



Explain what each of the functions in the code below does:

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <time.h>

//This class represents the root of a tree, the left and right child points
to it's sub trees a single node is represented as a binary tree of type
struct tree_root {
    int64_t val;
    int64_t type; //((2^(type+1)) - 1 amount of nodes, how many times it
has been merged
    struct tree_root *left;
    struct tree_root *right;
    struct tree_root *left_child;
    struct tree_root *right_child;
    int64_t left_most_value;
    int64_t right_most_value;
    struct tree_root *parent;
}typedef tree_root;

struct list{
    tree_root *head;
    tree_root *tail;
    int orientation;
    int size;
}typedef list;

tree_root *create_root(int64_t v){
    tree_root *new_root = (tree_root*)malloc(sizeof(tree_root));
    new_root->val = v;
    new_root->left = NULL;
    new_root->right = NULL;
    new_root->left_child = NULL;
    new_root->right_child = NULL;
    new_root->type = 0;
    new_root->left_most_value = v;
    new_root->right_most_value = v;
    return new_root;
}

void combine_trees_right (list *l){
    tree_root *current_tree = l->tail;
    tree_root *new_root = create_root(current_tree->left->val);
    new_root->left_child = current_tree->left;
    new_root->right_child = current_tree;
    new_root->left_most_value = new_root->left_child-
>left_most_value;
    new_root->right_most_value = new_root->right_child-
>right_most_value;
    new_root->type = new_root->left_child->type + 1;

    if (current_tree->left->left == NULL){
        l->head = new_root;
        l->tail = new_root;
        return;
    }

    current_tree->left->left->right = new_root;
    new_root->left = current_tree->left->left;
    l->tail = new_root;
}

void combine_trees_left (list *l){
    tree_root *current_tree = l->head;
    tree_root *new_root = create_root(current_tree->val);
    new_root->left_child = current_tree;
    new_root->right_child = current_tree->right;
    new_root->left_most_value = new_root->left_child->
```

```

new_root->left_most_value = new_root->left_child->left_most_value;
new_root->right_most_value = new_root->right_child->right_most_value;
new_root->type = new_root->left_child->type + 1;
if (current_tree->right->right == NULL){
    l->head = new_root;
    l->tail = new_root;
    return;
}

current_tree->right->right->left = new_root;
new_root->right = current_tree->right->right;
l->head = new_root;

}

void absolute_push_left(list *l, int64_t v){
tree_root *new_root = create_root(v);
if (l->size == 0){
    l->head = new_root;
    l->tail = new_root;
    l->size++;
    return;
}
new_root->right = l->head;
l->head->left = new_root;
l->head = new_root;

while (l->head->type == l->head->right->type){
    combine_trees_left(l);
    if (l->head->right == NULL){
        break;
    }
}
l->size++;
}

void absolute_push_right(list *l, int64_t v){
tree_root *new_root = create_root(v);
if (l->size == 0){
    l->head = new_root;
    l->tail = new_root;
    l->size++;
    return;
}
new_root->left = l->tail;
l->tail->right = new_root;
l->tail = new_root;

while (l->tail->type == l->tail->left->type){
    combine_trees_right(l);
    if (l->tail->left == NULL){
        break;
    }
}
l->size++;
}

void push_left(list *l, int64_t v){
if (l->orientation == 0){
    absolute_push_left(l, v);
}
else{
    absolute_push_right(l, v);
}
}

void push_right(list *l, int64_t v){
if (l->orientation == 0){
    absolute_push_right(l, v);
}
else{
    absolute_push_left(l, v);
}
}

```

```
}
```

```
list *make(int n, int64_t *seq){//Temporary Non O(l) make [O(l*logn)]
```

```
list * l= (list*)malloc(sizeof(list));  
l->size = 0;  
l->orientation = 0;  
if (n==0){  
    l->head = NULL;  
    l->tail = NULL;  
    return l;  
}  
for (int i = 0; i<n; i++){  
    push_right(l, seq[i]);  
}  
return l;
```

```
}
```

```
void decompose_left(list *l){  
if(l->head->right == NULL){  
    tree_root *temp = l->head;  
    l->head = l->head->left_child;  
    l->tail = l->head->right_child;  
    free(temp);  
}  
else{  
    tree_root *temp = l->head;  
    l->head->right->left = l->head->right_child;  
    l->head->right_child->right = l->head->right;  
    l->head = l->head->left_child;  
    free(temp);  
}  
}
```

```
void decompose_right(list *l){  
if(l->tail->left == NULL){  
    tree_root *temp = l->tail;  
    l->head = l->tail->left_child;  
    l->tail = l->tail->right_child;  
    free(temp);  
}  
else{  
    tree_root *temp = l->tail;  
    l->tail->left->right = l->tail->left_child;  
    l->tail->left_child->left = l->tail->left;  
    l->tail = l->tail->right_child;  
    free(temp);  
}  
}
```

```
bool absolute_pop_left(list *l){
```

```
if (l->size == 0){
```

```
printf("Size is 0 cannot pop\n");
```

```
return false;
```

```
}
```

```
if (l->size == 1) ↓
```

```
tree_root *temp = l->head;
```

```
l->head = NULL;
```

```
l->tail = NULL;
```

```
l->size--;
```

```
free(temp);
```

```
return true;
```

```
}
```

```
if (l->head->type == 0){
```

```
tree_root *temp = l->head;
```

```
l->head = temp->right;
```

```
l->head->left = NULL;
```

```
l->size--;
```

```
free(temp);
```

```
return true;
```

```
}
```

```
while(l->head-> ↓ e!=0){
```

```
decompose_left(l);
```

```
}
```

```
tree_root *temp = l->head;
```

```
l->head = temp->right;
```

```

    l->head = temp->right;
    l->head->left = NULL;
    l->size--;
    free(temp);
    return true;
}
bool absolute_pop_right(list *l){
    if (l->size == 0){
        printf("Size is 0 cannot pop\n");
        return false;
    }
    if (l->size == 1){ ↓
        tree_root *temp = l->tail;
        l->head = NULL;
        l->tail = NULL;
        l->size--;
        free(temp);
        return true;
    }
    if (l->tail->type == 0){
        tree_root *temp = l->tail;
        l->tail = temp->left;
        l->tail->right = NULL;
        l->size--;
        free(temp);
        return true;
    } ↓
    while(l->tail->type!=0){
        decompose_right(l);
    }
    tree_root *temp = l->tail;
    l->tail = temp->left;
    l->tail->right = NULL;
    l->size--;
    free(temp);
    return true;
}

bool pop_left(list *l){
    if (l->orientation == 0){
        return absolute_pop_left(l);
    }
    else{ ↓
        return absolute_pop_right(l);
    }
}

bool pop_right(list *l){
    if (l->orientation == 0){
        return absolute_pop_right(l);
    }
    else{
        return absolute_pop_left(l);
    }
}

int64_t absolute_peek_left(list *l){
    if (l->size == 0){ ↓
        printf("Size 0 .\nNOT PEEK\n");
        return -1;
    }
    else{
        return l->head->left_most_value;
    }
}
int64_t absolute_peek_right(list *l){
    if (l->size == 0){
        printf("Size 0 CANNOT PEEK\n");
        return -1;
    }
    else{
        return l->tail->right_most_value;
    }
}
int64_t peek_left(list *l){ ↓
}

```

```

int64_t peek_left(list *l){
    if (l->orientation == 0){
        return absolute_peek_left(l);
    }
    else{
        return absolute_peek_right(l);
    }
}
int64_t peek_right(list *l){
    if (l->orientation == 0){
        return absolute_peek_right(l);
    }
    else{
        return absolute_peek_left(l);
    }
}
int size(list *l){
    return l->size;
}
bool empty(list *l){
    if (size(l) == 0){
        return true;
    }
    else{
        return false;
    }
}
int64_t normal_get(list *l, int i){
    if (i<0 || i>((l->size) - 1)){
        printf("Index Out of Range\n");
        return -1;
    }
    int distance = i;
    tree_root *temp = l->head;
    while(distance>=(1LL<<(temp->type))){
        distance -= 1LL<<(temp->type);
        temp = temp->right;
    }
    while(temp->left_child!= NULL){
        if (distance>=(1LL<<(temp->left_child->type))){
            distance -= 1LL<<(temp->left_child->type);
            temp = temp->right_child;
        }
        else{
            temp = temp->left_child;
        }
    }
    return temp->val;
}

int64_t reverse_get(list *l, int i){
    if (i<0 || i>((l->size) - 1)){
        printf("Index Out of Range\n");
        return -1;
    }
    int distance = i;
    tree_root *temp = l->tail;
    while(distance>=(1LL<<(temp->type))){
        distance -= 1LL<<(temp->type);
        temp = temp->left;
    }
    while(temp->right_child!= NULL){
        if (distance>= ↓ L<<(temp->right_child->type))){
            distance -= 1LL<<(temp->right_child->type);
            temp = temp->left_child;
        }
        else{
            temp = temp->right_child;
        }
    }
    return temp->val;
}

int64_t get(list *l, int i){
    if (l->orientation == 0){
        return normal_get(l,i);
    }
}

```

```

        }
        else{
            return reverse_set(l,i);
        }
    }
void normal_set(list *l, int i, int64_t v){
    if (i<0 || i>((l->size) - 1)){
        printf("Index Out of Range\n");
        return;
    }
    int distance = i;
    tree_root *temp = l->head;
    while(distance>=(1LL<<(temp->type))){
        distance -= 1LL<<(temp->type);
        temp = temp->right;
    }
    if (temp->type == 0){
        temp->left_most_value = v;
        temp->right_most_value = v;
        temp->val = v;
        return;
    }
    int arr_size = temp->type;
    int *back_track_arr = (int*)malloc(arr_size*sizeof(int));
    /*Array takes note of whether we went left_child or right_child in
    the tree, 0 is left 1 is right
    We use this to update the left_most_val and right_most_val for the
    peek() functions
    After reaching the end and changing the value, we update the
    right_most_val and left_most_val of the higher trees*/
    int curr_index_bta = 0; //Keeps track of index we update in the array
    above

    while(temp->left_child!= NULL){
        if (distance>=(1LL<<(temp->left_child->type))){
            distance -= 1LL<<(temp->left_child->type);
            temp->right_child->parent = temp;
            temp = temp->right_child;
            back_track_arr[curr_index_bta] = 1;
            curr_index_bta++;
        }
        else{
            temp->left_child->parent = temp;
            temp = temp->left_child;
            back_track_arr[curr_index_bta] = 0;
            curr_index_bta++;
        }
    }

    temp->left_most_value = v;
    temp->right_most_value = v;
    temp->val = v;

    for (int i = arr_size - 1; i>=0; i--){
        temp = temp->parent;

        if (back_track_arr[i] == 0){
            temp->left_most_value = temp->left_child->left_most_value;
            temp->val = temp->left_most_value;
        }
        else{
            temp->right_most_value = temp->right_child-
>right_most_value;
        }
    }
    free(back_track_arr);
}

void reverse_set(list *l, int i, int64_t v){
    if (i<0 || i>((l->size) - 1)){
        printf("Index Out of Range\n");
        return;
    }
    int distance = i;
    tree_root *temp = l->tail;

```

```

while(distance>=(1LL<<(temp->type))){  

    distance -= 1LL<<(temp->type);  

    temp = temp->left;  

}  

if (temp->type == 0){  

    temp->left_n ↓ .value = v;  

    temp->right_most_value = v;  

    temp->val = v;  

    return;  

}  

int arr_size = temp->type;  

int *back_track_arr = (int*)malloc(arr_size*sizeof(int));  

/*Array takes note of whether we went left_child or right_child in  

the tree, 0 is left 1 is right  

We use this to update the left_most_val and right_most_val for the  

peek() functions  

After reaching the end and changing the value, we update the  

right_most_val and left_most_val of the higher trees*/  

int curr_index_bta = 0; //Keeps track of index we update in the array  

above

while(temp->rig. ↓ _child!= NULL){  

    if (distance>=(1LL<<(temp->right_child->type))){  

        distance -= 1LL<<(temp->right_child->type);  

        temp->left_child->parent = temp;  

        temp = temp->left_child;  

        back_track_arr[curr_index_bta] = 0;  

        curr_index_bta++;  

    }  

    else{  

        temp->right_child->parent = temp;  

        temp = temp->right_child;  

        back_track_arr[curr_index_bta] = 1;  

        curr_index_bta++;  

    }  

}  

temp->left_most_value = v;  

temp->right_most_value = v;  

temp->val = v;

for (int i = arr_size - 1; i>=0; i--){  

    temp = temp->parent;  

    if (back_track_arr[i] == 0){  

        temp->left_most_value = temp->left_child->left_most_value;  

        temp->val = temp->left_most_value;  

    }  

    else{  

        temp->right_most_value = temp->right_child->right_most_value;  

    }  

}
free(back_track_arr);
}

void set(list *l, int i, int64_t v){  

    if (l->orientation == 0){  

        normal_set(l,i,v);  

    }  

    else{  

        reverse_set(l,i,v);  

    }  

}
void reverse(list *l){  

    if (l->orientation == 0){  

        l->orientation = 1;  

    }  

    else{  

        l->orientation = 0;  

    }  

}

```



Function Descriptions:

1. `create_root(int64_t v)`

- Creates a new tree node (`tree_root`) with the value `v`.
- Initializes its child pointers (`left`, `right`, `left_child`, `right_child`) to `NULL`.
- Sets the `type` to 0 (indicating a single node).
- Sets the `left_most_value` and `right_most_value` to `v`.



2. `combine_trees_right(list *l)`

- Combines the last two tree nodes in the list into a single tree rooted at a new node.
- The new root's value is derived from the leftmost value of the previous subtree.
- Adjusts the list's `tail` pointer and updates connections.

3. `combine_trees_left(list *l)`

- Combines the first two tree nodes in the list into a single tree rooted at a new node.
- Similar to `combine_trees_right` but operates on the `head` of the list.

4. `absolute_push_left(list *l, int64_t v)`

- Adds a new node with value `v` to the left end of the list.
- Combines adjacent trees of the same `type` recursively using `combine_trees_left`.

5. `absolute_push_right(list *l, int64_t v)`

- Adds a new node with value `v` to the right end of the list.
- Combines adjacent trees of the same `type` recursively using `combine_trees_right`.

6. `push_left(list *l, int64_t v)`

- Adds a value `v` to the left end of the list, respecting the list's orientation.

7. `push_right(list *l, int64_t v)`

- Adds a value `v` to the right end of the list, respecting the list's orientation.

8. `make(int n, int64_t *seq)`

- Creates a list from an array of values (`seq`) of length `n`.
- Adds elements sequentially using `push_right`.

9. `decompose_left(list *l)`

- Decomposes the leftmost composite tree into its two child trees.
- Adjusts the `head` pointer accordingly.

10. `decompose_right(list *l)`

- Decomposes the rightmost composite tree into its two child trees.
 - Adjusts the `tail` pointer accordingly.
-

11. `absolute_pop_left(list *l)`



- Removes and frees the leftmost node or decomposes composite trees recursively until the leftmost node is reached.
 - Updates the list size.
-

12. `absolute_pop_right(list *l)`

- Removes and frees the rightmost node or decomposes composite trees recursively until the rightmost node is reached.
 - Updates the list size.
-



13. `pop_left(list *l)`

- Removes a value from the left end of the list, respecting the list's orientation.
-

14. `pop_right(list *l)`

- Removes a value from the right end of the list, respecting the list's orientation.
-



15. `absolute_peek_left(list *l)`

- Returns the leftmost value of the list without modifying it.
-

16. `absolute_peek_right(list *l)`

- Returns the rightmost value of the list without modifying it.
-

17. `peek_left(list *l)`

- Returns the leftmost value, respecting the list's orientation.
-



18. `peek_right(list *l)`

- Returns the rightmost value, respecting the list's orientation.
-

19. `size(list *l)`

- Returns the number of nodes in the list.
-



20. `empty(list *l)`

- Checks if the list is empty.
-

21. `normal_get(list *l, int i)`

- Retrieves the value at index `i` by traversing the list from the left.
- Uses binary tree traversal for composite trees.



22. `reverse_get(list *l, int i)`

- Retrieves the value at index `i` by traversing the list from the right.
-

23. `get(list *l, int i)`

- Retrieves a value at index `i`, respecting the list's orientation.



24. `normal_set(list *l, int i, int64_t v)`

- Updates the value at index `i` to `v` and ensures tree metadata is consistent (e.g., left/right most values).
-

25. `reverse_set(list *l, int i, int64_t v)`

- Similar to `normal_set`, but traverses from the right.
-



26. `set(list *l, int i, int64_t v)`

- Updates a value at index `i`, respecting the list's orientation.
-

27. `reverse(list *l)`

- Toggles the list's orientation between left-to-right (`0`) and right-to-left (`1`).
-

This implementation maintains a dynamic structure that balances binary trees efficiently, enabling fast updates, retrievals, and rebalancing operations.

⟳ ⟲ ⟳ ⟷ ⟸ ⟹

Message ChatGPT



ChatGPT can make mistakes. Check important info.

