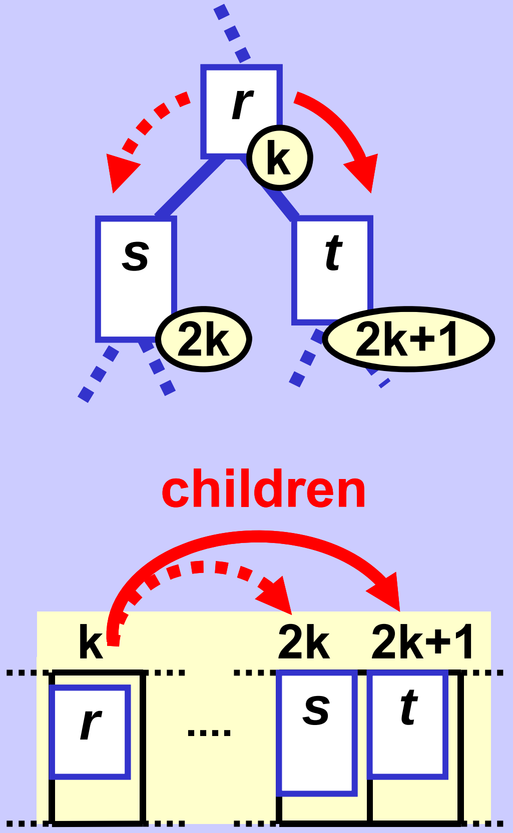
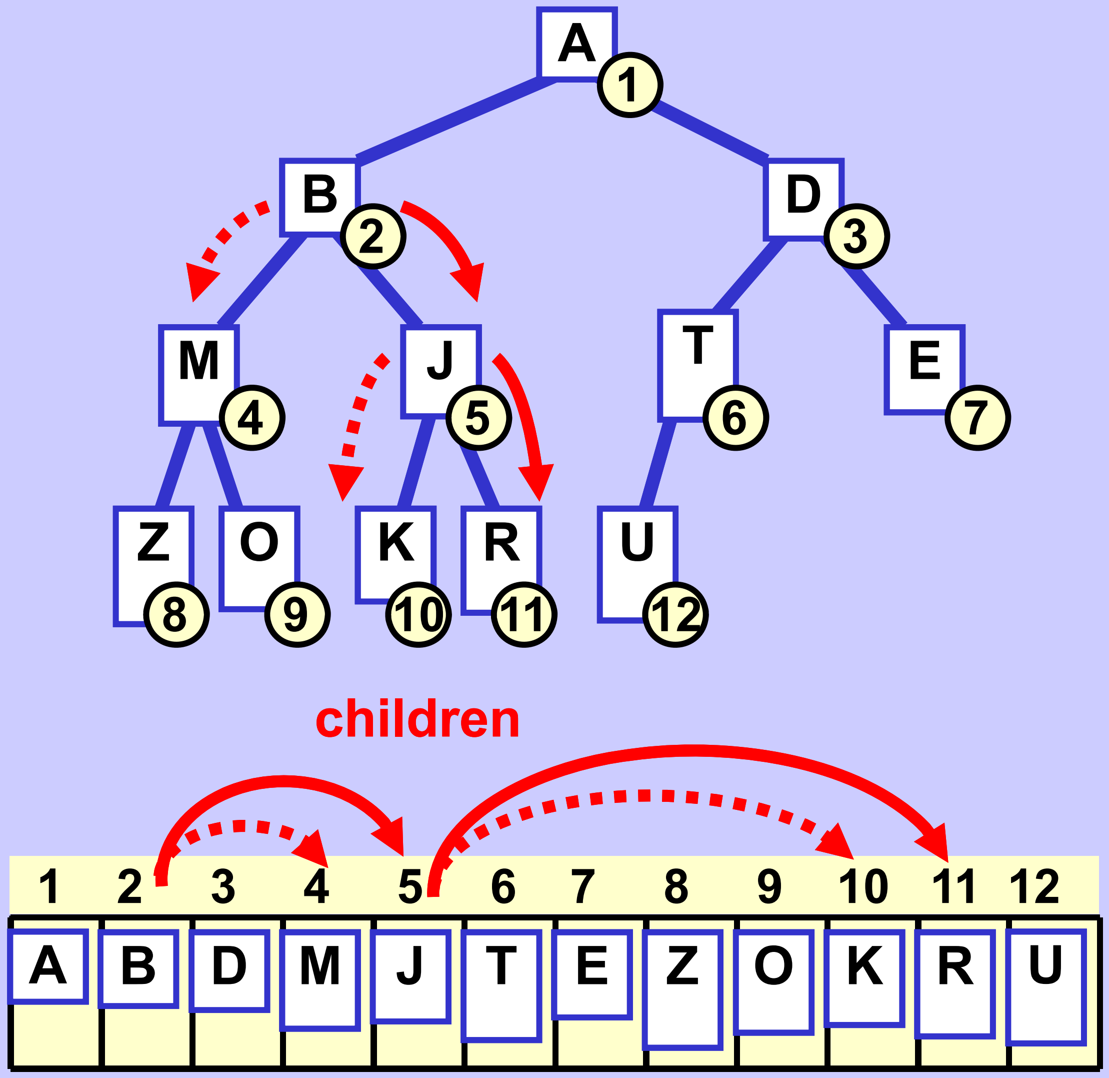
Heap in an array

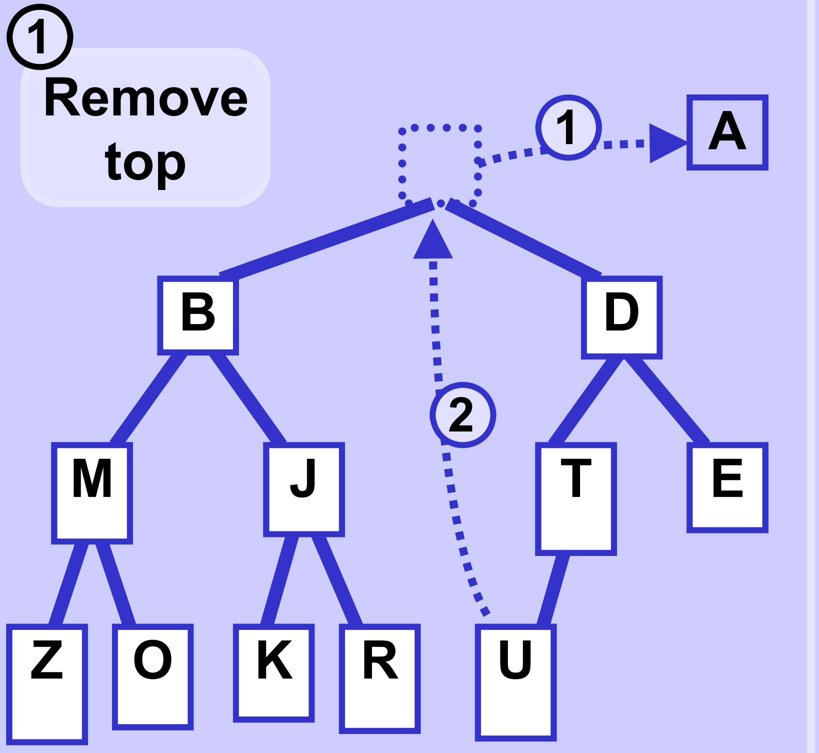
Heap stored in an array



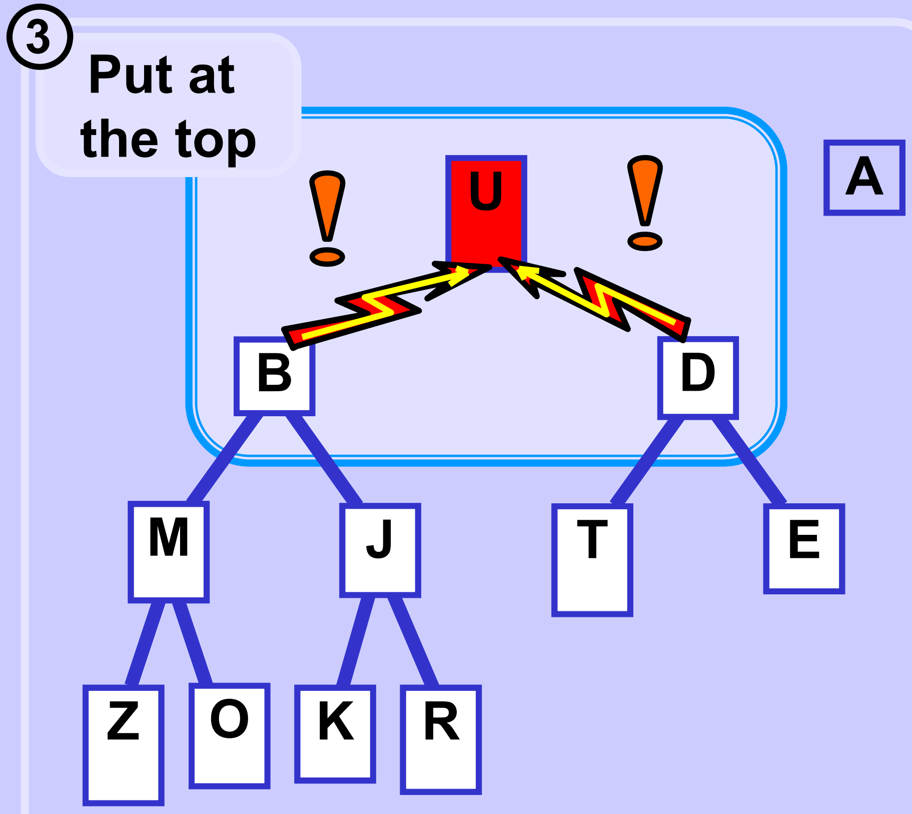


Heap repair

Top removed (1)



2 last $\longrightarrow$ first



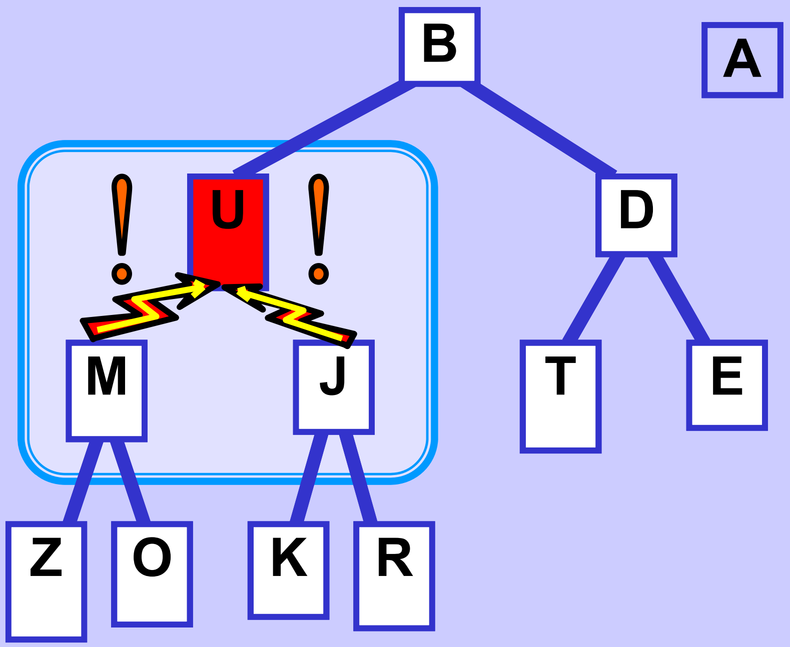
U > B, U > D, <u>B < D</u>

$\Rightarrow$ swap B $\longleftrightarrow$ U

Heap repair

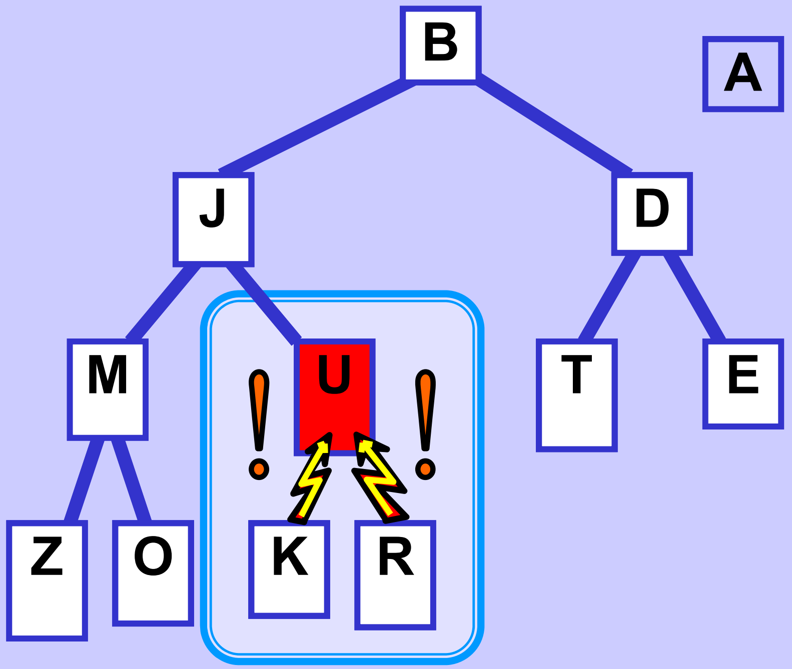
Top removed (2)

3 Put at the top - cont...



U > M, U > J, <u>J < M</u>

$\Rightarrow$ swap J $\longleftrightarrow$ U



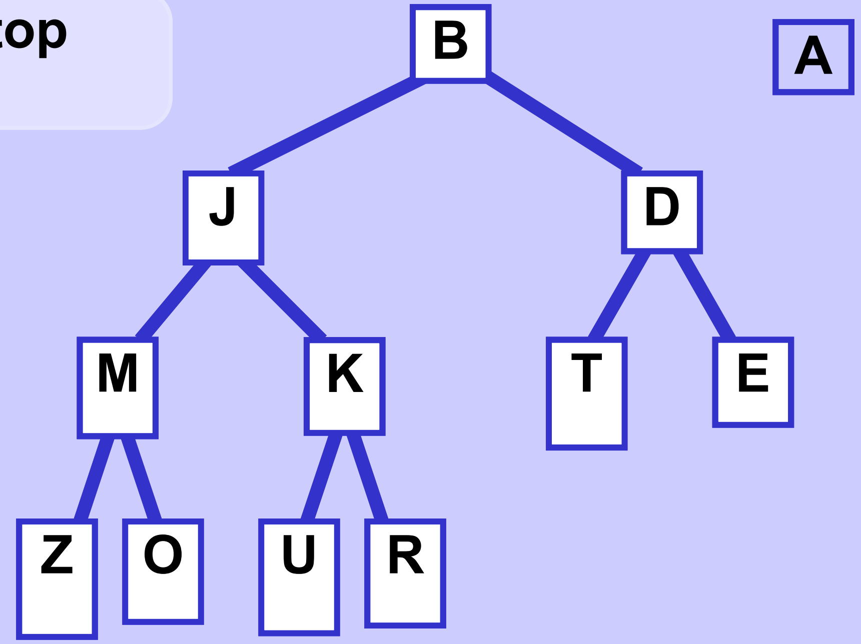
U > K, U > R, <u>K < R</u>

$\Rightarrow$ swap K $\longleftrightarrow$ U

Heap repair

Top removed (3)

③ Put at the top - done.



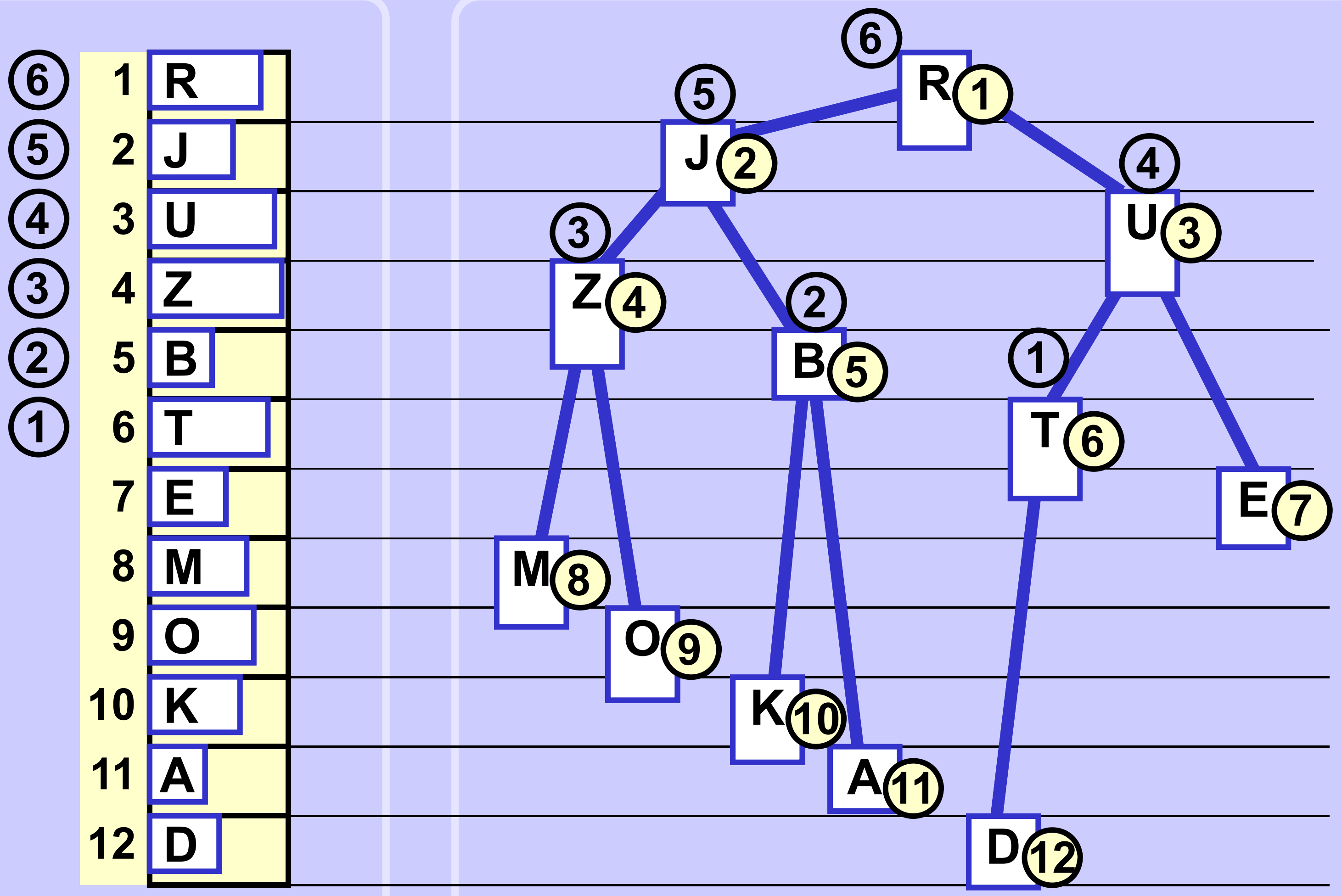
New heap

Make a heap -- Heapify

Array elements ordered randomly

Not a heap

Array

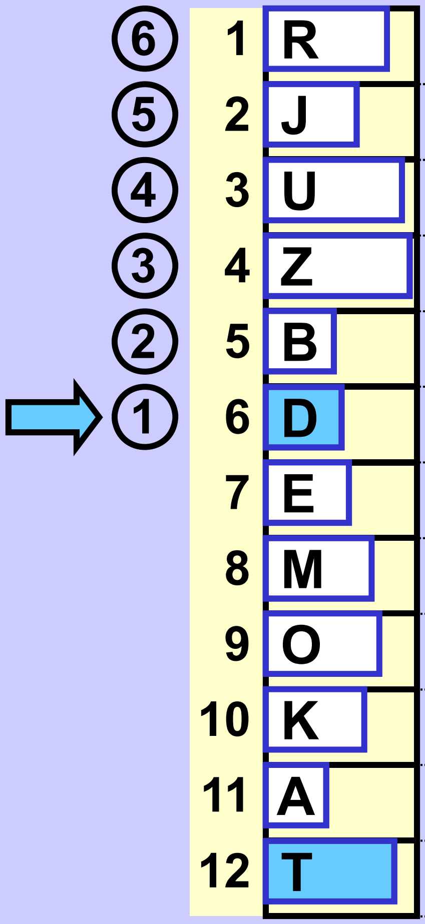


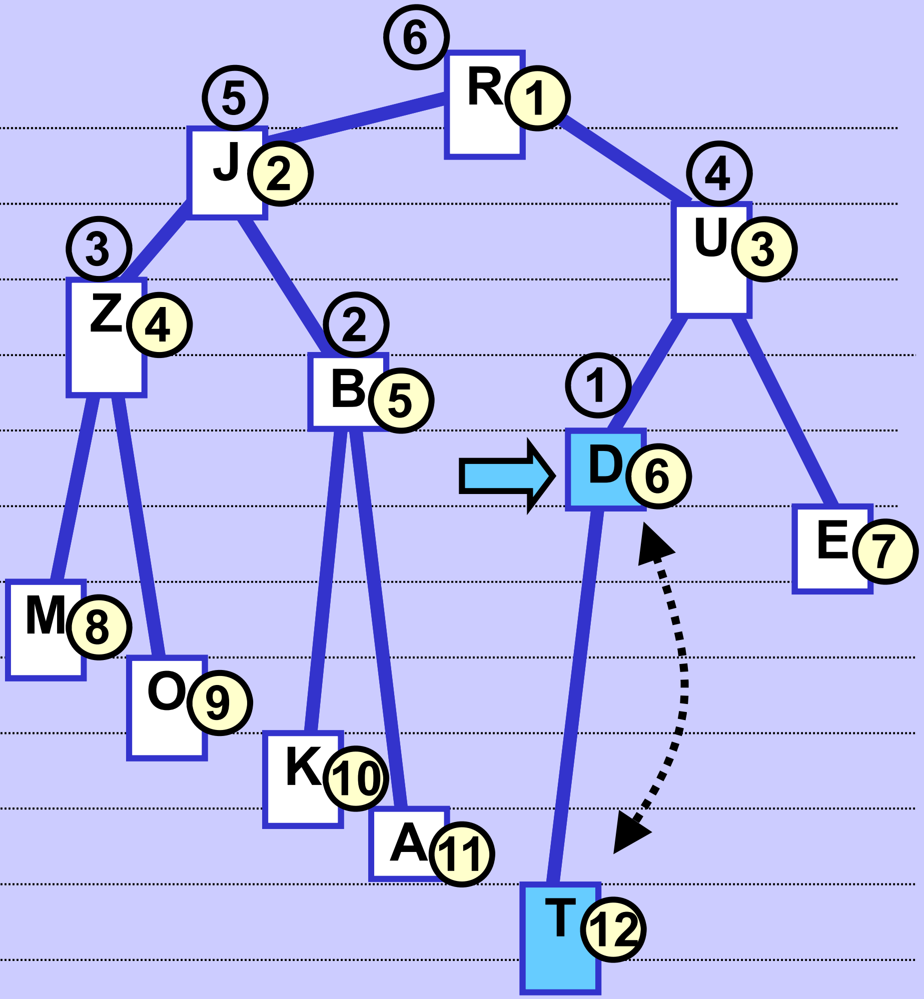
Make a heap -- Heapify

Moves

Currently created heap

Array





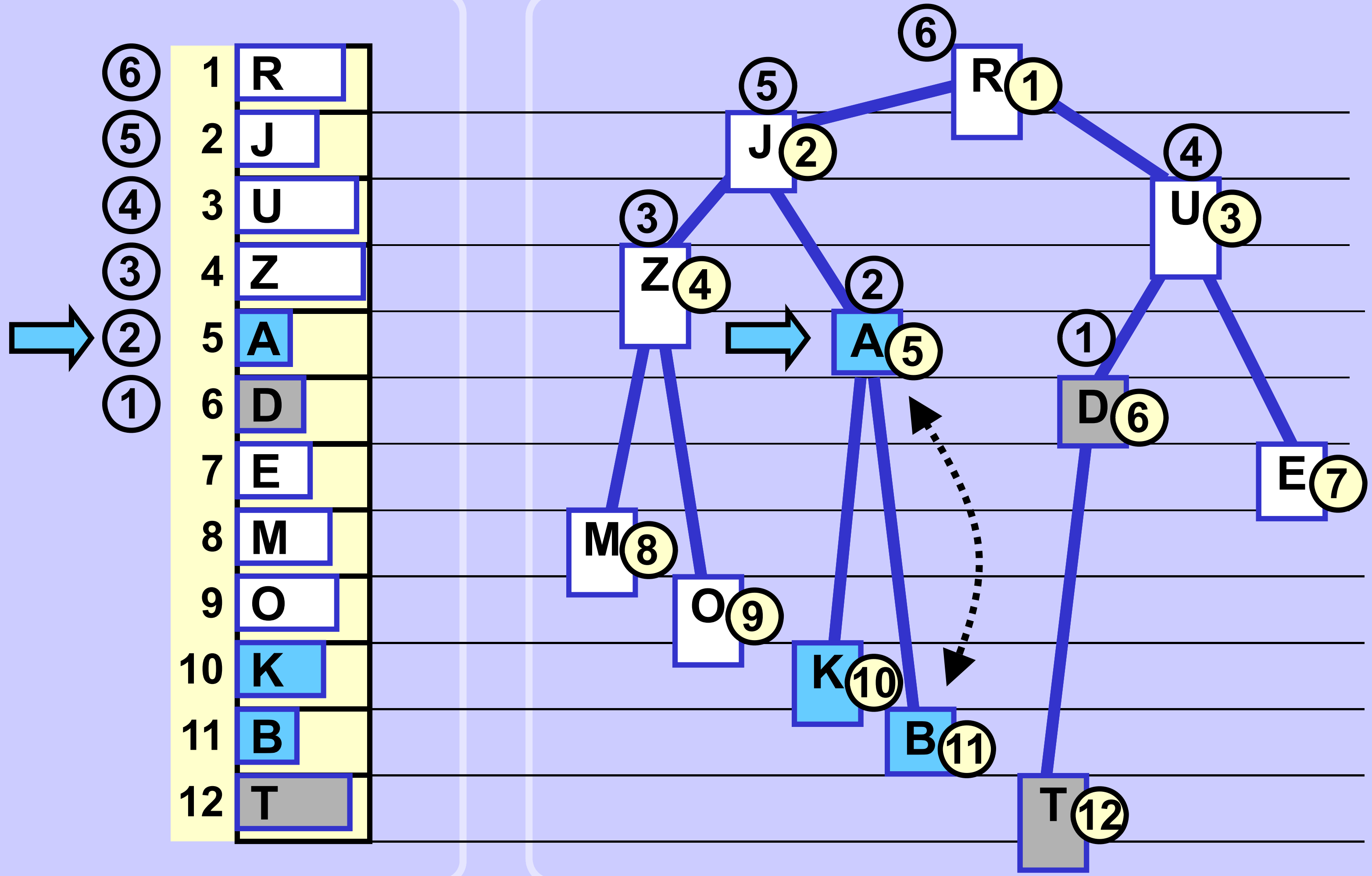
Make a heap -- Heapify

▨ Earlier heap(s)

Moves

🟦 Currently created heap

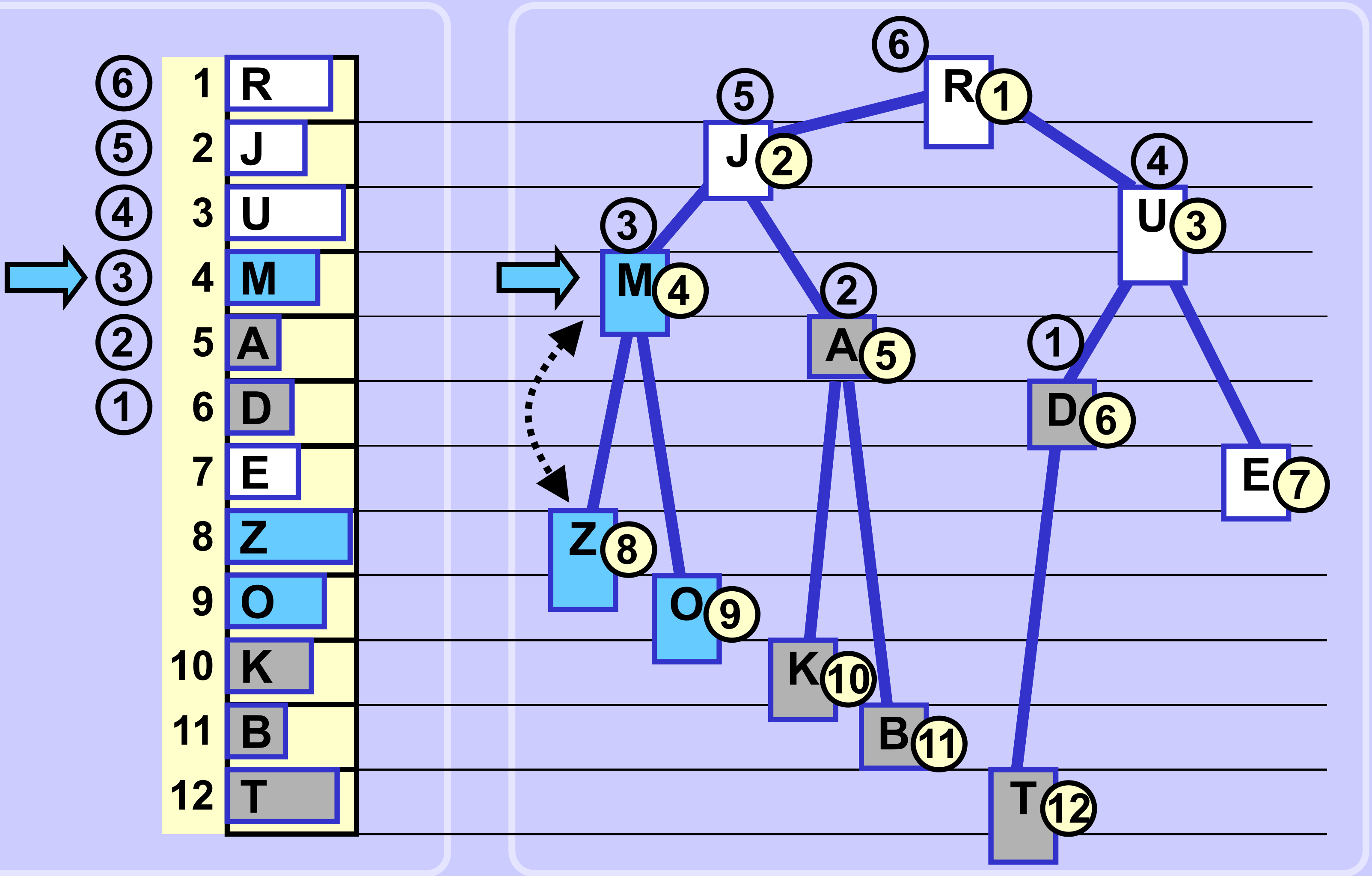
Array



Make a heap -- Heapify

▨ Earlier heap(s) ....➘ Moves

➡ 🟦 Currently created heap



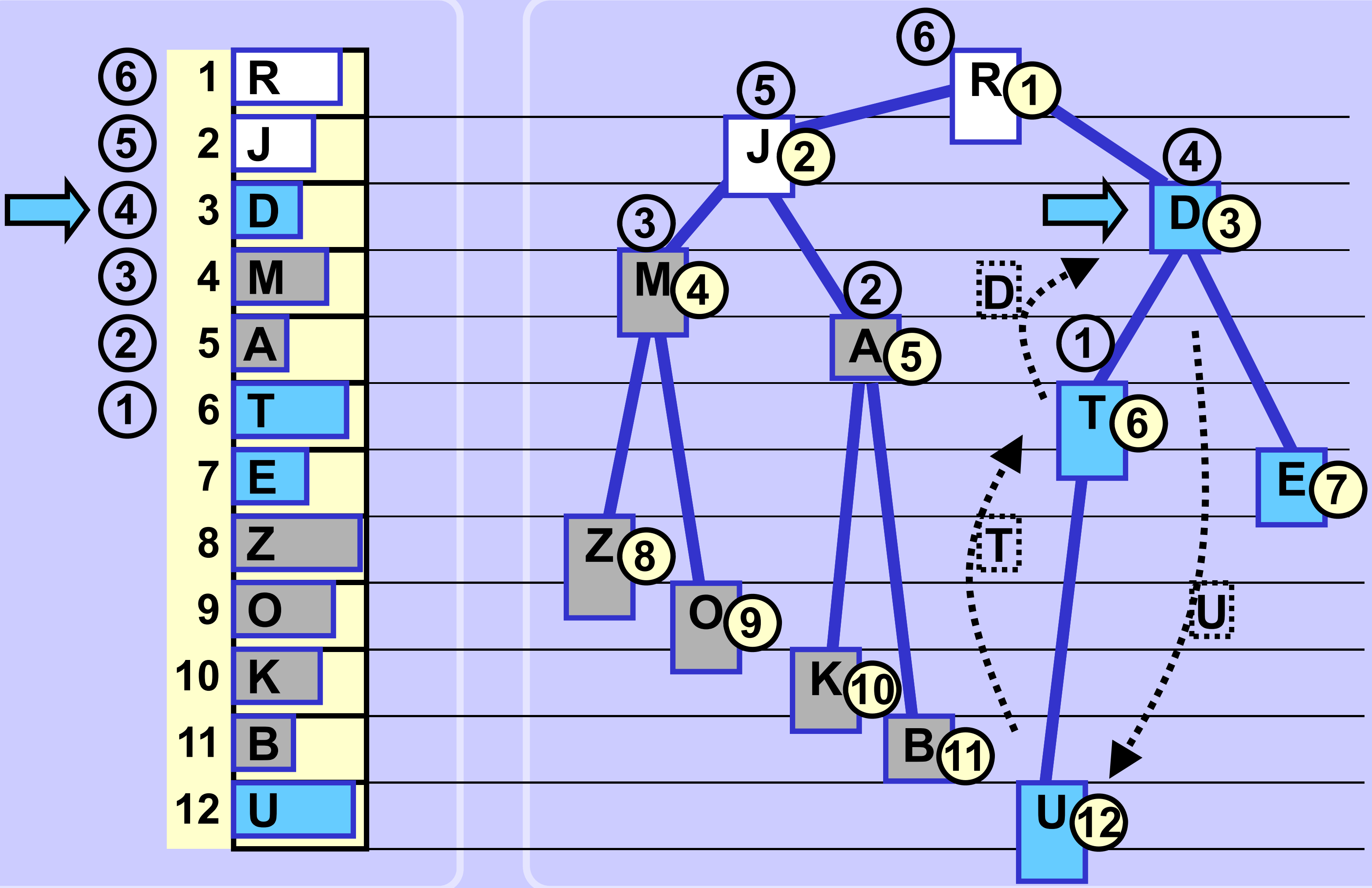
Make a heap -- Heapify

▨ Earlier heap(s)

Moves

🟦 Currently created heap

Array



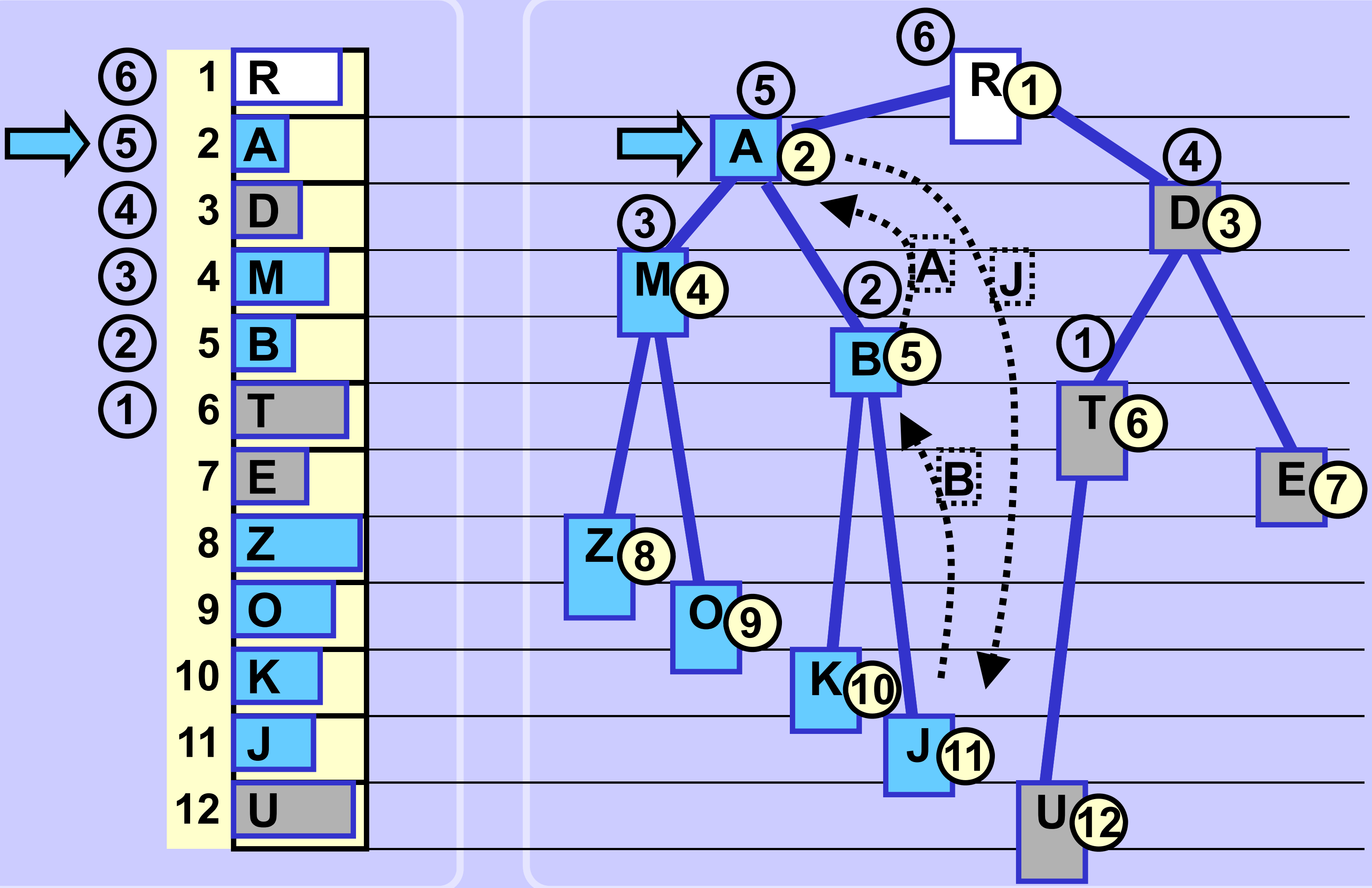
Make a heap -- Heapify

▨ Earlier heap(s)

....▸ Moves

🟦 Currently created heap

Array

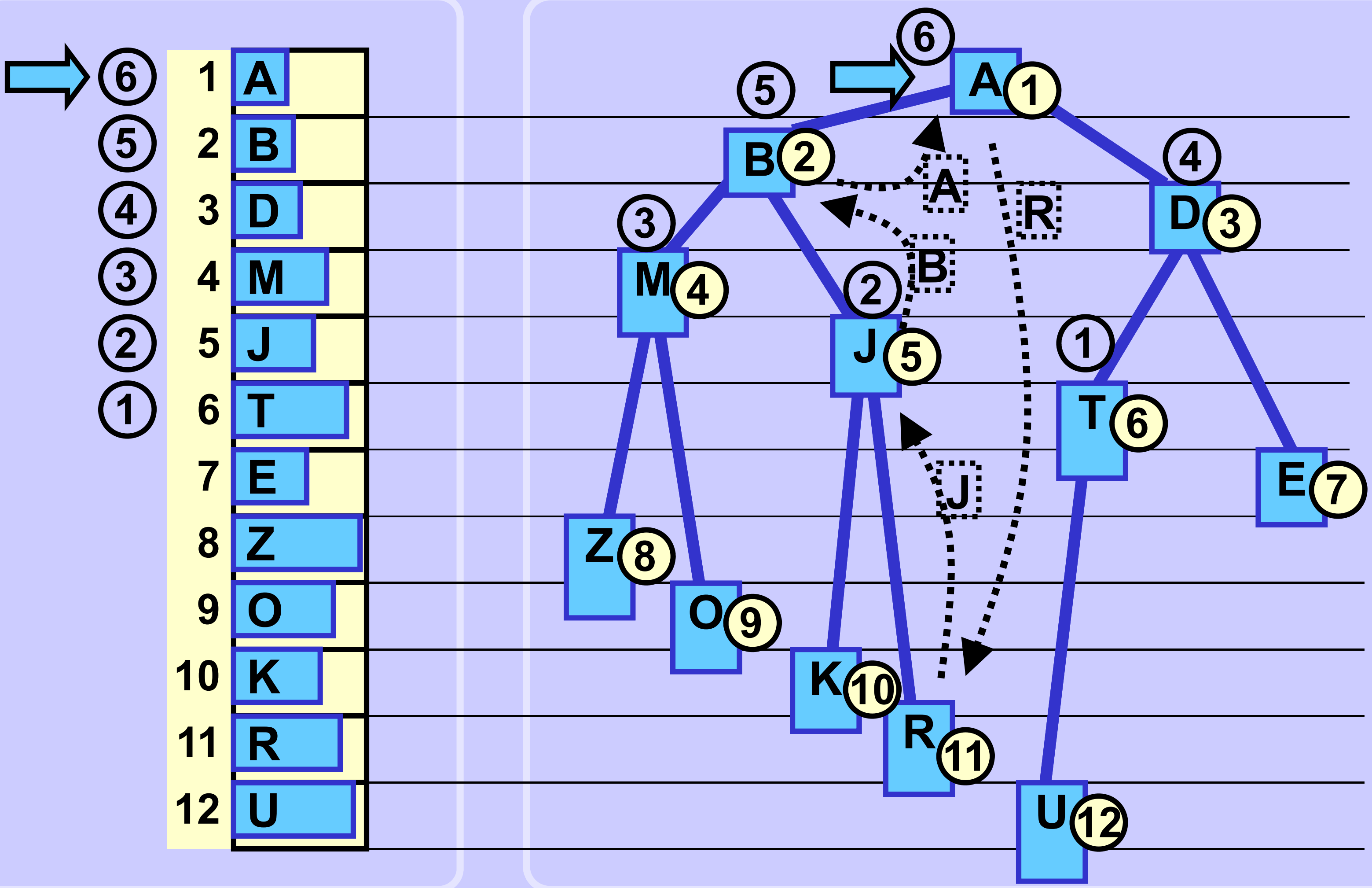


Make a heap -- Heapify

Moves

Currently created heap

Array

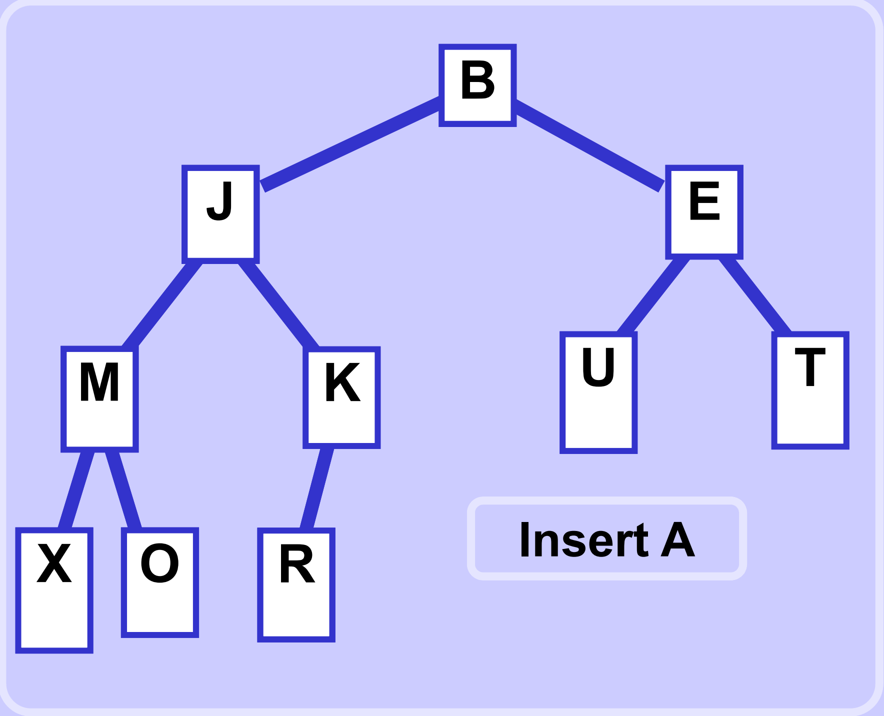


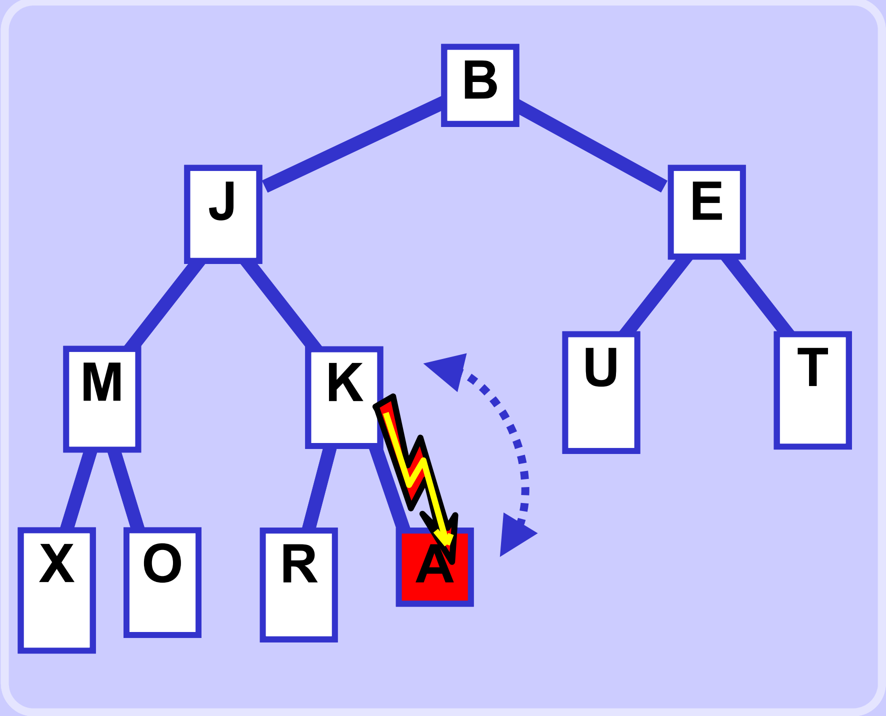
Heapify

```
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 suceessor
        if j < bottom and arr[j] > arr[j+1]: j += 1
    # put topVal to its correct place
    arr[i] = topVal
```

```
def heapify (arr):
    n = len(arr)-1
    for i in range(n//2, 0, -1): # progress backwards!
        repairTop(arr, i, n)
```

Priority queue implemented with binary heap -- Insert



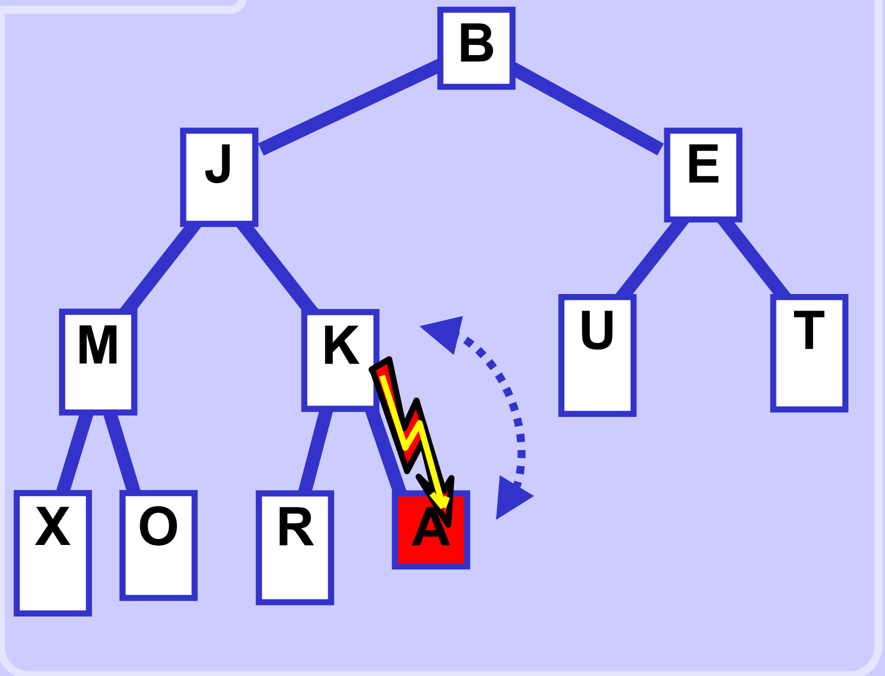


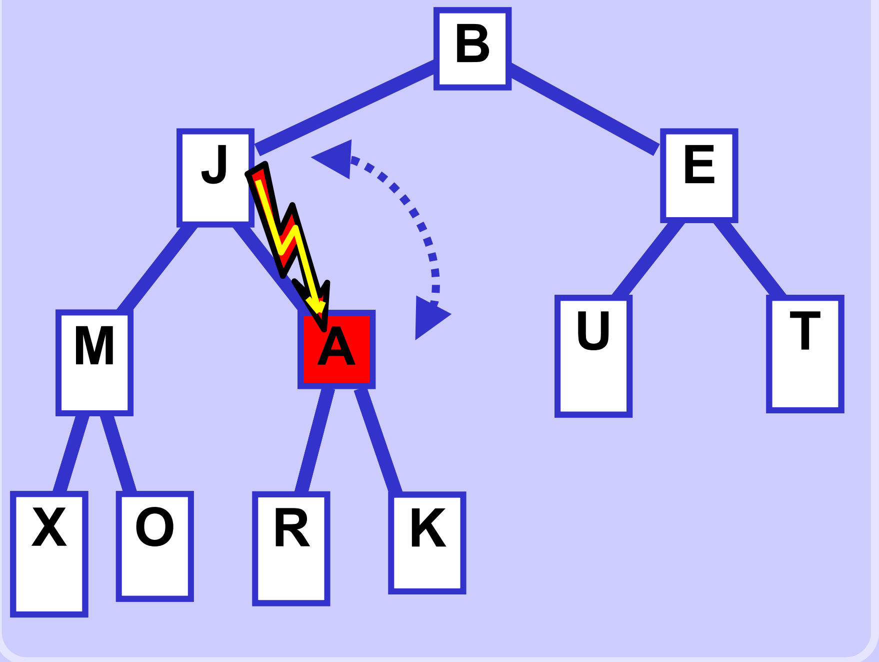
Insert the element at the end of the queue (end of the heap).

In most cases, this violates the heap property
and the heap has to be repaired.

Priority queue implemented with binary heap -- Insert

Insert A



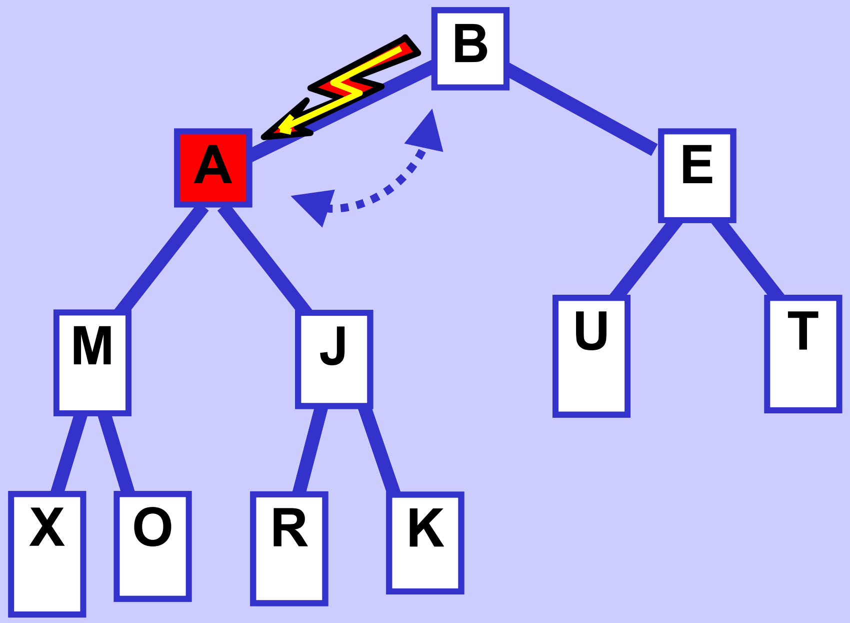


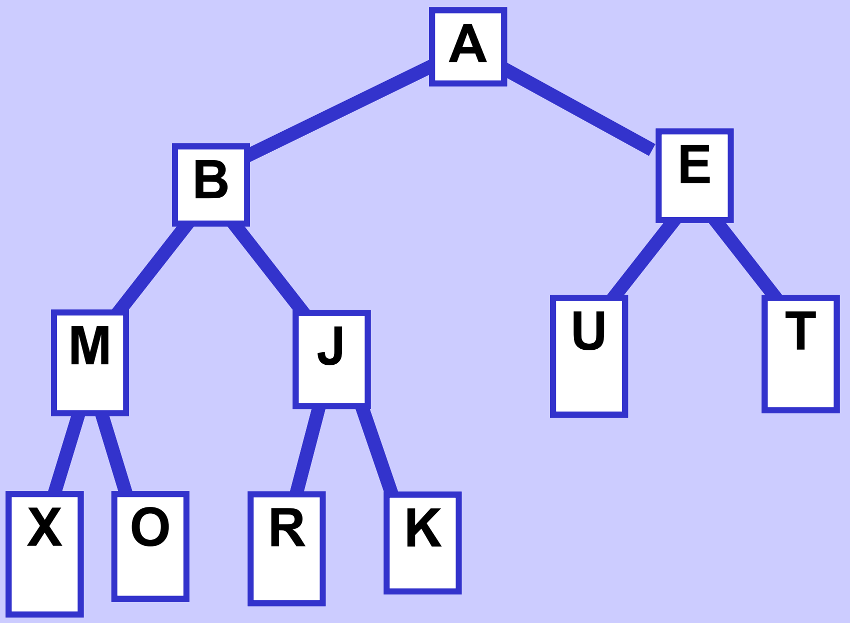
Heap property is violated,
swap the element with its parent.

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

Priority queue implemented with binary heap -- Insert

Inserting A

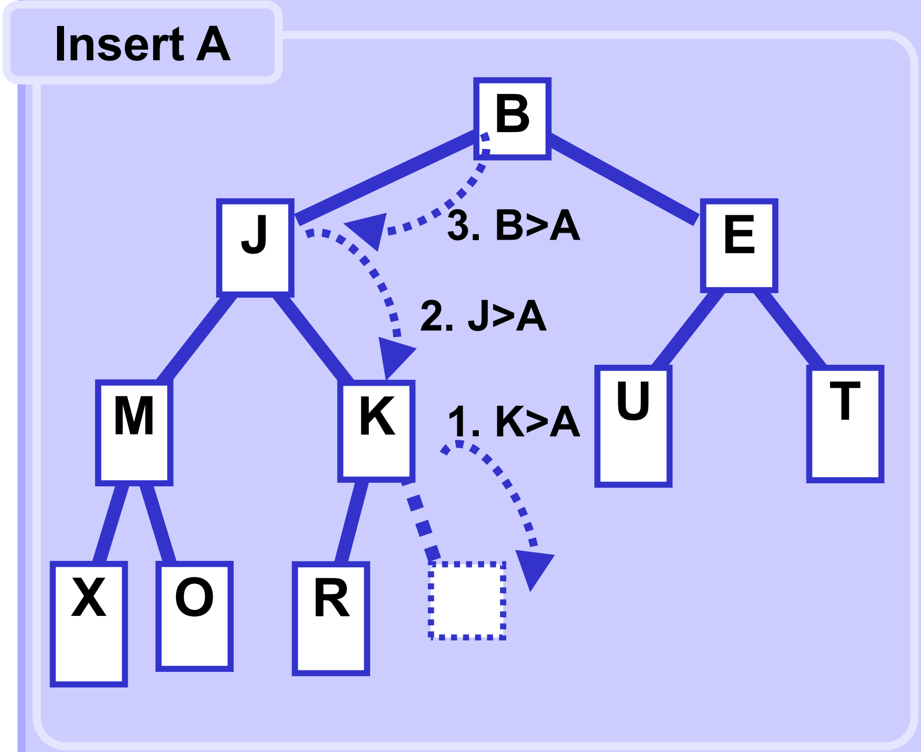


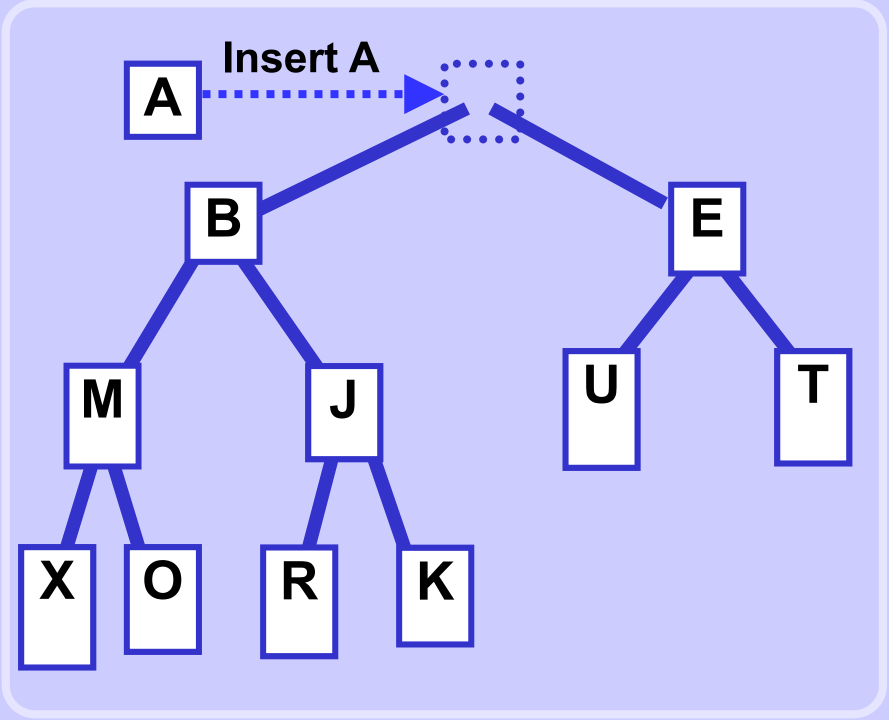


Heap property is still violated, swap the element with its parent.

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

Binary heap -- Insert element more effectively





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 $O(\log_2(n))$.