

Chapter 19

Hash Tables


```
hash( 89, 10 ) = 9
hash( 18, 10 ) = 8
hash( 49, 10 ) = 9
hash( 58, 10 ) = 8
hash(  9, 10 ) = 9
```

	After Insert 89	*After Insert 18*	*After Insert 49*	*After Insert 58*	*After Insert 9*
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Linear probing hash table after each insertion

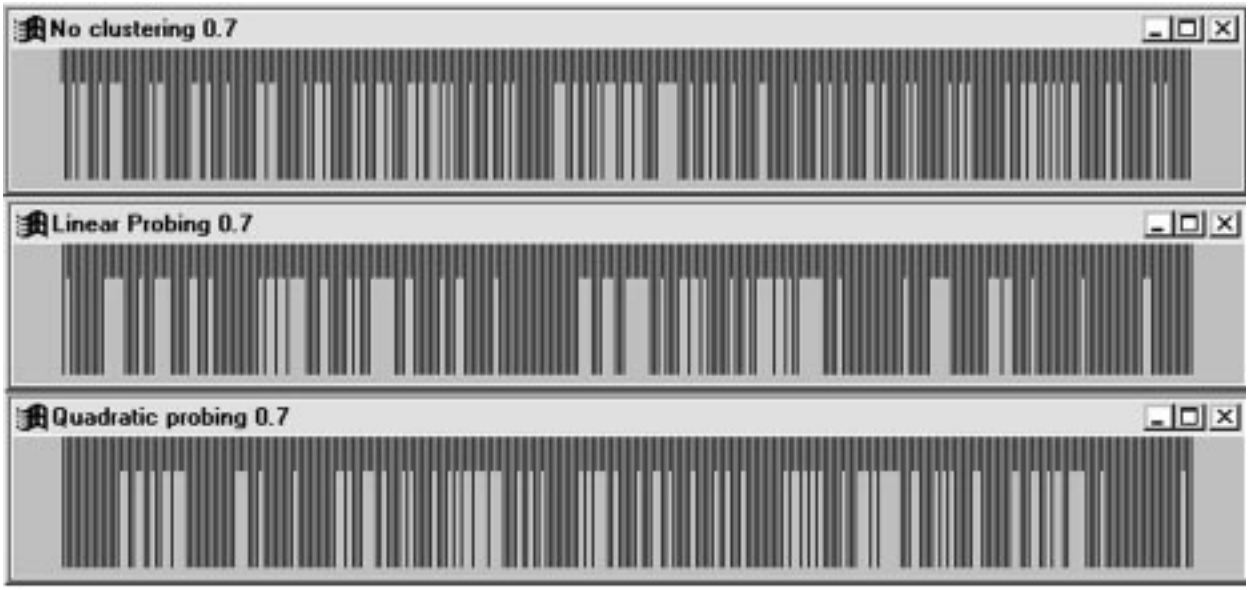


Illustration of primary clustering in linear probing (middle) versus no clustering (top) and the less significant secondary clustering in quadratic probing (bottom); long lines represent occupied cells; Load factor is 0.7

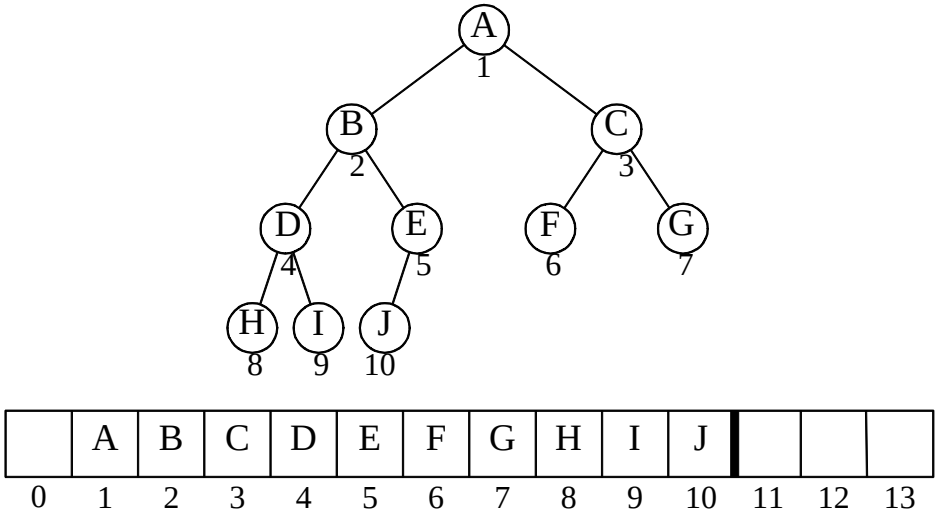

```
hash( 89, 10 ) = 9
hash( 18, 10 ) = 8
hash( 49, 10 ) = 9
hash( 58, 10 ) = 8
hash(  9, 10 ) = 9
```

	After Insert 89	*After Insert 18*	*After Insert 49*	*After Insert 58*	*After Insert 9*
0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

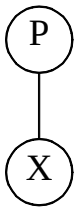
Quadratic probing hash table after each insertion (note that the table size is poorly chosen because it is not a prime number)

Chapter 20

A Priority Queue: The Binary Heap

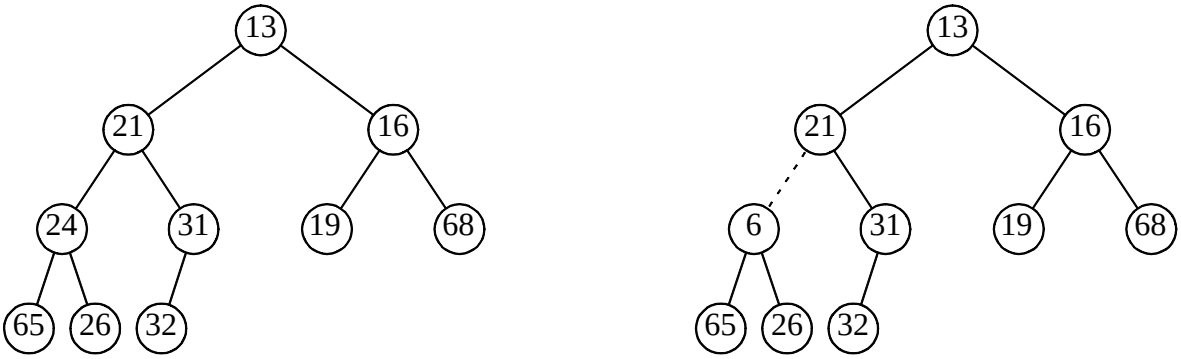


A complete binary tree and its array representation

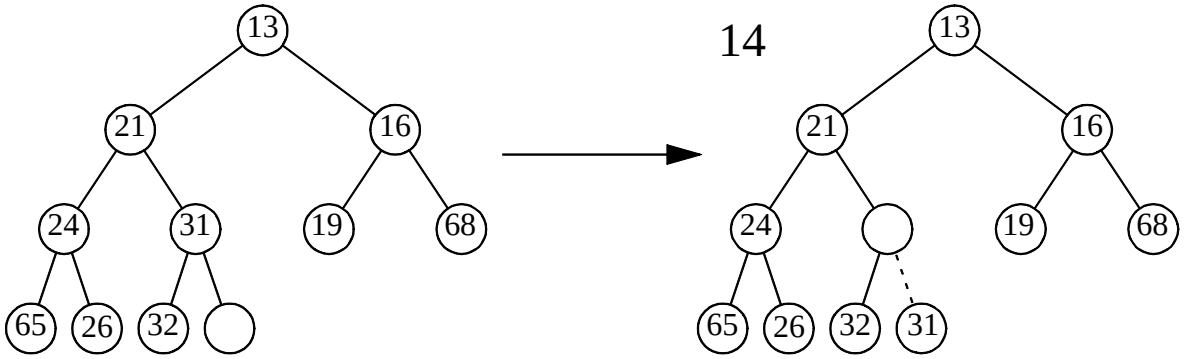


$P \leq X$

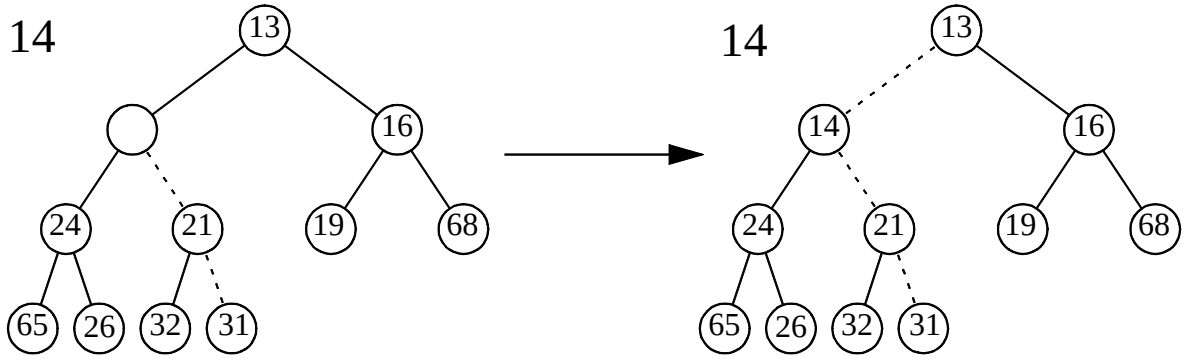
Heap-order property



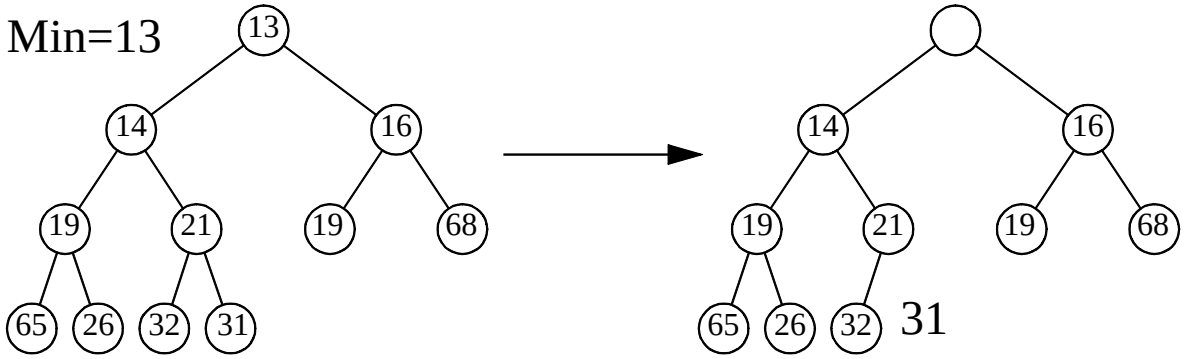
Two complete trees (only the left tree is a heap)



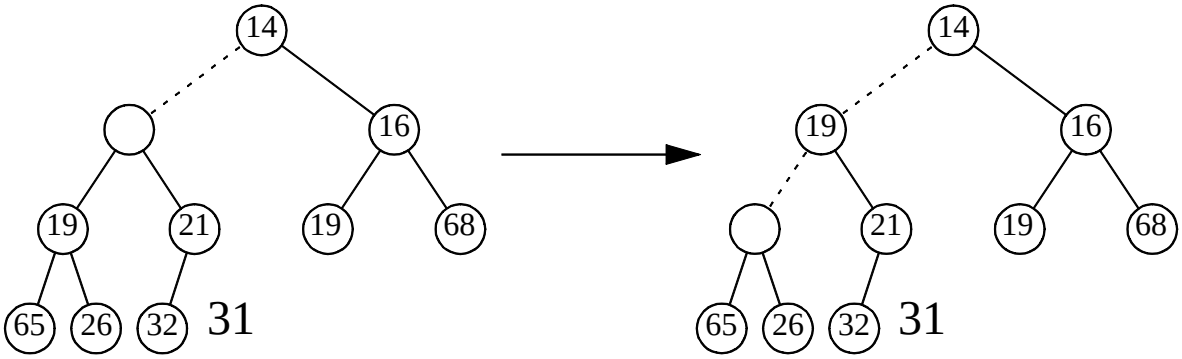
Attempt to insert 14, creating the hole and bubbling the hole up



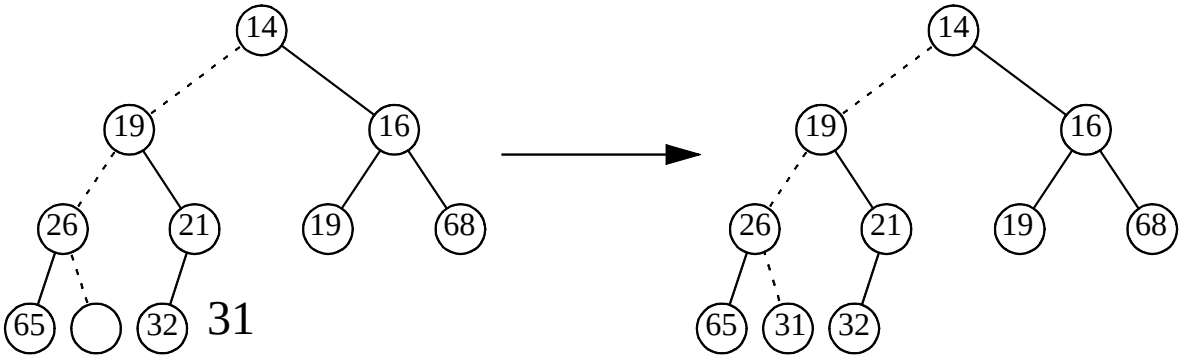
The remaining two steps to insert 14 in previous heap



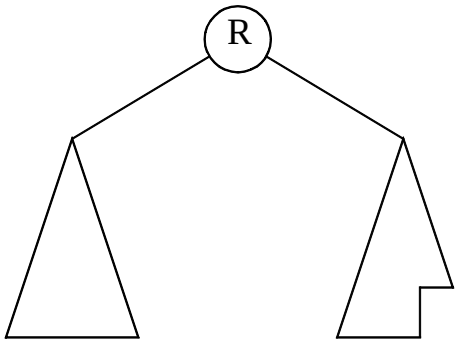
Creation of the hole at the root



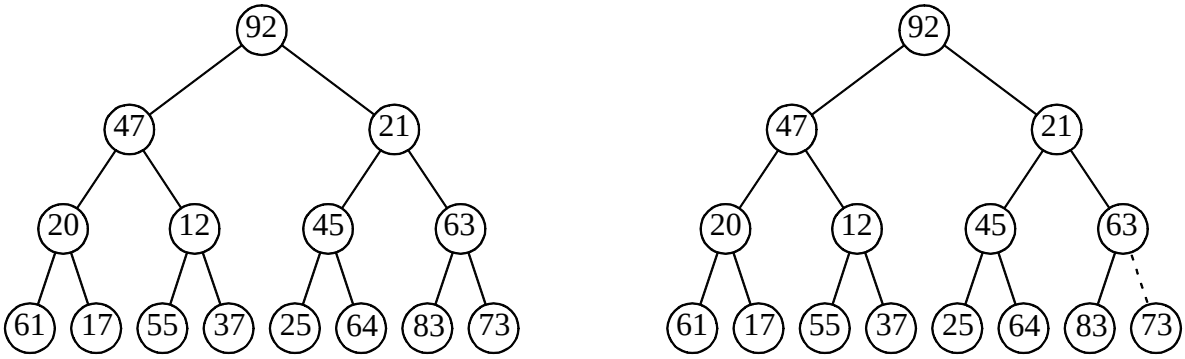
Next two steps in `deleteMin`



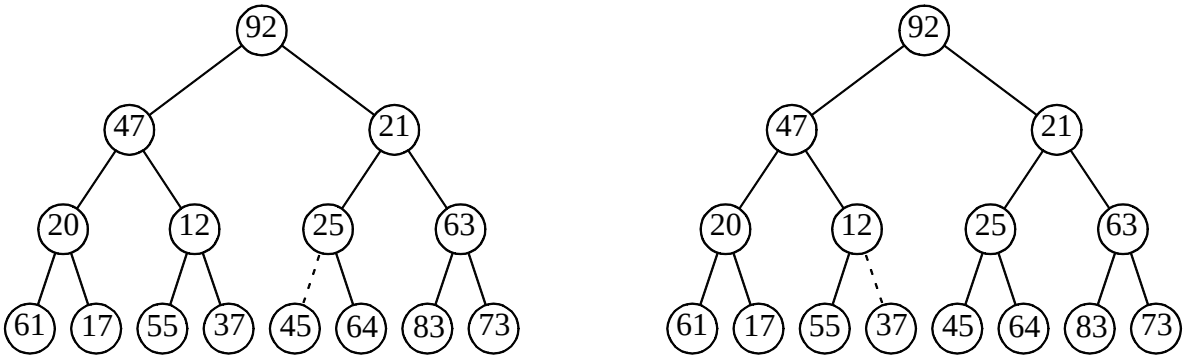
Last two steps in `deleteMin`



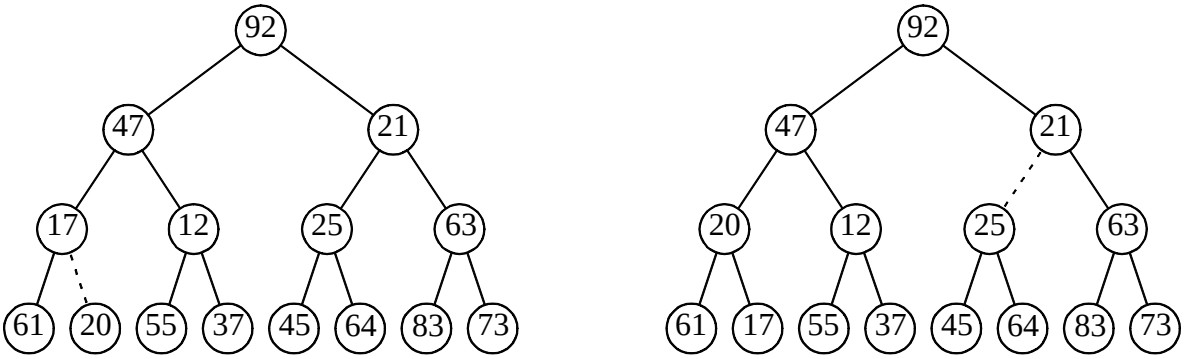
Recursive view of the heap



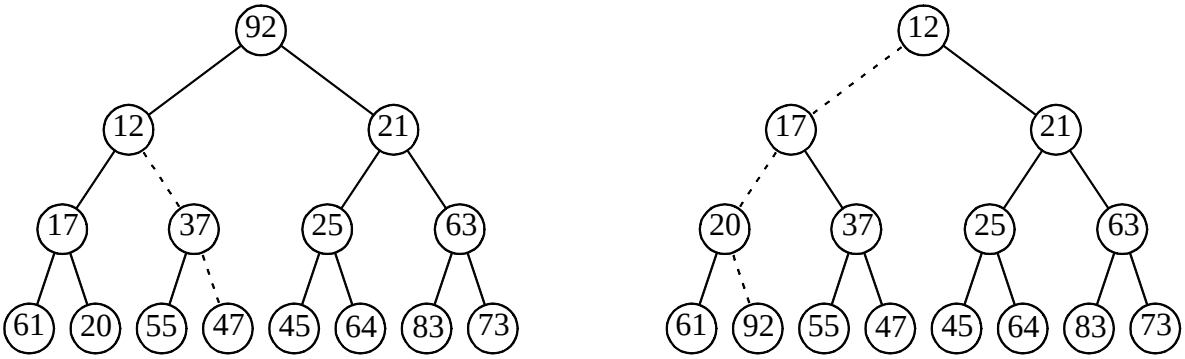
Initial heap (left); after `percolateDown(7)` (right)



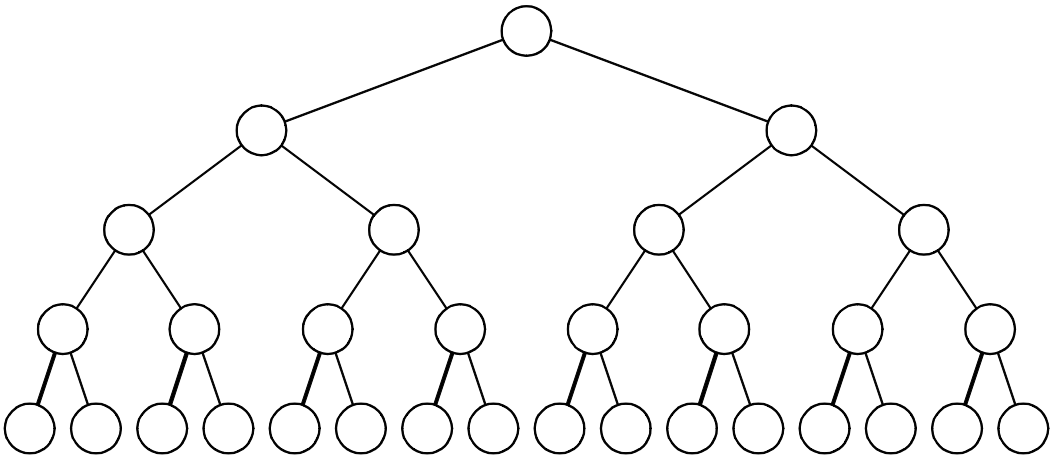
After `percolateDown(6)` (left); after `percolateDown(5)` (right)



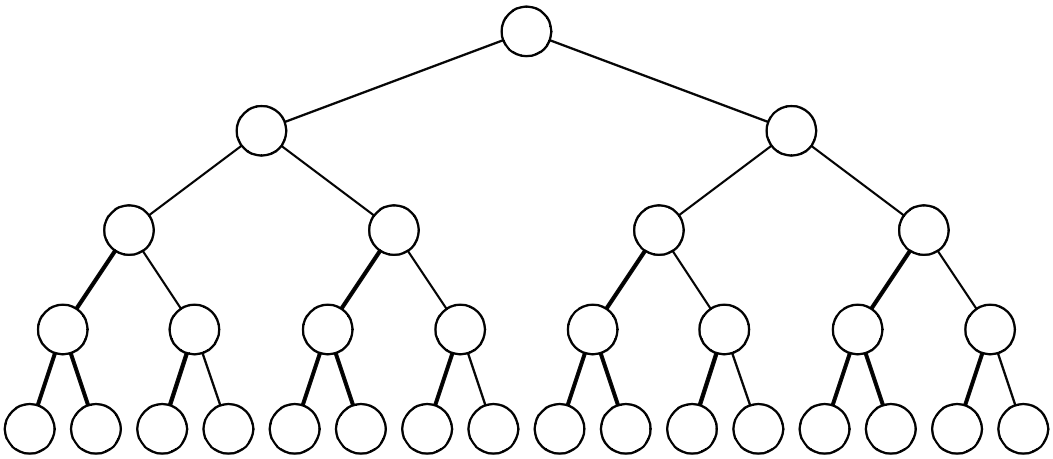
After `percolateDown(4)` (left); after `percolateDown(3)` (right)



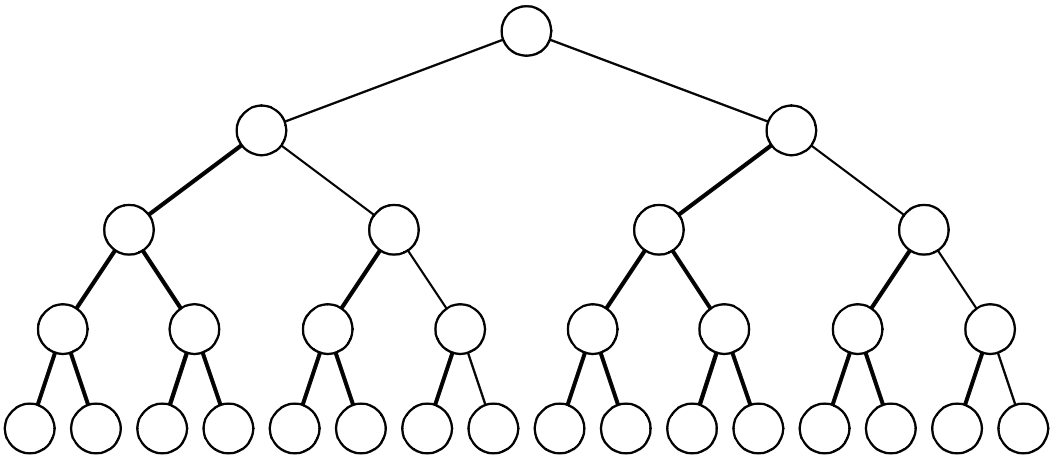
After `percolateDown(2)` (left); after `percolateDown(1)` and `fixHeap` terminates (right)



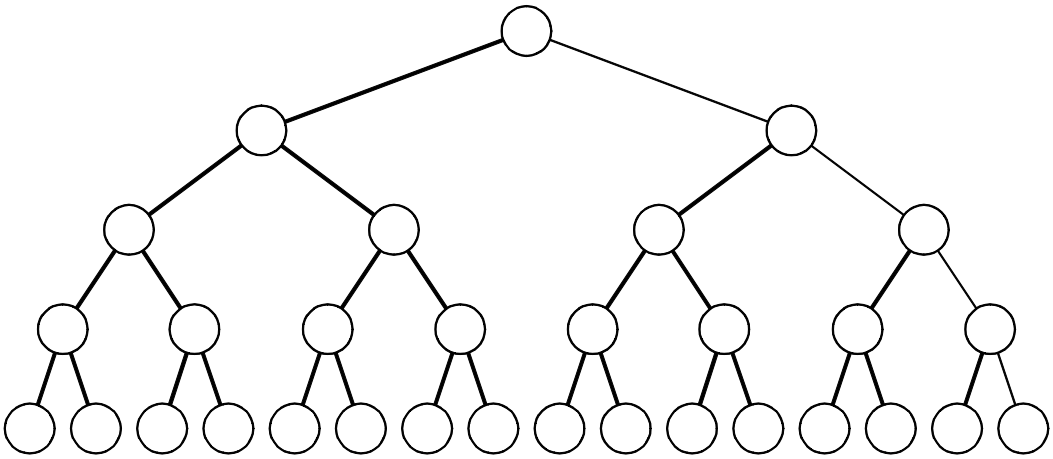
Marking of left edges for height-one nodes



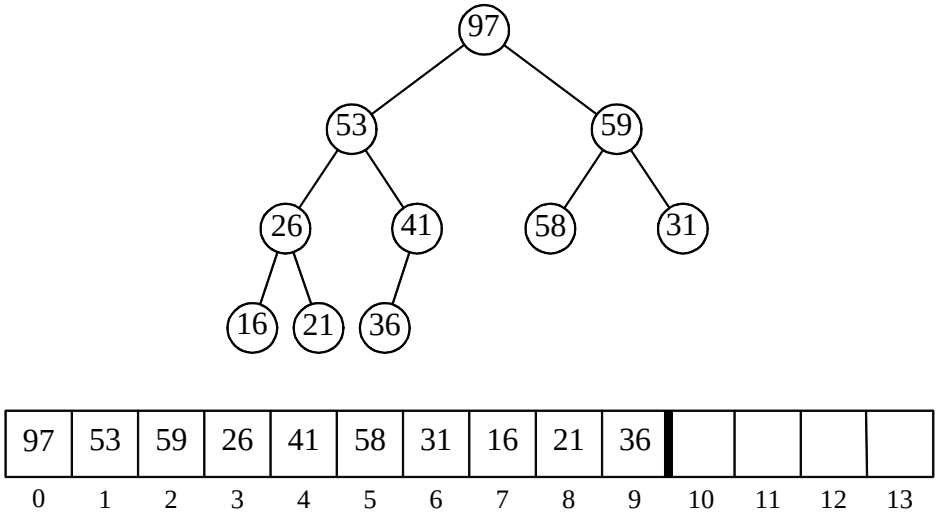
Marking of first left and subsequent right edge for height-two nodes



Marking of first left and subsequent two right edges for height-three nodes



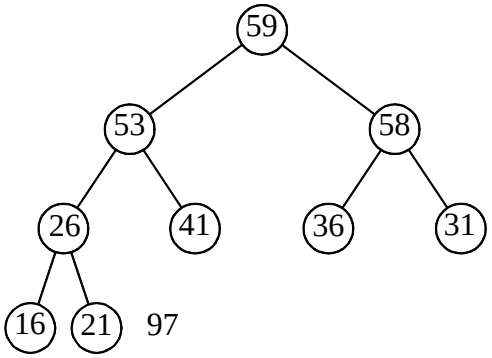
Marking of first left and subsequent right edges for height-four node

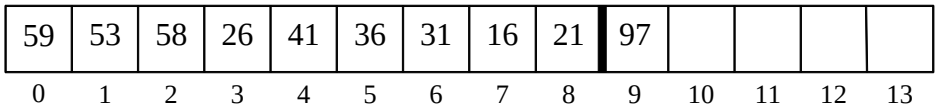


(Max) Heap after `fixHeap` phase

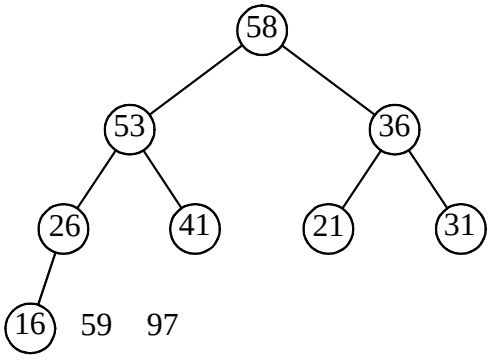
1. `toss` each item into a binary heap.
2. Apply `fixHeap`.
3. Call `deleteMin` *N* times; the items will exit the heap in sorted order.

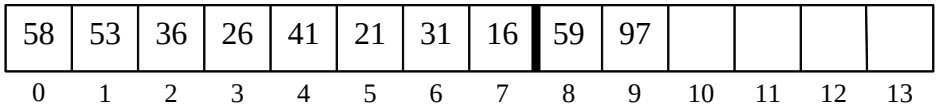
Heapsort algorithm (in principle)





Heap after first `deleteMax`





Heap after second `deleteMax`

A1	81	94	11	96	12	35	17	99	28	58	41	75	15
A2													
B1													
B2													

Initial tape configuration

A1							
A2							
B1	11	81	94	17	28	99	15
B2	12	35	96	41	58	75	

Distribution of length 3 runs onto two tapes

A1	11	12	35	81	94	96	15					
A2	17	28	41	58	75	99						
B1												
B2												

Tapes after first round of merging (run length = 6)

A1												
A2												
B1	11	12	17	28	35	41	58	75	81	94	96	99
B2	15											

Tapes after second round of merging (run length = 12)

A1	11	12	15	17	28	35	41	58	75	81	94	96	99
A2													
B1													
B2													

Tapes after third round of merging

A1												
A2												
A3												
B1	11	81	94	41	58	75						
B2	12	35	96	15								
B3	17	28	99									

Initial distribution of length 3 runs onto three tapes

A1	11	12	17	28	35	81	94	96	99			
A2	15	41	58	75								
A3												
B1												
B2												
B3												

After one round of three-way merging (run length = 9)

A1												
A2												
A3												
B1	11	12	15	17	28	35	41	58	75	81	94	96	99
B2												
B3												

After two rounds of three-way merging

	Run	After						
	Const.	T3+T2	T1+T2	T1+T3	T2+T3	T1+T2	T1+T3	T2+T3
T1	0	13	5	0	3	1	0	1
T2	21	8	0	5	2	0	1	0
T3	13	0	8	3	0	2	1	0

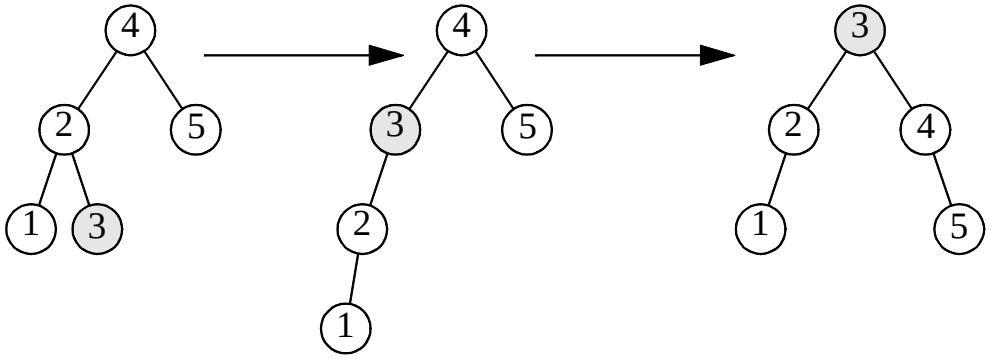
Number of runs using polyphase merge

	3 Elements in Heap Array			Output	Next Item Read
	array[1]	array[2]	array[3]		
Run 1	11	94	81	11	96
	81	94	96	81	12
	94	96	12	94	35
	96	35	12	96	17
	17	35	12	End of Run	Rebuild Heap
Run 2	12	35	17	12	99
	17	35	99	17	28
	28	99	35	28	58
	35	99	58	35	41
	41	99	58	41	75
	58	99	75	58	15
	75	99	15	75	End of Tape
	99		15	99	
			15	End of Run	Rebuild Heap
Run 3	15			15	

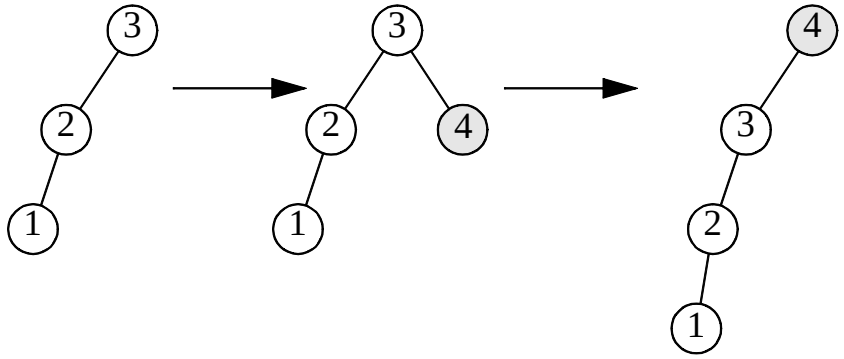
Example of run construction

Chapter 21

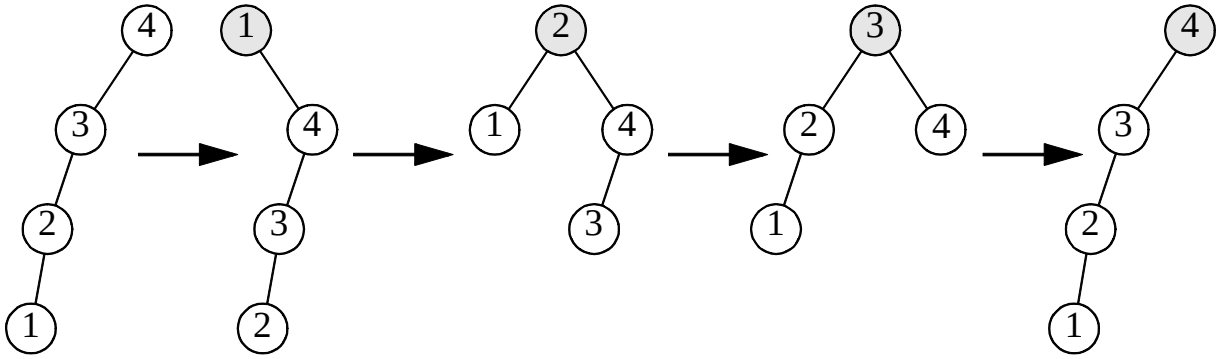
Splay Trees



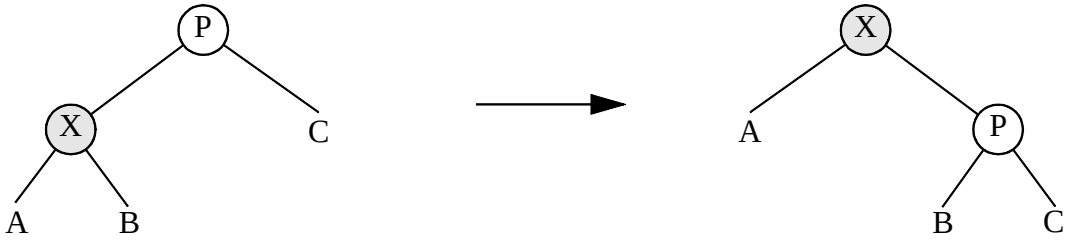
Rotate-to-root strategy applied when node 3 is accessed



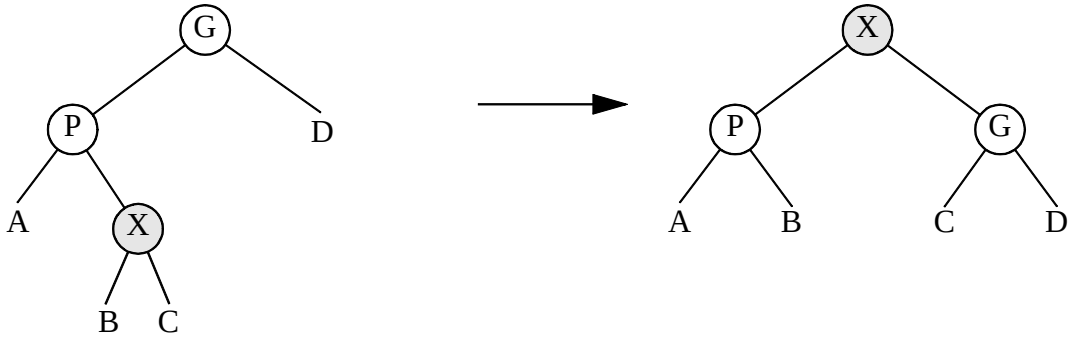
Insertion of 4 using rotate-to-root



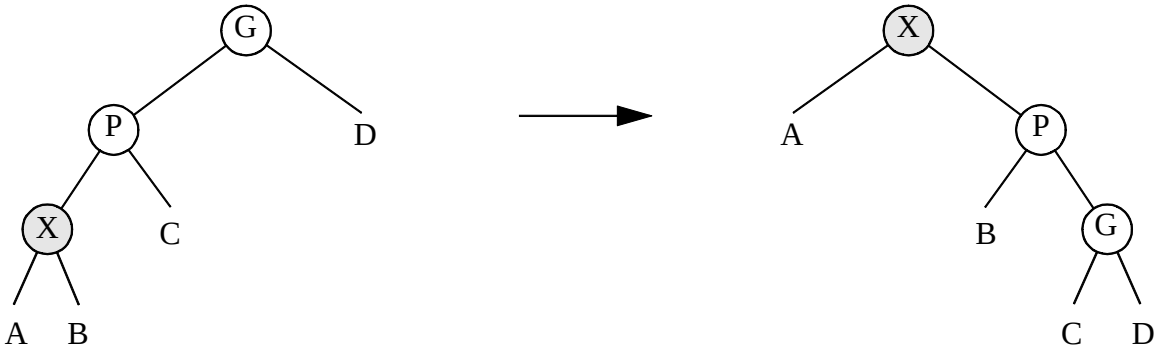
Sequential access of items takes quadratic time



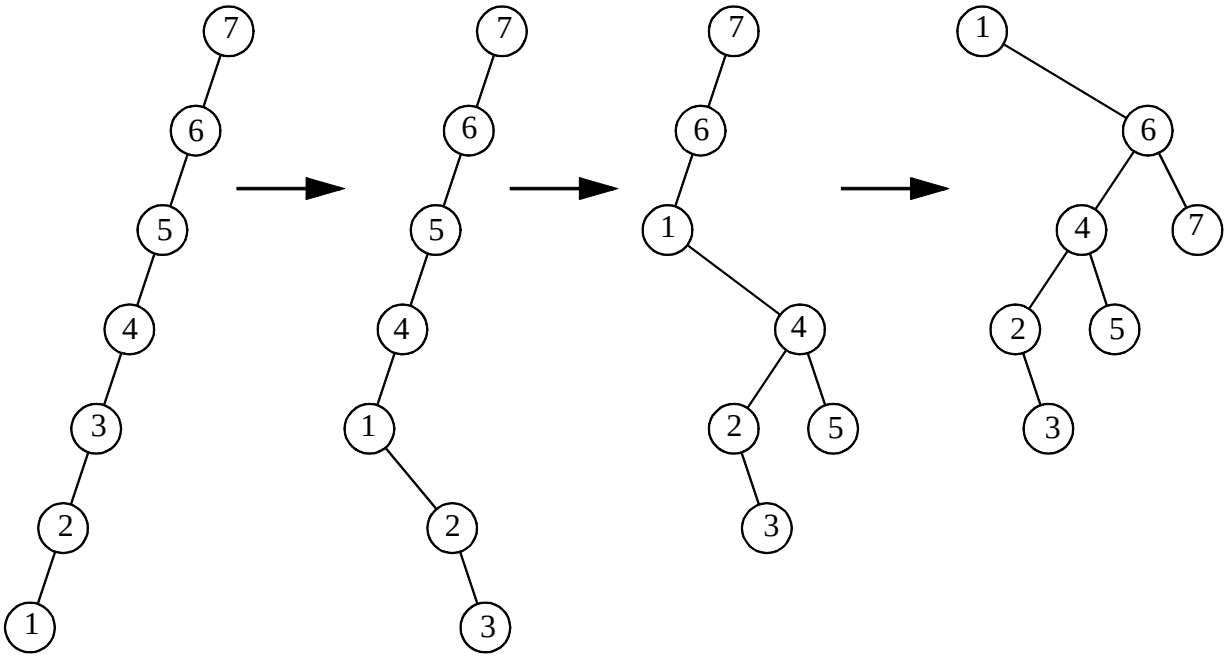
Zig case (normal single rotation)



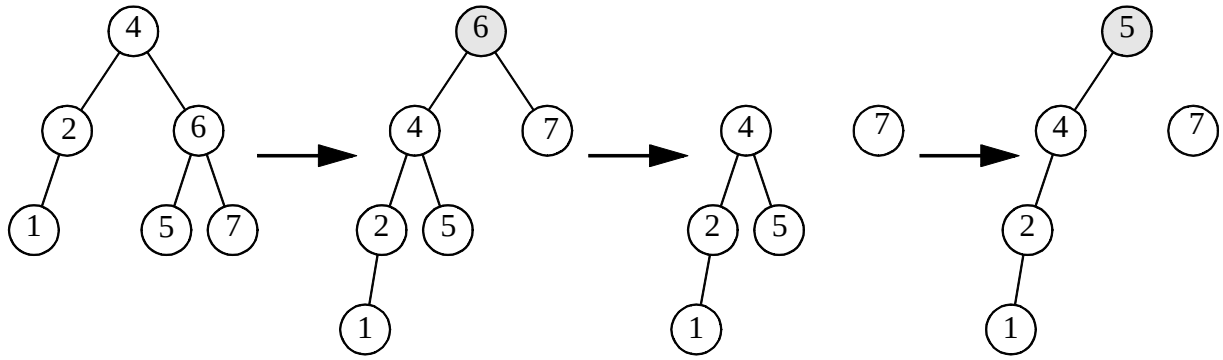
Zig-zag case (same as a double rotation); symmetric case omitted



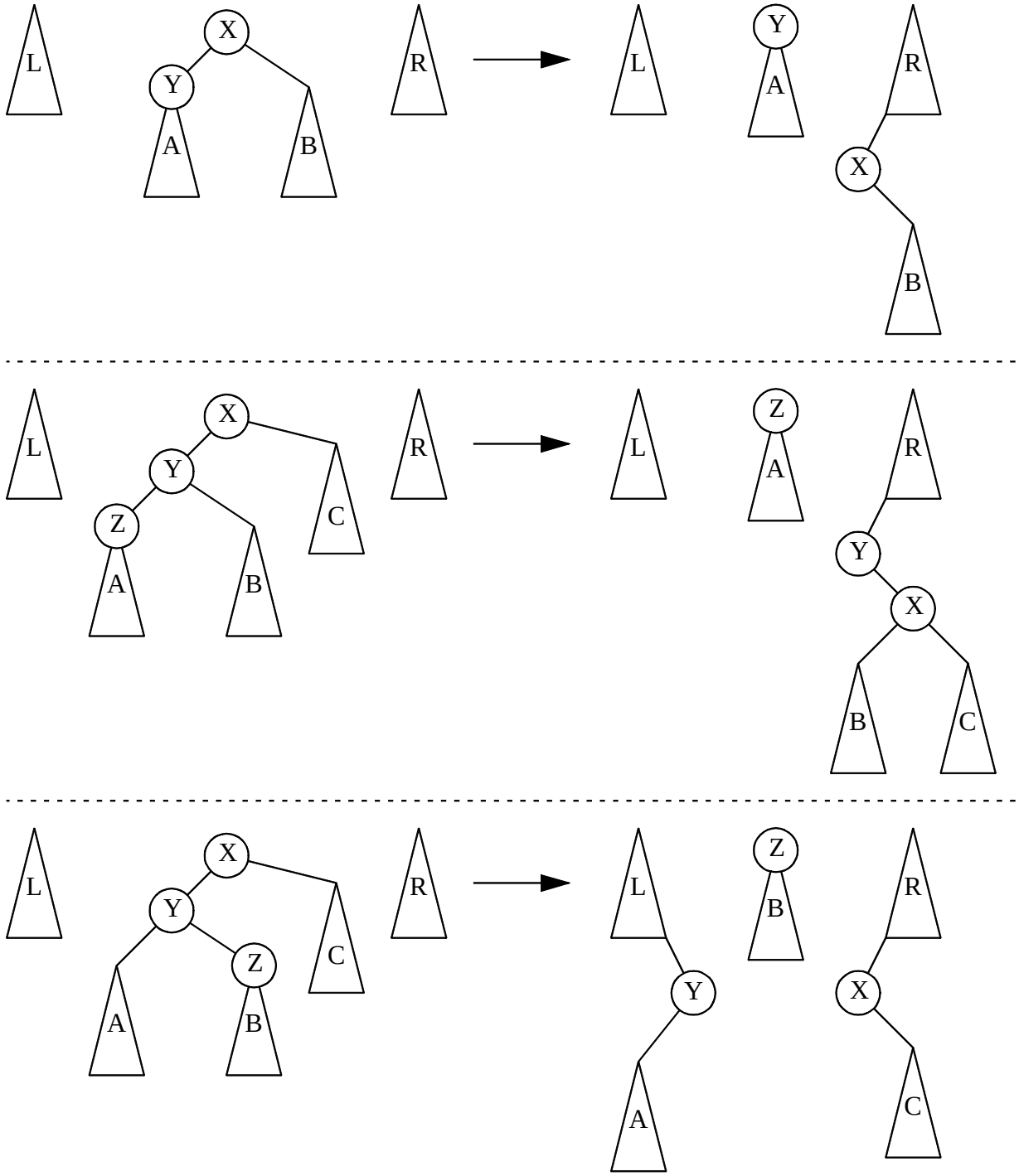
Zig-zig case (this is unique to the splay tree); symmetric case omitted



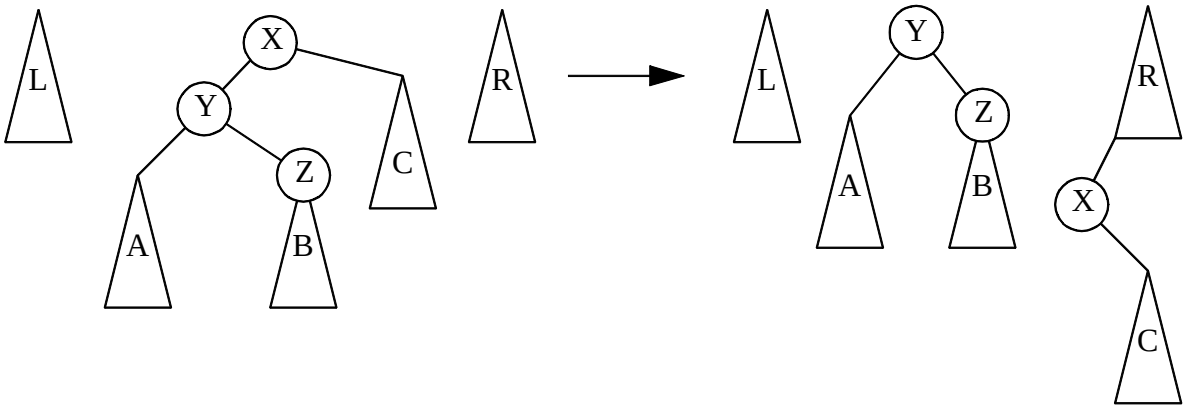
Result of splaying at node 1 (three zig-zigs and a zig)



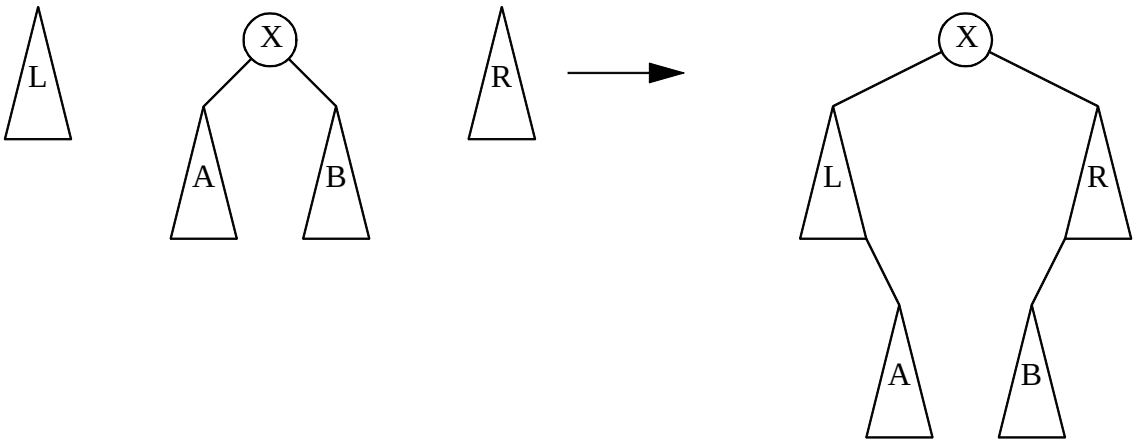
The `remove` operation applied to node 6: First 6 is splayed to the root, leaving two subtrees; a `findMax` on the left subtree is performed, raising 5 to the root of the left subtree; then the right subtree can be attached (not shown)



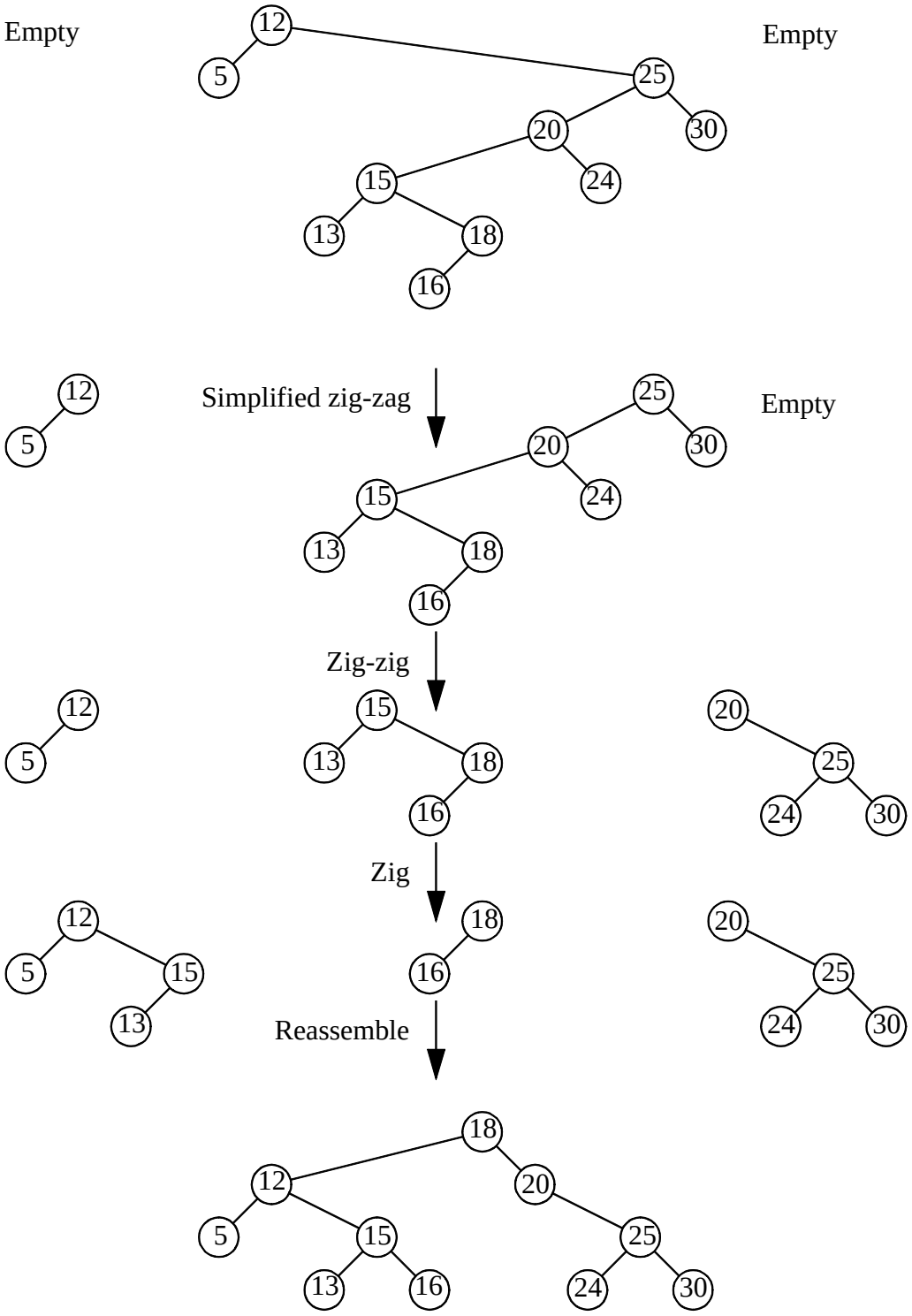
Top-down splay rotations: zig (top), zig-zig (middle), and zig-zag (bottom)



Simplified top-down zig-zag



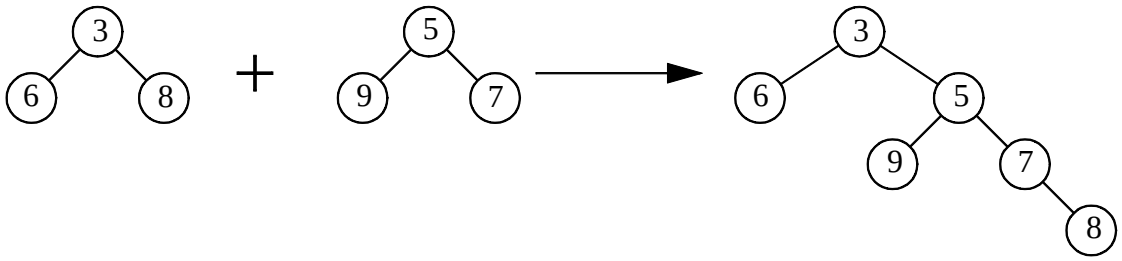
Final arrangement for top-down splaying



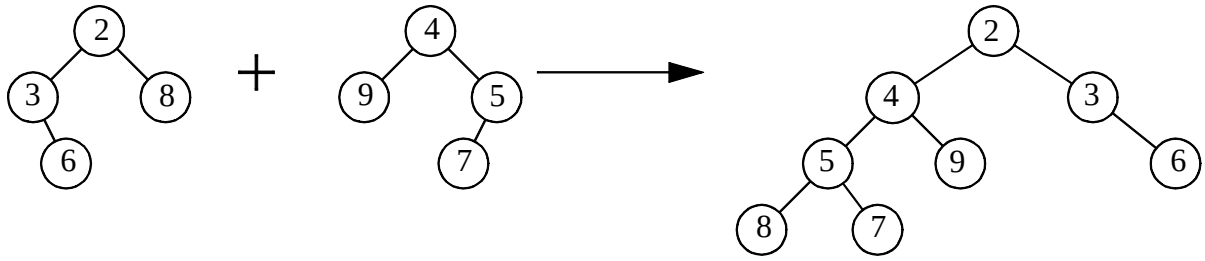
Steps in top-down splay (accessing 19 in top tree)

Chapter 22

Merging Priority Queues



Simplistic merging of heap-ordered trees; right paths are merged

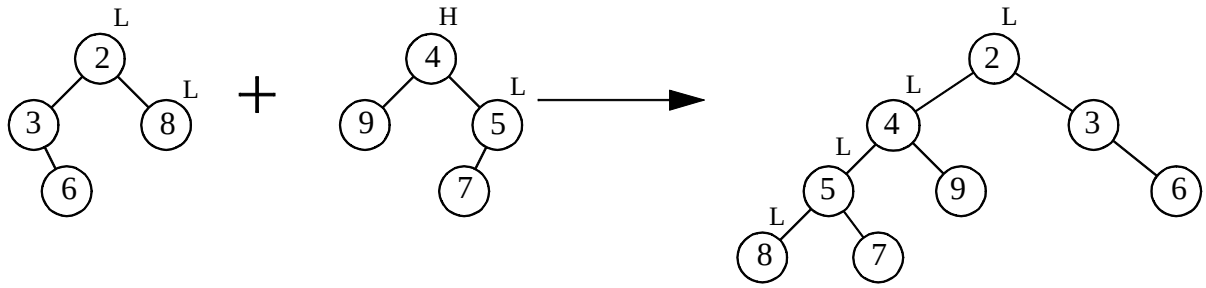


Merging of skew heap; right paths are merged, and the result is made a left path

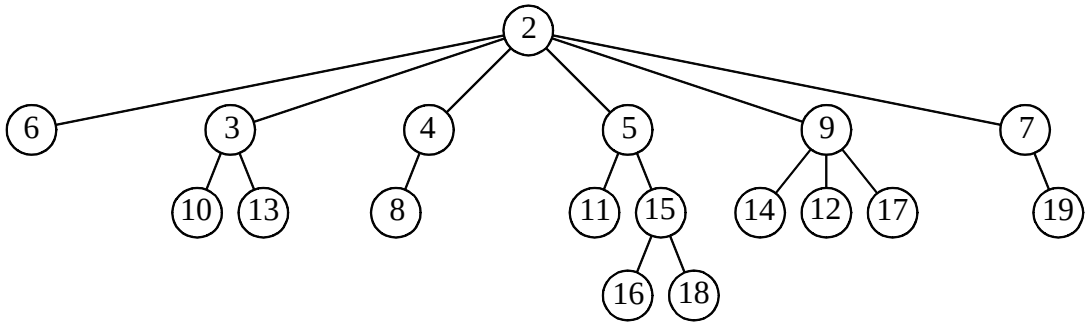
A recursive viewpoint is as follows: Let *L* be the tree with the smaller root, and let *R* be the other tree.

1. If one tree is empty, the other can be used as the merged result.
2. Otherwise, let *Temp* be the right subtree of *L*.
3. Make *L*'s left subtree its new right subtree.
4. Make the result of the recursive merge of *Temp* and *R* the new left subtree of *L*.

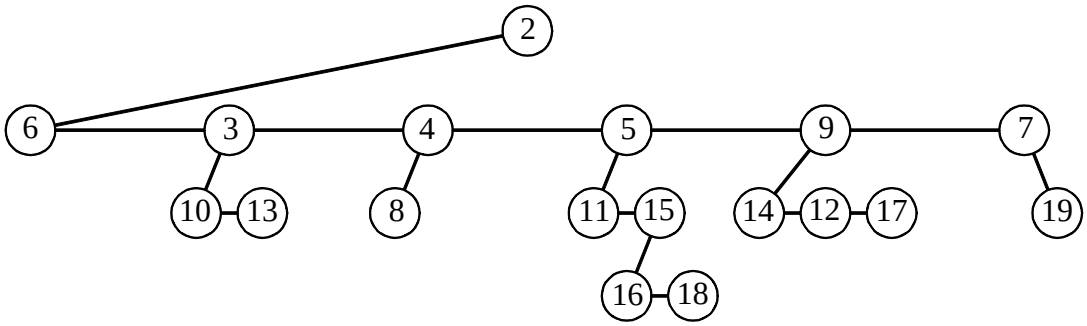
Skew heap algorithm (recursive viewpoint)



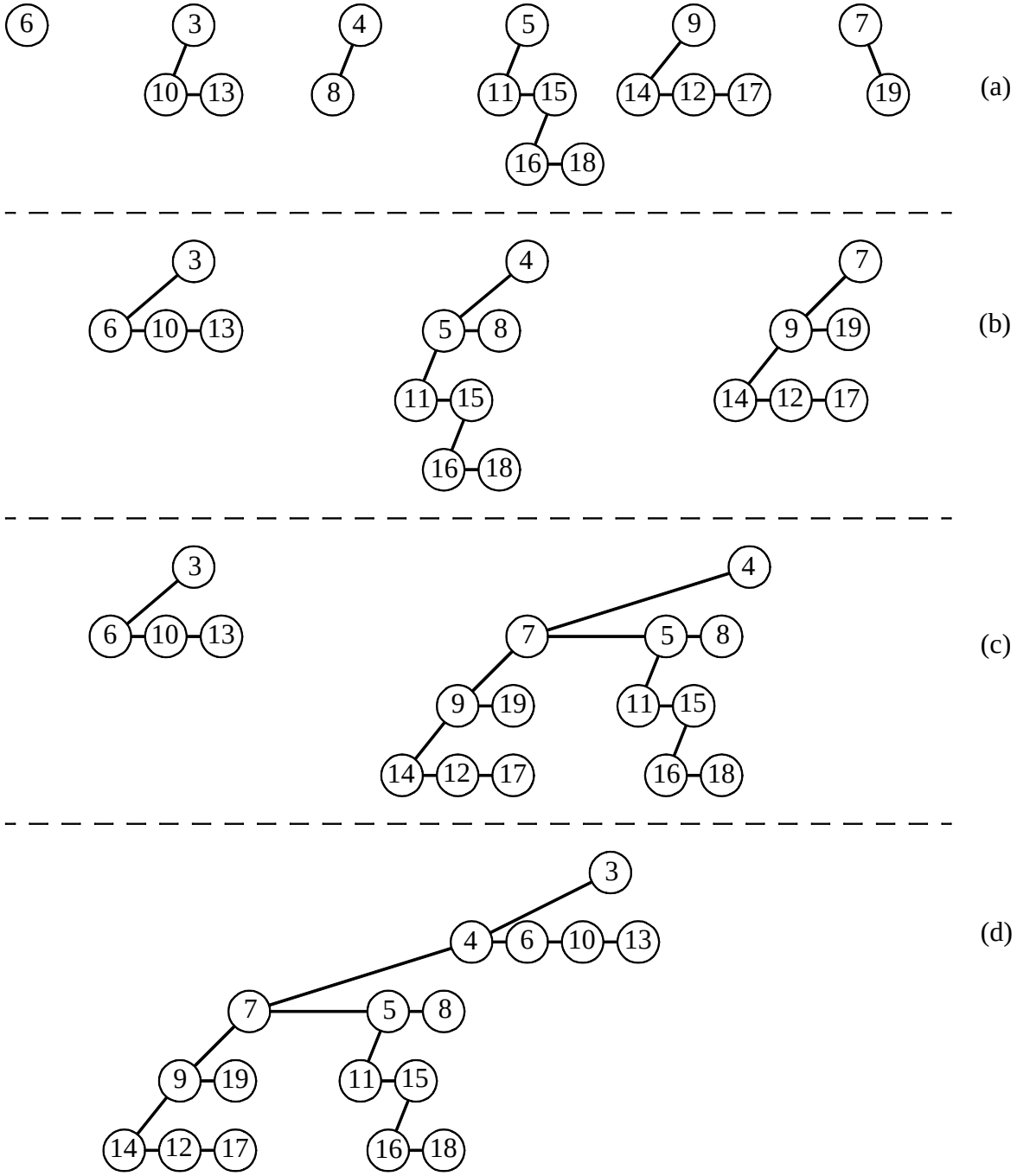
Change in heavy/light status after a merge



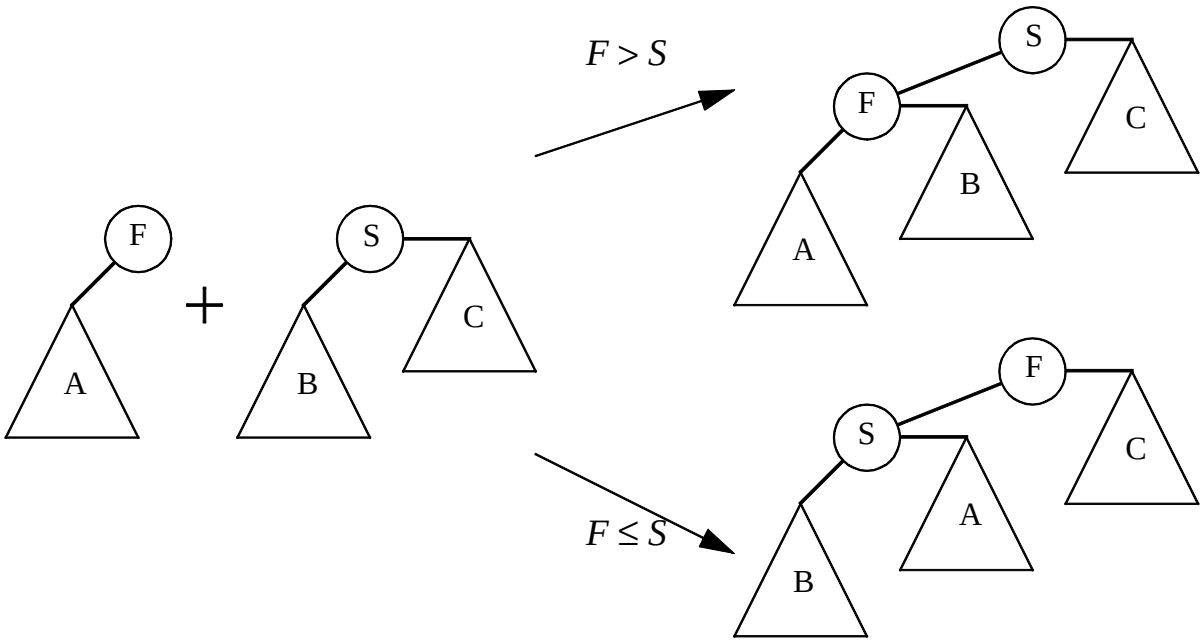
Abstract representation of sample pairing heap



Actual representation of above pairing heap; dark line represents a pair of references that connect nodes in both directions



Recombination of siblings after a `deleteMin`; in each merge the larger root tree is made the left child of the smaller root tree: (a) the resulting trees; (b) after the first pass; (c) after the first merge of the second pass; (d) after the second merge of the second pass



`compareAndLink` merges two trees

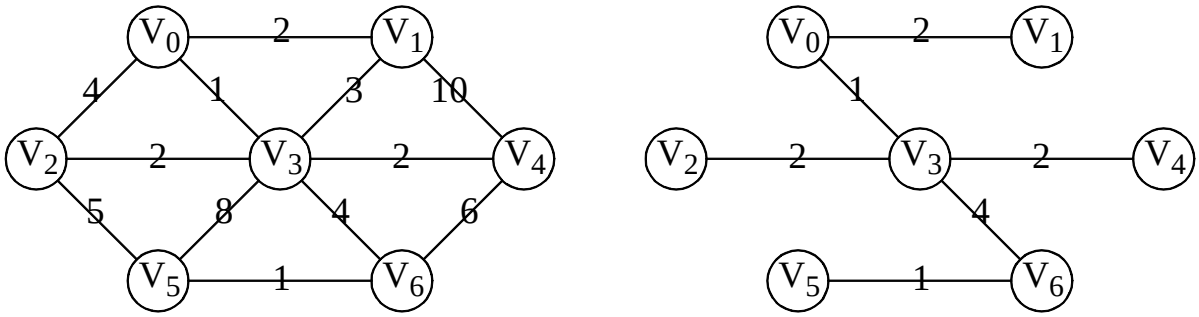
Chapter 23

The Disjoint Set Class

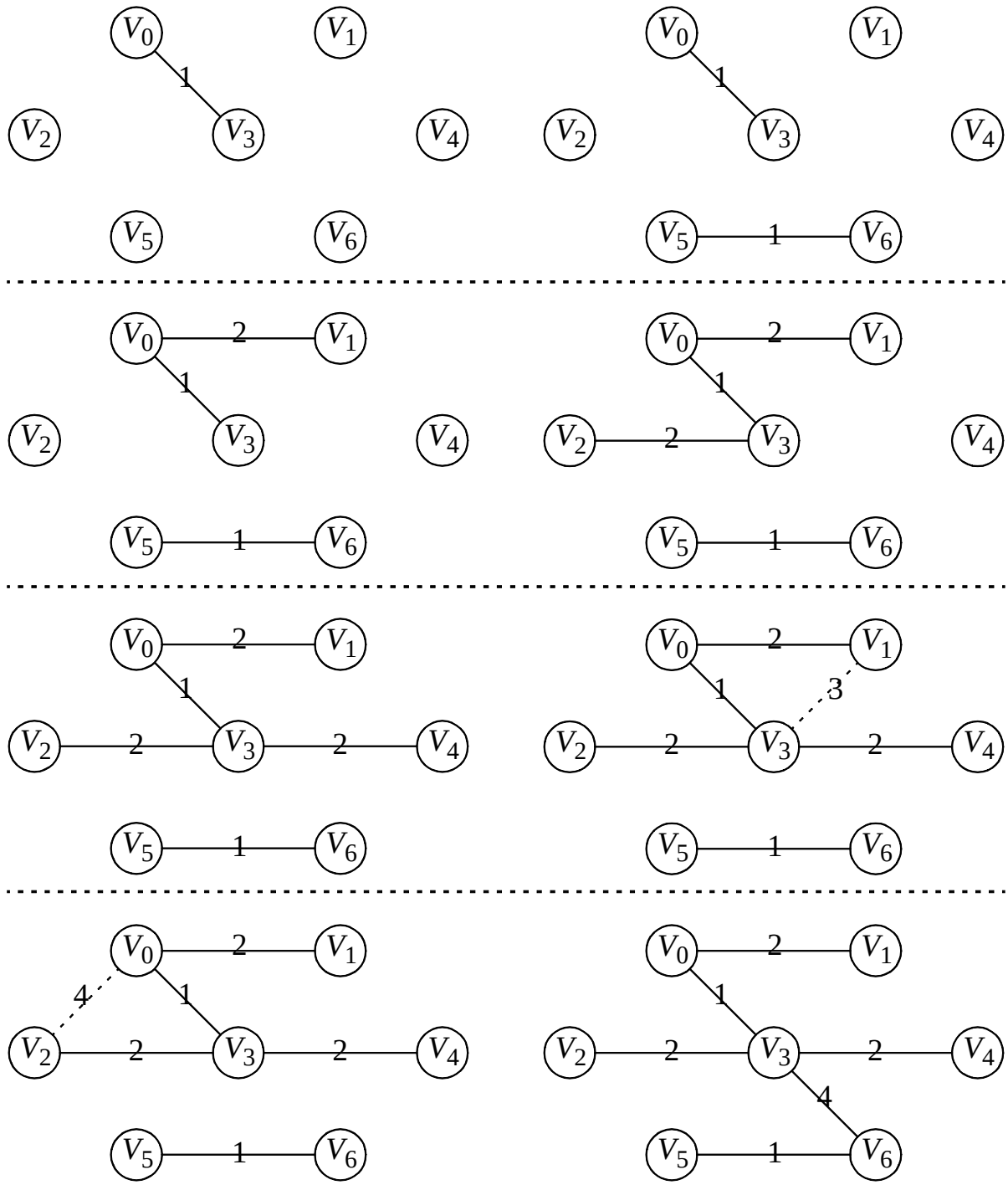
A *relation R* is defined on a set *S* if for every pair of elements (a, b), $a, b \in S$, $a\ R\ b$ is either true or false. If $a\ R\ b$ is true, then we say that *a* is related to *b*. An *equivalence relation* is a relation *R* that satisfies three properties:

- *Reflexive*: $a\ R\ a$ is true for all $a \in S$
- *Symmetric*: $a\ R\ b$ if and only if $b\ R\ a$
- *Transitive:* $a\ R\ b$ and $b\ R\ c$ implies that $a\ R\ c$

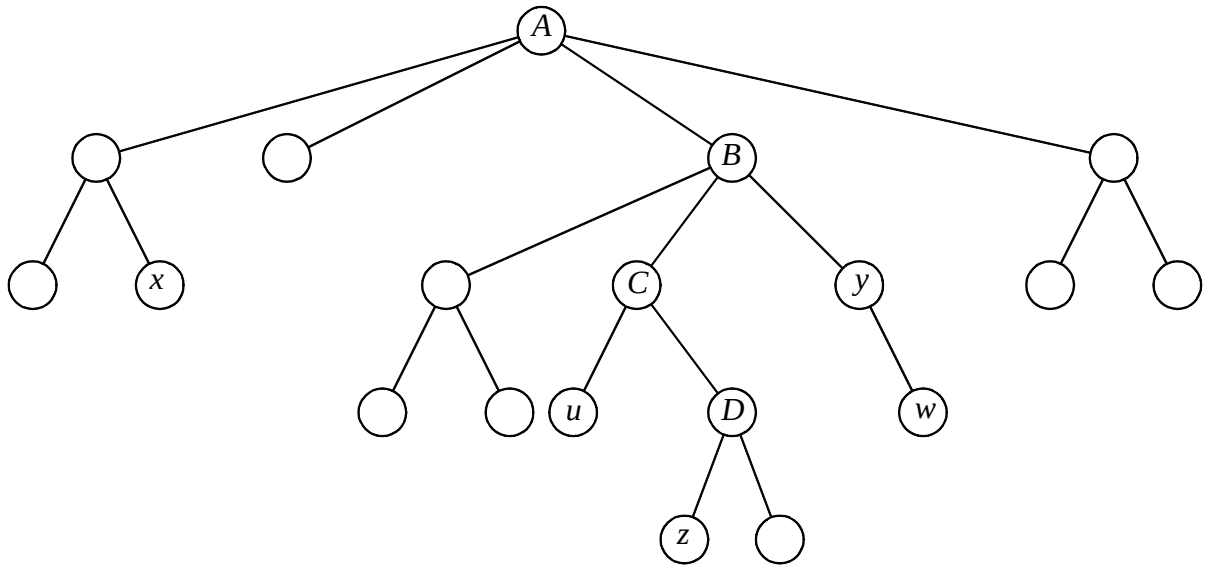
Definition of equivalence relation



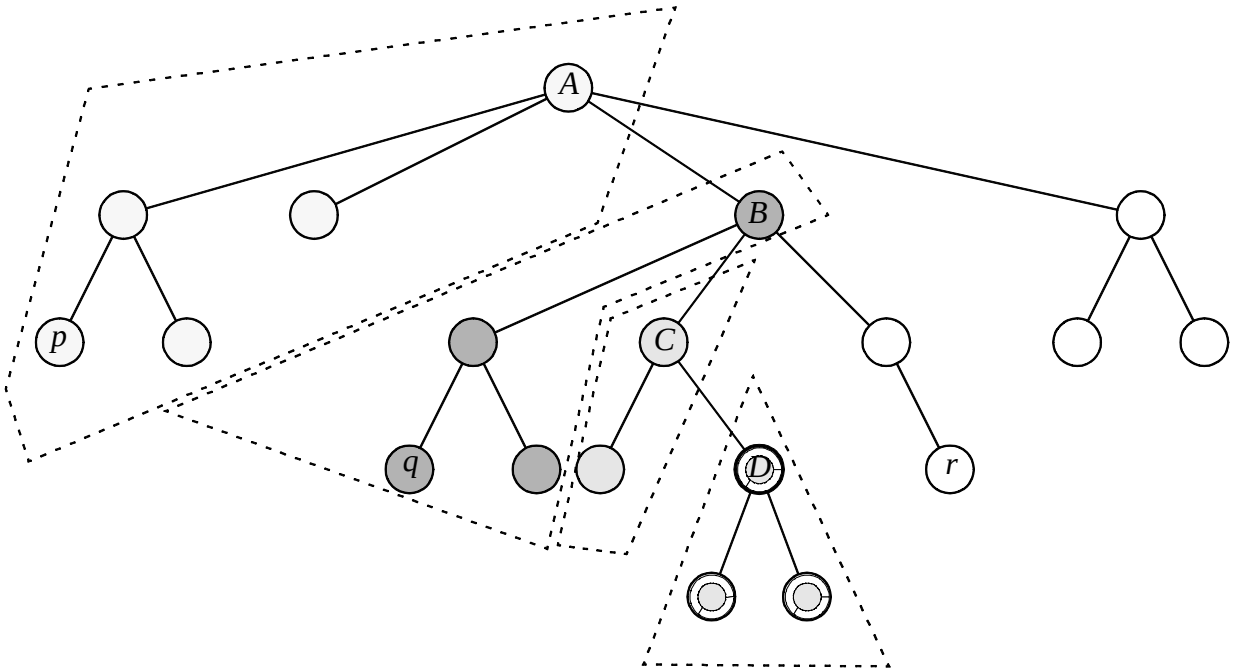
A graph *G* (left) and its minimum spanning tree



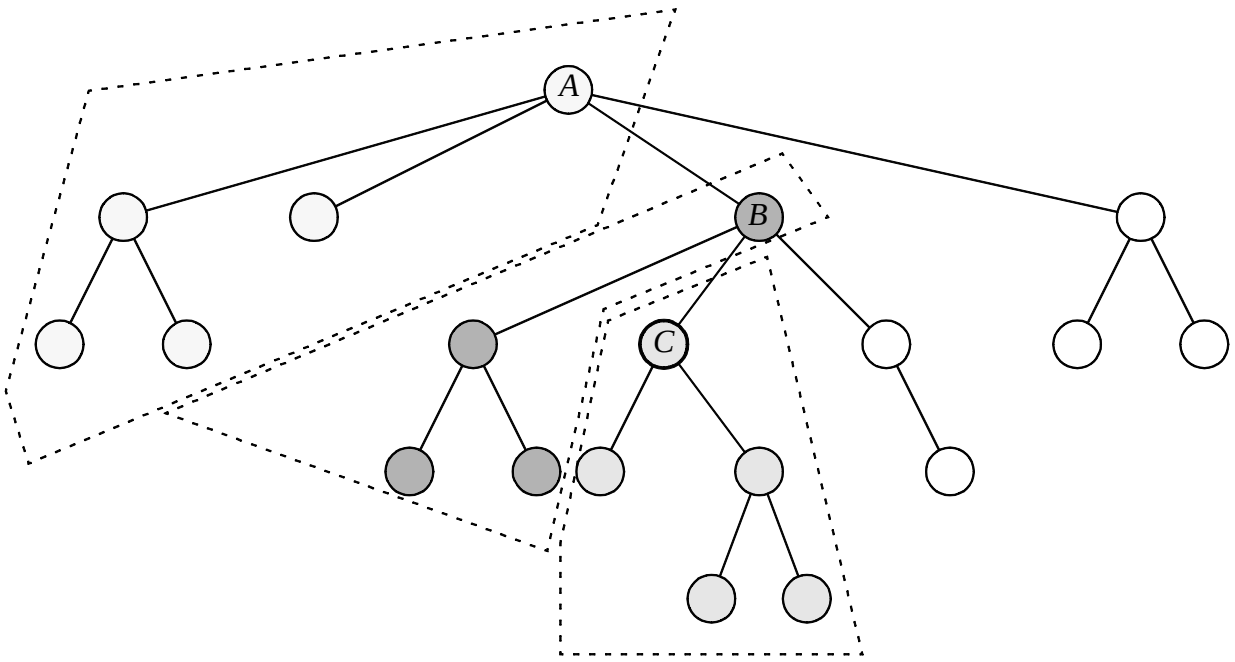
Kruskal's algorithm after each edge is considered



The nearest common ancestor for each request in the pair sequence (*x*,*y*), (*u*,*z*), (*w*,*x*), (*z*,*w*), (*w*,*y*), is *A*, *C*, *A*, *B*, and *y*, respectively

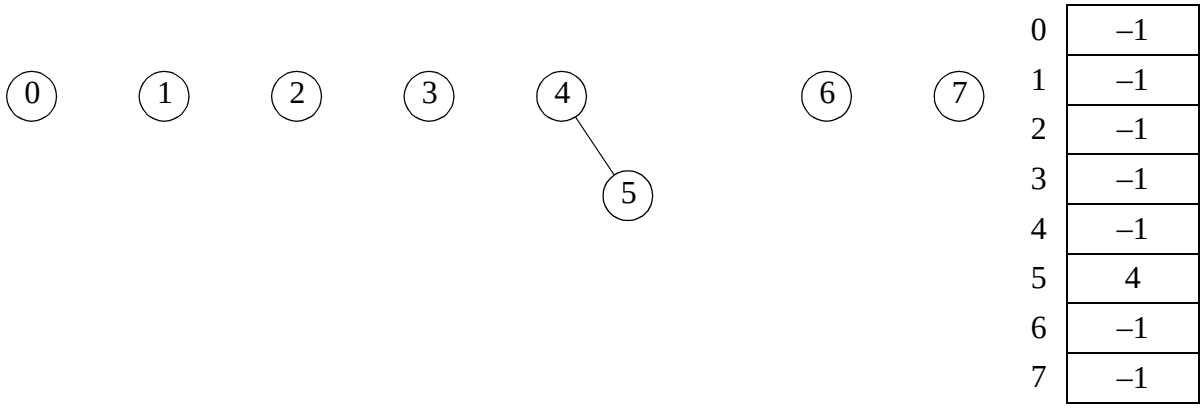


The sets immediately prior to the return from the recursive call to *D*; *D* is marked as visited and *NCA*(*D*, *v*) is *v*'s anchor to the current path

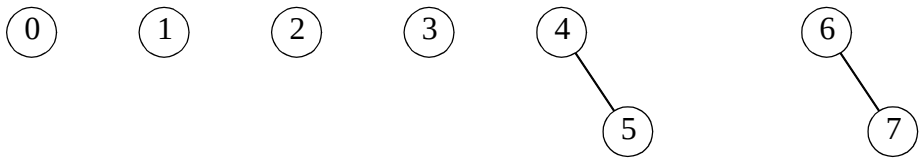


After the recursive call from *D* returns, we merge the set anchored by *D* into the set anchored by *C* and then compute all *NCA*(*C*, *v*) for nodes *v* that are marked prior to completing *C*'s recursive call

Forest and its eight elements, initially in different sets

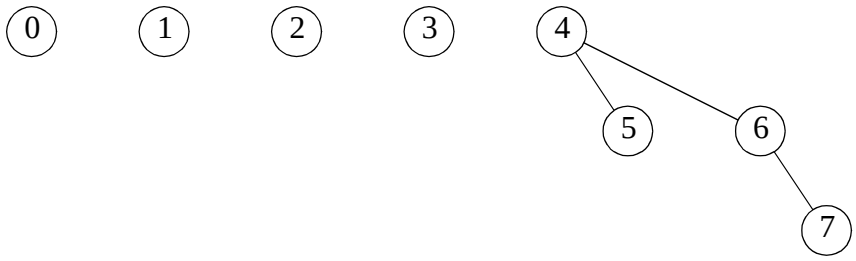


Forest after `union` of trees with roots 4 and 5



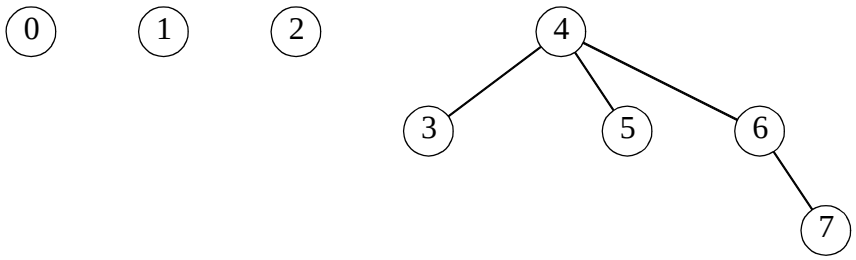
0	–1
1	–1
2	–1
3	–1
4	–1
5	4
6	–1
7	6

Forest after `union` of trees with roots 6 and 7



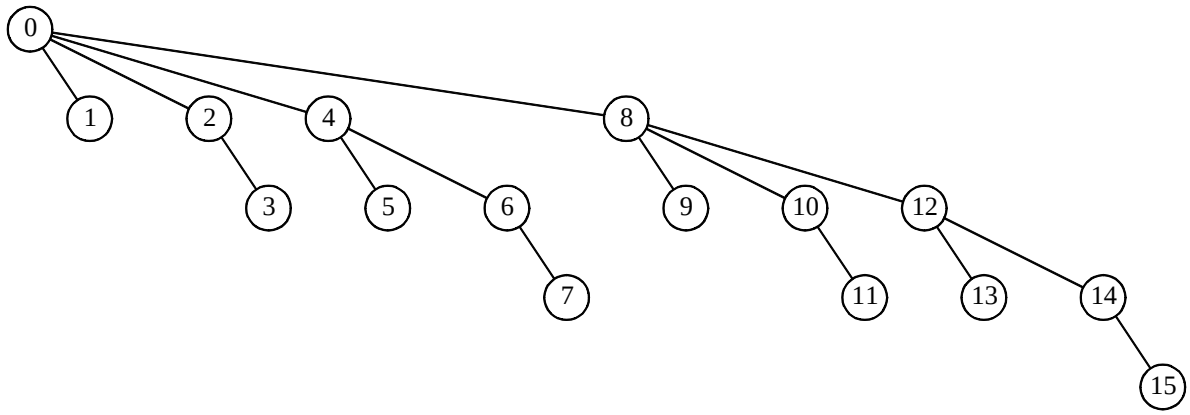
0	–1
1	–1
2	–1
3	–1
4	–1
5	4
6	4
7	6

Forest after `union` of trees with roots 4 and 6

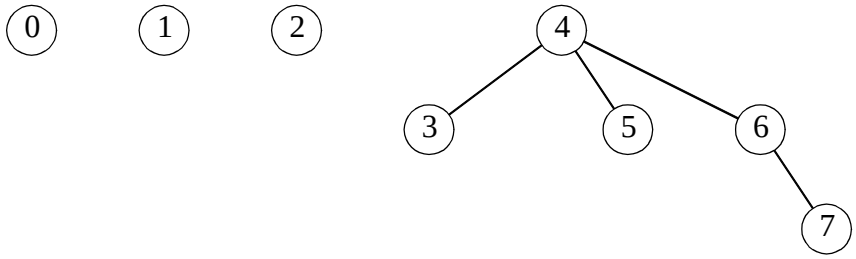


0	–1
1	–1
2	–1
3	4
4	–5
5	4
6	4
7	6

Forest formed by union-by-size, with size encoded as a negative number

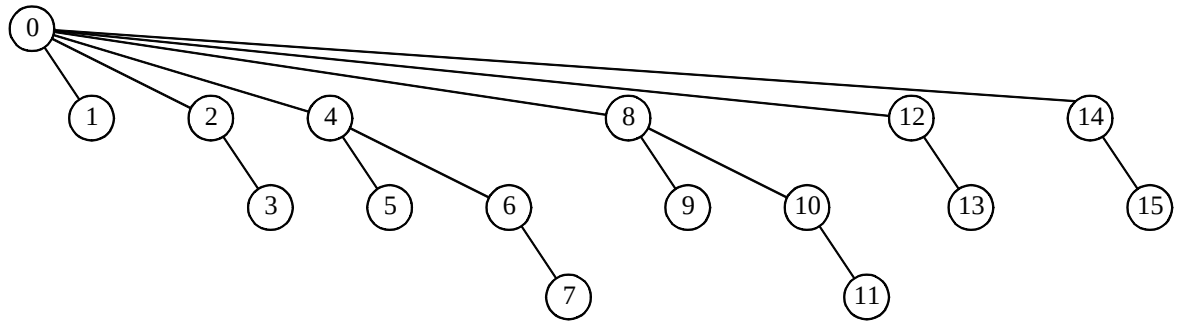


Worst-case tree for $N = 16$



0	–1
1	–1
2	–1
3	4
4	–3
5	4
6	4
7	6

Forest formed by union-by-height, with height encoded as a negative number



Path compression resulting from a `find`(14) on the tree in Figure 23.12

Ackermann's function is defined as:

$$A(1, j) = 2^j \qquad j \geq 1$$
$$A(i, 1) = A(i-1, 2) \qquad i \geq 2$$
$$A(i, j) = A(i-1, A(i, j-1)) \qquad i, j \geq 2$$

From this, we define the inverse Ackermann's function as

$$\alpha(M, N) = min\{i \geq 1 | (A(i, \lfloor M/N \rfloor) > \log N)\}$$

Ackermann's function and its inverse

To incorporate path compression into the proof, we use the following fancy accounting: For each node *v* on the path from the accessed node *i* to the root, we deposit one penny under one of two accounts:

1. If *v* is the root, or if the parent of *v* is the root, or if the parent of *v* is in a different rank group from *v*, then charge one unit under this rule. This deposits an American penny into the kitty.
2. Otherwise, deposit a Canadian penny into the node.

Accounting used in union-find proof

Group	Rank
0	0
1	1
2	2
3	3,4
4	5 through 16
5	17 through 65536
6	65537 through 2^{65536}
7	Truly huge ranks

Actual partitioning of ranks into groups used in the union-find proof