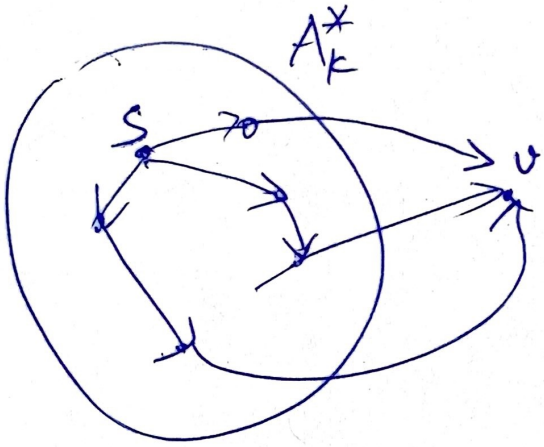
PQ stores the vertices in $V - A_k^*$

$A_k^* = \{v_1^*, v_2^*, \cdots, v_k^*\}$

dist(v) stores the length of the shortest path from s to v using vertices in A_k^*.



Fact: at the time v^* is removed from PQ,

$dist(v^*) = distance(s, v^*)$.

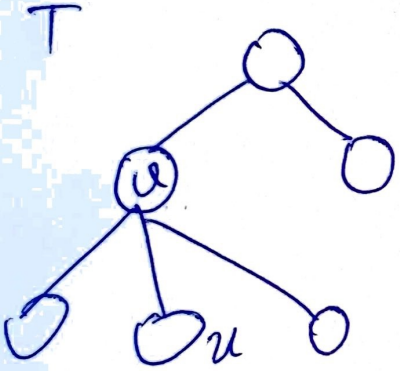
Fact: the order of vertices that are removed from PQ is exactly $(v_1^*, v_2^*, \cdots, v_n^*)$.

Running time:

operation	# times	Binary-Heap
insert	$\|V\|$	$O(\log n)$
decrease-key	$1+\|E\|$	$O(\log n)$
find-min	$\|V\|$	$O(1)$
delete-min	$\|V\|$	$O(\log n)$
Binary-Heap		$O(\|E\| \cdot \log\|V\| + \|V\| \cdot \log\|V\|)$.

Heap: tree data structure satisfying the **heap property**.

T

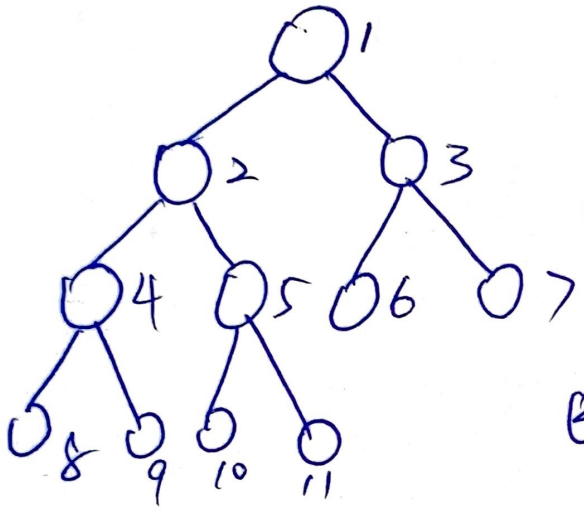


Heap property: for $\forall v \in T$, for ~~a~~ every child u of v.
key(v) ~~≥ key(u).~~ is better than key(u).

min-heap: if key(v) < key(u).

max-heap: if key(v) > key(u).

Binary Heap:



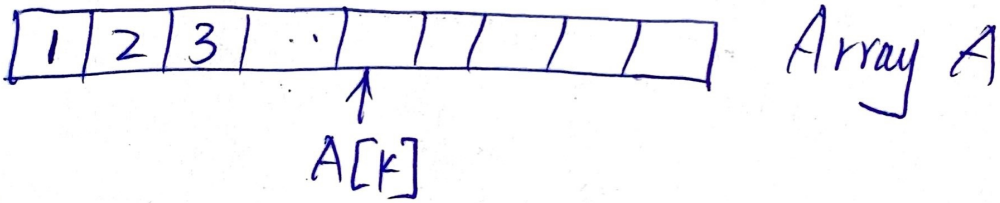
① heap

② each node has at most 2 children

③ one layer has to be full before starting the next layer

④ In the last layer, vertices are packed from left to right.

Use array to store the nodes.



V_K

parent (V_K) = $V[\lfloor K/2 \rfloor]$

left (V_K) = V_{2K}

right (V_K) = V_{2K+1}

A

procedure bubble-up (A, i)

  j = ⌊i/2⌋

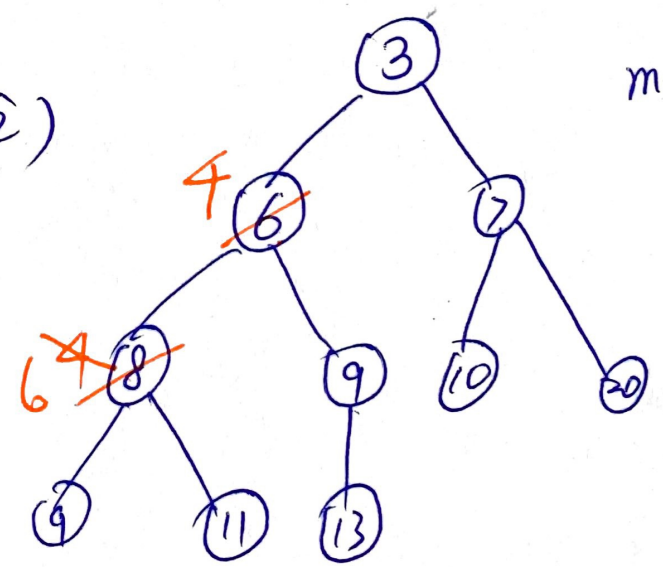
  if (key(V_i) < key(V_j))

    swap(A[i], A[j])

    bubble-up(A, j)

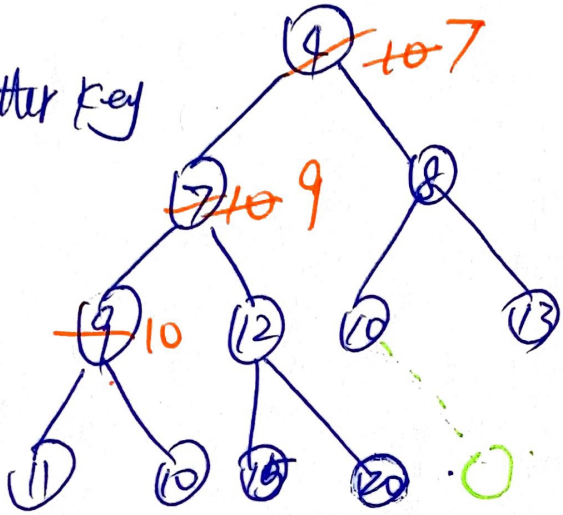
  endif

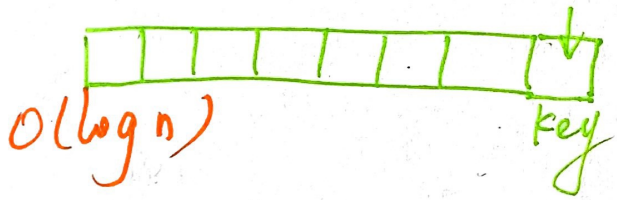
end



min-heap

```
Procedure sift-down (A, i)
    let j be the child of Vi with better key
    if (key(Vj) < key(Vi))
        swap(A[i], A[j])
        sift-down (A, j)
    end if
end procedure.
```






```
-insert (PQ, key)
    let n be the size of A.
    n++;
    A[n].key = key.
    bubble-up (A, n)
```

decrease-key (PQ, i, new-key) O(log n)

```
    A[i].key = new-key
    bubble-up (A, i).
```

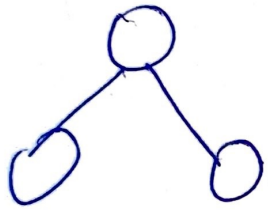
find-min return A[1] O(1)

delete-min (PQ) O(log n).

```
    A[1] ← A[n]
    n--;
    sift-down (A, 1).
```

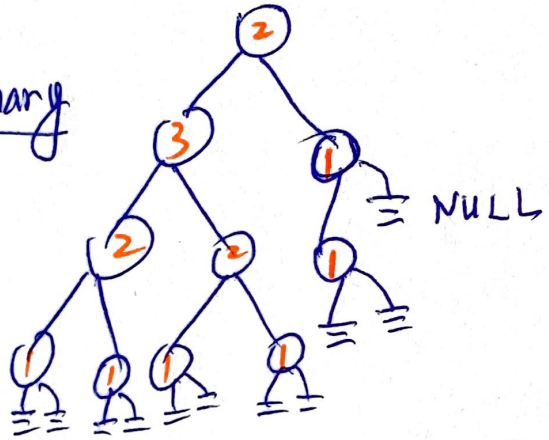
Fact: merge two binary heaps takes $O(n)$ time.

Leftist tree



① tree binary

② heap



③ rank(v) = null-path-length(v).

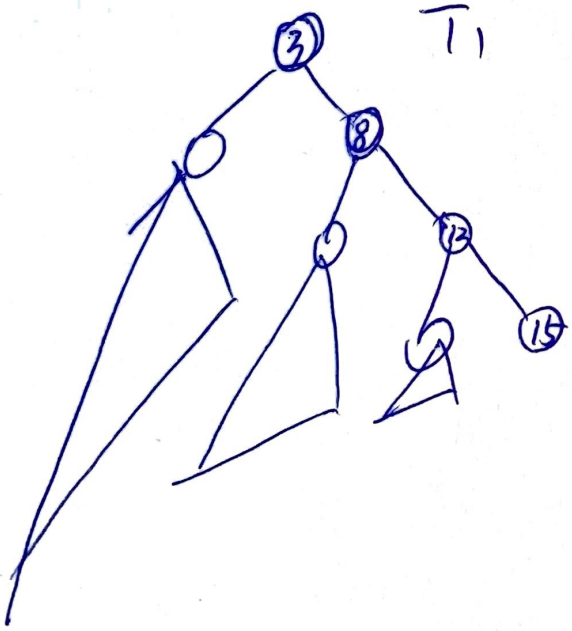
rank(v) = #nodes on the shortest path from v to a NULL.

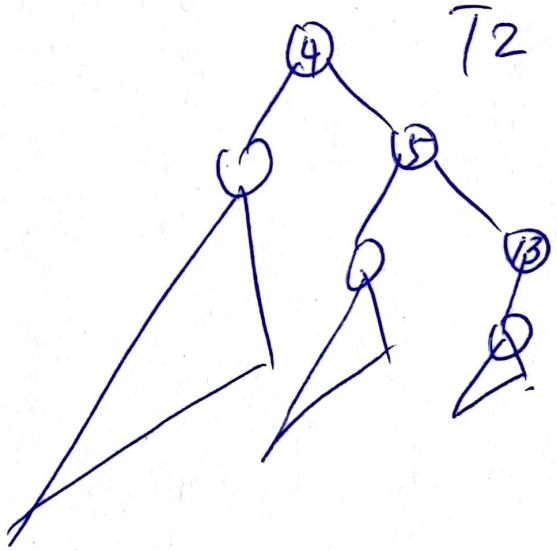
leftist property: for $\forall v \in T$, ~~for any child x of v~~.

$$rank(L(v)) \geq rank(R(v))$$ L(v): left child of v.

Fact: $rank(root) = O(\log n)$.

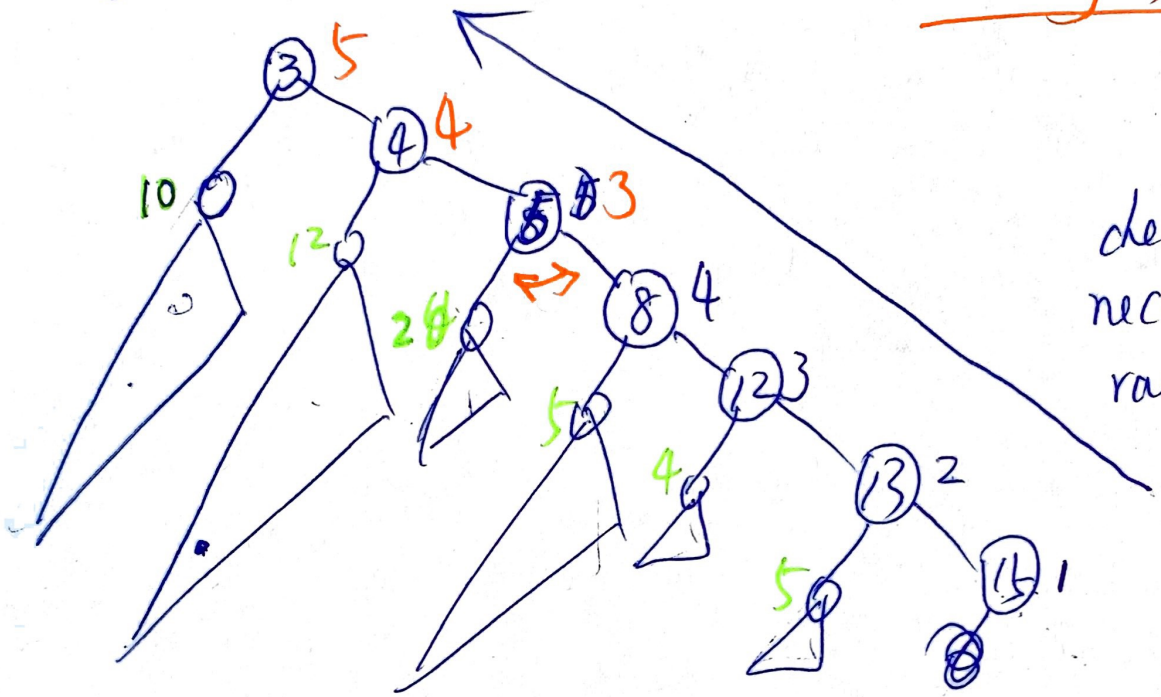
Merging two leftist trees





merge (T_1, T_2).

O(log n)



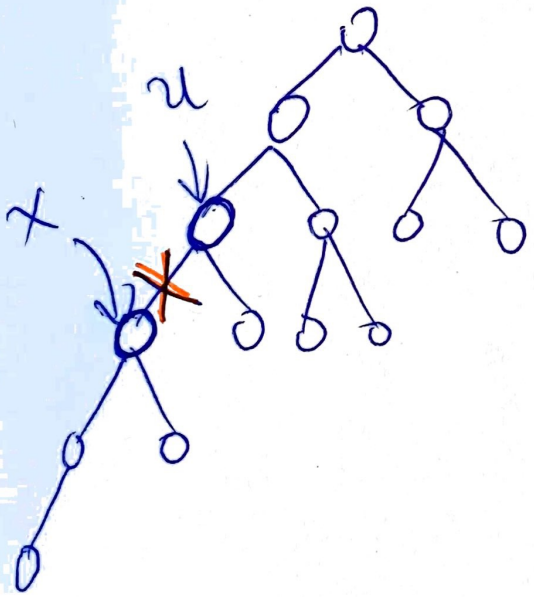
find-min(PQ) return root.

delete-min(PQ) merge(TL(root), TR(root))

TL(root): left subtree of root

insert(PQ, key) merge(PQ, new-tree-with-the-new-node).

decrease-key(PQ, x, new-key)



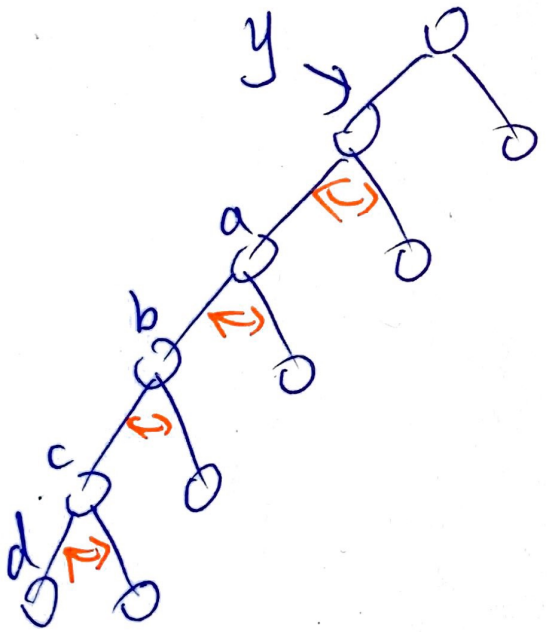
① cut off subtree rooted at x Tx

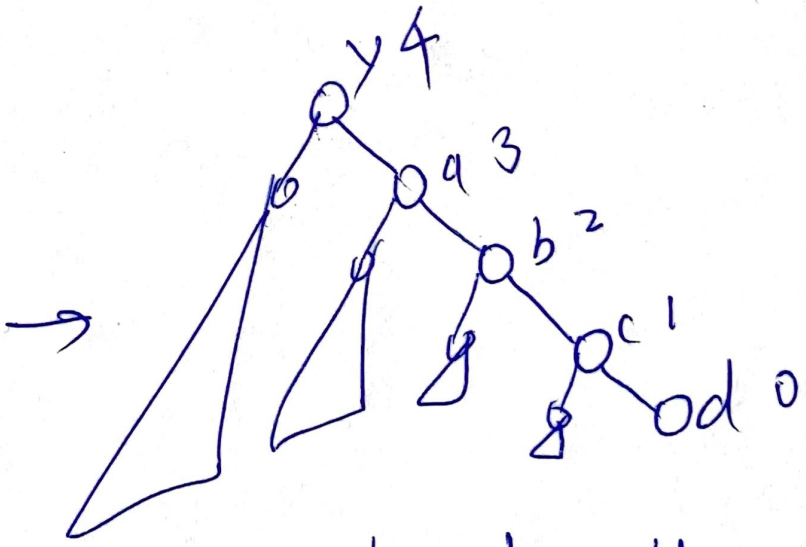
~~② check the vertices on the path from parent(x) to root~~

② adjust the remaining of T so that it satisfy the leftist property.

③ merge Tx and the adjusted T.

```
v = x
while(true)
    let u = parent(v)
    ~~if (rank(v) < rank(u.right child))~~
    if ( rank(u.left) < rank(u.right) )
        ~~swap two children~~
        rank(u) = rank(u.left) + 1
        swap two children of u
        v = u
    ~~end if~~
    else
        break;
    end if
~~end wh~~ end while
```





$rank(y) = \log |T_y| = O(\log n)$