Data Structures

Sorting

CS284

Objectives

- To learn how to implement the following sorting algorithms:
  - selection sort
  - bubble sort
  - insertion sort
  - heapsort
  - quicksort
- To understand the differences in performance of these algorithms, and which to use for small, medium arrays, and large arrays

Heapsort

Quicksort

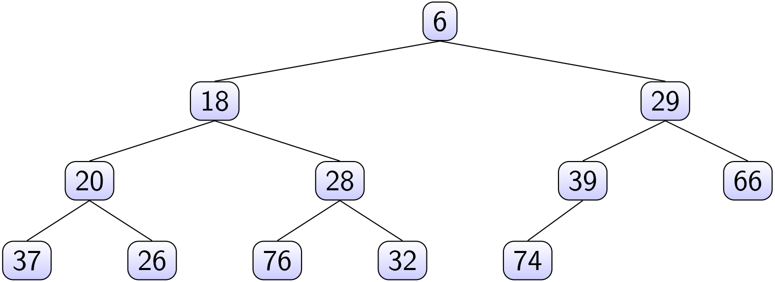
Comparison of Algorithms

Heapsort

- Heapsort does not require any additional storage
- As its name implies, heapsort uses a heap to store the array
  - When used as a priority queue, a heap maintains a smallest value at the top
  - Naive heapsort:
    - place an array's data into a heap,
    - then remove each heap item and move it back into the array

Heaps and Priority Queues

- A heap is a complete binary tree with the following properties
  - The value in the root is the smallest item in the tree
  - Every subtree is a heap



Naive Version of a Heapsort Algorithm (MinHeap)

- This version of the algorithm requires n extra storage locations

```
// Create a new array for the MinHeap, and insert from the original
// array into the heap array.
i=0
while (heap is not empty) {
   Remove next item from the heap
   Insert it back into the array at position i
   i++
}
```

- Tracing an example

0	1	2	3	4	5	6	7
15	20	30	45	50	60	80	90

An in-place solution with MaxHeap

- We can do better in terms of space usage
- We can build a heap so that each parent node value is not less than its children (maxHeap)
- Then,
  - move the top item to the bottom of the heap
  - reheap, ignoring the item moved to the bottom
- If we implement the heap as an array,
  - each element removed will be placed at end of the array, and
  - the heap part of the array decreases by one element

Algorithm for In-Place Heapsort

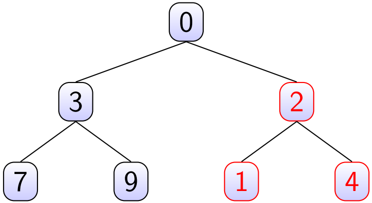
```
Heapify the input array
while (h is not empty) {
  Remove the first item h by swapping it with the last item in h
  Restore the heap property on h
}
```

0	1	2	3	4	5	6
0	3	2	7	9	1	4

Example: sorting the array 0, 3, 2, 7, 9, 1, 4

Step 1: Heapify the array

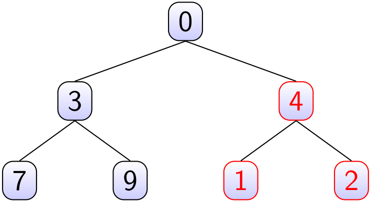
```
for (int i = this.heap.length - 1; i >= 0; i --) {
    find i's parent and sibling;
    swap parent with the larger sibling;
    heapify down the larger sibling;
}
```



0	1	2	3	4	5	6
0	3	2	7	9	1	4

Step 1: Heapify the array

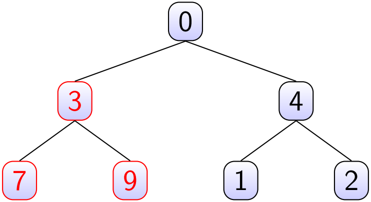
```
for (int i = this.heap.length - 1; i >= 0; i --) {
    find i's parent and sibling;
    swap parent with the larger sibling;
    heapify down the larger sibling;
}
```



0	1	2	3	4	5	6
0	3	4	7	9	1	2

Step 1: Heapify the array

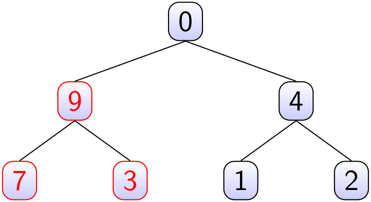
```
for (int i = this.heap.length - 1; i >= 0; i --) {
    find i‘s parent and sibling;
    swap parent with the larger sibling;
    heapify down the larger sibling;
}
```



0	1	2	3	4	5	6
0	3	4	7	9	1	2

Step 1: Heapify the array

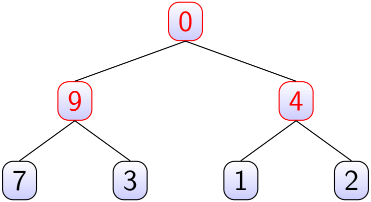
```
 for (int i = this.heap.length - 1; i >= 0; i --) {
    find i‘s parent and sibling;
    swap parent with the larger sibling;
    heapify down the larger sibling;
}
```



0	1	2	3	4	5	6
0	9	4	7	3	1	2

Step 1: Heapify the array

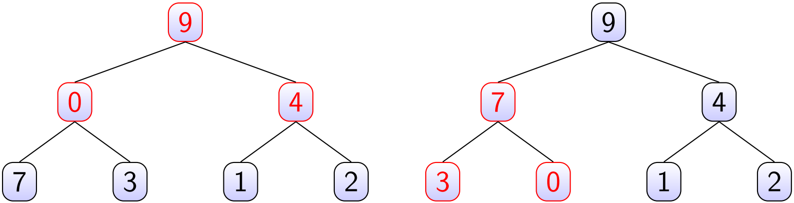
```
for (int i = this.heap.length - 1; i >= 0; i --) {
    find i's parent and sibling;
    swap parent with the larger sibling;
    heapify down the larger sibling;
}
```



0	1	2	3	4	5	6
0	9	4	7	3	1	2

Step 1: Heapify the array

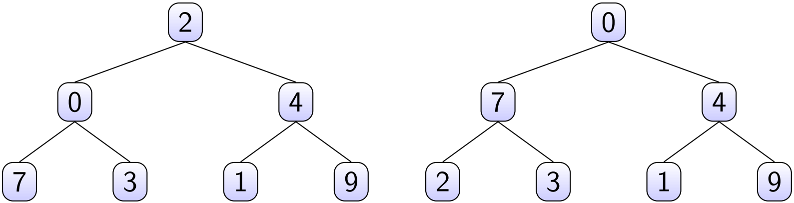
```
   for (int i = this.heap.length - 1; i >= 0; i --) {
    find i's parent and sibling;
    swap parent with the larger sibling;
    heapify down the larger sibling;
}
```



0	1	2	3	4	5	6
9	7	4	0	3	1	2

Step 2: Poll and place the next largest element

```
for (int i = this.heap.length - 1; i >= 0; i --) {
    int next_element = this.peek();
    this.poll();
    this.heap[i] = next_element;
}
```



0	1	2	3	4	5	6
0	7	4	2	3	1	9

Heapsort - Code

```java
/** heap sort: (1) heapify the input array
* (2) for i= length...0
* remove the i-th smallest element, and
* append it to the length - i-th position
* @param array
*/
private void heapSort(int[] array) {
  this.set_array(array);
  this.heapifyAll();
  for (int i = this.heap.length - 1; i >= 0; i --) {
      int next_element = this.peek();
      this.poll();
      this.heap[i] = next_element;
  }
  this.size = this.heap.length;
}
```

Heapsort - Code

```
private void heapifyAll() {
int[] visited = new int[this.heap.length];
for (int i = this.heap.length - 1; i >= 0; i --) {
    int parentIdx = this.getParentIndex(i);
    if (parentIdx == i)
        continue;
    if (visited[i] == 1) continue;
    int leftChildIdx = this.getLeftChildIndex(parentIdx);
    int rightChildIdx = this.getRightChildIndex(parentIdx);
```

Heapsort - Code

```
  if (this.heap[leftChildIdx] >
  this.heap[rightChildIdx]) {
      swap(leftChildIdx, parentIdx);
      heapifyDown(leftChildIdx);
  }
  else {
      swap(rightChildIdx, parentIdx);
      heapifyDown(rightChildIdx);
  }
  visited[leftChildIdx] = 1;
  visited[rightChildIdx] = 1;
  }
}
```

Heapsort - Code

```
private void heapifyDown(int index) {

while (hasLeftChild(index)) {
  int largerChildIndex = getLeftChildIndex(index);

  if (hasRightChild(index) &&
  rightChild(index) > leftChild(index)) {
    largerChildIndex = getRightChildIndex(index);
  }

  if (heap[index] < heap[largerChildIndex]) {
    swap(index, largerChildIndex);
  } else {
    break;
  }
  index = largerChildIndex;
}
}
```

Heapsort - Analysis

- Heapify: $O(n \log n)$
- Poll and place: $O(n \log n)$
- Total time complexity: $O(n \log n)$

Run the demo for heapsort and quicksort

Heapsort

Quicksort

Comparison of Algorithms

Quicksort

- Developed in 1962
- Selects a specific value called a pivot and rearranges the array into two parts (called partitioning)
  - all the elements in the left subarray are less than or equal to the pivot
  - all the elements in the right subarray are larger than the pivot
- The pivot is placed between the two subarrays
- The process is repeated until the array is sorted

Quicksort by Example

- We'll look at an example
- We will use a simple partitioning strategy
  - the pivot is chosen to be the first element in the current subarray to be partitioned

0	1	2	3	4	5	6	7	8
44	75	23	43	55	12	64	77	33

Divide-and-Conquer using Pivot

```
public void quick_sort(E[] a) {
  recursive_quicksort(a, 0, a.length - 1);
}

public void recursive_quicksort(E[] a, int first, int last)
  int pivIdx = partition(a, first, last);
  if (first < pivIdx - 1)
      recursive_quicksort(a, first, pivIdx - 1);
  if (pivIdx + 1 < last)
      recursive_quicksort(a, pivIdx + 1, last);
}
```

Quicksort Partition: Execution trace

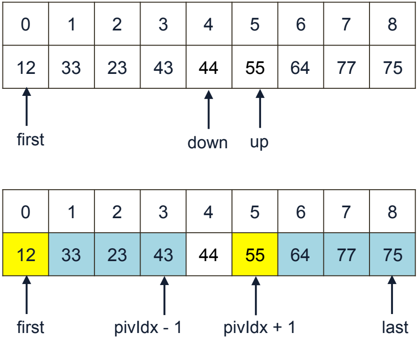
```
int pivIdx = partition(a, first, last);
```

0	1	2	3	4	5	6	7	8
44	75	23	43	55	12	64	77	33
first	up							down

0	1	2	3	4	5	6	7	8
44	33	23	43	55	12	64	77	75
first				up	down			

Quicksort Partition: Execution trace

```
int pivIdx = partition(a, first, last);
```



quickSort(a, first, pivIdx - 1); quickSort(a, pivIdx + 1, last);

Quicksort Partition: Code

```
public int partition(E[] a, int first, int last) {
   E pivot = a[first];
   int up = first;
   int down = last;
   do {
        while ((up < last) && (up<last)
        && (pivot.compareTo(a[up])>=0)) {
           up++; }
        while (down > first && pivot.compareTo(a[down]) < 0) {
          down--;
        }
         if (up < down) { // if up is to the left of down.
             swap(a, up, down);
        }
     } while (up < down); // Repeat while up is left of down.
   swap(a, first, down);
   return down;
}
```

Analysis of Quicksort

- If the pivot value is a random value selected from the current subarray, then statistically half of the items in the subarray will be less than the pivot and half will be greater
- If both subarrays have the same number of elements (best case), there will be $\log n$ levels of recursion

Analysis of Quicksort

- At each recursion level, the partitioning process involves moving every element to its correct position (n moves)
- Quicksort is $\mathcal{O}(n \log n)$
- The array split may not be the best case, i.e. 50-50
- An exact analysis is difficult (and beyond the scope of this class), but, the running time will be bounded by a constant * $n \log n$

Analysis of Quicksort (cont.)

- A quicksort will give very poor behavior if, each time the array is partitioned, a subarray is empty.
- In that case, the sort will be $\mathcal{O}(n^2)$
  - Prove this with an example
- Under these circumstances, the overhead of recursive calls and the extra run-time stack storage required by these calls makes this version of quicksort a poor performer relative to the quadratic sorts
- We'll discuss a solution later

Revised Partition Algorithm

- Quicksort is $\mathcal{O}(n^2)$ when each split yields one empty subarray, which is the case when the array is presorted
- A better solution is to pick the pivot value in a way that is less likely to lead to a bad split
  - Use three references: first, middle, last
  - Select the median of the these items as the pivot

Heapsort

Quicksort

Comparison of Algorithms

Comparison of Sorting

	Number of comparisons		
	Best	Average	Worst
Selection sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Bubble sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Insertion sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
Shell sort	$\mathcal{O}(n^{7/6})$	$\mathcal{O}(n^{5/4})$	$\mathcal{O}(n^2)$
Merge sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Heapsort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Quicksort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$