Stack Queue tree

Linked List & stack & Queue

Bubble Sort: $T(n) = (n-1)^2/2 + 1$

Merge Sort: $T(n) = (n+2) \cdot \log(2 \cdot n)/2$

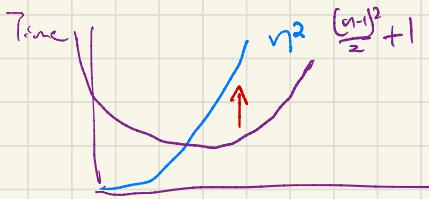
- 公式: $T(n) \leq k \cdot f(n)$ k = constant $T(n) = \frac{(n-1)^2}{2} + 1$

$T(n) \leq k \cdot (n^2)$

$\frac{(n-1)^2}{2} + 1 \leq k \cdot n^2$ ← $f(n) = n^2$

$T(n) = O(f(n)) = O(n^2)$

- $O(n)$ = upper bound



→ 1. n 取足夠大
2. 相同增長速率

- $O(n)$ 準確高等

- 快 → 慢: $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^3) < O(2^n)$

Constant

Linear

$< O(n!)$
$< O(n^n)$

Big O
↓
worst Case
upper bound

Omega
↓
best Case
lower bound

Theeta

- Linear Search:

```
for (int i=0 ; i<size; i++) {
        if (arr[i] == target)
            return true
    }
return false
```

T(N) == O(N)

对于集合类，运行效率不固定

1. Best Case: O(1)
2. Worst Case: O(N)
3. Ave Case: $O(\frac{1+N}{2}) \Rightarrow O(N)$ $\frac{1+N}{2} \leq N$

Linked List

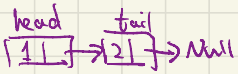
Pros: 1.动态 2.Size 3.头尾操作快

Cons: 1.无index, 2.访问某一个点，3.比Array, Vector更多时间/空间

- 两个class:

① → class Node<T> {
    1. data
    2. Address

② → class LinkedList<T> {

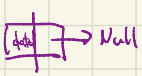


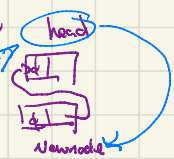
Public class LinkedList

```
Public class Node {
  Public int data;
  Public Node next;
  Public Node (int data) {
      this.data = data;
      this.next = null;
  }
};
```

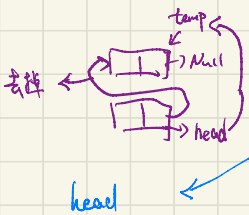
```
Node head;
Public void prepend (int data) {
    if (head == null) {
        Node NewNode = new Node(data)
        head = NewNode;
        NewNode.Next = null;
    }
    else {
        Node Newnode = NewNode(data);
        newnode.next = head;
        head = new node;
    }
```

最前/左侧 Node.






```
Public int removeHead() {
    int result = head.data;
    Node temp = head.next;
    head.next = null (断连接)
    head = temp
    return result;
}
```



断定方法:

```
if (head == null)
    throw new ....
```

Prepend
removeHead
append
removeTail

O(1)

O(n)

while (Current != num)

Linked List:

1. 分类: 不可往回走, Singly LL
   可往回走, double LL
2. 是否有尾巴 (tail): 没有尾巴: Singled - head
   有尾巴: double - head

Head / tail

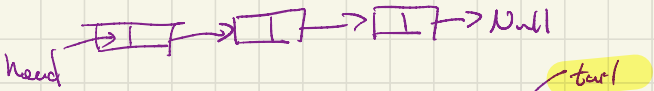
⇓

标记点

4 types Linked List

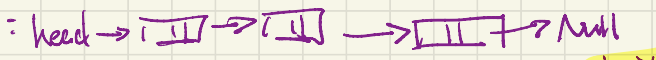
tail Like ≈ Last head.

1. Singly S-head LL:

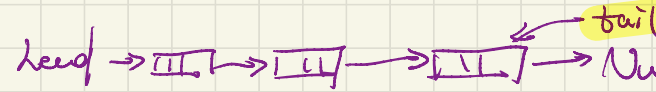


2. Singly D-head LL:

3. Doubly S-head LL:



4. Doubly D-head LL:



	①	②	③	④
Prepend	O(1)	O(1)	O(1)	O(1)
Append	O(N)	O(1)	O(N)	O(1)
rem H	O(1)	O(1)	O(1)	O(1)
rem T	O(N)	O(N)	O(N)	O(1)

{ Node : Node, Prev, Node. Next. T data

Linked List: Head, tail

Linked list VS Array

get ( )

↳ O(N)

Best: O(1)

Worst: O(N) ⇒ [O(N/2)] DDLL 최적화

ave : O(N)

Prepend (int data)

O(1)

```
get (int index)
    ↳ O(1)
return arr [index]
```

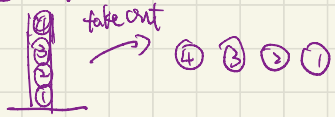
prepend (int data)

1→ [2|3|4] → O(N)

[1|2|3|4]

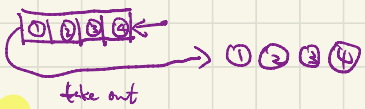
Stack & Queue.

Stack.



- push
- pop

Queue.



- enqueue
- dequeue

- ADT: abstract data type. Stack S; (create stack S)

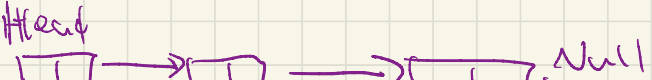
  存什麼 data → operation: add / take / search / ...) ADT basic operation.

- Stack:

  First in Last out

  → ADT: (用 array / LinkedList) 定義 push, pop, 因 Stack 是一種 ADT

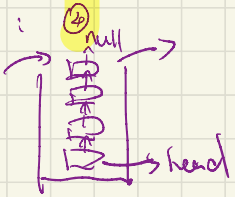
- → 用 LL 當 Stack:



1. Append 2. prepend 3. rem H 4. rem T

push(): ① ② ③

pop(): ④



which one?

① + ④ → Append + rem T → DDLL

② + ③ → prepend + rem H → SSLL

Stack:

	Array	LL
Push	O(1)	O(1)
Pop	O(1)	O(1)

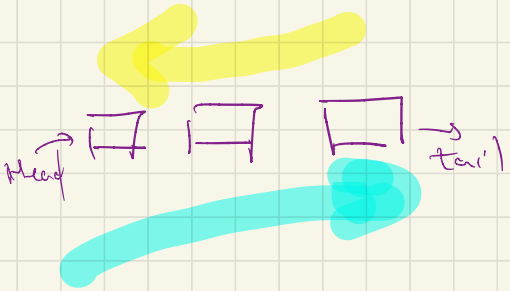
Queue:

LL:

1. Append 2. prepend 3. rem H 4. rem T

Enq

deq

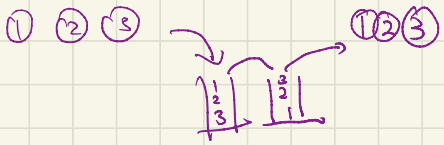


① + ③ ⇒ Append remove Head → SDLL O(1)

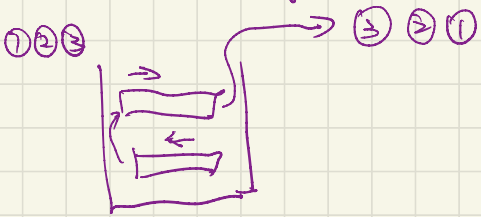
② + ④ ⇒ prepend remove Tail → DDLL O(1)

Ex.

2 stack implement 1 Queue in O(1)



2 Queue implement 1 stack in O(n)

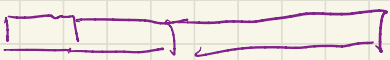


Stack by Array

Capacity x2 = O(1) ✓

Capacity ++ = O(N)

Cap ✗



Push worst cases, O(N)

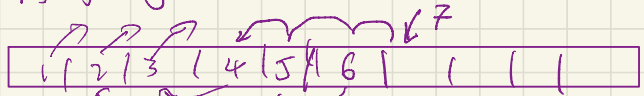
- Push ← size++
- Pop ← size--

Queue by Array

- Enqueue ( )
- Dequeue ( )

→ 用 LinkedList Enqueue, Dequeue O(1) ✓ proved above

→ 用 Array



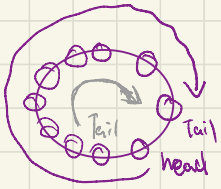
→ Bigger ? → Cap x2

enqueue ( ) dequeue ( ) Enqueue (7)

( ) ( ) O(N) → not good

O(1) (在 Cap x2 下)

→ Circular array



≈ Linked list doubly

⇒ need int head, tail

enqueue: Tail change (尾)

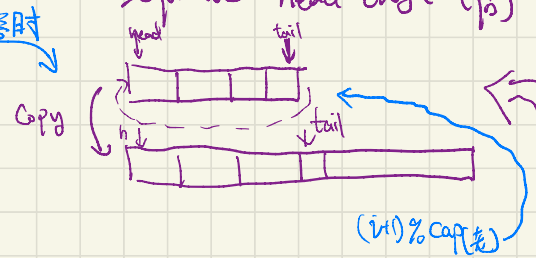
Dequeue: Head change (頭)

★ : (head+1)%Capacity

(tail+1)%Capacity

- enq (1) ⇒ (0+1)%Cap
- (2) ⇒ (1+1)%Cap
- deq (C) ⇒ (0+1)%Cap
- (6) ⇒ (5+1)%Cap

需扩容时



enqueue: arr[tail] = data

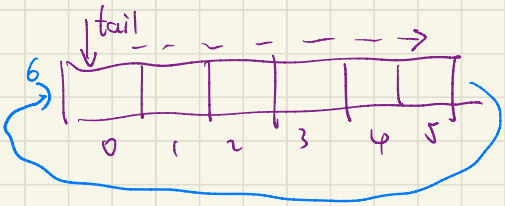
tail = (tail+1)%Cap

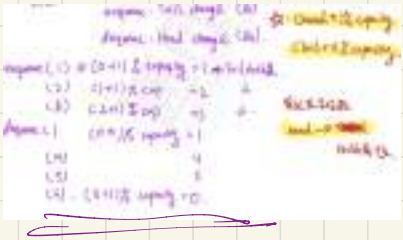
size++

dequeue: int result = arr[head]

head = (head+1)%cap

O(1)



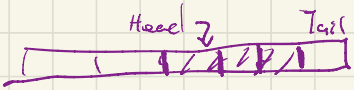


Ex.

Size+1 in array stack, while it is full

Push → O(n) Pop → O(1)

Tips: 刪除資料不在



↳ 不在有效 range.

Ex.

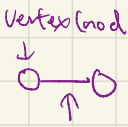
Size×2 in array queue, while its full

enqueue: O(1) deq: O(n)

向左移, when delete one

Tree (we cover array, LL, Tree)

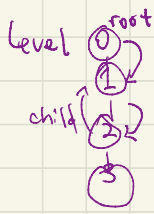
- graph before tree
- graph → 点 & 线

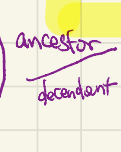


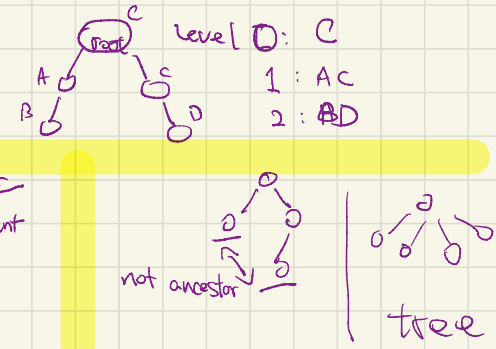
connected graph
cycle

- Tree
  - L> graph Connected without cycle

- Tree 无 朝型 → root.

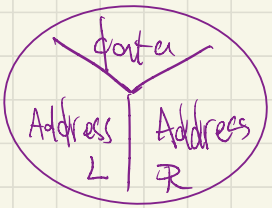






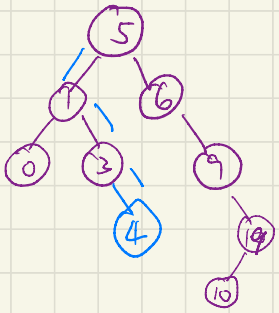
However, Its hard to set node,

SO, restraint → ~~binary~~ tree




```
class Node {
    int data
    Node left
    Node right
}
```

Binary Search Tree



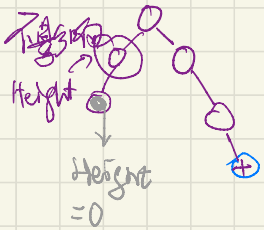
- 加入 4
  对比 3 次 → 3 次 recursion
- Search 10 → 4 次

Path：点到点

depth：当前节点 → root

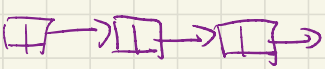
Height：当前节点 → 最下面（level 最大的节点）

Height of the tree = depth of the tree



（求 root）

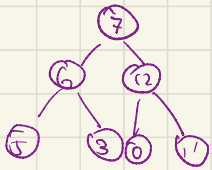
- Traversal：怎么样把 Tree 走遍

LL 

curr = head
curr = curr.next

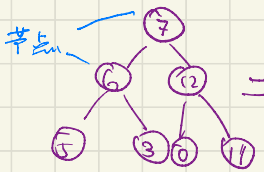
Array 

→ Tree：



⇒ BST ✗ BT is fine

How traversal?



Pre-order : 先自己，左，右 subtree ⇒ 7, 6, 5, 3, 12, 0, 11

In-order : 左，自己，右 5, 6, 3, 7, 0, 12, 11

Post-order traversal : 左，右，自己 5, 3, 6, 0, 11, 12, 7

Recurence :

Base Case

4个 Step :

1. 大问题
2. 小问题
3. 连接
4. Base case

Eg. fib Question

1. fib(n)
2. fib(n-1), fib(n-2)
3. fib(n) = fib(n-1) + fib(n-2)
4. fib(1) || fib(2) ⇒ 1

Eg. int exp(int m, int n) { m^n

```
public int exp(int m, int n) {
    if (n==0) {
        return 1;
    } else if (n==1) {
        return m
    } else {
        return m * exp(m, n-1);
    }
```

大：m^n exp(m,n)

小：m^{n-1} exp(m,n-1)

连：exp(m,n) = exp(m,n-1) × m

B：n==0 ⇒ return 1.

Eg.

```
public int MulCount(int m, int n)
    if (n==0 || m==0) {
        return 0
    } else if (n<0 || m<0) {
        return (-m)(-n)
    } else {
        return mul(m, n-1)+m;
    }
```

不用 m × n "×"

Eg. int Sum (int[] arr, int size)

```
if(size==0) {
    return 0
} else {
    return arr[size-1]+Sum(arr, size-1)
}
```

大：算大小为size的array的和
小：size-1长度的array的和
整：arr[size-1]+小
B：size==0 ⇒ 0

Eg.

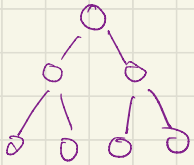
```
{ int Sum (Node head) }
if (head == null) {
    return 0
}
return head.data + Sum (head.next)
```

Eg.

```
1  boolean search(int[] arr, int size, int target){
2      if(size==0){
3          return false;
4      }else if(arr[size-1] == target){
5          return true;
6      }else{
7          return search(arr, size-1, target);
8      }
9  }
10
11 boolean LLsearch(Node head, int target){
12     if (head == null ){
13         return false;
14     }else if (head.data == target){
15         return true;
16     }else{
17         return LLsearch(head.next, target);
18     }
19 }
```

boolean Search(int[] arr, target, size)
if (size==0){
    return false
}
if (arr[size-1] == target){
    return true
}else if (arr[i-1] arr
size--
return boolean Search(int[] arr,
)

Tree → Recurrence



Eg.

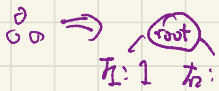
```
int height(Node root)
if (root == null) {
  return -1;
}
int LH = height(root.left);
int RH = height(root.right);
if (LH > RH) {
    return LH+1;
}else{
    return RH+1;
}
```

A: all height

+: ① 左 height ② 右 height

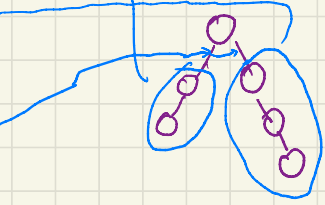
取: 左右 较大 + 1

B: root == null return −1

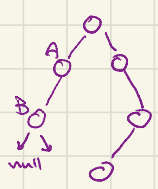


左: 1 右: 2

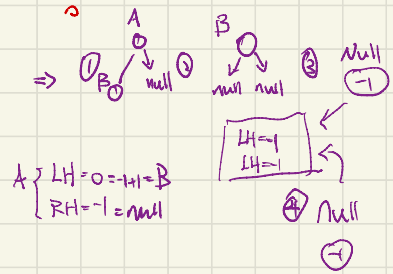
∴ 右 > 左, 右 + 1 = 3 = height



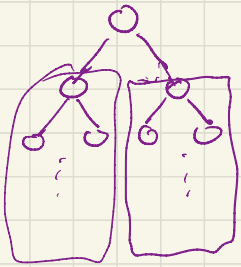
Suspicious ?



A { LH = 0 = -1 = B
  { RH = -1 = null



Eg. BT 有多少 Node?



大：所有node数量

小：①左subtree ②右subtree

合：①+②+1 → root

B：root == null ⇒ return 0

```
int node (Node<T> root)
    if (root == null) {
        return 0;
    }
    int left = node (root.getleftchild())
    int right = node (root.getrightchild())
    return left + right + 1;
}
```

Tips:

Big O: assume Tree has N nodes

Height: O(N)

Node: O(N)

LongPathPassingRoot: O(N)

Eg. 点穿过 root 到点最长 path

```
int LongPathPassingRoot (Node<T> root)
    if (root == null) {
        return 0;
    }
    return int heightleft + int heightright + 2
```

BST analysis

1. 左右最多两个节点

2. 自己比左大，比右小

3. BST + inorder print ⇒ result: sorted ascending (从小到大)

- Traversal: same BT

◦ BST. Insert

↳ 已加入的节点，位置不会变

↳ 对于加入的新node，总能加在一个位置上。

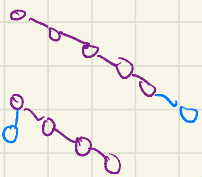
↳ Insert 运行效率

↳ worst Case : O(N)

↳ Best Case : O(1)

↳ ave Case : O(logN)

↳ 推论法 : O(height)



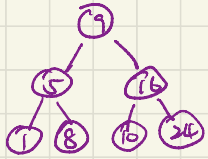
(给，四分... → logN)

◦ Code

```
BST Node<T> insert (BST Node<T> root, T data){
    If (root == null) {
     return new BTNode<T>( , )
      {
      ①往左
       If (data ≤ root.getData()) {
            insert (root.getLeftChild(), data);
       }                    Ch2 private
       else {② 往右
            insert (root.getRightChild(), data);
       }
       return root;
```

Recurrence

Remove BST



Remove (1): no child
Remove (5): one child
Remove (16): two children (find one child to replace)

1. 找到要删的点 search/traversal.
2. 连接删的 1) parent null (child ≤ 1) — single or double
   2) Pointer change
   &parent = &parent

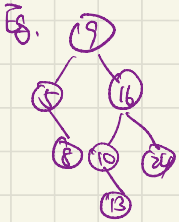
→ 让 parent 指向要删的点的 child node.
(当删的点，有子点 = 1)

↓

Predecessor and Successor

左 all tree，比我 node 大的最小的，和比 node 小的最大的 (相邻 node)
↓ inorder

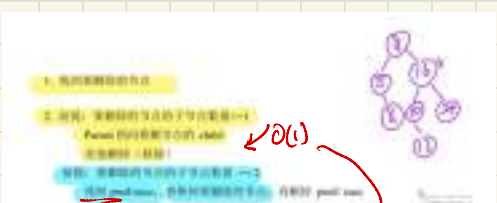
Eg.



remove 16
⇒ inorder ⇒ 5, 8, 9, 10, 13, 16, 24

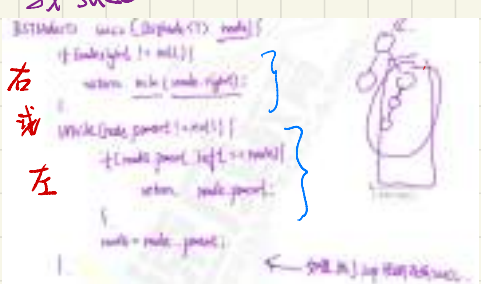
predecessor of 16 = inorder 前一个：13
successor of 16 后 ：24

Pred 和 Succ 至多有个 child (左 or 右)
13可替16
24可替24
} 2选1



找 Succ

△ 右
找
左



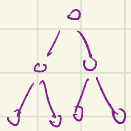
?

iterative function

- Search : O(h)

- Perfect Tree

每个节点子数都是满的，叶子节点在同 level



$\Rightarrow 1 = 2^{\text{之为level}}$

$\Rightarrow 2 = 2^1$

$\Rightarrow 4 = 2^2$

$\vdots$

2^k

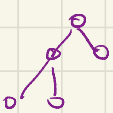
perfect tree nodes How many in total?

$\hookrightarrow \sum_{k=0} 2^k \Rightarrow 2^{k+1} - 1$

一共从上到下

- Complete Tree

1. 除了最下面的level之外，上面level全都放满
2. 最下面level所有节点靠左放



Tips: Perfect tree 也是 Complete Tree

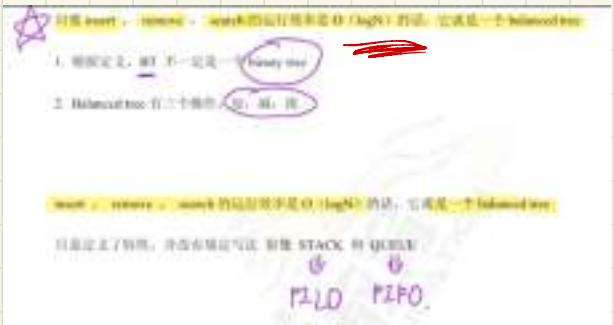
Q_1: Complete tree 一定是 BST 吗?

A_1: False, (data)

Q_2: BST 一定 Complete tree

A_1: False

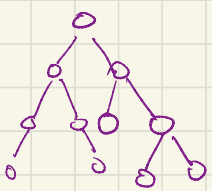
Balanced Tree



AVL Tree : Balanced Tree

1. 是一个BST

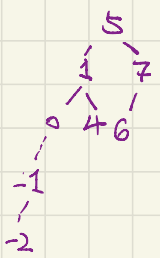
2. 任何节点，左tree Height 与右 tree Height 不超过 1 相差



# of nodes: 11
height: 3

insert: O(h)
search: O(h)
remove: O(h)
=> O(log n)
same

- Insert:
  → 用BST的 insert 放入?

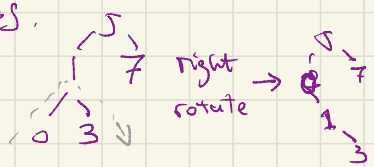


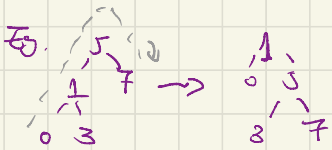
insert (6) ok
insert (-1)
insert (-2) => not ok => fix -> (height fix)

Fix → Rotate

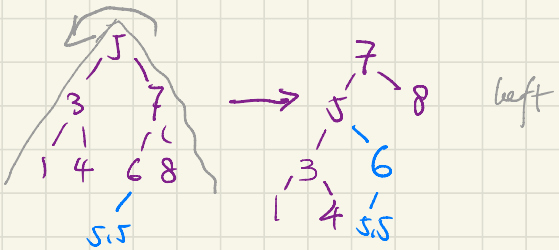
对于AVL rotation，对于某个点的rotate，可左可右

Eg.

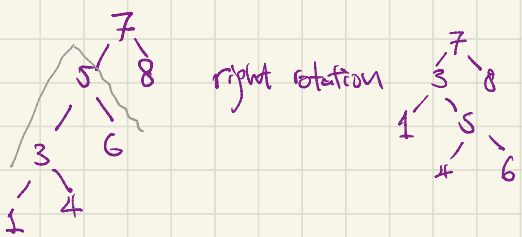




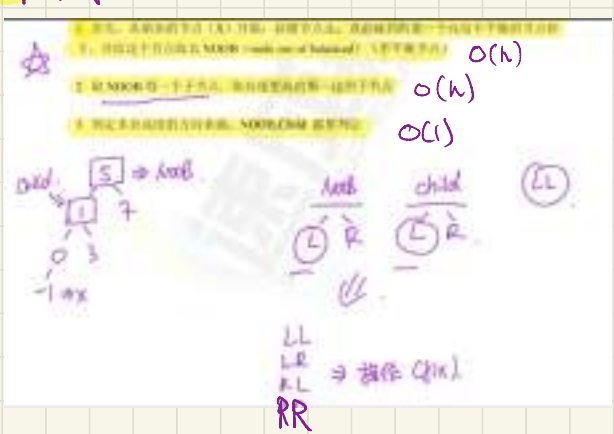
所谓变成一条链条，往右的方向一拉，then重新定义



BST旋转，rotate不改变BST特征



Fix

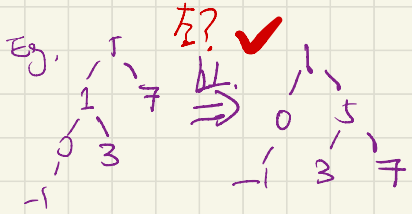


右？✔

LL：当NodeB向左rotate一次，fixed then

RR：当NodeB向右rotate一次，fixed then

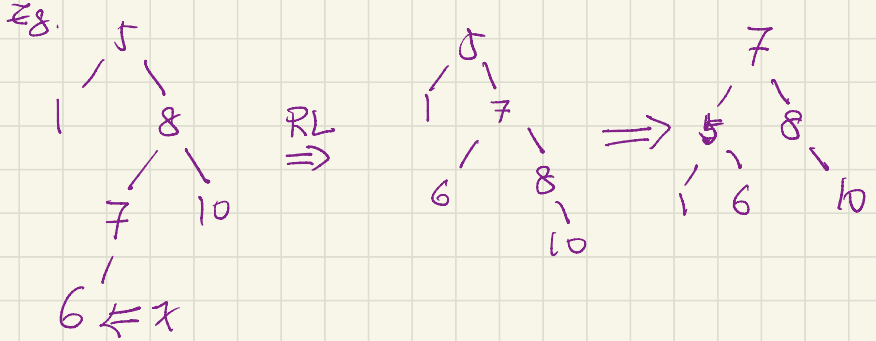
Eg.



RL：沿child向左转一次，then 沿Node向左转一次，then fixed

LR：沿child向左转一次，then 沿Node向右转一次，then fixed

Eg.



AVL insert:

① 用 BST insert

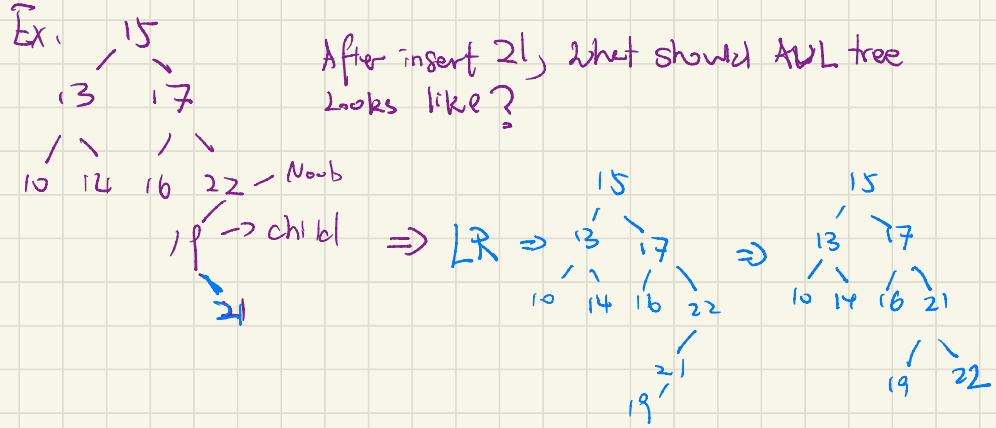
② fix Height

→ By O:

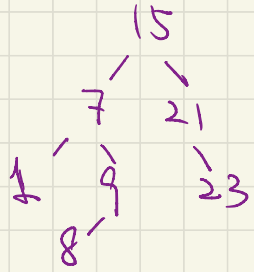
①：O(logN) / O(h)

②：

Ex.



Remove:

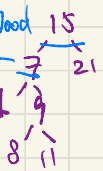


AVL:
remove(23)

1. 先套用BST删除
2. 左右 超过1?
∴fix

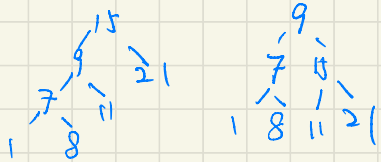
对于删掉节点开始，一直到root，进行修复 (rebalanced)

child — Node 15 — 21



remove 23
右+1=左+1

LR:

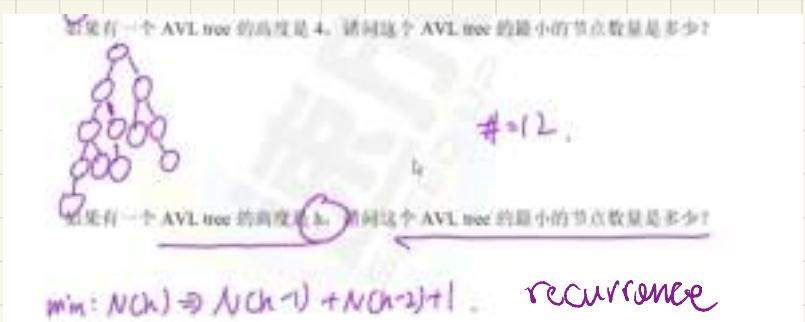


任何不 balanced node，把为高度当成多个高度，进行 insert to fix

⇒ O(h)

\+ 1. BST remove: O(h) ⇒ log(n)
2. 找 node ⇒ O(h)
rebalanced (rotation) ⇒ O(1)

} ⇒ O(h) ⇒ O(logN)



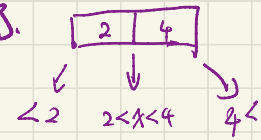
2-3 Tree

2-3 TREE

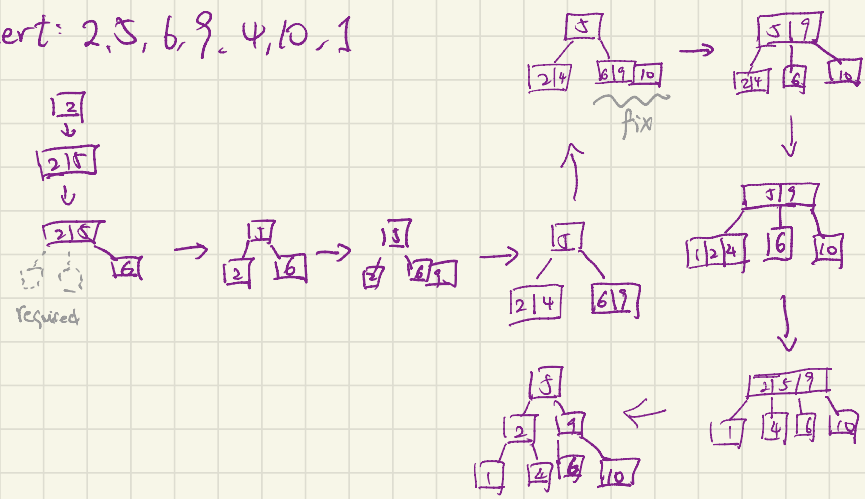
Properties of 2-3 tree:

1. Every non-leaf vertex has 2 or 3 children
2. Every vertex stores 1 or 2 values (as opposed to standard BSTs)
3. The values in a node are more than everything on the right, and less than everything on the left
4. All leaves has the same depth
5. Nodes with two children are called 2-nodes. The 2-nodes have one data value and two children
6. Nodes with three children are called 3-nodes. The 3-nodes have two data values and three children.
7. Always insertion is done at leaf.

Eg.

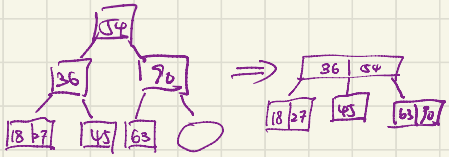


- Insert: 2, 5, 6, 9, 4, 10, 1



- Delete

Delete:
1. To delete a value, it is replaced by its in-order successor and then removed.
2. If a node is left with less than one data value then two nodes must be merged together.
3. If a node becomes empty after deleting a value, it is then merged with another node.

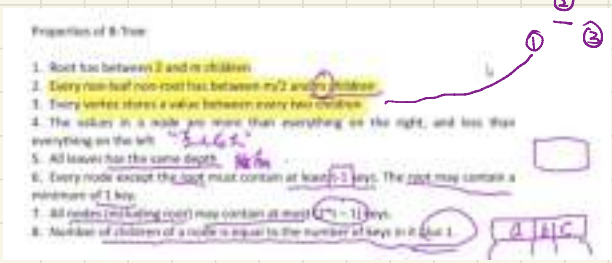


- Search:

Similar:

- B Tree

BST logic



Root → 1 key
node → $\frac{m}{2}$ & m
key → t-1 2t-1

key: node 存多少下去
存多少个

The minimum height of the B-Tree that can exist with n number of nodes and m is the maximum number of children of a node can have

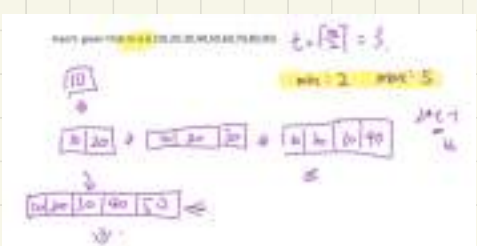
$$h_{min} = \lceil \log_m(n+1) \rceil - 1$$

The maximum height of the B-Tree that can exist with n number of nodes and t is the minimum number of children that a non-root node can have

$$h_{max} = \lfloor \log_t \frac{n+1}{2} \rfloor \quad t = \lceil \frac{m}{2} \rceil$$

Insertion

1) Initialize x as root.
2) While x is not leaf, do following
..a) Find the child of x that is going to be traversed next. Let the child be y.
..b) If y is not full, change x to point to y.
..c) If y is full, split it and change x to point to one of the two parts of y. If k is smaller than mid key in y, then set x as the first part of y. Else second part of y. When we split y, we move a key from y to its parent x.
3) The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x.



B-Tree delete

BST

B-Tree search

BST

When deleting a node from a leaf, the leaf can become empty
    This sometimes called "underflow"
In this case we need to rebalance the node
Rebalancing:
If possible, borrow from the right sibling
Otherwise, if possible, borrow from the left sibling
Else, all siblings have only one value
    Merge the parent with the right sibling, if possible
    Otherwise merge the parent with the left sibling
    ** Fix the parent if needed (e.g. using recursion)

Priority Queue - (Binary) Heap

- Priority Queue:
  带有优先级的排队

- operation:
  insert() / add()
  removeHeightPriority() 优先级

- Property: 优先级最高的先出来，优先级具体情况具体分析

  Tips: 不需要index 在PQ

- Design PQ:

```
class PriorityQueue {
  ArrayList<T> arr
  public void insert(int data); -> curr.add(data);
  public int removeMax(); -> 找到最大的 (O(N))
                             2. delete
```

Tips: insert -> O(1)

Remove -> O(N) ⇒ not good LL -> O(N)

Consider:

如果用 array 写，insert (加尾巴) + 每次找最大值

→ insert × N 次然后 remove N 次

↳ insert: $O(1) \times N \Rightarrow O(N)$
Remove: $O(N) \times N \Rightarrow O(N^2)$
} $O(N^2)$ total

$O(N^2)$ is not good, so: change insert/remove to $O(\log N)$ in some way.

→ $O(\log N) \times N + O(\log N) \times N \Rightarrow O(N \log N)$ desired

↓ How?

Complete tree (binary heap) ⇐ PQ 使用 logic of this

⇓

使用：概念 ↓ 使用写法

Binary heap

编写者：Complete binary tree → 逻辑

⇓

Array / LL

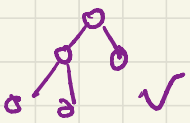
∵ PQ 没有规定用什么完成 (array, LL, array list, complete tree)

∴ Binary heap 是 PQ 的一种

⇒ Binary Heap: ① max heap => remove Max

② min heap => Remove Min

Tips: Complete Binary Tree:

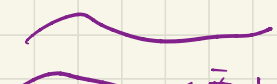


⇒ Binary Heap 逻辑上:

1. 一定是 Complete binary Tree

2. 如 remove Max ⇒ Max heap ⇒ 对于任何节点来说, "上大下小"
它比其它child更大.

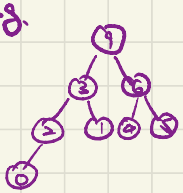
如 remove Min ⇒ Min heap ⇒



更小.

"上小下大"

Eg.



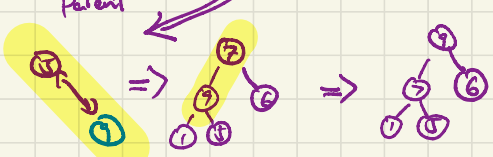
Max heap ✓.

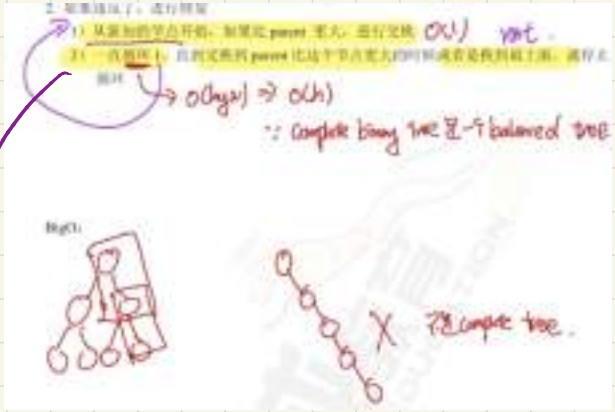
Max 顶 → root
Min 顶 → root
=> ROOT

⇒ Insert:

insert 9 ⇒ fix order needed

parent

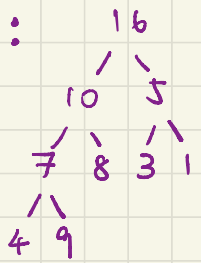




$O(h) \Rightarrow O(\log n)$

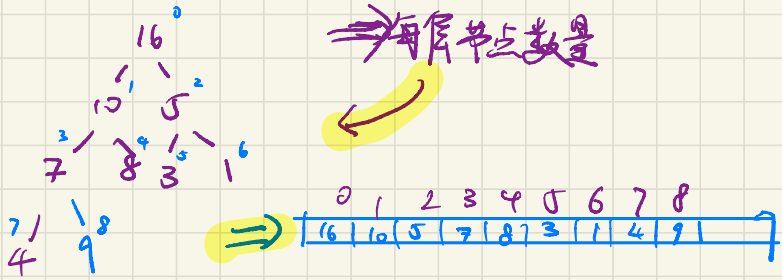
⇒ Bubble up (fix)

代码：



insert 9.

由于 binary heap 是一个 complete tree ⇒ ① 上面全部放满 ② 每层节点数量



+9 ⇒ arrayList 最后一位

tree node ⇒ index

找 index 规律

对比交换怎么做? → tree ⇒ array

Q: array 的 index ⇒ parent index ?

3 的 parent 3 ⇒ 3 = index 2

8	10	4	1
4	7	7	3
7	7	8	3
		i	parent

parent index

$$\Rightarrow i \Rightarrow \left\lfloor \frac{(i-1)}{2} \right\rfloor$$

Code:

```
public class Myarray{
    int[] arr;
public void push_back(int index){

}
```

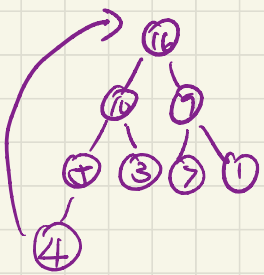
```
public class minHeap{
    Myarray arr;
    public minHeap(){
        arr = new Myarray();
    }
    public void insert(int data){
        arr.push_back(data);
        bubble_up(arr.size()-1);
    }
    public void bubble_up(int index){
        int parentIndex = (index-1)/2;
        while(index > 0){
            if(arr[index] < arr[parentIndex]){
                int temp = arr[index];
                arr[index] = arr[parentIndex];
                arr[parentIndex] = temp;
                index = parentIndex;
                parentIndex = (index -1)/2;
            }
            else{
                return;
            }
```

Swap

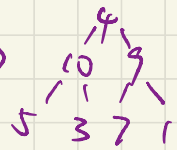
index

classin

Remove Max/min



① CBT ② heap

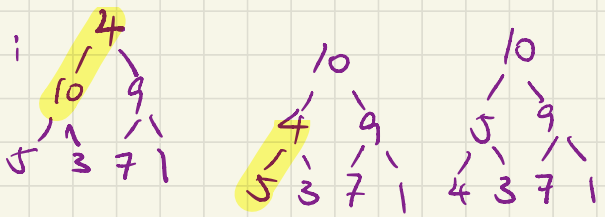


方法

1. 先把最后一直提到root上，保证第一条

2. fix

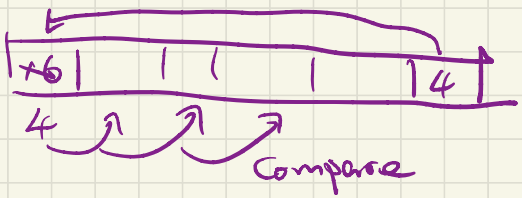
Fix：



2. 从root开始一直走到最下面，找 left child & right child (三个值比较)

1) 最大值是自己，不换

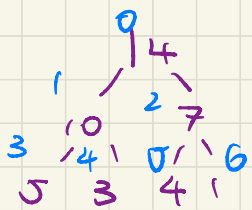
2) 不是自己，交换，循环这个过程



$O(\log N)$

$O(h)$

Bubble down:



index i => i 的 child

0 => 1, 2

1 => 3, 4

2 => 3, 4

3 => 7, 8

index i left child:

2i+1

right child:

2i+2

-> arraylist

```
public int removeMin(){
    int result = arr[0];
    arr[0] = arr[arr.size()-1];
    arr.pop();
    bubble_down(0);
    return result;
}
```

```
public void bubble_down(int i){
    int left = 2 * i +1;
    int right = 2*i+2;
    int min = i;
    if(arr[left] < arr[min] && left < arr.size()){
        min = left;
    }
    if(arr[right] < arr[min] && right < arr.size()){
        min = right;
    }
    if(min == i){
        return;
    }
    else{
        int temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
        bubble_down(min);
    }
}
```

If

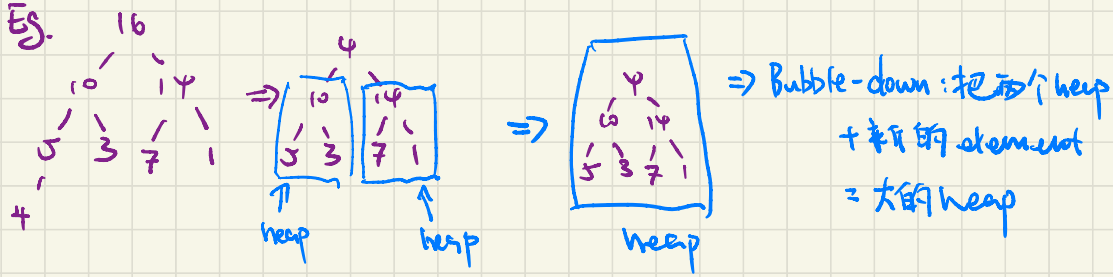
Swap

Rec

Heapify:

bubble-down (index) → 从 index 开始 down fix

Eg.



1. Heap:

2个属性：

1. Complete binary tree ↔ array (parent / children get) ← 快
2. 对于所有 node，its priority > child's priority (优先级)

Heap 是 PQ 的一种

Ex

```
boolean isMinHeap(ArrayList<int> arr) {
1. // check root 以下所有节点
2. // 0 ~ size()-1
3. // check i => parentIndex : parent < i ,  childIndex : i > childs.
```

$\frac{(i-1)}{2}$

$2i+1$ and $2i+2$

2. Heapify：把一个不是 heap 的 array 变成 heap array （有优先顺）

方法1：把 array 的所有 element，insert 到 heap 里

得到的 heap 里的 array 就是 heap array.

$O(N \log(N))$

↓ n个 elements insert　　↘ insert

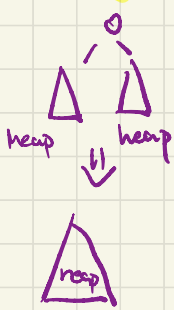
方法2：（变成 O(N) time）　本质

Bubble down：找出 Current 和 children 最大/最小，跟 current 交换，

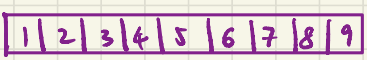
再 → 最大那也继续 bubble down.

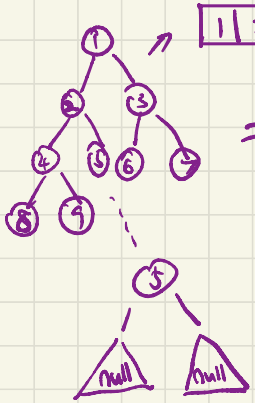
出现在 heap 的 remove 的 fix 中

fix：把不是 heap 的 array 修复成一个 heap.



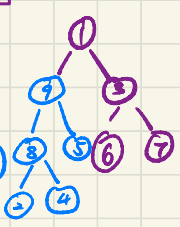
* 把两个 heap 和任意一个元素合成一个大的 heap

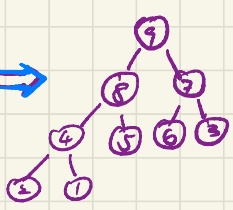


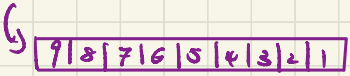


=) max heap =>

(by bubble down)







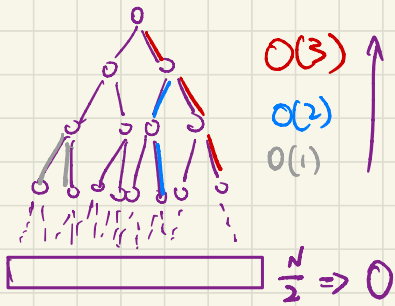
Q:

Bubble down 从哪开始？：最后一个 parent ⇒ $(\frac{n}{2}-1)$

从哪停？：到 root 结束（parent index = 0）

Eg. 4 → 7 → 8 → 9

Big O: O(N) Why?



$\frac{N}{2} \Rightarrow O(1) \Rightarrow N\text{个} O(1) > O(\text{root}) \Rightarrow O(N)$

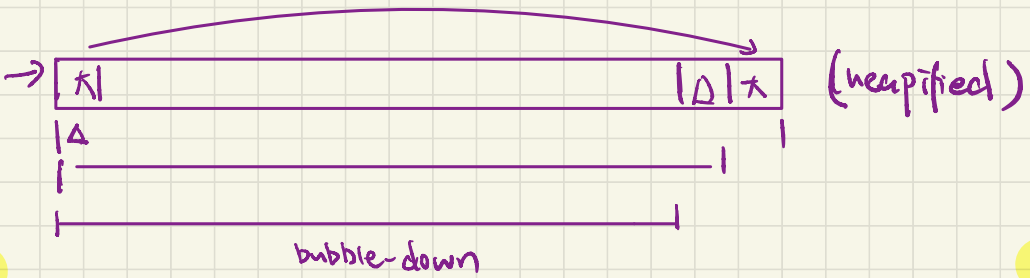
3. Heap Sort.

学过的：bubble - selection - insertion - merge - heap (省一个)

bubble-down 不一定排对序

1. Heapify O(N)

2.



把最后一位和 root 交换，

执行 bubble-down，把 heap 修复好

循环这个 logic，直到 heap 只剩一位为止

O(1)

O(N) from 1

+

O(logN)

x

O(NlogN)

O(NlogN)

任何对比类的 sort 算法，运行效率一定不快 O(NlogN)

4. Hash tables:

Previous: AVL: insert() / remove() / search() ⇒ O(log N)

→ 空间与时间 trade off: ① AVL Space Complexity: O(1) 不需额外空间

(空间 < 时间) Weight 空间换时间

Q: 为什么 previous 数据结构都是最快 O(log(n))
A: Comparsion.

How to get rid of this Comparsion? ⟹ Hash (不比较)

Hash Function: (数据本身决定它放在什么位置)

把一个数据 (key) ⟹ array's index
f(x)

(Hash function 是给定的)

Tips: 不存在 prefect hash function

Eg.

hash(n) = n^2 → 位置

insert: 根据 data 算出位置, (hash(n) % array.size) 位置 O(1)

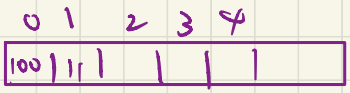
remove: 算出位置, delete, 返回 O(1)

search: 算出位置, 返回 O(1)

↳ Eg. $hash(n) = n^2$

insert (100)

insert (11)



Q: array's size 取多大?

Array Size 和数据 N 应该存在什么关系? (小 = 大)

对数据 → hash(n) % array.size => index是一样的 ↓

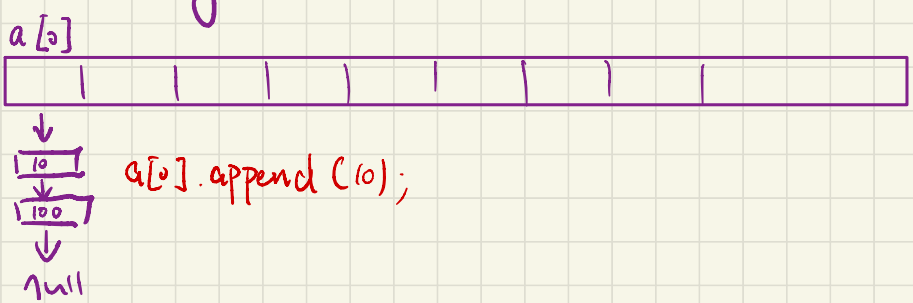
Array Size 应远大于 N. ↑ size↑ 重复near↓

≫ (N^2) (standard)

如果2个数据一样:

Hash Collision: 1. Open addressing 2. Chaining (Linked list)

Chaining:



把每一个array格子变成一个LL, 如果有collision, 把元素 append 到LL里,

Search也可找到, 把每一位LL走一遍

Open addressing (放在其它位置)

假设放 n=100 ⇒ index = 0 → arr[0] 已有 10

1. Probing: (向后顺延固定长度)

i) linear probing: if index 放不了, 那么 index + 1
if index+1 放不了, 那 index + 2, ... etc

ii) quadratic probing: if index 放不了, 那 index + 1^2
if index 放不了, 那 index + 2^2, ... etc

iii) Cube Probing: same 1^3, 2^3

2. double hashing: (后沿续)   hash 2 给定的

if index 放不了, 看 index + hash2(n)
if index + hash2(n) ..., 看 index + 2hash2(n)

分析一题:

real world chaining √ · open addr

如果出现碰撞: 影响的效率 (main table ⇒ O(1))

目标: 减小碰撞的概率 (控制在很小的范围里)

① hash function 找到的 index 分布越均匀越好, 要像随机分布

② array size * = 2

load factor： Collision后不是O(1)

= 已经放入hash table的元素数量 / hash table array.size

(占用比例)

Eg. 放入10个，array.size = 16

load factor = 10/16 = 0.625 ⇒ 62.5%

Q1：对于hash table来说，load factor有没有可能为1(100%)

A1：可以

① chaining ✓

② Open addressing ✗

if 使用 open addressing，一般load factor > 0.6，→ 扩容

chaining → 0.9 与语言设定

扩容后需重新算位置？

需要，位置 (hash(x) % array size)，全都新算位置

why O(1)

∵ 虽有collision，需solve (collision ≠ O(1)，但load factor在较小范围，hash function有效 ⇒ Pr(碰撞)↓

∴ O(1) is average.

Graph:

In tree / LL 最基本的单元：node edge. □ → □

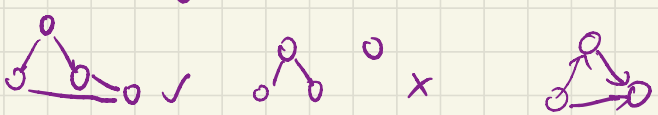
任何 graph 都是由 node 节点和 edge 边 构成的

e.g. 

tree: no cycle，所有 tree 都是 graph

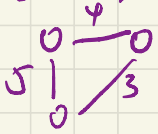
(只要由点和边构成的都是图像) Graph

Connected graph:



Directed / undirected

weighted / unweighted graph (距离)

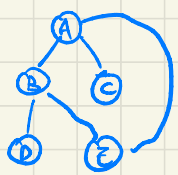


How Code ⟺ graph Connected?

1. adjacency list:

不好找 element

```
public class Node{
    T data;
    ArrayList<Node>; ⇒ Neighbors.
```



A : [T data | B | C | C]

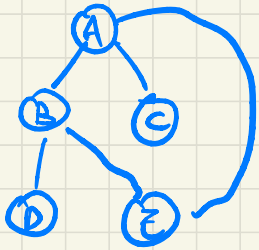
LL → LL
LL is fine

V : # of vertices (节点数)

E : # of edges (边数)

Adj list memory space : O(V+E)

2. adjacency Matrix



	A	B	C	D	E
A		1	1	0	1
B	1		0	1	1
C	1	0		0	0
D	0	1	0		0
E	1	1	0	0	

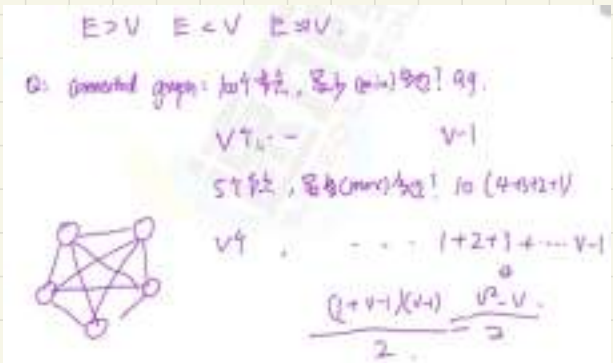
1 连
0 不连

memory space: O(V²)

Adj list VS Adj matrix

1. E, V是否存在数量上的关系

list O(V+E) Matrix O(V^2)



当graph E很多时 (dense graph) Matrix

当graph E 少时 (Sparse graph) List.

We have done: insert remove

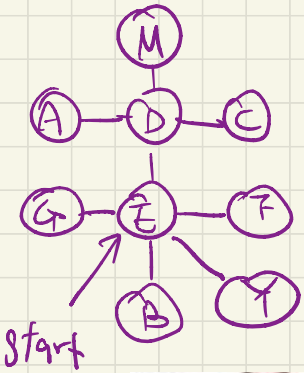
Now Search

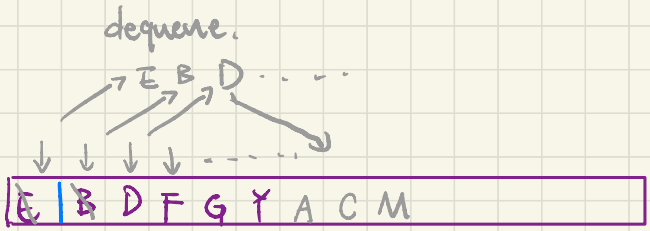
traversal : Graph traversal ⇒ BFS/DFS

DFS: Stack来存遍历节点

BFS: Queue

BFS



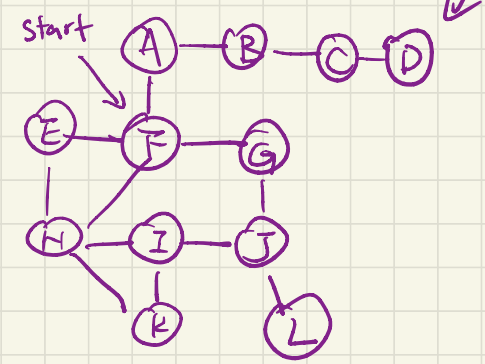


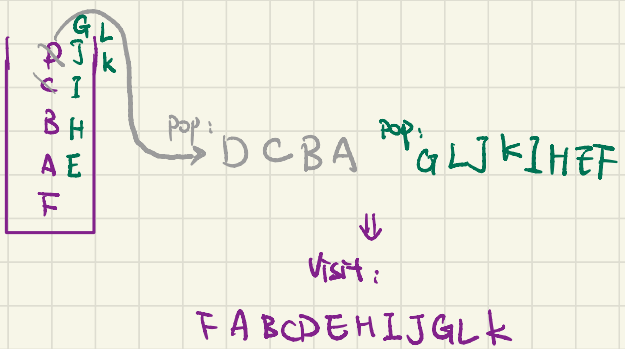
访问顺序

E B D F G Y A C M

1. 创建一个 queue，把起点标记为已走过（visited）并加进 queue 里
2. While queue is not empty:
   (1) Peek queue top node（看一眼 queue 头上的元素）
   (2) 把所有跟这个元素相连的点且没有被标记过的元素进行标记，并加进 queue 里。
   (3) 把 queue 进行 dequeue（刚刚这个元素去掉）

DFS





visit:

F A B C D E H I J G L K

1. 创建一个 stack，把起点标记为已走过（visited）并加进 stack 里。
2. While stack is not empty:
   (1) 看一眼（peek）stack 头上的元素
   If 如果有一个相邻的节点没有被标记，对它进行标记并加进 stack 里
   Else 如果所有的相邻的节点都被标记了，就 stack pop

Selection Sort: $O(n^2)$

把 array 分为两个部分，Sorted (空的)，unSorted

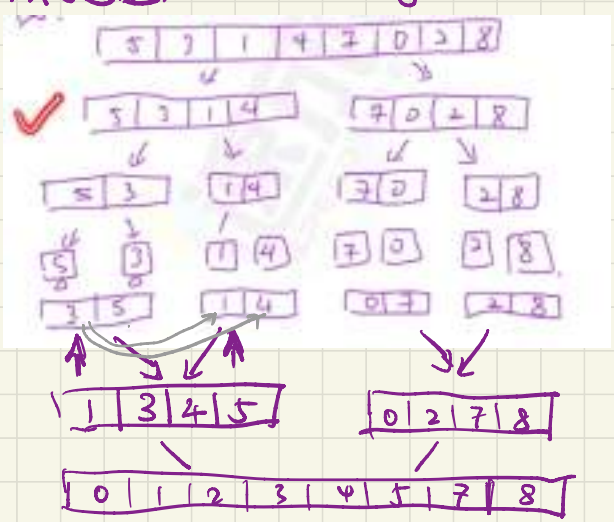
从 unsorted 找出最小值，放到 Sorted，重复，直到 unsorted 里没有 elements


```java
void selectionSort(int arr[], int n)
{
    int i, j, min_idx;
    for (i = 0; i < n-1; i++)
    {
        min_idx = i;
        for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
            min_idx = j;

        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

Merge Sort:

核心思想：$O(N\log N)$



Cons: 需要额外空间

Space Complexity $O(N)$

Big O: $O(N\log N)$

Merge Sort 不是 inPlace Sort

InPlace Sorting

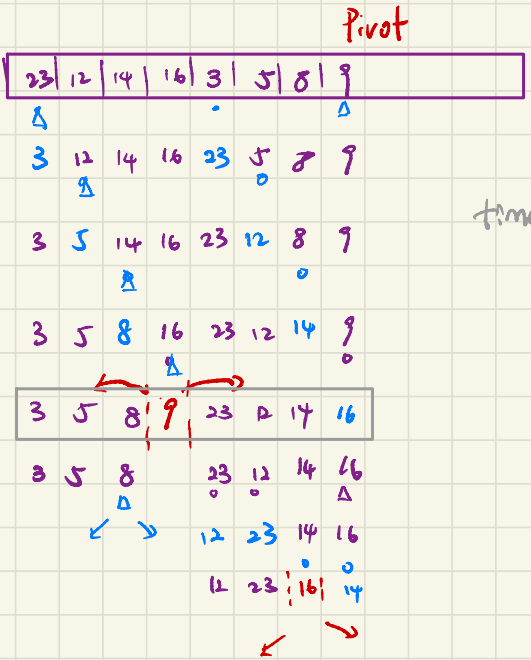
Sort 数据

Quick Sort: Pivot

灵感来自: Partition ()

↳ 分区, 站队 (array分成两个部分, 左边比某数小, 右边大)

Eg.



1. 找第一个比"9"大的数 (可换)
2. 找第一个比"9"小的数

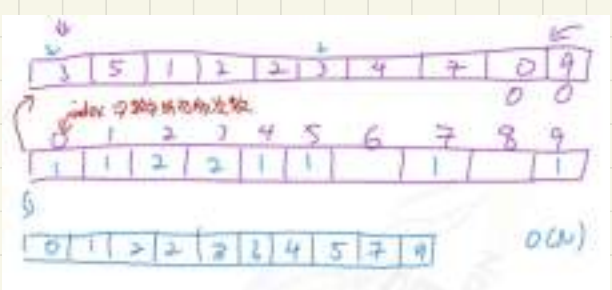
time: O(nlog(N)) => O(N^2) (worst)

↓

1 2 3 4 5 6 7 8 9 10 以排好情况 →N

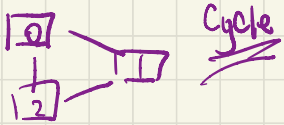
→9

Counting / Bucket Sort => O(N) not normal sort



radix Sorting

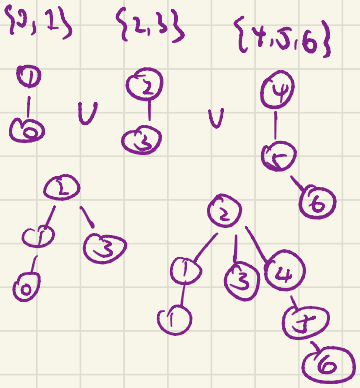
Union-Find Algorithm



undirected edge

0: {0} 1: {1}

子集 子集



We assume: 0 是 1 parent

0: {0,1}

or

→ 1 是 0 的 parent

1: {0,1}

⇓

{1,1,2}

1-2

1: {0,1} 2: {2}

→ We assume: 2 是 1 的 parent

2: {0,1,2},

{2,2,2} = {2}

子集

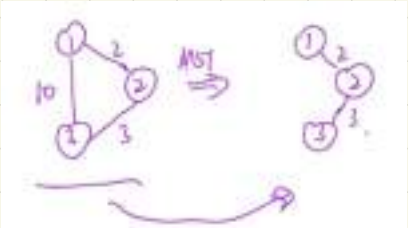
MST

3. Minimum Spanning Tree (MST)

A minimum spanning tree (MST) is defined as a spanning tree that has the minimum weight among all the possible spanning trees.

Properties of a Spanning Tree:

1) The spanning tree holds the below-mentioned principles:
2) The number of vertices (V) in the graph and the spanning tree is the same.
3) There is a fixed number of edges in the spanning tree which is equal to one less than the total number of vertices (E = V-1).
4) The spanning tree should not be disconnected, as in there should only be a single source of component, not more than that.
5) The spanning tree should be acyclic, which means there would not be any cycle in the tree.
6) The total cost (or weight) of the spanning tree is defined as the sum of the edge weights of all the edges of the spanning tree.
7) There can be many possible spanning trees for a graph.

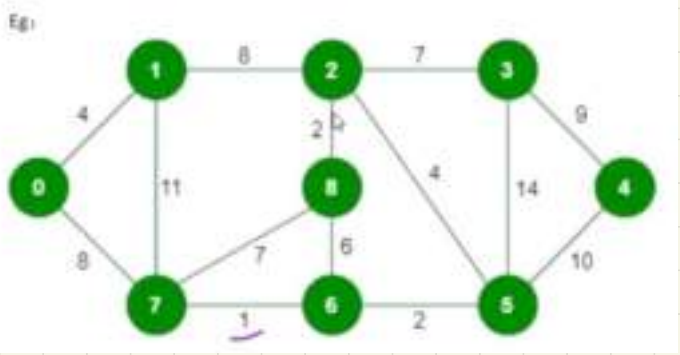


Kruskal's Algorithm ← leading to

4. Kruskal's Algorithm

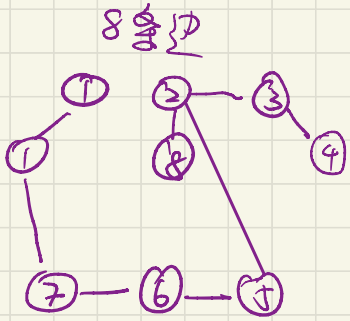
步骤:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.



MST

Weight	Source	Destination	
1	7	6	✓
2	2	8	✓
2	6	5	✓
4	0	1	✓
4	2	5	✓
6	8	6	× 사
7	2	3	✓
7	7	8	사
8	0	7	✓
8	1	2	×
9	3	4	○
10	5	4	✓
11	1	7	
14	3	5	



Review

- Hash table (Java 自带) O(1)
  - ↳ time 快, space 多
    - ↓
    - 不对比
  - hash Function: key → Array Index
  - ~~hash(n) = n²~~

最快算法 Sort: n log(n)

Heap / Merge /

↓

more space

bucket Sort → O(n)

O(1) - insert: 找到 data 算出位置, (hash(n) % array size)

O(1) - remove: 算出位置 删除

O(1) - search, 算出位置, 返回

- open Addressing → 1. Probing (eg. index + 1) → 2. Double hashing (eg. index + hash2(n) % array size)
- Chaining (DDL 也不选加快, ∵ 无 index)

→ avoid Collision
- hash Function
- more space

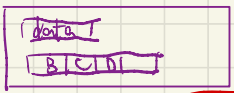
- load factor
  - 放入 Hash table 的元素数 / hash Table array Size.
  - 放 10, array size = 16 → 10/16 = 60%

↳ Tips: LL 可以 element's # > array size

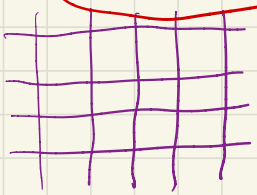
↳ Resize Array → 重新算位置 (hash(n) % array Size)

Graph.

Adjacency List: Space : $O(V+E)$

↳ Array A: 

Adjacency Matrix: Space : $O(V^2)$



Tips: E多, Matrix省
E少, List省

DFS / BFS { List : $O(V+E)$; Matrix : $O(V^2)$ }
Time

Space – $O(V)$

DFS/BFS (从小走大)

DFS