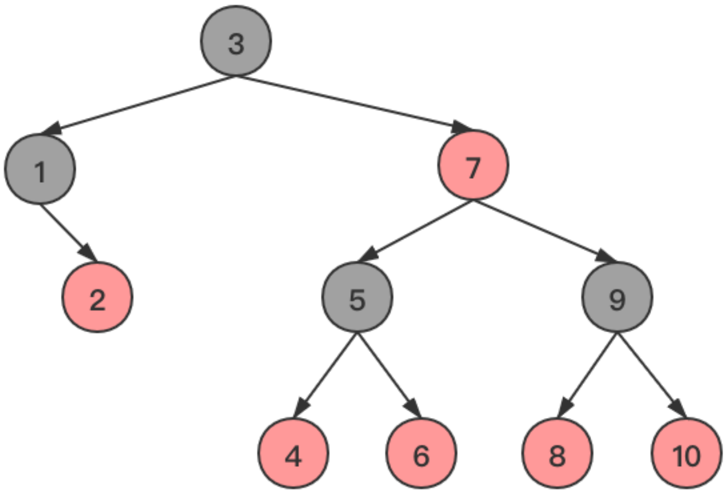
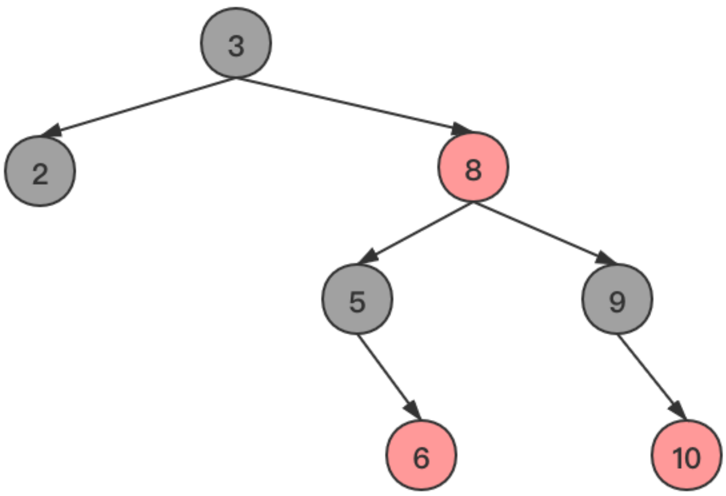
CSIT5500 Advanced Algorithm HW1

Lin Jialiang 20656855

1. (a) Inserting 1,3,5,7,9,2,4,6,8,10:



(b) Deleting 7,4,1:



2. The divide and conquer algorithm is:

Assume all the elements are stored in list L. The following algorithm is indeed

Merge sort algorithm with counting:

```
function find_inversions(L,low,high):
if low >= high, then return 0
mid = (low+high)/2
c1 = find_inversions(L,low,mid)
c2 = find_inversions(L,mid,high)
c3 = merge_part(L,low,mid,high)
return c1+c2+c3
```

The crucial part is merge_part():

```
function merge_part(L,low,mid,high):
create a new List nl
p = low, q = mid+1, cnt = 0
while (p<=mid) and (q<=high):
  if L[p]<=L[q]: nl.append(L[p]); p++
  else: nl.append(L[q]); q++; cnt += (mid-p+1) //crucial part of counting

if p>mid: copy the right leftover to nl
if q>high: copy the left leftover to nl
copy nl to L
return cnt
```

The correctness of the algorithm:

1) Firstly, by analyzing the base case the number of elements is 0 or 1, the algorithm correctly gets 0 as return.

2) Assuming that the algorithm works correctly when the number is k (k>1, k=2,…,n), then we consider the case that the number of elements is n+1: Since the algorithm works correctly when the number is k, thus the first two counting of sub-list are correct, so we correctly get c1 and c2; And the merge procedure correctly count the inversions in different two sub-lists and sort the entire list as well, so we get c3 as well.

   Total inversions appear in these 3 cases:

   i. i<j, L[i]>L[j], $i, j \in [low, mid]$

   ii. i<j, L[i]>L[j], $i, j \in [\mathrm{mid}, \mathrm{high}]$

iii. i<j, L[i]>L[j], $i \in [low, mid], \mathrm{j} \in [\mathrm{mid}, \mathrm{high}]$

c1 covers the first case, c2 covers the second case, and c3 covers the third case. So c1+c2+c3 cover all the case correctly, thus the algorithm is correct and it would work.

Runtime analysis:

The merge part takes O(n) time since the pointer only moves from the left to the right of entire list.

T(n) denotes the worst-case running time of the algorithm.

$$T(n) \leq O(1) + 2T\left(\frac{\mathrm{n}}{2}\right) + O(n)$$
$$T(1) = O(1)$$

Thus,

$$\begin{aligned} T(n) &\leq O(1) + 2T\left(\frac{\mathrm{n}}{2}\right) + O(n) \\ &\leq 4\mathrm{T}(\frac{\mathrm{n}}{4}) + 2\mathrm{O}(\mathrm{n}) \\ &\leq 2^k T\left(\frac{\mathrm{n}}{2^k}\right) + k \cdot O(n) \end{aligned}$$

When $k = log_2 n$,

$$\begin{aligned} T(n) &\leq nO(1) + O(nlogn) \\ &= O(nlogn) \end{aligned}$$

Thus, the algorithms works in O(nlogn) time.

3. According to the definition of a complete binary tree, we can get:

$$1 + 2^1 + \cdots + 2^{(h-2)} + f(h) = n$$

Where n denotes total number of nodes, h represents the height of tree and f(h) represents the number of nodes in the last level.

$$2^{(h-2)} < f(h) \leq 2^{(h-1)}$$

Therefore, we firstly assume $f(h) = 2^{(h-1)}$, and we get:

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

Assuming $f(h) = 2^{(h-2)}$, and we get:

$$1 + 2^1 + \cdots + 2^{(h-2)} = n$$
$$2^{h\text{-}1} - 1 = n$$
$$h = log_2(n+1) + 1$$

Thus, the relation between h and n is:

$$log_2(n+1) \le \mathrm{h} \le log_2(n+1) + 1$$

So, $\mathrm{h} = \mathrm{O}(\log \mathrm{n})$