Computer Science II

Professor Tom Ellman
Lecture 19
Sorting

- Given an array of data:
  - E.g. an array of integers
  - E.g., an array of strings.

- In which pairs of data items can be compared:
  - E.g., comparing integers by size.
  - E.g., comparing strings by lexicographic order.

- Put the array of data in order.
Sorting Algorithms

• Naïve algorithms:
  – BubbleSort.
  – SelectionSort.
  – InsertionSort.

• Fast algorithms:
  – MergeSort.
  – Quick Sort.
  – Heap Sort.
BubbleSort

• Repeatedly find the largest of the unsorted items:
  – Find the largest item and put it at location N-1.
  – Find the second largest and put it at N-2.
  – Etc.

• On each iteration, find the largest of the unsorted items as follows:
  – If data[0] > data[1], then swap data[0] & data[1].
  – Etc.
public <T extends Comparable<T>> void bubbleSort(T data[]) {
    for (int numSorted = 0; numSorted < data.length; numSorted++) {
        for (int index = 1; index < data.length-numSorted; index++) {
            if (data[index-1].compareTo(data[index]) > 0)
                swap(data, index-1, index);
        }
    }
}
public <T> void swap(T data[], int i, int j) {
    T temp;
    temp = data[i];
    data[i] = data[j];
    data[j] = temp;
}
SelectionSort

• Repeatedly find the largest of the unsorted items:
  – Find the largest item and put it at location N-1.
  – Find the second largest and put it at N-2.
  – Etc.

• On each iteration, find the largest of the unsorted items as follows:
  – Scan the unsorted part of the data array.
  – Find the index of the largest unsorted item.
  – Swap the largest unsorted item with the last unsorted item.
public <T extends Comparable<T>> void selectionSort(T data[]) {
    for (int numUnsorted = data.length; numUnsorted > 0; numUnsorted--) {
        int max = 0;
        for (int index = 1; index < numUnsorted; index++)
            if (data[index].compareTo(data[max]) > 0) max = index;
        swap(data, max, numUnsorted-1);
    }
}
InsertionSort

• Put successively larger prefixes of the array in order:
  – Put the first 1 item in order.
  – Put the first 2 items in order.
  – Etc.

• On the $i$th iteration, move the item at the $i$th location into its position $k$ among items at 0…$i$-1, after shifting items at $k$…$i$-1 one place over.
public <T extends Comparable<T>> void insertionSort(T data[]) {
    for (int numSorted = 1; numSorted < data.length; numSorted++) {
        T temp = data[numSorted];
        int index = numSorted;
        while (index > 0) {
            if (temp.compareTo(data[index-1]) < 0)
                data[index] = data[index-1];
            else break;
            index--;
        }
        data[index] = temp;
    }
}
Analysis of Naïve Sorting

• Let $N$ be the length of the array.
• Count comparisons or assignments.
• Main loop has $N$ iterations.
• The $i$th iteration requires $I$ steps in the worst case.
• $T(N) = 1 + 2 + \ldots + N$.
• $T(N) = N(N+1)/2$.
• $T(N) = \Theta(N^2)$. 
MergeSort

• Split the array into two equal sized parts.
• Recursively sort each part.
• Merge the two sorted parts into one sorted whole.
public <T extends Comparable<T>> void mergeSort(T[] data) {
    T[] temp = data.clone();
    mergeSortHelper(data, temp, 0, data.length);
}

private <T extends Comparable<T>>
void mergeSortHelper(T[] data, T[] temp, int low, int high) {
    int n = high-low;
    int middle = low + n/2;
    if (n < 2) return;
    mergeSortHelper(data, temp, low, middle);
    mergeSortHelper(data, temp, middle, high);
    merge(data, temp, low, middle, high);
}
private <T extends Comparable<T>>
void merge(T[] data, T[] temp, int low, int middle, int high) {
    int ri = low;
    int li = low;
    int hi = middle;
    while (li < middle && hi < high)
        if (data[li].compareTo(data[hi]) < 0) temp[ri++] = data[li++];
        else temp[ri++] = data[hi++];
    if (li < middle)
        do temp[ri++] = data[li++]; while (li < middle);
    else
        do temp[ri++] = data[hi++]; while (hi < high);
    for (int i=low; i<high; i++) data[i] = temp[i];
}
Analysis of MergeSort

- Let N be the length of the array.
- Count comparisons or assignments.
- A merge step that produces a subarray of length k requires $\Theta(k)$ comparisons or assignments.
- At the $i^{th}$ level of recursion, we call merge $2^i$ times, generating a subarray of length $N/2^i$ each time.
- Thus we do N comparisons or assignments at each level of recursion.
- We have at most $\log_2(N)$ levels of recursion.
- $T(N) = N \log_2(N)$.
- $T(N) = \Theta(N \log(N))$. 
QuickSort

• Choose an element in the array and call it the “pivot”.

• Partition the array into two parts:
  – Items less than the pivot.
  – Items greater than the pivot.

• Recursively sort each of the two parts.
public <T extends Comparable<T>> void quickSort(T[] data) {
    quickSortHelper(data, 0, data.length-1);
}

public <T extends Comparable<T>>
void quickSortHelper(T[] data, int low, int high) {
    int pivot;
    if (low >= high) return;
    pivot = partition(data, low, high);
    quickSortHelper(data, low, pivot-1);
    quickSortHelper(data, pivot+1, high);
}
public <T extends Comparable<T>>

int partition(T[] data, int left, int right) {
    while (true) {
        while (left < right && data[left].compareTo(data[right]) < 0) right--;
        if (left < right) swap(data, left++, right);
        else return left;
        while (left < right && data[left].compareTo(data[right]) < 0) left++;
        if (left < right) swap(data, left, right--);
        else return right;
    }
}
(Best Case) Analysis of QuickSort

• Let N be the length of the array.
• Count comparisons or assignments.
• A partition step that divides a subarray of length k requires $\Theta(k)$ comparisons or assignments.
• Assume (best case) that each partition divides the subarray into two equal size subsubarrays.
• At the $i$th level of recursion, we call partition $2^i$ times, dividing a subarray of length $N/2^i$ each time.
• Thus we do $N$ comparisons or assignments at each level of recursion.
• We have at most $\log_2(N)$ levels of recursion.
• $T(N) = N \log_2(N)$.
• $T(N) = \Theta(N \log(N))$. 
(Worst Case) Analysis of QuickSort

- Let N be the length of the array.
- Count comparisons or assignments.
- A partition step that divides a subarray of length k requires $\Theta(k)$ comparisons or assignments.
- Assume (worst case) that each partition divides a length k subarray into length 0 and length k-1 subsubarrays.
- At the $i$th level of recursion, we call partition once, dividing a subarray of length $N-i$ each time.
- Thus we do $N-i$ comparisons or assignments at the $i$th level of recursion.
- We have N levels of recursion.
- $T(N) = N + N-1 + N-2 + \ldots + 2 + 1 = \frac{N(N+1)}{2}$
- $T(N) = \Theta(N^2)$.
Heaps

- A complete binary tree.
- Partially orders data.
- Implements the `PriorityQueue` interface.
- Used in the `HeapSort` algorithm.
Structure of a Heap

Root

Objects greater than or equal to \textit{data}

Left Subheap

Right Subheap

Objects greater than or equal to \textit{data}
Example: \{0, 2, 3, 5, 7, 8, 9\}
Complete Binary Tree

- At most one level is incompletely filled.
- Missing nodes are to the right of present nodes.
- Can be implemented efficiently in an array.
Array Implementation of Heaps

• Store root at array index zero.
• Remaining nodes follow in *breadth-first* order.
• Index of left or right child easily computed from index of parents.
• Index of parent is easily computed from index of left or right child. (Note: Using integer division.)

```
   p
  /   \
2p+1  2p+2

   (c-1)/2
  / |
  c  (c-1)/2
```
Example: \{0, 2, 3, 5, 7, 8, 9\}

Indexes:

Data:

Array:
public interface PriorityQueue<T extends Comparable<T>> {  
    public boolean isEmpty();  
    public T insert(T item);  
    public T delete();  
    public T min();  
    public Iterator<T> iterator();  
}

Our Heap implementation will support this interface.
A heap will use an ArrayList to keep track of its data. This allows us to access data via indexing, while allowing for the container to grow as needed.
Use the `isEmpty` method of `ArrayList` to determine whether the heap is empty. Use the `get` method of `ArrayList` to access and return the element at index zero in the array. This will be the smallest element in the heap.
Inserting an Element into a Heap

1. Initially put the new element at the first free node position in the binary tree. (I.e., Put the new element at the end of the array.)
2. Percolate the new element up the tree. (I.e., While the new element is not the root and is smaller than its parent, swap it with its parent.)
Inserting a New Element (0) into a Heap
private T parentData(int index) { return data.get((index-1)/2); }

private T leftData(int index) { return data.get(2*index+1); }

private T rightData(int index) { return data.get(2*index+2); }

private int parentIndex(int index) { return (index-1)/2; }

private int leftIndex(int index) { return 2*index+1; }

private int rightIndex(int index) { return 2*index+2; }

Methods for easy access to parent and child indexes and data.
public T insert(T item) {
    int index = data.size();
    data.add(item);
    while (index > 0 && item.compareTo(parentData(index)) < 0) {
        data.set(index, parentData(index));
        index = parentIndex(index);
    }
    data.set(index, item);
    return item;
}
Deleting an Element from a Heap

1. Record the root data, to be returned.
2. Put the last tree element in the root position, and remove the last node from the tree.
3. Invoke the `heapify` helper method to restore the heap property.
4. Return the initial root data.
Deleting the Root Element (0) from a Heap
public T delete() {
    T item = data.get(0);
    data.set(0, data.get(data.size()-1));
    data.remove(data.size()-1);
    heapify(0);
    return item;
}
Heapify: Restoring the Heap Property

• If the root is smaller than both of its children, do nothing
• Otherwise:
  1. Swap the root with its smallest child.
  2. Recursively Heapify the modified subtree.
private void heapify(int index) {
    int last = data.size() - 1;
    if (leftIndex(index) > last) return;
    T item = data.get(index);
    if (rightIndex(index) > last) {
        if (item.compareTo(leftData(index)) <= 0)
            return;
        else {
            data.set(index, leftData(index));
            data.set(leftIndex(index), item);
            return;
        }
    }
    if (leftData(index).compareTo(rightData(index)) <= 0) {
        if (item.compareTo(leftData(index)) <= 0) return;
        else {
            data.set(index, leftData(index));
            data.set(leftIndex(index), item);
            heapify(leftIndex(index));
        }
    }
    else {
        if (item.compareTo(rightData(index)) <= 0) return;
        else {
            data.set(index, rightData(index));
            data.set(rightIndex(index), item);
            heapify(rightIndex(index));
        }
    }
}
Analysis of HeapSort

• Each heap operation takes $O(\log N)$ steps.
  – Insert: Percolating up the tree has at most one step for each level of the tree.
  – Delete: Heapify has at most one recursive call for each level of the tree.

• HeapSort calls $N$ inserts and $N$ deletes.

• HeapSort is $O(N\times \log N)$. 
public Iterator<T> iterator() {
    ArrayList<T> dataCopy = new ArrayList<T>();
    for (T item : data) {
        dataCopy.add(item);
    }
    Heap<T> heapCopy = new Heap(dataCopy);
    return new HeapIterator<T>(heapCopy);
}

First copy the ArrayList data into a new ArrayList dataCopy. Then use dataCopy to initialize a new Heap heapCopy. Finally, use heapCopy to initialize a HeapIterator.
public class HeapIterator<T extends Comparable<T>> implements Iterator<T> {
    private Heap<T> heap;

    HeapIterator(Heap<T> heap) {
        this.heap = heap;
    }

    public boolean hasNext() {
        return !heap.isEmpty();
    }

    public T next() {
        return heap.delete();
    }

    public void remove() {
    }
}