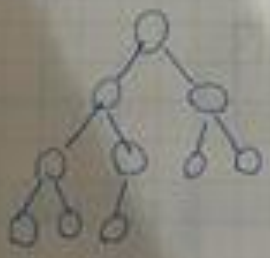
# What is Heap

property 1: it is a complete binary tree

Property 2: the value stored at a node is greater (smaller) or equal to the values stored at the children

ex:



Date

# How to build a heap

```
template < class ItemType >
struct HeapType {
  void ReheapDown (int, int);
  void ReheapUp (int, int);
  ItemType *elements;
  int numElements;
};
```

- ReheapUp()

  heap property is violated at the right most node at the last level (bottom) of the tree

- ReheapDown()

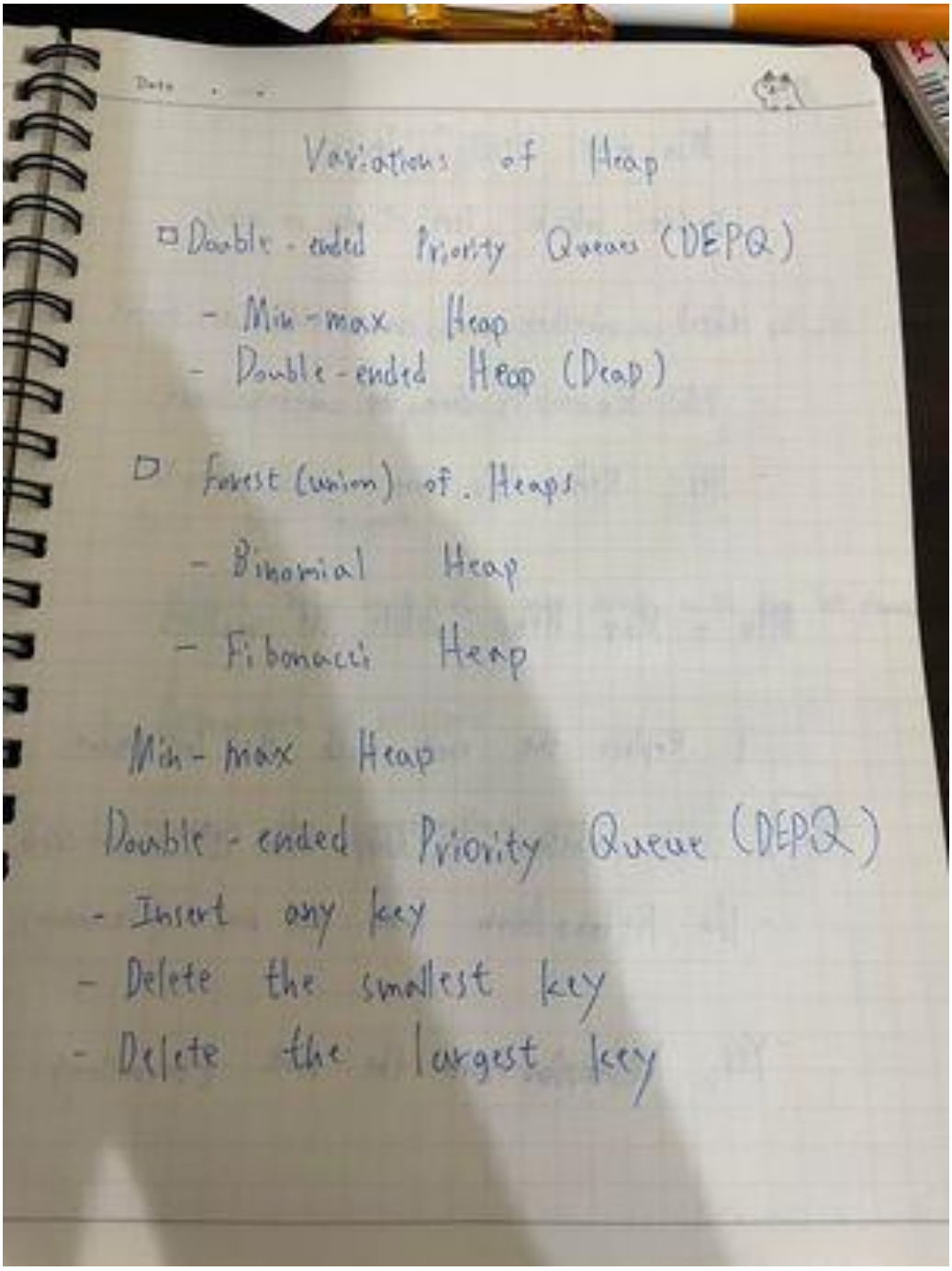
  heap property is violated at the root of the tree

pqDelete(): $O(\log n)$

Remove the largest element from heap

(1) copy the bottom rightmost element to the root
(2) Delete the bottom rightmost node
(3) Fix the Heap property by calling ReheapDown

```
HeapInsert ()
{ if(size >= MAX_HEAP)
    throw HeapException("Heap Full");
  items[size] = newItem;
  int place = size;
  int parent = (place-1)/2;
  while ((parent != 0) && (items[place] > items[parent]))
                                  Max heap
  {
    HeapItemType temp = items[parent];
    items[parent] = items[place];
    items[place] = temp;
    place = parent;
    parent = (place-1)/2;
  }
  ++size;
}
```



Variations of Heap

- Double-ended Priority Queues (DEPQ)
  - Min-max Heap
  - Double-ended Heap (Deap)
- Forest (union) of Heaps
  - Binomial Heap
  - Fibonacci Heap

Min-max Heap

Double-ended Priority Queue (DEPQ)

- Insert any key
- Delete the smallest key
- Delete the largest key

## Min-max Heap : Insert

1, Decide which level → min or max

2, check whether to swap with its parent

- No: Reheap Up from the current node
- Yes: Reheap Up from its parent

## Min - Max Heap : Delete the smallest

1, Replace the root with the last element

2, check whether to swap with its smaller child

- No, Reheap Down from the root (recursion)
- Yes, Reheap Down from the root (recursion)

# DEAP

Insert:

1, Examine the corresponding nodes: Left < Right

2, Reheap Up if necessary (recursion)

Delete the largest

1, Replace the root of max-heap with the last element

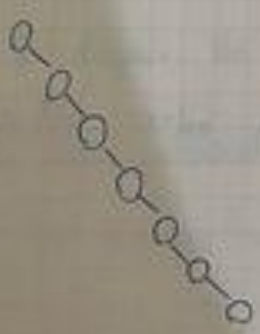
2, Reheap down if necessary

3. Examine the corresponding nodes: left < Right

# Binary Search Tree

- The efficiency of the BST implementation of the ADT table is related to the tree hight
  - Hight of a binary search tree of n items
    Max: n , Min: $\lceil \log_2(n+1) \rceil$
    Sensitive to the order of insertions and deletions

ex:



# Balance BST

- 2-3 tree
- 2-3-4 tree
- AVL tree
- Red-black tree

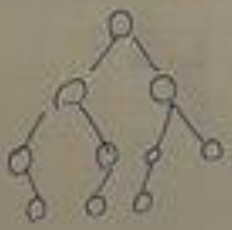
## 2-3 tree:

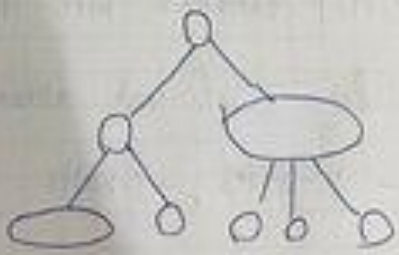
- All external nodes (leaves) are at the same level
- Degree of each internal node = 2 or 3
  - 2-nodes, 3-nodes
- Main operations
  - search == BST
  - Insertion
  - Deletion

# 2-3 Tree

- Have 2-nodes and 3-nodes
- Are general trees, not binary trees
- Are never taller than a minimum-height binary tree
  - A 2-3 tree with n nodes never has height greater than $\lceil \log_2(n+1) \rceil$

BST vs 2-3 Tree





# Red - black Tree

- A red - black tree
  - Represent each 3-node and 4-node in a 2-3-4 tree as an equivalent bST
  - A bst to represent a 2-3-4 tree
    - Maybe skewed ← rotations like AVL tree
  - Has the advantages of a 2-3-4 tree, without the storage overhead

## Red - Black Tree : Basics

- A binary search tree where each pointer is associated with a color, either red or black
  - External pointer (of a leaf) must be black
  - Every pointer to a new-added node must be red!

# Red-Black Tree : Insertion

□ Main idea

1, split of a node with two red pointers (like the 4node in a 2-3-4 tree) occur only on the path from the root to a leaf

2, set the pointer to a new-added node as red

3, Rotate if there are two consecutive red pointers

Ex : 45 85 10 70 20 60