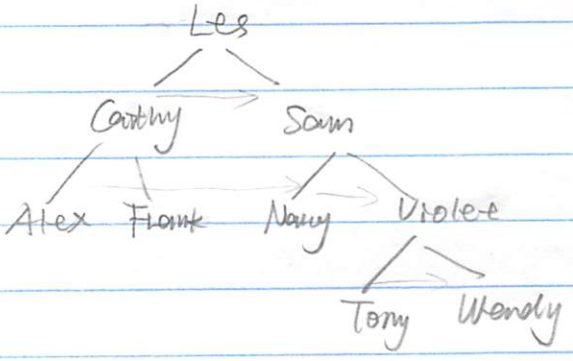
Breadth-first (level-Traversal). queue.

```
Level Traversal (tree)
  if tree = nil: return
  Queue q;
  q.Enqueue(tree)
  while not q.Empty():
      node ← q.Dequeue()
      print(node)
      if node.left ≠ nil:
          q.Enqueue(node.left)
      if node.right ≠ nil:
          q.Enqueue(node.right)
```



Output: Les Cathy Sam Alex Frank Nancy Violet Tony Wendy

Queue: Les | Cathy Sam | Alex Frank | Nancy Violet | Tony Wendy

Summary:
- Trees are used for lots of different things
- Trees have a key and children
- Tree walks: DFS (pre-order, post-order, inorder) and BFS
- When working with a tree, recursive algorithms are common
- In CS, trees grow down.

Dynamic Arrays.

Problem: static arrays are static! (determine size of array)
once declare them, they can't change size → int myArray[100];

Semi-solution: dynamically-allocated arrays. — allocate, determine size of array at run time
int *myArray = new int[size];

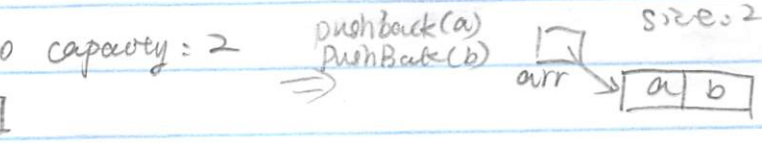
Problem: might not know the max size when allocating an array.

Solution: dynamic arrays (resizable arrays)
idea: store a pointer to a dynamically allocated array,
and replace it when a newly-allocated array as needed
如果 add to many 超过了原先 dynamically allocated array 的 size, 则新建一个更大的 array, 将原先的 elements 都 copy 过去, pointer 指向新的 array.

Definition: Abstract data type
Operations: Get(i): returns element at the location i*
Set(i, val): set val in index i.
PushBack(val): add val to the end
Remove(i): remove element at i*
Size:

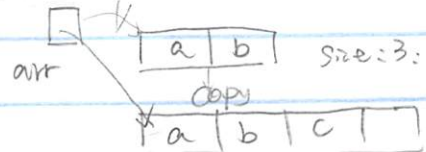
Implementation:
Store: arr: dynamically-allocated array
capacity: size of dynamically-allocated array
size: number of elements currently in the array

Dynamic Array Resizing:



⇒ PushBack(c).

不够了, 扩capacity
double old-capacity



Get(i) O(1)

```
if i<0 || i≥size:
    ERROR: index out of range
return arr[i];
```

Set(i, val) O(1)

```
if i<0 || i≥size:
    ERROR
arr[i]=val
```

PushBack(val): O(n)

```
if size = capacity {
    allocate new arr[2×capacity];
    for i from 0 to size-1:
        new_arr[i] ← arr[i]
    free arr
    arr ← new_arr;
    capacity ← 2×capacity;
}
arr[size] ← val
size ← size+1
```

Remove(i) O(n)

```
if i<0 || i≥size:
    ERROR
for j from i to size-2:
    arr[j] ← arr[j+1]
size ← size-1
```

Size() O(1)

```
return size.
```

Common Implementations:

C++ : vector

Java : ArrayList

Python : list → no static Array

Summary:

- Unlike static arrays, dynamic arrays can be resized
- Appending a new element to a dynamic arr is often O(1), But can take O(n) (扩容)
- Some space is wasted

Amortized Analysis

↓ Dynamic Arr only resize so often: many O(1) operations are followed by a O(n) operation → resize.

Definition: Amortized cost: Given a sequence of n operations,

$$\frac{\text{Cost(n operations)}}{n}$$

① Aggregate Method (brute-force sum)

ex: Dynamic Array: n calls to PushBack, start from an empty array.

Let c_i = cost of the i'th insertion

$$c_i = \underbrace{1}_{\text{write}} + \begin{cases} i-1 & \text{: if } i-1 \text{ is a power of 2 (resize)} \\ 0 & \text{: otherwise} \end{cases}$$

Total amortized cost: $\frac{\sum_{i=1}^{n} c_i}{n} = \frac{n + \sum_{j=1}^{\lfloor \log_2(n-1) \rfloor} 2^j}{n} = \frac{O(n)}{n} = O(1)$

Worst case cost is still $O(n)$ But amortized cost is $O(1)$

② Banker's Method.

- Charge extra for each cheap operation
- Save extra charges as tokens in your data structure (conceptually)
- Use the tokens to pay for expensive operations

Like an amortizing loan:

ex: Dynamic Array: n calls to pushBack starting from an empty array.

Charge 3 for each insertion: 1 token is the raw cost for insertion

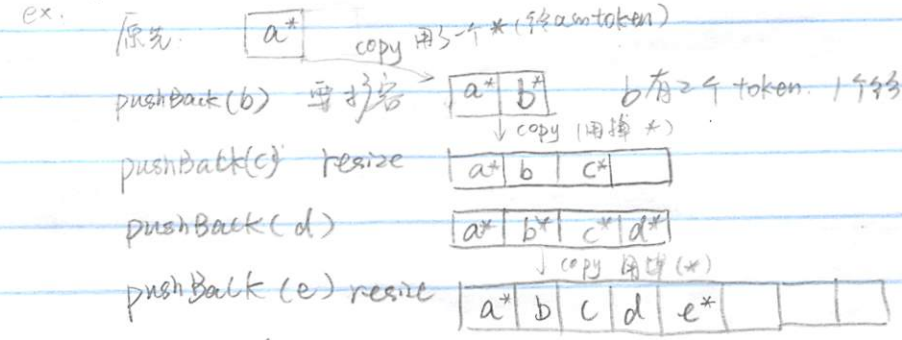
- Resize needed: To pay for moving the elements.

use the token that's present on each element that needs to move

- Place one token on the newly-inserted element,

and one token $\frac{\text{capacity}}{2}$ elements prior

ex.



∴ Total amortized cost: $\frac{3 \times n}{n} = 3 = O(1)$

像高中物理，把个球拿到山顶，(再扔下) potential energy (势能) 一直增加 最高，再扔下时，potential energy 变小，转换成动能

③ Physicist's Method

- Define a potential function, Φ (phi) which maps states of the data structure to integers.

Rules: ① $\Phi(h_0) = 0$ h – data structure 0: time 0.

② $\Phi(h_t) \geq 0$

- Amortized cost for operation t:

$$c_t + \Phi(h_t) - \Phi(h_{t-1})$$

(after) (before)

c_t: true cost of t; $\Phi(h_t) - \Phi(h_{t-1})$: change of potential between before doing the operation and after doing the operation

- Choose for Φ so that:
  - if c_t is small, the potential increases (save for future cost)
  - if c_t is large, the potential decreases by the same scale (pay for that work)

- The cost of n operation is: $\sum_{i=1}^{n} c_i$

- The sum of the amortized costs is $\sum_{i=1}^{n} (c_i + \Phi(h_i) - \Phi(h_{i-1}))$

$$\sum_{i=1}^{n} (c_i + \Phi(h_i) - \Phi(h_{i-1}))$$

$$= c_1 + \Phi(h_1) - \Phi(h_0)$$

$$+ c_2 + \Phi(h_2) - \Phi(h_1)$$

$$+ c_3 + \Phi(h_3) - \Phi(h_2)$$

$$\vdots$$

$$+ c_n + \Phi(h_n) - \Phi(h_{n-1})$$

$$= \sum_{i=1}^{n} c_i + \Phi(h_n) - \underbrace{\Phi(h_0)}_{=0} = \sum_{i=1}^{n} c_i + \Phi(h_n) \geq \sum_{i=1}^{n} c_i$$

at least the sum of true cost

ex: ~~Dynamic array: n calls to PushBack~~

Let $\Phi(h) = 2 \times size - capacity$

- $\Phi(h_0) = 2 \times 0 - 0 = 0$
- $\Phi(h_i) = 2 \times size - capacity \geq 0$ (since $size \geq \frac{capacity}{2}$)

a. Without resize when adding element i

Amortized cost of adding element i:

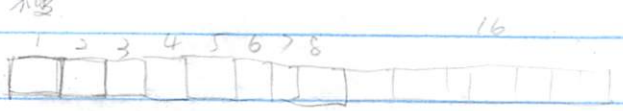
$$c_i + \Phi(h_i) - \Phi(h_{i-1}) \quad (cap_i = cap_{i-1})$$

$$= 1 + 2 \times size_i - cap_i - (2 \times size_{i-1} - cap_{i-1})$$

$$= 1 + 2 \times (size_i - size_{i-1})$$

$$= 3$$

Quiz 1: 24 PushBack 打怪. 24 PopBack 補. → no reallocate.



max final capacity ⇒ 32 先pushBack再PopBack

min final capacity ⇒ 1 1 pushBack 1 PopBack. 1

Quiz 2: PopBack → reallocate if size ≤ $\frac{\text{capacity}}{2}$

ex: before popBack(): size=5, capacity=8

after popBack(): size=4, capacity=4

$C_i = 1 + \begin{cases} i-1 & \text{if } i-1 \text{ is power of } 2 \\ 0 & \text{otherwise.} \end{cases}$

example of n operations requires $O(n^2)$ copies

b. With resize when adding element i

Let $k = \text{Size}_{i-1} = \text{cap}_{i-1}$; so $\text{Size}_i = \text{Size}_{i-1} + 1 = k+1$, $\text{cap}_i = 2\text{cap}_{i-1} = 2k$

Then $\Phi(h_{i-1}) = 2\,\text{Size}_{i-1} - \text{cap}_{i-1} = 2k - k = k$

$\Phi(h_i) = 2\,\text{Size}_i - \text{cap}_i = 2(k-1) - 2k = 2$

∴ Amortized cost of adding element i:

$C_i + \Phi(h_i) - \Phi(h_{i-1})$

$= (\text{Size } i) + 2 - k$

$= k+1+2-k = 3$

⇒ a. b 모두 3

- Alternatives to Doubling the Array Size

We could use some different growth factor (1.5, 2.5, ...)

OR use some constant amount? (add by particular num) ✗

No ⟶ Cannot use Constant Amount

∴ If we expand by 10 each time, then:

let C_i = cost of i'th insertion

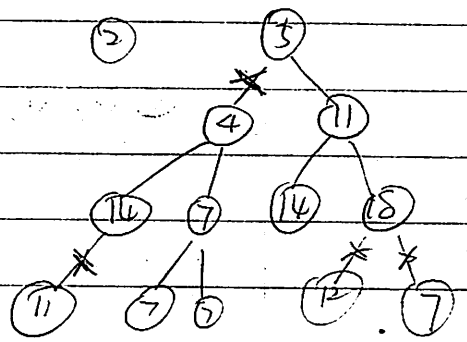
$C_i = \begin{cases} i & \text{if } i-1 \text{ is a multiple of } 10 \\ 1 & \text{otherwise} \end{cases}$ $\quad 1+2+3+\cdots+\frac{n-1}{5} = O(n^2)$

Amortized cost: $\frac{\sum_{i=1}^{n} C_i}{n} = \frac{n + \sum_{j=1}^{\frac{n-1}{10}} 10j}{n} = \frac{n + 10\sum_{j=1}^{\frac{n-1}{10}} j}{n} = \frac{n + 10\,O(n^2)}{n} = O(n)$

① n insert
n^2 Extract Max.

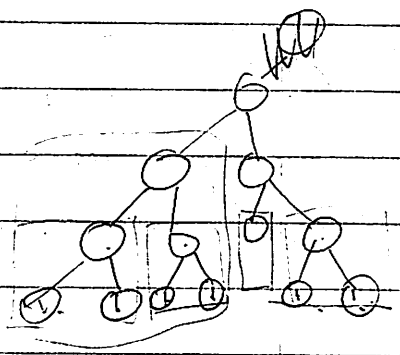
	Sorted array	array	Binary Heap
	$O(n)$	$O(1)$	$O(\log n)$
	$O(1)$	$O(n)$	$O(\log n)$
	$nO(n)+n^2O(1)$	$nO(1)+n^2O(n)$	$n(\log n)+n^2(\log n)$

n^2



4

② How many subtrees ~~complete~~



$6+3+1$

Priority Queues and Disjoint Sets

1. Priority Queue : introduction

1. Queue : an abstract data type : < pushBack (e)
   popFront ()

priority queue: a generalization of a queue where each element is assigned a priority and elements come out in order by priority

Typical use case : scheduling jobs & process jobs one by one
- add a job to the set of scheduled jobs : Insert(job)
- process a job with highest priority . get it by ExtractMax()

(Formally) : priority queue is an abstract data type supporting following operations:

① Insert (p) : adds a new element with priority p

② ExtractMax () : extracts an element with maximum priority

additional: ③ Remove (it) : removes an element pointed by an iterator it

④ GetMax () : returns an element with maximum priority (no changing the set)

⑤ ChangePriority (it, p) : changes the priority of an element pointed by it to p

Algorithms : Dijkstra's algorithm - finding shortest path in a graph
- Prim's algorithm:
- Huffman's algorithm
- Heap sort : sorting a given sequence

Array/List	Insert(e)	ExtractMax()
Unsorted	O(1)	O(n)
Sorted	O(n)	O(1)
☆ Binary heap	O(log(n))	O(log n)

2. Implementations

Unsorted Array/List (double)

Insert (e) running time: - add e to the end : O(1)

ExtractMax () running time: - scan the array/List : O(n)

Sorted Array (binary search)

- find a position for e (O(log n))
- shift all elements to right (O(n))
- insert e O(1)

∴ O(n)

- extract the last element : O(1)

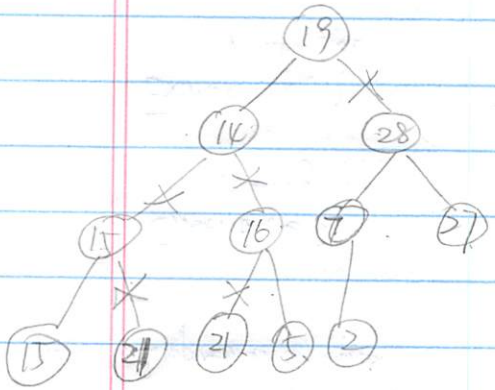
Sorted ☆ List

Insert(e) : find a position for e (O(n)) (can't use binary search) + insert (O(1))
⇒ O(n)

ExtractMax() : O(1)

4. a complete binary tree represented by an array

[19, 14, 28, 15, 16, 7, 27, 15, 21, 21, 5, 2]



5. 10^5 elements is stored in a complete 5-ary tree

How many comparisons a call to Insert() will make

$\log_d n$

7. Extra index

$d \log_d n$

8. d-ary

d^0

d^1

d^2

$(i-1) >> 1$

$= i/2$

$= 2i-1$

Priority Queue: Binary Heap.

1. Definition: Binary max-heap is a binary tree (each node has 0, 1 or 2 children) where the value of each node is at least the value of its children.

   OR: for each edge of the tree, the value of parent ≥ the value of children

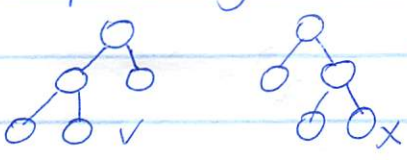
   Tree Height remark: define it to be equal to the number of edges on the longest path from root to leaf

   这样定义 = 最长路径上nodes个数 - 1 ∴ 1个Node in tree. height = 0

2. Basic Operations.

- GetMax() : return the root 的 value — O(1)
- Insert(e) : 1. attach the new node (e) to any leaf
  O(tree height) 2. if violate the heap property, to fix it by letting the node sift up
- Sift up() : 3. sift up :→ swap the problematic node with its parent until the property is satisfied
  (invariant) 不变性 : { heap property is violated on at least one edge
  { this edge gets closer to the root while sifting up
- ExtractMax() : 去掉最大. 与 GetMax() 不同. 会改变整个 heap
  1. replace the root with any leaf
  2. this may violate heap property
- Sift down() : 3. to fix it, let the problematic node sift down
  O(tree height) 4. swap the problematic node with larger child until the heap property satisfied
- ~~ChangeProper~~
- ChangePriority : change the priority of the node and let the changed node
  O(tree height) sift up or sift down depending on whether its priority increased or decreased
- Remove() : 1. change the priority of the element to ~~∞~~ ∞, let it
  O(tree height) 2. sift up and the extract maximum (∞)
  ↳ call ExtractMax()

2. Complete Binary Tree — if all its levels are filled except possibly the last one which is filled from left to right



$5^0+5+5^2+5^3+\cdots+5^n$

$= \frac{5^{n+1}-1}{5-1}$ $\quad \frac{5^{n+1}-1}{4} = 10^5$

$= 5^{n+1} = 10^5 \times 4$

$5^x = 400000$

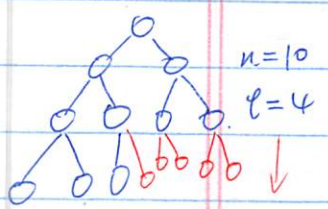
$\log_5 400000 = \frac{40}{0.69} = 6.5$

$7-\log_7 n$ $\quad \log_7 n$ $\quad \frac{6}{\log 10} \times 7$ $\quad 0.8$ $\quad 0.85$

[1] First advantage: Low Height.

Lemma: a complete binary tree with n nodes has height at most $O(\log n)$

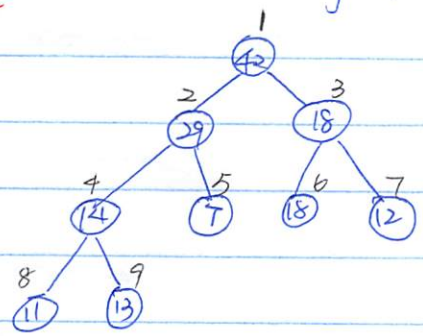
Proof: – Complete the last level to get a full binary tree on $n' \geq n$ nodes and the same number of levels ℓ.



– note that $n' \leq 2n$

– Then $n' = 2^{\ell} - 1$ and hence $\ell = \log_2(n'+1) \leq \log_2(2n+1)$
$= O(\log n)$

[2] Second Advantage: Store as Array.



Given the index i of node

then: parent index: $parent(i) = \lfloor \frac{i}{2} \rfloor$

Given the parent index is i

then $leftchild(i) = 2i$

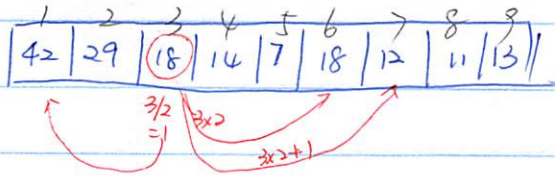
$rightchild(i) = 2i+1$

ex: given $i=4$

it's parent node index is $4/2 = 2$

it's left child is $4 \times 2 = 8$

right child is $4 \times 2 + 1 = 9$



We need to pay cost for the 2 advantages

the cost is to keep the tree complete

2 basic operations (binary heap) modify the shape of the tree:

① Insert ② ExtractMax (Remove calls this)

Keeping the Tree Complete.

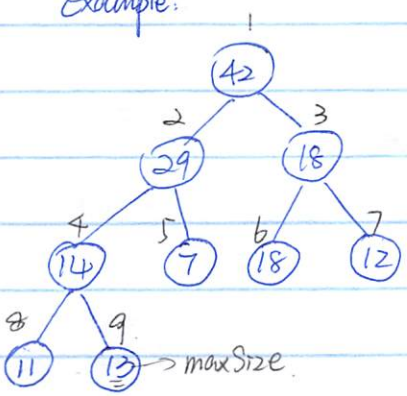
① Insert an element: insert as a leaf in the leftmost vacant position in the last level and let it sift up.

② ExtractMax(): replace the root by the last leaf in the last level and let it sift down

Pseudocode for binary heap:

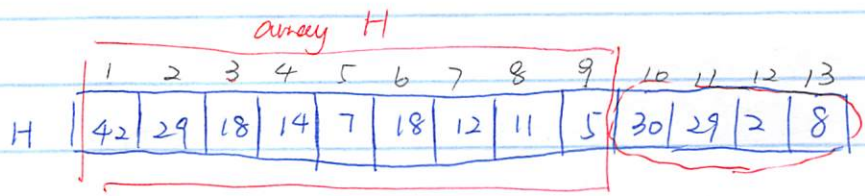
General setting: { maxSize — the maximum number of elements in the heap
size ≤ maxSize the size of the heap (actual size of heap)
H[1.... maxSize] is an array of length maxSize
where the heap occupies the first size elements

Example:



Size = 9

maxSize = 13



Parent(i) return $\lfloor \frac{i}{2} \rfloor$

LeftChild(i) return 2i

RightChild(i) return 2i+1

- SiftUp(i):

```
while i>0 and H[Parent(i)] < H[i]:
    swap H[Parent(i)] and H[i];
    i ← parent(i);
```

- SiftDown(i)

```
maxIndex ← i
① l ← LeftChild(i)
  if ( l ≤ size and H[l] > H[maxIndex]:
      maxIndex ← l
② r ← RightChild(i)
  if ( r ≤ size and H[r] > H[maxIndex]:
      maxIndex ← r
③ if i ≠ maxIndex:
    swap H[i] and H[maxIndex]
    SiftDown(maxIndex)
```

- Insert (p)

```
if size == maxSize:
    return ERROR
size = size + 1
H[size] ← p
SiftUp (size)
```

- Remove (i)

```
H[i] ← ∞
SiftUp (i)
ExtractMax()
```

- ExtractMax()

```
result ← H[1]
H[1] ← H[size]
size ← size - 1
SiftDown (1)
return result
```

- ChangPriority (i, p)

```
oldp ← H[i]
H[i] ← p
if p > oldp:
    SiftUp (i)
else:
    SiftDown (i)
```

Summary: The resulting implementation of complete binary heap's:

- fast: all operations work in time $O(\log n)$. (GetMax() is $O(1)$)
- space effient: we store an array of ~~proper~~ priorites; on the fly
  parent-child connections are not stored, but like computed
- easy to implement: only a few lines of codes

Using binary heap to design the heap sort algorithm (, fast, space efficient sorting algorithm)

HeapSort (A[1, 2, ... n])

1. Create an empty priority queue
2. for i from 1 to n:
   Insert (A[i]);
3. for i from n downto 1:
   A[i] ← ExtractMax();

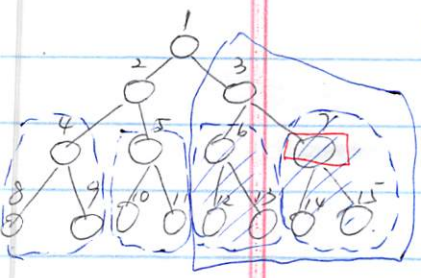
- The resulting algorithm is
  { Comparison-based
  and has running time $O(n \log n)$
  asymptotically optimal 渐进最优
- use a smart data structure - binary heap

Natural generalization of selection sort:
instead of simply scanning the rest of array to find maximum value

Disadvantage: Not in-place.
use additional space to store the priority queue

In-place heap sort algorithm { running time $O(n \log n)$; no additional space.

① For this, we will first turn a given array into a binary heap by permuting its elements.



Turn an array into a heap BuildHeap(A[1....n])

```
size ← n
for i from ⌊n/2⌋ down to 1:
    SiftDown(i)
```

- there're 15 nodes ⇒ so there're 15 subtrees
- for the leaves in the tree, they're subtrees with only a root for each (8~15) ⇒ heap property is satisfied (no edges in these subtrees)

Node: 7 → 6 → 5 → 4 (subtrees)
3 → 2 (subtrees)
1 (heap)

- so the first node where property might be related is node 7

details:
- We repair the heap property going from bottom to top.
- Initially, the heap property is satisfied in all the leaves (i.e. subtrees of depth 0)
- We then start repairing the heap property in all subtrees of depth 1
- When we reach the root, the heap property is satisfied in the whole tree
- Running time: $O(n \log n)$ (SiftDown(i))

```
HeapSort(A[1, ... n])
BuildHeap(A)        {size = n}
repeat (n-1) times:
    swap A[1] and A[size]
    size ← size - 1
    SiftDown(1)
```

Quick Sort 平均 run-time is $O(n \log n)$ Worst case $O(n^2)$
而 HeapSort worst case is $O(n \log n)$ 平均也是 $O(n \log n)$
But in practice, 一般来说 QuickSort, 如果没改到 $O(n \log n)$ 再转向 HeapSort

64 128

8 16 32

1 2 4

2^0 2^1 2^2 2^3 2^4 2^5

$n/2$

$1 \times 2^{n+1} - 1$

$1 + 2^1 + 2^2 + 2^3 + \cdots + 2^n$

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

1 2 4

1 1

2 3

4 8

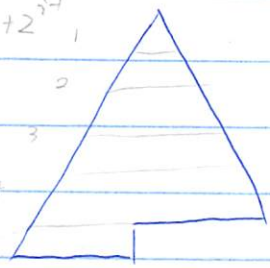
8 15

16 31

32 63

- Building a Heap running time:

发布对每个处理Node作 SiftDown 时 Time 为 $\log_2 n$. 很多点 leaves 进行的 SiftDown() 都远远小于 $\log_2 n$



$2^0+2^1+2^2+\cdots+2^{i-1}$

$=2^i-1=n$

$\Rightarrow i=\log(n+1)$

2^{i-1}

# nodes	T(SiftDown)
1	$\log_2 n$
2	$\log_2 n-1$
⋮	⋮
$\le n/4$	2
$\le n/2$	1

n-size

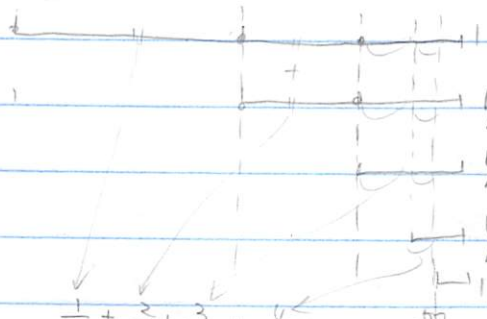
height $< \log_2 n$

$$T(\text{BuildHeap}) \le \frac{n}{2}\times 1 + \frac{n}{4}\times 2 + \frac{n}{8}\times 3 + \cdots n + 2\cdot(\log_2 n - 1) + \log_2 n$$

$$\le n\cdot\sum_{i=1}^{\infty}\frac{i}{2^i} = n\cdot 2 = 2n$$

$\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\frac{5}{32}+\cdots\frac{i}{2^i}$

$\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\cdots=\sum_{k=1}^{\infty}\frac{1}{2^k}=1$



$\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{4}{16}+\cdots=\sum_{k=1}^{\infty}\frac{k}{2^k}=1+1/2+\frac{1}{4}+\frac{1}{8}+\cdots=1+1=2.$

- Partial Sorting. { Input: An array $A[1\cdots n]$, an integer k, $1\le k\le n$
  Output: The last k elements of a sorted version of A

⇒ Can be solved in $O(n)$ if $k=O(\frac{n}{\log n})$!

```
PartialSorting(A[1...n])
    BuildHeap(A)
    for i from 1 to k:
        ExtractMax();
```

if $k=O(\frac{n}{\log n})$ ⇒ $O(n)$

⇒ Running time: $O(n)+kO(\log n)=O(n+k\log n)$

Final Remarks

- 0-based Arrays: Parent(i) → return $\lfloor \frac{i-1}{2} \rfloor$
  Left Child(i) → return $2i+1$
  Right Child(i) → return $2i+2$
- Binary Min-Heap: a binary tree (0.1.2 children each node) where
  父节点小于等于the value of each node is at most the values of its children
- d-array heap: in d-array heap nodes on all levels (except for possibly the last one) have exactly d children.
  - height: $\log_d n$
  - running time of SiftUp: $O(\log_d n)$
  - running time of SiftDown: $O(d \log_d n)$

Disjoint Sets.

- Naive Implementations

⇒ Overview:

maze 中有两点问是否有 path 连接?

某个点可以发出 all points joint 是否 ~~reach~~ 包含另一点 A (set)

Definition: A disjoint-set data structure supports following operations.

- MakeSet(x): Create a singleton set {x} (只包含1个element x)
- Find(x): returns ID of the set containg x:

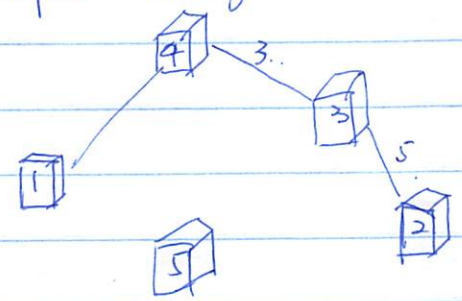
  ID 作为一个 set 的标识
  - 如果 x and y lie in same set, then Find(x) = Find(y) → 同一个set 同一个ID
  - otherwise, Find(x) ≠ Find(y)
- Union(x, y): merges two sets containg x and y.

  (如果先 call Union(x,y), 再 Find(x) 和 Find(y), 则有 Find(x)=Find(y))

⇒ Solve Maze problem

```
Preprocess (maze)
    for each cell c in maze:
        MakeSet(c);
    for each cell c in maze:
        for each neighbor n of c:
            Union(c, n);
IsReachable(A, B)
    return Find(A) = Find(B)
```

Example: Building a Network



1. MakeSet(1); MakeSet(2) ....
2. Find(1) ≠ Find(2) ≠ Find(3) ...
3. Union(3, 4). → Find(3) = Find(4)
4. MakeSet(5)
5. Union(3, 2) → Find(1) = Find(2) → False
6. Union(1, 4) 6. Find(1) = Find(2) → True

Naive Implementation

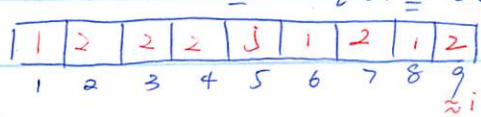
For simplicity, assume that our n objects are just integers 1, 2, ··· n

Our sets are just sets of integers

- using the smallest element of each set as the ID

- Use array. Smallest [1 . . . n]: Smallest[i] stores the smallest element in the set that i belongs to

ex. {9, 3, 2, 4, 7} {5} {6, 1, 8} → 3个 sets.



Smallest of size 9

i属于的那个 set 中的(最小值)ID 为 smallest[i] 的值.

After 满足以上条件后:

MakeSet (i): (i为 ID. 即 smallest[i] = i → 此时 set 中第一个 element

smallest [i] ← i.

Find (i): (find the ID of the set containing i 即返回 smallest[i] 的值.

return smallest [i];

Union (i, j):

i_id ← Find (i)

j_id ← Find(j)

if i_id = j_id : return (说明 i 和 j 本来就在一个set)

~~if i_id ≠~~

if i_id ≠ j_id:

m ← min(i_id, j_id); (两个set合并后, 找到最小的id为 id) → m

for k from 1 to n: (将 smallest[] array 中每个 object 对应的原来的 i_id 或 j_id 均改成 m) → i

if smallest[k] in [i_id, j_id]:

smallest [k] ← m;

Running Time: O(n)

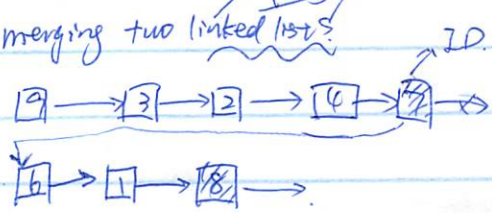
current bottleneck.

改用 Linked List 来表达 merging

Represent a set as linked list

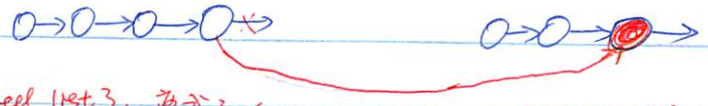
Easy → O(1) tails as ID.

ex: merging two linked lists



→ disadvantage: if linked list 变长, 时间变长 (每次查到 tail 都需 ID)

Merging = sets, linked list



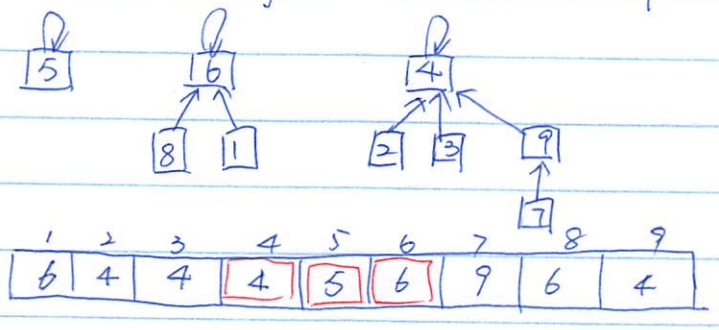
不是 linked list 3, 是这3个 tree, the root is the ID

Disjoint sets: Effecient Implementation

Trees: Definition:

- Each set is going to be represented as a rooted tree
- ID of a set is the root of the tree
- For each element i, its parent[i] will be stored in an array parent[1...n]
  Or if i is the root (parent[i] = i)

Sample:



MakeSet(i) → O(1)

parent[i] ← i

Find(i) → O(tree height)

while i ≠ parent[i] { (i不是root)

i ← parent[i] } (一直向上找, 直到找到root. root为ID

return i;