Data Structures and Algorithms
(ECEG 4171)

Chapter Four
Searching and Sorting Algorithms

Ephrem A. (M.Sc.)
School of Electrical and Computer Engineeing
Chair of Computer Engineering
11th November, 2019

Searching Algorithms

- Search can be viewed as a process to determine if an element with a particular value is a member of a particular set.
- There are two types of search algorithms:
  - algorithms that don't make any assumptions about the order of the sequence,
  - and algorithms that assume the sequence is already in order
- Two basic searching algorithms:
  - Linear/sequential search
  - Binary search (not to be confused with binary search tree (Ch-5)).
- The search algoirthms we discuss in this chapter are only applicable for sequences that are *indexable*.

Linear Search

- The simplest search algorithm.
- In a sequential search:
  1. Every element in the array will be examined sequentially, starting from the first element.
  2. The process will be repeated until the last element of the array or until the searched data is found.

```
public int sequentialSearch(int[] data, int item) {
    // if index is still -1 at the end of this method,
    // the item is not in this array.
    int index = -1;
    // loop through each element in the array.
    // if we find our search term, exit the loop.
    for (int i=0; i < data.length; i++) {
        if (data[i] == item) {
            index = i;
            break;
        }
    }
    return index;
}
```

Running Time:

Worst-case: $O(n)$
Average-case: $O(n)$
Best-case: $O(1)$

Linear Search

- The simplest search algorithm.
- In a sequential search:
  1. Every element in the array will be examined sequentially, starting from the first element.
  2. The process will be repeated until the last element of the array or until the searched data is found.

```
public int sequentialSearch(int[] data, int item) {
    // if index is still -1 at the end of this method,
    // the item is not in this array.
    int index = -1;
    // loop through each element in the array.
    // if we find our search term, exit the loop.
    for (int i=0; i < data.length; i++) {
        if (data[i] == item) {
            index = i;
            break;
        }
    }
    return index;
}
```

Running Time:

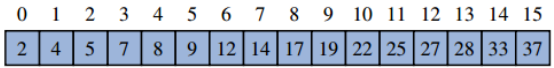
Worst-case: $O(n)$
Average-case: $O(n)$
Best-case: $O(1)$

Discuss how the worst case running time can be improved if the list is sorted?

Binary Search

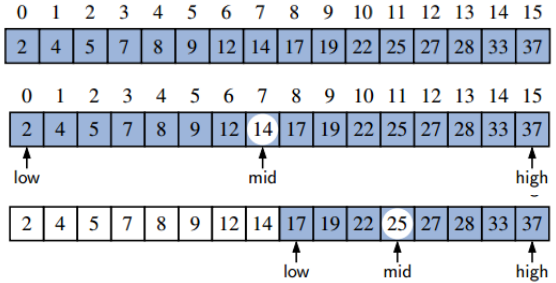
- When the sequence is sorted and indexable, there is a more efficient algorithm: binary search.
- It uses divide and conquer paradigm.
- Binary search works as follows:
  - To search for a target value in a sequence with minimum index `low` and maximum index `high`:
  - First we compare the target value with middle element in the list
  - If the target equals the middle element, then we have found the item we are looking for, and the search terminates successfully.
  - If the target is less than the middle element, then we recur on the first half of the sequence, that is, on the interval of indices from `low` to `mid - 1`.
  - If the target is greater than the median candidate, then we recur on the second half of the sequence, that is, on the interval of indices from `mid + 1` to `high`.

Example: Searching for 22.

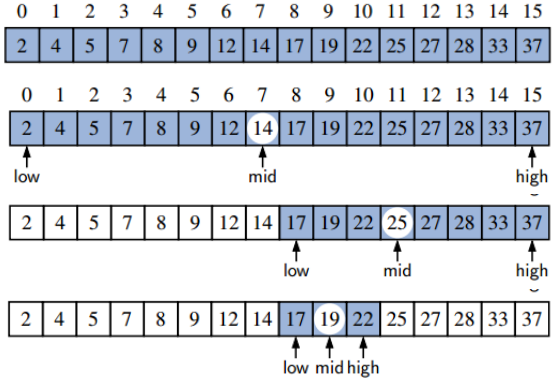


Example: Searching for 22.

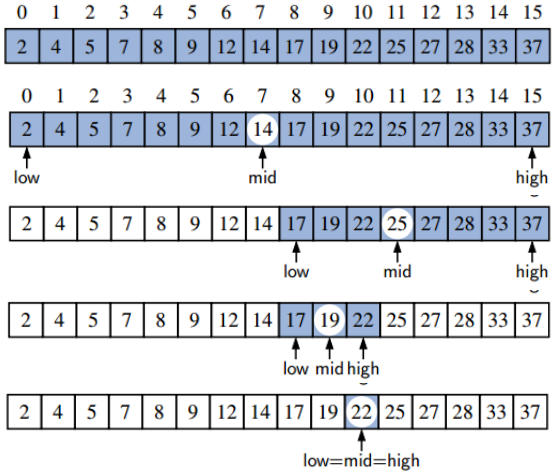
Example: Searching for 22.



Example: Searching for 22.



Example: Searching for 22.



Nonrecusrive Implementation

```
public int binarySearch(int [] data, int target) {
    int index = -1;
    int low = 0;
    int high = data.length-1;
    int mid;
    while (high >= low) {
        mid = (high + low)/2;
        if (target < list[mid]) {
            high = mid - 1;
        } else if (target > data[mid]) {
            low = mid + 1;
        } else {
            index = mid;
            break;
        }
    }
    return index;
}
```

Best-case analysis: occurs when the item is located in the middle of the list $\Rightarrow O(1)$

Worst-case analysis: search term is not in the list or the search term is one item away from the middle of the list or when the search term is the first or last item in the list. If it takes k for the loop to terminate:

$$\begin{aligned} n/2^k &= 1 \\ \Rightarrow k &= lgn \\ \Rightarrow T(n) &= O(lgn) \end{aligned}$$

Average case occurs when the search term is anywhere else in the list: $O(lgn)$

Recursive Implementation

```
public int binarySearch(int [] data, int target, int low, int high) {
    if (low > high)
        return -1; // interval empty; no match
    else {
        int mid = (low + high) / 2;
        if (target == data[mid])
            return mid; // found a match
        else if (target < data[mid])
            // recur left of the middle
            return binarySearch(data, target, low, mid-1);
        else
            // recur right of the middle
            return binarySearch(data, target, mid + 1, high);
    }
}
```

Worst-case running:

$T(n) = T(n/2) + \Theta(1)$
$\Rightarrow T(n) = \Theta(lgn)$ using case 2 of the Master theorem.

Sorting Algorithms

- Sorting algorithms solve the following problem:
  Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
  Output: A permutation (reordering) $\langle a_1', a_2', \ldots, a_n' \rangle$ of the input sequence such that $\langle a_1' \leq a_2' \leq \cdots \leq a_n' \rangle$
- Sorting algorithms involve two operations:
  - comparisons: comparing one value in a list with another (`a[1] > a[3]`),
  - data movement: copying an element from one position to another (aka assignment), e.g., `a[3] = a[1]`.
- Goal: minimize the number of comparisons and the number of moves needed to sort the array.
- Sorting algorithms we will cover in this chapter:

Elementary sorting algorithms

- Selection sort
- Bubble sort
- Insertion sort

Advanced sorting algorithms

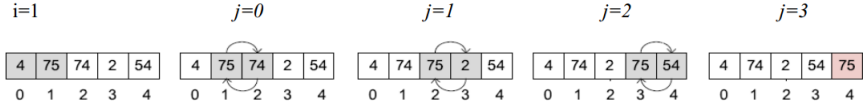
- Merge sort
- Heap sort
- Quick sort

Bubble Sort

- Is not used in practice.
- Strategy:
  - Go through multiple passes over the array.
  - In every pass:
    - Compare adjacent elements in the list.
    - Exchange the elements if they are out of order.
    - Each pass moves the largest (or smallest) elements to the end of the array
  - Repeating this process in several passes eventually sorts the array into ascending (or descending) order,
- Notice that in each pass, the largest item "bubbles"down the list until it settles in its final position and hence the name.

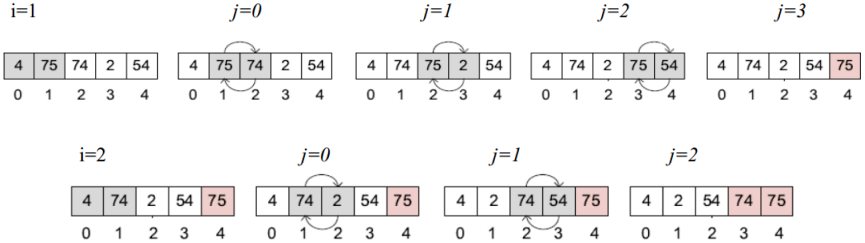
Bubble sort example,

Pass



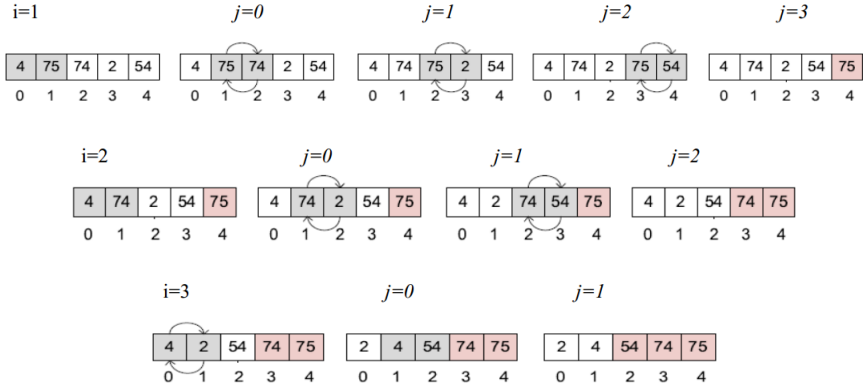
Bubble sort example,

Pass



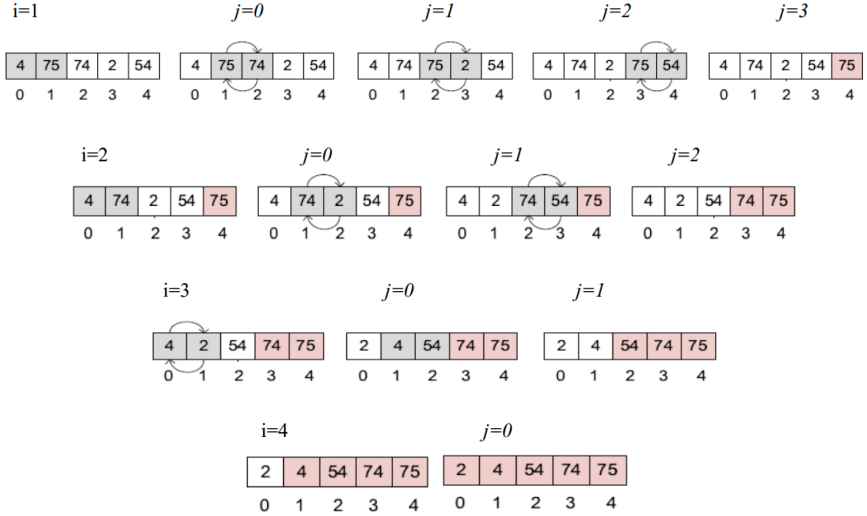
Bubble sort example,

Pass



Bubble sort example,

Pass



Implementation and Analysis

```
public static void bubbleSort(int [] a){
    for(int i = 1; i < a.length; i++){
        for(int j = 0; j < a.length−i; j++)
            if(a[j] > a[j+1])
                swap(a, j, j+1);
    }
}

private static void swap(int[] a, int i, int j){
    if(i == j) return;
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

For any case (best-, average-, or worst-):
# of comparison $C(n)$ is given as

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1 = \sum_{i=0}^{n-2} \big((n-2-i)-0+1\big)$$

$$= \sum_{i=0}^{n-2} (n-i-1) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

The number of swaps $S(n)$ is given as:
Worst-case (reverse order): $S_{worst}(n) = C(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$
Avg-case (half reverse): $S_{avg}(n) = C(n)/2 = \frac{n(n-1)}{2} \in \Theta(n^2)$
Best-case (sorted): $S_{best}(n) = 0$

Selection Sort

- One of the simplest algorithms.
- For every iteration i:
  - Find the smallest element in the remaining entry.
  - Swap the ith element and the minimum element

```
public static void selectionSort (int [] a){
    for(int i = 0; i < a.length; i++){
        int minInd = i;
        for(int j = i+1; j < a.length; j++){
            if(a[j] < a[minInd])
                minInd = j;
        }
        swap(a, i, minInd);
    }
}
```

Consider the following example

42	20	17	13	28	14	23	15
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

Consider the following example

42	20	17	13	28	14	23	15
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

		i=0	1	2	3	4	5	6
[0]	42	13	13	13	13	13	13	13
[1]	20	20	14	14	14	14	14	14
[2]	17	17	17	15	15	15	15	15
[3]	13	42	42	42	17	17	17	17
[4]	28	28	28	28	28	20	20	20
[5]	14	14	20	20	20	28	23	23
[6]	23	23	23	23	23	23	28	28
[7]	15	15	15	17	42	42	42	42

Analysis Selection Sort

- The number of comaprisons:

$$C(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} \big((n-1) - (i+1) + 1\big) \in \Theta(n^2)$$

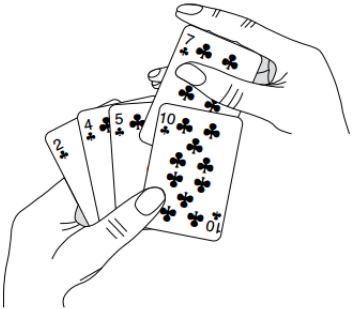
  For all best-, avg-, and worst-cases.

- The number of swaps:

$$S(n) = \Theta(n) \text{ For all cases (Good property of selection sort)}$$

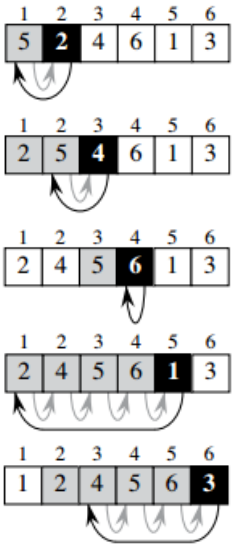
Insertion Sort

- Similar with card sorting: start with empty hand and keep inserting
- Two parts: Sorted part and Unsorted part



- Initially, all items in the unsorted group and the sorted group is empty.
- The principle behind insertion sort is to remove an element from an unsorted input list and insert in the correct position in the already-sorted list.

Insertion sort example,



Implementation:

```
1   public static void insertionSort (int [] a){
2       for(int i = 1; i < a.length; i++){
3           int key = a[i];
4           int j = i-1;
5           while(j >= 0 && a[j] > key){
6               a[j+1] = a[j];
7               j--;
8           }
9           a[j+1] = key;
10      }
11  }
```

Implementation and Analysis

```
public static void insertionSort (int [] a){
    for(int i = 1; i < a.length; i++){
        int key = a[i];
        int j = i-1;
        while(j >= 0 && a[j] > key){
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = key;
    }
}
```

Number of comparisons depends on the nature of the input:

Worst-case: when `a[j]` is always $>$ `key` in the `while` loop $\Rightarrow$ Reverse sorted.
$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \frac{n(n-1)}{2} \in \Theta(n^2)$

Best-case: when `a[j]` is always $<=$ `key` in the `while` loop $\Rightarrow$ sorted.
$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$

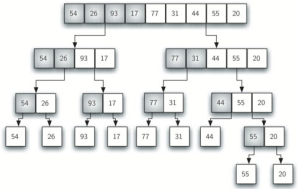
Avg-case: When elements are half sorted and half reverse sorted.
$C_{avg}(n) = \frac{n(n-1)}{4} \in \Theta(n^2)$

Merge Sort

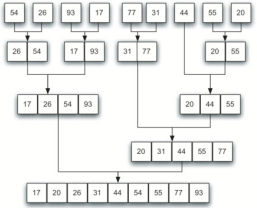
- Closely follows the divide-and-conquer paradigm.
  - ▷ **Divide:** Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.
  - ▷ **Conquer:** Sort the two subsequences recursively using merge sort.
  - ▷ **Combine:** Merge the two sorted subsequences to produce the sorted answer.
- The recursion bottoms out when the sequence to be sorted has length 1, in which case there is no work to be done.
- It does not sort in place.
  - – A sorting algorithm sorts in place if only a constant number of elements of the input array are ever stored outside the array.

Merge sort example

Splitting



Merging



Merge sort Implementation

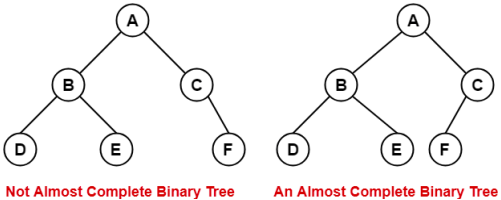
```
public static void mergeSort(int [] a, int lo, int hi){
    if (lo >= hi) return;
    int mid = (hi+lo)/2;
    mergeSort(a, lo, mid);
    mergeSort(a, mid+1, hi);
    merge(a, lo, mid, hi);
}

public static void merge(int [] a, int lo, int mid, int hi){
    int i = lo, j = mid+1;
    for(int k = lo; k <= hi; k++)
        aux[k] = a[k];
    for(int k = lo; k <= hi; k++){
        if (i > mid) a[k] = aux[j++];
        else if (j > hi) a[k] = aux[i++];
        else if (aux[j] < aux[i]) a[k] = aux[j++];
        else a[k] = aux[i++];
    }
}
```

Sorting Algorithms

- So far, we have introduced two sorting algorithms that sort n real numbers: the insertion sort and the merge sort.
- Insertion sort takes $\Theta(n^2)$ time in the worst case and sorts **in place**.
- A sorting algorithm sorts in place if only a constant number of elements of the input array are ever stored outside the array.
- Merge sort has a better asymptotic running time, $\Theta(nlgn)$ but the MERGE procedure it uses does not operate in place.
- In this chater, we will see two sorting algorithm:
  - Heapsort
  - Quicksort

Introduction to Heaps

- The **heap** data structure is an array object that must satisfy two properties:
  1. Can be viewed as a nearly complete binary tree.
  2. Satisfies the heap property.
- A nealy complete binary tree is a binary tree in which the last level of the tree is not completely full and all nodes are as far left as possible.
- The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.



Not Almost Complete Binary Tree An Almost Complete Binary Tree

- Each node of the tree corresponds to an element of the array.

Introduction to Heaps

- An array A that represents a heap has two attributes: *A.length* (the number of elements in the array), and *A.heap_size*, (how many elements in the heap are stored within array A).
  - Hence, $0 \leq A.heap\_size \leq A.length$
- The root of the tree is $A[1]$, and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

$$PARET(i) = \lfloor i/2 \rfloor$$
$$LEFT(i) = 2i$$
$$RIGHT(i) = 2i + 1$$

Heap Property

- There are two kinds of heaps: max-heap and min-heap.
- Max heaps satisfy the max-heap property: that for every node i other than the root,

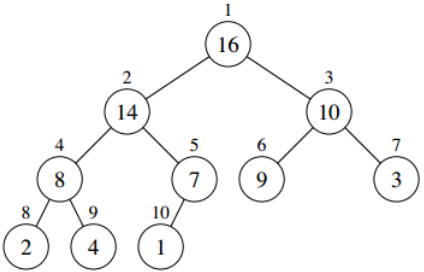
$$A[Parent(i)] \geq A[i]$$

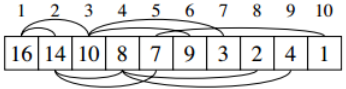
- That is, the value of a node is at most the value of its parent.
- Min-heaps work in the opposite way; the min-heap property is that for every node i other than the root,

$$A[Parent(i)] \leq A[i]$$

- That is, the value of a node is at least the value of its parent.
- The **height** of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf, and the height of the heap is the height of its root.

Example: max-heap





- The ith element in the array corresponds to the ith node in the tree.

Example: What is the minimum and maximum numbers of elements in a heap of height h?

Example: What is the minimum and maximum numbers of elements in a heap of height h?

Since a heap is an almost-complete binary tree (complete at all levels except possibly the lowest), it has at most $2^{h+1} - 1$ elements (if it is complete) and at least $2^h - 1 + 1 = 2^h$ elements (if the lowest level has just 1 element and the other levels are complete).

Example: What is the minimum and maximum numbers of elements in a heap of height h?

Since a heap is an almost-complete binary tree (complete at all levels except possibly the lowest), it has at most $2^{h+1} - 1$ elements (if it is complete) and at least $2^h - 1 + 1 = 2^h$ elements (if the lowest level has just 1 element and the other levels are complete).

Example: Show that an n-element heap has height $\lfloor lgn \rfloor$.

Example: What is the minimum and maximum numbers of elements in a heap of height h?

Since a heap is an almost-complete binary tree (complete at all levels except possibly the lowest), it has at most $2^{h+1} - 1$ elements (if it is complete) and at least $2^h - 1 + 1 = 2^h$ elements (if the lowest level has just 1 element and the other levels are complete).

Example: Show that an n-element heap has height $\lfloor lgn \rfloor$.

Given an n-element heap, we know that from the above example,

$$2^h \leq n \leq 2^{h+1} - 1 < 2^{h+1}$$
$$h \leq lgn < h + 1$$
$$\Rightarrow h = \lfloor lgn \rfloor \text{ since h is an integer}$$

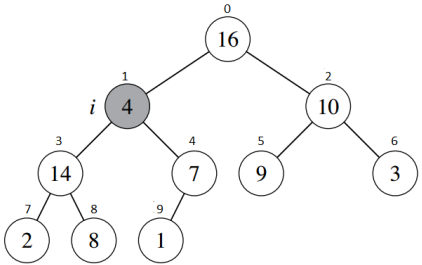
Maintaining Heap Property

- `maxHeapify` is a routine used to maintain the max-heap property.
- Its inputs are an array A and an index i into the array.
  - Before `maxHeapify`, A[i] may be smaller than its children.
  - Assume left and right subtrees of i are max-heaps.
  - After `maxHeapify`, subtree rooted at i is a max-heap.

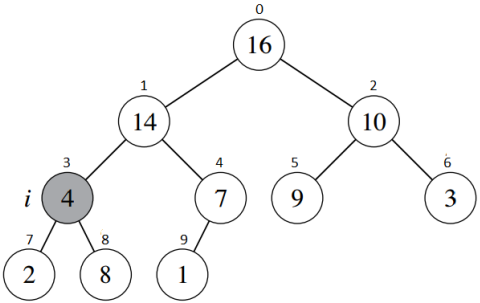
- How `maxHeapify` works:
  - Comapare $A[i]$, $A[Left(i)]$, and $A[Right(i)]$.
  - If necessary, swap $A[i]$ with the larger of the two children to preserve heap property.
  - Continue this process of comparing and swapping down the heap, until subtree rooted at i is max-heap. If we hit a leaf, then the subtree rooted at the leaf is trivially a max-heap.

```
public static void maxHeapify(int[] a, int i, int
     heapIndex){
    int left = 2*i + 1; // index of left child
    int right = 2*i + 2; // index of right child
    int largest = i;
    if(left <= heapIndex && a[left] > a[largest])
        largest = left;
    if(right <= heapIndex && a[right] > a[largest])
        largest = right;
    if(largest != i){
        swap(a, i, largest);
        maxHeapify(a, largest, heapIndex);
    }
}
```

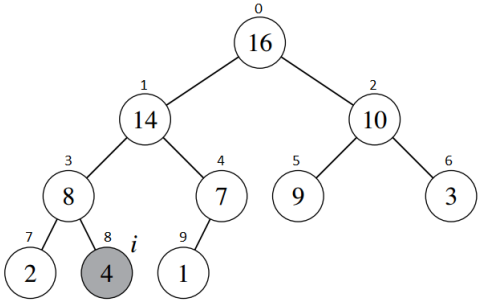
Example: Run `maxHeapify(A, 1)` on the following heap example.



A[1] at node i = 1 violates the max-heap property since it is not larger than both children.



The max-heap property is restored for node 1 by exchanging $A[1]$ with $A[3]$, which destroys the max-heap property for node 3.



The recursive call `maxHeapify(A,3)` now has $i = 3$. After swapping $A[3]$ with $A[8]$, node 3 is fixed up, and the recursive call `maxHeapify(A,8)` yields no further change to the data structure.

Running Time of MAX-HEAPIFY(A, i)

```
public static void maxHeapify(int[] a, int i, int
     heapIndex){
    int left = 2*i + 1;
    int right = 2*i + 2;
    int largest = i;
    if( left <= heapIndex && a[left] > a[largest])
        largest = left;
    if( right <= heapIndex && a[right] > a[largest])
        largest = right;
    if( largest != i){
        swap(a, i, largest);
        maxHeapify(a, largest, heapIndex);
    }
}
```

The running time of `maxHeapify` on a subtree of size n rooted at a given node i is the $\Theta(1)$ time to fix up the relationships among the elements $A[i]$, $A[LEFT(i)]$, and $A[RIGHT(i)]$, plus the time to run `maxHeapify` on a subtree rooted at one of the children of node i (assuming that the recursive call occurs). The children's subtrees each have size at most $2n/3$. Therefore, the recursive running time is:

$$T(n) \leq T(2n/3) + \Theta(1)$$

Using case 2 of the Master theorem the solution is $T(n) = O(lgn)$.

Alternatively (a simpler method), we can characterize the running time of `maxHeapify` on a node of height h as $O(h) = O(lgn)$

Building a Heap

- We can use the procedure `maxHeapify` in a bottom-up manner to convert an array $A[0..n-1]$, where $n = A.length$, into a max-heap.
- The elements in the subarray $A[\lfloor n/2 \rfloor], A[\lfloor n/2 \rfloor + 1], ..., A[n-1]$ are all leaves of the tree, and so each is a 1-element heap to begin with.
- The procedure `buildMaxHeap` goes through the remaining nodes of the tree and runs `maxHeapify` on each one.

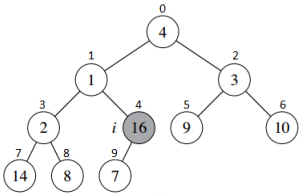
```
public static void buildMaxHeap(int[] a){
    int heapIndex = a.length−1; // index of the last element in the heap.
    for(int k = a.length/2−1; k >= 0; k−−)
        maxHeapify(a, k, heapIndex);
}
```

Example: Build a max-heap from the following unsorted array

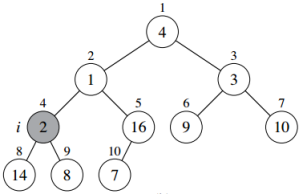
	0	1	2	3	4	5	6	7	8	9
A	4	1	3	2	16	9	10	14	8	7

	0	1	2	3	4	5	6	7	8	9
A	4	1	3	2	16	9	10	14	8	7

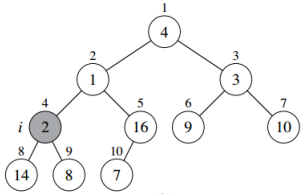
i = 4



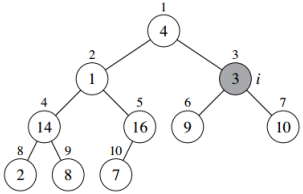
i = 4



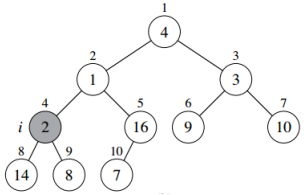
i = 4


1
4
2
1
3
3
4
i
2
5
16
6
9
7
10
8
14
9
8
10
7

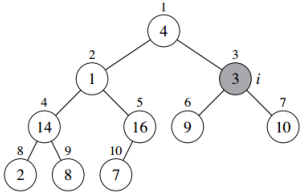
i = 3


1
4
2
1
3
3
i
4
14
5
16
6
9
7
10
8
2
9
8
10
7

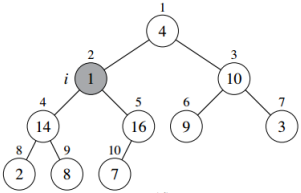
i = 4


1
4
2
1
3
3
4
i
2
5
16
6
9
7
10
8
14
9
8
10
7

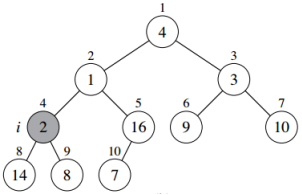
i = 3


1
4
2
1
3
3
i
4
14
5
16
6
9
7
10
8
2
9
8
10
7

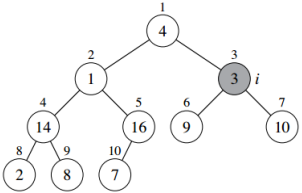
i = 2


1
4
2
i
1
3
10
4
14
5
16
6
9
7
3
8
2
9
8
10
7

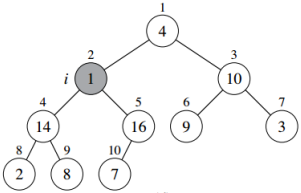
i = 4


1
4
2
1
3
3
4
i
2
5
16
6
9
7
10
8
14
9
8
10
7

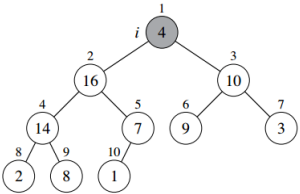
i = 3

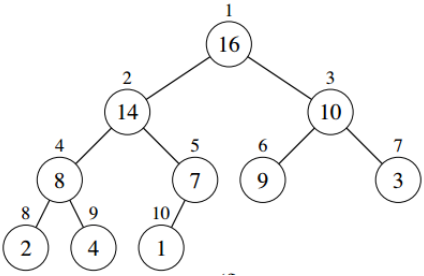

1
4
2
1
3
3
i
4
14
5
16
6
9
7
10
8
2
9
8
10
7

i = 2


1
4
2
i
1
3
10
4
14
5
16
6
9
7
3
8
2
9
8
10
7

i = 1


1
i
4
2
16
3
10
4
14
5
7
6
9
7
3
8
2
9
8
10
1



Observe that for subsequent iterations of the for loop in `BUILD-MAX-HEAP`, whenever `maxHeapify` is called on a node, the two subtrees of that node are both max-heaps.

buildMaxHeap Running Time

- Each call to `maxHeapify` costs $O(lgn)$ time and `buildMaxHeap` makes $O(n)$ such calls.
- Thus, the running time is $O(nlgn)$. This upper bound, though correct, is **not asymptotically tight**.
- We can observe that the time for `maxHeapify` to run at a node varies with the height of the node in the tree, and the heights of most nodes are small.
- Our tighter analysis relies on the properties that an n-element heap has height $\lfloor lgn \rfloor$
- And an n-element heap has at most $\cdot\lceil n/2^{h+1} \rceil$ nodes of any height h (see Exercise 6.3-3 on CLRS).

buildMaxHeap Running Time

- The time required by `maxHeapify` when called on a node of height h is $O(h)$.
- We can express the total cost of `buildMaxHeap` as being bounded from above by

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

- The last summation can be evaluated as

$$\begin{aligned} \sum_{h=0}^{\infty} \frac{h}{2^h} &= \frac{1/2}{(1-1/2)^2} \\ &= 2 . \end{aligned}$$

- Thus, we can bound the running time of `buildMaxHeap` as

$$\begin{aligned} O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(n) . \end{aligned}$$

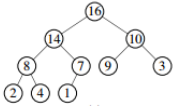
The Heapsort Algorithm

- Given an input array $A[0..n-1]$, the heapsort algorithm acts as follows:
  - Builds a max-heap from the array.
  - Starting with the root (the maximum element), the algorithm places the maximum element into the correct place in the array by swapping it with the element in the last position in the array.
  - "Discard" this last node (knowing that it is in its correct place) by decreasing the heap size, and calling `maxHeapify` on the new (possibly incorrectly-placed) root.
  - Repeat this "discarding" process until only one node (the smallest element) remains, and therefore is in the correct place in the array.

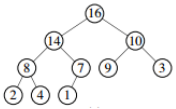
```
public static void heapSort(int [] a){
    int heapIndex = a.length−1;
    for(int k = a.length/2−1; k >= 0; k−−)
        maxHeapify(a, k, heapIndex);
    while(heapIndex >= 1){
        swap(a, 0, heapIndex−−);
        maxHeapify(a, 0, heapIndex);
    }
}
```

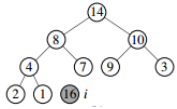
`heapSort` procedure takes time O(nlgn), (`buldMaxHeap` takes time O(n) and each of the n−1 calls to `maxHeapify` takes O(lgn)).

Example: Sort using Heapsort given the initial max-heap.

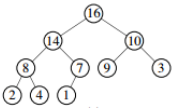


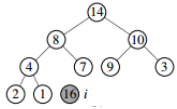
Example: Sort using Heapsort given the initial max-heap.

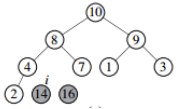




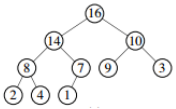
Example: Sort using Heapsort given the initial max-heap.

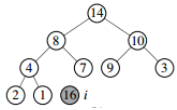


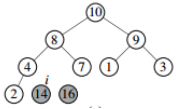


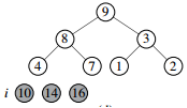


Example: Sort using Heapsort given the initial max-heap.

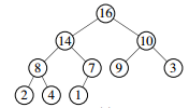


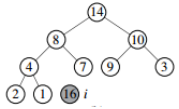


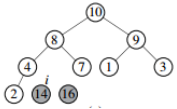


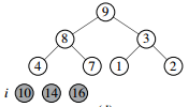


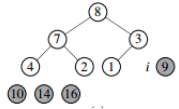
Example: Sort using Heapsort given the initial max-heap.



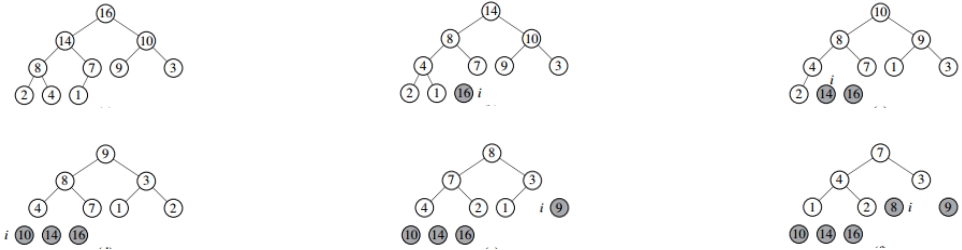




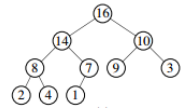


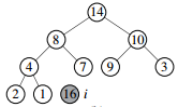


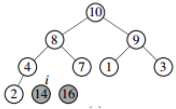
Example: Sort using Heapsort given the initial max-heap.

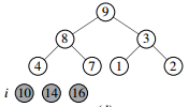


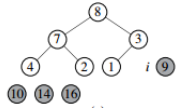
Example: Sort using Heapsort given the initial max-heap.

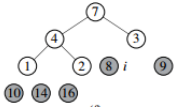


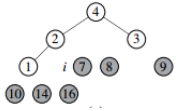




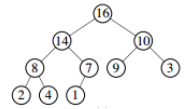


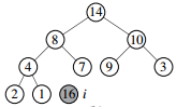


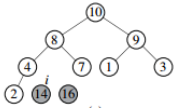


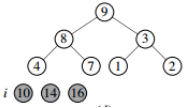


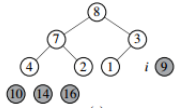
Example: Sort using Heapsort given the initial max-heap.

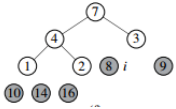


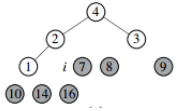


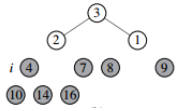




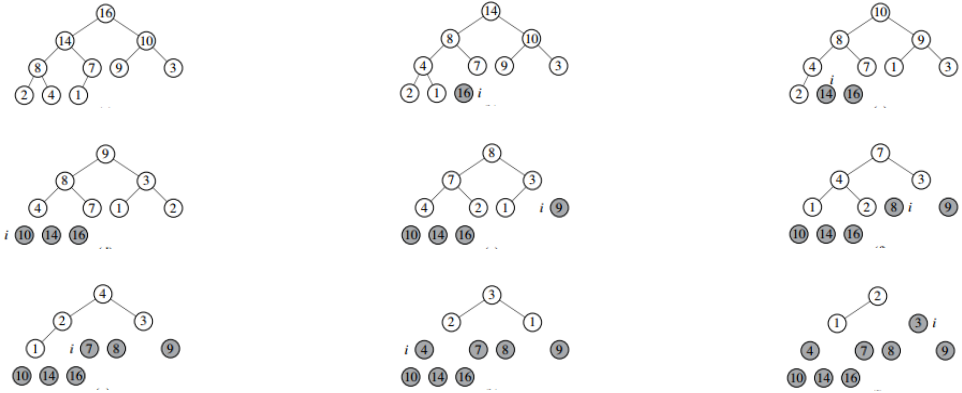






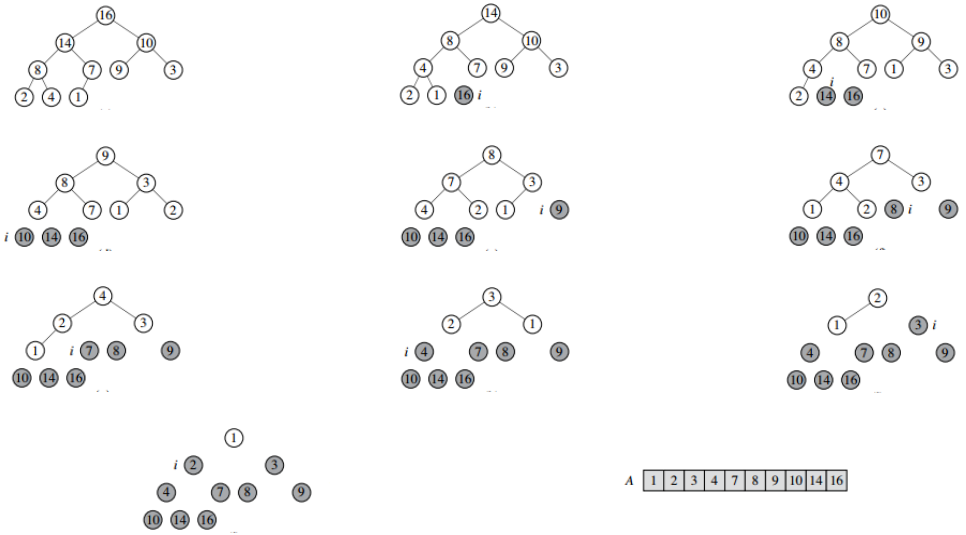


Example: Sort using Heapsort given the initial max-heap.



Example: Sort using Heapsort given the initial max-heap.

Example: Sort using Heapsort given the initial max-heap.



Quicksort

- Quicksort is the fastest comparison sort algorithm.
- It has a worst-case running time $\Theta(n^2)$.
- However, its average-case running time is remarkably efficient
  - Its expected running time is $\Theta(nlgn)$ and the constant terms hidden in the $\Theta(nlgn)$ are quite small.
  - It sorts in place.
  - It works even well in virtual-memory environments.
- Quicksort, like merge sort, applies the divide-and-conquer paradigm.
- To sort the subarray $A[p..r]$:
  - **Divide:** Partition $A[p..r]$, into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$, such that each element in the first subarray $A[p..q-1]$ is $\leq A[q]$ and $A[q]$ is $\leq$ each element in the second subarray $A[q+1..r]$.
  - **Conquer:** Sort the two subarrays by recursive calls to `QUICKSORT`.
  - **Combine:** No work is needed to combine the subarrays, because they are sorted in place.

Quicksort pseudocode

- Perform the divide step by a procedure `partition`, which returns the index q that marks the position separating the subarrays.

```
public static void quickSort(int [] a, int p, int r){
    if(p < r){
        int q = partition_(a, p, r);
        quickSort(a, p, q−1);
        quickSort(a, q+1, r);
    }
}
```

- To sort entire array, the initial call is `quickSort(a, 0, a.length-1)`.

Partitioning

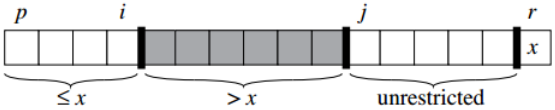
- Partition subarray $a[p..r]$ by the following procedure:

```
public static int partition(int[] a, int p, int r){
    int x = a[r];
    int i = p-1;
    for(int j = p; j <= r; j++){
        if(a[j] <= x){
            i++;
            swap(a, i, j);
        }
    }
    swap(a, i+1, r);
    return i+1;
}
```

- `partition` always selects the last element $a[r]$ in the subarray $a[p..r]$ as the **pivot** - the element around which to partition.

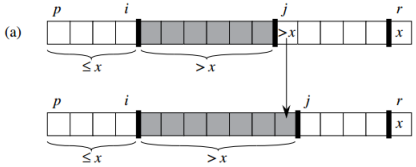
Partitioning

- As the procedure executes, the array is partitioned into four regions, some of which may be empty:
  1. All entries in $A[p..i]$ are $\leq$ pivot.
  2. All entries in $A[i+1..j-1]$ are $>$ pivot.
  3. $A[r] = pivot$.
  4. The fourth region is $A[j..r-1]$, whose entries have not yet been examined, and so we don't know how they compare to the pivot.

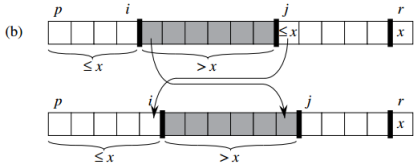


Partitioning regions

- If $A[j] > x$ the only action is to increment j.

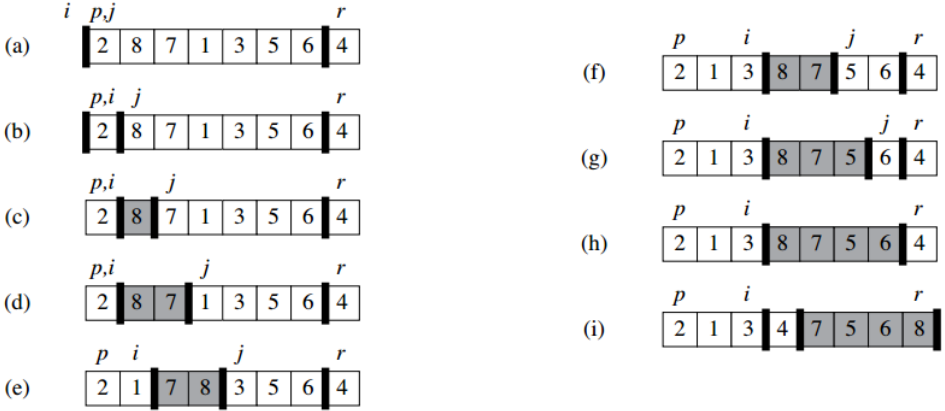


- If $A[j] \leq x$ index i is incremented, A[i] and A[j] are swapped, and then j is incremented.



Example: Run `partition(a, 0, a.length-1)` on the array $a = \langle 2, 8, 7, 1, 3, 5, 6, 4 \rangle$

Example: Run `partition(a, 0, a.length-1)` on the array $a = \langle 2, 8, 7, 1, 3, 5, 6, 4 \rangle$



Worst-Case Analysis of quicksort

- The running time of quicksort depends on whether the partitioning is **balanced** or **unbalanced**.
- The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with $n-1$ elements and one with 0 elements (this is the case when the pivot is either the smallest or largest element in the array).
- We assume that this unbalanced partitioning arises in each recursive call.
- The recursion costs $\Theta(n)$ time and the recursive call on an array of size 0 is just $T(0) = \Theta(1)$. So recurrence for the running time is

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(1) + \Theta(2) + \cdots + \Theta(n) \\ &= \sum_{i=1}^{n} \Theta(i) = \Theta(\sum_{i=1}^{n} i) = \Theta(n^2) \end{aligned}$$

Best-case Analysis

- Quicksort runs much faster when PARTITION produces two subproblems, each of size no more than $n/2$, since one is of size $\lfloor n/2 \rfloor$ and one of size $\lceil n/2 \rceil - 1$. The recurrence of the running time is then $T(n) = 2T(n/2) + \Theta(n)$
- By case 2 of the master theorem, this recurrence has the solution $\Theta(nlgn)$.

Average-case Analysis

- Using random sampling, we can select a randomly chosen element from the subarray $A[p..r]$ as the pivot.
- This ensures that the pivot element is equally likey to be any of the $r - p + 1$ elements in the subarray.

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

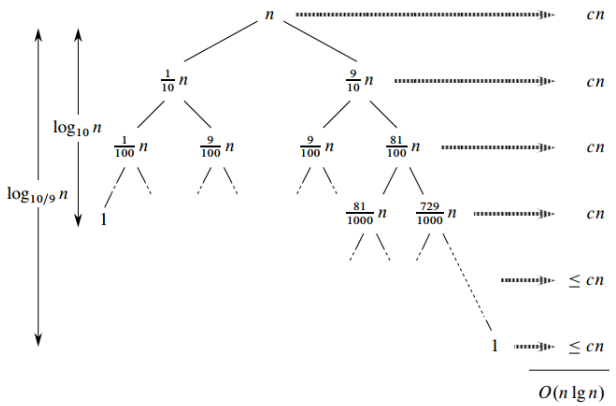
- The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION:

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Average-case Analysis

- The average-case running time of quicksort is much closer to the best case than to the worst case.
- In the average case, PARTITION produces a mix of "good" and "bad" splits.
- Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split which produces the recurrence:
$$T(n) = T(n/10) + T(9n/10) + \Theta(n)$$
- With a 9-to-1 proportional split at every level of recursion, which intuitively seems quite unbalanced, quicksort runs in $O(nlgn)$.
- In fact, any split of constant proportionality yields $O(nlgn)$ running time.

$T(n) = T(n/10) + T(9n/10) + \Theta(n)$



- We can also show that even a 99-to-1 split yields an $O(nlgn)$ running time.

Hoare Partition Scheme

- The `PARTITION` procedure we saw in previous slides is known as Lumuto's partition scheme.
- Another scheme called Hoare's partition scheme works by initializing two indexes that start at two end.

- The two indexes move towards each other until a smaller value on left side and greater value on the right side is found.
- Then the two values are swapped and the process is repeated.
- Hoare's scheme is more efficient than Lumuto's scheme because it does fewer swaps on average and creates efficient partitions even when all values are equal.

```
HOARE-PARTITION(A, p, r)
 1  x = A[p]
 2  i = p − 1
 3  j = r + 1
 4  while TRUE
 5      repeat
 6          j = j − 1
 7      until A[j] ≤ x
 8      repeat
 9          i = i + 1
10      until A[i] ≥ x
11      if i < j
12          exchange A[i] with A[j]
13      else return j
```

Comparing Sorting Algorithms

- There is no clear "best" sorting algorithm, because it depends of on properties such as efficiency, memory, and stability.
- Bubble sort
  - Runs $O(n^2)$ in both swaps and comparisons. For this reason, bubble sort is actually the least efficient sorting method.
- Selection sort
  - advantage is that the number of swaps is $O(n)$. Its disadvantage is that it does not stop early if the list is sorted.
- Insertion-Sort:
  - Mainly depends on the number of **inversions** (that is, the number of pairs of elements out of order).
  - Is quite effective for sorting sequences that are already almost sorted i.e. the number of inversions is small.

Comparing Sorting Algorithms

- Heap-Sort:
  - Is a natural choice on small- and medium-sized sequences, when input data can fit into main memory.
  - However, heap-sort tends to be outperformed by both quick-sort and merge-sort on larger sequences.
  - A standard heap-sort does not provide a stable sort, because of the swapping of elements.
- Quick-Sort:
  - Experimental studies have shown that it outperforms both heap-sort and merge-sort on many tests.
  - Quick-sort does not naturally provide a stable sort, due to the swapping of elements during the partitioning step.
  - For decades it was the default choice for a general-purpose, in-memory sorting algorithm.
  - Was included as the `qsort` sorting utility provided in C language libraries, and was the basis for sorting utilities on Unix operating systems for many years.
  - It has long been the standard algorithm for sorting arrays of primitive type in Java (`Arrays.sort(int[] a)`).

Comparing Sorting Algorithms

- Merge-Sort:
  - Is an excellent algorithm for situations where the input is stratified across various levels of the computer's memory hierarchy (e.g., cache, main memory, external memory).
  - The GNU sorting utility (and most current versions of the Linux operating system) relies on **Tim-sort**, a hybrid approach that is essentially a bottom-up merge-sort that takes advantage of initial runs in the data while using insertion-sort to build additional runs.
  - Java's `Collections.sort(list)` uses mergesort.

Application of Sorting

- Many important problems can be reduced to sorting:
  - *Search*: Binary search tests whether an item is in a dictionary in $O(lgn)$ time, provided the keys are all sorted.
  - *Closest pair* – Given a set of n numbers, how do you find the pair of numbers that have the smallest difference between them?
  - *Element uniqueness* – Are there any duplicates in a given set of n items?
  - *Frequency distribution* – Given a set of n items, which element occurs the largest number of times in the set?
  - *Selection* – What is the kth largest item in an array? If the keys are placed in sorted order, the kth largest can be found in constant time by simply looking at the kth position of the array.
  - etc...

Summary

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)

Java API for Sorting

- `java.util.Arrays` class contains `static` methods for processing arrays.
  - There are the 18 overloaded versions of the `sort()` method.
- `java.util.Collections` class provides over 50 static utility methods that implement algorithms for sorting, searching, shuffling, and maintaining collections, among other tasks.
- For sorting, use
  - `Arrays.sort(A)` to sort an array.
  - `Collections.sort(list)` to sort a list.
  - Both operate on objects that implement the `Comparable` interface.

Priority Queue

- The FIFO principle used in Queues may not always be an effective way to remove and insert objects:
- For example
  - An air-traffic control center decision to clear a flight for landing from among many approaching the airport may be influenced by factors such as each plane's distance from the runway, time spent waiting in a holding pattern, or amount of remaining fuel.
  - Due to the possibility of cancellations, airlines maintain a queue of standby passengers. The priority of a standby passenger is influenced by the check–in time of that passenger: Other considerations include the fare paid and frequent–flyer status.
  - Doctors in a hospital emergency room often choose to see next the "most critical" patient rather than the one who arrived first.

Priority Queues

- Priority Queue is an extension of queue with the following properties.
  - Every item has a priority associated with it.
  - An element with high priority is dequeued before an element with low priority.
  - If two elements have the same priority, they are served according to their order in the queue.

Priority Queue Entries

- In PQs, we have to keep track of an item and its key (priority).
- We model an element and its priority as a key-value composite known as an **entry**. This is known as **composition design pattern**.
- For priority queues, we use composition to pair a key k and a value v as a single object.
- To formalize this, we define the public interface, `Entry`.

```
public interface Entry<K, V> {
    K getKey(); // returns the key stored in this entry
    V getValue(); // returns the value stored in this entry
}
```

Priority Queue ADT

- By convention, we assume the entity with the lowest key has the highest priority.
- The priority queue ADT supports the following methods:

insert(k, v):	Creates an entry with key k and value v in the priority queue.
min():	Returns (but does not remove) a priority queue entry (k,v) having minimal key; returns null if the priority queue is empty.
removeMin():	Removes and returns an entry (k,v) having minimal key from the priority queue; returns null if the priority queue is empty.
size():	Returns the number of entries in the priority queue.
isEmpty():	Returns a boolean indicating whether the priority queue is empty.

Example: The following table shows a series of operations and their effects on an initially empty priority queue.

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }
removeMin( )	null	{ }
isEmpty( )	true	{ }

Comparing Keys in PQ

- Any object can serve as a key but we must be able to compare it in a meaningful way.
- Java provides two means of defining comparisons between object types.
  - ▷ The Comparable Interface (java.lang.Comparable)
    - Includes a single method, compareTo.
    - Defines *natural ordering* of its instances. E.g. compareTo method of the string class defines the natural ordering to be lexicographic.
    - The syntax a.compareTo(b) must return an integer i with the following meaning.
      $i < 0$ designates that $a < b$.
      $i = 0$ designates that $a = b$.
      $i > 0$ designates that $a > b$.
  - ▷ The Comparator Interface (java.util.Comparator)
    - Includes compare method which is external to class of the keys it compares.
    - We may want to compare objects according to some notion other than their natural ordering
    - Method signature is compare(a, b) that returns an integer with similar meaning to the compareTo method.

Comparable Example

```
public class Date implements Comparable<Date>{
    private final int day;
    private final int month;
    private final int year;
    public Date(int d, int m, int y){ day = d; month = m; year = y; }
    public int day() { return day; }
    public int month() { return month; }
    public int year() { return year; }
    public int compareTo(Date that){
        if(this.year > that.year ) return +1;
        if(this.year < that.year ) return -1;
        if(this.month > that.month) return +1;
        if(this.month < that.month) return -1;
        if(this.day > that.day ) return +1;
        if(this.day < that.day ) return -1;
        return 0;
    }
    public String toString(){ return month + "/" + day + "/" + year; }
}
```

Comparator Example

The following method defines a comparator that evaluates strings based on their length (rather than their natural lexicographic order).

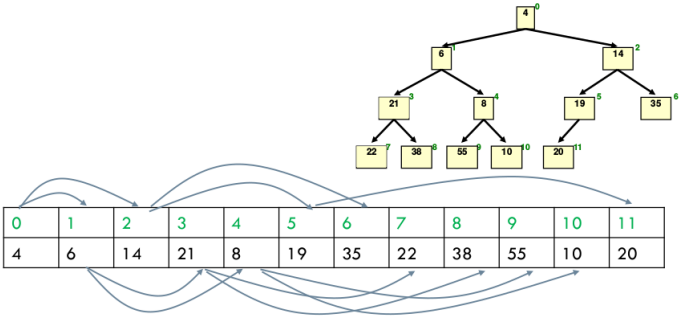
```
public class StringLengthComparator implements Comparator<String> {
   /** Compares two strings according to their lengths. */
   public int compare(String a, String b) {
      if (a.length() < b.length()) return -1;
      else if (a.length() == b.length()) return 0;
      else return 1;
   }
}
```

PQs as Linked Lists

- We can implement PQs using linked lists in two ways:
  - With unsorted list:
    - Each time a key-value pair is added to the priority queue, via the `insert(k, v)` method, we add that entry to the front of the list. This takes $O(1)$ time.
    - However, when we remove elements via the `removeMin()` method, because the entries are not sorted, we must inspect all entries to find one with minimal key. This takes $O(n)$ time.
  - With sorted list:
    - Inserting entries with the `insert(k, v)` method into a sorted list requires that we scan through the entire list until we find the appropriate position. This takes $O(n)$ time.
    - Removing entries, however, is straightforward given the knowledge that the first element has the minimal key. This takes $O(1)$ time.
- In either cases, $O(n^2)$ is required to process n elements.
- Can we do better?

Implementing PQs with a Heap

- Using heap, we can perform update operations (insert and delete) in time proportional to its height ($O(lgn)$).
- We use the ArrayList (from `java.util`) data structure to store key-value pairs as entries in the heap.
- We saw there are two types of heaps: max– and min– heaps. For PQs, we use the min–heap.
- This is due to the fact that the least (highest priority) element of any heap is found at the root of that heap.
- Elements of the heap are stored in the array in order, going across each level from left to right, top to bottom



Note that, $Parent(i) = \lfloor (i-1)/2 \rfloor$, $Left(i) = 2 * i + 1$, $Right(i) = 2 * i + 2$.

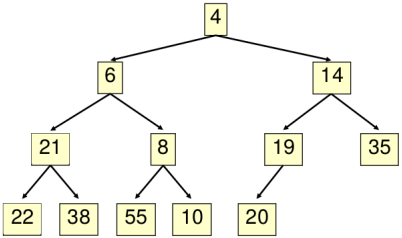
Implementing PQs with a Heap

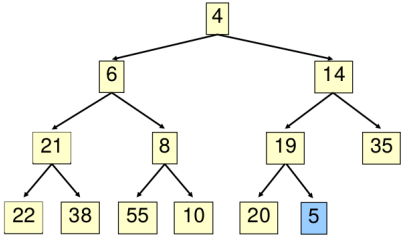
```
import java.util.ArrayList;
public class PriorityQueue<K extends
    Comparable<K>, V>{
  private static class PQEntry<K, V>
      implements Entry<K, V>{
    private K k;
    private V v;

    public PQEntry(K key, V val){
      k = key;
      v = val;
    }
    // methods of the Entry interface
    public K getKey(){return k;}
    public V getValue(){return v;}

    public void setKey(K key){k = key;}
    public void setValue(V val){ v = val; }
  }

  private ArrayList<Entry<K, V>> heap = new
      ArrayList<>();

  public int size() { return heap.size(); }
  public boolean isEmpty(){ return size() == 0; }
  public Entry<K,V> min() {
    if (heap.isEmpty()) return null;
    return heap.get(0);
  }
  public Entry<K,V> insert(K key, V value) {...}
  public Entry<K,V> removeMin() {...}
}
```

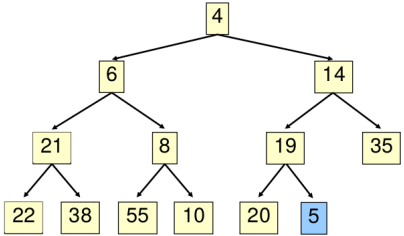
insert(k, v)

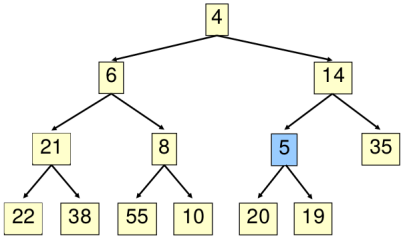
- Put the new element at the end of the array. If this violates heap order because it is smaller than its parent, swap it with its parent.
- Continue swapping it up until it finds its rightful place.
- The heap invariant is maintained!
- The upward movement of the newly inserted entry by means of swaps is conventionally called **up-heap bubbling**.

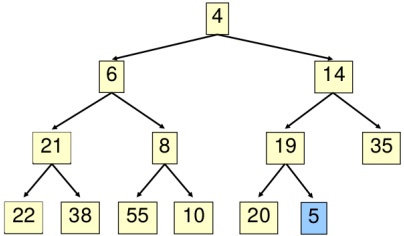
Example: Insert 5.

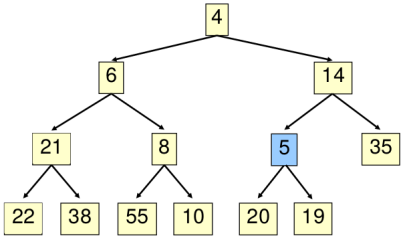


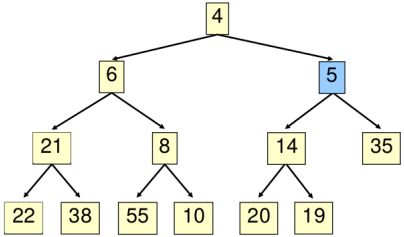

4
6
14
21
8
19
35
22
38
55
10
20
5

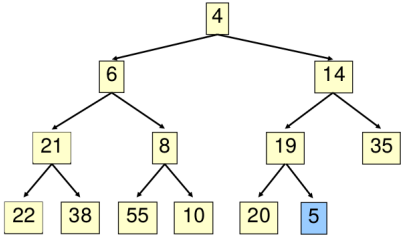

4
6
14
21
8
19
35
22
38
55
10
20
5

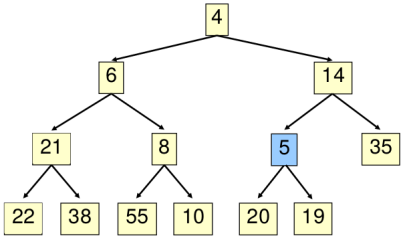

4
6
14
21
8
5
35
22
38
55
10
20
19

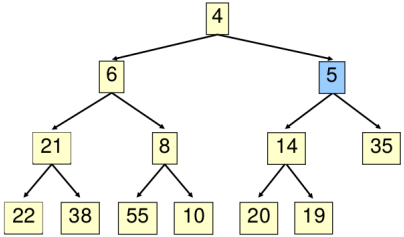

4
6
14
21
8
19
35
22
38
55
10
20
5

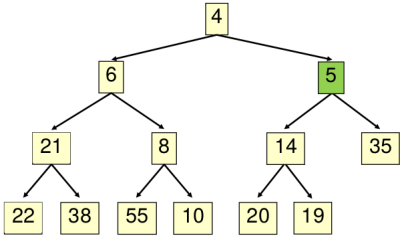

4
6
14
21
8
5
35
22
38
55
10
20
19

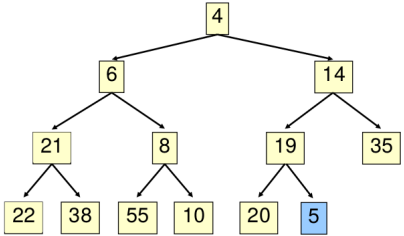

4
6
5
21
8
14
35
22
38
55
10
20
19

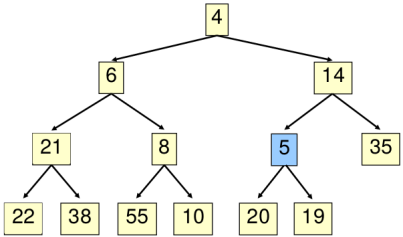

4
6
14
21
8
19
35
22
38
55
10
20
5

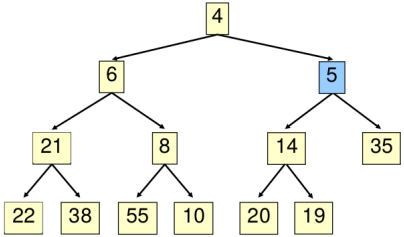

4
6
14
21
8
5
35
22
38
55
10
20
19

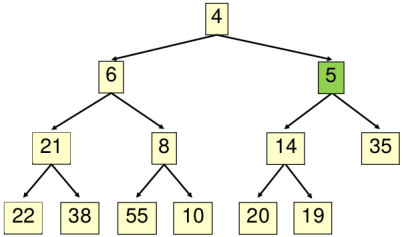

4
6
5
21
8
14
35
22
38
55
10
20
19


4
6
5
21
8
14
35
22
38
55
10
20
19


4
6
14
21
8
19
35
22
38
55
10
20
5


4
6
14
21
8
5
35
22
38
55
10
20
19


4
6
5
21
8
14
35
22
38
55
10
20
19


4
6
5
21
8
14
35
22
38
55
10
20
19

insert(k, v)

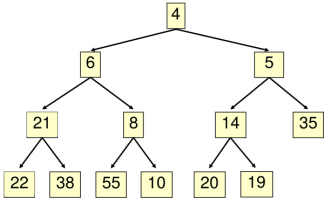
```
public Entry<K,V> insert(K key, V value) {
    Entry<K,V> newest = new PQEntry<>(key, value);
    heap.add(newest); // add to the end of the list
    upheap(heap.size() - 1); // upheap newly added entry
    return newest;
}
private void upheap(int j) {
    if (j==0) return;
    int parent = (j-1)/2;
    if (heap.get(j).getKey().compareTo(heap.get(parent).getKey()) >= 0)
        return;
    swap(j, parent);
    upheap(parent);
}
```

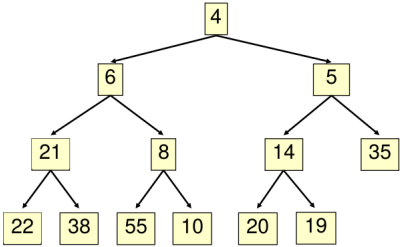
Running Time: $O(lgn)$

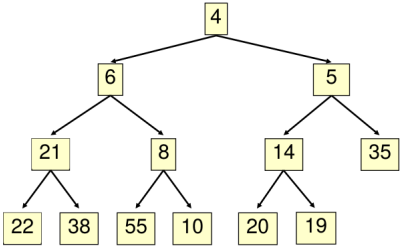
removeMin()

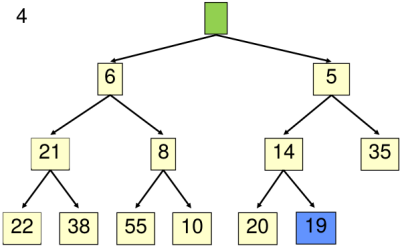
- Remove the least element – it is at the root. This leaves a hole at the root – fill it in with the last element of the array
- If this violates heap order because the root element is too big, swap it down with the smaller of its children.
- Continue swapping it down until it finds its rightful place
- The heap invariant is maintained!
- This downward swapping process is called **down-heap bubbling**.

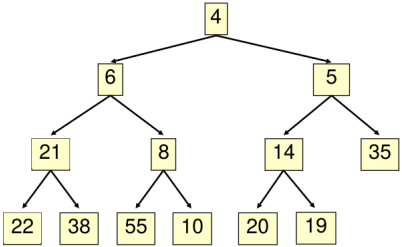
Example

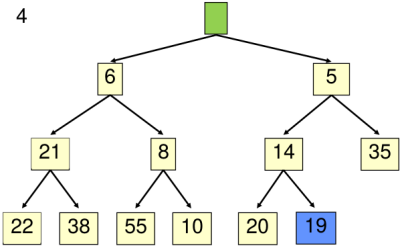


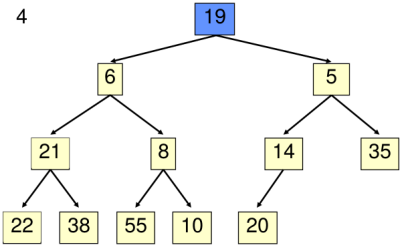

4
6
5
21
8
14
35
22
38
55
10
20
19

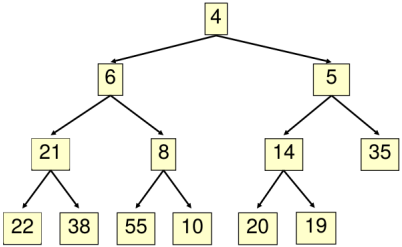

4
6
5
21
8
14
35
22
38
55
10
20
19

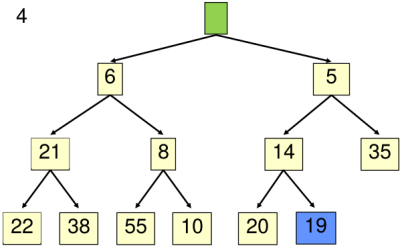

4
6
5
21
8
14
35
22
38
55
10
20
19

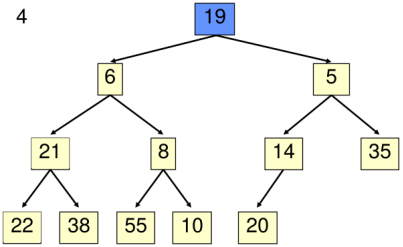

4
6
5
21
8
14
35
22
38
55
10
20
19

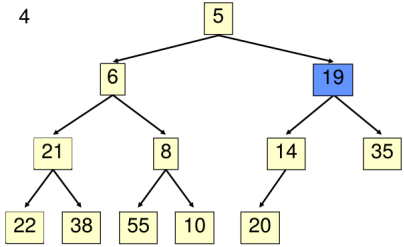

4
6
5
21
8
14
35
22
38
55
10
20
19


4
19
6
5
21
8
14
35
22
38
55
10
20

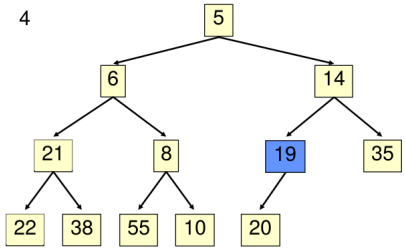

4
6
5
21
8
14
35
22
38
55
10
20
19

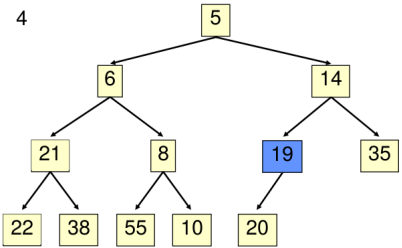

4
6
5
21
8
14
35
22
38
55
10
20
19

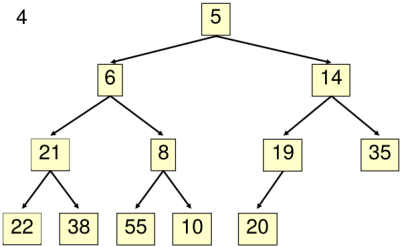

4
19
6
5
21
8
14
35
22
38
55
10
20

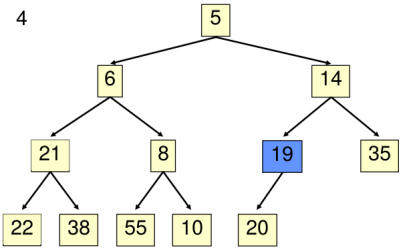

4
5
6
19
21
8
14
35
22
38
55
10
20

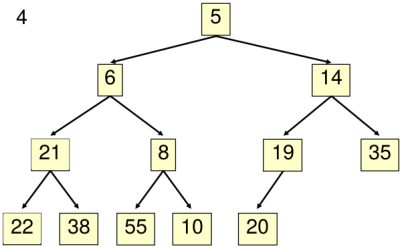
4




4
5
6
14
21
8
19
35
22
38
55
10
20


4
5
6
14
21
8
19
35
22
38
55
10
20


4
5
6
14
21
8
19
35
22
38
55
10
20


4
5
6
14
21
8
19
35
22
38
55
10
20

removeMin()

```
public Entry<K,V> removeMin() {
    if (heap.isEmpty()) return null;
    Entry<K,V> answer = heap.get(0);
    swap(0, heap.size() - 1); // put minimum item at the end
    heap.remove(heap.size() - 1); // and remove it from the list;
    downheap(0); // then fix new root
    return answer;
}
private void downheap(int j) {
    int child = 2*(j + 1); // start with right child
    if (child >= size() || heap.get(child - 1).getKey().compareTo(heap.get(child).getKey()) < 0)
        child -= 1;
    if(child >= heap.size()) return;
    if(heap.get(j).getKey().compareTo(heap.get(child).getKey()) <= 0)
    return;
    swap(j, child);
    downheap(child);
}
```

`removeMin()` running Time: $O(lgn)$

In general, to process n elements, $O(nlgn)$ time is required.

PQ Implementation with Single Parameter

```
public class PriorityQueue<K extends Comparable<K>> {
    private ArrayList<K> heap = new ArrayList<>();
    public Priorityqueue(){}
    public int size(){ return heap.size(); }
    public boolean isEmpty(){ return heap.isEmpty(); }
    private void swap(int i, int j){
        K temp = heap.get(i);
        heap.set(i, heap.get(j));
        heap.set(j, temp);
    }
    public K min(){
        if(isEmpty()) return null;
        return heap.get(0);
    }

    public K removeMin(){
        if(isEmpty()) return null;
        swap(0, heap.size()-1);
        K newest = heap.remove(heap.size()-1);
        downheap(0);
        return newest;
    }

    private void downheap(int j){
        int child = 2*j + 2;
        if(child > size()-1 ||
                heap.get(child).compareTo(heap.get(child-1)) > 0)
            child--1;
        if(child > size()-1) return;
        if(heap.get(j).compareTo(heap.get(child)) <= 0) return;
        swap(j, child);
        downheap(child);
    }

    public void insert(K k){
        heap.add(k);// add at the end
        upheap(heap.size()-1);

    }

    private void upheap(int j){
        if(j == 0) return; // root
        int parent = (j-1)/2;
        if(heap.get(parent).compareTo(heap.get(j)) <= 0)
            return;
        swap(j, parent);
        upheap(parent);
    }
}
```

The java.util.PriorityQueue<E> API

```
boolean add(E e) {...} //insert an element (insert)
void clear() {...} //remove all elements
E peek() {...} //return min element without removing (null if empty)
boolean offer(E e) // Inserts the specified element into this priority queue.
E poll() {...} //remove min element (extract)
//(null if empty)
int size() {...}
```

Priority Queue Applications

- In operating systems for load balancing (also in servers) and interrupt handling.
- Artificial intelligence: A* search algorithm.
- In data compression such as Huffman Coding.
- In graph algorithms: Dijkstra's Shortest Path Algorithm, Prim's algorithm etc.