

# DATA STRUCTURE AND ALGORITHM

## CLASS 11

---

Seongjin Lee

Updated: 2021-02-27  
DSA\_2017\_11

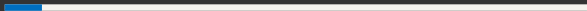
insight@gnu.ac.kr  
<http://resourceful.github.io>  
Systems Research Lab.  
GNU



# Table of contents

1. Sorting
2. Bubble sort
3. Selection sort
4. Insertion sort
5. Shell Sort
6. Quick Sort
7. Heap Sort
8. Radix Sort

# SORTING



# Sorting

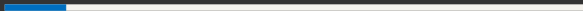
## Types of Sorting

- Internal Sorting - The list is small enough to sort entirely in main memory
- External Sorting - The list is too large to sort in main memory

## Internal sorting algorithms

- Bubble sort
- Selection sort
- Insertion sort
- Quick sort
- Shell sort
- Heap sort
- Radix sort

# BUBBLE SORT



# Bubble sort

```
1 repeat until no swaps
2     for i from 0 to n-2
3         if ith and i+1th elements out of order
4             swap them
```

# Bubble sort: example

initial	15	4	8	3	50	9	20	unsorted array
1 <sup>st</sup> run	4	15	8	3	50	9	20	cmp 1 and 2, swap

# Bubble sort: example

initial	15	4	8	3	50	9	20	unsorted array
1 <sup>st</sup> run	4	15	8	3	50	9	20	cmp 1 and 2, swap
1 <sup>st</sup> run	4	8	15	3	50	9	20	cmp 2 and 3, swap



# Bubble sort: example

initial	15	4	8	3	50	9	20	unsorted array
1 <sup>st</sup> run	4	15	8	3	50	9	20	cmp 1 and 2, swap
1 <sup>st</sup> run	4	8	15	3	50	9	20	cmp 2 and 3, swap
1 <sup>st</sup> run	4	8	3	15	50	9	20	cmp 3 and 4, swap

# Bubble sort: example

initial	15	4	8	3	50	9	20	unsorted array
1 <sup>st</sup> run	4	15	8	3	50	9	20	cmp 1 and 2, swap
1 <sup>st</sup> run	4	8	15	3	50	9	20	cmp 2 and 3, swap
1 <sup>st</sup> run	4	8	3	15	50	9	20	cmp 3 and 4, swap
1 <sup>st</sup> run	4	8	3	15	50	9	20	cmp 4 and 5, no swap

# Bubble sort: example

initial	15	4	8	3	50	9	20	unsorted array
1 <sup>st</sup> run	4	15	8	3	50	9	20	cmp 1 and 2, swap
1 <sup>st</sup> run	4	8	15	3	50	9	20	cmp 2 and 3, swap
1 <sup>st</sup> run	4	8	3	15	50	9	20	cmp 3 and 4, swap
1 <sup>st</sup> run	4	8	3	15	50	9	20	cmp 4 and 5, no swap
1 <sup>st</sup> run	4	8	3	15	9	50	20	cmp 5 and 6, swap

# Bubble sort: example

initial	15	4	8	3	50	9	20	unsorted array
1 <sup>st</sup> run	4	15	8	3	50	9	20	cmp 1 and 2, swap
1 <sup>st</sup> run	4	8	15	3	50	9	20	cmp 2 and 3, swap
1 <sup>st</sup> run	4	8	3	15	50	9	20	cmp 3 and 4, swap
1 <sup>st</sup> run	4	8	3	15	50	9	20	cmp 4 and 5, no swap
1 <sup>st</sup> run	4	8	3	15	9	50	20	cmp 5 and 6, swap
1 <sup>st</sup> run	4	8	3	15	9	20	50	cmp 6 and 7, swap

# Bubble sort: example

initial	15	4	8	3	50	9	20	unsorted array
1 <sup>st</sup> run	4	15	8	3	50	9	20	cmp 1 and 2, swap
1 <sup>st</sup> run	4	8	15	3	50	9	20	cmp 2 and 3, swap
1 <sup>st</sup> run	4	8	3	15	50	9	20	cmp 3 and 4, swap
1 <sup>st</sup> run	4	8	3	15	50	9	20	cmp 4 and 5, no swap
1 <sup>st</sup> run	4	8	3	15	9	50	20	cmp 5 and 6, swap
1 <sup>st</sup> run	4	8	3	15	9	20	50	cmp 6 and 7, swap
2 <sup>nd</sup> run	4	8	3	15	9	20	50	cmp 1 and 2, no swap
2 <sup>nd</sup> run	4	3	8	15	9	20	50	cmp 2 and 3, swap
2 <sup>nd</sup> run	4	3	8	15	9	20	50	cmp 3 and 4, no swap
2 <sup>nd</sup> run	4	3	8	9	15	20	50	cmp 4 and 5, swap
2 <sup>nd</sup> run	4	3	8	9	15	20	50	cmp 5 and 6, no swap
3 <sup>rd</sup> run	3	4	8	9	15	20	50	1, 2 swap
4 <sup>th</sup> run	3	4	8	9	15	20	50	no swap
5 <sup>th</sup> run	3	4	8	9	15	20	50	no swap
6 <sup>th</sup> run	3	4	8	9	15	20	50	no swap

# Bubble sort: the code (codes/bubble.c) I

```
1 // Unoptimized and Optimized implementation of bubble sort
2 #include <stdio.h>
3
4 void swap(int *x, int *y);
5 void bubble_unoptimized(int list[], int size);
6 void bubble(int list[], int size);
7 void printArray(int list[], int size);
8
9 int main(){
10     int list[] = {30, 28, 18, 29, 48, 40};
11     // get the number of items in the list
12     int n = sizeof(list)/sizeof(list[0]);
13
14     printf("initial list: \t");
15     printArray(list, n);
```

# Bubble sort: the code (codes/bubble.c) II

```
16
17 // bubble_unoptimized(list, n);
18     bubble(list, n);
19
20     printf("sorted list: \t");
21     printArray(list, n);
22
23     return 0;
24 }
25
26 void swap(int *x, int *y){ // swap the two value
27     int temp = *x;
28     *x = *y;
29     *y = temp;
30 }
31
```

# Bubble sort: the code (codes/bubble.c) III

```
32
33 void printArray(int list[], int size){
34     for ( int i = 0; i < size; i++)
35         printf("%d ", list[i]);
36     printf("\n");
37 }
38
39
40 // unoptimized bubble sort with worst case of  $O(n^2)$ 
41 void bubble_unoptimized(int list[], int size){
42     int i, j;
43     for (i = 0; i < size-1; i++)
44         for (j = 0; j < size-i-1; j++) // skip sorted items in the list
45             if (list[j] > list[j+1])
46                 swap(&list[j], &list[j+1]);
47 }
```



# Bubble sort: the code (codes/bubble.c) IV

```
48
49 // optimized bubble sort
50 void bubble(int list[], int size){
51     int i, j;
52     int swapped = 1;
53     // loop until no two elements were swapped by inner loop
54     for (i = size-1; swapped > 0; i--){
55         swapped = 0;
56         for (j = 0; j < i; j++){
57             if (list[j] > list[j+1]){
58                 swap(&list[j], &list[j+1]);
59                 swapped = 1;
60             }
61         }
62     }
```

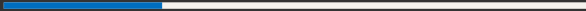
# Bubble sort: Time complexity

loop	number of comparison
1	n-1
2	n-2
3	n-3
.	.
.	.
.	.
n-1	1

Time complexity:  $O(n^2)$

<https://www.youtube.com/watch?v=lyZQPjUT5B4&t=50>

# SELECTION SORT



# Selection sort

```
1 for i from 0 to n-1
2     find smallest element between ith and n-1th
3     swap smallest with ith element
```

# Selection sort

initial	15	4	8	3	50	9	20	unsorted array
1 <sup>st</sup> run	3	4	8	15	50	9	20	midx 15, min 3, swap

# Selection sort

initial	15	4	8	3	50	9	20	unsorted array
1 <sup>st</sup> run	3	4	8	15	50	9	20	midx 15, min 3, swap
2 <sup>nd</sup> run	3	4	8	15	50	9	20	midx 4, min 4

# Selection sort

initial	15	4	8	3	50	9	20	unsorted array
1 <sup>st</sup> run	3	4	8	15	50	9	20	midx 15, min 3, swap
2 <sup>nd</sup> run	3	4	8	15	50	9	20	midx 4, min 4
3 <sup>rd</sup> run	3	4	8	15	50	9	20	midx 8, min 8

# Selection sort

initial	15	4	8	3	50	9	20	unsorted array
1 <sup>st</sup> run	3	4	8	15	50	9	20	midx 15, min 3, swap
2 <sup>nd</sup> run	3	4	8	15	50	9	20	midx 4, min 4
3 <sup>rd</sup> run	3	4	8	15	50	9	20	midx 8, min 8
4 <sup>th</sup> run	3	4	8	9	50	15	20	midx 15, min 9, swap



# Selection sort

initial	15	4	8	3	50	9	20	unsorted array
1 <sup>st</sup> run	3	4	8	15	50	9	20	midx 15, min 3, swap
2 <sup>nd</sup> run	3	4	8	15	50	9	20	midx 4, min 4
3 <sup>rd</sup> run	3	4	8	15	50	9	20	midx 8, min 8
4 <sup>th</sup> run	3	4	8	9	50	15	20	midx 15, min 9, swap
4 <sup>th</sup> run	3	4	8	9	15	50	20	midx 50, min 15, swap

# Selection sort

initial	15	4	8	3	50	9	20	unsorted array
1 <sup>st</sup> run	3	4	8	15	50	9	20	midx 15, min 3, swap
2 <sup>nd</sup> run	3	4	8	15	50	9	20	midx 4, min 4
3 <sup>rd</sup> run	3	4	8	15	50	9	20	midx 8, min 8
4 <sup>th</sup> run	3	4	8	9	50	15	20	midx 15, min 9, swap
4 <sup>th</sup> run	3	4	8	9	15	50	20	midx 50, min 15, swap
5 <sup>th</sup> run	3	4	8	9	15	20	50	midx 50, min 20, swap

## Selection sort: the Code (codes/selection.c)

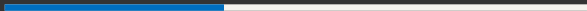
```
38 // selection sort
39 void selection(int list[], int size){
40     int i, j, midx;
41     for (i = 0; i < size-1 ; i++){
42         // find the minimum element in unsorted list
43         midx= i;
44         for (j = i+1; j < size; j++)
45             if (list[j] < list[midx])
46                 midx = j;
47
48         // swap the minimum element with the first element
49         swap(&list[midx], &list[i]);
50     }
51 }
```

# Select sort: Time complexity

loop	number of comparison
1	n-1
2	n-2
3	n-3
.	.
.	.
.	.
n-1	1

Time complexity:  $O(n^2)$

# INSERTION SORT



# Insertion sort

```
1 for i from 1 to n-1
2     call 0th through i-1th elements the ‘‘sorted side’’
3     remove ith element
4     insert it into sorted side in order
```

# Insertion sort

index	1	2	3	4	5	
initial state	36	28	30	17	23	
1 <sup>st</sup> run	28	36	30	17	23	28 is smaller than 36, move 28 to the sorted side

# Insertion sort

index	1	2	3	4	5	
initial state	36	28	30	17	23	
1 <sup>st</sup> run	28	36	30	17	23	28 is smaller than 36, move 28 to the sorted side
2 <sup>nd</sup> run	28	30	36	17	23	30 is smaller than 36, move it to the sorted side



# Insertion sort

index	1	2	3	4	5	
initial state	36	28	30	17	23	
1 <sup>st</sup> run	28	36	30	17	23	28 is smaller than 36, move 28 to the sorted side
2 <sup>nd</sup> run	28	30	36	17	23	30 is smaller than 36, move it to the sorted side
3 <sup>rd</sup> run	17	28	30	36	23	17 is smaller than 36, 30, and 28, move it to the sorted side

# Insertion sort

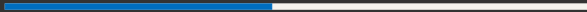
index	1	2	3	4	5	
initial state	36	28	30	17	23	
1 <sup>st</sup> run	28	36	30	17	23	28 is smaller than 36, move 28 to the sorted side
2 <sup>nd</sup> run	28	30	36	17	23	30 is smaller than 36, move it to the sorted side
3 <sup>rd</sup> run	17	28	30	36	23	17 is smaller than 36, 30, and 28, move it to the sorted side
4 <sup>th</sup> run	17	23	28	30	36	23 is smaller than 36, 30, and 28, move it to the sorted side

# Insertion sort: the Code (codes/insertion.c)

```
29 // insertion sort
30 void insertion(int list[], int size){
31     int i, j, key;
32     for (i = 1; i < size; i++){
33         key = list[i];
34
35         // if item is greater than the key
36         // move the item in list to one position ahead
37         for (j = i-1 ; j >= 0 && list[j] > key; j--)
38             list[j+1] = list[j];
39         list[j+1] = key;
40     }
41 }
```

time complexity: worst case  $O(n^2)$ , best case  $O(n)$

# SHELL SORT



# Shell sort

Shell sort is based on insertion sort algorithm to reduce the number of shifts. For example, if the list is in decreasing order and we want to sort in increasing order, the far left record has to be moved to the far right, one by one.

<https://www.youtube.com/watch?v=1yDcmjLTWOg>

# Shell sort: the Code (codes/shell.c)

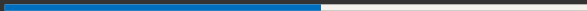
```
34 void shellsort(int list[], int n){
35     int i, gap;
36     for (gap = n/2; gap > 0 ; gap = gap/2){
37         if ( (gap%2) == 0 ) gap++;
38         for (i = 0; i < gap; i++)
39             incremental_insert(list, i, n-1, gap);
40     }
41 }
```

## Shell sort: the Code (codes/shell.c)

```
43 void incremental_insert(int list[], int first, int last, int
    gap){
44     int i, j, key;
45     for (i = first + gap; i <= last; i = i + gap){
46         key = list[i];
47         for ( j = i-gap; j >= first && key < list[j]; j = j - gap
            )
48             list[j + gap] = list[j];
49         list[j + gap] = key;
50     }
51 }
```

time complexity: worst case  $O(n^2)$ , best case  $O(n)$

# QUICK SORT





# Quick Sort

## Divide and Conquer

- two phase
- split and control

using recursion

- stack is needed

Best average time

- $O(n \cdot \log_2 n)$

# Quick sort: The principle

1. Pick a “**pivot**” element
2. “**partition**” the array into *three* parts
  - 1<sup>st</sup> part: all elements in this is less than the pivot
  - 2<sup>nd</sup> part: the pivot itself (only one element)
  - 3<sup>rd</sup> part: all elements in this part is greater than or equal to the pivot
3. Recursively apply quick sort to the first and the third part
  - swap the out of order entries from the first and the third part

# Quick Sort

Initial list 

31	11	41	12	51	90	20	64	53	55	32
----	----	----	----	----	----	----	----	----	----	----

31	11	41	12	51	90	20	64	53	55	32
----	----	----	----	----	----	----	----	----	----	----

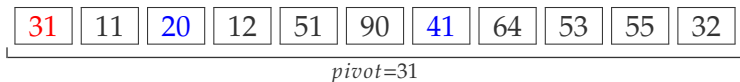
  
*pivot=31*

- examine the first, and last item of the full list that is 31 (first, pivot), and 32 (last)
- search forward until we find an item larger than the pivot  $41 > 31$
- search backward until we find an item smaller than the pivot  $20 < 31$
- swap the two items

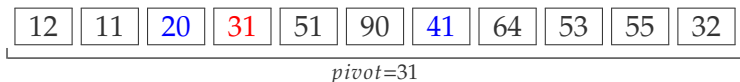
31	11	20	12	51	90	41	64	53	55	32
----	----	----	----	----	----	----	----	----	----	----

  
*pivot=31*

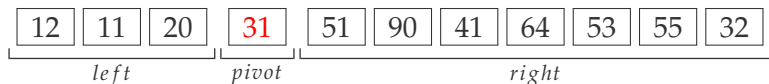
# Quick Sort



- repeat the process until the left and right indices cross each other
- then swap the item with the pivot



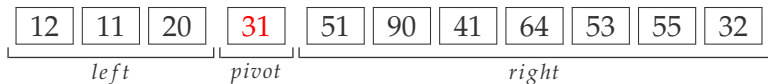
# Quick Sort



- repeat the quick sort for the left and right part



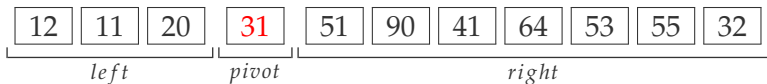
# Quick Sort



- repeat the quick sort for the left and right part



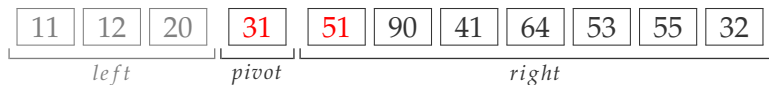
# Quick Sort



- repeat the quick sort for the left and right part

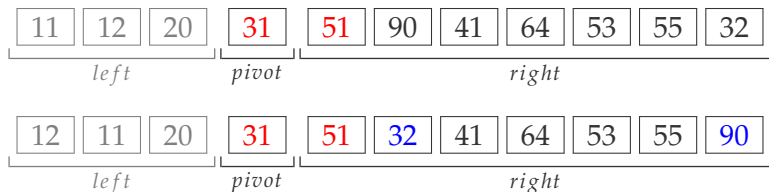


# Quick Sort

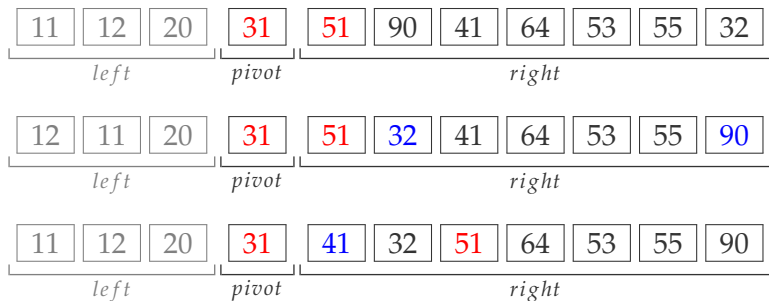




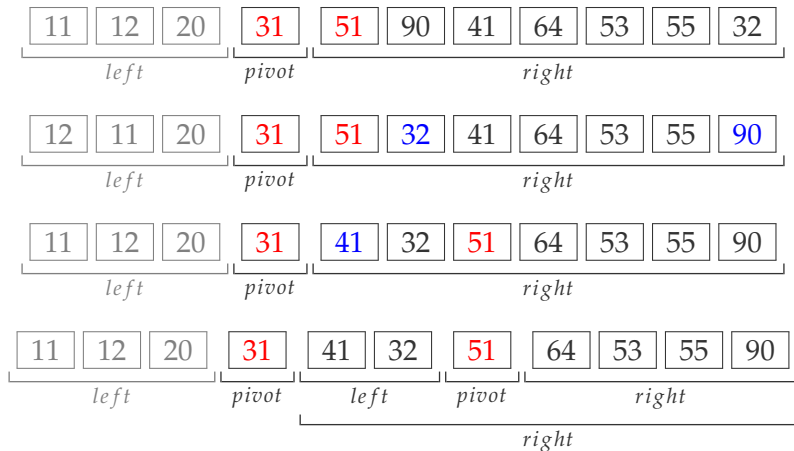
# Quick Sort



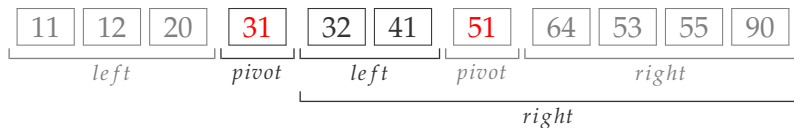
# Quick Sort



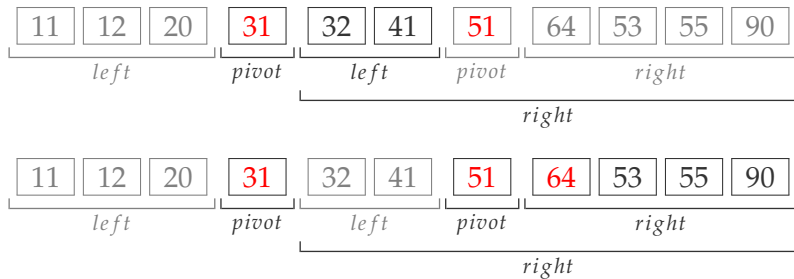
# Quick Sort



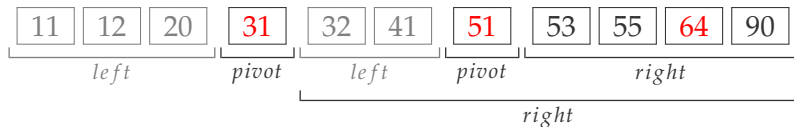
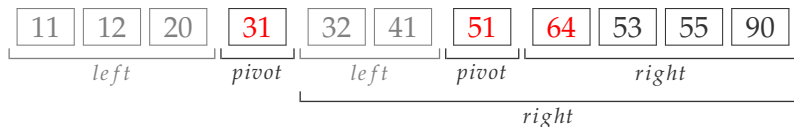
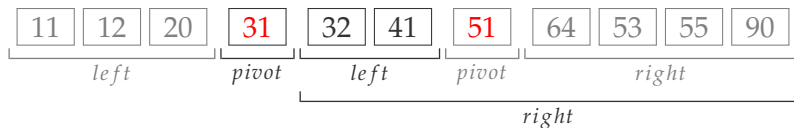
# Quick Sort



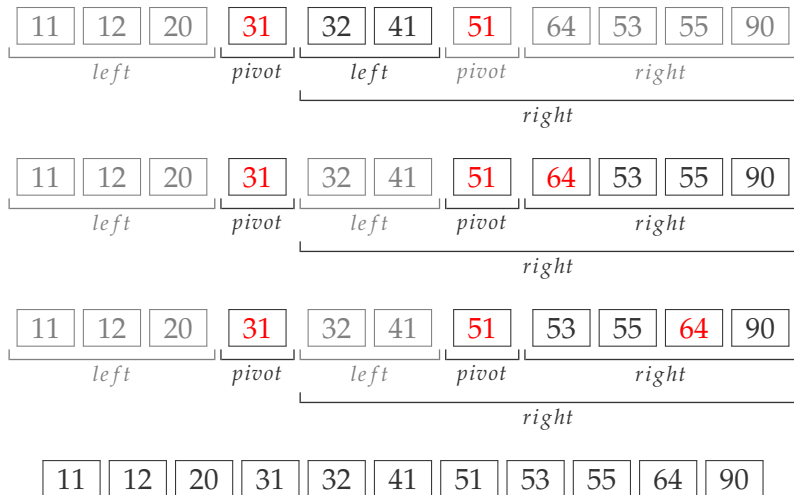
# Quick Sort



# Quick Sort



# Quick Sort



# Quick Sort: the Code (codes/quick.c)

```
34     if (left < right){
35         int pivot = partition(list, n, left, right);
36         quicksort(list, n, left, pivot-1);
37         quicksort(list, n, pivot+1, right);
38     }
39 }
```



# Quick Sort: the Code (codes/quick.c)

```
42     int pivot, temp;
43     int low, high;
44     low = left;
45     high = right + 1;
46     pivot = list[left];
47     do{
48         do
49             low++;
50         while(low <= right && list[low] < pivot);
51         do
52             high--;
53         while (high >= left && list[high] > pivot);
54         if (low < high) {
55             swap(list[low], list[high], temp);
56             printArray(list, n);
57         }
58     } while (low < high);
59     swap(list[left], list[high], temp);
60     printArray(list, n);
```

# Quick Sort: time complexity

Average case:  $O(n \cdot \log_2 n)$ , when elements can be split into “equal size”

Let  $T(n)$  be average time to sort  $n$  records

$$\begin{aligned}T(n) &= c \cdot n + 2 \cdot T(n/2) \\&\leq c \cdot n + 2 \cdot (c \cdot n/2 + 2 \cdot T(n/4)) \\&\leq 2 \cdot c \cdot n + 4 \cdot T(n/4) \\&\dots \\&\leq c \cdot n \cdot \log_2 n + n \cdot T(1) \\&O(n \cdot \log_2 n)\end{aligned}$$

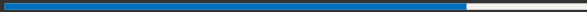
time complexity: worst case  $O(n^2)$ , best case  $O(n \log_2 n)$

# Quick Sort

Other approaches:

- <http://me.dt.in.th/page/Quicksort/>
- <https://ece.uwaterloo.ca/~cmoreno/ece250/quick-sort-complete-example.pdf>

# HEAP SORT



# Heap Sort

Utilize the min heap structure

time complexity

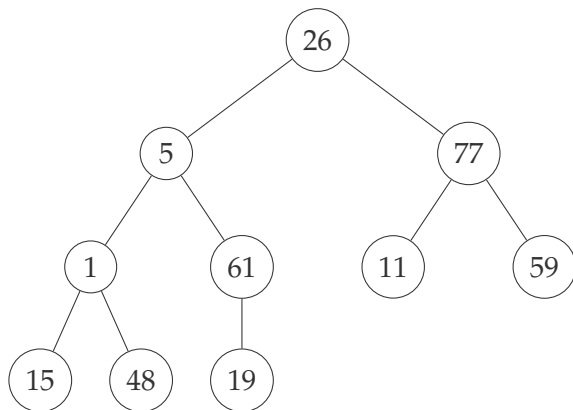
- average case:  $O(n \cdot \log_2 n)$
- average case:  $O(n \cdot \log_2 n)$

First insert the items to the binary tree and delete the min values from the min heap and put it in the sorted array.

# Heap Sort

## Heap sorting process

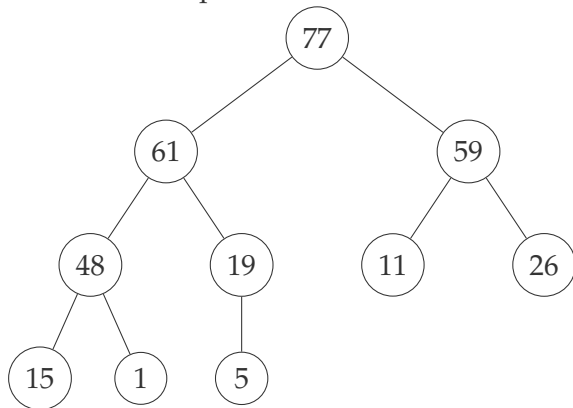
○ input list: (26, 5, 77, 1, 61, 11, 59, 15, 48, 19)



[1]	26
[2]	5
[3]	77
[4]	1
[5]	61
[6]	11
[7]	59
[8]	15
[9]	48
[10]	19

# Heap Sort

Initial max heap construction



# Heap Sort: the code (codes/heapsort.c) I

```
28 // i: an index in list[]; n: is size of heap
29 void heapify(int list[], int n, int i)
30 {
31     int temp;
32     int largest = i; // Initialize largest as root
33     int left = 2*i + 1; // left = 2*i + 1
34     int right = 2*i + 2; // right = 2*i + 2
35
36     // If left child is larger than root
37     if (left < n && list[left] > list[largest])
38         largest = left;
39
40     // If right child is larger than largest so far
41     if (right < n && list[right] > list[largest])
42         largest = right;
43
44     // If largest is not root
45     if (largest != i){
```



# Heap Sort: the code (codes/heapsort.c) II

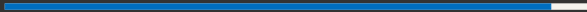
```
46     swap(list[i], list[largest], temp);
47
48     // Recursively heapify the affected sub-tree
49     heapify(list, n, largest);
50 }
51 }
52
53 // heap sort main body
54 void heapsort(int list[], int n)
55 {
56     int temp;
57     for (int i = n / 2 - 1; i >= 0; i--) // Build heap (rearrange list)
58         heapify(list, n, i);
59     for (int i = n - 1; i >= 0; i--) { // extract an element from heap
60         swap(list[0], list[i], temp); // Move current root to end
61         heapify(list, i, 0); // call max heapify on the reduced heap
62     }
63 }
```

# Heap Sort

Time complexity:

$$\log_2 n + \log_2(n-1) + \cdots + \log_2 2 = O(n \cdot \log_2 n)$$

# RADIX SORT



# Radix Sort

- It is a kind of distributive sort
- repeats the following 3 steps
  1. comparison
  2. distribution
  3. merging

# Radix Sort: Example

Initial unsorted array

110	12	380	2	32	41	151	253
-----	----	-----	---	----	----	-----	-----

First consider the One's place

110	12	380	2	32	41	151	253
110	380	41	151	12	2	32	253

Consider the Ten's place

110	380	41	151	12	2	32	253
2	110	12	32	41	151	253	380

Consider the Hundred's place

2	110	12	32	41	151	253	380
2	12	32	41	110	151	253	380

# Radix Sort: Time complexity

Let there be  $d$  digits in input integers.

In general, Radix sort takes  $O(d \cdot (n + b))$  time where  $b$  is the base of the given number. Assume that  $k = \max(n)$ , then  $d = O(\log_b k)$ . The overall

time complexity becomes  $O((n + b) \cdot \log_b k)$  If we further assume that

$k = n^c$ , where  $c$  is constant, the time complexity becomes  $O(n \cdot \log_b n)$