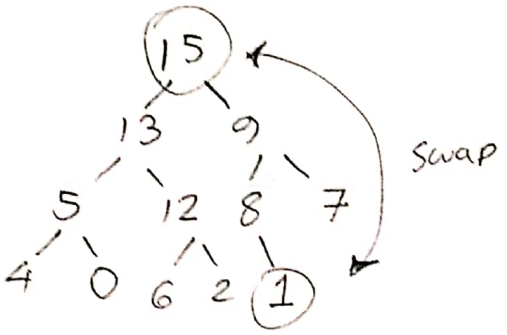
Name - Md Farhan Ishmam

ID - 1800411120

CSE - 4303

Ans. to Q.no 1

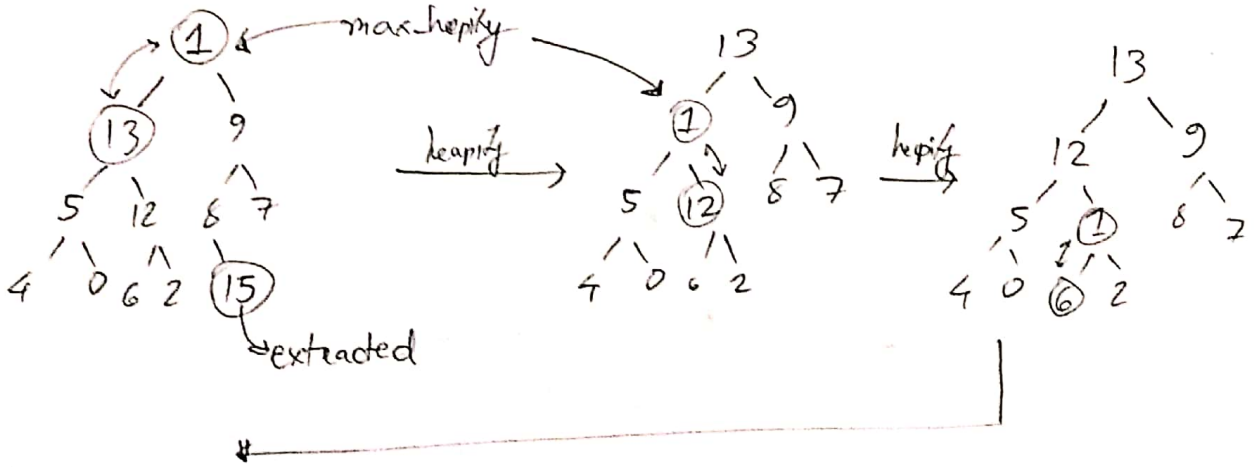


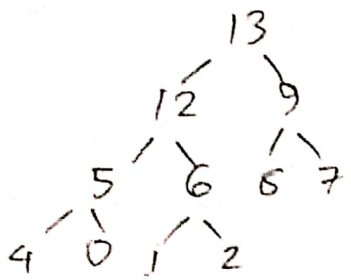
The operation heap extract will take the top-most element from the heap and remove it.

The algorithm is

- Replace with a leaf node (last one) i.e swap(A[1], A[heap-size]).
- Decrease heap-size and thus remove the last element
- max-heapify from the root

max-heapify (A[1])





Hence, 15 is extracted and heap property is maintained.

Max-heapify is recursive so all operations are done after one function call.

Ans to Q.no. 2

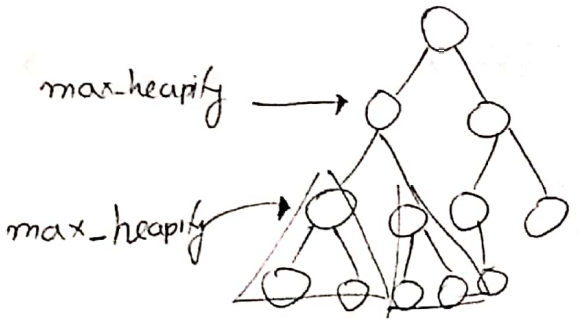
~~max-heapify ()~~

The function build_max_heap is written as

```
void build_max-heap (arr)
{
    for (i = heap_length/2 down to 1) // except all leaf nodes
    {
        max_heapify (A[i]);
    }
}
```

Now, the leaves are already following the heap property i.e. their children has a smaller value than their parents. So, we don't need to heapify the leaf nodes since they don't have children.

When max_heapify is called from the bottom, the leaves will form a heap with the parent. As we go up we form small heaps. When a parent doesn't follow



the max_heapify property then it will be further heapified. But if it DOES follow the property then no further step needs to be performed. Thus time is

saved and bottom-up can be called a more efficient approach in this case.

~~Max-heapify~~

Because in our max-heapify function we find the maximum value among parent and their two children. But if the parent is the largest the nothing is performed. In case of down we might need to do extra steps here.

Ans. to Q.no. 3

a) push (x)

To push in AVL tree we need $\log_2(n)$ time.

b) pop()

To pop we need find the minimum element and extract it. Finding minimum value is easy since it is the leftmost node of the tree. So, it will also take $\log_2(n)$ time. Since it is a leaf node it doesn't need to be replaced by a successor since it has none.

c) Finding minimum takes $\log_2(n)$ time.

Ans. to Q. no. 5

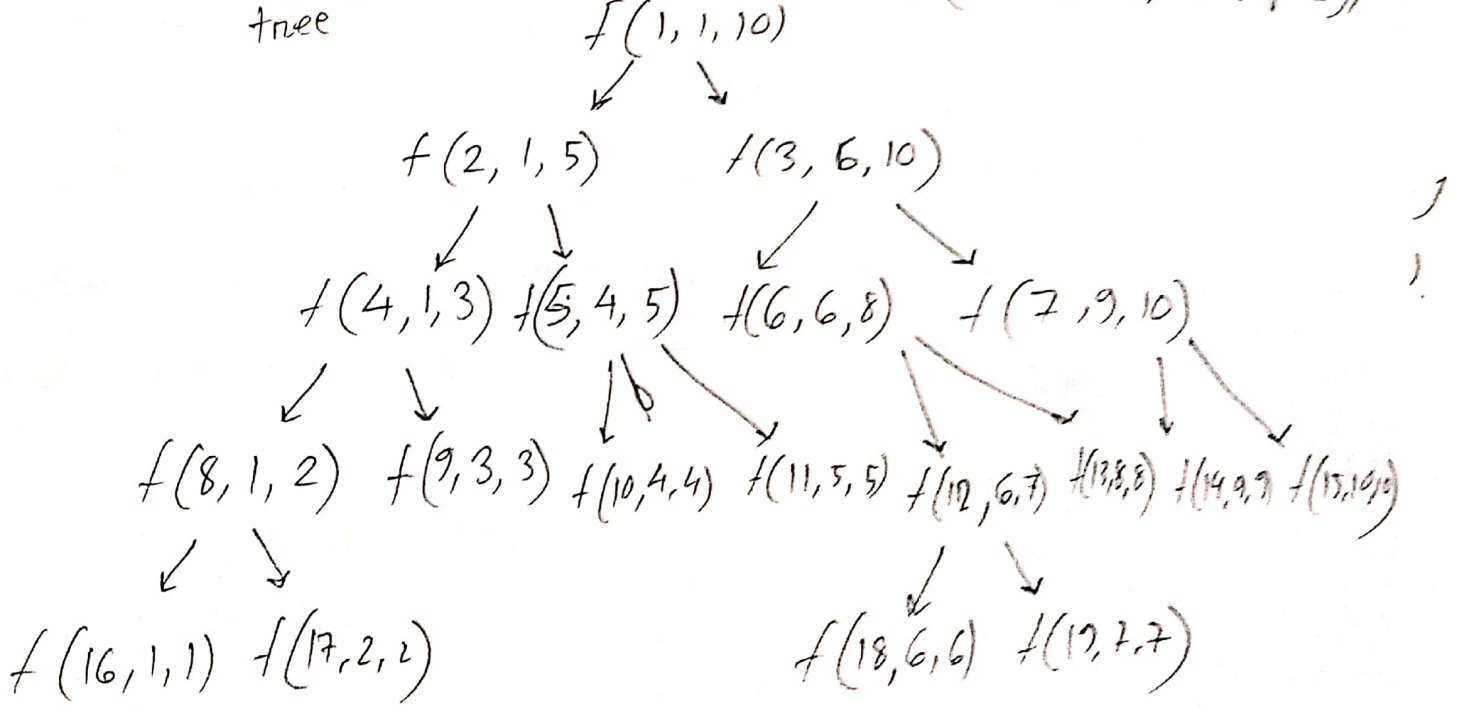
The segment tree can be built using the build segment tree function, which will produce a tree from an array.

Given, A = { 10, 5, 7, -2, 8, 14, 3, 0, 1, -12 }
(1 2 3 4 5 6 7 8 9 10)

For the steps of recursion we need,

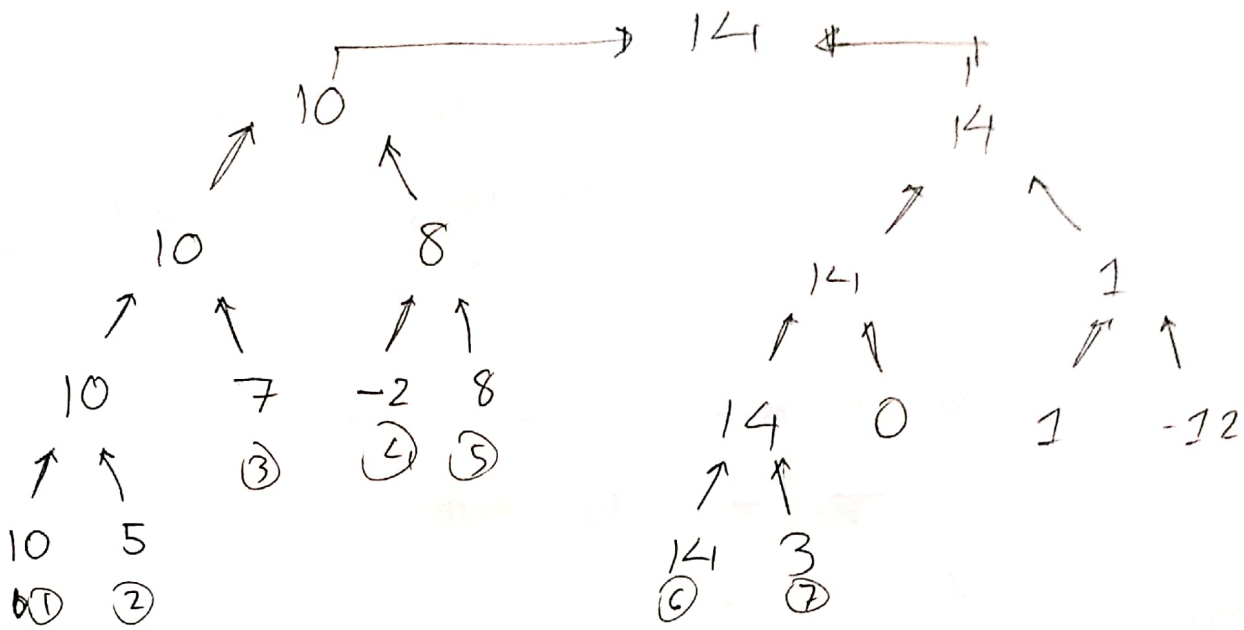
left = 2 * node
right = 2 * node + 1
mid = (begin + end) / 2

init (left, begin, mid);
init (right, mid+1, end);
tree[node] = max (tree[left], tree[right]);



if (begin == end) return;

So, the segment tree will be



Q: For, range (3..7)

If the array goes out of bound then we return minimum value.

query_max (left, begin, mid, i, j)
query_max (right, mid+1, end, i, j).

=1
=0

(6)

(a)

$S = \{4371, 1323, 6173, 4199, 4344, 9679, 1989\}$

$h(x) = x \bmod 10$

0 — null

1 — [4371] — null

2 — NULL

3 — [6173] — [1323] — null

4 — [4344] — null

5 — null

6 — null

7 — null

8 — null

9 — [1989] — [9679] — [4199] — null

(b) (7)

$f(i) = i \times hash_2(x)$; $hash_2(x) = 7 - (x \% 7)$

$\Rightarrow f(i) = i \times \{7 - (x \bmod 7)\}$

And, $h(x) = x \bmod 10 + f(i)$;

i) 4371 $\Rightarrow$ 4371 % 10 = 1

ii) ~~6173 $\Rightarrow$ 6173 % 10 $\Rightarrow$ 3~~ 1323 $\rightarrow$ 1323 % 10 = 3

iii) ~~6173 $\Rightarrow$ 6173 % 10 $\Rightarrow$ 3~~ 6173 6173 % 10 = 3 (collision)

$f(i)^0 = 7 - 6 = 1$ ~~(collision)~~

~~iv) 4344 $\Rightarrow$ 4344 % 10 = 4~~

~~v) 1989 $\Rightarrow$ 9~~

$\therefore h(x) = 3 + 1 = 4$

~~vi)~~

iv) 43441 $\rightarrow$ 4 (collision)

$f(i) = 3$

Similarly we do it for the rest

0 - 9679
1 - 4371
2 -
3 - 1323
4 - 6173
5 - 1989
6 -
7 - 4344
8 -
9 - 4199

9679 $\rightarrow$ (7 - 5) = 2

$f(i) = 9 + 2 \% 10 = 1$

$2 \times 9 + 2 \% 10 = 0$

Ans. to Q.no. 7

```
struct Node
{
    Node * next[26];
    bool end_flag;
    int letter_count;
};

Node* create_Node()
{
    Node* newnode = new Node();
    newnode -> end_flag = false;
    newnode -> letter_count = 0;
    for(i = 0 to 25)
        newnode -> next[i] = null;
    return newnode;
}

void spell_check(string s, node root)
{
    Node * current = root;
    for (i = 0 to strlen()-1)
    {
        key = str[i] - 'A'
```

```
        if (current->next[key] == null)
        {
            error "Misspelt word";
            return;
        }
        current = current->next[key];
    }
    if (current->end_flag == false)
    {
        error "Misspelt word";
        return;
    }
    return;
}
void insert_word (string s, node root)
{
    Node * current = root;
    for (i = 0 to length_of_string - 1)
    {
        key = str[i] - 'A';
        if (current->next[key] == null)
        {
            current->next[key] = createNode()
        }
        current = current->next[key];
        current->letter_count++;
```

```
        current->end_flag = true;
    }   return;
}
```

~~number~~

```
int number_of_prefix_words(string s), Node root)
{
    int no_of_prefix = INT_MAX;
    Node * current = root;
    for (i = 0 to strlen() - 1)
    {
        key = str[i] - 'A'
        if(current->next[key] != null)
            no_of_prefix = min(no_of_prefix,
                               current->next[key]->
                               letter count)
        else
            error("Prefix not found");
            return;
    }

    return no_of_prefix;
}
```

Ans. to Q.no. 4

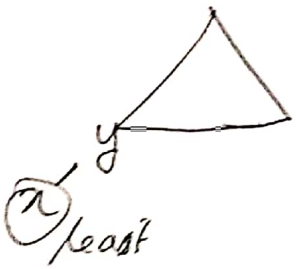
~~Hashcode = word [~~ x (leaf) y (parent)] T

y – is either smallest key in T langer than x key

or

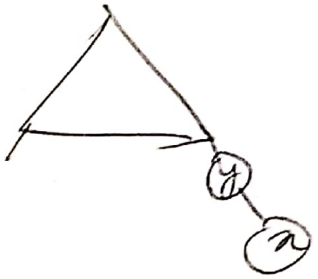
largest key in T smaller x key.

Now, in a BST, the leftmost element is the smallest. So,



Here, the parent y will be langer than x only if x is the left node. Other than that y is the smallest element except x.

Again, if the y is the rightmost key then the key y will be largest. If x is at right then other x-, the value of y will be largest.



~~But the tree can take a different form.~~

So, this proves the given statement. But for other forms, it will not be true.

Ans. to Q.no8

```
#define word_size 100;
hash (string s), ann[]
{   for(int i to strlen() -1)
        ann[word size - i -1] × pow(37,i)
    if (ann[wordsize -i-1] != null)
        ann[wordsize -i-1 + i]
}
```

linear probing.