Sorting Algorithms

Part 2

Quicksort

- Like mergesort, Quicksort is also based on the *divide-and-conquer* paradigm.
- But it uses this technique in a somewhat opposite manner, as all the hard work is done *before* the recursive calls.
- It works as follows:
  1. First, it partitions an array into two parts,
  2. Then, it sorts the parts independently,
  3. Finally, it combines the sorted subsequences by a simple concatenation.

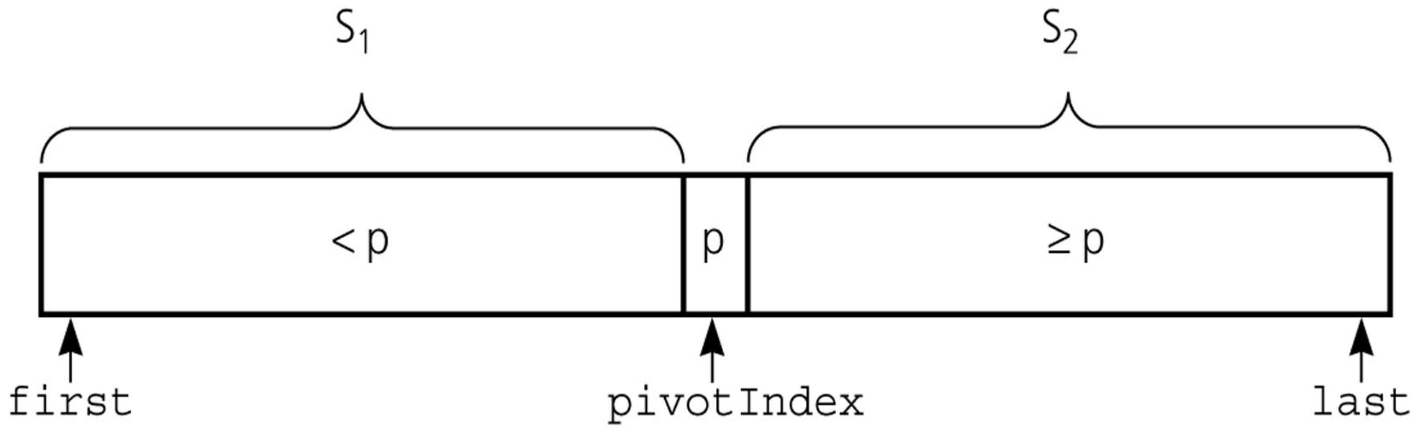
Quicksort (cont.)

The quick-sort algorithm consists of the following three steps:

1. ***Divide***: Partition the list.
   - To partition the list, we first choose some element from the list for which we hope about half the elements will come before and half after. Call this element the ***pivot***.
   - Then we partition the elements so that all those with values less than the pivot come in one sublist and all those with greater values come in another.
2. ***Recursion***: Recursively sort the sublists separately.
3. ***Conquer***: Put the sorted sublists together.

Partition

- Partitioning places the pivot in its correct place position within the array.



- Arranging the array elements around the pivot p generates two smaller sorting problems.
  - sort the left section of the array, and sort the right section of the array.
  - when these two smaller sorting problems are solved recursively, our bigger sorting problem is solved.

Partition – Choosing the pivot

- First, we have to select a pivot element among the elements of the given array, and we put this pivot into the first location of the array before partitioning.
- Which array item should be selected as pivot?
  - Somehow we have to select a pivot, and we hope that we will get a good partitioning.
  - If the items in the array arranged randomly, we choose a pivot randomly.
  - We can choose the first or last element as a pivot (it may not give a good partitioning).
  - We can use different techniques to select the pivot.

Quicksort

```
void quicksort(int D[], int left, int right) {

int k, j, q, temp;

   //partition the array into two parts
   k = left;
   j = right;

   q = D[(left+right)/2];   //pivot
```

Quicksort (cont.)

```
   do{
      while ((D[k] < q) && (k < right))
         k++;
      while ((D[j] > q) && (j > left))
         j--;
      if (k <= j) { //exchange D[k] & D[j]
         temp = D[k];
         D[k] = D[j];
         D[j] = temp;
         k++;    j--;
      }
   }while(k <= j);

// Sort each part using quicksort
   if (left < j)
      quicksort(D, left, j);
   if (k < right)
      quicksort(D, k, right);
 }
```

Quicksort – Analysis

Worst Case: (assume that we are selecting the min or max element as pivot)

- The pivot divides the list of size n into two sublists of sizes 0 and n-1.

- The number of key comparisons

  = n-1 + n-2 + ... + 1

  = $\mathbf{n^2/2 - n/2} \Rightarrow \mathbf{O(n^2)}$

- The number of swaps =

  = n-1 + n-1 + n-2 + ... + 1

  = $\mathbf{n^2/2 + n/2 - 1} \Rightarrow \mathbf{O(n^2)}$

- So, Quicksort is **$O(n^2)$** in worst case.

Quicksort: Example

7	3	5	8	6	9	4	15	2	Original List

left = 0
k = 0

Pivot = 6

right = 8
j = 8

Since 7 > 2, exchange 7 and 2, increment k, decrement j

2	3	5	8	**6**	9	4	15	**7**

k

j

Since 8 > 4, exchange 8 and 4, increment k, decrement j

2	3	5	**4**	**6**	9	**8**	15	**7**

k

j

Since 6 >= 6, exchange 6 and 6, increment k, decrement j

Quicksort: Example

2	3	5	**4**	**6**	9	**8**	15	**7**

j k

Since k > j, partition is over, sort each part using quicksort

2	3	5	4	**6**	9	8	15	7

quicksort(D, left, j) quicksort(D, k, right)

Quicksort – Analysis

- Quicksort is **$O(n*\log_2 n)$** in the best case and average case.
- Quicksort is slow when the array is sorted and we choose the first element as the pivot; or the minimum or the maximum value is chosen as the pivot → **$O(n^2)$**

Quicksort – Analysis

Advantage of Quicksort:

- Although the worst case behavior is not so good, its average case behavior is much better than its worst case.
  - So, Quicksort is one of best sorting algorithms using key comparisons.

Disadvantage of Quicksort:

- Because of recursion, it uses program stack too much.→ increases memory requirement.

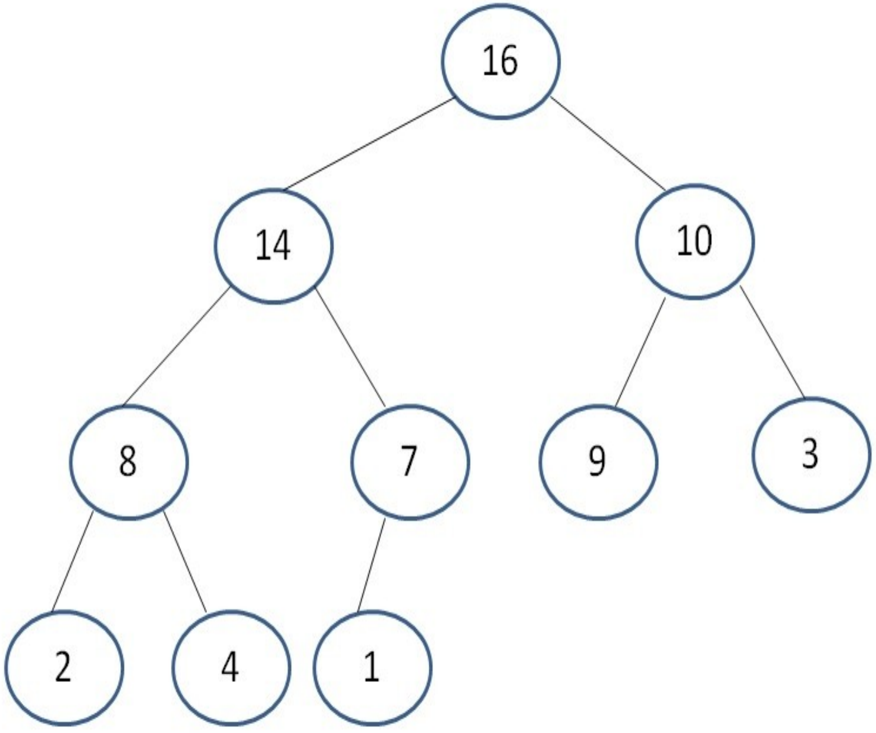
Heap Sort

- Uses «binary max heap» data structure to sort an array of data values.

Binary Max Heap Property:

16	14	10	8	7	9	3	2	4	1

- Each node has at most 2 children.
- The root has the max. valued element.
- The value stored in each node must be greater than or equal to the values stored in its children.



Heap Structure

- Each node of the tree corresponds to an element of the array,
- The tree is completely filled on all levels except the lowest, which is filled from the left up to a point.

Heap Sort Algorithm

- Let D be an n element array to be sorted.

1. Build a binary max heap over array D. So, the root has the largest element.
2. Exchange the values in the root node and in the last element of the array. So, the largest value is stored at the end of the array.
3. Build another binary max heap by using the first n-1 elements in the array.
4. Repeat steps 2 and 3 until the array becomes sorted.

Functions Used in Heap Sort

```
// Index of the left child of node i

int left(int i){
   return(2*i+1);
}

// Index of the right child of node i

int right(int i){
   return(2*i+2);
}
```

Functions Used in Heap Sort (cont.)

```
// Heap size is a global variable
int heap_size;   // index of the last element

void heapify(int D[], int i){
   int left_child, right_child, max, temp;
   left_child = left(i);
   right_child = right(i);
   // find the max of nodes left, right, and i
   if ((left_child <= heap_size) &&
       (D[left_child] > D[i]))
      max = left_child;
   else
      max = i;
```

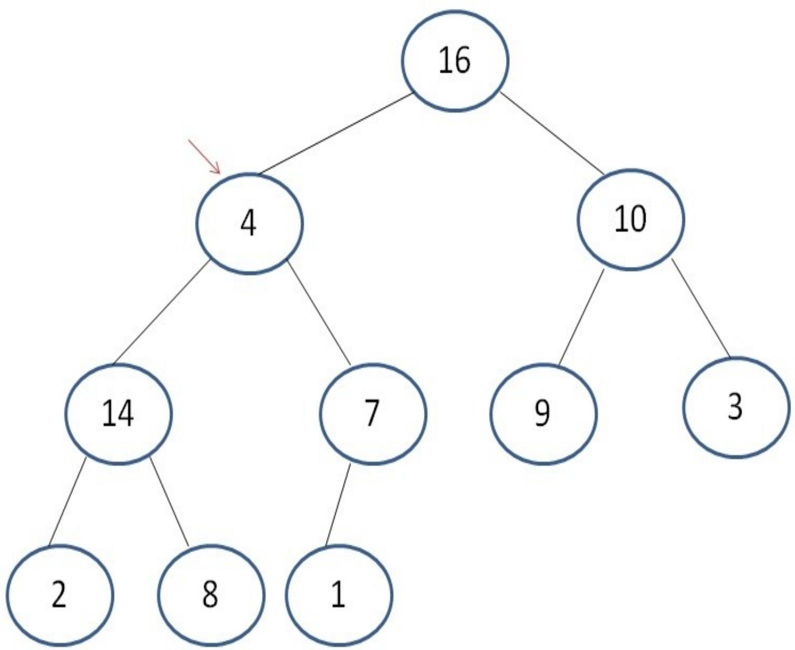
Functions Used in Heap Sort (cont.)

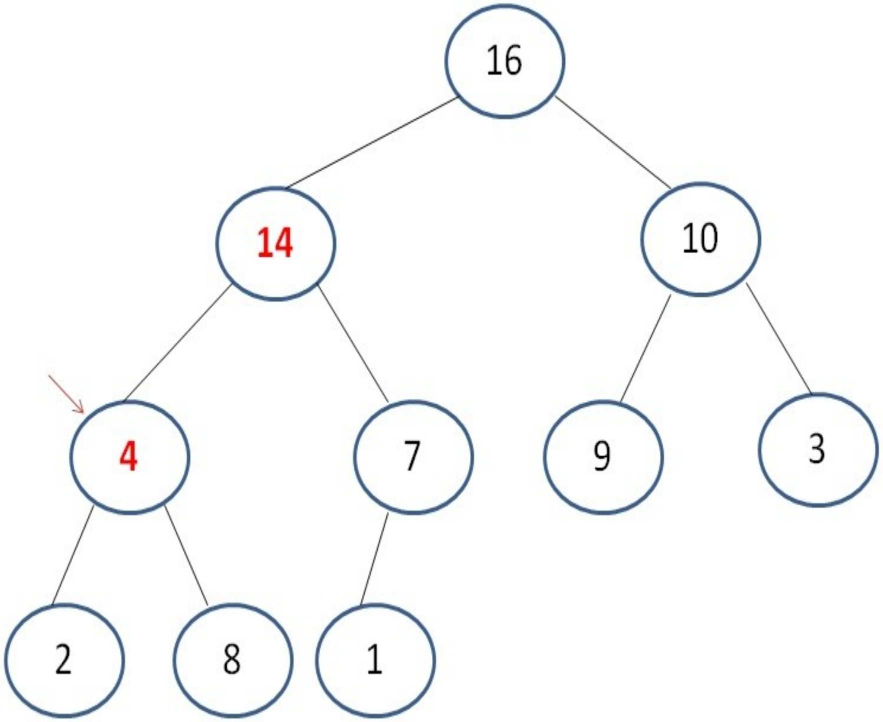
```
    if ((right_child <= heap_size) &&
         (D[right_child] > D[max]))
           max = right_child;

// if max is not the i th node, exchange

    if (max != i){
        temp = D[max];
        D[max]= D[i];
        D[i] = temp;
        heapify(D, max);
    }
}
```

Heapify: Example

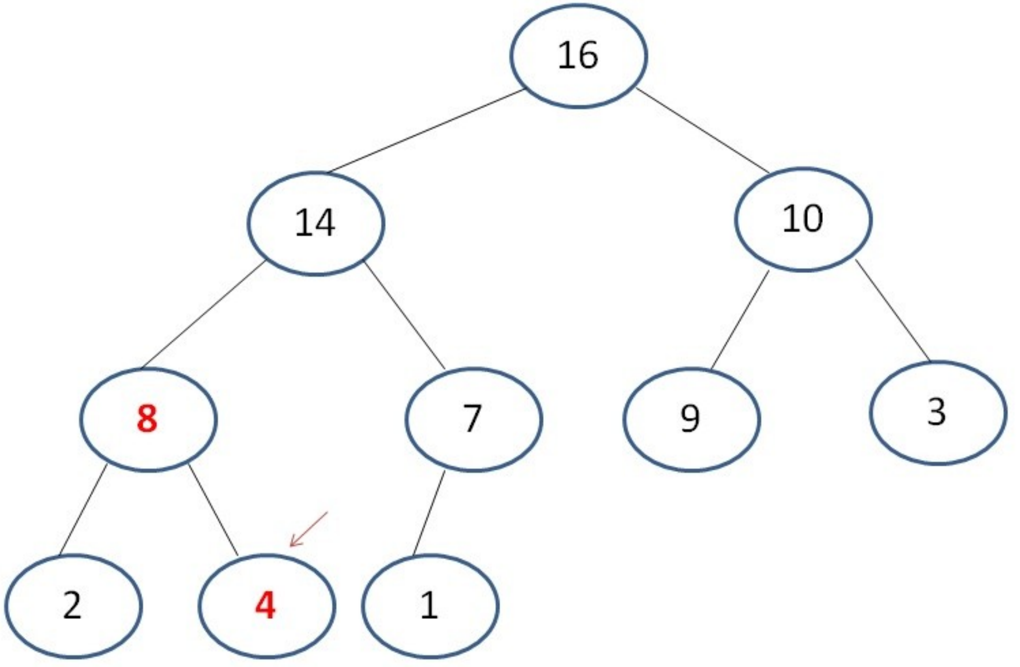
0	1	2	3	4	5	6	7	8	9
16	4	10	14	7	9	3	2	8	1

heapify(D,1)





Heapify: Example (cont.)



16	14	10	8	7	9	3	2	4	1

Functions Used in Heap Sort (cont.)

```
void build_heap(int D[], int n){
int i;
heap_size = n-1;

for (i = (n-1)/2; i >= 0; i--)
   heapify(D,i);
}
```

4	1	3	2	16	9	10	14	8	7

Array D

build_heap(D,10); → heapify(D,4); heapify(D,3); heapify(D,2);
heapify(D,1); heapify(D,0);

16	14	10	8	7	9	3	2	4	1

Array D after build_heap

Heap Sort Function

```
void heapsort(int D[], int n){
    int i, temp;
    build_heap(D,n);
    for (i = n-1; i >= 1; i--){
    // exchange the root with the ith
element
        temp = D[i];
        D[i] = D[0];
        D[0] = temp;
        heap_size--;
        heapify(D,0);
    }
}
```

Heap Sort -- Analysis

- Running time of heapify() → **O($\log_2 n$)**
- Running time of build_heap() → **O($n\log_2 n$)**
- Running time of heapsort() → **O($n\log_2 n$)**

Advantages of heapsort:

- The best sorting algorithm in this class.
- Runs in place (no extra array is needed like mergesort)
- The running time does not change whether the array D is sorted in advance or not.

Comparison of Sorting Algorithms

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	$n * \log n$	$n * \log n$
Quicksort	n^2	$n * \log n$
Radix sort	n	n
Treesort	n^2	$n * \log n$
Heapsort	$n * \log n$	$n * \log n$