- intro to sorting
- big-Oh notation O()
- exchange sorts
- insertion sorts
- selection sorts

Online Final exam is next week

- full details in 'Check Canvas, Grades' and 'About the Final exam' at the end of this doc

Next week and homework

- next week: searching; Final exam
- homework by next week:
  - download, run and understand this week's example programs
  - read Horstmann, sections 14.6 – 14.8
  - to prepare for Final exam: re-read all my lectures and my example programs
  - review all of the programs you have written
  - re-read Horstmann textbook, Chapters 14 – 17

Lab

- check Canvas, Grades
- about the Final exam
- current lab was assigned in 'Finish trees; hashing; graphs'. See Canvas for due date

Review last week

- finished binary tree implementation
- covered hashing
  - is a technique for very fast search
  - have to handle collisions
- introduced graphs
  - used to model relationships between vertices and edges

Introduction to this week

- introduce the important CS topic of sorting
- use Big-oh O() notation to compare algorithms
- will look at three different classifications of sorting
- learn about the Final exam, for next week!

Intro to sorting
Objective: introduce definition, ideas, terminology of this hugely important CS topic

<u>Re-arrange data into an order</u>

- re-arrange data items so that they are in order in some way:
  - ascending: $e_1 < e_2 < e_3 < \ldots < e_n$
  - descending: $e_1 > e_2 > e_3 > \ldots > e_n$

<u>Internal vs. external sorts</u>

- two general classifications of sort algorithm:
  - <u>internal</u> – all items can fit into main memory
  - <u>external</u> – on mass storage – too much data to fit into main memory
- internal sort typically involves
  - small amount of data
  - direct access to every element
  - the dominant factor is how many times an element is visited
  - so algorithm speed is typically proportional to the number of visits made
  - …we will be looking at internal sort algorithms that sort lists of numbers inside an array
- external sorts typically involve sorting files of records stored on magnetic media
  - huge amounts of data e.g. IRS sorting its file of taxpayers!
  - very important process
  - legacy computer systems
  - won't cover this

Efficiency of algorithms

- will be concerned with the efficiency of various internal sorting algorithms:
  - time – execution speed
  - space – amount of memory required
  - complexity – how easy to understand and implement
- we typically find tradeoffs here – one at the expense of the other
  - e.g. the "time vs. space" tradeoff – fast algorithms require a lot of memory
  - (…although recursion is usually both slow and big!)
  - e.g. a "time vs. complexity" tradeoff – fast algorithms are harder to understand and implement
  - (we'll see that execution speed is somewhat proportional to algorithm complexity)
- we are primarily concerned with execution time – how quickly an algo sorts a list
  - this is expressed using big-Oh notation
- will look at three of the classes of sorting algorithms:
  - exchange
  - insertion
  - selection
  - and at three examples of each class

Summary

- there is no single ideal sort for every situation
  - time vs. space vs. complexity tradeoffs are always involved
- (BTW, if you are interested, see a CS classic for complete coverage of sorting (…and many other CS topics)

- Donald Knuth, Stanford
- "The Art of Computer Programming", many volumes
- a CS classic – one of the books that defined the field
- written originally in the 60s, then updated
- also, find Knuth's homepage

Big-Oh notation O()

Objective: big-Oh notation O() expresses the complexity of an algorithm. See how we calculate it, and use it to classify the speed of algorithms

Execution speed of algorithms is most important

- we've seen that the overall efficiency of an algorithm has several different components:
    - time – execution speed
    - space – amount of memory required
    - complexity – how easy to understand and implement
- execution speed is usually the most important of these
    - an algorithm has to complete within an appropriate time

Algorithm races are a bad idea

- how should we compare two algorithms to say which is faster than another?
    - we could race algorithms! – run two algos on the same data, see which is faster
- algorithm races are a bad idea. They do not compare the fundamental properties of the algos
    - implementation of one may be better than the other
    - one may be more suited to the particular hardware architecture than the other
    - the test data may be more favorable to one than the other

Use big-Oh notation O() to express execution times

- instead we calculate the order of complexity (or magnitude) of each algorithm
    - "a mathematical analysis of algorithm to express speed/execution time in terms of problem size"
    - problem size is the number N of items to be sorted
    - this measure is independent of implementation details

   – is a part of CS called algorithmic analysis

- this order of complexity is expressed using big-Oh notation O()
   – e.g. we've seen search algorithm complexity in terms of N:

search algo	proportional to	"of order"
hashing	1	O(1)
binary	log N	O(logN)
sequential	N	O(N)
(others may be)	N^2	$O(N^2)$
	N^3	$O(N^3)$
	…	
	c^N	$O(c^N)$
	N!	O(N!)

Use big-Oh notation to compare execution times

- big-Oh notation is the basis of a classification scheme of algorithms:

search algo	proportional to	"of order"	class
hashing	1	O(1)	constant
binary	log N	O(logN)	logarithmic
sequential	N	O(N)	linear
(others may be)	N^2	$O(N^2)$	quadratic
	N^3	$O(N^3)$	cubic
	…		
	c^N	$O(c^N)$	exponential
	N!	O(N!)	factorial

- classes are shown in increasing order of execution time….

- the classification of an algorithm shows us how execution time will increase as N increases

  - e.g. say N increases 10 times, then execution times increase:

search algo	"of order"	increase in execution time as N increases 10 times
hashing	O(1)	constant time, no increase!
binary	O(logN)	(logarithmically – increases slowly)
sequential	O(N)	10 times
quadratic	$O(N^2)$	100 times
cubic	$O(N^3)$	1 000 times
exponential	$O(c^N)$	(exponentially – quickly)
factorial	O(N!)	(quickest increase)

- can illustrate execution times of different complexity classes as a graph

  - graph execution time t as N increases:

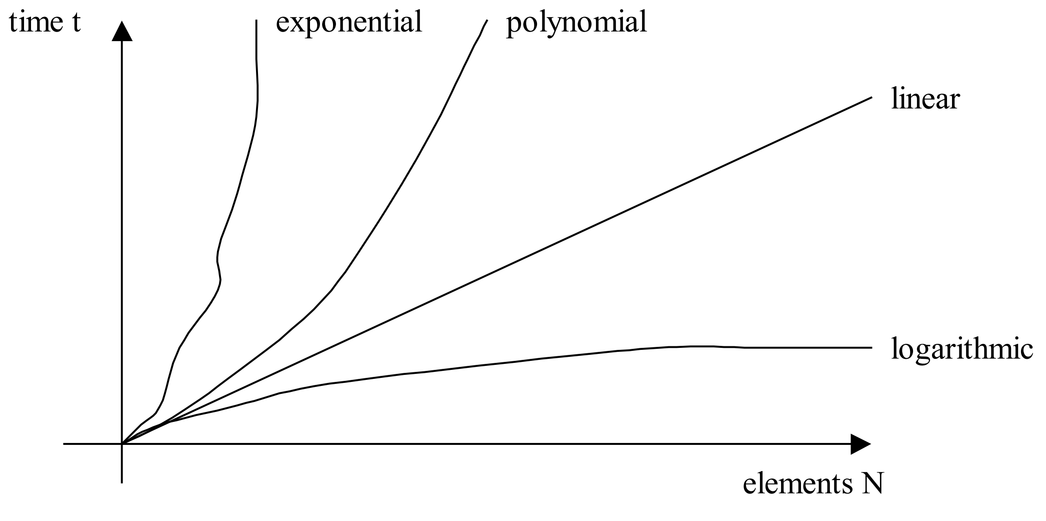


Figure 1 comparing different complexity classes

- all much the same when N is small…
- …huge differences when N is large!

- e.g. some execution time comparisons for large N
  - if N = 1 000 000 data items:

class	big-Oh	time
constant	O(1)	1
logarithmic	O(logN)	19
linear	O(N)	1 000 000
quadratic	$O(N^2)$	10^{12}
cubic	$O(N^3)$	10^{18}
exponential	$O(2^N)$	$10^{301\ 030}$
factorial	O(N!)	$\sim 10^{5\ 565\ 708}$

  - so maybe a logarithmic algo would be fine, but an exponential version would not!

<u>How to calculate big-Oh</u>

- how do we analyse an algo to determine its complexity?
  - the dominant factor is the number of visits made to the elements
  - therefore we want to approximate the <u>total number of visits</u> required, in terms of N
  - is frequently calculated in two parts, as:
  - number of passes * number of visits per pass

- then worst vs. average vs. best cases can be significant…
  - …particularly is they are widely different for an algorithm
  - so we generally produce big-Oh for <u>worst case</u>

- algorithm can only be this speed or better

- calculate big-Oh only as an approximation, in two steps:

1. drop any constant terms e.g. if the number of visits is:

   $N - 1$

   - the constant term – 1 here is significant for small N e.g. if $N = 2$
   - but insignificant for large N e.g. if $N = 1\ 000$
   - so, drop the constant terms from the following

   $N - 1$
   $N / 2$
   $3N + 4$
   $1\ 000N$

   - all are $O(N)$

2. identify dominant term e.g. if the number of visits is:

   $N^2 - N$

   - non-dominant term – N is significant for small N e.g. if $N = 2$
   - but insignificant for large N e.g. if $N = 1\ 000$
   - so, drop the non-dominant term from the following

   $N^2 - N$
   $2\ N^2 * 100N$
   $3\ N^2$
   $2\ N^2 + 10N$

   - all are $O(N^2)$

- approximating is more justified than an exact calculation, given implementation-dependent factors when algorithm runs

Summary
- will look at many different sorting algorithms and show their complexity

- will calculate big-Oh for a sort algorithm shortly
- understanding how an algorithm works is more important than memorizing an implementation

Exchange sorts
Objective: understand the general idea, see some examples

Exchange sorts swap two items

- "swap two out of order items until the list is sorted". Some well-known exchange sorts we'll look at:
  - bubble sort
  - shaker sort
  - Quicksort

1. Bubble sort

- many different variations. The one we will consider, first pass:
  - compare 1st and 2nd, swap if necessary
  - compare (new) 1st and 3rd,
  - compare (new) 1st and 4th,
  - and so on, e.g.

```
9    3    5    2    7
^    ^
3    9    5    2    7
^         ^    ^
2    9    5    3    7
^                   ^
```

  - end of first pass – see that smallest item 'bubbles to the top'

- second pass: now compare 2nd against remainder of list…

```
2    9    5    3    7
     ^    ^
2    5    9    3    7
     ^         ^
2    3    9    5    7
     ^              ^
```

  - end of second pass, and so on…

- third pass. Sets 3rd item to lowest value remaining. See how less work is done each pass:

```
2    3    9    5    7
          ^    ^
2    3    5    9    7
          ^         ^
```

  – end of third

- fourth pass

```
2    3    5    9    7
               ^    ^
2    3    5    7    9
```

  – end of fourth and end of sort

<u>Calculate big-Oh</u>

- calculate the order of complexity, big-Oh for N items
  – #visits = #passes * #visits per pass
  – #passes = N – 1
  – #visits per pass = 4 + 3 + 2 + 1

    = N / 2

  – #visits = (N – 1) * N / 2

$$= \frac{N^2}{2} - \frac{N}{2}$$

- remove constant terms (insignificant as N becomes very large)

$$= N^2 - N$$

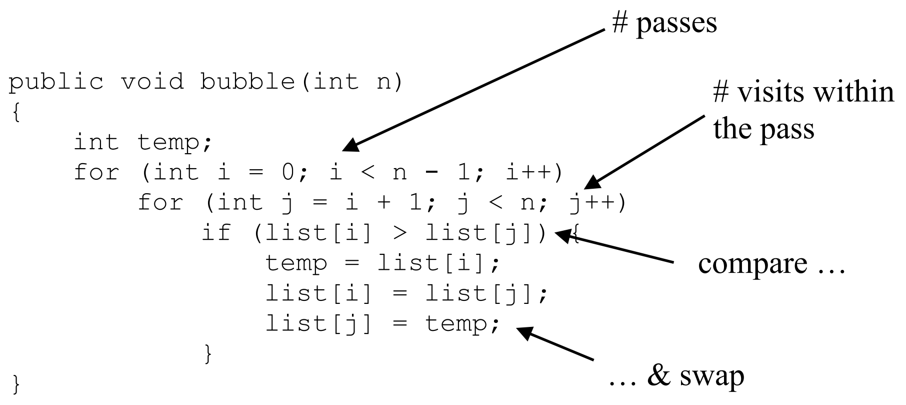
- identify dominant term (as N becomes very large)

$$= N^2$$

- CONCLUDE: bubble sort is $O(N^2)$

Review implementation

- review an implementation of this version of bubble sort



```
public void bubble(int n)
{
    int temp;
    for (int i = 0; i < n - 1; i++)
        for (int j = i + 1; j < n; j++)
            if (list[i] > list[j]) {
                temp = list[i];
                list[i] = list[j];
                list[j] = temp;
            }
}
```

Figure 2 an implementation of bubble sort

- parameter `n` is the number of items in the list
- HINT: do not declare local variables inside a loop – may be inefficient
- (understand, don't memorize)

2. Shaker sort

- a refinement of bubble sort
  - as smaller items bubble to the top (like the bubble sort),
  - larger items also drop to the bottom
  - (hence the name – it's like shaking something)

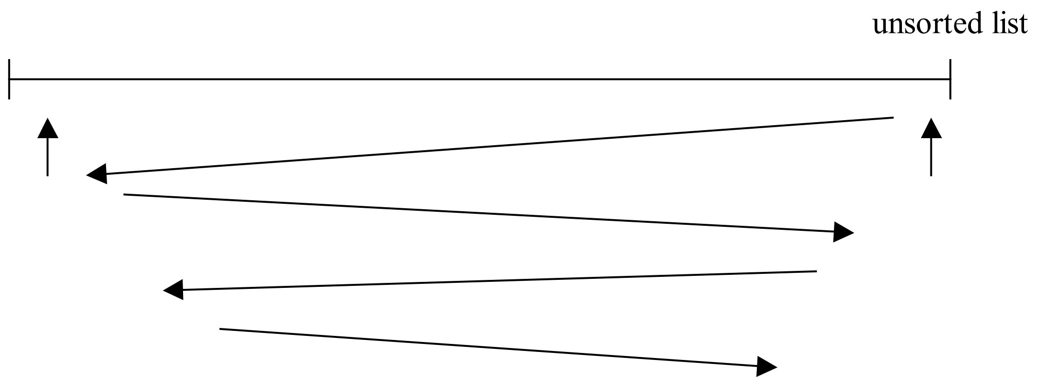


Figure 3 the shaker sort algorithm

- general idea of the algorithm:
  - start with boundary indexes at each end of the unsorted list
  - then sort left and right between boundaries
  - move boundaries each pass…
- order of complexity
  - still O(N^2) (but better by a constant factor)
- an implementation of shaker sort

```
public void shaker(int n)  start k at the left side
{
    int i, temp;
                           start m at the right
    int k = 1;
    int m = n - 1;
    int j = n - 1;
    do {
        for (i = m; i >= k; --i)
            if (list[i - 1] > list[i]) {
                temp = list[i - 1];
                list[i - 1] = list[i];
                list[i] = temp;
                j = i;              move k up towards m
            }
        k = j + 1;
        for (i = k; i <= m; ++i)
            if (list[i - 1] > list[i]) {
                temp = list[i - 1];
                list[i - 1] = list[i];
                list[i] = temp;
                j = i;              move m down towards
            }                       k
        m = j - 1;
    } while (k <= m);               until they cross
}
```

Figure 4 an implementation of shaker sort

3. <u>Quicksort</u>

- a really good exchange sort
  - by Tony Hoare (1962)

- involves recursive partitioning of the unsorted list:

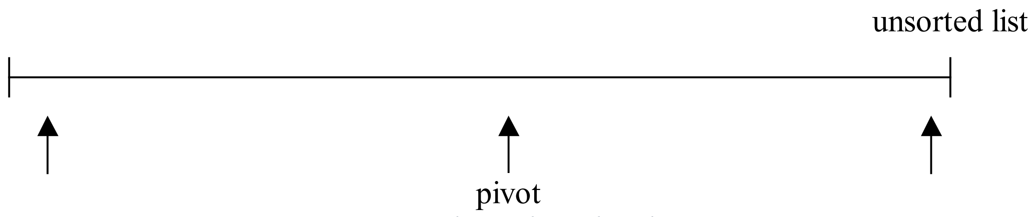


Figure 5 the Quicksort algorithm

  - choose a value from the list, called the pivot
  - (…we will choose the value in the middle of the list)
  - advance 2 indexes from left and right of the list
  - exchange items when left > pivot and right < pivot until indexes cross
  - anything < pivot is to the left
  - anything > pivot is to the right
  - so this pass results in two smaller, partially sorted lists
  - now repeat this process recursively for the lists left and right of the pivot
  - and so on
- Quicksort gives best performance when lists are of equal size each split…
  - …corresponding to a complete binary search tree
- order of complexity, average case
  - avg #visits per pass = N / 2
  - avg #passes – looks like binary tree = logN
  - Quicksort is O(NlogN) on average

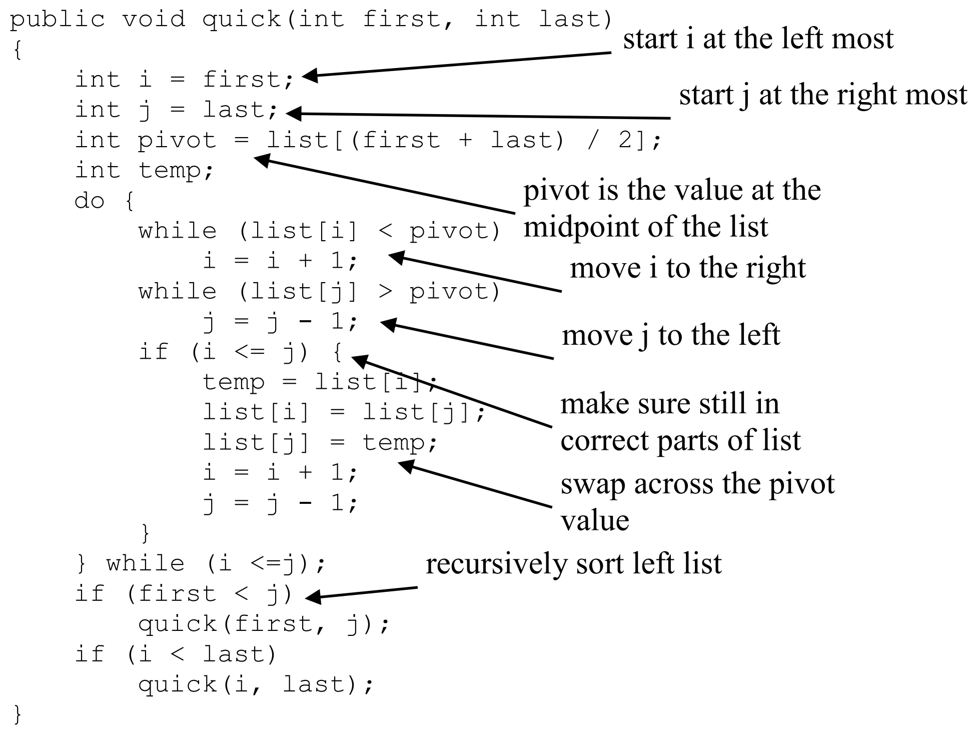
Quicksort's surprising worst case

- surprisingly, Quicksort's worst case scenario is when the list is already almost sorted!

- this gives a pivot of the mid point of the starting list and of every split – causing lots of visits, lots of passes, but few exchanges

- order of complexity, worst case
  - worst case #visits per pass = N
  - worst case #passes = N
  - Quicksort is $O(N^2)$ in worst case, when list is already almost sorted!

- an implementation of Quicksort



```
public void quick(int first, int last)
{
    int i = first;
    int j = last;
    int pivot = list[(first + last) / 2];
    int temp;
    do {
        while (list[i] < pivot)
            i = i + 1;
        while (list[j] > pivot)
            j = j - 1;
        if (i <= j) {
            temp = list[i];
            list[i] = list[j];
            list[j] = temp;
            i = i + 1;
            j = j - 1;
        }
    } while (i <=j);
    if (first < j)
        quick(first, j);
    if (i < last)
        quick(i, last);
}
```

start i at the left most
start j at the right most
pivot is the value at the midpoint of the list
move i to the right
move j to the left
make sure still in correct parts of list
swap across the pivot value
recursively sort left list

Figure 6 an implementation of Quicksort

- (BTW, Quicksort is the fastest of all the sorts we will look at)

<u>Summary</u>

- from Canvas, 'Sorting' module, Example programs, download, read and run my `Sorting` example program

- checkout the exchange sorts
- just enjoy, try to understand
- (no need to memorize the implementations!)

Insertion sorts

Objective: study some examples, understand general ideas

<u>Insertion sorts insert an item into the correct place</u>

- "take an item and insert it into correct place amongst sorted items". Will look at three common insertion sort algorithms:
  - linear insertion sort
  - binary insertion sort
  - shell sort

1. <u>linear insertion sort</u>

- divide list into sorted and unsorted parts, then repeatedly take first unsorted item and insert into sorted part
  - e.g., first pass

```
    7       3       5       2       6

sorted              unsorted
    7       |       3       5       2       6
                    ^
```

  - need to open an <u>insertion point</u> in the list, to allow for data movement, so assign 3 to temp:

```
    7       |       _       5       2       6               temp is 3
```

  - now move items in sorted partition up the list until we've opened the destination of 3:

```
    _       |       7       5       2       6
```

  - put the 3 in the correct place and update the partition:

```
sorted                      unsorted
    3       7       |       5       2       6
                            ^
```

  - end of first pass
- second pass – now insert first unsorted into sorted partition:

```
    3    7    |    _    2    6          temp is 5

    3    _    |    7    2    6

    3    5    7    |    2    6
```

  - end of second pass with partition updated, and so on…

- third pass

```
    3    5    7    |    _    6          temp is 2

    3    5    _    |    7    6

    3    _    5    |    7    6

    _    3    5    |    7    6

    2    3    5    7    |    6
```

  - end of third pass

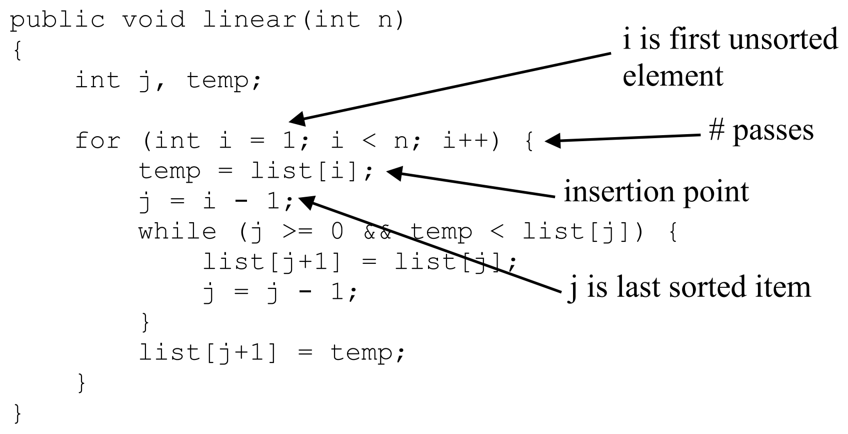
- final pass

```
    2    3    5    7    |    _          temp is 6

    2    3    5    _    |    7

    2    3    5    6    |    7
```

  - end of fourth pass and end of sort

- order of complexity
  - #passes = N – 1
  - avg #visits per pass = N / 2
  - linear insertion sort is $O(N^2)$ (average and worst cases)

- an implementation of linear insertion sort



```
public void linear(int n)
{
    int j, temp;

    for (int i = 1; i < n; i++) {
        temp = list[i];
        j = i - 1;
        while (j >= 0 && temp < list[j]) {
            list[j+1] = list[j];
            j = j - 1;
        }
        list[j+1] = temp;
    }
}
```

Figure 7 an implementation of linear insertion sort

2. <u>binary insertion sort</u>

- is a slight refinement to the linear insertion sort
  - use binary (not linear) search of sorted partition to find correct location of first unsorted item…
- is faster, but with so much data movement
  - binary insertion sort is still O(N^2)
- an implementation of binary insertion sort

```
public void binaryInsertion(int n)
{
    int i, j, left, right, middle, temp;

    for (i = 1; i < n; ++i) {        # passes
        temp = list[i];
        left = 0;
        right = i - 1;                    binary search of sorted
        while (left <= right) {           partition
            middle = (left + right) / 2;
            if (temp < list[middle])
                right = middle - 1;
            else
                left = middle + 1;
        }
        for (j = i - 1; j >= left; --j)
            list[j + 1] = list[j];
        list[left] = temp;
    }
}
```

Figure 8 an implementation of binary insertion sort

3. <u>shell sort</u>

- a really good insertion sort
  - by Donald L. Shell (1959)
- involves dividing the unsorted list into different groups and fully sort within these groups. Then divide into different groups and fully sort these, and so on…
  - …but the groups are not contiguous within the list!
  - e.g.

14	70	59	21	53	44	33	8	88	82	29

  - choose an <u>increment</u>, then group by this increment e.g. 4

	14	70	59	21	53	44	33	8	88	82	29
group:	1	2	3	4	1	2	3	4	1	2	3

  - linear insertion sort within each group until each group is sorted, gives:

14	44	29	8	53	70	33	21	88	82	59
1	2	3	4	1	2	3	4	1	2	3

- end of first pass

- notice here how:
  - each item has moved further than for a regular linear insertion sort
  - the list is now much closer to sorted

- pass two – now choose a smaller increment and sort again e.g. increment 3

14	44	29	8	53	70	33	21	88	82	59
1	2	3	1	2	3	1	2	3	1	2

  - linear insertion sort within these groups gives:

8	21	29	14	44	70	33	53	88	82	59
1	2	3	1	2	3	1	2	3	1	2

  - end of second pass, and so on

- in general, we can use any number of smaller increments, but the last increment must be 1
  - so that the last pass is regular linear insertion sort, but on an almost sorted list
  - this guarantees that final list is always sorted

- order of complexity analysis is difficult
  - shell sort is O($N^{5/4}$) (average case)
  - shell sort is O($N^{3/2}$) (worst case)
  - a very fast sort

- an implementation of shell sort

```
public void shell(int n)
{
    int temp;

    for (int incr = n / 2; incr > 0; incr = incr / 2)
        for (int i = incr; i < n; ++i)
            for (int j = i - incr;
                 j >= 0 && list[j] > list[j + incr];
                 j = j - incr) {
                temp = list[j];
                list[j] = list[j + incr];
                list[j + incr] = temp;
            }
}
```

Figure 9 an implementation of shell sort

- increment starts at half of list size, then halves each time

- difficult algo!

<u>Summary</u>

- from Canvas, 'Sorting' module, Example programs, download, read and run my `Sorting` example program

  - checkout the insertion sorts

  - just enjoy, try to understand

  - (no need to memorize the implementations!)

Selection sorts
Objective: last of our three kinds of sort. See some examples, understand general ideas

<u>Selection sorts select the smallest (or largest) unsorted item</u>

- "select smallest (or largest) unsorted item and add to a list of sorted items". Will look at three common selection sort algorithms:
  - straight selection sort
  - tree sort
  - heap sort

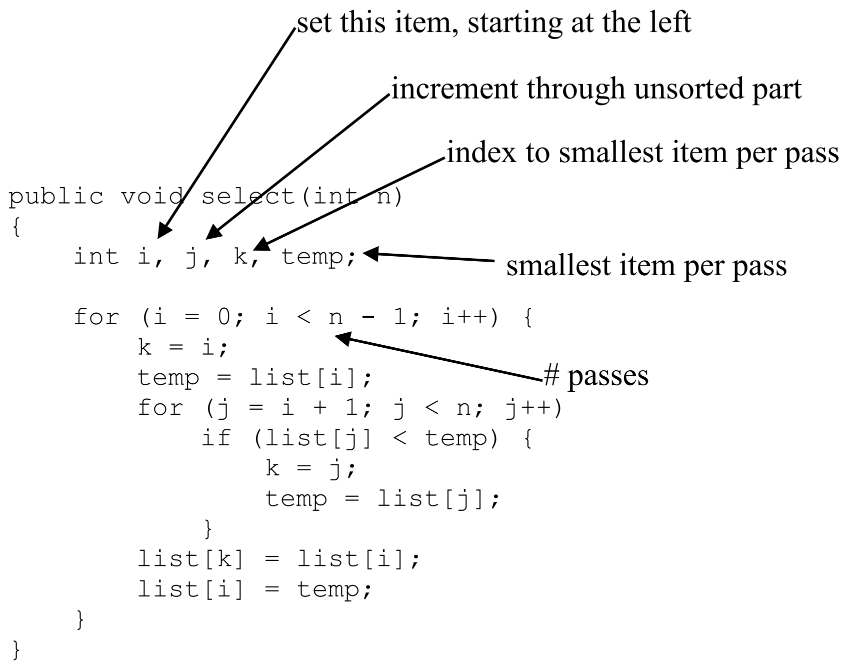
1. <u>straight selection sort</u>

- successively search list for the smallest item and add this to sorted items at beginning of list
  - e.g. create an insertion point | at the beginning of the list, for the sorted item:

```
|       53      22      7       15      84      63
                        ^

7       |       53      22      15      84      63
                                ^

7       15      |       53      22      84      63
                                ^

7       15      22      |       53      84      63
                                ^

7       15      22      53      |       84      63
                                                ^

7       15      22      53      63      |       84
                                                ^

7       15      22      53      63      84      |
```

- compare this with linear insertion sort:
  - insertion sort – takes first unsorted and inserts into sorted partition
  - selection sort – finds next smallest and appends to sorted partition

- imagine an animated display of both these algos sorting a list. At beginning of sort:
  - insertion seems faster, because it's inserting into a small list
  - at end of sorting:
  - selection looks faster, because it's searching a small list
- order of complexity
  - #passes = N
  - #visits = N/2
  - straight selection sort is O(N^2)
  - same as linear insertion sort
- an implementation of straight selection sort


set this item, starting at the left
increment through unsorted part
index to smallest item per pass
smallest item per pass
# passes

```
public void select(int n)
{
    int i, j, k, temp;

    for (i = 0; i < n - 1; i++) {
        k = i;
        temp = list[i];
        for (j = i + 1; j < n; j++)
            if (list[j] < temp) {
                k = j;
                temp = list[j];
            }
        list[k] = list[i];
        list[i] = temp;
    }
}
```

Figure 10 an implementation of straight selection sort

2. tree sort (also known as a tournament sort)

- begin by considering the list as the leaves of a binary tree and grow the tree upwards…
  - e.g. unsorted list

    6 5 7 9 4 2 8

  - grows to:

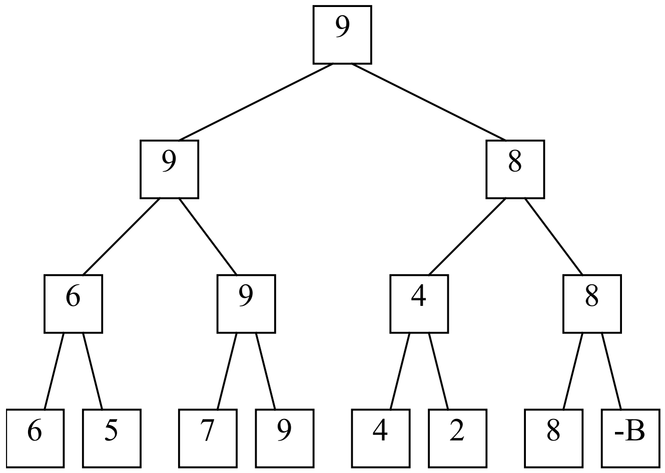


Figure 11 grow the tree upwards

  - notice that a special non-data item shown as –B here is used to complete the leaves
  - has a very negative value so that it loses every match
  - (-B is the closest I can come on the keyboard to showing 'negative infinity')
- from the tree, see how the largest value of each pair becomes the parent
  - hence the tournament sort name, as each winner of a match advances to the next round
- eventually the largest value will be at the root of the tree…
  - …so place the tree root at the end of the sorted list
  - then remove this value from every node in the tree by replacing it with the special –B value:

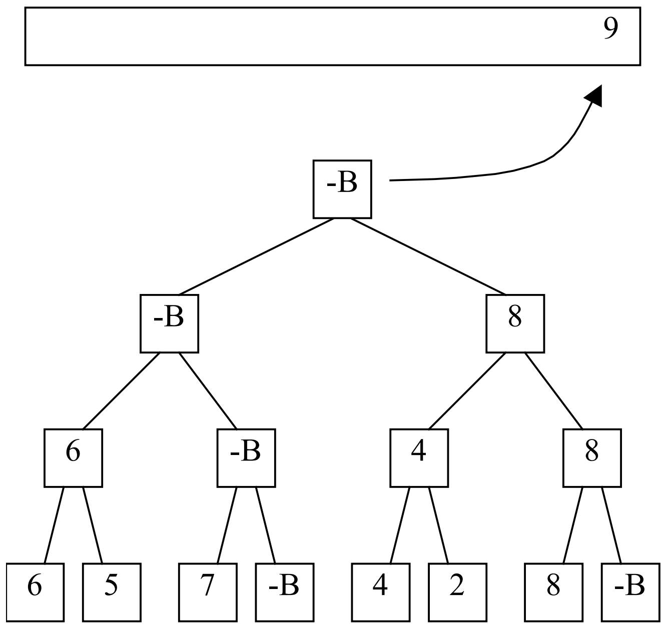


Figure 12 remove the winner

- now relabel nodes again to find a new root, and so on

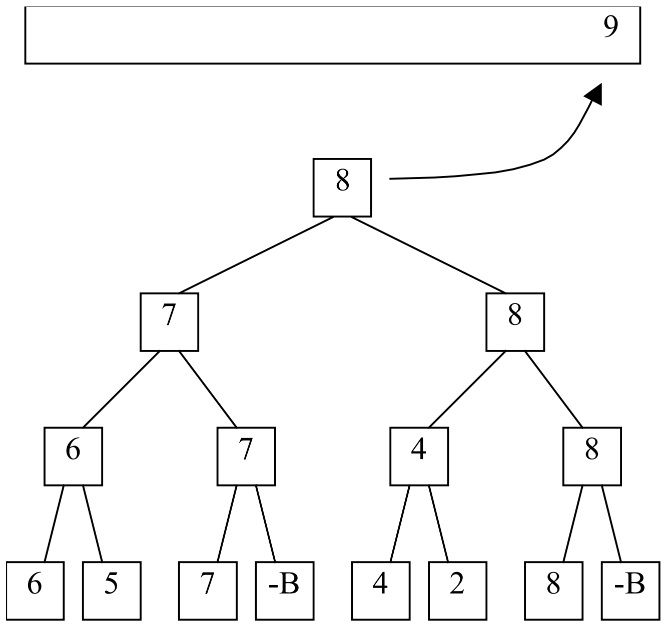


Figure 13 relabel to find the next winner

- order of complexity
  - #passes = N
  - #visits = logN

- tree sort is O(NlogN)

- very fast…

- …but needs lots of memory:

– #leaves	=	N	
– #internal nodes	=	N – 1	
– size of list	=	N	
– total	=	3N	memory locations!

- (compared with linear insertion, which required N + 1 locations!)

- (there's no implementation of tree sort)

- BTW: we know a different way to do sorting with a tree!

  - build a binary search tree from the data, then do an inorder traversal!

3. <u>heap sort</u>

- tree sort is fast, but may require too much memory. The heap sort is as fast, but can be done inside the list of numbers

  - a heap is a special case of a binary tree: is a "binary tree in which every parent is greater than its children"

  - so the root of a heap is always the largest value in the tree

- e.g. of a heap

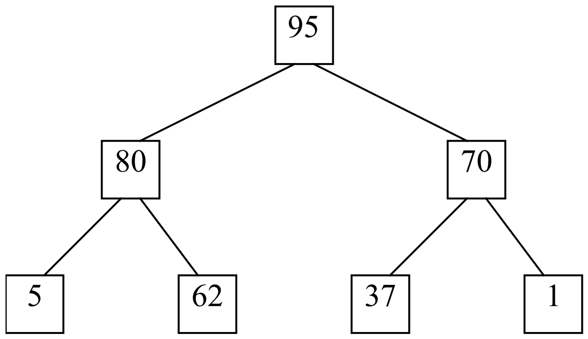


Figure 14 example of a heap

- note, is not a bst, since there is no requirement that lesser values go to the left

- so the heap sort algorithm is a loop:
  - create a heap
  - remove the root
  - repeat until the heap is empty
- can implement this heap sort inside an array, so that there's no requirement for extra memory
  - as follows
- start with an array of the random numbers that we want to sort e.g.

2	4	1	5	3	6	7
1	2	3	4	5	6	7

Figure 15 start with an array that we want to sort

  - assume that array indexes start from 1
- now conceptually build a tree from the array, <u>where the tree must be filled by level</u>
  - here gives (each tree node is labelled with its index from the array):

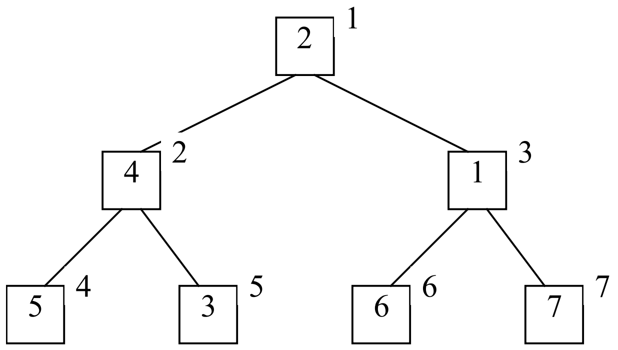


Figure 16 build a complete binary tree from the array

Canonical numbering

- this mapping from array index to tree node gives fast, direct access to parents and children in the heap. Is called 'canonical numbering'
  - if a node in the tree has index `i`, then:
  - left child is at `2i`
  - right child is at `2i + 1`
- we use this fast direct access:
  - for checking to see if a tree is a heap
  - and if not, for changing a tree towards a heap

Sifting

- given a random binary tree, can convert to a heap using a process called sifting:
  - if a tree is not a heap, "swap root with its largest child"
  - sifting happens from leaves toward root
  - then root towards leaves
  - until entire tree is a heap
  - (BTW, each swap will move items inside the array)
- e.g. starting from our random tree, first pass:

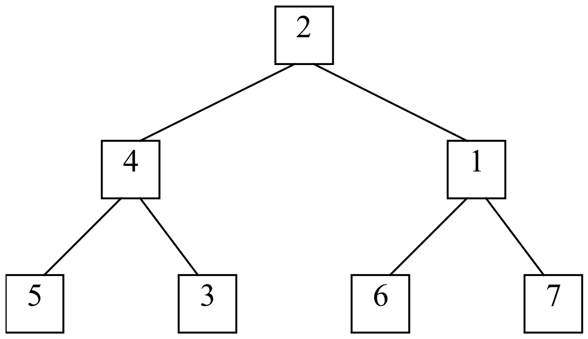


Figure 17 starting from our random tree, first pass

- starting from the leaves, sift any subtrees that are not heaps. Does left subtree first, to give:

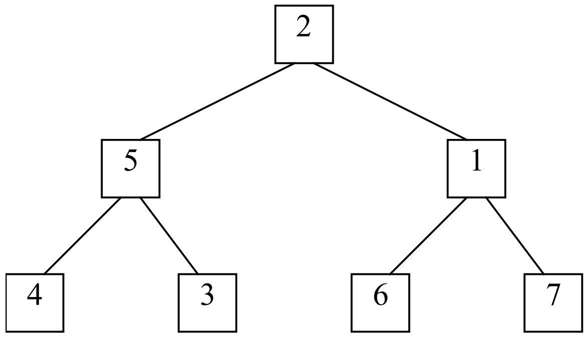


Figure 18 sift left subtree

- sift until tree is a heap. Does right subtree next:

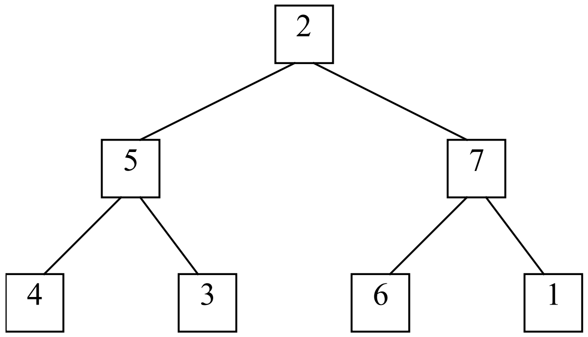


Figure 19 sift right subtree

- sift again. Now root's subtree:

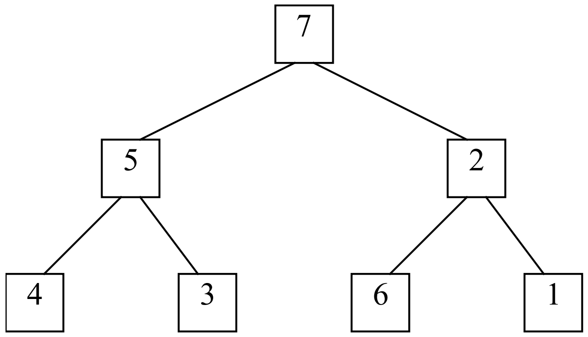


Figure 20 sift root's subtree

- still not a heap. Have to sift again, down towards leaves, right subtree:

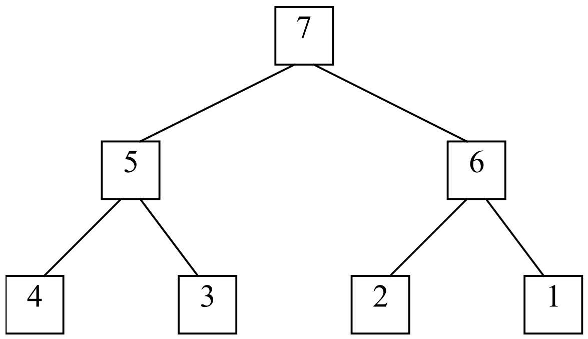


Figure 21 sift right subtree

- and finally have a heap!… remove root

<u>To remove the root of a heap</u>

- exchange root with <u>rightmost leaf</u>, and ignore this leaf in future i.e.:

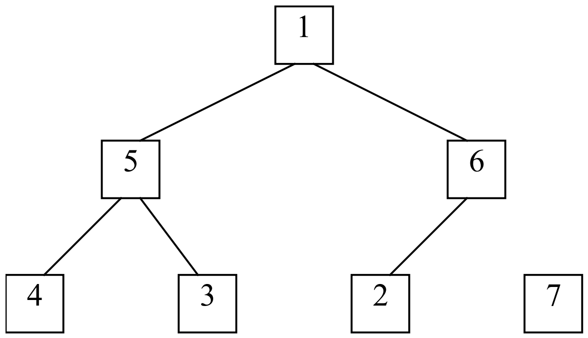


Figure 22 remove the root of the heap

  - end of the first pass

- now repeat the sifting process again until we get another heap. Second pass:
  - so, sifting from the leaves upwards, we sift the root subtree:

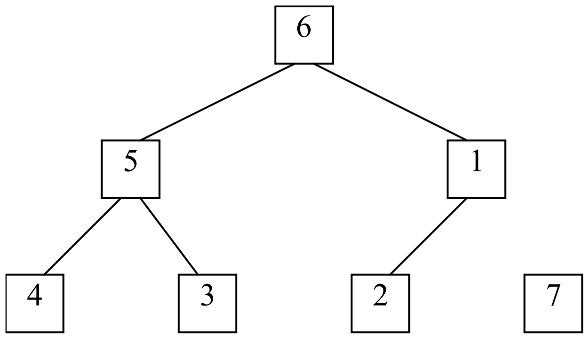


Figure 23 sift the root subtree

- sifting downwards. we sift again, right subtree:

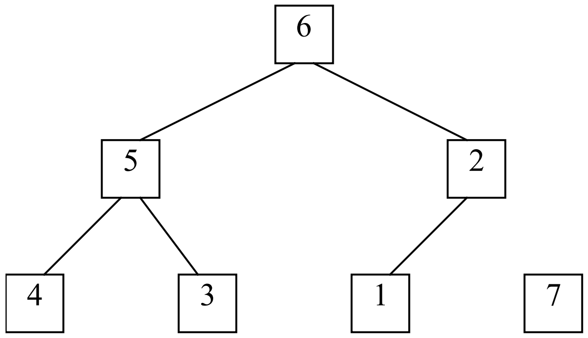


Figure 24 sift right subtree

- we have a heap! – exchange root with rightmost leaf, and ignore this leaf in future:

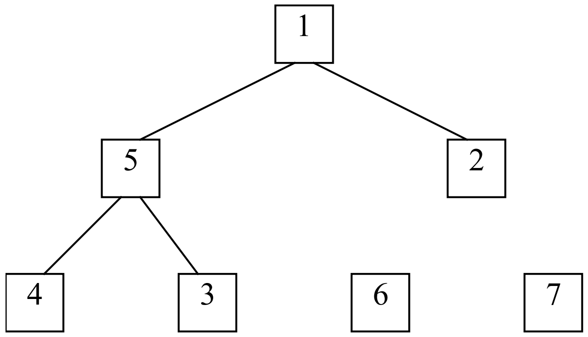


Figure 25 remove the root of the heap

- end of the second pass

• sift again until we get another heap. Third pass, sifting upwards, sift the root subtree:

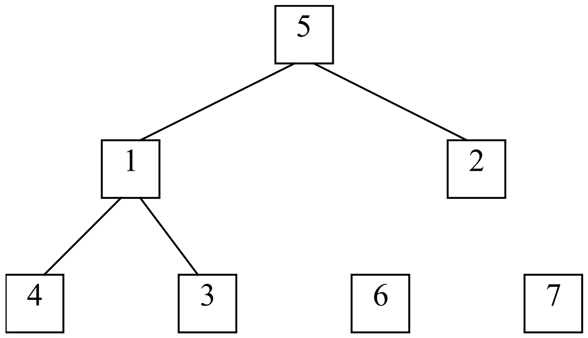


Figure 26 sift the root subtree

- sift again, left subtree:

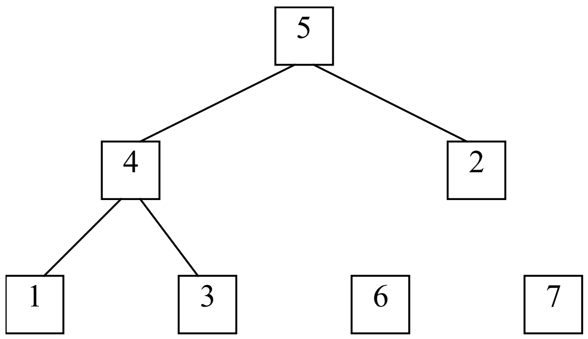


Figure 27 sift left subtree

- heap! – exchange root with rightmost leaf, and ignore this leaf in future:

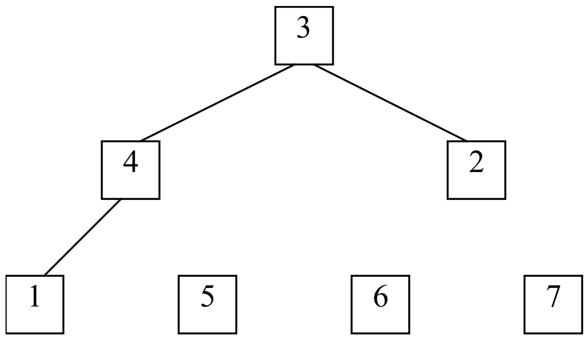


Figure 28 remove the root of the heap

- end of the third pass

- fourth pass. Sifting up, sift root subtree:

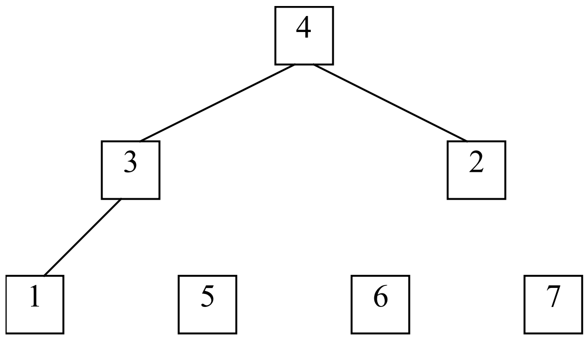


Figure 29 sift root subtree

- heap! – exchange root with rightmost leaf, and ignore this leaf in future:

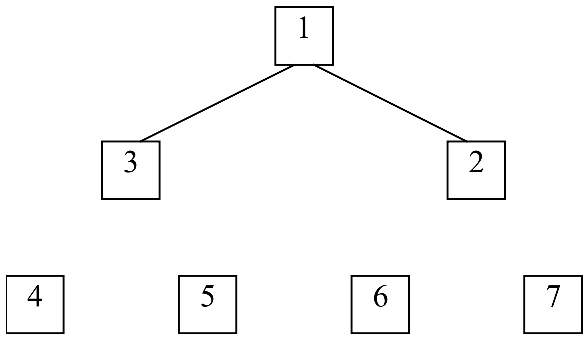


Figure 30 remove the root of the heap

- end of the fourth pass

- fifth pass. Sifting up, sift root subtree:

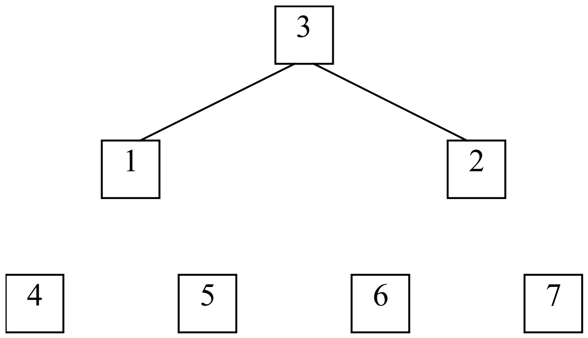


Figure 31 sift root subtree

  - heap!

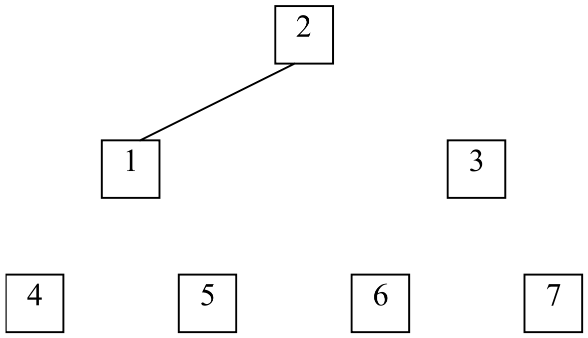


Figure 32 remove the root of the heap

  - end of the fifth pass

- sixth and last pass

  - no sift required here, already a heap!

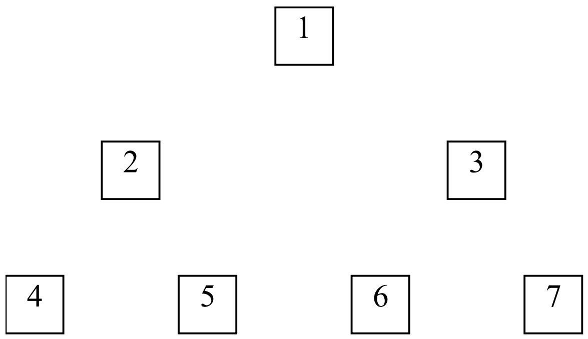


Figure 33 remove the root of the heap

- all done… array is sorted!

- order of complexity
  - heap sort is O(NlogN) (average and worst case)
  - same as tree sort, very fast
- but much less memory than tree sort!
  - heap sort inside the array using canonical numbering, doesn't need extra storage! – requires N locations + 1, in which to do the array swaping
  - remember, tree sort requires 3N locations
- an implementation of heap sort

```
public void heap(int n)
{
    int temp;

    for (int i = n / 2; i >= 0; i--)
        buildheap(i, n - 1);
    for (int i = n - 1; i >= 1; i--) {
        temp = list[0];
        list[0] = list[i];
        list[i] = temp;
        buildheap(0, i - 1);
    }
}

private void buildheap(int root, int i)
{
    int temp;
    int maxChild;
    Boolean isHeap = false;
    while ((root * 2 <= i) && !isHeap) {
        if (root * 2 == i)
            maxChild = root * 2;
        else if (list[root * 2] > list[root * 2 + 1])
            maxChild = root * 2;
        else
            maxChild = root * 2 + 1;

        if (list[root] < list[maxChild]) {
            temp = list[root];
            list[root] = list[maxChild];
            list[maxChild] = temp;
            root = maxChild;
        }
        else
            isHeap = true;
    }
}
```

Figure 34 an implementation of heap sort

- uses canonical numbering to do the sort inside the array

- tricky!

Summary

- from Canvas, 'Sorting' module, Example programs, download, read and run my `Sorting` example program

  - checkout the selection sorts

- just enjoy, try to understand
- (no need to memorize the implementations!)

Next week and homework

- next week: searching; Final exam

- homework by next week:

  - download, run and understand this week's example programs

  - read Horstmann, sections 14.6 – 14.8

  - to prepare for Final exam: re-read all my lectures and my example programs

  - review all of the programs you have written

  - re-read Horstmann textbook, Chapters 14 – 17

Lab

- check Canvas, Grades
- about the Final exam
- current lab was assigned in ‘Finish trees; hashing; graphs’. See Canvas for due date

Check Canvas, Grades

Objective: understand how your overall course letter grade is calculated, and check your scores in the Canvas gradebook

Grade weightings

- different components of this course have different weights, as published in Canvas, Course Information, the Syllabus document:

Homeworks	10%
Midterm	15%
Labs	60%
Final	15%
	100%

Overall course letter grade

- your overall course letter grade is determined by your weighted total:

weighted total	
90%	A
80%	B
70%	C
60%	D
< 60%	F

Check your scores in Canvas, Grades

- you must check all of your scores in Canvas, Grades
  - email proof ASAP of any corrections needed (awsmith@palomar.edu)

Your weighted total will be available when the last assignments have been graded

- your weighted total will be available in Canvas after the last labs have all been graded and the Final exam deadline has passed

About the Final exam
Objective: what to expect, how to prepare, to do super-well on the Final exam please!

Have a DRC Accommodations Request Form?

- YOU MUST email me ASAP (awsmith@palomar.edu) if you have a DRC Accommodations Request Form

Format of the online Final

- I will post the 'Final' module on Friday, then email everyone as usual. You take the Final exam online through Canvas, any time during the week. Due date will be given in Canvas. You must submit the exam before the end of that day
- format is the same as the online Midterm exam, will be 25 multiple-choice questions
- covers only the second half of the semester, everything since the Midterm exam
- strict 1 hour time limit once you start, cannot be exceeded
- you have to finish the exam once you've started. Cannot stop part way through and come back to it later
- must be your own independent work – do not collaborate or confer in any way
- the exam has to be taken inside a secure browser. Set-up instructions are given when you start the exam. Any set-up time does not count against the time limit
- is 15% of your overall course grade

Preparation completed so far

- every week you are required to print out then work through my written pdf lecture document, running and understanding my example programs where indicated
- you have done the assigned textbook reading
- you have completed the programming exercises and homeworks
- you have completed the graded programming labs

How to prepare

- the questions cover everything since the Midterm exam up to and including this week's material

- to prepare, review my lectures, my example programs and your labs in detail
  - every week, you have to print out my lecture, and study all of my example programs...
  - make sure you re-read all these lectures
  - make sure you understand every example program
- if you have time, then re-read Horstmann textbook, Chapters 14 – 17
  - review Horstmann's 'Self Check' questions
  - (answers are given at the end of each Chapter)

<u>Practice Final now posted</u>

- I've posted a Practice Final in 'Sorting'...
  - is intended to show you how the real Final will work...
  - contains only 5 questions – the real Final has 25
  - also, enables you to set-up and use the secure browser environment before the real Final
  - see Canvas for due date. Complete the Practice Final before the end of that day, to familiarize yourself with the process
- the scores on the Practice Final will be available in Grades by the end of the day after the deadline
- importantly, the Practice Final has absolutely no effect on your overall course grade!