COMP20003 Workshop Week 8

Divide & Conquer

Divide & Conquer:

- Example with Top-Down Merge Sort
- Recurrences
- Master Theorem: when & when-not applied

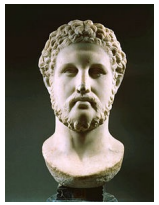
Other Merge Sort Variants

- Bottom-Up Merge Sort using Arrays
- Bottom-Up Merge Sort using Linked Lists
- Time & Space Complexity

Lab:

- MST
- implementing bottom-up mergesort (W8.3, W8.4)

Divide & Conquer

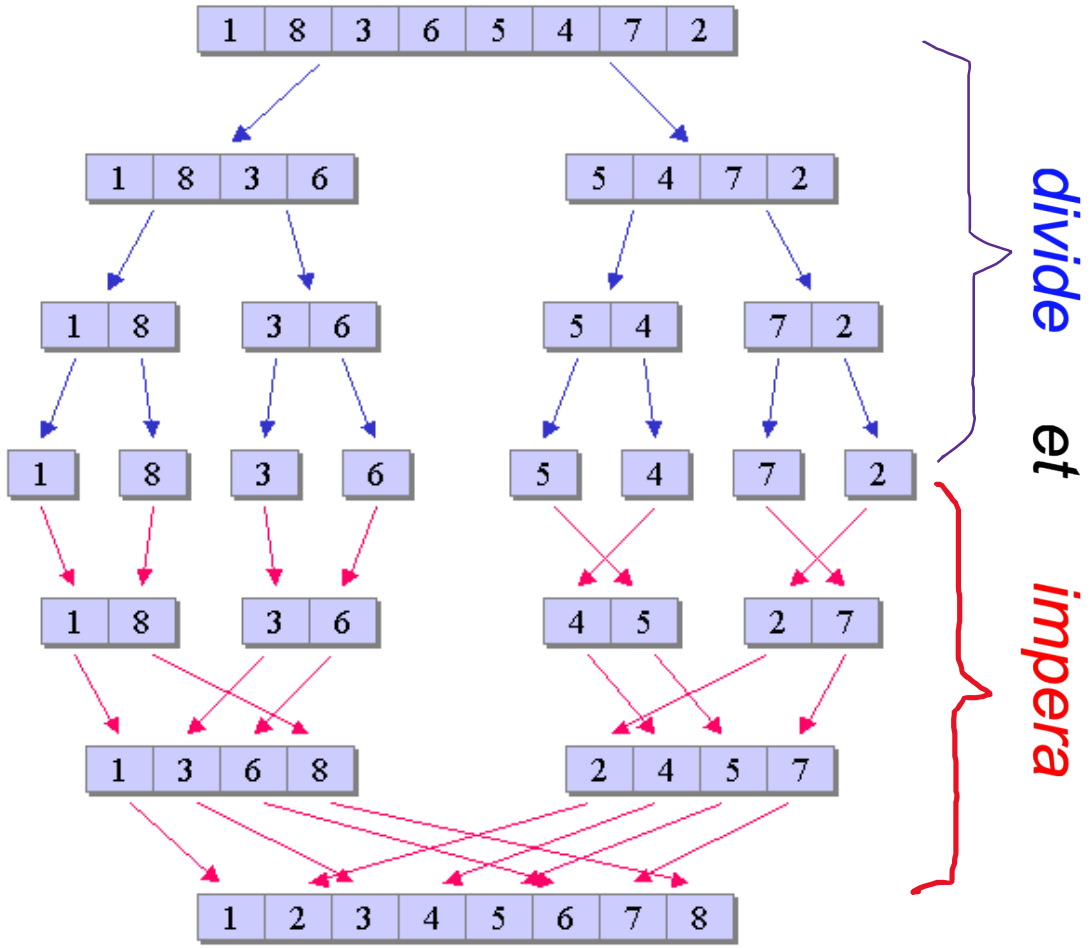


Philip II of Macedon

Julius Caesar

Image Sources: Wikipedia, cs.amherst.ed
https://commons.wikimedia.org/wiki
mpudding_%28cropped%29.j
pg

Divide-and-Conquer example: Top-Down MergeSort (an array or a linked list DS)



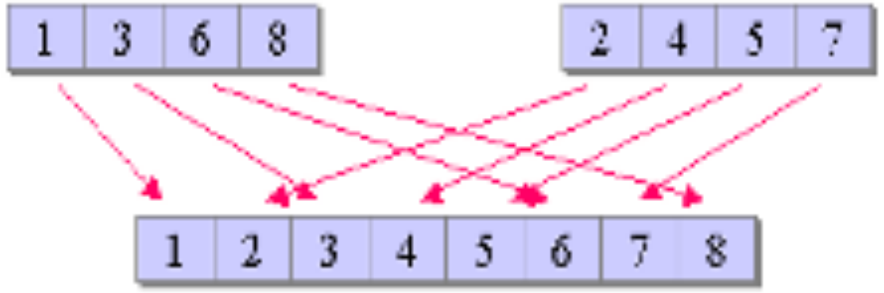
the sorting algorithm is :

- "*divide*":
  - break the DS into 2 halves
  - ditto with sub-DS until getting singleton DS-s

and

- "*conquer*":
  - merge 2 sorted DS into one
  - ditto until getting a single (sorted) DS

How to merge 2 sorted arrays? 2 sorted linked lists?



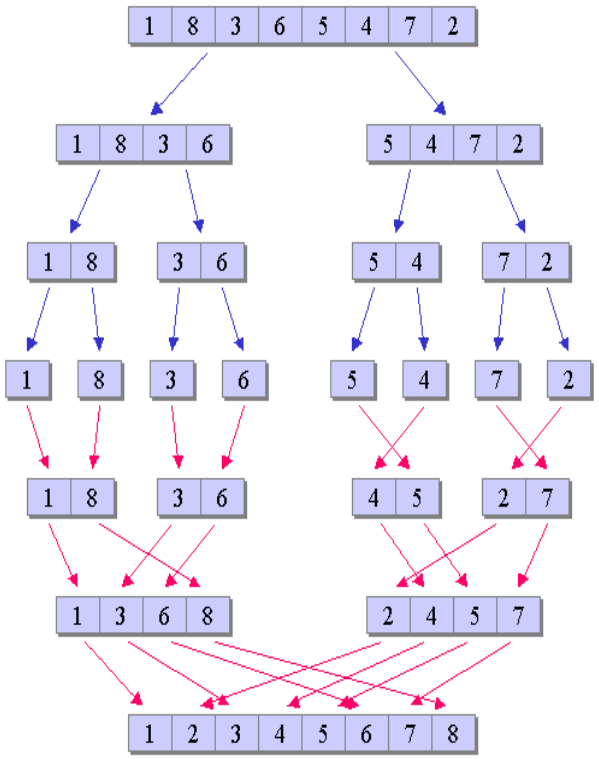
Input: two sorted sequences B and C(could be of different sizes)
Output: A= sorted union sequence B and C

Algorithm:

- Iterate B and C in parallel, and when doing so append the smaller value to A
- When B or C finishes, append the remainders of the other to A

	Merging 2 sorted arrays	Merging 2 sorted lists
Time Complexity	$\theta(?)$	$\theta(?)$
Additional-Space Complexity	$\theta(?)$	$\theta(?)$

Top-Down MergeSort: implementation note

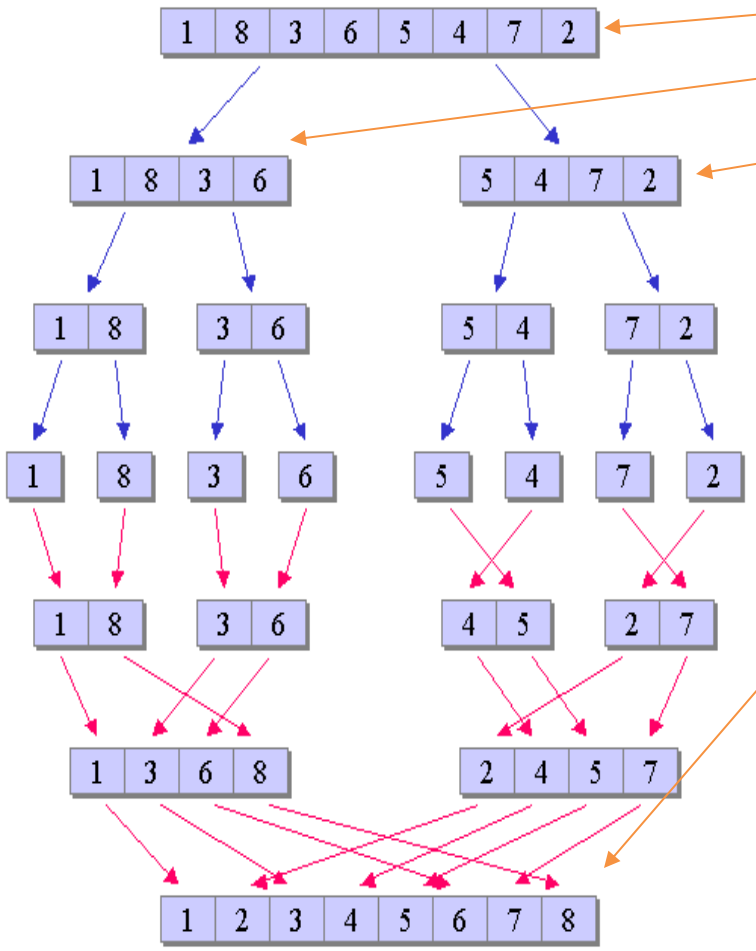


General Top-Down Merge Sort	Array-Based Top-Down Merge Sort

`mergesort(A) {`
`  if (A has > 1 element) {`
`  B= left half of A`
`  C= right half of A`
`  mergesort(B);`
`  mergesort(C);`
`  merge B and C to A;`
`}`
	`a_mergesort(A[], n) {`
	`  if (n>1) {`
	`  mid= n/2;`
	`  B[]= A[0..mid]`
	`  C[]= A[mid+1..n-1]`
	`  a_mergesort(B,mid+1);`
	`  a_mergesort(A,n - (mid+1) );`
	`  a_merge B and C to A;`
	`}`

	Additional-Space Efficiency	
	Array-Based TD Mergesort	Linked-List-Based
for recursive stack	$\theta(\log n)$	$\theta(?)$
for merging	$\theta(n)$	$\theta(?)$
all	$\theta(n)$	$\theta(?)$
Notes	*the algorithm is NOT in-place*	

Top-Down MergeSort: time complexity



Time complexity:

T(n)= T(n/2) # sort the left array of n/2 elements

+ T(n/2) # sort the right array

+ n # merge the 2 sorted arrays

→ T(n) = 2T(n/2) + n (a recurrence)

= ?

Recurrences: describing complexity of recursive algorithms

Time Complexity of a recursive algorithm can be expressed as a recurrence: $T(n)$ depends on $T(k)$ where $k<n$. For example:

merge sort	$T(n)= 2\ T(n/2) + n$
binary search on sorted arrays	$T(n)=$
the best case of quick sort	$T(n)=$
the worst case of quick sort	$T(n)=$

The Master Theorem: quick solving some special recurrences

If a task of size n is divided into

- a tasks of size n/b :

$$T(n) = aT(n/b) + \Theta(n^d)$$

where $a \geq 1$, $b > 1$, and $d \geq 0$,

$\Theta(n^d)$: cost of dividing and conquering

- and if

$$T(1) = \Theta(1)$$

Then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } d > \log_b a \\ \Theta(n^d \log n) & \text{if } d = \log_b a \\ \Theta(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Recurrences: solving using Master Theorem

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } d > \log_b a \\ \Theta(n^d \log n) & \text{if } d = \log_b a \\ \Theta(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

merge sort	*T(n)= 2 T(n/2) + n* *T(1)= 1*	*a= b= d=* *→ T(n) =*
binary search on sorted arrays	*T(n)= T(n/2) + 1* *T(1)= 1*	*a= b= d=* *→ T(n) =*
the best case of quick sort	*T(n)= T(n/2) + n* *T(1)= 1*	*a= b= d=* *→ T(n) =*
the worst case of quick sort	*T(n)= T(n-1) + n* *T(1)= 1*	*a= b= d=* *→ T(n) =*

Recurrences: When the Master Theorem *not-applicable*

- When we cannot find at least one of the *constants* *a*, *b*, *d*. Example (Quick Sort Worst Case)
  - *T(n) = T(n-1) + n*
  - *T(1)= 1*

- Solution for this case: We solve/“unroll” the recursion using substitution.

Recurrences: Example of using substitution

When we cannot find at least one of the *constants* *a*, *b*, *d*. Example (Quick Sort Worst Case)

$T(n) = T(n-1) + n \quad (1)$

$T(1) = 1$

$T(n) = T(n-1) + n$

$= (T((n-1) - 1) + (n-1)) + n$ [substitute *n* by *n-1* in *(1)*]

$= T(n-2) + (n-1) + n$ *#NOTE: be lazy, don't compute the sum, wait to see patterns!*

$= T(n-3) + (n-2) + (n-1) + n$ [substituting *n* by *n-2* in *(1)*]

$= \ldots$

$= T(n-k) + (n-k+1) + \ldots + (n-1) + n \quad (0 \le k < n) \quad (2)$

since we know $T(1)$ we choose k so that $T(n-k)$ is $T(1)$ → $k = (n-1)$

in *(2)* substitute *k with n-1,* we arrive to

$= T(1) + 2 + \ldots + (n-1) + n$

$= 1 + 2 + \ldots + (n-1) + n = n(n+1)/2 = \Theta(n^2)$

Peer Activities: Do W8.10 now

What is the strongest bound on space complexity of top-down linked-list-based mergesort?

a. $O(1)$
b. $\Theta(\log n)$
c. $\Theta(n)$
d. none of the above

An algorithm subdivides its input into the ranges: low, medium, and high. It discards all data that is not included within the highest range, and repeats the subdivision on the remaining data. Each subdivision requires the algorithm to iterate through each data item.

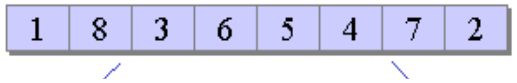
Which recurrence relation describes the algorithm in this scenario?

a. $T(n) = T(n/3) + 1$
b. $T(n) = T(n/3) + n$
c. $T(n) = 2T(n/3) + 1$
d. $T(n) = 2T(n/3) + n$

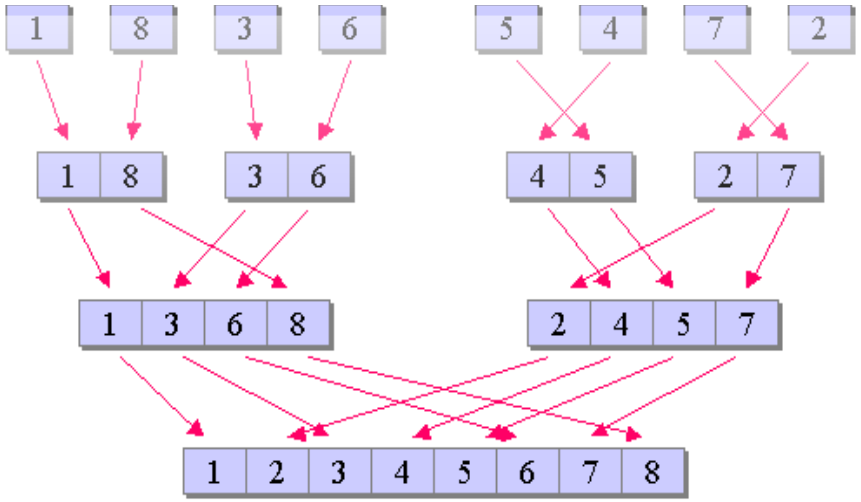
Other Variants: Bottom-Up Merge Sort

Bottom-Up Merge Sort can be *efficiently* done for both arrays and linked lists.

Merge Sort: Bottom-Up Merge Sort for Arrays



- Time Complexity: should be the same as top-down, $\theta(n \log n)$
- Additional space:
  - no stack
  - but need $\theta(n)$ memory for merging anyway



Start: consider the original array as n singleton sub-arrays.
Then do the merging process:

- merging pairs of adjacent sub-arrays of size 1 = 2^0
- merging pairs of adjacent sub-arrays sub-arrays of size 2^1
- merging pairs of adjacent sub-arrays sub-arrays of size 2^2

…

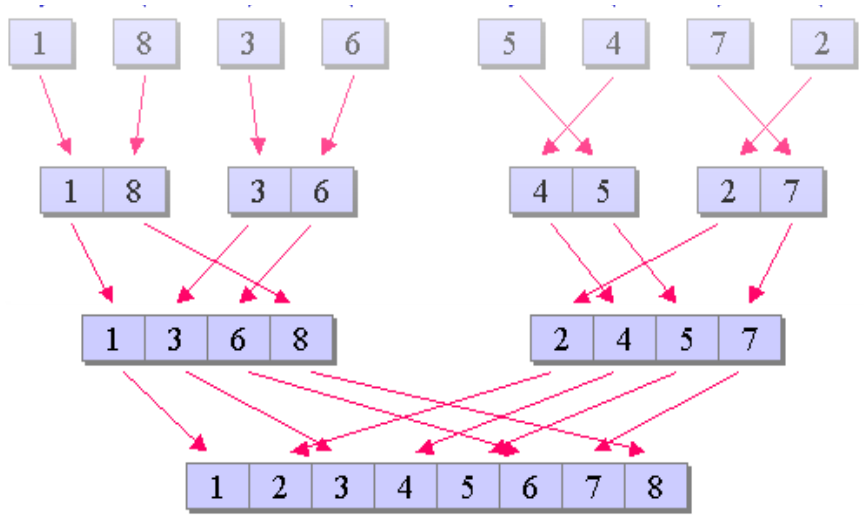
- merging pairs of adjacent sub-arrays sub-arrays of size $2^{\log_2 n - 1}$

Note:

- in round k, beware that the last pair might have just 1 sub-array, and the last sub-array might have less than 2^k elements

Merge Sort: Bottom-Up for Linked Lists

- Time Complexity: should be the same as top-down, $\theta(n \log n)$
- Additional space:
  - no stack, no memory for merging
  - but need memory for queue, it's $\theta($??? $)$



Use a queue!

initially put all singletons into an empty queue Q

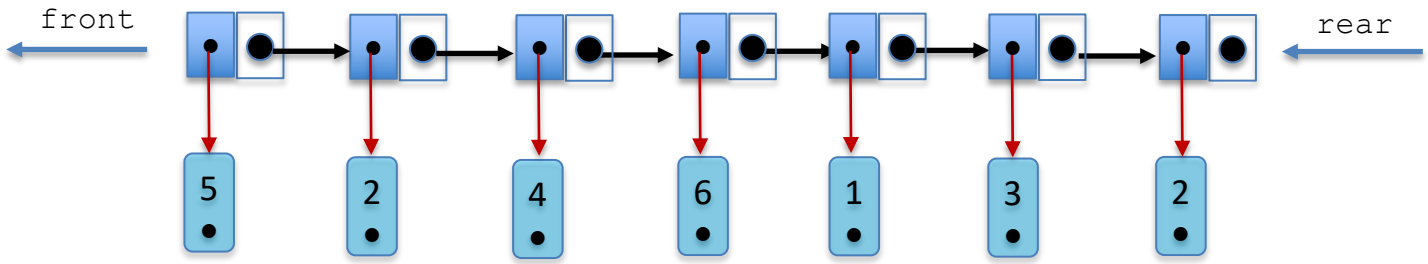
```
while (Q has 2 or more elements) {
   dequeue 2 elements
   merge them into one
   enqueue the merged element
}
// Q should have only one element
dequeue to get the sorted solution
```

Merge Sort in Linked Lists:

Bottom-Up for

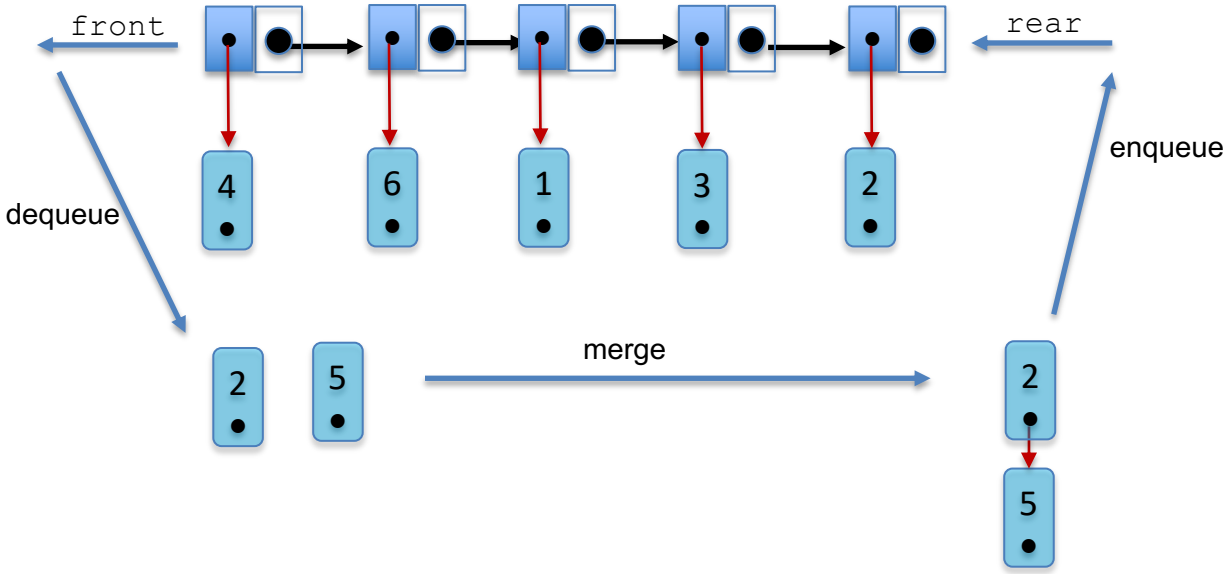
```
{
    5,
    2,
    4,
    6,
    1,
    3,
    2
}
```

1. Start with enqueuing all singleton (sorted) lists into Q:



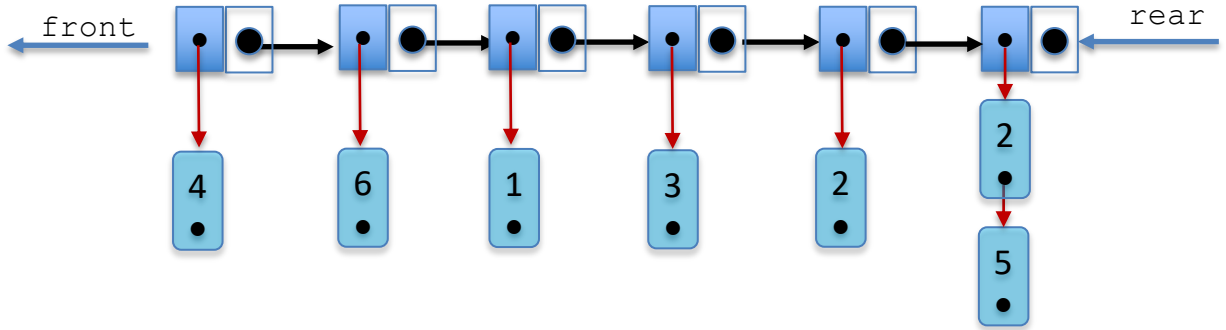
2.

- dequeue 2 sorted lists
- merge them into a single sorted list
- enqueue the sorted list

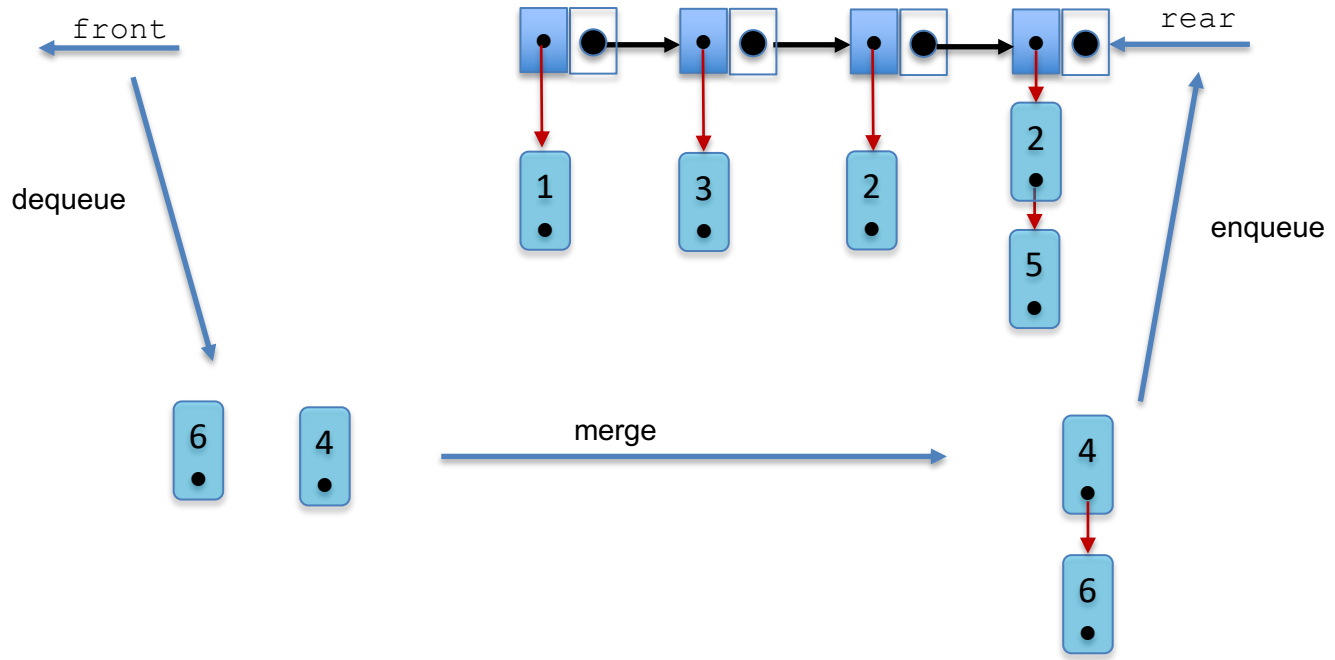


3.

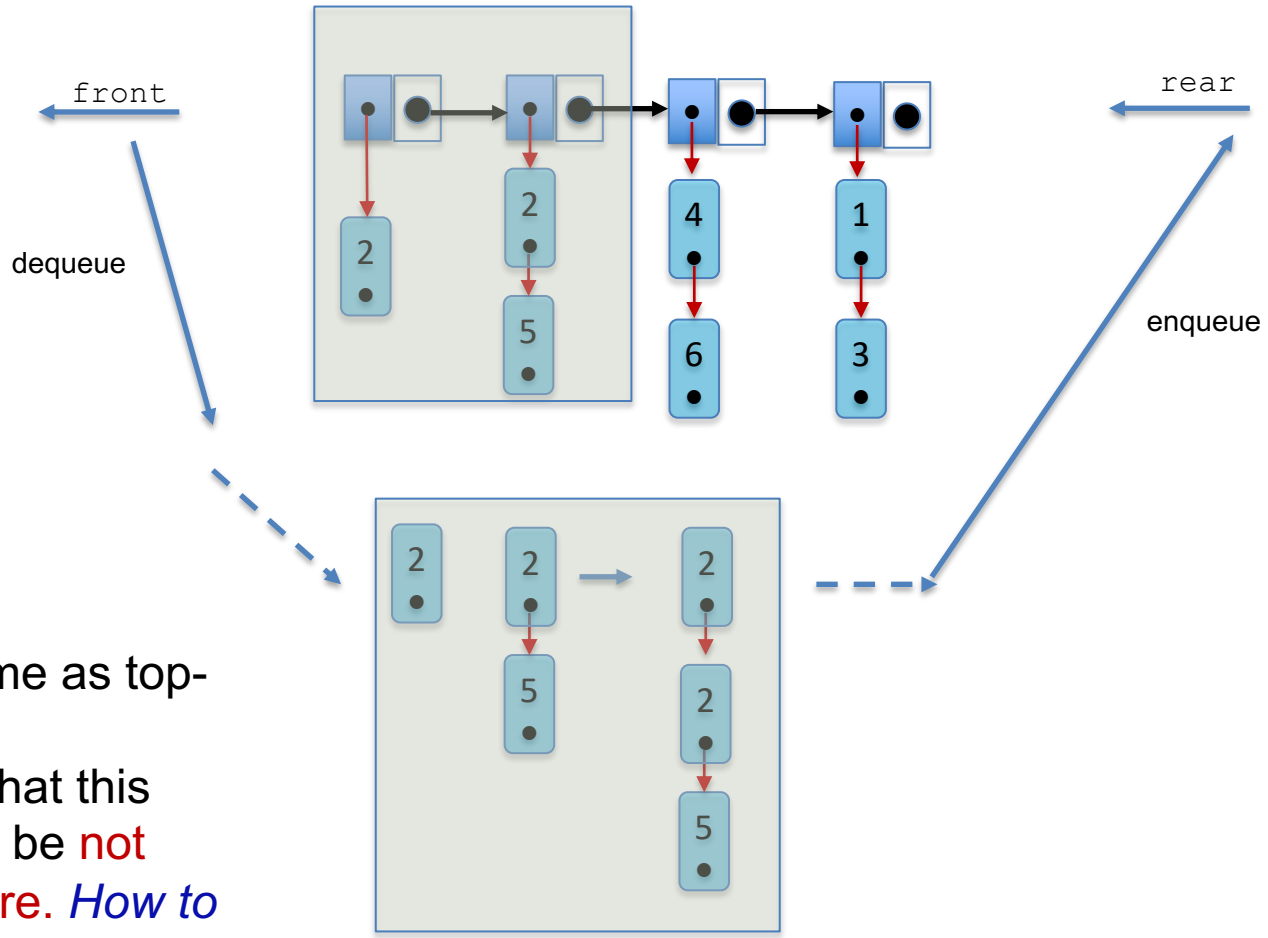
- repeat Step 2 until having a single list in Q (this last single list is the solution)



Merge Sort: Bottom-Up for 5,2,4,6,1,3,2



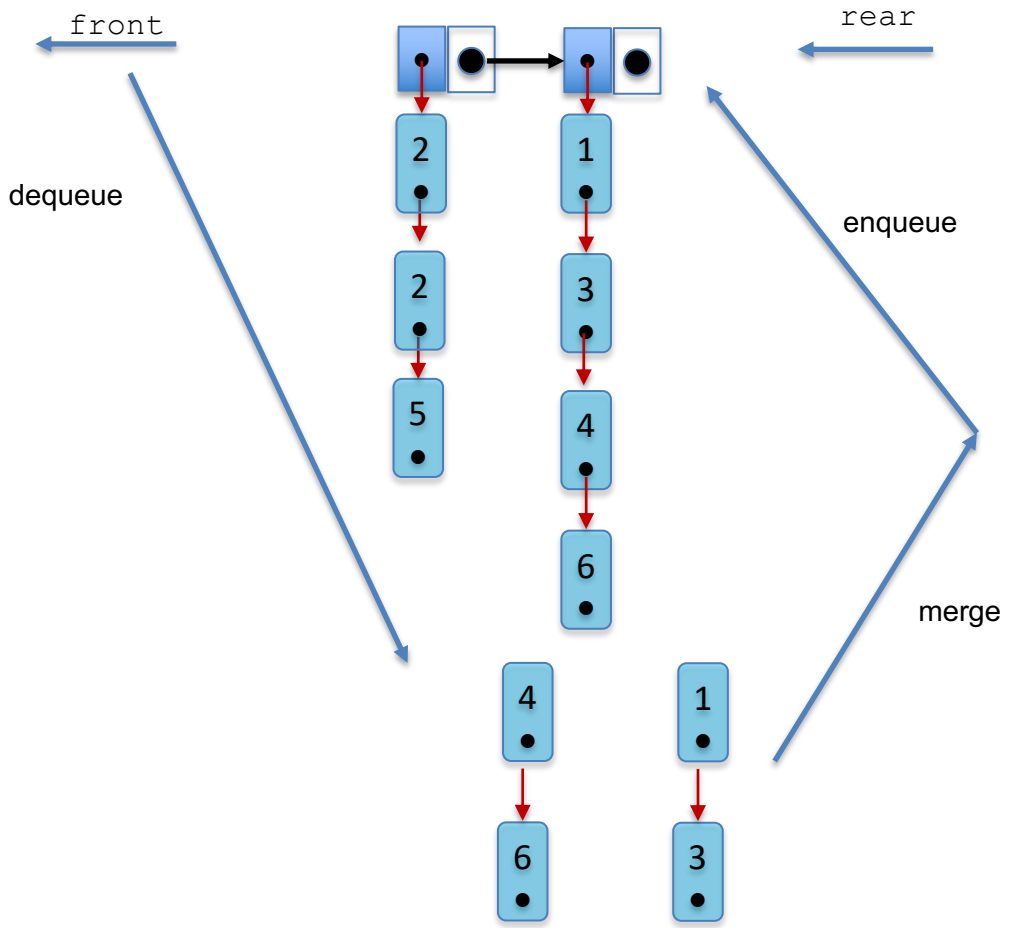
Merge Sort: Bottom-Up for 5,2,4,6,1,3,2



Note:

- Time complexity is the same as top-down
- The boxed merge shows that this bottom-up algorithm could be not stable if not careful like here. *How to make it stable?*
- Mergesort should be implemented as stable.

Merge Sort: Bottom-Up for 5,2,4,6,1,3,2

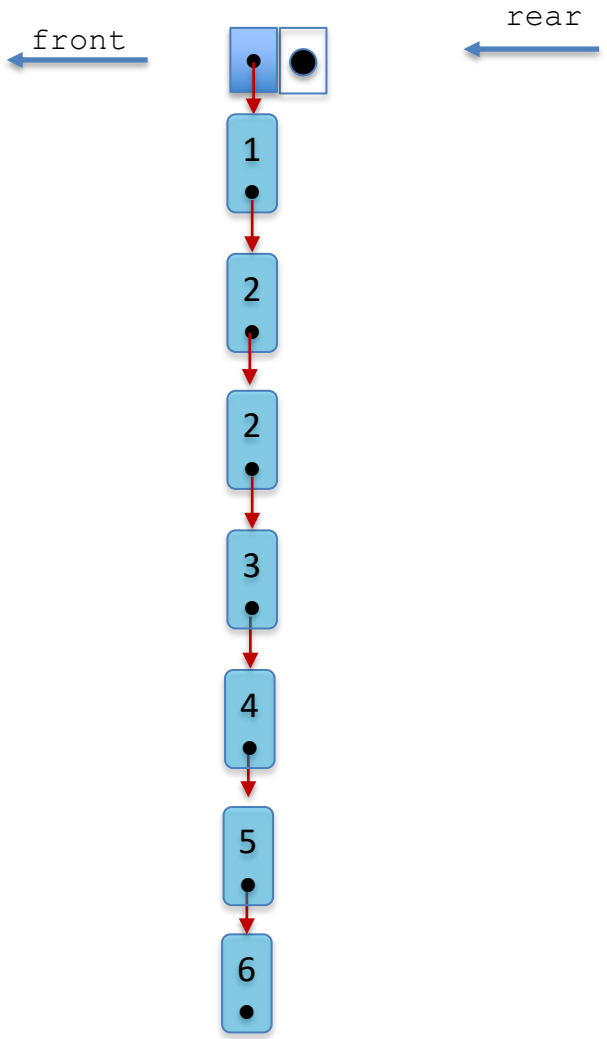


Merge Sort: Bottom-Up for 5,2,4,6,1,3,2

while Q has at least 2 elements:

- dequeue 2 sorted lists
- merge them into a single sorted list
- enqueue the merged list

At the end, the queue has only a single element. Dequeue that to get the final sorted list.



Additional memory need:

- *no recursive, no stack memory needed*
- *but, need $\theta(n)$ for the queue (more than for the stack ☹)*
- *no additional memory for merging*

To sum up, bottom-up mergesort for both arrays & linked lists:

- *$\theta(n \log n)$ time complexity*
- *$\theta(n)$ additional memory*

Top-down merge sort can be implemented using arrays:

- *$\theta(n \log n)$ time complexity*
- *$\theta(n)$ additional memory for merging*

But possible for linked lists:

- *$\theta(n \log n)$ time complexity*
- *$\theta(?)$ additional memory for merging*

mergesort: class exercises

For the sequence of character keys:

E X A M P L E

Show how mergesort work

a) top-down mergesort

b) bottom-up mergesort

MST Resources

Practice Test:

- around 15 multiple choice/ short answer questions for 30 minutes

Past MST:

- 2015 (complexity questions are relatively hard)
- 2017
- 2019

Topics in Workshops W1-W7:

- W1: C reviews, memory & pointers
- W2: Memory management, dynamic array, file IO
- W3: Linked Lists , intro to Complexity
- W4: Complexity; Stacks and Queues
- W5: Tree, Traversal, BST, AVL [***missing****: complete tree, deletion*]
- W6: Distribution Counting, Hashing
- W7: Sorting: properties; insertion, selection and quick sort
- W8: <not for this MST>

Class Exercises and Lab

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } d > \log_b a \\ \Theta(n^d \log n) & \text{if } d = \log_b a \\ \Theta(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

Class Exercises

Ex 1: Using the Master Theorem to solve the recurrence:

$T(n) = 5T(n/2) + n^2 + 9n\log n$

$T(1) = 1$

Ex 2: Using substitution, solve the recurrence (of best case quicksort, mergesort):

$T(n) = 2T(n/2) + n$

$T(1) = 1$

Ex 3: Show how top-down mergesort run for:

8 2 6 7 4 5

Ex 4: Show how bottom-up mergesort run for the sequence in Ex 3

---------- For MST ----------

5. Insert the sequence in **Ex 3** into an initially-empty AVL

6 quick-sort the sequence

7. double hashing for 8 2 6 7 4 3 h(x)= x%6, h2(x)= x%5 (table size m= 6)

8. selection-sort, insertion-sort the sequence

LAB

W8.3: bottom-up mergesort for linked lists:

- need a queue of linked lists
- linked lists: just need to keep track of the head
- element in queue= {list, next}

W8.4: bottom-up mergesort for arrays:
if done cleverly, we don't need a queue: