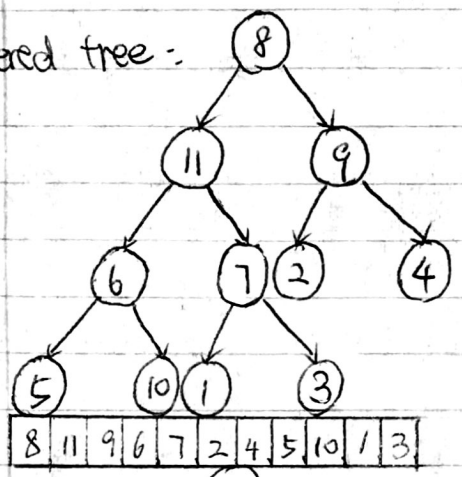
Q1 = ordered array :

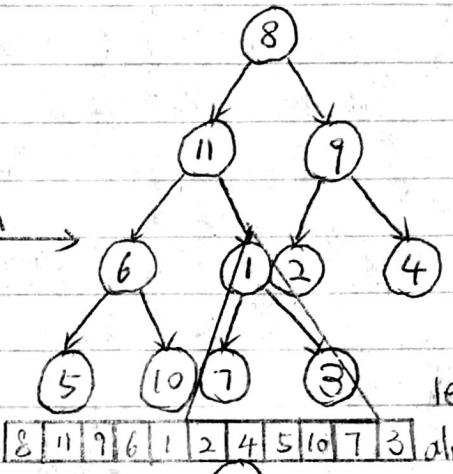
1	3	2	5	7	9	4	6	10	8	11

unordered tree :



8	11	9	6	7	2	4	5	10	1	3

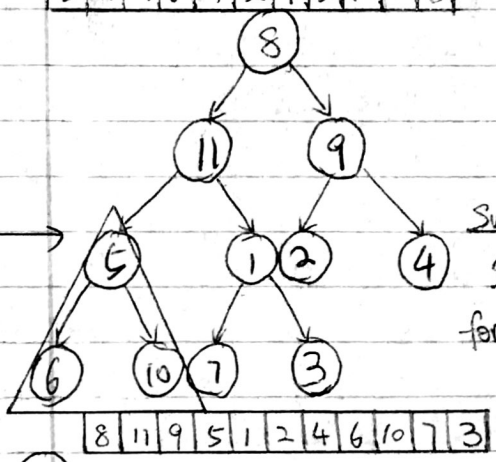
SwapDown →



8	11	9	6	1	2	4	5	10	7	3

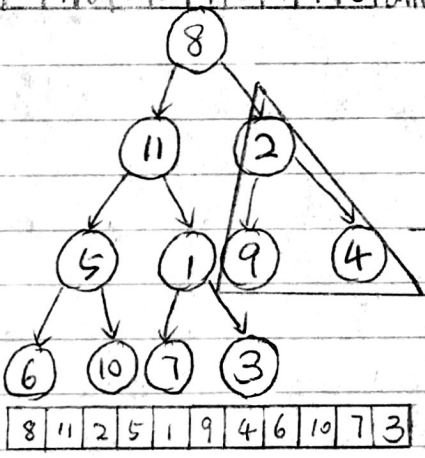
* SwapDown is also recursively called on (7) and (8), but they are leaves, so they are already in heap order

SwapDown →



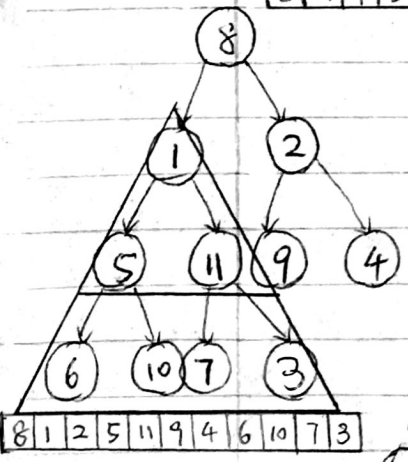
8	11	9	5	1	2	4	6	10	7	3

SwapDown → Similar to * for (6) and (10)



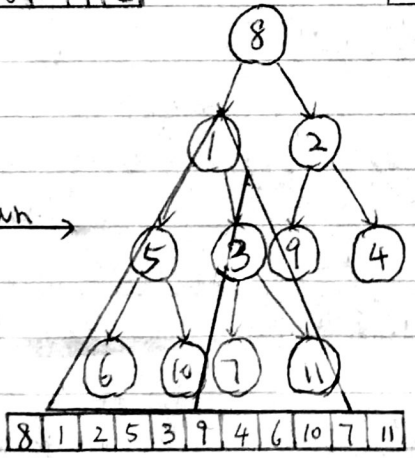
8	11	2	5	1	9	4	6	10	7	3

SwapDown → similar to * for (9) and (4)



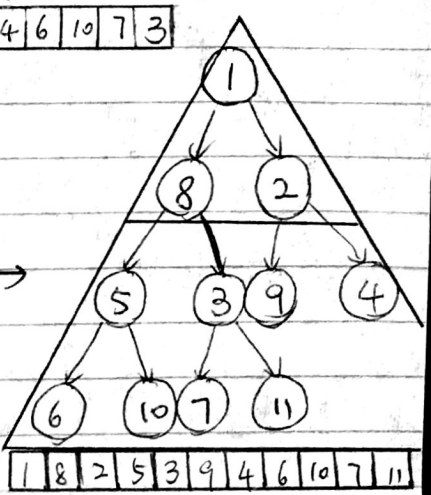
8	1	2	5	11	9	4	6	10	7	3

SwapDown →



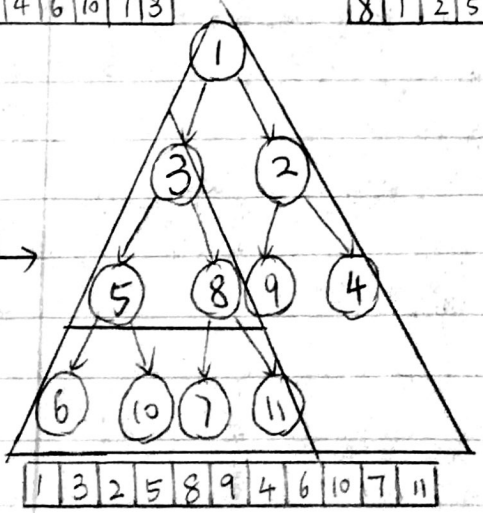
8	1	2	5	3	9	4	6	10	7	11

SwapDown →



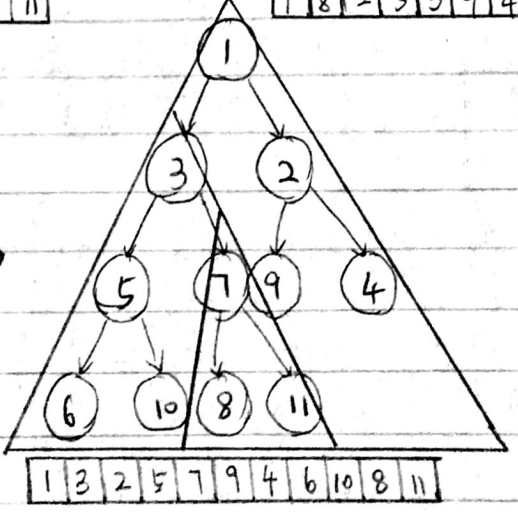
1	8	2	5	3	9	4	6	10	7	11

SwapDown →



1	3	2	5	8	9	4	6	10	7	11

SwapDown →



1	3	2	5	7	9	4	6	10	8	11

Q2:

```
#include <iostream>
#include <string>
using namespace std;

int countParensC(string S, int i=0, int numOpen=0){
    if(s.length()==0){return 0;}      # empty string
    if(S[i]=='('){                    # found opening bracked.
        numOpen++;                    # incr numOpen
        if(i==s.length()-1){return 0;}# end of string reached? check!
        i++;                          # incr i
        return countParensC(s,i,numOpen); # check rest of characters
    }                                   recursively
    if(S[i]==')'){                    # found closed bracked
        if(numOpen>0){                # preceding '(' exists.
            numOpen--;                # decr numOpen count
            if(i==s.length()-1){return 0;} # end of str check
            i++;
            return countParensC(s,i,numOpen);
        }
        if(i==S.length()-1){
            return 1;
        }else{                        # if numOpen=0, unmatched ')' found
            i++;
            return 1+countParensC(s,i,numOpen);
        }                             # incr unmatched ')' parentheses
    }                                   count
    else{
        if(i==s.length()-1){return 0;}  # if S[i] is neither '(' or ')', do
        i++;                            not incr closed parentheses #.
        return countParensC(s,i,numOpen);
    }
}
```

Q3.

(a)
```
int bitsort (int *A, int n){
    int i = 0;
    int j = n-1;
    while (i<j) {
        if (A[i] == 0){
            i++;
        }else if (A[j] == 1){
            j--;
        }else {
            swap (A[i], A[j]);
        }
    }
    return j;
}
```

(b) Loop invariant:

Before each iteration, A[0 ··· (i-1)] (head) contains "0" only (0s are at the front end of the array), and A[(j+1) ··· (n-1)] (tail) contains "1" only (1s are at the back end of the array); or, A[head] and A[tail] contain same elements (all "0s" or all "1s" in the array).

Base Case: before the iteration starts, when i=0, j=n-1, we have 4 cases for A[i ··· j]:

① A[0 ··· 0] } it is obviously true that A[head] and A[tail] have same element { "0"
② A[1 ··· 1] } { "1"

③ A[0 ··· 1] - it is obviously true that A[head] has "0" only and A[tail] has "1" only.

④ A[1 ··· 0] - after the swap call, A[i] and A[j] are swapped, so then this case turns to be A[0 ··· 1], which is same as case ③

Inductive Hypothesis: Assume that the invariant holds after k>0 iteration

with increasing number (k) of iterations, i is moving towards j and j moving towards i, so the middle sub-array (A[0 ··· (i-1) ··· (j+1) ··· n] middle sub-array) is narrowing down. At the beginning of each iteration, we can treat each sub-array (in the middle) as a new array, which can have 4 cases as stated in base case until $i \geq j$. Since the basic cases hold the invariant, the invariant is held for each sub-array at the beginning of each iteration as well. So the invariant holds after $k \geq 0$ iterations by inductive hypothesis.

(c) The code will terminate the loop when the increasing i reaches the decreasing j, that is, $i == j$. That means # of "head" indices plus # of "tail" indices equals to n, which is the size of the original array. Also, the middle sub-array is narrowed down to nothing. Thus, both A[0 ··· (i-1)] and A[(j+1) ··· n] are sorted ($i == j$), which means the entire array A is sorted. The invariant holds.

d) it will return 0 when the given array contains 1s only.

Q4.

(a) Since n is not a multiple of 3, we assume that for all $k \in \mathbb{N}$ $n = 3k+1$ or $n = 3k+2$.

when $k = 0$,

if $n = 1$, then Alice can win by taking 1 link from the necklace

if $n = 2$, then Alice can win by taking 2 links from the necklace

when $k > 0$,

if $n = 3k+1$, Alice can take 1 link first and leave Bob the necklace with 3k links. Then Bob can take 1 or 2 links from the necklace in his turn, and Alice can win by taking 2 or 1 links respectively.

if $n = 3k+2$, Alice can take 2 links first and leave Bob the necklace with 3k links again. Then, Bob can take 1 or 2 links

from the necklace, and Alice can win by taking 2 or 1 links respectively.

In order to win, Alice should always leave Bob the necklace with 3k links.

(b) ① Assume $n \geq 3$ where Alice goes first. Let g = the number of segmentations between necklace links (i.e $g=0$, $g=1$) after Alice's first turn, $g=1$ always.

② Assume Alice has used her first turn. She can choose to take 1 or 2 (adjacent) beads, which are $(n-1)$ or $(n-2)$ respectively. As a result of her first turn, the first segmentation will always ($g=1$) be the result. Also, $n \geq (3-2) \rightarrow n \geq 1$

③ Assume Bob can make the following moves after Alice's turn:

$\rightarrow n-2, g+1$	$\rightarrow n-1, g+1$
$\rightarrow n-2, g-1$	$\rightarrow n-1, g-1$
$\rightarrow n-2, g$	$\rightarrow n-1, g$

④ Assume that $n=0, g=0$ is the wrong condition

Case #1: $n=1, g=1$

Bob takes the link and wins. (Bob chooses $n-1$ $g-1 \rightarrow n=0$ $g=0$)

Case #2: $n=2, g=1$

Bob chooses $(n-2$ $g-1 \rightarrow n=0$ $g=0)$

Case #3: $n=1, g=0$

Bob choose $(n-1, g)$

Case #4: $n=2, g=0$

Bob choose $(n-2, g)$

As long as after Bob's turn, Alice is left with $n \bmod 2 = 0$ and $g \bmod 2 = 0$, Bob can use any of his 6 moves to maintain $n \bmod 2 = 0$ and $g \bmod 2 = 0$. After each turn, n becomes progressively smaller by 1 or 2. Therefore, n will eventually reach one of Case #1-4, and g will eventually reach 0!!

Invariant: After Alice's turn, Bob must make a move that leaves Alice with $n \bmod 2 = 0$ and $g \bmod 2 = 0$.

Q5:

(a) Given any number x, $x \in [0, 255]$, $x \in \mathbb{N}$, we have.

#0 $x + 0 \times (2^{16}-1) \Rightarrow (x + 0) \bmod (2^{16}-1) = x$

#1 $x + 1 \times (2^{16}-1) \Rightarrow (x + 255 + 255 \times 256) \bmod (2^{16}-1) = x$

#2 $x + 2 \times (2^{16}-1) \Rightarrow (x + 254 + 255 \times 256 + 1 \times 256^2) \bmod (2^{16}-1) = x$

#3 $x + 3 \times (2^{16}-1) \Rightarrow (x + 253 + 255 \times 256 + 2 \times 256^2) \bmod (2^{16}-1) = x$

#4 $x + 4 \times (2^{16}-1) \Rightarrow (x + 252 + 255 \times 256 + 3 \times 256^2) \bmod (2^{16}-1) = x$

⋮ ⋮ ⋮

#256 $x + 256 \times (2^{16}-1) \Rightarrow (x + 0 + 255 \times 256 + 255 \times 256^2) \bmod (2^{16}-1) = x$

When it reaches #257, $h(S_0 S_1 S_2)$ is not a three-character string any more. Since the size of the hash table is $(2^{16}-1)$, the # of three-character string that are hashed to the same slot (for each slot) are stated as above. So we have strings from #0 to #256, then there are at least 257 three-character strings are hashed to the same slot.

(b) In the given case, we have $h(ABCD) = h(CBAD) = h(CDAB)$

ABCD → CBAD → CDAB

and $h(ABCD) \neq h(BACD)$

ABCD → BACD.

let $i, j \in \mathbb{N}$ and $0 < i < j < k$. Then we can say that any two strings can be hashed to the same slot as long as we can get of the strings by switching S_i and S_j of the other ~~one~~ string. $i = (\text{odd \#} + j)$, which means S_i and S_j cannot be next to each other, and the # of characters between S_i and S_j has to be odd.