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

CSC3100 Data Structures
Lecture 10: Heap, HeapSort

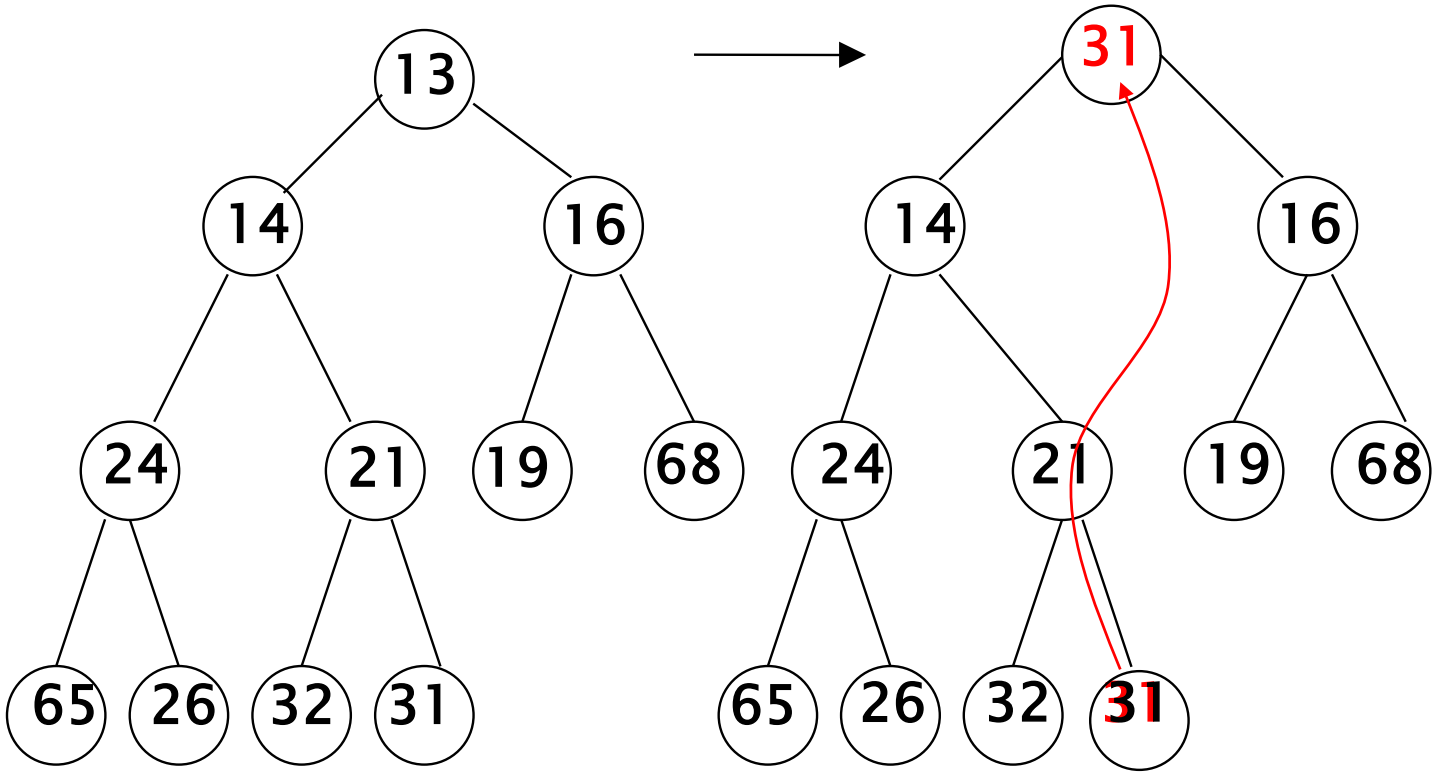
Yixiang Fang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

Outline

- Heap
  - Insert & delete
- HeapSort
  - Algorithm steps
  - Advantages
- Analysis of sorting algorithms
  - Comparison
  - A lower bound of time cost

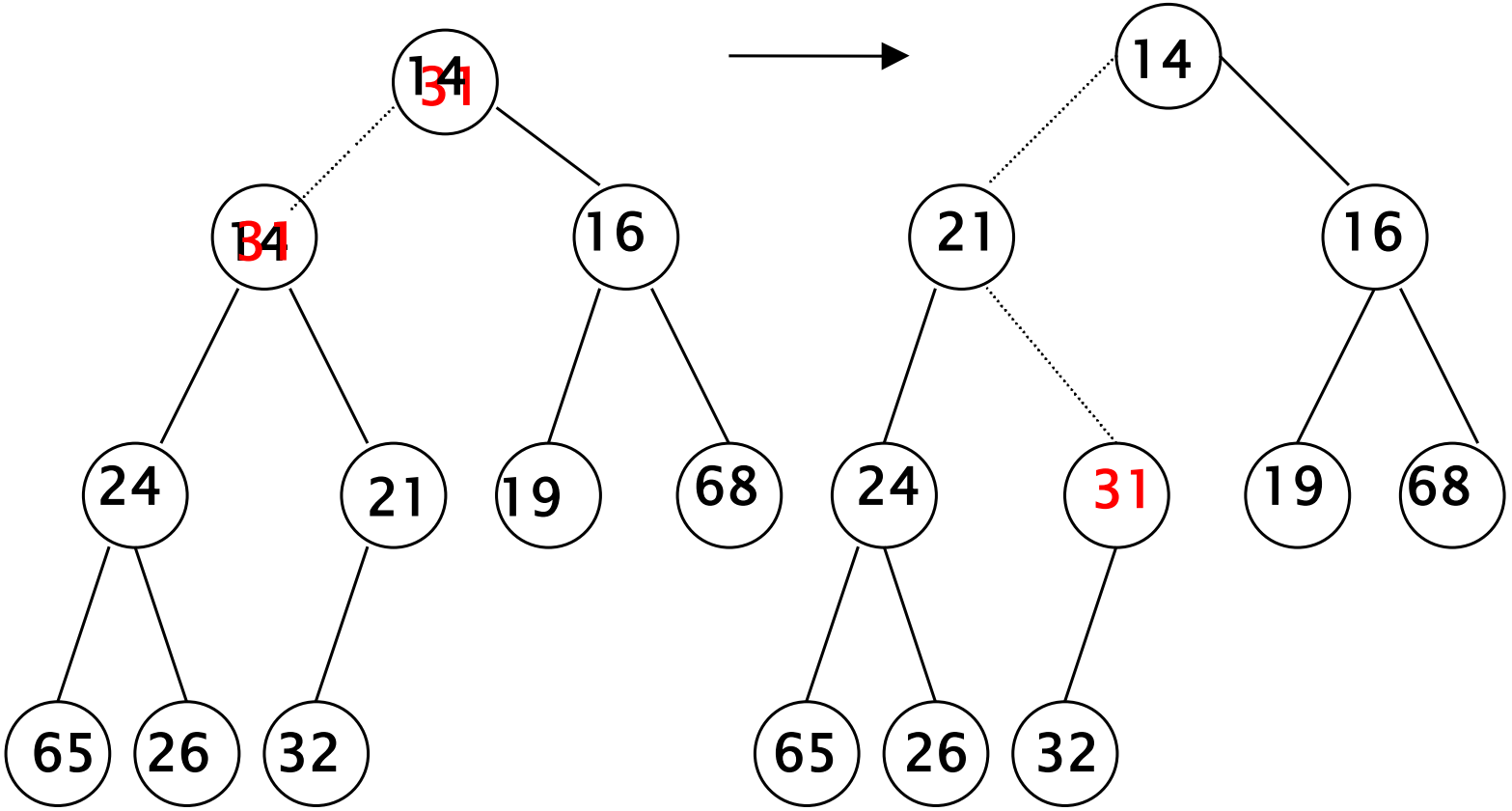
Binary Heap - DeleteMin

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

Binary Heap - DeleteMin

- 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

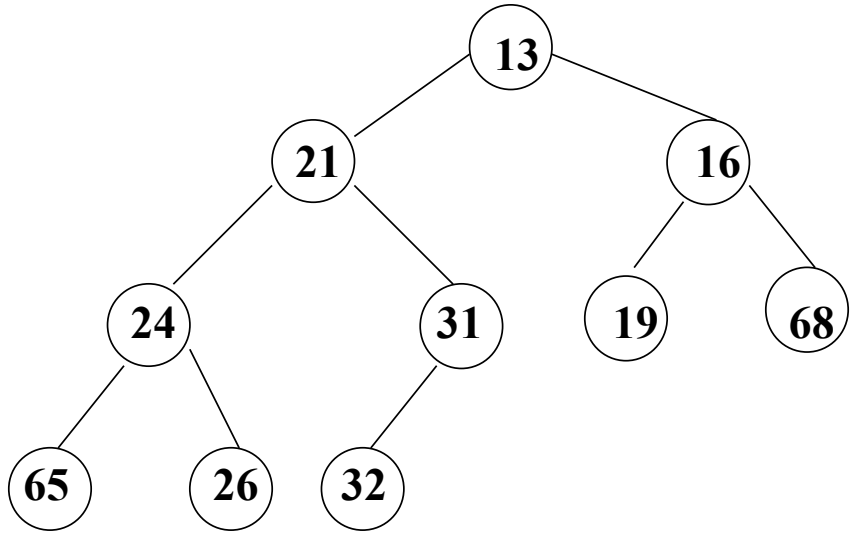
```
private void percolateDown(int hole) {
        int child;
        ElementType tmp = arr[hole];
        while (hole * 2 <= currentSize) {
                child = hole * 2;
                if (child != currentSize &&
                        arr[child +1].isHigherPriorityThan(arr[child]))
         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

Binary Heap - PercolateDown

```
            if (arr[child].isHigherPriorityThan(tmp))
                arr[hole] = array[child];
            else
                break;
            hole = child;
        }
        arr[hole] = tmp;
}
```

Practice

- Given a heap as shown below, show the procedure of heap delete operation 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 - Other Heap Operations

decreaseKey (P, Δ)

- Lower the key value at position P by Δ
- Fix the heap order by percolating up
- Application: advance the priority of a job

Binary Heap - Other Heap Operations

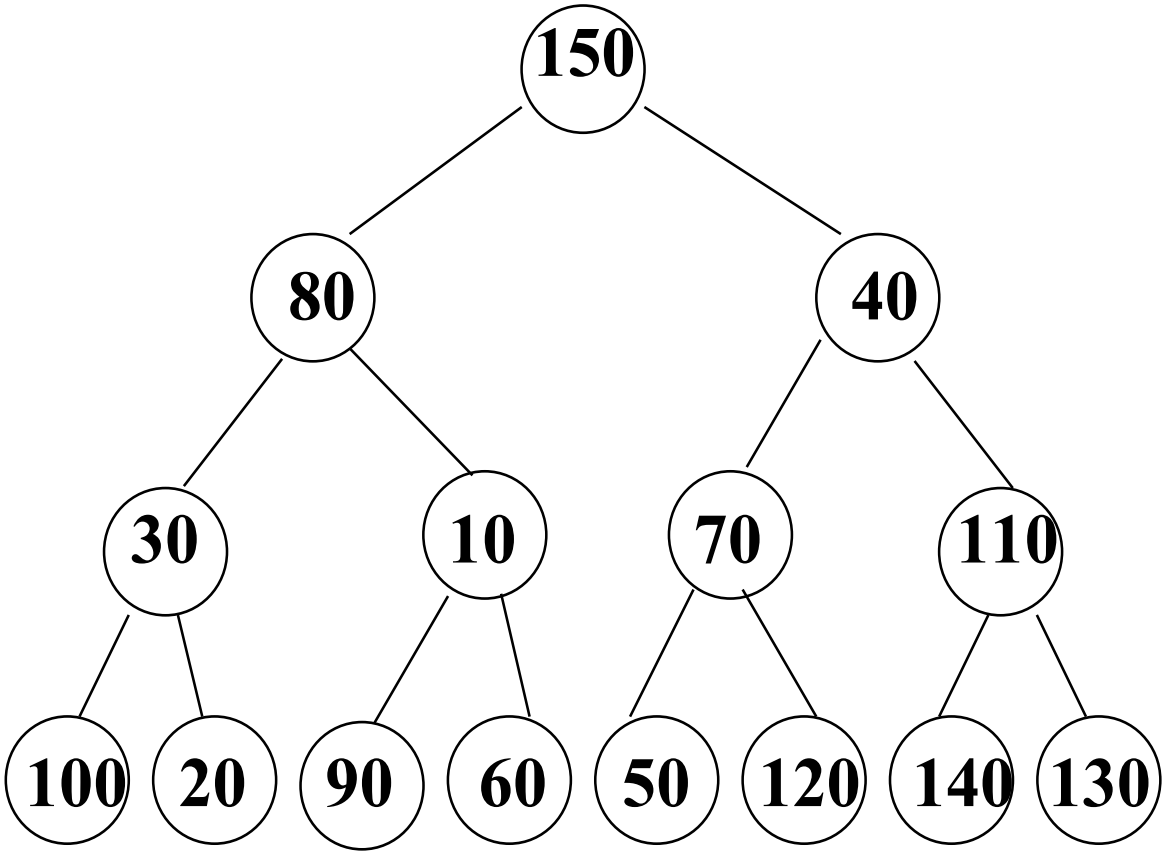
buildHeap ()

- *N* successive appends at the end of the array, each takes O (1). The tree is unordered.

- for (*i* = *N*/2; i > 0; i--)
  percolateDown (*i*);

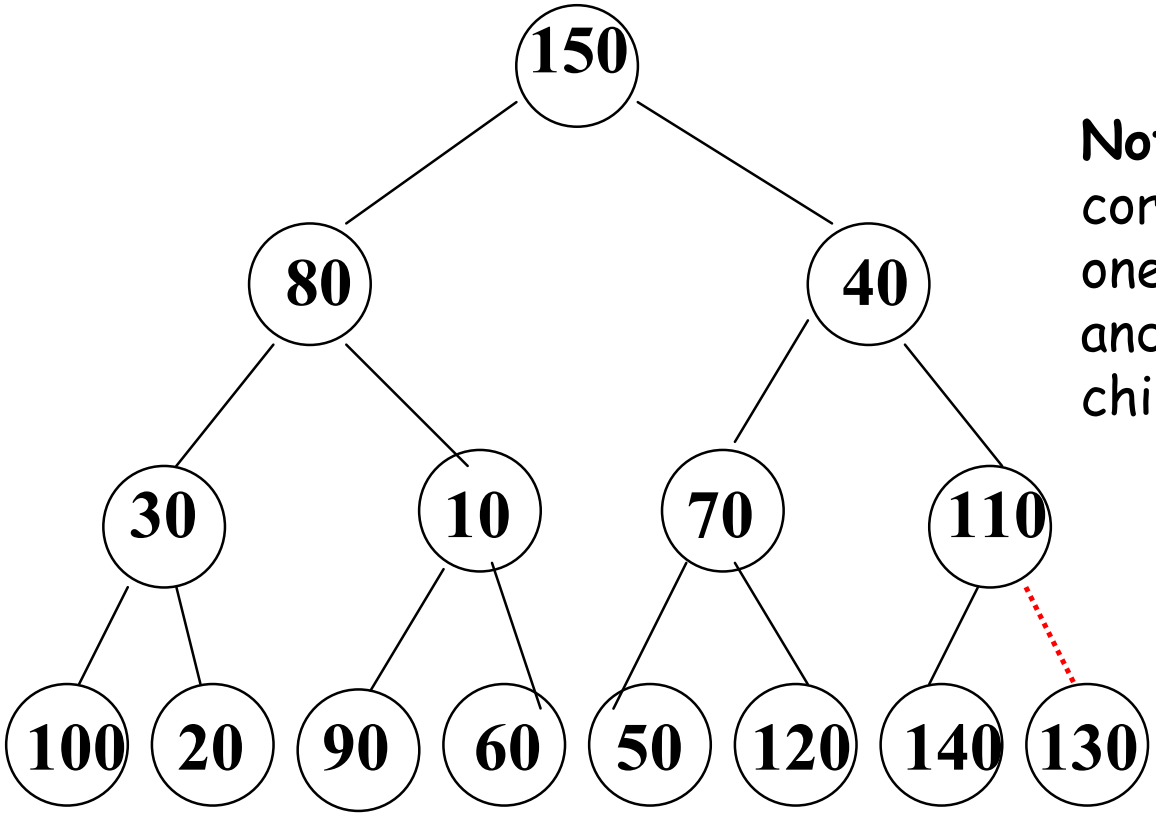
Binary Heap - Other Heap Operations

- Initial heap



Binary Heap - Other Heap Operations

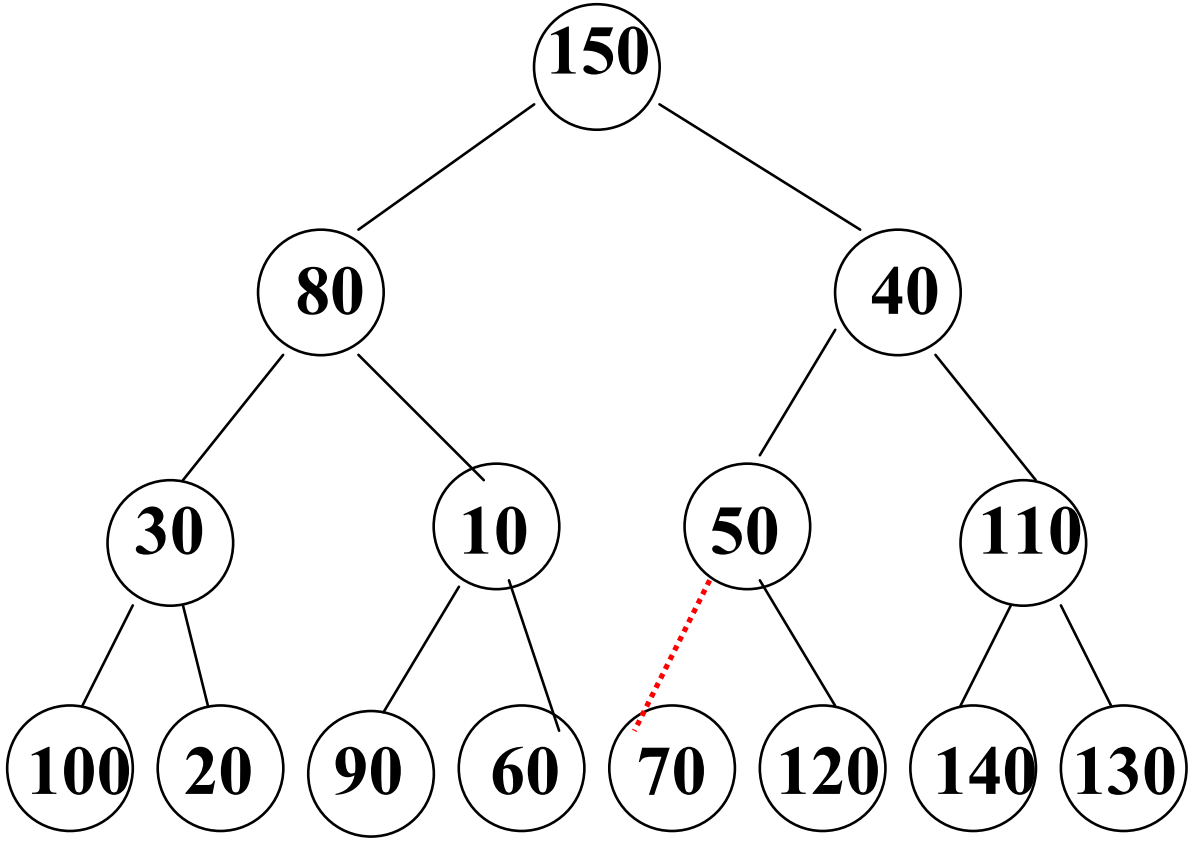
- After *percolateDown* (7)



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

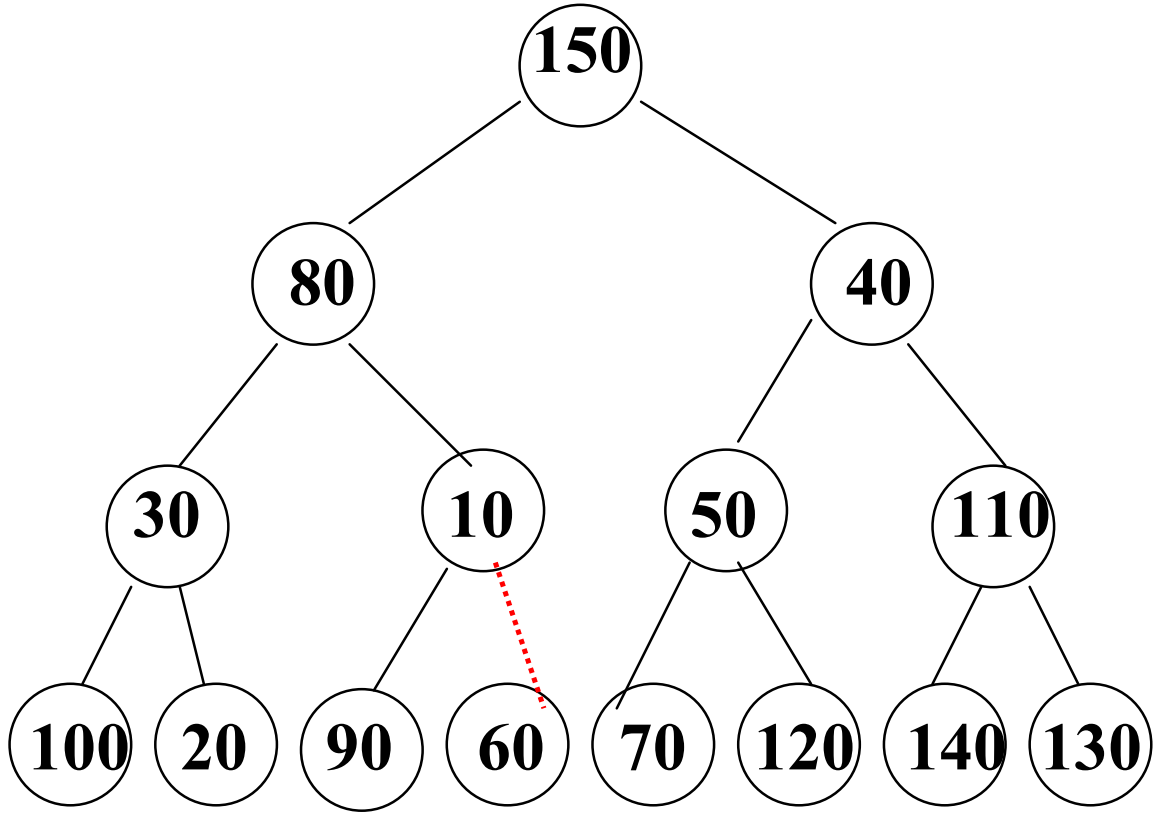
Binary Heap - Other Heap Operations

- After *percolateDown* (6)



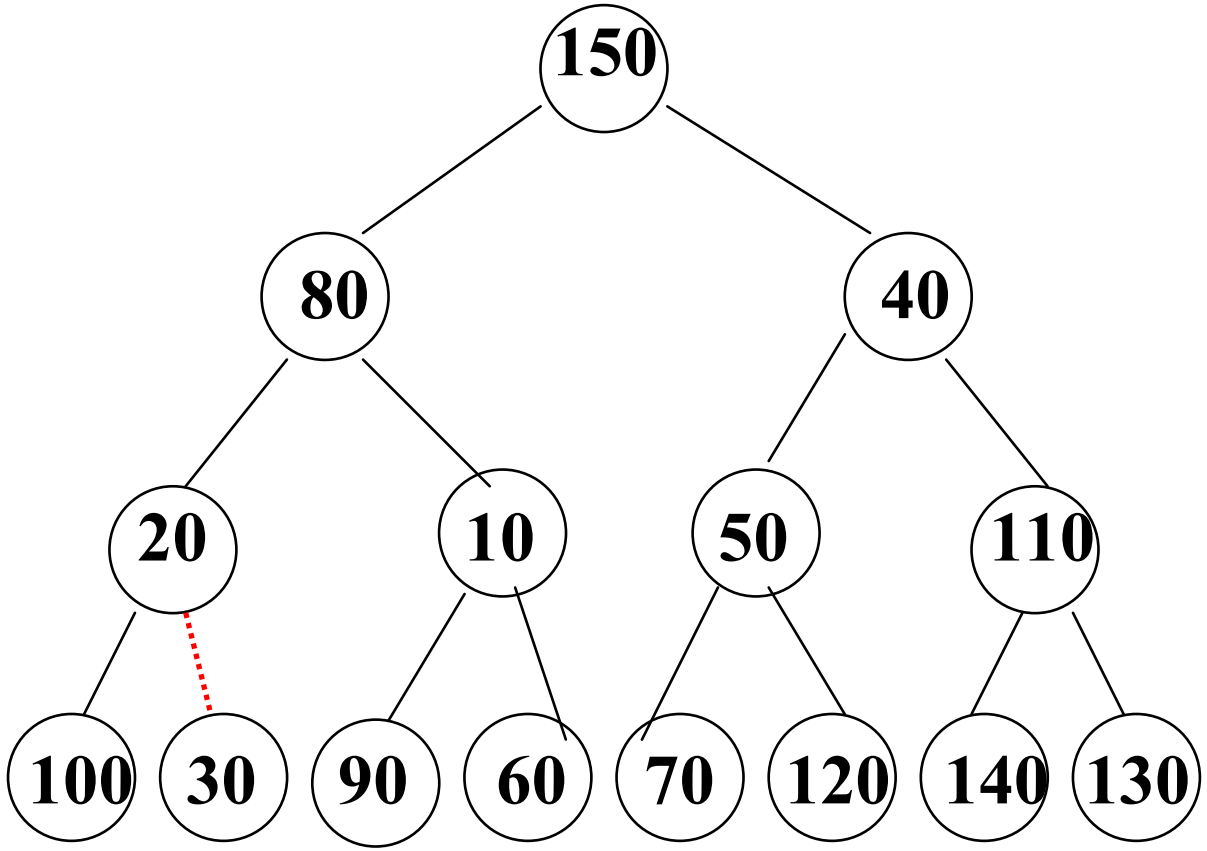
Binary Heap - Other Heap Operations

- After *percolateDown* (5)



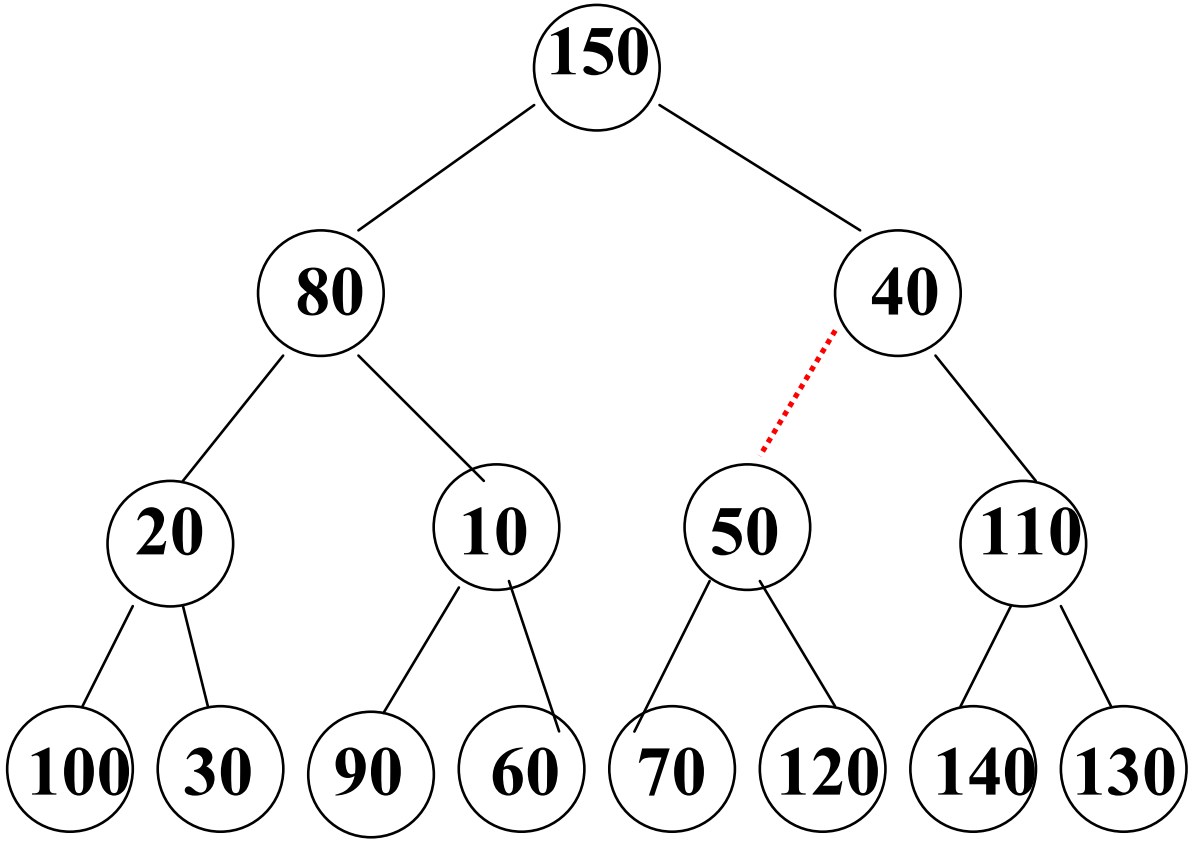
Binary Heap - Other Heap Operations

- After *percolateDown* (4)



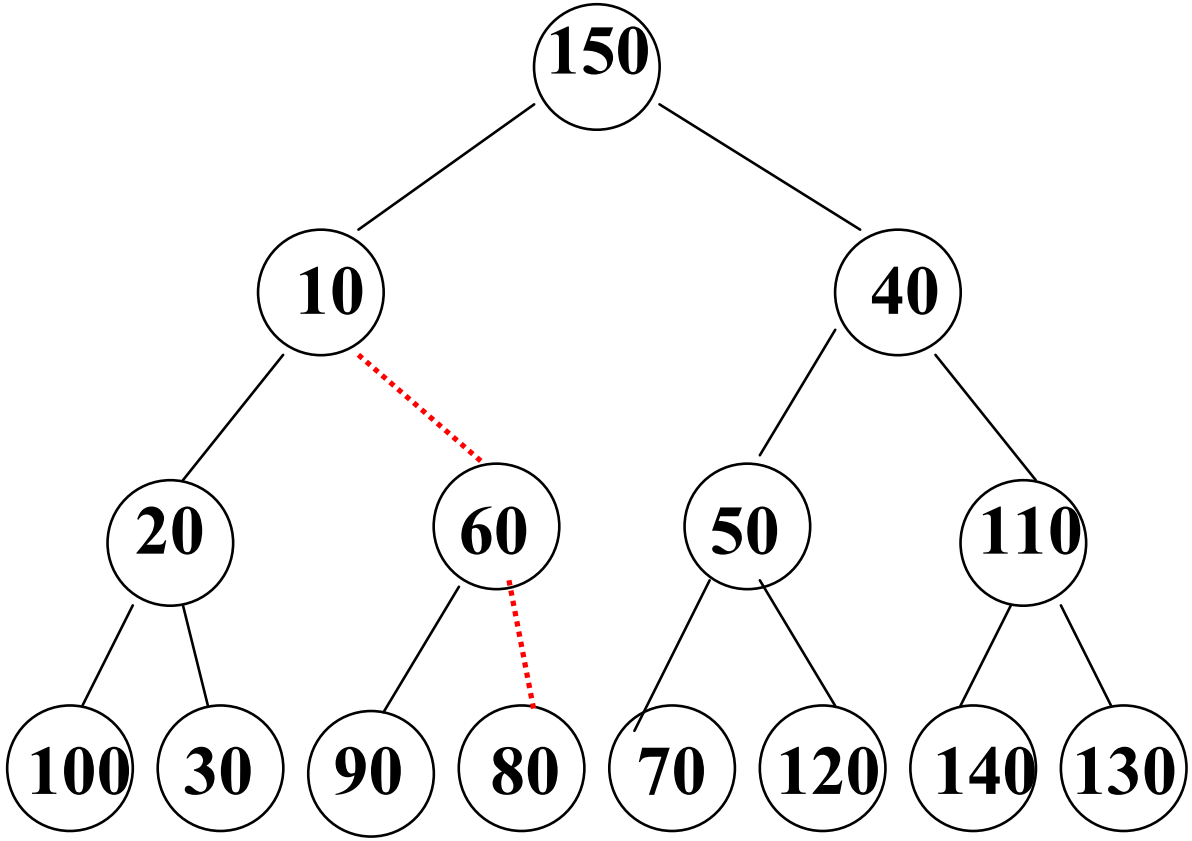
Binary Heap - Other Heap Operations

- After *percolateDown* (3)



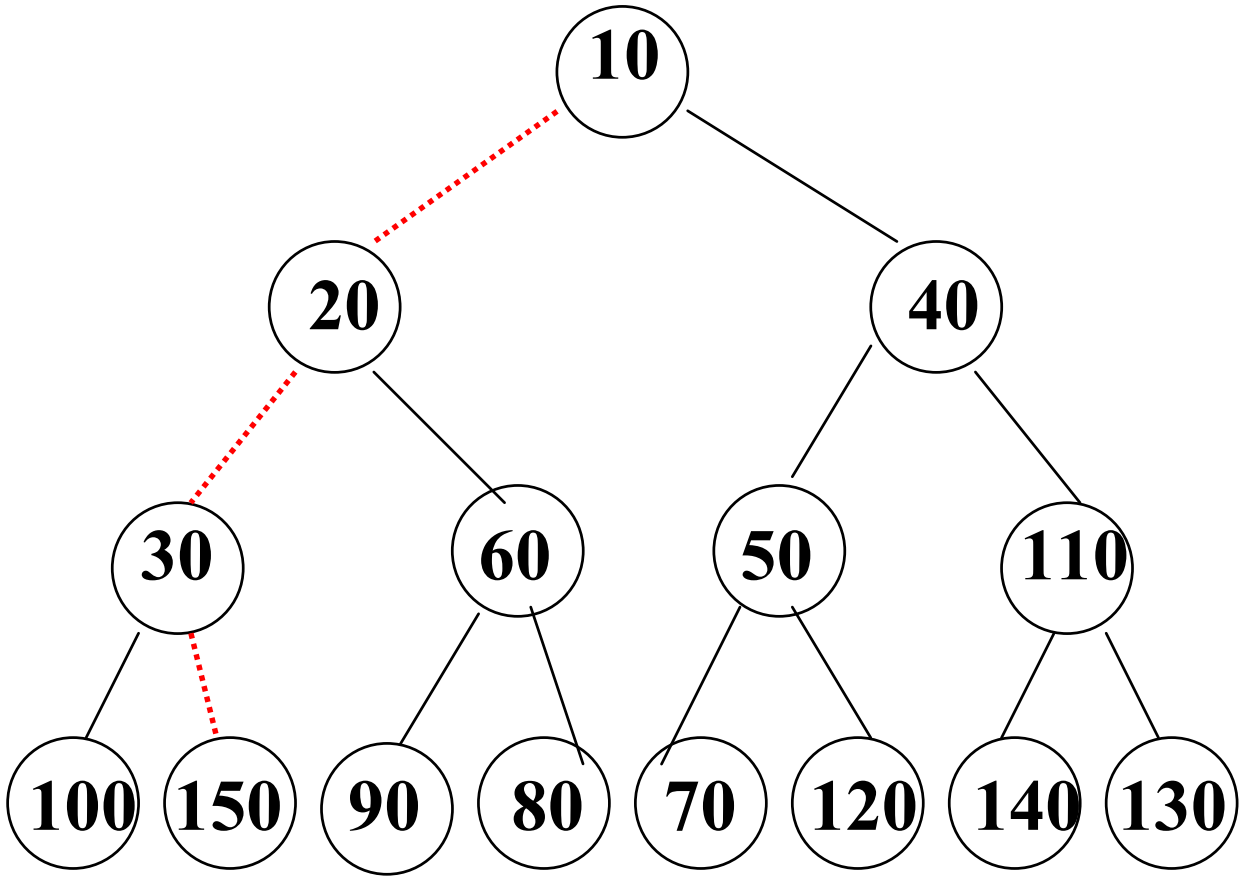
Binary Heap - Other Heap Operations

- After *percolateDown* (2)



Binary Heap - Other Heap Operations

- 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 very tight?

Complexity of BuildHeap?

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

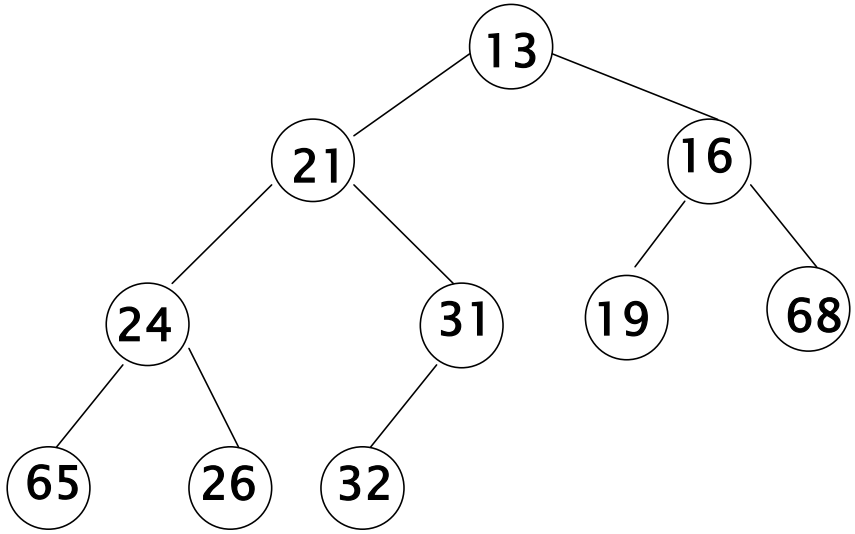
$$1\frac{n}{4}+2\frac{n}{8}+3\frac{n}{16}+\ldots=\sum_{i=1}^{\log n} i\frac{n}{2^{i+1}}=$$

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

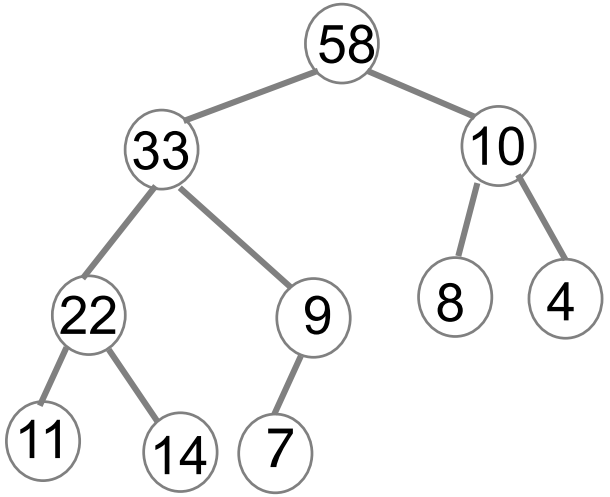
Conclusion: O(N)

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



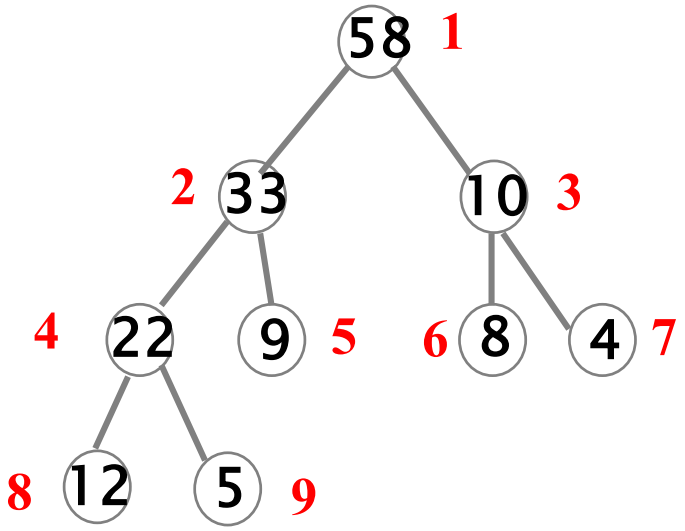
- Max-heap
  - The key present at the root node must be larger than or equal among the keys present at all of its children
  - The same property must be recursively true for all sub-trees



HeapSort

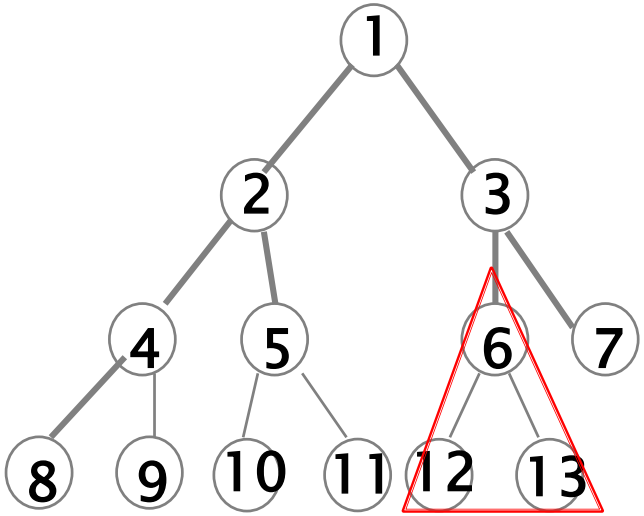
- Sorting using a max-heap
  - To sort an array arr, we first create a max-heap *H* with a capacity of arr.length+1
  - Then, we insert arr[0] into *H*, insert arr[1] into *H*,…, *insert arr[arr.length-1] into H*
  - Then, we repeatedly delete from the max-heap until the max-heap becomes empty

4	5	8	9	10	12	22	33	58



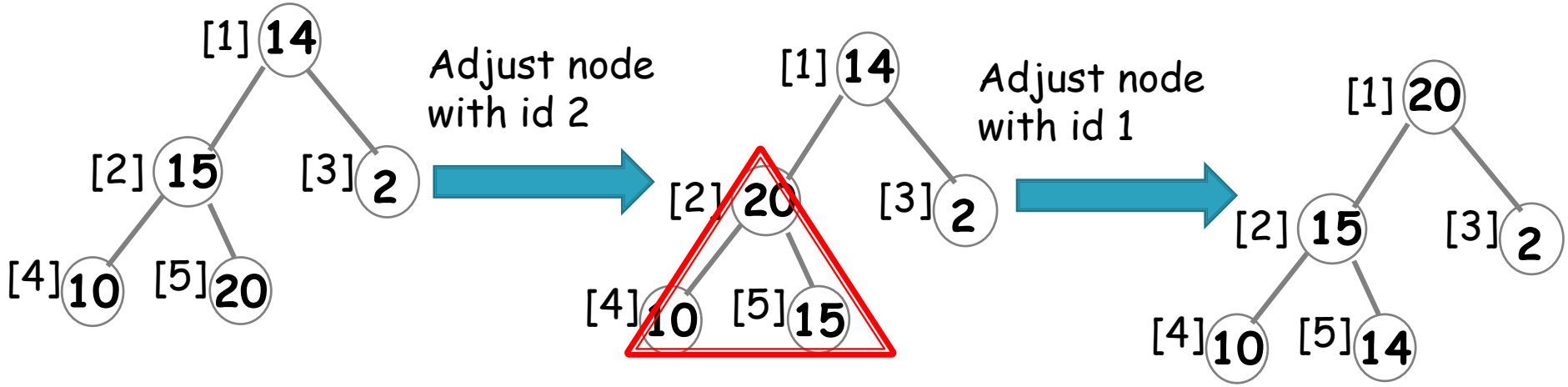
HeapSort: Heap Adjust

- Idea: Given a node v, if its left subtree and right subtrees are max tree, we can adjust the tree rooted at node v a max-tree
  - Given node 6, both its left and right subtrees are max-trees (no violation). How to adjust to make the tree rooted as 6 a max-heap?
  - Hint: How we do the heap adjustment after the deletion?
  - percolateDown!



Heap Adjust

- For a complete binary tree, we adjust from node with id $\lfloor \frac{n}{2} \rfloor, \lfloor \frac{n}{2} \rfloor - 1, \lfloor \frac{n}{2} \rfloor - 2, \cdots 1$
  - In the example of complete binary tree C_1, we only need to adjust from node with id 2 to id 1



A complete binary tree C_1

Heap Adjust: Implementation

- The implementation is quite similar to heap deletion

Algorithm 1: ***HeapAdjust(heap, nodeid, heapsize)***

```
1  temp ← heap.arr[nodeid]
2  parent ← nodeid, child ← 2*nodeid
3  While child <= heapsize
4      if child<heapsize and heap.arr[child].key<heap.arr[child+1].key
5          child ← child + 1 //choose the larger child
6      if temp.key >= heap.arr[child].key
7          break // no violation if we put at position 'parent'
8      heap.arr[parent]←heap.arr[child]//swap larger child with parent
9      parent ← child //attempt to insert at the child position
10     child ← 2*child
11 heap[parent] ← temp
```

HeapSort

- First adjust the nodes from $\lfloor \frac{n}{2} \rfloor$, $\lfloor \frac{n}{2} \rfloor - 1$, $\lfloor \frac{n}{2} \rfloor - 2, \cdots 1$ to make it a max-heap
  - Then, swap the largest number and the last element in the array, reduce the heap size by one and adjust the heap again

Algorithm: ***HeapSort(arr, n)***

```
1  //sort arr[1..n]
2  //adjust the complete binary tree so that it becomes a max-heap
3  for i ← n/2 downto 1
4      adjust(arr, i, n)
5  //swap the root and last element, reduce the heap size, and adjsut
6  for i ← n − 1 downto 1 // i is the heap size
7      arr[i+1] ↔  arr[1] //swap the root and the last element
8      adjust(arr, 1, i) // adjust the heap of size i
```

Practice

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

Complexity Analysis

- We adjust for $\frac{n}{2}$ nodes to make the complete binary tree a max-heap
  - $C_{adjust} = O(n)$
- We then repeatedly swap the root and the last element, and do the adjustment
  - $C_{sort} = O(n \cdot \log n)$
- Total time cost:
  - $C_{adjust} + C_{sort} = O(n \cdot \log n)$
- What is the extra space cost?

HeapSort advatanges

- Time cost
  - The time required to perform Heap sort increases logarithmically
  - HeapSort is particularly suitable for sorting a huge list of items
  - The performance of HeapSort is optimal (see analysis later)
- Space cost
  - HeapSort can be implemented as an in-place sorting algorithm, which means that its memory usage is minimal because it needs no additional memory space to work
  - In contrast, MergeSort requires more memory space, and similarly, QuickSort requires more stack space due to its recursive nature

HeapSort advantages

- Simplicity
  - HeapSort is simpler to understand than other equally efficient sorting algorithms
  - Because it does not use advanced techniques such as recursion, it is also easier for programmers to implement correctly
- Consistency
  - HeapSort exhibits consistent performance, which means it performs equally well in the best, average, and worst cases
  - Because of its guaranteed performance, it is particularly suitable to use in systems with critical response time

Comparison of 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)
Merge sort	√	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Heap sort	×	O(nlogn)	O(nlogn)	O(nlogn)	O(1)
Quick sort	×	O(nlogn)	O(nlogn)	$O(n^2)$	O(logn)

** A sorting algorithm is said to be stable, if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

Selection Sort: 5, 1, 2, 1* => 1*, 1, 2, 5

How Fast Can We Sort?

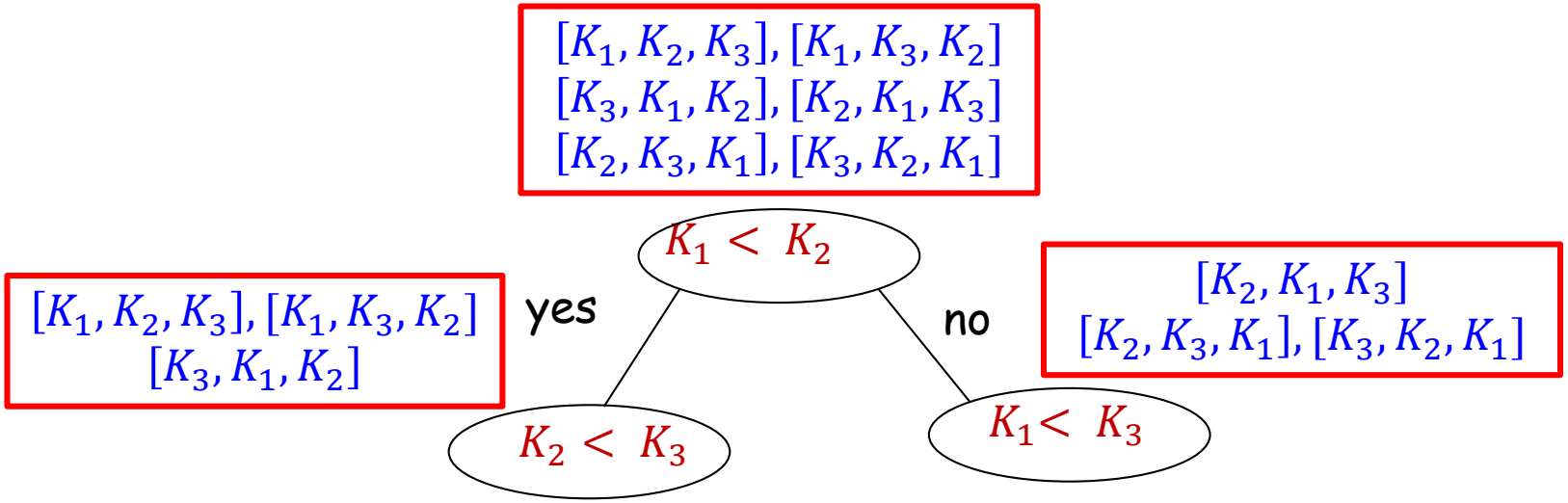
- In terms of the worst case analysis, we have seen algorithms with either $O(n \log n)$ or $O(n^2)$
- Is there any hope that we can do better than $O(n \log n)$, for example $O(n)$? In other words, what is the best we can achieve?
- Let's consider the scenario where the operations allowed on keys are only comparisons
  - E.g., <, >, =, …

How Fast Can We Sort?

- Given an array A with length n, there are $n!$ different permutations of the elements therein
  - If n = 3, then there are 6 permutations:
    - A[1], A[2], A[3]
    - A[1], A[3], A[2]
    - A[2], A[1], A[3]
    - A[2], A[3], A[1]
    - A[3], A[1], A[2]
    - A[3], A[2], A[1]
  - The goal of the sorting problem is essentially to decide which of the $n!$ permutations corresponds to the final sorted order

How Fast Can We Sort?

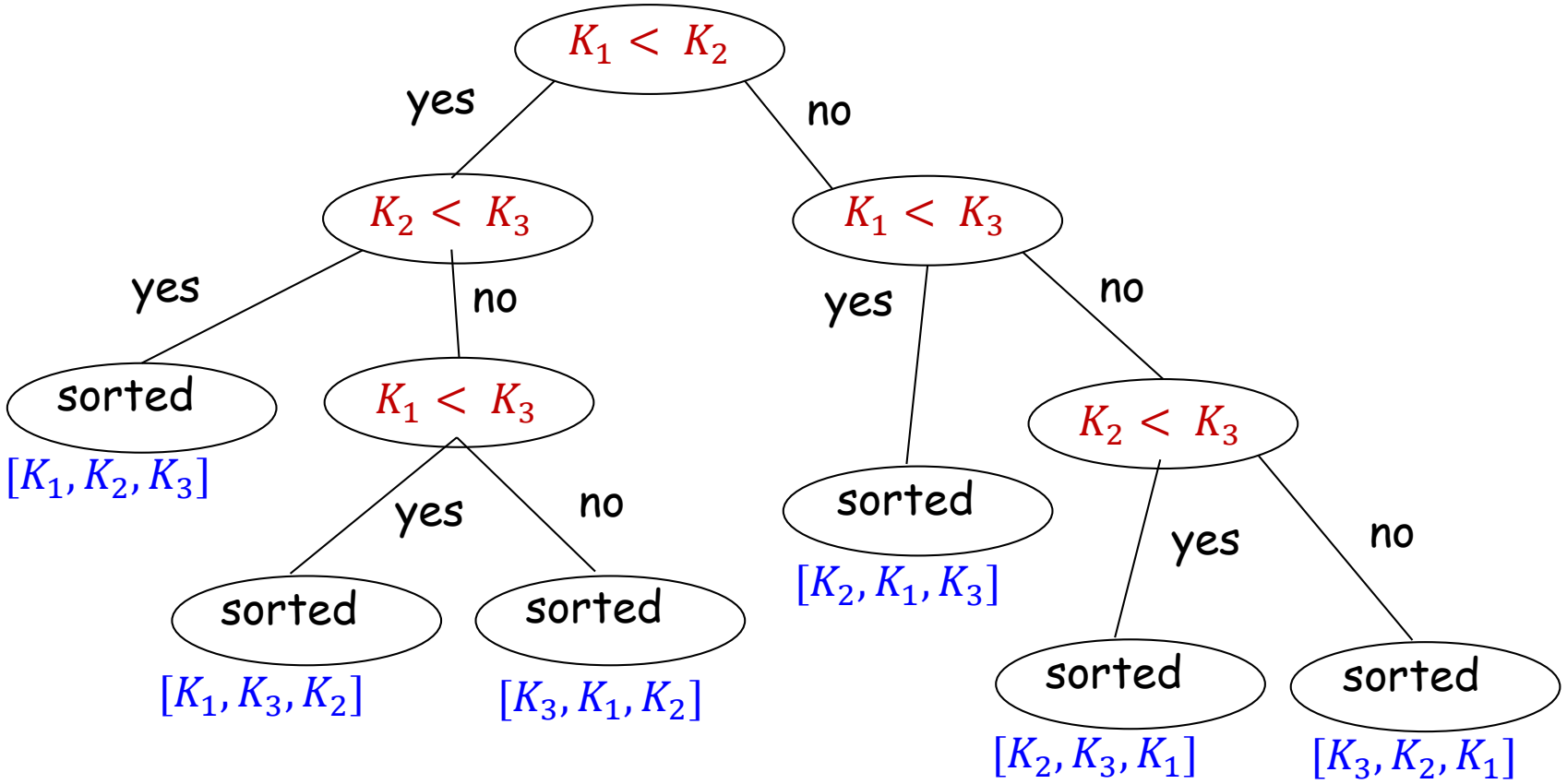
- Consider a decision tree that describes the sorting :
  - A node represents a key comparison
  - An edge indicates the result of the comparison (yes or no). We assume that all keys are distinct



The result of a comparison, e.g., $K_1 < K_2$, makes the possible number of permutation satisfying the constraint (e.g., $K_1 < K_2$) become smaller and smaller

How Fast Can We Sort?

- Consider a decision tree that describes the sorting:
  - A node represents a key comparison
  - An edge indicates the result of the comparison (yes or no). We assume that all keys are distinct



How Fast Can We Sort?

- **Theorem**: Any decision tree that sorts n distinct keys has a height of at least $\log_2 n! + 1$.

Proof: When sorting n keys, there are $n!$ different possible results. Thus, every decision tree for sorting must have at least $n!$ leaves.

Note a decision tree is a binary tree, which has at most 2^{k-1} leaves if its height is k. Therefore, $2^{k-1} \geq n!$, the height must be at least $k \geq \log_2 n! + 1$.

Notice: $\log_2 n! = \sum_{i=1}^{n} \log i \geq \sum_{i=\frac{n}{2}}^{n} \log i \geq \frac{n}{2} \cdot \log \frac{n}{2} = \Omega(n \cdot \log n)$

Therefore, the comparison based sorting algorithms needs $\Omega(n \cdot \log n)$ comparisons in the worst case

So no hope to beat $\Omega(n \cdot \log n)$?

- That is not true. Notice our constraint:
  - $\Omega(n \cdot \log n)$ if we only use comparison
  - We can reduce the sorting cost if we make certain assumptions on the data

- Let's consider the following special case
  - Given a set of n integers ranging from $[1, U]$ with $U = O(n)$
    - Can we sort in linear time?

Recommended reading

- Reading
  - Chapters 6&12, textbook
- Next lecture
  - Red-Black trees
  - Chapter 13, textbook