The Priority Queue ADT

- A priority queue ADT is
  - Similar to a queue, except that
  - Items are removed from the queue in priority order.
- If lower-numbered items have higher priority, then the operations on a priority queue are:
  - **Insert:** Enqueue an item.
  - **Delete minimum:** Find and remove the minimum-valued (highest priority) item from the queue.

Priority Queue Implementation

- Unsorted list
  - Insert: Insert at the end of the list.
  - Delete minimum: Scan the list to find the minimum.
- Sorted list
  - Insert: Insert in the proper position to maintain order.
  - Delete minimum: Delete from the head of the list.
- Binary tree
  - Inserts and deletes take $O(\log N)$ time on average.
- Binary heap
  - Inserts and deletes take $O(\log N)$ worst-case time.
  - No links required!

Binary Heap

- A binary heap (or just heap) is a binary tree that is complete.
  - All levels of the tree are full except possibly for the bottom level which is filled from left to right:

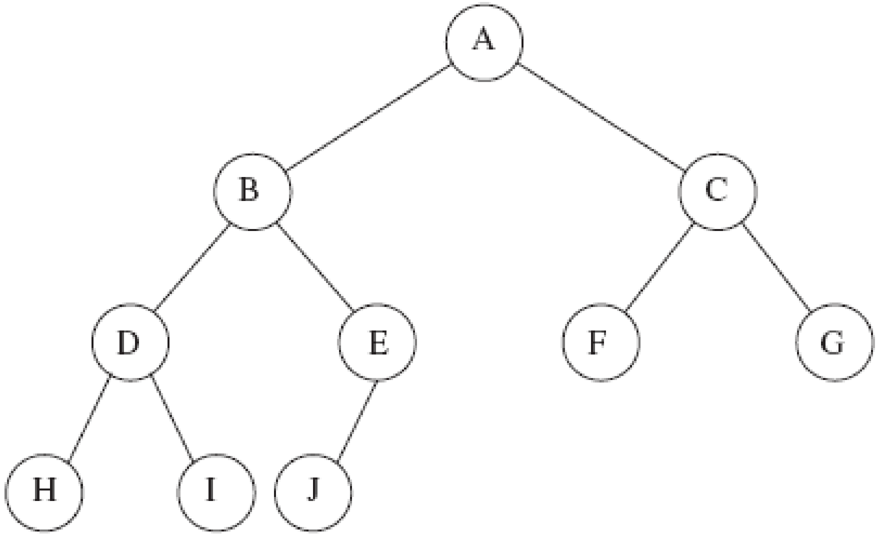
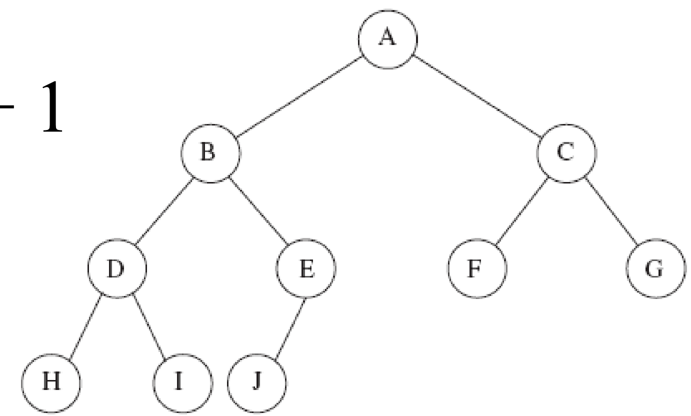


Figure 6.2 A complete binary tree

Data Structures and Algorithms in Java, 3rd ed.
by Mark Allen Weiss
Pearson Education, Inc., 2012

Binary Heap

- Conceptually, a heap is a binary tree.
- But we can implement it as an array.
- For any element in array position i:
  - Left child is at position $2i$
  - Right child is at position $2i + 1$
  - Parent is at position $\lfloor i/2 \rfloor$



	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Figure 6.3 Array implementation of complete binary tree

Data Structures and Algorithms in Java, 3rd ed.
by Mark Allen Weiss
Pearson Education, Inc., 2012

Heap-Order Priority

- We want to find the minimum value (highest priority) value quickly.
- Make the minimum value always at the root.
  - Apply this rule also to roots of subtrees.
- Weaker rule than for a binary search tree.
  - Not necessary that values in the left subtree be less than the root value and values in the right subtree be greater than the root value.

Heap-Order Priority

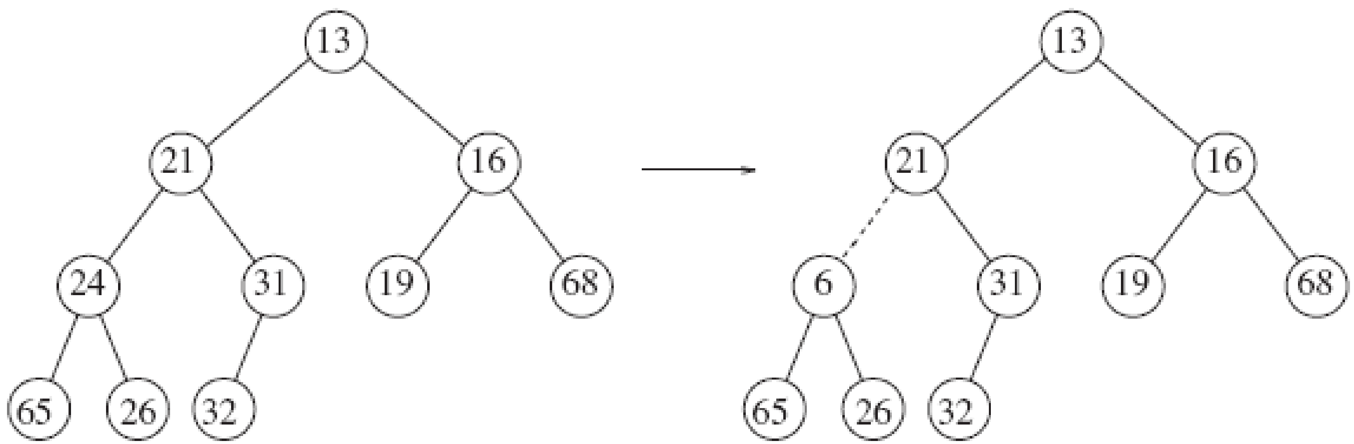


Figure 6.5 Two complete trees (only the left tree is a heap)

Heap Insertion

- Create a hole in the next available position at the bottom of the (conceptual) binary tree.
  - The tree must remain complete.
  - The hole is at the end of the implementation array.
- While the heap order is violated:
  - Slide the hole's parent into the hole.
  - "Bubble up" the hole towards the root.
  - The new value percolates up to its correct position.
- Insert the new value into the correct position.

Heap Insertion

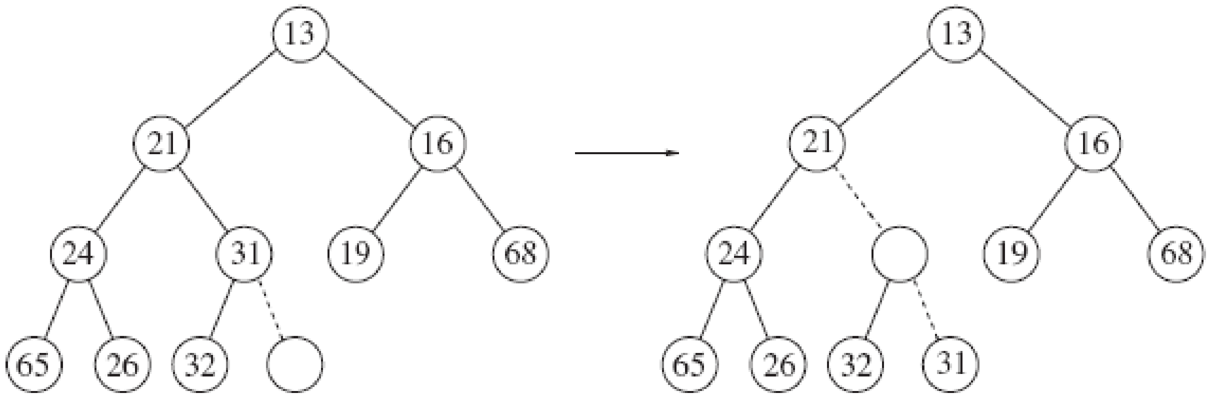


Figure 6.6 Attempt to insert 14: creating the hole and bubbling the hole up

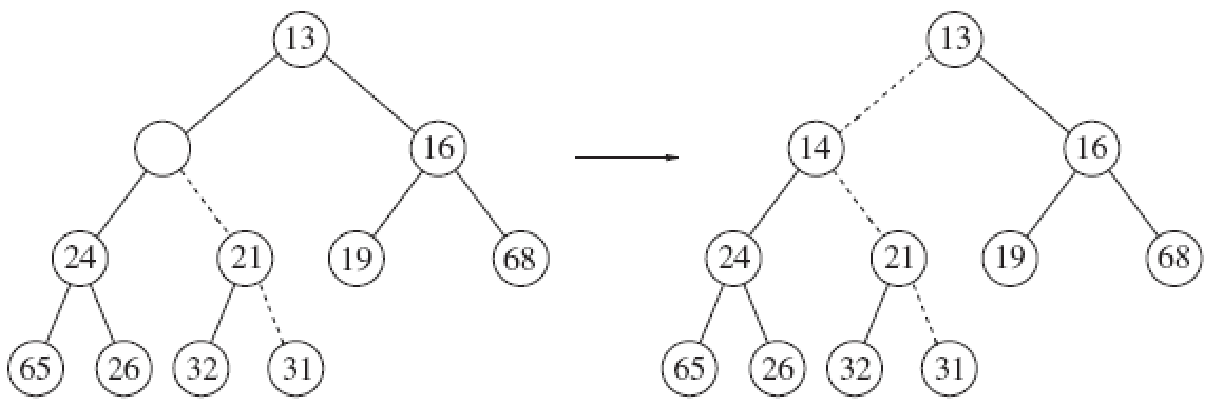


Figure 6.7 The remaining two steps to insert 14 in previous heap

Heap Insertion

```
public void insert(AnyType x)
{
    if (currentSize == array.length - 1) {
        enlargeArray(array.length*2 + 1);
    }

    // Percolate up.
    int hole = ++currentSize;
    for (array[0] = x; x.compareTo(array[hole/2]) < 0; hole
/= 2) {
        array[hole] = array[hole/2];
    }
    array[hole] = x;
}
```

Heap Deletion

- Delete the root node of the (conceptual) tree.
  - A hole is created at the root.
  - The tree must remain complete.
  - Put the last node of the heap into the hole.
- While the heap order is violated:
  - The hole percolates down.
  - The last node moves into the hole at the correct position.

Heap Deletion

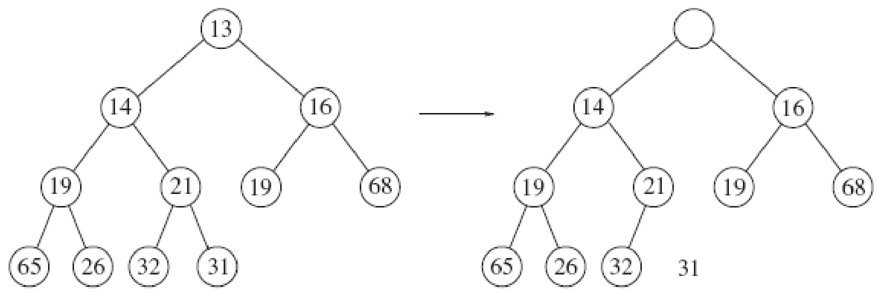


Figure 6.9 Creation of the hole at the root

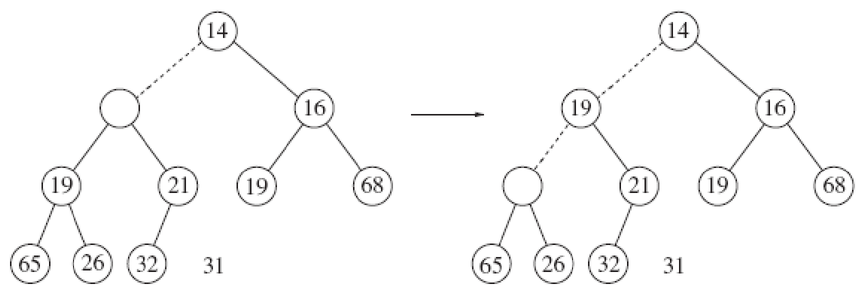


Figure 6.10 Next two steps in `deleteMin`

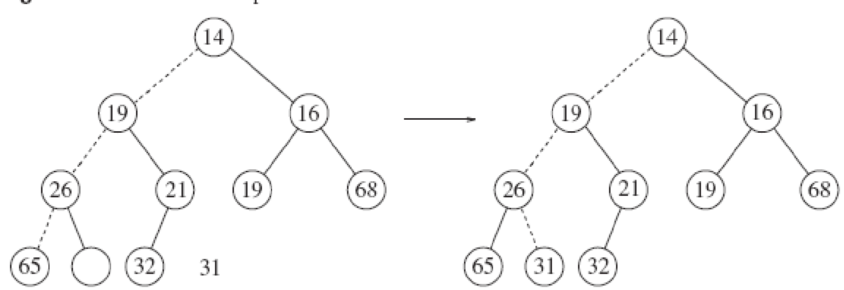


Figure 6.11 Last two steps in `deleteMin`

Data Structures and Algorithms in Java, 3rd ed.
by Mark Allen Weiss
Pearson Education, Inc., 2012

Heap Deletion

```
public AnyType deleteMin() throws Exception
{
    if (isEmpty()) throw new Exception();

    AnyType minItem = findMin();
    array[1] = array[currentSize

    percolateDown(1);
    return minItem;
}
```

It's the root node.

ast value temporarily into the root.

Heap Deletion

```
private void percolateDown(int hole)
{
    int child;
    AnyType tmp = array[hole];

    for (; hole*2 <= currentSize; hole = child) {
        child = hole*2;
        if (    (child != currentSize)
            && (array[child + 1].compareTo(array[child]))
< 0) {
            child++;
        }

        if (array[child].compareTo(tmp) <
            array[hole] = array[child];
        }
        else {
            break;
        }
    }
    array[hole] = tmp;
```

Percolate the root hole down.

Does the last value fit?

Heap Animation

```
appletviewer Chap12/Heap/Heap.html
```