HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

DATA STRUCTURES AND ALGORITHMS

Sorting

CONTENT

- Overview of sorting
- Insertion Sort
- Selection Sort
- Bubble Sort
- Merge Sort
- Quick Sort
- Heap Sort

Overview

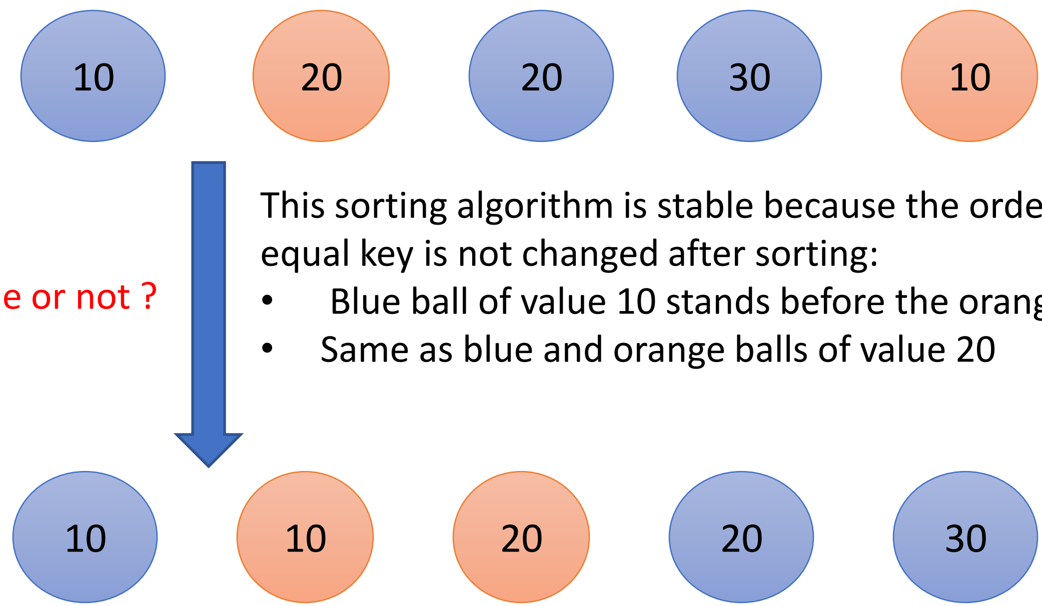
- ***Sorting*** is a process that organizes a collection of data into either ascending or descending order
- Different types of sorting algorithms:
  - ***Internal sort*** requires that the collection of data fit entirely in the computer's main memory.
  - We can use an ***external sort*** when the collection of data cannot fit in the computer's main memory all at once but must reside in secondary storage such as on a disk.
  - ***Parallel sort*** : uses multiple processors to sort, thus reduces the computation time

Overview

- In place
  - Sorting of a data structure does not require any external data structure for storing the intermediate steps
- 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."

Before sorting

This sorting algorithm is stable or not ?



This sorting algorithm is stable because the order of the balls with equal key is not changed after sorting:

- Blue ball of value 10 stands before the orange ball of value 10.
- Same as blue and orange balls of value 20

After sorting

Overview

- There are 2 basic operations that sorting algorithms often use:
  - **Exchange positions of 2 elements** (Swap): running time $O(1)$
  - **Compare:** Compare(a, b) returns true if a is put before b in the sorted order, false otherwise
- **Sorting analysis**: In general, we compare keys and move items (or exchange items) in a sorting algorithm (which uses key comparisons).

➔ So, to analyze a sorting algorithm we should count the number of data comparisons and the number of moves.

(Ignoring other operations does not affect our final result).

Insertion Sort

Algorithm:

- The array is divided into two subarrays, *sorted* and *unsorted*
- Each iteration: the first element of the unsorted subarray is picked up, transferred to the sorted subarray, and inserted at the appropriate place.

➔ An array of *n* elements requires *n*-1 iterations to completely sort the array.

Sorted **Unsorted**

23	78	45	8	32	56	Original array
23	78	45	8	32	56	After iteration 1
23	45	78	8	32	56	After iteration 2
8	23	45	78	32	56	After iteration 3
8	23	32	45	78	56	After iteration 4
8	23	32	45	56	78	After iteration 5

Insertion Sort

- Running time
  - Worst case: O(n^2)
  - Best case: O(n)

```
void insertionSort(int A[], int n) {
  // index tu 1 -> n
  for(int k = 2; k <= n; k++){
   int last = A[k];
   int j = k;
   while(j > 1 && A[j-1] >
        last){
     A[j] = A[j-1];
     j--;
   }
   A[j] = last;
  }
}
```

Selection Sort

- The array is divided into two subarrays, *sorted* and *unsorted*, which are divided by an imaginary wall.
- Each iteration: We find the smallest element from the unsorted subarray and swap it with the element at the beginning of the unsorted subarray

➔ After each iteration (after each selection and swapping): the imaginary wall between the two subarrays move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.

- An array of *n* elements requires *n*-1 iterations to completely sort the array.

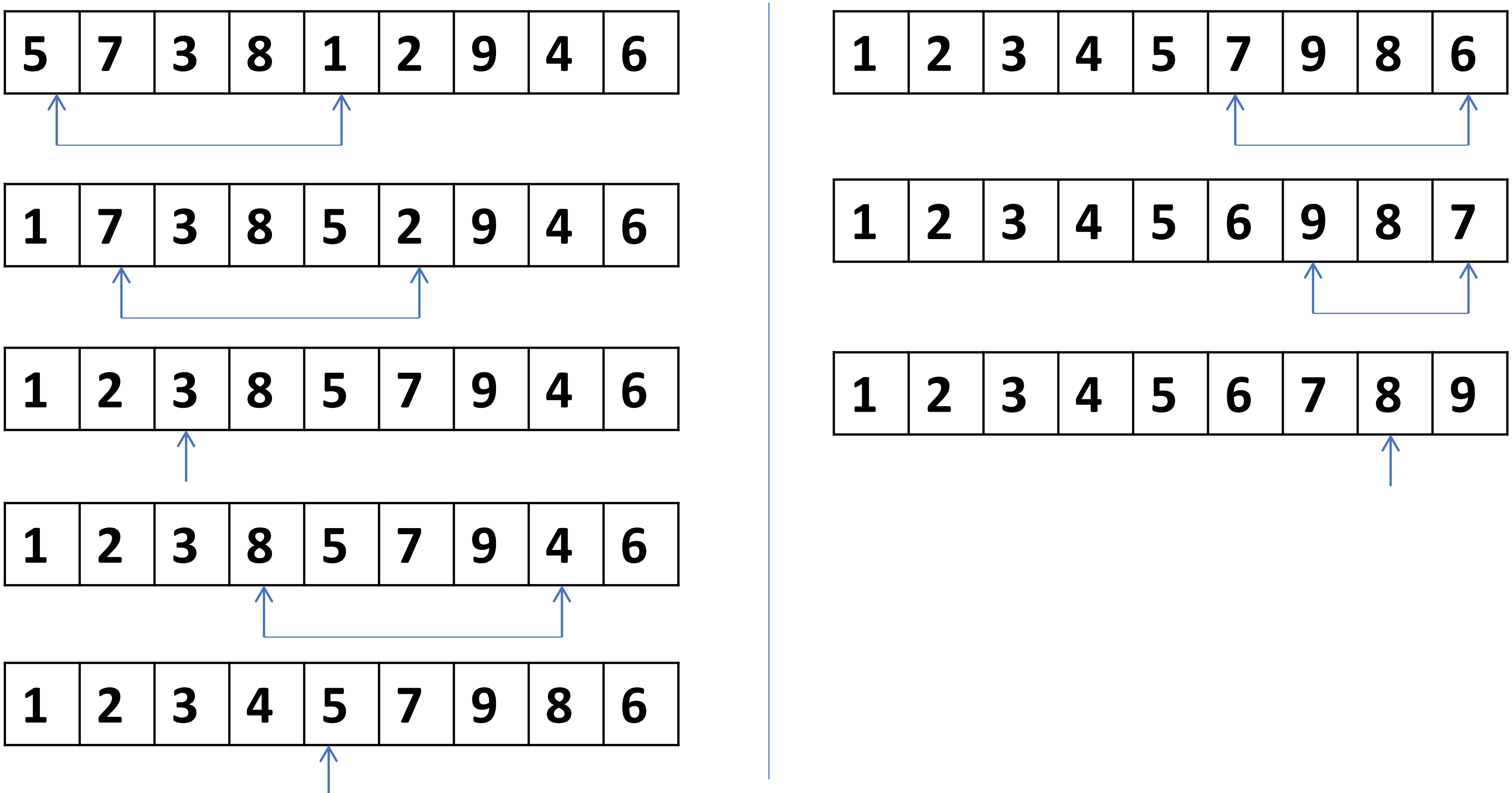
Selection Sort

- Running time
  - Worst case: O(n^2)
  - Best case: O(n^2)

```
void selectionSort(int A[], int n) {
  // index tu 1 -> n
  for(int k = 1; k <= n; k++){
    int min = k;
    for(int j = k+1; j <= n; j++){
      if(A[min] > A[j]) min = j;
    }
    int tmp = A[min];
    A[min] = A[k];
    A[k] = tmp;
  }
}
```

Selection Sort

- Example: 5, 7, 3, 8, 1, 2, 9, 4, 6



Bubble Sort

- Traverse the sequence from left to right
  - Swap two adjacent elements if they are not in the right order
- Repeat that traversal if the previous step has swaps
- Running time
  - Worst case: $O(n^2)$
  - Best case: $O(n)$

```
void bubleSort(int A[], int n) {
  // index tu 1 den n
  int swapped;
  do{
    swapped = 0;
    for(int i = 1; i < n; i++){
      if(A[i] > A[i+1]){
        int tmp = A[i];
        A[i] = A[i+1];
        A[i+1] = tmp;
        swapped = 1;
      }
    }
  }while(swapped == 1);
}
```

Bubble Sort

- Example: 5, 7, 3, 8, 1, 2, 9, 4, 6

5	3	7	1	2	8	4	6	9

3	5	1	2	7	4	6	8	9

3	1	2	5	4	6	7	8	9

1	2	3	4	5	6	7	8	9

Merge sort

- Problem: Sorting *n* elements of an array S[0]…S[*n*-1]
- Merge-sort on an input sequence ***S*** with ***n*** elements consists of three steps:
  - Divide: partition ***S*** into two sequences S_1 and S_2 of about ***n***/2 elements each
  - Conquer: recursively sort S_1 and S_2 by using merge sort
  - Combine: merge S_1 and S_2 into a unique sorted sequence

```
void mergeSort(int A[], int L, int R) {
  if(L < R){
    int M = (L+R)/2;
    mergeSort(A,L,M);
    mergeSort(A,M+1,R);
    merge(A,L,M,R);
  }
}
```

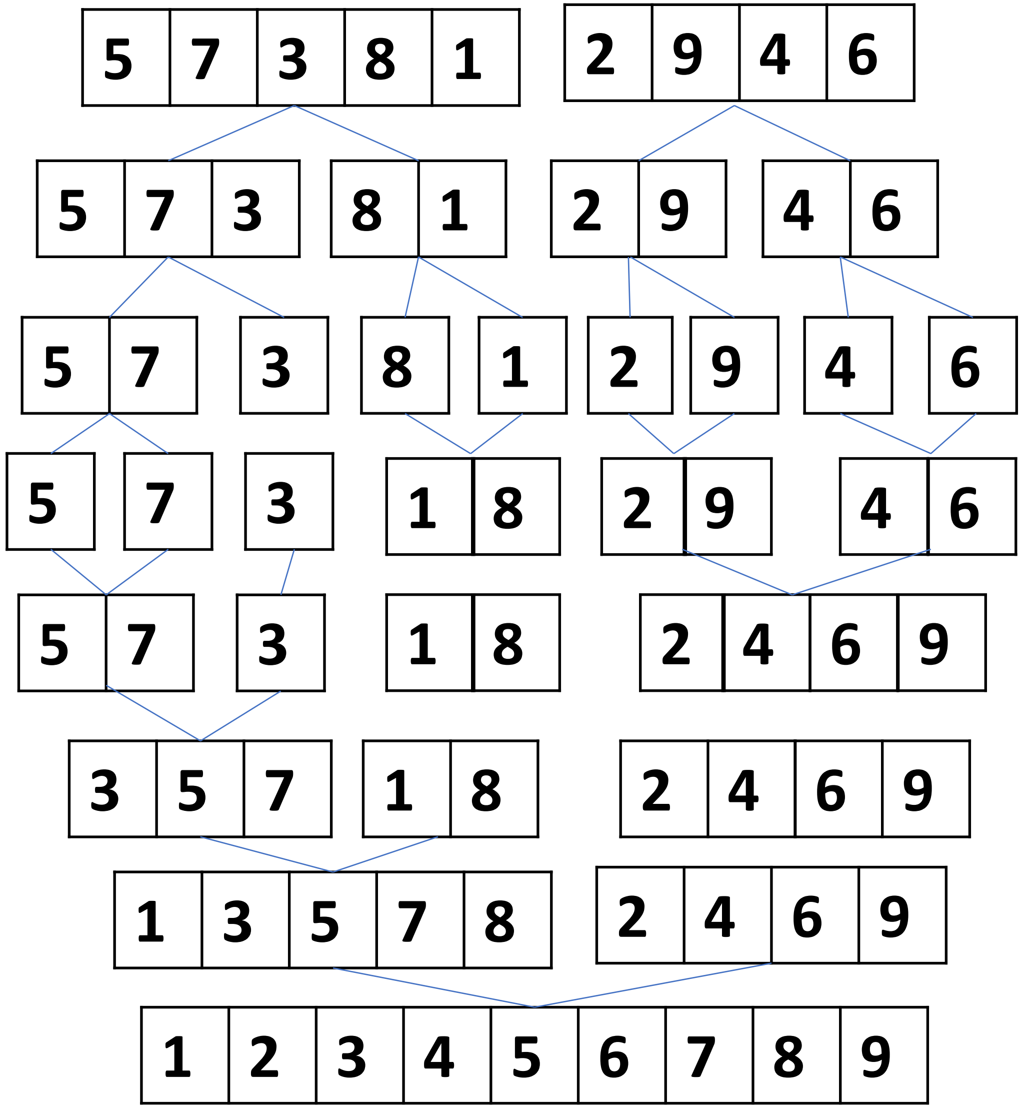
Merge sort

- Use auxiliary array
- Running time
  - Worst case O(*n*log*n*)
  - Best case: O(*n*log*n*)

```
void merge(int A[], int L, int M, int R) {
// tron 2 day da sap A[L..M] va A[M+1..R]
  int i = L; int j = M+1;
  for(int k = L; k <= R; k++){
    if(i > M){ TA[k] = A[j]; j++;}
    else if(j > R){TA[k] = A[i]; i++;}
    else{
      if(A[i] < A[j]){
         TA[k] = A[i]; i++;
      }
      else {
         TA[k] = A[j]; j++;
      }
    }
  }
  for(int k = L; k <= R; k++) A[k] = TA[k];
}
```

Merge sort

- Example: 5, 7, 3, 8, 1, 2, 9, 4, 6



Quick sort

The quick sort algorithm is described recursively as following (similar to merge sort):

1. **Base case**. If the array has only one element, then the array is sorted already, return it without doing anything.
2. **Divide**:
   - Select an element in the array, and call it as the pivot ***p***.
   - Divide the array into 2 subarrays: Left subarray (L) consists of elements $\leq$ the pivot, right subarray (R) consists of elements $\geq$ the pivot. This operation is called "Partition".
3. **Conquer**: recursively call QuickSort for 2 subarrays $L = A[p \ldots q]$ and $R = A[q+1 \ldots r]$.
4. **Combine**: The sorted array is $L\ p\ R$.

In contrast to Merge Sort, in Quick Sort: division operation is complicate, but the Partition operation is simple.

Quick sort

```
void quickSort(int A[], int L, int R) {
  if(L < R){
    int index = (L + R)/2;
    index = partition(A, L, R, index);
    if(L < index)
      quickSort(A, L, index-1);
    if(index < R)
      quickSort(A, index+1, R);
  }
}
```

```
int partition(int A[], int L, int R, int
                             indexPivot) {
  int pivot = A[indexPivot];
  swap(A[indexPivot], A[R]);
  int storeIndex = L;
  for(int i = L; i <= R-1; i++){
    if(A[i] < pivot){
      swap(A[storeIndex], A[i]);
      storeIndex++;
    }
  }
  swap(A[storeIndex], A[R]);
  return storeIndex;
}
```

Quick sort

- Example: 5, 7, 3, 8, 1, 2, 9, 4, 6

5	7	3	8	1	2	9	4	6

$\xrightarrow{\text{Pivot = 5}}$

3	1	2	4	5	6	9	8	7

3	1	2	4

$\xrightarrow{\text{Pivot = 3}}$

1	2	3	4

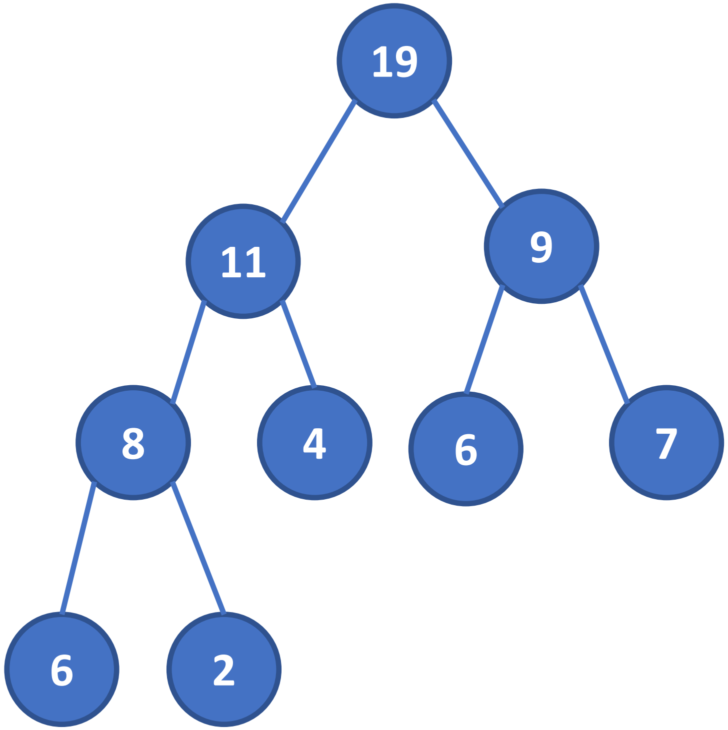
6	9	8	7

$\xrightarrow{\text{Pivot = 7}}$

6	7	8	9

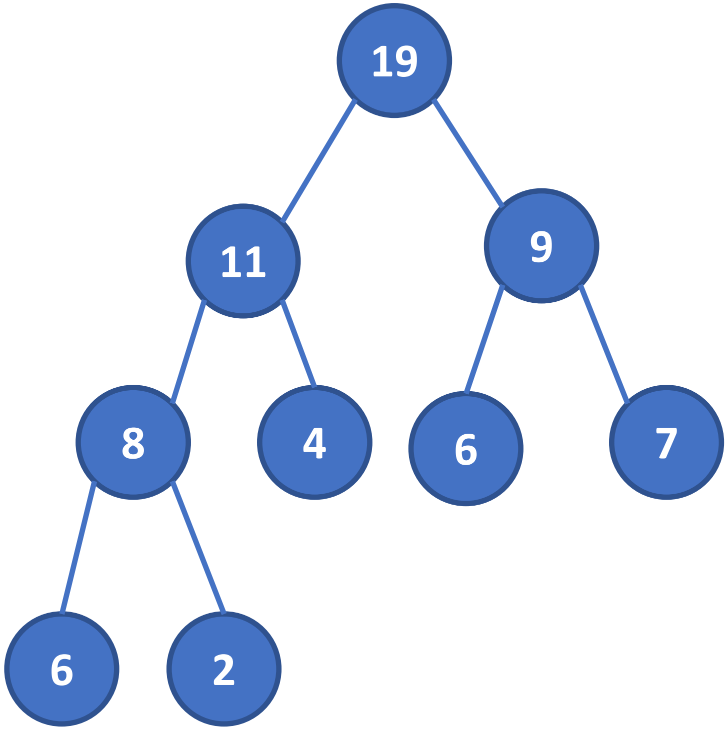
Heap sort

- Heap structure (max-heap)
  - Complete tree
  - Key of each node is greater or equal to the keys of its children (max-heap property)
- Map the sequence A[1…N] to a complete tree
  - Root is A[1]
  - A[2i] and A[2i+1] are respectively the left child and the right child of A[i]
  - The height of the tree is log*N* + 1



Heap sort

- Heapify
  - Status:
    - max-heap property at A[i] is destroyed
    - max-heap property at children of A[i] is satisfied
  - Adjust the tree to recover the max-heap property at the root A[i]



Heap sort

- Running time: O(log*n*)

```
void heapify(int A[], int i, int n) {
  int L = 2*i;
  int R = 2*i+1;
  int max = i;
  if(L <= n && A[L] > A[i])
    max = L;
  if(R <= n && A[R] > A[max])
    max = R;
  if(max != i){
    swap(A[i], A[max]);
    heapify(A,max,n);
  }
}
```

Heap sort

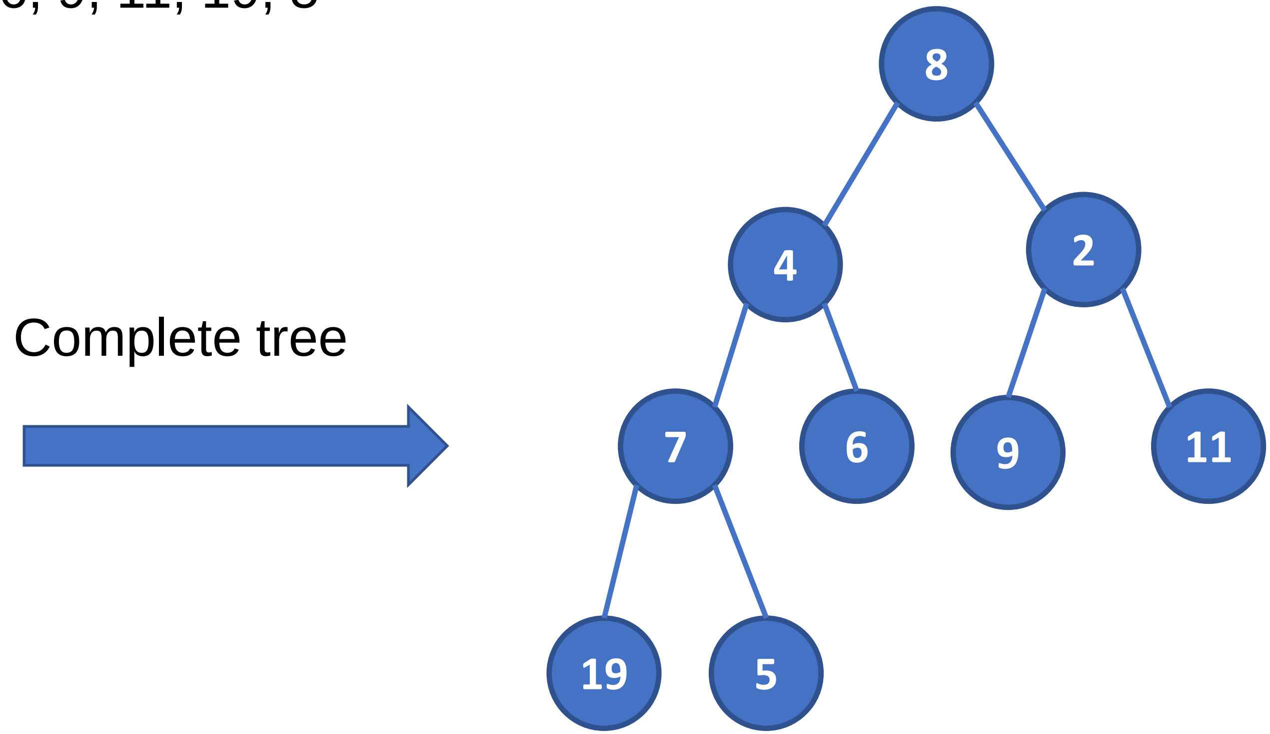
- Heap sort
  - Build max-heap
  - Swap A[1] and A[n]
  - Heapify at A[1] until A[1..n-1]
  - Swap A[1] and A[n-1]
  - Heapify at A[1] until A[1..n-2]
  - …

```
void buildHeap(int A[], int n) {
  for(int i = n/2; i >= 1; i--)
    heapify(A,i,n);
}
void heapSort(int A[], int n) {
  // index tu 1 -> n
  buildHeap(A,n);
  for(int i = n; i > 1; i--) {
    swap(A[1], A[i]);
    heapify(A, 1, i-1);
  }
}
```

Heap sort

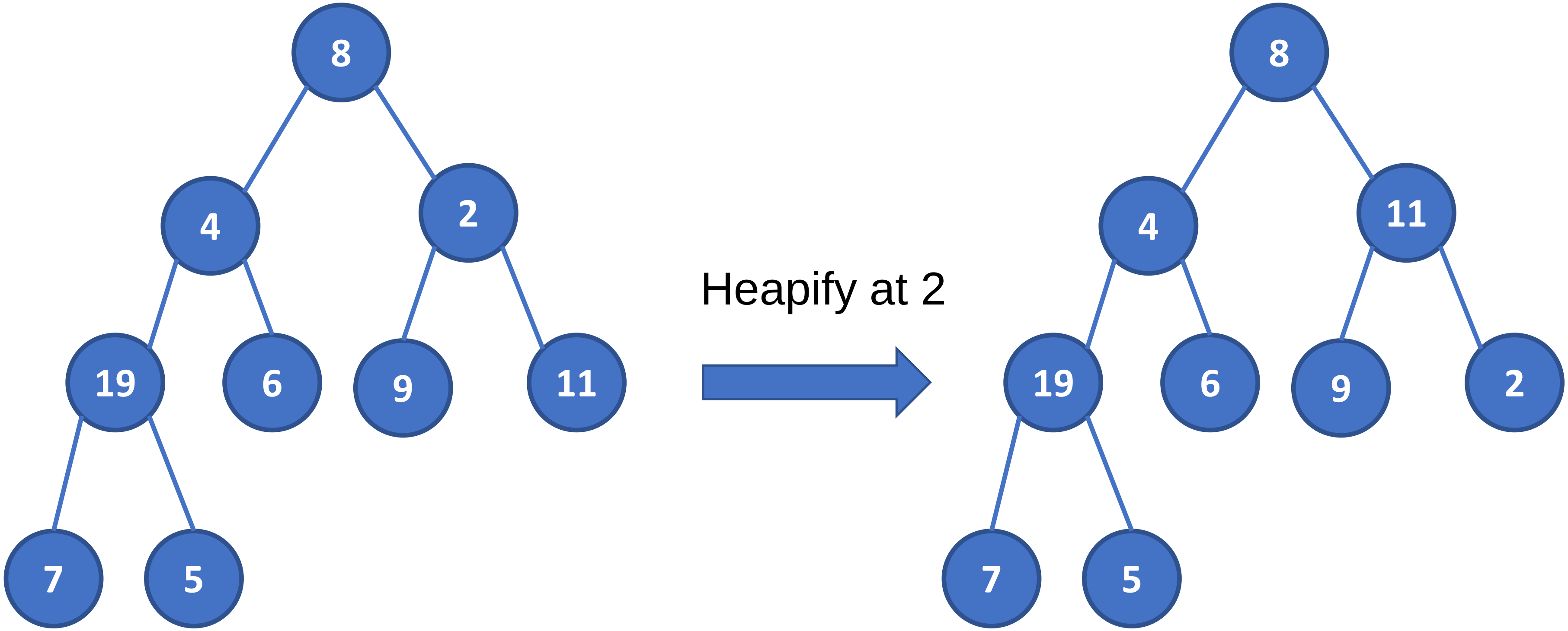
- Example: perform heap sort over the sequence: 8, 4, 2, 7, 6, 9, 11, 19, 5

Complete tree



8, 4, 2, 7, 6, 9, 11, 19, 5

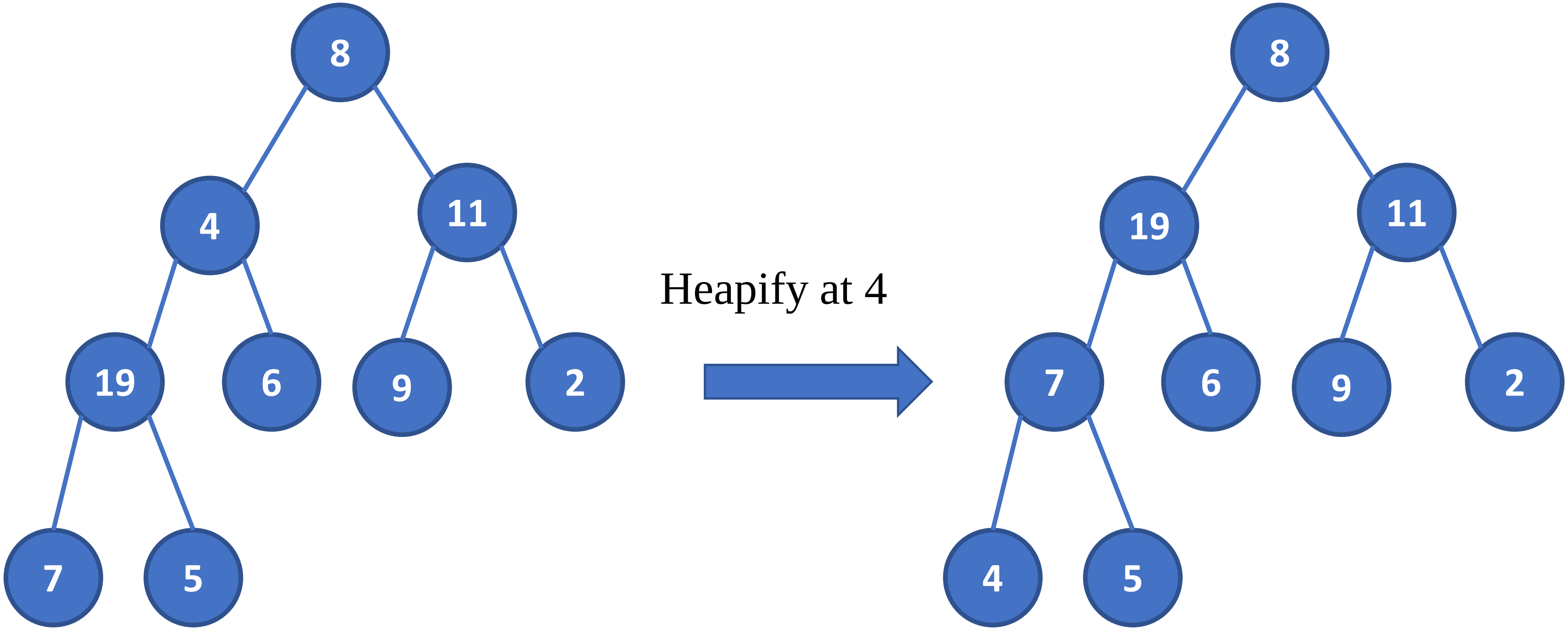
Heap sort



8, 4, 2, 19, 6, 9, 11, 7, 5

8, 4, 11, 19, 6, 9, 2, 7, 5

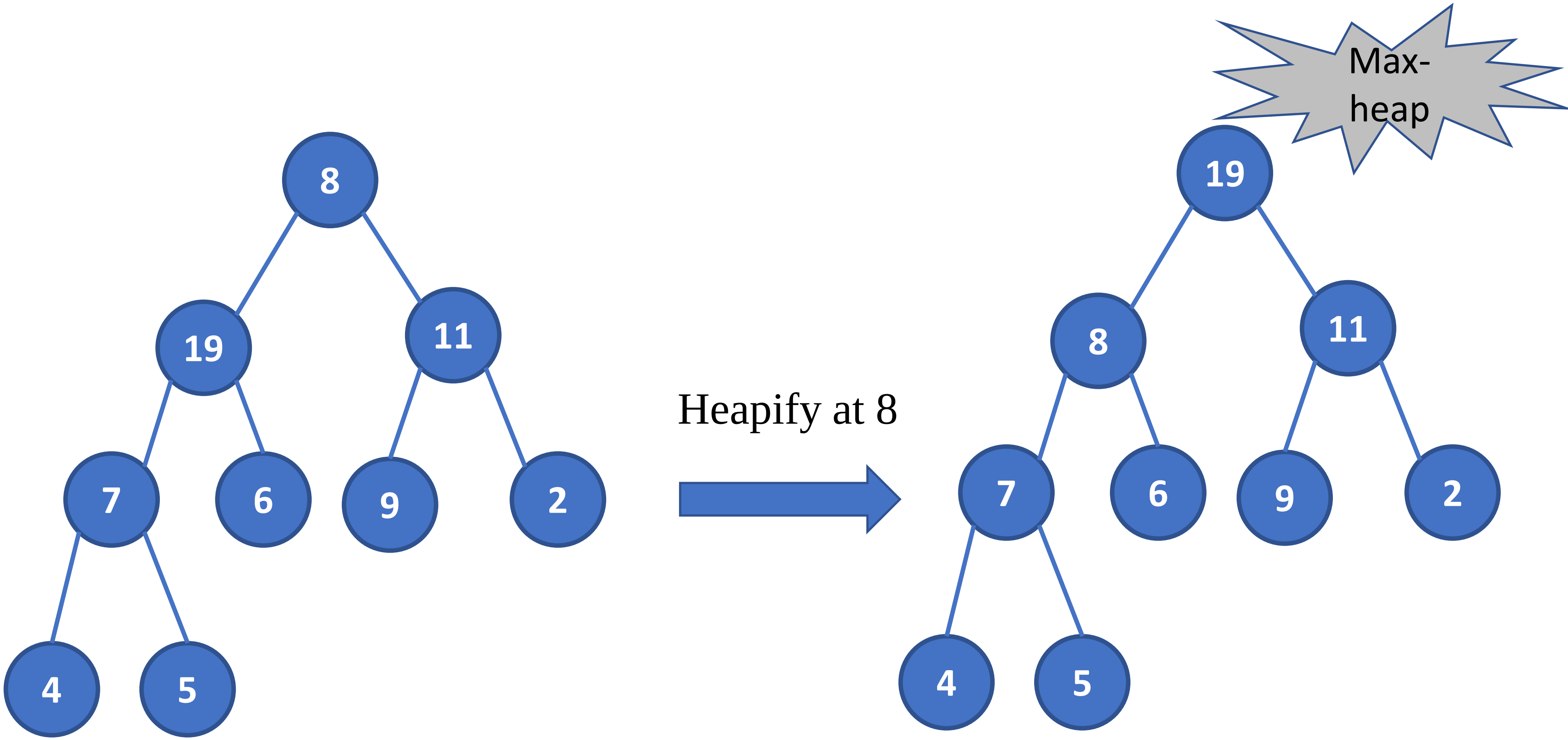
Heap sort



8, 4, 11, 19, 6, 9, 2, 7, 5

8, 19, 11, 7, 6, 9, 2, 4, 5

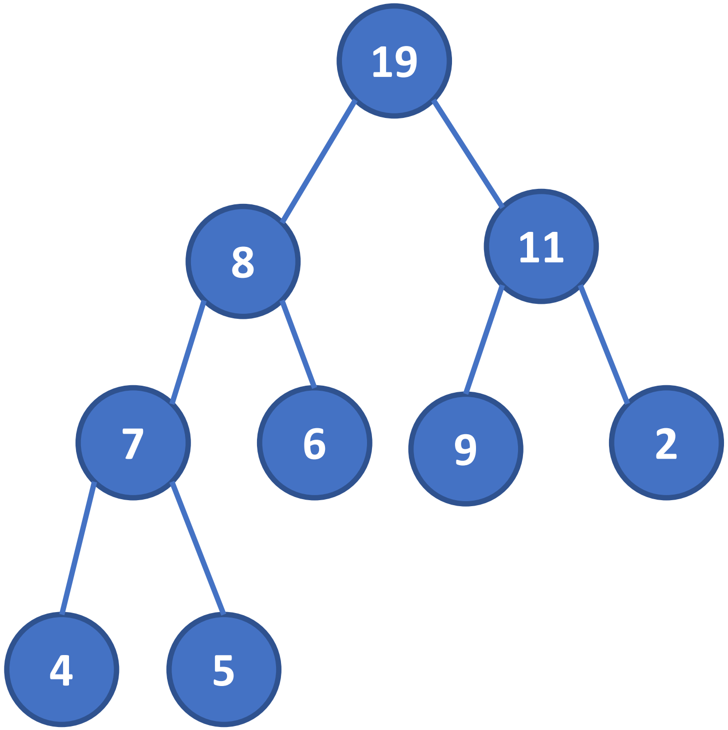
Heap sort



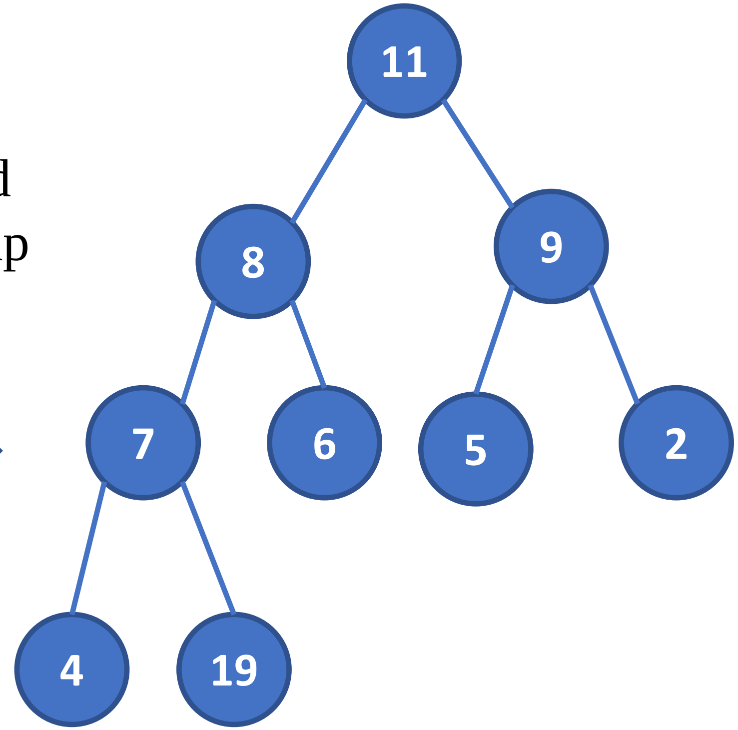
8, 19, 11, 7, 6, 9, 2, 4, 5

19, 8, 11, 7, 6, 9, 2, 4, 5

Heap sort



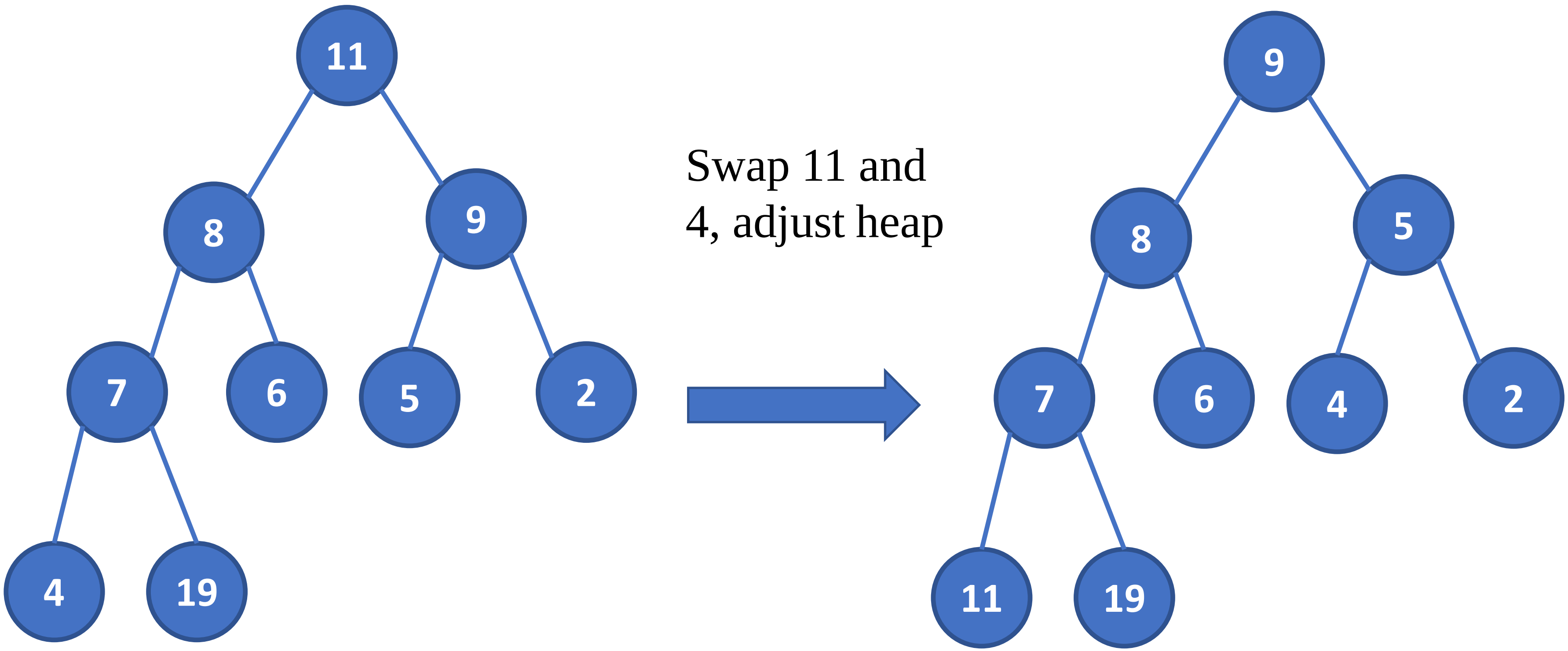
Swap 19 and
5, adjust heap



19, 8, 11, 7, 6, 9, 2, 4, 5

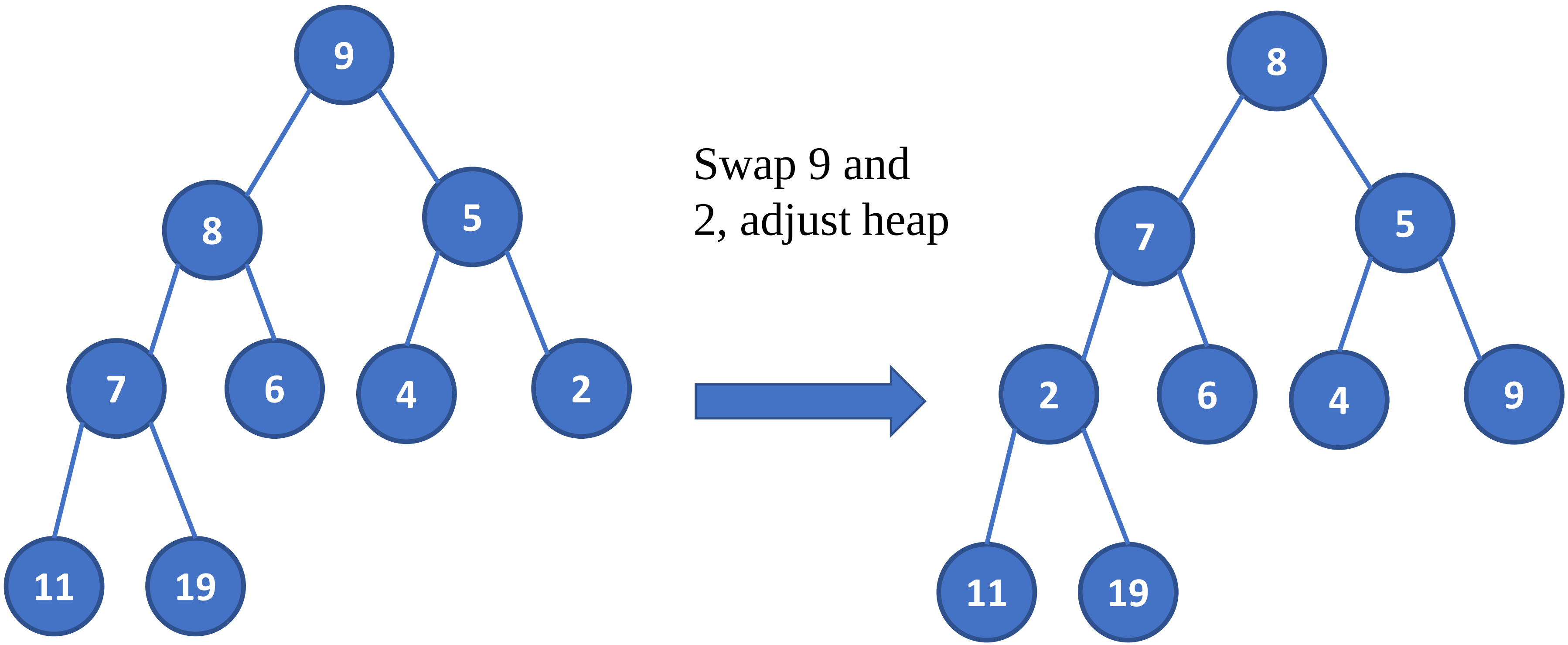
11, 8, 9, 7, 6, 5, 2, 4, 19

Heap sort



11, 8, 9, 7, 6, 5, 2, 4, 19 **9, 8, 5, 7, 6, 4, 2, 11, 19**

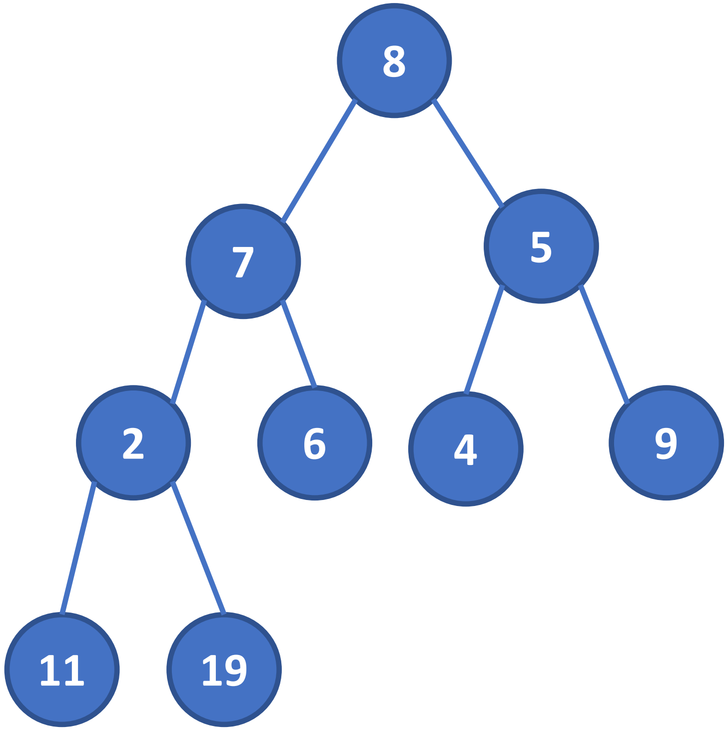
Heap sort



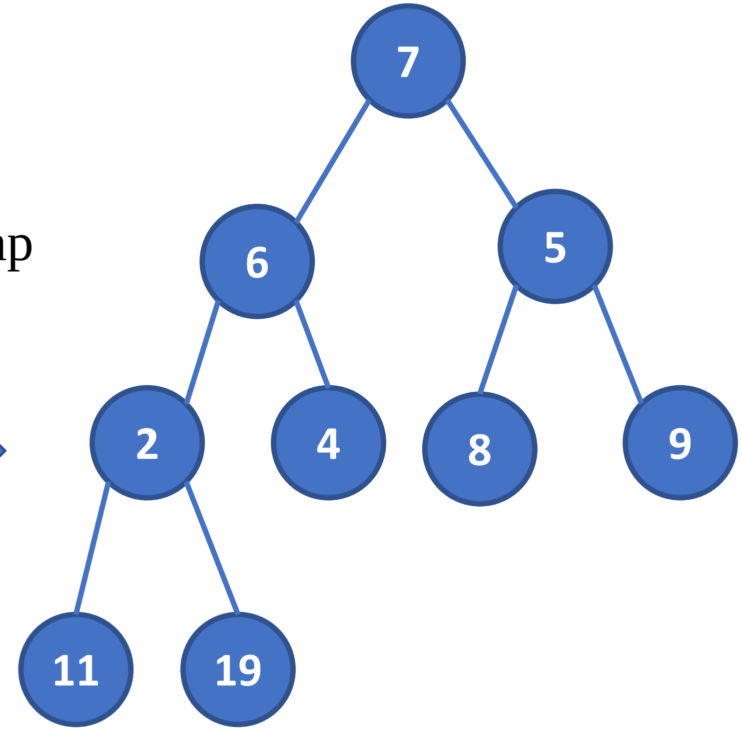
9, 8, 5, 7, 6, 4, 2, 11, 19

8, 7, 5, 2, 6, 4, 9, 11, 19

Heap sort



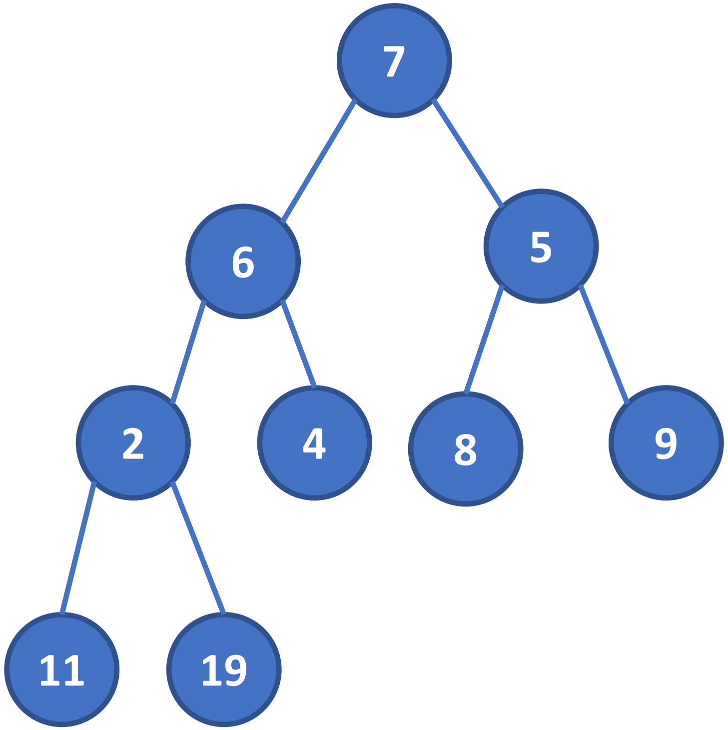
Swap 8 and
4, adjust heap



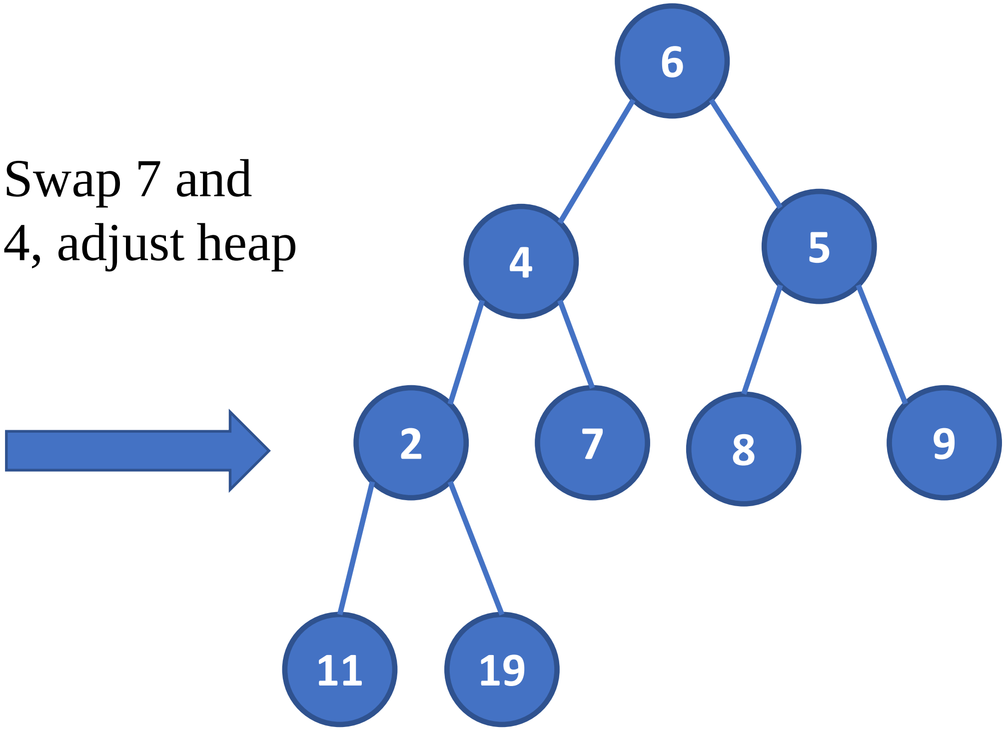
8, 7, 5, 2, 6, 4, 9, 11, 19

7, 6, 5, 2, 4, 8, 9, 11, 19

Heap sort



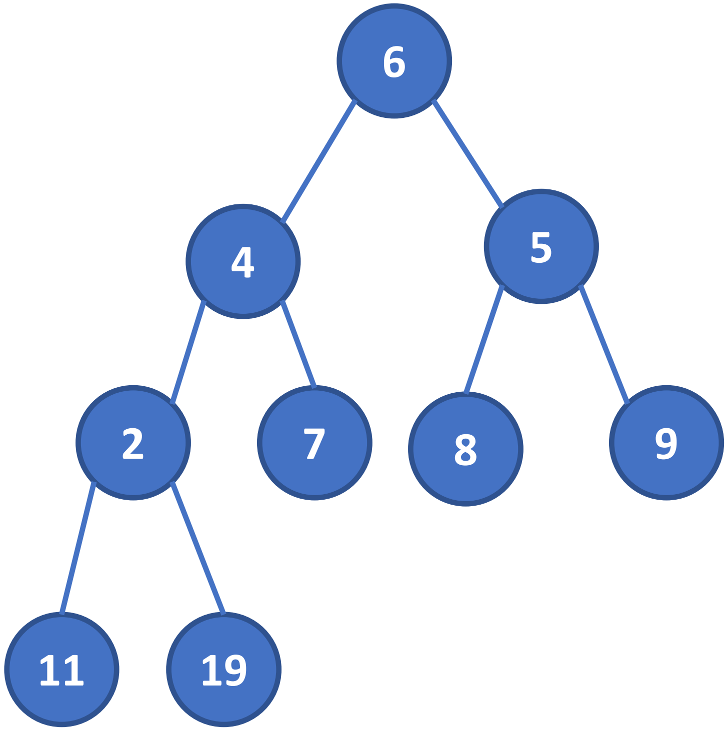
Swap 7 and 4, adjust heap



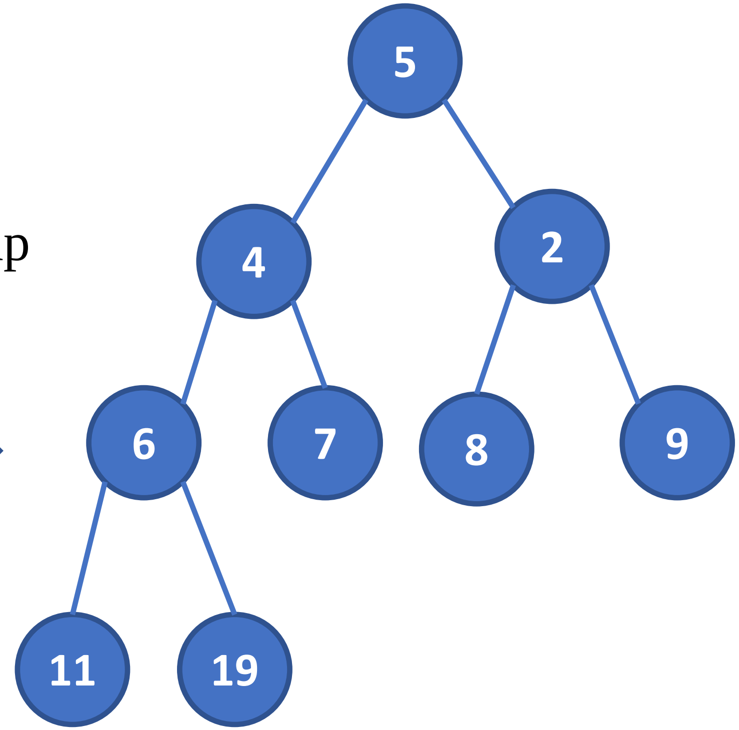
7, 6, 5, 2, 4, 8, 9, 11, 19

6, 4, 5, 2, 7, 8, 9, 11, 19

Heap sort



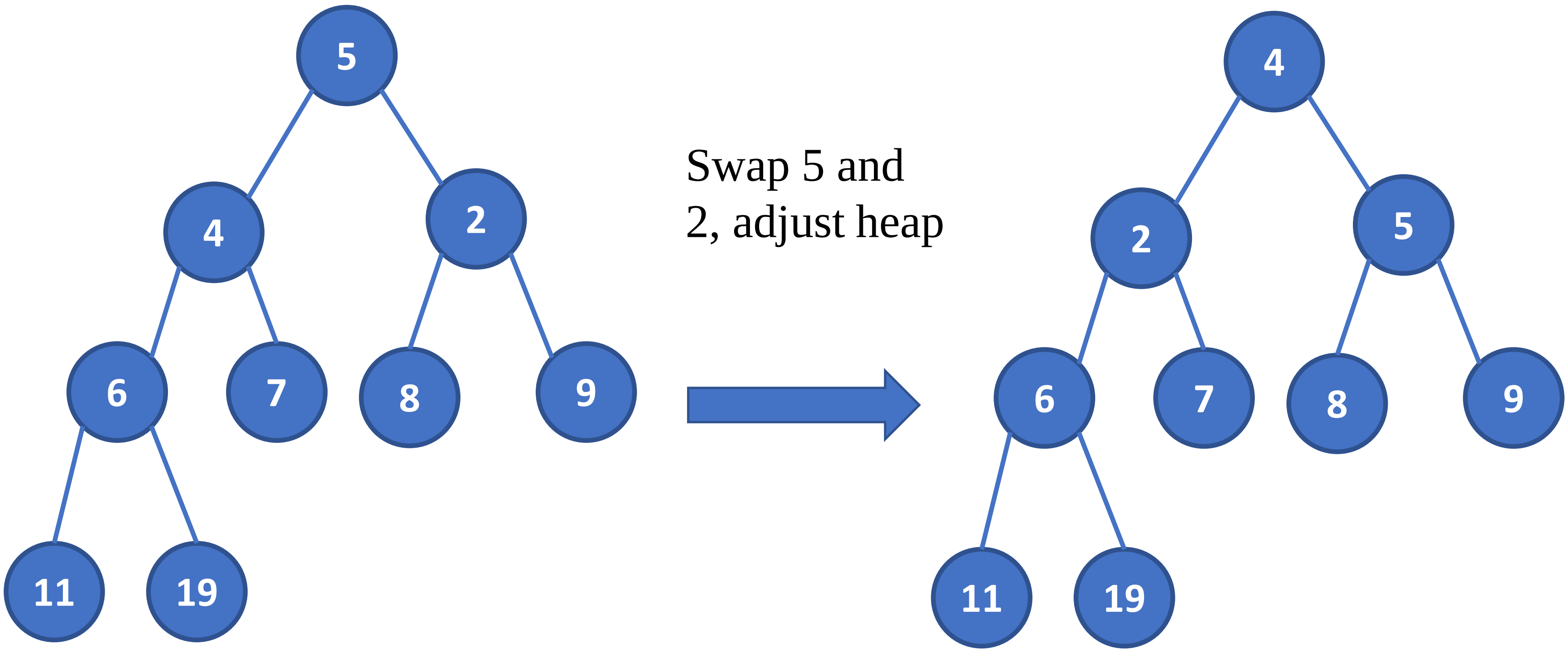
Swap 6 and 2, adjust heap



6, 4, 5, 2, 7, 8, 9, 11, 19

5, 4, 2, 6, 7, 8, 9, 11, 19

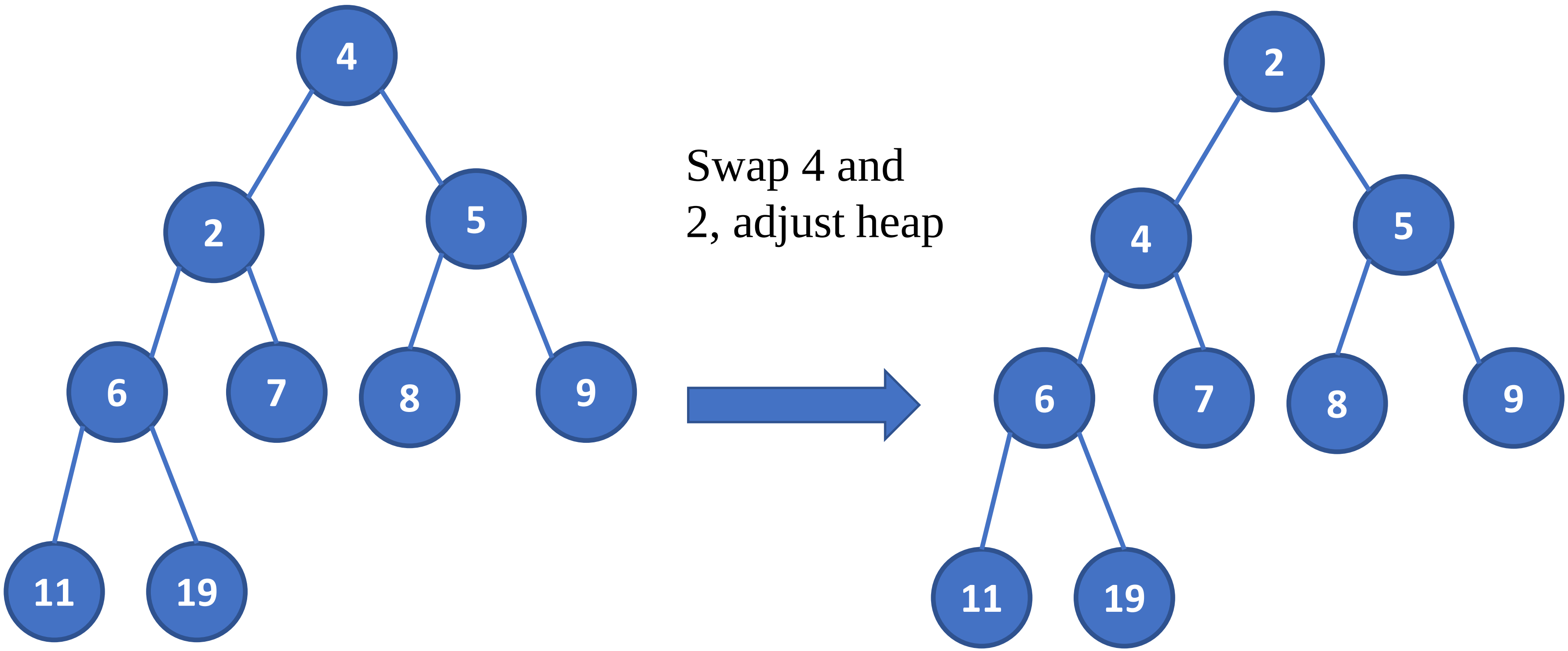
Heap sort



5, 4, 2, 6, 7, 8, 9, 11, 19

4, 2, 5, 6, 7, 8, 9, 11, 19

Heap sort



4, 2, 5, 6, 7, 8, 9, 11, 19

2, 4, 5, 6, 7, 8, 9, 11, 19