Representation of Sorted List

- input list (26, 5, 7, 1, 61, 11, 59, 15, 48, 19)

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	26	5	77	1	61	11	59	15	48	19
link	9	6	0	2	3	8	5	10	7	1

(a) Linked list following a list sort, *first* = 4

Figure 7.10: Sorted linked lists

Recursive Merge Sort: ListMerge

```
template <class T>
int ListMerge(T* a, int* link, const int start1, const int start2)
{ // The sorted chains beginning at start1 and start2, respectively, are merged.
  // link [0] is used as a temporary header. Return start of merged chain.
     int iResult = 0; // last record of result chain
     for (int i1 = start1, i2 = start2; i1 && i2; )
          if (a[i1] <= a[i2]) {
               link[iResult] = i1;
               iResult = i1; i1 = link[i1];
          }
          else {
               link[iResult] = i2;
               iResult = i2; i2 = link[i2];
          }

     // attach remaining records to result chain
     if(i1 == 0) link[iResult] = i2;
     else link[iResult] = i1;
     return link[0];
}
```

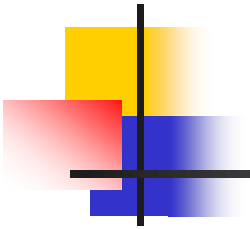
Program 7.11: Merging sorted chains

Recursive Merge Sort

- Divide the list into two sublists(left, right)
- Physical storage space is not changed.

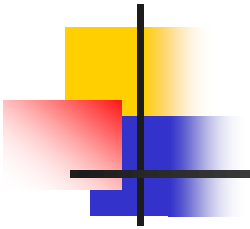
```
template <class T>
int rMergeSort(T *a, int* link, const int left, const int right)
{ // a[left:right] is to be sorted. link[i] is initially 0 for all i.
  // rMergeSort returns the index of the first element in the sorted chain.


    if (left >= right) return left;
    int mid = (left + right) / 2;
    return ListMerge(a, link,
                     rMergeSort(a, link, left, mid),          // sort left half
                     rMergeSort(a, link, mid + 1, right) );   // sort right half
}
```

Program 7.10: Recursive Merge Sort



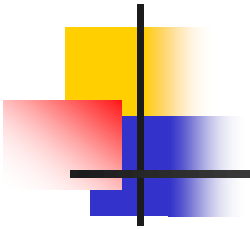
Heap Sort

- Save storage space
- Use the max heap structure in Chap. 5
- Store n element in the heap, and extract one at a time
- Adjust after the extract
- Store the extracted element in the last node



Heapsort

- Use BUILD-MAX-HEAP to build a max-heap.
- Since the maximum element of the array is stored at the root A[1], put it into its correct position by exchanging it with A[n].
- Restore the max-heap property by calling MAX-HEAPIFY(A, 1) to generate a max-heap in A[1..n-1].
- Repeat the steps for the max-heap of size n-1 down to a heap of size 2

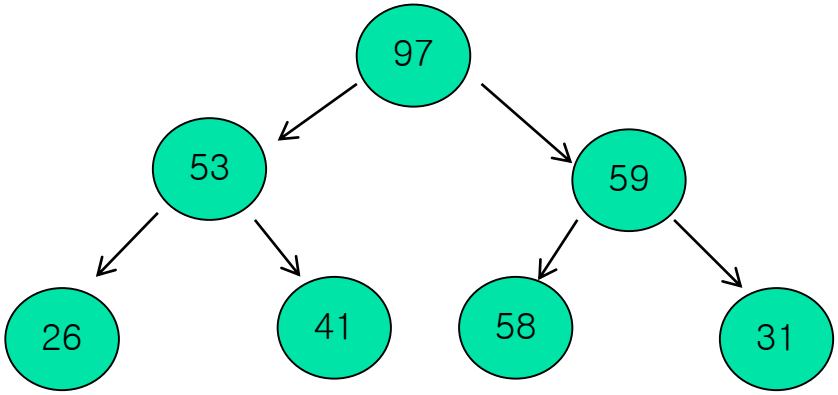


Heapsort(A)

HAEPSORT(A)

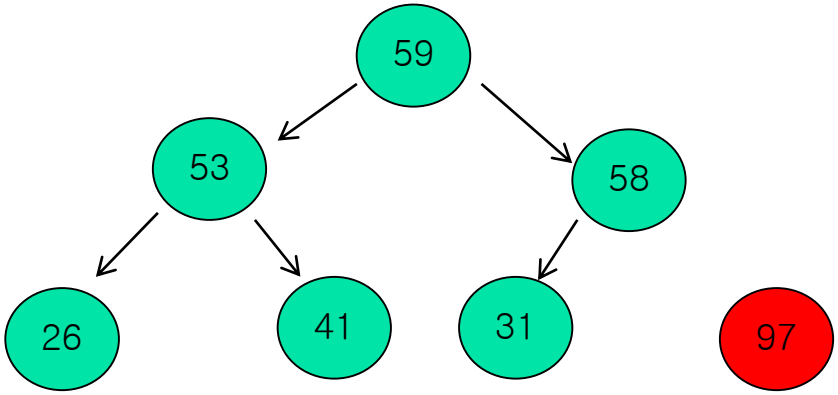
1. BUILD-MAX-HEAP(A)
2. **for** i = A.length **downto** 2
3. exchange A[1] with A[i]
4. A.heap-size = A.heap-size-1
5. MAX-HEAPIFY(A,1)

Operation of Heap Sort



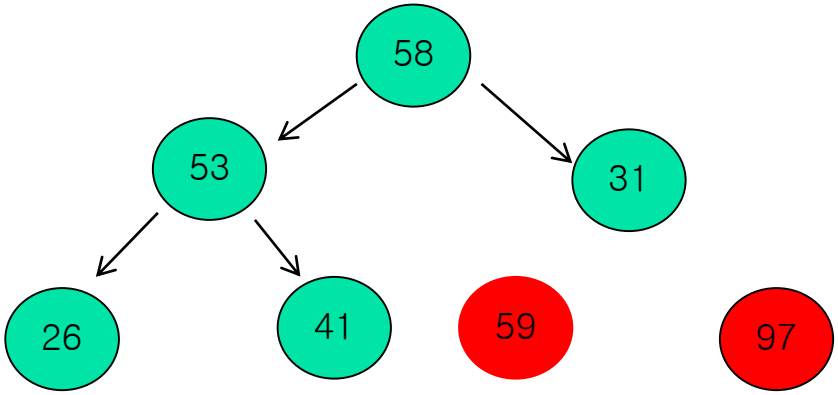
array		**97**	**53**	**59**	**26**	**41**	**58**	**31**				
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

Operation of Heap Sort



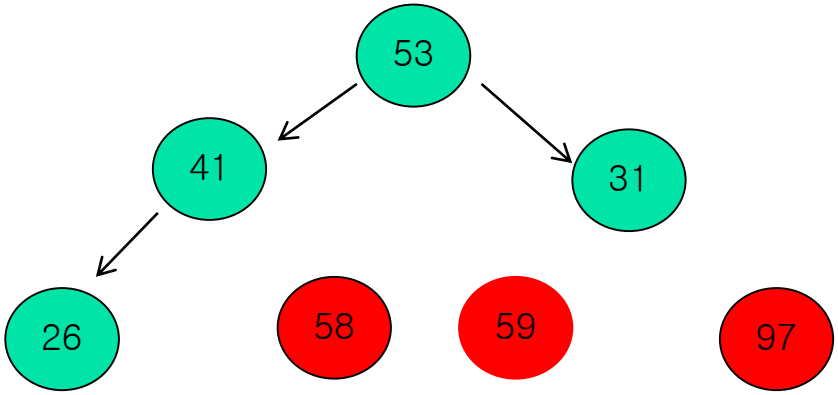
array		59	53	58	26	41	31	97				
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of Heap Sort



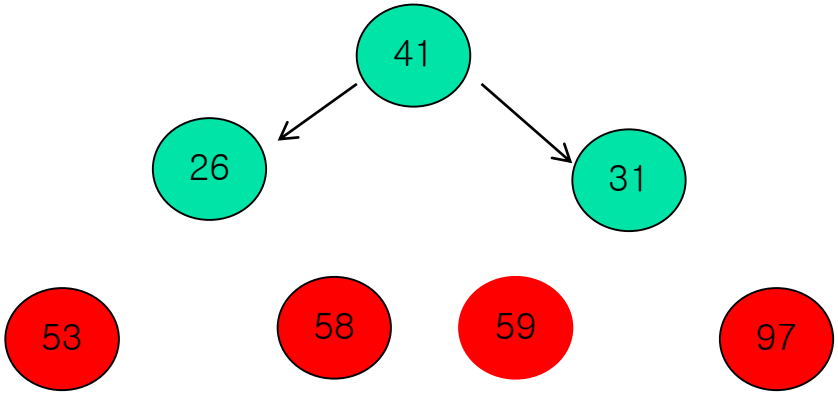
array		58	53	31	26	41	59	97				
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of Heap Sort



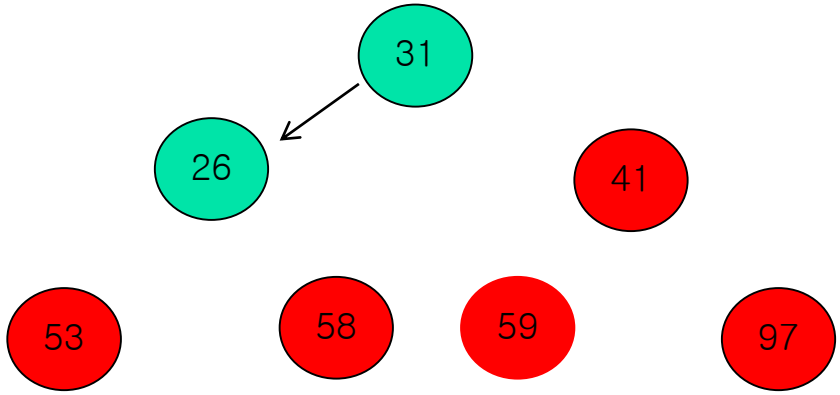
array		53	41	31	26	58	59	97				
index	0	1	2	3	4	5	6	7	8	9	10	11

Operation of Heap Sort



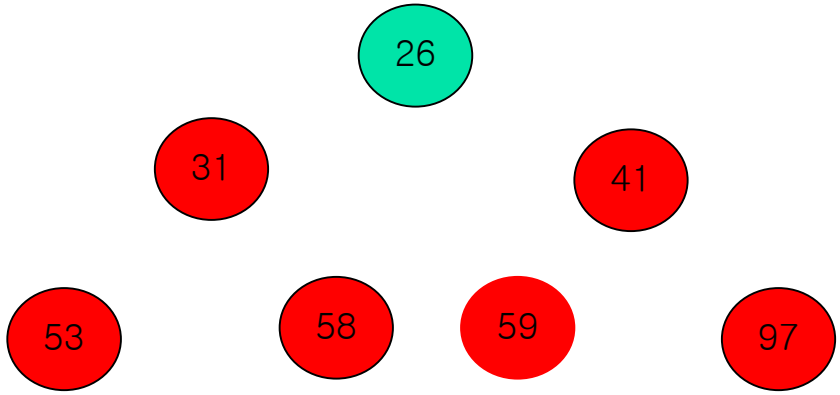
array		**41**	**26**	**31**	**53**	**58**	**59**	**97**				
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

Operation of Heap Sort

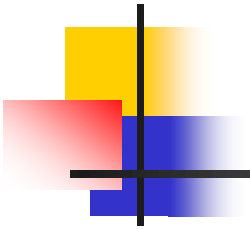


array		**31**	**26**	**41**	**53**	**58**	**59**	**97**				
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

Operation of Heap Sort



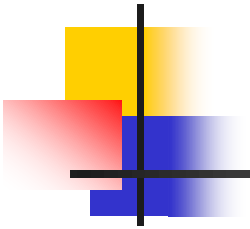
array		**26**	**31**	**41**	**53**	**58**	**59**	**97**				
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**



Heapsort(A)

HAEPSORT(A)

1. BUILD-MAX-HEAP(A)
2. **for** i = A.length **downto** 2
3. exchange A[1] with A[i]
4. A.heap-size = A.heap-size-1
5. MAX-HEAPIFY(A,1)



Heapsort

- It takes O(n lg n) time:
  - Calls to BUILD-MAX-HEAP takes O(n) time.
  - Each of the n-1 calls to MAX-HEAPIFY takes O(lg n) time.

Adjusting a Max Heap

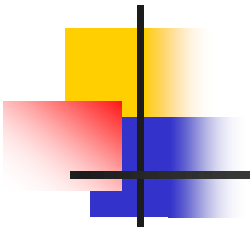
```
template <class T>
void Adjust(T *a, const int root, const int n)
{ // Adjust binary tree with root to satisfy heap property. The left and right
  // subtrees of root already satisfy the heap property. No node index is > n.

    T e = a[root];
    // find proper place for e
    for (int j = 2*root; j <= n; j *= 2) {
        if (j<n && a[j] < a[j+1]) j++; // j is max child of its parent
        if (e >= a[j]) break; // e may be inserted as parent of j
        a[j/2] = a[j]; // move j th record up the tree
    }
    a[j/2] = e;
}
```

Program 7.13: Adjusting a max heap

Heap Sort

```
template <class T>
void HeapSort(T *a, const int n)
{ // Sort a[1:n]) into nondecreasing order.
     for (int i=n/2; i>= 1; i--) // heapify
          Adjust(a, i, n);

     for(i=n-1; i>=1; i--) // sort
     {
          swap(a[1], a[i+1]);          // swap first and last of current heap
          Adjust(a,1,i);               // heapify
     }
}
```

Program 7.14: Heap Sort



Heap Sort

- Analysis : O(nlogn)
  - Adjust : $\log_2 n$
  - Call : n

Example

- Example 7.7
  - input list (26, 5, 77, 1, 61, 11, 59, 15, 48, 19)

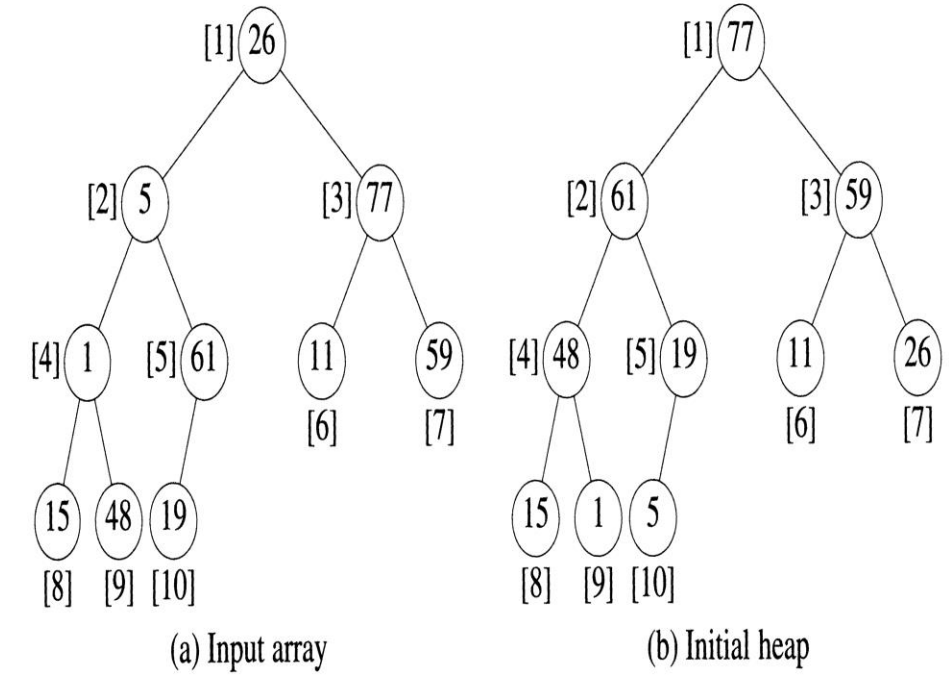


Figure 7.7: Array interpreted as a binary tree

Example

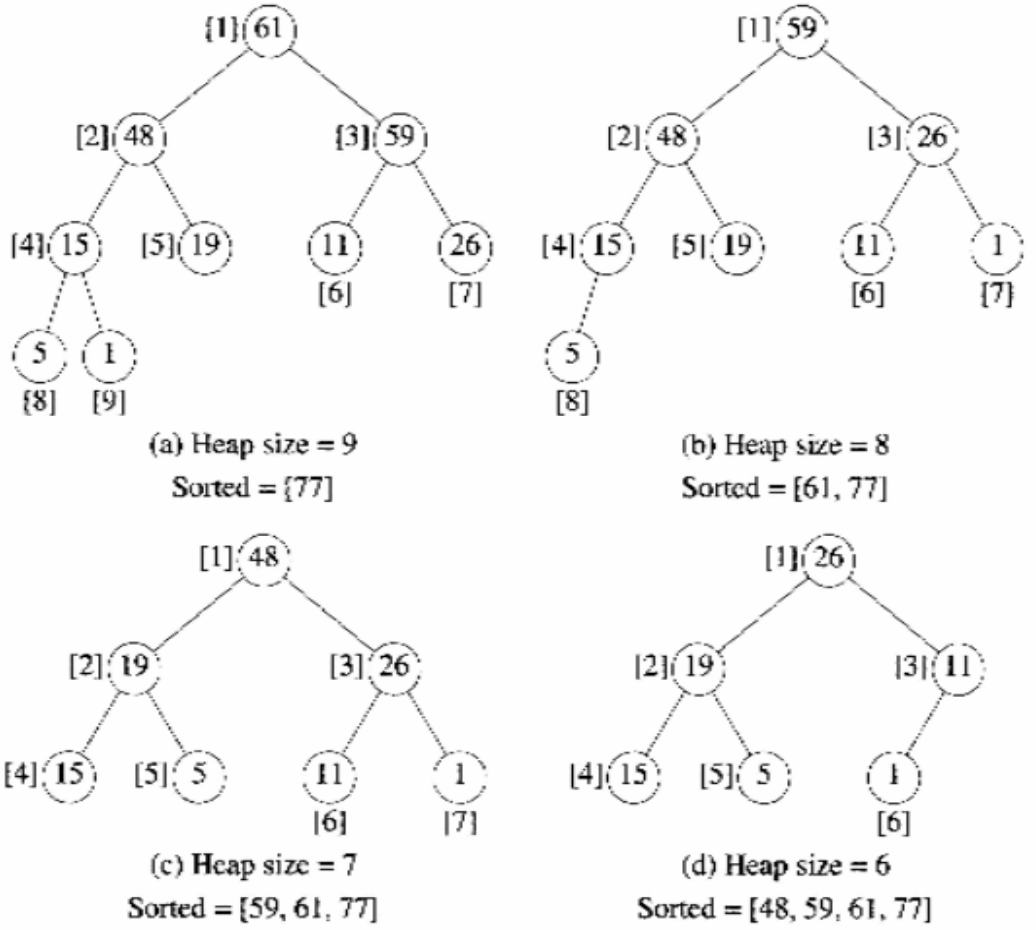


Figure 7.8: Heap Sort example (continued on next page)

Example

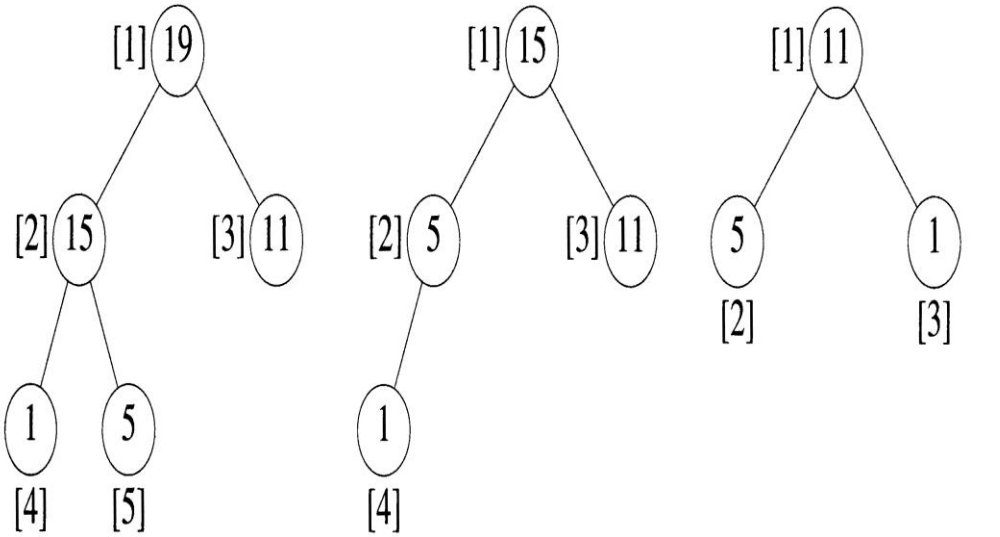


Figure 7.8: Heap Sort example

Quicksort

- Divide: Paritition the array $A[1..r]$ into two nonempty subarrays $A[1..p-1]$ and $A[p+1..r]$ around the pivot $A[p]$ such that
  - (every element in $A[1..p-1]$) $\leq A[p]$
  - $A[p] \leq$ (every element in $A[p+1..r]$)
- Conquer: Sort each of $A[1..p]$ and $A[p+1..r]$ by recursive calls to Quick sort
- Combine: Do nothing

Quick Sort

- One type of Insertion sort
- Pivot key K_i ,
  - If K_i is in position s(i),

    $K_j \leq K_{s(i)}$ for $j < s(i)$

    $K_j \geq K_{s(i)}$ for $j > s(i)$

    $\Rightarrow$ two sublists $(R_1, \ldots, R_{s(i)-1})$

    $(R_{s(i)+1}, \ldots, R_n)$

- Some procedure in the sublists

Quick Sort Code

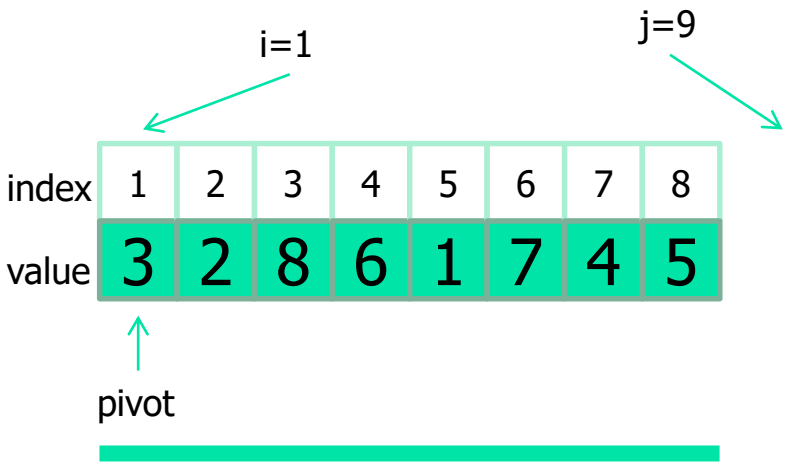
```
template <class T>
void QuickSort(T *a, const int left, const int right)
{ // Sort a[left:right] into nondecreasing order.
  // a[left] is arbitrarily chosen as the pivot. Variables i and j
  // are used to partition the subarray so that at any time a[m] ≤ pivot, m < i
  // and a[m] ≥ pivot, m > j. It is assumed that a[left] ≤ a[right + 1]
    if (left < right) {
        int i=left,
        j = right + 1,
        pivot = a[left];
        do {
            do i++; while (a[i] < pivot);
            do j--; while (a[j] > pivot);
            if(i < j) swap(a[i], a[j]);
        } while (i<j);
        swap(a[left], a[j]);

        QuickSort(a, left, j-1);
        QuickSort(a, j+1, right);
    }
}
```

Program 7.6: Quick Sort

Quicksort

```
if (left < right) {
        int i=left,
        j = right + 1,
        pivot = a[left];
        do {
            do i++; while (a[i] < pivot);
            do j--; while (a[j] > pivot);
            if(i < j) swap(a[i], a[j]);
        } while (i<j);
        swap(a[left], a[j]);

        QuickSort(a, left, j-1);
        QuickSort(a, j+1, right);
    }
```

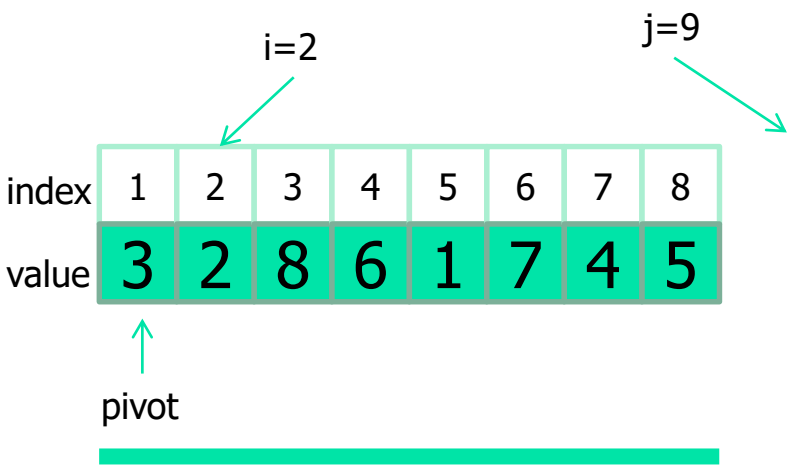
left = 1
right = 8



Quicksort

```
if (left < right) {
        int i=left,
        j = right + 1,
        pivot = a[left];
        do {
            do i++; while (a[i] < pivot);
            do j--; while (a[j] > pivot);
            if(i < j) swap(a[i], a[j]);
        } while (i<j);
        swap(a[left], a[j]);

        QuickSort(a, left, j-1);
        QuickSort(a, j+1, right);
    }
```

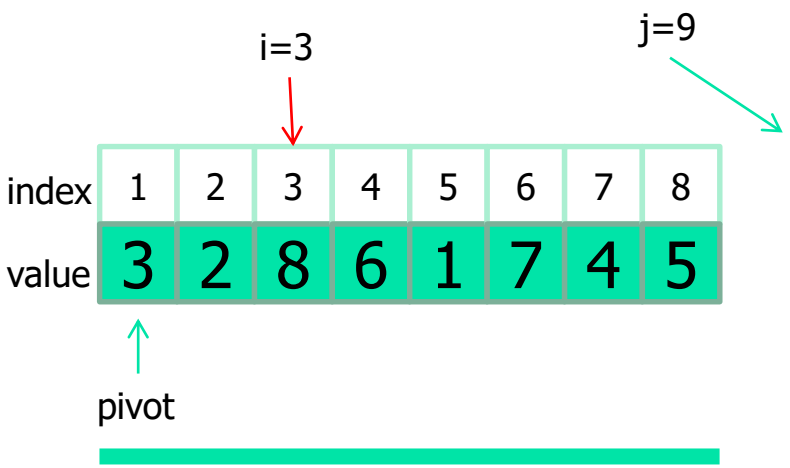
left = 1
right = 8



Quicksort

```
if (left < right) {
        int i=left,
        j = right + 1,
        pivot = a[left];
        do {
            do i++; while (a[i] < pivot);
            do j--; while (a[j] > pivot);
            if(i < j) swap(a[i], a[j]);
        } while (i<j);
        swap(a[left], a[j]);

        QuickSort(a, left, j-1);
        QuickSort(a, j+1, right);
    }
```

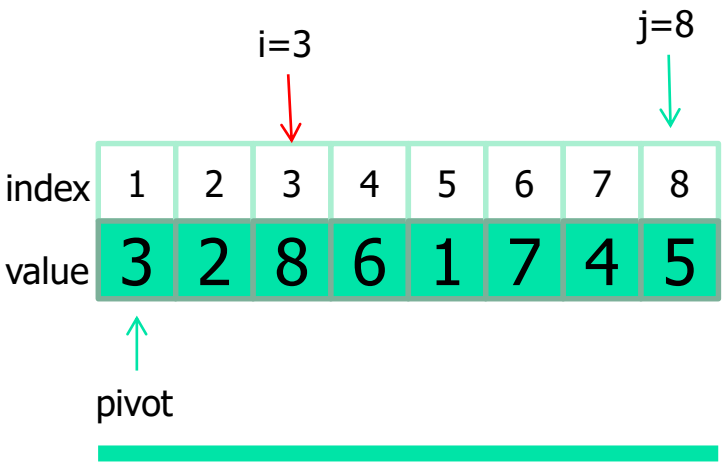
left = 1
right = 8



Quicksort

```
if (left < right) {
        int i=left,
        j = right + 1,
        pivot = a[left];
        do {
            do i++; while (a[i] < pivot);
            do j--; while (a[j] > pivot);
            if(i < j) swap(a[i], a[j]);
        } while (i<j);
        swap(a[left], a[j]);

        QuickSort(a, left, j-1);
        QuickSort(a, j+1, right);
    }
```

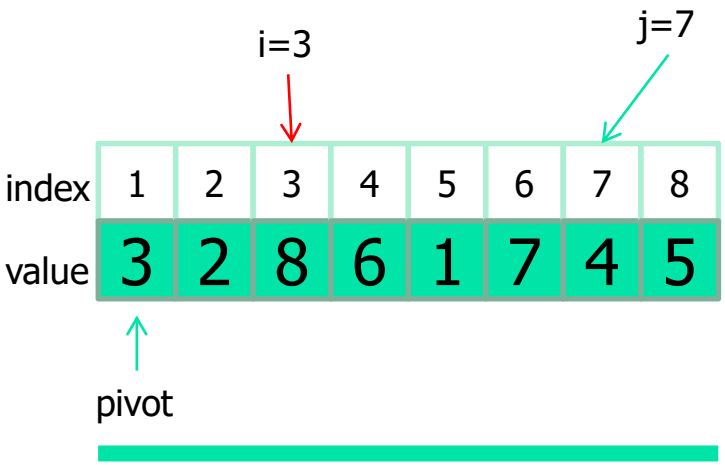
left = 1
right = 8



Quicksort

```
if (left < right) {
        int i=left,
        j = right + 1,
        pivot = a[left];
        do {
            do i++; while (a[i] < pivot);
            do j--; while (a[j] > pivot);
            if(i < j) swap(a[i], a[j]);
        } while (i<j);
        swap(a[left], a[j]);

        QuickSort(a, left, j-1);
        QuickSort(a, j+1, right);
    }
```

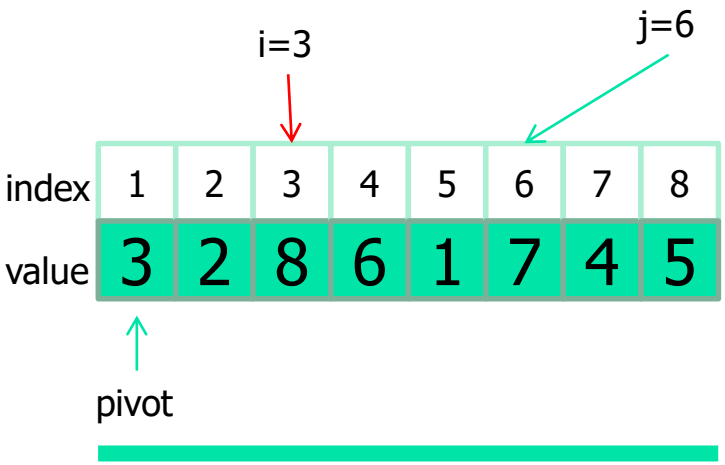
left = 1
right = 8



Quicksort

```
if (left < right) {
        int i=left,
        j = right + 1,
        pivot = a[left];
        do {
            do i++; while (a[i] < pivot);
            do j--; while (a[j] > pivot);
            if(i < j) swap(a[i], a[j]);
        } while (i<j);
        swap(a[left], a[j]);

        QuickSort(a, left, j-1);
        QuickSort(a, j+1, right);
    }
```

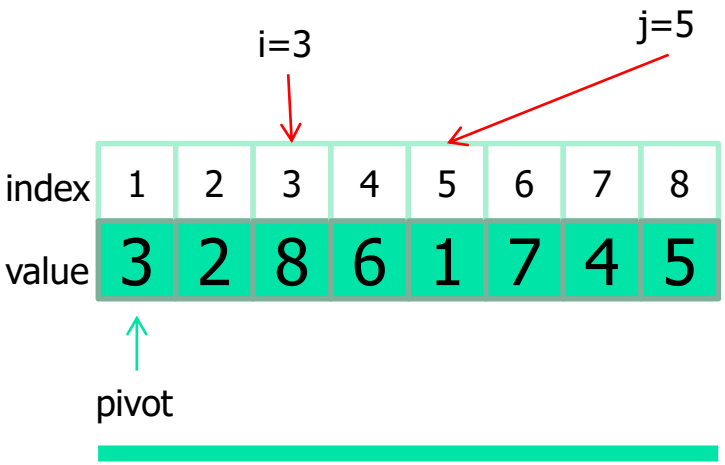
left = 1
right = 8



Quicksort

```
if (left < right) {
        int i=left,
        j = right + 1,
        pivot = a[left];
        do {
            do i++; while (a[i] < pivot);
            do j--; while (a[j] > pivot);
            if(i < j) swap(a[i], a[j]);
        } while (i<j);
        swap(a[left], a[j]);

        QuickSort(a, left, j-1);
        QuickSort(a, j+1, right);
    }
```

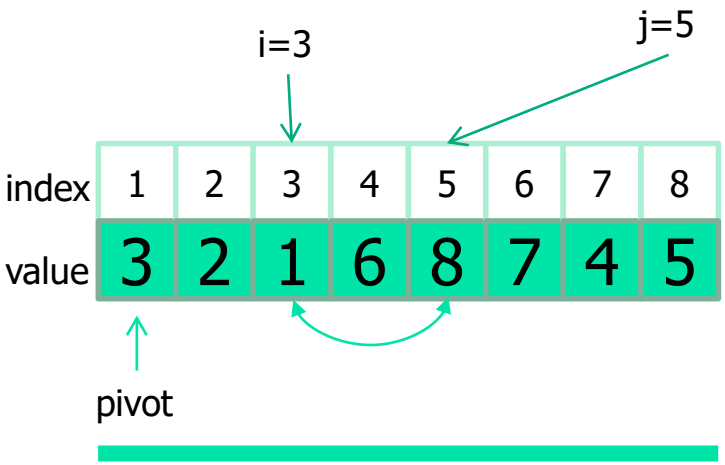
left = 1
right = 8



Quicksort

```
if (left < right) {
        int i=left,
        j = right + 1,
        pivot = a[left];
        do {
            do i++; while (a[i] < pivot);
            do j--; while (a[j] > pivot);
            if(i < j) swap(a[i], a[j]);
        } while (i<j);
        swap(a[left], a[j]);

        QuickSort(a, left, j-1);
        QuickSort(a, j+1, right);
    }
```

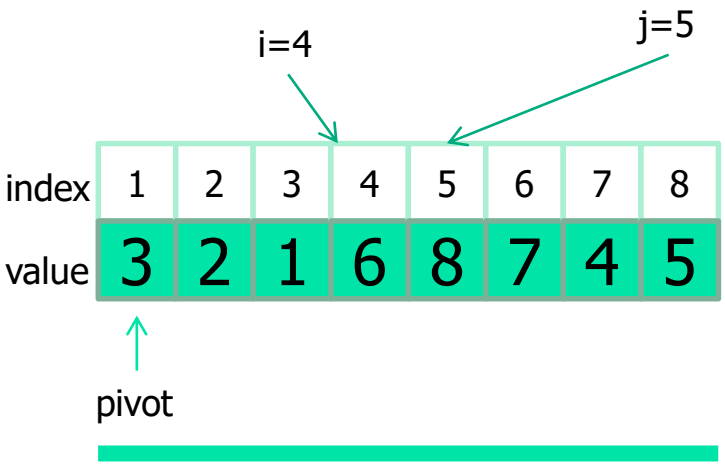
left = 1
right = 8



Quicksort

```
if (left < right) {
        int i=left,
        j = right + 1,
        pivot = a[left];
        do {
            do i++; while (a[i] < pivot);
            do j--; while (a[j] > pivot);
            if(i < j) swap(a[i], a[j]);
        } while (i<j);
        swap(a[left], a[j]);

        QuickSort(a, left, j-1);
        QuickSort(a, j+1, right);
    }
```

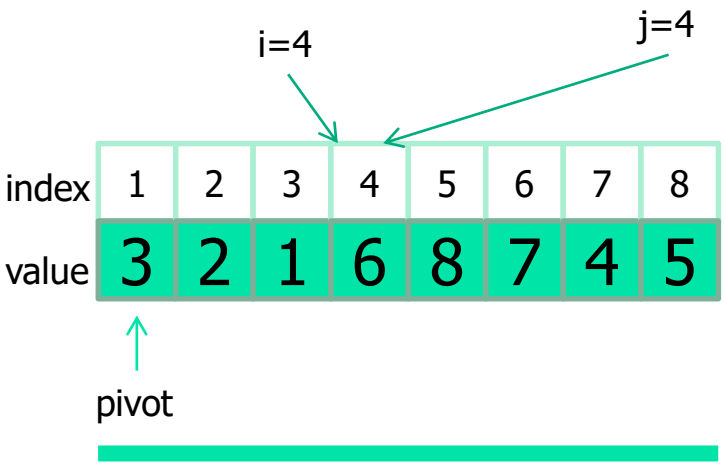
left = 1
right = 8



Quicksort

```
if (left < right) {
        int i=left,
        j = right + 1,
        pivot = a[left];
        do {
            do i++; while (a[i] < pivot);
            do j--; while (a[j] > pivot);
            if(i < j) swap(a[i], a[j]);
        } while (i<j);
        swap(a[left], a[j]);

        QuickSort(a, left, j-1);
        QuickSort(a, j+1, right);
    }
```

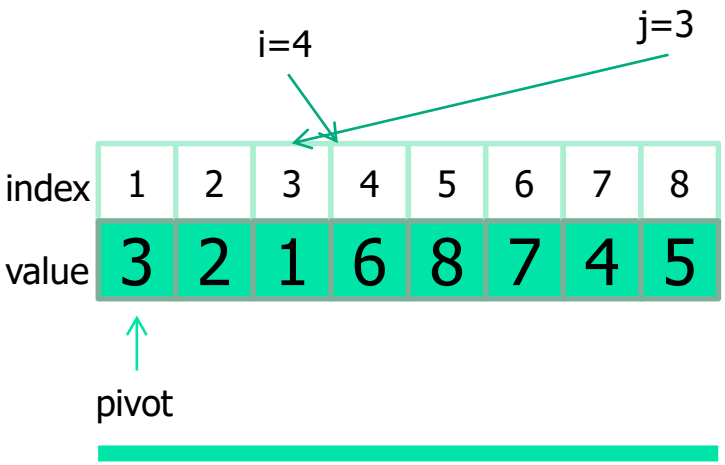
left = 1
right = 8



Quicksort

```
if (left < right) {
        int i=left,
        j = right + 1,
        pivot = a[left];
        do {
            do i++; while (a[i] < pivot);
            do j--; while (a[j] > pivot);
            if(i < j) swap(a[i], a[j]);
        } while (i<j);
        swap(a[left], a[j]);

        QuickSort(a, left, j-1);
        QuickSort(a, j+1, right);
    }
```

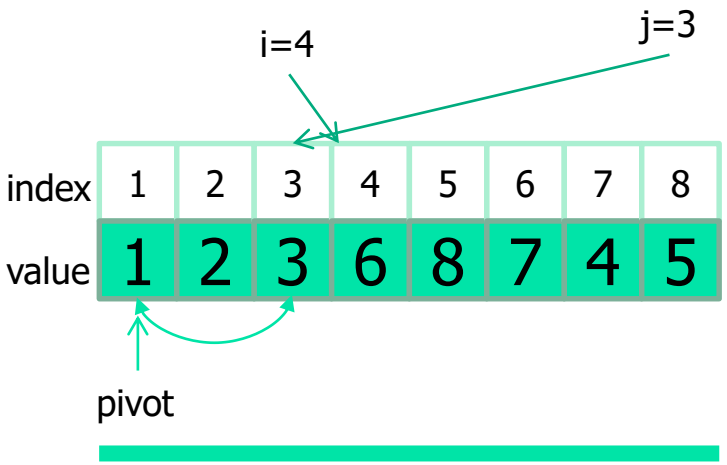
left = 1
right = 8



Quicksort

```
if (left < right) {
        int i=left,
        j = right + 1,
        pivot = a[left];
        do {
            do i++; while (a[i] < pivot);
            do j--; while (a[j] > pivot);
            if(i < j) swap(a[i], a[j]);
        } while (i<j);
        swap(a[left], a[j]);

        QuickSort(a, left, j-1);
        QuickSort(a, j+1, right);
    }
```

left = 1
right = 8



Quicksort

```
if (left < right) {
        int i=left,
        j = right + 1,
        pivot = a[left];
        do {
            do i++; while (a[i] < pivot);
            do j--; while (a[j] > pivot);
            if(i < j) swap(a[i], a[j]);
        } while (i<j);
        swap(a[left], a[j]);

        QuickSort(a, left, j-1);
        QuickSort(a, j+1, right);
    }
```

left = 1
right = 8