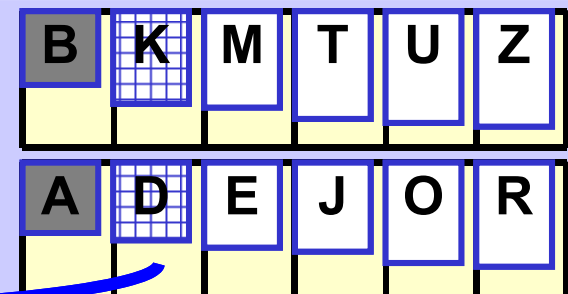
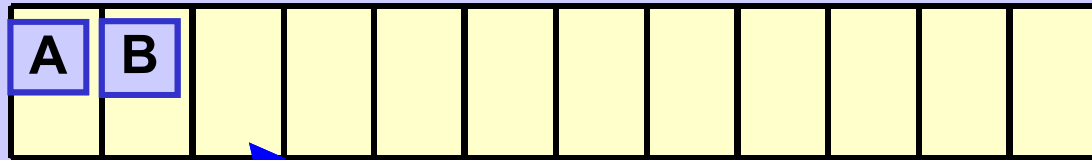
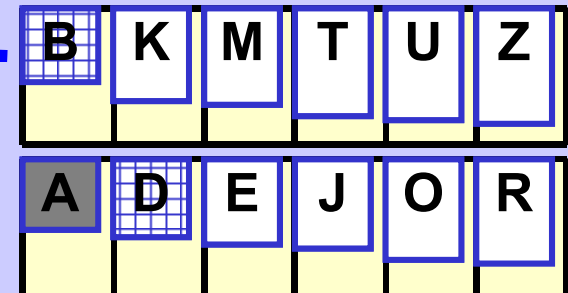
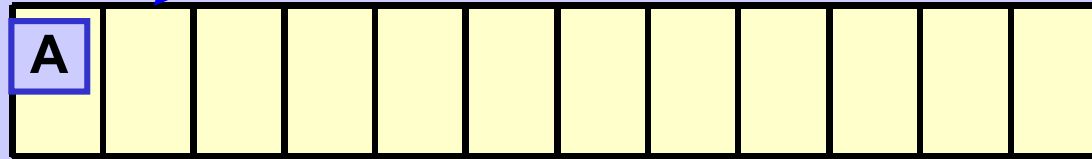
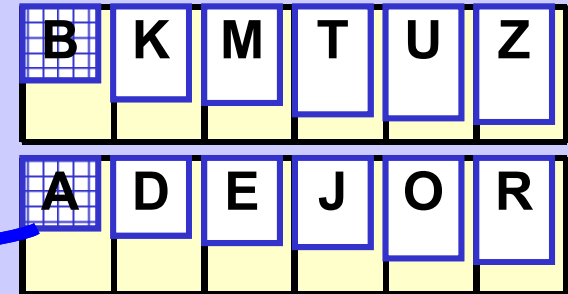
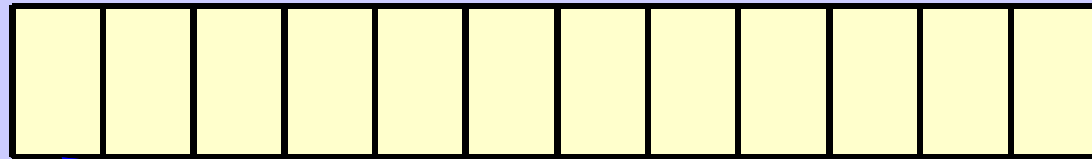


Merge sort

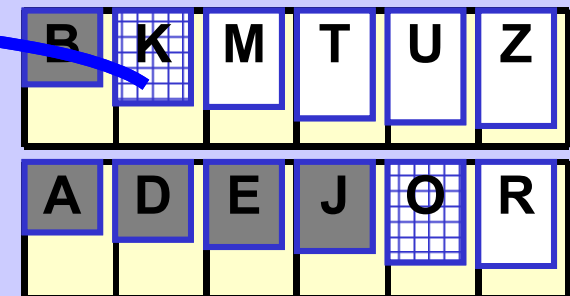
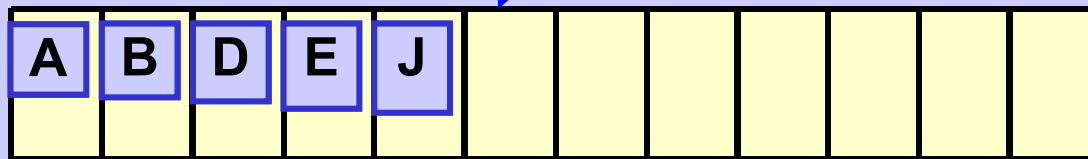
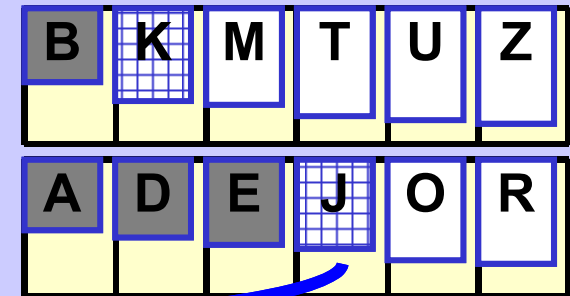
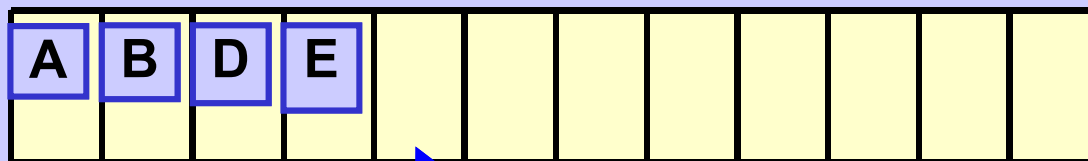
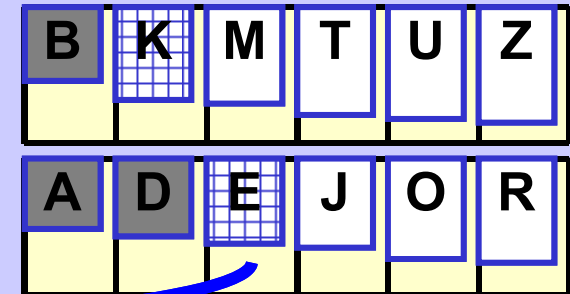
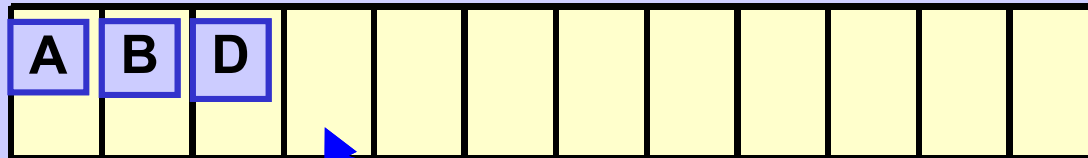
Merge two sorted arrays

Compared elements



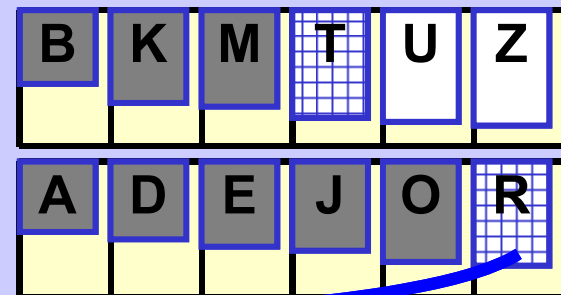
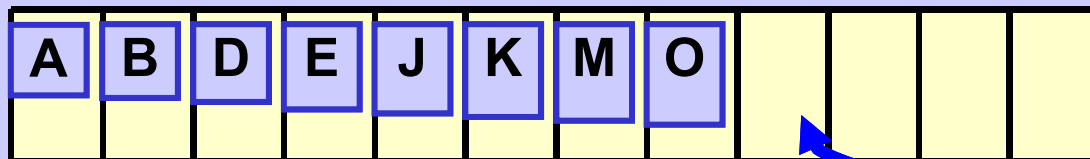
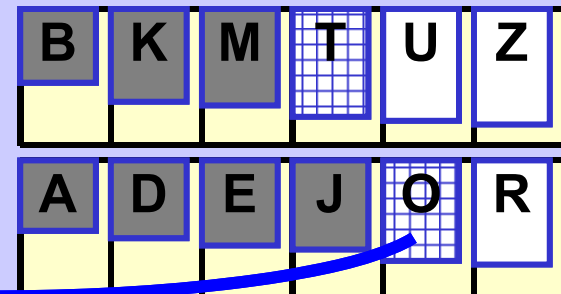
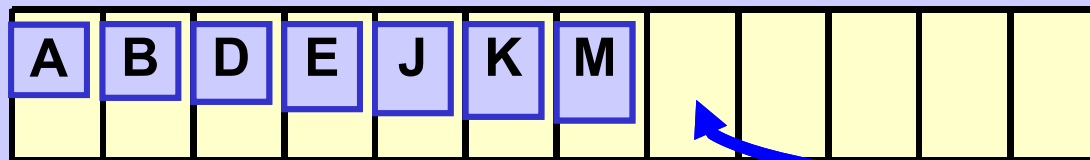
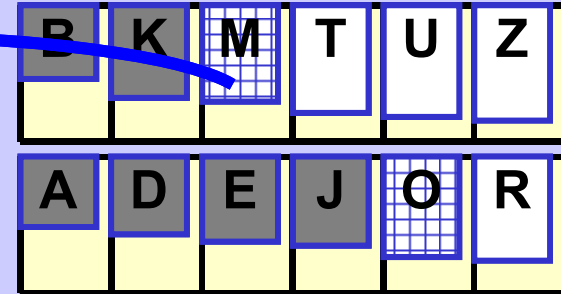
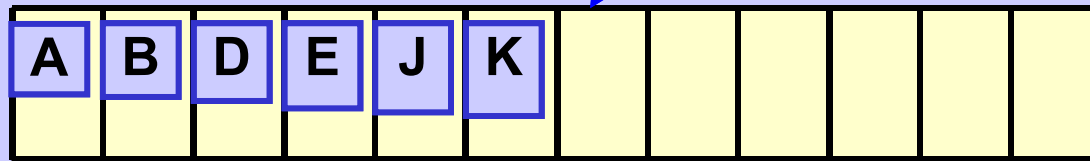
Merge sort

Merge two sorted arrays - cont.



Merge sort

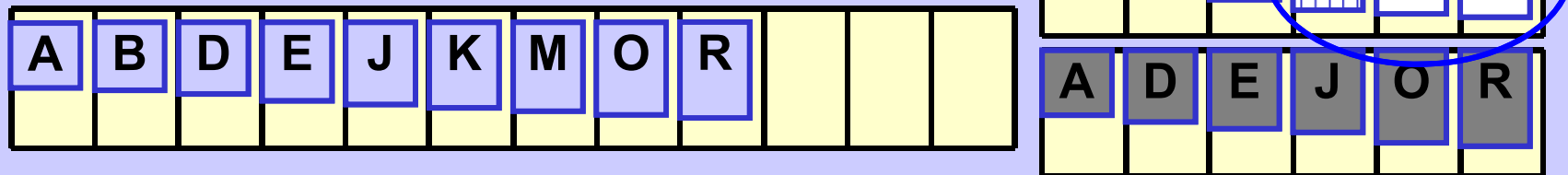
Merge two sorted arrays - cont.



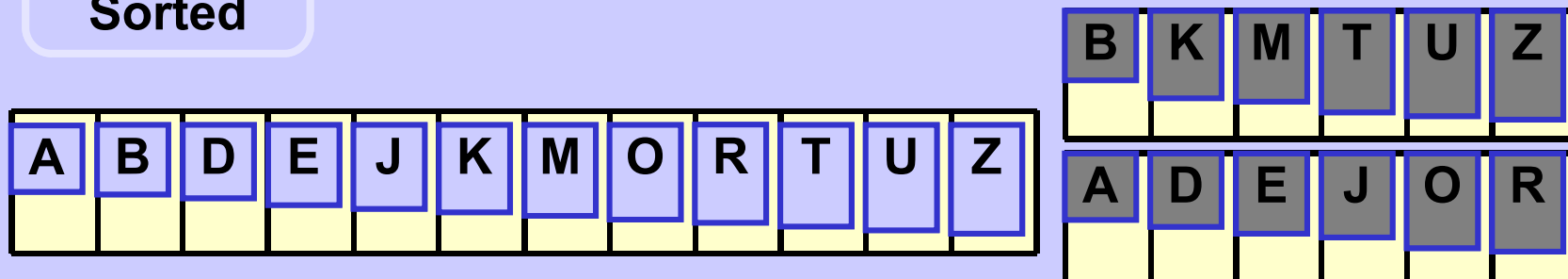
Merge sort

Merge two sorted arrays - cont.

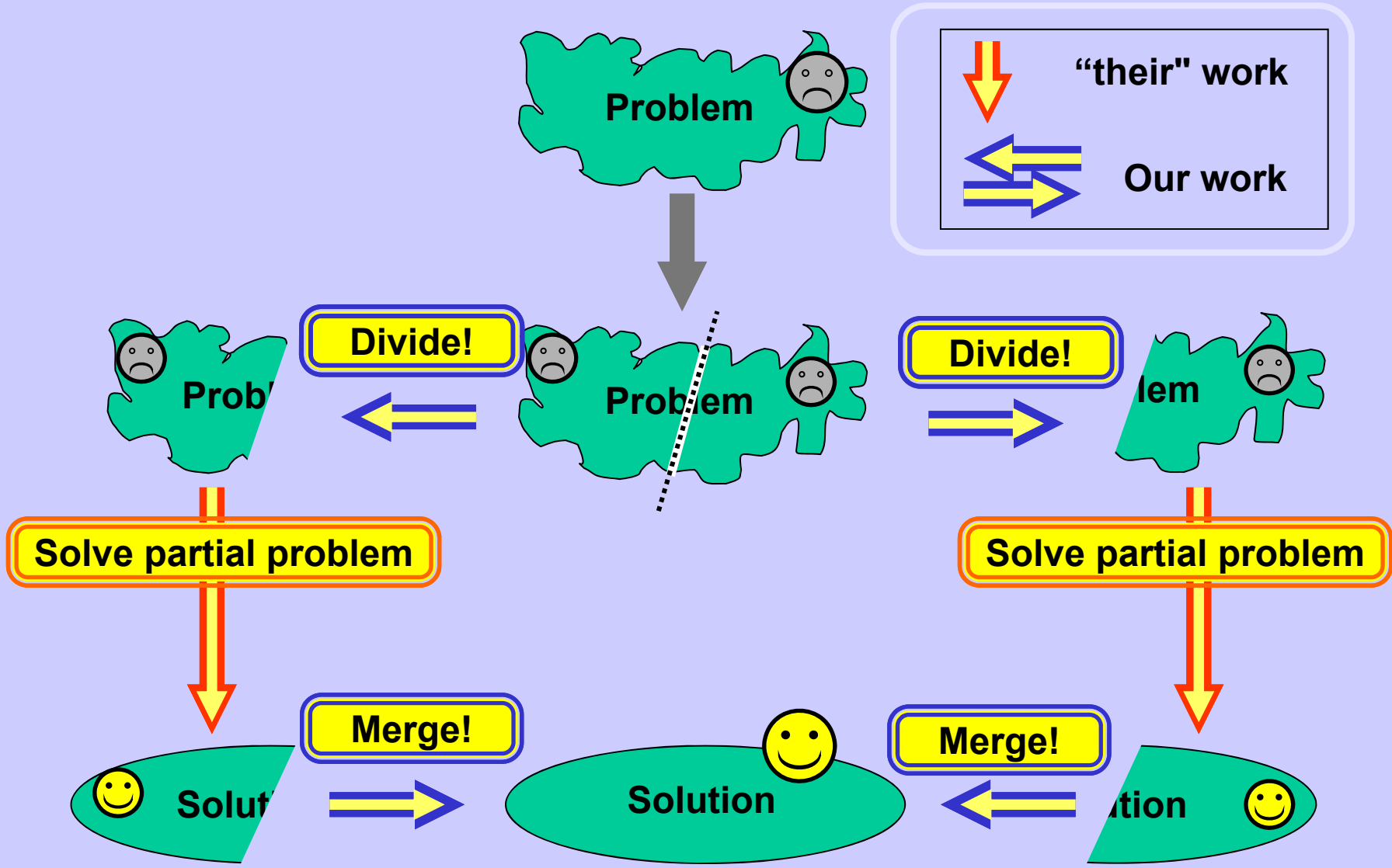
Copy the rest



Sorted



Divide and conquer! Divide et impera!



Merge sort

Unsorted

K	Z	U	B	M	T	E	A	J	R	D	O

Divide!

Process separately

K	Z	U	B	M	T

Sort!

B	K	M	T	U	Z

E	A	J	R	D	O

Sort!

A	D	E	J	O	R

Conquer!

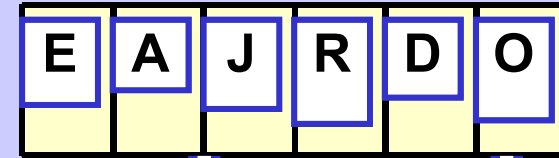
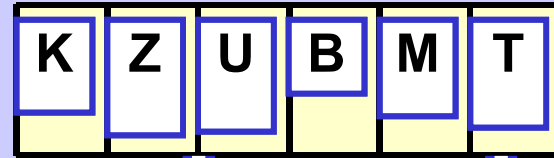
Merge!

Sorted

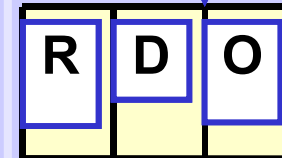
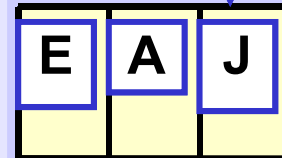
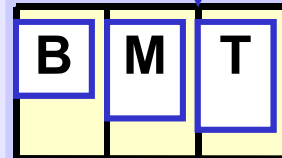
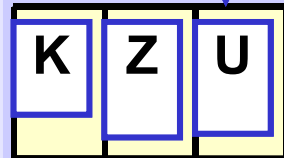
A	B	D	E	J	K	M	O	R	T	U	Z

Merge sort

Unsorted



Divide!



Divide!

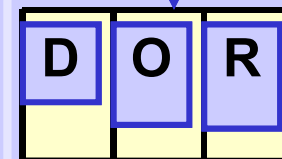
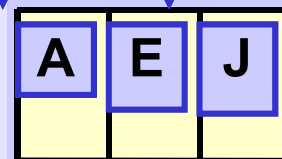
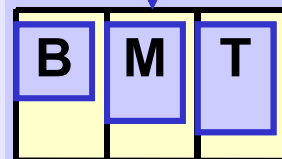
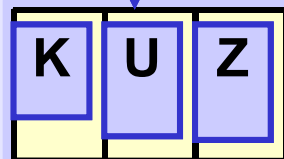
Divide!

.....

... ..

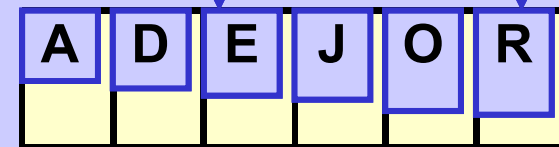
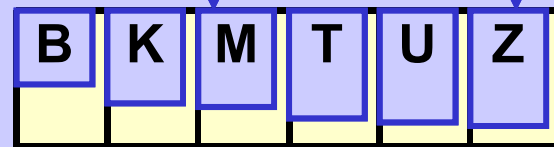
Merge!

Merge!



Merge!

Sorted



Merge sort

```
def _mergeSort (arr, auxArr, low, high):  
  
    if low >= high: return      # too small!  
    half = (low+high) // 2  
  
    # sort to auxArr  
    _mergeSort(arr, auxArr, low, half);      # left half  
    _mergeSort(arr, auxArr, half+1, high); # right half  
    merge(arr, auxArr, low, high)  
  
    # copy back from auxArr  
    for i in range(low, high+1):  
        arr[i] = auxArr[i]
```


Merge sort

```
def merge (inArr, outArr, low, high):  
    half = (low+high) // 2  
    i1 = low  
    i2 = half + 1  
    j = low;  
    # compare and merge  
    while i1 <= half and i2 <= high:  
        if inArr[i1] <= inArr[i2]:  
            outArr[j] = inArr[i1]; i1 += 1  
        else:  
            outArr[j] = inArr[i2]; i2 += 1  
        j += 1  
    # copy the rest  
    while i1 <= half:  
        outArr[j] = inArr[i1]; i1 += 1; j += 1;  
    while i2 <= high:  
        outArr[j] = inArr[i2]; i2 += 1; j += 1;
```

Merge sort

Asymptotic complexity

Divide! $\log_2(n)$ times \Rightarrow
 \Rightarrow Merge! $\log_2(n)$ times

Divide! $\Theta(1)$ operations

Merge! $\Theta(n)$ operations

Total $\Theta(n) \cdot \Theta(\log_2(n)) = \Theta(n \cdot \log_2(n))$ operations

Asymptotic complexity of Merge sort is $\Theta(n \cdot \log_2(n))$

Merge sort

Stability

Divide! Does not move the elements.

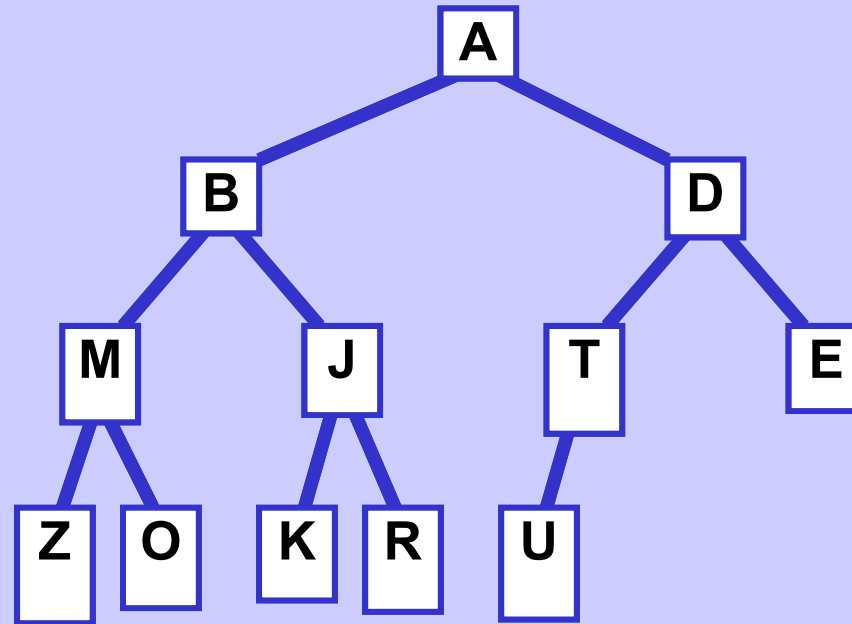
Merge! “ if ($in[i1] \leq in[i2]$) { $out[j] = in[i1]$; ...”

When the two compared and merged elements are equal, merge the left one first.

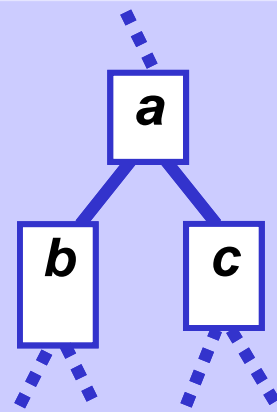
Merge sort is stable.

Heap sort

Heap



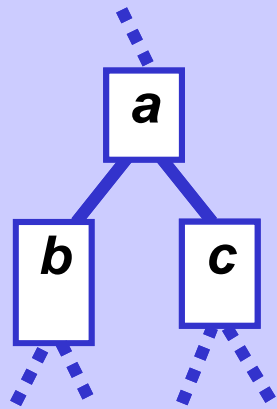
Heap property.



$$a \leq b \ \&\& \ a \leq c$$

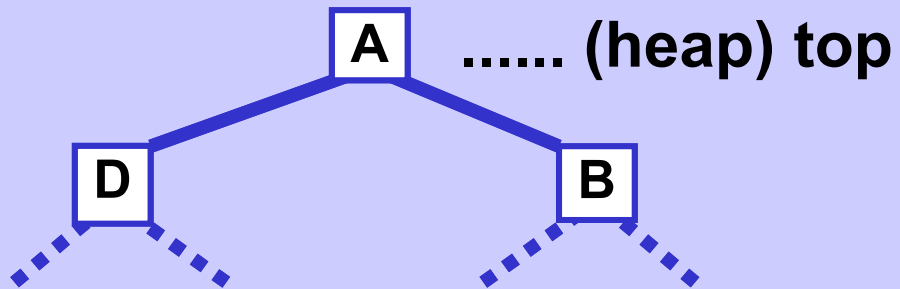
Heap sort

Terminology



a predecessor, parent of **b** **c**

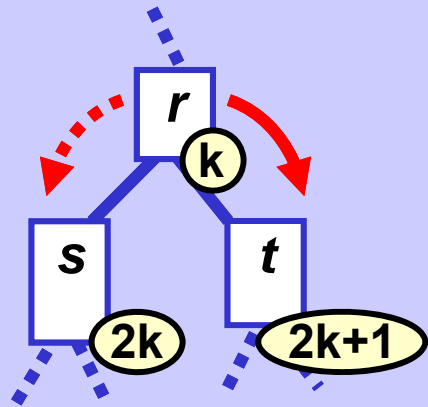
b, **c** successor, child of **a**



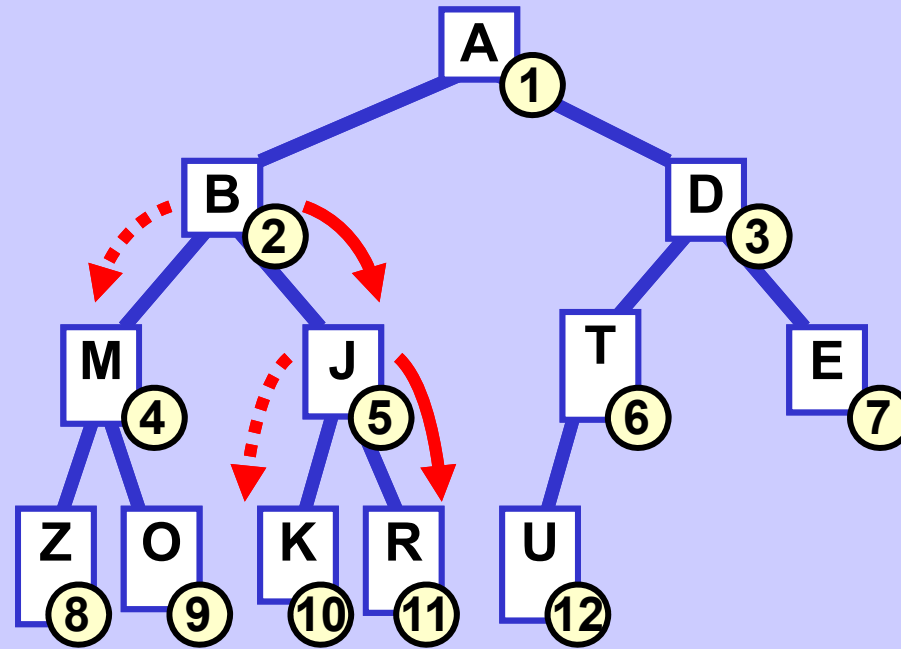
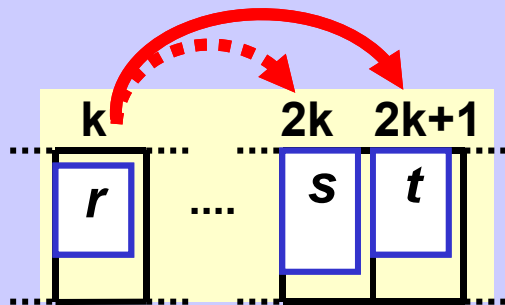
A (heap) top

Heap sort

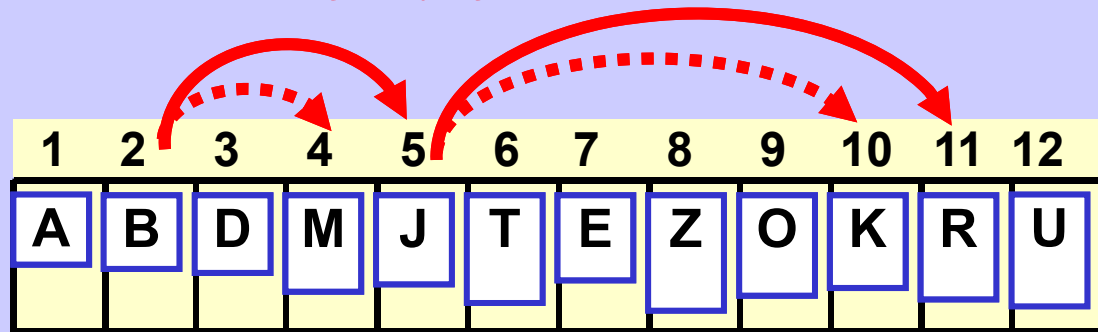
Heap stored in an array



children



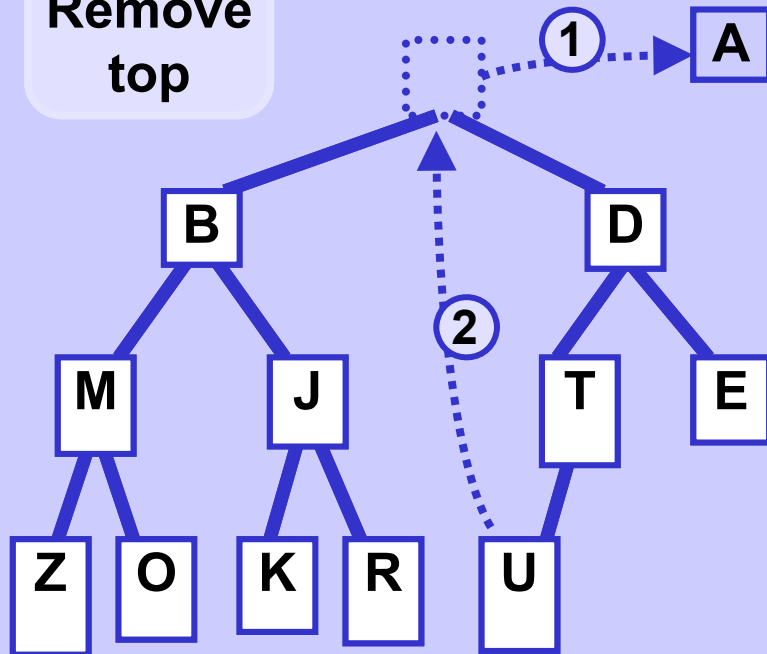
children



Heap repair

Top removed (1)

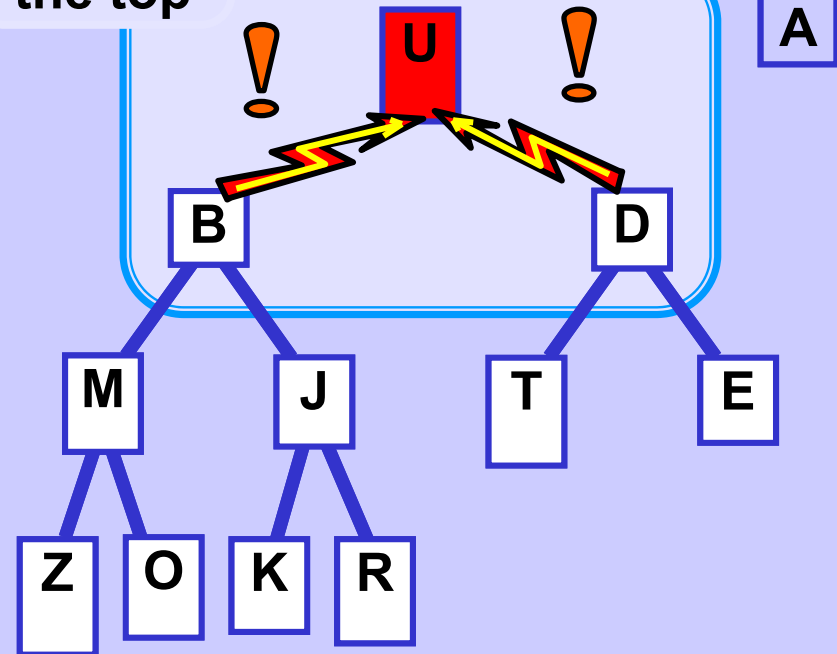
① Remove top



②

last \longrightarrow first

③ Put at the top



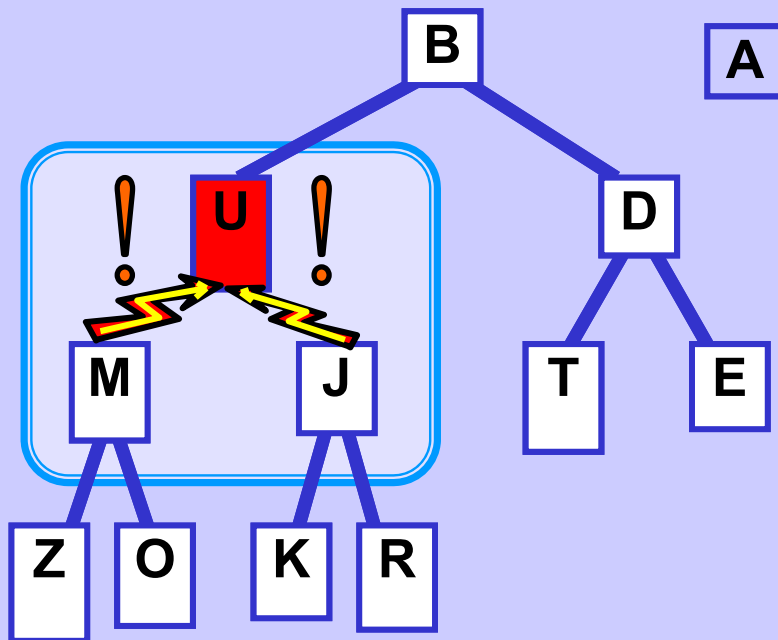
$U > B, U > D, \underline{B < D}$
 \Rightarrow swap $B \leftrightarrow U$

Heap repair

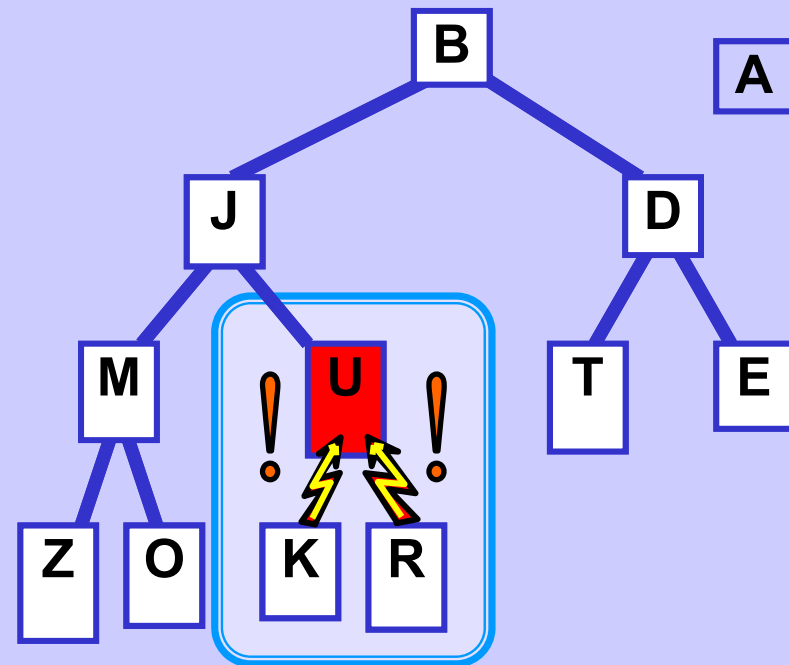
Top removed (2)

③

Put at the top - cont...



$U > M, U > J, \underline{J < M}$
 \Rightarrow swap $J \leftrightarrow U$

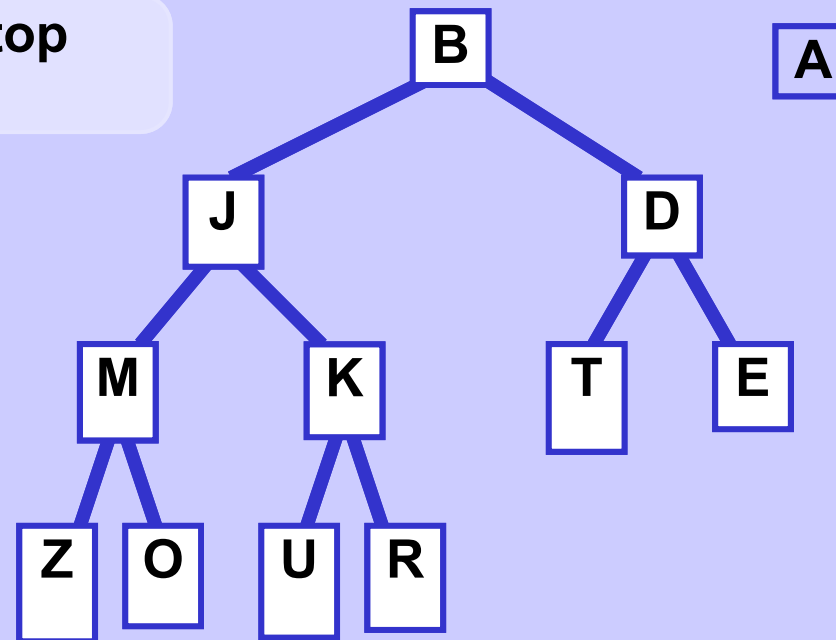


$U > K, U > R, \underline{K < R}$
 \Rightarrow swap $K \leftrightarrow U$

Heap repair

Top removed (3)

③ Put at the top
- done.

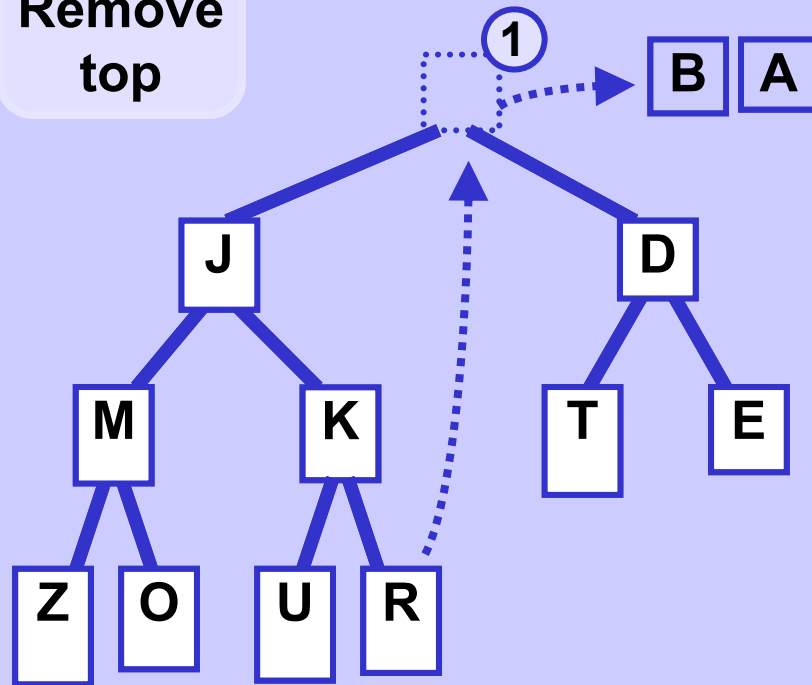


New heap

Heap repair

Top removed II (1)

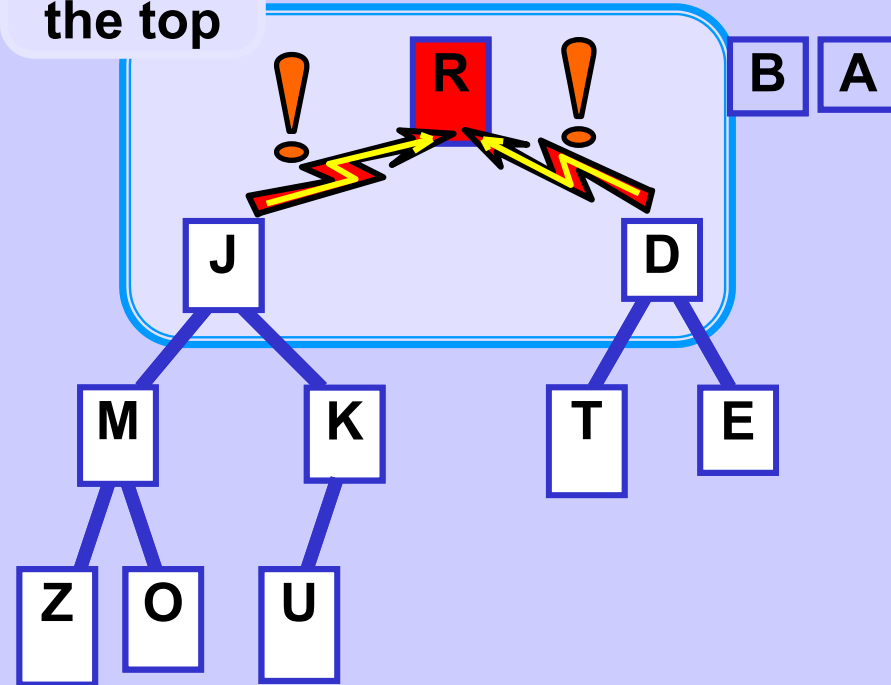
① Remove top



②

last \longrightarrow first

③ Put at the top



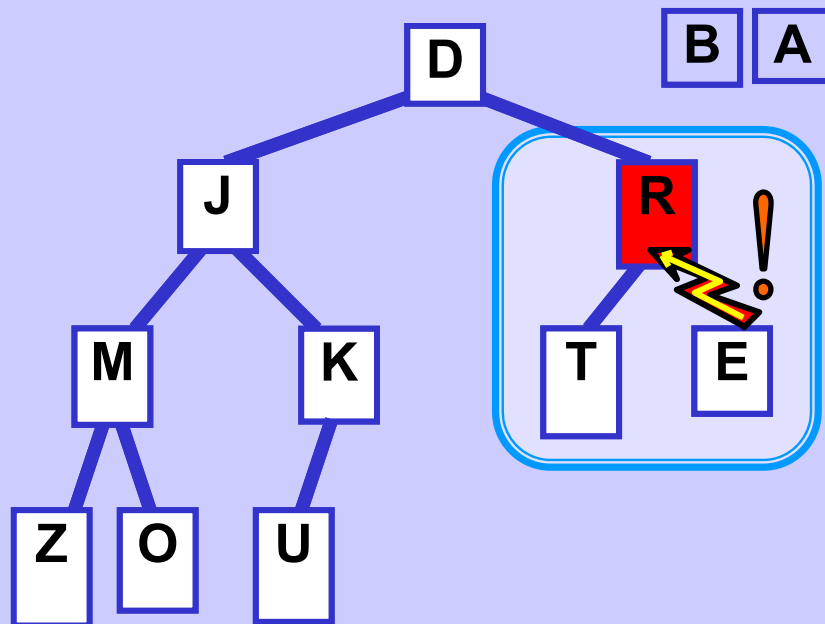
$R > J, R > D, \underline{D} < J$
 \Rightarrow swap $D \leftrightarrow R$

Heap repair

Top removed II (2)

③

Put at the top - cont.

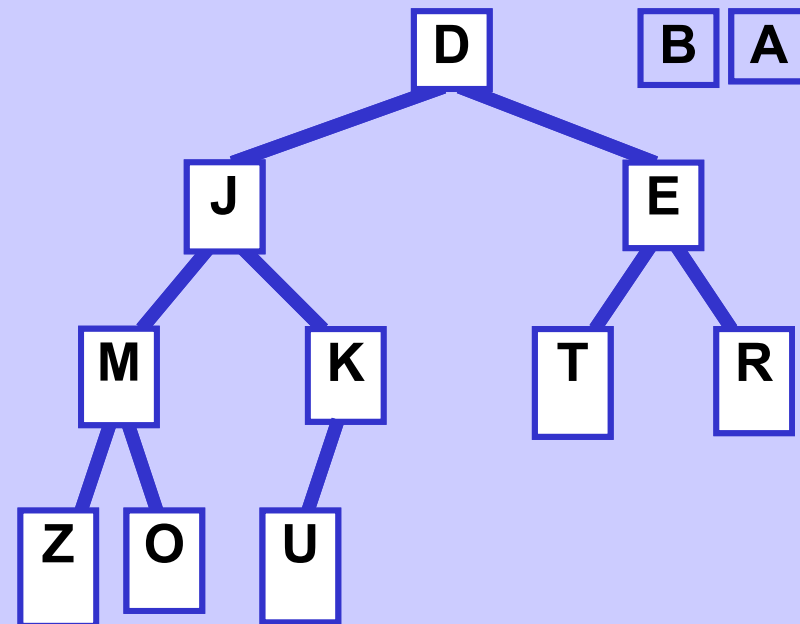


$R < T, R > E$
 \Rightarrow swap $E \leftrightarrow R$

Top removed II (3)

③

Put at the top
 - done.

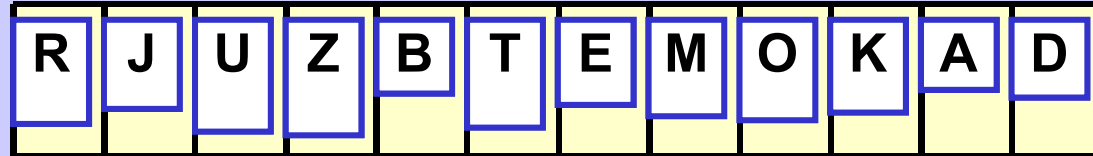


New heap

Heap sort

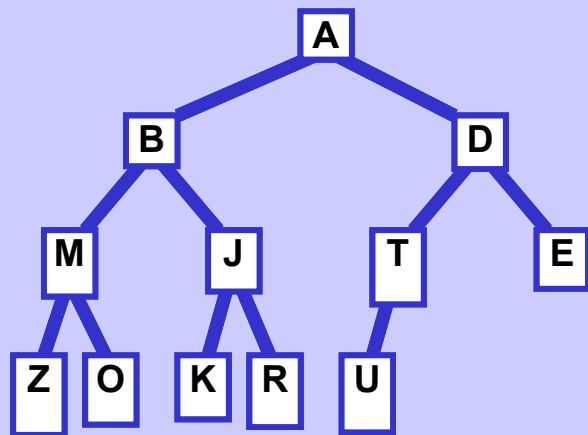
I

Unsorted



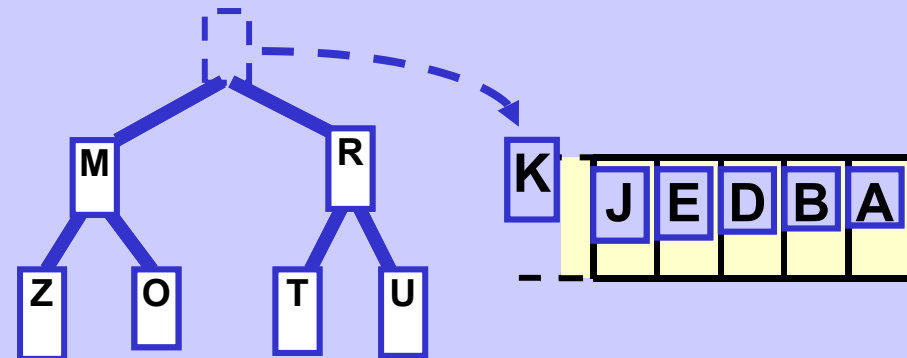
II

Make a heap



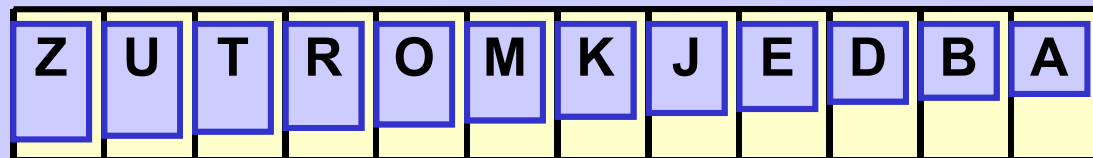
III

for (i = 0; i < n; i++)
a[i] = "remove top";

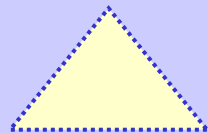
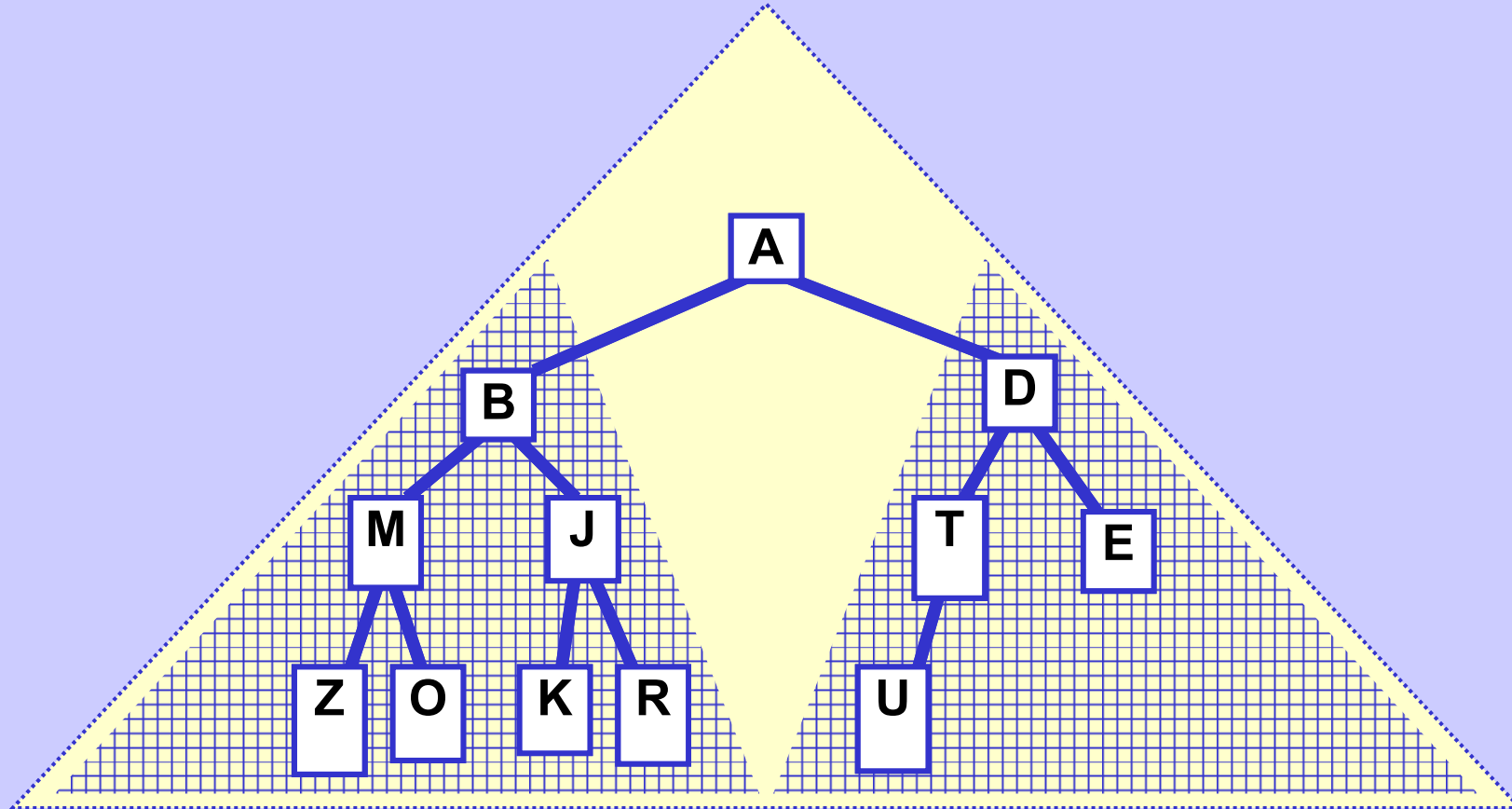


IV

Sorted



Recursive property of "Being a heap"



Is a heap

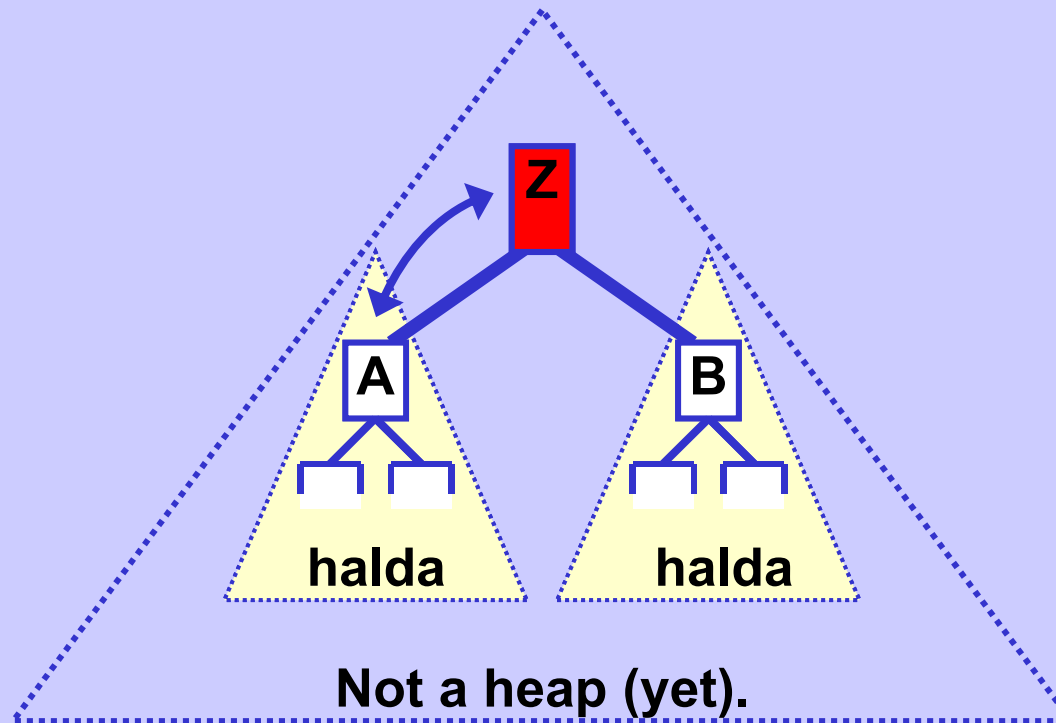


is a heap and



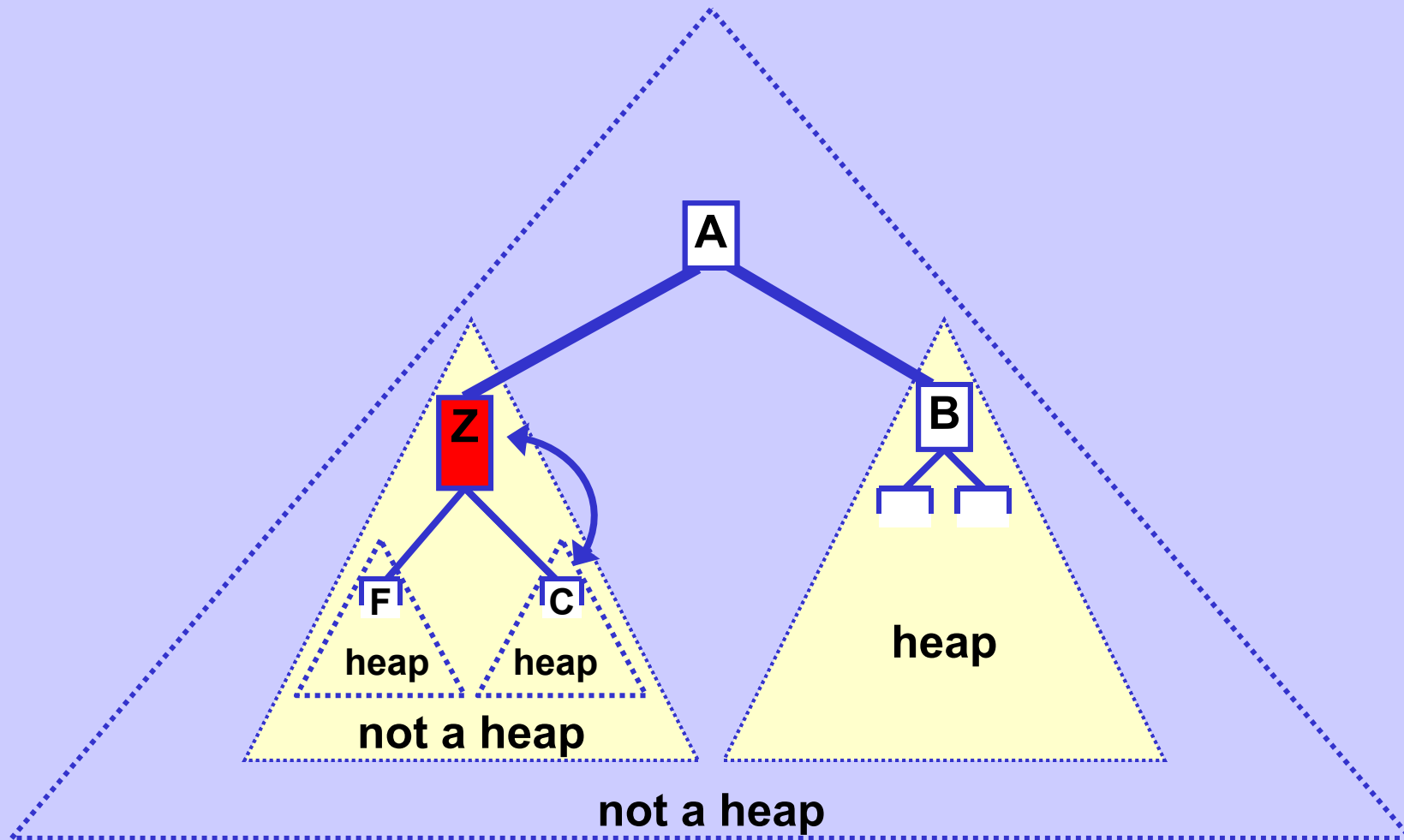
is a heap.

Make one bigger heap from two smaller ones

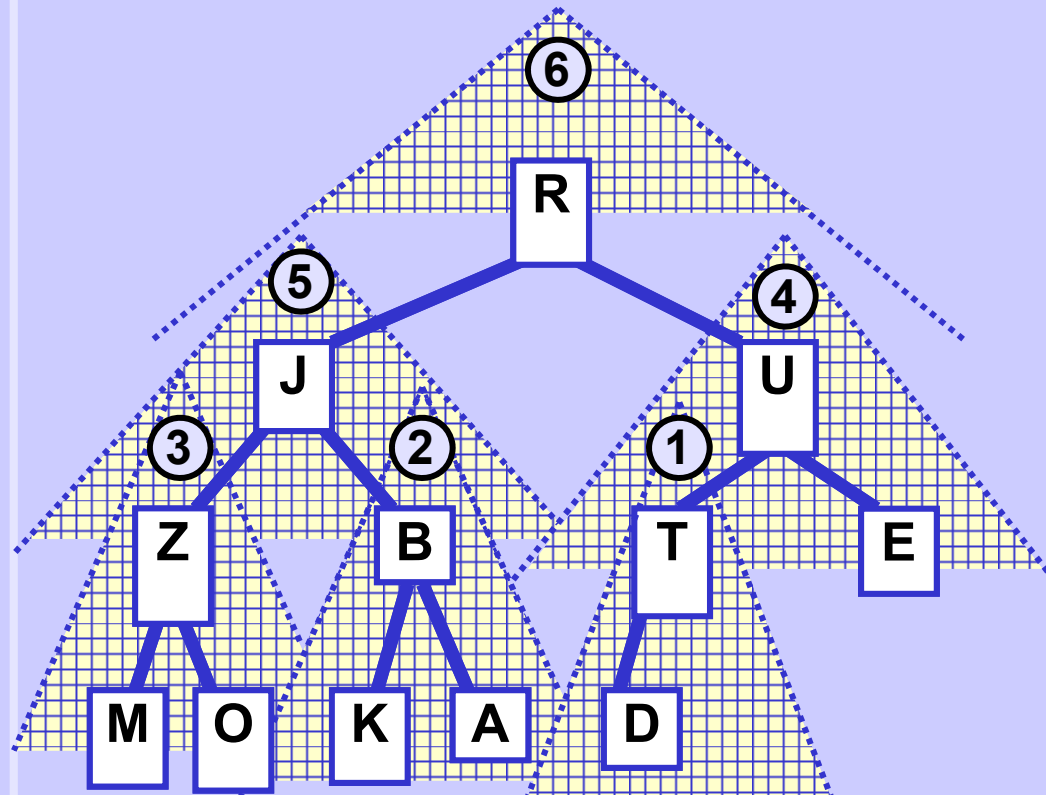


$Z > A$ or $Z > B$
 \Rightarrow swap: $Z \leftrightarrow \min(A, B)$

Make one bigger heap from two smaller ones



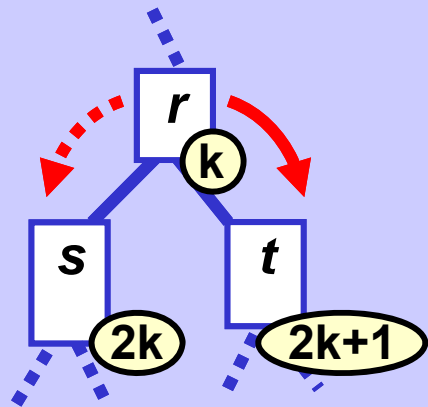
Create a heap



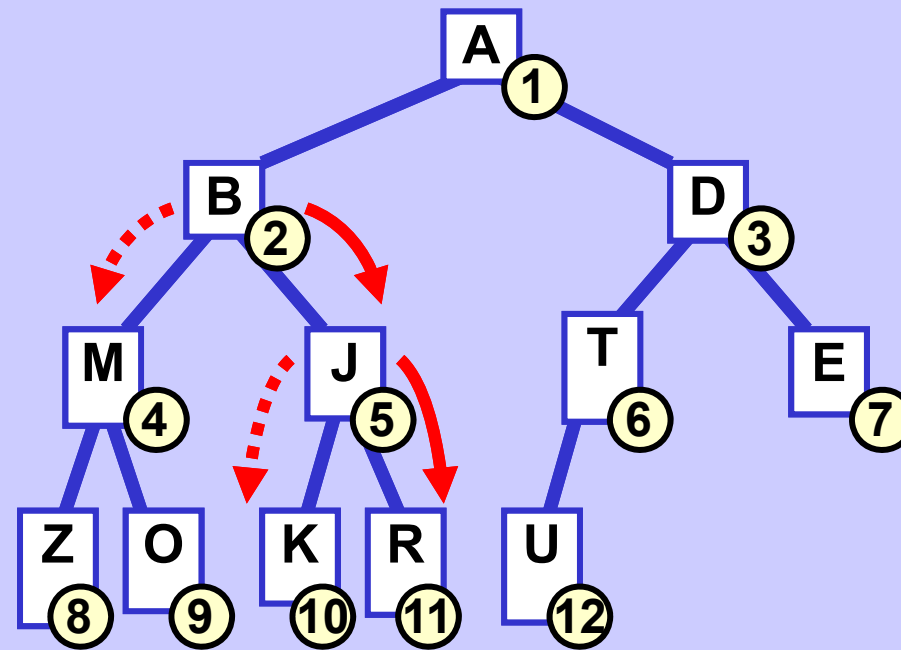
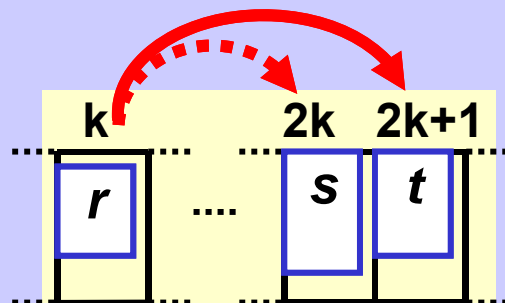
Make a heap in ① ...
 ... make a heap in ② ...
 ... make a heap in ③ ...
 ... make a heap in ④ ...
 ... make a heap in ⑤ ...
 ... make a heap in ⑥ ...
 ... and the whole heap
 is complete.

Heap in an array

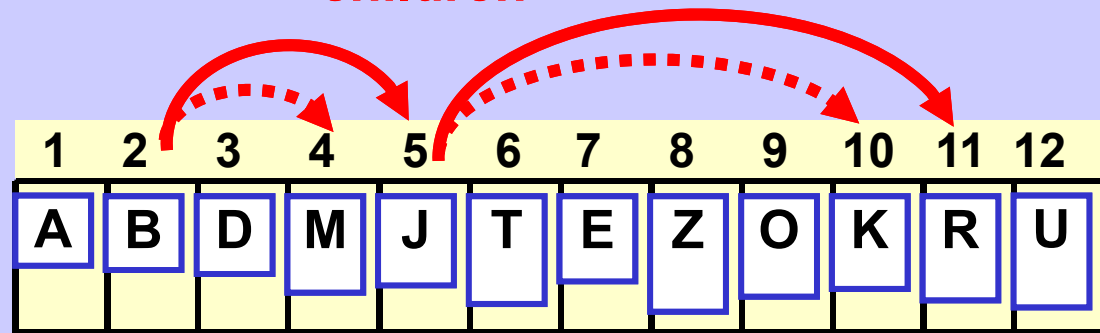
Heap stored in an array



children



children



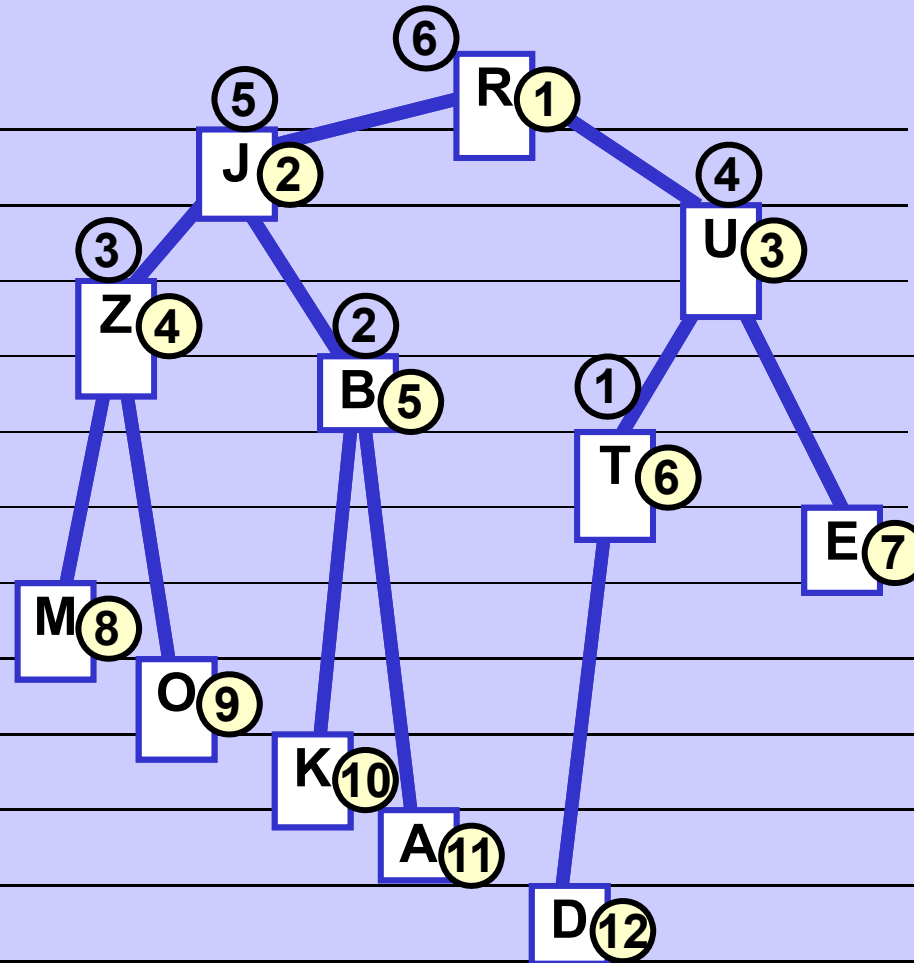
Heap in an array

Array

⑥	1	R
⑤	2	J
④	3	U
③	4	Z
②	5	B
①	6	T
	7	E
	8	M
	9	O
	10	K
	11	A
	12	D

Array elements ordered randomly

Not a heap



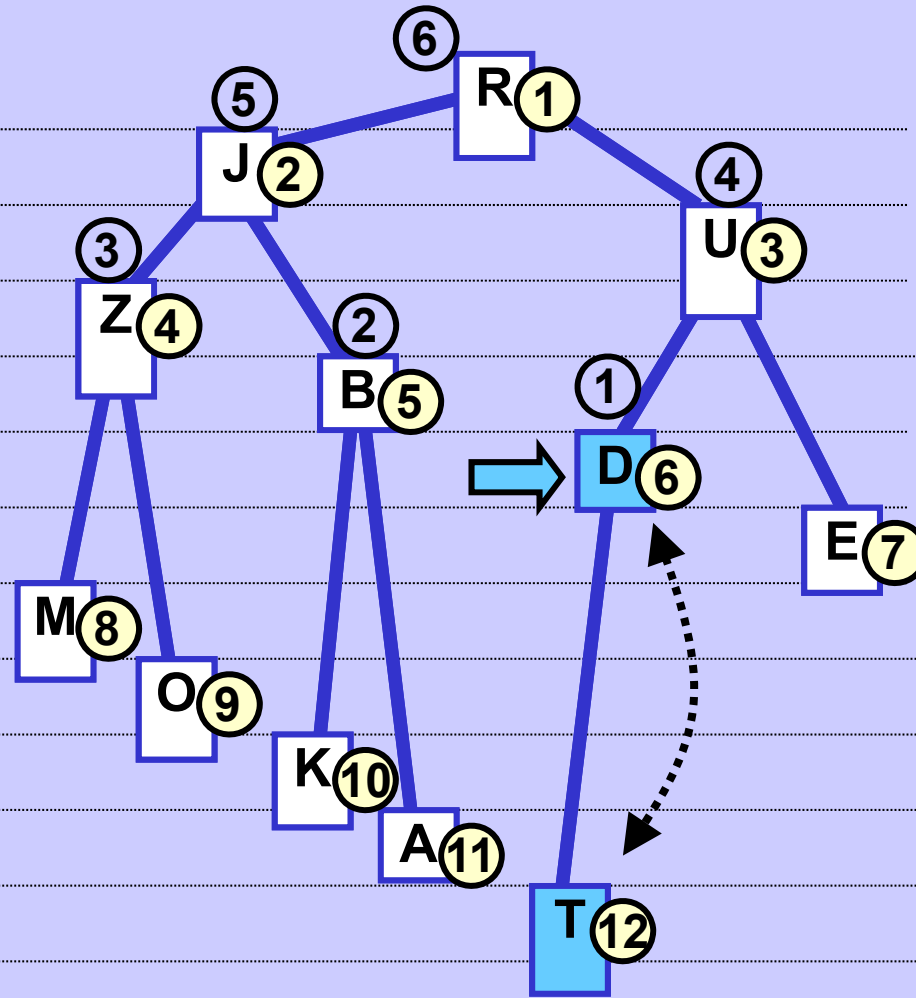
Heap in an array

Array

⑥	1	R
⑤	2	J
④	3	U
③	4	Z
②	5	B
①	6	D
	7	E
	8	M
	9	O
	10	K
	11	A
	12	T

➔ Currently created heap

⋯➔ Moves

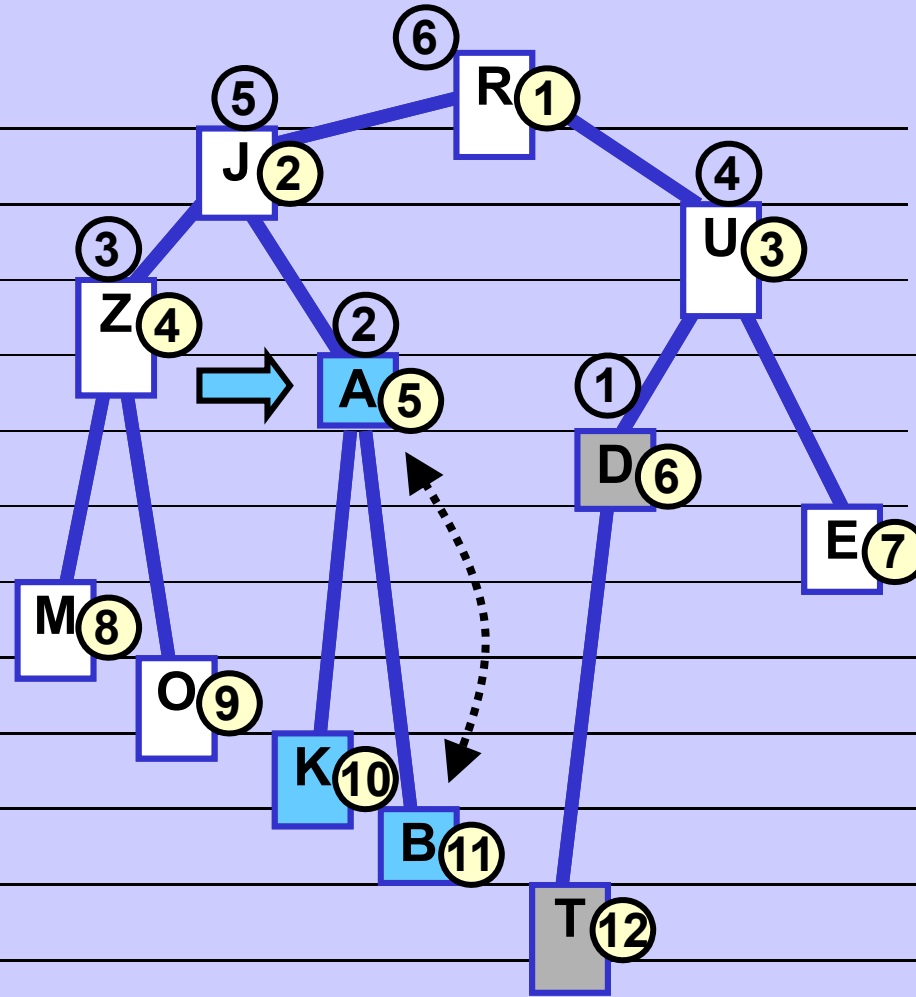


Heap in an array

Array

⑥	1	R
⑤	2	J
④	3	U
③	4	Z
②	5	A
①	6	D
	7	E
	8	M
	9	O
	10	K
	11	B
	12	T

Earlier heap(s)
 Currently created heap
 Moves

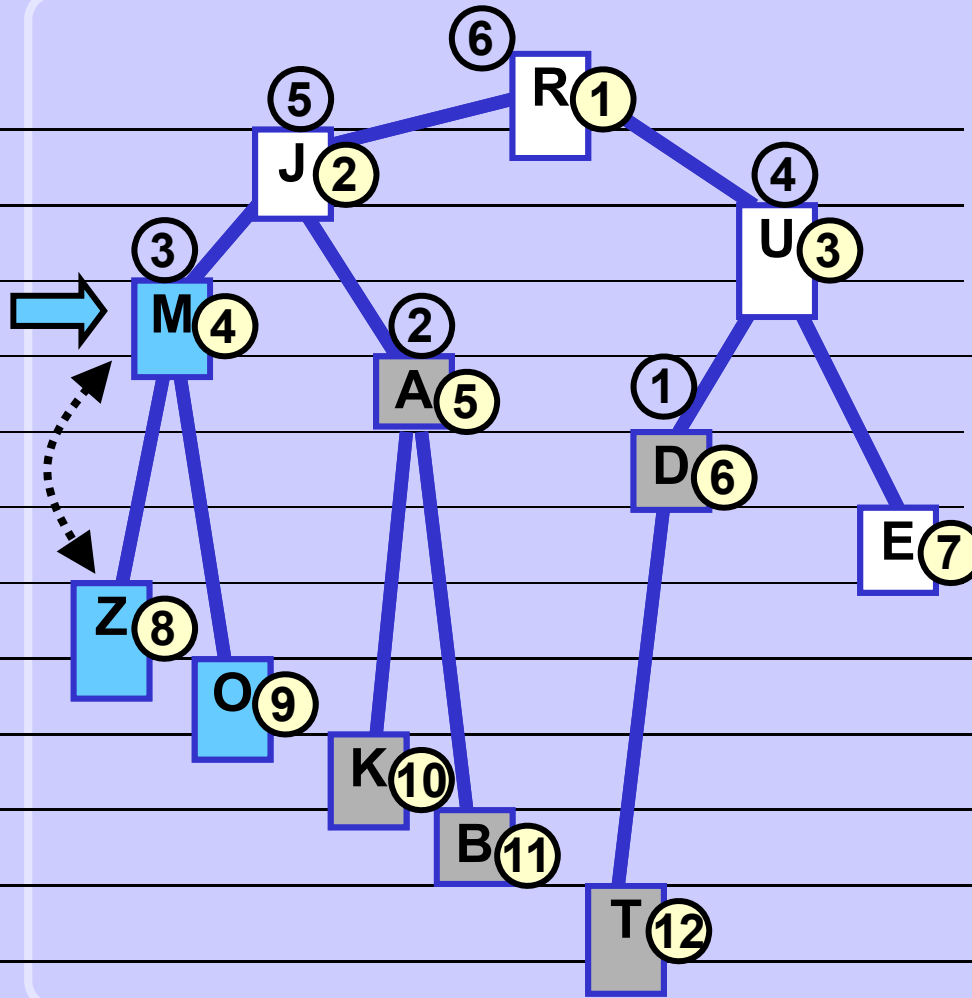


Heap in an array

Array

⑥	1	R
⑤	2	J
④	3	U
③	4	M
②	5	A
①	6	D
	7	E
	8	Z
	9	O
	10	K
	11	B
	12	T

Earlier heap(s)
 Currently created heap
 Moves



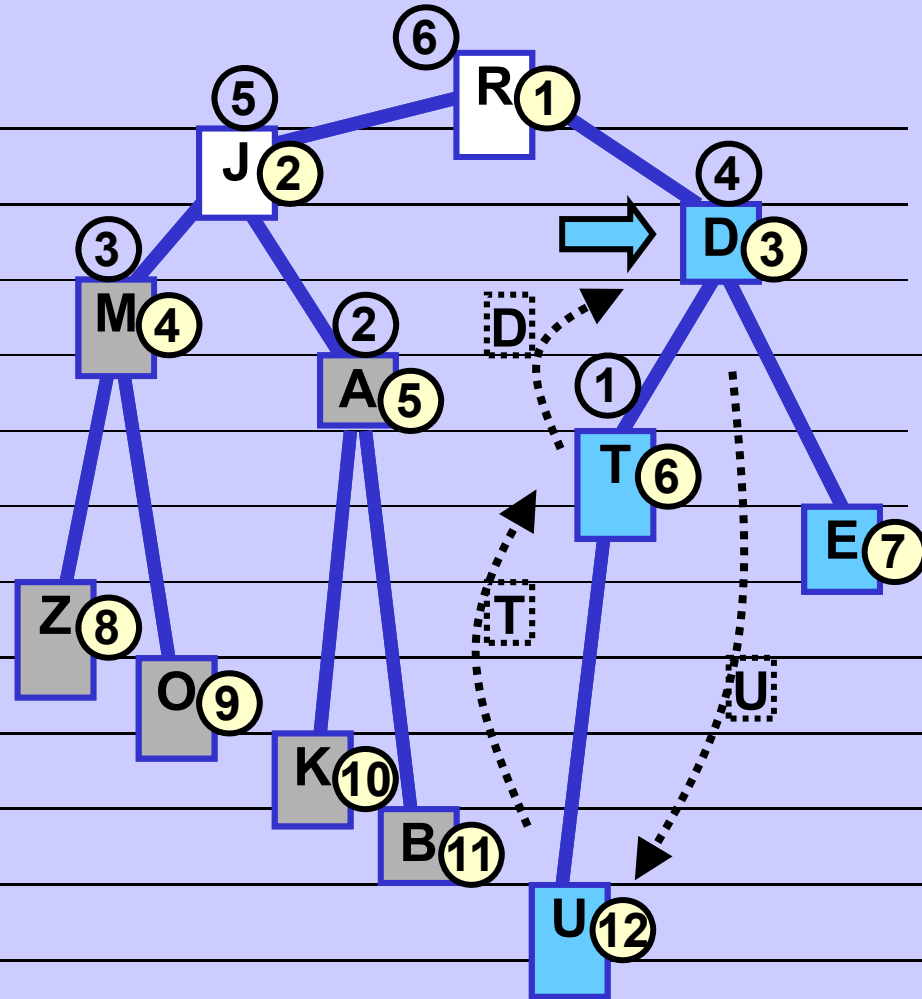
Heap in an array

Array

⑥	1	R
⑤	2	J
④	3	D
③	4	M
②	5	A
①	6	T
	7	E
	8	Z
	9	O
	10	K
	11	B
	12	U

Earlier heap(s)
 Currently created heap

➔ Moves

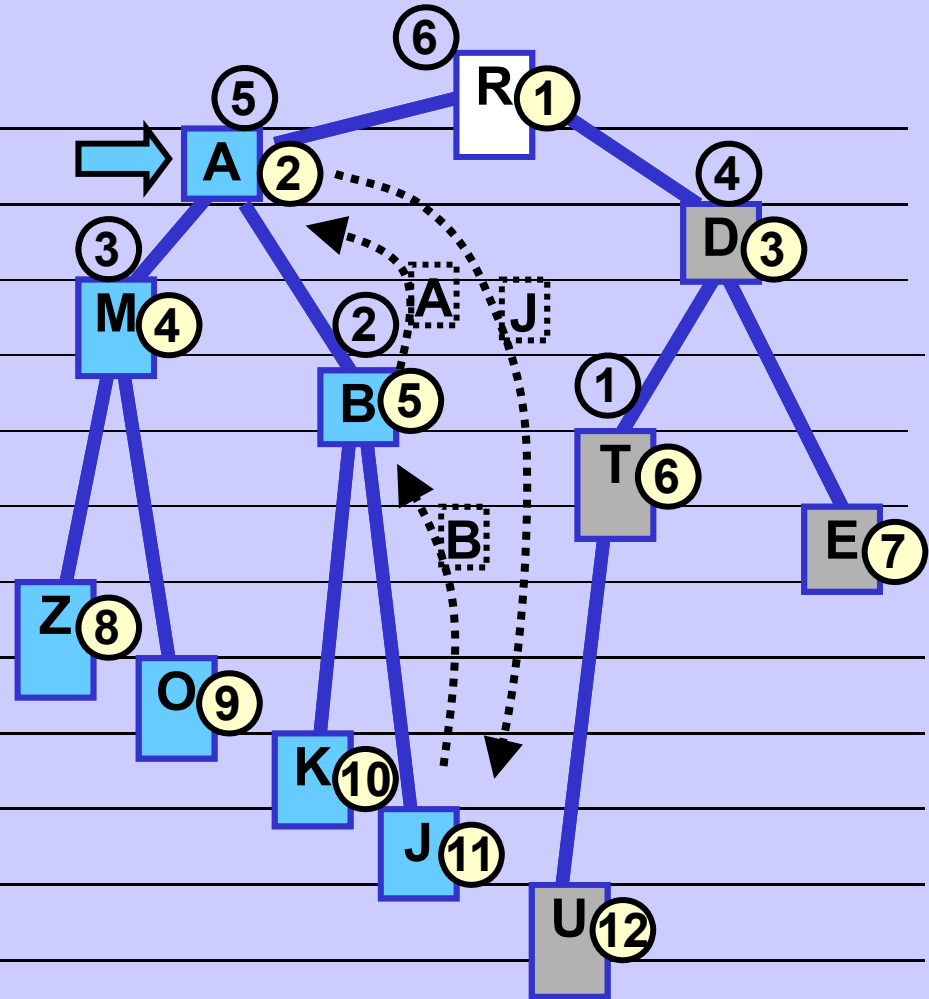


Heap in an array

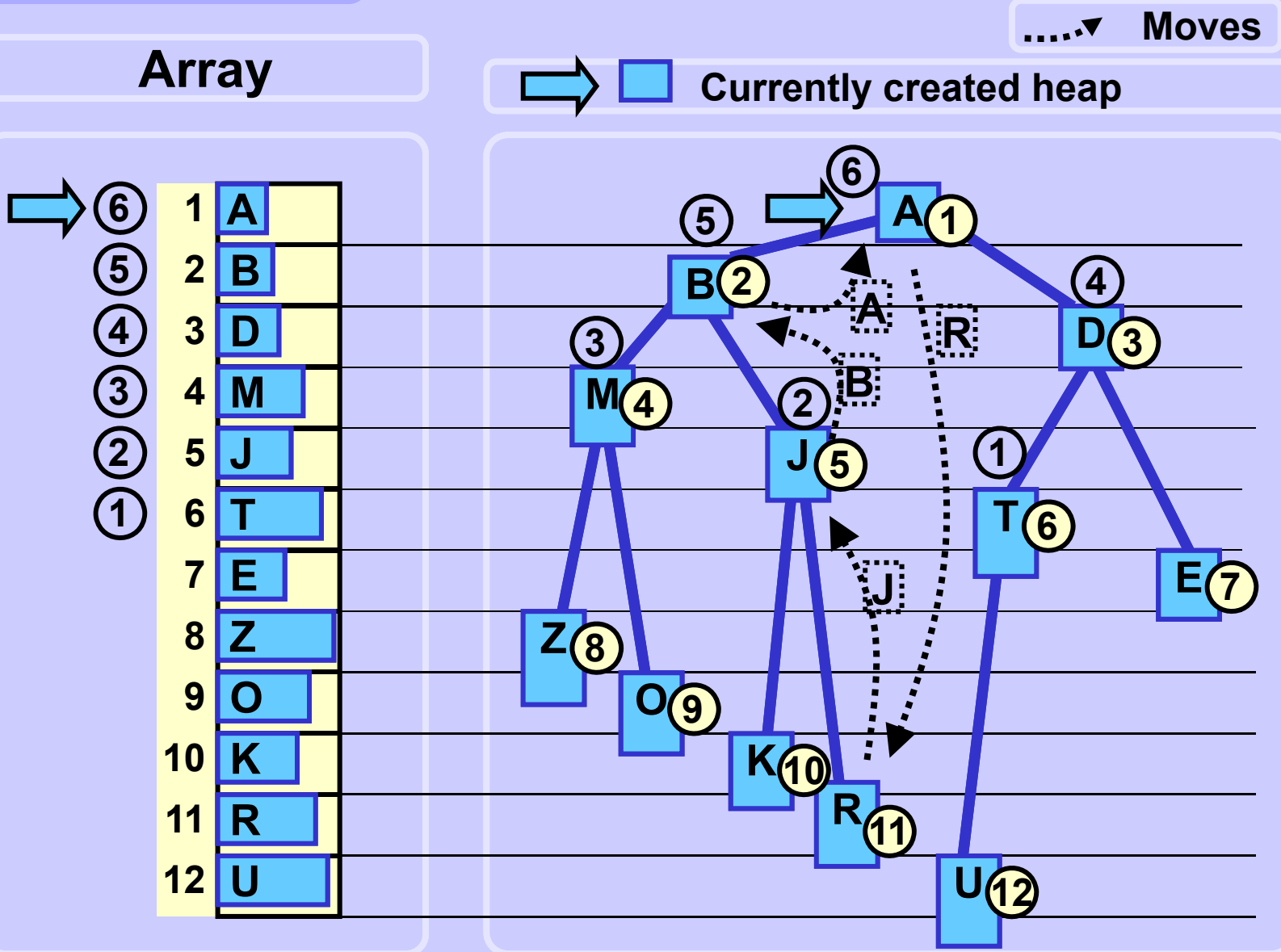
Array

⑥	1	R
⑤	2	A
④	3	D
③	4	M
②	5	B
①	6	T
	7	E
	8	Z
	9	O
	10	K
	11	J
	12	U

Earlier heap(s)
 Currently created heap
 Moves

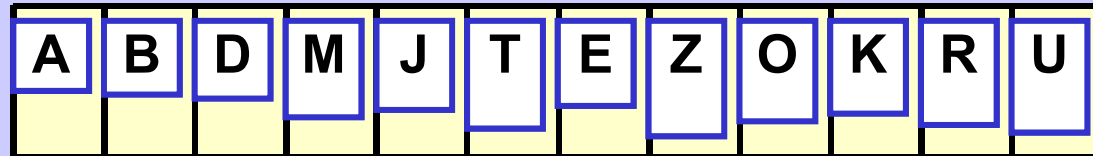


Heap in an array

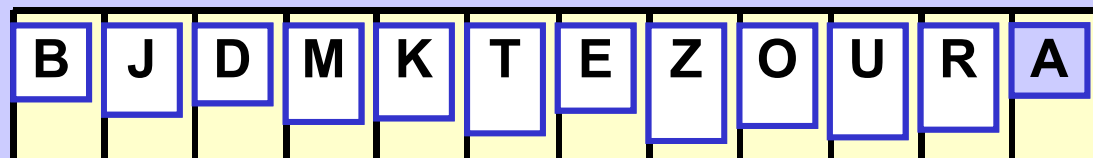
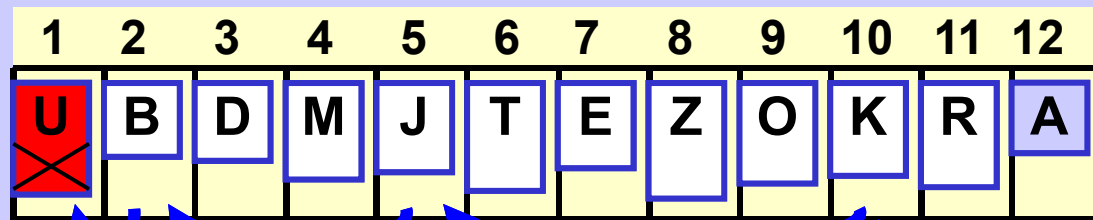
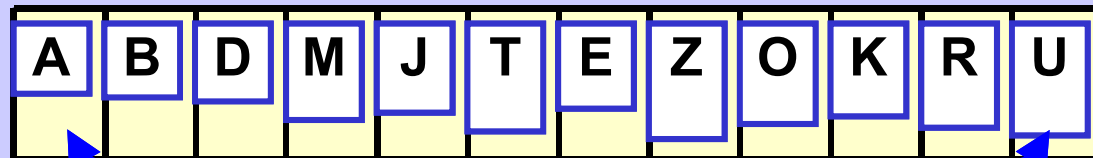


Heap sort

Heap



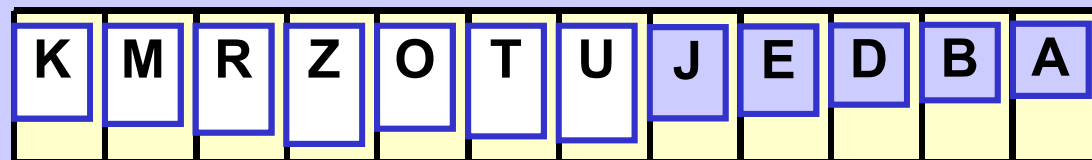
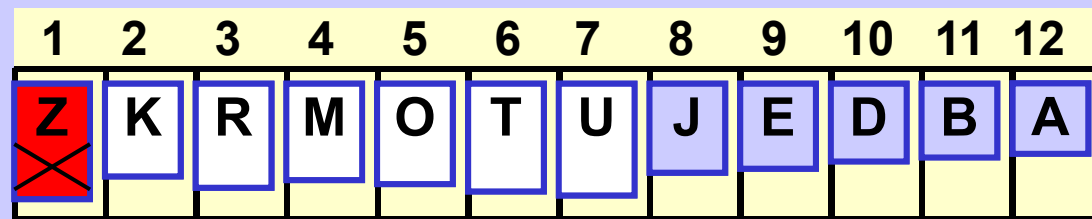
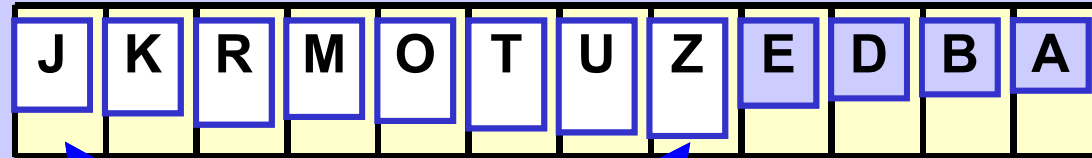
Step1 1



Heap

Heap sort

Step k



Heap

k

Heap sort

```
# beware! array is arr[1] ... arr[n]

def heapSort (arr):
    n = len(arr)-1

    # create a heap
    for i in range(n//2, -1, -1): # progress backwards!
        repairTop(arr, i, n)

    for i in range(n, 1, -1):      # progress backwards!
        swap(arr, 1, i)
        repairTop(arr, 1, i-1)
```

Heap sort

```
def repairTop (arr, top, bottom):  
    i = top      # arr[2*i] and arr[2*i+1]  
    j = i*2     # are successors of arr[i]  
    topVal = arr[top]  
  
    # try to find a successor < topVal  
    if j < bottom and arr[j] > arr[j+1]: j += 1  
  
    # while successors < topVal move successors up  
    while j <= bottom and topVal > arr[j]:  
        arr[i] = arr[j]  
        i = j; j = j*2      # move to next successor  
        if j < bottom and arr[j] > arr[j+1]: j += 1  
  
    # put topVal to its correct place  
    arr[i] = topVal
```

Heap sort

repairTop operation worst case ... $\log_2(n)$ ($n = \text{heap size}$)

make a heap ... $n/2$ repairTop calls

$$\log_2(n/2) + \log_2(n/2+1) + \dots + \log_2(n) \leq (n/2)(\log_2(n)) = \underline{\underline{O(n \cdot \log_2(n))}}$$

sort the heaps ... $n-1$ repairTop calls, worst case:

$$\log_2(n) + \log_2(n-1) + \dots + 1 \leq n \cdot \log_2(n) = O(n \cdot \log_2(n))$$

surprisingly, also the best case = $\underline{\underline{\Theta(n \cdot \log_2(n))}}$

total ... make a heap + sort the heap = $\underline{\underline{\Theta(n \cdot \log_2(n))}}$

Asymptotic complexity of Heap sort is $\Theta(n \cdot \log_2(n))$.

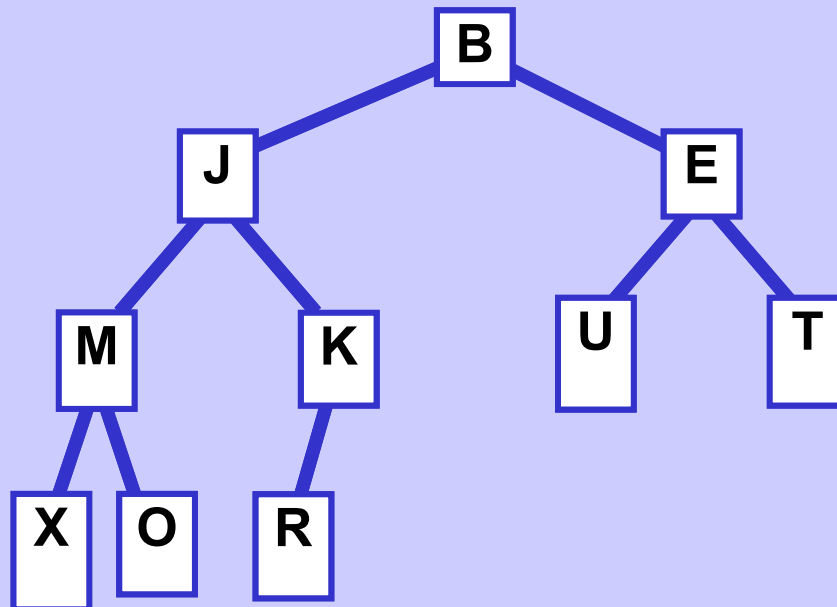
Heap sort is not stable.

Priority queue

Operations

- Insert or Enqueue
- Front, Top, Peek -- read topmost element
- Dequeue, Pop, Poll -- delete topmost element.

The element with the smallest value (biggest value in max-heaps) of all elements in the heap is always at the top.



Priority queue might be implemented using a heap.

Officially:

"A *binary* heap".

Priority queue implemented with binary heap -- operations

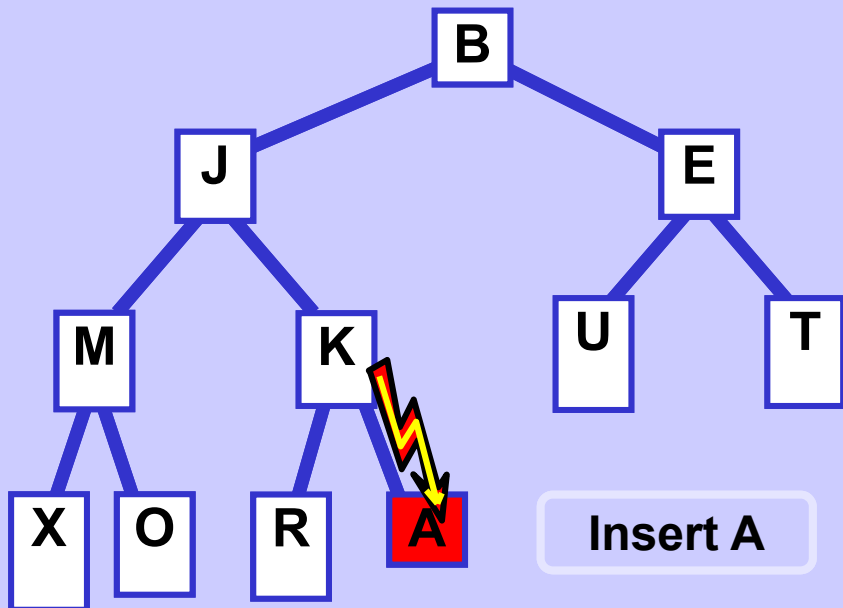
Read the topmost element (Front, Top, Peek, ...) .

Obvious.

Delete the topmost element (Dequeue, Pop, Poll, ...) =
Remove the top and repair the heap.

As before.

Insert an element to the queue (Insert, Enqueue, ...)



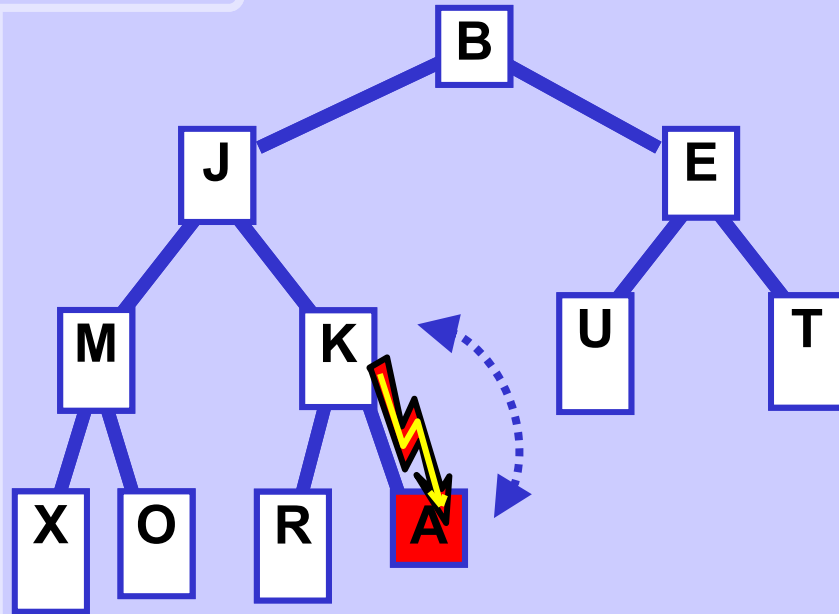
Insert the element
at the end of the queue
(end of the heap).

In most cases, this
violates the heap property

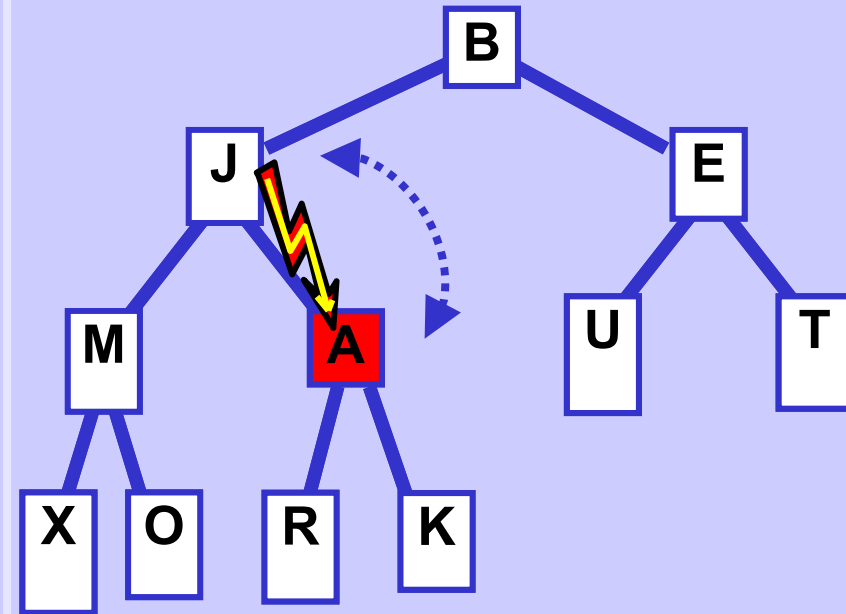
and the heap has to be
repaired.

Priority queue implemented with binary heap -- Insert

Insert A



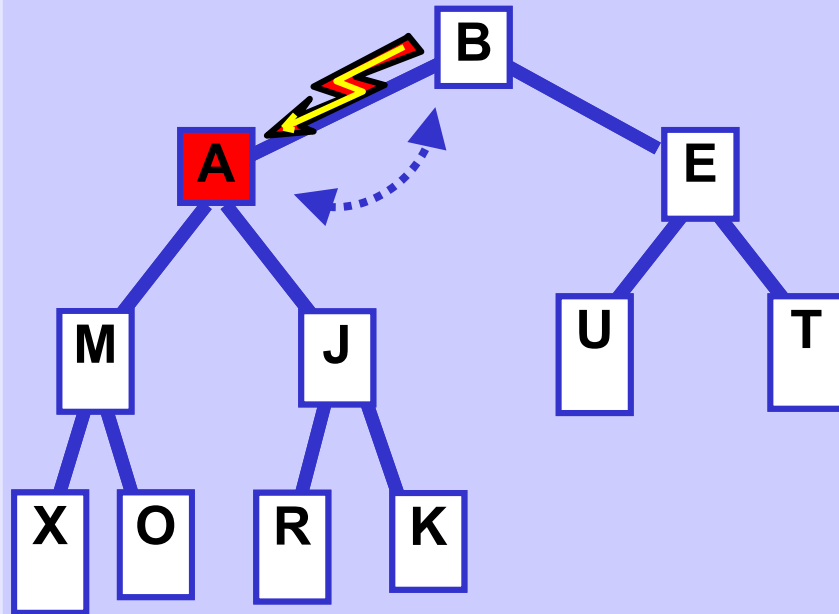
Heap property is violated,
swap the element with its parent.



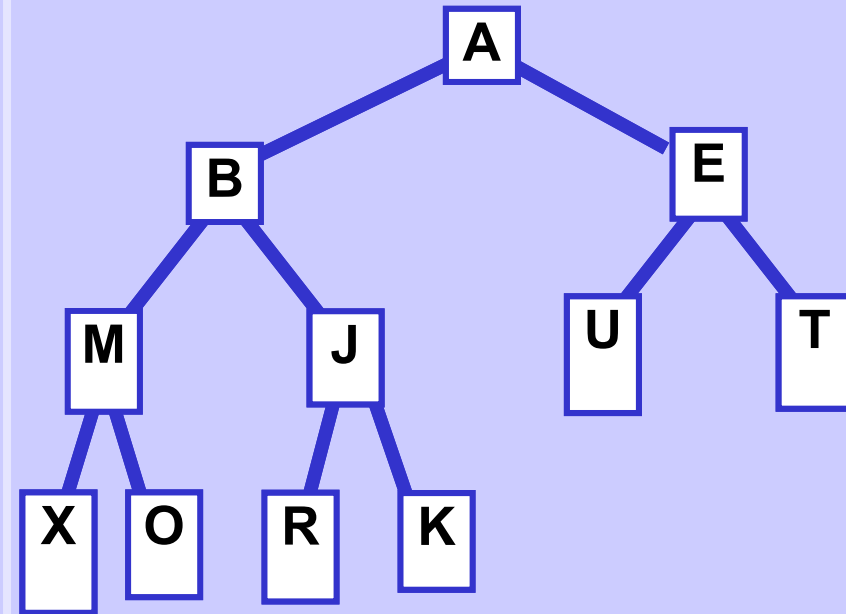
Heap property is still violated,
swap the element with its parent.

Priority queue implemented with binary heap -- Insert

Inserting A



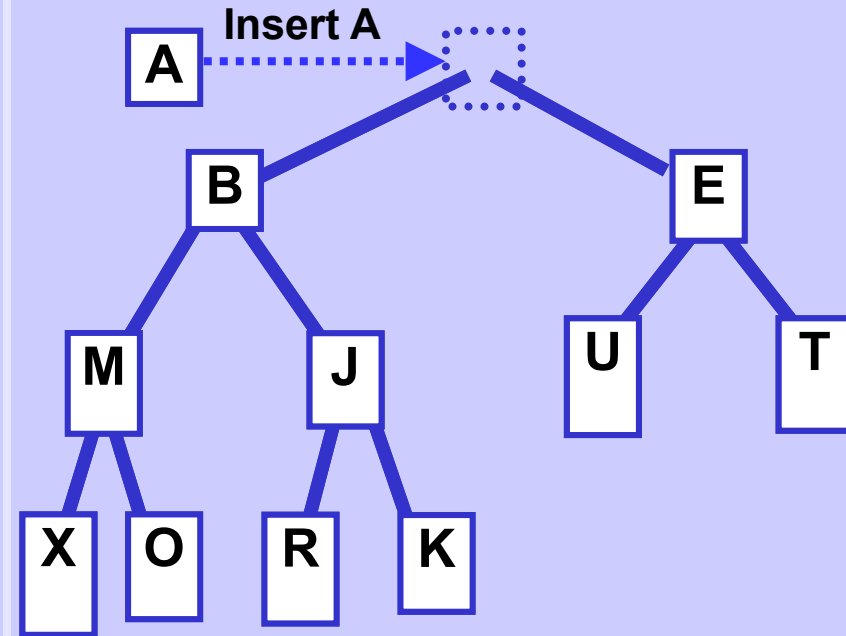
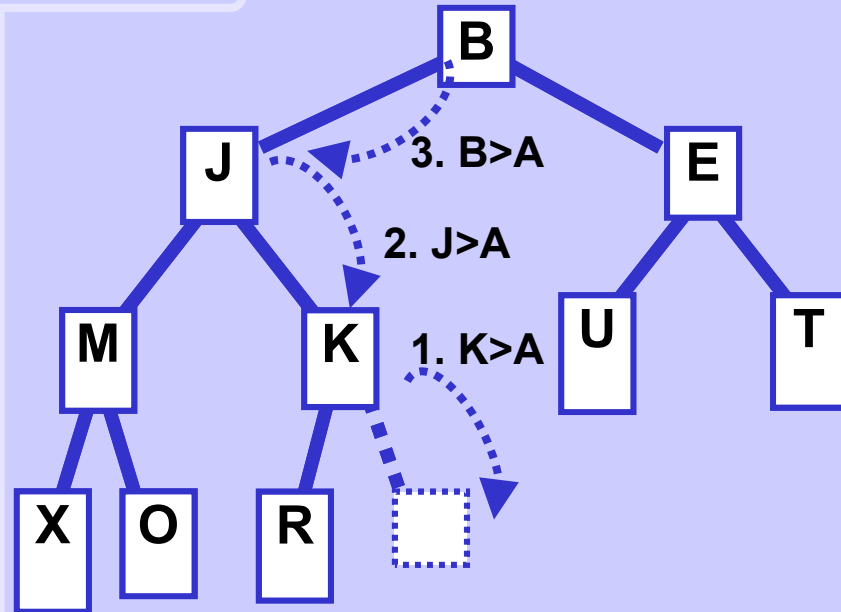
Heap property is still violated,
swap the element with its parent.



Heap property is respected,
the inserted element has found
its place in the queue (heap).

Binary heap -- Insert element more effectively

Insert A



Do not insert the element at the end of the queue.
First, find its place and while searching move down other elements encountered in the search.

Finally, store the inserted element at its correct position.

Binary heap – Insert

```

# beware!  array is arr[1] ... arr[n]
# bottom == ndx of last elem
def heapInsert(arr, x, bottom):
    bottom += 1          # expand the heap space
    j = bottom
    i = j/2              # parent index

    while i > 0 and arr[i] > x:
        arr[j] = arr[i]  # move elem down the heap
        j = i; i /= 2    # move indices up the heap

    arr[i] = x           # put inserted elem to its place
    return bottom

```

Insert -- Complexity

Inserting represents a traversal in a binary tree from a leaf to the root in the worst case. Therefore, the Insert complexity is $O(\log_2(n))$.