Sorting

Sunday, October 21, 2007
11:47 PM

- Two types of sort
    - internal - all done in memory
    - external - secondary storage may be used

- 13.1 Quadratic sorting methods
    - data to be sorted has relational operators such as < and ==
    - sort results in ascending or descending order based off data valve or a key in a record
    - Selection Sort
        - scan through list looking for smallest (or largest) element further in list
        - swap that element w/ current element
            - 67, 33, 21, 84, 49, 50, 75
            - 21, 33, 67, 84, 49, 50, 75
            - 21, 33, 49, 84, 67, 50, 75
            - 21, 33, 49, 50, 67, 84, 75
            - 21, 33, 49, 50, 67, 75, 84
        - Pseudocode

```
sort the array x[1] to x[n]
for i=1 to n-1
    set minPos to i
    set min to x[i]
    for j=i+1 to n
        if x[j]< min
            set minPos to j
            set min to x[j]
    set x[minPos] to x[i]
    set x[i] to min
```

    - Exchange Sort
        - systematically interchange elements
        - bubbletops is a common exchange sort
            - very inefficient but easy to learn
                - compare neighboring elements and put two in sorted order
                - result of one pass is that largest element is swapped to end of list
                - next pass excludes last element
        - Example:
            - 67, 33, 21, 84, 49, 50, 75
                - 33, 67
                    - 21, 67
                        - 67,84
                            - 49, 84
                                - 50, 84
                                    - 75, 84
            - 33, 21, 67, 49, 50, 75, 84
                - 21, 33
                    - 33, 67

```
49, 67
    50, 67
        67, 75
```

- 21, 33, 49, 50, 67, 75, 84
- would still do pass for 21-50 but would do no swaps

- Pseudocode

  ```
  sort x[1] to x[n]
  set passes to n-1
  while passes is not 0
      set last to 1
      for i=1 to passes
          if x[i]>x[i+1]
              swap x[i] and x[i+1]
              set last to i
      set passes to last-1
  ```

- Insertion Sort
  - insert element into already sorted list
  - start w/ 1 element list & grow
  - at pass p, elements 1 to p are sorted & p+1 inserted in sorted order
  - Example:
    - 67, 33, 21, 84, 49, 50, 75 p=1 do nothing, original array
    - 33, 67, 21, 84, 49, 50, 75 p=2
    - 21, 33, 67, 84, 49, 50, 75 p=3
    - 21, 33, 67, 84, 49, 50, 75 p=4
    - 21, 33, 49, 67, 84, 50, 75 p=5
    - 21, 33, 49, 50, 67, 84, 75 p=6
    - 21, 33, 49, 50, 67, 75, 84 p=7
  - Pseudocode

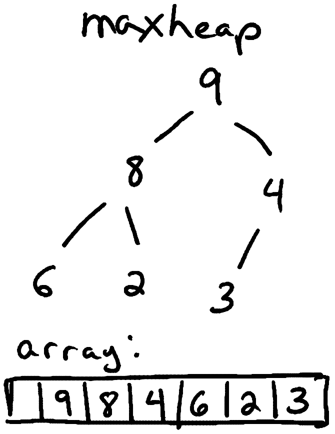
    ```
    sort x[1] to x[n], use x[0] to store x[p]
    for p=2 to n
        set x[0] to x[p]
        set j to p
        while x[0]< x[j-1]
            set x[j] to x[j-1]
            decrement j
        set x[j] to x[0]
    ```

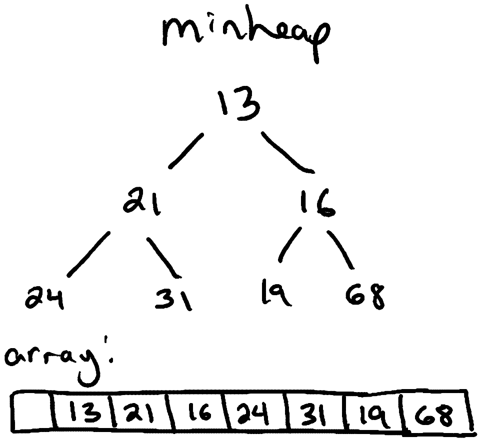
- Evaluation of sorting schemes
  - all have quadratic worst & average cases
  - selection sort
    - simple, but must scan list/array for next smallest/largest item
    - heapsort is a more efficient selection sort
    - performance does not improve when lists are partially/fully sorted
  - bubble sort
    - better for partially/fully sorted lists
    - inefficient due to volume of swaps
    - quicksort is a better exchange sort
  - insertion sort
    - better then selection/bubble sort
    - still inefficient
    - good for small lists (n<20) or partially sorted lists
  - Indirect Sorting

- Indirect Sorting
  - use index table to sort positions of large records
  - rather than swap large objects (like StudentRecord) swap their indexes in index table
  - scan index table sequentially to find order to traverse records
  - Example:
    - index table: 5, 3,1,2, 4, 0
    - means to traverse element 5, then 3, then 1, etc
- Shell sort & binary insertion sort are better insertion sorts
  - binary uses binary search to find hole
  - Shell produces partially ordered sublists

13.2 Heaps, Heapsort & Priority Queues

- O(n log2n) is best possible worst case sorting time
- heapsort is a type of selection sort that has this runtime
- Heap
  - a complete binary tree
    - all levels filled except possibly the bottom level
    - bottom level is filled in left positions
    - if represented as array, no holes would be left in array
  - tree & subtrees have heap-order property
    - max heap-order
      - root value is greater than or equal to value of its children
    - min heap-order
      - root value is less than or equal to value of its children
  - The 0th slot in the array is reserved for use by heapsort

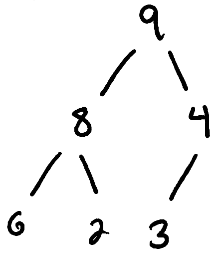




- Heap Operations
  - construct empty heap
    - set count to 0
  - check empty
    - return true if count is 0
  - retrieve max (or min for min heap) value
    - if empty()
      - issue "empty heap " error
    - else
      - return value of root
  - delete max (or min) valve

- delete max (or min) valve
  - Issue
    - must replace root w/ next sorted item
    - because of heap order, one of root's children is next
    - cannot just move it up because completeness must be maintained
  - Solution
    - move rightmost bottom level node up to root
      - maintains completeness because that node is at end of array
    - while this node violates heap order
      - swap w/ child that restores heap order

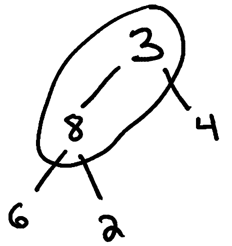
Example:



	9	8	4	6	2	3

remove 9

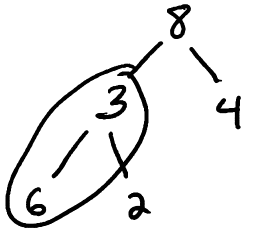
replace w/ 3



now have a semiheap

	3	8	4	6	2

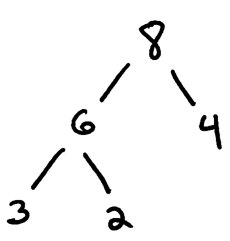
swap 3 w/ 8 to restore heap-order



Still a semiheap

	8	3	4	6	2

swap 3 w/ 6 to restore heap-order



now a heap again

	8	6	4	3	2

- this process is called percolate-down
- Remove Pseudocode
  - set x[1] to x[count]
  - decrement count
  - call percolate-down
- Percolate-down Pseudocode
  - Given: a semi-heap starting at slot r

```
while r<=count do
    set c to 2*r // left child
    if c<count // r has two children
        AND x[c]<x[c+1] // right is larger
        set c to c+1 // select right child
    if x[r]<x[c] // heap order violated
        AND c<=count // valid child
        swap x[c] and x[r]
        set r to c
    else
        break // heap order restored, end while loop
```

- Insert an item
    - place at end of array & percolate-up
    - Pseudocode

```
increment count
set x[count] to value
call percolate-up
```

- Percolate-up Pseudocode

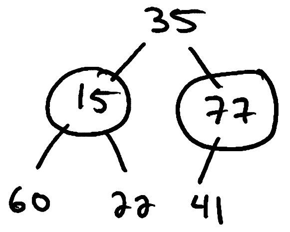
```
set loc to count
set parent to loc / 2
while parent >= 1 AND x[loc] > x[parent]
        swap x[loc] and x[parent]
        set loc to parent
        set parent to loc / 2
```

Heapsort

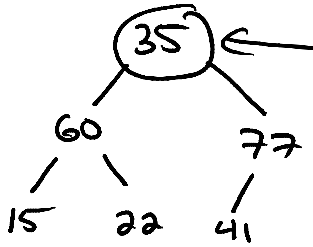
- given an array to sort
    - treat array as a complete tree
    - convert tree into heap
- How to convert time into heap?
    - keep applying percolate down to non-leaves
    - start at rightmost non-leaf

Example:

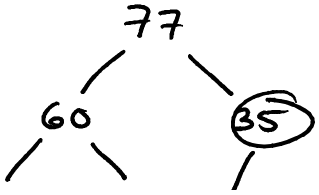
35, 15, 77, 60, 22, 41



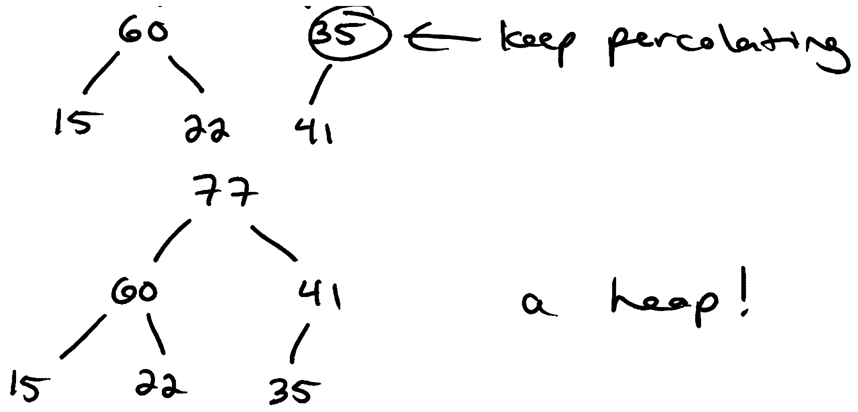
← start here, percolate-down
nothing to do
check 15 next



← now percolate-down 35



← keep percolating



- heapify Pseudocode

```
for r = n /2 down to 1
    percolate-down at r
```

- Once we have a heap, can now sort
    - delete root
        - this moves rightmost bottom node up to root & percolates it down
    - copy root value to end of array
        - fill in hole left by moving rightmost bottom node
    - repeat w/ subheap that excludes this coped root value
- Pseudocode

```
heapify x
for i=count down to 3
    set x[0] to x[1]
    delete root of x[1] to x[i] heap
    set x[i] to x[0]
swap x[1] and x[2]
```

Advantages of Heaps
- do not become lopsided
    - always complete
- O(n log2n) thus assured
- good for priority queues
    - highest priority is root

13.3 Quicksort

- fast method to sort
- uses divide-and-conquer strategy
- Algorithm

```
If number of elements is 0 or 1
    do nothing // stopping condition
Else
    select an element as the pivot
    split remaining elements in to:
        smaller : elements <= pivot
        greater: element > pivot
    return quicksort (smaller), pivot, quicksort (larger)
```

- Selecting the pivot
    - pivot can be any element
    - if select 1st element always, have poor performance w/ sorted lists
        - everything is either smaller or larger
        - makes runtime quadratic
    - want even distribution most of the time
    - choosing randomly gets good partition of elements
        - costly to generate random number
    - median-of -three
        - select median of first, middle & last elements
        - gets a pivot closer to median of the whole list than just selecting first element
- Splitting / Partitioning the list
    - several methods to generate smaller and larger
    - search method
        - swap pivot w/ either 1st or last element
        - Start two searches
            - i starts at 0 (1 if pivot is 0)
                - i looks for elements > pivot
            - j starts at size-1 (size-2 if pivot is size-1)
                - j looks for elements <= pivot
            - when both i & j have stopped, swap the elements
            - repeat search until i & j cross
                - then swap pivot
                    - if pivot in 0, swap w/ j
                    - if pivot in size-1, swap w/ i
    - now have smaller & larger subsets
        - subsets can be sorted w/ any scheme
        - can use fast method for small subsets like insertion sort
- Runtime
    - best case: n log2 n
        - pivot is median of list, partitions evenly
        - recursion creates a binary tree w/ log 2n levels
    - average case: n log 2 n
        - pivot is not perfect, but still creates tree-enough like structure
    - worst case: quadratic
        - pivot is largest or smallest element, partitions skewed
        - list is already sorted (ascending or descending)
        - creates linked list instead of binary tree
- Code

```
template <class T>
int median-of-three(T a[], int first, int last) {
    int c = (first +last) / 2;
    if(a[c]<a[first])
        swap(a[first], a[c]);
    if(a[first]<a[last])
        swap(a[first], a[last]);
    if(a[first]<a[c])
        swap(a[first, a[c ]);
    swap(a[first], a[c]);
    return first;
```

```
    return first;
}
template <class T>
int split(T a[], int first, int last) {
    int p = median-of-three(a, first, last);
    int pivot = a[p];
    swap(a[first], a[p]);
    int i = first + 1;
    int j = last;
    while(i<j) {
        while(pivot<a[j])
            j--;
        while(i<j && a[i] <=pivot)
            i++;
        if(i<j)
            swap(a[i], a[j]);
    }
    swap(a[first], a[j]);
    return j;
}
template <class T>
void quicksort(T a[], int first, int last) {
    int p;
    if(first<last) {
        p=split(a,first,last);
        quicksort(a,first,p-1);   // can use faster sort here
        quicksort(a,pos+1 ,last);  // and here
    }
}
```

13.4 Mergesort

- uses files as storage structure
- merges two files into third, sorted file
- Basic merge
  - take element from each file
  - place smaller in output file & replace w/ next element in its file
  - Example:
    - file1: 15 20 25 35 45 60 65 70
    - file2: to 30 40 so 55
    - x=15
    - y=10
    - place 10 in file3
    - y=30
    - place 15 in file 3
    - y=20
    - place 20 in file 3
    - x = 25
    - and so on
  - when run out of input in one file dump remaining contents of other file to output
  - Algorithm
    - read x from file1
    - read y from file2

```
while not Eof for either file
    if x<y
        write x to file3
        read x from file1
    else
        write y to file3
        read y from file2
if Eof of file1
    dump remaining file2 to file3
if EOF of file 2
    dump remaining file1 to file3
```

- Binary mergesort
    - given a single file to be sorted
    - how to split into two files?
        - send even slots to one file
        - send odd slots to other file
    - don't scan & output like w/ basic
        - instead sort groups of numbers
        - pass 1, take 1 element from each file
            - create 2 element sorted output
        - pass 2, take 2 elements from each file
            - create 4 element sorted output
        - pass 3, take 4 elements from each file
            - create 8 element sorted output
        - pass n, take 2^(n-1) elements
            - create 2^n element sorted output
- Natural mergesort
    - helpful for partially sorted files
        - instead of splitting on even/odd, splits when x [i +1] < x[i]
        - i.e. splits at end of a sorted run
    - merge also takes advantage of runs
        - merge runs regardless of length
    - Example:
        - input: 75 55 15 20 85 30 35 10 60 40 50 25 45 80 70 65
        - Split 1:
            - f1: 75 15 20 85 10 60 25 45 80 65
            - f2: 55 30 35 40 50 70
        - Merge 1:
            - file: 55 75 15 20 30 35 40 50 70 85 10 60 25 45 80 65
        - Split 2:
            - f1: 55 75 10 60 65
            - f2: 15 20 30 35 40 50 70 85 24 45 80
        - Merge 2:
            - file: 15 20 30 35 40 50 55 70 75 85 10 25 45 60 65 80
        - Split 3:
            - f1: 15 20 30 35 40 50 55 70 75 85
            - f2: 10 25 45 60 65 80
        - Merge 3:
            - file: 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85
    - Algorithms:
        - Split

```
Open F for input and F1 & F2 for output
While not EOF for F
```

```
        Copy elements from F into F1 until x[i+1]<x[i]
        Copy elements from F into F2 until x[i+1]<x[i]
Merge
    Open F1 & F2 for input, F for output
    Initialize numSub to 0
    While not EOF on F1 or EOF on F2
        While the end of  a run has not been met in either F1 or
        F2
            copy smaller of two elements to F
        if EOF on F1
            copy rest of F2's run to F
        else
            copy rest of F1's run to F
        increment numSub
    While run in F1
        copy run to F
        increment numSub
    While run in F2
        copy run to F
        increment numSub
    return numSub
Mergesort
    initialize numSub to 0
    do-while numSub is not 1
        split F
        set numSub to merge F1,F2
```

Runtime: O(nlog2n)

```
merging runs
    set runs to 0
    read f1 from Fl
    read f2 from F2
    while not EOF for F1 & F2
        set end1 to false
        set end2 to false
        while not end1 and not end2
            if f1 < f2
                output f1
                read f1 from F1
                if end of run
                    set end1 to true
            else
                output f2
                read f2 from F2
                if end of run
                    set end2 to true
            while end1 and not end2
                output f2
                read f2 from F2
                if end of run
                    set end2 to true
            while end 2 and not end1
```

```
        output f1
        read f1 from F1
        if end of run
            set end1 to true
    increment runs
if not EOF for F1
    output f1
    read f1 from F1
    if end of run
        increment runs
if not EOF for F2
    output f2
    read f2 from F2
    if end of run
        increment runs
return runs
```