9/16/2018 Week3 Heap and Binary Tree

Why Data structure?

⇒ Organize data so that it can be accessed quickly and usefully.

E.g. lists, stack, queue, heaps, search tree, hash tables, bloom fliters, union-find, etc.

⇒ Different data structure is suitable for different task.

⇒ Choose the "minimal" data structure that supports all the operation that you need.

Level of Knowledge of Data structre

LEVEL 0 : Don't know data structure.

LEVEL 1 : Can hold conversation and discuss but not confident with implementation and choice.

LEVEL 2 : Comfortable of using data structure and can choose the right data structure for problem.

LEVEL 3 : Know how implimented and can write their own.

Heap

feature : ① Insertion : Add a new object to a heap.
② Extract min : Remove the object that has min key.
- Container for the objects that have keys.

⇒ Only support either "Extract min" and "Extract max".
If need both, use binary tree.

Running time

Insertion : $O(\log n)$
Extract-min : $O(\log n)$
Heapify : $O(n)$
Delete arbitrary : $O(\log n)$

n = # of object in heap.

Application

① Sorting (heap sort)

1. Insert all n array elements into a heap.
2. Extract-min to pluck out element in sorted order.

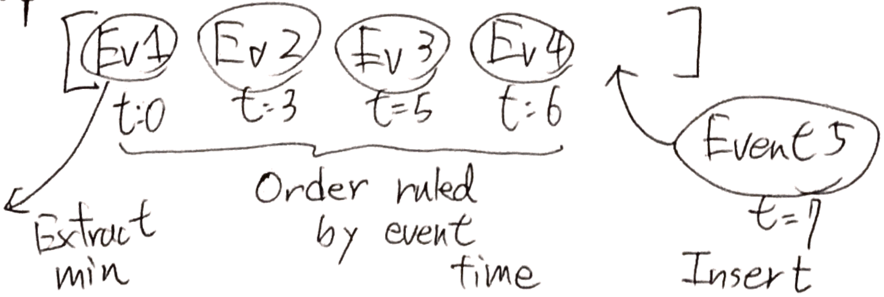
running time $\underbrace{n}_{\text{Do op. for every element}} \cdot \underbrace{\log n}_{\text{Insertion, Extraction.}}$ close to quick sort level

(2) Event manager

Simulation (e.g. for a video game)

- Objects = event records [action/update to occur at given time in the future]
- Key = time event scheduled to occur.

Heap



(3) Median Maintanence

Input: Sequence $x_1, \cdots, x_n$ of numbers, one-by-one

Output: At each time step i, the median of $\{x_1, \cdots, x_i\}$

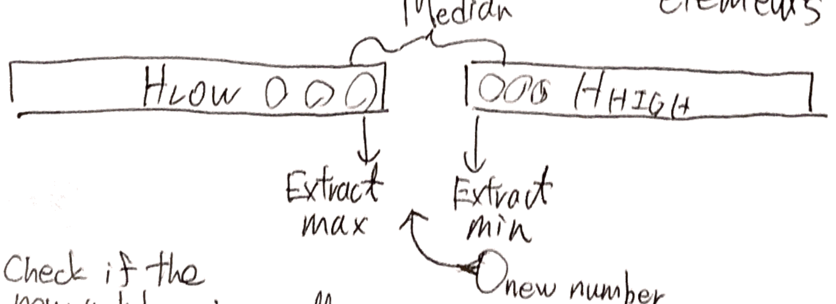
Constraint: Use $O(\log i)$ time at each step i.

⇒ Use two heap

H_{LOW} : Supports extract max

H_{HIGH} : Supports extract min

⇒ Maintain invariant that $\approx 1/2$ smallest (largest) elements in H_{LOW} (H_{HIGH})



(1) Check if the new number is smaller than H_{LOW} max or larger than H_{HIGH} min.

(2) Push into either of the heap.

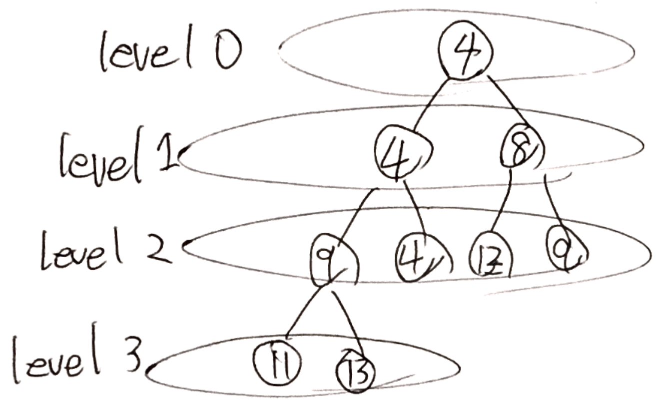
(3) If heaps got unbalanced, move min or max among heaps and maintain the balance.

4 Speeding up the dijkstra

⇒ Use priority queue instead of searching minimum key every time.

⇒ from $O(m \cdot n)$ to $O(m \log n)$

How to make heap?



- Tree like structre
- Min have to be on top.

⇒ But don't have to implement as a tree. (No need for pointer)

Array Implementation

A =

4	4	8	9	4	12	9	11	13
1	2	3	4	5	6	7	8	9

e.g. 9 in A[7].
7/2 round down = 3
A[3] = 8 is parents.

Note: $\text{parents}(i) = \begin{cases} i/2 & \text{if } i \text{ even} \\ \lfloor i/2 \rfloor & \text{if } i \text{ odd} \end{cases}$
ie, round down

v.v. children of i are $2i$, $2i+1$.

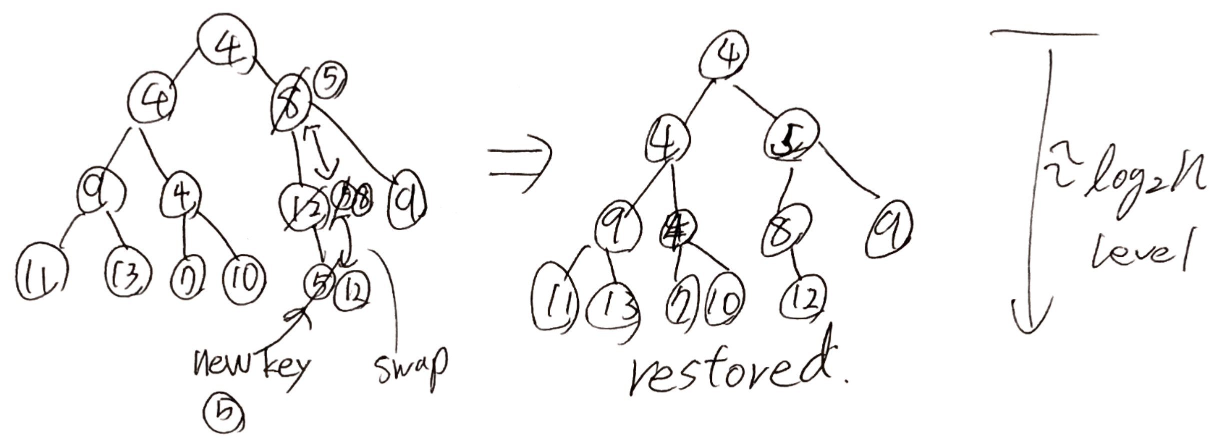
⇒ No pointer, No extra space, no need for traversing to get parents

⇒ Extremely fast!

Implementation of Insertion

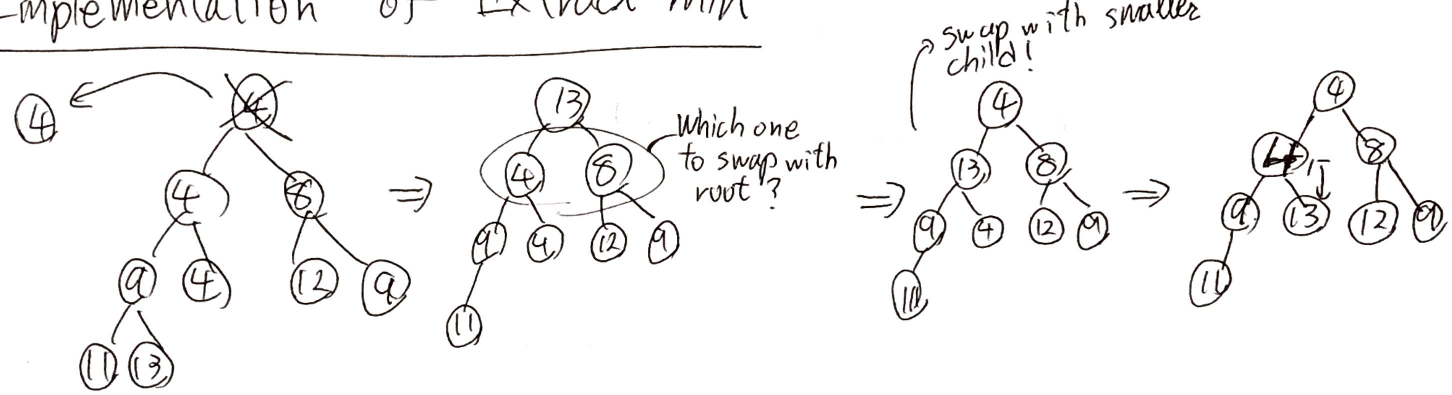
Step 1: stick new key at end last level

Step 2: Bubble-up k until heap property restored.



runtime = $O(\log n)$

Implementation of Extract-min



- The ~~least~~ less impactful node that can become the root node without changing other tree structure is 13.
- Bubble the 13 down by swaping with smaller child ~~every time~~ while it causing the violation of heap.

runtime = $O(\log n)$

Binary Tree

⇒ Dynamic version of sorted array.

Compare with sorted array.

3	6	10	11	17	23	30	36

Operation	Running time
search	$\Theta(\log n)$
select	$O(1)$
min/max	$O(1)$
PRED/SUCC	$O(1)$
RANK (Basically search)	$O(\log n)$
Output in order	$O(n)$

Pretty fast!

⇒ But for "Insertion" and "deletion", sorted array takes $O(n)$ time. (Slow...)

⇒ Purpose of the binary tree is to get as fast ~~f~~ operation with fast insertion and ~~del~~ deletion.

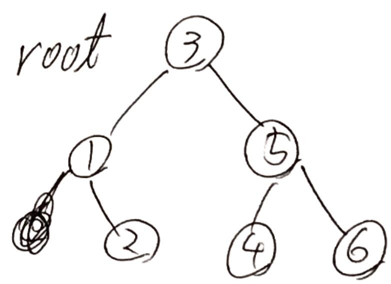
Binary tree operation

Operation	Running time
Search	$O(\log n)$
Select	$O(\log n)$
min/max	$O(\log n)$
PRED/SUCC	$O(\log n)$
RANK	$O(\log n)$
Output	$O(n)$
Insert	$O(\log n)$
DELETE	$O(\log n)$

Insert, DELETE: Only the Reason we use binary tree over sorted array.

Structure

- One node per key
- Most basic version: each node has
  - Left child pointer
  - Right child pointer
  - Parent pointer



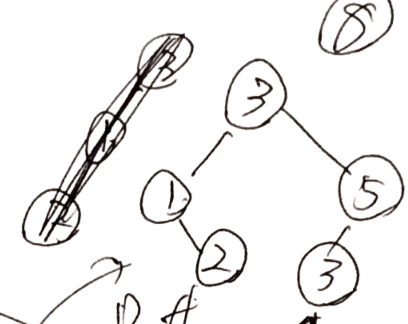
PROPERTY

· Apply to all nodes.

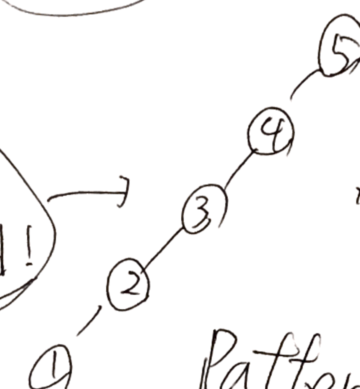
Height of BST

- Many possible trees for a set of keys.
- Height could be anywhere from $\sim \log_2 n$ to $\sim n$

valid!



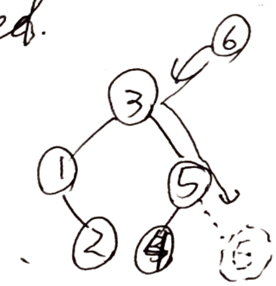
Also valid!



Search and Inserting

Search

- Start at the root
- Traverse left/right child pointers as needed.
  (left: if $k < key$ at current node; right: if $k > key$)
- Return node with key k or NULL



Insert

- Search for k (It will fail)
- Rewire the pointer the final null ptr to point to new node with key k.

⇒ Running time will be governed by the height of the tree.

⇒ Best when the tree is balanced.

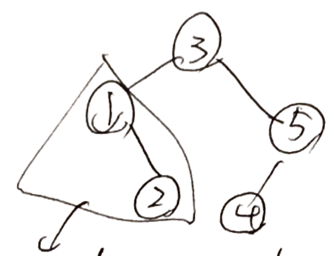
Min / Max

- start at root — right for max
- follow left child pointers until you can't anymore.
- return last node.

running (height)

Predecessor

- If k's left subtree nonempty, return max key in left structure e.g. Key = 3



Biggest ~~component~~ element = 2 → Predecessor.

- If k's left subtree is empty. Back track parents until find the node smaller than "k".

~~O(log n)~~ O(height)

In-order Traversal

- Let r = root of BST, with subtrees T_L and T_R
- Recursive on T_L and find MIN.
- print out key
- recurse on T_R

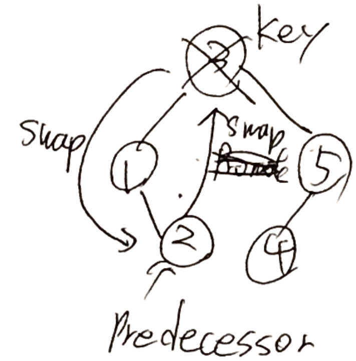
running O(n)

Deletion

Delete a key from a search tree.

- search for k

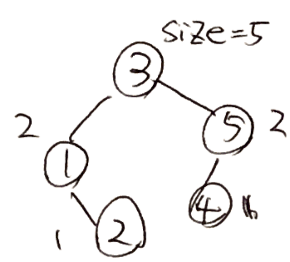
1. Easy case (k has no children)
   ⇒ Just delete k's node from the tree
2. Midium case (k has one children)
   ⇒ Unique node take over the position of k.
3. Difficult Case (k have two children)
   ⇒ Compute k's predecessor ℓ (left null search after right null search)
   ⇒ Swap k and ℓ!
   ⇒ Delete 3 (It is easy because 3 will not have right branch)



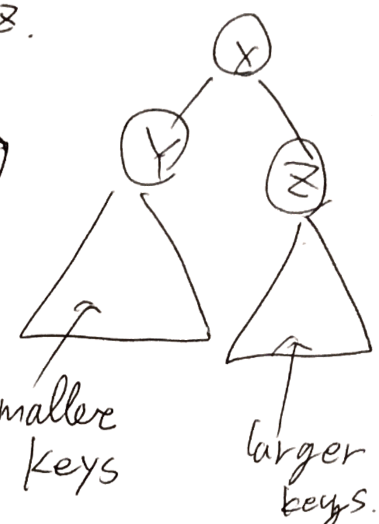
Select and rank

$O(\cancel{\log n}) + O(n) = O(n)$ (height)

- Store the size of the tree rooted at x, in node x.
- If x has children y and z,
  then size(y) + size(z) + 1 = size(x)
- Easy to keep size up-to-date during an insertion or deletion.



- Start at root x with children y and z.
- Let $a = size(y)$ [$a=0$ if x has no left child]
- If $a = i-1$ return x's key
- If $a \geq i$ recursively compute ith order statistic of search tree.
- If $a < i-1$ recursively compute $(i-(a+1))$th order statistic of search tree.



Running time $O($~~the~~ height$)$

Balanced Search trees

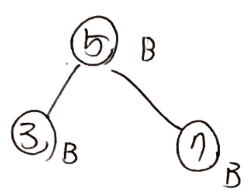
⇒ Multiple ways to balance

- Red-black tree
- AVL tree
- Splay tree
- B-tree

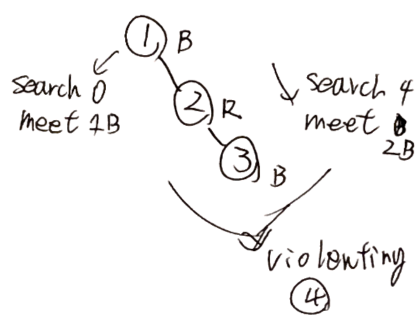
Red-Black Invariants

1. each node is red or black
2. Root is black
3. No 2 reds in a row
   [red node ⇒ Only black children]
4. Every root-null path has same number of black nodes.
   ↳ e.g. ansuccessful search.

Valid



Invalid



If the search tree meet these 1~4, height of the tree will be $2\log(n+1)$