F	1	v_3
v_7	F	∞	0
Q	v_6, v_2, v_4		

v_6 Dequeued

v	Known	d_v	p_v
v_1	T	1	v_3
v_2	F	2	v_1
v_3	T	0	0
v_4	F	2	v_1
v_5	F	∞	0
v_6	T	1	v_3
v_7	F	∞	0
Q	v_2, v_4		

v_2 Dequeued

v	Known	d_v	p_v
v_1	T	1	v_3
v_2	T	2	v_1
v_3	T	0	0
v_4	F	2	v_1
v_5	F	3	v_2
v_6	T	1	v_3
v_7	F	∞	0
Q	v_4, v_5		

v_4 Dequeued

v	Known	d_v	p_v
v_1	T	1	v_3
v_2	T	2	v_1
v_3	T	0	0
v_4	T	2	v_1
v_5	F	3	v_2
v_6	T	1	v_3
v_7	F	3	v_4
Q	v_5, v_7		

v_5 Dequeued

v	Known	d_v	p_v
v_1	T	1	v_3
v_2	T	2	v_1
v_3	T	0	0
v_4	T	2	v_1
v_5	T	3	v_2
v_6	T	1	v_3
v_7	F	3	v_4
Q	v_7		

v_7 Dequeued

v	Known	d_v	p_v
v_1	T	1	v_3
v_2	T	2	v_1
v_3	T	0	0
v_4	T	2	v_1
v_5	T	3	v_2
v_6	T	1	v_3
v_7	T	3	v_4
Q	empty		

5. Weighted Graph Single Source : Dijkstra ☆ only for non-negative path

(1) Dijkstra's algorithm for weighted graphs

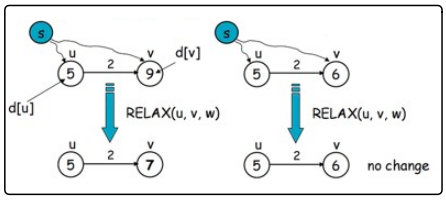
- A greedy algorithm, solving a problem by stages by doing what appears to be the best thing at each stage
- Select a vertex u, which has the smallest d_u among all the unknown vertices, and declare that the shortest path from s to u is known → 后续在declare vertex 时 update 前面 declare 过的
- For each adjacent vertex, v, update $d_v = d_u + c_{u,v}$ if this new value for d_v is an improvement

☆ Relaxation: check if this stage can improve shortest path.

If d[v] > d[u] + w(u, v)
we can improve the shortest path to v
⇒ d[v] = d[u] + w(u, v)
⇒ p[v] ← u

After relaxation:
d[v] = d[u] + w(u, v)

⇒

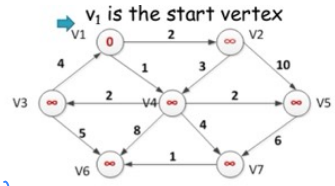


(2) Implementation:

① Initial configuration

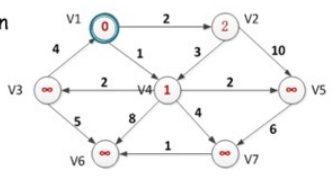
v	Known	d_v	p_v
v_1	0	0	0
v_2	0	∞	0
v_3	0	∞	0
v_4	0	∞	0
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0

v_1 is the start vertex



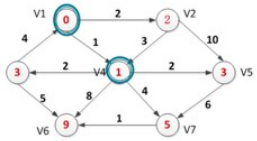
② After v_1 is declared known → {v_2, v_4}

v	Known	d_v	p_v
v_1	1	0	0
v_2	0	2	v_1
v_3	0	∞	0
v_4	0	1	v_1
v_5	0	∞	0
v_6	0	∞	0
v_7	0	∞	0



③ After v_4 is declared known → {v_2, v_5, v_6, v_7}

v	Known	d_v	p_v	
v_1	1	0	0	
v_2	0	2	v_1	
v_3	0	3	v_4	=1+2
v_4	1	1	v_1	
v_5	0	3	v_4	=1+2
v_6	0	9	v_4	=1+8
v_7	0	5	v_4	=1+4

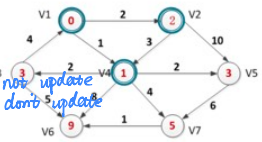


⇒ Then all nodes are marked.

④ After v_2 is declared known

v	Known	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	0	3	v_4
v_4	1	1	v_1
v_5	0	3	v_4
v_6	0	9	v_4
v_7	0	5	v_4

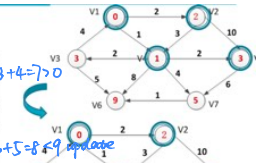
1+3=4>1 not update
1+10=11>3 don't update



⑤ After v_5 and then v_3 are declared known

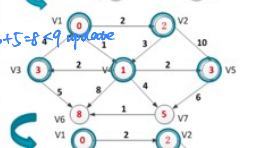
v	Known	d_v	p_v	
v_1	1	0	0	
v_2	1	2	v_1	
v_3	1	3	v_4	
v_4	1	1	v_1	
v_5	1	3	v_4	
v_6	0	8	v_3	3+5=8<9 update
v_7	0	5	v_4	

3+4=7>0



⑥ After v_7 is declared known

v	Known	d_v	p_v	
v_1	1	0	0	
v_2	1	2	v_1	
v_3	1	3	v_4	
v_4	1	1	v_1	
v_5	1	3	v_4	
v_6	0	6	v_7	5+1=6<8, update
v_7	1	5	v_4	



⑦ After v_6 is declared known

v	Known	d_v	p_v
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	1	6	v_7
v_7	1	5	v_4

☆ Declared known 为 Node 已经 to shortest path ☆

反证法：假设存在一个已访问顶点v，后续能找到更短路径

- 设新路径是通过某个未访问顶点u到达v
- 因为u尚未访问，dist[u] ≥ dist[v]（因为v先被访问）
- 新路径长度 = dist[u] + w(u,v) ≥ dist[v] + w(u,v) ≥ dist[v]
- 所以不可能更短

(3) Code:

① InitializeSingleSource(V, s):
  For v: V do:
    d[v] = ∞
    p[v] = null
  d[s] = 0

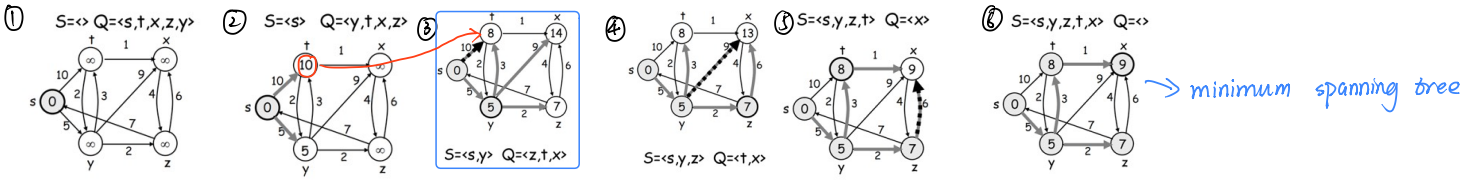
② Dijkstra(G, w, s)
  InitializeSingleSource(V, s)
  S ← ϕ (let set of shortest path 当前 u to s 的 dist
  Q ← V[G] a min heap, inner node: (d_u, vertex) ⇒ eg:
  while Q ≠ ϕ:
    u ← extractMin(Q) ① along with iteration, the node info in heap will update
    S ← S ∪ {u} ② heap will reorder.
    for v: adj[u]
      relax(u, w, v) compare d[v] and

```java
public static void dijkstra(List<List<Node>> graph, int source) {
    int vertices = graph.size();
    int[] distances = new int[vertices];
    boolean[] visited = new boolean[vertices];
    PriorityQueue<Node> minHeap = new PriorityQueue<>();
    Arrays.fill(distances, Integer.MAX_VALUE);

    distances[source] = 0;
    minHeap.add(new Node(source, distance:0));

    while (!minHeap.isEmpty()) {
        Node currentNode = minHeap.poll();
        int currentVertex = currentNode.vertex;
        if (visited[currentVertex]) continue;
        visited[currentVertex] = true;
        for (Node neighbor : graph.get(currentVertex)) {
            int neighborVertex = neighbor.vertex;
            int edgeWeight = neighbor.distance;

            if (!visited[neighborVertex]) {
                int newDistance = distances[currentVertex] + edgeWeight;
                if (newDistance < distances[neighborVertex]) {
                    distances[neighborVertex] = newDistance;
                    minHeap.add(new Node(neighborVertex, newDistance));
                }
            }
        }
    }
}
```

Declared as known

被 relax 的一定是 neighbor

Visualize steps:



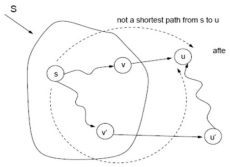
⇒ minimum spanning tree

(4) Correctness of Dijkstra:

- Theorem: For each vertex u ∈ V, we must have d[u] = δ(s, u) at the time when u is added to S ⇒ d[u] has the shortest path from s to u.

Proof:

- Assume that u is the first vertex for which d[u] ≠ δ(s, u) when added to S
  - For each vertex v in S, d[v] = δ(s, v)
  - Vertex u has the highest priority in Q: <u, ...> (i.e., d[u] <...)



let P' the path with smaller distance
↳ s → v' → u' → u
⇒ d[u'] < δ(s,u) < d[u]
u' is the first at Q ←
Contradiction.

path 与 vertex in Q connect together

eg: Exercise 1

- Given a directed graph G=(V,E) where each edge (u, v) has an associated value r(u,v), which is a real number in the range 0 ≤ r(u,v) ≤ 1 that represents the reliability of a communication channel from vertex u to vertex v
  - We interpret r(u,v) as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent
  - Give an efficient algorithm to find the most reliable path between two given vertices

- r(u,v) = Pr(channel from u to v will not fail)
- Assuming that the probabilities are independent, the reliability of a path p = <$v_1, v_2, ..., v_k$> is:
  $r(v_1,v_2)\, r(v_2,v_3) \dots r(v_{k-1},v_k)$

⇒ Find $\max_P \prod_{(u,v)\in p} r(u,v)$

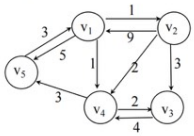
Sol1: Using dijkstra, but:
① Find the vertex with largest r
  using extractMax and Q is max heap
② update using d[u]·w(u,v)
  ⇒ Change relaxation to:
    if d[v] < d[u]·w(u,v)
      d[v] = d[u]·w(u,v)
      p[v] = u

Sol2: Taking log:

$$\max_P \sum_{(u,v)\in p} \log(r(u,v)) = \min_P \sum_{(u,v)\in p} -\log(r(u,v))$$

6. All pairs shortest path : Floyd

(1)
- A representation: a weight matrix where
  W(i, j) = 0 if i = j
  W(i, j) = ∞ if there is no edge between i and j
  W(i, j) = "weight of edge"



	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

=>
- Problem: find the shortest distance/path between every pair of vertices of a graph
- A naïve method is to run a single-source shortest path algorithm for each vertex
  - Run Dijkstra's algorithm |V| times
  - Dijkstra's algorithm's time complexity: O(|E| x log|V|)
  - Total time cost: O(|V| x |E| x log|V|)
- Floyd's algorithm
  - Total time cost: $O(|V|^3)$
  - For dense graphs, Floyd's algorithm is faster
  - It is easier to implement

(2) Idea:

1. **假设没有中间顶点**（即 **k = 0**）：
   - 一开始，我们只考虑直接的边，也就是 **D(0)**，它就是图中的权重矩阵 **W**，其中每一对顶点之间的距离是它们之间的直接路径权重（如果没有直接边，距离可能是无穷大）。
2. **逐渐增加允许的中间顶点**：
   - How to determine the alternative points
   - 接下来，我们逐步考虑引入中间顶点。对于每个 **k**（从 1 到 n），我们检查是否通过引入顶点 **vk** 可以找到更短的路径。也就是说，对于任意一对顶点 **vi** 和 **vj**，我们检查是否可以通过 **vk** 作为中间顶点，使得路径 **vi → vk → vj** 比直接的路径 **vi → vj** 更短。如果可以，则更新距离矩阵。
3. **最终包含所有顶点作为中间顶点**（即 **k = n**）：
   - 当所有顶点都被考虑作为中间顶点后，我们得到的矩阵 **D(n)** 就是最终的最短路径矩阵，其中包含了每一对顶点之间的最短路径。

recursively consider if can add an intermediate to make path shorter.

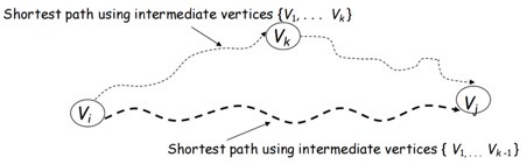
(3) Compute $D^{(k)}$ from $D^{(k-1)}$:

① Recursively compare two case, then update

=> Case 1: A shortest path from v_i to v_j restricted to using only vertices from $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices does not use v_k Then $D^{(k)}[i,j] = D^{(k-1)}[i,j]$

Case 2: A shortest path from v_i to v_j restricted to using only vertices from $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices does use v_k Then $D^{(k)}[i,j] = D^{(k-1)}[i,k] + D^{(k-1)}[k,j]$

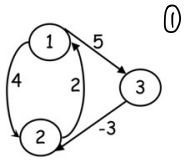
=> $D^{(k)}[i, j] = \min\{ D^{(k-1)}[i, j], D^{(k-1)}[i, k] + D^{(k-1)}[k, j] \}$



② preparation:

1. **Matrix** D keeps track of the shortest distances between vertices.
2. **Array** P tracks the intermediate vertices used to form the shortest paths.

eg:



① $W = D^0 =$

	1	2	3
1	0	4	5
2	2	0	∞
3	∞	-3	0

$P =$

	1	2	3
1	0	0	0
2	0	0	0
3	0	0	0

② $D^1 =$

	1	2	3
1	0	4	5
2	2	0	7
3	∞	-3	0

$D^1[2,3] = \min(D^0[2,3], D^0[2,1]+D^0[1,3])$
$= \min(\infty, 7)$
$= 7$

$P =$

	1	2	3
1	0	0	0
2	0	0	1
3	0	0	0

$D^1[3,2] = \min(D^0[3,2], D^0[3,1]+D^0[1,2])$
$= \min(-3, \infty)$
$= -3$

③

D^2 =

	1	2	3
1	0	4	5
2	2	0	7
3	-1	-3	0

$D^2[1,3] = \min(D^1[1,3], D^1[1,2]+D^1[2,3]) = \min(5, 4+7) = 5$

P =

	1	2	3
1	0	0	0
2	0	0	1
3	2	0	0

$D^2[3,1] = \min(D^1[3,1], D^1[3,2]+D^1[2,1]) = \min(\infty, -3+2) = -1$

④

D^3 =

	1	2	3
1	0	2	5
2	2	0	7
3	-1	-3	0

$D^3[1,2] = \min(D^2[1,2], D^2[1,3]+D^2[3,2]) = \min(4, 5+(-3)) = 2$

P =

	1	2	3
1	0	3	0
2	0	0	1
3	2	0	0

$D^3[2,1] = \min(D^2[2,1], D^2[2,3]+D^2[3,1]) = \min(2, 7+(-1)) = 2$

(4) ① Using 2 D matrix: ⇒ 对 $D^{(0)} = W$ 持续 update.

```
D ← W
P ← 0
for k ← 1 to n :
    do for i ← 1 to n :
        do for j ← 1 to n :
            if D[i,j] > D[i,k] + D[k,j]
                then D'[i,j] ← D[i,k] + D[k,j]
                     P[i,j] = k
                else : D'[i,j] ← D[i,j]
```

② Using single D matrix:

a. Principle: i. D[i, j] depends only on elements in the k-th column and row of the distance matrix

ii. When D^k update, k-th row and k-th colum remain unchanged:

- **The k-th row** contains the shortest distances from node `k` to all other nodes. When we're updating the distance from node `i` to node `j` via node `k`, the row entries (`D[k, j]`) are only used to check if the new path through `k` is shorter.
- **The k-th column** contains the shortest distances from all nodes to node `k`. Similarly, when updating the distance from `i` to `j`, we only use the column values (`D[i, k]`), but we don't modify the row or column values during the update.

⇒ At k: 从 i 到 k 与 k 到 j 的 dist 已经 shortest and known

1. **Before considering `k` as an intermediate node**, the shortest path between `i` and `k` is already known.
2. **After considering `k` as an intermediate node**, the shortest path from `i` to `k` will not change —

$$D^{(k)}[i,k] = \min\{D^{(k-1)}[i,k],\ D^{[k-1]}[i,k] + D^{(k-1)}[k,k]\} = D^{(k-1)}[i,k]$$

⇒ Once the shortest path to a node is computed, that path cannot be improved by using the node itself as intermediate vertex in subsequent steps.

eg:
- The 1-st row and 1-st column of D^1 equal to the 1-st row and 1-st column of D^0, respectively
- The 2-nd row and 2-nd column of D^2 equal to the 2-nd row and 2-nd column of D^1, respectively
- ……

D^0

	1	2	3
1	0	4	5
2	2	0	∞
3	∞	-3	0

D^1

	1	2	3
1	0	4	5
2	2	0	7
3	∞	-3	0

D^2

	1	2	3
1	0	4	5
2	2	0	7
3	-1	-3	0

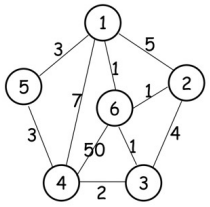
D^3

	1	2	3
1	0	2	5
2	2	0	7
3	-1	-3	0

b. code:

```
1. D ← W   // initialize D array to W[ ]
2. P ← 0    // initialize P array to [0]
3. for k ← 1 to n
4.     do for i ← 1 to n
5.         do for j ← 1 to n
6.             if (D[ i, j ] > D[ i, k ] + D[ k, j ] )
7.                 then  D[ i, j ] ← D[ i, k ] + D[ k, j ]
8.                       P[ i, j ] ← k;
```

eg:



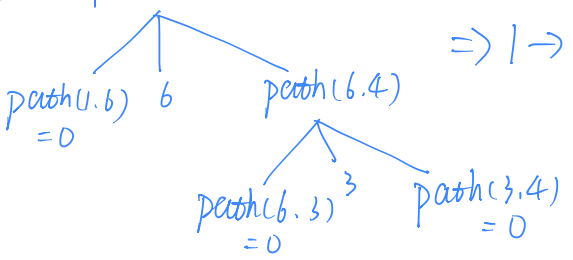
D^6 =	1	2	3	4	5	6
1	0	2(6)	2(6)	4(6)	3	1
2	2(6)	0	2(6)	4(6)	5(6)	1
3	2(6)	2(6)	0	2	5(4)	1
4	4(6)	4(6)	2	0	3	3(3)
5	3	5(6)	5(4)	3	0	4(1)
6	1	1	1	3(3)	4(1)	0

The values in parenthesis are the non-zero P values

How to find the shortest path from vertex 1 to vertex 4?

① Path: null
D[1,4] = 4(6) ⇒ Path = 1→6
D[6,4] = 3(3) ⇒ Path = 1→6→3
D[3,4] = 3(0) ⇒ Path = 1→6→3→4.

② path(1,4)



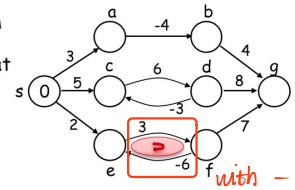
⇒ 1→6→3→4

⇒ i. print intermediate node:

```
Path(q, r):
    if (P[q,r] != 0):
        path(q, P[q,r])
        print(P[q,r])
        path(P[q,r], r)
```

7. Path with negative weight: Bellman-Ford ⇒ 可以detect 到 negative weight 与 cycle 并且防止负循环

(1) ▸ Negative-weight edges may form negative-weight cycles
  ◦ Negative weights are edge values that represent a cost or a loss, such as a penalty, a fee, or a discount

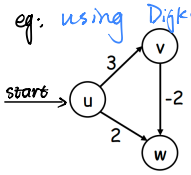


with $-6+3=-3<0$ ⇒ Negative cycle: within a cycle: $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$

☆ Shortest path 不能 simply compare which is smaller. ⇒ Dijkstra is not allowed.

  ▸ If such cycles are reachable from the source vertex s, then δ(s, v) is not properly defined!
    ◦ Keep going around the cycle, and get w(s, v) = - ∞ for all v on the cycle

eg: using Dijkstra:



	Dec	Pv	dv
u	1	0	0
v	0	u	3
w	0	u	2

⇒

	Dec	Pv	dv
u	1	0	0
v	0	u	3
w	1	u	2

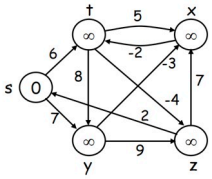
⇒

	Dec	Pv	dv
u	1	0	0
v	1	u	3
w	1	u	2

因为 visited[w] = true
⇒ not update
☆但是 update 才对. 3-2=1 < 2

(2) ▸ Idea:
  ◦ Each edge is relaxed |V| - 1 times by making |V| - 1 passes over the whole edge set
  ◦ Any path will contain at most |V| - 1 edges

☆ Main repeat loop:
relax edge: For edge (u,v)
⇒ We continue to find all paths



For each edge (u, v), do relaxation:
If d[v] > d[u] + w(u, v)
⇒ d[v] = d[u] + w(u, v)

☆After |V|-1 passes, if we can find one more relaxation ∃ ⇒ ∃ negative weight cycle.

(3) Code:

```
1. INITIALIZE-SINGLE-SOURCE(V, s)        ← Θ(|V|)
2. for i ← 1 to |V| - 1                  ← O(|V|)
3.     do for each edge (u, v) ∈ E       ← O(|E|)        } O
4.           do RELAX(u, v, w)
5. for each edge (u, v) ∈ E              ← O(|E|)
6.     do if d[v] > d[u] + w(u, v)
7.           then return FALSE
8.  return TRUE
```

Relax:
If d[v] > d[u] + w(u, v)
we can improve the shortest path to v
⇒ d[v] = d[u] + w(u, v)
⇒ p[v] ← u

After relaxation:
d[v] = d[u] + w(u, v)

※ ① According to different order
⇒ 可能每个 pass 的 output 不同但 final 结果一样.

i.

```
edges = [
    ('t', 'x', 5),
    ('t', 'y', 8),
    ('t', 'z', -4),
    ('x', 't', -2),
    ('y', 'x', -3),
    ('y', 'z', 9),
    ('z', 'x', 7),
    ('z', 's', 2),
    ('s', 't', 6),
    ('s', 'y', 7)
]
```

```
Pass 1
Updated: s → t, new distance: 6
Updated: s → y, new distance: 7
{'t': 6, 'x': inf, 'y': 7, 'z': inf, 's': 0}
Pass 2
Updated: t → x, new distance: 11
Updated: t → z, new distance: 2
Updated: y → x, new distance: 4
{'t': 6, 'x': 4, 'y': 7, 'z': 2, 's': 0}
Pass 3
Updated: x → t, new distance: 2
{'t': 2, 'x': 4, 'y': 7, 'z': 2, 's': 0}
Pass 4
Updated: t → z, new distance: -2
{'t': 2, 'x': 4, 'y': 7, 'z': -2, 's': 0}

Final distances: {'t': 2, 'x': 4, 'y': 7, 'z': -2, 's': 0}
```

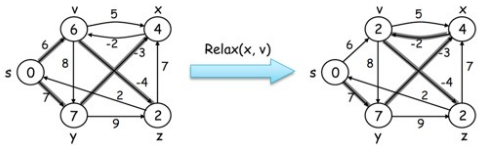
ii.

```
edges = [
    ('s', 't', 6),
    ('s', 'y', 7),
    ('t', 'x', 5),
    ('t', 'y', 8),
    ('t', 'z', -4),
    ('x', 't', -2),
    ('y', 'x', -3),
    ('y', 'z', 9),
    ('z', 'x', 7),
    ('z', 's', 2)
]
```

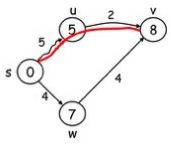
```
Pass 1
Updated: s → t, new distance: 6
Updated: s → y, new distance: 7
Updated: t → x, new distance: 11
Updated: t → z, new distance: 2
Updated: y → x, new distance: 4
{'s': 0, 't': 6, 'y': 7, 'x': 4, 'z': 2}
Pass 2
Updated: x → t, new distance: 2
{'s': 0, 't': 2, 'y': 7, 'x': 4, 'z': 2}
Pass 3
Updated: t → z, new distance: -2
{'s': 0, 't': 2, 'y': 7, 'x': 4, 'z': -2}
Pass 4
{'s': 0, 't': 2, 'y': 7, 'x': 4, 'z': -2}

Final distances: {'s': 0, 't': 2, 'y': 7, 'x': 4, 'z': -2}
```

(3) Shortest path properties:

① **Upper-bound property**
- We always have $d[v] \geq \delta(s, v)$ for all v
- The estimate never goes up - relaxation only lowers the estimate



② **Convergence property**
If $s \rightsquigarrow u \rightarrow v$ is a shortest path, and if $d[u] = \delta(s, u)$ at any time prior to relaxing edge (u, v), then $d[v] = \delta(s, v)$ at all times after relaxing (u, v)

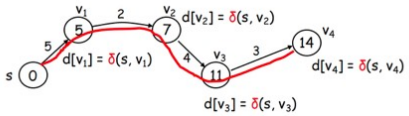


- If $d[v] > \delta(s, v) \Rightarrow$ after relaxation:
  d[v] = d[u] + w(u, v)
  d[v] = 5 + 2 = 7
- Otherwise, the value remains unchanged, because it must have been the shortest path value

⇒ Once a vertex u's shortest path is determined eg: $d[u] = \delta[s,u]$.
the edge starting from u (eg: (u,v)) will determine shortest path of v: d[v]
⇒ $d[v] = d[u] + w[u,v]$

③ **Path relaxation property**
Let $p = \langle v_0, v_1, \ldots, v_k \rangle$ be a shortest path from $s = v_0$ to v_k

If we relax, in order, $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, even intermixed with other relaxations, then $d[v_k] = \delta(s, v_k)$



不管 order relax, 但最后的 result 一样的
只是得到 result 的 step 不同.

(4) Correctness of Bellman-Ford

① **Theorem:** Show that $d[v] = \delta(s, v)$, for every v, after |V| - 1 passes

Case 1: G does not contain negative cycles which are reachable from s
⇒ 这个 Case 证明 BF 可以得到 shortest path
- Assume that the shortest path from s to v is $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $s = v_0$ and $v = v_k$, $k \leq |V|-1$
- Use mathematical induction on the number of passes i to show that:
  $d[v_i] = \delta(s, v_i)$, i=0,1,...,k

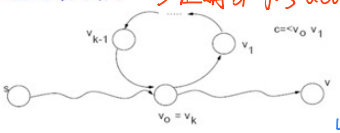
When $i=0$, $v_0 = s \Rightarrow d[v_0] = \delta(s, v_0) = \delta(s,s) = 0$
let $i = i-1$, $d[v_{i-1}] = \delta(s, v_{i-1})$
Using triangle inequality:

$d[v_i] \leq d[v_{i-1}] + w = \delta(s, v_{i-1}) + w = \delta(s, v_i)$ (using convergence)

using upper bound: $d[v_i] \geq \delta(s, v_i)$

$\Rightarrow d[v_i] = \delta[s, v_i]$

② ▸ Case 2: G contains a negative cycle which is reachable from s ⇒ 证明 BF 可以 detect 到 negative cycle



$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$$

using contradiction

we need prove

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$$

Because when meet negative cycle, there's no sol.

⇒ Assume at Case 2. we get a sol.

⇒ $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ $\quad v_k = v_0 \Rightarrow d[v_k] = 0$ ☆

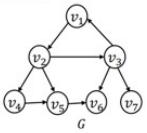
$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

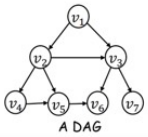
$$\Rightarrow \sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$$

Directed Acyclic Graph:

1. DAG:

(1)
- Cycle: A simple path that starts and ends at the same node
  - In directed graph G
    - Path $P = (v_1, v_2, v_3, v_1)$ is a cycle
- Directed acyclic graph (DAG)
  - A directed graph that contains no cycles



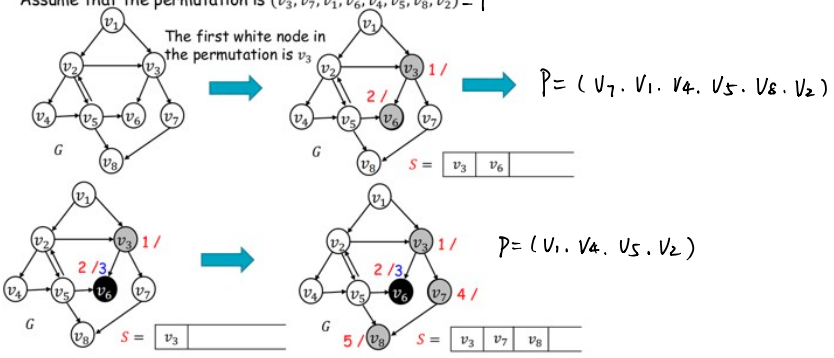


(2) Checking DAG:

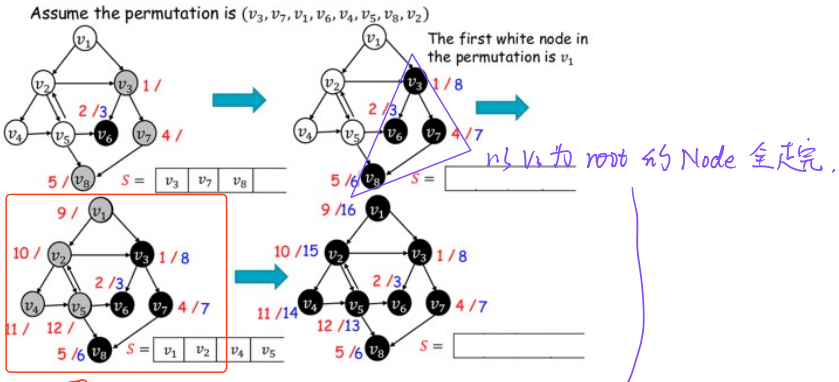
- Step 1: Do DFS traversal on graph G
  - Time complexity: $O(n+m)$ (permutation can be done in $O(n)$)
    - Find the **first white (unvisited)** node s in the permutation.
    - Do a **DFS from** s.
    - During DFS:
      - Mark nodes as **gray** (visiting),
      - Then **black** (fully processed),
      - Optionally: **record timestamps** (entry and exit time) if you're analyzing structure.
- Step 2: Classify edges according to the interval of each node derived with DFS
  - Time complexity: $O(m)$
- Step 3: If there exists a backward edge, G contains a cycle, otherwise, G is a DAG
- Total time complexity: $O(n+m)$

① Implementing DFS:

Assume that the permutation is $(v_3, v_7, v_1, v_6, v_4, v_5, v_8, v_2) = P$



Assume the permutation is $(v_3, v_7, v_1, v_6, v_4, v_5, v_8, v_2)$



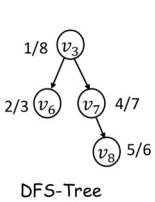
$P = (v_1, v_4, v_5, v_2)$

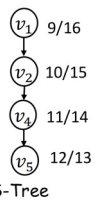
v_1 is 1st node in update P => start at v_1.

v_1 為 root 的 Node 全走完.

start new.

=> We get two path:





- v_3 is an ancestor of v_8 in the DFS tree rooted at v_3
- v_5 is a descendant of v_1 in the DFS tree rooted at v_1
- Neither v_1 or v_3 is the descendant of the other

② Edge classification:

- Assume we have done DFS on graph G. Let $\langle u, v\rangle$ be an edge in G. It can be classified into three types:

a.
- Forward edge: if u is an ancestor of v in one of the DFS-trees
- Backward edge: if u is a descendant of v in one of the DFS-trees
- Cross edge: if none of the above happens

path relation contrary to DFS tree.

using timestamps:
- Interval $I(u)$ of node u is $[u.d, u.f]$, where $u.d$ is the first discovery time and $u.f$ is the finish time
  - We will only have three cases for two nodes u and v
    - $I(u) \subset I(v)$, u is the descendant of v ⇒ backward
    - $I(v) \subset I(u)$, v is the descendant of u ⇒ forward
    - $I(u) \cap I(v) = \emptyset$, neither one is the descendant of the other.

b. Cycle theorem:

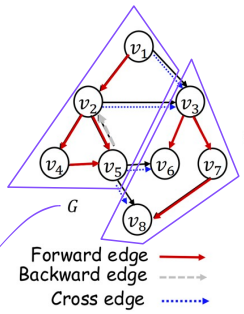
Theorem 1: Given the DFS result on graph G, then G contains a cycle if and only if there is a backward edge in the DFS result on G.

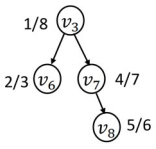
proof: i. backward → cycle:

Path: $\langle u, v\rangle$ is forward pass
and $\langle v, u\rangle$ is an another forward pass
⇒ $\langle u, u\rangle$ is a path. $u = u$
⇒ Cycle

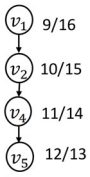
ii. cycle → backward:
⇒ to prove ancestor is also a descendant.
⇒ We have cycle: $\langle v_1, v_2 \cdots v_k, v_1\rangle$
↳ We have new cycle: $\langle v_2, v_3, \ldots, v_k, v_1, v_2\rangle$
⇒ let v_i be the 1st node pushed to stack
↳ an ancestor.
↳ A path from v_i to $v_1, v_2 \cdots v_k$ descendents: $\langle v_i, v_{i-1}\rangle$
↳ ∃ cycle ⇒ $\langle v_{i-1}, v_i\rangle$
backward.

eg:





DFS-Tree rooted at v_3



DFS-Tree rooted at v_1

$I(v_5) \subset I(v_1)$: v_5 is the descendant of v_1

$I(v_6) \cap I(v_7) = \emptyset$: neither one is the descendant of the other

How about v_3 and v_8?
How about v_3 and v_1?

Forward edge
Backward edge
Cross edge

不在一个 path 里但有 edge connects
⇒ cross edge.

2. SCC:

(1) Undirected connected graph:

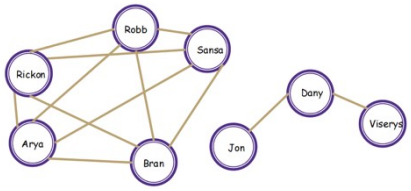
①
- Connected graph - a graph where every vertex is connected to every other vertex via some path
  - It is not required for every vertex to have an edge to every other vertex
  - There exists some way to get from each vertex to every other vertex

有1条即可.

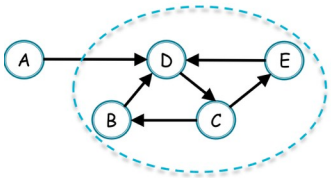
②
- Connected Component - a subgraph in which any two vertices are connected via some path, but is connected to no additional vertices in the supergraph

⇒ Componet is a connected subgraph with { Maximum connected subgraph isolated from rest of graph

eg: a graph with two CC:



(3) SCC: A subgraph C such that every pair of vertices in C is connected via some path **in both directions**, and there is no other vertex which is connected to every vertex of C in both directions.



A graph can be decomposed to multiple SCCs

① Reconstruction of graph: SCC as one node

1. **Vertices**: Each SCC in G becomes a single vertex in H. If G has k SCCs, then H will have k vertices.
2. **Edges**: You create a directed edge between two vertices in H if there is a directed edge between any vertex in one SCC and any vertex in another SCC in G. Specifically:
   - If there is an edge from vertex $u \in SCC_1$ to vertex $v \in SCC_2$ in G, then add a directed edge from SCC_1 to SCC_2 in H.

⇒ H summerize reachability between vertecces in G.
⇒ H must be DAG.

② Naive approach finding SCC:

i.
```
For each i,j in nodes:
    If i is reachable from j and vice versa
        Then i and j are in the same SCC
```

If n node ⇒ n(n-1) pairs : $O(n^2)$ performing DFS to search if {i reachable to j: O(n+m); j reachable to i: O(n+m)}

⇒ $O(n^2) \times O(n+m) = O(n^2(n+m))$

ii.
```
Array of bool reachable
For each i in nodes:
    DFS and put the visited array inside reachable of i
For each i,j in nodes:
    If reachable[i][j] and reachable[j][i]
        Then i,j are in the same SCC.
```

1. Initialize Reachable Matrix:

We start by creating a 2D boolean array `reachable` of size $n \times n$, initialized to `False`.

2. Perform DFS for Each Node:

For each node i, we perform a DFS and update the `reachable` matrix to indicate which nodes are reachable from i.

3. Identify SCCs:

After filling the `reachable` matrix, we can check for each pair of nodes i and j if both `reachable[i][j]` and `reachable[j][i]` are `True`. If they are, i and j belong to the same SCC.

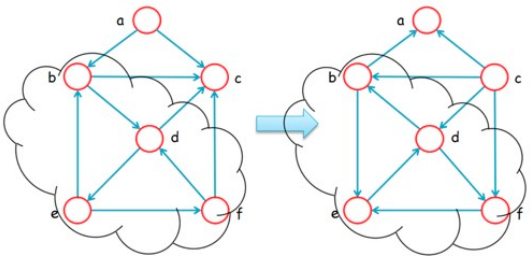
```java
public void DFS(Map<Integer, List<Integer>> graph, boolean[] visited, boolean[][] reachable, int i, int j){
    visited[i] = true;
    reachable[j][i] = true;
    for (int neighbor : graph.get(i)){
        if (!visited[neighbor]){
            visited[neighbor] = true;
            DFS(graph, visited, reachable, neighbor, j);
        }
    }
}
public boolean[][] Reachable(Map<Integer, List<Integer>> graph, int n){
    boolean[][] reachable = new boolean[n][n];
    for (int i = 0; i < n; i++){
        boolean[] visited = new boolean[n];
        DFS(graph, visited, reachable, i, i);
        }
    return reachable;
}
public List<List<Integer>> SCCs(boolean[][] reachable, int n){
    List<List<Integer>> SCCs = new ArrayList<>();
    boolean[] visited = new boolean[n];
    for (int i = 0; i < n; i ++){
        if (!visited[i]){
            List<Integer> scc = new ArrayList<>();
            for (int j = 0; j < n; j ++){
                if (reachable[i][j] && reachable[j][i]){
                    scc.add(j);
                    visited[j] = true;
                }
            }
            SCCs.add(scc);
        }
    }
    return SCCs;
}
```

j: initial vertex i: descendant vertex.

If <i,j> 与 <j,i> accessible
⇒ 不用再从 j 来 visit
因为 i 与 j reachable
与 j mutually reachable 的 i 也一定 reachable

② Kosaraju – Sharir :

- Fact: the transpose graph (the same graph with the direction of every edge reversed) has exactly the same SCCs as the original graph



i. algorithm:

Input: a directed graph G=(V, E)

Output: all the SCCs of G

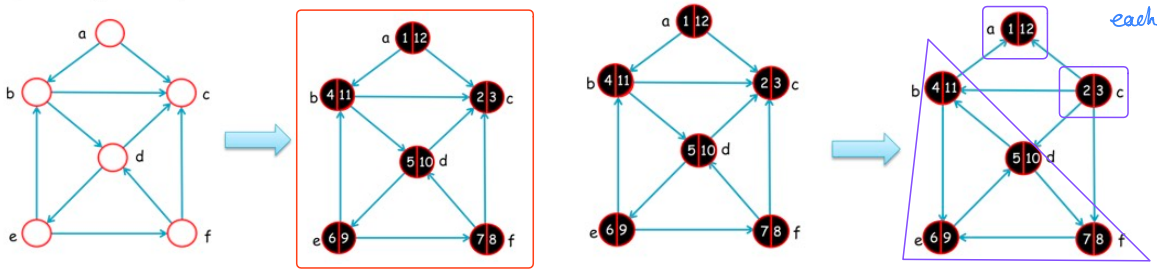
1. Run DFS on G, during which we compute the first discovery time and finish time of each vertex
2. Build the transpose graph G^T=(V, E^T)
3. Run DFS on G^T, by considering the vertices' finish time in descending order
4. Output the vertex set in each DFS traversal as an SCC

⇒ 用 adjacent matrix { 1 , neighbor / 0

G^T ⇒ Transpose matrix.

eg:

- Run DFS and compute the first discovery time and finishing time of each vertex
- Reverse the edge directions



using P^2 to implement DFS.

each DFS tree compose of SCC

{a}

{b. d. e. f}

{c}

Reorder the node in descending order of finish time

⇒ [a. b. d. e. f. c] = P^2

ii.

- What's the overall time complexity?

Input: a directed graph G=(V, E)

Output: all the SCCs of G

O(n+m) → 1. Run DFS on G, during which we compute the first discovery time and finish time of each vertex

O(m) → 2. Build the transpose graph G^T=(V, E^T)

O(n+m) → 3. Run DFS on G^T, by considering the vertices' finish time in descending order

O(n) → 4. Output the vertex set in each DFS traversal as an SCC

The overall time complexity: O(n+m)

iii.

```java
public static void dfs(int[][] graph, int v, boolean[] visited, Stack<Integer> stack) {
    visited[v] = true;
    for (int i = 0; i < graph.length; i++) {
        if (graph[v][i] == 1 && !visited[i]) {
            dfs(graph, i, visited, stack);
        }
    }
    stack.push(v);
}

public static int[][] transpose(int[][] graph) {
    int n = graph.length;
    int[][] transposed = new int[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            transposed[j][i] = graph[i][j];
        }
    }
    return transposed;
}

public static void dfsTranspose(int[][] transposedGraph, int v, boolean[] visited, List<Integer> scc) {
    visited[v] = true;
    scc.add(v);
    for (int i = 0; i < transposedGraph.length; i++) {
        if (transposedGraph[v][i] == 1 && !visited[i]) {
            dfsTranspose(transposedGraph, i, visited, scc);
        }
    }
}
```

```java
public static List<List<Integer>> kosaraju(int[][] graph) {
    int n = graph.length;
    Stack<Integer> stack = new Stack<>();
    boolean[] visited = new boolean[n];
    for (int i = 0; i < n; i++) {
        if (!visited[i]) {
            dfs(graph, i, visited, stack);
        }
    }
    int[][] transposedGraph = transpose(graph);
    Arrays.fill(visited, val:false);
    List<List<Integer>> sccs = new ArrayList<>();

    while (!stack.isEmpty()) {
        int v = stack.pop();
        if (!visited[v]) {
            List<Integer> scc = new ArrayList<>();
            dfsTranspose(transposedGraph, v, visited, scc);
            sccs.add(scc);
        }
    }

    return sccs;
}
```