Session 7

Sorting - II

Overview

- More sorting algorithms
  - Merge sort
  - Quick sort
  - Heap sort
- Comparison amongst these.

How fast can we sort?

- We saw simple algorithms offering $O(N^2)$.
- Shellsort provides $O(N^{1.25})$.
- To sort we need to visit each element at least once, hence no less than O(N).
- Comparison based algorithms require at least O(N log N).
- Now we look at a few algorithms with nearly this complexity.
- Other techniques can provide a better order (e.g. radix sort)

Divide and Conquer

- A general technique for algorithm design.
- Applicable for a large range of problems.
- Basic idea: Divide the problem into parts, solve the parts and then combine the solutions.
- First, a quick look at recursion...

Recursion

- A powerful model of problem solving.
- Binary Search is a good example.
- Other examples: factorial, fibonacci, tree traversals

  fib(N) = fib(N-1) + fib(N-2)

  fact(N) = N * fact(N-1)
- Divide and Conquer problems are easy to model recursively.

Binary Search

- Assume records are ordered on the key value (e.g. telephone directory)
- Subdivide the file into two equal parts, and examine the middle element.
- The target record is either =, < or > this record. If equal, return index.
- If ‘<‘, the right half can be discarded; otherwise, the left half can be discarded.

Binary Search

- Repeat the process with the other half.
- At each step, the volume of search reduces by half.
- Terminate when the file has only 1 element.

Binary Search

```
public int binarySearch(int [] data, int target, int low, int
    high)
{
    int mid;
    if(low>high)
        return –1; // not found
    mid=(low+high)/2;
    return (target ==a[mid] ? mid : (target <a[mid] ?
  binarysearch(low,mid-1) :  binarysearch(mid+1,high) ));
}
```

Recursion

- A base (termination) case essential to prevent run-away recursion.

  fib(1) = 1; fib(0) = 1;

  fact(1) = 1;

- Solution is done bottom-up, but invoked top-down.

Recursion

- fact(4)
  - 4 * fact(3)
    - 3 * fact(2)
      - 2 * fact(1)
        - » 1
      - 2 * 1 = 2
    - 3 * 2 = 6
  - 4 * 6 = 24
- return 24

Recursion

- A procedure calling itself, is no different from it calling another.
- Note that local variables are local to a call.
- Ensure that the recursive call is closer to base case than the original call.

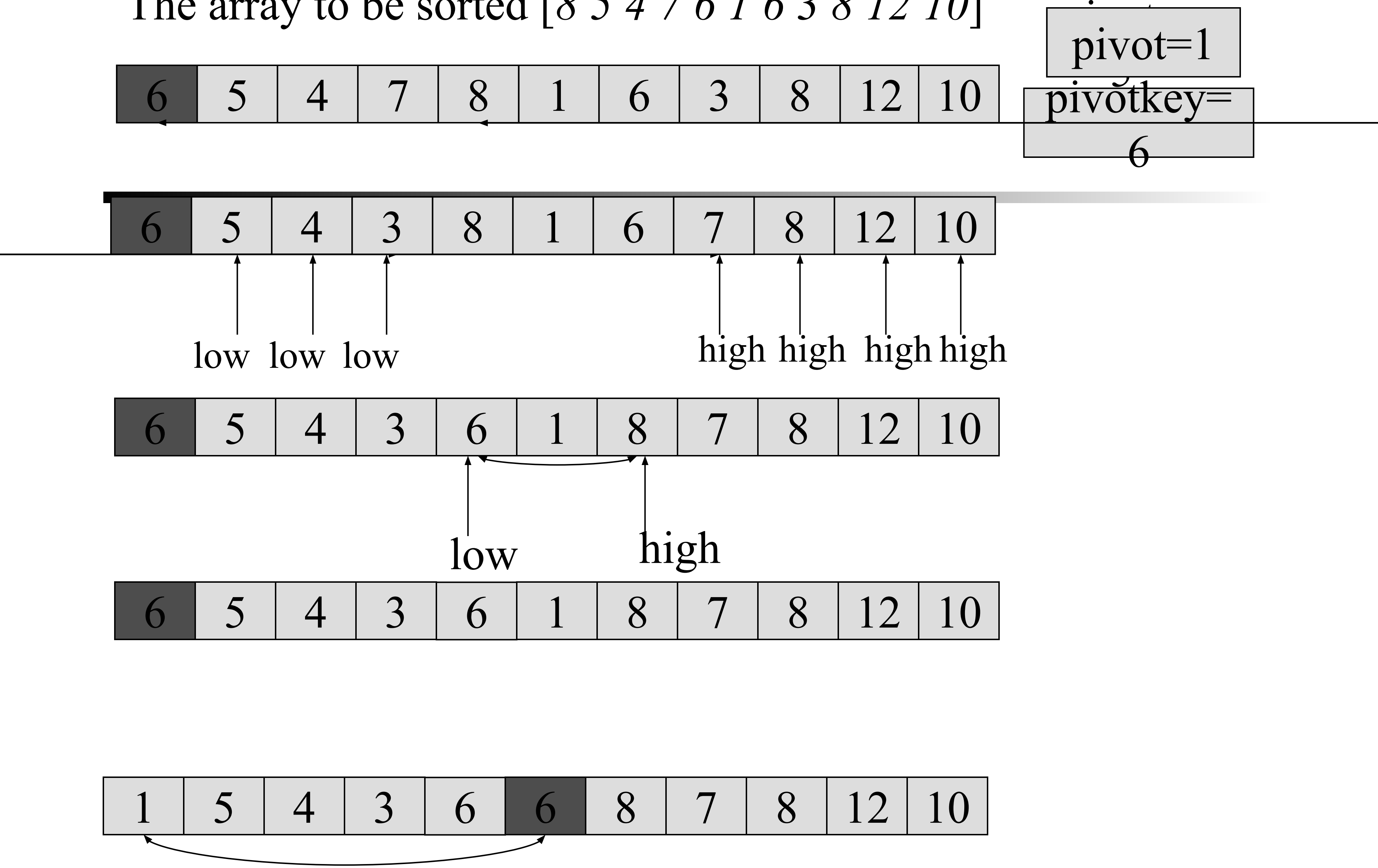
Divide and Conquer General Model

- To sort array 1 to N, divide the array into two pieces 1 to M, and M+1 to N. Sort the two arrays and combine the result.
- Simple split, sort and putting together will not do!
- Two methods:
  - One takes some effort to generate independent parts so that combining becomes easy.
  - Other splits arbitrarily, but takes effort to combine.

Quick Sort

- Ensure that fragments are independent.
- Choose a number as bound (pivot) –e.g $A[1]$, $A[N]$, $A[N/2]$.
- With a single scan, move all elements larger than pivot to right partition, and others to left partition.
- Now, all elements of left partition are less than pivot, which is less than right partition elements
- Hence if the two partitions are sorted, the array is sorted.
- Two partitions provide two recursive sub-problems.

The array to be sorted [*8 5 4 7 6 1 6 3 8 12 10*]

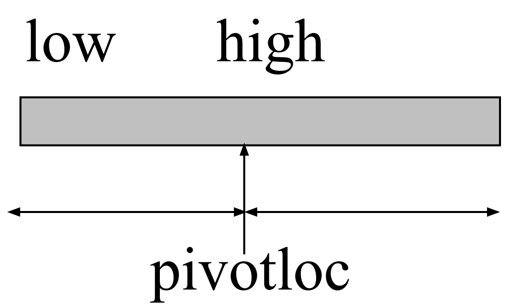


Implementation

```
public void quickSort(int[] A, int N)
{
   Sort (A, 0, N-1);
}
```

Implementation

```
void Sort(int[] A, int low, int high){
  int pivotloc;
  if (low <high){
        pivotloc = Partition(A,low,high);
        Sort(A, low, pivotloc -1);
        Sort(A, pivotloc +1,high);
    }
   }
```



Implementation - Partition

```
int Partition(int[]A, int low, int high)
{
   int down=low, up=high;
   int pivot=(down+up)/2 ,pivotkey=A[pivot];
   swap(A[low], A[pivot]);
   while(down<up){
      while(A[down]<=pivotkey && down< high)
      down++;
      while(A[up]>pivotkey)
         up--;
      if(down < up) swap(A[down],A[up]);
   }//while
   swap(A[low],A[up]); return up;
}//partition
```

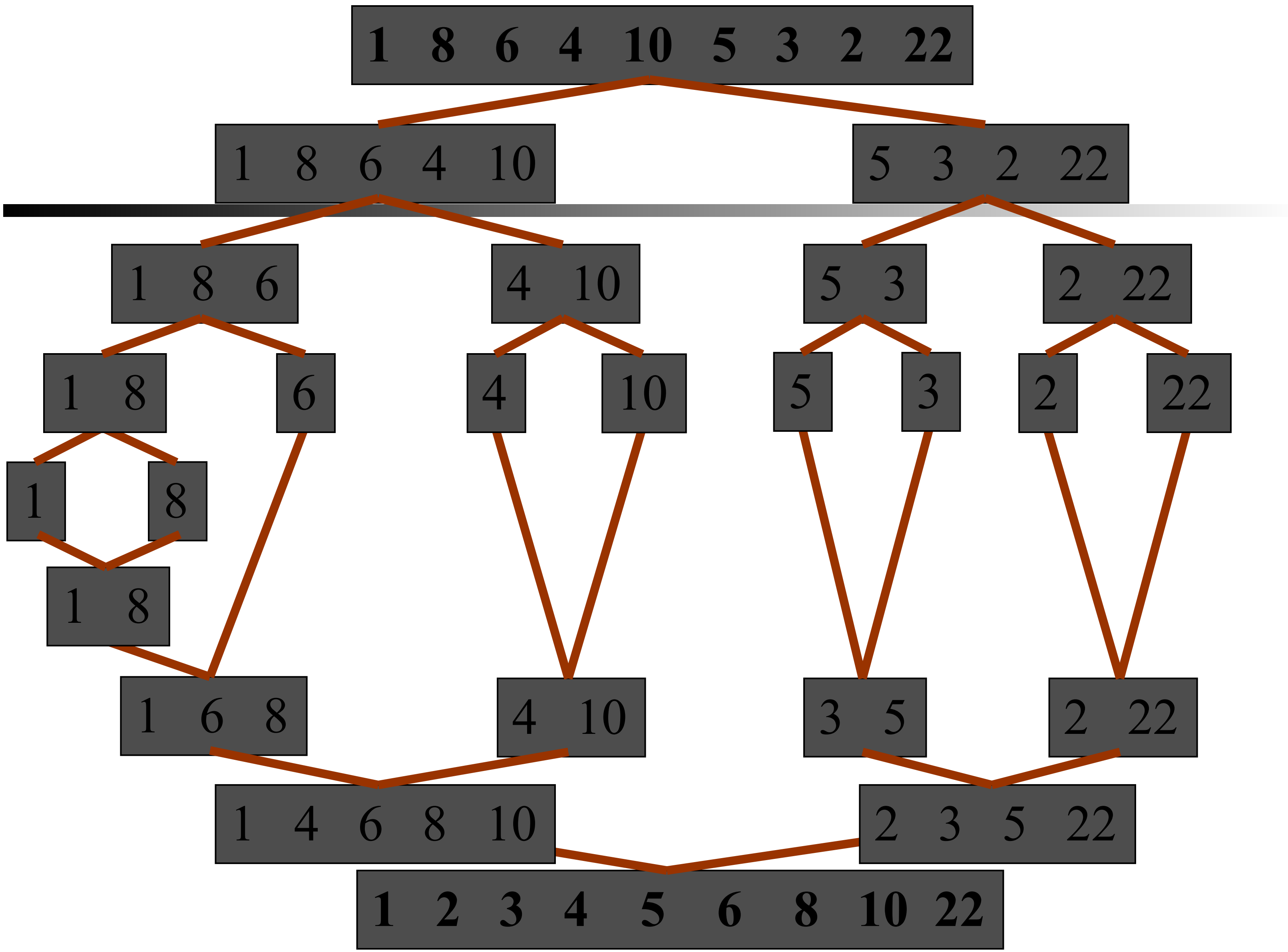
Analysis

- Choice of pivot affects the nature of partitioning.
- Best case when the pivot divides array into equal halves: both partitions have same nearly number of elements.
- Number of levels of partitioning - log N
- Complexity: N log N
- Worst case, when the pivot is the largest/smallest element of the array, one partition is empty and the other has all the elements.
- Number of levels of partitioning - N
- Complexity degrades to selection sort - $O(N^2)$
- Suitable for large and randomly sorted data

Merge Sort

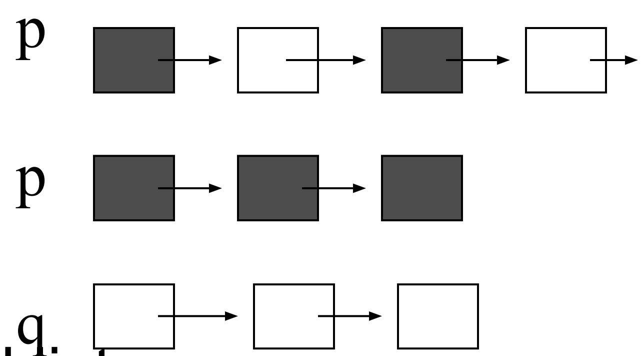
- Split the array in half.
- Sort the parts.
- Combine (Merge) the sorted parts.
- Merging can be done in linear time, since the parts are sorted.

The array to be sorted : [1 8 6 4 10 5 3 2 22]



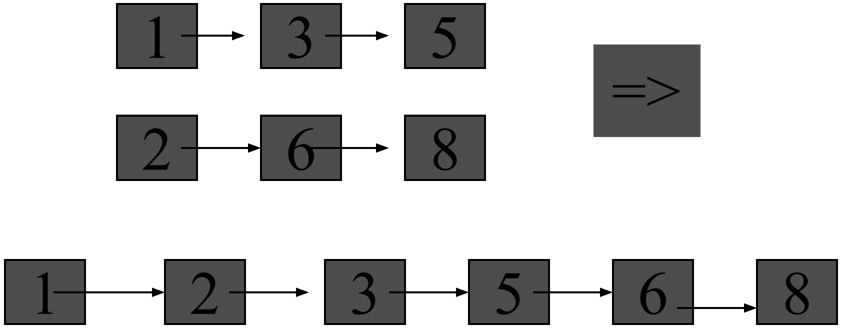
Algorithm

```
mergeSort(List_type[] p){
    if(atleast two elements){
        mergeSort(left half of data);
        mergeSort(right half of data);
        merge both halves into a sorted list;
    }
}
```



Merge

- Compare front element of both lists.
- Take the smaller, add to the output.
- Repeat till one list is empty.
- Output all elements of the other list.



Analysis

- No best/worst cases.
- Log N divisions before a size of 1 is reached.
- Each step requires O(N) for merging.
- Hence O(N log N).
- Auxiliary storage as large as the original is required

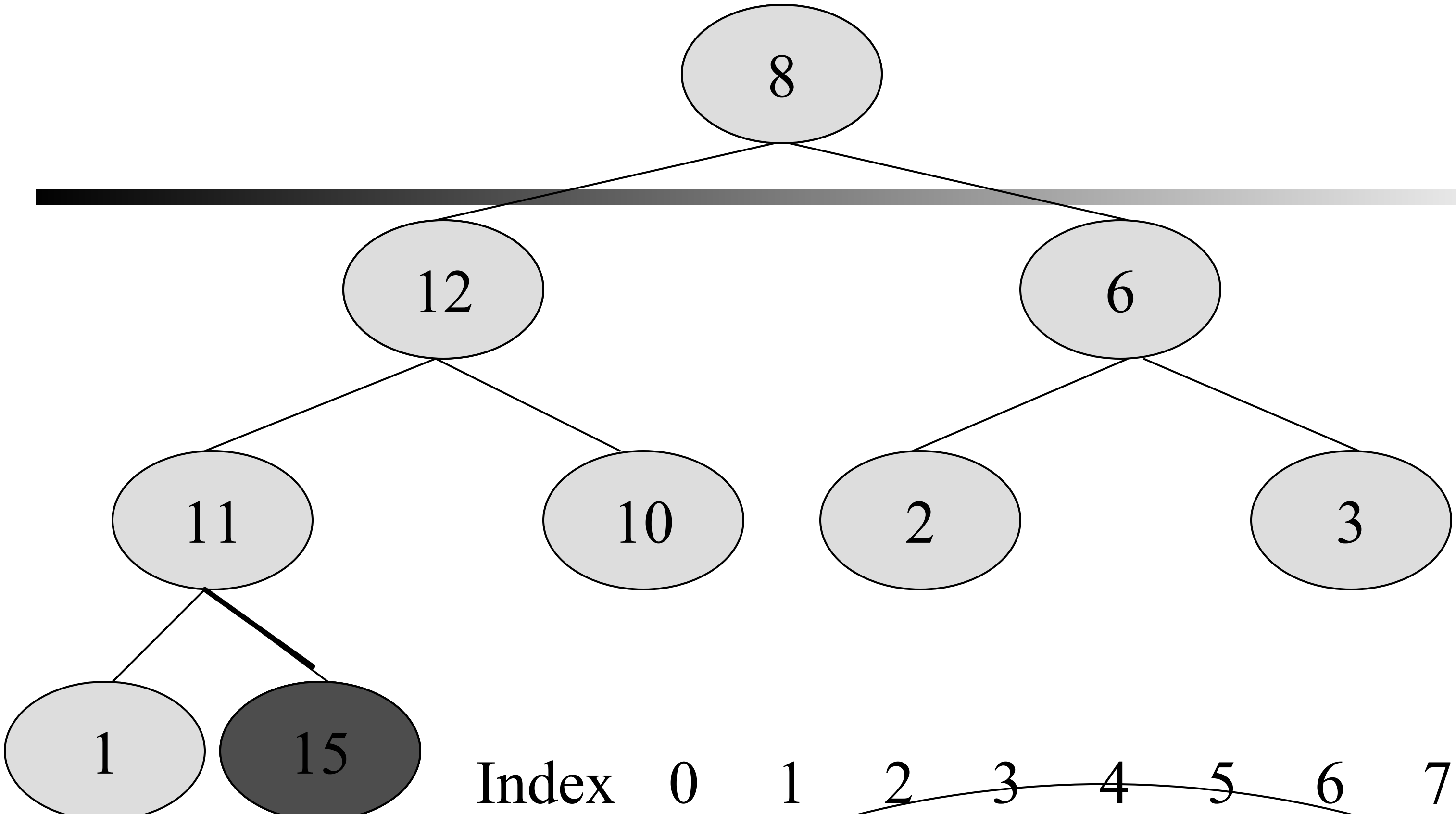
Heap Sort

- Given array of numbers, transform into a heap.
- The maximum is at the root, take it out and rearrange remaining as a heap.
- Repeat N times -> sorted output!

Heap Sort

- Assume array 1…M is a heap, a[1] largest element.
- Swap *a*[1] with *a*[*M*], highest value now in *a*[*M*].
- *a*[1] may not satisfy heap property.
- Compare against children, ensure highest at root.
- Repeat till leaf node.
- M = M – 1
- Repeat M times.

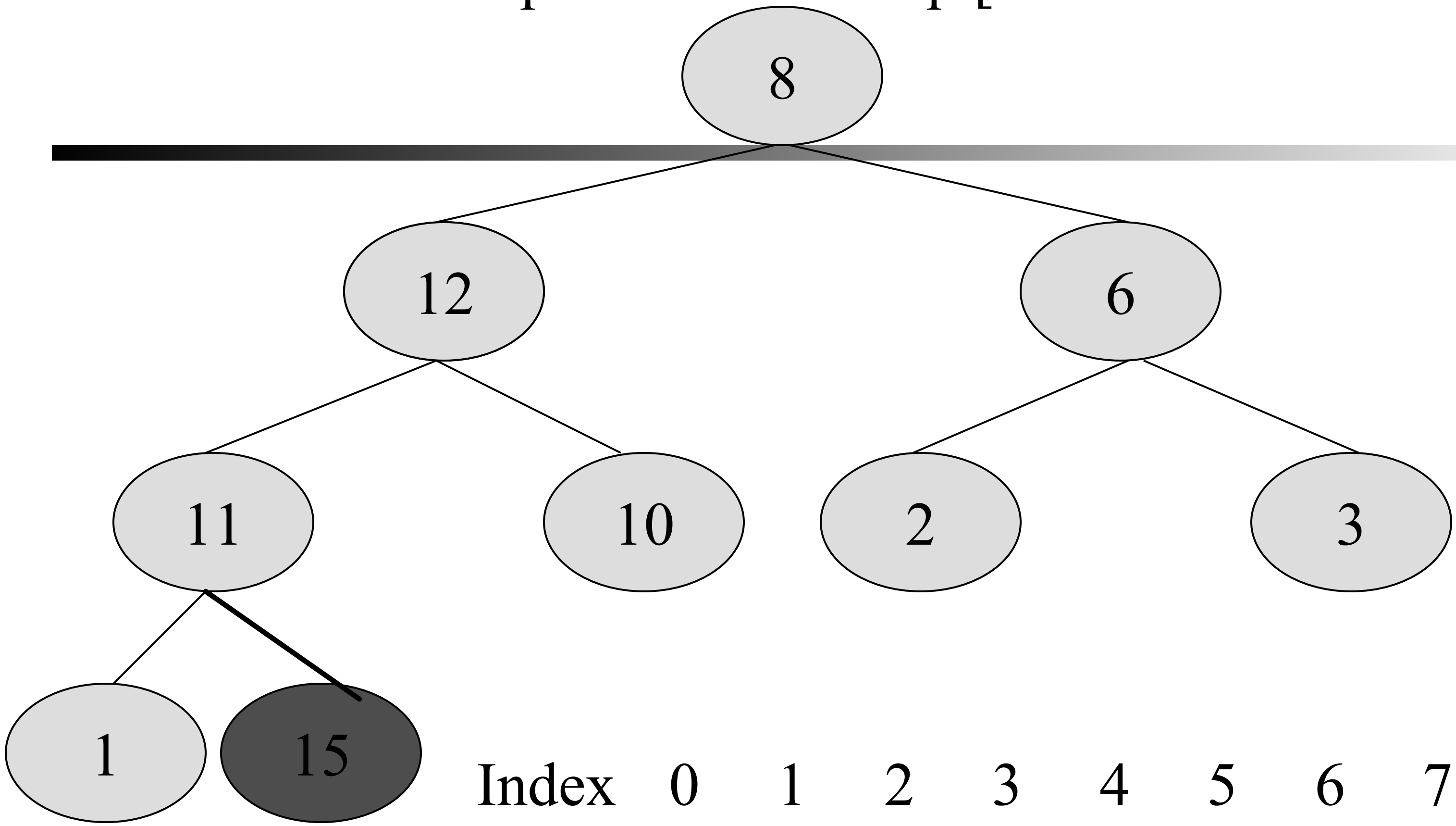
Execution of heap sort on the heap [*15 12 6 11 10 2 3 1 8*]



Index	0	1	2	3	4	5	6	7	8
data	8	12	6	11	10	2	3	1	15

Swap data[0] with data[data.length-1], highest value now in data[data.length-1]

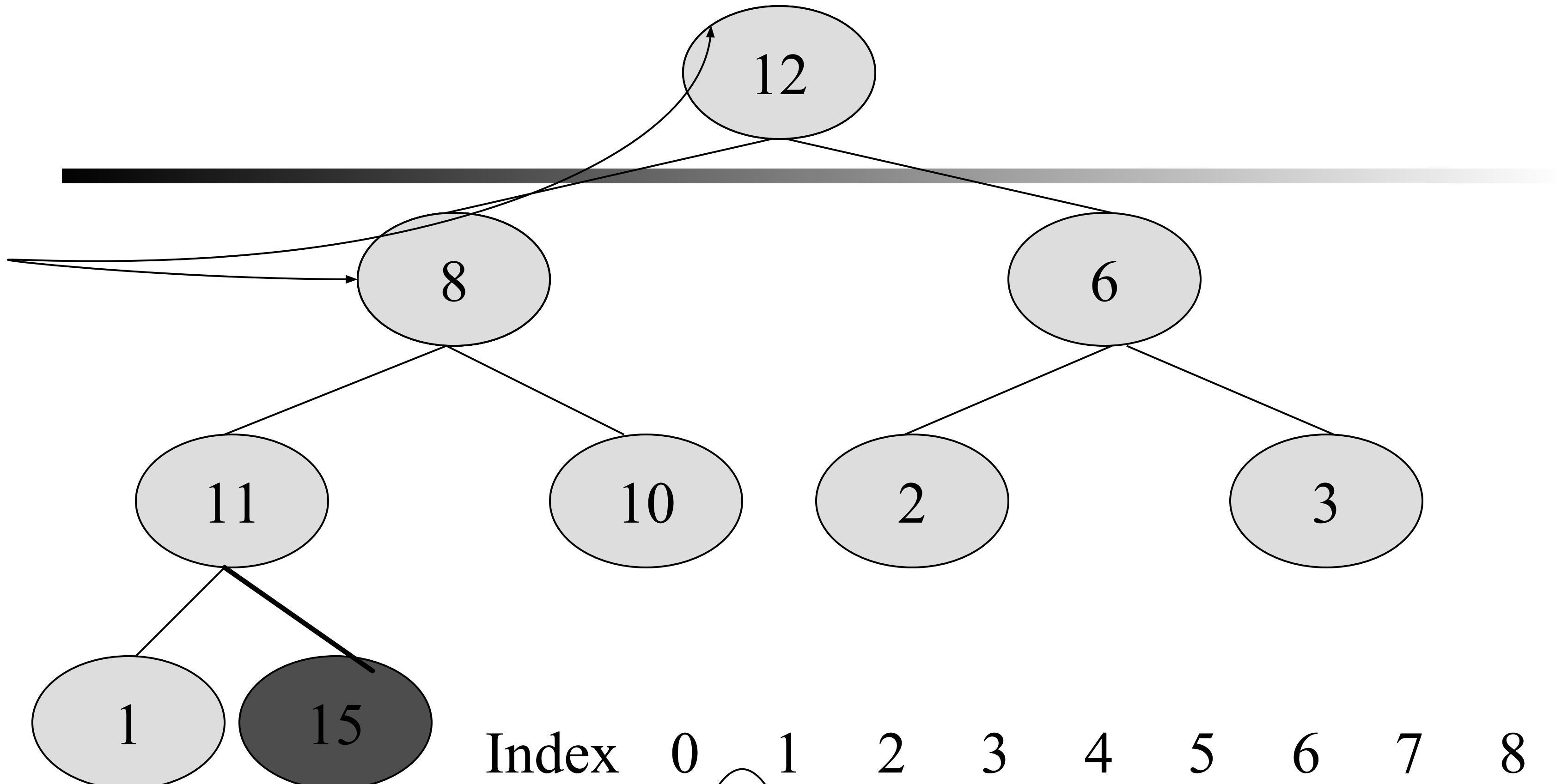
Execution of heap sort on the heap [*15 12 6 11 10 2 3 1 8*]



Index	0	1	2	3	4	5	6	7	8
data	8	12	6	11	10	2	3	1	15

Now, data[0] may not satisfy heap property.

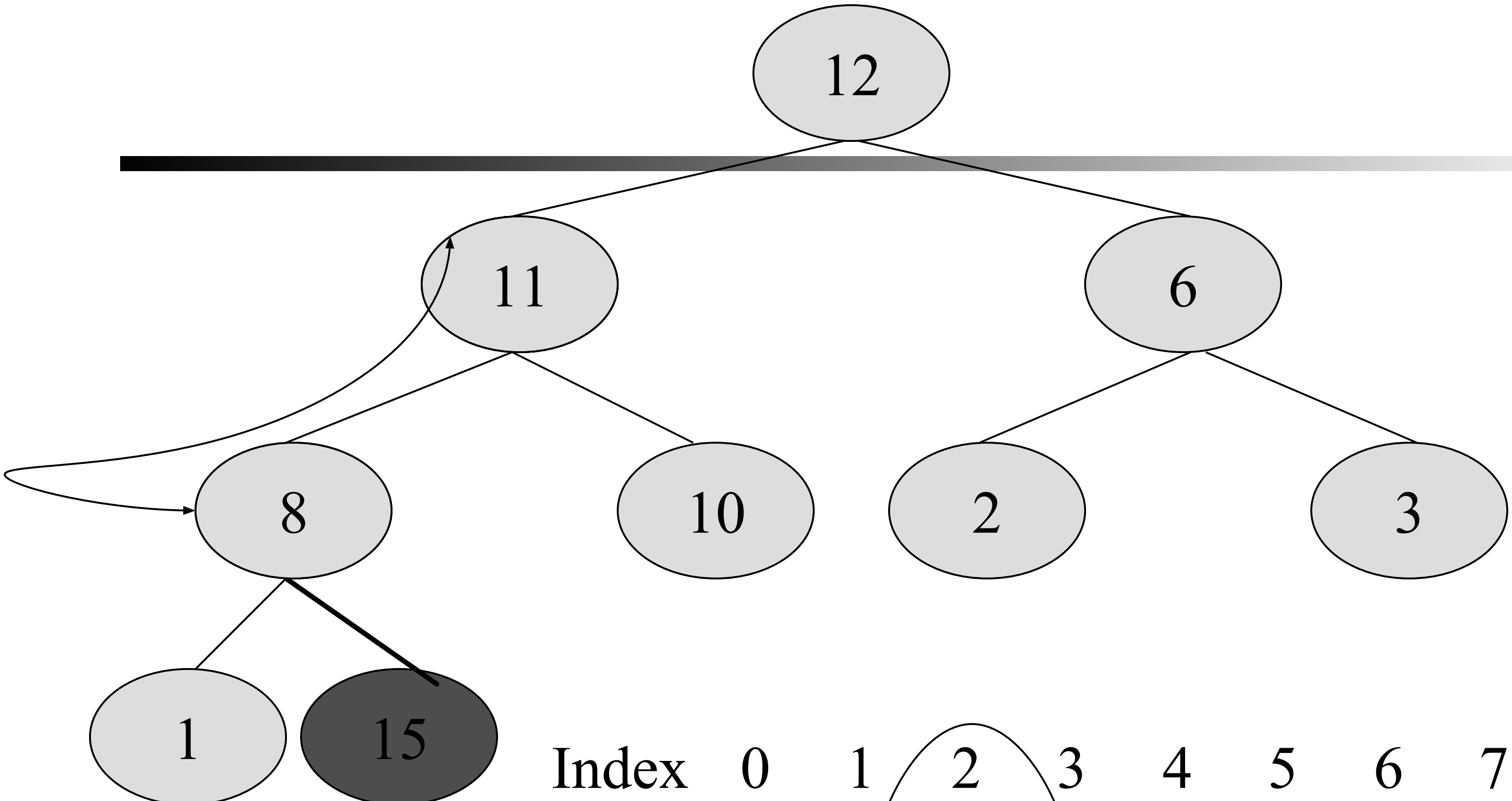
Execution of heap sort on the array [*15 12 6 11 10 2 3 1 8*]



Index	0	1	2	3	4	5	6	7	8
data	12	8	6	11	10	2	3	1	15

Compare against children, ensure heap property i.e. highest at root, except the last, i.e. data[Index-1].

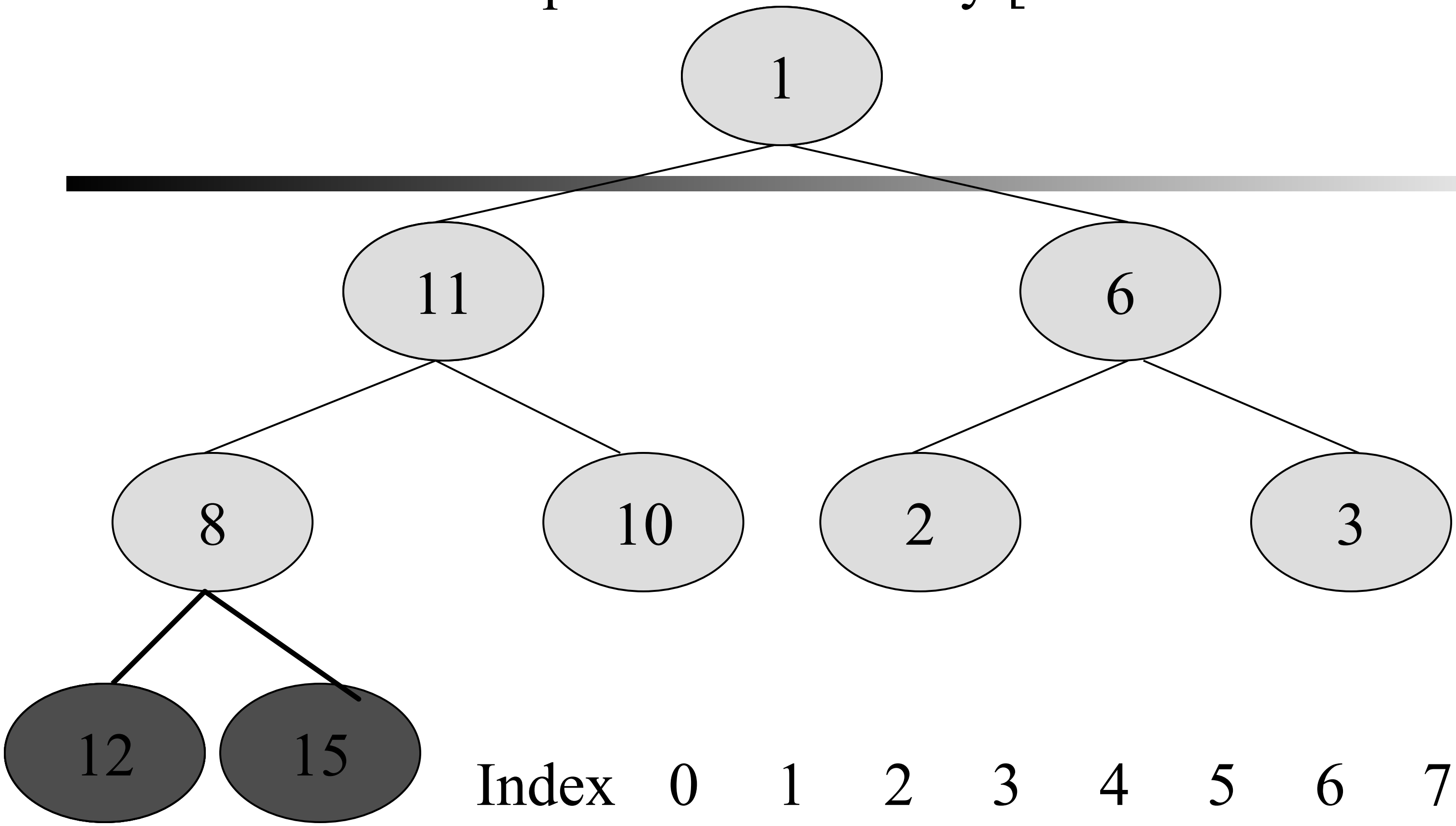
Execution of heap sort on the array [*15 12 6 11 10 2 3 1 8*]



Index	0	1	2	3	4	5	6	7	8
data	12	11	6	8	10	2	3	1	15

Compare against children, ensure heap i.e. highest at root, except the last, i.e. data[Index-1].

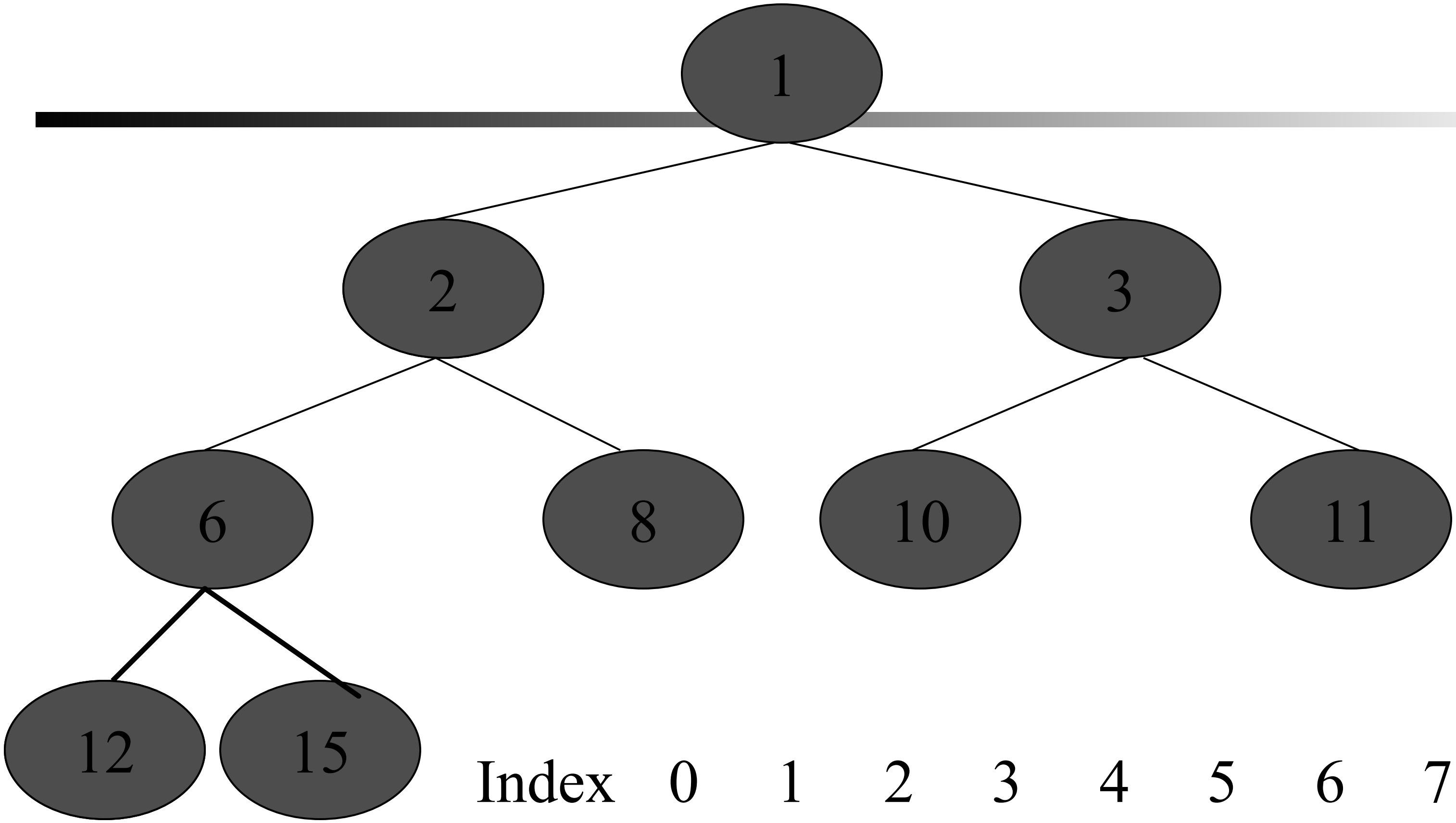
Execution of heap sort on the array [*15 12 6 11 10 2 3 1 8*]



Index	0	1	2	3	4	5	6	7	8
data	1	11	6	8	10	2	3	12	15

Swap data[0] with data[data.length-2], second highest value now in data[data.length-2]. Repeat these steps for all the nodes.

Execution of heap sort on the array [*15 12 6 11 10 2 3 1 8*]



Index	0	1	2	3	4	5	6	7	8
data	1	2	3	6	8	10	11	12	15

Final array will be sorted one.

Analysis

- Creation of heap: O(N log N).
- Sorting a heap: O(N log N) and N-1 swaps.
- So, complexity of heap sort is-
  O(N log N)+O(N log N) + N-1 => O(N log N)

Summary

- We saw Quick sort,merge sort and heap sort.
- Complexity of merge sort and heap sort is always O(N log N)
- Complexity of Quick sort can vary from O(N^2) to O(N log N), Can we always prefer merge sort over quick sort?
- Merge sort uses additional O(N) space.