page 364 of textbook please

Imp Concepts

Traversals

Inorder - LRoR
Preorder: RoLR
Postorder : LRRo

Intro To Sorting

given a set of records $(r_1, \ldots r_n)$ w/ key values $(k_1, \ldots k_n)$
arrange them in order s
such that records $r_{s1}, r_{s2}, \ldots r_{sn}$
have keys w/ property $k_{s1} \leq k_{s2} \leq \ldots \leq k_{sn}$

* in this course, assume increasing by default

when arranging nos in increasing:
we would find the smallest and then shift:
we have a global view of all the numbers.
A computer does not

- why do we learn sorting:
  → easier to find an obj. in a large collection
  → building block for algorithms
  → every single algo can be reduced to sorting algo (run on turing machine)

- safe to use big O always (binary: $O(\log n)$, sequential: $O(n)$)

- O/P of any sorting algorithm is an ordered pair (key, record)

- sorting → internal ← focus of this course
  ↓
  external

problem: are given sets disjoint?

soln 1: compare each element in A w/ each element in B
↳ n^2 comparisons

soln 2: sort first set (look up $O(n \ln n)$ Merge, Heap sort)
then compare w/
binary sort
↳ $O(n \lg n)$

- sorting has 2 actions: key comparisons and swapping records
  becomes a problem of finding an element in a sorted ($O(n \lg n)$) array → binary search.
  ↳ $O(n \lg n)$

together: $O(n \lg n)$

MAIN OBJECTIVE: min usage of memory
min number of key comparisons / swaps

2000
2002
2005
2001
cc0007

from the book

Sequential sort :

```
procedure SEQSRCH (F, n, i, K)
// search a file F with key values K1, .... Kn for a record Ri
// such that Ki = K. If there is no such record, i is set to 0
   K0 ← K ; i ← n
   while Ki ≠ K do
      i ← i - 1
   end
end SEQSRCH
```

introduce dummy record R_0 with key $K_0 = K$

- no need to keep test of $i < I$
- cuts time by 50% for large n
- worst case = $(n+1)$
- best case = 1
- avg case :

for arbitrary i : $(n-i+1)$ comparisons

avg : $\dfrac{\sum_{i=1}^{n} (n-i+1)}{n} = \dfrac{(n+1)}{2}$

$\sum 1$ $\quad 1+1+1 \ldots \rightarrow n$ $\qquad \sum_{i=1}^{n} n = n^2$

$\sum_{1=1}^{n} (-i) \rightarrow -\dfrac{(n)(n+1)}{2}$

$\dfrac{n + n^2 - n(n+1)/2}{n} = (n+1) - \dfrac{(n+1)}{2} = \left(\dfrac{n+1}{2}\right)$

Insertion Sort

(1) Incremental approach

↳ given an unordered set of objects, repeatedly remove an entry and insert it into a new ordered list which remains ordered always

(2) Code : within the set itself ⇒ given an array, sort within the array itself

given
```
class ALIST {
    key k;
    data d;
}
```

```
void InsertionSort ( ALIST slot[] , int n) {

  for ( int i = 1 , i < n ; i++ ){ // assume first is sorted; n = no of items
    for (int j = i ; j > 0 ; j--) {

      if ( slot[j].key < slot[j-1].key )
          swap(slot[j] , slot[j-1]);
      else
          break ;  // assumes prev is sorted
  }
```

⇑

ordered remains ordered.

③ Out-place sorting

would occur in lists. we don't keep swapping.
would be faster, but more expensive

④ Time Complexity

no of key comp:
- n-1 iterations ← outer loop, unavoidable.

$\Theta(n)$

- best case: 1 key comp per iteration (⇒ already sorted)
  worst case: $\sum_{i=1}^{n-1} i = \frac{(n-1)(n)}{2} = \frac{n^2}{2} - \frac{n}{2}$ $\Theta(n^2)$

avg case: i^{th} iteration may have 1, 2 ... i key comp each with a '1/i' chance of occuring

$\frac{1}{i} \sum_{j=1}^{i} j$ ← avg no of comp for i^{th} iteration

$\sum_{i=1}^{n-1} \frac{1}{i} \sum_{j=1}^{i} j$ ← sum through all iterations

$1 + \frac{1}{2}(1+2) + \frac{1}{3}(1+2+3) \cdots + \frac{1}{n-1}(1+\cdots+n-1)$

$= \sum_{i=1}^{n-1} \left(\frac{1}{i} \frac{i(i+1)}{2} \right) = \frac{1}{2} \sum_{i=1}^{n-1} i+1 = \frac{1}{2} \left(\frac{(n-1)(n+2)}{2} \right)$

$= \Theta(n^2)$

⑤ Strengths

- best for verifying / sorting almost sorted
- no movement of data → fast w/ linked storage implementation

⑥ Weaknesses

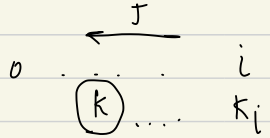
- inserting is hard, esp when slots are for large amounts of data, movement is expensive
  ↳ hence for contiguous data structures, it is tough.

from the text + practice.

① Psuedocode for insertion only

// insert record R w/ key K into ordered sequence $R_0, \ldots R_i$ in such a way that the resulting sequence is also ordered. Assume R_0 is a dummy record w/ key $K_0 = -\infty$ $(K_i \geq K_0)$

```
procedure INSERT (R, i)
    j ← i
    while K < Kj do      // move Rj one space up as R is to be inserted left of Rj //
    Rj+1 ← Rj ; j ← j-1
    end                  → at this point, there's an empty space at 'j+1'
Rj+1 ← R
end Insert
```



(2) the time complexity of INSERT is $O(n)$ for worst case

(3)
```
procedure INSORT (R, n)
// sort the records R_1, ... R_n in increasing value of key k
K_0 ← -∞      // dummy record R_0 : K_0 < K_i
for j ← 2 to n do          // assuming R_{j-1} is sorted
    T ← R_j
    call INSERT (T, j-1)
end
end INSORT
```

(4) here, insert is called $(n-1)$ times

worst case : for every call, we go to front. (i for i^{th})

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \rightsquigarrow O(n^2)$$

best case : for every call, we add at end only.

$$\sum_{i=1}^{n-1} 1 = n-1 \rightsquigarrow O(n)$$

avg case : for every call, each spot is equally possible:

i^{th} call : $\sum_{j=1}^{j=i} j/i = \frac{i(i+1)}{i \cdot 2} = \frac{(i+1)}{2}$

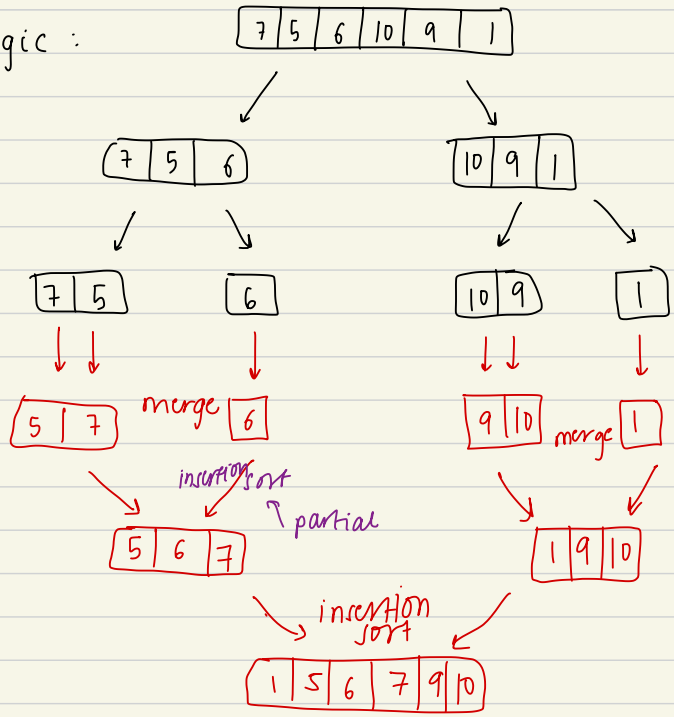
and theres $n-1$ calls : $\sum_{i=1}^{n-1} \sum_{j=1}^{j=i} j/i = \sum_{i=1}^{n-1} \frac{(i+1)}{2}$

$$= \frac{\frac{(n-1)(n)}{2} + (n-1)}{2} = \frac{(n-1)(n+2)}{4} \rightsquigarrow O(n^2)$$

try to implement w/ linked list

Merge Sort

① Logic :



② Psuedocode

```
solve (problem of size n) {
        if (n <= minimum size)
                solve directly
        else {
            divide the problem into p1 .... pk problem;
            for each pi
              soln = solve(pi)
            combine all solutions.
        }
}
```

③

```
mergeSort(list) {
    if (length of list > 1) {
        partition list in 2 approx. equal sized lists L1, L2;
        mergesort (L1);
        mergesort (L2);
        merge sorted L1, L2;

    }
}
```

④ exact psuedocode

```
void mergesort (int n, int m) {
   int mid = (n+m)/2;
   if (m-n <= 0)
       return;
   else if (m-n > 1) {
      mergesort (n, mid);
      megesort (mid+1, m);
   }
   merge (n, m)
}
```

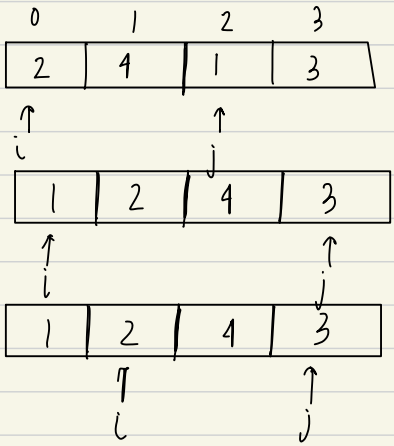
- for partial insertion sort, start from smaller no. (for ascending)

FROM THE TEXTBOOK

① merging 2 already sorted files

```
procedure MERGE (X, l, m, n, Z)

// (X_l .... X_m) and (X_{m+1} ... X_n) are 2 sorted files with keys
x_l ≤ .... ≤ x_m and x_{m+1} ≤ ... ≤ x_n. merged to obtain
sorted file (Z_l, .... Z_n) such that z_l ≤ ... ≤ z_n //

i ← K ← l, j ← m+1   // i, j, K are position in the three files

while i ≤ m and j ≤ n do
   if x_i ≤ x_j then [Z_K ← X_i ; i ← i+1]
                else [Z_K ← X_j ; j ← j+1]
   K ← K+1
end
if i > m then (Z_K, .... Z_n) ← (X_j, .... X_n)
         else (Z_K, .... Z_n) ← (X_i, .... X_m)

end MERGE
```

- no of records being sorted : $n - l + 1$



$\frac{0+3}{2} = 1$

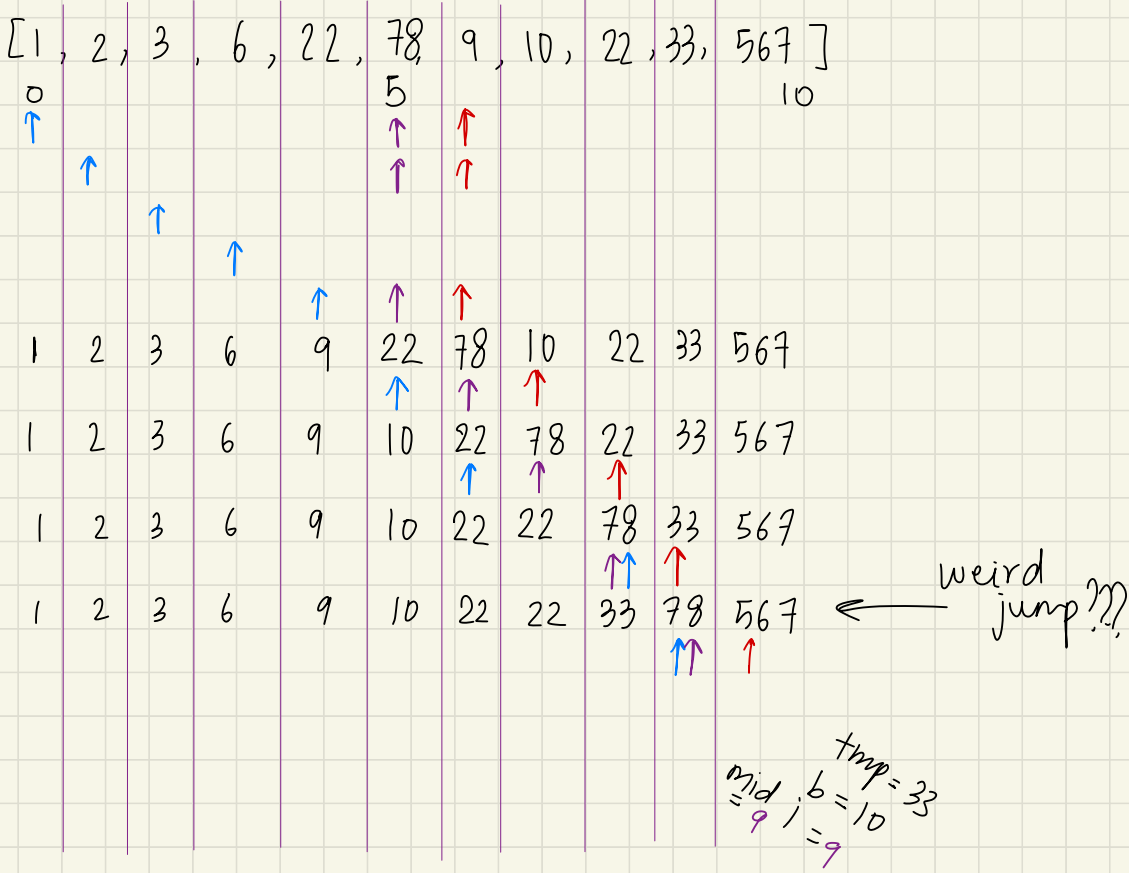
$n = 0$

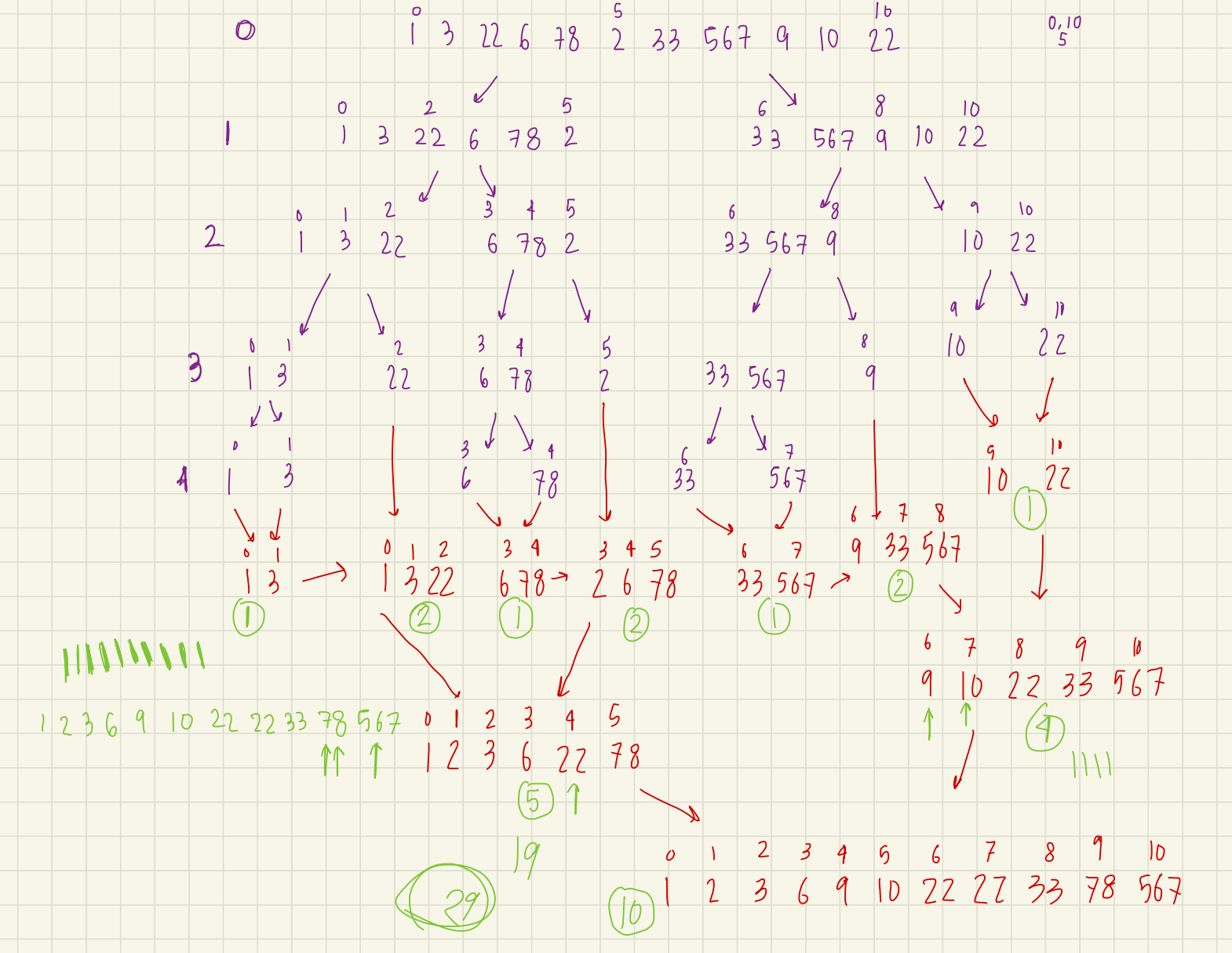
$m = 3$

$mid = 1$

obs:
b and mid always increase together
∴ always together
and a is always ≤ mid
↘ =?

when equal,
a advances by 2
b & mid, by 1




0
1 3 22 6 78 2 33 567 9 10 22
1
1 3 22 6 78 2
33 567 9 10 22
2
1 3 22
6 78 2
33 567 9
10 22
3
1 3
22
6 78
2
33 567
9
10
22
4
1 3
6 78
33
567
10
22
1 3
1 3 22
6 78
2 6 78
33 567
9 33 567
1 2 3 6 9 10 22 22 33 78 567
1 2 3 6 22 78
9 10 22 33 567
29
1 2 3 6 9 10 22 22 33 78 567
10

Complexity of merge()

- after each key comparison, at least 1 element is moved to the new merged list, never to be compared again
- after the last comparison, at least 2 elements are moved....
- so worst case: for 2 sorted list w/ n elements total at most $n-1$ comparisons are needed.

- mergesort as a whole

time complexity

$$T(n) = T(n/2) + T(n/2) + \overbrace{T(\text{merge}(n))}^{n-1}$$
$$T(n) = 2T(n/2) + n - 1$$
$$\vdots$$
$$T(1) = 0$$
$$T(n) = 2\left(2T(n/4) + \frac{n}{2} - 1\right) + n - 1$$
$$= 4T(n/4) + n - 2 + n - 1$$
$$= 4\left(2T(n/8) + \frac{n}{4} - 1\right) + 2n - 2 - 1$$
$$= 8T(n/8) + n - 4 + 2n - 2 - 1$$
$$= 8T(n/8) + 3n - 4 - 2 - 1$$
$$= nT(1) + n\log_2 n - \sum_{i=0}^{\log_2 n - 1} 2^i$$

$\log_2 n$ terms

geometric series $\rightarrow -(1 + 2 + 4 + \dots + 2^{\log_2 n - 1})$

$$= n\log_2 n - (2^{\log_2 n} - 1)$$
$$= n\log_2 n - (n - 1)$$

→ should also be worst case no of comparisons

HybridSort

At array size = threshold S, call insertion sort.

time complexity analysis on the next page.

stop when the array size = S

when $\frac{n}{2^m} = S$, we stop

$\therefore \boxed{\frac{n}{S} = 2^m}$

$S : 2^{k-m}$

$m : \log_2(n/s)$

let $n = 2^k$

$T(n) = 2T(n/2) + T(\text{merge})$ ← $n/2$

$T(n) = 2^m T(S) + \log_2(2^m)(n/2)$

$T(n) = 2^m T(S) + m \cdot (n/2)$

↳ time for insertion sort : best case $= (S-1)$

$T(n) = 2^m (S-1) + m(n/2)$

$T(n)$

$= \left(\frac{n}{S}\right)(S-1) + \frac{n}{2}\log_2\left(\frac{n}{S}\right)$

$= \left(n - \frac{n}{S}\right) + \frac{n}{2}\log_2\frac{n}{S}$

$= O(n) + O\left(n \log_2\left(\frac{n}{S}\right)\right)$

$= O\left(n + n\log_2\frac{n}{S}\right)$

---

key :

total elements = $n = 2^k$

threshold = $S = \frac{n}{2^m}$

2^m = number of split arrays

$k = \log_2 n$

$m = \log_2\left(\frac{n}{S}\right)$ $\quad \because 2^m = \frac{n}{S}$

worst case

$2^m = \frac{n}{s}$

$s = \frac{n}{2^m}$

$T(n) = 2T(n/2) + (n-1)$

$T(n) = 4T(n/4) + n + n - 2 - 1 = 4T(n/4) + 2n - 2 - 1$

$T(n) = 8T(n/8) + 3n - \underbrace{4 - 2 - 1}_{\hookrightarrow 2^{3-1} - 1}$

$\vdots$

$T(n) = 2^m T(s) + \log_2(2^m) n - (2^m - 1)$

$\hookrightarrow \frac{s(s-1)}{2}$

$T(n) = \frac{n}{s}\left(\frac{s(s-1)}{2}\right) + mn - 2^m + 1$

$= \frac{n}{s}\left(\frac{s(s-1)}{2}\right) + n\log_2\left(\frac{n}{s}\right) - \frac{n}{s} + 1$

$= \left(\frac{ns}{2} - \frac{n}{2} - \frac{n}{s} + 1\right) + n\log_2\left(\frac{n}{s}\right)$

$= O(ns) + O\left(n\log_2 \frac{n}{s}\right)$

$= O\left(ns + n\log_2 \frac{n}{s}\right)$

$\frac{16 \times 5}{2} - 8 - 3 + 1 + 16\log_2(3 \cdot 2)$

$40 + (8 - 3) + 1 + 5$

$\cdot 5 \qquad \cdot 5$

46 ← 47

mergesort v/s quicksort

So why on earth is quicksort faster than merge sort? This has to do with other factors that have nothing to do with the number of comparisons made. Primarily, since quicksort works in place while merge sort works out of place, the locality of reference is not nearly as good in merge sort as it is in quicksort. This is such a huge factor that quicksort ends up being much, much better than merge sort in practice, since the cost of a cache miss is pretty huge. Additionally, the time required to sort an array doesn't just take the number of comparisons into account. Other factors like the number of times each array element is moved can also be important. For example, in merge sort we need to allocate space for the buffered elements, move the elements so that they can be merged, then merge back into the array. These moves aren't counted in our analysis, but they definitely add up. Compare this to quicksort's partitioning step, which moves each array element exactly once and stays within the original array. These extra factors, not the number of comparisons made, dominate the algorithm's runtime.

try to implement with bst

Quicksort

- fastest general purpose in-memory sorting algorithm in the average case
- implemented in unix as qsort()

- steps:
- select 1 element in array as pivot
- partition into 2 sublists with pivot, such that all elements in left sublist are less than the pivot; all elements in right sublist are greater / equal.
- recursively partition until input list has 1/0 element
- no merging required since the pivot found during partitioning is already at its final position

FROM THE TEXT

Key K_i is placed in position $s(i)$ wrt the rest of the file
- so, $\forall\ j < s(i)$, $K_j < K_{s(i)}$ and $K_j \geq K_{s(i)}$ for $j > s(i)$
- then, file is split into the two, which are then sorted independently

n = no of elements

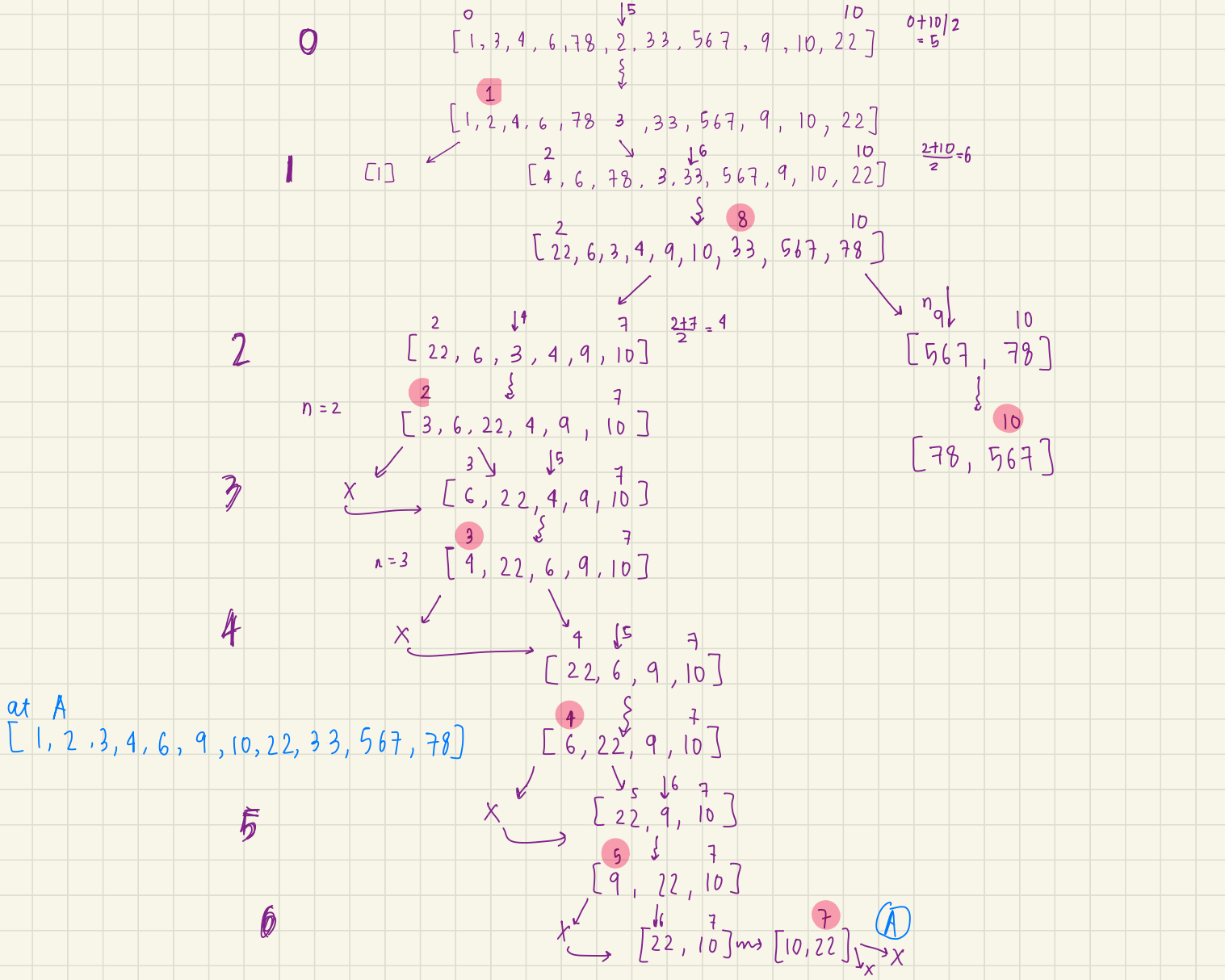
start with 0, n−1

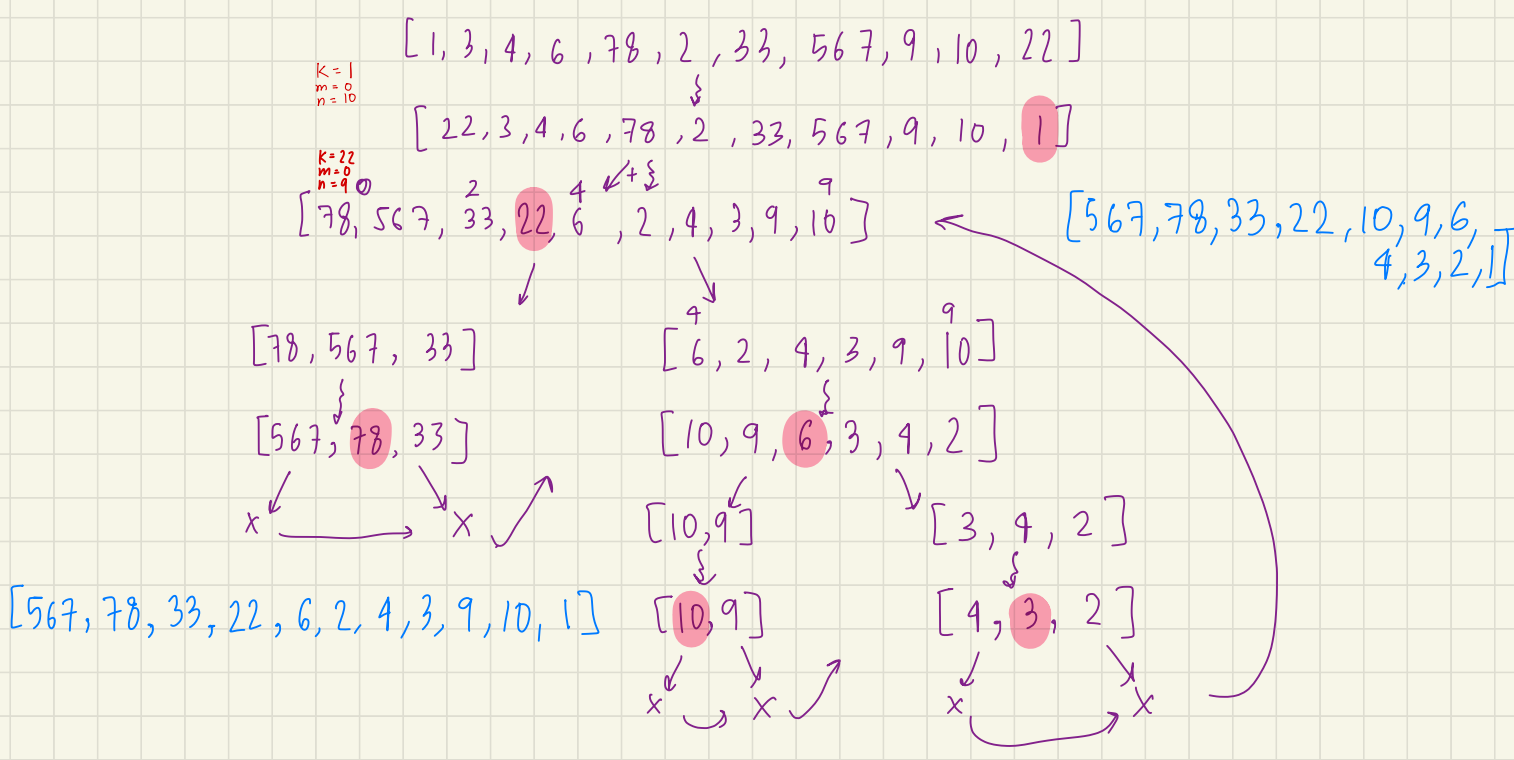
```
procedure QSORT(m, n)
// sort records R_m, .... R_n into nondecreasing order on key
K. Key K_m is arbitrarily chosen as the control key. Pointers
i and j are used to partition the subfile so that, at any
time K_l ≤ K (l < i) and K_l ≥ K (l > j).
Assuming K_m ≤ K_{n+1} //
                                   check base case
if m < n
                  low        high
  then [i ← m ; j ← n+1 ; K ← K_m
    loop
      repeat i ← i+1 , until K_i ≥ K, ≈ do (i←i+1) while (K_i < K)
      repeat j ← j−1 , until K_j ≤ K ; ≈ do (j←j+1) while (K_j > K)
      if i < j
          then call INTERCHANGE (R(i), R(j))
          else exit
    forever
    call INTERCHANGE (R(m), R(j))
    call QSORT (m, j−1)
    call QSORT (j+1, n )]
end QSORT

procedure INTERCHANGE (x, y)
    t ← x
    x ← y
    y ← t
end INTERCHANGE
```

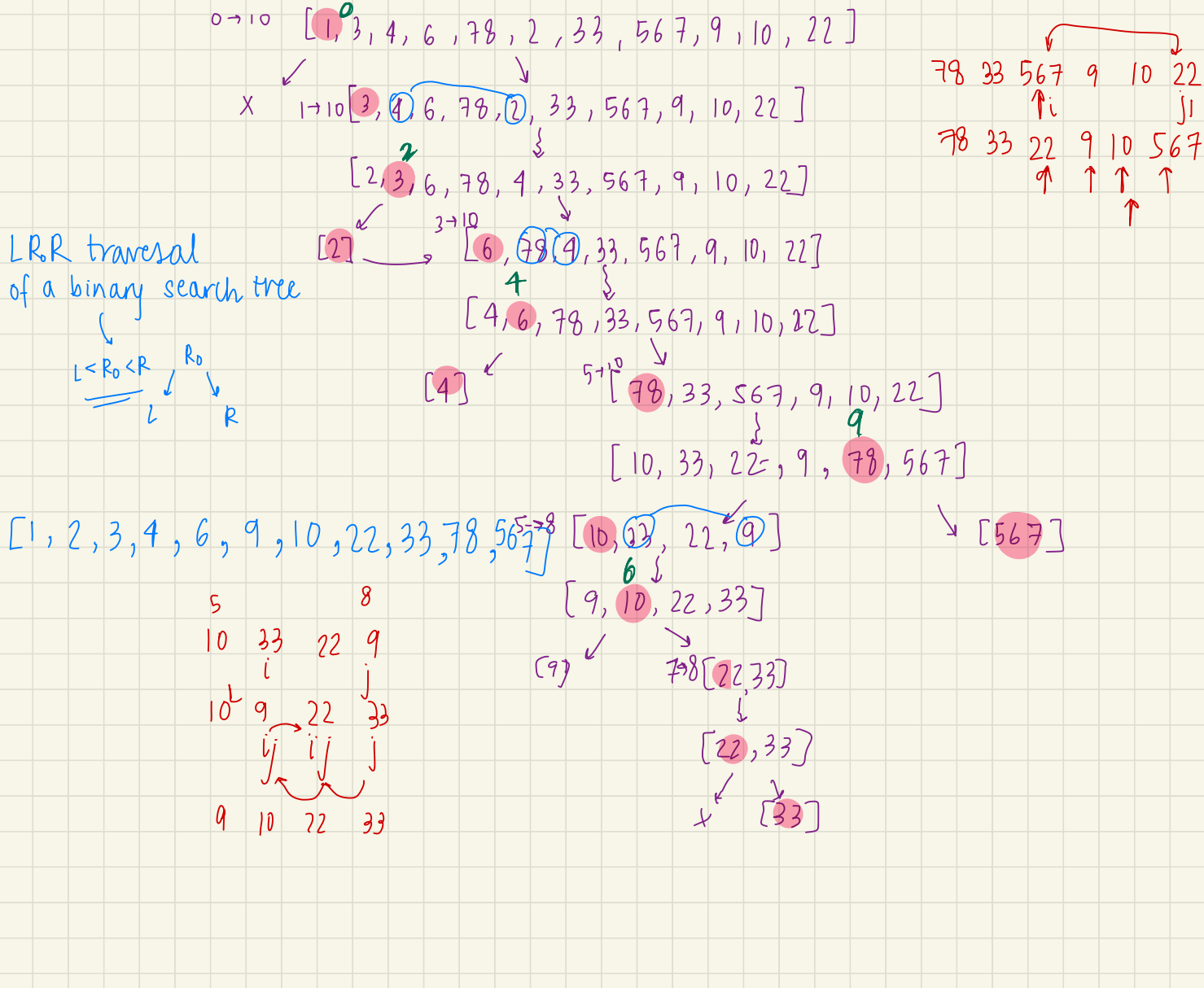
?

FROM THE NOTES

```
Void quicksort (int n, int m) {          <- just divides
    int pivot_pos;
    if (n >= m)
        return;
    pivot_pos = partition (n, m);
    quicksort (n, pivot_pos - 1);
    quicksort (pivot_pos+1, m);
}                                        <- conquers

int partition (int low, int high){       <- work girl!
    int i, last_small, pivot;            <- to scan; value of pivot
    int mid = (low + high)/2;            <- index of pivot
    swap (low, mid);                     <- move pivot to left most
    pivot = array[low];                  <- value of pivot : it is at low
    last_small = low;                    <- at beginning, left subclist is
    for (i=low+1; i<=high; i++)             empty
        if (array[i] < pivot)
            swap (++last_small, i);      <- value here is > pivot
    swap (low, last_small);              <- return pivot to its place
    return last_small;                   <- return pivot position
}
```


0
[1, 3, 4, 6, 78, 2, 33, 567, 9, 10, 22]
0+10/2 = 5
1
[1, 2, 4, 6, 78, 3, 33, 567, 9, 10, 22]
[1]
[4, 6, 78, 3, 33, 567, 9, 10, 22]
2+10/2=6
[22, 6, 3, 4, 9, 10, 33, 567, 78]
2
[22, 6, 3, 4, 9, 10]
2+7/2 = 4
[567, 78]
[78, 567]
n=2
[3, 6, 22, 4, 9, 10]
3
[6, 22, 4, 9, 10]
n=3
[4, 22, 6, 9, 10]
4
[22, 6, 9, 10]
at A
[1, 2, 3, 4, 6, 9, 10, 22, 33, 567, 78]
[6, 22, 9, 10]
5
[22, 9, 10]
[9, 22, 10]
6
[22, 10]
[10, 22]
A


[1, 3, 4, 6, 78, 2, 33, 567, 9, 10, 22]
k = 1
m = 0
n = 10
[22, 3, 4, 6, 78, 2, 33, 567, 9, 10, 1]
k = 22
m = 0
n = 9
[78, 567, 33, 22, 6, 2, 4, 3, 9, 10]
[567, 78, 33, 22, 10, 9, 6, 4, 3, 2, 1]
[78, 567, 33]
[567, 78, 33]
[6, 2, 4, 3, 9, 10]
[10, 9, 6, 3, 4, 2]
[10, 9]
[10, 9]
[3, 4, 2]
[4, 3, 2]
[567, 78, 33, 22, 6, 2, 4, 3, 9, 10, 1]


0→10
[1, 3, 4, 6, 78, 2, 33, 567, 9, 10, 22]
X
1→10 [3, 4, 6, 78, 2, 33, 567, 9, 10, 22]
[2, 3, 6, 78, 4, 33, 567, 9, 10, 22]
[2]
3→10 [6, 78, 4, 33, 567, 9, 10, 22]
[4, 6, 78, 33, 567, 9, 10, 22]
[4]
5→10 [78, 33, 567, 9, 10, 22]
[10, 33, 22, 9, 78, 567]
[567]
[9, 10, 22, 33]
[9]
[22, 33]
[22, 33]
[33]
78 33 567 9 10 22
i j
78 33 22 9 10 567
LRoR traversal
of a binary search tree
L<Ro<R
Ro
L
R
[1, 2, 3, 4, 6, 9, 10, 22, 33, 78, 567]
5→8
5 8
10 33 22 9
i j
10 9 22 33
ij i j
9 10 22 33

Quicksort Analysis

1. Worst case (stack space : $O(n)$) ↙ can be reduced cause size 2 need not be stored.

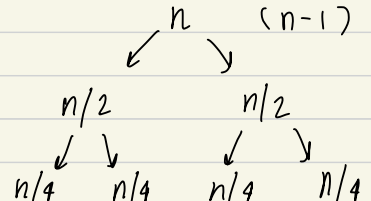
↳ every pivot is the smallest / largest value
↳ for iteration with n elements, $n-1$ comparisons
↳ each iteration, one element is dropped.

$$\sum_{i=n}^{1} i - 1 = \sum_{i=1}^{n} i - 1 = \sum_{i=1}^{n} i - \sum_{i=1}^{n} 1$$

$$= \frac{n(n+1)}{2} - n = \frac{n^2 - n}{2} = \frac{n(n-1)}{2}$$

$O(n^2)$

2. Best case → (stack space ($O(\log n)$)) → each pivot is the median of values being examined., still $n-1$ comparisons

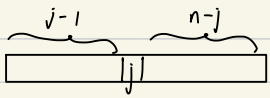


$T(n)$ = time to sort a file of n records
n = time to position a record correctly

$$\therefore T(n) \leq cn + 2T(n/2)$$
$$\leq cn + 2(cn/2 + 2T(n/4))$$
$$\leq 2cn + 4T(n/4)$$
$$\vdots$$
$$\leq cn \log_2 n + nT(1) = O(n \log_2 n)$$

check textbook (MIT book) Thomas H. Cormen
Charles E. Leiserson

3. Avg case : $\theta(n \log n)$



calling QSORT(1, n) , k_1 gets placed at j
so we have to sort 2 files : j-1 and n-j
↳ expected time for that is $T_{avg}(j-1) + T_{avg}(n-j) + cn$
j can take any value from 1 to n with equal probability

$$T_{avg}(n) \leq cn + \frac{1}{n} \sum_{j=1}^{n} (T_{avg}(j-1) + T_{avg}(n-j)) \quad n \geq 2$$

dynamically maintain local maximum

Heap Sort

- mergesort has time for worst case & average case = $O(n \log n)$ BUT requires additional storage $\propto$ no. of records being sorted

- heapsort only has a fixed amount of additional storage + time complexity for worst & average case = $O(n \log n)$

- file ⇝ binary tree or HEAP

h = height

↓ at least till depth $h-1$

also called (maximising) partial order tree

a complete binary tree w/ value of each node being equal to or greater than the value of its child

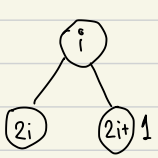
⇓ root of the tree is the largest value

AND : all paths to leaves of depth h are to the left of all paths to a leaf of depth $h-1$

- HEAPSORT : step 1. create heap
  step 2. gen. output sequence in decreasing order by successively outputting the root and restructuring the remaining tree back into a heap

- important tree facts:

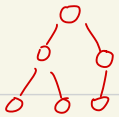
for n elements → n nodes (from 1 to n) → height max = $\log_2 n + 1$



so. If child is at location i parent is at location $\lfloor i/2 \rfloor$ and $1 < i \leq n$

'i' is a leaf iff $2i > n$

but the initial heap is filled in level order of array



the heap can be seen as an array (starting from 1) with level order traversal

need

constructheap (array, H) ← to create heap from given elements
heapifying (H)
deleteMax (H) ← obvious
fixHeap (H, k) ← restore heap after deletion
↳ iterative/recursive
↳ array or heap. (wrong understanding, heap is an array)
heapsort() ← sorts array in place (runs whole show)

```
constructHeap( array , H )
{
    put all elements of array into a heap structure H in
    arbitrary order ;   ← level order (literally just copy it)
    heapify(H)
}

heapify (H)
{
    if (H is not leaf) {
        heapify ( left subtree )
        heapify (right subtree)
        k = root(H) ;
        fixHeap (H, k) ;
}
```

post order traversal.

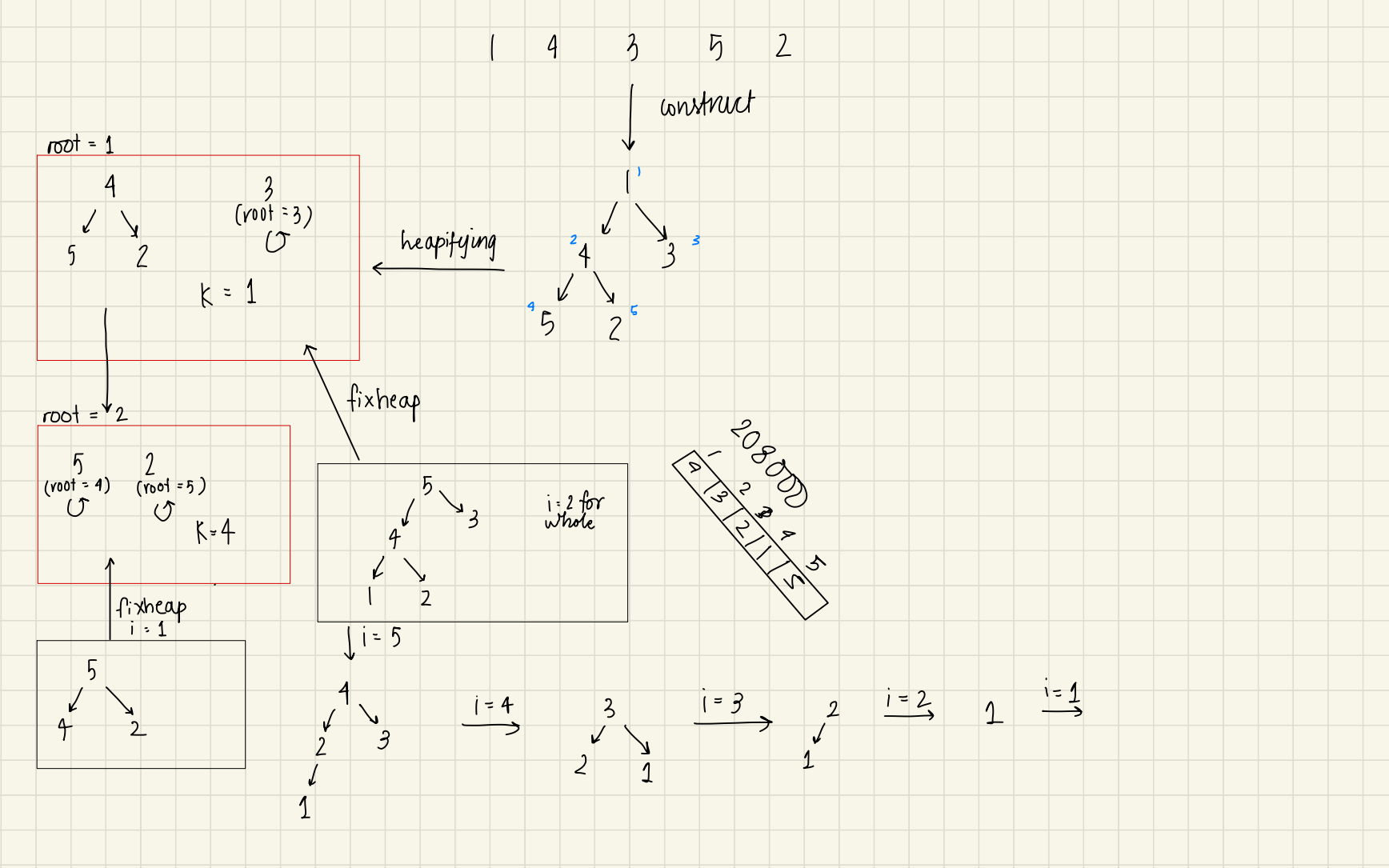
because we do L R R_0

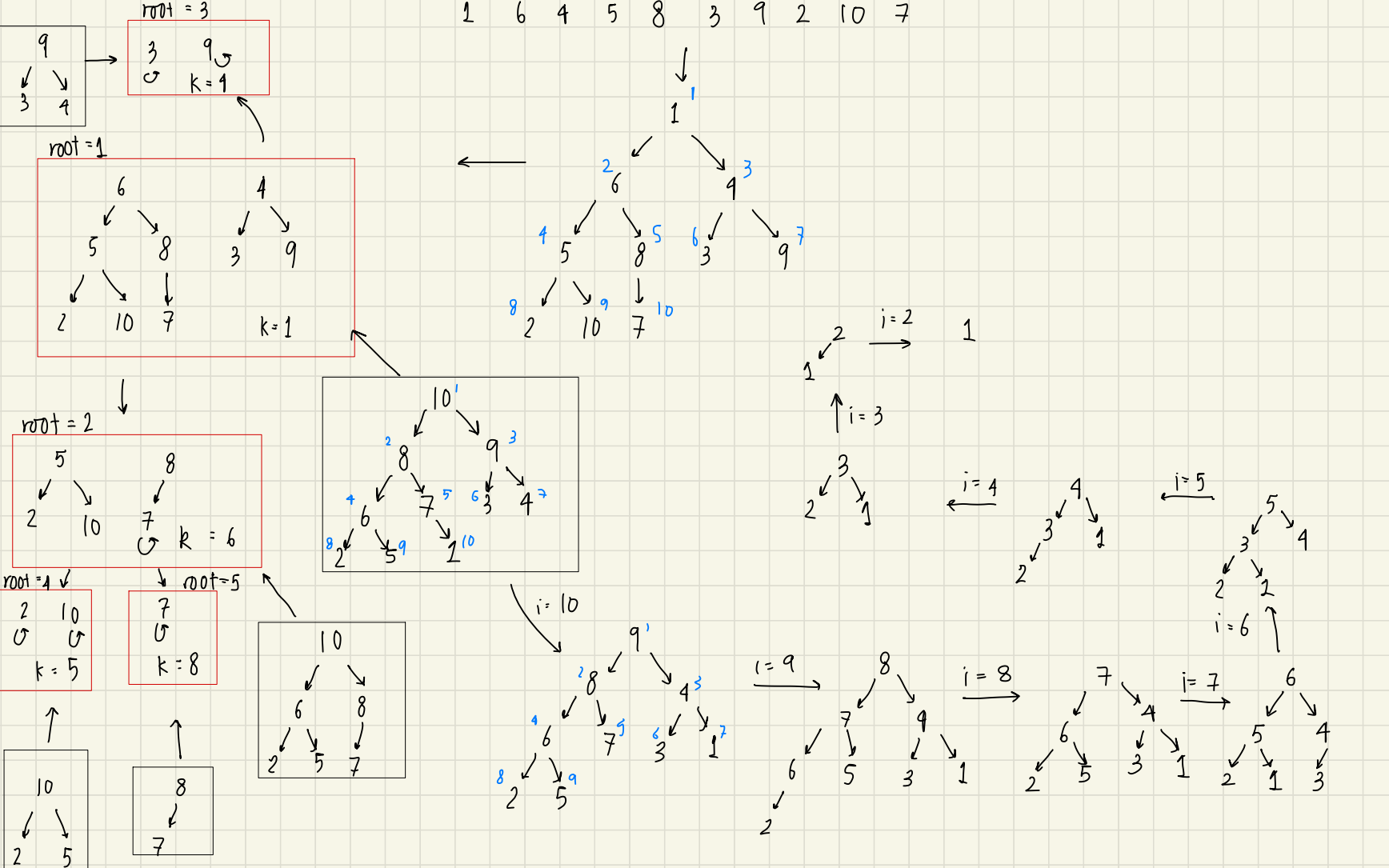
↳ also post order

```
deleteMax(H)
{
    copy the rightmost element at the lowest level to k
    delete rightmost element at the lowest level
    fixHeap(H, k);

}

fixHeap(H, k){

//iterative
    j = 1
    while (2j ≤ n)
    find bigger child of j + compare with k
     if k is bigger, H[j] = k + break()
     else H[j] = bigger child +
          j = index of bigger child


// recursive
    if (H is leaf)
       insert k at root
    else
       largerSubHeap = the larger child of H
       if (k ≥ key of root (largerSubHeap)
         insert k at root of H
       else
         fixHeap (largerSubHeap, k);
}
```

place a check for if 2j<n then check if A[2j+1] > A[2j]

```
Heapsort ( array , n ) {
    construct heap
    for (i = n ; i >= 1 ; i--) {
        curMax = getMax(H) ;
        deleteMax(H) ;
        array[i] = curMax ;
```


1 4 3 5 2
construct
1
2 4
3 3
4 5
5 2
heapifying
root = 1
4
5 2
3
(root = 3)
K = 1
root = 2
5
(root = 4)
2
(root = 5)
K = 4
fixheap
fixheap
i = 1
5
4 2
5
4 3
2
1 2
i = 2 for whole
i = 5
4
2 3
1
i = 4
3
2 1
i = 3
2
1
i = 2
1
i = 1
20800D
1 4
2 3
3 2
4 1
5 5

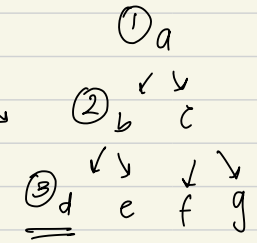

1 6 4 5 8 3 9 2 10 7
root = 3
k = 1
root = 1
k = 1
root = 2
k = 6
root = 4
k = 5
root = 5
k = 8
i = 10
i = 9
i = 8
i = 7
i = 6
i = 5
i = 4
i = 3
i = 2

Time Complexity

① FixHeap (H, K) recursive

↳ each call has 2 comp → bigger child
K > bigger child

how many calls?
best case → 1 (K has to be placed at the root)
worst case →
each call moves down a level



if leaf, no comp, just add so ignore ③

∴ worst case → tree height
(tree level - 1)

heap is "nearly complete" tree

complete tree has k levels, $2^k - 1$ nodes
∴ heap w/ k levels has n nodes

$$2^{k-1} - 1 < n \leq 2^k - 1$$
$$2^{k-1} \leq n \leq 2^k - 1$$
$$k - 1 \leq \lg n < k$$
$$k - 1 = \lfloor \lg n \rfloor$$
$$\therefore \text{height} = k - 1 = \lfloor \lg n \rfloor$$

height of heap w/ n nodes = $O(\lg n)$

and no of key comps = $2\lfloor \lg n \rfloor$
and time taken

time comp = $O(\lg n)$ of fixHeap worst case

O is slower or same

② How many times is fixheap called.

a) heapifying

↓

is called once

let $T(n)$ for heapifying on n nodes

$T(n) = \underset{4}{2}T((n-1)/2) + \underset{+(2+4)}{2}\lg n$ ← recursive, ← fixheap.

heapifying is called for all nodes

n (1)

↓

← $(n-1)/2$ (×2)

↓

$(n-3)/4$ (×4)

↓

$(n-7)/8$ (×8)

↓

$(n-m+1)/m$ (×m)

$T(1) = 0$

$\left(\frac{n-m+1}{m}\right) = 1$

$\therefore n+1 = 2m$

$m = \frac{n+1}{2}$

$n/2 - 1/2$

$n/4 - \frac{1+1}{3}$

$\frac{\frac{n}{4} - \frac{2}{4} - \frac{4}{4}}{2}$

$\frac{n}{8} - \frac{7}{8}$

$T(n) = \left(\frac{n+1}{2}\right)(T(1)) + \left(2^1 + 2^2 + \dots + 2^{\frac{n+1}{2}}\right)\lg n$

$= 2\left(2^{n-1/2} - 1\right)\lg n$

$= \left(2^{n+1/2} - 2\right)\lg n$

$= 2^n \lg n - 2\lg n$ ↖ nope!

$T(n) = O(n)$

b) deleteMax → calls fixHeap once

↓ is called n-1 times

} → 2(n-1) lg n

O(n lg n)

worst case → HeapSort (O(n lg n))

↓ construct [O(n)]

{

heapifying [O(n)]

↓ deleteMax [O(n lg n)]

↓ n times

fixHeap [O(lg n)]

→

$f_1 = O(g_1)$

$f_2 = O(g_2)$

$f_1 + f_2 = O(\max[g_1, g_2])$

Priority Queues

- a datastructure for maintaining data by keys
- supports insertion, reordering, deletion
  - insertion → insert
  - reordering → fixUp, fixDown
  - deletion → extractMax → Maximum

```
ALIST pq;   ← list
int N       ← size of pq
```

```
void insert(item i)
   pq[++N] = i;
   fixUp(pq, N);
```

} ∵ i is added at the end, we need to move it up worst case : O(lg n)

```
item extractMax()
   swap(pq[1], pq[N]);
   fixDown(pq, 1, N-1);
   return pq[N--];
```

} ∵ swap, smallest on top needs to be moved down. again O(lg n)

getMax : O(1)

∴ HEAP IS IDEAL TO SUPPORT PRIORITY QUEUE AT ANY POINT

Comparison Of Sorting Algorithms

	Best	Average	Worst
Insertion	n	n^2	n^2
Merge	n log n	n log n	n log n
Quick	n log n	n log n	n^2
Heap	n log n	n log n	n log n

↑ consider in practice

(1) Insertion sort

best case : already sorted
worst case : reverse sorted

(2) MergeSort : divide + conquer → evenly

strength : consistent performance for diff input types

(3) QuickSort : divide + conquer → about pivot

fastest

best case : pivot divides into half (eliminate half per call)

worst case : pivot eliminates one element per call

cause finding worst case input is hard.

(4) HeapSort : data structure approach

n lgn throughout, also consistent

has overhead due to the constant multiplier in big Oh

+ maintaining heap

Empirical Analysis

Insertion	0.1	168	342	23,382
Merge	2.0	2.3	2.2	30
Quick	0.7	0.9	0.7	12
*Radix	1.6	1.6	1.6	18
Heap	3.4	3.5	3.6	49
	Up		Down	100,000 numbers

This slide shows empirical time analyses of the sorting algorithms.
That is, we measure the CPU times used by the algorithms in a real computer,
The time unit is millisecond.
The algorithms are supposed to sort the numbers in ascending order
In the first column of data, labeled by “UP”, the input was ten thousand numbers in ascending order.
In the column labeled with “DOWN”, the input data are ten thousand numbers in descending order.
The column between “UP” and “DOWN” columns contains CPU time for input numbers that are in random order.
For the last column, the input are one hundred thousand numbers in random order.

As we can see from the table:
Insertion sort is indeed very fast when the input numbers are already sorted, because that is the best case for Insertion sort. However, when the input numbers are reversely sorted, then the performance of Insertion sort is very bad. It takes much longer than all the other algorithms.We can see that the empirical observation agrees with our theory about the running time. When input numbers are in random order, as shown in the column between “UP” and “DOWN”, we can see that Insertion sort also has very poor performance. Likewise, for the larger input data with one hundred thousand numbers, Insertion sort is much slower than other algorithms.

But for Mergesort, we can see that the performance is quite consistent. It has almost equally good performance for all the three input cases.

The speed of Quicksort is really impressive. It is much faster than all the other algorithms in most cases, except for the best case of Insertion sort. What is nice about Quicksort is that, it is not easy to find the worst-case input for Quicksort.

For Heapsort, it is consistently good, when compared with Insertion sort. But it has quite some overhead, which is measured by the constant multiplier in the big Oh notation. One reason is that we need to maintain the data structure of maximising heap, which will incur some overhead cost. But like Mergesort, Heapsort is quite reliable, meaning that it performs consistently well for different kinds of input data. On a whole, we can see that the empirical analyses of CPU times of the sorting algorithms agree with our theoretical analysis of the time complexity. That means our mathematical analysis of the sorting algorithms has good predictive power about their real performances.