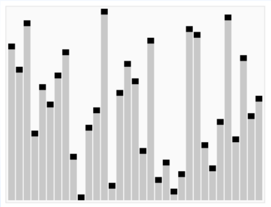
Data Structures and Algorithms

Semester 2 Sorting 4

Heapsort and Java Sorting



Heap Sort was invented by J. W. J Williams in 1964, `Communications of the ACM 7(6): 347–348`

Reading: Goodrich Chapter 11

Donald Knuth: The Art of Computer Programming, Volume 3: Sorting and Searching

Heap Sort Informal Algorithm

Heap sort builds on the fact that the max value in a complete binary tree can found, and the tree maintained, in O(log(n)) time.

1. Build a Heap from array to be sorted
2. Repeatedly remove the largest element, maintaining the heap using sift down

Heap Sort Informal Analysis

1. Build a Heap

Using the efficient constructor described in Geoff's lecture, this can be done in O(n) time

2. Repeatedly remove the largest element, maintaining the heap as you do so

Removing the largest element is O(log n). So removing n elements is O(n log n).

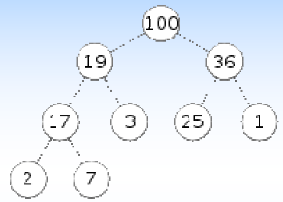
$$t(n) = a \cdot n + b \cdot n \log(n)$$

Heap sort is an improved version of Selection Sort that is more efficient at finding the next largest

Heap Revision: A Heap

Array Representation of a complete binary tree

A Heap is a Complete Binary Tree where all parents are equal to or bigger than their children



Array Representation									
	100	19	36	17	3	25	1	2	7

Children of an element in position p are in positions 2*p and 2*(p+1)

Note that a Heap is NOT a Binary Search Tree. There is no implied ordering between siblings or cousins and no implied sequence for an in-order traversal. Heap is an optimal implementation of a Priority Queue

Heap Revision: Sift Down

Given a Heap H, an integer k, and an integer m, where all the values in H[1…m] are in the correct place except H[k]

siftDown(H[], int k, int m)
  root:=k
  value:=H[k]
//While the root has more children
While root*2 ≤ m
  //Find the largest child: find the first child
    child:=root*2
  //If the child has a sibling and the child's value is less than its sibling's
    if child + 1 ≤ m and H[child] < H[child + 1] then
    // point to the right child instead
      child := child + 1
    //If out of order, swap
    if H[root] < H[child] then
      swap(H[root], H[child])
      root := child
    //Else, finished, return out of function
    else **return**

Sift Down Example

7	19	36	17	3	25	1	2	

```
root:=k
value:=H[k]
While root*2 <= m
    child:=root*2+1
    if child + 1 ≤ m and H[child] < H[child + 1]
        child := child + 1
    if H[root] < H[child] then
        swap(H[root], H[child])
        root := child
    else    return
```

siftDown(H,1):
root:=1
Loop 1: 2*1=2<8 **TRUE**
//Find the largest child
  child:=2 (VALUE 19)
  H[2]<H[3] **TRUE**
  child:=3
  H[1]<H[3] **TRUE**: **SWAP H[1],H[3]**

36	19	7	17	3	25	1	2	

root=3;
Loop 2: 3*2=6<8 **TRUE**
  child:=3*2=6
  H[6]<H[7] **FALSE**
  child:=6
  H[3]<H[6] **TRUE SWAP H[3],H[6]**

36	19	25	17	3	7	1	2	

root:=6
Loop 3: 2*6=12<8 **FALSE: TERMINATE**

Heap Revision: heapify

Build a Heap (In Place)

Informal Algorithm

- let T initially contain the list of elements to be heapified, and let the height of the underlying complete binary tree be *l*
- Start at the index of the last parent node, sift down this parent into its subtree.
- Repeat for all elements prior to the last parent

Heap Revision: heapify

Formal algorithm

heapify: converts Array T[1...n] into a Heap
heapify(T[1…n])
//start is assigned the index in T of the last parent node
  start:= floor[n/2]
  while start>0
//sift down the node at index start
    siftDown(T,start,n)
//repeat at previous parent node
    start:=start-1

Heap Revision: heapify

1	2	3	7	17	19	25	36	100

Example: size = 9

Start = floor(9/2)=4
siftDown(T,4,9)

1	2	3	100	17	19	25	36	7

Start =3
siftDown(T,3,9)

1	2	25	100	17	19	3	36	7

Start =2
siftDown(T,2,9)

1	100	25	2	17	19	3	36	7
1	100	25	36	17	19	3	2	7

Start =1
siftDown(T,1,9)

100	1	25	36	17	19	3	2	7
100	36	25	1	17	19	3	2	7
100	36	25	7	17	19	3	2	1

In Place Heap Sort: informal algorithm

1. Build a Heap of size n
Using the efficient constructor described in Geoff's lecture, this can be done in O(n) time
2. While the heap size is greater than 1
   2.1 swap the largest element in position 1 with the last element of the heap
   2.2 Decrease the heap size by 1
   2.3 Sift down the element in position 1

Heap Sort: formal algorithm

Heap Sort:
Pre: an array of comparable elements T[1..n]
Post a sorted array

```
heapSort(T[1.. n])
        heapify(T)
//For each position in T
        for i=n to 2 do
                //Swap the largest element to position i
                swap(T,1,i)
//Sift down the element now in position 1 to the new heap sized i-1
                siftDown(T,1,i-1)
```


Heap Sort

100	36	25	7	17	19	3	2	1

i=9

1	36	25	7	17	19	3	2	100

siftDown(H,1,8):

36	1	25	7	17	19	3	2	100
36	17	25	7	1	19	3	2	100

i=8

2	17	25	7	1	19	3	36	100

siftDown(H,1,7):

25	17	2	7	1	19	3	36	100
25	17	19	7	1	2	3	36	100

i=7

3	17	19	7	1	2	25	36	100

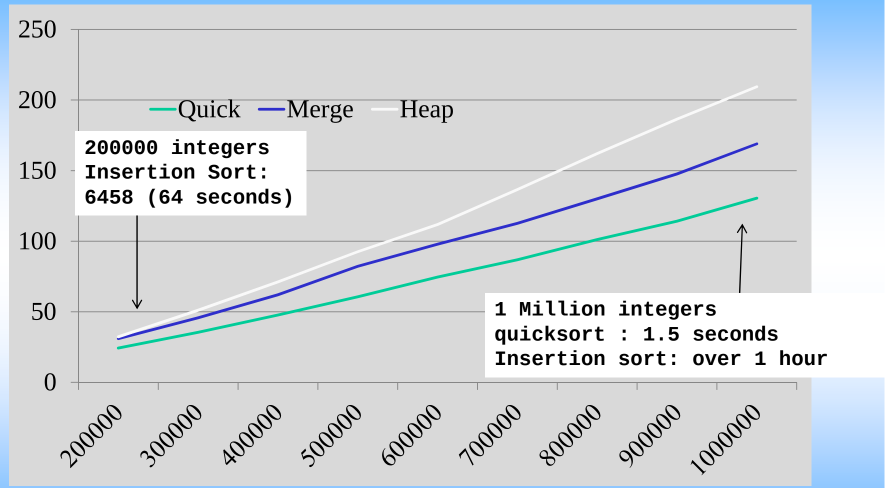
et. Cetera. Note the relationship to selection sort. At each stage we are selecting the largest element

Advanced Sorting Summary

		Merge sort	Quicksort	Heap Sort
Time Complexity (Comparison)	Worst Best Average	O(nlogn) O(nlogn) O(nlogn)	O(n^2) O(nlogn) O(nlogn)	O(nlogn) O(nlogn) O(nlogn)
Swaps		O(nlogn)	O(n^2)	O(nlogn)
Space Complexity	All cases	O(n) Not in place	O(1) In place	O(1) In place
Basis		Comparison	Comparison	Comparison
Stablility		Stable	Not Stable	Not Stable

Given this analysis, why does everyone use quicksort?

Sort Experiments



http://stackoverflow.com/questions/8311090/why-not-use-heap-sort-always

Java Sort Implementations

Java Sort Routines

The class `java.util.Array` provides sorting routines for sorting

Arrays of primitives

```
static void sort(double[] a)
static void sort(long[] a, int fromIndex, int
toIndex)
```

Arrays of Objects

(note objects are cast to Comparable, and an exception is thrown if the cast is illegal)

```
static void sort(Object[] a)
static sort(Object[] a, int fromIndex, int toIndex)
static void sort(Object[] a, Comparator c)
```

The algorithms used are different

Java Sort Implementations: primitives

Prior to java 7:

Modified quick sort is used to sort arrays of primitives

- If the sort segment size is less than 7, insertion sort is used
- When the segment size is between 7 and 40, the pivot is selected by the median of three rule
- When the segment size is greater than 40, the pivot is selected using a form of median of 9 (three groups of three are sampled, the median taken from each group, then the median of the three medians is selected)
- quicksort is **not stable**

Java quicksort for primitives

```
private static void sort1(int x[], int off, int len) {
        // Insertion sort on smallest arrays
        if (len < 7) {
            for (int i=off; i<len+off; i++)
                for (int j=i; j>off && x[j-1]>x[j]; j--)
                    swap(x, j, j-1);
            return;
        }
        // Choose a partition element, v
        int m = off + (len >> 1);  // Small arrays, middle element
        if (len > 7) {
            int l = off;
            int n = off + len - 1;
            if (len > 40) {        // Big arrays, pseudomedian of 9
                int s = len/8;
                l = med3(x, l,     l+s, l+2*s);
                m = med3(x, m-s,   m,   m+s);
                n = med3(x, n-2*s, n-s, n);
            }
            m = med3(x, l, m, n); // Mid-size, med of 3
        }
        int v = x[m];
```

Java quicksort

```
 private static void sort1(int x[], int off, int len) {

…// See previous slide
        // Establish Invariant: v* (<v)* (>v)* v*
        int a = off, b = a, c = off + len - 1, d = c;
        while(true) {
            while (b <= c && x[b] <= v) {
                if (x[b] == v)
                    swap(x, a++, b);
                b++;
            }
            while (c >= b && x[c] >= v) {
                if (x[c] == v)
                    swap(x, c, d--);
                c--;
            }
            if (b > c)
                break;
            swap(x, b++, c--);
        }
// Swap partition elements back to middle
        int s, n = off + len;
        s = Math.min(a-off, b-a  );  vecswap(x, off, b-s, s);
        s = Math.min(d-c,   n-d-1);  vecswap(x, b,   n-s, s);

        // Recursively sort non-partition-elements
        if ((s = b-a) > 1)
            sort1(x, off, s);
        if ((s = d-c) > 1)
            sort1(x, n-s, s);
    }
```

Java Sort Implementations: primitives

From java 7:
Dual Pivot quick sort is used to sort arrays of primitives.

- Uses standard quick sort on arrays smaller than 286
- Has loads of tweaks for using insertion sort on sub arrays (very complex, compressed code).
- It forms three partitions from two pivots and recursively sorts these

It is meant to reduce the likelihood of a pathological worst case

Java Sort Implementations: references

Prior to Java 1.7
Modified merge sort is used to sort arrays of object references

•Uses insertion sort for sub arrays size 7 or less
•The merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist.
•Mergesort is **stable**

Java Sort Implementations: references

```
private static void mergeSort(Object[] src,
                              Object[] dest,
                              int low,
                              int high,
                              int off) {
int length = high - low;

       // Insertion sort on smallest arrays
if (length < INSERTIONSORT_THRESHOLD) {
   for (int i=low; i<high; i++)
     for (int j=i; j>low &&
       ((Comparable) dest[j-1]).compareTo(dest[j])>0; j--)
     swap(dest, j, j-1);
   return;
}
```


Java Sort Implementations: references

```
        // Recursively sort halves of dest into src
        int destLow  = low;
        int destHigh = high;
        low  += off;
        high += off;
        int mid = (low + high) >>> 1;
        mergeSort(dest, src, low, mid, -off);
        mergeSort(dest, src, mid, high, -off);

        // If list is already sorted, just copy from src to dest.  This is an
        // optimization that results in faster sorts for nearly ordered lists.
        if (((Comparable)src[mid-1]).compareTo(src[mid]) <= 0) {
            System.arraycopy(src, low, dest, destLow, length);
            return;
        }

        // Merge sorted halves (now in src) into dest
        for(int i = destLow, p = low, q = mid; i < destHigh; i++) {
            if (q >= high || p < mid &&
((Comparable)src[p]).compareTo(src[q])<=0)
                dest[i] = src[p++];
            else
                dest[i] = src[q++];
        }
    }
```


Java Sort Implementations

From Java 1.7 onwards

- An algorithm called Timsort is used.
- It was invented in 2002 by Tim Peters for use in the Python programming language.
- It is a hybridization of binary insertion sort and iterative merge sort
- It is highly complex, involving counting and forming “runs” and “galloping” across them.

After Sorting Lecture 4 you should be able to …

1. **Describe heap sort both informally and in formal pseudo code**
2. **Heap sort an example array**
3. **Analyse the worst case time complexity**
4. **Know what the average case complexity is**

https://vaskoz.wordpress.com/2013/07/21/java-8-parallel-array-sorting/