Sorting

강지훈
jhkang@cnu.ac.kr
충남대학교 컴퓨터공학과

Sorting

❑ SORTING

- Estimates suggest that 25% of all computing time is spent on sorting, with some organizations spending more than 50%.

- So, finding an efficient sorting algorithm is very important.

- Unfortunately, no simple sorting technique is the "best" for all initial orderings and sizes of the list being sorted.

Definition of SORT

- Let $(R_0,\ R_1,\ \ldots.\ ,\ \ R_{n-1})$ be a list to be sorted.
  - Each record R_i has a key value K_i .
- Let < be an Ordering Relation.
  - For any key value x and y, either $x > y$, or $x < y$, or $x =$ y.
  - This ordering relation(<) is transitive,
    i.e, if $x < y$ and $y < z$ then $x < z$ for any values x, y, and z.
- Sorting problem:
  - Find a permutation σ such that
    $$K_{\sigma(i-1)} \leq K_{\sigma(i)}\ ,\ 1 \leq i \leq n\text{-}1\ .$$
  - The desired ordering is
    $$(R_{\sigma(0)},\ R_{\sigma(1)}\ ,\ldots.,\ R_{\sigma(n-1)})$$

Example

- $(32, 10, 44, 21, 57)$: a list to be sorted.
  - Eg., the key value K_2 of the record R_2 is *44*.
- Let $\sigma = (1, 3, 0, 2, 4)$ be a permutation being found by a sorting method.
  - $R_{\sigma(0)} = R_1 = 10$
  - $R_{\sigma(1)} = R_3 = 21$
  - $R_{\sigma(2)} = R_0 = 32$
  - $R_{\sigma(3)} = R_2 = 44$
  - $R_{\sigma(4)} = R_4 = 57$
- We can identify that
  $R_{\sigma(0)} \le R_{\sigma(1)} \le R_{\sigma(2)} \le R_{\sigma(3)} \le R_{\sigma(4)}$.
- Therefore, the sorted ordering is
  $(R_{\sigma(0)}, R_{\sigma(1)}, R_{\sigma(2)}, R_{\sigma(3)}, R_{\sigma(4)}) = (10, 21, 32, 44, 57)$

Stable Sorting

- A sorting method generating a permutation σ_s is said to be stable iff

  1. [sorted] $K_{\sigma_S(i-1)} \leq K_{\sigma_S(i)}$, *for* $1 \leq i \leq n-1$
  2. [stable] If $i < j$ and $K_i = K_j$ in the input list, then R_i *precedes* R_j in the sorted list.

- Example: Let *(14, 19, 12, 13, 14)* be a list.
  - Consider two sorting methods, *S* and *T*.
    - Let $\sigma_s = (2, 3, 0, 4, 1)$ be generated by *S*.
    - Let $\sigma_t = (2, 3, 4, 0, 1)$ by *T*.
  - Then, the sorted ordering by both *S* and *T* is identical, i.e., *(12, 13, 14, 14, 19)*.

- *S* is stable, but *T* is not stable.

  Let $i=0$ and $j=4$. Then $K_0 = K_4 = 4$.

  So, $i < j$ and $K_i = K_j$.

  But, $K_0 = K_{\sigma_t(3)}$ and $K_4 = K_{\sigma_t(2)}$.

  So, R_i (= R_0) does not precede R_j (= R_4) in the sorted ordering
  $(K_{\sigma_t(0)}, K_{\sigma_t(1)}, K_{\sigma_t(2)}, K_{\sigma_t(3)}, K_{\sigma_t(4)})$ generated by *T*.

  Therefore, *T* is not stable.

S is stable:

$(4, 9, 2, 3, 4)$

$(2, 3, 4, 4, 9)$

T is not stable:

$(4, 9, 2, 3, 4)$

$(2, 3, 4, 4, 9)$

Internal Sort vs. External Sort

- Internal Sort
  - The list is already in main memory.
  - Any internal sorting is done only using main memory.
- External Sort
  - The list is so huge that all the data in the list cannot be stored in memory at the same time.
  - The list may be stored in auxiliary memory, i.e., in hard disk or tape.
  - Basic Idea:
    - Split the list into several sublists so that each sublist can be stored in main memory.
    - Sort each sublist using an internal sorting method.
    - Merge all the sorted sublist into one sorted list.

Insertion Sort

INSERTION SORT

- Initially, (R_0, R_1) is already sorted.
  - Use a sentinel record R_0 with key value $-\infty$. (This is only for program efficiency.)
- Each time we insert the next record into the correct position.

i	[0]	[1]	[2]	[3]	[4]	[5]
-	$-\infty$	4	2	5	1	3
1	$-\infty$	2	4	5	1	3
2	$-\infty$	2	4	5	1	3
3	$-\infty$	1	2	4	5	3
4	$-\infty$	1	2	3	4	5

(R_0, R_1)
(R_0, R_2, R_1)
(R_0, R_2, R_1, R_3)
$(R_0, R_4, R_2, R_1, R_3)$
$(R_0, R_4, R_2, R_5, R_1, R_3)$

Example: Worst case vs. Best case

* Worst case

i	[0]	[1]	[2]	[3]	[4]	[5]
-	-∞	5	4	3	2	1
1	-∞	4	5	3	2	1
2	-∞	3	4	5	2	1
3	-∞	2	3	4	5	1
4	-∞	1	2	3	4	5

# of comparisons
1 (2)
2 (3)
3 (4)
4 (5)
10 (14)

* Best case

i	[0]	[1]	[2]	[3]	[4]	[5]
-	-∞	1	2	3	4	5
1	-∞	1	2	3	4	5
2	-∞	1	2	3	4	5
3	-∞	1	2	3	4	5
4	-∞	1	2	3	4	5

# of comparisons
1
1
1
1
4

Implementation of Insertion sort

```
private void insertion_sort(int n)
/* perform a insertion sort on the list */
{
    int i, j;
    Element next;
    for (i = 2; i <=n; i++) {
        next = _elements[i];
        for (j = i -1; next.key() <_elements[j].key() ;  j--)
            _elements[j+1] = _elements[j] ;
        _elements[j+1] = next;
    }
}
```

```
/* when the sentinel records not used */
j >= 0 && next.key ()
 < _elements[j].key ()
```

Analysis of Insertion Sort

- The record R_i is Left Out of Order (LOO) iff $max_{0 \leq j<i} \{R_j\} > R_i$.
- Example

[0]	[1]	[2]	[3]	[4]
4	2	5	1	3

  - R_1, R_2 and R_4 are LOO.

- If k records are LOO in the list, then the time complexity is $O(kn+n)=O((k+1)n)$.
  - There are only k times of the insertion step.
  - For each step, there are at most n comparisons.

Analysis (cont'd)

- Worst case: $k=n$
  - $O((k+1)n)=O((n+1)n)=O(n^2)$
- Best case: $k=0$
  - $O((k+1)n)=O(n)$
- Observations
  - If $k << n$, then INSERTION SORT is nice.
  - Good for $n \leq 20$.

Variations

1. Binary Insertion sort: Reducing comparisons
   - Sequential Search: $O(k)$
   - Binary Search: $O(\log k)$
2. List Insertion Sort: Reducing movement time
   - Array: moving whole record
   - Linked list: just moving the pointer to the record.

Quick Sort

Quick Sort (by C.A.R Hoare)

- Very good average behavior
- Basic idea
  - The pivot key (actually K_p, the key of the left most record) is placed at the right spot with respect to the whole file.
  - Thus, if K_p is placed in position $\sigma(p)$,
    then $K_s \le K_p$ *for* $s < \sigma(p)$, and
    $K_s \ge K_p$ *for* $s > \sigma(p)$.
  - Therefore, the original file is partitioned into two subfiles.
- Note
  - In Insertion sort, K_i is placed at the right position with respect to the previously sorted subfile $(R_0, \ldots, R_{i-1})$.

Partitioning

- Which is the pivot record (with the pivot key)?
  - One candidate is the leftmost record.
- How to locate the position $\sigma(l)$ for the pivot value K_l ?
  - From the position $l+1$ up to the position r, find a record R_i such that $K_l > K_i$.
  - From the position r down to the position $l+1$, find a record R_j such that $K_l > K_i$.
  - Then, exchange the two records R_i and R_j.
  - This step is repeated until $i>j$.
  - At the time of exiting the repetition,
    - For every $l<s \leq j, K_s \leq K_l$.
    - For every $j<s \leq r, K_s \geq K_l$.
  - Now, exchange the two records R_l and R_j. Then,
    - For every $l \leq s<j, K_s \leq K_l$.
    - For every $j<s \leq r, K_s \geq K_l$.

Example of Partitioning

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9
26	5	37	1	61	11	59	15	48	19
		i=2							j=9

0	1	2	3	4	5	6	7	8	9
26	5	19	1	61	11	59	15	48	37
				i=4			j=7		

0	1	2	3	4	5	6	7	8	9
26	5	19	1	15	11	59	61	48	37
					j=5	i=6			

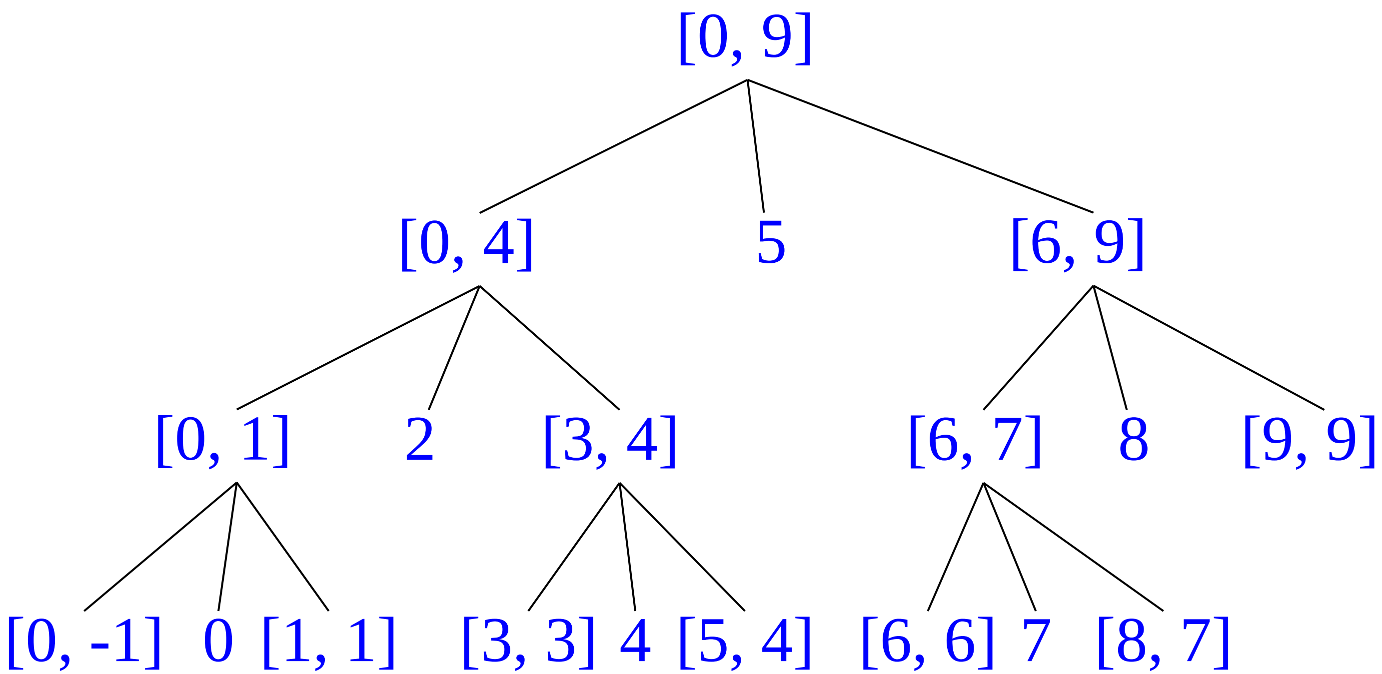
At this time, i>j

0	1	2	3	4	5	6	7	8	9
11	5	19	1	15	26	59	61	48	37

Example of Quick Sort

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	l	r
(26	5	37	1	61	11	59	15	48	19)	$+\infty$	0	9
(11	5	19	1	15)	26	(59	61	48	37)		0	4
(1	5)	11	(19	15)	26	(59	61	48	37)		0	1
()1	(5)	11	(19	15)	26	(59	61	48	37)		0	-1
1	(5)	11	(19	15)	26	(59	61	48	37)		1	1
1	5	11	(19	15)	26	(59	61	48	37)		3	4
1	5	11	(15)	19()	26	(59	61	48	37)		3	3
1	5	11	15	19()	26	(59	61	48	37)		5	4
1	5	11	15	19	26	(59	61	48	37)		6	9
1	5	11	15	19	26	(48	37)	59	(61)		6	7
1	5	11	15	19	26	(37)	48()	59	(61)		6	6
1	5	11	15	19	26	37	48()	59	(61)		8	7
1	5	11	15	19	26	37	48	59	(61)		9	9
1	5	11	15	19	26	37	48	59	61			

Recursive calls at Quick Sort



- Total # of calls: 13
- Max Depth of calls: 4

Recurrence Equation for the time complexity

- Assume the list of size n is partitioned into two sublists with size (j) and $(n\text{-}j\text{-}1)$, respectively.

$$T(n) = \begin{cases} c_0 & \text{if } n = 0 \\ c_1 n + T(j) + T(n-j-1) + c_2 & \text{if } n \geq 1 \end{cases}$$

Worst Case Analysis: $O(n^2)$

- It is when the input file is already in sorted order.

$$T_{wc}(n) = \begin{cases} c_0 & \text{if } n = 0 \\ c_1 n + T_{wc}(0) + T_{wc}(n-1) + c_2 & \text{if } n \geq 1 \end{cases}$$

Example of Worst case: Increasing Order

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	l	r
(10	11	22	33	44	55	66	77	88	99)	+∞	0	9
()10	(11	22	33	44	55	66	77	88	99)		0	-1
10	(11	22	33	44	55	66	77	88	99)		1	9
10	()11	(22	33	44	55	66	77	88	99)		1	0
10	11	(22	33	44	55	66	77	88	99)		2	9
10	11	()22	(33	44	55	66	77	88	99)		2	1
10	11	22	(33	44	55	66	77	88	99)		3	9
10	11	22	()33	(44	55	66	77	88	99)		3	2
10	11	22	33	(44	55	66	77	88	99)		4	9
10	11	22	33	()44	(55	66	77	88	99)		4	3
10	11	22	33	44	(55	66	77	88	99)		5	9
10	11	22	33	44	()55	(66	77	88	99)		5	4
10	11	22	33	44	55	(66	77	88	99)		6	9
10	11	22	33	44	55	()66	(77	88	99)		6	5
10	11	22	33	44	55	66	(77	88	99)		7	9
10	11	22	33	44	55	66	()77	(88	99)		7	6
10	11	22	33	44	55	66	77	(88	99)		8	9
10	11	22	33	44	55	66	77	()88	(99)		8	7
10	11	22	33	44	55	66	77	88	(99)		9	9
10	11	22	33	44	55	66	77	88	99			

- Example of Worst case: Decreasing Order.

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	*l*	*r*
(99	88	77	66	55	44	33	22	11	10)	+∞	*0*	*9*
(10	88	77	66	55	44	33	22	11)	99()		*0*	*8*
()10	(88	77	66	55	44	33	22	11)	99()		*0*	*-1*
10	(88	77	66	55	44	33	22	11)	99()		*1*	*8*
10	(11	77	66	55	44	33	22)	88()	99()		*1*	*7*
10	()11	(77	66	55	44	33	22)	88()	99()		*1*	*0*
10	11	(77	66	55	44	33	22)	88()	99()		*2*	*7*
10	11	(22	66	55	44	33)	77()	88()	99()		*2*	*6*
10	11	()22	(66	55	44	33)	77()	88()	99()		*2*	*1*
10	11	22	(66	55	44	33)	77()	88()	99()		*3*	*6*
10	11	22	(33	55	44)	66()	77()	88()	99()		*3*	*5*
10	11	22	()33	(55	44)	66()	77()	88()	99()		*3*	*2*
10	11	22	33	(55	44)	66()	77()	88()	99()		*4*	*5*
10	11	22	33	(44)	55()	66()	77()	88()	99()		*4*	*4*
10	11	22	33	44	55()	66()	77()	88()	99()		*6*	*5*
10	11	22	33	44	55	66()	77()	88()	99()		*7*	*6*
10	11	22	33	44	55	66	77()	88()	99()		*8*	*7*
10	11	22	33	44	55	66	77	88()	99()		*9*	*8*
10	11	22	33	44	55	66	77	88	99()		*10*	*9*
10	11	22	33	44	55	66	77	88	99			

Example of Worst case for Space complexity

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	l	r
(99	10	11	22	33	44	55	66	77	88)	+∞	0	9
(88	10	11	22	33	44	55	66	77)	99()		0	8
(77	10	11	22	33	44	55	66)	88()	99()		0	7
(66	10	11	22	33	44	55)	77()	88()	99()		0	6
(55	10	11	22	33	44)	66()	77()	88()	99()		0	5
(44	10	11	22	33)	55()	66()	77()	88()	99()		0	4
(33	10	11	22)	44()	55()	66()	77()	88()	99()		0	3
(22	10	11)	33()	44()	55()	66()	77()	88()	99()		0	2
(11	10)	22()	33()	44()	55()	66()	77()	88()	99()		0	0
(10)	11()	22()	33()	44()	55()	66()	77()	88()	99()		3	6
10	11()	22()	33()	44()	55()	66()	77()	88()	99()		2	1
10	11	22()	33()	44()	55()	66()	77()	88()	99()		3	2
10	11	22	33()	44()	55()	66()	77()	88()	99()		4	3
10	11	22	33	44()	55()	66()	77()	88()	99()		5	4
10	11	22	33	44	55()	66()	77()	88()	99()		6	5
10	11	22	33	44	55	66()	77()	88()	99()		7	6
10	11	22	33	44	55	66	77()	88()	99()		8	7
10	11	22	33	44	55	66	77	88()	99()		9	8
10	11	22	33	44	55	66	77	88	99()		10	9
10	11	22	33	44	55	66	77	88	99			

Worst case analysis

- The time complexity mainly depends on the partitioning time.

$$
\begin{aligned}
T_{wc}(n) &= \boxed{c_1 n} + T_{wc}(0) + T_{wc}(n-1) + c_2 \\
&= \boxed{c_1 n} + c_0 + (\boxed{c_1(n-1)} + T_{wc}(0) + T_{wc}(n-2) + c_2) + c_2 \\
&= \boxed{c_1(n+(n-1))} + 2(c_0 + c_2) + T_{wc}(n-2) \\
&\cdots\cdots \\
&= \boxed{c_1 \sum_{i=1}^{n} i} + n(c_0 + c_2) + T_{wc}(0) \\
&= \boxed{c_1 \times \frac{n(n+1)}{2}} + (c_0 + c_2)n + c_0 \\
&= \boxed{\frac{c_1}{2} n^2} + (c_0 + \frac{c_1}{2} + c_2)n + c_0 \\
&= \boxed{O(n^2)}
\end{aligned}
$$

Best case of Quick Sort

- When the list is split roughly into two equal sublists each time.
- The overall partitioning time becomes $O(n \log n)$.
- So, the total sorting time is $O(n \log n)$.

$$
\begin{aligned}
T(n) &\le cn + 2T(\tfrac{n}{2}) \text{ for some constant } c \\
&\le cn + 2(c \cdot \tfrac{n}{2} + 2T(\tfrac{n}{2^2})) = 2cn + 2^2 T(\tfrac{n}{2^2}) \\
&\;\cdots\cdots \\
&\le \alpha \cdot cn + 2^\alpha T(\tfrac{n}{2^\alpha}) \quad \text{(Let } \alpha \text{ be the number such that } \tfrac{n}{2^\alpha} = 1.) \\
&= cn \log_2 n + nT(1) \\
&= O(n \log n)
\end{aligned}
$$

Average Case (By lemma 7.1)

- *O(n log n)*

$$T(n) \quad = \begin{cases} c_0 & \text{if } n = 0 \\ c_1 n + T(j) + T(n-j-1) + c_2 & \text{if } n \geq 1 \end{cases}$$

$$T_{avg}(n) = \frac{1}{n}\sum_{j=0}^{n-1}\left(c_1 n + T_{avg}(j) + T_{avg}(n-j-1) + c_2\right)$$

$$= c_1 n + c_2 + \frac{1}{n}\sum_{j=0}^{n-1}\left(T_{avg}(j) + T_{avg}(n-j-1)\right)$$

$$= c_1 n + c_2 + \frac{2}{n}\sum_{j=0}^{n-1} T_{avg}(j)$$

$$\leq cn + \frac{2}{n}\sum_{j=0}^{n-1} T_{avg}(j), \;\; n \geq 2.$$

- Experimental results show that it is the best of the internal sorting methods as far as average computing time is concerned.

Proof of Lemma 7.1 by Induction

$$T_{avg}(n) \le cn + \frac{2}{n}\sum_{j=0}^{n-1} T_{avg}(j),\ \ n \ge 2.$$

We should prove that $T_{avg}(n) \le kn\log_e n$ for some constant k and for $n \ge 2$.

Assume that $T_{avg}(0) \le b$ and $T_{avg}(0) \le b$ for some constant b.

(1) Induction Base : For $n = 2$,

$$T_{avg}(n) \le c \cdot 2 + \frac{2}{2}\sum_{j=0}^{2-1} T_{avg}(j) = 2c + 2b \le k \cdot 2\log_e 2$$

(2) Induction Hypothesis : Assume $T_{avg}(n) \le kn\log_e n$ for $1 \le n < m$.

(3) Induction Step :

$$T_{avg}(m) \le cm + \frac{2}{m}\sum_{j=0}^{m-1} T_{avg}(j) \le cm + \frac{4b}{m} + \frac{2}{m}\sum_{j=2}^{m-1} T_{avg}(j) \le cm + \frac{4b}{m} + \frac{2k}{m}\sum_{j=2}^{m-1} j\log_e j$$

Since $j\log_e j$ is an increasing function of j,

$$T_{avg}(m) \le cm + \frac{4b}{m} + \frac{2k}{m}\int_2^m x\log_e x dx$$

Note that $\int x\log_e x dx = \left(\frac{1}{2}x^2\log x - \frac{1}{4}x^2\right) + C.$

$$T_{avg}(m) \le cm + \frac{4b}{m} + \frac{2k}{m}\left[\frac{m^2\log_e m}{2} - \frac{m^2}{4}\right] = cm + \frac{4b}{m} + km\log_e m - \frac{km}{2} \le km\log_e m$$

- Space Complexity
  - Worst case : $O(n)$
  - Best case : $O(1)$
  - Average Case : $O(\log n)$
  - Smaller list first :$O(\log n)$

- Quick sort is not stable.

- A variation of pivot value.
  - Pivot value : A median of three.
    - pivot = median { K_l , $K_{(l+r)/2}$, K_r }
  - Example
    - median {10, 5, 7} = 7
    - median { 0, 6, 6} = 6

“How Fast Can We Sort ?”

HOW FAST CAN WE SORT?

- *"What is the best computing time for sorting that we can hope for ?"*

  - We assume that the only operations permitted are comparison and interchange.

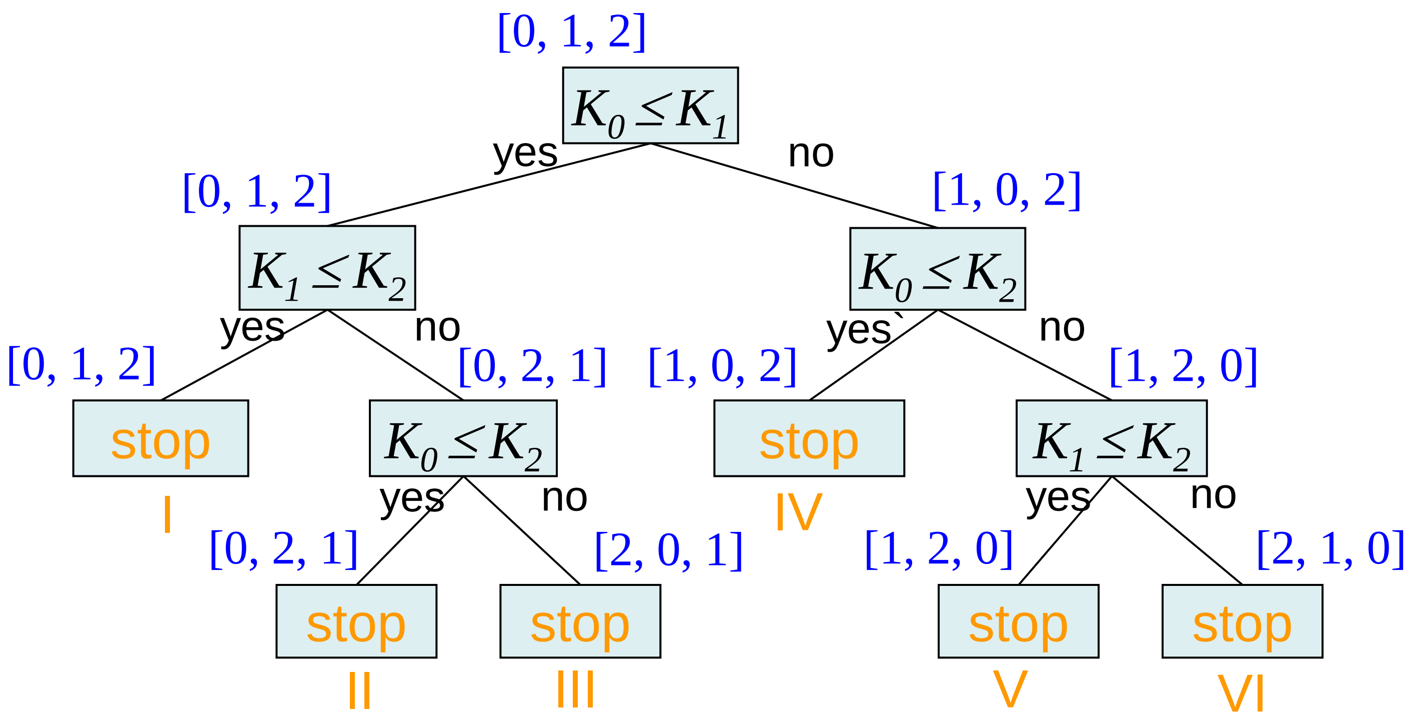
  - The conclusion is: $\Omega(n \log n)$.

Decision Tree

- Vertex and Branch
  - Vertex : key comparison
  - Branch : the result of Key comparison

- Each path in the tree represents a possible sequence of computations that an algorithm could produce.

- Decision Tree for Insertion Sort with (R_0, R_1, R_2) .



- Example: all the permutations of 7, 9, 10.

Leaf	Permutations	Sample key values that give the permutation.
I	0 1 2	(7, 9, 10)
II	0 2 1	(7, 10, 9)
III	2 0 1	(10, 7, 9)
IV	1 0 2	(9, 7, 10)
V	1 2 0	(9, 10, 7)
VI	2 1 0	(10, 9, 7)

- Theorem 7.1
  Any decision tree that sorts n distinct elements has a height of at least
  $$log(n!) + 1$$

(Proof)

There are $n!$ leaves in the decision tree.

The decision tree is a binary tree.

A binary tree can have at most 2^{k-1} leaves.

Height $k \Rightarrow$ at most 2^{k-1} leaves.

2^{k-1} leaves $\Rightarrow$ height of at least k.

n! leaves $\Rightarrow$ height of at least $\log_2(n!) + 1$.

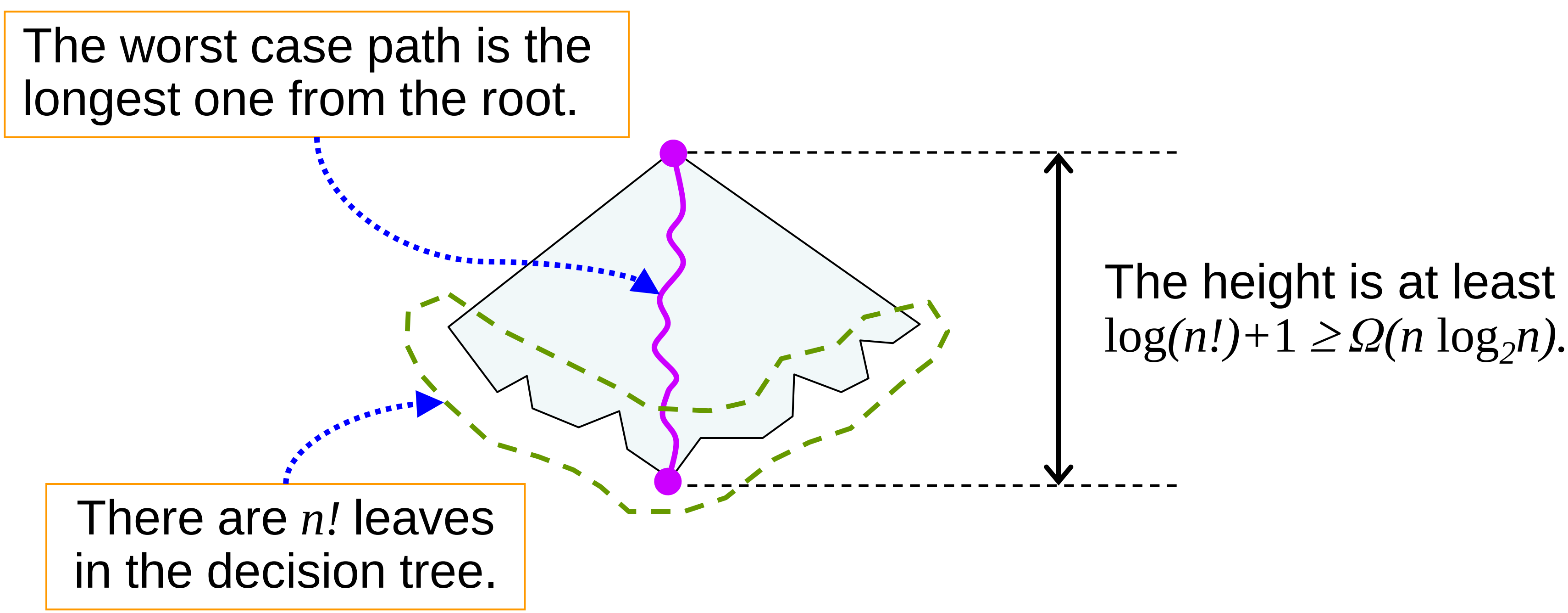
Therefore , the height of the decision tree is at least $\log(n!) + 1$.

- Corollary
  Any algorithm which sorts by comparisons only must have a worst case computing time of $\Omega(n \log_2 n)$.

(Proof)

$n! = n(n-1)(n-2) \cdot\cdot\cdot 3 \cdot 2 \cdot 1 \ \geq \ (n/2)^{n/2}$

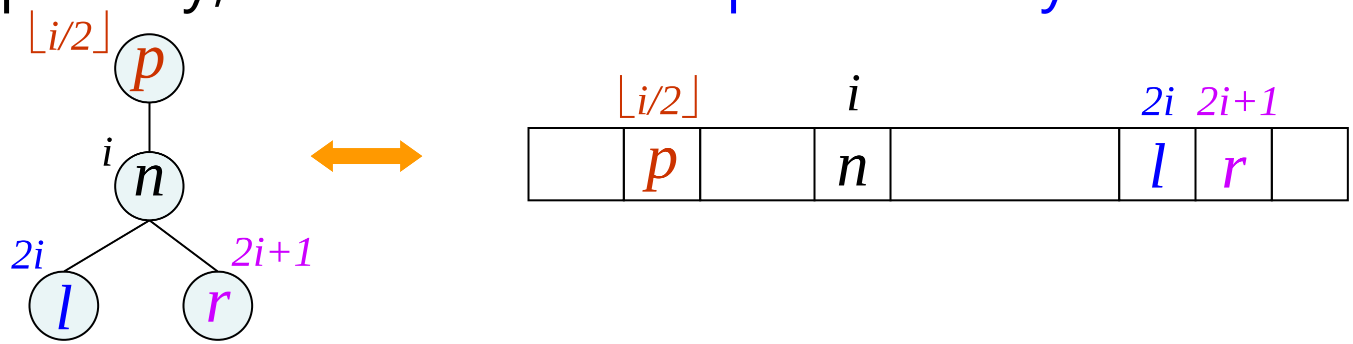
So, $\log_2(n!) \geq (n/2)\log_2(n/2) = \Omega(n \log_2 n)$.

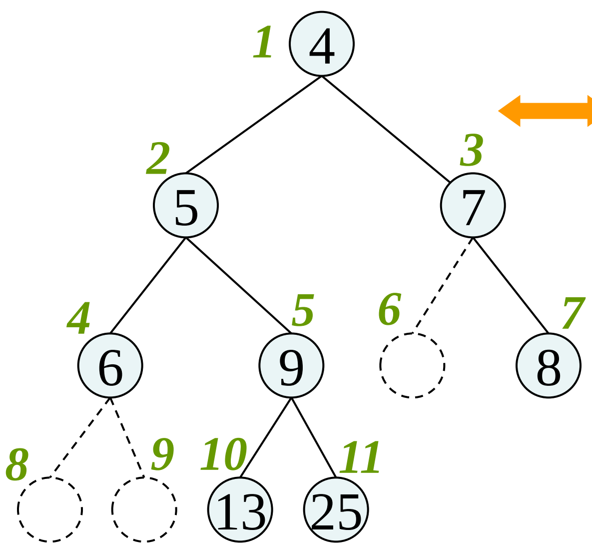


Heap Sort

Heap Sort

- Sequential Representation of Binary trees.
  - Especially, suitable for complete binary trees.



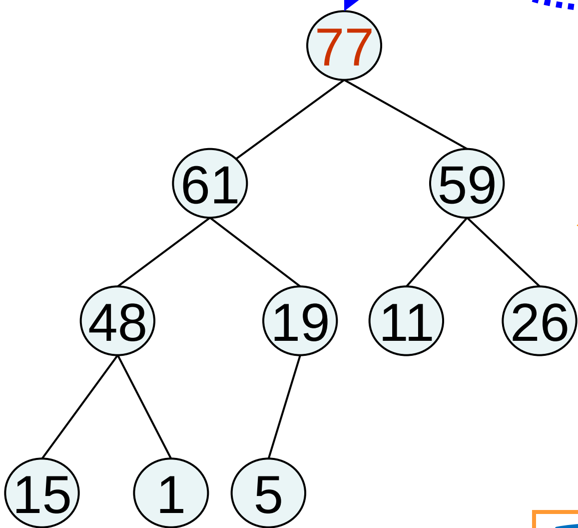


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
-	4	5	7	6	9	-	8	-	-	13	25

Current Node: $i = 5$	$[i] = 9$
Left child : $2i = 10$	$[2i] = 13$
Right child: $(2i+1) = 11$	$[2i+1] = 25$
Parent: $\lfloor i/2 \rfloor = 2$	$[\lfloor i/2 \rfloor] = 5$

- Heap: A complete binary tree such that the value of each node is at least as large as the value of its children nodes.

The root of a heap has the largest value in the tree.



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

The nodes in any path from the root to a leaf are in non-increasing order.

- 77 → 61 → 48 → 15
- 77 → 61 → 48 → 1
- 77 → 61 → 19 → 5
- 77 → 59 → 11
- 77 → 59 → 26

Basic idea of Heap Sort

1. Make a heap from input.
2. Repeat the next step until the heap becomes empty.
   - Output and delete the root , and adjust the heap.

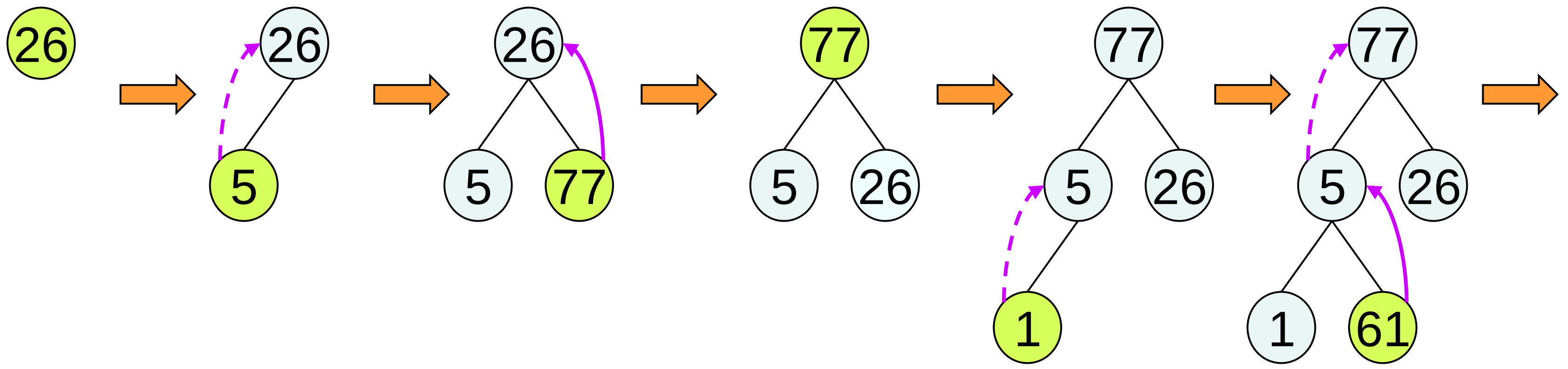
How to make a HEAP

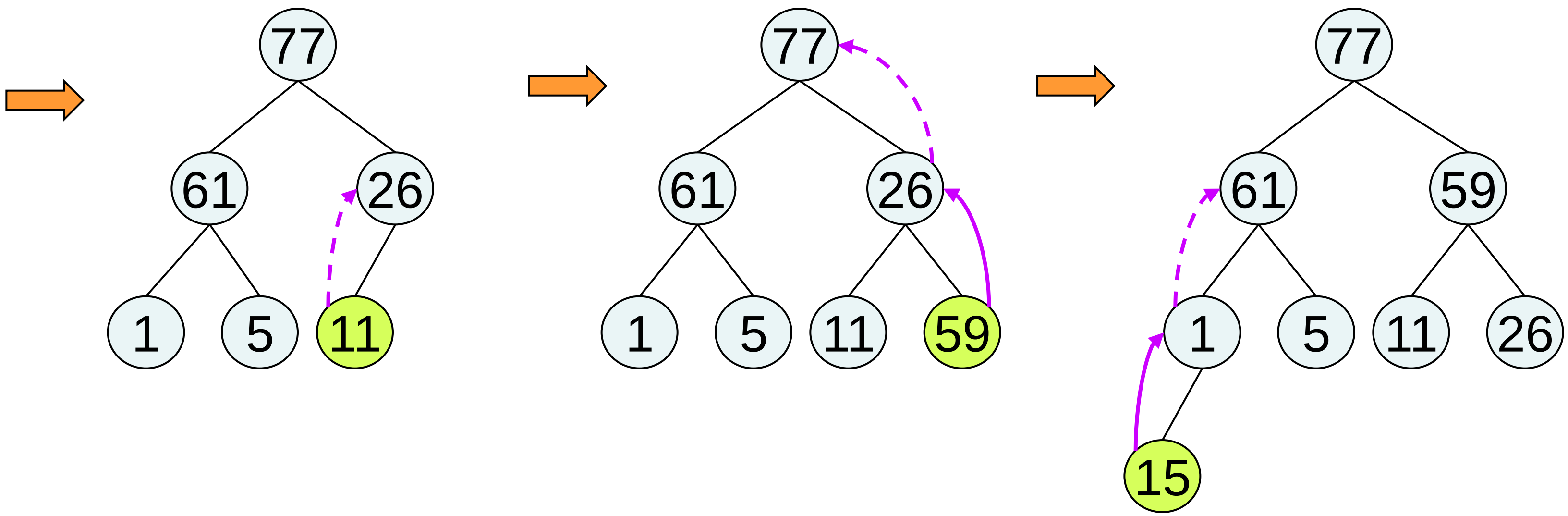
- By “INSERT”
- By “ADJUST”

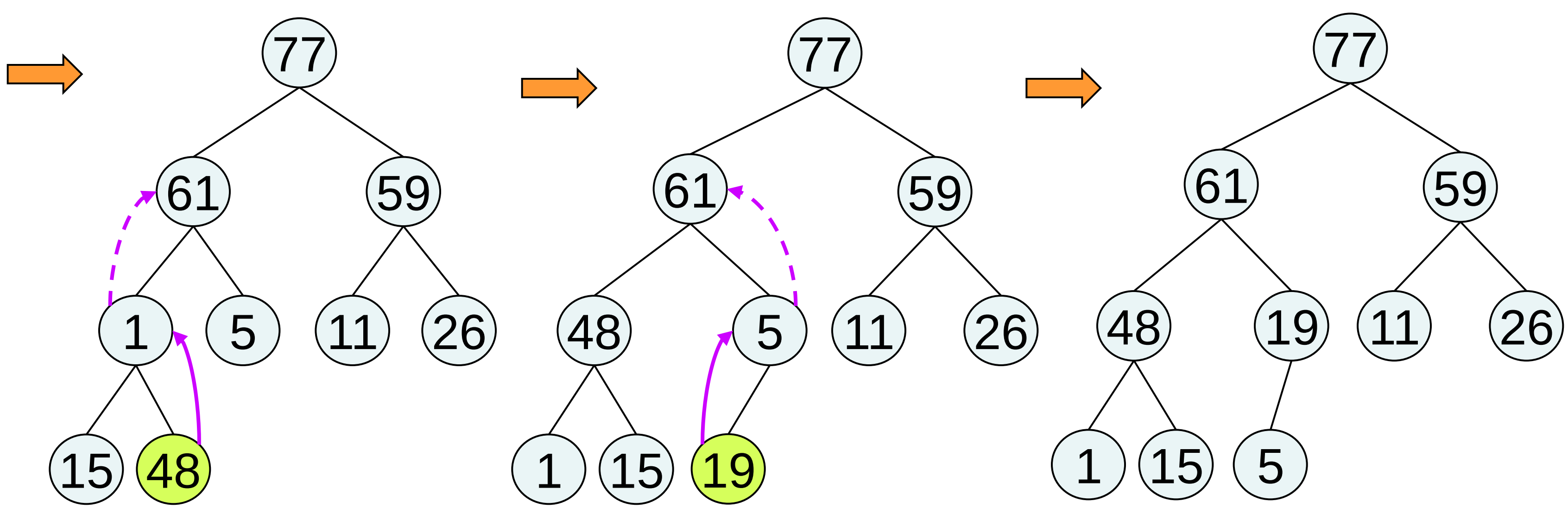
Making Initial Heaps by "INSERT"

- Example

  (26, 5, 77, 1, 61, 11, 59, 15, 48, 19)

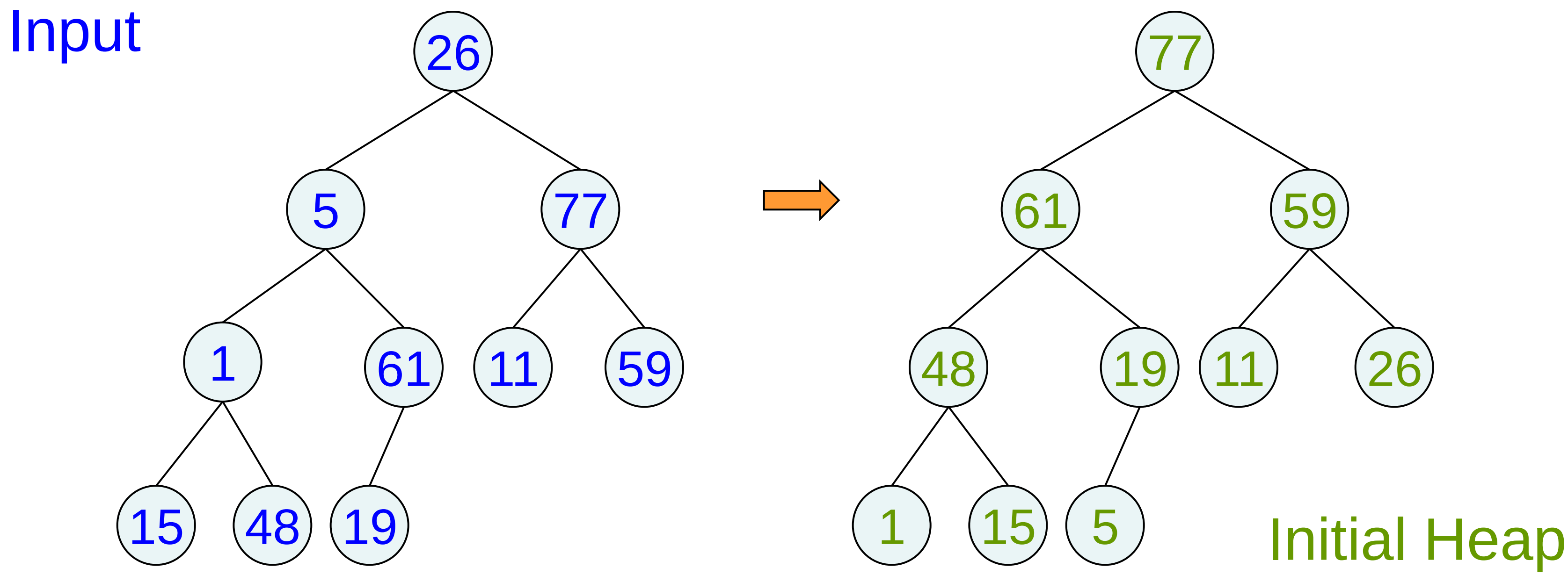



77
61 26
1 5 11
77
61 26
1 5 11 59
77
61 59
1 5 11 26
15


77
61 59
1 5 11 26
15 48
77
61 59
48 5 11 26
1 15 19
77
61 59
48 19 11 26
1 15 5

Representation of input and Initial Heap

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



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

- Worst Case Analysis of INSERT
  - It is when the elements are inserted in ascending order.
  - Each new element will rise to become the new root.
  - Complexity: $O(n \log n)$

- Average case of INSERT: $O(n)$

Proof of Worst Case Complexity of INSERT

- The level i in a complete binary tree has at most 2^{i-1} nodes.
- A complete binary tree with n nodes has maximum level $\log_2(n+1)$.

$$\sum_{i=2}^{\lceil \log_2(n+1) \rceil} (i-1)\, 2^{i-1}, \ \text{(Let } k = \lceil \log_2(n+1) \rceil\text{)}.$$

$$= \sum_{i=2}^{k} (i-1)\, 2^{i-1} = \sum_{i=1}^{k-1} i \cdot 2^{i}$$

$$= (k-1)2^{k+1} - k2^{k} + 2$$

$$= 2k \cdot 2^{k} - 2 \cdot 2^{k} - k \cdot 2^{k} + 2$$

$$= k \cdot 2^{k} - 2(2^{k} - 1) \le k \cdot 2^{k}$$

$$= \lceil \log_2(n+1) \rceil 2^{\lceil \log_2(n+1) \rceil} = O(n \log n)$$

Lemma : $\sum_{\mathrm{i=1}}^{\mathrm{k}} i \cdot x^{i} = \dfrac{k \cdot x^{x+2} - (k+1)x^{k+1} + x}{(x-1)^{2}}, x \neq 1$

(Proof)

Let $S = \sum_{i=1}^{k} i \cdot x^{i}$

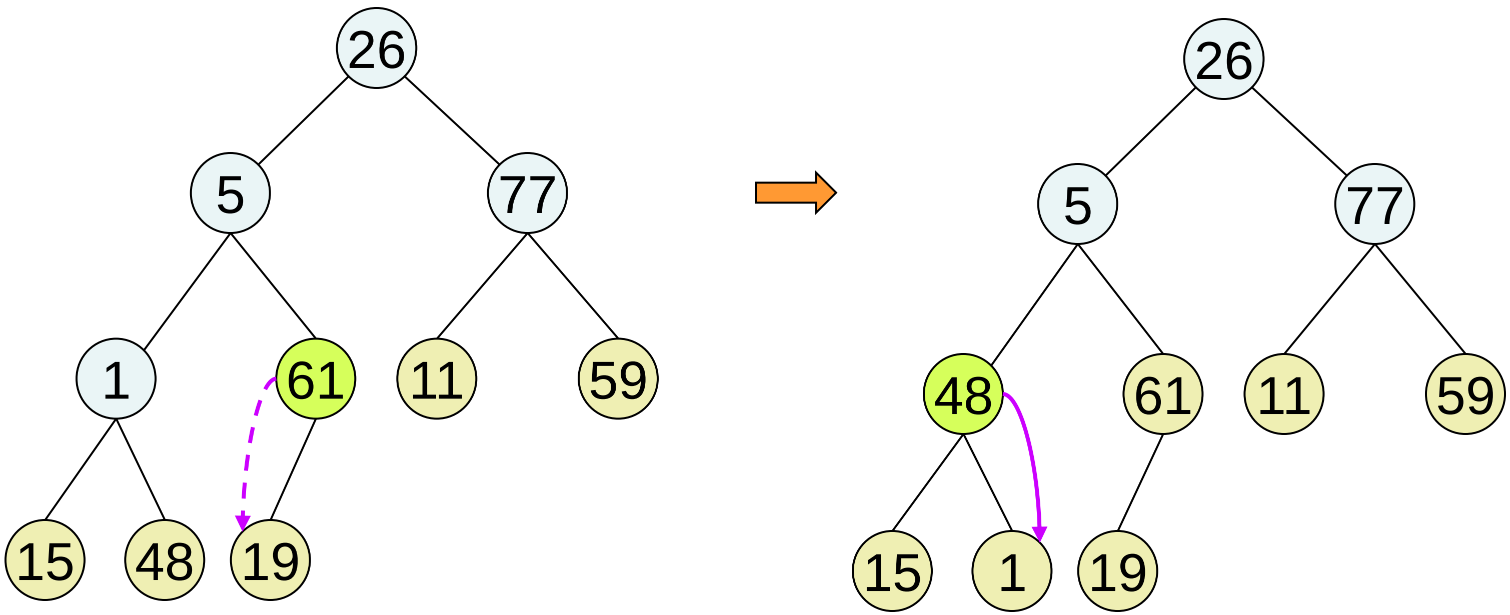
Then, $S - x \cdot S = \sum_{i=1}^{k} i \cdot x^{i} - x\sum_{i=1}^{k} i \cdot x^{i}$

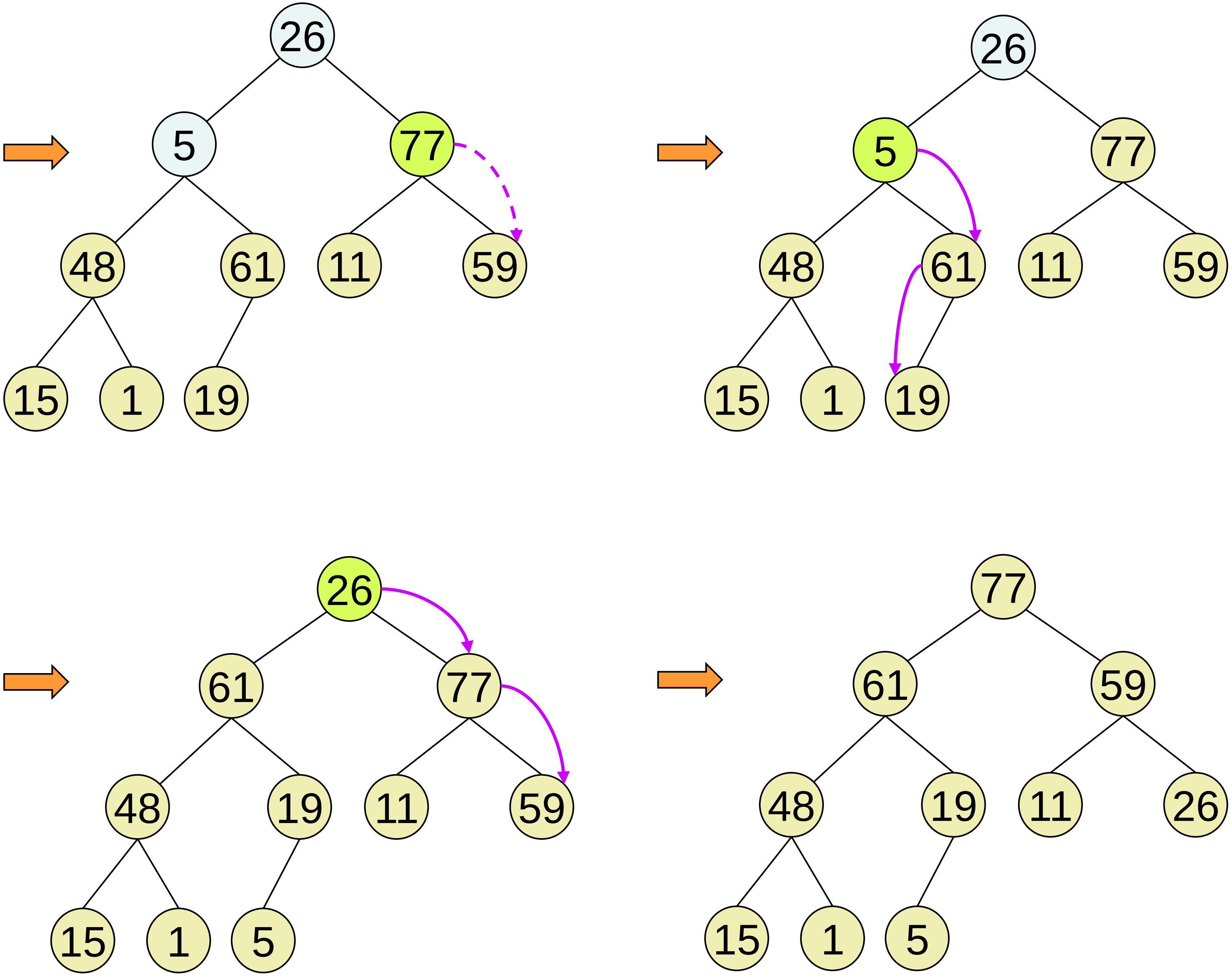
$$
\begin{aligned}
(1-x)S &= \sum_{i=1}^{k} i \cdot x^{i} - \sum_{i=1}^{k} i \cdot x^{i+1} \\
&= \sum_{i=1}^{k} i \cdot x^{i} - \sum_{i=1}^{k} (i+1-1) \cdot x^{i+1} \\
&= \sum_{i=1}^{k} i \cdot x^{i} - \left(\sum_{i=1}^{k} (i+1) \cdot x^{i+1} - \sum_{i=1}^{k} x^{i+1}\right) \\
&= \left(\sum_{i=1}^{k} i \cdot x^{i} - \sum_{i=2}^{k+1} i \cdot x^{i}\right) + \sum_{i=1}^{k} x^{i+1} \\
&= \left(x - (k+1)x^{k+1}\right) + \frac{x^{2}(1-x^{k})}{(1-x)} \\
&= \frac{x(1-x) - (k+1)x^{k+1}(1-x) + x^{2}(1-x^{k})}{(1-x)} \\
&= \frac{x - x^{2} - (k+1)x^{k+1} + (k+1)x^{k+2} + x^{2} - x^{k+2}}{(1-x)} \\
&= \frac{kx^{k+2} - (k+1)x^{k+1} + x}{(1-x)}
\end{aligned}
$$

$$\therefore S = \sum_{i=1}^{k} i \cdot x^{i} = \frac{kx^{k+2} - (k+1)x^{k+1} + x}{(x-1)^{2}}$$

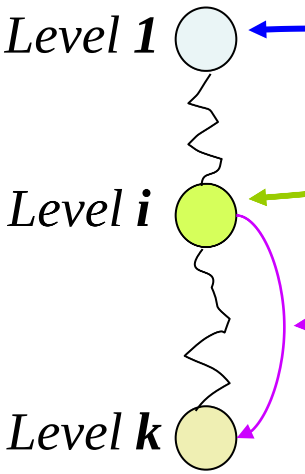
Making Initial Heaps by "ADJUST"

- Initially, each leaf is a heap.




26
5
77
48
61
11
59
15
1
19
26
5
77
48
61
11
59
15
1
19
26
61
77
48
19
11
59
15
1
5
77
61
59
48
19
11
26
15
1
5

Worst Case Analysis of ADJUST



- The root node
- The node to be adjusted.
  (There are maximum 2^{i-1} nodes in the level i.)
- The node at the level i can be moved down $(k - i)$ steps.
- $2^{k-1} \le n < 2^k$, where n is the number of all the nodes.

$$\sum_{i=1}^{k}(k-i)2^{i-1} = \sum_{j=0}^{k-1} j \cdot 2^{k-j-1} = 2^{k-1}\sum_{j=1}^{k-1} j(1/2)^j$$

$$(1)\ \ 2^{k-1} \le n$$

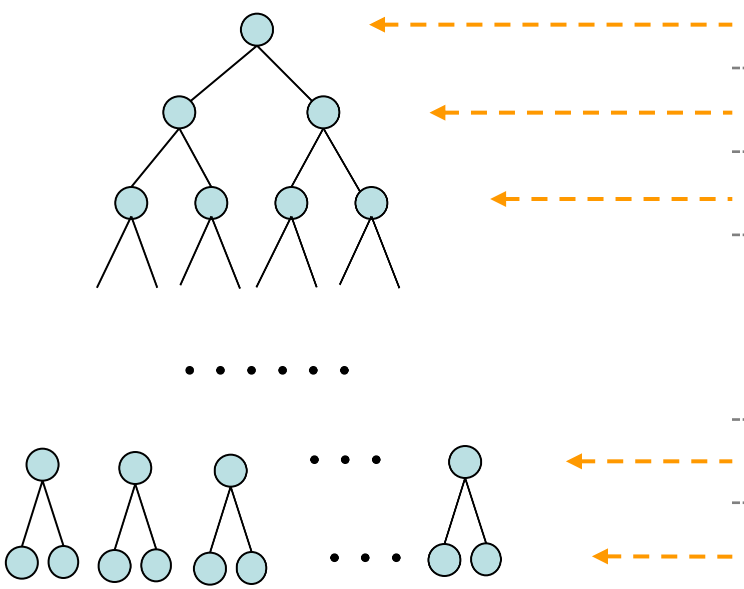
$$(2)\ \ \sum_{j=1}^{k-1} j(1/2)^j = \frac{(k-1)(1/2)^{k+1} - k(1/2)^k + 1/2}{(1/2)^2}$$

$$= (k-1)(1/2)^{k-1} - k(1/2)^{k-2} + 2$$

$$= 2 - (k+1)\ (1/2)^{k-1} \le 2$$

$$\therefore \sum_{i=1}^{k}(k-i)2^{i-1} \le n \cdot 2 = O(n)$$

- Relationship between the depth and the number of nodes in the complete binary trees.
  - Let n be the number of nodes in a complete binary tree.
  - Let k be the depth of it (i.e., the maximum level).



Level	Max Nodes	Total Nodes
1	2^0	2^1-1
2	2^1	2^2-1
3	2^2	2^3-1
⋮		
$k-1$	2^{k-2}	$2^{k-1}-1$
k	2^{k-1}	2^k-1

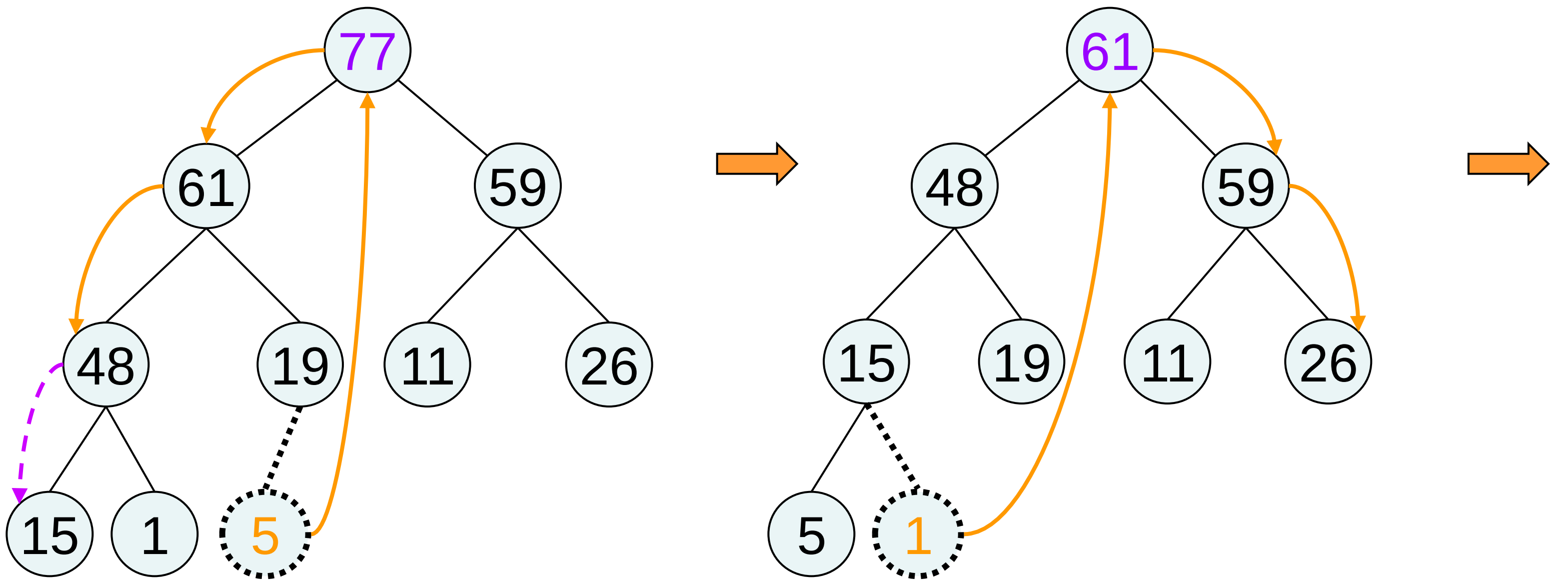
Relationship between the depth and the number of nodes in the complete binary trees. (Cont'd)

- From the complete binary tree,
  - $2^{k-1} \leq n < 2^k$
  - $2^{k-1} \leq n \leq 2^k - 1$
- By taking log_2 at each side, we get:
  - $k - 1 \leq \log_2 n$, i.e. $k \leq \log_2 n + 1$
  - $n+1 \leq 2^k$, i.e. $\log_2(n+1) \leq k$
- By combining the two inequalities,
  - $\log_2(n+1) \leq k \leq \log_2 n + 1 < \log_2(n+1)+1$,
    i.e., $\log_2(n + 1) \leq k < \log_2(n+1)+1$
- Therefore, we finally get the equation:
  - $k = \lceil \log_2(n + 1) \rceil$

Heap sort
after making an initial heap

Heap sort after making an initial heap [1]

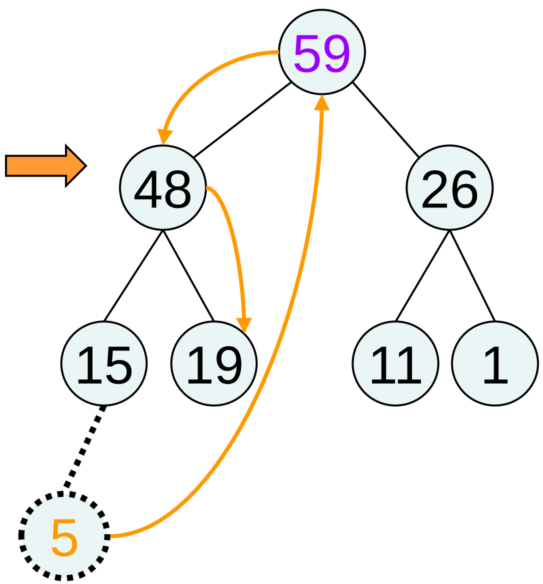
1	2	3	4	5	6	7	8	9	10
77	71	59	48	19	11	26	15	1	5



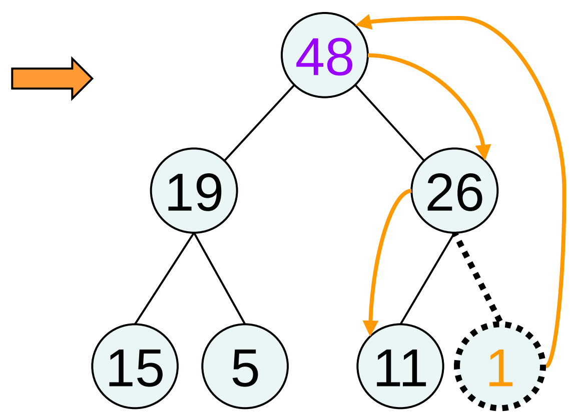
1	2	3	4	5	6	7	8	9	10
61	48	59	15	19	11	26	5	1	77

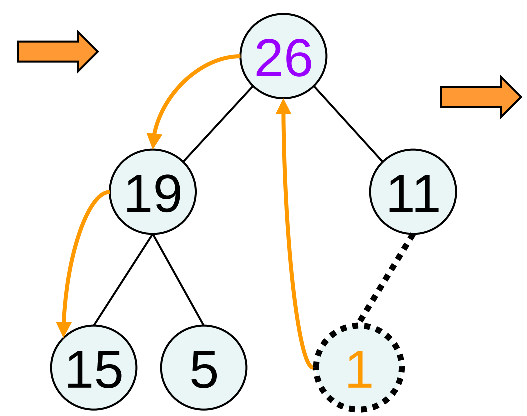
Heap sort after making an initial heap [2]

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



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

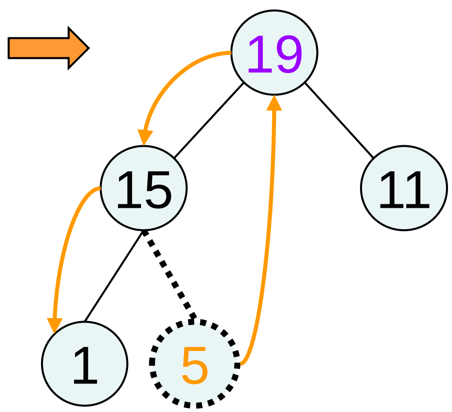




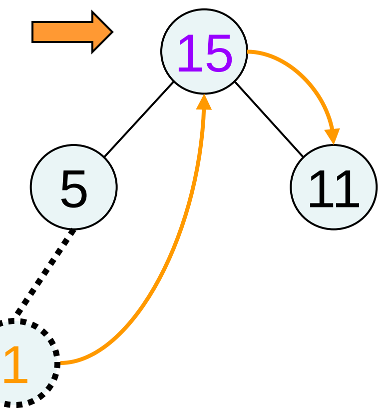
1	2	3	4	5	6	7	8	9	10
26	19	11	15	5	1	48	59	61	77

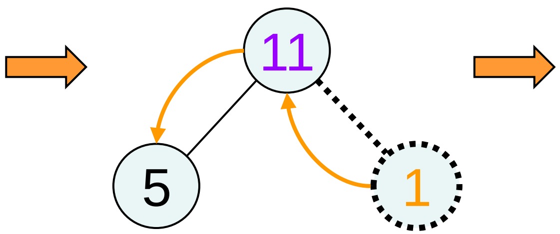
Heap sort after making an initial heap [3]

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



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

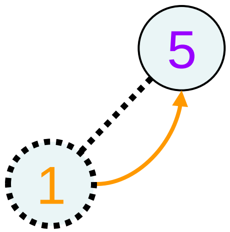




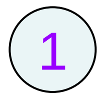
1	2	3	4	5	6	7	8	9	10
11	5	1	15	19	26	48	59	61	77

Heap sort after making an initial heap [4]

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



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



At this time, the tree has only one node. So, we stop the sorting. And the array has now been sorted.

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

Complexity of Heap Sort

- Overall Sorting time:
  - Total ADJUST time + Making an initial heap
- Total ADJUST time: $O(n \log n)$
  - Each ADJUST with n nodes. : $O(\log n)$
  - We need n times of ADJUST after deleting each root.
- Making an initial heap
  - $O(n \log n)$ by INSERT
  - $O(n)$ by ADJUST
- Therefore, totally $O(n \log n)$.

INSERT vs ADJUST

- ADJUST : $O(n)$
  - But it requires that all the elements should be available before the heap creation begins.
- INSERT : $O(n \log n)$
  - We can add a new element into the heap at any time.

Partial Sorting using Heap Sort

- We need only first k records in the sorted nodes from n records, especially when $k << n$.

  $\Rightarrow O(k \log n)$

Sorting with Several Keys

Radix Sort

Several Keys

- Example: sorting a deck of cards.
  - [suit] ♣< ◊ < ♥ < ♠
  - [Face Value] 2 < 3 < . . . < 9 < 10 < J < Q < K < A
  - Two possible Sorted Decks:
    - 2♣ < 3♣ < . . . < A♣ < . . . . < 2♠ < 3♠ . . . < A♠
    - 2♣ < 2◊ < 2♥ < 2♠ < 3♣ < 3◊ < . . . < A♣ < A◊ < A♥ < A♠
- Basic idea:
  - Sort by each key separately.

Several Keys

- Notation:
  - Each Record has the keys $K^0, \dots, K^{r-1}$
  - So, the record R_i has the keys $K_i^0, \dots, K_i^{r-1}$
- Comparison of multiple keys:
  - $(x_0, x_1, \dots, x_{r-1}) \le (y_0, y_1, \dots, y_{r-1})$

    iff either $x_i = y_i, 0 \le i < j$ and $x_{j+1} < y_{j+1}$ for some $j < r - 1$

    or $x_i = y_i, 0 \le i < r$.
- Definition of Sorted Order with several keys:
  - A list of records, $R_0, \dots, R_{n-1}$, is lexically sorted with respect to the keys $K^0, K^1, \dots, K^{r-1}$

    iff $(K_i^0, K_i^1, \dots, K_i^{r-1}) \le (K_{i+!}^0, K_{i+1}^1, \dots, K_{i+1}^{r-1}), 0 \le i < n-1$

Radix Sort

Two approaches for Radix Sort

- MSD (Most Significant Digit) Sort

- LSD (Least Significant Digit) Sort

MSD (Most Significant Digit) Sort

- K^0 first
- Example for sorting a card deck: K^0[suit] and K^1[face value]
  1. Sort on suit, and have 4 piles of cards.
  2. Sort each suit file on face value separately.
  3. Stack the 4 files so that the space(♠) file is on the bottom and the club(♣) file is on the top.

LSD (Least Significant Digit) Sort

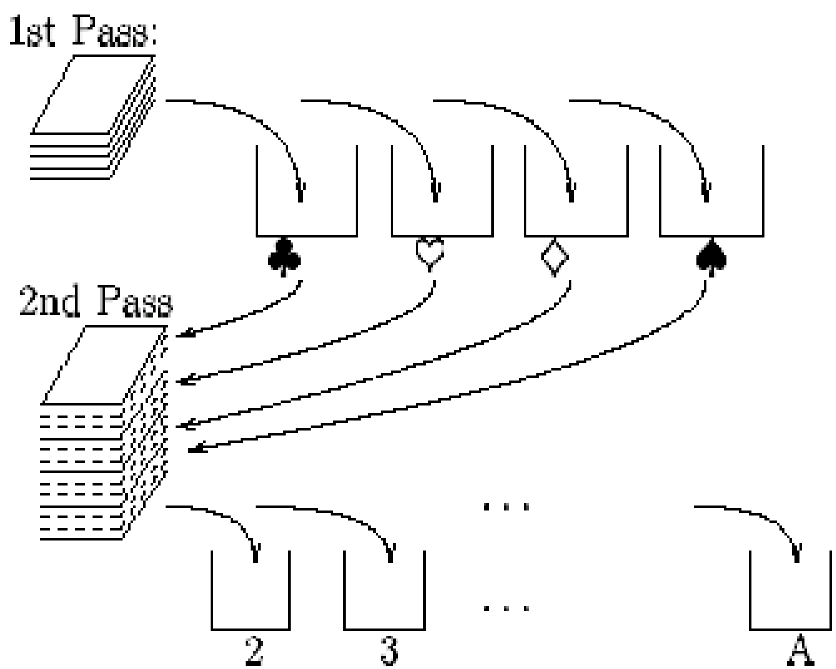
- K^{r-1} first
- Example for sorting a card deck: K^0[suit] and K^1[face value]
    1. Sort on face value, and have 13 piles of cards.
    2. Stack the piles to obtain a single pile.
    3. Sort on suit, and have 4 piles.
    4. Stack the piles to obtain a sorted deck.
- The sorting method in the second and the later passes must be stable.

LSD (Least Significant Digit) Sort

- The LSD approach is simpler than the MSD approach.
  - We do not have to sort subpiles independently.
  - Less overhead than the MSD approach.
- The term LSD or MSD indicates only the order in which the keys are sorted
  - They do not specify how each key is to be sorted.
  - We usually create bins.
    - Bin Sort

Bin Sort

- An implementation for MSD or LSD.
- Stable.
- Time Complexity
  - $O(rn) = O(n)$, where n is the number of records and r is the number of different key values.



Applying Radix Sort to sorting with only one logical key [1]

We interpret the key as a composite of several keys:

- $K = (K^0, K^1, \ldots, K^{r-1})$
- Example: $0 \le K \le 999$
  - Let each K^i be as follows:
    - K^0 is the digit in the 100s place
    - K^1 is the digit in the 10s place
    - K^2 is the digit in the units place
  - Then, each K^i is $0 \le K^i \le 999$.
  - The Sort for each key requires only 10 bins.
- We decompose the sort key into digits using a radix r.
  - When r=10, we get the decimal decomposition.
  - When r=2, we get the binary decomposition.
- With a radix of r, r bins are needed to sort each digit.

Applying Radix Sort to sorting with only one logical key [2]

LSD radix r Sort

- Assume that the records, R_0, …, R_{n-1}, have the keys that are d-tuples $(x_0, x_1, \ldots, x_{d-1})$, and $0 \le x_i < r$.
- Analysis:
  - O(d(r+n))
  - O(n) if r ≪ n and d ≪ n .

Summary

Practical Considerations

- Sorting method vs Data size
  - $n \leq 20\sim25$: Use Insertion sort ($O(n^2)$)
  - $n > 20\sim25$: Use Quick sort ($O(n \log n)$)
  - For quick sort:
    - If the subfile size ≤ 20 after partitioning, then sort this subfile using Insertion Sort instead of Quick Sort.
    - This situation is similar in other sorts.
- Long Records
  - Exchanging records requires much time.
  - Use linked lists or index
- Running Time Comparisons among Sorts

	n=256	n=512
Insertion	336	1444
Bubble	1026	4054
Heap	110	241
Quick	60	146
Merge	102	242

End of Sorting