香港中文大學（深圳）
The Chinese University of Hong Kong

CSC3100 Data Structures
Lecture 17: Heap

Yixiang Fang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

Outline

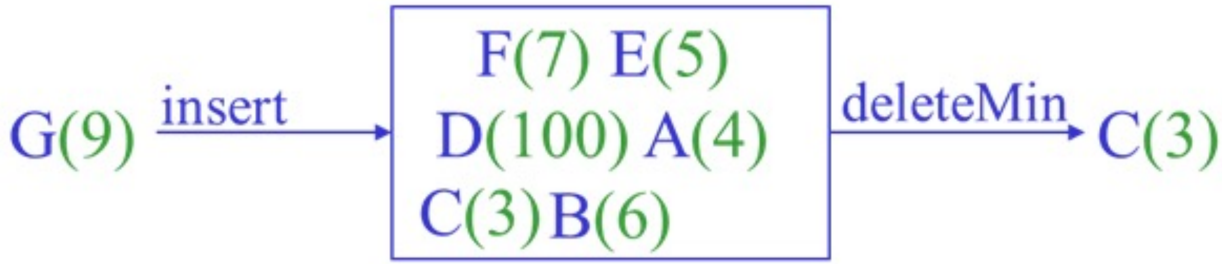
- Heap
  - Motivation
  - Priority queue
  - Binary heap
- Insert & delete & build
- HeapSort

Motivation

- Have you ever been jammed by a huge job while you are waiting for just one-page printout ?
  - This is a typical situation for a first-in first-out (FIFO) queue
- Other applications
  - Scheduling CPU jobs
  - Emergency room admission processing
- Practical requirements
  - Short jobs may go first
  - Most urgent cases should go first
  - Task with highest priority/lowest priority should go first

Priority queue ADT

- Priority queue operations
  - insert
  - deleteMin
  - create
  - isEmpty
  - …



- Priority queue property:
  - For two elements in the queue, *x* and *y*, if *x* has a *lower priority* value than *y*, *x* will be deleted before *y*

Simple implementations

- Multiple possibilities for the implementation
  - Singly linked list (Suggestion 1)
    - Insert at the front in $O(1)$
    - Delete minimum in $O(N)$
  - Sorted array (Suggestion 2)
    - Insert in $O(N)$
    - Delete minimum in $O(N)$
  - Binary heap (Suggestion 3)
    - Insert in $O(\log N)$
    - Delete minimum in $O(\log N)$
    - Two properties: structure property & heap order property

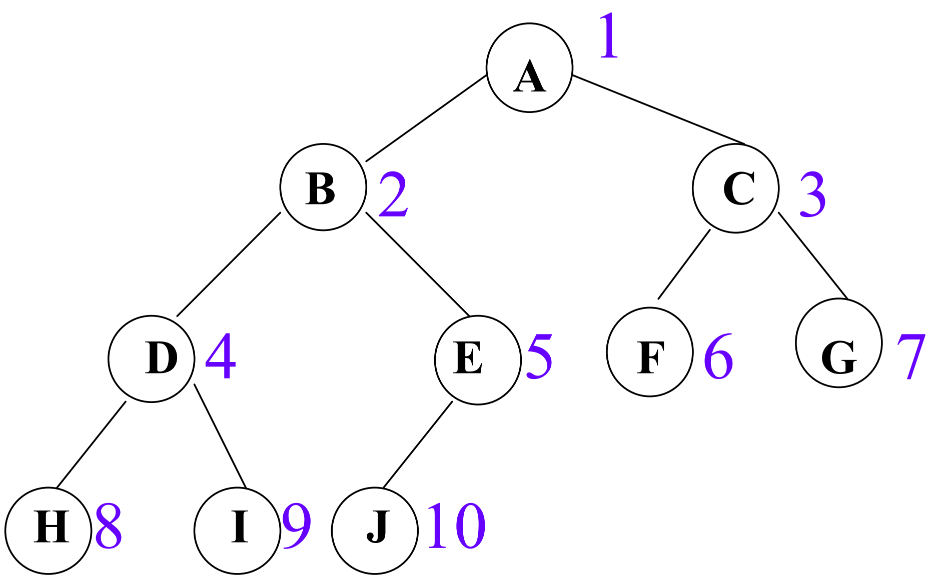
Binary heap

- (A) Structure property
  - A heap is a complete binary tree
    - A binary tree that is completely filled, except at the bottom level, which is filled from left to right
  - A complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes
  - The height of a complete binary tree = $\lfloor \log N \rfloor$
    - round down, e.g., $\lfloor 2.7 \rfloor = 2$

Binary heap: example

A complete binary tree can be represented in an array

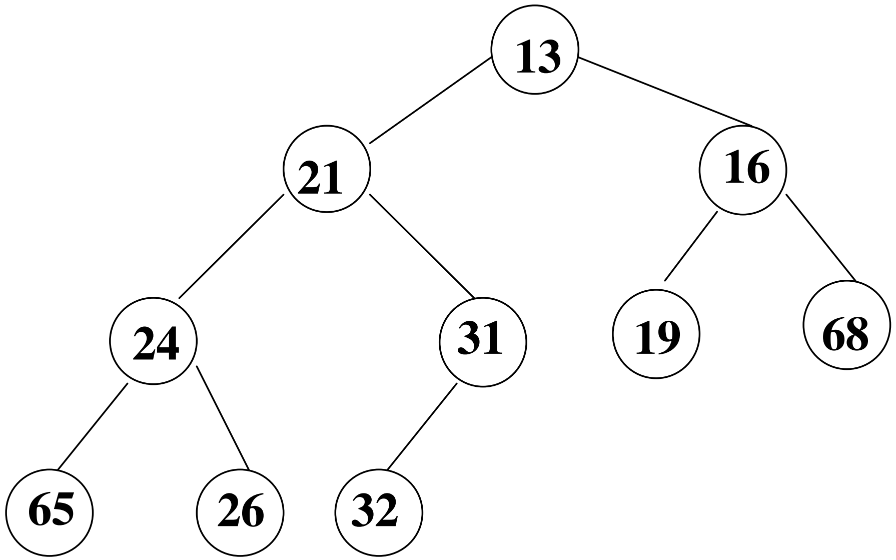
	A	B	C	D	E	F	G	H	I	J					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

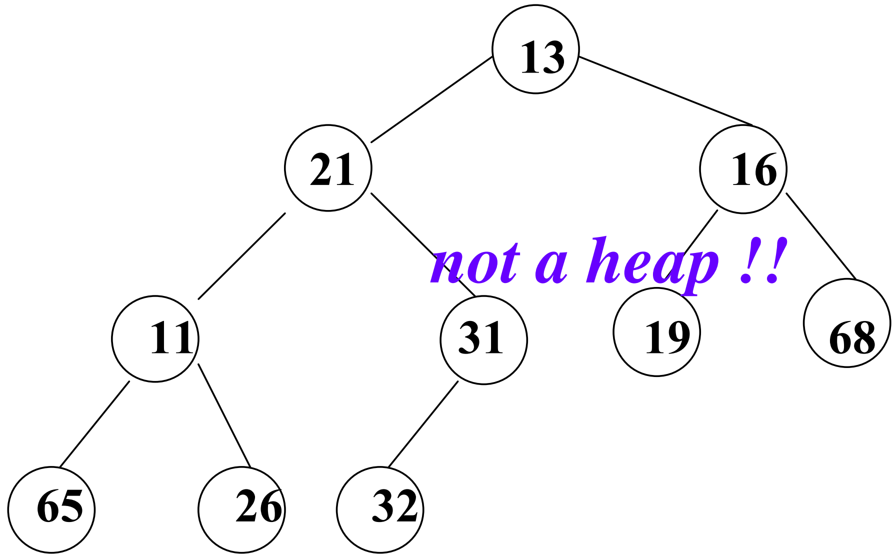


- The root is at position 1 (reserve position 0 for the implementation purpose)
- For an element at position i,
  - its left child is at position $2i$
  - its right child at $2i+1$; its parent is at floor $\lfloor i/2 \rfloor$

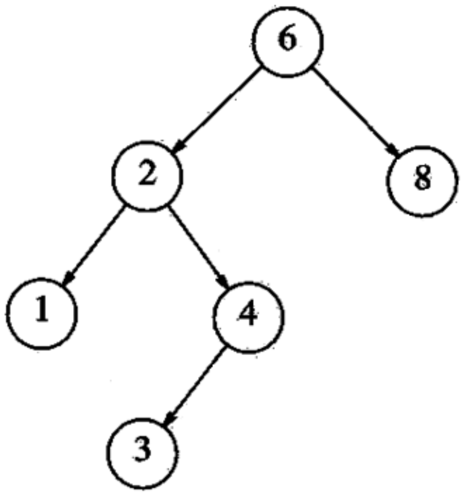
Binary heap

- (B) Heap order property
  - The value at any node should be smaller than (or equal to) all of its descendants (guarantee that the node with the minimum value is at the root)

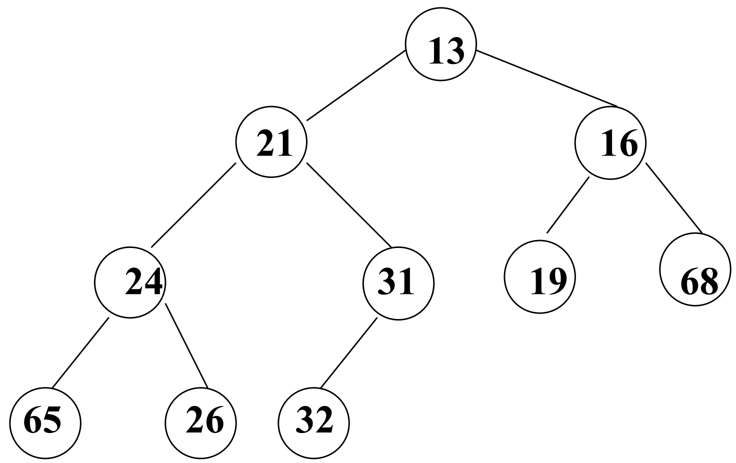




BST vs heap



A binary search tree



A binary heap

Notice the difference in node ordering!!

Binary heap

- Class skeleton for **Elements**

```
class ElementType {
    int priority;
    String data;

    public ElementType(int priority, String data) {
        this.priority = priority;
        this.data = data;
    }

    public boolean isHigherPriorityThan(ElementType e) {
        return priority < e.priority;
    }
}
```

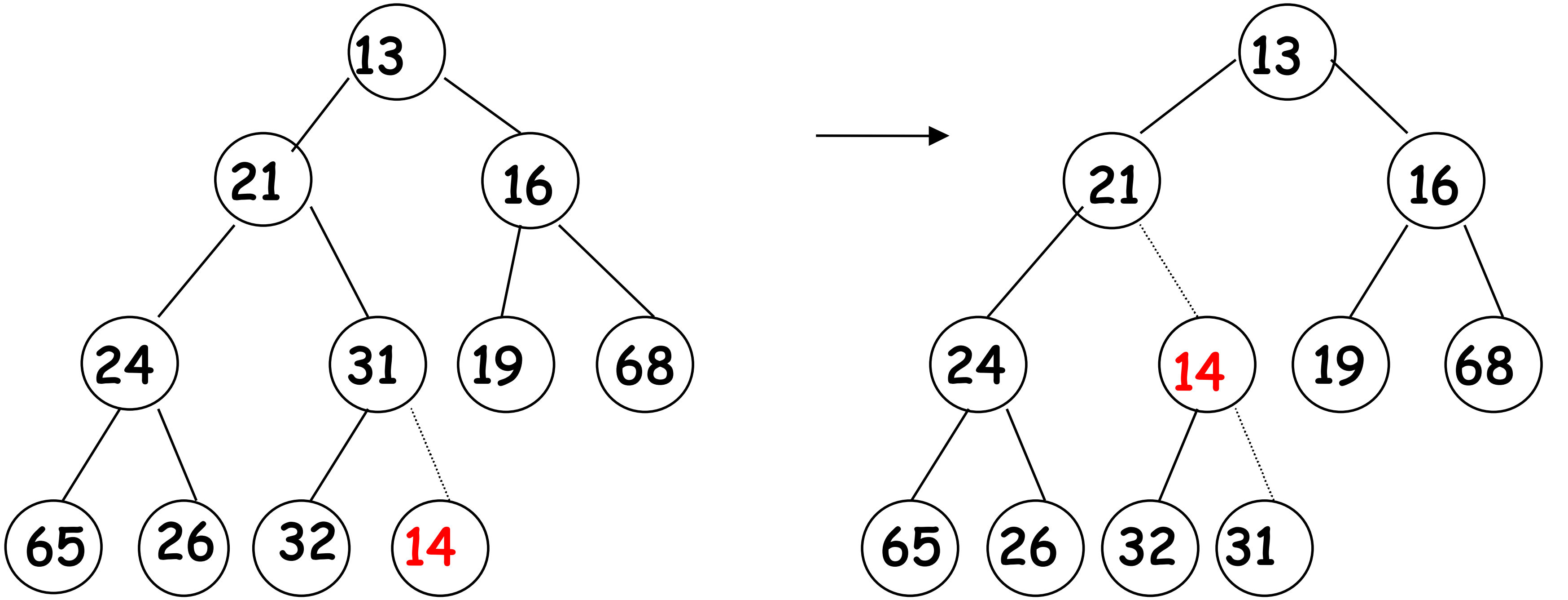
Binary heap

- Definition and constructor of priority queue

```
public class BinaryHeap {
        private int currentSize;      // Number of elements in heap
        private ElementType arr[ ]; // The heap array

        public BinaryHeap (int capacity) {
                currentSize = 0;
                arr = new ElementType[capacity + 1];
        }
}
```

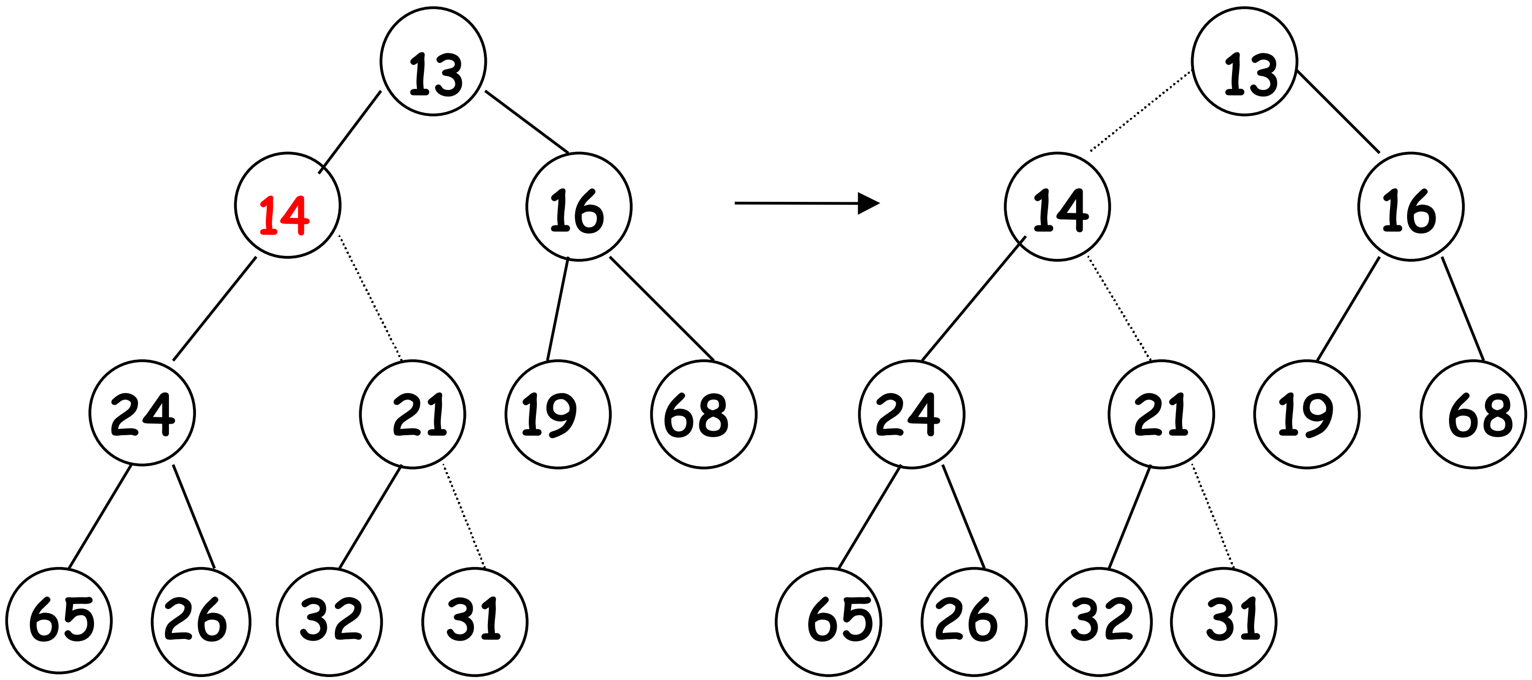
Binary heap: insert

Attempt to insert 14:
(1) creating the hole, and (2) bubbling the hole up



Binary heap: insert

The remaining two steps to insert 14 in previous heap



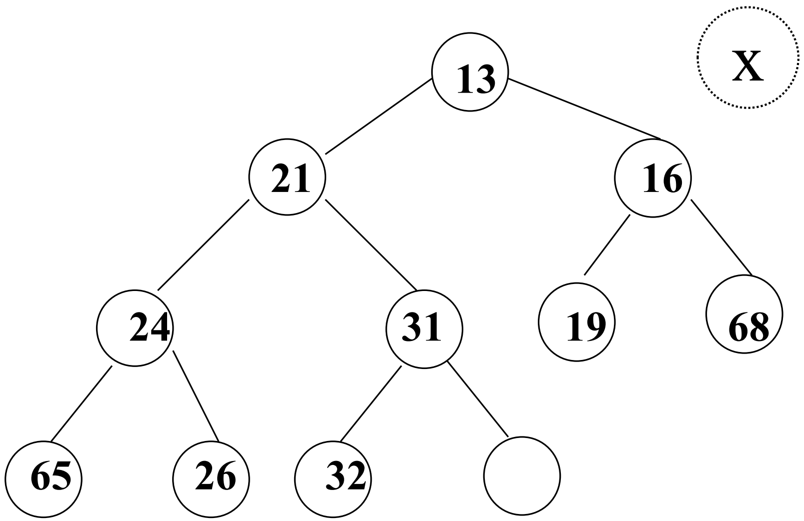
Binary heap: insert

- To insert an element X,
  - Create a hole in the next available location
  - If X can be placed in the hole without violating heap order, insertion is complete
  - Otherwise slide the element that is in the hole's parent node into the hole, i.e., bubbling the hole up towards the root
  - Continue this process until X can be placed in the hole (a *percolating up* process)

Attention!
Worst case running time is $O(\log N)$ - the new element is percolating up all the way to the root

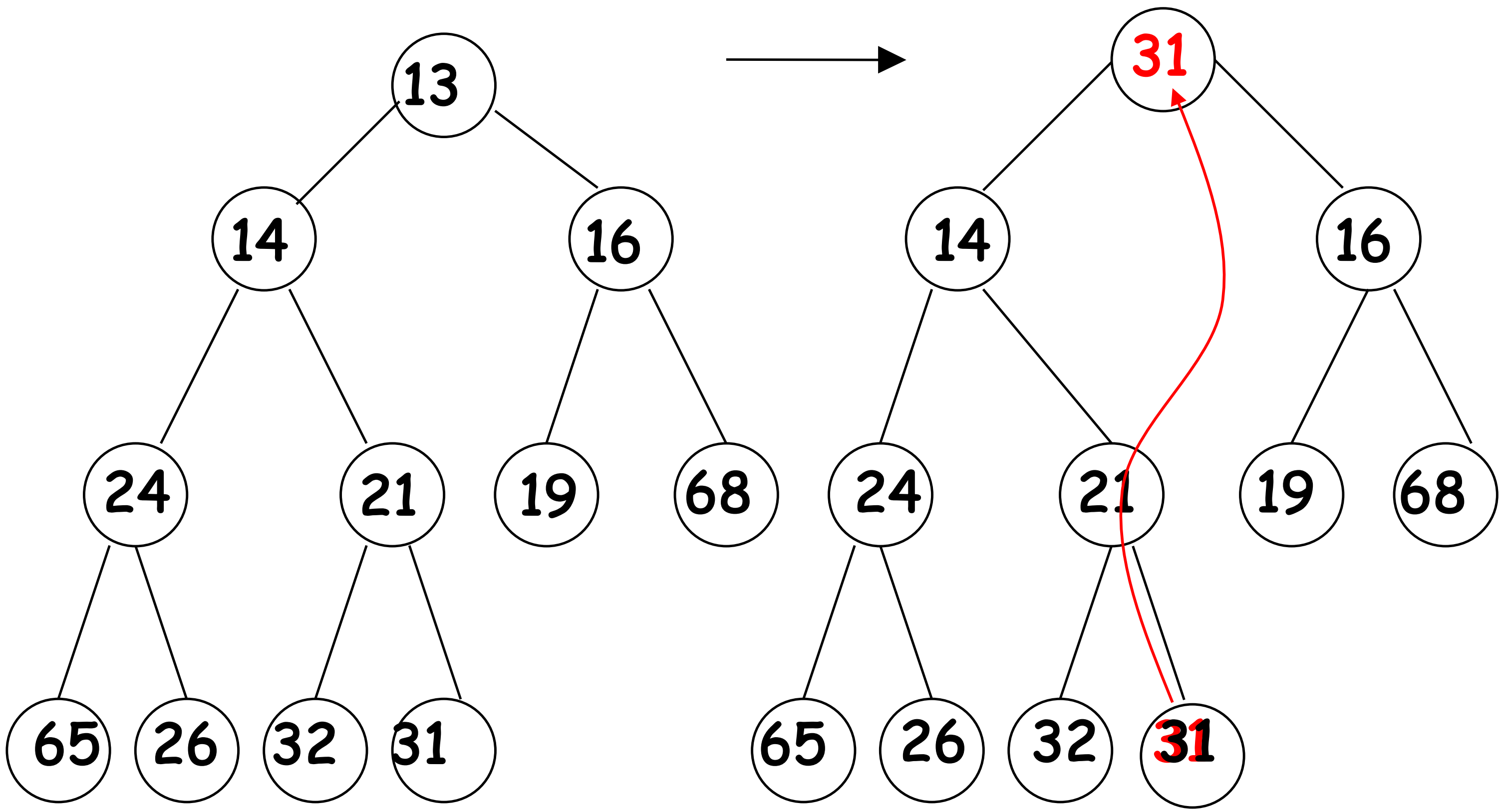
Binary heap: insert

```
public void insert(ElementType x) throws Exception {
        if (isFull())
                throw new Exception("Overflow");

        // Percolate up
        int hole = ++currentSize;
        while(hole > 1 && x.isHigherPriorityThan(arr[hole/2])) {
                arr[hole] = array[hole / 2];
                hole /= 2;
        }
        arr[hole] = x;
}
```



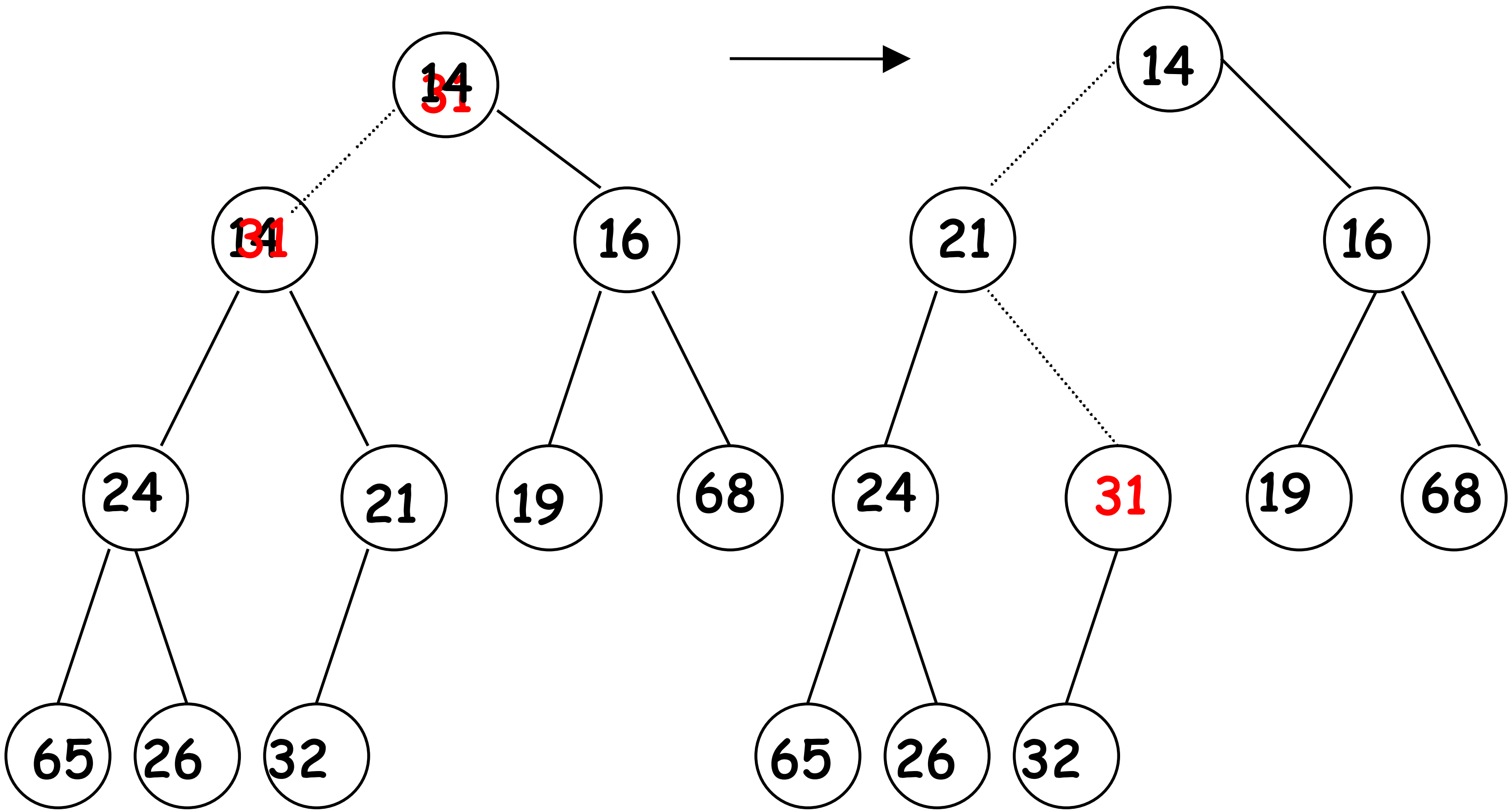
Binary heap: deleteMin

Creation of the hole at the root



Binary heap: deleteMin

Next two steps in DeleteMin



Binary heap: deleteMin

- The element at the root (position 1) is to be removed, and a hole is created
- Fill the root with the last node X
- Percolate X down (switch X with the smaller child) until the heap order property is satisfied

- Note that
  - Some node may have only one child (be careful when coding!)
  - Worst case running time is $O(\log N)$

Binary heap: deleteMin

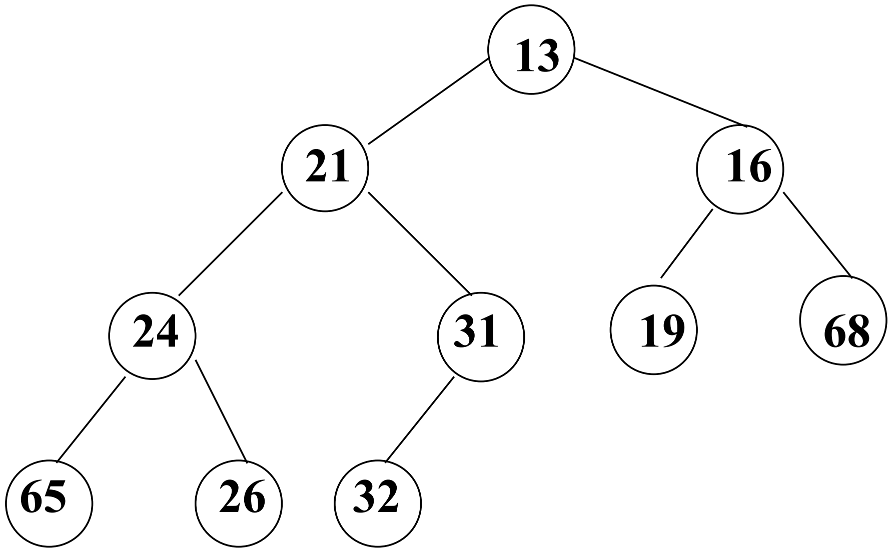
```
public String deleteMin() {
        if (isEmpty())
                return null;

        String data = arr[1].data;
        arr[1] = arr[currentSize--];

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

Binary heap: percolateDown

```
private void percolateDown(int hole) {
        int child;
        ElementType tmp = arr[hole];
        while (hole * 2 <= currentSize) {
                child = hole * 2;
                if (child != currentSize &&
                        arr[child +1].isHigherPriorityThan(arr[child]))
                        child++;
                if (arr[child].isHigherPriorityThan(tmp))
                        arr[hole] = array[child];
                else
                        break;
                hole = child;
        }
        arr[hole] = tmp;
}
```

Practice

- Given a binary heap as shown below, show the procedure of deletion on the heap step by step



Complexity analysis

- Given a heap with n elements
  - The height/depth of the heap is $O(\log n)$
    - Why?
  - During insertion/deletion, the worst-case time complexity depends linearly to the height/depth of the heap
- Heap insertion
  - $O(\log n)$
- Heap deletion
  - $O(\log n)$

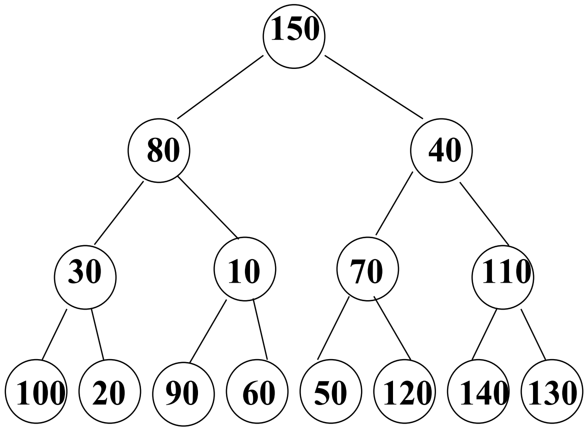
Binary heap construction

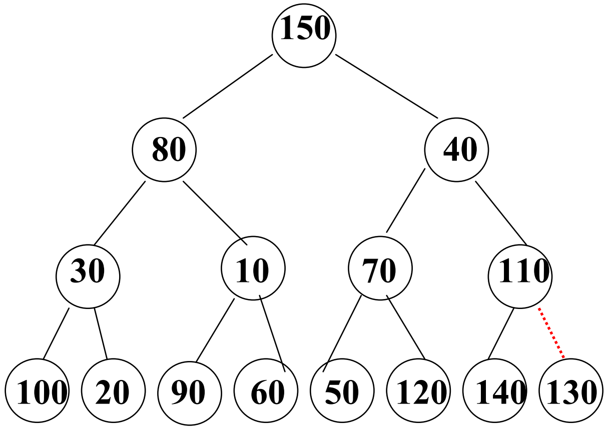
A naïve algorithm to build the binary heap is to repeatedly insert nodes one by one, which completes in O(nlogn) time

A faster algorithm to build the binary heap:

- *N* successive appends at the end of the array, each taking O (1), so the tree is unordered
- for (*i* = *N*/2; i > 0; i--)

  percolateDown (*i*);

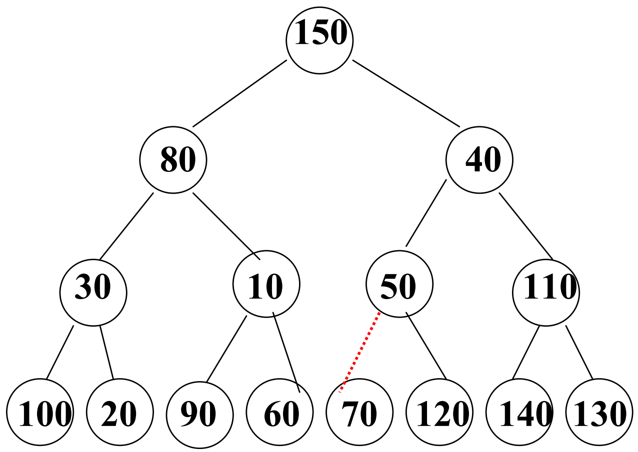




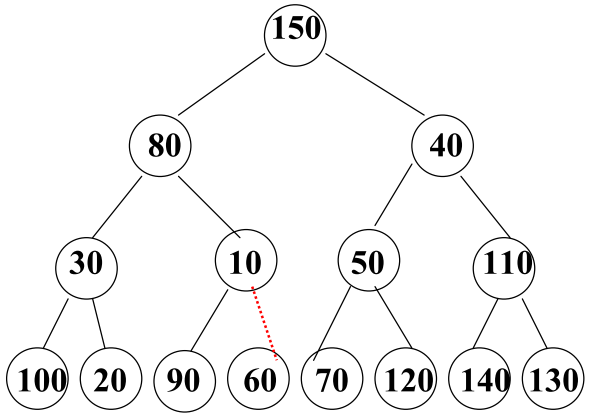
After *percolateDown* (7)

Note: Each dashed line corresponds to two comparisons: one to find the smaller child, and one to compare the smaller child with the node.

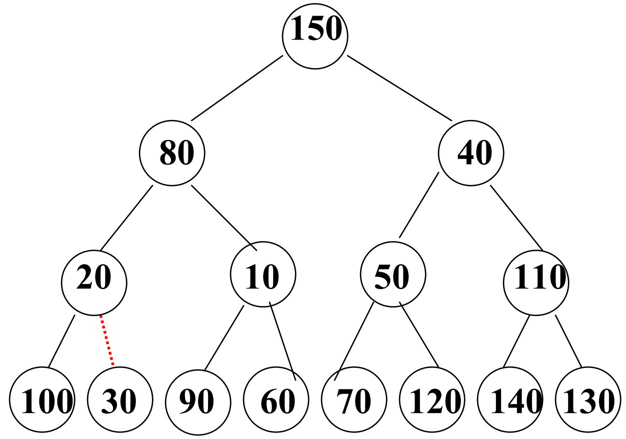
Binary heap construction



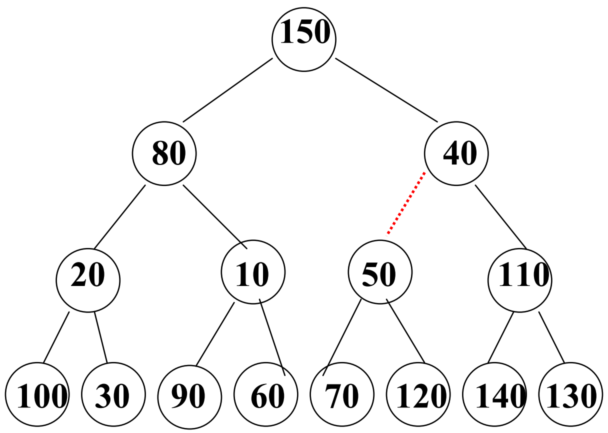
After *percolateDown* (6)



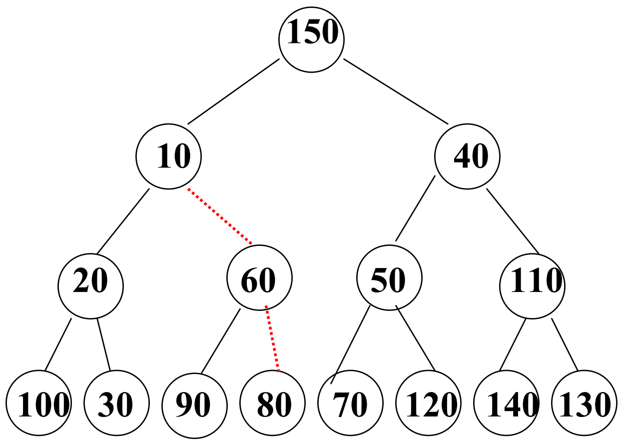
After *percolateDown* (5)



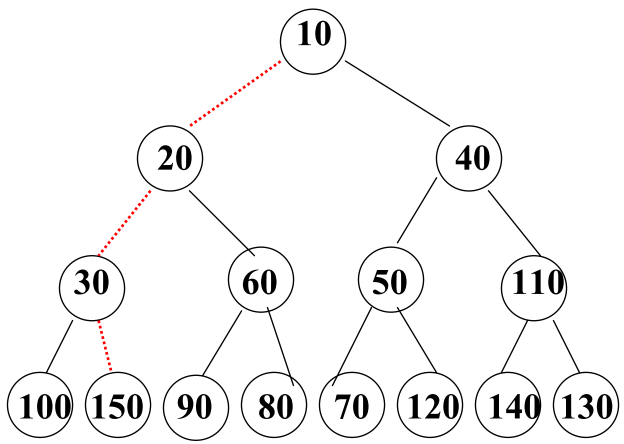
After *percolateDown* (4)



After *percolateDown* (3)



After *percolateDown* (2)



After *percolateDown* (1)

Complexity of buildHeap?

- Analysis
  - percolateDown for n/2 keys
  - Each key takes up to O(logn) cost
  - Thus, the total cost of BuildHeap is O(nlogn)

Is this upper bound tight?

- Notice
  - At most n/4 percolate down 1 level
    at most n/8 percolate down 2 levels
    at most n/16 percolate down 3 levels

    ...

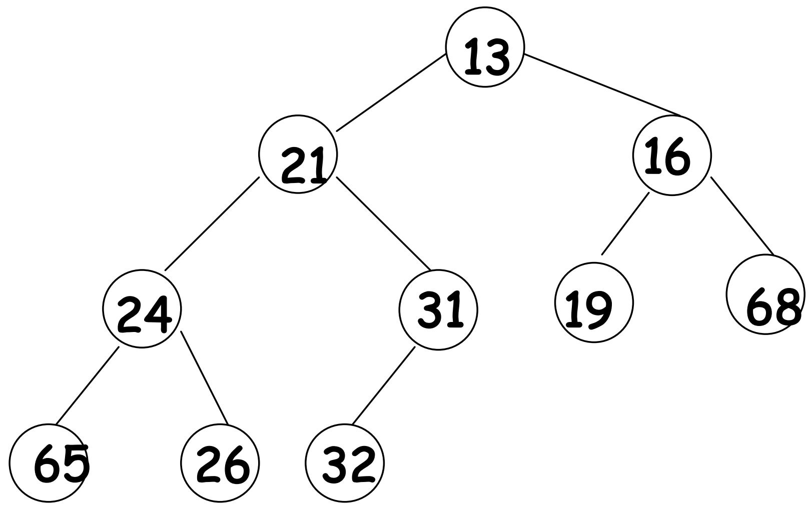
$$1\frac{n}{4}+2\frac{n}{8}+3\frac{n}{16}+\ldots=\sum_{i=1}^{\log n} i\frac{n}{2^{i+1}}=$$

$$\frac{n}{2}\sum_{i=1}^{\log n}\frac{i}{2^i}\approx\frac{n}{2}(2)=n$$

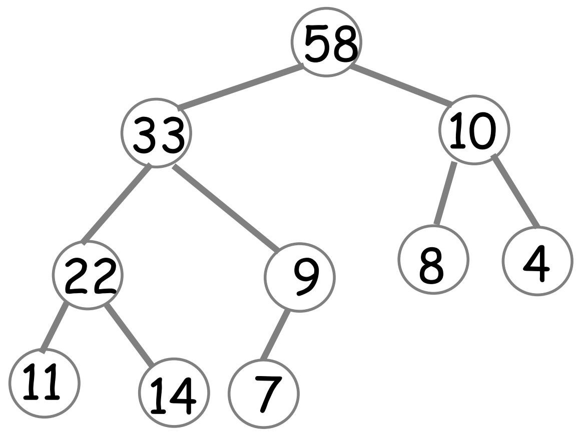
Conclusion: O(n)

Variants of heap

- Min-heap
  - The key present at the root node must be less than or equal among the keys present at all of its children
  - The same property must be recursively true for all sub-trees



- Max-heap
  - The key present at the root node must be larger than or equal among the keys present at all of its children
  - The same property must be recursively true for all sub-trees

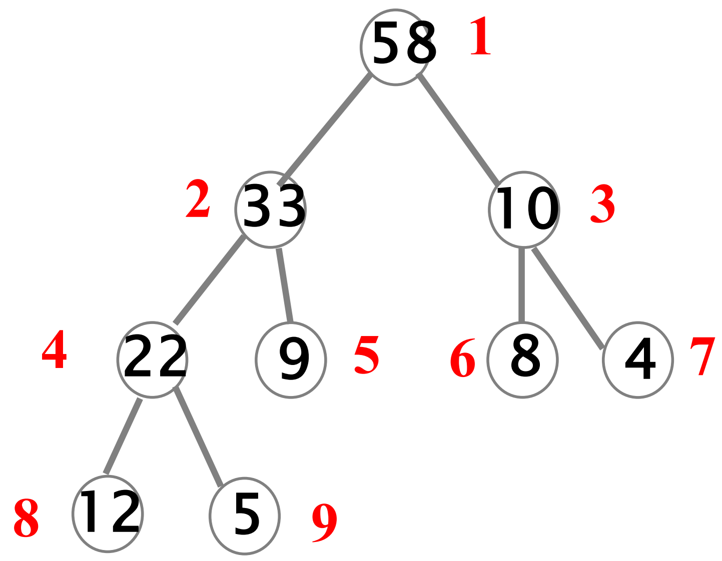


HeapSort

Complexity?

- Sorting using a max-heap
  - To sort an array arr, we first create a max-heap *H* with a capacity of arr.length+1
  - Then, we repeatedly delete from the max-heap until the max-heap becomes empty

4	5	8	9	10	12	22	33	58



10 classic sorting algorithms

Sorting algorithm	Stability	Time cost			Extra space cost
		Best	Average	Worst	
Bubble sort	√	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Insertion sort	√	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Selection sort	×	O(n)	$O(n^2)$	$O(n^2)$	O(1)
MergeSort	√	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
HeapSort	×	O(nlogn)	O(nlogn)	O(nlogn)	O(1)
QuickSort	×	O(nlogn)	O(nlogn)	$O(n^2)$	O(1)
ShellSort	×	O(n)	$O(n^{1.3})$	$O(n^2)$	O(1)
CountingSort	√	O(n+k)	O(n+k)	O(n+k)	O(k)
BucketSort	√	O(n)	O(n+k)	$O(n^2)$	O(k)
RadixSort	√	O(nk)	O(nk)	O(nk)	O(n)

Stable sorting: if two objects with equal keys appear in the same order in sorted output, as they appear in the input array

Exercise

- Sort an array $A[1 \ldots 8] = [4, 1, 3, 2, 16, 9, 10, 14]$ in ascending order by HeapSort
  - Show the contents of A in the sorting process step by step
- Write the codes of HeapSort

Recommended reading

- Reading
  - Chapters 6&12, textbook
- Next lecture
  - Hashing, Chapters 11.1-11.4 of textbook