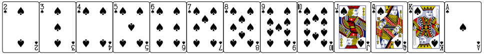
COMP 2011: Data Structures

Lecture 9. Binary Heaps

Dr. CAO Yixin

November, 2021



Review of Lecture 8

trees:	basic concepts
binary trees:	link-based storage, traversals
binary search trees:	insert, search, predecessor, successor

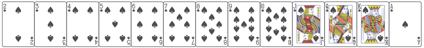
A binary tree is not a linear data structure,
all its operations are essentially following a linked list.

Tracing from root is easy, but tracing back is a pain.
Use a stack, or, recursion.

Preorder + Inorder uniquely determine a binary tree.

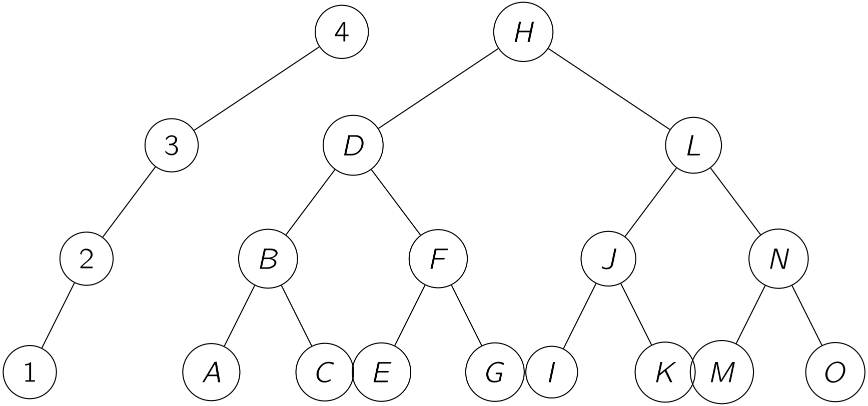
Inorder traversal: □□□X■■■■■

Preorder traversal: X□□□■■■■■

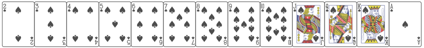


Trees are inherently recursive.

Self-balancing Search Trees



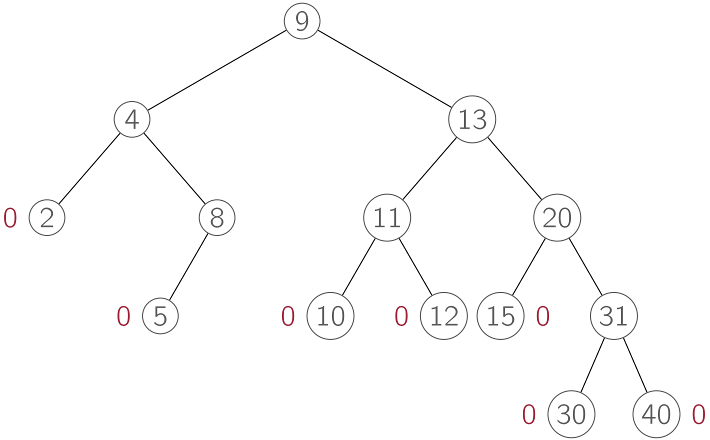
$n - 1$ $\qquad$ $log n$



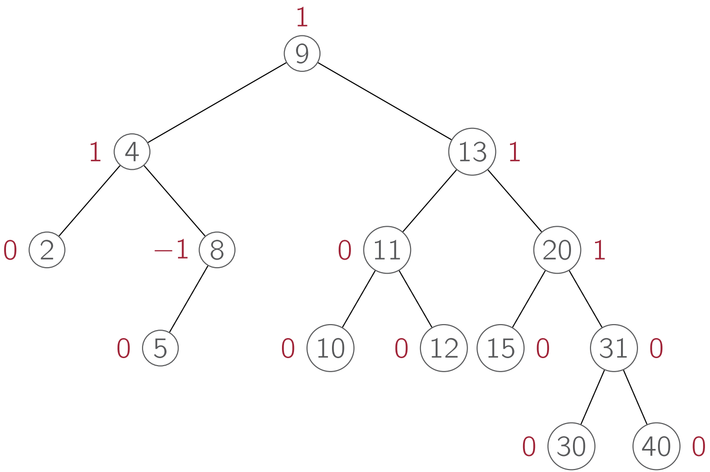
AVL trees

In an AVL (Georgy Adelson-Velsky and Evgenii Landis) tree ( ),

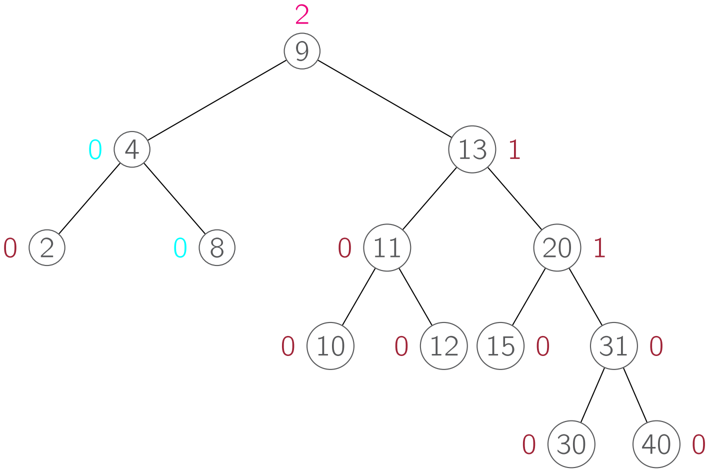
the heights of the two child subtrees of any node differ by at most one.



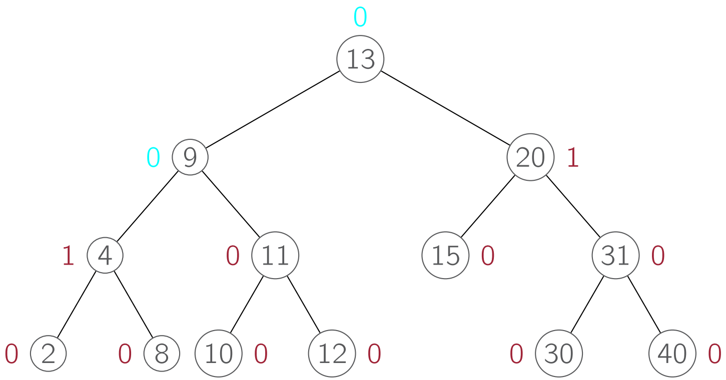
Each node has a balance factor $\in \{-1, 0, 1\}$:
height of right subtree $-$ height of left subtree



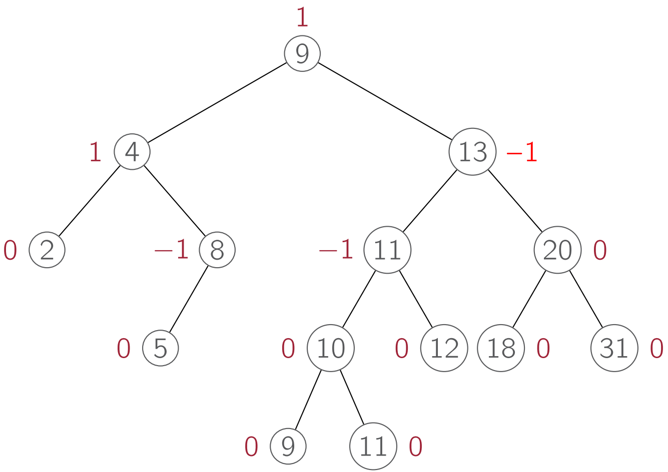
Each node has a balance factor $\in \{-1, 0, 1\}$:
height of right subtree $-$ height of left subtree



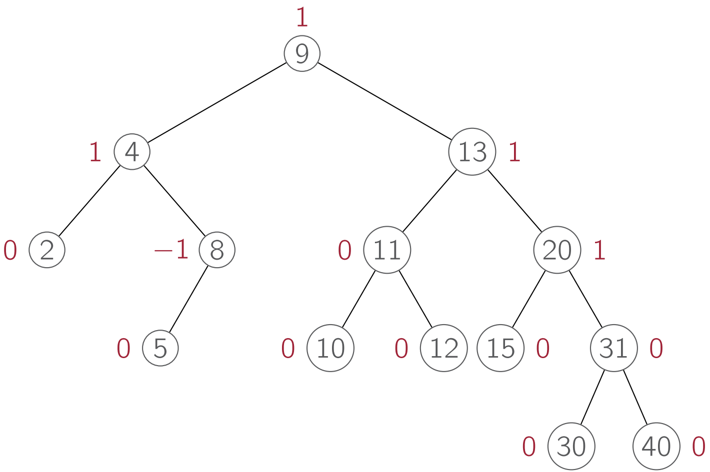
after deleting node 5, the balance factor the root is 2



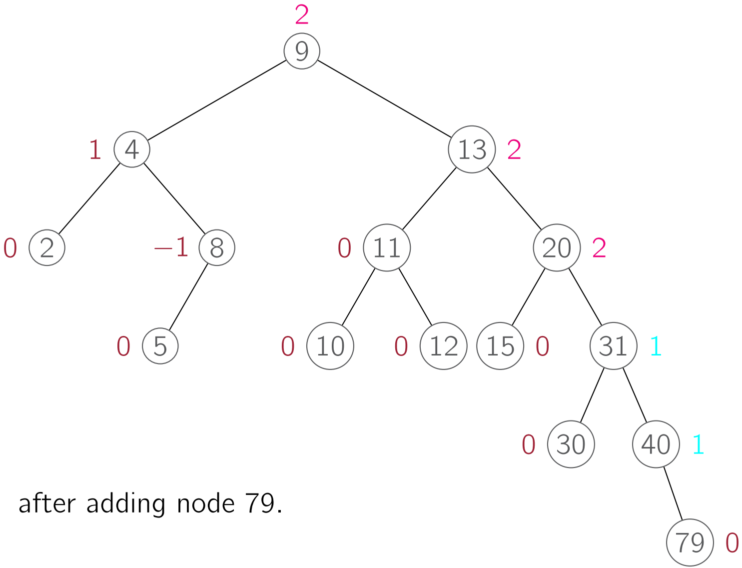
If the AVL tree property is violated after inserting or deleting a node, then we rearrange the shape of the tree using "rotations."



more complicated if bf(13) = −1

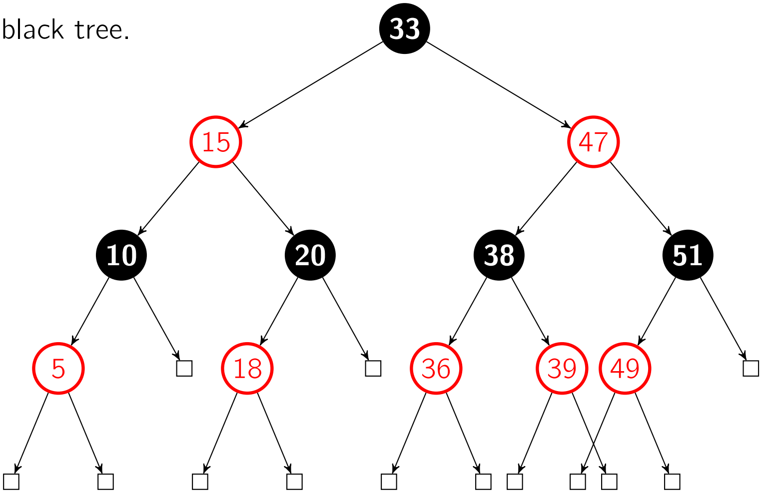


back to the original tree



after adding node 79.

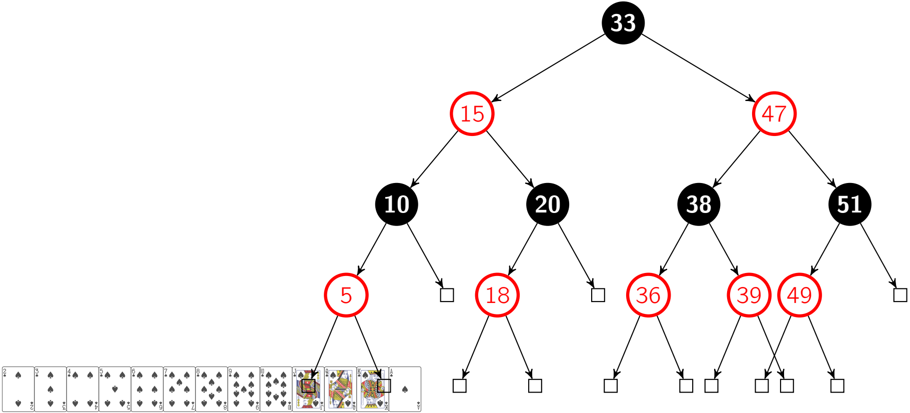
A red-black tree.



Red-black trees

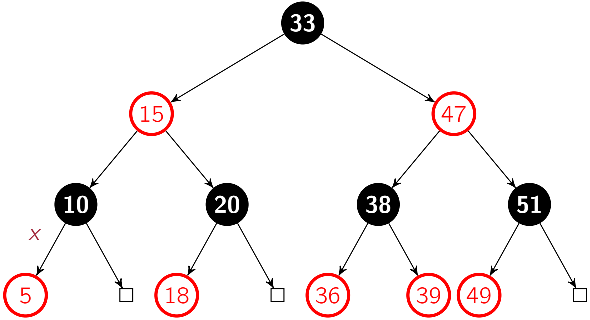
In a red-black tree (), every "real" node is given 0, 1, or 2 "fake" NIL children to ensure that it has two children; and every node is colored either red or black s.t.:

- every leaf node is black,
- if a node is red, then both its children are black,
- every path from a node to a leaf contains the same number of black nodes



Red-black trees

- In a red-black tree (🌐), every "real" node is given 0, 1, or 2 "fake" NIL children to ensure that it has two children; and every node is colored either red or black s.t.:
  - every leaf node is black,
  - if a node is red, then both its children are black,
  - every path from a node to a leaf contains the same number of black nodes
- Insertion and deletion are quite involved.



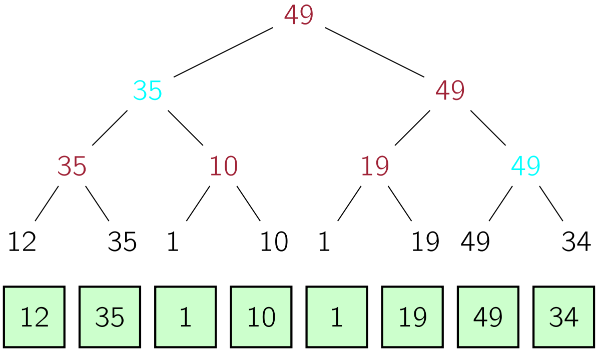
Self-balancing search trees

Theorem.
The depth of an AVL tree or a red-black tree on n nodes is $O(\log n)$.

An alternative idea (): all leaves are at the same level, but allow > 2 children.

Binary Heaps

A maximum tree (the tree we built to find a second)

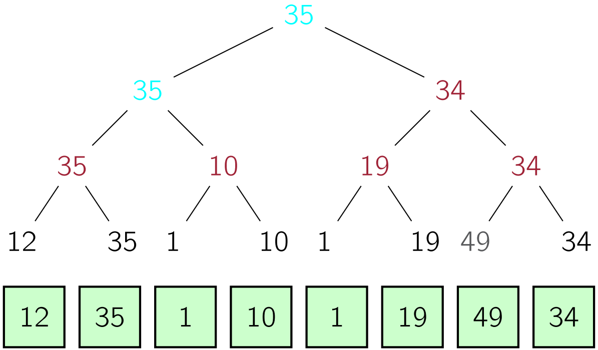


Each node store the larger of its children.
cyan: left. red: right.

Questions:

- Can you use this tree to sort?
- How many nodes in this tree?
- How to store this tree?

A maximum tree (the tree we built to find a second)

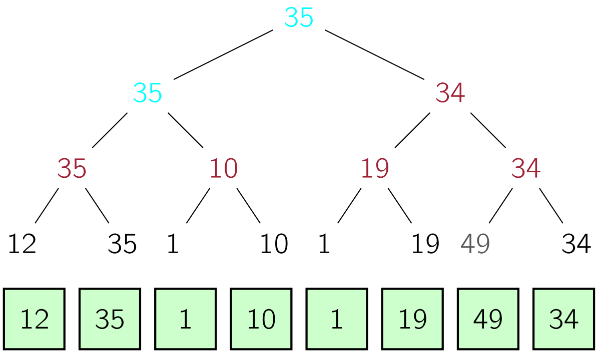


Each node store the larger of its children.
cyan: left. red: right.

Questions:

- Can you use this tree to sort?
- How many nodes in this tree?
- How to store this tree?

A maximum tree (the tree we built to find a second)

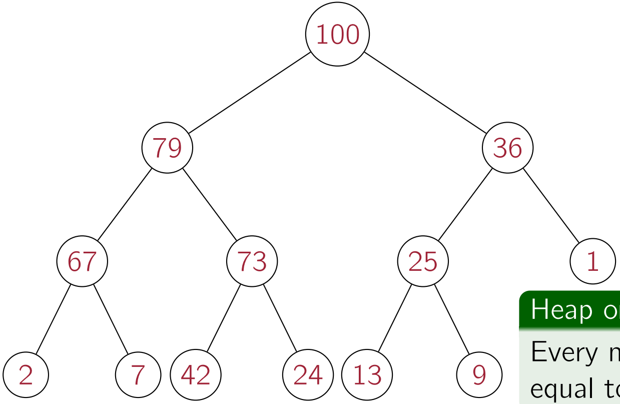


Each node store the larger of its children.
cyan: left. red: right.

Questions:

- Can you use this tree to sort?
- How many nodes in this tree?
- How to store this tree?

Binary heaps



Heap ordering property

Every node has a key greater than or equal to the keys of its both children.

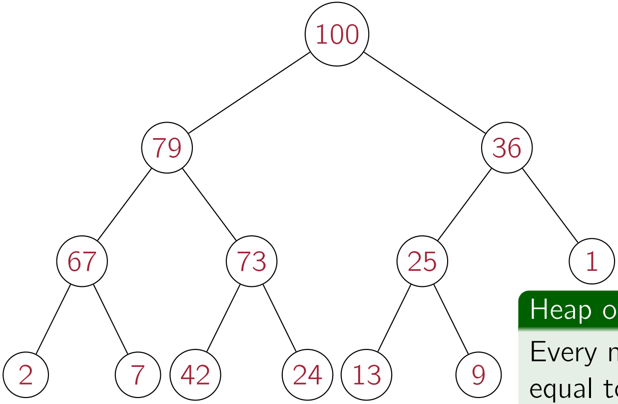
Complete binary trees:

Leaves are at the last two levels and those at bottom level are as far to the left as possible.

A *binary heap (or max-heap)* is

a complete binary tree satisfying the heap ordering property.

Binary heaps



Heap ordering property

Every node has a key greater than or equal to the keys of its both children.

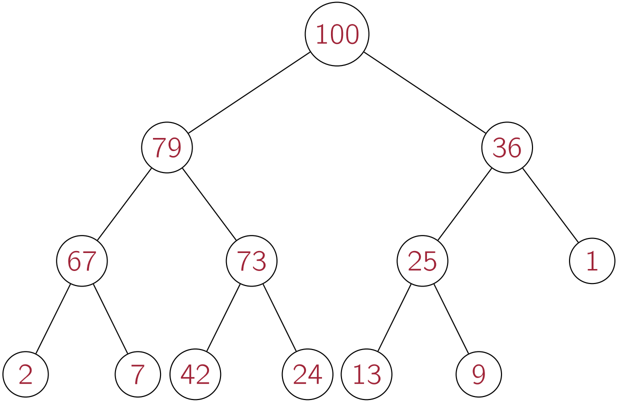
Complete binary trees:

Leaves are at the last two levels and those at bottom level are as far to the left as possible.

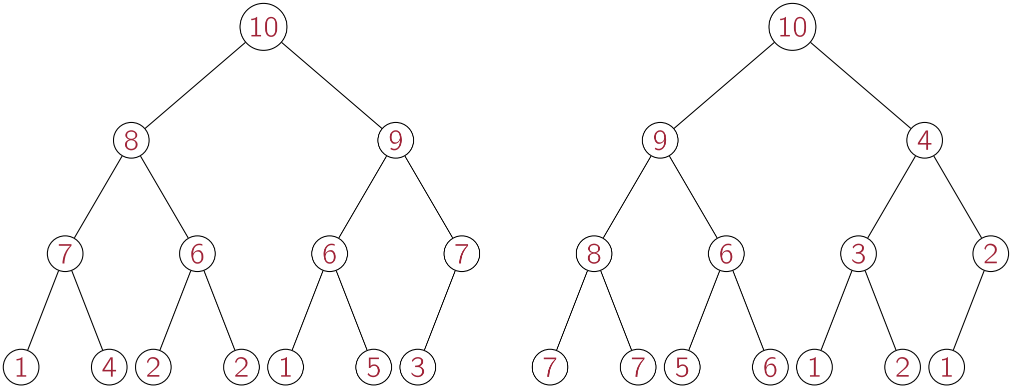
A *binary heap (or max-heap)* is

a complete binary tree satisfying the heap ordering property.

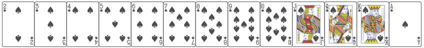
Questions on binary heaps



- What is the maximum key?
- What is the minimum key? How many steps (you can focus on comparisons) you need to find it?
- How about min-heaps (there are min-heaps, right)?

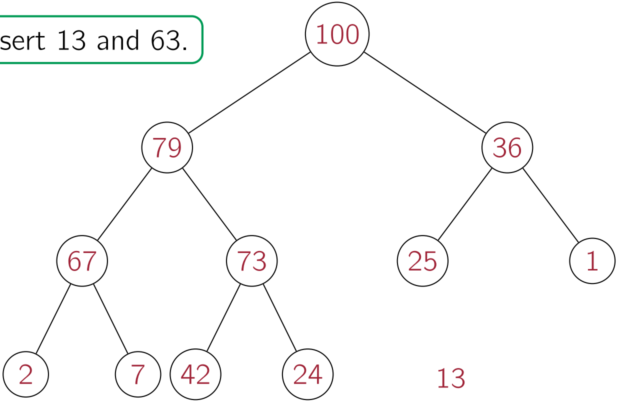


Two heaps on the same set of elements.

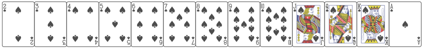


Insertion to a heap

We want to insert 13 and 63.

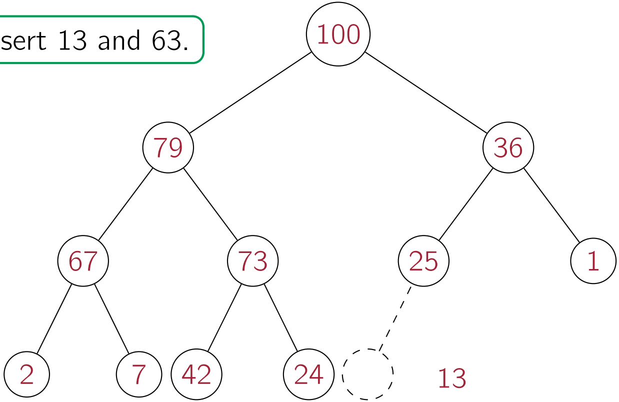


Where to put the new number 13?



Insertion to a heap

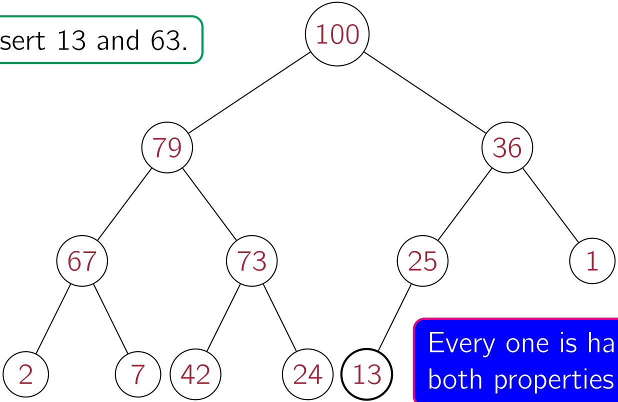
We want to insert 13 and 63.



Where to put the new number 13?

Insertion to a heap

We want to insert 13 and 63.

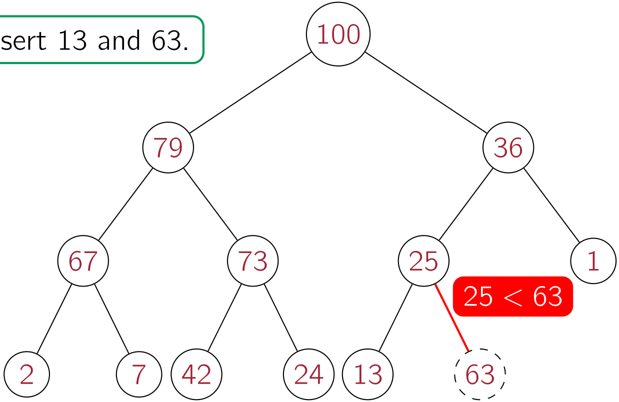


Every one is happy:
both properties are satisfied.

Where to put 63?

Insertion to a heap

We want to insert 13 and 63.

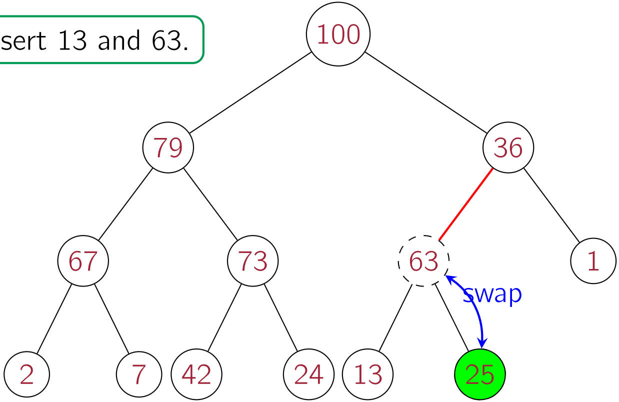


But somebody is unhappy; what we do?

We have two properties to satisfy.
Start from the simpler (trivial) one.

Insertion to a heap

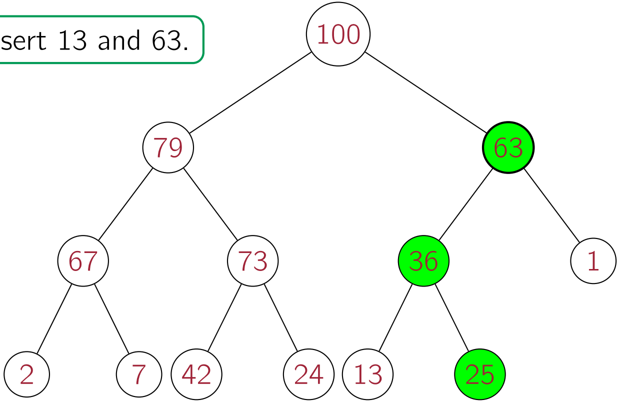
We want to insert 13 and 63.



Somebody else is unhappy, continue...

Insertion to a heap

We want to insert 13 and 63.

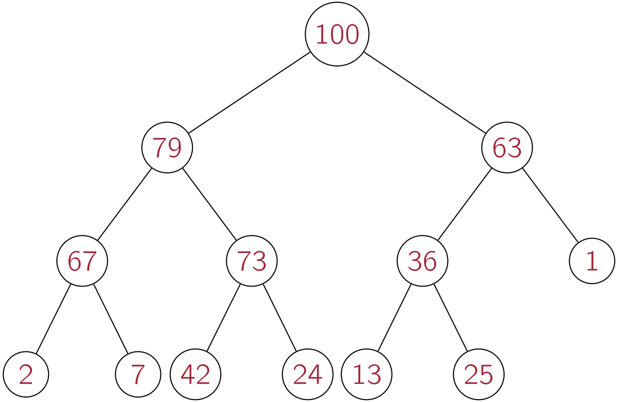


Do we need to check the other child?

Now everybody is happy and we re done.
All the swappings occur in one single path.

Exercise: insert 89.

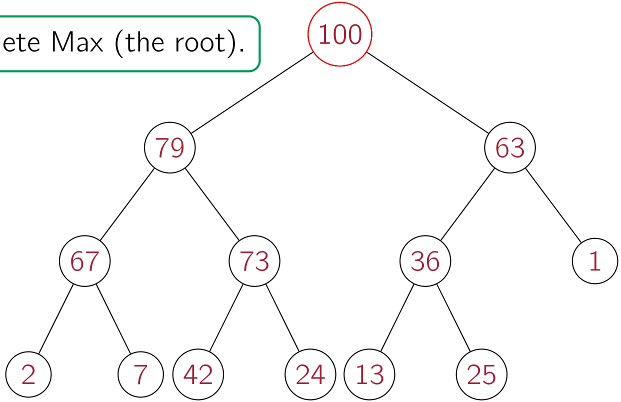
Questions on binary heaps



Write at least three orderings that lead to this heap

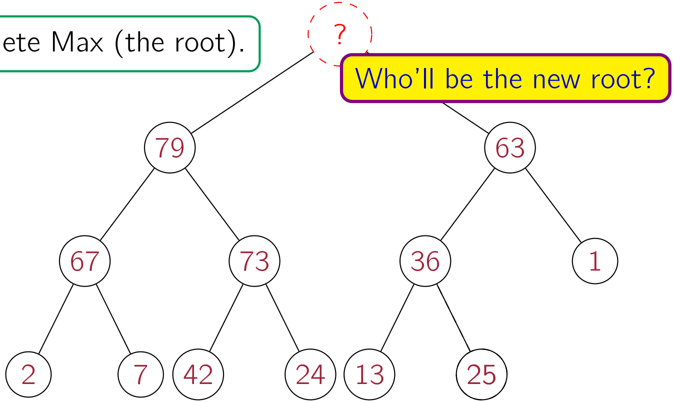
Deletion from a heap

Now let us delete Max (the root).



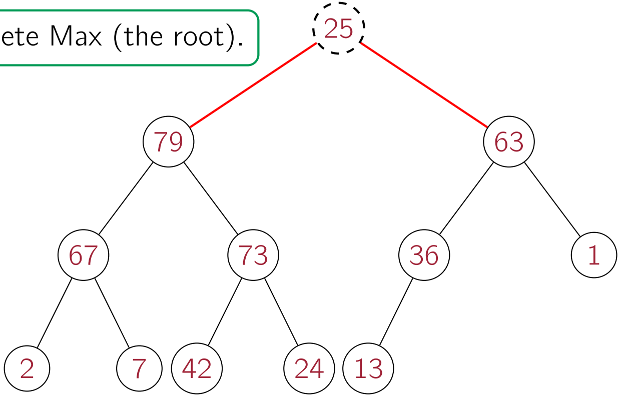
Deletion from a heap

Now let us delete Max (the root).



Deletion from a heap

Now let us delete Max (the root).

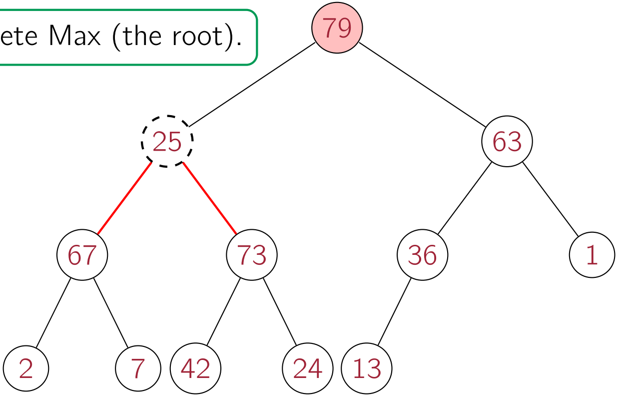


We have two properties to satisfy.
Start from the simpler (trivial) one.

But somebody are unhappy; what we do?

Deletion from a heap

Now let us delete Max (the root).

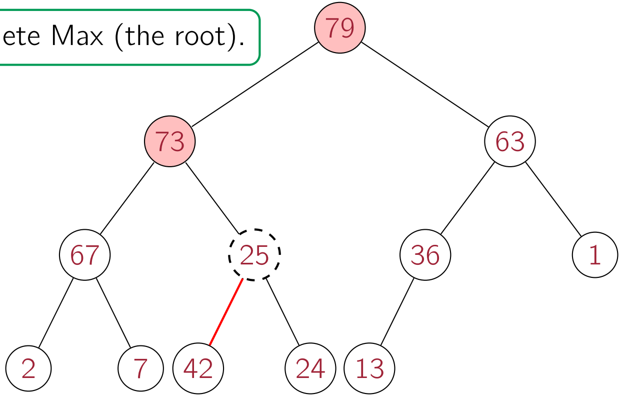


Let the unhappy ones fight.
The winner is the new boss.

Somebody else are unhappy, continue...

Deletion from a heap

Now let us delete Max (the root).

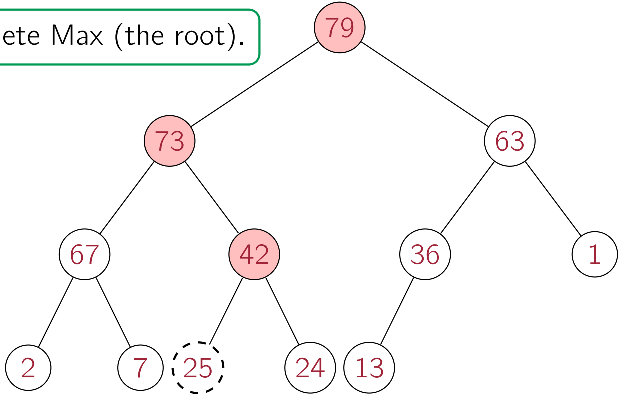


No fight needed this time.

Somebody else are unhappy, continue...

Deletion from a heap

Now let us delete Max (the root).

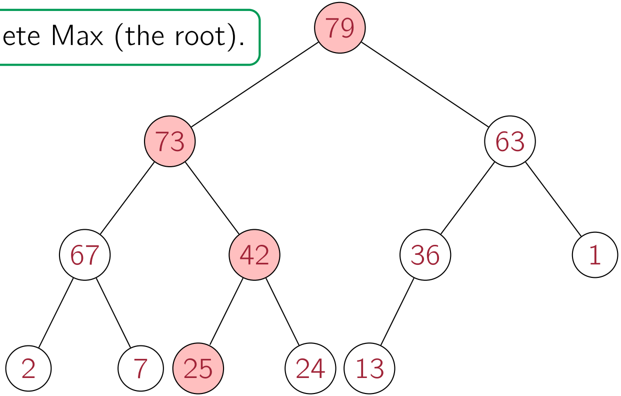


Again, all the swappings occur on one single path.

Exercise: delete the current Max (79)

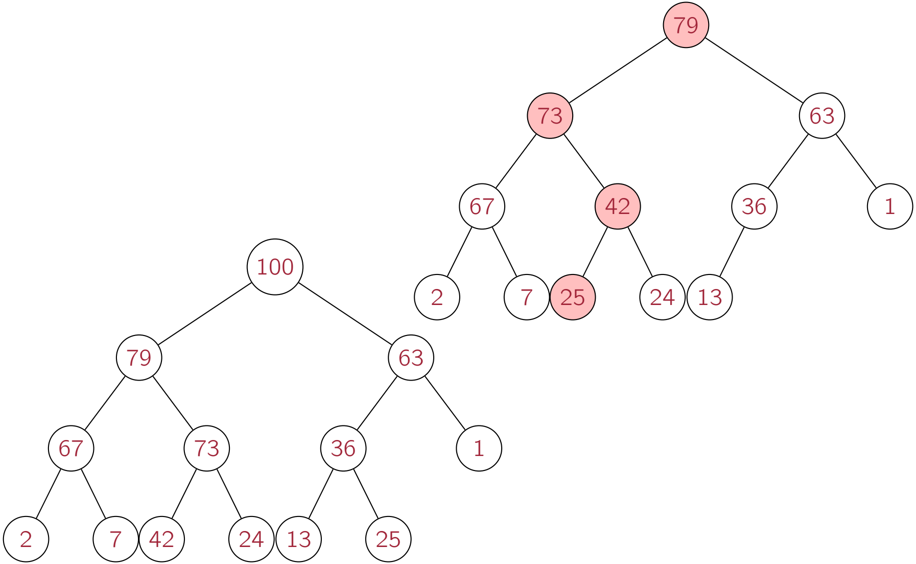
Deletion from a heap

Now let us delete Max (the root).



What if we then add back 100? Try it!

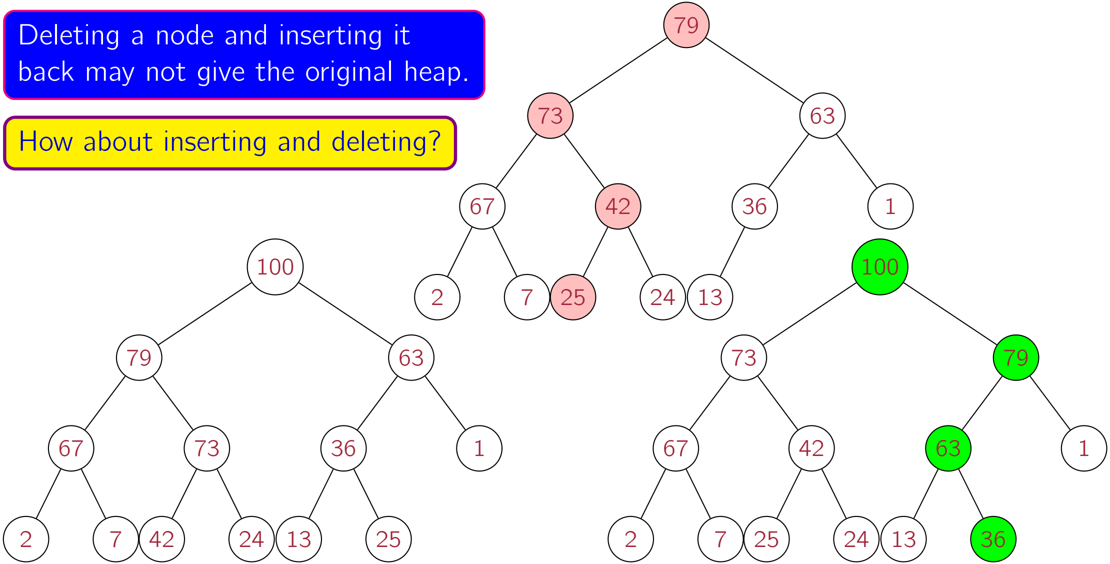
Modifying a heap



Modifying a heap

Deleting a node and inserting it back may not give the original heap.

How about inserting and deleting?



Insertion and deletion

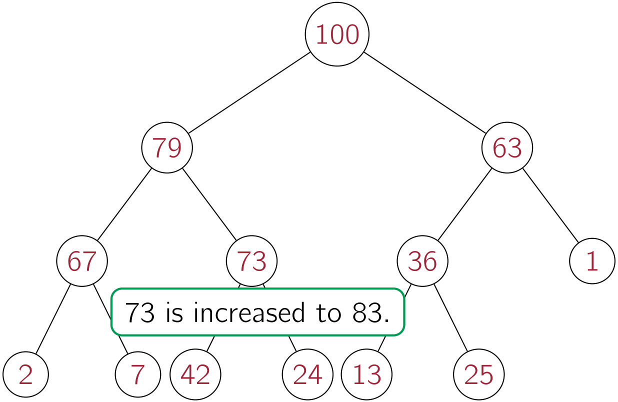
`insert(x)`

1. Make a new node with `data` x in the tree in the next available location.
2. "Bubble x up" the tree until finding a correct place:
   if the key of x is larger than its parent's key, then swap them and continue.

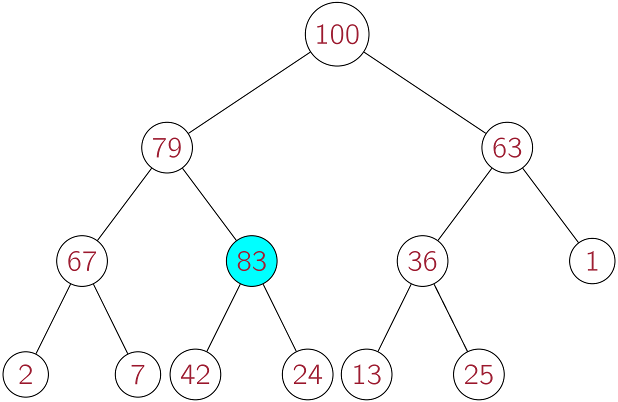
`removeMax()`

1. Remove the rightmost node y on the bottom level, and put it in the root.
2. "Bubble down" the new root's y until finding a correct place:
   if the key of y is smaller than at least one child's key, then swap y with largest child's key and continue.

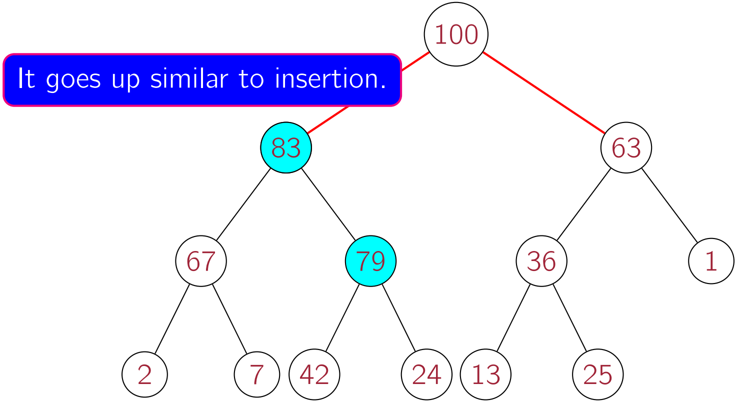
One item changed



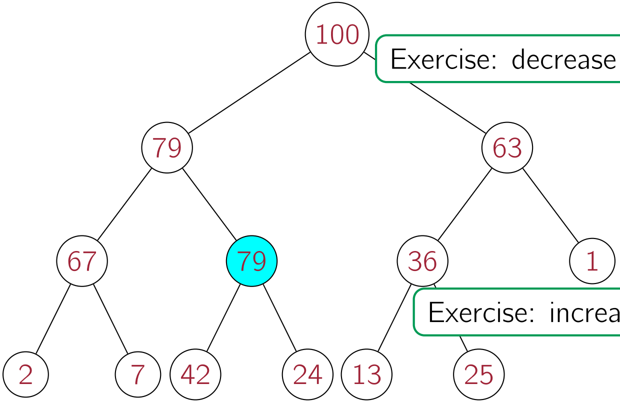
One item changed



One item changed



One item changed



Exercise: decrease 100 to 45.

Exercise: increase 36 to 79.

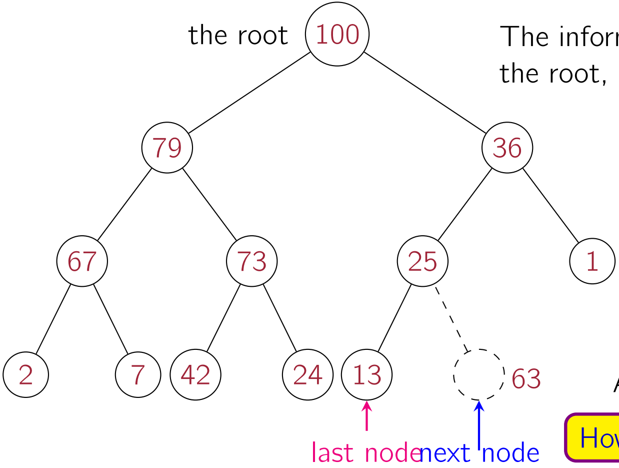
Summary

- A binary heap is a complete binary tree in which shape
- each node has a key no less than its children. order
- The largest item is always in the root, and it can be removed in $O(\log n)$ time.
- The insertion of a new element can also be done in $O(\log n)$ time.

- For the maintenance of a binary heap, we restore the *shape* before the *order*.
- New item is placed in the first vacant and then trickled up to its correct position.
- With the root removed, the last item takes its position and is then tricked down to is appropriate position.
- Both trickle-up/down processes can be thought of as a sequence of swaps, but are more efficiently implemented as a sequence of copies. (insertion sort)
- If an item is changed, then the node is trickled up/down depends on whether it was increased/decreased.

Implementation of Heapsort

A naïve approach: using a binary tree



The information we have:
the root, and the children of each node.

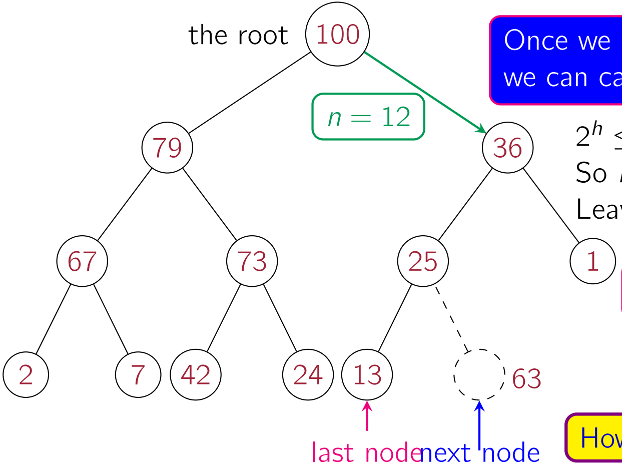
A level-wise traversal needs time $O(n)$.

How to find them?

For removeMax, we need to find the last node;
for insert, the next available location.

The order is not helping much,
so we should exploit the shape.

A naïve approach: using a binary tree



Once we know the number of nodes, we can calculate the positions.

$2^h \leq n < 2^{h+1}$ $(h = \lfloor \log n \rfloor)$

So $n - 2^h$ tells us which side to go.

Leaves at the bottom: $2^h, 2^h + 1, \ldots$

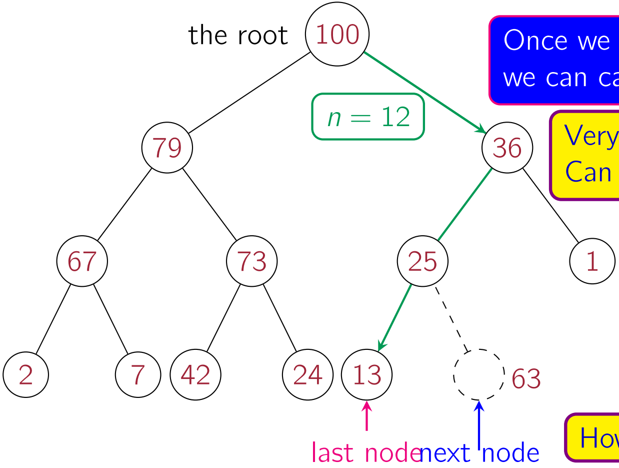
If $n - 2^h < 2^{h-1}$, go left, else right.

How to find them?

For removeMax, we need to find the last node; for insert, the next available location.

The order is not helping much, so we should exploit the shape.

A naïve approach: using a binary tree



Once we know the number of nodes, we can calculate the positions.

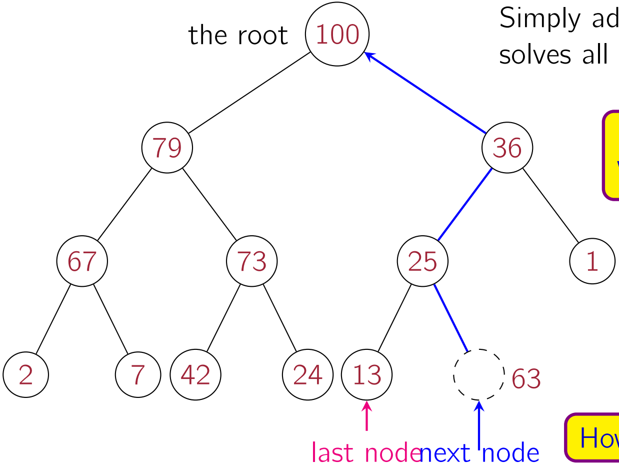
Very similar for next available location. Can you find the formula?

How to find them?

For removeMax, we need to find the last node; for insert, the next available location.

The order is not helping much, so we should exploit the shape.

A naïve approach: using a binary tree



Simply adding one variable `size` resolves all the problems. MAGICAL?

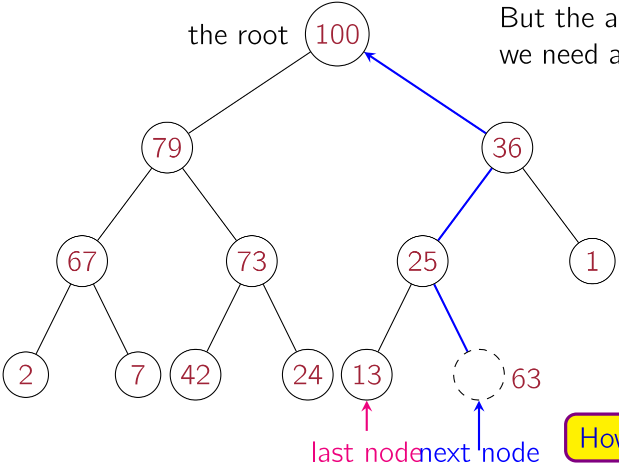
Is adding a `size` variable costly? What is the cost of maintaining it.

How to find them?

For removeMax, we need to find the last node; for insert, the next available location.

The order is not helping much, so we should exploit the shape.

A naïve approach: using a binary tree



But the algorithm is very complicated, and we need a stack to keep track of the path.

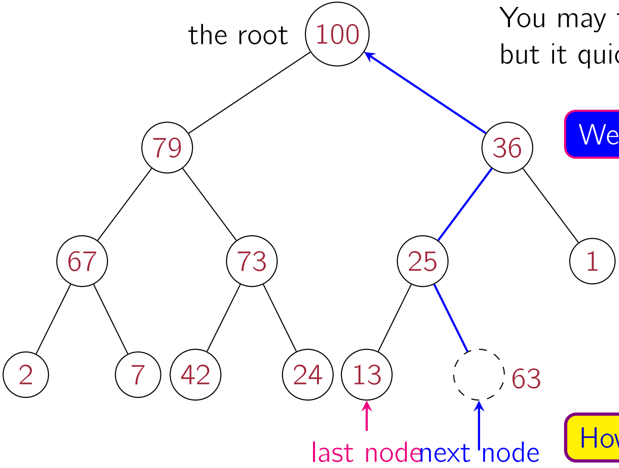
Is there a better idea?

How to find them?

For removeMax, we need to find the last node; for insert, the next available location.

The order is not helping much, so we should exploit the shape.

A naïve approach: using a binary tree



You may further augment the tree, but it quickly become messy.

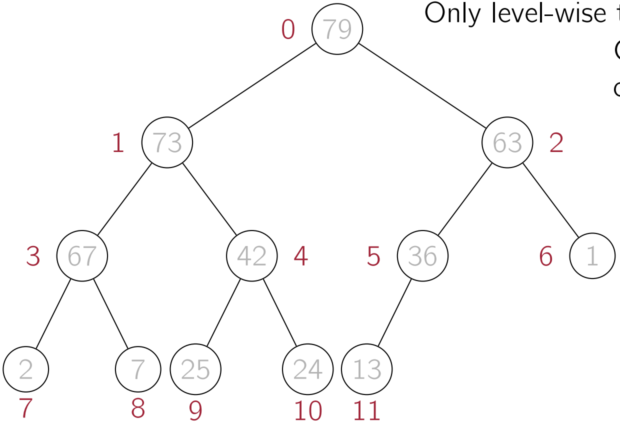
We need to reconsider.

How to find them?

For removeMax, we need to find the last node; for insert, the next available location.

The order is not helping much, so we should exploit the shape.

Array implementation of heaps



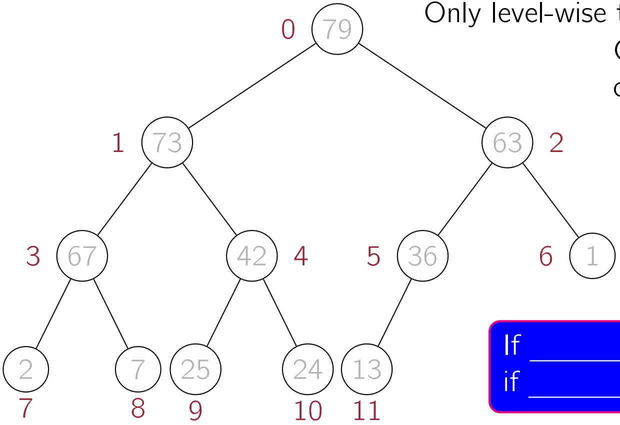
Only level-wise traversal is meaningful on a heap.

Children of 0 are 1 and 2;

children of 4 are 9 and 10;

Can you see a pattern?

Array implementation of heaps



Only level-wise traversal is meaningful on a heap.

Children of 0 are 1 and 2;

children of 4 are 9 and 10;

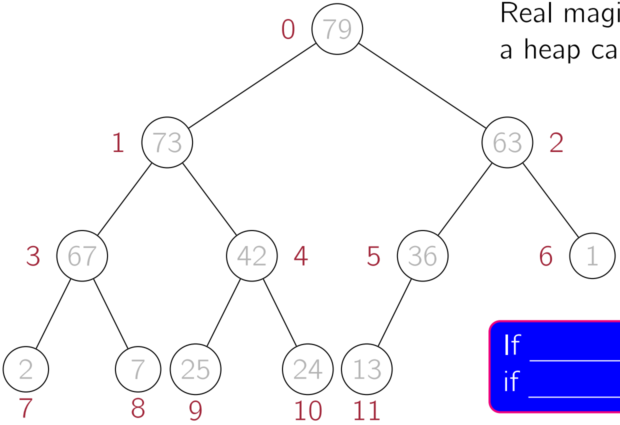
Can you see a pattern?

Left and right children of node i are ______ and ______ respectively.

If __________, then node i has no child;
if __________, then node i has only left child.

For $0 < k < n$, then parent of node k is $\lfloor \frac{k-1}{2} \rfloor$.

Array implementation of heaps



79	73	63	67	42	36	1	2	7	25	24	13
0	1	2	3	4	5	6	7	8	9	10	11

Real magical:
a heap can be stored without references.

Left and right children of node i are ______ and ______ respectively.

If __________, then node i has no child;
if __________, then node i has only left child.

For $0 < k < n$, then parent of node k is $\lfloor \frac{k-1}{2} \rfloor$.

The codes

```java
public class Heap<T> {
    private static class Node<T> {
        int key;
        T obj;
    }

    private Node<T>[] data;
    int size;
}
```

The insert method

```
void insert(int key, T x) {
    if (size == data.length) {err("overflow"); return;}
    data[size] = new Node<T>(key, x);
    up(size++);
}
```

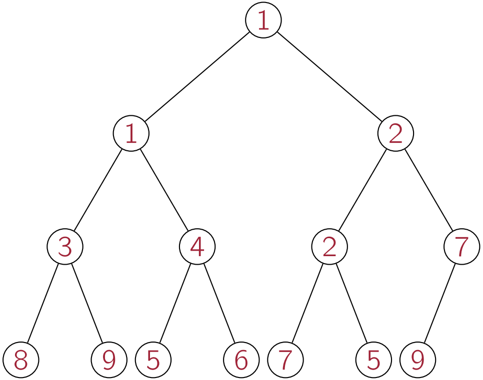
```
void up(int c) {
    if (c == 0) return; //root.
    int p = (c - 1) / 2;
    if (data[c].key <= data[p].key) return;
    swap(c, p);
    up(p);
}
```

The removeMax method

```
T removeMax() {
   if (size == 0) {err("downflow"); return null;}
   T ans = data[0].obj;
   data[0] = data[--size];
   down(0);
   return ans;
}
```

```
void down(int i) {
      if (size < 2 * i + 1) return;
      int lC = i * 2 + 1;
      int rC = lC + 1;
      int max = lC;
      if (rC<size && data[lC].key<data[rC].key)
         max = rC;
      if (data[i].key >= data[max].key) return;
      swap(i, max);
      down(max);
}
```

A minimum heap

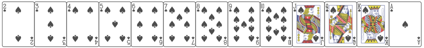


0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	1	2	3	4	2	7	8	9	5	6	7	5	9

Merging two heaps

$$[10, 5, 6, 2] + [12, 7, 9]$$

$$= [12, 10, 9, 2, 5, 7, 6]$$



Does this array represent a heap?

- [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
- [79, 73, 63, 67, 42, 36, 1, 2, 7, 25, 24, 13]
- [96, 95, 85, 85, 65, 17, 66, 71, 45, 38, 48, 18, 68, 60, 55]

Write an algorithm to decide whether an array represents a heap?

Lecture 10: Heapsort