

Chapter 19

Hash Tables

Hash(89, 10) = 8
 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 A

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

Linear probing hash table after each insertion

Hash(89, 10) = 8
 Hash(18, 10) = 8
 Hash(49, 10) = 9
 Hash(58, 10) = 8
 Hash(9, 10) = 9

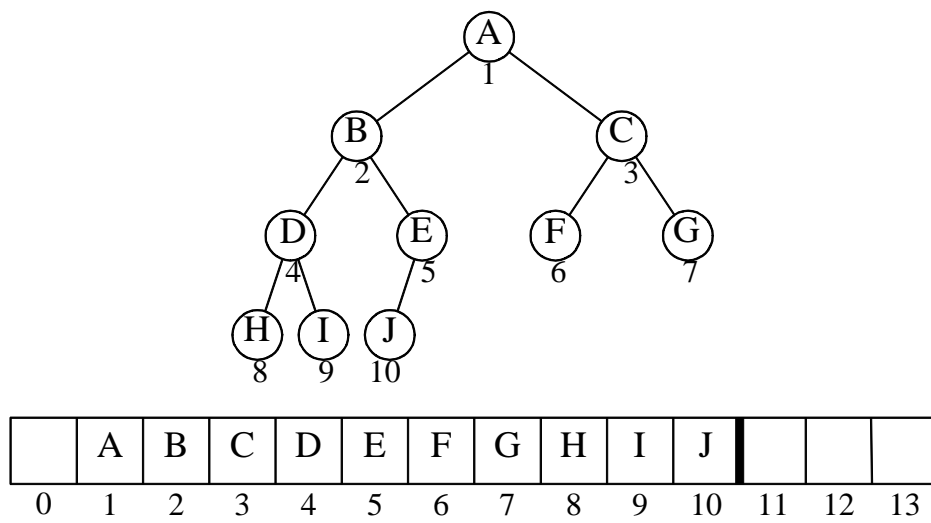
After Insert 89 After Insert 18 After Insert 49 After Insert 58

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

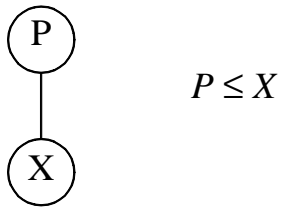
Quadratic probing hash table after each insertion (note that the table size is poorly chosen because it is not a prime number)

Chapter 20

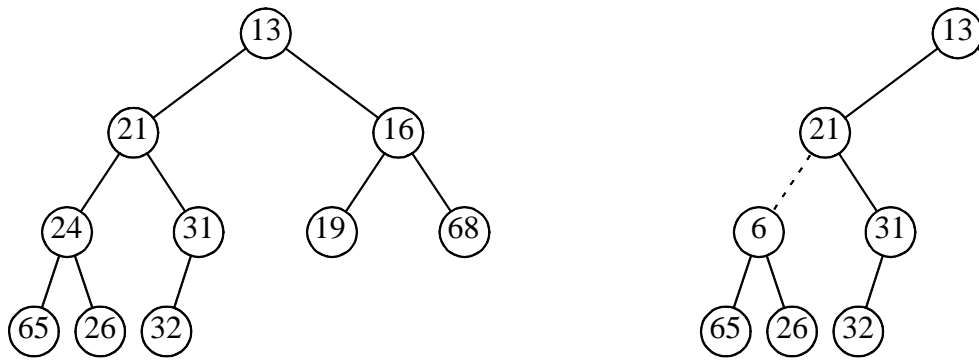
A Priority Queue: The Binary Heap



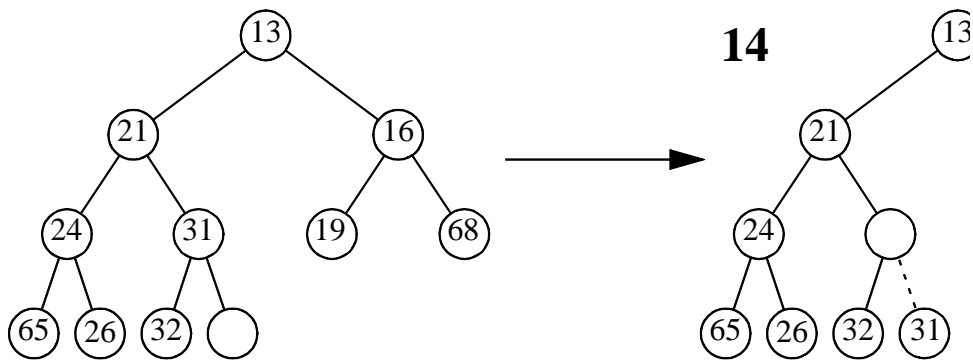
A complete binary tree and its array representation



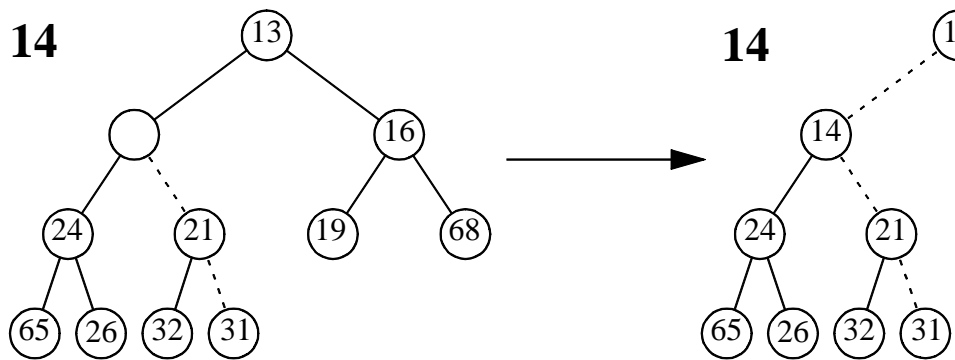
Heap order property



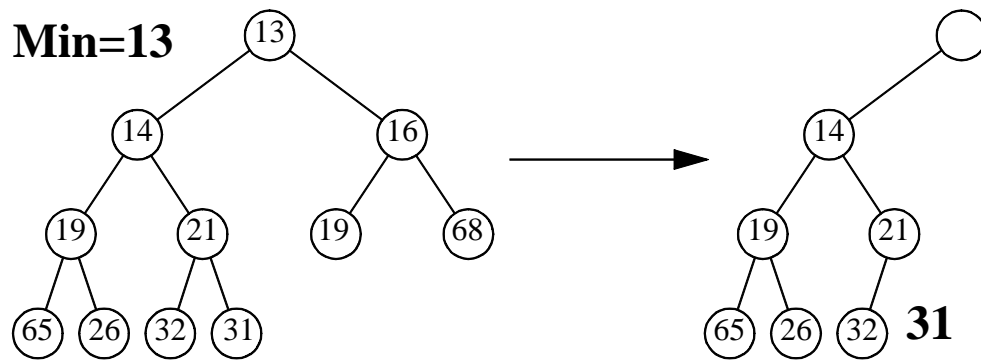
Two complete trees (only the left tree is a heap)



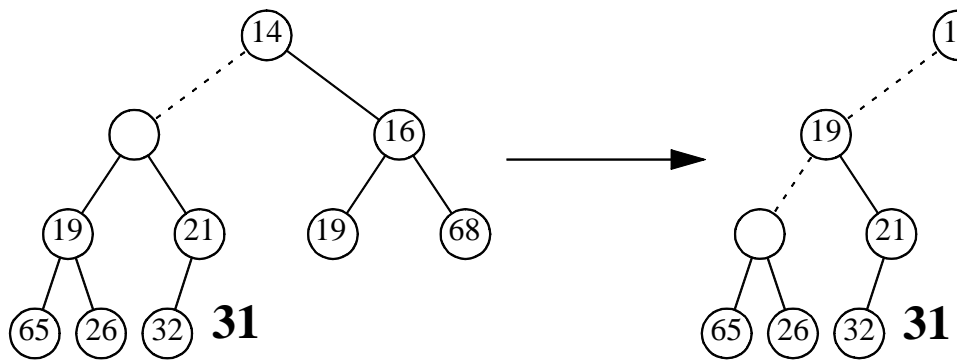
Attempt to insert 14, creating the hole and bubbling the hole up



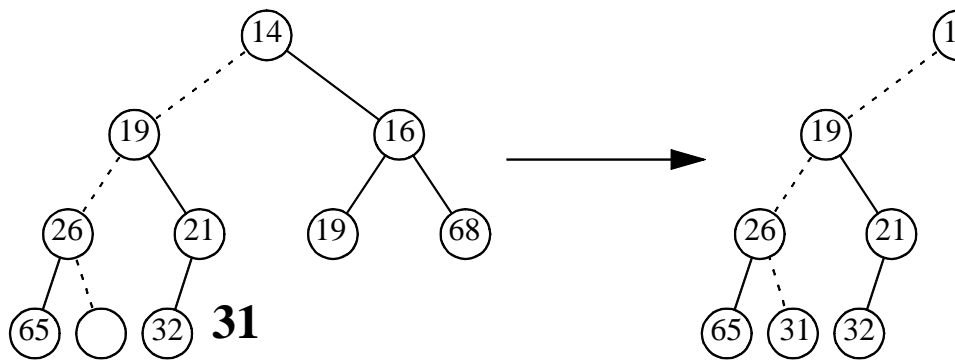
The remaining two steps to insert 14 in previous heap



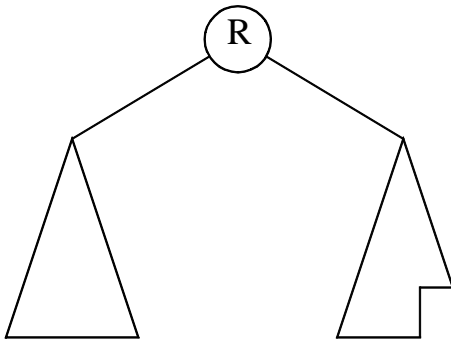
Creation of the hole at the root



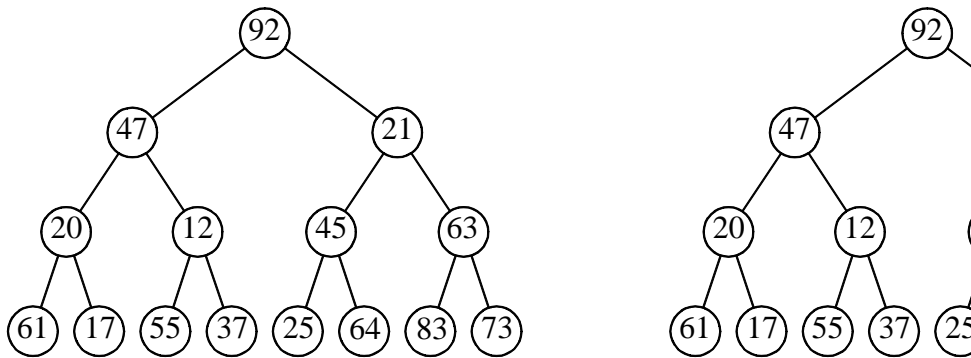
Next two steps in DeleteMin



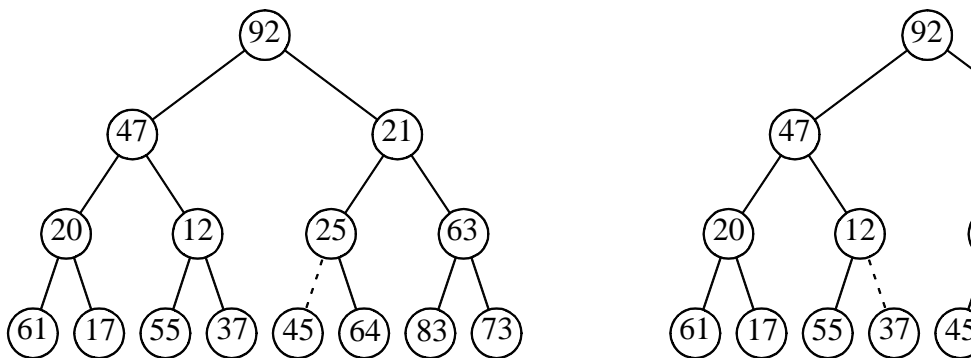
Last two steps in `DeleteMin`



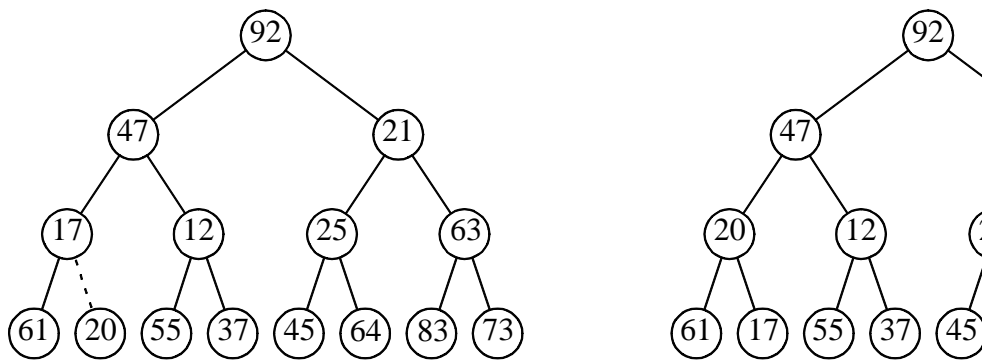
Recursive view of the heap



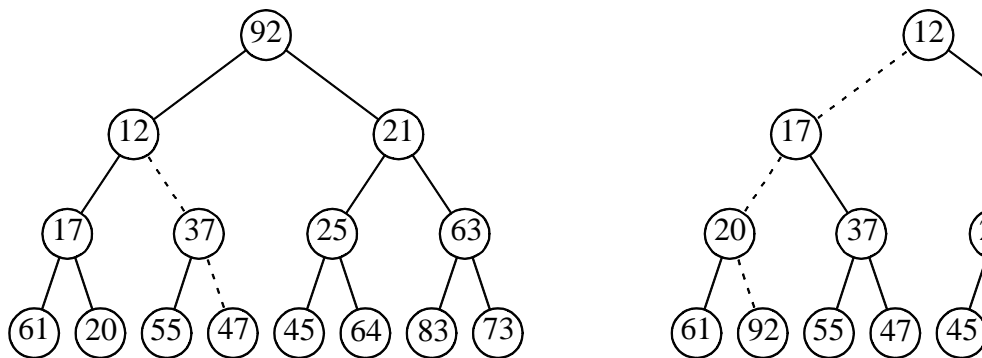
Initial heap (left); after `PercolateDown(7)` (right)



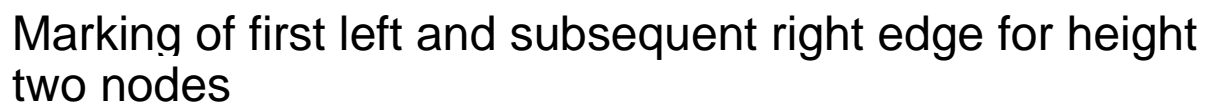
After `PercolateDown(6)` (left); after
`PercolateDown(5)` (right)

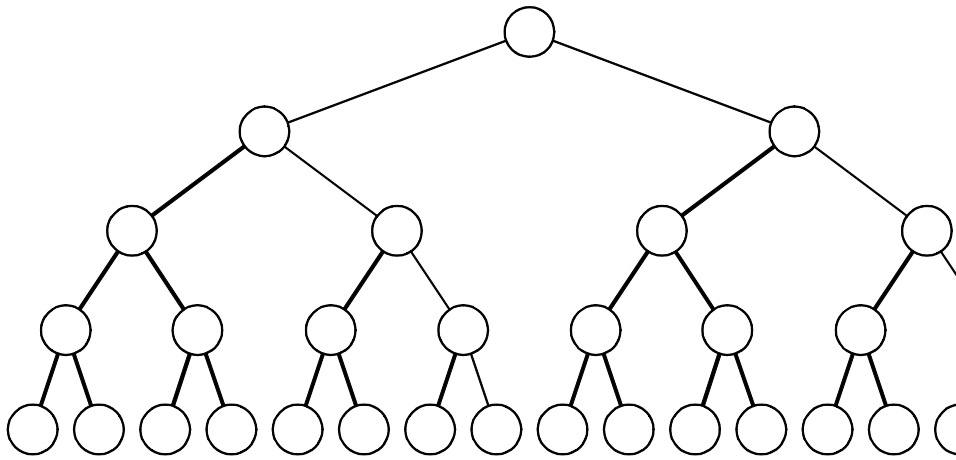


After `PercolateDown(4)` (left); after
`PercolateDown(3)` (right)

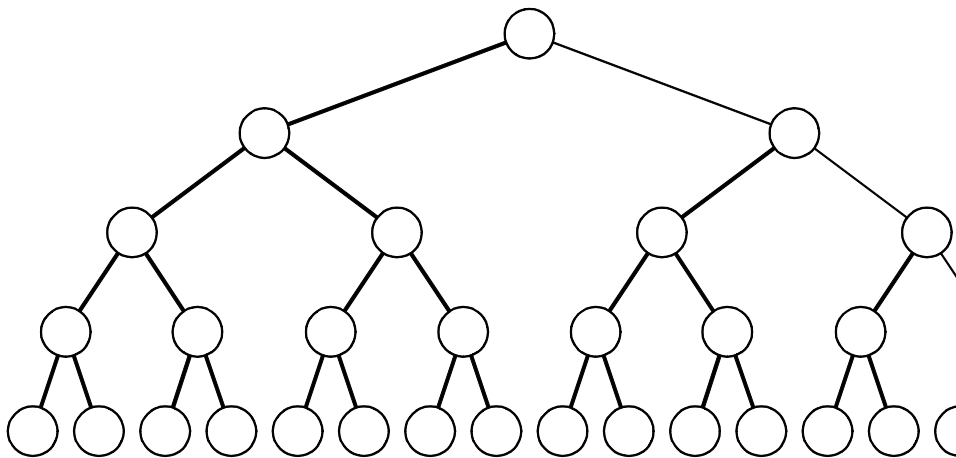


After `PercolateDown(2)` (left); after
`PercolateDown(1)` and `FixHeap` terminates
(right)

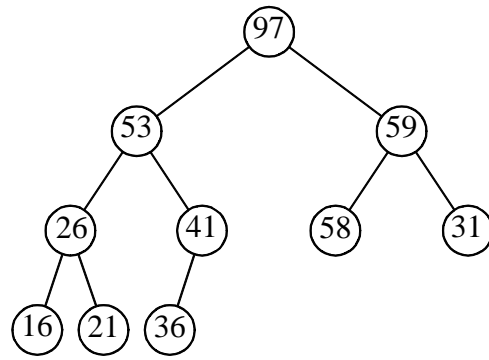




Marking of first left and subsequent two right edges for height three nodes



Marking of first left and subsequent right edges for height 4 node

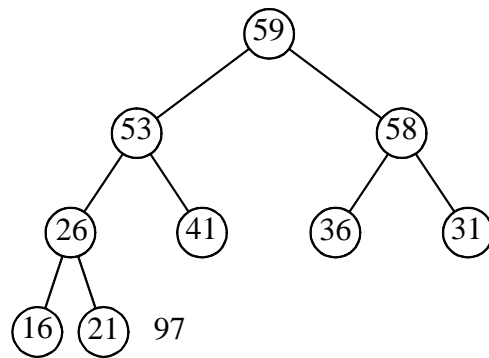


97	53	59	26	41	58	31	16	21	36				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

(Max) Heap after `FixHeap` phase

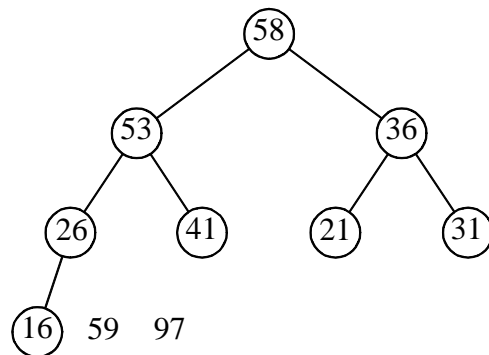
1. Toss each item into a binary heap.
2. Apply `FixHeap`.
3. Call `DeleteMin` N times; the items will exit the heap in sorted order.

Heapsort algorithm



59	53	58	26	41	36	31	16	21	97				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Heap after first DeleteMax



58	53	36	26	41	21	31	16	59	97				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

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

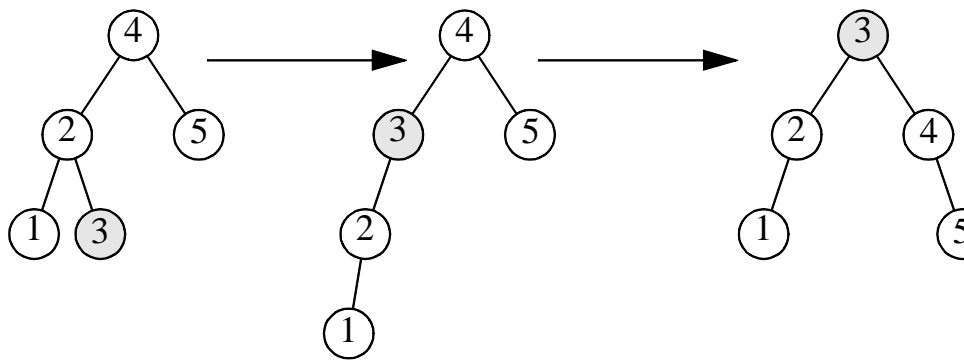
Number of runs using polyphase merge

	3 Elements in Heap Array			Output	Next Item Read
	Array[1]	Array[2]	Array[3]		
Run1	11	94	81	11	96
Run 1	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	End of Tape
	99		75	99	
			75	End of Run	Rebuild Heap
Run 3	75			75	

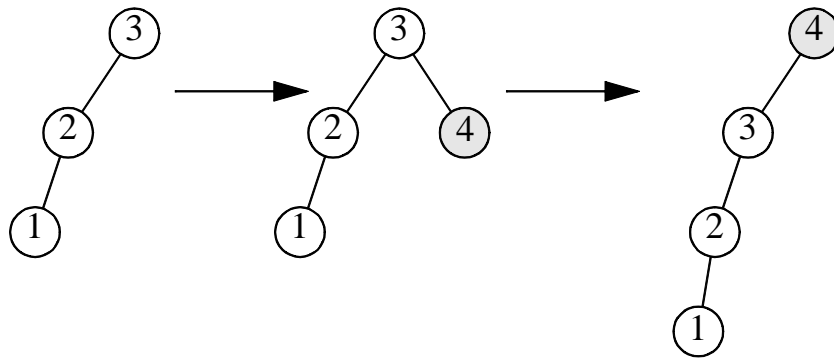
Example of run construction

Chapter 21

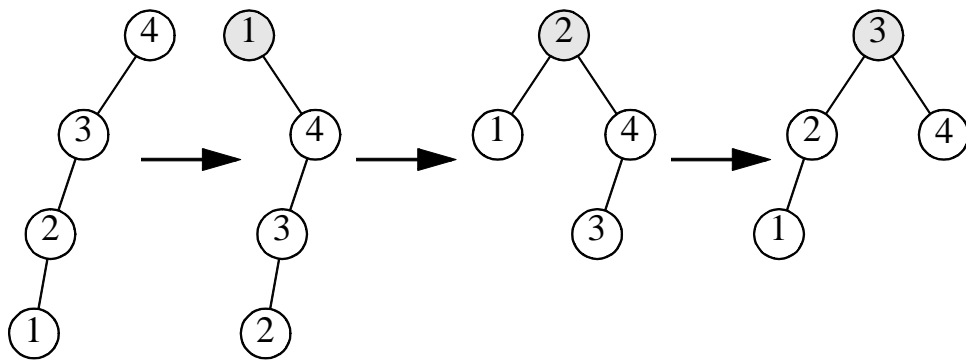
Splay Trees



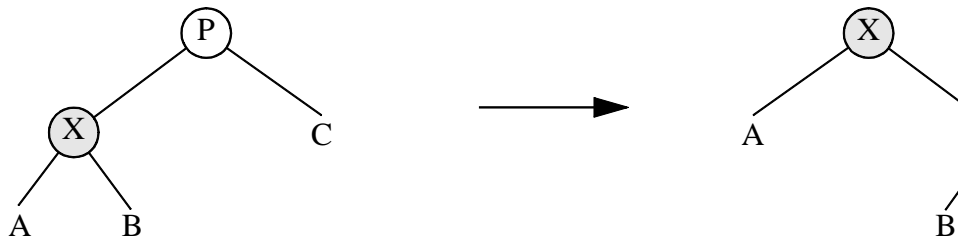
Rotate-to-root strategy applied when node 3 is accessed



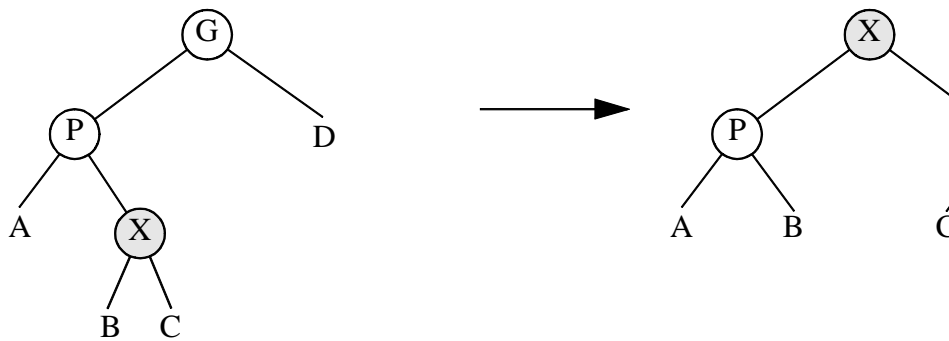
Insertion of 4 using rotate-to-root



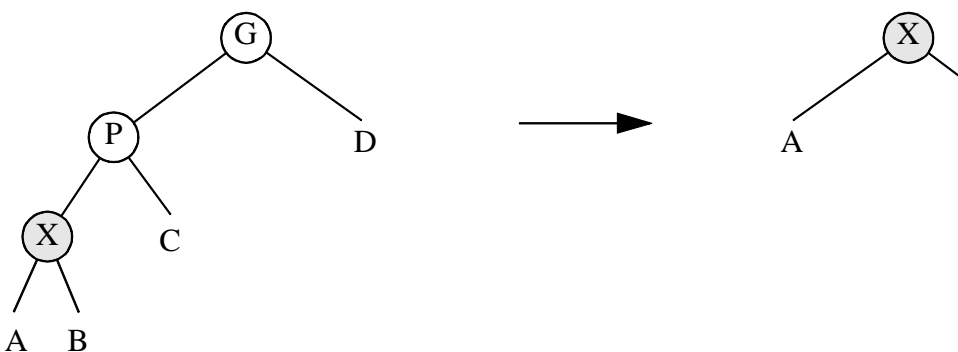
Sequential access of items takes quadratic time



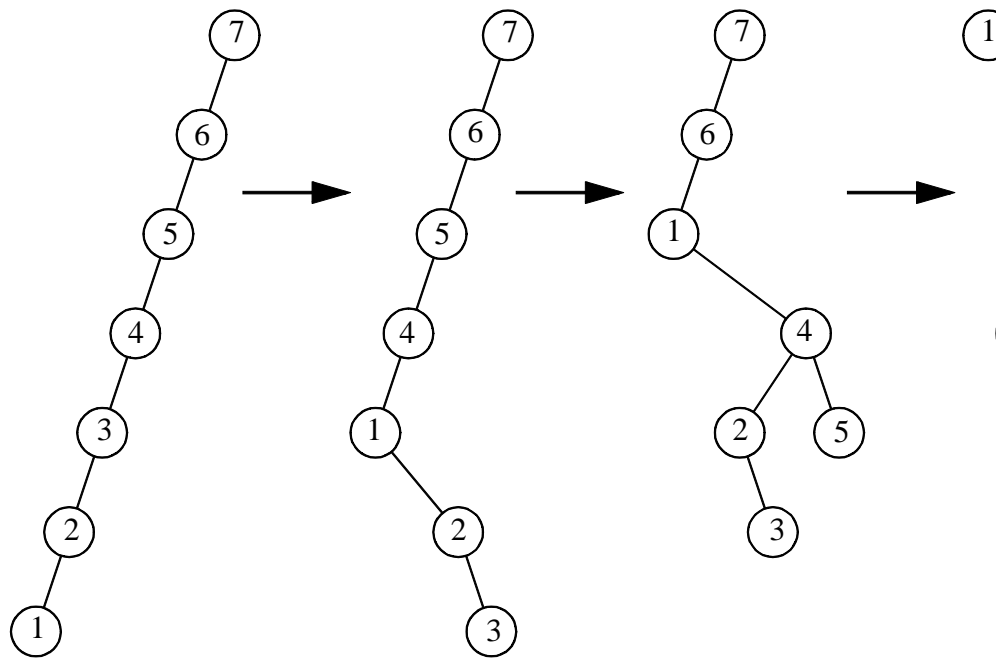
Zig case (normal single rotation)



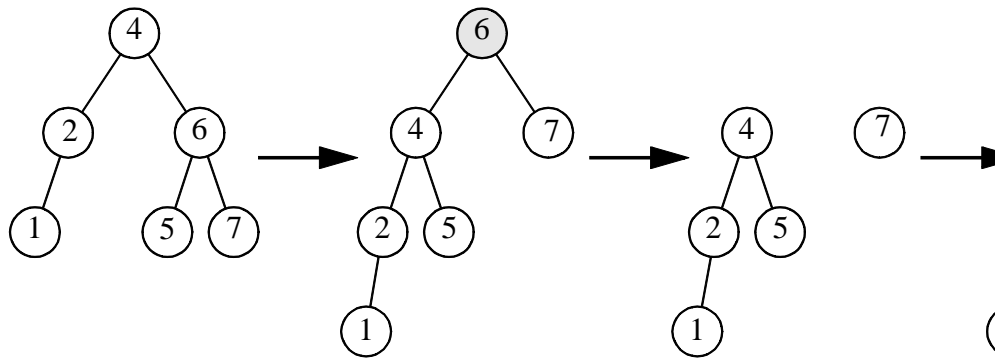
Zig-zag case (same as a double rotation); symmetric case omitted



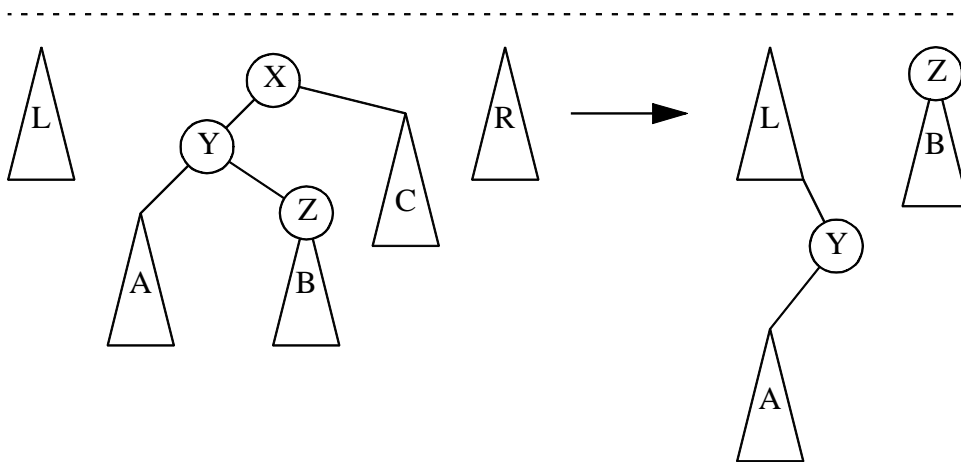
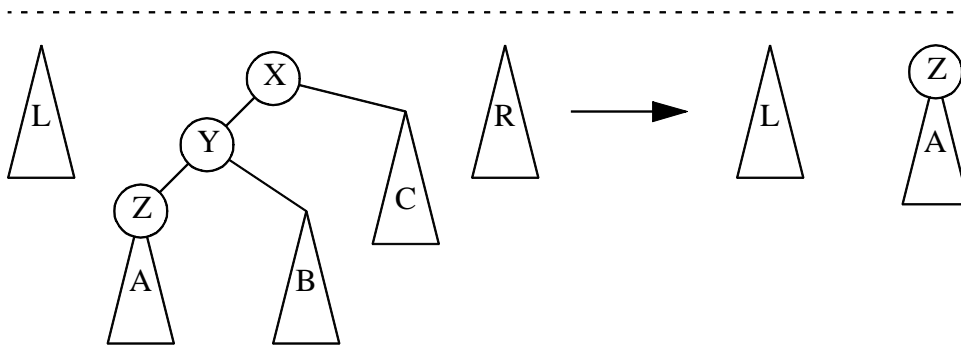
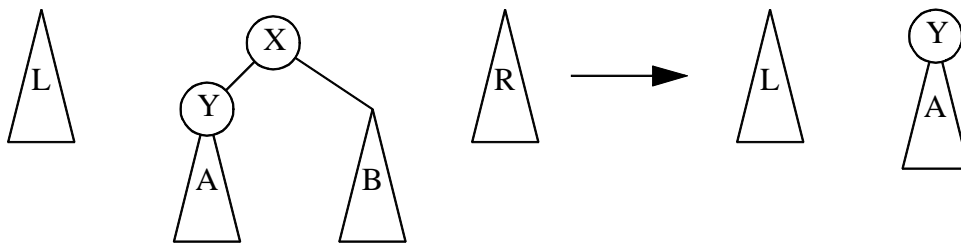
Zig-zig case (this is unique to the splay tree); symmetric case omitted



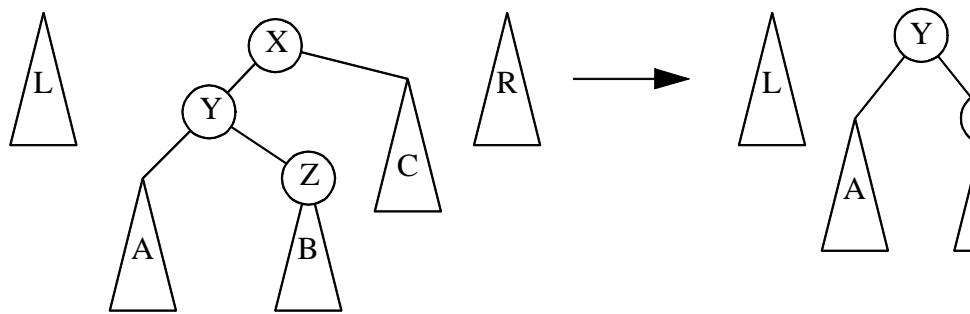
Result of splaying at node 1 (three zig-zigs and a zig)



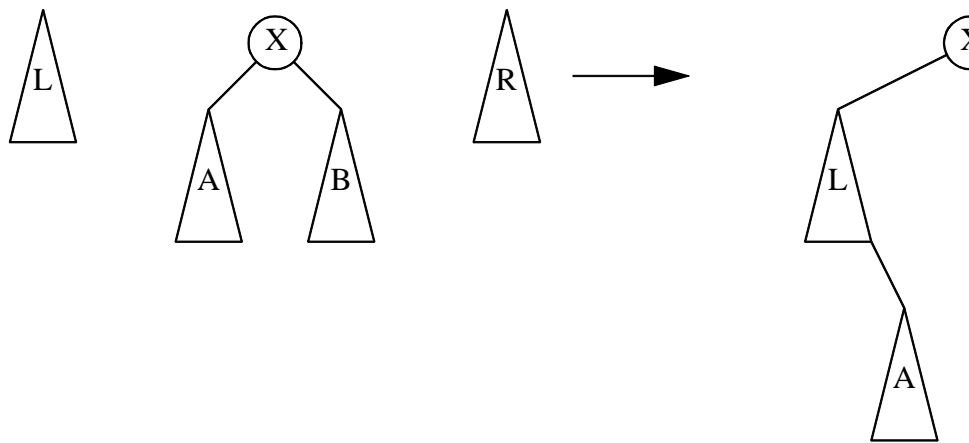
The Remove operation applied to node 6: First 6 is splayed to the root, leaving two subtrees; a FindMax on the left subtree is performed, raising 5 to the root of the left subtree; then the right subtree can be attached (not shown)



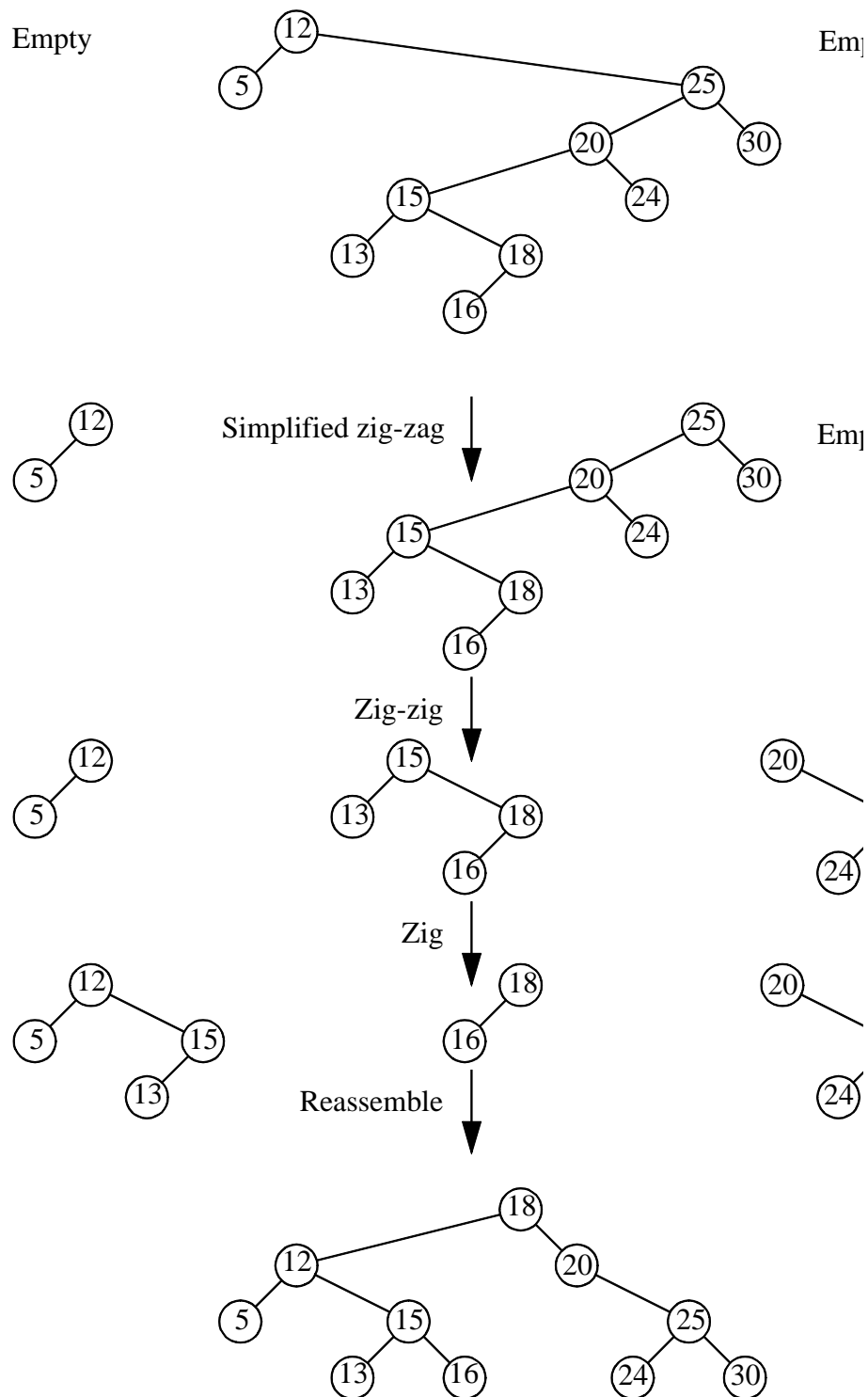
Top-down splay rotations: zig (top), zig-zig (middle), and zig-zag (bottom)



Simplified top-down zig-zag



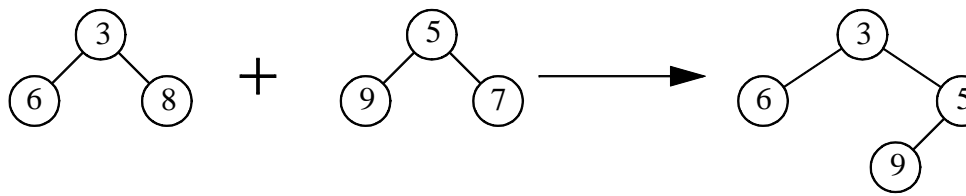
Final arrangement for top-down splaying



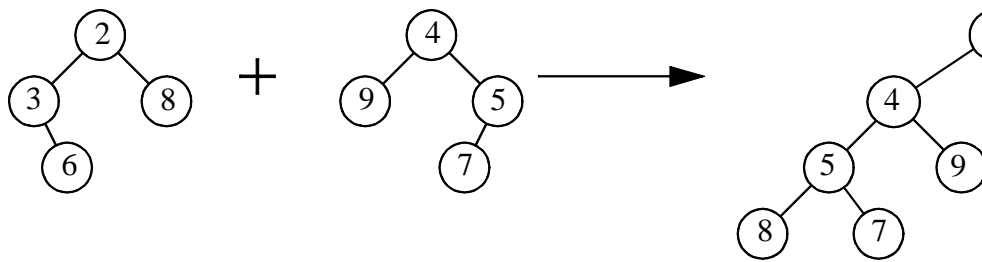
Steps in top-down splay (accessing 19 in top tree)

Chapter 22

Merging Priority Queues



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

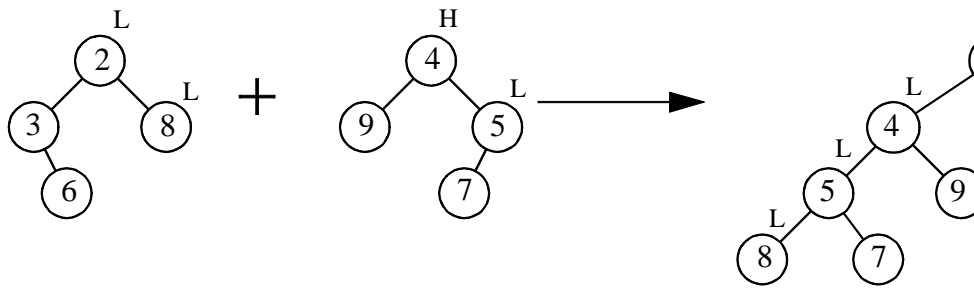


Merging of skew heap; right paths are merged, and the result is made a left path

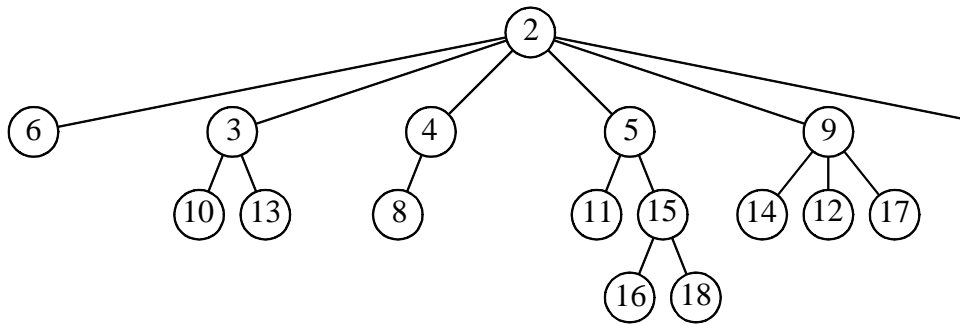
A recursive viewpoint is as follows: Let S be the tree with the smaller root, and let R be the other tree.

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 .

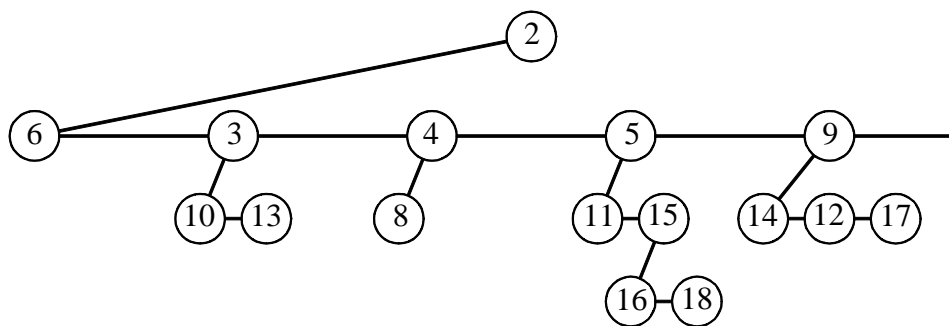
Skew heap algorithm (recursive viewpoint)



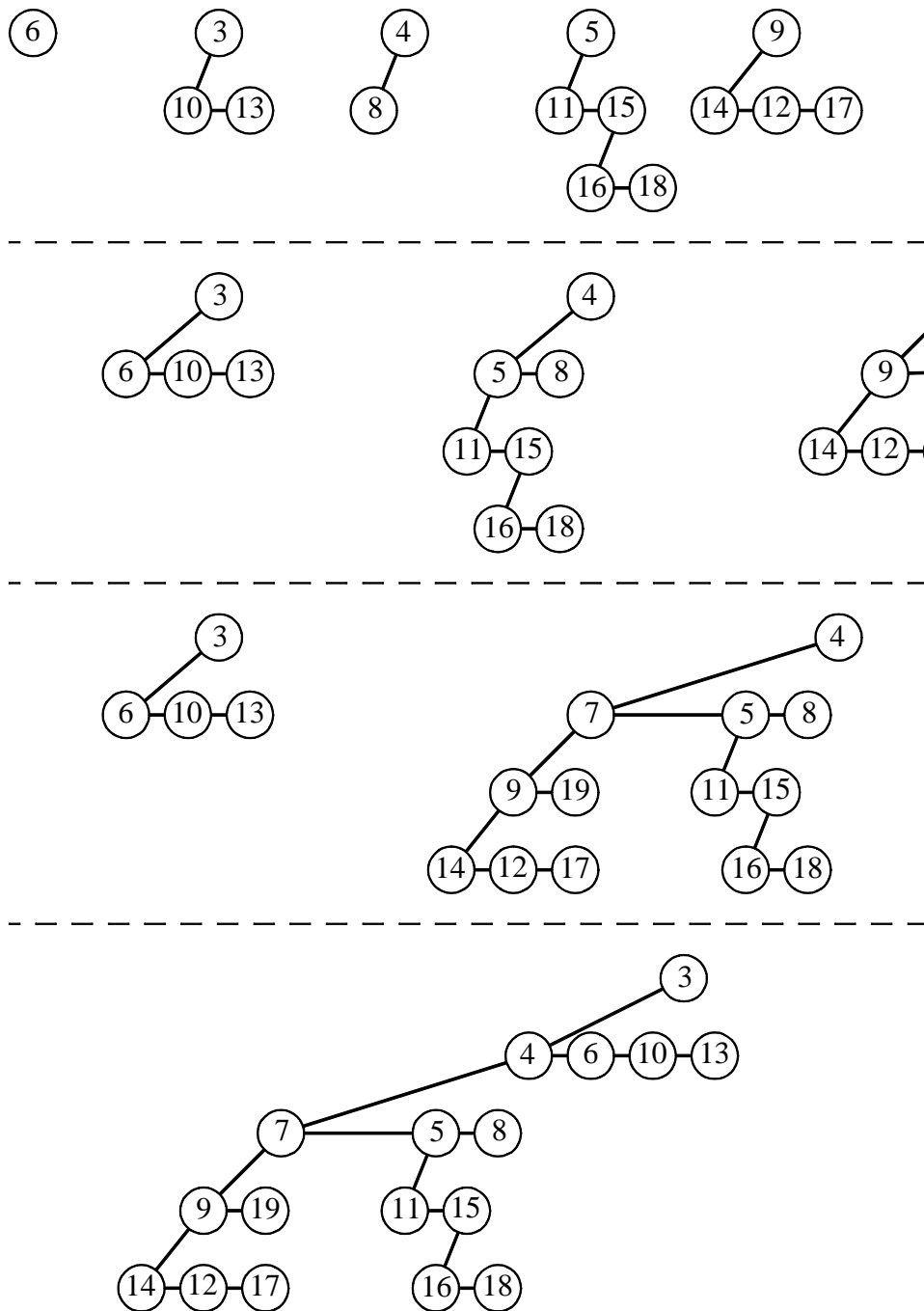
Change in heavy/light status after a merge



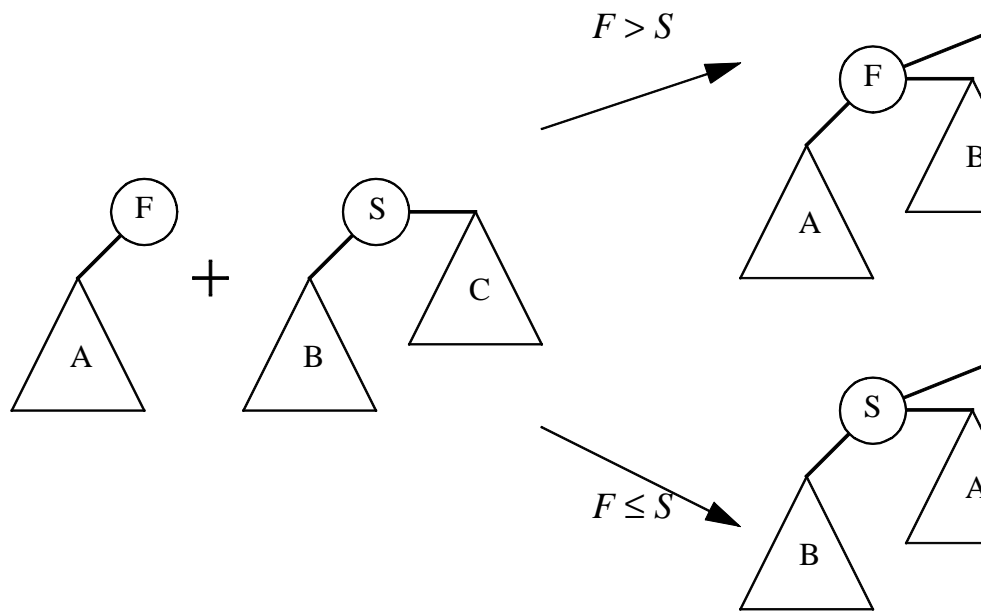
Abstract representation of sample pairing heap



Actual representation of above pairing heap; dark line represents a pair of pointers 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

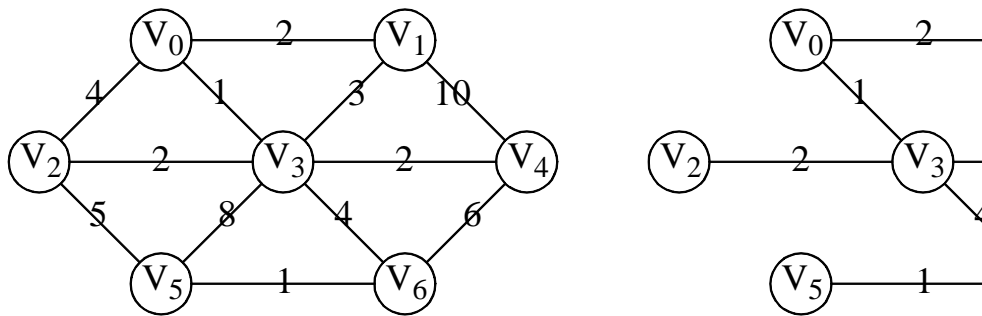
Chapter 23

The Disjoint Set Class

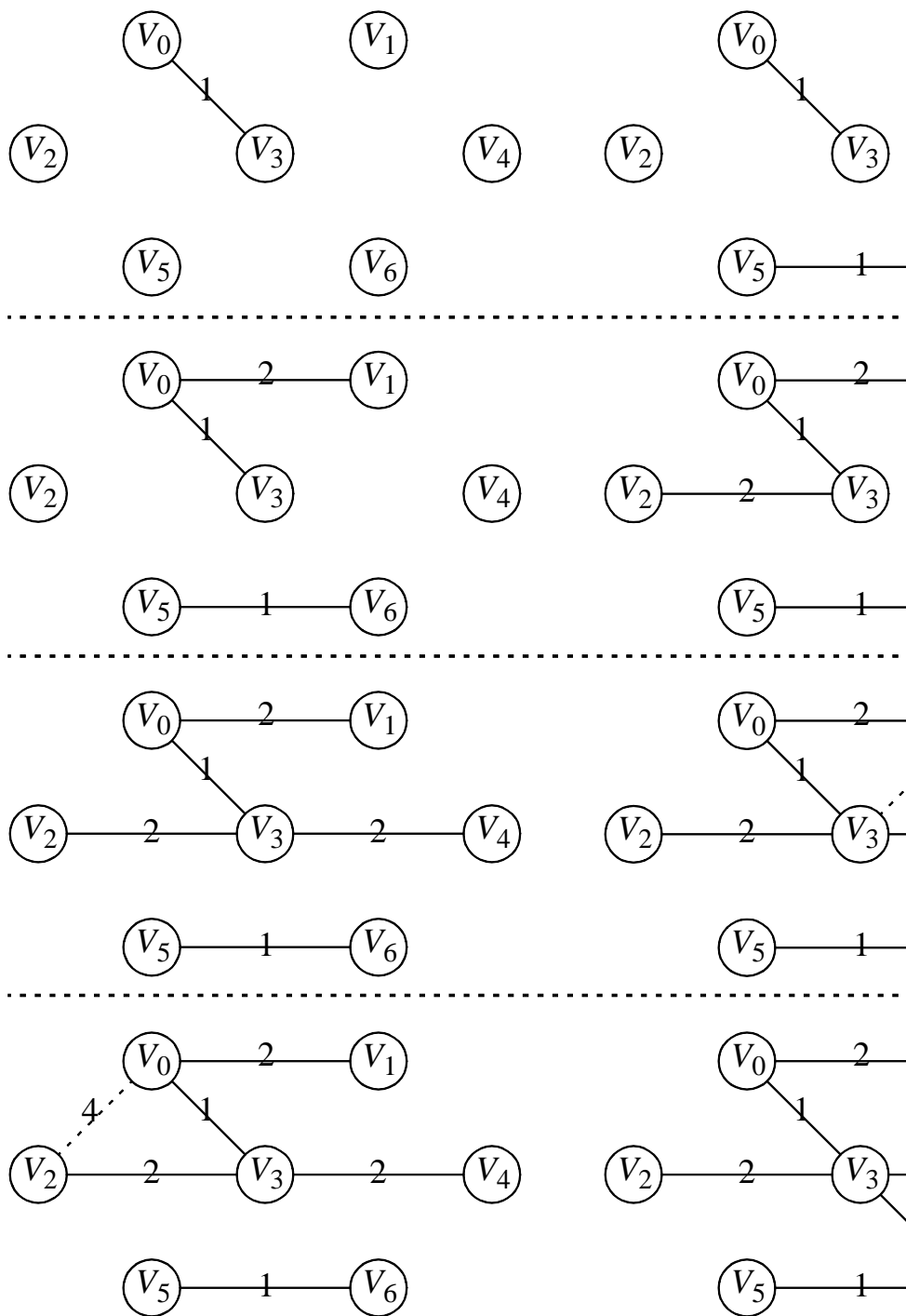
A *relation* R is defined on a set S if for every pair of elements (a, b) , $a, b \in S$, $a R b$ is either true or false. If $a R b$ is true, then we say that a is related to b . An *equivalence relation* is a relation R that satisfies three properties:

- *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 $a R c$

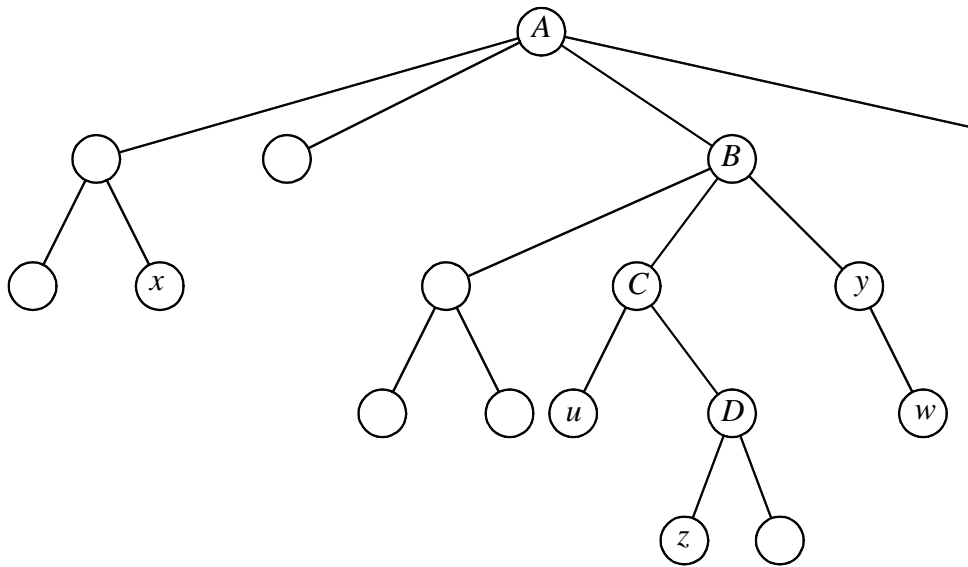
Definition of equivalence relation



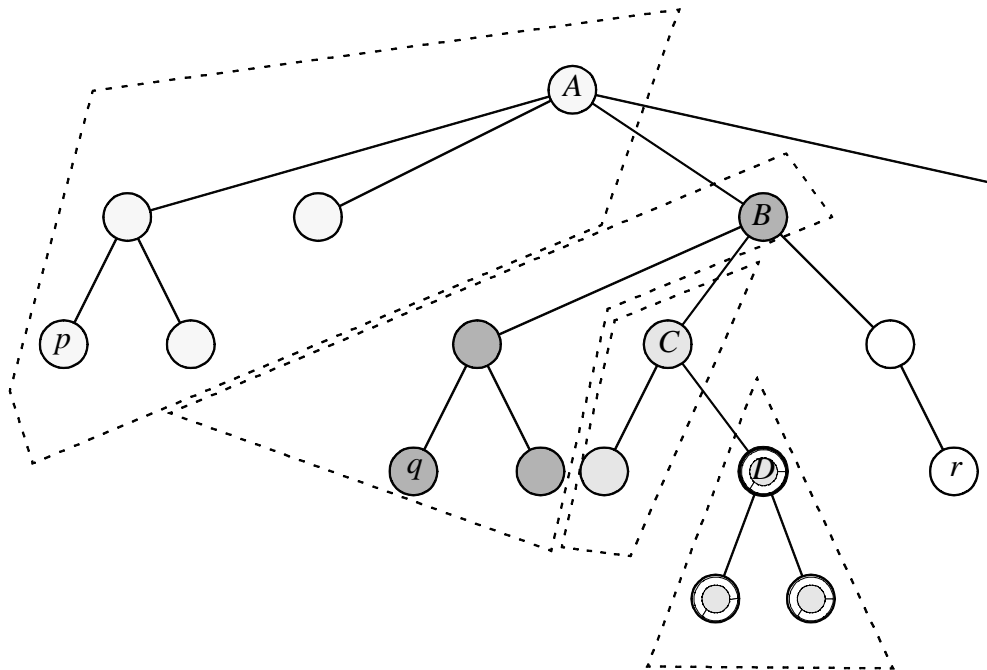
A graph G (left) and its minimum spanning tree



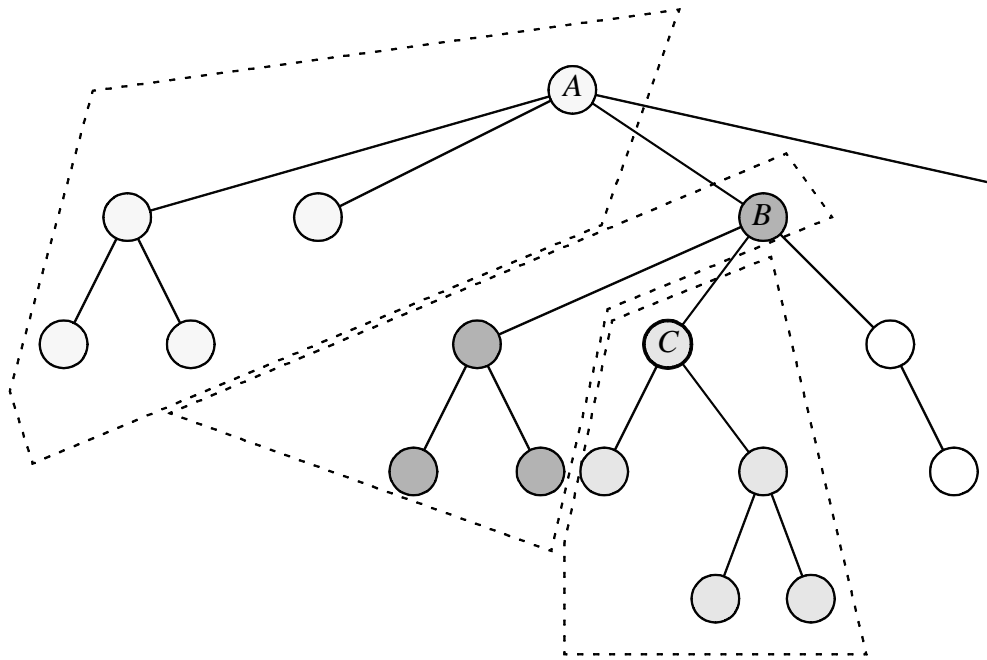
Kruskal's algorithm after each edge is considered



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



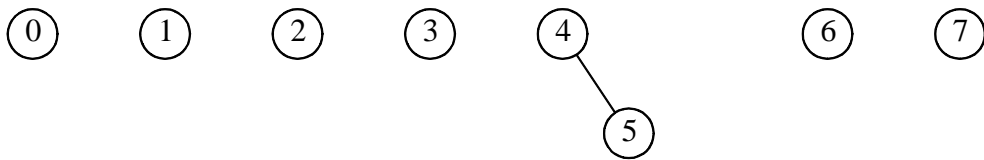
The sets immediately prior to the return from the recursive call to D ; D is marked as visited and $NCA(D, v)$ is v 's anchor to the current path



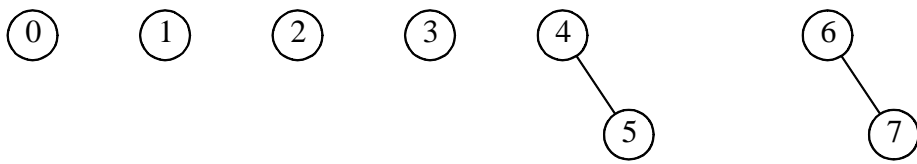
After the recursive call from D returns, we merge the set anchored by D into the set anchored by C and then compute all $NCA(C, v)$ for nodes v that are marked prior to completing C 's recursive call



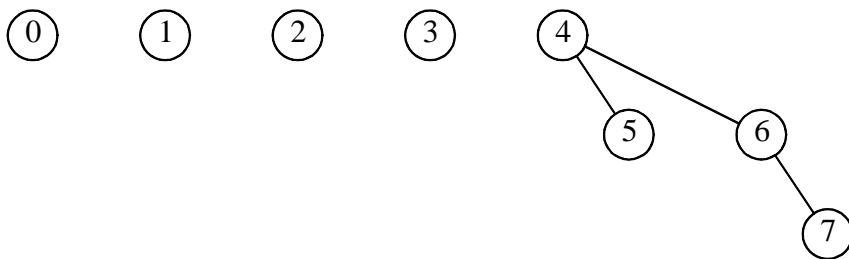
Forest and its eight elements, initially in different sets



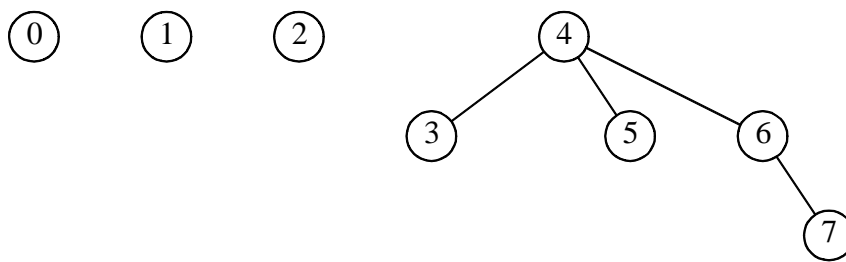
Forest after Union of trees with roots 4 and 5



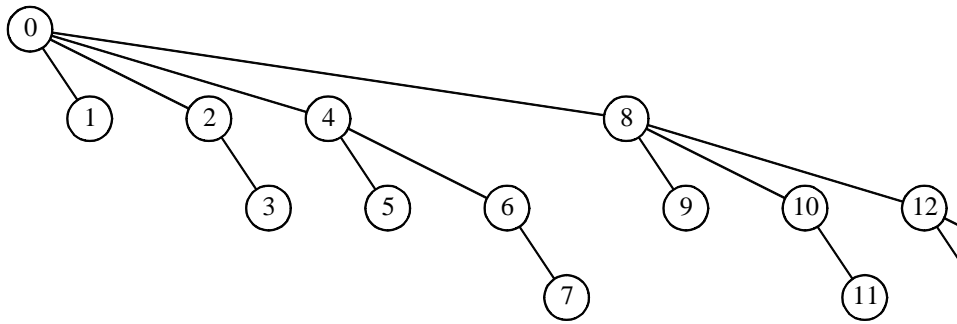
Forest after Union of trees with roots 6 and 7



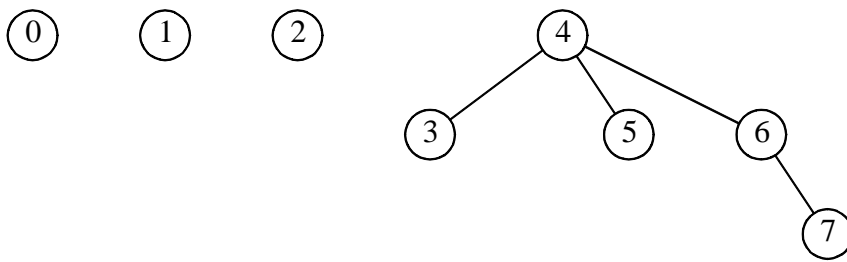
Forest after Union of trees with roots 4 and 6



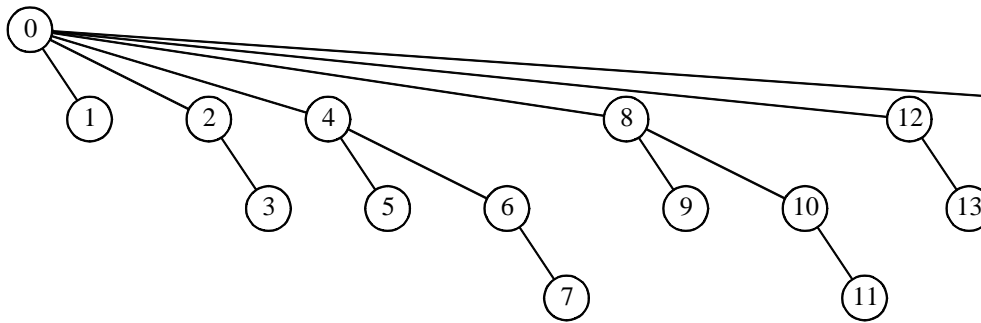
Forest formed by union-by-size, with size encoded as a negative number



Worst-case tree for $N=16$



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

Ackerman's function is defined as:

$$\begin{aligned} A(i, 0) &= 2^i & i \geq 0 \\ A(i, j) &= A(i-1, 1) & i \geq 1, j = 0 \\ A(i, j) &= A(i, A(i-1, j-1)) & i, j \geq 1 \end{aligned}$$

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

$$\alpha(n) = \min\{i \mid A(i, i) \geq n\}$$

Ackerman'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